

Part 2 - 分析

為什麼新的V8引擎這麼快？

Google研發的新V8 JavaScript引擎作業速度非常快。我們請熟悉內部程式語言實作的作者依已公開的原始碼來看看V8是如何加速的。

作者 Community Engine公司研發部研發工程師Hajime Morita

裝在美國Google公司的新網頁瀏覽器Chrome中的V8 JavaScript引擎，由於性能良好吸引了相當的注目。它是Google本身特別為了讓Chrome可以高速執行網頁應用而開發的。

Chrome利用美國蘋果電腦領導的WebKit研發計畫作為轉譯引擎* (rendering engine)。WebKit也被用在蘋果電腦正在研發中的Safari瀏覽器中。WebKit的標準配備有稱為JavaScriptCore的JavaScript引擎，但Chrome則以

V8取代此配備(圖1)。

V8開發小組是一群程式語言專家。核心工程師Lars Bak之前研發了HotSpot，這是用在美國Sun Microsystems公司開發的Java虛擬機器(VM)之加速技術。他也在美國的Animorphic Systems公司(於1997年被Sun Microsystems所併購)研發了稱為Strongtalk的實驗Smalltalk*系統。V8善用了研發HotSpot和Strongtalk等等時，所獲得的知識。

快速引擎的需求

Google研發小組在2006年開始研發V8，部分的原因是Google對既有JavaScript引擎的執行速度不滿意。我認為當時JavaScript引擎很慢是有兩個原因的：它們研發的歷史背景，以及JavaScript語言規範的複雜性。

JavaScript存在至少10年了。在1995年，它出現在美國Netscape Communications公司所研發的網頁瀏覽器Netscape Navigator 2.0中。然而有段時間人們對於性能的要求不高，因為它只用在少數的動畫、互動操作或網頁上類似的動作上。網頁瀏覽器的顯示速度視網路下載影像和超文字語言(HTML)以及轉譯引擎(rendering engine)詮釋HTML、串接樣式表(cascading style sheets, CSS)及其他代碼的速度而定。瀏覽器之強化工作把優先順序放在提升轉譯引擎的速

* **Rendering engine**轉譯引擎：讀取和詮釋HTML及其他代碼之網頁瀏覽器，並顯示對應的網頁。

* **Smalltalk**：針對以物件為導向之程式而設計的程式語言。

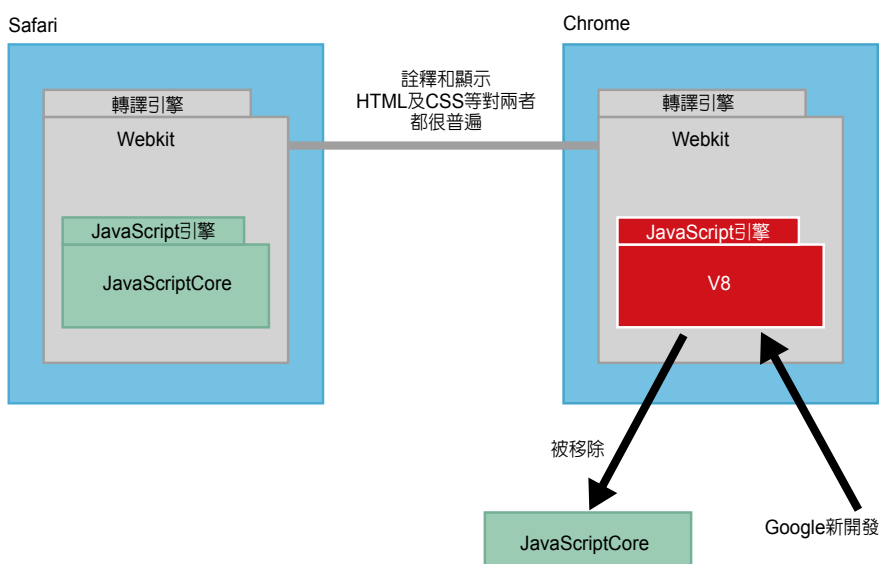


圖1 開發他們自己的JavaScript引擎 蘋果電腦的Safari網頁瀏覽器和Google的Chrome使用相同的轉譯引擎。配有JavaScriptCore的WebKit轉譯引擎在JavaScript引擎中是標準設備，但在Chrome卻被V8取代了。

度，而JavaScript的處理速度不是太重要。同時出現的Java有相當非的差異，它被做得愈來愈快，以便和C++競爭。

然而，在過去幾年，JavaScript突然受到廣泛使用。原因是之前被當成桌上型應用的軟體（其中包括辦公室套件等），現已成為可以在瀏覽器中執行的軟體。Google本身就推出了好幾種JavaScript網路應用，其中包括它的Gmail電子郵件服務、Google Maps地圖數據服務、以及Google Docs辦公室套件。

這些應用的表面速度不僅受到伺服器、網路、轉譯引擎以及其他因素強烈的影響，同時也受到JavaScript本身執行速度的影響。然而既有的JavaScript引擎無法滿足新的需求，而不良的性能是網路應用開發商最關心的。

語言本身的問題

JavaScript語言的規範現在特別強調性能。例如，這在當它判定變數類型時就相當顯而易見。

如C++和Java等主流語言採用靜態分類法。在此方法中，當代碼編譯時，就可宣告變數類型。由於不需要在執行期間檢查類型，因此靜態分類佔有性能上的優勢。

在例如C++和Java等一般處理系統中，fields*和methods*等的內容是以陣列儲存，以1:1位

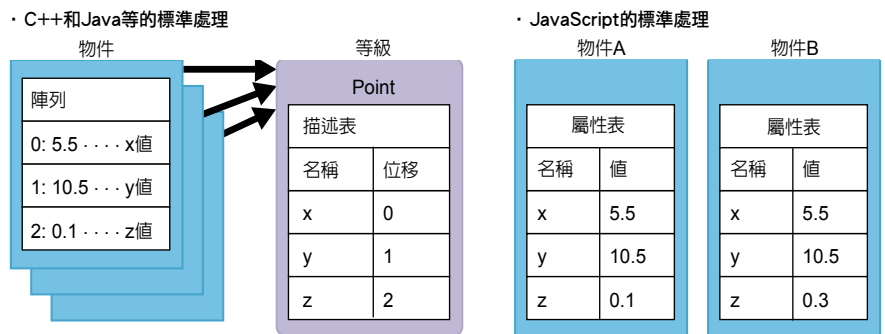


圖2 JavaScript和C++、Java等的不同 C++、Java及其他處理系統將fields和methods等，以它們的名稱以1:1對應陣列內的位移值儲存在陣列中。會事先知道要存取的變數類型（等級），因此可以只用陣列和位移就可以存取fields和methods等。然而在JavaScript，個別的物件都有自己屬性和方法等的表格。每一次程式存取屬性或是呼叫方法時，都必須檢查物件的等級並執行適當的處理。

移（offset）對應fields和methods等的名稱（圖2）。個別變數和methods等儲存的位置，是針對各個等級定義。在C++和Java等，已事先知道所存取的變數（等級）類型，所以語言詮釋系統只要利用陣列和位移來存取field和method等。位移使它只要幾個機器語言指令，就可以存取field、找出field或執行其他任務。

此外，JavaScript則是利用動態分類法。JavaScript變數沒有類型，而所指訂定物件的類型在第一次執行時（換言之，動態地）就已判定了。每次在JavaScript中存取屬性*，或是尋求方法等，必須檢查物件的類別，並照著進行處理。

許多JavaScript引擎都使用哈希表*（hash table）來存取屬性和尋找方法等。換言之，每次存取屬性或是尋找方法時，就會使用特性串作為尋找物件哈希表的關鍵（圖3）。

搜尋哈希表是一個連續動作，包含從雜湊值中判定陣列內位置，然後查看該位置的關鍵（key）是否符合等。和可以使用位移直接讀取數據的陣列比較起來，利用此方法存取較費時。

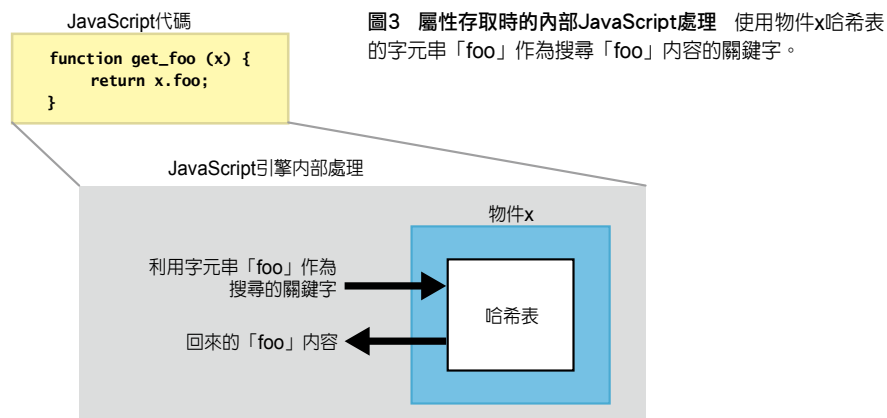
使用動態分類的其他程式語言，有Smalltalk和Ruby等。這些語言基本上也是搜尋哈希表，但它們利用等級來縮短搜尋所需的時間。然而，JavaScript沒有等級之分。除了「Numbers」指示數字值、「Strings」為特性串以及一些其他的之外，其他物件都是

* Field：屬物件的變數。C++中稱為成員變數。

* Method：屬物件的處理類型。C++中稱為成員函式。

* Property屬性：JavaScript屬性是物件自己擁有的變數。在JavaScript中，屬性中不只可以是標準的值，也可以是methods。

* Hash table哈希表：一種數據結構會傳回與特定關鍵相關之對應值。它有一個內部陣列，使用關鍵所產生之Hash值作為陣列中特定位置清單值的位移。如果剛好在相同的位置上產生不同關鍵之Hash值時，清單位置會儲存多個值，這意味著在傳回任何值之前必須先檢查Hash值是否符合。



「Object」型。程式員無法宣告類別（等級），因此無法使用清楚的等級來加速處理。

JavaScript有彈性可以在任何時間，在物件上新增或是刪除屬性和方法等（請參閱第32頁，《熟悉等級之分的程式員之考量》）。JavaScript語言規範非常動態，而業界的一般看法是動態語言比C++或Java等靜態語言更難加速。儘管有困難，但V8利用好幾項技術來達到加快速度，大綱如下。

1. 適時的編譯：

不用位元組碼產生機器語言

從性能的觀點來看，V8具有4個主要特性。首先，它在執行時以稱為及時（just-in-time, JIT）的編譯方法，來產生機器語言。這是個普遍用來改善詮釋速度的方法，在Java和.NET等語言中也可以發現此方法。V8比美國Mozilla Foundation開發的Firefox瀏覽器中的SpiderMonkey JavaScript引擎，或Safari的JavaScriptCore等競

爭引擎還要早實踐了此技術。

V8 JIT編譯器在產生機器語言時，不會產生中間碼（圖4）。例如，在Java編譯器和先將原始碼轉換成一個以虛擬中間語言（稱為位元組碼，bytecode）表示的等級檔。Java編譯器和位元組碼編譯器產生位元組碼，而非機器語言。Java VM按順序地在執行中詮釋等級檔案位元組碼。此執行模式稱為位元組碼翻譯器。Firefox的SpiderMonkey具有一個內部的位元組碼編譯器和位元組翻譯器，將JavaScript原始碼轉換成它自家特色的位元組代碼，以便執行。

事實上，Java VM目前使用一個以HotSpot為基礎的JIT編譯器。它扮演位元組碼編譯器的角色，來詮釋代碼，將常執行的代碼區塊轉換成機器語言然後執行：混合模式。

位元組碼編譯器和混合模式等，具有製作簡單且有絕佳可攜性的優點。只要是引擎可以編譯

的原始碼，那麼就可以在任何中央處理器（CPU）架構上執行位元組碼，這正是為什麼該技術被稱為「虛擬機器」的原因。即使在產生機器代碼的混合模式中，可以藉由寫入位元組碼譯器開始進行研發，然後實作機器語言產生器。藉由使用簡單的位元碼，在機器代碼產生時，要將輸出最佳化就變得容易許多。

V8不是將原始程式轉換成中間語言，而是將JavaScript伺服器所產生的精簡語法，直接產生機器語言並加以執行。沒有虛擬機器，且因為不需要中間表示式，程式處理更快就開始了。然而，另一方面，它也喪失了虛擬機器的好處，例如透過位元組碼編譯器和混合模式等，所帶來的高可攜性和簡易最佳化等。

2. 記憶體回收管理：

與Java同等的複雜執行

第二個關鍵的特性是，V8將記憶體回收管理（garbage collection, GC*）實作為「精確的GC*」。相反的，大部分的JavaScript引擎、Ruby及其他語言編譯器都是使用保守的GC*，因為保守的GC實

* Garbage collection (GC) 記憶體回收管理：自動偵測被程式保留但已不再使用的記憶體空間並釋放。

* 保守GC：沒有分別嚴格管理指標和數字值之記憶體回收管理。此方法是如果它可以成為指標，那就以指標來看待它，即使它可能個數值。此方法防止物件被意外回收，但它也無法釋出可能的記憶體。

作簡單許多。雖然精確的GC在實作上更為複雜，但也有性能上的優點。Sun Microsystems的Java VM就是使用精確GC。

雖然精確GC本身就是高效率的，但以精確GC為基礎的高階實作，如世代GC、複製GC以

及標記和精簡處理（mark-and-compact processing）等在性能上有明顯的改善。世代GC藉由分開管理「年青世代」物件（經常收集）和「舊世代」物件（相對長壽的物件）而提升了GC效率。

V8使用了世代GC，在新世代

處理上使用輕載複製GC，而在舊GC上使用標記和精簡GC，因為它須在記憶體空間內移動物件。這很難在保守GC中執行。

在物件的複製中，壓縮（compaction）（在硬碟方面稱為defrag）和類似動作時，物件



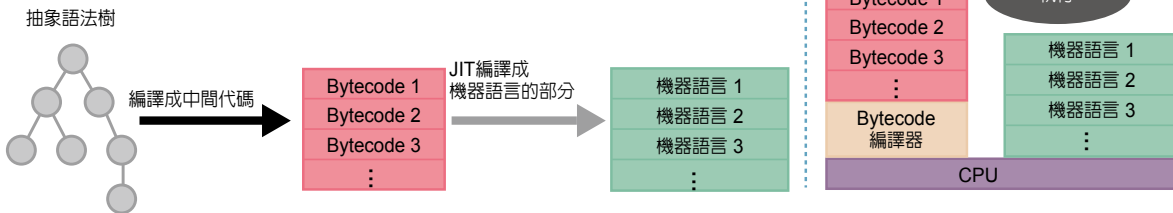
圖4 V8的JIT編譯器直接輸出機器語言 程式語言系統先使用語法分析器將原始碼轉換成抽象語法樹（abstract syntax tree）。之前有幾種方式來處理。位元組碼編譯器將抽象語法樹編輯中間代碼，然後在編譯器中執行。如Java JIT等混合模式將這中間代碼的一部分編輯成機器語言，以改善處理性能。Chrome不使用中間代碼，JIT直接從抽象語法樹來編譯機器語言。也有抽象語法樹編譯器，直接編譯抽象語法樹。

步驟2

· Bytecode編譯器（其他JavaScript引擎等）



· 混合模式（Java）



· 直接生成機器語言（Chrome）



的位址會改變，且基於這個原因，最普遍的方法是用「句柄」(handles)間接地參考位址。然而，V8不使用handles，而是重寫該物件的所有參考資料。不使用handles會造成實作更困難，但卻能改善性能因為少了間接參考。Java VM HotSpot也使用相同的技術。

3. 內嵌緩存：

JavaScript中不可用？

V8目前可以針對x86和ARM架構產生適合的機器語言。雖然沒使用C++或Java的最佳化，V8還是有動態語言與生俱來的速度。

其中一項良好範例是內嵌緩存 (inline cache)，這是一項技

JavaScript代碼

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}
```

```
var p = new Point(1,2);
var q = new Point(3,4);
```

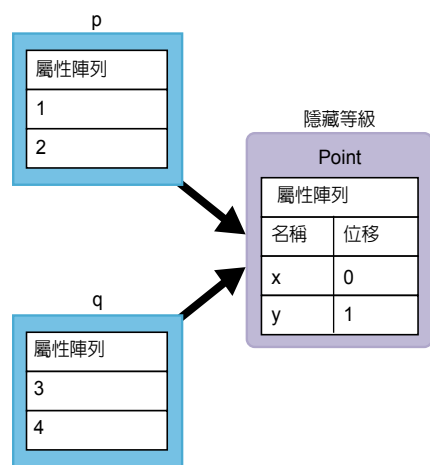


圖5 V8物件有隱藏等級的參考資料。如果物件的描述是相同的，那麼隱藏等級也會相同。在此範例中，物件p和q都屬於相同的隱藏等級。

巧可以免除方法呼叫、屬性存取、哈希表搜尋等。它可以立即緩存之前的搜尋結果，因此稱為「內嵌」。人們知道此技術已有一段時間了，使用在Smalltalk、Java和Ruby等。

內嵌緩存假設物件都有等級之分，但在JavaScript語言規範中它們卻沒有。直到V8出現後，而這這就是為什麼JavaScript引擎都沒有內嵌緩存的原因。

為了突破此限制，V8在執行時就分析程式作業，並利用「隱藏等級」(hidden classes)為物件指定暫時的等級。有了隱藏等級，即使是JavaScript也可以使用內嵌緩存。但是這些等級是提升執行速度之技巧，不是語言規範的延伸。它們無法成為JavaScript程式的等級參考。

4. 隱藏等級：

儲存等級變動資訊

隱藏等級為特別表示沒有等級之分的JavaScript語言規範帶來有趣的挑戰，且或許是V8用來提升速度最獨特的技巧。它們值得更深入的探究。

在V8中建立等級有兩個主要的原因，即(1)將屬性名稱相同的物件歸類，及(2)識別屬性名稱不同的物件。前一類中的物件有完全相同的物件描述，而這可以加速屬性存取。

在V8，符合歸類條件的等級會

JavaScript代碼

```
p.z = 5;
```

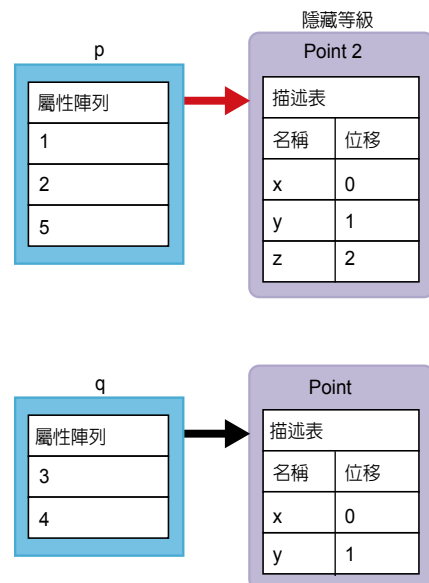


圖6 配置新等級：等級轉換。屬性改變的物件會被歸為新等級。當物件p增加了新屬性z時，物件p就會被歸為新等級。

配置在各種JavaScript物件上。物件參考所配置的等級(圖5)。然而這些等級只存在於V8作為方便之用，所以它們是「隱藏」的。

我上面提到隨時可以在JavaScript中新增或刪除屬性。然而當此事發生時會毀壞歸類條件(歸納名稱相同的屬性)。V8藉由建立屬性變化所需的新等級來解決。屬性改變的物件透過一個稱為「等級轉換」的程序納入新級別中。

第二個目標－識別屬性名稱不同的物件－則是藉由建立新等級來達成。然而，如果每一次屬性改變就建立一個新等級的話，那就無法持續達到第一個目標了(歸納名稱相同的屬性)。

文轉第30頁▶

競爭瀏覽器也強化了 JavaScript引擎

V8透過各式各樣的加速技術達到了比其他網頁瀏覽器更優越的性能。由於Google是在2006年開始研發V8，但如Firefox、Safari及挪威的Opera Software推出的Opera等競爭瀏覽器也一直在改進他們的JavaScript引擎。

尤其是去年，Firefox的SpiderMonkey、Safari的JavaScriptCode引擎等等在性能上都有明顯的改進。還在實驗室中的最新瀏覽器裡的JavaScript引擎據說性能完全和V8的相等。

在2006年，Adobe Systems公司將Tamarin（Flash ActionScript引擎）捐贈給Firefox的研發廠商Mozilla Foundation。SpiderMonkey計劃裝置一個新的JIT編譯器（以Adobe Systems和Mozilla Foundation共同研發的「追蹤樹」演算法為基礎）。

此高速計畫稱為TraceMonkey。TraceMonkey利用部分的內嵌擴大功能來提升速度，並且預計會使用在預定2008年底推出的Firefox 3.1中。

與V8同等的技術

目前在Safari中實作的JavaScriptCore是個簡易的抽象語法樹編譯器。這類的編譯器只從頭開始作代碼追蹤，一般都比位元組編譯器慢。Google研發的V8被認為是緩慢JavaScriptCore JavaScript引擎部分的原因是，由Chrome所使用的轉譯引擎WebKit所提供的標準配備。

為了解決此情形，WebKit研發小組推出了SquirrelFish－把位元組碼編譯器放入JavaScriptCore的計畫。具有SquirrelFish成果的新

JavaScriptCore預定會出現在Safari 4.0。

在2008年9月18日，就在Google公佈Chrome後不到一個月，同一研發小組發表了SquirrelFish Extreme－將JIT編譯器放入SquirrelFish的計畫。它們顯然是還擊對手之作。SquirrelFish Extreme實作StructureID（與V8的隱藏等級同等）、內嵌緩存等等。

JIT編譯器是進一步邁向更快的位元組碼編譯，因此SquirrelFish也計劃與它搭配。然而在Chrome發表之後，開發藍圖有可能加速。

雖然目前尚未透露何種版本的Safari會提供SquirrelFish Extreme，但現在有配有此引擎的測試版WebKit可供下載（註A-1）。與此同時，也宣佈了SquirrelFish Extreme及基準測試結果（註A-2）。

Firefox和Safari加速計畫的提出原因之一是Internet Explorer中的JavaScript引擎，現在的性能是最差的。考慮它擁有的龐大市占率，除非Internet Explorer加速它自己的JavaScript執行，否則要研發真的利用JavaScript功能的網頁應用，是很困難的。許多網路應用研發商把要在Internet Explorer執行較快的JavaScript，視為一項重大的瓶頸。

註A-1：具SquirrelFish Extreme之測試版WebKit的下載網站：<http://nightly.webkit.org/>

註A-2：基準測試結果（Charles H Ying）
<http://www.satine.org/archives/2008/09/19/squirrelfish-extreme-fastest-javascript-engine-yet/>

V8將變換資訊儲存在等級內，來解決此問題。考量圖7，它說明了圖6中所示的情形，當隱藏等級Point有x和y屬性時，新屬性x就會新增至Point級的物件p中。

當新屬性z加到物件p時，V8會將「新增屬性p，建立Point2等級」的資訊儲存在Point級的內部表格中（圖7，步驟1）。當新屬性z新增至也是Point級的物件q時，V8會先搜尋Point級的表格，並發現Point2級已加入屬性z。

在表格中找到等級時，物件q就會被設定至該等級（Point2），

而不建立新等級（圖7，步驟2）。這達到了歸納屬性名稱相同的物件之目的。

然而此方法，意味著與隱藏等級對應的空物件會有龐大的轉換表格。V8透過為各個建構器功能建立隱藏等級來處理。如果建構器功能不同，就算物件的陳述完全相同，也會為它建立一個新的隱藏等級。

內嵌緩存

JavaScript引擎和V8不同，它將物件屬性儲存在哈希表中，但V8

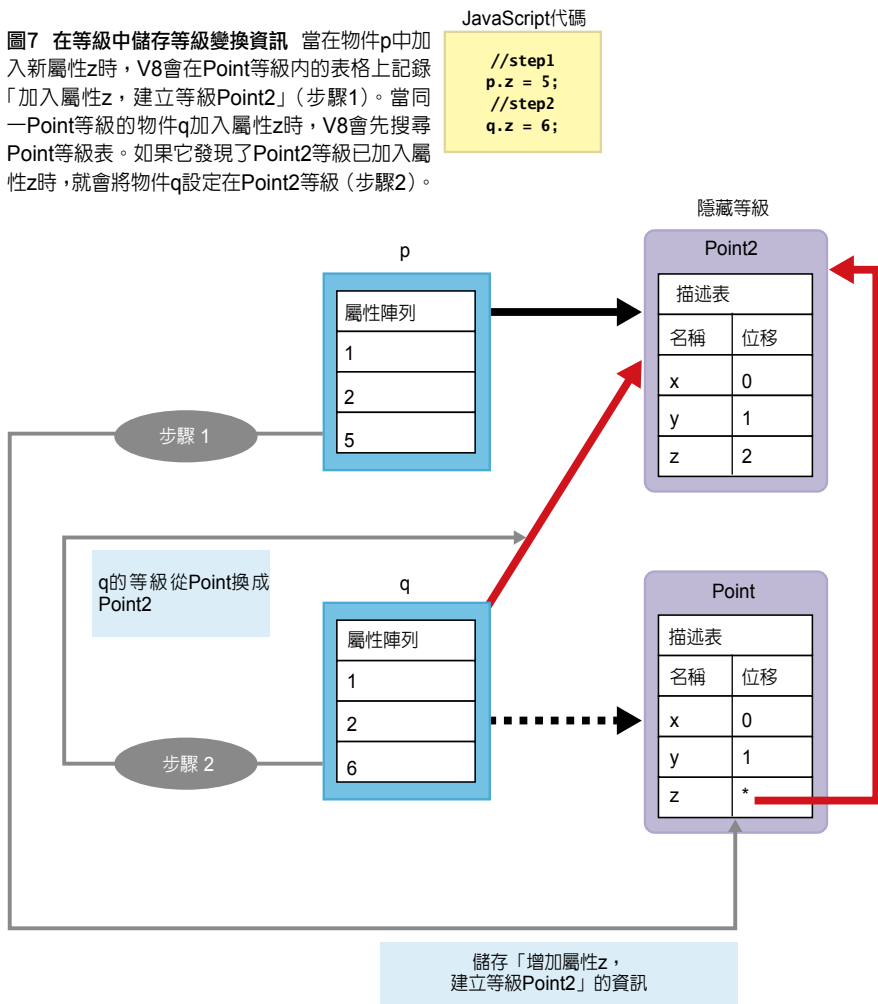
則將它們儲存在陣列中。位移資訊—指定個別屬性在陣列中的位置—是儲存在隱藏等級的哈希表中。同一隱藏等級的物件具有相同的屬性名稱。如果知道物件等級，那麼就可以利用位移依陣列作業存取屬性。這比搜尋哈希表快速許多。

然而，在JavaScript等動態語言中，很難事先知道物件類型。例如，圖8的原始碼為物件類型p和q呼叫lengthSquared()函數。物件類型p和q的屬性不同，隱藏等級也不同。因此無法判定lengthSquared()函數代碼的引數（arguments）類型。

若要讀取函數中的物件屬性，必須先檢查物件的隱藏等級，以及所搜尋的等級哈希表，以找出該屬性的位移。然後利用位移存取陣列。儘管是在陣列中存取屬性，需先搜尋哈希表的需求就毀掉了使用陣列的優點。

然而，從不同的觀點來看，情況有所不同。在實際的程式中，要求代碼執行的等級並不多。例如，在圖8的lengthSquared()函數甚至假設大部分通過成為引數的值，都是Point等級物件，而一般而言這是正確的。

內嵌緩存是一項加速技術，此設計是為了利用程式中局部（local）類別的方法。若要程式性的屬性存取，V8會產生一個指令串來搜尋隱藏的等級陳列表（圖



9)。此代碼稱為premonomorphic stub。此stub是為了在中函數存取屬性（圖10）。premonomorphic stub擁有兩個資訊：搜尋用的隱藏等級，以及取自隱藏的位移。因而產生新代碼以緩存此資訊（圖11）。

在搜尋表格之前，具屬性的物件之隱藏等級會與緩存隱藏等級比較。如果相符就不需要再搜尋，且可以使用快取位移來存取屬性。如果隱藏等級不相符，就透過隱藏等級哈希表以一般方式判斷位移快取。

新產生的代碼被稱為monomorphic stub。「內嵌」這個字的意思是查詢隱藏等級所需的位移，是以立即可用的形式嵌入在所產生的代碼中。當第一次叫出monomorphic stub時，它會將功能從pre-monomorphic stub位址中所叫出的第一個位址重寫成monomorphic stub位址（圖12）。自該點之後，使用高速monomorphic stub，單靠等級比較和陣列存取就可以處理屬性存取。

如果只有一個物件等級具屬性，monomorphic stub的效率就會很高。然而，如果等級類型愈多，緩存失誤就會更頻繁，進而降低monomorphic stub的效率。

當緩存失誤時，V8藉由產生另一個稱為megamorphic stub的代碼來解決（圖13）。與個別等

圖8 代碼樣本：JavaScript 無法判斷函數引數類型 在執行之前根本無法判斷引數是Point型或是lengthSquared()函數的LabeledLocation型。

```
function lengthSquared(p) {
    return p.x * p.x + p.y * p.y;
}

function LabeledLocation(name, x, y) {
    this.name = name;
    this.x = x;
    this.y = y;
}

var p = new Point(10, 20);
var q = new LabeledLocation("hello", 10, 20);

var plen = lengthSquared(p);
var qlen = lengthSquared(q);
```

圖9 在虛擬碼 (pseudocode) 中的 premonomorphic stub 從隱藏等級中取得屬性位移。

```
Object* find_x_for_p_preomorphic(Object* p) {
    Class* klass = p->get_class();
    int offset = klass->lookup_offset("x");
    update_cache(klass, offset);
    return p->properties[offset];
}
```

圖10 premonomorphic stub 呼叫 存取功能中的屬性時會呼叫 premonomorphic stub。

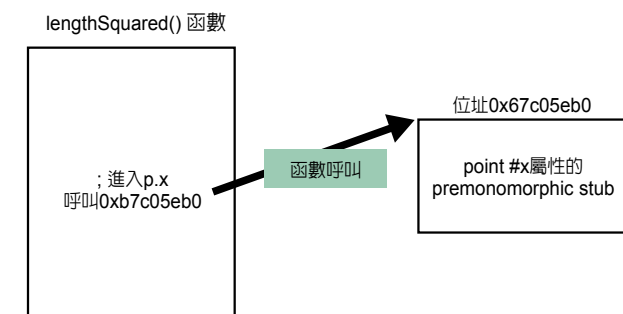


圖11 虛擬碼的 monomorphic stub 處理 直接嵌入代碼中的位移是用來存取屬性的常數。

```
Object* find_x_for_p_monomorphic(Object* p) {
    if (CACHED_KLASS == p->get_class()) {
        return p->properties[CACHED_OFFSET];
    } else {
        return lookup_property_on_monomorphic(p, "x");
    }
}
```

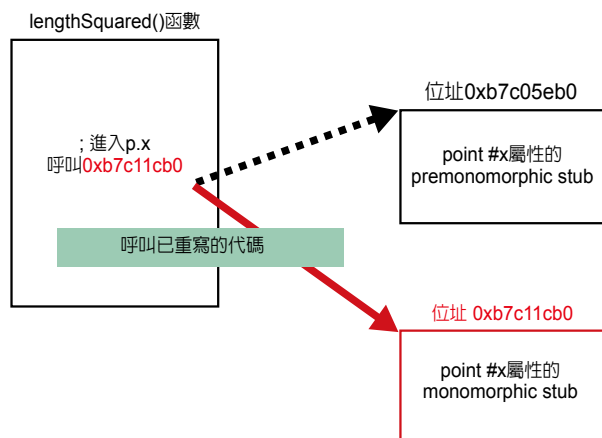


圖12 monomorphic stub 呼叫 當呼叫monomorphic stub時，它會將功能從premonomorphic stub位址中叫出的第一個位址，重寫成monomorphic stub位址。

```
Object* find_x_for_p_megamorphic(Object* p) {
  Class* klass = p->get_class();
  // 內嵌處理實際的搜尋
  Stub* stub = klass->lookup_cached_stub("x")
  if (NULL != stub) {
    return (*stub)(p);
  } else {
    return lookup_property_on_megamorphic(p, "x");
  }
}
```

圖13 虛擬碼中的Megamorphic stub處理 與等級對應的monomorphic stub事先儲存在哈希表中，並在執行時被搜尋和叫出。如果無法找到對應的monomorphic stub，就會在等級哈希表中搜尋位移。

熟悉等級之分的程式員 之考量

JavaScript沒有等級，但為了讓熟悉使用等級（以物件為導向的代碼）之程式員更方便使用，可以使用「new」的運算元來建立物件，就像在Java一樣。在「new」運算元之後會定義一個特別的「constructor」建構器函數（圖B-1 a, b）。

然而，即使沒有建構器函數，也可以建立物件（圖B-1 c）和設定屬性的（圖B-1 d）。JavaScript物件的屬性和方法等隨時都可以新增或刪除。除了用點標記（dot notation）存取JavaScript屬性以外，也可以使用括號，建議雜湊存取（圖B-1 e、f）或是以變數特定屬性名稱字元串（圖B-1 g）。從這些範例中明確顯示JavaScript物件的設計是為了使用哈希表。

```
a) 定義建構函數「Point」
function Point(x, y) {
  // this 是指它自己
  this.x = x;
  this.y = y;
}

b) 當增加新的及呼叫建構器函數時所建立的物件
var p = new Point(10, 20);

c) 沒有建構器函數也可以建立物件
var p = { x: 10, y: 20 };

d) 可以自由地在物件上新增屬性
p.z = 30;

e) 使用點標記存取屬性
var y = p.y

f) 使用括號之雜湊存取
var y = p["y"];

g) 也可以使用變數進行雜湊存取
var name = "y";
var p[name];
```

圖B-1 JavaScript代碼範例

級對應的monomorphic stub都寫在哈希表中，其在執行時搜尋和叫出stub。如果沒有等級對應的monomorphic stub時，就會從等級哈希表中搜尋位移。

當monomorphic stub發生緩存失誤時，monomorphic stub會將功能從monomorphic stub位址叫出的第一個位址以megamorphic stub位址重寫。在代碼搜尋方面，megamorphic stub的性能比monomorphic stub低，但是megamorphic代碼卻比使用緩存升級、代碼生成及其他輔助處理的premonomorphic stubs快許多。

涵蓋多種等級的內嵌緩存稱為多型態內嵌緩存。V8內嵌緩存系統被用來呼叫方法以及存取屬性。

機器語言的特性

如以上所述，X8是為了例如利用內嵌緩存等，來達到動態語言中天生的速度而設計的。創作使用於內嵌緩存之stub的機器語言生成模組密切地與JIT編譯器連結。一些經常使用的方法也被創作成機器語言以達到與內嵌擴大相同的效果，使它們成為「內在」的。V8原始碼列出了內在轉換的候選名單。

V8所含的shell程式可以用來檢查V8所產生的機器語言（請參閱第34頁《V8原始碼之建立和執行》）。所產生的指令串可以和V8代碼比較，以便顯出它的特性。

a) JavaScript代碼

```
function plus_one(n) {
  var one = 1;
  return n + one;
}
```



由V8編輯

b) 產生x86機器語言

```
0  push ebp                ;; debug: statement 18
1  mov ebp,esp
3  push esi
4  push edi
5  mov eax,0xb7c00135      ;; object: 0xb7c00135 <undefined>
11 push eax
12 mp esp,[0x820baa8]     ;; external reference (StackGuard::
                           address_of_limit())
18 jc 50 (0xb7c11cce)
24 push 0x2               ;; debug: statement 26
26 mov eax,[esp]          ;; debug: statement 34
29 mov [ebp+0xf4],eax
32 pop eax
33 push [ebp+0x8]         ;; debug: statement 41
36 push [ebp+0xf4]
39 call 0xb7c06c64        ;; code target (STUB,
                           GenericBinaryOp, minor: 144)
44 mov esp,ebp           ;; js return
46 pop ebp
47 ret 0x8
```

圖14 V8從JavaScript代碼產生的機器語言 加法處理被轉換成函數呼叫的機器語言 (a、b)。

例如，在執行圖14a所示的JavaScript函數時，就會產生一個如圖14b所示的x86機器語言指令串。此函數在第39個指令中被呼叫，是個「n+one」加法。在JavaScript中，「+」運算元指示數字變數的加法，以及字元串的連續性。編譯器不是產生代碼來判決這是哪一種，而是呼叫函數來負責判斷。

如果圖14的函數稍做更改（圖15），那圖14b的函數呼叫就會消失，但會有個加法指令（第20），及分支指令（JNZ的若不是零就跳出，第31）。

當使用整數作為「+」運算元的運算元，V8編譯器在不呼叫函數下會產生一個有「加法」指令的指令串。如果發現運算元（在此為「n」）成了Number物件或

a) JavaScript代碼

```
function plus_one(n) {
  return n + 1;
}
```



由V8編輯

b) 產生x86機器語言

```
0  push ebp                ;; debug: statement 78
1  mov ebp,esp
3  push esi
4  push edi
5  cmp esp,[0x820baa8]     ;; external reference (StackGuard::
                           address_of_limit())
11 jc 56 (0xb7c11ce8)
17 mov eax,[ebp+0x8]      ;; debug: statement 86
20 add eax,0x2
23 jo 43 (0xb7c11cdb)
29 test al,0x1
31 jnz 43 (0xb7c11cdb)
37 mov esp,ebp           ;; js return
39 pop ebp
40 ret 0x8
```

圖15 V8從圖14之JavaScript中所產生的機器語言，經小幅修改

a) C代碼

```
int add_one(int n) {
  int one = 1;
  return n + one;
}
```



以C編譯器編譯

b) 產生x86機器語言

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %eax
addl    $1, %eax
popl    %ebp
ret
```

圖16 C編譯器從C代碼所產生的機器語言 所產生的機器語言比V8所產生的乾淨許多 (a、b)，大部分是因為C和JavaScript語言規範的差異所致。

String物件等的指標 (pointer)，就會叫出函數。「加法」只會發生在當兩個「+」運算的運算元都是整數時。在這種情況下，因為可以省略函數呼叫因此執行就會比較快。

此外，0x2會加上「加法」指令，因為為最低有效位元 (least significant bit, LSB) 被用來區別整數 (0) 和指標 (1)。加0x2 (二進位中的十) 就如同在該值加上1，LSB除外。在jo指令的溢位

(overflow) 處理中，利用測試和jnz指令來判定指標，跳到下游處理 (註1)。

這類的竅門在編譯器中到處都有。然而，產生器代碼也透露了編譯器的限制。具傳統最佳化的編譯器可以針對圖14和15產生完全一樣的機器語言，這是由於常數進位的關係。然而V8編譯器是在抽象語法樹* (abstract syntax tree) 單元中產生代碼，因此在處理延伸多個節點時就沒有最佳

化。這在大量的push和pop指令也非常明顯。

圖16顯示了C裡相同的處理提供參考。由於C和JavaScript之間的語言規範不同，因此所產生的機器語言是圖14和圖15的不同，這和編譯器的性能無關。■

註1：當溢位信號出現時，jo指令會跳至特定的位址。測試指令將邏輯AND結果反映成零和符號指標等。除非零信號出現，否則jnz指令會跳至特定的位址。

* Abstract syntax tree抽象語法樹：在樹狀架構中代表程式架構的數據。

V8原始碼之建立和執行

V8的原始碼已做成開放原始碼產品，而且可以從計畫的網站上下載。可以從Subversion檔案庫 (repository) 查看原始碼的最新版本，上面也有解釋如何執行樣品shell程式。

先在Subversion檔案庫查出代碼，然後取得原始碼。

```
$ svn checkout http://v8.
googlecode.com/svn/
branches/bleeding_edge/ v8
```

在Linux下，此建立使用scons指令 (需gcc和g++)。可以使用yum和aptitude等安裝命令，進

行scons安裝。從Subversion查到的原始碼包括了Visual Studio 2005的計畫檔案。Windows 建立使用此計畫檔案。

在Linux下，scons如下圖所示執行，以建立樣本shell程式。

```
$ cd v8
$ scons sample=shell
```

有了成功的建立，「shell」程式會輸出至同一目錄。將檔案名稱特定為引數讓「shell」執行。

```
• hello.js content
print("hello, V8!");
• run
$ ./shell hello.js
hello, V8!
```

重建成除錯版本以輸出編譯器機器語言。

```
$ scons sample=shell
mode=debug
```

除錯問題是shell_g。在執行時將機器語言輸出特定為選項以輸出機器語言 (圖C-1)。

```
$ ./shell_g --print-code hello.js
kind = FUNCTION
Instructions (size = 65)
0xb792fef4 0 55      push ebp
0xb792fef5 1 8bec     mov ebp,esp
0xb792fef7 3 56      push esi
0xb792fef8 4 57      push edi
0xb792fef9 5 c7c03521c9b7  moveax,0xb7c92135
....
hello, V8!
```

圖C-1 V8之機器語言輸出