如何編寫有利於編譯器優化的代碼

■作者：IAR Systems

在嵌入式開發中，代碼的體積和運行效率非常重要，代碼體積往往和晶片的 FLASH、RAM 容量對應，程式的運行效率也要求在相應能力的處理器上運行。在大多數情況下，成熟的開發人員都希望降低代碼體積、提高代碼運行效率，然而具體該怎麼做呢？本篇文章將編譯器廠商 IAR Systems 的編譯器為例，來解答開發人員在實際工作中常常遇到的問題，工程師朋友們可以在 IAR 編譯器上進行實踐驗證。

對於嵌入式系統，最終代碼的體積和效率取決於由編譯器生成的可執行代碼，而非開發人員編寫的原始程式碼；但是原始程式碼的優化，可以幫助編譯器生成更加優質的可執行代碼。因此，開發人員不僅要從整體效率等因素上去構思原始程式碼體系，也要高度關注編譯器的性能和編譯優化的便捷性。

有優化功能的編譯器可生成既小又快的可執行代碼，編譯器是通過對原始程式碼的重複轉換來實現優化。通常，編譯器優化會遵循完善的數學或邏輯理論基礎。但是某些編譯優化則是通過啓發式的方法，經驗表明，一些代碼轉換往往會產生更好的代碼，或者開拓出進一步編譯優化的空間。

編譯優化只有少數情況依賴于編譯器的黑科技，大多數時候編寫原始程式碼的方式決定了程式是否可以被編譯器優化。在某些情況下，即使對原始程式碼做微小改動也會對編譯器生成的代碼效率產生重大影響。

本文將講述在編寫代碼時需要注意的事項，但我們首先應明確一點，我們沒有必要儘量減少代碼量，因為即使在一個運算式中使用 ?:- 運算式、後增量和逗號運算式來消除副作用，也不會使編譯器產生更有效的代碼。這只會使你的原始程式碼變得晦澀難懂，難以維護。例如在一個複雜的運算式中間加入一個後增量或賦值，則在讀代碼的時候很容易被忽略。請儘量用一種易於閱讀的風格來編寫代碼。

迴圈

下面看似簡單的迴圈會報錯嗎？

```
for (i = 0; i != n; ++i)
{
a[i] = b[i];
}
```

雖然不會報錯，但其中有幾點會影響到編譯器生成的代碼效率。

例如，索引變數的類型應與指標相匹配。

a[i] 這樣的陣列運算式實際上是 *(&a[0]+i*sizeof(a[0])，或者通俗地說：將第 i 個元素的偏移量加到 a 的第一個元素的指標上。對於指標運算， 索引運算式的類型最好與指標所指向的類型一致 (__far 指標除外，因為其指標所指向的類型和索引運算式的類型不同)。如果索引運算式的類型與指標所指向的類型不匹配，那麼在把它與指標相加之前，必須將它強制轉換為正確的類型。

如果在應用中，堆疊空間資源 (堆疊一般放在 RAM 中) 比代碼尺寸資源 (代碼一般放在 ROM 或者 Flash 中) 更寶貴，則可以為索引變數選擇一個更小的類型來減少堆疊空間的使用，但這往往會犧牲代

碼尺寸和執行時間 (代碼尺寸變大，執行時間變慢)。不僅如此，這種轉換也會妨礙迴圈代碼的優化。

除上述問題外，我們也要關注迴圈條件，因為只有在進入迴圈之前可以計算出反覆運算次數的情況下，才可以進行迴圈優化。然而，這項計算工作非常複雜，並非用最終值減去初始值並除以增量那麼簡單。例如，如果 i 是一個無符號字元，n 是一個整數，而 n 的值是 1000，那麼會發生什麼情況？答案是變數 i 在達到 1000 之前就會溢出。

雖然程式師肯定不想要一個無限迴圈，重複地將 256 個元素從 b 複製到 a，但是編譯器無法瞭解程式師的意圖。它必須假設最壞的情況，並且不能應用需要在進入迴圈之前提供行程數的優化。此外，如果最終值是一個變數，您還應該避免在迴圈條件中使用關係運算子 <= 和 >=。如果迴圈條件是 i <= n，那麼 n 有可能是該類型中可表示的最高值，因此編譯器必須假定這是一個潛在的無限迴圈。

別名

通常，我們不建議使用全域變數。這是因為您可在程式的任何地方修改全域變數，並且程式會因全域變數的值而變化。這就會形成複雜的依賴關係，使人很難理解程式，也很難確定改變全域變數的值會對程式產生怎樣的影響。從優化器的角度來看，這種情況更糟糕，因為通過指標的存儲就可以改變任意全域變數的值。如果能通過多種方式訪問一個變數，這種情況就會被稱為別名，而別名使代碼更難優化。

```
char *buf
void clear_buf()
{
 int i;
 for (i = 0; i < 128; ++i)
 {
 buf[i] = 0;
 }
}
```

儘管程式師知道向 buf 所指向的緩存區進行寫操作不會改變這個 buf 變數本身，但編譯器還是不得不做最壞的打算，在迴圈的每一次反覆運算中從記憶體中重新載入 buf。

如果將緩存區的位址作為參數傳遞，而不是使用全域變數，則可以消除別名：

```
void clear_buf(char *buf)
{
 int i;
 for (i = 0; i < 128; ++i)
 {
 buf[i] = 0;
 }
}
```

使用這個解決方案後，指標 buf 就不會被通過指標的存儲影響。如此一來，指標 buf 在迴圈中就可以保持不變，其值只需在迴圈前載入一次即可，而不是在每次反覆運算時都要重新載入。

然而，如果需要在不共用調用者 / 被調用者關係的程式碼片段之間傳遞資訊，則直接使用全域變數即可。但是，對於計算密集型任務，尤其是涉及指標操作時，最好使用自動變數。

儘量不用後增量和後減量

在下文中，關於後增量的所有內容也適用於後減量。C 語言中關於後增量語義的標準文本指出：“尾碼 ++ 運算子的結果是運算元的值。在得到結果後，運算元的值會遞增”。雖然微控制器普遍擁有可在載入或存儲操作後增加指標的定址模式，但其中只有很少能以同樣的效率處理其他類型的後增量。為符合標準，編譯器必須在執行增量之前將運算元複製到一個臨時變數。對於直線代碼來說，可以從運算式中取出增量，然後放在運算式之後。比如以下運算式：

```
foo = a[i++];
```

可以改為

```
foo = a[i];
```

```
i = i + 1;
```

但如果後增量屬於 while 迴圈中的條件，又會發生什麼？由於在條件後面沒有可以插入增量的地方，因此必須在測試前添加增量。對於這些常見但是又與生成可執行代碼效率密切相關的設計，諸如 IAR Systems 的 Embedded Workbench 這樣的工具都在總結了大量實踐後提供了優化方案。

比如以下迴圈

```
i = 0;
while (a[i++] != 0)
 {
 ...
}
```

應改為

```
loop:
 temp = i; /* 保存運算元的值 */
 i = temp + 1; /* 遞增運算元 */
 if (a[temp] == 0) /* 使用保存的值 */
 goto no_loop;
 ...
 goto loop;
no_loop:
```

或

```
loop:
 temp = a[i]; /* 使用運算元的值  */
 i = i + 1; /* 遞增運算元 */
 if (temp == 0)
 goto no_loop;
 ...
 goto loop;
no_loop:
```

如果迴圈後的 i 的值不相關，最好將增量放在迴圈內。比如以下幾乎相同的迴圈

```
i = 0;
while (a[i] != 0)
{
++i;
...
}
```

可以在沒有臨時變數的情況下執行：

```
loop:
 if (a[i] == 0)
 goto no_loop;
 i = i + 1;
 ...
 goto loop;
 no_loop:
```

優化編譯器的開發者們很清楚後增量會使代碼編寫變得更複雜，儘管我們已盡力去識別這些模式，並儘量消除臨時變數，但總有一些情況使我們無法產生有效代碼，尤其是遇到比上述更複雜的迴圈條件時。通常，我們會將一個複雜的運算式分割成若干個更簡單的運算式，就像上面的迴圈條件被分割成一個測試和一個增量那樣。

在 C++ 環境中，選擇前增量還是後增量的重要性更高。這是因為 operator++ 和 operator-- 都可以以首碼和尾碼的形式重載。將運算子作為類對象重載時，雖然沒必要模仿基本類型運算子的行為，但也應儘量接近。因此，對於那些可以直觀地對物件進行遞增和遞減的類，例如反覆運算器，通常會有首碼 (operator++() 和 operator--()) 和尾碼形式 (operator++(int) 和 operator--(int))。

為了類比基本類型的首碼 ++ 的行為，operator++() 可以修改物件並返回對修改後物件的引用。那麼模擬基本類型的尾碼 ++ 的行為會怎樣？您還記得嗎？“尾碼 ++ 運算子的結果是運算元的值。在得到結果後，運算元的值會遞增”。就像上面的非直線代碼一樣，operator++(int) 的實現者必須複製原始物件，修改原始物件，並按值返回副本。由於存在複製操作，因此 operator++(int) 的開銷要高於 operator++()。

對於基本類型，如果忽略 i++ 的結果，優化器通常可以消除不必要的複製，但優化器不能將對一個重載運算子的調用變為另一個。如果您出於習慣編寫 i++ 而不是 ++i，您就會調用開銷更大的增量運算子。

雖然我們一直在反對使用後增量，但不得不承認，後增量在有些情況下還是有用的。如果確實要給一個變數進行後置增量操作，那就繼續吧。如果

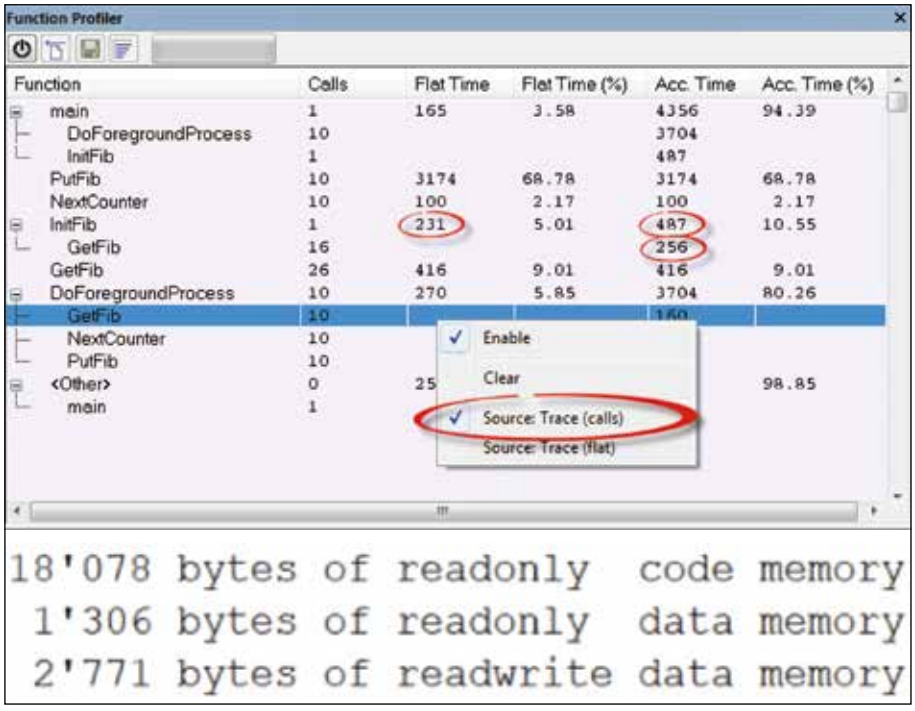
後增量操作和您期望的操作一致，可以使用後增量操作。但請注意，切勿為避免多寫一行代碼來遞增變數，而使用後增量操作。

每當您在迴圈條件、if 條件、switch 運算式、?:- 運算式或函式呼叫參數中添加不必要的後增量時，都會使編譯器不得不生成更大、更慢的代碼。這個清單是不是太長了，記不住？今天就開始培養好的習慣吧！在使用後增量操作前，先問問自己能不能把增量操作作為下一條語句。

結語

當然，軟體發展工作並不是只要求開發人員去“將就”編譯器，他們與編譯器之間的相互協同是快速而高效地完成程式設計工作的基礎之一。此外，從編譯器的發展過程來看，它們不僅要跟隨技術和語言的演進而反覆運算和創新，而且還要廣泛參考更多的開發習慣，那些歷史更悠久、使用更廣泛的編譯器可以為開發人員帶來更高的效率。

因此，在瞭解了如何編寫利於一款優秀編譯器優化的代碼之後，用戶們的工作效率就可以事半功倍。本文中提到的這些原理和 tips，也是 IAR Systems 這樣的公司長時間總結的最優實踐，而且都可以在該公司的 Embedded Workbench 中進行驗證和探索，在其工具介面中可以查看代碼的執行時間和代碼尺寸，從而找到最佳解決方案。



好的工具除了通用的代碼編譯優化，還支持高度靈活的自訂優化設置，如 IAR Embedded Workbench 包含針對運行效率和代碼體積的不同優化等級，對於不同的應用需求，還可以設置從整個工程，到每個原始程式碼檔，甚至是每個函數的優化等級，說明工程師為自己的應用適配出最佳的優化方案。希望此篇文章對於開發人員更深度地瞭解程式優化有所說明。 CTA

下期預告

Outlook and Review
回顧與展望