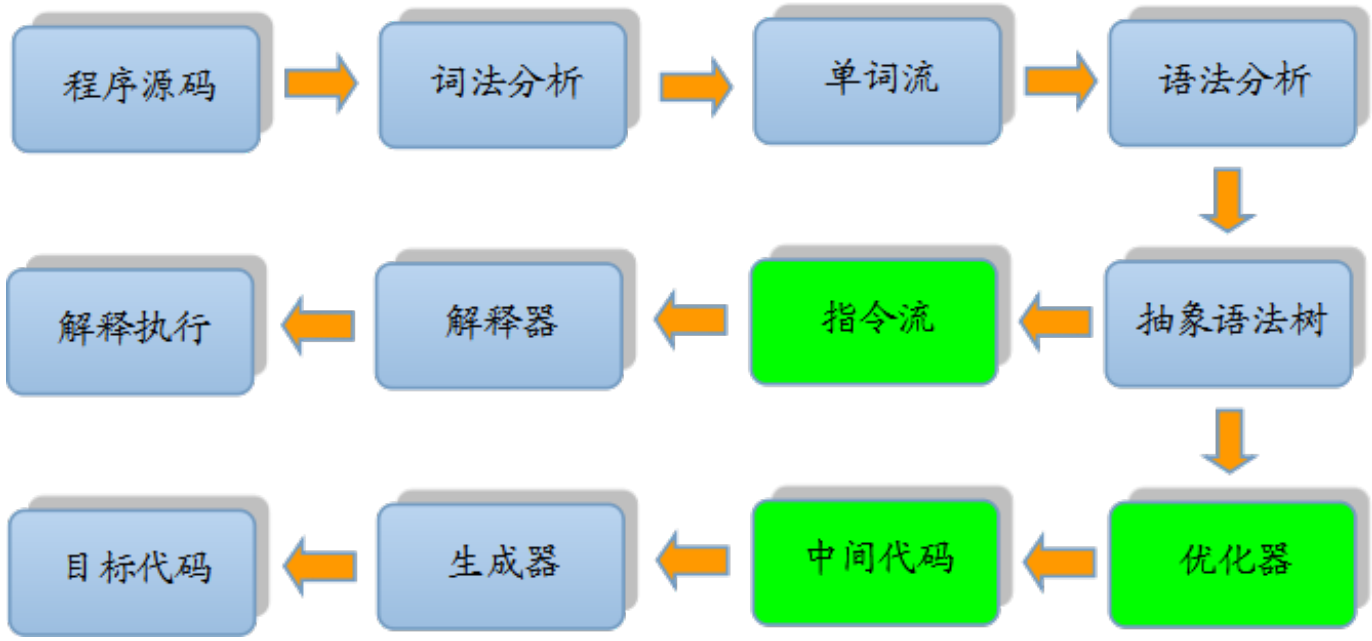
javac 編譯與 JIT 編譯

編譯過程

不論是物理機還是虛擬機，大部分的程序代碼從開始編譯到最終轉化成物理機的目標代碼或虛擬機能執行的指令集之前，都會按照如下圖所示的各個步驟進行：



其中綠色的模塊可以選擇性實現。很容易看出，上圖中間的那條分支是解釋執行的過程（即一條字節碼一條字節碼地解釋執行，如 JavaScript），而下面的那條分支就是傳統編譯原理中從源代碼到目標機器代碼的生成過程。

如今，基於物理機、虛擬機等的語言，大多都遵循這種基於現代經典編譯原理的思路，在執行前先對程序源碼進行詞法解析和語法解析處理，把源碼轉化為抽象語法樹。對於一門具體語言的實現來說，詞法和語法分析乃至後面的優化器和目標代碼生成器都可以選擇獨立於執行引擎，形成一個完整意義的編譯器去實現，這類代表是 C/C++語言。也可以把抽象語法樹或指令流之前的步驟實現一個半獨立的編譯器，這類代表是 Java 語言。又或者可以把這些步驟和執行引擎全部集中在一起實現，如大多數的 JavaScript 執行器。

javac 編譯

在 Java 中提到「編譯」，自然很容易想到 javac 編譯器將`*.java`文件編譯成為`*.class`文件的過程，這裡的 javac 編譯器稱為前端編譯器，其他的前端編譯器還有諸如 Eclipse JDT 中的增量式編譯器 ECJ 等。相對應的還有後端編譯器，它在程序運行期間將字節碼轉變成機器碼（現在的 Java 程序在運行時基本都是解釋執行加編譯執行），如 HotSpot 虛擬機自帶的 JIT（Just In Time Compiler）編譯器（分 Client 端和 Server 端）。另外，有時候還有可能會碰到靜態提前編譯器（AOT，Ahead Of Time Compiler）直接把`*.java`文件編譯成本地機器代碼，如 GCJ、Excelsior JET 等，這類編譯器我們應該比較少遇到。

下面簡要說下 javac 編譯（前端編譯）的過程。

詞法、語法分析

詞法分析是將源代碼的字符流轉變為標記（Token）集合。單個字符是程序編寫過程中的的最小元素，而標記則是編譯過程的最小元素，關鍵字、變量名、字面量、運算符等都可以成為標記，比如整型標誌 int 由三個字符構成，但是

它只是一個標記，不可拆分。

語法分析是根據Token序列來構造抽象語法樹的過程。抽象語法樹是一種用來描述程序代碼語法結構的樹形表示方式，語法樹的每一個節點都代表著程序代碼中的一個語法結構，如 bao、類型、修飾符、運算符等。經過這個步驟後，編譯器就基本不會再對源碼文件進行操作了，後續的操作都建立在抽象語法樹之上。

填充符號表

完成了語法分析和詞法分析之後，下一步就是填充符號表的過程。符號表是由一組符號地址和符號信息構成的表格。符號表中所登記的信息在編譯的不同階段都要用到，在語義分析（後面的步驟）中，符號表所登記的內容將用於語義檢查和產生中間代碼，在目標代碼生成階段，黨對符號名進行地址分配時，符號表是地址分配的依據。

語義分析

語法樹能表示一個結構正確的源程序的抽象，但無法保證源程序是符合邏輯的。而語義分析的主要任務是讀結構上正確的源程序進行上下文有關性質的審查。語義分析過程分為標註檢查和數據及控制流分析兩個步驟：

- 標註檢查步驟檢查的內容包括諸如變量使用前是否已被聲明、變量和賦值之間的數據類型是否匹配等。
- 數據及控制流分析是對程序上下文邏輯更進一步的驗證，它可以檢查出諸如程序局部變量在使用前是否有賦值、方法的每條路徑是否都有返回值、是否所有的受查異常都被正確處理了等問題。

字節碼生成

字節碼生成是 javac 編譯過程的最後一個階段。字節碼生成階段不僅僅是把前面各個步驟所生成的信息轉化成字節碼寫到磁盤中，編譯器還進行了少量的代碼添加和轉換工作。 實例構造器()方法和類構造器()方法就是在這個階段添加到語法樹之中的（這裡的實例構造器並不是指默認的構造函數，而是指我們自己重載的構造函數，如果用戶代碼中沒有提供任何構造函數，那編譯器會自動添加一個沒有參數、訪問權限與當前類一致的默認構造函數，這個工作在填充符號表階段就已經完成了）。

JIT 編譯

Java 程序最初是僅僅通過解釋器解釋執行的，即對字節碼逐條解釋執行，這種方式的執行速度相對會比較慢，尤其當某個方法或代碼塊運行的特別頻繁時，這種方式的執行效率就顯得很低。於是後來在虛擬機中引入了 JIT 編譯器（即時編譯器），當虛擬機發現某個方法或代碼塊運行特別頻繁時，就會把這些代碼認定為「Hot Spot Code」（熱點代碼），為了提高熱點代碼的執行效率，在運行時，虛擬機將會把這些代碼編譯成與本地平台相關的機器碼，並進行各層次的優化，完成這項任務的正是 JIT 編譯器。

現在主流的商用虛擬機（如Sun HotSpot、IBM J9）中幾乎都同時包含解釋器和編譯器（三大商用虛擬機之一的 JRockit 是個例外，它內部沒有解釋器，因此會有啟動相應時間長之類的缺點，但它主要是面向服務端的應用，這類應用一般不會重點關注啟動時間）。二者各有優勢：當程序需要迅速啟動和執行時，解釋器可以首先發揮作用，省去編譯的時間，立即執行；當程序運行後，隨著時間的推移，編譯器逐漸會返回作用，把越來越多的代碼編譯成本地代碼後，可以獲取更高的執行效率。解釋執行可以節約內存，而編譯執行可以提升效率。

HotSpot 虛擬機中內置了兩個JIT編譯器：Client Complier 和 Server Complier，分別用在客戶端和服務端，目前主流的 HotSpot 虛擬機中默認是採用解釋器與其中一個編譯器直接配合的方式工作。

運行過程中會被即時編譯器編譯的「熱點代碼」有兩類：

- 被多次調用的方法。
- 被多次調用的循環體。

兩種情況，編譯器都是以整個方法作為編譯對象，這種編譯也是虛擬機中標準的編譯方式。要知道一段代碼或方法是

不是熱點代碼，是不是需要觸發即時編譯，需要進行 Hot Spot Detection（熱點探測）。目前主要的熱點 判定方式有以下兩種：

- **基於採樣的熱點探測**：採用這種方法的虛擬機會週期性地檢查各個線程的棧頂，如果發現某些方法經常出現在棧頂，那這段方法代碼就是「熱點代碼」。這種探測方法的好處是實現簡單高效，還可以很容易地獲取方法調用關係，缺點是很難精確地確認一個方法的熱度，容易因為受到線程阻塞或別的外界因素的影響而擾亂熱點探測。
- **基於計數器的熱點探測**：採用這種方法的虛擬機會為每個方法，甚至是代碼塊建立計數器，統計方法的執行次數，如果執行次數超過一定的閥值，就認為它是「熱點方法」。這種統計方法實現複雜一些，需要為每個方法建立並維護計數器，而且不能直接獲取到方法的調用關係，但是它的統計結果相對更加精確嚴謹。

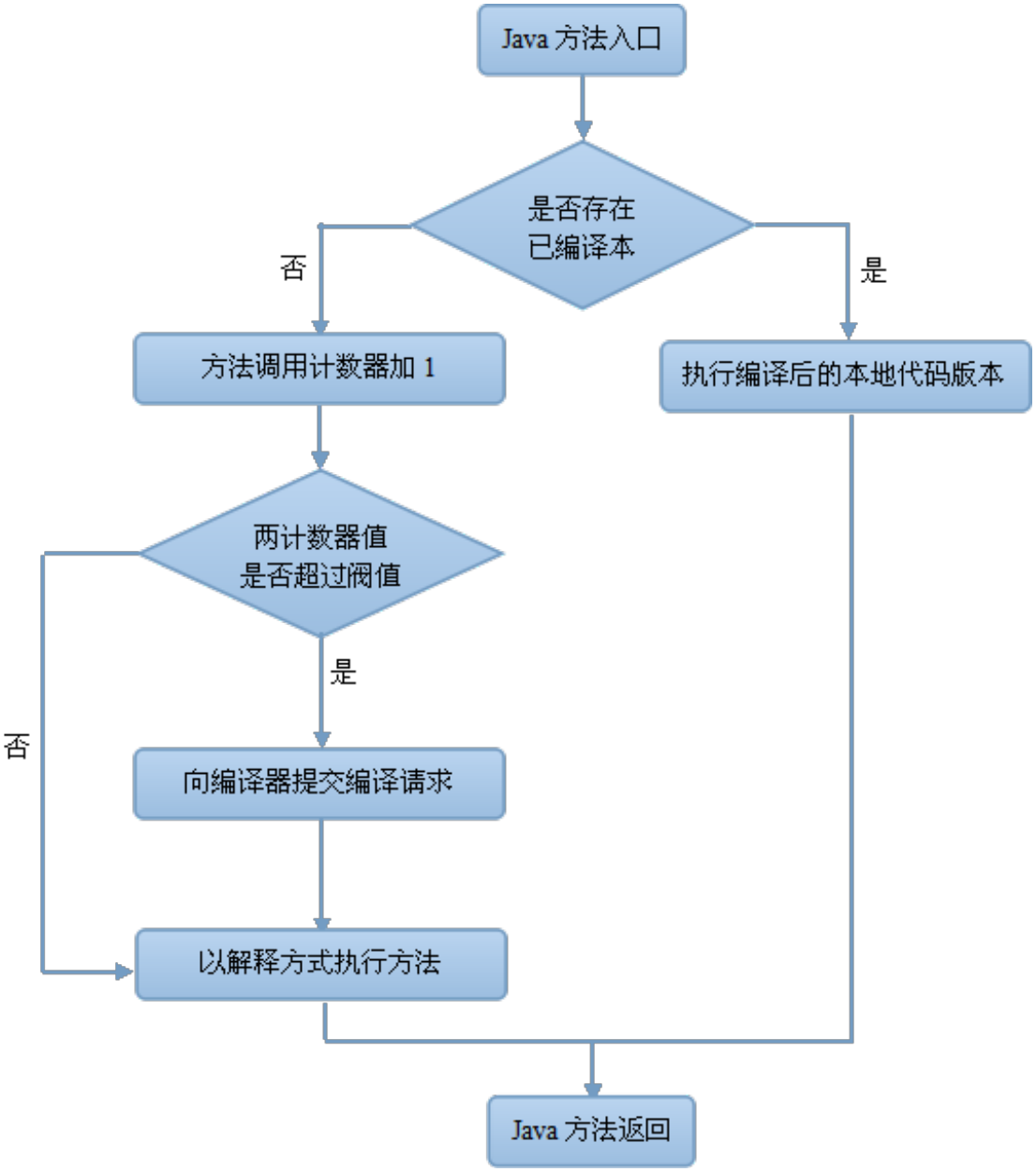
在 HotSpot 虛擬機中使用的是第二種——基於計數器的熱點探測方法，因此它為每個方法準備了兩個計數器：方法調用計數器和回邊計數器。

方法調用計數器用來統計方法調用的次數，在默認設置下，方法調用計數器統計的並不是方法被調用的絕對次數，而是一個相對的執行頻率，即一段時間內方法被調用的次數。

回邊計數器用於統計一個方法中循環體代碼執行的次數（準確地說，應該是回邊的次數，因為並非所有的循環都是回邊），在字節碼中遇到控制流向後跳轉的指令就稱為「回邊」。

在確定虛擬機運行參數的前提下，這兩個計數器都有一個確定的閥值，當計數器的值超過了閥值，就會觸發JIT編譯。觸發了 JIT 編譯後，在默認設置下，執行引擎並不會同步等待編譯請求完成，而是繼續進入解釋器按照解釋方式執行字節碼，直到提交的請求被編譯器編譯完成為止（編譯工作在後台線程中進行）。當編譯工作完成後，下一次調用該方法或代碼時，就會使用已編譯的版本。

由於方法計數器觸發即時編譯的過程與回邊計數器觸發即時編譯的過程類似，因此這裡僅給出方法調用計數器觸發即時編譯的流程：



javac 字節碼編譯器與虛擬機內的 JIT 編譯器的執行過程合起來其實就等同於一個傳統的編譯器所執行的編譯過程。