

第十八章 编译

↓ 第十八章 编译

- ↓ 18.1. Perl 程序的生命周期
 - ↓ 18.2 编译你的代码
 - ↓ 18.3 执行你的代码
 - ↓ 18.4 编译器后端
 - ↓ 18.5 代码生成器
 - ↓ 18.5.1 字节码生成器
 - ↓ 18.5.2. C 代码生成器
 - ↓ 18.7 提前编译, 回头解释
-

如果你到这里来是为了找一个 Perl 的编译器, 你可能很奇怪地发现你已经有一个了 —— 你的perl 程序 (通常是 `/usr/bin/perl`) 已经包含一个 Perl 编译器。这个东西可能不是你想要的, 如果不是你想象的东西, 你可能会很开心地得知我们还提供代码生成器 (也就是那些要求意义严格的人所谓的“编译器”), 我们将在本章讨论那些东西。但是首先我们想讲讲我们眼中的编译器是什么。本章不可避免地要讲一些底层的细节, 而有些人会喜欢这些内容, 有些人则不。如果你发现自己并不喜欢这些内容, 那就把它当作一个提高你阅读速度的练习吧。(呵呵, 不能不看, 但是可以不用全明白。)

假设你是一个指挥家, 任务是给一个大管弦乐队排练乐谱。当乐谱到货的时候, 你会发现有一些小册子, 管弦乐队成员人手一册, 每人只有自己需要演奏的部分乐章。但奇怪的是, 你的主乐谱里面什么东西也没有。而更奇怪的是, 那些有部分乐章的乐谱都是用纯英语写的, 而不是用音乐符号写的。在你开始准备一个演奏的节目之前, 或者甚至把乐谱给你的乐队演奏之前, 你首先得把这样的散文翻译成普通的音符和小节线。然后你要把所有独立的部分编辑成一个完整的乐谱, 这样你才能对整个节目有一个完整的印象。

与之类似, 当你把你的 Perl 脚本的源程序交给 perl 执行的时候, 对计算机而言, 它就象用英语写的交响乐对音乐家一样毫无用处。在你的程序开始运行之前, Perl 需要把这些看着象英文似的东西编译 (注: 或曰翻译, 或转换, 或改变或变形) 为一种特殊符号表现形式。不过你的程序仍然不会运行, 因为编译器只是编译。就象指挥的乐谱一样, 就算你的程序已经转换成一种容易解释的指令格式, 它仍然需要一个活跃的对象来解释那些指令。

18.1. Perl 程序的生命周期

你可以把一个 Perl 程序分裂为四个独立的阶段, 每个阶段都有自己的独立的子阶段。第一个和最后一个是最让人感兴趣的两个, 而中间的两个是可选的。这些阶段在图 18-1 里描绘。

1. 编译阶段

在第一阶段: 编译阶段里, Perl 编译器把你的程序转换成一个叫分析树的 数据结构。除了使用标准的分析技术以外, Perl 还使用了一种更强大的分析技术: 它使用 **BEGIN** 块引导编译进行得更深入。**BEGIN** 块一完成分析就转交给解释器运行, 结果是它们以 **FIFO** (先进先出) 顺序运行。这样处理的包括 **use** 和 **no** 声明; 它们实际上是伪装的 **BEGIN** 块。任何 **CHECK**, **INIT**, 和 **END** 块都由编译器安排推迟的执行。

词法声明也做上记号, 但是还不执行给它们的赋值。所有 **eval BLOCKS**, **s///e** 构造, 和非代换的规则表达式都在这里编译, 并且常量表达式都预先计算。现在编译器完成工作, 除非稍后它又被再次调用进行服务。在这个阶段的结尾, 再次调用解释器, 以 **LIFO** 顺序 (后进先出) 执行任何安排好了的**CHECK** 块。是否有 **CHECK** 块出现决定了我们下一步是进入第二阶段还是直接进入第四阶段。

2. 代码生成阶段（可选）

CHECK 块是由代码生成器装配的，因此这个阶段是在你明确地使用一个代码生成器（稍后在“代码生成器”里描述）的时候发生的。这时候把编译完成的（但还没运行的）程序转换成 **C** 源程序或者串行的 **Perl** 字节码——一个代表内部 **Perl** 指令的数值序列。如果你选择生成 **C** 源程序，它最后可以生成一个称做可执行影象的本机机器语言。（注：你最初的脚本也是一个可执行文件，但它不是机器语言，因此我们不把它称做影象。之所以称其为影象文件是因为它是一个你的 **CPU** 用于直接执行的机器编码的逐字拷贝。）

这时候，你的程序暂时休眠。如果你制作了一个可执行影象，那么你可以直接进入阶段 **4**；否则，你需要在阶段三里重新组成冻干的字节码。

3. 分析树重构阶段（可选）

要复活你的程序，它的分析树必须重新构造。这个阶段只有在发生了代码生成并且你选择生成字节码的情况下存在。**Perl** 必须首先从字节码序列中重新构造它的分析树，然后才能运行程序。**Perl** 并不直接从字节码运行程序，因为那样会比较慢。

4. 执行阶段

最后，就是你等待的时刻：运行你的程序。因此，这个阶段也称做运行阶段。解释器拿来分析树（可能是直接从编译器来的或者间接从代码生成阶段以及随后的分析树重构阶段过来的）并且运行之。（或者，如果你生成了一个可执行影象文件，它可以当作一个独立的程序运行，因为它包含一个内嵌的**Perl** 解释器。）

这个阶段的开始，在你的主程序运行之前，所有安排好了的 **INIT** 块以 **FIFO** 顺序执行。然后你的主程序运行。解释器在碰到一个 **eval STRING**，一个 **do FILE**或者 **require** 语句，一个 **s///ee** 构造，或者一个代换变量里含有合法代码断言的模式匹配的时候会根据需要回过头调用编译器。

当你的主程序结束之后，最后执行任何推迟的 **END** 块，这回是以 **LIFO** 顺序。最早的一个最后执行，然后你的程序结束。（**END** 块只有在你的 **exec** 或者你的进程被一个未捕获的灾难性错误摧毁后才能忽略。普通的例外都不认为是灾难性的。）

下面我们将以非常详细的方式讨论这些阶段，而且是以不同的顺序。

18.2 编译你的代码

Perl 总是处于两种操作模式之一：要么它在编译你的程序，要么是在执行它——从来不会同时处于两种模式。在我们这本书里，我们把某些事件称做在编译时发生，或者我们说“**Perl** 编译器做这做那”。在其他地方，我们说某些东西在运行时发生，或者说“**Perl** 的解释器做这做那”。尽管你可以认为“**Perl**”就是编译器和解释器的合体，但是把这 **Perl** 在不同场合所扮演的两个角色之一区分清楚还是非常重要的，这样你才能理解许多事情发生的原因。（也可能有其他角色：**perl** 还是一个优化器和代码生成器。有时候，它还是一个骗子——不过都是善意的玩笑。）

同时，区分编译阶段和编译时，以及运行阶段和运行时之间的区别也非常重要。一个典型的 **Perl** 程序只有一个编译阶段，然后只有一个运行阶段。“阶段”是一个大范围的概念。但是编译时和运行时是小范围的概念。一个给定的编译阶段大多数时间做的是编译时工作，但是也通过 **BEGIN** 块做一些运行时工作。一个给定的运行阶段大多数时间做的是运行时工作，但是它也可能通过类似 **eval**

STRING 这样的操作符做编译时任务。

在典型的事件过程中，**Perl** 先阅读你的整个程序然后才开始执行。这就是 **Perl** 分析声明，语句和表达式，确保它们语法上是正确的时候。（注：不，这个过程中没有正式的象 **BNF** 那样的语法图表，不过我们欢迎你细读 **Perl** 源程序树里的 **perly.y** 文件，里面包含 **Perl** 用的 **yacc(1)** 语法。我们建议你离词法远一些，因为它让小白鼠产生进食紊乱的症状了。译注：呵呵，象大话里的唐僧。）如果它发现语法错误，编译器会试图从错误中恢复过来，这样它才能汇报任何后面的源程序的错误。有时候可以这样恢复，但是有时候不行；语法错误有一个很讨厌的问题是会出发一系列错误警告。**Perl** 在汇报近 10 个错误后暂时退出。

除了处理 **BEGIN** 块的解释器以外，编译器默许三个概念上的过程处理你的程序。词法分析器（**lexer**）扫描你的程序里的每一个最小的单元。这些东西有时候称为“词位”（**lexemes**），但你在讲述编程语言的文章里看到的可能更多的是“记号”（**token**）。词法分析器有时候被称做标记器或扫描器，而它干的工作有时候被称做是词法分析或记号分析。然后分析器（**parser**）以 **Perl** 语言的语法为基础，试图通过把这些记号组合成更大的构造，比如表达式和语句，来获取合适的意义，优化器（**optimizer**）对这些词法的组合进行重新排列并且把它们归减成更有效的序列。优化器仔细地选择最优的方法，它不会在边缘优化上花费时间，因为 **Perl** 编译器用做即时编译器时必须运行得极快。

这些过程并不是在相互独立的阶段进行的，而是同时发生，并且相互之间有大量交互。词法分析器偶尔需要来自 **parser** 的提示，这样它才能够知道它需要注意哪几种记号类型。（很奇怪的是，词法范围就是词法分析器不能理解的事物之一，因为那是“词法”的其他含义。）优化器同时还需要跟踪分析器的处理，因为有些优化在分析器到达某一点之前是无法进行的，比如完成一个表达式，语句，块，或者子过程。

你可能会奇怪，为什么 **Perl** 编译器同时做这些事情，而不是一件一件做呢？因为这个混乱的过程就是当你在听取或者读取自然语言的时候，你即时地理解它们的过程。你用不着直到读到本章的结束才理解第一句话的含义。你可以想象下面的对应关系：

计算机语言	自然语言
字符	字母
记号	词素
术语	词
表达式	短语
语句	句子
块	段落
文件	章节
程序	故事

如果分析过程进展顺利，编译器就认为你输入了一则合法的故事，哦，是程序。如果你运行程序的时候使用了 **-c** 开关，那么编译器会打印一条“**syntax OK**”消息然后退出。否则，编译器会把它自己的成果转交给其他过程。这些“成果”是以分析树的形式表示的。在分析树上的每个“果实”——或者称做节点——代表一个 **Perl** 内部的操作码，而树上的分支代表树的历史增长模式。最后，这些节点都会线性地一个接一个地串在一起，以表示运行时系统的访问这些节点的执行顺序。

每个操作码都是 **Perl** 认为的最小的可执行单元。你可能见过一条象 **\$a = -(\$b + \$c)** 这样的语句，但是 **Perl** 认为它是六个独立的操作码。如果让我们用一种简单的格式来表示，上面那个表达式的分析树看起来象图 18-2。黑圆点里的数字代表 **Perl** 运行时系统将遵循的访问顺序。

Perl 不是有些人想象的那种单回合编译器。（单回合编译器是那种把事情做得对计算机简单而对程序员复杂的东西。）**Perl** 编译器是那种多回合的，优化的编译器，它是由至少三种不同的，相互交错的

逻辑回合组成的。回合 1 和 2 在编译器在分析树上上窜下跳构造执行流的时候轮流运行，而回合 3 在一个子过程或者文件完全分析完的时候运行。下面是那些回合：

- 回合 1：自底向上分析

在这个回合里，分析树是由 **yacc(1)** 分析器建造的，用的记号是从下层的词法分析器处理出来的（那个过程也可以认为是另外一个逻辑回合）。自底向上只是意味着该分析器先知道树叶后知道树支和树根。它并不是象在图 18-2 里那样从底向上处理各种元素，因为我们是在顶部画树根的，这是

计算机科学家（和语言学家）的特殊的传统。

在构造每个操作码节点的时候同时对每个操作码进行完成性检查，以校验语意是否正确，比如是否使用了正确数量和类型的参数来调用内建函数。在完成一条分析树的分支以后，优化器就参与进来看看现在它能否对整个子树做某些转换。比如，一旦它知道我们给某个接收一定数量参数的函数一列数值，那么它就可以把记录可变量参数函数的参数个数的操作码仍掉。还有一个更重要的优化，我们称之为常量消除，将在本节稍后讲述。

这个回合同时还构造稍后执行的时候要用的节点访问顺序，这个处理也几乎完全是戏法，因为第一个 要访问的地方几乎从来都不是顶端节点。编译器把操作码作成临时的循环，把顶端节点指向第一个要访问的操作码。如果顶端操作码无法和更大的操作码匹配，那么这个操作码环就破裂了，于是更大的操作码就成为新的顶端节点。最后是这个环因为进入其他结构里（比如子过程描述符）而合理破裂。尽管其分析树会一直延伸到树的底部（象图18-2里那样）子过程调用者仍然可以找到第一个操作码。而且这里不需要解释器回溯到分析树里寻找从哪里开始。

- 回合 2：自顶向下优化

如果你在阅读一小段 **Perl** 代码（或者英文文章），那么如果你不检查上下文的词法元素的话，你就无法判断环境。有时候你在获取更多的消息之前无法判断真正将要发生什么事情。不过，不必害怕，因为你并不孤独：编译器也

一样。在这个回合里，在它刚创建的子树上向回退，以便进行局部优化，最需要注意的事情是环境传播。编译器用当前节点产生的恰当的环境（空，标量，列表，引用或者左值等）标记到相邻的下层节点上。不需要的操作码被清空但并不删除，因为现在重新构造执行顺序已经太晚了。我们将依赖第三回合把他们从第一回合决定了的临时执行顺序中删除。

- 回合 3：窥探孔优化器

有些代码单元有自己的存储空间，它们在里面保存词法范围的变量。（在 **Perl** 的说法里，这样的空间称为便条簿（**scratchpad**））。这样的单元包括 **eval STRING**，子过程和整个文件。从优化器的角度来说，更重要的是它们 1. 有自己的进入点，这就意味着尽管我们知道从这里开始的执行顺序，我们也不知道以前发生过什么，因为这个构造可能是从其他什么地方调用的。因此如果一个这样的单元被分析器分析完成，**Perl** 就在那段代码上运行一个窥探孔优化器。和前面两个回合不同的是，前面两个回合在分析树的各个分支上运行，而这个回合是以线性执行顺序横跨代码，因为这里基本上是我们采取这个步骤的最后的机会了，然后我们就要从分析器上砍断操作码列表了。大多数优化已经在头两个回合里完成了，但是有些不行。

最后的分类优化在这个阶段发生，包括把最后的执行顺序缝合在一起，忽略清空了的操作码，以及识别什么时候可以把各种操作码缩减成更简单的东西。识别字串的链接就是一个非常重要的优化，因为你肯定不想每次向字串结尾加一点东西的时候就要到处拷贝字串。这个回合不仅仅做优化；它还做大量“实际”的工作：捕获光字，在有问题的构造上生成警告信息，检查那些可能无法抵达的代码，分析伪哈希键字，以及在子过程的原型被编译前寻找它们。

- 回合 4：代码生成

这个回合是可选的；在普通的情况下并不出现这个回合。但是，如果调用了任何三个代码生成器之一——**B::Bytecode**，**B::C**，和 **B::CC**，那么就会最后再访问一次分析树。代码生成器要么发出用于稍后重新构造分析树的串行的 **Perl** 字节码，要么是代表编译时分析树状态的文本 **C** 代码。

C 代码的生成源自两种不同的风格。**B::C** 简单地重新构造分析树，然后用**Perl** 在执行的

时候自己用的普通的 `runops()` 循环运行之。**B::CC** 生成一个线性化了并且优化过的运行时代码路径（组成一个巨大的跳转表）的 **C** 等效物，并且执行之。

在编译的时候，**Perl** 用许多方法优化你的代码。它重新排列代码好让它在运行时更有效。它删除那些在执行时可能永远不会到达的代码，比如一个 `if(0)` 块，或者在 `if(1)` 块里的 `elsif` 和 `else`。如果你使用 `my ClassName2 $var` 或 `our ClassName2 $var` 声明的词法类型，而且 `ClassName2` 包是用 `use fields` 用法设置的，那么对下层的伪哈希的常量域的访问会在编译时进行拼写检查并且转换成一个数组访问。如果你给 `sort` 操作符一个足够简单的比较路径，比如 `{$a <=> $b}` 或者 `{$b cmp $a}`，那么它会被一个编译好的 **C** 代码代替。

Perl 里最富戏剧性的优化可能就是它尽可能快地解析常量表达式的方法。比如，让我们看看图 18-2 的分析树。如果节点 1 和 2 都有文本和常量函数，节点 1 到 4 将已经被那些计算代替了，就象图 18-3 的分析树：

图 18-3（略。。。）

这就叫常量消除。常量消除并不仅限于象把 `2**10` 转成 `1024` 这么简单的场合。它还解析函数调用——包括内建的和用户定义的子过程，只要它们符合第六章，子过程，的“内联常量函数”的标准。回想一下 **FORTRAN** 编译器对它们内在函数臭名昭著的知识，**Perl** 在编译的过程中也知道要调用它的哪些内建函数。这就是为什么如果你试着做 `log(0.0)` 或者 `sqrt`（求平方根）一个负数的时候，会导致一个编译错误，而不是一个运行时错误，并且解释器根本没有运行。（注：实际上，我们在这里实在是简化的太厉害了。解释器实际上是运行了，因为那就是常量消除器实现的方法。不过它是在编译时立即运行的，类似 **BEGIN** 块执行的方式。）

甚至任意复杂的表达式都是提前解析的，有时候导致整个块的删除，象下面这个：

```
if (2* sin(1)/cos(1) < 3 && somefn() ) { whatever() }
```

那些永不计算的东西不会生成任何代码。因为第一部分总是假，所以 `somefn` 和 `whatever` 都不会调用。（所以不必期待那个语句块里会有 `goto` 标签，因为它甚至都不会在运行时出现。）如果 `somefn` 是一个可以内联的常量函数，那么即使你把上面的顺序换成下面这样：

```
if ( somefn() && 2*sin(1)/cos(1) <3 )) { whatever() }
```

也不会改变输出，因为整个表达式仍然在编译时解析。如果 `whatever` 可以内联，那么它在运行时不会被调用，甚至在编译的时候也不会；它的值会被当作一个文本常量那样嵌入程序中。然后你会收到一个警告 “**Useless use of a constant in void context**”。如果你没有意识到它是常量，你可能会觉得奇怪。不过，如果 `whatever` 是一个在非空的环境中（就象由优化器决定的那样）计算的最后一条语句，那么你就看不到警告。

你可以用 `perl -Dx` 看到在所有优化阶段完成之后构造的分析树的最终结果。（`-D` 要求你用的是特殊的，制作时打开调试的 **Perl**）。还可以看看后面描述的 **B::Deparse** 节。

总而言之，**Perl** 编译器为优化代码工作得很努力（不过不是特别努力），然后来的就是运行时整体的执行速度提高了。现在就是让你的程序运行的时候了，所以让我们讨论它吧。

18.3 执行你的代码

打个比方，**Sparc** 程序只能运行在 **Sparc** 机器上，**Intel** 程序只能运行在 **Intel** 的机器上，而 **Perl** 程序只能运行在 **Perl** 机器上。**Perl** 机器处理那些 **Perl** 程序认为在一台计算机里完美的属性：内存是自动分配和释放的，基本数据类型是动态的字串，数组和哈希，而且没有尺寸限制，并且系统表现得都非常相象。**Perl** 解释器的工作就是把它所运行的任何计算机都搞得象那种理想的 **Perl** 机器一样。

这样的假想的机器就好像是一种特别设计来运行 Perl 程序的机器一样。编译器生成的每个操作码都是这种假想的指令集中的一个基本命令。Perl 里没有使用硬件程序计数器，而是由解释器跟踪当前要执行的操作数。Perl 里也没有硬件堆栈指针，解释器有它自己的虚拟堆栈。这个堆栈非常重要，因为 Perl 虚拟机（我们拒绝称之为 PVM）是一个基于堆栈的机器。Perl 操作码在内部称做 PP 代码（“压栈-出栈代码”（“push-pop codes”）因为它们操作解释器的虚拟堆栈以寻找所有操作数，处理临时数值，还有存储所有结果。

如果你曾经用 Forth 或者 PostScript² 写过程序，或者用 RPN（“反转润色符号” "Reverse Polish Notation"）一个 HP 的科学计算器记录，你就知道堆栈机器是如何运行的了。甚至如果你没有用过这些东西，它的概念也是简单的：要把 3 和 4 相加，你按照 3 4 + 这样的顺序做处理而不是习惯的 3 + 4。从堆栈的角度来看，这里的意思是你把 3 然后是 4，压入堆栈，而 + 则把两个参数弹出堆栈，把它们相加，然后把 7 压回堆栈，然后 7 就会留在那里直到你对它进行其他处理。

与 Perl 的编译器相比，Perl 的解释器是非常直接的，直接得几乎让人厌倦的程序。它所做的一切就是走过那些编译出来的操作码，每次一个，然后把它们发配给 Perl 运行时环境，也就是 Perl 虚拟机。它只不过是一小堆 C 代码，对吧？

实际上，它一点也不乏味。Perl 的虚拟机替你跟踪一堆动态环境，这样你就不用跟踪了。Perl 维护不少堆栈，你用不着理解它们，但是我们会在这里列出来好加深你的印象：

- 操作数堆栈（operand stack）

这个堆栈我们已经讲过了。

- 保存堆栈（save stack）

在这里存放等待恢复的局部数值。许多内部过程也有许多你不知道的局部值。

- 范围堆栈（scope stack）

轻量的动态环境，它控制何时保存堆栈应该”弹出“。

- 环境堆栈（context stack）

重量级的动态环境；是谁调用了谁最后把你放到了你现在所处的位置。caller 函数遍历这个堆栈。循环控制函数扫描这个堆栈以找出需要控制哪个循环。如果你从环境堆栈剥出，那么范围堆栈也相应剥出，这样就从你的保存堆栈里恢复所有局部变量，甚至你用一些极端的方法，比如抛出例外或者 longjmp(3) 出去等也是如此。

- 环境跳转堆栈（jumpenv stack）

longjmp(3) 环境的堆栈，它允许你抛出错误或者迅速退出。

- 返回堆栈（return stack）

我们进入这个子过程时的来路。

- 标记堆栈（mark stack）

在操作数堆栈里的列出的当前的杂参数的起点。

- 递归词法填充堆栈（recursive lexical pad stacks）

当子过程被递归地调用时词法变量和其他“原始寄存器”存放的地方。

当然，还有存放所有 C 变量的 C 堆栈。Perl 实际上力图避免依赖 C 的堆栈存储保存的数值，因为 longjmp(3) 忽略了这样的数值的合理的恢复。

所有的这些就是说，我们对解释器的通常的看法：一个解释另外一个程序的程序，是非常不足以描述其内部的真正情况的。的确，它的内部是一些 C 的代码实现了一些操作码，但当我们提到“解释器”的时候，我们所说的含义要比上面这句话更多些，就好象我们谈到“音乐家”时，我们说的含义可不仅仅是一个可以把符号转换成声音的 DNA 指令集。音乐家是活生生的“有状态”的有机组织。解释器也一样。

具体来说，所有这些动态和词法范围，加上全局符号表，带上分析树，最后加上一个执行的线索，就是我们所谓的一个解释器。从执行的环境来看，解释器实际上在编译器开始运行之前就存在了，并且甚至是在编译器正在制作它的环境的时候解释器就开始进行初步的运行了。实际上，这就是当编译器调用解释器执行 BEGIN 块等的时候发生的事情。而解释器可以回过头来使用编译器进一步制作自己的运行环境。每次你定义另外一个子过程或者装载另外一个模块，那么我们称之为解释器的特定的虚拟 Perl 机器实际上就在重新定义自身。你实际上不能说是编译器还是解释器在控制这一切，因为它们合作控制我们通常称之为“运行 Perl 脚本”的引导过程。就象启动一个孩子的大脑。是 DNA 还是蛋白质在做处理？我们认为两个都在起作用，并且还有一一些来自外部程序员的输入。

我们可以在一个进程里运行多个解释器；它们可以或者不可以共享分析树，取决于它们是通过克隆一个现有解释器生成的还是通过从头开始制作一个新的解释器生成的。我们也可能在一个解释器里运行多个线程，这种情况下我们不仅共享分析树而且还共享全局符号——参阅第十七章，线程。

不过大多数 Perl 程序只使用一个 Perl 解释器执行它们编译好了的代码。并且尽管你可以在一个进程里运行多个独立的 Perl 解释器，目前可以实现这个目的 API 只能从 C 里访问。（注：到目前为止只有一个例外：Perl 5.6.0 可以在 Microsoft Windows 的仿真 fork 的支持下实现克隆解释器。到你阅读到此处时，可能也有一个实现 "ithread" 的 Perl API。）每个独立的 Perl 解释器起到一个完全独立的进程的作用，但是并不象创建一个完全新的进程那样开销巨大。这就是为什么 Apache 的 mod_perl 扩展的性能如此突出的原因：当你在 mod_perl 里启动一个 CGI 脚本时，那个脚本已经被编译成 Perl 的操作码了，消除了重新编译的需要——但是更重要的是，消除了启动一个新进程的需要，这才是真正的瓶颈。Apache 在一个现存的进程里初始化一个新的 Perl 解释器然后把前面编译完了的代码交给它执行。当然，这里头的东西要远比我们说的多——一直是这样的。更多关于 mod_perl 的东西，请参考 Writing Apache Modules with Perl and C (O'Reilly, 1999)。

许多其他应用都可以内嵌 Perl 解释器，比如 nvi, vim 和 innd；我们可不指望在这里把它们都列出来。而且还有许多甚至都不敢宣传它们有内嵌的 Perl 引擎的商业产品。它们只是在内部使用它，因为它能按照他们的风格实现他们的程序。

18.4 编译器后端

所以，如果 Apache 可以现在编译一个 Perl 程序而稍后才执行它，你为什么不行？Apache 和其他包含内嵌 Perl 解释器的程序做得非常简单——它们从来不把分析树存到一个外部文件中。如果你对这样的做法表示满意，而且不介意使用 C API 获得这样的特性，那么你可以做一样的事情。参阅第二十一章，内部和外部，里的“嵌入 Perl”一节，获取如何从一个闭合的 C 框架里访问 Perl 的信息。

如果你不想走这条路或者有其他需要，那么还有几个选择可用。你可以不让来自 Perl 编译器的输出立即输入 Perl 解释器，而是调用任意可用的后端。这些后端可以把编译好的操作码串行化和存储到任何外部文件中，甚至可以把它们转换成几种不同风格的 C 代码。

请注意那些代码生成器都是非常试验性的工具，在生产环境中不可靠。实际上，你甚至都不能指望它们在非生产环境里面能用——除了极为稀有的情况以外。现在我们已经把你的期望值降得足够低了，这样任何成功都可以比较容易超过它们，这时候我们才能放心地告诉你后端是如何运行的。

有些后端模块是代码生成器，比如 B::Bytecode, B::C, 和 B::CC。其他的实际上都是代码分析和

调试工具，比如 `B::Deparse`，`B::Lint`，和 `B::Xref`。除了这些后端以外，标准版还包括几种其他的底层模块，那些潜在的 Perl 代码开发工具的作者可能对它们感兴趣。其他的后端模块可以在 CPAN 找到，包括（到我们写这些为止）`B::Fathom`，`B::Graph`，`B::JVM::Jasmin`，和 `B::Size`。

如果你除了给解释器提供输入以外还有其他地方使用 Perl 编译器，那么 `O` 模块（也就是 `O.pm` 文件）位于编译器和你分配的后端模块之间。你并不直接调用该后端；相反，你调用中间层，然后由它调用你指定的后端。因此如果你有一个模块调用 `B::Backend`，你可以在一个脚本里这样来调用：

```
%perl -MO=Backend SCRIPTNAME
```

有些后端需要选项，用下面的方法声明：

```
%perl -MO=Backend, OPTS SCRIPTNAME
```

有些后端已经有调用它们的中间层的前端了，所以你不必费心记忆它们的 `M.O`。尤其是 `perlcc(1)` 调用那个代码生成器，而代码生成器启动起来可能比较麻烦。

18.5 代码生成器

目前的三种把 Perl 操作码转换成其他格式的后端都是处于实验阶段的。（没错，我们前面说过这些，但是我们不想你忘记这点。）甚至就算它们生成的输出碰巧能正确运行，生成的程序也可能比平常需要更多的磁盘空间，更多的存储器，和更多的 CPU 时间。这是一个正在进行的研究可开发领域。不过一切都会慢慢好起来的。

18.5.1 字节码生成器

`B::Bytecode` 模块将分析树的操作码以平台无关的编码写出。你可以把一个 Perl 脚本编译成字节码然后把它们拷贝到另外一台安装了 Perl 的机器上跑。

`perlcc` 命令知道怎么把一个 Perl 源程序转换成一个编译成字节码的 Perl 程序。这个命令是标准的，不过仍然处于实验阶段。你要做的事情只是：

```
%perlcc -b -o pbyscript srcscript
```

然后你就应该能直接“执行”所生成的 `pbyscript`。该文件的开头看起来象下面这样：

```
#!/usr/bin/perl
use ByteLoader 0.03;
^C^@^E^A^C^@^@^@^A^F^@^C^@^@^@^B^F^@^C^@^@^@^C^F^@^C^@^@^@
B^@^@^@^H9^A8M-^?M-^?M-^?M-^?7M-^?M-^?M-^?6^@^@^@^A6^@
^G^D^D^@^@^@^KR^@^@^@HS^@^@^@HV^@M-2W<^FU^@^@^@X^Y^Z^@
...
```

你会看到一小段脚本头后面跟着一堆纯二进制数据。这些东西看起来非常神秘，不过其实不过是一个技巧而已。`ByteLoader` 模块使用一种叫做“源码过滤器”的技巧在 Perl 能够看到源程序之前修改它们。源码过滤器是一种预处理器，它接收当前文件中在它后面的所有内容。与类似 `cpp(1)` 和 `m4(1)` 这样的宏预处理器不同，它们只能做简单的转换，而源码过滤器没有限制。源码过滤器已经用于检查 Perl 的语法，用于压缩或加密源代码，甚至用 Latin. E `perlibus unicode; cogito, ergo substr`；挑剔的 `dbm`，等写 Perl 程序；

[ByteLoader²](#) 模块是源码过滤器，它知道如何把 `B::Bytecode` 生成的串行的字节码分解并重新组合成原来的分析树。这样重新组合的 Perl 代码被接合进入当前分析树，不需要通过编译器。当解释器拿到这些操作码，它就开始执行它们，就好象它们早就在那里一样。

18.5.2. C 代码生成器

剩下的代码生成器，**B::C** 和 **B::C** 都生成 C 代码，而不是串行化的 Perl 操作码。它们生成的代码非常难读，如果你想试着读他们那你就傻了。它可不是那种转换好了的 Perl 到 C 的代码片段，可以插入到一个更大的 C 程序里。关于那方面的内容，请参阅第二十一章。

B::C 模块只是把创建整个 Perl 运行时环境所需要的 C 数据结构写出来。你得到一个专用的解释器，它的所有由编译器制作的数据结构都已经初始化好了。从某种意义上来说，所生成的代码类似 **B::Bytecode** 生成的东西。它们都是编译器制作的操作码树的直接翻译，只不过 **B::Bytecodes** 把他们以符号的形式输出，这些符号稍后可以重建操作码树并且插入到一个运行着的 Perl 解释器，而 **B::C** 把那些操作码输出为 C 代码。当你用你的 C 编译器编译这些 C 程序并且把它们和 Perl 库链接，生成的程序就不需要在目标系统安装 Perl 解释器就可以运行。（不过，它可能需要一些共享库——如果你没有把所有的东西都静态链接的话。）不过，这个程序和那些运行你的脚本的普通的 Perl 解释器没有什么根本的区别。它只不过是预先编译成一个独立的可执行影像而已。

不过，**B::CC** 模块试图做得更多。它生成的 C 源文件的开头看起来很像 **B::C** 生成的东西，（不过，当你犯傻的时候当然什么东西看起来都一样。我们难道没有告诉你别看吗？）不过，最终所有相似都会消失。在 **B::C** 生成的代码里，有一个 C 程序的很大的操作码表，它的作用就象解释器自己做的处理，而 **B::CC** 生成的 C 代码是以你的程序对应的运行时顺序为布局输出的。它甚至还有 C 函数对应你的程序的每个函数。它做了一些基于变量类型的优化；有些速度测试可以比在标准的解释器里快一倍。这是目前的代码生成器中最具野心的一个，也是对未来做出了最多承诺的一个。不过同时也是最不稳定的一个。

那些为毕业设计找主题的计算机科学系的学生可以在这里仔细找找。这里有大量还未琢磨的钻石等待你们发掘。

18.6 代码开发工具

O 模块里有许多很有趣的操作数方法（**Modi Operandi**），可以用来给易怒的实验性代码生成器做输入用。这个模块给你提供了相对而言痛苦少些的访问 Perl 编译器输出的方法，这样你就可以比较轻松地制作那些认识 Perl 程序所有内涵所需要的其他工具。

B::Lint 模块是参考 **lint(1)** 命名的，**lint(1)** 是 C 程序的校验器。它检查那些常常绊倒初学者但又不会触发警告的有问题的构造。直接调用这个模块：

```
%perl -MO=Lint, all myprog
```

目前只定义了几个检查，象在一个隐含的标量环境里使用数组啦，依赖缺省变量啦，以及访问另外一个包（通常是私有包）中以_开头的标识啦。参阅 **B::lint(3)** 获取细节。

B::Xref 模块生成一个交叉引用，里面列出一个程序里的声明以及所有变量（包括全局和词法范围）的使用，子过程，和格式，用文件和子过程分隔。用下面方法调用这个模块：

```
%perl -MO=Xref myprog > myprof.pxref
```

举例来说，下面就是一部分输出：

```
Subroutine parse_argv
  Package (lexical)
    $on          i113, 114
    $opt         i113, 114
    %getopt_cfg  i107, 113
    @cfg_args    i112, 114, 116, 116
  Package Getopt::Long
```

```

$ignorecase      101
&GetOptions      &124
Package main
$Options          123, 124, 141, 150, 165, 169
%$Options         141, 150, 165, 169
&check_read      &167
@ARGV            121, 157, 157, 162, 166, 166

```

这里显示出 `parse_argv` 子过程自己有四个词法变量；它还通过 `main` 包和 `Getopt::Long` 包访问全局标识符。列表中的数字是使用这些项的行号：前导的 `i` 表明该项在随后的行号首次引入，而一个前导 `&` 意味着在这里有一个子过程调用。析引用分别列出，于是 `$Options` 和 `%$Options` 都会显示出来。

B::Deparse 是一个很好的打印机，它可以揭开 Perl 代码神秘的面纱，帮助你理解优化器为你的代码做了那些转换。比如，下面的东西显示了 Perl 给各种构造使用了什么缺省：

```

% perl -MO=Deparse -ne 'for (1 .. 10) { print if -t }'
LINE: while (defined($_ = )) {
    foreach $_ (1 .. 10) {
        print $_ if -t STDIN;
    }
}

```

`-p` 开关给你的程序加圆括号，这样你就可以看到 Perl 对优先级的看法：

```

%perl -MO=Deparse, -p -e 'print $a ** 3 + sqrt(2) /10 ** -2 ** $c'
print((((($a ** 3) + (1.4142135623731 / (10 ** -(2 ** %c))))));

```

你可以使用 `-p` 看看哪个优先代换的字串编译到代码里：

```

%perl -MO=Deparse, -q -e '"A $name and some @ARGV\n"'
'A ' . $name . ' and some ' . join("$", @ARGV) . "\n";

```

下面的例子显示了 Perl 是怎样把一个三部分的 `for` 循环变成一个 `while` 循环：

```

%perl -MO=Deparse -e 'for ($i=0;$i<0;$i++) { $x++ }'
$i = 0;
while ( $i < 10 ) {
    ++$x;
}
continue {
    ++$i
}

```

你甚至可以在一个 `perlcc -b` 生成的 Perl 字节码上调用 **B::Deparse**，让它为你反编译那段二进制文件。串行化的 Perl 操作码可能有点难读，但并不是强加密的东西。

18.7提前编译，回头解释

做事情的时候总有考虑所有事情的合适时机：有时候是在做事之前，有时候是做事之后。有时候是做事情的过程中。Perl 并不假定何时是适合考虑的好时机，所以它给程序员许多选项，好让程序员告诉它什么时候思考。其他时间里它知道有些东西是必要的，但它不知道应该考虑哪个方案，因此它需要一些手段来询问你的程序。你的程序通过定义一些子过程来回答这些问题，这些子过程名字与 Perl 试图找出来的答案相对应。

不仅编译器可以在需要提前思考的时候调用解释器，而且解释器也可以在想修改历史的时候回过头来

调用编译器。你的程序可以使用好几个操作符回过头来调用编译器。和编译器类似，解释器也可以在需要的时候调用命名子过程。因为所有这些来来回回都是在编译器，解释器，和你的程序之间进行的，所以你需要清楚何时发生何事。首先我们谈谈这些命名子过程何时被触发。

在第十章，包，里，我们讲了如果该包里的一个未定义函数被调用的时候，包的 **AUTOLOAD** 子过程是如何触发的。在第十二章，对象，里我们提到了 **DESTROY** 方法，它是在对象的内存要自动被 **Perl** 回收的时候调用的。以及在第十四章，捆绑变量，里，我们碰到了许多访问一个捆绑了的变量是要隐含地调用的函数。

这些子过程都遵循一个传统：如果一个子过程会被编译器或者解释器自动触发，那么我们用大写字母为之命名。与你的程序的生命期的不同阶段相联的是另外四个子过程，分别是 **BEGIN**，**CHECK**，**INIT**，和 **END**。它们前面的 **sub** 关键字是可选的。可能最好叫它们“语句块”，因为它们从某种程度上来说更象命名语句块而不象真的子过程。

比如，和普通的子过程不同的是，你多次定义这些块不会有任何问题，因为 **Perl** 会跟踪何时调用它们，因此你不用通过名字调用它们。（它们还和普通子过程不同的是 **shift** 和 **pop** 表现得象在主程序里面，因此它们缺省时对 **@ARGV** 进行操作，而不是 **@_**。）

这四种块类型以下面顺序运行：

- **BEGIN**

如果在编译过程中碰到则在编译其他文件之前尽可能快地运行。

- **CHECK**

当编译完成之后，但在程序开始之前运行。（**CHECK** 可以理解为“检查点”或者“仔细检查”或者就是“停止”。）

- **INIT**

在你的程序的主流程开始执行之前运行。

- **END**

在程序执行结束之后运行。

如果你声明了多于一个这样的同名语句块，即使它们在不同的模块里，**BEGIN** 也都是在 **CHECK** 前面运行的，而 **CHECK** 也都是在 **INIT** 前面运行，以及 **INIT** 都在 **END** 前面——**END** 都是在最后，你的主程序退出的时候运行，多个 **BEGIN** 和 **INIT** 以声明的顺序运行（**FIFO**），而 **CHECK** 和 **END** 以相反的顺序运行（**LIFO**）。

下面的可能是最容易演示的例子：

```
#!/usr/bin/perl -l
print "start main running here";
die "main now dying here\n";
die "XXX: not reached\n";
END { print "1st END: done running" }
CHECK { print "1st CHECK : done compiling" }
INIT { print "1st INIT: started running" }
END { print "2nd END: done running" }
BEGIN { print "1st BEGIN: still compiling" }
INIT { print "2nd INIT: started running" }
BEGIN { print "2nd CHECK: done compiling" }
END { print "3rd END: done running" }
```

如果运行它，这个演示程序输出下面的结果：

```
1st BEGIN: still compiling
2nd BEGIN: still compiling
2nd CHECK: done compiling
1st CHECK: done compiling
1st INIT: started running
2nd INIT: started running
start main running here
main now dying here
3rd END: done running
2nd END: done running
1st END: done running
```

因为一个 **BEGIN** 块立即就执行了，所以它甚至可以在其他文件编译前把子过程声明，定义以及输入等抓过来。这些动作可能改变编译器对当前文件其他部分分析的结果，特别是在你输入了子过程定义的情况下。至少，声明一个子过程就把它当作一个列表操作符使用，这样就令圆括号是可选的。如果输入的子过程定义了原型，那么调用它的时候就会当作内建函数分析，甚至覆盖同名的内建函数，这样就可以给它们不同的语意。**use** 声明就是一个带有目的的 **BEGIN** 块声明。

相比之下，**END** 块是尽可能晚地执行：在你的程序退出 Perl 解释器的时候，甚至是因为一个没有捕获的 **die** 或者其他致命错误。有两种情况下会忽略 **END** 块（或者一个 **DESTROY** 方法）。如果一个程序不是退出，而是用 **exec** 从一个程序变形到另外一个程序，那么 **END** 就不会运行。一个进程被一个未捕获的信号杀死的时候也不会执行 **END** 过程。（参阅在第三十一章，用法模块，里描述的 **use sigtrap** 用法。那里面有将可捕获信号转换成例外的一个比较容易的方法。关于信号操作的通用信息，请参考第十六章，进程间通讯，里的“信号”。）想要绕开所有 **END** 处理，你可以调用 **POSIX::exit**，也就是 **kill -9, \$\$**，或者就是 **exec** 任何无伤大雅的程序，比如 Unix 系统里的 **/bin/true**。

在一个 **END** 块里面，**\$?** 包含程序 **exit** 时准备的状态。你可以在 **END** 块里修改 **\$?** 以修改程序的退出值。要小心不要碰巧用 **system** 或者反勾号运行了其它程序而改变了 **\$?**。

如果你在一个文件里有好几个 **END** 块，那么它们以定义它们的相反顺序执行。也就是说，你的程序结束的时候定义在最后的 **END** 块首先执行。如果你把 **BEGIN** 和 **END** 成对使用的话，这样的反序允许相关的 **BEGIN** 和 **END** 块按照你预期的方法嵌套，比如，如果主程序和它装载的模块都有自己的成对的 **BEGIN** 和 **END** 子过程，象下面这样：

```
BEGIN { print "main begun" }
END { print "main ended" }
use Module;
```

并且在那个模块里，定义了下面的声明：

```
BEGIN { print "module begun" }
END { print "module ended" }
```

那么主程序就知道它的 **BEGIN** 总是首先发生，而它的 **END** 总是最后使用。（不错，**BEGIN** 实际上是一个编译时的块，但类似的现象对运行时的 **INIT** 和 **END** 对身上也会发生。）如果一个文件包含另外一个文件，而且它们都有类似这样的声明的时候，这个原则是递归地正确的。这样的嵌套属性令这些块可以很好地当作包构造器和析构器来使用。每个模块都可以有它们自己的安装和删除函数，而 Perl 可以自动地调用它们。这样，程序员就不用总是记住是否用了某个库，是否在某时需要调用特殊的初始化或者清理代码。这些事件都由模块的声明来保证。

如果你把 **eval STRING** 当作一个从解释器到编译器的回调函数，那么你可以把 **BEGIN** 看作从编译

器到解释器的前进函数。它们两个都是暂时把当前正在处理的事情挂起来然后切换操作的模式。如果我们说一个 **BEGIN** 块是尽可能早地执行，我们的意思就是说它在完成定义以后马上就执行，甚至是在包含它的文件的其他部分分析之前。因此 **BEGIN** 块是在编译时执行的，而不是在运行时。一旦一个 **BEGIN** 块开始运行，那么它马上就取消定义并且它使用的任何代码都返回到 **Perl** 的内存池中。你不能把 **BEGIN** 当作一个子过程调用，（你试了也没有用。）因为当它存在那里的时候，它就已经运行（消失）了。

和 **BEGIN** 块类似，**INIT** 块都是在 **Perl** 运行时开始执行之前运行的，顺序是“先进先出”（**FIFO**）。比如，在 **perlcc** 里讲到的代码生成器使用 **INIT** 块初始化并解析指向 **XSUB** 的指针。**INIT** 块实际上和 **BEGIN** 块一样，只不过是它们让程序员分开了必须在编译阶段发生的构造和必须在运行阶段发生的构造。如果你直接运行一个脚本，这两者没什么大区别，因为每次运行编译器都要运行；但是如果编译和执行是分开的，那么这样的区别就可能是关键的。编译器可能只调用一次，而生成的可执行文件可以运行多次。

和 **END** 块类似，**CHECK** 块在 **Perl** 编译阶段完成之后而在开始运行阶段之前运行。顺序是 **LIFO**。**CHECK** 块可以用于“退出”编译器，就好象 **END** 块可以用于退出你的程序一样。特别是后端都把 **CHECK** 块当作挂钩使用，这样它们可以调用相应的代码生成器。它们需要做的只是把一个 **CHECK** 块放到它们自己的模块里，而这个 **CHECK** 块在合适的时刻就会运行，这样你就不用把 **CHECK** 写到你的程序里。因此，你很少需要写自己的 **CHECK** 块，除非你正在写这样的模块。

把上面的内容都放到一起，表 18-1 列出了各种构造，列出了它们何时编译或者运行 “...” 代表的代码。

表18-1 何时发生何事

Block	Compiles	Traps	Runs	Traps	Call
or	During	Compile	During	Run	Trigger
Expression	Phase	Errors	Phase	Errors	Policy
use ...	C	No	C	No	Now
no ...	C	No	C	No	Now
BEGIN {...}	C	No	C	No	Now
CHECK {...}	C	No	C	No	Late
INIT {...}	C	No	R	No	Early
END {...}	C	No	R	No	Late
eval {...}	C	No	R	Yes	Inline
eval "..."	R	Yes	R	Yes	Inline
foo(...)	C	No	R	No	Inline
sub foo {...}	C	No	R	No	Call anytime
eval "sub {...}"	R	Yes	R	No	Call later
s/pat/.../e	C	No	R	No	Inline
s/pat/"..."/ee	R	Yes	R	Yes	Inline

现在你知道结果了，我们希望你更有信心的编辑和使用的 **Perl** 程序。

Revision: r1.3 - 09 Sep 2005 - 10:37 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > [PerlCompiling](#)

版权 © 1999-2006 归这里所有作者。 [PostgreSQL](#) 的中文文档版权归何伟平所有。

向为这里贡献想法,文章的人致敬 [PostgreSQL 中文网](#)
[反馈意见](#)