

34 | 运行时优化：即时编译的原理和作用

2019-11-11 宫文学 来自北京

《编译原理之美》



前面所讲的编译过程，都存在一个明确的编译期，编译成可执行文件后，再执行，这种编译方式叫做**提前编译（AOT）**。与之对应的，另一个编译方式是**即时编译（JIT）**，也就是，在需要运行某段代码的时候，再去编译。其实，Java、JavaScript 等语言，都是通过即时编译来提高性能的。

那么问题来了：

什么时候用 AOT，什么时候用 JIT 呢？

在讲运行期原理时，我提到程序编译后，会生成二进制的可执行文件，加载到内存以后，目标代码会放到代码区，然后开始执行。那么即时编译时，对应的过程是什么？目标代码会存放到哪里呢？

在什么情况下，我们可以利用即时编译技术，获得运行时的优化效果，又该如何实现呢？

本节课，我会带你掌握，即时编译技术的特点，和它的实现机理，并通过一个实际的案例，探讨如何充分利用即时编译技术，让系统获得更好的优化。这样一来，你对即时编译技术的理解会更透彻，也会更清楚怎样利用即时编译技术，来优化自己的软件。

首先，来了解一下，即时编译的特点和原理。

了解即时编译的特点及原理

根据计算机程序运行的机制，我们把，不需要编译成机器码的执行方式，**叫做解释执行**。解释执行，通常都会基于虚拟机来实现，比如，基于栈的虚拟机，和基于寄存器的虚拟机（在 [🔗 32 讲](#)中，我带你了解过）。

与解释执行对应的，是编译成目标代码，直接在 CPU 上运行。而根据编译时机的不同，又分为 AOT 和 JIT。**那么，JIT 的特点，和使用场景是什么呢？**

一般来说，一个稍微大点儿的程序，静态编译一次，花费的时间很长，而这个等待时间是很煎熬的。如果采用 JIT 机制，你的程序就可以像，解释执行的程序一样，马上运行，得到反馈结果。

其次，JIT 能保证代码的可移植性。在某些场景下，我们没法提前知道，程序运行的目标机器，所以，也就没有办法提前编译。Java 语言，先编译成字节码，到了具体运行的平台上，再即时编译成，目标代码来运行，这种机制，使 Java 程序具有很好的可移植性。

再比如，很多程序会用到 GPU 的功能，加速图像的渲染，但针对不同的 GPU，需要编译成不同的目标代码，这里也会用到即时编译功能。

最后，JIT 是编译成机器码的，在这个过程中，可以进行深度的优化，因此程序的性能要比解释执行高很多。

这样看来，JIT 非常有优势。

而从实际应用来看，原来一些解释执行的语言，后来也采用 JIT 技术，优化其运行机制，在保留即时运行、可移植的优点的同时，又提升了运行效率，**JavaScript 就是其中的典型代表**。

基于谷歌推出的 V8 开源项目，JavaScript 的性能获得了极大的提升，使基于 Web 的前端应用的体验，越来越好，这其中就有 JIT 的功劳。

而且据我了解，R 语言也加入了 JIT 功能，从而提升了性能；Python 的数据计算模块 numba 也采用了 JIT。

在了解 JIT 的特点，和使用场景之后，你有必要对 JIT 和 AOT 在技术上的不同之处有所了解，以便掌握 JIT 的技术原理。

静态编译的时候，首先要把程序，编译成二进制目标文件，再链接形成可执行文件，然后加载到内存中运行。JIT 也需要编译成二进制的目标代码，但是目标代码的加载和链接过程，就不太一样了。

首先说说目标代码的加载。

在静态编译的情况下，应用程序会被操作系统加载，目标代码被放到了代码区。从安全角度出发，操作系统给每个内存区域，设置了不同的权限，代码区具备可执行权限，所以我们才能运行程序。

在 JIT 的情况下，我们需要为这种动态生成的目标代码，申请内存，并给内存设置可执行权限。我写个实际的 C 语言程序，让你直观地理解一下这个过程。

我们在一个数组里，存一段小小的机器码，只有 9 个字节。这段代码的功能，相当于一个 C 语言函数的功能，也就是把输入的参数加上 2，并返回。

 复制代码

```
1  /*
2   * 机器码，对应下面函数的功能：
3   * int foo(int a){
4   *     return a + 2;
5   * }
6   */
7  uint8_t machine_code[] = {
8      0x55, 0x48, 0x89, 0xe5,
9      0x8d, 0x47, 0x02, 0x5d, 0xc3
10 }
```

你可能问了：你怎么知道这个函数，对应的机器码是这 9 个字节呢？

这不难，你把 `foo.c` 编译成目标文件，然后查看里面的机器码就行了。

[复制代码](#)

```
1 clang -c -O2 foo.c -o foo.o
2 或者
3 gcc -c -O2 foo.c -o foo.o
4
5 objdump -d foo.o
```

`objdump` 命令，能够反编译一个目标文件，并把机器码，和对应的汇编码都显示出来：

```
richard@mac 33-jit $ objdump -d foo.o

foo.o:  file format Mach-O 64-bit x86-64

Disassembly of section __TEXT,__text:
foo:
      0:      55      pushq   %rbp
      1:      48 89 e5      movq    %rsp, %rbp
      4:      8d 47 02      leal    2(%rdi), %eax
      7:      5d      popq    %rbp
      8:      c3      retq
```

另外，用 “`hexdump foo.o`” 命令显示这个二进制文件的内容，也能找到这 9 个字节（图中橙色框中的内容）。


```

0000180 60 02 00 00 08 00 00 00 0b 00 00 00 50 00 00 00
0000190 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00
00001a0 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00001d0 00 00 00 00 00 00 00 00 00 55 48 89 e5 8d 47 02 5d
00001e0 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001f0 09 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00
0000200 00 00 00 00 00 00 00 00 14 00 00 00 00 00 00 00
0000210 01 7a 52 00 01 78 10 01 10 0c 07 08 90 01 00 00
0000220 24 00 00 00 1c 00 00 00 b0 ff ff ff ff ff ff
0000230 09 00 00 00 00 00 00 00 00 41 0e 10 86 02 43 0d
0000240 06 00 00 00 00 00 00 00 00 00 00 00 01 00 00 06
0000250 01 00 00 00 0f 01 00 00 00 00 00 00 00 00 00 00
0000260 00 5f 66 75 6e 31 00 00
0000268

```

这里多说一句，如果你喜欢深入钻研的话，那么我建议你研究一下，如何从汇编代码生成机器码（实际上就是研究汇编器的原理）。比如，第一行汇编指令“pushq %rbp”，为什么对应的机器码，只有一个字节？如果指令一个字节，操作数一个字节，应该两个字节才对啊？

其实，你阅读 Intel 的手册之后，就会知道这个机器码为什么这么设计。因为它要尽量节省空间，所以实际上，很多指令和操作码，会被压缩进，一个字节中去表示。在 32 讲中，研究字节码的设计时，你应该发现了这个规律。这些设计思路都是相通的，如果你要设计自己的字节码，也可以借鉴这些思想。

说回我们的话题，既然已经有了机器码，那我们接着再做下面几步：

用 mmap 申请 9 个字节的一块内存。用这个函数（不是 malloc 函数）的好处是，可以指定内存的权限。我们先让它的权限是可读可写的。

然后用 memcpy 函数，把刚才那 9 个字节的数组，拷贝到，所申请的内存块中。

用 mprotect 函数，把内存的权限，修改为可读和可执行。

再接着，用一个 int(*) (int) 型的函数指针，指向这块内存的起始地址，也就是说，该函数有一个 int 型参数，返回值也是 int。


最后，通过这个函数指针，调用这段机器码，比如 `fun(1)`。你打印它的值，看看是否符合预期。

完整的代码在 [jit.cpp](#) 里。

借这个例子，你可能会知道，通过内存溢出来攻击计算机是怎么回事了。因为只要一块内存是可执行的，你又通过程序写了一些代码进去，就可以攻击系统。是不是有点儿黑客的感觉？所以在 `jit.cpp` 里，我们其实很小心地，把内存地址的写权限去掉了。

如果你愿意深究，我建议你，再看一眼 `objdump` 打印的汇编码。你会发现，其中开头为 0、1 和 7 的三行是没有用的。根据你之前学过的汇编知识，你应该知道，这三行实际是保存栈指针、设置新的栈指针的。但这个例子中，都是用寄存器来操作的，没用到栈，所以这三行代码对应的机器码可以省掉。

最后，只用 4 个字节的机器码也能完成同样的功能：

 复制代码

```
1 //省略了三行汇编代码的机器码：
2 uint8_t machine_code1[] = {
3     0x8d, 0x47, 0x02, 0xc3
4 };
```

现在，你应该清楚了，动态生成的代码，是如何加载到内存，然后执行了吧？

不过，刚刚这个函数比较简单，只做了一点儿算术计算。通常情况下，你的程序会比较复杂，往往在一个函数里，要调用另一个函数。比如，需要在 `foo` 函数里，调用 `bar` 函数。这个 `bar` 函数可能是你自己写的，也可能是一个库函数。执行的时候，需要能从 `foo` 函数，跳转到 `bar` 函数的地址，执行完毕以后再跳转回来。**那么，你要如何确定 `bar` 函数的地址呢？**

这就涉及目标代码的链接问题了。

原来，编译器生成的二进制对象，都是可重定位的。在静态编译的时候，链接程序最重要的工作，就是重定位（Relocatable），各个符号的地址，包括全局变量、常量的地址和函数的地

址，这样，你就可以访问这些变量，或者跳转到函数的入口。

JIT 没有静态链接的过程，但是，也可以运用同样的思路，解决地址解析的问题。你编写的程序里的所有全局变量，和函数，都会被放到一个符号表里，在符号表里记录下它们的地址。这样，引用它们的函数就知道正确的地址了。

更进一步，你写的函数，不仅可以引用你自己编写的，程序中的对象，还可以访问共享库中的对象。比如，很多程序，都会共享 libc 库中的标准功能，这个库的源代码超过了几百万行，像打印输出这样的基础功能，都可以用这个库来实现。

这时候，你可以用到动态链接技术。动态链接技术运用得很普遍，它是在应用程序加载的时候，来做地址的重定位。

动态链接，通常会采用“位置无关代码（PIC）”的技术，使动态库，映射进每个应用程序的空间时，其地址看上去都不同。这样一来，可以让动态库被很多应用共享，从而节省内存空间，而且可以提升安全性。因为固定的地址，有利于恶意的程序，去攻击共享库中的代码，从而危及整个系统。

到目前为止，你已经了解了实现 JIT 的两个关键技术：

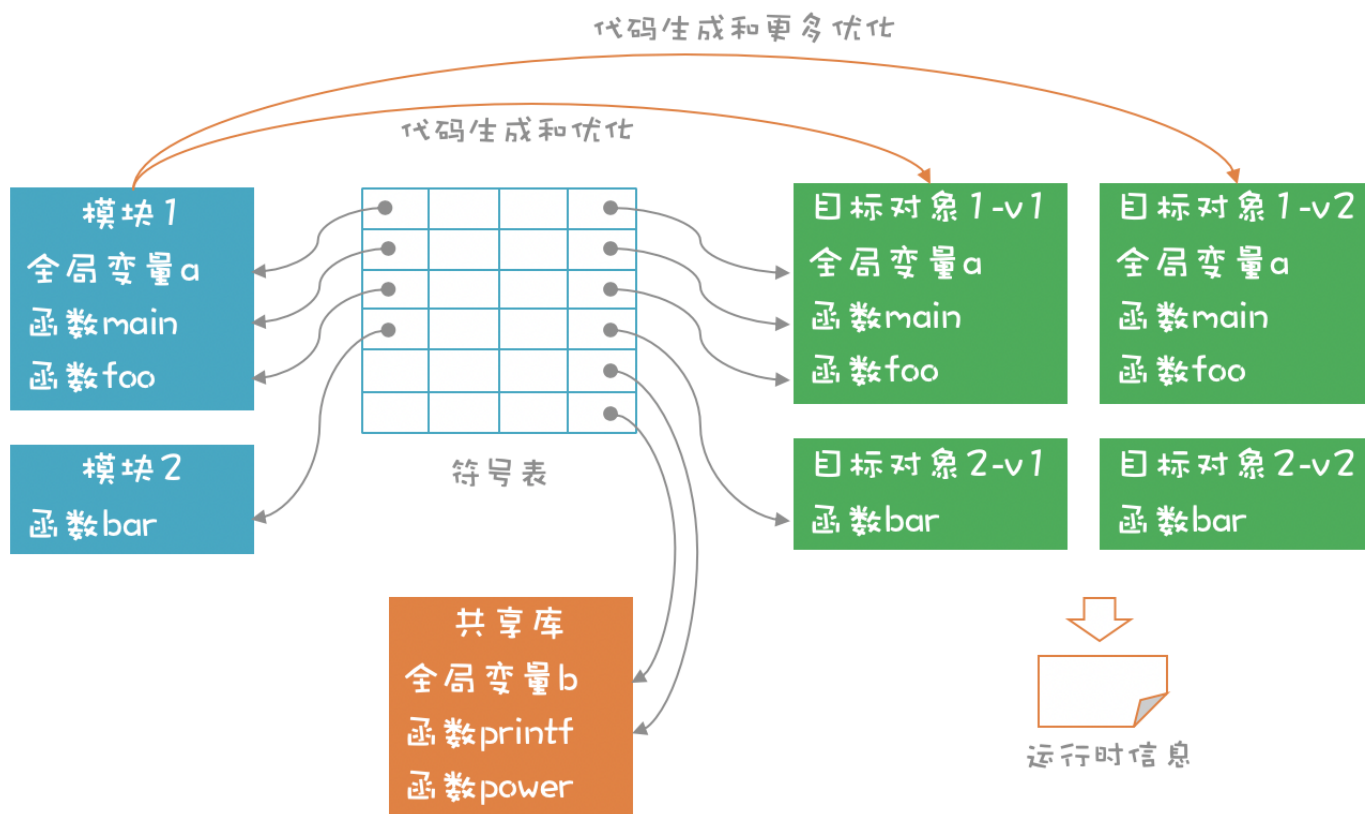
让代码动态加载和执行。

访问自己写的程序和共享库中的对象。

它们是 JIT 的核心。至于代码优化和目标代码生成，与静态编译是一样的。了解这些内容之后，你应该更加理解 Java、JavaScript 等语言，即时编译运行的过程了吧？

当然，LLVM 对即时编译提供了很好的支持，**它大致的机制是这样的：**

我们编写的任何模块 (Module)，都以内存 IR 的形式存在，LLVM 会把模块中的符号，都统一保存到符号表中。当程序要调用模块的方法时，这个模块就会被即时编译，形成可重定位的目标对象，并被调用执行。动态链接库中的方法（如 printf）也能够被重定位并调用。



在第一次编译时，你可以让 LLVM，仅运行少量的优化算法，这样编译速度比较快，马上就可以运行。而对于被多次调用的函数，你可以让 LLVM 执行更多的优化算法，生成更优化版本的目标代码。而运行时所收集的某些信息，可以作为某些优化算法的输入，像 Java 和 JavaScript 都带有这种多次生成目标代码的功能。

带你了解 JIT 的原理之后，接下来，我再通过一个案例，让你对 JIT 的作用有更加直观的认识。

用 JIT 提升系统性能

著名的开源数据库软件，PostgreSQL，你可能听说过。它的历史比 MySQL 久，功能也比 MySQL 多一些。在最近的版本中，它添加了基于 LLVM 的，即时编译功能，性能大大提高。

看一下下面的 SQL 语句：

```
1 select count(*) from table_name where (x + y) > 100
```

复制代码

这个语句的意思是：针对某个表，统计一下字段 x 和 y 的和大于 100 的记录有多少条。这个 SQL 在运行时，需要遍历所有的行，并对每一行，计算 $(x + y) > 100$ 这个表达式的值。如果有 1000 万行，这个表达式就要被执行 1000 万次。

PostgreSQL 的团队发现，直接基于 AST 或者某种 IR，解释执行这个表达式的话，所用的时间，占到了处理一行记录所需时间的 56%。而基于 LLVM 实现 JIT 以后，所用的时间只占到 6%，性能整整提高了一倍。

在这里，我联系 [31 讲](#) 内存计算的内容，**带你拓展一下**。上面的需求，是典型的基于列进行汇总计算的需求。如果对代码进行向量化处理，再保证数据的局部性，针对这个需求，性能还可以提升很多倍。

再说回来。除了针对表达式的计算进行优化，PostgreSQL 的团队还利用 LLVM 的 JIT 功能，实现了其他的优化。比如，编译 SQL 执行计划的时间，缩短了 5.5 倍；创建 b 树索引的时间，降低了 5%~19%。

那么 32 讲中，我提到，将一个规则引擎，编译成字节码，这样在处理大量数据时，可以提高性能。这是因为，JVM 也会针对字节码做即时编译。道理是一样的。

课程小结

对现代编程语言来说，编译期和运行期的界限，越来越模糊了，解释型语言和编译型语言的区别，也越来越模糊了。即时编译技术可以生成，最满足你需求的目标代码。那么通过今天的内容，我强调这样几点：

1. 为了实现 JIT 功能，你可以动态申请内存，加载目标代码到内存，并赋予内存可执行的权限。在这个过程中，你要注意安全因素。比如，向内存写完代码以后，要取消写权限。
2. 可重定位的目标代码，加上动态链接技术，让 JIT 产生的代码可以互相调用，以及调用共享库的功能。
3. JIT 技术可以让数据库这类基础软件，获得性能上的提升，如果你有志参与研发这类软件，掌握 JIT 技术会给你加分很多。

一课一思

你参与开发的软件，特别是支持 DSL 的软件，是否可以用 JIT 技术提升性能？欢迎在留言区分享你的观点。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (3)



刘強

2019-11-11

感觉越来越像lisp的“代码即数据”了。在内存中运行时编译（生成）代码（数据），然后执行。编译期和解释期界限越来越模糊，怪不得有人说所有的编程语言都在向lisp进化。

作者回复：看得出你是Lisp粉。

我曾经相当迷恋Lisp。用过一段时间的Common Lisp，甚至试图向同事推广。可是后来发现还是有点小众，解决一些技术问题资料也太少，后来就搁下了。但在设计我手头的语言的时候，还是经常会想想Lisp的特性。

我在后面关于元编程的一讲，会探讨一下Lisp。



👍 8



无名氏

2022-04-16

老师，请教下，应用进程的静态编译能使用的共享库吗，如果能使用那也要编译进符号表？动态编译使用共享库，共享库已什么形式存在，在一个独立的内存中，别的程序进行引用？



Cryhard

2019-12-31

看来postgresql 11的生产级别试用可以提上日程了。似乎需要使用llvm编译一下？

作者回复：官方应该有预编译好的版本吧？

