

# 35 | libffi：动态调用和定义 C 函数

戴铭 2019-05-30



00:00

讲述：冯永奇

大小：13.60M

14:50

你好，我是戴铭。

在 iOS 开发中，我们可以使用 Runtime 接口动态地调用 Objective-C 方法，但是却无法动态调用 C 的函数。那么，我们怎么才能动态地调用 C 语言函数呢？

C 语言编译后，在可执行文件里会有原函数名信息，我们可以通过函数名字符串来找到函数的地址。现在，我们只要能够通过函数名找到函数地址，就能够实现动态地去调用 C 语言函数。

而在动态链接器中，有一个接口 dlsym() 可以通过函数名字符串拿到函数地址，如果所有 C 函数的参数类型和数量都一样，而且返回类型也一样，那么我们使用 dlsym() 就能实现动态地调用 C 函数。

但是，在实际项目中，函数的参数定义不可能都一样，返回类型也不会都是 void 或者 int 类型。所以，dlsym() 这条路走不通。那么，还有什么办法可以实现动态地调用 C 函数呢？

## 如何动态地调用 C 函数？

要想动态地调用 C 函数，你需要先了解函数底层是怎么调用的。

高级编程语言的函数在调用时，需要约定好参数的传递顺序、传递方式，栈维护的方式，名字修饰。这种函数调用者和被调用者对函数如何调用的约定，就叫作调用惯例（Calling Convention）。高级语言编译时，会生成遵循调用惯例的代码。

不同 CPU 架构的调用惯例不一样，比如 64 位机器的寄存器多些、传递参数快些，所以参数传递会优先采用寄存器传递，当参数数量超出寄存器数量后才会使用栈传递。

所以，编译时需要按照调用惯例针对不同 CPU 架构编译，生成汇编代码，确定好栈和寄存器。如果少了编译过程，直接在运行时去动态地调用函数，就需要先生成动态调用相应寄存器和栈状态的汇编指令。而要达到事先生成相应寄存器和栈的目的，就不能使用遵循调用惯例的高级编程语言，而需要使用汇编语言。

Objective-C 的函数调用采用的是发送消息的方式，使用的是 objc\_msgSend 函数。objc\_msgSend 函数就是使用汇编语言编写的，其结构分为序言准备（Prologue）、函数体（Body）、结束收尾（Epilogue）三部分。

序言准备部分的作用是会保存之前程序执行的状态，还会将输入的参数保存到寄存器和栈上。这样，objc\_msgSend 就能够先将未知的参数保存到寄存器和栈上，然后在函数体执行自身指令或者跳转其他函数，最后在结束收尾部分恢复寄存器，回到调用函数之前的状态。

得益于序言准备部分可以事先准备好寄存器和栈，objc\_msgSend 可以做到函数调用无需通过编译生成汇编代码来遵循调用惯例，进而使得 Objective-C 具备了动态调用函数的能力。

但是，不同的 CPU 架构，在编译时会执行不同的 objc\_msgSend 函数，而且 objc\_msgSend 函数无法直接调用 C 函数，所以想要实现动态地调用 C 函数就需要使用另一个用汇编语言编写的库 libffi。

那么，libffi 是什么呢，又怎么使用 libffi 来动态地调用 C 函数？接下来，我就和你分析一下这两个问题应该如何解决。

## libffi 原理分析

libffi 中 ffi 的全称是 Foreign Function Interface（外部函数接口），提供最底层的接口，在不确定参数个数和类型的情况下，根据相应规则，完成所需数据的准备，生成相应汇编指令的代码来完成函数调用。

libffi 还提供了可移植的高级语言接口，可以不使用函数签名间接调用 C 函数。比如，脚本语言 Python 在运行时会使用 libffi 高级语言的接口去调用 C 函数。libffi 的作用类似于一个动态的编译器，在运行时就能够完成编译时所做的调用惯例函数调用代码生成。

libffi 通过调用 ffi\_call（函数调用）来进行函数调用，ffi\_call 的输入是 ffi\_cif（模板）、函数指针、参数地址。其中，ffi\_cif 由 ffi\_type（参数类型）和参数个数生成，也可以是 ffi\_closure（闭包）。

libffi 是开源的，代码在 [GitHub](#) 上。接下来，我将结合 libffi 中的关键代码，和你详细说一下 ffi\_call 调用函数的过程。这样，可以帮助你更好地了解 libffi 的原理。

首先，我们来看看 ffi\_type。

### ffi\_type（参数类型）

ffi\_type 的作用是，描述 C 语言的基本类型，比如 uint32、void \*、struct 等，定义如下：

```
1 typedef struct _ffi_type
2 {
3     size_t size; // 所占大小
4     unsigned short alignment; // 对齐大小
5     unsigned short type; // 标记类型的数字
6     struct _ffi_type **elements; // 结构体中的元素
7 } ffi_type;
```

其中，size 表述该类型所占的大小，alignment 表示该类型的对齐大小，type 表示标记类型的数字，element 表示结构体的元素。

当类型是 uint32 时，size 的值是 4，alignment 也是 4，type 的值是 9，elements 是空。

### ffi\_cif（模板）

ffi\_cif 由参数类型（ffi\_type）和参数个数生成，定义如下：

```
1 typedef struct {
2     ffi_abi abi; // 不同 CPU 架构下的 ABI，一般设置为 FFI_DEFAULT_ABI
3     unsigned nargs; // 参数个数
4     ffi_type **arg_types; // 参数类型
5     ffi_type *rtype; // 返回类型
6     unsigned bytes; // 参数所占空间大小、16 的倍数
7     unsigned flags; // 返回类型是结构体时要做的标记
8 #ifdef FFI_EXTRA_CIF_FIELDS
9     FFI_EXTRA_CIF_FIELDS;
10 #endif
11 } ffi_cif;
```

如代码所示，ffi\_cif 包含了函数调用时需要的一些信息。

abi 表示的是不同 CPU 架构下的 ABI，一般设置为 FFI\_DEFAULT\_ABI；在移动设备上 CPU 架构是 ARM64 时，FFI\_DEFAULT\_ABI 就是 FFI\_SYSV；使用苹果公司笔记本 CPU 架构是 X86\_DARWIN 时，FFI\_DEFAULT\_ABI 就是 FFI\_UNIX64。

nargs 表示输入参数的个数。arg\_types 表示参数的类型，比如 ffi\_type\_uint32。rtype 表示返回类型，如果返回类型是结构体，字段 flags 需要设置数值作为标记，以便在 ffi\_prep\_cif\_machdep 函数中处理，如果返回的不是结构体，flags 不做标记。

bytes 表示输入参数所占空间的大小，是 16 的倍数。

ffi\_cif 是由 ffi\_prep\_cif 函数生成的，而 ffi\_prep\_cif 实际上调用的又是 ffi\_prep\_cif\_core 函数。

了解 ffi\_prep\_cif\_core 就能够知道 ffi\_cif 是怎么生成的。接下来，我继续跟你说说 ffi\_prep\_cif\_core 里是怎么生成 ffi\_cif 的。ffi\_prep\_cif\_core 函数会先初始化返回类型，然后对返回类型使用 ffi\_type\_test 进行完整性检查，为返回类型留出空间。

接着，使用 initialize\_aggregate 函数初始化栈，对参数类型进行完整性检查，对栈进行填充，通过 ffi\_prep\_cif\_machdep 函数执行 ffi\_cif 平台相关处理。具体实现代码，你可以点击[这个链接](#)查看，其在文件路径是 libffi/src/prepare\_c.c。

之所以将准备 ffi\_cif 和 ffi\_call 分开，是因为 ffi\_call 可能会调用多次参数个数、参数类型、函数指针相同，只有参数地址不同的函数。将它们分开，ffi\_call 只需要处理不同参数地址，而其他工作只需要 ffi\_cif 做一遍就行了。

接着，准备好了 ffi\_cif 后，我们就可以开始函数调用了。

### ffi\_call（函数调用）

ffi\_call 函数的主要处理都交给了 ffi\_call\_SYSV 这个汇编函数。ffi\_call\_SYSV 的实现代码，你可以点击[这个链接](#)，其在文件路径是 libffi/src/aarch64/sysv.S。

下面，我来跟你说说 **ffi\_call\_SYSV 汇编函数做了什么**。

首先，我们一起看看 ffi\_call\_SYSV 函数的定义：

```
1 extern void ffi_call_SYSV (void *stack, void *frame,
2                             void (*fn)(void), void *rvalue,
3                             int flags, void *closure);
```

可以看到，通过 ffi\_call\_SYSV 函数，我们可以得到 stack、frame、fn、rvalue、flags、closure 参数。

各参数会依次保存在参数寄存器中，参数栈 stack 在 x0 寄存器中，参数地址 frame 在 x1 寄存器中，函数指针 fn 在 x2 寄存器中，用于存放返回值的 rvalue 在 x3 里，结构体标识 flags 在 x4 寄存器中，闭包 closure 在 x5 寄存器中。

然后，我们再看看 ffi\_call\_SYSV 处理的核心代码：

```
1 // 分配 stack 和 frame
2 cfi_def_cfa(x1, 32);
3 stp x29, x30, [x1]
4 mov x29, x1
5 mov sp, x0
6 cfi_def_cfa_register(x29)
7 cfi_rel_offset (x29, 0)
8 cfi_rel_offset (x30, 8)
9
10 // 记录函数指针 fn
11 mov x9, x2          /* save fn */
12
13 // 记录返回值 rvalue
14 mov x8, x3          /* install structure return */
15 #ifdef FFI_G0_CLOSURES
16 // 记录闭包 closure
17 mov x18, x5         /* install static chain */
18 #endif
19 // 保存 rvalue 和 flags
20 stp x3, x4, [x29, #16] /* save rvalue and flags */
21
22 // 先将向量参数传到寄存器
23 tbz w4, #AARCH64_FLAG_ARG_V_BIT, 1f
24 ldp q0, q1, [sp, #0]
25 ldp q2, q3, [sp, #32]
26 ldp q4, q5, [sp, #64]
27 ldp q6, q7, [sp, #96]
28 1:
29 // 再将参数传到寄存器
30 ldp x0, x1, [sp, #16*N_V_ARG_REG + 0]
31 ldp x2, x3, [sp, #16*N_V_ARG_REG + 16]
32 ldp x4, x5, [sp, #16*N_V_ARG_REG + 32]
33 ldp x6, x7, [sp, #16*N_V_ARG_REG + 48]
34
35 // 释放上下文，留下栈里参数
36 add sp, sp, #CALL_CONTEXT_SIZE
37
38 // 调用函数指针 fn
39 blr x9
40
41 // 重新读取 rvalue 和 flags
42 ldp x3, x4, [x29, #16]
43
44 // 析构部分栈指针
45 mov sp, x29
46 cfi_def_cfa_register(x29)
47 ldp x29, x30, [sp]
48
49 // 保存返回值
50 adr x5, 0f
51 and w4, w4, #AARCH64_RET_MASK
52 add x5, x5, x4, lsl #3
53 br x5
54 0f
```

如上面代码所示，**ffi\_call\_SYSV 处理过程分为下面几步**：

第一步，ffi\_call\_SYSV 会先分配 stack 和 frame，保存记录 fn、rvalue、closure、flags。

第二步，将向量参数传到寄存器，按照参数放置规则，调整 sp 的位置，

第三步，将参数放入寄存器，存放完毕，就开始释放上下文，留下栈里的参数。

第四步，通过 blr 指令调用 x9 中的函数指针 fn，以调用函数。

第五步，调用完函数指针，就重新读取 rvalue 和 flags，析构部分栈指针。

第六步，保存返回值。

可以看出，libffi 调用函数的原理和 objc\_msgSend 的实现原理非常类似。objc\_msgSend 原理，你可以参考 Mike Ash 的“[Dissecting objc\\_msgSend on ARM64](#)”这篇文章。

这里我要多说一句，在专栏[第 2 篇文章](#)中我和你分享 App 启动速度优化时，用到了些汇编代码，有很多用户反馈看不懂这部分内容。针对这个情况，我特意在[第 11 篇答疑文章](#)中，和你分享了些汇编语言学习的方法、参考资料。如果你对上述的汇编代码感兴趣，但又感觉读起来有些吃力的话，建议你再看一下第 11 篇文章中的相关内容。

了解了 libffi 调用函数的原理后，相信你迫不及待就想在你的 iOS 工程中集成 libffi 了吧。

## 如何使用 libffi？

孙源在 GitHub 上有个[Demo](#)，已经集成了 iOS 可以用的 libffi 库，你可以将这个库集成到自己的工程中。接下来，我借用孙源这个 Demo 中的示例代码，来分别和你说说如何使用 libffi 库来调用 C 函数和定义 C 函数。代码所在文件路径是 libffi-iOS/Demo/ViewController.m。在这里，我也特别感谢孙源的这个 Demo。

## 调用 C 函数

首先，声明一个函数，实现两个整数相加：

```
1 - (int)fooWithBar:(int)bar baz:(int)baz {
2     return bar + baz;
3 }
4
```

然后，定义一个函数，使用 libffi 来调用 fooWithBar:baz 函数，也就是刚刚声明的实现两个整数相加的函数。

```
1 void testFFICall() {
2     // ffi_call 调用需要准备的模板 ffi_cif
3     ffi_cif cif;
4     // 参数类型数组，根据被调用的函数入参的类型来定
5     ffi_type *argumentTypes[] = {&ffi_type_pointer, &ffi_type_pointer, &ffi_type
6     // 通过 ffi_prep_cif 用 ffi_prep_cif_core 来设置 ffi_cif 结构体所需要的数据，包括
7     ffi_prep_cif(&cif, FFI_DEFAULT_ABI, 4, &ffi_type_pointer, argumentTypes);
8
9     Sark *sark = [Sark new];
10    SEL selector = @selector(fooWithBar:baz);
11
12    // 函数参数的设置
13    int bar = 123;
14    int baz = 456;
15    void *arguments[] = {&sark, &selector, &bar, &baz};
16
17    // 函数指针 fn
18    IMP imp = [sark methodForSelector:selector];
19    // 返回值声明
20    int retValue;
21
22    // ffi_call 所需的 ffi_cif、函数指针、返回值、函数参数都准备好，就可以通过 ffi_call
23    ffi_call(&cif, imp, &retValue, arguments);
24    NSLog(@"ffi_call: %d", retValue);
25 }
```

如上面代码所示，在 testFFIClosure 函数准备好 cif 后，会声明一个新的函数指针，这个新的函数指针会和分配的 ffi\_closure 关联，ffi\_closure 还会通过 ffi\_prep\_closure\_loc 函数关联到 cif、closure、函数实体 closureCalled。

有了这种能力，你就具备了在运行时将一个函数指针和函数实体绑定的能力，也就能够很容易地实现动态地定义一个 C 函数了。

## 小结

今天，我和你分享了 libffi 的原理，以及怎么使用 libffi 调用和定义 C 函数。

当你理解了 libffi 的原理以后，再面对语言之间运行时动态调用的问题，也就做到了心中有数。在方案选择动态调用方式时，也就能够找出更多的方案，更加得心应手。

比如，使用 Aspect 进行方法替换，如果使用不当，会有较大的风险；再比如，hook 已经被 hook 过的方法，那么之前的 hook 会失效，新的 hook 也会出错，而使用 libffi 进行 hook 不会出现这样的问题。

## 课后作业

Block 是一个 Objective-C 对象，表面看类似 C 函数，实际上却有很大不同。你可以点击[这个链接](#)查看 Block 的定义，也可以再看看 Mike Ash 的[MABlockClosure](#)库。然后，请你在留