35-libffi：动态调用和定义C函数

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：

```
typedef struct _ffi_type
{
  size_t size; // 所占大小
  unsigned short alignment; //对齐大小
  unsigned short type; // 标记类型的数字
  struct _ffi_type **elements; // 结构体中的元素
} 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：

```
typedef struct {
  ffi_abi abi; // 不同 CPU 架构下的 ABI，一般设置为 FFI_DEFAULT_ABI
  unsigned nargs; // 参数个数
  ffi_type **arg_types; // 参数类型
  ffi_type *rtype; // 返回值类型
  unsigned bytes; // 参数所占空间大小，16的倍数
  unsigned flags; // 返回类型是结构体时要做的标记
#ifdef FFI_EXTRA_CIF_FIELDS
  FFI_EXTRA_CIF_FIELDS;
#endif
} 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，其所在文件路径是 libffi/src/prep_cif.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，其所在文件路径是 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：

```
extern void ffi_call_SYSV (void *stack, void *frame,
                  void (*fn)(void), void *rvalue,
                  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：

```
    //分配 stack 和 frame
    cfi_def_cfa(x1, 32);
    stp x29, x30, [x1]
    mov x29, x1
    mov sp, x0
    cfi_def_cfa_register(x29)
    cfi_rel_offset (x29, 0)
    cfi_rel_offset (x30, 8)

    // 记录函数指针 fn
    mov x9, x2          /* save fn */

    // 记录返回值 rvalue
    mov x8, x3          /* install structure return */
#ifdef FFI_GO_CLOSURES
    // 记录闭包 closure
    mov x18, x5         /* install static chain */
#endif
    // 保存 rvalue 和 flags
    stp x3, x4, [x29, #16]  /* save rvalue and flags */

    //先将向量参数传到寄存器
    tbz w4, #AARCH64_FLAG_ARG_V_BIT, 1f
    ldp     q0, q1, [sp, #0]
    ldp     q2, q3, [sp, #32]
    ldp     q4, q5, [sp, #64]
    ldp     q6, q7, [sp, #96]
1:
    // 再将参数传到寄存器
    ldp     x0, x1, [sp, #16*N_V_ARG_REG + 0]
    ldp     x2, x3, [sp, #16*N_V_ARG_REG + 16]
    ldp     x4, x5, [sp, #16*N_V_ARG_REG + 32]
    ldp     x6, x7, [sp, #16*N_V_ARG_REG + 48]

    //释放上下文，留下栈里参数
    add sp, sp, #CALL_CONTEXT_SIZE

    // 调用函数指针 fn
    blr     x9

    // 重新读取 rvalue 和 flags
    ldp x3, x4, [x29, #16]

    // 析构部分栈指针
    mov     sp, x29
    cfi_def_cfa_register (sp)
    ldp     x29, x30, [x29]

    // 保存返回值
    adr x5, 0f
    and w4, w4, #AARCH64_RET_MASK
    add x5, x5, x4, lsl #3
    br  x5
```

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：

```
- (int)fooWithBar:(int)bar baz:(int)baz {
    return bar + baz;
}
```

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。

```
void testFFICall() {
    // ffi_call 调用需要准备的模板 ffi_cif
    ffi_cif cif;
    // 参数类型指针数组，根据被调用的函数入参的类型来定
```

```
    ffi_type *argumentTypes[] = {&ffi_type_pointer, &ffi_type_pointer, &ffi_type_sint32, &ffi_type_sint32};
    // 通过 ffi_prep_cif 内 ffi_prep_cif_core 来设置 ffi_cif 结构体所需要的数据，包括 ABI、参数个数、参数类型等。
    ffi_prep_cif(&cif, FFI_DEFAULT_ABI, 4, &ffi_type_pointer, argumentTypes);

    Sark *sark = [Sark new];
    SEL selector = @selector(fooWithBar:baz:);

    // 函数参数的设置
    int bar = 123;
    int baz = 456;
    void *arguments[] = {&sark, &selector, &bar, &baz};

    // 函数指针 fn
    IMP imp = [sark methodForSelector:selector];
    // 返回值声明
    int retValue;

    // 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 进行函数调用了
    ffi_call(&cif, imp, &retValue, arguments);
    NSLog(@"ffi_call: %d", retValue);
}
```

如上面代码所示，先将 ffi_call 所需要的 ffi_cif 通过 ffi_prep_cif 函数准备好，然后设置好参数，通过 Runtime 接口获取 fooWithBar:baz 方法的函数指针 imp，最后就可以通过 ffi_call 进行函数调用了。

在这个例子中，函数指针是使用 Objective-C 的 Runtime 得到的。如果是 C 语言函数，你就可以通过 dlsym 函数获得。dlsym 获得函数指针示例如下：

```
// 计算矩形面积
int rectangleArea(int length, int width) {
    printf("Rectangle length is %d, and with is %d, so area is %d \n", length, width, length * width);
    return length * width;
}

void run() {
    // dlsym 返回 rectangleArea 函数指针
    void *dlsymFuncPtr = dlsym(RTLD_DEFAULT, "rectangleArea");
}
```

如上代码所示，dlsym 根据计算矩形面积的函数 rectangleArea 的函数名，返回 rectangleArea 函数指针给 dlsymFuncPtr。

无论是 Runtime 获取的函数指针还是 dlsym 获取的函数指针都可以在运行时去完成，接着使用 libffi 在运行时处理好参数。这样，就能够实现运行时动态地调用 C 函数了。

接下来，我再跟你说下如何使用 libffi 定义 C 函数。

定义 C 函数

首先，声明一个两数相乘的函数。

```
void closureCalled(ffi_cif *cif, void *ret, void **args, void *userdata) {
    int bar = *((int *)args[2]);
    int baz = *((int *)args[3]);
    *((int *)ret) = bar * baz;
}
```

然后，再写个函数，用来定义 C 函数。

```
void testFFIClosure() {
    ffi_cif cif;
    ffi_type *argumentTypes[] = {&ffi_type_pointer, &ffi_type_pointer, &ffi_type_sint32, &ffi_type_sint32};
    // 准备模板 cif
    ffi_prep_cif(&cif, FFI_DEFAULT_ABI, 4, &ffi_type_pointer, argumentTypes);

    // 声明一个新的函数指针
    IMP newIMP;

    // 分配一个 closure 关联新声明的函数指针
    ffi_closure *closure = ffi_closure_alloc(sizeof(ffi_closure), (void *)&newIMP);

    // ffi_closure 关联 cif、closure、函数实体 closureCalled
    ffi_prep_closure_loc(closure, &cif, closureCalled, NULL, NULL);

    // 使用 Runtime 接口动态地将 fooWithBar:baz 方法绑定到 closureCalled 函数指针上
    Method method = class_getInstanceMethod([Sark class], @selector(fooWithBar:baz:));
    method_setImplementation(method, newIMP);

    // after hook
    Sark *sark = [Sark new];
    int ret = [sark fooWithBar:123 baz:456];
    NSLog(@"ffi_closure: %d", ret);
}
```

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，请你在留言区说说如何通过 libffi 调用 Block。

感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎把它分享给更多的朋友一起阅读。

精选留言：

- opooc 2019-05-30 17:09:04
  没有汇编基础，看起来好痛苦。

- mersa 2019-05-30 10:25:02
  这个库可以用在线上审核么