Mad Mark's Blog

剖析ARM64下的objc_msgSend

2017-08-01 | iOS

建议结合objc_msgSend源码来阅读本文。在了解 `objc_msgSend` 的原理的同时，也可作为ARM64汇编的入门。

本文结合原文评论区Greg Parker的评论略做修改。

原文： Dissecting objc_msgSend on ARM64

原文作者： Mike Ash

概述

每一个OC对象有一个类，每一个OC类都有一个方法列表。每一个方法都有一个selector，一个指向方法实现的函数指针，以及一些元数据。 `objc_msgSend` 的工作就是传入对象和selector，查找相应方法的函数指针，然后跳到函数指针所指向的位置。

查找方法的过程可能是非常复杂的。如果在一个类里没有找到这个方法，那么它会继续到superclass里去查找。如果在所有的superclass中都没有找到，就会调用运行时的消息转发代码。当一个类第一次收到消息时，他会去调用类的 `+initialize` 方法。

通常查找一个方法必须是迅速的，因为每次消息的调用都需要有这个过程。这就和复杂的查找过程有冲突了，复杂但是要快。

OC解决这个冲突的方案是做方法缓存。每一个类有一个cache，用于存储方法的selectors和函数指针，也就是所谓的IMP。他们被组成一个哈希表，所以查找的时候是非常快的。当查找一个方法时，运行时首先询问cache。如果cache里没有这个方法，后续就会有一个缓慢而又复杂的过程，最后会把找到的结果放到cache里，这样下次查找该方法的时候就会很快了。

`objc_msgSend` 是用汇编写的。有两个原因：一是因为在C语言中不可能通过写一个函数来保留未知的参数并且跳转到一个任意的函数指针。C语言没有满足做这件事情的必要特性。另一个原因是 `objc_msgSend` 必须够快。

当然，谁都不会想要用汇编写下整个复杂的消息查找过程。这没必要。消息发送的代码可以被分为两部分：`objc_msgSend` 中有一个快速路径，是用汇编写的，还有一个慢速的路径，是用C实现的。汇编部分主要实现的是在缓存中查找方法，并且如果找到的话就跳转过去的一个过程。如果在缓存中没有找到方法的实现，就会调用C的代码来处理后续的事情。

因此， `objc_msgSend` 主要有以下几个步骤：

1. 获取传入的对象的类
2. 获取这个类的方法缓存
3. 通过传入的selector，在缓存中查找方法
4. 如果缓存中没有，调用C代码
5. 跳到这个方法的IMP

让我们来看看它是如何完成这些工作的。

逐条指令分析

`objc_msgSend` 根据不同的情况，会有不同的处理路径。它有部分特殊的代码来处理类似发送消息给 `nil` ，`tagged pointer` ，以及哈希表碰撞。我会从最常见的正常情况开始讲解：发送消息给一个非 `nil` 、非 `tagged pointer` 对象，并在消息缓存中找到对应的实现，而不需要额外的扫描等操作的一个过程。 描述完正常情况后，我们将会回来再看一下其他的一些分支情况。

我会罗列每条或者每组指令，并描述它做了些什么，为什么这么做。请注意我将会在罗列出来的指令**下面**做描述。

每条指令前都会有一个相对函数开始处的偏移量。这可以方便你辨识跳转到哪个目标代码。

ARM64架构下有31个通用寄存器，每个都是64位宽的。他们被标记为x0~x30。同样也有可能使用 `w0` 到 `w30` 来访问寄存器的低32位。寄存器x0~x7被用于函数入参的前8个参数。这就表示 `objc_msgSend` 收到的 `self` 参数是保存在 `x0` 中，selector `_cmd` 参数在 `x1` 里。

开始吧！

```
1   0x0000 cmp     x0, #0x0
```

```
2   0x0004 b.le    0x6c
```

这里将存储在 x0 中的 self 和0做了一个带符号的比较，如果结果小于等于0，则跳转到 0x6c 。如果值等于0则说明是 nil ，所以跳转到的地方就是执行当发送消息给 nil 的情况。这里也处理了tagged pointer的情况。在ARM64上 通过设置指针的高位来指明是tagged pointer。（x86-64上是设置低位）。如果高位被设置了1，且被作为一个带符号的整型解析的时候，那么值就是负数。一般情况下 self 是正常的，不会进入这些分支。

```
1   0x0008 ldr    x13, [x0]
```

这条指令通过加载x0所指向的内存中的64位，来加载 self 的 isa 指针。因为一个对象的第一个指针就是 isa 指针。此时 x13 寄存器包含了 isa 。

```
1   0x000c and    x16, x13, #0xfffffff8
```

ARM64可以使用非指针的isa。通常isa指针指向的是对象的类，但是非指针的isa利用了备用的bit位，填充了一些其他的信息。这条汇编指令执行了一个逻辑与运算，掩盖掉了所有额外的位，把实际的指向类的指针保存在x16寄存器中。

```
1   0x0010 ldp    x10, x11, [x16, #0x10]
```

这是 objc_msgSend 中我最喜欢的指令。它把类的缓存信息加载到 x10 和 x11 中。 ldp 指令从内存中提取了两个寄存器的数据保存到前两个参数指定的寄存器中。第三个参数告诉从哪里加载数据，这里我们看到的是在 x16 寄存器中的值再偏移16，这块属于保存了持有缓存信息的类的区域。缓存的结构如下：

```
1   typedef uint32_t mask_t;
2
3   struct cache_t {
4       struct bucket_t *_buckets;
5       mask_t _mask;
6       mask_t _occupied;
7   }
```

ldp 指令执行完后， x10 包含了 _buckets 的值， x11 在它的高32位保存了 _occupied ，低32位保存了 _mask 。

_occupied 指定了哈希表中包含了多少条目，在 objc_msgSend 中不起什么作用。 _mask 很重要：它描述了哈希表的尺寸，方便用于与运算的掩码。它的值总是一个2的幂减一，用二进制的方法描述看起来就像是

`0000000001111111`，末尾是可变数量的1。通过这个值可以知道selector的查找索引，并在查找表的时候包裹着结尾。

```
1   0x0014 and    w12, w1, w11
```

这条指令用于计算传入的selector的起始哈希表的索引，selector是作为 `_cmd` 传入的。 `x1` 中包含 `_cmd`，所以w1包含了 `_cmd` 的低32位。 `w11` 包含了上面提到的 `_mask` 。这条指令将这两个值做与运算并将结果放到 `w12` 中。结果相当于是计算 `_cmd % table_size`，但是避免了开销很大的模运算。

```
1   0x0018 add    x12, x10, x12, lsl #4
```

光有索引还不够。为了从表里加载数据，我们需要一个实际的地址来加载。这个指令通过表索引加上表的指针来计算这个地址。它先将表索引向左位移4，相当于是乘以16，因为每个表的bucket是16字节。 `x12` 现在包含了第一个查找的bucket的地址。

```
1   0x001c ldp    x9, x17, [x12]
```

我们的朋友 `ldp` 又出现了。这次是从保存在 `x12` 中的指针加载，这个指针指向了查找的bucket。每个bucket包含一个selector和一个 `IMP` 。 `x9` 现在包含了当前bucket的selector， `x17` 中包含的是 `IMP` 。

```
1   0x0020 cmp    x9, x1
2   0x0024 b.ne   0x2c
```

这两条指令首先对 `x9` 中的selector和 `x1` 中的 `cmd` 做一个比较。如果头他们不相等，说明这个bucket中不包含我们正在查找的selector的条目，随后跳转到 `0x2c` 的位置，处理bucket不相等的逻辑。如果 `x9` 中的selector和我们正在查找的条目匹配，则执行接下去的指令。

```
1   0x0028 br     x17
```

这是一个无条件的跳转，跳转到 `x17`，包含了从当前bucket中加载的 `IMP` 。从这里开始，接下去就是执行目标方法的代码了， `objc_msgSend` 的快速路径到此已经结束了。所有参数寄存器不会受到干扰，原封不动的传给目标方法，就好像直接调用了目标方法一样。

当所有需要的信息都被缓存了，这条路径的代码在现代的机器设备上3纳秒之内就可以完成执行。

我们继续看一下在缓存中没有匹配到的情况。

```
1   0x002c cbz    x9, __objc_msgSend_uncached
```

`x9` 包含了从bucket加载到的selector。这条指令先是用它和0作比较，如果等于0则跳转到 `__objc_msgSend_uncached` 。这说明这是一个空的bucket，并且意味着这次查找失败了。目标方法不在缓存中，这时候会回到C代码(`__objc_msgSend_uncached`)，执行更详细的查找。否则就说明bucket不是空的，只是没有匹配，则继续查找。

```
1   0x0030 cmp    x12, x10
2   0x0034 b.eq   0x40
```

这里将 `x12` 中当前bucket的地址和 `x10` 中的，哈希表的开头做比较。如果他们匹配，则跳转到查找到哈希表末端后需要执行的代码块。我们还没有看到，但是哈希表的查找实际上是向后执行。搜索检查会逐渐减小索引，直到它命中表的开头，就结束了。这样做的原因是，表的开头我们是已知的，但是表的结尾是未知的，索引递增的查找需要更多的指令来判断是否已经到达表尾。

偏移量 `0x40` 的代码处理了这种情况。如果不匹配，继续执行接下去的指令。

```
1   0x0038 ldp    x9, x17, [x12, #-0x10]!
```

又出现 `ldp` 了，再一次从缓存的bucket中加载。这次他从偏移量为0x10的地方加载当前缓存bucket的地址。地址引用末尾的感叹号是一个有趣的特性。这指定一个寄存器进行回写，意思就是寄存器会更新为计算后的值。这条指令有效的执行了 `x12 -= 16` 来加载新的bucket，并使 `x12` 指向这个新的bucket。

```
1   0x003c b      0x20
```

现在已经加载了一个新的bucket，所以接下去的执行就要回到之前的检查当前bucket是否匹配的代码。这条指令代表回到上面的 `0x0020` ，使用新的值再执行一次所有代码。如果仍然没有找到匹配的bucket，这些代码会持续执行，直到找到匹配的，或者空的bucket，或者命中表的开头。

```
1   0x0040 add    x12, x12, w11, uxtw #4
```

`x12` 包含了当前bucket的指针，这里同样指的是第一个bucket。 `w11` 包含了表的掩码，即表的大小。这里将两个值做了相加，同时将 `w11` 左移4位。现在 `x12` 中的结果是指向表的末尾，并且从这里可以恢复查找。

```
1   0x0044 ldp    x9, x17, [x12]
```

`ldp` 加载了一个新的bucket到 `x9` 和 `x17` 。

```
1   0x0048 cmp    x9, x1
2   0x004c b.ne   0x54
3   0x0050 br     x17
```

这段代码检查bucket是否匹配，并跳转到bucket的 `IMP` 。这和 `0x0020` 处的代码是重复的。

```
1   0x0054 cbz    x9, __objc_msgSend_uncached
```

和之前一样，如果bucket是空的就说明缓存miss了，接下去执行用C实现的更完整更详尽的代码。

```
1   0x0058 cmp    x12, x10
2   0x005c b.eq   0x68
```

这一步再次检查是否已到表头，如果再次命中表头的话就跳转到 `0x68` 。这里的情况是直接跳到C实现的，进行全面查找的代码：

```
1   0x0068 b      __objc_msgSend_uncached
```

这种情况应该不会发生。表会随着条目的增加而增长，并且它永远不会100%满。哈希表会太满会变得很低效，因为经常会发生哈希碰撞。

为什么这段代码会在这，源码中有一段注释做了解释：

> 当缓存被破坏时，循环扫描将会miss而不是挂起。
> 缓慢的路径（C实现的代码）可能会检测到破坏，并在之后终止。

额外的二次扫描检查是为了在遇到内存被破坏或者无效对象时，防止陷入无限循环而榨干性能。举个例子，堆损坏能够在缓存中塞满非0的数据，或者设置缓存的掩码为0，缓存不命中就会一直循环执行缓存扫描。额外的检查可

以停止循环，将问题转变为崩溃日志。

还有一种情况，当有另一个线程同时修改缓存时会引起这个线程即不命中也不miss。C代码做了额外的工作来解决竞争。之前一个版本的 `objc_msgSend` 的做法是错误的，它会立即终止，而不是回到C代码，这样做的话运气不好的时候会发生罕见的崩溃。

Tagged Pointer Handler

在第一行汇编指令中做了空指针和 `tagged pointer` 的判断，如果小于等于0则会跳转到偏移量 `0x6c` 处的代码。我们继续往下看：

```
1   0x006c b.eq    0xa4
```

负数说明是tagged pointer，而0则说明是 `nil` 。这两种情况处理起来是不一样的，所以这里首先做的是如果为 `nil` 时跳转到 `0xa4` ，否则继续执行下面的代码，处理tagged pointer的情况。

在我们继续往下之前，简单讨论下tagged pointer是如何工作的。tagged pointer支持多个类。tagged pointer的前四位（ARM 64上）指明对象的类是哪个。本质上就是tagged pointer的isa。当然4位不够保存一个类的指针。实际上，有一张特殊的表存储了可用的tagged pointer的类。这个对象的类的查找是通过搜索这张表中的索引，是否对应于这个tagged pointer的前4位。

tagged pointer（至少在AMR64上）也支持扩展类。当前四位都设置为1，接下去的8位用于索引tagged pointer扩展类的表。减少存储他们的代价，就允许运行时能够支持更多的tagged pointer类。

Let's continue.

```
1   0x0070 mov     x10, #-0x1000000000000000
```

这里将 `x10` 设置成一个整型值，只有前四位被设置，其余位都为0。作为掩码用于从 `self` 中提取标签位。

```
1   0x0074 cmp     x0, x10
2   0x0078 b.hs    0x90
```

这步检查是为了扩展的tagged pointer。如果 `self` 大于等于 `x10` 的值，意味着前四位都被设置了。这种情况下会跳转到 `0x90` ，处理扩展类。否则，使用tagged pointer主表。

```
1   0x007c adrp   x10, _objc_debug_taggedpointer_classes@PAGE
2   0x0080 add    x10, x10, _objc_debug_taggedpointer_classes@PAGEOFF
```

这里加载了 `_objc_debug_taggedpointer_classes` 的地址，即tagged pointer主表。ARM64需要两条指令来加载一个符号的地址。这是RISC样架构上的一个标准技术。AMR64上的指针是64位宽的，指令是32位宽。所以一个指令无法保存一个完整的指针。

x86不会遇到这种问题，因为他有可变长指令。它只能使用10字节的指令，两个字节用于标识指令自己，以及目标寄存器，8个字节用于持有指针的值。

在一个固定长度指令的机器上，就需要分块加载。这里我们需要两块， `adrp` 指令加载前半部分的值， `add` 指令添加了后半部分。

```
1   0x0084 lsr    x11, x0, #60
```

`x0` 的前四位保存了tagged pointer的索引。如果需要把它用于索引，则需要将其右移60位，这样它就变成一个0-15的整数了。这个指令执行了位移并将索引放到 `x11` 中。

```
1   0x0088 ldr    x16, [x10, x11, lsl #3]
```

这里通过 `x11` 里的索引到 `x10` 所指向的表中查找条目。 `x16` 寄存器现在包含了这个tagged pointer的类。

```
1   0x008c b      0x10
```

有了 `x16` 中的类后，我们就能够回到主要的逻辑代码了。在偏移量为 `0x10` 的代码处开始，使用 `x16` 中的类执行后续的操作。

```
1   0x0090 adrp   x10, _objc_debug_taggedpointer_ext_classes@PAGE
2   0x0094 add    x10, x10, _objc_debug_taggedpointer_ext_classes@PAGEOFF
```

扩展的tagged类执行起来也是一样的。这两条指令加载了指向扩展表的指针。

```
1   0x0098 ubfx   x11, x0, #52, #8
```

这条指令加载了扩展类的索引。它从 `self` 中的第52位开始，提取8位，保存到 `x11` 中。

```
1   0x009c ldr    x16, [x10, x11, lsl #3]
```

和之前一样，这个索引用于在表中查找类，并存入 `x16` 。

```
1   0x00a0 b      0x10
```

也是一样，回到 `0x10` 处的主逻辑代码。

OK，接下来是 `nil` 的处理方法了。

nil Handler

以下全部代码就是 `nil` 的处理过程。

```
1   0x00a4 mov    x1, #0x0
2   0x00a8 movi   d0, #0000000000000000
3   0x00ac movi   d1, #0000000000000000
4   0x00b0 movi   d2, #0000000000000000
5   0x00b4 movi   d3, #0000000000000000
6   0x00b8 ret
```

`nil` 的处理方式和其他代码完全不同。没有类的查找也没有方法的派发。这里为 `nil` 做的所有事情就是返回0给调用者。

事实上这个过程有点复杂， `objc_msgSend` 不知道调用者希望得到什么类型的返回值，是一个整型？两个？还是一个浮点数类型，或是其他?

幸运的是，所有用于返回值的寄存器都能够被安全的覆盖，即使他们没有被用于这次特定的调用者的返回值。整型的返回值被保存在 `x0` 和 `x1` 中，浮点数返回值被保存在向量寄存器 `v0` 至 `v3` 中。还有多个寄存器被用于返回更小的结构。

上面的代码清除了 `x1` ，以及 `v0` 至 `v3` 。 `d0` 至 `d3` 指的是对应的 `v` 寄存器的底部后半部分，存储在其中可以清楚前半部分，所以4条 `movi` 指令的作用就是清空这4个寄存器。然后将控制权返回给调用者。

你可能想要知道为什么不清楚 `x0` 。回答很简单： `x0` 中放的是 `self` ，现在的情况中是 `nil` ，所以它原本就是0！你可以节省一条清零的指令。

对于寄存器不够存储的，更大结构的返回值会怎样？这需要调用者的一些合作。通过调用者来分配足够多的内存存储大型的结构体，并将内存地址传入 `x8` 。函数通过写入这块内存来返回值。 `objc_msgSend` 不能清除这块内存，因为它不知道返回值到底有多大。为了解决这个问题，编译器生成的代码会在调用 `objc_msgSend` 之前用0填满这块内存。

以上就是 `nil` 的处理方法，以及 `objc_msgSend` 的全部。

‹ MMeTokenDecrypt实验

对Mac上恶意软件MacRansom的分析 ›

© 2021 ♥ Mad Mark

Powered by Hexo | Theme - NexT.Mist