



黑幕背后的Autorelease

2014年10月15日

我是前言

Autorelease机制是iOS开发者管理对象内存的好伙伴，MRC中，调用 `[obj autorelease]` 来延迟内存的释放是一件简单自然的事，ARC下，我们甚至可以完全不知道Autorelease就能管理好内存。而在这背后，objc和编译器都帮我们做了哪些事呢，它们是如何协作来正确管理内存的呢？刨根问底，一起来探究下黑幕背后的Autorelease机制。

Autorelease对象什么时候释放？

这个问题拿来面试题，问过很多人，没有几个能答对的。很多答案都是“当前作用域大括号结束时释放”，显然木有正确理解Autorelease机制。

在没有手加Autorelease Pool的情况下，Autorelease对象是在当前的 `runloop` 迭代结束时释放的，而它能够释放的原因是系统在每个runloop迭代中都加入了自动释放池Push和Pop

小实验

```
__weak id reference = nil;
- (void)viewDidLoad {
    [super viewDidLoad];
    NSString *str = [NSString stringWithFormat:@"sunnyxx"];
    // str是一个autorelease对象，设置一个weak的引用来观察它
    reference = str;
}
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    NSLog(@"%@", reference); // Console: sunnyxx
}
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    NSLog(@"%@", reference); // Console: (null)
}
```

这个实验同时也证明了 `viewDidLoad` 和 `viewWillAppear` 是在同一个runloop调用的，而 `viewDidAppear` 是在之后的某个runloop调用的。

由于这个vc在loadView之后便add到了window层级上，所以 `viewDidLoad` 和 `viewWillAppear` 是在同一个runloop调用的，因此在 `viewWillAppear` 中，这个autorelease的变量依然有值。

当然，我们也可以手动干预Autorelease对象的释放时机：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    @autoreleasepool {
        NSString *str = [NSString stringWithFormat:@"sunnyxx"];
    }
    NSLog(@"%@", str); // Console: (null)
}
```

Autorelease原理

AutoreleasePoolPage

ARC下，我们使用 `@autoreleasepool{}` 来使用一个AutoreleasePool，随后编译器将其改写成下面的样子：

```
void *context = objc_autoreleasePoolPush();
// {}中的代码
objc_autoreleasePoolPop(context);
```

而这两个函数都是对 `AutoreleasePoolPage` 的简单封装，所以自动释放机制的核心就在于这个类。

`AutoreleasePoolPage`是一个C++实现的类

- `AutoreleasePool`并没有单独的结构，而是由若干个`AutoreleasePoolPage`以 **双向链表** 的形式组合而成（分别对应结构中的parent指针和child指针）
- `AutoreleasePool`是按线程——对应的（结构中的thread指针指向当前线程）
- `AutoreleasePoolPage`每个对象会开辟4096字节内存（也就是虚拟内存一页的大小），除了上面的实例变量所占空间，剩下的空间全部用来储存autorelease对象的地址
- 上面的 `id *next` 指针作为游标指向栈顶最新add进来的autorelease对象的下一个位置
- 一个`AutoreleasePoolPage`的空间被占满时，会新建一个`AutoreleasePoolPage`对象，连接链表，后来的autorelease对象在新的page加入

所以，若当前线程中只有一个AutoreleasePoolPage对象，并记录了很多autorelease对象地址时内存如下图：

图中的情况，这一页再加入一个autorelease对象就要满了（也就是next指针马上指向栈顶），这时就要执行上面说的操作，建立下一页page对象，与这一页链表连接完成后，新page的 **next** 指针被初始化在栈底（begin的位置），然后继续向栈顶添加新对象。

所以，向一个对象发送 - **autorelease** 消息，就是将这个对象加入到当前AutoreleasePoolPage的栈顶next指针指向的位置

释放时刻

每当进行一次 **objc_autoreleasePoolPush** 调用时，runtime向当前的AutoreleasePoolPage中add进一个 **哨兵对象**，值为0（也就是个nil），那么这一个page就变成了下面的样子：

objc_autoreleasePoolPush 的返回值正是这个哨兵对象的地址，被 **objc_autoreleasePoolPop(哨兵对象)** 作为入参，于是：

1. 根据传入的哨兵对象地址找到哨兵对象所处的page
2. 在当前page中，将晚于哨兵对象插入的所有autorelease对象都发送一次 - **release** 消息，并向回移动 **next** 指针到正确位置
3. 补充2：从最新加入的对象一直向前清理，可以向前跨越若干个page，直到哨兵所在的page

刚才的objc_autoreleasePoolPop执行后，最终变成了下面的样子：

嵌套的AutoreleasePool

知道了上面的原理，嵌套的AutoreleasePool就非常简单了，pop的时候总会释放到上次push的位置为止，多层的pool就是多个哨兵对象而已，就像剥洋葱一样，每次一层，互不影响。

【附加内容】

Autorelease返回值的快速释放机制

值得一提的是，ARC下，runtime有一套对autorelease返回值的优化策略。
比如一个工厂方法：

```
+ (instancetype)createSark {  
    return [self new];  
}
```

```
// caller
Sark *sark = [Sark createSark];
```

秉着谁创建谁释放的原则，返回值需要是一个autorelease对象才能配合调用方正确管理内存，于是乎编译器改写成了形如下面的代码：

```
+ (instancetype)createSark {
    id tmp = [self new];
    return objc_autoreleaseReturnValue(tmp); // 代替我们调用autorelease
}
// caller
id tmp = objc_retainAutoreleasedReturnValue([Sark createSark]) // 代替我们调用retain
Sark *sark = tmp;
objc_storeStrong(&sark, nil); // 相当于代替我们调用了release
```

一切看上去都很好，不过既然编译器知道了这么多信息，干嘛还要劳烦autorelease这个开销不小的机制呢？于是乎，runtime使用了一些黑魔法将这个问题解决了。

黑魔法之Thread Local Storage

Thread Local Storage (TLS) 线程局部存储，目的很简单，将一块内存作为某个线程专有的存储，以key-value的形式进行读写，比如在非arm架构下，使用pthread提供的方法实现：

```
void* pthread_getspecific(pthread_key_t);
int pthread_setspecific(pthread_key_t , const void *);
```

说它是黑魔法可能被懂pthread的笑话- -

在返回值身上调用 `objc_autoreleaseReturnValue` 方法时，runtime将这个返回值object储存在TLS中，然后直接返回这个object（不调用autorelease）；同时，在外部接收这个返回值的 `objc_retainAutoreleasedReturnValue` 里，发现TLS中正好存了这个对象，那么直接返回这个object（不调用retain）。

于是乎，调用方和被调方利用TLS做中转，很有默契的免去了对返回值的内存管理。

于是问题又来了，假如被调方和主调方只有一边是ARC环境编译的该咋办？（比如我们在ARC环境下用了非ARC编译的第三方库，或者反之）

只能动用更高级的黑魔法。

黑魔法之__builtin_return_address

这个内建函数原型是 `char *__builtin_return_address(int level)`，作用是得到函数的返回地址，参数表示层数，如 `__builtin_return_address(0)` 表示当前函数体返回地址，传1是调用这个函数的

外层函数的返回值地址，以此类推。

```
- (int)foo {
    NSLog(@"%p", __builtin_return_address(0)); // 根据这个地址能找到下面ret的地址
    return 1;
}
// caller
int ret = [sark foo];
```

看上去也没啥厉害的，不过要知道，函数的返回值地址，也就对应着调用者结束这次调用的地址（或者相差某个固定的偏移量，根据编译器决定）

也就是说，被调用的函数也有翻身做地主的机会了，可以反过来对主调方干点坏事。

回到上面的问题，如果一个函数返回前知道调用方是ARC还是非ARC，就有机会对于不同情况做不同的处理

黑魔法之反查汇编指令

通过上面的__builtin_return_address加某些偏移量，被调方可以定位到主调方在返回值后面的 汇编指令：

```
// caller
int ret = [sark foo];
// 内存中接下来的汇编指令（x86，我不懂汇编，瞎写的）
movq ??? ???
callq ???
```

而这些汇编指令在内存中的值是固定的，比如movq对应着0x48。

于是乎，就有了下面的这个函数，入参是调用方__builtin_return_address传入值

```
static bool callerAcceptsFastAutorelease(const void * const ra0) {
    const uint8_t *ra1 = (const uint8_t *)ra0;
    const uint16_t *ra2;
    const uint32_t *ra4 = (const uint32_t *)ra1;
    const void **sym;
    // 48 89 c7    movq  %rax,%rdi
    // e8        callq symbol
    if (*ra4 != 0xe8c78948) {
        return false;
    }
    ra1 += (long)*(const int32_t *) (ra1 + 4) + 8;
    ra2 = (const uint16_t *)ra1;
    // ff 25     jmpq *symbol@DYLDMAGIC(%rip)
    if (*ra2 != 0x25ff) {
        return false;
    }
}
```

```
}
ra1 += 6l + (long)*(const int32_t *)(ra1 + 2);
sym = (const void **)ra1;
if (*sym != objc_retainAutoreleasedReturnValue)
{
    return false;
}
return true;
}
```

它检验了主调方在返回值之后是否紧接着调用了 `objc_retainAutoreleasedReturnValue`，如果是，就知道了外部是ARC环境，反之就走没被优化的老逻辑。

其他Autorelease相关知识点

使用容器的block版本的枚举器时，内部会自动添加一个AutoreleasePool：

```
[array enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
    // 这里被一个局部@autoreleasepool包围着
}];
```

当然，在普通for循环和for in循环中没有，所以，还是新版的block版本枚举器更加方便。for循环中遍历产生大量autorelease变量时，就需要手加局部AutoreleasePool咯。

[上一篇](#)[下一篇](#)

原创文章，版权声明：署名-非商业性使用-相同方式共享 2.5

对博主感兴趣？微信订阅号中关注 sunnyxx 或关注微博@我就叫Sunny怎么了

© 2015 - 2016 sunnyxx | Powered by Hexo