



Block 面试题

一、什么是 Block?

Block 是将函数及其执行上下文封装起来的对象。

比如:

```
NSInteger num = 3;

NSInteger(^block)(NSInteger) = ^NSInteger(NSInteger n){

    return n*num;

};

block(2);
```

通过 `clang -rewrite-objc WYTest.m` 命令编译该.m 文件，发现该 block 被编译成这个形式:

```
NSInteger num = 3;

NSInteger(*block)(NSInteger) = ((NSInteger (*)(
NSInteger))&__WYTest__blockTest_block_impl_0(
(void *)__WYTest__blockTest_block_func_0,
&__WYTest__blockTest_block_desc_0_DATA, num));

((NSInteger (*)(__block_impl *, NSInteger))((__block_impl *)block)-
>FuncPtr)((__block_impl *)block, 2);
```

其中 `WYTest` 是文件名，`blockTest` 是方法名，这些可以忽略。

其中 `__WYTest__blockTest_block_impl_0` 结构体为

```

struct __WYTest__blockTest_block_impl_0 {
    struct __block_impl impl;
    struct __WYTest__blockTest_block_desc_0* Desc;
    NSInteger num;
    __WYTest__blockTest_block_impl_0(void *fp, struct
    __WYTest__blockTest_block_desc_0 *desc, NSInteger _num, int
    flags=0) : num(_num) {
        impl.isa = &_NSConcreteStackBlock;
        impl.Flags = flags;
        impl.FuncPtr = fp;
        Desc = desc;
    }
};

```

__block_impl 结构体为

```

struct __block_impl {
    void *isa; // isa 指针, 所以说 block 是对象
    int Flags;
    int Reserved;
    void *FuncPtr; // 函数指针
};

```

block 内部有 isa 指针, 所以说其本质也是 OC 对象

block 内部则为:

```

static NSInteger __WYTest__blockTest_block_func_0(struct __WYTest__blockTest_block_impl_0 *__cself, NSInteger n) {
    NSInteger num = __cself->num; // bound by copy

    return n*num;
}

```

所以说 Block 是将函数及其执行上下文封装起来的对象

既然 block 内部封装了函数, 那么它同样也有参数和返回值。

二、Block 变量截获

1、局部变量截获 是值截获。 比如:

```
NSInteger num = 3;

NSInteger(^block)(NSInteger) = ^NSInteger(NSInteger n){

    return n*num;
};

num = 1;

NSLog(@"%zd",block(2));
```

这里的输出是 6 而不是 2，原因就是局部变量 `num` 的截获是值截获。
同样，在 `block` 里如果修改变量 `num`，也是无效的，甚至编译器会报错。

```
NSMutableArray * arr = [NSMutableArray arrayWithObjects:@"1",@"2", nil];

void(^block)(void) = ^{

    NSLog(@"%@",arr); //局部变量

    [arr addObject:@"4"];

};

[arr addObject:@"3"];

arr = nil;

block();
```

打印为 1, 2, 3

局部对象变量也是一样，截获的是值，而不是指针，在外部将其置为 `nil`，对 `block` 没有影响，而该对象调用方法会影响

2、局部静态变量截获 是指针截获。

```
static NSInteger num = 3;

NSInteger(^block)(NSInteger) = ^NSInteger(NSInteger n){

    return n*num;
};

num = 1;

NSLog(@"%zd",block(2));
```

输出为 2，意味着 `num = 1` 这里的修改 `num` 值是有效的，即是指针截获。
同样，在 `block` 里去修改变量 `m`，也是有效的。

3、全局变量，静态全局变量截获：不截获,直接取值。

我们同样用 clang 编译看下结果。

```
static NSInteger num3 = 300;

NSInteger num4 = 3000;

- (void)blockTest
{
    NSInteger num = 30;

    static NSInteger num2 = 3;

    __block NSInteger num5 = 30000;

    void(^block)(void) = ^{

        NSLog(@"%zd", num); // 局部变量

        NSLog(@"%zd", num2); // 静态变量

        NSLog(@"%zd", num3); // 全局变量

        NSLog(@"%zd", num4); // 全局静态变量

        NSLog(@"%zd", num5); // __block 修饰变量
    };

    block();
}
```

编译后

```
struct __WYTest__blockTest_block_impl_0 {
    struct __block_impl impl;
    struct __WYTest__blockTest_block_desc_0* Desc;
    NSInteger num; // 局部变量
    NSInteger *num2; // 静态变量
    __Block_byref_num5_0 *num5; // by ref // __block 修饰变量
    __WYTest__blockTest_block_impl_0(void *fp, struct
    __WYTest__blockTest_block_desc_0 *desc, NSInteger _num, NSInteger
    *_num2, __Block_byref_num5_0 *_num5, int flags=0) : num(_num),
    num2(_num2), num5(_num5->__forwarding) {
        impl.isa = &_NSConcreteStackBlock;
        impl.Flags = flags;
        impl.FuncPtr = fp;
        Desc = desc;
    }
};
```

(`impl.isa = &_NSConcreteStackBlock`;这里注意到这一句, 即说明该 `block` 是栈 `block`)
可以看到局部变量被编译成值形式, 而静态变量被编成指针形式, 全局变量并未截获。而 `__block` 修饰的变量也是以指针形式截获的, 并且生成了一个新的结构体对象:

```
struct __Block_byref_num5_0 {  
    void *__isa;  
    __Block_byref_num5_0 *__forwarding;  
    int __flags;  
    int __size;  
    NSInteger num5;  
};
```

该对象有个属性: `num5`, 即我们用 `__block` 修饰的变量。

这里 `__forwarding` 是指向自身的(栈 `block`)。

一般情况下, 如果我们要对 `block` 截获的局部变量进行赋值操作需添加 `__block` 修饰符, 而对全局变量, 静态变量是不需要添加 `__block` 修饰符的。

另外, `block` 里访问 `self` 或成员变量都会去截获 `self`。

三、Block 的几种形式

- 分为全局 Block(`_NSConcreteGlobalBlock`)、栈 Block(`_NSConcreteStackBlock`)、堆 Block(`_NSConcreteMallocBlock`)三种形式
其中栈 Block 存储在栈(stack)区，堆 Block 存储在堆(heap)区，全局 Block 存储在已初始化数据(.data)区

1、不使用外部变量的 block 是全局 block

比如:

```
NSLog(@"%@", [^{
    NSLog(@"globalBlock");
} class]);
```

输出:

```
__NSGlobalBlock__
```

2、使用外部变量并且未进行 copy 操作的 block 是栈 block

比如:

```
NSInteger num = 10;
NSLog(@"%@", [^{
    NSLog(@"stackBlock:%zd", num);
} class]);
```

输出:

```
__NSStackBlock__
```

日常开发常用于这种情况:

```
[self testWithBlock:^(
    NSLog(@"%@", self);
)];

- (void)testWithBlock:(dispatch_block_t)block {
    block();

    NSLog(@"%@", [block class]);
}
```

3、对栈 block 进行 copy 操作，就是堆 block，而对全局 block 进行 copy，仍是全局 block

- 比如堆 1 中的全局进行 copy 操作，即赋值：

```
void (^globalBlock)(void) = ^{
    NSLog(@"globalBlock");
};

NSLog(@"%@",[globalBlock class]);
```

输出：

```
__NSGlobalBlock__
```

仍是全局 block

- 而对 2 中的栈 block 进行赋值操作：

```
NSInteger num = 10;

void (^mallocBlock)(void) = ^{
    NSLog(@"stackBlock:%zd",num);
};

NSLog(@"%@",[mallocBlock class]);
```

输出：

```
__NSMallocBlock__
```

对栈 blockcopy 之后，并不代表着栈 block 就消失了，左边的 malloc 是堆 block，右边被 copy 的仍是栈 block
比如：

```
[self testWithBlock:^(
    NSLog(@"%@",self);
)];

- (void)testWithBlock:(dispatch_block_t)block
{
    block();

    dispatch_block_t tempBlock = block;

    NSLog(@"%@",[block class],[tempBlock class]);
}
```

输出：

```
__NSStackBlock__, __NSMallocBlock__
```

- 即如果对栈 Block 进行 copy，将会 copy 到堆区，对堆 Block 进行 copy，将会增加引用计数，对全局 Block 进行 copy，因为是已经初始化的，所以什么也不做。

另外，__block 变量在 copy 时，由于 __forwarding 的存在，栈上的 __forwarding 指针会指向堆上的 __forwarding 变量，而堆上的 __forwarding 指针指向其自身，所以，如果对 __block 的修改，实际上是在修改堆上的 __block 变量。

即 __forwarding 指针存在的意义就是，无论在任何内存位置，都可以顺利地访问同一个 __block 变量。

- 另外由于 block 捕获的 __block 修饰的变量会去持有变量，那么如果用 __block 修饰 self，且 self 持有 block，并且 block 内部使用到 __block 修饰的 self 时，就会造成多循环引用，即 self 持有 block，block 持有 __block 变量，而 __block 变量持有 self，造成内存泄漏。

比如：

```
__block typeof(self) weakSelf = self;

_testBlock = ^{

    NSLog(@"%@", weakSelf);

};

_testBlock();
```

如果要解决这种循环引用，可以主动断开 __block 变量对 self 的持有，即在 block 内部使用完 weakSelf 后，将其置为 nil，但这种方式有个问题，如果 block 一直不被调用，那么循环引用将一直存在。

所以，我们最好还是用 __weak 来修饰 self