

## 目录结构

- 1.相关概念
  - 1.1 操作系统中的栈和堆
  - 1.2 结构体 (Struct)
  - 1.3 闭包 (Closure)
- 2.block基础知识
  - 2.1 block的原型及定义
  - 2.2 将block作为参数传递
  - 2.3 闭包性
  - 2.4 block中变量的复制与修改
- 3.编译器中的block
  - 3.1 block的数据结构定义
  - 3.2 block的类型
  - 3.3 编译器如何编译
  - 3.4 copy()和dispose()

文章 (/blogs) > DevTalking (/blog/devtalking) > 文章详情

## Objective-C中的Block (/blog/devtalking/1190000002446149)



DevTalking (/u/devtalking) 40 3天前 发布

推荐

2 推荐

收藏

3 收藏, 193 浏览

## 1.相关概念

在这篇笔记开始之前，我们需要对以下概念有所了解。

### 1.1 操作系统中的栈和堆

注：这里所说的堆和栈与数据结构中的堆和栈不是一回事。

我们先来看看一个由C/C++/OBJC编译的程序占用内存分布的结构：



- 栈区 (stack)：由系统自动分配，一般存放函数参数值、局部变量的值等。由编译器自动创建与释放。其操作方式类似于数据结构中的栈，即后进先出、先进后出的原则。

例如：在函数中申明一个局部变量 `int b` ;系统自动在栈中为b开辟空间。

- 堆区 (heap)：一般由程序员申请并指明大小，最终也由程序员释放。如果程序员不释放，程序结束时可能会由OS回收。对于堆区的管理是采用链表式管理的，操作系统有一个记录空闲内存地址的链表，当接收到程序分配内存的申请时，操作系统就会遍历该链表，遍历到一个记录的内存地址大于申请内存的链表节点，并将该节点从该链表中删除，然后将该节点记录的内存地址分配给程序。

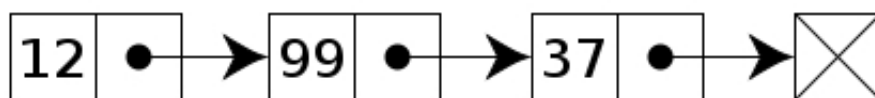
例如：在C中malloc函数

```
char *p1;
```

```
p1 = (char *)malloc(10);
```

但是p1本身是在栈中的。

链表：是一种常见的基础数据结构，一般分为单向链表、双向链表、循环链表。以下为单向链表的结构图：



一个单向链表包含两个值：当前节点的值和一个指向下一个节点的链接

单向链表是链表中最简单的一种，它包含两个区域，一个信息域和一个指针域。信息域保存或显示关于节点的信息，指针域储存下一个节点的地址。

上述的空闲内存地址链表的信息域保存的就是空闲内存的地址。

- 全局区/静态区：顾名思义，全局变量和静态变量存储在这个区域。只不过初始化的全局变量和静态变量存储在一块，未初始化的全局变量和静态变量存储在一块。程序结束后由系统释放。
- 文字常量区：这个区域主要存储字符串常量。程序结束后由系统释放。
- 程序代码区：这个区域主要存放函数体的二进制代码。

下面举一个前辈写的例子：

```
//main.cpp
int a = 0; // 全局初始化区
char *p1; // 全局未初始化区
main {
    int b; // 栈
    char s[] = "abc"; // 栈
    char *p2; // 栈
    char *p3 = "123456"; // 123456\0在常量区，p3在栈上
    static int c = 0; // 全局静态初始化区
    p1 = (char *)malloc(10);
    p2 = (char *)malloc(20); // 分配得来的10和20字节的区域就在堆区
    strcpy(p1, "123456"); // 123456\0在常量区，这个函数的作用是将"123456" 这串字符串复制一份放在p1申请的10个字节的堆区域中。
    // p3指向的"123456"与这里的"123456"可能会被编译器优化成一个地址。
}
```

strcpy函数

原型声明：extern char \*strcpy(char\* dest, const char \*src);

功能：把从src地址开始且含有NULL结束符的字符串复制到以dest开始的地址空间。

## 1.2 结构体 (Struct)

在C语言中，结构体(struct)指的是一种数据结构。结构体可以被声明为变量、指针或数组等，用以实现较复杂的数据结构。结构体同时也是一些元素的集合，这些元素称为结构体的成员(member)，且这些成员可以为不同的类型，成员一般用名字访问。

我们来看看结构体的定义：

```
struct tag { member-list } variable-list;
```

- **struct**：结构体关键字。
- **tag**：结构体标签。
- **member-list**：结构体成员列表。
- **variable-list**：为结构体声明的变量列表。

在一般情况下，**tag**，**member-list**，**variable-list** 这三部分至少要出现两个。以下为示例：

```
// 该结构体拥有3个成员，整型的a，字符型的b，双精度型的c
// 并且为该结构体声明了一个变量s1
// 该结构体没有标明其标签
struct{
    int a;
    char b;
    double c;
} s1;

// 该结构体拥有同样的三个成员
// 并且该结构体标明了标签EXAMPLE
// 该结构体没有声明变量
struct EXAMPLE{
    int a;
    char b;
    double c;
};

//用EXAMPLE标签的结构体，另外声明了变量t1、t2、t3
struct EXAMPLE t1, t2[20], *t3;
```

以上就是简单结构体的代码示例。结构体的成员可以包含其他结构体，也可以包含指向自己结构体类型的指针。结构体的变量也可以是指针。

下面我们来看看结构体成员的访问。结构体成员依据结构体变量类型的不同，一般有2种访问方式，一种为直接访问，一种为间接访问。直接访问应用于普通的结构体变量，间接访问应用于指向结构体变量的指针。直接访问使用结构体变量名.成员名，间接访问使用(\*结构体指针名).成员名或者使用结构体指针名->成员名。相同的成员名称依靠不同的变量前缀区分。

```

struct EXAMPLE{
    int a;
    char b;
};

//声明结构体变量s1和指向结构体变量的指针s2
struct EXAMPLE s1, *s2;

//给变量s1和s2的成员赋值,注意s1.a和s2->a并不是同一成员
s1.a = 5;
s1.b = 6;
s2->a = 3;
s2->b = 4;

```

最后我们来看看结构体成员存储。在内存中，编译器按照成员列表顺序分别为每个结构体成员分配内存。如果想确认结构体占多少存储空间，则使用关键字 `sizeof`，如果想得知结构体的某个特定成员在结构体的位置，则使用 `offsetof` 宏(定义于`stddef.h`)。

```

struct EXAMPLE{
    int a;
    char b;
};

//获得EXAMPLE类型结构体所占内存大小
int size_example = sizeof( struct EXAMPLE );

//获得成员b相对于EXAMPLE储存地址的偏移量
int offset_b = offsetof( struct EXAMPLE, b );

```

## 1.3 闭包 (Closure)

---

闭包就是一个函数，或者一个指向函数的指针，加上这个函数执行的非局部变量。

说的通俗一点，就是闭包允许一个函数访问声明该函数运行上下文中的变量，甚至可以访问不同运行上下文中的变量。

我们用脚本语言来看一下：

```

function funA(callback){
    alert(callback());
}

function funB(){
    var str = "Hello World"; // 函数funB的局部变量，函数funA的非局部变量
    funA (
        function () {
            return str;
        }
    );
}

```

通过上面的代码我们可以看出，按常规思维来说，变量 `str` 是函数 `funB` 的局部变量，作用域只在函数 `funB` 中，函数 `funA` 是无法访问到 `str` 的。但是上述代码示例中函数 `funA` 中的 `callback` 可以访问到 `str`，这是为什么呢，因为闭包性。

## 2.block基础知识

---

block实际上就是Objective-C语言对闭包的实现。

### 2.1 block的原型及定义

---

我们来看看block的原型：

```
NSString * ( ^ myBlock )( int );
```

上面的代码声明了一个block(^)原型，名字叫做 `myBlock`，包含一个 `int` 型的参数，返回值为 `NSString` 类型的指针。

下面来看看block的定义：

```
myBlock = ^( int paramA )
{
    return [ NSString stringWithFormat: @"Passed number: %i", paramA ];
};
```

上面的代码中，将一个函数体赋值给了 `myBlock` 变量，其接收一个名为 `paramA` 的参数，返回一个 `NSString` 对象。

注意：一定不要忘记block后面的分号。

定义好block后，就可以像使用标准函数一样使用它了：

```
myBlock(7);
```

由于block数据类型的语法会降低整个代码的阅读性，所以常使用 `typedef` 来定义block类型。例如，下面的代码创建了 `GetPersonEducationInfo` 和 `GetPersonFamilyInfo` 两个新类型，这样我们就可以在下面的方法中使用更加有语义的数据类型。

```

// Person.h
#import <Foundation/Foundation.h>

// Define a new type for the block
typedef NSString * (^GetPersonEducationInfo)(NSString *);
typedef NSString * (^GetPersonFamilyInfo)(NSString *);

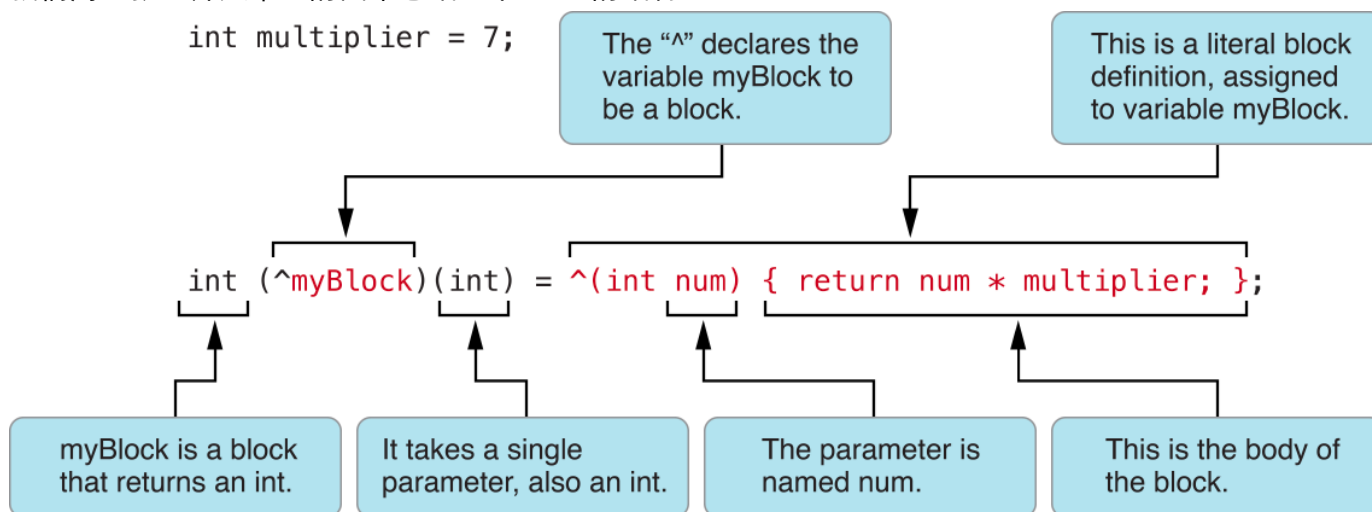
@interface Person : NSObject

- (NSString *)getPersonInfoWithEducation:(GetPersonEducationInfo)educationInfo
  andFamily:(GetPersonFamilyInfo)familyInfo;

@end

```

我们用一张大师文章里的图来总结一下block的结构：



## 2.2 将block作为参数传递

```

// .h
-(void) testBlock:( NSString * ( ^ )( int ) )myBlock;

// .m
-(void) testBlock:( NSString * ( ^ )( int ) )myBlock
{
    NSLog(@"Block returned: %@", myBlock(7) );
}

```

由于Objective-C是强制类型语言，所以作为函数参数的block也必须要指定返回值的类型，以及相关参数类型。

## 2.3 闭包性

上文说过，block实际是Objc对闭包的实现。

我们来看看下面代码：

```

#import <Cocoa/Cocoa.h>

void logBlock( int ( ^ theBlock )( void ) )
{
    NSLog( @"Closure var X: %i", theBlock() );
}

int main( void )
{
    NSAutoreleasePool * pool;
    int ( ^ myBlock )( void );
    int x;

    pool = [ [ NSAutoreleasePool alloc ] init ];
    x = 42;

    myBlock = ^( void )
    {
        return x;
    };

    logBlock( myBlock );

    [ pool release ];

    return EXIT_SUCCESS;
}

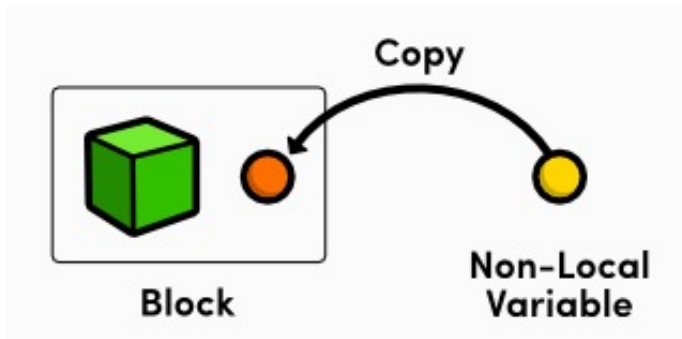
```

上面的代码在 main 函数中声明了一个整型，并赋值42，另外还声明了一个block，该block会将42返回。然后将block传递给 logBlock 函数，该函数会显示出返回的值42。即使是在函数 logBlock 中执行block，而block又声明在 main 函数中，但是block仍然可以访问到 x 变量，并将这个值返回。

注意：block同样可以访问全局变量，即使是 static 。

## 2.4 block中变量的复制与修改

对于block外的变量引用，block默认是将其复制到其数据结构中来实现访问的，如下图：



通过block进行闭包的变量是 const 的。也就是说不能在block中直接修改这些变量。来看看当block试着增加 x 的值时，会发生什么：



```
myBlock = ^( void )
{
    x++;

    return x;
};
```

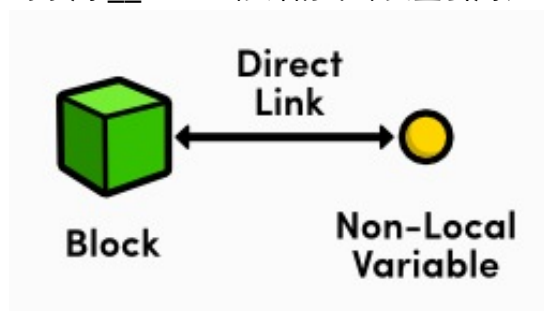
编译器会报错，表明在block中变量 x 是只读的。

有时候确实需要在block中处理变量，怎么办？别着急，我们可以用 `__block` 关键字来声明变量，这样就可以在block中修改变量了。

基于之前的代码，给x变量添加 `__block` 关键字，如下：

```
__block int x;
```

对于用 `__block` 修饰的外部变量引用，block是复制其引用地址来实现访问的，如下图：



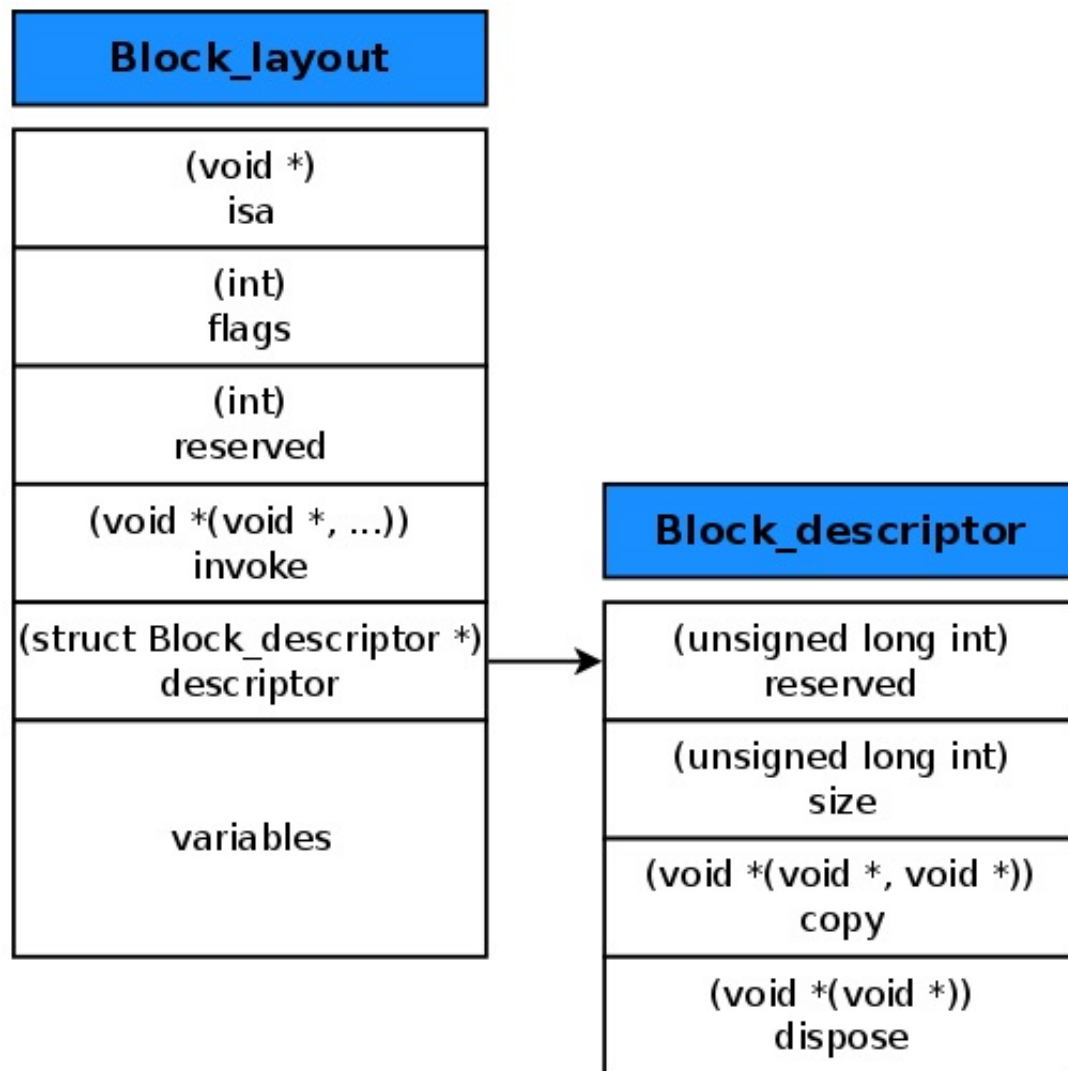
## 3.编译器中的block

---

### 3.1 block的数据结构定义

---

我们通过大师文章中的一张图来说明：



上图这个结构是在栈中的结构，我们来看看对应的结构体定义：

```
struct Block_descriptor {
    unsigned long int reserved;
    unsigned long int size;
    void (*copy)(void *dst, void *src);
    void (*dispose)(void *);
};

struct Block_layout {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(void *, ...);
    struct Block_descriptor *descriptor;
    /* Imported variables. */
};
```

从上面代码看出，Block\_layout 就是对block结构体的定义：

- isa 指针：指向表明该block类型的类。
- flags：按bit位表示一些block的附加信息，比如判断block类型、判断block引用计数、判断block是否需要执行辅助函数等。
- reserved：保留变量，我的理解是表示block内部的变量数。

- `invoke`：函数指针，指向具体的block实现的函数调用地址。
- `descriptor`：block的附加描述信息，比如保留变量数、block的大小、进行 `copy` 或 `dispose` 的辅助函数指针。
- `variables`：因为block有闭包性，所以可以访问block外部的局部变量。这些 `variables` 就是复制到结构体中的外部局部变量或变量的地址。

## 3.2 block的类型

---

block有几种不同的类型，每种类型都有对应的类，上述中 `isa` 指针就是指向这个类。这里列出常见的三种类型：

- `_NSConcreteGlobalBlock`：全局的静态block，不会访问任何外部变量，不会涉及到任何拷贝，比如一个空的block。例如：

```
#include <stdio.h>

int main()
{
    ^{ printf("Hello, World!\n"); } ();
    return 0;
}
```

- `_NSConcreteStackBlock`：保存在栈中的block，当函数返回时被销毁。例如：

```
#include <stdio.h>

int main()
{
    char a = 'A';
    ^{ printf("%c\n",a); } ();
    return 0;
}
```

- `_NSConcreteMallocBlock`：保存在堆中的block，当引用计数为0时被销毁。该类型的block都是由 `_NSConcreteStackBlock` 类型的block从栈中复制到堆中形成的。例如下面代码中，在 `exampleB_addBlockToArray` 方法中的block还是 `_NSConcreteStackBlock` 类型的，在 `exampleB` 方法中就被复制到了堆中，成为 `_NSConcreteMallocBlock` 类型的block：

```

void exampleB_addBlockToArray(NSMutableArray *array) {
    char b = 'B';
    [array addObject:^(
        printf("%c\n", b);
    )];
}

void exampleB() {
    NSMutableArray *array = [NSMutableArray array];
    exampleB_addBlockToArray(array);
    void (^block)() = [array objectAtIndex:0];
    block();
}

```

总结一下：

- `_NSConcreteGlobalBlock` 类型的block要么是空block，要么是不访问任何外部变量的block。它既不在栈中，也不在堆中，我理解为它可能在内存的全局区。
- `_NSConcreteStackBlock` 类型的block有闭包行为，也就是有访问外部变量，并且该block只且只有一次执行，因为栈中的空间是可重复使用的，所以当栈中的block执行一次之后就被清除出栈了，所以无法多次使用。
- `_NSConcreteMallocBlock` 类型的block有闭包行为，并且该block需要被多次执行。当需要多次执行时，就会把该block从栈中复制到堆中，供以多次执行。

### 3.3 编译器如何编译

我们通过一个简单的示例来说明：

```


#import <dispatch/dispatch.h>

typedef void(^BlockA)(void);

__attribute__((noinline))
void runBlockA(BlockA block) {
    block();
}

void doBlockA() {
    BlockA block = ^{
        // Empty block
    };
    runBlockA(block);
}

```



(/user/login)

上面的代码定义了一个名为 `BlockA` 的block类型，该block在函数 `doBlockA` 中实现，并将其作为函数 `runBlockA` 的参数，最后在函数 `doBlockA` 中调用函数 `runBlockA`。

注意：如果block的创建和调用都在一个函数里面，那么优化器(optimiser)可能会对代码做优化处理，从而导致我们看不到编译器中的一些操作，所以用 `__attribute__((noinline))` 给函数 `runBlockA` 添加 `noinline`，这样优化器就不会在 `doBlockA` 函数中对 `runBlockA` 的调用做内联优化处理。

我们来看看编译器做的工作内容：

```
#import <dispatch/dispatch.h>

__attribute__((noinline))
void runBlockA(struct Block_layout *block) {
    block->invoke();
}

void block_invoke(struct Block_layout *block) {
    // Empty block function
}

void doBlockA() {
    struct Block_descriptor descriptor;
    descriptor->reserved = 0;
    descriptor->size = 20;
    descriptor->copy = NULL;
    descriptor->dispose = NULL;

    struct Block_layout block;
    block->isa = _NSConcreteGlobalBlock;
    block->flags = 1342177280;
    block->reserved = 0;
    block->invoke = block_invoke;
    block->descriptor = descriptor;

    runBlockA(&block);
}
```

上面的代码结合block的数据结构定义，我们能很容易得理解编译器内部对block的工作内容。

## 3.4 copy()和dispose()

上文中提到，如果我们想要在以后继续使用某个block，就必须要对该block进行拷贝操作，即从栈空间复制到堆空间。所以拷贝操作就需要调用 `Block_copy()` 函数，block的 `descriptor` 中有一个 `copy()` 辅助函数，该函数在 `Block_copy()` 中执行，用于当block需要拷贝对象的时候，拷贝辅助函数会retain住已经拷贝的对象。

既然有有copy那么就应该有release，与 `Block_copy()` 对应的函数是 `Block_release()`，它的作用不言而喻，就是释放我们不需要再使用的block，block的 `descriptor` 中有一个 `dispose()` 辅助函数，该函数在 `Block_release()` 中执行，负责做和 `copy()` 辅助函数相反的操作，例如释放掉所有在block中拷贝的变量等。

## 4.总结

以上内容是我学习各大师的文章后对自己学习情况的一个记录，其中有部分文字和代码示例是来自大师的文章，还有一些自己的理解，如有错误还请大家勘误。

本文首发地址：Objective-C中的Block (<http://www.devtalking.com/articles/you-should-know-block/>)

[objective-c \(/t/objective-c/blogs\)](#)   [ios \(/t/ios/blogs\)](#)

[举报](#)

2 推荐

收藏

上一篇：在Swift中构建布尔类型 (</blog/devtalking/1190000002421145>)

下一篇：Objective-C中的@property (</blog/devtalking/1190000002446181>)

## 讨论区

请先 [登录](#) 后评论



### 最专业的开发者社区

最前沿的技术问答，最纯粹的技术切磋。让你不知不觉中开拓眼界，提高技能，认识更多朋友。

[Google 账号登录 \(/user/oauth/google\)](/user/oauth/google)

[微博账号登录 \(/user/oauth/weibo\)](/user/oauth/weibo)

## 相似文章

Android与iOS的对决 (</blog/news/1190000000338269>) 8 收藏, 1368 浏览

Objective-C浅拷贝和深拷贝 (</blog/channe/1190000000604331>) 1 收藏, 600 浏览

杭州iOS开发者沙龙 第2期 (</blog/devblog/1190000000323328>) 371 浏览

iOS基础（三） (</blog/darkdust/1190000000450609>) 2 收藏, 619 浏览

iOS开发60分钟入门 (</blog/news/1190000000479589>) 13 收藏, 1567 浏览