objc 中的 block

由 ibireme | 2013-11-27 | iOS, 技术

关于block的语法，请使劲戳这里→fuckingblocksyntax.com

这篇文章只记录一下block的实现，和block使用的注意事项。

正文：

1.block的数据结构

首先，关于block的数据结构和runtime是开源的，可以在llvm项目看到，或者下载苹果的libclosure库的源码来看。苹果也提供了在线的代码查看方式，其中包含了很多示例和文档说明。

这两个地方的定义是相同的：

```
struct Block_descriptor_1 {
    uintptr_t reserved;
    uintptr_t size;
};

struct Block_layout {
    void *isa;
    volatile int32_t flags; // contains ref count
    int32_t reserved;
    void (*invoke)(void *, ...);
    struct Block_descriptor_1 *descriptor;
    // imported variables
};
```

在objc中，根据对象的定义，凡是首地址是*isa的结构体指针，都可以认为是对象(id)。这样在objc中，block实际上就算是对象。

为了查看编译器具体的工作，这里可以用clang重写一段代码试试看：

```
void foo_(){
    int i = 2;
    NSNumber *num = @3;

    long (^myBlock)(void) = ^long() {
        return i * num.intValue;
    };

    long r = myBlock();
}
```

上面这是一个很简单的block，捕获了两个变量：一个int，一个NSNumber。

用clang翻译成C++后变出了一大坨代码，看着别扭不贴上来了。为了方便理解，这里稍微简化和调整一下：

```
struct __block_impl {
    void *isa;
    int Flags;
    int Reserved;
    void *FuncPtr;
};

struct __foo_block_desc_0 {
    size_t reserved;
    size_t Block_size;
    void (*copy)(struct __foo_block_impl_0*, struct __foo_bl
ock_impl_0*);
    void (*dispose)(struct __foo_block_impl_0*);
};

//myBlock的数据结构定义
struct __foo_block_impl_0 {
    struct __block_impl impl;
    struct __foo_block_desc_0* Desc;
    int i;
    NSNumber *num;
};

//block数据的描述
static struct __foo_block_desc_0 __foo_block_desc_0_DATA = {
    0,
    sizeof(struct __foo_block_impl_0),
    __foo_block_copy_0,
    __foo_block_dispose_0
};

//block中的方法
static long __foo_block_func_0(struct __foo_block_impl_0 *__
cself) {
```

```
    int i = __cself->i; // bound by copy
    NSNumber *num = __cself->num; // bound by copy

    return i * num.intValue;
}

void foo(){
    int i = 2;
    NSNumber *num = @3;

    struct __foo_block_impl_0 myBlockT;
    struct __foo_block_impl_0 *myBlock = &myBlockT;
    myBlock->impl.isa = &_NSConcreteStackBlock;
    myBlock->impl.Flags = 570425344;
    myBlock->impl.FuncPtr = __foo_block_func_0;
    myBlock->Desc = &__foo_block_desc_0_DATA;
    myBlock->i = i;
    myBlock->num = num;

    long r = myBlock->impl.FuncPtr(myBlock);
}
```

编译器会根据block捕获的变量，生成具体的结构体定义。block内部的代码将会提取出来，成为一个单独的C函数。创建block时，实际就是在方法中声明一个struct，并且初始化该struct的成员。而执行block时，就是调用那个单独的C函数，并把该struct指针传递过去。

block中包含了被引用的自由变量(由struct持有)，也包含了控制成分的代码块(由函数指针持有)，符合闭包(closure)的概念。

2.block的Copy

block中的isa指向的是该block的Class。在block runtime中，定义了6种类：

_NSConcreteStackBlock　　栈上创建的block
_NSConcreteMallocBlock　堆上创建的block
_NSConcreteGlobalBlock　作为全局变量的block
_NSConcreteWeakBlockVariable
_NSConcreteAutoBlock
_NSConcreteFinalizingBlock

其中我们能接触到的主要是前3种，后三种用于GC不再讨论..

上面代码可以看到，当struct第一次被创建时，它是存在于该函数的栈帧上的，其Class是固定的_NSConcreteStackBlock。其捕获的变量是会赋值到结构体的成员上，所以当block初始化完成后，捕获到的变量不能更改。

当函数返回时，函数的栈帧被销毁，这个block的内存也会被清除。所以在函数结束后仍然需要这个block时，就必须用Block_copy()方法将它拷贝到堆上。这个方法的核心动作很简单：申请内存，将栈数据复制过去，将Class改一下，最后向捕获到的对象发送retain，增加block的引用计数。详细代码可以直接点这里查看。

```
struct Block_layout *result = malloc(aBlock->descriptor->siz
e);
memmove(result, aBlock, aBlock->descriptor->size);
result->isa = _NSConcreteMallocBlock;
_Block_call_copy_helper(result, aBlock);
return result;
```

3.__block类型的变量

默认block捕获到的变量，都是赋值给block的结构体的，相当于const不可改。为了让block能访问并修改外部变量，需要加上__block修饰词。

举个例子：

```
void foo(){
    __block int i = 3;
    void(^myBlock)(void) = ^{
        i *= 2;
    };
    myBlock();
}
```

让clang重写一下：

```
struct Block_byref { //Block_private.h中的定义
    void *isa;
    struct Block_byref *forwarding;
    volatile int32_t flags; // contains ref count
    uint32_t size;
};

//__block count的实现
struct __Block_byref_count_0 {
    void *__isa;
    __Block_byref_count_0 *__forwarding;
    int __flags;
    int __size;
    int count;
};

void foo_(){
    __attribute__((__blocks__(byref))) __Block_byref_count_0
count = {(void*)0,(__Block_byref_count_0 *)&count, 0, sizeof
(__Block_byref_count_0), 1};

    void(*myBlock)(void) = (void (*)())&__foo__block_impl_0(
(void *)__foo__block_func_0, &__foo__block_desc_0_DATA, (__B
lock_byref_count_0 *)&count, 570425344);

    ((void (*)(__block_impl *))((__block_impl *)myBlock)->Fu
ncPtr)((__block_impl *)myBlock);
}
```

哗～一下子变出来一坨东西。就因为加了个__block，原本的int值的位置变成了一个struct（struct __Block_byref）。这个struct的首地址为同样为*isa。

正是如此，这个值才能被block共享、并且不受栈帧生命周期的限制、在block被copy后，能够随着block复制到堆上。

4.使用注意事项

block对变量的捕获规则：

1.静态存储区的变量：例如全局变量、方法中的static变量
引用，可修改。

2.block接受的参数
传值，可修改，和一般函数的参数相同。

3.栈变量 (被捕获的上下文变量)
const，不可修改。 当block被copy后，block会对 id类型的变量产生强引用。
每次执行block时,捕获到的变量都是最初的值。

4.栈变量 (有__block前缀)
引用，可以修改。如果时id类型则不会被block retain,必须手动处理其内存管理。
如果该类型是C类型变量，block被copy到heap后,该值也会被挪动到heap

注意1.内存

Block_copy()和Block_release()必须一一匹配，否则会内存泄漏或crash。

__block这个修饰词会将原本的简单类型转化为较大的struct，这会给内存、调用带来额外的开销，使用时需要注意。

注意2.ARC

在开启ARC后，block的内存会比较微妙。ARC会自动处理block的内存，不用手动copy/release。
但是，和非ARC的情况有所不同：

```
void (^aBlock)(void);
aBlock = ^{ printf("ok"); };
```

block是对象，所以这个aBlock默认是有__strong修饰符的，即aBlock对该block有strong references。即aBlock在被赋值的那一刻，这个block会被copy。所以，ARC开启后，所能接触到的block基本都是在堆上的。。

```
void (^aBlock)(void) = nil;
if (!aBlock) {
    aBlock = ^{ printf("hehe"); };
}
//block此时block已经被释放,该处留下了一个dangling pointer
aBlock();
```

上面这个例子，如果是非ARC时，block还在栈帧上，所以没问题。但开启ARC后，block会被先copy到堆上，然后再被释放，这里就会crash了。所以这时就必须手动调用Block_copy了。苹果建议尽量避免这种情况。

注意3.循环引用

当block被copy之后(如开启了ARC、或把block放入dispatch queue)，该block对它捕获的对象产生strong references (非ARC下是retain),
所以有时需要避免block copy后产生的循环引用。

如果用self引用了block，block又捕获了self，这样就会有循环引用。
因此，需要用weak来声明self

```
- (void)configureBlock {
    XYZBlockKeeper * __weak weakSelf = self;
    self.block = ^{
```

```
        [weakSelf doSomething]; //捕获到的是弱引用
    }
}
```

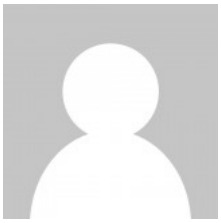
如果捕获到的是当前对象的成员变量对象，同样也会造成对self的引用，同样也要避免。

```
- (void)configureBlock {
    id tmpIvar = _ivar; //临时变量,避免了self引用
    self.block = ^{
        [tmpIvar msg];
    }
}
```

为了避免循环引用，可以这样理解block：block就是一个对象，它捕获到的值就是这个对象的@property(strong)。这样在遇到问题时，就能迅速确定是否有循环引用了。Xcode5已经能自动发现这种问题了，不错～

PS：Pro Multithreading and Memory Management for iOS and OS X 这是一本好书，强烈推荐。
PSS：后来才发现原来这是本日文原版书，并且有中文版翻译。名字叫做”Objective-C高级编程：iOS与OS X多线程和内存管理”。名字差那么多啊！！唉。。买到中文版才发现之前看过。。

7 评论



markgz 在 2015 年 6 月 1 日 的 下午 4:45

回复

注意2.ARC
但开启ARC后，block会被先copy到堆上，然后再被释放，这里就会crash了。我实验了一下，发现并没有crash.

xiexie 在 2015 年 7 月 27 日 的 上午 9:51

回复

void (^aBlock)(void) = nil;

```
if (!aBlock) {
aBlock = ^{ printf(“hehe”); };
}
```

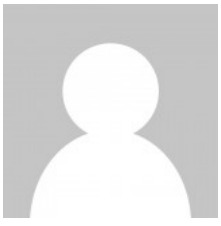
aBlock();//此处并不会crash，因为aBlock对copy到堆上的block是强引用，block并不会被释放。

回复

ibireme 在 2015 年 7 月 27 日 的 下午 1:25

确实是不会crash了。。
苹果在文档里说这种方式是应该避免的，那篇文档和这篇博客都已经过期了，看看就好。 ;-)

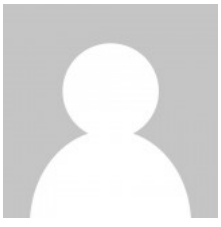
回复



君 在 2017 年 3 月 21 日 的 下午 1:28

还是更新一下吧

回复



liangzai123 在 2017 年 9 月 11 日 的 上午 11:30

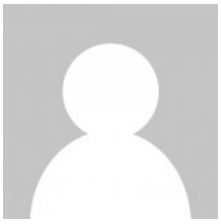
如果此处crash ，就跟这句[block是对象，所以这个aBlock默认是有__strong修饰符的，即aBlock对该block有strong references。即aBlock在被赋值的那一刻，这个block会被copy]矛盾了

回复

liangzai 在 2017 年 9 月 11 日 的 上午 11:32

回复

可以看看这个
http://www.jianshu.com/p/2ad720287ef4,
arc确实微妙



Too 在 2016 年 8 月 3 日 的 上午 10:44

回复

void (^aBlock)(void);
aBlock = ^{ printf("ok"); };
block是对象，所以这个aBlock默认是有__strong修饰符的，即aBlock对该block有strong references。即aBlock在被赋值的那一刻，这个block会被copy。所以，ARC开启后，所能接触到的block基本都是在堆上的。。

上面的在arc下 是global block吧。引用外部会到stack，strong refrence然后拷贝到堆区..

引用/广播

1. BAT 面试指南 | 神刀安全网 – […] block […]
2. 让 BAT 的 Offer 不再难拿 — kyoucr博客 – […] block […]
3. 让 BAT的Offer不再难拿 | Codeba – […] block […]
4. 让 BAT 的 Offer 不再难拿 — 项目经验积累与分享 – […] block […]
5. BAT 面试指南–IT文库 – […] block […]
6. 让 BAT 的 Offer 不再难拿 | 秀品折 – […] block […]
7. Block 到底啥时候会崩溃–IT文库 – […] objc 中的 block […]
8. Block 到底啥时候崩溃? – 莹莹之色 – […] objc 中

的 block […]

9. Block 到底啥时候崩溃? –IT文库    – […] objc 中的 block […]

10. Block 到底啥时候崩溃? | 恰同学少年–不称    – […] objc 中的 block […]

关于

伽蓝之堂——

一只魔法师的工坊

相关链接

Github

Weibo

Twitter

LinkedIn

DeviantART

功能

登录

文章RSS

评论RSS

WordPress.org