Blocks深入理解和详解

介绍

Block是C级语法和运行时特性。它们类似于标准C函数，但是除了可执行代码之外，它们还可能包含对自动(堆栈)或托管(堆)内存的变量绑定。因此，Block可以维护一组状态(数据)，它可以用来在执行时影响行为。

您可以使用Blocks来组合函数表达式，这些表达式可以被传递给API，可选地存储，并由多个线程使用。Block对于回调来说特别有用，因为块包含在回调上执行的代码和执行过程中需要的数据。

可以在GCC和Clang中使用OS X v10.6 Xcode开发工具。您可以使用OS X v10.6和之后的模块，以及随后的iOS 4.0。Block运行时是开源的，可以在LLVM的编译器-rt子项目存储库中找到。block也被提交给C标准工作组作为N1370:苹果对C的扩展，因为Objective-C和c++都是从C派生出来的，block被设计用于与所有三种语言(以及Objective-C++)一起工作。语法反映了这个目标。

概念普及

- **Block本质是Objective-C的对象，不是函数指针（这个有点混淆，网上普遍都说是函数指针）**
- **Block类型变量本质是函数指针(如下声明其实在编译成C语言源码的时候就是一个函数指针)**
- 截获自动变量 所有的变量都会截获吗？比如全局变量、静态变量。其实截获的只是局部变量而已
- `__block` 变量加了它为啥就能被修改了？很多人也许会说是传入了地址，这种说法很片面的，如果是这样的其实完全不用加 `__block` ,苹果帮你做了就行了呀。其实__block之后翻译成C语言源码之后，变量会变成一个对象，该对象存储了原来变量的地址，函数传入的是一个对象。

探寻Block本质

通过LLVM的编译器-rt子项目存储库中找到runtime下的Block源码

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

当我们声明一个Block的时候，编译器其实会将block转换成以上struct结构体。 其中isa指向的是Block具体的类。有如下6中，不过其中`StackBlock`、`MallocBlock`、`GlobalBlock`是比较常见的

```
/* the raw data space for runtime classes for blocks */
/* class+meta used for stack, malloc, and collectable based blocks
*/
BLOCK_EXPORT void * _NSConcreteStackBlock[32];
BLOCK_EXPORT void * _NSConcreteMallocBlock[32];
BLOCK_EXPORT void * _NSConcreteAutoBlock[32];
BLOCK_EXPORT void * _NSConcreteFinalizingBlock[32];
BLOCK_EXPORT void * _NSConcreteGlobalBlock[32];
BLOCK_EXPORT void * _NSConcreteWeakBlockVariable[32];
```

invoke函数指针则是对应Objective-C中代码的具体实现

看到这里不知大家有没有想到runtime中`objc_object`的isa呢？ 其实两个原理是一样的。这里就不具体介绍`objc_object`

所以Block即为Objective-C的对象

Block类型变量

Block的类型变量声明如下:

> int (^blk) (int);

我们声明一个Block类型变量并且赋值

```
int (^blk) (int) = ^(int count){return count+1};
```

然后我们通过如下

> xcrun -sdk iphonesimulator11.2 clang -rewrite-objc -F /Applications/Xcode\2.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS11.2.sdk/System/Library/Frameworks ViewController.m

我们可以得到以下源码:

```
//对应的Block具体struct结构体
struct __ViewController__viewDidLoad_block_impl_0 {
  struct __block_impl impl;
  struct __ViewController__viewDidLoad_block_desc_0* Desc;
  __ViewController__viewDidLoad_block_impl_0(void *fp, struct
__ViewController__viewDidLoad_block_desc_0 *desc, int flags=0) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
//Block方法的具体实现
static int __ViewController__viewDidLoad_block_func_0(struct
__ViewController__viewDidLoad_block_impl_0 *__cself, int count) {

        return count+1;
    }

static struct __ViewController__viewDidLoad_block_desc_0 {
  size_t reserved;
  size_t Block_size;
} __ViewController__viewDidLoad_block_desc_0_DATA = { 0,
sizeof(struct __ViewController__viewDidLoad_block_impl_0)};

static void _I_ViewController_viewDidLoad(ViewController * self,
SEL _cmd) {
```

```
    ((void (*)(__rw_objc_super *, SEL))(void *)objc_msgSendSuper)
((__rw_objc_super){(id)self,
(id)class_getSuperclass(objc_getClass("ViewController"))},
sel_registerName("viewDidLoad"));
    int (*blk) (int) = ((int (*)
(int))&__ViewController__viewDidLoad_block_impl_0((void
*)__ViewController__viewDidLoad_block_func_0,
&__ViewController__viewDidLoad_block_desc_0_DATA));
    ((int (*)(__block_impl *, int))((__block_impl *)blk)->FuncPtr)
((__block_impl *)blk, 2);
}
```

其中`int (*blk) (int)`则就是对应我们的Block类型变量，这里就就可以看到了Block类型变量的本质了，其实就是C语言的函数指针

截获自动变量

说截获自动变量之前我们先看以下代码

```
int tmp = 2;
int (^blk) (int) = ^(int count){
    return count+tmp;
};
tmp = 3;
int result = blk(2);
NSLog(@"%d",result);
```

以上代码会打印出多少呢？5还是4？？ 正确答案是4

为什么是4呢？其实就是因为Block截获自动变量的原因

```
struct __ViewController__viewDidLoad_block_impl_0 {
  struct __block_impl impl;
  struct __ViewController__viewDidLoad_block_desc_0* Desc;
  int tmp;
  __ViewController__viewDidLoad_block_impl_0(void *fp, struct
__ViewController__viewDidLoad_block_desc_0 *desc, int _tmp, int
flags=0) : tmp(_tmp) {
    impl.isa = &_NSConcreteStackBlock;
```

```
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
static int __ViewController__viewDidLoad_block_func_0(struct
__ViewController__viewDidLoad_block_impl_0 *__cself, int count) {
  int tmp = __cself->tmp; // bound by copy

        return count+tmp;
    }

static struct __ViewController__viewDidLoad_block_desc_0 {
  size_t reserved;
  size_t Block_size;
} __ViewController__viewDidLoad_block_desc_0_DATA = { 0,
sizeof(struct __ViewController__viewDidLoad_block_impl_0)};

static void _I_ViewController_viewDidLoad(ViewController * self,
SEL _cmd) {
    ((void (*)(__rw_objc_super *, SEL))(void *)objc_msgSendSuper)
((__rw_objc_super){(id)self,
(id)class_getSuperclass(objc_getClass("ViewController"))},
sel_registerName("viewDidLoad"));
    int tmp = 2;
    int (*blk) (int) = ((int (*)
(int))&__ViewController__viewDidLoad_block_impl_0((void
*)__ViewController__viewDidLoad_block_func_0,
&__ViewController__viewDidLoad_block_desc_0_DATA, tmp));
    tmp = 3;
    int result = ((int (*)(__block_impl *, int))((__block_impl
*)blk)->FuncPtr)((__block_impl *)blk, 2);
    NSLog((NSString
*)&__NSConstantStringImpl__var_folders_6n_mdf6rn0d5f5cw6r1h2620h3w0
000gn_T_ViewController_b80e84_mi_0,result);
}`__ViewController__viewDidLoad_block_impl_0`
```

通过以下代码可以看出,当我们在给Block类型变量赋值的时候，tmp变量同时被传入，并且被保存到了`__ViewController__viewDidLoad_block_impl_0`的struct中。这时候其实就是截获了自动变量，由于已经在struct类中保存了一份，即使后边更改，也不会影响Block截获的值。

```
int (*blk) (int) = ((int (*)
(int))&__ViewController__viewDidLoad_block_impl_0((void
```

```
*)__ViewController__viewDidLoad_block_func_0,
&__ViewController__viewDidLoad_block_desc_0_DATA, tmp));
```

为什么要对局部变量进行截获呢？而全局变量和静态变量不需要截获，并且修改的的时候也不需要加`__block`呢？

主要原因就是变量的生命周期。局部变量在代码块执行结束之后就会被释放，但是Block不一定在此时释放。所以就会出现变量超过生命周期的现象，此时对局部变量进行截获，即使局部变量被释放，但是Block同样还是可以正常使用的。因为全局变量和静态变量的释放时间肯定不会在Block之前，所以不必对他们进行截获。

> 全局变量和静态变量存储在全局数据区；局部变量存储在栈中

__block变量存储域探究

不知道大家有没有想过一个问题，为什么需要__block呢？如果没有__block难道就修改不了变量了吗？

我们先看一下加了__block编译器给我们做了啥

```
struct __Block_byref_tmpBlock_0 {
  void *__isa;
__Block_byref_tmpBlock_0 *__forwarding;
 int __flags;
 int __size;
 int tmpBlock;
};

struct __ViewController__viewDidLoad_block_impl_0 {
  struct __block_impl impl;
  struct __ViewController__viewDidLoad_block_desc_0* Desc;
  __Block_byref_tmpBlock_0 *tmpBlock; // by ref
  __ViewController__viewDidLoad_block_impl_0(void *fp, struct
__ViewController__viewDidLoad_block_desc_0 *desc,
__Block_byref_tmpBlock_0 *_tmpBlock, int flags=0) :
tmpBlock(_tmpBlock->__forwarding) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
```

```
};
static int __ViewController__viewDidLoad_block_func_0(struct
__ViewController__viewDidLoad_block_impl_0 *__cself, int count) {
  __Block_byref_tmpBlock_0 *tmpBlock = __cself->tmpBlock; // bound
by ref

        (tmpBlock->__forwarding->tmpBlock) = 100;
        return count+(tmpBlock->__forwarding->tmpBlock);
    }
static void __ViewController__viewDidLoad_block_copy_0(struct
__ViewController__viewDidLoad_block_impl_0*dst, struct
__ViewController__viewDidLoad_block_impl_0*src)
{_Block_object_assign((void*)&dst->tmpBlock, (void*)src->tmpBlock,
8/*BLOCK_FIELD_IS_BYREF*/);}

static void __ViewController__viewDidLoad_block_dispose_0(struct
__ViewController__viewDidLoad_block_impl_0*src)
{_Block_object_dispose((void*)src->tmpBlock,
8/*BLOCK_FIELD_IS_BYREF*/);}

static struct __ViewController__viewDidLoad_block_desc_0 {
  size_t reserved;
  size_t Block_size;
  void (*copy)(struct __ViewController__viewDidLoad_block_impl_0*,
struct __ViewController__viewDidLoad_block_impl_0*);
  void (*dispose)(struct
__ViewController__viewDidLoad_block_impl_0*);
} __ViewController__viewDidLoad_block_desc_0_DATA = { 0,
sizeof(struct __ViewController__viewDidLoad_block_impl_0),
__ViewController__viewDidLoad_block_copy_0,
__ViewController__viewDidLoad_block_dispose_0};

static void _I_ViewController_viewDidLoad(ViewController * self,
SEL _cmd) {
    ((void (*)(__rw_objc_super *, SEL))(void *)objc_msgSendSuper)
((__rw_objc_super){(id)self,
(id)class_getSuperclass(objc_getClass("ViewController"))},
sel_registerName("viewDidLoad"));
    __attribute__((__blocks__(byref))) __Block_byref_tmpBlock_0
tmpBlock = {(void*)0,(__Block_byref_tmpBlock_0 *)&tmpBlock, 0,
sizeof(__Block_byref_tmpBlock_0), 2};
    int (*blk) (int) = ((int (*)
(int))&__ViewController__viewDidLoad_block_impl_0((void
*)__ViewController__viewDidLoad_block_func_0,
&__ViewController__viewDidLoad_block_desc_0_DATA,
(__Block_byref_tmpBlock_0 *)&tmpBlock, 570425344));
    int result = ((int (*)(__block_impl *, int))((__block_impl
*)blk)->FuncPtr)((__block_impl *)blk, 2);
```

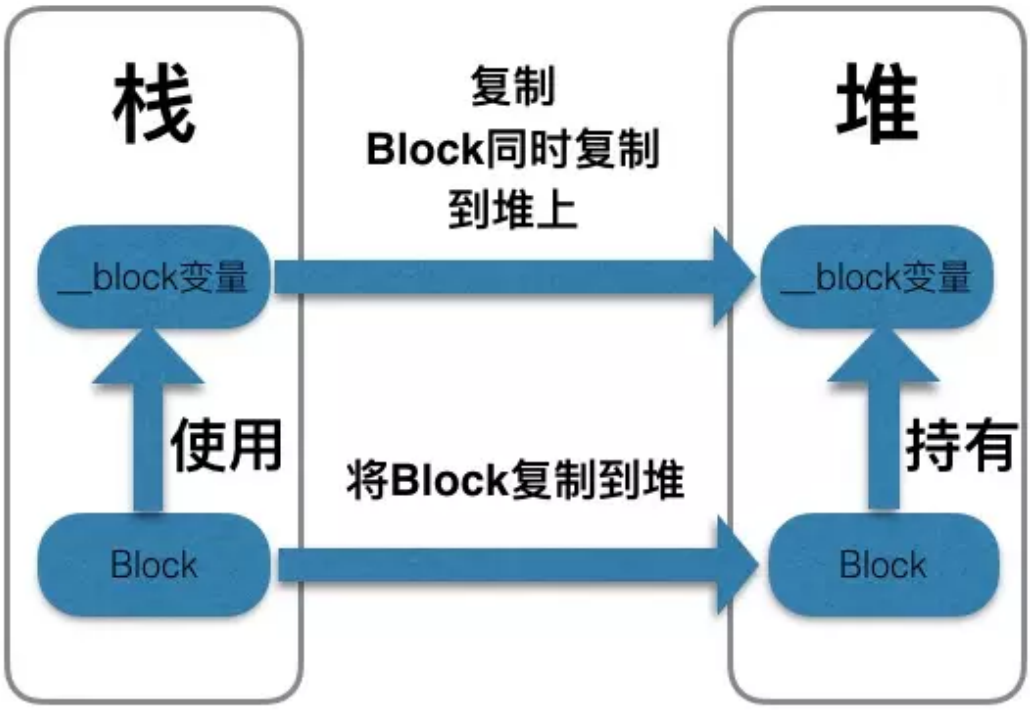
```
    NSLog((NSString
*)&__NSConstantStringImpl__var_folders_6n_mdf6rn0d5f5cw6r1h2620h3w0
000gn_T_ViewController_a1c583_mi_0,result);
}
```

这时候大家可以与前边没有加 `__block` 的代码进行比较，两者的差别在哪里。 其实大家应该很容易发现加了 `__block` 之后，变量形成了一个struct，这个struct中保存了变量的值，同时还有一个 `__forwarding` 。这个 `__forwarding` 其实就是为什么需要__block的关键。

Block从栈复制到堆的时候，__block变量也会受到影响。如下:

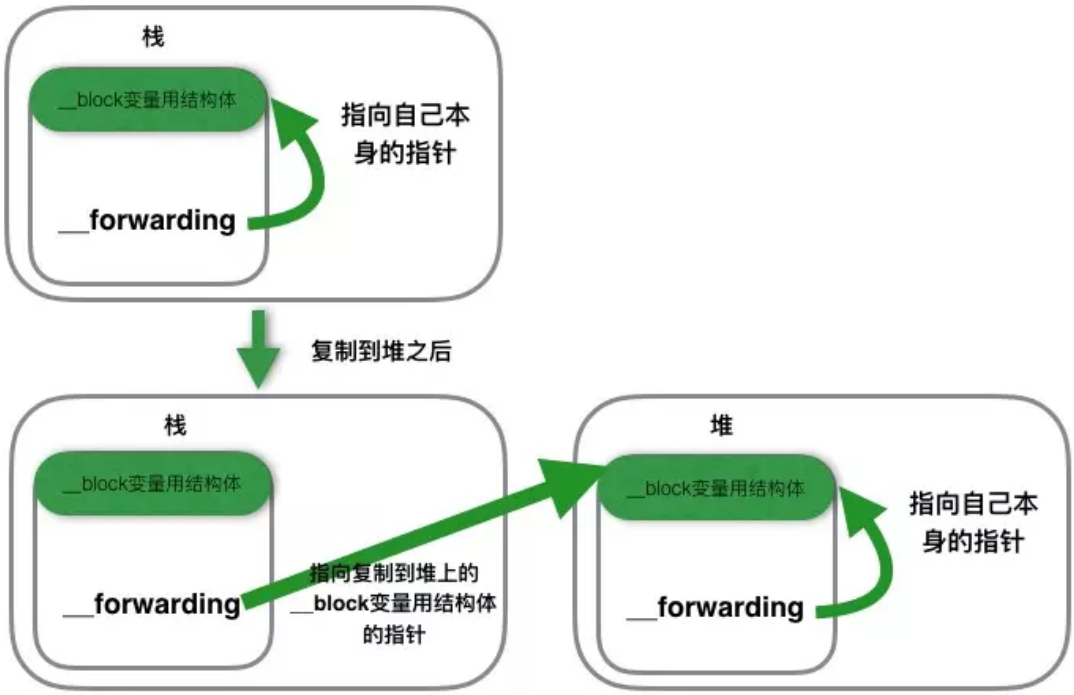
__block变量的配置存储域	Block从栈到堆时的影响
栈	从栈复制到堆并被Block持有
堆	被Block持有

ARC下，Block如果是栈的话，默认会copy到堆上。此时所使用的__block变量同时也会从栈被复制到堆上如下图



那如果Block在堆上了，我们在Block中修改了变量，怎么让栈上的变量也同时能正确访问呢？这其实就是`__forwarding`功劳了。

`__block`变量初始化的时候`__forwarding`是指向本身自己的。当`__block`变量从栈复制到堆上的时候，此时会将`__forwarding`的值替换为复制到目标堆上的`__block`变量用结构体实例的地址。如下图：



通过此功能，无论是在Block语法中、Block语法外使用`__block`变量，还是`__block`变量配置在栈上或堆上，都可以顺利地访问一个`__block`变量。 到这里大家应该明白了"`__block`加了之后，是把变量的地址传入Block"的说法是很片面的吧啦