

iOS中block的探究

发布于：2012-07-18 11:12 阅读数：14416

“Block是iOS4.0+ 和Mac OS X 10.6+ 引进的对C语言的扩展，用来实现匿名函数的特性。”

阅读器

iOS Objective-C block

文/CocoaChina社区会员casual0402

```
/* ----- */
[0. Brief introduction of block]
Block是iOS4.0+ 和Mac OS X 10.6+ 引进的对C语言的扩展，用来实现匿名函数的特性。
用维基百科的话来说，Block是Apple Inc.为C、C++以及Objective-C添加的特性，使得这些语言可以用类lambda表达式的语法来创建闭包。
用Apple文档的话来说，A block is an anonymous inline collection of code, and sometimes also called a "closure".

关于闭包，我觉得阮一峰的一句话解释简洁明了：闭包就是能够读取其它函数内部变量的函数。
这个解释用到block来也很恰当：一个函数里定义了一个block，这个block可以访问该函数的内部变量。
一个简单的Block示例如下：
int (^maxBlock)(int, int) = ^(int x, int y) { return x > y ? x : y; };
如果用Python的lambda表达式来写，可以写成如下形式：
f = lambda x, y : x if x > y else y
不过由于Python自身的语言特性，在def定义的函数体中，可以很自然地再用def语句定义内嵌函数，因为这些函数本质上都是对象。
如果用BNF来表示block的上下文无关文法，大致如下：
block_expression ::= ^ block_declare block_statement
block_declare ::= block_return_type block_argument_list
block_return_type ::= return_type | 空
block_argument_list ::= argument_list | 空

/* ----- */
[1. Why block]
Block除了能够定义参数列表、返回类型外，还能够获取被定义时的词法范围内的状态（比如局部变量），并且在一定条件下（比如使用__block变量）能够修改这些状态。此外，这些可修改的状态在相同词法范围内的多个block之间是共享的，即便出了该词法范围（比如栈展开，出了作用域），仍可以继续共享或者修改这些状态。

通常来说，block都是一些简短代码片段的封装，适用作工作单元，通常用来做并发任务、遍历、以及回调。
比如我们可以在遍历NSArray时做一些事情：
- (void)enumerateObjectsUsingBlock:(void (^)(id obj, NSUInteger idx, BOOL *stop))block;
其中将stop设为YES，就跳出循环，不继续遍历了。
而在很多框架中，block越来越经常被用作回调函数，取代传统的回调方式。
用block作为回调函数，可以使得程序员在写代码更顺畅，不用中途跑到另一个地方写一个回调函数，有时还要考虑这个回调函数放在哪里比较合适。采用block，可以在调用函数时直接写后续处理代码，将其作为参数传递过去，供其任务执行结束时回调。
```

开发者通道

排行榜

代码库

图书库

网站库

发码区

工具库

招聘区

外包区

问答区

最近更新

1 iOS安全：黑客与反黑客 2012-12-20

2 使用Xcode和Instruments调试解决iC 2012-12-04

3 优秀开源代码解读：JS与iOS Native 2012-11-26

4 CocoaPods：一个Objective-C第三... 2012-11-23

5 iOS开发——图片转PDF的实现方法 2012-11-22

6 iOS 6新特性UIActivityIndicatorView 2012-11-16

7 iOS 6中NSString新用法 2012-11-14

8 iOS中arc的设置与使用 2012-10-16

9 CGContext小记 2012-08-31

10 论坛用户mhmwadmiOS开发心得分... 2012-08-16

推荐内容

热点内容

iOS中arc的设置与使用

CGContext小记

另一个好处，就是采用block作为回调，可以直接访问局部变量。比如我要在一批用户中修改一个用户的name，修改完成后通过回调更新对应用户的单元格UI。这时候我需要知道对应用户单元格的index，如果采用传统回调方式，要嘛需要将index带过去，回调时再回传过来；要嘛通过外部作用域记录当前操作单元格的index（这限制了一次只能修改一个用户的name）；要嘛遍历找到对应用户。而使用block，则可以直接访问单元格的index。

这份文档中提到block的几种适用场合：

- 任务完成时回调
- 处理消息监听回调处理
- 错误回调处理
- 枚举回调
- 视图动画、变换
- 排序

```
/* ----- */

[2. About __block_impl]
```

Clang提供了中间代码展示的选项供我们进一步了解block的原理。

以一段很简单的代码为例：

```
signin:block Jason$ cat block0.c
#include <stdio.h>

int main()
{
    return 0;
}
```

使用-rewrite-objc选项编译：

```
signin:block Jason$ clang -rewrite-objc block0.c
```

得到一份block0.cpp文件，在这份文件中可以看到如下代码片段：

```
#ifndef BLOCK_IMPL
#define BLOCK_IMPL
struct __block_impl {
    void *isa;
    int Flags;
    int Reserved;
    void *FuncPtr;
};
```

从命名可以看出这是block的实现，并且得知block在Clang编译器前端得到实现，可以生成C中间代码。很多语言都可以只实现编译器前端，生成C中间代码，然后利用现有的很多C编译器后端。

从结构体的成员可以看出，Flags、Reserved可以先略过，isa指针表明了block可以是一个NSObject，而FuncPtr指针显然是block对应的函数指针。

由此，揭开了block的神秘面纱。

不过，block相关的变量放哪里呢？上面提到block可以capture词法范围内（或者说是外层上下文、作用域）的状态，即便是出了该范围，仍然可以修改这些状态。这是如何做到的呢？

```
/* ----- */

[3. Implementation of a simple block]
```

先看一个只输出一句话的block是怎么样的。

```
signin:block Jason$ cat block1.c
#include <stdio.h>

int main()
{
    ListData *p = malloc(sizeof(ListData));
    void (^blk)(void) = ^{ printf("Hello,block!\n"); };
    blk();
}
```

生成中间代码，得到片段如下：



Objective-C 内存管理精髓



iOS新手入门视频教程



iOS中block的探究

```
#include <stdio.h>

struct __main_block_impl_0 {
    struct __block_impl impl;
    struct __main_block_desc_0* Desc;
    __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, int flags=0) {
        impl.isa = 6_NSCConcreteStackBlock;
        impl.Flags = flags;
        impl.FuncPtr = fp;
        Desc = desc;
    }
};

static void __main_block_func_0(struct __main_block_impl_0 *__cself) {
    printf("Hello,block!\n");
}

static struct __main_block_desc_0 {
    unsigned long reserved;
    unsigned long Block_size;
    __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0)};
} __main_block_desc_0_DATA;

int main()
{
    void (*blk)(void) = (void (*)(void))__main_block_impl_0((void *)__main_block_func_0, &__main_block_desc_0_DATA);
    ((void (*)(struct __block_impl *))((struct __block_impl *)blk)->FuncPtr)((struct __block_impl *)blk);
}

sigin:block Jason$
```

首先出现的结构体就是\_\_main\_block\_impl\_0，可以看出是根据所在函数（main函数）以及出现序列（第0个）进行命名的。如果是全局block，就根据变量名和出现序列进行命名。\_\_main\_block\_impl\_0中包含了两个成员变量和一个构造函数，成员变量分别是\_\_block\_impl结构体和描述信息Desc，之后在构造函数中初始化block的类型信息和函数指针等信息。

接着出现的是\_\_main\_block\_func\_0函数，即block对应的函数体。该函数接受一个\_\_cself参数，即对应的block自身。

再下面是\_\_main\_block\_desc\_0结构体，其中比较有价值的信息是block大小。

最后就是main函数中对block的创建和调用，可以看出执行block就是调用一个以block自身作为参数的函数，这个函数对应着block的执行体。

这里，block的类型用\_NSCConcreteStackBlock来表示，表明这个block位于栈中。同样地，还有\_NSCConcreteMallocBlock和\_NSCConcreteGlobalBlock。

由于block也是NSObject，我们可以对其进行retain操作。不过在将block作为回调函数传递给底层框架时，底层框架需要对其copy一份。比方说，如果将回调block作为属性，不能用retain，而要用copy。我们通常会将block写在栈中，而需要回调时，往往回调block已经不在栈中了，使用copy属性可以将block放到堆中。或者使用Block\_copy()和Block\_release()。

/\* ----- \*/

#### [4. Capture local variable]

再看一个访问局部变量的block是怎样的。

```
sigin:block Jason$ cat block2.c
#include <stdio.h>

int main()
{
    int i = 1024;
    void (^blk)(void) = ^{ printf("%d\n", i); };
    blk();
    return 0;
}
```

生成中间代码，得到片段如下：

```
#include <stdio.h>
// 一个 __cself 参数，即对应的 block。
// 的信息是 block 大小。
// block 就是调用一个以 block 自身作为参数的函数。这个函数对应着 block 的执行体。
struct __main_block_impl_0 {
    struct __block_impl impl;
    struct __main_block_desc_0* Desc;
    int i;
    __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, int _i, int flags=0) : i(_i) {
        impl.isa = 6_NSCConcreteStackBlock;
        impl.Flags = flags;
        impl.FuncPtr = fp;
        Desc = desc;
    }
};

static void __main_block_func_0(struct __main_block_impl_0 *__cself) {
    int i = __cself->i; // bound by copy
    printf("%d\n", i);
}

static struct __main_block_desc_0 {
    unsigned long reserved;
    unsigned long Block_size;
    __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0)};
} __main_block_desc_0_DATA;

int main()
{
    int i = 1024;
    void (*blk)(void) = (void (*)(void))__main_block_impl_0((void *)__main_block_func_0, &__main_block_desc_0_DATA, i);
    ((void (*)(struct __block_impl *))((struct __block_impl *)blk)->FuncPtr)((struct __block_impl *)blk);
    return 0;
}
```

可以看出这次的block结构体\_\_main\_block\_impl\_0多了个成员变量i，用来存储使用到的局部变量i（值为1024）；并且此时可以看到\_\_cself参数的作用，类似C++中的this和Objective-C的self。

如果我们尝试修改局部变量i，则会得到如下错误：

```
siqin:block Jason$ vi block2.c
siqin:block Jason$ clang -rewrite-objc block2.c
block2.c:6:27: error: variable is not assignable (missing __block type specifier)
    void (^blk)(void) = ^{ i = 0; printf("%d\n", i); };
                          ^~ ^
1 error generated.
siqin:block Jason$
```

错误信息很详细，既告诉我们变量不可赋值，也提醒我们要使用\_\_block类型标识符。

为什么不能给变量i赋值呢？

因为main函数中的局部变量i和函数\_\_main\_block\_func\_0不在同一个作用域中，调用过程中只是进行了值传递。当然，在上面代码中，我们可以通过指针来实现局部变量的修改。不过这是由于在调用\_\_main\_block\_func\_0时，main函数栈还没展开完成，变量i还在栈中。但是在很多情况下，block是作为参数传递以供后续回调执行的。通常在这些情况下，block被执行时，定义时所在的函数栈已经被展开，局部变量已经不在栈中了（block此时在哪里？），再用指针访问就.....。

所以，对于auto类型的局部变量，不允许block进行修改是合理的。

(47)

共2页: 上一页 1 2 下一页

猜你喜欢



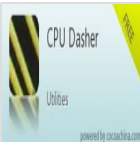
奇葩应用  
Hidden Apps已下



iOS安全: 黑客与反  
黑客



关于block中使用  
self的疑问



iphone sdk 4 -  
block object - 实



Objective-C  
Block语法递归函

- NSRunLoop 概述和原理
  - UIAlertView与UIActionSheet的Block实现方式
- iOS新手入门视频教程
  - MUBlockDelegate: 一个基于Block的通用委托实

现在评论

有事没事留个话：)

0 喜欢 社区



请输入你的评论

140

昵称 (必填) 发布

按时间排序 | 新浪微博 | 腾讯微博

还没有评论内容

友言[?]

<div><p>苹果开发中文站</p></div> <div>网站地图</div> <div>关于我们</div> <div>联系我们</div>	资讯频道	开发者频道	市场频道	下载频道	开发者中心
	开发相关 苹果相关	Mac开发 iPhone开发 新手教学 游戏开发 开发综合 用户体验 iPad开发	AppStore研究 会员作品 软件销售 市场推广 上线经验 案例分析	教学视频 电子文档 源码下载	应用排行 补充个人信息 华人应用大全
	友情链接				

