Blocks编程要点

原著： Apple Inc.

翻译： 謝業蘭【老狼】

联系： xyl.layne@gmail.com

鸣谢： 有米移动广告平台

CocoaChina 社区

目录

简介

Block 对象是 C 级别的语法和运行时特性。它们和标准 C 函数很类似，但是除了可执行代码外，它们还可能包含了变量自动绑定(栈)或内存托管(堆)。所以一个 block 维护一个状态集（数据），它们可以在执行的时候用来影响程序行为。

你可以用 blocks 来编写函数表达式，这些表达式可以作为 API 使用，或可选的存储，或被多个线程使用。Blocks 作为回调特别有用，因为 block 携带了进行回调所需要的执行代码和执行过程中需要的数据。

Blocks 在 GCC 和 Clang 里面可用，它附带在 Mac OS X v10.6 里面的 Xcode 开发工具里面。你可以在 Mac OS X v10.6 及其之后，和 iOS 4.0 及其之后上面使用 blocks。Blocks 运行时是开源的，你可以在 LLVM’s compiler-rt subproject repository（LLVM 的 RT 编译器的子项目库）里面找到它。Blocks 同样作为标准 C 工作组出现在 N1370:Apple’s Extensions to C(该文档同样包括了垃圾回收机制)。因为 Objective-C 和 C++都是从 C 发展而来，blocks 被设计在三种语言上面使用（也包括 Objective-C++)。(语法反应了这一目标)

你应该阅读该文档来掌握 block 对象是什么和如何在 C,C++或 Objective-C 上面使用它们来让你的程序更高效和更易于维护。

本文档结构

该文档包含了以下几个章节:

- “Blocks入门”提供了一个对blocks的快速的和实际的介绍。
- “概念概述”提供了对blocks概念的介绍。
- “声明和创建Blocks”描述如何声明block变量，并实现该blocks。
- “Blocks和变量”介绍了blocks和变量的交互，并定义__block存储类型修饰符。
- “使用Blocks”说明不同的使用模式。

第一章　Blocks入门

以下部分使用实际的例子帮助你开始使用 Blocks。

1.1　声明和使用一个Block

使用^操作符来来声明一个 block 变量和指示 block 文本的开始。Block 本身的主体被{}包含着，如下面的例子那样(通常使用 C 的 ；符合指示 block 的结束)：

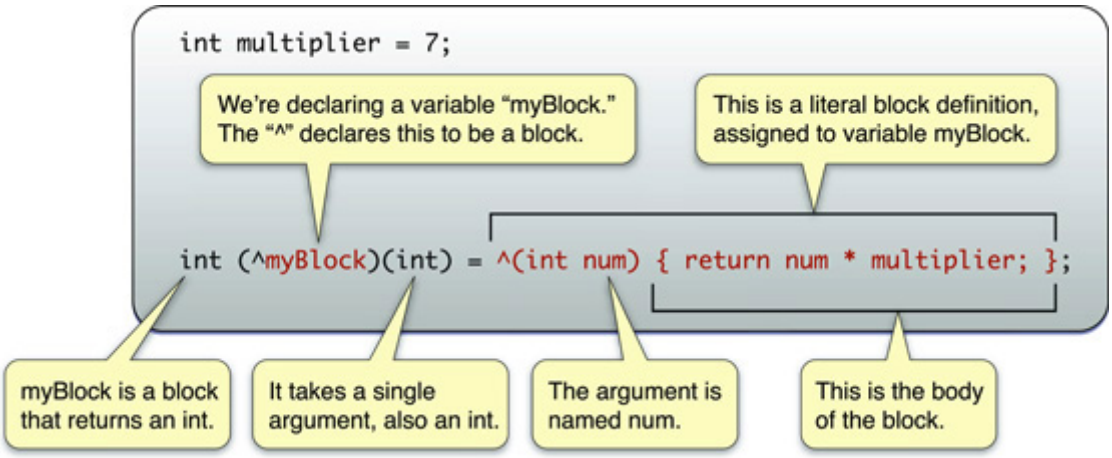
```
int multiplier = 7;

int (^myBlock)(int) = ^(int num) {

    return num * multiplier;

};
```

该示例的解析如下图：



注意 block 可以使用相同作用域范围内定义的变量。

如果你声明一个 block 作为变量，你可以把它简单的作为一个函数使用：

```
int multiplier = 7;

int (^myBlock)(int) = ^(int num) {

    return num * multiplier;

};


printf("%d", myBlock(3));

// prints "21"
```

1.2 直接使用Block

在很多情况下，你不需要声明一个 block 变量；相反你可以简单的写一个内联（inline）的 block 文本，它需要作为一个参数使用。以下的代码使用 qsort_b 函数。qsort_b 和标准 qsort_r 函数类似，但是它使用 block 作为最后一个参数。

```
char *myCharacters[3] = { "TomJohn", "George", "Charles Condomine" };


qsort_b(myCharacters, 3, sizeof(char *), ^(const void *l, const void *r) {
    char *left = *(char **)l;
    char *right = *(char **)r;
    return strncmp(left, right, 1);
});


// myCharacters is now { "Charles Condomine", "George", "TomJohn" }
```

1.3 Cocoa的Blocks

在 Cocoa frameworks 里面有部分方法使用 block 作为参数，通常不是执行一个对象的集合操作，就是在操作完成的时候作为回调使用。下面的例子显示了如何通过 NSArray 的方法 sortedArrayUsingComparator:使用 block。该方法使用一个参数, 即 block。为了举例说明，该情况下 block 被定义为 NSComparator 的局部变量：

```
NSArray *stringsArray = [NSArray arrayWithObjects:
                            @"string 1",
                            @"String 21",
                            @"string 12",
                            @"String 11",
                            @"String 02", nil];


static NSStringCompareOptions comparisonOptions = NSCaseInsensitiveSearch | NSNumericSearch
|
       NSWidthInsensitiveSearch | NSForcedOrderingSearch;
NSLocale *currentLocale = [NSLocale currentLocale];
```

```
NSComparator finderSortBlock = ^(id string1, id string2) {



    NSRange string1Range = NSMakeRange(0, [string1 length]);

    return [string1 compare:string2 options:comparisonOptions range:string1Range
locale:currentLocale];

};



NSArray *finderSortArray = [stringsArray sortedArrayUsingComparator:finderSortBlock];

NSLog(@"finderSortArray: %@", finderSortArray);



/*

Output:

finderSortArray: (

    "string 1",

    "String 02",

    "String 11",

    "string 12",

    "String 21"

)

*/
```

1.4 __block变量

Blocks 的最大一个特色就是可以修改相同作用域的变量。你可以使用__block 存储类型修饰符来给出信号要修改一个变量。改编“Cocoa 的 Blocks”所示的例子，你可以使用一个 block 来计数多少个字符串和 block 中只读变量 currentLocal 相同：

```
NSArray *stringsArray = [NSArray arrayWithObjects:

                    @"string 1",

                    @"String 21", // <-

                    @"string 12",

                    @"String 11",

                    @"Strîng 21", // <-

                    @"Striñg 21", // <-
```

```
                    @"String 02", nil];


NSLocale *currentLocale = [NSLocale currentLocale];

  block NSUInteger orderedSameCount = 0;


NSArray *diacriticInsensitiveSortArray = [stringsArray sortedArrayUsingComparator:^(id
string1, id string2) {


   NSRange string1Range = NSMakeRange(0, [string1 length]);

   NSComparisonResult comparisonResult = [string1 compare:string2
options:NSDiacriticInsensitiveSearch range:string1Range locale:currentLocale];


   if (comparisonResult == NSOrderedSame) {

      orderedSameCount++;

   }

   return comparisonResult;

}];


NSLog(@"diacriticInsensitiveSortArray: %@", diacriticInsensitiveSortArray);

NSLog(@"orderedSameCount: %d", orderedSameCount);


/*

Output:


diacriticInsensitiveSortArray: (

   "String 02",

   "string 1",

   "String 11",

   "string 12",

   "String 21",

   "Str\U00eeng 21",

   "Stri\U00f1g 21"

)
```

```
orderedSameCount: 2

*/
```

这会在“Blocks 和变量”部分里面有更多的讨论。

第二章　概念概述

Block 对象提供了一个使用 C 语言和 C 派生语言（如 Objective-C 和 C++）来创建表达式作为一个特别(ad hoc)的函数。在其他语言和环境中，一个 block 对象有时候被成为“闭包(closure)”。在这里，它们通常被口语化为”块(blocks)”，除非在某些范围它们容易和标准 C 表达式的块代码混淆。

2.1　Block功能

一个 block 就是一个匿名的内联代码集合体：

- 和函数一样拥有参数类型
- 有推断和声明的返回类型
- 可以捕获它的声明所在相同作用域的状态

  可以和其他定义在相同作用域范围的 blocks 进行共享更改
- 在相同作用域范围（栈帧）被销毁后持续共享和更改相同作用域范围(栈帧)的状态

你可以拷贝一个 block, 甚至可以把它作为可执行路径传递给其他线程（或者在自己的线程内传递给 run loop)。编译器和运行时会在整个 block 生命周期里面为所有 block 引用变量保留一个副本。尽管 blocks 在纯 C 和 C++上面可用，但是一个 block 也同样是一个 Objective-C 的对象。

2.2　用处

Blocks 通常代表一个很小、自包的代码片段。因此它们作为封装的工作单元在并发执行，或在一个集合项上，或当其他操作完成时的回调的时候非常实用。

Blocks 作为传统回调函数的一个实用的替代办法，有以下两个原因:

1. 它们可以让你在调用的地方编写代码实现后面将要执行的操作。

   因此 Blocks 通常作为框架方法的参数。
2. 它们允许你访问局部变量。

   而不是需要使用一个你想要执行操作时集成所有上下文的信息的数据结构来进行回调，你可以直接简单的访问局部变量。

第三章　声明和创建Blocks

3.1　声明一个block的引用

Block 变量拥有 blocks 的引用。你可以使用和声明函数指针类似的语法来声明它们，除了它们使用 ^ 修饰符来替代 * 修饰符。Block 类型可以完全操作其他 C 系统类型。以下都是合法的 block 声明:

```
void (^blockReturningVoidWithVoidArgument)(void);

int (^blockReturningIntWithIntAndCharArguments)(int, char);

void (^arrayOfTenBlocksReturningVoidWithIntArgument[10])(int);
```

Blocks 还支持可变参数（...）。一个没有使用任何参数的 block 必须在参数列表上面用 void 标明。

Blocks 被设计为类型安全的，它通过给编译器完整的元数据来合法使用 blocks、传递到 blocks 的参数和分配的返回值。你可以把一个 block 引用强制转换为任意类型的指针，反之亦然。但是你不能通过修饰符 * 来解引用一个 block, 因此一个 block 的大小是无法在编译的时候计算的。

你同样可以创建 blocks 的类型。当你在多个地方使用同一个给定的签名的 block 时，这通常被认为是最佳的办法。

```
typedef float (^MyBlockType)(float, float);


MyBlockType myFirstBlock = // ... ;

MyBlockType mySecondBlock = // ... ;
```

3.2　创建一个block

你可以使用 ^ 修饰符来标识一个 block 表达式的开始。它通常后面跟着一个被 () 包含起来的参数列表。Block 的主体一般被包含在 {} 里面。下面的示例定义了一个简单的 block，并把它赋值给前面声明的变量(oneFrom)。这里 block 使用一个标准 C 的结束符 ; 来结束。

```
int (^oneFrom)(int);
```

```
oneFrom = ^(int anInt) {
    return anInt - 1;
};
```

如果你没有显式的给 block 表达式声明一个返回值，它会自动的从 block 内容推断出来。如果返回值是推断的，而且参数列表也是 void，那么你同样可以省略参数列表的 void。如果或者当出现多个返回状态的时候，它们必须是完全匹配的（如果有必要可以使用强制转换）。

3.3 全局blocks

在文件级别，你可以把 block 作为全局标示符：

```
#import <stdio.h>


int GlobalInt = 0;
int (^getGlobalInt)(void) = ^{ return GlobalInt; };
```

第四章　Blocks和变量

本文描述 blocks 和变量之间的交互，包括内存管理。

4.1　变量类型

在 block 的主体代码里面，变量可以被使用五种方法来处理。

你可以引用三种标准类型的变量，就像你在函数里面引用那样:

- 全局变量，包括静态局部变量。
- 全局函数（在技术上而言这不是变量）。
- 封闭范围内的局部变量和参数。

Blocks 同样支持其他两种类型的变量:

1. 在函数级别是__block 变量。这些在 block 里面是可变的(和封闭范围)，并任何引用 block 的都被保存一份副本到堆里面。
2. 引入 const。
3. 最后，在实现方法里面，blocks 也许会引用 Objective-C 的实例变量。参阅“对象和 Block 变量”部分。

在 block 里面使用变量遵循以下规则:

1. 全局变量可访问，包括在相同作用域范围内的静态变量。
2. 传递给 block 的参数可访问（和函数的参数一样）。
3. 程序里面属于同一作用域范围的堆（非静态的）变量作为 const 变量(即只读)。它们的值在程序里面的 block 表达式内使用。在嵌套 block 里面，该值在最近的封闭范围内被捕获。
4. 属于同一作用域范围内并被__block 存储修饰符标识的变量作为引用传递因此是可变的。
5. 属于同一作用域范围内 block 的变量，就和函数的局部变量操作一样。

每次调用 block 都提供了变量的一个拷贝。这些变量可以作为 const 来使用，或在 block 封闭范围内作为引用变量。

下面的例子演示了使用本地非静态变量：

```
int x = 123;

void (^printXAndY)(int) = ^(int y) {

    printf("%d %d\n", x, y);
};

printXAndY(456); // prints: 123 456
```

正如上面提到的，在 block 内试图给 x 赋一个新值会导致错误发生：

```
int x = 123;

void (^printXAndY)(int) = ^(int y) {

    x = x + y; // error
    printf("%d %d\n", x, y);
};
```

为了可以在 block 内修改一个变量，你需要使用__block 存储类型修饰符来标识该变量。参阅“__block 存储类型”部分。

4.2 __block存储类型

你可以指定引入一个变量为可更改的，即读-写的，通过应用__block 存储类型修饰符。局部变量的__block 的存储和 register、auto、static 等存储类型相似，但它们之间不兼容。

__block 变量保存在变量共享的作用域范围内，所有的 blocks 和 block 副本都声明或创建在和变量的作用域相同范围内。所以，如果任何 blocks 副本声明在栈内并未超出栈的结束时，该存储会让栈帧免于被破坏（比如封装为以后执行）。同一作用域范围内给定的多个 block 可以同时使用一个共享变量。

作为一种优化，block 存储在栈上面，就像 blocks 本身一样。如果使用 Block_copy 拷贝了 block 的一个副本（或者在 Objective-C 里面给 block 发送了一条 copy 消息），变量会被拷贝到堆上面。所以一个__block 变量的地址可以随时间推移而被更改。

使用__block 的变量有两个限制：它们不能是可变长的数组，并且它们不能是包含有 C99 可变长度的数组变量的数据结构。

以下举例说明了如何使用__block 变量：

```
  block int x = 123; //  x lives in block storage


void (^printXAndY)(int) = ^(int y) {


   x = x + y;
   printf("%d %d\n", x, y);
};
printXAndY(456); // prints: 579 456
// x is now 579
```

下面的例子显示了 blocks 和其他几个类型变量间的交互：

```
extern NSInteger CounterGlobal;
static NSInteger CounterStatic;


{
   NSInteger localCounter = 42;
     block char localCharacter;


   void (^aBlock)(void) = ^(void) {
       ++CounterGlobal;
       ++CounterStatic;
       CounterGlobal = localCounter; // localCounter fixed at block
creation
       localCharacter = 'a'; // sets localCharacter in enclosing scope
   };


   ++localCounter; // unseen by the block
   localCharacter = 'b';


   aBlock(); // execute the block
```

```
    // localCharacter now 'a'
}
```

4.3 对象(Object)和Block变量

Block 提供了支持 Objective-C 和 Objective-C++的对象，和其他 blocks 的变量。

4.3.1 Objective-C对象

在引用计数的环境里面,默认情况下当你在 block 里面引用一个 Objective-C 对象的时候，该对象会被 retain。当你简单的引用了一个对象的实例变量时，它同样被 retain。但是被__block 存储类型修饰符标记的对象变量不会被 retain.

注意:在垃圾回收机制里面，如果你同时使用__weak 和__block 来标识一个变量，那么该 block 将不会保证它是一直是有效的。

如果你在实现方法的时候使用了 block, 对象的内存管理规则更微妙:

- 如果你通过引用来访问一个实例变量，self 会被 retain。
- 如果你通过值来访问一个实例变量，那么变量会被 retain。

下面举例说明两个方式的不同:

```
dispatch_async(queue, ^{
    // instanceVariable is used by reference, self is retained
    doSomethingWithObject(instanceVariable);
});



id localVariable = instanceVariable;
dispatch_async(queue, ^{
    // localVariable is used by value, localVariable is retained (not self)
    doSomethingWithObject(localVariable);
});
```

4.3.2 C++对象

通常你可以在 block 内使用 C++的对象。在成员函数里面，通过隐式的导入 this 指针引用成员变量和函数，结果会很微妙。有两个条件可以让 block 被拷贝:

- 如果你拥有__block 存储的类，它本来是一个基于栈的 C++对象，那么通常会使用 copy 的构造函数。
- 如果你在 block 里面使用任何其他 C++基于栈的对象，它必须包含一个 const copy 的构造函数。该 C++对象使用该构造函数来拷贝。

4.3.3 Blocks

当你拷贝一个 block 时，任何在该 block 里面对其他 blocks 的引用都会在需要的时候被拷贝，即拷贝整个目录树(从顶部开始)。如果你有 block 变量并在该 block 里面引用其他的 block，那么那个其他的 block 会被拷贝一份。

当你拷贝一个基于栈的 block 时，你会获得一个新的 block。但是如果你拷贝一个基于堆的 block，你只是简单的递增了该 block 的引用数，并把原始的 block 作为函数或方法的返回值。

第五章　使用Blocks

5.1　调用一个Block

如果你声明了一个 block 作为变量，你可以把它作为一个函数来使用，如下面的两个例子所示:

```
int (^oneFrom)(int) = ^(int anInt) {
    return anInt - 1;
};

printf("1 from 10 is %d", oneFrom(10));
// Prints "1 from 10 is 9"

float (^distanceTraveled) (float, float, float) =
                        ^(float startingSpeed, float acceleration, float time) {
    float distance = (startingSpeed * time) + (0.5 * acceleration * time * time);
    return distance;
};

float howFar = distanceTraveled(0.0, 9.8, 1.0);
// howFar = 4.9
```

然而你通常会把 block 作为参数传递给一个函数或方法。在这种情况下，你通过需要创建一个”内联（inline）”的 block。

5.2　使用Block作为函数的参数

你可以把一个 block 作为函数的参数就像其他任何参数那样。然而在很多情况下，你不需要声明 blocks;相反你只要简单在需要它们作为参数的地方内联实现它们。下面的例子使用 qsort_b 函数。qsort_b 和标准 qsort_r 函数类似，但是它最后一个参数用 block.

```
char *myCharacters[3] = { "TomJohn", "George", "Charles Condomine" };


qsort_b(myCharacters, 3, sizeof(char *), ^(const void *l, const void *r) {
    char *left = *(char **)l;
    char *right = *(char **)r;
    return strncmp(left, right, 1);
});
// Block implementation ends at "}"


// myCharacters is now { "Charles Condomine", "George", "TomJohn" }
```

注意函数参数列表包含了一个 block。

下一个例子显示了如何在 dispatch_apply 函数里面使用 block。dispatch_apply 声明如下：

```
void dispatch_apply(size_t iterations, dispatch_queue_t queue, void (^block)(size_t));
```

该函数提交一个 block 给批处理队列来多次调用。它需要三个参数；第一个指定迭代器的数量；第二个指定一个要提交 block 的队列；第三个是 block 它本身，它自己需要一个参数（当前迭代器的下标）。

你可以使用 dispatch_apply 来简单的打印出迭代器的下标，如下：

```
#include <dispatch/dispatch.h>
size_t count = 10;
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);


dispatch_apply(count, queue, ^(size_t i) {
    printf("%u\n", i);
});
```

5.3 使用Block作为方法的参数

Cocoa 提供了一系列使用 block 的方法。你可以把一个 block 作为方法的参数就像其他参数那样。

下面的例子确定数组前面五个元素中第一个出现在给定的过滤器集中任何一个的下标。

```
NSArray *array = [NSArray arrayWithObjects: @"A", @"B", @"C", @"A", @"B", @"Z",@"G", @"are",
@"Q", nil];

NSSet *filterSet = [NSSet setWithObjects: @"A", @"Z", @"Q", nil];



BOOL (^test)(id obj, NSUInteger idx, BOOL *stop);



test = ^ (id obj, NSUInteger idx, BOOL *stop) {



    if (idx < 5) {

        if ([filterSet containsObject: obj]) {

            return YES;

        }

    }

    return NO;

};



NSIndexSet *indexes = [array indexesOfObjectsPassingTest:test];



NSLog(@"indexes: %@", indexes);



/*

Output:

indexes: <NSIndexSet: 0x10236f0>[number of indexes: 2 (in 2 ranges), indexes: (0 3)]

*/
```

下面的例子确定一个 NSSet 是否包含一个由局部变量指定的单词，并且如果条件成立把另外一个局部变量(found)设置为 YES（并停止搜索）。注意到 found 同时被声明为__block 变量，并且该 block 是内联定义的：

```
  block BOOL found = NO;

NSSet *aSet = [NSSet setWithObjects: @"Alpha", @"Beta", @"Gamma", @"X", nil];

NSString *string = @"gamma";



[aSet enumerateObjectsUsingBlock:^(id obj, BOOL *stop) {
```

```
    if ([obj localizedCaseInsensitiveCompare:string] == NSOrderedSame) {
        *stop = YES;
        found = YES;
    }
}];

// At this point, found == YES
```

5.4 拷贝Blocks

通常，你不需要 copy（或 retain）一个 block. 在你希望 block 在它被声明的作用域被销毁后继续使用的话，你子需要做一份拷贝。拷贝会把 block 移到堆里面。

你可以使用 C 函数来 copy 和 release 一个 block：

```
Block copy();
Block release();
```

如果你使用 Objective-C，你可以给一个 block 发送 copy、retain 和 release（或 autorelease）消息。

为了避免内存泄露，你必须总是平衡 Block_copy()和 Block_release()。你必须平衡 copy 或 retain 和 release（或 autorelease）--除非是在垃圾回收的环境里面。

5.5 需要避免的模式

一个 block 的文本（通常是^{...}）是一个代表 block 的本地栈数据结构地址。因此该本地栈数据结构的作用范围是封闭的复合状态，所以你应该避免下面例子显示的模式：

```
void dontDoThis() {
    void (^blockArray[3])(void);  // an array of 3 block references

    for (int i = 0; i < 3; ++i) {
        blockArray[i] = ^{ printf("hello, %d\n", i); };
        // WRONG: The block literal scope is the "for" loop
    }
}
```

```
void dontDoThisEither() {
    void (^block)(void);

    int i = random():
    if (i > 1000) {
        block = ^{ printf("got i at: %d\n", i); };
        // WRONG: The block literal scope is the "then" clause
    }
    // ...
}
```

5.6 调试

你可以在 blocks 里面设置断点并单步调试。你可以在一个 GDB 的对话里面使用 invoke-block 来调用一个 block。如下面的例子所示:

```
$ invoke-block myBlock 10 20
```

如果你想传递一个 C 字符串，你必须用引号括这它。例如，为了传递 this string 给 doSomethingWithString 的 block，你可以类似下面这样写:

```
$ invoke-block doSomethingWithString "\"this string\""
```

结束语

Block 是 iOS 4.0 之后添加的新特性支持。本人亲测感觉使用 Block 最大的便利就是简化的回调过程。以前使用 UIView 的动画，进程要控制动画结束后进行相应的处理。iOS 4.0 之后，UIView 新增了对 Block 的支持，现在只要使用简单的一个 Block 代码就可以在写动画的代码部分直接添加动画结束后的操作。还有就是在使用 Notification 时候 Block 也非常有帮助。反正多用就可以体会到 Block 的优美了。

对了，使用 Block 要谨记别造成对象互相引用对方导致引用计数进入一个循环导致对象无法被释放。iOS 5.0 之后的 ARC 也是无法解决该潜在的互相引用的问题的。所以写 Block 的时候要注意这点。因为 Block 往往在后台自动对一些它引用了的对象进行 retain 操作。具体形式这里就不距离了，大家在使用的时候多体会一下。

最后，本文在翻译过程中发现很多地方直译成中文比较晦涩，所以采用了意译的方式，这不可避免的造成有一些地方可能和原文有一定的出入，所以如果你阅读的时候发现有任何的错误都可以给我发邮件:xyl.layne@gmail.com。

最后可以关注我微博大家一起沟通交流学习。

微博地址: http://weibo.com/u/1826448972

推荐资源

- 核心动画编程指南【**Core Animation Programming Guide**】

下载地址:
http://www.cocoachina.com/bbs/read.php?tid=84461

- 多线程编程指南【**Threading Programming Guide**】

下载地址:
http://www.cocoachina.com/bbs/read.php?tid=87592

- **Instruments** 用户指南【**Instruments User Guide**】

下载地址:【近期推出，敬请关注微博动态】