



# 深入Object C

---



# 一、类别(Category)与类扩展

## 1、类别(Category)

●当封装了一个类后却又需要修改或新增新的方法时，可以通过为该类添加一个类别（category）来实现。

●通过类别的方式，可以将类的实现分散到不同的文件里

●类别是一种为现有的类添加新方法的方式。

---





## HelloWorld+newgreeting.h

```
#import "helloworld.h"  
@interface helloworld (newgreeting)  
- (void) sayNewgreeting;  
@end
```

## HelloWorld+newgreeting.m

```
#import "helloworld+newgreeting.h"  
@implementation helloworld (newgreeting)  
- (void) sayNewgreeting  
{  
    NSLog(@"Hello Category");  
}  
@end
```

---



类别声明:

```
@interface ClassName(CategoryName)
    //method declarations
@end
```

类别实现

```
@implementation ClassName(CategoryName)
    // the implementation of new methods
@end
```

---





```
#import "helloworld.h"
#import "helloworld+newgreeting.h"
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        helloworld *h=[[helloworld alloc] init];
        [h sayNewgreeting];
        h.something=@"IOS world";
        [h sayHello:@"IOS"];
        [h saysomething];
        [helloworld sayHelloWorld];
    }
    return 0;
}
```

```
Hello Category
hello IOS
hello IOS world😄
Hellow world
```



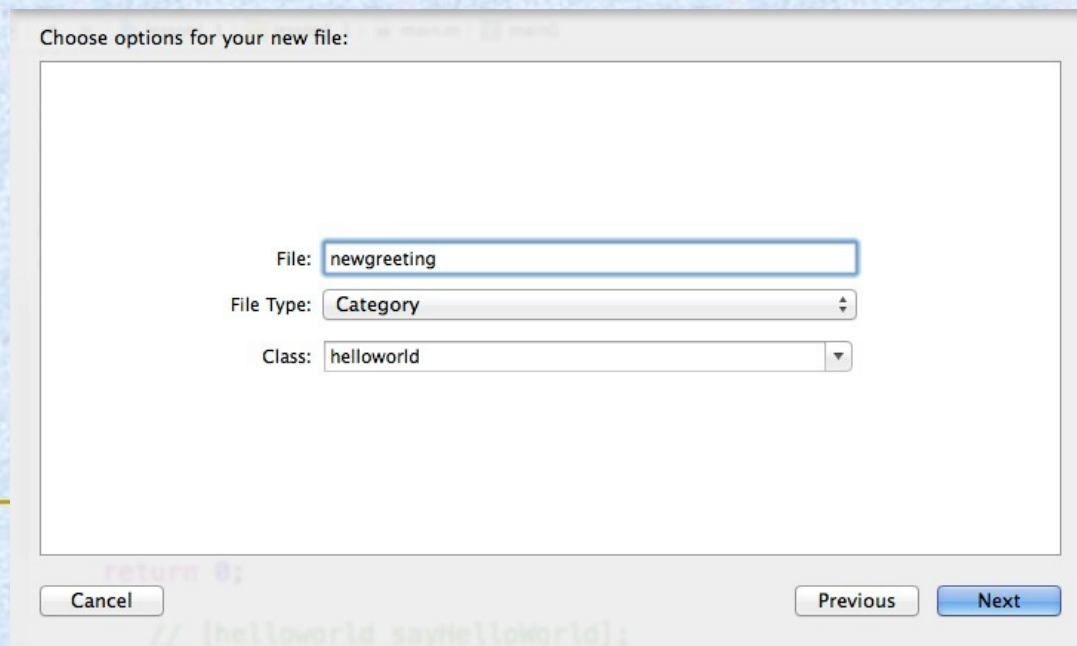
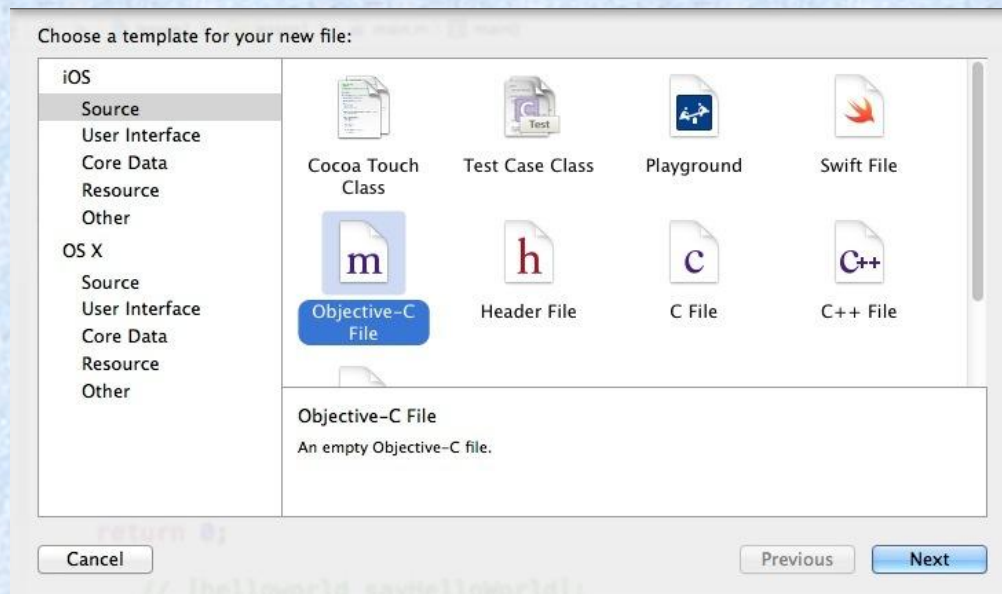
注意：

- 类别只能添加新方法，并不能添加新的数据成员
  - 当类别中新增方法和原来类中的方法产生名称冲突，则覆盖原来的方法。
  - 类别代表了为一个类声明了一些方法而不是一个新类。
-





## 利用Xcode 模板生成类别





## 2、类扩展

类扩展声明可以在类扩展中声明属性和数据类型。类扩展允许在主类的@interface以外的地方为类声明一些必要的方法。

语法：

```
@interface ClassName()
```

```
{ //私有数据成员 }
```

```
//私有方法与属性
```

```
@end
```

已经在.h中声明过







helloworld.h

```
#import <Foundation/Foundation.h>
@interface helloworld : NSObject
@property (strong, readonly, nonatomic) NSString *something;
+ (void) sayHelloWorld;
- (void) sayHello:(NSString *) greeting ;
- (void) saySomething;
@end
```

私有

helloworld.m

```
#import "helloworld.h"
@interface helloworld ()
{
    NSString *privateVariable;
}

@property (strong, nonatomic) NSString *privateProperty;
@property (strong, readwrite, nonatomic) NSString *something;
- (void) sayPrivateGreeting;

@end

@implementation helloworld
- (void) sayPrivateGreeting
{
    privateVariable=@"Hello world";
    self.privateProperty=privateVariable;
    NSLog(@"%@",privateVariable);
}
}
```

重新声明为可读写



注意：

- 类扩展声明一些必要的私有方法
  - 对在.h的interface中声明为只读属性，可以在扩展中重新声明为读写。
-





## 二、协议与委托

### 1、协议(Protocol)

Objective-C中的协议是通过一个简单的方法声明列表发布一个方法列表，任何类都可以选择实现。协议中的方法通过其它类实例发送的消息来进行调用。

#### 协议声明

@protocol ProtocolName

基协议  
协议嵌套

//method declarations

@end

必须实现的方法  
默认情况下是Required

```
@protocol NewGreeting<NSObject>
@required
-(void)sayNewgreetingByProtocol;
@optional
-(void)sayNewgreetingByProtocolOptional;
@end
```

可选实现的方法

NewGreeting.h



## 协议实现

协议必须在类中实现，一个类要采用某个协议则需要用@interface后接着父类的位置后用< >指定协议名称。一个类可以采纳多个协议，不同的协议之间用逗号分隔。

```
#import "NewGreeting.h"
@interface helloworld : NSObject<NewGreeting>
@property (strong, readonly, nonatomic) NSString *something;
+ (void) sayHelloWorld;
- (void) sayHello:(NSString *) greeting ;
- (void) saySomething;
@end
```



```
@interface helloworld ()
{
    NSString *privateVariable;
}
@property (strong, nonatomic) NSString *privateProperty;
@property (strong, readwrite, nonatomic) NSString *something;
-(void) sayPrivateGreeting;
@end

@implementation helloworld
//必须实现的协议方法
-(void) sayNewgreetingByProtocol
{
    NSLog(@"Hello Protocol");
}

//可选实现的方法
-(void) sayNewgreetingByProtocolOptional
{
    NSLog(@"Hello optional method in Protocol ");
}

//下面实现其它方法
- (void) sayPrivateGreeting
{
    privateVariable=@"Hello world";
    self.privateProperty=privateVariable;
    NSLog(@"%@",privateVariable);
}
}
```



定义变量或属性时可限制变量保存的对象遵守某个协议，编译器会对静态类型变量是否遵守协议进行检查。

例如：

```
id<NewGreeting> obj1;
```

```
NSObject <NewGreeting> *obj2;
```

```
@property (nonatomic) id<NewGreeting> *obj3;
```


---





也可以在代码中使用协议对象来判断某个对象是否实现了给定的协议。

```
Protocol *p=@protocol(NewGreeting);  
if([h conformsToProtocol:p])  
    [h sayNewgreetingByProtocol];
```



判断是否遵循了给定的协议

---



## 避免协议嵌套

```
#import "B.h"  
@protocol A  
    -foo: (id<B>) obj1  
@end
```

```
#import "A.h"  
@protocol B  
    -bar: (id<A>) obj2  
@end
```

## 避免嵌套

```
@protocol B;  
@protocol A:  
    -foo: (id<B>) obj1  
@end
```

通知编译器B是一个稍后定义的协议。

---





## 非正式协议

上面的内容都是正式协议，我们也可以定义非正式协议。非正式协议可以看成是一个特殊的类别。可以把方法分到一个类别中定义一个非正式协议。非正式协议通常定义为NSObject的类别。

```
@interface NSObject(NewGreeting)
    -(void) sayNewgreetingByProtocol;
@end
```

---



协议的优点:

- 协议是一个类在声明接口的同时隐藏自身的一种方式。
  - 可以弥补Object C无法多重继承的不足。
  - 其它的类可以通过协议和某个类进行特定的(协议规定的方法)交互。
  - 抽出那些没有继承关系的类之间的相似之处。
-





## 2、委托(Delegate)

委托是一种模式，一个对象可代表另外一个对象执行某个动作，或者与之相互协作共同完成某个任务。发布委托的对象持有其他对象（委托）的引用。在适当的时候，它会向委托发送消息。消息用于通知委托对象将要处理或者已经处理某个事件。作为响应，委托对象会更新外观或者更新自身或应用程序其他对象的状态。

参见视频中文本框委托的例子。

---





### 三、内省机制 ( Introspection )

内省是对象揭示自己作为一个运行时对象的详细信息的一种能力。这些详细信息包括对象在继承树上的位置，对象是否遵循特定的协议，以及是否可以响应特定的消息。

NSObject协议和类定义了很多内省方法，用于查询运行时信息，以便根据对象的特征进行识别。

明智地使用内省可以使面向对象的程序更加高效 和强 壮。它有助于避免错误地进行消息派发、错误地假设对象相等、以及类似的问题。

---





## SEL类型

Objective-C在编译的时将根据方法名字（包括参数序列）生成一个用来区分该方法的唯一ID。这个ID就是SEL类型的。

**注意：**只要方法的名字（包括参数序列）相同，那么它们的ID都是相同的。

### SEL类型赋值：

```
SEL    变量名  =  @selector(方法名字);
```

```
SEL    变量名  =  NSStringFromSelector(方法名字的字符串);
```

```
NSString *变量名  =  NSStringFromSelector(SEL参数);
```

---



所以我们可以代码中利用SEL类型来派发消息。

```
- (id)performSelector:(SEL) aSelector  
- (id)performSelector:(SEL) aSelector withObject:(id) anObject  
- (void)performSelectorOnMainThread:(SEL) aSelector  
    withObject:(id) arg  
    waitUntilDone:(BOOL) wait  
- (void)performSelector:(SEL) aSelector  
    onThread:(NSThread *) thread  
    withObject:(id) arg  
    waitUntilDone:(BOOL) wait  
    modes:(NSArray *) array
```

输出相同

```
helloworld *h=[[helloworld alloc] init];  
h.something=@"new world";  
SEL s1= @selector(saySomething);  
SEL s2=@selector(sayHello:);  
[h performSelector:s1];  
[h performSelector:s1 withObject:nil];  
[h performSelector:s2 withObject:@"SEL TYPE"];
```

:不可忽略





## 常用内省方法

### 1、**class**和 **superclass**

分别以class对象返回接收者的类和父类

### 2、**isKindOfClass:Class**

检查对象是否是那个类或其继承类实例化的对象

### 3、**isMemberOfClass:Class**

检查对象是否是那个类但不包括继承类而实例化的对象

```
if ([h isKindOfClass:[helloworld class]])  
{  
    [h sayHello:@"Hello world"];  
}
```



#### 4、respondsToSelector: selector

用来判断是否有以某个名字命名的方法(被封装在一个selector的对象里传递)

```
SEL s1= @selector(saySomething);  
if ([h respondsToSelector:s1]) {  
    [h performSelector:s1];  
}
```

#### 5、conformsToProtocol:protocol

检查对象是否符合协议，是否实现了协议中所有的必选方法

```
Protocol *p=@protocol(NewGreeting);  
if([h conformsToProtocol:p])  
    [h sayNewgreetingByProtocol];
```





### 三、内存管理

Object C没有像JAVA一样的垃圾回收机制，所以需要程序员手动去管理内存。考虑c和c++中的内存管理)

#### Object C中的两类数据的存储

- 基本数据类型存放在栈中，由系统管理。
- 对象存放在堆中，由程序员自己管理。



## 1、对象内存空间的开辟和释放

```
helloworld *h = [[helloworld alloc] init];
```

在堆上开辟内存

```
h=nil;
```

内存泄露





## 1、对象内存空间的开辟和释放

```
helloworld *h = [[helloworld alloc] init];
```

在堆上开辟内存

```
[h dealloc];
```

在堆上释放内存

在实际的使用中，并不需要直接调用dealloc方法，而是使用了Cocoa的引用计数机制来管理和释放内存

---



## 2、引用计数机制

Cocoa在内存管理上采用了引用计数（retain count）机制来管理内容，在对象内部保存一个数字，用来表示被引用的次数。init、new和copy都会让retain count加1。当销毁对象的时候，系统不会直接调用dealloc方法，而是先调用release，让retain count减1，当retain count等于0的时候，系统才会调用dealloc方法来销毁对象。

常用关键字：

copy：是拷贝一份对象的副本，引用计数为1。

retain：使一个对象的引用计数加1。

release：使得一个对象的引用计数减少1。

autorelease：引用计数在未来某个阶段减少1。

---





## 引入引用计数后的内存开辟和释放

retain count = 1

helloworld \*h1 = [[helloworld alloc] init];

helloworld \*h2=h1;

[h2 retain]; → retain count = 2

[h1 release]; → retain count = 1

h1=nil;

[h2 release]; → retain count = 0  
回收内存

[h2 release]; → 程序出错



### 3、Autorelease Pool

为了方便管理内存，Object C中引入了自动释放池 (Autorelease Pool)。在遵守一些规则的情况下，可以自动释放对象。

```
helloworld *h1 = [[helloworld alloc] init] autorelease];
```

此时retain count = 1，但无需release

---





自动释放池其实是Cocoa的一个类NSAutoreleasePool所以也需要创建和初始化。当我们设置一个对象指针为Autorelease时，其将被加入到其中。

```
NSAutoreleasePool *pool =[[NSAutoreleasePool alloc] init];
```

注意：

- 自动释放池也要release. Eg: [pool release];
  - 当NSAutoreleasePool被销毁的时候，它会遍历数组，release数组中的每一个成员（注意，这里只是release，并没有直接销毁对象）。若成员的retain count 大于1，那么对象没有被销毁，造成内存泄露。
-



#### 4、自动引用计数 ARC(Automatic Reference Counting)

ARC在IOS 5推出的新功能， IOS7.0以后，强制使用ARC。ARC可以看成是代码中自动加入了retain/release，内存管理的引用计数可以自动地由编译器完成。

两个概念：

强引用: strong 或 \_\_strong

弱引用: weak 或 \_\_weak

修饰属性

修饰变量

对变量来说 \_\_strong 是缺省的关键





## 强引用

可以增加引用计数的值。在不需要的时候，需要手动设置为nil。引用计数来记录有多少使用者在使用一个对象，如果所有使用者都放弃了对该对象的引用，则该对象将被自动销毁。

```
helloworld *h=[[helloworld alloc] init];  
NSString *str;  
h.something=@"Hello world";  
str=h.something;  
h.something=nil;  
NSLog(@"%@",str);
```

输出:hello world

---



```
helloworld *h=[[helloworld alloc] init];  
NSString *str;  
h.something=@"Hello world";  
h=nil;  
str=h.something;  
NSLog(@"%@",str);
```

输出: null

```
helloworld *h=[[helloworld alloc] init];  
helloworld *hStrong;  
h.something=@"Hello world";  
hStrong=h;  
h=nil;  
NSLog(@"%@",hStrong.something);
```

输出:hello world

---





## 弱引用

它也被称为“归零弱引用”。可以只是持有指针而不增加引用计数。当指针指向的内存被销毁后，声明weak的属性指针会自动置为nil，这也是它被称为归零弱引用的原因。

```
NSString *strWeak;  
NSMutableString *strStrong=[[NSMutableString alloc] initWithString:@"Hello world"];  
strWeak=strStrong;  
strStrong=nil;  
NSLog(@"%@", strWeak);
```

输出:hello world

---



```
__weak NSString *strWeak;  
NSString *strStrong=@"Hello world";  
strWeak=strStrong;  
strStrong=nil;  
NSLog(@"%@", strWeak);
```

输出:hello world.

```
__weak NSString *strWeak;  
NSMutableString *strStrong=[[NSMutableString alloc] initWithString:@"Hello world"];  
strWeak=strStrong;  
strStrong=nil;  
NSLog(@"%@", strWeak);
```

输出:null

---





```
helloworld *hStrong=[[helloworld alloc] init];  
helloworld *hWeak;  
hStrong.something=[[NSMutableString alloc] initWithFormat:@"Hello %@",@"world"];  
hWeak=hStrong;  
hStrong=nil;  
NSLog(@"%@",hWeak.something);
```

输出: Hello world

```
helloworld *hStrong=[[helloworld alloc] init];  
__weak helloworld *hWeak;  
hStrong.something=[[NSMutableString alloc] initWithFormat:@"Hello %@",@"world"];  
hWeak=hStrong;  
hStrong=nil;  
NSLog(@"%@",hWeak.something);
```

输出: null

---



## 使用ARC的一些规则：

1. 不允许调用对象的retain, release, autorelease等方法，也不允许通过类似@selector(retain)的方法进行间接调用。
  2. 不允许调用对象的dealloc方法。。
  3. 在@property声明中，不再允许使用retain, copy, assign等关键字，取而代之的是两个新关键字：strong和weak。
  4. 当结构中包含对象指针时，最好不要使用struct，改为class，因为ARC有可能无法辨识struct中的对象指针。
  5. 不允许使用NSAutoreleasePool对象，取而代之的是@autoreleasepool {} 块。
-





## 四、块 ( BLOCK )

问题？

- 什么是块(BLOCK)?
- 为什么要用块(BLOCK)?



# 1、什么是块(BLOCK)?

Blocks are **objects** that **encapsulate** a unit of work—or, in less abstract terms, a segment of code—that can be executed at any time. They are essentially portable and anonymous functions that one can pass in as arguments of methods and functions or that can be returned from methods and functions. Blocks themselves have a typed argument list and may have inferred or declared returned type. You may also assign a block to a **variable** and then call it just as you would a function.

---





## 2、为什么要用块(BLOCK)?

● An advantage of blocks as function and method parameters is that they enable the caller to provide the **callback** code at the point of invocation. Because this code does not have to be implemented in a separate method or function, your implementation code can be simpler and easier to understand.

考虑回调函数，函数指针

可以提供回调功能



● An even more valuable advantage of blocks over other forms of callback is that a block **shares data** in the local lexical scope. If you implement a method and in that method define a block, the block has access to the local variables and parameters of the method (including stack variables) as well as to functions and global variables, including instance variables. This access is **read-only** by default, but if you declare a variable with the **\_\_block modifier**, you can change its value within the block. Even after the method or function enclosing a block has returned and its local scope is destroyed, the local variables persist as part of the block object as long as there is a reference to the block.

可以扩展对象的作用区域





# 1、块(BLOCK)的定义

比较块和函数指针:

int (\* myfunction) (int a);

函数指针

返回类型

变量名

参数类型与参数名

int (^ myblock)

(int a);

块的声明

定义了一个block变量



## 块的实现部分的语法

### ● 方式1: `^{.....}`

说明: 不带参数的块

eg: `^{NSLog(@"Hello world",);}`

### ● 方式2: `^(传入参数列){.....}`

说明: 带传入参数的块

eg: `^(NSString *str){ NSLog(@"%@@", str); }`

### ● 方式3: `^类型(传入参数列){ ..... return ..... }`

说明: 带传入参数且有返回值的块.

eg: `^BOOL(id obj)`

`{ return [obj isKindOfClass: [helloworld class]]; }`





● 方式4: ^ (传入参数列){ ..... return ..... }

说明: 和(2.3)用法一样, 但省略了类型

eg: ^(id obj)

{ return [obj isKindOfClass: [helloworld class]]; }

方式1:

```
void (^myblock) (void);  
myblock=^{NSLog(@"Hello world");};  
myblock();
```



方式2:

```
void (^myblock) (NSString *str);  
myblock=^(NSString *s){NSLog(@"Hello %@",s);};  
myblock(@"world");
```

方式3:

```
int (^myblock) (int a);  
myblock=^int(int num){ NSLog(@"%d",num); return num+1;};  
int result=myblock(1);  
NSLog(@"%d",result);
```

方式4:

```
int (^myblock) (int a);  
myblock=^(int num){ NSLog(@"%d",num); return num+1;};  
int result=myblock(1);  
NSLog(@"%d",result);
```

---





## BLOCK作为参数传递的例子。

- enumerateKeysAndObjectsUsingBlock:

Applies a given block object to the entries of the dictionary.

### Declaration

#### OBJECTIVE-C

```
- (void)enumerateKeysAndObjectsUsingBlock:(void (^)(id key,  
                                                    id obj,  
                                                    BOOL *stop)) block
```

### Parameters

<i>block</i>	A block object to operate on entries in the dictionary.
--------------	---

### Discussion

If the block sets *\*stop* to YES, the enumeration stops.

```
NSMutableDictionary *dict=[NSMutableDictionary dictionaryWithObjectsAndKeys:  
    @"one",@"1",@"two",@"2",@"three",@"3",@"four",@"4",@"five",@"5", nil];  
[dict enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL *stop) {  
    if ([@"1" isEqualToString:key]) {  
        *stop=YES;  
    }  
}];
```



可把BLOCK当对象一样进行存储操作，例如在变量和数组。

```
NSMutableArray *myBlockArray=[[NSMutableArray alloc] init];;
void (^myblock1) (void);
void (^myblock2) (NSString *str);
void (^myblock3) (void);
void (^myblock4) (NSString *str);
myblock1=^{NSLog(@"Hello world");};
myblock2=^(NSString *s){NSLog(@"Hello %@",s);};
[myBlockArray addObject:myblock1];
[myBlockArray addObject:myblock2];
myblock3=myBlockArray[0];
myblock4=myBlockArray[1];
myblock3();
myblock4(@"world");
```





## BLOCK的作用域

- 在块内可以使用块外的变量，但只能读取不能赋值。
- 块实际上是复制一份变量，然后对复制品进行操作。

readonly

```
int b=0;
int (^myblock) (int a);
myblock=^int(int num){ NSLog(@"%d",num); return num+b;};
b++;
int result=myblock(1);
NSLog(@"%d",result);
```

输出: 1  
1



可使用\_\_block 扩展修改块外的变量

```
__block int b=0;  
int (^myblock) (int a);  
myblock=^int(int num){ NSLog(@"%d",num);b++; return num+b;};  
b++;  
int result=myblock(1);  
NSLog(@"%d",result);
```

输出: 1  
3

---





如果变量是对象实例会怎样？

```
helloworld *h=[[helloworld alloc] init];  
h.something=@"world";  
void (^myblock) (NSString *str);  
myblock=^(NSString *newgreeting){ h.something=newgreeting;};  
myblock(@"Block");  
[h saySomething];  
NSLog(@"%@",h.something);
```

输出: hello Block  
Block

---



由于块可以像对象一样被存储在变量中，ARC会进行计数，所以使用时候要注意存储循环问题。

```
helloworld *h=[[helloworld alloc] init];  
h.something=@"world";  
h.myBlocks=[[NSMutableArray alloc] init];  
[h.myBlocks addObject:^(h saySomething);];  
[h.myBlocks addObject:^(NSString *s){h sayHello:s};];
```

警告

⚠ Capturing 'h' strongly in this block is likely to lead to a retain cycle

内存泄露





解决方案：使用weak类型。

```
helloworld *h=[[helloworld alloc] init];  
__weak helloworld *weakh=h;  
h.something=@"world";  
h.myBlocks=[[NSMutableArray alloc] init];  
[h.myBlocks addObject:^(weakh saySomething);)];  
[h.myBlocks addObject:^(NSString *s){[weakh sayHello:s];}];
```

使用\_\_weak