Object-C runtime speciality

前言：

OC的动态运行时是OC语言不可忽视的一个非常重要的特性。理解运行时的特性可以加深你对Object-C语言本身的理解并能够了解你的应用是如何运行的。所以作为Mac或iPhone的开发者，无论你的水平阅历如何都会从对OC动态运行时特性的理解中获取一些有益的东西。

补充：

C语言编译分为以下四个流程：

我们以objc.c源文件为例，看看它是如何最终被编译成可执行文件的：

1. 预处理（Pre-Processing）

gcc -E objc.c -o objc.i

执行结果生成了objc.i文件，内容如下：

……

```
struct __sFILEX;
# 120 "/Applications/Xcode6.app/Contents/Developer/Platforms/
MacOSX.platform/Developer/SDKs/MacOSX10.9.sdk/usr/include/
stdio.h" 3 4
typedef struct __sFILE {
 unsigned char *_p;
 int _r;
 int _w;
 short _flags;
 short _file;
 struct __sbuf _bf;
 int _lbfsize;


 void *_cookie;
 int (*_close)(void *);
 int (*_read) (void *, char *, int);
 fpos_t (*_seek) (void *, fpos_t, int);
 int (*_write)(void *, const char *, int);

……

int main(int argc, const char * argv[]) {
```

```
        Class cls = objc_msgSend(objc_getClass("UIWindow"),
    "alloc");
        printf("Hello, World!\n");
        return 0;
}
```

……

我们发现，该过程其实是把 stdio.h文件中的内容插入到了objc.i文件中，并且将用到宏OBJC的地方都替换成它对应的“NSObject”，所以该阶段主要处理#ifdef、#include和#define等命令。

2. 编译（Compiling）

gcc -S objc.i -o objc.s

执行结果生成objc.s文件，内容如下：

……

```
_main:                                  ## @main
 .cfi_startproc
## BB#0:
 pushq  %rbp
Ltmp2:
 .cfi_def_cfa_offset 16
Ltmp3:
 .cfi_offset %rbp, -16
 movq   %rsp, %rbp
Ltmp4:
 .cfi_def_cfa_register %rbp
 subq   $32, %rsp
 leaq   L_.str(%rip), %rax
 movl   $0, -4(%rbp)
 movl   %edi, -8(%rbp)
 movq   %rsi, -16(%rbp)
 movq   %rax, %rdi
 callq  _objc_getClass
 leaq   L_.str1(%rip), %rsi
 movq   %rax, %rdi
 callq  _objc_msgSend
 leaq   L_.str2(%rip), %rdi
 movq   %rax, -24(%rbp)
 movb   $0, %al
 callq  _printf
```

……

我们发现该阶段生成了汇编代码，但是在进行转换之前编译器会对代码规范性及语法做检查然后给我警告或错误，如果没有检测到错误那么就会继续生成汇编代码文件。

3. 汇编（Assembling）

gcc -c objc.s -o objc.o

此时生成了可重定位目标文件（ELF文件结构）objc.o，该文件中包含了二进制代码和数据，可以在链接的时候与其他可重定位目标文件合并起来，创建一个可执行目标文件。

4. 链接（Linking）

gcc objc.o -o objc

生成了可执行目标文件（ELF文件结构）objc，此文件包含的也是二进制代码和数据，它可以被直接拷贝到内存中去运行，我们可以通过./objc命令执行该文件(程序)。

回过头我们再看objc.c中我们调用了“printf”函数，但是在预编译生成的objc.i文件中，我们只找到了该函数的声明并没有定义，那该函数在哪里实现的呢？在默认情况下，这些系统级的调用的实现会被放到libc.so.6的库文件中，链接要做的就是把该函数的实现链接到libc.so.6的库中，这样在执行文件的时候就能够找到实现并完成程序的运行。

tips：函数库一般分为静态库和动态库两种。静态库是指编译链接时，把库文件的代码全部加入到可执行文件中，因此生成的文件比较大，但在运行时也就不再需要库文件了。其后缀名一般为”.a”。动态库与之相反，在编译链接时并没有把库文件的代码加入到可执行文件中，而是在程序执行时由运行时链接文件加载库，这样可以节省系统的开销。动态库一般后缀名为”.so”，如前面所述的libc.so.6就是动态库。Gcc在编译时默认使用动态库。

最后运行生成的可执行文件objc

./objc

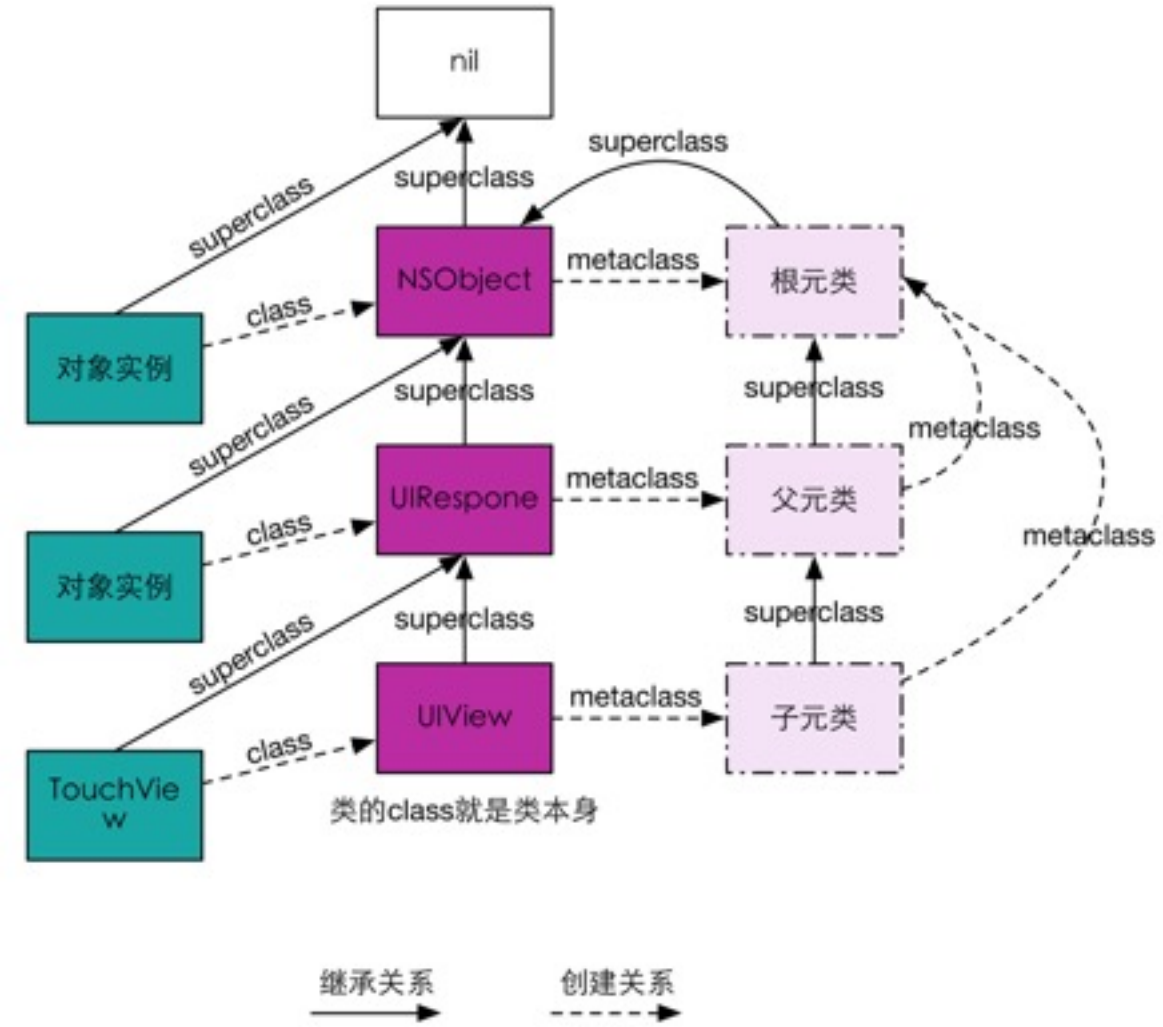
结果如下：

Hello, World!

一、对象、类及元类的关系：

在C++中类是抽象的，不占用内存，而对象是由类初始化而来的，占有内存空间。但是OC中所谓的类其本质上只是对结构体

typedef struct objc_class *Class;



的封装，OC中没有C++中真正意义上的类，所有的对象、类或元类(个别除外)在运行时候都是占有内存的，它们最终都继承自同一个父类——NSObject。

根据C++中的经验，对象实例是由类创建并初始化而来的，实例对象可以调用在类中声明的方法，而在OC中出现了类方法，即类本身可以调用自己的类方法。延伸一下思考的话，在OC中我们可以把类也当成对象来看待，那么这些类方法又声明在哪里呢？

struct objc_class {

```
    Class isa;
    ……
} OBJC2_UNAVAILABLE;
```

观察 struct objc_class 结构可以得知每一个类或对象中都有一个isa属性，对于对象而言isa指向（创建该对象）持有对象方法声明的类，对于类而言isa指向（创建该类）持有类方法声明的元类。

如：UIView的类方法

```
+(void)setAnimationsEnabled:(BOOL)enabled;
```

就是在UIView的元类中声明的。（见demo）

我们知道获取对象的isa指针对象可以调用

```
Class object_getClass(id obj)
{
    if (obj) return obj->isa;
    else return nil;
}
```

方法，那么上面我们说了将类当做对象来看待的话传入object_getClass()方法，返回的就是该类的元类；而object_getClass()接收的是一个id对象，我们可以通过objc_getClass()方法将类名传进去，返回的就是名称对应类。

其实还有一个方法objc_getMetaClass()，如下：

```
id objc_getMetaClass(const char *aClassName)
{
    Class cls;
    if (!aClassName) return Nil;
    cls = (Class)objc_getClass (aClassName);
    ...
    return (id)cls->isa;
}
```

观察objc_getMetaClass()的实现，以下两个函数返回的内容是相同的。

```
object_getClass(objc_getClass(“UIView”))
```

objc_getMetaClass(“UIView”)

类的元类我们获取到了，那么要获取元类的元类也是一样的道理。

上面我们提到了objc_getClass()函数，我们发现该方法仅仅通过一个类名称的字符串就可以获取到该名称对应的类结构，观察objc_getClass()函数的实现可以看出，苹果官方framework中提供的所有类都是以key-value的形式在内存中存在的。key代表类名称的字符串，对应的value则代表对应的类结构。

objc_getClass方法的跟踪。

二、消息：

怎么做才能够将对象的确定、消息的发送推迟到运行时去处理呢

objc_msgSend等方法就是解决该问题的核心

id objc_msgSend(id self, SEL op, ...)；

我们知道oc的编译器在编译的时候将oc格式的函数调用即：

[target sendMessage:var]；

转换为c语言函数的调用形式即：

objc_msgSend(target, @selector(sendMessage), var)；

如此一来只要objc_msgSend函数存在就能够通过编译链接，在编译的时候编译器不会去检查objc_msgSend函数的参数target和sendMessage到底是什么关系，而target和sendMessage的关系以及他们之间发生了什么是在去执行objc_msgSend函数的时候(即运行时)才能够真正的去确定的，这样就巧妙地做到了将对象的确定、消息的发送推迟到运行时去处理。所以所谓的动态特性，就是在静态C语言基础上，维护了一套能够动态的运行时系统，该系统能够对该系统中封装的类进行一些操作。

至于objc_msgSend函数的具体实现苹果是没有公开的。

三、Method SEL IMP：

1). Method 代表真正存在的代码总称。

如:- (int)meaning { return 42; }

2). Selecter 即 SEL 又名方法选择器，实际上它仅仅是一个指向 char 类型的指针(char *)，在 objc.h 中的定义为:

typedef struct objc_selector *SEL;

例如:

SEL selector = @selector(message);//@selector 不是函数调用,只是编译器能够明白的一个标示

NSLog (@"%s", (char *)selector); //结果输出 message 字符串

Objective-C 在编译的时候，会根据方法的名字，生成一个用来区分这个方法的唯一的一个 ID，这个 ID 就是 SEL 类型的。我们需要注意的是，只要方法的名字相同,那么它们的 ID 都是相同的。而这也就导致了 Objective-C 在处理有相同函数名和参数个数但参数类型不同的函数时会出现编译错误，比如：当你想在同一个类中实现下面两个方法:

-(void)setWidth:(int)width;

-(void)setWidth:(double)width;

而在 C++中对于函数名相同参数个数相同但是参数类型不同的函数是可以在同一类中定义的， 编译的时候编译器会将函数转换为类似 setWidth_int 和 setWidth_double 从而进行区分。

3). IMP

IMP 在 objc.h 中的定义为:

typedef id (*IMP)(id, SEL, ...);

熟悉 C 语言的同学应该清楚 IMP 实际上就以一个指向函数的指针。

上面我们提到过每一个类中都保存了指向父类的指针和一个方法列表（当然还有其他的属性），你可以将这个方法列表理解为一个有键值关系的数据模型(hash表或字典)，key 为 SEL指向的字符串，value 为 IMP。 在方法列表中 SEL 与 IMP 形成了一种一一对应的唯一映射关系。由于每个方法都对应唯一的SEL(同一个类中不能出现名称相同的函数即使它们有不同类型的参数)，因此我们可以通过 SEL 方便、快速、准确的获得它所对应的IMP(也就是函数指针)，而在取得了函数指针之后，也就意味着我们取得了执行的时候的这段方法的代码的入口，这样我们就可以像普通的 C 语言函数调用一样使用这个函数指针。当然我们可以把函数指针作为参数传递到其他的方法，或者实例变量里面，从而获得极大的动态性。很重要的一点是：selector 和 IMP 之间的

关系是在运行时才决定的，而不是编译时。 下面的例子，介绍了取得函数指针，即函数指针的用法:

void (* performMessage)(id,SEL);//定义一个 IMP(函数指针)

performMessage = (void (*)(id,SEL))[self methodForSelector:@selector(message)]; //通过 methodForSelector 方法根据 SEL 获取对应的函数指针

performMessage(self,@selector(message));//通过取到的 IMP(函数指针)跳过 runtime 消息分发机制，直接执行 message 方法

需要指出的是：用 IMP 的方式，省去了 runtime 消息分发过程中所做的一系列动作，比直接向对象发送消息高效一些。(见示例项目)

四、消息的分发：

1、消息分发流程

一切还是从消息表达式 [receiver message] 开始，在被转换成 objc_msgSend(receiver, SEL) 后，在运行时，runtime system 会做以下事情:

1) 检查忽略的 Selector,比如当我们运行在有垃圾回收机制的环境中，将会忽略 retain 和 release 消息。

2) 检查 receiver 是否为 nil。不像其他语言,nil 在 objective-C 中是完全合法的，并且这里有

很多原因你也愿意这样，比如，至少我们省去了给一个对象发送消息前检查对象是否为空的操作。如 果 receiver 为空，则会将 selector 也设置为空,并且直接返回到消息调用的地方。如果对象非空，就继续下一步。

3) 接下来会根据 SEL 到当前类中查找对应的 IMP，首先会在 cache 中检索它，如果找到了就根据函数指针跳转到这个函数执行，否则进行下一步。

4) 检索当前类对象中的方法表(dispathch table)，如果找到了，加入 cache 中，并且就跳转到这个函数之行,否则进行下一步。

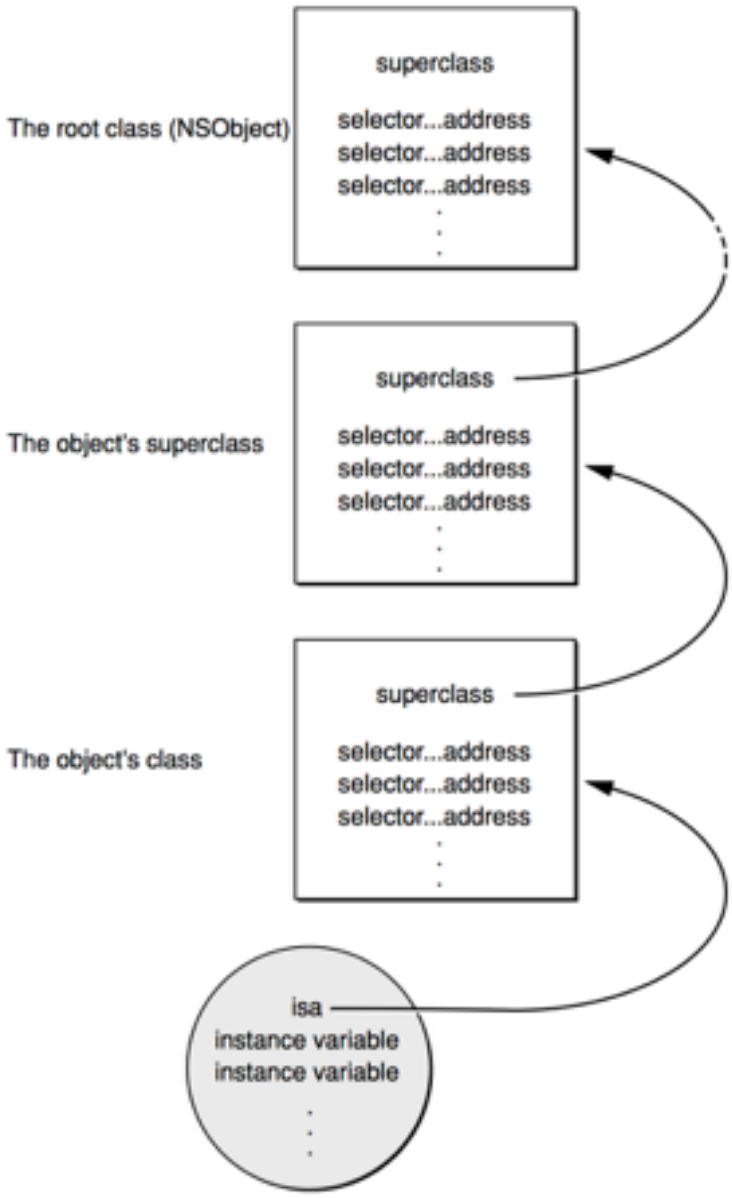
5) 从父类中寻找,直到根类：NSObject 类。找到了就将方法加入对应类的 cache 表中；如果仍未找到，则要进入动态方法决议(resolveInstanceMethod)。

6) 如果动态方法决议仍不能解决问题，只能进行最后一次尝试，进入消息转发流程 (forwardingTargetForSelector)。

7) 如果还不行，崩溃……

class_getMethodImplementation方法的跟踪

下面的图部分展示了这个调用过程



2、函数检索优化措施

1) 通过SEL进行IMP匹配编译器根据每个方法的名称为那个方法生成一个唯一的selector，而 selector 的唯一性，能确保查找方法时提高一部分的效率。

2) cache 缓存

cache 的原则就是缓存那些可能要执行的函数地址，那么下次调用的时候,速度就可以快速很多。这个和 CPU 的各种缓存原理相通。objc_msgSend 首先在 cache list 中找 SEL，没有找到就 在 class method 中找，super class method 中找(当然 super class 也有 cache list)。而 cache 的机制则非常复杂了，由于 Objective-C 是动态语言。所以，这里面还有很多的多线程同步问题，而这些锁又是效率的大敌,相关的内容已经远远超过本文讨论的范围。

如果在缓存中已经有了需要的方法选标，则消息仅仅比函数调用慢一点点。如果程序运行了足够长的时间，几乎每个消息都能在缓存中找到方法实现。程序运行时，缓存也将随着新的消息的增加而增加。据测试，苹果通过这些优化，使消息分发和直接的函数调用效率上的差距已经 相当的小。

五、动态方法解析

1、方法重定向

1)动态添加方法

有时，你只想在运行时才创建某个方法，比如有些信息只有在运行时才能得到。要实现这个效果，你需要重写

+resolveInstanceMethod: 和/或 +resolveClassMethod:。

如果确实增加了一个方法,记得返回 YES。

```
+ (BOOL)resolveInstanceMethod:(SEL)aSelector {
  if (aSelector == @selector(myDynamicMethod)) {
        class_addMethod(self, aSelector, (IMP)myDynamicIMP, "v@:");
        return YES;
  }
  return [super resolveInstanceMethod:aSelector];
}
```

2)方法的重定向Objective-C 2.0 提供了@dynamic 关键字,它的作用为:

a.告诉编译器不要创建实现属性所用的实例变量

b.告诉编译器不要创建属性的 get 和 setter 方法 如果我们在@interface 接口文件中声明了一个属性，如下所示:

@property (nonatomic, retain) NSString *name;

默认情况下，编译器会为当前类自动生成一个 NSString *_name 的实例变量(如果想改变实例变量的名称可以用@synthesize 关键字)，同时会生成两个名为

-(NSString *)name (void)

-setName:(NSString *)aName

的存取方法。

而@dynamic 关键字就是告诉编译器不要做这些事，同时在使用了存取方法时也不要报错，即让编译器相信存取方法会在运行时找到。比如在@implementation 文件中做了如下声明:@dynamic name;

如果使用了 name 属性的 setter 方法，又不想在运行时崩溃，就可以进行一下操作:

```
 void dynamicMethodIMP(id self, SEL _cmd){
// implementation ....
}
+ (BOOL)resolveInstanceMethod:(SEL)sel {
  NSLog(@"sel is %@", NSStringFromSelector(sel));
  if(sel == @selector(name)){
        class_addMethod([self class],sel,(IMP)dynamicMethodIMP,"v@:");
        return YES;
  }
  return [super resolveInstanceMethod:sel];
}
```

在 resolveInstanceMethod 的实现中，我们通过 class_addMethod 方法动态的向当前对象增加了 dynamicMethodIMP 函数，来代替

-(void)setName:(NSString *)name

的实现部分，从而 达到了动态生成 name 属性方法的目的。

值得说明的是:

a.在上个例子中，我们自己实现了-(void)setName:(NSString *)name 方法，则在运行的时候, 调用完我们实现的

-(void)setName:(NSString *)name

方法后，运行时系统仍然会调

+(BOOL) resolveInstanceMethod:(SEL) sel

方法，只不过这里的 sel 会变成_doZombieMe，从而我们实现重定向的 if 分支就进不去了，即我们实现的方法不会被覆盖。

b."v@:"属于 Objective-C 类型编码的内容,感兴趣的同学可以自己 google 一下。

注意：resolveInstanceMethod:是 NSObject 根类提供的类方法，调用时机为被调用的方法实现部分没有找，而消息转发机制启动之前的这个中间时刻。

2、消息转发机制

当对象无法响应发送给他的消息时怎么办?

runtime 提供了消息转发机制来处理该问题。 当外部调用的某个方法对象没有实现,而且 resolveInstanceMethod 方法中也没有做重定向处理时，就会触发

-(id)forwardingTargetForSelector:(SEL)aSelector

方法。在该方法中你可以去将不能响应的消息转发给其他的 target，返回值就是该 target 的值，即最终 aSelector 将会被转发给返回的 target。代码如下:

```
- (id)forwardingTargetForSelector:(SEL)aSelector{
  if (aSelector == @selector(uppercaseString)) {
        return @"Hello";
  }else {
        return nil;
  }
}
```

六、对象、类的操作函数：

如果像静态语言一样对象属性及方法在编译的时候就已经确定了，那么在此基础上再维护一个运行时系统就显得多此一举了，因此在OC中对象的类型及其属性，方法，

协议等内容到运行时才能确定，说明在运行时之前你可以对对象的属性，方法，协议等进行“编辑”，所谓“编辑”就包括“增删改查”等操作。那么我们来看一下OC都提供了那些方法是我们能够对对象的哪些属性进行什么样的修改：

针对class的操作函数如：

class_addIvar, class_addMethod, class_addProperty 和 class_addProtocol 允许重建 classe。

class_copyIvarList, class_copyMethodList,class_copyProtocolList 和 class_copyPropertyList 能拿到一个 class 的所有内容。

class_getClassMethod, class_getClassVariable, class_getInstanceMethod,class_getInstan ceVariable, class_getMethodImplementation 和 class_getProperty 返回单个内容。

也有一些通用的自省方法,如class_conformsToProtocol, class_respondsToSelector,class_getSuperclass。

最后,你可以使用 class_createInstance 来创建一个 object。

针对object的操作函数如：

你可以 get/set ivar, 使用 object_copy 和 object_dispose 来 copy 和 free object 的内存。而且不仅是拿到一个 class，而是可以使用 object_setClass 来改变一个 object 的 class。

针对method的操作函数如：

主要用来自省的方法：method_getName, method_getImplementation, method_getReturnType 等等。 也有一些修改的方法,包括 method_setImplementation 和 method_exchangeImplementations。

针对property的操作函数如：

property属性保存了很大一部分信息。除了拿到名字,你还可以使用 property_getAttributes 来发现 property 的更多信息,如返回值、是否为 atomic、getter/setter 名字、是否为 dynamic、背后使用的 ivar 名字、是否为弱引用。

其他还包括针对protocol，selelctor等函数，这里就不一一列举。

runtime应用场景：http://limboy.me/ios/2013/08/03/dynamic-tips-and-tricks-with-objective-c.html

理解元类：

http://alexliyu.blog.163.com/blog/static/162754496201252115612810/

http://www.cocoawithlove.com/2010/01/what-is-meta-class-in-objective-c.html