# Runtime

## Runtime是什么

Runtime 又叫运行时，是一套底层的 C 语言 API，其为 iOS 内部的核心之一，我们平时编写的 OC 代码，底层都是基于它来实现的。比如：

```
[receiver message];
// 底层运行时会被编译器转化为：
objc_msgSend(receiver, selector)
// 如果其还有参数比如：
[receiver message:(id)arg...];
// 底层运行时会被编译器转化为：
objc_msgSend(receiver, selector, arg1, arg2, ...)
```

## 为什么需要Runtime

- Objective-C 是一门动态语言，它会将一些工作放在代码运行时才处理而并非编译时。也就是说，有很多类和成员变量在我们编译的时是不知道的，而在运行时，我们所编写的代码会转换成完整的确定的代码运行。

- 因此，编译器是不够的，我们还需要一个运行时系统(Runtime system)来处理编译后的代码。

- Runtime 基本是用 C 和汇编写的，由此可见苹果为了动态系统的高效而做出的努力。苹果和 GNU 各自维护一个开源的 Runtime 版本，这两个版本之间都在努力保持一致。

## Runtime 的作用

OC 在三种层面上与 Runtime 系统进行交互：

```
1.通过 Objective-C 源代码
    只需要编写 OC 代码，Runtime 系统自动在幕后搞定一切，调用方法，编译器会将 OC 代码转换成运行时代码，在运行时确

2.通过 Foundation 框架的 NSObject 类定义的方法
    Cocoa 程序中绝大部分类都是 NSObject 类的子类，所以都继承了 NSObject 的行为。(NSProxy 类时个例外，它是个抽象
    一些情况下，NSObject 类仅仅定义了完成某件事情的模板，并没有提供所需要的代码。例如 - description方法，该方法
    还有一些NSObject的方法可以从Runtime系统中获取信息，允许对象进行自我检查。例如：
    -class方法返回对象的类；
    -isKindOfClass: 和 -isMemberOfClass: 方法检查对象是否存在于指定的类的继承体系中(是否是其子类或者父类或者当
    -respondsToSelector: 检查对象能否响应指定的消息；
    -conformsToProtocol:检查对象是否实现了指定协议类的方法；
    -methodForSelector: 返回指定方法实现的地址。

3.通过对 Runtime 库函数的直接调用
    Runtime 系统是具有公共接口的动态共享库。头文件存放于/usr/include/objc目录下，这意味着我们使用时只需要引入ob
    许多函数可以让你使用纯 C 代码来实现 Objc 中同样的功能。除非是写一些 Objc 与其他语言的桥接或是底层的 debug 工
```

# Runtime的相关术语

1.SEL
    它是selector在 Objc 中的表示(Swift 中是 Selector 类)。selector 是方法选择器，其实作用就和名字一样，日常生
typedef struct objc_selector *SEL;
我们可以看出它是个映射到方法的 C 字符串，你可以通过 Objc 编译器器命令@selector() 或者 Runtime 系统的 sel_regis
注意：不同类中相同名字的方法所对应的 selector 是相同的，由于变量的类型不同，所以不会导致它们调用方法实现混乱。

2.id
id 是一个参数类型，它是指向某个类的实例的指针。定义如下：
typedef struct objc_object *id;
struct objc_object { Class isa; };
以上定义，看到 objc_object 结构体包含一个 isa 指针，根据 isa 指针就可以找到对象所属的类。
注意：isa 指针在代码运行时并不总指向实例对象所属的类型，所以不能依靠它来确定类型，要想确定类型还是需要用对象的 -c

3.Class
typedef struct objc_class *Class;
Class 其实是指向 objc_class 结构体的指针。objc_class 的数据结构如下：

```
struct objc_class {
    Class isa  OBJC_ISA_AVAILABILITY;
#if !__OBJC2__
    Class super_class                                       OBJC2_UNAVAILABLE;
    const char *name                                        OBJC2_UNAVAILABLE;
    long version                                            OBJC2_UNAVAILABLE;
    long info                                               OBJC2_UNAVAILABLE;
    long instance_size                                      OBJC2_UNAVAILABLE;
    struct objc_ivar_list *ivars                            OBJC2_UNAVAILABLE;
    struct objc_method_list **methodLists                   OBJC2_UNAVAILABLE;
    struct objc_cache *cache                                OBJC2_UNAVAILABLE;
    struct objc_protocol_list *protocols                    OBJC2_UNAVAILABLE;
#endif
} OBJC2_UNAVAILABLE;
```

从 objc_class 可以看到，一个运行时类中关联了它的父类指针、类名、成员变量、方法、缓存以及附属的协议。

其中 objc_ivar_list 和 objc_method_list 分别是成员变量列表和方法列表：

```
// 成员变量列表
struct objc_ivar_list {
    int ivar_count                                          OBJC2_UNAVAILABLE;
#ifdef __LP64__
    int space                                               OBJC2_UNAVAILABLE;
#endif
    /* variable length structure */
    struct objc_ivar ivar_list[1]                           OBJC2_UNAVAILABLE;
}                                                           OBJC2_UNAVAILABLE;

// 方法列表
struct objc_method_list {
    struct objc_method_list *obsolete                       OBJC2_UNAVAILABLE;

    int method_count                                        OBJC2_UNAVAILABLE;
#ifdef __LP64__
    int space                                               OBJC2_UNAVAILABLE;
#endif
    /* variable length structure */
    struct objc_method method_list[1]                       OBJC2_UNAVAILABLE;
```

```
}
```

由此可见，我们可以动态修改 *methodList 的值来添加成员方法，这也是 Category 实现的原理，同样解释了 Category 不能

objc_ivar_list 结构体用来存储成员变量的列表，而 objc_ivar 则是存储了单个成员变量的信息；同理，objc_method_list

值得注意的时，objc_class 中也有一个 isa 指针，这说明 Objc 类本身也是一个对象。为了处理类和对象的关系，Runtime 库

我们所熟悉的类方法，就源自于 Meta Class。我们可以理解为类方法就是类对象的实例方法。每个类仅有一个类对象，而每个类

当你发出一个类似 [NSObject alloc](类方法) 的消息时，实际上，这个消息被发送给了一个类对象(Class Object)，这个类对

所以当 [NSObject alloc] 这条消息发送给类对象的时候，运行时代码 objc_msgSend() 会去它元类中查找能够响应消息的方法

最后 objc_class 中还有一个 objc_cache ，缓存，它的作用很重要，后面会提到。

4.Method
Method 代表类中某个方法的类型

```
typedef struct objc_method *Method;
struct objc_method {
    SEL method_name                                          OBJC2_UNAVAILABLE;
    char *method_types                                       OBJC2_UNAVAILABLE;
    IMP method_imp                                           OBJC2_UNAVAILABLE;
}
```

objc_method 存储了方法名，方法类型和方法实现：

方法名类型为 SEL
方法类型 method_types 是个 char 指针，存储方法的参数类型和返回值类型
method_imp 指向了方法的实现，本质是一个函数指针
Ivar
Ivar 是表示成员变量的类型。

```
typedef struct objc_ivar *Ivar;
struct objc_ivar {
    char *ivar_name                                          OBJC2_UNAVAILABLE;
    char *ivar_type                                          OBJC2_UNAVAILABLE;
    int ivar_offset                                          OBJC2_UNAVAILABLE;
#ifdef __LP64__
    int space                                                OBJC2_UNAVAILABLE;
#endif
}
```

其中 ivar_offset 是基地址偏移字节

5.IMP
IMP在objc.h中的定义是：

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

它就是一个函数指针，这是由编译器生成的。当你发起一个 ObjC 消息之后，最终它会执行的那段代码，就是由这个函数指针指定

如果得到了执行某个实例某个方法的入口，我们就可以绕开消息传递阶段，直接执行方法，这在后面 Cache 中会提到。

你会发现 IMP 指向的方法与 objc_msgSend 函数类型相同，参数都包含 id 和 SEL 类型。每个方法名都对应一个 SEL 类型的

而一个确定的方法也只有唯一的一组 id 和 SEL 参数。

6.Cache
Cache 定义如下：

```
typedef struct objc_cache *Cache
struct objc_cache {
    unsigned int mask /* total = mask + 1 */                 OBJC2_UNAVAILABLE;
```

```
    unsigned int occupied                                    OBJC2_UNAVAILABLE;
    Method buckets[1]                                        OBJC2_UNAVAILABLE;
};
```

Cache 为方法调用的性能进行优化，每当实例对象接收到一个消息时，它不会直接在 isa 指针指向的类的方法列表中遍历查找能

Runtime 系统会把被调用的方法存到 Cache 中，如果一个方法被调用，那么它有可能今后还会被调用，下次查找的时候就会效率

7.Property

```
typedef struct objc_property *Property;
typedef struct objc_property *objc_property_t;//这个更常用
```

可以通过class_copyPropertyList 和 protocol_copyPropertyList 方法获取类和协议中的属性：

```
objc_property_t *class_copyPropertyList(Class cls, unsigned int *outCount)
objc_property_t *protocol_copyPropertyList(Protocol *proto, unsigned int *outCount)
```

注意：

返回的是属性列表，列表中每个元素都是一个 objc_property_t 指针

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
/** 姓名 */
@property (strong, nonatomic) NSString *name;
/** age */
@property (assign, nonatomic) int age;
/** weight */
@property (assign, nonatomic) double weight;
@end
```

以上是一个 Person 类，有3个属性。让我们用上述方法获取类的运行时属性。

```
    unsigned int outCount = 0;

    objc_property_t *properties = class_copyPropertyList([Person class], &outCount);

    NSLog(@"%d", outCount);

    for (NSInteger i = 0; i < outCount; i++) {
        NSString *name = @(property_getName(properties[i]));
        NSString *attributes = @(property_getAttributes(properties[i]));
        NSLog(@"%@--------%@", name, attributes);
    }
```

打印结果如下：

```
test[2321:451525] 3
test[2321:451525] name--------T@"NSString",&,N,V_name
test[2321:451525] age--------Ti,N,V_age
test[2321:451525] weight--------Td,N,V_weight
```

property_getName 用来查找属性的名称，返回 c 字符串。property_getAttributes 函数挖掘属性的真实名称和 @encode 类

```
objc_property_t class_getProperty(Class cls, const char *name)
objc_property_t protocol_getProperty(Protocol *proto, const char *name, BOOL isRequiredProperty, BOOL is
```

class_getProperty 和 protocol_getProperty 通过给出属性名在类和协议中获得属性的引用。

## Runtime与消息

- 一些 Runtime 术语讲完了，接下来就要说到消息了。体会苹果官方文档中的 messages aren't bound to method implementations until Runtime。消息直到运行时才会与方法实现进行绑定。

- 这里要清楚一点，objc_msgSend 方法看清来好像返回了数据，其实objc_msgSend 从不返回数据，而是你的方法在运行时实现被调用后才会返回数据。下面详细叙述消息发送的步骤：

  - 首先检测这个 selector 是不是要忽略。比如 Mac OS X 开发，有了垃圾回收就不理会 retain，release 这些函数。
  - 检测这个 selector 的 target 是不是 nil，Objc 允许我们对一个 nil 对象执行任何方法不会 Crash，因为运行时会被忽略掉。
  - 如果上面两步都通过了，那么就开始查找这个类的实现 IMP，先从 cache 里查找，如果找到了就运行对应的函数去执行相应的代码。
  - 如果 cache 找不到就找类的方法列表中是否有对应的方法。
  - 如果类的方法列表中找不到就到父类的方法列表中查找，一直找到 NSObject 类为止。
    如果还找不到，就要开始进入动态方法解析了，后面会提到。

- 在消息的传递中，编译器会根据情况在 objc_msgSend ， objc_msgSend_stret ， objc_msgSendSuper ， objc_msgSendSuper_stret 这四个方法中选择一个调用。如果消息是传递给父类，那么会调用名字带有 Super 的函数，如果消息返回值是数据结构而不是简单值时，会调用名字带有 stret 的函数。

## 方法中的隐藏参数

```
疑问：
我们经常用到关键字 self ，但是 self 是如何获取当前方法的对象呢？
其实，这也是 Runtime 系统的作用，self 实在方法运行时被动态传入的。

当 objc_msgSend 找到方法对应实现时，它将直接调用该方法实现，并将消息中所有参数都传递给方法实现，同时，它还将传递

接受消息的对象(self 所指向的内容，当前方法的对象指针)
方法选择器(_cmd 指向的内容，当前方法的 SEL 指针)
因为在源代码方法的定义中，我们并没有发现这两个参数的声明。它们时在代码被编译时被插入方法实现中的。尽管这些参数没有

这两个参数中， self更实用。它是在方法实现中访问消息接收者对象的实例变量的途径。

这时我们可能会想到另一个关键字 super ，实际上 super 关键字接收到消息时，编译器会创建一个 objc_super 结构体：

struct objc_super { id receiver; Class class; };
这个结构体指明了消息应该被传递给特定的父类。 receiver 仍然是 self 本身，当我们想通过 [super class] 获取父类时，

// 这句话并不能获取父类的类型，只能获取当前类的类型名
NSLog(@"%@", NSStringFromClass([super class]));
获取方法地址
NSObject 类中有一个实例方法：methodForSelector，你可以用它来获取某个方法选择器对应的 IMP ，举个例子：

void (*setter)(id, SEL, BOOL);
int i;

setter = (void (*)(id, SEL, BOOL))[target
    methodForSelector:@selector(setFilled:)];
```

```
for ( i = 0 ; i < 1000 ; i++ )
    setter(targetList[i], @selector(setFilled:), YES);
```

当方法被当做函数调用时，两个隐藏参数也必须明确给出，上面的例子调用了1000次函数，你也可以尝试给 target 发送1000次

虽然可以更高效的调用方法，但是这种做法很少用，除非时需要持续大量重复调用某个方法的情况，才会选择使用以免消息发送泛

注意：
methodForSelector:方法是由 Runtime 系统提供的，而不是 Objc 自身的特性
动态方法解析
你可以动态提供一个方法实现。如果我们使用关键字 @dynamic 在类的实现文件中修饰一个属性，表明我们会为这个属性动态提供

```
@dynamic propertyName;
```

这时，我们可以通过分别重载 resolveInstanceMethod: 和 resolveClassMethod: 方法添加实例方法实现和类方法实现。

当 Runtime 系统在 Cache 和类的方法列表(包括父类)中找不到要执行的方法时，Runtime 会调用 resolveInstanceMethod:

```
void dynamicMethodIMP(id self, SEL _cmd) {
    // implementation ....
}
@implementation MyClass
+ (BOOL)resolveInstanceMethod:(SEL)aSEL
{
    if (aSEL == @selector(resolveThisMethodDynamically)) {
          class_addMethod([self class], aSEL, (IMP) dynamicMethodIMP, "v@:");
          return YES;
    }
    return [super resolveInstanceMethod:aSEL];
}
@end
```

上面的例子为 resolveThisMethodDynamically 方法添加了实现内容，就是 dynamicMethodIMP 方法中的代码。其中 "v@:"

注意：
动态方法解析会在消息转发机制侵入前执行，动态方法解析器将会首先给予提供该方法选择器对应的 IMP 的机会。如果你想让该

## 消息转发

1.重定向
消息转发机制执行前，Runtime 系统允许我们替换消息的接收者为其他对象。通过 - (id)forwardingTargetForSelector:(SE

```
- (id)forwardingTargetForSelector:(SEL)aSelector
{
    if(aSelector == @selector(mysteriousMethod:)){
        return alternateObject;
    }
    return [super forwardingTargetForSelector:aSelector];
}
```

如果此方法返回 nil 或者 self，则会计入消息转发机制(forwardInvocation:)，否则将向返回的对象重新发送消息。

2.转发
当动态方法解析不做处理返回 NO 时，则会触发消息转发机制。这时 forwardInvocation: 方法会被执行，我们可以重写这个方

```
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    if ([someOtherObject respondsToSelector:
            [anInvocation selector]])
        [anInvocation invokeWithTarget:someOtherObject];
    else
```

```
        [super forwardInvocation:anInvocation];
}
```

唯一参数是个 NSInvocation 类型的对象，该对象封装了原始的消息和消息的参数。我们可以实现 forwardInvocation: 方法

注意：参数 anInvocation 是从哪来的？
在 forwardInvocation: 消息发送前，Runtime 系统会向对象发送methodSignatureForSelector: 消息，并取到返回的方法
当一个对象由于没有相应的方法实现而无法相应某消息时，运行时系统将通过 forwardInvocation: 消息通知该对象。每个对象

forwardInvocation: 方法就是一个不能识别消息的分发中心，将这些不能识别的消息转发给不同的接收对象，或者转发给同一个

注意：
forwardInvocation:方法只有在消息接收对象中无法正常响应消息时才会被调用。所以，如果我们向往一个对象将一个消息转发
转发和多继承
转发和继承相似，可用于为 Objc 编程添加一些多继承的效果。就像下图那样，一个对象把消息转发出去，就好像它把另一个对象

这使得在不同继承体系分支下的两个类可以实现“继承”对方的方法，在上图中 Warrior 和 Diplomat 没有继承关系，但是 Warr

消息转发弥补了 Objc 不支持多继承的性质，也避免了因为多继承导致单个类变得臃肿复杂。

## 转发与继承

虽然转发可以实现继承的功能，但是 NSObject 还是必须表面上很严谨，像 respondsToSelector: 和 isKindOfClass: 这类

Warrior 对象被问到是否能响应 negotiate消息：

```
if ( [aWarrior respondsToSelector:@selector(negotiate)] )
    ...
```

回答当然是 NO， 尽管它能接受 negotiate 消息而不报错，因为它靠转发消息给 Diplomat 类响应消息。

如果你就是想要让别人以为 Warrior 继承到了 Diplomat 的 negotiate 方法，你得重新实现 respondsToSelector: 和 isK

```
- (BOOL)respondsToSelector:(SEL)aSelector
{
    if ( [super respondsToSelector:aSelector] )
        return YES;
    else {
        /* Here, test whether the aSelector message can     *
         * be forwarded to another object and whether that  *
         * object can respond to it. Return YES if it can.  */
    }
    return NO;
}
```

除了 respondsToSelector: 和 isKindOfClass: 之外，instancesRespondToSelector: 中也应该写一份转发算法。如果使

如果一个对象想要转发它接受的任何远程消息，它得给出一个方法标签来返回准确的方法描述 methodSignatureForSelector:，

```
- (NSMethodSignature*)methodSignatureForSelector:(SEL)selector
{
    NSMethodSignature* signature = [super methodSignatureForSelector:selector];
    if (!signature) {
       signature = [surrogate methodSignatureForSelector:selector];
    }
    return signature;
}
```

## 健壮的实例变量(Non Fragile ivars)

```
在 Runtime 的现行版本中，最大的特点就是健壮的实例变量了。当一个类被编译时，实例变量的内存布局就形成了，它表明访问

上图左是 NSObject 类的实例变量布局。右边是我们写的类的布局。这样子有一个很大的缺陷，就是缺乏拓展性。哪天苹果更新了

我们自定义的类的区域和父类的区域重叠了。只有苹果将父类改为以前的布局才能拯救我们，但这样导致它们不能再拓展它们的框

在健壮的实例变量下，编译器生成的实例变量布局跟以前一样，但是当 Runtime 系统检测到与父类有部分重叠时它会调整你新添

注意：
在健壮的实例变量下，不要使用 siof(SomeClass)，而是用 class_getInstanceSize([SomeClass class]) 代替；也不要使
总结
我们让自己的类继承自 NSObject 不仅仅是因为基类有很多复杂的内存分配问题，更是因为这使得我们可以享受到 Runtime 系统

虽然平时我们很少会考虑一句简单的调用方法，发送消息底层所做的复杂的操作，但深入理解 Runtime 系统的细节使得我们可以
```

## runtime实现的机制是什么,怎么用，一般用于干嘛. 你还能记得你所使用的相关的头文件或者某些方法的名称吗?

- 需要导入<objc/message.h><objc/runtime.h>
- runtime，运行时机制，它是一套C语言库
- 实际上我们编写的所有OC代码，最终都是转成了runtime库的东西，比如类转成了runtime库里面的结构体等数据类型，方法转成了runtime库里面的C语言函数，平时调方法都是转成了objc_msgSend函数（所以说OC有个消息发送机制）
- 因此，可以说runtime是OC的底层实现，是OC的幕后执行者
- 有了runtime库，能做什么事情呢？runtime库里面包含了跟类、成员变量、方法相关的API，比如获取类里面的所有成员变量，为类动态添加成员变量，动态改变类的方法实现，为类动态添加新的方法等
- 因此，有了runtime，想怎么改就怎么改

## Objective-C 如何对已有的方法，添加自己的功能代码以实现类似记录日志这样的功能?

这题目主要考察的是runtime如何交换方法。先在分类中添加一个方法,注意不能重写系统方法,会覆盖

```
+(NSString *)myLog
```

```
{

  // 这里写打印行号,什么方法,哪个类调用等等


}

// 加载分类到内存的时候调用

    +(void)load

    {

    // 获取imageWithName方法地址
    Method description = class_getClassMethod(self, @selector(description));

    // 获取imageWithName方法地址
    Method myLog = class_getClassMethod(self, @selector(myLog));

    // 交换方法地址，相当于交换实现方式
    method_exchangeImplementations(description, myLog);
    }
```

## 如何让 Category 支持属性?

使用runtime可以实现

```
头文件

@interface NSObject (test)

@property (nonatomic, copy) NSString *name;

@end

.m文件

@implementation NSObject (test)

// 定义关联的key

static const char *key = "name";

-(NSString *)name

{
    // 根据关联的key，获取关联的值。

   return objc_getAssociatedObject(self, key);

}

-(void)setName:(NSString *)name

{

    // 第一个参数：给哪个对象添加关联
    // 第二个参数：关联的key，通过这个key获取
```

```
    // 第三个参数：关联的value
    // 第四个参数:关联的策略
    objc_setAssociatedObject(self, key, name, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}
```

## Toll-Free Bridging 是什么？什么情况下会使用?

- Toll-Free Bridging用于在Foundation对象与Core Foundation对象之间交换数据,俗称桥接
- 在ARC环境下,Foundation对象转成 Core Foundation对象
- 使用__bridge桥接以后ARC会自动管理2个对象
- 使用__bridge_retained桥接需要手动释放Core Foundation对象
- 在ARC环境下, Core Foundation对象转成 Foundation对象
- 使用__bridge桥接,如果Core Foundation对象被释放,Foundation对象也同时不能使用了,需要手动管理Core Foundation对象
- 使用__bridge_transfer桥接,系统会自动管理2个对象

## performSelector:withObject:afterDelay: 内部大概是怎么实现的，有什么注意事项么?

- 创建一个定时器,时间结束后系统会使用runtime通过方法名称(Selector本质就是方法名称)去方法列表中找到对应的方法实现并调用方法
- 注意事项
    - 调用performSelector:withObject:afterDelay:方法时,先判断希望调用的方法是否存在respondsToSelector:
    - 这个方法是异步方法,必须在主线程调用,在子线程调用永远不会调用到想调用的方法

## 什么是 Method Swizzle（黑魔法），什么情况下会使用?

- 在没有一个类的实现源码的情况下，想改变其中一个方法的实现，除了继承它重写、和借助类别重名方法暴力抢先之外，还有更加灵活的方法Method Swizzle。
- Method swizzling指的是改变一个已存在的选择器对应的实现的过程。OC中方法的调用能够在运行时通过改变——通过改变类的调度表（dispatch table）中选择器到最终函数间的映射关系。

- 在OC中调用一个方法，其实是向一个对象发送消息，查找消息的唯一依据是selector的名字。利用OC的动态特性，可以实现在运行时偷换selector对应的方法实现。
- 每个类都有一个方法列表，存放着selector的名字和方法实现的映射关系。IMP有点类似函数指针，指向具体的Method实现。
- 我们可以利用 method_exchangeImplementations 来交换2个方法中的IMP，
- 我们可以利用 class_replaceMethod 来修改类，
- 我们可以利用 method_setImplementation 来直接设置某个方法的IMP，
- 归根结底，都是偷换了selector的IMP

## 能否向编译后得到的类中增加实例变量？能否向运行时创建的类中添加实例变量？为什么？

```
- 不能向编译后得到的类中增加实例变量；
- 能向运行时创建的类中添加实例变量；

解释如下：
因为编译后的类已经注册在 runtime 中，类结构体中的 objc_ivar_list 实例变量的链表 和 instance_size 实例变量的内存
运行时创建的类是可以添加实例变量，调用 class_addIvar 函数。但是得在调用 objc_allocateClassPair 之后，objc_regi
```

## 为什么其他语言里叫函数调用， objective c里则是给对象发消息（或者谈下对runtime的理解）

```
先来看看怎么理解发送消息的含义：
[receiver message]会被编译器转化为：
objc_msgSend(receiver, selector)
如果消息含有参数，则为：
objc_msgSend(receiver, selector, arg1, arg2, ...)
如果消息的接收者能够找到对应的selector，那么就相当于直接执行了接收者这个对象的特定方法；否则，消息要么被转发，或是
现在可以看出[receiver message]真的不是一个简简单单的方法调用。因为这只是在编译阶段确定了要向接收者发送message这
OC 的 Runtime 铸就了它动态语言的特性，Objc Runtime使得C具有了面向对象能力，在程序运行时创建，检查，修改类、对象和
顺便附上OC中一个类的数据结构 /usr/include/objc/runtime.h
struct objc_class {

Class isa OBJC_ISA_AVAILABILITY; //isa指针指向Meta Class，因为Objc的类的本身也是一个Object，为了处理这个关系

 #if !__OBJC2__
     Class super_class
OBJC2_UNAVAILABLE; // 父类
     const char *name
OBJC2_UNAVAILABLE; // 类名
     long version
OBJC2_UNAVAILABLE; // 类的版本信息，默认为0
     long info
OBJC2_UNAVAILABLE; // 类信息，供运行期使用的一些位标识
     long instance_size
OBJC2_UNAVAILABLE; // 该类的实例变量大小
```

```
     struct objc_ivar_list *ivars
OBJC2_UNAVAILABLE; // 该类的成员变量链表
     struct objc_method_list **methodLists
OBJC2_UNAVAILABLE; // 方法定义的链表
     struct objc_cache *cache
OBJC2_UNAVAILABLE; // 方法缓存，对象接到一个消息会根据isa指针查找消息对象，这时会在method       Lists中遍历，
    struct objc_protocol_list *protocols
OBJC2_UNAVAILABLE; // 协议链表
 #endif

} OBJC2_UNAVAILABLE;
OC中一个类的对象实例的数据结构（/usr/include/objc/objc.h）：
typedef struct objc_class *Class;      /// Represents an instance of a class.      struct objc_object {
    Class isa
OBJC_ISA_AVAILABILITY;
};      /// A pointer to an instance of a class.
typedef struct objc_object *id;

向object发送消息时，Runtime库会根据object的isa指针找到这个实例object所属于的类，然后在类的方法列表以及父类方法列
然后再来看看消息发送的函数：objc_msgSend函数
在引言中已经对objc_msgSend进行了一点介绍，看起来像是objc_msgSend返回了数据，其实objc_msgSend从不返回数据而是你
检测这个 selector 是不是要忽略的。比如 Mac OS X 开发，有了垃圾回收就不理会 retain,release 这些函数了。
检测这个 target 是不是 nil 对象。ObjC 的特性是允许对一个 nil 对象执行任何一个方法不会 Crash，因为会被忽略掉。
如果上面两个都过了，那就开始查找这个类的 IMP，先从 cache 里面找，完了找得到就跳到对应的函数去执行。
如果 cache 找不到就找一下方法分发表。
如果分发表找不到就到超类的分发表去找，一直找，直到找到NSObject类为止。
如果还找不到就要开始进入动态方法解析了，后面会提到。
后面还有：
动态方法解析resolveThisMethodDynamically
消息转发forwardingTargetForSelector
```

## runtime如何实现weak属性?

- 通过关联属性来实现：
- // 声明一个weak属性，这里假设delegate，其实weak关键字可以不使用，
- // 因为我们重写了getter/setter方法
- @property (nonatomic, weak) id delegate;
-
- - (id)delegate {
-   return objc_getAssociatedObject(self, @"__delegate__key");
- }
-
- // 指定使用OBJC_ASSOCIATION_ASSIGN，官方注释是：
- // Specifies a weak reference to the associated object.
- // 也就是说对于对象类型，就是weak了
- - (void)setDelegate:(id)delegate {
-   objc_setAssociatedObject(self, @"__delegate__key", delegate, OBJC_ASSOCIATION_ASSIGN);
- }
- 通过objc_storeWeak函数来实现，不过这种方式几乎没有遇到有人这么使用过，因为这里不细说了。

## runtime如何通过selector找到对应的IMP地址?

- 每个selector都与对应的IMP是一一对应的关系，通过selector就可以直接找到对应的IMP：

## objc_msgForward函数是做什么的，直接调用它将会发生什么?

```
_objc_msgForward是IMP类型，用于消息转发的：当向一个对象发送一条消息，但它并没有实现的时候，_objc_msgForward会尝
IMP msgForward =  _objc_msgForward;
如果手动调用objcmsgForward，将跳过查找IMP的过程，而是直接触发“消息转发”，进入如下流程：

• 第一步：+ (BOOL)resolveInstanceMethod:(SEL)sel实现方法，指定是否动态添加方法。若返回NO，则进入下一步，若返
• 第二步：在第一步返回的是NO时，就会进入- (id)forwardingTargetForSelector:(SEL)aSelector方法，这是运行时给我
• 第三步：若第二步返回的是nil，则我们首先要通过- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSel
• 第四步：当第三步返回方法方法签名后，就会调用- (void)forwardInvocation:(NSInvocation *)anInvocation方法，
• 第五步：若没有实现- (void)forwardInvocation:(NSInvocation *)anInvocation方法，那么会进入- (void)doesNot
```

## runtime如何实现weak变量的自动置nil?

```
runtime对注册的类会进行布局，对于weak对象会放入一个hash表中。 用weak指向的对象内存地址作为key，当此对象的引用计数
weak修饰的指针默认值是nil（在Objective-C中向nil发送消息是安全的）
```

## 动态绑定

- 在运行时确定要调用的方法,动态绑定将调用方法的确定也推迟到运行时。在编译时，方法的调用并不和代码绑定在一起，只有在消实发送出来之后，才确定被调用的代码。通过动态类型和动态绑定技术，代码每次执行都可以得到不同的结果。运行时因子负责确定消息的接收者和被调用的方法。运行时的消息分发机制为动态绑定提供支持。当向一个动态类型确定了的对象发送消息时，运行环境系统会通过接收者的isa指针定位对象的类，并以此为起点确定被调用的方法，方法和消息是动态绑定的。而且，不必在Objective-C 代码中做任何工作，就可以自动获取动态绑定的好处。在每次发送消息时，特别是当消息的接收者是动态类型已经确定的对象时，动态绑定就会例行而透明地发生

文章如有问题，请留言，我将及时更正。

**满地打滚卖萌求赞，如果本文帮助到你，轻点下方的红心，给作者君增加更新的动力。**