Objective-C Runtime

杨萧玉 • 发表于 2014-11-05

本文详细整理了 Cocoa 的 Runtime 系统的知识，它使得 Objective-C 如虎添翼，具备了灵活的动态特性，使这门古老的语言焕发生机。主要内容如下：

引言

曾经觉得Objc特别方便上手，面对着 Cocoa 中大量 API，只知道简单的查文档和调用。还记得初学 Objective-C 时把`[receiver message]`当成简单的方法调用，而无视了**“发送消息”**这句话的深刻含义。其实`[receiver message]`会被编译器转化为：

```
1  objc_msgSend(receiver, selector)
```

如果消息含有参数，则为：

```
1  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，或是临时向接收者动态添加这个selector对应的实现内容，要么就干脆玩完崩溃掉。

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这条消息，而receive将要如何响应这条消息，那就要看运行时发生的情况来决定了。

Objective-C 的 Runtime 铸就了它动态语言的特性，这些深层次的知识虽然平时写代码用的少一些，但是却是每个 Objc 程序员需要了解的。

简介

因为Objc是一门动态语言，所以它总是想办法把一些决定工作从编译连接推迟到运行时。也就是说只有编译器是不够的，还需要一个运行时系统 (runtime system) 来执行编译后的代码。这就是 Objective-C Runtime 系统存在的意义，它是整个Objc运行框架的一块基石。

Runtime其实有两个版本:“modern”和 “legacy”。我们现在用的 Objective-C 2.0 采用的是现行(Modern)版的Runtime系统，只能运行在 iOS 和 OS X 10.5 之后的64位程序中。而OS X较老的32位程序仍采用 Objective-C 1中的（早期）Legacy 版本的 Runtime 系统。这两个版本最大的区别在于当你更改一个类的实例变量的布局时，在早期版本中你需要重新编译它的子类，而现行版就不需要。

Runtime基本是用C和汇编写的，可见苹果为了动态系统的高效而作出的努力。你可以在这里下到苹果维护的开源代码。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的保持一致。

与Runtime交互

Objc 从三种不同的层级上与 Runtime 系统进行交互，分别是通过 Objective-C 源代码，通过 Foundation 框架的`NSObject`类定义的方法，通过对 runtime 函数的直接调用。

Objective-C源代码

大部分情况下你就只管写你的Objc代码就行，runtime 系统自动在幕后辛勤劳作着。
还记得引言中举的例子吧，消息的执行会使用到一些编译器为实现动态语言特性而创建的数据结构和函数，Objc中的类、方法和协议等在 runtime 中都由一些数据结构来定义，这些内容在后面会讲到。（比如`objc_msgSend`函数及其参数列表中的`id`和`SEL`都是啥）

NSObject的方法

Cocoa 中大多数类都继承于`NSObject`类，也就自然继承了它的方法。最特殊的例外是`NSProxy`，它是个抽象超类，它实现了一些消息转发有关的方法，可以通过继承它来实现一个其他类的替身类或是虚拟出一个不存在的类，说白了就是领导把自己展现给大家风光无限，但是把活儿都交给幕后小弟去干。

有的`NSObject`中的方法起到了抽象接口的作用，比如`description`方法需要你重载它并为你定义的类提供描述内容。`NSObject`还有些方法能在运行时获得类的信息，并检查一些特性，比如`class`返回对象的类；`isKindOfClass:`和`isMemberOfClass:`则检查对象是否在指定的类继承体系中；`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:`则返回指定方法实现的地址。

Runtime的函数

Runtime 系统是一个由一系列函数和数据结构组成，具有公共接口的动态共享库。头文件存放于/usr/include/objc目录下。许多函数允许你用纯C代码来重复实现 Objc 中同样的功能。虽然有一些方法构成了NSObject类的基础，但是你在写 Objc 代码时一般不会直接用到这些函数的，除非是写一些 Objc 与其他语言的桥接或是底层的debug工作。在Objective-C Runtime Reference中有对 Runtime 函数的详细文档。

Runtime术语

还记得引言中的objc_msgSend:方法吧，它的真身是这样的：

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

下面将会逐渐展开介绍一些术语，其实它们都对应着数据结构。

SEL

objc_msgSend函数第二个参数类型为SEL，它是selector在Objc中的表示类型（Swift中是Selector类）。selector是方法选择器，可以理解为区分方法的 ID，而这个 ID 的数据结构是SEL:

```
1  typedef struct objc_selector *SEL;
```

其实它就是个映射到方法的C字符串，你可以用 Objc 编译器命令@selector()或者 Runtime 系统的sel_registerName函数来获得一个SEL类型的方法选择器。

不同类中相同名字的方法所对应的方法选择器是相同的，即使方法名字相同而变量类型不同也会导致它们具有相同的方法选择器，于是 Objc 中方法命名有时会带上参数类型(NSNumber一堆抽象工厂方法拿走不谢)，Cocoa 中有好多长长的方法哦。

id

objc_msgSend第一个参数类型为id，大家对它都不陌生，它是一个指向类实例的指针：

```
1  typedef struct objc_object *id;
```

那objc_object又是啥呢：

```
1  struct objc_object { Class isa; };
```

objc_object结构体包含一个isa指针，根据isa指针就可以顺藤摸瓜找到对象所属的类。

PS:isa指针不总是指向实例对象所属的类，不能依靠它来确定类型，而是应该用class方法来确定实例对象的类。因为KVO的实现机理就是将被观察对象的isa指针指向一个中间类而不是真实的类，这是一种叫做 **isa-swizzling** 的技术，详见官方文档

Class

之所以说isa是指针是因为Class其实是一个指向objc_class结构体的指针：

```
1  typedef struct objc_class *Class;
```

而objc_class就是我们摸到的那个瓜，里面的东西多着呢：

```
1  struct objc_class {
2      Class isa  OBJC_ISA_AVAILABILITY;
3
4  #if !__OBJC2__
5      Class super_class                                       OBJC2_UNAVAILABLE;
6      const char *name                                        OBJC2_UNAVAILABLE;
7      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;
```

可以看到运行时一个类还关联了它的超类指针，类名，成员变量，方法，缓存，还有附属的协议。

PS:`OBJC2_UNAVAILABLE`之类的宏定义是苹果在 Objc 中对系统运行版本进行约束的黑魔法，为的是兼容非Objective-C 2.0的遗留逻辑，但我们仍能从中获得一些有价值的信息，有兴趣的可以查看源代码。

Objective-C 2.0 的头文件虽然没暴露出`objc_class`结构体更详细的设计，我们依然可以从Objective-C 1.0 的定义中小窥端倪：

在`objc_class`结构体中：`ivars`是`objc_ivar_list`指针；`methodLists`是指向`objc_method_list`指针的指针。也就是说可以动态修改`*methodLists`的值来添加成员方法，这也是Category实现的原理，同样解释了Category不能添加属性的原因。而最新版的 Runtime 源码对这一块的描述已经有很大变化，可以参考下美团技术团队的深入理解Objective-C：Category。
PS：任性的话可以在Category中添加`@dynamic`的属性，并利用运行期动态提供存取方法或干脆动态转发；或者干脆使用关联度对象（AssociatedObject）

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

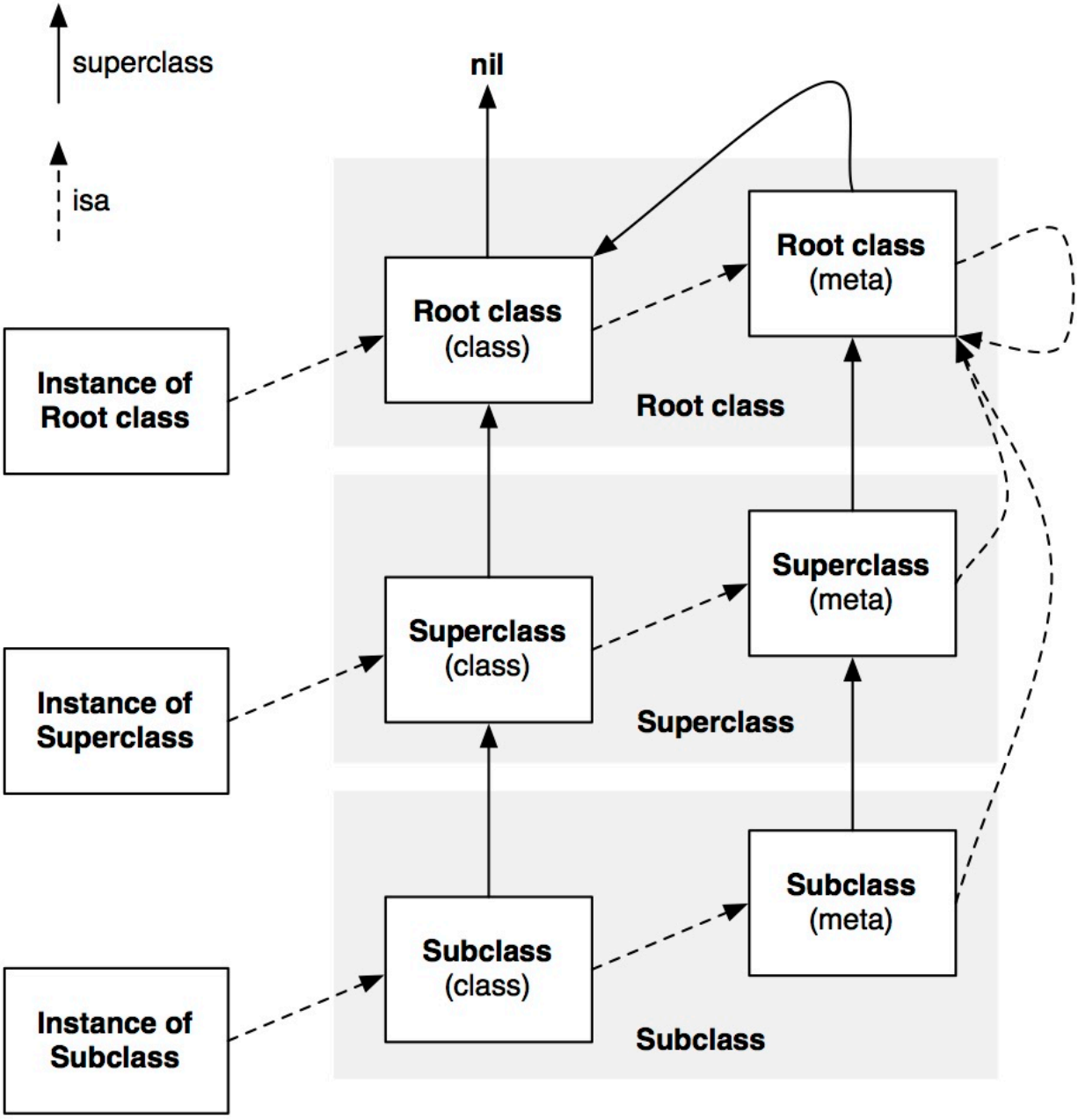
```
struct objc_ivar_list {
    int ivar_count                                      OBJC2_UNAVAILABLE;
#ifdef __LP64__
    int space                                           OBJC2_UNAVAILABLE;
#endif

```

```
7       struct objc_ivar ivar_list[1]                           OBJC2_UNAVAILABLE;
8   }                                                           OBJC2_UNAVAILABLE;
9
10  struct objc_method_list {
11      struct objc_method_list *obsolete                       OBJC2_UNAVAILABLE;
12
13      int method_count                                        OBJC2_UNAVAILABLE;
14  #ifdef __LP64__
15      int space                                               OBJC2_UNAVAILABLE;
16  #endif
17
18      struct objc_method method_list[1]                       OBJC2_UNAVAILABLE;
19  }
```

如果你C语言不是特别好，可以直接理解为`objc_ivar_list`结构体存储着`objc_ivar`数组列表，而`objc_ivar`结构体存储了类的单个成员变量的信息；同理`objc_method_list`结构体存储着`objc_method`数组列表，而`objc_method`结构体存储了类的某个方法的信息。

最后要提到的还有一个`objc_cache`，顾名思义它是缓存，它在`objc_class`的作用很重要，在后面会讲到。

不知道你是否注意到了`objc_class`中也有一个`isa`对象，这是因为一个 ObjC 类本身同时也是一个对象，为了处理类和对象的关系，runtime 库创建了一种叫做元类 (Meta Class) 的东西，类对象所属类型就叫做元类，它用来表述类对象本身所具备的元数据。类方法就定义于此处，因为这些方法可以理解成类对象的实例方法。每个类仅有一个类对象，而每个类对象仅有一个与之相关的元类。当你发出一个类似`[NSObject alloc]`的消息时，你事实上是把这个消息发给了一个类对象 (Class Object)，这个类对象必须是一个元类的实例，而这个元类同时也是一个根元类 (root meta class) 的实例。所有的元类最终都指向根元类为其超类。所有的元类的方法列表都有能够响应消息的类方法。所以当 `[NSObject alloc]` 这条消息发给类对象的时候，`objc_msgSend()`会去它的元类里面去查找能够响应消息的方法，如果找到了，然后对这个类对象执行方法调用。



上图实线是 `super_class` 指针，虚线是`isa`指针。 有趣的是根元类的超类是`NSObject`，而`isa`指向了自己，而`NSObject`的超类为`nil`，也就是它没有超类。

Method

`Method`是一种代表类中的某个方法的类型。

```
1  typedef struct objc_method *Method;
```

而objc_method在上面的方法列表中提到过，它存储了方法名，方法类型和方法实现：

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

- 方法名类型为SEL，前面提到过相同名字的方法即使在不同类中定义，它们的方法选择器也相同。
- 方法类型method_types是个char指针，其实存储着方法的参数类型和返回值类型。
- method_imp指向了方法的实现，本质上是一个函数指针，后面会详细讲到。

Ivar

Ivar是一种代表类中实例变量的类型。

```
typedef struct objc_ivar *Ivar;
```

而objc_ivar在上面的成员变量列表中也提到过：

```
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
}                                                            OBJC2_UNAVAILABLE;
```

可以根据实例查找其在类中的名字，也就是“反射”：

```
-(NSString *)nameWithInstance:(id)instance {
    unsigned int numIvars = 0;
    NSString *key=nil;
    Ivar * ivars = class_copyIvarList([self class], &numIvars);
```

```
    for(int i = 0; i < numIvars; i++) {
        Ivar thisIvar = ivars[i];
        const char *type = ivar_getTypeEncoding(thisIvar);
        NSString *stringType =  [NSString stringWithCString:type encoding:NSUTF8StringEncodi
        if (![stringType hasPrefix:@"@"]) {
            continue;
        }
        if ((object_getIvar(self, thisIvar) == instance)) {
            key = [NSString stringWithUTF8String:ivar_getName(thisIvar)];
            break;
        }
    }
    free(ivars);
    return key;
}
```

class_copyIvarList 函数获取的不仅有实例变量，还有属性。但会在原本的属性名前加上一个下划线。

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的。而 IMP 这个函数指针就指向了这个方法的实现。既然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类型的方法选择器，而每个实例对象中的SEL对应的方法实现肯定是唯一的，通过一组id和SEL参数就能确定唯一的方法实现地址；反之亦然。

Cache

在runtime.h中Cache的定义如下：

```
typedef struct objc_cache *Cache
```

还记得之前objc_class结构体中有一个struct objc_cache *cache吧，它到底是缓存啥的呢，先看看objc_cache的实现：

```
struct objc_cache {
    unsigned int mask                 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，通俗地讲，每当实例对象接收到一个消息时，它不会直接在isa指向的类的方法列表中遍历查找能够响应消息的方法，因为这样效率太低了，而是优先在Cache中查找。Runtime 系统会把被调用的方法存到Cache中（理论上讲一个方法如果被调用，那么它有可能今后还会被调用），下次查找的时候效率更高。这根计算机组成原理中学过的 CPU 绕过主存先访问Cache的道理挺像，而我猜苹果为提高Cache命中率应该也做了努力吧。

Property

@property标记了类中的属性，这个不必多说大家都很熟悉，它是一个指向objc_property结构体的指针：

```
typedef struct objc_property *Property;
typedef struct objc_property *objc_property_t;
```

可以通过class_copyPropertyList 和 protocol_copyPropertyList方法来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)
```

返回类型为指向指针的指针，哈哈，因为属性列表是个数组，每个元素内容都是一个`objc_property_t`指针，而这两个函数返回的值是指向这个数组的指针。

举个栗子，先声明一个类：

```
@interface Lender : NSObject {
    float alone;
}
@property float alone;
@end
```

你可以用下面的代码获取属性列表：

```
id LenderClass = objc_getClass("Lender")
unsigned int outCount
objc_property_t *properties = class_copyPropertyList(LenderClass, &outCount)
```

你可以用`property_getName`函数来查找属性名称：

```
const char *property_getName(objc_property_t property)
```

你可以用`class_getProperty` 和 `protocol_getProperty`通过给出的名称来在类和协议中获取属性的引用：

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

你可以用`property_getAttributes`函数来发掘属性的名称和`@encode`类型字符串：

```
const char *property_getAttributes(objc_property_t property)
```

把上面的代码放一起，你就能从一个类中获取它的属性啦：

```
```

```
id LenderClass = objc_getClass("Lender");
unsigned int outCount, i;
objc_property_t *properties = class_copyPropertyList(LenderClass, &outCount);
for (i = 0; i < outCount; i++) {
    objc_property_t property = properties[i];
    fprintf(stdout, "%s %s\n", property_getName(property), property_getAttributes(property));
}
```

对比下 `class_copyIvarList` 函数，使用 `class_copyPropertyList` 函数只能获取类的属性，而不包含成员变量。但此时获取的属性名是不带下划线的。

消息

前面做了这么多铺垫，现在终于说到了消息了。Objc 中发送消息是用中括号（[]）把接收者和消息括起来，而直到运行时才会把消息与方法实现绑定。

有关消息发送和消息转发机制的原理，可以查看[这篇文章](#)。

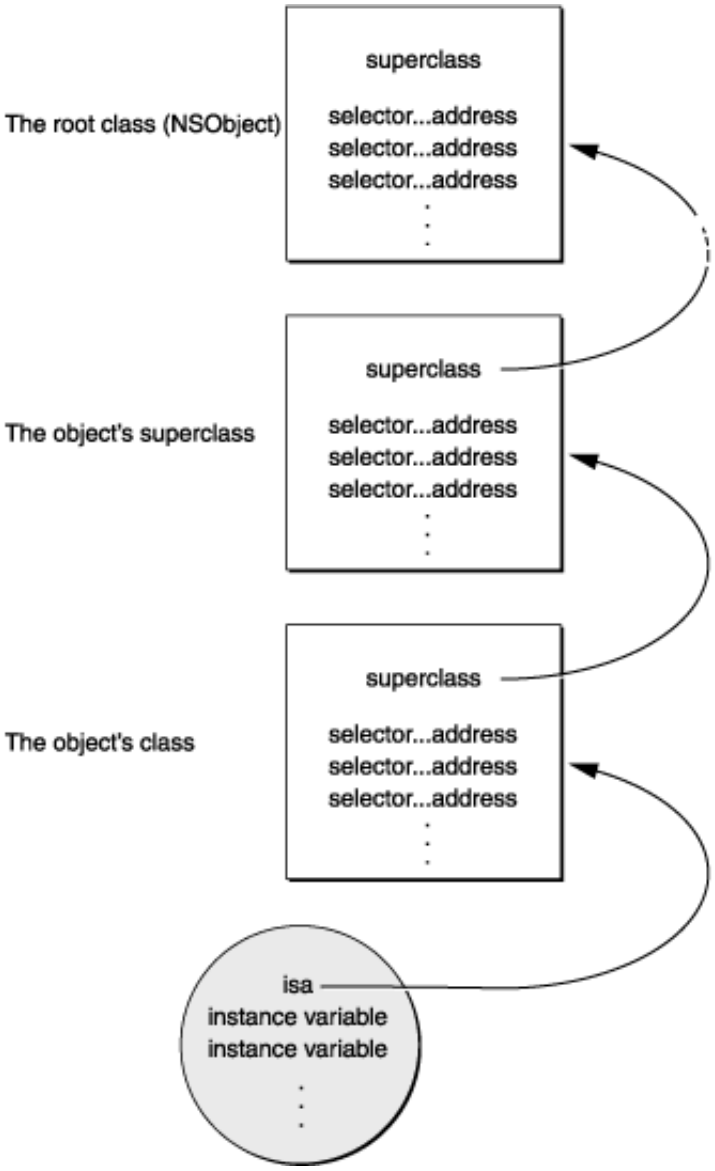
objc_msgSend函数

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的方法被调用后返回了数据。下面详细叙述下消息发送步骤：

1. 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。
2. 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。
3. 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。
4. 如果 `cache` 找不到就找一下方法分发表。
5. 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。

6. 如果还找不到就要开始进入**动态方法**解析了，后面会提到。

PS:这里说的分发表其实就是`Class`中的方法列表，它将方法选择器和方法实现地址联系起来。



其实编译器会根据情况在`objc_msgSend`, `objc_msgSend_stret`, `objc_msgSendSuper`, 或 `objc_msgSendSuper_stret`四个方法中选择一个来调用。如果消息是传递给超类，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，那么会调用名字带有”stret”的函数。排列组合正好四个方法。

值得一提的是在 i386 平台处理返回类型为浮点数的消息时，需要用

到objc_msgSend_fpret函数来进行处理，这是因为返回类型为浮点数的函数对应的 ABI(Application Binary Interface) 与返回整型的函数的 ABI 不兼容。此时objc_msgSend不再适用，于是objc_msgSend_fpret被派上用场，它会对浮点数寄存器做特殊处理。不过在 PPC 或 PPC64 平台是不需要麻烦它的。

PS：有木有发现这些函数的命名规律哦？带“Super”的是消息传递给超类；“stret”可分为“st”+“ret”两部分，分别代表“struct”和“return”；“fpret”就是“fp”+“ret”，分别代表“floating-point”和“return”。

方法中的隐藏参数

我们经常在方法中使用self关键字来引用实例本身，但从没有想过为什么self就能取到调用当前方法的对象吧。其实self的内容是在方法运行时被偷偷的动态传入的。

当objc_msgSend找到方法对应的实现时，它将直接调用该方法实现，并将消息中所有的参数都传递给方法实现,同时,它还将传递两个隐藏的参数:

- 接收消息的对象（也就是self指向的内容）
- 方法选择器（_cmd指向的内容）

之所以说它们是隐藏的是因为在源代码方法的定义中并没有声明这两个参数。它们是在代码被编译时被插入实现中的。尽管这些参数没有被明确声明，在源代码中我们仍然可以引用它们。在下面的例子中，self引用了接收者对象，而_cmd引用了方法本身的选择器：

```
- strange
{
    id  target = getTheReceiver();
    SEL method = getTheMethod();

    if ( target == self || method == _cmd )
```

```
7           return nil;
8       return [target performSelector:method];
9   }
```

在这两个参数中，self 更有用。实际上,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。

而当方法中的super关键字接收到消息时，编译器会创建一个objc_super结构体：

```
1   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超类的定义。但receiver仍然是self本身，这点需要注意，因为当我们想通过[super class]获取超类时，编译器只是将指向self的id指针和class的SEL传递给了objc_msgSendSuper函数，因为只有在NSObject类才能找到class方法，然后class方法调用object_getClass()，接着调用objc_msgSend(objc_super->receiver, @selector(class))，传入的第一个参数是指向self的id指针，与调用[self class]相同，所以我们得到的永远都是self的类型。

获取方法地址

在IMP那节提到过可以避开消息绑定而直接获取方法的地址并调用方法。这种做法很少用，除非是需要持续大量重复调用某方法的极端情况，避开消息发送泛滥而直接调用该方法会更高效。

NSObject类中有个methodForSelector:实例方法，你可以用它来获取某个方法选择器对应的IMP，举个栗子：

```
1   void
2   int i;
3
4   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，上节提到的两个隐藏参数就需要我们明确给出了。上面的例子调用了1000次函数，你可以试试直接给target发送1000次setFilled:消息会花多久。

PS：methodForSelector:方法是由 Cocoa 的 Runtime 系统提供的，而不是 Objc 自身的特性。

动态方法解析

你可以动态地提供一个方法的实现。例如我们可以用@dynamic关键字在类的实现文件中修饰一个属性：

这表明我们会为这个属性动态提供存取方法，也就是说编译器不会再默认为我们生成setPropertyName:和propertyName方法，而需要我们动态提供。我们可以通过分别重载resolveInstanceMethod:和resolveClassMethod:方法分别添加实例方法实现和类方法实现。因为当 Runtime 系统在Cache和方法分发表中（包括超类）找不到要执行的方法时，Runtime会调用resolveInstanceMethod:或resolveClassMethod:来给程序员一次动态添加方法实现的机会。我们需要用class_addMethod函数完成向特定类添加特定方法实现的操作：

```
void dynamicMethodIMP(id self, SEL _cmd) {

}
@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，也就是`dynamicMethodIMP`方法中的代码。其中“`v@:`”表示返回值和参数，这个符号涉及 Type Encoding

PS：动态方法解析会在消息转发机制浸入前执行。如果`respondsToSelector:` 或 `instancesRespondToSelector:`方法被执行，动态方法解析器将会被首先给予一个提供该方法选择器对应的`IMP`的机会。如果你想让该方法选择器被传送到转发机制，那么就让`resolveInstanceMethod:`返回`NO`。

评论区有人问如何用 `resolveClassMethod:` 解析类方法，我将他贴出有问题的代码做了纠正和优化后如下，可以顺便将实例方法和类方法的动态方法解析对比下：
头文件：

```
#import <Foundation/Foundation.h>

@interface Student : NSObject
+ (void)learnClass:(NSString *) string;
- (void)goToSchool:(NSString *) name;
@end
```

m 文件：

```
#import "Student.h"
#import <objc/runtime.h>

@implementation Student
+ (BOOL)resolveClassMethod:(SEL)sel {
    if (sel == @selector(learnClass:)) {
        class_addMethod(object_getClass(self), sel, class_getMethodImplementation(object_get
        return YES;
    }
    return [class_getSuperclass(self) resolveClassMethod:sel];
}

+ (BOOL)resolveInstanceMethod:(SEL)aSEL
{
    if (aSEL == @selector(goToSchool:)) {
        class_addMethod([self class], aSEL, class_getMethodImplementation([self class], @sel
        return YES;
```

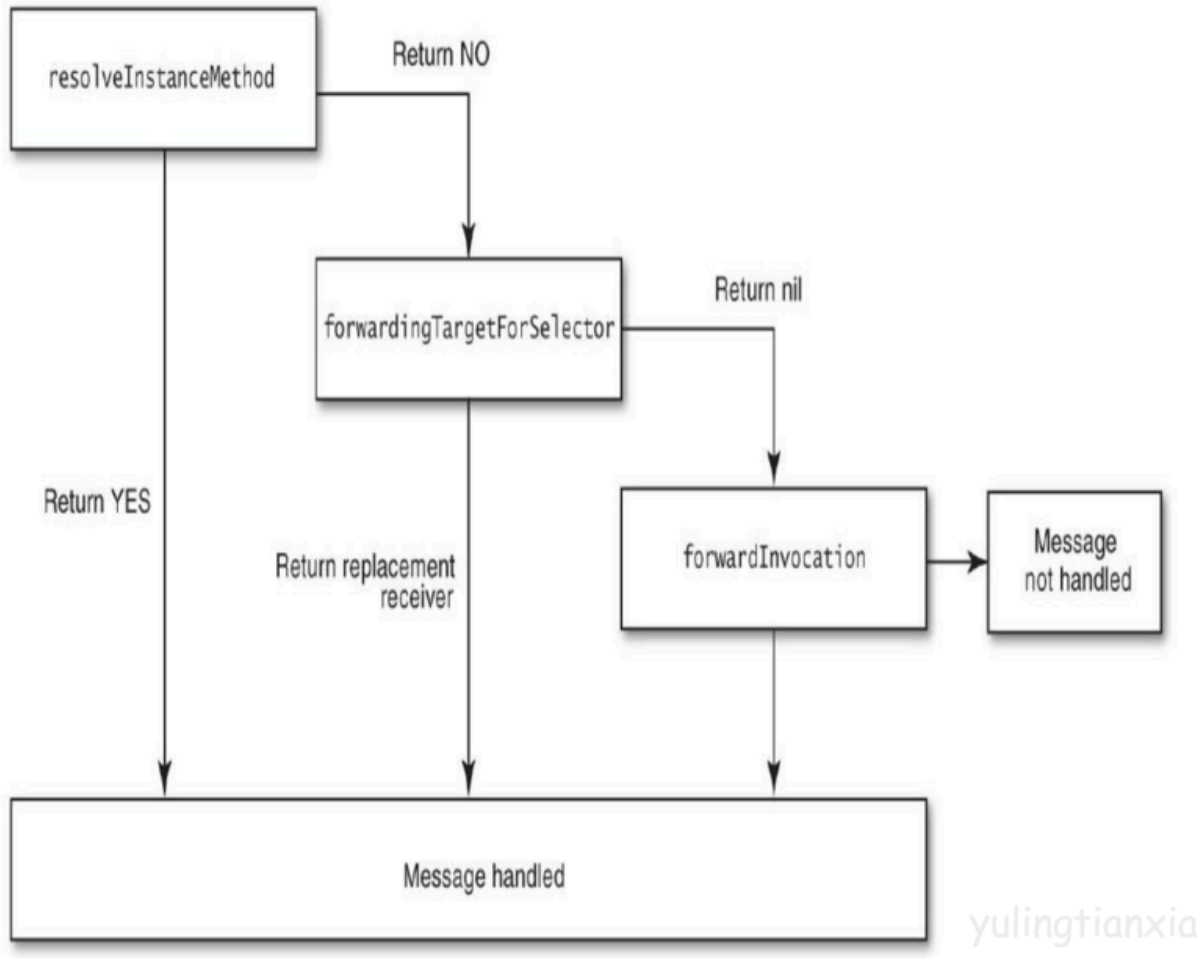
```
    }
    return [super resolveInstanceMethod:aSEL];
}

+ (void)myClassMethod:(NSString *)string {
    NSLog(@"myClassMethod = %@", string);
}

- (void)myInstanceMethod:(NSString *)string {
    NSLog(@"myInstanceMethod = %@", string);
}
@end
```

需要深刻理解 `[self class]` 与 `object_getClass(self)` 甚至 `object_getClass([self class])` 的关系，其实并不难，重点在于 `self` 的类型：

1. 当 `self` 为实例对象时，`[self class]` 与 `object_getClass(self)` 等价，因为前者会调用后者。`object_getClass([self class])` 得到元类。
2. 当 `self` 为类对象时，`[self class]` 返回值为自身，还是 `self`。`object_getClass(self)` 与 `object_getClass([self class])` 等价。

凡是涉及到类方法时，一定要弄清楚元类、selector、IMP 等概念，这样才能做到举一反三，随机应变。

消息转发



重定向

在消息转发机制执行前，Runtime 系统会再给我们一次偷梁换柱的机会，即通过重载`- (id)forwardingTargetForSelector:(SEL)aSelector`方法替换消息的接受者为其他对象：

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

毕竟消息转发要耗费更多时间，抓住这次机会将消息重定向给别人是个不错的选择，~~不过千万别返回`self`，因为那样会死循环。~~ 如果此方法返回nil或self,则会进入消息转发机制(`forwardInvocation:`);否则

将向返回的对象重新发送消息。

如果想替换**类方法**的接受者，需要覆写 + (id)forwardingTargetForSelector:(SEL)aSelector 方法，并返回**类对象**：

```
+ (id)forwardingTargetForSelector:(SEL)aSelector {
        if(aSelector == @selector(xxx)) {
                return NSClassFromString(@"Class name");
        }
        return [super forwardingTargetForSelector:aSelector];
}
```

转发

当动态方法解析不作处理返回NO时，消息转发机制会被触发。在这时forwardInvocation:方法会被执行，我们可以重写这个方法来定义我们的转发逻辑：

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

该消息的唯一参数是个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:方法来对不能处理的消息做一些默认的处理，也可以将消息转发给其他对象来处理，而不抛出错误。

这里需要注意的是参数anInvocation是从哪的来的呢？其实在forwardInvocation:消息发送前，Runtime系统会向对象发送methodSignatureForSelector:消息，并取到返回的方法签名用于生成NSInvocation对象。所以我们在重写forwardInvocation:的同时

也要重写methodSignatureForSelector:方法，否则会抛异常。

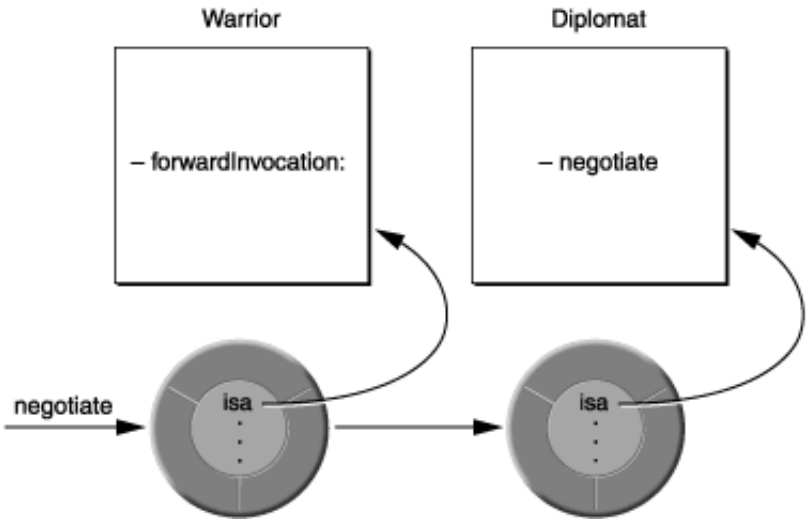
当一个对象由于没有相应的方法实现而无法响应某消息时，运行时系统将通过forwardInvocation:消息通知该对象。每个对象都从NSObject类中继承了forwardInvocation:方法。然而，NSObject中的方法实现只是简单地调用了doesNotRecognizeSelector:。通过实现我们自己的forwardInvocation:方法，我们可以在该方法实现中将消息转发给其它对象。

forwardInvocation:方法就像一个不能识别的消息的分发中心，将这些消息转发给不同接收对象。或者它也可以象一个运输站将所有的消息都发送给同一个接收对象。它可以将一个消息翻译成另外一个消息，或者简单的”吃掉“某些消息，因此没有响应也没有错误。forwardInvocation:方法也可以对不同的消息提供同样的响应，这一切都取决于方法的具体实现。该方法所提供是将不同的对象链接到消息链的能力。

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，如果我们希望一个对象将negotiate消息转发给其它对象，则这个对象不能有negotiate方法。否则，forwardInvocation:将不可能会被调用。

转发和多继承

转发和继承相似，可以用于为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，就好似它把另一个对象中的方法借过来或是“继承”过来一样。



这使得不同继承体系分支下的两个类可以“继承”对方的方法，在上图中Warrior和Diplomat没有继承关系，但是Warrior将negotiate消息转发给了Diplomat后，就好似Diplomat是Warrior的超类一样。

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。它将问题分解得很细，只针对想要借鉴的方法才转发，而且转发机制是透明的。

替代者对象(Surrogate Objects)

转发不仅能模拟多继承，也能使轻量级对象代表重量级对象。弱小的女人背后是强大的男人，毕竟女人遇到难题都把它们转发给男人来做了。这里有一些适用案例，可以参看官方文档。

转发与继承

尽管转发很像继承，但是NSObject类不会将两者混淆。像 respondsToSelector: 和 isKindOfClass:这类方法只会考虑继承体系，不会考虑转发链。比如上图中一个Warrior对象如果被问到是否能响应negotiate消息：

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

结果是NO，尽管它能够接受negotiate消息而不报错，因为它靠转发消息给Diplomat类来响应消息。

如果你为了某些意图偏要“弄虚作假”让别人以为Warrior继承到了Diplomat的negotiate方法，你得重新实现 respondsToSelector: 和 isKindOfClass:来加入你的转发算法：

```
- (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。如果使用了协议，conformsToProtocol:同样也要加入到这一行列中。类似地，如果一个对象转发它接受的任何远程消息，它得给出一个methodSignatureForSelector:来返回准确的方法描述，这个方法会最终响应被转发的消息。比如一个对象能给它的替代者对象转发消息，它需要像下面这样实现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。当一

个类被编译时，实例变量的布局也就形成了，它表明访问类的实例变量的位置。从对象头部开始，实例变量依次根据自己所占空间而产生位移：

NSObject	
0	Class isa

MyObject : NSObject	
0	Class isa
4	NSArray students
8	NSArray teachers

上图左边是`NSObject`类的实例变量布局，右边是我们写的类的布局，也就是在超类后面加上我们自己类的实例变量，看起来不错。但试想如果哪天苹果更新了`NSObject`类，发布新版本的系统的话，那就悲剧了：

NSObject	
0	Class isa
4	NSArray secretAry
8	NSImage secretImg

MyObject : NSObject	
0	Class isa
4	~~NSArray students~~
8	~~NSArray teachers~~

我们自定义的类被划了两道线，那是因为那块区域跟超类重叠了。唯有苹果将超类改为以前的布局才能拯救我们，但这样也导致它们不能再拓展它们的框架了，因为成员变量布局被死死地固定了。在脆弱的实例变量(Fragile ivars) 环境下我们需要重新编译继承自 Apple 的类来恢复兼容性。那么在健壮的实例变量下会发生什么呢?

NSObject	
0	Class isa
4	NSArray secretAry
8	NSImage secretImg

MyObject : NSObject	
0	Class isa
4	NSArray secretAry
8	NSImage secretImg
12	NSArray students
16	NSArray teachers

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 系统检测到与超类有部分重叠时它会调整你新添加的实例变量的位移，那样你在子类中新添加的成员就被保护起来了。

需要注意的是在健壮的实例变量下，不要使用`sizeof(SomeClass)`，而是用`class_getInstanceSize([SomeClass class])`代替；也不要使用`offsetof(SomeClass, SomeIvar)`，而要用`ivar_getOffset(class_getInstanceVariable([SomeClass class], "SomeIvar"))`来代替。

Objective-C Associated Objects

在 OS X 10.6 之后，Runtime系统让Objc支持向对象动态添加变量。涉及到的函数有以下三个：

```
void objc_setAssociatedObject ( id object, const void *key, id value, objc_AssociationPolicy
id objc_getAssociatedObject ( id object, const void *key );
void objc_removeAssociatedObjects ( id object );
```

这些方法以键值对的形式动态地向对象添加、获取或删除关联值。其中关联政策是一组枚举常量：

```
enum {
   OBJC_ASSOCIATION_ASSIGN  = 0,
   OBJC_ASSOCIATION_RETAIN_NONATOMIC  = 1,
   OBJC_ASSOCIATION_COPY_NONATOMIC  = 3,
   OBJC_ASSOCIATION_RETAIN  = 01401,
   OBJC_ASSOCIATION_COPY  = 01403
};
```

这些常量对应着引用关联值的政策，也就是 Objc 内存管理的引用计数机制。**有关 Objective-C 引用计数机制的原理，可以查看[这篇文章]。**

Method Swizzling

之前所说的消息转发虽然功能强大，但需要我们了解并且能更改对应类的源代码，因为我们需要实现自己的转发逻辑。当我们无法触碰到某个类的源代码，却想更改这个类某个方法的实现时，该怎么办呢？可能继承类并重写方法是一种想法，但是有时无法达到目的。这里介绍的是 Method Swizzling ，它通过重新映射方法对应的实现来达到“偷天换日”的目的。跟消息转发相比，Method Swizzling 的做法更为隐蔽，甚至有些冒险，也增大了debug的难度。

这里摘抄一个 NSHipster 的例子：

```
#import <objc/runtime.h>

@implementation UIViewController (Tracking)

+ (void)load {
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        Class aClass = [self class];

        SEL originalSelector = @selector(viewWillAppear:);
        SEL swizzledSelector = @selector(xxx_viewWillAppear:);

        Method originalMethod = class_getInstanceMethod(aClass, originalSelector);
        Method swizzledMethod = class_getInstanceMethod(aClass, swizzledSelector);









        BOOL didAddMethod =
            class_addMethod(aClass,
                originalSelector,
                method_getImplementation(swizzledMethod),
                method_getTypeEncoding(swizzledMethod));

        if (didAddMethod) {
            class_replaceMethod(aClass,
                swizzledSelector,
                method_getImplementation(originalMethod),
                method_getTypeEncoding(originalMethod));
        } else {
            method_exchangeImplementations(originalMethod, swizzledMethod);
        }
    });
}

#pragma mark - Method Swizzling

- (void)xxx_viewWillAppear:(BOOL)animated {
    [self xxx_viewWillAppear:animated];
    NSLog(@"viewWillAppear: %@", self);
```

```
}

@end
```

上面的代码通过添加一个Tracking类别到UIViewController类中，将UIViewController类的viewWillAppear:方法和Tracking类别中xxx_viewWillAppear:方法的实现相互调换。Swizzling 应该在+load方法中实现，因为+load是在一个类最开始加载时调用。dispatch_once是GCD中的一个方法，它保证了代码块只执行一次，并让其为一个原子操作，线程安全是很重要的。

如果类中不存在要替换的方法，那就先用class_addMethod和class_replaceMethod函数添加和替换两个方法的实现；如果类中已经有了想要替换的方法，那么就调用method_exchangeImplementations函数交换了两个方法的 IMP，这是苹果提供给我们用于实现 Method Swizzling 的便捷方法。

可能有人注意到了这行:

```
// When swizzling a class method, use the following:
// Class aClass = object_getClass((id)self);
// ...
// Method originalMethod = class_getClassMethod(aClass, originalSelector);
// Method swizzledMethod = class_getClassMethod(aClass, swizzledSelector);
```

object_getClass((id)self) 与 [self class] 返回的结果类型都是 Class,但前者为元类,后者为其本身,因为此时 self 为 Class 而不是实例.注意 [NSObject class] 与 [object class] 的区别:

```
+ (Class)class {
    return self;
}

- (Class)class {
    return object_getClass(self);
}
```

PS:如果类中没有想被替换实现的原方法时，class_replaceMethod相当于直接调用class_addMethod向类中添加该方法的实现；否则调用method_setImplementation方法，types参数会被忽略。method_exchangeImplementations方法做的事情与如下的原子操作等价：

```
IMP imp1 = method_getImplementation(m1)
IMP imp2 = method_getImplementation(m2)
method_setImplementation(m1, imp2)
method_setImplementation(m2, imp1)
```

最后xxx_viewWillAppear:方法的定义看似是递归调用引发死循环，其实不会的。因为[self xxx_viewWillAppear:animated]消息会动态找到xxx_viewWillAppear:方法的实现，而它的实现已经被我们与viewWillAppear:方法实现进行了互换，所以这段代码不仅不会死循环，如果你把[self xxx_viewWillAppear:animated]换成[self viewWillAppear:animated]反而会引发死循环。

看到有人说+load方法本身就是线程安全的，因为它在程序刚开始就被调用，很少会碰到并发问题，于是 stackoverflow 上也有大神给出了另一个 Method Swizzling 的实现：

```
- (void)replacementReceiveMessage:(const struct BInstantMessage *)arg1 {
    NSLog(@"arg1 is %@", arg1)
    [self replacementReceiveMessage:arg1]
}
+ (void)load {
    SEL originalSelector = @selector(ReceiveMessage:)
    SEL overrideSelector = @selector(replacementReceiveMessage:)
    Method originalMethod = class_getInstanceMethod(self, originalSelector);
    Method overrideMethod = class_getInstanceMethod(self, overrideSelector)
    if (class_addMethod(self, originalSelector, method_getImplementation(overrideMethod), me
            class_replaceMethod(self, overrideSelector, method_getImplementation(originalMet
    } else {
            method_exchangeImplementations(originalMethod, overrideMethod)
    }
}
```

上面的代码同样要添加在某个类的类别中，相比第一个种实现，只是去掉了`dispatch_once`部分。

Method Swizzling 的确是一个值得深入研究的话题，Method Swizzling 的最佳实现是什么呢？小弟才疏学浅理解的不深刻，找了几篇不错的资源推荐给大家：

- Objective-C的hook方案（一）：Method Swizzling
- Method Swizzling
- How do I implement method swizzling?
- What are the Dangers of Method Swizzling in Objective C?
- JRSwizzle

在用 SpriteKit 写游戏的时候,因为 API 本身有一些缺陷(增删节点时不考虑父节点是否存在啊,很容易崩溃啊有木有!),我在 Swift 上使用 Method Swizzling弥补这个缺陷:

```
extension SKNode {

    class func yxy_swizzleAddChild() {
        let cls = SKNode.self
        let originalSelector = Selector("addChild:")
        let swizzledSelector = Selector("yxy_addChild:")
        let originalMethod = class_getInstanceMethod(cls, originalSelector)
        let swizzledMethod = class_getInstanceMethod(cls, swizzledSelector)
        method_exchangeImplementations(originalMethod, swizzledMethod)
    }

    class func yxy_swizzleRemoveFromParent() {
        let cls = SKNode.self
        let originalSelector = Selector("removeFromParent")
        let swizzledSelector = Selector("yxy_removeFromParent")
        let originalMethod = class_getInstanceMethod(cls, originalSelector)
        let swizzledMethod = class_getInstanceMethod(cls, swizzledSelector)
        method_exchangeImplementations(originalMethod, swizzledMethod)
    }

    func yxy_addChild(node: SKNode) {
        if node.parent == nil {
            self.yxy_addChild(node)
        }
        else {
            println("This node has already a parent!\(node.name)")
        }
    }

    func yxy_removeFromParent() {
        if parent != nil {
```

```
            dispatch_async(dispatch_get_main_queue(), { () -> Void in
                self.yxy_removeFromParent()
            })
        }
        else {
            println("This node has no parent!\(name)")
        }
    }

}
```

然后其他地方调用那两个类方法:

```
SKNode.yxy_swizzleAddChild()
SKNode.yxy_swizzleRemoveFromParent()
```

因为 Swift 中的 extension 的特殊性,最好在某个类的`load()` 方法中调用上面的两个方法.我是在AppDelegate 中调用的,于是保证了应用启动时能够执行上面两个方法.

总结

我们之所以让自己的类继承`NSObject`不仅仅因为苹果帮我们完成了复杂的内存分配问题，更是因为这使得我们能够用上 Runtime 系统带来的便利。可能我们平时写代码时可能很少会考虑一句简单的`[receiver message]`背后发生了什么，而只是当做方法或函数调用。深入理解 Runtime 系统的细节更有利于我们利用消息机制写出功能更强大的代码，比如 Method Swizzling 等。

参考链接:

- Objective-C Runtime Programming Guide
- Objective-C runtime之运行时的基本特点
- Understanding the Objective-C Runtime