Objective-C Runtime

顾 鹏 04 January 2015

Objective-C 扩展了 C 语言，并加入了面向对象特性和 Smalltalk 式的消息传递机制。而这个扩展的核心是一个用 C 和 编译语言 写的 Runtime 库。它是 Objective-C 面向对象和动态机制的基石。

Objective-C 是一个动态语言，这意味着它不仅需要一个编译器，也需要一个运行时系统来动态得创建类和对象、进行消息传递和转发。理解 Objective-C 的 Runtime 机制可以帮我们更好的了解这个语言，适当的时候还能对语言进行扩展，从系统层面解决项目中的一些设计或技术问题。了解 Runtime ，要先了解它的核心 - 消息传递 （Messaging）。

消息传递（Messaging）

> *I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.* ***The big idea is "messaging"*** *– that is what the kernal[sic] of Smalltalk is all about... The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.*

Alan Kay 曾多次强调 Smalltalk 的核心不是面向对象，面向对象只是 **the lesser ideas**，消息传递才是 **the big idea**。

在很多语言，比如 C ，调用一个方法其实就是跳到内存中的某一点并开始执行一段代码。没有任何动态的特性，因为这在编译时就决定好了。而在 Objective-C 中，`[object foo]` 语法并不会立即执行 foo 这个方法的代码。它是在运行时给 object 发送一条叫 foo 的消息。这个消息，也许会由 object 来处理，也许会被转发给另一个对象，或者不予理睬假装没收到这个消息。多条不同的消息也可以对应同一个方法实现。这些都是在程序运行的时候决定的。

事实上，在编译时你写的 Objective-C 函数调用的语法都会被翻译成一个 C 的函数调用 - `objc_msgSend()` 。比如，下面两行代码就是等价的：

```
[array insertObject:foo atIndex:5];

objc_msgSend(array, @selector(insertObject:atIndex:), foo, 5);
```

消息传递的关键藏于 objc_object 中的 isa 指针和 objc_class 中的 class dispatch table。

objc_object, objc_class 以及 Ojbc_method

在 Objective-C 中，类、对象和方法都是一个 C 的结构体，从 objc/objc.h 头文件中，我们可以找到他们的定义：

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

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

struct objc_method_list {
    struct objc_method_list *obsolete;
    int method_count;

#ifdef __LP64__
    int space;
#endif

    /* variable length structure */
    struct objc_method method_list[1];
};

struct objc_method {
```

```
    SEL method_name;
    char *method_types;    /* a string representing argument/return types *
    IMP method_imp;
};
```

`objc_method_list` 本质是一个有 `objc_method` 元素的可变长度的数组。一个 `objc_method` 结构体中有函数名，也就是SEL，有表示函数类型的字符串(见 Type Encoding)，以及函数的实现IMP。

从这些定义中可以看出发送一条消息也就 `objc_msgSend` 做了什么事。举 `objc_msgSend(obj, foo)` 这个例子来说：

1. 首先，通过 obj 的 isa 指针找到它的 class ;
2. 在 class 的 method list 找 foo ;
3. 如果 class 中没到 foo，继续往它的 superclass 中找 ;
4. 一旦找到 foo 这个函数，就去执行它的实现IMP .

但这种实现有个问题，效率低。但一个 class 往往只有 20% 的函数会被经常调用，可能占总调用次数的 80% 。每个消息都需要遍历一次 `objc_method_list` 并不合理。如果把经常被调用的函数缓存下来，那可以大大提高函数查询的效率。这也就是 `objc_class` 中另一个重要成员 `objc_cache` 做的事情 - 再找到 foo 之后，把 foo 的 `method_name` 作为 key ，`method_imp` 作为 value 给存起来。当再次收到 foo 消息的时候，可以直接在 cache 里找到，避免去遍历 `objc_method_list`.

动态方法解析和转发

在上面的例子中，如果 `foo` 没有找到会发生什么？通常情况下，程序会在运行时挂掉并抛出 *unrecognized selector sent to …* 的异常。但在异常抛出前，Objective-C 的运行时会给你三次拯救程序的机会：

1. Method resolution
2. Fast forwarding
3. Normal forwarding

Method Resolution

首先，Objective-C 运行时会调用 +resolveInstanceMethod: 或者 +resolveClassMethod:，让你有机会提供一个函数实现。如果你添加了函数并返回 YES， 那运行时系统就会重新启动一次消息发送的过程。还是以 foo 为例，你可以这么实现：

```
void fooMethod(id obj, SEL _cmd)
{
    NSLog(@"Doing foo");
}

+ (BOOL)resolveInstanceMethod:(SEL)aSEL
{
    if(aSEL == @selector(foo:)){
        class_addMethod([self class], aSEL, (IMP)fooMethod, "v@:");
        return YES;
    }
    return [super resolveInstanceMethod];
}
```

Core Data 有用到这个方法。NSManagedObjects 中 properties 的 getter 和 setter 就是在运行时动态添加的。

如果 resolve 方法返回 NO ，运行时就会移到下一步：**消息转发（Message Forwarding）** 。

PS：iOS 4.3 加入很多新的 runtime 方法，主要都是以 imp 为前缀的方法，比如 imp_implementationWithBlock() 用 block 快速创建一个 imp 。
上面的例子可以重写成：

```
IMP fooIMP = imp_implementationWithBlock(^(id _self) {
    NSLog(@"Doing foo");
});

class_addMethod([self class], aSEL, fooIMP, "v@:");
```

Fast forwarding

如果目标对象实现了 -forwardingTargetForSelector:，Runtime 这时就会调用这个方法，给你把这个消息转发给其他对象的机会。

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

只要这个方法返回的不是 nil 和 self，整个消息发送的过程就会被重启，当然发送的对象会变成你返回的那个对象。否则，就会继续 Normal Fowarding 。

这里叫 Fast ，只是为了区别下一步的转发机制。因为这一步不会创建任何新的对象，但下一步转发会创建一个 NSInvocation 对象，所以相对更快点。

Normal forwarding

这一步是 Runtime 最后一次给你挽救的机会。首先它会发送 `-methodSignatureForSelector:` 消息获得函数的参数和返回值类型。如果 `-methodSignatureForSelector:` 返回 nil ，Runtime 则会发出 `-doesNotRecognizeSelector:` 消息，程序这时也就挂掉了。如果返回了一个函数签名，Runtime 就会创建一个 NSInvocation 对象并发送 `-forwardInvocation:` 消息给目标对象。

NSInvocation 实际上就是对一个消息的描述，包括selector 以及参数等信息。所以你可以在 `-forwardInvocation:` 里修改传进来的 NSInvocation 对象，然后发送 `-invokeWithTarget:` 消息给它，传进去一个新的目标：

```
- (void)forwardInvocation:(NSInvocation *)invocation
{
    SEL sel = invocation.selector;

    if([alternateObject respondsToSelector:sel]) {
        [invocation invokeWithTarget:alternateObject];
    }
    else {
        [self doesNotRecognizeSelector:sel];
    }
}
```

Cocoa 里很多地方都利用到了消息传递机制来对语言进行扩展，如 Proxies、NSUndoManager 跟 Responder Chain。NSProxy 就是专门用来作为代理转发消息的；NSUndoManager 截取一个消息之后再发送；而 Responder Chain 保证一个消息转发给合适的响应者。

总结

Objective-C 中给一个对象发送消息会经过以下几个步骤：

1. 在对象类的 *dispatch table* 中尝试找到该消息。如果找到了，跳到相应的函数IMP去执行实现代码；
2. 如果没有找到，Runtime 会发送 `+resolveInstanceMethod:` 或者 `+resolveClassMethod:` 尝试去 resolve 这个消息；
3. 如果 resolve 方法返回 NO，Runtime 就发送 `-forwardingTargetForSelector:` 允许你把这个消息转发给另一个对象；
4. 如果没有新的目标对象返回， Runtime 就会发送 `-methodSignatureForSelector:` 和 `-forwardInvocation:` 消息。你可以发送 `-invokeWithTarget:` 消息来手动转发消息或者发送 `-doesNotRecognizeSelector:` 抛出异常。

利用 Objective-C 的 runtime 特性，我们可以自己来对语言进行扩展，解决项目开发中的一些设计和技术问题。下一篇文章，我会介绍 Method Swizzling 技术以及如何利用 Method Swizzling 做 Logging。

Reference

Message forwarding

Objective-c-messaging

The faster objc_msgSend

Understanding objective-c runtime

Live with less, share with more. weibo: @no-computer

代码规范和Android项目中的一些可用工具

这里主要讲一下关于代码规范的相关问题，和在Android项目中如何利用一些工具进行规范和检查。代码规范不是一个Android项目特有的问题，所以前部分内容是不单针对Android的。 什么是代码规范? 代码规范一般是指在编程过程中的一系列规则规范。 一般来说代码规范可以分为两种。 一是编程语言本身在设计时所规定的一些原则，这类规则大部分都是强制的，像Python里用缩进表示逻辑块，Go里用首字母大小写表示可见度。 另外一种是在一些组织约定的一些规范模式或个人在编写代码时的一些偏好，这种一般都是非强制的。比如大括号是放在方法名的同一行呢还是另起一行，不同的人有不同的想法，我也不知道谁好，所以别问我。 假如是强制的，大家暂时也不能反抗，…