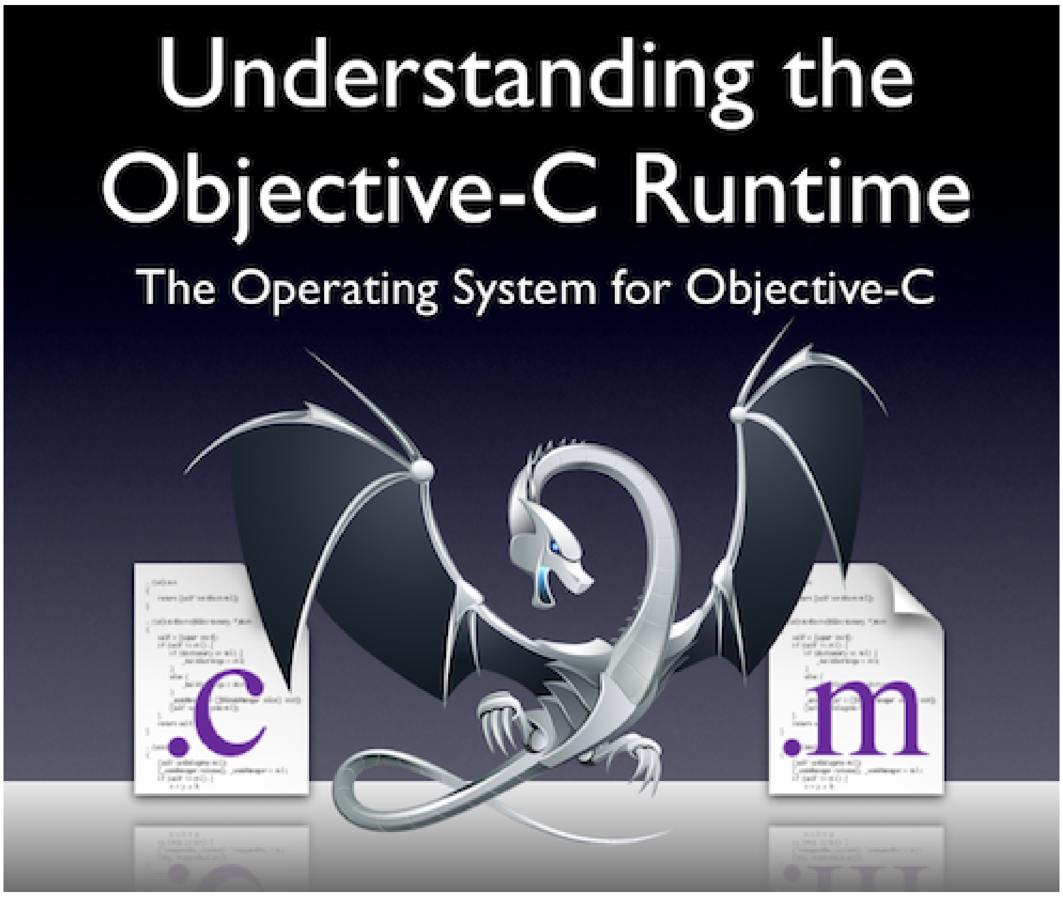
COCOA SAMURAI

Wednesday, January 20, 2010

Understanding the Objective-C Runtime



The Objective-C Runtime is one of the overlooked features of Objective-C initially when people are generally introduced to Cocoa/Objective-C. The reason for this is that while Objective-C (the language) is easy to pick up in only a couple hours, newcomers to Cocoa spend most of their time wrapping their heads around the Cocoa Framework and adjusting to how it works. However the runtime is something that everybody should at least know how it works in some detail beyond knowing that code like `[target doMethodWith:var1];` gets translated into `objc_msgSend(target,@selector(doMethodWith:),var1);` by the compiler. Knowing what the Objective-C runtime is doing will help you gain a much deeper understanding of Objective-C itself and how your app is run. I think Mac/iPhone Developers will gain something from this, regardless of your level of experience.

The Objective-C Runtime is Open Source

The Objective-C Runtime is open source and available anytime from http://opensource.apple.com. In fact examining the Objective-C is one of the first ways I went through to figure out how it worked, beyond reading Apples documentation on the matter. You can download the current version of the runtime (as of this writting) for Mac OS X 10.6.2 here objc4-437.1.tar.gz.

Dynamic vs Static Languages

Objective-C is a runtime oriented language, which means that when it's possible it defers decisions about what will actually be executed from compile & link time to when it's actually executing on the runtime. This gives you a lot of flexibility in that you can redirect messages to appropriate objects as you need to or you can even intentionally swap method implementations, etc. This requires the use of a runtime which can introspect objects to see what they do & don't respond to and dispatch methods appropriately. If we contrast this to a language like C. In C you start out with a `main()` method and then from there it's pretty much a top down design of following your logic and executing functions as you've written your code. A C struct can't forward requests to perform a function onto other targets. Pretty much you have a program like so

```
#include < stdio.h >

int main(int argc, const char **argv[])
{
        printf("Hello World!");
        return 0;
}
```

which a compiler parses, optimizes and then transforms your optimized code into assembly

```
.text
 .align 4,0x90
 .globl _main
_main:
Leh_func_begin1:
 pushq %rbp
Llabel1:
 movq %rsp, %rbp
Llabel2:
 subq $16, %rsp
Llabel3:
 movq %rsi, %rax
 movl %edi, %ecx
 movl %ecx, -8(%rbp)
 movq %rax, -16(%rbp)
 xorb %al, %al
 leaq LC(%rip), %rcx
 movq %rcx, %rdi
 call _printf
 movl $0, -4(%rbp)
 movl -4(%rbp), %eax
 addq $16, %rsp
 popq %rbp
 ret
Leh_func_end1:
 .cstring
LC:
 .asciz "Hello World!"
```

and then links it together with a library and produces a executable. This contrasts from Objective-C in that while the process is similar the code that the compiler generates depends on the presence of the Objective-C Runtime Library. When we are all initially introduced to Objective-C we are told that (at a simplistic level) what happens to our Objective-C bracket code is something like...

```
[self doSomethingWithVar:var1];
```

gets translated to...

```
objc_msgSend(self,@selector(doSomethingWithVar:),var1);
```

but beyond this we don't really know much till much later on what the runtime is doing.

What is the Objective-C Runtime?

The Objective-C Runtime is a Runtime Library, it's a library written mainly in C & Assembler that adds the Object Oriented capabilities to C to create Objective-C. This means it loads in Class information, does all method dispatching, method forwarding, etc. The Objective-C runtime essentially creates all the support structures that make Object Oriented Programming with Objective-C Possible.

Objective-C Runtime Terminology

So before we go on much further, let's get some terminology out of the way so we are all on the same page about everything. 2 Runtimes As far as Mac & iPhone Developers are concerned there are 2 runtimes: The Modern Runtime & the Legacy Runtime Modern Runtime: Covers all 64 bit Mac OS X Apps & all iPhone OS Apps Legacy Runtime: Covers everything else (all 32 bit Mac OS X Apps) Method There are 2 basic types of methods. Instance Methods (begin with a '-' like `-(void)doFoo;` that operate on Object Instances. And Class Methods (begin with a

'+' like `+ (id)alloc`. Methods are just like C Functions in that they are a grouping of code that performs a small task like

```
-(NSString *)movieTitle
{
    return @"Futurama: Into the Wild Green Yonder";
}
```

Selector A selector in Objective-C is essentially a C data struct that serves as a mean to identify an Objective-C method you want an object to perform. In the runtime it's defined like so...

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

and used like so...

```
SEL aSel = @selector(movieTitle);
```

Message

```
[target getMovieTitleForObject:obj];
```

An Objective-C Message is everything between the 2 brackets '[ ]' and consists of the target you are sending a message to, the method you want it to perform and any arguments you are sending it. A Objective-C message while similar to a C function call is different. The fact that you send a message to an object doesn't mean that it'll perform it. The Object could check who the sender of the message is and based on that decide to perform a different method or forward the message onto a different target object. Class If you look in the runtime for a class you'll come across this...

```
typedef struct objc_class *Class;
typedef struct objc_object {
    Class isa;
} *id;
```

Here there are several things going on. We have a struct for an Objective-C Class and a struct for an object. All the objc_object has is a class pointer defined as isa, this is what we mean by the term 'isa pointer'. This isa pointer is all the Objective-C Runtime needs to inspect an object and see what it's class is and then begin seeing if it responds to selectors when you are messaging objects. And lastly we see the id pointer. The id pointer by default tells us nothing about Objective-C objects except that they are Objective-C objects. When you have a id pointer you can then ask that object for it's class, see if it responds to a method, etc and then act more specifically when you know what the object is that you are pointing to. You can see this as well on Blocks in the LLVM/Clang docs

```
struct Block_literal_1 {
    void *isa; // initialized to &_NSConcreteStackBlock or &_NSCon
creteGlobalBlock
    int flags;
    int reserved;
    void (*invoke)(void *, ...);
    struct Block_descriptor_1 {
 unsigned long int reserved; // NULL
     unsigned long int size;  // sizeof(struct Block_literal_1)
 // optional helper functions
     void (*copy_helper)(void *dst, void *src);
     void (*dispose_helper)(void *src);
    } *descriptor;
    // imported variables
};
```

Blocks themselves are designed to be compatible with the Objective-C runtime so they are treated as objects so they can respond to messages like `-retain`,`-release`,`-copy`,etc. IMP (Method Implementations)

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

IMP's are function pointers to the method implementations that the compiler will generate for you. If your new to Objective-C you don't need to deal with these directly until much later on, but this is how the Objective-C runtime invokes your methods as we'll see soon. Objective-C Classes So what's in an Objectve-C Class? The basic implementation of a class in Objective-C looks like

```
@interface MyClass : NSObject {
//vars
NSInteger counter;
}
//methods
-(void)doFoo;
@end
```

but the runtime has more than that to keep track of

```
#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
```

We can see a class has a reference to it's superclass, it's name, instance variables, methods, cache and protocols it claims to adhere to. The runtime needs this information when responding to messages that message your class or it's instances.

So Classes define objects and yet are objects themselves? How does this work

Yes earlier I said that in objective-c classes themselves are objects as well, and the runtime deals with this by creating Meta Classes. When you send a message like `[NSObject alloc]` you are actually sending a message to the class object, and that class object needs to be an instance of the MetaClass which itself is an instance of the root meta class. While if you say subclass from NSObject, your class points to NSObject as it's superclass. However all meta classes point to the root metaclass as their superclass. All meta classes simply have the class methods for their method list of messages that they respond to. So when you send a message to a class object like `[NSObject alloc]` then `objc_msgSend()` actually looks through the meta class to see what it responds to then if it finds a method, operates on the Class object.

Why we subclass from Apples Classes

So initially when you start Cocoa development, tutorials all say to do things like subclass NSObject and start then coding something and you enjoy a lot of benefits simply by inheriting from Apples Classes. One thing you don't even realize that happens for you is setting your objects up to work with the Objective-C runtime. When we allocate an instance of one of our classes it's done like so...

```
MyObject *object = [[MyObject alloc] init];
```

the very first message that gets executed is `+alloc`. If you look at the documentation it says that "The isa instance variable of the new instance is initialized to a data structure that describes the class; memory for all other instance variables is set to 0." So by inheriting from Apples classes we not only inherit some great attributes, but we inherit the ability to easily allocate and create our objects in memory that matches a structure the runtime expects (with a isa pointer that points to our class) & is the size of our class.

So what's with the Class Cache? (objc_cache *cache)

When the Objective-C runtime inspects an object by following it's isa pointer it can find an object that implements many methods. However you may only call a small portion of them and it makes no sense to search the classes dispatch table for all the selectors every time it does a lookup. So the class implements a cache, whenever you search through a classes dispatch table and find the corresponding selector it puts that into it's cache. So when `objc_msgSend()` looks through a class for a selector it searches through the class cache first. This operates on the theory that if you call a message on a class once, you are likely to call that same message on it again later. So if we take this into account this means that if we have a subclass of `NSObject` called `MyObject` and run the following code

```
MyObject *obj = [[MyObject alloc] init];

@implementation MyObject
-(id)init {
    if(self = [super init]){
        [self setVarA:@"blah"];
    }
    return self;
}
@end
```

the following happens (1) `[MyObject alloc]` gets executed first. MyObject class doesn't implement alloc so we will fail to find `+alloc` in the class and follow the superclass pointer which points to `NSObject` (2) We ask `NSObject` if it responds to `+alloc` and it does. `+alloc` checks the receiver class which is `MyObject` and allocates a block of memory the size of our class and initializes it's isa pointer to the MyObject class and we now have an instance and lastly we put `+alloc in NSObject's class cache for the class object` (3) Up till now we were sending a class messages but now we send an instance message which simply calls `-init` or our designated initializer. Of course our class responds to that message so `-(id)init` get's put into the cache (4) Then `self = [super init]` gets called. Super being a magic keyword that points to the objects superclass so we go to `NSObject` and call it's init method. This is done to insure that OOP Inheritance works correctly in that all your super classes will initialize their variables correctly and then you (being in the subclass) can initialize your variables correctly and then override the superclasses if you really need to. In the case of NSObject, nothing of huge importance goes on, but that is not always the case. Sometimes important initialization happens. Take this...

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

@interface MyObject : NSObject
{
 NSString *aString;
}

@property(retain) NSString *aString;

@end

@implementation MyObject

-(id)init
{
 if (self = [super init]) {
  [self setAString:nil];
 }
 return self;
}

@synthesize aString;

@end



int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

 id obj1 = [NSMutableArray alloc];
 id obj2 = [[NSMutableArray alloc] init];

 id obj3 = [NSArray alloc];
 id obj4 = [[NSArray alloc] initWithObjects:@"Hello",nil];

 NSLog(@"obj1 class is %@",NSStringFromClass([obj1 class]));
 NSLog(@"obj2 class is %@",NSStringFromClass([obj2 class]));

 NSLog(@"obj3 class is %@",NSStringFromClass([obj3 class]));
 NSLog(@"obj4 class is %@",NSStringFromClass([obj4 class]));
 id obj5 = [MyObject alloc];
 id obj6 = [[MyObject alloc] init];

 NSLog(@"obj5 class is %@",NSStringFromClass([obj5 class]));
 NSLog(@"obj6 class is %@",NSStringFromClass([obj6 class]));

 [pool drain];
    return 0;
}
```

Now if you were new to Cocoa and I asked you to guess as to what would be printed you'd probably say

```
NSMutableArray
NSMutableArray
NSArray
NSArray
MyObject
MyObject
```

but this is what happens

```
obj1 class is __NSPlaceholderArray
obj2 class is NSCFArray
obj3 class is __NSPlaceholderArray
obj4 class is NSCFArray
obj5 class is MyObject
obj6 class is MyObject
```

This is because in Objective-C there is a potential for `+alloc` to return an object of one class and then `-init` to return an object of another class.

So what happens in objc_msgSend anyway?

There is actually a lot that happens in `objc_msgSend()`. Lets say we have code like this...

```
[self printMessageWithString:@"Hello World!"];
```

it actually get's translated by the compiler to...

```
objc_msgSend(self,@selector(printMessageWithString:),@"Hello World
!");
```

From there we follow the target objects isa pointer to lookup and see if the object (or any of it's superclasses) respond to the selector `@selector(printMessageWithString:)`. Assuming we find the selector in the class dispatch table or it's cache we follow the function pointer and execute it. Thus `objc_msgSend()` never returns, it begins executing and then follows a pointer to your methods and then your methods return, thus looking like `objc_msgSend()` returned. Bill Bumgarner went into much more detail (Part 1, Part 2 & Part 3) on `objc_msgSend()` than I will here. But to summarize what he said and what you'd see looking at the Objective-C runtime code... 1. Checks for Ignored Selectors & Short Circut - Obviously if we are running under garbage collection we can ignore calls to `-retain`,`-release`, etc 2. Check for nil target. Unlike other languages messaging nil in Objective-C is perfectly legal & there are some valid reasons you'd want to. Assuming we have a non nil target we go on... 3. Then we need to find the IMP on the class, so we first search the class cache for it, if found then follow the pointer and jump to the function 4. If the IMP isn't found in the cache then the class dispatch table is searched next, if it's found there follow the pointer and jump to the pointer 5. If the IMP isn't found in the cache or class dispatch table then we jump to the forwarding mechanism This means in the end your code is transformed by the compiler into C functions. So a method you write like say...

```
-(int)doComputeWithNum:(int)aNum
```

would be transformed into...

```
int aClass_doComputeWithNum(aClass *self,SEL _cmd,int aNum)
```

And the Objective-C Runtime calls your methods by invoking function pointers to those methods. Now I said that you cannot call those translated methods directly, however the Cocoa Framework does provide a method to get at the pointer...

```
//declare C function pointer
int (computeNum *)(id,SEL,int);

//methodForSelector is COCOA & not ObjC Runtime
//gets the same function pointer objc_msgSend gets
computeNum = (int (*)(id,SEL,int))[target methodForSelector:@selec
tor(doComputeWithNum:)];

//execute the C function pointer returned by the runtime
computeNum(obj,@selector(doComputeWithNum:),aNum);
```

In this way you can get direct access to the function and directly invoke it at runtime and even use this to circumvent the dynamism of the runtime if you absolutely need to make sure that a specific method is executed. This is the same way the Objective-C Runtime invokes your method, but using `objc_msgSend()`.

Objective-C Message Forwarding

In Objective-C it's very legal (and may even be an intentional design decision) to send messages to objects to which they don't know how to respond to. One reason Apple gives for this in their docs is to simulate multiple inheritance which

Objective-C doesn't natively support, or you may just want to abstract your design and hide another object/class behind the scenes that deals with the message. This is one thing that the runtime is very necessary for. It works like so 1. The Runtime searches through the class cache and class dispatch table of your class and all the super classes, but fails to to find the specified method 2. The Objective-C Runtime will call `+ (BOOL) resolveInstanceMethod:(SEL)aSEL` on your class. This gives you a chance to provide a method implementation and tell the runtime that you've resolved this method and if it should begin to do it's search it'll find the method now. You could accomplish this like so... define a function...

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

you could then resolve it like so using `class_addMethod()`...

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

The "v@:" in the last part of `class_addMethod()` is what the method is returning and it's arguments. You can see what you can put there in the Type Encodings section of the Runtime Guide. 3. The Runtime then calls `- (id)forwardingTargetForSelector:(SEL)aSelector`. What this does is give you a chance (since we couldn't resolve the method (see #2 above)) to point the Objective-C runtime at another object which should respond to the message, also this is better to do before the more expensive process of invoking `- (void)forwardInvocation:(NSInvocation *)anInvocation` takes over. You could implement it like so

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

Obviously you don't want to ever return self from this method or it could result in an infinite loop. 4. The Runtime then tries one last time to get a message sent to it's intended target and calls `- (void)forwardInvocation:(NSInvocation *)anInvocation`. If you've never seen `NSInvocation`, it's essentially an Objective-C Message in object form. Once you have an NSInvocation you essentially can change anything about the message including it's target, selector & arguments. So you could do...

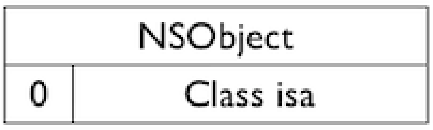
```
-(void)forwardInvocation:(NSInvocation *)invocation
{
    SEL invSEL = invocation.selector;

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

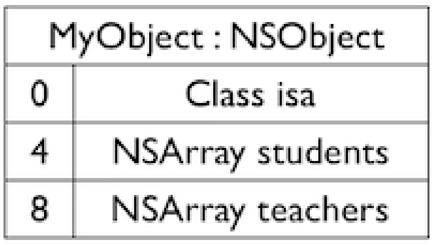
by default if you inherit from NSObject it's `- (void)forwardInvocation:(NSInvocation *)anInvocation` implementation simply calls `-doesNotRecognizeSelector:` which you could override if you wanted to for one last chance to do something about it.

Non Fragile ivars (Modern Runtime)

One of the things we recently gained in the modern runtime is the concept of Non Fragile ivars. When compiling your classes a ivar layout is made by the compiler that shows where to access your ivars in your classes, this is the low level detail of getting a pointer to your object, seeing where the ivar is offset in relation to the beginning of the bytes the object points at, and reading in the amount of bytes that is the size of the type of variable you are reading in. So your ivar layout may look like this, with the number in the left column being the byte offset.

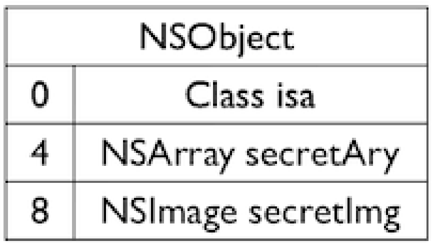


NSObject	
0	Class isa

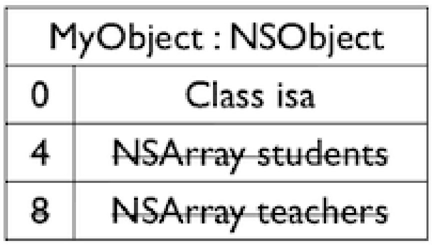


MyObject : NSObject	
0	Class isa
4	NSArray students
8	NSArray teachers

Here we have the ivar layout for NSObject and then we subclass NSObject to extend it and add on our own ivars. This works fine until Apple ships a update or all new Mac OS X 10.x release and this happens

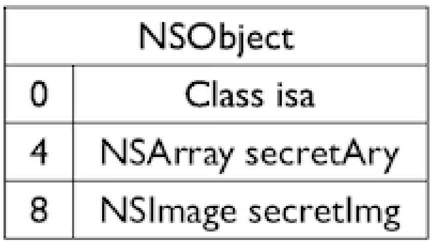


NSObject	
0	Class isa
4	NSArray secretAry
8	NSImage secretImg

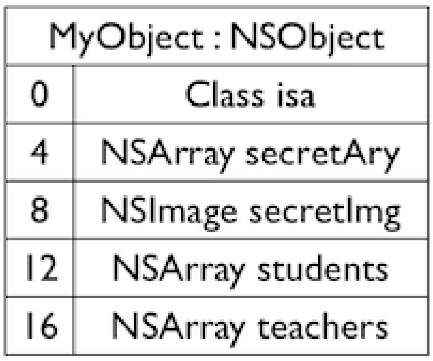


MyObject : NSObject	
0	Class isa
4	~~NSArray students~~
8	~~NSArray teachers~~

Your custom objects get wiped out because we have an overlapping superclass. The only alternative that could prevent this is if Apple sticked with the layout it had before, but if they did that then their Frameworks could never advance because their ivar layouts were frozen in stone. Under fragile ivars you have to recompile your classes that inherit from Apples classes to restore compatibility. So what Happens under non fragile ivars?



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

Under Non Fragile ivars the compiler generates the same ivar layout as under fragile ivars. However when the runtime detects an overlapping superclass it adjusts the offsets to your additions to the class, thus your additions in a subclass are preserved.

Objective-C Associated Objects

One thing recently introduced in Mac OS X 10.6 Snow Leopard was called Associated References. Objective-C has no support for dynamically adding on variables to objects unlike some other languages that have native support for this. So up until now you would have had to go to great lengths to build the infrastructure to pretend that you are adding a variable onto a class. Now in Mac OS X 10.6, the Objective-C Runtime has native support for this. If we wanted to add a variable to every class that already exists like say NSView we could do so like this...

```
#import < Cocoa/Cocoa.h> //Cocoa
#include < objc/runtime.h> //objc runtime api's

@interface NSView (CustomAdditions)
@property(retain) NSImage *customImage;
@end

```

```
@implementation NSView (CustomAdditions)

static char img_key; //has a unique address (identifier)

-(NSImage *)customImage
{
    return objc_getAssociatedObject(self,&img_key);
}

-(void)setCustomImage:(NSImage *)image
{
    objc_setAssociatedObject(self,&img_key,image,
                             OBJC_ASSOCIATION_RETAIN);
}

@end
```

you can see in runtime.h the options for how to store the values passed to `objc_setAssociatedObject()`.

```
/* Associated Object support. */

/* objc_setAssociatedObject() options */
enum {
    OBJC_ASSOCIATION_ASSIGN = 0,
    OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1,
    OBJC_ASSOCIATION_COPY_NONATOMIC = 3,
    OBJC_ASSOCIATION_RETAIN = 01401,
    OBJC_ASSOCIATION_COPY = 01403
};
```

These match up with the options you can pass in the `@property` syntax.

Hybrid vTable Dispatch

If you look through the modern runtime code you'll come across this (in objc-runtime-new.m)...

```
/******************************************************************
******
* vtable dispatch
*
* Every class gets a vtable pointer. The vtable is an array of IMP
s.
* The selectors represented in the vtable are the same for all cla
```

```
sses
*   (i.e. no class has a bigger or smaller vtable).
* Each vtable index has an associated trampoline which dispatches
to
*   the IMP at that index for the receiver class's vtable (after
*   checking for NULL). Dispatch fixup uses these trampolines inst
ead
*   of objc_msgSend.
* Fragility: The vtable size and list of selectors is chosen at la
unch
*   time. No compiler-generated code depends on any particular vta
ble
*   configuration, or even the use of vtable dispatch at all.
* Memory size: If a class's vtable is identical to its superclass'
s
*   (i.e. the class overrides none of the vtable selectors), then
*   the class points directly to its superclass's vtable. This mea
ns
*   selectors to be included in the vtable should be chosen so the
y are
*   (1) frequently called, but (2) not too frequently overridden.
In
*   particular, -dealloc is a bad choice.
* Forwarding: If a class doesn't implement some vtable selector, t
hat
*   selector's IMP is set to objc_msgSend in that class's vtable.
* +initialize: Each class keeps the default vtable (which always
*   redirects to objc_msgSend) until its +initialize is completed.
*   Otherwise, the first message to a class could be a vtable disp
atch,
*   and the vtable trampoline doesn't include +initialize checking
.
* Changes: Categories, addMethod, and setImplementation all force
vtable
*   reconstruction for the class and all of its subclasses, if the

*   vtable selectors are affected.
******************************************************************
****/
```

The idea behind this is that the runtime is trying to store in this vtable the most called selectors so this in turn speeds up your app because it uses fewer instructions than `objc_msgSend`. This vtable is the 16 most called selectors which make up an overwheling majority of all the selectors called globally, in fact further down in the code you can see the default selectors for Garbage Collected & non Garbage Collected apps...

```
static const char * const defaultVtable[] = {
    "allocWithZone:",
    "alloc",
    "class",
    "self",
    "isKindOfClass:",
    "respondsToSelector:",
    "isFlipped",
    "length",
    "objectForKey:",
    "count",
    "objectAtIndex:",
    "isEqualToString:",
    "isEqual:",
    "retain",
    "release",
    "autorelease",
};
static const char * const defaultVtableGC[] = {
    "allocWithZone:",
    "alloc",
    "class",
    "self",
    "isKindOfClass:",
    "respondsToSelector:",
    "isFlipped",
    "length",
    "objectForKey:",
    "count",
    "objectAtIndex:",
    "isEqualToString:",
    "isEqual:",
    "hash",
    "addObject:",
    "countByEnumeratingWithState:objects:count:",
};
```

So how will you know if your dealing with it? You'll see one of several methods called in your stack traces while your debugging. All of these you should basically treat just like they are `objc_msgSend()` for debugging purposes... `objc_msgSend_fixup` happens when the runtime is assigning one of these methods that your calling a slot in the vtable. `objc_msgSend_fixedup` occurs when your calling one of these methods that was supposed to be in the vtable but is no longer in there `objc_msgSend_vtable[0-15]` you'll might see a call to something like `objc_msgSend_vtable5` this means you are calling one of these common methods in the vtable. The runtime can assign and unassign these as it wants to, so you shouldn't count on the fact that `objc_msgSend_vtable10` corresponds to `-length` on one run means it'll ever be there on any of your next runs.

Conclusion

I hope you liked this, this article essentially makes up the content I covered in my Objective-C Runtime talk to the Des Moines Cocoaheads (a lot to pack in for as long a talk as we had.) The Objective-C Runtime is a great piece of work, it does a lot powering our Cocoa/Objective-C apps and makes possible so many features we just take for granted. Hope I hope if you haven't yet you'll take a look through these docs Apple has that show how you can take advantage of the Objective-C Runtime. Thanks! Objective-C Runtime Programming Guide Objective-C Runtime Reference

Posted by Colin Wheeler at 3:53 PM

Labels: CocoaHeads, Objective-C, Objective-C Runtime

26 comments:

TJ said...

Thanks for the post!

7:47 PM

Anonymous said...

"it's" = "it is" (a contraction)
"its" = "belongs to it" (possessive)

It's better to avoid the contraction *entirely* if you have problems keeping track of which is which.

1:20 PM

Anonymous said...

To add on to anon above..

"gets" is *never, ever, ever* written as "get's".

Yhank you for the exhaustive coverage on the runtime, however. :)

3:44 PM

Anonymous said...

Oops @ typo.

3:45 PM

Matt said...

Apparently Anonymous makes mistakes as well.

7:07 PM

Anonymous said...

Excellent article.

11:23 PM

mj said...

Great article, Colin... regardless of "its" anonymous English-language snob commentators

11:21 AM

sshjason said...

Amazing post, great job and thanks a ton!

4:01 AM

Daniel Higginbotham said...

holy crap, I just found your blog and read this post, and it was very helpful! Thank you for putting so much effort into your posts.

11:13 PM

owf said...

@Anonymous:

get's my favourite word.

7:38 AM

shunyuan said...

Thanks for your post.

2:12 PM

Anonymous said...

Not to ask a dumb question, but in your article you state "The Object could check who the sender of the message is and based on that decide to perform a different method or forward the message onto a different target object." How does one do this?

12:08 PM

Colin Wheeler said...

Anonymous
one easy way is through the IB Methods usually you write them like

```
-(IBAction)doFoo:(id)sender;
```

and in the method you can do

```
if([sender isEqual:thatButton]) {
//do something special because thatButton sent msg
}
```

2:07 PM

Colin Wheeler said...

Anonymous
one easy way is through the IB Methods usually you write them like
-(IBAction)doFoo:(id)sender;

and in the method you can do

```
if([sender isEqual:thatButton]) {
//do something special because thatButton sent msg
}
```

2:07 PM

SAKrisT said...

Thanks! Very nice post!

4:23 PM

Anonymous said...

awesome post!!! god bless you...:D

7:07 AM

Julian Yap said...

That was an interesting read. Thanks.

3:34 AM

Stephen van Egmond said...

Your markup is a little bit borked; where you are describing the terminology, e.g. IMP, there ought to be an of some kind or dl/dt/dd set.

11:37 AM

Anonymous said...

very interesting, thanks

2:24 PM

purnachandra said...

Thanks for very good explanation ofrun time support for the Objective-C language.

1:57 AM

purnachandra said...

Very good article of run time support of Objective-C.

1:58 AM

Shadow said...

Fabulous article!

Unlike Smalltalk, which seems to be held together by a combination of magic and philosophical paradoxes, it's amazing how rational, pragmatic, and sensibly-wrought Objective-C is. Much like a car, open it up, dig in, and you can see what makes it go! Beautiful!

BTW, I'm a proud English snob commentator, and found myself twitching uncontrollably at every misplaced "it's" I saw---though I won't hide behind anonymity to say it! ^_^

7:48 PM

Anonymous said...

I appreciate the post ... I feel I understand many advantages of Object-C runtime environment now.

Thanks,

3:31 PM

Anonymous said...

Thank you for this detailed and neat article

7:36 AM

Anonymous said...

thanks. Great article.

4:19 AM

Anonymous said...

Great article! Insightful and detailed, without being overly complex (though some parts were hard). Keep up the good work!

2:00 PM

Post a Comment

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

About Me

Colin Wheeler

I am a Cocoa Developer & Enthusiast working on Mac & iOS apps.

About.me/colinwheeler

View my complete profile

Me

Github: Machx (My Profile & Open Source Code)

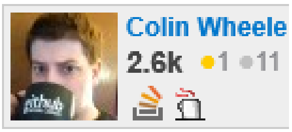
Follow @cocoasamu

Follow @CocoaSam

cocoasamurai at gmail.com

My About.me page

LinkedIn



Amazon Wishlist

Categories

- [at]synchronized Lock (1)
- Blocks (2)
- Book Review (2)
- Carbon (1)
- Clang Static Analyzer (1)
- CocoaHeads (4)
- Debugging (3)
- DTrace (5)
- DVCS (3)
- Events (1)
- Git (3)
- Grand Central Dispatch (3)
- Instruments (2)
- iPhone SDK (1)
- Keyboard Shortcuts (2)
- Late Night Cocoa (1)
- libxml2 (1)
- LLVM (1)
- Mac Developer Network (2)
- Mac Developer Roundtable Podcast (1)
- Multithreading (4)
- MVVM (1)
- NSCoder Night (1)
- NSConference (1)

- NSLock (1)
- NSObject Threading (1)
- NSOperation (5)
- NSRunLoop (2)
- NSThread (3)
- Objective-C (6)
- Objective-C Runtime (2)
- OSSpinLock (1)
- Podcasts (1)
- POSIX Mutex Lock (1)
- POSIX RWLock (1)
- PubSub Framework (1)
- ReactiveCocoa (1)
- SCM (2)
- Shark (1)
- TextMate Bundle (2)
- Thread Locking (1)
- Threading (2)
- UI Design (3)
- VMware (1)
- WWDC (3)
- WWDC 2008 (2)
- WWDC 2009 (1)
- WWDC Students (1)
- Xcode (3)
- XML Parsing (1)

Interesting Mac/iOS Dev Blogs & Links

- Borkware
- Wolf Rentzsch
- Cocoa with Love
- LLVM Project Blog
- NSBlog
- Domain of the Bored
- iOS Dev Tips
- carpeaqua
- Eschantology
- Will Shipley
- Cocoa Dev

Central
- CocoaDev
- Google Mac Blog
- WebKit Blog

Blog Archive

- ▶ 2013 (1)
- ▶ 2012 (2)
- ▶ 2011 (3)
- ▼ 2010 (9)
  - ▶ December (1)
  - ▶ November (1)
  - ▶ September (1)
  - ▶ August (1)
  - ▶ June (1)
  - ▶ April (1)
  - ▶ February (1)
  - ▼ January (2)
    - Understanding the Objective-C Runtime
    - Des Moines Cocoaheads 1/14: Understanding the Obje...
- ▶ 2009 (19)
- ▶ 2008 (19)
- ▶ 2007 (12)
- ▶ 2006 (20)

...