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

中文　http://www.tuicool.com/articles/M7ZJFn

What is a meta-class in Objective-C?

In this post, I look at one of the stranger concepts in Objective-C — the meta-class. Every class in Objective-C has its own associated meta-class but since you rarely ever use a meta-class directly, they can remain enigmatic. I'll start by looking at how to create a class at runtime. By examining the "class pair" that this creates, I'll explain what the meta-class is and also cover the more general topic of what it means for data to be an object or a class in Objective-C.

Creating a class at runtime

The following code creates a new subclass of `NSError` at runtime and adds one method to it:

```
Class newClass =
    objc_allocateClassPair([NSError class], "RuntimeErrorSubclass", 0);
class_addMethod(newClass, @selector(report), (IMP)ReportFunction, "v@:");
objc_registerClassPair(newClass);
```

The method added uses the function named `ReportFunction` as its implementation, which is defined as follows:

```
void ReportFunction(id self, SEL _cmd)
{
    NSLog(@"This object is %p.", self);
    NSLog(@"Class is %@, and super is %@.", [self class], [self superclass]);

    Class currentClass = [self class];
    for (int i = 1; i < 5; i++)
    {
        NSLog(@"Following the isa pointer %d times gives %p", i, currentClass);
        currentClass = object_getClass(currentClass);
    }

    NSLog(@"NSObject's class is %p", [NSObject class]);
    NSLog(@"NSObject's meta class is %p", object_getClass([NSObject class]));
}
```

On the surface, this is all pretty simple. Creating a class at runtime is just three easy steps:

1. Allocate storage for the "class pair" (using `objc_allocateClassPair`).
2. Add methods and ivars to the class as needed (I've added one method using `class_addMethod`).
3. Register the class so that it can be used (using `objc_registerClassPair`).

However, the immediate question is: what is a "class pair"? The function `objc_allocateClassPair` only returns one value: the class. Where is the other half of the pair?

I'm sure you've guessed that the other half of the pair is the meta-class (it's the title of this post) but to explain what that is and why you need it, I'm going to give some background on objects and classes in Objective-C.

What is needed for a data structure to be an object?

Every object has a class. This is a fundamental object-oriented concept but in Objective-C, it is also a fundamental part of the data. Any data structure which has a pointer to a class in the right location can be treated as an object.

In Objective-C, an object's class is determined by its `isa` pointer. The `isa` pointer points to the object's Class.

In fact, the basic definition of an object in Objective-C looks like this:

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

What this says is: any structure which starts with a pointer to a `Class` structure can be treated as an `objc_object`.

The most important feature of objects in Objective-C is that you can send messages to them:

```
[@"stringValue"
    writeToFile:@"/file.txt" atomically:YES encoding:NSUTF8StringEncoding error:NULL];
```

This works because when you send a message to an Objective-C object (like the `NSCFString` here), the runtime follows object's `isa` pointer to get to the object's `Class` (the `NSCFString` class in this case). The `Class` then contains a list of the `Method`s which apply to all objects of that `Class` and a pointer to the `superclass` to look up inherited methods. The runtime looks through the list of `Method`s on the `Class` and superclasses to find one that matches the message selector (in the above case, `writeToFile:atomically:encoding:error` on `NSString`). The runtime then invokes the function (`IMP`) for that method.

The important point is that the `Class` defines the messages that you can send to an object.

What is a meta-class?

Now, as you probably already know, a `Class` in Objective-C is also an object. This means that you can send messages to a `Class`.

```
NSStringEncoding defaultStringEncoding = [NSString defaultStringEncoding];
```

In this case, `defaultStringEncoding` is sent to the `NSString` class.

This works because every `Class` in Objective-C is an object itself. This means that the `Class` structure must start with an `isa` pointer so that it is binary compatible with the `objc_object` structure I showed above and the next field in the structure must be a pointer to the `superclass` (or `nil` for base classes).

As I showed last week, there are a couple different ways that a `Class` can be defined, depending on the version of the runtime you are running, but yes, they all start with an `isa` field followed by a `superclass` field.

```
typedef struct objc_class *Class;
struct objc_class {
    Class isa;
    Class super_class;
    /* followed by runtime specific details... */
};
```

However, in order to let us invoke a method on a `Class`, the `isa` pointer of the `Class` must itself point to a `Class` structure and that `Class` structure must contain the list of `Method`s that we can invoke on the Class.

This leads to the definition of a meta-class: the meta-class is the class for a `Class` object.

Simply put:

- When you send a message to an object, that message is looked up in the method list on the object's class.
- When you send a message to a class, that message is looked up in the method list on the class' meta-class.

The meta-class is essential because it stores the class methods for a `Class`. There must be a unique meta-class for every `Class` because every `Class` has a potentially unique list of class methods.

What is the class of the meta-class?

The meta-class, like the `Class` before it, is also an object. This means that you can invoke methods on it too. Naturally, this means that it must also have a class.

All meta-classes use the base class' meta-class (the meta-class of the top `Class` in their inheritance hierarchy) as their class. This means that for all classes that descend from `NSObject` (most classes), the meta-class has the `NSObject` meta-class as its class.

Following the rule that all meta-classes use the base class' meta-class as their class, any base meta-classes will be its own class (their `isa` pointer points to themselves). This means that the `isa` pointer on the `NSObject` meta-class points to itself (it is an instance of itself).

Inheritance for classes and meta-classes

In the same way that the `Class` points to the superclass with its `super_class` pointer, the meta-class points to the meta-class of the `Class`' `super_class` using its own `super_class` pointer.

As a further quirk, the base class' meta-class sets its `super_class` to the base class itself.

The result of this inheritance hierarchy is that all instances, classes and meta-classes in the hierarchy inherit from the hierarchy's base class.

For all instances, classes and meta-classes in the `NSObject` hierarchy, this means that all `NSObject` instance methods are valid. For the classes and meta-classes, all `NSObject` class methods are also valid.

All this is pretty confusing in text. Greg Parker has put together an excellent diagram of instances, classes, meta-classes and their super classes and how they all fit together.

Experimental confirmation of this

To confirm all of this, let's look at the output of the `ReportFunction` I gave at the start of this post. The purpose of this function is to follow the `isa` pointers and log what it finds.

To run the `ReportFunction`, we need to create an instance of the dynamically created class and invoke the `report` method on it.

```
id instanceOfNewClass =
    [[newClass alloc] initWithDomain:@"someDomain" code:0 userInfo:nil];
[instanceOfNewClass performSelector:@selector(report)];
[instanceOfNewClass release];
```

Since there is no declaration of the `report` method, I invoke it using `performSelector:` so the compiler doesn't give a warning.

The `ReportFunction` will now traverse through the `isa` pointers and tell us what objects are used as the class, meta-class and class of the meta-class.

> **Getting the class of an object:** the `ReportFunction` uses `object_getClass` to follow the `isa` pointers because the `isa` pointer is a protected member of the class (you can't directly access other object's `isa` pointers). The `ReportFunction` does not use the `class` method to do this because invoking the `class` method on a `Class` object does not return the meta-class, it instead returns the `Class` again (so `[NSString class]` will return the `NSString` class instead of the `NSString` meta-class).

This is the output (minus `NSLog` prefixes) when the program runs:

```
This object is 0x10010c810.
Class is RuntimeErrorSubclass, and super is NSError.
Following the isa pointer 1 times gives 0x10010c600
Following the isa pointer 2 times gives 0x10010c630
Following the isa pointer 3 times gives 0x7fff71038480
Following the isa pointer 4 times gives 0x7fff71038480
NSObject's class is 0x7fff710384a8
NSObject's meta class is 0x7fff71038480
```

Looking at the addresses reached by following the `isa` value repeatedly:

- the object is address `0x10010c810`.
- the class is address `0x10010c600`.
- the meta-class is address `0x10010c630`.
- the meta-class's class (i.e. the `NSObject` meta-class) is address `0x7fff71038480`.
- the `NSObject` meta-class' class is itself.

The value of the addresses is not really important except that it shows the progress from class to meta-class to `NSObject` meta-class as discussed.

Conclusion

The meta-class is the class for a `Class` object. Every `Class` has its own unique meta-class (since every `Class` can have its own unique list of methods). This means that all `Class` objects are not themselves all of the same class.

The meta-class will always ensure that the `Class` object has all the instance *and* class methods of the base class in the hierarchy, plus all of the class methods in-between. For classes descended from `NSObject`, this means that all the `NSObject` instance and protocol methods are defined for all `Class` (and meta-class) objects.

All meta-classes themselves use the base class' meta-class (`NSObject` meta-class for `NSObject` hierarchy classes) as their class, including the base level meta-class which is the only self-defining class in the runtime.

Posted by Matt Gallagher
on Sunday, January 17, 2010
Filed in categories: Foundation, Objective-C