Effective Objective 2.0 读书笔记 之 动态方法解析实现@dynamic属性

2017-01-04

我们要知道在Objective-C中，如果像某对象传递消息，那就会使用动态绑定机制来决定需要调用的方法。在底层，所有方法都是普通的C语言函数，然而对象收到消息之后，究竟该调用哪个方法则完全于运行期决定，甚至可以在程序运行时改变(最常见的运用场景就是`Method Swizzle`黑魔法)，这些特性使得Objective-C成为一门真正的动态语言。
给对象发送消息：

```
1  id returnValue = objc_msgSend(someObject,@selector(messageName:),parame
```

objc_msgSend方法会在接收者所属的类中搜寻其”方法列表 list of methods (可通过`class_copyMethodList(objClass,&count)`方法获取对象所有方法列表) “,如果能找到与选择子(也就是messageName)名称相符的方法就跳转至其实现代码。若找不到则沿着继承体系向上查找，找到方法再跳转(note : 找到后，objc_msgSend会将匹配的结果缓存在`快速映射表(fast map)`里面,每个类都有这样一块缓存，若是稍后还向该类发送与选择子相同的消息，那么执行就很快。当然还是稍微不如静态绑定的函数调用快，不过消息派发机制并

不会成为应用程序的瓶颈啦！），
如果最终还是找不到相符的方法，那就执行消息转发（message forwarding）操作。消息转发分为两大阶段，动态方法解析阶段和完整的消息转发阶段。
本文将用例子来演示第一阶段的运用。

以完整例子演示动态方法解析

对象在收到无法解读的消息后，首先将调用其所属类的下列类方法：

```
+(BOOL)resolveInstanceMethod:(SEL)selector
```

该方法就是那个未知的选择子，其返回值为Boolean类型，标识这个类是否能新增一个实例方法用以处理此选择子。在继续往下执行转发机制之前，本类有机会新增一个处理此选择自的方法。假如尚未实现的方法不是实例方法而是类方法，那么就是另一个方法resolveClassMethod:.
使用这种方法的前提是：相关方法的实现代码已经写好，只等着运行的时候动态插在类里面就可以了。比如接下来的实现@dynamic属性实例。
假设要编写一个类似于“字典”的对象，他里面可以容纳其他对象，只不过开发者要直接通过属性存取其中的数据，这个类的设计思路是：由开发者来添加属性定义，
并将其声明为@dynamic，而类则会自动处理相关属性值的存放与获取操作。

```
//类接口  .h
#import <Foundation/Foundation.h>
@interface TestDynamicDict : NSObject
@property (nonatomic, strong) NSString *string;
```

```
@property (nonatomic, strong) NSNumber *number;
@property (nonatomic, strong) NSData *data;
@property (nonatomic, strong) id opaqueObject;
@end
//note：属性具体是什么数据类型无关紧要，只是显示此功能的作用。
```

```
//类实现 .m
#import "TestDynamicDict.h"
#import <objc/runtime.h>
@interface TestDynamicDict(){
}
//将属性声明为@dynamic，让编译器不要为其自动合成实例变量的存取方法。
@property (nonatomic, strong) NSMutableDictionary *backStore;
@end

@implementation TestDynamicDict
@dynamic string, number, data, opaqueObject;
-(id)init{
    if(self = [super init]){
        _backStore = [NSMutableDictionary new];
    }
    return self;
}
//关键代码，resolveInstanceMethod的实现
+(BOOL)resolveInstanceMethod:(SEL)selector{
    NSString *selectorString = NSStringFromSelector(selector);
    if([selectorString hasPrefix:@"set"]){//setter
        class_addMethod(self, selector, (IMP)autoDictionarySetter, "v@
    }else{//getter
        class_addMethod(self, selector, (IMP)autoDictionaryGetter, "v@
    }
    return YES;
}
//getter 函数实现
id autoDictionaryGetter(id self, SEL _cmd){
    TestDynamicDict *typeSelf = (TestDynamicDict *)self;
    NSMutableDictionary *backStore = typeSelf.backStore;
    NSString *key = NSStringFromSelector(_cmd);
    return [backStore objectForKey:key];

}
//setter 函数实现
void autoDictionarySetter(id self, SEL _cmd, id value){
    TestDynamicDict *typeSelf = (TestDynamicDict *)self;
    NSMutableDictionary *backStore = typeSelf.backStore;
    NSString *selectorKey = NSStringFromSelector(_cmd);
    NSMutableString *key = [selectorKey mutableCopy];
    //remove : end
    [key deleteCharactersInRange:NSMakeRange(key.length - 1, 1)];
    //remove set
```

```
    [key deleteCharactersInRange:NSMakeRange(0, 3)];
    NSString *lowerFirstChar = [[key substringToIndex:1]lowercaseStrin
    [key replaceCharactersInRange:NSMakeRange(0, 1) withString:lowerFi
    if(value){
        [backStore setObject:value forKey:key];
    }else{
        [backStore removeObjectForKey:key];
    }
}
@end
```

首次在TestDynamicDict实力上访问某个属性时，运行期系统还找不到对应选择子，因为所需的选择子既没有直接实现，也没有合成出来。假设写入string属性，那么系统会以`setString:`为选择子调用上面的这个方法，同理读取时选择子为`string`。此时就利用resolveInstanceMethod方法向类中新增一个处理选择子所用的方法，这两个方法分别以autoDictionarySetter和autoDictionaryGetter函数指针的形式出现。此时就用到了class_addMethod来向类中动态新增方法，其中最后一个参数标识待添加方法的类型编码 （`type encoding`），可以通过`method_getTypeEncoding(Method m)`方法打印查看。

```
//用法
TestDynamicDict *dict = [TestDynamicDict new];
dict.string = @"dynamic is work";
NSLog(@"dict.string = %@" ,dict.string);
```

output

```
dict.string = dynamic is work
```

其他属性的访问方式与string类似，想要添加新属性，只需要用@property来定义，并将其声明为@dynamic即可。

在iOS的CoreAnimation框架中，CALayer类就用到与本例相似的实现方式，这使得CALayer成为兼容于键值编码的容器类，能够向里面随意添加属性，然后以键值对的形式来访问。键值存储工作由基类负责，我们只需在CALayer的子类中定义新属性即可。

若经过上述两步之后还没办法处理选择子，那就启动完整的消息转发机制。有时间再整理完整消息转发机制的内容。