了解 RunTime 运行机制

1. 在既有类中使用关联对象存放自定义数据

(1) 概要：

1) 有时需在一个封装好的实例对象中存放更多相关信息，通常我们会从对象所属类中继承一个子类，然后改用这个子类对象。然而，有时候类的实例可能是由某种机制创建的，而开发者无法令这种机制创建子类实例。此时，可用关联对象特性来解决这个问题。

2) 存储对象时可指明存储策略，以维护相应内存管理语义 objc_AssociationPolicy

3) 使用 objc_setAssociatedObject(id object, void *key, id value, objc_AssociationPolicy policy)//设置关联对象

使用 objc_getAssociatedObject(id object,void *key)//获取关联对象

使用 objc_removeAssociatedObject(id object)//移除指定对象所有关联对象

4) 字典中 2 个 key isEqual 为真字典认为二者是相等的；而关联对象，若想让两个 key 匹配同一个值，必须地址一样，所以设置关联对象值时，通常使用静态常量，取其地址为键

5) 可以在 alertViewShow 附近写一个 block，void(^block)(NSInteger param)=^(){};关联到 alertView，在代理方法里把点击的值回传过来，那么 alertViewShow 和 alert 事件逻辑就可以写一起了。但若是多次调用 alert 视图，那么从 alertView 继承，添加 block 作为属性更好。

(2) 要点：

1) 可通过关联对象机制来把两个对象关联起来。

2) 定义关联对象时可指定内存管理语义，用以模仿定义属性时所采用的拥有关系和非拥有关系。

3) 只有在其他做法均不可行时，迫不得已才用此法，否者可能引发难以查找的 bug

2. 理解 objc_msgSend 的作用

(1) 概要

1) 给对象发送消息可以这么写

id returnValue = [someObject messageName:paramter];

编译器见此消息后，转化为标准 c 语言函数 objc_msgSend，原型如下：

void objc_msgSend(id self, SEL _cmd, paramters)，具体转化为

id returnValue = objc_msgSend(someObject, @selector(messageName:), paramter);

消息发送真正的执行者是 objc_msgSend，会从接受者中开始查找消息并执行相应代码。Objc_msgSend 函数会在接受者所属类搜索方法列表，找到与选择子名称相符,则跳到对应方法体内执行代码,找不到,则沿着继承体向上查找，仍无则进行消息转发

2) 消息发送过程虽然烦琐，但 objc_msgSend 会将匹配过的结果缓存在每个类均有的快速映射表这块缓存里。下次同样消息发送到该类就快了。

3) 其他特殊情况需交 oc 运行环境的另一些函数处理

objc_msgSend_stret:待发消息欲返回结构体时

objc_msgSend_fpret:待发消息欲返回浮点数时

objc_msgSendSuper:给超类发消息，上面方法也有对应超类发送方法

4) 之所以 objc_msgSend 能快速跳转至接收者类中执行相应函数，是因为接收者的每个方法均可以视为简单 c 函数，原型如下：

<return_type>Class_Selector(id self, SEL _cmd, …)，这是为了利用尾调用优化技术，如果某函数最后一项操作是调用其他函数，不会将返回值另作他用时，编译器会生成跳转至另一函数所需的指令码，而不会想调用堆栈中推入新栈帧，若无此优化，则会过早发生栈溢出现象

当然，实际编写 oc 代码无需担心这一点，不过应了解消息机制底层工作原理

(2) 要点

1) 消息由接收者，选择子及参数构成。给某对象发送消息也相当于 objc_msgSend 在该对象上调用方法

2) 发送给某对象的全部消息，均需要由动态消息派发系统 obc_msgSend 处理，该系统会查处对应方法并执行其代码

3. 理解消息转发机制

(1) 概要

1) 因为运行期可继续向类中添加方法，所以编译器在编译时还无法确知类中还有其他方式实现，当对象接收到无法解读的消息时，就会启动消息转发机制，程序员可以由此过程告诉对象如何处理这些未知消息，若无处理，则交给 NSObject 的默认实现，抛出异常：unrecognized selector send to instance

2) 消息转发阶段

第一阶段询问接收者所属类，能否通过动态解析动态添加方法实现

第二阶段涉及完整的消息转发机制：第一小步，运行期系统请求接收者有无其他对象能够处理，有则运行期系统把消息转给该对象。若无则启动完整消息转发机制，运行期系统把消息有关全部细节封装到 NSInvocation 对象中，再给接收者最后一次机会，令其设法解决当前未处理消息

3) 动态方法解析

对象收到无法解读消息，将调用如下方法：返回能否新增方法来处理此选择子

- (id)resolveInstanceMethod:(SEL)selector;

使用前提：相关代码已经准备好，等运行时动态添加到类里面

4) 备缓接收者(完整消息转发机制第一小步)

这是运行球系统第二次问话，能否交由其他接收者来处理

- (id)forwardingTargetForSelector:(SEL)selector;

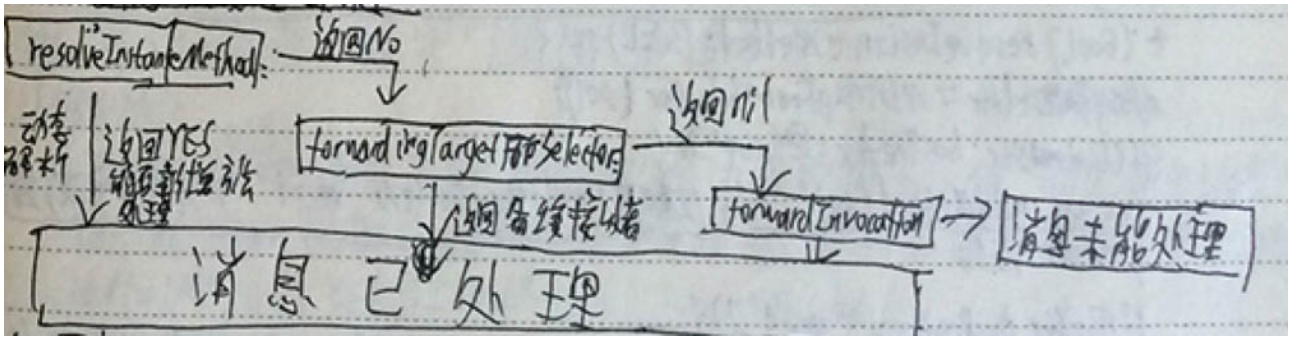
5) 完整的消息转发机制

创建 NSInvocation 对象，把尚未处理消息有关细节封装于其中，包含选择子、目标、参数。在触发 NSInvocation 对象时，消息派发系统把消息指派给目标对象。此步骤调用如下方法：

-(void)forwardInvocatiion:(NSInvocation *)invocation;

可以触发消息前先改细节，追加参数、交换选择子等，若发现调用操作不应由本类实现，则需调用父类同名方法，直至 NSObject 的同名方法被调用，继而调用 doesNotRecognizeSelector 抛出异常，表选择子未能得到处理

6) 消息转发全过程



(2) 要点

1) 若对象无法响应某个选择子(objc_msgSend 查不到 selector)则进入消息转发流程

2) 通过运行期动态方法解析功能，可在需要用到某个方法时再将其加入类中。

3) 对象可以其无法解读的某些选择子转交给其他对象处理

4) 还无法处理选择子，则启动完整消息转发机制

(3) 示例

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

@interface Person1 : NSObject

@property (assign, nonatomic) NSInteger wolegequ;
- (NSInteger)sayhello:(NSString *)words;

@end
```

```
#import "Person1.h"
#import <objc/runtime.h>

int say(id self, SEL _cmd, NSString *str)
{
    NSLog(@"%@", str);
    return 100;//随便返回个值
}

int wolegequGetter(id self, SEL _cmd){
    //读取sef.字典，key为cmd的string，将读到的返回
    return 100;
}

void wolegequSetter(id self, SEL _cmd, NSInteger wolequ){
    NSLog(@"%ld", wolequ);
    //写个self.字典，设置key为cmd的string，value 为 str;
//    Person *person = (Person *)self;
}

@implementation Person1
@dynamic wolegequ;
```

```
+ (BOOL)resolveInstanceMethod:(SEL)sel{
   NSString *selStr = NSStringFromSelector(sel);
//    if([selStr hasPrefix:@"say"]){
//        class_addMethod(self, sel, (IMP)say, "i@:@");
//        return YES;
//    }
   if([selStr hasPrefix:@"set"]){
      class_addMethod(self, sel, (IMP)wolegequSetter, "v@:i");
      return YES;
   }
   if([selStr hasPrefix:@"wole"]){
      class_addMethod(self, sel, (IMP)wolegequGetter, "i@:");
      return YES;
   }

   return [super resolveInstanceMethod:sel];
}

- (id)forwardingTargetForSelector:(SEL)aSelector{
   NSString *selector = NSStringFromSelector(aSelector);
   if([selector hasPrefix:@"say"]){
      //注意以下区别
      return [[[NSClassFromString(@"PersonHelper") class] alloc] init];//实例方法
//        return [NSClassFromString(@"PersonHelper") class];//类方法
   }
   return nil;
}
```

```
- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector{
    NSMethodSignature *signature = [NSMethodSignature methodSignatureForSelector:aSelector];
    return signature;
}
- (void)forwardInvocation:(NSInvocation *)anInvocation{
    [super forwardInvocation:anInvocation];
}
```

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

@interface PersonHelper : NSObject

- (NSInteger)sayhello:(NSString *)words;

@end
```

```
@implementation PersonHelper

- (NSInteger)sayhello:(NSString *)words{
    NSLog(@"这是备援接受者%@", words);
    return 100;//随便返回个值
}
```

4. 用方法调配技术调试黑盒方法

(1) 概述

1) 类的方法列表会把选择子映射到相关方法的实现上，使动态消息派发系统能据此找到并执行相应的方法。这些方法以函数指针表示 IMP，原型如下

id(*IMP)(id, _cmd,…)

2) 交换两个已写好的方法实现

void method_exchangeImplementations(Method m1, Method m2)

参数表示方法实现，可由 Method class_getInstanceMethod(Class aClass, SEL aSelector)获得

如交换 lowercaseString 和 UppercaseString 方法实现

Method originalMethod = class_getInstanceMethod([NSString class], @selector(lower… String));

Method swapedMethod = class_getInstanceMethod([NSString class], @selector(upper..String));

Method_exchangeImplementations(originalMethod, swapedMethod);

3) 添加新方法，在调用 lowercaseString 时打印一条信息

在 NSString 分类中

```
- (NSString*)eoc_myLowercaseString{
    NSString *lowercase = [self eoc_myLowerCaseString];
    NSLog(“%@=>%@”,self,lowercase);
    return lowercase;
  }
```

上述方案可以为不知具体实现的封装方法增加日志记录易于程序调试，一般仅适用于调试用，滥用会令代码不易读与维护

(2) 要点

1) 在运行期，可向类中新增或替换选择子对应的方法实现

2) 使用另一份实现替换原有实现，这道工序叫方法调配，开发者常用此技术向原有实现中添加新功能

3) 一般，调试程序时候才在运行期修改方法实现，这种方法不宜滥用，调试未知崩溃原因时可以以此定位错误位置