iOS Runtime & Memory

文档修订记录

日期	修订版本	修改描述	作者
2016-03-21	0.1	添加 class、runtime 部分描述	Coder4869
2016-03-27	0.1.1	更新了消息机制的部分描述	Coder4869
2016-07-05	0.2	新增 property 部分的描述	Coder4869
2016-07-08	0.2.1	完善 class 和 runtime 下类操作描述	Coder4869
2016-07-12	0.3	添加内存管理面试题描述	Coder4869
2016-07-18	0.3.1	对之前内容进行整理和完善	Coder4869

目 录

iOS基础知识

1 类与对象基础数据结构

1.1 objc_class类

Class 定义<objc/runtime.h>：

```
typedef struct objc_class *Class;
struct objc_class {
    Class isa  OBJC_ISA_AVAILABILITY;  //指向 metaclass (元类)
#if !__OBJC2__
    Class super_class                          OBJC2_UNAVAILABLE;  //指向其父类
    const char *name                           OBJC2_UNAVAILABLE;  //类名
    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
} OBJC2_UNAVAILABLE;
```

- isa：指向 metaclass，即静态的 Class。一般一个 Obj 对象中的 isa 指向普通 Class，该 Class 中存储普通成员变量和对象方法（“-”开头），普通 Class 中的 isa 指针指向静态 Class，静态 Class 中存储 static 类型成员变量和类方法（“+”开头）。
- super_class:指向父类，如果这个类是根类，则为 NULL。
- 所有 metaclass 中 isa 指针都指向根 metaclass。而根 metaclass 则指向自身。Root metaclass 是通过继承 Root class 产生的。与 root class 结构体成员一致，也就是前面提到的结构。不同的是 Root metaclass 的 isa 指针指向自身。
- version：类的版本信息，初始化默认为 0，可以通过 runtime 函数 class_setVersion 和 class_getVersion 进行修改、读取，有助于了解不同版本的实例变量布局的改变。
- info：一些标识信息,如 CLS_CLASS (0x1L) 表示该类为普通 class ，其中包含对象方法和成员变量; CLS_META (0x2L) 表示该类为 metaclass，其中包含类方法;
- instance_size：该类的实例变量大小(包括从父类继承下来的实例变量);
- ivars：用于存储每个成员变量的地址。在 objc_class 中，成员变量、属性的信息放在链表 ivars 中，ivars 是数组，各元素是指向 Ivar(变量信息)的指针。
- methodLists ：与 info 的一些标志位有关,如 CLS_CLASS (0x1L),则存储对象方法，如 CLS_META (0x2L)，则存储类方法;
- cache：指向最近使用的方法的指针，用于提升效率；
- protocols：存储该类遵守的协议

1.2 objc_cache

```
struct objc_cache {
```

```
    unsigned int mask /* total = mask + 1 */        OBJC2_UNAVAILABLE;
    unsigned int occupied                           OBJC2_UNAVAILABLE;
    Method buckets[1]                               OBJC2_UNAVAILABLE;
};
```

mask：一个整数，指定分配的缓存 bucket 的总数。在方法查找过程中，Objective-C runtime 使用这个字段来确定开始线性查找数组的索引位置。指向方法 selector 的指针与该字段做一个 AND 位操作(index = (mask & selector))。这可以作为一个简单的 hash 散列算法。

occupied：一个整数，指定实际占用的缓存 bucket 的总数。

buckets：指向 Method 数据结构指针的数组。这个数组可能包含不超过 mask+1 个元素。需要注意的是，指针可能是 NULL，表示这个缓存 bucket 没有被占用，另外被占用的 bucket 可能是不连续的。这个数组可能会随着时间而增长。

1.2.1 cache用例分析

NSArray *array = [[NSArray alloc] init];

流程解析：

[1]. [NSArray alloc]先被执行。因为 NSArray 没有+alloc 方法，于是去父类 NSObject 去查找。

[2]. 检测 NSObject 是否响应+alloc 方法，发现响应，于是检测 NSArray 类，并根据其所需的内存空间大小开始分配内存空间，然后把 isa 指针指向 NSArray 类。同时，+alloc 也被加进 cache 列表里面。

[3]. 接着，执行-init 方法，如果 NSArray 响应该方法，则直接将其加入 cache；如果不响应，则去父类查找。

[4]. 在后期的操作中，如果再以[[NSArray alloc] init]这种方式来创建数组，则会直接从 cache 中取出相应的方法，直接调用。

1.3 objc_object & id & Object & Class & Meta Class

objc_object 是表示一个类的实例的结构体，它的定义如下(objc/objc.h)：

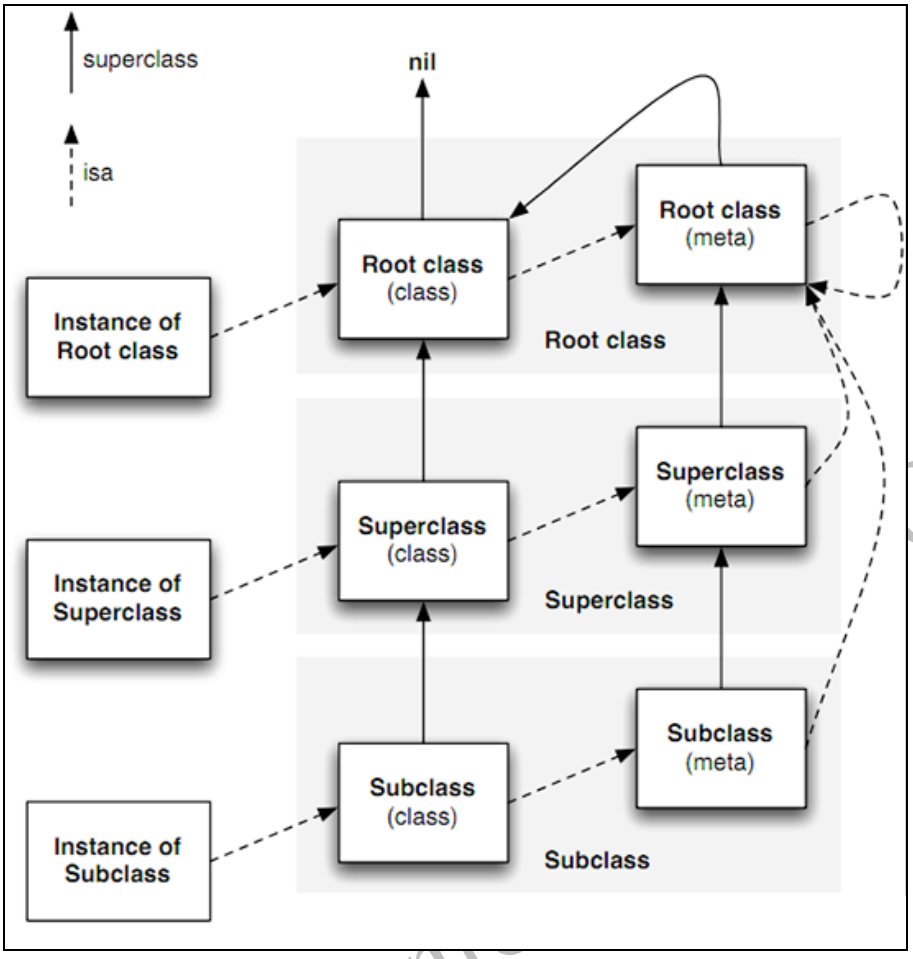
```
struct objc_object {
    Class isa    OBJC_ISA_AVAILABILITY;
};
typedef struct objc_object *id;
```

Class 定义(objc.h)：typedef struct objc_class *Class; Class 本身指向 C 的 struct objc_class。

当创建一个特定类的实例对象时，分配的内存包含一个 objc_object 数据结构，然后是类的实例变量的数据。NSObject 类的 alloc 和 allocWithZone:方法使用函数 class_createInstance 来创建 objc_object 数据结构。

id 是一个 objc_object 结构类型的指针，所以在使用其他 NSObject 类型的实例时需要在前面加上*，而使用 id 时却不用。这有助于实现类似于 C++中泛型的一些操作。该类型的对象可以转换为任何一种对象，类似于 C 语言中 void *指针类型的作用。

可以把 Meta Class 理解为一个 Class 对象的 Class。向一个 Objective-C 对象发送消息时，Runtime 库根据实例对象的 isa 指针找到这个实例对象所属的类，在类的方法列表及父类的方法列表中去寻找与消息对应的 selector 指向的方法。找到后即运行这个方法。向一个类发送消息时，这条消息会在类的 Meta Class 的方法列表里查找。而 Meta Class 本身也是一个 Class，它跟其他 Class 一样也有自己的 isa 和 super_class 指针。如下图：



- 每个 Class 都有一个 isa 指针指向一个唯一的 Meta Class
- 每一个 Meta Class 的 isa 指针都指向最上层的 Meta Class（NSObject 的 Meta Class）
- 最上层的 Meta Class 的 isa 指针指向自己，形成一个回路
- 每一个 Meta Class 的 super class 指针指向它原本 Class 的 Super Class 的 Meta Class。但是最上层的 Meta Class 的 Super Class 指向 NSObject Class 本身
- 最上层的 NSObject Class 的 super class 指向 nil

对于 NSObject 继承体系来说，其实例方法对体系中的所有实例、类和 meta-class 都是有效的；而类方法对于体系内的所有类和 meta-class 都是有效的。

注意：在一个类对象调用 class 方法是无法获取 meta-class，它只是返回类而已。

1.3.1 示例代码

@interface Sark : NSObject @end	@implementation Sark @end

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        BOOL res1 = [(id)[NSObject class] isKindOfClass:[NSObject class]];
        BOOL res2 = [(id)[NSObject class] isMemberOfClass:[NSObject class]];
        BOOL res3 = [(id)[Sark class] isKindOfClass:[Sark class]];
        BOOL res4 = [(id)[Sark class] isMemberOfClass:[Sark class]];
        NSLog(@"%d %d %d %d", res1, res2, res3, res4);
    }
```

```
    return 0;
}
```

- (BOOL)isKindOf:aClass {
  Class cls;
  for (cls = isa; cls; cls = cls->superclass)
  if (cls == (Class)aClass)
  return YES;
  return NO;
}
	- (BOOL)isMemberOf:aClass
	{
	  return isa == (Class)aClass;
	}

结论分析（运行结果是：1 0 0 0）：

- res1：当 NSObject Class 对象第一次进行比较时，得到它的 isa 为 NSObject 的 Meta Class，此时 NSObject Meta Class != NSObject Class。
  然后取 NSObject 的 Meta Class 的 Super class（即 NSObject Class），返回相等。
- res2：当前的 isa 指向 NSObject 的 Meta Class，和 NSObject Class 不相等。
- res3&res4：Sark Class 的 isa 指向的是 Sark 的 Meta Class，和 Sark Class 不相等；
  Sark Meta Class 的 super class 指向的是 NSObject Meta Class，和 Sark Class 不相等；
  NSObject Meta Class 的 super class 指向 NSObject Class，和 Sark Class 不相等；
  NSObject Class 的 super class 指向 nil，和 Sark Class 不相等。

1.4 Self & Super

self 是类的隐藏参数，指向当前调用方法的这个类的实例。super 是一个 Magic Keyword，本质是一个编译器标示符，和 self 是指向的同一个消息接受者。不同的是，super 告诉编译器，调用 class 这个方法时，要去父类的方法，而不是本类里的。最后在 NSObject 类中发现这个方法。而 - (Class)class 的实现就是返回 self 的类别，objc Runtime 开源代码对- (Class)class 方法的实现:

```
- (Class)class {
    return object_getClass(self);
}
```

下例中调用[self class]或[super class]，接受消息的对象都是 Son *xxx 这个对象。输出均为 Son：

```
@implementation Son : Father
- (id)init {
    self = [super init];
    if (self) {
        NSLog(@"%@", NSStringFromClass([self class]));
        NSLog(@"%@", NSStringFromClass([super class]));
    }
    return self;
}
@end
```

1.5 Reference

http://www.cocoachina.com/ios/20141031/10105.html (南峰子的技术博客)

2 Runtime-方法与消息

RunTime 简称运行时。就是系统在运行的时候的一些机制，其中最主要的是消息机制。对于 C 语言，函数的调用在编译的时候会决定调用哪个函数。编译完成之后直接顺序执行，无任何二义性。OC 的函数调用成为消息发送。属于动态调用过程。在编译的时候并不能决定真正调用哪个函数（事实证明，在编译阶段，OC 可以调用任何函数，即使这个函数并未实现，只要申明过就不会报错。而 C 语言在编译阶段就会报错）。只有在真正运行的时候才会根据函数的名称找到对应的函数来调用。

2.1 消息机制

2.1.1 为何iOS中无方法重载

@selector (function name)是一个 SEL 方法选择器。SEL 的主要作用是快速的通过方法名字（function name）查找到对应方法的函数指针，然后调用其函数。SEL 其本身是一个 Int 类型的一个地址，地址中存放着方法的名字。在一个类中。方法与 SEL 一一对应。所以 iOS 类中不能存在 2 个名称相同的方法，即使参数类型不同，因为 SEL 是根据方法名字生成的，相同的方法名称只能对应一个 SEL。

2.1.2 消息调用流程

以[obj makeText];为例（obj 是类，makeText 是方法）：

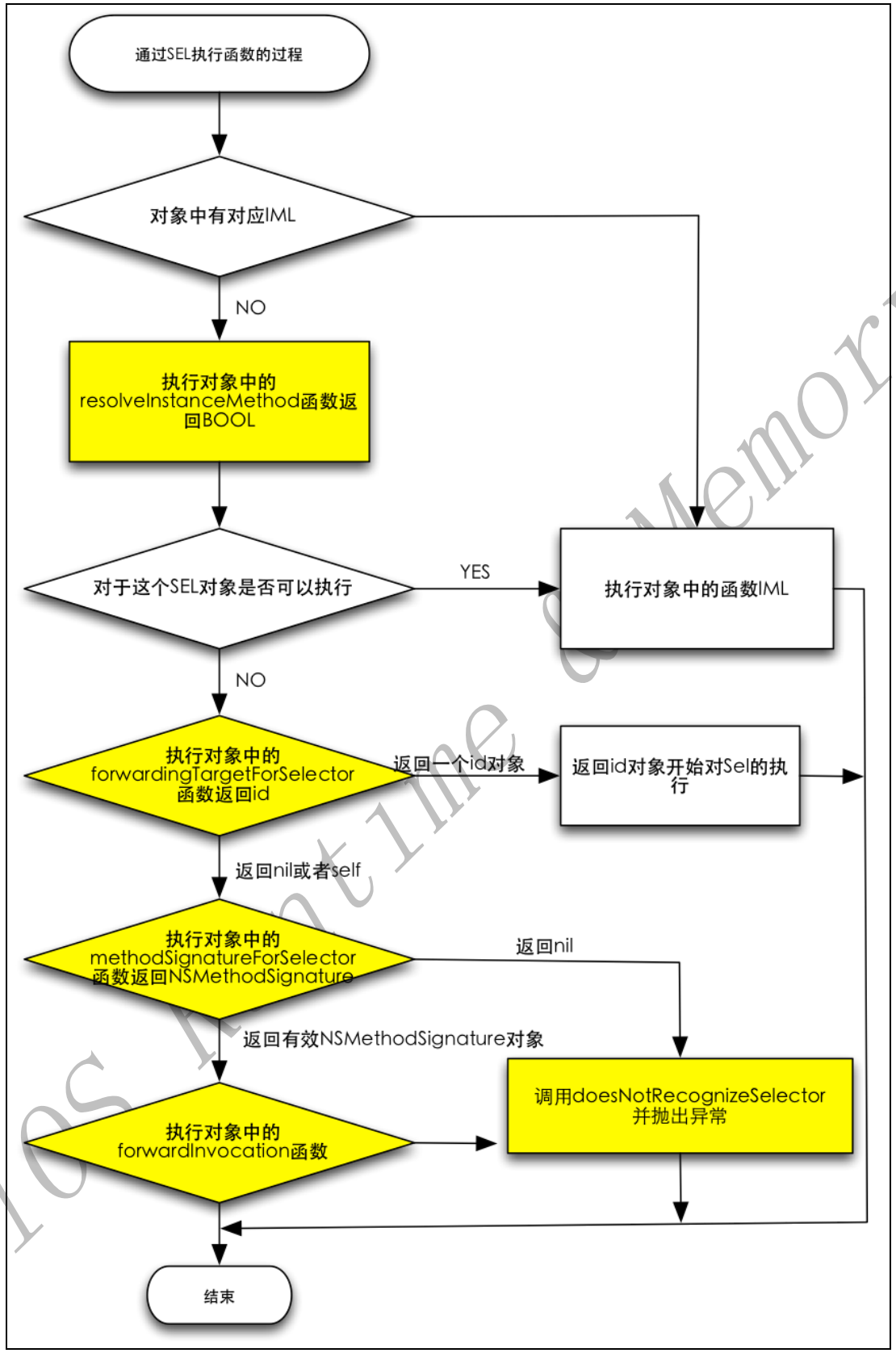
A. 编译器将代码[obj makeText]; 转化为 objc_msgSend(obj, @selector (makeText)); 该方法完成动态绑定的所有事情（B-D）。
B. 在 objc_msgSend 函数中：检查 selector 是否需要忽略。例如：Mac 开发中开启 GC 就会忽略 retain, release 方法。
C. 在 objc_msgSend 函数中：检查 target 是否为 nil。如果为 nil，直接 cleanup，然后 return。因此可以向 nil 发送消息。
D. 在 objc_msgSend 函数中：通过 obj 的 isa 指针找到 obj 对应的 class。
E. 在 Class 中先去 cache 中：通过 SEL 查找对应函数 method（猜测 cache 中 method 列表是以 SEL 为 key 通过 hash 表来存储的，这样能提高函数查找速度）。
F. 若 cache 中未找到。再去 methodList 中查找，若 methodlist 中未找到，则取 superClass 中查找(直到 NSObject)。
G. 若能找到，则将 method 加入到 cache 中，以方便下次查找，并通过 method 中的函数指针跳转到对应的函数(IMP)中去执行。

objc.h 文件中 SEL 的定义: typedef struct objc_selector *SEL;

IMP 的定义：typedef id (*IMP)(id, SEL, ...); 本质是一个函数指针，它包含接收消息的对象 id，调用方法的 SEL，以及一些方法参数，并返回一个 id。因此可以通过 SEL 获得它所对应的 IMP，即可取得需要执行方法的代码入口，就可以像普通的 C 语言函数调用一样使用这个函数指针。

2.1.3 消息解读流程

对象收到无法解读的消息时会导致程序崩溃，此部分将会解析此种情况。下图为每个函数调用的先后以及执行的前提的基本情况:



+ (BOOL)resolveInstanceMethod:(SEL)sel;

可在该方法中利用 class_addMethod 为类添加函数。如果实现了添加函数代码则返回 YES，未实现返回 NO。示例：新建 ViewController，在 viewDidLoad 方法中调用 runtimeTest:方法，但不实现该方法。程序在运行时，会因找不到 runtimeTest 方法而崩溃。如果在 ViewController

进行如下处理可避免报错：

```
+(BOOL)resolveInstanceMethod:(SEL)sel
{
    class_addMethod([self class], sel, (IMP)methodNotImplemented, "v@:");
    return [super resolveInstanceMethod:sel];
}
void methodNotImplemented (id self, SEL _cmd)
{
    NSString * method = NSStringFromSelector(_cmd);
    if (_cmd == @selector(runtimeTest:))
    {
        NSLog(@"[%@ %@]: method not implemented", [self class], method);
    }
}
```

重写父类的方法，并没有覆盖掉父类的方法，只是在当前类对象中找到了这个方法后就不会再去父类中找了。

如果想调用已经重写过的方法的父类的实现，只需使用 super 这个编译器标识，它会在运行时跳过在当前的类对象中寻找方法的过程。

2.2 Method

```
typedef struct objc_method *Method;
struct objc_method {    //定义中对 SEL 和 IMP 进行了绑定
    SEL method_name         OBJC2_UNAVAILABLE;    // 方法名
    char *method_types      OBJC2_UNAVAILABLE;
    IMP method_imp          OBJC2_UNAVAILABLE;    // 方法实现
}
```

2.2.1 操作方法

[1]. 调用指定方法的实现

```
id method_invoke ( id receiver, Method m, ... );
```

返回的是实际实现的返回值。参数 receiver 不能为空。这个方法的效率会比 method_getImplementation 和 method_getName 更快。

[2]. 调用返回一个数据结构的方法的实现

```
void method_invoke_stret ( id receiver, Method m, ... );
```

[3]. 获取方法名

```
SEL method_getName ( Method m );
```

返回的一个 SEL。要获取方法名的 C 字符串，可用 sel_getName(method_getName(method))。

[4]. 返回方法的实现

```
IMP method_getImplementation ( Method m );
```

[5]. 获取描述方法参数和返回值类型的字符串

```
const char * method_getTypeEncoding ( Method m );
```

[6]. 获取方法的返回值类型的字符串

```
char * method_copyReturnType ( Method m );
```

[7]. 获取方法的指定位置参数的类型字符串

char * method_copyArgumentType (Method m, unsigned int index);

[8]. 通过引用返回方法的返回值类型字符串

void method_getReturnType (Method m, char *dst, size_t dst_len);

类型字符串会被拷贝到 dst 中

[9]. 返回方法的参数的个数

unsigned int method_getNumberOfArguments (Method m);

[10]. 通过引用返回方法指定位置参数的类型字符串

void method_getArgumentType (Method m, unsigned int index, char *dst, size_t dst_len);

[11]. 返回指定方法的方法描述结构体

struct objc_method_description * method_getDescription (Method m);

[12]. 设置方法的实现

IMP method_setImplementation (Method m, IMP imp);

注意：该函数返回值是方法之前的实现

[13]. 交换两个方法的实现

void method_exchangeImplementations (Method m1, Method m2);

2.3 方法选择器

2.3.1 调用方法

[1]. 返回给定选择器指定的方法的名称

const char * sel_getName (SEL sel);

[2]. 注册方法

SEL sel_registerName (const char *str);

在 Objective-C Runtime 系统中注册一个方法，将方法名映射到一个选择器，并返回这个选择器。

[3]. 获取方法 ID

SEL sel_getUid (const char *str);

[4]. 比较两个选择器

BOOL sel_isEqual (SEL lhs, SEL rhs);

2.3.2 获取方法地址

NSObject 类提供了 methodForSelector:方法，让我们可以获取到方法的指针，然后通过这个指针来调用实现代码。我们需要将 methodForSelector:返回的指针转换为合适的函数类型，函数参数和返回值都需要匹配上。

```
void (*setter)(id, SEL, BOOL);
int i;
setter = (void (*)(id, SEL, BOOL))[target
        methodForSelector:@selector(setFilled:)];
for ( i = 0 ; i < 1000 ; i++ )
        setter(targetList[i], @selector(setFilled:), YES);
```

注意：函数指针的前两个参数必须是 id 和 SEL

这种方式只适合于在类似于 for 循环这种情况下频繁调用同一方法，以提高性能的情况。

另外，methodForSelector:是由 Cocoa 运行时提供的；它不是 Objective-C 语言的特性。

2.4 Category

category_t 在 objc-runtime-new.h 中的定义：

struct category_t {
const char *name;
classref_t cls;
struct method_list_t *instanceMethods;
struct method_list_t *classMethods;
struct protocol_list_t *protocols;
struct property_list_t *instanceProperties;
};
	说明：
	1. name 是 class name, not category_name；
	2. cls 是要扩展的类对象，编译期间不会定义，在 Runtime 阶段通过 name 对应到对应的类对象；
	3. instanceProperties 含 Category 的全部 properties，可用 objc_setAssociatedObject 和 objc_getAssociatedObject 增加实例变量，不过这与一般的实例变量不一样。

Category 方法的加载：

A. 打开 objc 源码，找到 objc-os.mm，函数_objc_init 为 runtime 的加载入口，由 libSystem 调用，进行初始化操作。

B. 之后调用 objc-runtime-new.mm -> map_images 加载 map 到内存

C. 之后调用 objc-runtime-new.mm->_read_images 初始化内存中的 map，这个时候将会 load 所有的类，协议还有 Category。NSObject 的+load 方法就是这个时候调用的

Category 方法的加载代码如下：

```
// Discover categories.
for (EACH_HEADER) {
    category_t **catlist = _getObjc2CategoryList(hi, &count);
    for (i = 0; i < count; i++) {
        category_t *cat = catlist[i];
        Class cls = remapClass(cat->cls);
        if (!cls) {
            // Category's target class is missing (probably weak-linked).
            // Disavow any knowledge of this category.
            catlist[i] = nil;
            if (PrintConnecting) {
                _objc_inform("CLASS: IGNORING category \?\?\?(%s) %p with "
                             "missing weak-linked target class",
                             cat->name, cat);
            }
            continue;
        }
        // Process this category.  First, register the category with its target class.
        // Then, rebuild the class's method lists (etc) if  the class is realized.
        BOOL classExists = NO;
        if (cat->instanceMethods ||  cat->protocols  ||  cat->instanceProperties)
        {
```

```
            addUnattachedCategoryForClass(cat, cls, hi);
            if (cls->isRealized()) {
                remethodizeClass(cls);
                classExists = YES;
            }
            if (PrintConnecting) {
                _objc_inform("CLASS: found category -%s(%s) %s",
                             cls->nameForLogging(), cat->name,
                             classExists ? "on existing class" : "");
            }
        }
        if (cat->classMethods  ||  cat->protocols)
        {
            addUnattachedCategoryForClass(cat, cls->ISA(), hi);
            if (cls->ISA()->isRealized()) {
                remethodizeClass(cls->ISA());
            }
            if (PrintConnecting) {
                _objc_inform("CLASS: found category +%s(%s)",
                             cls->nameForLogging(), cat->name);
            }
        }
    }
}
```

1) 循环调用了 _getObjc2CategoryList 方法，这个方法的实现是:

GETSECT(_getObjc2CategoryList, category_t *, "__objc_catlist");

方法中最后一个参数__objc_catlist 就是编译器刚刚生成的 category 数组

2) load 完所有的 categories 之后，开始对 Category 进行处理。

实例方法被加入到当前的类对象中，类方法被加入到当前类的 Meta Class 中(cls->ISA)

Step 1. 调用 addUnattachedCategoryForClass 方法

Step 2. 调用 remethodizeClass 方法，并在其实现中调用 attachCategoryMethods

```
static void attachCategoryMethods(Class cls, category_list *cats, bool flushCaches)
{
    if (!cats) return;
    if (PrintReplacedMethods) printReplacements(cls, cats);
    bool isMeta = cls->isMetaClass();
    method_list_t **mlists = (method_list_t **)
        _malloc_internal(cats->count * sizeof(*mlists));
    // Count backwards through cats to get newest categories first
    int mcount = 0;
    int i = cats->count;
    BOOL fromBundle = NO;
```

```
    while (i--) {
        method_list_t *mlist = cat_method_list(cats->list[i].cat, isMeta);
        if (mlist) {
            mlists[mcount++] = mlist;
            fromBundle |= cats->list[i].fromBundle;
        }
    }
    attachMethodLists(cls, mlists, mcount, NO, fromBundle, flushCaches);
    _free_internal(mlists);
}
```

这里把一个类的 category_list 的所有方法取出来生成了 method list。这里是倒序添加的，也就是说，新生成的 category 的方法会先于旧的 category 的方法插入。

之后调用 attachMethodLists 将所有方法前序添加进类的 method list 中，如果原来类的方法列表是 a，b，Category 的方法列表是 c，d。那么插入之后的方法列表将会是 c，d，a，b。

看上面被编译器转换的代码，可以发现 Category 头文件被注释掉了，结合上面 category 的加载过程。这使得即使没有 import category 的头文件，都能够成功调用到 Category 方法。

runtime 加载完成后，Category 的原始信息在类结构中将不会存在。

```
@interface NSObject (Sark)
+ (void)foo;
@end
@implementation NSObject (Sark)
- (void)foo {    NSLog(@"IMP: -[NSObject(Sark) foo]");    }
@end
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        [NSObject foo];
        [[NSObject new] foo];
    }
    return 0;
}
```

解析：

A. objc runtime 加载完后，NSObject 的 Sark Category 被加载。而 NSObject 的 Sark Category 的头文件 + (void)foo 并没有实质参与到工作中，只是给编译器进行静态检查，所有我们编译上述代码会出现警告，提示我们没有实现 + (void)foo 方法。而在代码编译中，它已经被注释掉了。

B. 实际被加入到 Class 的 method list 的方法是 - (void)foo，它是实例方法，故加入到当前类对象 NSObject 的方法列表中，而非 NSObject Meta class 的方法列表。

C. 当执行 [NSObject foo]时，objc_msgSend 的过程如下:

objc_msgSend 第一参数为“(id)objc_getClass("NSObject")”，获得 NSObject Class 的对象。类方法在 Meta Class 的方法列表中找，在 load Category 方法时加入的是- (void)foo 实例方法，故不在 NSOBject Meta Class 的方法列表中。下一步查找 super class，NSObject Meta Class 的 super

class 是 NSObject 本身。故此时能找到- (void)foo 方法。正常输出结果。

D. 当执行[[NSObject new] foo]，objc_msgSend 的过程如下:

[NSObject new]生成一个 NSObject 对象；直接在该对象的类（NSObject）的方法列表里找；能够找到，正常输出结果。

2.5 Reference

http://blog.csdn.net/a19860903/article/details/44853841

http://www.cocoachina.com/ios/20141224/10740.html

http://www.cocoachina.com/ios/20160302/15494.html

http://www.cocoachina.com/ios/20141106/10150.html (南峰子的技术博客)

3 Runtime-类与对象操作函数

类的操作方法大部分是以 class 为前缀的，而对象的操作方法大部分是以 objc 或 object_为前缀。

3.1 类相关操作函数

runtime 提供的操作类的方法主要针对 objc_class 结构体中的各个字段。

3.1.1 基本方法

[1]. const char * class_getName (Class cls);

获取类的类名，如果传入的 cls 为 Nil，则返回一个空字符串。

[2]. Class class_getSuperclass (Class cls);

获取类的父类，当 cls 为 Nil 或者 cls 为根类时，返回 Nil。通常可用 NSObject 类的 superclass 方法来达到同样的目的。

[3]. BOOL class_isMetaClass (Class cls);

判断给定的 Class 是否是一个元类，如果是 cls 是元类，则返回 YES；如果否或者传入的 cls 为 Nil，则返回 NO。

[4]. size_t class_getInstanceSize (Class cls); 获取实例大小

[5]. Ivar class_getInstanceVariable (Class cls, const char *name);

获取类中指定名称实例成员变量的信息，返回指向包含 name 指定的成员变量信息的 objc_ivar 结构体的指针(Ivar)。

[6]. Ivar class_getClassVariable (Class cls, const char *name);

获取类成员变量的信息，目前没有找到关于 Objective-C 中类变量的信息，一般认为 Objective-C 不支持类变量。注意，返回的列表不包含父类的成员变量和属性。

3.1.2 Ivars与Property相关API

Objective-C不支持往已存在的类中添加实例变量，因此系统库提供的类、自定义的类，都无法动态添加成员变量。如果通过运行时来创建一个类，可以使用class_addIvar函数给它添加成员变量。并且此方法只能在objc_allocateClassPair函数与objc_registerClassPair之间调用。另外，这个类也不能是元类。成员变量的按字节最小对齐量是1<

[1]. BOOL class_addIvar (Class cls, const char *name, size_t size, uint8_t alignment, const char *types); 添加成员变量

[2]. Ivar * class_copyIvarList (Class cls, unsigned int *outCount);

获取整个成员变量列表，返回指向成员变量信息的数组（不含父类中声明的变量），数组中每个元素是指向该成员变量信息的objc_ivar结构体的指针。outCount指针返回数组的大小。必须使用free()来释放此数组。

[3]. objc_property_t class_getProperty (Class cls, const char *name); 获取指定的属性

[4]. objc_property_t * class_copyPropertyList (Class cls, unsigned int *outCount); 获取属性列表

[5]. BOOL class_addProperty (Class cls, const char *name, const objc_property_attribute_t *attributes, unsigned int attributeCount); 为类添加属性

[6]. void class_replaceProperty (Class cls, const char *name, const objc_property_attribute_t *attributes, unsigned int attributeCount); 替换类的属性

3.1.3 垃圾回收

在MAC OS X系统中，可以使用垃圾回收器。runtime提供了几个函数来确定一个对象的内存区域是否可以被垃圾回收器扫描，以处理strong/weak引用。

```
const uint8_t * class_getIvarLayout ( Class cls );
void class_setIvarLayout ( Class cls, const uint8_t *layout );
const uint8_t * class_getWeakIvarLayout ( Class cls );
void class_setWeakIvarLayout ( Class cls, const uint8_t *layout );
```

但通常情况下，我们不需要去主动调用这些方法；在调用objc_registerClassPair时，会生成合理的布局。

3.1.4 methodList相关API

用途	声明
添加方法。	BOOL class_addMethod (Class cls, SEL name, IMP imp, const char *types);
获取实例方法。	Method class_getInstanceMethod (Class cls, SEL name);
获取类方法。	Method class_getClassMethod (Class cls, SEL name);
获取实例大小	Method * class_copyMethodList (Class cls, unsigned int *outCount);
替代方法的实现。	IMP class_replaceMethod (Class cls, SEL name, IMP imp, const char *types);
返回方法的具体实现。	IMP class_getMethodImplementation (Class cls, SEL name); IMP class_getMethodImplementation_stret (Class cls, SEL name);
类实例是否响应指定的selector	BOOL class_respondsToSelector (Class cls, SEL sel);

class_addMethod的实现会覆盖父类的方法实现，但不会取代本类中已存在的实现，如果本类中包含一个同名的实现，则函数会返回NO。如果要修改已存在实现，可以使用method_setImplementation。一个Objective-C方法是一个简单的C函数，它至少包含两个参数—self和_cmd。所以，我们的实现函数(IMP参数指向的函数)至少需要两个参数，如下所示：

```
void myMethodIMP(id self, SEL _cmd) {
    // implementation ....
}
```

与成员变量不同，不管一个类是否已存在，都可以为类动态添加方法。参数types是一个描述传递给方法的参数类型的字符数组，这涉及类型编码。

class_getInstanceMethod、class_getClassMethod函数，与class_copyMethodList不同的是，这两个函数都会去搜索父类的实现。

class_copyMethodList函数，返回包含所有实例方法的数组，如果需要获取类方法，则可以使用class_copyMethodList(object_getClass(cls), &count)(类的实例方法定义在元类里面)。该列表不包含父类实现的方法。outCount参数返回方法的个数。在获取到列表后，我们需要使用free()方法来释放它。

class_replaceMethod函数，该函数的行为可以分为两种：如果类中不存在name指定的方法，则类似于class_addMethod函数一样会添加方法；如果类中已存在name指定的方法，则类似于

method_setImplementation一样替代原方法的实现。

class_getMethodImplementation函数，该函数在向类实例发送消息时会被调用，并返回一个指向方法实现函数的指针。这个函数会比method_getImplementation(class_getInstanceMethod(cls, name))更快。返回的函数指针可能是一个指向runtime内部的函数，而不一定是方法的实际实现。例如，如果类实例无法响应selector，则返回的函数指针将是运行时消息转发机制的一部分。

class_respondsToSelector函数，我们通常使用NSObject类的respondsToSelector:或instancesRespondToSelector:方法来达到相同目的。

3.1.5 objc_protocol_list相关API

[1]. BOOL class_addProtocol (Class cls, Protocol *protocol);

添加协议

[2]. BOOL class_conformsToProtocol (Class cls, Protocol *protocol);

返回类是否实现指定的协议，可以使用NSObject类的conformsToProtocol:方法来替代。

[3]. Protocol * class_copyProtocolList (Class cls, unsigned int *outCount);

返回类实现的协议列表，返回的是一个数组，在使用后我们需要使用free()手动释放。

3.1.6 其他API

版本相关的操作包含以下函数：

[1]. int class_getVersion (Class cls);

获取版本号

[2]. void class_setVersion (Class cls, int version);

设置版本号

[3]. 其它

runtime还提供了两个函数来供CoreFoundation的tool-free bridging使用，即：

Class objc_getFutureClass (const char *name);

void objc_setFutureClass (Class cls, const char *name);

通常我们不直接使用这两个函数。

3.1.7 使用示例

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

void printIvarList(Class cls) {
    printf("Current class is %s:\n", [NSStringFromClass(cls) UTF8String]);
    unsigned int varCount;
    Ivar *vars = class_copyIvarList(cls, &varCount);
    for(int ind=0; ind<varCount; ind++) {
        const char * name = ivar_getName(vars[ind]);
        const char * type = ivar_getTypeEncoding(vars[ind]);
        printf("var[%d]=%s\t\tType=%s\n", ind, name, type);
    }
    free(vars);
}

void printPropertyList(Class cls) {
```

```
    printf("Current class is %s:\n", [NSStringFromClass(cls) UTF8String]);
    unsigned int proCount;
    objc_property_t * properties = class_copyPropertyList(cls, &proCount);
    for(int ind=0; ind<proCount; ind++) {
        const char * proName = property_getName(properties[ind]);
        const char * proAttr = property_getAttributes(properties[ind]);
        printf("property[%d]=%s\t\tAttribute=%s\n", ind, proName, proAttr);
    }
    free(properties);
}

void printMethodList(Class cls) {
    printf("Current class is %s:\n", [NSStringFromClass(cls) UTF8String]);
    unsigned int methodCount;
    Method * methods = class_copyMethodList(cls, &methodCount);
    for (int ind=0; ind<methodCount; ind++) {
        SEL sel = method_getName(methods[ind]);
        const char *sel_name = sel_getName(sel);
        printf("method[%d]=%s\n", ind, sel_name);
    }
    free(methods);
}
```

3.2 动态创建类和对象

3.2.1 动态创建类

[1]. 创建一个新类和元类

Class objc_allocateClassPair (Class superclass, const char *name, size_t extraBytes);

如果要创建一个根类，则superclass指定为Nil。extraBytes通常指定为0，该参数是分配给类和元类对象尾部的索引ivars的字节数。

[2]. 在应用中注册由objc_allocateClassPair创建的类

void objc_registerClassPair (Class cls);

[3]. 销毁一个类及其相关联的类

void objc_disposeClassPair (Class cls);

如果程序运行中还存在类或其子类的实例，则不能调用针对类调用该方法。

调用objc_allocateClassPair创建一个类，然后使用class_addMethod，class_addIvar等函数为新创建的类添加方法、实例变量和属性等。完成这些后，调用objc_registerClassPair函数来注册类，之后在程序中使用这个新类。

[4]. 示例：

```
Class cls = objc_allocateClassPair(MyClass.class, "MySubClass", 0);
class_addMethod(cls, @selector(submethod1), (IMP)imp_submethod1, "v@:");
class_replaceMethod(cls, @selector(method1), (IMP)imp_submethod1, "v@:");
class_addIvar(cls, "_ivar1", sizeof(NSString *), log(sizeof(NSString *)), "i");
```

```
objc_property_attribute_t type = {"T", "@\"NSString\""};
objc_property_attribute_t ownership = { "C", "" };
objc_property_attribute_t backingivar = { "V", "_ivar1"};
objc_property_attribute_t attrs[] = {type, ownership, backingivar};

class_addProperty(cls, "property2", attrs, 3);
objc_registerClassPair(cls);

id instance = [[cls alloc] init];
[instance performSelector:@selector(submethod1)];
[instance performSelector:@selector(method1)];
```

3.2.2 动态创建对象

[1]. 创建类实例

```
id class_createInstance ( Class cls, size_t extraBytes );
```

创建实例时，会在默认的内存区域为类分配内存。extraBytes参数表示分配的额外字节数。这些额外的字节可用于存储在类定义中所定义的实例变量之外的实例变量。该函数在ARC环境下无法使用。

该函数效果与+alloc方法类似。不过在使用时，需要确切的知道要用它来做什么。在下面的例子中，用NSString来测试一下该函数的实际效果：

```
id theObject = class_createInstance(NSString.class, sizeof(unsigned));
id str1 = [theObject init];
NSLog(@"%@", [str1 class]);
id str2 = [[NSString alloc] initWithString:@"test"];
NSLog(@"%@", [str2 class]);
```

输出结果是：

```
2014-10-23 12:46:50.781 RuntimeTest[4039:89088] NSString
2014-10-23 12:46:50.781 RuntimeTest[4039:89088] __NSCFConstantString
```

可以看到，使用class_createInstance函数获取的是NSString实例，而不是类簇中的默认占位符类__NSCFConstantString。

[2]. 在指定位置(bytes)创建类实例

```
id objc_constructInstance ( Class cls, void *bytes );
```

[3]. 销毁类实例，但不会释放并移除任何与其相关的引用。

```
void * objc_destructInstance ( id obj );
```

3.3 实例操作函数

3.3.1 针对整个对象

[1]. id object_copy (id obj, size_t size);

返回指定对象的一份拷贝

[2]. id object_dispose (id obj);

释放指定对象占用的内存

示例场景：假设有类A (父类) 和类B (子类)。类B通过添加一些额外的属性来扩展类A。现在创建一个A类的实例对象，并希望在运行时将这个对象转换为B类的实例对象，进而可以添

加数据到B类的属性中。这种情况下，无法直接转换，因为B类的实例比A类的大，没有足够的空间来放置对象。此时，可以使用上述函数，代码如下：

```
NSObject *a = [[NSObject alloc] init];
id newB = object_copy(a, class_getInstanceSize(MyClass.class));
object_setClass(newB, MyClass.class);
object_dispose(a);
```

3.3.2 针对对象实例变量

[1]. Ivar object_setInstanceVariable (id obj, const char *name, void *value);
修改类实例的实例变量的值

[2]. Ivar object_getInstanceVariable (id obj, const char *name, void **outValue);
获取对象实例变量的值

[3]. void * object_getIndexedIvars (id obj);
返回指向给定对象分配的任何额外字节的指针

[4]. id object_getIvar (id obj, Ivar ivar);
返回对象中实例变量的值

[5]. void object_setIvar (id obj, Ivar ivar, id value);
设置对象中实例变量的值

如果实例变量的Ivar已经知道，那么调用object_getIvar会比object_getInstanceVariable函数快，相同情况下，object_setIvar也比object_setInstanceVariable快。

3.3.3 针对对象的类

[1]. const char * object_getClassName (id obj);
返回给定对象的类名

[2]. Class object_getClass (id obj);
返回对象的类

[3]. Class object_setClass (id obj, Class cls);
设置对象的类

3.3.4 获取类定义

Objective-C动态运行库会自动注册代码中定义的所有的类。也可以在运行时创建类定义并使用objc_addClass函数来注册它们。获取类定义信息runtime函数主要包括：

[1]. 获取已注册的类定义的列表

int objc_getClassList (Class *buffer, int bufferCount); 不能假设从该函数中获取的类对象是继承自NSObject体系的，所以在这些类上调用方法时，都应该先检测一下这个方法是否在这个类中实现。示例代码如下：

```
int numClasses;
Class * classes = NULL;
numClasses = objc_getClassList(NULL, 0);
if (numClasses > 0) {
    classes = malloc(sizeof(Class) * numClasses);
    numClasses = objc_getClassList(classes, numClasses);
    NSLog(@"number of classes: %d", numClasses);
    for (int i = 0; i < numClasses; i++) {
```

```
            Class cls = classes[i];
            NSLog(@"class name: %s", class_getName(cls));
        }
        free(classes);
    }
```

[2]. 创建并返回一个指向所有已注册类的指针列表

```
Class * objc_copyClassList ( unsigned int *outCount );
```

[3]. 返回指定类的类定义

```
Class objc_lookUpClass ( const char *name );
Class objc_getClass ( const char *name );
Class objc_getRequiredClass ( const char *name );
```

如果类在运行时未注册，则objc_lookUpClass会返回nil，而objc_getClass会调用类处理回调，并再次确认类是否注册，如果确认未注册，再返回nil。而objc_getRequiredClass函数的操作与objc_getClass相同，只不过如果没有找到类，则会杀死进程。

[4]. 返回指定类的元类

```
Class objc_getMetaClass ( const char *name );
```

如果指定的类没有注册，则该函数会调用类处理回调，并再次确认类是否注册，如果确认未注册，再返回nil。不过，每个类定义都必须有一个有效的元类定义，所以这个函数总是会返回一个元类定义，不管它是否有效。

关联对象

3.4 Reference

http://www.cocoachina.com/ios/20150901/13173.html

http://www.cocoachina.com/ios/20141031/10105.html (南峰子的技术博客)

4 Runtime-成员变量与属性

类的操作方法大部分是以 class 为前缀的，而对象的操作方法大部分是以 objc 或 object_为前缀。

4.1 类型编码

作为对Runtime的补充，编译器将每个方法的返回值和参数类型编码为一个字符串，并将其与方法的selector关联在一起。这种编码方案在其它情况下也是非常有用的，因此我们可以使用@encode编译器指令来获取它。当给定一个类型时，@encode返回这个类型的字符串编码。这些类型可以是诸如int、指针这样的基本类型，也可以是结构体、类等类型。事实上，任何可以作为sizeof()操作参数的类型都可以用于@encode()。

关于Objective-C Runtime Programming Guide中的所有Type Encoding参见(https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtTypeEncodings.html。需要注意的是这些类型很多是与我们用于存档和分发的编码类型是相同的。但有一些不能在存档时使用。

注：Objective-C不支持long double类型。@encode(long double)返回d，与double是一样的。

一个数组的类型编码位于方括号中；其中包含数组元素的个数及元素类型。如以下示例：

```
float a[] = {1.0, 2.0, 3.0};
NSLog(@"array encoding type: %s", @encode(typeof(a)));
```

输出为: array encoding type: [3f]

还有些编码类型，@encode虽然不会直接返回它们，但它们可以作为协议中声明的方法的类型限定符。可以参考Type Encoding。

对于属性而言，还会有一些特殊的类型编码，以表明属性是只读、拷贝、retain等等，详情可以参考Property Type String，参考链接如下：
https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtPropertyIntrospection.html

4.2 成员变量-Ivar

objc_class 结构体中存储着 objc_ivar 数组列表，而 objc_ivar 结构体存储了类的单个成员变量的信息。

Ivar 在 objc 中被定义为：typedef struct objc_ivar *Ivar;

```
struct objc_ivar {
    char *ivar_name;
    char *ivar_type;
    int ivar_offset;     //基地址偏移字节
#ifdef __LP64__
    int space;
#endif
}
```

编译器在编译类时，生成一个 ivar 布局，显示在类中从哪可以访问 ivars 。看下图:

NSObject	
0	Class isa

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

上图中，左侧数据为 ivar_offset，对 ivar 的访问就可以通过“对象地址+ivar_offset” 完成。如果增加父类的 ivar，会导致布局出错，需要重新编译子类来恢复兼容性。看下图:

NSObject	
0	Class isa
4	NSArray secretAry
8	NSImage secretImg

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

在 Objective-C Runtime 中使用 Non Fragile ivars 处理此问题，Runtime 会进行检测来调整类中新增的 ivar_offset，通过“对象地址＋基类大小＋ivar_offset”计算出 ivar 地址，并访问到相应的 ivar。如下图:

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

4.2.1 操作方法

[1]. 获取成员变量名称

```
const char * ivar_getName ( Ivar v );
```

[2]. 获取成员变量类型编码

```
const char * ivar_getTypeEncoding ( Ivar v );
```

[3]. 获取成员变量的偏移量

```
ptrdiff_t ivar_getOffset ( Ivar v );
```

4.2.2 案例分析

```
@interface Student : NSObject
{
    @private
    int age;
}
@end
```

```
@implementation Student
- (NSString *)description
{
    NSLog(@"current pointer = %p", self);
    NSLog(@"age pointer = %p", &age);
    return [NSString stringWithFormat:@"age = %d", age];
}
```

	@end

```
Student *student = [[Student alloc] init];
//student->age = 24;  //此时会报错，无法通过->访问私有变量
Ivar age_ivar = class_getInstanceVariable(object_getClass(student), "age");
int *age_pointer = (int *)((__bridge void *)(student) + ivar_getOffset(age_ivar));
NSLog(@"age ivar offset = %td", ivar_getOffset(age_ivar));
*age_pointer = 10;  //利用似有变量指针，设置私有变量值
NSLog(@"%@", student);  //会调用- (NSString *)description;方法
```

```
@interface Sark : NSObject
@property (nonatomic, copy) NSString *name;
@end
@implementation Sark
- (void)speak {
    NSLog(@"my name is %@", self.name);
}
@end
@interface Test : NSObject
@end
@implementation Test
- (instancetype)init {
    self = [super init];
    if (self) {
        id cls = [Sark class];
        void *obj = &cls;
        [(__bridge id)obj speak];
    }
    return self;
}
@end
调用：[[Test alloc] init];
代码正常输出，输出结果为：my name is
```

解析：

1.为何能正常运行并调用 speak

obj 被转换成了一个指向 Sark Class 的指针，然后使用 id 转换成了 objc_object 类型。这个时候的 obj 已经相当于一个 Sark 的实例对象(但是和使用[Sark new]生成的对象还是不一样的)，因为 objc_object 结构体的构成就是一个指向 Class 的 isa 指针。

根据 objc_msgSend 的工作流程，在代码中的 obj 指向的 Sark Class 中能够找到 speak 方法，所以代码能够正常运行。

2.为何 self.name 输出为空

Sark 中 Propertyname 最终被转换成了 Ivar 加入到了类的结构中，Runtime 通过计算成员变量的地址偏移来寻找最终 Ivar 的地址，Sark 的对象指针地址加上 Ivar 的偏移量之后刚好指向的是 Test 对象指针地址。

可以用下面代码来验证上述解析：

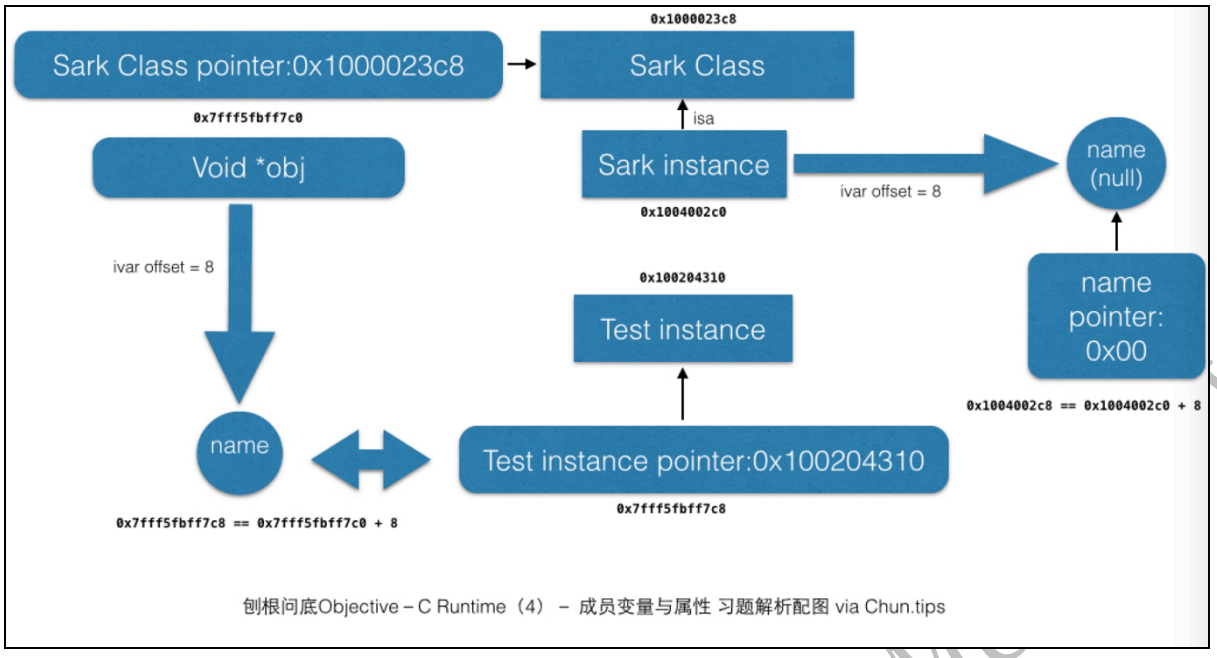
```
- (void)speak
{
    unsigned int numberOfIvars = 0;
    Ivar *ivars = class_copyIvarList([self class], &numberOfIvars);
    for(const Ivar *p = ivars; p < ivars+numberOfIvars; p++) {
        Ivar const ivar = *p;
        ptrdiff_t offset = ivar_getOffset(ivar);
        const char *name = ivar_getName(ivar);
        NSLog(@"Sark ivar name = %s, offset = %td", name, offset);
    }
```

```
        NSLog(@"my name is %p", &_name);
        NSLog(@"my name is %@", *(&_name));
    }
    @implementation Test
    - (instancetype)init
    {
        self = [super init];
        if (self) {
            NSLog(@"Test instance = %@", self);
            void *self2 = (__bridge void *)self;
            NSLog(@"Test instance pointer = %p", &self2);
            id cls = [Sark class];
            NSLog(@"Class instance address = %p", cls);
            void *obj = &cls;
            NSLog(@"Void *obj = %@", obj);
            [(__bridge id)obj speak];
        }
        return self;
    }
    @end
```

输出结果如下：

```
    Test instance =
    Test instance pointer = 0x7fff5fbff7c8
    Class instance address = 0x1000023c8
    Void *obj =
    Sark ivar name = _name, offset = 8
    my name is 0x7fff5fbff7c8
    my name is
```

在 C 中，局部变量是存储到内存的栈区，程序运行时栈的生长规律是从地址高到地址低。C 语言到头来讲是一个顺序运行的语言，随着程序运行，栈中的地址依次往下走。




```
@interface Father : NSObject
@end
@implementation Father
@end
@implementation Test
- (instancetype)init
{
    self = [super init];
    if (self) {
        NSLog(@"Test instance = %@", self);
        id fatherCls = [Father class];
        void *father;
        father = (void *)&fatherCls;
        id cls = [Sark class];
        void *obj;
        obj = (void *)&cls;
        [(__bridge id)obj speak];
    }
    return self;
}
@end
```

代码输出结果为：

```
Test instance =
ivar name = _name, offset = 8
Sark instance = 0x7fff5fbff7b8
my name is 0x7fff5fbff7c0
my name is
```

4.3 私有变量的访问

私有变量访问的 2 种方式：KVC 或 runtime 机制。

4.3.1 KVC访问

私有变量声明：	调用示例：
@interface Father () @property (nonatomic, copy) NSString *name; @end	Father *obj = [[Father alloc] init]; [obj setValue:@"test" forKey:@"name"]; NSLog(@"%@", [obj valueForKey:@"name"]);

4.3.2 Runtime机制访问

```
Father *father = [[Father alloc] init];
NSLog(@"before runtime:%@", [father description]);

unsigned int count = 0;
Ivar *members = class_copyIvarList([Father class], &count);
for (int i = 0 ; i < count; i++) {
    Ivar var = members[i];
    const char *memberName = ivar_getName(var);
    const char *memberType = ivar_getTypeEncoding(var);
    NSLog(@"%s----%s", memberName, memberType);
}
```

class_copyIvarList：获取类的所有属性变量，count 记录变量的数量 IVar 是 runtime 声明的一个宏，是实例变量的意思,instance variable，在 runtime 中定义为 typedef struct objc_ivar *Ivari

var_getName：将 IVar 变量转化为字符串

ivar_getTypeEncoding：获取 IVar 的类型

修改示例：

```
Ivar m_name = members[0];
object_setIvar(father, m_name, @"MYNAME");
NSLog(@"after runtime:%@", [father description]);
```

4.4 关联对象

4.4.1 操作方法

[1]. 设置关联对象

```
void objc_setAssociatedObject ( id object, const void *key, id value, objc_AssociationPolicy policy );
```

[2]. 获取关联对象

```
id objc_getAssociatedObject ( id object, const void *key );
```

[3]. 获取关联对象

```
void objc_removeAssociatedObjects ( id object );
```

4.4.2 使用示例

场景一：不能在分类中添加成员变量，可以用全局变量(会导致全局变量的滥用)，可以用关联对象的方式解决。

场景二：准备调用系统的类，并且需要向类中额外添加一个属性，此时若只为增加一个属性，继承一个类，偏麻烦，可以用关联属性方式。示例如下：

```
static char associatedObjectKey; //全局变量的地址作为关联对象的key
objc_setAssociatedObject(target,        //要设置的目标被关联对象
        &associatedObjectKey,           //关联对象的唯一的key
        @"新增的字符串属性",             //关联对象
         OBJC_ASSOCIATION_RETAIN_NONATOMIC); //关联策略
NSString *string = objc_getAssociatedObject(target, //获取谁的关联对象
                    &associatedObjectKey);  //获取关联对象的唯一的key
```

objc_AssociationPolicy关联策略：

```
enum {
    OBJC_ASSOCIATION_ASSIGN = 0,
    OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1,
    OBJC_ASSOCIATION_COPY_NONATOMIC = 3,
    OBJC_ASSOCIATION_RETAIN = 01401,
    OBJC_ASSOCIATION_COPY = 01403
};
```

可以使用objc_removeAssociatedObjects函数来移除一个关联对象，或者使用objc_setAssociatedObject函数将key指定的关联对象设置为nil。

使用示例：

```
- (void)setTapActionWithBlock:(void (^)(void))block
{
    UITapGestureRecognizer *gesture = objc_getAssociatedObject(self, &
kDTActionHandlerTapGestureKey kDTActionHandlerTapGestureKey);
    if (!gesture)
    {
        gesture = [[UITapGestureRecognizer alloc] initWithTarget:self
    action:@selector(__handleActionForTapGesture:)];
        [self addGestureRecognizer:gesture];
        objc_setAssociatedObject(self, &kDTActionHandlerTapGestureKey, gesture,
    OBJC_ASSOCIATION_RETAIN);
    }
    objc_setAssociatedObject(self, &kDTActionHandlerTapBlockKey, block,
OBJC_ASSOCIATION_COPY);
}
```

这段代码检测了手势识别的关联对象。如果没有，则创建并建立关联关系。同时，将传入的块对象连接到指定的key上。注意block对象的关联内存管理策略。

```
- (void)__handleActionForTapGesture:(UITapGestureRecognizer *)gesture
{
    if (gesture.state == UIGestureRecognizerStateRecognized)
    {
        void(^action)(void) = objc_getAssociatedObject(self,
    &kDTActionHandlerTapBlockKey);
```

```
            if (action)
            {
                action();
            }
        }
    }
```

4.5 属性

4.5.1 操作方法

[1]. 获取属性名称

```
const char * property_getName ( objc_property_t property );
```

[2]. 获取属性特性描述字符串

```
const char * property_getAttributes ( objc_property_t property );
```

[3]. 获取属性中指定的特性

```
char * property_copyAttributeValue ( objc_property_t property, const char *attributeName );
```

返回的char *在使用完后需要调用free()释放

[4]. 获取属性的特性列表

```
objc_property_attribute_t * property_copyAttributeList ( objc_property_t property, unsigned int *outCount );
```

返回的char *在使用完后需要调用free()释放

4.5.2 使用示例

场景：从服务端两个不同的接口获取相同的字典数据，但这两个接口是由两个人写的，相同的信息使用了不同的字段表示。我们在接收到数据时，可将这些数据保存在相同的对象中。对象类如下定义：

```
@interface MyObject: NSObject
    @property (nonatomic, copy) NSString * name;
    @property (nonatomic, copy) NSString * status;
@end
```

接口A、B返回的字典数据如下所示：

```
@{@"name1": "张三", @"status1": @"start"}
@{@"name2": "张三", @"status2": @"end"}
```

通常的方法是写两个方法分别做转换，不过如果能灵活地运用Runtime的话，可以只实现一个转换方法，为此，我们需要先定义一个映射字典(全局变量)

```
static NSMutableDictionary *map = nil;
@implementation MyObject
+ (void)load {
    map = [NSMutableDictionary dictionary];
    map[@"name1"] = @"name";
    map[@"status1"] = @"status";
    map[@"name2"] = @"name";
    map[@"status2"] = @"status";
}
```

@end

上面的代码将两个字典中不同的字段映射到MyObject中相同的属性上，转换方法做如下处理：

```
- (void)setDataWithDic:(NSDictionary *)dic {
    [dic enumerateKeysAndObjectsUsingBlock:^(NSString *key, id obj, BOOL *stop) {
        NSString *propertyKey = [self propertyForKey:key];
        if (propertyKey) {
            objc_property_t property = class_getProperty([self class], [propertyKey
UTF8String]);
            // TODO: 针对特殊数据类型做处理
            NSString *attributeString = [NSString
stringWithCString:property_getAttributes(property) encoding:NSUTF8StringEncoding];
            ...
            [self setValue:obj forKey:propertyKey];
        }
    }];
}
```

上述方式处理的前提是：该属性支持KVC。

4.6 Reference

http://www.cocoachina.com/ios/20141105/10134.html (南峰子的技术博客)

http://www.cnblogs.com/wengzilin/p/4344952.html?utm_source=tuicool

http://www.jianshu.com/p/009cfa6f5e93

5 Runtime-方法操作

5.1 私有函数的访问

```
unsigned int count = 0;
Method *memberFuncs = class_copyMethodList([Father class], &count);
for (int i = 0; i < count; i++) {
    SEL name = method_getName(memberFuncs[i]);
    NSString *methodName = [NSString stringWithCString:sel_getName(name)
encoding:NSUTF8StringEncoding];
    NSLog(@"member method:%@", methodName);
}
```

Method：runtime 声明的一个宏，表示一个方法，typedef struct objc_method *Method;

class_copyMethodList：获取所有方法，所有在.m 文件显式实现的方法都会被找到

method_getName：读取一个 Method 类型的变量，输出我们在上层中很熟悉的 SEL

添加新方法：

```
- (void)tryAddingFunction {
    class_addMethod([Father class], @selector(method::), (IMP)myAddingFunction, "i@:i@");
}
//具体的实现，即 IMP 所指向的方法
int myAddingFunction(id self, SEL _cmd, int var1, NSString *str) {
    NSLog(@"I am added funciton");
    return 10;
}
```

调用新方法：

```
[self tryAddingFunction];
Father *father = [[Father alloc] init];
[father method:10 :@"111"]; //输入 father 实例后无法获得 method 提示，只能靠手敲。对于
编译器给出的"-method" not found 警告，忽略即可
```

在其他类文件中实例化 Father 类，也能调用到-method 方法。尽管 MRC 下无法获得代码提示，但是仍可用[father method:xx :xx]方法！(在 ARC 下会报 no visible @interface 错误)

功能对调示例：

```
Method method1 = class_getInstanceMethod([NSString class], @selector(lowercaseString));
Method method2 = class_getInstanceMethod([NSString class], @selector(uppercaseString));
method_exchangeImplementations(method1, method2);
```

5.2 Method Swizzling

Method swizzling指的是改变一个已存在的选择器对应的实现的过程，它依赖于Objectvie-C中方法的调用能够在运行时进改变——通过改变类的调度表（dispatch table）中选择器到最终函数间的映射关系。

示例场景：跟踪在一个iOS应用中每个视图控制器展现给用户的次数：

实现一：在各视图控制器的viewWillAppear:方法中增加相应的跟踪代码。会产生大量重复代码。

实现二：子类化，需要将UIViewController、UITableViewController、UINavigationController以及所有其他视图控制器类都子类化，也会导致代码重复。

实现三：在分类中进行method swizzling。示例代码如下：

```
#import    <UIKit/UIKit.h>
@implementation UIViewController (Tracking)
+ (void)load {
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        Class class = [self class];
        // When swizzling a class method, use the following:
        // Class class = object_getClass((id)self);
        SEL originalSelector = @selector(viewWillAppear:);
        SEL swizzledSelector = @selector(xxx_viewWillAppear:);

        Method originalMethod = class_getInstanceMethod(class, originalSelector);
        Method swizzledMethod = class_getInstanceMethod(class, swizzledSelector);

        BOOL didAddMethod = class_addMethod(class, originalSelector,
                                            method_getImplementation(swizzledMethod),
                                            method_getTypeEncoding(swizzledMethod));
        if (didAddMethod) {
            class_replaceMethod(class, swizzledSelector,
                                method_getImplementation(originalMethod),
                                method_getTypeEncoding(originalMethod));
        } else {
            method_exchangeImplementations(originalMethod, swizzledMethod);
        }
    });
}

#pragma mark - Method Swizzling
- (void)xxx_viewWillAppear:(BOOL)animated {
    [self xxx_viewWillAppear:animated];
    NSLog(@"viewWillAppear: %@", self);
}
@end
```


5.3 Reference

http://www.cocoachina.com/ios/20140225/7880.html (南峰子的技术博客)
http://www.cnblogs.com/wengzilin/p/4344952.html?utm_source=tuicool
http://www.jianshu.com/p/009cfa6f5e93

6 属性 property

类的 Property 属性在编译时会被编译器转换成 Ivar，并自动生成 Setter 和 Getter 方法。在 objc_class 结构体中没有专门记录 Property 的 list。在 objc-runtime-new.h 中，可以发现：在 class_ro_t 结构体中使用 property_list_t 存储对应的 propertyies。

6.1 权限访问

6.1.1 readwrite

有getter和setter方法

6.1.2 readonly

只有getter方法，没有setter方法，不能和copy / retain / assign / strong / weak / unsafe_unretauined等修饰setter的属性组合使用。此时若用点操作符为属性赋值，会报错。

6.2 线程管理

atomic和nonatomic用来决定编译器生成的getter和setter是否为原子操作。在多线程环境下，原子操作是必要的，否则有可能引起错误的结果。加了atomic，setter函数会变成下面这样：

```
{lock} //非ARC
if (property != newValue) {
    [property release];
    property = [newValue retain];
}
{unlock}  //非ARC
```

6.2.1 atomic

默认属性，当一个变量声明为atomic时，在多线程中只能有一个线程能对它进行访问；该变量为线程安全型，但是会影响访问速度；在非ARC编译环境下，需要设置访问锁来保证对该变量进行正确的get/set

6.2.2 nonatomic

当一个变量声明为nonatomic时，多个线程可以同时对其进行访问，它是非线程安全型，访问速度快；当两个不同的线程对其访问时，容易失控。

6.3 MRC属性(ARC下仍能用)

6.3.1 assign

简单赋值，不更改索引计数。使用之后如果没有置为nil，可能就会产生野指针。

若用malloc分配一块内存，并将其地址赋值给指针a，又把a赋值给（assign）指针b，让二者共享该内存。当a不再需要这块内存时，不能直接释放它。因为a并不知道b是否还在使用这块内存，如果a释放了，b在使用这块内存的时候会引起程序crash掉。

应用场合：
基础数据类型（NSInterger、CGFloat等）、C数据类型（int、float等）、简单数据类型
ARC下等同于unsafe_unretained
	setter格式：
	-(void)setA:(int)a{
	_a=a;
	}

6.3.2 retain

与strong相对应，使用了引用计数，retain+1，	setter格式：

release -1，当引用计数为0时，dealloc会被调用，内存被释放。只能用于OC对象类型，不能用于基本数据类型和Core Foundation对象，因为二者没有引用计数。
ARC下等同于strong
	-(void)setA:(Car *)a{
	if(_a!=a){
	[_a release];
	_a=[a retain];
	}
	}

6.4 ARC属性

当strong类型的指针被释放掉之后，所有的指向同一个对象的weak指针都会被清零。strong好比放风筝的，对风筝具有控制权，weak好比看风筝的，只能看，指向风筝位置。

6.4.1 strong

ARC默认属性，与retain相对应

6.4.2 weak

与assign相对应，weak对象一旦不进行使用后，永远不会使用了，不会产生野指针。

6.4.3 unsafe_unretained

与assign相似。只是告诉ARC如何正确地调用声明为unsafe_unretained变量的retain和release

6.5 MRC&ARC公共属性

6.5.1 Copy

用于非共享内存时，每个指针有自己的内存空间。只对实行了NSCopying协议的对象类型有效。
ARC与MRC效用一样，ARC下同时具有strong效果。
	setter格式（MRC）：
	-(void)setA:(NSString *)a{
	if(_a!=a){
	[_a release];
	_a=[a copy];
	}
	}

6.6 property与dynamic和synthesize关系

在 Objective-C 中，使用@property ＋ @dynamic 关键字是告诉编译器由开发人员自己实现访问方法。使用@property ＋ @synthesize 是让编译器自动生成 getter/setter 方法。

6.7 setter与getter

setter ：为外部提供的修改内部属性的接口。

getter ：为外部提供的查看内部变量的接口。在内部写法：self.name 和_name。区别：self.name=@"test"是通过调用 setter 设置属性，a= self.name 是通过调用 getter 方法获取属性。_name 表示直接去访问成员变量。

6.8 copy与mutablecopy

深拷贝：对对象的所有属性、成员变量的拷贝。与原对象功能一样。

浅拷贝：多一个指针指向原对象，原对象引用计数＋1。只有不可变对象的不可变拷贝（copy）才是浅拷贝，其余都是深拷贝。

要实现拷贝就必须遵守 NSCopy 或者 NSMutableCopy 协议，并实现相关的方法。

6.9 成员变量、实例变量、属性之间的关系

成员变量形象的说就是写在花括中的变量。例如下面的 NSString *name 和 NSInteger age

但是 name 又是一个对象指针，又被称为实例变量，所以成员变量包含实例变量，成员变量中除了基本数据类型是都是实例变量；属性是通过@property 定义的变量，xcode 编译器会自动生成一个成员变量 NSString *_name

```
@interface HHPeerson : NSObject {
  NSString *name;
  NSInteger age;
}
@property (nonatomic,copy) NSString *name;
@end
```

属性和成员变量有三种修饰 @private、@public、@protect。默认是@protect，可以省略。

@private 表示类的私有内容，只允许类内和类的对象访问，其它类和他的子类不能访问；

@protect 表示只允许该类和该类的子类访问

@public 表示所有的对象都能访问

6.10 注意事项

有时系统不会 autosynthsis，就不会生成 ivar 变量(如 4.9 中的_name 变量)，使用 ivar 变量会报错，可能导致此情况的原因如下：

[1]. 同时重写了 setter 和 getter 时

[2]. 重写了只读属性的 getter 时

[3]. 使用了@dynamic 时

[4]. 在 @protocol 中定义的所有属性

[5]. 在 category 中定义的所有属性

[6]. 重载的属性

6.11 Reference

http://blog.csdn.net/yanxiaoqing/article/details/7402632

http://www.jianshu.com/p/28196578b1e9

http://blog.csdn.net/jasonjwl/article/details/49427377http://www.linuxidc.com/Linux/2014-03/97744.htm

http://blog.sina.com.cn/s/blog_6531b9b80101c6cr.html

7 内存管理

目前内存管理三种机制：引用计数器、属性参数、自动释放池。

7.1 引用计数器

7.1.1 内存管理原则

在 ObjC 中每个对象内部都有一个与之对应的整数（retainCount），叫“引用计数器”，当一个对象在创建之后它的引用计数器为 1，当调用这个对象的 alloc、retain、new、copy 方法之后引用计数器自动在原来的基础上加 1（ObjC 中调用一个对象的方法就是给这个对象发送一个消息），当调用这个对象的 release 方法之后它的引用计数器减 1，如果一个对象的引用计数器为 0，则系统会自动调用这个对象的 dealloc 方法来销毁这个对象。

可以通过 dealloc 方法来查看一个对象是否已经被回收，如果没有被回收则有可能造成内存泄露。如果一个对象被释放之后，那么最后引用它的变量我们手动设置为 nil，否则可能造成野指针错误，而且需要注意在 ObjC 中给空对象发送消息是不会引起错误的。

野指针错误在 Xcode 中的通常表现：Thread 1：EXC_BAD_ACCESS(code=EXC_I386_GPFLT)

7.1.2 内存释放原则

内存释放原则：谁创建，谁释放。

7.2 属性参数（参见property部分）

7.3 自动释放池

在 ObjC 中也有一种内存自动释放的机制叫做“自动引用计数”（或“自动释放池”），这是一种半自动的机制，有些操作还是需要手动设置。自动内存释放使用@autoreleasepool 关键字声明一个代码块，如果一个对象在初始化时调用了 autorelase 方法，那么当代码块执行完之后，在块中调用过 autorelease 方法的对象都会自动调用一次 release 方法。这样一来就起到了自动释放的作用，同时对象的销毁过程也得到了延迟（统一调用 release 方法）。

[1]. autorelease 方法不会改变对象的引用计数器，只是将这个对象放到自动释放池中；

[2]. 自动释放池实质是当自动释放池销毁后调用对象的 release 方法，不一定就能销毁对象（例如如果一个对象的引用计数器>1 则此时就无法销毁）；

[3]. 由于自动释放池最后统一销毁对象，因此如果一个操作比较占用内存（对象比较多或者对象占用资源比较多），最好不要放到自动释放池或者考虑放到多个自动释放池；

[4]. ObjC 中类库中的静态方法一般都不需要手动释放，内部已经调用了 autorelease 方法；

7.4 面试题

7.4.1 block一般用那个关键字修饰

block一般使用copy关键之进行修饰，block使用copy是从MRC遗留下来的“传统”，在MRC中，方法内容的block是在栈区的，使用copy可以把它放到堆区。但在ARC中写不写都行：编译器自动对block进行了copy操作。

7.4.2 @property声明变量与copy

用@property声明的NSString（或NSArray，NSDictionary）经常使用copy关键字，为什么？如果改用strong关键字，可能造成什么问题？

用@property声明 NSString、NSArray、NSDictionary经常使用copy关键字，是因为他们有对应的可变类型：NSMutableString、NSMutableArray、NSMutableDictionary，他们之间可能进行赋值操作，为确保对象中的字符串值不会无意间变动，应该在设置新属性值时拷贝一份。

如果使用是strong，则该属性就有可能指向一个可变对象，如果这个可变对象在外部被修改了，那么会影响该属性。

copy此特质所表达的所属关系与strong类似。然而设置方法并不保留新值，而是将其“拷贝”(copy)。当属性类型为NSString时，经常用此特质来保护其封装性，因为传递给设置方法的新值有可能指向一个NSMutableString类的实例。这个类是NSString的子类，表示一种可修改其值的字符串，此时若是不拷贝字符串，那么设置完属性之后，字符串的值就可能会在对象不知情的情况下遭人更改。所以，这时就要拷贝一份“不可变”(immutable)的字符串，确保对象中的字符串值不会无意间变动。只要实现属性所用的对象是“可变的”(mutable)，就应该在设置新属性值时拷贝一份。

7.4.3 ARC与MRC的理解

ARC 是 automatic reference counting 自动引用计数，在程序编译时自动加入 retain/release。在对象被创建时 retain count+1，在对象被 release 时 count-1，当 count=0 时，销毁对象。程序中加入 autoreleasepool 对象会由系统自动加上 autorelease 方法，如果该对象引用计数为 0，则销毁。那么 ARC 是为了解决 MRC 手动管理内存存在的一些而诞生的。

MRC 下内存管理的缺点：

[1].释放一个堆内存时，首先要确定指向这个堆空间的指针都被 release 了。(避免提前释放)

[2].释放指针指向的堆空间，首先要确定哪些指向同一个堆，这些指针只能释放一次。(避免释放多次，造成内存泄露)

[3].模块化操作时，对象可能被多个模块创建和使用，不能确定最后由谁释放

[4].多线程操作时，不确定哪个线程最后使用完毕。

ARC 可能出现内存泄露。如下面两种情况：

[1]. 循环参照：A 有个属性参照 B，B 有个属性参照 A，如果都是 strong 参照的话，两个对象都无法释放。

[2]. 死循环：如果有个 ViewController 中有无限循环，也会导致即使 ViewController 对应的 view 消失了，ViewController 也不能释放。

7.4.4 runloop、autorelease pool以及线程之间的关系

每个线程(包含主线程)都有一个 Runloop。系统会为每个 Runloop 隐式创建一个 Autorelease pool，所有的 release pool 会构成一个像 callstack 一样的一个栈式结构，在每一个 Runloop 结束时，当前栈顶的 Autorelease pool 会被销毁，这样这个 pool 里的每个 Object 会被 release。

7.4.5 @property与setter和getter

“属性”(property)有两大概念：ivar(实例变量)、存取方法(access method=getter)，即@property = ivar + getter + setter。类完成属性的定以后，编译器会自动编写访问这些属性的方法(自动合成 autosynthesis)。

[1]. 写@property (nonatomic,retain)NSString *name 的 setter 方法

```
-(void)setName:(NSString *)name
{
```

```
    [name retain];
    [_name release];
    _name = name;
}
```

[2]. 写@property (nonatomic,copy) NSString *name 的 setter 方法

```
-(void)setName:(NSString *)name
{
    [_name release];
    _name = [name copy];
}
```

retain 属性的 setter 方法是保留新值并释放旧值，然后更新实例变量，令其指向新值。顺序很重要。假如还未保留新值就先把旧值释放了，而且两个值又指向同一个对象，先执行的 release 操作就可能导致系统将此对象永久回收。

7.4.6 assign / weak，_block / _weak的区别

assign 适用于基本数据类型，weak 是适用于 NSObject 对象，并且是一个弱引用。

assign 其实也可以用来修饰对象，那么为什么不用它呢？因为被 assign 修饰的对象在释放之后，指针的地址还是存在的，也就是说指针并没有被置为 nil。如果在后续内存分配中，刚才分到了这块地址，程序就会崩溃掉。而 weak 修饰的对象在释放之后，指针地址会被置为 nil。

_block 是用来修饰一个变量，这个变量就可以在 block 中被修改。

_block:使用_block 修饰的变量在 block 代码块中会被 retain(ARC 下，MRC 下不会 retain)

_weak:使用_weak 修饰的变量不会在 block 代码块中被 retain

7.4.7 代码纠错-1

修改前：

```
@autoreleasepool {
    for (int i=0; i<largeNumber; i++) {
        Person *per = [[Person alloc] init];
        [per autorelease];
    }
}
```

修改后：

```
@autoreleasepool {
    for (int i=0; i<largeNumber; i++) {
        @autoreleasepool {
            Person *per = [[Person alloc] init];
            [per autorelease];
        }
    }
}
```

内存管理的原则：如果对一个对象使用了 alloc、copy、retain，那么你必须使用相应的 release 或者 autorelease。咋一看，这道题目有 alloc，也有 autorelease，两者对应起来，应该没问题。但 autorelease 虽然会使引用计数减一，但是它并不是立即减一，它的本质功能只是把对象放到离他最近的自动释放池里。当自动释放池销毁了，才会向自动释放池中的每一个对象发送 release 消息。这道题的问题就在 autorelease。因为 largeNumber 是一个很大的数，autorelease 又不能使引用计数立即减一，所以在循环结束前会造成内存溢出的问题。在循环内部再加一个自动释放池，这样就能保证每创建一个对象就能及时释放。

7.4.8 代码纠错-2

代码：	解释：

@autoreleasepool {
NSString *str = [[NSString alloc] init];
[str retain];
[str retain];
str = @"jxl";
[str release];
[str release];
[str release];
}
	代码存在内存泄露问题，1.内存泄露 2.指向常量区的对象不能 release。
	指针变量 str 原本指向一块开辟的堆区空间，但是经过重新给 str 赋值，str 的指向发生了变化，由原来指向堆区空间，到指向常量区。常量区的变量根本不需要释放，这就导致了原来开辟的堆区空间没有释放，照成内存泄露。

7.4.9 weak与assign的使用

[1]. 在 ARC 中，在有可能出现循环引用的时候，可以让其中一端使用 weak。如 delegate 代理

[2]. 自身已经对它进行一次强引用，没有必要再强引用一次，会使用 weak，自定义控件属性一般也使用 weak。

不同点：

[1]. weak 此特质表明该属性定义了一种“非拥有关系”。为这种属性设置新值时，设置方法既不保留新值，也不释放旧值。此特性与 assign 一样，然而在属性所指的对象遭到推毁时，属性值也会清空。而 assign 的“设置方法”只会执行针对“纯量类型” (scalar type，例如 CGFloat 或 NSlnteger 等)的简单赋值操作。

[2]. assign 可以用非 OC 对象，而 weak 必须用于 OC 对象。

7.4.10 内存管理语义(assign、strong、weak等的区别)

[1]. assign “设置方法”只会执行针对“纯量”的简单赋值操作。

[2]. strong 此特质表明该属性定义了一种“拥有关系”。为这种属性设置新值时，设置方法会先保留新值，并释放旧值，然后再将新值设置上去。

[3]. weak 此特质表明该属性定义了一种“非拥有关系”。为这种属性设置新值时，设置方法既不保留新值，也不释放旧值。此特质同 assign 类似，然而在属性所指的对象遭到推毁时，属性值也会清空。

[4]. unsafe_unretained 此特质的语义和 assign 相同，但是它适用于“对象类型”，该特质表达一种“非拥有关系”，当目标对象遭到推毁时，属性值不会自动清空，这不同于 weak。

[5]. copy 此特质所表达的所属关系与 strong 类似。然而设置方法并不保留新值，而是设置方法并不保留新值，而是将其“拷贝”。当属性类型为 NSString*时，经常用此特质来保护其封装性，因为传递给设置方法的新值有可能指向一个 NSMutableString 类的实例。这个类是 NSString 的子类，表示一种可以修改其值的字符串，此时若是不拷贝字符串，那么设置完属性之后，字符串的值就可能会在对象不知情的情况下遭人更改。所以，这时就要拷贝一份“不可变”的字符串，确保对象中的字符串值不会无意间变动。只要实现属性所用的对象是“可变的”，就应该在设置新属性值时拷贝一份。

7.5 Reference

http://blog.csdn.net/fightingbull/article/details/8098133

http://www.cnblogs.com/kenshincui/p/3870325.html

http://www.cocoachina.com/ios/20150625/12234.html

(S) http://www.cocoachina.com/ios/20141031/10107.html