

Runtime介绍



作者：刘小壮

博客：<https://www.jianshu.com/u/2de707c93dc4>

Github：<https://github.com/DeveloperErenLiu>

描述

OC语言是一门动态语言，会将程序的一些决定工作从编译期推迟到运行期。由于OC语言运行时的特性，所以其不只需要依赖编译器，还需要依赖运行时环境。

OC语言在编译期都会被编译为C语言的 Runtime 代码，二进制执行过程中执行的都是C语言代码。而OC的类本质上都是结构体，在编译时都会以结构体的形式被编译到二进制中。 Runtime 是一套由C、C++、汇编实现的API，所有的方法调用都叫做发送消息。

根据 Apple 官方文档的描述，目前OC运行时分为两个版本， Modern 和 Legacy 。二者的区别在于 Legacy 在实例变量发生改变后，需要重新编译其子类。 Modern 在实例变量发生改变后，不需要重新编译其子类。

Runtime 不只是一些C语言的API，其由 Class 、 Meta Class 、 Instance、 Class Instance 组成，是一套完整的面向对象的数据结构。所以研究Runtime整体的对象模型，比研究API是怎么实现的更有意义。

使用Runtime

Runtime 是一个共享动态库，其目录位于 /usr/include/objc ，由一系列的C函数和结构体构成。和 Runtime 系统发生交互的方式有三种，一般都是用前两种：

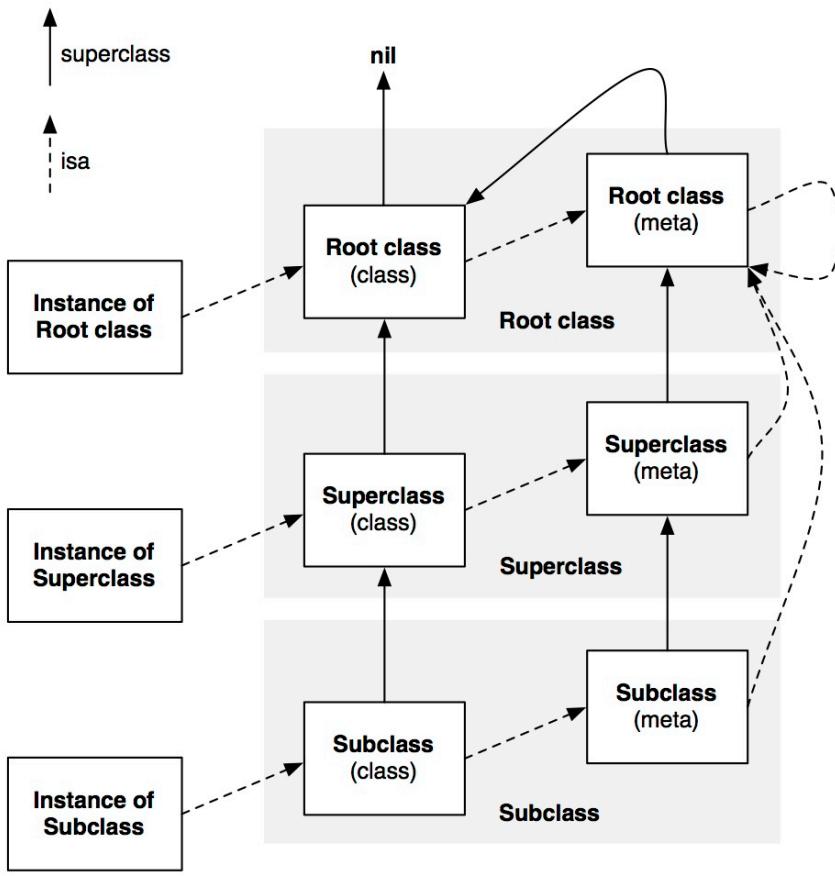
1. 使用OC源码 直接使用上层OC源码，底层会通过 Runtime 为其提供运行支持，上层不需要关心 Runtime 运行。
2. NSObject 在OC代码中绝大多数的类都是继承自 NSObject 的， NSProxy 类例外。 Runtime 在 NSObject 中定义了一些基础操作， NSObject 的子类也具备这些特性。
3. Runtime 动态库 上层的OC源码都是通过 Runtime 实现的，我们一般不直接使用 Runtime ，直接和OC代码打交道就可以。

使用 Runtime 需要引入下面两个头文件，一些基础方法都定义在这两个文件中。

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

对象模型

下面图中表示了对象间 isa 的关系，以及类的继承关系。



从 Runtime 源码可以看出，每个对象都是一个 `objc_object` 的结构体，在结构体中有一个 `isa` 指针，该指针指向自己所属的类，由 Runtime 负责创建对象。

类被定义为 `objc_class` 结构体，`objc_class` 结构体继承自 `objc_object`，所以类也是对象。在应用程序中，类对象只会被创建一份。在 `objc_class` 结构体中定义了对象的 `method list`、`protocol`、`ivar list` 等，表示对象的行为。

既然类是对象，那类对象也是其他类的实例。所以 Runtime 中设计出了 `meta class`，通过 `meta class` 来创建类对象，所以类对象的 `isa` 指向对应的 `meta class`。而 `meta class` 也是一个对象，所有元类的 `isa` 都指向其根元类，根元类的 `isa` 指针指向自己。通过这种设计，`isa` 的整体结构形成了一个闭环。

```
// 精简版定义
typedef struct objc_class *Class;

struct objc_class : objc_object {
    // Class ISA;
    Class superclass;
}

struct objc_object {
    Class _Nonnull isa _OBJC_ISA_AVAILABILITY;
};
```

在对象的继承体系中，类和元类都有各自的继承体系，但它们都有共同的根父类 `NSObject`，而 `NSObject` 的父类指向 `nil`。需要注意的是，上图中 `Root Class(Class)` 是 `NSObject` 类对象，而 `Root Class(Meta)` 是 `NSObject` 的元类对象。

基础定义

在 `objc-private.h` 文件中，有一些项目中常用的基础定义，这是最新的 `objc-723` 中的定义，可以来看一下。

```
typedef struct objc_class *Class;
typedef struct objc_object *id;

typedef struct method_t *Method;
typedef struct ivar_t *Ivar;
typedef struct category_t *Category;
typedef struct property_t *objc_property_t;
```

IMP

在 `Runtime` 中 `IMP` 本质上就是一个函数指针，其定义如下。在 `IMP` 中有两个默认的参数 `id` 和 `SEL`，`id` 也就是方法中的 `self`，这和 `objc_msgSend()` 函数传递的参数一样。

```
typedef void (*IMP)(void /* id, SEL, ... */ );
```

`Runtime` 中提供了很多对于 `IMP` 操作的 API，下面就是不分 `IMP` 相关的函数定义。我们比较常见的是 `method_exchangeImplementations` 函数，`Method Swizzling` 就是通过这个 API 实现的。

```
OBJC_EXPORT void
method_exchangeImplementations(Method _Nonnull m1, Method _Nonnull m2)
   _OBJC_AVAILABLE(10.5, 2.0, 9.0, 1.0, 2.0);

OBJC_EXPORT IMP _Nonnull
method_setImplementation(Method _Nonnull m, IMP _Nonnull imp)
   _OBJC_AVAILABLE(10.5, 2.0, 9.0, 1.0, 2.0);

OBJC_EXPORT IMP _Nonnull
method_getImplementation(Method _Nonnull m)
   _OBJC_AVAILABLE(10.5, 2.0, 9.0, 1.0, 2.0);

OBJC_EXPORT IMP _Nullable
class_getMethodImplementation(Class _Nullable cls, SEL _Nonnull name)
```

```
OBJC_AVAILABLE(10.5, 2.0, 9.0, 1.0, 2.0);  
// ....
```

获取IMP

通过定义在 `NSObject` 中的下面两个方法，可以根据传入的 `SEL` 获取到对应的 `IMP`。`methodForSelector:` 方法不只实例对象可以调用，类对象也可以调用。

```
- (IMP)methodForSelector:(SEL)aSelector;  
+ (IMP)instanceMethodForSelector:(SEL)aSelector;
```

例如下面创建C函数指针用来接收 `IMP`，获取到 `IMP` 后可以手动调用 `IMP`，在定义的C函数中需要加上两个隐藏参数。

```
void (*function) (id self, SEL _cmd, NSObject *object);  
  
function = (id self, SEL _cmd, NSObject *object)[self methodForSelector:@selector(object:)];  
  
function(instance, @selector(object:), [NSObject *new]);
```

性能优化

通过这些 API 可以进行一些优化操作。如果遇到大量的方法执行，可以通过 `Runtime` 获取到 `IMP`，直接调用 `IMP` 实现优化。

```
TestObject *object = [[TestObject alloc] init];  
void(*function)(id, SEL) = (void*)(id, SEL))class_getMethodImplementation([TestObject  
class], @selector(testMethod));  
function(object, @selector(testMethod));
```

在获取和调用 `IMP` 的时候需要注意，每个方法默认都有两个隐藏参数，所以在函数声明的时候需要加上这两个隐藏参数，调用的时候也需要把相应的对象和 `SEL` 传进去，否则可能会导致 `Crash`。

IMP for block

`Runtime` 还支持 `block` 方式的回调，我们可以通过 `Runtime` 的 API，将原来的方法回调改为 `block` 的回调。

```

// 类定义
@interface TestObject : NSObject
- (void)testMethod:(NSString *)text;
@end

// 类实现
@implementation TestObject
- (void)testMethod:(NSString *)text {
    NSLog(@"testMethod : %@", text);
}
@end

// runtime
IMP function = imp_implementationWithBlock^(id self, NSString *text) {
    NSLog(@"callback block : %@", text);
};
const char *types = sel_getName(@selector(testMethod:));
class_replaceMethod([TestObject class], @selector(testMethod:), function, types);

TestObject *object = [[TestObject alloc] init];
[object testMethod:@"lxz"];

// 输出
callback block : lxz

```

Method

Method 用来表示方法，其包含 SEL 和 IMP，下面可以看一下 Method 结构体的定义。

```

typedef struct method_t *Method;

struct method_t {
    SEL name;
    const char *types;
    IMP imp;
};

```

在运行过程中是这样。

```
▼ L meth = (Method) 0x100001150
  ► name = (SEL) "testMethod"
  ► types = (const char *) "v16@0:8"
  ► imp = (IMP) (debug-objc`-[TestObject testMethod] at TestObject.m:14)
```

在 Xcode 进行编译的时候，只会将 Xcode 的 Compile Sources 中 .m 声明的方法编译到 Method List，而 .h 文件中声明的方法对 Method List 没有影响。

Property

在 Runtime 中定义了属性的结构体，用来表示对象中定义的属性。@property 修饰符用来修饰属性，修饰后的属性为 objc_property_t 类型，其本质是 property_t 结构体。其结构体定义如下。

```
typedef struct property_t *objc_property_t;

struct property_t {
    const char *name;
    const char *attributes;
};
```

可以通过下面两个函数，分别获取实例对象的属性列表，和协议的属性列表。

```
objc_property_t * class_copyPropertyList (Class cls, unsigned int * outCount)
objc_property_t * protocol_copyPropertyList (Protocol * proto, unsigned int * outCount)
```

可以通过下面两个方法，传入指定的 class 和 propertyName，获取对应的 objc_property_t 属性结构体。

```
objc_property_t class_getProperty (Class cls, const char * name)
objc_property_t protocol_getProperty (Protocol * proto, const char * name, BOOL isRequiredProperty, BOOL isInstanceProperty)
```

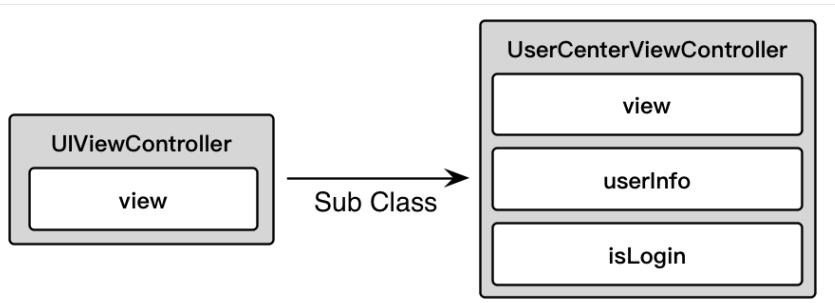
分析实例变量

对象间关系

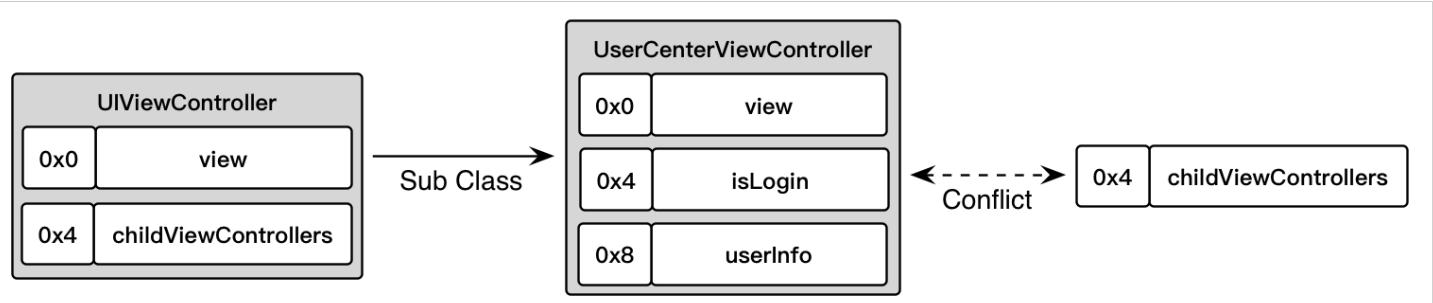
在OC中绝大多数类都是继承自 NSObject 的(NSProxy 例外)，类与类之间都会存在继承关系。通

过子类创建对象时，继承链中所有成员变量都会存在对象中。

例如下图中，父类是 `UIViewController`，具有一个 `view` 属性。子类 `UserCenterViewController` 继承自 `UIViewController`，并定义了两个新属性。这时如果通过子类创建对象，就会同时包含着三个实例变量。



但是类的结构在编译时都是固定的，如果想要修改类的结构需要重新编译。如果上线后用户安装到设备上，新版本的iOS系统中更新了父类的结构，也就是 `UIViewController` 的结构，为其加入了新的实例变量，这时用户更新新的iOS系统后就会导致问题。

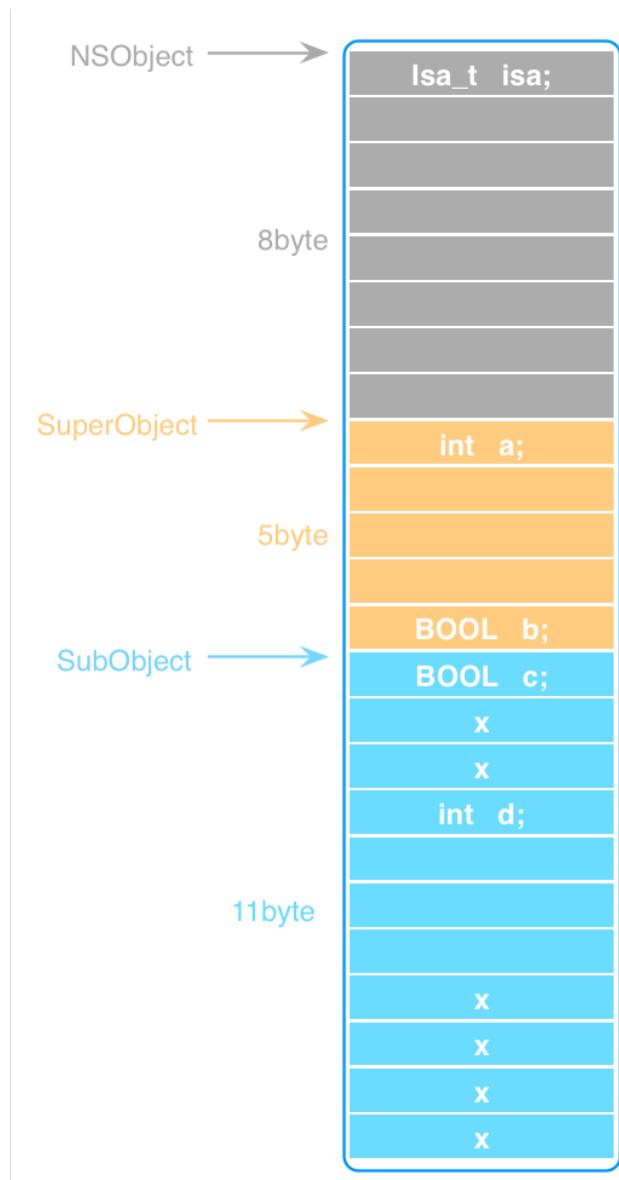


原来 `UIViewController` 的结构中增加了 `childViewControllers` 属性，这时候和子类的内存偏移就发生冲突了。只不过，Runtime 有检测内存冲突的机制，在类生成实例变量时，会判断实例变量是否有地址冲突，如果发生冲突则调整对象的地址偏移，这样就在运行时解决了地址冲突的问题。

内存布局

类的本质是结构体，在结构体中包含一些成员变量，例如 `method list`、`ivar list` 等，这些都是结构体的一部分。`method`、`protocol`、`property` 的实现这些都可以放到类中，所有对象调用同一份即可，但对象的成员变量不可以放在一起，因为每个对象的成员变量值都是不同的。

创建实例对象时，会根据其对应的Class分配内存，内存构成是`ivars+isa_t`。并且实例变量不只包含当前 Class 的 `ivars`，也会包含其继承链中的 `ivars`。`ivars` 的内存布局在编译时就已经决定，运行时需要根据 `ivars` 内存布局创建对象，所以 Runtime 不能动态修改 `ivars`，会破坏已有内存布局。



(上图中，“x”表示地址对其后的空位)

以上图为例，创建的对象中包含所属类及其继承者链中，所有的成员变量。因为对象是结构体，所以需要进行地址对其，一般OC对象的大小都是8的倍数。

也不是所有对象都不能动态修改ivars，如果是通过runtime动态创建的类，是可以修改ivars的。这个在后面会有讲到。

ivar读写

实例变量的 `isa_t` 指针会指向其所属的类，对象中并不会包含 `method`、`protocol`、`property`、`ivar` 等信息，这些信息在编译时都保存在只读结构体 `class_ro_t` 中。在 `class_ro_t` 中 `ivars` 是 `const` 只读的，在 `image load` 时 `copy` 到 `class_rw_t` 中时，是不会 `copy ivars` 的，并且 `class_rw_t` 中并没有定义 `ivars` 的字段。

在访问某个成员变量时，直接通过 `isa_t` 找到对应的 `objc_class`，并通过其 `class_ro_t` 的 `ivar`

`list` 做地址偏移，查找对应的对象内存。正是由于这种方式，所以对象的内存地址是固定不可改变的。

方法传参

当调用实例变量的方法时，会通过 `objc_msgSend()` 发起调用，调用时会传入 `self` 和 `SEL`。函数内部通过 `isa` 在类的内部查找方法列表对应的 `IMP`，传入对应的参数并发起调用。如果调用的方法时涉及到当前对象的成员变量的访问，这时候就是在 `objc_msgSend()` 内部，通过类的 `ivar list` 判断地址偏移，取出 `ivar` 并传入调用的 `IMP` 中的。

调用 `super` 的方式时则调用 `objc_msgSendSuper()` 函数实现，调用时将实例变量的父类传进去。但是需要注意的是，调用 `objc_msgSendSuper` 函数时传入的对象，也是当前实例变量，所以是在向自己发送父类的消息。具体可以看一下 `[self class]` 和 `[super class]` 的结果，结果应该都是一样的。

在项目中经常会通过 `[super xxx]` 的方式调用父类方法，这是因为需要先完成父类的操作，当然也可以不调用，视情况而定。以经常见到的自定义 `init` 方法中，经常会出现 `if (self = [super init])` 的调用，这是在完成自己的初始化之前先对父类进行初始化，否则只初始化自身可能会存在问题。在调用 `[super init]` 时如果返回 `nil`，则表示父类初始化失败，这时候初始化子类肯定会出现问题，所以需要做判断。

参考资料

[Apple Runtime Program Guild](#)

[维基百科-Objective-C](#)

[维基百科-Clang](#)

[维基百科-GCC\(GNU\)](#)

苹果开源代码不建议去Github，上面的版本一般更新不及时，建议去苹果的开源官网。

[Apple Opensource](#)

剖析Runtime结构体

NSObject

之前的定义

在 oc1.0 中， Runtime 很多定义都写在 `NSObject.h` 文件中，如果之前研究过 Runtime 的同学可以应该见过下面的定义，定义了一些基础的信息。

```
// 声明Class和id
typedef struct objc_class *Class;
typedef struct objc_object *id;

// 声明常用变量
typedef struct objc_method *Method;
typedef struct objc_ivar *Ivar;
typedef struct objc_category *Category;
typedef struct objc_property *objc_property_t;

// objc_object和objc_class
struct objc_object {
    Class _Nonnull isa _OBJC_ISA_AVAILABILITY;
};

struct objc_class {
    Class isa _OBJC_ISA_AVAILABILITY;

#if !_OBJC2_
    Class super_class
    const char *name
    long version
    long info
    long instance_size
    struct objc_ivar_list *ivars
    struct objc_method_list **methodLists
    struct objc_cache *cache
    struct objc_protocol_list *protocols
#endif

}_OBJC2_UNAVAILABLE;
```

之前的 Runtime 结构也比较简单，都是一些很直接的结构体定义，现在新版的 Runtime 在操作的

时候，各种地址偏移操作和位运算。

之后的定义

后来可能苹果也不太想让开发者知道 Runtime 内部的实现，所以就把源码定义从 `NSObject` 中搬到 `Runtime` 中了。而且之前的定义也不用了，通过 `OBJC_TYPES_DEFINED` 预编译指令，将之前的代码废弃调了。

现在 `NSObject` 中的定义非常简单，直接就是一个 `Class` 类型的 `isa` 变量，其他信息都隐藏起来了。

```
@interface NSObject <NSObject> {
    #pragma clang diagnostic push
    #pragma clang diagnostic ignored "-Wobjc-interface-ivars"
    Class isa OBJC_ISA_AVAILABILITY;
    #pragma clang diagnostic pop
}
```

这是最新的一些常用 `Runtime` 定义，和之前的定义也不太一样了，用了最新的结构体对象，之前的结构体也都废弃了。

```
typedef struct objc_class *Class;
typedef struct objc_object *id;

typedef struct method_t *Method;
typedef struct ivar_t *Ivar;
typedef struct category_t *Category;
typedef struct property_t *objc_property_t;
```

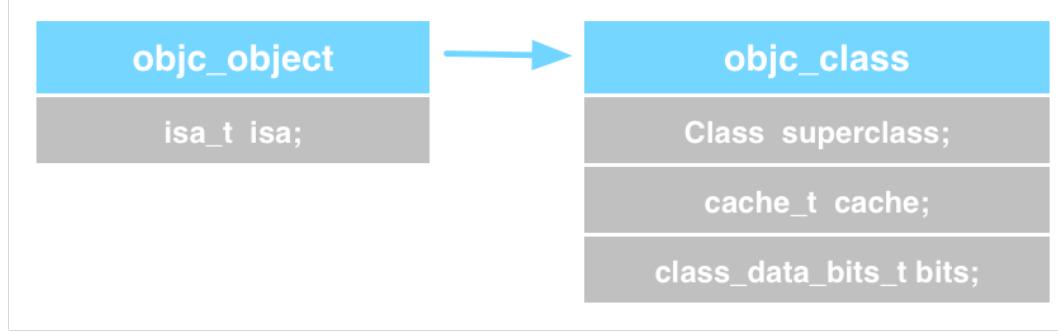
对象结构体

`objc_object` 定义

在OC中每个对象都是一个结构体，结构体中都包含一个`isa`的成员变量，其位于成员变量的第一位。`isa` 的成员变量之前都是 `Class` 类型的，后来苹果将其改为 `isa_t`。

```
struct objc_object {
private:
    isa_t isa;
};
```

OC中的类和元类也是一样，都是结构体构成的。由于类的结构体定义继承自 `objc_object`，所以其也是一个对象，并且具有对象的 `isa` 特征。



所以可以通过 `isa_t` 来查找对应的类或元类，查找方法应该是通过 `uintptr_t` 类型的 `bits`，通过按位操作来查找 `isa_t` 指向的类的地址。

实例对象或类对象的方法，并不会定义在各个对象中，而是都定义在 `isa_t` 指向的类中。查找到对应的类后，通过类的 `class_data_bits_t` 类型的 `bits` 结构体查找方法，对象、类、元类都是同样的查找原理。

isa_t定义

`isa_t` 是一个 `union` 的结构对象，`union` 类似于 C++ 结构体，其内部可以定义成员变量和函数。在 `isa_t` 中定义了 `cls`、`bits`、`isa_t` 三部分，下面的 `struct` 结构体就是 `isa_t` 的结构体构成。

下面对 `isa_t` 中的结构体进行了位域声明，地址从 `nonpointer` 起到 `extra_rc` 结束，从低到高进行排列。位域也是对结构体内存布局进行了一个声明，通过下面的结构体成员变量可以直接操作某个地址。位域总共占8字节，所有的位域加在一起正好是64位。

小提示：`union` 中 `bits` 可以操作整个内存区，而位域只能操作对应的位。

下面的代码是不完整代码，只保留了 `arm64` 部分，其他部分被忽略掉了。

```
union isa_t
{
    isa_t() { }
    isa_t(uintptr_t value) : bits(value) { }

    Class cls;
    uintptr_t bits;

    # if __arm64__
    # define ISA_MASK          0x0000000fffffff8ULL
    # define ISA_MAGIC_MASK    0x000003f000000001ULL
    # define ISA_MAGIC_VALUE  0x000001a000000001ULL

```

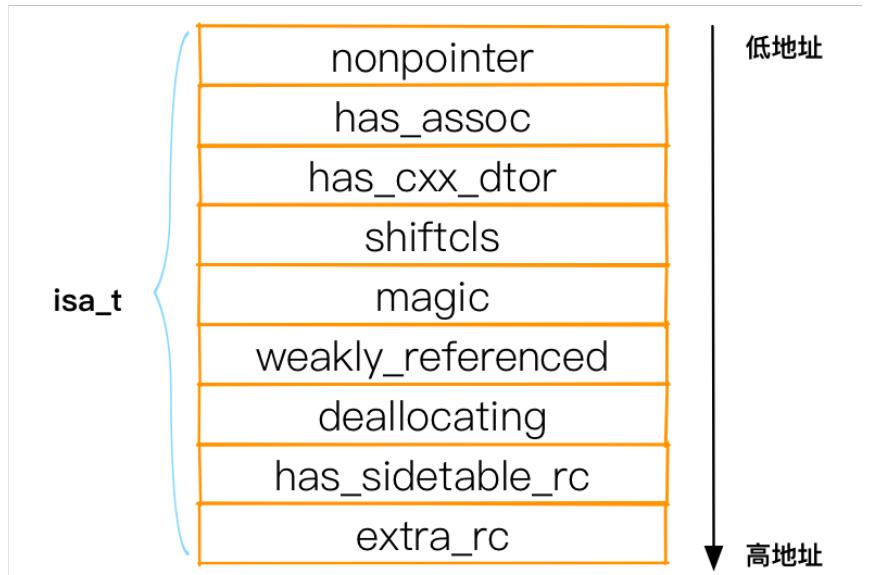
```

struct {
    uintptr_t nonpointer : 1; // 是32位还是64位
    uintptr_t has_assoc : 1; // 对象是否含有或曾经含有关联引用, 如果没有关联引用, 可以更快的释放对象
    uintptr_t has_cxx_dtor : 1; // 表示是否有C++析构函数或OC的析构函数
    uintptr_t shiftcls : 33; // 对象指向类的内存地址, 也就是isa指向的地址
    uintptr_t magic : 6; // 对象是否初始化完成
    uintptr_t weakly_referenced : 1; // 对象是否被弱引用或曾经被弱引用
    uintptr_t deallocating : 1; // 对象是否被释放中
    uintptr_t has_sidetable_rc : 1; // 对象引用计数太大, 是否超出存储区域
    uintptr_t extra_rc : 19; // 对象引用计数
#define RC_ONE (1ULL<<45)
#define RC_HALF (1ULL<<18)
};

# elif __x86_64__
// ...
# else
// ...
# endif
};

```

在 ARM64 架构下, `isa_t` 以以下结构进行布局。在不同的 CPU 架构下, 布局方式会有所不同, 但参数都是一样的。



类结构体

objc_class 结构体

在 `Runtime` 中类也是一个对象, 类的结构体 `objc_class` 是继承自 `objc_object` 的, 具备对象所

有的特征。在 `objc_class` 中定义了三个成员变量，`superclass` 是一个 `objc_class` 类型的指针，指向其父类的 `objc_class` 结构体。`cache` 用来处理已调用方法的缓存。

`bits` 是 `objc_class` 的主角，其内部只定义了一个 `uintptr_t` 类型的 `bits` 成员变量，存储了 `class_rw_t` 的地址。`bits` 中还定义了一些基本操作，例如获取 `class_rw_t`、`raw isa` 状态、是否 `swift` 等函数。`objc_class` 结构体中定义的一些函数，其内部都是通过 `bits` 实现的。

```
struct objc_class : objc_object {
    // Class ISA;
    Class superclass;
    cache_t cache;
    class_data_bits_t bits;

    class_rw_t *data() {
        return bits.data();
    }
    void setData(class_rw_t *newData) {
        bits.setData(newData);
    }
    // .....
}
```

从 `objc_class` 的源码可以看出，可以通过 `bits` 结构体的 `data()` 函数，获取 `class_rw_t` 指针。我们进入源代码中看一下，可以看出是通过对 `uintptr_t` 类型的 `bits` 变量，做位运算查找对应的值。

```
class_rw_t* data() {
    return (class_rw_t *)(bits & FAST_DATA_MASK);
}
```

`uintptr_t` 本质上是一个 `unsigned long` 的 `typedef`，`unsigned long` 在64位处理器中占8字节，正好是64位二进制。通过 `FAST_DATA_MASK` 转换为二进制后，是取 `bits` 中的**47-3**的位置，正好是取出 `class_rw_t` 指针。

在OC中一个指针的长度是47，例如打印一个 `UIViewController` 的地址是 `0x7faf1b580450`，转换为二进制是 `111111101011110001101101011000000010001010000`，最后面三位是占位的，所以在取地址的时候会忽略最后三位。

```
// 查找第0位，表示是否swift
#define FAST_IS_SWIFT          (1UL<<0)
// 当前类或父类是否定义了retain、release等方法
#define FAST_HAS_DEFAULT_RR    (1UL<<1)
```

因为在 `bits` 中最后三位是没用的，所以可以用来存储一些其他信息。在 `class_data_bits_t` 还定义了三个宏，用来对后三位做位运算。

class_ro_t和**class_rw_t**

和 `class_data_bits_t` 相关的有两个很重要结构体，`class_rw_t` 和 `class_ro_t`，其中都定义着 `method list`、`protocol list`、`property list` 等关键信息。

```
struct class_rw_t {
    uint32_t flags;
    uint32_t version;

    const class_ro_t *ro;

    method_array_t methods;
    property_array_t properties;
    protocol_array_t protocols;

    Class firstSubclass;
    Class nextSiblingClass;

    char *demangledName;
};
```

在编译后 `class_data_bits_t` 指向的是一个 `class_ro_t` 的地址，这个结构体是不可变的(只读)。在运行时，才会通过 `realizeClass` 函数将 `bits` 指向 `class_rw_t`。

```
struct class_ro_t {
    uint32_t flags;
    uint32_t instanceStart;
    uint32_t instanceSize;
    uint32_t reserved;

    const uint8_t * ivarLayout;

    const char * name;
    method_list_t * baseMethodList;
    protocol_list_t * baseProtocols;
```

```

const ivar_list_t * ivars;

const uint8_t * weakIvarLayout;
property_list_t *baseProperties;
};

```

在程序开始运行后会初始化 `Class`，在这个过程中，会把编译器存储在 `bits` 中的 `class_ro_t` 取出，然后创建 `class_rw_t`，并把 `ro` 赋值给 `rw`，成为 `rw` 的一个成员变量，最后把 `rw` 设置给 `bits`，替代之前 `bits` 中存储的 `ro`。除了这些操作外，还会有一些其他赋值的操作，下面是初始化 `Class` 的精简版代码。

```

static Class realizeClass(Class cls)
{
    const class_ro_t *ro;
    class_rw_t *rw;
    Class supercls;
    Class metacls;
    bool isMeta;

    if (!cls) return nil;
    if (cls->isRealized()) return cls;

    ro = (const class_ro_t *)cls->data();
    rw = (class_rw_t *)calloc(sizeof(class_rw_t), 1);
    rw->ro = ro;
    rw->flags = RW_REALIZED|RW_REALIZING;
    cls->setData(rw);

    isMeta = ro->flags & RO_META;
    rw->version = isMeta ? 7 : 0;

    supercls = realizeClass(remapClass(cls->superclass));
    metacls = realizeClass(remapClass(cls->ISA()));

    cls->superclass = supercls;
    cls->initClassIsa(metacls);
    cls->setInstanceSize(ro->instanceSize);

    if (supercls) {
        addSubclass(supercls, cls);
    } else {
        addRootClass(cls);
    }

    methodizeClass(cls);
    return cls;
}

```

```
}
```

在上面的代码中我们还发现了两个函数，`addRootClass` 和 `addSubclass` 函数，这两个函数的职责是将某个类的子类串成一个列表，大致是下面的链接顺序。由此可知，我们是可以通过 `class_rw_t`，获取到当前类的所有子类。

```
superClass.firstSubclass -> subClass1.nextSiblingClass -> subClass2.nextSiblingClass  
s -> ...
```

初始化 `rw` 和 `ro` 之后，`rw` 的 `method list`、`protocol list`、`property list` 都是空的，需要在下面 `methodizeClass` 函数中进行赋值。函数中会把 `ro` 的 `list` 都取出来，然后赋值给 `rw`，如果在运行时动态修改，也是对 `rw` 做的操作。所以 `ro` 中存储的是编译时就已经决定的原数据，`rw` 才是运行时动态修改的数据。

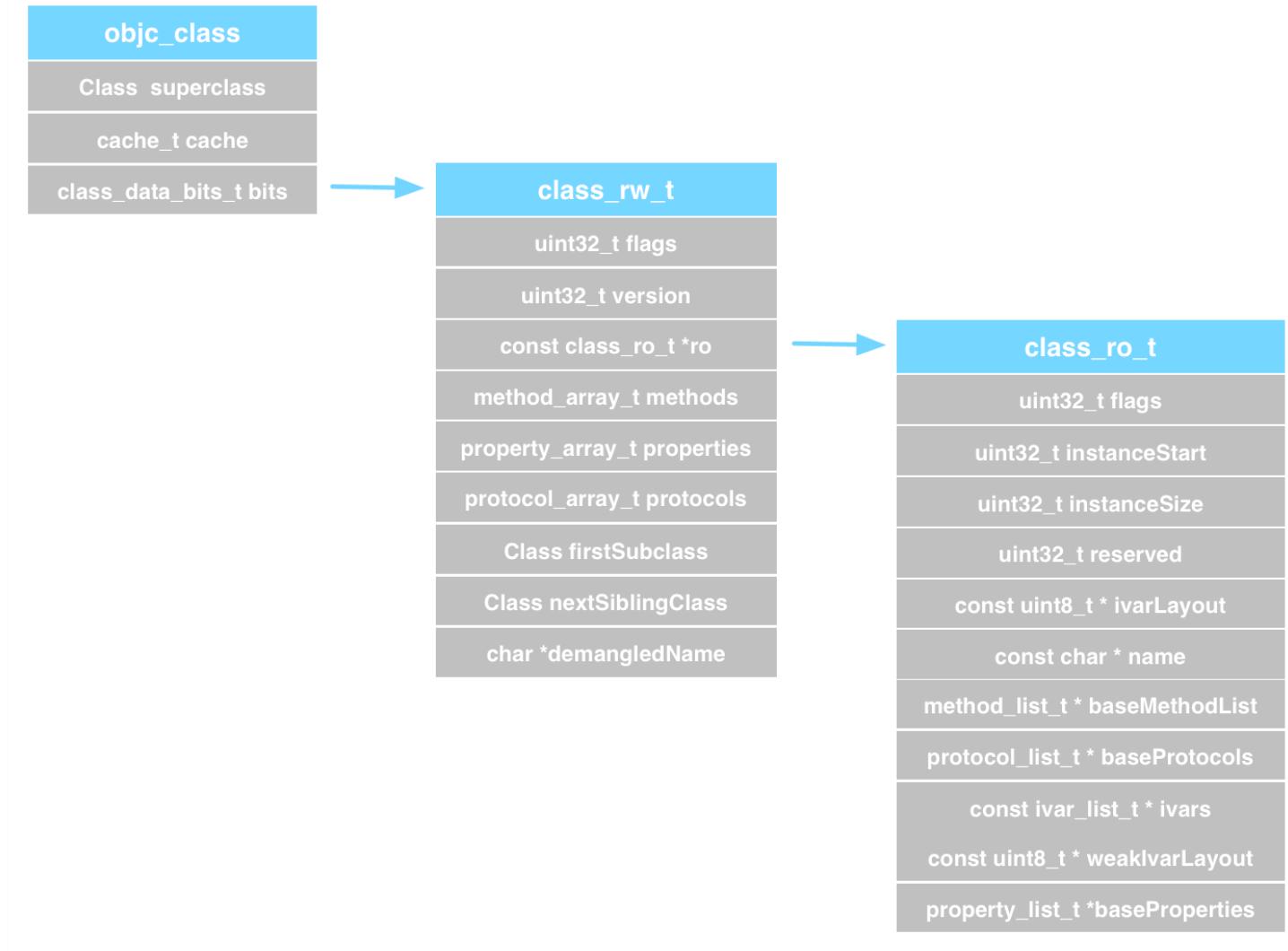
```
static void methodizeClass(Class cls)  
{  
    bool isMeta = cls->isMetaClass();  
    auto rw = cls->data();  
    auto ro = rw->ro;  
  
    method_list_t *list = ro->baseMethods();  
    if (list) {  
        prepareMethodLists(cls, &list, 1, YES, isBundleClass(cls));  
        rw->methods.attachLists(&list, 1);  
    }  
  
    property_list_t *proplist = ro->baseProperties;  
    if (proplist) {  
        rw->properties.attachLists(&proplist, 1);  
    }  
  
    protocol_list_t *protolist = ro->baseProtocols;  
    if (protolist) {  
        rw->protocols.attachLists(&protolist, 1);  
    }  
  
    if (cls->isRootMetaclass()) {  
        // root metaclass  
        addMethod(cls, SEL_initialize, (IMP)&objc_noop_imp, "", NO);  
    }  
  
    // Attach categories.  
    category_list *cats = unattachedCategoriesForClass(cls, true /*realizing*/);  
    attachCategories(cls, cats, false /*don't flush caches*/);
```

```
}
```

假设创建一个类 `LXZObject`，继承自 `NSObject`，并为其加入一个 `testMethod` 方法，不做其他操作。因为在编译后 `objc_class` 的 `bits` 对应的是 `class_ro_t` 结构体，所以我们打印一下结构体的成员变量，看一下编译后的 `class_ro_t` 是什么样的。

```
struct class_ro_t {
    flags = 128
    instanceStart = 8
    instanceSize = 8
    reserved = 0
    ivarLayout = 0x0000000000000000 <no value available>
    name = 0x00000010000f7a "LXZObject"
    baseMethodList = 0x00000001000010c8
    baseProtocols = 0x0000000000000000
    ivars = 0x0000000000000000
    weakIvarLayout = 0x0000000000000000 <no value available>
    baseProperties = 0x0000000000000000
}
```

经过打印可以看出，一个类的 `class_ro_t` 中只会包含当前类的信息，不会包含其父类的信息，在 `LXZObject` 类中只会包含 `name` 和 `baseMethodList` 两个字段，而 `baseMethodList` 中只有一个 `testMethod` 方法。由此可知，`class_rw_t` 结构体也是一样的。



初始化过程

下面是已经初始化后的 `isa_t` 结构体的布局，以及各个结构体成员在结构体中的位置。



`union` 经常配合结构体使用，第一次使用 `union` 就是对结构体区域做初始化。在对象初始化时，会对 `isa_t` 的 `bits` 字段赋值为 `ISA_MAGIC_VALUE`，这就是对 `union` 联合体初始化的过程。

```
// 在objc-723中已经没有了
inline void objc_object::initIsa(Class cls, bool indexed, bool hasCxxDtor)
{
    if (!indexed) {
        isa.cls = cls;
    } else {
        isa.bits = ISA_MAGIC_VALUE;
        isa.has_cxx_dtor = hasCxxDtor;
        isa.shiftcls = (uintptr_t)cls >> 3;
    }
}
```

在对象通过 `initIsa()` 函数初始化时，会通过 `ISA_MAGIC_VALUE` 对 `isa` 进行初始化。`ISA_MAGIC_VALUE` 是一个16进制的值，将其转换为二进制后，会发现 `ISA_MAGIC_VALUE` 是对 `nonpointer` 和 `magic` 做初始化。

`nonpointer` 是对之前32位处理器的兼容。在访问对象所属的类时，如果是32位则返回之前的 `isa` 指针地址，否则表示是64位处理器，则返回 `isa_t` 结构体。

随后会通过位域，对 `has_cxx_dtor` 和 `shiftcls` 做初始化，这时候就已经有四个字段被初始化了。`has_cxx_dtor` 表示是否有 C++ 或 OC 的析构方法，在打印方法列表时，经常能看到一个名为 `.cxx_destruct` 的方法，就和这个字段有关系。

在计算机中为了对存储区 (Memory or Disk) 读取方便, 所以在写入和读取时, 会对内存有对其操作。一般是以字节为单位进行对其, 这样也是对读写速度的优化。在对 `shiftcls` 进行赋值时, 对 `class` 的指针进行了位移操作, 向右位移三位。这是因为类指针为了内存对其, 将最后三位用0填充, 所以这三位是没有意义的。

isa结构体

00000000010111011000000000000010000000001110101110000011111001
0xd8001003ae0f8

类对象地址

1000000000111010111000011111000
0x1003ae0f8

将类对象地址右移三位为100000000001110101110000011111，正好符合isa_t地址中shiftcls的部分，前面不足补零。

外界获取 `Class` 时，应该通过 `ISA()` 函数，而不是像之前一样直接访问 `isa` 指针。在 `ISA()` 函数中，是对 `isa_t` 的结构体做与运算，是通过 `ISA_MASK` 宏进行的，转换为二进制的话，正好是把 `shiftcls` 的地址取出来。

Tagged Pointer

从 iPhone5s 开始，ios 设备开始引入了64位处理器，之前的处理器一直都是32位的。

但是在64位处理器中，指针长度以及一些变量所占内存都发生了改变，32位一个指针占用4字

节，但64位一个指针占用8字节；32位一个 long 占用4字节，64位一个 long 占用8字节等，所以在64位上内存占用会多出很多。

苹果为了优化这个问题，推出了Tagged Pointer新特性。之前一个指针指向一个地址，而 Tagged Pointer中一个指针就代表一个值，以NSNumber为例。

```
NSNumber *number1 = @1;
NSNumber *number2 = @3;
NSNumber *number3 = @54;

// 输出
(lldb) p number1
(__NSCFNumber *) $3 = 0xb000000000000012 (int)1
(lldb) p number2
(__NSCFNumber *) $4 = 0xb000000000000032 (int)3
(lldb) p number3
(__NSCFNumber *) $5 = 0xb000000000000362 (int)54
```

通过上面代码可以看出，使用了 Tagged Pointer 新特性后，指针中就存储着对象的值。例如一个值为1的 NSNumber，指针就是 0xb000000000000012，如果抛去前面的 0xb 和后面的2，中间正好就是16进制的值。

苹果通过 Tagged Pointer 的特性，明显的提升了执行效率并节省了很多内存。在64位处理器下，内存占用减少了将近一半，执行效率也大大提升。由于通过指针来直接表示数值，所以没有了 malloc 和 free 的过程，对象的创建和销毁速度提升几十倍。

isa_t

对于对象指针也是一样，在 oc1.0 时代 isa 是一个真的指针，指向一个堆区的地址。而 oc2.0 时代，一个指针长度是八字节也就是64位，在64位中直接存储着对象的信息。当查找对象所属的类时，直接在 isa 指针中进行位运算即可，而且由于是在栈区进行操作，查找速度是非常快的。

```
struct {
    uintptr_t nonpointer      : 1;
    uintptr_t has_assoc        : 1;
    uintptr_t has_cxx_dtor     : 1;
    uintptr_t shiftcls         : 33;
    uintptr_t magic             : 6;
    uintptr_t weakly_referenced : 1;
    uintptr_t deallocating      : 1;
    uintptr_t has_sidetable_rc  : 1;
    uintptr_t extra_rc          : 19;
};
```

例如 `isa_t` 本质上是一个结构体，如果创建结构体再用指针指向这个结构体，内存占用是很大的。但是 `Tagged Pointer` 特性中，直接把结构体的值都存储到指针中，这就相当节省内存了。

苹果不允许直接访问 `isa` 指针，和 `Tagged Pointer` 也是有关系的。因为在 `Tagged Pointer` 的情况下，`isa` 并不是一个指针指向另一块内存区，而是直接表示对象的值，所以通过直接访问 `isa` 获取到的信息是错误的。

Runtime源码分析

本文基于 objc-723 版本，在[Apple Github](#)和[Apple OpenSource](#)上有源码，但是需要自己编译。

重点来了~，可以到我的 [Github](#) 上下载编译好的源码，源码中已经写了大量的注释，方便读者研究。(如果觉得还不错，各位大佬麻烦点个 Star 😊)

Runtime Analyze

对象的初始化流程

在对象初始化的时候，一般都会调用 `alloc+init` 方法实例化，或者通过 `new` 方法进行实例化。下面将会分析通过 `alloc+init` 的方式实例化的过程，以下代码都是关键代码。

前面两步很简单，都是直接进行函数调用。

```
+ (id)alloc {
    return _objc_rootAlloc(self);
}

id _objc_rootAlloc(Class cls)
{
    return callAlloc(cls, false/*checkNil*/, true/*allocWithZone*/);
}
```

在创建对象的地方有两种方式，一种是通过 `callAlloc` 开辟内存，然后通过 `initInstanceIsa` 函数初始化这块内存。第二种是直接调用 `class_createInstance` 函数，由内部实现初始化逻辑。

```
static ALWAYS_INLINE id
callAlloc(Class cls, bool checkNil, bool allocWithZone=false)
{
    if (fastpath(cls->canAllocFast())) {
        bool dtor = cls->hasCxxDtor();
        id obj = (id)calloc(1, cls->bits.fastInstanceState());
        if (slowpath(!obj)) return callBadAllocHandler(cls);
        obj->initInstanceIsa(cls, dtor);
        return obj;
    }
    else {
        id obj = class_createInstance(cls, 0);
        if (slowpath(!obj)) return callBadAllocHandler(cls);
        return obj;
    }
}
```

```
    }
}
```

但是在最新版的 objc-723 中，调用 `canAllocFast` 函数直接返回 `false`，所以只会执行上面第二个 `else` 代码块。

```
bool canAllocFast() {
    return false;
}
```

初始化代码最终会调用到 `_class_createInstanceFromZone` 函数，这个函数是初始化的关键代码。下面代码中会进入 `if` 语句内，根据 `instanceSize` 函数返回的 `size`，通过 `calloc` 函数分配内存，并初始化 `isa_t` 指针。

```
id class_createInstance(Class cls, size_t extraBytes)
{
    return _class_createInstanceFromZone(cls, extraBytes, nil);
}

static __attribute__((always_inline))
id _class_createInstanceFromZone(Class cls, size_t extraBytes, void *zone,
                                  bool cxxConstruct = true,
                                  size_t *outAllocatedSize = nil)
{
    bool hasCxxCtor = cls->hasCxxCtor();
    bool hasCxxDtor = cls->hasCxxDtor();
    bool fast = cls->canAllocNonpointer();
    size_t size = cls->instanceSize(extraBytes);

    id obj;
    if (!zone && fast) {
        obj = (id)calloc(1, size);
        if (!obj) return nil;
        obj->initInstanceIsa(cls, hasCxxDtor);
    }
    else {
        if (zone) {
            obj = (id)malloc_zone_calloc ((malloc_zone_t *)zone, 1, size);
        } else {
            obj = (id)calloc(1, size);
        }
        if (!obj) return nil;
        obj->initIsa(cls);
    }
}
```

```
    return obj;
}
```

在 `instanceSize()` 函数中，会通过 `alignedInstanceSize` 函数获取对象原始大小，在 `class_ro_t` 结构体中的 `instanceSize` 变量中定义。这个变量中存储了对象实例化时，所有变量所占的内存大小，这个大小是在编译器就已经决定的，不能在运行时进行动态改变。

获取到 `instanceSize` 后，对获取到的 `size` 进行地址对其。需要注意的是，CF 框架要求所有对象大小最少是16字节，如果不够则直接定义为16字节。

```
size_t instanceSize(size_t extraBytes) {
    size_t size = alignedInstanceSize() + extraBytes;
    // CF requires all objects be at Least 16 bytes.
    if (size < 16) size = 16;
    return size;
}
```

这也是很关键的一步，由于调用 `initIsa` 函数时，`nonpointer` 字段传入 `true`，所以直接执行 `if` 语句，设置 `isa` 的 `cls` 为传入的 `class`。`isa` 是 `objc_object` 的结构体成员变量，也就是 `isa_t` 的类型。

```
inline void objc_object::initInstanceIsa(Class cls, bool hasCxxDtor)
{
    initIsa(cls, true, hasCxxDtor);
}

inline void objc_object::initIsa(Class cls, bool nonpointer, bool hasCxxDtor)
{
    if (!nonpointer) {
        isa.cls = cls;
    } else {
        isa_t newisa(0);
        newisa.bits = ISA_MAGIC_VALUE;
        newisa.has_cxx_dtor = hasCxxDtor;
        newisa.shiftcls = (uintptr_t)cls >> 3;
        isa = newisa;
    }
}
```

通过 `new` 函数创建对象其实是一样的，内部通过 `callAlloc` 函数执行创建操作，如果调用 `alloc` 方法的话也是调用的 `callAlloc` 函数。所以调用 `new` 函数初始化对象时，可以等同于 `alloc+init` 的调用。

```
+ (id)new {
    return [callAlloc(self, false/*checkNil*/) init];
}
```

在runtime源码中，执行init操作本质上就是直接把self返回。

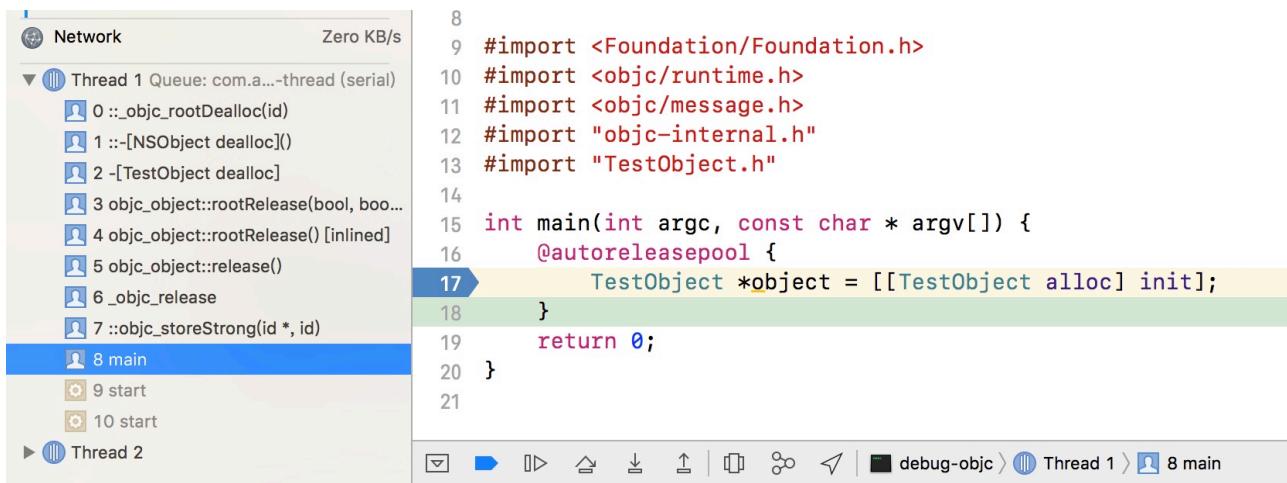
```
- (id)init {
    return _objc_rootInit(self);
}

id _objc_rootInit(id obj)
{
    return obj;
}
```

dealloc

在对象销毁时，运行时环境会调用 `NSObject` 的 `dealloc` 方法执行销毁代码，并不需要我们手动去调用。接着会调用到 Runtime 内部的 `objc_object::rootDealloc` (C++命名空间)函数。

在 `rootDealloc` 函数中会执行一些释放前的操作，例如将对象所有的引用指向 `nil`，并且调用 `free` 函数释放内存空间等。



下面的 `if-else` 语句中有判断条件，如果是 ARC 环境，并且当前对象定义了实例变量，才会进入 `else` 中执行 `object_dispose` 函数，否则进入上面的 `if` 语句。上面的 `if` 语句表示当前对象没有实例变量，则直接将当前对象 `free`。

```
inline void
objc_object::rootDealloc()
{
    if (isTaggedPointer()) return; // fixme necessary?
```

```

if (fastpath(isa.nonpointer &&
              !isa.weakly_referenced &&
              !isa.has_assoc &&
              !isa.has_cxx_dtor &&
              !isa.has_sidetable_rc))
{
    assert(!sidetable_present());
    free(this);
}
else {
    object_dispose((id)this);
}
}

```

在 `object_dispose` 函数中，主要是通过 `objc_destructInstance` 函数实现的。在函数内部主要做了三件事：

1. 对当前对象进行析构，会调用析构函数 `.cxx_destruct` 函数，在函数内部还会进行对应的 `release` 操作。
2. 移除当前对象的所有关联关系。
3. 进行最后的 `clear` 操作。

```

// dealloc方法的核心实现，内部会做判断和析构操作
void *objc_destructInstance(id obj)
{
    if (obj) {
        // 判断是否有OC或C++的析构函数
        bool cxx = obj->hasCxxDtor();
        // 对象是否有相关联的引用
        bool assoc = obj->hasAssociatedObjects();

        // 对当前对象进行析构
        if (cxx) object_cxxDestruct(obj);
        // 移除所有对象的关联，例如把weak指针置nil
        if (assoc) _object_remove_associations(obj);
        obj->clearDeallocating();
    }

    return obj;
}

```

上面的函数中会调用 `object_cxxDestruct` 函数进行析构，而函数内部是通过 `object_cxxDestructFromClass` 函数实现的。

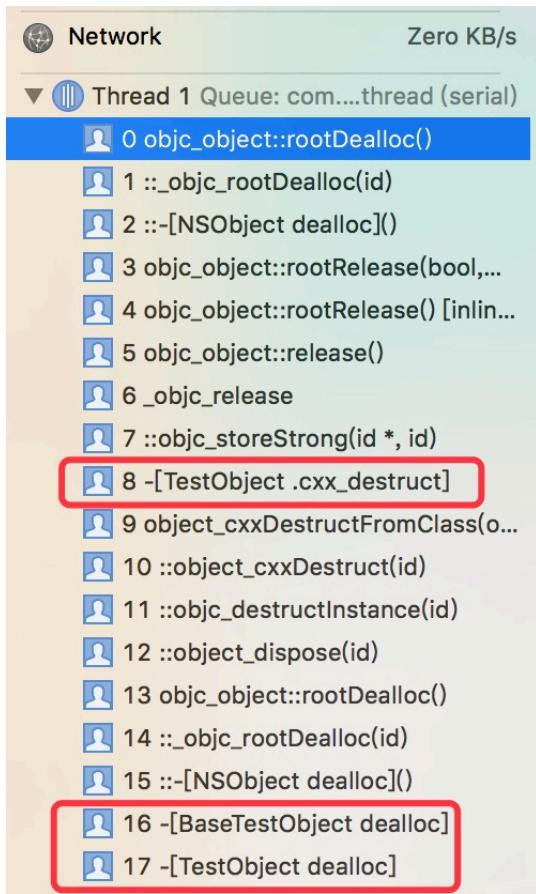
函数内部会从当前对象所属的类开始遍历，一直遍历到根类位置。在遍历的过程中，会不断执行 `.cxx_destruct` 函数，对传入的对象进行析构。

因为在继承者链中，每个类都会有自己的析构代码，所以需要将当前对象传入，并逐个执行析构操作，将对象的所有析构操作都执行完成才可以。

```
// 调用C++的析构函数
static void object_cxxDestructFromClass(id obj, Class cls)
{
    void (*dtor)(id);

    // 从当前类开始遍历，直到遍历到根类
    for ( ; cls; cls = cls->superclass) {
        if (!cls->hasCxxDtor()) return;
        // SEL_cxx_destruct就是.cxx_destruct的selector
        dtor = (void*)(id)
            lookupMethodInClassAndLoadCache(cls, SEL_cxx_destruct);
        if (dtor != (void*)(id))_objc_msgForward_impcache) {
            // 获取到.cxx_destruct的函数指针并调用
            (*dtor)(obj);
        }
    }
}
```

在对象被执行 `.cxx_destruct` 析构函数后，析构函数内部还会调用一次 `release` 函数，完成最后的释放操作。



addMethod实现

在项目中经常会动态对方法列表进行操作，例如动态添加或替换一个方法，这时候会用到下面两个 `Runtime` 函数。在下面两个函数中，本质上都是通过 `addMethod` 函数实现的，在 `class_addMethod` 中对返回值进行了一个取反，所以如果此函数返回 `NO` 则表示方法已存在，不要重复添加。

```
BOOL class_addMethod(Class cls, SEL name, IMP imp, const char *types)
{
    if (!cls) return NO;

    rwlock_writer_t lock(runtimeLock);
    return ! addMethod(cls, name, imp, types ?: "", NO);
}

IMP class_replaceMethod(Class cls, SEL name, IMP imp, const char *types)
{
    if (!cls) return nil;

    rwlock_writer_t lock(runtimeLock);
    return addMethod(cls, name, imp, types ?: "", YES);
}
```

下面我们就分析一下 `addMethod` 函数的实现，依然只保留核心源码。

在 `addMethod` 函数中会先判断需要添加的方法是否存在，如果已经存在则直接返回对应的 `IMP`，否则就动态添加一个方法。在 `class_addMethod` 函数中有一个 `replace` 字段，表示区别是否 `class_replaceMethod` 函数调用过来的。如果 `replace` 是 `NO` 则直接返回 `IMP`，如果是 `YES` 则替换方法原有实现。

如果添加的方法不存在，则创建一个 `method_list_t` 结构体指针，并设置三个基本参数 `name`、`types`、`imp`，然后通过 `attachLists` 函数将新创建的 `method_list_t` 结构体添加到方法列表中。

```
static IMP
addMethod(Class cls, SEL name, IMP imp, const char *types, bool replace)
{
    IMP result = nil;
    method_t *m;
    if ((m = getMethodNoSuper_nolock(cls, name))) {
        // already exists
        if (!replace) {
            result = m->imp;
        } else {
            result = _method_setImplementation(cls, m, imp);
        }
    } else {
        // fixme optimize
        method_list_t *newlist;
        newlist = (method_list_t *)calloc(sizeof(*newlist), 1);
        newlist->entsizeAndFlags =
            (uint32_t)sizeof(method_t) | fixed_up_method_list;
        newlist->count = 1;
        newlist->first.name = name;
        newlist->first.types = strdupIfMutable(types);
        newlist->first.imp = imp;

        prepareMethodLists(cls, &newlist, 1, NO, NO);
        cls->data()->methods.attachLists(&newlist, 1);
        flushCaches(cls);

        result = nil;
    }

    return result;
}
```

在 `attachLists` 函数中实现比较简单，通过对原有地址做位移，并将新创建的 `method_list_t` 结

构体 copy 到方法列表中。

```
void attachLists(List* const * addedLists, uint32_t addedCount) {  
    // ...  
    memmove(array()->lists + addedCount, array()->lists, oldCount * sizeof(array()->lists[0]));  
    memcpy(array()->lists, addedLists, addedCount * sizeof(array()->lists[0]));  
    // ...  
}
```

添加Ivar

在 Runtime 中可以通过 `class_addIvar` 函数，向一个类添加实例对象。但是需要注意的是，这个函数不能向一个已经存在的类添加实例变量，只能想通过 Runtime API 创建的类动态添加实例变量。

函数应该在调用 `objc_allocateClassPair` 函数创建类之后，以及调用 `objc_registerClassPair` 函数注册的类之间添加实例变量，否则就会失败。也不能向一个元类添加实例变量，只能想类添加实例变量。

下面是动态创建一个类，并向新创建的类添加实例变量的代码。

```
Class testClass = objc_allocateClassPair([NSObject class], "TestObject", 0);  
BOOL isAdded = class_addIvar(testClass, "password", sizeof(NSString *), log2(sizeof(NSString *)), @encode(NSString *));  
objc_registerClassPair(testClass);  
  
if (isAdded) {  
    id object = [[testClass alloc] init];  
    [object setValue:@"1xz" forKey:@"password"];  
}
```

那么，为什么需要把动态添加实例变量的代码放在这两个函数中间呢？让我们一起来探究一下吧。

首先通过 `objc_allocateClassPair` 函数来创建类，创建时通过 `getClass` 函数判断类名是否已用，然后通过 `verifySuperclass` 函数判断 `superclass` 是否合适，如果任意条件不符合则创建类失败。

下面通过 `alloc_class_for_subclass` 函数创建类和元类，在 `alloc` 函数内部本质上是通过 `calloc` 函数分配内存空间，没有做其他操作。然后就执行 `objc_initializeClassPair_internal` 函数， `initialize` 函数内部都是初始化操作，用来初始

化刚刚创建的 Class 和 metaClass。

```
Class objc_allocateClassPair(Class superclass, const char *name,
                           size_t extraBytes)
{
    Class cls, meta;
    if (getClass(name) || !verifySuperclass(superclass, true/*rootOK*/)) {
        return nil;
    }

    cls = alloc_class_for_subclass(superclass, extraBytes);
    meta = alloc_class_for_subclass(superclass, extraBytes);

    objc_initializeClassPair_internal(superclass, name, cls, meta);
    return cls;
}
```

这就是 initialize 函数内部的实现，都是各种初始化代码，没有做其他逻辑操作。至此，类的初始化完成，可以在外面通过 class_addIvar 函数添加实例变量了。

```
static void objc_initializeClassPair_internal(Class superclass, const char *name, Class cls, Class meta)
{
    class_rw_t *cls_rw_w, *meta_rw_w;

    cls->setData((class_rw_t *)calloc(sizeof(class_rw_t), 1));
    meta->setData((class_rw_t *)calloc(sizeof(class_rw_t), 1));
    cls_rw_w = (class_rw_t *)calloc(sizeof(class_rw_t), 1);
    meta_rw_w = (class_rw_t *)calloc(sizeof(class_rw_t), 1);
    cls->data()->ro = cls_rw_w;
    meta->data()->ro = meta_rw_w;

    // Set basic info
    cls->data()->flags = RW_CONSTRUCTING | RW_COPIED_RO | RW_REALIZED | RW_REALIZING;
    meta->data()->flags = RW_CONSTRUCTING | RW_COPIED_RO | RW_REALIZED | RW_REALIZING;
    cls->data()->version = 0;
    meta->data()->version = 7;

    // ....
}
```

在创建类之后，会通过 objc_registerClassPair 函数注册新类。和创建新类一样，注册新类也分为注册类和注册元类。通过下面的 addNonMetaClass 函数注册元类，通过直接调

用 `NXMapInsert` 函数注册类。

```
void objc_registerClassPair(Class cls)
{
    cls->ISA()->changeInfo(RW_CONSTRUCTED, RW_CONSTRUCTING | RW_REALIZING);
    cls->changeInfo(RW_CONSTRUCTED, RW_CONSTRUCTING | RW_REALIZING);

    addNamedClass(cls, cls->data()->ro->name);
}

static void addNamedClass(Class cls, const char *name, Class replacing = nil)
{
    Class old;
    if ((old = getClass(name)) && old != replacing) {
        inform_duplicate(name, old, cls);
        addNonMetaClass(cls);
    } else {
        NXMapInsert(gdb_objc_realized_classes, name, cls);
    }
}
```

无论是注册类还是注册元类，内部都是通过 `NXMapInsert` 函数实现的。在 `Runtime` 中，所有类都是存在一个哈希表中的，在 `table` 的 `buckets` 中存储。每次新创建类之后，都需要把该类加入到哈希表中，下面是向哈希表插入的逻辑。

```
void *NXMapInsert(NXMapTable *table, const void *key, const void *value) {
    MapPair *pairs = (MapPair *)table->buckets;
    // 计算key在当前hash表中的下标, hash下标不一定是最后
    unsigned index = bucketOf(table, key);
    // 找到buckets的首地址, 并通过index下标计算对应位置, 获取到index对应的MapPair
    MapPair *pair = pairs + index;
    // 如果key为空, 则返回
    if (key == NX_MAPNOTKEY) {
        _objc_inform("!!! NXMapInsert: invalid key: -1\n");
        return NULL;
    }

    unsigned numBuckets = table->nbBucketsMinusOne + 1;
    // 如果当前地址未冲突, 则直接对pair赋值
    if (pair->key == NX_MAPNOTKEY) {
        pair->key = key; pair->value = value;
        table->count++;
        if (table->count * 4 > numBuckets * 3) _NXMapRehash(table);
        return NULL;
    }
}
```

```

/* 到这一步，则表示hash表冲突了 */

// 如果同名，则将旧类换为新类
if (isEqual(table, pair->key, key)) {
    const void *old = pair->value;
    if (old != value) pair->value = value;
    return (void *)old;

// hash表满了，对hash表做重哈希，然后再次执行这个函数
} else if (table->count == numBuckets) {
    /* no room: rehash and retry */
    _NXMapRehash(table);
    return NXMapInsert(table, key, value);

// hash表冲突了
} else {
    unsigned index2 = index;
    // 解决hash表冲突，这里采用的是线性探测法，解决哈希表冲突
    while ((index2 = nextIndex(table, index2)) != index) {
        pair = pairs + index2;
        if (pair->key == NX_MAPNOTAKEY) {
            pair->key = key; pair->value = value;
            table->count++;
            // 在查找过程中，发现哈希表不够用了，则进行重哈希
            if (table->count * 4 > numBuckets * 3) _NXMapRehash(table);
            return NULL;
        }
        // 找到同名类，则用新类替换旧类，并返回
        if (isEqual(table, pair->key, key)) {
            const void *old = pair->value;
            if (old != value) pair->value = value;
            return (void *)old;
        }
    }
    return NULL;
}
}

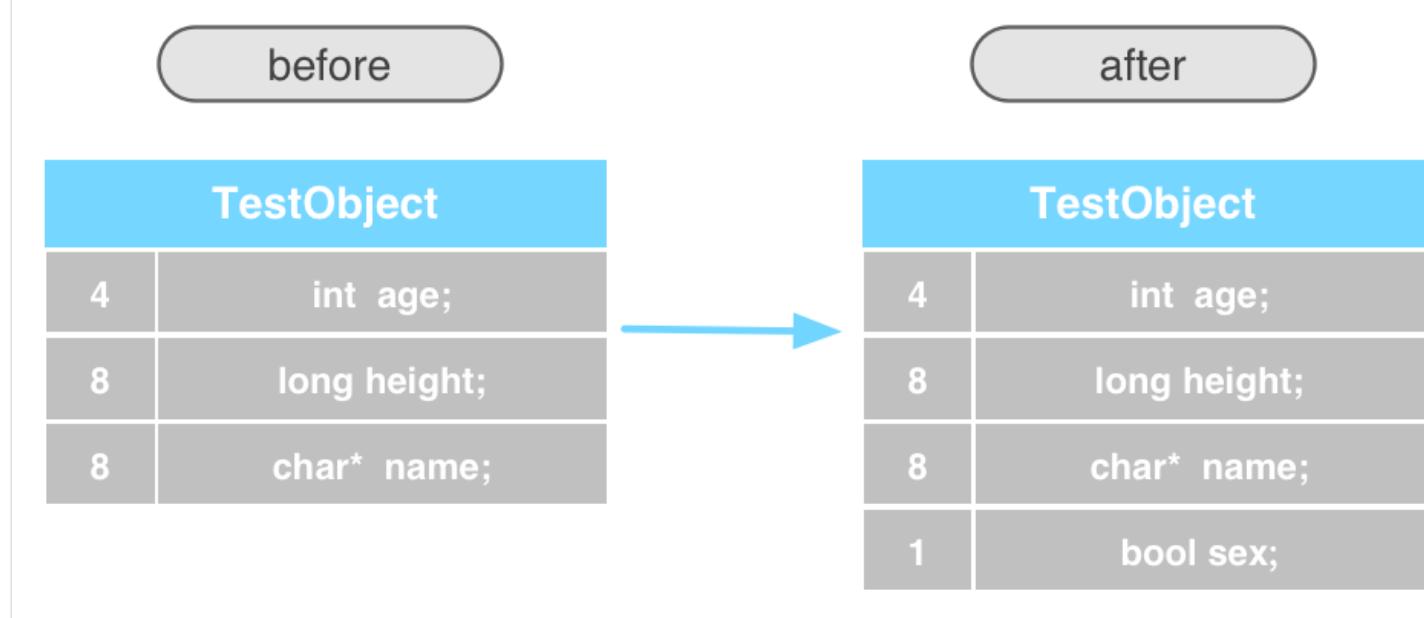
```

思考

那为什么只能向运行时动态创建的类添加 `ivars`，不能向已经存在的类添加 `ivars` 呢？

这是因为在编译时只读结构体 `class_ro_t` 就会被确定，在运行时是不可更改的。`ro` 结构体中有一个字段是 `instanceSize`，表示当前类在创建对象时需要多少空间，后面的创建都根据这个 `size` 分配类的内存。

如果对一个已经存在的类增加一个参数，改变了 `ivars` 的结构，这样在访问改变之前创建的对象时，就会出现问题。



以上图为例，在项目中创建 `TestObject` 类，并且添加三个成员变量，其 `ivars` 的内存结构占用 20字节。如果在运行时动态添加一个 `bool` 型参数，之后创建的对象 `ivars` 都占用21字节。

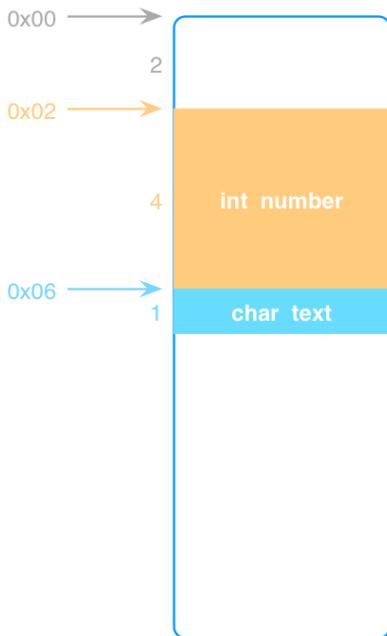
在通过 `ivars` 结构体访问之前创建的对象时，因为之前创建的对象没有 `sex`，所以还是按照20字节分配的内存空间，这时候访问 `sex` 就会导致地址越界。

数据访问

定义对象时都会给其设置类型，类型本质上并不是一个对象，而是用来标示当前对象所占空间的。以C语言为例，访问对象都是通过地址做访问的，而类型就是从首地址开始读取多少位是当前对象。

```
int number = 18;
char text = 'i';
```

以上面代码为例，定义了一个 `int` 类型的 `number`，占用四字节，定义一个 `char` 类型的 `text` 变量，占用一字节。在内存中访问对象时，就是根据指针地址找到对应的内存区，然后按照指针类型取多少范围的内存，就完成对象的读取操作。



而在面向对象语言中，函数或方法的命名规则还需要保留在运行期。以 C++ 为例，C++ 中有一个概念叫做“函数重载”，函数重载指的是允许有一组相同函数名，但参数列表类型不同的函数。

原函数: `void print(char c)`
 重载结果: `_ZN4test5printEc`

C++ 函数重载是有一定规则的，例如上面就是对 `print` 函数重载后的结果，重载结果才是运行时真正执行的函数。函数重载发生在编译期，会包含 `namespace`、`class name`、`function name`、返回值、参数等部分，根据这些部分重新生成函数名。

在OC中其实也存在函数重载的概念，只不过OC并不是直接对原有方法名做修改，而是增加对返回值和参数按照一定规则进行编码，然后放在 `method_t` 结构体中。

`method_t` 结构体存储着方法的信息，其中 `types` 字段就是返回值和参数的编码。编码后的字符串类似于 `"iv@:d"`，完整的编码规则可以查看[官方文档](#)。

下面就是 `Method` 的定义，主要包含了三个关键信息。

```
struct method_t {
    SEL name;
    const char *types;
    IMP imp;
};
```

Protocol

我们在项目中经常使用协议，那协议又是怎么实现的呢？

根据 Runtime 源码可以看出，协议都是 `protocol_t` 结构体的对象，而 `protocol_t` 结构体是继承自 `objc_object` 的，所以具备对象的特征。

除了 `objc_object` 中定义的一些结构体参数外，`protocol_t` 中还定义了一些独有的参数，例如常用的 `name`、`method list`、`property list`、`size` 等。所以可以看出，一个协议中可以声明对象方法、类方法，以及对象属性和类属性。

```
struct protocol_t : objc_object {
    const char *mangledName;
    struct protocol_list_t *protocols;
    method_list_t *instanceMethods;
    method_list_t *classMethods;
    method_list_t *optionalInstanceMethods;
    method_list_t *optionalClassMethods;
    property_list_t *instanceProperties;
    uint32_t size;    // sizeof(protocol_t)
    uint32_t flags;
    // Fields below this point are not always present on disk.
    const char **_extendedMethodTypes;
    const char *_demangledName;
    property_list_t *_classProperties;
};
```

既然具备了对象的特征，那也是有 `isa` 指针的。在 `Protocol` 中所有的 `isa` 都指向同一个类 `Protocol`。在 `Protocol` 类中没有做太复杂的处理，只是实现了一些基础的方法。

```
@implementation Protocol

+ (void) load {

}

- (BOOL) conformsTo: (Protocol *)aProtocolObj {
    return protocol_conformsToProtocol(self, aProtocolObj);
}

- (struct objc_method_description *) descriptionForInstanceMethod:(SEL)aSel {
    return method_getDescription(protocol_getMethod((struct protocol_t *)self,
                                                    aSel, YES, YES, YES));
}

- (struct objc_method_description *) descriptionForClassMethod:(SEL)aSel {
    return method_getDescription(protocol_getMethod((struct protocol_t *)self,
```

```
    aSel, YES, NO, YES));  
}  
  
- (const char *)name {  
    return protocol_getName(self);  
}  
  
// Protocol 重写了isEqual方法, 内部不断查找其父类, 判断是否Protocol的子类。  
- (BOOL)isEqual:other {  
    Class cls;  
    Class protoClass = objc_getClass("Protocol");  
    for (cls = object_getClass(other); cls; cls = cls->superclass) {  
        if (cls == protoClass) break;  
    }  
    if (!cls) return NO;  
    // check equality  
    return protocol_isEqual(self, other);  
}  
  
- (NSUInteger)hash {  
    return 23;  
}  
  
@end
```

协议的初始化也是在 `_read_images` 函数中完成的，初始化过程主要是一个遍历。逻辑就是获取 `Protocol list`，然后遍历这个数组，并调用 `readProtocol` 函数进行初始化操作。

```
// 遍历所有协议列表，并且将协议列表加载到Protocol的哈希表中
for (EACH_HEADER) {
    extern objc_class_OBJC_CLASS_$_Protocol;
    // cls = Protocol类，所有协议和对象的结构体都类似，isa都对应Protocol类
    Class cls = (Class)&OBJC_CLASS_$_Protocol;
    assert(cls);
    // 获取protocol哈希表
    NXMapTable *protocol_map = protocols();
    bool isPreoptimized = hi->isPreoptimized();
    bool isBundle = hi->isBundle();

    // 从编译器中读取并初始化Protocol
    protocol_t **protolist = _getObjc2ProtocolList(hi, &count);
    for (i = 0; i < count; i++) {
        readProtocol(protolist[i], cls, protocol_map,
                    isPreoptimized, isBundle);
    }
}
```

在 `readProtocol` 函数中，会根据传入的协议进行初始化操作。在传入参数中，`protocol_class` 就是 `Protocol` 类，所有的协议类的 `isa` 都指向这个类。

根据 `Protocol` 的源码可以看出，其对象模型是比较简单的，和 `Class` 的对象模型还不太一样。`Protocol` 的对象模型只有从 `Protocol list` 中加载的对象和 `isa` 指向的 `Protocol` 类构成，没有其他的实例化过程，`Protocol` 类并没有元类。

```
// 初始化传入的所有Protocol，如果哈希表中已经存在初始化的Protocol，则不做任何处理
static void
readProtocol(protocol_t *newproto, Class protocol_class,
             NXMapTable *protocol_map,
             bool headerIsPreoptimized, bool headerIsBundle)
{
    auto insertFn = headerIsBundle ? NXMapKeyCopyingInsert : NXMapInsert;
    // 根据名字获得对应的Protocol对象
    protocol_t *oldproto = (protocol_t *)getProtocol(newproto->mangledName);

    // 如果Protocol不为NULL，表示已经存在相同的Protocol，则不做任何处理，进入下面if语句。
    if (oldproto) {
        // nothing
    }
    // 如果Protocol为NULL，则对其进行简单的初始化，并将Protocol的isa设置为Protocol类
    else if (headerIsPreoptimized) {
        protocol_t *cacheproto = (protocol_t *)
            getPreoptimizedProtocol(newproto->mangledName);
        protocol_t *installedproto;
        if (cacheproto && cacheproto != newproto) {
            installedproto = cacheproto;
        }
        else {
            installedproto = newproto;
        }
        // 哈希表插入函数的指针
        insertFn(protocol_map, installedproto->mangledName,
                 installedproto);
    }
    // 下面两个else都是初始化protocol_t的过程
    else if (newproto->size >= sizeof(protocol_t)) {
        newproto->initIsa(protocol_class);
        insertFn(protocol_map, newproto->mangledName, newproto);
    }
    else {
        size_t size = max(sizeof(protocol_t), (size_t)newproto->size);
        protocol_t *installedproto = (protocol_t *)calloc(size, 1);
        memcpy(installedproto, newproto, newproto->size);
        installedproto->size = (__typeof__(installedproto->size))size;
    }
}
```

```
    installedproto->initIsa(protocol_class);
    insertFn(protocol_map, installedproto->mangledName, installedproto);
}
}
```

Protocol是可以在运行时动态创建添加的，和创建Class的过程类似，分为创建和注册两部分。创建 Protocol 之后，Protocol 处于一个未完成的状态，只有注册后才是可以使用的 Protocol 。

```
// 创建新的Protocol，创建后还需要调用下面的register方法
Protocol *
objc_allocateProtocol(const char *name)
{
    if (getProtocol(name)) {
        return nil;
    }

    protocol_t *result = (protocol_t *)calloc(sizeof(protocol_t), 1);

    // 下面的cls是__IncompleteProtocol类，表示是未完成的Protocol
    extern objc_class_OBJC_CLASS_$_IncompleteProtocol;
    Class cls = (Class)&OBJC_CLASS_$_IncompleteProtocol;
    result->initProtocolIsa(cls);
    result->size = sizeof(protocol_t);
    result->mangledName = strdupIfMutable(name);

    return (Protocol *)result;
}
```

注册 Protocol 。

```
// 向protocol的哈希表中，注册新创建的Protocol对象
void objc_registerProtocol(Protocol *proto_gen)
{
    protocol_t *proto = newprotocol(proto_gen);

    extern objc_class_OBJC_CLASS_$_IncompleteProtocol;
    Class oldcls = (Class)&OBJC_CLASS_$_IncompleteProtocol;
    extern objc_class_OBJC_CLASS_$_Protocol;
    Class cls = (Class)&OBJC_CLASS_$_Protocol;

    // 如果已经被注册到哈希表中，则直接返回
    if (proto->ISA() == cls) {
        return;
    }
```

```
// 如果当前proto的isa不是__IncompleteProtocol, 表示这个proto是有问题的, 则返回
if (proto->ISA() != oldcls) {
    return;
}
proto->changeIsa(cls);
NXMapKeyCopyingInsert(protocols(), proto->mangledName, proto);
}
```

SEL

之前 SEL 是由 `objc_selector` 结构体实现的，但是从现在的源码来看，SEL 是一个 `const char*` 的常量字符串，只是代表一个名字而已。

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

为什么说 SEL 只是一个常量字符串呢？我们在 Runtime 源码中探究一下。

这是在 `_read_images` 函数中 SEL list 的实现，主要逻辑是加载 SEL list 到内存中，然后通过 `sel_registerNameNoLock` 函数，将所有 SEL 都注册到属于 SEL 的哈希表中。

但是我们从这段代码中可以看出，大部分的 SEL 和 `const char*` 的转换，都是直接进行强制类型转换的，所以二者是同一块内存。

```
// 将所有SEL都注册到哈希表中, 是另外一张哈希表
static size_t UnfixedSelectors;
sel_lock();
for (EACH_HEADER) {
    if (hi->isPreoptimized()) continue;

    bool isBundle = hi->isBundle();
    // 取出的是字符串数组, 例如首地址是"class"
    SEL *sels = _getObjc2SelectorRefs(hi, &count);
    UnfixedSelectors += count;
    for (i = 0; i < count; i++) {
        // sel_cname函数内部就是将SEL强转为常量字符串
        const char *name = sel_cname(sels[i]);
        // 注册SEL的操作
        sels[i] = sel_registerNameNoLock(name, isBundle);
    }
}
```

再进入 `sel_registerNameNoLock` 函数中可以看出，SEL 的哈希表也是将字符串注册到哈希表中，

并不是之前的 `objc_selector` 结构体，所以可以看出现在 `SEL` 就是单纯的 `const char*` 常量字符串。

```
static SEL sel_alloc(const char *name, bool copy)
{
    return (SEL)(copy ? strdupIfMutable(name) : name);
}
```

对等交换协议

研究 Apple 的源码时，还可以通过 `GNUSStep` 研究，`GNUSStep` 是苹果的一套对等交换源码，将OC代码以重新实现了一遍，内部实现大致和苹果的类似。

[GNUSStep](#)

Runtime加载过程

程序加载过程

在iOS程序中会用到很多系统的动态库，这些动态库都是动态加载的。所有iOS程序共用一套系统动态库，在程序开始运行时才会开始链接动态库。

▼ Linked Frameworks and Libraries

Name	Status
libAFNetworking.a	Required ▾
SystemConfiguration.framework	Required ▾
libz.tbd	Required ▾
libc++.tbd	Required ▾
CoreTelephony.framework	Required ▾

除了在项目设置里显式出现的动态库外，还会有一些隐式存在的动态库。例如 `objc` 和 `Runtime` 所属的 `libobjc.dylib` 和 `libSystem.dylib`，在 `libSystem` 中包含常用的 `libdispatch(GCD)`、`libsystem_c` (C语言基础库)、`libsystem_blocks(Block)` 等。

使用动态库的优点：

1. 防止重复。iOS系统中所有 App 公用一套系统动态库，防止重复的内存占用。
2. 减少包体积。因为系统动态库被内置到iOS系统中，所以打包时不需要把这部分代码打进去，可以减小包体积。
3. 动态性。因为系统动态库是动态加载的，所以可以在更新系统后，将动态库换成新的动态库。

加载过程

在应用程序启动后，由 `dyld` (the dynamic link editor) 进行程序的初始化操作。大概流程就像下面列出的步骤，其中第3、4、5步会执行多次，在 `ImageLoader` 加载新的 `image` 进内存后就会执行一次。

1. 在应用程序启动后，由 `dyld` 将应用程序加载到二进制中，并完成一些文件的初始化操作。
2. `Runtime` 向 `dyld` 中注册回调函数。
3. 通过 `ImageLoader` 将所有 `image` 加载到内存中。
4. `dyld` 在 `image` 发生改变时，主动调用回调函数。
5. `Runtime` 接收到 `dyld` 的函数回调，开始执行 `map_images`、`load_images` 等操作，并回

调 `+load` 方法。

6. 调用 `main()` 函数，开始执行业务代码。

`ImageLoader` 是 `image` 的加载器，`image` 可以理解为编译后的二进制。

下面是在 `Runtime` 的 `map_images` 函数打断点，观察回调情况的汇编代码。可以看出，调用是由 `dyld` 发起的，由 `ImageLoader` 通知 `dyld` 进行调用。



```
283 0x100009c40 <+1034>: addq $0x18, %r15
284 0x100009c44 <+1038>: decq %r14
285 0x100009c47 <+1041>: jne 0x100009c0e ; <+984>
286 0x100009c49 <+1043>: testl %r13d, %r13d
287 0x100009c4c <+1046>: je 0x100009c77 ; <+1089>
288 0x100009c4e <+1048>: callq 0x10003560d ; mach_absolute_time
289 0x100009c53 <+1053>: movq %rax, %r14
290 0x100009c56 <+1056>: movl %r13d, %edi
291 0x100009c59 <+1059>: movq %rbx, %rsi
292 0x100009c5c <+1062>: movq %r12, %rdx
293 0x100009c5f <+1065>: callq *0x492b3(%rip) ; dyld::sNotifyObjCMapped
294 0x100009c65 <+1071>: callq 0x10003560d ; mach_absolute_time Thread 1: br
295 0x100009c6a <+1076>: subq %r14, %rax
296 0x100009c6d <+1079>: leaq 0x7d88c(%rip), %rcx ; ImageLoader::fgTotalObjCSetupTime
297 0x100009c74 <+1086>: addq %rax, (%rcx)
298 0x100009c77 <+1089>: movq -0x68(%rbp), %rsp
299 0x100009c7b <+1093>: movl -0x50(%rbp), %r15d
```

关于 `dyld` 我并没有深入研究，有兴趣的同学可以到[Github](#)上下载源码研究一下。

动态加载

一个OC程序可以在运行过程中动态加载和链接新类或 `Category`，新类或 `Category` 会加载到程序中，其处理方式和其他类是相同的。动态加载还可以做许多不同的事，动态加载允许应用程序进行自定义处理。

OC提供了 `objc_loadModules` 运行时函数，执行 `Mach-O` 中模块的动态加载，在上层 `NSBundle` 对象提供了更简单的访问 `API`。

map images

在 `Runtime` 加载时，会调用 `_objc_init` 函数，并在内部注册三个函数指针。其中 `map_images` 函数是初始化的关键，内部完成了大量 `Runtime` 环境的初始化操作。

在 `map_images` 函数中，内部也是做了一个调用中转。然后调用到 `map_images_nolock` 函数，内部核心就是 `_read_images` 函数。

```
void _objc_init(void)
{
    // .... 各种init
    _dyld_objc_notify_register(&map_images, load_images, unmap_image);
}
```

```

void map_images(unsigned count, const char * const paths[],
               const struct mach_header * const mhdrs[])
{
    rwlock_writer_t lock(runtimeLock);
    return map_images_nolock(count, paths, mhdrs);
}

void map_images_nolock(unsigned mhCount, const char * const mhPaths[],
                      const struct mach_header * const mhdrs[])
{
    if (hCount > 0) {
        _read_images(hList, hCount, totalClasses, unoptimizedTotalClasses);
    }
}

```

在 `_read_images` 函数中完成了大量的初始化操作，函数内部代码量比较大，下面是精简版带注释的源代码。

先整体梳理一遍 `_read_images` 函数内部的逻辑：

1. 加载所有类到类的 `gdb_objc_realized_classes` 表中。
2. 对所有类做重映射。
3. 将所有 `SEL` 都注册到 `namedSelectors` 表中。
4. 修复函数指针遗留。
5. 将所有 `Protocol` 都添加到 `protocol_map` 表中。
6. 对所有 `Protocol` 做重映射。
7. 初始化所有非懒加载的类，进行 `rw`、`ro` 等操作。
8. 遍历已标记的懒加载的类，并做初始化操作。
9. 处理所有 `Category`，包括 `Class` 和 `Meta Class`。
10. 初始化所有未初始化的类。

```

void _read_images(header_info **hList, uint32_t hCount, int totalClasses, int unopt
imizedTotalClasses)
{
    header_info *hi;
    uint32_t hIndex;
    size_t count;
    size_t i;
    Class *resolvedFutureClasses = nil;
    size_t resolvedFutureClassCount = 0;
    static bool doneOnce;
    TimeLogger ts(PrintImageTimes);

#define EACH_HEADER \

```

```

hIndex = 0;           \
hIndex < hCount && (hi = hList[hIndex]); \
hIndex++

if (!doneOnce) {
    doneOnce = YES;
    // 实例化存储类的哈希表，并且根据当前类数量做动态扩容
    int namedClassesSize =
        (isPreoptimized() ? unoptimizedTotalClasses : totalClasses) * 4 / 3;
    gdb_objc_realized_classes =
        NXCreateMapTable(NXStrValueMapPrototype, namedClassesSize);
}

// 由编译器读取类列表，并将所有类添加到类的哈希表中，并且标记懒加载的类并初始化内存空间
for (EACH_HEADER) {
    if (!mustReadClasses(hi)) {
        continue;
    }

    bool headerIsBundle = hi->isBundle();
    bool headerIsPreoptimized = hi->isPreoptimized();

    /** 将新类添加到哈希表中 */

    // 从编译后的类列表中取出所有类，获取到的是一个classref_t类型的指针
    classref_t *classlist = _getObjc2classList(hi, &count);
    for (i = 0; i < count; i++) {
        // 数组中会取出OS_dispatch_queue_concurrent、OS_xpc_object、NSRunLoop等系统类，例如CF、Foundation、Libdispatch中的类。以及自己创建的类
        Class cls = (Class)classlist[i];
        // 通过readClass函数获取处理后的新类，内部主要操作ro和rw结构体
        Class newCls = readClass(cls, headerIsBundle, headerIsPreoptimized);

        // 初始化所有懒加载的类需要的内存空间
        if (newCls != cls && newCls) {
            // 将懒加载的类添加到数组中
            resolvedFutureClasses = (Class *)
                realloc(resolvedFutureClasses,
                        (resolvedFutureClassCount+1) * sizeof(Class));
            resolvedFutureClasses[resolvedFutureClassCount++] = newCls;
        }
    }
}

// 将未映射Class和Super Class重映射，被remap的类都是非懒加载的类
if (!noClassesRemapped()) {
    for (EACH_HEADER) {
        // 重映射Class，注意是从_getObjc2ClassRefs函数中取出类的引用
    }
}

```

```

        Class *classrefs = _getObjc2ClassRefs(hi, &count);
        for (i = 0; i < count; i++) {
            remapClassRef(&classrefs[i]);
        }
        // 重映射父类
        classrefs = _getObjc2SuperRefs(hi, &count);
        for (i = 0; i < count; i++) {
            remapClassRef(&classrefs[i]);
        }
    }

// 将所有SEL都注册到哈希表中，是另外一张哈希表
static size_t UnfixedSelectors;
sel_lock();
for (EACH_HEADER) {
    if (hi->isPreoptimized()) continue;

    bool isBundle = hi->isBundle();
    SEL *sels = _getObjc2SelectorRefs(hi, &count);
    UnfixedSelectors += count;
    for (i = 0; i < count; i++) {
        const char *name = sel_cname(sels[i]);
        // 注册SEL的操作
        sels[i] = sel_registerNameNoLock(name, isBundle);
    }
}

// 修复旧的函数指针调用遗留
for (EACH_HEADER) {
    message_ref_t *refs = _getObjc2MessageRefs(hi, &count);
    if (count == 0) continue;
    for (i = 0; i < count; i++) {
        // 内部将常用的alloc、objc_msgSend等函数指针进行注册，并fix为新的函数指针
        fixupMessageRef(refs+i);
    }
}

// 遍历所有协议列表，并且将协议列表加载到Protocol的哈希表中
for (EACH_HEADER) {
    extern objc_class_OBJC_CLASS_$_Protocol;
    // cls = Protocol类，所有协议和对象的结构体都类似，isa都对应Protocol类
    Class cls = (Class)&OBJC_CLASS_$_Protocol;
    assert(cls);
    // 获取protocol哈希表
    NXMapTable *protocol_map = protocols();
    bool isPreoptimized = hi->isPreoptimized();
    bool isBundle = hi->isBundle();
}

```

```

// 从编译器中读取并初始化Protocol
protocol_t **protolist = _objc2ProtocolList(hi, &count);
for (i = 0; i < count; i++) {
    readProtocol(protolist[i], cls, protocol_map,
                 isPreoptimized, isBundle);
}
}

// 修复协议列表引用, 优化后的images可能是正确的, 但是并不确定
for (EACH_HEADER) {
    // 需要注意到是, 下面的函数是_getobjc2ProtocolRefs, 和上面的_getobjc2ProtocolList不一样
    protocol_t **protolist = _getobjc2ProtocolRefs(hi, &count);
    for (i = 0; i < count; i++) {
        remapProtocolRef(&protolist[i]);
    }
}

// 实现非懒加载的类, 对于Load方法和静态实例变量
for (EACH_HEADER) {
    classref_t *classlist =
        _objc2NonlazyClassList(hi, &count);
    for (i = 0; i < count; i++) {
        Class cls = remapClass(classlist[i]);
        if (!cls) continue;
        // 实现所有非懒加载的类(实例化类对象的一些信息, 例如rw)
        realizeClass(cls);
    }
}

// 遍历resolvedFutureClasses数组, 实现所有懒加载的类
if (resolvedFutureClasses) {
    for (i = 0; i < resolvedFutureClassCount; i++) {
        // 实现懒加载的类
        realizeClass(resolvedFutureClasses[i]);
        resolvedFutureClasses[i]->setInstancesRequireRawIsa(false/*inherited*/)
    }
    free(resolvedFutureClasses);
}

// 发现和处理所有Category
for (EACH_HEADER) {
    // 外部循环遍历找到当前类, 查找类对应的Category数组
    category_t **catlist =
        _objc2CategoryList(hi, &count);
    bool hasClassProperties = hi->info()->hasCategoryClassProperties();
}

```

```

// 内部循环遍历当前类的所有Category
for (i = 0; i < count; i++) {
    category_t *cat = catlist[i];
    Class cls = remapClass(cat->cls);

    // 首先，通过其所属的类注册Category。如果这个类已经被实现，则重新构造类的方法
    // 列表。
    bool classExists = NO;
    if (cat->instanceMethods || cat->protocols
        || cat->instanceProperties)
    {
        // 将Category添加到对应Class的value中，value是Class对应的所有category
        // 数组
        addUnattachedCategoryForClass(cat, cls, hi);
        // 将Category的method、protocol、property添加到Class
        if (cls->isRealized()) {
            remethodizeClass(cls);
            classExists = YES;
        }
    }
}

// 这块和上面逻辑一样，区别在于这块是对Meta Class做操作，而上面则是对Class做
// 操作
// 根据下面的逻辑，从代码的角度来说，是可以对原类添加Category的
if (cat->classMethods || cat->protocols
    || (hasClassProperties && cat->_classProperties))
{
    addUnattachedCategoryForClass(cat, cls->ISA(), hi);
    if (cls->ISA()->isRealized()) {
        remethodizeClass(cls->ISA());
    }
}
}

// 初始化从磁盘中加载的所有类，发现Category必须是最后执行的
// 从runtime objc4-532版本源码来看，DebugNonFragileIvars字段一直是-1，所以不会进入
// 这个方法中
if (DebugNonFragileIvars) {
    realizeAllClasses();
}
#endif EACH_HEADER
}

```

其内部还调用了很多其他函数，后面会详细介绍函数内部实现。

load images

在项目中经常用到 `load` 类方法，`load` 类方法的调用时机比 `main` 函数还要靠前。`load` 方法是由系统来调用的，并且在整个程序运行期间，只会调用一次，所以可以在 `load` 方法中执行一些只执行一次的操作。

一般 `Method Swizzling` 都会放在 `load` 方法中执行，这样在执行 `main` 函数前，就可以对类方法进行交换。可以确保正式执行代码时，方法肯定是被交换过的。

如果对一个类添加 **Category** 后，并且重写其原有方法，这样会导致 **Category** 中的方法覆盖原类的方法。但是 `load` 方法却是例外，所有 **Category** 和原类的 `load` 方法都会被执行。

源码分析

`load` 方法由 `Runtime` 进行调用，下面我们分析一下 `load` 方法的实现，`load` 的实现源码都在 `objc-loadmethod.mm` 中。

在一个新的工程中，我们创建一个 `TestObject` 类，并在其 `load` 方法中打一个断点，看一下系统的调用堆栈。

```
0 +[TestObject load]
1 call_class_loads()
2 ::call_load_methods()
3 ::load_images(const char *, const mach_header *)
4 dyld::notifySingle(dyld_image_states, ImageLoader const*, ImageLoader::InitializerTimingList*)
11 _dyld_start
```

从调用栈可以看出，是通过系统的动态链接器 `dyld` 开始的调用，然后调到 `Runtime` 的 `load_images` 函数中。`load_images` 函数是通过 `_dyld_objc_notify_register` 函数，将自己的函数指针注册给 `dyld` 的。

```
void _objc_init(void)
{
    static bool initialized = false;
    if (initialized) return;
    initialized = true;

    // fixme defer initialization until an objc-using image is found?
    environ_init();
    tls_init();
    static_init();
    lock_init();
```

```
exception_init();

_dyld_objc_notify_register(&map_images, load_images, unmap_image);
}
```

在 `load_images` 函数中主要做了两件事，首先通过 `prepare_load_methods` 函数准备 `Class load list` 和 `Category load list`，然后通过 `call_load_methods` 函数调用已经准备好的两个方法列表。

```
void
load_images(const char *path __unused, const struct mach_header *mh)
{
    if (!hasLoadMethods((const headerType *)mh)) return;
    prepare_load_methods((const headerType *)mh);
    call_load_methods();
}
```

首先我们看一下 `prepare_load_methods` 函数的实现，看一下其内部是怎么查找 `load` 方法的。可以看到，其内部主要分为两部分，查找 `Class` 的 `load` 方法列表和查找 `Category` 方法列表。

准备类的方法列表时，首先通过 `_getObjc2NonlazyClassList` 函数获取所有非懒加载类的列表，这时候获取到的是一个 `classref_t` 类型的数组，然后遍历数组添加 `load` 方法列表。

`Category` 过程也是类似，通过 `_getObjc2NonlazyCategoryList` 函数获取所有非懒加载 `Category` 的列表，得到一个 `category_t` 类型的数组，需要注意的是这是一个指向指针的指针。然后对其进行遍历并添加到 `load` 方法列表，`Class` 和 `Category` 的 `load` 方法列表是两个列表。

```
void prepare_load_methods(const headerType *mhdr)
{
    size_t count, i;

    // 获取到非懒加载的类的列表
    classref_t *classlist =
        _getObjc2NonlazyClassList(mhdr, &count);
    for (i = 0; i < count; i++) {
        // 设置Class的调用列表
        schedule_class_load(remapClass(classlist[i]));
    }

    // 获取到非懒加载的Category列表
    category_t **categorylist = _getObjc2NonlazyCategoryList(mhdr, &count);
    for (i = 0; i < count; i++) {
        category_t *cat = categorylist[i];
    }
}
```

```

    Class cls = remapClass(cat->cls);
    // 忽略弱链接的类别
    if (!cls) continue;
    // 实例化所属的类
    realizeClass(cls);
    // 设置Category的调用列表
    add_category_to_loadable_list(cat);
}
}

```

在添加类的 `load` 方法列表时，内部会递归遍历把所有父类的 `load` 方法都添加进去，顺着继承者链的顺序添加，会先把父类添加在前面。然后会调用 `add_class_to_loadable_list` 函数，将自己的 `load` 方法添加到对应的数组中。

```

static void schedule_class_load(Class cls)
{
    if (!cls) return;
    // 已经添加Class的Load方法到调用列表中
    if (cls->data()->flags & RW_LOADED) return;

    // 确保super已经被添加到Load列表中，默认是整个继承者链的顺序
    schedule_class_load(cls->superclass);

    // 将IMP和Class添加到调用列表
    add_class_to_loadable_list(cls);
    // 设置Class的Flags，表示已经添加Class到调用列表中
    cls->setInfo(RW_LOADED);
}

```

而 `Category` 则不需要考虑父类的问题，所以直接在 `prepare_load_methods` 函数中遍历 `Category` 数组，然后调用 `add_category_to_loadable_list` 函数即可。

在 `add_category_to_loadable_list` 函数中，会判断当前 `Category` 有没有实现 `load` 方法，如果没有则直接 `return`，如果实现了则添加到 `loadable_categories` 数组中。

类的 `add_class_to_loadable_list` 函数内部实现也是类似，区别在于类的数组叫做 `loadable_classes`。

```

void add_category_to_loadable_list(Category cat)
{
    IMP method;

    // 获取Category的Load方法的IMP
    method = _category_getLoadMethod(cat);
}

```

```

// 如果Category没有Load方法则return
if (!method) return;
// 如果已使用大小等于数组大小, 对数组进行动态扩容
if (loadable_categories_used == loadable_categories_allocated) {
    loadable_categories_allocated = loadable_categories_allocated*2 + 16;
    loadable_categories = (struct loadable_category *)
        realloc(loadable_categories,
                loadable_categories_allocated *
                sizeof(struct loadable_category));
}
loadable_categories[loadable_categories_used].cat = cat;
loadable_categories[loadable_categories_used].method = method;
loadable_categories_used++;
}

```

到此为止, `loadable_classes` 和 `loadable_categories` 两个数组已经准备好了, `load_images` 会调用 `call_load_methods` 函数执行这些 `load` 方法。在这个方法中, `call_class_loads` 函数是负责调用类方法列表的, `call_category_loads` 负责调用 `Category` 的方法列表。

```

void call_load_methods(void)
{
    bool more_categories;
    void *pool = objc_autoreleasePoolPush();

    do {
        // 反复执行call_class_loads函数, 直到数组中没有可执行的Load方法
        while (loadable_classes_used > 0) {
            call_class_loads();
        }
        more_categories = call_category_loads();
    } while (loadable_classes_used > 0 || more_categories);

    objc_autoreleasePoolPop(pool);
    loading = NO;
}

```

下面是调用类方法列表的代码, 内部主要是通过对 `loadable_classes` 数组进行遍历, 并获取到 `loadable_class` 的结构体, 结构体中存在 `Class` 和 `IMP`, 然后直接调用即可。

`Category` 的调用方式和类的一样, 就不在下面贴代码了。需要注意的是, `load` 方法都是直接调用的, 并没有走运行时的 `objc_msgSend` 函数。

```

static void call_class_loads(void)
{
    int i;
    struct loadable_class *classes = loadable_classes;
    int used = loadable_classes_used;
    loadable_classes = nil;
    loadable_classes_allocated = 0;
    loadable_classes_used = 0;

    for (i = 0; i < used; i++) {
        Class cls = classes[i].cls;
        load_method_t load_method = (load_method_t)classes[i].method;
        if (!cls) continue;
        (*load_method)(cls, SEL_load);
    }

    if (classes) free(classes);
}

struct loadable_class {
    Class cls; // may be nil
    IMP method;
};

```

根据上面的源码分析，我们可以看出 `load` 方法的调用顺序应该是“父类 -> 子类 -> 分类”的顺序。因为执行加载 `Class` 的时机是在 `Category` 之前的，而且 `load` 子类之前会先 `load` 父类，所以是这种顺序。

initialize

和 `load` 方法类似的也有 `initialize` 方法，`initialize` 方法也是由 `Runtime` 进行调用的，自己不可以直接调用。与 `load` 方法不同的是，`initialize` 方法是在第一次调用类所属的方法时，才会调用 `initialize` 方法，而 `load` 方法是在 `main` 函数之前就全部调用了。所以理论上来讲 `initialize` 可能永远都不会执行，如果当前类的方法永远不被调用的话。

下面我们研究一下 `initialize` 在 `Runtime` 中的源码。

在向对象发送消息时，`lookUpImpOrForward` 函数中会判断当前类是否被初始化，如果没有被初始化，则先进行初始化再调用类的方法。

```

IMP lookUpImpOrForward(Class cls, SEL sel, id inst, bool initialize, bool cache, bool resolver);

```

```

// ....省略好多代码

// 第一次调用当前类的话, 执行initialize的代码
if (initialize && !cls->isInitialized()) {
    _class_initialize (_class_getNonMetaClass(cls, inst));
}

// ....省略好多代码

```

在进行初始化的时候, 和 `load` 方法的调用顺序一样, 会按照继承者链先初始化父类。`_class_initialize` 函数中关键的两行代码是 `callInitialize` 和 `lockAndFinishInitializing` 的调用。

```

// 第一次调用类的方法, 初始化类对象
void _class_initialize(Class cls)
{
    Class supercls;
    bool reallyInitialize = NO;

    // 递归初始化父类。initizlize不用显式的调用super, 因为runtime已经在内部调用了
    supercls = cls->superclass;
    if (supercls && !supercls->isInitialized()) {
        _class_initialize(supercls);
    }

    {
        monitor_locker_t lock(classInitLock);
        if (!cls->isInitialized() && !cls->isInitializing()) {
            cls->setInitializing();
            reallyInitialize = YES;
        }
    }

    if (reallyInitialize) {
        _setThisThreadIsInitializingClass(cls);

        if (MultithreadedForkChild) {
            performForkChildInitialize(cls, supercls);
            return;
        }
        @try {
            // 通过objc_msgSend()函数调用initialize方法
            callInitialize(cls);
        }
        @catch (...) {
            @throw;
        }
    }
}

```

```

    }
    @finally {
        // 执行initialize方法后, 进行系统的initialize过程
        lockAndFinishInitializing(cls, supercls);
    }
    return;
}

else if (cls->isInitializing()) {
    if (_thisThreadIsInitializingClass(cls)) {
        return;
    } else if (!MultithreadedForkChild) {
        waitForInitializeToComplete(cls);
        return;
    } else {
        _setThisThreadIsInitializingClass(cls);
        performForkChildInitialize(cls, supercls);
    }
}
}
}

```

通过 `objc_msgSend` 函数调用 `initialize` 方法。

```

void callInitialize(Class cls)
{
    ((void*)(Class, SEL)objc_msgSend)(cls, SEL_initialize);
    asm("");
}

```

`lockAndFinishInitializing` 函数中会完成一些初始化操作, 其内部会调用 `_finishInitializing` 函数, 在函数内部会调用 `class` 的 `setInitialized` 函数, 核心工作都由 `setInitialized` 函数完成。

```

static void lockAndFinishInitializing(Class cls, Class supercls)
{
    monitor_locker_t lock(classInitLock);
    if (!supercls || supercls->isInitialized()) {
        _finishInitializing(cls, supercls);
    } else {
        _finishInitializingAfter(cls, supercls);
    }
}

```

负责初始化类和元类, 函数内部主要是查找当前类和元类中是否定义了某些方法, 然后根据查找

结果设置类和元类的一些标志位。

```
void
objc_class::setInitialized()
{
    Class metacls;
    Class cls;

    // 获取类和元类对象
    cls = (Class)this;
    metacls = cls->ISA();

    bool inherited;
    bool metaCustomAWZ = NO;
    if (MetaclassNSObjectAWZSwizzled) {
        metaCustomAWZ = YES;
        inherited = NO;
    }
    else if (metacls == classNSObject()->ISA()) {
        // 查找是否实现了alloc和allocWithZone方法
        auto& methods = metacls->data()->methods;
        for (auto mlists = methods.beginCategoryMethodLists(),
              end = methods.endCategoryMethodLists(metacls);
              mlists != end;
              ++mlists)
        {
            if (methodListImplementsAWZ(*mlists)) {
                metaCustomAWZ = YES;
                inherited = NO;
                break;
            }
        }
    }
    else if (metacls->superclass->hasCustomAWZ()) {
        metaCustomAWZ = YES;
        inherited = YES;
    }
    else {
        auto& methods = metacls->data()->methods;
        for (auto mlists = methods.beginLists(),
              end = methods.endLists();
              mlists != end;
              ++mlists)
        {
            if (methodListImplementsAWZ(*mlists)) {
                metaCustomAWZ = YES;
                inherited = NO;
            }
        }
    }
}
```

```
        break;
    }
}
}

if (!metaCustomAWZ) metacls->setHasDefaultAWZ();
if (PrintCustomAWZ && metaCustomAWZ) metacls->printCustomAWZ(inherited);

bool clsCustomRR = NO;
if (ClassNSObjectRRSwizzled) {
    clsCustomRR = YES;
    inherited = NO;
}

// 查找元类是否实现MRC方法, 如果是则进入if语句中
if (cls == classNSObject()) {
    auto& methods = cls->data()->methods;
    for (auto mlists = methods.beginCategoryMethodLists(),
          end = methods.endCategoryMethodLists(cls);
         mlists != end;
         ++mlists)
    {
        if (methodListImplementsRR(*mlists)) {
            clsCustomRR = YES;
            inherited = NO;
            break;
        }
    }
}

else if (!cls->superclass) {
    clsCustomRR = YES;
    inherited = NO;
}

else if (cls->superclass->hasCustomRR()) {
    clsCustomRR = YES;
    inherited = YES;
}

else {
    // 查找类是否实现MRC方法, 如果是则进入if语句中
    auto& methods = cls->data()->methods;
    for (auto mlists = methods.beginLists(),
          end = methods.endLists();
         mlists != end;
         ++mlists)
    {
        if (methodListImplementsRR(*mlists)) {
            clsCustomRR = YES;
            inherited = NO;
            break;
        }
    }
}
```

```
    }

    if (!clsCustomRR) cls->setHasDefaultRR();
    if (PrintCustomRR && clsCustomRR) cls->printCustomRR(inherited);
    metacls->changeInfo(RW_INITIALIZED, RW_INITIALIZING);
}
```

需要注意的是，`initialize` 方法和 `load` 方法不太一样，`Category` 中定义的 `initialize` 方法会覆盖原方法而不是像 `load` 方法一样都可以调用。

Runtime消息发送机制

方法调用

在OC中方法调用是通过 `Runtime` 实现的， `Runtime` 进行方法调用本质上是发送消息， 通过 `objc_msgSend()` 函数进行消息发送。

例如下面的OC代码会被转换为 `Runtime` 代码。

```
原方法: [object testMethod]  
转换后的调用: objc_msgSend(object, @selector(testMethod));
```

发送消息的第二个参数是一个 `SEL` 类型的参数，在项目里经常会出现，不同的类定义了相同的方法，这样就会有相同的 `SEL`。那么问题就来了，也是很多人博客里都问过的一个问题，不同类的 `SEL` 是同一个吗？

然而，事实是通过我们的验证，创建两个不同的类，并定义两个相同的方法，通过 `@selector()` 获取 `SEL` 并打印。我们发现 `SEL` 都是同一个对象，地址都是相同的。由此证明，不同类的相同 `SEL` 是同一个对象。

```
@interface TestObject : NSObject  
- (void)testMethod;  
@end  
  
@interface TestObject2 : NSObject  
- (void)testMethod;  
@end  
  
// TestObject2实现文件也一样  
@implementation TestObject  
- (void)testMethod {  
    NSLog(@"TestObject testMethod %p", @selector(testMethod));  
}  
@end  
  
// 结果:  
TestObject testMethod 0x100000f81  
TestObject2 testMethod 0x100000f81
```

在 `Runtime` 中维护了一个 `SEL` 的表，这个表存储 `SEL` 不按照类来存储，只要相同的 `SEL` 就会被看

做一个，并存储到表中。在项目加载时，会将所有方法都加载到这个表中，而动态生成的方法也会被加载到表中。

隐藏参数

我们在方法内部可以通过 `self` 获取到当前对象，但是 `self` 又是从哪来的呢？

方法实现的本质也是C函数，C函数除了方法传入的参数外，还会有两个默认参数，这两个参数在通过 `objc_msgSend()` 调用时也会传入。这两个参数在 `Runtime` 中并没有声明，而是在编译时自动生成的。

从 `objc_msgSend` 的声明中可以看出这两个隐藏参数的存在。

```
objc_msgSend(void /* id self, SEL op, ... */ )
```

- `self`，调用当前方法的对象。
- `_cmd`，当前被调用方法的 `SEL`。

虽然这两个参数在调用和实现方法中都没有明确声明，但是我们仍然可以使用它。响应对象就是 `self`，被调用方法的 `selector` 是 `_cmd`。

```
- (void)method {
    id target = getTheReceiver();
    SEL method = getTheMethod();

    if ( target == self || method == _cmd )
        return nil;
    return [target performSelector:method];
}
```

函数调用

一个对象被创建后，自身的类及其父类一直到 `NSObject` 类的部分，都会包含在对象的内存中，例如其父类的实例变量。当通过 `[super class]` 的方式调用其父类的方法时，会创建一个结构体。

```
struct objc_super { id receiver; Class class; };
```

对 `super` 的调用会被转化为 `objc_msgSendSuper()` 的调用，并在其内部调用 `objc_msgSend()` 函数。有一点需要注意，尽管是通过 `[super class]` 的方式调用的，但传入的 `receiver` 对象仍然

是 `self`，返回结果也是 `self` 的 `class`。由此可知，当前对象无论调用任何方法，`receiver`都是当前对象。

```
objc_msgSend(objc_super->receiver, @selector(class))
```

在 `objc_msg.s` 中，存在多个版本的 `objc_msgSend` 函数。内部实现逻辑大体一致，都是通过汇编实现的，只是根据不同的情况有不同的调用。

```
objc_msgSend
objc_msgSend_fpret
objc_msgSend_fp2ret
objc_msgSend_stret
objc_msgSendSuper
objc_msgSendSuper_stret
objc_msgSendSuper2
objc_msgSendSuper2_stret
```

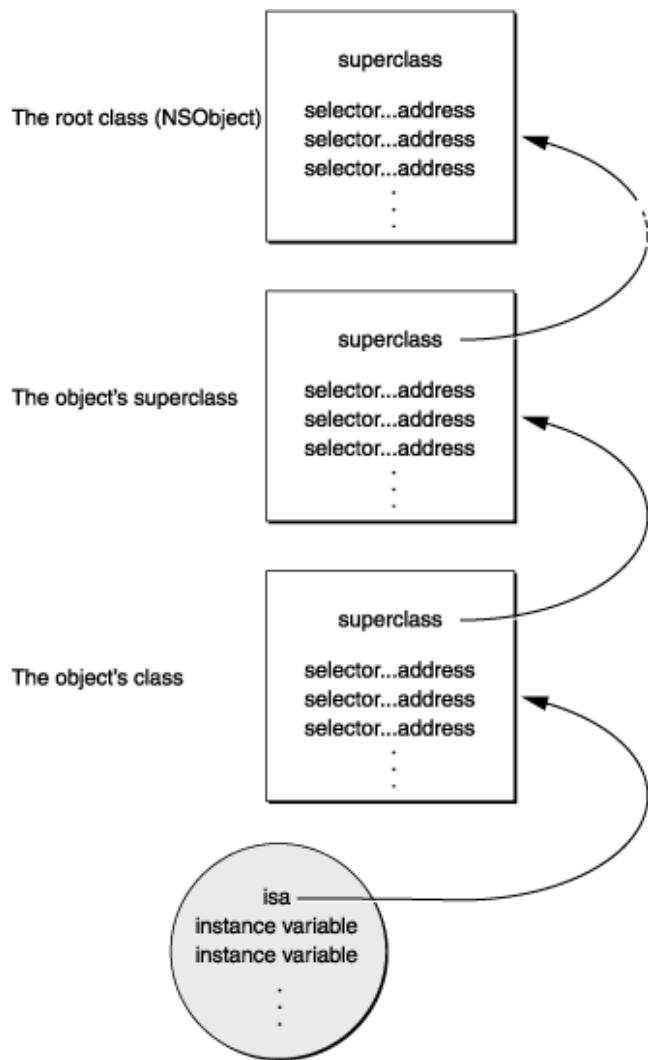
在上面源码中，带有 `super` 的会在外界传入一个 `objc_super` 的结构体对象。`stret` 表示返回的是 `struct` 类型，`super2` 是 `objc_msgSendSuper()` 的一种实现方式，不对外暴露。

```
struct objc_super {
    id receiver;
    Class class;
};
```

`fp` 则表示返回一个 `long double` 的浮点型，而 `fp2` 则返回一个 `complex long double` 的复杂浮点型，其他 `float`、`double` 的普通浮点型都用 `objc_msgSend`。除了上面这些情况外，其他都通过 `objc_msgSend()` 调用。

消息发送流程

当一个对象被创建时，系统会为其分配内存，并完成默认的初始化工作，例如对实例变量进行初始化。对象第一个变量是指向其类对象的指针- `isa`，`isa` 指针可以访问其类对象，并且通过其类对象拥有访问其所有继承者链中的类。



`isa` 指针不是语言的一部分，主要为 `Runtime` 机制提供服务。

当对象接收到一条消息时，消息函数随着对象 `isa` 指针到类的结构体中，在 `method list` 中查找方法 `selector`。如果在本类中找不到对应的 `selector`，则 `objc_msgSend` 会向其父类的 `method list` 中查找 `selector`，如果还不能找到则沿着继承关系一直向上查找，直到找到 `NSObject` 类。

`Runtime` 在 `selector` 查找的过程做了优化，为类的结构体中增加了 `cache` 字段，每个类都有独立的 `cache`，在一个 `selector` 被调用后就会加入到 `cache` 中。在每次搜索方法列表之前，都会先检查 `cache` 中有没有，如果没有才调用方法列表，这样会提高方法的查找效率。

如果通过OC代码的调用都会走消息发送的阶段，如果不想要消息发送的过程，可以获取到方法的函数指针直接调用。通过 `NSObject` 的 `methodForSelector:` 方法可以获取到函数指针，获取到指针后需要对指针进行类型转换，转换为和调用函数相符的函数指针，然后发起调用即可。

```

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);
```

实现原理

在 Runtime 中，`objc_msgSend` 函数也是开源的，但其是通过汇编代码实现的，`arm64` 架构代码可以在 `objc-msg-arm64.s` 中找到。在 Runtime 中，很多执行频率比较高的函数，都是用汇编写的。

`objc_msgSend` 并不是完全开源的，在 `_class_lookupMethodAndLoadCache3` 函数中已经获取到 `Class` 参数了。所以在下面中有一个肯定是对对象中获取 `isa_t` 的过程，从方法命名和注释来看，应该是 `GetIsaFast` 汇编命令。如果这样的话，就可以从消息发送到调用流程衔接起来了。

```
ENTRY _objc_msgSend
MESSENGER_START

NilTest NORMAL

GetIsaFast NORMAL          // r11 = self->isa
CacheLookup NORMAL          // calls IMP on success

NilTestSupport NORMAL

GetIsaSupport NORMAL

// cache miss: go search the method lists
LCacheMiss:
// isa still in r11
MethodTableLookup %a1, %a2 // r11 = IMP
cmp %r11, %r11           // set eq (nonstret) for forwarding
jmp *%r11                 // goto *imp

END_ENTRY _objc_msgSend
```

- `MESSENGER_START`：消息开始执行。
- `NilTest`：判断接收消息的对象是否为 `nil`，如果为 `nil` 则直接返回，这就是对 `nil` 发送消息无效的原因。
- `GetIsaFast`：快速获取到 `isa` 指向的对象，是一个类对象或元类对象。
- `CacheLookup`：从 `ache list` 中获取缓存 `selector`，如果查到则调用其对应的 `IMP`。
- `LCacheMiss`：缓存没有命中，则执行此条汇编下面的方法。
- `MethodTableLookup`：如果缓存中没有找到，则从 `method list` 中查找。

cache_t

如果每次进行方法调用时，都按照对象模型来进行方法列表的查找，这样是很消耗时间的。Runtime 为了优化调用时间，在 `objc_class` 中添加了一个 `cache_t` 类型的 `cache` 字段，通过缓存来优化调用时间。

在执行 `objc_msgSend` 函数的消息发送过程中，同一个方法第一次调用是没有缓存的，但调用之后就会存在缓存，之后的调用就直接调用缓存。所以方法的调用，可以分为有缓存和无缓存两种，这两种情况下的调用堆栈是不同的。

首先是从缓存中查找 `IMP`，但是由于 `cache3` 调用 `lookUpImpOrForward` 函数时，已经查找过 `cache` 了，所以传入的是 `NO`，不进入查找 `cahce` 的代码块中。

```
struct cache_t {
    // 存储被缓存方法的哈希表
    struct bucket_t *_buckets;
    // 占用的总大小
    mask_t _mask;
    // 已使用大小
    mask_t _occupied;
}

struct bucket_t {
    cache_key_t _key;
    IMP _imp;
};
```

当给一个对象发送消息时，Runtime 会沿着 `isa` 找到对应的类对象，但并不会立刻查找 `method_list`，而是先查找 `cache_list`，如果有缓存的话优先查找缓存，没有再查找方法列表。

这是 Runtime 对查找 `method` 的优化，理论上来说在 `cache` 中的 `method` 被访问的频率会更高。`cache_list` 由 `cache_t` 定义，内部有一个 `bucket_t` 的数组，数组中保存 `IMP` 和 `key`，通过 `key` 找到对应的 `IMP` 并调用。具体源码可以查看 `objc-cache.mm`。

如果类对象没有被初始化，并且 `lookUpImpOrForward` 函数的 `initialize` 参数为 `YES`，则表示需要对该类进行创建。函数内部主要是一些基础的初始化操作，而且会递归检查父类，如果父类未初始化，则先初始化其父类对象。

```
STATIC_ENTRY _cache_getImp

    mov r9, r0
    CacheLookup NORMAL
    // cache hit, IMP in r12
    mov r0, r12
```

```
bx lr          // return imp

CacheLookup2 GETIMP
// cache miss, return nil
mov r0, #0
bx lr

END_ENTRY _cache_getImp
```

下面会进入 `cache_getImp` 的代码中，然而这个函数不是开源的，但是有一部分源码可以看到，是通过汇编写的。其内部调用了 `CacheLookup` 和 `CacheLookup2` 两个函数，这两个函数也都是汇编写的。

经过第一次调用后，就会存在缓存。进入 `objc_msgSend` 后会调用 `CacheLookup` 命令，如果找到缓存则直接调用。但是 `Runtime` 并不是完全开源的，内部很多实现我们依然看不到，`CacheLookup` 命令内部也一样，只能看到调用完命令后就开始执行我们的方法了。

```
CacheLookup NORMAL, CALL
```

源码分析

在上面 `objc_msgSend` 汇编实现中，存在一个 `MethodTableLookup` 的汇编调用。在这条汇编调用中，调用了查找方法列表的C函数。下面是精简版代码。

```
.macro MethodTableLookup

// 调用MethodTableLookup并在内部执行cache3函数(C函数)
blx __class_lookupMethodAndLoadCache3
mov r12, r0          // r12 = IMP

.endmacro
```

在 `MethodTableLookup` 中通过调用 `_class_lookupMethodAndLoadCache3` 函数，来查找方法列表。函数内部是通过 `lookUpImpOrForward` 函数实现的，在调用时 `cache` 字段传入 `NO`，表示不需要查找缓存了，因为在 `cache3` 函数上面已经通过汇编查找过了。

```
IMP _class_lookupMethodAndLoadCache3(id obj, SEL sel, Class cls)
{
    // 通过cache3内部调用LookUpImpOrForward函数
    return lookUpImpOrForward(cls, sel, obj,
                               YES/*initialize*/, NO/*cache*/, YES/*resolver*/);
}
```

`lookUpImpOrForward` 函数是支持多线程的，所以内部会有很多锁操作。其内部有一个 `rwlock_t` 类型的 `runtimeLock` 变量，有 `runtimeLock` 控制读写锁。其内部有很多逻辑代码，这里把函数内部实现做了精简，把核心代码贴到下面。

通过类对象的 `isRealized` 函数，判断当前类是否被实现，如果没有被实现，则通过 `realizeClass` 函数实现该类。在 `realizeClass` 函数中，会设置 `version`、`rw`、`superClass` 等一些信息。

```
// 执行查找imp和转发的代码
IMP lookUpImpOrForward(Class cls, SEL sel, id inst,
                         bool initialize, bool cache, bool resolver)
{
    IMP imp = nil;
    bool triedResolver = NO;

    runtimeLock.assertUnlocked();

    // 如果cache是YES，则从缓存中查找IMP。如果是从cache3函数进来，则不会执行cache_getImp()
    // 函数
    if (cache) {
        // 通过cache_getImp函数查找IMP，查找到则返回IMP并结束调用
        imp = cache_getImp(cls, sel);
        if (imp) return imp;
    }

    runtimeLock.read();

    // 判断类是否已经被创建，如果没有被创建，则将类实例化
    if (!cls->isRealized()) {
        // 对类进行实例化操作
        realizeClass(cls);
    }

    // 第一次调用当前类的话，执行initialize的代码
    if (initialize && !cls->isInitialized()) {
        // 对类进行初始化，并开辟内存空间
        _class_initialize (_class_getNonMetaClass(cls, inst));
    }

retry:
    runtimeLock.assertReading();

    // 尝试获取这个类的缓存
    imp = cache_getImp(cls, sel);
    if (imp) goto done;
```

```

{
    // 如果没有从cache中查找到, 则从方法列表中获取Method
    Method meth = getMethodNoSuper_nolock(cls, sel);
    if (meth) {
        // 如果获取到对应的Method, 则加入缓存并从Method获取IMP
        log_and_fill_cache(cls, meth->imp, sel, inst, cls);
        imp = meth->imp;
        goto done;
    }
}

{
    unsigned attempts = unreasonableClassCount();
    // 循环获取这个类的缓存IMP 或 方法列表的IMP
    for (Class curClass = cls->superclass;
         curClass != nil;
         curClass = curClass->superclass)
    {
        if (--attempts == 0) {
            _objc_fatal("Memory corruption in class list.");
        }

        // 获取父类缓存的IMP
        imp = cache_getImp(curClass, sel);
        if (imp) {
            if (imp != (IMP)_objc_msgForward_impcache) {
                // 如果发现父类的方法, 并且不再缓存中, 在下面的函数中缓存方法
                log_and_fill_cache(cls, imp, sel, inst, curClass);
                goto done;
            }
            else {
                break;
            }
        }
    }

    // 在父类的方法列表中, 获取method_t对象。如果找到则缓存查找到的IMP
    Method meth = getMethodNoSuper_nolock(curClass, sel);
    if (meth) {
        log_and_fill_cache(cls, meth->imp, sel, inst, curClass);
        imp = meth->imp;
        goto done;
    }
}

// 如果没有找到, 则尝试动态方法解析
if (resolver && !triedResolver) {

```

```

        runtimeLock.unlockRead();
        _class_resolveMethod(cls, sel, inst);
        runtimeLock.read();
        triedResolver = YES;
        goto retry;
    }

    // 如果没有IMP被发现，并且动态方法解析也没有处理，则进入消息转发阶段
    imp = (IMP)_objc_msgForward_impcache;
    cache_fill(cls, sel, imp, inst);

done:
    runtimeLock.unlockRead();

    return imp;
}

```

在方法第一次调用时，可以通过 `cache_getImp` 函数查找到缓存的 `IMP`。但如果是第一次调用，就查不到缓存的 `IMP`，就会进入到 `getMethodNoSuper_nolock` 函数中执行。下面是 `getMethod` 函数的关键代码。

```

getMethodNoSuper_nolock(Class cls, SEL sel) {
    // 根据for循环，从methodList列表中，从头开始遍历，每次遍历后向后移动一位地址。
    for (auto mlists = cls->data()->methods.beginLists(),
        end = cls->data()->methods.endLists();
        mlists != end;
        ++mlists)
    {
        // 对sel参数和method_t做匹配，如果匹配上则返回。
        method_t *m = search_method_list(*mlists, sel);
        if (m) return m;
    }

    return nil;
}

```

当调用一个对象的方法时，查找对象的方法，本质上就是遍历对象 `isa` 所指向类的方法列表，并用调用方法的 `SEL` 和遍历的 `method_t` 结构体的 `name` 字段做对比，如果相等则将 `IMP` 函数指针返回。

```

// 根据传入的SEL，查找对应的method_t结构体
static method_t *search_method_list(const method_list_t *mlist, SEL sel)
{
    int methodListIsFixedUp = mlist->isFixedUp();

```

```

int methodListHasExpectedSize = mlist->entsize() == sizeof(method_t);

if (__builtin_expect(methodListIsFixedUp && methodListHasExpectedSize, 1)) {
    return findMethodInSortedMethodList(sel, mlist);
} else {
    for (auto& meth : *mlist) {
        // SEL本质上就是字符串, 查找的过程就是进行字符串对比
        if (meth.name == sel) return &meth;
    }
}

if (mlist->isFixedUp()) {
    for (auto& meth : *mlist) {
        if (meth.name == sel) {
            _objc_fatal("linear search worked when binary search did not");
        }
    }
}

return nil;
}

```

在 `getMethod` 函数中, 主要是对 `Class` 的 `methods` 方法列表进行查找和匹配。类的方法列表都在 `Class` 的 `class_data_bits_t` 中, 通过 `data()` 函数从 `bits` 中获取到 `class_rw_t` 的结构体, 然后获取到方法列表 `methods`, 并遍历方法列表。

如果从当前类中获取不到对应的 `IMP`, 则进入循环中。循环是从当前类出发, 沿着继承者链的关系, 一直向根类查找, 直到找到对应的 `IMP` 实现。

查找步骤和上面也一样, 先通过 `cache_getImp` 函数查找父类的缓存, 如果找到则调用对应的实现。如果没找到缓存, 表示第一次调用父类的方法, 则调用 `getMethodNoSuper_nolock` 函数从方法列表中获取实现。

```

for (Class curClass = cls->superclass;
     curClass != nil;
     curClass = curClass->superclass)
{
    imp = cache_getImp(curClass, sel);
    if (imp) {
        if (imp != (IMP)_objc_msgForward_impcache) {
            log_and_fill_cache(cls, imp, sel, inst, curClass);
            goto done;
        }
    }
}

Method meth = getMethodNoSuper_nolock(curClass, sel);

```

```
if (meth) {
    log_and_fill_cache(cls, meth->imp, sel, inst, curClass);
    imp = meth->imp;
    goto done;
}
}
```

如果没有找到方法实现，则会进入动态方法决议的步骤。在 `if` 语句中会判断传入的 `resolver` 参数是否为 `YES`，并且会判断是否已经有过动态决议，因为下面是 `goto retry`，所以这段代码可能会执行多次。

```
if (resolver && !triedResolver) {
    _class_resolveMethod(cls, sel, inst);
    triedResolver = YES;
    goto retry;
}
```

如果满足条件并且是第一次进行动态方法决议，则进入 `if` 语句中调用 `_class_resolveMethod` 函数。动态方法决议有两种，`_class_resolveClassMethod` 类方法决议和 `_class_resolveInstanceMethod` 实例方法决议。

```
BOOL (*msg)(Class, SEL, SEL) = (__typeof__(msg))objc_msgSend;
bool resolved = msg(cls, SEL_resolveInstanceMethod, sel);
```

在这两个动态方法决议的函数实现中，本质上都是通过 `objc_msgSend` 函数，调用 `NSObject` 中定义的 `resolveInstanceMethod:` 和 `resolveClassMethod:` 两个方法。

可以在这两个方法中动态添加方法，添加方法实现后，会在下面执行 `goto retry`，然后再次进入方法查找的过程中。从 `triedResolver` 参数可以看出，动态方法决议的机会只有一次，如果这次再没有找到，则进入消息转发流程。

```
imp = (IMP)_objc_msgForward_impcache;
cache_fill(cls, sel, imp, inst);
```

如果经过上面这些步骤，还是没有找到方法实现的话，则进入动态消息转发中。在动态消息转发中，还可以对没有实现的方法做一些弥补措施。

下面是通过 `objc_msgSend` 函数发送一条消息后，所经过的调用堆栈，调用顺序是从上到下的。

CacheLookup NORMAL, CALL

```
__objc_msgSend_uncached
MethodTableLookup NORMAL
_class_lookupMethodAndLoadCache3
lookUpImpOrForward
```

调用总结

在调用 `objc_msgSend` 函数后，会有一系列复杂的判断逻辑，总结如下。

1. 判断当前调用的 `SEL` 是否需要忽略，例如 Mac OS 中的垃圾处理机制启动的话，则忽略 `retain`、`release` 等方法，并返回一个 `_objc_ignored_method` 的 `IMP`，用来标记忽略。
2. 判断接收消息的对象是否为 `nil`，因为在OC中对 `nil` 发消息是无效的，这是因为在调用时就通过判断条件过滤掉了。
3. 从方法的缓存列表中查找，通过 `cache_getImp` 函数进行查找，如果找到缓存则直接返回 `IMP`。
4. 查找当前类的 `method list`，查找是否有对应的 `SEL`，如果有则获取到 `Method` 对象，并从 `Method` 对象中获取 `IMP`，并返回 `IMP` (这步查找结果是 `Method` 对象)。
5. 如果在当前类中没有找到 `SEL`，则去父类中查找。首先查找 `cache list`，如果缓存中没有则查找 `method list`，并以此类推直到查找到 `NSObject` 为止。
6. 如果在类的继承体系中，始终没有查找到对应的 `SEL`，则进入动态方法解析中。可以在 `resolveInstanceMethod` 和 `resolveClassMethod` 两个方法中动态添加实现。
7. 动态消息解析如果没有做出响应，则进入动态消息转发阶段。此时可以在动态消息转发阶段做一些处理，否则就会 `Crash`。

整体分析

总体可以被分为三部分：

1. 刚调用 `objc_msgSend` 函数后，内部的一些处理逻辑。
2. 复杂的查找 `IMP` 的过程，会涉及到 `cache list` 和 `method list` 等。
3. 进入消息转发阶段。

在 `cache list` 中找不到方法的情况下，会通过 `MethodTableLookup` 宏定义从类的方法列表中，查找对应的方法。在 `MethodTableLookup` 中本质上也是调用 `_class_lookupMethodAndLoadCache3` 函数，只是在传参时 `cache` 字段传 `NO`，表示不从 `cache list` 中查找。

在 `cache3` 函数中，是直接调用的 `lookUpImpOrForward` 函数，这个函数内部实现很复杂，可以看一下 [Runtime Analyze](#)。在这个里面直接搜 `lookUpImpOrForward` 函数名即可，可以详细看一下内部实现逻辑。

深入剖析Category

Category

有了之前 Runtime 的基础，一些内部实现就很好理解了。在 OC 中可以通过 Category 添加属性、方法、协议，在 Runtime 中 Class 和 Category 都是通过结构体实现的。

和 Category 语法很相似的还有 Extension，二者的区别在于，Extension 在编译期就直接和原类编译在一起，而 Category 是在运行时动态添加到原类中的。

基于之前的源码分析，我们来分析一下 Category 的实现原理。

在 _read_images 函数中会执行一个循环嵌套，外部循环遍历所有类，并取出当前类对应 Category 数组。内部循环会遍历取出的 Category 数组，将每个 category_t 对象取出，最终执行 addUnattachedCategoryForClass 函数添加到 Category 哈希表中。

```
// 将category_t添加到List中，并通过NXMapInsert函数，更新所属类的Category列表
static void addUnattachedCategoryForClass(category_t *cat, Class cls,
                                             header_info *catHeader)
{
    // 获取到未添加的Category哈希表
    NXMapTable *cats = unattachedCategories();
    category_list *list;

    // 获取到buckets中的value，并向value对应的数组中添加category_t
    list = (category_list *)NXMapGet(cats, cls);
    if (!list) {
        list = (category_list *)
            calloc(sizeof(*list) + sizeof(list->list[0]), 1);
    } else {
        list = (category_list *)
            realloc(list, sizeof(*list) + sizeof(list->list[0]) * (list->count + 1));
    }
    // 替换之前的List字段
    list->list[list->count++] = (locstamped_category_t){cat, catHeader};
    NXMapInsert(cats, cls, list);
}
```

Category 维护了一个名为 category_map 的哈希表，哈希表存储所有 category_t 对象。

```

// 获取未添加到Class中的category哈希表
static NXMapTable *unattachedCategories(void)
{
    // 未添加到Class中的category哈希表
    static NXMapTable *category_map = nil;

    if (category_map) return category_map;

    // fixme initial map size
    category_map = NXCreateMapTable(NXPtrValueMapPrototype, 16);

    return category_map;
}

```

上面只是完成了向 Category 哈希表中添加的操作，这时候哈希表中存储了所有 category_t 对象。然后需要调用 remethodizeClass 函数，向对应的 class 中添加 Category 的信息。

在 remethodizeClass 函数中会查找传入的 Class 参数对应的 Category 数组，然后将数组传给 attachCategories 函数，执行具体的添加操作。

```

// 将Category的信息添加到Class，包含method、property、protocol
static void remethodizeClass(Class cls)
{
    category_list *cats;
    bool isMeta;
    isMeta = cls->isMetaClass();

    // 从Category哈希表中查找category_t对象，并将已找到的对象从哈希表中删除
    if ((cats = unattachedCategoriesForClass(cls, false/*not realizing*/))) {
        attachCategories(cls, cats, true /*flush caches*/);
        free(cats);
    }
}

```

在 attachCategories 函数中，查找到 Category 的方法列表、属性列表、协议列表，然后通过对的 attachLists 函数，添加到 Class 对应的 class_rw_t 结构体中。

```

// 获取到Category的Protocol List、Property List、Method List，然后通过attachLists函数
// 添加到所属的类中
static void attachCategories(Class cls, category_list *cats, bool flush_caches)
{
    if (!cats) return;
    if (PrintReplacedMethods) printReplacements(cls, cats);
}

```

```
bool isMeta = cls->isMetaClass();

// 按照Category个数, 分配对应的内存空间
method_list_t **mlists = (method_list_t **)
    malloc(cats->count * sizeof(*mlists));
property_list_t **proplists = (property_list_t **)
    malloc(cats->count * sizeof(*proplists));
protocol_list_t **protolists = (protocol_list_t **)
    malloc(cats->count * sizeof(*protolists));

int mcount = 0;
int propcount = 0;
int protocount = 0;
int i = cats->count;
bool fromBundle = NO;

// 循环查找出Protocol list、Property list、Method list
while (i--) {
    auto& entry = cats->list[i];

    method_list_t *mlist = entry.cat->methodsForMeta(isMeta);
    if (mlist) {
        mlists[mcount++] = mlist;
        fromBundle |= entry.hi->isBundle();
    }

    property_list_t *proplist =
        entry.cat->propertiesForMeta(isMeta, entry.hi);
    if (proplist) {
        proplists[propcount++] = proplist;
    }

    protocol_list_t *protolist = entry.cat->protocols;
    if (protolist) {
        protolists[protocount++] = protolist;
    }
}

auto rw = cls->data();

// 执行添加操作
prepareMethodLists(cls, mlists, mcount, NO, fromBundle);
rw->methods.attachLists(mlists, mcount);
free(mlists);
if (flush_caches && mcount > 0) flushCaches(cls);

rw->properties.attachLists(proplists, propcount);
free(proplists);
```

```
rw->protocols.attachLists(protolists, protocount);
free(protolists);
}
```

这个过程就是将 `Category` 中的信息，添加到对应的 `class` 中，一个类的 `Category` 可能不只有一个，在这个过程中会将所有 `Category` 的信息都合并到 `class` 中。

方法覆盖

在有多个 `Category` 和原类的方法重复定义的时候，原类和所有 `Category` 的方法都会存在，并不会被后面的覆盖。假设有一个方法叫做 `method`，`Category` 和原类的方法都会被添加到方法列表中，只是存在的顺序不同。

▼ Compile Sources (5 items)

Name	Compiler Flags
 main.m ...in debug-objc	
 TestObject.m ...in debug-objc	
 TestObject+Category1.m ...in debug-objc	
 TestObject+Category2.m ...in debug-objc	
 TestObject+Category3.m ...in debug-objc	
+	
-	

在进行方法调用的时候，会优先遍历 `Category` 的方法，并且后面被添加到项目里的 `Category`，会被优先调用。上面的例子调用顺序就是 `Category3 -> Category2 -> Category1 -> TestObject`。如果从方法列表中找到方法后，就不会继续向后查找，这就是类方法被 `Category` “覆盖”的原因。

问题

在有多个 `Category` 和原类方法重名的情况下，怎样在一个 `Category` 的方法被调用后，调用所有 `Category` 和原类的方法？

可以在一个 `Category` 方法被调用后，遍历方法列表并调用其他同名方法。但是需要注意一点是，遍历过程中不能再调用自己的方法，否则会导致递归调用。为了避免这个问题，可以在调用前判断被调用的方法 `IMP` 是否当前方法的 `IMP`。

那怎样在任何一个 `Category` 的方法被调用后，只调用原类方法呢？

根据上面对方法调用的分析，`Runtime` 在调用方法时会优先所有 `Category` 调用，所以可以倒叙遍历方法列表，只遍历第一个方法即可，这个方法就是原类的方法。

Category Associate

在项目中经常会用到 Category，有时候会遇到给 Category 添加属性的需求，这时候就需要用到 associated 的 Runtime API 了。例如下面的例子中，需要在属性的 set、get 方法中动态添加实现。

```
// 声明文件
@interface TestObject (Category)
@property (nonatomic, strong) NSObject *object;
@end

// 实现文件
#import <objc/runtime.h>
#import <objc/message.h>
static void *const kAssociatedObjectKey = (void *)&kAssociatedObjectKey;

@implementation TestObject (Category)

- (NSObject *)object {
    return objc_getAssociatedObject(self, kAssociatedObjectKey);
}

- (void)setObject:(NSObject *)object {
    objc_setAssociatedObject(self, kAssociatedObjectKey, object, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

@end
```

在 Category 中添加属性后，默认是没有实现方法的，如果调用属性则会崩溃，而且还会提示下面两个警告信息。

```
Property 'object' requires method 'object' to be defined - use @dynamic or provide
a method implementation in this category

Property 'object' requires method 'setObject:' to be defined - use @dynamic or provide
a method implementation in this category
```

下面让我们看一下 associated 的源码，看 Runtime 是怎么通过 Runtime 动态添加 set、get 的。下面是 objc_getAssociatedObject 函数的实现代码，objc_setAssociatedObject 实现也是类似，这里节省地方就不贴出来了。

```
id _object_get_associative_reference(id object, void *key) {
```

```
id value = nil;
uintptr_t policy =_OBJC_ASSOCIATION_ASSIGN;
{
    AssociationsManager manager;
    AssociationsHashMap &associations(manager.associations());
    disguised_ptr_t disguised_object = DISGUISE(object);
    AssociationsHashMap::iterator i = associations.find(disguised_object);
    if (i != associations.end()) {
        ObjectAssociationMap *refs = i->second;
        ObjectAssociationMap::iterator j = refs->find(key);
        if (j != refs->end()) {
            ObjcAssociation &entry = j->second;
            value = entry.value();
            policy = entry.policy();
            if (policy &_OBJC_ASSOCIATION_GETTER_RETAIN) {
                objc_retain(value);
            }
        }
    }
}
if (value && (policy &_OBJC_ASSOCIATION_GETTER_AUTORELEASE)) {
    objc_autorelease(value);
}
return value;
}
```

从源码可以看出，所有通过 `associated` 添加的属性，都被存在一个单独的哈希表 `AssociationsHashMap` 中。`objc_setAssociatedObject` 和 `objc_getAssociatedObject` 函数本质上都是在操作这个哈希表，通过对哈希表进行映射来存取对象。

在 `associated` 的 API 中会设置一些内存管理的关键字，例如 `OBJC_ASSOCIATION_ASSIGN`，这是用来指定对象的内存管理的，这些关键字在 `Runtime` 源码中也有对应的处理。

Message Forward

当一个对象的方法被调用时，首先在对象所属的类中查找方法列表，如果当前类中没有则向父类查找，一种找到根类 `NSObject`。如果始终没有找到方法实现，则进入消息转发步骤中。

动态消息解析

当一个方法没有实现时，也就是在 `cache list` 和其继承关系的 `method list` 中，没有找到对应的方法。这时会进入消息转发阶段，但是在进入消息转发阶段前，`Runtime` 会给一次机会动态添加方法实现。

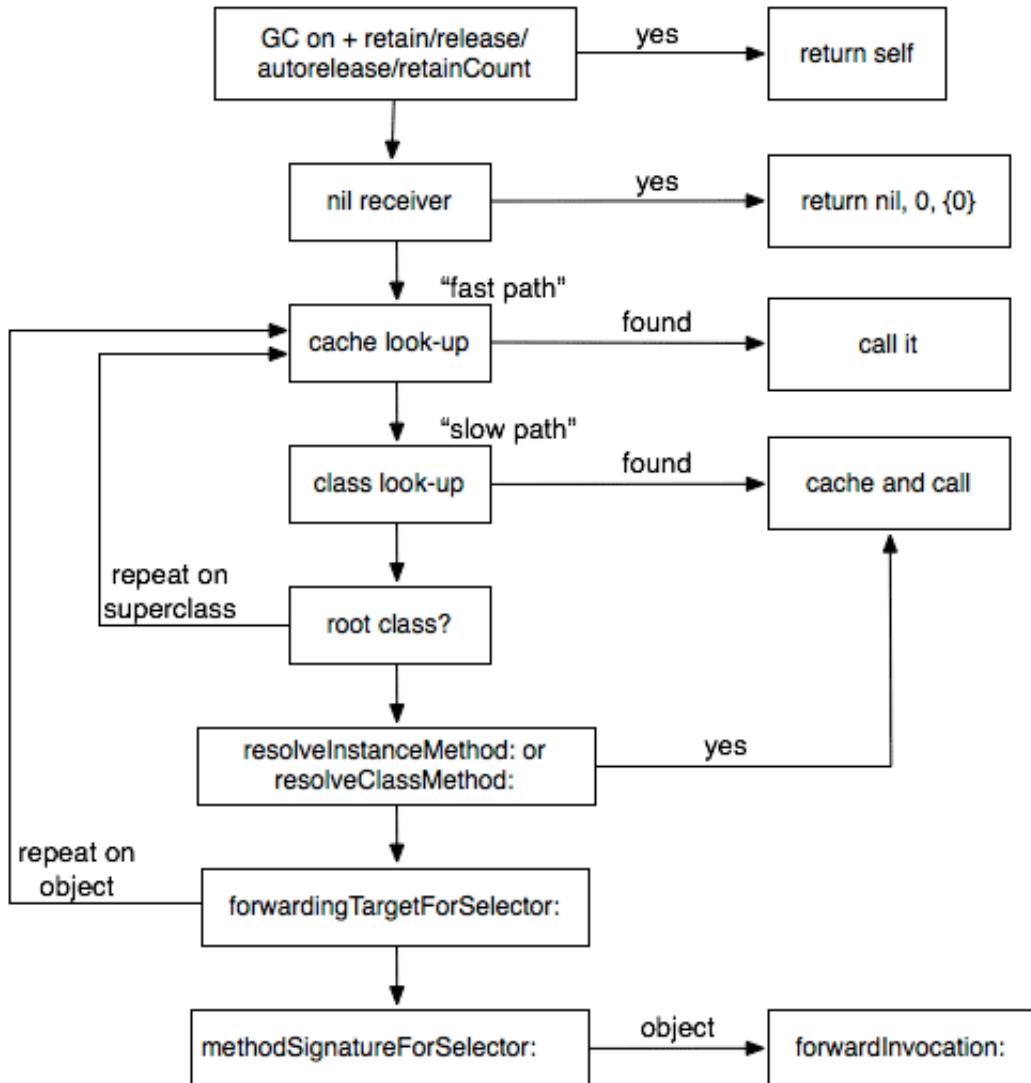
可以通过重写 `resolveInstanceMethod:` 和 `resolveClassMethod:` 方法，动态添加未实现的方法。其中第一个是添加实例方法，第二个是添加类方法。这两个方法都有一个 `BOOL` 返回值，返回 `NO` 则进入消息转发机制。

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

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

在通过 `class_addMethod` 函数动态添加实现时，后面有一个 `"v@:"` 来描述 `SEL` 对应的函数实现，具体的描述可以参考[官方文档](#)。

Message Forwarding



在进行消息转发之前，还可以在 `forwardingTargetForSelector:` 方法中将未实现的消息，转发给其他对象。可以在下面方法中，返回响应未实现方法的其他对象。

```

- (id)forwardingTargetForSelector:(SEL)aSelector {
    NSString *selectorName = NSStringFromSelector(aSelector);
    if ([selectorName isEqualToString:@"selector"]) {
        return object;
    }
    return [super forwardingTargetForSelector:aSelector];
}
  
```

当 `forwardingTargetForSelector:` 方法未做出任何响应的话，会来到消息转发流程。消息转发时会首先调用 `methodSignatureForSelector:` 方法，在方法内部生成 `NSMethodSignature` 类型的方法签名对象。在生成签名对象时，可以指定 `target` 和 `SEL`，可以将这两个参数换成其他参数，将消息转发给其他对象。

```
[otherObject methodSignatureForSelector:otherSelector];
```

生成 `NSMethodSignature` 签名对象后，就会调用 `forwardInvocation:` 方法，这是消息转发中最后一步了，如果在这步还没有对消息进行处理，则会导致崩溃。

这个方法中会传入一个 `NSInvocation` 对象，这个对象就是通过刚才生成的签名对象创建的，可以通过 `invocation` 调用其他对象的方法，调用其 `invokeWithTarget:` 即可。

```
- (void)forwardInvocation:(NSInvocation *)anInvocation {
    if ([object respondsToSelector:[anInvocation selector]]) {
        [anInvocation invokeWithTarget:object];
    } else {
        [super forwardInvocation:anInvocation];
    }
}
```

消息转发

将一条消息发送给一个不能处理的对象会引起崩溃，但是在崩溃之前，系统给响应对象一次处理异常的机会。

当发送一条对象不能处理的消息，产生 `Crash` 之前，系统会调用响应者的 `forwardInvocation:` 方法，并传入一个 `NSInvocation` 对象，`NSInvocation` 对象中包含原始消息及参数。这个方法只有方法未实现的时候才会调用。

你可以实现 `forwardInvocation:` 方法，将消息转发给另一个对象。`forwardInvocation:` 方法是一个动态方法，在响应者无法响应方法时，会调用 `forwardInvocation:` 方法，可以重写这个方法实现消息转发。

消息转发中 `forwardInvocation:` 需要做的是，确认消息将发送到哪里，以及用原始参数发送消息。可以通过 `invokeWithTarget:` 方法，发送被转发的消息。调用 `invoke` 方法后，原方法的返回值将被返回给调用方。

```
- (void)forwardInvocation:(NSInvocation *)anInvocation {
    if ([someOtherObject respondsToSelector:[anInvocation selector]]) {
        [anInvocation invokeWithTarget:someOtherObject];
    } else {
        [super forwardInvocation:anInvocation];
    }
}
```

forwardInvocation: 方法不仅可以处理一个方法，可以通过 selector 进行判断，来处理多个需要转发的方法。

动态方法解析

在OC中有时候可以动态的提供方法实现，例如属性可以通过 @dynamic propertyName; 的形式，表示将在运行过程中动态的提供属性方法。

如果想实现动态方法解析，需要实现 resolveInstanceMethod: 或 resolveClassMethod: 方法，在这两个方法中动态的添加方法实现。通过 class_addMethod 方法可以动态添加方法，添加方法时需要关联对应的函数指针，函数指针需要声明两个隐藏参数 self 和 _cmd 。

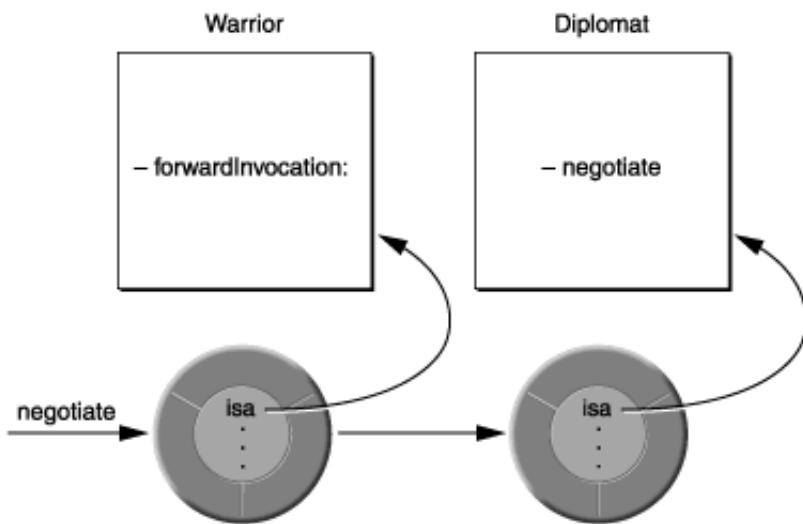
```
void dynamicMethodIMP(id self, SEL _cmd) {
    // implementation ....
}

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

消息转发和动态方法解析大部分是相同的，在消息转发之前一个类有机会动态解析这个方法。如果已经调用 respondsToSelector: 或 instancesRespondToSelector: 方法，动态方法解析有机会优先为 Selector 添加 IMP 。如果你实现了 resolveInstanceMethod: 方法，但想要特定的 Selector 走消息转发流程，则将此方法返回 NO 即可。

转发和多继承

可以通过消息转发机制来模拟多继承，例如下面这张图中，两个类中虽然不存在继承关系，但是却由另一个类处理了 Warrior 的事件。



由上面的例子可以看出，分属两个继承分支的类，通过消息转发机制实现了继承的关系。`Warrior` 的 `negotiate` 消息由其“父类” `Diplomat` 来实现。

通过消息转发实现的多重继承相对于普通继承来说更有优势，消息转发可以将消息转发给多个对象，这样就可以将代码按不同职责封装为不同对象，并通过消息转发给不同对象处理。

需要注意的是，`Warrior` 虽然通过消息转发机制可以响应 `negotiate` 消息，但如果通过 `respondsToSelector:` 和 `isKindOfClass:` 方法进行判断的话，依然是返回 `NO` 的。如果想让这两个方法可以在判断 `negotiate` 方法时返回 `YES`，需要重写这两个方法并在其中加入判断逻辑。

```

- (BOOL)respondsToSelector:(SEL)aSelector {
    if ([super respondsToSelector:aSelector]) {
        return YES;
    } else {
        //
    }
    return NO;
}
  
```

在执行 `forwardInvocation:` 之前，需要通过 `methodSignatureForSelector:` 方法返回方法签名，如果不实现则用默认的方法签名。

在方法签名的过程中，可以将未实现的方法转发给其代理。

```

- (NSMethodSignature*)methodSignatureForSelector:(SEL)selector {
    NSMethodSignature* signature = [super methodSignatureForSelector:selector];
    if (!signature) {
        signature = [surrogate methodSignatureForSelector:selector];
    }
    return signature;
}
  
```

使用技巧

在项目中经常会出现因为调用未实现的方法，导致程序崩溃的情况。在学习消息转发后，就可以通过消息转发来解决这个问题。

所有的类的基类都是 `NSObject` 类(`NSProxy` 除外)，可以将 `NSObject` 类的消息转发流程拦截，然后做一些统一的处理，这样就可以解决方法未实现导致的崩溃。根据 `Category` 可以将原类方法“覆盖”的特点，可以在 `Category` 中实现相应的拦截方法。

```
// 自定义类
#import <Foundation/Foundation.h>
@interface TestObject : NSObject
- (void)testMethod;
@end

// 接收消息的IMP
void dynamicResolveMethod(id self, SEL _cmd) {
    NSLog(@"method forward");
}

// 对NSObject创建的Category
@implementation NSObject (ExceptionForward)

#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wobjc-protocol-method-implementation"

+ (BOOL)resolveInstanceMethod:(SEL)sel {
    const char *types = sel_getName(sel);
    class_addMethod([self class], sel, (IMP)dynamicResolveMethod, types);
    return YES;
}

#pragma clang diagnostic pop

@end
```

我们的拦截方案是在 `resolveInstanceMethod` 方法中，动态创建未实现的方法，并将 `IMP` 统一设置为 `dynamicResolveMethod` 函数进行处理。这样所有未实现的方法都会执行 `dynamicResolveMethod` 函数，而不崩溃，在 `dynamicResolveMethod` 函数中可以做崩溃统计等操作。

多继承

在OC中是不支持多继承的，但是可以通过消息转发模拟多继承。在子类中实例化多个父类，当消息发送过来的时候，在消息转发的方法中，将调用重定向到父类的实例对象中，以实现多继承的效果。

下面是多继承的例子，创建两个父类 `Cat` 和 `Dog`，并将需要子类继承的方法都定义到 `Protocol` 中，在 `Cat` 和 `Dog` 中实现 `Protocol` 中的方法。

```
@protocol CatProtocol <NSObject>
- (void)eatFish;
@end

@interface Cat : NSObject <CatProtocol>
@end

@implementation Cat
- (void)eatFish {
    NSLog(@"Cat Eat Fish");
}
@end

@protocol DogProtocol <NSObject>
- (void)eatBone;
@end

@interface Dog : NSObject
@end

@implementation Dog
- (void)eatBone {
    NSLog(@"Dog Eat Bone");
}
@end
```

子类直接通过遵守父类的协议，来表示自己“继承”哪些类，并在内部实例化对应的父类对象。在外界调用协议方法时，子类其实是没有实现这些父类的方法的，所以通过转发方法将消息转发给响应的父类。

```
@interface TestObject : NSObject <CatProtocol, DogProtocol>
@end

@interface TestObject()
@property (nonatomic, strong) Cat *cat;
```

```
@property (nonatomic, strong) Dog *dog;
@end

@implementation TestObject

- (id)forwardingTargetForSelector:(SEL)aSelector {
    if ([self.cat respondsToSelector:aSelector]) {
        return self.cat;
    } else if ([self.dog respondsToSelector:aSelector]) {
        return self.dog;
    } else {
        return self;
    }
}

// 忽略Cat和Dog的初始化过程
@end
```

Method Swizzling

需求

就拿我们公司项目来说吧，我们公司是做导航的，而且项目规模比较大，各个控制器功能都已经实现。突然有一天老大过来，说我们要在所有页面添加统计功能，也就是用户进入这个页面就统计一次。我们会想到下面的一些方法：

手动添加

直接简单粗暴的在每个控制器中加入统计，复制、粘贴、复制、粘贴...

上面这种方法太Low了，消耗时间而且以后非常难以维护，会让后面的开发人员骂死的。

继承

我们可以使用 OOP 的特性之一，继承的方式来解决这个问题。创建一个基类，在这个基类中添加统计方法，其他类都继承自这个基类。

然而，这种方式修改还是很大，而且定制性很差。以后有新人加入之后，都要嘱咐其继承自这个基类，所以这种方式并不可取。

Category

我们可以为 `UIViewController` 建一个 `Category`，然后在所有控制器中引入这个 `Category`。当然我们也可以添加一个 `PCH` 文件，然后将这个 `Category` 添加到 `PCH` 文件中。

我们创建一个 `Category` 来覆盖系统方法，系统会优先调用 `Category` 中的代码，然后在调用原类中的代码。

我们可以通过下面的这段伪代码来看一下：

```
#import "UIViewController+EventGather.h"

@implementation UIViewController (EventGather)

- (void)viewDidLoad {
    NSLog(@"页面统计:%@", self);
}

@end
```

Method Swizzling

我们可以使用苹果的“黑魔法”Method Swizzling，Method Swizzling 本质上就是对 IMP 和 SEL 进行交换。

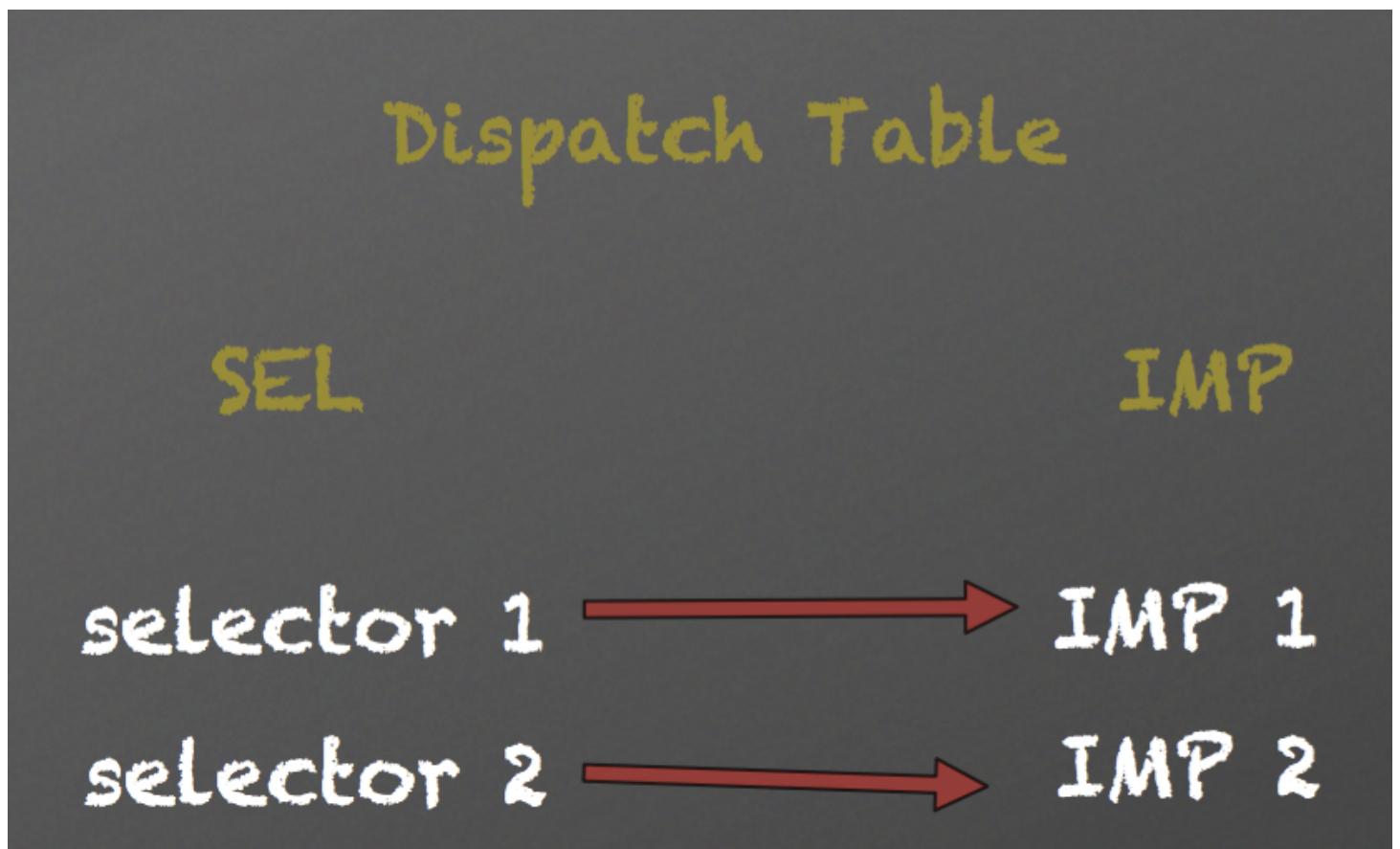
Method Swizzling原理

Method Swizzling 是发生在运行时的，主要用于在运行时将两个 Method 进行交换，我们可以将 Method Swizzling 代码写到任何地方，但是只有在这段 Method Swizzling 代码执行完毕之后互换才起作用。

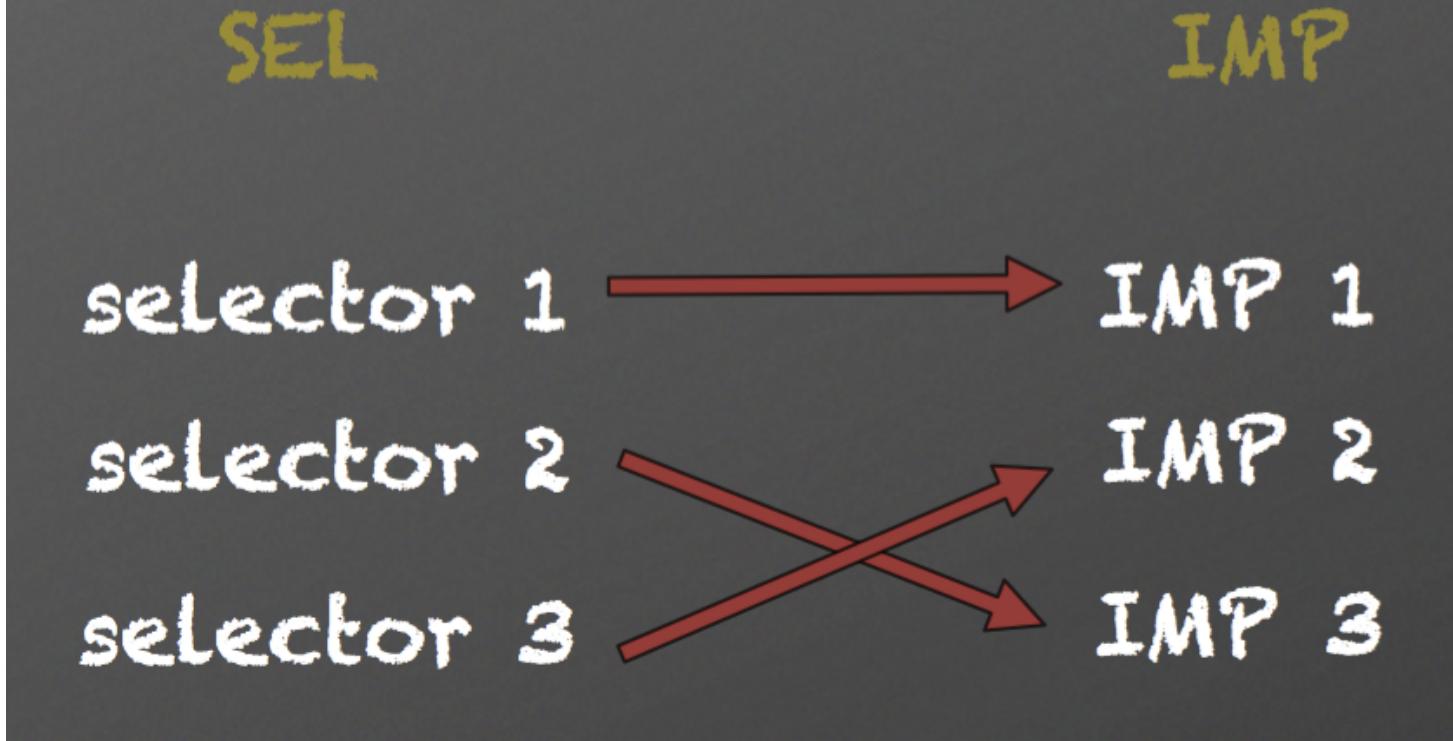
而且 Method Swizzling 也是 iOS 中 AOP (面向切面编程)的一种实现方式，我们可以利用苹果这一特性来实现 AOP 编程。

原理分析

首先，让我们通过两张图片来了解一下 Method Swizzling 的实现原理



Dispatch Table



上面图一中 selector2 原本对应着 IMP2，但是为了更方便的实现特定业务需求，我们在图二中添加了 selector3 和 IMP3，并且让 selector2 指向了 IMP3，而 selector3 则指向了 IMP2，这样就实现了“方法互换”。

在 oc 语言的 runtime 特性中，调用一个对象的方法就是给这个对象发送消息。是通过查找接收消息对象的方法列表，从方法列表中查找对应的 SEL，这个 SEL 对应着一个 IMP (一个 IMP 可以对应多个 SEL)，通过这个 IMP 找到对应的方法调用。

在每个类中都有一个 Dispatch Table，这个 Dispatch Table 本质是将类中的 SEL 和 IMP (可以理解为函数指针)进行对应。而我们的 Method Swizzling 就是对这个 table 进行了操作，让 SEL 对应另一个 IMP。

Method Swizzling使用

在实现 Method Swizzling 时，核心代码主要就是一个 runtime 的C语言API：

```
OBJC_EXPORT void method_exchangeImplementations(Method m1, Method m2) __OSX_AVAILABILITY_STARTING(__MAC_10_5, __IPHONE_2_0);
```

代码示例

就拿上面我们说的页面统计的需求来说吧，这个需求在很多公司都很常见，我们下面的Demo就通过 Method Swizzling 简单的实现这个需求。

我们先给 `UIViewController` 添加一个 `Category`，然后在 `Category` 中的 `+(void)load` 方法中添加 Method Swizzling 方法，我们用来替换的方法也写在这个 `Category` 中。由于 `load` 类方法是程序运行时这个类被加载到内存中就调用的一个方法，执行比较早，并且不需要我们手动调用。而且这个方法具有唯一性，也就是只会被调用一次，不用担心资源抢夺的问题。

定义 Method Swizzling 中我们自定义的方法时，需要注意尽量加前缀，以防止和其他地方命名冲突，Method Swizzling 的替换方法命名一定要是唯一的，至少在被替换的类中必须是唯一的。

```
#import "UIViewController+swizzling.h"
#import <objc/runtime.h>

@implementation UIViewController (swizzling)

+ (void)load {
    // 通过class_getInstanceMethod()函数从当前对象中的method List获取method结构体，如果是类方法就使用class_getClassMethod()函数获取。
    Method fromMethod = class_getInstanceMethod([self class], @selector(viewDidLoad));
    Method toMethod = class_getInstanceMethod([self class], @selector(swizzlingViewDidLoad));
    /**
     我们在这里使用class_addMethod()函数对Method Swizzling做了一层验证，如果self没有实现被交换的方法，会导致失败。
     而且self没有交换的方法实现，但是父类有这个方法，这样就会调用父类的方法，结果就不是我们想要的结果了。
     所以我们在这里通过class_addMethod()的验证，如果self实现了这个方法，class_addMethod()函数将会返回NO，我们就可以对其进行交换了。
    */
    if (!class_addMethod([self class], @selector(swizzlingViewDidLoad), method_getImplementation(toMethod), method_getTypeEncoding(toMethod))) {
        method_exchangeImplementations(fromMethod, toMethod);
    }
}

// 我们自己实现的方法，也就是和self的viewDidLoad方法进行交换的方法。
- (void)swizzlingViewDidLoad {
    NSString *str = [NSString stringWithFormat:@"%@", self.class];
    // 我们在这里加一个判断，将系统的UIViewController的对象剔除掉
```

```
if (![str containsString:@"UI"]){
    NSLog(@"统计打点 : %@", self.class);
}
[self swizzlingViewDidLoad];
}
@end
```

看到上面的代码，肯定有人会问：楼主，你太粗心了，你在 `swizzlingViewDidLoad` 方法中又调用了 `[self swizzlingViewDidLoad];`，这难道不会产生递归调用吗？

答：然而....并不会😊。

还记得我们上面的图一和图二吗？Method Swizzling 的实现原理可以理解为“方法互换”。假设我们将 A 和 B 两个方法进行互换，向 A 方法发送消息时执行的却是 B 方法，向 B 方法发送消息时执行的是 A 方法。

例如我们上面的代码，系统调用 `UIViewController` 的 `viewDidLoad` 方法时，实际上执行的是我们实现的 `swizzlingViewDidLoad` 方法。而我们在 `swizzlingViewDidLoad` 方法内部调用 `[self swizzlingViewDidLoad];` 时，执行的是 `UIViewController` 的 `viewDidLoad` 方法。

Method Swizzling类簇

之前我也说到，在我们项目开发过程中，经常因为 `NSArray` 数组越界或者 `NSDictionary` 的 `key` 或者 `value` 值为 `nil` 等问题导致的崩溃，对于这些问题苹果并不会报一个警告，而是直接崩溃，感觉苹果这样确实有点“太狠了”。

由此，我们可以根据上面所学，

对 `NSArray`、`NSMutableArray`、`NSDictionary`、`NSSMutableDictionary` 等类进行 Method Swizzling，实现方式还是按照上面的例子来做。但是....你发现 Method Swizzling 根本就不起作用，代码也没写错啊，到底是什么鬼？

这是因为 Method Swizzling 对 `NSArray` 这些的类簇是不起作用的。因为这些类簇类，其实是一种抽象工厂的设计模式。抽象工厂内部有很多其它继承自当前类的子类，抽象工厂类会根据不同情况，创建不同的抽象对象来进行使用。例如我们调用 `NSArray` 的 `objectAtIndex:` 方法，这个类会在方法内部判断，内部创建不同抽象类进行操作。

所以也就是我们对 `NSArray` 类进行操作其实只是对父类进行了操作，在 `NSArray` 内部会创建其他子类来执行操作，真正执行操作的并不是 `NSArray` 自身，所以我们应该对其“真身”进行操作。

代码示例

下面我们实现了防止 `NSArray` 因为调用 `objectAtIndex:` 方法，取下标时数组越界导致的崩溃：

```

#import "NSArray+LXZArray.h"
#import "objc/runtime.h"

@implementation NSArray (LXZArray)

+ (void)load {
    Method fromMethod = class_getInstanceMethod(objc_getClass("__NSArrayI"), @selector(objectAtIndex:));
    Method toMethod = class_getInstanceMethod(objc_getClass("__NSArrayI"), @selector(lxz_objectAtIndex:));
    method_exchangeImplementations(fromMethod, toMethod);
}

- (id)lxz_objectAtIndex:(NSUInteger)index {
    if (self.count-1 < index) {
        // 这里做一下异常处理，不然都不知道出错了。
        @try {
            return [self lxz objectAtIndex:index];
        }
        @catch (NSEException *exception) {
            // 在崩溃后会打印崩溃信息，方便我们调试。
            NSLog(@"----- %s Crash Because Method %s -----\\n", class_getName(self.class), __func__);
            NSLog(@"%@", [exception callStackSymbols]);
            return nil;
        }
        @finally {}
    } else {
        return [self lxz objectAtIndex:index];
    }
}
@end

```

大家发现了吗，`__NSArrayI` 才是 `NSArray` 真正的类，而 `NSMutableArray` 又不一样。。我们可以通过 `runtime` 函数获取真正的类：

```
objc_getClass("__NSArrayI");
```

举例

下面我们列举一些常用的类簇的“真身”：

类	“真身”
<code>NSArray</code>	<code>__NSArrayI</code>

NSMutableArray	__NSArrayM
NSDictionary	__NSDictionaryI
NSMutableDictionary	__NSDictionaryM

其他自行Google....

JRSwizzle

在项目中我们肯定会在很多地方用到 Method Swizzling，而且在使用这个特性时有很多需要注意的地方。我们可以将 Method Swizzling 封装起来，也可以使用一些比较成熟的第三方。在这里我推荐 [Github](#) 上星最多的一个第三方 — [jrswizzle](#)

里面核心就两个类，代码看起来非常清爽。

```
#import <Foundation/Foundation.h>
@interface NSObject (JRSwizzle)
+ (BOOL)jr_swizzleMethod:(SEL)origSel_ withMethod:(SEL)altSel_ error:(NSError**)error_;
+ (BOOL)jr_swizzleClassMethod:(SEL)origSel_ withClassMethod:(SEL)altSel_ error:(NSError**)error_;
@end

// MethodSwizzle类
#import <objc/objc.h>
BOOL ClassMethodSwizzle(Class klass, SEL origSel, SEL altSel);
BOOL MethodSwizzle(Class klass, SEL origSel, SEL altSel);
```

Method Swizzling 错误剖析

在上面的例子中，如果只是单独对 NSArray 或 NSMutableArray 中的单个类进行 Method Swizzling，是可以正常使用并且不会发生异常的。如果进行 Method Swizzling 的类中，有两个类有继承关系的，并且 Swizzling 了同一个方法。例如同时对 NSArray 和 NSMutableArray 中的 objectAtIndex: 方法都进行了 Swizzling，这样可能会导致父类 Swizzling 失效的问题。

对于这种问题主要是两个原因导致的，首先是不要在 + (void)load 方法中调用 [super load] 方法，这会导致父类的 Swizzling 被重复执行两次，这样父类的 Swizzling 就会失效。例如下面的两张图片，你会发现由于 NSMutableArray 调用了 [super load] 导致父类 NSArray 的 Swizzling 代码被执行了两次。

错误代码：

```

#import "NSMutableArray+LXZArrayM.h"

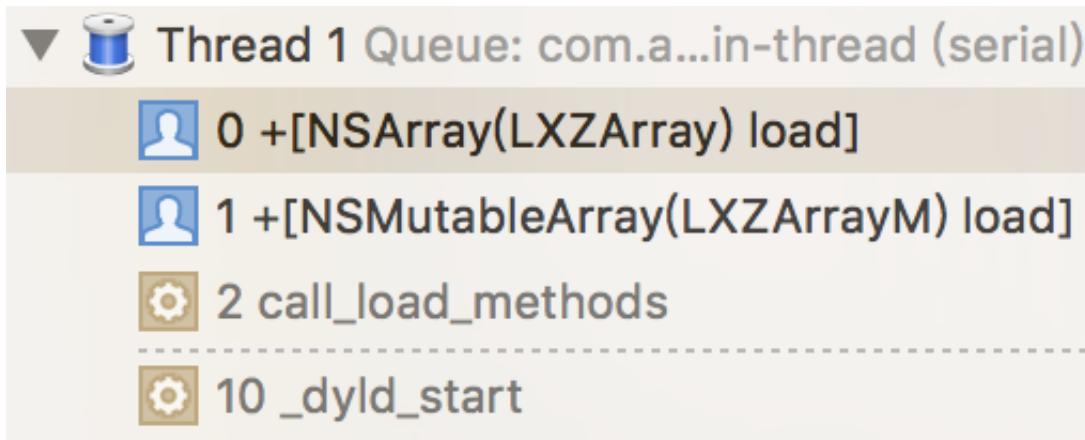
@implementation NSMutableArray (LXZArrayM)

+ (void)load {
    // 这里不应该调用super, 会导致父类被重复Swizzling
    [super load];

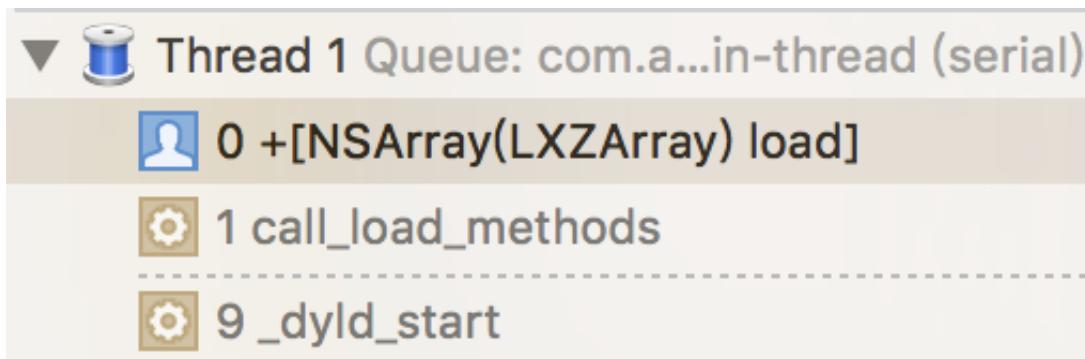
    Method fromMethod = class_getInstanceMethod(objc_getClass("__NSArrayM"), @selector(objectAtIndex:));
    Method toMethod = class_getInstanceMethod(objc_getClass("__NSArrayM"), @selector(lxz_objectAtIndexM:));
    method_exchangeImplementations(fromMethod, toMethod);
}

```

这里由于在子类中调用了super, 导致NSMutableArray执行时, 父类NSArray也被执行了一次。



父类NSArray执行了第二次Swizzling, 这时候就会出现问题, 后面会讲具体原因。



这样就会导致程序运行过程中, 子类调用 Swizzling 的方法是没有问题的, 父类调用同一个方法就会发现 Swizzling 失效了.....具体原因我们后面讲!

还有一个原因就是因为代码逻辑导致 Swizzling 代码被执行了多次, 这也会导致 Swizzling 失效, 其实原理和上面的问题是一样的, 我们下面讲讲为什么会出现这个问题。

问题原因

我们上面提到过 Method Swizzling 的实现原理就是对类的 Dispatch Table 进行操作，每进行一次 Swizzling 就交换一次 SEL 和 IMP (可以理解为函数指针)，如果 Swizzling 被执行了多次，就相当于 SEL 和 IMP 被交换了多次。这就会导致第一次执行成功交换了、第二次执行又换回去了、第三次执行.....这样换来换去的结果，能不能成功就看运气了😊，这也是好多人说 Method Swizzling 不好用的原因之一。

一图胜千言：

Dispatch Table

SEL

IMP

第一次成功交换

selector1

IMP1

selector2

IMP2

第二次又被交换为最初的状态

selector2

IMP2

selector1

IMP1

第三次和第一次交换的结果相同

selector1

IMP1

selector2

IMP2

从这张图中我们也可以看出问题产生的原因了，就是 Swizzling 的代码被重复执行，为了避免这样的原因出现，我们可以通过__GCD__的 dispatch_once 函数来解决，利用 dispatch_once 函数内代码只会执行一次的特性。

在每个 Method Swizzling 的地方，加上 dispatch_once 函数保证代码只被执行一次。当然在实际使用中也可以对下面代码进行封装，这里只是给一个示例代码。

```
#import "NSMutableArray+LXZArrayM.h"

@implementation NSMutableArray (LXZArrayM)

+ (void)load {
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        Method fromMethod = class_getInstanceMethod(objc_getClass("__NSArrayM"), @selector(objectAtIndex:));
        Method toMethod = class_getInstanceMethod(objc_getClass("__NSArrayM"), @selector(lxz_objectAtIndexM:));
        method_exchangeImplementations(fromMethod, toMethod);
    });
}
```

这里还要告诉大家一个调试小技巧，已经知道的可以略过😊。我们之前说过 IMP 本质上就是函数指针，所以我们可以通过打印函数指针的方式，查看 SEL 和 IMP 的交换流程。

先来一段测试代码

```
Method fromMethod = class_getInstanceMethod(objc_getClass("__NSArrayI"), @selector(objectAtIndex:));
Method toMethod = class_getInstanceMethod(objc_getClass("__NSArrayI"), @selector(lxz_objectAtIndex:));

 NSLog(@"%@", method_getImplementation(fromMethod));
 NSLog(@"%@", method_getImplementation(toMethod));
 method_exchangeImplementations(fromMethod, toMethod);

 NSLog(@"%@", method_getImplementation(fromMethod));
 NSLog(@"%@", method_getImplementation(toMethod));
 method_exchangeImplementations(fromMethod, toMethod);

 NSLog(@"%@", method_getImplementation(fromMethod));
 NSLog(@"%@", method_getImplementation(toMethod));
 method_exchangeImplementations(fromMethod, toMethod);

 NSLog(@"%@", method_getImplementation(fromMethod));
```

```
NSLog(@"%@", method_getImplementation(toMethod));
```

看到这个打印结果，大家应该明白什么问题了吧：

```
2016-04-13 14:16:33.477 [16314:4979302] 0x1851b7020
2016-04-13 14:16:33.479 [16314:4979302] 0x1000fb3c8
2016-04-13 14:16:33.479 [16314:4979302] 0x1000fb3c8
2016-04-13 14:16:33.480 [16314:4979302] 0x1851b7020
2016-04-13 14:16:33.480 [16314:4979302] 0x1851b7020
2016-04-13 14:16:33.480 [16314:4979302] 0x1000fb3c8
2016-04-13 14:16:33.481 [16314:4979302] 0x1000fb3c8
2016-04-13 14:16:33.481 [16314:4979302] 0x1851b7020
```

Method Swizzling源码分析

下面是 Method Swizzling 的实现源码，从源码来看，其实内部实现很简单。核心代码就是交换两个 Method 的 imp 函数指针，这也就是方法被 swizzling 多次，可能会被换回去的原因，因为每次调用都会执行一次交换操作。

```
void method_exchangeImplementations(Method m1, Method m2)
{
    if (!m1 || !m2) return;

    rwlock_writer_t lock(runtimeLock);

    IMP m1_imp = m1->imp;
    m1->imp = m2->imp;
    m2->imp = m1_imp;

    flushCaches(nil);

    updateCustomRR_AWZ(nil, m1);
    updateCustomRR_AWZ(nil, m2);
}
```

Method Swizzling危险吗？

既然 Method Swizzling 可以对这个类的 Dispatch Table 进行操作，操作后的结果对所有当前类及子类都会产生影响，所以有人认为 Method Swizzling 是一种危险的技术，用不好很容易导致一些不可预见的`_bug_`，这些`_bug_`一般都是非常难发现和调试的。

这个问题可以引用[念茜](#)大神的一句话： 使用 Method Swizzling 编程就好比切菜时使用锋利的刀，一些人因为担心切到自己所以害怕锋利的刀具，可是事实上，使用钝刀往往更容易出事，而利刀更为安全。

在这个 Demo 中通过 Method Swizzling，简单实现了一个崩溃拦截功能。实现方式就是将原方法 Swizzling 为自己定义的方法，在执行时先在自己方法中做判断，根据是否异常再做下一步处理。

Demo 只是用来辅助读者更好的理解文章中的内容，[应该博客结合 Demo 一起学习，只看 Demo 还是不能理解更深层的原理。](#) Demo 中代码都会有注释，各位可以打断点跟着 Demo 执行流程走一遍，看看各个阶段变量的值。

Demo地址：[刘小壮的Github](#)

Runtime的应用

__attribute__

`__attribute__` 是一套编译器指令，被 `GNU` 和 `LLVM` 编译器所支持，允许对于 `__attribute__` 增加一些参数，做一些高级检查和优化。

`__attribute__` 的语法是，在后面加两个括号，然后写属性列表，属性列表以逗号分隔。在 iOS 中，很多例如 `NS_CLASS_AVAILABLE_IOS` 的宏定义，内部也是通过 `__attribute__` 实现的。

```
__attribute__((attribute1, attribute2));
```

下面是一些 `__attribute__` 的常用属性，更完整的属性列表可以到[llvm的官网查看](#)。

Supported Syntaxes						
GNU	C++11	C2x	<code>__declspec</code>	Keyword	Pragma	Pragma clang attribute
X	X	X				X
By default, the Objective-C interface or protocol identifier is used in the metadata name for that object. The <code>objc_runtime_name</code> attribute allows annotated interfaces or protocols to use the specified string argument in the object's metadata name instead of the default name.						
Usage: <code>__attribute__((objc_runtime_name("MyLocalName")))</code> . This attribute can only be placed before an @protocol or @interface declaration:						
<pre>__attribute__((objc_runtime_name("MyLocalName"))) @interface Message @end</pre>						

objc_subclassing_restricted

`objc_subclassing_restricted` 属性表示被修饰的类不能被其他类继承，否则会报下面的错误。

```
__attribute__((objc_subclassing_restricted))
@interface TestObject : NSObject
@property (nonatomic, strong) NSObject *object;
@property (nonatomic, assign) NSInteger age;
@end

@interface Child : TestObject
@end
```

错误信息：

Cannot subclass a class that was declared with the '`objc_subclassing_restricted`' attribute

objc_requires_super

`objc_requires_super` 属性表示子类必须调用被修饰的方法 `super`，否则报黄色警告。

```
@interface TestObject : NSObject
- (void)testMethod __attribute__((objc_requires_super));
@end

@interface Child : TestObject
@end
```

警告信息：(不报错)

Method possibly missing a [super testMethod] call

constructor / destructor

`constructor` 属性表示在 `main` 函数执行之前，可以执行一些操作。`destructor` 属性表示在 `main` 函数执行之后做一些操作。`constructor` 的执行时机是在所有 `load` 方法都执行完之后，才会执行所有 `constructor` 属性修饰的函数。

```
_attribute__((constructor)) static void beforeMain() {
    NSLog(@"before main");
}

_attribute__((destructor)) static void afterMain() {
    NSLog(@"after main");
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSLog(@"execute main");
    }
    return 0;
}
```

执行结果：

```
debug-objc[23391:1143291] before main
debug-objc[23391:1143291] execute main
debug-objc[23391:1143291] after main
```

在有多个 `constructor` 或 `destructor` 属性修饰的函数时，可以通过设置优先级来指定执行顺序。格式是 `_attribute__((constructor(101)))` 的方式，在属性后面直接跟优先级。

```
__attribute__((constructor(103))) static void beforeMain3() {
    NSLog(@"after main 3");
}

__attribute__((constructor(101))) static void beforeMain1() {
    NSLog(@"after main 1");
}

__attribute__((constructor(102))) static void beforeMain2() {
    NSLog(@"after main 2");
}
```

在 `constructor` 中根据优先级越低，执行顺序越高。而 `destructor` 则相反，优先级越高则执行顺序越高。

overloadable

`overloadable` 属性允许定义多个同名但不同参数类型的函数，在调用时编译器会根据传入参数类型自动匹配函数。这个有点类似于 C++ 的函数重载，而且都是发生在编译期的行为。

```
__attribute__((overloadable)) void testMethod(int age) {}
__attribute__((overloadable)) void testMethod(NSString *name) {}
__attribute__((overloadable)) void testMethod(BOOL gender) {}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        testMethod(18);
        testMethod(@"lxz");
        testMethod(YES);
    }
    return 0;
}
```

objc_runtime_name

`objc_runtime_name` 属性可以在编译时，将 `Class` 或 `Protocol` 指定为另一个名字，并且新名字不受命名规范制约，可以以数字开头。

```
__attribute__((objc_runtime_name("TestObject")))
@interface Object : NSObject
@end

 NSLog(@"%@", NSStringFromClass([TestObject class]));
```

执行结果：
TestObject

这个属性可以用来做代码混淆，例如写一个宏定义，宏定义内部实现混淆逻辑。例如通过 MD5 对 Object 做混淆，32位的混淆结果就是 497031794414a552435f90151ac3b54b，谁能看出来这是什么类。如果怕彩虹表匹配出来，再增加加盐逻辑。

cleanup

通过 cleanup 属性，可以指定给一个变量，当变量释放之前执行一个函数。指定的函数执行的时间，是在 dealloc 之前的。在指定的函数中，可以传入一个形参，参数就是 cleanup 修饰的变量，形参是一个地址。

```
static void releaseBefore(NSObject **object) {
    NSLog(@"%@", *object);
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        TestObject *object __attribute__((cleanup(releaseBefore))) = [[TestObject alloc] init];
    }
    return 0;
}
```

如果遇到同一个代码块中，同时出现多个 cleanup 属性时，在代码块作用域结束时，会以添加的顺序进行调用。

unused

还有一个属性很实用，在项目里经常会有未使用的变量，会报一个黄色警告。有时候可能会通过其他方式获取这个对象，所以不想出现这个警告，可以通过 unused 属性消除这个警告。

```
NSObject *object __attribute__((unused)) = [[NSObject alloc] init];
```

系统定义

在系统里也大量使用了 __attribute__ 关键字，只不过系统不会直接在外部使用 __attribute__，一般都是将其定义为宏定义，以宏定义的形式出现在外面。

```
// NSLog
FOUNDATION_EXPORT void NSLog(NSString *format, ...) NS_FORMAT_FUNCTION(1,2) NS_NO_T
AIL_CALL;
#define NS_FORMAT_FUNCTION(F,A) __attribute__((format(__NSString__, F, A)))

// 必须调用父类的方法
#define NS_REQUIRES_SUPER __attribute__((objc_requires_super))

// 指定初始化方法, 必须直接或间接调用修饰的方法
#define NS_DESIGNATED_INITIALIZER __attribute__((objc_designated_initializer))
```

ORM

对象关系映射 (Object Relational Mapping) , 简称 ORM , 用于面向对象语言中不同系统数据之间的转换。

可以通过对象关系映射来实现 JSON 转模型, 使用比较多的是 Mantle 、 MJExtension 、 YYKit 、 JSONModel 等框架, 这些框架在进行转换的时候, 都是使用 Runtime 的方式实现的。

Mantle 使用和 MJExtension 有些类似, 只不过 MJExtension 使用起来更加方便。 Mantle 在使用时主要是通过继承的方式处理, 而 MJExtension 是通过 Category 处理, 代码依赖性更小, 无侵入性。

性能评测

这些第三方中 Mantle 功能最强大, 但是太臃肿, 使用起来性能比其他第三方都差一些。 JSONModel 、 MJExtension 这些第三方几乎都在一个水平级, YYKit 相对来说性能可以比肩手写赋值代码, 性价比最高。

对于模型转换需求不是太大的工程来说, 尽量用 YYKit 来进行转换性能会更好一些。功能可能略逊于 MJExtension , 我个人还是比较习惯用 MJExtension 。

YYKit作者评测

实现思路

也可以自己实现模型转换的逻辑, 以字典转模型为例, 大体逻辑如下:

1. 创建一个 Category 用来做模型转换, 对外提供方法并传入字典对象。
2. 通过 Runtime 对应的函数, 获取属性列表并遍历, 根据属性名从字典中取出对应的对象。
3. 通过 KVC 将从字典中取出的值, 赋值给对象。
4. 有时候会遇到多层嵌套的情况, 例如字典包含数组, 数组中还是一个字典。这种情况就可以

做判断，如果模型对象是数组则取出字典对应字段的数组，然后遍历数组再调用字典赋值的方法。

下面简单实现了一个字典转模型的代码，通过 Runtime 遍历属性列表，并根据属性名取出字典中的对象，然后通过 KVC 进行赋值操作。调用方式和 MJExtension 、 YYModel 类似，直接通过模型类调用类方法即可。如果想在其他类中也使用的话，应该把下面的实现写在 NSObject 的 Category 中，这样所有类都可以调用。

```
// 调用部分
NSDictionary *dict = @{@"name" : @"lxz",
                      @"age" : @18,
                      @"gender" : @YES};
TestObject *object = [TestObject objectWithDict:dict];

// 实现代码
@interface TestObject : NSObject
@property (nonatomic, copy) NSString *name;
@property (nonatomic, assign) NSInteger age;
@property (nonatomic, assign) BOOL gender;

+ (instancetype)objectWithDict:(NSDictionary *)dict;
@end

@implementation TestObject

+ (instancetype)objectWithDict:(NSDictionary *)dict {
    return [[TestObject alloc] initWithDict:dict];
}

- (instancetype)initWithDict:(NSDictionary *)dict {
    self = [super init];
    if (self) {
        unsigned int count = 0;
        objc_property_t *propertys = class_copyPropertyList([self class], &count);
        for (int i = 0; i < count; i++) {
            objc_property_t property = propertys[i];
            const char *name = property_getName(property);
            NSString *nameStr = [[NSString alloc] initWithUTF8String:name];
            id value = [dict objectForKey:nameStr];
            [self setValue:value forKey:nameStr];
        }
        free(propertys);
    }
    return self;
}
```

```
@end
```

通过 Runtime 可以获取到对象的 Method List 、 Property List 等，不只可以用来做字典模型转换，还可以做很多工作。例如还可以通过 Runtime 实现自动归档和反归档，下面是自动进行归档操作。

```
// 1. 获取所有的属性
unsigned int count = 0;
Ivar *ivars = class_copyIvarList([NJPPerson class], &count);
// 遍历所有的属性进行归档
for (int i = 0; i < count; i++) {
    // 取出对应的属性
    Ivar ivar = ivars[i];
    const char * name = ivar_getName(ivar);
    // 将对应的属性名称转换为OC字符串
    NSString *key = [[NSString alloc] initWithUTF8String:name];
    // 根据属性名称利用KVC获取数据
    id value = [self valueForKeyPath:key];
    [encoder encodeObject:value forKey:key];
}
free(ivars);
```

我写了一个简单的 Category ，可以自动实现 NSCoding 、 NSCopying 协议。这是开源地址：[Easy NSCoding](#)

Runtime面试题

题1

下面的代码输出什么？

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

答案：都输出 Son。

第一个 NSLog 输出 Son 肯定是不用说的。

第二个输出中，[super class] 会被转换为下面代码。

```
struct objc_super objcSuper = {
    self,
    class_getSuperclass([self class]),
};

id (*sendSuper)(struct objc_super*, SEL) = (void *)objc_msgSendSuper;
sendSuper(&objcSuper, @selector(class));
```

super 的调用会被转换为 objc_msgSendSuper 的调用，并传入一个 objc_super 类型的结构体。结构体有两个参数，第一个就是接受消息的对象，第二个是 [super class] 对应的父类。

```
struct objc_super {
    __unsafe_unretained _Nonnull id receiver;
    __unsafe_unretained _Nonnull Class super_class;
};
```

由此可知，虽然调用的是 [super class]，但是接受消息的对象还是 self。然后来到父类 Father 的 class 方法中，输出 self 对应的类 Son。

题2

下面代码的结果？

```
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]];
```

答案：

除了第一个是 YES，其他三个都是 NO。

在推测结果之前，首先要明白两个问题。isKindOfClass 和 isMemberOfClass 的区别是什么？

isKindOfClass:cls，调用该方法的对象所属的类，继承者链中包含传入的 cls 则返回 YES。

isMemberOfClass:cls，调用改方法的对象所属的类，必须是传入的 cls 则返回 YES。

我们从 Runtime 源码的角度来分析一下结果。

```
+ (BOOL)isMemberOfClass:(Class)cls {
    return object_getClass((id)self) == cls;
}

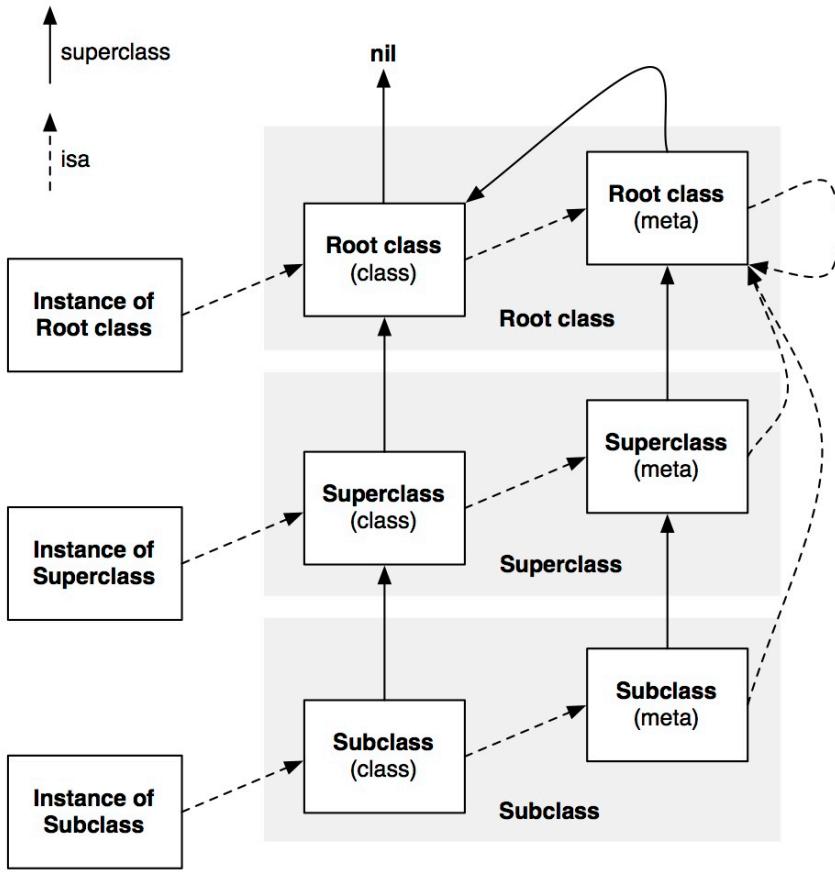
- (BOOL)isMemberOfClass:(Class)cls {
    return [self class] == cls;
}

+ (BOOL)isKindOfClass:(Class)cls {
    for (Class tcls = object_getClass((id)self); tcls; tcls = tcls->superclass) {
        if (tcls == cls) return YES;
    }
    return NO;
}

- (BOOL)isKindOfClass:(Class)cls {
    for (Class tcls = [self class]; tcls; tcls = tcls->superclass) {
        if (tcls == cls) return YES;
    }
    return NO;
}
```

平时开发过程中只会接触到对象方法的 `isKindOfClass` 和 `isMemberOfClass`，但是在 `NSObject` 类中还隐式的实现了类方法版本。不只这两个方法，其他 `NSObject` 中的对象方法，都有其对应的类方法版本。因为在OC中，类和元类也都是对象。这四个调用由于都是类对象发起调用的，所以最终执行的都是类方法版本。

先把 `Runtime` 的对象模型拿出来，方便后面的分析。



第一次调用方是 `NSObject` 类对象，调用 `isKindOfClass` 方法传入的也是类对象。因为调用类的 `class` 方法，会把类自身直接返回，所以还是类对象自己。

然后进入到 `for` 循环中，会从 `NSObject` 的元类开始遍历，所以第一次 `NSObject meta class != NSObject class`，匹配失败。第二次循环将 `tcls` 设置为 `superclass` 的 `NSObject` `class`，`NSObject class == NSObject class`，匹配成功。

`NSObject` 能匹配成功，是因为这个类比较特殊，在第二次获取 `superclass` 的时候，`NSObject` 元类的 `superclass` 就是 `NSObject` 的类对象，所以会匹配成功。而其他三种匹配，则都会失败，各位同学可以去自己分析一下剩下三种。

题3

下面的代码会？ `Compile Error / Runtime Crash / NSLog... ?`

```

@interface NSObject (Sark)
+ (void)foo;
@end

@implementation NSObject (Sark)
- (void)foo {
    NSLog(@"IMP: -[NSObject (Sark) foo]");
}
@end

```

```
// 测试代码
[NSObject foo];
[[NSObject new] performSelector:@selector(foo)];
```

答案：

全都正常输出，编译和运行都没有问题。

这道题和上一道题很相似，第二个调用肯定没有问题，第一个调用后会从元类中查找方法，然而方法并不在元类中，所以找元类的 `superclass`。方法定义在是 `NSObject` 的 `Category`，由于 `NSObject` 的对象模型比较特殊，元类的 `superclass` 是类对象，所以从类对象中找到了方法并调用。

题4

下面的代码会？ `Compile Error` / `Runtime Crash` / `NSLog...` ?

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

@implementation Sark
- (void)speak {
    NSLog(@"my name's %@", self.name);
}
@end

// 测试代码
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    id cls = [Sark class];
    void *obj = &cls;
    [(__bridge id)obj speak];
}
@end
```

答案：

正常执行，不会导致 `Crash`。

执行 `[Sark class]` 后获取到类对象，然后通过 `obj` 指针指向获取到的类对象首地址，这就构成了对象的基本结构，可以进行正常调用。

原题出处

题5

为什么 MRC 下没有 weak ?

其实 MRC 下并不是没有 weak , 在 MRC 环境下也可以通过 Runtime 源码调用 weak 源码的。 weak 源码定义在 Private Headers 私有文件夹下, 需要引入 #import "objc-internal.h" 文件。

以以下 ARC 的源码为例, 定义了一个 TestObject 类型的对象, 并用一个 weak 指针指向已创建对象。

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        TestObject *object = [[TestObject alloc] init];
        __weak TestObject *newObject = object;
    }
    return 0;
}
```

这段代码会被编译器转移为下面代码, 这段代码中的两个函数就是 weak 的实现函数, 在 MRC 下也可以调用这两个函数。

```
objc_initWeak(&newObject, object);
objc_destroyWeak(&newObject);
```

题6

相同的一个类, 创建不同的对象, 怎样实现指定的某个对象在 dealloc 时打印一段文字?

这个问题最简单的方法就是在类的 .h 文件里, 定义一个标记属性, 如果属性被赋值为 YES , 则在 dealloc 中打印文字。但是, 这种实现方式显然不是面试官想要的, 会被直接pass~

可以参考 KVO 的实现方案, 在运行时动态创建一个类, 这个类是对象的子类, 将新创建类的 dealloc 实现指向自定义的 IMP , 并在 IMP 中打印一段文字。将对象的 isa 设置为新创建的类, 当执行 dealloc 方法时就会执行 isa 所指向的新类。

思考

小问题

什么叫做技术大牛，怎样就表示技术强？

我前段时间看过一句话，我感觉可以解释上面的问题：“市面上所有应用的功能，产品提出来我都能做”。

这句话并不够全面，应该不只是做出来，而是更好的做出来。这个好要从很多方面去评估，性能、可维护性、完成时间、产品效果等，如果这些都做的很好，那足以证明这个人技术很强大。

Runtime有什么用？

Runtime 是比较偏底层的，但是研究这么深有什么用吗，有什么实际意义吗？

Runtime 当然是由实际用处的，先不说整个OC都是通过 Runtime 实现的。例如现在需要实现消息转发的功能，这时候就需要用到 Runtime，或者是拦截方法，也需要用到 Method Swizzling，除了这些，还有更多的用法待我们去发掘。

不只是使用，其实最重要的是，通过Runtime了解一个语言的设计。Runtime中不只是各种函数调用，从整体来看，可以明白OC的对象模型是什么样的。