RunTime总结

```objc
// Q1:
@interface Son : Father
@end

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

/** 主要考察的是super关键字的调用形式
分析： [super class] 调用的方法是 objc_msgSendSuper(objc_super struct, cmd),
相当于调用objc_msgSend(objc_super->receiver, cmd) + 从父类中开始查找方法的组合;
struct objc_super {
__unsafe_unretained id receiver;  // 指向的是自己
__unsafe_unretained Class super_class;  // 指向的是父类
}
- (Class)class {
return  object_getClass(self); isa 指针
}

即自己本身去调用父类的class方法
返回的isa指针，对象就会返回类对象，所以二者返回的都是Son
*/
```

```objc
// Q2:
@implementation ViewController
- (void)viewDidLoad {
    BOOL res1 = [[NSObject class] isKindOfClass:[NSObject class]];
    BOOL res2 = [[NSObject class] isMemberOfClass:[NSObject class]];
```

```
    BOOL res3 = [[Son class] isKindOfClass:[Son class]];
    BOOL res4 = [[Son class] isMemberOfClass:[Son class]];
}

/**  本题主要考察的是对于对象-类-元类 父类之间的关系链
+ (Class)class {
    return  self;
}

// isa指针（元类）与 cls是否相同
+ (BOOL)isMemberOfClass:(Class)cls {
    return object_getClass((id)self) == cls
}
// isa指针（类对象）与 cls是否相同
- (BOOL)isMemberOfClass:(Class)cls {
    return [self class] == cls;
}
// isa指针(元类)及其superClass链上与cls相同
+ (BOOL)isKindOfClass:(Class)cls {
    for (Class tcls = object_getClass((id)self); tcls; tcls = tcls->superclass) {
        if (tcls == cls) return YES;
    }
    return NO;
}
// isa指针（类对象）及其superClass链上与cls相同
- (BOOL)isKindOfClass:(Class)cls {
     for (Class tcls = [self class]; tcls; tcls = tcls->superclass) {
        if (tcls == cls) return YES;
    }
    return NO;
}

*/


```

```
// Q3:
@interface NSObject (Sark)
+ (void)foo;
- (void)foo;
@end

@implementation NSObject (Sark)
- (void)foo {
```

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

@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    [NSObject foo];
    [[NSObject new] foo];
}

/**



*/


```

```
// Q4:
@interface Father : NSObject
@property(nonatomic, strong) NSString *name;
- (void)speak ;
@end
@implementation Father
- (void)speak {
    NSLog(@"my name is %@", self.name);
}
@end

@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    id fatherCls = [Father class];
    void *fatherObj = &fatherCls;
    [(__bridge id)fatherObj speak];
}
```

```
- (void)viewDidLoad
{
    // 以下数字越高表示地址越大
    // 压栈参数1:  id self (4)
    // 压栈参数2:  SEL _cmd (3)
    [super viewDidLoad]; // objc_msgSendSuper2(struct objc_super, SEL)
    // struct objc_super2 {
    //   id receiver (等价于self)                    (1)
    //   Class super_class (等价于self.class)        (2)
    // }
    // objc_msgSendSuper2的SEL不需要申请栈空间

    id cls = [Sark class];
    void *obj = &cls; // (0)
    [(__bridge id)obj speak];
}
```

Runtime简介

C语言在编译期就决定了函数的调用

OC的函数，属于动态调用的过程，在编译期并不能决定真正调用哪个函数，只有在真正运行时才会根据函数的名称找到对应的函数来调用，这意味着它不仅需要一个编译器，也需要一个运行时系统来动态创建类和对象、进行消息传递和转发

|Objective-C Code|Framework & Service| Runtime API|

|Compiler|

|Runtime System Library|

通过Objective-C源代码

一般情况我们只需要编写OC代码即可，Runtime系统会在幕后把我们写的源代码在编译阶段转换成运行时代码，在运行时确定对应的数据结构和调用具体哪个方法。

通过Foundation框架的NSObject类定义的方法

NSObject协议，有以下5个方法，可以从Runtime中获取信息

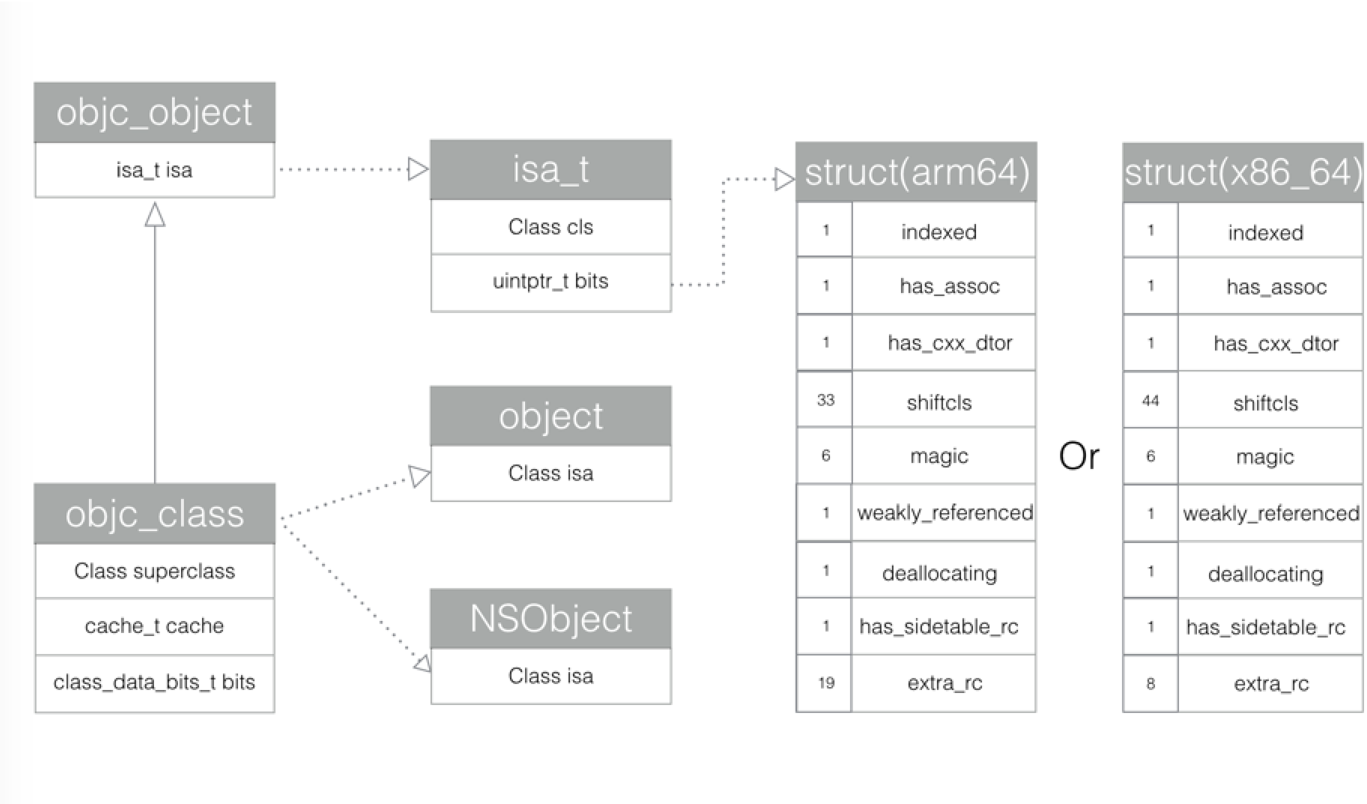
```
// 返回对象的类
- (Class)class OBJC_SWIFT_UNAVAILABLE("use 'anObject.dynamicType' instead");
// 返回对象的类
- (BOOL)isKindOfClass:(Class)aClass;
// 判断是不是该类
- (BOOL)isMemberOfClass:(Class)aClass;
// 判断是否遵守协议
- (BOOL)conformsToProtocol:(Protocol *)aProtocol;
// 判断能否响应指定的消息
- (BOOL)respondsToSelector:(SEL)aSelector;
```

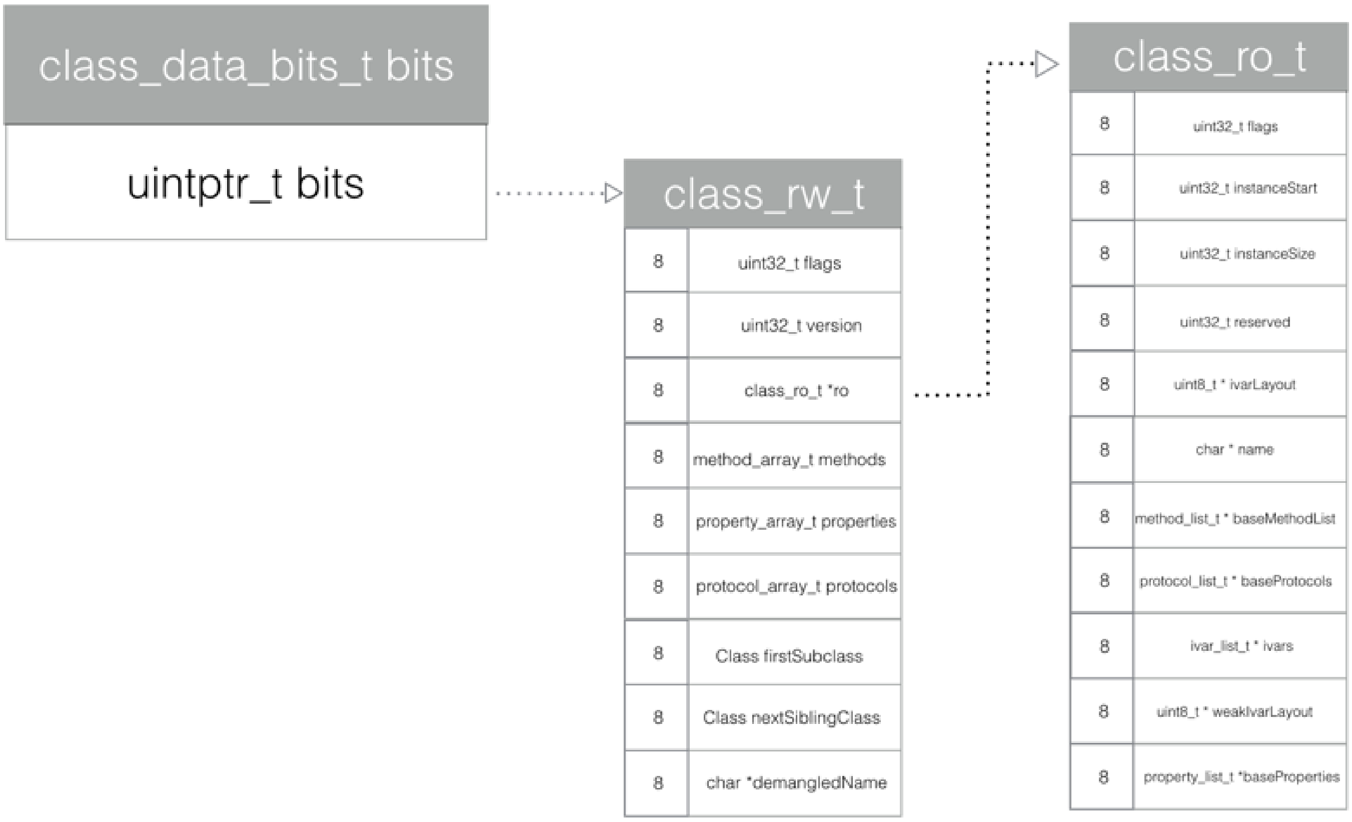
通过对Runtime库函数的直接调用

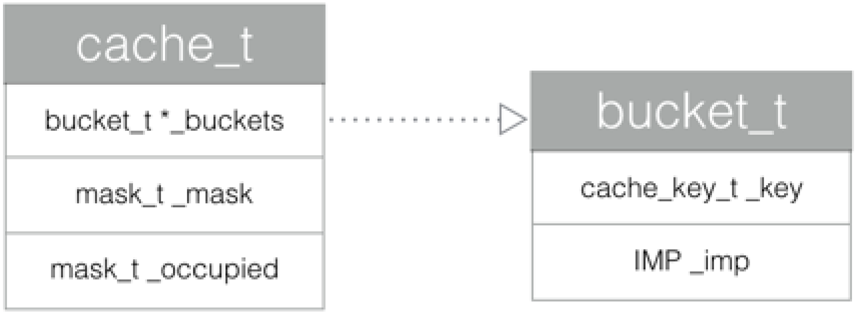
小Tips：Xcode上Enable Strict Checking of objc_msgSend Calls 参数可以添加代码提示

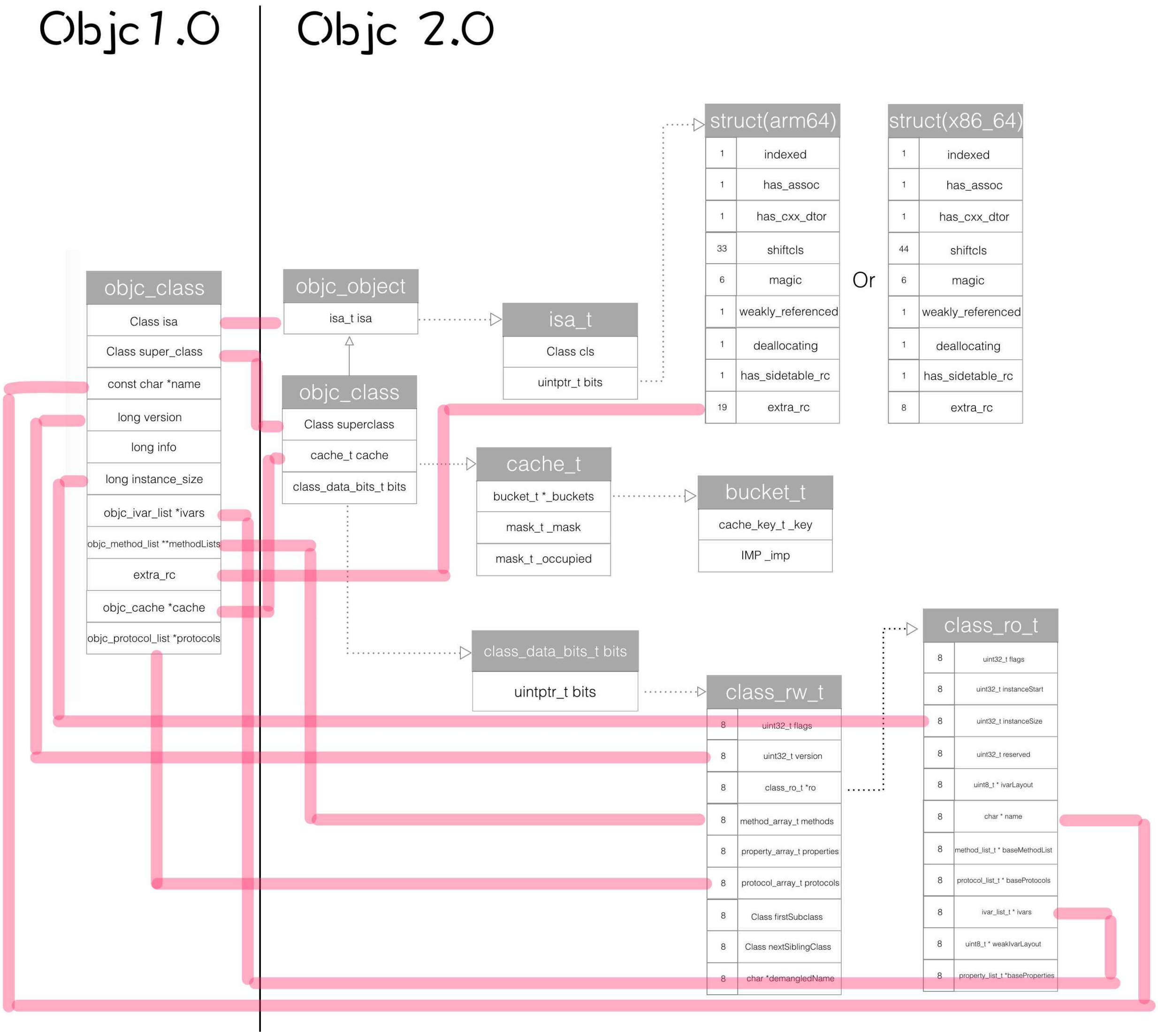
```
struct objc_class {
    Class isa;
 #if !__OBJC2__
    Class super_class;
    const char *name;
    long version;
    long info;
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists; // 这里是指针的指针，所以可以通过修改*methodLists的值来添加方法
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
#endif
}

// OBJC2.0 以后
typedef struct objc_class *Class;
typedef struct objc_object *id;

struct objc_object {
private:
    isa_t isa;
public:
    // initIsa() should be used to init the isa of new objects only
    // If this object already has an isa, use changeIsa() for correctness.
    // initInstanceIsa(): objects with no custom RR/AWZ
    void initIsa(Class cls);
    void initInstanceIsa(Class cls, bool hasCxxDtor);
private:
    void initIsa(Class newCls, bool indexed, bool hasCxxDtor);
}
struct objc_class : objc_object {
    Class superclass; // 父类指针
    cache_t cache; // 方法缓存，提高查找效率
    class_data_bits_t bits; // 实例方法链表，指向了类对象的数据区域，该数据区域内查找响应方法的对应实现
}

struct class_data_bits_t {// Values are the FAST_ flags above.
    uintptr_t bits;
```

```
}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;
}struct class_ro_t {
    uint32_t flags;
    uint32_t instanceStart;
    uint32_t instanceSize;
#ifdef __LP64__
    uint32_t reserved;
#endifconst 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;

    method_list_t *baseMethods() const {return baseMethodList;}
};

union isa_t
{
    isa_t() {}
    isa_t(uintptr_t value) : bits(value) {}
    Class cls;
    uintptr_t bits;
}

struct cache_t {
    struct bucket_t *_buckets;
    mask_t _mask;
    mask_t _occupied;
}
typedef unsigned int uint32_t;
typedef uint32_t mask_t;  // x86_64 & arm64 asm are less efficient with 16-bitstypedef unsigned long  uintptr_t;
typedef uintptr_t cache_key_t;struct bucket_t {
private:
    cache_key_t _key;
    IMP _imp;
}
```


objc_object
isa_t isa
isa_t
Class cls
uintptr_t bits
struct(arm64)
1 indexed
1 has_assoc
1 has_cxx_dtor
33 shiftcls
6 magic
1 weakly_referenced
1 deallocating
1 has_sidetable_rc
19 extra_rc
Or
struct(x86_64)
1 indexed
1 has_assoc
1 has_cxx_dtor
44 shiftcls
6 magic
1 weakly_referenced
1 deallocating
1 has_sidetable_rc
8 extra_rc
object
Class isa
objc_class
Class superclass
cache_t cache
class_data_bits_t bits
NSObject
Class isa


class_data_bits_t bits
uintptr_t bits
class_rw_t
8 uint32_t flags
8 uint32_t version
8 class_ro_t *ro
8 method_array_t methods
8 property_array_t properties
8 protocol_array_t protocols
8 Class firstSubclass
8 Class nextSiblingClass
8 char *demangledName
class_ro_t
8 uint32_t flags
8 uint32_t instanceStart
8 uint32_t instanceSize
8 uint32_t reserved
8 uint8_t * ivarLayout
8 char * name
8 method_list_t * baseMethodList
8 protocol_list_t * baseProtocols
8 ivar_list_t * ivars
8 uint8_t * weakIvarLayout
8 property_list_t *baseProperties





Objective-C对象都是C语言结构体实现的，在objc2.0中，所有对象都会包含一个isa_t类型的结构体；

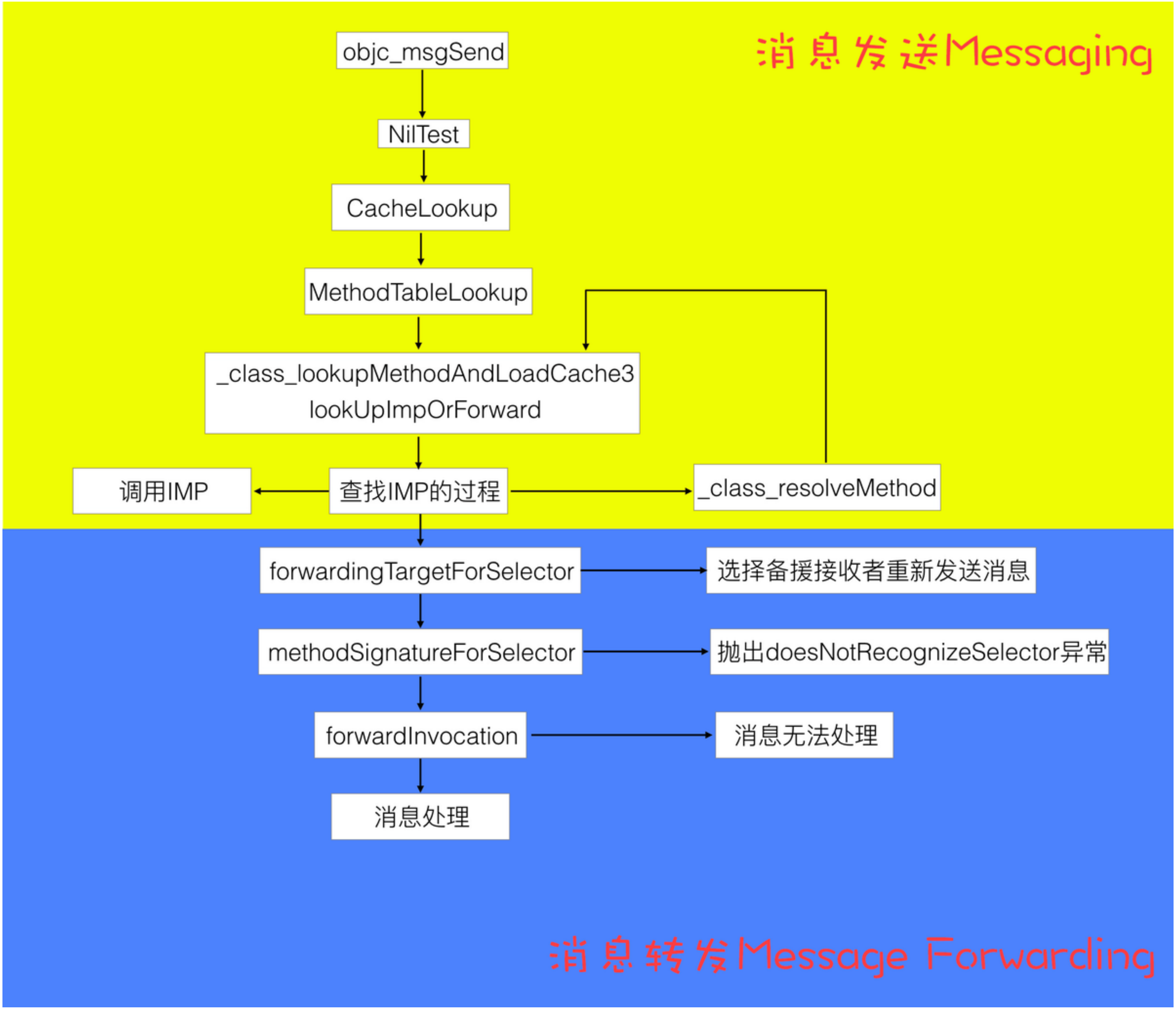
objc_object -- id类型 objc_class -- Class类型，二者存在继承关系，所以类也是一个对象。

当一个对象调用实例方法的时候，会通过isa找到对应的类，然后在该累得class_data_bits_t中去查找方法；

在调用类方法时，类的isa指向元类，通过引入元类的方式，类对象和对象查找方法的机制就完全统一了。

都是通过isa指针去找响应的类，然后通过类的 class_data_bits_t 查找响应方法的对应实现。

类对象和元类对象是唯一的，而实例对象是可以在运行时创建无数个的。在main方法执行之前，从dyld到runtime这期间，类对象和元类对象在这期间被创建。




```
//获取cls类对象所有成员ivar结构体
Ivar *class_copyIvarList(Class cls, unsigned int *outCount)
//获取cls类对象name对应的实例方法结构体
Method class_getInstanceMethod(Class cls, SEL name)
//获取cls类对象name对应类方法结构体
Method class_getClassMethod(Class cls, SEL name)
```

```
//获取cls类对象name对应方法imp实现
IMP class_getMethodImplementation(Class cls, SEL name)
//测试cls对应的实例是否响应sel对应的方法
BOOL class_respondsToSelector(Class cls, SEL sel)
//获取cls对应方法列表
Method *class_copyMethodList(Class cls, unsigned int *outCount)
//测试cls是否遵守protocol协议
BOOL class_conformsToProtocol(Class cls, Protocol *protocol)
//为cls类对象添加新方法
BOOL class_addMethod(Class cls, SEL name, IMP imp, const char *types)
//替换cls类对象中name对应方法的实现
IMP class_replaceMethod(Class cls, SEL name, IMP imp, const char *types)
//为cls添加新成员
BOOL class_addIvar(Class cls, const char *name, size_t size, uint8_t alignment, const char *types)
//为cls添加新属性
BOOL class_addProperty(Class cls, const char *name, const objc_property_attribute_t *attributes, unsigned int attributeCount)
//获取m对应的选择器
SEL method_getName(Method m)
//获取m对应的方法实现的imp指针
IMP method_getImplementation(Method m)
//获取m方法的对应编码
const char *method_getTypeEncoding(Method m)
//获取m方法参数的个数
unsigned int method_getNumberOfArguments(Method m)
//copy方法返回值类型
char *method_copyReturnType(Method m)
//获取m方法index索引参数的类型
char *method_copyArgumentType(Method m, unsigned int index)
//获取m方法返回值类型
void method_getReturnType(Method m, char *dst, size_t dst_len)
//获取方法的参数类型
void method_getArgumentType(Method m, unsigned int index, char *dst, size_t dst_len)
//设置m方法的具体实现指针
IMP method_setImplementation(Method m, IMP imp)
//交换m1，m2方法对应具体实现的函数指针
void method_exchangeImplementations(Method m1, Method m2)
//获取v的名称
const char *ivar_getName(Ivar v)
//获取v的类型编码
const char *ivar_getTypeEncoding(Ivar v)
//设置object对象关联的对象
void objc_setAssociatedObject(id object, const void *key, id value, objc_AssociationPolicy policy)
//获取object关联的对象
```

```
id objc_getAssociatedObject(id object, const void *key)
//移除object关联的对象
void objc_removeAssociatedObjects(id object)
```