> **摘要**：这篇文章首先介绍runtime原理，包括类，超类，元类，super_class，isa，对象，方法，SEL，IMP等概念，同时分别介绍与这些概念有关的API。接着介绍方法调用流程，以及寻找IMP的过程。然后，介绍一下这些API的常见用法，并介绍runtime的冷门知识。最后介绍一下runtime的实战指南。

Tips：苹果公开的源代码在这里可以查，opensource.apple.com/tarballs/

例如，其中，有两个比较常见需要学习源码的地址：

- runtime的源代码在opensource.apple.com/tarballs/ob...
- runloop(其实是整个 CoreFoundation)的源代码在 opensource.apple.com/tarballs/CF...

当然，如果你想在github上在线查看源代码，可以点这里：runtime，runloop

1. 运行时

1.1 基本概念: 运行时

Runtime 的概念

Runtime 又叫运行时，是一套底层的 C 语言 API，其为 iOS 内部的核心之一，我们平时编写的 OC 代码，底层都是基于它来实现的。比如：

```
// 发送消息
[receiver message];
```

```
// 底层运行时会被编译器转化为：
objc_msgSend(receiver, selector)
```

```
// 如果其还有参数比如：
[receiver message:(id)arg...];
```

```
// 底层运行时会被编译器转化为:
objc_msgSend(receiver, selector, arg1, arg2, ...)
```

以上你可能看不出它的价值，但是我们需要了解的是 Objective-C 是一门动态语言，它会将一些工作放在代码运行时才处理而并非编译时。也就是说，有很多类和成员变量在我们编译的时是不知道的，而在运行时，我们所编写的代码会转换成完整的确定的代码运行。

因此，编译器是不够的，我们还需要一个运行时系统(Runtime system)来处理编译后的代码。Runtime 基本是用 C 和汇编写的，由此可见苹果为了动态系统的高效而做出的努力。苹果和 GNU 各自维护一个开源的 Runtime 版本，这两个版本之间都在努力保持一致。

Runtime 的作用

Objc 在三种层面上与 Runtime 系统进行交互：

1. 通过 Objective-C 源代码
2. 通过 Foundation 框架的 NSObject 类定义的方法
3. 通过对 Runtime 库函数的直接调用

1.2 各种基本概念的C表达

在 Objective-C 中，类、对象和方法都是一个 C 的结构体，从 `objc/objc.h`（对象，`objc_object`，`id`）以及`objc/runtime.h`（其它，类，方法，方法列表，变量列表，属性列表等相关的）以及中，我们可以找到他们的定义。

① 类

类对象(Class)是由程序员定义并在运行时由编译器创建的，它没有自己的实例变量，这里需要注意的是类的**成员变量**和**实例方法列表**是属于实例对象的，但其存储于类对象当中的。我们在`objc/objc.h`下看看Class的定义：

- Class

```
/// An opaque type that represents an Objective-C class.
typedef struct objc_class *Class;
```

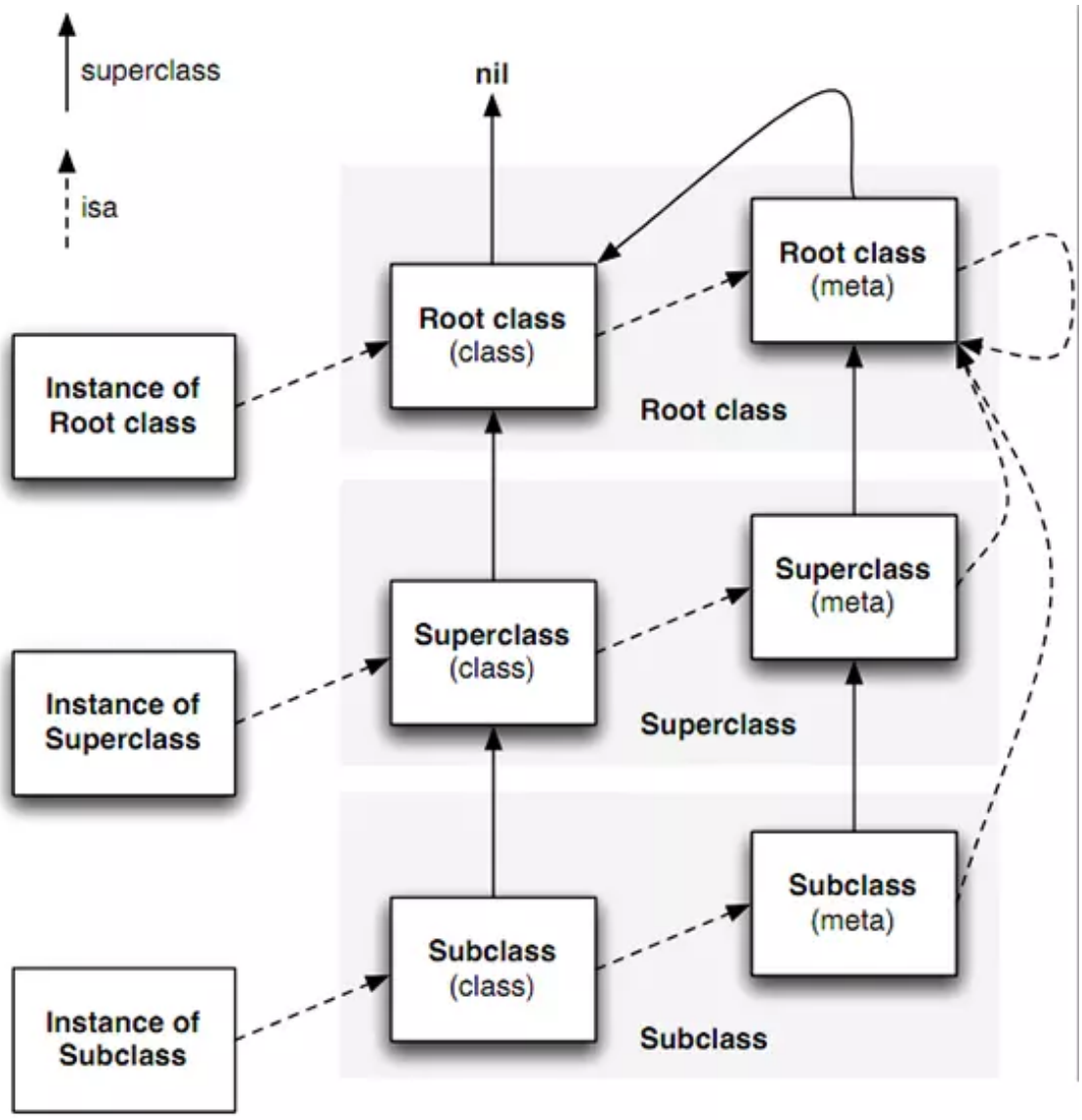
可以看到类是由Class类型来表示的，它是一个`objc_class`结构类型的指针。我们接着来看`objc_class`结构体的定义：

- objc_class

```
struct objc_class {
    Class _Nonnull isa  OBJC_ISA_AVAILABILITY;

#if !__OBJC2__
    Class _Nullable super_class
OBJC2_UNAVAILABLE;
    const char * _Nonnull name
OBJC2_UNAVAILABLE;
    long version
OBJC2_UNAVAILABLE;
    long info
OBJC2_UNAVAILABLE;
    long instance_size
OBJC2_UNAVAILABLE;
    struct objc_ivar_list * _Nullable ivars
OBJC2_UNAVAILABLE;
    struct objc_method_list * _Nullable * _Nullable methodLists
OBJC2_UNAVAILABLE;
    struct objc_cache * _Nonnull cache
OBJC2_UNAVAILABLE;
    struct objc_protocol_list * _Nullable protocols
OBJC2_UNAVAILABLE;
#endif

} OBJC2_UNAVAILABLE;
/* Use `Class` instead of `struct objc_class *` */
```

> 参数解析

- **isa指针**是和Class同类型的objc_class结构指针，类对象的指针指向其所属的类，即**元类**。元类中存储着类对象的类方法，当访问某个类的类方法时会通过该isa指针从元类中寻找方法对应的函数指针。
- **super_class指针**指向该类所继承的**父类对象**，如果该类已经是最顶层的根类(如NSObject或NSProxy), 则 super_class为NULL。



- **cache**：用于**缓存**最近使用的方法。一个接收者对象接收到一个消息时，它会根据 `isa` 指针去查找能够响应这个消息的对象。在实际使用中，这个对象只有一部分方法是常用的，很多方法其实很少用或者根本用不上。这种情况下，如果每次消息来时，我们都是 `methodLists` 中遍历一遍，性能势必很差。这时，cache就派上用场了。在我们每次调用过一个方法后，这个方法就会被**缓存**到cache列表中，下次调用的时候runtime就会优先去cache中查找，如果cache没有，才去 `methodLists` 中查找方法。这样，对于那些经常用到的方法的调用，但提高了调用的效率。

- **version**：我们可以使用这个字段来提供类的版本信息。这对于对象的序列化非常有用，它可是让我们识别出不同类定义版本中实例变量布局的改变。

- **protocols**：当然可以看出这一个`objc_protocol_list`的指针。关于`objc_protocol_list`的结构体构成后面会讲。

> 获取类的类名

```
// 获取类的类名
const char * class_getName ( Class cls );
```

> 动态创建类

```
// 创建一个新类和元类
Class objc_allocateClassPair ( Class superclass, const char *name,
size_t extraBytes ); //如果创建的是root class，则superclass为Nil。
extraBytes通常为0

// 销毁一个类及其相关联的类
void objc_disposeClassPair ( Class cls ); //在运行中还存在或存在子类实
例，就不能够调用这个。

// 在应用中注册由objc_allocateClassPair创建的类
void objc_registerClassPair ( Class cls ); //创建了新类后，然后使
用class_addMethod，class_addIvar函数为新类添加方法，实例变量和属性后再调用这
个来注册类，再之后就能够用了。
```

② 对象

实例对象是我们对类对象`alloc`或者`new`操作时所创建的，在这个过程中会**拷贝**实例所属的类的**成员变量**，但并不拷贝类定义的**方法**。调用实例方法时，系统会根据实例的`isa`指针去类的方法列表及父类的**方法列表**中寻找与消息对应的`selector`指向的方法。同样的，我们也来看下其定义:

- objc_object

```
/// Represents an instance of a class.
struct objc_object {
    Class _Nonnull isa  OBJC_ISA_AVAILABILITY;
};
```

可以看到，这个结构体只有一个`isa`变量，指向实例对象所属的**类**。任何带有以**指针**开始并指向**类结构**的结构都可以被视作`objc_object`，对象最重要的特点是可以给其发送消息。 NSObject类的`alloc`和`allocWithZone:`方法使用函数`class_createInstance`来创建`objc_object`数据结构。

另外我们常见的`id`类型，它是一个`objc_object`结构类型的指针。该类型的对象可以转换为任何一种对象，类似于C语言中`void *`指针类型的作用。其定义如下所示:

- id

```
/// A pointer to an instance of a class.
typedef struct objc_object *id;
#endif
```

> 对对象的类操作

```
// 返回给定对象的类名
const char * object_getClassName ( id obj );
// 返回对象的类
Class object_getClass ( id obj );
// 设置对象的类
Class object_setClass ( id obj, Class cls );
```

> 获取对象的类定义

```
// 获取已注册的类定义的列表
int objc_getClassList ( Class *buffer, int bufferCount );

// 创建并返回一个指向所有已注册类的指针列表
Class * objc_copyClassList ( unsigned int *outCount );

// 返回指定类的类定义
Class objc_lookUpClass ( const char *name );
Class objc_getClass ( const char *name );
Class objc_getRequiredClass ( const char *name );

// 返回指定类的元类
Class objc_getMetaClass ( const char *name );
```

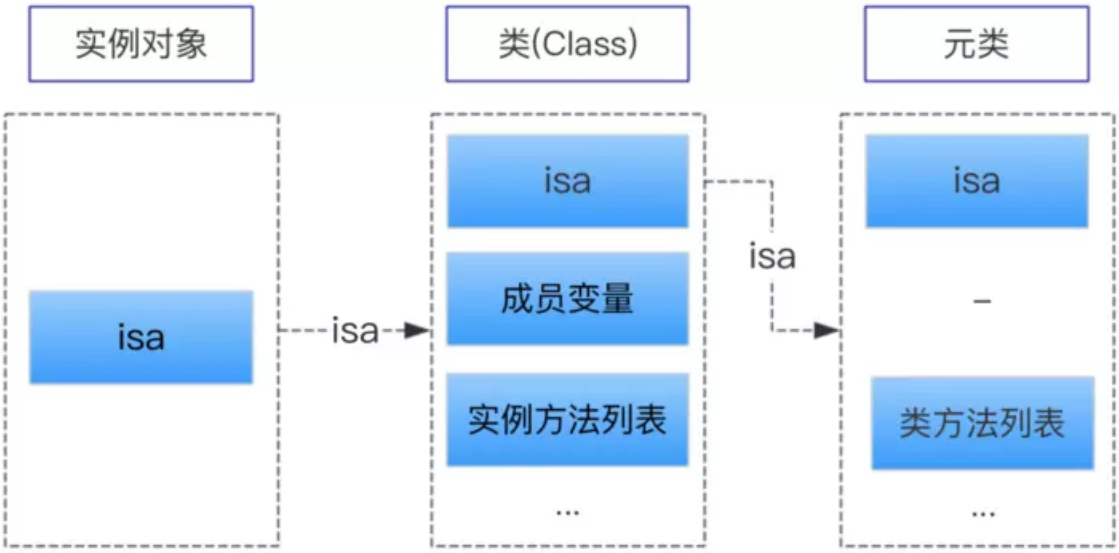
> 动态创建对象

```
// 创建类实例
id class_createInstance ( Class cls, size_t extraBytes ); //会在heap里给类分配内存。这个方法和+alloc方法类似。

// 在指定位置创建类实例
id objc_constructInstance ( Class cls, void *bytes );

// 销毁类实例
void * objc_destructInstance ( id obj ); //不会释放移除任何相关引用
```

③ 元类

元类(Metaclass)就是**类对象**的**类**，每个**类**都有自己的**元类**，也就是`objc_class`结构体里面`isa`指针所指向的类. Objective-C的类方法是使用元类的根本原因，因为其中存储着对应的类对象调用的方法即**类方法**。

当向对象发消息，runtime会在这个对象所属类方法列表中查找发送消息对应的方法，但当向类发送消息时，runtime就会在这个类的meta class方法列表里查找。所有的meta class，包括Root class，Superclass，Subclass的isa都指向Root class的meta class，这样能够形成一个闭环。



所以由上图可以看到，在给实例对象或类对象发送消息时，寻找方法列表的规则为:

- 当发送消息给实例对象时，消息是在寻找这个对象的类的方法列表(实例方法)
- 当发送消息给类对象时，消息是在寻找这个类的元类的方法列表(类方法)

元类，就像之前的类一样，它也是一个对象，也可以调用它的方法。所以这就意味着它必须也有一个类。所有的元类都使用根元类作为他们的类。比如所有NSObject的子类的元类都会以NSObject的元类作为他们的类。

根据这个规则，所有的元类使用根元类作为他们的类，根元类的元类则就是它自己。也就是说基类的元类的isa指针指向他自己。

> 操作函数

- super_class和meta-class

```
// 获取类的父类
Class class_getSuperclass ( Class cls );
// 判断给定的Class是否是一个meta class
BOOL class_isMetaClass ( Class cls );
```

- instance_size

```
// 获取实例大小
size_t class_getInstanceSize ( Class cls );
```

④ 属性

在Objective-C中，属性(property)和成员变量是不同的。那么，属性的本质是什么？它和成员变量之间有什么区别？简单来说属性是添加了存取方法的成员变量，也就是:

> @property = ivar + getter + setter;

因此，我们每定义一个@property都会添加对应的ivar, getter和setter到类结构体`objc_class`中。具体来说，系统会在`objc_ivar_list`中添加一个成员变量的描述，然后在`methodLists`中分别添加setter和getter方法的描述。下面的`objc_property_t`是声明的属性的类型，是一个指向objc_property结构体的指

针。

用法举例

```
//遍历获取所有属性Property
- (void) getAllProperty {
    unsigned int propertyCount = 0;
    objc_property_t *propertyList = class_copyPropertyList([Person
class], &propertyCount);
    for (unsigned int i = 0; i < propertyCount; i++ ) {
        objc_property_t *thisProperty = propertyList[i];
        const char* propertyName = property_getName(*thisProperty);
        NSLog(@"Person拥有的属性为: '%s'", propertyName);
    }
}
```

- objc_property_t

```
/// An opaque type that represents an Objective-C declared
property.
typedef struct objc_property *objc_property_t;
```

另外，关于属性有一个`objc_property_attribute_t`结构体列表，`objc_property_attribute_t`结构体包含`name`和`value`

- objc_property_attribute_t

```
typedef struct {
    const char * _Nonnull name;           /**< The name of the
attribute */
    const char * _Nonnull value;          /**< The value of the
attribute (usually empty) */
} objc_property_attribute_t;
```

> 常用的属性如下：

- 属性类型 name值：T value：变化
- 编码类型 name值：C(copy) &(strong) W(weak)空(assign) 等 value：无

- 非/原子性 name值：空(atomic) N(Nonatomic) value：无
- 变量名称 name值：V value：变化

例如

```
@interface person : NSObjec{
  NSString *_name;
}
int main(){
  objc_property_attribute_t nonatomic = {"N", ""};
  objc_property_attribute_t strong = {"&", ""};
  objc_property_attribute_t type = {"T", "@\"NSString\""};
  objc_property_attribute_t ivar = {"V", "_name"};
  objc_property_attribute_t attributes[] = {nonatomic, strong,
type, ivar};
  BOOL result = class_addProperty([person class], "name",
attributes, 4);
}
```

> 操作函数

```
// 获取属性名
const char * property_getName ( objc_property_t property );
// 获取属性特性描述字符串
const char * property_getAttributes ( objc_property_t property );
// 获取属性中指定的特性
char * property_copyAttributeValue ( objc_property_t property,
const char *attributeName );
// 获取属性的特性列表
objc_property_attribute_t * property_copyAttributeList (
objc_property_t property, unsigned int *outCount );
```

⑤ 成员变量

Ivar: 实例变量类型，是一个指向`objc_ivar`结构体的指针

- Ivar

```
/// An opaque type that represents an instance variable.
typedef struct objc_ivar *Ivar;
```

`objc_ivar` 结构体的组成如下：

- objc_ivar

```
- struct objc_ivar {
    char * _Nullable ivar_name
OBJC2_UNAVAILABLE;
    char * _Nullable ivar_type
OBJC2_UNAVAILABLE;
    int ivar_offset
OBJC2_UNAVAILABLE;
#ifdef __LP64__
    int space
OBJC2_UNAVAILABLE;
#endif
}
```

这里我们注意第三个成员 `ivar_offset` 。它表示基地址偏移字节。

> 操作函数

```
//成员变量操作函数
// 修改类实例的实例变量的值
Ivar object_setInstanceVariable ( id obj, const char *name, void
*value );
// 获取对象实例变量的值
Ivar object_getInstanceVariable ( id obj, const char *name, void
**outValue );
// 返回指向给定对象分配的任何额外字节的指针
void * object_getIndexedIvars ( id obj );
// 返回对象中实例变量的值
id object_getIvar ( id obj, Ivar ivar );
// 设置对象中实例变量的值
void object_setIvar ( id obj, Ivar ivar, id value );

// 获取类成员变量的信息
Ivar class_getClassVariable ( Class cls, const char *name );
// 添加成员变量
BOOL class_addIvar ( Class cls, const char *name, size_t size,
uint8_t alignment, const char *types ); //这个只能够向在runtime时创建的
```

```
类添加成员变量
// 获取整个成员变量列表
Ivar * class_copyIvarList ( Class cls, unsigned int *outCount ); //
必须使用free()来释放这个数组
```

⑥ 成员变量列表

在`objc_class`中，所有的成员变量、属性的信息是放在链表`ivars`中的。`ivars`是一个数组，数组中每个元素是指向`Ivar`(变量信息)的指针。

- objc_ivar_list

```
struct objc_ivar_list {
    int ivar_count
OBJC2_UNAVAILABLE;
#ifdef __LP64__
    int space
OBJC2_UNAVAILABLE;
#endif
    /* variable length structure */
    struct objc_ivar ivar_list[1]
OBJC2_UNAVAILABLE;
}
```

例子，获取所有成员变量

```
//遍历获取Person类所有的成员变量IvarList
- (void) getAllIvarList {
    unsigned int methodCount = 0;
    Ivar * ivars = class_copyIvarList([Person class],
&methodCount);
    for (unsigned int i = 0; i < methodCount; i ++) {
        Ivar ivar = ivars[i];
        const char * name = ivar_getName(ivar);
        const char * type = ivar_getTypeEncoding(ivar);
        NSLog(@"Person拥有的成员变量的类型为%s，名字为 %s ",type, name);
    }
    free(ivars);
}
```

⑦ 方法

Method 代表类中某个方法的类型

- Method

```
/// An opaque type that represents a method in a class definition.
typedef struct objc_method *Method;
```

objc_method 存储了方法名，方法类型和方法实现：

- objc_method

```
struct objc_method {
    SEL _Nonnull method_name
OBJC2_UNAVAILABLE;
    char * _Nullable method_types
OBJC2_UNAVAILABLE;
    IMP _Nonnull method_imp
OBJC2_UNAVAILABLE;
}
OBJC2_UNAVAILABLE;
```

其中，

- 方法名类型为 SEL
- 方法类型 method_types 是个 char 指针，存储方法的参数类型和返回值类型
- method_imp 指向了方法的实现，本质是一个函数指针

简言之，Method = SEL + IMP + method_types，相当于在SEL和IMP之间建立了一个映射。

> 操作函数

```
// 调用指定方法的实现，返回的是方法实现时的返回，参数receiver不能为空，这个比method_getImplementation和method_getName快
id method_invoke ( id receiver, Method m, ... );
// 调用返回一个数据结构的方法的实现
void method_invoke_stret ( id receiver, Method m, ... );
```

```
// 获取方法名，希望获得方法明的C字符串，使用
sel_getName(method_getName(method))
SEL method_getName ( Method m );
// 返回方法的实现
IMP method_getImplementation ( Method m );
// 获取描述方法参数和返回值类型的字符串
const char * method_getTypeEncoding ( Method m );
// 获取方法的返回值类型的字符串
char * method_copyReturnType ( Method m );
// 获取方法的指定位置参数的类型字符串
char * method_copyArgumentType ( Method m, unsigned int index );
// 通过引用返回方法的返回值类型字符串
void method_getReturnType ( Method m, char *dst, size_t dst_len );
// 返回方法的参数的个数
unsigned int method_getNumberOfArguments ( Method m );
// 通过引用返回方法指定位置参数的类型字符串
void method_getArgumentType ( Method m, unsigned int index, char
*dst, size_t dst_len );
// 返回指定方法的方法描述结构体
struct objc_method_description * method_getDescription ( Method m
);
// 设置方法的实现
IMP method_setImplementation ( Method m, IMP imp );
// 交换两个方法的实现
void method_exchangeImplementations ( Method m1, Method m2 );
```

⑧ 方法列表

方法调用是通过查询对象的isa指针所指向归属类中的methodLists来完成。

- objc_method_list

```
struct objc_method_list {
    struct objc_method_list * _Nullable obsolete
OBJC2_UNAVAILABLE;

    int method_count
OBJC2_UNAVAILABLE;
#ifdef __LP64__
    int space
OBJC2_UNAVAILABLE;
#endif
    /* variable length structure */
    struct objc_method method_list[1]
```

```
OBJC2_UNAVAILABLE;
}
```

> 操作函数

```
// 添加方法
BOOL class_addMethod ( Class cls, SEL name, IMP imp, const char
*types ); //和成员变量不同的是可以为类动态添加方法。如果有同名会返回NO，修改的
话需要使用method_setImplementation

// 获取实例方法
Method class_getInstanceMethod ( Class cls, SEL name );

// 获取类方法
Method class_getClassMethod ( Class cls, SEL name );

// 获取所有方法的数组
Method * class_copyMethodList ( Class cls, unsigned int *outCount
);

// 替代方法的实现
IMP class_replaceMethod ( Class cls, SEL name, IMP imp, const char
*types );

// 返回方法的具体实现
IMP class_getMethodImplementation ( Class cls, SEL name );
IMP class_getMethodImplementation_stret ( Class cls, SEL name );

// 类实例是否响应指定的selector
BOOL class_respondsToSelector ( Class cls, SEL sel );
```

⑨ 指针：指向方法名与实现

- SEL

```
/// An opaque type that represents a method selector.
typedef struct objc_selector *SEL;
```

在源码中没有直接找到 `objc_selector` 的定义，从一些书籍上与 Blog 上看到可以

将 SEL 理解为一个 `char*` 指针。

具体这 `objc_selector` 结构体是什么取决与使用GNU的还是Apple的运行时， 在Mac OS X中SEL其实被映射为一个C**字符串**，可以看作是方法的名字，它并不一个指向具体方法实现（IMP类型才是）。

对于所有的类，只要方法名是相同的，产生的selector都是一样的。

> 操作函数

```
// 返回给定选择器指定的方法的名称
const char * sel_getName ( SEL sel );
// 在Objective-C Runtime系统中注册一个方法，将方法名映射到一个选择器，并返回这个选择器
SEL sel_registerName ( const char *str );
// 在Objective-C Runtime系统中注册一个方法
SEL sel_getUid ( const char *str );
// 比较两个选择器
BOOL sel_isEqual ( SEL lhs, SEL rhs );
```

- IMP

```
/// A pointer to the function of a method implementation.
#if !OBJC_OLD_DISPATCH_PROTOTYPES
typedef void (*IMP)(void /* id, SEL, ... */ );
#else
typedef id _Nullable (*IMP)(id _Nonnull, SEL _Nonnull, ...);
#endif
```

实际上就是一个函数指针，指向方法实现的首地址。通过取得 IMP，我们可以跳过 runtime 的消息传递机制，直接执行 IMP指向的函数实现，这样省去了 runtime 消息传递过程中所做的一系列查找操作，会比直接向对象发送消息高效一些，当然必须说明的是，这种方式只适用于极特殊的优化场景，如效率敏感的场景下大量循环的调用某方法。

> 操作函数

```
// 返回方法的实现
IMP method_getImplementation ( Method m );
```

```
// 设置方法的实现
IMP method_setImplementation ( Method m, IMP imp );
// 替代方法的实现
IMP class_replaceMethod ( Class cls, SEL name, IMP imp, const char
*types );
```

⑩ 缓存

上面提到了objc_class结构体中的cache字段，它用于缓存调用过的方法。这个字段是一个指向objc_cache结构体的指针，其定义如下：

- Cache

```
typedef struct objc_cache *Cache
```

- objc_cache

```
typedef struct objc_cache *Cache
OBJC2_UNAVAILABLE;

#define CACHE_BUCKET_NAME(B)  ((B)->method_name)
#define CACHE_BUCKET_IMP(B)   ((B)->method_imp)
#define CACHE_BUCKET_VALID(B) (B)
#ifndef __LP64__
#define CACHE_HASH(sel, mask) (((uintptr_t)(sel)>>2) & (mask))
#else
#define CACHE_HASH(sel, mask) (((unsigned int)((uintptr_t)
(sel)>>3)) & (mask))
#endif
struct objc_cache {
    unsigned int mask /* total = mask + 1 */
OBJC2_UNAVAILABLE;
    unsigned int occupied
OBJC2_UNAVAILABLE;
    Method _Nullable buckets[1]
OBJC2_UNAVAILABLE;
};
```

该结构体的字段描述如下：

- mask：一个整数，指定分配的缓存bucket的总数。在方法查找过程中，Objective-C runtime使用这个字段来确定开始线性查找数组的索引位置。指向方法selector的指针与该字段做一个AND位操作(index = (mask & selector))。这可以作为一个简单的hash散列算法。
- occupied：一个整数，指定实际占用的缓存bucket的总数。
- buckets：指向Method数据结构指针的数组。这个数组可能包含不超过mask+1个元素。需要注意的是，指针可能是NULL，表示这个缓存bucket没有被占用，另外被占用的bucket可能是不连续的。这个数组可能会随着时间而增长。

⑪ 协议链表

前面`objc_class`的结构体中有个协议链表的参数，协议链表用来存储声明遵守的正式协议

- objc_protocol_list

```
struct objc_protocol_list {
    struct objc_protocol_list * _Nullable next;
    long count;
    __unsafe_unretained Protocol * _Nullable list[1];
};
```

> 操作函数

```
// 添加协议
BOOL class_addProtocol ( Class cls, Protocol *protocol );
// 返回类是否实现指定的协议
BOOL class_conformsToProtocol ( Class cls, Protocol *protocol );
// 返回类实现的协议列表
Protocol * class_copyProtocolList ( Class cls, unsigned int
*outCount );

// 返回指定的协议
Protocol * objc_getProtocol ( const char *name );
// 获取运行时所知道的所有协议的数组
Protocol ** objc_copyProtocolList ( unsigned int *outCount );
// 创建新的协议实例
Protocol * objc_allocateProtocol ( const char *name );
// 在运行时中注册新创建的协议
void objc_registerProtocol ( Protocol *proto ); //创建一个新协议后必须
```

```
使用这个进行注册这个新协议，但是注册后不能够再修改和添加新方法。

// 为协议添加方法
void protocol_addMethodDescription ( Protocol *proto, SEL name,
const char *types, BOOL isRequiredMethod, BOOL isInstanceMethod );
// 添加一个已注册的协议到协议中
void protocol_addProtocol ( Protocol *proto, Protocol *addition );
// 为协议添加属性
void protocol_addProperty ( Protocol *proto, const char *name,
const objc_property_attribute_t *attributes, unsigned int
attributeCount, BOOL isRequiredProperty, BOOL isInstanceProperty );
// 返回协议名
const char * protocol_getName ( Protocol *p );
// 测试两个协议是否相等
BOOL protocol_isEqual ( Protocol *proto, Protocol *other );
// 获取协议中指定条件的方法的方法描述数组
struct objc_method_description * protocol_copyMethodDescriptionList
( Protocol *p, BOOL isRequiredMethod, BOOL isInstanceMethod,
unsigned int *outCount );
// 获取协议中指定方法的方法描述
struct objc_method_description protocol_getMethodDescription (
Protocol *p, SEL aSel, BOOL isRequiredMethod, BOOL isInstanceMethod
);
// 获取协议中的属性列表
objc_property_t * protocol_copyPropertyList ( Protocol *proto,
unsigned int *outCount );
// 获取协议的指定属性
objc_property_t protocol_getProperty ( Protocol *proto, const char
*name, BOOL isRequiredProperty, BOOL isInstanceProperty );
// 获取协议采用的协议
Protocol ** protocol_copyProtocolList ( Protocol *proto, unsigned
int *outCount );
// 查看协议是否采用了另一个协议
BOOL protocol_conformsToProtocol ( Protocol *proto, Protocol *other
);
```

⑫ 分类

- Category

```
/// An opaque type that represents a category.
typedef struct objc_category *Category;
```

- objc_category

```
struct objc_category {
    char * _Nonnull category_name
OBJC2_UNAVAILABLE;
    char * _Nonnull class_name
OBJC2_UNAVAILABLE;
    struct objc_method_list * _Nullable instance_methods
OBJC2_UNAVAILABLE;
    struct objc_method_list * _Nullable class_methods
OBJC2_UNAVAILABLE;
    struct objc_protocol_list * _Nullable protocols
OBJC2_UNAVAILABLE;
}
```

2. 方法调用流程

objc_msgSend() Tour 系列文章通过对 `objc_msgSend` 的汇编源码分析，总结出以下流程：

2.1 方法调用流程

1. 检查 selector 是否需要忽略
2. 检查 target 是否为 nil，如果是 nil 就直接 cleanup，然后 return
3. 在 target 的 Class 中根据 selector 去找 IMP

2.2 寻找 IMP 的过程:

1. 在当前 class 的方法缓存里寻找（cache methodLists）
2. 找到了跳到对应的方法实现，没找到继续往下执行
3. 从当前 class 的 方法列表里查找（methodLists），找到了添加到缓存列表里，然后跳转到对应的方法实现；没找到继续往下执行
4. 从 superClass 的缓存列表和方法列表里查找，直到找到基类为止
5. 以上步骤还找不到 IMP，则进入消息动态处理和消息转发流程，详见这篇文章

我们能在 objc4官方源码 中找到上述寻找 IMP 的过程，具体对应的代码如下：

objc-class.mm

```
IMP class_getMethodImplementation(Class cls, SEL sel)
{
    IMP imp;

    if (!cls  ||  !sel) return nil;

    imp = lookUpImpOrNil(cls, sel, nil,
                         YES/*initialize*/, YES/*cache*/,
YES/*resolver*/);

    // Translate forwarding function to C-callable external version
    if (!imp) {
        return _objc_msgForward;
    }

    return imp;
}
```

objc-runtime-new.mm

```
IMP lookUpImpOrNil(Class cls, SEL sel, id inst,
                   bool initialize, bool cache, bool resolver)
{
    IMP imp = lookUpImpOrForward(cls, sel, inst, initialize, cache,
resolver);
    if (imp == _objc_msgForward_impcache) return nil;
    else return imp;
}
```

等等...

3. 运行时相关的API

3.1 通过 Foundation 框架的 NSObject 类定义的方法

Cocoa 程序中绝大部分类都是 NSObject 类的子类，所以都继承了 NSObject 的行为。(NSProxy 类时个例外，它是个抽象超类)

一些情况下，NSObject 类仅仅定义了完成某件事情的模板，并没有提供所需要的代

码。例如 -description 方法，该方法返回类内容的字符串表示，该方法主要用来调试程序。NSObject 类并不知道子类的内容，所以它只是返回类的名字和对象的地址，NSObject 的子类可以重新实现。

还有一些 NSObject 的方法可以从 Runtime 系统中获取信息，允许对象进行自我检查。例如：

- `-class` 方法返回对象的类；
- `-isKindOfClass:` 和 `-isMemberOfClass:` 方法检查对象是否存在于指定的类的继承体系中(是否是其子类或者父类或者当前类的成员变量)；
- `-respondsToSelector:` 检查对象能否响应指定的消息；
- `-conformsToProtocol:` 检查对象是否实现了指定协议类的方法；
- `-methodForSelector:` 返回指定方法实现的地址。

常见的一个例子：

```
//先调用respondsToSelector:来判断一下
if ([self respondsToSelector:@selector(method)]) {
     [self performSelector:@selector(method)];
}
```

3.2 runtime的常见API举例

- 获取列表及名字

```
    unsigned int count;
    //获取属性列表
    objc_property_t *propertyList = class_copyPropertyList([self
class], &count);
    for (unsigned int i=0; i<count; i++) {
        const char *propertyName =
property_getName(propertyList[i]);
        NSLog(@"property---->%@", [NSString
stringWithUTF8String:propertyName]);
    }

    //获取方法列表
    Method *methodList = class_copyMethodList([self class],
&count);
    for (unsigned int i; i<count; i++) {
```

```
        Method method = methodList[i];
        NSLog(@"method---->%@",
NSStringFromSelector(method_getName(method)));
    }

    //获取成员变量列表
    Ivar *ivarList = class_copyIvarList([self class], &count);
    for (unsigned int i; i<count; i++) {
        Ivar myIvar = ivarList[i];
        const char *ivarName = ivar_getName(myIvar);
        NSLog(@"Ivar---->%@", [NSString
stringWithUTF8String:ivarName]);
    }

    //获取协议列表
    __unsafe_unretained Protocol **protocolList =
class_copyProtocolList([self class], &count);
    for (unsigned int i; i<count; i++) {
        Protocol *myProtocal = protocolList[i];
        const char *protocolName = protocol_getName(myProtocal);
        NSLog(@"protocol---->%@", [NSString
stringWithUTF8String:protocolName]);
    }
```

- 动态创建类

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

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

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

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

输出结果

```
2014-10-23 11:35:31.006 RuntimeTest[3800:66152] run sub method 1
2014-10-23 11:35:31.006 RuntimeTest[3800:66152] run sub method 1
```

- 动态创建对象

```
//可以看出class_createInstance和alloc的不同
id theObject = class_createInstance(NSString.class,
sizeof(unsigned));
id str1 = [theObject init];
NSLog(@"%@", [str1 class]);
id str2 = [[NSString alloc] initWithString:@"test"];
NSLog(@"%@", [str2 class]);
```

输出结果

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

3.3 runtime的少见API解析

- `id` 和 `void *` 转换API：`(__bridge void *)`

在 ARC 有效时，通过 `(__bridge void *)` 转换 `id` 和 `void *` 就能够相互转换。为什么转换？这是因为 `objc_getAssociatedObject` 的参数要求的。先看一下它的API：

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

可以知道，这个“属性名”的key是必须是一个`void *`类型的参数。所以需要转换。关于这个转换，下面给一个转换的例子：

```
id obj = [[NSObject alloc] init];

void *p = (__bridge void *)obj;
id o = (__bridge id)p;
```

关于这个转换可以了解更多：ARC 类型转换：显示转换 id 和 void *

当然，如果不通过转换使用这个API，就需要这样使用：

- 方式1：

```
objc_getAssociatedObject(self, @"AddClickedEvent");
```

- 方式2：

```
static const void *registerNibArrayKey = &registerNibArrayKey;
```

```
NSMutableArray *array = objc_getAssociatedObject(self,
registerNibArrayKey);
```

- 方式3：

```
static const char MJErrorKey = '\0';
```

```
objc_getAssociatedObject(self, &MJErrorKey);
```

- 方式4：

```
+ (instancetype)cachedPropertyWithProperty:
(objc_property_t)property
{
    MJProperty *propertyObj = objc_getAssociatedObject(self,
property);
    //省略
}
```

其中`objc_property_t`是runtime的类型

```
typedef struct objc_property *objc_property_t;
```

4. 运行时实战指南

上面的API不是提供大家背的，而是用来**查阅**的，当你要用到的时候查阅。因为这些原理和API光看没用，需要实战之后再回过头来查阅和理解。笔者另外写了runtime的原理与实践。如果想了解runtime的更多知识，可以选择阅读这些文章：

- iOS开发·runtime原理与实践: 消息转发篇
- iOS开发·runtime原理与实践: 关联对象篇
- iOS开发·runtime原理与实践: 方法交换篇
- iOS开发·KVO用法，原理与底层实现: runtime模拟实现KVO监听机制（Blcok及Delgate方式）