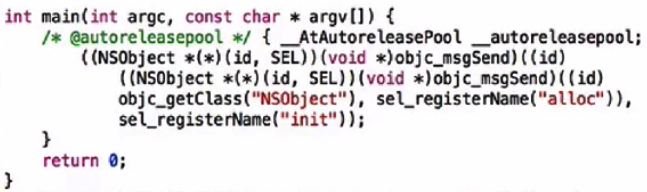
# OC最实用的runtime总结，面试、工作你看我就足够了！

# 前言

runtime的资料网上有很多了，部分有些晦涩难懂，我通过自己的学习方法总结一遍，主要讲一些常用的方法功能，以实用为主，我觉得用到印象才是最深刻的，并且最后两个demo也是MJExtension的实现原理，面试的时候也可以多扯点。  
另外runtime的知识还有很多，想要了解更多可以看我翻译的[官方文档](http://www.jianshu.com/p/158c5d118937" \t "http://www.jianshu.com/p/_blank)（有点枯燥），本文的demo[下载地址](https://github.com/XHTeng/XHRuntimeDemo" \t "http://www.jianshu.com/p/_blank)

# 什么是runtime？

runtime 是 OC底层的一套C语言的API（引入 <objc/runtime.h> 或<objc/message.h>），编译器最终都会将OC代码转化为运行时代码，通过终端命令编译.m 文件：clang -rewrite-objc xxx.m可以看到编译后的xxx.cpp（C++文件）。  
比如我们创建了一个对象 [[NSObject alloc]init]，最终被转换为几万行代码，截取最关键的一句可以看到底层是通过runtime创建的对象



.cpp 文件

删除掉一些强制转换语句，可以看到调用方法本质就是发消息，[[NSObject alloc]init]语句发了两次消息，第一次发了alloc 消息，第二次发送init 消息。利用这个功能我们可以探究底层，比如block的实现原理。  
需要注意的是，使用objc\_msgSend() sel\_registerName()方法需要导入头文件<objc/message.h>

IMG_257

消息机制

另外利用runtime 可以做一些OC不容易实现的功能

* 动态交换两个方法的实现（特别是交换系统自带的方法）
* 动态添加对象的成员变量和成员方法
* 获得某个类的所有成员方法、所有成员变量

# 如何应用运行时？

1.将某些OC代码转为运行时代码，探究底层，比如block的实现原理（上边已讲到）；  
2.拦截系统自带的方法调用（Swizzle 黑魔法），比如拦截imageNamed:、viewDidLoad、alloc；  
3.实现分类也可以增加属性；  
4.实现NSCoding的自动归档和自动解档；  
5.实现字典和模型的自动转换。

## 下面我通过demo 我一个个来讲解

#### 一、交换两个方法的实现，拦截系统自带的方法调用功能

需要用到的方法 <objc/runtime.h>

* 获得某个类的类方法

Method class\_getClassMethod(Class cls , SEL name)

* 获得某个类的实例对象方法

Method class\_getInstanceMethod(Class cls , SEL name)

* 交换两个方法的实现

void method\_exchangeImplementations(Method m1 , Method m2)

###### 案例1：方法简单的交换

创建一个Person类，类中实现以下两个类方法，并在.h 文件中声明

+ (void)run {

NSLog(@"跑");

}

+ (void)study {

NSLog(@"学习");

}

控制器中调用，则先打印跑，后打印学习

[Person run];[Person study];

下面通过runtime 实现方法交换，类方法用class\_getClassMethod ，对象方法用class\_getInstanceMethod

// 获取两个类的类方法Method m1 = class\_getClassMethod([Person class], @selector(run));Method m2 = class\_getClassMethod([Person class], @selector(study));// 开始交换方法实现

method\_exchangeImplementations(m1, m2);// 交换后，先打印学习，再打印跑！

[Person run];

[Person study];

###### 案例2：拦截系统方法

需求：比如iOS6 升级 iOS7 后需要版本适配，根据不同系统使用不同样式图片（拟物化和扁平化），如何通过不去手动一个个修改每个UIImage的imageNamed：方法就可以实现为该方法中加入版本判断语句？

步骤：  
1、为UIImage建一个分类（UIImage+Category）  
2、在分类中实现一个自定义方法，方法中写要在系统方法中加入的语句，比如版本判断

+ (UIImage \*)xh\_imageNamed:(NSString \*)name {

double version = [[UIDevice currentDevice].systemVersion doubleValue];

if (version >= 7.0) {

// 如果系统版本是7.0以上，使用另外一套文件名结尾是‘\_os7’的扁平化图片

name = [name stringByAppendingString:@"\_os7"];

}

return [UIImage xh\_imageNamed:name];

}

3、分类中重写UIImage的load方法，实现方法的交换（只要能让其执行一次方法交换语句，load再合适不过了）

+ (void)load {

// 获取两个类的类方法

Method m1 = class\_getClassMethod([UIImage class], @selector(imageNamed:));

Method m2 = class\_getClassMethod([UIImage class], @selector(xh\_imageNamed:));

// 开始交换方法实现

method\_exchangeImplementations(m1, m2);

}

###### 注意：自定义方法中最后一定要再调用一下系统的方法，让其有加载图片的功能，但是由于方法交换，系统的方法名已经变成了我们自定义的方法名（有点绕，就是用我们的名字能调用系统的方法，用系统的名字能调用我们的方法），这就实现了系统方法的拦截！

利用以上思路，我们还可以给 NSObject 添加分类，统计创建了多少个对象，给控制器添加分类，统计有创建了多少个控制器，特别是公司需求总变的时候，在一些原有控件或模块上添加一个功能，建议使用该方法！

#### 二、在分类中设置属性，给任何一个对象设置属性

众所周知，分类中是无法设置属性的，如果在分类的声明中写@property 只能为其生成get 和 set 方法的声明，但无法生成成员变量，就是虽然点语法能调用出来，但程序执行后会crash，有人会想到使用全局变量呢？比如这样：

int \_age;

- (int )age {

return \_age;

}

- (void)setAge:(int)age {

\_age = age;

}

但是全局变量程序整个执行过程中内存中只有一份，我们创建多个对象修改其属性值都会修改同一个变量，这样就无法保证像属性一样每个对象都拥有其自己的属性值。这时我们就需要借助runtime为分类增加属性的功能了。

需要用到的方法 <objc/runtime.h>

* set方法，将值value 跟对象object 关联起来（将值value 存储到对象object 中）  
  参数 object：给哪个对象设置属性  
  参数 key：一个属性对应一个Key，将来可以通过key取出这个存储的值，key 可以是任何类型：double、int 等，建议用char 可以节省字节  
  参数 value：给属性设置的值  
  参数policy：存储策略 （assign 、copy 、 retain就是strong）

void objc\_setAssociatedObject(id object , const void \*key ,id value ,objc\_AssociationPolicy policy)

* 利用参数key 将对象object中存储的对应值取出来

id objc\_getAssociatedObject(id object , const void \*key)

步骤：  
1、创建一个分类，比如给任何一个对象都添加一个name属性，就是NSObject添加分类（NSObject+Category）  
2、先在.h 中@property 声明出get 和 set 方法，方便点语法调用

@property(nonatomic,copy)NSString \*name;

3、在.m 中重写set 和 get 方法，内部利用runtime 给属性赋值和取值

char nameKey;

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

// 将某个值跟某个对象关联起来，将某个值存储到某个对象中

objc\_setAssociatedObject(self, &nameKey, name, OBJC\_ASSOCIATION\_COPY\_NONATOMIC);

}

- (NSString \*)name {

return objc\_getAssociatedObject(self, &nameKey);

}

#### 三、获得一个类的所有成员变量

最典型的用法就是一个对象在归档和解档的 encodeWithCoder和initWithCoder:方法中需要该对象所有的属性进行decodeObjectForKey: 和 encodeObject:，通过runtime我们声明中无论写多少个属性，都不需要再修改实现中的代码了。

需要用到的方法 <objc/runtime.h>

* 获得某个类的所有成员变量（outCount 会返回成员变量的总数）  
  参数：  
  1、哪个类  
  2、放一个接收值的地址，用来存放属性的个数  
  3、返回值：存放所有获取到的属性，通过下面两个方法可以调出名字和类型

Ivar \*class\_copyIvarList(Class cls , unsigned int \*outCount)

* 获得成员变量的名字

const char \*ivar\_getName(Ivar v)

* 获得成员变量的类型

const char \*ivar\_getTypeEndcoding(Ivar v)

###### 案例1：获取Person类中所有成员变量的名字和类型

unsigned int outCount = 0;

Ivar \*ivars = class\_copyIvarList([Person class], &outCount);

// 遍历所有成员变量for (int i = 0; i < outCount; i++) {

// 取出i位置对应的成员变量

Ivar ivar = ivars[i];

const char \*name = ivar\_getName(ivar);

const char \*type = ivar\_getTypeEncoding(ivar);

NSLog(@"成员变量名：%s 成员变量类型：%s",name,type);

}// 注意释放内存！free(ivars);

###### 案例2：利用runtime 获取所有属性来重写归档解档方法

// 设置不需要归解档的属性

- (NSArray \*)ignoredNames {

return @[@"\_aaa",@"\_bbb",@"\_ccc"];

}

// 解档方法

- (instancetype)initWithCoder:(NSCoder \*)aDecoder {

if (self = [super initWithCoder:aDecoder]) {

// 获取所有成员变量

unsigned int outCount = 0;

Ivar \*ivars = class\_copyIvarList([self class], &outCount);

for (int i = 0; i < outCount; i++) {

Ivar ivar = ivars[i];

// 将每个成员变量名转换为NSString对象类型

NSString \*key = [NSString stringWithUTF8String:ivar\_getName(ivar)];

// 忽略不需要解档的属性

if ([[self ignoredNames] containsObject:key]) {

continue;

}

// 根据变量名解档取值，无论是什么类型

id value = [aDecoder decodeObjectForKey:key];

// 取出的值再设置给属性

[self setValue:value forKey:key];

// 这两步就相当于以前的 self.age = [aDecoder decodeObjectForKey:@"\_age"];

}

free(ivars);

}

return self;

}

// 归档调用方法

- (void)encodeWithCoder:(NSCoder \*)aCoder {

// 获取所有成员变量

unsigned int outCount = 0;

Ivar \*ivars = class\_copyIvarList([self class], &outCount);

for (int i = 0; i < outCount; i++) {

Ivar ivar = ivars[i];

// 将每个成员变量名转换为NSString对象类型

NSString \*key = [NSString stringWithUTF8String:ivar\_getName(ivar)];

// 忽略不需要归档的属性

if ([[self ignoredNames] containsObject:key]) {

continue;

}

// 通过成员变量名，取出成员变量的值

id value = [self valueForKeyPath:key];

// 再将值归档

[aCoder encodeObject:value forKey:key];

// 这两步就相当于 [aCoder encodeObject:@(self.age) forKey:@"\_age"];

}

free(ivars);

}

依据上面的原理我们就可以给NSObject做一个分类，让我们不需要每次都写这么一长串代码，只要实现一小段代码就可以让一个对象具有归解档的能力。

###### 注意，下面的代码我换了一个方法名（不然会覆盖系统原来的方法！），加了一个忽略属性方法是否被实现的判断，并加上了对父类属性的归解档循环。

NSObject+Extension.h

#import <Foundation/Foundation.h>

@interface NSObject (Extension)

- (NSArray \*)ignoredNames;

- (void)encode:(NSCoder \*)aCoder;

- (void)decode:(NSCoder \*)aDecoder;

@end

NSObject+Extension.m

#import "NSObject+Extension.h"#import <objc/runtime.h>

@implementation NSObject (Extension)

- (void)decode:(NSCoder \*)aDecoder {

// 一层层父类往上查找，对父类的属性执行归解档方法

Class c = self.class;

while (c &&c != [NSObject class]) {

unsigned int outCount = 0;

Ivar \*ivars = class\_copyIvarList(c, &outCount);

for (int i = 0; i < outCount; i++) {

Ivar ivar = ivars[i];

NSString \*key = [NSString stringWithUTF8String:ivar\_getName(ivar)];

// 如果有实现该方法再去调用

if ([self respondsToSelector:@selector(ignoredNames)]) {

if ([[self ignoredNames] containsObject:key]) continue;

}

id value = [aDecoder decodeObjectForKey:key];

[self setValue:value forKey:key];

}

free(ivars);

c = [c superclass];

}

}

- (void)encode:(NSCoder \*)aCoder {

// 一层层父类往上查找，对父类的属性执行归解档方法

Class c = self.class;

while (c &&c != [NSObject class]) {

unsigned int outCount = 0;

Ivar \*ivars = class\_copyIvarList([self class], &outCount);

for (int i = 0; i < outCount; i++) {

Ivar ivar = ivars[i];

NSString \*key = [NSString stringWithUTF8String:ivar\_getName(ivar)];

// 如果有实现该方法再去调用

if ([self respondsToSelector:@selector(ignoredNames)]) {

if ([[self ignoredNames] containsObject:key]) continue;

}

id value = [self valueForKeyPath:key];

[aCoder encodeObject:value forKey:key];

}

free(ivars);

c = [c superclass];

}

}@end

上面分类使用方法：在需要归解档的对象中实现下面方法即可：

// 设置需要忽略的属性

- (NSArray \*)ignoredNames {

return @[@"bone"];

}

// 在系统方法内来调用我们的方法

- (instancetype)initWithCoder:(NSCoder \*)aDecoder {

if (self = [super init]) {

[self decode:aDecoder];

}

return self;

}

- (void)encodeWithCoder:(NSCoder \*)aCoder {

[self encode:aCoder];

}

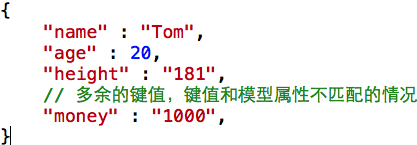
这样看来，我们每次又要写同样的代码，我们可以将归解档两个方法封装为宏，在需要的地方一句宏搞定，如果有不需要归解档的属性就实现ignoredNames 方法，具体可以看我的demo，这个也是MJExtension中那个一句宏就可以解决归解档的实现原理。

###### 案例3：利用runtime 获取所有属性来进行字典转模型

以往我们都是利用KVC进行字典转模型，但是它还是有一定的局限性，例如：模型属性和键值对对应不上会crash（虽然可以重写 setValue:forUndefinedKey:方法防止报错），模型属性是一个对象或者数组时不好处理等问题，所以无论是效率还是功能上，利用 runtime进行字典转模型都是比较好的选择。

字典转模型我们需要考虑三种特殊情况：  
1.当字典的key和模型的属性匹配不上  
2.模型中嵌套模型（模型属性是另外一个模型对象）  
3.数组中装着模型（模型的属性是一个数组，数组中是一个个模型对象）

根据上面的三种特殊情况，我们一个个处理，先是字典的key和模型的属性不对应的情况。  
不对应有两种，一种是字典的键值大于模型属性数量， 这时候我们不需要任何处理，因为runtime是先遍历模型所有属性，再去字典中根据属性名找对应值进行赋值，多余的键值对也当然不会去看了；另外一种是 模型属性数量大于字典的键值对，这时候由于属性没有对应值会被赋值为nil，就会导致crash，我们只需加一个判断即可，JSON数据和sample如 下：



JSON数据

- (void)setDict:(NSDictionary \*)dict {

Class c = self.class;

while (c &&c != [NSObject class]) {

unsigned int outCount = 0;

Ivar \*ivars = class\_copyIvarList(c, &outCount);

for (int i = 0; i < outCount; i++) {

Ivar ivar = ivars[i];

NSString \*key = [NSString stringWithUTF8String:ivar\_getName(ivar)];

// 成员变量名转为属性名（去掉下划线 \_ ）

key = [key substringFromIndex:1];

// 取出字典的值

id value = dict[key];

// 如果模型属性数量大于字典键值对数理，模型属性会被赋值为nil而报错

if (value == nil) continue;

// 将字典中的值设置到模型上

[self setValue:value forKeyPath:key];

}

free(ivars);

c = [c superclass];

}

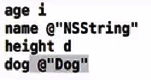
}

第二种情况是模型的属性是另外一个模型对象



JSON数据

这时候我们就需要利用runtime的ivar\_getTypeEncoding 方法获取模型对象类型，对该模型对象类型再进行字典转模型，也就是进行递归，需要注意的是我们要排除系统的对象类型，例如NSString，下面的方法中我添加了一个类方法方便递归。



打印可以看到各属性类型

#import "NSObject+JSONExtension.h"#import <objc/runtime.h>

@implementation NSObject (JSONExtension)

- (void)setDict:(NSDictionary \*)dict {

Class c = self.class;

while (c &&c != [NSObject class]) {

unsigned int outCount = 0;

Ivar \*ivars = class\_copyIvarList(c, &outCount);

for (int i = 0; i < outCount; i++) {

Ivar ivar = ivars[i];

NSString \*key = [NSString stringWithUTF8String:ivar\_getName(ivar)];

// 成员变量名转为属性名（去掉下划线 \_ ）

key = [key substringFromIndex:1];

// 取出字典的值

id value = dict[key];

// 如果模型属性数量大于字典键值对数理，模型属性会被赋值为nil而报错

if (value == nil) continue;

// 获得成员变量的类型

NSString \*type = [NSString stringWithUTF8String:ivar\_getTypeEncoding(ivar)];

// 如果属性是对象类型

NSRange range = [type rangeOfString:@"@"];

if (range.location != NSNotFound) {

// 那么截取对象的名字（比如@"Dog"，截取为Dog）

type = [type substringWithRange:NSMakeRange(2, type.length - 3)];

// 排除系统的对象类型

if (![type hasPrefix:@"NS"]) {

// 将对象名转换为对象的类型，将新的对象字典转模型（递归）

Class class = NSClassFromString(type);

value = [class objectWithDict:value];

}

}

// 将字典中的值设置到模型上

[self setValue:value forKeyPath:key];

}

free(ivars);

c = [c superclass];

}

}

+ (instancetype )objectWithDict:(NSDictionary \*)dict {

NSObject \*obj = [[self alloc]init];

[obj setDict:dict];

return obj;

}

第三种情况是模型的属性是一个数组，数组中是一个个模型对象，例如下面的数据我就可以通过books[0].name获取到C语言程序设计



JSON数据

我们既然能获取到属性类型，那就可以拦截到模型的那个数组属性，进而对数组中每个模型遍历并字典转模型，但是我们不知道数组中的模型都是什么类型，我们可以声明一个方法，该方法目的不是让其调用，而是让其实现并返回模型的类型。  
这块语言可能解释不太清楚，可以参考我的demo，直接运行即可。

NSObject+JSONExtension.h

// 返回数组中都是什么类型的模型对象

- (NSString \*)arrayObjectClass ;

NSObject+JSONExtension.m

#import "NSObject+JSONExtension.h"#import <objc/runtime.h>

@implementation NSObject (JSONExtension)

- (void)setDict:(NSDictionary \*)dict {

Class c = self.class;

while (c &&c != [NSObject class]) {

unsigned int outCount = 0;

Ivar \*ivars = class\_copyIvarList(c, &outCount);

for (int i = 0; i < outCount; i++) {

Ivar ivar = ivars[i];

NSString \*key = [NSString stringWithUTF8String:ivar\_getName(ivar)];

// 成员变量名转为属性名（去掉下划线 \_ ）

key = [key substringFromIndex:1];

// 取出字典的值

id value = dict[key];

// 如果模型属性数量大于字典键值对数理，模型属性会被赋值为nil而报错

if (value == nil) continue;

// 获得成员变量的类型

NSString \*type = [NSString stringWithUTF8String:ivar\_getTypeEncoding(ivar)];

// 如果属性是对象类型

NSRange range = [type rangeOfString:@"@"];

if (range.location != NSNotFound) {

// 那么截取对象的名字（比如@"Dog"，截取为Dog）

type = [type substringWithRange:NSMakeRange(2, type.length - 3)];

// 排除系统的对象类型

if (![type hasPrefix:@"NS"]) {

// 将对象名转换为对象的类型，将新的对象字典转模型（递归）

Class class = NSClassFromString(type);

value = [class objectWithDict:value];

}else if ([type isEqualToString:@"NSArray"]) {

// 如果是数组类型，将数组中的每个模型进行字典转模型，先创建一个临时数组存放模型

NSArray \*array = (NSArray \*)value;

NSMutableArray \*mArray = [NSMutableArray array];

// 获取到每个模型的类型

id class ;

if ([self respondsToSelector:@selector(arrayObjectClass)]) {

NSString \*classStr = [self arrayObjectClass];

class = NSClassFromString(classStr);

}

// 将数组中的所有模型进行字典转模型

for (int i = 0; i < array.count; i++) {

[mArray addObject:[class objectWithDict:value[i]]];

}

value = mArray;

}

}

// 将字典中的值设置到模型上

[self setValue:value forKeyPath:key];

}

free(ivars);

c = [c superclass];

}

}

+ (instancetype )objectWithDict:(NSDictionary \*)dict {

NSObject \*obj = [[self alloc]init];

[obj setDict:dict];

return obj;

}

@end