Runtime学习

> *我们知道OC是基于C的，也知道C是面向过程的，OC是面向对象的，如何实现从面向过程转换为面向对象呢？另外，C语言在编译完成后已经明确了函数调用顺序，我们称之为静态语言，而OC却只有运行时才能确定代码执行的顺序，相应的称之为动态语言，如何实现的动态语言呢？这就是我们本节课研究的课题。*

Runtime提供了引言中问题的解决方案。它是OC语言的一个核心机制，使OC具有了面向对象的特征，实现了把函数转换为方法的机制等等。本节结束后，你会对OC的机制有更加深刻的理解，比如我们熟知的self隐藏参数是在哪定义的等等。

什么是Runtime

Runtime是一个纯C语言的库，其中定义了非常多的结构体和函数，通过这些结构体和函数，实现了我们平时使用的OC语言的基本机制。Runtime保证了我们平时觉得习以为常的东西能够正常运作。Runtime是使OC变为动态语言的核心。

Runtime存放在usr/include/objc/目录下，usr文件系统包含所有的命令、库文件等一些正常使用时不会修改的文件。inclue中包含了一些重要的头文件，比如我们以前经常用到的`stdio.h`（标准输入输出库的头文件），objc内存放的是oc的实现的头文件。

如果想使用其中的函数或者结构体，我们同样需要引入头文件：

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

NSObject

当我们在使用OC编写代码，我们都知道所有的类都继承于NSObject（其实还有一个NSProxy），我们都知道NSObject是一个根类，那么问题来了。NSObject是怎么来的？

想探明究竟很简单，我们跳转到头文件中：

```
@interface NSObject <NSObject> {
    Class isa  OBJC_ISA_AVAILABILITY;
}

+ (void)load;

+ (void)initialize;
- (instancetype)init;

+ (instancetype)new;
+ (instancetype)allocWithZone:(struct _NSZone *)zone;
+ (instancetype)alloc;
- (void)dealloc;
....
```

头文件中，除了一些方法，最鲜明的应该是成员变量isa，满足数据结构要求的，就可以称之为对象。

对象的数据结构定义为：

```
struct objc_object {
    Class isa  OBJC_ISA_AVAILABILITY;
};
```

那么isa是什么呢，我们可以从它的类型上一探究竟：

```
typedef struct objc_class *Class;
struct objc_class {
    Class isa  OBJC_ISA_AVAILABILITY;

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

}
```

Class很明显就是一个结构体，那么这个结构体里都是一些什么呢？下面会具体介绍。

但是看到这里我们应该能明白了， NSObject就是内部含有一个isa指针，指向Class即可，来表明它的类。

Class是什么

我们平时说的类，其实就是这里的Class，实现了一个类应该有的功能。

类的功能：

1. 保存成员变量
2. 有父类，可以继承父类成员变量和方法
3. 保存所有方法
4. 保存确认的协议
5. 保存类的信息，比如一个实例对象的大小等等

Class结构体中的变量具体作用如下：

- Class isa; // 指向metaclass 元类
- Class super_class ; // 指向其父类
- const char *name ; // 类名
- long version ; // 类的版本信息，初始化默认为0，可以通过runtime函数class_setVersion和class_getVersion进行修改、读取
- long info; // 一些标识信息,如CLS_CLASS (0x1L) 表示该类为普通 class ， 其中包含对象方法和成员变量;CLS_META (0x2L) 表示该类为 metaclass;
- long instance_size ; // 该类的实例变量大小(包括从父类继承下来的实例变量);
- struct objc_ivar_list *ivars; // 存储成员变量列表
- struct objc_method_list **methodLists ; // 储存方法列表

- struct objc_cache *cache; // 指向最近使用的方法的指针，用于提升效率；
- struct objc_protocol_list *protocols; // 存储该类遵守的协议

执行的过程也就很简单了，举个例子，定义一个CNCNParent类继承于NSObject，一个CNChild类继承于CNParent。

通过使用实例方法-(Class)Class,得到对象的isa指针指向的Class结构体。

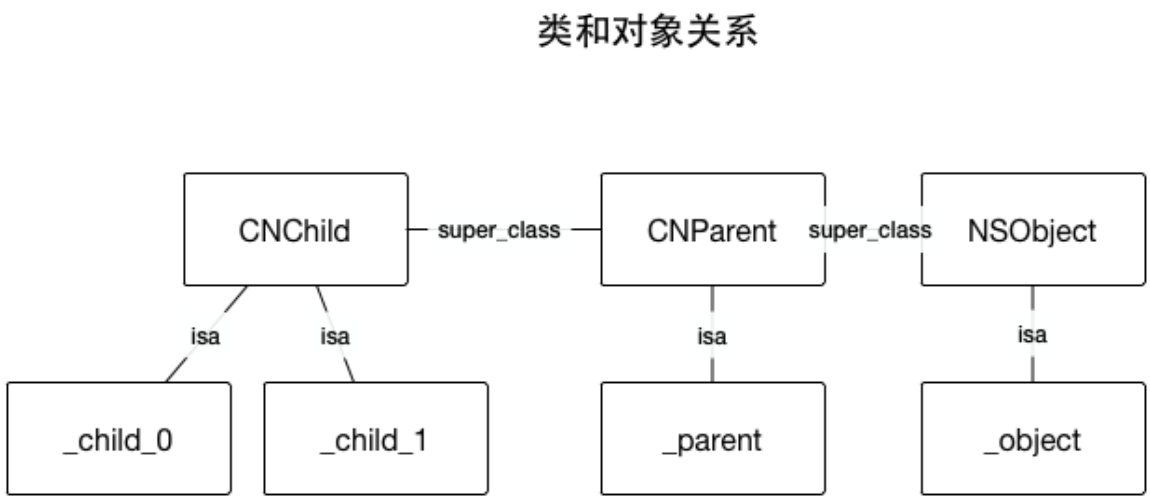
代码示例：

```
    CNParent *_parent = [[CNParent alloc] init];
    CNChild *_child = [[CNChild alloc] init];
    CNChild *_child2 = [[CNChild alloc] init];

    //使用实例方法class 获取isa指针,也可以使用函数object_getClass(_child);
    NSLog(@"CNChild struct add:%p",[_child class]);
    NSLog(@"CNChild struct add:%p",[_child2 class]);

    //使用类方法superclass 获取super_class指针
    NSLog(@"CNChild superclass struct add: %p",[CNChild superclass]);
    NSLog(@"CNParent Class struct add:%p",[_parent class]);
```

结果如下：

```
CNChild struct add:0x1000012d0
CNChild struct add:0x1000012d0
CNChild superclass struct add: 0x100001280
CNParent Class struct add:0x100001280
```

类、对象之间的关系用下图表示：



类本身也有一个isa指针，那么它指向哪里呢？请看下文。

MetaClass元类

类的isa指针也指向一个类，我们称之为元类。

从数据结构上分析，一个类的结构体（objc_class）定义满足前文中对象（objc_object）的定义，具体体现

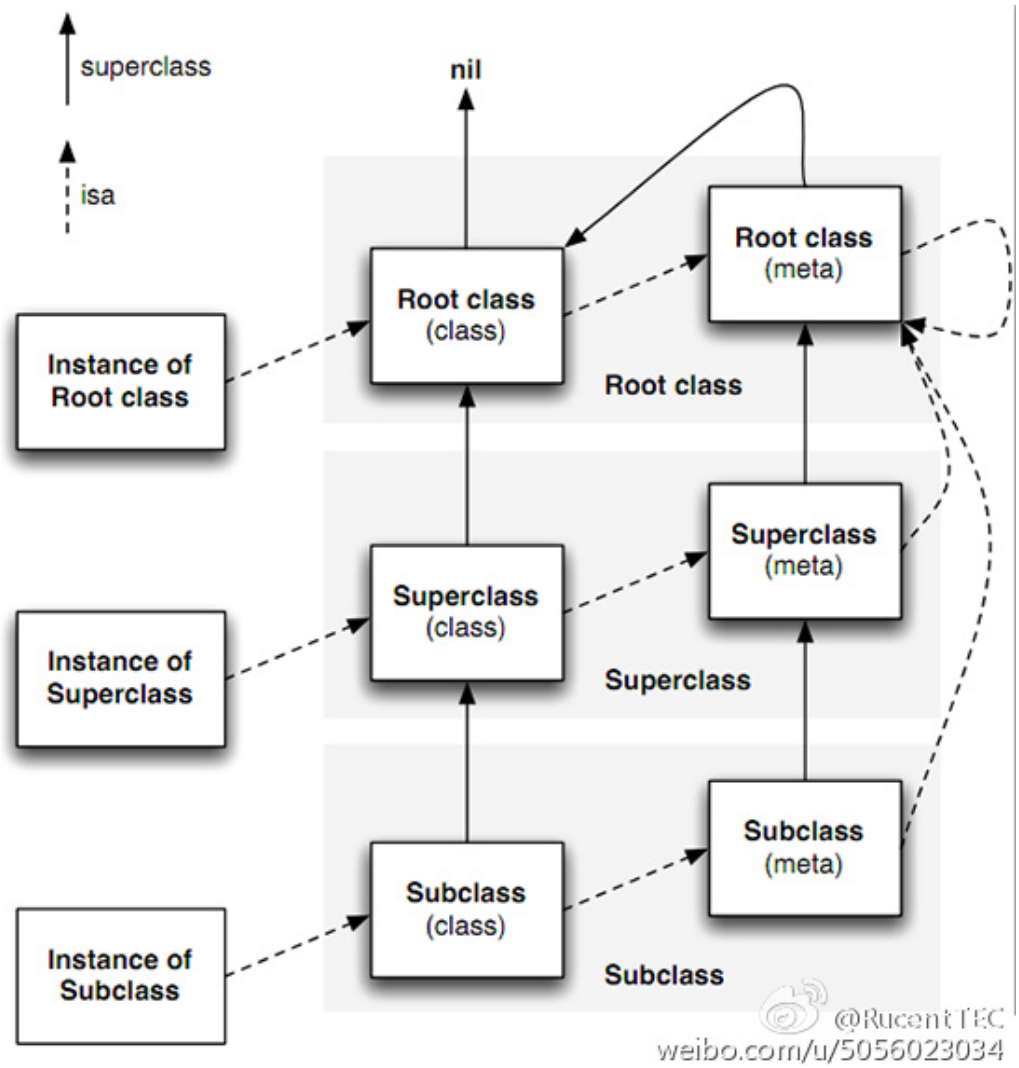
在类方法，我们在使用类方法时，如:

```
[CNChild doSomething];
```

相当于把CNChild看做了一个对象，因为只能给具有isa指针的结构体发送消息（下文消息机制中会详解）。所以我们把CNChild看做一个对象，那么对象必然要有对应的objc_class结构保存他的方法列表等等，那么这里就引出了类的类，即元类（MetaClass）。

元类的定义为：描述一个类对象的类。

也就是说类中的isa指向它的元类，元类和类是一一对应的。我们知道类方法也是可以从父类中继承的，所以元类的super_class当然就指向了对应类父类的元类。如下图:



为了机制的完整性，根类的元类的isa指向自身，super_class指向根类。

总结:

- objc_class中info用16进制描述了两种类的类型，1表示该结构体描述的是一个类，2表示元类。
- objc_class中objc_method_list，如果info是1，则保存变量和实例方法，是2则保存类方法。
- cache用于保存最近使用过的方法，逻辑很简单，调用方法时首先在cache中看一下有没有，没有再去objc_method_list找，再没有就去super_class指向的objc_class结构体的cache中找，以此类推，提高

了效率。

以上内容，让我们了解了类是如何保存，如何继承的。结构体中的成员我们都无法直接修改，如果想要修改，就一定要通过系统提供给我们的函数， 那么有哪些常用的函数呢？请看下文。

消息机制

目前为止还有一个问题没有解释，C中是没有方法的，只有函数，那么OC中的方法是怎么来的，怎么实现调用的?

方法调用

首先看方法是如何调用的，很多书中，或者很多人都会说发送消息，什么叫做发送消息呢？其实就是给调用方法的对象发送一条消息，发送这条消息的参数很简单，一个就是消息接受者，一个就是你要调用的方法名。

使用clang命令工具可以帮我们把OC的代码转换成C代码，具体命令如下：

```
clang -rewrite-objc main.m
```

在main函数中写下：

```
NSObject *ob = [NSObject alloc];
ob = [ob init];
```

使用命令，我们可以看到结果如下：

```
    NSObject *ob = ((NSObject *(*)(id, SEL))(void *)objc_msgSend)
((id)objc_getClass("NSObject"), sel_registerName("alloc"));
    ob = ((NSObject *(*)(id, SEL))(void *)objc_msgSend)((id)ob,
sel_registerName("init"));
```

这一行复合了太多函数，我们拆分一下：

```
    NSObject *(*action)(id, SEL) = (NSObject *(*)(id, SEL))(void *)objc_msgSend;
    id object = (id)objc_getClass("NSObject");
    SEL sel = sel_registerName("alloc");
    NSObject *ob2 = action(object,sel);
```

原因是：我们无法直接调用objc_msgSend函数，需要使用函数指针调用该函数，其实如果可以直接调用的话，核心部分如下：

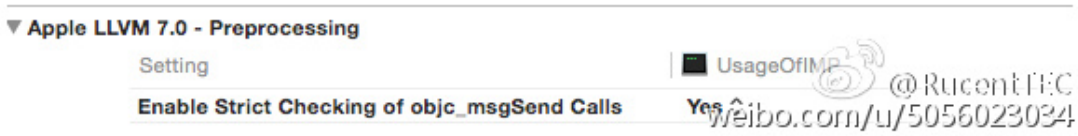
```
objc_msgSend(ob,selector);
```

我们可以看到，如前文所说，调用函数，传入对象和方法，然后就会到ob的isa指针指向的objc_class结构体中寻找selector方法。

注：为什么无法调用objc_msgSend(),原因如下

```
#if !OBJC_OLD_DISPATCH_PROTOTYPES
OBJC_EXPORT void objc_msgSend(void /* id self, SEL op, ... */ )
    __OSX_AVAILABLE_STARTING(__MAC_10_0, __IPHONE_2_0);
OBJC_EXPORT void objc_msgSendSuper(void /* struct objc_super *super, SEL op, ... */ )
    __OSX_AVAILABLE_STARTING(__MAC_10_0, __IPHONE_2_0);
#else

OBJC_EXPORT id objc_msgSend(id self, SEL op, ...)
    __OSX_AVAILABLE_STARTING(__MAC_10_0, __IPHONE_2_0);

OBJC_EXPORT id objc_msgSendSuper(struct objc_super *super, SEL op, ...)
    __OSX_AVAILABLE_STARTING(__MAC_10_0, __IPHONE_2_0);
#endif
```

上面的条件编译代码，使objc_msgSend函数变成了无返回值无参数的状态，所以我们使用objc_msgSend没什么用，怎么使条件改变呢?



修改为NO即可，不过每次修改也挺麻烦，不如前面使用的方法，而且直接使用有可能会造成问题。所以编译器提供的编译后代码也是之前的解决方案。 详情

SEL是什么

上文中我们使用:

```
SEL sel = sel_registerName("alloc");
```

创建了一个SEL变量，和我们熟知的@selector(@"alloc")是同等作用，通过字符串就可以创建出一个SEL变量，那么也就是说SEL其实关键数据只是一个字符串而已,从objc_method_description结构体的描述中，我们可以看到注释也是这么说的，SEL类型的name变量用来表示方法的名字。types用来保存参数的类型。

struct objc_method_description { SEL name; /**< The name of the method */ *char* **types;** /**< The types of the method arguments */ };

那么方法是如何与函数互相绑定的呢?

动态地为一个类添加方法

```
childClass = [CNChild class];
class_addMethod(childClass, @selector(methodName), (IMP)impFunction,nil);
```

使用函数class_addMethod() 可以修改结构体中的methodLists:

```
struct objc_method_list **methodLists                    OBJC2_UNAVAILABLE;
```

这样我们就为CNChild添加了一个名为report的实例方法，方法的实现是名为ReportFunction的IMP变量，最后的nil是参数类型。

动态添加类方法只需要使用元类作为第一个参数。

通过上面的函数methodName就和impFunction进行了绑定，当调用methodName方法，就会走到IMP impFunction指向的函数中。

那么IMP是什么呢？

IMP

IMP就implementation的缩写，IMP的定义如下，其实就是一个函数类型：

```
#if !OBJC_OLD_DISPATCH_PROTOTYPES
typedef void (*IMP)(void /* id, SEL, ... */ );
#else
typedef id (*IMP)(id, SEL, ...);
#endif
```

可以看到IMP中默认最少有两个参数，其实就是以前我们常用的self和代表方法名的_cmd。

---综上：---

总结一下消息机制，在运行期，当遇到方法的调用代码，首先会找到对象的isa指向的objc_class中的方法列表，找到以后会调用方法名绑定的对应的函数，函数返回，最后objc_msgSend函数返回。

重要结构体

1.协议列表：

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

2.成员变量：

```
typedef struct objc_ivar *Ivar;

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

3.成员变量列表

```
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;
}                                                           OBJC2_UNAVAILABLE;
```

4.方法

```
typedef struct objc_method *Method;

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

5.方法列表

```
struct objc_method_list {
    struct objc_method_list *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;
}                                                           OBJC2_UNAVAILABLE;
```

6.方法缓存

```
typedef struct objc_cache *Cache                            OBJC2_UNAVAILABLE;

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

重要函数

下面的函数的使用可以在需要是查阅

1. 对象拷贝：id object_copy(id obj, size_t size)

```
CNChild *obj = [CNChild new];
NSLog(@"%p", &obj);

id objTest = object_copy(obj,sizeof(obj));
NSLog(@"%p", &objTest);
```

2. 对象释放：id object_dispose(id obj)

```
CNChild *obj = [CNChild snew];
object_dispose(obj);
```

3. 更改对象的类/获取对象的类：

   Class object_setClass(id obj, Class cls) & Class object_getClass(id obj)

```
CNChild *obj = [CNChild new];
[obj fun1];

Class aClass =object_setClass(obj, [CNParent class]);

NSLog(@"aClass:%@",NSStringFromClass(aClass));
NSLog(@"obj class:%@",NSStringFromClass([objclass]));
[obj fun2];
```

4. 获取对象的类名： const char *object_getClassName(id obj)

```
CNParent *obj = [CNParent new];
NSString *className =
[NSStringstringWithCString:object_getClassName(obj)encoding:NSUTF8StringEncoding];
NSLog(@"className:%@", className);
```

5. 获取一个类的所有方法：

   Method *class_copyMethodList(Class cls, unsigned int* outCount) & SEL method_getName(Method m) & const char *sel_getName(SEL sel)

```
u_int count;
Method* methods= class_copyMethodList([UIViewController class], &count);
for (int i = 0; i < count ; i++)
{
    SEL name = method_getName(methods[i]);
    NSString *strName = [NSString
stringWithCString:sel_getName(name)encoding:NSUTF8StringEncoding];
    NSLog(@"%@",strName);
}
free(methods);
```

6. 获取/替换方法对应的实现：

   Method class_getInstanceMethod(Class cls, SEL name) & void method_exchangeImplementations(Method m1, Method m2) & IMP

method_setImplementation(Method m, IMP imp)

7. 替换方法对应的函数:

   IMP class_replaceMethod(Class cls, SEL name, IMP imp, const char *types)

更多函数请参考文件中的注释内容

常用方法总结

新的BaseModel

```
-(void)setItemfree:(NSArray*)uselessKeys{
    unsigned int numIvars = 0;

    NSString *key=nil;

    Ivar * ivars = class_copyIvarList([self class], &numIvars);

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

        Ivar thisIvar = ivars[i];
        const char *type = ivar_getTypeEncoding(thisIvar);
        //d --> double i--> int @"NSString" -->nsstring
        NSString *stringType =  [NSString stringWithCString:type
encoding:NSUTF8StringEncoding];
        //属性都带有下划线 如 _payType
        key = [NSString stringWithUTF8String:ivar_getName(thisIvar)];


        //移除不需要的变量
        if ([uselessKeys containsObject:key]) {
            continue;
        }


        NSString* realKey = [key substringFromIndex:1];

//        NSLog(@" key :%@",stringType);

        //string 型
        if ([stringType isEqualToString:@"@\"NSString\""]) {
            [self setValue:@"" forKey:realKey];
        }

        //double 型
        if ([stringType isEqualToString:@"d"]) {
            double string = 0.0;

            [self setValue:[NSNumber numberWithDouble:string] forKey:realKey];


        }
        //int 型
        if ([stringType isEqualToString:@"i"]) {
            int string = 0;

            [self setValue:[NSNumber numberWithInt:string] forKey:realKey];
```

```
        }

        //数组型

        if ([stringType isEqualToString:@"@\"NSArray\""]) {
            [self setValue:@[] forKey:realKey];
        }

    }

    free(ivars);

}

-(void)setItemFromDic:(NSDictionary*)dic :(NSArray*)uselessKeys{
    unsigned int numIvars = 0;

    NSString *key=nil;

    Ivar * ivars = class_copyIvarList([self class], &numIvars);

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

        Ivar thisIvar = ivars[i];
        const char *type = ivar_getTypeEncoding(thisIvar);
        //d --> double i--> int @"NSString" -->nsstring
        NSString *stringType =  [NSString stringWithCString:type
encoding:NSUTF8StringEncoding];
        //属性都带有下划线 如 _payType
        key = [NSString stringWithUTF8String:ivar_getName(thisIvar)];


        //移除不需要的变量
        if ([uselessKeys containsObject:key]) {
            continue;
        }



        NSString* realKey = [key substringFromIndex:1];

       // NSLog(@" key :%@",realKey);

        //string 型
        if ([stringType isEqualToString:@"@\"NSString\""]) {



            //如果类型为_id 对应的key 改为id （因为objc 不允许有id名的变量）
            if ([realKey isEqualToString:@"_id"]) {
                NSString* string = [[dic objectForKey:@"id"] copy];
                [self setValue:string forKey:realKey];
                continue;
            }

            NSString* string = [[dic objectForKey:realKey] copy];
            [self setValue:string forKey:realKey];
        }
```

```
        //double 型
        if ([stringType isEqualToString:@"d"]) {

            if([[dic objectForKey:realKey] isKindOfClass:[NSNull class]]){
                continue;
            }

            double string = [[dic objectForKey:realKey] doubleValue];

            [self setValue:[NSNumber numberWithDouble:string] forKey:realKey];


        }
        //int 型
        if ([stringType isEqualToString:@"i"]) {
            int string = [[dic objectForKey:realKey] intValue];

            [self setValue:[NSNumber numberWithInt:string] forKey:realKey];

        }

        //数组型

        NSLog(@"=======%@",stringType);
        if ([stringType isEqualToString:@"@\"NSArray\""]) {

            NSArray* arr = [NSArray arrayWithArray:[dic objectForKey:realKey]];
            NSLog(@"%@",arr);
            [self setValue:arr forKey:realKey];
        }


    }

    free(ivars);



}
```

动态创建类

```
 Class newClass = objc_allocateClassPair([Person class], "Student", 0);

class_addMethod(newClass, @selector(report), (IMP)ReportFunction,nil);
objc_registerClassPair(newClass);


id  s = [[newClass alloc] init];
[s performSelector:@selector(report)];
```

自动序列化

```
- (void)encodeWithCoder:(NSCoder *)encoder {
    Class cls = [selfclass];
    while (cls != [NSObjectclass]) {
        unsigned int numberOfIvars =0;
        Ivar* ivars = class_copyIvarList(cls, &numberOfIvars);
        for(const Ivar* p = ivars; p < ivars+numberOfIvars; p++){
            Ivar const ivar = *p;
            const char *type =ivar_getTypeEncoding(ivar);
            NSString *key = [NSStringstringWithUTF8String:ivar_getName(ivar)];
            id value = [selfvalueForKey:key];
            if (value) {
                switch (type[0]) {
                    case _C_STRUCT_B: {
                        NSUInteger ivarSize =0;
                        NSUInteger ivarAlignment =0;
                        NSGetSizeAndAlignment(type, &ivarSize, &ivarAlignment);
                        NSData *data = [NSDatadataWithBytes:(constchar *)self +
ivar_getOffset(ivar)
                                                     length:ivarSize];
                        [encoder encodeObject:dataforKey:key];
                    }
                        break;
                    default:
                        [encoder encodeObject:value
                                       forKey:key];
                        break;
                }
            }
        }
        free(ivars);
        cls = class_getSuperclass(cls);
    }
}

- (id)initWithCoder:(NSCoder *)decoder {
    self = [self init];

    if (self) {
        Class cls = [selfclass];
        while (cls != [NSObjectclass]) {
            unsigned int numberOfIvars =0;
            Ivar* ivars = class_copyIvarList(cls, &numberOfIvars);

            for(constIvar* p = ivars; p < ivars+numberOfIvars; p++){
                Ivar const ivar = *p;
                const char *type =ivar_getTypeEncoding(ivar);
                NSString *key = [NSStringstringWithUTF8String:ivar_getName(ivar)];
                id value = [decoder decodeObjectForKey:key];
                if (value) {
                    switch (type[0]) {
                        case _C_STRUCT_B: {
                            NSUInteger ivarSize =0;
                            NSUInteger ivarAlignment =0;
                            NSGetSizeAndAlignment(type, &ivarSize, &ivarAlignment);
                            NSData *data = [decoderdecodeObjectForKey:key];
                            char *sourceIvarLocation = (char*)self+ivar_getOffset(ivar);
                            [data getBytes:sourceIvarLocationlength:ivarSize];
                        }
```

```
                            break;
                        default:
                            [self setValue:[decoder decodeObjectForKey:key]
                                    forKey:key];
                            break;
                    }
                }
            }
            free(ivars);
            cls = class_getSuperclass(cls);
        }
    }

    return self;
}
```