YYModel源码（一）

YYModel整体上可以看作只有一行代码，搞懂了这一行代码，就可以说对YYModel有了初步的理解

代码块

```
1   _YYModelMeta *modelMeta = [_YYModelMeta metaWithClass:self.class];
```

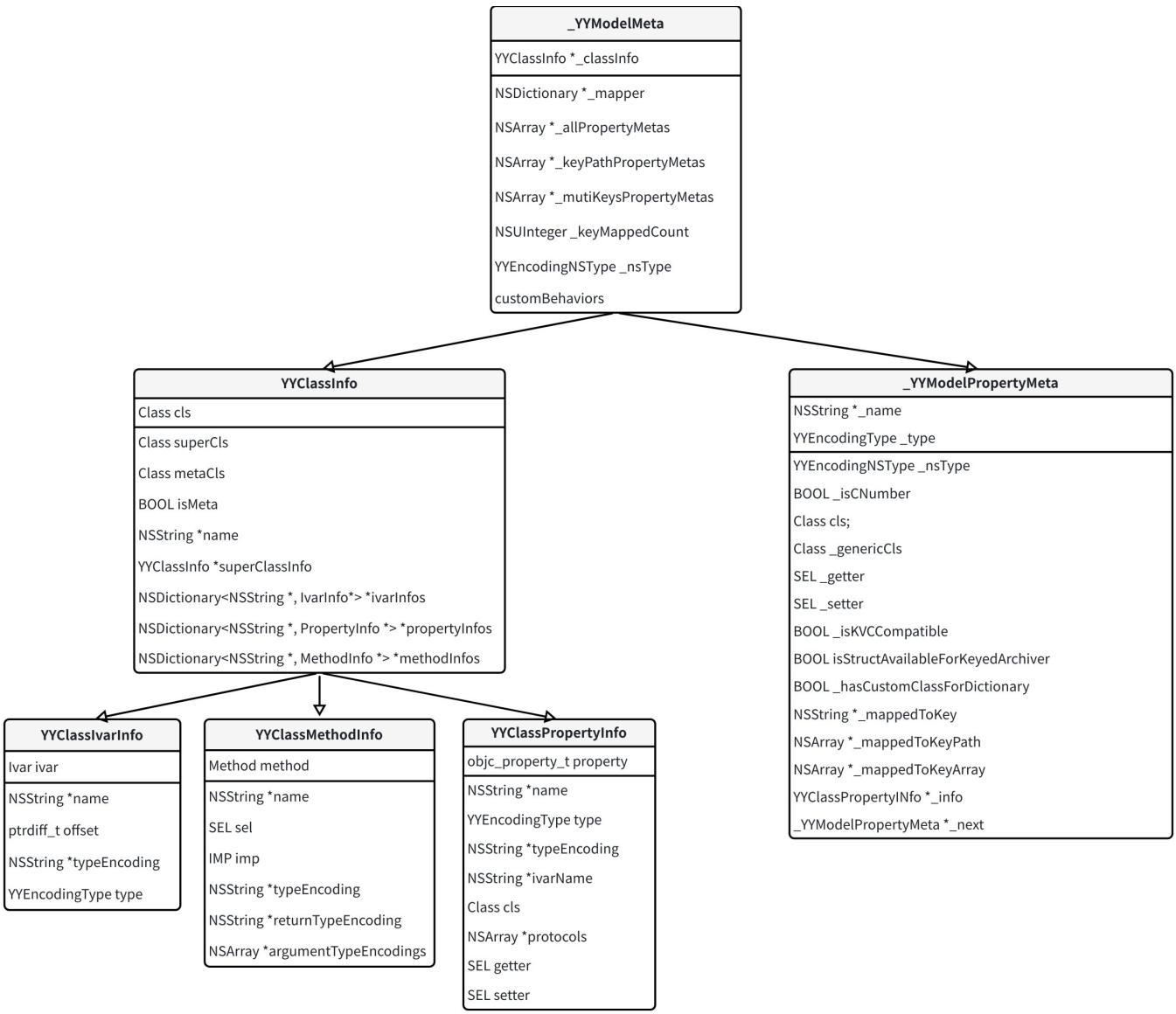
1. 优化

YYModel有非常多的优化，可以说是把性能做到了极致

1. 缓存_ModelMeta和ClassInfo，缓存ClassInfo的时候会一直缓存到Root Class
2. 大量的C结构体
3. Core Foundation函数调用
4. 高效的结构体设计，缓存SEL等
5. 直接调用msg_send给属性赋值而不是通过KVC
6. 使用__unsafe_unretained，跳过retain/release
7. mapToKeyCount和NSDictionary中key数量比较优化，跳过了无效的赋值步骤
8. 大量的dispatch_once，不可变字典全部创建后缓存

2. 架构

YYModel和MJExtension不同的是，YYModel使用协议让业务自定义行为，而MJExtension使用的是硬编码到NSObject的类方法



- 可以看到私有的结构体都是存储业务行为，缓存解析需要的数据，还有元数据的整合
- 左侧的结构体都是对runtime结构体解析后的封装，除了YYEncoding和存储了各自的runtime结构（ivar, property, method）外都和runtime结构体保持一致

2.1 _YYModelMeta

公开的类都和其对应的runtime结构体保持高度一致，所以只需要着重分析私有类即可

- YYEncodingNSType：标记属性是不是基础类型，如果是基础类型，会进行特殊处理，比如
  - NSString/NSMutableString：根据传入的参数是NSData、NSURL、NSAttributedString，就会对传入的value做一定的处理，然后再赋值，比如如果value是NSNumber，就会调用value.string，如果是NSData就会使用encoding
  - NSArray/NSMutableArray：根据参数进行转换赋值，如果类型是其它对象，value是NSDictionary，会调用modelSetWithDictionary(前提是这个属性被标记了genericCls)
- _mapper：自定义属性名和属性名到PropertyMeta的映射

- _allPropertyMetas：属性封装后的元数据
- _keyPathPropertyMetas：业务的keyPath的映射，只存储了keyPath而不是key
- _mutiKeyPropertyMetas：业务的多个key映射到属性元数据
- 其它bool值字段存储业务有没有实现协议方法

_YYModelMeta的初始化流程：

1. 初始化YYClassInfo
2. 根据协议初始化对应的属性
3. 映射属性名-属性

2.2 _YYModelPropertyMeta

- YYEncodingNSType：和ModelMeta中的nstype一样，是根据[Class isKindOfClass:]解析出来的类型，用来标识属性是不是需要特殊处理的Foundation类型
- YYEncodingType：就是属性解析attributes后的类型，和PropertyInfo基本保持一致，追加了一些CG类型的判断，用来判断属性能不能keyArchiver
- _isKVCCompatible：根据EncodingType判断类型能不能通过KVC赋值

3. 流程

3.1 _yy_dictionaryWithJSON

代码块

```
+ (NSDictionary *)_yy_dictionaryWithJSON:(id)json {
    if (!json || json == (id)kCFNull) return nil;
    NSDictionary *dic = nil;
    NSData *jsonData = nil;
    if ([json isKindOfClass:[NSDictionary class]]) {
        dic = json;
    } else if ([json isKindOfClass:[NSString class]]) {
        jsonData = [(NSString *)json dataUsingEncoding : NSUTF8StringEncoding];
    } else if ([json isKindOfClass:[NSData class]]) {
        jsonData = json;
    }
    if (jsonData) {
        dic = [NSJSONSerialization JSONObjectWithData:jsonData
options:kNilOptions error:NULL];
        if (![dic isKindOfClass:[NSDictionary class]]) dic = nil;
    }
    return dic;
}
```

- 转换的时候，json必须是NSDictionary/NSString/NSData，不然就会返回nil

3.2 Json/Dictionary转换Model

代码块

```
+ (instancetype)modelWithJSON:(id)json;
- (BOOL)modelSetWithJSON:(id)json;
+ (instancetype)modelWithDictionary:(NSDictionary *)dictionary;
- (BOOL)modelSetWithDictionary:(NSDictionary *)dic;
```

- 这四个方法都是同一种方法，其中最重要的是第四个方法，其它三个都是做了一些处理后直接调用第四个方法

3.3 _YYModelMeta

代码块

```
+ (instancetype)modelWithDictionary:(NSDictionary *)dictionary {
    if (!dictionary || dictionary == (id)kCFNull) return nil;
    if (![dictionary isKindOfClass:[NSDictionary class]]) return nil;

    Class cls = [self class];
    _YYModelMeta *modelMeta = [_YYModelMeta metaWithClass:cls];
    if (modelMeta->_hasCustomClassFromDictionary) {
        cls = [cls modelCustomClassForDictionary:dictionary] ?: cls;
    }

    NSObject *one = [cls new];
    if ([one modelSetWithDictionary:dictionary]) return one;
    return nil;
}
```

- 拿到/创建缓存的ModelMeta，如果业务实现了自定义的Class，就使用业务的Class
- 创建实例，modelSetWithDictionary给实例赋值

关于modelCustomClassForDictionary，有例子更容易理解，如果要根据服务端返回的值创建不同的对象，也就是一个工厂模式，可以这样用

代码块

```
@interface Message : NSObject
@property (nonatomic, copy) NSString *type;
+ (Class)modelCustomClassForDictionary:(NSDictionary *)dictionary;
@end
```

```
5
6   @implementation Message
7   + (Class)modelCustomClassForDictionary:(NSDictionary *)dictionary {
8       NSString *type = dictionary[@"type"];
9       if ([type isEqualToString:@"text"]) return [TextMessage class];
10      if ([type isEqualToString:@"image"]) return [ImageMessage class];
11      if ([type isEqualToString:@"video"]) return [VideoMessage class];
12      return self;
13  }
14  @end
```

_YYModelMeta的创建

代码块

```
1   + (instancetype)metaWithClass:(Class)cls {
2       if (!cls) return nil;
3       static CFMutableDictionaryRef cache;
4       static dispatch_once_t onceToken;
5       static dispatch_semaphore_t lock;
6       dispatch_once(&onceToken, ^{
7           cache = CFDictionaryCreateMutable(CFAllocatorGetDefault(), 0,
    &kCFTypeDictionaryKeyCallBacks, &kCFTypeDictionaryValueCallBacks);
8           lock = dispatch_semaphore_create(1);
9       });
10      dispatch_semaphore_wait(lock, DISPATCH_TIME_FOREVER);
11      _YYModelMeta *meta = CFDictionaryGetValue(cache, (__bridge const void *)
    (cls));
12      dispatch_semaphore_signal(lock);
13      if (!meta || meta->_classInfo.needUpdate) {
14          meta = [[_YYModelMeta alloc] initWithClass:cls];
15          if (meta) {
16              dispatch_semaphore_wait(lock, DISPATCH_TIME_FOREVER);
17              CFDictionarySetValue(cache, (__bridge const void *)(cls), (__bridge
     const void *)(meta));
18              dispatch_semaphore_signal(lock);
19          }
20      }
21      return meta;
22  }
```

- 全局缓存：cache是一个静态变量，使用dispatch_once让它只被创建一次，全局都能使用，也就是说只要一个Model被创建，下一次就直接从缓存中取就可以了

    - 但是这里留个疑问，needUpdate是一个局部变量里的私有Ivar，YYModel并没有任何地方能够获取到这个needUpdate
    - 如果业务先转换了model，再用反射对这个model进行了修改，YYModel将不能处理这样的情况
- 这个初始化方法也比较简单，初始化_YYModelMeta然后把它放进缓存里，返回结果
- dispatch_semaphore_t保证创建时候的线程安全

_YYModelMeta的初始化方法太长了，拆分开来分析会比较好

代码块

```
    NSSet *blacklist = nil;
    if ([cls respondsToSelector:@selector(modelPropertyBlacklist)]) {
        NSArray *properties = [(id<YYModel>)cls modelPropertyBlacklist];
        if (properties) {
            blacklist = [NSSet setWithArray:properties];
        }
    }

    // Get white list
    NSSet *whitelist = nil;
    if ([cls respondsToSelector:@selector(modelPropertyWhitelist)]) {
        NSArray *properties = [(id<YYModel>)cls modelPropertyWhitelist];
        if (properties) {
            whitelist = [NSSet setWithArray:properties];
        }
    }
```

- blacklist：业务通过协议实现，哪些属性不参与转换
- whitelist：只有在白名单里的属性才参与转换

代码块

```
    // Get container property's generic class
    NSDictionary *genericMapper = nil;
    if ([cls respondsToSelector:@selector(modelContainerPropertyGenericClass)]) {
        genericMapper = [(id<YYModel>)cls modelContainerPropertyGenericClass];
        if (genericMapper) {
            NSMutableDictionary *tmp = [NSMutableDictionary new];
            [genericMapper enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL *stop) {
                if (![key isKindOfClass:[NSString class]]) return;
                Class meta = object_getClass(obj);
                if (!meta) return;
```

```
                if (class_isMetaClass(meta)) {
                    tmp[key] = obj;
                } else if ([obj isKindOfClass:[NSString class]]) {
                    Class cls = NSClassFromString(obj);
                    if (cls) {
                        tmp[key] = cls;
                    }
                }
            }];
            genericMapper = tmp;
        }
    }
```

- genericMapper：集合类型自定义Class的映射，业务实现，这里还兼容了NSString的情况（如果是NSString，尝试用NSClassFromString转换成Class）

YYClassPropertyInfo

代码块

```
    // Create all property metas.
    NSMutableDictionary *allPropertyMetas = [NSMutableDictionary new];
    YYClassInfo *curClassInfo = classInfo;
    while (curClassInfo && curClassInfo.superCls != nil) { // recursive parse super class, but ignore root class (NSObject/NSProxy)
        for (YYClassPropertyInfo *propertyInfo in curClassInfo.propertyInfos.allValues) {
            if (!propertyInfo.name) continue;
            if (blacklist && [blacklist containsObject:propertyInfo.name]) continue;
            if (whitelist && ![whitelist containsObject:propertyInfo.name]) continue;
            _YYModelPropertyMeta *meta = [_YYModelPropertyMeta metaWithClassInfo:classInfo

propertyInfo:propertyInfo

generic:genericMapper[propertyInfo.name]];
            if (!meta || !meta->_name) continue;
            if (!meta->_getter || !meta->_setter) continue;
            if (allPropertyMetas[meta->_name]) continue;
            allPropertyMetas[meta->_name] = meta;
        }
        curClassInfo = curClassInfo.superClassInfo;
    }
    if (allPropertyMetas.count) _allPropertyMetas = allPropertyMetas.allValues.copy;
```

- 到这里实际上漏了一行代码，modelInfo的创建漏了，下面补上
- 这行代码也非常简单
  - 根据之前的blacklist和whitelist，给符合条件的属性创建Info，然后和属性名称关联
  - 必须保证属性同时实现了getter/setter，不然不能安全读写
  - 防重入，防止属性名称重复
  - 一直循环到没有父类的父类为nil的时候再结束，除了NSObject和NSProxy这两个基类，收集继承链上的有效属性

YYClassInfo的初始化

之前漏了一行代码分析，在_YYModelMeta初始化的第一行，会初始化这个ClassInfo，之后都会用到

代码块

```
+ (instancetype)classInfoWithClass:(Class)cls {
    if (!cls) return nil;
    static CFMutableDictionaryRef classCache;
    static CFMutableDictionaryRef metaCache;
    static dispatch_once_t onceToken;
    static dispatch_semaphore_t lock;
    dispatch_once(&onceToken, ^{
        classCache = CFDictionaryCreateMutable(CFAllocatorGetDefault(), 0, &kCFTypeDictionaryKeyCallBacks, &kCFTypeDictionaryValueCallBacks);
        metaCache = CFDictionaryCreateMutable(CFAllocatorGetDefault(), 0, &kCFTypeDictionaryKeyCallBacks, &kCFTypeDictionaryValueCallBacks);
        lock = dispatch_semaphore_create(1);
    });
    dispatch_semaphore_wait(lock, DISPATCH_TIME_FOREVER);
    YYClassInfo *info = CFDictionaryGetValue(class_isMetaClass(cls) ? metaCache : classCache, (__bridge const void *)(cls));
    if (info && info->_needUpdate) {
        [info _update];
    }
    dispatch_semaphore_signal(lock);
    if (!info) {
        info = [[YYClassInfo alloc] initWithClass:cls];
        if (info) {
            dispatch_semaphore_wait(lock, DISPATCH_TIME_FOREVER);
            CFDictionarySetValue(info.isMeta ? metaCache : classCache, (__bridge const void *)(cls), (__bridge const void *)(info));
            dispatch_semaphore_signal(lock);
        }
    }
    return info;
```

```
27  }
```

- 区分Class和MetaClass的存储和创建，和之前的ModelMeta一样，都是使用缓存，只创建一次
- 唯一的区别是，如果直接使用ClassInfo，这个needUpdate是有效的

ModelInfo的初始化

代码块

```
1   - (instancetype)initWithClass:(Class)cls {
2       if (!cls) return nil;
3       self = [super init];
4       _cls = cls;
5       _superCls = class_getSuperclass(cls);
6       _isMeta = class_isMetaClass(cls);
7       if (!_isMeta) {
8           _metaCls = objc_getMetaClass(class_getName(cls));
9       }
10      _name = NSStringFromClass(cls);
11      [self _update];
12
13      _superClassInfo = [self.class classInfoWithClass:_superCls];
14      return self;
15  }
```

- 初始化实例变量，包括Class，superClass，是不是metaClass，还有就是name
- modelInfo递归创建了superClassInfo的全部信息，也就是说，使用ModelInfo会一次性缓存所有继承链上的Class
- 这里的缓存包括NSObject和NSProxy，因为递归结束的条件是cls != nil 而不是 cls->_superClassInfo != nil

_update方法

代码块

```
1   - (void)_update {
2       _ivarInfos = nil;
3       _methodInfos = nil;
4       _propertyInfos = nil;
5
6       Class cls = self.cls;
7       unsigned int methodCount = 0;
8       Method *methods = class_copyMethodList(cls, &methodCount);
9       if (methods) {
10          NSMutableDictionary *methodInfos = [NSMutableDictionary new];
```

```
        _methodInfos = methodInfos;
        for (unsigned int i = 0; i < methodCount; i++) {
            YYClassMethodInfo *info = [[YYClassMethodInfo alloc]
initWithMethod:methods[i]];
            if (info.name) methodInfos[info.name] = info;
        }
        free(methods);
    }
    unsigned int propertyCount = 0;
    objc_property_t *properties = class_copyPropertyList(cls, &propertyCount);
    if (properties) {
        NSMutableDictionary *propertyInfos = [NSMutableDictionary new];
        _propertyInfos = propertyInfos;
        for (unsigned int i = 0; i < propertyCount; i++) {
            YYClassPropertyInfo *info = [[YYClassPropertyInfo alloc]
initWithProperty:properties[i]];
            if (info.name) propertyInfos[info.name] = info;
        }
        free(properties);
    }

    unsigned int ivarCount = 0;
    Ivar *ivars = class_copyIvarList(cls, &ivarCount);
    if (ivars) {
        NSMutableDictionary *ivarInfos = [NSMutableDictionary new];
        _ivarInfos = ivarInfos;
        for (unsigned int i = 0; i < ivarCount; i++) {
            YYClassIvarInfo *info = [[YYClassIvarInfo alloc]
initWithIvar:ivars[i]];
            if (info.name) ivarInfos[info.name] = info;
        }
        free(ivars);
    }

    if (!_ivarInfos) _ivarInfos = @{};
    if (!_methodInfos) _methodInfos = @{};
    if (!_propertyInfos) _propertyInfos = @{};

    _needUpdate = NO;
}
```

- 通过runtime的方法copyMethodList，copyPropertyList，copyIvarList初始化ModelInfo的信息
  - 这里要注意的就是copyIvarList这些方法不会返回父类的相关信息，会同时访问类对象的rw_t、ro_t
  - 还有就是在使用完之后，需要手动释放ivars对象

- 这里ivar和method没什么好分析的，这两个Class都是存储runtime中对应结构体的信息，和runtime中的结构体只有略微的区别
- 主要分析一下属性的创建，属性实际上就是name和attributes，但是attributes中如何解析ivar，如何解析getter/setter比较有意思

ClassPropertyInfo

代码块

```
- (instancetype)initWithProperty:(objc_property_t)property {
    // 基本的初始化
    objc_property_attribute_t *attrs = property_copyAttributeList(property, &attrCount);
    for (unsigned int i = 0; i < attrCount; i++) {
        switch (attrs[i].name[0]) {
            case 'T': { // Type encoding
                if (attrs[i].value) {
                    _typeEncoding = [NSString stringWithUTF8String:attrs[i].value];
                    type = YYEncodingGetType(attrs[i].value);

                    if ((type & YYEncodingTypeMask) == YYEncodingTypeObject && _typeEncoding.length) {
                        NSScanner *scanner = [NSScanner scannerWithString:_typeEncoding];
                        if (![scanner scanString:@"@\"" intoString:NULL]) continue;

                        NSString *clsName = nil;
                        if ([scanner scanUpToCharactersFromSet: [NSCharacterSet characterSetWithCharactersInString:@"\"<"] intoString:&clsName]) {
                            if (clsName.length) _cls = objc_getClass(clsName.UTF8String);
                        }

                        NSMutableArray *protocols = nil;
                        while ([scanner scanString:@"<" intoString:NULL]) {
                            NSString* protocol = nil;
                            if ([scanner scanUpToString:@">" intoString: &protocol]) {
                                if (protocol.length) {
                                    if (!protocols) protocols = [NSMutableArray new];
                                    [protocols addObject:protocol];
                                }
                            }
                            [scanner scanString:@">" intoString:NULL];
```

```
                        }
                        _protocols = protocols;
                    }
                }
            } break;
            case 'V': { // Instance variable
                if (attrs[i].value) {
                    _ivarName = [NSString stringWithUTF8String:attrs[i].value];
                }
            } break;
            case 'R': {
                type |= YYEncodingTypePropertyReadonly;
            } break;
            case 'C': {
                type |= YYEncodingTypePropertyCopy;
            } break;
            case '&': {
                type |= YYEncodingTypePropertyRetain;
            } break;
            case 'N': {
                type |= YYEncodingTypePropertyNonatomic;
            } break;
            case 'D': {
                type |= YYEncodingTypePropertyDynamic;
            } break;
            case 'W': {
                type |= YYEncodingTypePropertyWeak;
            } break;
            case 'G': {
                type |= YYEncodingTypePropertyCustomGetter;
                if (attrs[i].value) {
                    _getter = NSSelectorFromString([NSString
stringWithUTF8String:attrs[i].value]);
                }
            } break;
            case 'S': {
                type |= YYEncodingTypePropertyCustomSetter;
                if (attrs[i].value) {
                    _setter = NSSelectorFromString([NSString
stringWithUTF8String:attrs[i].value]);
                }
            } // break; commented for code coverage in next line
            default: break;
        }
    }
}
```

- objc_property_attribute_t：只有两个字段，分别是name和value，通过解析name和value来解析属性的修饰符
- 使用OR的方式通过name来给这个属性的NS_OPTIONS赋值，表示这个属性有什么修饰符，比如是不是原子的，是copy还是strong，是rw还是ro等
- 这里没必要硬记解析方式，如果开发中用到的话可以作为参考

YYModelInfo到这里就分析完了，实际上就是存储了当前类的信息，继承链，属性，实例变量，方法，是不是MetaClass

YYModelPropertyMeta的初始化也比较长，分成几段分析会比较好

代码块

```
    // support pseudo generic class with protocol name
    if (!generic && propertyInfo.protocols) {
        for (NSString *protocol in propertyInfo.protocols) {
            Class cls = objc_getClass(protocol.UTF8String);
            if (cls) {
                generic = cls;
                break;
            }
        }
    }
```

- 如果没有指定集合类型的具体类型，尝试在解析到的协议里找到一个能转为Class的协议，如果可以就给generic赋值
- 关于genericCls的作用，更加详细的分析见源码分析（二）

代码块

```
    _YYModelPropertyMeta *meta = [self new];
    meta->_name = propertyInfo.name;
    meta->_type = propertyInfo.type;
    meta->_info = propertyInfo;
    meta->_genericCls = generic;

    if ((meta->_type & YYEncodingTypeMask) == YYEncodingTypeObject) {
        meta->_nsType = YYClassGetNSType(propertyInfo.cls);
    } else {
        meta->_isCNumber = YYEncodingTypeIsCNumber(meta->_type);
    }
```

- 在这里判断如果meta->_type是Object类型，就尝试给nsType赋值，nsType主要是Foundation的基础类型，用来优化后续的赋值操作

- 在赋值的时候一般都是NSNumber，在需要给property赋值的时候会对CNumber做特殊处理

代码块

```
    if ((meta->_type & YYEncodingTypeMask) == YYEncodingTypeStruct) {
        /*
         It seems that NSKeyedUnarchiver cannot decode NSValue except these structs:
         */
        static NSSet *types = nil;
        static dispatch_once_t onceToken;
        dispatch_once(&onceToken, ^{
            NSMutableSet *set = [NSMutableSet new];
            // 32 bit
            [set addObject:@"{CGSize=ff}"];
            [set addObject:@"{CGPoint=ff}"];
            [set addObject:@"{CGRect={CGPoint=ff}{CGSize=ff}}"];
            [set addObject:@"{CGAffineTransform=ffffff}"];
            [set addObject:@"{UIEdgeInsets=ffff}"];
            [set addObject:@"{UIOffset=ff}"];
            // 64 bit
            [set addObject:@"{CGSize=dd}"];
            [set addObject:@"{CGPoint=dd}"];
            [set addObject:@"{CGRect={CGPoint=dd}{CGSize=dd}}"];
            [set addObject:@"{CGAffineTransform=dddddd}"];
            [set addObject:@"{UIEdgeInsets=dddd}"];
            [set addObject:@"{UIOffset=dd}"];
            types = set;
        });
        if ([types containsObject:propertyInfo.typeEncoding]) {
            meta->_isStructAvailableForKeyedArchiver = YES;
        }
    }
```

- dispatch_once：创建能够序列化的结构体类型，如果属性属于其中之一，标记一下
- 只创建一次，这是yymodel最常见的性能优化之一

代码块

```
    if (meta->_getter && meta->_setter) {
        /*
         KVC invalid type:
         long double
         pointer (such as SEL/CoreFoundation object)
         */
        switch (meta->_type & YYEncodingTypeMask) {
```

```
            case YYEncodingTypeBool:
            case YYEncodingTypeInt8:
            case YYEncodingTypeUInt8:
            case YYEncodingTypeInt16:
            case YYEncodingTypeUInt16:
            case YYEncodingTypeInt32:
            case YYEncodingTypeUInt32:
            case YYEncodingTypeInt64:
            case YYEncodingTypeUInt64:
            case YYEncodingTypeFloat:
            case YYEncodingTypeDouble:
            case YYEncodingTypeObject:
            case YYEncodingTypeClass:
            case YYEncodingTypeBlock:
            case YYEncodingTypeStruct:
            case YYEncodingTypeUnion: {
                meta->_isKVCCompatible = YES;
            } break;
            default: break;
        }
    }
```

- 标记能不能通过KVC赋值，具体作用见YYModel源码分析（二）

回到ModelMeta的初始化方法

代码块

```
    if ([cls respondsToSelector:@selector(modelCustomPropertyMapper)]) {
        NSDictionary *customMapper = [(id <YYModel>)cls modelCustomPropertyMapper];
        [customMapper enumerateKeysAndObjectsUsingBlock:^(NSString *propertyName, NSString *mappedToKey, BOOL *stop) {
            _YYModelPropertyMeta *propertyMeta = allPropertyMetas[propertyName];
            if (!propertyMeta) return;
            [allPropertyMetas removeObjectForKey:propertyName];

            if ([mappedToKey isKindOfClass:[NSString class]]) {
                if (mappedToKey.length == 0) return;

                propertyMeta->_mappedToKey = mappedToKey;
                NSArray *keyPath = [mappedToKey componentsSeparatedByString:@"."];
                for (NSString *onePath in keyPath) {
                    if (onePath.length == 0) {
                        NSMutableArray *tmp = keyPath.mutableCopy;
```

```
16                          [tmp removeObject:@""];
17                          keyPath = tmp;
18                          break;
19                      }
20                  }
21                  if (keyPath.count > 1) {
22                      propertyMeta->_mappedToKeyPath = keyPath;
23                      [keyPathPropertyMetas addObject:propertyMeta];
24                  }
25                  propertyMeta->_next = mapper[mappedToKey] ?: nil;
26                  mapper[mappedToKey] = propertyMeta;
27
28              } else if ([mappedToKey isKindOfClass:[NSArray class]]) {
29
30                  NSMutableArray *mappedToKeyArray = [NSMutableArray new];
31                  for (NSString *oneKey in ((NSArray *)mappedToKey)) {
32                      if (![oneKey isKindOfClass:[NSString class]]) continue;
33                      if (oneKey.length == 0) continue;
34
35                      NSArray *keyPath = [oneKey
    componentsSeparatedByString:@"."];
36                      if (keyPath.count > 1) {
37                          [mappedToKeyArray addObject:keyPath];
38                      } else {
39                          [mappedToKeyArray addObject:oneKey];
40                      }
41
42                      if (!propertyMeta->_mappedToKey) {
43                          propertyMeta->_mappedToKey = oneKey;
44                          propertyMeta->_mappedToKeyPath = keyPath.count > 1 ?
    keyPath : nil;
45                      }
46                  }
47                  if (!propertyMeta->_mappedToKey) return;
48
49                  propertyMeta->_mappedToKeyArray = mappedToKeyArray;
50                  [multiKeysPropertyMetas addObject:propertyMeta];
51
52                  propertyMeta->_next = mapper[mappedToKey] ?: nil;
53                  mapper[mappedToKey] = propertyMeta;
54              }
55          }];
56      }
```

分析源码，理解关键的数据结构和属性含义非常重要

数据结构解释：

- mapper：key/keyPath映射到属性
- keyPathPropertyMetas：keyPath映射到的属性
- multiKeysPropertyMetas：保存多个key映射到同一个上面的属性

逻辑解释：

- 找到对应的PropertyMeta并从原数组中移除，表示它已经被业务自定义的逻辑处理过了
- 接下来构建mappedToKey、mappedToKeyPath、mappedToKeyArray方便之后构建JSON
- Key to Property一对一映射处理，_next表示这个属性之前已经被映射到了别的key，这里用链表连接；根据mappedToKey的情况分别处理keyPath和普通的key
- 多key映射一个property：处理多个key映射到一个属性的情况，mappedToKey、mappedToKeyPath都先只取第一个进行复制，然后处理next的问题
- 最后会给非业务自定义propertyMeta对应的mappedToKey、next的赋值，然后在mapper中建立映射

3.3.1 解释

1. 把对应的属性移除，表示它已经在自定义业务被处理了

代码块

```
if (allPropertyMetas.count) _allPropertyMetas =
allPropertyMetas.allValues.copy;
```

2. 在处理自定义的map的时候，会从原数组中删除对应的propertyMeta，处理完业务自定义的属性，会给其余的属性的mapToKey进行赋值、处理多属性映射到同一个key的链表连接、mapper映射

代码块

```
_YYModelPropertyMeta *propertyMeta = allPropertyMetas[propertyName];
if (!propertyMeta) return;
[allPropertyMetas removeObjectForKey:propertyName];
// 其它业务自定义逻辑

    [allPropertyMetas enumerateKeysAndObjectsUsingBlock:^(NSString *name,
_YYModelPropertyMeta *propertyMeta, BOOL *stop) {
        propertyMeta->_mappedToKey = name;
        propertyMeta->_next = mapper[name] ?: nil;
        mapper[name] = propertyMeta;
    }];

```

3.4 ModelSetWithDictionary

代码块

```
- (BOOL)modelSetWithDictionary:(NSDictionary *)dic {
    if (!dic || dic == (id)kCFNull) return NO;
    if (![dic isKindOfClass:[NSDictionary class]]) return NO;

    _YYModelMeta *modelMeta = [_YYModelMeta metaWithClass:object_getClass(self)];
    if (modelMeta->_keyMappedCount == 0) return NO;

    if (modelMeta->_hasCustomWillTransformFromDictionary) {
        dic = [((id<YYModel>)self) modelCustomWillTransformFromDictionary:dic];
        if (![dic isKindOfClass:[NSDictionary class]]) return NO;
    }

    ModelSetContext context = {0};
    context.modelMeta = (__bridge void *)(modelMeta);
    context.model = (__bridge void *)(self);
    context.dictionary = (__bridge void *)(dic);

    if (modelMeta->_keyMappedCount >= CFDictionaryGetCount((CFDictionaryRef)dic)) {
        CFDictionaryApplyFunction((CFDictionaryRef)dic, ModelSetWithDictionaryFunction, &context);
        if (modelMeta->_keyPathPropertyMetas) {
            CFArrayApplyFunction((CFArrayRef)modelMeta->_keyPathPropertyMetas,
                                 CFRangeMake(0, CFArrayGetCount((CFArrayRef)modelMeta->_keyPathPropertyMetas)),
                                 ModelSetWithPropertyMetaArrayFunction,
                                 &context);
        }
        if (modelMeta->_multiKeysPropertyMetas) {
            CFArrayApplyFunction((CFArrayRef)modelMeta->_multiKeysPropertyMetas,
                                 CFRangeMake(0, CFArrayGetCount((CFArrayRef)modelMeta->_multiKeysPropertyMetas)),
                                 ModelSetWithPropertyMetaArrayFunction,
                                 &context);
        }
    } else {
        CFArrayApplyFunction((CFArrayRef)modelMeta->_allPropertyMetas,
                             CFRangeMake(0, modelMeta->_keyMappedCount),
                             ModelSetWithPropertyMetaArrayFunction,
                             &context);
```

```
    }

    if (modelMeta->_hasCustomTransformFromDictionary) {
        return [((id<YYModel>)self) modelCustomTransformFromDictionary:dic];
    }
    return YES;
}
```

- 使用C数据结构、Core Foundation数据结构和方法调用优化性能
- 区分两种情况进行赋值，如果实际映射大于dict给的数量，使用第一种方式赋值遍历字典的key（循环更少），如果小于dict给的数量，遍历meta中的key（循环更少）
- 单独处理keyPath、多个key映射到一个属性的情况，这样直接处理只有key的情况的属性会更快，因为不用在额外的判断

代码块

```
static void ModelSetWithPropertyMetaArrayFunction(const void *_propertyMeta, void *_context) {
    ModelSetContext *context = _context;
    __unsafe_unretained NSDictionary *dictionary = (__bridge NSDictionary *)(context->dictionary);
    __unsafe_unretained _YYModelPropertyMeta *propertyMeta = (__bridge _YYModelPropertyMeta *)(_propertyMeta);
    if (!propertyMeta->_setter) return;
    id value = nil;

    if (propertyMeta->_mappedToKeyArray) {
        value = YYValueForMultiKeys(dictionary, propertyMeta->_mappedToKeyArray);
    } else if (propertyMeta->_mappedToKeyPath) {
        value = YYValueForKeyPath(dictionary, propertyMeta->_mappedToKeyPath);
    } else {
        value = [dictionary objectForKey:propertyMeta->_mappedToKey];
    }

    if (value) {
        __unsafe_unretained id model = (__bridge id)(context->model);
        ModelSetValueForProperty(model, value, propertyMeta);
    }
}
```

- 根据不同的情况解析key，YYValueForMultiKeys内部，还是会取dictionary中第一个找到的key对应的值，给属性进行赋值

- YYValueForKeyPath会根据dict、keyPath一层一层找到value，非常安全，因为中间只要一层key对应的value不是dictionary类型，就会返回nil
- 最后是根据key找到对应的value
- CFDictionaryApplyFunction直接遍历ModelSetValueForProperty所以性能更好，不用进行if判断，当确定属性不是keyPath或者multiKeys的时候优化性能

3.4.1 ModelValueForProperty

这个方法超级长，所以不引用代码了

- isCNumber：判断是不是C类型的数字，如果是就尝试把value进行转换，可能包括NSString、NSNumber
- 如果是nsType类型是NSString类：判断value是不是NSString、NSData、NSNumber、NSAttributedString，如果是就进行转换
- 前半部分逻辑类似，就是判断属性的nsType，如果是基础类型，就枚举能转换成这个基础类型的其它基础类型，然后尝试把value转换成nsType进行赋值
- 如果是集合类型，会先判断也没有genericCls，然后遍历value数组中的元素one进行赋值
  - 如果one是这个类型的就尝试，成功添加进结果数组
  - 如果one不是这个类型(不是这个类型也必须是字典类型)就会先判断业务有没有_hasCustomClassFromDictionary，如果没有就还是用genericCls，然后把当前的one转换成model，还是通过modelSetWithDictionary
  - 如果不是这个类型，但是能转换成这个类型的值，就添加进结果数组
- 集合类型如果没genericCls，就直接赋值，不进行转换
- 字典和Set类型类似数组，只是类型上不同
- 如果是对象类型，区分当前getter有没有值，有值直接赋值，没有需要先创建对象的实例然后赋值，这里创建对象的实例的优先级还是 业务自定义类型 > genericCls > 属性解析处理的cls
- 类对象类型直接赋值，会判断如果value是NSString类型，就用NSClassFromString尝试进行转换
- 其它类型都会做基本的处理，和业务无关了，这里不再分析了

4. 总结

代码块

```
- (BOOL)modelSetWithDictionary:(NSDictionary *)dic;
```

- 创建或者获取modelMeta的缓存对象
- 再给业务一次机会在赋值前改变传入的dictionary，可以再更改一些值等

- 对比allPropertyMetas的数量和业务传入的字典的键值对数量，判断赋值方式，遍历较少的一方
- 最后允许业务对一些属性做自定义的处理，比如你想把服务端下发的字符串类型的Date（此时也许已经被转化成NSDate）转成时间戳类型，就可以在这里做

思考：为什么当dict键值对数量大于propertyMetas的时候不区分key、keyPath、多个key的情况赋值？

首先：在这里yymodel是做了优化

当属性数量 > 字典键值对：先遍历字典给能对应的属性赋值（这里不包括keyPath、multiKeyProperty，因为这些存储的还是点语法和数组，所以一般无法对应），然后再处理keyPath、multiKeysProperty

这样遍历的次数更少，而且直接赋值的部分不需要做if语句的判断，性能非常好

字典键值对 > 属性数量的时候：直接用ModelSetWithPropertyMetaArrayFunction遍历属性赋值，这个方法里会做关于这个属性存储的是key、keyPath、multiKeysProperty的判断

- 所以为什么后者不继续前者的逻辑，把普通的key和keyPath这些分开来赋值

答：分开赋值相当于要先遍历m次字典，再遍历n+k次分别判断keyPath、multiKeysProperty，时间复杂度是O(m + n + k)，如果这样遍历赋值，优化if语句带来的收益会小于多判断的n + k次循环，所以直接遍历propertyMetas赋值是最快的

思考：在这过程中都有什么性能优化？

1. 缓存ModelMeta、ClassInfo，多次解析提升性能
2. 缓存不变的集合类型，在多次解析中只创建一次
3. 解析属性的时候区分key、keyPath、multiKeysProperty的情况，为后续优化性能做准备
4. 通过runtime方法解析属性、实例变量、方法，然后通过objc_msgSend直接给属性的setter发消息，绕过KVC赋值，性能更好
5. 在解析的时候区分基础类型，优先赋值（其它业务类型的属性可能还需调用modelSetWithDictionary进行解析）

思考：yymodel如何做到比其它sdk更加全面，对比mjextension的优势是什么？

1. yymodel在解析、赋值的时候会根据属性类型对value进行必要的转换，比如把NSString类型的value尝试转换成NSNumber、把NSData、NSURL类型的value转换成NSString等
2. yymodel在解析的时候只缓存一次类对象的结构体、赋值通过objc_msgSend而不是KVC，性能更好
3. yymodel面向协议编程，而mjextension通过Category把方法硬编码进NSObject，通过继承获得能力

思考：如何在业务中更好的使用yymodel？

1. 集合对象总是使用genericCls，不要让yymodel推测类型，如果要它推测类型，复杂类型可能不会正确赋值

代码块

```
if (meta->_nsType == YYEncodingTypeNSDictionary) {
    ((void (*)(id, SEL, id))(void *) objc_msgSend)((id)model, meta->_setter, value);
} else {
    ((void (*)(id, SEL, id))(void *) objc_msgSend)((id)model,
                                                   meta->_setter,
                                                   ((NSDictionary *)value).mutableCopy);
}
```

考虑这段代码，没有genericCls的时候会走到这里，如果字典对象中有很多自定义类型，而value中这些对象类型还没有转换（如果是对象，可能这些key对应的value还是字典），那么会直接给这些对象赋值字典，这个时候就会crash

2. modelCustomTransformFromDictionary:可以做一些更加复杂的转换，比如把NSNumber类型转换成NSDate等
3. 无法解析NSProxy类型对象，别忘了放进blacklist
4. 无法解析protocol，除非这个protocol和某个类对象的名称完全一致，不然无法赋值

补充：block对象如何获取基类

代码块

```
static force_inline Class YYNSBlockClass() {
    static Class cls;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        void (^block)(void) = ^{};
        cls = ((NSObject *)block).class;
        while (class_getSuperclass(cls) != [NSObject class]) {
            cls = class_getSuperclass(cls);
        }
    });
    return cls; // current is "NSBlock"
}
```

block是一个有isa的对象，属于objective-c的对象定义，一般碰到的block对象都是NSMallocBlock、NSStackBlock、NSGlobalBlock，而这里没有捕获任何变量的block就是一个NSGlobalBlock对象

这里找到block对象的基类的方式很有意思

😁 为什么“有 isa 的就是对象”

在 Objective-C 的 runtime 里，有一些关键约定：

1. **对象的内存起始总是 isa**
   - 访问对象时，runtime 第一步是通过 `isa` 找类
   - runtime 可以通过 `objc_msgSend` 查找方法实现：
   - `IMP imp = class_getMethodImplementation(obj->isa, @selector(foo));`
2. **没有 isa 的内存块不是对象**
   - 任何没有 `isa` 指针的 struct/内存块无法参与消息发送
   - runtime 会认为它不是对象，调用方法会 crash

所以，一个 **合法 Objective-C 对象的最小要求就是拥有 isa 指针**。

A. 必备知识

1. 为什么要 & YYEncodingTypeMask?

代码块

```
typedef NS_OPTIONS(NSUInteger, YYEncodingType) {
    YYEncodingTypeMask       = 0xFF, ///< mask of type value
    YYEncodingTypeUnknown    = 0, ///< unknown
    YYEncodingTypeVoid       = 1, ///< void
    YYEncodingTypeBool       = 2, ///< bool
    YYEncodingTypeInt8       = 3, ///< char / BOOL
    YYEncodingTypeUInt8      = 4, ///< unsigned char
    YYEncodingTypeInt16      = 5, ///< short
    // 其它
    YYEncodingTypePropertyMask         = 0xFF0000, ///< mask of property
    YYEncodingTypePropertyReadonly     = 1 << 16, ///< readonly
    YYEncodingTypePropertyCopy         = 1 << 17, ///< copy
}

switch (meta->_type & YYEncodingTypeMask) {
    // 逻辑
}
```

- 0xFF：十六进制，二进制低八位全为1
- &：计算公式速记，只有1 & 1 = 1，其它都等于0，也就是说 type & 0xFF，只会保留第八位原本为1的值
- |：只要为1，就等于1

- Type & YYEncodingTypeMask：只取低八位转换为类型，其余mask同理

2. CFArrayApplyFunction

代码块

```
CFArrayApplyFunction((CFArrayRef)modelMeta->_keyPathPropertyMetas,
                     CFRangeMake(0, CFArrayGetCount((CFArrayRef)modelMeta->_keyPathPropertyMetas)),
                     ModelSetWithPropertyMetaArrayFunction,
                     &context);
```

- CoreFoudation的高性能遍历方法：实际作用和 for 循环一致，但**用 C 函数指针回调，无需 Objective-C runtime、无引用计数、无对象包装拆包，极致快**