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];
```

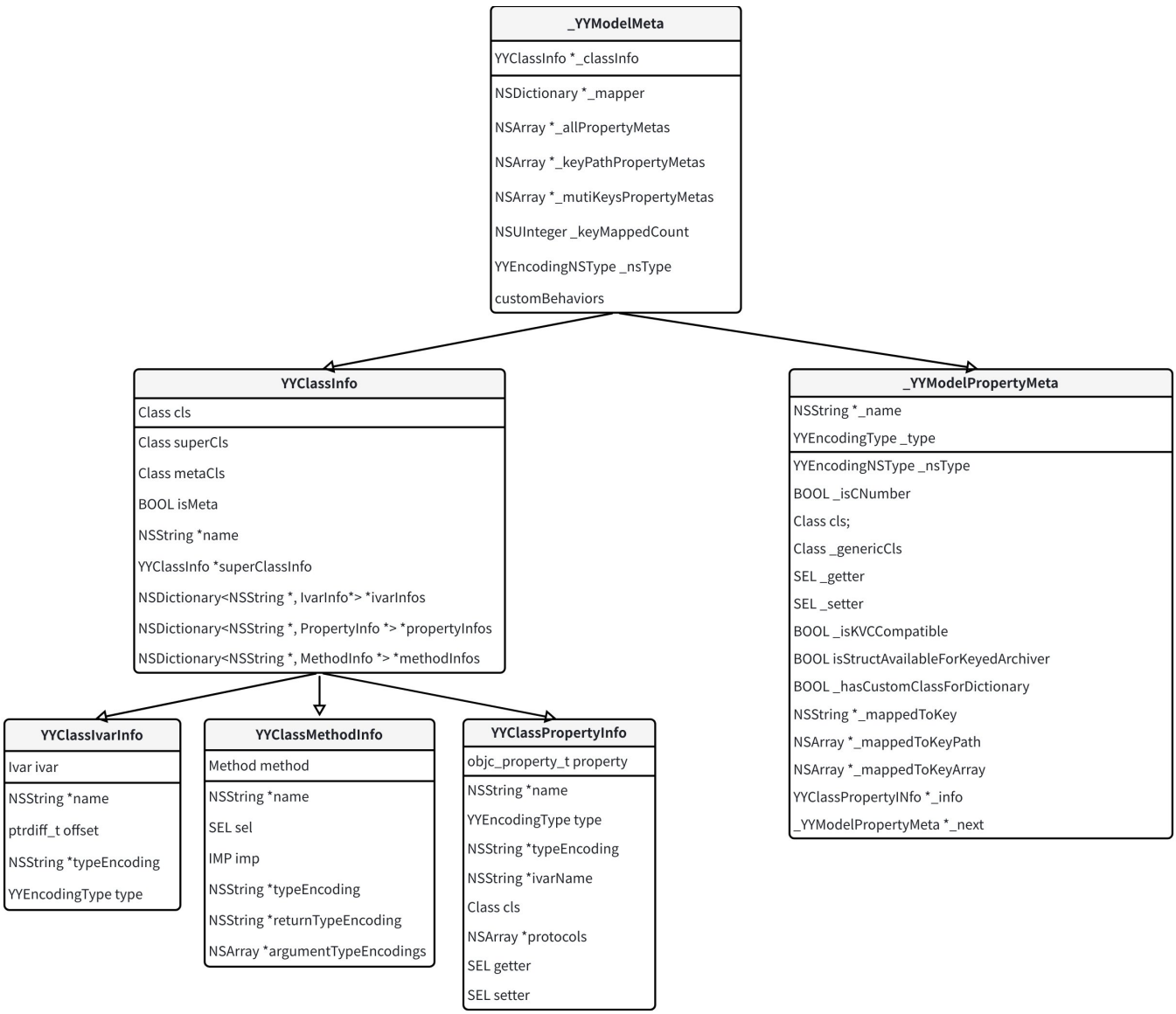
1. 优化

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;
}
```

- 这里有两个疑惑的点，为什么要创建ModelMeta
  - 因为通过respondsToSelector()也能判断这个方法有没有实现，实际上不影响调用
  - modelCustomClassForDictionary这个方法又是在做什么呢，为什么要赋值给class

_YYModelMeta的创建

代码块

```
+ (instancetype)metaWithClass:(Class)cls {
    if (!cls) return nil;
    static CFMutableDictionaryRef cache;
    static dispatch_once_t onceToken;
```

```
    static dispatch_semaphore_t lock;
    dispatch_once(&onceToken, ^{
        cache = CFDictionaryCreateMutable(CFAllocatorGetDefault(), 0, &kCFTypeDictionaryKeyCallBacks, &kCFTypeDictionaryValueCallBacks);
        lock = dispatch_semaphore_create(1);
    });
    dispatch_semaphore_wait(lock, DISPATCH_TIME_FOREVER);
    _YYModelMeta *meta = CFDictionaryGetValue(cache, (__bridge const void *)(cls));
    dispatch_semaphore_signal(lock);
    if (!meta || meta->_classInfo.needUpdate) {
        meta = [[_YYModelMeta alloc] initWithClass:cls];
        if (meta) {
            dispatch_semaphore_wait(lock, DISPATCH_TIME_FOREVER);
            CFDictionarySetValue(cache, (__bridge const void *)(cls), (__bridge const void *)(meta));
            dispatch_semaphore_signal(lock);
        }
    }
    return meta;
}
```

- 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
```

```
10      NSSet *whitelist = nil;
11      if ([cls respondsToSelector:@selector(modelPropertyWhitelist)]) {
12          NSArray *properties = [(id<YYModel>)cls modelPropertyWhitelist];
13          if (properties) {
14              whitelist = [NSSet setWithArray:properties];
15          }
16      }
```

- 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

```
1       // Get container property's generic class
2       NSDictionary *genericMapper = nil;
3       if ([cls
    respondsToSelector:@selector(modelContainerPropertyGenericClass)]) {
4           genericMapper = [(id<YYModel>)cls modelContainerPropertyGenericClass];
5           if (genericMapper) {
6               NSMutableDictionary *tmp = [NSMutableDictionary new];
7               [genericMapper enumerateKeysAndObjectsUsingBlock:^(id key, id obj,
    BOOL *stop) {
8                   if (![key isKindOfClass:[NSString class]]) return;
9                   Class meta = object_getClass(obj);
10                  if (!meta) return;
11                  if (class_isMetaClass(meta)) {
12                      tmp[key] = obj;
13                  } else if ([obj isKindOfClass:[NSString class]]) {
14                      Class cls = NSClassFromString(obj);
15                      if (cls) {
16                          tmp[key] = cls;
17                      }
18                  }
19              }];
20              genericMapper = tmp;
21          }
22      }
```

- 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

YYClassPropertyInfo

代码块

```
1       // Create all property metas.
2       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;
}
```

- 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

```
- (instancetype)initWithClass:(Class)cls {
    if (!cls) return nil;
    self = [super init];
    _cls = cls;
    _superCls = class_getSuperclass(cls);
    _isMeta = class_isMetaClass(cls);
    if (!_isMeta) {
        _metaCls = objc_getMetaClass(class_getName(cls));
    }
    _name = NSStringFromClass(cls);
    [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];
11          _methodInfos = methodInfos;
12          for (unsigned int i = 0; i < methodCount; i++) {
13              YYClassMethodInfo *info = [[YYClassMethodInfo alloc]
    initWithMethod:methods[i]];
14              if (info.name) methodInfos[info.name] = info;
15          }
16          free(methods);
17      }
18      unsigned int propertyCount = 0;
19      objc_property_t *properties = class_copyPropertyList(cls, &propertyCount);
20      if (properties) {
21          NSMutableDictionary *propertyInfos = [NSMutableDictionary new];
22          _propertyInfos = propertyInfos;
23          for (unsigned int i = 0; i < propertyCount; i++) {
24              YYClassPropertyInfo *info = [[YYClassPropertyInfo alloc]
    initWithProperty:properties[i]];
25              if (info.name) propertyInfos[info.name] = info;
26          }
27          free(properties);
28      }
```

```
29
30      unsigned int ivarCount = 0;
31      Ivar *ivars = class_copyIvarList(cls, &ivarCount);
32      if (ivars) {
33          NSMutableDictionary *ivarInfos = [NSMutableDictionary new];
34          _ivarInfos = ivarInfos;
35          for (unsigned int i = 0; i < ivarCount; i++) {
36              YYClassIvarInfo *info = [[YYClassIvarInfo alloc]
    initWithIvar:ivars[i]];
37              if (info.name) ivarInfos[info.name] = info;
38          }
39          free(ivars);
40      }
41
42      if (!_ivarInfos) _ivarInfos = @{};
43      if (!_methodInfos) _methodInfos = @{};
44      if (!_propertyInfos) _propertyInfos = @{};
45
46      _needUpdate = NO;
47  }
```

- 通过runtime的方法copyMethodList，copyPropertyList，copyIvarList初始化ModelInfo的信息
- 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

```
1   - (instancetype)initWithProperty:(objc_property_t)property {
2       // 基本的初始化
3       objc_property_attribute_t *attrs = property_copyAttributeList(property,
    &attrCount);
4       for (unsigned int i = 0; i < attrCount; i++) {
5           switch (attrs[i].name[0]) {
6               case 'T': { // Type encoding
7                   if (attrs[i].value) {
8                       _typeEncoding = [NSString
    stringWithUTF8String:attrs[i].value];
9                       type = YYEncodingGetType(attrs[i].value);
10
11                      if ((type & YYEncodingTypeMask) == YYEncodingTypeObject &&
    _typeEncoding.length) {
```

```
12                          NSScanner *scanner = [NSScanner
    scannerWithString:_typeEncoding];
13                          if (![scanner scanString:@"@\"" intoString:NULL])
    continue;
14
15                          NSString *clsName = nil;
16                          if ([scanner scanUpToCharactersFromSet: [NSCharacterSet
     characterSetWithCharactersInString:@"\"<"] intoString:&clsName]) {
17                              if (clsName.length) _cls =
    objc_getClass(clsName.UTF8String);
18                          }
19
20                          NSMutableArray *protocols = nil;
21                          while ([scanner scanString:@"<" intoString:NULL]) {
22                              NSString* protocol = nil;
23                              if ([scanner scanUpToString:@">" intoString:
    &protocol]) {
24                                  if (protocol.length) {
25                                      if (!protocols) protocols = [NSMutableArray
     new];
26                                      [protocols addObject:protocol];
27                                  }
28                              }
29                              [scanner scanString:@">" intoString:NULL];
30                          }
31                          _protocols = protocols;
32                      }
33                  }
34              } break;
35              case 'V': { // Instance variable
36                  if (attrs[i].value) {
37                      _ivarName = [NSString stringWithUTF8String:attrs[i].value];
38                  }
39              } break;
40              case 'R': {
41                  type |= YYEncodingTypePropertyReadonly;
42              } break;
43              case 'C': {
44                  type |= YYEncodingTypePropertyCopy;
45              } break;
46              case '&': {
47                  type |= YYEncodingTypePropertyRetain;
48              } break;
49              case 'N': {
50                  type |= YYEncodingTypePropertyNonatomic;
51              } break;
52              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;
            }
        }
```

```
10      }
```

- 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

```
1       _YYModelPropertyMeta *meta = [self new];
2       meta->_name = propertyInfo.name;
3       meta->_type = propertyInfo.type;
4       meta->_info = propertyInfo;
5       meta->_genericCls = generic;
6
7       if ((meta->_type & YYEncodingTypeMask) == YYEncodingTypeObject) {
8           meta->_nsType = YYClassGetNSType(propertyInfo.cls);
9       } else {
10          meta->_isCNumber = YYEncodingTypeIsCNumber(meta->_type);
11      }
```

- 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

```
1       if ((meta->_type & YYEncodingTypeMask) == YYEncodingTypeStruct) {
2           /*
3            It seems that NSKeyedUnarchiver cannot decode NSValue except these
    structs:
4            */
5           static NSSet *types = nil;
6           static dispatch_once_t onceToken;
7           dispatch_once(&onceToken, ^{
8               NSMutableSet *set = [NSMutableSet new];
9               // 32 bit
10              [set addObject:@"{CGSize=ff}"];
11              [set addObject:@"{CGPoint=ff}"];
12              [set addObject:@"{CGRect={CGPoint=ff}{CGSize=ff}}"];
13              [set addObject:@"{CGAffineTransform=ffffff}"];
14              [set addObject:@"{UIEdgeInsets=ffff}"];
15              [set addObject:@"{UIOffset=ff}"];
16              // 64 bit
17              [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;
                        [tmp removeObject:@""];
                        keyPath = tmp;
                        break;
                    }
                }
                if (keyPath.count > 1) {
                    propertyMeta->_mappedToKeyPath = keyPath;
                    [keyPathPropertyMetas addObject:propertyMeta];
                }
                propertyMeta->_next = mapper[mappedToKey] ?: nil;
                mapper[mappedToKey] = propertyMeta;

            } else if ([mappedToKey isKindOfClass:[NSArray class]]) {

                NSMutableArray *mappedToKeyArray = [NSMutableArray new];
                for (NSString *oneKey in ((NSArray *)mappedToKey)) {
                    if (![oneKey isKindOfClass:[NSString class]]) continue;
                    if (oneKey.length == 0) continue;

                    NSArray *keyPath = [oneKey componentsSeparatedByString:@"."];
                    if (keyPath.count > 1) {
                        [mappedToKeyArray addObject:keyPath];
```

```
                    } else {
                        [mappedToKeyArray addObject:oneKey];
                    }

                    if (!propertyMeta->_mappedToKey) {
                        propertyMeta->_mappedToKey = oneKey;
                        propertyMeta->_mappedToKeyPath = keyPath.count > 1 ? keyPath : nil;
                    }
                }
                if (!propertyMeta->_mappedToKey) return;

                propertyMeta->_mappedToKeyArray = mappedToKeyArray;
                [multiKeysPropertyMetas addObject:propertyMeta];

                propertyMeta->_next = mapper[mappedToKey] ?: nil;
                mapper[mappedToKey] = propertyMeta;
            }
        }];
    }
```

- 最重要的是使用keyPath的方式处理json中嵌套形式的数据，第一个if如果发现属性是一个点语法连接的字符串，就会转成数组，删除空字符串
- 如果mapper中已经有同名的字符串了，就把它以链表形式接在上一个映射后面，这里主要为了处理客户端兼容服务端下发不同字段的情况
- 如果映射到的是Array，就会遍历这个数组，把每一个字符串区分key和keyPath，如果处理过一次就不会再处理，也就是说如果数组中有重复的，会优先考虑前面的
- 保存解析的上下文

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**