

YYModel源码（二）

1. YYModel的优势

- MJExtension也有和YYModel相似的功能，而它主要使用KVC进行属性的赋值

KVC（Key-Value Coding）是 Foundation 框架提供的，通过 setValue:forKey:、setValuesForKeysWithDictionary: 赋值。KVC 效率偏低，因为它会走 willChangeValueForKey:、字符串查找、容错等多层逻辑，还会处理未定义 Key、嵌套 KVC、自动类型转换等边界场景

- YYModel使用runtime运行时的反射能力动态获取属性并赋值，并在首次解析的时候缓存classInfo和modelMeta，可以说把性能优化做到了极致

YYModel作者对于不同JSON模型转换差异分析这篇文章对于性能、容错、功能、侵入（和业务解藕程度）几个方面都解释的很好了，总的来说，不同的SDK都各有所长。

总结下来就是mantle最全面，yymodel性能最好，容错性更好，解藕程度更好

- 解藕能力

面向协议编程

Mantle	MJExtension	YYModel
通过继承获得能力，和业务高度耦合	通过Category给NSObject添加方法，业务实现时生效	面向协议编程，运行时动态查找，更加解藕

1.1 容错性

容错性的目标是：

1. 程序不崩溃
2. 输入异常时仍能保证功能可用和结果合理
3. 对异常有可预期的处理策略，而不是直接丢掉或静默失败

- 输入有效性

- 非字典直接拒绝：!dic || dic == (id)kCFNull || ![dic isKindOfClass:NSDictionary.class] → 返回 NO，不崩。
- 属性元信息缓存失败/无可映射属性（_keyMappedCount == 0）→ 返回 NO。

- 未知/缺失字段

- 缺失字段：跳过，不赋值（不崩）

- 多余字段：忽略，不影响已有属性（不崩）
- **Key 映射与回退**
 - +modelCustomPropertyMapper 支持单 key / keyPath / 多 key。
多 key 时采用“第一命中即用”的回退链策略（_next 链），提升容错（接口字段变动仍能取到值）
- **类型兼容与安全转换（高频）**
 - NSString - NSNumber：常见数字字符串会被安全转换为数值；数值也会转字符串。
 - NSString - NSURL：通过 +[NSURL URLWithString:]。
 - NSString → NSDate：内置多格式解析（常见formatter、时间戳等，解析失败则置空/跳过）

思考：实际上直接赋值nil、跳过、crash都还是要依赖于具体场景，因为有的问题就是需要在debug环境就发现，但是同时要尽量保证线上不会crash，考虑这一段代码，类似的代码也还有

代码块

```

1  case YYEncodingTypeNSURL: {
2      if ([value isKindOfClass:[NSURL class]]) {
3          ((void (*)(id, SEL, id))(void *) objc_msgSend)((id)model, meta-
>_setter, value);
4      } else if ([value isKindOfClass:[NSString class]]) {
5          NSCharacterSet *set = [NSCharacterSet
whitespaceAndNewlineCharacterSet];
6          NSString *str = [value stringByTrimmingCharactersInSet:set];
7          if (str.length == 0) {
8              ((void (*)(id, SEL, id))(void *) objc_msgSend)((id)model, meta-
>_setter, nil);
9          } else {
10             ((void (*)(id, SEL, id))(void *) objc_msgSend)((id)model, meta-
>_setter, [[NSURL alloc] initWithString:str]);
11         }
12     }
13 } break;

```

实际上在业务中，如果value无法转换成业务需要的值，也许应该让业务提前感知，而不是跳过，但是yymodel的做法也不是完全静默，分析一下它的行为：

- 先尝试赋值，如果无法赋值，它选择相信业务能够在modelCustomTransformFromDictionary方法里合理的处理无法赋值的属性
- 在这里直接跳过赋值

但实际上这样的方式还是不够好，如果modelCustomTransformFromDictionary方法在赋值之前发生会不会更好，这样的时序也许更加合理

1. 业务调用modelCustomTransformFromDictionary给特殊的值赋值，比如NSNumber转换成NSDate，yymodel可以在赋值的时候判断这个属性有没有被业务处理，如果有就跳过
 2. 在最后，如果有key、有value的情况下，还是应该在debug环境下断言，让业务方提前发现问题
- 类似这样做：

代码块

```
1  #if DEBUG
2      NSAssert(NO, @"Unexpected type for URL: %@", [value class]);
3  #endif
4      return nil;
```

1.2 全面性

体现全面性的主要方法是ModelSetValueForProperty，因为方法比较长，所以我们拆分成几份进行分析

1. C语言数字类型赋值，支持多种类型的value转换成C语言数字

代码块

```
1  {
2      if (meta->_isCNumber) {
3          NSNumber *num = YYNSNumberCreateFromID(value);
4          ModelSetNumberToProperty(model, num, meta);
5          if (num != nil) [num class]; // hold the number
6      }
7  }
8
9  static force_inline NSNumber *YYNSNumberCreateFromID(__unsafe_unretained id
value) {
10     static NSCharacterSet *dot;
11     static NSDictionary *dic;
12     static dispatch_once_t onceToken;
13     dispatch_once(&onceToken, ^{
14         dot = [NSCharacterSet characterSetWithRange:NSMakeRange('.', 1)];
15         dic = @{@"TRUE" : @(YES),
16                 @"True" : @(YES),
17                 @"true" : @(YES),
18                 @"FALSE" : @(NO),
19                 @"False" : @(NO),
20                 @"false" : @(NO),
21                 @"YES" : @(YES),
22                 @"Yes" : @(YES),
23                 @"yes" : @(YES),
24                 @"NO" : @(NO),
```

```

25         @"No" :      @(NO),
26         @"no" :      @(NO),
27         @"NIL" :     (id)kCFNull,
28         @"Nil" :     (id)kCFNull,
29         @"nil" :     (id)kCFNull,
30         @"NULL" :    (id)kCFNull,
31         @"Null" :    (id)kCFNull,
32         @"null" :    (id)kCFNull,
33         @"(NULL)" :  (id)kCFNull,
34         @"(Null)" :  (id)kCFNull,
35         @"(null)" :  (id)kCFNull,
36         @"<NULL>" :  (id)kCFNull,
37         @"<Null>" :  (id)kCFNull,
38         @"<null>" :  (id)kCFNull};
39     });
40
41     if (!value || value == (id)kCFNull) return nil;
42     if ([value isKindOfClass:[NSNumber class]]) return value;
43     if ([value isKindOfClass:[NSString class]]) {
44         NSNumber *num = dic[value];
45         if (num != nil) {
46             if (num == (id)kCFNull) return nil;
47             return num;
48         }
49         if ([([NSString *)value rangeOfCharacterFromSet:dot].location !=
50             NSRangeNotFound) {
51             const char *cstring = ((NSString *)value).UTF8String;
52             if (!cstring) return nil;
53             double num = atof(cstring);
54             if (isnan(num) || isinf(num)) return nil;
55             return @(num);
56         } else {
57             const char *cstring = ((NSString *)value).UTF8String;
58             if (!cstring) return nil;
59             return @(atoll(cstring));
60         }
61     }
62     return nil;

```

- dispatch_once: 缓存不可变集合类型，性能敏感的场景会极大的优化性能
- 这一部分内容主要是支持C语言数字类型的转换
 - 枚举了多种可能的场景如NSNumber、NSString、NSString还区分了bool、null、数字类型
 - 数字类型又区分了小数和整数不同的情况，不同的风格可能是为了兼容后端的代码风格

- atof和atoll主要是处理C语言中对不同类型的字符串转换成不同类型数字的情况
- ModelSetNumberToProperty: 之前讲过, 在创建modelMeta和classInfo的时候, 就已经解析了不同属性的具体类型, 这个方法主要就是根据YYNSNumberCreateFromID解析后的value, 区分属性的类型, 给属性赋值



内联函数: 内联这个名称就可以反映出它的工作方式, 函数会在它所调用的位置展开, 这么做可以消除函数调用和返回所带来的开销 (寄存器存储和恢复), 而且由于编译器会把调用函数的代码和函数本身放在一起优化, 所以也有进一步优化代码的可能。

但是这么做也是有代价的, 代码会变长, 也就意味着占用更多的内存空间或者更多的指令缓存。内核开发者通常会把对时间要求比较高, 比较短的函数定义成内联函数。如果函数比较大, 会被反复调用, 又没有特别的时间限制, 不建议做成内联函数

- 我们可以看到, 在上面这个方法中, 使用到了内联函数来优化性能, 这段代码实际上只有几个if-else判断, 因为dispatch_once只会被调用一次, 所以用来优化性能实际上是合理的

下个部分是基础类型赋值

基础类型赋值主要分为NSString类、NSNumber类、NSDate、NSNumber、NSData、NSURL、NSNumber、集合类型这些基础类型, 这些实际上没什么好分析的, 主要流程如下:

- 根据解析好属性的nsType, 区分类型走到不同的switch分支
- 根据value的类型 (使用isKindOfClass判断) 进行转换
- 使用objc_msgSend直接进行赋值, 速度极快, 这也是yymodel的主要特点之一

接下来是对于泛型、对象类型, 类对象类型、C语言类型的支持, 还有block、SEL, 其中简要分析如下, 代码会略过:

- block和SEL和指针等类型: 主要通过objc_msgSend进行赋值, 和之前的基础类型一样
- C语言结构体、CFArray: 使用KVC进行赋值, 因为objc_msgSend不支持这些类型的数据

代码主要分析对象、类对象类型

代码块

```
1  {
2      BOOL isNull = (value == (id)kCFNull);
3      switch (meta->_type & YYEncodingTypeMask) {
4          case YYEncodingTypeObject: {
5              Class cls = meta->_genericCls ?: meta->_cls;
6              if (isNull) {
7                  ((void (*)(id, SEL, id))(void *) objc_msgSend)((id)model,
meta->_setter, (id)nil);
8              } else if ([value isKindOfClass:cls] || !cls) {
```

```

9          ((void (*)(id, SEL, id))(void *) objc_msgSend)((id)model,
meta->_setter, (id)value);
10      } else if ([value isKindOfClass:[NSDictionary class]]) {
11          NSObject *one = nil;
12          if (meta->_getter) {
13              one = ((id (*)(id, SEL))(void *) objc_msgSend)
((id)model, meta->_getter);
14          }
15          if (one) {
16              [one modelSetWithDictionary:value];
17          } else {
18              if (meta->_hasCustomClassFromDictionary) {
19                  cls = [cls modelCustomClassForDictionary:value] ?:
cls;
20              }
21              one = [cls new];
22              [one modelSetWithDictionary:value];
23              ((void (*)(id, SEL, id))(void *) objc_msgSend)
((id)model, meta->_setter, (id)one);
24          }
25      }
26      } break;
27      case YYEncodingTypeClass: {
28          if (isNull) {
29              ((void (*)(id, SEL, Class))(void *) objc_msgSend)
((id)model, meta->_setter, (Class)NULL);
30          } else {
31              Class cls = nil;
32              if ([value isKindOfClass:[NSString class]]) {
33                  cls = NSClassFromString(value);
34                  if (cls) {
35                      ((void (*)(id, SEL, Class))(void *) objc_msgSend)
((id)model, meta->_setter, (Class)cls);
36                  }
37              } else {
38                  cls = object_getClass(value);
39                  if (cls) {
40                      if (class_isMetaClass(cls)) {
41                          ((void (*)(id, SEL, Class))(void *)
objc_msgSend)((id)model, meta->_setter, (Class)value);
42                      }
43                  }
44              }
45          }
46      } break;
47      default: break;
48  }

```

- 对象类型：支持泛型、对象、字典类型赋值
- 这里使用字典给对象类型赋值非常巧妙，展开讲一讲：
 - 先使用getter获取当前属性指向的对象，如果对象已经赋值了，就不用new一个对象了，就直接原地使用字典给这个对象赋值即可
 - 如果没有对象，再根据业务自定义行为判断它的类型，再进行new一个对象进行赋值
 - 这里性能优化也做的非常细
- 类对象赋值：区分值是NSString还是Class
 - NSString先转换成Class再进行赋值
 - Class类型会先判断它的class是不是metaClass，如果是才赋值，因为传入的value可能本身就是一个metaClass

1.3 GenericCls

代码块

```

1  {
2      case YYEncodingTypeNSDictionary:
3      case YYEncodingTypeNSMutableDictionary: {
4          if ([value isKindOfClass:[NSDictionary class]]) {
5              if (meta->_genericCls) {
6                  NSMutableDictionary *dic = [NSMutableDictionary new];
7                  [((NSDictionary *)value)
enumerateKeysAndObjectsUsingBlock:^(NSString *oneKey, id oneValue, BOOL *stop)
{
8                      if ([oneValue isKindOfClass:[NSDictionary class]]) {
9                          Class cls = meta->_genericCls;
10                         if (meta->_hasCustomClassFromDictionary) {
11                             cls = [cls modelCustomClassForDictionary:oneValue];
12                             if (!cls) cls = meta->_genericCls; // for xcode
13                         }
14                         NSObject *newOne = [cls new];
15                         [newOne modelSetWithDictionary:(id)oneValue];
16                         if (newOne) dic[oneKey] = newOne;
17                     }
18                 }];
19                 ((void (*)(id, SEL, id))(void *) objc_msgSend)((id)model, meta->_setter, dic);
20             } else {
21                 if (meta->_nsType == YYEncodingTypeNSDictionary) {

```

```

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     }
21 }

```



```

18         }
19     }];
20     genericMapper = tmp;
21 }
22 }

```

根据propertyName试着查找业务有没有指定的class，而且yymodel还支持string to class的转换，如果有就会先缓存下来，之后进行处理

业务定制，优先级极高，定制的类型后面不会再进行修改，而非定制的类型后面还需要继续推断

- 这里要提到一个小细节，可以看到在给object赋值的时候，也用到了generic，这里很有可能是yymodel的一个兼容性设置，也就是如果modelContainerPropertyGenericClass错误的写了非集合类型的class，yymodel也能正确的给这个属性进行赋值

1.3.2 协议

代码块

```

1      // support pseudo generic class with protocol name
2      if (!generic && propertyInfo.protocols) {
3          for (NSString *protocol in propertyInfo.protocols) {
4              Class cls = objc_getClass(protocol.UTF8String);
5              if (cls) {
6                  generic = cls;
7                  break;
8              }
9          }
10     }

```

如果业务没有指定属性的类型，并且在解析属性的时候有解析到协议，找到第一个解析的协议，给generic赋值

运行时判断有点 trick：

- objc_getClass(protocol.UTF8String) 通常只会得到类名和协议名一样的 Class。
- 这在 iOS 项目实际中很少这样写，所以协议补救只是个“保底手段”

1.3.3 局限性

可以看到yymodel在处理的时候是带有一些局限性的，比如

- NSArray<NSArray<User *> *> *这样的嵌套类型是无法支持的，因为协议方法只能定制class，这里嵌套类型的本质还是NSArray
- 无法转换协议类型，比如id<SomeProtocol>，赋值之后将会是nil

1.3.4 对于业务使用的思考

generic在多处赋值都是优先级极高的，业务方可以通过尽可能的指定准确的Class来帮助SDK判断类型，这样在赋值的时候就可以跳过一些判断逻辑

1.4 功能性

上篇文章主要分析了会调用modelSetWithDictionary相关的方法，这也是yymodel的核心功能，接下来分析一些其它的方法

1.4.1 NSObject实用方法

实现了NSObject的一些常用的方法，像isEqual、hash、description，不过都用了不同的命名而不是重写，不入侵业务

代码块

```
1  - (NSUInteger)modelHash {
2      if (self == (id)kCFNull) return [self hash];
3      _YYModelMeta *modelMeta = [_YYModelMeta metaWithClass:self.class];
4      if (modelMeta->_nsType) return [self hash];
5
6      NSUInteger value = 0;
7      NSUInteger count = 0;
8      for (_YYModelPropertyMeta *propertyMeta in modelMeta->_allPropertyMetas) {
9          if (!propertyMeta->_isKVCCCompatible) continue;
10         value ^= [[self valueForKey:NSStringFromSelector(propertyMeta-
11 >_getter))] hash];
12         count++;
13     }
14     if (count == 0) value = (long)((__bridge void *)self);
15     return value;
16 }
17 - (BOOL)modelIsEqual:(id)model {
18     if (self == model) return YES;
19     if (![model isKindOfClass:self.class]) return NO;
20     _YYModelMeta *modelMeta = [_YYModelMeta metaWithClass:self.class];
21     if (modelMeta->_nsType) return [self isEqual:model];
22     if ([self hash] != [model hash]) return NO;
23
24     for (_YYModelPropertyMeta *propertyMeta in modelMeta->_allPropertyMetas) {
25         if (!propertyMeta->_isKVCCCompatible) continue;
26         id this = [self valueForKey:NSStringFromSelector(propertyMeta-
27 >_getter)];
28         id that = [model valueForKey:NSStringFromSelector(propertyMeta-
29 >_getter)];
```

```

28         if (this == that) continue;
29         if (this == nil || that == nil) return NO;
30         if (![this isEqual:that]) return NO;
31     }
32     return YES;
33 }
34
35 - (NSString *)modelDescription {
36     return ModelDescription(self);
37 }

```

- modelHash: 如果是基础类型, 直接返回基础类型的hash, 如果不是根据isKVCCompatible尝试计算hash值
 - isKVCCompatible: 是在modelMeta初始化的时候, 根据属性类型进行赋值的, 具体在PropertyMetaInfo的初始化方法中
- isEqual: 先对比指针、判断是不是同一个类型、是不是基础类型、对比默认的hash实现, 如果都没有, 就通过对比isKVCCompatible的属性来判断相等
- modelDescription: 先判断是不是基础类型, 如果是会有一套针对基础类型的实现, 这里就不展开了; 如果不是会遍历model的属性打印

1.4.2 Encode和copy

无论是encode、initWithCoder、copy本质是就是在给属性赋值或者encode, 本质上还是操作属性

- 如果是Foundation基础类型, 调用默认实现
- 如果不是, 还是使用modelMeta, 根据属性的类型进行对应的操作 (encode、kvc、objc_msgSend)
 - 基础类型 (如 int、float、BOOL) : 直接 encode/decode/copy。
 - 对象类型: 优先通过 getter/setter (objc_msgSend) 直接读写; 若遇到结构体、C 数组等特殊类型, 则降级用 KVC (setValue:forKey:)。

1.4.3 Model To Json

代码块

```

1  - (id)modelToJSONObject {
2      id jsonObject = ModelToJSONObjectRecursive(self);
3      if ([jsonObject isKindOfClass:[NSArray class]]) return jsonObject;
4      if ([jsonObject isKindOfClass:[NSDictionary class]]) return jsonObject;
5      return nil;
6  }
7
8  - (NSData *)modelToJSONData {

```

```

9      id jsonObject = [self modelToJSONObject];
10     if (!jsonObject) return nil;
11     return [NSJSONSerialization dataWithJSONObject:jsonObject options:0
        error:NULL];
12 }
13
14 - (NSString *)modelToJSONString {
15     NSData *jsonData = [self modelToJSONData];
16     if (jsonData.length == 0) return nil;
17     return [[NSString alloc] initWithData:jsonData
        encoding:NSUTF8StringEncoding];
18 }

```

- 最关键的方法只有一个，ModelToJSONObjectRecursive

先来看这个方法，理解一下在Objective-C里，什么是JSON

代码块

```

1  /* Returns YES if the given object can be converted to JSON data, NO
   otherwise. The object must have the following properties:
2      - Top level object is an NSArray or NSDictionary
3      - All objects are NSString, NSNumber, NSArray, NSDictionary, or NSNull
4      - All dictionary keys are NSStrings
5      - NSNumbers are not NaN or infinity
6  Other rules may apply. Calling this method or attempting a conversion are the
   definitive ways to tell if a given object can be converted to JSON data.
7  */
8  + (BOOL)isValidJSONObject:(id)obj;

```

- **顶层对象：**必须是 NSArray 或 NSDictionary
- **支持的类型：**
 - 只能包含：NSString、NSNumber、NSArray、NSDictionary、NSNull。
 - 所有子对象（递归下去所有层级）都只能是这五类之一。
- **字典的 key：**必须是 NSString。不能是数字、对象等其他类型。
- **NSNumber 限制：**
 - 不能是 NaN、正无穷或负无穷。
 - 也就是说，@(NaN)、@(INFINITY)、@(-INFINITY) 都会判为不合法。
- **其他规则：**不能有自定义对象、NSDate、NSData 等类型。

基础类型转换

```

代码块 1 if (!model || model == (id)kCFNull) return model;
2 if ([model isKindOfClass:[NSString class]]) return model;
3 if ([model isKindOfClass:[NSNumber class]]) return model;
4 if ([model isKindOfClass:[NSDictionary class]]) {
5     if ([NSJSONSerialization isValidJSONObject:model]) return model;
6     NSMutableDictionary *newDic = [NSMutableDictionary new];
7     [((NSDictionary *)model) enumerateKeysAndObjectsUsingBlock:^(NSString
*key, id obj, BOOL *stop) {
8         NSString *stringKey = [key isKindOfClass:[NSString class]] ? key :
key.description;
9         if (!stringKey) return;
10        id jsonObj = ModelToJSONObjectRecursive(obj);
11        if (!jsonObj) jsonObj = (id)kCFNull;
12        newDic[stringKey] = jsonObj;
13    }];
14    return newDic;
15 }
16 if ([model isKindOfClass:[NSSet class]]) {
17     NSArray *array = ((NSSet *)model).allObjects;
18     if ([NSJSONSerialization isValidJSONObject:array]) return array;
19     NSMutableArray *newArray = [NSMutableArray new];
20     for (id obj in array) {
21         if ([obj isKindOfClass:[NSString class]] || [obj isKindOfClass:
[NSNumber class]]) {
22             [newArray addObject:obj];
23         } else {
24             id jsonObj = ModelToJSONObjectRecursive(obj);
25             if (jsonObj && jsonObj != (id)kCFNull) [newArray
addObject:jsonObj];
26         }
27     }
28     return newArray;
29 }
30 if ([model isKindOfClass:[NSArray class]]) {
31     if ([NSJSONSerialization isValidJSONObject:model]) return model;
32     NSMutableArray *newArray = [NSMutableArray new];
33     for (id obj in (NSArray *)model) {
34         if ([obj isKindOfClass:[NSString class]] || [obj isKindOfClass:
[NSNumber class]]) {
35             [newArray addObject:obj];
36         } else {
37             id jsonObj = ModelToJSONObjectRecursive(obj);
38             if (jsonObj && jsonObj != (id)kCFNull) [newArray
addObject:jsonObj];
39         }
40     }
41     return newArray;

```

```

42     }
43     if ([model isKindOfClass:[NSURL class]]) return ((NSURL
*)model).absoluteString;
44     if ([model isKindOfClass:[NSAttributedString class]]) return
((NSAttributedString *)model).string;
45     if ([model isKindOfClass:[NSDate class]]) return [YYISODateFormatter()
stringFromDate:(id)model];
46     if ([model isKindOfClass:[NSData class]]) return nil;

```

- 对于基础类型的转换，分为三种：
 - 自己就是json对象的，直接返回
 - NSDate类型特殊处理
 - 判断符不符合JSON要求，符合的直接返回；如果不符合，就通过 ModelToJSONObjectRecursive把其中不是json类型的数据转换成json object

对于非基础类型的转换

代码块

```

1     _YYModelMeta *modelMeta = [_YYModelMeta metaWithClass:[model class]];
2     if (!modelMeta || modelMeta->_keyMappedCount == 0) return nil;
3     NSMutableDictionary *result = [[NSMutableDictionary alloc]
initWithCapacity:64];
4     __unsafe_unretained NSMutableDictionary *dic = result; // avoid retain and
release in block
5     [modelMeta->_mapper enumerateKeysAndObjectsUsingBlock:^(NSString
*propertyMappedKey, _YYModelPropertyMeta *propertyMeta, BOOL *stop) {
6         if (!propertyMeta->_getter) return;
7
8         id value = nil;
9         if (propertyMeta->_isCNumber) {
10             value = ModelCreateNumberFromProperty(model, propertyMeta);
11         } else if (propertyMeta->_nsType) {
12             id v = ((id (*)(id, SEL))(void *) objc_msgSend)((id)model,
propertyMeta->_getter);
13             value = ModelToJSONObjectRecursive(v);
14         } else {
15             switch (propertyMeta->_type & YYEncodingTypeMask) {
16                 case YYEncodingTypeObject: {
17                     id v = ((id (*)(id, SEL))(void *) objc_msgSend)((id)model,
propertyMeta->_getter);
18                     value = ModelToJSONObjectRecursive(v);
19                     if (value == (id)kCFNull) value = nil;
20                 } break;
21                 case YYEncodingTypeClass: {

```

```

22         Class v = ((Class (*)(id, SEL))(void *) objc_msgSend)
((id)model, propertyMeta->_getter);
23         value = v ? NSStringFromClass(v) : nil;
24     } break;
25     case YYEncodingTypeSEL: {
26         SEL v = ((SEL (*)(id, SEL))(void *) objc_msgSend)
((id)model, propertyMeta->_getter);
27         value = v ? NSStringFromSelector(v) : nil;
28     } break;
29     default: break;
30 }
31 }
32 if (!value) return;
33
34 if (propertyMeta->_mappedToKeyPath) {
35     NSMutableDictionary *superDic = dic;
36     NSMutableDictionary *subDic = nil;
37     for (NSUInteger i = 0, max = propertyMeta->_mappedToKeyPath.count;
i < max; i++) {
38         NSString *key = propertyMeta->_mappedToKeyPath[i];
39         if (i + 1 == max) { // end
40             if (!superDic[key]) superDic[key] = value;
41             break;
42         }
43
44         subDic = superDic[key];
45         if (subDic) {
46             if ([subDic isKindOfClass:[NSDictionary class]]) {
47                 subDic = subDic.mutableCopy;
48                 superDic[key] = subDic;
49             } else {
50                 break;
51             }
52         } else {
53             subDic = [NSMutableDictionary new];
54             superDic[key] = subDic;
55         }
56         superDic = subDic;
57         subDic = nil;
58     }
59 } else {
60     if (!dic[propertyMeta->_mappedToKey]) {
61         dic[propertyMeta->_mappedToKey] = value;
62     }
63 }
64 }];
65

```

```

66     if (modelMeta->_hasCustomTransformToDictionary) {
67         BOOL suc = [((id<YYModel>)model) modelCustomTransformToDictionary:dic];
68         if (!suc) return nil;
69     }
70     return result;

```

- 性能优化：把__strong类型的对象在block持有之前使用__unsafe_unretained持有，避免arc的自动retain/release
- 遍历modelMeta中的属性：把value转换成符合json要求的对象
- mappedToKeyPath提供了多级映射的能力：
 - 比如一个属性 @property NSString *provinceName; 可以映射到字典的 @"address.province" 字段（这样就不是一级 key，而是多级）。
 - 这时 _mappedToKeyPath 就是一个数组：@[@"address", @"province"]。
 - 依次遍历 keyPath，每一级都创建（或拿到已有的）嵌套字典，直到最后一级，把 value 赋值。
 - 这段代码递归/循环地保证了“多级嵌套”的字典结构都被正确生成。
 - 如果只需要一级 key，则直接 dic[propertyMeta->_mappedToKey] = value。

代码块

```

1     if (modelMeta->_hasCustomTransformToDictionary) {
2         BOOL suc = [((id<YYModel>)model) modelCustomTransformToDictionary:dic];
3         if (!suc) return nil;
4     }

```

- 这是给 **业务方** 一个“最后自定义修改”字典的机会。
- 如果模型实现了 `modelCustomTransformToDictionary:` 方法，YYModel 会在所有属性都处理完后**最后一步**调用它。
- 比如，你想调整部分值、补充数据、转换格式，都可以在这里做。
- 如果返回 `NO`，整个序列化就被判定失败，返回 `nil`。

2. 业务思考

- 1.2.4小节已经分析过，对于集合类型，最好指定class帮助yymodel快速判断类型，优化性能
- yymodel会在第一次解析的时候缓存metaClass和classInfo，后续会直接使用缓存，所以如果需要runtime反射功能的class使用的时候要小心，如果动态添加属性发生在解析model之后，再次解析model将不会有这个动态添加的属性

3. 线程安全：所有数据包括modelMeta, classInfo等都使用dispatch_once+dispatch_semaphore_t保证了线程安全，只写入一次
 - a. 也就是说虽然缓存是存储在数据段的，整个进程共享，但是因为只会写入一次，所以线程安全
4. 性能：yymodel的思路是可以借鉴的，比如寻梦记账也有一些硬编码数组，这些数组的创建也可以使用dispatch_once，而且还可以在第一次创建的时候，使用NSDictionary建立id to model的映射，同样缓存这个dictionary，之后在获取这个model的时候就可以O(1)获取了
5. 防御式编程：yymodel还是缺少断言，个人认为在某些地方还是应该添加一些合理的断言，以此帮助业务在debug环境提前发现问题

参考

[YYModel作者对于不同JSON模型转换差异分析](#)