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字对于性能、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 容错性分析

容错性也可以理解为全面性，支不支持不同数据类型之间的转换，比如使用NSString给NSDate赋值，能不能正常赋值；

体现容错性的主要方法是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

```
{
    if (meta->_isCNumber) {
        NSNumber *num = YYNSNumberCreateFromID(value);
        ModelSetNumberToProperty(model, num, meta);
        if (num != nil) [num class]; // hold the number
    }
```

```objc
}

static force_inline NSNumber *YYNSNumberCreateFromID(__unsafe_unretained id value) {
    static NSCharacterSet *dot;
    static NSDictionary *dic;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        dot = [NSCharacterSet characterSetWithRange:NSMakeRange('.', 1)];
        dic = @{@"TRUE" :   @(YES),
                @"True" :   @(YES),
                @"true" :   @(YES),
                @"FALSE" :  @(NO),
                @"False" :  @(NO),
                @"false" :  @(NO),
                @"YES" :    @(YES),
                @"Yes" :    @(YES),
                @"yes" :    @(YES),
                @"NO" :     @(NO),
                @"No" :     @(NO),
                @"no" :     @(NO),
                @"NIL" :    (id)kCFNull,
                @"Nil" :    (id)kCFNull,
                @"nil" :    (id)kCFNull,
                @"NULL" :   (id)kCFNull,
                @"Null" :   (id)kCFNull,
                @"null" :   (id)kCFNull,
                @"(NULL)" : (id)kCFNull,
                @"(Null)" : (id)kCFNull,
                @"(null)" : (id)kCFNull,
                @"<NULL>" : (id)kCFNull,
                @"<Null>" : (id)kCFNull,
                @"<null>" : (id)kCFNull};
    });

    if (!value || value == (id)kCFNull) return nil;
    if ([value isKindOfClass:[NSNumber class]]) return value;
    if ([value isKindOfClass:[NSString class]]) {
        NSNumber *num = dic[value];
        if (num != nil) {
            if (num == (id)kCFNull) return nil;
            return num;
        }
        if ([(NSString *)value rangeOfCharacterFromSet:dot].location != NSNotFound) {
            const char *cstring = ((NSString *)value).UTF8String;
            if (!cstring) return nil;
```

```
52              double num = atof(cstring);
53              if (isnan(num) || isinf(num)) return nil;
54              return @(num);
55          } else {
56              const char *cstring = ((NSString *)value).UTF8String;
57              if (!cstring) return nil;
58              return @(atoll(cstring));
59          }
60      }
61      return nil;
62  }
```

- 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、NSValue、NSData、NSURL、NSValue、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

```
{
        BOOL isNull = (value == (id)kCFNull);
        switch (meta->_type & YYEncodingTypeMask) {
            case YYEncodingTypeObject: {
                Class cls = meta->_genericCls ?: meta->_cls;
                if (isNull) {
                    ((void (*)(id, SEL, id))(void *) objc_msgSend)((id)model, meta->_setter, (id)nil);
                } else if ([value isKindOfClass:cls] || !cls) {
                    ((void (*)(id, SEL, id))(void *) objc_msgSend)((id)model, meta->_setter, (id)value);
                } else if ([value isKindOfClass:[NSDictionary class]]) {
                    NSObject *one = nil;
                    if (meta->_getter) {
                        one = ((id (*)(id, SEL))(void *) objc_msgSend)((id)model, meta->_getter);
                    }
                    if (one) {
                        [one modelSetWithDictionary:value];
                    } else {
                        if (meta->_hasCustomClassFromDictionary) {
                            cls = [cls modelCustomClassForDictionary:value] ?: cls;
                        }
                        one = [cls new];
                        [one modelSetWithDictionary:value];
                        ((void (*)(id, SEL, id))(void *) objc_msgSend)((id)model, meta->_setter, (id)one);
                    }
                }
            } break;
            case YYEncodingTypeClass: {
                if (isNull) {
                    ((void (*)(id, SEL, Class))(void *) objc_msgSend)((id)model, meta->_setter, (Class)NULL);
                } else {
                    Class cls = nil;
                    if ([value isKindOfClass:[NSString class]]) {
```

```
                        cls = NSClassFromString(value);
                        if (cls) {
                            ((void (*)(id, SEL, Class))(void *) objc_msgSend)((id)model, meta->_setter, (Class)cls);
                        }
                    } else {
                        cls = object_getClass(value);
                        if (cls) {
                            if (class_isMetaClass(cls)) {
                                ((void (*)(id, SEL, Class))(void *) objc_msgSend)((id)model, meta->_setter, (Class)value);
                            }
                        }
                    }
                }
            } break;
            default: break;
        }
    }
```

- 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

1.2 泛型支持

代码块

```
{
    case YYEncodingTypeNSDictionary:
    case YYEncodingTypeNSMutableDictionary: {
        if ([value isKindOfClass:[NSDictionary class]]) {
            if (meta->_genericCls) {
                NSMutableDictionary *dic = [NSMutableDictionary new];
```

```
                [((NSDictionary *)value)
enumerateKeysAndObjectsUsingBlock:^(NSString *oneKey, id oneValue, BOOL *stop)
{
                    if ([oneValue isKindOfClass:[NSDictionary class]]) {
                        Class cls = meta->_genericCls;
                        if (meta->_hasCustomClassFromDictionary) {
                            cls = [cls modelCustomClassForDictionary:oneValue];
                            if (!cls) cls = meta->_genericCls; // for xcode
code coverage
                        }
                        NSObject *newOne = [cls new];
                        [newOne modelSetWithDictionary:(id)oneValue];
                        if (newOne) dic[oneKey] = newOne;
                    }
                }];
                ((void (*)(id, SEL, id))(void *) objc_msgSend)((id)model, meta-
>_setter, dic);
            } else {
                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);
                }
            }
        }
    } break;
}
```

这样的代码在YYModel中几乎随处可见，所以genericCls到底是什么呢?

1.2.1 业务定制

代码块

```
+ (nullable NSDictionary<NSString *, id> *)modelContainerPropertyGenericClass;
```

业务可以为集合类型属性订制它的类型，这样在赋值时就不用推测和判断，可以直接赋值

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;
        }
    }
```

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

1.2.2 协议

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”

1.2.3 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

1.2.4 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

1.3 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

1.3.1 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->_isKVCCompatible) continue;
10          value ^= [[self valueForKey:NSStringFromSelector(propertyMeta-
    >_getter)] hash];
11          count++;
12      }
```

```
    if (count == 0) value = (long)((__bridge void *)self);
    return value;
}

- (BOOL)modelIsEqual:(id)model {
    if (self == model) return YES;
    if (![model isMemberOfClass:self.class]) return NO;
    _YYModelMeta *modelMeta = [_YYModelMeta metaWithClass:self.class];
    if (modelMeta->_nsType) return [self isEqual:model];
    if ([self hash] != [model hash]) return NO;

    for (_YYModelPropertyMeta *propertyMeta in modelMeta->_allPropertyMetas) {
        if (!propertyMeta->_isKVCCompatible) continue;
        id this = [self valueForKey:NSStringFromSelector(propertyMeta->_getter)];
        id that = [model valueForKey:NSStringFromSelector(propertyMeta->_getter)];
        if (this == that) continue;
        if (this == nil || that == nil) return NO;
        if (![this isEqual:that]) return NO;
    }
    return YES;
}

- (NSString *)modelDescription {
    return ModelDescription(self);
}
```

- 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

1.3.2 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:）。

1.3.3 Model To Json

代码块

```
- (id)modelToJSONObject {
    id jsonObject = ModelToJSONObjectRecursive(self);
    if ([jsonObject isKindOfClass:[NSArray class]]) return jsonObject;
    if ([jsonObject isKindOfClass:[NSDictionary class]]) return jsonObject;
    return nil;
}

- (NSData *)modelToJSONData {
    id jsonObject = [self modelToJSONObject];
    if (!jsonObject) return nil;
    return [NSJSONSerialization dataWithJSONObject:jsonObject options:0
error:NULL];
}

- (NSString *)modelToJSONString {
    NSData *jsonData = [self modelToJSONData];
    if (jsonData.length == 0) return nil;
    return [[NSString alloc] initWithData:jsonData
encoding:NSUTF8StringEncoding];
}
```

- 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

```
/* Returns YES if the given object can be converted to JSON data, NO
otherwise. The object must have the following properties:
    - Top level object is an NSArray or NSDictionary
    - All objects are NSString, NSNumber, NSArray, NSDictionary, or NSNull
    - All dictionary keys are NSStrings
    - NSNumbers are not NaN or infinity
 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.
 */
+ (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

```
    if (!model || model == (id)kCFNull) return model;
    if ([model isKindOfClass:[NSString class]]) return model;
    if ([model isKindOfClass:[NSNumber class]]) return model;
    if ([model isKindOfClass:[NSDictionary class]]) {
        if ([NSJSONSerialization isValidJSONObject:model]) return model;
        NSMutableDictionary *newDic = [NSMutableDictionary new];
        [((NSDictionary *)model) enumerateKeysAndObjectsUsingBlock:^(NSString *key, id obj, BOOL *stop) {
            NSString *stringKey = [key isKindOfClass:[NSString class]] ? key : key.description;
            if (!stringKey) return;
            id jsonObj = ModelToJSONObjectRecursive(obj);
            if (!jsonObj) jsonObj = (id)kCFNull;
            newDic[stringKey] = jsonObj;
        }];
        return newDic;
    }
    if ([model isKindOfClass:[NSSet class]]) {
        NSArray *array = ((NSSet *)model).allObjects;
        if ([NSJSONSerialization isValidJSONObject:array]) return array;
        NSMutableArray *newArray = [NSMutableArray new];
        for (id obj in array) {
            if ([obj isKindOfClass:[NSString class]] || [obj isKindOfClass:[NSNumber class]]) {
                [newArray addObject:obj];
            } else {
                id jsonObj = ModelToJSONObjectRecursive(obj);
                if (jsonObj && jsonObj != (id)kCFNull) [newArray addObject:jsonObj];
```

```
            }
        }
        return newArray;
    }
    if ([model isKindOfClass:[NSArray class]]) {
        if ([NSJSONSerialization isValidJSONObject:model]) return model;
        NSMutableArray *newArray = [NSMutableArray new];
        for (id obj in (NSArray *)model) {
            if ([obj isKindOfClass:[NSString class]] || [obj isKindOfClass:
[NSNumber class]]) {
                [newArray addObject:obj];
            } else {
                id jsonObj = ModelToJSONObjectRecursive(obj);
                if (jsonObj && jsonObj != (id)kCFNull) [newArray
addObject:jsonObj];
            }
        }
        return newArray;
    }
    if ([model isKindOfClass:[NSURL class]]) return ((NSURL
*)model).absoluteString;
    if ([model isKindOfClass:[NSAttributedString class]]) return
((NSAttributedString *)model).string;
    if ([model isKindOfClass:[NSDate class]]) return [YYISODateFormatter()
stringFromDate:(id)model];
    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

```
    _YYModelMeta *modelMeta = [_YYModelMeta metaWithClass:[model class]];
    if (!modelMeta || modelMeta->_keyMappedCount == 0) return nil;
    NSMutableDictionary *result = [[NSMutableDictionary alloc]
initWithCapacity:64];
    __unsafe_unretained NSMutableDictionary *dic = result; // avoid retain and
release in block
    [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;
```

```
                        superDic[key] = subDic;
                    } else {
                        break;
                    }
                } else {
                    subDic = [NSMutableDictionary new];
                    superDic[key] = subDic;
                }
                superDic = subDic;
                subDic = nil;
            }
        } else {
            if (!dic[propertyMeta->_mappedToKey]) {
                dic[propertyMeta->_mappedToKey] = value;
            }
        }
    }];

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

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

- 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. 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
2. 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：

参考

YYModel作者对于不同JSON模型转换差异分析