我们在 JSON <-> Dictionary <-> Model 中面临的一个很大的问题就是判断数据需要转换成什么样的类型。好在 ObjC 作为一款动态语言，利用 runtime 可以轻松解决这个问题。再配合转换器和 KVC，就可以轻松把我们解析好的值放进对应 Model 里。今天要给大家介绍的就是这个类型编码（Type Encodings）的具体细节。

ObjC 的 type encodings 列表

编码	意义
c	char 类型
i	int 类型
s	short 类型
l	long 类型，仅用在 32-bit 设备上
q	long long 类型
C	unsigned char 类型
I	unsigned int 类型
S	unsigned short 类型
L	unsigned long 类型
Q	unsigned long long 类型
f	float 类型
d	double 类型，long double 不被 ObjC 支持，所以也是指向此编码
B	bool 或 _Bool 类型
v	void 类型
*	C 字串（char *）类型
@	对象（id）类型
#	Class 类型

:	SEL 类型
[array type]	C 数组类型（注意这不是 NSArray）
{name=type...}	结构体类型
(name=type...)	联合体类型
bnum	位段（bit field）类型用 b 表示，num 表示字节数，这个类型很少用
^type	一个指向 type 类型的指针类型
?	未知类型

C 语言基础数据类型的 type encodings

整型和浮点型数据

简单给大家举个例子，我们先来看看常用的数值类型，用下面的代码来打印日志：

```
NSLog(@"char     : %s, %lu", @encode(char), sizeof(char));
NSLog(@"short    : %s, %lu", @encode(short), sizeof(short));
NSLog(@"int      : %s, %lu", @encode(int), sizeof(int));
NSLog(@"long     : %s, %lu", @encode(long), sizeof(long));
NSLog(@"long long: %s, %lu", @encode(long long), sizeof(long
long));
NSLog(@"float    : %s, %lu", @encode(float), sizeof(float));
NSLog(@"double   : %s, %lu", @encode(double), sizeof(double));
NSLog(@"NSInteger: %s, %lu", @encode(NSInteger),
sizeof(NSInteger));
NSLog(@"CGFloat  : %s, %lu", @encode(CGFloat), sizeof(CGFloat));
NSLog(@"int32_t  : %s, %lu", @encode(int32_t), sizeof(int32_t));
NSLog(@"int64_t  : %s, %lu", @encode(int64_t), sizeof(int64_t));
```

在 32-bit 设备上输出日志如下：

```
char     : c, 1
short    : s, 2
int      : i, 4
long     : l, 4
long long: q, 8
float    : f, 4
double   : d, 8
NSInteger: i, 4
CGFloat  : f, 4
int32_t  : i, 4
int64_t  : q, 8
```

大家注意下上面日志里的 `long` 类型输出结果，然后我们再看下在 64-bit 设备上的输出日志：

```
char     : c, 1
short    : s, 2
int      : i, 4
long     : q, 8
long long: q, 8
float    : f, 4
double   : d, 8
NSInteger: q, 8
CGFloat  : d, 8
int32_t  : i, 4
int64_t  : q, 8
```

可以看到 `long` 的长度变成了 8，而且类型编码也变成 `q`，这就是表格里那段话的意思。

所以呢，一般如果想要整形的长度固定且长度能被一眼看出，建议使用例子最后的 `int32_t` 和 `int64_t`，尽量少去使用 `long` 类型。

然后要提一下 `NSInteger` 和 `CGFloat`，这俩都是针对不同 CPU 分开定义的：

```
#if __LP64__ || (TARGET_OS_EMBEDDED && !TARGET_OS_IPHONE) ||
TARGET_OS_WIN32 || NS_BUILD_32_LIKE_64
typedef long NSInteger;
```

```
#else
typedef int NSInteger;
#endif

#if defined(__LP64__) && __LP64__
# define CGFLOAT_TYPE double
#else
# define CGFLOAT_TYPE float
#endif
typedef CGFLOAT_TYPE CGFloat;
```

所以他们在 32-bit 设备上长度为 4，在 64-bit 设备上长度为 8，对应类型编码也会有变化。

布尔数据

用下面的代码打印日志：

```
NSLog(@"bool     : %s, %lu", @encode(bool), sizeof(bool));
NSLog(@"_Bool    : %s, %lu", @encode(_Bool), sizeof(_Bool));
NSLog(@"BOOL     : %s, %lu", @encode(BOOL), sizeof(BOOL));
NSLog(@"Boolean  : %s, %lu", @encode(Boolean), sizeof(Boolean));
NSLog(@"boolean_t: %s, %lu", @encode(boolean_t),
sizeof(boolean_t));
```

在 32-bit 设备上输出日志如下：

```
bool     : B, 1
_Bool    : B, 1
BOOL     : c, 1
Boolean  : C, 1
boolean_t: i, 4
```

在 64-bit 设备上输出日志如下：

```
bool     : B, 1
_Bool    : B, 1
```

```
BOOL     : B, 1
Boolean  : C, 1
boolean_t: I, 4
```

可以看到我们最常用的 `BOOL` 类型还真的是有点妖，这个妖一句两句还说不清楚，我在下一篇博客里会介绍一下。在本篇博客里，这个变化倒是对我们解析模型不会产生很大的影响，所以先略过。

void、指针和数组

用下面的代码打印日志：

```
NSLog(@"void    : %s, %lu", @encode(void), sizeof(void));
NSLog(@"char *  : %s, %lu", @encode(char *), sizeof(char *));
NSLog(@"short * : %s, %lu", @encode(short *), sizeof(short *));
NSLog(@"int *   : %s, %lu", @encode(int *), sizeof(int *));
NSLog(@"char[3] : %s, %lu", @encode(char[3]), sizeof(char[3]));
NSLog(@"short[3]: %s, %lu", @encode(short[3]), sizeof(short[3]));
NSLog(@"int[3]  : %s, %lu", @encode(int[3]), sizeof(int[3]));
```

在 64-bit 设备上输出日志如下：

```
void    : v, 1
char *  : *, 8
short * : ^s, 8
int *   : ^i, 8
char[3] : [3c], 3
short[3]: [3s], 6
int[3]  : [3i], 12
```

在 32-bit 设备上指针类型的长度会变成 4，这个就不多介绍了。

可以看到只有 C 字串类型比较特殊，会处理成 `*` 编码，其它整形数据的指针类型还是正常处理的。

结构体和联合体

用下面的代码打印日志：

```
NSLog(@"CGSize: %s, %lu", @encode(CGSize), sizeof(CGSize));
```

在 64-bit 设备上输出日志如下：

```
CGSize: {CGSize=dd}, 16
```

因为 `CGSize` 内部的字段都是 `CGFloat` 的，在 64-bit 设备上实际是 `double` 类型，所以等于号后面是两个 `d` 编码，总长度是 16。

联合体的编码格式十分类似，不多赘述。而位段现在用到的十分少，也不介绍了，有兴趣了解位段的可以参考维基百科。

ObjC 数据类型的 type encodings

ObjC 数据类型大部分情况下要配合 runtime 使用，单独用 `@encode` 操作符的话，基本上也就能做到下面这些：

```
NSLog(@"Class    : %s", @encode(Class));
NSLog(@"NSObject: %s", @encode(NSObject));
NSLog(@"NSString: %s", @encode(NSString));
NSLog(@"id       : %s", @encode(id));
NSLog(@"Selector: %s", @encode(SEL));
```

输出日志：

```
Class    : #
NSObject: {NSObject=#}
NSString: {NSString=#}
id       : @
Selector: :
```

可以看到对象的类名称的编码方式跟结构体相似，等于号后面那个 `#` 就是 `isa` 指针了，是一个 `Class` 类型的数据。

类属性和成员变量的 type encodings

我们可以用 runtime 去获得类的属性对应的 type encoding：

```
objc_property_t property = class_getProperty([NSObject class],
"description");
if (property) {
    NSLog(@"%s - %s", property_getName(property),
property_getAttributes(property));
} else {
    NSLog(@"not found");
}
```

我们会获得这么一段输出：

```
description - T@"NSString",R,C
```

这里的 `R` 表示 `readonly`，`C` 表示 `copy`，这都是属性的修饰词，不过在本篇先不多介绍。

主要要说的是这里的 `T`，也就是 `type`，后面跟的这段 `@"NSString"` 就是 type encoding 了。可以看到 runtime 比较贴心的用双引号的方式告诉了我们这个对象的实际类型是什么。

关于属性的修饰词，更多内容可以参考 Apple 文档。其中 `T` 段始终会是第一个 attribute，所以处理起来会简单点。

而如果是成员变量的话，我们可以用类似下面的办法去获得 type encoding：

```
@interface TestObject : NSObject {
    int testInt;
    NSString *testStr;
}
@end
```

```
Ivar ivar = class_getInstanceVariable([TestObject class],
"testInt");
if (ivar) {
    NSLog(@"%s - %s", ivar_getName(ivar),
ivar_getTypeEncoding(ivar));
} else {
    NSLog(@"not found");
}
ivar = class_getInstanceVariable([TestObject class], "testStr");
if (ivar) {
    NSLog(@"%s - %s", ivar_getName(ivar),
ivar_getTypeEncoding(ivar));
} else {
    NSLog(@"not found");
}
```

获得的输出会是这样：

```
testInt - i
testStr - @"NSString"
```

因为成员变量没有属性修饰词那些，所以直接获得的就是 type encoding，格式和属性的 `T` attribute 一样。

类方法的 type encoding

有的时候模型设置数据的方式并不是用属性的方式，而是用方法的方式。我们举个例子：

```
Method method = class_getInstanceMethod([UIView class],
@selector(setFrame:));
if (method) {
    NSLog(@"%@ - %s", NSStringFromSelector(method_getName(method)),
method_getTypeEncoding(method));
} else {
    NSLog(@"not found");
}
```

可以获得输出：

```
setFrame: - v48@0:8{CGRect={CGPoint=dd}{CGSize=dd}}16
```

输出就是整个类方法的 type encoding，关于这个我没找到官方文档的介绍，所以只能根据自己的推测来介绍这个编码的格式：

- 第一个字符 `v` 是表示函数的返回值是 void 类型
- 后续的 `48` 表示函数参数表的长度（指返回值之后的所有参数，虽然返回值在 runtime 里也算是个参数）
- 后续的 `@` 表示一个对象，在 ObjC 里这里传递的是 `self`，实例方法是要传递实例对象给函数的
- 后续的 `0` 上面参数对应的 offset
- 后续的 `:` 表示一个 selector，用来指出要调用的函数是哪个
- 后续的 `8` 是 selector 参数的 offset，因为这是跑在 64-bit 设备上的，所以 `@` 和 `:` 的长度都是 8
- 后续的 `{CGRect={CGPoint=dd}{CGSize=dd}}` 是 CGRect 结构体的 type encoding，从这里也可以看出结构体嵌套使用时对应的 type encoding 是这种格式的，这个结构体包含 4 个 double 类型的数据，所以总长度应该是 32
- 最后的 `16` 是最后一个参数的 offset，加上刚刚的参数长度 32 正好是整个函数参数表的长度

我们拿另一个类方法来验证下：

```
Method method = class_getInstanceMethod([UIViewController class],
@selector(title));
if (method) {
    NSLog(@"%@ - %s", NSStringFromSelector(method_getName(method)),
method_getTypeEncoding(method));
} else {
    NSLog(@"not found");
}
```

输出：

```
@16@0:8
```

可以看到很可惜，NSString 类型在类方法的 type encoding 里是不会有引号内容的，所以我们只能知道这个参数是个 id 类型。编码的具体解析：

- `@` - 返回 id 类型
- `16` - 参数表总长度
- `@` - 用来传递 `self`，是 id 类型
- `0` - `self` 参数的 offset
- `:` - 传递具体要调用哪个方法，selector 类型
- `8` - selector 参数的 offset

如果是类的静态方法而不是实例方法，我们可以用类似这样的代码获得 Method 结构体：

```
Method method = class_getClassMethod([TestObject class],
@selector(testMethod));
```

不过说起来这种格式的编码还是不容易解析，所以我们可以用另一种方式直接拿对应位置的参数的编码：

```
Method method = class_getInstanceMethod([UIView class],
@selector(setFrame:));
if (method) {
    NSLog(@"%@ - %d", NSStringFromSelector(method_getName(method)),
method_getNumberOfArguments(method));
    NSLog(@"%@ - %s", NSStringFromSelector(method_getName(method)),
method_copyArgumentType(method, 2));
} else {
    NSLog(@"not found");
}
```

输出内容如下，这里是获得了 index 为 2 的参数的编码：

```
setFrame: - 3
setFrame: - {CGRect={CGPoint=dd}{CGSize=dd}}
```

这样就只会获得 type encoding 而不会带上 offset 信息，就容易解析多了。

另外从这里也可以看到，返回值其实也是算一个参数。

其它一些 type encodings 细节

还有些 type encodings 的细节和解析模型其实不太相关，不过也在这里介绍一下。

protocol 类型的 type encoding

用以下代码打印日志：

```
objc_property_t property = class_getProperty([UIScrollView class],
"delegate");
if (property) {
    NSLog(@"%s - %s", property_getName(property),
property_getAttributes(property));
} else {
    NSLog(@"not found");
}
```

会获得输出：

```
delegate - T@"<UIScrollViewDelegate>",W,N,V_delegate
```

可以看到在属性的 type encoding 里，会用双引号和尖括号表示出 protocol 的类型

但是去查看方法的话：

```
Method method = class_getInstanceMethod([UIScrollView class],
@selector(setDelegate:));
if (method) {
    NSLog(@"%@ - %d", NSStringFromSelector(method_getName(method)),
method_getNumberOfArguments(method));
    NSLog(@"%@ - %s", NSStringFromSelector(method_getName(method)),
```

```
method_copyArgumentType(method, 2));
} else {
    NSLog(@"not found");
}
```

依然还是只能得到这样的编码：

```
setDelegate: - 3
setDelegate: - @
```

protocol 类型在模型解析中并没有很大的指导作用，因为我们无法知道具体实现了 protocol 协议的 class 是什么。

block 类型的 type encoding

直接亮结果吧，获得的 type encoding 是 `@?`，没有任何参考意义，还好我们做模型解析用不到这个。

关于方法参数的内存对齐

对 `setEnable:` 方法取 type encoding 的话会得到：

```
setEnabled: - v20@0:8B16
```

可是 bool 的长度明明只有 1 啊，所以这是为什么呢？感兴趣的朋友可以了解下内存对齐。

总结

关于 Type Encodings，要讲的差不多就这么多了。暂时没有想到还有什么要补充的，后面想到了再补上来吧。

希望对大家有帮助，也欢迎大家指正错误或者进行讨论