一. RunTime简介

RunTime简称运行时。OC就是运行时机制，也就是在运行时候的一些机制，其中最主要的是消息机制。

对于C语言，函数的调用在编译的时候会决定调用哪个函数，如果调用未实现的函数就会报错。 对于OC语言，属于动态调用过程，在编译的时候并不能决定真正调用哪个函数，只有在真正运行的时候才会根据函数的名称找到对应的函数来调用。在编译阶段，OC可以调用任何函数，即使这个函数并未实现，只要声明过就不会报错。

二. RunTime消息机制

消息机制是运行时里面最重要的机制，OC中任何方法的调用，本质都是发送消息。使用运行时，发送消息需要导入框架`<objc/message.h>`并且xcode5之后，苹果不建议使用底层方法，如果想要使用运行时，需要关闭严格检查objc_msgSend的调用，BuildSetting->搜索msg 改为NO。

下来看一下实例方法调用底层实现

```
Person *p = [[Person alloc] init];
[p eat];
// 底层会转化成
//SEL：方法编号，根据方法编号就可以找到对应方法的实现。
[p performSelector:@selector(eat)];
//performSelector本质即为运行时，发送消息，谁做事情就调用谁
objc_msgSend(p, @selector(eat));
// 带参数
objc_msgSend(p, @selector(eat:),10);
```

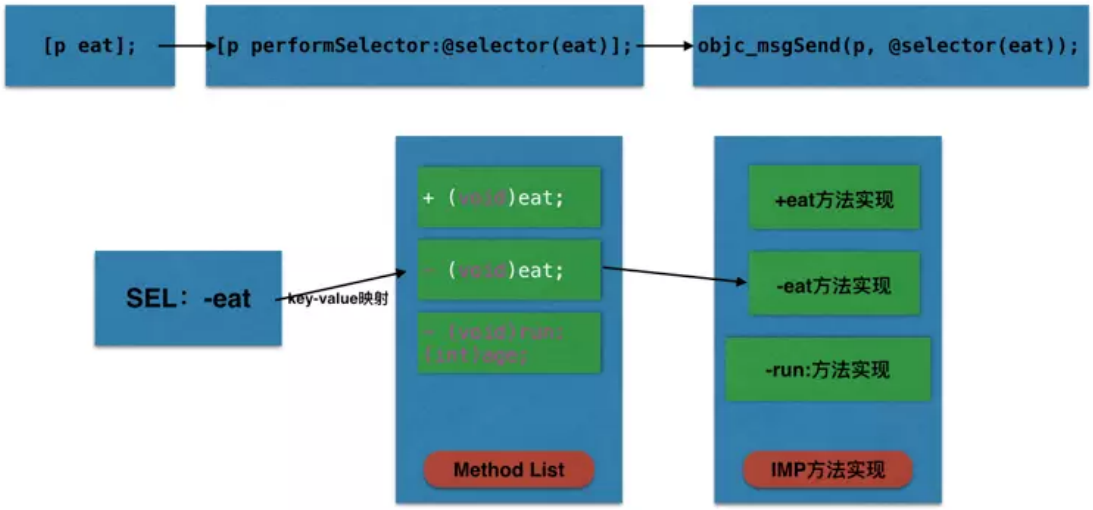
类方法的调用底层

```
// 本质是会将类名转化成类对象，初始化方法其实是在创建类对象。
[Person eat];
// Person只是表示一个类名，并不是一个真实的对象。只要是方法必须要对象去调用。
// RunTime 调用类方法同样，类方法也是类对象去调用，所以需要获取类对象，然后使用类对象去调用方法。
```

```
Class personclass = [Persion class];
[[Persion class] performSelector:@selector(eat)];
// 类对象发送消息
objc_msgSend(personclass, @selector(eat));
```

@selector (SEL)：是一个SEL方法选择器。SEL其主要作用是快速的通过方法名字查找到对应方法的函数指针，然后调用其函数。SEL其本身是一个Int类型的地址，地址中存放着方法的名字。 对于一个类中。每一个方法对应着一个SEL。所以一个类中不能存在2个名称相同的方法，即使参数类型不同，因为SEL是根据方法名字生成的，相同的方法名称只能对应一个SEL。

运行时发送消息的底层实现 每一个类都有一个方法列表 Method List，保存这类里面所有的方法，根据SEL传入的方法编号找到方法，相当于value - key的映射。然后找到方法的实现。去方法的实现里面去实现。如图所示。



那么内部是如何动态查找对应的方法的？ 首先我们知道所有的类中都继承自NSObject类，在NSObjcet中存在一个Class的isa指针。

```
typedef struct objc_class *Class;
@interface NSObject <NSObject> {
    Class isa  OBJC_ISA_AVAILABILITY;
}
```

我们来到objc_class中查看，其中包含着类的一些基本信息。

```
struct objc_class {
  Class isa; // 指向metaclass

  Class super_class ; // 指向其父类
  const char *name ; // 类名
  long version ; // 类的版本信息，初始化默认为0，可以通过runtime函数class_setVersion和class_getVersion进行修改、读取
  long info; // 一些标识信息,如CLS_CLASS (0x1L) 表示该类为普通 class ，其中包含对象方法和成员变量;CLS_META (0x2L) 表示该类为 metaclass，其中包含类方法;
  long instance_size ; // 该类的实例变量大小(包括从父类继承下来的实例变量);
  struct objc_ivar_list *ivars; // 用于存储每个成员变量的地址
  struct objc_method_list **methodLists ; // 与 info 的一些标志位有关,如CLS_CLASS (0x1L),则存储对象方法，如CLS_META (0x2L)，则存储类方法;
  struct objc_cache *cache; // 指向最近使用的方法的指针，用于提升效率;
  struct objc_protocol_list *protocols; // 存储该类遵守的协议
}
```

下面我们就以p实例的eat方法来看看具体消息发送之后是怎么来动态查找对应的方法的。

1. 实例方法 `[p eat];` 底层调用 `[p performSelector:@selector(eat)];` 方法，编译器在将代码转化为 `objc_msgSend(p, @selector(eat));`
2. 在 `objc_msgSend` 函数中。首先通过 `p` 的 `isa` 指针找到 `p` 对应的 `class` 。在 `Class` 中先去 `cache` 中通过 `SEL` 查找对应函数 `method` ，如果找到则通过 `method` 中的函数指针跳转到对应的函数中去执行。
3. 若 `cache` 中未找到。再去 `methodList` 中查找。若能找到，则将 `method` 加入到 `cache` 中，以方便下次查找，并通过 `method` 中的函数指针跳转到对应的函数中去执行。
4. 若 `methodlist` 中未找到，则去 `superClass` 中查找。若能找到，则将 `method` 加入到 `cache` 中，以方便下次查找，并通过 `method` 中的函数指针跳转到对应的函数中去执行。

三. 使用RunTime交换方法：

当系统自带的方法功能不够，需要给系统自带的方法扩展一些功能，并且保持原有的功能时，可以使用RunTime交换方法实现。 这里要实现image添加图片的时候，

自动判断image是否为空，如果为空则提醒图片不存在。 方法一：使用分类

```
+ (nullable UIImage *)xx_ccimageNamed:(NSString *)name
{
    // 加载图片     如果图片不存在则提醒或发出异常
   UIImage *image = [UIImage imageNamed:name];
    if (image == nil) {
        NSLog(@"图片不存在");
    }
    return image;
}
```

缺点：每次使用都需要导入头文件，并且如果项目比较大，之前使用的方法全部需要更改。

方法二： RunTime交换方法 交换方法的本质其实是交换两个方法的实现，即调换xx_ccimageNamed和imageName方法，达到调用xx_ccimageNamed其实就是调用imageNamed方法的目的

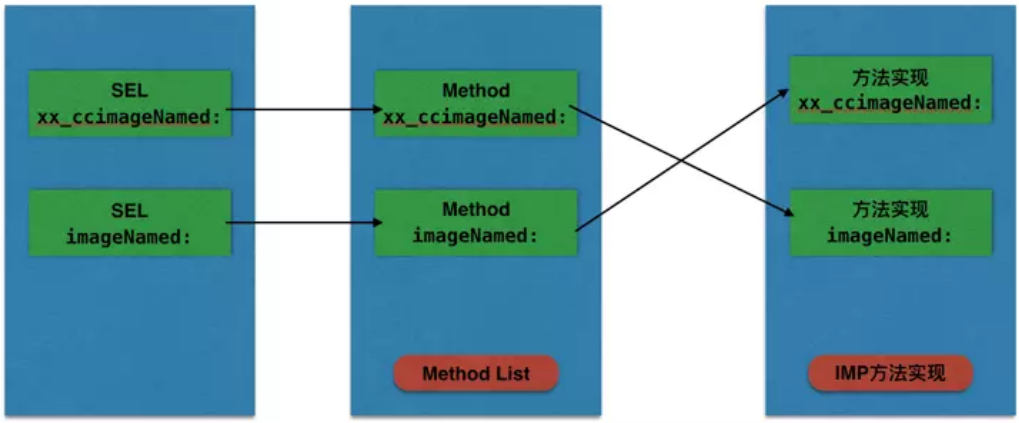
那么首先需要明白方法在哪里交换，因为交换只需要进行一次，所以在分类的load方法中，当加载分类的时候交换方法即可。

```
 +(void)load
{
    // 获取要交换的两个方法
    // 获取类方法  用Method 接受一下
    // class ：获取哪个类方法
    // SEL ：获取方法编号，根据SEL就能去对应的类找方法。
    Method imageNameMethod = class_getClassMethod([UIImage class],
@selector(imageNamed:));
    // 获取第二个类方法
    Method xx_ccimageNameMrthod = class_getClassMethod([UIImage
class], @selector(xx_ccimageNamed:));
    // 交换两个方法的实现 方法一 ， 方法二。
    method_exchangeImplementations(imageNameMethod,
xx_ccimageNameMrthod);
    // IMP其实就是 implementation的缩写：表示方法实现。
}
```

交换方法内部实现：

1. 根据SEL方法编号在Method中找到方法，两个方法都找到

2. 交换方法的实现，指针交叉指向。如图所示：



注意：交换方法时候 xx_ccimageNamed方法中就不能再调用imageNamed方法了，因为调用imageNamed方法实质上相当于调用 xx_ccimageNamed方法，会循环引用造成死循环。

RunTime也提供了获取对象方法和方法实现的方法。

```
// 获取方法的实现
class_getMethodImplementation(<#__unsafe_unretained Class cls#>,
<#SEL name#>)
// 获取对象方法
class_getInstanceMethod(<#__unsafe_unretained Class cls#>, <#SEL
name#>)
```

此时，当调用imageNamed:方法的时候就会调用xx_ccimageNamed:方法，为image添加图片，并判断图片是否存在，如果不存在则提醒图片不存在。

四. 动态添加方法

如果一个类方法非常多，其中可能许多方法暂时用不到。而加载类方法到内存的时候需要给每个方法生成映射表，又比较耗费资源。此时可以使用RunTime动态添加方法

动态给某个类添加方法，相当于懒加载机制，类中许多方法暂时用不到，那么就先

不加载，等用到的时候再去加载方法。

动态添加方法的方法： 首先我们先不实现对象方法，当调用performSelector: 方法的时候，再去动态加载方法。 这里同上创建Person类，使用performSelector: 调用Person类对象的eat方法。

```
Person *p = [[Person alloc]init];
// 当调用 P中没有实现的方法时，动态加载方法
[p performSelector:@selector(eat)];
```

此时编译的时候是不会报错的，程序运行时才会报错，因为Person类中并没有实现eat方法，当去类中的Method List中发现找不到eat方法，会报错找不到eat方法。

```
2016-10-14 21:26:20.922 RunTime 动态添加方法[5218:1107516] -[Person eat]: unrecognized selector sent to instance 0x7f8bb179b2d0
2016-10-14 21:26:20.932 RunTime 动态添加方法[5218:1107516] *** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: '-[Person eat]: unrecognized selector sent to instance 0x7f8bb179b2d0'
```

而当找不到对应的方法时就会来到拦截调用，在找不到调用的方法程序崩溃之前调用的方法。 当调用了没有实现的对象方法的时，就会调用**`+(BOOL)resolveInstanceMethod:(SEL)sel`**方法。 **当调用了没有实现的类方法的时候，就会调用**`+(BOOL)resolveClassMethod:(SEL)sel`**方法。

首先我们来到API中看一下苹果的说明，搜索 Dynamic Method Resolution 来到动态方法解析。

You can implement the methods `resolveInstanceMethod:` and `resolveClassMethod:` to dynamically provide an implementation for a given selector for an instance and class method respectively.

An Objective-C method is simply a C function that take at least two arguments—`self` and `_cmd`. You can add a function to a class as a method using the function `class_addMethod`. Therefore, given the following function:

```
void dynamicMethodIMP(id self, SEL _cmd) {
    // implementation ....
}
```

you can dynamically add it to a class as a method (called `resolveThisMethodDynamically`) using `resolveInstanceMethod:` like this:

```
@implementation MyClass
+ (BOOL)resolveInstanceMethod:(SEL)aSEL
{
    if (aSEL == @selector(resolveThisMethodDynamically)) {
          class_addMethod([self class], aSEL, (IMP) dynamicMethodIMP, "v@:");
          return YES;
    }
    return [super resolveInstanceMethod:aSEL];
}
@end
```

Dynamic Method Resolution的API中已经讲解的很清晰，我们可以实现方法**`resolveInstanceMethod:`**或者**`resolveClassMethod:`**方法，动态的给实例方法

或者类方法添加方法和方法实现。

所以通过这两个方法就可以知道哪些方法没有实现，从而动态添加方法。参数sel即表示没有实现的方法。

一个objective - C方法最终都是一个C函数，默认任何一个方法都有两个参数。self : 方法调用者 _cmd : 调用方法编号。我们可以使用函数class_addMethod为类添加一个方法以及实现。

这里仿照API给的例子，动态的为P实例添加eat对象

```
+(BOOL)resolveInstanceMethod:(SEL)sel
{
    // 动态添加eat方法
    // 首先判断sel是不是eat方法 也可以转化成字符串进行比较。
    if (sel == @selector(eat)) {
    /**
     第一个参数： cls:给哪个类添加方法
     第二个参数： SEL name:添加方法的编号
     第三个参数： IMP imp: 方法的实现，函数入口，函数名可与方法名不同（建议与方法名相同）
     第四个参数： types :方法类型，需要用特定符号，参考API
     */
      class_addMethod(self, sel, (IMP)eat , "v@:");
        // 处理完返回YES
        return YES;
    }
    return [super resolveInstanceMethod:sel];
}
```

重点来看一下class_addMethod方法

```
class_addMethod(__unsafe_unretained Class cls, SEL name, IMP imp,
const char *types)
```

class_addMethod中的四个参数。第一，二个参数比较好理解，重点是第三，四个参数。

1. cls : 表示给哪个类添加方法，这里要给Person类添加方法，self即代表Person。

2. SEL name : 表示添加方法的编号。因为这里只有一个方法需要动态添加，并且之前通过判断确定sel就是eat方法，所以这里可以使用sel。
3. IMP imp : 表示方法的实现，函数入口，函数名可与方法名不同（建议与方法名相同）需要自己来实现这个函数。每一个方法都默认带有两个隐式参数 **self : 方法调用者 _cmd : 调用方法的标号**，可以写也可以不写。

```
void eat(id self ,SEL _cmd)
{
      // 实现内容
      NSLog(@"%@的%@方法动态实现了",self,NSStringFromSelector(_cmd));
}
```

4. types : 表示方法类型，需要用特定符号。系统提供的例子中使用的是 ** `"v@:"` **，我们来到API中看看** `"v@:"` **指定的方法是什么类型的。

Table 6-1 Objective-C type encodings

Code	Meaning
`c`	A `char`
`i`	An `int`
`s`	A `short`
`l`	A `long` `l` is treated as a 32-bit quantity on 64-bit programs.
`q`	A `long long`
`C`	An `unsigned char`
`I`	An `unsigned int`
`S`	An `unsigned short`
`L`	An `unsigned long`
`Q`	An `unsigned long long`
`f`	A `float`
`d`	A `double`
`B`	A C++ `bool` or a C99 `_Bool`
`v`	A `void`
`*`	A character string (`char *`)
`@`	An object (whether statically typed or typed `id`)
`#`	A class object (`Class`)
`:`	A method selector (`SEL`)
[*array type*]	An array
{*name=type...*}	A structure
(*name=type...*)	A union
`b`num	A bit field of *num* bits
^type	A pointer to *type*
`?`	An unknown type (among other things, this code is used for function pointers)

> `v` -> void 表示无返回值 `@` -> object 表示id参数 `:` -> method selector 表示SEL

至此已经完成了P实例eat方法的动态添加。当P调用eat方法时输出

```
RunTime 动态添加方法[5561:1170796] <Person: 0x7fa5894111e0>的eat方法动态实现了
```

动态添加有参数的方法 如果是有参数的方法，需要对方法的实现和class_addMethod方法内方法类型参数做一些修改。 方法实现：因为在C语言函数中，所以对象参数类型只能用id代替。 方法类型参数：因为添加了一个id参数，所以方法类型应该为** `"v@:@"` ** 来看一下代码

```
+(BOOL)resolveInstanceMethod:(SEL)sel
{
    if (sel == @selector(eat:)) {
        class_addMethod(self, sel, (IMP)aaaa , "v@:@");
        return YES;
    }
    return [super resolveInstanceMethod:sel];
}
void aaaa(id self ,SEL _cmd,id Num)
{
    // 实现内容
    NSLog(@"%@的%@方法动态实现了,参数为%@",self,NSStringFromSelector(_cmd),Num);
}
```

调用 `eat:` 函数

```
Person *p = [[Person alloc]init];
[p performSelector:@selector(eat:)withObject:@"xx_cc"];
```

输出为

```
RunTime 动态添加方法[5764:1189343] <Person: 0x7fbdf5906c50>的eat:方法动态实现了,参数为xx_cc
```

五. RunTime动态添加属性

使用RunTime给系统的类添加属性，首先需要了解对象与属性的关系。

[图片上传失败...(image-7506e7-1525853287211)]

对象一开始初始化的时候其属性name为nil，给属性赋值其实就是让name属性指向

一块存储字符串的内存，使这个对象的属性跟这块内存产生一种关联，个人理解对象的属性就是一个指针，指向一块内存区域。

那么如果想动态的添加属性，其实就是动态的产生某种关联就好了。而想要给系统的类添加属性，只能通过分类。

这里给NSObject添加name属性，创建NSObject的分类 我们可以使用@property给分类添加属性

```
@property(nonatomic,strong)NSString *name;
```

虽然在分类中可以写@property 添加属性，但是不会自动生成私有属性，也不会生成set,get方法的实现，只会生成set,get的声明，需要我们自己去实现。

方法一：我们可以通过使用静态全局变量给分类添加属性

```
static NSString *_name;
-(void)setName:(NSString *)name
{
    _name = name;
}
-(NSString *)name
{
    return _name;
}
```

但是这样_name静态全局变量与类并没有关联，无论对象创建与销毁，只要程序在运行_name变量就存在，并不是真正意义上的属性。

方法二：使用RunTime动态添加属性 RunTime提供了动态添加属性和获得属性的方法。

```
-(void)setName:(NSString *)name
{
    objc_setAssociatedObject(self, @"name",name,
OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}
-(NSString *)name
{
```

```
    return objc_getAssociatedObject(self, @"name");
}
```

1. 动态添加属性

```
objc_setAssociatedObject(id object, const void *key, id value,
objc_AssociationPolicy policy);
```

参数一：**`id object`**：给哪个对象添加属性，这里要给自己添加属性，用self。 参数二：**`void * == id key`**：属性名，根据key获取关联对象的属性的值，在**`objc_getAssociatedObject`**中通过次key获得属性的值并返回。 参数三：`id value`：关联的值，也就是set方法传入的值给属性去保存。 参数四：**`objc_AssociationPolicy policy`**：策略，属性以什么形式保存。 有以下几种

```
typedef OBJC_ENUM(uintptr_t, objc_AssociationPolicy) {
    OBJC_ASSOCIATION_ASSIGN = 0,  // 指定一个弱引用相关联的对象
    OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1, // 指定相关对象的强引用，非原子性
    OBJC_ASSOCIATION_COPY_NONATOMIC = 3,  // 指定相关的对象被复制，非原子性
    OBJC_ASSOCIATION_RETAIN = 01401,  // 指定相关对象的强引用，原子性
    OBJC_ASSOCIATION_COPY = 01403     // 指定相关的对象被复制，原子性
};
```

2. 获得属性

```
objc_getAssociatedObject(id object, const void *key);
```

参数一：**`id object`**：获取哪个对象里面的关联的属性。 参数二：**`void * == id key`**：什么属性，与**`objc_setAssociatedObject`**中的key相对应，即通过key值取出value。

此时已经成功给NSObject添加name属性，并且NSObject对象可以通过点语法为属性赋值。

```
NSObject *objc = [[NSObject alloc]init];
objc.name = @"xx_cc";
NSLog(@"%@",objc.name);
```

六. RunTime字典转模型

为了方便以后重用，这里通过给NSObject添加分类，声明并实现使用RunTime字典转模型的类方法。

```
+ (instancetype)modelWithDict:(NSDictionary *)dict
```

首先来看一下KVC字典转模型和RunTime字典转模型的区别

> KVC：KVC字典转模型实现原理是遍历字典中所有Key，然后去模型中查找相对应的属性名，要求属性名与Key必须一一对应，字典中所有key必须在模型中存在。 RunTime：RunTime字典转模型实现原理是遍历模型中的所有属性名，然后去字典查找相对应的Key，也就是以模型为准，模型中有哪些属性，就去字典中找那些属性。

RunTime字典转模型的优点：当服务器返回的数据过多，而我们只使用其中很少一部分时，没有用的属性就没有必要定义成属性浪费不必要的资源。只保存最有用的属性即可。

RunTime字典转模型过程 首先需要了解，属性定义在类里面，那么类里面就有一个属性列表，属性列表以数组的形式存在，根据属性列表就可以获得类里面的所有属性名，所以遍历属性列表，也就可以遍历模型中的所有属性名。 所以RunTime字典转模型过程就很清晰了。

1. 创建模型对象

```
id objc = [[self alloc] init];
```

2. 使用**`class_copyIvarList`**方法拷贝成员属性列表

```
unsigned int count = 0;
Ivar *ivarList = class_copyIvarList(self, &count);
```

参数一：`__unsafe_unretained Class cls`：获取哪个类的成员属性列表。这里是self，因为谁调用分类中类方法，谁就是self。 参数二：`unsigned int *outCount`：无符号int型指针，这里创建unsigned int型count，&count就是他的地址，保证在方法中可以拿到count的地址为count赋值。传出来的值为成员属性总数。 返回值：`Ivar *`：返回的是一个Ivar类型的指针 。指针默认指向的是数组的第0个元素，指针+1会向高地址移动一个Ivar单位的字节，也就是指向第一个元素。Ivar表示成员属性。 3. 遍历成员属性列表，获得属性列表

```
for (int i = 0 ; i < count; i++) {
        // 获取成员属性
        Ivar ivar = ivarList[i];
}
```

4. 使用**`ivar_getName(ivar)`**获得成员属性名，因为成员属性名返回的是C语言字符串，将其转化成OC字符串

```
NSString *propertyName = [NSString
stringWithUTF8String:ivar_getName(ivar)];
```

通过**`ivar_getTypeEncoding(ivar)`**也可以获得成员属性类型。 5. 因为获得的是成员属性名，是带_的成员属性，所以需要将下划线去掉，获得属性名，也就是字典的key。

```
// 获取key
NSString *key = [propertyName substringFromIndex:1];
```

6. 获取字典中key对应的Value。

```
// 获取字典的value
id value = dict[key];
```

7. 给模型属性赋值，并将模型返回

```
if (value) {
 // KVC赋值:不能传空
[objc setValue:value forKey:key];
}
return objc;
```

至此已成功将字典转为模型。

七. RunTime字典转模型的二级转换

在开发过程中经常用到模型嵌套，也就是模型中还有一个模型，这里尝试用RunTime进行模型的二级转换，实现思路其实比较简单清晰。

1. 首先获得一级模型中的成员属性的类型

```
// 成员属性类型
NSString *propertyType = [NSString
stringWithUTF8String:ivar_getTypeEncoding(ivar)];
```

2. **判断当一级字典中的value是字典，并且一级模型中的成员属性类型不是NSDictionary的时候才需要进行二级转化。** 首先value是字典才进行转化是必须的，因为我们通常将字典转化为模型，其次，成员属性类型不是系统类，说明成员属性是我们自定义的类，也就是要转化的二级模型。而当成员属性类型就是NSDictionary的话就表明，我们本就想让成员属性是一个字典，不需要进行模型的转换。

```
id value = dict[key];
if ([value isKindOfClass:[NSDictionary class]] && ![propertyType
containsString:@"NS"])
{
      // 进行二级转换。
}
```

3. 获取要转换的模型类型，这里需要对propertyType成员属性类型做一些处理，因为propertyType返回给我们成员属性类型的是**`@\"Mode\"`，**我们需要对他进行截取为`Mode`**。这里需要注意的是\只是转义符，不占位。

```
// @\"Mode\"去掉前面的@\"
NSRange range = [propertyType rangeOfString:@"\""];
propertyType = [propertyType substringFromIndex:range.location +
range.length];
// Mode\"去掉后面的\"
range = [propertyType rangeOfString:@"\""];
propertyType = [propertyType substringToIndex:range.location];
```

4. 获取需要转换类的类对象，将字符串转化为类名。

```
Class modelClass =  NSClassFromString(propertyType);
```

5. 判断如果类名不为空则调用分类的modelWithDict方法，传value字典，进行二级模型转换，返回二级模型在赋值给value。

```
if (modelClass) {
      value =  [modelClass modelWithDict:value];
}
```

这里可能有些绕，重新理一下，我们通过判断value是字典并且需要进行二级转换，然后将value字典转化为模型返回，并重新赋值给value，最后给一级模型中相对应的key赋值模型value即可完成二级字典对模型的转换。

最后附上二级转换的完整方法

```
+ (instancetype)modelWithDict:(NSDictionary *)dict{
    // 1.创建对应类的对象
    id objc = [[self alloc] init];
    // count:成员属性总数
    unsigned int count = 0;
   // 获得成员属性列表和成员属性数量
```

```
    Ivar *ivarList = class_copyIvarList(self, &count);
    for (int i = 0 ; i < count; i++) {
        // 获取成员属性
        Ivar ivar = ivarList[i];
        // 获取成员名
       NSString *propertyName = [NSString
stringWithUTF8String:ivar_getName(ivar)];
        // 获取key
        NSString *key = [propertyName substringFromIndex:1];
        // 获取字典的value key:属性名 value:字典的值
        id value = dict[key];
        // 获取成员属性类型
        NSString *propertyType = [NSString
stringWithUTF8String:ivar_getTypeEncoding(ivar)];
        // 二级转换
        // value值是字典并且成员属性的类型不是字典,才需要转换成模型
        if ([value isKindOfClass:[NSDictionary class]] && !
[propertyType containsString:@"NS"]) {
            // 进行二级转换
            // 获取二级模型类型进行字符串截取，转换为类名
            NSRange range = [propertyType rangeOfString:@"\""];
            propertyType = [propertyType
substringFromIndex:range.location + range.length];
            range = [propertyType rangeOfString:@"\""];
            propertyType = [propertyType
substringToIndex:range.location];
            // 获取需要转换类的类对象
           Class modelClass =  NSClassFromString(propertyType);
           // 如果类名不为空则进行二级转换
            if (modelClass) {
                // 返回二级模型赋值给value
                value =  [modelClass modelWithDict:value];
            }
        }
        if (value) {
            // KVC赋值:不能传空
            [objc setValue:value forKey:key];
        }
    }
    // 返回模型
    return objc;
}
```