**Objective-C 2.0的新特性与运行时编程**

(1.)属性访问器属性访问器：： 第一篇文档中我们都要给成员变量编写样板式的getter、setter方法，Objective-C 2.0中给出了一些新的指令，简化这些代码。我们先看一个示例：

Person.h:

@interface Person : NSObject{

NSString \*name;

float weight;

}

-(Person\*) initWithWeight: (int) weight;

@property (retain,readwrite) NSString\* name;

@property (readonly)float weight;

@end

这里的@property (parameter1,parameter2)在h文件中，自动声明属性，也就是在编译之后的代码中添加成员变量的setter、getter方法。

如果你希望控制属性是只读的（不生成setter方法），请使用readonly参数，默认是readwrite的。

如果属性是对象类型，你需要使用retain、assign、copy参数，表示setter方法内部实现的时候，持有对象的方式。

其中retain就是增加引用计数，强引用类型；

assign就是变量的直接赋值，弱引用类型，也是默认值；

copy就是把setter的参数复制一份，再赋给成员变量，也就是第一篇文档中的对象复制一节中最后面所说的setter的第三种赋值策略。

如果你不给出持有对象的方式，编译器会给出警告。

另外，@property的( )中的参数还有一种不常用的atomic、nonatomic，前者是默认值，表示属性是原子的，支持多线程并发访问（实际就是setter的实现中加入了同步锁），后者是非原子的，也就是适合在非多线程的环境提升效率（因为 setter中没有同步锁的代码）。

Person.m:

@implementation Person

-(Person\*) initWithWeight: (int) w{

self=[super init];

if (self) {

weight=w;

//我们看到这里不会报错，因为我们不是通过setter方法访问的。

}

return self;

}

@synthesize name;

@synthesize weight;

-(void) dealloc{

[self setName:nil];

// [super dealloc];

}

@end

这里的@synthesize表示在m文件中，自动实现访问器方法，注意这里不需要再声明类型。另外，dealloc中的第一行，我们没有像之前一样[name release]，而是使用编译器为我们生成的setter方法释放name的所有权，这样对于assign、retain、copy三种情况都是适用的。

main.m:

int main(int argc, const char \* argv[]) {

@autoreleasepool {

Person \*person=[[Person alloc] initWithWeight:68];

person.name=@"Jetta";

//person.weight=68;//此行会报错，因为weight是只读的。

printf("The person's name is %s, weight is %f!", [person.name cStringUsingEncoding:NSUTF8StringEncoding], person.weight);

}

return 0;

}

这里我们看到访问setter、getter的方法，居然用JAVA中的.语法，这也是Objective-C 2.0的新特性，支持用.

语法访问属性，自然也不用使用中缀语法的[ ]进行包围。

同样是person.name，它在=左侧就是调用setter方法，在=右侧或者作为参数传递的就是调用getter方法。

这里还需要注意的是我们使用cStringUsingEncoding方法把NSString转换为C语言的字符串，因为第一篇文档里用的cString方法已经过时，不推荐使用。

NSUTF8StringEncoding是一个枚举类型，你可以选中它，然后按下control+shift+D查看还有其他哪些枚举值。

(2.)for-each循环循环：： 第一篇文档已经介绍过，主要用于快速迭代集合类。

(3.)GC机制机制：： Xcode中的工程默认是不启用gc机制的，你可以在工程上鼠标右键---Get Info，然后切换到第二个选项卡Build，在搜索框中输入gc，然后找到下图中被选中的一行，我们看到是Unsupported的，你可以选择为Supported、Required启用gc机制。在gc模式下，我们第一篇文档说的retain、release、autorelease都干什么呢？实际上这几个操作在gc模式下会被忽略，也就是执行的是空操作。这也就是说你基于手动管理内存的代码，在gc机制下不需要做特别的改动。

另外，再重申一下第一篇文档中提到的内容，就是iOS平台不支持GC机制。

(4.)协议的必选与可选方法协议的必选与可选方法：：

在Objective-C 2.0中增加了如下两个指令，如下所示：

@protocol MyProtocol <NSObject>

@required

- (void)requiredMethod;

@optional

- (void)optionalMethod;

@end

协议中的方法声明前使用@required的表示这个方法必须实现，否则将会得到编译器的警告，而@optional为可选实现，为默认值。

Objective-C 2.0的运行时运行时编程编程编程

Objective-C 2.0的运行时环境叫做Morden Runtime，iOS和Mac OS X 64-bit的程序都运行在这个环境，也就是说Mac OS X 32-bit的程序运行在旧的Objective-C 1.0的运行时环境Legacy Runtime，这里我们只讲解Morden Runtime。

同运行时交互主要在三个不同的地方，分别是

A.Objective-C源码（譬如：你定义的Category中的新方法会在运行时自动添加到原始类）、

B.NSObject的方法（isMemberClassOf等动态判定的方法）、

C.运行时函数。由于前两者在第一篇文档中讲解过，这里我们讲一下运行时函数的相关内容。

(1.)isa指针指针：：

NSObject中有一个Class isa的指针类型的成员变量，因为我们的对象大都直接或者间接的从NSObject继承而来，因此都会继承这个isa成员变量，isa在运行时会指向对象的Class对象，一个类的所有对象的Class对象都是同一个（JAVA也是如此），这保证了在内存中每一个类型都有唯一的类型描述。这个Class对象中也有个isa指针，它指向了上一级的父类的Class对象。

在明白了这个isa之后，你就可以明白在继承的时候，A extends B，你调用A的方法a()，首先A的isa到A的Class对象中去查找a()方法，找到了就调用，如果没找到，就驱使A的Class对象中的isa到父类B的Class对象中去查找。

(2.)SEL与IMP：

第一篇文档中，我们提到了方法选择器SEL，它可以通过如下两种方式获得：

(SEL) @selector(方法的名字)

(SEL) NSSelectorFromString(方法的名字的字符串)

另外，你还可以通过(NSString\*) NSStringFromSelector(SEL)函数来获取SEL所指定的方法名称字符串。

其实Objective-C在编译的时候，会依据每一个定义的方法的名字、参数序列，生成一个唯一的整数标识，这个标识就是SEL。因此，在运行时查找方法都是通过这个唯一的标识，而不是通过方法的名字。

其实Objective-C在编译的时候，会依据每一个定义的方法的名字、参数序列，生成一个唯一的整数标识，这个标识就是SEL。因此，在运行时查找方法都是通过这个唯一的标识，而不是通过方法的名字。

Objective-C又提供了IMP类型，IMP表示指向实现方法的指针（函数指针），通过它，你可以直接访问一个实现方法，从而避免了[xxx message]的静态调用方式，需要首先通过SEL确定方法，然后再通过IMP找到具体的实现方法，最后再发送消息所带来的执行效率问题。一般，如果你在多次循环中反复调用一个方法，用IMP的方式，会比直接向对象发送消息高效一些。

例：

Person.m:

#import "Person.h"

@implementation Person

@synthesize name;

@synthesize weight;

- (Person \*)initWithWeight:(int)w {

self = [super init];

if (self) {

weight = w;

}

return self;

}

- (void)print:(NSString \*)str {

NSLog(@"%@ %@", str, name);

}

- (void)dealloc {

[self setName:nil];

// [super dealloc];

}

@end

main.m:

int main(int argc, const char \* argv[]) {

@autoreleasepool {

Person \*person=[[Person alloc] initWithWeight:68];

person.name=@"Jetta";

SEL print\_sel=NSSelectorFromString(@"print:");

IMP imp=[person methodForSelector: print\_sel];

imp(person,print\_sel,@"\*\*\*\*\*\*\*\*\*");

}

return 0;

}

这里我们看到要获得IMP的指针，可以通过NSObject中的methodForSelector: (SEL)方法，访问这个指针函数，我们使用imp(id,SEL,argument1,… …)，第一个参数是调用方法的对象，第二个方法是方法的选择器对象，第三个参数是可变参数，表示传递方法需要的参数。

(3.)objc\_msgSend函数函数：：

通过isa指针的讲解，我们知道Objective-C中的方法调用是在运行时才去绑定的，再进一步看，编译器会把对象消息发送[xxx method]转换为objc\_msgSend(id receiver,SEL selector,参数…)的函数调用。因此上面例子中的print方法你也可以像下面这样调用： objc\_msgSend(person,print\_sel,@"++++++++");

当然，这是编译器要做的事情，你在写代码的时候，是不需要直接使用这种写法的。 综合isa、SEL、IMP的讲解，实际上objc\_msgSend的调用过程就应该是这样的。

A.首先通过第一个参数的receiver，找到它的isa指针，然后在isa指向的Class对象中使用第二个参数selector查找方法；

B.如果没有找到，就使用当前Class对象中的新的isa指针到上一级的父类的Class对象中查找；

C.当找到方法后，再依据receiver的中的self指针找到当前的对象，调用当前对象的具体实现的方法（IMP指针函数），然后传递参数，调用实现方法。

D.假如一直找到NSObject的Class对象，也没有找到你调用的方法，就会报告不能识别发送消息的错误。

(4.)动态动态方法解析方法解析方法解析：：

我们在Objective-C 2.0的新特性中的属性访问器一节中，实际忽略了一个内容，那就是动态属性。

Objective-C 2.0中增加了@dynamic指令，表示变量对应的属性访问器方法，是动态实现的，你需要在NSObject中继承而来的+(BOOL) resolveInstanceMethod:(SEL) sel方法中指定动态实现的方法或者函数。

例:

Person.h:

@interface Person:NSObject{

NSString \*name;

float weight;

}

- (Person \*) initWithWeight: (int) weight;

@property (retain,readwrite) NSString\* name;

@property (readonly)float weight;

@property float height;

- (void) print: (NSString\*) str;

@end

Person.m:

#include <objc/runtime.h>

void dynamicMethod(id self, SEL \_cmd, float w){

printf("dynamicMethod-%s\n",[NSStringFromSelector(\_cmd) cStringUsingEncoding:NSUTF8StringEncoding]);

printf("%f\n",w);

}

@implementation Person

@synthesize name;

@synthesize weight;

@dynamic height;

-(Person\*) initWithWeight: (int) w{

self=[super init];

if (self) {

weight=w;

}

return self;

}

-(void) print: (NSString\*) str{

NSLog(@"%@%@",str,name);

}

+(BOOL) resolveInstanceMethod: (SEL) sel{

NSString \*methodName=NSStringFromSelector(sel);

BOOL result=NO;

//看看是不是我们要动态实现的方法名称

if ([methodName isEqualToString:@"setHeight:"]) {

class\_addMethod([self class], sel, (IMP) dynamicMethod, "v@:f");

result=YES;

}

return result;

}

-(void) dealloc{

[self setName:nil];

// [super dealloc];

}

@end

这里我们对于接口中的height在实现类中使用了@dynamic指令，紧接着，你需要指定一个函数或者其他类的方法作为height的setter、getter方法的运行时实现。为了简单，我们指定了Person.m中定义的函数（注意这是C语言的函数，不是Objective-C的方法）dynamicMethod作为height的setter方法的运行时实现。被指定为动态实现的方法的dynamicMethod的参数有如下的要求：

A.第一个、第二个参数必须是id、SEL；

B.第三个参数开始，你可以按照原方法（例如：setHeight:(float)）的参数定义。

再接下来，你需要覆盖NSObject的类方法resolveInstanceMethod，这个方法会把需要动态实现的方法（setHeight:）的选择器传递进来，我们判断一下是否是需要动态实现的选择器，如果是就把处理权转交给dynamicMethod。如何转交呢？这里我们就要用到运行时函数class\_addMethod(Class,SEL,IMP,char[])。

运行时函数位于objc/runtime.h，正如名字一样，这里面都是C语言的函数。按照这些函数的功能的不同，主要分为如下几类：操作类型、操作对象、操作协议等。大多数的函数都可以通过名字看出是什么意思，例如：class\_addProtocol动态的为一个类型在运行时增加协议、objc\_getProtocol把一个字符串转换为协议等。具体这些运行时函数都是做什么用的，你可以参看Apple官方页面：http://developer.apple.com/library/ios/documentation/Cocoa/Reference/ObjCRuntimeRef/Reference/reference.html#//apple\_ref/doc/uid/TP40001418

言归正传，我们来解释一下这里需要用到的class\_addmethod方法，这个方法有四个参数，Class表示你要为哪个类型增加方法，SEL参数表示你要增加的方法的选择器，IMP表示你要添加的方法的运行时的具体实现的函数指针。其实在这里你能够看出SEL并不能在运行时找到真正要调用的方法，IMP才可以真正的找到实现方法的。

在讲解第四个参数char[]之前，我们先看一下第一篇文档中提到的@encode指令，在把任意非Objective-C对象类型封装为NSValue类型的时候使用到了@encode指令，但当时我们没有详细说明这个指令的含义。实际上@encode()可以接受任何类型，Objective-C中用这个指令做类型编码，它可以把任何一个类型转换为字符串，譬如：void类型被编码之后为v，对象类型为@，SEL类型为:等，具体的你可以参看Apple官方页面关于Type Encoding的描述： http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtTypeEncodings.html#//apple\_ref/doc/uid/TP40008048-CH100-SW

现在我们来正式的看以下第四个参数v@:f的含义，它描述了IMP指向的函数的描述信息，按照@encode指令编译之后的字符说明，第一个字符v表示返回值为void，剩余的字符为dynamicMethod函数的参数描述，@表示第一个参数id，:自然就是第二个参数SEL，f就是第三个参数float。由于前面说过动态方法的实现的前两个参数必须是id、SEL，所以第四个参数中的字符串的第二、三个字符一定是@:。

我们看到resolveInstanceMethod方法的返回值为BOOL，也就是这个方法返回YES表示找到了动态方法的具体实现，否则就表示没有在运行时找到真实的实现，程序就汇报错。

经过了上面的处理，Objective-C的运行时只要发现你调用了@dynamic标注的属性的setter、getter方法，就会自动到resolveInstanceMethod里去寻找真实的实现。这也就是说你在main.m中调用peson.height的时候，实际上dynamicMethod函数被调用了。

实际上除了@dynamic标注的属性之外，如果你调用了类型中不存在的方法，也会被resolveInstanceMethod或者resolveClassMethod截获，但由于你没有处理，所以会报告不能识别的消息的错误。

你可能在感叹一个@dynamic指令用起来真是麻烦，我也是研究了半天Apple官方的晦涩的鸟语才搞明白的。不过好在一般Objective-C的运行时编程用到的并不多，除非你想设计一个动态化的功能，譬如：从网络下载一个升级包，不需要退出原有的程序，就可以动态的替换掉旧的功能等类似的需求。

5.)消息转发消息转发：：

在前面的objc\_msgSend()函数的最后，我们总结了Objective-C的方法调用过程，在最后一步我们说如果一路找下来还是没有找到调用的方法，就会报告错误，实际上这里有个细节，那就是最终找不到调用的方法的时候，系统会调用-(void) forwardInvocation: (NSInvocation\*) invocation方法，如果你的对象没有实现这个方法，就调用NSObject的forwardInvocation方法，那句不能识别消息的错误，实际就是NSObject的forwardInvocation抛出来的异常。 我们这里告诉你这个系统内部的实现过程，实际是要告诉你，你可以覆盖forwardInvocation方法，来改变NSObject的抛异常的处理方式。譬如：你可以把A不能处理的消息转发给B去处理。

NSInvocation是一个包含了receiver、selector的对象，也就是它包含了向一个对象发送消息的所有元素：对象、方法名、参数序列，你可以调用NSInvocation的invoke方法将这个消息激活。

例：

main.m:

int main(int argc, const char \* argv[]) {

@autoreleasepool {

Person \*person=[[Person alloc] init];

person.name=@"Jetta";

[person fly];

}

return 0;

}

这里我们调用了一个Person中不存在的方法fly。

Bird.m:

#import "Bird.h"

@implementation Bird

-(void) fly{

printf("Bird Can fly!");

}

@end

Person.m

@implementation Person

@synthesize name;

@synthesize weight;

-(NSMethodSignature\*) methodSignatureForSelector:(SEL)selector{

//首先调用父类的方法

NSMethodSignature \*signature= [super methodSignatureForSelector: selector];

//如果当前对象无法回应此selector，那么selector构造的方法签名必然为nil

if (!signature) {

//首先判断Bird的实例是否有能力回应此selector

if ([Bird instancesRespondToSelector:selector]) {

//获取Bird的selector的方法签名对象

signature=[Bird instanceMethodSignatureForSelector: selector];

}

}

return signature;

}

-(void) forwardInvocation: (NSInvocation\*) invocation{

//首先验证Bird是否有能力回应invocation中包含的selector

if ([Bird instancesRespondToSelector:[invocation selector]]) {

//创建要移交消息响应权的实例bird

Bird \*bird=[Bird new];

//激活invocation中的消息，但是消息的响应者是bird，而不是默认的self。

[invocation invokeWithTarget:bird];

}

}

-(void) dealloc{

[self setName:nil];

// [super dealloc];

}

@end

下面我们来详细分析一下如果你想把不能处理的消息转发给其他的对象，需要经过哪个几个步骤：

A.首先，你要覆盖NSObject中的methodSignatureForSelector方法。这是因为你如果想把消息fly从Person转发给Bird处理，那么你必须将NSInvocation中包含的Person的fly的方法签名转换为Bird的fly的方法签名，也就是把方法签名纠正一下。

由此，你也看出来NSInvocation的创建，内部使用了两个对象，一个是receiver，一个是NSMethodSignature，而NSMethodSignature是由SEL创建的。NSInvocation确实存在一个类方法invocationWithMethodSignature返回自身的实例。

B.然后我们覆盖forwardInvocation方法，使用的不是invoke方法，而是invokeWithTarget方法，也就是把调用权由self转交给bird。

实际上消息转发机制不仅可以用来处理找不到方法的错误，你还可以变相的实现多继承。假如我们的Person想要拥有Bird、Fish的所有功能，其实你可以尽情的用Person的实例调用Bird、Fish的方法，只要在Person的forwardInvocation里，把消息的响应权转交给Bird或者Fish的实例就可以了。不过这种做法实在有点儿BT，除非万不得已，否则千万不要这么做，但是你也从这里能够看出来Objective-C这种语言有多么的灵活、强大，这是JAVA所完全不能相比的。