第六条：理解属性这一概念

对象的属性用于封装数据，通过设值函数和取值函数设置和获取对应的数据。编译器已经可以自动实现设值函数和取值函数，用点语法可以方便的书写。一般情况下，对象内存布局在编译期就固定好了，而编译器记录成员变量的内存地址是通过偏移量来实现的，这个偏移量表示该变量距离对象的内存区域有多远，然后就可以得到此变量。因此在原来的实例变量中加入一些新的，他们的顺序就会打乱，导致程序出问题。

对于此问题，**OC给出的解决方案是把实例变量当做一种存储偏移量所用的"特殊变量"——类对象保管**。偏移量会在运行期查找，如果类定义变化了对应的偏移量也就改变了。这样无论何时总能给出正确的偏移量。除此之外，还有另一种解决方案，就是不要直接使用实例变量，尽量使用存取方法。

如果用@property NSString *name声明一个对象，编译器将自动为类添加适当的实例变量（默认是_name），生成对应的存取方法。当然我们也可以由编译器合成存取方法，在实现文件中用关键字synthesize可以自己指定实例变量的名称,具体如下：

@synthesize name = _myName;//不再会生成默认的实例变量，将会生成自定的实例变量

如果不想合成存取方法，也可以自己实现。用@dynamic关键字可以明确告诉编译器不要自动创建自动合成所需要的实例变量，也不要创建存取方法，不实现也不会报错，编译器相信会在运行期找到。

属性特质：

原子性（atomic）：atomic会使用锁机制设置对象属性以保护操作的原子性。这就是说，如果多个线程在同时访问该属性，那么它们总会得到有效的值。会保证线程安全，然而iOS中使用同步锁会出现系统开销太大，从而造成性能问题；而不加锁的话，也就是使用nonatomic时，不会保证原子性，如果一个线程正在修改对象的属性值，另一个线程又正在取值，就有可能把正在修改还未完成的属性值取到，导致此线程取到的属性值不对。

读写特性：（readwrite，readonly，writeonly）

内存管理语义：

- assign：只会执行纯值类型的简单赋值操作；
- strong：表明属性定义了一种"拥有关系"，设置新值时会保留新值，释放旧值，最后将新值设置上去；
- weak：表明属性定义了一种"非拥有关系"，为这种属性设置新值时，既不保留新值，也不释放旧值，而是像assign一样简单的赋值，当属性值所指的对象摧毁时，属性值会清空；
- copy：表达的关系和strong类似，然而设值方法不会保留新值而是"拷贝"，当属性类型为NSString时，也会起到保护其封装性的特点，意为不可修改。如果实现属性所用的对象为可变的，就会复制一份。

要点：

1. 可以只用@property来声明对象中封装的数据；
2. 通过"特质"来说明存储数据需要的正确语义；
3. 在设置属性所对应的实例变量时一定要遵从属性所声明的语义。

第七条：在对象之内尽量直接访问实例变量

在对象之外访问实例变量时需要通过属性来访问，在对象里面访问实例变量时应该尽量通过直接访问的方式来完成，因为这种方式会绕过存取方式，速度快。但是直接访问不会触发KVO（键值对观测）通知，有一种折中的方法便是写入实例变量时用设值方法，取值时直接用实例变量。

要点：

1. 在对象内部读取属性数据时，应该使用实例变量，写入数据时，应该使用属性；
2. 在对象初始化或者dealloc中，应该尽量使用实例变量；
3. 用懒加载存储数据时需要用属性操作。

第八条：理解对象的同等性概念

对象之间的比对操作不应该用==，因为该操作比较的是两个指针本身，而不是所指的指针，应该使用NSObject对中中声明的isEqual:方法。但是对于NSSting对象系统提供了一种更加快速高效的方法isEqualToString:。

NSObject中声明的两个判断等同性的方法：

```
-(BOOL)isEqual:(id)object;
-(NSUInteger)hash;
```

NSObject对这两个方法的实现是：当且仅当其"指针值"完全一样时才相等。

要重写这两个方法，就必须清楚他们之间的约定：如果isEqual：返回TRUE，表示两个对象相同，则返回的hash值是一样的；但是依据hash返回相同的值并不能判断对象相同。

在实现isEqual：方法时，可以根据指针是否相等和所有的实例变量、实例方法相同来判断；在实现hash：时，如果所有的实例返回相同的值会在collection中产生极大的性能问题。因为在collection中检测对象时会根据哈希码做索引，这样如果collection中有很多个对象，他们的哈希吗都一样，则在插入等操作中将会非常耗时，因为它要扫描所有的对象。

可以这样重写hash方法

```
-(NSUInteger)hash {
    NSString *hashStr = [NSString stringWithFormat:@"%@:%@:%@",_firstName,_lastName,_age];//所有的实例变量字符串拼接
    return [hashStr hash];
}
```

也可以用下面高效的方法，当然它也可能碰撞

```
-(NSUInteger)hash {
    NSUInteger _f = [_firstName hash];
    NSUInteger _l = [_lastName hash];
    NSUInteger _a = _age;
    return _f ^ _l ^_a;//按位异或运算
}
```

通常在自定义类中这样定义判等操作

```
-(BOOL)isEqualToPerson:(WRPerson *)otherPerson {
    if (self == otherPerson) return YES;
    if (![_firstName isEqualToString:otherPerson.firstName]) return NO;
    if (![_lastName isEqualToString:otherPerson.lastName]) return NO;
    if (_age != otherPerson.age) return NO;
    return YES;
}
-(BOOL)isEqul:(id)object {
    if ([self class] == [object class])
         return [self isEqualToPerson:(WRPerson *)object];
    else
        return [super isEqual:object];
}
```

要点：

1. 要想检测对象的等同性，需要提供isEqual:和hash:两个方法；
2. 相同的对象具有相同的hash码，但是具有相同的hash码的对象却未必相同；
3. 不要盲目的逐条检测属性，而是要根据需求定制检测方案；
4. 编写hash方法时，应该使用速度快而且哈希码碰撞几率小的算法。

第九条：使用“族类模式”隐藏实现细节

一般情况下，有些对象会是一个族类，它里面包含多种子类，可以为族类抽象出一个根据类型创建子类的方法，然后用其子类依次创建对应的对象。

首先要定义抽象基类：

```
typedef NS_ENUM(NSInteger,WREmplyeeType) {
    WREmplyeeTypeDesigner,
    WREmplyeeTypeDeveloper,
    WREmplyeeTypeFinance,
};

@interface WREmplyee:NSObject
@property (copy) NSString *name;
@property (assign) NSUInteger age;
+(WREmplyee)employeeWithType;(WREmplyeeType)type;
@end

@implemention WREmployee

+(WREmployee *)employeeWithType:(WREmplyeeType)type {
    switch (type) {
        case WREmplyeeTypeDesigner:
            return [WREmplyeeTypeDesigner new];
            break;
        case WREmplyeeTypeDeveloper:
            return [WREmplyeeTypeDeveloper new];
            break;
        case WREmplyeeTypeFinance:
            reurn [WREmplyeeTypeFinance new];
```

```
            break;
    }
}
```

每个子类都由基类继承而来

```
@interface WREmplyeeDeveloper:WREmployee
-(void)doSomeTask;
@end
```

Cocoa中的collection类大多都是族类，比如NSArray类和NSMtableArray类，后者是前者的子类，前者的公共接口后者都可以使用，后者也因此具有前者没有的功能和接口。

要点：

1. 族类模式可以把一些实现细节隐藏在简单的公共接口之后；
2. 系统框架也经常用到族类；
3. 从族类的公共抽象类中继承子类时要当心，如有开发文档最好先看之。

第十条：在既有类中使用关联对象存放自定义数据

可以为对象关联很多其他的对象，这些对象通过“键”来区分，存储对象值的时候可以指明“存储策略“，用来维护对应的“内存管理语义”。存储策略有：

```
typedef OBJC_ENUM(uintptr_t, objc_AssociationPolicy) {
    OBJC_ASSOCIATION_ASSIGN = 0,
    OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1,
    OBJC_ASSOCIATION_COPY_NONATOMIC = 3,
    OBJC_ASSOCIATION_RETAIN = 01401,
    OBJC_ASSOCIATION_COPY = 01403
};
```

可以通过下面的方法管理关联对象：

```
//来关联对象;
void objc_setAssociatedObject(id object,const void *key,id value,objc_AssociationPolicy policy)
//获取关联值
id objc_getAssociatedObject(id object,const void *key)
// 移除掉所有的关联对象
void objc_removeAssociatedObjects(id object)
```

一般来说，有以下三种推荐的 key 值：

1. 声明 static char kAssociatedObjectKey; ，使用 &kAssociatedObjectKey 作为 key 值;
2. 声明 static void *kAssociatedObjectKey = &kAssociatedObjectKey; ，使用 kAssociatedObjectKey 作为 key 值；
3. 用 selector ，使用 getter 方法的名称作为 key 值。

我们可以使用一个例子来看看怎样关联对象：

```
-(void)askUesrQuestion {
    UIAlertView *alert = [UIAlertView alloc] initWithTitle:"Question" message:"What do you want to do?" delegate:self cancel
    [alert show];
-(void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex {
    if (buttonIndex == 0) [self doCancel];
    else [seld doContinue];
}
```

这样在一个视图控制器中处理多个警告信息视图代码就会变得更为复杂，要是能在警告视图中写好每个按钮的逻辑就好了，这就可以用到关联对象，通关联块（block）来写好后续工作。

```
#include<objc/runtime.h>
static void *WRAlertViewKey = @"WRAlertViewKey"';
-(void)askUserQuestion {
    UIAlertView *alert = [UIAlertView alloc] initWithTitle:"Question" message:"What do you want to do?" delegate:self cancel
    void (^block)(NSInteger) = ^(NSInteger buttonIndex) {
        if (buttonIndex == 0) [self doCandel];
        else [self doContinue];
```

```
    }
    objc_setAssociatedObject(alert,WRAlertViewKey,block,BJC_ASSOCIATION_COPY);
    [alert show];
}
//UIAlertViewDelegate
-(void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger buttonIndex) {
    void (^block)(NUInteger) = objc_getAssociatedObjct(alertView,WRAlertViewKey);
    block(buttonIndex);
}
```

这种方式虽然把代码放在一起了，但是这种方式可能会造成循环引用的问题，只是在没有其他办法的时候再使用。

要点：

1. 可以通过关联对象机制在把两个对象连起来；
2. 定义关联对象时可以指定内存管理语义，用以模仿定义属性时所采用的“拥有关系”与“非拥有关系”；
3. 只有在其他方法不可行的时候才选用关联对象，因为这种做法可能导致循环引用使内存泄露，并且不好找到缘故。

第十一条：理解objc_msgSend方法的作用

在OC中，对象调用方法是非常普遍的事情，但是它有一个专业术语“传递消息”，消息有名字和选择器，可以接受参数，可能会有返回值。

我们知道C的函数调用方式为“静态调用”，也就是说编译器在编译的时候就已经知道函数是干什么的了，以下面的代码为例：

```
#include<stdio.h>
void printHello() {
    printf("Hello World!"\n);
}
void printGoodBye(){
    printf("Goodbye,World"\n);
}
void doTheThing(int type) {
    if(type == 0) printHello();
    else printGoodBye();
}
```

如果不考虑内联，那么编译器在编译代码的时候就已经知道这两个函数的存在了，于是会直接调用这些他们，函数地址实际上是硬编码在指令中的。如果把上面的代码写成下面的样子：

```
#include<stdio.h>
void printHello() {
    printf("Hello World!"\n);
}
void printGoodBye(){
    printf("Goodbye,World"\n);
}
void doTheThing(int type) {
    void(*fnc)();
    if (type == 0) fnc = printHello();
    else fnc = printGoodbye();
    fnc;
    return 0;
}
```

这种方式就是“动态绑定”，只有在程序运行的时候，才会确定具体的函数fnc，然后才执行它。

在OC中，如果向某个对象传递消息就会用到动态绑定，在底层这些方法都是普通的C函数，然而在对象收到消息之后才能确定具体用到那个函数，甚至这时可以改变方法的实现，也就是OC的runtime机制，它使OC成为一门真正的动态语言。

在运行期为传递消息起核心作用的是 void objc_msgSend(id self,SEL,cmd,...)这个方法。objc_msgSend方法会根据接收者和选择器来确定为对象调用那种方法。而为了找到对应的方法，此方法的接收者会在自己的list_of_methods中寻找，如果找不到就会沿着继承体系向上寻找，直到找到，如果没找到就会执行消息转发。

要点：

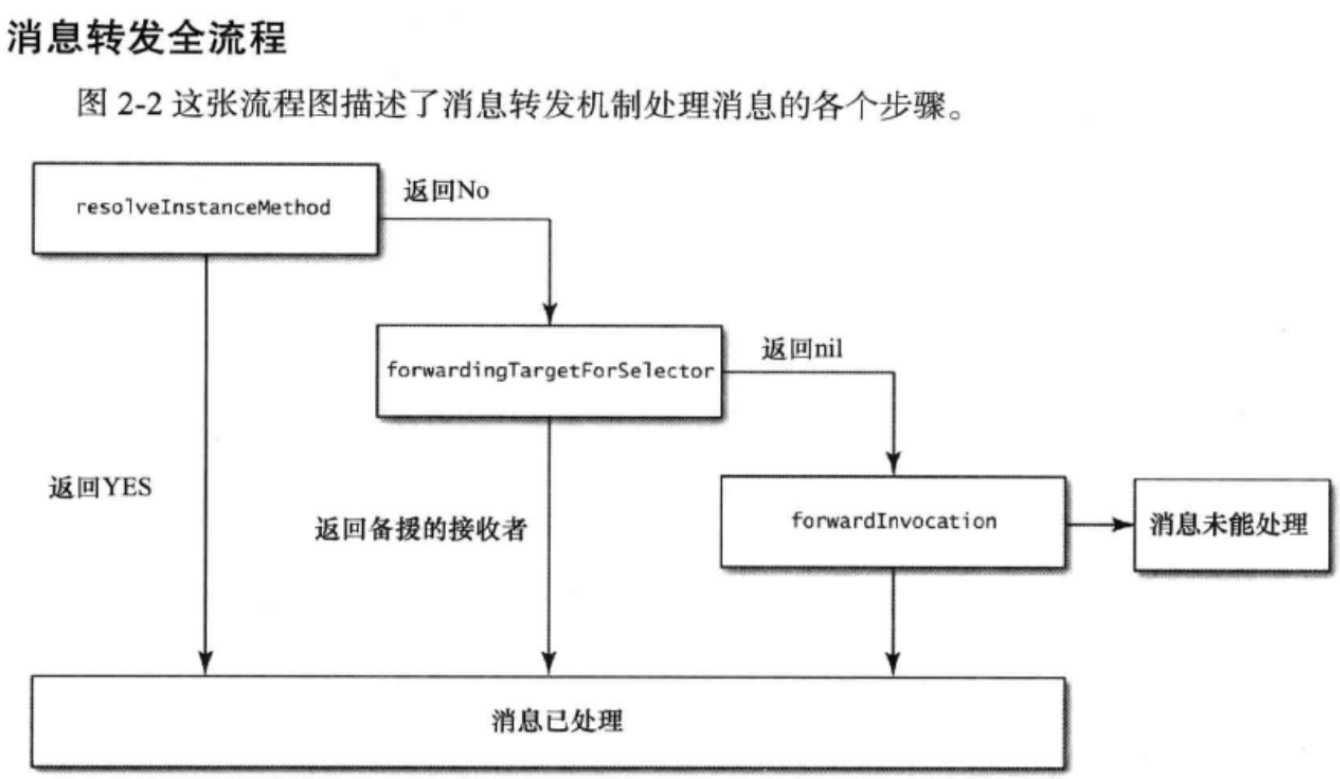
1. 消息由接收者，选择器和参数构成，给某个对象“发送消息”也就是在该对象上调用方法（call a method）；
2. 发送给某个对象的全部消息都是由“动态消息派发系统”（dynamic message dispatch system)来处理，该系统会查出对应的方法，并执行其代码。

第十二条 理解消息转发机制

如果为某个对象传递一条消息，但是编译器并没有找到这条消息中的方法的实现，这样不会报错，因为在运行期可以动态为此对象所属的类添加方法。如果在运行期也没有找到此方法的实现，就会执行“消息转发”，这样可告诉此对象如何处理未知消息。

消息转发分为两个阶段：

1. 先征询接收者所属的类，看其是否能动态添加方法，以处理当前无法处理的方法，这叫做“**动态方法解析**”；
2. 如果第一阶段已经完成，但是无法还是无法处理这个方法。此时运行期会以其他手段处理与消息相关的方法调用，这分为两步：
   - 接受者看有没有其他对象能处理这条消息，若有，则把这条消息转发给那个对象，于是，消息转发完成，一切如常；
   - 若没有找到“备援的接受者”，则启动完整的消息转发机制，运行期会把消息有关的全部细节封装带NSInvocation对象中，再给接收者最后一次机会，令其设法解决当前未处理的这条消息。



要点：

1. 若对象无法响应某个选择器，则进入消息转发流程；
2. 通过运行期的动态方法解析功能，我们可以在需要用到某个方法时再将其加入类中；
3. 对象可以把无法处理的选择器交给其他对象来处理；
4. 经过上述两个步骤之后，如果还是没有办法处理选择器，那就启动完成的消息转发机制。

第十三条 用“方法调配技术”调试“黑盒方法”

在类的方法列表中，每一个方法名称（方法选择器）对应各自的实现，在运行期，我们可以交换某个类的两个方法实现。比如，NSString 的两个方法upperCaseString和lowerCaseString可以在运行期相互交换：

```
Method upperMethod = class_getInstanceMethod([NSString class], @selector(uppercaseString));
Method lowerMethod = class_getInstanceMethod([NSString class], @selector(lowercaseString));
method_exchangeImplementations(upperMethod, lowerMethod);
NSString *str = @"aBcDeF";
NSLog(@"upperCase:%@",[str uppercaseString]);
NSLog(@"lowerCase:%@",[str lowercaseString]);
// 2016-12-16 10:50:16.506 WRKit[1666:52997] upperCase:abcdef
// 2016-12-16 10:50:16.507 WRKit[1666:52997] lowerCase:ABCDEF
```

但是，这样做实际上并没有多大的用处，我们可以通过为类添加自定义方法，然后和类的某个方法进行交换，从而实现为类的原有方法添加新功能或者改变实现。

比如：为NSString添加一个分类方法

```
@interface NSString(WRString)
-(NSString)w_LowerCaseString;
```

```
@end
```

```objc

@implementation NSString(WRString)
-(NSString)w_LowerCaseString {
    NSString *lower = [self lowerCaseString];
    NSLog(@"%@ => %@",self,lower)
    return lower;
}

Method NSLower = class_getInstanceMethod([NSString class],@Selector(lowerCaseString));
Method WRLower = class_getInstaceMethod([NSString class],@Selector(w_lowerCaseString));

method_exchangeImplementations(NSLower,WRLower);

执行上面的代码：
NSString *str = @"This is method exchange";
NSString *lowerStr = [str lowerCaseString];
// output:This is method exchange => this is method exchange
```

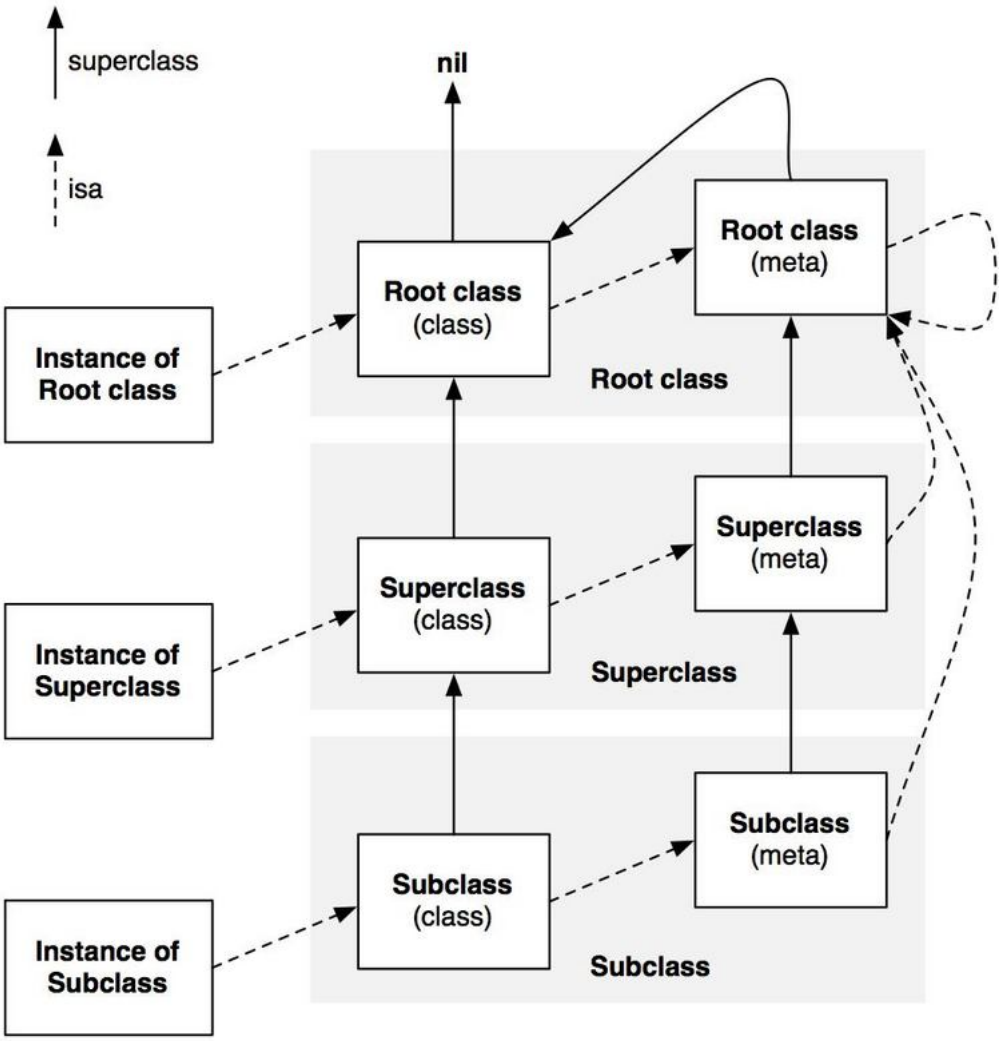
通常这种做法适用用调试阶段记录下日志，在真正的开发中并不会使用上去。

要点：

1. 在运行期，可以为类新增或者改变类的实现方法；
2. 使用另一份实现替换原有方法叫做“方法调配”或者“方法搅拌”，开发者通过此为方式添加新功能；
3. 一般来说只有在调试阶段使用，不宜滥用。

第十四条：理解“类对象”的用意

在OC中，每个对象的内部结构是这样设计的：



Objective-C class inheritance instructure

可以看见，每个对象的首个成员是Class变量，该变量指明了对象所属的类等信息，通常称为isa指针。这个结构体中存的信息还有超类，变量名，大小，实例变量，方法列表等。

类对象所属的类（也就是isa指针的类型）是另一个类，叫做“元类”（meta class），用于描述类对象本身所具备的元数据。类方法就定义于此，可以理解为类对象的实例方法，每个类只有一个“类对象”，每个类对象只有一个与之对应的“元类”。

假设有一个SomeClass对象继承自NSObject，则其继承体系为：

superClass指针确立的继承体系，而isa指针描述了实例所属的类型信息。

NSObject中查询类型信息的方法有：

- isMemberClass:判断是否为某个类的实例；
- isKindOfClass:判断是否为某类或者派生类的实例。

要点：

1. 每个对象都有一个指向其Class的指针，用于表示其类型，而这些Class对象则构成类的继承体系；
2. 如果类对象信息无法在编译器得知，那就应该在运行期用类型信息查询方法类探知；
3. 尽量使用类型信息查询方法来得知对象类型，而不要直接比较类对象，因为某个对象可能实现了消息转发功能。