Broadview
www.broadview.com.cn

FK 疯狂软件教育 www.fkit.org 标准教材

疯狂 iOS讲义（上）

Objective-C 2.0与iPhone/iPad应用开发基础（第2版）

李刚 编著

疯狂源自梦想

技术成就辉煌

疯狂源自梦想

技术成就辉煌

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

移动互联网已经成为当今世界发展最快、市场潜力最大、前景最诱人的业务，iOS 作为与 Android 并驾齐驱的操作系统，具有完善的生态圈，而且用户具有良好的付费习惯，因此 iOS 应用开发者具有良好的发展前景：不仅可入职移动互联网企业，更可以个人、小团队方式独立创业。

本书是《疯狂 iOS 讲义（上）》的第 2 版，本书基于最新的 iOS 8.1、Xcode 6.1 为平台，全面介绍了 iOS 应用开发的基础知识。书中每个案例、每个截图都全面升级到 iOS 8.1。本书全面、细致地介绍了 Objective-C 的面向对象语法、Foundation 框架的核心类库用法、文件 IO 体系等知识，掌握 Objective-C 语言是 iOS 应用开发的基础。掌握 Objective-C 语言之后，本书将带领读者掌握 iOS 应用开发的基本理论以及 iOS 应用的 MVC 设计和事件处理编程，重点介绍了 iOS 界面编程、iOS 应用的生命周期、Quartz 2D 绘图、坐标变换、Core Image 滤镜、Core Animation 动画等内容，掌握这些基础内容之后即可开发简单的 iOS 游戏，因此本书最后介绍了一个疯狂连连看的实例。

本书并不局限于介绍 iOS 编程的各种理论知识，而是从“项目驱动”的角度来讲授理论，全书一共包括近百个实例，这些示范性的实例既可帮助读者更好地理解各知识点在实际开发中的应用，也可供读者在实际开发时作为参考、拿来就用。如果读者在阅读本书时遇到了技术问题，可以登录疯狂 Java 联盟（http://www.crazyit.org）发帖，笔者将会及时予以解答。

本书为所有打算深入掌握 iOS 编程的读者而编写，适合各种层次的 iOS 学习者和开发者阅读，也适合作为大学教育、培训机构的 iOS 教材。但如果只是想简单涉猎 iOS 应用开发，则本书过于庞大，不适合阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

疯狂 iOS 讲义. 上，Objective-C 2.0 与 iPhone/iPad 应用开发基础 / 李刚编著. —2 版. —北京：电子工业出版社，2015.4
ISBN 978-7-121-25613-4

Ⅰ. ①疯… Ⅱ. ①李… Ⅲ. ①移动终端－应用程序－程序设计②C 语言－程序设计 Ⅳ. ①TN929.53②TP312

中国版本图书馆 CIP 数据核字（2015）第 040515 号

策划编辑：张月萍
责任编辑：葛　娜
印　　刷：北京京科印刷有限公司
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×1092　1 /16　　印张：44.75　　字数：1293千字　　彩插：2
版　　次：2015年4月第1版
印　　次：2015年4月第1次印刷
印　　数：5000册　　定价：99.00元（含光盘1张）

CHAPTER

6

第 6 章

面向对象（下）

本章要点

- 包装类及其用法
- description 方法的功能
- ==和 isEqual:的区别
- 定义类别（category）
- 类别的用法与常用功能
- 利用扩展
- 理解协议的功能
- 利用类别实现非正式协议
- 定义正式协议
- 开发遵守协议的类
- 协议与委托
- Objective-C 的异常处理
- 使用@try...@catch 处理异常
- Objective-C 动态机制与反射
- 检查对象的继承关系
- 动态调用方法的几种方式
- 对象的引用计数
- 手动管理引用计数
- 自动释放池的功能与用法
- 手动内存管理的规则总结
- 自动引用计数

除了第 5 章介绍的关于类、对象的基本语法外，本章将继续介绍 Objective-C 面向对象的高级特性，包括 Objective-C 提供的包装类，对象的 description、isEqual:方法等内容。重点介绍 Objective-C 的类别、扩展、协议等高级语法，会详细介绍关于类别、扩展的语法，以及类别在 Objective-C 语言中的功能和用法。Objective-C 的协议相当于其他语言中接口的概念，协议体现了多个类必须遵守的规范，而且协议与委托是 iOS 开发的重要内容，因此希望读者对这部分知识加以重视。

本章还会介绍 Objective-C 的异常机制，Objective-C 提供了@try、@catch、@finally、@throw 关键字来支持异常机制。也将重点介绍 Objective-C 的反射和动态机制，这些是 Objective-C 编程的高级内容，一旦读者真正掌握了 Objective-C 的动态调用机制，就可以开发更加通用的程序了。

本章的难点是 Objective-C 的内存管理，Objective-C 采用引用计数机制来管理对象的回收，当一个对象的引用计数为 0 时，系统会自动回收该对象。本章将会详细介绍 Objective-C 提供的手动引用计数和自动引用计数两种机制。虽然自动引用计数简单、易用，但如果读者需要开发兼容早期版本 iOS 的应用，那就只能使用手动引用计数；如果只需要开发高版本的 iOS 应用，则可以直接使用自动引用计数。关于 Objective-C 内存回收的内容，既是本章的重点，也是难点，需要读者多花一些时间和精力才能真正掌握它们。

6.1 Objective-C 的包装类

Objective-C 是面向对象的编程语言，但它也是从 C 语言扩展而来的，而 C 语言中包括的基本数据类型（如 short、int、float、double 等）都不是对象，它们也不具备“对象”的特性——没有属性、方法可以被调用。

Objective-C 提供了 NSValue、NSNumber 来封装 C 语言的基本类型，这样就可以让它们具有面向对象的特征。

6.1.1 它们不是包装类

在进行 iOS 开发时，可能会遇到如下 3 个类型。

- NSInteger：大致等于 long 型整数。
- NSUInteger：大致等于 unsigned long 型整数。
- CGFLoat：在 64 位平台上大致相当于 double，在 32 位平台上大致相当于 float。

读者不要被这 3 个类型前面的 NS、CG 前缀迷惑了，它们并不是包装类，它们依然只是基本类型。

从系统文件中查看 NSInteger 和 NSUInteger 的定义代码如下：

```
#if __LP64__ || (TARGET_OS_EMBEDDED && !TARGET_OS_IPHONE)
    || TARGET_OS_WIN32 || NS_BUILD_32_LIKE_64
typedef long NSInteger;
typedef unsigned long NSUInteger;
#else
typedef int NSInteger;
typedef unsigned int NSUInteger;
#endif
```

从上面的定义可以看出，在 64 位平台和类似于 64 位的各种平台上，NSInteger、NSUInteger 都相当于 long、unsigned long。为了更好地兼容不同的平台，当程序需要定义整型变量时，建

议使用 NSInteger、NSUInteger。

从系统文件中查看 CGFloat 的定义代码如下：

```
#if defined(__LP64__) && __LP64__
# define CGFLOAT_TYPE double
# define CGFLOAT_IS_DOUBLE 1
# define CGFLOAT_MIN DBL_MIN
# define CGFLOAT_MAX DBL_MAX
#else
# define CGFLOAT_TYPE float
# define CGFLOAT_IS_DOUBLE 0
# define CGFLOAT_MIN FLT_MIN
# define CGFLOAT_MAX FLT_MAX
#endif

/* Definition of the `CGFloat' type and `CGFLOAT_DEFINED'. */

typedef CGFLOAT_TYPE CGFloat;
#define CGFLOAT_DEFINED 1
```

从上面定义的代码中可以看出，在 64 位平台上，CGFloat 相当于 double 类型；在 32 位平台上，CGFloat 相当于 float 类型。为了更好地兼顾不同的平台，当程序需要定义浮点型变量时，建议使用 CGFLoat。

6.1.2 NSValue 和 NSNumber

NSValue 和 NSNumber 都是包装类，其中 NSValue 是 NSNumber 的父类；NSValue 代表一个更通用的包装类，它可用于包装单个 short、int、long、float、char、指针、对象 id 等数据项，通过该包装类，就可以把 short、int、long、float、char、指针等添加到 NSArray、NSSet 等集合（这些集合要求它们的元素必须是对象）中。

NSNumber 是更具体的包装类，主要用于包装 C 语言的各种数值类型，NSNumber 主要包括如下 3 类方法。

- + numberWithXxx:：该类方法直接将特定类型的值包装成 NSNumber。
- - initWithXxx:：该实例方法需要先创建一个 NSNumber 对象，再用一个基本类型的值来初始化 NSNumber。
- - xxxValue：该实例方法返回该 NSNumber 对象包装的基本类型的值。

上面 3 类方法中的 xxx 可以代表 int、char 等各种基本类型，具体可以代表哪些类型，读者可以参考 Xcode 的帮助系统。

在上面 3 类方法中，前两类方法的功能基本相似，都是把一个基本类型的值包装成包装类的实例，相比之前直接用类方法 numberWithXxx 更加简单。第 3 类方法则用于从包装类中取出基本类型的值。

下面的程序示范了基本类型的值与包装类之间的转换。

程序清单：codes/06/6.1/NSNumberTest.m

```
#import <Foundation/Foundation.h>

int main(int argc , char * argv[])
{
    @autoreleasepool{
        // 调用类方法将 int 类型的值包装成 NSNumber 对象
        NSNumber* num = [NSNumber numberWithInt:20];
        // 调用类方法将 double 类型的值包装成 NSNumber 对象
        NSNumber* de = [NSNumber numberWithDouble: 3.4];
        NSLog(@"%d" , [num intValue]);
```

```
        NSLog(@"%g" , [de doubleValue]);
        // 先创建 NSNumber 对象，再调用 initWithXxx:方法执行初始化
        NSNumber* ch = [[NSNumber alloc] initWithChar:'J'];
        // 直接输出 NSNumber 对象，使用%@格式字符串
        NSLog(@"%@" , ch);
    }
}
```

从上面程序的粗体字代码中可以看到，将基本类型的值转换为包装类简单的做法就是调用 numberWithXxx:类方法，调用该方法时传入一个基本类型的值，该方法就会返回包装该值的包装类实例。如果需要从包装类实例中获取基本类型的值，则调用 xxxValue 实例方法即可。

通过上面的程序不难看出，基本类型变量和包装类对象之间的转换关系如图 6.1 所示。

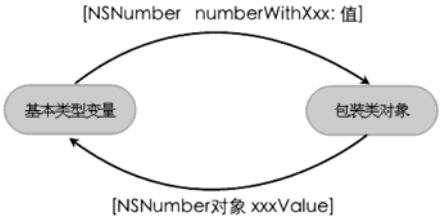

图 6.1　基本类型变量和包装类对象之间的转换关系

提示： 虽然 Objective-C 也提供了类似于自动装箱的机制，比如，可以直接把一个整型值赋给 NSNumber 变量。但这种机制并不完善，使用自动装箱生成的 NSNumber 不支持 ARC，而且不能把浮点数赋给 NSNumber 类型的变量。因此，通常建议显式地将基本类型的值包装成 NSNumber 对象。

6.2　处理对象

Objective-C 对象都是 NSObject 子类的实例，都可直接调用该类中定义的方法，这些方法提供了处理 Objective-C 对象的通用方法。

6.2.1　打印对象和 description 方法

NSLog()函数不仅可用于输出基本类型的值，也可用于输出 Objective-C 对象，当使用 NSLog()函数输出 Objective-C 对象时，输出的是该对象的 description 方法的返回值。也就是说，下面两行代码的效果完全一样（假设 p 是指向任何对象的指针变量）。

```
NSLog(@"%@" , p);
NSLog(@"%@" , [p description]);
```

description 方法是 NSObject 类的一个实例方法，所有的 Objective-C 类都是 NSObject 类的子类，因此，所有的 Objective-C 对象都具有 description 方法。

description 方法是一个非常特殊的方法，它是一个“自我描述”方法，该方法通常用于实现这样一个功能：当程序员直接打印该对象时，系统将会输出该对象的“自我描述”信息，用以告诉外界该对象具有的状态信息。

NSObject 类提供的 description 方法总是返回<FKPerson: 十六进制的首地址>，这个返回值并不能真正实现“自我描述”的功能，因此，如果用户需要自定义类实现“自我描述”的功能，则必须重写 NSObject 类的 description 方法。

大部分时候，重写 description 方法总是可以返回该对象所有令人感兴趣的信息所组成的字符串，通常可返回如下格式的字符串：

```
<类名[实例变量 1=值 1, 实例变量 2=值 2,...]>
```

例如，如下程序将会定义一个 FKApple 类，该类将会重写 NSObject 提供的 description 方法。下面是 FKApple 类的接口部分代码。

程序清单：codes/06/6.2/FKApple.h

```
#import <Foundation/Foundation.h>

@interface FKApple : NSObject
@property (nonatomic , copy) NSString* color;
@property (nonatomic , assign) double weight;
- (id) initWithColor: (NSString*) color weight: (double) weight;
@end
```

上面 FKApple 类的接口部分只是定义了 color、weight 两个属性，并定义了一个 initWithColor: weight:初始化方法，接下来为该 FKApple 类提供实现部分。在 FKApple 的实现部分，让 FKApple 类重写 NSObject 的 description 方法。下面是 FKApple 类的实现部分代码。

程序清单：codes/06/6.2/FKApple.m

```
#import "FKApple.h"

@implementation FKApple
- (id) initWithColor: (NSString*) color weight:(double) weight
{
    if(self = [super init])
    {
        self.color = color;
        self.weight = weight;
    }
    return self;
}
// 重写父类的 decription 方法
- (NSString*) description
{
    // 返回一个字符串
    return [NSString stringWithFormat:@"<FKApple[_color=%@, _weight=%g]>"
        , self.color , self.weight];
}
@end
```

当程序重写了 FKApple 的 description 方法之后，接下来输出 FKApple 对象时，程序实际上输出的是 FKApple 对象的 description 方法的返回值。

下面的程序测试了输出 FKApple 对象的效果。

程序清单：codes/06/6.2/FKAppleTest.m

```
#import <Foundation/Foundation.h>
#import "FKApple.h"

int main(int argc , char * argv[])
{
    @autoreleasepool{
        // 创建一个 FKApple 对象，将之赋给 a 变量
        FKApple* a = [[FKApple alloc] initWithColor:@"红色"
            weight:5.68];
        // 打印 FKApple 对象
        NSLog(@"%@" , a);
```

```
    }
}
```

编译、运行上面的程序，可以看到如下运行结果：

```
<FKApple[_color=红色, _weight=5.68]>
```

从上面的运行结果可以看出，通过重写 FKApple 类的 description 方法，就可以让系统在打印 FKApple 对象时打印出该对象的“自我描述”信息。

这个 description 方法提供了足够的有效信息来描述这个 FKApple 对象，也就实现了 description 方法的作用：对该对象进行自我描述。

6.2.2 ==和 isEqual:方法

在 Objective-C 程序中测试两个变量是否相等有两种方式：一种是利用==运算符；另一种是利用 isEqual:方法。

当使用==来判断两个变量是否相等时，如果两个变量是基本类型的变量，且都是数值型（不一定要求数据类型严格相同），则只要两个变量的值相等，使用==判断就将返回真。

对于两个指针类型的变量，它们必须指向同一个对象（也就是两个指针变量保存的内存地址相同）时，使用==判断才会返回真。当使用==比较类型上没有继承关系的两个指针变量时，编译器会提示警告。下面程序示范了使用==来判断两种类型变量是否相等的结果。

程序清单：codes/06/6.2/EqualTest.m

```
#import <Foundation/Foundation.h>

int main(int argc , char * argv[])
{
    @autoreleasepool{
        int it = 65;
        float fl = 65.0f;
        NSLog(@"65 和 65.0f 是否相等？: %d" , (it == fl));  // 将输出 1 代表真
        char ch = 'A';
        NSLog(@"65 和'A'是否相等？%d" , (it == ch));  // 将输出 1 代表真
        NSString* str1 = [NSString stringWithFormat:@"疯狂 iOS 讲义"];
        NSString* str2 = [NSString stringWithFormat:@"疯狂 iOS 讲义"];
        NSLog(@"str1 和 str2 是否相等？%d"
            , (str1 == str2));   // 将输出 0 代表假
        NSLog(@"str1 是否 isEqual: str2？%d"
            , [str1 isEqual:str2]);  // 将输出 1 代表真
        // 由于 NSDate 与 NSString 类没有继承关系，
        // 所以下面的语句导致编译警告
        NSLog(@"%d" , [NSDate new] == [NSString new]);
    }
}
```

运行上面的程序，可以看到 65、65.0f 和'A'相等。但对于 str1 和 str2，因为它们都是指针类型变量，它们分别指向两个通过 NSString 的类方法创建出来的 NSString 对象，因此 str1 和 str2 两个变量不相等。

对初学者而言，NSString 还有一个非常容易迷惑的地方：@"疯狂 iOS 讲义"直接量和[NSString stringWithFormat:@"hello"]有什么区别呢？当 Objective-C 程序直接使用形如“@"疯狂 iOS 讲义"”的字符串直接量时，系统将会使用常量池来管理这些字符串。

下面的程序示范了系统使用常量池管理字符串直接量的情形。

程序清单：codes/06/6.2/NSStringCompareTest.m

```
#import <Foundation/Foundation.h>

int main(int argc , char * argv[])
{
    @autoreleasepool{
        // s1、s2 直接指向常量池中的"疯狂 iOS"
        NSString* s1 = @"疯狂 iOS";
        NSString* s2 = @"疯狂 iOS";
        // 看到 s1、s2 两个指针变量中保存的地址值完全相等
        NSLog(@"s1 地址：%p, s2 地址：%p" , s1 , s2);
        // 所以下面的程序输出 1 代表真
        NSLog(@"s1 与 s2 是否相等：%d", (s1 == s2));
        // 让 s3 指向新生成的对象
        NSString* s3 = [NSString stringWithFormat:@"疯狂 iOS"];
        // 输出 s3 指针变量中保存的地址值与 s1、s2 并不相同
        NSLog(@"s3 地址：%p" , s3);
        // 所以下面的程序输出 0 代表假
        NSLog(@"s1 与 s3 是否相等：%d" , (s1 == s3));
    }
}
```

常量池保证相同的字符串直接量只有一个，不会产生多个副本。例子中的 s1、s2 都将指向常量池中的同一个字符串对象，因此，程序可以看到 s1、s2 两个指针变量中保存的地址值完全相同。

使用 NSString 的 stringWithFormat:类方法创建的字符串对象是运行时创建出来的，它被保存在运行时内存区（即堆内存）内，不会放入常量池中，因此，程序中的 s3 指针变量中保存的地址与 s1、s2 指针变量中保存的地址并不相同。

但在很多时候，程序判断两个指针变量是否相等时，也希望有一种类似于“值相等”的判断规则，并不严格要求两个指针变量指向同一个对象。例如，对于两个字符串变量，可能只是要求它们指向的字符串对象里包含的字符序列相同即可认为相等，此时就可以利用 NSString 对象的 isEqual:方法来判断。例如，上面程序中的[str1 isEqual:str2]将返回代表真的 1。

isEqual:方法是 NSObject 类提供的一个实例方法，因此所有的指针变量都可调用该方法来判断是否与其他指针变量相等。在默认情况下，NSObject 提供的 isEqual:方法判断两个对象相等的标准与==运算符没有区别，同样要求两个指针变量指向同一个对象才会返回真。因此，NSObject 类提供的 isEqual:方法没有太大的实际意义，如果希望采用自定义的相等标准，则可通过重写 isEqual:方法来实现。

NSString 已经重写了 NSObject 的 isEqual:方法，NSString 的 isEqual:方法判断两个字符串相等的标准是：只要两个字符串所包含的字符序列相同，通过 isEqual:比较就将返回真；否则将返回假。

注意：

重写 isEqual:方法就是提供自定义的相等标准。在极端的情况下，你可以让 FKDog 对象和 FKItem 对象相等。

下面的程序示范了重写 isEqual:方法产生 FKDog 对象和 FKItem 对象相等的情形。

该程序先定义了 FKDog 类的接口部分。

程序清单：codes/06/6.2/FKDog.h

```
#import <Foundation/Foundation.h>

@interface FKDog : NSObject
@end
```

上面的 FKDog 类并未提供更多额外的内容，仅仅相当于一个空类，接下来为 FKDog 类提供实现部分，代码如下。

程序清单：codes/06/6.2/FKDog.m

```
#import "FKDog.h"

@implementation FKDog
- (BOOL) isEqual:(id)other
{
    // 不进行判断，总是返回 YES，即 FKDog 对象与任何对象都相等
    return YES;
}
@end
```

上面程序重写了 NSObject 类的 isEqual:方法，重写该方法时并未进行任何判断，直接返回了 YES，这将导致该 FKDog 对象与任何对象比较都会返回 YES。

接下来随便定义一个 FKItem 空类，这个空类仅仅是继承 NSObject，分别提供接口部分和实现部分。由于这个类十分简单，故此处不再给出 FKItem 类的接口部分和实现部分的代码。

下面使用测试程序来测试 FKDog 对象与其他对象相等的情形。

程序清单：codes/06/6.2/FKDogTest.m

```
#import <Foundation/Foundation.h>
#import "FKDog.h"
#import "FKItem.h"

int main(int argc , char * argv[])
{
    @autoreleasepool{
        FKDog* dog = [[FKDog alloc] init];
        NSLog(@"FKDog 对象是否 isEqual: FKItem 对象？%d"
            , [dog isEqual: [FKItem new]]);
        NSLog(@"FKDog 对象是否 isEqual: NSString 对象？%d"
            , [dog isEqual: [NSString stringWithFormat:@"Hello"]]);
    }
}
```

编译、运行上面的程序，将可看到 FKDog 对象和 FKItem 对象相等，FKDog 对象和 NSString 对象也相等的“荒唐结果”。造成这种结果的原因是由于重写 FKDog 类的 isEqual:方法时没有任何判断、无条件地返回 YES。实际上，这种结果也不算太荒唐，因为 FKDog 和 FKItem 对象也不是完全不可能相等，这要看关心的角度，比如，仅仅关心 FKDog 对象和 FKItem 对象的标识符，从标识符的角度来看，可以认为只要 FKDog 对象与 FKItem 对象的标识符相等，FKDog 对象和 FKItem 对象就相等。

当然大部分时候，并不希望出现 FKDog 对象和 FKItem 对象相等的“荒唐局面”，还是希望两个类型相同的对象才可能相等，而且必须两个对象的关键属性相等才算相等。看下面重写 FKUser 类的 isEqual:方法，更符合实际情况。

下面先定义 FKUser 类的接口部分。

程序清单：code\06\6.2\FKUser.h

```
#import <Foundation/Foundation.h>
```

```
@interface FKUser : NSObject
@property (nonatomic , copy) NSString* name;
@property (nonatomic , copy) NSString* idStr;
- (id) initWithName: (NSString*) name
   idStr: (NSString*) idStr;
@end
```

上面定义了 FKUser 类，该类包含两个属性：name 和 idStr。假如系统业务要求：只要两个 FKUser 的 idStr 相等，即可认为两个 FKUser 对象相等。

下面为 FKUser 类提供实现部分，实现部分将会重写 isEqual:方法，重写该方法将以 idStr 作为唯一的判断标准：当两个 FKUser 的 idStr 相等时，即可认为两个 FKUser 相等。下面是 FKUser 的实现部分代码。

程序清单：code\06\6.2\FKUser.m

```
#import "FKUser.h"

@implementation FKUser
- (id) initWithName: (NSString*) name idStr: (NSString*) idStr
{
   if(self = [super init])
   {
      self.name = name;
      self.idStr = idStr;
   }
   return self;
}
// 重写 isEqual:方法，提供自定义的相等标准
- (BOOL) isEqual: (id) other
{
   // 如果两个对象为同一个对象
   if (self == other)
      return YES;
   // 当 other 不为 nil，且它是 FKUser 类的实例时
   if (other != nil && [other isMemberOfClass:FKUser.class])
   {
      FKUser* target = (FKUser*)other;
      // 并且当前对象的 idStr 与 target 对象的 idStr 相等才可判断两个对象相等
      return [self.idStr isEqual: target.idStr];
   }
   return NO;
}
@end
```

对于上面的 FKUser 类，使用如下程序来测试它。

程序清单：code\06\6.2\FKUserTest.m

```
#import <Foundation/Foundation.h>
#import "FKUser.h"

int main(int argc , char * argv[])
{
   @autoreleasepool{
      FKUser* p1 = [[FKUser alloc] initWithName:@"孙悟空"
         idStr: @"12343433433"];
      FKUser* p2 = [[FKUser alloc] initWithName:@"孙行者"
         idStr: @"12343433433"];
      FKUser* p3 = [[FKUser alloc] initWithName:@"孙悟饭"
         idStr: @"99933433"];
      // p1 和 p2 的 idStr 相等，所以输出代表真的 1
```

```
        NSLog(@"p1 和 p2 是否相等？%d", [p1 isEqual: p2]);
        // p2 和 p3 的 idStr 不相等，所以输出代表假的 0
        NSLog(@"p2 和 p3 是否相等？%d", [p2 isEqual: p3]);
    }
}
```

上面程序用 isEqual:判断 p1 是否等于 p2 时，将会返回代表真的 1，这是因为 p1 和 p2 所指向的两个对象的 idStr 相等；判断 p2 是否等于 p3 时，将会返回代表假的 0，这是因为 p2 和 p3 所指向的两个对象的 idStr 不相等。

上面程序重写 FKUser 类的 isEqual:方法，指定了 FKUser 对象和其他对象相等的标准：另一个对象必须是 FKUser 类的实例，且两个 FKUser 对象的 idStr 相等，即可判断两个 FKUser 对象相等。在这种判断标准下，只要两个 FKUser 对象的 idStr 相等，即可判断相等。

通常而言，正确地重写 isEqual:方法应该满足下列条件。

- 自反性：对任意 x，[x isEqual: x]一定返回真。
- 对称性：对任意 x 和 y，如果[y isEqual :x]返回真，则[x isEqual: y]也返回真。
- 传递性：对任意 x、y、z，如果有[x isEqual: y]返回真，[y isEqual: z]返回真，则[x isEqual: z]一定返回真。
- 一致性：对任意 x 和 y，如果对象中用于等价比较的关键属性没有改变，那么无论调用[x isEqual: y]多少次，返回的结果都应该保持一致，要么一直是真，要么一直是假。
- 对任何不是 nil 的 x，[x isEqual: nil]一定返回假。

NSObject 默认提供的 isEqual:只是比较对象的地址，即 NSObject 类的 isEqual:方法比较的结果与==运算符比较的结果完全相同。因此，在实际应用中常常需要重写 isEqual:方法，重写 isEqual:方法时，等价条件是由业务要求决定的，因此，isEqual:方法的实现也由业务要求来决定。

提示：

在以后的 iOS 开发中，还会看到程序中使用 isEqualToString:方法来判断两个字符串相等的代码。实际上，NSString 不仅重写了 isEqual:方法，用于判断两个字符串的字符序列是否相等，还定义了一个 isEqualToString:方法，该方法专门用于判断当前字符串与另一个字符串的字符序列是否相等。

6.3　类别与扩展

在开发项目的过程中，有时候需要为已有的类扩展一些新的行为，关于这种扩展方式，最容易想到的就是使用继承。通过继承，子类可以在父类的基础上添加新的方法，甚至重写父类已有的方法。

但有些时候，使用继承并不是最好的选择，比如希望为 NSNumber 类新增一些方法，但由于 NSNumber 实际上只是一个类簇（cluster）的前端类，例如，通过[NSNumber numberWithInt:5]方法所生成的 NSNumber 对象其实只是 NSNumber 子类的实例。这样即使为 NSNumber 派生子类也没有任何意义，派生的子类对 NSNumber 现有的子类并没有任何影响。此时就需要借助类别（category）来实现。

提示：

由于 Objective-C 并没有提供抽象类的语法支持，而在实际项目的开发中，总有需要抽象类的时候，此时就会选择定义一个父类，并以该父类派生多个子类，其他程序使用这些类时，总是面向父类编程，当程序调用父类的初始化方法、类方法来返回对象时，实际上返回的是子类的实例。这一系列的类被称为一个类簇（cluster），这个父类也就模拟了抽象类的功能。

6.3.1 类别（category）

Objective-C 的动态特征允许使用类别为现有的类添加新方法，并且不需要创建子类，不需要访问原有类的源代码。

通过使用类别即可动态地为现有的类添加新方法，而且可以将类定义模块化地分布到多个相关文件中。

类别同样由接口和实现部分组成，接口部分的语法格式如下：

```
@interface 已有类(类别名)
// 方法定义
...
@end
```

注意上面的语法格式，虽然这个语法格式看上去很像在定义类，但在类名后有一个圆括号，而且圆括号中带一个类别名。

定义类别的语法与定义类的语法存在如下差异。

- 定义类时使用的类名必须是该项目中没有的类，而定义类别时使用的类名必须是已有的类。
- 定义类别时必须使用圆括号来包含类别名。
- 类别中通常只定义方法。

类别实现部分的语法格式如下：

```
@implmentation 已有类(类别名)
// 方法实现
...
@end
```

下面的程序考虑为 NSNumber 增加一个类别，类别接口部分的代码如下。

程序清单：codes/06/6.3/NSNumber+fk.h

```
// 定义一个类别
@interface NSNumber (fk)
// 在类别中定义 4 个方法
- (NSNumber*) add: (double) num2;
- (NSNumber*) substract: (double) num2;
- (NSNumber*) multiply: (double) num2;
- (NSNumber*) divide: (double) num2;
@end
```

就编程习惯来说，一般习惯将类别的接口文件命名为“类名+类别名.h”的形式，与此类似，类别实现部分的文件通常命名为“类名+类别名.m”的形式。

有了上面的类别接口文件之后，接下来为该类别提供实现部分，代码如下。

程序清单：codes/06/6.3/NSNumber+fk.m

```
#import "NSNumber+fk.h"

// 为类别提供实现部分
```

```
@implementation NSNumber (fk)
// 实现类别接口部分定义的 4 个方法
- (NSNumber*) add: (double) num2
{
   return [NSNumber numberWithDouble:
      ([self doubleValue] + num2)];
}
- (NSNumber*) substract: (double) num2
{
   return [NSNumber numberWithDouble:
      ([self doubleValue] - num2)];
}
- (NSNumber*) multiply: (double) num2
{
   return [NSNumber numberWithDouble:
      ([self doubleValue] * num2)];
}
- (NSNumber*) divide: (double) num2
{
   return [NSNumber numberWithDouble:
      ([self doubleValue] / num2)];
}
@end
```

上面为 NSNumber 定义了 fk 类别，接下来只要在程序中导入 NSNumber+fk.h 头文件，并在测试程序中使用 NSNumber 类，该类的实例就会具有 add:、substract:、multiply:和 divide:方法，这就实现了对原有 NSNumber 类的动态扩展。

使用如下程序来测试 NSNumber 的 fk 类别。

程序清单：codes/06/6.3/NSNumberTest.h

```
#import <Foundation/Foundation.h>
#import "NSNumber+fk.h"

int main(int argc , char * argv[])
{
   @autoreleasepool{
      NSNumber* myNum = [NSNumber numberWithInt: 3];
      // 测试 add:方法
      NSNumber* add = [myNum add:2.4];
      NSLog(@"%@" , add);
      // 测试 substract:方法
      NSNumber* substract = [myNum substract:2.4];
      NSLog(@"%@" , substract);
      // 测试 multiply:方法
      NSNumber* multiply = [myNum multiply:2.4];
      NSLog(@"%@" , multiply);
      // 测试 divide:方法
      NSNumber* divide = [myNum divide:2.4];
      NSLog(@"%@" , divide);
   }
}
```

在上面的程序中首先导入了 NSNumber+fk.h 头文件，接下来程序创建了 NSNumber 对象（虽然它实际上只是 NSNumber 子类的实例），但由于程序已经通过 fk 类别对 NSNumber 类进行了动态扩展，因此，可以在测试程序中调用 add:、substract:、multiply:和 divide:方法。

编译、运行该程序，可以看到如下输出：

```
5.4
0.6000000000000001
7.199999999999999
1.25
```

就上面的运行结果来看，NSNumber 完全可以调用 add:、substract:、multiply:和 divide:方法——虽然运行结果可能与预期有些差异，这是正常的，因为浮点数在底层存储时有精度丢失。

虽然类别可以重写原有类中的方法，但通常并不推荐这么做，如果需要重写原有类的方法，更好的建议是通过原有类派生子类，然后在子类中重写父类原有的方法。

关于类别，还有如下两点说明。

- 通过类别为指定类添加新方法之后，这个新方法不仅会影响 NSNumber 类，还会影响 NSNumber 类的所有子类，每个子类都会获取类别扩展的方法。
- 可根据需要为一个类定义多个类别，不同的类别都可对原有的类增加方法定义。

类别是 Objective-C 中一个非常重要的知识，它通常有如下 3 种用法。

- 利用类别对类进行模块化设计。
- 使用类别来调用私有方法。
- 使用类别来实现非正式协议。

接下来将会详细介绍有关类别的前两种用法，至于使用类别来实现非正式协议的相关内容，将会在下一节介绍。

6.3.2 利用类别对类进行模块化设计

在前面类的设计中，可以使用*.h 文件来定义类接口部分，使用*.m 文件来定义类实现部分，但不能将类实现部分分布到多个*.m 文件中。

当某个类非常大时，如果将该类的所有实现代码放在一个文件中，将会导致这个文件非常大，以至于维护起来非常困难。如果需要将一个较大的类分模块设计，使用类别是一个不错的选择。下面将以 AppKit 提供的 NSWindow 类为例来介绍利用类别对类进行模块化设计的方式。

如果查看 NSWindow 类的参考文档，将会发现 NSWindow 类中包含了几百个方法，如果将这些方法都放在一个*.m 文件中实现，你能想象这个文件有多么庞大吗？

如果在 Xcode 中按住 command 键，然后在 NSWindow 类上单击鼠标，Xcode 将会打开 NSWindow 的头文件：NSWindow.h。接下来在该文件中可以看到 NSWindow 的如下定义：

```
@interface NSWindow : NSResponder <NSAnimatablePropertyContainer,
NSUserInterfaceValidations, NSUserInterfaceItemIdentification>
```

除了上面的接口定义之外，还可以看到如下类别：

```
@interface NSWindow(NSKeyboardUI)
@interface NSWindow (NSToolbarSupport)
@interface NSWindow(NSDrag)
@interface NSWindow(NSCarbonExtensions)
```

从上面类的接口定义和类别的接口定义来看，通过这种方式，NSWindow 可以分别提供 NSWindow.m、NSWindow+NSKeyboardUI.m、NSWindow+NSToolbarSupport.m、NSWindow+NSDrag.m 和 NSWindow+NSCarbonExtensions.m 实现文件，通过这种方式就可以对类实现按模块分布到不同的*.m 文件中，从而提高项目后期的可维护性。

6.3.3 使用类别来调用私有方法

前面在介绍定义类时提到，没有在接口部分定义而是在类实现部分定义的方法相当于私有方法，通常不允许被调用。但后面也说过，Objective-C 实际上并没有真正私有的方法，如果使用 NSObject 的 performSelector:方法来执行动态调用，则完全可以调用那些私有方法。

但如果使用 performSelector:来执行调用，则完全避开了编译器的语法检查，有时候未必

是一种好的做法。除了使用 performSelector:方法来动态调用那些私有方法之外，还可以通过类别来定义前向引用，从而实现对私有方法的调用。

例如，如下程序定义了 FKItem 的接口代码。

程序清单：codes/06/6.3/FKItem.h

```
#import <Foundation/Foundation.h>

// 定义类的接口部分
@interface FKItem : NSObject
@property (nonatomic , assign) double price;
- (void) info;
@end
```

在 FKItem 接口部分只定义了一个 info 方法，接下来为 FKItem 定义实现部分，实现部分不仅实现了该 info 方法，还额外新增了一个方法，代码如下。

程序清单：codes/06/6.3/FKItem.m

```
#import "FKItem.h"

// 为 FKItem 提供实现部分
@implementation FKItem
@synthesize price;
// 实现接口部分定义的方法
- (void) info
{
    NSLog(@"这是一个普通的方法");
}
// 类实现部分新增的方法，相当于私有方法
- (double) calDiscount:(double) discount
{
    return self.price * discount;
}
@end
```

上面的粗体字代码是 FKItem 实现部分额外新增的方法，这就是 Objective-C 所谓的“私有方法”。在通常情况下，直接通过 FKItem 对象调用 calDiscount:方法必然导致错误。例如，如下测试程序。

程序清单：codes/06/6.3/FKItemTest.m

```
#import <Foundation/Foundation.h>
#import "FKItem.h"

int main(int argc , char * argv[])
{
    @autoreleasepool{
        FKItem* item = [[FKItem alloc] init];
        item.price = 109;
        [item info];
        NSLog(@"物品打折的价格为：%g" , [item calDiscount:.75]);
    }
}
```

上面程序中调用了 FKItem 对象的 calDiscount:方法，但这个方法仅在 FKItem 类的实现部分定义，接口部分并没有定义这个方法。尝试编译该程序，将可看到编译器显示如下提示：

```
FKItemTest.m:13:45: error: no visible @interface for 'FKItem' declares the
     selector 'calDiscount:'
             NSLog(@"物品打折的价格为：%g" , [item calDiscount:.75]);
                                             ^
```

从上面的错误提示可以看出，编译器会提示：FKItem 对象不能调用 calDiscount:方法（Objective-C 编译器习惯把方法称为 selector）。

为了能在该程序中正常调用 calDiscount:方法，程序可以在该 main()函数前增加如下类别定义：

```
// 为 FKItem 定义一个类别
@interface FKItem (fk)
// 在类别中声明 calDiscount:方法
- (double) calDiscount:(double)discount;
@end
```

这样在 main()函数前定义了该类别之后，FKItem 的 fk 类别中自然就增加了 calDiscount:方法，但类别的方法并不强制程序去实现它——实际上，FKItem 类的实现部分已经实现了 calDiscount:方法。在该类别中增加定义 calDiscount:方法仅仅是告诉编译器，在 FKItem 类中可以包含 calDiscount:方法，这样编译器就不再提示上面所示的错误了。

再次编译、运行该程序，可以看到如下输出：

```
这是一个普通的方法
物品打折的价格为：81.75
```

6.3.4 扩展（extension）

扩展与类别相似，扩展相当于匿名类别，定义扩展的语法格式如下：

```
@interface 已有类 ()
{
    实例变量
}
// 方法定义
...
@end
```

从上面的语法格式中可以看出，扩展相当于定义一个匿名的类别。但就用法来看，类别通常有单独的*.h 和*.m 文件，扩展则用于临时对某个类的接口进行扩展，类实现部分同时实现类接口部分定义的方法和扩展中定义的方法。

在定义类的扩展时，可以额外增加实例变量，也可以使用@property 来合成属性（包括 setter、getter 方法和对应的成员变量），但定义类的类别时，则不允许额外定义实例变量，也不能用@property 合成属性。

例如，下面的程序先定义了 FKCar 的接口部分。

程序清单：codes/06/6.3/FKCar.h

```
#import <Foundation/Foundation.h>

// 定义类的接口部分
@interface FKCar : NSObject
@property (nonatomic , copy) NSString* brand;
@property (nonatomic , copy) NSString* model;
- (void) drive;
@end
```

上面只是定义了 FKCar 类的接口部分，在该接口部分定义了两个属性和一个方法，接下来对该类进行扩展，扩展的接口代码如下。

程序清单：codes/06/6.3/FKCar+drive.h

```
#import "FKCar.h"
```

```
// 定义 FKCar 的扩展
@interface FKCar ()
@property (nonatomic , copy) NSString* color;
- (void) drive:(NSString*)owner;
@end
```

在这个类扩展中再次新定义了一个 color 属性，并新增了 drive:方法，接下来为该 FKCar 类提供实现部分，该实现部分不仅需要实现接口部分定义的所有方法，还需要实现扩展中定义的方法。如下是实现部分的代码。

程序清单：codes/06/6.3/FKCar.m

```
#import "FKCar+drive.h"

// 为 FKCar 提供实现部分
@implementation FKCar
- (void) drive;
{
   NSLog(@"%@汽车正在路上奔驰" , self);
}
- (void) drive:(NSString*) owner
{
   NSLog(@"%@正驾驶%@汽车在路上奔驰" , owner , self);
}
- (NSString*) description
{
   return [NSString stringWithFormat:@"<FK[_brand=%@,_model=%@,_color=%@]>"
      , self.brand , self.model ,self.color];
}
@end
```

上面的实现部分实现了接口部分和扩展部分的所有方法，因此，该 FKCar 将具有接口部分、扩展部分定义的所有方法。

注意：

在该实现部分仅导入接口部分的*.h 文件是不够的，还需要导入类扩展部分的*.h 文件，因此，上面的程序使用了#import 导入 FKCar+drive.h 文件。

接下来使用如下程序测试 FKCar 类。

程序清单：codes/06/6.3/FKCarTest.m

```
#import <Foundation/Foundation.h>
#import "FKCar+drive.h"

int main(int argc , char * argv[])
{
   @autoreleasepool{
      // 创建一个 FKCar 对象
      FKCar* car = [[FKCar alloc] init];
      // 使用点语法为 car 对象的属性赋值
      car.brand = @"宝马";
      car.model = @"X5";
      car.color = @"黑色";
      // 调用 car 的方法
      [car drive];
      [car drive:@"孙悟空"];
   }
}
```

注意：

在测试 FKCar 类时，同样需要 import 类扩展的接口文件，仅导入类接口定义文件是不够的。

在上面的程序中，可以使用点语法来访问 color 属性，这个 color 属性是在扩展部分定义的。除此之外，也可调用 FKCar 对象的 drive:方法，这个方法也是在扩展部分定义的。