**iOS应用主题（图片，颜色）统一管理**

[ YxxxHao](https://www.jianshu.com/u/a4b11b398b1e) 关注

2016.08.24 19:14\* 字数 1089 阅读 2211评论 5喜欢 16

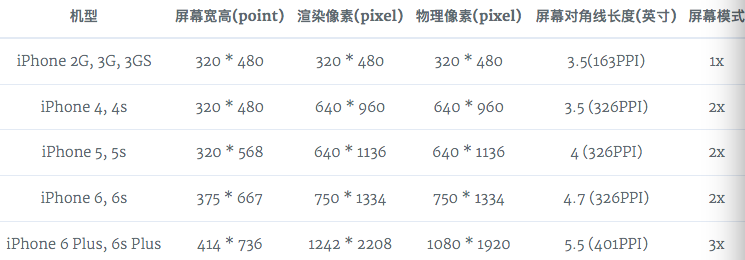
// 2017.8.24 更新

进阶篇：[再谈 Swift 换肤功能](https://www.jianshu.com/p/ea0f60dfd0f1)，看完该文后强烈建议看下进阶篇，彩蛋彩蛋彩蛋哦。

在我过去的一个多月里，发现很多不愉快的事情，导致 [OpenGL SE](https://www.jianshu.com/notebooks/4888514/latest)系列的文章好久没有更新了，今天来分享下以前做的主题管理来做一个新开始，OpenGL SE的文章会继续坚持写下去，欢迎关注。

**只需@3x图片**

现在工作改做SDK后，发现很少和界面相关的东西打交道了，但做过APP的同学们都应该知道，为了适应各种屏幕的尺寸，图片资源需要提供@1x、@2x和@3x来适配屏幕界面，现在基本没有 1x屏幕的设备了，可以不用提供这个分辨率的图片了。但@2x和@3x可以说是重复的资源，这只会增大应用包的大小。



C69924BD-2C0E-4279-A4EB-3D822C5AB340.png

在这，我只使用@3x的图片来做适配：

先写在前面，iOS 8后系统自动会将@3x图片自动适配图片，也就是说你的应用不支持iOS 8以下系统的话，你可以直接使用@3x的资源就可以了，你可以直接跳过这一节。

这里为了实现换肤功能，所有的资源我都会存在Bundle里面，首先解释下，Bundle是静态的，作为一个资源包是不参加项目编译的，也就是说，bundle包中不能包含可执行的文件，它仅仅是作为资源，被解析成为特定的2进制数据。对于在iOS 8系统上会自动将@3x的资源自动适配后，我们只需要考虑iOS 8下的系统，这个时候我们只需要手动去重新绘制图片的大小（比较消耗性能的动作），实现如下：

Swift:

private func scaledImageFrom3x() -> UIImage {

let locScale = UIScreen.mainScreen().scale

let theRate: CGFloat = 1.0 / 3.0

let oldSize = self.size

let scaleWidth = CGFloat(oldSize.width) \* theRate

let scaleHeight = CGFloat(oldSize.height) \* theRate

var scaleRect = CGRectZero

scaleRect.size.width = scaleWidth

scaleRect.size.height = scaleHeight

UIGraphicsBeginImageContextWithOptions(scaleRect.size, false, locScale)

drawInRect(scaleRect)

var newImage = UIImage()

newImage = UIGraphicsGetImageFromCurrentImageContext()

UIGraphicsEndImageContext()

return newImage

}

OC:

- (UIImage \*)scaledImageFrom3x

{

float locScale = [UIScreen mainScreen].scale;

float theRate = 2.0 / 3.0;

UIImage \*newImage = nil;

CGSize oldSize = self.size;

CGFloat scaledWidth = oldSize.width \* theRate;

CGFloat scaledHeight = oldSize.height \* theRate;

CGRect scaledRect = CGRectZero;

scaledRect.size.width = scaledWidth;

scaledRect.size.height = scaledHeight;

UIGraphicsBeginImageContextWithOptions(scaledRect.size, NO, locScale);

[self drawInRect:scaledRect];

newImage = UIGraphicsGetImageFromCurrentImageContext();

UIGraphicsEndImageContext();

if(newImage == nil) {

NSLog(@"could not scale image");

}

return newImage;

}

**换肤功能**

换肤功能，其实就是图片和颜色等资源的切换，也就是说你有几套皮肤，就提供对应的几套资源，当切换皮肤的时候，切换资源访问的路径并发出要换肤的通知，当前界面监听换肤的通知后再去刷新界面就完成了换肤的功能了。

我们实现一个ThemeManager的主题管理类，应用的所有资源访问都通过这个类来实现统一管理，所有的主题基本上都是由颜色和资源（图片，音频，文本等）来决定的，所以换肤时只要更改主题颜色库（themeColors）和主题资源库（themeBundle），实现如下：

Swift:

class CPThemeManager: NSObject {

private var themeStyle: CPThemeType?

private var themeBundle: NSBundle?

private var themeColors: Dictionary<String, AnyObject>?

*// MARK: 单例*

static let shareInstance = CPThemeManager()

private override init() {}

}

OC：

static BTThemeManager \* \_themeManager = nil;

@interface BTThemeManager () {

NSDictionary \*\_themeColors;

}

@property (nonatomic, strong) NSDictionary \*themeColors;

@property (nonatomic, strong) NSBundle \*themeBundle;

@end

@implementation BTThemeManager

@synthesize themeStyle = \_themeStyle;

@synthesize themeColors = \_themeColors;

+ (BTThemeManager \*)getInstance

{

static dispatch\_once\_t onceToken;

dispatch\_once(&onceToken, ^{

\_themeManager = [[BTThemeManager alloc]init];

});

return \_themeManager;

}

- (id) init

{

if (self = [super init]) {}

return self;

}

@end

上面也说了，换肤实质只是更换资源访问的路径，所以提供一个设置主题的方法来进行资源路径的设置（setThemeStyle），在重新设置完资源路径后，再对外发出更新界面的通知，实现如下：

Swift:

func setThemeStyle(themeStyle: CPThemeType) {

*//设置资源路径*

NSNotificationCenter.defaultCenter().postNotificationName("CPThemeChangeNotification", object: nil)

}

OC:

- (void)setThemeStyle:(BTThemeType)themeStyle

{

if (\_themeStyle == themeStyle ) {

return;

}

\_themeStyle = themeStyle;

*//设置资源路径*

[[NSNotificationCenter defaultCenter] postNotificationName:BTThemeChangeNotification object:nil];

}

下面将说整个主题管理功能的最重要一步：监听主题的切换。

首先定义一个需要更新主题的协议方法：

Swift:

protocol CPThemeListenerProtocol {

func CPThemeDidNeedUpdateStyle() -> Void

}

OC:

@protocol BTThemeListenerProtocol <NSObject>

- (void) BTThemeDidNeedUpdateStyle;

@end

然后在主题管理理里面添加一个注册监听主题切换方法：

Swift:

func addThemeListener(object: CPBaseViewController) {

NSNotificationCenter.defaultCenter().addObserver(object,

selector:#selector(object.CPThemeDidNeedUpdateStyle),

name: "CPThemeChangeNotification",

object: nil)

}

func removeThemeListener(object: AnyObject) {

NSNotificationCenter.defaultCenter().removeObserver(object)

}

因为Swift selector现在只能通过类名.方法名来设置，导致如果要使用则必须要继承一个基类，如果你们有更好的方法，求分享下。

OC：

- (void)addThemeListener:(id )obj

{

if([obj respondsToSelector:@selector(BTThemeDidNeedUpdateStyle)]){

[[NSNotificationCenter defaultCenter] addObserver:obj selector:@selector(BTThemeDidNeedUpdateStyle) name:BTThemeChangeNotification object:nil];

}

}

- (void) removeThemeListener:(id)obj

{

if (obj) {

[[NSNotificationCenter defaultCenter] removeObserver:obj];

}

}

最后在要实现主题切换的页面里添加主题管理类的监听切换方法，并实现协议的方法，把需要做主题切换的资源访问都放在这个方法里面，然后就搞定啦。搞了？好像少了点什么，还没有说如何去访问资源呢，这个我想大家都能自己去去实现，就是在基类里实现一个统一访问资源的方法：

- (void )BTThemeImage:(NSString \*)imageName completionHandler:(void (^)(UIImage \*image))handler;

{

dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0), ^{

*// 耗时的操作*

NSString \*imagePath = [NSString stringWithFormat:@"image/%@",imageName];

UIImage \*image = nil;

*//通过资源路径去访问*

if (image == nil) {

image = [UIImage imageNamed:imageName];

}

dispatch\_async(dispatch\_get\_main\_queue(), ^{

*// 更新界面*

handler(image);

});

});

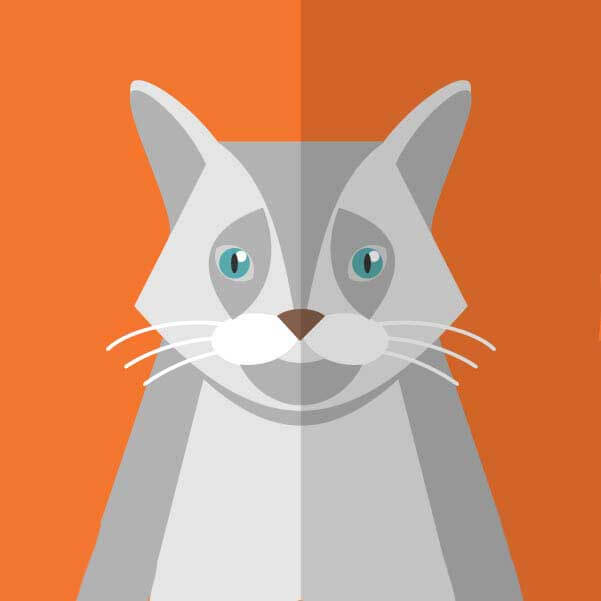
}

写在最后：欢迎大家一起多交流多学习，有更好的想法实现什么的， 求分享~~~

<https://github.com/fengfengaima/ColorChangesForVC>

<https://github.com/yyj2013/simpleTheme>

**iOS APP 开发中的主题切换设计思路**

[ 翻滚着的牛宝宝](https://www.jianshu.com/u/7e654466b422) 关注

2016.06.04 19:05\* 字数 1458 阅读 2115评论 3喜欢 9

**主题切换通知变化的方式**

* 说到主题切换，那么就要做到切换主题瞬间，使所有相关的界面都发生变化，这就需要一种机制来将主题切换这个事件抛出来，并且接受主题切换事件的相关page(View)做出相应改变。看到这里，你可定也想到了NSNotification。没错，这是一个不错的选择，很适合我们的场景。
* 那还有没有其他好的方式呢？答案当然是有的，另一个类似的机制就是KVO。主题系统中一定需要一个单例来存储当前的主题状态，我们就可以去使用系统内建的KVO方式来观察这个主题状态，状态切换时，每个观察者就能够拿到这个事件去做一些处理。
* 第三种思路就是使用delegate。我们可以在主题管理Manager的单件中提供注册delegate的方法，将所有设置的delegate保存在一个list中，这样就可以在主题状态切换时候，遍历delegate去通知。

以上三种思路各有优劣，具体分析如下：

|  |  |  |
| --- | --- | --- |
| **通知方式** | **优点** | **缺点** |
| NSNotification方式 | 不需要自己管理观察者序列 | main线程同步方式，性能欠佳 |
| KVO方式 | 不需要自己管理观察者序列 | main主线程同步方式，性能上欠佳 |
| 自己管理delegate方式 | 可以做一些优化，如非顶层View可延后通知，通知时采用异步通知 | 需要自己维护观察者队列 |

当你看到这里，可能比较疑惑，该如何选择呢？事实上，NSNotification和KVO方式没有太大的差异，可能存在的问题就是多层级VIEW叠加是是否会出现性能问题，导致APP出现ANR。第三种方式可以在某种程度上解决这个问题。因为自己管理delegate的话可以控制是否有必要通知，而且可以在异步线程去通知，性能上有所优化空间。但是，虽然可以异步线程去通知，但我们主题切换一般是UI层级的操作，也就是必须要在主线程操作，so，个人认为异步线程通知，主线程修改UI，相较前两种方式**优化效果有限**。

对比了这么多，最终我实现使用了NSNotification方式，后面会贴出源码。

**主题切换通知对象的讨论**

按照我们的一般设计思想，UIView只处理UI相关的绘制而不去处理逻辑，这一点是大多数设计模式所要遵从的设计思路。那么，按照这个思路，我们设计主题切换思路大体如下:

在管理主题管理Manager中保存当前主题的状态，当主题发生变化时候，post出Notification。在我们在ViewController的基类里面接收这个通知，所有需要在主题切换时发生变化的ViewController只需要复写这个方法(姑且叫它onGetThemeSwitch:(GNThemeState \*)state)，在这个方法中更改View的样式即可。设计合情合理，符合大多数的设计规范。

**看起来没什么问题。但是在具体实现起来，发现问题不小。**

* 首先，ViewContrller中View一般较多，改变起来会导致onGetThemeSwitch逻辑相当复杂，逻辑不清晰\*
* 其次，更要命的是很多ViewController中的View很多层次比较深，例如ViewController中有一个HeaderView，HeaderView中又有一个CtlView(用来盛放操作按钮),CtlView有可能有一个BackView,BackView中又有一个UIImageView(啊，看着就头疼)，如果要在主题切换时改变UIImageView，那么面临的问题就是：  ViewController--->CtlView--->BackView--->UIImageView

这么长一个通知链。估计写起代码来会忍不住吐槽。同时代码的可维护性也是一个很大的问题。

**基于以上问题，我改变了设计思路，决定采用View主动接受通知。因此想到了对UIView做手脚，为UIView搞一个主题扩展，下面直接上代码:**

首先看看头文件:

*//*

*// UIView+DayNight.h*

*// GameNews*

*//*

*// Copyright © 2016年 youxibar. All rights reserved.*

*//*

#import <UIKit/UIKit.h>

#import "GNDayNightManager.h"

*/\*\**

*\* 主题切换block*

*\**

*\* @param state 当前主题状态(GNDayNightState)*

*\*/*

typedef void (^UIViewDayNight\_themeChangeBlock)(GNDayNightState state);

*/\*\**

*\* 对UIView进行扩展，支持主题切换时两种回调方式*

*\* 方式一: 子类重写dn\_onDayNightStateHasChange:方法，该方法在主题变化时会被调用,注此方式需要主动设置self.dn\_isNeedTheme=YES*

*\* 方式二: 注册block dn\_setThemeChangeBlock:,主题变化时回调block*

*\*/*

@interface UIView (DayNight)

*/\*\**

*\* 设置背景色的ID*

*\*/*

@property (nonatomic, strong) NSString \*dn\_backgroundColorID;

*/\*\**

*\* 是否注册主题通知，YES == 注册主题通知，主题切换时*

*\* dn\_onDayNightStateHasChange: 会被调用*

*\*/*

@property (nonatomic, assign) BOOL dn\_isNeedTheme;

*/\*\**

*\* 子类通过复写该方法来做主题切换相关操作(切换图片，改变颜色等)*

*\**

*\* @param state 主题状态*

*\*/*

- (void)dn\_onDayNightStateHasChange:(GNDayNightState) state;

*/\*\**

*\* 注册主题变化的block*

*\**

*\* @param themeChangeBlock 主题切换block*

*\*/*

- (void)dn\_setThemeChangeBlock:(UIViewDayNight\_themeChangeBlock)themeChangeBlock;

@end

**我们可以看到，我们为外界提供了两种主题通知方式：**

* 覆盖dn\_onDayNightStateHasChange:父类这一方法，在主题变化时候会回调此方法；
* 采用dn\_setThemeChangeBlock这种方式，主题变化时block会被回调。这种方式是为了解决直接使用系统UIView及子类时无法使用上面第一种方法而添加的一种方式。此方式的另一好处是代码看起来比较紧凑。

**再看下实现文件**

*//*

*// UIView+DayNight.m*

*// GameNews*

*//*

*// Created by baidu on 16/5/27.*

*// Copyright © 2016年 youxibar. All rights reserved.*

*//*

#import "UIView+DayNight.h"

#import <objc/runtime.h>

static int gn\_dn\_backgroundColorId; *//backgroundColorID*

static int gn\_dn\_isNeedThemeId; *//是否需要注册主题ID*

static int gn\_dn\_isHasRegistNotifId; *//标识是否已经注册主题切换通知ID*

static int gn\_dn\_themeChangeBlockId; *//主题切换blockID*

@implementation UIView (DayNight)

*/\*\**

*\* 交换dealloc函数，需要在dealloc之前remove通知*

*\*/*

+ (void)load {

NSString \*className = NSStringFromClass(self.class);

NSLog(@"classname %@", className);

static dispatch\_once\_t onceToken;

dispatch\_once(&onceToken, ^{

Class class = [self class];

*//为了在dealloc之前去除theme通知，交换dealloc函数*

SEL originalDealloc = NSSelectorFromString(@"dealloc");

SEL swizzledDealloc = @selector(gn\_dealloc);

Method origMethod = class\_getInstanceMethod(class, originalDealloc);

Method swizzMethod = class\_getInstanceMethod(class, swizzledDealloc);

method\_exchangeImplementations(origMethod, swizzMethod);

});

}

- (void)gn\_dealloc {

if (self.dn\_isHasRegistNotif) {

[[NSNotificationCenter defaultCenter] removeObserver:self name:GNDAYNIGHT\_STATE\_CHANGE object:nil];

}

[self gn\_dealloc];

}

#pragma mark - add property

*//为UIView提供基础的主题背景色设置*

- (NSString \*)dn\_backgroundColorID {

return objc\_getAssociatedObject(self, &gn\_dn\_backgroundColorId);

}

- (void)setDn\_backgroundColorID:(NSString \*)colorID {

objc\_setAssociatedObject(self, &gn\_dn\_backgroundColorId, colorID, OBJC\_ASSOCIATION\_RETAIN\_NONATOMIC);

self.dn\_isNeedTheme = @(YES);

[self setBackgroundColor:UIColorFromRGB([[GNDayNightManager sharedInstance] getColorWithColorID:colorID])];

}

*//标识是否需要主题，当设置为需要主题时，注册 GNDAYNIGHT\_STATE\_CHANGE 主题切换通知*

- (BOOL)dn\_isNeedTheme {

NSNumber \*isNeedThemeNum = objc\_getAssociatedObject(self, &gn\_dn\_isNeedThemeId);

return [isNeedThemeNum boolValue];

}

- (void)setDn\_isNeedTheme:(BOOL)dn\_isNeedTheme {

objc\_setAssociatedObject(self, &gn\_dn\_isNeedThemeId, @(dn\_isNeedTheme), OBJC\_ASSOCIATION\_RETAIN\_NONATOMIC);

if (dn\_isNeedTheme && !self.dn\_isHasRegistNotif) { *//未注测过通知且需要主题通知，进行注册*

[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(dn\_onDayNightStateChange:) name:GNDAYNIGHT\_STATE\_CHANGE object:nil];

} else if(!dn\_isNeedTheme && self.dn\_isHasRegistNotif){ *//设置为不需要注册，且已经注册，则把通知移除*

[[NSNotificationCenter defaultCenter] removeObserver:self name:GNDAYNIGHT\_STATE\_CHANGE object:nil];

self.dn\_isHasRegistNotif = NO;

}

}

*//标识是否已经注册通知，防止dn\_isNeedTheme被多次设置后导致同一个UIView注册多次通知*

- (BOOL)dn\_isHasRegistNotif {

NSNumber \*isHasReg = objc\_getAssociatedObject(self, &gn\_dn\_isHasRegistNotifId);

return [isHasReg boolValue];

}

- (void)setDn\_isHasRegistNotif:(BOOL)dn\_isHasRegistNotif {

NSNumber \*hasRegistNotif = @(dn\_isHasRegistNotif);

objc\_setAssociatedObject(self, &gn\_dn\_isHasRegistNotifId, hasRegistNotif, OBJC\_ASSOCIATION\_RETAIN\_NONATOMIC);

}

*//themeChangeBlock*

- (UIViewDayNight\_themeChangeBlock)dn\_themeChangeBlock {

return objc\_getAssociatedObject(self, &gn\_dn\_themeChangeBlockId);

}

#pragma mark - public

- (void)dn\_setThemeChangeBlock:(UIViewDayNight\_themeChangeBlock)themeChangeBlock {

if (themeChangeBlock) {

self.dn\_isNeedTheme = YES;

objc\_setAssociatedObject(self, &gn\_dn\_themeChangeBlockId, themeChangeBlock, OBJC\_ASSOCIATION\_RETAIN\_NONATOMIC);

}else {

self.dn\_isNeedTheme = NO;

}

}

#pragma mark - callback

- (void)dn\_onDayNightStateChange:(NSNotification\*)notification {

*//处理backgroundColor*

if ([[self dn\_backgroundColorID] length] > 0) {

UIColor \*backColor = UIColorFromRGB([[GNDayNightManager sharedInstance] getColorWithColorID:[self dn\_backgroundColorID]]);

[UIView animateWithDuration:GNCommonAnimationsTime animations:^{

[self setBackgroundColor:backColor];

}];

}

if ([self dn\_themeChangeBlock]) {

UIViewDayNight\_themeChangeBlock block = [self dn\_themeChangeBlock];

block([GNDayNightManagerInstance state]);

}

*//子类可以复写此方法*

[self dn\_onDayNightStateHasChange:[[GNDayNightManager sharedInstance] state]];

}

- (void)dn\_onDayNightStateHasChange:(GNDayNightState) state {

*//子类按需实现*

}

@end

**实现文件中主要做了以下几件事**

1. hook系统的dealloc方法，这是为了能够在dealloc之前remove notification；
2. 为UIView添加几个属性。通过关联对象，为UIView添加几个属性，主要是为了注册notification、保存block；
3. 在接到notification回调的时候，将事件派发给注册的block及dn\_onDayNightStateHasChange:方法。

**再来看看主题管理类**

**主题管理类的基础功能如下：**

* 保存当前主题状态；
* 提供主题切换的能力；
* 在主题切换时将该时间抛出去；
* 提供取image、color、font等的方法；
* 如果要支持在线下载的话还需要提供下载、解压、安装等能力。

**我们来看下代码（m文件就不贴了）：**

*//*

*// GNDayNightManager.h*

*// GameNews*

*//*

*// Created by baidu on 16/5/25.*

*// Copyright © 2016年 youxibar. All rights reserved.*

*//*

#import <UIKit/UIKit.h>

#define GNDayNightManagerInstance [GNDayNightManager sharedInstance]

extern NSString \* const GNDAYNIGHT\_STATE\_CHANGE; *//夜间<->日间模式切换通知*

*/\*\**

*\* 主题状态*

*\*/*

typedef NS\_ENUM(NSUInteger, GNDayNightState) {

GNDayNightInvalidState = 0, *//error state*

GNDayNightDayState, *//日间主题*

GNDayNightNightState, *//夜间主题*

};

*/\*\**

*\* 按钮状态*

*\*/*

typedef NS\_ENUM(NSUInteger, GNButtonImgType) {

GNButtonImgNormalType, *//按钮normal态*

GNButtonImgPressType, *//按钮press态*

};

@interface GNDayNightManager : UIImageView

@property (nonatomic, assign) GNDayNightState state;

*/\*\**

*\* 获得主题管理单例*

*\**

*\* @return 主题单例*

*\*/*

+ (GNDayNightManager\*)sharedInstance;

*/\*\**

*\* 切换当前主题*

*\*/*

- (void)switchState;

*/\*\**

*\* 设置当前主题状态*

*\**

*\* @param state (GNDayNightState)*

*\*/*

- (void)setState:(GNDayNightState)state;

*/\*\**

*\* 通过iconID及状态(GNButtonImgType)获取图片name*

*\**

*\* @param iconID iconID*

*\* @param type 所需图片状态(GNButtonImgType)*

*\**

*\* @return 图片name*

*\*/*

- (NSString \*)getIconNameWithIcon:(NSString \*)iconID type:(GNButtonImgType) type;

*/\*\**

*\* 通过colorID获得当前主题对应的Color(16进制rgb)*

*\**

*\* @param colorID colorID*

*\**

*\* @return 16进制rgb*

*\*/*

- (int)getColorWithColorID:(NSString \*)colorID;

@end

**针对page也需要接受通知**

**上面针对UIView做了处理，实际应用中，可能要配合页面处理，因此，最好在page父类同样提供主题相关方法。**

[源码地址](https://link.jianshu.com/?t=https://github.com/yyj2013/simpleTheme)

[https://github.com/yyj2013/simpleTheme](https://link.jianshu.com/?t=https://github.com/yyj2013/simpleTheme)

ios的框架设计系列——让App丰富起来 iOS App主题皮肤切换的实现

[技术](http://www.dookay.com/insight/technology) [王翔宇](http://www.dookay.com/author?userId=23) 2016-11-10 阅读：1588



发送到手机

第一篇 基本思路及实现原理

当我们进行[App开发](http://www.dookay.com/product/mobile_ios_android_app_design_develop/index.html)的时候，有没有考虑过这样一个问题？当用户长时间使用浏览这款App的时候，是否会产生审美疲劳？每天面对着一成不变的配色，会不会觉得单调？每天面对着千篇一律的UI界面，会不会觉得无趣？

如何吸引用户的眼球，抓住用户的胃口，让用户和App始终处于”蜜月期“呢？在产品设计的时候就应该考虑到，那么今天，我就来讲一下关于[iOS App](http://www.dookay.com/product/mobile_ios_android_app_design_develop/index.html)主题皮肤切换功能的实现。让你的App丰富起来！

一、基本原理

所谓App主题皮肤切换功能，就是用户可以根据自己的口味，手动切换App整体色调风格，以及具体控件的类型（如按钮图片，文字字体等）。

就iOS App而言，Controller的色调，以及各个控件的类型，实在Controller创建之初，即在Controller的生命周期 ViewDideLoad 中就已经确定了，并且在控制器的整个生命周期中是不会改变的。那么如何在App的运行过程中，动态的改变Controller的色调，以及各控件的类型呢？

我们需要在用户点击切换皮肤的时候，把这个事件告知每个需要改变风格的控制器。在iOS中最常用到的事件告知方法就是通知和KVO。在[稻壳](http://www.dookay.com/about)互联以往的案例中得到启发，由于作为一个完整的大型App项目，可能需要改变风格的控制器非常多，所以这里推荐使用通知的方法。

二、App结构搭建

上面提到，用通知可以使控制器监听并接受到更改主题皮肤这个事件并加以响应。但是在iOS中，只有注册了通知的Controller才能够监听具体的事件。如果一个App有上百个页面需要做到动态换肤，那么在上百个Controller中分别添加注册通知的代码就足够程序员崩溃的了。

所以在App开发工程创建之初，我们需要创建一个Controller的父类，暂且成为BaseController。在这个父类中，我们注册了监听通知的方法。并且在BaseController中添加了一个公用方法，用来在接收到通知后加以响应。

// 接收到通知后响应方法

- (void)configUIAppearance:(NSNotification \*)notification

{

    NSLog(@"base config ui ");

}

// 注册监听通知

#pragma mark - theme changed notification -

- (void)registerThemeChangedNotification{

    [Default\_Notification\_Center addObserver:self selector:@selector(handleThemeChangedNotification:) name:Notification\_For\_ThemeChanged object:nil];

}

// 接受到通知后的处理

- (void)handleThemeChangedNotification:(NSNotification\*)notification{

    UIImage \*navBarBackgroundImg=[[[ThemeManager sharedInstance] themedImageWithName:@"themeColor.png"] resizableImageWithCapInsets:UIEdgeInsetsMake(0.0f, 0.0f, 1.0f, 1.0f) resizingMode:UIImageResizingModeTile];

    [self.navigationController.navigationBar setBackgroundImage:navBarBackgroundImg forBarMetrics:UIBarMetricsDefault];

    [self configUIAppearance:notification];

}

在创建好父类之后，那么在App之后的开发过程中，但凡遇到需要动态换肤的Controller，我们就继承与这个父类。自然地，这个子类Controller也就注册了监听通知的方法。我们只需要在子类里面重写父类的-(void)configUIAppearace:(NSNotification \*)notification这个方法，就可以对Controller里的控件进行动态改变。

// 继承于父类 BaseController

@interface XYUserCenterController : BKBaseController

// 重写父类方法改变控制器背景颜色

- (void)configUIAppearance:(NSNotification \*)notification

{

    self.headBgV.backgroundColor = [BKTool transformColorToUIColor:MAINCOLOR];

}

三、发送换肤通知

在App中，当用户点击了更换主题皮肤的按钮之后，需要通过消息中心对所有注册了该通知的控制器发送通知。是其监听到更换主题皮肤这个通知，并调用响应方法做出对应的改变。

// 发送通知

 [[NSNotificationCenter defaultCenter] postNotificationName:Notification\_For\_ThemeChanged object:siteDic];

四、Controller对通知进行响应

当控制器接收到换肤的通知后，即可在重写的父类方法中对Controller中的控件进行需要的改变啦。

不论是按钮的背景图片，文字的颜色，还是Navigation或者TabBar的风格。甚至可以改变整个Controller的风格。在这个方法里你可以随心所欲，天马行空。

- (void)configUIAppearance:(NSNotification \*)notification

{

    // 修改按钮字体颜色

    [\_mainSiteButton setTitleColor:[BKTool transformColorToUIColor:MAINCOLOR] forState:UIControlStateSelected];

    [\_myConcernButton setTitleColor:[BKTool transformColorToUIColor:MAINCOLOR] forState:UIControlStateSelected];

    [\_allActivityButton setTitleColor:[BKTool transformColorToUIColor:MAINCOLOR] forState:UIControlStateSelected];

    [\_allDataButton setTitleColor:[BKTool transformColorToUIColor:MAINCOLOR] forState:UIControlStateSelected];

    \_bottomLineView.backgroundColor = [BKTool transformColorToUIColor:MAINCOLOR];

    // 修改顶部轮播图指示器颜色

    self.bannerSV.currentPageDotColor = [BKTool transformColorToUIColor:MAINCOLOR];

    self.mainSitebannerSV.currentPageDotColor = [BKTool transformColorToUIColor:MAINCOLOR];

    // 修改导航栏球队Logo

    NSString \*teamLogo = [[BKTool convertJsonStringToDict:SITELOGO][0] valueForKey:@"file"];

    [self.teamIV setImageWithURL:[NSURL URLWithString:[NSString stringWithFormat:@"%@%@",APIPrefix,teamLogo]] forState:UIControlStateNormal placeholderImage:[UIImage imageNamed:@"placeholder"]];

    //修改导航栏球队名称

    \_titleLabel.text = SITETITLE;

}

至此，你的App就可以实现更换主题皮肤的功能啦！

然而，怎么样才能从服务器获取更多丰富多彩的主题资源并加以使用呢？[稻壳洞察](http://wwww.dookay.com/insight)将会在下篇文章中为大家介绍如何从服务器下载皮肤压缩包，解压并存储资源，以及沙盒路径图片使用的方法！

### ios的框架设计系列——让App丰富起来 iOS App主题皮肤切换的实现 （二）

[技术](http://www.dookay.com/insight/technology) [王翔宇](http://www.dookay.com/author?userId=23) 2016-11-18 阅读：600

[发送到手机](http://www.dookay.com/n/12234)

**第二篇 皮肤资源下载及解压**

在上篇文章里，我向大家介绍了App主题皮肤切换功能实现的基本思路和方法。在本篇中，我将向大家介绍如何从服务器下载皮肤资源及解压使用。

关于皮肤资源的获取，有两种方法。第一种是将所有的皮肤资源在App打包的时候一同打包进App，在本地直接使用。这种方法有两个弊端：第一，皮肤资源无法即时更新，所有新增资源必须在App版本升级的时候重新打包进App,然后上传App Store。而用户也必须下载最新版本的App才能获取到新增的皮肤。第二，一次性将所有皮肤资源打包进App，无疑会增加App的容量，对于那些手机容量偏小的用户来说是非常不友好的。

我们采用的方式是从服务器端获取皮肤资源地址，然后通过网络下载到本地存储，然后加以使用。这种方法一来可以在后台即时新增皮肤资源，二来用户可以根据个人的喜好选择自己喜欢的皮肤资源，大大减小了App的容量，节省了手机的内存空间。那么如何实现从服务器下载文件并使用呢？

**一、下载皮肤**

说到下载文件，那就不得不提到iOS的存储机制。众所周知，[iOS App](http://www.dookay.com/product/mobile_ios_android_app_design_develop/index.html)采用的是沙盒（sandbox）存储机制，每个App都有一个独立的沙盒结构，各App之间互不相通。一个沙盒内部默认包含三个文件夹：Documents，Library和tmp。

Documents：苹果建议将程序中建立的或在程序中浏览到的文件数据保存在该目录下，iTunes备份和恢复的时候回包括此目录；

Library：存储程序的默认设置或其他状态信息；

Library/Caches：存放缓存文件，iTunes不会备份此目录，此目录下文件不会在应用退出是删除；

tmp：提供一个即使创建临时文件的地方。

因此我们从服务器下载下来的资源一般存储在Documents或者Caches文件夹下，以防文件在应用退出的时候被删除。

准备下载：

在下载的时候我们用到了系统的一个类——NSURLSession。它是在iOS9之后用来取代NSUrlConnection的类。

要把大象放冰箱，总共分三步，而使用NSURLSession下载文件，只需要两步：

第一步：通过NSURLSession的实例创建task;

第二步：执行task。

- (void)startDownload:(NSString \*)urlString {     // 获取资源的Url     NSURL \*url = [NSURL URLWithString:urlString];     // 创建NSSessionDownloadTask的抽象类执行下载     self.downloadTask = [self.session downloadTaskWithURL:url];     [self.downloadTask resume]; }

通过两个代理方法我们可以监控下载进度，以及在下载完成后进行相应操作

- (void)URLSession:(NSURLSession \*)session downloadTask:(nonnull NSURLSessionDownloadTask \*)downloadTask didWriteData:(int64\_t)bytesWritten totalBytesWritten:(int64\_t)totalBytesWritten totalBytesExpectedToWrite:(int64\_t)totalBytesExpectedToWrite {     //获取下载进度     double progress = (double)totalBytesWritten/totalBytesExpectedToWrite;     NSLog(@"下载进度:%f",progress); } - (void)URLSession:(NSURLSession \*)session downloadTask:(NSURLSessionDownloadTask \*)downloadTask didFinishDownloadingToURL:(NSURL \*)location {         执行后续操作； }

存储下载的文件：

当文件从服务器下载完毕后，我们需要将文件存储到我上面提到的两个文件夹中。

- (void)URLSession:(NSURLSession \*)session downloadTask:(NSURLSessionDownloadTask \*)downloadTask didFinishDownloadingToURL:(NSURL \*)location {    // 获取沙盒Documents路径     NSString \*caches = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES) lastObject];    // 拼接下载文件的文件名，形成绝对路径     self.filePath = [caches stringByAppendingPathComponent:downloadTask.response.suggestedFilename];    // 文件管理器单例     NSFileManager \*mgr = [NSFileManager defaultManager];     // 将文件移动到指定目录位置       [mgr moveItemAtPath:location.path toPath:self.filePath error:nil];     // 准备执行解压方法     [self onFileSelected:self.filePath]; }

判断下载文件的类型：

- (void)onFileSelected:(NSString \*)selectedPath {     NSURL \*fileURL = [NSURL fileURLWithPath:selectedPath];          NSString \*fileNameComponent = fileURL.lastPathComponent;     // 获取文件的扩展名     NSString \*extension = [[fileNameComponent pathExtension] lowercaseString];     // 如果是zip类型的压缩包文件，则进行解压     if ([extension isEqualToString:@"zip"]) {         // 设置解压路径         NSArray \*paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);         NSString \*documentPath = ([paths count] > 0) ? [paths objectAtIndex:0] : nil;         NSString\* unZipTo = documentPath;         // 开始解压缩         [self releaseZipFilesWithUnzipFileAtPath:self.filePath Destination:unZipTo];             } }

**二、解压缩及使用**

在解压缩的部分，用到了SSZipArchive这个工具类。因为这个类是用C语言写的，所以在使用的时候需要在工程中添加libz.tbd库。

// 解压 - (void)releaseZipFilesWithUnzipFileAtPath:(NSString \*)zipPath Destination:(NSString \*)unzipPath{     NSError \*error;     // 如果解压成功     if ([SSZipArchive unzipFileAtPath:zipPath toDestination:unzipPath overwrite:YES password:nil error:&error delegate:self]) {         // 存储主题的色调         UserDefaultsWriteObj([BKTool getThemeColorWithDocName:self.model.siteId], @"mainColor");         UserDefaultsSynchronize;                  // 重新加载页面样式         AppDelegate \*appDelegate = (AppDelegate \*)[UIApplication sharedApplication].delegate;                  [UIView animateWithDuration:0.5 animations:^{             self.view.alpha = 0;             self.view.transform = CGAffineTransformMakeScale(1.2, 1.2);         } completion:^(BOOL finished) {             if ([SITEID integerValue] == 38) {                 appDelegate.window.rootViewController = [[XYZhongBingTabBarController alloc] init];             } else {                 appDelegate.window.rootViewController = [[BKMainTabBarController alloc] init];             }         }];     // 输出解压失败的错误问题     }else {         NSLog(@"%@",error);     } }

至此，皮肤文件已经成功下载到了本地，接下来要注意的一点是关于图片使用的方法。系统提供的[UIImage imageWithNamed:]这个方法是从App包文件里面获取图片的，无法获取沙盒里的图片，所以需要自定义一个方法来使用存储在沙盒里的图片：

+ (UIImage \*)imageWithDocumentoryName:(NSString \*)documentoryName imageName:(NSString \*)imageName {     // 如果文件名不存在或者文件名为空，则返回空     if (!imageName || [imageName isEqualToString:@""]) {         return nil;     }          // 获取图片完整路径     NSString \*documentPath = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES) firstObject];         NSString \*docPath=[documentPath stringByAppendingString:[NSString stringWithFormat:@"/%@",documentoryName]];     NSString \*imgPath=[docPath stringByAppendingPathComponent:[NSString stringWithFormat:@"%@",imageName]];          // 如果文件存在，则返回对应文件     if ([BKTool isFileExist:[NSString stringWithFormat:@"%@/%@.png",documentoryName,imageName]]) {         return [UIImage imageWithContentsOfFile:imgPath];     // 否则获取本地的默认文件     } else {         return [UIImage imageNamed:imageName];     } }

通过两篇文章的讲解，相信大家对主题皮肤的切换功能已经有了一定的了解。快让App变得丰富起来吧！

# iOS【开发之 APP自定义换肤、主题更换】

2017年09月11日 00:07:14 [阿阿阿阿俊](https://me.csdn.net/luolianxi" \t "_blank) 阅读数：1342更多

个人分类： [OC](https://blog.csdn.net/luolianxi/article/category/5847289" \t "_blank)

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/luolianxi/article/details/77927577

# 主题更换的设计思路

现在App类似桌面软件的趋势，在功能完善之后，渐渐追求个性化，以满足不同用户的审美，主题更换就是其中一项，像Weico微博客户端，UC浏览器。所谓主题，可以看成**相同功能不同展现可视资源的集合**，例如，按钮无论在什么主题下都需要背景图片这个资源，只是在不同主题下是不同的背景图片而已。

如何在iOS中实现主题更换的核心思路为：

1. 资源按主题放置：相同功能的资源名称相同，放在不同的主题路径或者前缀使用主题名。
2. 增加中间层，隔离不同主题相同功能资源使用的变化。

## 1. 主题管理

主题的特性导致代码不关心资源的表现是什么，只关心资源的功能，而主题是易变化的，因此需要将易变化的部分抽离出来，整合到一个管理者中，主题的变化在管理者中完成，而不影响资源使用的地方。而且这个管理者是全局唯一的，因此使用单例。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | + (ThemeManager \*)sharedInstance  {  static ThemeManager \*sharedInstance = nil;  if (sharedInstance == nil)  {  sharedInstance = [[ThemeManager alloc] init];  }  return sharedInstance;  } |

主题中的资源使用plist进行存储，颜色的RGBA值跟字体的信息可以直接存入plist，而图片则可以存入图片的位置。按主题命名plist文件，ThemeManager的初始化跟主题更换就从main bundle中按主题名字读取对应的plist文件。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | - (id)init  {  if (self = [super init])  {  NSUserDefaults \*defaults = [NSUserDefaults standardUserDefaults];  NSString \*themeName = [defaults objectForKey:@"theme"] ?: @"default";  NSString \*path = [[NSBundle mainBundle] pathForResource:themeName ofType:@"plist"];  self.theme = [NSDictionary dictionaryWithContentsOfFile:path];  }  return self;  } |

代码中不再是直接使用主题相关的资源，而是通过ThemeManager得到对应主题下的资源。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // 直接使用资源：  UIImage \*image = [UIImage imageNamed:@"xxx\_btn\_background"];  // 通过主题管理器使用资源：  NSDictionary \*theme = [ThemeManager sharedInstance].theme;  NSString \*imageName = [theme objectForKey:@"xxx\_btn\_background"];  UIImage \*image = [UIImage imageNamed:imageName]; |

上面的代码在使用时还是有些复杂，代码只关心资源的功能，不关系也不应该关心取资源的细节，因此应在ThemeManager对取资源进行如下封装：

|  |  |
| --- | --- |
| 1 | - (UIImage \*)imageForKey:(NSString \*)key; |

在使用主题中的资源时，代码就变成了：

|  |  |
| --- | --- |
| 1 | UIImage \*image = [[ThemeManager sharedInstance] imageForKey:@"xxx\_btn\_background"]; |

## 2. 资源的放置

当系统将主题相关的资源文件部署到ios设备中时，在默认情况下，系统会将所有的资源**plat平铺**拷贝到mainBundle目录下，即使你的资源是按文件夹来组织的。（我们可以在模拟器中查看Bundle的情况，模拟器的路径是：~/Library/Application Support/iPhone Simulator）

因此，在将资源文件加入到工程时，不要选默认的”Recursively create groups for any add folders”，要选择**“Create Folder Reference for any add folders”**，这样才能保证资源文件按照原有文件夹的组织格式被拷贝到mainBundle中。

关于上述的两个选项，就涉及到Xcode的Group（黄色）跟Folder Reference（蓝色）的概念了，参见从别处摘抄来的理解：

XCode项目中的文件夹分成两类: group 和 directory reference, 分别是虚结构和实结构. 黄颜色的 group 是默认的格式, 它的结构和磁盘上的文件夹毫无关系, 仅仅表示资源的逻辑组织结构, 这在管理源文件是非常方便. 同一段代码可以被很多项目使用, 也可能只使用一个目录的部分文件, 它不需要被拷贝到当前项目中, 但可以在当前项目中保持一个清晰的逻辑结构. 而且引用头文件时不需要指明复杂的层次结构, 因为这些文件在XCode看来是 flat 的, 即它们处在同一层文件夹里.

但是 group 带来便利的同时也导致更加棘手的麻烦, 文件重名冲突问题; 尤其当你要使用上千个资源文件时, 这种问题已经极难避免; 而且, 资源文件一般是要拷贝到目标程序中的, 虽然它们在项目中可以有结构的组织, 但是复制到程序中时将会 flat 地输出到程序的根目录中, 这将是怎样的一个灾难! 同时, 如果你在外部向文件夹中加入了上百幅图片, 你不得不把它们再向xcode中加入一遍. 归根结底, 还要求助于我们传统的蓝色的 directory reference。

## 3. 主题更换通知

对于没有显示的界面，更换主题是不需要通知的，因为在取资源时是根据当前主题取的，但是对于正在显示的界面，更换主题时就需要进行通知，让界面重新取资源后再重绘。由于这类通知是全局性的，因此应该使用NSNotification实现通知机制。

在ThemeManager的changeTheme中调用[NSNotificaitonCenter defaultCenter]的postNotificationName:object:发出通知，而在各个涉及到主题更换的ViewController中使用addObserver:selector:name:object:监听通知事件。

## 4. 总结

其实主题的设计思路跟类簇很像，例如对于NSNumber，不同类型的数据其实真正返回的是NSNumber相对于此类型的子类，但是对于NSNumber的使用者而言，其并不关心NSNumber返回的具体子类是什么，只要满足NSNumber定义的接口就行。设计总是类似的，针对易变化的部分，增加一个中间层（接口）将易变化的部分封装起来，提供给使用者稳定不易变的服务。

总之，OOP跟DesignPattern在我看来主要做了两件事：

1. 隔离变化
2. 封装细节

iOS 实现快速切换主题详细教程（附上源码）

2016-12-20 00:14 编辑： suiling 分类：[iOS开发](http://www.cocoachina.com/ios/) 来源：[CoderKo1o 投稿](http://www.jianshu.com/p/26abfb7ce5b2)

7 7712

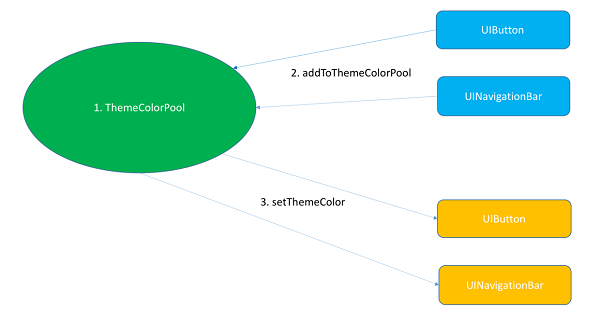
**前言**

iOS 实现主题切换，相信在未来的app里也是会频繁出现的，尽管现在只是出现在主流的APP，如（QQ、新浪微博、酷狗音乐、网易云音乐等），但是现在是看颜值、追求个性的年代，所以根据用户喜好自定义/切换主题也是未来app的必备功能了。

**实现思路**

为了降低耦合度，决定采用的方案是使用NSObject的分类来实现主题设置，有些读者可能会想为何不使用UIView的分类而是使用NSObject的分类？建议这部分读者看一下UIBarItem父类，然后仔细思考一下，就会理解了。

**设置主题色**



PYThemeColor.png

* 创建主题色池
* 将需要设置主题色的控件及其对应属性/方法添加到主题色池中
* 调用设置主题色方法时，遍历主题色池中的控件，使用KVC设置对应属性或调用对应的方法来实现主题色的设置

**代码实现**

建议读者在理解思路以后先下载源码大概看一下（纵观全局）再阅读以下内容：

源码地址：<https://github.com/iphone5solo/PYTheme>

1. 创建主题色池

由于是在NSObject的分类里面创建，为了方便管理，设置全局变量\_themeColorPool，并通过懒加载完成\_themeColorPool的实例化。数组中的对象采用为NSDictionary的原因见下一步就会理解了

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | /\*\* 主题颜色池 \*/  static NSMutableArray \*\_themeColorPool;    #pragma mark - 懒加载  - (NSMutableArray \*)themeColorPool  {  **if** (!\_themeColorPool) {          \_themeColorPool = [NSMutableArray array];      }  **return** \_themeColorPool;  } |

2. 添加控件到主题色池中

由于颜色设置有的可以直接通过属性设置也有的需要通过调用方法才可设置。以UIButton为例，设置背景色可通过属性button.backgroundColor设置，设置选中状态时的字体颜色则要调用setTitleColor:forState:方法才可设置，于是，就得提供两个方法供使用者调用，如下

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | /\*\*   \* 添加到主题色池   \* selector : 执行方法   \* objects : 方法参数数组   \* 注意：方法参数必须按顺序一一对应，如果涉及到的主题色设置使用 PYTHEME\_THEME\_COLOR 宏定义代替   \* 如果数组中某个参数为nil，需包装为 [NSNull null] 对象再添加到数组中   \*/  - (void)py\_addToThemeColorPoolWithSelector:(SEL)selector objects:(NSArray \*)objects;  /\*\*   \* 添加到主题色池   \* propertyName : 属性名   \*/  - (void)py\_addToThemeColorPool:(NSString \*)propertyName; |

实现如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66 | #pragma mark - Theme Color  /\*\*   \* 添加到主题色池   \* selector : 执行方法   \* objects : 方法参数数组   \* 注意：方法参数必须按顺序一一对应，如果涉及到的主题色设置使用 PYTHEME\_THEME\_COLOR 宏定义代替   \* 如果数组中某个参数为nil，需包装为 [NSNull null] 对象再添加到数组中   \*/  - (void)py\_addToThemeColorPoolWithSelector:(SEL)selector objects:(NSArray \*)objects  {      // 判断参数是否为空  **if** (!objects) **return**;      Class appearanceClass = NSClassFromString(@"\_UIAppearance");      // 如果对象为\_UIAppearance，直接返回  **if** ([self isMemberOfClass:appearanceClass]) **return**;      // 键：对象地址+方法名 值：对象      NSString \*pointSelectorString = [NSString stringWithFormat:@"%p%@", self, NSStringFromSelector(selector)];      NSDictionary \*dic = @{ pointSelectorString      : self,                             PYTHEME\_COLOR\_ARGS\_KEY   : objects };      // 判断是否已经在主题色池中  **if** (![[self themeColorPool] containsObject:dic]) { // 不在主题色池中          [[self themeColorPool] addObject:dic];  **if** (\_currentThemeColor) { // 已经设置主题色，直接设置              [self py\_performSelector:selector withObjects:objects];          }      }  }    /\*\*   \* 添加到主题色池   \* propertyName : 属性名   \*/  - (void)py\_addToThemeColorPool:(NSString \*)propertyName  {      // 如果对象为\_UIAppearance，直接返回      Class appearanceClass = NSClassFromString(@"\_UIAppearance");  **if** ([self isMemberOfClass:appearanceClass]) **return**;      // 键：对象地址+属性名 值：对象      NSString \*pointString = [NSString stringWithFormat:@"%p%@", self, propertyName];      NSDictionary \*dic = @{ pointString : self };      // 判断是否已经在主题色中  **if** (![[self themeColorPool] containsObject:dic]) { // 不在主题色池中          [[self themeColorPool] addObject:dic];  **if** (\_currentThemeColor) { // 已经设置主题色，直接设置              [self setValue:\_currentThemeColor forKey:propertyName];          }      }      // 遍历主题色池(移除应该被回收的对象)  **for** (NSDictionary \*dict **in** [[self themeColorPool] copy]) {          // 取出key          NSString \*objectKey = nil;  **for** (NSString \*key **in** [dict allKeys]) {  **if** (![key isEqualToString:PYTHEME\_COLOR\_ARGS\_KEY]) {                  objectKey = key;  **break**;              }          }          // 取出对象          id object = [dict valueForKey:objectKey];          // 取出对象的引用计数          NSInteger retainCount = [[object valueForKey:@"retainCount"] integerValue];  **if** (retainCount == 2) { // 对象应该被回收了              [[self themeColorPool] removeObject:dict];          }      }  } |

为了满足个别需求，所以还是提供一下从主题色池中移除控件的方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | /\*\*   \* 从主题色池移除   \* propertyName : 属性名   \*/  - (void)py\_removeFromThemeColorPool:(NSString \*)propertyName;  /\*\*   \* 从主题色池移除   \* selector : 方法选择器   \*/  - (void)py\_removeFromThemeColorPoolWithSelector:(SEL)selector; |

实现如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | /\*\*   \* 从主题色池移除   \* propertyName : 属性名   \*/  - (void)py\_removeFromThemeColorPool:(NSString \*)propertyName  {      // 如果对象为\_UIAppearance，直接返回      Class appearanceClass = NSClassFromString(@"\_UIAppearance");  **if** ([self isMemberOfClass:appearanceClass]) **return**;      // 键：对象地址+属性名 值：对象      NSString \*pointString = [NSString stringWithFormat:@"%p%@", self, propertyName];      NSDictionary \*dic = @{ pointString : self };      // 判断是否已经在主题色池中  **if** ([[self themeColorPool] containsObject:dic]) { // 在主题色池中          [[self themeColorPool] removeObject:dic];      }  }    /\*\*   \* 从主题色池移除   \* selector : 方法选择器   \*/  - (void)py\_removeFromThemeColorPoolWithSelector:(SEL)selector  {      // 如果对象为\_UIAppearance，直接返回      Class appearanceClass = NSClassFromString(@"\_UIAppearance");  **if** ([self isMemberOfClass:appearanceClass]) **return**;      // 键：对象地址+方法名      NSString \*pointSelectorString = [NSString stringWithFormat:@"%p%@", self, NSStringFromSelector(selector)];      // 遍历主题色池(移除应该被回收的对象)  **for** (NSDictionary \*dict **in** [[self themeColorPool] copy]) {  **for** (NSString \*key **in** [dict allKeys]) {  **if** ([key isEqualToString:pointSelectorString]) { // 存在，移除                  [[self themeColorPool] removeObject:dict];  **return**;              }          }      }  } |

3. 设置主题色

实现如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | /\*\*   \* 设置主题色   \* color : 主题色   \*/  - (void)py\_setThemeColor:(UIColor \*)color  {      \_currentThemeColor = color;      // 遍历缓主题池，设置统一主题色  **for** (NSDictionary \*dict **in** [\_themeColorPool copy]) {          // 取出key          NSString \*objectKey = nil;  **for** (NSString \*key **in** [dict allKeys]) {  **if** (![key isEqualToString:PYTHEME\_COLOR\_ARGS\_KEY]) {                  objectKey = key;  **break**;              }          }          // 取出对象          id object = [dict valueForKey:objectKey];  **if** ([objectKey containsString:@":"]) { // 方法              // 取出参数              NSArray \*args = dict[PYTHEME\_COLOR\_ARGS\_KEY];              // 取出方法              NSString \*selectorName = [objectKey substringFromIndex:[[NSString stringWithFormat:@"%p", object] length]];              SEL selector = NSSelectorFromString(selectorName);              // 调用方法，设置属性              [object py\_performSelector:selector withObjects:args];          } **else** { // 成员属性              // 取出属性值              NSString \*propertyName = [objectKey substringFromIndex:[[NSString stringWithFormat:@"%p", object] length]];              // 给对象的对应属性赋值（使用KVC）              [object setValue:color forKeyPath:propertyName];          }      }  } |

**使用**

假设有个需求：UINavigationBar的背景颜色和UIButton选中时的字体颜色会随着主题颜色的变化而变化，实现如下：

将navigationBar的background和UIButton的setTitleColor:forState:方法添加到主题池中，方法参数中如果是设置为主题色的参数则用PYTHEME\_THEME\_COLOR占位，如果参数为nil，则使用[NSNull null]代替

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // 创建导航栏  UINavigationBar \*navigationBar = [[UINavigationBar alloc] init];  // 添加到主题色池中  [navigationBar py\_addToThemeColorPool:@"barTintColor"];    // 创建按钮  UIButton \*button = [[UIButton alloc] init];  // 添加到主题色中  [button py\_addToThemeColorPoolWithSelector:@selector(setTitleColor:forState:) objects:@[PYTHEME\_THEME\_COLOR, @(UIControlStateSelected)]]; |

设置主题色

|  |  |
| --- | --- |
| 1  2 | // 设置主题色为红色  [self py\_setThemeColor:[UIColor redColor]]; |

这里有一点注意的是[object py\_performSelector:selector withObjects:args];这是自己实现的performSelector 多参调用关于这方面的网上已经有很多教程了，这里就不多介绍了。直接附上的我实现(内部方法，主要考虑到自己的使用)：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113 | #pragma mark - performSelector 多参调用  - (id)py\_performSelector:(SEL)selector withObjects:(const NSArray \*)objects  {      // 1. 创建方法签名      // 根据方法来初始化NSMethodSignature      NSMethodSignature \*methodSignate = [[self class] instanceMethodSignatureForSelector:selector];  **if** (!methodSignate) { // 没有该方法  **return** self;      }      // 2. 创建invocation对象（包装方法）      NSInvocation \*invocation = [NSInvocation invocationWithMethodSignature:methodSignate];      // 3. 设置相关属性      // 调用者      invocation.target = self;      // 调用方法      invocation.selector = selector;      // 获取除self、\_cmd的参数个数      NSInteger paramsCount = methodSignate.numberOfArguments - 2;      // 取最少的，防止越界      NSInteger count = MIN(paramsCount, objects.count);      // 用于dictionary的拷贝(用于保住objCopy，避免非法内存访问)      NSMutableDictionary \*objCopy = nil;      // 设置参数  **for** (int i = 0; i < count; i++) {          // 取出参数对象          id obj = objects[i];          // 如果是主题颜色参数颜色，则设置  **if** ([obj isKindOfClass:[NSString class]] && [obj isEqualToString:PYTHEME\_THEME\_COLOR]) {              obj = \_currentThemeColor;          }          // 判断需要设置的参数是否是NSNull, 如果是就设置为nil  **if** ([obj isKindOfClass:[NSNull class]]) {              obj = nil;          }          // 获取参数类型          const char \*argumentType = [methodSignate getArgumentTypeAtIndex:i + 2];          // 判断参数类型 根据类型转化数据类型（如果有必要）          NSString \*argumentTypeString = [NSString stringWithUTF8String:argumentType];  **if** ([argumentTypeString isEqualToString:@"@"]) { // id              // 如果是dictionary，可能存在 PYTHEME\_THEME\_COLOR  **if** ([obj isKindOfClass:[NSDictionary class]]) { // NSDictionary                  objCopy = [obj mutableCopy];                  // 取出所有键                  NSArray \*keys = [objCopy allKeys];  **for** (NSString \*key **in** keys) {                      // 取出值                      id value = objCopy[key];  **if** ([value isKindOfClass:[NSString class]] && [value isEqualToString:PYTHEME\_THEME\_COLOR]) {                          // 替换成颜色                          [objCopy setValue:\_currentThemeColor forKey:key];                      }                  }                  [invocation setArgument:&objCopy atIndex:i + 2];              } **else** { // 其他                  [invocation setArgument:&obj atIndex:i + 2];              }          }  **else** **if** ([argumentTypeString isEqualToString:@"B"]) { // bool              bool objVaule = [obj boolValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"f"]) { // float              float objVaule = [obj floatValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"d"]) { // double              double objVaule = [obj doubleValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"c"]) { // char              char objVaule = [obj charValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"i"]) { // int              int objVaule = [obj intValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"I"]) { // unsigned int              unsigned int objVaule = [obj unsignedIntValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"S"]) { // unsigned short              unsigned short objVaule = [obj unsignedShortValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"L"]) { // unsigned long              unsigned long objVaule = [obj unsignedLongValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"s"]) { // shrot              short objVaule = [obj shortValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"l"]) { // long              long objVaule = [obj longValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"q"]) { // long long              long long objVaule = [obj longLongValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"C"]) { // unsigned char              unsigned char objVaule = [obj unsignedCharValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"Q"]) { // unsigned long long              unsigned long long objVaule = [obj unsignedLongLongValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"{CGRect={CGPoint=dd}{CGSize=dd}}"]) { // CGRect              CGRect objVaule = [obj CGRectValue];              [invocation setArgument:&objVaule atIndex:i + 2];          } **else** **if** ([argumentTypeString isEqualToString:@"{UIEdgeInsets=dddd}"]) { // UIEdgeInsets              UIEdgeInsets objVaule = [obj UIEdgeInsetsValue];              [invocation setArgument:&objVaule atIndex:i + 2];          }      }      // 4.调用方法      [invocation invoke];      // 5. 设置返回值      id returnValue = nil;  **if** (methodSignate.methodReturnLength != 0) { // 有返回值          // 将返回值赋值给returnValue          [invocation getReturnValue:&returnValue];      }  **return** returnValue;  } |

**细节处理**

1. 设置主题色的方式

* 通过属性直接设置主题色
* 通过调用方法并以主题色为参数来设置主题色
* 通过调用方法但主题色被封装后(如：NSDictionary)作为参数设置主题色

2. 自动管理内存管理

当对象应该被释放后，下一次当主题色池有新元素添加时，会遍历主题色池，根据对象的引用计数来决定是否移除对象(实现自动管理内存)，因此：主题色池中最多可能会残留一个对象，这对内存几乎没有任何影响，如果要及时释放对象本人认为可以采用KVO监听对象的引用计数（未尝试），但是耗能高，不建议这么做！

3. 当对象为\_UIAppearance类时，不添加到主题色池

了解UIAppearance的读者应该可以理解，而且使用UIAppearance的目的也为为了设置全局色，所以为了避免冲突，如果使用了该“技术”就不添加到主题色池

**设置主题图片**

观察了新浪微博、酷狗音乐等app，发现设置主题图片还是很有必要的，而且发现每套主题皮肤/图片都有对应的主题色，所以在设计接口的时候都考虑了这方面的需求。先看一下设置主题图片的基本原理，如下：

1. 创建一个主题图片池（使用懒加载）
2. 将相关控件对象直接添加到主题图片池中
3. 设置主题图片时，通过block把主题图片池中的所有对象传递给用户，用户实现block，在block中获得对象，并根据需求设置相关属性完成主题图片的设置

**代码实现：**

1. 创建一个主题图片池（使用懒加载）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | /\*\* 主题图片池 \*/  static NSMutableArray(id) \*\_themeImagePool;（此处圆括号代替尖括号）    - (NSMutableArray \*)themeImagePool  {  **if** (!\_themeImagePool) {          \_themeImagePool = [NSMutableArray array];      }  **return** \_themeImagePool;  } |

2. 添加相关控件到主题图片池中

因为在设置图片是，比较复杂，如UITabBar上面的UIBarItem的图片、字体颜色等，所以为了满足大部分用户的需求，决定采用的是直接存储控件对象

|  |  |
| --- | --- |
| 1  2  3  4  5 | /\*\* 添加到主题图片池 \*/  - (void)py\_addToThemeImagePool;    /\*\* 从主题图片池中移除 \*/  - (void)py\_removeFromThemeImagePoo |

实现如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | #pragma mark - Theme Image  /\*\* 添加到主题图片池 \*/  - (void)py\_addToThemeImagePool  {      // 如果对象为\_UIAppearance，直接返回      Class appearanceClass = NSClassFromString(@"\_UIAppearance");  **if** ([self isMemberOfClass:appearanceClass]) **return**;    **if** ([self isKindOfClass:[UITabBarItem class]]) { // 如果是UITabBarItem，判断是否有设置图片          UITabBarItem \*item = (UITabBarItem \*)self;  **if** (!item.image) { // 没有设置图片              item.image = [[UIImage alloc] init];          }  **if** (!item.selectedImage) { // 没有设置图片              item.selectedImage = [[UIImage alloc] init];          }      }      // 判断是否已经在主题图片池中  **if** (![[self themeImagePool] containsObject:self]) { // 不在主题图片池中          [[self themeImagePool] addObject:self];      }      // 遍历主题图片池(移除应该被回收的对象)  **for** (id object **in** [self themeImagePool]) {          NSInteger retainCount = [[object valueForKey:@"retainCount"] integerValue];  **if** (retainCount == 2) { // 对象应该被回收了              [[self themeImagePool] removeObject:self];          }      }  }    /\*\* 从主题图片池中移除 \*/  - (void)py\_removeFromThemeImagePool  {      // 如果对象为\_UIAppearance，直接返回      Class appearanceClass = NSClassFromString(@"\_UIAppearance");  **if** ([self isMemberOfClass:appearanceClass]) **return**;        // 判断是否已经在图片池中  **if** ([[self themeImagePool] containsObject:self]) { // 在主题图片池中          [[self themeImagePool] removeObject:self];      }  } |

3. 设置主题图片和相关配色

当设置图片时，会通过block将主题图片池里面的所有控件传递给用户，用户根据需求进行相关设置，如果提供了配色，就会采用上面设置主题色功能来设置主题色

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\*\*   \* 重新加载主题图片   \* themeColor : 主题色   \* block : 设置主题图片时调用的block   \*/  - (void)py\_reloadThemeImageWithThemeColor:(UIColor \*)themeColor setting:(PYThemeImageSettingBlock)block; |

实现如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | /\*\* 重新加载主题图片 \*/  - (void)py\_reloadThemeImageWithThemeColor:(UIColor \*)themeColor setting:(PYThemeImageSettingBlock)block  {  **if** (themeColor) { // 有主题色，设置主题色          [self py\_setThemeColor:themeColor];      }    **if** (block) { // 存在block，直接调用          block([self themeImagePool]);      }  } |

**使用**

假设现在有这么一个需求：更换主题图片时，更换UITabBarItem的图片

1.将UITabBarItem添加到图片池

|  |  |
| --- | --- |
| 1  2 | // UITabBarItem  [childController.tabBarItem py\_addToThemeImagePool]; |

2.切换主题图片并设置配色为红色

|  |  |
| --- | --- |
| 1  2  3  4 | // 重新加载主题图片，并设置主题色为红色  [self py\_reloadThemeImageWithThemeColor:[UIColor redColor] setting:^(const NSArray \*objects) {      // 根据控件类型完成相关设置  } |

**总结**

篇幅可能有点大，能耐心读到这里的读者相信会有不少收获的，希望读者在阅读此教程的时候，千万不要学习代码实现，而是要多思考：为什么要这样实现？那样实现有什么不好？多学学接口为什么要这样设计，那样设计是不是更合理？当你带着这些问题再回过头来去看看源码时，希望你会有更多的收货！当然，这里只是提供了一种思路，你也可以在此基础上实现夜间模式的切换等。期待你们的实现！

**期望**

当然如果您有更多的想法想表达或者交流的话，欢迎到留言/评论！因为本人比较喜欢活跃在[GitHub](https://github.com/)社区，所以，如果您有什么想反馈的也可以[issuse me](http://www.jianshu.com/p/26abfb7ce5b2)，在这也鼓励大家去多多发现优秀源码，并且共享给大家。毕竟分享是双方获利的，何乐而不为？

源码地址：<https://github.com/iphone5solo/PYTheme>

源码作者：[CoderKo1o](https://github.com/iphone5solo)

[iOS开发之App主题切换完整解决方案(Swift版)](https://www.cnblogs.com/ludashi/p/6289741.html)

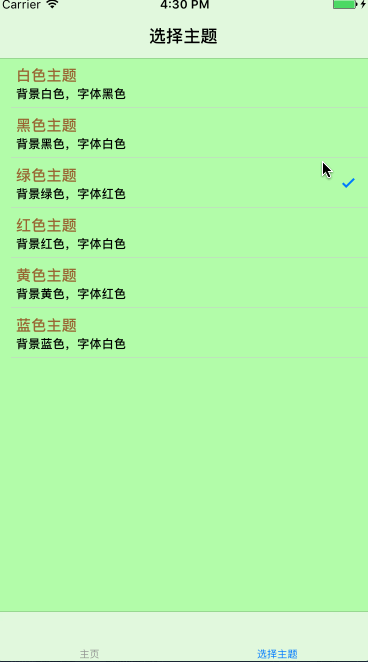
本篇博客就来介绍一下iOS App中主题切换的常规做法，当然本篇博客中只是提到了一种主题切换的方法，当然还有其他方法，在此就不做过多赘述了。本篇博客中所涉及的Demo完全使用Swift3.0编写完成，并使用iOS的NSNotification来触发主题切换的动作。本篇博客我们先对我们的主题系统进行设计，然后给出具体实现方式。当然在我们设计本篇博客所涉及的Demo时，我们要遵循“高内聚，低耦合”，“面向接口编程”，“便于维护与扩充”等特点。

本篇博客我们先看一下Demo的运行效果，然后给出设计方案已经对应的类图，然后再根据设计的类图给出相应的代码实现。

**一、主题切换效果展示**

先入为主，接下来我们先看一下本篇博客所涉及Demo的最终运行效果。为了看到整体主题切换的效果，所以我们添加了一个主页以及主页Push进去的子页面。另一个页面就是选择相应主题的页面，也就是下方你所看到的TabieView的列表页，该列表中提供了6种可选的主题，点击相应的主题即可切换到该主题上。

当然主题切换，意味着整个App的风格都得改变，所以当修改完主题后，主页以及其子页面的主题也得随之改变。具体效果如下gif图所示。当然Demo比较简单，但是麻雀虽小，五脏俱全，用来了解App的主题切换足以。当然有更好的实现方式，欢迎留言交流。



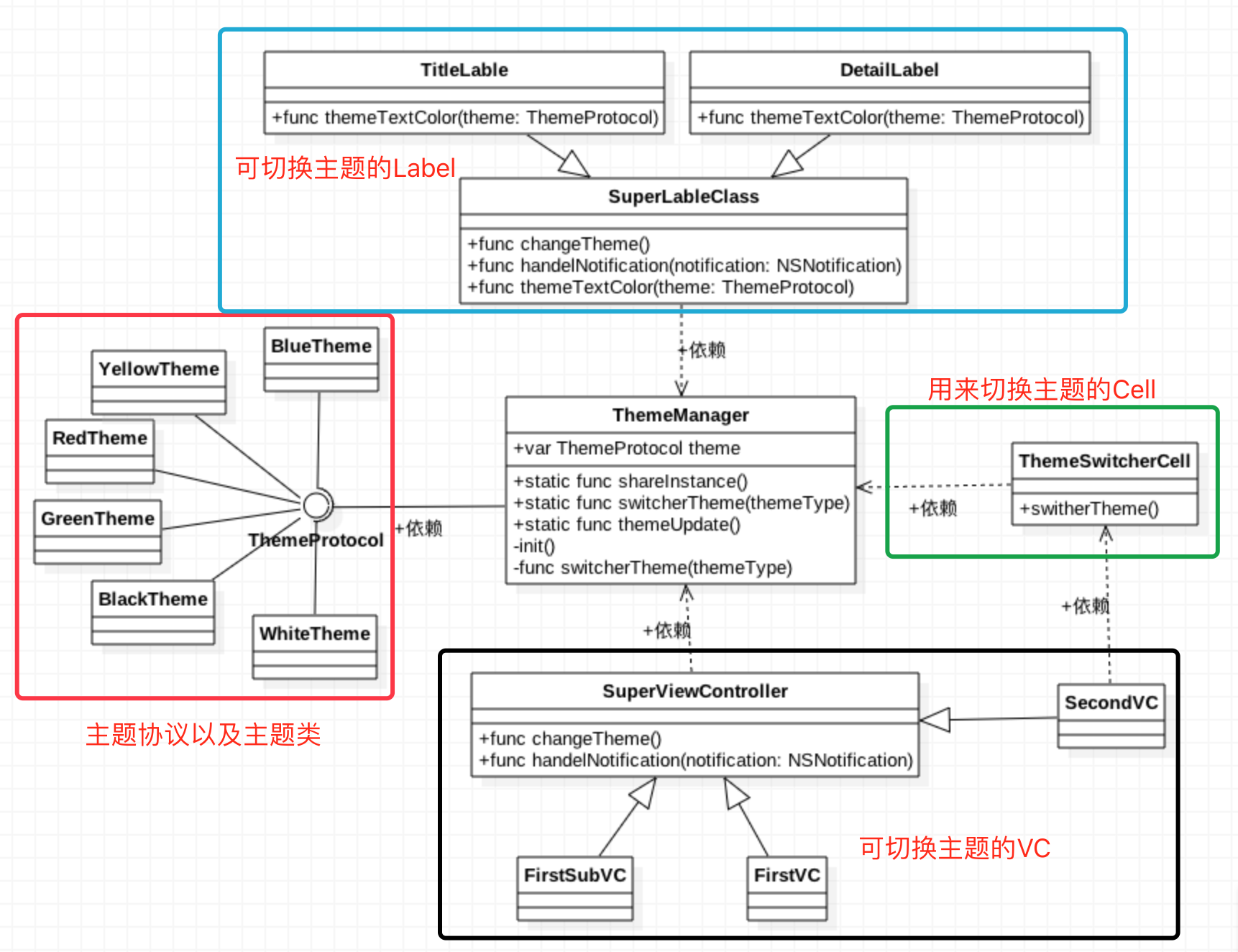
**二、设计主题切换功能的整体结构**

看完效果后，接下来我们就进入了设计阶段。根据上述Demo的特点，我们先给出相应的类图，当然我们敲代码时要根据该类图进行实现。下方截图就是我们本篇博客主题切换工程的类图了。该类图中省略了一些细节，给出了核心的内容。接下来我们来详细的看一下下方这个类图的结构，如果下方的类图搞明白了，那么在下部分中看相应代码时，就会显得游刃有余了，因为我们的代码是按照下方的类图所实现的。只不过比下方类图更为详细。

接下来我们就来看一下类图的结构，我们就按照不同颜色的框逐一介绍，框中的内容算是一个模块。

* **红色框**：下方类图红色框中就是我们的主题协议ThemeProtocol以及遵循该协议的所有主题。当然，依据“面向接口”编程的规则，外部所依赖的不是所有的主题类，而是主题协议ThemeProtocol。也就是ThemeManager类依赖于ThemeProtocol。而所有的主题类通过工厂类或者工厂方法间接的与ThemeManager。因为鉴于Swift中的枚举中可以添加相应的计算属性以及方法，所以我们可以使用相应的主题枚举来充当这个工厂类，当然下方类图中没有给出，这个要看具体的代码实现。
* **蓝色框**：接下来我们就来看蓝色框，蓝色框中的SuperLabelClass是所有可以改变主题的Label的父类，也就是说在改变主题时，需要改变相应的Label就要继承该类。
* **黑色框**：该框中中则是SuperViewController及其子类，需要更换主题的VC都要继承自该基类。
* **绿色框**：绿框中则负责修改主题，其中根据Cell的类型来确定用户选择的是哪个主题。
* **ThemeManager**: 该类以单例的形式存在，负责管理相应的主题，切换主题时，直接调用ThemeManager中的相应的方法即可。

上面只是简单的介绍一下类图，具体的还要看代码的实现。



**三、主题以及主题管理者的实现**

看完运行效果以及设计类图，接下来我们就来一下具体的代码实现本部分我们就来看一下相应的主题类以及主题管理者的代码实现。下方会先给出主题协议以及各个主题类的实现，然后再给出主题工厂方法以及主题管理者的实现。

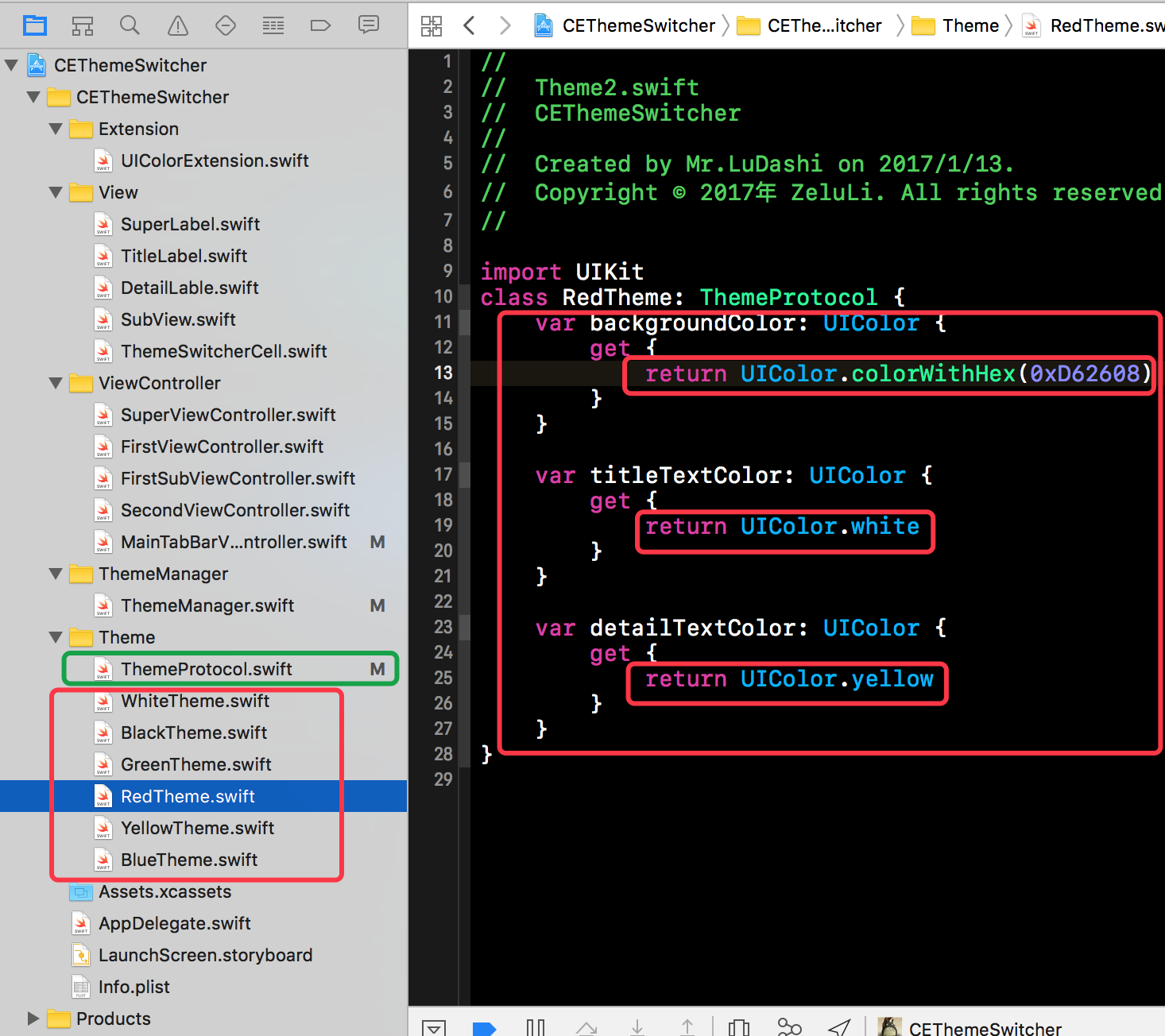
**1、主题协议的实现**

下方就是我们定义的主题协议，所有主题都要遵循该协议。因为本篇博客的Demo在切换主题时只涉及到了三个元素，一个是背景色（backgroundColor），一个是标题的颜色（titleTextColor），在一个就是详情的颜色（detailTextColor）。所以在我们的主题协议中有三个只读的计算属性与之对应。具体如下所示：

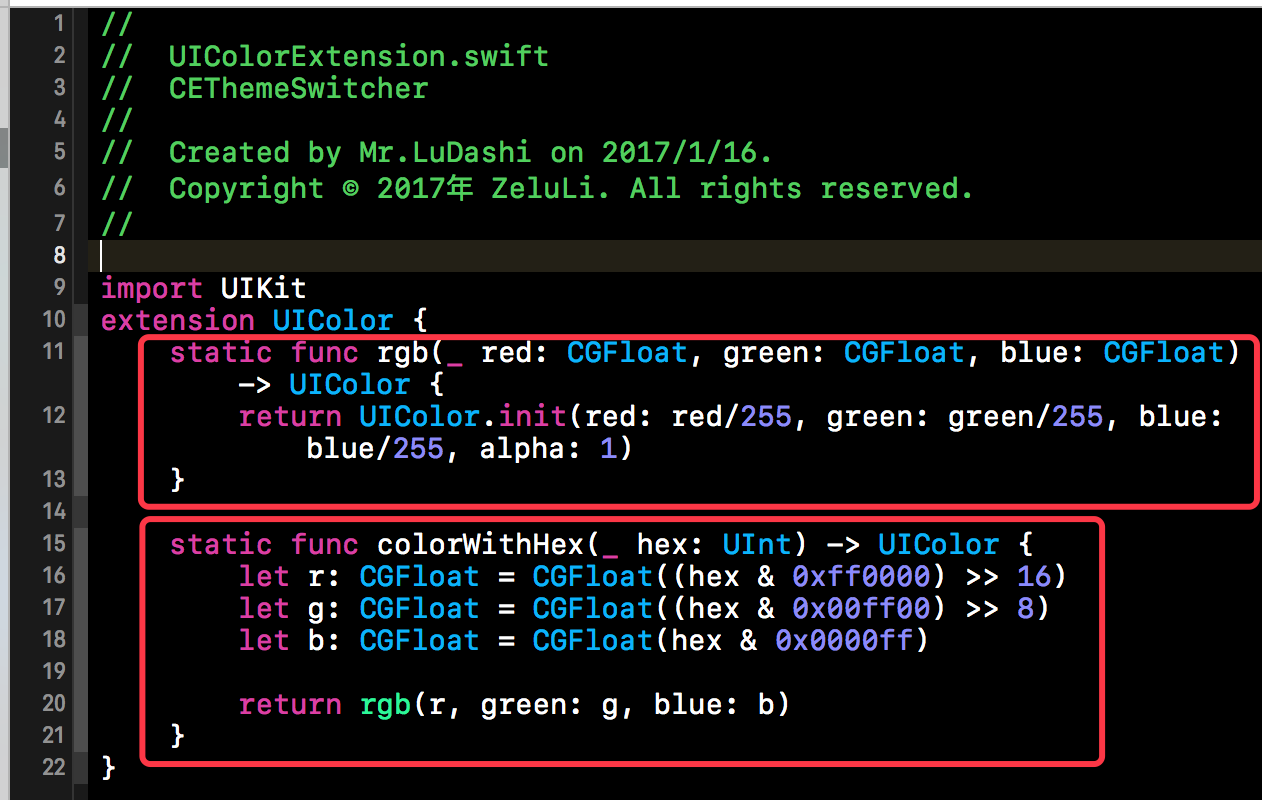


**2.各个主题的实现**

接下来我们就要看看各个主题的实现了，我们就以红色主题（RedTheme）为例。RedTheme要遵循ThemeProtocol，并且实现协议中规定的方法。在RedTheme中每个计算属性都会返回当前主题所对应的属性。具体如下所示。当然其他主题与下方的RedTheme类似，只是每个计算属性返回的值不同。



上面在返回UIColor对象时，我们用到了UIColor.colorWithHex(十六进制数)。colorWithHex()是我们在UIColor的延展中添加的静态方法，该方法可以根据十六进制的RGB参数来初始化相应的UIColor对象。下方是该UIColor延展中的具体内容。下方代码的核心就是位操作的使用。



**3.简单工厂的创建**

主题协议已经主题类构建完毕后，接下来我们就该将这些主题进行打包输出了。 下方的代码段本质上就是主题类的简单工厂，因为Swift的枚举类型有很多好用的特性，所以我们使用了Swift的枚举来实现这些主题的简单工厂。代码结构比较简单，在此就不做过多赘述了。



**4.主题管理者(ThemeManager)的创建**

上面关于主题的基本工作已经做完，接下来我们就为这些主题来创建一个主题管理者ThemeManager。**ThemeManager会根据用户需求从上面的主题工厂中取出相应的主题对象提供给用户**，这也就是ThemeManager的职能所在。

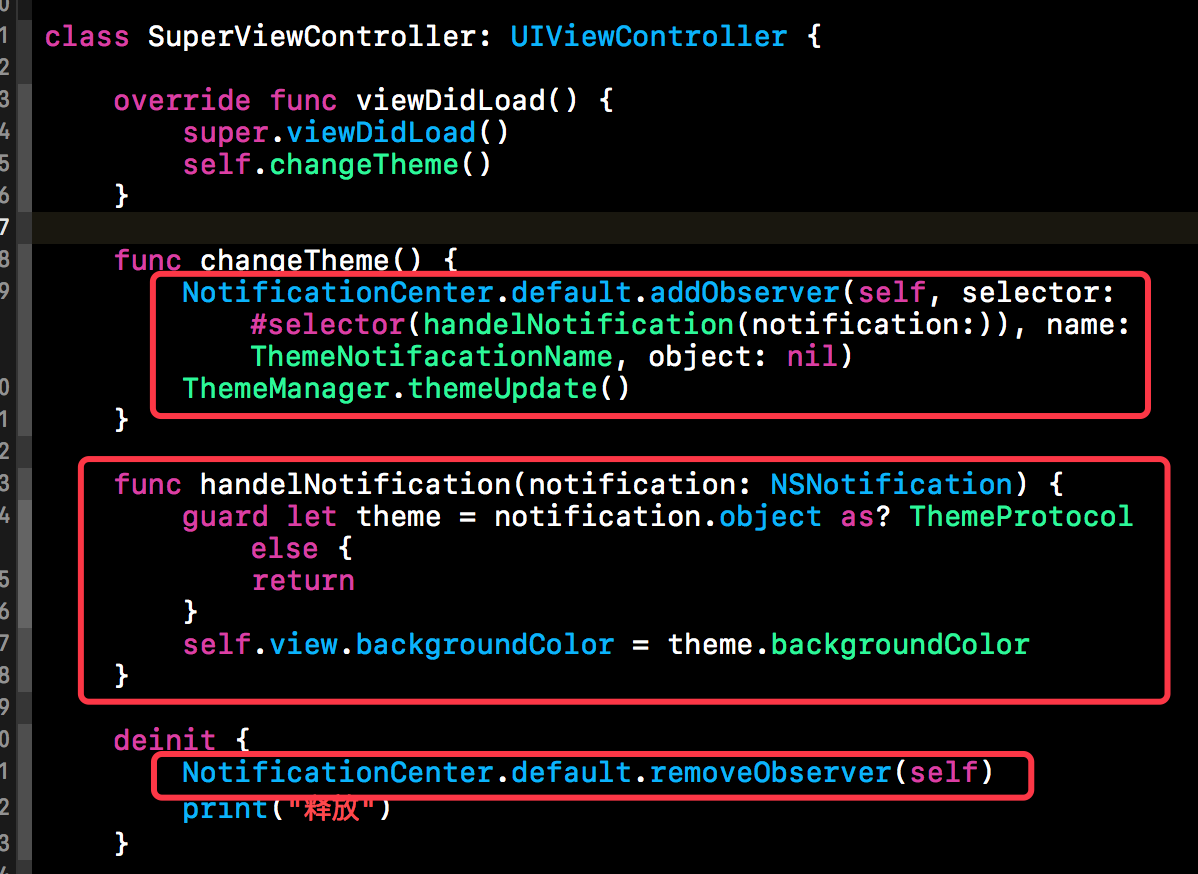
下方就是我们主题管理者ThemeManager的具体实现代码，ThemeManager是以单例的姿态对外展现的。因为主题管理者在整个App运行时只需要一个，所以我们赋予了ThemeManager一个单例的身份与用户进行交流。单例实现完毕后，接着社一些便利方法，这些便利方法都是静态方法，便于用户直接使用。接着是一些ThemeManager的私有方法。

在ThemeManager的私有方法中，switcherTheme(themeType)是关键，该方法在收到修改主题的事件后，会发出修改主题的通知，而通知的内容就是将要进行修改的主题。那些可以修改主题的控件，在收到该通知的消息后，会根据通知内容修改其自身的主题。具体代码如下所示：



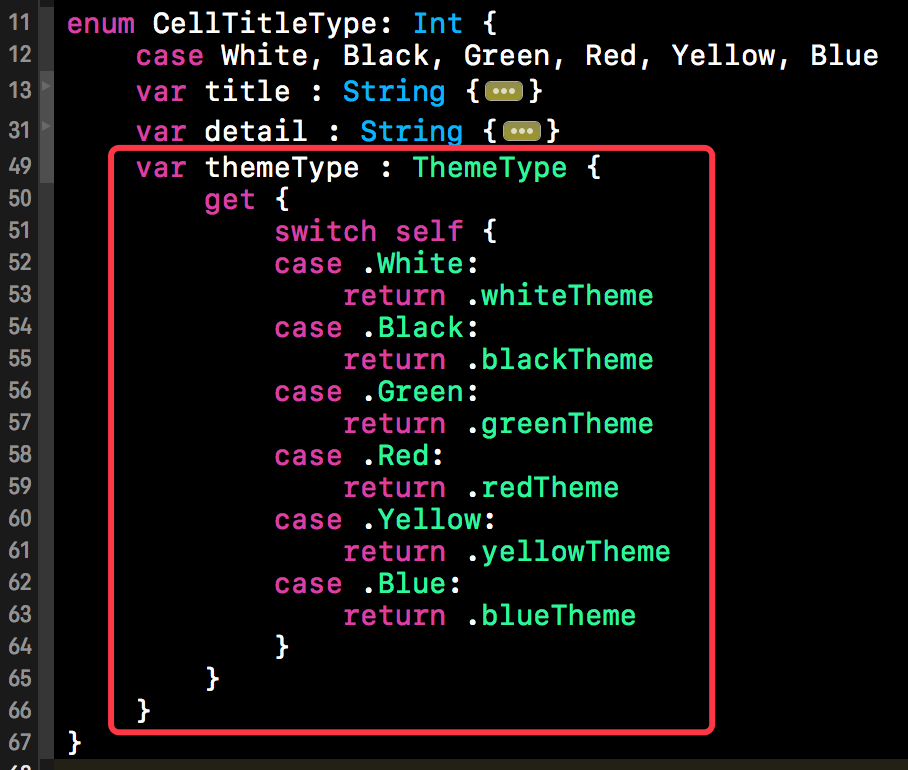
**四、可更换主题的控件处理**

主题以及主题管理者实现完毕，接下来就该使用了。在主题切换时，需要修改属性的控件我们需要进行处理，使其监听主题切换的通知，并根据通知内容修改控件的属性。在本部分我们就以SuperViewController为例。首先我们要将SuperViewController添加为主题切换的观察者，然后实现收到通知后的方法。在析构函数中，切记要移除观察者。具体代码如下所示。

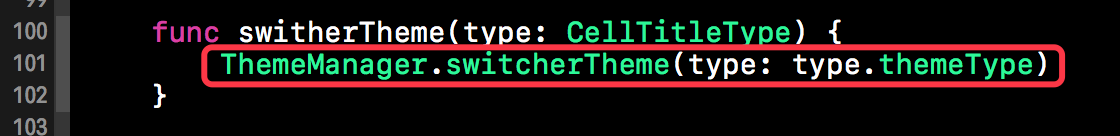


**五、调用ThemeManager更换主题**

万事俱备只欠东风，接下来我们就要在恰当的地方调用ThemeManager的switcherTheme()的方法进行替换了。当然我们是在我们的Cell来处理的，因为每种cell都对应着主题的类型，下方就是Cell类型的枚举。CellTitleType中的themeType计算属性就对应着相应的主题类型，我们可以将该类型提供给ThemeManager，让后ThemeManager发出更改主题的通知。



下方就是调用ThemeManager的switcherTheme()方法来更换主题。

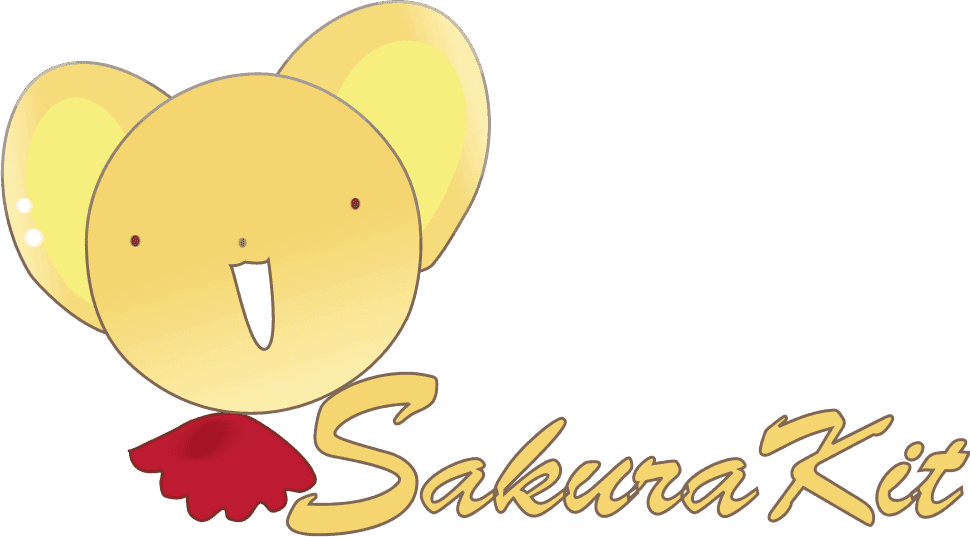


至此，本篇博客所涉及的主题切换的Demo就介绍完毕了，所涉及的Demo按照惯例，依然会在github上进行分享。下方就是Demo在github上分享地址。

代码分享地址：[**https://github.com/lizelu/CEThemeSwitcher**](https://github.com/lizelu/CEThemeSwitcher)

iOS 主题/皮肤之 SakuraKit

时间：17-08-28 15:39:14 点击：3540 来源:tingxins



sakura-kit-logo

**前言**

目前市场上很多 App 都有主题变更、皮肤切换的功能。随着项目代码量的不断增长，业务不断完善，功能性代码逐渐趋于模块化，尤其是在多人协作开发同一个项目时，模块解耦尤为重要，同时，公共基础库的功能性代码使用越简单越好。

前段时间在维护旧项目时，收到 App 主题变更、皮肤切换的需求，其包括 App 中各种图标、色值、文字、字体等都包括在内，都需实现主题化。主要用于：

* **活动主题展示**：比较典型的是类似京东618、天猫淘宝购物节主题变更。
* **用户夜间模式**：类似阅读相关 App 的夜间模式，如：简书等。
* **用户主题变更**：用户可通过本地或者远程下载喜欢的主题，如：网易云音乐、QQ 音乐等 App 主题变更。

由于老项目代码比较混乱，功能模块耦合严重以及开发时间等综合因素，在实现 App 主题变更、皮肤切换的功能的同时，想要在尽量不修改旧代码的基础上增加新的功能是比较麻烦的。

由于没有合适的第三方库，于是自己手撸了一个库 [SakuraKit](https://github.com/tingxins/SakuraKit)，并开源，希望能帮到需要的朋友。

下面我们开始介绍 SakuraKit 及快速入门。

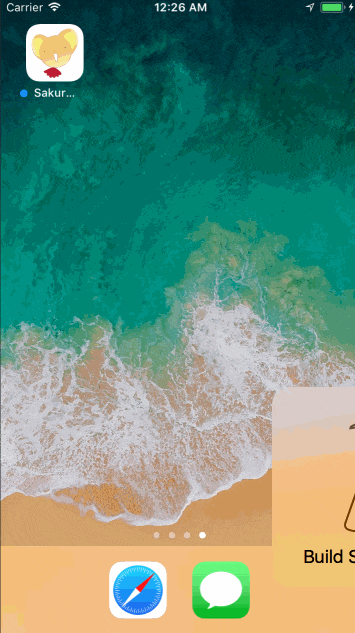
**SakuraKit**

[SakuraKit](https://github.com/tingxins/SakuraKit)，是一个轻量级的、专门用于 App 主题变更、皮肤切换的开源库（灵感源自 SwiftTheme、DKNightVersion等），采用函数式 + 链式的编码方式，简单实用、方便理解、利于维护。

**快速入门**

**效果**

在体验前，我们先来看看效果图：



sakura-kit-demo

**体验**

下面以 UIButton 为例，介绍如何使用 [SakuraKit](https://github.com/tingxins/SakuraKit) 进行主题化：

UIButton \*button = [[UIButton alloc] initWithFrame:CGRectMake(100, 100, 100, 100)];

button.sakura

.backgroundColor(@"Home.buttonBackgroundColor")

.titleColor(@"Home.buttonTitleColor", UIControlStateNormal);

上述代码是给一个 button 的背景色（backgroundColor）以及标题颜色（titleColor）进行主题化。其中 Home.buttonBackgroundColor 与 Home.buttonTitleColor 属配置文件中的 KeyPath，配置文件的功能有点类似语言本地化文件（Localizable.strings）。后文会重点介绍如何设置配置文件。

到此为止，我们已经实现了 button 按钮主题化功能，如果你想切换主题，可以调用如下 API：

+ (BOOL)shiftSakuraWithName:(TXSakuraName \*)name type:(TXSakuraType)type;

其中 name 参数代表主题的名称，type 参数代表主题类型（目前有两种：沙盒和本地）。

现在我们再具体的介绍一下如何使用 SakuraKit。

**配置文件**

做过 App 语言本地化的童鞋，应该比较熟悉 Localizable.strings 文件配置，同理，我们在使用 [SakuraKit](https://github.com/tingxins/SakuraKit) 对 App 进行主题化时，也需要进行类似的配置。目前支持 **.json** 和 **.plist** 两种文件格式。

下面我们以 .json 文件格式做示例：

{

    "Home":{

            "buttonBackgroundColor":"#BB503D",

            "buttonTitleColor":"#4AF2A1"

        }

}

在上述体验代码中，我们看到这样的字符串：Home.buttonBackgroundColor 和 Home.buttonTitleColor，这其实就是配置文件中字典的 KeyPath，通过 KeyPath 可以取得不同主题下的值，如：色值、图片名称、文字、字体大小等等。

**注意事项：**

* 每个主题都有自己配置文件，包括本地和沙盒主题。（本地主题名叫 **default**）。
* 主题名称与配置文件名称一致，如：某个主题名叫 **fish**，那么该主题相应的配置文件就应命名为**fish.json。（建议遵守该约定）**
* 不同本地主题的切图命名要做**区分**，不同远程主题的切图命名应**一致**。

**本地主题**

本地主题，即用户无需下载的主题，在 App Bundle 中。除了 App 本身自带的默认主题外，[SakuraKit](https://github.com/tingxins/SakuraKit) 还能够为 App 新增多种本地主题。

配置步骤如下：

**步骤一**

新建 .json 配置文件，比如新建一个名叫 typewriter 的主题，因此配置文件命名为 typewriter.json。

**步骤二**

配置一套切图，并且命名与已有的主题要做区分。

步骤三

完成上述步骤后，在 AppDelegate 中 -application:application didFinishLaunchingWithOptions:launchOptions API 注册所有本地主题：

// 注意：本地默认主题无需注册

[TXSakuraManager registerLocalSakuraWithNames:@[@"typewriter"]];

**步骤四**

调用切换主题 API 即可切换至该指定主题：

[TXSakuraManager shiftSakuraWithName:@"typewriter" type:TXSakuraTypeMainBundle];

**远程主题**

远程主题（资源压缩包.zip），即用户通过网络下载的主题，后台可动态配置。同本地主题一致，分为两部分：**配置文件 + 切图**。当配置文件和切图都弄好后，将文件夹打包成zip文件，传给后台即可。主题数据格式如下（仅供参考）：

{

    "name": "嘻多猴",

    "sakuraName": "monkey",

    "url": "http:\\image.tingxins.cn\sakura\monkey.zip"

}

sakuraName 是切换主题时用的名称，而 url 是该主题的下载地址。（**注：如果 sakuraName 字段传空，那么主题的名称将默认为下载的压缩包名称**）

当远程主题下载完毕后，可以这样切换主题：

[TXSakuraManager shiftSakuraWithName:sakuraName type:TXSakuraTypeSandBox];

值得一提的是，SakuraKit 提供了一些主题下载的简单接口，支持多种主题同时下载等操作，并且支持 Block 和 Delegate 两种方式的回调，同时用户还可自定义下载操作。

下面我们来依次介绍一下主题下载。

**Block 方式**

我们直接来介绍 API ：

[[TXSakuraManager manager] tx\_sakuraDownloadWithInfos:sakuraModel downloadProgressHandler:^(int64\_t bytesWritten, int64\_t totalBytesWritten, int64\_t totalBytesExpectedToWrite) {

    // 下载进度回调

} downloadErrorHandler:^(NSError \* \_Nullable error) {

    // 下载过程出现错误回调

} unzipProgressHandler:^(unsigned long long loaded, unsigned long long total) {

    // 主题下载完成后，解压进度回调

} completedHandler:^(id \_Nullable infos, NSURL \* \_Nullable location) {

    // 主题包解压完毕回调

} ];

其中 sakuraModel 模型数据遵守了 TXSakuraDownloadProtocol 协议，具体使用详见 [SakuraDemo\_OC](https://github.com/tingxins/SakuraKit)，在 DownloadSakuraController 控制器演示了该操作。

**Delegate 方式**

**步骤一**

直接调用 API 实现主题下载：

[[TXSakuraManager manager] tx\_sakuraDownloadWithInfos:sakuraModel delegate:self];

**步骤二**

如果针对步骤一的下载操作需要回调，那么可以选择性的再实现以下方法：

// 重复点击下载某一主题，如果该主题已经处于下载中或者本地存在时将会回调，其中 status 标识该 downloadTask 状态。

- (void)sakuraManagerDownload:(TXSakuraManager \*)manager

                 downloadTask:(NSURLSessionDownloadTask \*)downloadTask

                       status:(TXSakuraDownloadTaskStatus)status;

// 主题下载完毕时回调，其中 infos 包括主题名称，可通过该参数直接切换至该主题

- (void)sakuraManagerDownload:(TXSakuraManager \*)manager

                 downloadTask:(NSURLSessionDownloadTask \*)downloadTask

                  sakuraInfos:(id)infos

    didFinishDownloadingToURL:(NSURL \*)location;

// 主题下载进度

- (void)sakuraManagerDownload:(TXSakuraManager \*)manager

                downloadTask:(NSURLSessionDownloadTask \*)downloadTask

                didWriteData:(int64\_t)bytesWritten

           totalBytesWritten:(int64\_t)totalBytesWritten

   totalBytesExpectedToWrite:(int64\_t)totalBytesExpectedToWrite;

/\*\* Reserved for future use \*/

- (void)sakuraManagerDownload:(TXSakuraManager \*)manager

                downloadTask:(NSURLSessionDownloadTask \*)downloadTask

           didResumeAtOffset:(int64\_t)fileOffset

          expectedTotalBytes:(int64\_t)expectedTotalBytes;

// 下载操作出现错误时回调

- (void)sakuraManagerDownload:(TXSakuraManager \*)manager

                 sessionTask:(NSURLSessionTask \*)downloadTask

        didCompleteWithError:(nullable NSError \*)error;

// 主题下载包解压进度回调

- (void)sakuraManagerDownload:(TXSakuraManager \*)manager

                 downloadTask:(NSURLSessionDownloadTask \*)downloadTask

                progressEvent:(unsigned long long)loaded

                        total:(unsigned long long)total;

具体使用详见 [SakuraDemo\_OC](https://github.com/tingxins/SakuraKit)，在 AppDelegate 中演示了该操作。

**自定义下载操作**

除了上述自带的下载操作外，SakuraKit 还提供了自定义下载操作相关的 API ：

// sakuraModel 模型数据遵守了 TXSakuraDownloadProtocol 协议，location 即自定义下载下来的主题包地址。

[[TXSakuraManager manager] tx\_generatePathWithInfos:sakuraModel downloadFileLocalURL:location successHandler:^(NSString \*toFilePath, NSString \*sakuraPath, TXSakuraName \*sakuraName) {

                  dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0), ^{

      BOOL isSuccess = [SSZipArchive unzipFileAtPath:toFilePath toDestination:sakuraPath delegate:self];

      // 注意：自定义下载操作，必须进行 Sakura 路径格式化！Required！

      [TXSakuraManager formatSakuraPath:sakuraPath cleanCachePath:toFilePath];

      dispatch\_sync(dispatch\_get\_main\_queue(), ^{

          if (isSuccess) {

              [TXSakuraManager shiftSakuraWithName:sakuraName type:TXSakuraTypeSandBox];

          }

      });

   });

} errorHandler:^(NSError \* \_Nullable error) {

   NSLog(@"errorDescription:%@",error);

}];

**FQA**

**1.为何每个主题都有自己配置文件？**

答：由于每个主题，除了切图的命名是是一致的外，不同的主题背景色、字体大小可能不一样，因此，每个主题都要有自己的配置文件，除非只对切图进行本地化。

**2.为何主题名称与配置文件名称一致？**

答：这只是一个约定，[SakuraKit](https://github.com/tingxins/SakuraKit) 会通过主题名称找到该主题在本地或者在沙盒中的路径，使得主题名称与配置文件名称一致，可以减少不必要的工作量。

**3.本地与沙盒主题有什么区别？**

答：在本地主题称为 mainBundle 主题，远程主题称为 Sandbox 主题。

**开源**

关于 [SakuraKit](https://github.com/tingxins/SakuraKit) 具体使用，详见 Demo。

GitHub 项目地址：<https://github.com/tingxins/SakuraKit>