伴随这iOS 8 系统多达4000项API更新而来同样还有Today Extension。而对iOS而言，有了Today Extension 开发者可以很好借助系统提供的接入点为系统定制的服务，提供自定义的附加功能.这意味着什么呢？从iOS 7版本尝试开路到现在iOS 8更新的到来终于向开发者开放Widget接入，这意味着系统应用和第三方应用都可以通知中心(Notification Center)里面实现交互。

Notification Center Widget [Via Apple]

其实相对于Android，因其特有开放性Widget插件已经发展了很多年，拥有极高自由定制性，在新版本的Android系统中甚至可以将部分插件摆在锁屏页.而Google和各大软件厂商制作的Widget插件也能很好与系统的整体风格进行无缝的融合，而直到目前iOS 8版本中，Widget也就只是能摆在通知中心(Notification Center)今天通知栏中而已，相对于Android也听到很多人把这个作为"iOS不够开放"一个有力的依据。针对这个问题其实Apple也在iOS Human Interface Guidelines中提到：

iOS 8 中开发者的中心并不应该发生改变，依然应该是围绕 app.在 app 中提供优秀交互和有用的功能，现在是，将来也会是 iOS 应用开发的核心任务。而Widget在 iOS 中是不能以单独的形式存在的,一定是随着一个应用一起打包提供的。

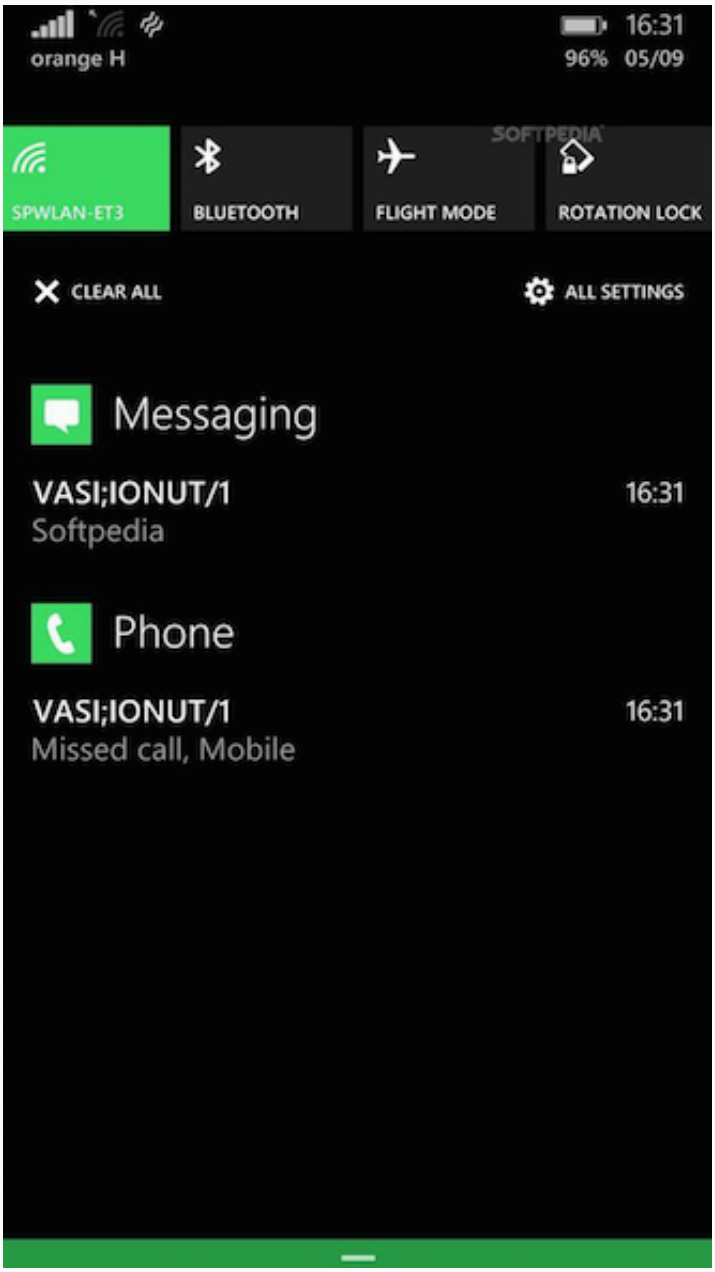
从这个侧面可见，Apple对开放一直持有审慎的态度，开放的目的是力求保证整体体验完整性，虽然iOS的Widget相比Android自定义性太低，但基于Apple目前的开放程度而言是能够很有效控制Widget与系统的更好的融合.虽似戴着镣铐起舞，但却能捕获人心。

而从用户角度来看，在无需打开应用前提下就可以对消息进行处理的交互特性,使它在很多场景里有效提升了用户操作效率.例如在Widget中快速回复email，即时完成Todo日程等.这种交互更多从更宏观角度重新定义了消息,通知中心(Notification Center)通过获取用户上一行为，还可以起到承接下一行为的作用(虽然目前

开放API只能做到系统级的行为).点虽小，但这对用户使用习惯改变却是巨大的。

Widget on hands [Via Yalantis]

有人看到这肯定一定会问为何没有提到Windows Phone平台?因为无论从通知中心快捷入口数量还是谈到可以交互的点一句话而概之WP的现状是“一穷二白”，你想作为曾经走过WP7时代用户根本不知道通知中心为何物的，而是用了足足两年时间WP8上才有体现，而那些被其他平台玩腻的希望习以为常通知中心交互，就像这样：



WP 通知中心[Via PCGGroup]

你就像看这张静态图片一样也就是停留只是看看程度而已（除了删除操作之外）,MS针对通知中心现在最新消息是未来会支持类似可以通知中心直接回复短信等交互，至于什么时候能够等到，谁知道呢.

说了这么多，回归正题.

1.交互

在开始构建Widget之前，如果想对Widget实现技术细节和交互特点有一个完整概览，我觉得没有什么文档比官方App Extension Programming Guide更值得一读了.刚开始接触iOS通知中心，一直很疑惑为何通知中心采用两个不同Tab“今日”和“通知”来对消息进行分离.其实这和Widget工作机制有关.

Widget是放在“今日”Tab之中，而它工作机制是只有用户下拉通知中心时才会去刷新获取最新数据，这种做

法和Android不同在于，Android更偏向于把整个Widget一直放在后台实时持续的更新.设想一下，如果我们看同样天气信息，Android会持续消耗资源去做一件用户不会实时预览信息,这也就能解释为何经常看到Android用户抱怨耗电问题.而对于即时消息，iOS做法是直接把这些消息实时归类到”通知“Tab中.其实这种做法很好解决采用消耗最少资源前提下保证其操作的灵活性.

因为现有Widget一般来说是展现在系统级别的 UI上，所以在App Extension Programming Guide中Apple对Widget交互提出如下明确的要求:

扩展应该保持轻巧迅速，并且专注功能单一，在不打扰或者中断用户使用当前应用的前提下完成自己的功能点.

类似一直挚爱Todo应用Clear则交互上堪称上典范:

Clear's Widget

当然如果动点脑子会发现，Widget开放iOS上实现应用之间Launcher成为了可能，类似早期一直很魔性应用"Launcher":

Launcher's Widget

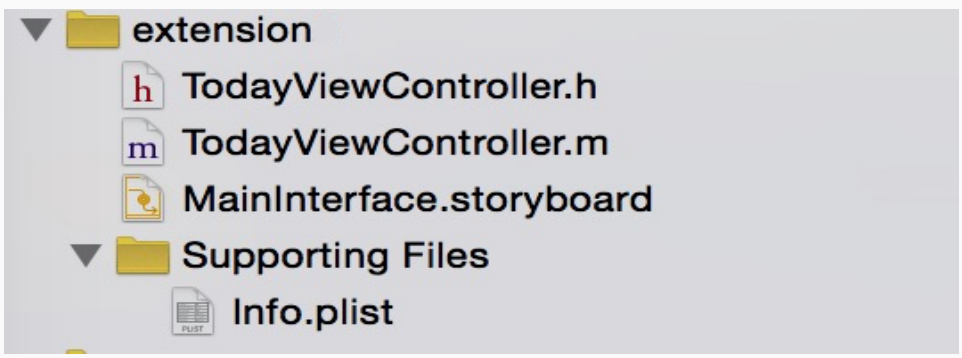
可以让用在 iOS 的通知中心里，以类似应用程序捷径的方式直接快速切换 App 的小工具，其实当初在推出没多久后，便被 Apple 以"误用 / 滥用"Widgets 为理由下架，但有意思的就在几天前3月20日又重新上架.

2.构建

在Widget技术实现细节上，并不打算在本篇把所有技术细节通览一遍，我只会写我个人(其实就是初学者)认为值得写的容易出错的点或者耗费一些时间找到一些问题的解决方案.

2.1 纯代码构建

Xcode 6中已经支持Today Extension创建Widget的模板，该模板会默认创建MainInterface.storyboard文件来构建UI:



StoryBoard UI

当然对于一个纯代码的拥趸而言，肯定直接删除storyboard文件采用纯代码方式来进行构建，删除完后之后注意需要找到Supporting Files下面的Info.plist中NSExtension字段做如下两个操作:

A：直接删除NSExtensionMainStoryboard字段

B：添加NSExtensionPrincipalClass字段 并设为TodayViewController

如下:

修改后

注意当采用Xcode默认模板创建Widget时会自动把ViewController文件命名设置为“TodayViewController”.当然这个ViewController命名其实是可以修改的，唯一值得注意的修改该ViewController文件命名后还需要设置NSExtensionPrincipalClass的值与其保持一致即可.不然Widget编译时会报找不到对应入口.

2.2 左侧间隔

当第一次添加UI元素采用真机来运行Widget会发现，Widget左侧到屏幕之间始终会有一段距离的间隔，导致调整布局和效果图差距甚远，类似这样:

左侧间隔

其实这个问题主要是因为Widget里面的视图默认居左居下都会有一定距离的间隔，可以采用如下方式取消间隔，使布局区域填充整个Widget:

```
- (UIEdgeInsets)widgetMarginInsetsForProposedMarginInsets:(UIEdgeInsets)defaultMarginInsets{
    return UIEdgeInsetsMake(0, 0, 0, 0);
}
```

取消间隔

这种方式把整个布局填充区域间隔都设置为0，当然更简洁的方式是你可以直接采用“return UIEdgeInsetsZero;”方式.而关于Widget上布局处理则采用Masonry框架做的相对布局,简单快捷推荐.当然关于Masonry框架快速上手则不得不推荐阅读Masonry介绍与使用实践(快速上手Autolayout).

2.3 整个点击区域实现

如你所看当用户拉开Widget时，因为Widget是依赖于应用程序在分发时是跟应用程序一块打包的，希望点击Widget布局任何区域都能唤起主应用程序,常用的方式在整个View增加Tap事件订阅处理:

```
//点击打开主程序
UITapGestureRecognizer *tapGesture = [[UITapGestureRecognizer alloc]initWithTarget:self action:@selector(openApp)];
[self.view addGestureRecognizer:tapGesture];
```

Tap事件

但这种方式会额外产生一个问题，如果Widget空白区域没有任何UI元素则无法触发该事件,那这里有一个小技巧可以解决改问题,可以整个Widget增加一个透明的ImageView:

```
_ivWholeBackground = [[UIImageView alloc] init];
_ivWholeBackground.userInteractionEnabled = YES;
_ivWholeBackground.alpha = 0.01;
_ivWholeBackground.backgroundColor = [UIColor grayColor];
[self.view addSubview:_ivWholeBackground];//Widget整个背景
```

设置透明度

初始化时注意把imageview透明度设置为0.01最小值，那么无论设置其背景色为什么值肉眼都是不可见的.然后使用Masonry框架布局来填充Widget整个背景如下:

```
[_ivWholeBackground mas_makeConstraints:^(MASConstraintMaker *make) {
    make.top.equalTo(superView.mas_top);
    make.leading.equalTo(superView.mas_leading);
    make.trailing.equalTo(superView.mas_trailing);
    make.width.equalTo(superView.mas_width);
    make.height.equalTo(superView.mas_height);
}];//Widget整个背景
```

填充整个背景

然后为imageview增加Tap事件订阅即可:

```
//点击打开主程序
UITapGestureRecognizer *tapGesture = [[UITapGestureRecognizer alloc]initWithTarget:self action:@selector(openApp)];
[_ivWholeBackground addGestureRecognizer:tapGesture];
[self.view addGestureRecognizer:tapGesture];
```

增加事件订阅

这样就能整个Widget区域可点击效果.另外针对通过Widget中唤起主应用程序方式目前只支持url scheme方式来实现.同时也是Widget向主应用程序反馈数据和交互的渠道之一.

2.4 定时更新机制

Widget自身更新机制当用户下拉通知中心(Notification Center)时立即更新数据，但我们仔细研究Widget用

户使用场景时发现，如果用户锁屏时间过长，打开Widget后不做任何操作，这个时候针对一些即时类应用，类似我们天气中可能涉及到灾害预警它要求场景数据一旦产生就要实时展现给用户，这就需要我们基于Widget自身机制外还要处理这个场景下天气数据自动更新的问题.

这个时候我们需要构建一个定时更新的NSTimer:

```
- (void)initAutoUpdateTimer{
    updateTimer = [NSTimer scheduledTimerWithTimeInterval:kAutoUpdateInterval
                                                   target:self
                                                 selector:@selector(autoUpdateWidgetWeatherData)
                                                 userInfo:nil
                                                  repeats:YES];
    [[NSRunLoop currentRunLoop] addTimer:updateTimer forMode:NSDefaultRunLoopMode];
}
```

初始化NSTimer

非常简单，在NSTimer固定更新间隔执行的方法调用就是更新数据方法，当然重点不在这里，而是触发和关闭这个NSTimer时机.按照Widget生命周期来说，如果用户是第一次下拉查看Widget其实就是执行整个ViewController生命周期调用过程，这个并没有什么问题，但是还是存在一个特殊情况.系统为了保证Widget上数据是及时更新的,默认会截取上次显示成功Widget的快照.这个快照会一直保存到新的数据或UI被更新才回被替换，那这就会带来一个问题,当你拖拽通知中心(Notification Center)下拉过于频繁时，Debug跟踪代码执行路径你会发现整个Widget生命周期执行过程和第一次下拉执行的路径发生了变化.

第一次下拉执行路径是viewDidLoad->viewWillAppear,而如果下拉过于频繁你就会发现代码执行路径直接只会执行viewWillAppear方法，这个就是系统默认保存上次快照而导致的执行路径上变化.这对我们选择NSTimer更新时机以及后面会提到的Widget横竖屏处理都会有影响.

那么很明显，为了保证这个定时更新机制能够无论用户什么情况下操作都能起作用，我们需要把NSTimer fire触发代码调用放到viewWillAppear方法中来.同理当Widget关闭后在viewDidDisappear方法取消NSTimer invalidate定时更新即可.

2.5 Widget横屏支持

关于Widget横屏支持在开发中耽误一点时间来解决这个问题,在iPhone 6 & Plus上已经横竖屏直接切换，Widget默认是竖屏，但如果你需求中横屏UI的布局和竖屏布局完全不同，这个时候你就需要判断当前Widget横竖屏状态来切换对应的布局.

当然一般思路我们都会按照端内处理横竖屏方式来处理Widget，如果你翻过官方的开发文档，你会发现在iOS 6.0版本之前UIViewController之间横竖屏切换，只需要设置shouldAutorotateToInterfaceOrientation函数即可.UIInterfaceOrientation是UIApplication.h头文件中定义的枚举类型，总共有四个方向.在shouldAutorotateToInterfaceOrientation方法中返回相应的结果即可，如果直接返回YES将支持所有方向.而

在iOS 6.0版本之后，UIViewController之间横竖屏切换需要多设置一个supportedInterfaceOrientations函数返回UIInterfaceOrientationMask枚举类型.除了设置shouldAutorotateToInterfaceOrientation之外,还要将supportedInterfaceOrientations返回的方向与shouldAutorotateToInterfaceOrientation保持一致，否则会在两个支持不同横竖屏ViewController中切换时，会出现竖屏变横屏，横屏变竖屏的情况.但问题是这种方式是否适用Widget横屏处理呢?

使用UIDeviceOrientationIsPortrait来判断:

```
//判断横屏 方法一
[[UIDevice currentDevice] beginGeneratingDeviceOrientationNotifications];
UIDeviceOrientation orientation = [[UIDevice currentDevice] orientation];
BOOL isPortrait = UIDeviceOrientationIsPortrait(orientation);
if (isPortrait) {
    NSLog(@"横屏");
}else{
    NSLog(@"竖屏");
}
```

判断横屏方法一

当你执行这段代码调试时你会发现，orientation方向的值始终都会是UIDeviceOrientationUnknown.如果你点开UIDeviceOrientation枚举你会看到.它包含了两个扁平方向UIDeviceOrientationFaceUp和UIDeviceOrientationFaceDown，其实它代表的意思屏幕朝上或朝下平躺两个方向的判断.所以当你设备平躺桌面时.即时你有时已经切换了横屏你会发现它会返回FaceUp或FaceDown，所以你当你调用UIDeviceOrientationIsPortrait方法时它返回值其实是没有意义的，因为设备目前方向在平躺下Faceup和FaceDown既不是横屏也不是竖屏.难道没有更好的方式嘛?

可以采用如下方式能够完美解决Widget横竖屏切换状态判断的问题:

```
- (BOOL)isPortrait {
    BOOL screenIsPortrait = FALSE;
    CGFloat scale = [UIScreen mainScreen].scale;
    CGSize nativeSize = [UIScreen mainScreen].currentMode.size;
    CGSize sizeInPoints = [UIScreen mainScreen].bounds.size;

    if(scale * sizeInPoints.width == nativeSize.width){
        screenIsPortrait = TRUE;//竖屏
    }
    return screenIsPortrait;
}
```

Widget横竖屏状态判断

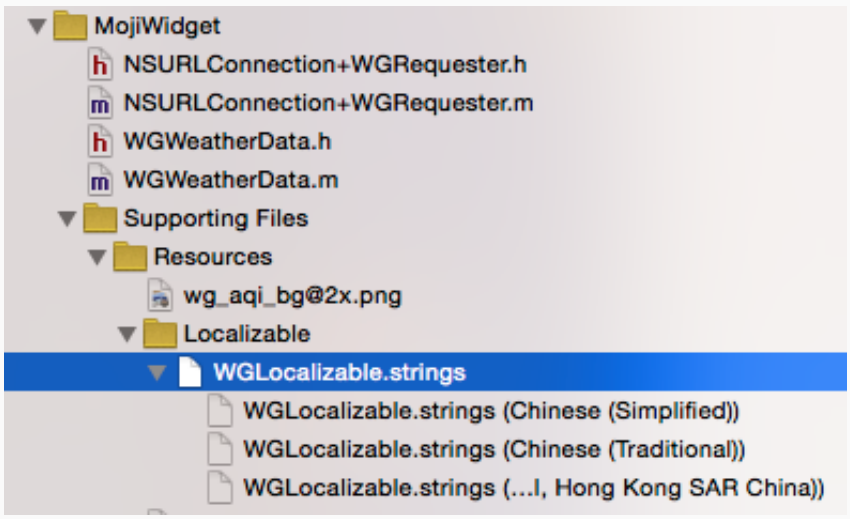
其实设置Widget显示高度时就会发现，高度在横竖屏状态切换是不会变化的，但宽度会随着横竖屏状态切换会发生变化，所以判断屏幕宽度这个思路是可取的.因为横竖屏UI布局不同，调用时机则可以选择在

viewWillLayoutSubviews或viewDidLayoutSubviews方法中进行.因为这两个方法都是viewWillAppear方法是必然执行的，这也就自然规避Widget自身因为下拉快照保存机制导致代码执行路径变化导致布局更新的问题.

2.6 Widget国际化

在来说说这个Widget国际化，因为我们客户端自身已经支持三种不同语言,这就是导致Widget也是需要根据端内语言变化必须有国际化的支持.其实我们端内已经做了一套完整的国际化机制.Widget最好处理方式能够复用端内机制，而不需要单独开发支持.iOS 8 新引入的自制 framework 的方式来组织需要重用的代码，这样在链接 framework 后 app 和Widget就都能使用相同的代码. 包含Widget中数据请求和数据记忆其他能够复用的代码。

这也是我们一开始打算解决方式，但发现剥离这部分代码时间周期明显超过我们预期.所以在国际化处理上我们Widget独立做了一套国际化处理，它和端内在处理机制上并没有多大的不同：



Widget国际化处理

当然重点不再于它的实现，你可以发现我们Widget中国际化文本文件Locallizable.string命名加了一个"WG",这个问题是刚开始开发之初我们一直认为Widget作为端是独立于主应用程序的.所以当初理解为只有把这个文件命名为的“Locallizable.string”才是正常的能够被识别的，但我们调试时发现，Widget打包时会把这些国际化单独放到PlugIns文件下,这里给出一个简体中文全路径：

/private/var/mobile/Containers/Bundle/Application/61C637FF-B5BC-432A-ADD5-BA64EBFE98E8/MojiWeather.app/PlugIns/MojiWidget.appex/zh-Hans.lproj

根据这个路径你会发现文件时可以找到的，但调试时发现国际化取对应Key的值一直是取不到的，但我们任意非“Locallizable.string”时则是没有问题的，后来我们发现当我们打包在不同机型上测试这个问题时，如

果“Locallizable.string”名称命名会导致调试时ok，而最终打包上会出现找不到对应key值得问题.这个原因到我写这篇blog一直没有找到具体的原因.所以我们给出解决方案是一定要和主应用程序“Locallizable.string”保持不同即可解决.

当然关于Widget中闪现的问题，因为我们Widget存在两个不同尺寸切换，导致这个问题很明显，处理方式自然是viewWillLoad方式中做好Widget高度在不同场景高度初始化就可以完美避免.这里就不做赘述.