View-Layer 协作

onevcat (http://onev.cat) 2014/05/13

在 iOS 中，所有的 view 都是由一个底层的 layer 来驱动的。view 和它的 layer 之间有着紧密的联系，view 其实直接从 layer 对象中获取了绝大多数它所需要的数据。在 iOS 中也有一些单独的 layer，比如 `AVCaptureVideoPreviewLayer` 和 `CAShapeLayer`，它们不需要附加到 view 上就可以在屏幕上显示内容。两种情况下其实都是 layer 在起决定作用。当然了，附加到 view 上的 layer 和单独的 layer 在行为上还是稍有不同的。

基本上你改变一个单独的 layer 的任何属性的时候，都会触发一个从旧的值过渡到新值的简单动画（这就是所谓的可动画 `animatable`）。然而，如果你改变的是 view 中 layer 的同一个属性，它只会从这一帧直接跳变到下一帧。尽管两种情况中都有 layer，但是当 layer 附加在 view 上时，它的默认的隐式动画的 layer 行为就不起作用了。

> animatable；几乎所有的层的属性都是隐性可动画的。你可以在文档中看到它们的简介是以 'animatable' 结尾的。这不仅包括了比如位置，尺寸，颜色或者透明度这样的绝大多数的数值属性，甚至也囊括了像 isHidden 和 doubleSided 这样的布尔值。 像 paths 这样的属性也是 animatable 的，但是它不支持隐式动画。

在 Core Animation 编程指南的 “How to Animate Layer-Backed Views” 中，对*为什么*会这样做出了一个解释：

> UIView 默认情况下禁止了 layer 动画，但是在 animation block 中又重新启用了它们

这正是我们所看到的行为；当一个属性在动画 block 之外被改变时，没有动画，但是当属性在动画 block 内被改变时，就带上了动画。对于这是_如何_发生的这一问题的答案十分简单和优雅，它优美地阐明和揭示了 view 和 layer 之间是如何协同工作和被精心设计的。

无论何时一个可动画的 layer 属性改变时，layer 都会寻找并运行合适的 'action' 来实行这个改变。在 Core Animation 的专业术语中就把这样的动画统称为动作 (action，或者 `CAAction`)。

> CAAction：技术上来说，这是一个接口，并可以用来做各种事情。但是实际中，某种程度上你可以只把它理解为用来处理动画。

layer 将像文档中所写的的那样去寻找动作，整个过程分为五个步骤。第一步中的在 view 和 layer 中交互的部分是最有意思的：

layer 通过向它的 delegate 发送 `actionForLayer:forKey:` 消息来询问提供一个对应属性变化的 action。delegate 可以通过返回以下三者之一来进行响应：

1. 它可以返回一个动作对象，这种情况下 layer 将使用这个动作。
2. 它可以返回一个 `nil`， 这样 layer 就会到其他地方继续寻找。
3. 它可以返回一个 `NSNull` 对象，告诉 layer 这里不需要执行一个动作，搜索也会就此停止。

而让这一切变得有趣的是，当 layer 在背后支持一个 view 的时候，view 就是它的 delegate；

> 在 iOS 中，如果 layer 与一个 UIView 对象关联时，这个属性 必须 被设置为持有这个 layer 的那个 view。

理解这些之后，前一分钟解释起来还复杂无比的现象瞬间就易如反掌了：属性改变时 layer 会向 view 请求一个动作，而一般情况下 view 将返回一个 `NSNull`，只有当属性改变发生在动画 block 中时，view 才会返回实际的动作。哈，但是请别轻信我的这些话，你可以非常容易地验证到底是不是这样。只要对一个一般来说可以动画的 layer 属性向 view 询问动作就可以了，比如对于 'position'：

```
NSLog(@"outside animation block: %@",
      [myView actionForLayer:myView.layer forKey:@"position"]);

[UIView animateWithDuration:0.3 animations:^{
    NSLog(@"inside animation block: %@",
          [myView actionForLayer:myView.layer forKey:@"position"]);
}];
```

运行上面的代码，可以看到在 block 外 view 返回的是 NSNull 对象，而在 block 中时返回的是一个 CABasicAnimation。很优雅，对吧？值得注意的是打印出的 NSNull 是带着一对尖括号的 (" `<null>` ")，这和其他对象一样，而打印 nil 的时候我们得到的是普通括号(`(null)`)：

```
outside animation block: <null>
inside animation block: <CABasicAnimation: 0x8c2ff10>
```

对于 view 中的 layer 来说，对动作的搜索只会到第一步为止（至少我没有见过 view 返回一个 `nil` 然后导致继续搜索动作的情况）。对于单独的 layer 来说，剩余的四个步骤可以在 CALayer 的 `actionForKey:` 文档 (https://developer.apple.com/library/mac/documentation/graphicsimaging/reference/CALayer_class/Introduction/Introduction.ht 中找到。

从 UIKit 中学习

我很确定我们都会同意 UIView 动画是一组非常优秀的 API，它简洁明确。实际上，它使用了 Core Animation 来执行动画，这给了我们一个绝佳的机会来深入研究 UIKit 是如何使用 Core Animation 的。在这里甚至还有很多非常棒的实践和技巧可以让我们借鉴。:)

当属性在动画 block 中改变时，view 将向 layer 返回一个基本的动画，然后动画通过通常的 `addAnimation:forKey:` 方法被添加到 layer 中，就像显式地添加动画那样。再一次，别直接信我，让我们实践检验一下。

归功于 UIView 的 `+layerClass` 类方法，view 和 layer 之间的交互很容易被观测到。通过这个方法我们可以在为 view 创建 layer 时为其指定要使用的类。通过子类一个 UIView，以及用这个方法返回一个自定义的 layer 类，我们就可以重写 layer 子类中的 `addAnimation:forKey:` 并输出一些东西来验证它是否确实被调用。唯一要记住的是我们需要调用 super 方法，不然的话我们就把要观测的行为完全改变了：

```
@interface DRInspectionLayer : CALayer
@end

@implementation DRInspectionLayer
- (void)addAnimation:(CAAnimation *)anim forKey:(NSString *)key
{
    NSLog(@"adding animation: %@", [anim debugDescription]);
    [super addAnimation:anim forKey:key];
}
@end


@interface DRInspectionView : UIView
@end

@implementation DRInspectionView
+ (Class)layerClass
{
    return [DRInspectionLayer class];
}
@end
```

通过输出动画的 debug 信息，我们不仅可以验证它确实如预期一样被调用了，还可以看到动画是如何组织构建的：

```
<CABasicAnimation:0x8c73680;
    delegate = <UIViewAnimationState: 0x8e91fa0>;
    fillMode = both;
    timingFunction = easeInEaseOut;
    duration = 0.3;
    fromValue = NSPoint: {5, 5};
    keyPath = position
>
```

当动画刚被添加到 layer 时，属性的新值还没有被改变。在构建动画时，只有 `fromValue` (也就是当前值) 被显式地指定了。CABasicAnimation 的文档 (https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CABasicAnimation_class/Introduction/Intro 向我们简单介绍了这么做对于动画的插值来说的的行为应该是：

> 只有 `fromValue` 不是 `nil` 时，在 `fromValue` 和属性当前显示层的值之间进行插值。

这也是我在处理显式动画时选择的做法，将一个属性改变为新的值，然后将动画对象添加到 layer 上：

```
CABasicAnimation *fadeIn = [CABasicAnimation animationWithKeyPath:@"opacity"];
fadeIn.duration  = 0.75;
fadeIn.fromValue = @0;

myLayer.opacity = 1.0; // 更改 model 的值 ...
// ... 然后添加动画对象
[myLayer addAnimation:fadeIn forKey:@"fade in slowly"];
```

这很简洁，你也不需要在动画被移除的时候做什么额外操作。如果动画是在一段延迟后才开始的话，你可以使用 backward 填充模式 (或者 'both' 填充模式)，就像 UIKit 所创建的动画那样。

可能你看见上面输出中的动画的 delegate 了，想知道这个类是用来做什么的吗？我们可以来看看 dump 出来的头文件 (https://github.com/rpetrich/iphoneheaders/blob/master/UIKit/UIViewAnimationState.h)，它主要用来维护动画的一些状态 (持续时间，延时，重复次数等等)。它还负责对一个栈做 push 和 pop，这是为了在多个动画 block 嵌套时能够获取正确的动画状态。这些都是些实现细节，除非你想要写一套自己的基于 block 的动画 API，否则可能你不会用到它们 (实际上这是一个很有趣的点子)。

然后真正*有意思*的是这个 delegate 实现了 `animationDidStart:` 和 `animationDidStop:finished:`，并将信息传给了它自己的 delegate。

> **编者注** 这里不太容易理解，加以说明：从上面的头文件中可以看出，作为 CAAnimation 的 delegate 的私有类 `UIViewAnimationState` 中还有一个 `_delegate` 成员，并且 `animationDidStart:` 和 `animationDidStop:finished:` 也是典型的 delegate 的实现方法。

通过打印这个 delegate 的 delegate，我们可以发现它也是一个私有类：UIViewAnimationBlockDelegate。同样进行 class dump 得到它的头文件 (https://github.com/EthanArbuckle/IOS-7-Headers/blob/master/Frameworks/UIKit.framework/UIViewAnimationBlockDelegate.h)，这是一个很小的类，只负责一件事情：响应动画的 delegate 回调并且执行相应的 block。如果我们使用自己的 Core Animation 代码，并且选择 block 而不是 delegate 做回调的话，添加这个是很容易的：

```
@interface DRAnimationBlockDelegate : NSObject

@property (copy) void(^start)(void);
@property (copy) void(^stop)(BOOL);

+(instancetype)animationDelegateWithBeginning:(void(^)(void))beginning
                                   completion:(void(^)(BOOL finished))completion;

@end

@implementation DRAnimationBlockDelegate

+ (instancetype)animationDelegateWithBeginning:(void (^)(void))beginning
                                    completion:(void (^)(BOOL))completion
{
    DRAnimationBlockDelegate *result = [DRAnimationBlockDelegate new];
    result.start = beginning;
    result.stop  = completion;
    return result;
}
```

```
- (void)animationDidStart:(CAAnimation *)anim
{
    if (self.start) {
        self.start();
    }
    self.start = nil;
}

- (void)animationDidStop:(CAAnimation *)anim finished:(BOOL)flag
{
    if (self.stop) {
        self.stop(flag);
    }
    self.stop = nil;
}

@end
```

虽然是我个人的喜好，但是我觉得像这样的基于 block 的回调风格可能会比实现一个 delegate 回调更适合你的代码：

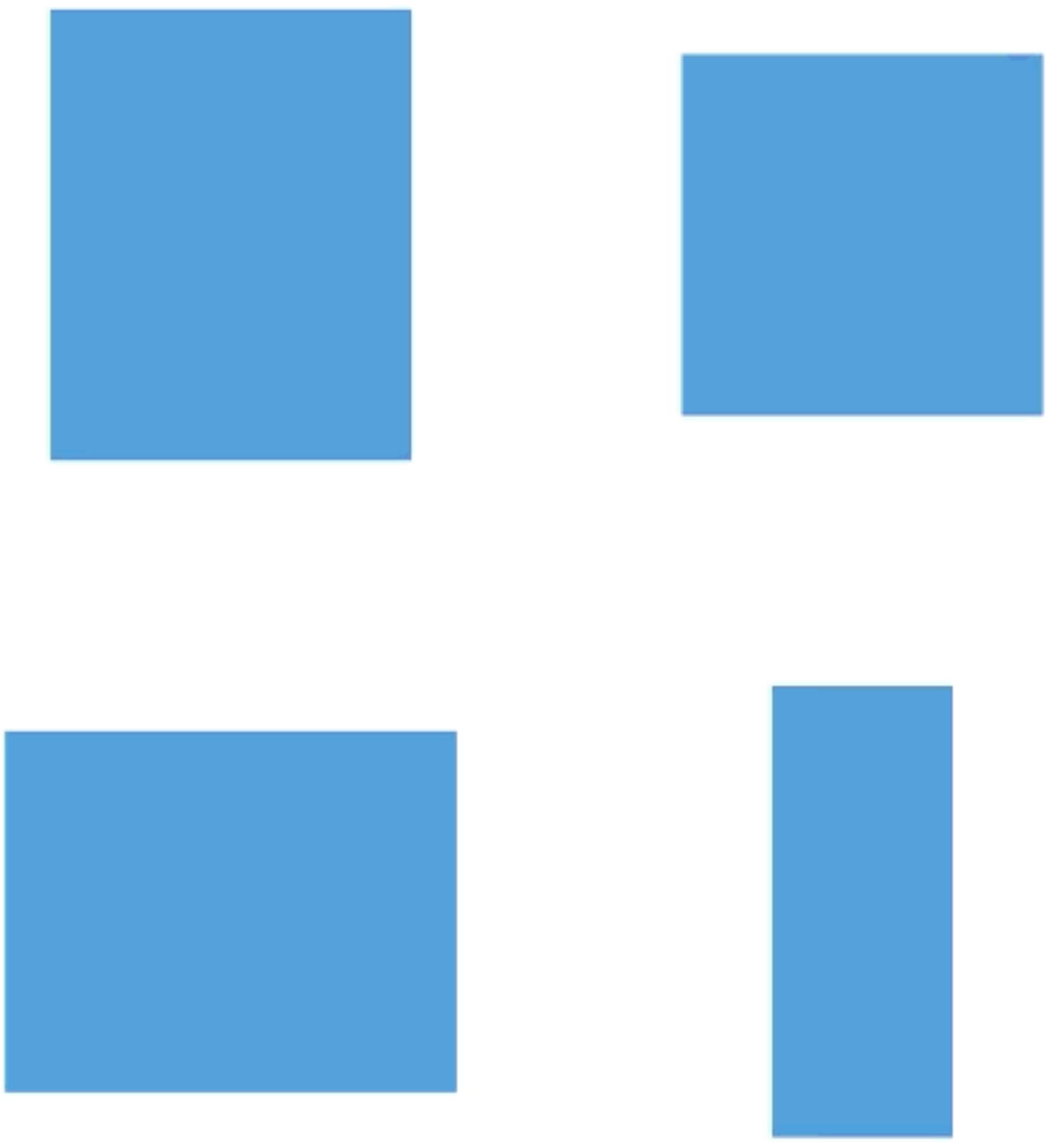
```
fadeIn.delegate = [DRAnimationBlockDelegate animationDelegateWithBeginning:^{
    NSLog(@"beginning to fade in");
} completion:^(BOOL finished) {
    NSLog(@"did fade %@", finished ? @"to the end" : @"but was cancelled");
}];
```

自定义基于 block 的动画 APIs

一旦你知道了 `actionForKey:` 的机理之后，UIView 就远没有它一开始看起来那么神秘了。实际上我们完全可以按照我们的需求量身定制地写出一套自己的基于 block 的动画 APIs。我所设计的动画将通过在 block 中用一个很激进的时间曲线来做动画，以吸引用户对该 view 的注意，之后做一个缓慢的动画回到原始状态。你可以把它看作一种类似 pop (请不要和 Facebook 最新的 Pop 框架弄混了)的行为。与一般使用 `UIViewAnimationOptionAutoreverse` 的动画 block 不同，因为动画设计和概念上的需要，我自己实现了将 model 值改变回原始值的过程。自定义的动画 API 的使用方法就像这样：

```
[UIView DR_popAnimationWithDuration:0.7
                         animations:^{
                             myView.transform = CGAffineTransformMakeRotation(M_PI_2);
                         }];
```

当我们完成后，效果是这个样子的 (对四个不同的 view 为位置，尺寸，颜色和旋转进行动画)：



要开始实现它，我们首先要做的是当一个 layer 属性变化时获取 delegate 的回调。因为我们无法事先预测 layer 要改变什么，所以我选择在一个 UIView 的 category 中 swizzle `actionForLayer:forKey:` 方法：

```
@implementation UIView (DR_CustomBlockAnimations)

+ (void)load
{
    SEL originalSelector = @selector(actionForLayer:forKey:);
    SEL extendedSelector = @selector(DR_actionForLayer:forKey:);

    Method originalMethod = class_getInstanceMethod(self, originalSelector);
    Method extendedMethod = class_getInstanceMethod(self, extendedSelector);

    NSAssert(originalMethod, @"original method should exist");
    NSAssert(extendedMethod, @"exchanged method should exist");

    if(class_addMethod(self, originalSelector, method_getImplementation(extendedMethod), method_getTyp
eEncoding(extendedMethod))) {
        class_replaceMethod(self, extendedSelector, method_getImplementation(originalMethod), method_g
etTypeEncoding(originalMethod));
    } else {
        method_exchangeImplementations(originalMethod, extendedMethod);
    }
}
```

为了保证我们不破坏其他依赖于 `actionForLayer:forKey:` 回调的代码，我们使用一个静态变量来判断现在是不是处于我们自己定义的上下文中。对于这个例子来说一个简单的 `BOOL` 其实就够了，但是如果我们之后要写更多内容的话，上下文的话就要灵活得多了：

```
static void *DR_currentAnimationContext = NULL;
static void *DR_popAnimationContext     = &DR_popAnimationContext;

- (id<CAAction>)DR_actionForLayer:(CALayer *)layer forKey:(NSString *)event
{
    if (DR_currentAnimationContext == DR_popAnimationContext) {
        // 这里写我们自定义的代码...
    }

    // 调用原始方法
    return [self DR_actionForLayer:layer forKey:event]; // 没错，你没看错。因为它们已经被交换了
}
```

在我们的实现中，我们要确保在执行动画 block 之前设置动画的上下文，并且在执行后恢复上下文：

```
+ (void)DR_popAnimationWithDuration:(NSTimeInterval)duration
                         animations:(void (^)(void))animations
{
    DR_currentAnimationContext = DR_popAnimationContext;
    // 执行动画（它将触发交换后的 delegate 方法）
    animations();
    /* 一会儿再添加 */
    DR_currentAnimationContext = NULL;
}
```

如果我们想要做的不过是添加一个从旧的值向新的值过度的动画的话，我们可以直接在 delegate 的回调中来做。然而因为我们想要更精确地控制动画，我们需要用一个帧动画来实现。帧动画需要所有的值都是已知的，而对我们的情况来说，新的值还没有被设定，因此我们也就无从知晓。

有意思的是，iOS 添加的一个基于 block 的动画 API 也遇到了同样的问题。使用和上面一样的观察手段，我们就能知道它是如何绕开这个麻烦的。对于每个关键帧，在属性变化时，view 返回 `nil`，但是却存储下需要的状态。这样就能在所有关键帧 block 执行后创建一个 `CAKeyframeAnimationz` 对象。

受到这种方法的启发，我们可以创建一个小的类来存储我们创建动画时所需要的信息：什么 layer 被更改了，什么 key path 的值被改变了，以及原来的值是什么：

```
@interface DRSavedPopAnimationState : NSObject

@property (strong) CALayer  *layer;
@property (copy)   NSString *keyPath;
@property (strong) id        oldValue;

+ (instancetype)savedStateWithLayer:(CALayer *)layer
                            keyPath:(NSString *)keyPath;

@end

@implementation DRSavedPopAnimationState

+ (instancetype)savedStateWithLayer:(CALayer *)layer
                            keyPath:(NSString *)keyPath
{
    DRSavedPopAnimationState *savedState = [DRSavedPopAnimationState new];
    savedState.layer    = layer;
    savedState.keyPath  = keyPath;
    savedState.oldValue = [layer valueForKeyPath:keyPath];
    return savedState;
}

@end
```

接下来，在我们的交换后的 delegate 回调中，我们简单地将被变更的属性的状态存入一个静态可变数组中：

```
if (DR_currentAnimationContext == DR_popAnimationContext) {
      [[UIView DR_savedPopAnimationStates] addObject:[DRSavedPopAnimationState savedStateWithLayer:la
yer
                                                                                  keyPath:event]];
    // 没有隐式的动画（稍后添加）
    return (id<CAAction>)[NSNull null];
}
```

在动画 block 执行完毕后，所有的属性都被变更了，它们的状态也被保存了。现在，创建关键帧动画：

```
+ (void)DR_popAnimationWithDuration:(NSTimeInterval)duration
                         animations:(void (^)(void))animations
{
    DR_currentAnimationContext = DR_popAnimationContext;

    // 执行动画（它将触发交换后的 delegate 方法）
    animations();

    [[self DR_savedPopAnimationStates] enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL
*stop) {
        DRSavedPopAnimationState *savedState   = (DRSavedPopAnimationState *)obj;
        CALayer *layer    = savedState.layer;
        NSString *keyPath = savedState.keyPath;
        id oldValue       = savedState.oldValue;
        id newValue       = [layer valueForKeyPath:keyPath];

        CAKeyframeAnimation *anim = [CAKeyframeAnimation animationWithKeyPath:keyPath];

        CGFloat easing = 0.2;
        CAMediaTimingFunction *easeIn  = [CAMediaTimingFunction functionWithControlPoints:1.0 :0.0 :(
1.0-easing) :1.0];
        CAMediaTimingFunction *easeOut = [CAMediaTimingFunction functionWithControlPoints:easing :0.0
:0.0 :1.0];

        anim.duration = duration;
        anim.keyTimes = @[@0, @(0.35), @1];
        anim.values = @[oldValue, newValue, oldValue];
        anim.timingFunctions = @[easeIn, easeOut];

        // 不带动画地返回原来的值
        [CATransaction begin];
        [CATransaction setDisableActions:YES];
        [layer setValue:oldValue forKeyPath:keyPath];
        [CATransaction commit];

        // 添加 "pop" 动画
        [layer addAnimation:anim forKey:keyPath];

    }];

    // 扫除工作（移除所有存储的状态）
    [[self DR_savedPopAnimationStates] removeAllObjects];

    DR_currentAnimationContext = nil;
}
```

注意老的 model 值被设到了 layer 上，所以在当动画结束和移除后，model 的值和 presentation 的值是相符合的。

创建像这样的你自己的 API 不会对每种情况都很适合，但是如果你需要在你的应用中的很多地方都做同样的动画的话，这可以帮助你写出整洁的代码，并减少重复。就算你之后从来不会使用这种方法，实际做一遍也能帮助你搞懂 UIView block 动画的 APIs，特别是你已经在 Core Animation 的舒适区的时候，这非常有助于你的提高。

其他的动画灵感

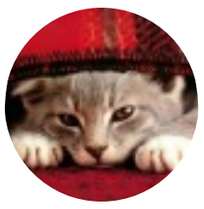
UIImageView 动画是一个完全不同的更高层次的动画 API 的实现方式，我会把它留给你来探索。表面上，它只不过是重新组装了一个传统的动画 API。你所要做的事情就是指定一个图片数组和一段时间，然后告诉 image view 开始动画。在抽象背后，其实是一个添加在 image view 的 layer 上的 contents 属性的离散的关键帧动画：

```
<CAKeyframeAnimation:0x8e5b020;
    removedOnCompletion = 0;
    delegate = <_UIImageViewExtendedStorage: 0x8e49230>;
    duration = 2.5;
    repeatCount = 2.14748e+09;
    calculationMode = discrete;
    values = (
        "<CGImage 0x8d6ce80>",
        "<CGImage 0x8d6d2d0>",
        "<CGImage 0x8d5cd30>"
    );
    keyPath = contents
>
```

动画 APIs 可以以很多不同形式出现，而对于你自己写的动画 API 来说，也是这样的。

原文 View-Layer Synergy (http://www.objc.io/issue-12/view-layer-synergy.html)

译者简介



onevcat (http://onev.cat)

王巍 (@onevcat) 是一名 iOS 和 Unity3D 开发者，现旅居日本，寻求创意之源