从零开始学iOS7开发系列3-我的地盘我做主-Cha19

说明:

本系列文章的原文及示例代码来自raywenderlich store中的iOS Apprentice 系列3教程，经过翻译和改编。

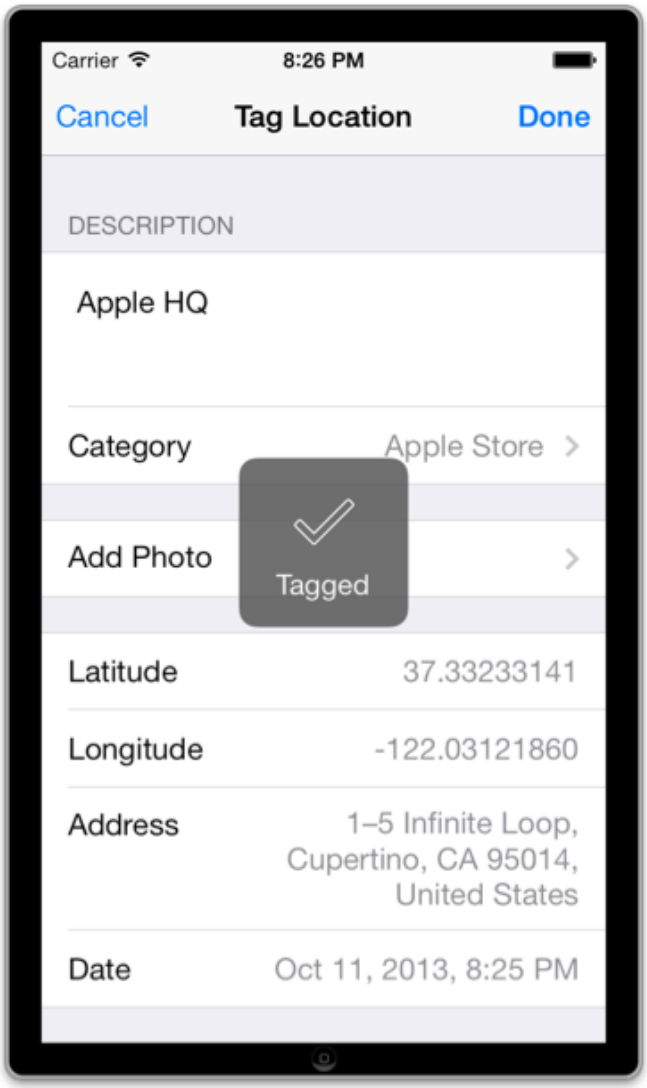
版权归原作者所有，本系列教程仅供学习参考使用，感兴趣的朋友建议购买原英文教程教程(The iOS Apprentice Second Edition: Learn iPhone and iPad Programming via Tutorials!)。
购买链接:
http://www.raywenderlich.com/store

欢迎继续我们的学习。

接下来我们将继续完善应用的交互体验。

首先是-HUD

为了让界面交互更加吸引人，我决定添加一个新的小玩意。当用户触碰Done按钮关闭界面时，应用会很快显示一个短动画，让用户知道自己已经成功保存了地址:



这种类型的显示通常被称为HUD（头戴式显示，Heads-Up Display）。好吧，移动应用和战斗机不同，但当我们下载文件或是执行某个比较耗时的任务时，会使用HUD来显示进度条或spinner（菊花？）。这里，我们将在界面关闭前显示自己的HUD视图一小会儿。通过这种小小的改进，可以让整个应用显得生气勃勃。

好吧，你可能会想，该如何在一个表视图的上面显示其它内容呢？答案是，这个HUD其实是一个UIView的子类。我们可以在视图的顶部添加其它视图。其实我们已经这样做过了：比如在cell的上面添加label，而label标签本身也是视图。cell本身则添加在表视图的上面，而表视图则添加在导航控制器的content view（内容视图）的上面。

到目前为止，我们所创建的定制化对象都是试图控制器或者数据模型对象，但实际上我们还可以创建定制化的视图。
虽然一般情况下使用标准的按钮或标签就足以实现我们想要的效果了，但如果想让自己的界面更加有特色，就需要制作定制化的视图。我们可以创建UIView或UIControl的子类，或者自己来绘制。

首先让我们来添加代码调用HUD。

在Xcode中切换到LocationDetailsViewController.m，更改done:方法的内容如下：

```
-(IBAction)done:(id)sender{

    HudView *hudView = [HudView hudInView:self.navigationController.view animated:YES];
    hudView.text = @"Tagged";
}
```

这里我们创建了一个HudView对象，然后将其添加到导航控制器的视图中，并带有一个动画。当然，我们需要手动编写所偶的代码，因为这不是一个常规的UIView。我们可以设置新对象的text属性。

在之前的done:方法中，我们会调用closeScreen方法来退出当前的视图控制器。不过为了测试的目的，我们先不要这么做。我们需要看看HudView是否实现了所想要的效果，因此先暂时删除关闭界面的代码。

当然，我们还需要在LocationDetailsViewController.m的顶部导入对应的头文件。
```
#import “HudView.h"
```

当然，Xcode肯定不会让你很爽，因为现在项目中根本就没有HudView的源文件。因此我们需要手动创建对应的文件。

在项目中使用Objective-C class模板创建一个新文件，然后设置其子类为UIView，名称为HudView。

```
@interface HudView : UIView


+(instancetype)hudInView:(UIView*) view
                animated:(BOOL)animated;

@property(nonatomic,strong) NSString *text;

@end
```

在HudView类的接口声明部分只包含了我们刚才调用的方法，以及一个text属性变量。

让我们先创建该类的最简单版本，这样就可以先在界面上看到一些内容，然后再进行优化。

使用以下代码替代HudView.m中的内容：

```
#import "HudView.h"

@implementation HudView

+(instancetype)hudInView:(UIView *)view animated:(BOOL)animated{

    HudView *hudView = [[HudView alloc]initWithFrame:view.bounds];
    hudView.opaque = NO;

    [view addSubview:hudView];
    view.userInteractionEnabled = NO;

    hudView.backgroundColor = [UIColor colorWithRed:1.0f green:0 blue:0 alpha:0.5f];

    return hudView;

}

@end
```

这里的hudInView:animated:方法是一个convenience constructor(便利构造器)。它的作用是创建和返回一个新的HudView实例。所谓的convenience constructor通常是一个class method（类方法）。类方法作用于整个类，而不是某个特殊的实例变量。通过最前面的+号我们就可以知道它是一个类方法，而不是-开头的实例方法。（convenience constructor的返回类型是instancetype，它是一个特殊的Objective-C关键字）。

当我们调用[HudView hudInView:someView animated:YES]时，我们还没有获得HudView的实例。在这种情况下，[]括号中的第一个单词并非实例变量的名称，或包含一个HudView对象的本地变量名称，而是HudView类自身。这个方法的作用就是创建HUD 视图的一个实例，然后将其放到另一个视图的上面。

在类方法中，有这样的代码：

```
HudView *hudView = [[HudView alloc] initWithFrame:view.bounds];
...
return hudView;
```

这里用到了alloc和init，也就是在iOS中创建一个新对象的必备步骤。在方法的最后我们返回了所创建的新实例。

好吧，这个类方法的代码可以理解，但为什么要用这种convenience constructor的形式呢？为什么不手动创建和初始化实例对象呢？答案很简单，convenience，为了方便起见。因为除了初始化视图外还需要很多步骤，通过将这些内容都放到一个convenience constructor里面，方法的调用者就可以省掉了不少麻烦。

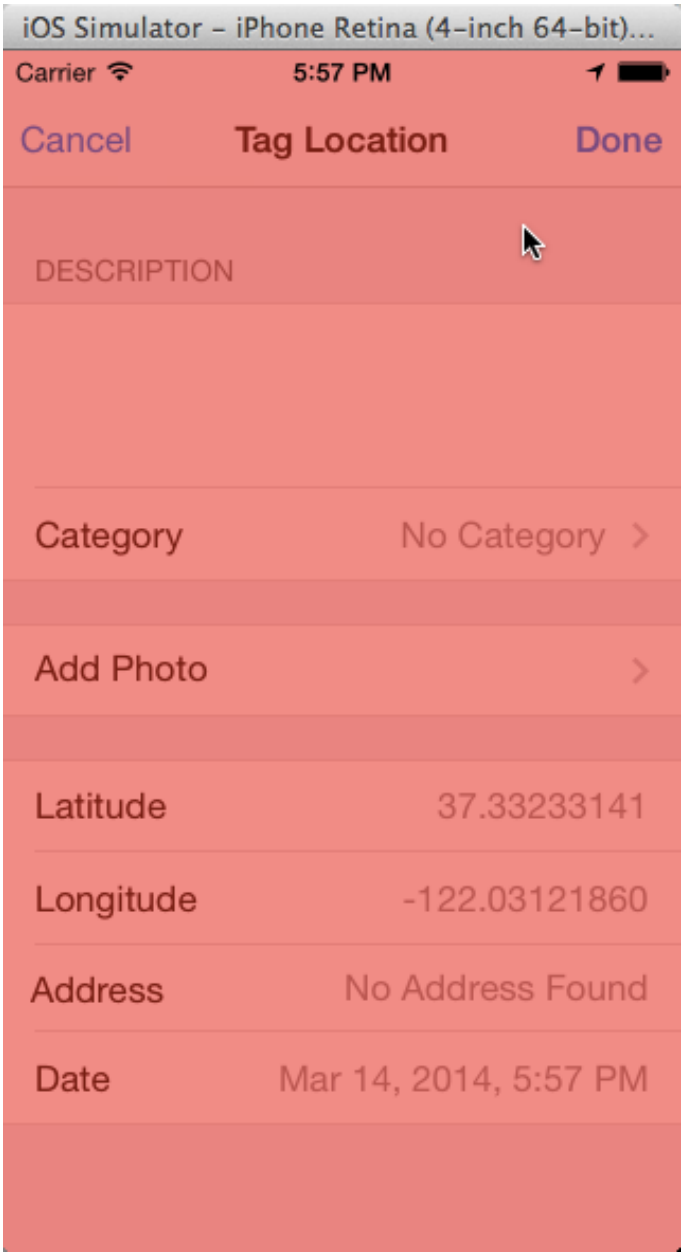
所以说，程序猿真是一个很懒惰的群体~

其中一个步骤就是将HudView对象添加为view对象的子视图。它实际上是导航控制器的视图，因此HUD将覆盖整个界面。

同时我们还将视图的userInteractionEnabled属性设置为NO。当我们显示HUD的时候，通常不希望用户此时和界面进行任何交互。当用户触碰Done按钮的时候，界面就处于等待关闭的过程中。大多数用户都会在此时离开界面，不过总有一些淘气的家伙想尝试搞搞破坏。通过将userInteractionEnabed属性设置为NO，就可以忽略用户对界面的任何互动。

此外，为了测试需要，我们将HUD视图的背景颜色设置为50%透明的红色。这样我们就可以看到它覆盖了整个界面。顺便提一下，任何时候当我们看到某个数字的后面带有f时，比如1.0f，就意味着这个数字是一个浮点数。

编译运行应用。当我们触碰Done按钮的时候，界面效果如下：



这里有个小小的建议，当我们和自定义视图打交道的时候，最好把背景颜色设置为某个亮色，比如红色或蓝色，这样就可以清楚的知道视图大小了。

好了，现在可以从hudInView:animated:方法中删除backgroundColor这一行代码了。

然后在HudView.m中添加以下方法：

```
-(void)drawRect:(CGRect)rect{

    const CGFloat boxWidth = 96.0f;
    const CGFloat boxHeight = 96.0f;

    CGRect boxRect = CGRectMake(
roundf(self.bounds.size.width- boxWidth)/2.0f,
roundf(self.bounds.size.height-boxHeight)/2.0f,
boxWidth,
boxHeight);
    UIBezierPath *roundedRect = [UIBezierPath bezierPathWithRoundedRect:boxRect
cornerRadius:10.0f];
    [[UIColor colorWithWhite:0.3f alpha:0.8f]setFill];
    [roundedRect fill];
}
```

当UIKit需要重新绘制视图的时候就会调用drawRect方法。记住一点，在iOS开发中，所有的事情都是事件驱动的。除非UIKit发送了drawRect事件，我们不会在界面上绘制任何内容。这就意味着我们不应自己调用drawRect方法。

如果我们需要强制重新绘制视图，可以发送setNeedsDisplay消息。然后UIKit将会在需要绘制的时候触发一个drawRect事件。如果你有过其它平台的开发经验，可能觉得这一点很奇怪。在其它平台的开发中，只要你喜欢，可以随时重新绘制界面。不过在iOS中，UIKit掌管着与之相关的一切。

通过上面的代码，可以在界面中间绘制圆角矩形。矩形的大小是96*96points：

```
   const CGFloat boxWidth = 96.0f;
    const CGFloat boxHeight = 96.0f;
```

这里我们又碰到了一个新东西，const关键字。它的意思是说boxWidth和boxHeight这两个属于常量而非变量。什么是常量？当我们给它赋值以后它的数值就保持不变了。通常来说我们使用常量给某个数值一个有意义的名称。比如boxWidth比96.0f看起来更加符合人类的口味。
接着看上面的代码：

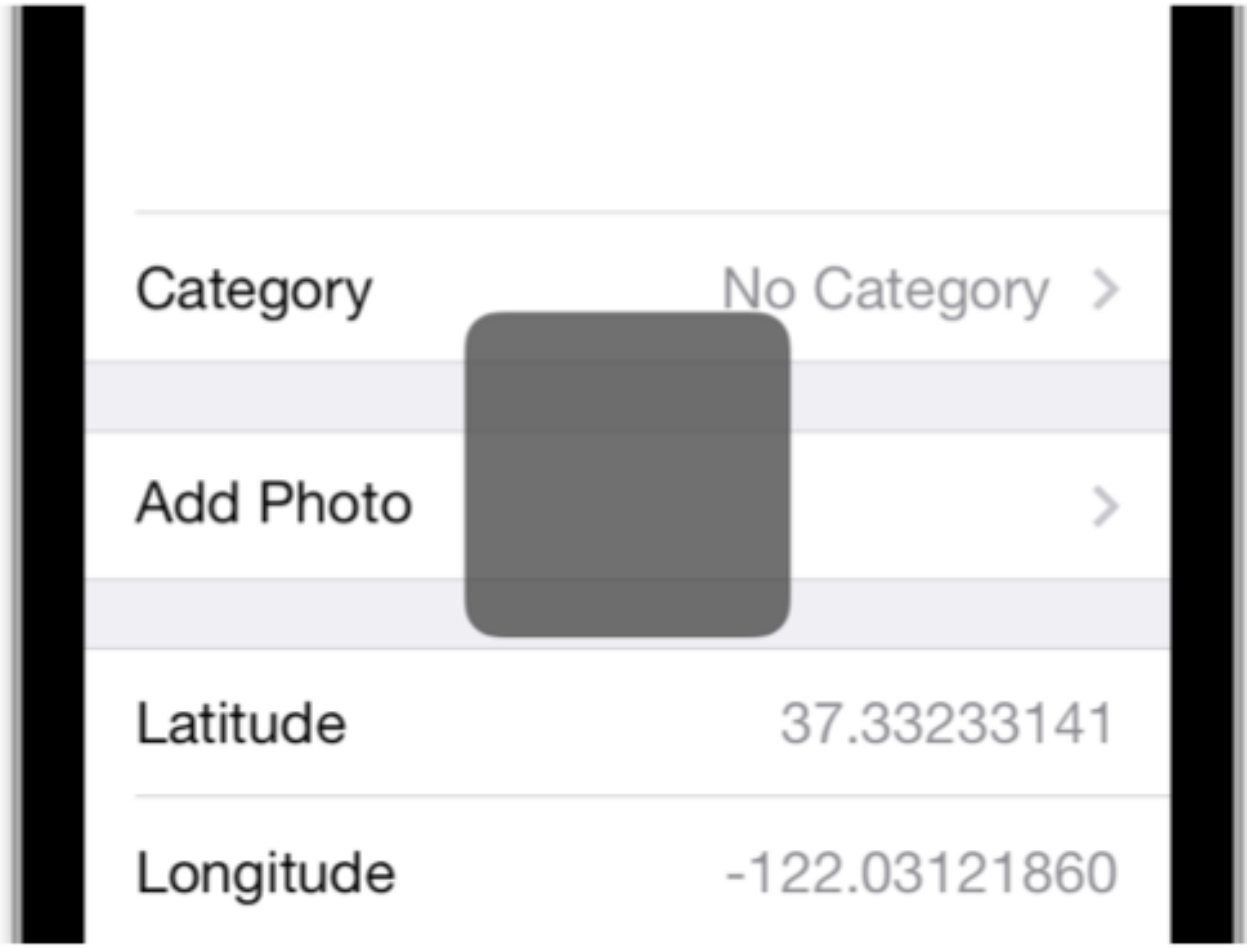
```
CGRect boxRect = CGRectMake(

roundf(self.bounds.size.width - boxWidth) / 2.0f,
roundf(self.bounds.size.height - boxHeight) / 2.0f,
boxWidth,
boxHeight
);
```

这里我们又碰到了一个CGRect，可以使用CGRect来计算将要绘制的矩形的位置。根据设计，这个矩形应该放在界面的正中间。self.bounds.size就是HudView的大小。在上面的 计算中，我们使用roundf()函数来确保矩形的大小是整数，这样就不会让图像看起来很奇怪。

继续看代码：

```
UIBezierPath *roundedRect = [UIBezierPath bezierPathWithRoundedRect:boxRect
cornerRadius:10.0f];
    [[UIColor colorWithWhite:0.3f alpha:0.8f]setFill];
    [roundedRect fill];
```

UIBezierPath是一个非常有用的对象，可以使用它来绘制圆角矩形。我们只需要提供矩形的大小和边缘的角度就好了。然后我们使用80%透明的灰黑色来填充这个矩形，最后的结果如下：



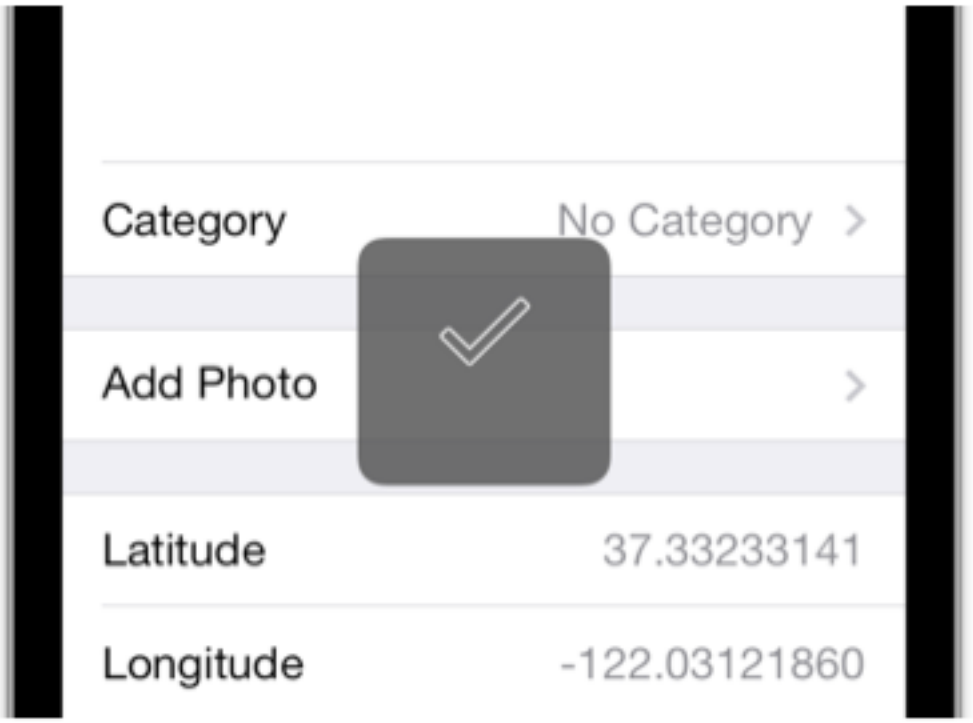
当然，我们还需要在HUD上面添加两个东西，一个勾选标志和一个文本标签。这里的勾选标志是一个图片。

从示例项目文件夹中的Resources文件夹中找到Hud Images文件夹，里面有Checkmark.png和Checkmark@2x.png两张图片。把这些文件添加到asset catalog中，也就是Images.xcassets。我们可以使用+按钮，或者直接从Finder中拖到Xcode里面（因为图片本身是白色的，所以在asset list中可能看不清楚）。

回到HudView.m，在drawRect方法中添加以下代码（黄色高亮部分）：

```
-(void)drawRect:(CGRect)rect{

    const CGFloat boxWidth = 96.0f;
    const CGFloat boxHeight = 96.0f;

    CGRect boxRect = CGRectMake(roundf(self.bounds.size.width- boxWidth)/2.0f,
roundf(self.bounds.size.height-boxHeight)/2.0f, boxWidth, boxHeight);
    UIBezierPath *roundedRect = [UIBezierPath bezierPathWithRoundedRect:boxRect
cornerRadius:10.0f];
    [[UIColor colorWithWhite:0.3f alpha:0.8f]setFill];
    [roundedRect fill];


    UIImage *image = [UIImage imageNamed:@"Checkmark"];

    CGPoint imagePoint = CGPointMake(self.center.x-roundf(image.size.width/2.0f),
self.center.y-roundf(image.size.height/2.0f)-boxHeight/8.0f);
    [image drawAtPoint:imagePoint];
}
```

通过以上代码，就把一个checkmark勾选图片放到UIImage对象中了。接着我们基于HUD 视图的中心坐标（self.center）和图片的大小（image.size)计算出了图片的所在位置。
当然，最后就是在所需要的位置绘制这个图像。



通常情况下，为了在自己的定制化视图中绘制文本，我们会添加一个UILabel对象作为子视图，然后让UILabel完成所有的困难工作。不过对于如此简单的视图，我们完全可以自己来绘制相关的文本内容。

在drawRect:方法中添加以下代码（黄色高亮部分）：

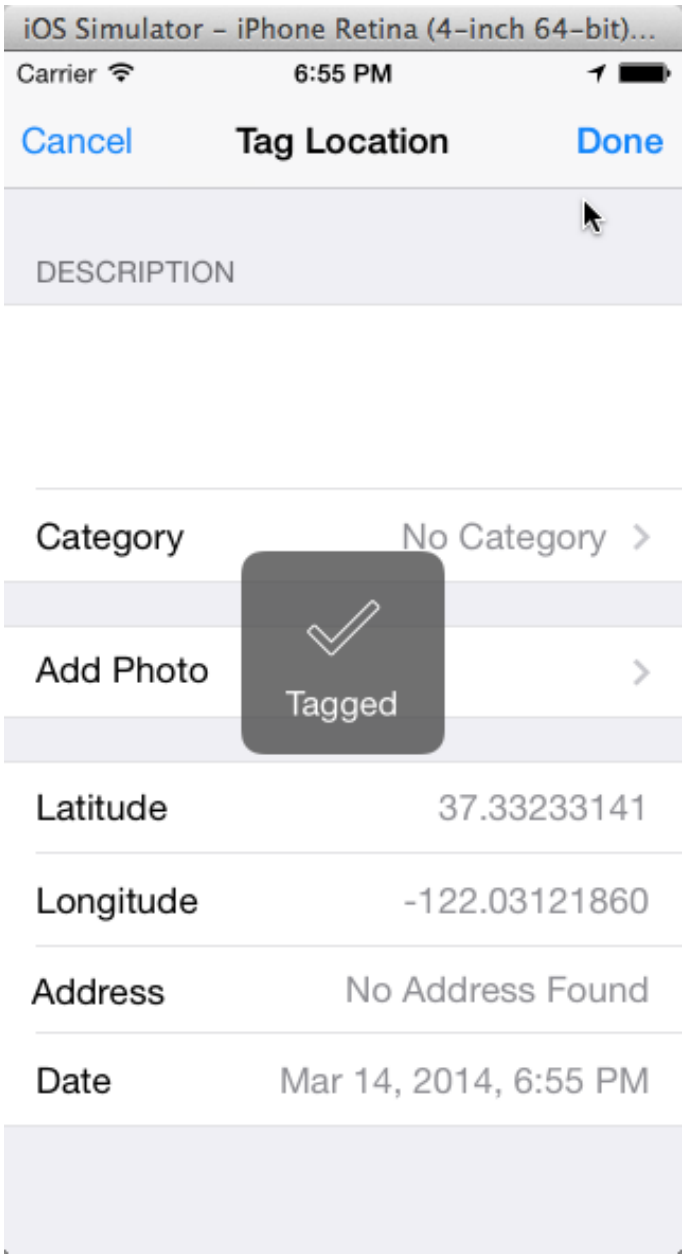
```
-(void)drawRect:(CGRect)rect{

    const CGFloat boxWidth = 96.0f;
    const CGFloat boxHeight = 96.0f;

    CGRect boxRect = CGRectMake(roundf(self.bounds.size.width- boxWidth)/2.0f,
roundf(self.bounds.size.height-boxHeight)/2.0f, boxWidth, boxHeight);
    UIBezierPath *roundedRect = [UIBezierPath bezierPathWithRoundedRect:boxRect
cornerRadius:10.0f];
    [[UIColor colorWithWhite:0.3f alpha:0.8f]setFill];
    [roundedRect fill];


    UIImage *image = [UIImage imageNamed:@"Checkmark"];

    CGPoint imagePoint = CGPointMake(self.center.x-roundf(image.size.width/2.0f),
self.center.y-roundf(image.size.height/2.0f)-boxHeight/8.0f);
    [image drawAtPoint:imagePoint];


    NSDictionary *attributes = @{
        NSFontAttributeName:[UIFont systemFontOfSize:16.0f],
        NSForegroundColorAttributeName:[UIColor whiteColor]
        };

    CGSize textSize = [self.text sizeWithAttributes:attributes];

    CGPoint textPoint = CGPointMake(
        self.center.x -roundf(textSize.width/2.0f),
        self.center.y-roundf(textSize.height/2.0f)+boxHeight/4.0f);

    [self.text drawAtPoint:textPoint withAttributes:attributes];
}
```

当我们需要绘制文本的时候，首先需要知道文本的大小，NSString类提供了一系列的方法来处理这些事情。
首先我们创建了文字所需要使用的UIFont对象。它是一个System的16大小。iOS7的默认系统字体是Helvetica Neue。顺便说一下，在Pages中创建文件的时候里面的默认字体也是这个。
同时我们选择了简单的白色作为文字颜色。

接着我们使用self.text属性的字体和字符串来计算文字的宽度和高度，并保存到一个CGSize变量中。CGSize也是一个结构体。CGPoint,CGSize和CGRect是创建自定义视图的三种常见数据类型。

最后我们计算出要绘制文本的位置，并执行绘制操作。

编译运行应用，看看是怎样的效果。



好了，现在看起来有点那个意思了，不过还不够好。现在我们来添加一个简单的小动画效果。

在HudView.m中添加一个showAnimated:方法如下：

```
-(void)showAnimated:(BOOL)animated{

    if(animated){

        //1
        self.alpha = 0.0f;
        self.transform = CGAffineTransformMakeScale(1.3f, 1.3f);

        //2
        [UIView animateWithDuration:0.3 animations:^{

            //3
            self.alpha = 1.0f;
```

```
            self.transform = CGAffineTransformIdentity;
        }];
    }


}
```

在系列1的教程中，我们曾使用Core Animation框架来创建一个crossfade的动画效果。不过UIView也有自己的动画机制，它在背后仍然使用了Core Animation，只是对于开发者来说更加方便。

创建UIView的动画效果需要按照以下步骤进行：

1.设置动画开始前的视图初始状态。这里我们将alpha透明度设置为0，也就是说视图是完全透明的。同时我们还将transform的scale因子设置为1.3。这里我们不打算深入讨论transform的内部机制，你只需要知道它代表视图初始情况下是拉伸状态的。

2.调用[UIView animationWithDuration:...]方法

3.在块语句中，设置视图在动画完成后的新状态。这里我们将alpha设置为1.0，意味着完全可见。同时我们将transform设置为identifty，意味着scale大小会恢复到正常。

最终生成的动画就是在duration的时间里从初始状态过渡到最终状态。这里HUD 视图将从完全透明过渡到完全可见，同时从原始大小的1.3倍恢复到正常大小。

这个动画虽然很简单，不过实际效果还是让人很赏心悦目的。

接下来更改hudInView的convenience constructor，在返回实例对象前调用showAnimated:方法：

```
+(instancetype)hudInView:(UIView *)view animated:(BOOL)animated{

    HudView *hudView = [[HudView alloc]initWithFrame:view.bounds];
    hudView.opaque = NO;

    [view addSubview:hudView];
    view.userInteractionEnabled = NO;

    [hudView showAnimated:animated];
    return hudView;

}
```

编译运行应用，看看现在的效果吧。

好吧，这个小小的提示已经可以完美工作了，不过我们希望当用户触碰Done按钮的时候仍然可以关闭界面。

在Xcode中切换到LocationDetailsViewController.m，然后更改done:动作方法的代码如下：

```
-(IBAction)done:(id)sender{
```

```
    HudView *hudView = [HudView hudInView:self.navigationController.view animated:YES];
    hudView.text = @"Tagged";

    [self performSelector:@selector(closeScreen) withObject:nil afterDelay:0.6];
}
```

oops,这是个神马东东？为什么我们不在显示HUD视图后直接调用[self closeScreen]呢？

当然你可以这样做，只不过最后的效果不是很好看，因为在HUD显示完动画前界面就已经被关闭了。通过调用performSelector:withObject:afterDelay:方法，我们可以让closeScreen方法在0.6秒后再被调用。

0.6秒是肿么算出来的？好吧，因为HUD 视图需要0.3秒的时间来完成自己的动画，而在界面关闭前我们需要再等上0.3秒。我们不希望让HUD没有用武之地，也不希望等待太久被用户痛骂。动画是为了改善用户体验，而不是破坏用户体验的！

好了，现在编译运行应用。触碰Done按钮，观察界面是如何消失的。恩，看起来相当不错，至少我自己是这么认为的，哈哈~

好了，今天的学习就到此结束吧。

提醒一下大家，下一课开始我们又要进行一些理论知识的充电了~