交互式动画

黄宣冬 (http://morisunshine.com) 2014/05/18

在2007年，乔布斯在第一次介绍 iPhone 的时候，iPhone 的触摸屏交互简直就像是一种魔法。最好的例子就是在他第一次滑动 TableView 的展示上 (https://www.youtube.com/watch?v=t4OEsI0Sc_s&t=16m9s)。你可以感受到当时观众的反应是多么惊讶，但是对于现在的我们来说早已习以为常。在展示的后面一部分，他特别指出当他给别人看了这个滑动例子，别人说的一句话: “当这个界面滑动的时候我就已经被征服了” (https://www.youtube.com/watch?v=t4OEsI0Sc_s&t=16m9s).

是什么样的滑动能让人有‘哇哦’的效果呢?

滑动是最完美地展示了通过触摸屏直接操作的例子。滚动视图遵从于你的手指，当你的手指离开屏幕的时，视图会自然地继续滑动直到该停止的时候停止。它用自然的方式减速，甚至在快到界限的时候也能表现出细腻的弹力效果。滑动在任何时候都保持相应，并且看上去非常真实。

动画的状态

在 iOS 中的大部分动画仍然没有按照最初 iPhone 指定的滑动标准实现。这里有很多动画一旦它们运行就不能交互（比如说解锁动画，主界面中打开文件夹和关闭文件夹的动画，和导航栏切换的动画，还有很多）。

然而现在有一些应用给我一种始终在控制动画的体验，我们可以直接操作那些我在用的动画。当我们将这些应用和其他的应用相比较之后，我们就能感觉到明显的区别。这些应用中最优秀的有最初的 Twitter iPad app，和现在的 Facebook Paper。但目前，使用直接操作为主并且可以中断动画的应用仍然很少。这就给我们做出更好的应用提供了机会，让我们的应用有更不同的，更高质量的体验。

真实交互式动画的挑战

当我们用 UIView 或者 CAAnimation 来实现交互式动画时会有两个大问题: 这些动画会将你在屏幕上的内容和 layer 上的实际的空间属性分离开来，并且他们直接操作这些空间属性。

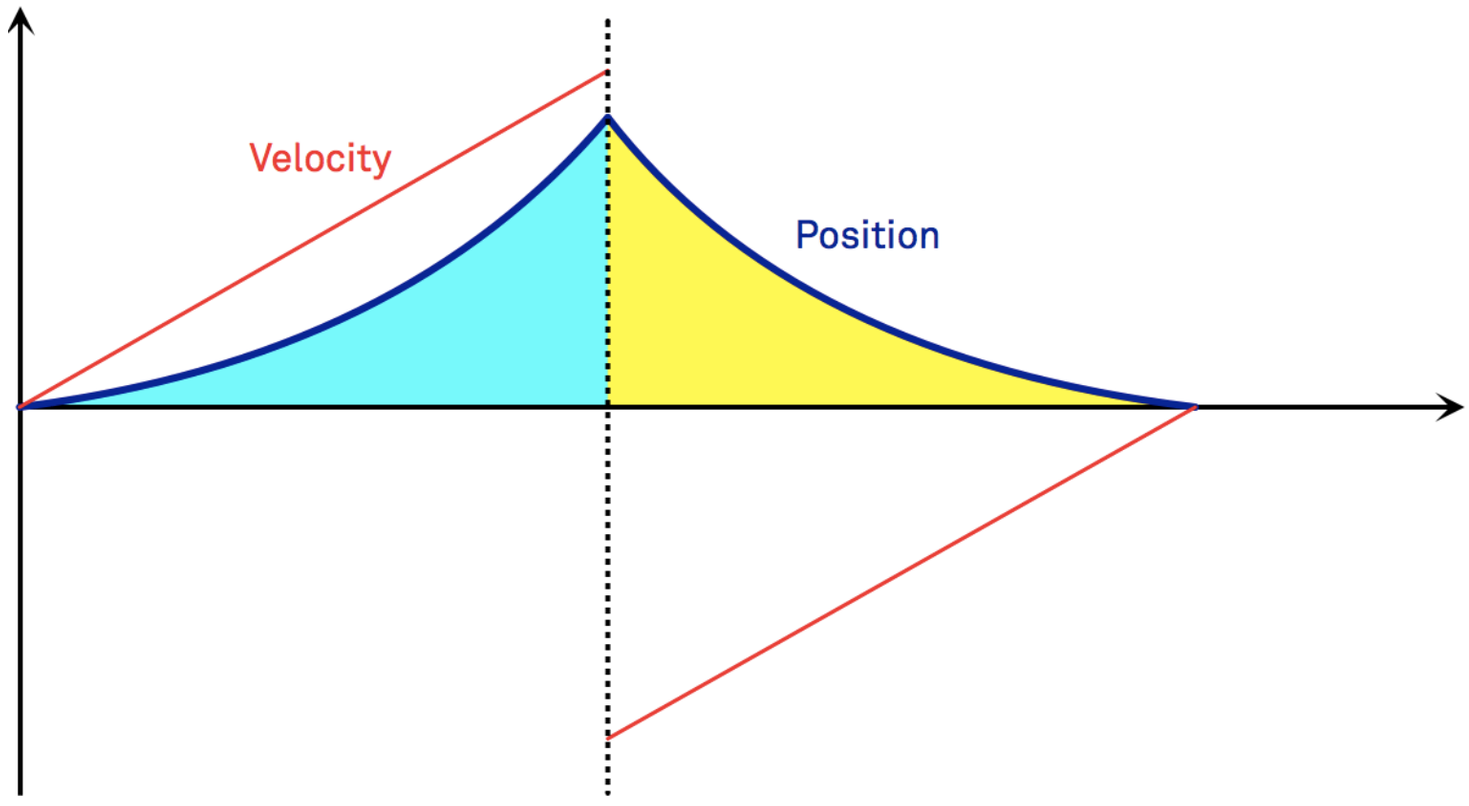
模型 (Model) 和显示 (Presentation) 的分离

Core Animation 是通过分离 layer 的模型属性和你在屏幕上看到的界面 (显示层) 的方式来设计的，这就导致我们很难去创建一个可以在任何时候能交互的动画，因为在动画时，模型和界面已经不能匹配了。这时，我们不得不通过手动的方式来同步这两个的状态，来达到改变动画的效果：

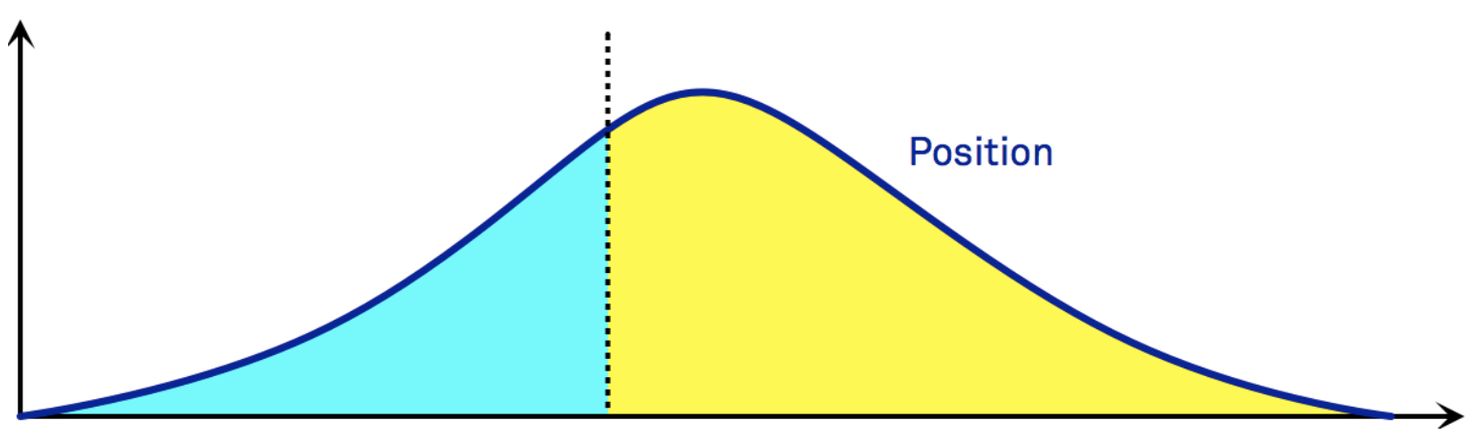
```
view.layer.center = view.layer.presentationLayer.center;
[view.layer removeAnimationForKey:@"animation"];
// 添加新动画
```

直接控制 vs 间接控制

`CAAnimation` 动画的更大的问题是它们是直接在 layer 上对属性进行操作的。这意味着什么呢? 比如我们想指定一个 layer 从坐标为 (100, 100) 的位置运动到 (300, 300) 的位置，但是在它运动到中间的时候，我们想它停下来并且让它回到它原来的位置，事情就变得非常复杂了。如果你只是简单地删除当前的动画然后再添加一个新的，那么这个 layer 的速率就会不连续。



然而，我们想要的是一个漂亮的，流畅地减速和加速的动画。



只有通过*间接*操作动画才能达到上面的效果，比如通过模拟力在界面上的表现。新的动画需要用 layer 的当前*速度矢量*作为参数传入来达到流畅的效果。

看一下 UIView 中关于弹簧动画的 API

```
(animateWithDuration:delay:usingSpringWithDamping:initialSpringVelocity:options:animations:completion:
```

你会注意到速率是个 `CGFloat` 。所以当我们给一个移动 view 的动画在其运动的方向上加一个初始的速率时，你没法告知动画这个 view 现在的运动状态，比如我们不知道要添加的动画的方向是不是和原来的 view 的速度方向垂直。为了使其成为可能，这个速度需要用向量来表示。

解决方案

让我们看一下我们怎样来正确实现一个可交互并且可以中断的动画。我们来做一个类似于控制中心板的东西来实现这个效果：

这个控制板有两个状态：打开和关闭。你可以通过点击来切换这两个状态，或者通过上下拖动来调调整它向上或向下。我要将这个控制面板的所有状态都做到可以交互，甚至是在动画的过程中也可以，这是一个很大的挑战。比如，当你在这个控制板还没有切换到打开状态的动画过程中，你点击了它，那么它应该从现在这个点的位置马上回到关闭状态的位置。在现在很多的应用中，大部分都是用默认的动画 API，你必须要等一个动画结束之后你才能做自己想做的事情。或者，如果你不等待的话，就会看到一个不连续的速度曲线。我们要解决这个问题。

UIKit 力学

随着 iOS7 的发布，苹果向我们展示了一个叫 UIKit 力学的动画框架 (可以参见 WWDC 2013 sessions 206 (https://developer.apple.com/videos/wwdc/2013/index.php?id=206) 和 221 (https://developer.apple.com/videos/wwdc/2013/index.php?id=221))。UIKit 力学是一个基于模拟物理引擎的框架，只要你添加指定的行为到动画对象上来实现 UIDynamicItem (https://developer.apple.com/library/ios/documentation/uikit/reference/UIDynamicItem_Protocol/Reference/Reference.htr 协议就能实现很多动画。这个框架非常强大，并且它能够在多个物体间启用像是附着和碰撞这样的复杂行为。请看一下 UIKit Dynamics Catalog (https://developer.apple.com/library/ios/samplecode/DynamicsCatalog/Introduction/Intro.html)，确认一下什么是可用的。

因为 UIKit 力学中的的动画是被间接驱动的，就像我在上面提到的，这使我们实现真实的交互式动画成为可能，它能在任何时候被中断并且拥有连续的加速度。同时，UIKit 力学在物理层的抽象上能完全胜任我们一般情况下在用户界面中的所需要的所有动画。其实在大部分情况下，我们只会用到其中的一小部分功能。

定义行为

为了实现我们的控制板的行为，我们将使用 UIkit 力学中的两个不同行为：UIAttachmentBehavior (https://developer.apple.com/library/ios/documentation/uikit/reference/UIAttachmentBehavior_Class/Reference/Referenc 和 UIDynamicItemBehavior (https://developer.apple.com/library/ios/documentation/uikit/reference/UIDynamicItemBehavior_Class/Reference/Referen 附着行为用来扮演弹簧的角色，它将界面向目标点拉动。另一方面，我们用动态 item behvaior 定义了比如摩擦系数这样的界面的内置属性。

我创建了一个我们自己的行为子类，以将这两个行为封装到我们的控制板上:

```
@interface PaneBehavior : UIDynamicBehavior

@property (nonatomic) CGPoint targetPoint;
@property (nonatomic) CGPoint velocity;

- (instancetype)initWithItem:(id <UIDynamicItem>)item;

@end
```

我们通过一个 dynamic item 来初始化这个行为，然后就可以设置它的目标点和我们想要的任何速度。在内部，我们创建了附着行为和 dynamic item 行为，并且将这些行为添加到我们自定义的行为中:

```
- (void)setup
{
    UIAttachmentBehavior *attachmentBehavior = [[UIAttachmentBehavior alloc] initWithItem:self.item at
tachedToAnchor:CGPointZero];
    attachmentBehavior.frequency = 3.5;
    attachmentBehavior.damping = .4;
    attachmentBehavior.length = 0;
    [self addChildBehavior:attachmentBehavior];
    self.attachmentBehavior = attachmentBehavior;

    UIDynamicItemBehavior *itemBehavior = [[UIDynamicItemBehavior alloc] initWithItems:@[self.item]];
    itemBehavior.density = 100;
    itemBehavior.resistance = 10;
    [self addChildBehavior:itemBehavior];
    self.itemBehavior = itemBehavior;
}
```

为了用 `targetPoint` 和 `velocity` 属性来影响 item 的 behavior，我们需要重写它们的 setter 方法，并且分别修改在附着行为和 item behaviors 中的对应的属性。我们对目标点的 setter 方法来说，这个改动很简单：

```
- (void)setTargetPoint:(CGPoint)targetPoint
{
    _targetPoint = targetPoint;
    self.attachmentBehavior.anchorPoint = targetPoint;
}
```

对于 velocity 属性，我们需要多做一些工作，因为 dynamic item behavior 只允许相对地改变速度。这就意味如果我们要将 velocity 设置为绝对值，首先我们就需要得到当前的速度，然后再加上速度差才能得到我们的目标速度。

```
- (void)setVelocity:(CGPoint)velocity
{
    _velocity = velocity;
    CGPoint currentVelocity = [self.itemBehavior linearVelocityForItem:self.item];
    CGPoint velocityDelta = CGPointMake(velocity.x - currentVelocity.x, velocity.y - currentVelocity.y
);
    [self.itemBehavior addLinearVelocity:velocityDelta forItem:self.item];
}
```

将Behavior投入使用

我们的控制板有三个不同状态：在开始或结束位置的静止状态，正在被用户拖动的状态，以及在没有用户控制时运动到结束位置的动画状态。

为了将从直接操作状态 (用户拖动这个滑动板) 过渡到动画状态这个过程做的流畅，我们还有很多其他的事要做。当用户停止拖动控制板时，它会发送一个消息到它的 delegate。根据这个方法，我们可以知道这个板应该朝哪个方向运动，然后在我们自定义的 PaneBehavior 上设置结束点，以及初始速度 (这非常重要)，并将行为添加到动画器中去，以此确保从拖动操作到动画状态这个过程能够非常流畅。

```
- (void)draggableView:(DraggableView *)view draggingEndedWithVelocity:(CGPoint)velocity
{
    PaneState targetState = velocity.y >= 0 ? PaneStateClosed : PaneStateOpen;
    [self animatePaneToState:targetState initialVelocity:velocity];
}

- (void)animatePaneToState:(PaneState)targetState initialVelocity:(CGPoint)velocity
{
    if (!self.paneBehavior) {
        PaneBehavior *behavior = [[PaneBehavior alloc] initWithItem:self.pane];
        self.paneBehavior = behavior;
    }
    self.paneBehavior.targetPoint = [self targetPointForState:targetState];
    if (!CGPointEqualToPoint(velocity, CGPointZero)) {
        self.paneBehavior.velocity = velocity;
    }
    [self.animator addBehavior:self.paneBehavior];
    self.paneState = targetState;
}
```

一旦用户用他的手指再次触动控制板时，我必须要将所有的 dynamic behavior 从 animator 删除，这样才不会影响控制板对拖动手势的响应：

```
- (void)draggableViewBeganDragging:(DraggableView *)view
{
    [self.animator removeAllBehaviors];
}
```

我们不仅仅允许控制板可以被拖动，还要允许它可以被点击，让它可以从一个位置跳转到另一个位置以达到开关的效果。一旦点击事件发生，我们就会立即调整这个滑动板的目标位置。因为我们不能直接控制动画，但是通过弹力和摩擦力，我们的动画可以非常流畅地执行这个动作：

```
- (void)didTap:(UITapGestureRecognizer *)tapRecognizer
{
    PaneState targetState = self.paneState == PaneStateOpen ? PaneStateClosed : PaneStateOpen;
    [self animatePaneToState:targetState initialVelocity:CGPointZero];
}
```

这样就实现了我们的大部分功能了。你可以在 GitHub (https://github.com/objcio/issue-12-interactive-animations-uidynamics) 上查看完整的例子。

重申一点：UIKit 力学可以通过在界面上模拟力来间接地驱动动画（我们的例子中，使用的是弹力和摩擦力）。这间接地使我们在任何时候都能以连续的速度曲线来与界面进行交互。

现在我们已经通过 UIKit 力学实现了整个交互，让我们回顾一下这个场景。这个例子的动画中我们只用了 UIKit 力学中一小部分功能，并且它的实现方式也非常简单。对于我们来说这是一个去理解它其中的过程的很好的例子，但是如果我们使用的环境中没有 UIKit 力学 (比如说在 Mac 上)，或者你的使用场景中不能很好的适用 UIKit 力学呢。

自己操作动画

至于在你的应用中大部分时间会用的动画，比如简单的弹力动画，我们控制它真的不难。我们可以做一个练习，来看看如何抛弃 UIKit 力学这个巨大的黑盒子，看要如何“手动”来实现一个简单的交互。想法非常简单：我们只要每秒修改这个 view 的 frame 60 次。每一帧我们都基于当前速度和作用在 view 上的力来调整 view 的 frame 就可以了。

物理原理

首先让我们看一下我们需要知道的基础物理知识，这样我们才能实现出刚才使用 UIKit 力学实现的那种弹簧动画效果。为了简化问题，虽然引入第二个维度也是很直接的，但我们在这里只关注一维的情况 (在我们的例子中就是这样的情况)。

我们的目标是依据控制面板当前的位置和上一次动画后到现在为止经过的时间，来计算它的新位置。我们可以把它表示成这样：

```
y = y0 + Δy
```

位置的偏移量可以通过速度和时间的函数来表达：

```
Δy = v · Δt
```

这个速度可以通过前一次的速度加上速度偏移量算出来，这个速度偏移量是由力在 view 上的作用引起的。

```
v = v0 + Δv
```

速度的变化可以通过作用在这个 view 上的冲量计算出来：

```
Δv = (F · Δt) / m
```

现在，让我们看一下作用在这个界面上的力。为了得到弹簧效果，我们必须要将摩擦力和弹力结合起来：

```
F = F_spring + F_friction
```

弹力的计算方法我们可以从任何一本教科书中得到 (编者注：简单的胡克定律)：

```
F_spring = k · x
```

k 是弹簧的劲度系数， x 是 view 到目标结束位置的距离 (也就是弹簧的长度)。因此，我们可以把它写成这样：

```
F_spring = k · abs(y_target − y0)
```

摩擦力和 view 的速度成正比：

```
F_friction = μ · v
```

μ 是一个简单的摩擦系数。你可以通过别的方式来计算摩擦力，但是这个方法能很好地做出我们想要的动画效果。

将上面的表达式放在一起，我们就可以算出作用在界面上的力：

```
F = k · abs(y_target − y0) + μ · v
```

为了实现起来更简单点些，我们将 view 的质量设为 1，这样我们就能计算在位置上的变化：

```
Δy = (v0 + (k · abs(y_target − y0) + μ · v) · Δt) · Δt
```

实现动画

为了实现这个动画，我们首先需要创建我们自己的 `Animator` 类，它将扮演驱动动画的角色。这个类使用了 `CADisplayLink`， `CADisplayLink` 是专门用来将绘图与屏幕刷新频率相同步的定时器。换句话说，如果你的动画是流畅的，这个定时器就会每秒调用你的方法60次。接下来，我们需要实现 `Animation` 协议来和我们的 `Animator` 一起工作。这个协议只有一个方法， `animationTick:finished:` 。屏幕每次被刷新时都会调用这个方法，并且在方法中会得到两个参数：第一个参数是前一个 frame 的持续时间，第二个参数是一个指向 `BOOL` 的指针。当我们设置这个指针的值为 `YES` 时，我们就可以与 `Animator` 取得通讯并汇报动画完成；

```
@protocol Animation <NSObject>
- (void)animationTick:(CFTimeInterval)dt finished:(BOOL *)finished;
@end
```

我们会在下面实现这个方法。首先，根据时间间隔我们来计算由弹力和摩擦力的合力。然后根据这个力来更新速度，并调整 view 的中心位置。最后，当这个速度降低并且 view 到达结束位置时，我们就停止这个动画：

```
- (void)animationTick:(CFTimeInterval)dt finished:(BOOL *)finished
{
    static const float frictionConstant = 20;
    static const float springConstant = 300;
    CGFloat time = (CGFloat) dt;

    //摩擦力 = 速度 * 摩擦系数
    CGPoint frictionForce = CGPointMultiply(self.velocity, frictionConstant);
    //弹力 = (目标位置 - 当前位置) * 弹簧劲度系数
    CGPoint springForce = CGPointMultiply(CGPointSubtract(self.targetPoint, self.view.center), springC
onstant);
    //力 = 弹力 - 摩擦力
    CGPoint force = CGPointSubtract(springForce, frictionForce);

    //速度 = 当前速度 + 力 * 时间 / 质量
    self.velocity = CGPointAdd(self.velocity, CGPointMultiply(force, time));
    //位置 = 当前位置 + 速度 * 时间
    self.view.center = CGPointAdd(self.view.center, CGPointMultiply(self.velocity, time));

    CGFloat speed = CGPointLength(self.velocity);
    CGFloat distanceToGoal = CGPointLength(CGPointSubtract(self.targetPoint, self.view.center));
    if (speed < 0.05 && distanceToGoal < 1) {
        self.view.center = self.targetPoint;
        *finished = YES;
    }
}
```

这就是这个方法里的全部内容。我们把这个方法封装到一个 `SpringAnimation` 对象中。除了这个方法之外，这个对象中还有一个初始化方法，它指定了 view 中心的目标位置 (在我们的例子中，就是打开状态时界面的中心位置，或者关闭状态时界面的中心位置) 和初始的速度。

将动画添加到 view 上

我们的 view 类刚好和使用 UIDynamic 的例子一样：它有一个拖动手势，并且根据拖动手势来更新中心位置。它也有两个同样的 delegate 方法，这两个方法会实现动画的初始化。首先，一旦用户开始拖动控制板时，我们就取消所有动画：

```
- (void)draggableViewBeganDragging:(DraggableView *)view
{
    [self cancelSpringAnimation];
}
```

一旦停止拖动，我们就根据从拖动手势中得到的最后一个速率值来开始我们的动画。我们根据拖动状态 `paneState` 计算出动画的结束位置：

```
- (void)draggableView:(DraggableView *)view draggingEndedWithVelocity:(CGPoint)velocity
{
    PaneState targetState = velocity.y >= 0 ? PaneStateClosed : PaneStateOpen;
    self.paneState = targetState;
    [self startAnimatingView:view initialVelocity:velocity];
}

- (void)startAnimatingView:(DraggableView *)view initialVelocity:(CGPoint)velocity
{
    [self cancelSpringAnimation];
    self.springAnimation = [UINTSpringAnimation animationWithView:view target:self.targetPoint velocit
y:velocity];
    [view.animator addAnimation:self.springAnimation];
}
```

剩下来要做的就是添加点击动画了，这很简单。一旦我们触发这个状态，就开始动画。如果这里正在进行弹簧动画，我们就用当时的速度作为开始。如果这个弹簧动画是 nil，那么这个开始速度就是 CGPointZero。想要知道为什么依然可以进行动画，可以看看 `animationTick:finished:` 里的代码。当这个起始速度为 0 的时候，弹力就会使速度缓慢地增长，直到面板到达目标位置：

```
- (void)didTap:(UITapGestureRecognizer *)tapRecognizer
{
    PaneState targetState = self.paneState == PaneStateOpen ? PaneStateClosed : PaneStateOpen;
    self.paneState = targetState;
    [self startAnimatingView:self.pane initialVelocity:self.springAnimation.velocity];
}
```

动画驱动

最后，我们需要一个 `Animator`，也就是动画的驱动者。Animator 封装了 display link。因为每个 display link 都链接一个指定的 `UIScreen`，所以我们根据这个指定的 UIScreen 来初始化我们的 animator。我们初始化一个 display link，并且将它加入到 run loop 中。因为现在还没有动画，所以我们是从暂停状态开始的：

```
- (instancetype)initWithScreen:(UIScreen *)screen
{
    self = [super init];
    if (self) {
        self.displayLink = [screen displayLinkWithTarget:self selector:@selector(animationTick:)];
        self.displayLink.paused = YES;
        [self.displayLink addToRunLoop:[NSRunLoop mainRunLoop] forMode:NSRunLoopCommonModes];
        self.animations = [NSMutableSet new];
    }
    return self;
}
```

一旦我们添加了这个动画，我们要确保这个 display link 不再是停止状态：

```
- (void)addAnimation:(id<Animation>)animation
{
    [self.animations addObject:animation];
    if (self.animations.count == 1) {
        self.displayLink.paused = NO;
    }
}
```

我们设置这个 display link 来调用 `animationTick:` 方法，在每个 Tick 中，我们都遍历它的动画数组，并且给这些动画数组中的每个动画发送一个消息。如果这个动画数组中已经没有动画了，我们就暂停这个 display link。

```
- (void)animationTick:(CADisplayLink *)displayLink
{
    CFTimeInterval dt = displayLink.duration;
    for (id<Animation> a in [self.animations copy]) {
        BOOL finished = NO;
        [a animationTick:dt finished:&finished];
        if (finished) {
            [self.animations removeObject:a];
        }
    }
    if (self.animations.count == 0) {
        self.displayLink.paused = YES;
    }
}
```

完整的项目在 GitHub (https://github.com/objcio/issue-12-interactive-animations) 上。

权衡

我们必须记住，通过 display link 来驱动动画 (就像我们刚才演示的例子，或者我们使用UIkit力学来做的例子，又或者是使用 Facebook 的 Pop 框架) 是有代价需要进行权衡的。就像 Andy Matuschar 指出的 (https://twitter.com/andy_matuschak/status/464790108072206337)那样，UIView 和 CAAnimation 动画比其他任务更少受系统的影响，因为比起你的应用来说，渲染处于更高的优先级。

回到 Mac

现在 Mac 中还没有 UIKit 力学。如果你想在 Mac 中创建一个真实的交互式动画，你必须自己去实现这些动画。我们已经向你展示了如何在 iOS 中实现这些动画，所以在 OS X 中实现相似的功能也是非常简单的。你可以查看在 GitHub 中的完整项目 (https://github.com/objcio/issue-12-interactive-animations-osx)，如果你想要应用到 OS X 中，这里还有一些地方需要修改：

- 第一个要修改的就是 `Animator`。在Mac中没有 `CADisplayLink`，但是取而代之的有 `CVDisplayLink`，它是以 C 语言为基础的 API。创建它需要做更多的工作，但也是很直接。
- iOS 中的弹簧动画是基于调整 view 的中心位置来实现的。而 OS X 中的 `NSView` 类没有 center 这个属性，所以我们用为 frame 中的 origin 做动画来代替。
- 在 Mac 中是没有手势识别，所以我要在我们自定义的 view 子类中实现 `mouseDown:`， `mouseUp:` 和 `mouseDragged:` 方法。

上面就是我们需要在 Mac 中使用我们的动画效果在代码所需要做的修改。对于像这样的简单 view，它能很好的胜任。但对于更复杂的动画，你可能就不会想通过为 frame 做动画来实现了，我们可以用 `transform` 来代替，浏览 Jonathan Willing 写的关于 OS X 动画 (http://jwilling.com/osx-animations)的博客，你会获益良多。

Facebook 的 POP 框架

上个星期围绕着 Facebook 的 POP 框架 (https://github.com/facebook/pop)的讨论络绎不绝。POP 框架是 Paper 应用背后支持的动画引擎。它的操作非常像我们上面讲的驱动动画的例子，但是它以非常灵活的方式巧妙地封装到了一个程序包中。

让我们动手用 POP 来驱动我们的动画吧。因为我们自己的类中已经封装了弹簧动画，这些改变就非常简单了。我们所要做的就是初始化一个 POP 动画来代替我们刚才自己做的动画，并将下面这段代码加入到 view 中：

```
- (void)animatePaneWithInitialVelocity:(CGPoint)initialVelocity
{
    [self.pane pop_removeAllAnimations];
    POPSpringAnimation *animation = [POPSpringAnimation animationWithPropertyNamed:kPOPViewCenter];
    animation.velocity = [NSValue valueWithCGPoint:initialVelocity];
    animation.toValue = [NSValue valueWithCGPoint:self.targetPoint];
    animation.springSpeed = 15;
    animation.springBounciness = 6;
    [self.pane pop_addAnimation:animation forKey:@"animation"];
    self.animation = animation;
}
```

你可以在 GitHub (https://github.com/objcio/issue-12-interactive-animations-pop) 中找到使用 POP 框架的完整例子。

让其工作非常简单，并且通过它我们可以实现很多更复杂的动画。但是它真正强大的地方在于它能够实现真正的可交互和可中断的动画，就像我们上面提到的那样，因为它直接支持以速度作为输入参数。如果你打算从一开始到被中断这过程中的任何时候都能交互，像 POP 这样的框架就能帮你实现这些动画，并且它能始终保证动画一直很平滑。

如果你不满足于用 `POPSpringAnimation` 和 `POPDecayAnimation` 的开箱即用的处理方式的话，POP 还提供了 `POPCustomAnimation` 类，它基本上是一个 display link 的方便的转换，来在动画的每一个 tick 的回调 block 中驱动你自己的动画。

展望未来

随着 iOS7 中从对拟物化的视觉效果的远离，以及对 UI 行为的关注，真实的交互式动画通向未来的大道变得越来越明显。它们也是将初代 iPhone 中滑动行为的魔力延续到交互的各个方面的一条康庄大道。为了让这些魔力成为现实，我们就不能在开发过程中才想到这些动画，而是应该在设计时就要考虑这些交互，这一点非常重要。

非常感谢 Loren Brichter (https://twitter.com/lorenb) 给这篇文章提出的一些意见。

原文 Interactive Animations (http://www.objc.io/issue-12/interactive-animations.html)

译者简介

黄宣冬 (http://morisunshine.com)

从事 iOS 应用开发的程序猿一枚，江湖自称治愈系萌汉子，优美代码追求者，正在人生漫漫路中缓缓前行。