-----------Layer----------------------------------------------------------------------

绘图父类：CALayer（transaction提及）

两个重要属性：

@property CGPoint position;

用来设置CALayer在父层中的位置，以父层的左上角为原点(0, 0)

@property CGPoint anchorPoint;

称为“定位点”、“锚点”，决定着CALayer身上的哪个点会在position属性所指的位置

以自己的左上角为原点(0, 0)，它的x、y取值范围都是0~1，默认值为（0.5, 0.5）

其他属性： —— Tab6 — 第二屏

|  |  |
| --- | --- |
| backgroundColor | 图层背景颜色 |
| borderColor | 边框颜色 |
| borderWidth | 边框宽度 |
| bounds | 图层大小 |
| contents | 图层显示内容，例如可以将图片作为图层内容显示 |
| contentsRect | 图层显示内容的大小和位置 |
| cornerRadius | 圆角半径 |
| doubleSided | 图层背面是否显示，默认为YES |
| frame | 图层大小和位置，不支持隐式动画，所以CALayer中很少使用frame，通常使用bounds和position代替 |
| hidden | 是否隐藏 |
| mask | 图层蒙版 |
| maskToBounds | 子图层是否剪切图层边界，默认为NO |
| opacity | 透明度 ，类似于UIView的alpha |
| position | 图层中心点位置，类似于UIView的center |
| shadowColor | 阴影颜色 |
| shadowOffset | 阴影偏移量 |
| shadowOpacity | 阴影透明度，注意默认为0，如果设置阴影必须设置此属性 |
| shadowPath | 阴影的形状 |
| shadowRadius | 阴影模糊半径 |
| sublayers | 子图层 |
| sublayerTransform | 子图层形变 |
| transform | 图层形变 |

示范代码：

[super viewDidLoad];

//创建图层

CALayer \*layer=[CALayer layer];

//设置图层的属性

layer.backgroundColor=[UIColor redColor].CGColor;

layer.bounds=CGRectMake(0, 0, 100, 100);

//设置锚点为（0，0）

layer.anchorPoint=CGPointZero;

//添加图层

[self.view.layer addSublayer:layer];

隐式动画

每一个UIView内部都默认关联着一个CALayer，我们可用称这个Layer为Root Layer（根层），所有非Root Layer，也就是手动创建的CALayer对象，都存在着隐式动画（当对非Root Layer的部分属性进行修改 时，默认会自动产生一些动画效果，而这些属性称为Animatable Properties(可动画属性)

列举几个常见的Animatable Properties：

bounds：用于设置CALayer的宽度和高度。修改这个属性会产生缩放动画

backgroundColor：用于设置CALayer的背景色。修改这个属性会产生背景色的渐变动画

position：用于设置CALayer的位置。修改这个属性会产生平移动画

关闭隐式动画 — 该地方涉及图层事务这个知识点

[CATransaction begin];

[CATransaction setDisableActions:YES];

//包裹的变更不发生隐式动画

self.layer.bounds=CGRectMake(0, 0, 200, 60);

self.layer.backgroundColor=[UIColor yellowColor].CGColor;

[CATransaction commit];

和UIView的关联与区别

1. UIView是iOS系统中界面元素的基础，所有的界面元素都继承自它。它本身完全由CoreAnimation来实现 的（Mac下似乎不是这样）。它真正的绘图部分，是由一个叫CALayer（Core Animation Layer）的类来 管理。UIView本身，更像是一个CALayer的管理器，访问它的跟绘图和跟坐标有关的属性，例如frame， bounds 等等，实际上内部都是在访问它所包含的CALayer的相关属性。

2.UIView有个layer属性，可以返回它的主CALayer实例，UIView有一个layerClass方法，返回主layer所 使用的类，UIView的子类，可以通过重载这个方法，来让UIView使用不同的CALayer来显示，例如

- (class) layerClass { return ([CAEAGLLayer class]); }

使某个UIView的子类使用GL来进行绘制。

比如返回具体子类CASHapeLayer，获取view的rootLayer后就可以直接添加path执行绘图

3. CALayer类似UIView的子View树形结构，也可以向它的layer上添加子layer，来完成某些特殊的表示。

grayCover = [[CALayer alloc] init];

grayCover.backgroundColor = [[[UIColor blackColor] colorWithAlphaComponent:0.2] CGColor];

[self.layer addSubLayer: grayCover];

//会在目标View上敷上一层黑色的透明薄膜。

4.UIView的layer树形在系统内部，被系统维护着三份copy（这段理解有点吃不准）。

第一份，逻辑树，就是代码里可以操纵的，例如更改layer的属性等等就在这一份。

第二份，动画树，这是一个中间层，系统正在这一层上更改属性，进行各种渲染操作。

第三份，显示树，这棵树的内容是当前正被显示在屏幕上的内容。

这三棵树的逻辑结构都是一样的，区别只有各自的属性。

图形绘制板：CAShapeLayer —> CALayer

和CALayer区别：

普通CALayer在被初始化时是需要给一个frame值的,这个frame值一般都与给定view的bounds值一致, 它本身是有形状的,而且是矩形

CAShapeLayer在初始化时也需要给一个frame值,但是,它本身没有形状,它的形状来源于你给定的一个path, 然后它去取CGPath值,它与CALayer有着很大的区

原则上：

CALayer中的许多属性，在发生变化时，默认使用（隐式动画，）方式展示这个变化

CAShapeLayer需要传入一个描画路径，然后设定相关属性来确定需要描画的路径是从什么到什么地方

然而还有第2中方式去改变一个对象的某个属性的值（通过键路径）（CABasicAnimation）

实例：Tab4 — 第3屏的viewWillAppear

// 贝塞尔曲线(创建一个圆路径，等待描画)

UIBezierPath \*path = [UIBezierPath bezierPathWithArcCenter:CGPointMake(100 / 2.f, 100 / 2.f) //圆心

radius:100 / 2.f //半径

startAngle:0 //开始角度，从右水平开始

endAngle:M\_PI \* 2 //结束角度

clockwise:YES]; //The direction in which to draw the arc

// 创建一个shapeLayer

CAShapeLayer \*layer = [CAShapeLayer layer];

layer.frame = showView.bounds; // 与showView的frame一致

layer.strokeColor = [UIColor greenColor].CGColor; // 路径的颜色（从start到end）

layer.fillColor = [UIColor yellowColor].CGColor; // 闭路径构成的图形的填充色

layer.lineCap = kCALineCapRound; // 线头是圆的（边框线在起点和端点的样式）

layer.path = path.CGPath; // 从路径对象获取路径

layer.lineWidth = 2.0f; // 线条宽度

layer.strokeStart = 0.0f; // 距离路径规定起点的距离（百分比）

layer.strokeEnd = 0.25f;

// 2s后执行动画操作(直接赋值就能产生动画效果)

double delayInSeconds = 1.0;

dispatch\_time\_t popTime = dispatch\_time(DISPATCH\_TIME\_NOW, (int64\_t)(delayInSeconds \* NSEC\_PER\_SEC));

dispatch\_after(popTime, dispatch\_get\_main\_queue(), ^(void){

layer.speed = 0.1; //durition/speed = 实际执行时间

layer.strokeStart = 0.5;

layer.strokeEnd = 1.0f;

layer.lineWidth = 5.0f;

});

// 将layer添加进shouView的根layer图层的子layer组中

[showView.layer addSublayer:layer];

渐变色图层：CAGradientLayer —> CALayer

// 创建一个gradientLayer(梯度颜色图层)

CAGradientLayer \*gradientLayer = [CAGradientLayer layer];

gradientLayer.frame = self.view.bounds;

//渐变色数组

[gradientLayer setColors:[NSArray arrayWithObjects:

(id)[[UIColor blueColor] CGColor],

(id)[[UIColor blackColor] CGColor], nil]];

//渐变色的区间分布（和渐变色数组一致 + 1，默认均匀分布）

[gradientLayer setLocations:@[@0.0,@0.7,@1.0]]; //@表示转化数字为数字对象NSNumber

//映射locations中第一个位置，用单位向量表示，比如（0，0）表示从左上角开始变化。默认值是(0.5,0.0)。

[gradientLayer setStartPoint:CGPointMake(0.5, 1)];

//映射locations中最后一个位置，用单位向量表示，比如（1，1）表示到右下角变化结束。默认值是(0.5,1.0)。

[gradientLayer setEndPoint:CGPointMake(0.5, 0)];

图层的几何变换：CATransform3D —— Tab6 — 第一屏/第二屏

原则：

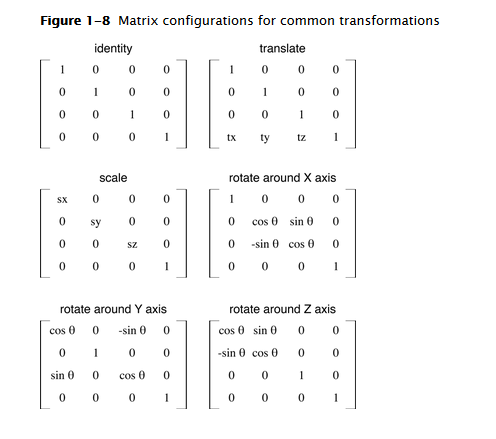
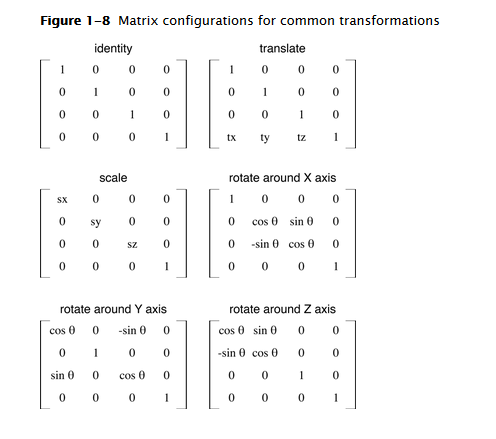
1. 而且父layer变换时，并不会吧子layer当做坐标空间的一部分来变换，仅仅是分别调用子layer的transform

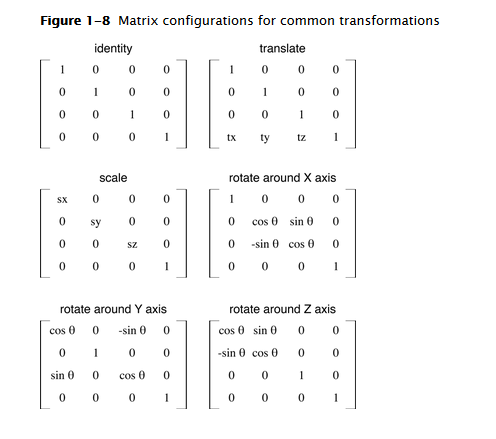
2. 一个layer的所有子layer，包括自己，都坐标空间独立，变换独立

3. 平移变换加上z后无效，可以看出平移是按顺序分别移动x,y,z然后显示最终位置，而不是求向量

4. 调用方法的旋转默认以直视对应正轴的顺时针（改变正负可以改变旋转方向）：

一张图明白一切：





0. 坐标原点在layer的锚点处

1. 头3行分别代表x，y，z单位向量方向，改变这些单位向量的指向，就会改变包含在其中的图像状态

3. 若是发生多次变换，第二次单位向量方向的改变，是相对于新扭曲坐标系的？？新坐标

4. 最后一行表示坐标原点(也就是锚点)的平移方向

5. 这个方向的确定也是相对于所在坐标系的？？是的，新坐标

核心成员：

struct CATransform3D

{

CGFloat m11, m12, m13, m14;

CGFloat m21, m22, m23, m24;

CGFloat m31, m32, m33, m34;

CGFloat m41, m42, m43, m44;

};

获取旋转的角度：

//获取旋转的相关信息

CATransform3D transform = self.rightImageView.layer.transform;

float routaion = [(NSNumber\*)[self.rightImageView.layer

valueForKeyPath:@"transform.rotation.y"] floatValue];

float angle = 0;

//因为angle会在越过90度后变为89，所以根据x，z的正负来确定角度

//直视z轴正向，顺时针为 0~180，逆时针为负

if (transform.m11 < 0 && transform.m13 >= 0) {

angle = -routaion\*180/M\_PI - 90 - 90.0;

}else if(transform.m11 < 0 && transform.m13 <= 0){

angle = 90 - routaion\*180/M\_PI + 90.0;

}else{

angle = routaion\*180/M\_PI;

}

NSLog(@"XoZ坐标轴上：");

NSLog(@" z:(%0.2f,%0.2f)",transform.m31,transform.m33);

NSLog(@" x:(%0.2f,%0.2f)",transform.m11,transform.m13);

NSLog(@" 旋转角度：%0.2f",angle);

涉及的变换 Tab6 — 第二屏 （函数名去掉make，换成动词就可以修改现有已经变换过的矩阵）：

0. 获取单位矩阵

self.view.subviews[2].layer.sublayers[0].transform = CATransform3DIdentity;

1. 平移变换

CATransform3D CATransform3DMakeTranslation (CGFloat tx, CGFloat ty, CGFloat tz)

两个图层关于z值的不同可以确定遮盖效果

2. 缩放

CATransform3D CATransform3DMakeScale (CGFloat sx, CGFloat sy, CGFloat sz);

当x,y单位向量的向量长，方向不变时（平面图形不变），z的变化仅改变图层遮罩顺序，不改变大小

3. 旋转

CATransform3D CATransform3DMakeRotation (CGFloat angle, CGFloat x, CGFloat y, CGFloat z);

传入的值需要变为π型角度：角度 \* M\_PI / 180。

CATransform3D trans\_1 = CATransform3DMakeRotation(20\*M\_PI/180, 0, 0, 1);

self.view.subviews[2].layer.sublayers[0].transform = trans\_1;

4. 镜像翻转

CATransform3D CATransform3DInvert (CATransform3D t);

怎么确定是上下镜像，还是左右镜像？？

5. 仿射效果

CGAffineTransform CATransform3DGetAffineTransform (CATransform3D t);

bool CATransform3DIsAffine (CATransform3D t);

就是把一个 CATransform3D 对象转换成一个 CGAffineTransform 对象。

也就是把 CATransform3D 矩阵 转换成 CGAffineTransform 矩阵

6. 比较矩阵

是否是单位矩阵：bool CATransform3DIsIdentity (CATransform3D t);

两个矩阵是否相同：bool CATransform3DEqualToTransform (CATransform3D a, CATransform3D b);

贝塞尔曲线绘图：UIBezierPath — MyWavePull大量实验

范例1 绘制弧线：Tab6—pageView文件

// 贝塞尔曲线(创建一个圆路径，等待描画)

UIBezierPath \*path = [UIBezierPath

bezierPathWithArcCenter:CGPointMake(300 / 2.f, 300 / 2.f) //圆心

radius:300 / 2.f //半径

startAngle:0 //开始角度，从右水平开始

endAngle:M\_PI \* 2 //结束角度

clockwise:YES]; //The direction in which to draw the arc

范例2 绘制圆角矩形：Tab4 — 的第3屏

//设置需要被圆角的角

UIRectCorner corner = isLeft ?

UIRectCornerTopLeft|UIRectCornerBottomLeft:

UIRectCornerTopRight|UIRectCornerBottomRight;

//将绘图曲线设置为……

layer.path = [UIBezierPath bezierPathWithRoundedRect:rect byRoundingCorners:corner cornerRadii:CGSizeMake(10, 10)].CGPath;

范例3 绘制三次曲线Curve ： MyWavePull — 第一屏 — secondLayer\_drawCurve方法

//上下两点拉扯方向变为：小正方形内的右上，左下 （切线点有变，看出和controlPoint连线一致）

CGPoint controlPoint1\_5 = CGPointMake(160,110);

CGPoint controlPoint2\_5 = CGPointMake(110,160);

[bezierPath moveToPoint:CGPointMake(110, 110)];

[bezierPath addCurveToPoint:CGPointMake(210,210) controlPoint1:controlPoint1\_5 controlPoint2:controlPoint2\_5];

[bezierPath addLineToPoint:CGPointMake(210,110)];

[bezierPath closePath];

[bezierPath moveToPoint:controlPoint1\_5];

[bezierPath addLineToPoint:controlPoint2\_5];

[bezierPath closePath];

\_secondLayer\_curve.path = bezierPath.CGPath;

1. 4个重要节点： 落笔点 ， 抬笔点 ， 前段弧控制点（前控制） ， 后端弧控制点（后控制）

2. 在控制点分立 起止连线 上下时：

曲向： 会各自偏向各自的控制点

分段点：为 两个控制点的连线 ， 与 起止点连线 的 交点

曲率：前控制 与 落笔点 连线，为 前段弧 的 起始切线

最高曲率点：为 控制点 和 各段中点 连线，与弧的交点

3. 在控制点位于 起止连线 同一侧时

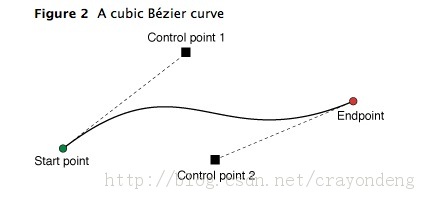
最高曲率点：使弧尽可能靠近 控制点连线

4. wavePull第二段感悟

start 与 end 水平一致时，pointer2和end一样，pointer1的x向右任意变化，pointer1的y向下， 但值保持不变，那么弧最低点距离水平的垂直距离在变化中不变！！

5. wavePull第1段曲线感悟（start不一定是弧的端点）

start左上，end右下，pointer2在连线下方时，可以使pointer1指向start的水平向左，使得弧度依旧 向上（但弧start端会神奇的越过start点）



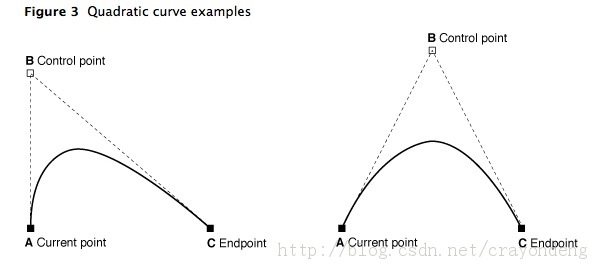
其他绘图：

1. 矩形：UIBezierPath\* aPath = [UIBezierPath bezierPathWithRect:CGRectMake(20, 20, 100, 50)];

2. 圆或椭圆：+ (UIBezierPath \*)bezierPathWithOvalInRect:(CGRect)rect

3. 二次贝塞尔曲线

- (void)addQuadCurveToPoint:(CGPoint)endPoint controlPoint:(CGPoint)controlPoint



4. 三次贝塞尔曲线 — 看范例3

-----------动画----------------------------------------------------------------------

动画时间协议：CAMediaTiming （CAAnimation实现该协议，协议包含8个属性）

代码实例（滑块控制渐变颜色）：

CABasicAnimation \*changeColor = [CABasicAnimation animationWithKeyPath:@"backgroundColor"];

changeColor.fromValue = (id)[UIColor orangeColor].CGColor;

changeColor.toValue = (id)[UIColor blueColor].CGColor;

changeColor.duration = 1.0; // For convenience

[self.myLayer addAnimation:changeColor forKey:@"Change color"];

self.myLayer.speed = 0.0;

- (IBAction)sliderChanged:(UISlider \*)sender {

self.myLayer.timeOffset = sender.value; // Update "current time"

}

设定变化的端点值后，通过设定speed值为0使得动画每次执行1帧

然后通过滑块回调方法不断设置timeoffset为0~1某个值，每次设定都会触发动画执行一帧

最终达到使用滑块调整layer颜色渐变的效果

代码实例：Tab4 — 第3面板 暂停动画和继续动画（但是继续动画是从“如果没有暂停”的动画位置继续的）

-(double)pauseLayer:(CALayer\*)layer

{

//Returns the current absolute time, in seconds.

NSLog(@"到现在的绝对时间是：%lf",CACurrentMediaTime());

//layer的时间空间是以10秒计数，记录的同样是绝对时间

//Converts the time interval from the specified layer’s time space to receiver’s time space.

CFTimeInterval pausedTime = [layer convertTime:CACurrentMediaTime() fromLayer:nil];

NSLog(@"layer空间的时间\*10 为：%lf",pausedTime\*10);

double oldSpeed = layer.speed;

layer.speed = 0.0;

//offset设定的是绝对时间，动画会根据绝对时间与自空间开始时间(也是绝对时间)来获取对应的状态

layer.timeOffset = pausedTime;

return oldSpeed;

}

-(void)resumeLayer:(CALayer\*)layer oldSpeed:(double)oldSpeed

{

// CFTimeInterval pausedTime = layer.timeOffset;

// CFTimeInterval timeSincePause = [layer convertTime:CACurrentMediaTime() fromLayer:nil] - pausedTime;

// layer.beginTime = timeSincePause;

layer.speed = oldSpeed;

layer.timeOffset = 0.0;

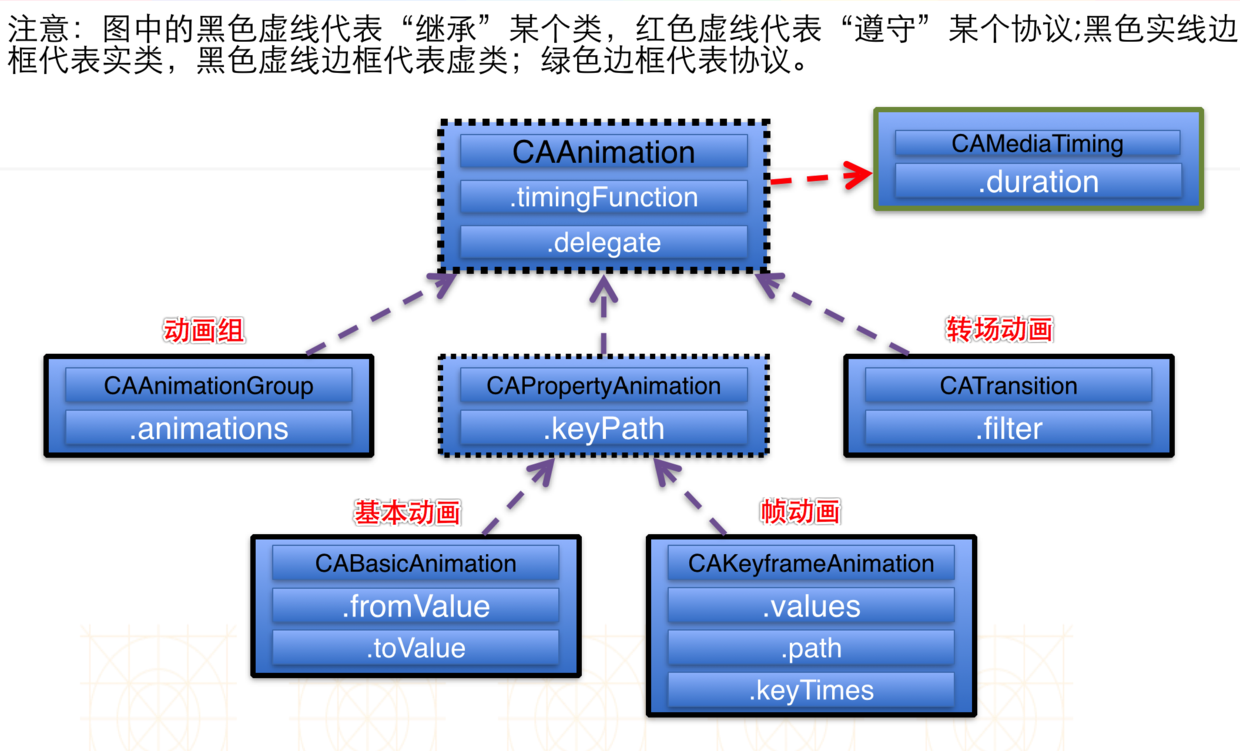
}

属性表：

|  |  |
| --- | --- |
| duration | beginTime |
| 设置fillMode为kCAFillModeBackwards（使得开始前状态为动画初始状态） | autoreverses属性可以产生从初始值到最终值，并反过来回到初始值的动画。这意味着动画发生了两次 |
| repeatCount可以将动画重复两次(如下所示)或任意次 | repeat duration |
| 组合使用 | duration为3秒，但speed为2，动画加快执行为1.5秒 |
| duration为3秒，offset为1秒的动画 |  |

动画编辑器 CAAnimation：

类图树



CAAnimation基本属性（红色为不在CAMediaTiming出现的）

duration：动画的持续时间

repeatCount：重复次数，无限循环可以设置HUGE\_VALF或者MAXFLOAT

repeatDuration：重复时间

removedOnCompletion：默认为YES，代表动画执行完毕后就从图层上移除，图形会恢复到动画执行前的状 态。如果想让图层保持显示动画执行后的状态，那就设置为NO，不过还要设置fillMode为 kCAFillModeForwards

beginTime：可以用来设置动画延迟执行时间，若想延迟2s，就设置为CACurrentMediaTime()+2，

timingFunction：速度控制函数，控制动画运行的节奏

kCAMediaTimingFunctionLinear（线性）：匀速，给你一个相对静态的感觉

kCAMediaTimingFunctionEaseIn（渐进）：动画缓慢进入，然后加速离开

kCAMediaTimingFunctionEaseOut（渐出）：动画全速进入，然后减速的到达目的地

kCAMediaTimingFunctionEaseInEaseOut（渐进渐出）：动画缓慢的进入，中间加速，然后减速的到达 目的地。这个是默认的动画行为。

delegate：动画代理

CAAnimation给NSobject添加了分类中定义了代理方法，所以任何对象都可以直接成为代理者

其实和实现协议然后实现方法一样，只不过这里不用实现协议，覆盖父类方法即可

动画开始的时候调用

- (void)animationDidStart:(CAAnimation \*)anim;

动画停止的时候调用

- (void)animationDidStop:(CAAnimation \*)anim finished:(BOOL)flag;

fillMode决定当前对象在非active时间段的行为。（要想fillMode有效，设置removedOnCompletion = NO）

kCAFillModeRemoved 这个是默认值，也就是说当动画开始前和动画结束后，动画对layer都没有影响， 动画结束后，layer会恢复到之前的状态

kCAFillModeForwards 当动画结束后，layer会一直保持着动画最后的状态

kCAFillModeBackwards 在动画开始前，只需要将动画加入了一个layer，layer便立即进入动画的初始 状态并等待动画开始（因为有时动画的初始状态和layer那时状态不一样）。

kCAFillModeBoth 这个其实就是上面两个的合成.动画加入后开始之前，layer便处于动画初始状态，动 画结束后layer保持动画最后的状态

单值动画 CABasicAnimation <— CAPropertyAnimation <— CAAnimation

示范代码

CABasicAnimation \*pathAnimation = [CABasicAnimation animationWithKeyPath:@"strokeEnd"];

pathAnimation.duration = 1.0;

pathAnimation.fromValue = [NSNumber numberWithFloat:0.25f];

pathAnimation.toValue = [NSNumber numberWithFloat:0.5f];

[layer addAnimation:pathAnimation forKey:nil];

将会看到加载到时立刻发生动画，但是1秒动画结束后立刻回到原状态

1. 首先得有CALayer（因为CoreAnimation是作用在CALayer上的）

2. 初始化一个CAAnimation对象，并设置一些动画相关属性

3. 通过调用CALayer的addAnimation:forKey:方法，增加CAAnimation对象到CALayer中，这样就能开 始执行动画了（事后证明对于许多属性，你在改变他时会有隐式动画发生）

4. 通过调用CALayer的removeAnimationForKey:方法可以停止CALayer中的动画

关键帧动画 CAKeyFrameAnimation <— CAPropertyAnimation <— CAAnimation

解释：

关键帧动画就是在动画控制过程中开发者指定主要的动画状态，至于各个状态间动画如何进行则由系统自 动运算补充

关键帧动画开发分为两种形式：一种是通过设置不同的属性值进行关键帧控制，另一种是通过绘制路径进 行关键帧控制。后者优先级高于前者，如果设置了路径则属性值就不再起作用

http://www.cnblogs.com/kenshincui/p/3972100.html有很详细的示范（樱花飘落，贝塞尔曲线）

属性说明：

values：上述的NSArray对象。里面的元素称为“关键帧”(keyframe)。动画对象会在指定的时间（duration） 内，依次显示values数组中的每一个关键帧

path：可以设置一个CGPathRef、CGMutablePathRef，让图层按照路径轨迹移动。path只对CALayer 的anchorPoint和position起作用。如果设置了path，那么values将被忽略

keyTimes：可以为对应的关键帧指定对应的时间点，其取值范围为0到1.0，keyTimes中的每一个时间 值都对应values中的每一帧。如果没有设置keyTimes，各个关键帧的时间是平分的

示范代码

//设置关键帧,这里有四个关键帧

//对于关键帧动画初始值不能省略

NSValue \*key1=[NSValue valueWithCGPoint:layer.position];

NSValue \*key2=[NSValue valueWithCGPoint:CGPointMake(80, 220)];

NSValue \*key3=[NSValue valueWithCGPoint:CGPointMake(45, 300)];

NSValue \*key4=[NSValue valueWithCGPoint:CGPointMake(55, 400)];

NSArray \*values=@[key1,key2,key3,key4];

keyframeAnimation.values=values;

//设置其他属性

keyframeAnimation.duration=8.0;

keyframeAnimation.beginTime=CACurrentMediaTime()+2;//设置延迟2秒执行

//3.添加动画到图层，添加动画后就会执行动画

[\_layer addAnimation:keyframeAnimation forKey:@"KCKeyframeAnimation\_Position"];

转场动画 CATransition <— CAAnimation

CATransition是CAAnimation的子类，用于做转场动画，能够为层提供移出屏幕和移入屏幕的动画效果。iOS 比Mac OS X的转场动画效果少一点UINavigationController就是通过CATransition实现了将控制器的 视图推入屏幕的动画效果

代码实例：Tab4 — 第3面板

CATransition \*transition\_1 = [[CATransition alloc]init];

transition\_1.type = @"suckEffect"; //转场效果

[transition\_1 setDuration:2.0];

[transition\_1 setFillMode:kCAFillModeForwards];

[transition\_1 setSubtype:kCATransitionFromTop]; //进入方向

[showView.layer addAnimation:transition\_1 forKey:nil];

[showView.layer addSublayer:gradientLayer];

注意：为个rootlayer添加该转场特性，其子layer才会在add时会触发该转场

动画属性:

type：动画过渡类型

subtype：动画过渡方向

startProgress：动画起点(在整体动画的百分比)

endProgress：动画终点(在整体动画的百分比)

转场动画的类型（NSString \*type）

fade : 交叉淡化过渡 push : 新视图把旧视图推出去

moveIn: 新视图移到旧视图上面 reveal: 将旧视图移开,显示下面的新视图

cube : 立方体翻滚效果 oglFlip: 上下左右翻转效果

suckEffect : 收缩效果，如一块布被抽走 rippleEffect: 水滴效果

pageCurl : 向上翻页效果 pageUnCurl : 向下翻页效果

cameraIrisHollowOpen : 相机镜头打开效果 cameraIrisHollowClos : 相机镜头关闭效果

转场动画的方向（NSString \*subtype）

从某个方向开始：fromLeft, fromRight, fromTop ,fromBottom

单视图

+ (void)transitionWithView:(UIView \*)view duration:(NSTimeInterval)duration options:(UIViewAnimationOptions)options animations:(void (^)(void))animations completion:(void (^)(BOOL finished))completion;

参数说明：

duration：动画的持续时间

view：需要进行转场动画的视图

options：转场动画的类型

animations：将改变视图属性的代码放在这个block中

completion：动画结束后，会自动调用这个block

双视图

+ (void)transitionFromView:(UIView \*)fromView toView:(UIView \*)toView duration:(NSTimeInterval)duration options:(UIViewAnimationOptions)options completion:(void (^)(BOOL finished))completion;

参数说明：

duration：动画的持续时间

options：转场动画的类型

animations：将改变视图属性的代码放在这个block中

completion：动画结束后，会自动调用这个block

动画组 CAAnimationGroup <— CAAnimation

动画组，是CAAnimation的子类，可以保存一组动画对象，将CAAnimationGroup对象加入层后，组中所 有动画对象可以同时并发运行

属性说明：

animations：用来保存一组动画对象的NSArray

默认情况下，一组动画对象是同时运行的，也可以通过设置动画对象的beginTime属性来更改动画的开始时间

逐帧动画—CADisplayLink <— NSObject

说到逐帧动画相信很多朋友第一个想到的就是UIImageView，通过设置UIImageView的animationImages 属性，然后调用它的startAnimating方法去播放这组图片。当然这种方法在某些场景下是可以达到逐帧 的动画效果，但是它也存在着很大的性能问题，并且这种方法一旦设置完图片中间的过程就无法控制了。 当然，也许有朋友会想到利用iOS的定时器NSTimer定时更新图片来达到逐帧动画的效果。这种方式确实可 以解决UIImageView一次性加载大量图片的问题，而且让播放过程可控，唯一的缺点就是定时器方法调 用有时可能会因为当前系统执行某种比较占用时间的任务造成动画连续性出现问题

虽然在核心动画没有直接提供逐帧动画类型，但是却提供了用于完成逐帧动画的相关对象CADisplayLink。 CADisplayLink是一个计时器，但是同NSTimer不同的是，CADisplayLink的刷新周期同屏幕完全一致。 例如在iOS中屏幕刷新周期是60次/秒，CADisplayLink刷新周期同屏幕刷新一致也是60次/秒，这样一 来使用它完成的逐帧动画（又称为“时钟动画”）完全感觉不到动画的停滞情况

iOS程序在运行后就进入一个消息循环中（这个消息循环称为“主运行循环”），整个程序相当于进入一个死循 环中，始终等待用户输入。将CADisplayLink加入到主运行循环队列后，它的时钟周期就和主运行循环保 持一致，而主运行循环周期就是屏幕刷新周期。在CADisplayLink加入到主运行循环队列后就会循环调用 目标方法，在这个方法中更新视图内容就可以完成逐帧动画。

代码实例—Tab6—第一屏

-(void) addDisplayLink{

//定义时钟对象(并传入委托人和回调方法)

CADisplayLink \*displayLink=[CADisplayLink displayLinkWithTarget:self selector:@selector(step)];

//添加时钟对象到主运行循环

[displayLink addToRunLoop:[NSRunLoop mainRunLoop] forMode:NSDefaultRunLoopMode];

}

#pragma mark 每次屏幕刷新就会执行一次此方法(每秒接近60次)

-(void)step{

//定义一个变量记录执行次数

static int s=0;

//每次调用都递增（溢出后并不妨碍%运算）

s++;

//每秒执行6次就可以了

if (s%10==0) {

UIImage \*image=\_images[\_index];

\_layer.contents=(id)image.CGImage;//更新图片

\_index=(\_index+1)%IMAGE\_COUNT;

}

}

猜想：

link对象在addToRunLoop方法中，将对象中的某个方法（runInLoopFunc）交给runloop去执行了

removeFromRunLoop方法调用runLoop接口，将runInLoopFunc从runloop中移除

runInLoopFunc包含一个判断入口，判断paused，若YES则不执行link回调，但是runloop依旧执行着这段代码

//invalidate将removeFromAllRunLoop，并且link对象释放target

//The default value is NO. If YES, the display link does not send notifications to the target.

self.bounceLink.paused = YES;

//Removing the display link from all run loop modes causes it to be released by the run loop. display link releases the target.

[self.bounceLink invalidate];

//每秒执行30次：The number of frames that must before the display link notifies the target again.

[\_bounceLink setFrameInterval:2];

UIView封装动画 - 弹簧动效：

执行代码

[UIView animateWithDuration:0.5 delay:0 options:UIViewAnimationOptionCurveLinear animations:^{

\_imageView.center=location;

} completion:^(BOOL finished) {

NSLog(@"Animation end. 坐标：%0.1f,%0.1f",location.x,location.y);

}];

弹簧动效

/\*创建弹性动画

damping:阻尼，范围0-1，阻尼越接近于0，回弹次数越多

velocity:弹性复位的速度，越大，回复时间越长

\*/

[UIView animateWithDuration:1 delay:0 usingSpringWithDamping:0.1 initialSpringVelocity:4 options:UIViewAnimationOptionCurveLinear animations:^{

\_imageView.center=location;

} completion:nil];

options的可选项：

1.常规动画属性设置（可以同时选择多个进行设置）

UIViewAnimationOptionLayoutSubviews：动画过程中保证子视图跟随运动。

UIViewAnimationOptionAllowUserInteraction：动画过程中允许用户交互。

UIViewAnimationOptionBeginFromCurrentState：所有视图从当前状态开始运行。

UIViewAnimationOptionRepeat：重复运行动画。

UIViewAnimationOptionAutoreverse ：动画运行到结束点后仍然以动画方式回到初始点。

UIViewAnimationOptionOverrideInheritedDuration：忽略嵌套动画时间设置。

UIViewAnimationOptionOverrideInheritedCurve：忽略嵌套动画速度设置。

UIViewAnimationOptionAllowAnimatedContent：动画过程中重绘视图（注意仅适用于转场动画）。

UIViewAnimationOptionShowHideTransitionViews：视图切换时直接隐藏旧视图、显示新视图， 而不是将旧视图从父视图移除（仅仅适用于转场动画）

UIViewAnimationOptionOverrideInheritedOptions ：不继承父动画设置或动画类型。

2.动画速度控制（可从其中选择一个设置）

UIViewAnimationOptionCurveEaseInOut：慢进 加快 慢出。

UIViewAnimationOptionCurveEaseIn ：慢进快出

UIViewAnimationOptionCurveEaseOut：快进慢出

UIViewAnimationOptionCurveLinear ：动画匀速执行，默认值。

3.转场类型（仅适用于转场动画设置，可以从中选择一个进行设置，基本动画、关键帧动画不需要设置）

UIViewAnimationOptionTransitionNone：没有转场动画效果

UIViewAnimationOptionTransitionFlipFromLeft ：从左侧翻转效果。

UIViewAnimationOptionTransitionFlipFromRight：从右侧翻转效果。

UIViewAnimationOptionTransitionCurlUp：向后翻页的动画过渡效果。

UIViewAnimationOptionTransitionCurlDown ：向前翻页的动画过渡效果。

UIViewAnimationOptionTransitionCrossDissolve：旧视图溶解消失显示下一个新视图的效果。

UIViewAnimationOptionTransitionFlipFromTop ：从上方翻转效果。

UIViewAnimationOptionTransitionFlipFromBottom：从底部翻转效果。

SpringView弹簧动效

网址：http://www.cnblogs.com/itlover2013/p/4848432.html

源码：

let spring = CASpringAnimation(keyPath: "position.x")

spring.damping = 5;

spring.stiffness = 100;

spring.mass = 1;

spring.initialVelocity = 0;

spring.fromValue = label.layer.position.x;

spring.toValue = label.layer.position.x + 50;

spring.duration = spring.settlingDuration;

label.layer.addAnimation(spring, forKey: spring.keyPath);

mass:

质量，影响图层运动时的弹簧惯性，质量越大，弹簧拉伸和压缩的幅度越大

如果把质量改成10， 动画的速度变慢，并且波动幅度变大

stiffness:

刚度系数(劲度系数/弹性系数)，刚度系数越大，形变产生的力就越大，运动越快

damping:

阻尼系数，阻止弹簧伸缩的系数，阻尼系数越大，停止越快

initialVelocity:

初始速率，动画视图的初始速度大小

速率为正数时，速度方向与运动方向一致，速率为负数时，速度方向与运动方向相反

如果把速率改成-20， 这里会看到 label 先往左边移动一小段距离，然后再开始往右移动，因为初始速度往左， 所以会经历先减速，后加速的过程

settlingDuration:

结算时间 返回弹簧动画到停止时的估算时间，根据当前的动画参数估算

通常弹簧动画的时间使用结算时间比较准确

----------微博 Demo 性能优化技巧（待研究）-------------------------------

几个技巧预览

预排版

当获取到 API JSON 数据后，我会把每条 Cell 需要的数据都在后台线程计算并封装为一个布局对象 CellLayout。CellLayout 包含所有文本的 CoreText 排版结果、Cell 内部每个控件的高度、Cell 的整体 高度。每个 CellLayout 的内存占用并不多，所以当生成后，可以全部缓存到内存，以供稍后使用。这样， TableView 在请求各个高度函数时，不会消耗任何多余计算量；当把 CellLayout 设置到 Cell 内部时， Cell 内部也不用再计算布局了。

对于通常的 TableView 来说，提前在后台计算好布局结果是非常重要的一个性能优化点。为了达到最高性能， 你可能需要牺牲一些开发速度，不要用 Autolayout 等技术，少用 UILabel 等文本控件。但如果你对性 能的要求并不那么高，可以尝试用 TableView 的预估高度的功能，并把每个 Cell 高度缓存下来。这里 有个来自百度知道团队的开源项目可以很方便的帮你实现这一点：FDTemplateLayoutCell

预渲染

微博的头像在某次改版中换成了圆形，所以我也跟进了一下。当头像下载下来后，我会在后台线程将头像预先 渲染为圆形并单独保存到一个 ImageCache 中去。

对于 TableView 来说，Cell 内容的离屏渲染会带来较大的 GPU 消耗。在 Twitter Demo 中，我为了图省 事儿用到了不少 layer 的圆角属性，你可以在低性能的设备（比如 iPad 3）上快速滑动一下这个列表， 能感受到虽然列表并没有较大的卡顿，但是整体的平均帧数降了下来。用 Instument 查看时能够看到 GPU 已经满负荷运转，而 CPU 却比较清闲。为了避免离屏渲染，你应当尽量避免使用 layer 的 border、 corner、shadow、mask 等技术，而尽量在后台线程预先绘制好对应内容。

异步绘制

我只在显示文本的控件上用到了异步绘制的功能，但效果很不错。我参考 ASDK 的原理，实现了一个简单的 异步绘制控件。这块代码我单独提取出来，放到了这里：YYAsyncLayer。YYAsyncLayer 是 CALayer 的 子类，当它需要显示内容（比如调用了 [layer setNeedDisplay]）时，它会向 delegate，也就是 UIView 请求一个异步绘制的任务。在异步绘制时，Layer 会传递一个 BOOL(^isCancelled)() 这样的 block，绘 制代码可以随时调用该 block 判断绘制任务是否已经被取消。

当 TableView 快速滑动时，会有大量异步绘制任务提交到后台线程去执行。但是有时滑动速度过快时，绘制 任务还没有完成就可能已经被取消了。如果这时仍然继续绘制，就会造成大量的 CPU 资源浪费，甚至阻 塞线程并造成后续的绘制任务迟迟无法完成。我的做法是尽量快速、提前判断当前绘制任务是否已经被取 消；在绘制每一行文本前，我都会调用 isCancelled() 来进行判断，保证被取消的任务能及时退出，不至 于影响后续操作。

目前有些第三方微博客户端（比如 VVebo、墨客等），使用了一种方式来避免高速滑动时 Cell 的绘制过程， 相关实现见这个项目：VVeboTableViewDemo。它的原理是，当滑动时，松开手指后，立刻计算出滑动 停止时 Cell 的位置，并预先绘制那个位置附近的几个 Cell，而忽略当前滑动中的 Cell。这个方法比较有 技巧性，并且对于滑动性能来说提升也很大，唯一的缺点就是快速滑动中会出现大量空白内容。如果你不 想实现比较麻烦的异步绘制但又想保证滑动的流畅性，这个技巧是个不错的选择。

全局并发控制（作者自己的线程池）

当我用 concurrent queue 来执行大量绘制任务时，偶尔会遇到大量奇怪并发线程的问题

大量的任务提交到后台队列时，某些任务会因为某些原因（此处是 CGFont 锁）被锁住导致线程休眠，或者被 阻塞，concurrent queue 随后会创建新的线程来执行其他任务。当这种情况变多时，或者 App 中使用 了大量 concurrent queue 来执行较多任务时，App 在同一时刻就会存在几十个线程同时运行、创建、 销毁。CPU 是用时间片轮转来实现线程并发的，尽管 concurrent queue 能控制线程的优先级，但当大 量线程同时创建运行销毁时，这些操作仍然会挤占掉主线程的 CPU 资源。ASDK 有个 Feed 列表的 Demo：SocialAppLayout，当列表内 Cell 过多，并且非常快速的滑动时，界面仍然会出现少量卡顿， 我谨慎的猜测可能与这个问题有关。

使用 concurrent queue 时不可避免会遇到这种问题，但使用 serial queue 又不能充分利用多核 CPU 的资 源。我写了一个简单的工具 YYDispatchQueuePool，为不同优先级创建和 CPU 数量相同的 serial queue，每次从 pool 中获取 queue 时，会轮询返回其中一个 queue。我把 App 内所有异步操作， 包括图像解码、对象释放、异步绘制等，都按优先级不同放入了全局的 serial queue 中执行，这样尽量 避免了过多线程导致的性能问题。

更高效的异步图片加载

SDWebImage 在这个 Demo 里仍然会产生少量性能问题，并且有些地方不能满足我的需求，所以我自己实 现了一个性能更高的图片加载库。在显示简单的单张图片时，利用 UIView.layer.contents 就足够了，没 必要使用 UIImageView 带来额外的资源消耗，为此我在 CALayer 上添加了 setImageWithURL 等方 法。除此之外，我还把图片解码等操作通过 YYDispatchQueuePool 管理，控制了 App 总线程数量。

微博Demo代码精析

见《微博Demo代码精析》文档