核心动画编程指南

原著：Apple Inc.

翻译：謝業蘭

联系：xyl.layne@gmail.com

鸣谢：有米移动广告平台

CocoaChina 社区

目录

核心动画编程介绍

本文档介绍了在使用核心动画时所涉及的基本概念。核心动画的是 Objective－C 的框架，它通过简单的动画编程接口来提供一套高性能的动画引擎。

你应该阅读此文档来理解Cocoa应用程序核心动画工作的机制。 阅读此文档的前提是你已经掌握了Objective-C语言的基础，因为核心动画内部广泛的使用了Objective-C的相关属性。你还应该熟悉键-值编程方法（参考文档 *Key-Value Coding Programming Guide*）。同时如果你对Quartz 2D编程有一定了解的话，将会对您掌握本文档带来帮助，但这不是必须的（参考文档 *Quartz 2D Programming Guide*）。

您可以建立两个平台的 Cocoa 应用程序：在 Mac OS X 操作系统和 iOS 操作系统，如 iPhone 和 iPod touch 多点触控设备。核心动画编程指南对这两个平台同时适应，它尽可能的整合两个平台的相同性，同时在必要的时候指出它们的差异性。

本文档结构

核心动画编程指南包含以下内容：

- “核心动画概念”提供核心动画的概述。
- “图层（Layer）的几何和变换” 描述图层的几何和变换。
- “图层树的层次结构”描述图层的结构树和如何在应用程序中使用它。
- “提供图层内容” 介绍如何提供基本图层的内容。
- “动画” 介绍了核心动画的动画模型。
- “图层行为”介绍图层的行为，同时实现隐式动画。
- “事务（transactions）”介绍如果通过事务来组合动画。
- “布局核心动画的图层”描述布局管理器的限制。
- “核心动画的扩展键-值编码”描述核心动画提供的键-值编码。
- “图层样式属性”描述了图层样式属性，并提供其视觉效果的例子。
- “范例：核心动画菜单样式应用”解剖一个核心动画驱动的用户界面。
- “动画的属性”概括层和滤镜的动画属性。

第一章　核心动画概念

核心动画是一套包含图形绘制，投影，动画的 Objective－C 类集合。它通过开发人员所熟悉的应用程序套件和 Cocoa Touch 视图架构的抽象分层模式，同时使用先进的合作效果提供了一套流畅的动画。

动态的动画接口很难创建，但是核心动画通过提供如下接口使这些创建起来变得更加简单:

- 简单易用的高性能混合编程模型。
- 类似视图一样，你可以通过使用图层来创建复杂的接口。
- 轻量级的数据结构，它可以同时显示并让上百个图层产生动画效果。
- 一套简单的动画接口，可以让你的动画运行在独立的线程里面，并可以独立于主线程之外。
- 一旦动画配置完成并启动，核心动画完全控制并独立完成相应的动画帧。
- 提高应用性能。应用程序只当发生改变的时候才重绘内容。再小的应用程序也需要改变和提供布局服务层。核心动画还消除了在动画的帧速率上运行的应用程序代码。
- 灵活的布局管理模型。包括允许图层相对同级图层的关系来设置相应属性的位置和大小。

使用核心动画，开发人员可以为他们的应用创建动态用户界面，而无需使用低级别的图形 API，如 OpenGL 来获取高效的动画性能。

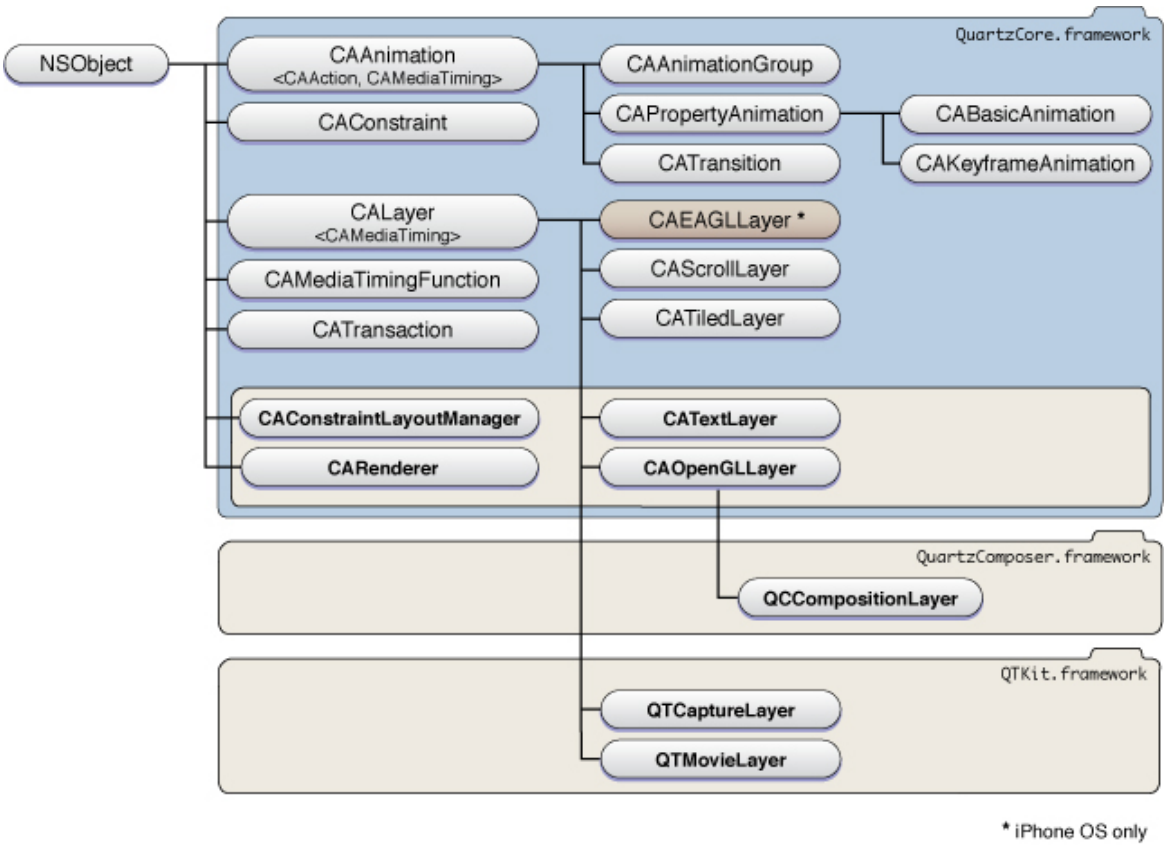
1.1　核心动画类

核心动画类有以下分类：

- 提供显示内容的图层类。
- 动画和计时类。
- 布局和约束类。
- 事务类，在原子更新的时候组合图层类。

核心动画的基础类包含在 Quartz 核心框架（Quartz Core framework）里面，虽然它的其他图层类在其他框架里面定义。下图显示了核心动画的类层次结构。

Figure 1 Core Animation class hierarchy



1.1.1 图层类（Layer Classes）

图层类是核心动画的核心基础，它提供了一套抽象的概念（假如你使用过 NSView 或者 UIView 的话，你一定会对它很熟悉）。CALayer 是整个图层类的基础，它是所有核心动画图层类的父类。

和视图类（NSView 或 UIView）一样，CALayer 有自己的父图层类，同时也拥有自己子图层类的集合，它们构成了一个图层树的层次结构。图层绘制类似视图一样自底向上，并指定其几何形状相对他们 superlayer，创建一个相对的局部坐标系。然而图层类通过合并变换矩阵允许你旋转、偏移、拉伸对应的图层内容。具体内容将会在后面“图层（Layer）的几何和变换”部分介绍。

CALayer 从 Application Kit 和 Cocoa Touch 的视图类分离出来，因为没有必要为了显示内容而继承 CALayer 类。因为 CALayer 类的内容显示可以通过以下方法提供：

- 可以直接或者委托的方式把图层的内容属性设置为 Core Graphics image。
- 提供直接绘制到一个 Core Graphics image 上下文委托。

- 设置所有图层所具有的可视化样式属性，比如背景颜色、不透明属性、蒙版等。

  Mac OS X 应用同样可以通过核心图像滤镜来访问它的可视化样式属性。

  继承 CALayer 并通过封装方法实现以上任何技术。

“提供图层内容”描述了提供内容层可用方法。可视化样式样式属性和它们的顺序将会在 “图层样式属性”部分详细介绍。

除了 CALayer 类，核心动画类同时提供了显示其他内容的类。这些类在 Mac OS X 和 iOS 上有细微的差别，以下类在 Mac OS X 和 iOS 上都可用:

- CAScrollLayer 是 CALayer 的子类，简化显示图层的一部分内容。CAScrollLayer 对象的滚动区域的范围在它的子图层里面定义。CAScrollLaye 不提供键盘或鼠标事件处理，也不提供可见的滚动条。
- CATextLayer 可以方便的从字符串或字符串的内容创建一个图层类的内容。
- CATiledLayer 允许递增的显示大而复杂的图片。

Mac OS X 提供如下额外的类：

- CAOpenGLLayer 提供了一个 OpenGL 渲染环境。你必须继承这个类来使用 OpenGL 提供的内容。内容可以是静态的，或可随着时间的推移更新。
- QCCompositionLayer（由 Quartz 框架提供）可以把 Quartz 合成的内容动画显示。
- QTMovieLayer and QTCaptureLayer （QTKit 框架提供）提供播放 QuickTime 影片和视频直播。

iOS adds the following class:

- CAEAGLLayer 提供了一个 OpenGLES 渲染环境。

CALayer 的类引入键-值编码兼容的容器类概念，也就是说一个类可以使用键 - 值编码的方法存储任意值，而无需创建一个子类。CALayer 的还扩展了 NSKeyValueCoding 的非正式协议，加入默认键值和额外的结构类型的自动对象包装（CGPoint，CGSize，CGRect，CGAffineTransform 和 CATransform3D）的支持，并提供许多这些结构的关键路径领域的访问。

CALayer 同时管理与层关联的动画和行为，。图层接受层树的插入和删除层动作，修改层的属性，或者明确的开发请求。这些行为通常会导致动画发生。见“动画”和“图层操作”的更多信息。

1.1.2 动画和计时类

图层的很多可视化属性是可以隐式动画的。通过简单的改变图层的可动画显示的属性，可以让图层现有属性从当前值动画渐变到新的属性值。例如设置图层的 hidden 属性为 YES 将会触发动画使层逐渐淡出。大多数动画属性拥有自己关联的默认动画，你可以轻松地定制和替换。我们将会在后面“动画属性”部分列出一个完整的动画属性列表和它们相应的默认动画。动画的属性也可以显式动画。要显式动画的属性，你需要创建核心动画动画类的一个实例，并指定所需的视觉效果。显式动画不会改变该属性的值，它只是用于动画显示。

核心动画的动画类使用基本的动画和关键帧动画把图层的内容和选取的属性动画的显示出来。所有核心动画的动画类都是从 CAAnimation 类继承而来。CAAnimation 实现了 CAMediaTiming 协议，提供了动画的持续时间，速度，和重复计数。CAAnimation 也实现了 CAAction 协议。该协议为图层触发一个动画动作提供了提供标准化响应。

动画类同时定义了一个使用贝塞尔曲线来描述动画改变的时间函数。例如，一个匀速时间函数（linear timing function）在动画的整个生命周期里面一直保持速度不变，而渐缓时间函数（ease-out timing function）则在动画接近其生命周期的时候减慢速度。

核心动画额外提供了一系列抽象的和细化的动画类，比如：

CATransition 提供了一个图层变化的过渡效果，它能影响图层的整个内容。动画进行的时候淡入淡出（fade）、推（push）、显露(reveal)图层的内容。这些过渡效果可以扩展到你自己定制的 Core Image 滤镜。

- CAAnimationGroup 允许一系列动画效果组合在一起，并行显示动画。
- CAPropertyAnimation 是一个抽象的子类，它支持动画的显示图层的关键路径中指定的属性
- CABasicAnimation 简单的为图层的属性提供修改。
- CAKeyframeAnimation 支持关键帧动画，你可以指定的图层属性的关键路径动画，包括动画的每个阶段的价值，以及关键帧时间和计时功能的一系列值。在动画运行是，每个值被特定的插入值替代。

核心动画 和 Cocoa Animation 同时使用这些动画类。使用动画描述，是因为这些类涉及到核心动画, 这些将会在Animation Types and Timing Programming Guide 有较深入的讨论。

1.1.3 布局管理器类

Application Kit 的视图类相对于 superlayer 提供了经典的“struts and springs”定位模型。图层类兼容这个模型，同时 Mac OS X 上面的核心动画提供了一套更加灵活的布局管理机制，它允许开发者自己修改布局管理器。核心动画的 CAConstraint 类是一个布局管理器，它可以指定子图层类限制于你指定的约束集合。每个约束（CAConstraint 类的实例封装）描述层的几何属性（左，右，顶部或底部的边缘或水平或垂直中心）的关系，关系到其同级之一的几何属性层或 superlayer。

通用的布局管理器和约束性布局管理器将会在“布局核心动画的图层”部分讨论。

1.1.4 事务管理类

图层的动画属性的每一个修改必然是事务的一个部分。CATransaction 是核心动画

里面负责协调多个动画原子更新显示操作。事务支持嵌套使用。

核心动画支持两种事务:**隐式事务**和**显式事务**。在图层的动画属性被一个线程修改，同时该线程下次迭代的时候自动提交该修改的时候隐式事务自动创建。显式事务发生在程序在修改动画属性之前给 CATransaction 发送了一个开始消息，在动画属性修改之后提交该消息。

事务管理将会在后面的“事务”部分详细介绍。

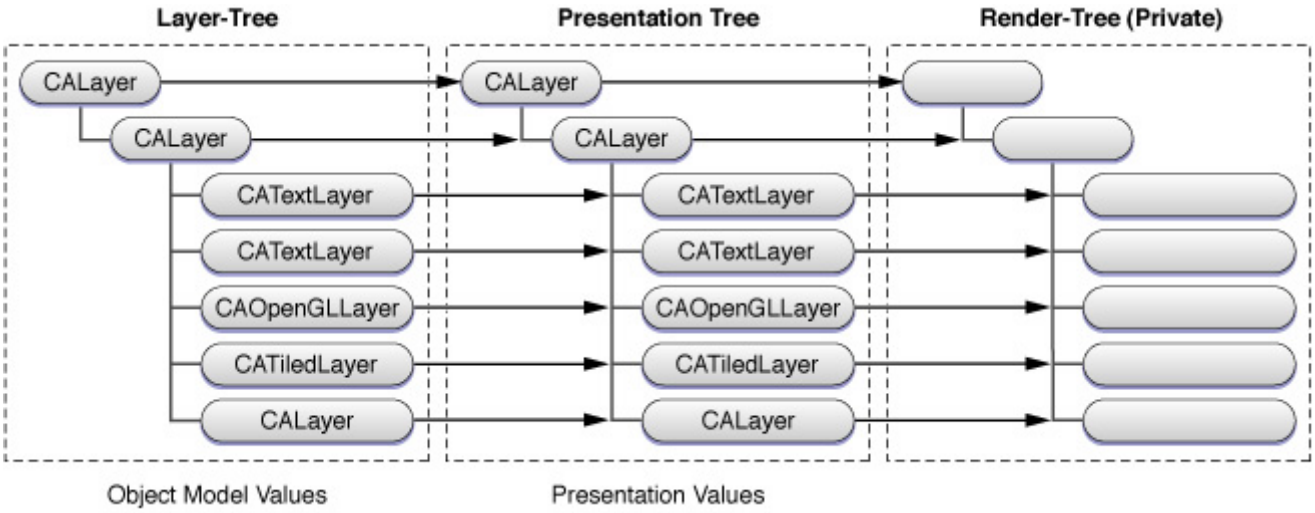
第二章　核心动画渲染框架

虽然核心动画的图层和 Cocoa 的视图在很大程度上没有一定的相似性，但是他们两者最大的区别是，图层不会直接渲染到屏幕上。

在模型-视图-控制器（model-view-controller）概念里面 NSView 和 UIView 是典型的视图部分，但是在核心动画里面图层是模型部分。图层封装了几何、时间、可视化属性，同时它提供了图层现实的内容，但是实际显示的过程则不是由它来完成。

每个可见的图层树由两个相应的树组成:一个是呈现树，一个是渲染树。下图显示在 Mac OS X 上面使用核心动画图层类显示一个图层树的例子。

Figure 1 Core Animation Rendering Architecture



图层树包含每一层的对象模型值。他们就是你设定的图层的属性值。

呈现树包含了当前动画发生时候将要显示的值，例如你要给图层背景颜色设置新的值的时候，它会立即修改图层树里面相应的值。但是在呈现树里面背景颜色值在将要显示给用户的时候才被更新为新值。

渲染树在渲染图层的时候使用呈现树的值。渲染树负责执行独立于应用活动的复杂操作。渲染由一个单独的进程或线程来执行，使其对应用程序的运行循环影响最小。

在原子动画事务执行过程中，你可以查看一个 CALayer 的实例。如果你打算改变当前的动画，要当前显示的状态开始新的动画，这将会对你有非常大的帮助。

第三章　图层的几何和变换

本章介绍图层的几何组成部分，及他们之间的相互关，同时介绍如何变换矩阵可以产生复杂的视觉效果。

3.1　图层的坐标系

图层的坐标系在不同平台上面具有差异性。在 iOS 系统中，默认的坐标系统原点在图层的中心左上角地方，原点向右和向下为正值。在 Mac OS X 系统中，默认的坐标系原点在图层的中心左下角地方，原点向右和向上为正值。坐标系的所有值都是浮点类型。你在任何平台上面创建的图层都采用该平台默认的坐标系。

每个图层定义并维护自己的坐标系，它里面的全部内容都由此相关的坐标系指定位置。该准则同时适应于图层自己的内容和它的任何子图层。因为任何图层定义了它自己的坐标系，CALayer 类提供相应的方法用于从一个图层坐标系的点、矩形、大小值转化为另一个图层坐标系相应的值。

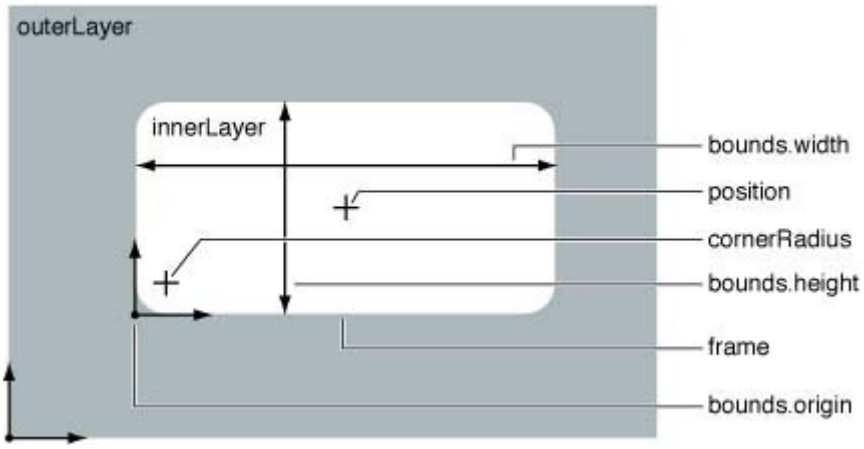
一些基于图层的属性使用单元坐标空间测量它们的值。单元坐标空间指定图层边界的相对值，而不是绝对值。单元坐标空间给定的 x 和 y 的值总是在 0.0 到 1.0 之间。指定一个沿 X 轴的值为 0.0 的点，得到的是图层左边缘的一个点，而指定一个 1.0 的点，则是图层右边缘的一个点。（对 Y 轴而言，如果是在 iOS 系统，则 0.0 对应于顶部的点，而 1.0 则是底部的点，而在 Mac OS X 系统，得到的刚好相反，就如之前提到的坐标系不同一样）。而点（0.5，0.5）则刚好是图层的中心点。

3.2　指定图层的几何

虽然图层和图层树与视图和视图的结构在很多方面具有相似性，但是图层的几何却不同，它更加简单通俗。图层的所有几何属性，包括图层的矩阵变换，都可以隐式和显式动画。

下图显示可以在上下文中指定图层几何的属性：

Figure 1 CALayer geometry properties



图层的 position 属性是一个 CGPoint 的值，它指定图层相当于它父图层的位置，该值基于父图层的坐标系。

图层的 bounds 属性是一个 CGRect 的值，指定图层的大小(bounds.size)和图层的原点(bounds.origin)。当你重写图层的重画方法的时候，bounds 的原点可以作为图形上下文的原点。

图层拥有一个隐式的 frame, 它是 position，bounds，anchorPoint 和 transform 属性的一部分。设置新的 frame 将会相应的改变图层的 position 和 bounds 属性，但是 frame 本身并没有被保存。但是设置新的 frame 时候，bounds 的原点不受干扰，bounds 的大小变为 frame 的大小，即 bounds.size=frame.size。图层的位置被设置为相对于锚点（anchor point）的适合位置。当你设置 frame 的值的时候，它的计算方式和 position、bounds、和 anchorPoint 的属性相关。

图层的 anchorPoint 属性是一个 CGPoint 值，它指定了一个基于图层 bounds 的符合位置坐标系的位置。锚点（anchor point）指定了 bounds 相对于 position 的值，同时也作为变换时候的支点。锚点使用单元空间坐标系表示，（0.0，0.0）点接近图层的原点，而（1.0，1.0）是原点的对角点。改变图层的父图层的变换属性（如果存在的话）将会影响到 anchorPoint 的方向，具体变化取决于父图层坐标系的 Y 轴。

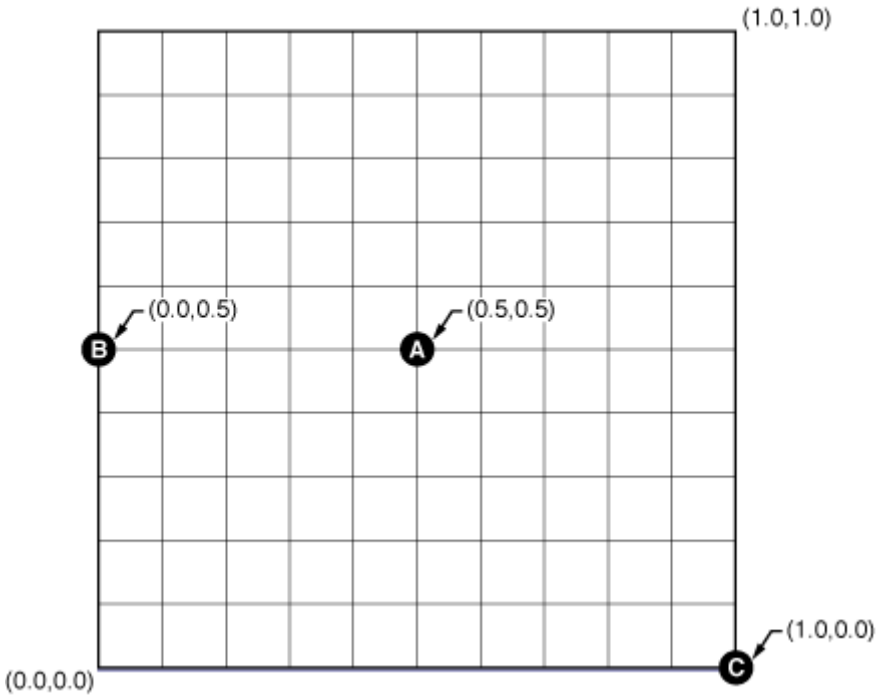
当你设置图层的 frame 属性的时候，position 会根据锚点（anchorPoint）相应的改变，而当你设置图层的 position 属性的时候，bounds 会根据锚点(anchorPoint)做相应的改变。

iOS 注意:*以下示例描述基于 Mac OS X 的图层，它的坐标系原点基于左下角。在 iOS 上面，图层的坐标系原点位于左上角，原点向下和向右为正值。这变化用具体数值显示，而不是概念描*

述。

下图描述了基于锚点的三个示例值：

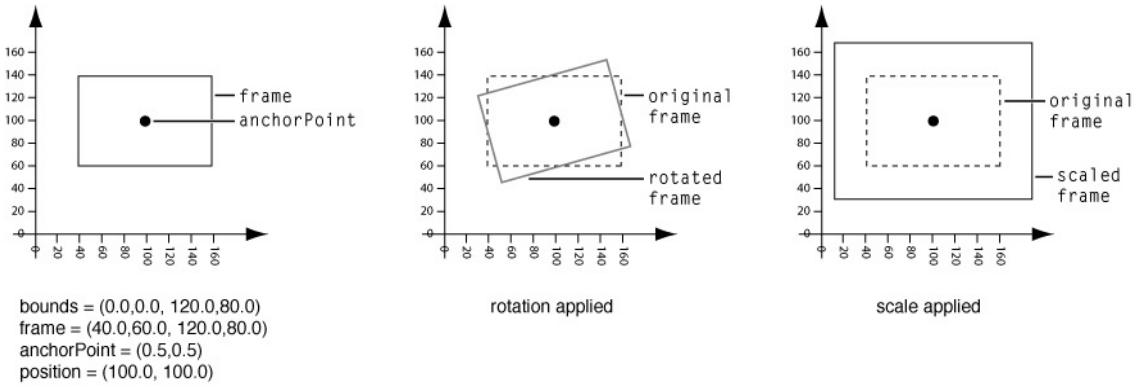
Figure 2 Three anchorPoint values



anchorPoint 默认值是（0.5，0.5），位于图层边界的中心点（如上图显示），B 点把 anchorPoint 设置为（0.0，0.5）。最后 C 点（1.0，0.0）把图层的 position 设置为图层 frame 的右下角。该图适用于 Mac OS X 的图层。在 iOS 系统里面，图层使用不同的坐标系，相应的（0.0，0.0）位于左上角，而（1.0，1.0）位于右下角。

图层的 frame、bounds、position 和 anchorPoint 关系如下图所示：

Figure 3 Layer Origin of (0.5,0.5)



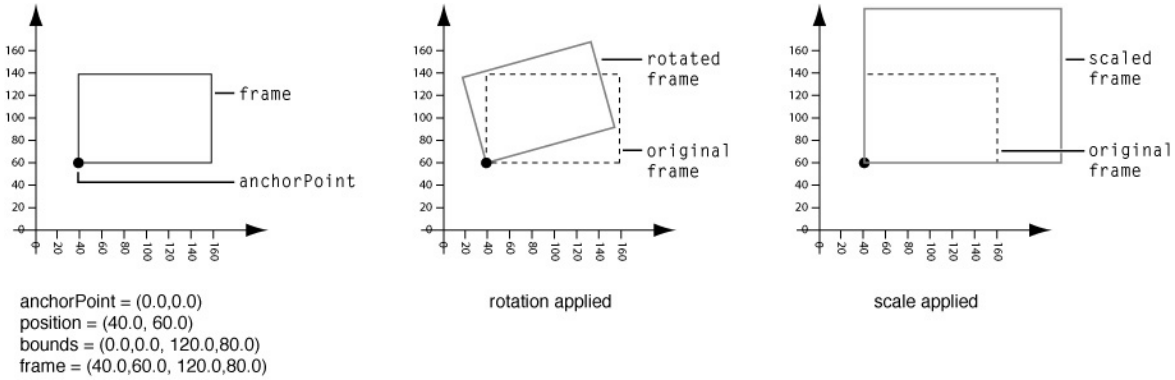
在该示例中，anchorPoint 默认值为(0.5,0.5)，位于图层的中心点。图层的 position 值为（100.0，100.0），bounds 为（0.0，0.0，120，80.0）。通过计算得到图层的 frame

为（40.0，60.0，120.0，80.0）。

如果你新创建一个图层，则只有设置图层的 frame 为（40.0，60.0，120.0，80.0），相应的 position 属性值将会自动设置为（100.0，100.0），而 bounds 会自动设置为(0.0, 0.0, 120.0, 80.0)。

下图显示一个图层具有相同的 frame（如上图），但是在该图中它的 anchorPoint 属性值被设置为(0.0, 0.0)，位于图层的左下角位置。

Figure 4 Layer Origin of (0.0,0.0)



图层的 frame 值同样为(40.0, 60.0, 120.0, 80.0)，bounds 的值不变，但是图层的 position 值已经改变为(40.0, 60.0)。

图层的几何外形和 Cocoa 视图另外一个不同地方是，你可以设置图层的一个边角的半径来把图层显示为圆角。图层的 cornerRadius 属性指定了重绘图层内容，剪切子图层，绘制图层的边界和阴影的时候时候圆角的半径。

图层的 zPosition 属性值指定了该图层位于 Z 轴上面位置，zPosition 用于设置图层相对于图层的同级图层的可视位置。

3.3 图层的几何变换

图层一旦创建，你就可以通过矩阵变换来改变一个图层的几何形状。CATransform3D 的数据结构定义一个同质的三维变换（4x4 CGFloat 值的矩阵），用于图层的旋转，缩放，偏移，歪斜和应用的透视。

图层的两个属性指定了变换矩阵：transform 和 sublayerTransform 属性。图层的 transform 属性指定的矩阵结合图层的 anchorPoint 属性作用于图层和图层的子图层上面。图 3 显示在使用 anchorPoint 默认值(0.5, 0.5)的时候旋转和缩放变换如何影响一

个图层。而图 4 显示了同样的矩阵变换在 anchorPoint 为(0.0, 0.0)的时候如何改变一个图层。图层的 sublayerTransform 属性指定的矩阵只会影响图层的子图层，而不会对图层本身产生影响。

你可以通过以下的任何一个方法改变 CATransform3D 的数据结构：

- 使用 CATransform3D 函数
- 直接修改数据结构的成员
- 使用键-值编码改变键路径

CATransform3DIdentity 是单位矩阵，该矩阵没有缩放、旋转、歪斜、透视。把该矩阵应用到图层上面，会把图层几何属性修改为默认值。

3.3.1 变换函数

使用变换函数可以在核心动画里面在操作矩阵。你可以使用这些函数(如下表)去创建一个矩阵一般后面用于改变图层或者它的子图层的 transform 和 sublayerTransform 属性。变换函数或者直接操或者返回一个 CATransform3D 的数据结构。这可以让你能够构建简单或复杂的转换，以便重复使用。

Table 1 CATransform3D transform functions for translation, rotation, and scaling

Function	Use
CATransform3DMakeTranslation	Returns a transform that translates by '(tx, ty, tz)'. t' = [1 0 0 0; 0 1 0 0; 0 0 1 0; tx ty tz 1].
CATransform3DTranslate	Translate 't' by '(tx, ty, tz)' and return the result: * t' = translate(tx, ty, tz) * t.
CATransform3DMakeScale	Returns a transform that scales by `(sx, sy, sz)': * t' = [sx 0 0 0; 0 sy 0 0; 0 0 sz 0; 0 0 0 1].
CATransform3DScale	Scale 't' by '(sx, sy, sz)' and return the result: * t' = scale(sx, sy, sz) * t.
CATransform3DMakeRotation	Returns a transform that rotates by 'angle' radians about the vector '(x, y, z)'. If the vector has length zero the identity transform is returned.
CATransform3DRotate	Rotate 't' by 'angle' radians about the vector '(x, y, z)' and return the result. t' = rotation(angle, x, y, z) * t.

旋转的单位采用弧度(radians)，而不是角度（degress）。以下两个函数，你可以在弧度和角度之间切换。

```
CGFloat DegreesToRadians(CGFloat degrees) {return degrees * M_PI / 180;};
```

```
CGFloat RadiansToDegrees(CGFloat radians) {return radians * 180 / M_PI;};
```

核心动画 提供了用于转换矩阵的变换函数 CATransform3DInvert。一般是用反转点内转化对象提供反向转换。当你需要恢复一个已经被变换了的矩阵的时候，反转将会非常有帮助。反转矩阵乘以逆矩阵值，结果是原始值。

变换函数同时允许你把 CATransform3D 矩阵转化为 CGAffineTransform（仿射）矩阵，前提是 CATransform3D 矩阵采用如下表示方法。

Table 2 CATransform3D transform functions for CGAffineTransform conversion

Function	Use
CATransform3DMakeAffineTransform	Returns a `CATransform3D` with the same effect as the passed affine transform.
CATransform3DIsAffine	Returns `YES` if the passed`CATransform3D` can be exactly represented an affine transform.
CATransform3DGetAffineTransform	Returns the affine transform represented by the passed`CATransform3D`.

变换函数同时提供了可以比较一个变换矩阵是否是单位矩阵，或者两个矩阵是否相等。

Table 3 CATransform3D transform functions for testing equality

Function	Use
CATransform3DIsIdentity	Returns `YES` if the transform is the identity transform.
CATransform3DEqualToTransform	Returns `YES` if the two transforms are exactly equal..

3.3.2 修改变换的数据结构

你可以修改 CATransform3D 的数据结构的元素为任何其他你想要的数据值。清单 1 包含了 CATransform3D 数据结构的定义，结构的成员都在其相应的矩阵位置。

Listing 1 CATransform3D structure

```
struct CATransform3D
{
  CGFloat m11, m12, m13, m14;
  CGFloat m21, m22, m23, m24;
  CGFloat m31, m32, m33, m34;
  CGFloat m41, m42, m43, m44;
```

```
};


typedef struct CATransform3D CATransform3D;
```

清单 2 中的示例说明了如何配置一个 CATransform3D 一个角度变换。

Listing 2 Modifying the CATransform3D data structure directly

```
CATransform3D aTransform = CATransform3DIdentity;
// the value of zDistance affects the sharpness of the transform.
zDistance = 850;
aTransform.m34 = 1.0 / -zDistance;
```

3.3.3 通过键值路径修改变换

核心动画扩展了键-值编码协议，允许通过关键路径获取和设置一个图层的 CATransform3D 矩阵的值。表 4 描述了图层的 transform 和 sublayerTransform 属性的相应关键路径。

Table 4 CATransform3D key paths

Field Key Path	Description
`rotation.x`	The rotation, in radians, in the x axis.
`rotation.y`	The rotation, in radians, in the y axis.
`rotation.z`	The rotation, in radians, in the z axis.
`rotation`	The rotation, in radians, in the z axis. This is identical to setting the `rotation.z` field.
`scale.x`	Scale factor for the x axis.
`scale.y`	Scale factor for the y axis.
`scale.z`	Scale factor for the z axis.
`scale`	Average of all three scale factors.
`translation.x`	Translate in the x axis.
`translation.y`	Translate in the y axis.
`translation.z`	Translate in the z axis.
`translation`	Translate in the x and y axis. Value is an NSSize or CGSize.

你不可以通过 Objective-C 2.0 的属性来设置结构域的值，比如下面的代码将会

无法正常运行:

```
myLayer.transform.rotation.x=0;
```

替换的办法是，你必须通过 setValue:forKeyPath:或者 valueForKeyPath:方法，具体如下:

```
[myLayer setValue:[NSNumber numberWithInt:0] forKeyPath:@"transform.rotation.x"];
```

第四章　图层树的层次结构

图层不但给自己提供可视化的内容和管理动画，而且充当了其他图层的容器类，构建图层层次结构。

本章介绍了图层层次结构，以及如何操纵该图层层次结构。

4.1　什么是图层树的层次结构

图层树是核心动画里面类似 Cocoa 视图的层次结构。比如一个 NSView 或者 UIView 的实例拥有父视图(superview)和子视图(subview)，一个核心动画的图层拥有父图层(suplayer)和子图层(sublayer)。图层树和视图结构一样提供了很多便利：

- 复杂的接口可以由简单的图层来组合，避免了硕大和复杂的继承化子类。图层非常合适于这种堆叠方式来合成复杂的功能。
- 每个图层定义了一个基于其父图层的坐标系的坐标系。当一个图层变换的时候，它的子图层同样变换。
- 一个动态的图层树，可以在程序运行的时候重新设置。图层可以创建并添加为一个图层的第一个子图层，然后从其他图层的图层树上面删除。

4.2　在视图里面显示图层

核心动画不提供在一个窗口(window)实际显示图层的手段，它们必须通过视图来托管。当视图和图层一起的时候，视图为图层提供了底层的事件处理，而图层为视图提供了显示的内容。

iOS 上面的视图系统直接建立在核心动画的图层上面。每个 UIView 的实例会自动的创建一个 CALayer 类的实例，然后把该实例赋值给视图的 layer 属性。你可以在需要的时候向视图的图层里面添加子图层。

在 Mac OS X，您必须配置一个 NSView 的实例，通过这样一种方式才可以让它托管图层。为了显示图层树的根图层，你可以设置一个视图的图层和配置视图以便使用图层。具体如表 2.

Listing 1 Inserting a layer into a view

```
// theView is an existing view in a window
// theRootLayer is the root layer of a layer tree

[theView setLayer: theRootLayer];
[theView setWantsLayer:YES];
```

4.3 从图层结构里面添加和删除图层

简单的实例化一个图层并不意味已经把它插入了一个图层树。而是通过以下表 1 的方法来实现从图层树里面添加、插入、替换和删除图层。

Table 1 Layer-tree management methods.

Method	Result
`addSublayer:`	Appends the layer to the receiver's sublayers array.
`insertSublayer:atIndex:`	Inserts the layer as a sublayer of the receiver at the specified index.
`insertSublayer:below:`	Inserts the layer into the receiver's sublayers array, below the specified sublayer.
`insertSublayer:above:`	Inserts the layer into the receiver's sublayers array, above the specified sublayer.
`removeFromSuperlayer`	Removes the receiver from the sublayers array or mask property of the receiver's superlayer.
`replaceSublayer:with:`	Replaces the layer in the receiver's sublayers array with the specified new layer.

你也可以通过使用一个图层的数组来设置图层的子图层，甚至可以扩展设置父图层的 sublayers 属性。当把图层的 sublayers 属性设置了一个图层的数组值的时候，你必须保证数组里面每个图层的父图层已经被设置为 nil。

默认情况下从一个可视化图层树里面插入和删除图层将会触发动画。当把一个图层添加为子图层的时候，将会触发父图层返回标识符为 kCAOnOrderIn 动画。当从图层的子图层里面删除一个图层的时候，将会触发父图层返回一个标识符为 kCAOnOrderOut 的动画。当替换图层的子图层里面的一个图层的时候，将会触发父图层返回一个标识符为 KCATransition 的动画。当你操作图层树的时候，你可以禁用动画或者改变使用任何标识符的动画。

4.4 图层的位置调整和大小改变

图层创建以后，你可以通过改变图层的几何属性:frame、bounds、position、anchorPoint 和 zPosition 来编程式移动和改变图层大小。

如果一个图层的属性 needsDisplayOnBoundsChange 被设置为 YES 的时候，当图层的 bounds 属性改变的时候，图层的内容将会被重新缓存起来。默认情况下图层的 needsDisplayOnBoundsChange 属性值为 NO。

默认情况下，设置图层的属性 frame、bounds、position、anchorPoint 和 zPosition 属性将会导致图层动画显示新值。

4.4.1 自动调整图层大小

CALayer 提供了一个机制，在父图层被移动或者改变大小的时候，子图层可以自动的跟着移动和调整大小。在很多情况下简单的配置一个图层的自动调整掩码（autoresizing mask）可以适当的适应程序的行为。

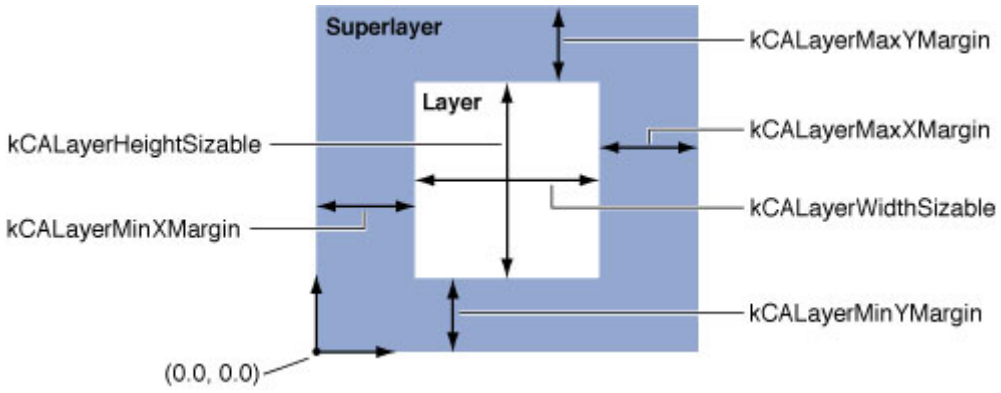
一个图层的自动调整掩码可以通过指定 CAAutoresizingMask 的常量结合或运算（OR）所得的结果赋值给图层的 autoresizingMask 属性值。表 2 列举了掩码常量和这些掩码如何影响图层的大小调整行为。

Table 2 Autoresizing mask values and descriptions

Autoresizing Mask	Description
kCALayerHeightSizable	If set, the layer's height changes proportionally to the change in the superlayer's height. Otherwise, the layer's height does not change relative to the superlayer's height.
kCALayerWidthSizable	If set, the layer's width changes proportionally to the change in the superlayer's width. Otherwise, the layer's width does not change relative to the superlayer's width.
kCALayerMinXMargin	If set, the layer's left edge is repositioned proportionally to the change in the superlayer's width. Otherwise, the layer's left edge remains in the same position relative to the superlayer's left edge.
kCALayerMaxXMargin	If set, the layer's right edge is repositioned proportionally to the change in the superlayer's width. Otherwise, the layer's right edge remains in the same position relative to the superlayer.
kCALayerMaxYMargin	If set, the layer's top edge is repositioned proportionally to the change in the superlayer's height. Otherwise, the layer's top edge remains in the same position relative to the superlayer.
kCALayerMinYMargin	If set, the layer's bottom edge is repositioned proportional to the change in the superlayer's height. Otherwise, the layer's bottom edge remains in the same position relative to the superlayer.

例如，为了把保持图层位于它父图层的相对左下角位置，你可以使用 kCALayerMaxXMargin | kCALayerMaxYMargin。当沿着一个轴具有多个方向被设置为适应可变的时候，那么调整大小的尺寸为使其均匀分布的值。图 1 提供了一个常量值的位置的图形表示。

Figure 1 Layer autoresizing mask constants



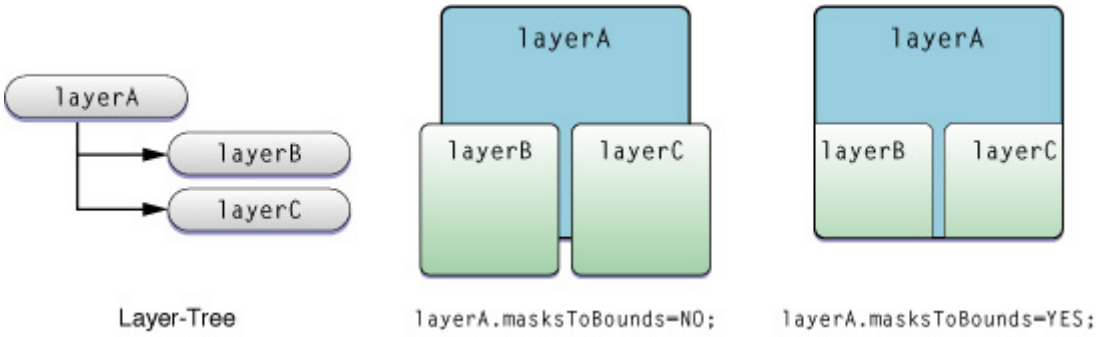
当这些常量里面的任何一个被省略的时候，图层的布局在这个方向上值是固定的。当一个常量包含在图层的自动调整掩码里面的时候，该方向上的图层的布局值是适应可变的。

CALayer 的子类可以重写函数 resizeSublayersWithOldSize：和 resizeWithOldSuperlayerSize：来定制化的自动调整图层大小的行为。图层的函数 resizeSublayersWithOldSize：将会在 bounds 属性被修改的时候自动的触发执行，同时发送一个消息 resizeWithOldSuperlayerSize：给图层的每个子图层。图层的每个子图层根据自动调整掩码的属性来比较就的边界值和新的边界值来调整它的位置和大小。

4.5 裁剪子图层

在 Cocoa 的视图里面，当子视图超出父视图的边界的时候，视图将会被裁剪以适应父视图的大小。图层去掉了这个限制，允许子层全部显示，无论自己相对于父层位置如何。图层的 masksToBounds 属性决定了是否子图层是否相对父图层裁剪。该属性 masksToBounds 的默认值为 NO，即防止子图层被相对于父图层裁剪。表 2 显示了当设置图层的 masksToBounds 属性导致的结果，和它如何影响 layerB 和 layerC 的显示。

Figure 2 Example Values of the masksToBounds property



第五章　提供图层内容

当我们使用 Cocoa 的视图的时候，我们必须继承 NSView 或者 UIView 并且重载函数 drawRect:来显示任何内容。但是 CALayer 实例可以直接使用，而无需继承子类。因为 CALayer 是一个键-值编码兼容的容器类，你可以在实例里面存储任意值，所以子类实例化完全可以避免。

5.1　给CALayer提供内容

你可以通过以下任何一种方法指定 CALayer 实例的内容：

- 使用包含图片内容的 CGImageRef 来显式的设置图层的 contents 的属性。
- 指定一个委托，它提供或者重绘内容。
- 继承 CALayer 类重载显示的函数。

5.1.1　设置contents属性

图层的图片内容可以通过指定 contents 属性的值为 CGImageRef。当图层被创建的时候或者在任何其他时候，这个操作可以在其他实体上面完成（如表 3 所示）。

Listing 1 Setting a layer's contents property

```
CALayer *theLayer;


// create the layer and set the bounds and position
theLayer=[CALayer layer];
theLayer.position=CGPointMake(50.0f,50.0f);
theLayer.bounds=CGRectMake(0.0f,0.0f,100.0f,100.0f);


// set the contents property to a CGImageRef
// specified by theImage (loaded elsewhere)
theLayer.contents=theImage;
```

5.1.2　通过委托提供内容

你可以绘制图层的内容，或更好的封装图层的内容图片，通过创建一个委托类实

现下列方法之一：

displayLayer:或 drawLayer:inContext:

实现委托重绘的方法并不意味会自动的触发图层使用实现的方法来重绘内容。而是你要显式的告诉一个图层实例来重新缓存内容，通过发送以下任何一个方法 setNeedsDisplay 或者 setNeedsDisplayInRect: 的消息，或者把图层的 needsDisplayOnBoundsChange 属性值设置为 YES。

通过委托实现方法 displayLayer:可以根据特定的图层决定显示什么图片，还可以更加需要设置图层的 contents 属性值。下面的例子是“图层的坐标系”部分的，它实现 displayerLayer:方法根据 state 的值设置 theLayer 的 contents 属性。子类不需要存储 state 的值，因为 CALayer 的实例是一个键-值编码容器。

Listing 2 Example implementation of the delegate method displayLayer:

```
- (void)displayLayer:(CALayer *)theLayer
{
    // check the value of the layer's state key
    if ([[theLayer valueForKey:@"state"] boolValue])
    {
        // display the yes image
        theLayer.contents=[someHelperObject loadStateYesImage];
    }
    else {
        // display the no image
        theLayer.contents=[someHelperObject loadStateNoImage];
    }
}
```

如果你必须重绘图层的内容，而不是通过加载图片，那你需要实现 drawLayer:inContext:方法。通过委托可以决定哪些内容是需要的并使用 CGContextRef 来重绘内容。

下面的例子是“指定图层的几何”部分内容，它实现了 drawLayer:inContext:方法使用 lineWidth 键值来重绘一个路径(path)，返回 therLayer。

Listing 3 Example implementation of the delegate method drawLayer:inContext:

```
- (void)drawLayer:(CALayer *)theLayer
        inContext:(CGContextRef)theContext
{
    CGMutablePathRef thePath = CGPathCreateMutable();

    CGPathMoveToPoint(thePath,NULL,15.0f,15.f);
    CGPathAddCurveToPoint(thePath,
                          NULL,
                          15.f,250.0f,
                          295.0f,250.0f,
                          295.0f,15.0f);

    CGContextBeginPath(theContext);
    CGContextAddPath(theContext, thePath );

    CGContextSetLineWidth(theContext,
                          [[theLayer valueForKey:@"lineWidth"] floatValue]);
    CGContextStrokePath(theContext);

    // release the path
    CFRelease(thePath);
}
```

5.1.3 通过子类提供图层的内容

虽然通常情况不需要这样做，但是你仍可以继承 CALayer 直接重载重绘和显示方法。这个通常发生在你的图层需要定制行为而委托又无法满足需求的时候。

子类可以重载 CALayer 的显示方法，设置图层的内容为适当的图片。下面的例子是“变换图层的几何”部分的内容，它提供了和“图层的坐标系”例子相同的功能。不同的是子类定义 state 为实例的属性，而不是根据 CALayer 的键-值编码容器获取。

Listing 4 Example override of the CALayer display method

```
- (void)drawInContext:(CGContextRef)theContext
{
    CGMutablePathRef thePath = CGPathCreateMutable();

    CGPathMoveToPoint(thePath,NULL,15.0f,15.f);
    CGPathAddCurveToPoint(thePath,
                          NULL,
                          15.f,250.0f,
                          295.0f,250.0f,
                          295.0f,15.0f);

    CGContextBeginPath(theContext);
    CGContextAddPath(theContext, thePath );

    CGContextSetLineWidth(theContext,
                          self.lineWidth);
    CGContextSetStrokeColorWithColor(theContext,
                                self.lineColor);
    CGContextStrokePath(theContext);
    CFRelease(thePath);
}
```

CALayer 子类可以通过重载 drawInContext:绘制图层的内容到一个图形上下文。下面的例子是“修改变换的数据结构”的内容，它和“指定图层的几何”里面实现委托的办法一样产生相同的图片内容。唯一的不同的是实现委托里面的 lineWidth 和 lineColor 现在是子类实例的属性。

Listing 5 Example override of the CALayer drawInContext: method

```
- (void)drawInContext:(CGContextRef)theContext
{
    CGMutablePathRef thePath = CGPathCreateMutable();
```

```
    CGPathMoveToPoint(thePath,NULL,15.0f,15.f);
    CGPathAddCurveToPoint(thePath,
                          NULL,
                          15.f,250.0f,
                          295.0f,250.0f,
                          295.0f,15.0f);

    CGContextBeginPath(theContext);
    CGContextAddPath(theContext, thePath );

    CGContextSetLineWidth(theContext,
                          self.lineWidth);
    CGContextSetStrokeColorWithColor(theContext,
                                     self.lineColor);
    CGContextStrokePath(theContext);
    CFRelease(thePath);
}
```

继承 CALayer 并且实现其中的重绘方法并不意味重绘会自动发生。你必须显式的促使实例重新缓存其内容，可以通过发送以下任何一个方法 setNeedsDisplay 或 setNeedsDisplayInRect:的消息，亦或者设置图层的 needsDisplaOnBoundsChange 属性为 YES。

5.2 修改图层内容的位置

CALayer 的属性 contentsGravity 允许你在图层的边界内容修改图层的 contents 图片的位置或者伸缩值。默认情况下，内容的图像完全填充层的边界，忽视自然的图像宽高比。

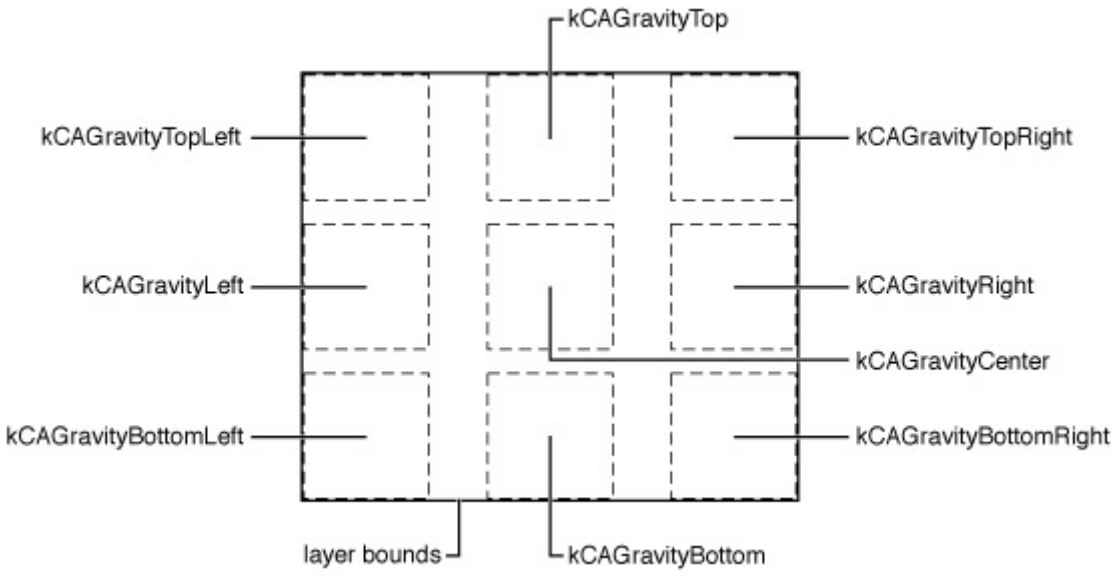
使用 contentsGravity 位置常量，你可以指定图片位于图层任何一个边界，比如位于图层的角落，或者图层边界的中心。然而当你使用位置常量的时候，contentsCenter 属性会被忽略。表 1 列举了位置常量和他们相应的位置。

Table 1 Positioning constants for a layer's contentsGravity property

Position constant	Description
kCAGravityTopLeft	Positions the content image in the top left corner of the layer.
kCAGravityTop	Positions the content image horizontally centered along the top edge of the layer.
kCAGravityTopRight	Positions the content image in the top right corner of the layer.
kCAGravityLeft	Positions the content image vertically centered on the left edge of the layer.
kCAGravityCenter	Positions the content image at the center of the layer.
kCAGravityRight	Positions the content image vertically centered on the right edge of the layer.
kCAGravityBottomLeft	Positions the content image in the bottom left corner of the layer.
kCAGravityBottom	Positions the content image centered along the bottom edge of the layer.
kCAGravityBottomRight	Positions the content image in the top right corner of the layer.

"图层的坐标系"标识了所支持的内容位置和他们相应的常量。

Figure 1 Position constants for a layer's contentsGravity property



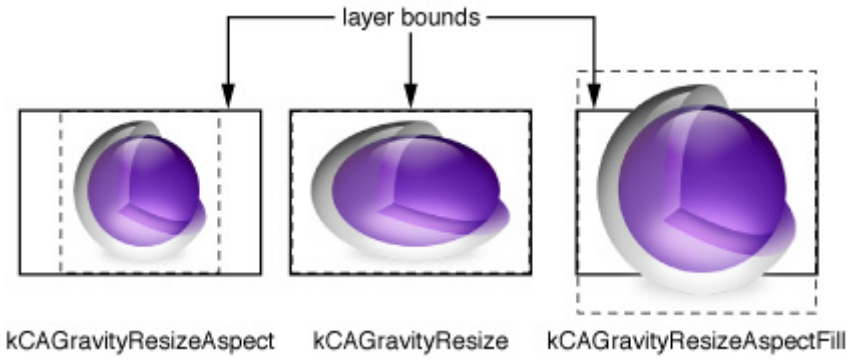
通过设置 contentsGravity 属性为其他一个常量(如表 2 所示)。图层的内容图片可以被向上或者向下拉伸， 仅当使用其他任何一个调整大小的常量的时候，contentsCenter 属性才会对内容图片起作用。

Table 2 Scaling Constants For A Layer's `contentsGravity` Property

Scaling constant	Description
kCAGravityResize	Resize the content image to completely fill the layer bounds, potentially ignoring the natural aspect of the content. This is the default.
kCAGravityResizeAspect	Resize the content image to scale such that it is displayed as large as possible within the layer bounds, yet still retains its natural aspect.
kCAGravityResizeAspectFill	Resize the content image to scale such that it is displayed filling the layer bounds, yet retaining its natural aspect. This may cause the content to extend outside the layer bounds.

"变换图层的几何"演示了如何使用调整大小的模式来调整一个正方形图像的大小让其适应图层的方形边界。

Figure 2 Scaling constants for a layer's contentsGravity property



注意：使用任何常量 kCAGravityResize、kCAGravityResizeAspect 和 kCAGravityResizeAspectFill 和表 1 中的重心位置常量无关。图层的内容将会填充整个边界，所以使用这些常量无法改变图层内容的位置。

第六章　动画

动画是当今用户界面的关键因素。当使用核心动画的时候，动画是自动完成的。没有动画的循环和计数器。你的应用程序不负负责重绘，也不负责跟踪动画的当前状态。动画在独立线程里面自动执行，没有和你的应用程序交互。

本章提供了对动画类的概览，和介绍如何创建隐式的和显式的动画。

6.1　动画类和时序

核心动画提供了一套你可以在你应用程序里面使用的动画类的表现：

- CABasicAnimation 提供了在图层的属性值间简单的插入。
- CAKeyframeAnimation 提供支持关键帧动画。你指定动画的一个图层属性的关键路径，一个表示在动画的每个阶段的价值的数组，还有一个关键帧时间的数组和时间函数。
- CATransition 提供了一个影响整个图层的内容过渡效果。在动画显示过程中采用淡出（fade）、推出(push)、显露(reveal)图层的内容。 常用的过渡效果可以通过提供你自己定制的核心图像滤镜来扩展。

除了要指定显示的动画类型，你还必须指定动画的间隔、它的速度(它的插值如何分布在整个动画过程)、动画循环时候的循环次数、动画周期完成的时候是否自动的反转、还有动画结束的时候它的可视化状态。动画类和 CAMediaTiming 协议提供所有这些功能甚至更多的功能。

CAAnimation、它的子类、时序协议被核心动画和Cocoa Animation Proxy功能共享。这些类将会在“动画类型和时序编程指南（*Animation Types and Timing Programming Guide*）”里面详细介绍。

6.2　隐式动画

核心动画的隐式动画模型假定所有动画图层属性的变化应该是渐进的和异步的。动态的动画场景可以在没有显式的动画图层时候实现。改变可动画显示的图层的属性将会导致图层隐式把图层从旧的值动画显示为新的值。虽然动画是持续的，但是设置新的目标值时会导致图层从当前状态动画过渡到新的目标值。

清单 1 显示了如果简单的触发一个隐式的动画，把图层从当前位置动画改变到新的位置。

Listing 1 Implicitly animating a layer's position property

```
// assume that the layer is current positioned at (100.0,100.0)

theLayer.position=CGPointMake(500.0,500.0);
```

你可以隐式的一次动画改变图层的一个或者多个属性。你还可以隐式的同时动画改变多个图层。清单 2 的代码实现了 4 个同时触发的隐式动画。

Listing 2 Implicitly animating multiple properties of multiple layers

```
// animate theLayer's opacity to 0 while moving it

// further away in the layer

theLayer.opacity=0.0;

theLayer.zPosition=-100;


// animate anotherLayer's opacity to 1

//  while moving it closer in the layer

anotherLayer.opacity=1.0;

anotherLayer.zPosition=100.0;
```

隐式动画使用动画属性中默认指定的动画时间，除非该时间已经被隐式或者显式的修改过。阅读“重载覆盖隐式动画时间”获取更多详情。

6.3 显式动画

核心动画同时提供了一个显式的动画模型。该显式动画模型需要你创建一个动画对象，并设置开始和结束的值。显式动画不会开始执行，直到你把该动画应用到某个图层上面。清单 3 中的代码片段创建了一个显式动画，它实现一个层的不透明度从完全不透明过渡到完全透明的，3 秒后返回重新执行。动画没有开始，直到它被添加到某一图层层。

Listing 3 Explicit animation

```
CABasicAnimation *theAnimation;


theAnimation=[CABasicAnimation animationWithKeyPath:@"opacity"];
```

```
theAnimation.duration=3.0;
theAnimation.repeatCount=2;
theAnimation.autoreverses=YES;
theAnimation.fromValue=[NSNumber numberWithFloat:1.0];
theAnimation.toValue=[NSNumber numberWithFloat:0.0];
[theLayer addAnimation:theAnimation forKey:@"animateOpacity"];
```

显式动画对于创建连续执行的动画非常有帮助。清单 4 显示了如何创建一个显式动画，把一个 CoreImage 滤镜应用到图层上面，动画显示其强度。这将导致“选择的图层”跳动，吸引用户的注意力。

Listing 4 Continuous explicit animation example

```
// The selection layer will pulse continuously.
// This is accomplished by setting a bloom filter on the layer

// create the filter and set its default values
CIFilter *filter = [CIFilter filterWithName:@"CIBloom"];
[filter setDefaults];
[filter setValue:[NSNumber numberWithFloat:5.0] forKey:@"inputRadius"];

// name the filter so we can use the keypath to animate the inputIntensity
// attribute of the filter
[filter setName:@"pulseFilter"];

// set the filter to the selection layer's filters
[selectionLayer setFilters:[NSArray arrayWithObject:filter]];

// create the animation that will handle the pulsing.
CABasicAnimation* pulseAnimation = [CABasicAnimation animation];

// the attribute we want to animate is the inputIntensity
// of the pulseFilter
pulseAnimation.keyPath = @"filters.pulseFilter.inputIntensity";
```

```
// we want it to animate from the value 0 to 1
pulseAnimation.fromValue = [NSNumber numberWithFloat: 0.0];
pulseAnimation.toValue = [NSNumber numberWithFloat: 1.5];

// over a one second duration, and run an infinite
// number of times
pulseAnimation.duration = 1.0;
pulseAnimation.repeatCount = HUGE VALF;

// we want it to fade on, and fade off, so it needs to
// automatically autoreverse.. this causes the intensity
// input to go from 0 to 1 to 0
pulseAnimation.autoreverses = YES;

// use a timing curve of easy in, easy out..
pulseAnimation.timingFunction = [CAMediaTimingFunction functionWithName:
kCAMediaTimingFunctionEaseInEaseOut];

// add the animation to the selection layer. This causes
// it to begin animating. We'll use pulseAnimation as the
// animation key name
[selectionLayer addAnimation:pulseAnimation forKey:@"pulseAnimation"];
```

6.4 开始和结束显式动画

你可以发送 addAnimation:forKey:消息给目标图层来开始一个显式动画，把动画和标识符作为参数。一旦把动画添加到目标图层，动画将会一直执行直到动画完成，或者动画被从图层上面移除。把动画添加到图层时添加的标识符，同样也可以在停止动画的时候使用，通过调用 removeAnimationForKey:。你可以通过给图层发送一个 removeAllAnimations 消息来停止图层所有的动画。

第七章　图层的行为

图层的行为在以下情况发生的时候被触发：从图层树里面插入或者删除一个图层，图层的属性值被修改了，或者程序显式要求。通常情况下，行为触发器是动画显示的结果所在。

7.1　行为对象的角色

一个行为对象是一个通过 CAAction 协议响应行为标识符的对象。行为标识符使用标准圆点分隔的关键路径来命名。图层负责把行为标识符映射到特定的行为对象。当一个特定标识符的行为对象被确定的时候，它会发送一个 CAAction 协议定义的消息。

CALayer 类提供了默认的 CAAnimation 的行为对象实例，一个兼容类所有动画层属性 CAAction 协议。表 1 中 CALayer 同样定义了以下没有直接对应到属性的行为触发器和他们的行为标识符。

Table 1 Action triggers and their corresponding identifiers

Trigger	Action identifier
A layer is inserted into a visible layer-tree, or the`hidden` property is set to `NO`.	The action identifier constant`kCAOnOrderIn`.
A layer is removed from a visible layer-tree, or the`hidden` property is set to `YES`.	The action identifier constant`kCAOnOrderOut`.
A layer replaces an existing layer in a visible layer tree using `replaceSublayer:with:`.	The action identifier constant`kCATransition`.

7.2　已定义搜索模式的行为键值

当一个行为触发器发生的时候，图层的 actionForKey:方法被调用。此方法返回一个行为对象，对应的标识符作为参数，或如果行为对象不存在的话返回 nil。

当 CALayer 为一个标识符实现的 actionForKey:方法被调用的时候，以下的搜索模式将会被用到：

1. 如果一个图层有委托，那方法 actionForLayer:forKey:的实现将会被调用，把图层和行为标识符作为参数。委托的 actionForLayer:forKey:的实现需要响应如下：

- 返回一个行为标识符对应的行为对象。
- 返回 nil，当无法处理行为标识符的时候。
- 返回 NSNull，当无法处理行为标识符，而且搜索需要被终止的时候。

2. 图层的 actions 字典被搜索以便找到一个和行为标识符对应的对象。
3. 图层的 style 属性被搜索以便找到一个包含行为标识符的 actions 字典。
4. 图层类发生一个 defaultActionForKey:的消息。它将会返回一个和标识符对应的行为对象，如果不存在的话则返回 nil。

7.3 采用CAAction协议

CAAction 协议定义了行为对象如何被调用。实现 CAAction 协议的类包含一个方法 runActionForKey:object:arguments:。

当行为对象收到一个 runActionForKey:object:arguments:的消息时，行为标识符、行为发生所在的图层、额外的参数字典会被作为参数传递给方法。

通常行为对象是 CAAnimation 的子类实例，它实现了 CAAction 协议。然而你也可以返回任何实现了 CAAction 协议的类对象。当实例收到 runActionForKey:object:arguments:的消息时，它需要执行相应的行为。

当 CAAnimation 实例受到消息 runActionForKey:object:arguments:的时候，它把自己添加到图层的动力里面，触发动画的执行（查看清单 1）。

Listing 1 runActionForKey:object:arguments: implementation that initiates an animation

```
- (void)runActionForKey:(NSString *)key
                 object:(id)anObject
              arguments:(NSDictionary *)dict
{
    [(CALayer *)anObject addAnimation:self forKey:key];
}
```

7.4 重载隐式动画

你可以为行为标识符提供隐式的动画，通过插入一个 CAAnimation 的实例到 style 字典里面的 actions 的字典里面，通过实现委托方法 actionForLayer:forKey:或者继承

图层类并重载 defaultActionForKey:方法返回一个相应的行为对象。

清单 2 的示例通过委托替换 contents 属性的隐式动画。

Listing 2 Implied animation for the contents property

```
- (id<CAAction>)actionForLayer:(CALayer *)theLayer

                    forKey:(NSString *)theKey

{

    CATransition *theAnimation=nil;



    if ([theKey isEqualToString:@"contents"])

    {



        theAnimation = [[CATransition alloc] init];

        theAnimation.duration = 1.0;

        theAnimation.timingFunction = [CAMediaTimingFunction
functionWithName:kCAMediaTimingFunctionEaseIn];

        theAnimation.type = kCATransitionPush;

        theAnimation.subtype = kCATransitionFromRight;

    }



    return theAnimation;

}
```

清单 3 的示例使用 actions 字典模式禁用 sublayers 属性的默认动画。

Listing 3 Implied animation for the sublayers property

```
// get a mutable version of the current actions dictionary

NSMutableDictionary *customActions=[NSMutableDictionary
dictionaryWithDictionary:[theLayer actions]];



// add the new action for sublayers

[customActions setObject:[NSNull null] forKey:@"sublayers"];



// set theLayer actions to the updated dictionary

theLayer.actions=customActions;
```

7.5 暂时禁用行为

默认情况下，你任何时候改变一个可动画显示的属性时，相应的动画将会伴随发生。

在修改图层属性的时候，你可以通过使用事务暂时禁用行为。查看“暂时禁用图层的行为”部分来获取更多信息。

第八章　事务

图层的每个改变都是事务的一部分。CATransaction 是核心动画类，它负责成批的把多个图层树的修改作为一个原子更新到渲染树。

本章介绍了核心动画支持的两种事务。隐式事务和显式事务。

8.1　隐式事务

当图层树被没有获得事务的线程修改的时候将会自动创建隐式事务，当线程的运行循环（run-loop）执行下次迭代的时候将会自动提交事务。

清单 1 的示例修改图层的 opacity, zPosition 和 position 数字，依赖隐式事务来确保动画同时一起发生。

Listing 1 Animation using an implicit transaction

```
theLayer.opacity=0.0;

theLayer.zPosition=-200;

thelayer.position=CGPointMake(0.0,0.0);
```

重要:*当在一个没有运行循环(runloop)的线程修改图层的属性的时候，你必须使用显式的事务。*

8.2　显式事务

在你修改图层树之前，可以通过给 CATransaction 类发送一个 begin 消息来创建一个显式事务，修改完成之后发送 comit 消息。显式事务在同时设置多个图层的属性的时候（例如当布局多个图层的时候），暂时的禁用图层的行为，或者暂时修改动画的时间的时候非常有用。

8.2.1　暂时禁用图层的行为

你可以在修改图层属性值的时候通过设置事务的 kCATransactionDisableActions 值为 YES 来暂时禁用图层的行为。在事务范围所作的任何更改也不会因此而发生的动画。清单 2 显示了一个示例，当把 aLayer 从可视化图层树移除的时候禁用淡出动画。

Listing 2 Temporarily disabling a layer's actions

```
[CATransaction begin];
[CATransaction setValue:(id)kCFBooleanTrue
                 forKey:kCATransactionDisableActions];
[aLayer removeFromSuperlayer];
[CATransaction commit];
```

8.2.2 重载隐式动画的时间

你可以暂时改变响应改变图层属性的动画的时间，通过设置事务的 kCATransactionAnimationDuration 键的值为新的时间。事务范围内所产生的任何动画都会使用该新设置的时间值而不是他们原有的值。清单 3 显示了一个示例，把动画的发生时间改为 10 秒而不是 zPosition 和 opacity 所指定的动画的默认时间。

Listing 3 Overriding the animation duration

```
[CATransaction begin];
[CATransaction setValue:[NSNumber numberWithFloat:10.0f]
                 forKey:kCATransactionAnimationDuration];
theLayer.zPosition=200.0;
theLayer.opacity=0.0;
[CATransaction commit];
```

即使上面的示例中显示了 begin 和 commit 所包围的显式事务的时间，你也可以忽略这些而采用隐式事务来替代。

8.2.3 事务的嵌套

显式事务可以被嵌套，允许你禁用部分动画的行为或者在属性被修改的时候产生的动画使用不同的时间。仅当最外层的事务被提交的时候，动画才会发生。

清单 4 中显示了一个嵌套两个事务的例子。最外层的事务设置隐式动画的时间为 2 秒，并设置图层的 position 属性值。内层的事务设置隐式动画的时间为 5 秒，并修改图层的 opacity 和 zPosition 属性值。

Listing 4 Nesting explicit transactions

```
[CATransaction begin]; // outer transaction
```

```
// change the animation duration to 2 seconds
[CATransaction setValue:[NSNumber numberWithFloat:2.0f]
             forKey:kCATransactionAnimationDuration];
// move the layer to a new position
theLayer.position = CGPointMake(0.0,0.0);

[CATransaction begin]; // inner transaction
// change the animation duration to 5 seconds
[CATransaction setValue:[NSNumber numberWithFloat:5.0f]
              forKey:kCATransactionAnimationDuration];

// change the zPosition and opacity
theLayer.zPosition=200.0;
theLayer.opacity=0.0;

[CATransaction commit]; // inner transaction

[CATransaction commit]; // outer transaction
```

第九章　布局核心动画的图层

NSView 提供了经典的“stuts and springs”模式，用于视图调整大小的时候把关联到它父图层的视图重新调整位置。图层支持该模式，而且 Mac OS X 上面的核心动画提供了一个更通用的布局管理器机制，允许开发者自己写他们自己的布局管理器。可以为图层定制一个布局管理器（它通常实现 CALayoutManager 协议），负责给图层的子图层提供布局功能。

本章介绍了约束布局管理器和如何配置一系列约束条件。

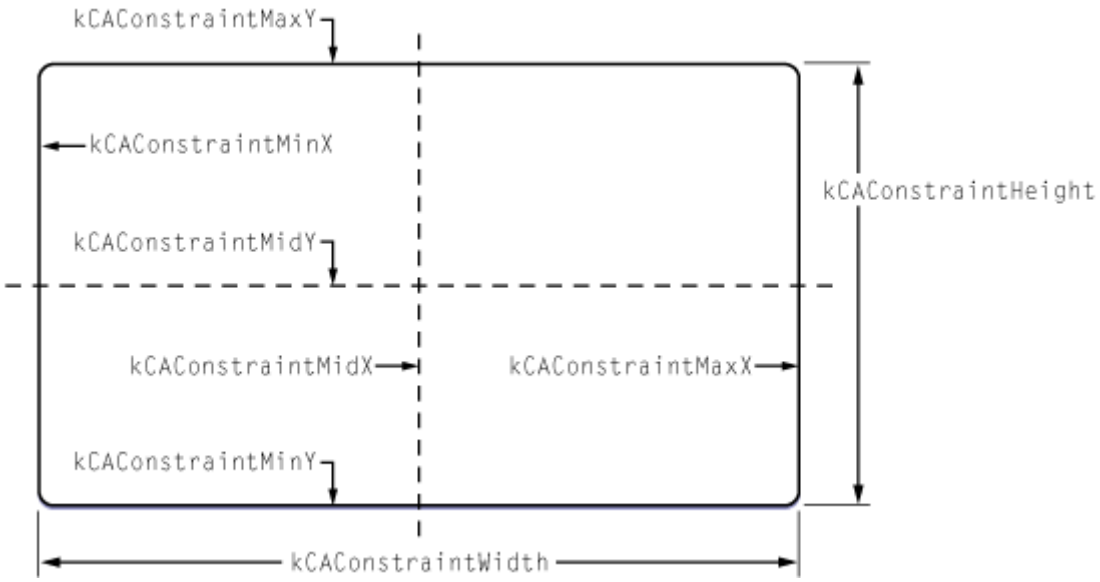
iOS 注意: *iOS 的 CALayer 类仅提供“stuts and springs”模式，不提供定制的布局管理器。然而如果你想人工修改关联到特别视图的图层的位置的话，你可以重载相应视图的 layoutSubviews 方法，在这里面实现你定制的布局代码。你可以查看“iOS 视图编程指南（View Programming Guide for iOS）”来获取更多关于如何在 iOS 应用里面基于视图的布局方法。*

9.1　约束布局管理器

基于条件的布局允许你根据图层和它同级图层或者它的父图层的相应关系指定图层的位置和大小。通过 CAConstraint 类描述的关系被保存在子图层的 constraints 数组属性里面。

图 1 描述了在指定关系的的时候你可以使用的布局特性。

Figure 1 Constraint layout manager attributes



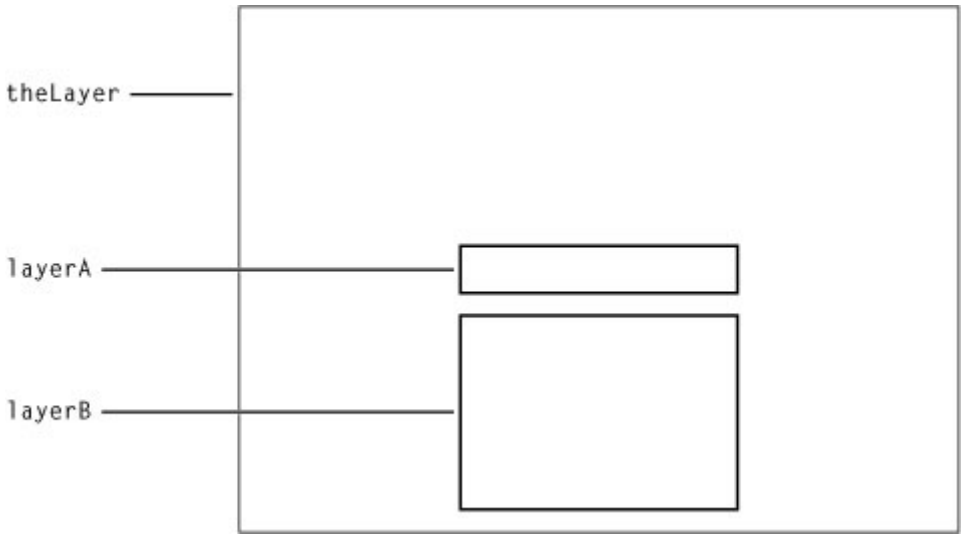
当使用约束布局的时候，你首先创建一个 CAConstraintLayoutManager 的实例，并把它设置为父图层的布局管理器。然后你通过实例化 CAConstraint 对象为子图层创建约束条件，并把这些约束条件通过使用 addConstraint:方法添加到子图层的约束属性里面。每个 CAConstraint 实例封装了一个两个图层在同一轴上的几何关系。

同级层引用的名称，使用图层的 name 属性。特定的名称 superlayer 被使用来引用图层的父图层。

每个轴上面最多只能指定两个关系。如果你给图层的左边和右边都指定约束关系，那么图层的宽度就会不同。如果你给图层的左边和宽度指定约束关系，则图层的右边就会从根据父图层的 frame 移动。通常你一般只会指定单个边的约束条件，图层在同一个轴上面的大小将会作为第二个约束关系。

清单 1 里面的示例代码创建了一个图层，然后使用位置约束条件添加子图层。图 2 描述了布局的结果。

Figure 2 Example constraints based layout



Listing 1 Configuring a layer's constraints

```
// create and set a constraint layout manager for theLayer
theLayer.layoutManager=[CAConstraintLayoutManager layoutManager];

CALayer *layerA = [CALayer layer];
layerA.name = @"layerA";
```

```
layerA.bounds = CGRectMake(0.0,0.0,100.0,25.0);
layerA.borderWidth = 2.0;

[layerA addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMidY
                                          relativeTo:@"superlayer"
                                           attribute:kCAConstraintMidY]];

[layerA addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMidX
                                          relativeTo:@"superlayer"
                                           attribute:kCAConstraintMidX]];

[theLayer addSublayer:layerA];

CALayer *layerB = [CALayer layer];
layerB.name = @"layerB";
layerB.borderWidth = 2.0;

[layerB addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintWidth
                                          relativeTo:@"layerA"
                                           attribute:kCAConstraintWidth]];

[layerB addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMidX
                                          relativeTo:@"layerA"
                                           attribute:kCAConstraintMidX]];


[layerB addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMaxY
                                          relativeTo:@"layerA"
                                           attribute:kCAConstraintMinY
                                              offset:-10.0]];

[layerB addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMinY
```

```
                                        relativeTo:@"superlayer"

                                         attribute:kCAConstraintMinY

                                            offset:+10.0]];


[theLayer addSublayer:layerB];
```

以上是代码执行过程:

1. 创建一个 CAConstraintLayoutmanager 实例，然后把它设置为 theLayer 的 layoutManger 的属性。
2. 创建一个 CALayer(layerA)的实例，设置图层的 name 属性为“layerA”。
3. 设置 layerA 的 bounds 为(0.0,0.0,10.0,25.0)。
4. 创建一个 CAConstraint 对象，把它作为约束条件添加到 layerA 里面。该约束条件是把 layerA 的水平中心对齐它的父图层的水平中心。
5. 创建第二个 CAConstraint 对象，把它作为 layerA 的约束条件。该约束条件是把 layerA 的垂直中心对齐父图层的垂直中心。
6. 把 layerA 添加为 theLayer 的子图层。
7. 创建一个 CALayer(layerB)的实例，设置图层的 name 属性为“layerB”。
8. 创建一个 CAConstraint 对象，给 layerA 添加该约束条件，该约束条件是设置 layerB 的宽度设置为与 layerA 的宽度相同。
9. 创建第二个 CAConstraint 对象，把该约束条件添加到 layerB 里面。该约束条件是设置 layerB 的水平中心对齐 layerA 的水平中心。
10. 创建第三个 CAConstraint 对象，并把它添加为 layerB 的约束条件。该约束条件设置 layerB 的顶边低于 layerA 底边 10 像素。
11. 创建第四个 CAConstraint 对象，把它作为约束条件添加到 layerB 里面。该约束条件是把 layerB 的底边高于父图层底边 10 像素。

注意：*有可能创建约束条件导致在相同的属性的循环引用。在布局是无法计算的情况下，行为结果是不可预知的。*

第十章　核心动画的键-值编码扩展

CAAnimation 和 CALayer 类扩展了 NSKeyValueCoding 协议，给键添加默认值，扩展了封装协议，支持 CGPoint、CGRect、CGSize 和 CATransform3D 关键路径。

10.1 键-值编码兼容的容器类

CALayer 和 CAAnimation 都是键-值编码兼容的容器类，允许你修改属性键对应的值。即使键为“someKey”对应的属性没有被定义，你也可以给“someKey”的键设置一个值，如下：

```
[theLayer setValue:[NSNumber numberWithInteger:50] forKey:@"someKey"];
```

你可以通过下面的代码检索“someKey”对应的值：

```
someKeyValue=[theLayer valueForKey:@"someKey"];
```

Mac OS X 注意:*在 Mac OS X 上面，CALayer 和 CAAnimation 类支持 NSCoding 协议，会自动归档这些你设置的额外键。*

10.2 支持默认值

核心动画添加的键值编码约定，允许一个类在被使用时键没有被设置相应值的时候提供默认值。CALayer 或 CAAnimation 支持该约定，通过使用方法 defaultValueForKey:。

为了给键提供默认值，你创建相应的子类，并重载 defaultValueForKey:。子类实现相应的键参数检查并返回适当的默认值。清单 1 描述了一个实现 defaultValueForKey: 的例子，它给 masksToBounds 提供新的默认值。

Listing 1 Example implementation of defaultValueForKey:

```
+ (id)defaultValueForKey:(NSString *)key
{
    if ([key isEqualToString:@"masksToBounds"])
        return [NSNumber numberWithBool:YES];

    return [super defaultValueForKey:key];
}
```

10.3 封装约定

当使用键值编码方法访问属性，而属性的值不支持标准键-值编码封装约定的对象(NSObject)时候，你可以使用如下的封装约定：

C Type	Class
CGPoint	NSValue
CGSize	NSValue
CGRect	NSValue
CGAffineTransform	NSAffineTransform (Mac OS X only)
CATransform3D	NSValue

10.4 支持结构字段的关键路径

CAAnimation 提供支持使用关键路径访问选择的结构字段。这在为动画关键路径指定结构字段的时候非常有帮助，同时你可以使用 setValue:forKeyPath: 和 valueForKeyPath 来设置和读取相应的值。

CATransform3D 公开如下的字段：

Structure Field	Description
rotation.x	The rotation, in radians, in the x axis.
rotation.y	The rotation, in radians, in the y axis.
rotation.z	The rotation, in radians, in the z axis.
rotation	The rotation, in radians, in the z axis. This is identical to setting the rotation.z field.
scale.x	Scale factor for the x axis.
scale.y	Scale factor for the y axis.
scale.z	Scale factor for the z axis.
scale	Average of all three scale factors.
translation.x	Translate in the x axis.
translation.y	Translate in the y axis.

translation.z	Translate in the z axis.
translation	Translate in the x and y axis. Value is an NSSize or CGSize.

CGPoint 公开如下字段：

Structure Field	Description
x	The x component of the point.
y	The y component of the point.

CGSize 公开如下字段：

Structure Field	Description
width	The width component of the size.
height	The height component of the size.

CGRect 公开如下字段：

Structure Field	Description
origin	The origin of the rectangle as a CGPoint.
origin.x	The x component of the rectangle origin.
origin.y	The y component of the rectangle origin.
size	The size of the rectangle as a CGSize.
size.width	The width component of the rectangle size.
size.height	The height component of the rectangle size.

你不可以通过 Objective-C 2.0 的属性方法来指定一个结构字段的关键路径。如下的代码是无法正常执行的：

```
myLayer.transform.rotation.x=0;
```

相反你必须使用 setValue:forKeyPath:或者 valuForKeyPath:，如下：

```
[myLayer setValue:[NSNumber numberWithInt:0] forKeyPath:@"transform.rotation.x"];
```

第十一章　图层样式属性

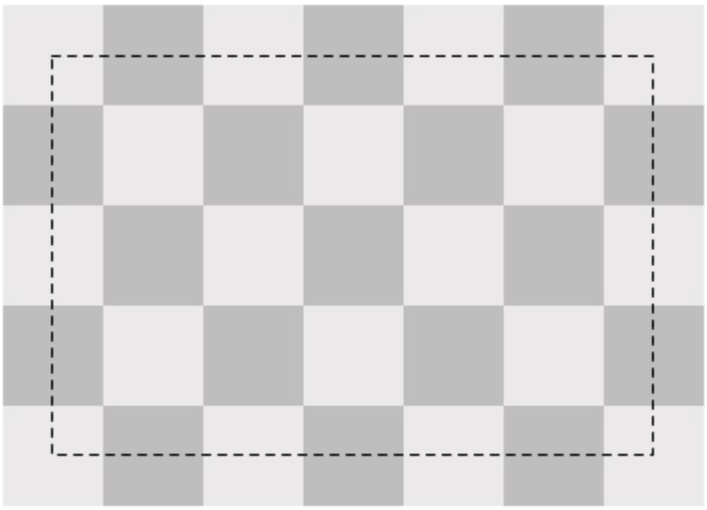
不管图层显示的媒体类型是什么，图层的样式属性被渲染树应用到复杂的图层。本章介绍图层的样式属性和提供例子说明它们对示例图层的影响。

注意:*图层的样式属性在 Mac OS X 和 iOS 上面是不同，或后面将会介绍到。*

11.1 几何属性

图层的几何属性指定图层的显示和它父图层相关性。几何属性指定了使图层边角变为圆角的角度，和应用到图层和它子图层的变换。

Figure 1 shows the geometry of the example layer.



以下 CALayer 的属性指定了图层的几何：

- frame
- bounds
- position
- anchorPoint
- cornerRadius
- transform
- zPosition

iOS 注意:*在 iOS 3.0 之后支持 cornerRadius 属性。*

11.2 背景属性

图层渲染它的背景。你可以定义背景的颜色，也可以定义图像滤镜。图 2 列举了同一个图层设置相应的 backgroundColor 值。

Figure 2 Layer with background color

背景滤镜被应用到图层内容的下面。例如，你不妨套用模糊滤镜滤镜作为图层的背景，这样可以更换把内容地凸现出来。

以下是影响图层背景的 CALayer 属性：

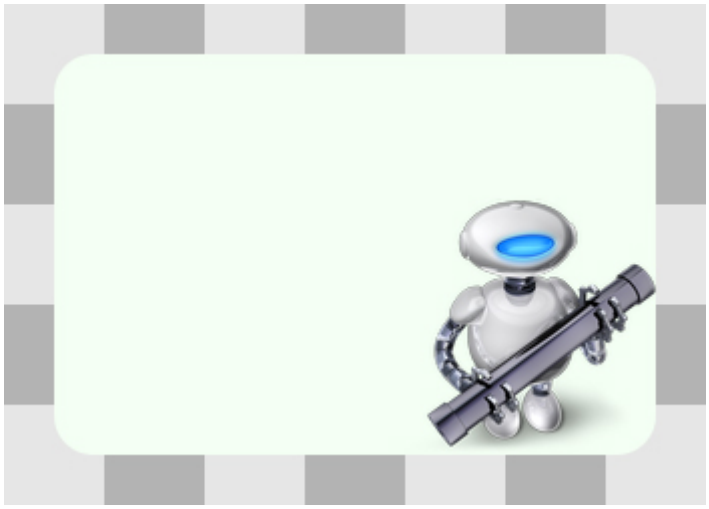
- backgroundColor
- backgroundFilters

iOS 注意:即使 iOS 上面的 CALayer 类公开 bankgroundColor 属性，核心图像(Core Image)依然不可用。该属性可用的滤镜目前暂未定义。

11.3 图层内容

如果你设置了图层内容，则它将会被渲染出来。图层的内容可以通过 Quartz 图像环境、OpenGL、QuickTime 或者 Quartz Composer 来创建。图 4 显示了如何合成图层的内容。

Figure 3 Layer displaying a content image



默认情况下，图层的内容没有被依据它的边界和圆角而裁剪适配。你可以通过设置 masksToBounds 属性值为 YES 来使它裁剪图层内容到适配它的边界和圆角区域。

以下显示了影响图层内容显示的 CALayer 属性：

- contents
- contentsGravity

11.4 子图层内容

通常图层具有层次结构的子图层。这些子图层依据和它父图层的几何关系被递归的渲染到界面。父图层的 sublayerTransform 属性根据它的 anchorPoint 属性被应用到每个子图层。

Figure 4 Layer displaying the sublayers content

默认情况下，图层的子图层不会被裁剪至适配图层的边界和圆角。你可以设置 masksToBounds 属性来让它裁剪图层的内容到适配边界和圆角上。示例中图层的 masksToBounds 属性为 NO。注意到子类图层显示它的监控器和测试模式部分超出父图层的边界。

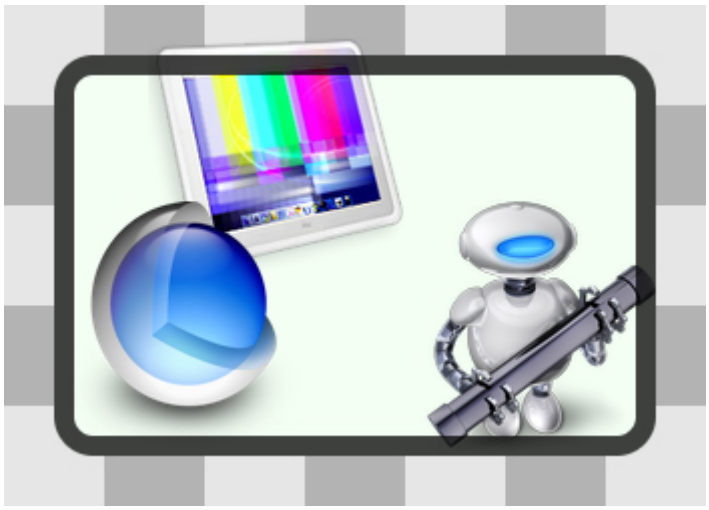
以下显示应用到图层的子图层显示的 CALayer 属性：

- sublayers
- masksToBounds
- sublayerTransform

11.5 边框属性

图层可以使用指定的颜色和宽度来显示一个额外的边框。图 5 显示了示例图层显示一个边框后的情况。

Figure 5 Layer displaying the border attributes content



以下显示了应用到图层边框显示的 CALayer 属性：

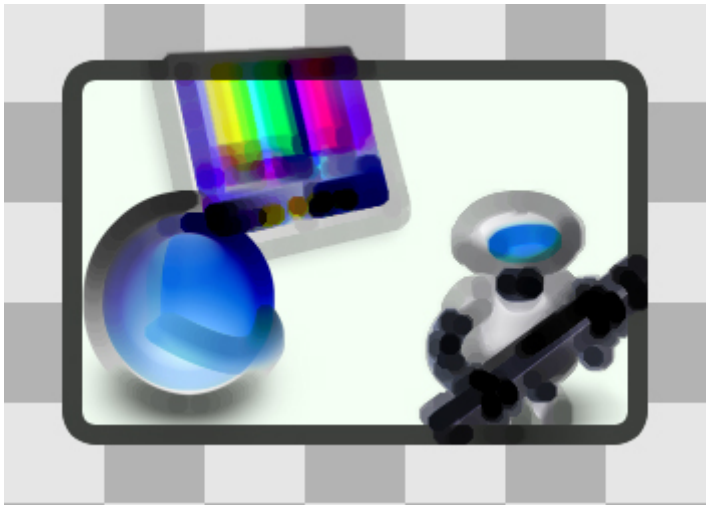
- borderColor
- borderWidth

iOS 注意:borderColor 和 borderWidth 属性仅在 iOS 3.0 之后才支持。

11.6 滤镜属性

一组核心图像的滤镜可以被应用到图层上面。滤镜影响图层的边框、内容和背景。图 6 显示了示例图层被应用了色调分离滤镜后的效果。

Figure 6 Layer displaying the filters properties



以下 CALayer 属性指定了图层内容滤镜：

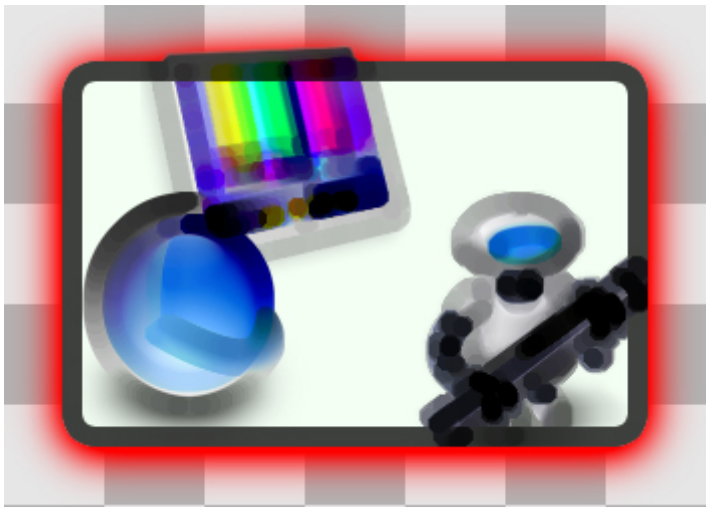
- filters

iOS 注意:即使 iOS 的 CALayer 类公开了 filters 属性，核心图像依然不可用。当前属性可用的滤镜还未定义。

11.7 阴影属性

另外，一个图层可以显示阴影，指定它的不透明度，颜色，偏移量和模糊半径。图 7 显示了示例图层使用一个红色的阴影。

Figure 7 Layer displaying the shadow properties



以下 CALayer 属性影响图层的阴影：

- shadowColor

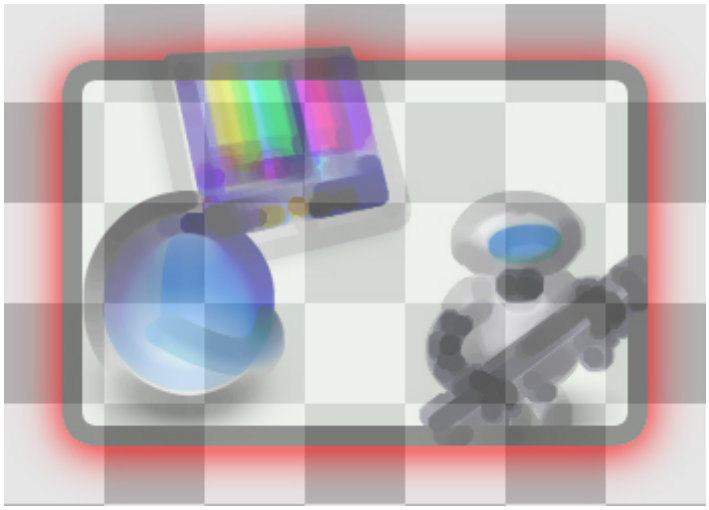
- shadowOffset
- shadowOpacity
- shadowRadius

iOS 注意:shadowColor、shadowOffset、shadowOpacity 和 shadowRadius 属性仅在 iOS 3.2 之后才被支持。

11.8 不透明属性

通过设置图层的不透明属性，你可以控制图层的透明度。图 3 显示了示例图层具有不透明度为 0.5 时候的效果。

Figure 8 Layer including the opacity property



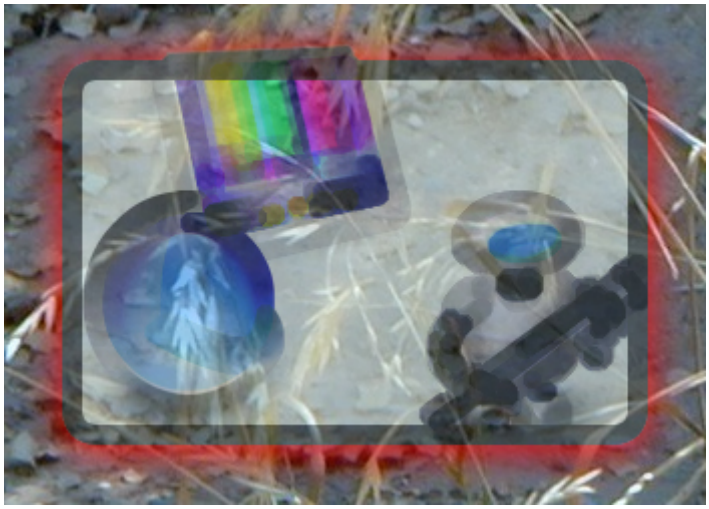
以下 CALayer 的属性指定了图层的不透明度：

- opacity

11.9 混合属性

图层的混合滤镜可以使用来组合图层层与层之间的背后内容。默认情况下，使用源的图层是混合的。图 9 显示了示例图层使用混合滤镜后的效果。

Figure 9 Layer composited using the compositingFilter property



以下 CALayer 的属性为图层指定了混合滤镜：

- compositingFilter

iOS 注意:即使 iOS 的 CALayer 类公开了 compsotingFilter 属性，核心图像依然不可用。当前该属性的可以滤镜尚未定义。

11.10 遮罩属性

最后，你可以指定一个图层作为遮罩，甚至可以修改如何渲染图层的显示。当图层是混合的时候，遮罩图层的不透明度决定了遮罩的效果。图 10 显示了示例图层混合后被一个遮罩图层作用的效果。

Figure 10 Layer composited with the mask property

以下 CALayer 的属性为一个图层指定遮罩：

- mask

iOS 注意:mask 属性仅在 iOS 3.0 之后才支持。

第十二章　示例：核心动画的菜单样式报刊应用

核心动画的菜单样式报刊应用例子显示一个简单的选项示例，使用核心动画的图层生成和动画用户界面。在不到 100 行代码情况下，它演示了以下功能和设计模式：

- 在视图里面托管图层层次结构的根图层。
- 在图层层次结构里面创建插入图层。

使用 QCCompositionLayer 作为图层内容显示 Quartz 混合效果。同时它还演示了使用纯色提高性能。

- 使用显式的动画连续运行。
- 动画的核心图像滤镜输入。
- 隐式动画选择项的位置。
- 通过掌管视图的 MenuView 实例来处理关键事件。

该应用过渡使用核心图像滤镜和 Quartz 混合模式，所以该应用只能运行在 Mac OS X 上面。管理图层的层次结构，隐式和显式动画，事件处理等技术都可以在 iOS 和 Mac OS X 上面通用。

该应用有两个可用的示例代码版本：一个是使用QCCCompositionLayer作为背景，另一个是使用纯黑色作为背景。查看工程“CoreAnimationKioskStyleMenu”来获取可执行的代码。初始化的代码演示描述了采用Quartz混合模式作为背景的应用的版本信息。当使用纯黑色作为背景时可以提升性能，这个再后面再详细讨论。

12.1 用户界面

QCCoreAnimationKioskMenuStyle 的示例提供了一个基础的报刊类应用的用户界面。它占据整个屏幕，用户可以在菜单中选择单一的选项。用户导航菜单使用键盘上的向上和向下箭头。由于选择的变化，选项标志（白色圆角长方形）动画到新的位置。由于为选项标志设置了连续的动画，导致它巧妙地引起你的注意。背景使用 Quartz 混合模式的连续动画，提供了一个受瞩目的背景。

Figure 1 核心动画 Kiosk Menu Interface

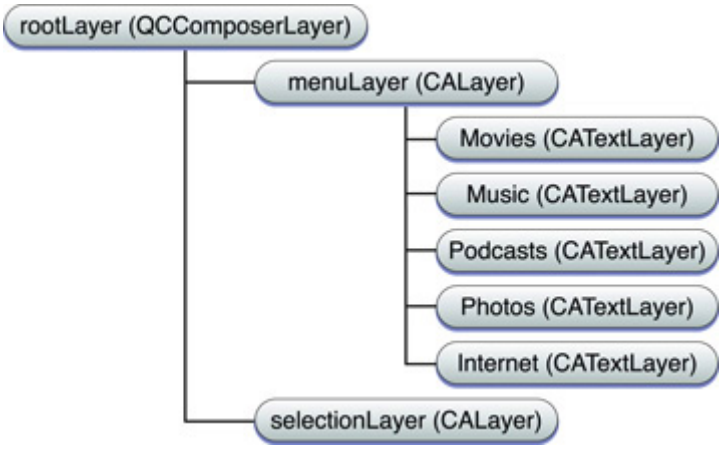
视频观看地址：http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreAnimation_guide/art/QCBackground.m4v

12.1.1检测Nib文件

12.1.2 图层的层次结构

在 CoreAnimationKioskMenuStyle 的应用中图层的层次结构也同样被图层树引用，显示如下：

Figure 2 Layer Hierarchy For QCCoreAnimationKioskStyleMenu Application



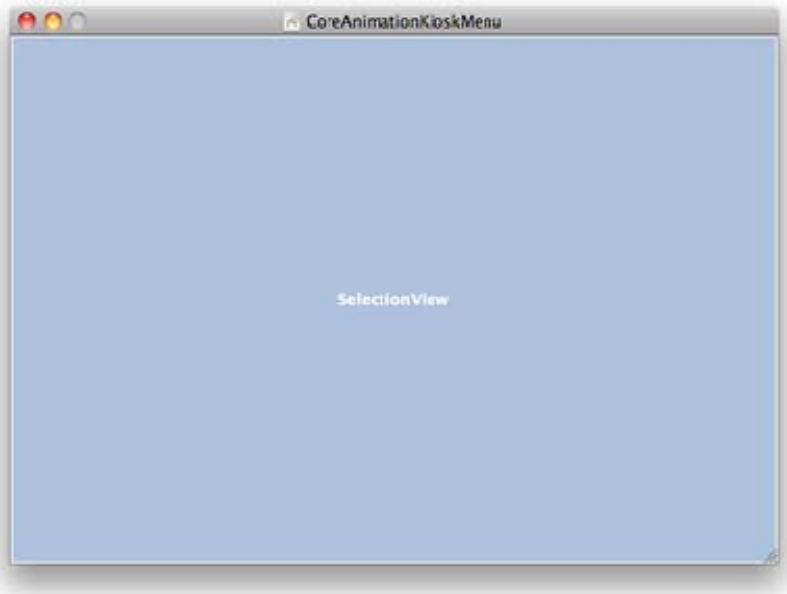
rootLayer 是 QCComposerLayer 的一个实例。根图层和 MenuView 实例的大小相同，采用调整窗口的大小方式。

menuLayer 是 rootLayer 的子图层。它是一个空图层，它的 contents 属性没有被设置任何值，而且它的样式属性也没有被设置相应值。menuLayer 只是简单的作为其他选项图层的容器。它的做法是为了方便程序可以通过访问 menuLayers. sublayers 数组来访问相应的选项子图层。menusLayer 和 rootLayer 一样大小，重叠在一起。这样做是故意的，目的是在定位 selectionLayer 相对当前菜单项时减少两个坐标系之间的转换。

12.2 检测应用程序的Nib文件

MainMenu. nib 是非常简单的。把 CustomView 的一个实例拖到 Interface Builder 的调色板和定位窗口。重新调整大小以便它布满整个窗口。通过把 SelectionView.h 拖到 Menu.nib 窗口导入该头文件。此刻在选择视图中的标识调色板 CustomView 被选中，同时 CustomView 的类被改为 SelectionView。

Figure 3 MainMenu.nib File



不需要做任何的连接。当 nib 文件被加载的时候，窗口会一起反归档 SelectionView 的实例。SelectionView 类的图层在类的 awakenFormNib 里面被相应的设置。

12.3 检测程序的代码

查看程序的 nib 文件和整体设计，你现在可以开始研究实施 SelectionView 类。SelectionView.h 头文件定义了 SelectionView 的属性和方法，SelectionView.m 文件包含了相应的实现。为了在实现文件中更容易找到的代码段的，#pragma mark 被添加到相应的行上面。

12.3.1 QCCoreAnimationKioskStyleMenu.h 和 QCCoreAnimationKioskStyleMenu.m 文件

在程序创建的时候，CodeAnimationKioskMenuAppDelegate.h 文件和它的.m 文件被自动创建。它和本示例没有相关性代码。

12.3.2 检测SelectionView.h

SelectionView 类是 NSView 的子类。在 SelectionView.h 里面它定义了四个属性和

其他方法。在应用程序窗口里面它是唯一包含 rootLayer 和图层树里面的其他图层的视图。

Listing 1 SelectionView.h File Listing

```
#import <Cocoa/Cocoa.h>
#import <QuartzCore/Quartz.h>



// The SelectionView class is the view subclass that is inserted into
// the window.  It hosts the rootLayer, and responds to events.
@interface SelectionView : NSView {

    // Contains the selected menu item index
    NSInteger selectedIndex;

    // The layer that contains the menu item layers
    CALayer *menuLayer;

    // The layer that is displays the selection
    CALayer *selectionLayer;

    // The array of layers that contain the menu item names.
    NSArray *names;

}

@property  NSInteger selectedIndex;
@property (retain) CALayer *menuLayer;
@property (retain) CALayer *selectionLayer;
@property (retain) NSArray *names;

-(void)awakeFromNib;
-(void)setupLayers;
```

```
-(void)changeSelectedIndex:(NSInteger)theSelectedIndex;

-(void)moveUp:(id)sender;

-(void)moveDown:(id)sender;

-(void)dealloc;


@end
```

注意:注意到 Quartz/CoreAnimation.h 被导入。在使用核心动画的任何时候，都必须导入 QuartzCore.framework。因为本示例使用了 Quartz 混合模式，所以 SelectionView.h 头文件也导入了 Quartz/Quartz.h,而且 Quartz.framework 也被添加到工程里面。

12.3.3检测SelectionView.h

SelectionView 类是整个应用程序的核心。当视图加载 nib 文件后，它负责填充当前屏幕，设置要显示的图层，创建动画，处理移动选项。

SelectionView.m 文件划分如下:

- 实现方法 awakenFromNib
- 设置图层
- 动画显示选项图层的移动
- 处理重要的事件
- 清理

实现方法 awakenFromNib

当 MainMenu.nib 被加载和反归档的时候，awakeFromNib 被调用。视图应该在 awakenFormNib 完成它的相应设置。

MainMenuView 实现 awakenFormNib 做了以下操作:

- 创建一个名称字符串数组，用来显示菜单选项的名称。
- 隐藏当前光标。全屏应用一般不显示光标，本应用程序完全依赖键盘的输入。
- 视图调整大小以适应填充整个屏幕，同时窗口也跟着调整大小。
- 使窗口成为 firstResponder，以便它获取键盘的向上和向下箭头事件。
- 调用 setupLayers 方法来设置图层（在“setupLayers”方法里面讨论）。
- 最后把窗口显示在最上面，并让它可视化。

清单 2 的代码显示了如何实现上述的功能。在项目工程的 SelectionView.m 里面，

它通过#pragma mark “Implementantion of awakeFromNib”标识位置。

Listing 2 Implementation of awakeFromNib

```
#pragma mark - Listing 1: Implementation of awakeFromNib

- (void)awakeFromNib

{


    // create an array that contains the various

    // strings

    self.names=[NSArray arrayWithObjects:@"Movies",@"Music",

                @"Podcasts",@"Photos",@"Internet",

                nil] ;


    // The cursor isn't used for selection, so we hide it

    [NSCursor hide];


    // go full screen, as a kiosk application

    [self enterFullScreenMode:[self.window screen] withOptions:NULL];


    // Make the window the first responder to get keystrokes

    [self.window makeFirstResponder:self];


    // setup the individual layers

    [self setupLayers];


    // bring the window to the front

    [self.window makeKeyAndOrderFront:self];


}
```

setupLayers 方法

QCCoreAnimationKioskStyleMenu 程序的主要代码都放在 setupLayers 方法里面。该方法负责创建图层、动画和设置当前的选择选。

清单 3 显示了#pragma mark “Configuration of the Background rootLayer”。这个代码片段为所有的子层设置了 rootLayer。

在这部分代码中有两个重要的地方:

只有在视图的图层可用的时候，才会创建 rootLayer 并把它设置给视图。这样处理的顺序导致视图使用特定的图层，而不是自己给自己创建特定的图层。这就是图层-视图托管模式（**layer-view hosting**）。图层-视图托管模式要求所有的重绘都是由核心动画来完成。你不能使用的 NSView 的绘图功能和方法，但图层-视图托管模式通常可以使用。

- 另外和性能有关。在背景运行 Quartz 混合模式动画可能会导致性能问题，这依赖于其硬件设施。使用纯色背景可以提高性能。这在后面“Performance Considerations”部分讨论。

Listing 3 Configuration of the Background rootLayer

```
#pragma mark Listing: "Configuration of the Background rootLayer"


-(void)setupLayers;
{
    // make a Quartz Composition Layer.
    // Note: Running a QCComposition can significantly impact performance
    QCCompositionLayer* rootLayer=[QCCompositionLayer compositionLayerWithFile:
              [[NSBundle mainBundle] pathForResource:@"Background" ofType:@"qtz"]];


    // Set the QCCompositionLayer as the root layer
    // and then turn on wantsLayer. This order cases
    // layer-hosting behavior on the part of the view..
    [self setLayer:rootLayer];
    [self setWantsLayer:YES];
```

清单 4 中显示了#pragma mark Setup menuLayers Array. The Selectable Menu Items. 该代码初始化 menuLayer, 并把它插入到 rootLayer 的子图层里面（查看清单 2 中图层的层次结构提示）。它插入每个选项文字插入独立的图层到 menuLayer。该代码在例子中通过#prragma mark - Setup menuLayers Array. The Selectable Menu

Items 标识。

Listing 4 Setup menuLayers Array. The Selectable Menu Items.

```
// Create a layer to contain the menus
self.menuLayer=[CALayer layer] ;
self.menuLayer.frame=rootLayer.frame;
self.menuLayer.layoutManager=[CAConstraintLayoutManager layoutManager];
[rootLayer addSublayer:self.menuLayer];

// setup and calculate the size and location of the individually selectable items.
CGFloat width=400.0;
CGFloat height=50.0;
CGFloat spacing=20.0;
CGFloat fontSize=32.0;
CGFloat initialOffset=self.bounds.size.height/2-(height*5+spacing*4)/2.0;

//Create whiteColor it's used to draw the text and also in the selectionLayer
CGColorRef whiteColor=CGColorCreateGenericRGB(1.0f,1.0f,1.0f,1.0f);


// Iterate over the list of selection names and create layers for each.
// The menuItemLayer's are also positioned during this loop.
NSInteger i;
for (i=0;i<[names count];i++) {

    CATextLayer *menuItemLayer=[CATextLayer layer];
    menuItemLayer.string=[self.names objectAtIndex:i];
    menuItemLayer.font=@"Lucida-Grande";
    menuItemLayer.fontSize=fontSize;
    menuItemLayer.foregroundColor=whiteColor;
    [menuItemLayer addConstraint:[CAConstraint
                                  constraintWithAttribute:kCAConstraintMaxY
                                  relativeTo:@"superlayer"
```

```
                                attribute:kCAConstraintMaxY
                                offset:-(i*height+spacing+initialOffset)]];
        [menuItemLayer addConstraint:[CAConstraint
                                constraintWithAttribute:kCAConstraintMidX
                                relativeTo:@"superlayer"
                                attribute:kCAConstraintMidX]];
        [self.menuLayer addSublayer:menuItemLayer];
    } // end of for loop
    [self.menuLayer layoutIfNeeded];
```

清单 5 中显示了#pragma mark Setup selectionLayer. Used to Display the Currently Selected. 该被注释掉的代码，但可以概括如下：

- 创建 CALayer 的 selectionLayer。
- 给图层添加 CIBloom 滤镜，设置连续的动画来移动选项标志。
- 设置 selectionIndex 初始化值为 0。
- 把 selectionLayer 添加到 rootLayer 作为子图层。

可以在示例代码中在#pragma mark - Setup selectionLayer.Displays the Currently Selected Item 后面查看实现。

Listing 5 Setup selectionLayer. Used to Display the Currently Selected Item.

```
#pragma mark - Setup selectionLayer. Displays the Currently Selected Item.


    // we use an additional layer, selectionLayer
    // to indicate that the current item is selected
    self.selectionLayer=[CALayer layer];
    self.selectionLayer.bounds=CGRectMake(0.0,0.0,width,height);
    self.selectionLayer.borderWidth=2.0;
    self.selectionLayer.cornerRadius=25;
    self.selectionLayer.borderColor=whiteColor;
    CIFilter *filter = [CIFilter filterWithName:@"CIBloom"];
    [filter setDefaults];
    [filter setValue:[NSNumber numberWithFloat:5.0] forKey:@"inputRadius"];
    [filter setName:@"pulseFilter"];
```

```
    [selectionLayer setFilters:[NSArray arrayWithObject:filter]];

    // The selectionLayer shows a subtle pulse as it
    // is displayed. This section of the code create the pulse animation
    // setting the filters.pulsefilter.inputintensity to range from 0 to 2.
    // This will happen every second, autoreverse, and repeat forever
    CABasicAnimation* pulseAnimation = [CABasicAnimation animation];
    pulseAnimation.keyPath = @"filters.pulseFilter.inputIntensity";
    pulseAnimation.fromValue = [NSNumber numberWithFloat: 0.0];
    pulseAnimation.toValue = [NSNumber numberWithFloat: 2.0];
    pulseAnimation.duration = 1.0;
    pulseAnimation.repeatCount = 1e100f;
    pulseAnimation.autoreverses = YES;
    pulseAnimation.timingFunction = [CAMediaTimingFunction functionWithName:
                                kCAMediaTimingFunctionEaseInEaseOut];
    [selectionLayer addAnimation:pulseAnimation forKey:@"pulseAnimation"];

    // set the first item as selected
    [self changeSelectedIndex:0];

    // finally, the selection layer is added to the root layer
    [rootLayer addSublayer:self.selectionLayer];

    // cleanup
    CGColorRelease(whiteColor);
    // end of setupLayers
}
```

处理关键事件

因为图层没有参与响应链，接收事件，所以 SelectionView 视图实例作为图层托管角色。这是为什么在 awakeFromNib 方法里面 SelectionView 被注册为第一监听者。

NSResponder 提供 moveUp:和 moveDown 消息，他们是 SelectionView，同时也是

其他所有视图类的后代。当向上箭头和向下箭头被分别按下的时候，会调用相应的 moveUp:和 moveDown：消息。这些方法允许用户指定的任何新映射箭头键到应用程序方面的功能。虽然你需要在你的应用程序里面处理很多负责的键，但实现 keyDown: 方法很简单。清单 6 显示了设置当前选中图层的实现方法。你可以在#pragma mark “Handle Changes in the Selection”标志地方找到相应代码。

Listing 6 Handle Changes in Selection

```
#pragma mark Handle Changes in the Selection


-(void)changeSelectedIndex:(NSInteger)theSelectedIndex

{

    self.selectedIndex=theSelectedIndex;


    if (self.selectedIndex == [names count])

        self.selectedIndex=[names count]-1;

    if (self.selectedIndex < 0)

        self.selectedIndex=0;


    CALayer *theSelectedLayer=[[self.menuLayer sublayers]
objectAtIndex:self.selectedIndex];


    // Moves the selectionLayer to illustrate the

    // currently selected item. It does this

    // using an animation so that the transition

    // is visible.

    self.selectionLayer.position=theSelectedLayer.position;

};
```

当向上箭头被按下的时候，selectedIndex 的值会递减，通过调用 changeSelectedIndex:来更新。就如清单 6 所显示的，移动选择项以突出正确的选项。示例代码在#pragma mark Handle Keystrokes 部分显示。

Listing 7 Handling Up and Down Key Presses

```
#pragma mark Handle Keystrokes



-(void)moveUp:(id)sender
{
    [self changeSelectedIndex:self.selectedIndex-1];
}

-(void)moveDown:(id)sender
{
    [self changeSelectedIndex:self.selectedIndex+1];

}
```

当 SelectionView 被关闭的时候，我们负责清理实例变量。在 dealloc 实现里面把 menuLayer、selectionLayer 和 names 属性设置为空（通过这样可以释放他们）。部分示例代码在#pragma mark Dealloc and Cleanup 部分后面。

Listing 8 Dealloc and Cleanup

```
#pragma mark Dealloc and Cleanup



-(void)dealloc
{
    [self setLayer:nil];
    self.menuLayer=nil;
    self.selectionLayer=nil;
    self.names=nil;
    [super dealloc];
}
```

12.4 性能注意事项

在这个应用界面里面把背景设置为 Quartz 混合模式动画是应用更有趣而且更吸引眼

球。然而你需要在目标硬件配置上面做性能测试。当运行在 Quartz 混合动画模式上面的时候，并不是所有的硬件配置的都可以保证选项导航栏流畅的动画效果。你需要考虑使用纯演示或者一个静态图片作为背景。

为了帮助你评估任何影响性能的任何因素，示例代码工程包含了一个额外的工程 CoreAnmationKioskStyleMenu 工程，它使用纯颜色作为内容背景而不是使用 Quartz 混合模式动画。它将会对你比较两个应用的性能潜在影响因素非常有帮助。

Figure 4 Alternate Interface With Black Background.

视频观看地址：http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreAnimation_guide/art/BlackBackground.m4v

第十三章　动画的属性

CALayer 和 CIFilter 的很多属性都是可以动画的。本文列出了这些属性，和默认情况下他们使用的动画。

13.1 CALayer的动画属性

以下CALayer类的属性可以被核心动画执行动画。可以查看CALayer来获取更多的信息。

- anchorPoint

  使用表 1 中所述默认隐式 CABasicAnimation。

- backgroundColor

  使用表 1 中所述默认隐式 CABasicAnimation。（子属性是使用基本的动画动画）

- backgroundFilters

  使用表 2 中所述默认隐式 CATransitionAnimation。滤镜的子属性使用表 1 中的默认隐式 CABasicAnimation 来动画。

- borderColor

  使用表 1 中所述默认隐式 CABasicAnimation。

- borderWidth

  使用表 1 中所述默认隐式 CABasicAnimation。

- bounds

  使用表 1 中所述默认隐式 CABasicAnimation。

- compositingFilter

  使用表 2 中所述默认隐式 CATransitionAnimation。滤镜的子属性使用表 1 中的默认隐式 CABasicAnimation 来动画。

- contents

- contentsRect

  使用表 1 中所述默认隐式 CABasicAnimation。

- cornerRadius

使用表 1 中所述默认隐式 CABasicAnimation。

- doubleSided

  没有设置默认隐式动画。

- filters

  使用表 1 中所述默认隐式 CABasicAnimation。滤镜的子属性使用表 1 中的默认隐式 CABasicAnimation 来动画。

- frame

  frame 属性本身自己是无法动画的，你可以通过修改 bounds 和 position 属性来替代产生相同的效果。

- hidden

  使用表 1 中所述默认隐式 CABasicAnimation。

- mask

  使用表 1 中所述默认隐式 CABasicAnimation。

- masksToBounds

  使用表 1 中所述默认隐式 CABasicAnimation。

- opacity

  使用表 1 中所述默认隐式 CABasicAnimation。

- position

  使用表 1 中所述默认隐式 CABasicAnimation。

- shadowColor

  使用表 1 中所述默认隐式 CABasicAnimation。

- shadowOffset

  使用表 1 中所述默认隐式 CABasicAnimation。

- shadowOpacity

  使用表 1 中所述默认隐式 CABasicAnimation。

- shadowRadius

  使用表 1 中所述默认隐式 CABasicAnimation。

- sublayers

  使用表 2 中所述默认隐式 `CATransitionAnimation` 。

- sublayerTransform

  使用表 1 中所述默认隐式 CABasicAnimation。

- transform

  使用表 1 中所述默认隐式 CABasicAnimation。

- zPosition

  使用表 1 中所述默认隐式 CABasicAnimation。

Table 1 Default Implied Basic Animation

Description	Value
Class	CABasicAnimation
duration	.25 seconds, or the duration of the current transaction
keyPath	Dependent on layer property type

Table 2 Default Implied Transition

Description	Value
Class	CATransition
duration	.25 seconds, or the duration of the current transaction
type	Fade (kCATransitionFade)
startProgress	0.0
endProgress	1.0

13.2 CIFilter动画的属性

核心动画添加如下的动画属性到Core Image的CIFilter类。查看CIFilter 核心动画 Additions来获取更多的信息。以下的属性在仅在Mac OS X上面可用。

- name
- enabled

结束语

本文在翻译过程中发现很多地方直译成中文比较晦涩，所以采用了意译的方式，这不可避免的造成有一些地方可能和原文有一定的出入，所以如果你阅读的时候发现有任何的错误都可以给我发邮件:xyl.layne@gmail.com。

最后可以关注我微博大家一起沟通交流学习。

微博地址：http://weibo.com/u/1826448972