iOS绘图教程

发布于：2014-01-15 11:15 阅读数：90857

> Core Graphics Framework是一套基于C的API框架，使用了Quartz作为绘图引擎。它提供了低级别、轻量级、高保真度的2D渲染。该框架可以用于基于路径的绘图、变换、颜色管理、脱屏渲染，模板、渐变、

阅读器 iOS开发 iOS iOS绘图

本文是《Programming iOS5》中Drawing一章的翻译，考虑到主题完整性，翻译版本中加入了一些书中未涉及到的内容。希望本文能够对你有所帮助。（本文由海水的味道翻译整理，转载请注明译者和出处，请勿用于商业用途！原文）

Core Graphics Framework是一套基于C的API框架，使用了Quartz作为绘图引擎。它提供了低级别、轻量级、高保真度的2D渲染。该框架可以用于基于路径的绘图、变换、颜色管理、脱屏渲染，模板、渐变、遮蔽、图像数据管理、图像的创建、遮罩以及PDF文档的创建、显示和分析。为了从感官上对这些概念做一个入门的认识，你可以运行一下官方的example code。

iOS支持两套图形API族：Core Graphics/QuartZ 2D 和OpenGL ES。OpenGL ES是跨平台的图形API，属于OpenGL的一个简化版本。QuartZ 2D是苹果公司开发的一套API，它是Core Graphics Framework的一部分。需要注意的是：OpenGL ES是应用程序编程接口，该接口描述了方法、结构、函数应具有的行为以及应该如何被使用的语义。也就是说它只定义了一套规范，具体的实现由设备制造商根据规范去做。而往往很多人对接口和实现存在误解。举一个不恰当的比喻：上发条的时钟和装电池的时钟都有相同的可视行为，但两者的内部实现截然不同。因为制造商可以自由的实现Open GL ES，所以不同系统实现的OpenGL ES也存在着巨大的性能差异。

Core Graphics API所有的操作都在一个上下文中进行。所以在绘图之前需要获取该上下文并传入执行渲染的函数中。如果你正在渲染一副在内存中的图片，此时就需要传入图片所属的上下文。获得一个图形上下文是我们完成绘图任务的第一步，你可以将图形上下文理解为一块画布。如果你没有得到这块画布，那么你就无法完成任何绘图操作。当然，有许多方式获得一个图形上下文，这里我介绍两种最为常用的获取方法。

第一种方法就是创建一个图片类型的上下文。调用UIGraphicsBeginImageContextWithOptions函数就可获得用来处理图片的图形上下文。利用该上下文，你就可以在其上进行绘图，并生成图片。调用UIGraphicsGetImageFromCurrentImageContext函数可从当前上下文中获取一个UIImage对象。记住在你所有的绘图操作后别忘了调用UIGraphicsEndImageContext函数关闭图形上下文。

第二种方法是利用cocoa为你生成的图形上下文。当你子类化了一个UIView并实现了自己的drawRect：方法后，一旦drawRect：方法被调用，Cocoa就会为你创建一个图形上下文，此时你对图形上下文的所有绘图操

开发者通道

排行榜 代码库 图书库

网站库 发码区 工具库

招聘区 外包区 问答区

关注CocoaChina

关注微信 移动版

最近更新

1 OCaml 发布 iOS 7 版编译器（OCam 2014-08-26

2 iOS 8自动调整UITableView和UIColl 2014-08-25

3 对访问控制与protected的理解 2014-08-20

4 Facebook Shimmer 实现原理 2014-08-18

5 在企业内部分发 iOS 应用程序 2014-08-18

6 iOS中图形图像处理第一部分:位图图 2014-08-12

7 WWDC 2014 大会中的 Playgroun 2014-08-11

8 Collection View 动画 2014-07-25

9 iTerm2 2.0版本已发布,添加大量新功 2014-07-21

10 重制Skype应用中Action Sheet的动i

作都会显示在UIView上。

判断一个上下文是否为当前图形上下文需要注意的几点：
1.UIGraphicsBeginImageContextWithOptions函数不仅仅是创建了一个适用于图形操作的上下文，并且该上下文也属于当前上下文。
2.当drawRect方法被调用时，UIView的绘图上下文属于当前图形上下文。
3.回调方法所持有的context：参数并不会让任何上下文成为当前图形上下文。此参数仅仅是对一个图形上下文的引用罢了。

作为初学者，很容易被**UIKit**和**Core Graphics**两个支持绘图的框架迷惑。

UIKit

像UIImage、NSString（绘制文本）、UIBezierPath（绘制形状）、UIColor都知道如何绘制自己。这些类提供了功能有限但使用方便的方法来让我们完成绘图任务。一般情况下，UIKit就是我们所需要的。

使用UiKit，你只能在当前上下文中绘图，所以如果你当前处于UIGraphicsBeginImageContextWithOptions函数或drawRect：方法中，你就可以直接使用UIKit提供的方法进行绘图。如果你持有一个context：参数，那么使用UIKit提供的方法之前，必须将该上下文参数转化为当前上下文。幸运的是，调用UIGraphicsPushContext 函数可以方便的将context：参数转化为当前上下文，记住最后别忘了调用UIGraphicsPopContext函数恢复上下文环境。

Core Graphics

这是一个绘图专用的API族，它经常被称为QuartZ或QuartZ 2D。Core Graphics是iOS上所有绘图功能的基石，包括UIKit。

使用Core Graphics之前需要指定一个用于绘图的图形上下文（CGContextRef），这个图形上下文会在每个绘图函数中都会被用到。如果你持有一个图形上下文context：参数，那么你等同于有了一个图形上下文，这个上下文也许就是你需要用来绘图的那个。如果你当前处于UIGraphicsBeginImageContextWithOptions函数或drawRect：方法中，并没有引用一个上下文。为了使用Core Graphics，你可以调用UIGraphicsGetCurrentContext函数获得当前的图形上下文。

至此，我们有了两大绘图框架的支持以及三种获得图形上下文的方法（drawRect:、drawRect: inContext:、UIGraphicsBeginImageContextWithOptions）。那么我们就有6种绘图的形式。如果你有些困惑了，不用怕，我接下来将说明这6种情况。无需担心还没有具体的绘图命令，你只需关注上下文如何被创建以及我们是在使用UIKit还是Core Graphics。

第一种绘图形式：在UIView的子类方法drawRect：中绘制一个蓝色圆，使用UIKit在Cocoa为我们提供的当前上下文中完成绘图任务。

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

UIBezierPath* p = [UIBezierPathbezierPathWithOvalInRect:CGRectMake(0,0,100,100)];

[[UIColor blueColor] setFill];

[p fill];

}
```

第二种绘图形式：使用Core Graphics实现绘制蓝色圆。

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

CGContextRef con = UIGraphicsGetCurrentContext();

CGContextAddEllipseInRect(con, CGRectMake(0,0,100,100));

```

```
CGContextSetFillColorWithColor(con, [UIColor blueColor].CGColor);

CGContextFillPath(con);

}
```

第三种绘图形式：我将在UIView子类的drawLayer:inContext：方法中实现绘图任务。drawLayer:inContext：方法是一个绘制图层内容的代理方法。为了能够调用drawLayer:inContext：方法，我们需要设定图层的代理对象。但要注意，不应该将UIView对象设置为显示层的委托对象，这是因为UIView对象已经是隐式层的代理对象，再将它设置为另一个层的委托对象就会出问题。轻量级的做法是：编写负责绘图形的代理类。在MyView.h文件中声明如下代码：

```
@interface MyLayerDelegate : NSObject

@end
```

然后MyView.m文件中实现接口代码：

```
@implementation MyLayerDelegate

- (void)drawLayer:(CALayer*)layer inContext:(CGContextRef)ctx {

  UIGraphicsPushContext(ctx);

  UIBezierPath* p = [UIBezierPath bezierPathWithOvalInRect:CGRectMake(0,0,100,100)];

  [[UIColor blueColor] setFill];

  [p fill];

  UIGraphicsPopContext();

}

@end
```

直接将代理类的实现代码放在MyView.m文件的#import代码的下面，这样感觉好像在使用私有类完成绘图任务（虽然这不是私有类）。需要注意的是，我们所引用的上下文并不是当前上下文，所以为了能够使用UIKit，我们需要将引用的上下文转变成当前上下文。

因为图层的代理是assign内存管理策略，那么这里就不能以局部变量的形式创建MyLayerDelegate实例对象赋值给图层代理。这里选择在MyView.m中增加一个实例变量，因为实例变量默认是strong:

```
@interface MyView () {

MyLayerDelegate* _layerDeleagete;

}

@end
```

使用该图层代理：

```
MyView *myView = [[MyView alloc] initWithFrame: CGRectMake(0, 0, 320, 480)];

CALayer *myLayer = [CALayer layer];

_layerDelegate = [[MyLayerDelegate alloc] init];

myLayer.delegate = _layerDelegate;

[myView.layer addSublayer:myLayer];

[myView setNeedsDisplay]; // 调用此方法，drawLayer: inContext:方法才会被调用。
```

第四种绘图形式： 使用Core Graphics在drawLayer:inContext：方法中实现同样操作，代码如下：

```
- (void)drawLayer:(CALayer*)lay inContext:(CGContextRef)con {

CGContextAddEllipseInRect(con, CGRectMake(0,0,100,100));

CGContextSetFillColorWithColor(con, [UIColor blueColor].CGColor);

CGContextFillPath(con);

}
```

最后，演示UIGraphicsBeginImageContextWithOptions的用法，并从上下文中生成一个UIImage对象。生成UIImage对象的代码并不需要等待某些方法被调用后或在UIView的子类中才能去做。

第五种绘图形式： 使用UIKit实现：

```
UIGraphicsBeginImageContextWithOptions(CGSizeMake(100,100), NO, 0);

UIBezierPath* p = [UIBezierPath bezierPathWithOvalInRect:CGRectMake(0,0,100,100)];

[[UIColor blueColor] setFill];

[p fill];

UIImage* im = UIGraphicsGetImageFromCurrentImageContext();

UIGraphicsEndImageContext();
```

解释一下UIGraphicsBeginImageContextWithOptions函数参数的含义：第一个参数表示所要创建的图片的尺寸；第二个参数用来指定所生成图片的背景是否为不透明，如上我们使用YES而不是NO，则我们得到的图片背景将会是黑色，显然这不是我想要的；第三个参数指定生成图片的缩放因子，这个缩放因子与UIImage的scale属性所指的含义是一致的。传入0则表示让图片的缩放因子根据屏幕的分辨率而变化，所以我们得到的图片不管是在单分辨率还是视网膜屏上看起来都会很好。

第六种绘图形式： 使用Core Graphics实现：

```
UIGraphicsBeginImageContextWithOptions(CGSizeMake(100,100), NO, 0);

CGContextRef con = UIGraphicsGetCurrentContext();

CGContextAddEllipseInRect(con, CGRectMake(0,0,100,100));

CGContextSetFillColorWithColor(con, [UIColor blueColor].CGColor);

CGContextFillPath(con);

UIImage* im = UIGraphicsGetImageFromCurrentImageContext();

UIGraphicsEndImageContext();
```

UIKit和Core Graphics可以在相同的图形上下文中混合使用。在iOS 4.0之前，使用UIKit和UIGraphicsGetCurrentContext被认为是线程不安全的。而在iOS4.0以后苹果让绘图操作在第二个线程中执行解决了此问题。

UIImage常用的绘图操作

一个UIImage对象提供了向当前上下文绘制自身的方法。我们现在已经知道如何获取一个图片类型的上下文并将它转变成当前上下文。

平移操作：下面的代码展示了如何将UIImage绘制在当前的上下文中。

```
UIImage* mars = [UIImage imageNamed:@"Mars.png"];

```

```
CGSize sz = [mars size];

UIGraphicsBeginImageContextWithOptions(CGSizeMake(sz.width*2, sz.height), NO, 0);

[mars drawAtPoint:CGPointMake(0,0)];

[mars drawAtPoint:CGPointMake(sz.width,0)];

UIImage* im = UIGraphicsGetImageFromCurrentImageContext();

UIGraphicsEndImageContext();

UIImageView* iv = [[UIImageView alloc] initWithImage:im];

[self.window.rootViewController.view addSubview: iv];

    iv.center = self.window.center;
```

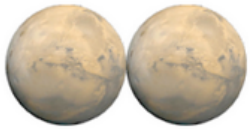

图1 UIImage平移处理

缩放操作：下面代码展示了如何对UIImage进行缩放操作：

```
UIImage* mars = [UIImage imageNamed:@"Mars.png"];

CGSize sz = [mars size];

UIGraphicsBeginImageContextWithOptions(CGSizeMake(sz.width*2, sz.height*2), NO, 0);

[mars drawInRect:CGRectMake(0,0,sz.width*2,sz.height*2)];

[mars drawInRect:CGRectMake(sz.width/2.0, sz.height/2.0, sz.width, sz.height) blendM
ode:kCGBlendModeMultiply alpha:1.0];

UIImage* im = UIGraphicsGetImageFromCurrentImageContext();

UIGraphicsEndImageContext();
```

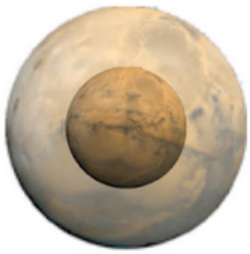

图2 UIImage缩放处理

UIImage没有提供截取图片指定区域的功能。但通过创建一个较小的图形上下文并移动图片到一个适当的图形上下文坐标系内，指定区域内的图片就会被获取。

裁剪操作：下面代码展示了如何获取图片的右半边：

```
UIImage* mars = [UIImage imageNamed:@"Mars.png"];

CGSize sz = [mars size];

UIGraphicsBeginImageContextWithOptions(CGSizeMake(sz.width/2.0, sz.height), NO, 0);

[mars drawAtPoint:CGPointMake(-sz.width/2.0, 0)];

UIImage* im = UIGraphicsGetImageFromCurrentImageContext();

UIGraphicsEndImageContext();
```

以上的代码首先创建一个一半图片宽度的图形上下文，然后将图片左上角原点移动到与图形上下文负X坐标对齐，从而让图片只有右半部分与图形上下文相交。

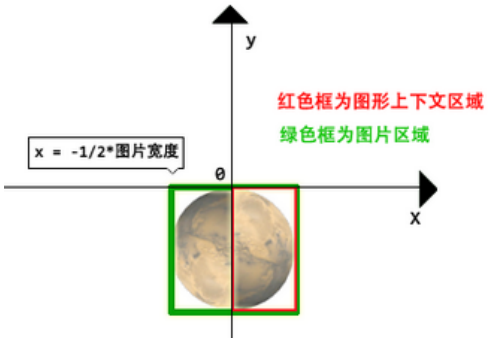

图3 UIImage裁剪原理

CGImage常用的绘图操作

UIImage的Core Graphics版本是CGImage（具体类型是CGImageRef）。两者可以直接相互转化: 使用UIImage的CGImage属性可以访问Quartz图片数据；将CGImage作为UIImage方法imageWithCGImage:或initWithCGImage:的参数创建UIImage对象。

一个CGImage对象可以让你获取原始图片中指定区域的图片（也可以获取指定区域外的图片，UIImage却办不到）。

下面的代码展示了将图片拆分成两半，并分别绘制在上下文的左右两边：

```
UIImage* mars = [UIImage imageNamed:@"Mars.png"];

// 抽取图片的左右半边

CGSize sz = [mars size];

CGImageRef marsLeft = CGImageCreateWithImageInRect([mars CGImage],CGRectMake(0,0,sz.width/2.0,sz.height));

CGImageRef marsRight = CGImageCreateWithImageInRect([mars CGImage],CGRectMake(sz.width/2.0,0,sz.width/2.0,sz.height));

// 将每一个CGImage绘制到图形上下文中

UIGraphicsBeginImageContextWithOptions(CGSizeMake(sz.width*1.5, sz.height), NO, 0);

CGContextRef con = UIGraphicsGetCurrentContext();

CGContextDrawImage(con, CGRectMake(0,0,sz.width/2.0,sz.height), marsLeft);

CGContextDrawImage(con, CGRectMake(sz.width,0,sz.width/2.0,sz.height), marsRight);

UIImage* im = UIGraphicsGetImageFromCurrentImageContext();

UIGraphicsEndImageContext();

// 记得释放内存，ARC在这里无效

CGImageRelease(marsLeft);

CGImageRelease(marsRight);
```

你也许发现绘出的图是上下颠倒的！图片的颠倒并不是因为被旋转了。当你创建了一个CGImage并使用CGContextDrawImage方法绘图就会引起这种问题。这主要是因为原始的本地坐标系统（坐标原点在左上角）与目标上下文（坐标原点在左下角）不匹配。有很多方法可以修复这个问题，其中一种方法就是使用CGContextDrawImage方法先将CGImage绘制到UIImage上，然后获取UIImage对应的CGImage，此时就得到了一个倒转的CGImage。当再调用CGContextDrawImage方法，我们就将倒转的图片还原回来了。实现代码如下：

```
CGImageRef flip (CGImageRef im) {
```

```
02.
03.  CGSize sz = CGSizeMake(CGImageGetWidth(im), CGImageGetHeight(im));
04.
05.  UIGraphicsBeginImageContextWithOptions(sz, NO, 0);
06.
07.  CGContextDrawImage(UIGraphicsGetCurrentContext(), CGRectMake(0, 0, sz.width, sz.heig
     ht), im);
08.
09.  CGImageRef result = [UIGraphicsGetImageFromCurrentImageContext() CGImage];
10.
11.  UIGraphicsEndImageContext();
12.
13.  return result;
14.
15.  }
```

现在将之前的代码修改如下：

```
01.  CGContextDrawImage(con, CGRectMake(0,0,sz.width/2.0,sz.height), flip(marsLeft));
02.
03.  CGContextDrawImage(con, CGRectMake(sz.width,0,sz.width/2.0,sz.height), flip(marsRigh
     t));
```

然而，这里又出现了另外一个问题：在双分辨率的设备上，如果我们的图片文件是高分辨率（@2x）版本，上面的绘图就是错误的。原因在于对于UIImage来说，在加载原始图片时使用imageNamed:方法，它会自动根据所在设备的分辨率类型选择图片，并且UIImage通过设置用来适配的scale属性补偿图片的两倍尺寸。但是一个CGImage对象并没有scale属性，它不知道图片文件的尺寸是否为两倍！所以当调用UIImage的CGImage方法，你不能假定所获得的CGImage尺寸与原始UIImage是一样的。在单分辨率和双分辨率下，一个UIImage对象的size属性值都是一样的，但是双分辨率UIImage对应的CGImage是单分辨率UIImage对应的CGImage的两倍大。所以我们需要修改上面的代码，让其在单双分辨率下都可以工作。代码如下：

```
01.  UIImage* mars = [UIImage imageNamed:@"Mars.png"];
02.
03.  CGSize sz = [mars size];
04.
05.  // 转换CGImage并使用对应的CGImage尺寸截取图片的左右部分
06.
07.  CGImageRef marsCG = [mars CGImage];
08.
09.  CGSize szCG = CGSizeMake(CGImageGetWidth(marsCG), CGImageGetHeight(marsCG));
10.
11.  CGImageRef marsLeft = CGImageCreateWithImageInRect(marsCG,CGRectMake(0,0,szCG.width/
     2.0,szCG.height));
12.
13.  CGImageRef marsRight = CGImageCreateWithImageInRect(marsCG, CGRectMake(szCG.width/2.
     0,0,szCG.width/2.0,szCG.height));
14.
15.  UIGraphicsBeginImageContextWithOptions(CGSizeMake(sz.width*1.5, sz.height), NO, 0);
16.
17.  //剩下的和之前的代码一样，修复倒置问题
18.
19.  CGContextRef con = UIGraphicsGetCurrentContext();
20.
21.  CGContextDrawImage(con, CGRectMake(0,0,sz.width/2.0,sz.height),flip(marsLeft));
22.
23.  CGContextDrawImage(con, CGRectMake(sz.width,0,sz.width/2.0,sz.height),flip(marsRight
     ));
24.
25.  UIImage* im = UIGraphicsGetImageFromCurrentImageContext();
26.
27.  UIGraphicsEndImageContext();
28.
29.  CGImageRelease(marsLeft);
30.
31.  CGImageRelease(marsRight);
```

上面的代码初看上去很繁杂，不过不用担心，这里还有另一种修复倒置问题的方案。相对于使用flip函数，你可以在绘图之前将CGImage包装进UIImage中，这样做有两大优点：

1.当UIImage绘图时它会自动修复倒置问题

2.当你从CGImage转化为Uimage时，可调用imageWithCGImage:scale:orientation:方法生成CGImage作为对缩放性的补偿。

所以这是一个解决倒置和缩放问题的自包含方法。

代码如下：

```
UIImage* mars = [UIImage imageNamed:@"Mars.png"];

CGSize sz = [mars size];

CGImageRef marsCG = [mars CGImage];

CGSize szCG = CGSizeMake(CGImageGetWidth(marsCG), CGImageGetHeight(marsCG));

CGImageRef marsLeft = CGImageCreateWithImageInRect(marsCG, CGRectMake(0,0,szCG.width
/2.0,szCG.height));

CGImageRef marsRight = CGImageCreateWithImageInRect(marsCG, CGRectMake(szCG.width/2.
0,0,szCG.width/2.0,szCG.height));

UIGraphicsBeginImageContextWithOptions(CGSizeMake(sz.width*1.5, sz.height), NO, 0);

[[UIImage imageWithCGImage:marsLeft scale:[mars scale] orientation:UIImageOrientatio
nUp] drawAtPoint:CGPointMake(0,0)];

[[UIImage imageWithCGImage:marsRight scale:[mars scale] orientation:UIImageOrientati
onUp] drawAtPoint:CGPointMake(sz.width,0)];

UIImage* im = UIGraphicsGetImageFromCurrentImageContext();

UIGraphicsEndImageContext();

CGImageRelease(marsLeft); CGImageRelease(marsRight);
```

还有另一种解决倒置问题的方案是在绘制CGImage之前，对上下文应用变换操作，有效地倒置上下文的内部坐标系统。这里先不做讨论。

为什么会发生倒置问题
究其原因是因为Core Graphics源于Mac OS X系统，在Mac OS X中，坐标原点在左下方并且正y坐标是朝上的，而在iOS中，原点坐标是在左上方并且正y坐标是朝下的。在大多数情况下，这不会出现任何问题，因为图形上下文的坐标系统是会自动调节补偿的。但是创建和绘制一个CGImage对象时就会暴露出倒置问题。

CIFilter与CIImage
CIFilter与CIImage是iOS 5新引入的，虽然它们已在MAX OS X系统中存在多年。前缀“CI”表示Core Image，这是一种使用数学滤镜变换图片的技术。但是你不要去幻想iOS提供了像Photoshop软件那样强大的滤镜功能。使用Core Image之前你需要将CoreImage.framework框架导入到你的target之中。

所谓滤镜指的是CIFilter类，滤镜可被分为以下几类：

模板与渐变类
这两类滤镜创建的CIImage可以和其他的CIImage进行合并，比如一种单色，一个棋盘，条纹，亦或是渐变。

合成类
此类滤镜可以将一张图片与另外的图片合并，合成滤镜模式常见于图形处理软件Photoshop中。

色彩类
此滤镜调整、修改图片的色彩。因此你可以改变一张图片的饱和度、色度、亮度、对比度、伽马、白点、曝光度、阴影、高亮等属性。

几何变换类
此类滤镜可对图片执行基本的几何变换，比如缩放、旋转、裁剪。

CIFilter使用起来非常的简单。CIFilter看上去就像一个由键值组成的字典。它生成一个CIImage对象作为其输出。一般地，一个滤镜有一个或多个输入，而对于部分滤镜，生成的图片是基于其他类型的参数值。CIFilter

对象是一个集合，可使用键值对进行检索。通过提供滤镜的字符串名称创建一个滤镜，如果想知道有哪些滤镜，可以查询苹果的Core Image Filter Reference文档，或是调用CIFilter的类方法filterNamesInCategories：，参数值为nil。每一个滤镜拥有一小部分用来确定其行为的键值。如果你想修改某一个键（比如亮度键）对应的值，你可以调用setValue：forKey：方法或当你指定一个滤镜名时提供所有键值对。

需要处理的图片必须是CIImage类型，调用initWithCGImage：方法可获得CIImage。因为CGImage又是作为滤镜的输出，因此滤镜之间可被连接在一起（将滤镜的输出作为initWithCGImage：方法的输入参数）

当你构建一个滤镜链时，并没有做复杂的运算。只有当整个滤镜链需要输出一个CGImage时，密集型计算才会发生。调用contextWithOptions：和createCGImage: fromRect:方法创建CIContext。与以往不同的地方是CIImage没有frame与bounds属性；只有extent属性。你将非常频繁的使用这个属性作为createCGImage: fromRect:方法的第二个参数。

接下来我将演示Core Image的使用。首先创建一个径向渐变的滤镜，该滤镜是从白到黑的渐变方式，白色区域的半径默认是100。接着将其与一张使用CIDarkenBlendMode滤镜的图片合成。CIDarkenBlendMode的作用是背景图片样本将被源图片的黑色部分替换掉。

代码如下：

```
01.  UIImage* moi = [UIImage imageNamed:@"Mars.jpeg"];
02.
03.  CIImage* moi2 = [[CIImage alloc] initWithCGImage:moi.CGImage];
04.
05.  CIFilter* grad = [CIFilter filterWithName:@"CIRadialGradient"];
06.
07.  CIVector* center = [CIVector vectorWithX:moi.size.width / 2.0 Y:moi.size.height / 2.
     0];
08.
09.  // 使用setValue：forKey：方法设置滤镜属性
10.
11.  [grad setValue:center forKey:@"inputCenter"];
12.
13.  // 在指定滤镜名时提供所有滤镜键值对
14.
15.  CIFilter* dark = [CIFilter filterWithName:@"CIDarkenBlendMode" keysAndValues:@"input
     Image", grad.outputImage, @"inputBackgroundImage", moi2, nil];
16.
17.  CIContext* c = [CIContext contextWithOptions:nil];
18.
19.  CGImageRef moi3 = [c createCGImage:dark.outputImage fromRect:moi2.extent];
20.
21.  UIImage* moi4 = [UIImage imageWithCGImage:moi3 scale:moi.scale orientation:moi.image
     Orientation];
22.
23.  CGImageRelease(moi3);
```

图4 图片合成快照

这个例子可能没有什么吸引人的地方，因为所有一切都可以使用Core Graphics完成。除了Core Image是使用GPU处理，可能有点吸引人。Core Graphics也可以做到径向渐变并使用混合模式合成图片。但Core Image要简单得多，特别是当你有多个图片输入想重用一个滤镜链时。并且Core Image的颜色调整功能比Core Graphics更加强大。对了，Core Image还能实现自动人脸识别哦！

绘制一个**UIView**

绘制一个UIVIew最灵活的方式就是由它自己完成绘制。实际上你不是绘制一个UIView，你只是子类化了UIView并赋予子类绘制自己的能力。当一个UIVIew需要执行绘图操作的时， drawRect:方法就会被调用。覆盖此方法让你获得绘图操作的机会。当drawRect：方法被调用，当前图形上下文也被设置为属于视图的图形上下文。你可以使用Core Graphics或UIKit提供的方法将图形画到该上下文中。

你不应该手动调用drawRect：方法！如果你想调用drawRect：方法更新视图，只需发送setNeedsDisplay方法。这将使得drawRect：方法会在下一个适当的时间调用。当然，不要覆盖drawRect：方法除非你知道这样做绝对合法。比方说，在UIImageView子类中覆盖drawRect：方法是不合法的，你将得不到你绘制的图形。

在UIView子类的drawRect：方法中无需调用super，因为本身UIView的drawRect：方法是空的。为了提高一些绘图性能，你可以调用setNeedsDisplayInRect方法重新绘制视图的子区域，而视图的其他部分依然保持不变。

一般情况下，你不应该过早的进行优化。绘图代码可能看上去非常的繁琐，但它们是非常快的。并且iOS绘图系统自身也是非常高效，它不会频繁调用drawRect：方法，除非迫不得已（或调用了setNeedsDisplay方法）。一旦一个视图已由自己绘制完成，那么绘制的结果会被缓存下来留待重用，而不是每次重头再来。(苹果公司将缓存绘图称为视图的位图存储回填（bitmap backing store）)。你可能会发现drawRect：方法中的代码在整个应用程序生命周期内只被调用了一次！事实上，将代码移到drawRect：方法中是提高性能的普遍做法。这是因为绘图引擎直接对屏幕进行渲染相对于先是脱屏渲染然后再将像素拷贝到屏幕要来的高效。

当视图的backgroundColor为nil并且opaque属性为YES，视图的背景颜色就会变成黑色。

Core Graphics上下文属性设置

当你在图形上下文中绘图时，当前图形上下文的相关属性设置将决定绘图的行为与外观。因此，绘图的一般过程是先设定好图形上下文参数，然后绘图。比方说，要画一根红线，接着画一根蓝线。那么首先需要将上下文的线条颜色属性设定为为红色，然后画红线；接着设置上下文的线条颜色属性为蓝色，再画出蓝线。表面上看,红线和蓝线是分开的，但事实上，在你画每一条线时，线条颜色却是整个上下文的属性。无论你用的是UIKit方法还是Core Graphics函数。

因为图形上下文在每一时刻都有一个确定的状态，该状态概括了图形上下文所有属性的设置。为了便于操作这些状态，图形上下文提供了一个用来持有状态的栈。调用CGContextSaveGState函数，上下文会将完整的当前状态压入栈顶；调用CGContextRestoreGState函数，上下文查找处在栈顶的状态，并设置当前上下文状态为栈顶状态。

因此一般绘图模式是：在绘图之前调用CGContextSaveGState函数保存当前状态，接着根据需要设置某些上下文状态，然后绘图，最后调用CGContextRestoreGState函数将当前状态恢复到绘图之前的状态。要注意的是，CGContextSaveGState函数和CGContextRestoreGState函数必须成对出现，否则绘图很可能出现意想不到的错误，这里有一个简单的做法避免这种情况。代码如下：

```
01.  - (void)drawRect:(CGRect)rect {
02.
03.  CGContextRef ctx = UIGraphicsGetCurrentContext();
04.
05.  CGContextSaveGState(ctx);
06.
07.  {
08.
09.  // 绘图代码
10.
11.  }
12.
13.  CGContextRestoreGState(ctx);
14.
15.      }
```

但你不需要在每次修改上下文状态之前都这样做，因为你对某一上下文属性的设置并不一定会和之前的属性

设置或其他的属性设置产生冲突。你完全可以在不调用保存和恢复函数的情况下先设置线条颜色为红色，然后再设置为蓝色。但在一定情况下，你希望你对状态的设置是可撤销的，我将在接下来讨论这样的情况。

许多的属性组成了一个图形上下文状态，这些属性设置决定了在你绘图时图形的外观和行为。下面我列出了一些属性和对应修改属性的函数；虽然这些函数是关于Core Graphics的，但记住，实际上UIKit同样是调用这些函数操纵上下文状态。

线条的宽度和线条的虚线样式
CGContextSetLineWidth、CGContextSetLineDash

线帽和线条联接点样式
CGContextSetLineCap、CGContextSetLineJoin、CGContextSetMiterLimit

线条颜色和线条模式
CGContextSetRGBStrokeColor、CGContextSetGrayStrokeColor、CGContextSetStrokeColorWithColor、CGContextSetStrokePattern

填充颜色和模式
CGContextSetRGBFillColor,CGContextSetGrayFillColor,CGContextSetFillColorWithColor, CGContextSetFillPattern

阴影
CGContextSetShadow、CGContextSetShadowWithColor

混合模式
CGContextSetBlendMode（决定你当前绘制的图形与已经存在的图形如何被合成）

整体透明度
CGContextSetAlpha（个别颜色也具有alpha成分）

文本属性
CGContextSelectFont、CGContextSetFont、CGContextSetFontSize、CGContextSetTextDrawingMode、CGContextSetCharacterSpacing

是否开启反锯齿和字体平滑
CGContextSetShouldAntialias、CGContextSetShouldSmoothFonts

另外一些属性设置：

裁剪区域:在裁剪区域外绘图不会被实际的画出来。

变换（或称为“CTM“，意为当前变换矩阵): 改变你随后指定的绘图命令中的点如何被映射到画布的物理空间。

许多这些属性设置接下来我都会举例说明。

路径与绘图
通过编写移动虚拟画笔的代码描画一段路径，这样的路径并不构成一个图形。绘制路径意味着对路径描边或填充该路径，也或者两者都做。同样，你应该从某些绘图程序中得到过相似的体会。

一段路径是由点到点的描画构成。想象一下绘图系统是你手里的一只画笔，你首先必须要设置画笔当前所处的位置，然后给出一系列命令告诉画笔如何描画随后的每段路径。每一段新增的路径开始于当前点，当完成

一条路径的描画，路径的终点就变成了当前点。

下面列出了一些路径描画的命令：

定位当前点
CGContextMoveToPoint

描画一条线
CGContextAddLineToPoint、CGContextAddLines

描画一个矩形
CGContextAddRect、CGContextAddRects

描画一个椭圆或圆形
CGContextAddEllipseInRect

描画一段圆弧
CGContextAddArcToPoint、CGContextAddArc

通过一到两个控制点描画一段贝赛尔曲线
CGContextAddQuadCurveToPoint、CGContextAddCurveToPoint

关闭当前路径
CGContextClosePath 这将从路径的终点到起点追加一条线。如果你打算填充一段路径，那么就不需要使用该命令，因为该命令会被自动调用。

描边或填充当前路径
CGContextStrokePath、CGContextFillPath、CGContextEOFillPath、CGContextDrawPath。对当前路径描边或填充会清除掉路径。如果你只想使用一条命令完成描边和填充任务，可以使用CGContextDrawPath命令，因为如果你只是使用CGContextStrokePath对路径描边，路径就会被清除掉，你就不能再对它进行填充了。

创建路径并描边路径或填充路径只需一条命令就可完成的函数：CGContextStrokeLineSegments、CGContextStrokeRect、CGContextStrokeRectWithWidth、CGContextFillRect、CGContextFillRects、CGContextStrokeEllipseInRect、CGContextFillEllipseInRect。

一段路径是被合成的，意思是它是由多条独立的路径组成。举个例子，一条单独的路径可能由两个独立的闭合形状组成：一个矩形和一个圆形。当你在构造一条路径的中间过程（意思是在描画了一条路径后没有调用描边或填充命令，或调用CGContextBeginPath函数来清除路径）调用CGContextMoveToPoint函数，就像是你拾起画笔，并将画笔移动到一个新的位置，如此来准备开始一段独立的相同路径。如果你担心当你开始描画一条路径的时候，已经存在的路径和新的路径会被认为是已存在路径的一个合成部分，你可以调用CGContextBeginPath函数指定你绘制的路径是一条独立的路径；苹果的许多例子都是这样做的，但在实际开发中我发现这是非必要的。

CGContextClearRect函数的功能是擦除一个区域。这个函数会擦除一个矩形内的所有已存在的绘图；并对该区域执行裁剪。结果像是打了一个贯穿所有已存在绘图的孔。

CGContextClearRect函数的行为依赖于上下文是透明还是不透明。当在图形上下文中绘图时，这会尤为明显和直观。如果图片上下文是透明的（UIGraphicsBeginImageContextWithOptions第二个参数为NO），那么CGContextClearRect函数执行擦除后的颜色为透明，反之则为黑色。

当在一个视图中直接绘图（使用drawRect：或drawLayer：inContext：方法），如果视图的背景颜色为nil或

颜色哪怕有一点点透明度，那么CGContextClearRect的矩形区域将会显示为透明的，打出的孔将穿过视图包括它的背景颜色。如果背景颜色完全不透明，那么CGContextClearRect函数的结果将会是黑色。这是因为视图的背景颜色决定了是否视图的图形上下文是透明的还是不透明的。

图5 CGContextClearRect函数的应用

如图5，在左边的蓝色正方形被挖去部分留为黑色，然而在右边的蓝色正方形也被挖去部分留为透明。但这两个正方形都是UIView子类的实例，采用相同的绘图代码！不同之处在于视图的背景颜色，左边的正方形的背景颜色在nib文件中

但是这却完全改变了CGContextClearRect函数的效果。UIView子类的drawRect：方法看起来像这样：

```
CGContextRef con = UIGraphicsGetCurrentContext();

CGContextSetFillColorWithColor(con, [UIColor blueColor].CGColor);

CGContextFillRect(con, rect);

CGContextClearRect(con, CGRectMake(0,0,30,30));
```

为了说明典型路径的描画命令，我将生成一个向上的箭头图案，我谨慎避免使用便利函数操作，也许这不是创建箭头最好的方式，但依然清楚的展示了各种典型命令的用法。

图6 一个简单的路径绘图

```
CGContextRef con = UIGraphicsGetCurrentContext();

// 绘制一个黑色的垂直黑色线，作为箭头的杆子

CGContextMoveToPoint(con, 100, 100);

CGContextAddLineToPoint(con, 100, 19);

CGContextSetLineWidth(con, 20);

CGContextStrokePath(con);

// 绘制一个红色三角形箭头

CGContextSetFillColorWithColor(con, [[UIColor redColor] CGColor]);

CGContextMoveToPoint(con, 80, 25);

CGContextAddLineToPoint(con, 100, 0);

CGContextAddLineToPoint(con, 120, 25);

CGContextFillPath(con);

// 从箭头杆子上裁掉一个三角形，使用清除混合模式

CGContextMoveToPoint(con, 90, 101);

CGContextAddLineToPoint(con, 100, 90);

CGContextAddLineToPoint(con, 110, 101);
```

```
32.
33.  CGContextSetBlendMode(con, kCGBlendModeClear);
34.
35.  CGContextFillPath(con);
```

确切的说，为了以防万一，我们应该在绘图代码周围使用CGContextSaveGState和CGContextRestoreGState函数。可对于这个例子来说，添加与否不会有任何的区别。因为上下文在调用drawRect：方法中不会被持久，所以不会被破坏。

如果一段路径需要重用或共享，你可以将路径封装为CGPath（具体类型是CGPathRef）。你可以创建一个新的CGMutablePathRef对象并使用多个类似于图形的路径函数的CGPath函数构造路径，或者使用CGContextCopyPath函数复制图形上下文的当前路径。有许多CGPath函数可用于创建基于简单几何形状的路径（CGPathCreateWithRect、CGPathCreateWithEllipseInRect）或基于已存在路径（CGPathCreateCopyByStrokingPath、CGPathCreateCopyDashingPath、CGPathCreateCopyByTransformingPath）。

UIKit的UIBezierPath类包装了CGPath。它提供了用于绘制某种形状路径的方法，以及用于描边、填充、存取某些当前上下文状态的设置方法。类似地，UIColor提供了用于设置当前上下文描边与填充的颜色。因此我们可以重写我们之前绘制箭头的代码：

```
01.  UIBezierPath* p = [UIBezierPath bezierPath];
02.
03.  [p moveToPoint:CGPointMake(100,100)];
04.
05.  [p addLineToPoint:CGPointMake(100, 19)];
06.
07.  [p setLineWidth:20];
08.
09.  [p stroke];
10.
11.  [[UIColor redColor] set];
12.
13.  [p removeAllPoints];
14.
15.  [p moveToPoint:CGPointMake(80,25)];
16.
17.  [p addLineToPoint:CGPointMake(100, 0)];
18.
19.  [p addLineToPoint:CGPointMake(120, 25)];
20.
21.  [p fill];
22.
23.  [p removeAllPoints];
24.
25.  [p moveToPoint:CGPointMake(90,101)];
26.
27.  [p addLineToPoint:CGPointMake(100, 90)];
28.
29.  [p addLineToPoint:CGPointMake(110, 101)];
30.
31.  [p fillWithBlendMode:kCGBlendModeClear alpha:1.0];
```

在这种特殊情况下，完成同样的工作并没有节省多少代码，但是UIBezierPath仍然还是有用的。如果你需要对象特性，UIBezierPath提供了一个便利方法：bezierPathWithRoundedRect：cornerRadius：，它可用于绘制带有圆角的矩形，如果是使用Core Graphics就相当冗长乏味了。还可以只让圆角出现在左上角和右上角。

```
01.  - (void)drawRect:(CGRect)rect {
02.
03.    CGContextRef ctx = UIGraphicsGetCurrentContext();
04.
05.    CGContextSetStrokeColorWithColor(ctx, [UIColor blackColor].CGColor);
06.
07.    CGContextSetLineWidth(ctx, 3);
08.
09.    UIBezierPath *path;
10.
11.    path = [UIBezierPath bezierPathWithRoundedRect:CGRectMake(100, 100, 100, 100) byRo
     undingCorners:(UIRectCornerTopLeft |UIRectCornerTopRight) cornerRadii:CGSizeMake(10,
      10)];
12.
13.    [path stroke];
14.
```

```
15. }
```

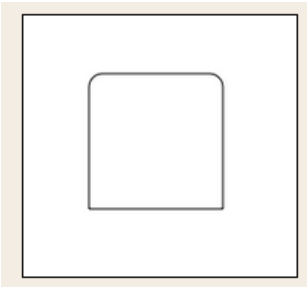

图7 左右圆角矩形

裁剪

路径的另一用处是遮蔽区域，以防对遮蔽区域进一步绘图。这种用法被称为裁剪。裁剪区域外的图形不会被绘制到。默认情况下，一个图形上下文的裁剪区域是整个图形上下文。你可在上下文中的任何地方绘图。

总的来说，裁剪区域是上下文的一个特性。与已存在的裁剪区域相交会出现新的裁剪区域。所以如果你应用了你自己的裁剪区域，稍后将它从图形上下文中移除的做法是使用CGContextSaveGState和CGContextRestoreGState函数将代码包装起来。

为了便于说明这一点，我使用裁剪而不是使用混合模式在箭头杆子上打孔的方法重写了生成箭头的代码。这样做有点小复杂，因为我们想要裁剪区域不在三角形内而在三角形外部。为了表明这一点，我们使用了一个三角形和一个矩形组成了一个组合路径。

当填充一个组合路径并使用它表示一个裁剪区域时，系统遵循以下两规则之一：

环绕规则（Winding rule）

如果边界是顺时针绘制，那么在其内部逆时针绘制的边界所包含的内容为空。如果边界是逆时针绘制，那么在其内部顺时针绘制的边界所包含的内容为空。

奇偶规则

最外层的边界代表内部都有效，都要填充；之后向内第二个边界代表它的内部无效，不需填充；如此规则继续向内寻找边界线。我们的情况非常简单，所以使用奇偶规则就很容易了。这里我们使用CGContextEOClip设置裁剪区域然后进行绘图。（如果不是很明白，可以参见这篇文章：五种方法绘制有孔的2d形状）

```
01. CGContextRef con = UIGraphicsGetCurrentContext();
02.
03. // 在上下文裁剪区域中挖一个三角形状的孔
04.
05. CGContextMoveToPoint(con, 90, 100);
06.
07. CGContextAddLineToPoint(con, 100, 90);
08.
09. CGContextAddLineToPoint(con, 110, 100);
10.
11. CGContextClosePath(con);
12.
13. CGContextAddRect(con, CGContextGetClipBoundingBox(con));
14.
15. // 使用奇偶规则，裁剪区域为矩形减去三角形区域
16.
17. CGContextEOClip(con);
18.
19. // 绘制垂线
20.
21. CGContextMoveToPoint(con, 100, 100);
22.
23. CGContextAddLineToPoint(con, 100, 19);
24.
25. CGContextSetLineWidth(con, 20);
26.
```

```
CGContextStrokePath(con);

// 画红色箭头

CGContextSetFillColorWithColor(con, [[UIColor redColor] CGColor]);

CGContextMoveToPoint(con, 80, 25);

CGContextAddLineToPoint(con, 100, 0);

CGContextAddLineToPoint(con, 120, 25);

CGContextFillPath(con);
```

渐变

渐变可以很简单也可以很复杂。一个简单的渐变（接下来要讨论的）由一端点的颜色与另一端点的颜色决定，如果在中间点加入颜色（可选），那么渐变会在上下文的两个点之间线性的绘制或在上下文的两个圆之间放射状的绘制。不能使用渐变作为路径的填充色，但可使用裁剪限制对路径形状的渐变。

我重写了绘制箭头的代码，箭杆使用了线性渐变。效果如图7所示。

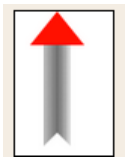

图8 箭头杆子渐变

```
CGContextRef con = UIGraphicsGetCurrentContext();

CGContextSaveGState(con);

// 在上下文裁剪区域挖一个三角形孔

CGContextMoveToPoint(con, 90, 100);

CGContextAddLineToPoint(con, 100, 90);

CGContextAddLineToPoint(con, 110, 100);

CGContextClosePath(con);

CGContextAddRect(con, CGContextGetClipBoundingBox(con));

CGContextEOClip(con);

//绘制一个垂线，让它的轮廓形状成为裁剪区域

CGContextMoveToPoint(con, 100, 100);

CGContextAddLineToPoint(con, 100, 19);

CGContextSetLineWidth(con, 20);

// 使用路径的描边版本替换图形上下文的路径

CGContextReplacePathWithStrokedPath(con);

// 对路径的描边版本实施裁剪

CGContextClip(con);

// 绘制渐变

CGFloat locs[3] = { 0.0, 0.5, 1.0 };

CGFloat colors[12] = {

0.3,0.3,0.3,0.8, // 开始颜色，透明灰

0.0,0.0,0.0,1.0, // 中间颜色，黑色

0.3,0.3,0.3,0.8 // 末尾颜色，透明灰

};

```

```
CGColorSpaceRef sp = CGColorSpaceCreateDeviceGray();

CGGradientRef grad = CGGradientCreateWithColorComponents (sp, colors, locs, 3);

CGContextDrawLinearGradient(con, grad, CGPointMake(89,0), CGPointMake(111,0), 0);

CGColorSpaceRelease(sp);

CGGradientRelease(grad);

CGContextRestoreGState(con); // 完成裁剪

// 绘制红色箭头

CGContextSetFillColorWithColor(con, [[UIColor redColor] CGColor]);

CGContextMoveToPoint(con, 80, 25);

CGContextAddLineToPoint(con, 100, 0);

CGContextAddLineToPoint(con, 120, 25);

CGContextFillPath(con);
```

调用CGContextReplacePathWithStrokedPath函数假装对当前路径描边，并使用当前线段宽度和与线段相关的上下文状态设置。但接着创建的是描边路径外部的一个新的路径。因此，相对于使用粗的线条，我们使用了一个矩形区域作为裁剪区域。

虽然过程比较冗长但是非常的简单；我们将渐变描述为一组在一端点（0.0）和另一端点（1.0）之间连续区上的位置，以及设置与每个位置相对应的颜色。为了提亮边缘的渐变，加深中间的渐变，我使用了三个位置，黑色点的位置是0.5。为了创建渐变，还需要提供一个颜色空间。最后，我创建出了该渐变，并对裁剪区域绘制线性渐变，最后释放了颜色空间和渐变。

颜色与模板
在iOS中，CGColor表示颜色（具体类型为CGColorRef）。使用UIColor的colorWithCGColor：和CGColor方法可bridged cast到UIColor。

在iOS中，模板表示为CGPattern（具体类型为CGPatternRef）。你可以创建一个模板并使用它进行描边或填充。其过程是相当复杂的。作为一个非常简单的例子，我将使用红蓝相间的三角形替换箭头的三角形部分。
现在移除下面行：
CGContextSetFillColorWithColor（con， [UIColor redColor].CGColor））；

在被移除的地方填入下面代码：

```
CGColorSpaceRef sp2 = CGColorSpaceCreatePattern(NULL);

CGContextSetFillColorSpace (con, sp2);

CGColorSpaceRelease (sp2);

CGPatternCallbacks callback = {0, &drawStripes, NULL };

CGAffineTransform tr = CGAffineTransformIdentity;

CGPatternRef patt = CGPatternCreate(NULL,CGRectMake(0,0,4,4), tr, 4, 4, kCGPatternTi
lingConstantSpacingMinimalDistortion, true, &callback);

CGFloat alph = 1.0;

CGContextSetFillPattern(con, patt, &alph);

CGPatternRelease(patt);
```

代码非常冗长，但它却是一个完整的样板。现在我们从后往前分析代码: 我们调用CGContextSetFillPattern不是设置填充颜色，我们设置的是填充的模板。函数的第三个参数是一个指向CGFloat的指针，所以我们事先设置CGFloat自身。第二个参数是一个CGPatternRef对象，所以我们需要事先创建CGPatternRef，并在最后释放它。

现在开始讨论CGPatternCreate。一个模板是在一个矩形元中的绘图。我们需要矩形元的尺寸（第二个参数）以及矩形元原始点之间的间隙（第四和第五个参数）。这这种情况下，矩形元是4*4的，每一个矩形元与它的周围矩形元是紧密贴合的。我们需要提供一个应用到矩形元的变换参数（第三个参数）；在这种情况下，我们不需要变换做什么工作，所以我们应用了一个恒等变换。我们应用了一个瓷砖规则（第六个参数）。我们需要声明的是颜色模板不是漏印（stencil）模板，所以参数值为true。并且我们需要提供一个指向回调函数的指针，回调函数的工作是向矩形元绘制模板。第八个参数是一个指向CGPatternCallbacks结构体的指针。这个结构体由数字0和两个指向函数的指针构成。第一个函数指针指向的函数当模板被绘制到矩形元中被调用，第二个函数指针指向的函数当模板被释放后调用。第二个函数指针我们没有指定，它的存在主要是为了内存管理的需要。但在这个简单的例子中，我们并不需要。

在你使用颜色模板调用CGContextSetFillPattern函数之前，你需要设置将应用到模板颜色空间的上下文填充颜色空间。如果你忽略这项工作，那么当你调用CGContextSetFillPattern函数时会发生错误。所以我们创建了颜色空间，设置它作为上下文的填充颜色空间，并在后面做了释放。

到这里我们仍然没有完成绘图。因为我还没有编写向矩形元中绘图的函数！绘图函数地址被表示为&drawStripes。绘图代码如下所示：

```
01. void drawStripes (void *info, CGContextRef con) {
02.
03. // assume 4 x 4 cell
04.
05. CGContextSetFillColorWithColor(con, [[UIColor redColor] CGColor]);
06.
07. CGContextFillRect(con, CGRectMake(0,0,4,4));
08.
09. CGContextSetFillColorWithColor(con, [[UIColor blueColor] CGColor]);
10.
11. CGContextFillRect(con, CGRectMake(0,0,4,2));
12.
13. }
```

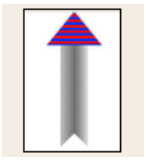

图9 模板填充

如你所见，实际的模板绘图代码是非常简单的。唯一的复杂点在于CGPatternCreate函数必须与模板绘图函数的矩形元尺寸相同。我们知道矩形元的尺寸为4*4，所以我们用红色填充它，并接着填充它的下半部分为绿色。当这些矩形元被水平垂直平铺时，我们得到了如图8所示的条纹图案。

注意，最后图形上下文遗留下了一个不可取的状态，即填充颜色空间被设置为了一个模板颜色空间。如果稍后尝试设置填充颜色为常规颜色，就会引起错误。通常的解决方案是，使用CGContextSaveGState和CGContextRestoreGState函数将代码包起来。

你可能观察到图8的平铺效果并不与箭头的三角形内部相符合：最底部的似乎只平铺了一半蓝色。这是因为一个模板的定位并不关心你填充（描边）的形状，总的来说它只关心图形上下文。我们可以调用CGContextSetPatternPhase函数改变模板的定位。

图形上下文变换

就像UIView可以实现变换，同样图形上下文也具备这项功能。然而对图形上下文应用一个变换操作不会对已在图形上下文上的绘图产生什么影响，它只会影响到在上下文变换之后被绘制的图形，并改变被映射到图形上下文区域的坐标方式。一个图形上下文变换被称为CTM，意为“当前变换矩阵“（current transformation ma

trix）。

完全利用图形上下文的CTM来免于即使是简单的计算操作是很常见的。你可以使用CGContextConcatCTM函数将当前变换乘上任何CGAffineTransform，还有一些便利函数可对当前变换应用平移、缩放，旋转变换。

当你获得上下文的时候，对图形上下文的基本变换已经设置好了；这就是系统能映射上下文绘图坐标到屏幕坐标的原因。无论你对当前变换应用了什么变换，基本变换变换依然有效并且绘图继续工作。通过将你的变换代码封装到CGContextSaveGState和CGContextRestoreGState函数调用中，对基本变换应用的变换操作可以被还原。

举个例子，对于我们迄今为止使用代码绘制的向上箭头来说，已知的放置箭头的方式仅仅只有一个位置：箭头矩形框的左上角被硬编码在坐标{80，0}。这样代码很难理解、灵活性差、且很难被重用。最明智的做法是通过将所有代码中的x坐标值减去80，让箭头矩形框左上角在坐标{0，0}。事先应用一个简单的平移变换，很容易将箭头画在任何位置。为了映射坐标到箭头的左上角，我们使用下面代码：

CGContextTranslateCTM（con, 80, 0）; //在坐标{0,0}处绘制箭头

旋转变换特别的有用，它可以让你在一个被旋转的方向上进行绘制而无需使用任何复杂的三角函数。然而这略有点复杂，因为旋转变换围绕的点是原点坐标。这几乎不是你所想要的，所以你先是应用了一个平移变换，为的是映射原点到你真正想绕其旋转的点。但是接着，在旋转之后，为了算出你在哪里绘图，你可能需要做一次逆向平移变换。
为了说明这个做法，我将绕箭头杆子尾部旋转多个角度重复绘制箭头，并把对箭头的绘图封装为UIImage对象。接着我们简单重复绘制UIImage对象。
具体代码如下：

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

UIGraphicsBeginImageContextWithOptions(CGSizeMake(40,100), NO, 0.0);

CGContextRef con = UIGraphicsGetCurrentContext();

CGContextSaveGState(con);

CGContextMoveToPoint(con, 90 - 80, 100);

CGContextAddLineToPoint(con, 100 - 80, 90);

CGContextAddLineToPoint(con, 110 - 80, 100);

CGContextMoveToPoint(con, 110 - 80, 100);

CGContextAddLineToPoint(con, 100 - 80, 90);

CGContextAddLineToPoint(con, 90 - 80, 100);

CGContextClosePath(con);

CGContextAddRect(con, CGContextGetClipBoundingBox(con));

CGContextEOClip(con);

CGContextMoveToPoint(con, 100 - 80, 100);

CGContextAddLineToPoint(con, 100 - 80, 19);

CGContextSetLineWidth(con, 20);

CGContextReplacePathWithStrokedPath(con);

CGContextClip(con);

CGFloat locs[3] = { 0.0, 0.5, 1.0 };

CGFloat colors[12] = {

0.3,0.3,0.3,0.8,

0.0,0.0,0.0,1.0,

0.3,0.3,0.3,0.8

};
```

```
48.
49.  CGColorSpaceRef sp = CGColorSpaceCreateDeviceGray();
50.
51.  CGGradientRef grad = CGGradientCreateWithColorComponents (sp, colors, locs, 3);
52.
53.  CGContextDrawLinearGradient (con, grad, CGPointMake(89 - 80,0), CGPointMake(111 - 80
     ,0), 0);
54.
55.  CGColorSpaceRelease(sp);
56.
57.  CGGradientRelease(grad);
58.
59.  CGContextRestoreGState(con);
60.
61.  CGColorSpaceRef sp2 = CGColorSpaceCreatePattern(NULL);
62.
63.  CGContextSetFillColorSpace (con, sp2);
64.
65.  CGColorSpaceRelease (sp2);
66.
67.  CGPatternCallbacks callback = {0, &drawStripes, NULL };
68.
69.  CGAffineTransform tr = CGAffineTransformIdentity;
70.
71.  CGPatternRef patt = CGPatternCreate(NULL,CGRectMake(0,0,4,4),tr,4,4, kCGPatternTiling
     ConstantSpacingMinimalDistortion,true, &callback);
72.
73.  CGFloat alph = 1.0;
74.
75.  CGContextSetFillPattern(con, patt, &alph);
76.
77.  CGPatternRelease(patt);
78.
79.  CGContextMoveToPoint(con, 80 - 80, 25);
80.
81.  CGContextAddLineToPoint(con, 100 - 80, 0);
82.
83.  CGContextAddLineToPoint(con, 120 - 80, 25);
84.
85.  CGContextFillPath(con);
86.
87.  UIImage* im = UIGraphicsGetImageFromCurrentImageContext();
88.
89.  UIGraphicsEndImageContext();
90.
91.  con = UIGraphicsGetCurrentContext();
92.
93.  [im drawAtPoint:CGPointMake(0,0)];
94.
95.  for (int i=0; i<3; i++) {
96.
97.  CGContextTranslateCTM(con, 20, 100);
98.
99.  CGContextRotateCTM(con, 30 * M_PI/180.0);
100.
101. CGContextTranslateCTM(con, -20, -100);
102.
103. [im drawAtPoint:CGPointMake(0,0)];
104.
105. }
106.
107. }
```

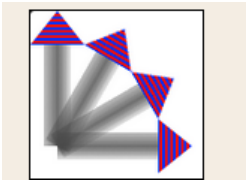

图10 使用CTM旋转变换

变换有多个方法解决我们早期使用CGContextDrawImage函数遇到的倒置问题。相对于逆向绘图，我们选择逆向我们绘图的上下文。实质上，我们对上下文坐标系统应用了一个“倒置”变换。你自上而下移动上下文，接着你通过应用一个让y坐标乘以-1的缩放变换逆向y坐标的方向。

```
01.  CGContextTranslateCTM(con, 0, theHeight);
02.
03.  CGContextScaleCTM(con, 1.0, -1.0);
```

上下文的顶部应该被你往下移动多远依赖于你绘制的图片。比如说我们可以绘制没有倒置问题的两个半边的火星图形（前面讨论的一个例子）。

```
CGContextTranslateCTM(con, 0, sz.height); // sz为[mars size]

CGContextScaleCTM(con, 1.0, -1.0);

CGContextDrawImage(con, CGRectMake(0, 0, sz.width/2.0, sz.height), marsLeft);

CGContextDrawImage(con, CGRectMake(b.size.width-sz.width/2.0, 0, sz.width/2.0, sz.he
ight),marsRight);
```

阴影

为了在绘图上加入阴影，可在绘图之前设置上下文的阴影值。阴影的位置表示为CGSize，如果CGSize的两个值都是正数，则表示阴影是朝下和朝右的。模糊度被表示为任何一个正数。苹果没有解释缩放的工作方式，但实验表明12是最佳的模糊度，99及以上的模糊度会让阴影变得不成形。

我在图9的基础上给上下文加了一个阴影：

```
con = UIGraphicsGetCurrentContext();

CGContextSetShadow(con, CGSizeMake(7, 7), 12);

[im drawAtPoint:CGPointMake(0,0)];
```

然而，使用这种方法有一个不太明显的问题。我们是在每绘制一个箭头的时候加上的阴影。因此，箭头的阴影会投射在另一个箭头上面。我们想要的是让所有的箭头集体地投射出一个阴影。解决方法是使用一个透明的图层；该图层类似一个先是叠加所有绘图然后加上阴影的一个子上下文。代码如下：

```
con = UIGraphicsGetCurrentContext();

CGContextSetShadow(con, CGSizeMake(7, 7), 12);

CGContextBeginTransparencyLayer(con, NULL);

[im drawAtPoint:CGPointMake(0,0)];

for (int i=0; i<3; i++) {

CGContextTranslateCTM(con, 20, 100);

CGContextRotateCTM(con, 30 * M_PI/180.0);

CGContextTranslateCTM(con, -20, -100);

[im drawAtPoint:CGPointMake(0,0)];

}

// 在调用了CGContextEndTransparencyLayer函数之后，

// 图层内容会在应用全局alpha和上下文阴影状态之后被合成到上下文中

CGContextEndTransparencyLayer(con);
```

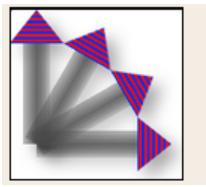

图11 阴影效果

点与像素

一个点是由xy坐标描述的一个无穷小量的位置。通过指定点实现在图形上下文中的绘图。我们并没有关心设备的分辨率，因为Core Graphics已经精细地将绘图映射到物理输出设备（基于CTM、反锯齿和平滑技术）。因此，文章之前的讨论只关心图形上下文的点，不关注点与屏幕像素的关系。

然而像素是真实存在的。一个像素是真实世界中一个具有完整物理尺寸的显示单元。整数的点实际上介于像素之间。在单分辨率设备上，这可能会让人感到迷惑。比方说，如果使用线宽为1的线条对一个整数坐标的垂直路径描边，那么线条将会被分为两半，分别落在路径的两侧。所以在单分辨率设备上线宽会变成2px（因为设备无法表示半个像素）。

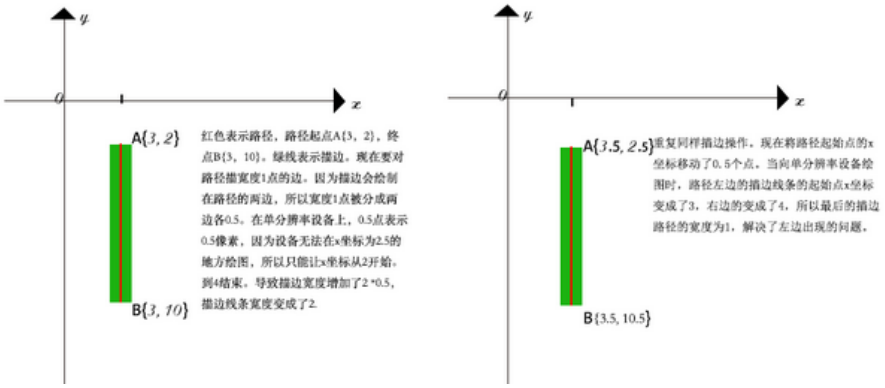

图12 整数的点坐标与偏移0.5点的坐标对应的描边处理

当你遇到显示效果不佳的时，可能会被建议通过对坐标增减0.5让它在像素中居中。这个建议可能有效，如图11。但它只是做了一些头脑简单的假设。一个复杂的做法是获得UIView的contentScaleFactor属性。这个值为1.0或2.0，所以你可以除以这个属性值得到从像素到点的转换。还可以想想用最精确的方式绘制一条水平或垂直的线条的方式不是描边路径，而是填充路径。使用这种方法UIView的子类代码将可以在任何设备上绘制一条完美的1px宽的垂线，代码如下：

```
01.  CGContextFillRect(con, CGRectMake(100,0,1.0/self.contentScaleFactor,100));
```

内容模式

一个视图向它自身绘图，相对于只有背景颜色和子视图，它还有内容。这意味着每当视图被调整大小它的contentMode属性就变得非常重要。正如我之前提到的，绘图系统会尽可能避免重头开始绘制视图。相反，绘图系统将使用之前绘图操作的缓存结果（位图回填）。所以，如果视图被重新调整大小，系统可能简单的伸缩或重定位缓存绘图，前提是你的contentMode设置指令是是这样设置的。

说明这一点略有点复杂。因为我需要安排调整视图大小而不引起重绘操作（调用drawRect：方法）。当程序启动时，我将创建一个MyView实例，并将它放在window上。接着将执行调整MyView尺寸的操作延迟到window出现和界面初次显示之后：

```
01.  - (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDi
     ctionary *)launchOptions {
02.
03.    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
04.
05.    self.window.rootViewController = [UIViewController new];
06.
07.    MyView* mv =[[MyView alloc] initWithFrame:CGRectMake(0, 0, self.window.bounds.size
     .width - 50, 150)];
08.
09.    mv.center = self.window.center;
10.
11.    [self.window.rootViewController.view addSubview: mv];
12.
13.    mv.opaque = NO;
14.
15.    mv.tag = 111; // so I can get a reference to this view later
16.
17.    [self performSelector:@selector(resize:) withObject:nil afterDelay:0.1];
18.
19.    self.window.backgroundColor = [UIColor whiteColor];
20.
21.    [self.window makeKeyAndVisible];
22.
23.    return YES;
24.
25.  }
```

我们将视图的高度调成之前的2倍。没有触发drawRect：方法的调用。如果我们视图的drawRect：方法代码和生成图9的代码相同，则我们得到如图12的结果，视图被显示在正确高度上。

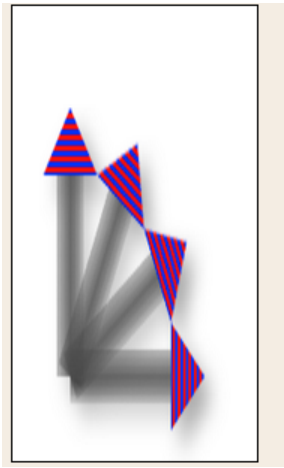

图13 内容自动伸展

可是早晚drawRect：方法会被调用，绘图将按照drawRect：方法中的代码被刷新。代码不会将箭头绘制在相对于视图边界的高度。它是在一个固定的高度。因此箭头会伸展，而且会在以后某个时间返回到原始的尺寸。

通常我们的视图的contentMode属性需要与视图绘制自己的方式一致。假设我们的drawRect：方法中的代码让箭头的尺寸和位置相对于视图的边界原点，即它的左上方。所以我们可以设置它的contentMode为UIViewContentModeTopLeft。又或者，我们可以将contentMode设置为UIVIewContentModeRedraw，这将引起缓存内容的自动缩放和重定位被关闭，最终结果是视图的setNeedsDisplay方法将被调用，触发drawRect：方法重绘视图内容。

在另一方面，如果一个视图只是暂时被调整大小。假设是作为动画的一部分，那么伸缩行为正是你所想要的。假设我们的动画是想要让视图变大然后还原回原始大小以达到作为吸引用户的一种手段。这就需要视图伸缩的时候视图的内容也跟着伸缩，正确的contentMode的值是UIViewContentModeScaleToFill，被伸缩的内容仅仅是视图内容的一副缓存图片，所以它运行起来十分的高效。

完。

译者说明：译文中的错误或不当之处望不吝指出。

Drop me a line: xdreamarshal@gmail.com, http://weibo.com/xdream86

---

(81)

看过此文章的用户还看过

论坛源码推荐（5月20日）：把AGGeometryKit和POP结合起来使用 文件数据库CUSFile Storag

BCMeshTransformView实现了三种形式的过渡，拉开“幕帘”展示隐藏的内容、缩放展示需要突出的内容（点击屏幕上的位置则放大该处内容），以及果冻胶状的交互方式（...

访问人数：3375 查看详细

如何在iOS地图上高效的显示大量数据

原文：How To Efficiently Display Large Amounts of Data on iOS Maps 如何在iOS地图上以用户可以理解并乐于接受的方式来处理和显示大量数据？这个教程将会给大...

访问人数：11666 查看详细

理解Frame

Frame是布局的核心。每个开发者都使用frame定位和改变UIView和CALayer的大小。在本文中我将把焦点集中在CALayer上，因为它是UIView的底层实现，view.frame简单的...

访问人数：4563 查看详细

7 条评论

最新 最早 最热

haha

在第三种和第四种绘图形式的时候[myview setNeedsDisplay]进入不了DrawLayer 但是[mylayer setNeedsDisplay]就可以进入代理方面了。

2014年12月16日 回复 顶 转发

天冷堆雪人

回复 haha: 

2015年3月10日 回复 顶 转发

小酒

请问如何获得图形的绘制路径

2015年8月13日 回复 顶 转发

繁华落尽

上下文 翻译得太难懂了, 应该翻译成背景/画布

1月9日 回复 顶 转发

陈天石

mark一下先

1月19日 回复 顶 转发

ksflying

mark

2月1日 回复 顶 转发

111

mark

3月31日 回复 顶 转发

社交帐号登录: 微信 微博 QQ 人人 更多»