Core Animation:

Simplified Animation Techniques for Mac and iPhone Development

第三部分

核心动画的层

第十二章

性能

版本 1.0

翻译时间：2012-12-17

DevDiv 翻译：animeng
DevDiv 校对：symbian_love BeyondVincent(破船)
DevDiv 编辑：BeyondVincent(破船)

写在前面

目前，移动开发被广大的开发者们看好，并大量的加入移动领域的开发。

鉴于以下原因：

- 国内的相关中文资料缺乏
- 许多开发者对 E 文很是感冒
- 电子版的文档利于技术传播和交流

DevDiv.com 移动开发论坛特此成立了翻译组，翻译组成员具有丰富的移动开发经验和英语翻译水平。组员们利用业余时间，把一些好的相关英文资料翻译成中文，为广大移动开发者尽一点绵薄之力，希望能对读者有些许作用，在此也感谢组员们的辛勤付出。

关于 DevDiv

DevDiv 已成长为国内最具人气的综合性移动开发社区

更多相关信息请访问 DevDiv 移动开发论坛。

技术支持

首先 DevDiv 翻译组对您能够阅读本文以及关注 DevDiv 表示由衷的感谢。

在您学习和开发过程中，或多或少会遇到一些问题。DevDiv 论坛集结了一流的移动专家，我们很乐意与您一起探讨移动开发。如果您有什么问题和技术需要支持的话，请访问网站 www.devdiv.com 或者发送邮件到 BeyondVincent@DevDiv.com，我们将尽力所能及的帮助你。

关于本文的翻译

感谢 animeng 对本文的翻译，同时非常感谢 symbian_love 和 BeyondVincent(破船)在百忙中抽出时间对翻译初稿的认真校验。才使本文与读者尽快见面。由于书稿内容多，我们的知识有限，尽管我们进行了细心的检查，但是还是会存在错误，这里恳请广大读者批评指正，并发送邮件至 Vincent@devdiv.com，在此我们表示衷心的感谢。

推荐资源

iOS

iOS 5 Programming Cookbook 中文翻译各章节汇总

iOS6 新特征：参考资料和示例汇总

Android

DEVDIV 原创 ANDROID 学习系列教程实例

Windows Phone

Windows Phone 8 新特征讲义与示例汇总

Windows 8

Building Windows 8 apps with XAML and C#中文翻译全部汇总

Building Windows 8 apps with HTML5 and JavaScript 中文翻译汇总

Windows 8 Metro 开发书籍汇总

Windows 8 Metro App 开发 Step by Step

其它

DevDiv 出版作品汇总

目录

本书翻译贴各章汇总

第 12 章 性能

核心动画在设计的时候就考虑了性能。它首先是层级别的呈现，并且设计运行在小型的设备上（iphone 和 itouch），这些设备内存有限，并且 cpu 和 gpu 不如桌面电脑上的强大，核心动画是被设计的比较高效的，但是并不意味着你就可以在代码中随便用。

任何复杂的系统都会考虑性能的问题。幸运的是核心动画在处理复杂动画时，已经帮你处理的很多性能问题，你也需要改善一下代码让基于核心动画的应用程序具有更好的性能提升。

这一章展示给你的是，如何来充分利用核心动画。本章的开始会有一些指南。这些都是你应该记住的，当你做核心动画的程序时，你应该通过这些指南精炼你的代码。然后你会了解到如何平衡 GPU，怎么用多线程的动画，并且使用 CATiledLayer 呈现大的图像给用户，使你的应用程序不至于图像太大而陷入困境。

12.1.1. 硬件加速

核心动画的一个重要的好处就是它利用了 mac 的硬件的优点。先前，开发者需要在应用上做动画时，动画产生需要在 CPU 上，并且会耗掉 cpu 的循环，这样就会使程序慢。制作用户界面动画会消耗大量的时间，通常还要涉及到 OpenGL 或其他的一些技术与图形处理单元（GPU）的工作。使用核心动画，你会能够自由的使用硬件加速。你仅仅需要做的就是定义一个动画，开启它，让它运行。你不必担心装载到 GPU 中，因为核心动画都帮你自动处理了。

当您在设计用户界面时，要牢记。那些你认为是一个复杂的动画，但是放到 GPU 上执行可能微不足道。 GPU 是专门来做屏幕上矩形的移动，三维空间转化等。只有当 GPU 的限制产生时，我们开始看到性能下降。

12.1.2. 最小原则

下面的段落所呈现的原则，都是你需要在做核心动画时，需要牢记的规则。

避免幕后渲染

不管你是在桌面应用程序还是在 touch 设备上，你需要限制你的绘制，仅仅当这些区域对于用户可见时再绘制。幸运的是，-drawRect:方法只有当矩形区域为脏时，才会传递过来，因此你可以精确的控制每个循环的绘画次数。

限制滤镜和阴影

滤镜和阴影需要耗费大量的时间来计算和渲染在核心动画上。因而，建议尽量保持小并且有可能的话，避免上述的情况。这有一些技巧：

尤其是滤镜的情况，它可能要渲染一个静态的，临时的图像。在动画假设被渲染时，这个静态的图像可能被使用来代替需要动画渲染的层。当动画完成时，渲染的层就被交换到了后面。虽然这不能被应用到所有的情况，这也会产生性能的优化。这些动作可以通过在最上层调用-drawInContext 来创建这个用来交换的静态图片。

阴影也是代价很高的。因为他们属性部分透明的层，它需要大量的计算，来决定每个像素（因为每个像素都需要计算，直到有不透明的层遇到。如果阴影重叠的话，就增加了消耗。考虑限制只有最外层的阴影，并允许内层不产生任何阴影。参见“最小 alpha 混合”在本章的后面。

明智的使用变换

转换提供了一个很好的方式，给用户一个视觉感受，应用程序怎么改变并且变成了什么样。从窗口滑出或者滑入的方式，给用户了一个明显的视觉感触。然而例如要转换滤镜和阴影话，这是对性能的一个很大考验。

因而，在设计应用程序时，你应该考虑在可视层中有多少转换或者它们应该转换多快。

避免嵌套转换

作为核心动画的强大之处，可以彼此同时转换多个层。例如，你可以在一些层中转换层中所有 z 轴。然而，要注意到这些转换都是实时执行的，没有缓存。因而，当你移动或者让层做动画时，这个层有了很多层的转换，每个转换都需要动画重新计算它们的帧。

为了增加性能，避免使用多层级的转换，当动画运行时。

限制透明度的混合

前面讨论了嵌套转换，透明度混合也是实时计算的。当一个层是部分透明时，透明部分的动画就要在动画帧中重新计算。记住这些，当动画层中有透明时，尽量减少计算。滑动层之后的层是透明的会导致极大的性能消耗。

幸运的是，有个方法可以找到那个层有透明度混合，从而移除它。对于 iphone 这种有限的资源它是非常有用的。来核对透明度混合，启动 iphone 下的应用程序，使用 instruments。在里面增加核心动画 instrument。

当你的应用程序在 iphone 上运行时并且你也附带 instrument 到程序上，那么转换到核心动画的 instruments 上，颜色混合层就会展示如图 12-1.

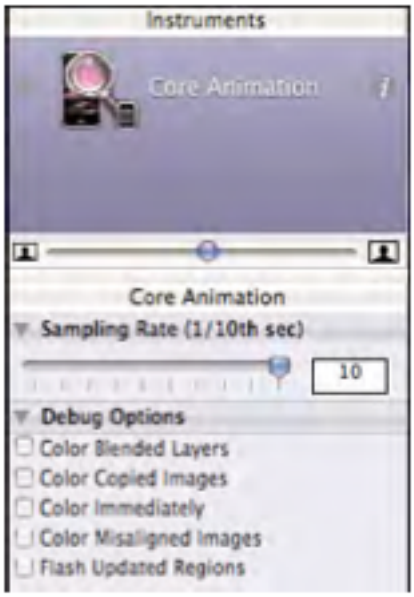


图 12-1 颜色混合层的指令转换

只要运行，你立马就可以看到效果。整个 iphone 程序被分成红色和绿色。事实上，这里没用应用程序运行，你可以看到 iphone 主屏幕的效果，如图 12-2

12-2 颜色混合层模式在 iphone 上

当可用时，不同的颜色混合就会展示不同的颜色，因此你能快速的定位到应用程序那些有问题。绿色罩住的区域是没有颜色混合的，而红色的是有的。目标就是来减少红色区域。例如，在 TransparentCocoaTouch 应用程序中，你可以看到所有的 UILabel 对象都有一个透明背景，如图 12-3

图 12-3 iphone 颜色混合模式开启

当你检查你的代码时，你就会看到那些区域是有问题的。这里看-initWithFrame:reuseIdentifier:中的 CustomTableViewCell 对象，问题就在这里，看清单 12-1

```
- (id)initWithFrame:(CGRect)frame reuseIdentifier:(NSString*)ident {
if (!(self = [super initWithFrame:frame reuseIdentifier:ident])) return nil;
UIView *contentView = [self contentView];
imageView = [[UIImageView alloc] initWithFrame:CGRectMake(5, 5, 64, 64)];
[imageView setContentMode:UIViewContentModeScaleAspectFit];
[contentView addSubview:imageView];
[imageView release];
titleLabel = [[UILabel alloc] initWithFrame:CGRectZero];
[titleLabel setFont:[UIFont boldSystemFontOfSize:14.0f]];
[titleLabel setBackgroundColor:[UIColor clearColor]];
[contentView addSubview:titleLabel];
[titleLabel release];
descriptionLabel = [[UILabel alloc] initWithFrame:CGRectZero];
[descriptionLabel setNumberOfLines:0];
[descriptionLabel setFont:[UIFont systemFontOfSize:6.0f]];
[descriptionLabel setBackgroundColor:[UIColor clearColor]];
[contentView addSubview:descriptionLabel];
[descriptionLabel release];
[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(imageUpdated:)
name:kImageDownloadComplete object:nil];
return self; }
```

就像你看到的，UILabel 对象，titleLabel 和 descriptionLabel，两个的背景颜色都是被设定为[UIColor clearColor]，引起了颜色混合的发生。为了纠正这些，改变 UIColor 和整个 UITableViewCell 的背景颜色一样。这就减少了透明度的混合，提高了性能，如清单 12-2.

```
- (id)initWithFrame:(CGRect)frame reuseIdentifier:(NSString*)ident {
if (!(self = [super initWithFrame:frame reuseIdentifier:ident])) return nil;
UIView *contentView = [self contentView];
imageView = [[UIImageView alloc] initWithFrame:CGRectMake(5, 5, 64, 64)];
[imageView setContentMode:UIViewContentModeScaleAspectFit];
[contentView addSubview:imageView];
[imageView release];
titleLabel = [[UILabel alloc] initWithFrame:CGRectZero];
[titleLabel setFont:[UIFont boldSystemFontOfSize:14.0f]];
[titleLabel setBackgroundColor:[UIColor whiteColor]];
[contentView addSubview:titleLabel];
[titleLabel release];
descriptionLabel = [[UILabel alloc] initWithFrame:CGRectZero];
[descriptionLabel setNumberOfLines:0];
[descriptionLabel setFont:[UIFont systemFontOfSize:6.0f]];
[descriptionLabel setBackgroundColor:[UIColor whiteColor]];
[contentView addSubview:descriptionLabel];
[descriptionLabel release];
[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(imageUpdated:)
name:kImageDownloadComplete object:nil];
return self; }
```

清单 12-2

平铺层

核心动画有一个强大的功能，使您可以快速查看应用程序的详细图像。CATileLayer 是被设计成来展示大图像，但是不用把整个图像导入到内存中，因而引起了性能的提升。

为了演示 CATiledLayer 这个有用的特征，打开 TiledLayers 示例应用程序。在这个应用程序中一个很大的图像(6064*4128)是被导入并且还具有放大缩小的能力。正常情况下，一个很大的图像是被导入到内存中，这样会引起性能问题。然而，通过导入图像到 CATiledLayer 中，核心动画来控制所有的内存问题。核心动画会装载图像需要展示的一部分而非整个图像。你需要配置的就是在 CATileLayer 上给它一些细节的东西，就是根据尺寸来指定缩放的等级。

在 TiledLayer 应用程序中，一个简单的 CATileLayer 是在主窗口中初始化并且分配了一个 NSSegmentedControl 来管理缩放等级。结果窗口展示如图 12-4

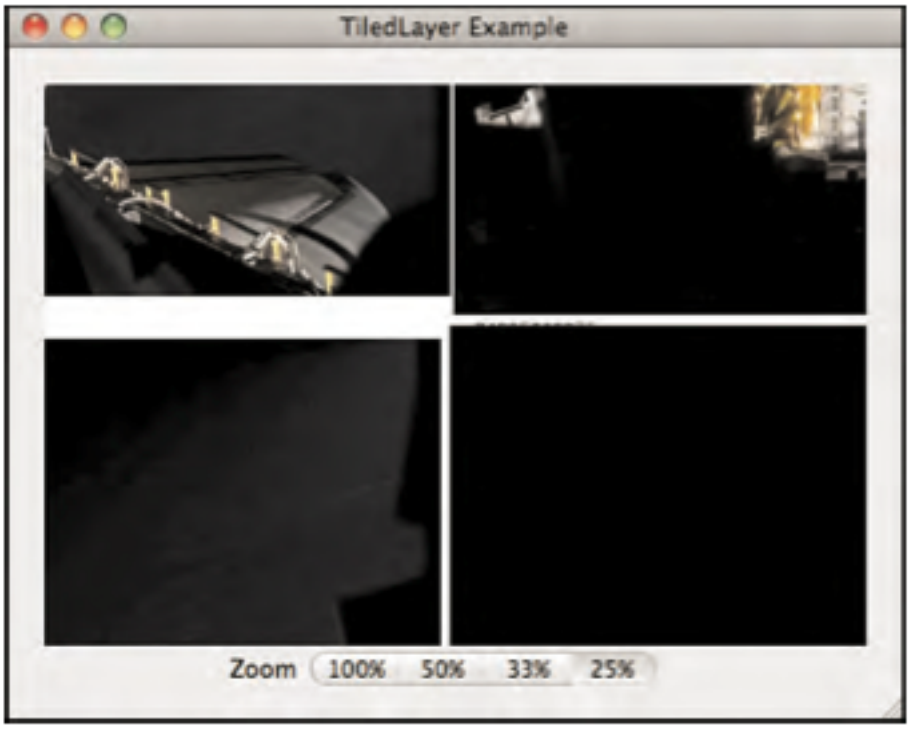


图 12-4 平铺层应用

当界面是被设计时，构造 CATiledLayer 在-awakfromNib 方法中，如清单 12-3

```
- (void)awakeFromNib {
NSString *imagePath = [[NSBundle mainBundle] pathForResource:@"BigSpaceImage" ofType:@"png"];
NSData *data = [NSData dataWithContentsOfFile:imagePath];
image = [[CIImage imageWithData:data] retain];
tiledLayer = [CATiledLayer layer];
[tiledLayer setBounds:CGRectMake(0, 0, 6064, 4128)];
float midX = NSMidX(NSRectFromCGRect([[view layer] frame]));
float midY = NSMidY(NSRectFromCGRect([[view layer] frame]));
[tiledLayer setPosition:CGPointMake(midX, midY)];
[tiledLayer setLevelsOfDetail:4]; //number of levels [tiledLayer setTileSize:CGSizeMake(256, 256)];
[tiledLayer setDelegate:self];
[[view layer] addSublayer:tiledLayer]; }
```

清单 12-3

当应用程序获得图像的路径时，那个路径通过 CIImage 的调用被导入然后存储起来用。CATileLayer 是被初始化然后初始化将要展示的图像边框。下面就定位图像的位置并且配置可用缩放的等级。最后一步就是吧 AppDelegate 作为代理分配给该层，以便于接收绘图事件。

NSSegmentedControl 是绑定在-zoom:方法上，每当 segmented control 改变状态时就发送事件。选择段的位置是被使用来决定图像目前的缩放比例，在 CATileLayer 上展示的图像。这些可以通过它的 sublayerTransform 的 CATransform3DMakeScale 调用来实现，在段控制器上传递一个 x 和 y 的值，清单 12-4

```
- (IBAction)zoom:(id)sender {
CGFloat zoom = 1.0 / ([sender selectedSegment] + 1);
[[view layer] setSublayerTransform:CATransform3DMakeScale(zoom, zoom, 1.0)];
}
```

清单 12-4

最后你要实现的方法就是 CATiledLayer 的一个代理。这个方法可以使我们重载-drawInContext:方法，而不用继承 CALayer。在默认的-drawInContext:方法实现中，它会寻找一个代理，核对代理是否实现了-drawLayer:inContext:方法。如果这个情况为真，那么-

drawLayer:inContext 就会被调用，如清单 12-5.

```
- (void)drawLayer:(CALayer*)layer inContext:(CGContextRef)context {
[[CIContext contextWithCGContext:context options:nil] drawImage:image atPoint:CGPointMake(0, 0)
fromRect:CGRectMake(0, 0, 6064, 4128)];
}
```

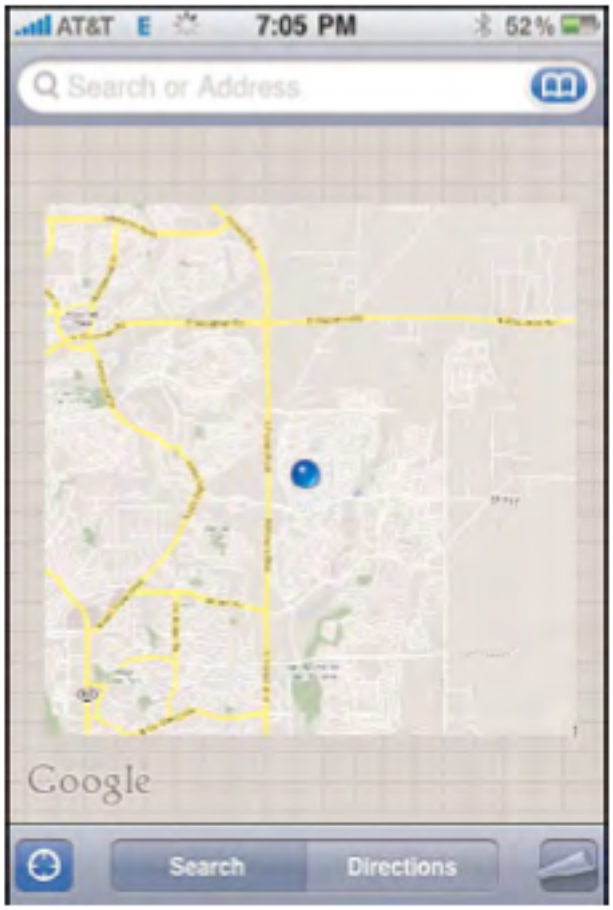
清单 12-5

这个方法使用了在-awakFromNib 中初始化的图像，并且使用了图像尺寸的边框绘制了它到了图像上下文中。应该注意的是，你不需要控制任何的缩放计算，变换等方法。CATileLayer 都自动帮你控制了。

这些如何工作

CATiledLayer 导入图像成快，名字就是由此而来。每个块都响应一个图像的细节，根据你设定的块的尺寸。当你要求某个图像的细节 CATileLayer 会根据原始的图像，缩放到渴望的尺寸，并且在屏幕上渲染之前，分割它成不同的块。因为这些块都被 CATiledlayer 缓存下来，所以你可以滚动图像，并且很快的改变图像的细节。

CATiledLayer 会缓存足够多的块，而当它也会根据需要扔掉不同的块。如果将来你再次需要他就会再次的生成。这意味着块的呈现是一个异步的过程，那么用户就会看到延时，之后才会绘制到层上。一个很好的例子就是 google 地图的应用程序，如图 12-5.如果你缩放块的等级，图像不会立刻展示，而是有一些网格组成。当块装载完成时，就会代替网格显示出来。



图像 12-5 google map 应用程序

12.1.3. 多线程的动画

核心动画是线程安全的，因为它是安装多核系统的思想设计的。

这意味着你在多线程中操控核心动画，不会影响到其他的内部数据，这点非常的重要。然而，核心动画有一个地方不是线程安全的，就是当它进入到 CALayer 的属性中时。

如果你的应用程序在主线程进入到属性中时，又在其他线程中进入到该属性，结果是不确定的。它可能的结果是，要么动画没有效果，要么程序 crash。当你要尝试获得一个属性，并且改变它时，你需要先存储属性先前的状态，并且这个行为要加锁，使这个过程具有原子性。如清单 12-6

```
[layer lock]
float opacity = layer.opacity; layer.opacity = opacity + .1;
[layer unlock];
```

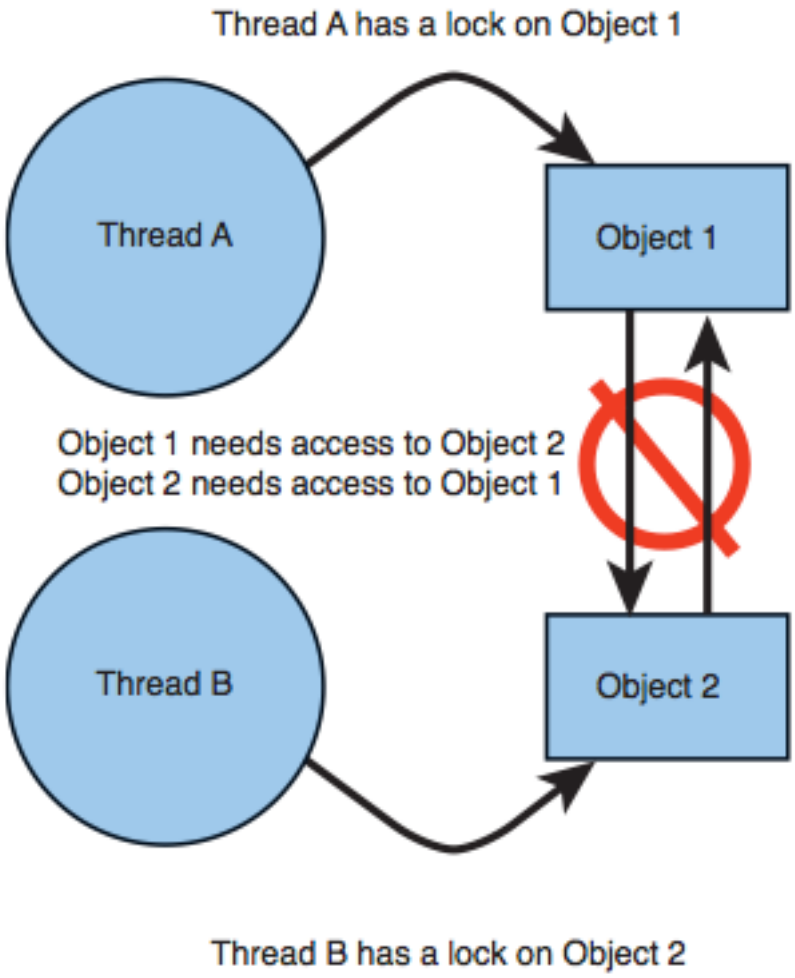
清单 12-6

在这个例子中，[CATransaction lock]用来防止任何其他线程获得锁，会有效的阻碍其他线程。这种序列话的进入到属性中，保证在我们处理的过程中，属性不会被改变。

当需要层工作在多线程中，我们需要注意的如下：

获取一个锁时间太长的话，会让界面停止绘制。因为进入到层是被锁住了，不能在其他线程中进入，系统的其他部分也不能进入直到这个锁被释放了。记住任何时候你锁住了一个层，就要尽快的解锁。

第二个问题是所有线程都遇到的问题，不仅仅是核心动画，那就是死锁。如果你锁住了一个层在一个线程中，这个层需要进入另一个属性，但是这个属性是被另一个线程锁住，那么整个应用程序就会无响应，结果就是程序的 crash。



滤镜的多线程

不像核心动画，核心图像滤镜不是线程安全的。这以为这你应该仅仅在单线程中处理核心图像。然而，也可以通过键值对(KVC)和关键路径在多线程中控制滤镜。例如，如果你有一个层滤镜名字是 acme，并且有一个属性叫做 job，你可以通过下面的代码进行调整

```
[layer setvalue:myValue forKeyPath:@"layer.filters.acme.job"];
```

这将保证改变是原子的，因而是线程安全的。

线程和运行循环

当核心动画在多线程中工作时，你需要意识到的是在任何线程上调用-setNeedsDisplay，都会给层一个-display 的消息。如果标记一个层需要显示，而非在主线程中，那么你就必须等待一个运行循环去确保线程已经运行了-display 的方法。如果不这样，-display 就从来不被调用，因为线程已经终结了，会引起一些异常的效果。

尽管它是可能在线程中调用-setNeedsDisplay，而不在主线程中。但是这对绘制工作影响很小的，但多数情况下建议使用-performSelectorOnMainThread:方法，让它在主线程中运行。

12.1.4. 总结

这一章，我们看到了关于性能方面的一些建议和策略。但是在处理性能之前，建议先完成代码。不要试着花费大量的时间来优化代码，除非性能的问题已经发生。

这里也建议通过一些记录结果来测试一些潜在的瓶颈。有时候，一个简答的 fps(每秒的帧率)记录就可以测试出来瓶颈。利用这些记录，我们就能确定那些性能改变是正确的，代替盲目的找到一些无关紧要的因素。


DevDiv
点击这里访问：DevDiv.com 移动开发论坛