**文档：**

**https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/**

**Instuments**

Instuments是Xcode套件中没有被充分利用的工具，很多iOS开发者从来没用过Instrument

Activity Monitor

（活动监视器）：显示器处理的CPU、内存和网络使用情况统计；

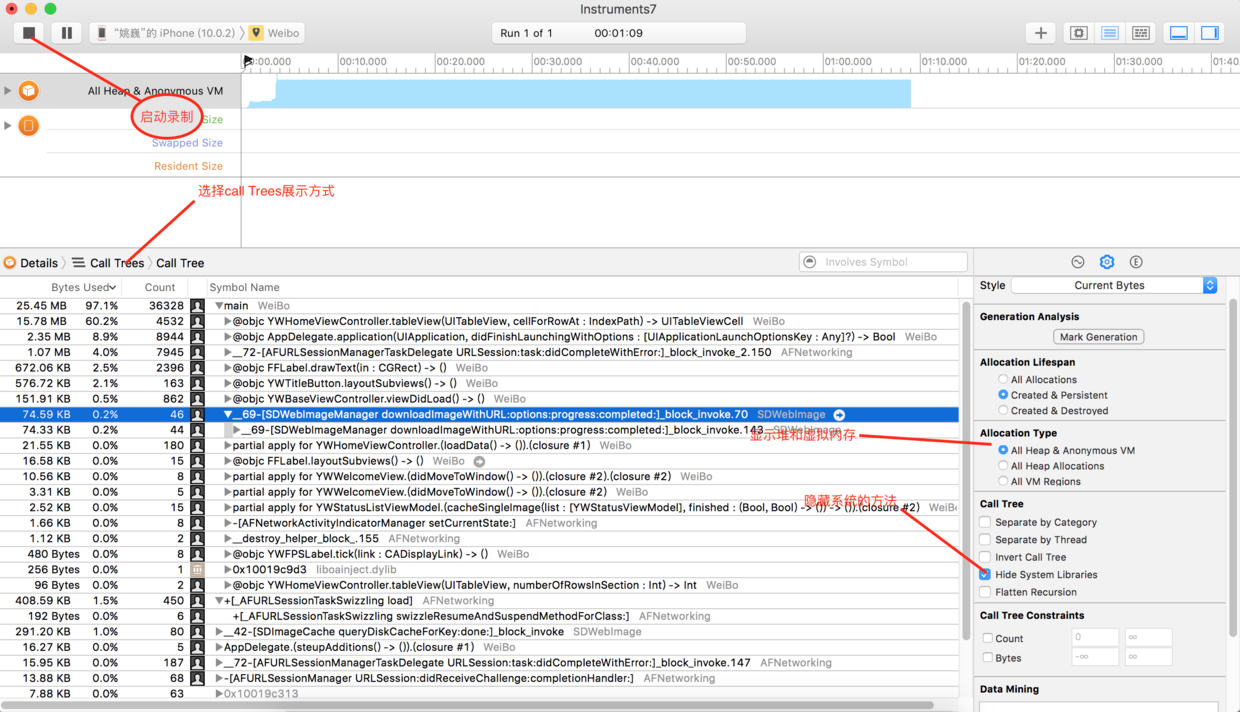
Allocations

（内存分配）：跟踪过程的匿名虚拟内存和堆的对象提供类名和可选保留/释放历史；

管理内存是app开发中最重要的一个方面，对于开发者来说，在程序架构中减少内存的使用通常都是使用Allocations去定位和找出减少内存使用的方式接下来，谈一下内存泄漏的两种情况

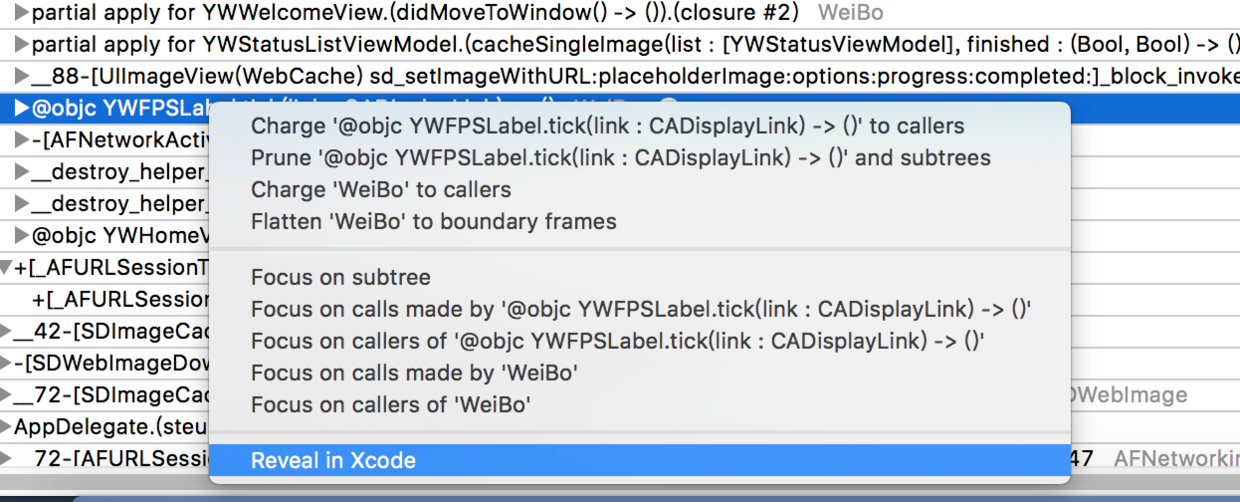
第一种：为对象A申请了内存空间，之后再也没用过对象A，也没释放过A导致内存泄漏，这种是Leaked Memory内存泄漏

第二种：类似于递归，不断地申请内存空间导致的内存泄漏，这种情况是Abandoned Momory此工具可以让开发者很好的了解每个方法占用内存的情况，并定位相关的代码



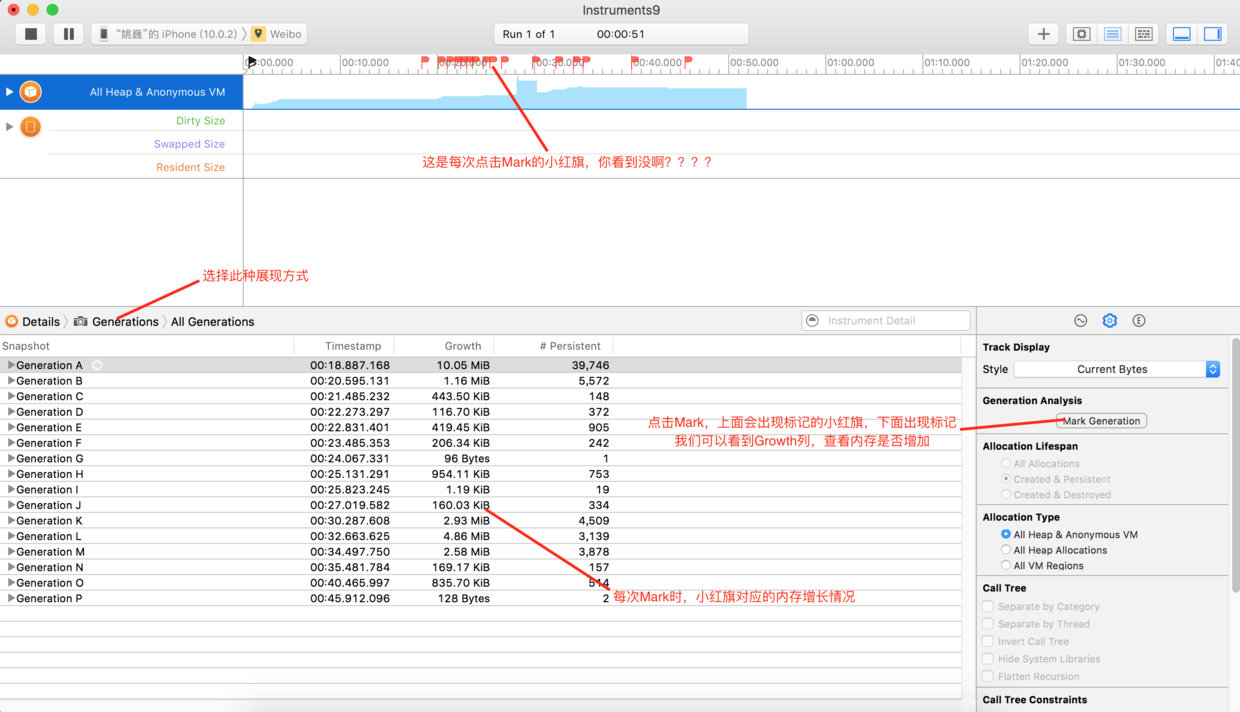
Allocations查看方法占用内存.png

右键就可以打开Xcode自动定位到相关占用内存方法的代码上



定位到相关代码.png

第二种情况可以根据下图的操作清晰的找到对用的代码问题



定位Abandoned Momory.png

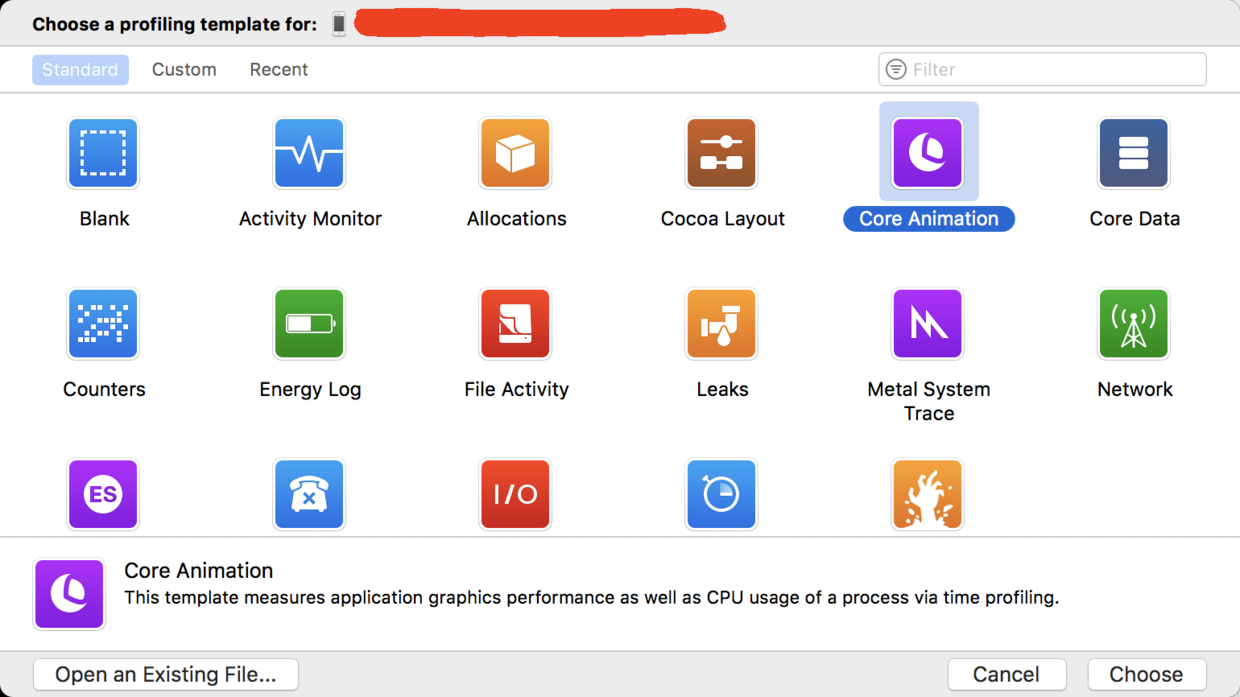
解释一下，第二种情况我们应该如何操作，重复的执行一系列的操作时候内存不会继续增加，比如打开和关闭一个窗口，这样的操作，每一次操作的前后，内存应该是相同的，通过多次循环操作，内存不会递增下去，通过这种分析结果，观察内存分配趋势，当发现不正确的结果或者矛盾的结果，就可以研究是不是Abandoned Momory的问题，并可以修正这个问题了

Cell Trees

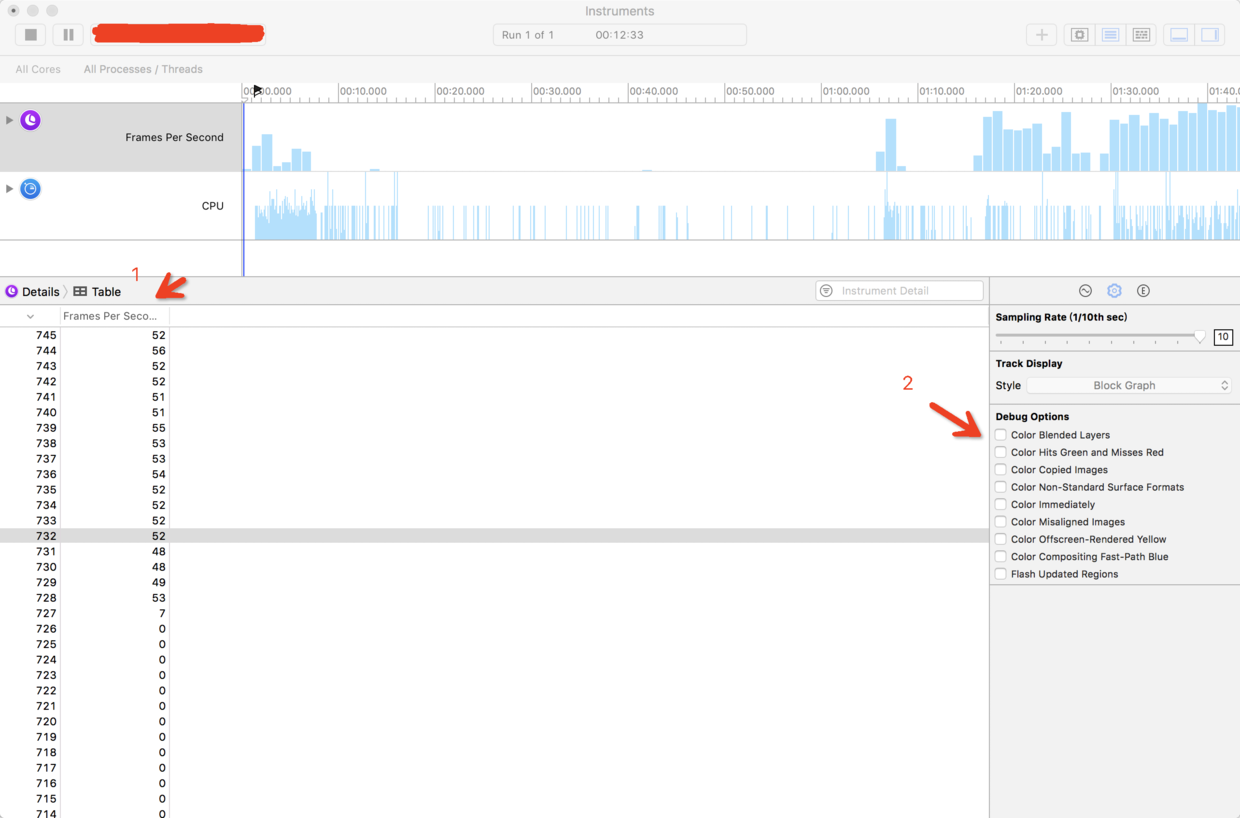
Generations

Core Animation

在性能优化中一个最具参考价值的属性是FPS:全称Frames Per Second,其实就是屏幕刷新率，苹果的iphone推荐的刷新率是60Hz，也就是说GPU每秒钟刷新屏幕60次，这每刷新一次就是一帧frame，FPS也就是每秒钟刷新多少帧画面。静止不变的页面FPS值是0，这个值是没有参考意义的，只有当页面在执行动画或者滑动的时候，FPS值才具有参考价值，FPS值的大小体现了页面的流畅程度高低，当低于45的时候卡顿会比较明显。  
注意点:  
（1）使用真机调试。  
（2）最好使用release包测试（release是发布版本，苹果会在release包中做很多优化工作，因此用release包测试出来的性能才是最真实的）。  
启动程序点击XCode选择左上角-XCode->Open Developer Tool ->Instruments，打开Instruments再选择CoreAnimation：



打开CoreAnimation



CoreAnimation调试界面

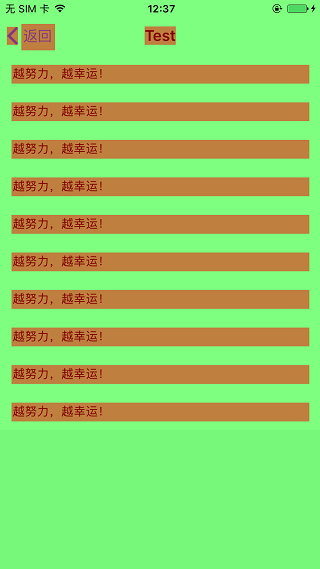
图中1是FPS值。  
图中2是不同纬度的调试选项（下面会逐个介绍）。

**Color Blended Layers (图层混合)**

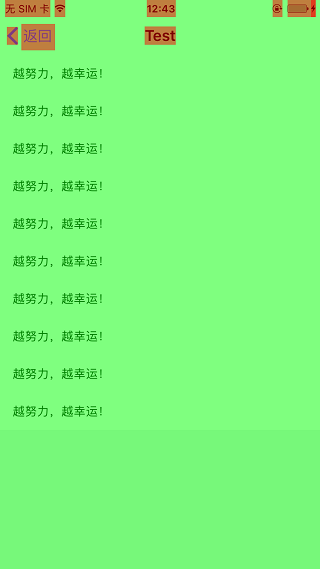
这个选项是检测哪里发生了图层混合，先介绍一下什么是图层混合？很多情况下，界面都是会出现多个UI控件叠加的情况，如果有透明或者半透明的控件，那么GPU会去计算这些这些layer最终的显示的颜色，也就是我们肉眼所看到的效果。例如一个上层Veiw颜色是绿色RGB(0,255,0)，下层又放了一个View颜色是红色RGB(0,0,255)，透明度是50%，那么最终显示到我们眼前的颜色是蓝色RGB(0,127.5,127.5)。这个计算过程会消耗一定的GPU资源损耗性能。如果我们把上层的绿色View改为不透明， 那么GPU就不用耗费资源计算，直接显示绿色。混合颜色计算公式：

|  |  |
| --- | --- |
| 1 | R(C)=alpha\*R(B)+(1-alpha)\*R(A)    R(x)、G(x)、B(x)分别指颜色x的RGB分量 |

如果出现图层混合了，打开Color Blended Layers选项，那块区域会显示红色，所以我们调试的目的就是将红色区域消减的越少越好。那么如何减少红色区域的出现呢？只要设置控件不透明即可。  
（1）设置opaque 属性为true。  
（2）给View设置一个不透明的颜色，没有特殊需要设置白色即可。  
如果你在lldb中po打印某个控件，你会发现打印出来的数据中，控件的opaque都是true，因为控件这个属性的默认值都是true，所以第一种方法可以直接忽略掉。使用第二种方法你会发现之前红色的都消除掉了。



设置不透明之前



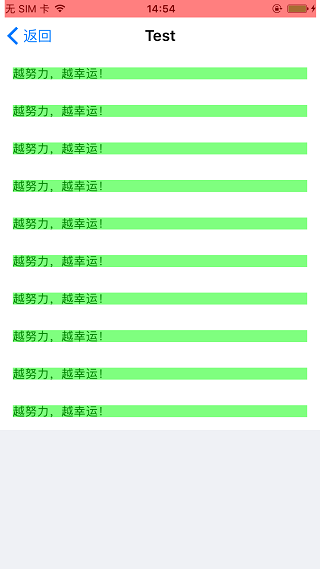
设置不透明之后

|  |  |
| --- | --- |
| 1  2 | label.backgroundColor = [UIColor whiteColor];  label.layer.masksToBounds = YES; |

到这里你可能奇怪，设置label的背景色第一行不就够了么，为什么还有第二行？这是因为如果label的内容是中文，label实际渲染区域要大于label的size，最外层多了一个sublayer，如果不设置第二行label的边缘外层灰出现图层混合的红色，因此需要在label内容是中文的情况下加第二句。单独使用label.layer.masksToBounds = YES是不会发生离屏渲染，下文会讲离屏渲染。  
注意点：UIImageView控件比较特殊，不仅需要自身这个容器是不透明的，并且imageView包含的内容图片也必须是不透明的，如果你自己的图片出现了图层混合红色，先检查是不是自己的代码有问题，如果确认代码没问题，就是图片自身的问题，可以联系你们的UI眉眉～

**Color Hits Green and Misses Red（光栅化）**

这个选项主要是检测我们是是否正确使用layer的shouldRasterize属性，shouldRasterize = YES开启光栅化。什么是光栅化？光栅化是将一个layer预先渲染成位图(bitmap)，再加入到缓存中，成功被缓存的layer会标注为绿色,没有成功缓存的会标注为红色，正确使用光栅化可以得到一定程度的性能提升。  
适用情况：一般在图像内容不变的情况下才使用光栅化，例如设置阴影耗费资源比较多的静态内容，如果使用光栅化对性能的提升有一定帮助。  
非适用情况：如果内容会经常变动,这个时候不要开启,否则会造成性能的浪费。例如我们在使用tableViewCell中，一般不要用光栅化，因为tableViewCell的绘制非常频繁，内容在不断的变化，如果使用了光栅化，会造成大量的离屏渲染降低性能。  
如果你在一个界面中使用了光栅化，刚进去这个页面的所有使用了光栅化的控件layer都会是红色，因为还没有缓存成功，如果上下滑动你会发现，layer变成了绿色。但是如果你滑动幅度较大会发现，新出现的控件会是红色然后变成绿色，因为刚开始这些控件的layer还没有缓存。  
注意点：  
（1）系统给光栅化缓存分配了一个固定的大小，因此不能过度使用，如果超出了缓存也会造成离屏渲染。  
（2）缓存的时间为100ms，因此如果在100ms内没有使用缓存的对象，则会从缓存中清除。



**Color Copied Images（图片颜色格式）**

Shows images that are copied by Core Animation in blue苹果官方注释被拷贝给CPU进行转化的图片显示为绿色。那么这句话怎么理解呢？如果GPU不支持当前图片的颜色格式，那么就会将图片交给CPU预先进行格式转化，并且这张图片标记为蓝色。那么GPU支持什么格式呢？苹果的GPU只解析32bit的颜色格式，如果使用Color Copied Images去调试发现是蓝色，这个时候你也可以去找你们的UI眉眉了～  
知识扩展：32bit指的是图片颜色深度，用“位”来表示，用来表示显示颜色数量，例如一个图片支持256种颜色，那么就需要256个不同的值来表示不同的颜色，也就是从0到255，二进制表示就是从00000000到11111111，一共需要8位二进制数，所以颜色深度是8。通常32bit色彩中使用三个8bit分别表示R红G绿B蓝,还有一个8bit常用来表示透明度（Alpha）。

**Color Non-Standard Surface Formats (不标准的表面颜色格式)**

这个调试选项没有一篇博文讲过，都是直接略过，我也尝试很多途径去找这个选项到底是什么作用，然而苹果开发文档以及stack overflow都没有对这个有所解释。自己真机调试尝试了很多发现有个规律，就是打开这个选项，某些Label和Button的背景颜色都会出现银白色，但是不是必先现的，有些Label和Button依然正常颜色背景。其他ImageView等控件是不会出现银白色的背景颜色，猜想是不是和文本Text的设置有关系。如果您对这个有所了解，欢迎讨论。



Color Non-Standard Surface Formats调试效果

**Color Immediately（颜色刷新频率）**

当执行颜色刷新的时候移除10ms的延迟，因为可能在特定情况下你不需要这些延迟，所以使用此选项加快颜色刷新的频率。不过一般这个调试选项我们是用不到的。

**Color Misaligned Images(图片大小)**

这个选项可以帮助我们查看图片大小是否正确显示。如果image size和imageView size不匹配，image会出现黄色。要尽可能的减少黄色的出现，因为image size与imageView size不匹配，会消耗资源压缩图片。下图中的image实际size（81，110），顶部image正常，底部image出现黄色因为放在了一个size x 2的imageView容器中。



Color Misaligned Images调试效果图

**Color Offscreen-Rendered Yellow（离屏渲染）**

离屏渲染Off-Screen Rendering 指的是GPU在当前屏幕缓冲区以外新开辟一个缓冲区进行渲染操作。还有另外一种屏幕渲染方式-当前屏幕渲染On-Screen Rendering ，指的是GPU的渲染操作是在当前用于显示的屏幕缓冲区中进行。 离屏渲染会先在屏幕外创建新缓冲区，离屏渲染结束后，再从离屏切到当前屏幕， 把离屏的渲染结果显示到当前屏幕上，这个上下文切换的过程是非常消耗性能的，实际开发中尽可能避免离屏渲染。  
触发离屏渲染Offscreen rendering的行为：  
（1）drawRect:方法  
（2）layer.shadow  
（3）layer.allowsGroupOpacity or layer.allowsEdgeAntialiasing  
（4）layer.shouldRasterize  
（5）layer.mask  
（6）layer.masksToBounds && layer.cornerRadius  
这里有需要注意的是第三条layer.shouldRasterize ，其实就是我们本文讲的第三个选项光栅化，光栅化会触发离屏渲染，因此光栅化慎用。  
第六条设置圆角会触发离屏渲染，如果在某个页面大量使用了圆角，会非常消耗性能造成FPS急剧下降，设置圆角触发离屏渲染要同时满足下面两个条件:

|  |  |
| --- | --- |
| 1  2 | layer.masksToBounds = YES;  layer.cornerRadius = 5; |

下图是给一个label设置了圆角，触发离屏渲染：



离屏渲染效果图.jpg

为了尽可能避免触发离屏渲染，我们可以换其他手段来实现必要的功能：  
（1）阴影绘制shadow:使用ShadowPath来替代shadowOffset等属性的设置  
imageViewLayer.shadowPath = CGPathCreateWithRect(imageRect, NULL);  
（2）利用GraphicsContex生成一张带圆角的图片或者view，这里不写具体实现过程，需要的可以度娘Copy，很多现成的代码。

**Color Compositing Fast-Path Blue (快速路径)**

Places a blue overlay over content that is detached from the compositor.标记由硬件绘制的路径为蓝色，蓝色越多越好，可以对直接使用OpenGL绘制的图层进行高亮。没有对OpenGL有过多的研究，所以这里没办法给出demo，大家只需要记住蓝色越多越好就ok。

**Flash Updated Regions (重绘区域)**

Colors regions on your iOS device in yellow when those regions are updated by the graphics processor.这个选项会对重绘的内容高亮成黄色,重绘就是指使用Core Graphics绘制，绘制会损耗一定的性能，因此重绘区域应该越小越好。下图是用真机进入原生地图开启Flash Updated Regions 调试的效果图，很可惜截屏不能截到黄色的区域，因此我用红框圈起来，一共两处，坐上角的是在不停的刷新页面，右下角是在不停的刷新当前位置，因此都是使用Core Graphics重绘刷新的一种场景，并且你可以发现黄色区域很小，区域越小性能越好。



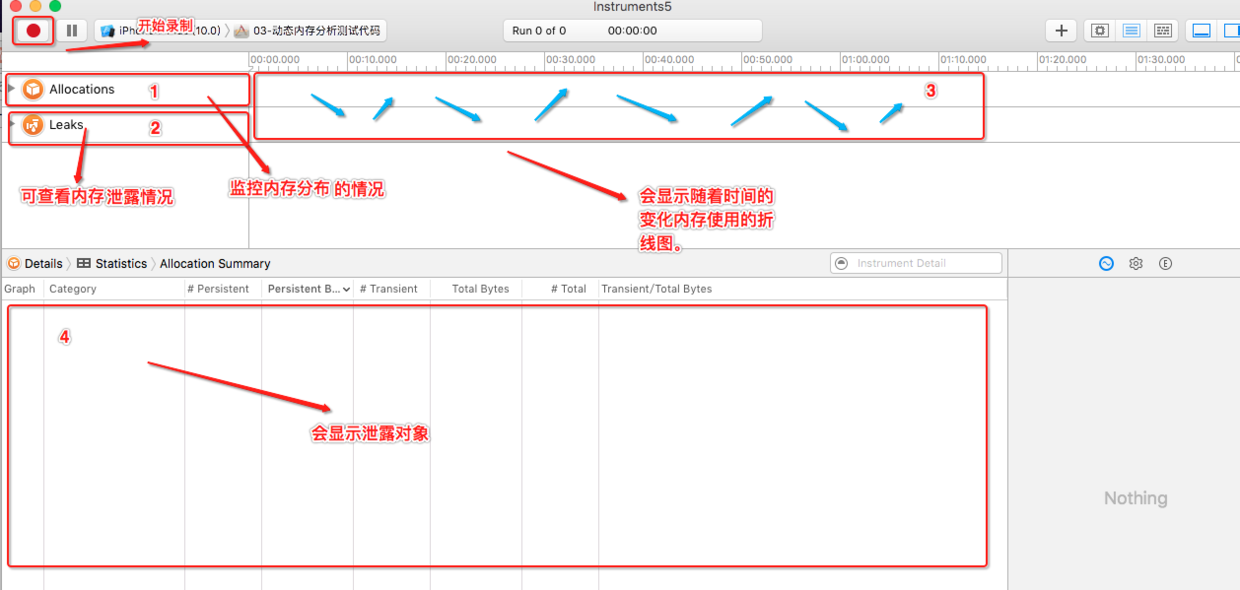
Flash Updated Regions开启地图效果图

好了，花了几天的精力终于写完了～对Core Animation的全部调试选项也都讲解了，如果你能在项目中合理使用，对App的性能提升一定不可小觑。当然这是针对试图的调试方式，功能代码的设计是否合理也是影响性能很重要的一方面。

Leaks

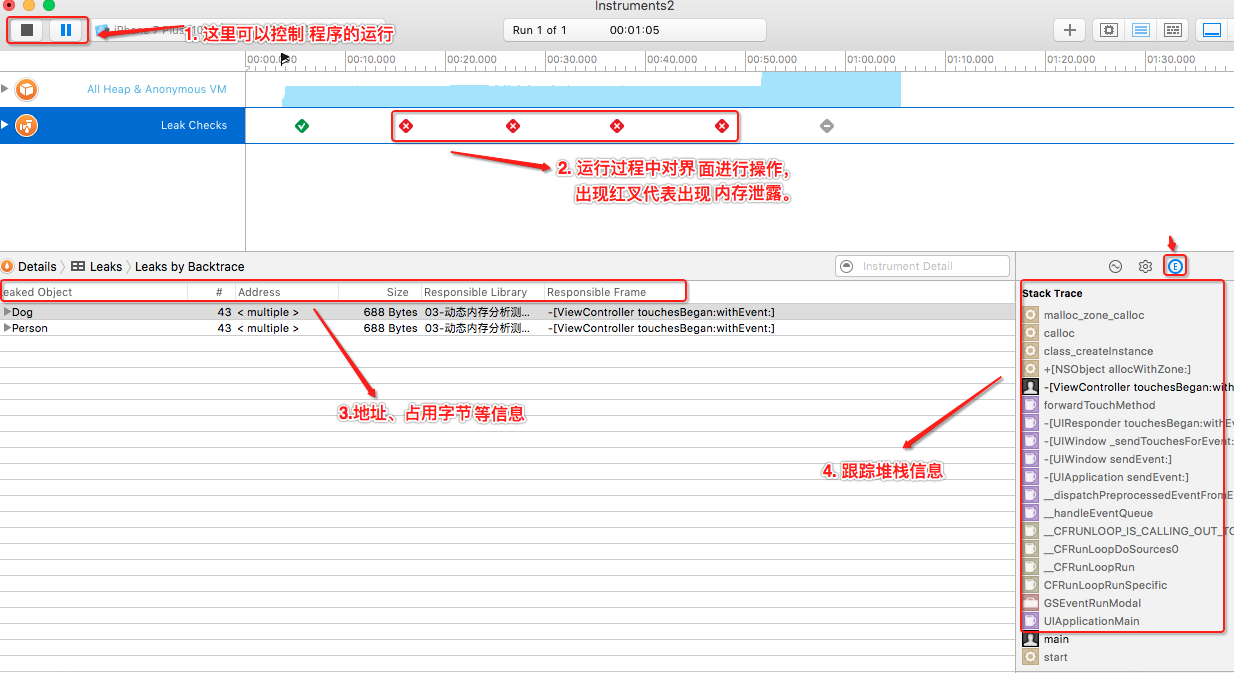
（泄漏）：一般的查看内存使用情况，检查泄漏的内存，并提供了所有活动的分配和泄漏模块的类对象分配统计信息以及内存地址历史记录；

又一个灰常重要的工具，主要检查内存泄漏，在前面Allcations里面我们提到内存泄漏分两种，现在我们研究Leaked Memory， 从用户使用角度来看，内存泄漏本身不会产生什么危害，作为用户，根本感觉不到内存泄漏的存在，真正的危害在于内存泄漏的堆积，最终会耗尽系统所有的内存。我们直接看图：

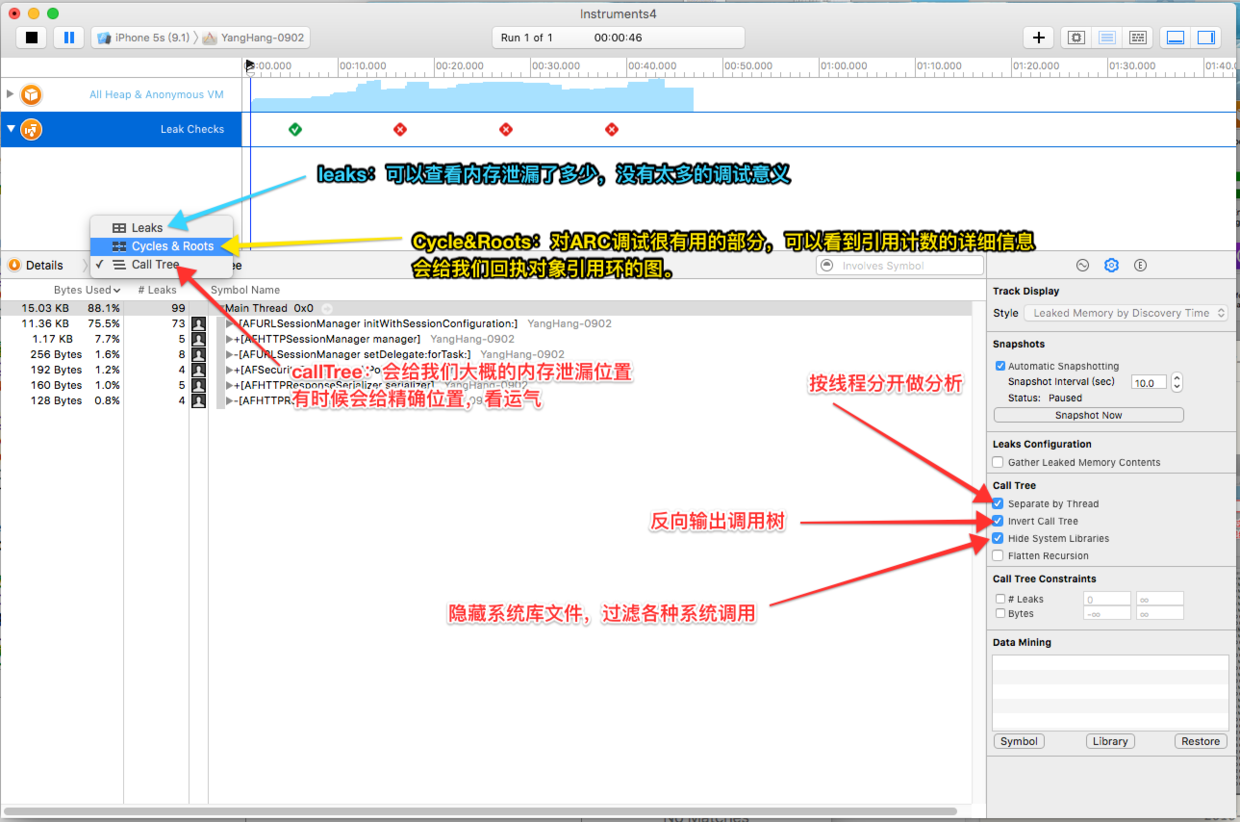


界面的介绍

在 instruments 中,虽然选择了 Leaks 模板,但默认情况下也会添加 Allocations 模板.基本上凡是内存分析都会使用 Allocations 模板, 它可以监控内存分布情况。  
选中 Allocations 模板3区域会显示随着时间的变化内存使用的折线图,同时在4区域会显示内存使用的详细信息,以及对象分配情况.  
点击 Leaks 模板, 可以查看内存泄露情况。如果在3区域有 红X 出现, 则有内存泄露, 4区域则会显示泄露的对象.  
打用leaks进行监测：点击泄露对象可以在(下图)看到它们的内存地址, 占用字节, 所属框架和响应方法等信息.打开扩展视图, 可以看到右边的跟踪堆栈信息，4 黑色代码最有可能出现内存泄漏的方法



监测结果的分析，



这里对右侧Display Settings中的Call tree选项有必要做一下说明[官方user guide翻译]:

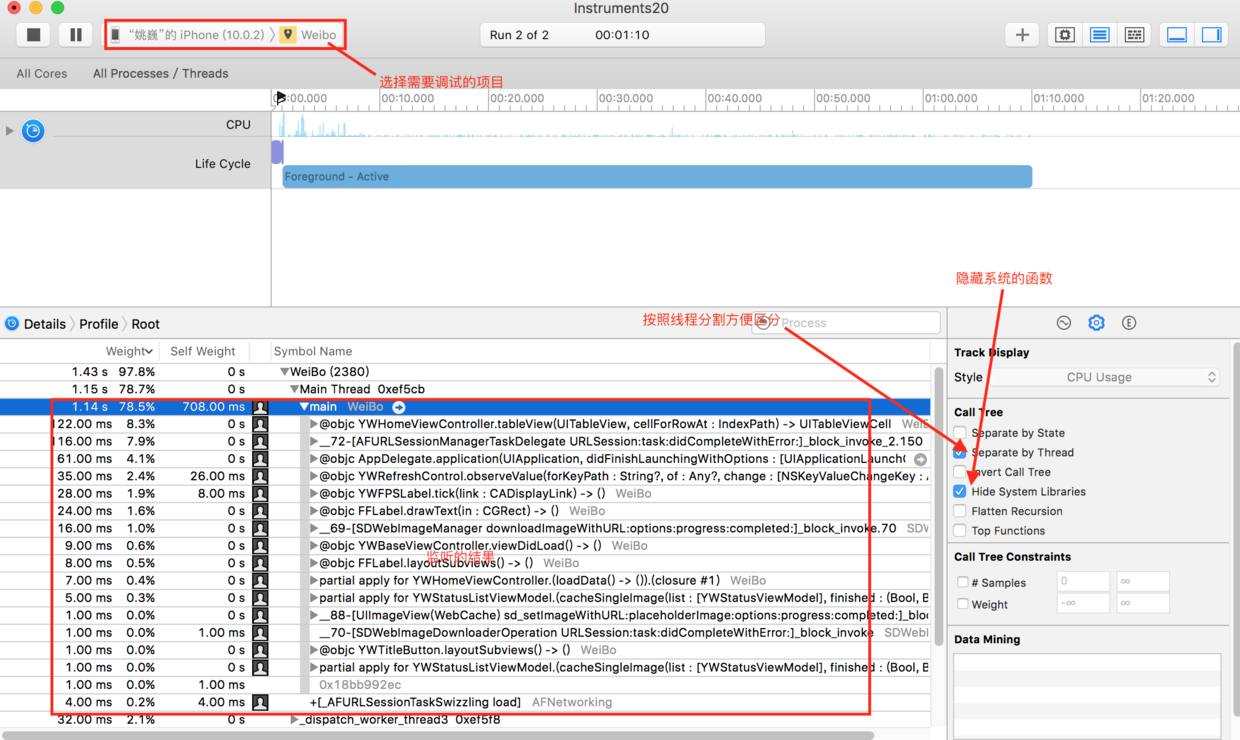
Separate By Thread:线程分离,只有这样才能在调用路径中能够清晰看到占用CPU最大的线程.每个线程应该分开考虑。只有这样你才能揪出那些大量占用CPU的"重"线程，按线程分开做分析，这样更容易揪出那些吃资源的问题线程。特别是对于主线程，它要处理和渲染所有的接口数据，一旦受到阻塞，程序必然卡顿或停止响应。  
Invert Call Tree:从上到下跟踪堆栈信息.这个选项可以快捷的看到方法调用路径最深方法占用CPU耗时（这意味着你看到的表中的方法,将已从第0帧开始取样,这通常你是想要的,只有这样你才能看到CPU中话费时间最深的方法）,比如FuncA{FunB{FunC}},勾选后堆栈以C->B->A把调用层级最深的C显示最外面.反向输出调用树。把调用层级最深的方法显示在最上面，更容易找到最耗时的操作。

Hide Missing Symbols:如果dSYM无法找到你的APP或者调用系统框架的话，那么表中将看到调用方法名只能看到16进制的数值,勾选这个选项则可以隐藏这些符号，便于简化分析数据.  
Hide System Libraries:表示隐藏系统的函数，调用这个就更有用了,勾选后耗时调用路径只会显示app耗时的代码,性能分析普遍我们都比较关系自己代码的耗时而不是系统的.基本是必选项.注意有些代码耗时也会纳入系统层级，可以进行勾选前后前后对执行路径进行比对会非常有用.因为通常你只关心cpu花在自己代码上的时间不是系统上的，隐藏系统库文件。过滤掉各种系统调用，只显示自己的代码调用。隐藏缺失符号。如果 dSYM 文件或其他系统架构缺失，列表中会出现很多奇怪的十六进制的数值，用此选项把这些干扰元素屏蔽掉，让列表回归清爽。  
Show Obj-C Only: 只显示oc代码 ,如果你的程序是像OpenGl这样的程序,不要勾选侧向因为他有可能是C++的  
Flatten Recursion: 递归函数, 每个堆栈跟踪一个条目，拼合递归。将同一递归函数产生的多条堆栈（因为递归函数会调用自己）合并为一条。  
Top Functions:找到最耗时的函数或方法。 一个函数花费的时间直接在该函数中的总和，以及在函数调用该函数所花费的时间的总时间。因此，如果函数A调用B，那么A的时间报告在A花费的时间加上B.花费的时间,这非常真有用，因为它可以让你每次下到调用堆栈时挑最大的时间数字，归零在你最耗时的方法。

Time Profiler

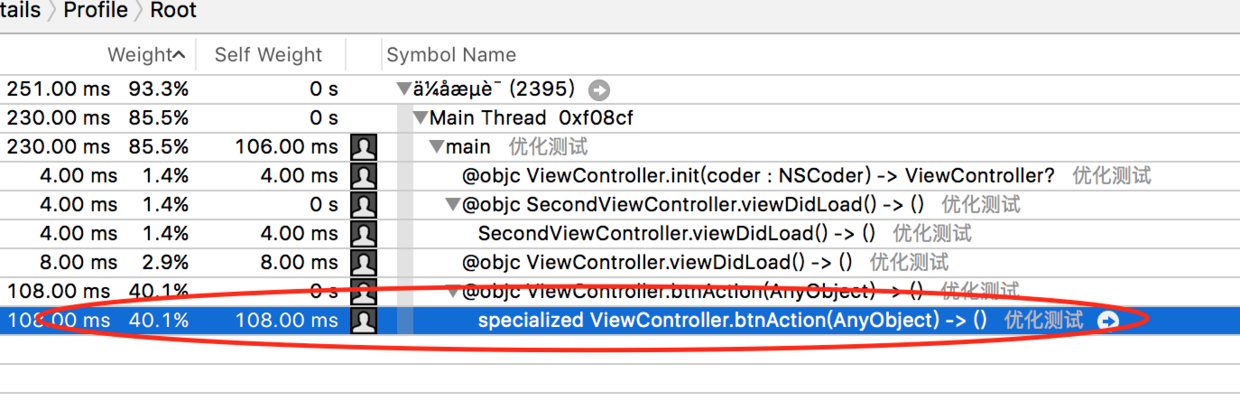
（时间探查）：执行对系统的CPU上运行的进程低负载时间为基础采样。

在开发的过程中，我们经常能感觉到，点击某一按钮，或者做了某一操作，有卡顿，这就是延迟，那使用此工具，就可以揪出耗时的函数



time Profiler.png

根据查看的相关耗时操作，我们就可以右键定位当耗时的方法：  
写一个简单例子看一下：



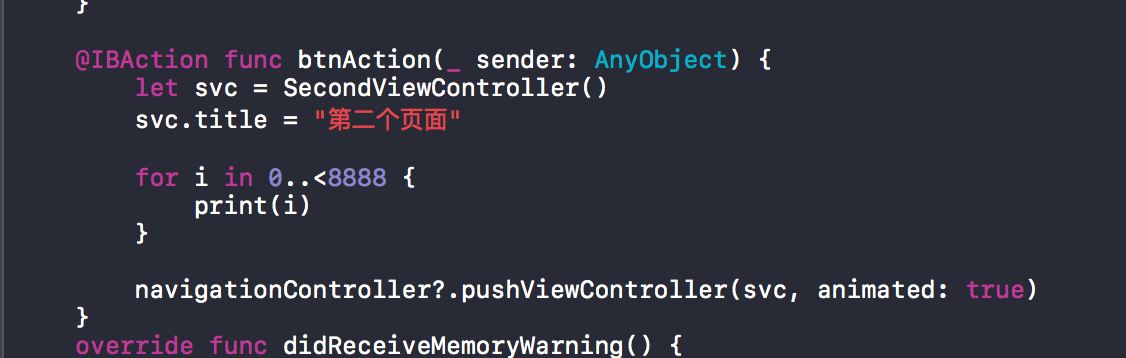
屏幕快照 2016-10-30 下午4.23.02.png

看上图，可以很清楚看到此方法耗时比较严重，右键打开定位到此方法的代码：



屏幕快照 2016-10-30 下午4.24.57.png

代码截图如下：



屏幕快照 2016-10-30 下午4.25.06.png

这时候，我们把循环放到子线程来做

看效果图：



屏幕快照 2016-10-30 下午4.33.43.png

Unit Tests

UI Tests

Unit Tests选项，就是测试业务逻辑等单元测试

UI Tests选项，才是我们要重点讲的UI自动化测试

在创建项目的时候，这两个建议都勾选上。

**UI Tests是什么？**

UI Tests是一个自动测试UI与交互的Testing组件

**UI Tests有什么用？**

它可以通过编写代码、或者是记录开发者的操作过程并代码化，来实现自动点击某个按钮、视图，或者自动输入文字等功能。

**UI Tests的重要性**

在实际的开发过程中，随着项目越做越大，功能越来越多，仅仅靠人工操作的方式来覆盖所有测试用例是非常困难的，尤其是加入新功能以后，旧的功能也要重新测试一遍，这导致了测试需要花非常多的时间来进行回归测试，这里产生了大量重复的工作，而这些重复的工作有些是可以自动完成的，这时候UI Tests就可以帮助解决这个问题了。

单元测试的内容：

单元测试的测试目的

模块接口测试

局部数据结构测试

路径测试

错误处理测试

边界测试

在现有的开发工作中，我们一般都会忽略掉单元测试的重要性，功能开发完成以后开发者拿到现有的测试用例，直接针对每条用例进行手工的测试，测试通过就进行提测，之后测试人员还是重复手工测试的流程、数据的mock、专项测试等，这样以来白盒测试的流程有时间份量会变的很低，开发人员不知道自己模块代码的覆盖路问题，更多的时间可能某些代码一直到到上线都从来没有跑过，以至于到了真实环境下会产生一些意想不到的问题，这样以来风险极高，整体来说单元测试还是至关重要的。

下来我说一些需要注意的地方。

1.点击按钮，有汉字时，需要把大写U该成小写u。

2.模拟器点击textField和textView会出现问题，系统问题，暂未解决，不过在真机上未返现此问题。

3.点击cell时需要先判断cell的个数，因为获取不到指针，所以只能通过数组元素获取。

4.给输入框等输入值，时应先获取焦点，调用tap方法即可。

什么时候用到单元测试：

1、写完代码以后：想要验证一下自己写的代码是否有问题。

2、写代码之前：就是写代码之前所有的功能分模块的设计好，测试通过了再写。（我反正是没用过）。

3、修复某个bug后：一般修复完某个bug，为了确保修复是成功的，会写测试。