

iOS性能优化

浏览：353 发布日期：2016-09-05 分类：ios

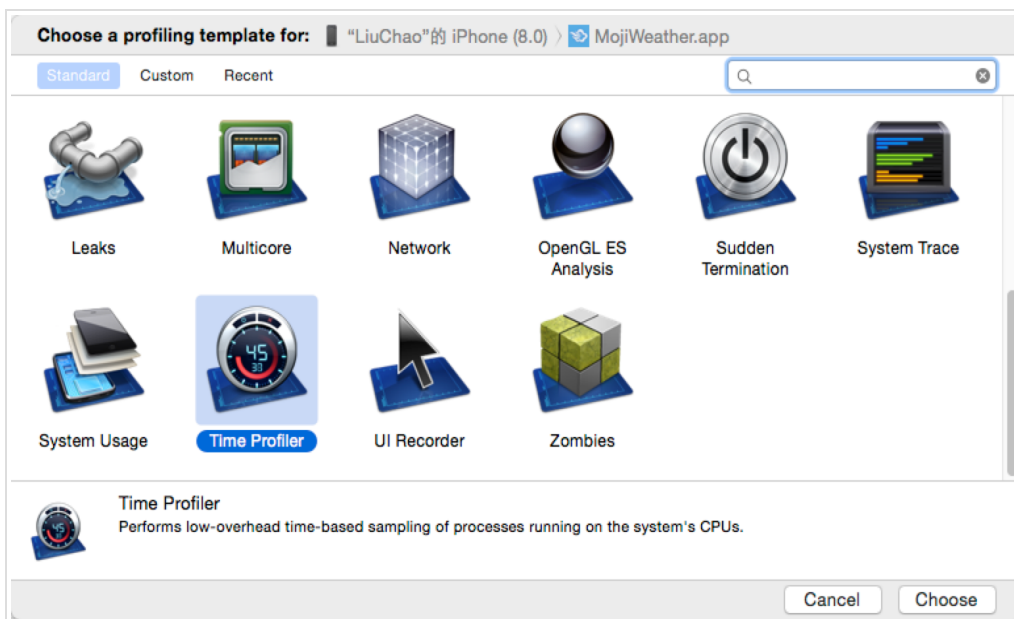
最近采用Instruments 来分析整个应用程序的性能.发现很多有意思的点，以及性能优化和一些分析性能消耗的技巧.小结如下.

Instruments使用技巧

关于Instruments官方有一个很有用的[用户使用Guide](#),当然如果不习惯官方英文可以在[这里](#)找到中文本翻译版本PDF参阅.Instruments 确实是一个很强大的工具，用它来收集关于一个或多个系统进程的性能和行为的数据极为方便，并能及时跟踪随着时间产生的数据.还可以广泛收集不同类型的数 据.关于Instrument工具基本使用不在赘述.如下重点说明一些使用技巧.

1.概览

工具通过Xcode工具栏中Product->Profile可以启动.启动后界面如下:



Instrument概览[via by chen kai]

当点击Time Profiler应用程序开始运行后.就能获取到整个应用程序运行消耗时间分布和百分比.为了保证数据分析在统一使用场景真实行有如下点需要注意:

在 开始进行应用程序性能分析的时候.一定要使用真机.模拟器运行在Mac上，然而Mac上的CPU往往比iOS设备要快。相反，Mac上的GPU和iOS设备的完全不一样，模拟器不得已要在软件层面（CPU）模拟设备的GPU，这意味着GPU相关的操作在模拟器上运行的更慢，尤其是使用 CAEAGLLayer来写一些OpenGL的代码时候.这就导致模拟器性能数据和用户真机使用性能数据相去甚远。

收藏

3

赞

0

浏览

353

0

热门推荐

- 1 Android常用的工具类
- 2 JavaScript-数组去重由慢...
- 3 12个用得着的JQuery代码...
- 4 简单又好用的聊天室技术一...
- 5 让广大开发者相见恨晚的A...

最新更新

- 1 gulp前端构建工具白话讲...
- 2 javascript基础之String
- 3 react-router 按需加载
- 4 「daza.io」这将是独立...
- 5 立足Docker运行MySQL：...

另外在开始性能分析前另外一件重要的事情是，应用程序运行一定要**发布配置** 而不是**调试配置**。

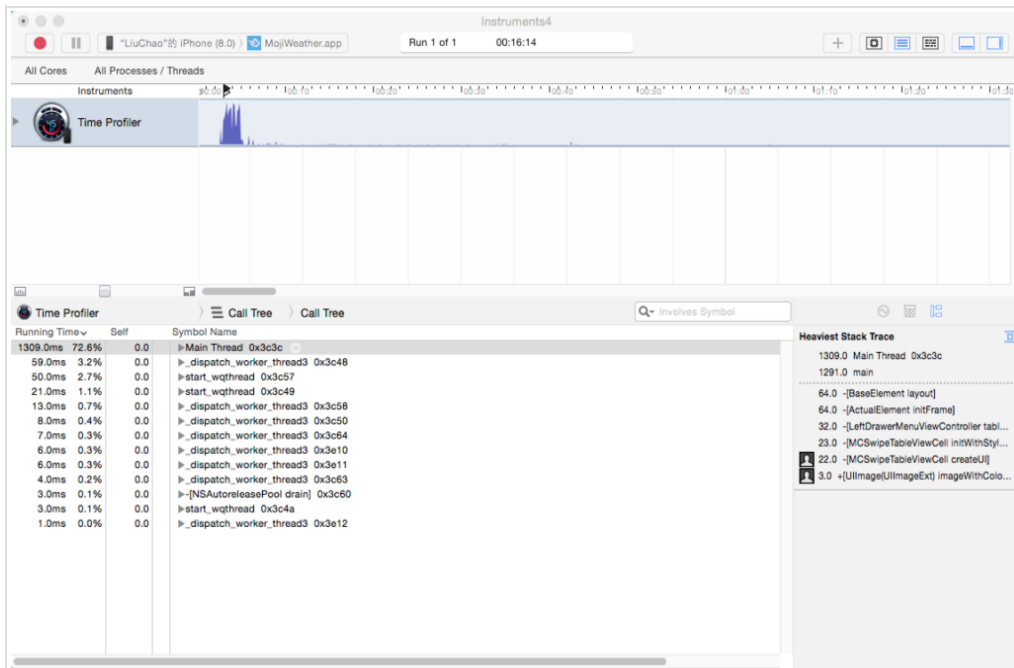
在 发布环境打包的时候，编译器会引入一系列提高性能的优化，例如去掉调试符号或者移除并重新组织代码。另iOS引入一种"Watch Dog"[看门狗]机制。不同的场景下，“看门狗”会监测应用的性能。如果超出了该场景所规定的运行时间，“看门狗”就会强制终结这个应用的进程。开发者 可以crashlog看到对应的日志。但Xcode在调试配置下会禁用"Watch Dog"。

2.Time Profiler

选择Time Profiler启动。

time profile时间分析工具用来检测应用CPU的使用情况。可以看到应用程序中各个方法正在消耗CPU时间。使用大量CPU不一定是个问题。类似我们客户 端中不同场景的天气动画[类似大雨]的路径就对CPU依赖就非常高，动画本身也是非常苛刻且耗费资源较多的任务。

点击Record 开始运行。



Time Profile 分析界面[via by chenka]

刚开始我们拿到分析数据时往往是这样的：



这里显示的是执行代码完整路径，其中系统和应用本身一些调用路径完全揉捏在一起，完全看不到我们关心的应用程序中实际代码执行耗时和代码路径实际所在位置。简单的方式可以快速勾选右边Call Tree中Separate Thread和Hide System Libraries两个选项[后面会解释选项作用]：



可以看到直接能够看到应用程序各个方法调用耗时直接路径,剔除了系统相关方法和反向调用树路径,清爽很多.如果觉得这还不够直观,选择任意一个耗时方法分支[这里选择 WeatherViewController viewDidLoad]双击进入会看到:



可以直接定位到viewDidLoad的代码，也可以直观的看到改方法下调用其他方法耗时的时间.类似[sell loadCityWeatherScroollerView]耗时是121x, x既耗时单位这里为ms毫秒.当然如果直接

在Instrument找到问题 觉得不方便修改,可以直接点击右上角Xcode按钮会直接定位Xcode对应调用方法入口.这样很容易能够快速定位代码占用CPU最多的方法.也可以打开 Xcode快速修改并重新运行Profile来看修改后耗时前后对比.简单便捷.

这里对右侧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.

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,比如FuncA{FuncB{FuncC}},勾选后堆栈以C->B->A把调用层级最深的C显示最外面.

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: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,可以进行勾选前后对执行路径进行比对会非常有用.

关于其他方法不再赘述.

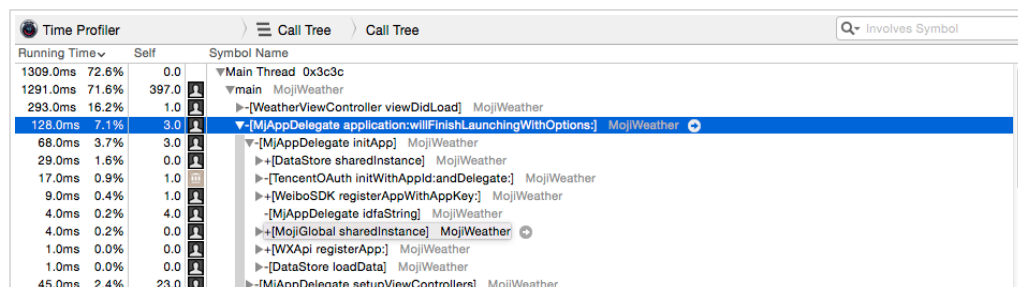
性能分析&代码优化

我们这次性能优化主要针对如下两个使用场景:

A: 应用程序第一次启动到进入天气首页的时间.

B: 从后台切到前台天气首页占用时间.

在还没有拿到性能分析数据之前,一直认为第一次启动耗时主要浪费AppDelegate中第三方框架初始化上[类似WeiBo&WeChat 相关SDK初始化调用].当我们拿到实际性能数据耗时占用比时发现实际情况并非如此:



Running Time	Self	Symbol Name
1309.0ms	72.6%	0.0
1291.0ms	71.6%	397.0
293.0ms	16.2%	1.0
128.0ms	7.1%	3.0
68.0ms	3.7%	3.0
29.0ms	1.6%	0.0
17.0ms	0.9%	1.0
9.0ms	0.4%	1.0
4.0ms	0.2%	4.0
4.0ms	0.2%	0.0
1.0ms	0.0%	0.0
1.0ms	0.0%	0.0
45.0ms	2.4%	23.0

启动耗时

如上可以看到应用程序启动初始化工作主要会在MJAppDelegate如下两个方法展开:**willFinishLaunchingWithOptions**和**didFinishLaunchingWithOptions**.其中第三方框架初始化工作主要是willFinishLaunchingWithOptions中完成的.而实际情况耗时占比非常小.基本可以

忽略不计。

而我们要优化两个启动时间场景,不同在于,第一次进入应用需要经过新手教程、添加城市、请求城市数据、解析数据、初始化天气首页UI元素并加载场景动画,而从后台进入时则从本地存储DT文件中解析天气数据、初始化天气首页UI元素并加载天气动画。

1.NSDateFormatter问题凸显

针对这点重点分析应用启动&天气首页耗时,在AB两个场景均发现加载首页元素发现如下问题:

153	153.0ms	5.0%	0.0	▼start_wqthread 0x51c3
152	152.0ms	4.9%	0.0	▼_pthread_wqthread libsystem_pthread.dylib
151	151.0ms	4.9%	0.0	▼_dispatch_worker_thread3 libdispatch.dylib
151	151.0ms	4.9%	0.0	▼_dispatch_root_queue_drain libdispatch.dylib
115	115.0ms	3.7%	0.0	▼_dispatch_client_callout libdispatch.dylib
114	114.0ms	3.7%	0.0	▼_dispatch_call_block_and_release libdispatch.dylib
114	114.0ms	3.7%	0.0	▼_49-[MJLineChartView drawTimeLab:alpha:index:isNow:]_block_invoke MojiWeather
112	112.0ms	3.6%	0.0	▼+[NSDate(TimeAgo) getDateStrFromDate:timeZone:inputFormat:] MojiWeather
112	112.0ms	3.6%	0.0	▼+[NSDate(TimeAgo) getDateStrByTimeZone:timeZone:inputFormat:containAPM:] MojiWeather
112	112.0ms	3.6%	0.0	▶-[NSDateFormatter stringForObjectValue:] Foundation
2	2.0ms	0.0%	0.0	▶-[NSBundle bundleWithPath:] Foundation
1	1.0ms	0.0%	0.0	▶_dispatch_async_redirect_invoke libdispatch.dylib
36	36.0ms	1.1%	0.0	▶_dispatch_queue_invoke libdispatch.dylib
1	1.0ms	0.0%	0.0	▶_pthread_exit libsystem_pthread.dylib
1	1.0ms	0.0%	1.0	start_wqthread libsystem_pthread.dylib

NSDate(TimeAgo)getDateStrByTimeZone耗时

继续跟踪发现:

88	88.0ms	1.0%	0.0	▼_dispatch_call_block_and_release libdispatch.dylib
61	61.0ms	0.7%	0.0	▼_49-[MJLineChartView drawTimeLab:alpha:index:isNow:]_block_invoke MojiWeather
61	61.0ms	0.7%	0.0	▼+[NSDate(TimeAgo) getDateStrFromDate:timeZone:inputFormat:] MojiWeather
61	61.0ms	0.7%	0.0	▼+[NSDate(TimeAgo) getDateStrByTimeZone:timeZone:inputFormat:containAPM:] MojiWeather
61	61.0ms	0.7%	0.0	▼-[NSDateFormatter stringForObjectValue:] Foundation
61	61.0ms	0.7%	0.0	▼-[NSDateFormatter regenerateFormatter] Foundation
36	36.0ms	0.4%	0.0	▶_CFDateFormatterSetProperty CoreFoundation
25	25.0ms	0.2%	0.0	▶CFDateFormatterCreate CoreFoundation
26	26.0ms	0.3%	0.0	▼_41-[TendencyChartView strokeTendencyChart:]_block_invoke MojiWeather
26	26.0ms	0.3%	0.0	▼-[TendencyChartView updateData] MojiWeather
22	22.0ms	0.2%	0.0	▼-[WeatherDataContainer getForecastWeatherIndexWithDate:] MojiWeather
16	16.0ms	0.1%	0.0	▼+[NSDate(TimeAgo) getDateStrFromDate:timeZone:inputFormat:] MojiWeather
16	16.0ms	0.1%	0.0	▼+[NSDate(TimeAgo) getDateStrByTimeZone:timeZone:inputFormat:containAPM:] MojiWeather
16	16.0ms	0.1%	0.0	▶-[NSDateFormatter stringForObjectValue:] Foundation
6	6.0ms	0.0%	0.0	▶+[NSDate(TimeAgo) getDateFromInputDateStr:timeZone:inputFormat:] MojiWeather
4	4.0ms	0.0%	0.0	▶-[TendencyChartView isNight] MojiWeather
1	1.0ms	0.0%	0.0	▶_69-[CSIRenditionBlockData expandCSIBitmapData:fromSlice:makeReadOnly:]_block_invoke CoreUI

NSDate耗时

在AB两个场景里均出现加载MJLineChartView 和 TendencyChartView 时获取时区对应时间上耗时较大,而耗时主要在getDateStrByTimeZone这个方法调用上。

```
243 + (NSString *)getDateStrByTimeZone:(NSDate *) date timeZone:(NSTimeZone *)timeZone inputFormat:(NSString *) inputFormat containAPM:(BOOL)
244 ) {
245     if (timeZone == nil) {
246         timeZone = [NSTimeZone systemTimeZone];
247     }
248
249     NSDateFormatter *outputFormatter = [[NSDateFormatter alloc] init];
250     [outputFormatter setTimeZone:timeZone];
251     [outputFormatter setLocale:[NSLocale systemLocale]];
252     [outputFormatter setDateFormat:inputFormat];
253
254     if (isContain){
255         [outputFormatter setAMSymbol:@"am"];
256         [outputFormatter setPMSymbol:@"pm"];
257     }
258
259     if (date == nil) {
260         date = [NSDate date];
261     }
262
263     return [outputFormatter stringFromDate: date];
264 }
265
```

getDateStrByTimeZone方法

其中创建一个NSDateFormatter对象平均耗时33ms左右 而设置NSDateFormatter的3个属性平均耗时也在30ms左右,因为首页24小时天气和未来几天预报中,需要for循环中遍历数据,导致这个

方法别重复调用多次，则消耗时间不断叠加。

针对这个问题:

NSDateFormatter对象本身初始化很慢,同样还有NSCalendar也是如此.然而在一些使用场景中不可避免要使用他们,比如Json数据解析中.使用这个对象同时避免其性能开销带来性能开销,一般比较好的方式是通过**添加属性**(推荐)或创建**静态变量**保持该对象只被初始化一次,而被多次复用.不得不值得一提的是设置一个NSDateFormatter属性速度差不多是和创建新的实例对象一样慢!

添加属性方式如下:

```
14
15 //.h [extension]
16 @property (nonatomic, strong) NSDateFormatter *formatter;
17
18 //.m
19 - (NSDateFormatter *)formatter {
20     if (!_formatter) {
21         _formatter = [[NSDateFormatter alloc] init];
22     }
23     return _formatter;
24 }
25
```

属性方式

针对NSDateFormatter时间开销出了重用对象外, 尽量避免采用其处理多个日期格式.当然针对日期格式处理如果需要提高更多速度, 可以直接采用C,可以采用**第三方库**来规避这个问题..

2.UIColor缓存取舍

在项目代码中看到大量使用如下代码:

```
30
31
32 UIImage *img = [UIImage imageNamed:@"myImage"];
33
34
```

UIImage使用

在Main Thread中发现不同动画场景中Image IO 开销和耗时所占比例均不一,在UIImage元素较多总体叠加耗时也会占用一定比例.内存开销也会明显增高.

UIImage加载图片方式一般有两种:

A : imageNamed初始化

B : initWithContentsOfFile初始化

二者不同之处在于,imageNamed默认加载图片成功后会内存中缓存图片.这个方法用一个指定的名字在系统缓存中查找并返回一个图片对象.如果缓存中没有找到相应的图片对象,则从指定地方加载图片然后缓存对象,并返回这个图片对象.

而initWithContentsOfFile则仅只加载图片,不缓存.

大量使用imageName方式会在不需要缓存的地方额外增加开销CPU的时间来做这件事.当应用程序需要加载一张比较大的图片并且使用一次性,那么其实是没有必要去缓存这个图片的,用imageWithContentsOfFile是最为经济的方式.这样不会因为UIImage元素较多情况下,CPU会被逐个分散在不必要缓存上浪费过多时间.

使用场景需要编程时，应该根据实际应用场景加以区分，UImage虽小，但使用元素较多问题会有所凸显.

3.天气首页加载策略

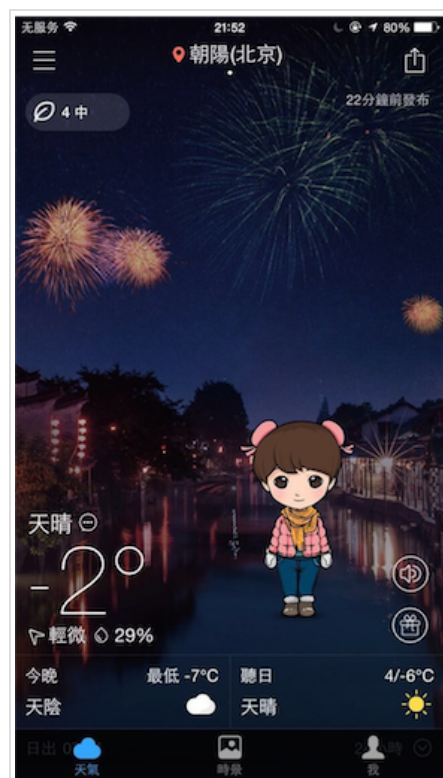
在AB两种场景把性能数据对比分析发现:

Index	Time (ms)	Percentage	Category	Method
305	305.0ms	3.5%	0.0	[WeatherViewController viewDidLoad:] MojWeather
305	305.0ms	3.5%	0.0	[WeatherViewController updateAllWeatherView:] MojWeather
305	305.0ms	3.5%	1.0	[WeatherView updateView:] MojWeather
226	226.0ms	2.6%	0.0	[WeatherInfoView updateAllInfo:] MojWeather
60	60.0ms	0.6%	0.0	[WeatherView avatarStatusChange:] MojWeather
10	10.0ms	0.1%	0.0	[WeatherView updateAirStatus:] MojWeather
5	5.0ms	0.0%	0.0	[GlassScrollView setCustomView:] MojWeather
3	3.0ms	0.0%	0.0	[WeatherTimeLabView updateWeatherTime:] MojWeather
1	1.0ms	0.0%	0.0	[UIView(Hierarchy) _updateConstraintsAsNecessaryAndApplyLayoutFromEngine] UIKit
25	25.0ms	0.2%	0.0	GSEventRunModal GraphicsServices

天气首页WeatherView更新耗时

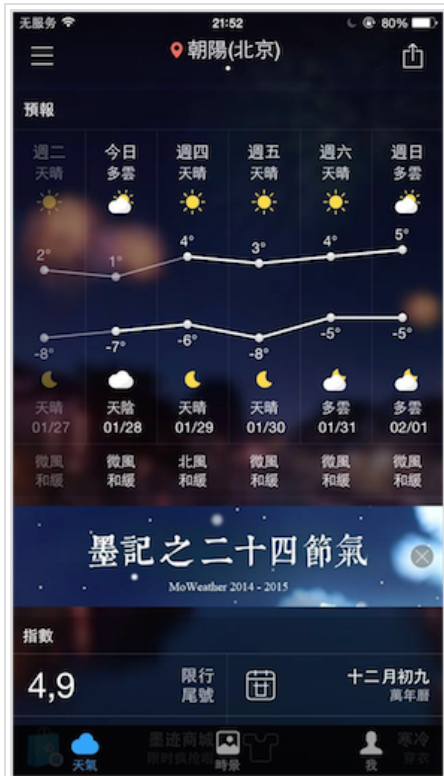
天气首页WeatherView初始化耗时一直300ms-450ms之间,占据首页耗时很大一部分.且一直固定的开销.占据Main Thread3分之一.

而用户进入最先看到是天气首页上半部分:



上半部分

而下半部分需要滚动才能看到下半部分,且不一定触发:



下半部分

而现在整个首页View的初始化和更新全部放到主线程来做.其中WeatherInfoView updateAllInfo方法更新耗时最长.更多的view意味着更多的渲染,也就是意味更多的CPU和内存消耗,对于我们天气首页在 UIScrollView里边嵌套了很多view更是如此

而针对这种情况不要在主线程承载过多的操作.uikit渲染,用户输入回应都需要 主进程上完成.主线程被意外block或者加载响应耗时过多都会影响到用户体验.而针对资源消耗过大操作,处理原则是最小化主线程的CPU占用,将工作“搬离”主线程,不要阻塞主线程.类似本地一些IO完全移到其他线程来做.

调试time profiler过程中发现.即使占用了很少的CPU时间(如果你在Time Profiler中看到这些数据),也可能会阻塞主线程。磁盘、网络、Lock、dispatch_sync以及向其它进程/线程发送消息都会阻塞主线程。Time Profiler只能检测出占用CPU过多的堆栈,但检测不了这些IO的问题.很奇怪.在System Trace里面突然发现了CPU Time很低,但Wait Time很高的调用,说明在主

首页 前端技术 编程语言 移动开发 数据库 服务器 web服务 开发工具

而针对我们应用首页ui中多个view,在加载策略完全可以采用多线程进行同步加载.只把上半部分放在主线程中加载,下班可以同时开一个线程进行同步加载.这样可以大大降低组线程初始化和更新时间,当首页初始化完毕已经呈现是,下半部分其实已经另外一个线程处理完毕.

另 外针对单个view 尽量不要在viewWillAppear费时的操作,viewWillAppear在 view 显示之前被调用,出于效率考虑,在这个方法中不要处理复杂费时的事情;只应该在这个方法设置view 的显示属性之类的简单事情,比如背景色,字体等。不然,用户会明显感觉到 view 显示迟钝.

4 : 应用首次加载时间

应用首次启动加载操作 :

Total Samples	Running Time	Self	Symbol Name
3239	3239.0ms	78.3%	0.0
3086	3086.0ms	74.6%	0.0
3086	3086.0ms	74.6%	0.0
3086	3086.0ms	74.6%	0.0
2788	2788.0ms	67.4%	0.0
285	285.0ms	6.8%	0.0
10	10.0ms	0.2%	0.0
1	1.0ms	0.0%	0.0
1	1.0ms	0.0%	0.0
1	1.0ms	0.0%	0.0
153	153.0ms	3.7%	0.0

首次加载

首次加载坐了如下操作：

A: 链接和载入：可以在Time Profile中显示dyld载入库函数，库会被映射到地址空间，同时完成绑定以及静态初始化.

B: UIKit初始化：如果应用的Root View Controller是由XIB实现的，也会在启动时被初始化.

C: 应用回调：调用UIApplicationDeleagte的回调：
application:didFinishLaunchingWithOptions.

D: 第一次Core Animation调用：在启动后的方法-[UIApplication
_resportAppLaunchFinished]中调用CA::Transaction::commit实现第一帧画面的绘制.

应用程序首次加载中启动方法willFinishLaunchingWithOptions和
didFinishLaunchingWithOptions只做应用程序首次启动必须的要操作,而针对_dyld_start在初
始化库 framework函数的操作.不必要的Framework不要链接，避免首次加载耗时.

小结如上.很多地方代码调用和底层机制看的不是特别明白,整理总结关于优化部分实在有限，如
上仅供各位参考.另外Instruments确实是把分析代码利器.目前没有任何一个第三方工具可以去替
代.推荐各位使用.

来自：<http://www.jianshu.com/p/9e1f0b44935c>

X枫林提供全面的网络编程、脚本编程、网页制作、网页特效，网站建设为站长与网络编程从业者提供学习资料。

天朝-备0101001号-01 本站由菊爆大队支持维护，站内内容全部来源网络，如果侵犯了您的权益请邮件致songshoukui@yeah.net