# Android性能专项测试之TraceView工具(Device Monitor)





> 参考文章: Traceview Walkthrough
> Android 编程下的 TraceView 简介及其案例实战
> 正确使用Android性能分析工具——TraceView
> Android内存使用分析和程序性能分析

## TraceView工具能做什么？

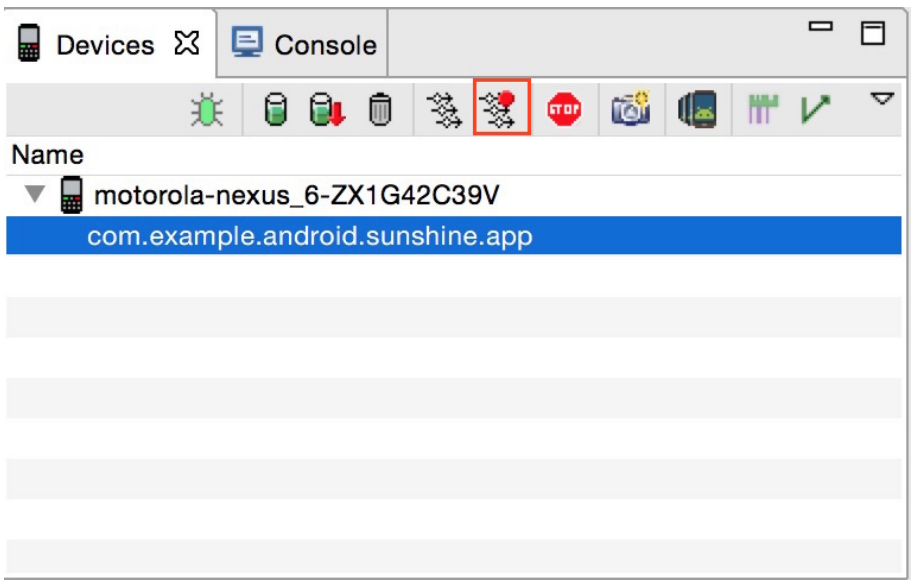
从代码层面分析性能问题，针对每个方法来分析，比如当我们发现我们的应用出现卡顿的时候，我们可以来分析出现卡顿时在方法的调用上有没有很耗时的操作，关注以下两个问题：

- 调用次数不多，但是每一次执行都很耗时
- 方法耗时不大，但是调用次数太多

简单一点来说就是我们能找到频繁被调用的方法，也能找到执行非常耗时的方法，前者可能会造成Cpu频繁调用，手机发烫的问题，后者就是卡顿的问题

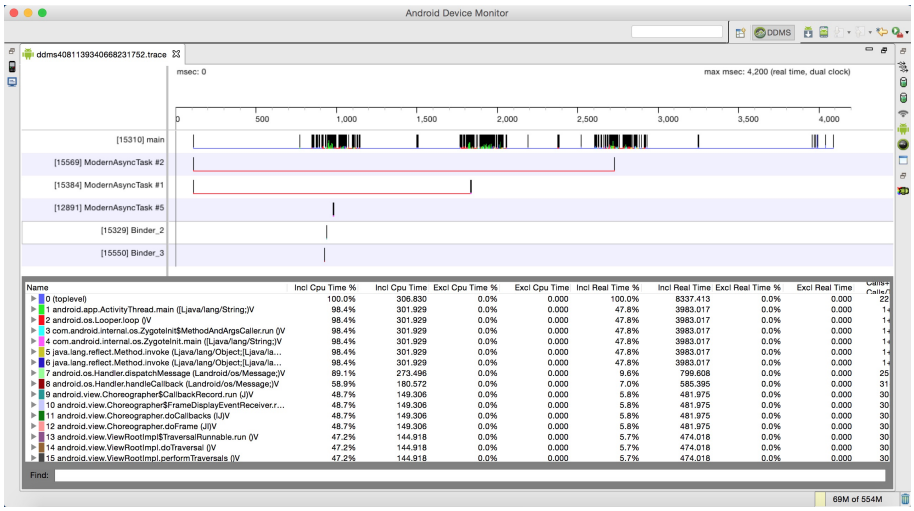
## TraceView工具启动

打开Monitor,点击图中的标注的按钮，启动追踪:




## TraceView工具面板

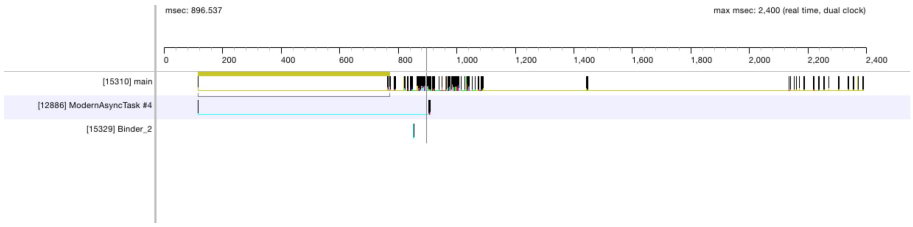
打开App操作你的应用后，再次点击的话就停止追踪并且自动打开traceview分析面板:



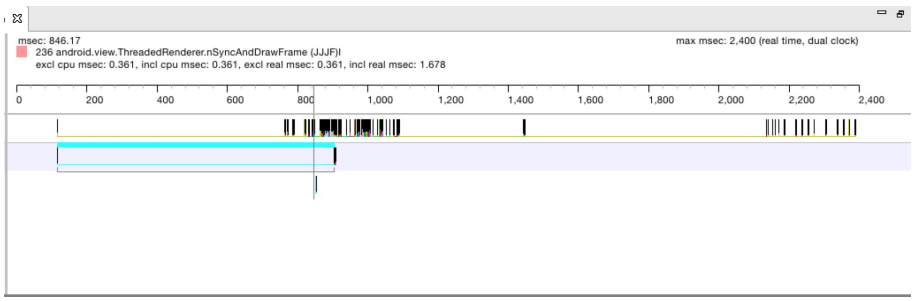

traceview的面板分上下两个部分:

- 时间线面板以每个线程为一行，右边是该线程在整个过程中方法执行的情况
- 分析面板是以表格的形式展示所有线程的方法的各项指标

## 时间线面板

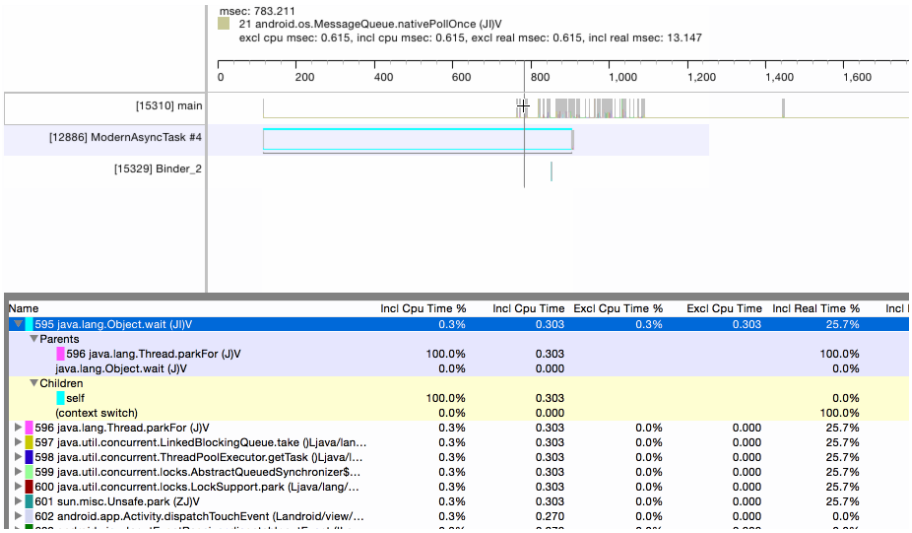



左边是线程信息,main线程就是Android应用的主线程，这个线程是都会有的，其他的线程可能因操作不同而发生改变.每个线程的右边对应的是该线程中每个方法的执行信息，左边为第一个方法执行开始，最右边为最后一个方法执行结束，其中的每一个小立柱就代表一次方法的调用，你可以把鼠标放到立柱上，就会显示该方法调用的详细信息:




你可以随意滑动你的鼠标，滑倒哪里，左上角就会显示该方法调用的信息。

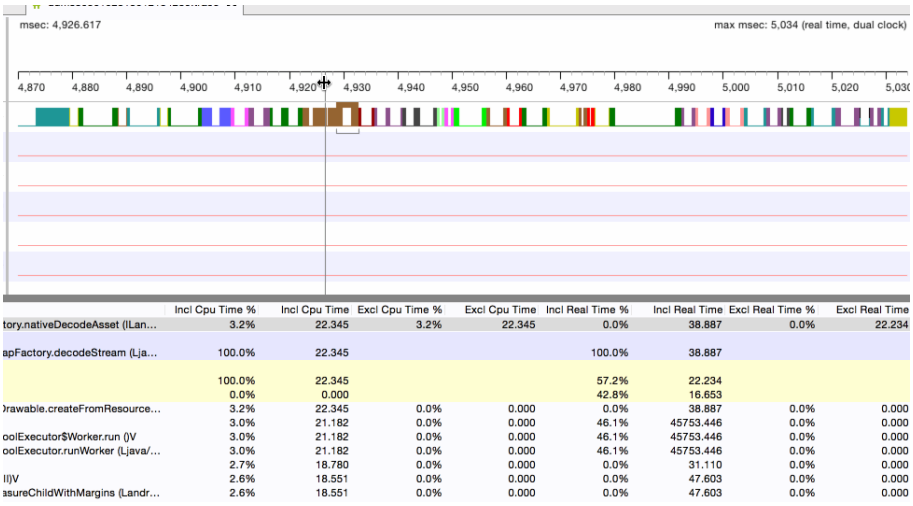
1.如果你想在分析面板中详细查看该方法，可以双击该立柱，分析面板自动跳转到该方法:




| Name | Incl Cpu Time % | Incl Cpu Time | Excl Cpu Time % | Excl Cpu Time | Incl Real Time % | Incl |
|---|---|---|---|---|---|---|
| 595 java.lang.Object.wait (JI)V | 0.3% | 0.303 | 0.3% | 0.303 | 25.7% | |
| Parents | | | | | | |
| 596 java.lang.Thread.parkFor (J)V | 100.0% | 0.303 | | | 100.0% | |
| java.lang.Object.wait (J)V | 0.0% | 0.000 | | | 0.0% | |
| Children | | | | | | |
| self | 100.0% | 0.303 | | | 0.0% | |
| (context switch) | 0.0% | 0.000 | | | 100.0% | |
| 596 java.lang.Thread.parkFor (J)V | 0.3% | 0.303 | 0.0% | 0.000 | 25.7% | |
| 597 java.util.concurrent.LinkedBlockingQueue.take ()Ljava/lan… | 0.3% | 0.303 | 0.0% | 0.000 | 25.7% | |
| 598 java.util.concurrent.ThreadPoolExecutor.getTask ()Ljava/l… | 0.3% | 0.303 | 0.0% | 0.000 | 25.7% | |
| 599 java.util.concurrent.locks.AbstractQueuedSynchronizer$… | 0.3% | 0.303 | 0.0% | 0.000 | 25.7% | |
| 600 java.util.concurrent.locks.LockSupport.park (Ljava/lang/… | 0.3% | 0.303 | 0.0% | 0.000 | 25.7% | |
| 601 sun.misc.Unsafe.park (ZJ)V | 0.3% | 0.303 | 0.0% | 0.000 | 25.7% | |
| 602 android.app.Activity.dispatchTouchEvent (Landroid/view/… | 0.3% | 0.270 | 0.0% | 0.000 | 0.0% | |

2.放大某个区域

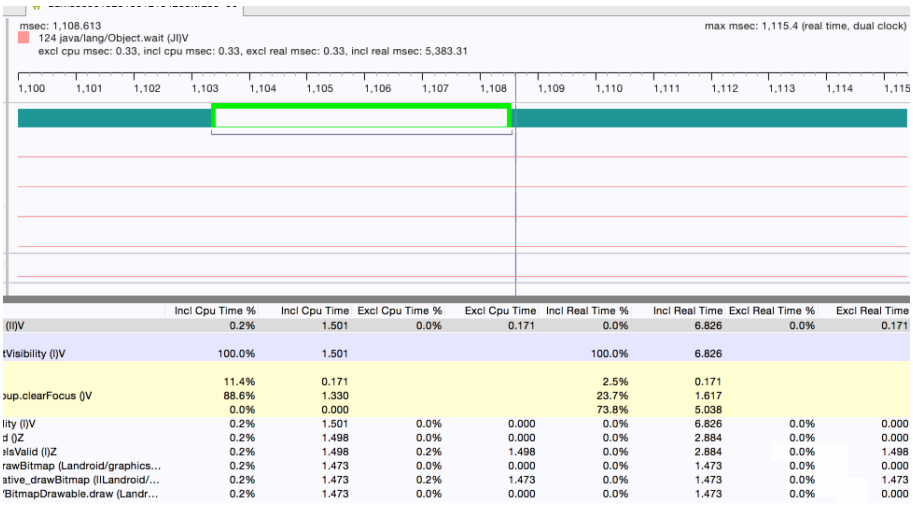
刚打开的面板中，是我们采集信息的总览，但是一些局部的细节我们看不太清，没关系，该工具支持我们放大某个特殊的时间段:




| | Incl Cpu Time % | Incl Cpu Time | Excl Cpu Time % | Excl Cpu Time | Incl Real Time % | Incl Real Time | Excl Real Time % | Excl Real Time |
|---|---|---|---|---|---|---|---|---|
| tory.nativeDecodeAsset (ILan… | 3.2% | 22.345 | 3.2% | 22.345 | 0.0% | 38.887 | 0.0% | 22.234 |
| apFactory.decodeStream (Lja… | 100.0% | 22.345 | | | 100.0% | 38.887 | | |
| | 100.0% | 22.345 | | | 57.2% | 22.234 | | |
| | 0.0% | 0.000 | | | 42.8% | 16.653 | | |
| rawable.createFromResource… | 3.2% | 22.345 | 0.0% | 0.000 | 0.0% | 38.887 | 0.0% | 0.000 |
| | 3.0% | 21.182 | 0.0% | 0.000 | 46.1% | 45753.446 | 0.0% | 0.000 |
| oolExecutor$Worker.run ()V | 3.0% | 21.182 | 0.0% | 0.000 | 46.1% | 45753.446 | 0.0% | 0.000 |
| oolExecutor.runWorker (Ljava/… | 3.0% | 21.182 | 0.0% | 0.000 | 46.1% | 45753.446 | 0.0% | 0.000 |
| | 2.7% | 18.780 | 0.0% | 0.000 | 0.0% | 31.110 | 0.0% | 0.000 |
| I)V | 2.6% | 18.551 | 0.0% | 0.000 | 0.0% | 47.603 | 0.0% | 0.000 |
| asureChildWithMargins (Landr… | 2.6% | 18.551 | 0.0% | 0.000 | 0.0% | 47.603 | 0.0% | 0.000 |

如果想回到最初的状态，双击时间线就可以。

3.每一个方法的表示:




| | Incl Cpu Time % | Incl Cpu Time | Excl Cpu Time % | Excl Cpu Time | Incl Real Time % | Incl Real Time | Excl Real Time % | Excl Real Time |
|---|---|---|---|---|---|---|---|---|
| (II)V | 0.2% | 1.501 | 0.0% | 0.171 | 0.0% | 6.826 | 0.0% | 0.171 |
| tVisibility (I)V | 100.0% | 1.501 | | | 100.0% | 6.826 | | |
| | 11.4% | 0.171 | | | 2.5% | 0.171 | | |
| oup.clearFocus ()V | 88.6% | 1.330 | | | 23.7% | 1.617 | | |
| | 0.0% | 0.000 | | | 73.8% | 5.038 | | |
| lity (I)V | 0.2% | 1.501 | 0.0% | 0.000 | 0.0% | 6.826 | 0.0% | 0.000 |
| d ()Z | 0.2% | 1.498 | 0.0% | 0.000 | 0.0% | 2.884 | 0.0% | 0.000 |
| lsValid (I)Z | 0.2% | 1.498 | 0.2% | 1.498 | 0.0% | 2.884 | 0.0% | 1.498 |
| rawBitmap (Landroid/graphics… | 0.2% | 1.473 | 0.0% | 0.000 | 0.0% | 1.473 | 0.0% | 0.000 |
| ative_drawBitmap (IILandroid/… | 0.2% | 1.473 | 0.2% | 1.473 | 0.0% | 1.473 | 0.0% | 1.473 |
| BitmapDrawable.draw (Landr… | 0.2% | 1.473 | 0.0% | 0.000 | 0.0% | 1.473 | 0.0% | 0.000 |

可以看出来，每一个方法都是用一个 凹 型结构来表示，坐标的凸起部分表示方法的开始，右边的凸起部分表示方法的结束，中间的直线表示方法的持续.

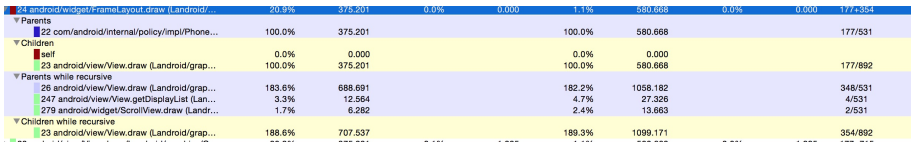
## 分析面板

面板列名含义如下:

| 名称 | 意义 |
|---|---|

| | |
|---|---|
| Name | 方法的详细信息，包括包名和参数信息 |
| Incl Cpu Time | Cpu执行该方法该方法及其子方法所花费的时间 |
| Incl Cpu Time % | Cpu执行该方法该方法及其子方法所花费占Cpu总执行时间的百分比 |
| Excl Cpu Time | Cpu执行该方法所话费的时间 |
| Excl Cpu Time % | Cpu执行该方法所话费的时间占Cpu总时间的百分比 |
| Incl Real Time | 该方法及其子方法执行所话费的实际时间，从执行该方法到结束一共花了多少时间 |
| Incl Real Time % | 上述时间占总的运行时间的百分比 |
| Excl Real Time % | 该方法自身的实际允许时间 |
| Excl Real Time | 上述时间占总的允许时间的百分比 |
| Calls+Recur | 调用次数+递归次数，只在方法中显示，在子展开后的父类和子类方法这一栏被下面的数据代替 |
| Calls/Total | 调用次数和总次数的占比 |
| Cpu Time/Call | Cpu执行时间和调用次数的百分比，代表该函数消耗cpu的平均时间 |
| Real Time/Call | 实际时间于调用次数的百分比，该表该函数平均执行时间 |

你可以点击某个函数展开更详细的信息:




展开后，大多数有以下两个类别:

- Parents:调用该方法的父类方法
- Children:该方法调用的子类方法

如果该方法含有递归调用，可能还会多出两个类别:

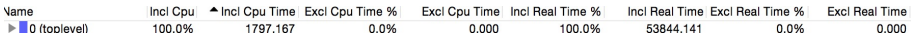
- Parents while recursive:递归调用时所涉及的父类方法
- Children while recursive:递归调用时所涉及的子类方法

首先我们来看当前方法的信息:

| 列 | 值 |
|---|---|
| Name | 24 android/widget/FrameLayout.draw(L android/graphics/Canvas;)V |
| Incl Cpu% | 20.9% |
| Incl Cpu Time | 375.201 |
| Excl Cpu Time % | 0.0% |
| Excl Cpu Time | 0.000 |
| Incl Real Time % | 1.1% |
| Incl Real Time | 580.668 |
| Excl Real Time % | 0.0% |
| Excl Real Time | 0.000 |
| Calls+Recur | 177+354 |
| Cpu Time/Call | 0.707 |

| | |
|---|---|
| Real Time/Call | 1.094 |

根据下图中的toplevel可以看出总的cpu执行时间为1797.167ms，当前方法占用cpu的时间为375.201，375.201/1797.167=0.2087，和我们的Incl Cpu Time%是吻合的。当前方法消耗的时间为580.668，而toplevel的时间为53844.141ms,580.668/53844.141=1.07%,和Incl Real Time %也是吻合的。在来看调用次数为177，递归次数为354，和为177+354=531，375.201/531 = 0.7065和Cpu Time/Call也是吻合的，580.668/531=1.0935,和Real Time/Call一栏也是吻合的。



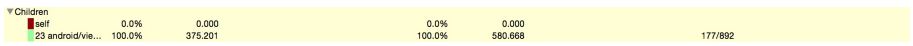

## Parents

现在我们来看该方法的Parents一栏:


| 列 | 值 |
|---|---|
| Name | 22 com/android/internal/policy/impl/PhoneWindow$DecorView.draw(Landroid/graphics/Canvas;)V |
| Incl Cpu% | 100% |
| Incl Cpu Time | 375.201 |
| Excl Cpu Time % | 无 |
| Excl Cpu Time | 无 |
| Incl Real Time % | 100% |
| Incl Real Time | 580.668 |
| Excl Real Time % | 无 |
| Excl Real Time | 无 |
| Call/Total | 177/531 |
| Cpu Time/Call | 无 |
| Real Time/Call | 无 |

其中的Incl Cpu Time%变成了100%，因为在这个地方，总时间为当前方法的执行时间，这个时候的Incl Cpu Time%只是计算该方法调用的总时间中被各父类方法调用的时间占比，比如Parents有2个父类方法，那就能看出每个父类方法调用该方法的时间分布。因为我们父类只有一个，所以肯定是100%，Incl Real Time一栏也是一样的，重点是Call/Total，之前我们看当前方式时，这一栏的列名为Call+Recur，而现在变成了Call/Total,这个里面的数值变成了177/531，因为总次数为531次，父类调用了177次，其他531次是递归调用。这一数据能得到的信息是，当前方法被调用了多少次，其中有多少次是父类方法调用的。

## Children




可以看出来，我们的子类有2个，一个是自身，一个是 `23android/view/View.draw(L android/graphics/Canvas;)V` ，self代表自身方法中的语句执行情况，由上面可以看出来，该方法没有多余语句，直接调用了其子类方法。另外一个子类方法，可以看出被当前方法调用了177次，但是该方法被其他方法调用过，因为他的总调用次数为892次，你可以点击进入子类方法的详细信息中。

## Parents while recursive


列举了递归调用当前方法的父类方法，以及其递归次数的占比，犹豫我们当前的方法递归了354次，以上三个父类方法递归的次数分别为348+4+2 = 354次。

## Children while recursive


列举了当递归调用时调用的子类方法。