**1. Systrace简介**

Systrace是基于Linux ftrace的图形化的分析系统和应用性能的利器，AliOS集成了这个工具，本文会基于AliOS来阐述。

Systrace的功能包括跟踪系统的I/O操作、内核工作队列、CPU负载以及AliOS各个子系统的运行状况等。在AliOS平台中，它主要由3部分组成：

* **内核部分：**Systrace利用了Linux Kernel中的ftrace功能。所以，如果要使用Systrace的话，必须开启kernel中和ftrace相关的模块。
* **数据采集部分**：AliOS在CAF层定义了一个Trace类。应用程序可利用该类把统计信息输出给ftrace。同时，AliOS还有一个atrace程序，它可以从ftrace中读取统计信息然后交给数据分析工具来处理。
* **数据分析工具**：AliOS提供一个Systrace.py（python脚本文件，其内部将调用atrace程序）用来配置数据采集的方式（如采集数据的标签、输出文件名等）和收集ftrace统计数据并生成一个结果网页文件供用户查看。解析ftrace数据实际调用的是Chrome的Catapult，所以atrace抓出来的.atrace文件可以直接通过chrome://tracing来加载。

一句话概括，Systrace 使用 atrace 来启用跟踪，然后读取 ftrace 缓冲区并将其全部封装到一个文件，交给Chrome来解析和查看。

简单说明一下Systrace的原理：它的思想很朴素，在系统的一些关键链路（比如CAF，agil，weston）插入一些信息（我这里称之为Label），通过Label的开始和结束来确定某个核心过程的执行时间，然后把这些Label信息收集起来得到系统关键路径的运行时间信息，进而得到整个系统的运行性能信息。AliOS framework里面一些重要的模块都插入了Label信息，用户App中可以添加自定义的Label，这样就组成了一个完成的性能分析系统。

为什么说Systrace是性能分析的利器呢？因为它抓取的是整个系统的线程运行状态，从logcat或者cpuprofile是看不到这些信息的，比如一个函数执行慢，从logcat或者cpuprofile只能看到结果，但看不到原因。因此Systrace主要在分析性能问题的一开始被用到，可以快速得知是应用本身的问题，还是其它外部因素导致的问题，至少可以定位到线程级别，如果有埋点可以定位到函数级别。

**2.埋点方法**

上文所述的统计信息都来自于trace埋点，AliOS framework已经有足够的埋点，一般需要在自己开发的应用里面进行trace埋点。

**JavaScript埋点样例**：

const Trace = require("yunos/util/Trace");

//beginTrace(tag, methodName)

Trace.beginTrace(Trace.Tag.VIEW, "requestTextLayout-" + this.\_view.\_id);

//some code

//endTrace(tag)

Trace.endTrace(Trace.Tag.VIEW);

其中tag有较多类型（见下文），应用相关的一般都采用VIEW。

**JavaScript异步埋点样例**（最好在执行代码的开始和结束都加上begin和end）：

const Trace = require("yunos/util/Trace");

getCurrentData(args) {

Trace.beginTrace(Trace.Tag.VIEW, "getCurrentData-start");

Trace.endTrace(Trace.Tag.VIEW);

return new Promise((resolve, reject) => {

this.dataManager.getCurrentData(args.platform, (data) => {

Trace.beginTrace(Trace.Tag.VIEW, "getCurrentData-end");

Trace.endTrace(Trace.Tag.VIEW);

if (data) {

resolve("");

} else {

reject("");

}

});

});

}

**Native埋点函数定义**（详见third\_party/atrace/include/trace.h）：

#define TRACE\_BEGIN() trace\_begin(YTRACE\_TAG, \_\_FUNCTION\_\_)

#define TRACE\_BEGIN2(name, ...) trace\_begin(YTRACE\_TAG, name, ##\_\_VA\_ARGS\_\_)

#define TRACE\_END() trace\_end(YTRACE\_TAG)

#define TRACE\_ASYNC\_BEGIN(name, cookie) trace\_async\_begin(YTRACE\_TAG, name, cookie)

#define TRACE\_ASYNC\_END(name, cookie) trace\_async\_end(YTRACE\_TAG, name, cookie)

**Native埋点样例**：

#include <atrace.h>

#undef YTRACE\_TAG

#define YTRACE\_TAG YTRACE\_TAG\_VIEW

TRACE\_BEGIN2(name, ...)

//some code

TRACE\_END()

**Native异步埋点样例**：

#include <atrace.h>

#undef YTRACE\_TAG

#define YTRACE\_TAG YTRACE\_TAG\_VIEW

TRACE\_ASYNC\_BEGIN(name, ...)

//some code

TRACE\_ASYNC\_END(name, ...)

**注意：**

1. begin和end都需要加载一个函数里面，不要分开在两个函数，否则容易匹配不上；

2. begin和end都需要在一个线程里面，否则匹配不上；

3. 埋点要尽量精简，虽然trace的开销较低，但每一个埋点的写入需要消耗约0.1ms，在性能敏感的场景还是有一点影响的；

4. Native需要ldflags里加-latrace。

**3.抓取方法**

atrace命令：

# atrace --help

atrace --help

usage: atrace [item] [group...]

item is:

-A name the comma separate the cmdlines

-B M the buffer size of the trace

-C use a cyclic queuetrace for trace

-K fname,... get trace of the kernel functions listed

-S M trace after sleeping M seconds

-T M trace duration M seconds

-Z reduce the trace

--BEGIN\_ASYNC begin trace and rapidly return

--DUMP\_ASYNC dump the trace buffer

--STOP\_ASYNC stop tracing and rapidly dump buffer

--SHOW\_CATEGORY show all the categories

--STREAM stream trace to stdout like logctl

常用的几个item：

|  |  |
| --- | --- |
| items | description |
| -b M | buffer大小（单位kB),用于限制trace总大小 |
| -C | 循环抓取 |
| -K fname,... | 追踪kernel函数，用逗号分隔 |
| -S M | 延时启动trace |
| -T M | 设置抓取时间长度 |
| -Z | 对trace数据进行压缩 |
| --BEGIN\_ASYNC | 启动trace，立即返回 |
| --DUMP\_ASYNC | dump循环buffer的trace数据 |
| --STOP\_ASYNC | 停止trace并dump循环buffer的trace数据 |
| --SHOW\_CATEGORY | 列出当前支持的所有trace tag |

AliOS当前支持的trace tag如下：

|  |  |
| --- | --- |
| category | description |
| gfx | Graphics |
| input | Input |
| view | View System |
| webview | WebView |
| wm | Window Manager |
| pm | Page Manager |
| audio | Audio |
| video | Video |
| camera | Camera |
| hal | Hardare Modules |
| res | Resource Loading |
| vm | Virtual Machine |
| seed | Seed |
| agilng | Agil NG |
| perf | Performance |
| startup | Page Start up |
| db | Database |
| net | Network |
| ubus | Ubus |
| page | Page Lifecycle |
| sched | CPU Scheduling |
| freq | CPU Frequency |
| idle | CPU Idle |

抓取命令：

atrace gfx input page wm pm vm perf view agilng sched freq idle startup ubus -T 10 -B 10240 -Z > /tmp/atrace.log.z

我写了个atrace的抓取脚本已经打包在 [atrace\_for\_RD.zip](http://30.16.70.3/download/attachments/112296181/atrace_for_RD.zip?version=1&modificationDate=1573635523000&api=v2)。

**4.分析界面**

横坐标是以时间为单位，纵坐标是以进程-线程的方式来划分，同一进程的线程为一组放在一起，可收缩/展开。

每块颜色占据的长度即为系统或者应用定义的trace执行所占据的时间长度，也就是每一个begin和end之间的长度，颜色没有特别的意义，不同的颜色只是用来区分。

下图中红色圈起来的，都是可以点击操作的地方，上方是搜索栏，下方是鼠标操作模式。



### 4.1 快捷键

点击浏览器界面上右上角“？”，可以查看到各个快捷键提示

|  |  |
| --- | --- |
| 快捷键 | 作用 |
| w | 放大，[+shift]速度更快 |
| s | 缩小，[+shift]速度更快 |
| a | 放大，[+shift]速度更快 |
| d | 放大，[+shift]速度更快 |
| f | 放大当前选定区域 |
| m | 标记当前选定区域 |
| g | 切换是否显示60hz的网格线 |
| 0 | 恢复trace到初始态，这里是数字0而非字母o |
| / | 搜索关键字 |
| enter | 显示搜索结果，可通过← →定位搜索结果 |
| 右箭头 | 在当前选中的时间线，选择下一个事件，鼠标点不上的时候很好用 |
| 左箭头 | 在当前选中的时间线，选择上一个事件，鼠标点不上的时候很好用 |
| ? | 显示帮助功能 |

### 4.2 鼠标模式切换

1. Select mode: 单击选中，双击已选定区能将所有相同的块高亮选中；（对应数字1）
2. Pan mode: 拖动平移视图（对应数字2）
3. Zoom mode:通过上/下拖动鼠标来实现放大/缩小功能；（对应数字3）
4. Timing mode:拖动来创建或移除时间窗口线。（对应数字4）

可通过按数字1~4，用于切换鼠标模式； 另外，按住alt键，再滚动鼠标滚轮能实现放大/缩小功能。

## 5.线程状态查看

### 5.1 术语解释

|  |  |
| --- | --- |
| terminology | description |
| Wall Duration | 墙上时间，函数（包括子函数）执行实际花费的时间 |
| CPU Duration | CPU时间，函数（包括子函数）执行在CPU上花费的时间 |
| Self Time | 函数自身执行实际花费的时间 |
| CPU Self Time | 函数自身执行在CPU上花费的时间 |

Systrace 会用不同的颜色来标识不同的线程状态, 在每个方法上面都会有对应的线程状态来标识目前线程所处的状态。

通过查看线程状态我们可以知道目前的瓶颈是什么, 是 CPU 执行慢还是因为dbus 调用, 又或是进行 IO 操作, 又或是拿不到 CPU 时间片。

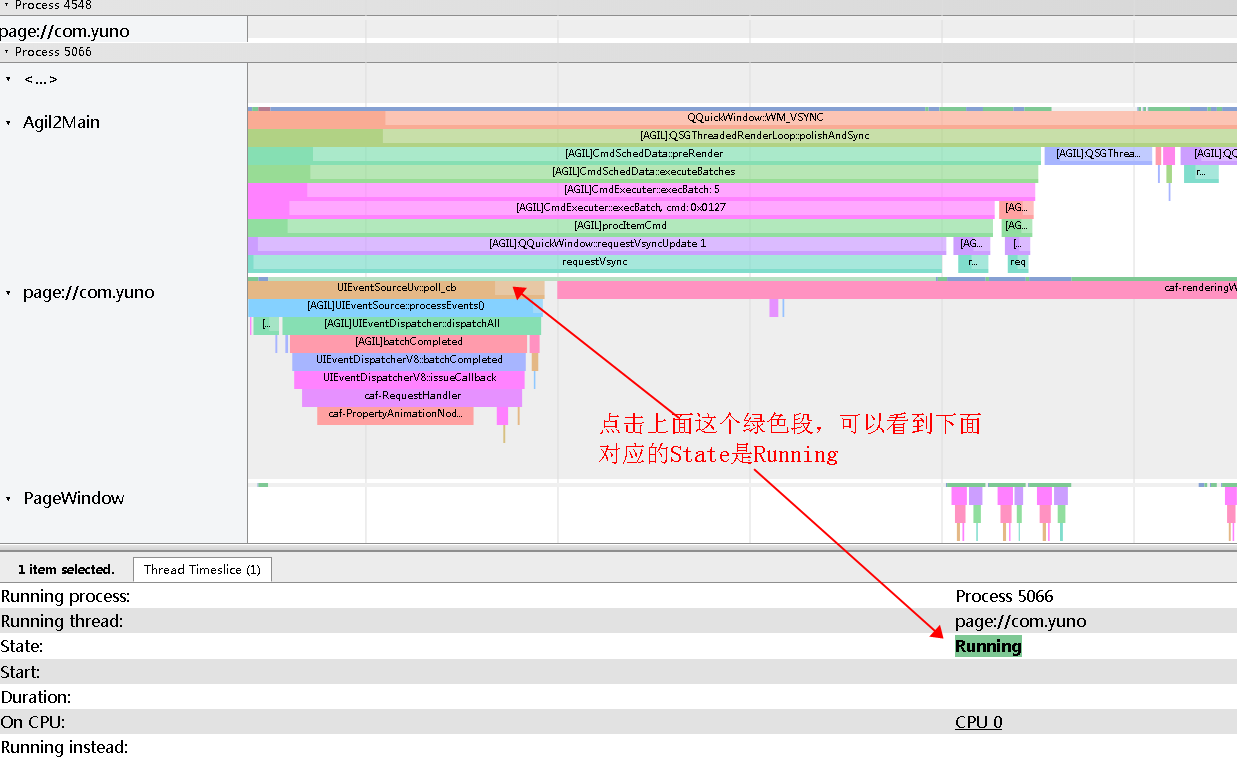
线程状态主要有下面几个：

### 5.2 绿色 : 运行中

只有在该状态的进程才可能在CPU上运行。而同一时刻可能有多个进程处于可执行状态，这些进程的task\_struct结构（进程控制块）被放入对应CPU的可执行队列中（一个进程最多只能出现在一个CPU的可执行队列中）。进程调度器的任务就是从各个CPU的可执行队列中分别选择一个进程在该CPU上运行。

作用：我们经常会查看 Running 状态的线程，查看其运行的时间，与竞品做对比，分析快或者慢的原因：

* 是否频率不够？
* 是否跑在了小核上？
* 是否频繁在 Running 和 Runnable 之间切换？为什么？
* 是否频繁在 Running 和 Sleep 之间切换？为什么？
* 是否跑在了不该跑的核上面？比如不重要的线程占用了超大核。

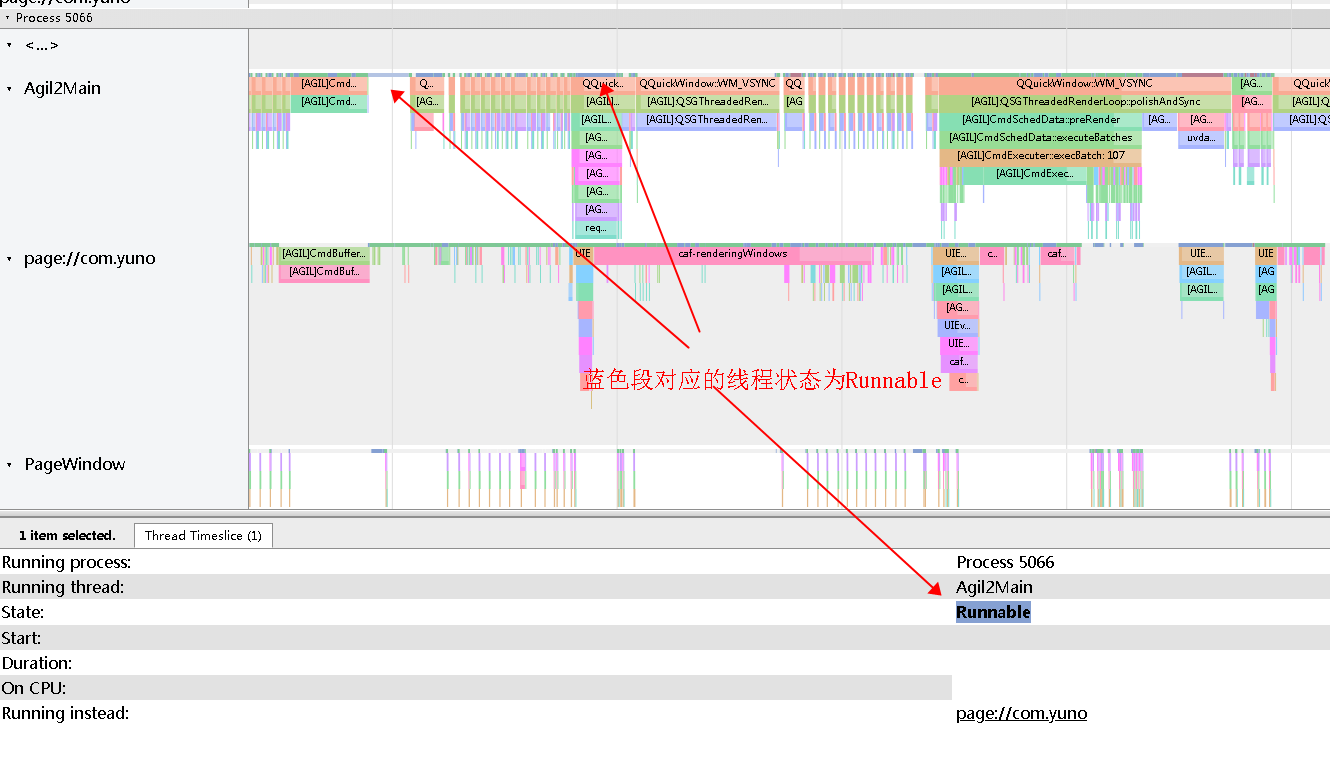


### 5.3 蓝色 : 可运行

线程可以运行但当前没有安排，在等待 cpu 调度

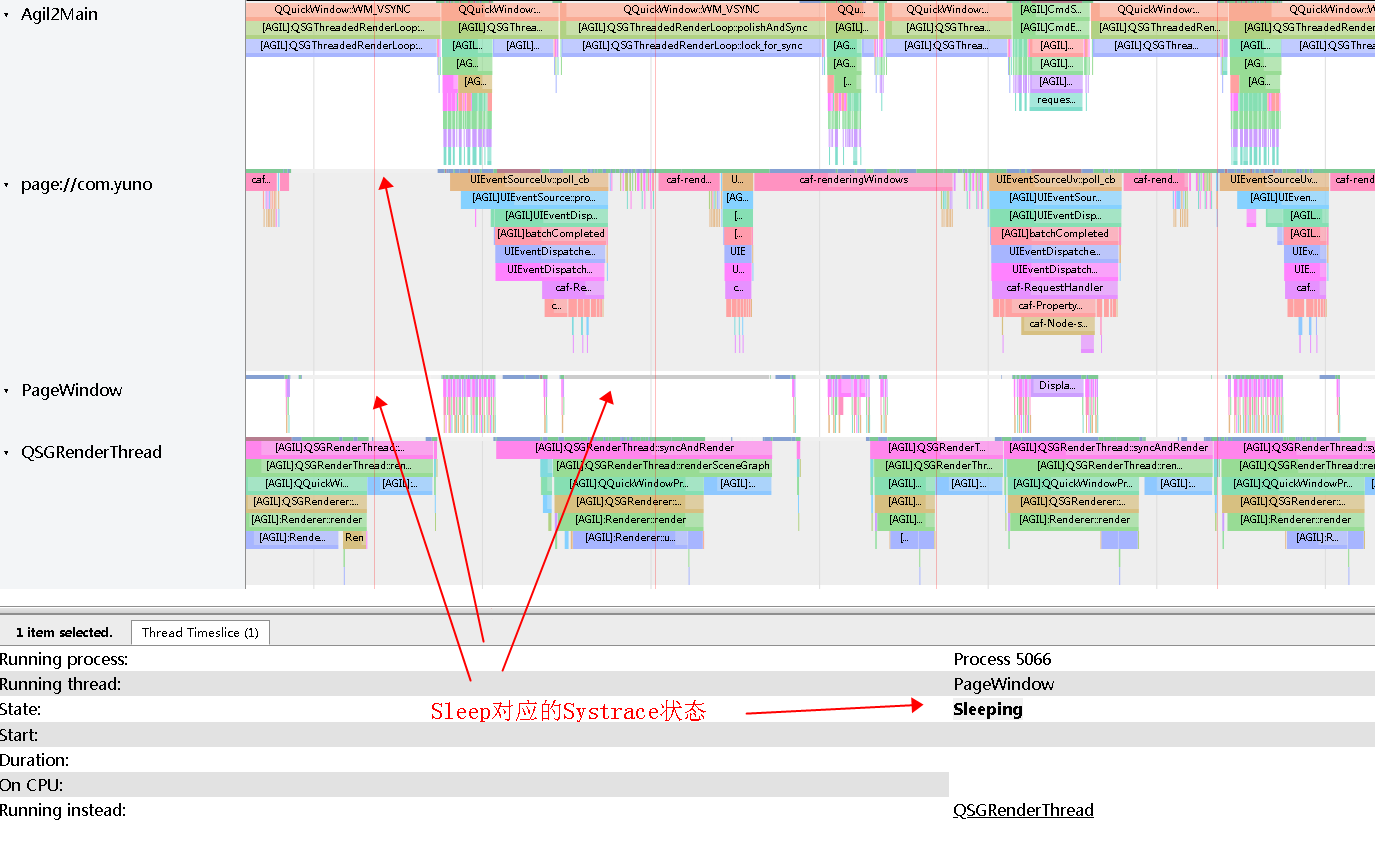
作用：Runnable 状态的线程状态持续时间越长，则表示 cpu 的调度越忙，没有及时处理到这个任务：

1. 是否后台有太多的任务在跑？
2. 没有及时处理是因为频率太低？
3. 没有及时处理是因为被限制到某个 cpuset 里面，但是 cpu 很满？
4. 此时 Running 的任务是什么？为什么？



### 5.4 白色 : 休眠中

线程没有工作要做，可能是因为线程发起了请求在等待返回，或者在等锁。



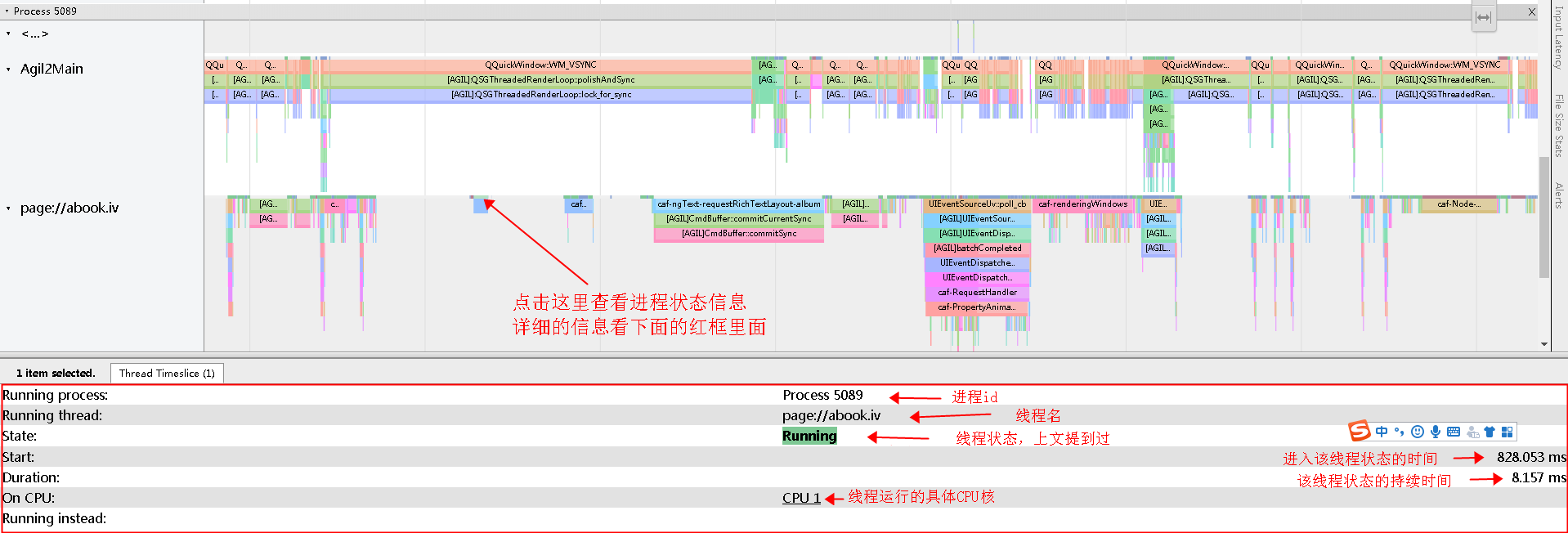
### 5.4 暗红色：不可中断的睡眠态

线程在I / O上被阻塞或等待磁盘操作完成

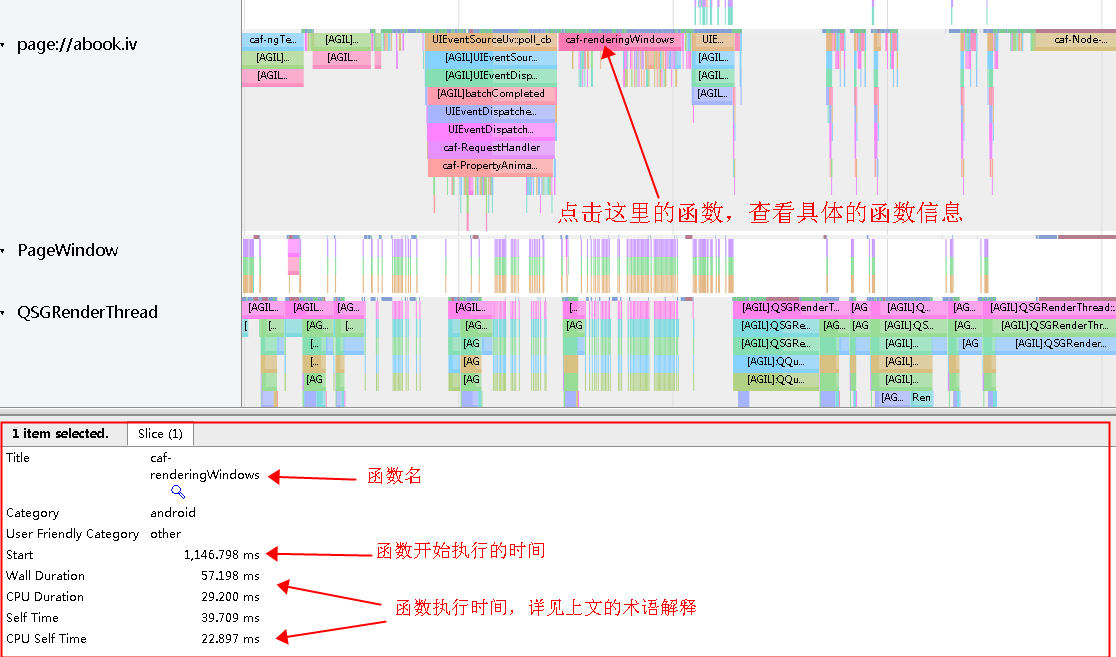


## 6.信息解读

### 6.1 进程状态信息解析



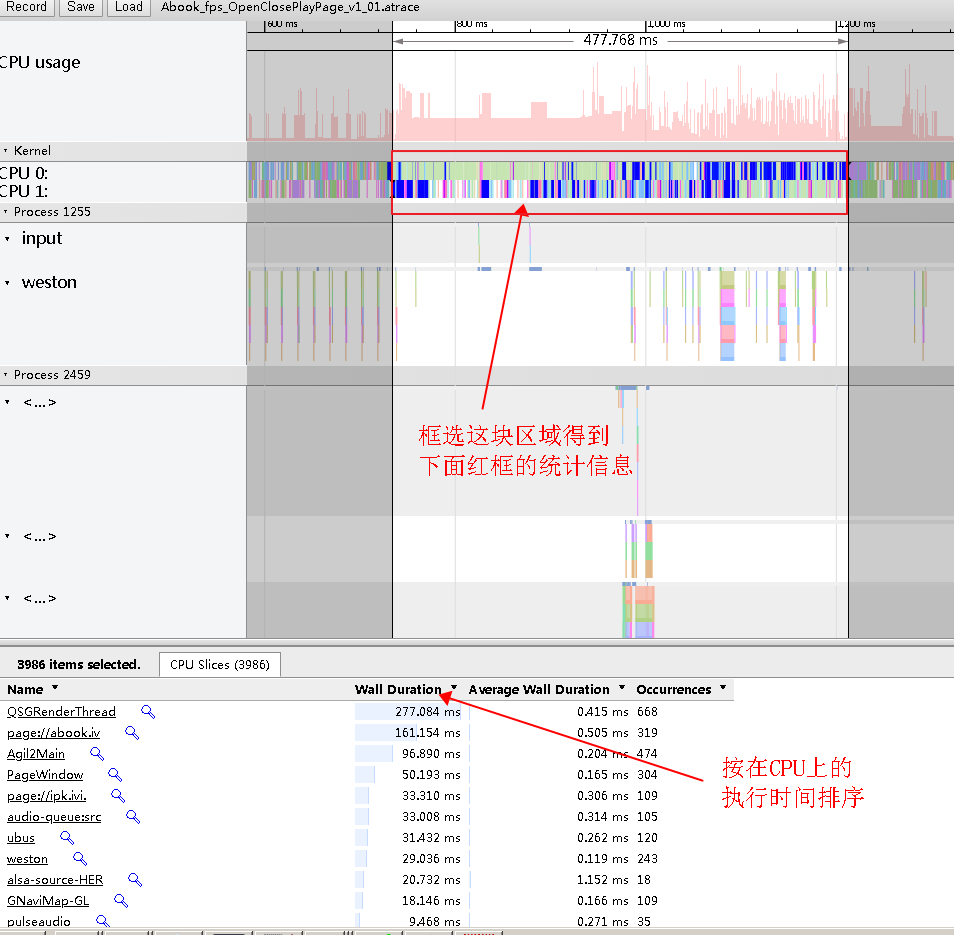
### 6.2 函数 Slice 信息解析



## 7. 框选统计

### 7.1 统计CPU占用

通过mark或者Timing mode选择一段时间后，框选CPU区域，可以得到CPU执行时间统计：



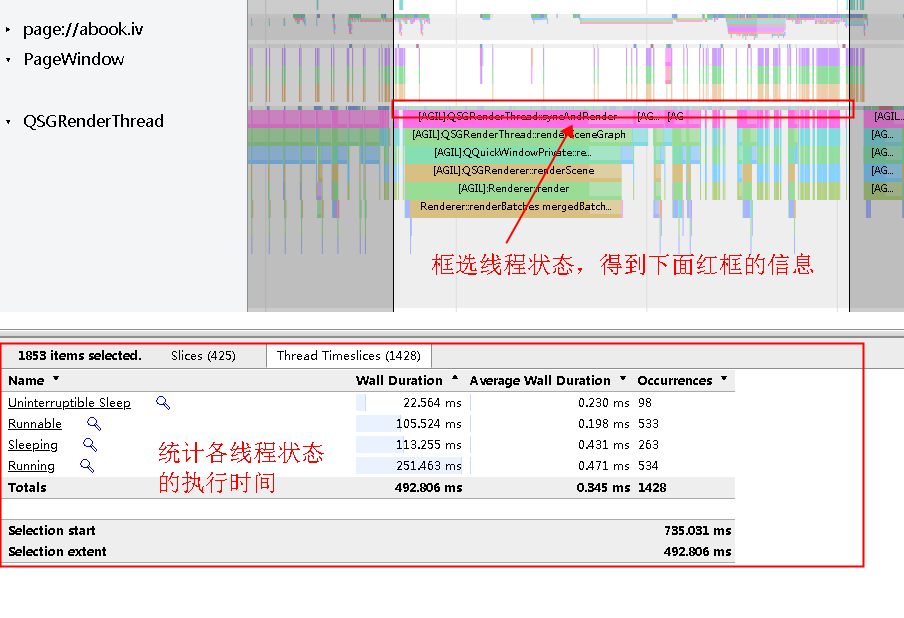
### 7.2 统计函数执行时间

通过mark或者Timing mode选择一段时间后，框选整个进程的所有线程或者几个线程，可以得到所选线程的函数执行时间统计信息：



### 7.3 统计线程状态持续时间

通过mark或者Timing mode选择一段时间后，框选某个线程的线程状态，可以得到所选线程的线程状态持续时间信息：



参考资料：<https://source.android.com/devices/tech/debug/systrace>