微信读书 iOS 性能优化总结

2016-05-03 | hypo

微信读书作为一款阅读类的新产品，目前还处于快速迭代，不断尝试的过程中，性能问题也在业务的不断累积中逐渐体现出来。最近的 1.3.0 版本发布后，关于性能问题的用户反馈逐渐增多，为此，团队开始做一些针对性的性能问题优化。本文将从发现问题、解决问题和预防问题三个方面进行总结。

如何发现性能问题

不同于一般的 bug，性能问题因为并没有统一的标准，而且与用户的机器环境相关性较大，所以往往是在产品上线后才被发现，也导致解决问题的周期很长。微信读书 1.3.0 版本之前，性能问题基本都来自于用户反馈（包括测试人员），受限于测试时间和用户反馈的积极性，性能问题往往到了比较严重的程度，开发人员才真正发现问题。

但是，移动应用要保证良好的用户体验，产品在性能方面的表现极其重要。为了尽可能早、尽可能全面地收集产品的性能问题，就避免不了对产品做性能监控。我们主要从两个维度进行了监控：

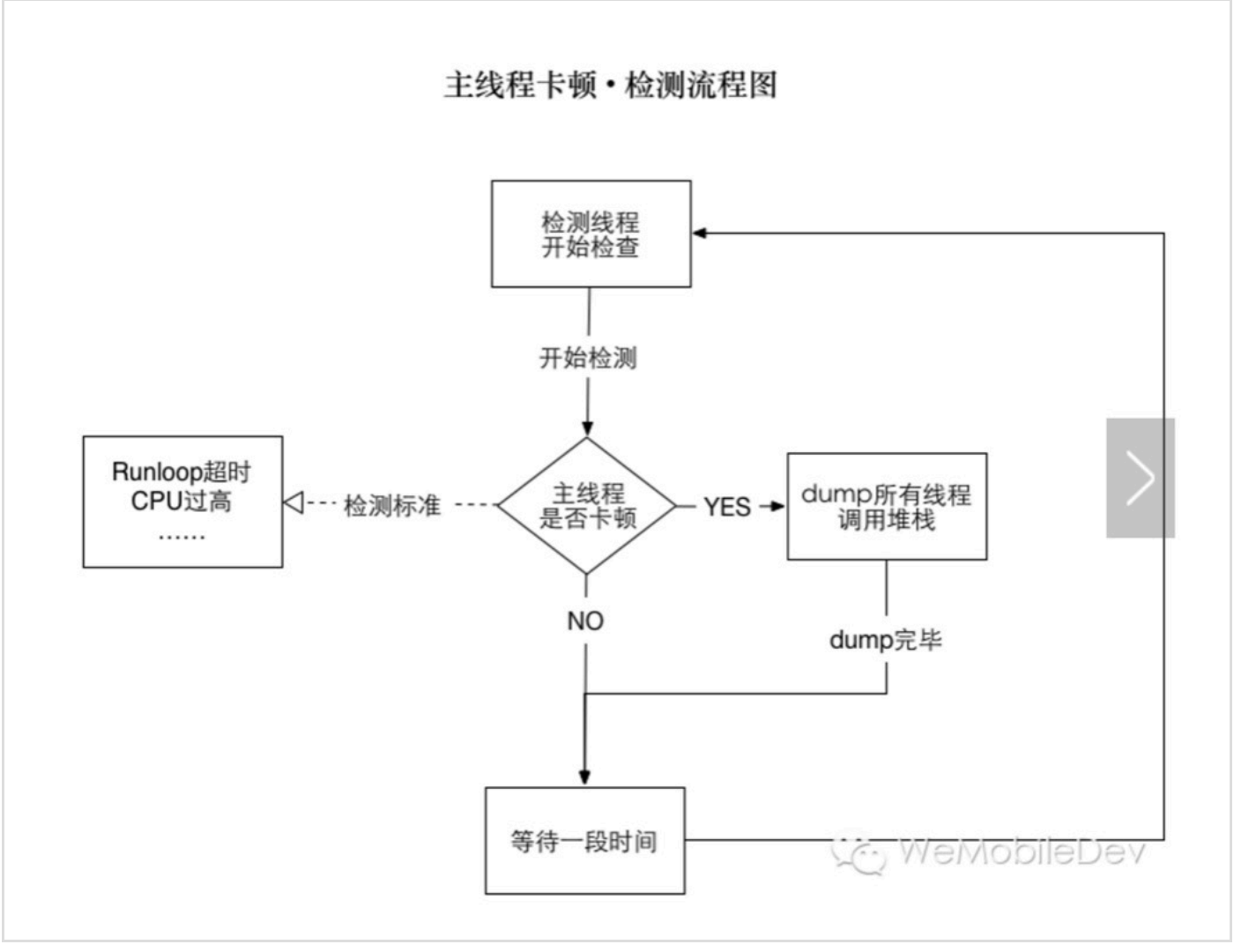
1. 业务性能监控，是指在App本地，业务的开始和结束处打点上报，然后后台统计达到监控目的；

2. 卡顿监控。卡顿监控的实现一般有两种方案：

（1）主线程卡顿监控。通过子线程监测主线程的 runLoop，判断两个状态区域之间的耗时是否达到一定阈值。具体原理和实现，这篇文章介绍得比较详细。

（2）FPS监控。要保持流畅的UI交互，App 刷新率应该当努力保持在 60fps。监控实现原理比较简单，通过记录两次刷新时间间隔，就可以计算出当前的 FPS。

但是，在实际应用过程我们发现，无论是主线程监控，还是 FPS 监控，抖动都比较大。因此，微信团队提出了一套综合的判断方法，结合了主线程监控、FPS监控，以及CPU使用率等指标，作为判断卡顿的标准。



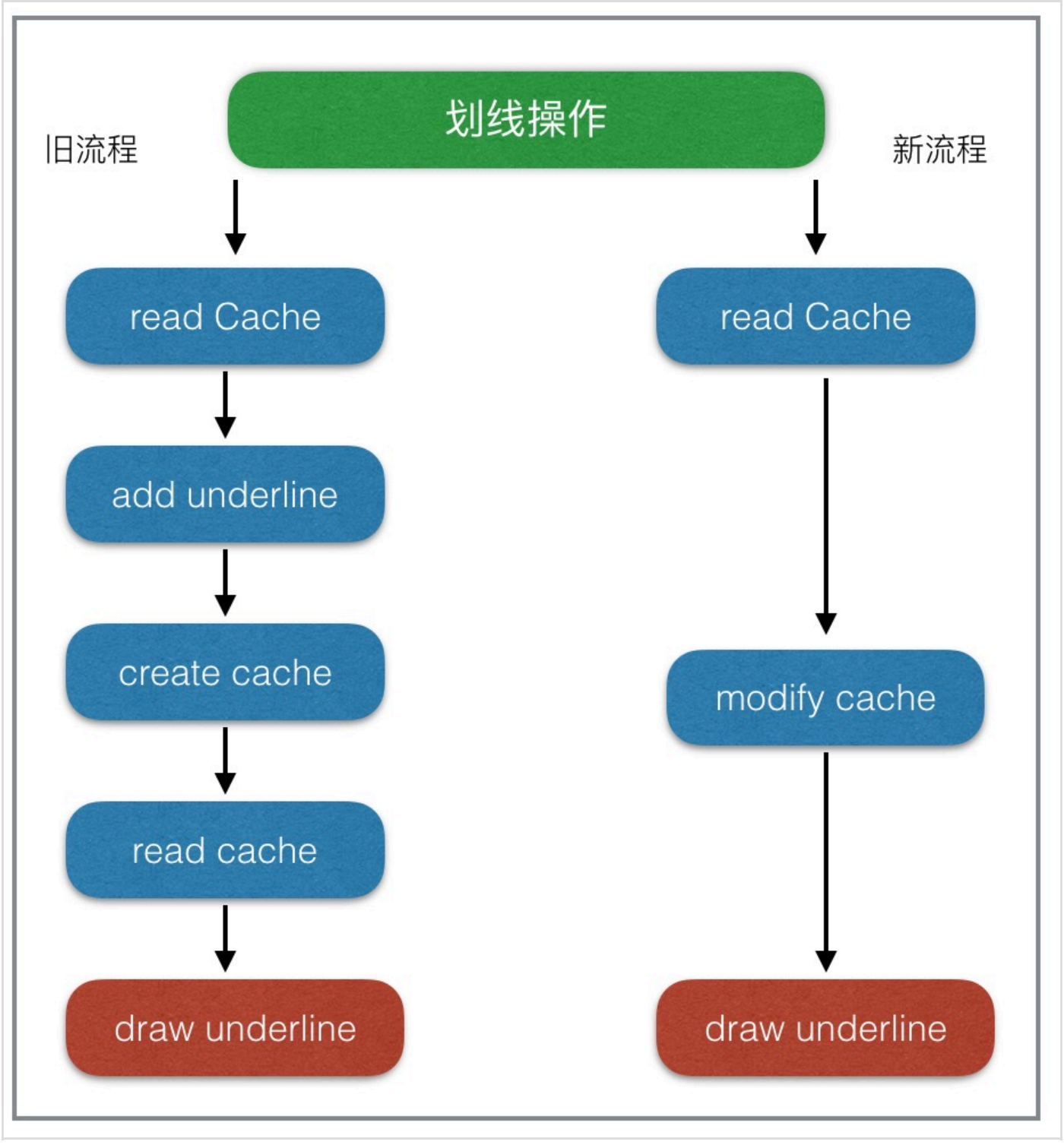
微信读书接入了RDM(bugly)的卡顿监控(也是基于微信团队的卡顿标准)，通过下发配置，对现网用户进行抽样检测，并上报卡顿的堆栈信息。这对于我们掌握现网用户的卡顿状况起到了非常大的帮助。

性能问题的解决方法

产生性能问题的原因多种多样，因此解决的办法也不尽相同，比较常用的大概有以下几种：

1.优化业务流程

性能优化看似高深，真正落到实处才会发现，最大的坑往往都隐藏在于业务不断累积和频繁变更之处。优化业务流程就是在满足需求的同时，提出更加高效优雅的解决方案，从根本上解决问题。从实践来看，这种方法解决问题是最彻底的，但通常也是难度最大的。微信读书在优化阅读中各种操作（如，书签、划想、想法等）性能时，就是从业务流程的角度来进行优化。如下图：



2.合理的线程分配

由于 GCD 实在太方便了，如果不加控制，大部分需要抛到子线程操作都会被直接加到 global 队列，这样会导致两个问题，1.开的子线程越来越多，线程的开销逐渐明显，因为开启线程需要占用一定的内存空间（默认的情况下，主线程占1M,子线程占用512KB）。2.多线程情况下，网络回调的时序问题，导致数据处理错乱，而且不容易发现。为此，我们项目定了一些基本原则。

- UI 操作和 DataSource 的操作一定在主线程。
- DB 操作、日志记录、网络回调都在各自的固定线程。
- 不同业务，可以通过创建队列保证数据一致性。例如，想法列表的数据加载、书籍章节下载、书架加载等。

合理的线程分配，最终目的就是保证主线程尽量少的处理非UI操作，同时控制整个App的子线程数量在合理的范围内。

3.预处理和延时加载

预处理，是将初次显示需要耗费大量线程时间的操作，提前放到后台线程进行计算，再将结果数据拿来显示。

延时加载，是指首先加载当前必须的可视内容，在稍后一段时间内或特定事件时，再触发其他内容的加载。这种方式可以很有效的提升界面绘制速度，使体验更加流畅。（UITableView 就是最典型的例子）

这两种方法都是在资源比较紧张的情况下，优先处理马上要用到的数据，同时尽可能提前加载即将要用到的数据。在微信读书中阅读的排版是优先级最高的，所在在阅读过程中会预处理下一页、下一章的排版，同时可能会延时加载阅读相关的其它数据（如想法、划线、书签等）。

4.缓存

cache可能是所有性能优化中最常用的手段，但也是我们极不推荐的手段。cache建立的成本低，见效快，但是带来维护的成本却很高。如果一定要用，也请谨慎使用，并注意以下几点：

- 并发访问 cache 时，数据一致性问题。
- cache 线程安全问题，防止一边修改一边遍历的 crash。
- cache 查找时性能问题。
- cache 的释放与重建，避免占用空间无限扩大，同时释放的粒度也要

依实际需求而定。

5.使用正确的API

使用正确的 API，是指在满足业务的同时，能够选择性能更优的API。

- 选择合适的容器;
- 了解 `imageNamed:` 与 `imageWithContentsOfFile:` 的差异(`imageNamed:` 适用于会重复加载的小图片，因为系统会自动缓存加载的图片， `imageWithContentsOfFile:` 仅加载图片)
- 缓存 `NSDateFormatter` 的结果。
- 寻找 `(NSDate *)dateFromString:(NSString )string` 的替换品。

```
//#include <time.h>
time_t t;
struct tm tm;
strptime([iso8601String cStringUsingEncoding:NSUTF8StringEnco
tm.tm_isdst = -1;
t = mktime(&tm);
[NSDate dateWithTimeIntervalSince1970:t + [[NSTimeZone localT
```

- 不要随意使用 `NSLog()`.
- 当试图获取磁盘中一个文件的属性信息时，使用 `[NSFileManager attributesOfItemAtPath:error:]` 会浪费大量时间读取可能根本不需要的附加属性。这时可以使用 `stat` 代替 `NSFileManager`，直接获取文件属性:

```
#import <sys/stat.h>
struct stat statbuf;
const char *cpath = [filePath fileSystemRepresentation];
```

```
4   if (cpath && stat(cpath, &statbuf) == 0) {
5       NSNumber *fileSize = [NSNumber numberWithUnsignedLongLong
6       NSDate *modificationDate = [NSDate dateWithTimeIntervalSi
7       NSDate *creationDate = [NSDate dateWithTimeIntervalSince1
8       // etc
9   }
```

如何预防性能问题

大部分性能问题可以通过程序员经验和能力的提升得以减少，但是因为团队成员更新、业务累积，性能问题无法避免，如何在开发测试阶段发现问题解决问题，是预防性能问题的关键。为此，我们开发了一些比较有意思的工具，用于发现各种性能问题。

1. 内存泄露检测工具

MLeakFinder是团队成员zepo在github开源的一款内存泄露检测工具，具体原理和使用方法可以参见这篇文章。在此之前，内存泄露引起的性能问题是很难被察觉的，只有泄露到了相当严重的程度，然后通过Instrument工具，不断尝试才得以定位。MLeakFinder能在开发阶段，把内存泄露问题暴露无遗，减少了很多潜在的性能问题。

2. FPS/SQL性能监测工具条

该工具条是在DEBUG模式下，以浮窗的形式，实时展示当前可能存在问题的FPS次数和执行时间较长的SQL语句个数，是团队成员tower的杰作。FPS监测的原理并不复杂，前文也有介绍，虽然并不百分百准确，但非常实用，因为可以随时查看FPS低于某个阈值时的堆栈信息，再结合当时的使用场景，开发人员使用起来非常便利，可以很快定位到引起卡顿的场景和原因。SQL语句的监测也非常实用，对于微信读书，DB的读写速度是影响性能的瓶颈之一。因此在DEBUG阶段，我们监测了每一条SQL语句的执行速度，一旦执行时间超出某个阈值，就会表现在工具条的数字上，点击

后可以进一步查询到具体的SQL操作以及实际耗时。

这个工具帮助我们在开发阶段发现了很多卡顿问题，尤其是一些不合理的SQL语句，例如：
在想法圈的优化过程中，利用这个工具，我们就发现想法圈第一次加载更多，执行的SQL语句耗时竟然达到了1000多毫秒。

```
_SELECT * FROM WRReview INNER JOIN WRUser ON WRReview.fromId
```

通过explain，可以发现这条SQL效率之低:

```
SEARCH TABLE WRReview
SEARCH TABLE WRUser USING INTEGER PRIMARY KEY (rowid=?)
USE TEMP B-TREE FOR ORDER BY
```

- 没有建立合适的索引，导致WRReview全表扫描。
- 排序字段没有索引，导致SQLite需要再一次B-TREE排序。
- 两字段排序，性能更低。

优化：给WRReview的 `fromId` `createTime` 两个字段增加了索引，并去掉一个排序字段:

```
SELECT * FROM WRReview INNER JOIN WRUser ON WRReview.fromId =
```

Explain的结果：

```
SCAN TABLE WRReview USING INDEX WRReview_createTime
SEARCH TABLE WRUser USING INTEGER PRIMARY KEY (rowid=?)
```

SQL执行时间直接降了一个数量级，到100毫秒左右。

3. UI / DataSource主线程检测工具。

该工具是为了保证所有的UI的操作和 DataSource 操作一定是在主线程进行，同样是由tower同学贡献。实现原理是通过 hook UIView 的 `-setNeedsLayout`，`-setNeedsDisplay`，`-setNeedsDisplayInRect` 三个方法，确保它们都是在主线程执行。子线程操作UI可能会引起什么问题，苹果说得并不清楚，实际开发中我们遇到几种神奇的问题似乎都是跟这个有关。

- app 突然丢动画，似乎 iOS 系统也有这个 bug。虽然没有确切的证据，但使用这个工具，改完所有的问题后，bug 也好了(不止一次是这样)。
- UI 操作偶尔响应特别慢，从代码看没有任何耗时操作，只是简单的 push 某个 controller。
- 莫名的 crash，这当然是因为 UI 操作非线程安全引起的。

更多时候，子线程操作 UI 也并不一定会发生什么问题，也正因为不知道会发生什么，所以更需要我们警惕，这个工具替我们扫除了这些隐患。虽然，苹果表示，现在部分的 UI 操作也已经是线程安全了，但毕竟大部分还不是。DataSource 的监测是因为我们业务定下的原则，保证列表 DataSource 的线程安全。

4. 排版引擎自动化检测工具

排版引擎是微信读书最核心的功能，排版引擎检测工具原本是为了检验排版引擎改进过程中准确性，防止因为业务变更，而影响原来的排版特性。实现原理是结合自动化脚本和 App 本身的排版引擎，给书库中的每一本书建立一个镜像，镜像的内容包括书籍的每一章每一页的截图，然后分析同

一页码的两个不同版本的图片差异，就可以知道不同版本的排版引擎渲染效果。但是我发现，只要稍加改进，排版后记录每个章节排版耗时，就可以知道每个版本变化后同一个章节的耗时变化，以此作为排版引擎的性能指标。这个工具保证了微信读书，即使在快速迭代过程中也不会丢失阅读的核心体验。虽然这个工具无法在其它项目中复用，但是提醒了我们，可以通过自动化工具来保证产品最核心功能的体验。

5. 书源检测工具

微信读书为了支持正版版权，目前书源完全依赖于后台，不允许本地导入。书源的优劣的直接影响排版的效果和性能。为了解决了部分书籍无法打开或者乱码的问题，我们借助了后台同学的书源检测工具。对线上所有 epub 书籍进行扫描，按照章节大小进行排序。对于章节内容特别大的书籍重点检测，重新排版，解决了一批 epub 书籍无法打开的问题。同时针对章节内容乱码的问题，对所有 txt 的书籍进行了一次全量扫描，发现了一些问题，但还无法准确找出所有乱码的章节，这一点还在努力改善中。

优化成果

1. 整体使用感受上，已经可以明显区分两个版本的性能差异，这一点也可以通过每天的用户反馈数据中得到验证。1.3.0 和 1.3.1分别发布一周后反馈的卡顿数从 10 个降到了 3 个，从总体反馈比例的 2.8% 降到 0.8%。
2. 某些关键业务，耗时也有明显改善。

关键路径	1.3.0（ms）	1.3.1(ms)	优化减少
启动时间	1929.48	1839.65	**7%**
打开发现	85.02	53.44	**33%**
打开书籍详情	18.1	9.21	**49%**
打开书籍	273.7	226.75	**17%**
翻页	8.72	7.47	**14%**
打开书城分类	130.89	116.82	**15%**
打开书城推荐	58.71	40.15	**31%**

3. 极端案例的修复。超大的epub书籍已通过后台进行拆分，解决了无法打开书籍的情况。
4. 针对低端机型，去掉了某些动画，交互更加流畅。

总结

通过上述介绍，我们可以看出，性能问题普遍存在，无可避免，与其花费大量时间，查找线上版本的性能问题，不如提高整体团队成员性能优化意识，借助性能查找工具，将性能问题尽早暴露在开发阶段，达到预防为主的效果。

‹ iOS 组件化方案探索

iOS 启动连续闪退保护方案 ›