链滴

浅谈 App 响应时间优化

作者：xuexiangjys
原文链接：https://ld246.com/article/1682011437638
来源网站：链滴

响应时间，它是用来衡量系统运行效率的一个重要指标。评价一个应用的响应时间，可以从用户感知系统性能这两个角度来考量。

响应时间的长短，可能影响用户对某个功能、某个应用、乃至某个系统的使用。毕竟如果有选择，没哪个人会愿意去使用卡顿的应用，运行慢的手机。

作为一名开发者，虽然我们平时可能只关注于堆业务，根本就没有时间或者机会去优化我们程序的响时间，但是这些内容对我们个人的技术成长是至关重要的。大的不说，这部分也是面试中经常考察的容，知道了也不至于吃亏。

那么接下来我们就长话短说，赶紧来瞧瞧，到底如何来优化我们应用的响应时间。

1. 核心原则

在算法中，我们经常会从 时间复杂度和 空间复杂度这两个纬度来衡量算法的优劣。

很多时候，我们无法做到 时间复杂度和 空间复杂度两者都最佳，只能在"时间"和"空间"中，取折中的优解。同样的，如果我们追求最极致的"时间"最佳，就可能需要牺牲一部分的"空间"，这就是拿"空间换"时间"的解法。

即**响应时间优化的核心**：空间 -> 时间 （用空间换时间）

那么我们应该怎么做呢？下面是我归纳总结出来的四项基本原则：

- 1.缓存优先：能读缓存读缓存。
- 2.减少新建：能复用绝不新建。
- 3.减少任务：能不做的尽量不做。
- 4.具体问题具体分析：针对具体事务本身进行分析，必须做的能提前做就提前做，不必须做的延后。

2. 优化措施

可能我上面说的这些核心和基本原则，对绝大多数人来说都非常好理解，但是知道了这些，并不代表懂得如何进行优化。 这就好比你高中学数学，即便告诉了你一堆的公式，但真要让你来一道相关的应题，你还真不一定能解得出来，这个时候"例题"就很关键了。

同样的，即便你知道了一些关于应用响应时间优化的核心和原则后，当你真正面临具体的优化问题时你可能也会手足无措。

所以，接下来我就从 任务执行、资源加载、数据结构、线程/IO和 页面渲染这五个角度，来给出我的化建议。

2.1 任务执行

- 1.业务/任务梳理：对业务进行拆分，对任务进行整合。
- 2.任务转换：串行 -> 并行, 同步 -> 异步。
- 3.执行顺序按优先级调整。
- 4.延迟执行、空闲执行，如： IdleHandler。

2.1.1 业务/任务梳理

业务往往是由一个个任务流组合而成。合理的业务/任务粒度可以有效提高响应的速度。

对业务和任务的梳理，正确的方式是先进行业务的拆分，将业务拆分为一个个子任务，再根据需要对任务进行整合。

（1）对不合理的业务流进行拆分。

- 对业务进行拆分，拆分出主要（必要）业务和次要（非必要）业务。
- 分别对主要业务和次要业务进行优先级评估，业务执行按优先级从高到底依次执行。

（2）对任务流进行整合。

- 多个相关的串行任务，可以整合为统一的业务整体。
- 多个不相关的串行任务，可以整合为一个并行的业务。

2.1.2 任务转换

1.串行 -> 并行的适用范围：

- 多个不相关的串行任务。
- 多个任务弱相关且耗时，但是耗时接近。例如某个页面你需要调用多个模块的接口查询数据进行展。

2.同步 -> 异步的适用范围：

- 非必要（重要性不高）且耗时的任务。
- 耗时且关联性不大的任务。
- 耗时且存在一定相关性的任务。使用 异步线程 + 同步锁的方式执行。

2.1.3 任务优先级

类似线程中的优先级Priority，当系统资源紧张的时候，优先执行优先级高的线程。

首先我们要对应用内所有需要优化的业务以及其子任务的优先级进行定义，然后按优先级顺序进行排和执行。

那么如何才能保证任务被按优先级进行执行呢？

1.对于线程，我们可以直接设置其Priority值。（但是一般我们不能直接使用线程，所有这个可以忽略）

2.对于线程池，我们可以从代码层将任务按优先级顺序加入到线程池中。注意，这里的线程池最好是塞式的，例如：使用PriorityBlockingQueue实现的优先级线程池 PriorityThreadPoolExecutor 。

3.使用第三方的任务执行框架，这里推荐我开源的 XTask 供大家参考。

2.1.4 延迟执行

延迟执行，是将一些不必要、重要性不高或者高耗时的任务暂停执行，等后面资源充足或者要使用时执行。

常见的延迟执行有以下几种：

- 延迟某个特定的时间执行。例如：某应用启动后，每隔2分钟同步一下用户状态。
- 待某个特定的任务执行完成之后再执行。例如：导航应用定位获取成功后，再执行目的地推荐获取任务。
- 直接不执行，等相关业务用到的时候再执行。
- 空闲执行，等待页面都完全渲染完毕之后再执行。例如：使用 IdleHandler，具体使用如下：

```
Looper.myQueue().addIdleHandler(new MessageQueue.IdleHandler() {
    @Override
    public boolean queueIdle() {
        // 执行你的任务
        return false;
    }
});
```

当然，如果你想在空闲的时候执行多个任务，你也可以这样写：

```
public class DelayTaskQueue {

  private final Queue<Runnable> mDelayTasks = new LinkedList<>();

  private final MessageQueue.IdleHandler mIdleHandler = () -> {
    if (mDelayTasks.size() > 0) {
      Runnable task = mDelayTasks.poll();
      if (task != null) {
        task.run();
      }
    }
    // mDelayTasks非空时返回ture表示下次继续执行，为空时返回false系统会移除该IdleHandler不
执行
    return !mDelayTasks.isEmpty();
  };

  public DelayTaskQueue addTask(Runnable task) {
    mDelayTasks.add(task);
    return this;
  }

  public void start() {
    Looper.myQueue().addIdleHandler(mIdleHandler);
  }
}
```

2.2 资源加载

- 1.懒加载
- 2.分段加载（部分加载）
- 3.预加载（数据、布局页面等）

2.2.1 懒加载

对于一些不常用或者不重要的数据、图片、控件以及其他一些资源，我们可以在用到时再进行加载。

1.数据懒加载

- kotlin中的 lazy标签：修饰val变量，程序第一次使用到这个变量(或者对象)时再初始化。
- Map、List和SharedPreferences等大数据的延迟初始化。

```
private Map getSystemSettings() {
    if (mSettingMap == null) {
        mSettingMap = initSystemSettings();
    }
    return mSettingMap;
}
```

2.图片资源懒加载

- 对于不常用的图片，可以使用云端图片的资源url来替代。
- 对于非程序预置的图片（本地图片文件或者云端图片），用到时再加载。

3.控件懒加载

- 使用ViewStub进行布局的延迟加载。
- 使用ViewPager2+Fragment进行Fragment的懒加载。
- 使用RecyclerView替代ListView。

2.2.2 分段加载

分段加载常见应用于大数据的加载，这里包括大图和长视频等多媒体资源的加载。做到用到哪，加载哪，完全不必要等全部加载完才给用户使用。

1.大图的分段加载：对于大图，我们可以将其按一定尺寸进行切分，分割成一块一块的小瓦片，然后定一个预览预加载范围，用户预览到哪里我们就加载到哪里。（就类似地图的加载）

2.长视频的分段加载：对于长视频，我们可以将其按时间片进行拆分，并设置一个加载缓存池。这样户浏览一个长视频时，就可以快速打开加载。

3.大文件或者长WebView的分段加载：对于一些阅读类的app，经常会遇到大文件和长WebView的载，这里我们也可以同理对其进行拆分处理。

2.2.3 预加载

分段加载常和预加载一起组合使用。对于一些加载非常耗时的内容，我们可以将加载时机提前，从而小用户感知的加载时间。

预加载的本质是提前加载，这样这个提前加载的时机就非常的关键和重要。因为预加载时机如果太晚几乎看不出效果；但是如果预加载的时机过早，有可能抢占其他模块资源，造成资源紧张。

那么我们何时可以触发预加载，预加载的时机是什么呢？下面我举几个简单的例子。

1.用户操作时。如果用户点击了第2章，我们就开始预加载下一章和上一章；用户上滑到了第3页，我预加载第4页，用户下滑到第5页，我们预加载第4页.

2.应用空闲时。例如之前说的 IdleHandler。或者在 onUserInteraction中监听用户的操作，一段时没有操作即视为空闲。

3.耗时等待时。对于一些常见的耗时操作，我们可以在其开始时，并行进行一些预加载操作，从而提时间的利用率。例如Activity的创建比较耗时，我们可以在startActivity前就开始预加载数据，这样Activity创建完之后有可能数据就已经加载好了，直接可以拿来渲染。例如一些有开屏广告的app，可以广告开始时，同步进行一些数据资源的预加载。

2.3 数据结构

- 1.数据结构优化（空间大小、读取速度、复用性、扩展性）。
- 2.数据缓存（内存缓存、磁盘缓存、网络缓存），分段缓存。这里可以参考glide.
- 3.锁优化（减少过度锁，避免死锁），悲观锁/乐观锁。
- 4.内存优化，避免内存抖动，频繁GC（尤其关注bitmap）

2.3.1 数据结构优化

不同的数据结构有不同的使用场景，选择适合的数据结构能够事半功倍。

1.ArrayList和LinkedList：

- ArrayList：底层数据结构是数组，查询快、增删慢。
- LinkedList：底层数据结构是链表，查询慢、增删快。

2.HashMap和SparseArray：

- HashMap：底层数据结构是数组和链表（或红黑树）的组合，结合了ArrayList和LinkedList的优，查询快、增删也快。但是扩容很耗性能，且空间利用率不高(75%)，浪费内存。
- SparseArray：底层数据结构是双数组，一个数组存key，一个数组存value。使用二分法查询进行化，在数据量小（一百条以下）的情况下，速度和HashMap相当，但是空间利用率大大提升。
- ArrayMap：底层数据结构是双数组，一个数组存key的hash值，一个数组存value。设计与SparseArray类似，在数据量小的情况下，可完全替代HashMap。

3.Set: 保证每个元素都必须是唯一的。

4.TreeSet和TreeMap：有序的集合，保证存放的元素是排过序的，速度慢于HashSet和HashMap。

可以看到，在不考虑空间利用率的情况下，HashMap的性能是不错的。

但是由于存在初始化大小和扩展因子对其性能有所影响，我们在使用时，尽量根据实际需要设置合理初始化大小：避免设置小了扩容带来性能消耗，设置大了造成空间浪费。

因为HashMap的默认扩容因子是0.75，如果你实际使用的数量是8，那你初始化大小就设置16；如果实际使用的数量是60，那你初始化大小就设置128。

2.3.2 数据缓存

对于一些变化不是很频繁的数据资源，我们可以将其缓存下来。这样我们下次需要使用它们的时候，可以直接读取缓存，这样极大地减少了加载和渲染所需要的时间。

一般意义上的缓存，按读取的时间由快到慢，我们可分为内存缓存、磁盘缓存、网络缓存。

- 内存缓存，就是存储在内存中，我们可以直接读取使用。而如果从界面渲染的角度，我们又可以将存缓存分为Active(活跃/正在显示)缓存和InActive(非活跃/不可显示)缓存。
- 磁盘缓存，就是存储在磁盘文件中，每次读取都需要将磁盘文件内容读取到内存中，方可使用。
- 网络缓存，就是存储在远端服务器中，每次读取需要我们进行一次网络请求。一般来说，我们也可将一次网络缓存请求到的数据缓存到磁盘中，将网络缓存转化为磁盘缓存，通过减少网络请求，来提读取速度。

某种意义上来说，内存缓存、磁盘缓存和网络缓存，它们又是可以相互转化的，一般来说，我们会将络缓存->磁盘缓存->内存缓存，进行使用，从而提升读取速度。

具体我们可以参考glide框架和RecyclerView的实现原理。

2.3.3 锁优化

锁是我们解决并发的重要手段，但是如果滥用锁的话，很可能造成执行效率下降，更严重的可能造成锁等无法挽回的场景。

当我们需要处理高并发的场景时，同步调用尤其需要考量锁的性能损耗：

- 能用无锁数据结构，就不要用锁。
- 缩小锁的范围。能锁区块，就不要锁住方法体;能用对象锁，就不要用类锁。

那么我们具体应该怎么做呢？下面我简单讲几个例子。

1.使用乐观锁代替悲观锁，轻量级锁代替重量级锁。

利用 CAS机制, 全称是Compare And Swap，即先比较，然后再替换。就是每次执行或者修改某个变时，我们都会将新旧值进行比较，如果发生偏移了就更新。这就好比在一些无锁的数据库中，每次的据库操作都会携带一个唯一的版本号，每次进行数据库修改的时候都会对比一下数据库记录和操作请的版本号，如果版本号是最新的版本号，则进行修改，否则丢弃。

需要注意的是，CAS必须借助 volatile才能读取到共享变量的最新值来实现【比较并交换】的效果，为 volatile会保证变量的可见性。

在Java中，JDK给我们默认提供了一些 CAS机制实现的原子类，如 AtomicInteger、AtomicReferen e等。

2.缩小同步范围，避免直接使用 synchronized，即使使用也要尽量使用同步块而不是同步方法。多用JDK提供给我们的同步工具：CountDownLatch，CyclicBarrier，ConcurrentHashMap。

3.针对不同使用场景，使用不同类型的锁。

- 针对并发读多，写少的，我们可以使用读写锁（多个读锁不互斥，读锁与写锁互斥）：ReentrantR adWriteLock，CopyOnWriteArrayList，CopyOnWriteArraySet。
- 针对某一个并发操作通常由某一特定线程执行时，可尝试使用偏向锁（偏向于第一个获得它的线程 。
- 针对存在大量并发资源竞争的场景，推荐使用重量级锁synchronized。

2.3.4 内存优化

内存优化的核心是避免内存抖动。不合理的内存分配、内存泄漏、对象的频繁创建和销毁，都会导致存发生抖动，最终导致系统的频繁GC。

频繁的GC，必定会导致系统运行效率的下降，严重的可能会导致页面卡顿，造成不好的用户体验。么我们应该着手从哪些地方进行优化呢？

- 解决应用的内存泄漏问题。这里我们可以使用 LeakCanary 或者 Android Profile 等工具来检查们查询可能存在的内存泄漏。
- 平时编码应当注意避免内存泄漏。如避免全局静态变量和常量、单例持有资源对象（Activity，Fra ment，View等），资源使用完立即释放或者recycle（回收）等。
- 避免创建大内存对象，频繁创建和释放对象（尤其是在循环体内），频繁创建的对象需要考虑复用者使用缓存。
- 加载图片可以适当降低图片质量，小图标尽量使用SVG，大图/复杂的图片考虑使用webp。尽量使图片加载框架，如glide，这些框架都会帮我们进行加载优化。
- 避免大量bitmap的绘制。
- 避免在自定义View的 onMeasure、onLayout和 onDraw中创建对象。
- 使用SpareArray、ArrayMap替代HashMap。
- 避免进行大量的字符串操作，特别是序列化和反序列化。不要使用+(加号)进行字符串拼接。
- 使用线程池（可设置适当的最大线程池数）执行线程任务，避免大量Thread的创建及泄漏。

2.4 线程/IO

- 1.线程优化（统一、优先级调度、任务特性）
- 2.IO优化（网络IO和磁盘IO），核心是减少IO次数
  - 网络：请求合并，请求链路优化，请求体优化，系列化和反序列化优化，请求复用等。
  - 磁盘：文件随机读写、SharePreference读写等（例如对于读多写少的，可使用内存缓存）
- 3.log优化（循环中的log打印，不必要的log打印，log等级）

2.4.1 线程优化

当我们创建一个线程时，需要向系统申请资源，分配内存空间，这是一笔不小的开销，所以我们平时发的过程中都不会直接操作线程，而是选择使用线程池来执行任务。所以线程优化的本质是对线程池优化。

线程池使用的最大问题就在于如果线程池设置不对的话，很容易被人滥用，引发内存溢出的问题。而通常一个应用会有多个线程池，不同功能、不同模块乃至是不同三方库都会有自己的线程池，这样大各用各的，就很难做到资源的协调统一，劲不往一处使。

那么我们应该如何进行线程池优化呢？

1.建立主线程池+副线程池的组合线程池，由线程池管理者统一协调管理。主线程池负责优先级较高任务，副线程池负责优先级不高以及被主线程池拒绝降级下来的任务。

这里执行的任务都需要设置优先级，任务优先级的调度通过 PriorityBlockingQueue队列实现，以下主副线程池的设置，仅供参考：

- 主线程池：核心线程数和最大线程数：2n（n为CPU核心数），60s keepTime，PriorityBlockingQ

eue（128）。

- 副线程池：核心线程数和最大线程数：n（n为CPU核心数），60s keepTime，PriorityBlockingQue（64）。

2.使用Hook的方式，收集应用内所以使用 newThread方法的地方，改为由线程池管理者统一协调管。

3.将所有提供了设置线程池接口的第三方库，通过其开放的接口，设置为线程池管理者管理。没有提设置接口的，考虑替换库或者插桩的方式，替换线程池的使用。

2.4.2 IO优化

IO优化的核心是减少IO次数。

1.网络请求优化。

- 避免不必要的网络请求。对于那些非必要执行的网络请求，可以延时请求或者使用缓存。
- 对于需要进行多次串行网络请求的接口进行优化整合，控制好请求接口的粒度。比如后台有获取用信息的接口、获取用户推荐信息的接口、获取用户账户信息的接口。这三个接口都是必要的接口，且在先后关系。如果依次进行三次请求，那么时间基本上都花在网络传输上，尤其是在网络不稳定的情下耗时尤为明显。但如果将这三个接口整合为获取用户的启动（初始化）信息，这样数据在网络中传的时间就会大大节省，同时也能提高接口的稳定性。

2.磁盘IO优化

- 避免不必要的磁盘IO操作。这里的磁盘IO包括：文件读写、数据库（sqlite）读写和SharePreferene等。
- 对于数据加载，选择合适的数据结构。可以选择支持随机读写、延时解析的数据存储结构以替代ShaePreference。
- 避免程序执行出现大量的序列化和反序列化（会造成大量的对象创建）。

2.5 页面渲染

下面是我简单列举的几点加快页面渲染的方法，相信大家或多或少都用过，这里我就不详细阐述了：

- 1.降低布局层级、减少嵌套、避免过度渲染（背景）(merge，ConstraintLayout)
- 2.页面复用(include)
- 3.页面懒加载
- 4.布局延迟加载(ViewStub)
- 5.inflate优化（布局预加载+异步加载，动态new控件/X2C）
- 6.动画优化（注意动画的执行耗时和内存占用，不可见时暂停动画，可见时再恢复动画）
- 7.自定义view优化（减少onDraw、onLayout、onMeasure的对象创建和执行耗时）
- 8.bitmap和canvas优化（bitmap大小、质量、压缩、复用；canvas复用：clipRect，translate）
- 9.RecycleView优化（减少刷新次数，缓存复用）

3. 推荐工具

- systrace、 Perfetto 、Android Profile
- DoKit
- LeakCanary
- performance

最后

还是那句话，百闻不如一见，百见不如一试。写了这么多，我还是希望大家在平时开发的过程中，多视一些应用响应时间优化的相关技巧，让我们开发出流畅顺滑的应用吧。（尽管很多时候，我们所谓优化会被产品或者设计diss）