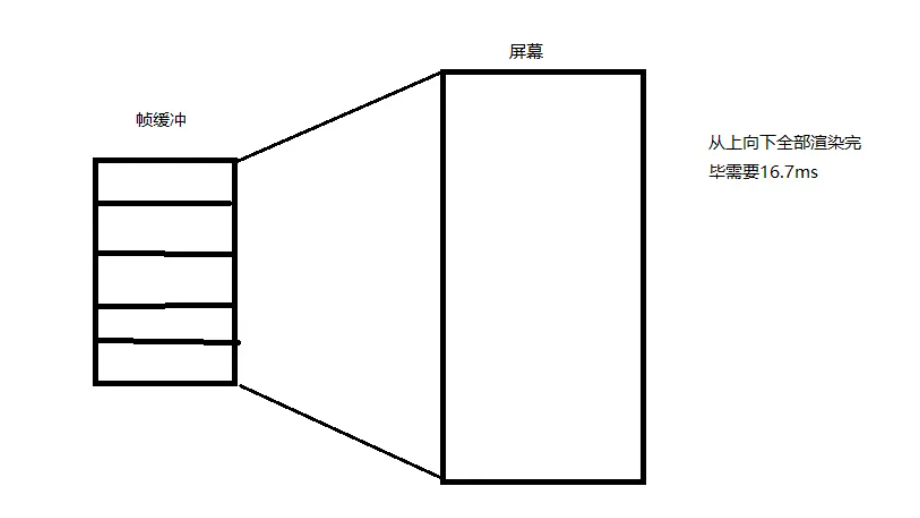
屏幕就是专门来"渲染"图像的，既然要显示图像，肯定要有显示的数据，这些数据从哪来呢？答案就是cpu(**这里为了方便，把cpu、gpu、sf等统一称为cpu**)，这些数据由cpu提供，cpu经过各种运算，将数据写入一块内存中，这块内存叫做**帧缓冲**，我们可以将帧缓冲理解为一个M\*N矩阵，数据从上到下一行一行保存，显示器在显示的时候，从上到下逐行扫描，依次显示在屏幕上，我们把这样的一屏数据叫做**一帧**，当一帧数据渲染完后，就开始新一轮扫描，如果CPU**正好**(不正好后面再说)也把下一帧数据写入帧缓冲，那么就会显示下一帧画面，如此循环，我们就看到了不断变化的画面，也就是图像。这个过程很简单，但是实现起来却很难，具体有两点:

* 1 屏幕需要在16.7毫秒内绘制完一帧，因为根据研究，16.7ms正符合人类能觉察到卡顿的分割点，如果低于16.7ms，则可能感觉卡顿，高于16.7ms则没必要。
* 2 CPU需要在屏幕渲染完毕后，正好把下一帧数据写入帧缓冲。如果早了，那么屏幕上就会绘制一半上一帧的数据，一半下一帧的数据。比如:绘制到第一帧的a行时，cpu把下一帧数据送进来了，屏幕会接着从a+1行接着绘制，这样导致前a行是第一帧的数据，后面几行是第二帧的数据，在我们看来就是两张图片撕开各取一部分拼起来，这叫做**撕裂**。如果晚了，那么屏幕会在下一次继续绘制上一帧，导致画面没有变化，这样就会出现画面不变的情况，在我们看起来就是卡了，也叫做**卡顿**。所以，CPU和屏幕的这个交互时机很重要。



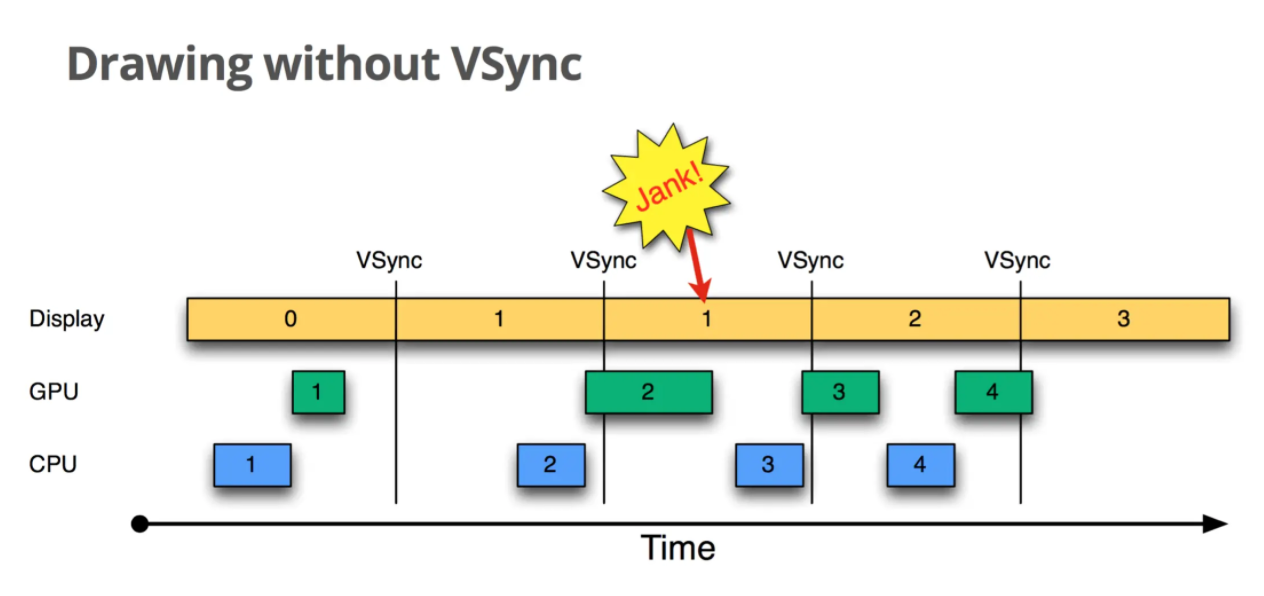
我们来看两个概念:

* 屏幕刷新率(Hz): 屏幕在一秒内刷新的次数，Android手机一般都是60Hz，也就是一秒刷新60次，当然也有高刷的，但是60Hz足矣。
* 帧速率(FPS): cpu在一秒内合成的帧数，比如60FPS，就是60 frame per sconds，意思就是一秒合成60帧。

如上所述，当屏幕刷新率大于帧速率的时候，会发生卡顿；屏幕刷新率小于帧速率的时候，会发生撕裂。

**解决撕裂问题(VSYNC)**

我们知道，撕裂是因为: cpu太快 从而导致 屏幕还没渲染完毕 就把正在渲染的数据 给覆盖掉了，那么我们可以限制cpu的速度吗？当然可以，但是不划算，因为这样就等于把cpu的长处给扼杀了，所以我们只要让cpu的数据不覆盖掉屏幕正在渲染的数据即可，也就是说，给cpu新来的数据提供一个存放点，而不是往帧缓冲里面写，这个存放点叫做**后缓冲(BackBuffer)**，相应的，**帧缓冲(FrameBuffer)也叫做前缓冲**，这样，cpu新来的数据就会放在后缓冲，而屏幕则继续从前缓冲取数据来渲染，等到后缓冲数据写入完了，前后缓冲的数据就会交换，屏幕此时读取的数据就是后缓冲的数据，也就是下一帧的数据，循环往复，我们就看到了画面。但是！还是不行，举个列子，如果cpu非常快，前缓冲数据还没刷新完毕，后缓冲已经写满，此时，就会交换数据，又发生了撕裂！



从图中可以看到: 没有vsync的情况下，cpu在任意地方开始，随心所欲!

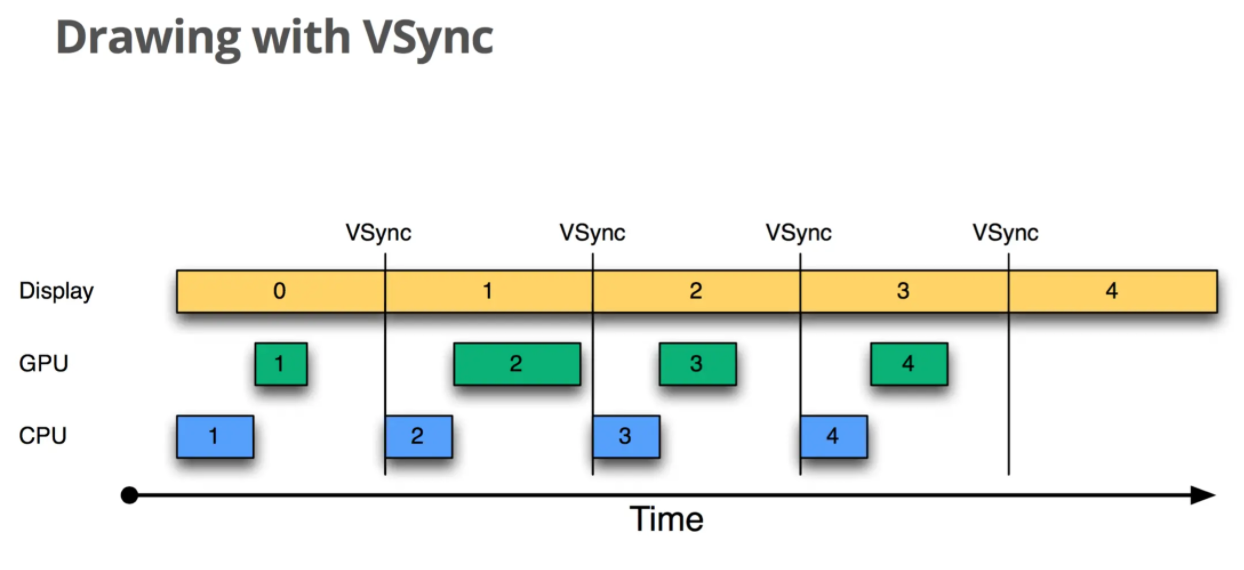
我们追究原因: 核心点在与数据交换的时机由谁来控制，**数据交换的发生点应该是在屏幕渲染完一帧后，而不是cpu写入一帧数据后**，所以，控制数据是否交换应该由屏幕来决定，但是！计算机五大组成部分各司其职，屏幕只是输出设备和输入设备(因为能触屏)，他不是控制器，如何控制数据的交换呢？当然可以，答案就是:VSYNC。

VSYNC(vertical sync): 也就是垂直同步，当屏幕渲染完一帧数据后，即将开始渲染下一帧之前，发出的一个同步信号。

cpu只要监听VSYNC信号，接收到信号后再开始交换后缓冲和前缓冲的数据，就等价于屏幕控制了数据交换，也就解决了撕裂问题，这很明显是设计模式中的监听器模式。

现在我们来捋一下流程:

* 1 屏幕正在从前缓冲读取第一帧数据并渲染，此时cpu计算完第二帧数据，放在后缓冲，等待VSYNC信号。
* 2 屏幕将第一帧数据渲染完毕，发出VSYNC信号，cpu收到VSYNC信号，将后缓冲的第二帧数据复制到前缓冲。
* 3 同时屏幕继续绘制第二帧数据，cpu开始计算下一帧数据，循环往复。



从图中可以看到，有了VSYNC，cpu总是在指定的地方开始。

有人会问: 说白了，真正解决问题的是VSYNC，而不是双缓冲，那不要双缓冲只要VSYNC不是也可以吗？

好，我们假设只有VSYNC，现在假设屏幕正在渲染数据，而cpu在等VSYNC信号，屏幕将数据渲染完毕后，发送VSYNC信号，cpu收到信号后，就去计算数据，计算完后才会写入帧缓冲，那么，在cpu计算数据这段时间内，屏幕干什么呢？嗯，它接着刷新帧缓冲的数据，反正cpu还没有将新数据计算完毕刷入帧缓冲，所以还是上一帧的数据，这样就会卡顿，说白了，有双缓冲的情况下，cpu使用后缓冲计算数据，屏幕使用前缓冲渲染数据，两者可以同时工作，你计算一个我渲染一个，典型的"生产者消费者模式"，只不过使用VSYNC信号来进行数据的交换；而没有双缓冲的情况下，两者需要排队使用帧缓冲，不能同时工作，就变成了我等着你计算，你计算完了等着我渲染，VSYNC此时的作用就是进行排队，这样会大大增加卡顿率，所以: VSYNC真正**解决了撕裂问题**，而双缓**冲优化了卡顿问题**。

那么，怎么解决卡顿问题呢？答曰: 无法根本解决，只能优化!

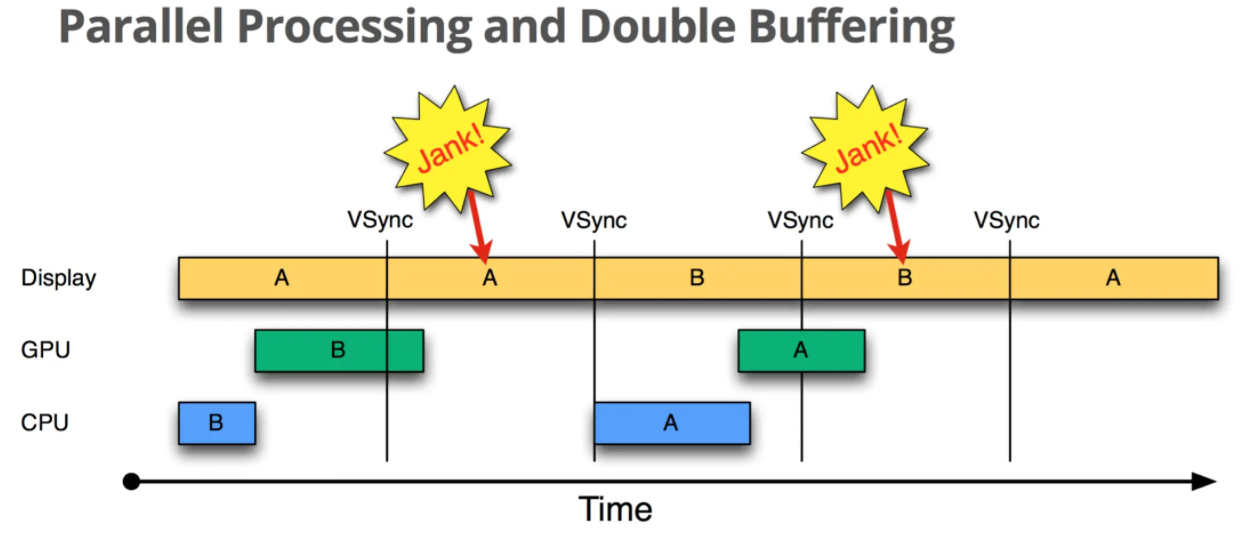
**优化卡顿问题(多缓冲)**

我们知道，卡顿是因为**帧速率<屏幕刷新率**，这是不严谨的，准确的说应该是因为:**帧速率<60fps**，因为现在屏幕刷新率基本都是60hz的，所以帧速率只要取下限60fps即可，换句话说，1秒内需要计算60个帧，也就是16.7ms就能计算完一帧。如果计算不完，那么在一个vsync信号过来后，cpu还在计算，缓冲区的数据并没有改变，就还是老数据，屏幕就又把老数据刷新一遍，就出现了卡顿，所以，cpu要尽可能在16.7ms内把所有数据计算完准备好，以等待vsync信号过来后直接交换数据。

我们又知道，双缓冲只是优化了卡顿问题，并没有根本解决卡顿问题，为何呢？我们先来大致说明一下Android的屏幕绘制流程:

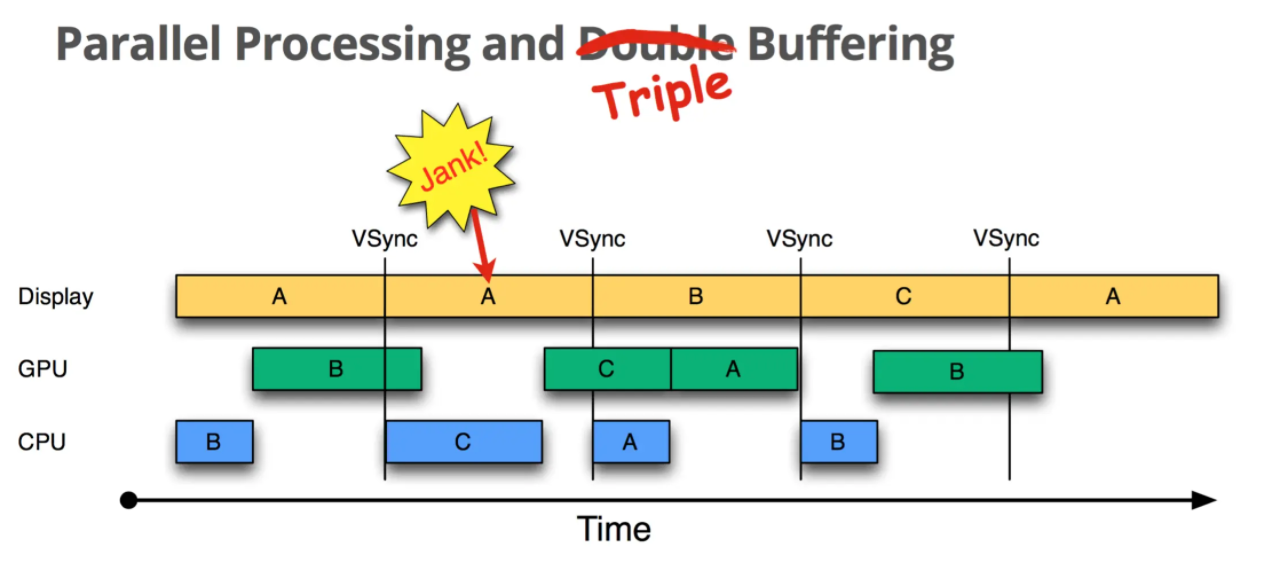
* 1 任何一个View都是依附于window的
* 2 一个window对应一个surface
* 3 view的measure、layout、draw等均是计算数据，这些是cpu干的事
* 4 cpu把这些事干好后，在经过一系列计算将数据转交给gpu
* 5 gpu将数据栅格化后，就交给SurfeceFlinger(以下简称SF)
* 6 SF将多个surfece数据合并处理后，就放入后缓冲区
* 7 屏幕以固定频率从前缓冲区拿出数据渲染，渲染完毕后发送VSYNC，此时前后缓冲区数据交换，屏幕绘制下一帧

上述7步是建立在开启硬件加速的情况下的，如果没有硬件加速，就去掉gpu部分，就可以简单理解为cpu直接将数据转交给sf，我们简单整理一下数据的传递流程:  
**cpu -> gpu -> display**，而且我们看到，cpu和gpu是排队工作的，它俩和屏幕是并行工作的。好，我们来看发生卡顿(jank)的场景:



我们可以将Display那一行看作是前缓冲，将GPU和CPU两行叠加起来看作是后缓冲(因为它俩排队使用)，将VSYNC线隔离开的竖行看作一个帧。

我们看到，在第一帧里面，GPU墨迹了半天没搞完，以至于在第二帧里面，Display(屏幕)显示的还是第一帧的A数据，此时就产生了Jank(卡顿)，并且在一个vsync信号过来后，cpu什么都没做，因为gpu占着后缓冲(那个绿色的长B块)，所以cpu只能再等下一个vsync，在下一个vsync里面，cpu终于拿到了后缓冲的使用权，但是cpu计算时间比较长，导致了gpu时间不够用，数据又没算完，再次发生了卡顿，可以说，这次卡顿直接受到了第一次卡顿的影响，试想: 如果在第一次卡顿的时候，cpu也能计算数据，那么，第二次卡顿可能就不存在了，因为cpu已经在第一次卡顿的时候把蓝色的A给计算完了，第二次完全可以让gpu独自计算(绿色的A)，就不存在因为排队导致的时间不够用了，但是！cpu和gpu共用后缓冲，这就导致它们只能轮流使用后缓冲，怎么解决呢？再加一个后缓冲区，让cpu、gpu各用一块。我们来看引入三缓冲后的效果:



我们看到，在第一次jank内，cpu使用了第三块缓冲区，自己计算了C帧的数据，假如此时没有三缓冲，那么cpu就只能再继续等下一个vsync信号，也就是在图中蓝色A块的地方，才能开始计算C帧数据，就又引发下一次卡顿。我们看到，通过引入三缓冲，虽然不能避免卡顿问题，但是却可以大幅优化卡顿问题，尤其是避免连续卡顿，但是，三缓冲也有缺点，就是耗资源，所以系统并非一直开启三缓冲，要想真正解决问题，还需要在cpu层对数据尽量优化，从而减小cpu和gpu的计算量，比如:View尽量扁平化，少嵌套，少在UI线程做耗时操作等。

Tips:

* Android 3.0引入了硬件加速(GPU)。
* Android 4.0默认开启了硬件加速。
* Android 4.1引入了黄油计划(VSYNC)，上层开始接收VSYNC(Choreographer)，并且加入了三缓冲.
* VSYNC不仅控制了后缓冲和前缓冲的数据交换，还控制了cpu何时开始进行绘制计算。