你好，我是李兵。

我们都知道，要想利用JavaScript实现高性能的动画，那就得使用requestAnimationFrame这个API，我们简称rAF，那么为什么都推荐使用rAF而不是setTimeOut呢？

要解释清楚这个问题，就要从渲染进程的任务调度系统讲起，理解了渲染进程任务调度系统，你自然就明白了rAF和setTimeOut的区别。其次，如果你理解任务调度系统，那么你就能将渲染流水线和浏览器系统架构等知识串起来，理解了这些概念也有助于你理解Performance标签是如何工作的。

要想了解最新Chrome的任务调度系统是怎么工作的，我们得先来回顾下之前介绍的消息循环系统，我们知道了渲染进程内部的大多数任务都是在主线程上执行的，诸如JavaScript执行、DOM、CSS、计算布局、V8的垃圾回收等任务。要让这些任务能够在主线程上有条不紊地运行，就需要引入消息队列。

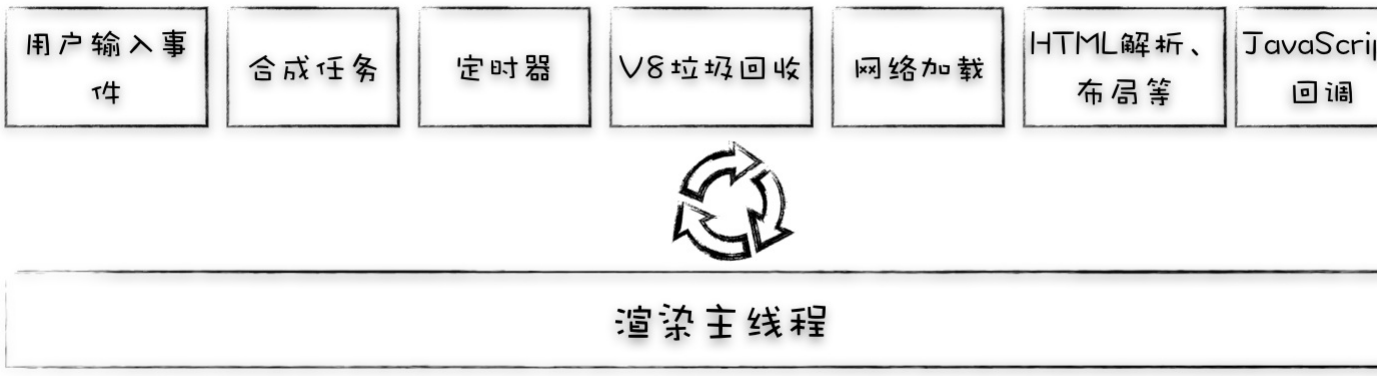
在前面的《16 | WebAPI：setTimeout是如何实现的？》这篇文章中，我们还介绍了，主线程维护了一个普通的消息队列和一个延迟消息队列，调度模块会按照规则依次取出这两个消息队列中的任务，并在主线程上执行。为了下文讲述方便，在这里我把普通的消息队列和延迟队列都当成一个消息队列。

新的任务都是被放进消息队列中去的，然后主线程再依次从消息队列中取出这些任务来顺序执行。这就是我们之前介绍的消息队列和事件循环系统。

单消息队列的队头阻塞问题

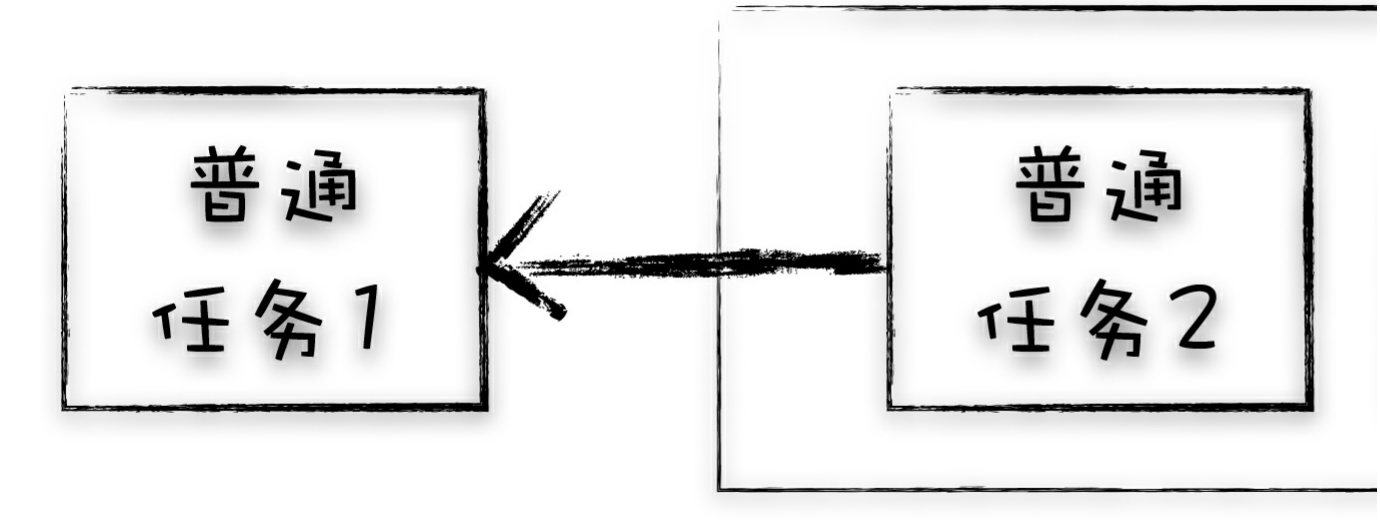
我们知道，渲染主线程会按照先进先出的顺序执行消息队列中的任务，具体地讲，当产生了新的任务，渲染进程会将其添加到消息队列尾部，在执行任务过程中，渲染进程会顺序地从消息队列头部取出任务并依次执行。

在最初，采用这种方式没有太大的问题，因为页面中的任务还不算太多，渲染主线程也不是太繁忙。不过浏览器是向前不停进化的，其进化路线体现在架构的调整、功能的增加以及更加精细的优化策略等方面，这些变化让渲染进程所需要处理的任务变多了，对应的渲染进程的主线程也变得越拥挤。下图所展示的仅仅是部分运行在主线程上的任务，你可以参考下：



任务和消息队列

你可以试想一下，在基于这种单消息队列的架构下，如果用户发出一个点击事件或者缩放页面的事件，而在此时，该任务前面可能还有很多不太重要的任务在排队等待着被执行，诸如V8的垃圾回收、DOM定时器等任务，如果执行这些任务需要花费的时间过久的话，那么就会让用户产生卡顿的感觉。你可以参看下图：



队头阻塞问题

因此，**在单消息队列架构下，存在着低优先级任务会阻塞高优先级任务的情况**，比如在一些性能不高的手机上，有时候滚动页面需要等待一秒以上。这像极了我们在介绍HTTP协议时所谈论的队头阻塞问题，那么我们也把这个问题称为消息队列的队头阻塞问题吧。

Chromium是如何解决队头阻塞问题的？

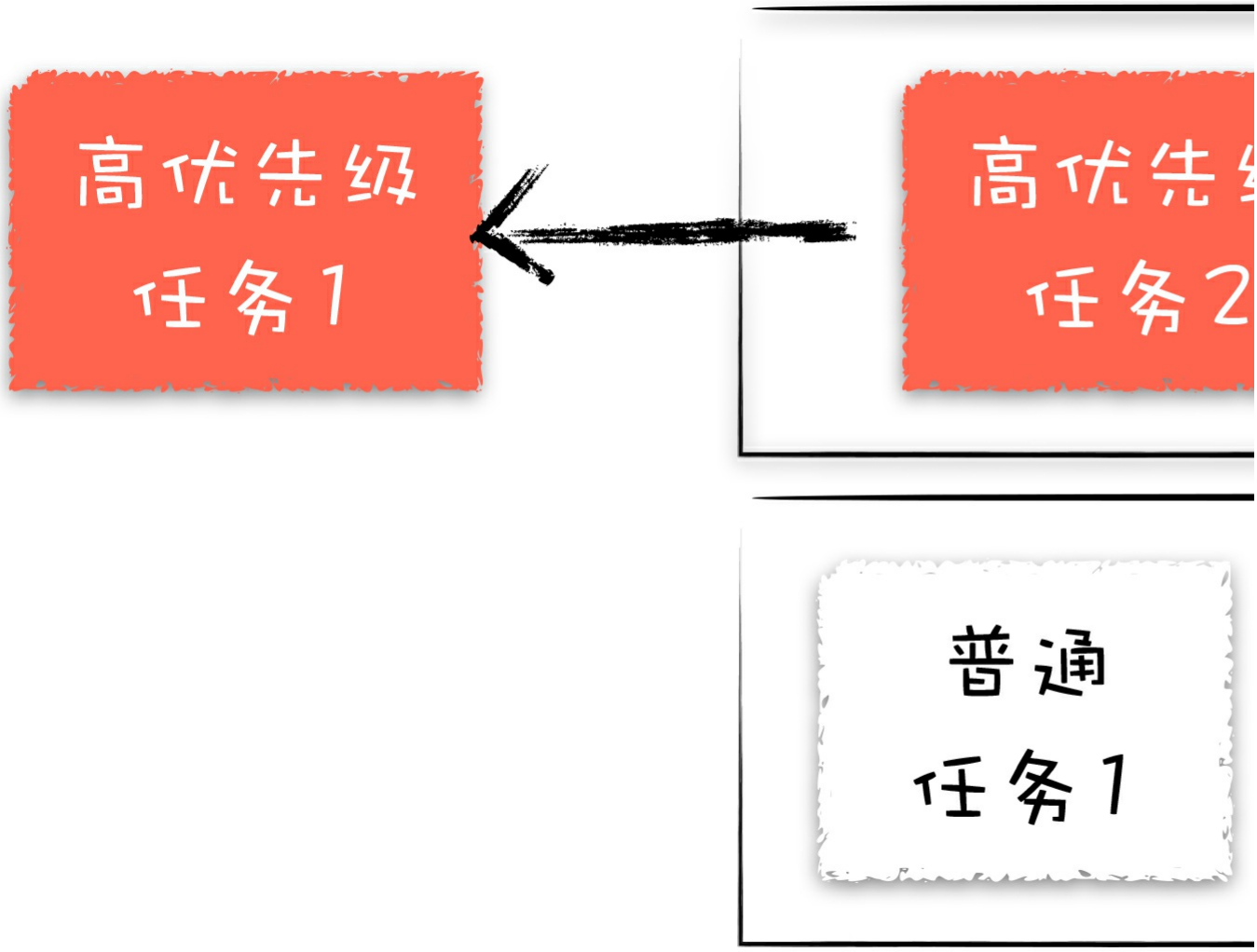
为了解决由于单消息队列而造成的队头阻塞问题，Chromium团队从2013年到现在，花了大量的精力在持续重构底层消息机制。在接下来的篇幅里，我会按照Chromium团队的重构消息系统的思路，来带你分析下他们是如何解决掉队头阻塞问题的。

1. 第一次迭代：引入一个高优先级队列

首先在最理想的情况下，我们希望能够快速跟踪高优先级任务，比如在交互阶段，下面几种任务都应该视为高优先级的任务：

- 通过鼠标触发的点击任务、滚动页面任务；
- 通过手势触发的页面缩放任务；
- 通过CSS、JavaScript等操作触发的动画特效等任务。

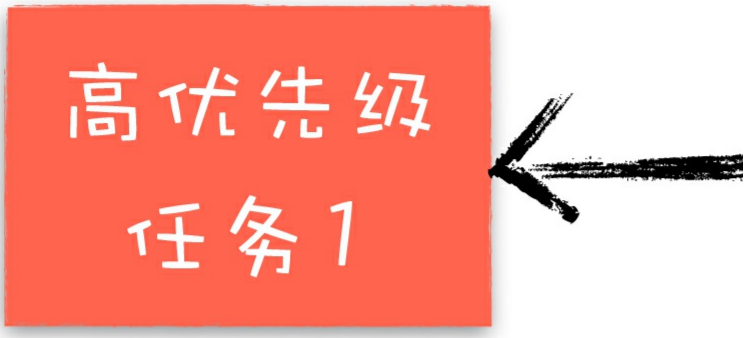
这些任务被触发后，用户想立即得到页面的反馈，所以我们需要让这些任务能够优先与其他的任务执行。要实现这种效果，我们可以增加一个高优级的消息队列，将高优先级的任务都添加到这个队列里面，然后优先执行该消息队列中的任务。最终效果如下图所示:



引入高优先级的消息队列

观察上图，我们使用了一个优先级高的消息队列和一个优先级低消息队列，渲染进程会将它认为是紧急的任务添加到高优先级队列中，不紧急的任务就添加到低优先级的队列中。然后我们再在渲染进程中引入一个**任务调度器**，负责从多个消息队列中选出合适的任务，通常实现的逻辑，先按照顺序从高优先级队列中取出任务，如果高优先级的队列为空，那么再按照顺序从低优级队列中取出任务。

我们还可以更进一步，将任务划分为多个不同的优先级，来实现更加细粒度的任务调度，比如可以划分为高优先级，普通优先级和低优先级，最终效果如下图所示：



增加多个不同优先级的消息队列

观察上图，我们实现了三个不同优先级的消息队列，然后可以使用任务调度器来统一调度这三个不同消息队列中的任务。

好了，现在我们引入了多个消息队列，结合任务调度器我们就可以灵活地调度任务了，这样我们就可以让高优先级的任务提前执行，采用这种方式似乎解决了消息队列的队头阻塞问题。

不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，甚至有可能出现还未处理输入事件，就合成了该事件要显示的图片。因此我们需要让一些相同类型的任务保持其相对执行顺序。

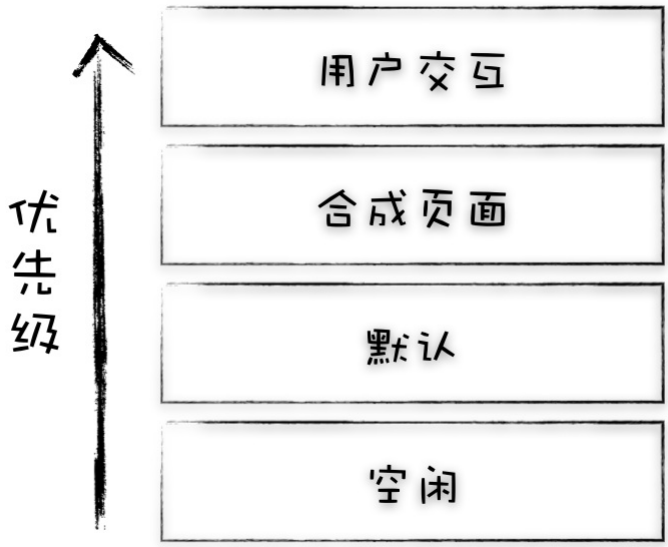
2. 第二次迭代：根据消息类型来实现消息队列

要解决上述问题，我们可以为不同类型的任务创建不同优先级的消息队列，比如：

- 可以创建输入事件的消息队列，用来存放输入事件。

- 可以创建合成任务的消息队列，用来存放合成事件。
- 可以创建默认消息队列，用来保存如资源加载的事件和定时器回调等事件。
- 还可以创建一个空闲消息队列，用来存放V8的垃圾自动垃圾回收这一类实时性不高的事件。

最终实现效果如下图所示：



根据消息类型实现不同优先级的消息队列

通过迭代，这种策略已经相当实用了，但是它依然存在着问题，那就是这几种消息队列的优先级都是固定的，任务调度器会按照这种固定好的静态的优先级来分别调度任务。那么静态优先级会带来什么问题呢？

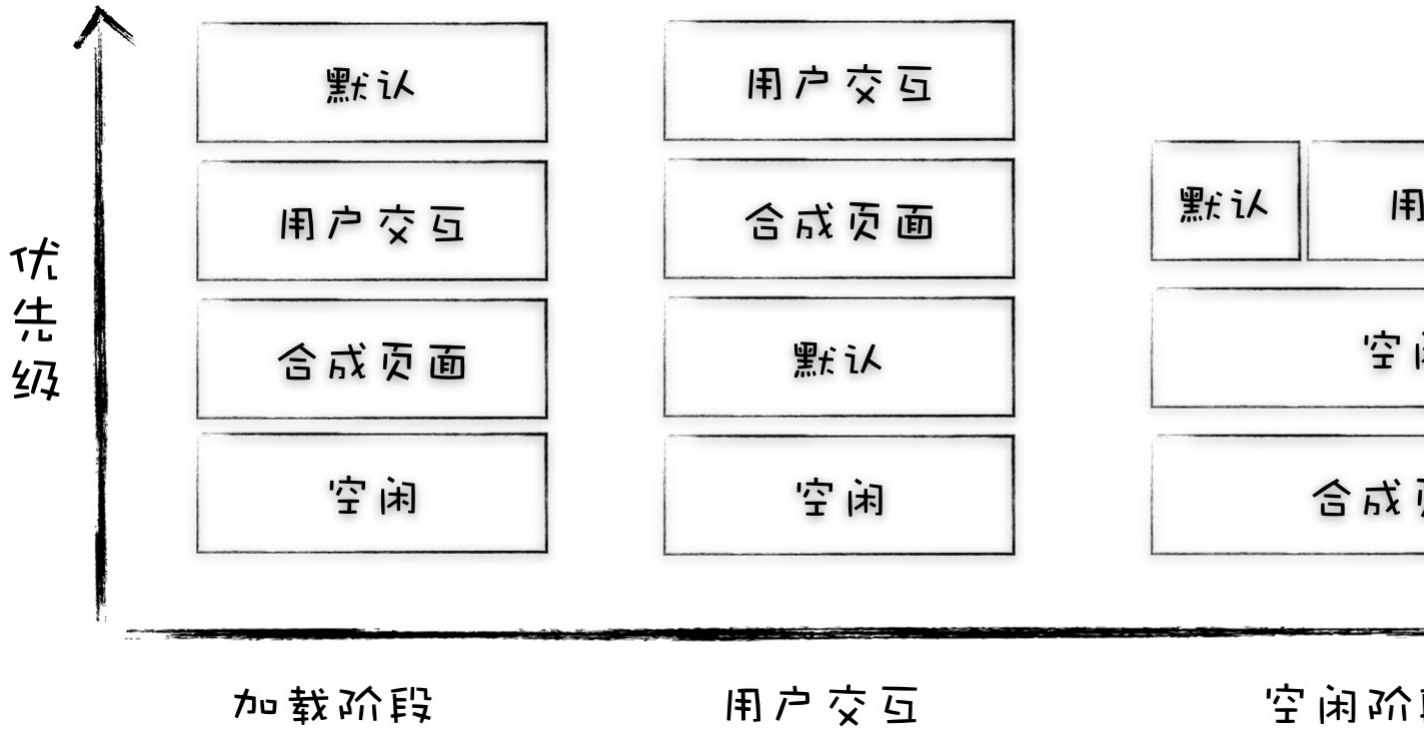
我们在《25 | 页面性能：如何系统地优化页面？》这节分析过页面的生存周期，页面大致的生存周期大体分为两个阶段，加载阶段和交互阶段。

虽然在交互阶段，采用上述这种静态优先级的策略没有什么太大问题的，但是在页面加载阶段，如果依然要优先执行用户输入事件和合成事件，那么页面的解析速度将会被拖慢。Chromium团队曾测试过这种情况，使用静态优先级策略，网页的加载速度会被拖慢14%。

3. 第三次迭代：动态调度策略

可以看出，我们所采用的优化策略像个跷跷板，虽然优化了高优先级任务，却拖慢低优先级任务，之所以会这样，是因为我们采取了静态的任务调度策略，对于各种不同的场景，这种静态策略就显得过于死板。

所以我们还得根据实际场景来继续平衡这个跷跷板，也就是说在不同的场景下，根据实际情况，动态调整消息队列的优先级。一图胜过千言，我们先看下图：



动态调度策略

这张图展示了Chromium在不同的场景下，是如何调整消息队列优先级的。通过这种动态调度策略，就可以满足不同场景的核心诉求了，同时这也是Chromium当前所采用的任务调度策略。

上图列出了三个不同的场景，分别是加载过程，合成过程以及正常状态。下面我们就结合这三种场景，来分析下Chromium为何做这种调整。

首先我们来看看**页面加载阶段**的场景，在这个阶段，用户的最高诉求是在尽可能短的时间内看到页面，至于交互和合成并不是这个阶段的核心诉求，因此我们需要调整策略，在加载阶段将页面解析，JavaScript脚本执行等任务调整为优先级最高的队列，降低交互合成这些队列的优先级。

页面加载完成之后就进入了**交互阶段**，在介绍Chromium是如何调整交互阶段的任务调度策略之前，我们还需要岔开一下，来回顾下页面的渲染过程。

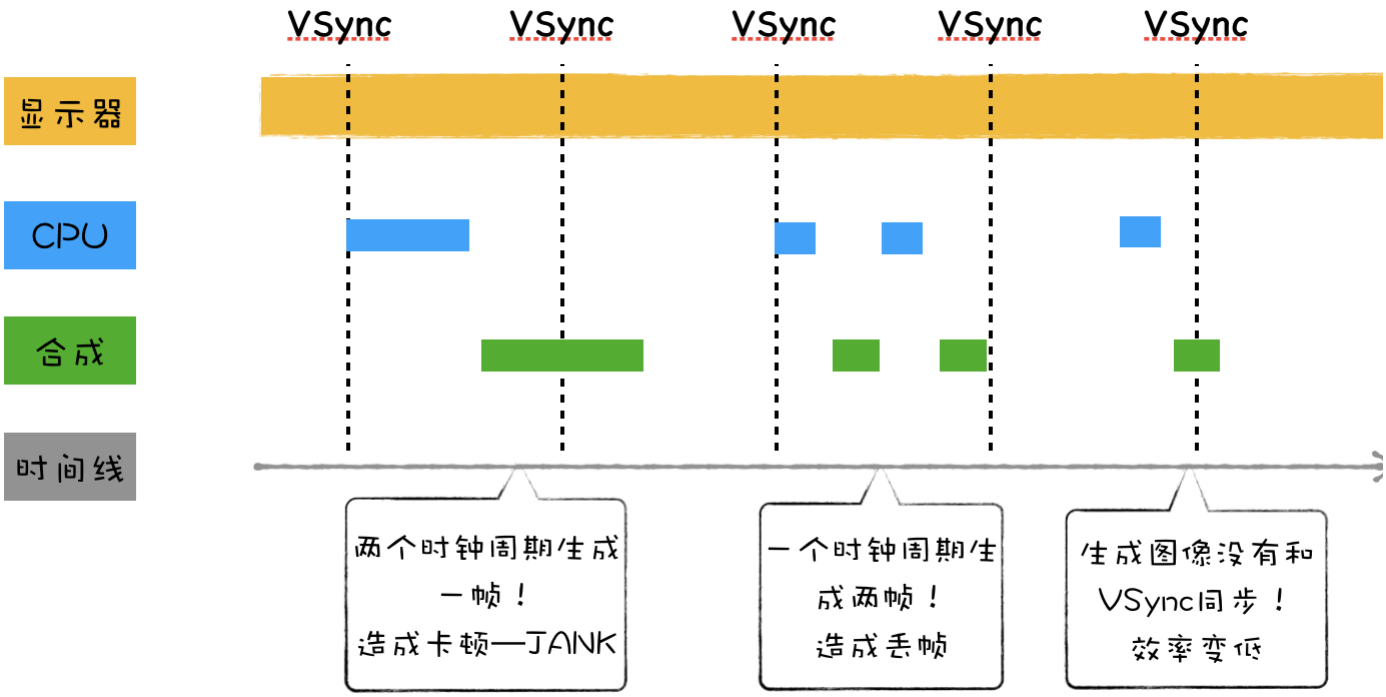
在《06 | 渲染流程（下）：HTML、CSS和JavaScript，是如何变成页面的？》和《24 | 分层和合成机制：为什么CSS动画比JavaScript高效？》这两节，我们分析了一个页面是如何渲染并显示出来的。

在显卡中有一块叫着**前缓冲区**的地方，这里存放着显示器要显示的图像，显示器会按照一定的频率来读取这块前缓冲区，并将前缓冲区中的图像显示在显示器上，不同的显示器读取的频率是不同的，通

常情况下是60HZ，也就是说显示器会每间隔1/60秒就读取一次前缓冲区。

如果浏览器要更新显示的图片，那么浏览器会将新生成的图片提交到显卡的**后缓冲区**中，提交完成之后，GPU会将**后缓冲区和前缓冲区互换位置**，也就是前缓冲区变成了后缓冲区，后缓冲区变成了前缓冲区，这就保证了显示器下次能读取到GPU中最新的图片。

这时候我们会发现，显示器从前缓冲区读取图片，和浏览器生成新的图像到后缓冲区的过程是不同步的，如下图所示：



VSync时钟周期和渲染引擎生成图片不同步问题

这种显示器读取图片和浏览器生成图片不同步，容易造成众多问题。

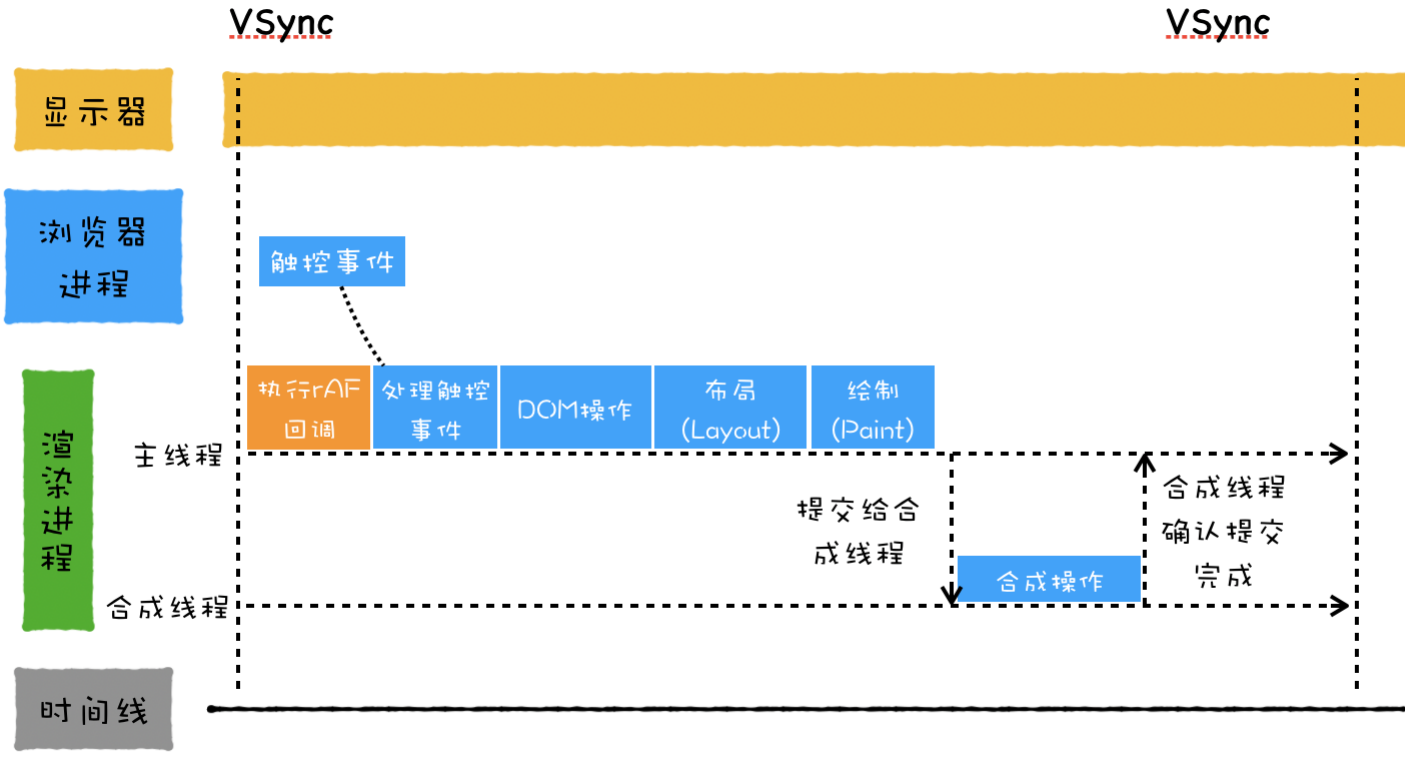
- 如果渲染进程生成的帧速比屏幕的刷新率慢，那么屏幕会在两帧中显示同一个画面，当这种断断续续的情况持续发生时，用户将会很明显地察觉到动画卡住了。
- 如果渲染进程生成的帧速率实际上比屏幕刷新率快，那么也会出现一些视觉上的问题，比如当帧速率在100fps而刷新率只有60Hz的时候，GPU所渲染的图像并非全都被显示出来，这就会造成丢帧现象。
- 就算屏幕的刷新频率和GPU更新图片的频率一样，由于它们是两个不同的系统，所以屏幕生成帧的周期和VSync的周期也是很难同步起来的。

所以VSync和系统的时钟不同步就会造成掉帧、卡顿、不连贯等问题。

为了解决这些问题，就需要将显示器的时钟同步周期和浏览器生成页面的周期绑定起来，Chromium也是这样实现，那么下面我们就来看看Chromium具体是怎么实现的？

当显示器将一帧画面绘制完成后，并在准备读取下一帧之前，显示器会发出一个垂直同步信号（vertical synchronization）给GPU，简称 VSync。这时候浏览器就会充分利用好VSync信号。

具体地讲，当GPU接收到VSync信号后，会将VSync信号同步给浏览器进程，浏览器进程再将其同步到对应的渲染进程，渲染进程接收到VSync信号之后，就可以准备绘制新的一帧了，具体流程你可以参考下图：



绑定VSync时钟同步周期和浏览器生成页面周期

上面其实是非常粗略的介绍，实际实现过程也是非常复杂的，如果感兴趣，你可以参考这篇文章。

好了，我们花了很大篇幅介绍了VSync和页面中的一帧是怎么显示出来，有了这些知识，我们就可以回到主线了，来分析下渲染进程是如何优化交互阶段页面的任务调度策略的？

从上图可以看出，当渲染进程接收到用户交互的任务后，接下来大概率是要进行绘制合成操作，因此我们可以设置，**当在执行用户交互的任务时，将合成任务的优先级调整到最高。**

接下来，处理完成DOM，计算好布局和绘制，就需要将信息提交给合成线程来合成最终图片了，然后合成线程进入工作状态。现在的场景是合成线程在工作了，**那么我们就可以把下个合成任务的优先级调整为最低，并将页面解析、定时器等任务优先级提升。**

在合成完成之后，合成线程会提交给渲染主线程提交完成合成的消息，如果当前合成操作执行的非常快，比如从用户发出消息到完成合成操作只花了8毫秒，因为VSync同步周期是16.66（1/60）毫秒，那么这个VSync时钟周期内就不需要再次生成新的页面了。那么从合成结束到下个VSync周期内，就进入了一个空闲时间阶段，那么就可以在这段空闲时间内执行一些不那么紧急的任务，比如V8的垃圾回收，或者通过window.requestIdleCallback()设置的回调任务等，都会在这段空闲时间内执行。

4. 第四次迭代：任务饿死

好了，以上方案看上去似乎非常完美了，不过依然存在一个问题，那就是在某个状态下，一直有新的高优先级的任务加入到队列中，这样就会导致其他低优先级的任务得不到执行，这称为任务饿死。

Chromium为了解决任务饿死的问题，给每个队列设置了执行权重，也就是如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样就缓解了任务饿死的情况。

总结

好了，本节的内容就介绍到这里，下面我来总结下本文的主要内容：

首先我们分析了基于单消息队列会引起队头阻塞的问题，为了解决队头阻塞问题，我们引入了多个不同优级的消息队列，并将紧急的任务添加到高优先级队列，不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，所以我们又迭代了第二个版本。

在第二个版本中，按照不同的任务类型来划分任务优先级，不过由于采用的静态优先级策略，对于其他一些场景，这种静态调度的策略并不是太适合，所以接下来，我们又迭代了第三版。

第三个版本，基于不同的场景来动态调整消息队列的优先级，到了这里已经非常完美了，不过依然存在着任务饿死的问题，为了解决任务饿死的问题，我们给每个队列一个权重，如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样我们就完成了Chromium的任务改造。

通过整个过程的分析，我们应该能理解，在开发一个项目时，不要试图去找最完美的方案，完美的方案往往是不存在的，我们需要根据实际的场景来寻找最适合我们的方案。

思考题

我们知道CSS动画是由渲染进程自动处理的，所以渲染进程会让CSS渲染每帧动画的过程与VSync的时钟保持一致,这样就能保证CSS动画的高效率执行。

但是JavaScript是由用户控制的，如果采用setTimeout来触发动画每帧的绘制，那么其绘制时机是很难和VSync时钟保持一致的，所以JavaScript中又引入了window.requestAnimationFrame，用来和VSync的时钟周期同步，那么我留给你的问题是：你知道requestAnimationFrame回调函数的执行时机吗？

参考资料

下面是我参考的一些资料：

- Blink Scheduler
- Blink Scheduler PPT
- Chrome的消息类型
- Chrome消息优先级
- 无头浏览器

欢迎在留言区分享你的想法。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

你好，我是李兵。

我们都知道，要想利用JavaScript实现高性能的动画，那就得使用requestAnimationFrame这个API，我们简称rAF，那么为什么都推荐使用rAF而不是setTimeOut呢？

要解释清楚这个问题，就要从渲染进程的任务调度系统讲起，理解了渲染进程任务调度系统，你自然就明白了rAF和setTimeOut的区别。其次，如果你理解任务调度系统，那么你就能将渲染流水线和浏览器系统架构等知识串起来，理解了这些概念也有助于你理解Performance标签是如何工作的。

要想了解最新Chrome的任务调度系统是怎么工作的，我们得先来回顾下之前介绍的消息循环系统，我们知道了渲染进程内部的大多数任务都是在主线程上执行的，诸如JavaScript执行、DOM、CSS、计算布局、V8的垃圾回收等任务。要让这些任务能够在主线程上有条不紊地运行，就需要引入消息队列。

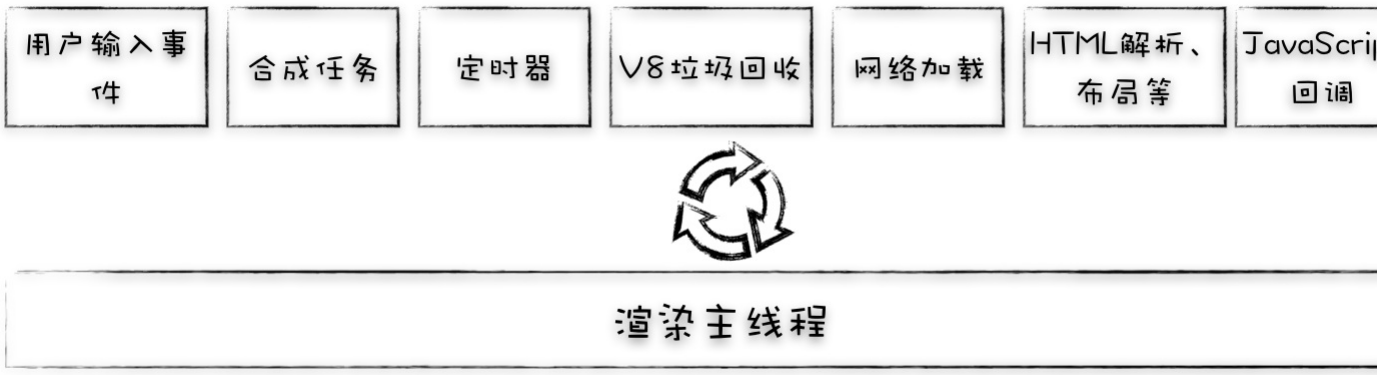
在前面的《16 | WebAPI：setTimeout是如何实现的？》这篇文章中，我们还介绍了，主线程维护了一个普通的消息队列和一个延迟消息队列，调度模块会按照规则依次取出这两个消息队列中的任务，并在主线程上执行。为了下文讲述方便，在这里我把普通的消息队列和延迟队列都当成一个消息队列。

新的任务都是被放进消息队列中去的，然后主线程再依次从消息队列中取出这些任务来顺序执行。这就是我们之前介绍的消息队列和事件循环系统。

单消息队列的队头阻塞问题

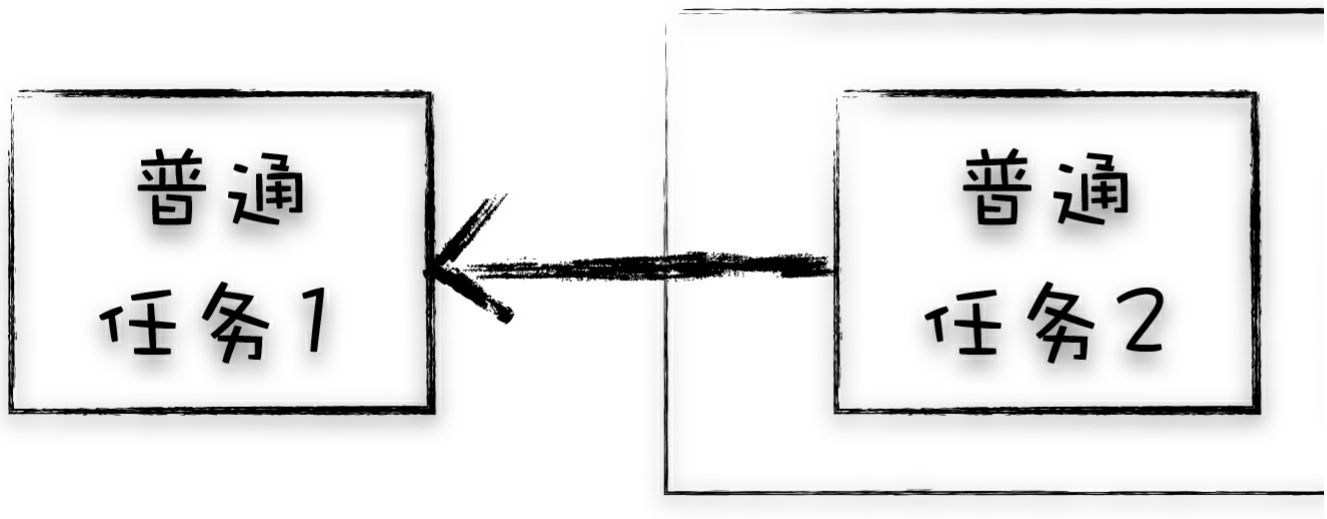
我们知道，渲染主线程会按照先进先出的顺序执行消息队列中的任务，具体地讲，当产生了新的任务，渲染进程会将其添加到消息队列尾部，在执行任务过程中，渲染进程会顺序地从消息队列头部取出任务并依次执行。

在最初，采用这种方式没有太大的问题，因为页面中的任务还不算太多，渲染主线程也不是太繁忙。不过浏览器是向前不停进化的，其进化路线体现在架构的调整、功能的增加以及更加精细的优化策略等方面，这些变化让渲染进程所需要处理的任务变多了，对应的渲染进程的主线程也变得越拥挤。下图所展示的仅仅是部分运行在主线程上的任务，你可以参考下：



任务和消息队列

你可以试想一下，在基于这种单消息队列的架构下，如果用户发出一个点击事件或者缩放页面的事件，而在此时，该任务前面可能还有很多不太重要的任务在排队等待着被执行，诸如V8的垃圾回收、DOM定时器等任务，如果执行这些任务需要花费的时间过久的话，那么就会让用户产生卡顿的感觉。你可以参看下图：



队头阻塞问题

因此，**在单消息队列架构下，存在着低优先级任务会阻塞高优先级任务的情况**，比如在一些性能不高的手机上，有时候滚动页面需要等待一秒以上。这像极了我们在介绍HTTP协议时所谈论的队头阻塞问题，那么我们也把这个问题称为消息队列的队头阻塞问题吧。

Chromium是如何解决队头阻塞问题的？

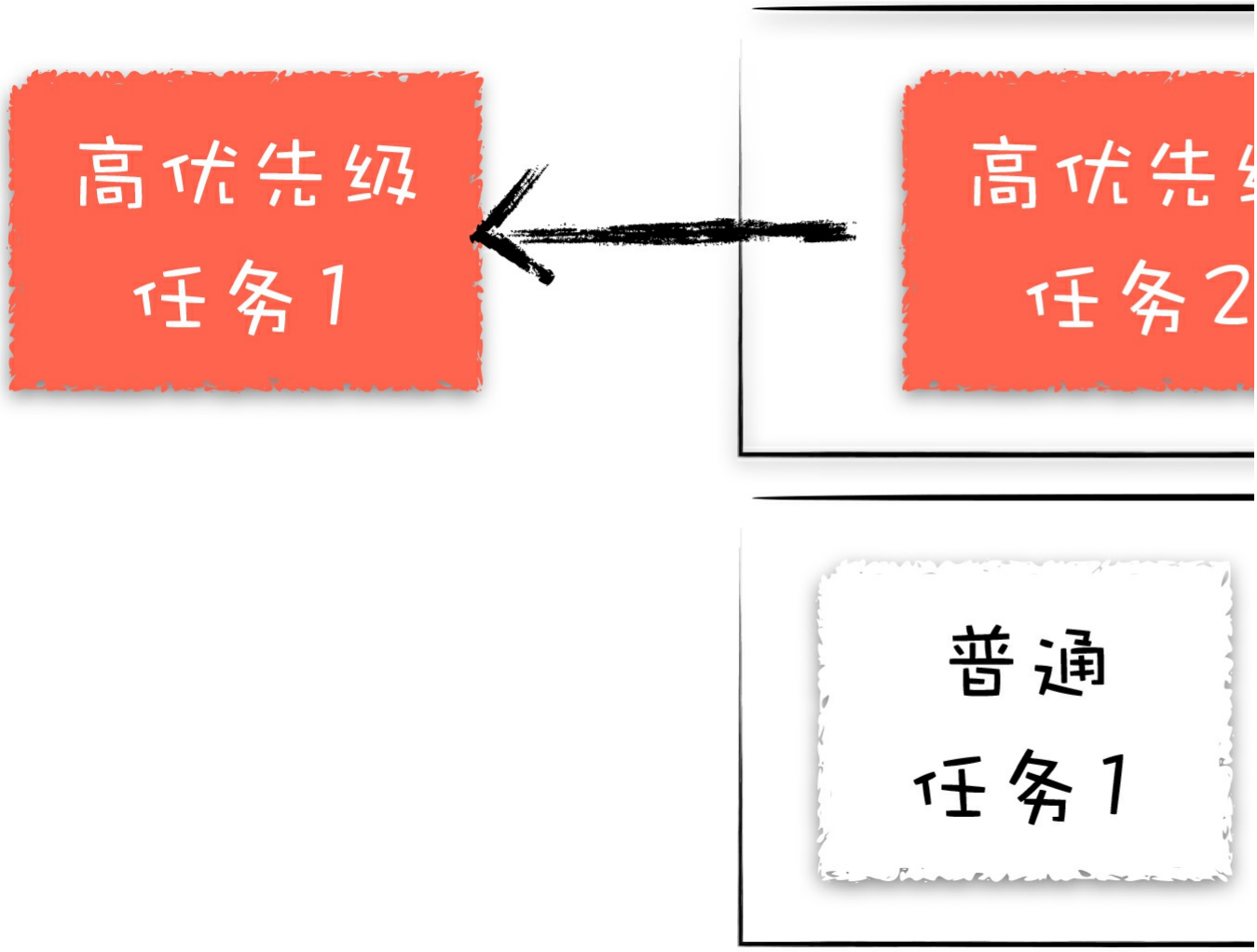
为了解决由于单消息队列而造成的队头阻塞问题，Chromium团队从2013年到现在，花了大量的精力在持续重构底层消息机制。在接下来的篇幅里，我会按照Chromium团队的重构消息系统的思路，来带你分析下他们是如何解决掉队头阻塞问题的。

1. 第一次迭代：引入一个高优先级队列

首先在最理想的情况下，我们希望能够快速跟踪高优先级任务，比如在交互阶段，下面几种任务都应该视为高优先级的任务：

- 通过鼠标触发的点击任务、滚动页面任务；
- 通过手势触发的页面缩放任务；
- 通过CSS、JavaScript等操作触发的动画特效等任务。

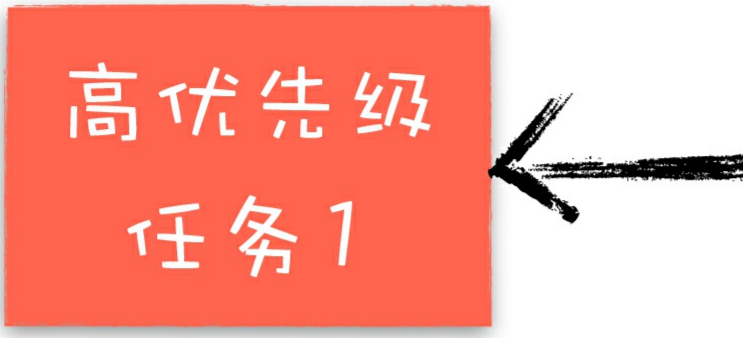
这些任务被触发后，用户想立即得到页面的反馈，所以我们需要让这些任务能够优先与其他的任务执行。要实现这种效果，我们可以增加一个高优级的消息队列，将高优先级的任务都添加到这个队列里面，然后优先执行该消息队列中的任务。最终效果如下图所示:



引入高优先级的消息队列

观察上图，我们使用了一个优先级高的消息队列和一个优先级低消息队列，渲染进程会将它认为是紧急的任务添加到高优先级队列中，不紧急的任务就添加到低优先级的队列中。然后我们再在渲染进程中引入一个**任务调度器**，负责从多个消息队列中选出合适的任务，通常实现的逻辑，先按照顺序从高优先级队列中取出任务，如果高优先级的队列为空，那么再按照顺序从低优级队列中取出任务。

我们还可以更进一步，将任务划分为多个不同的优先级，来实现更加细粒度的任务调度，比如可以划分为高优先级，普通优先级和低优先级，最终效果如下图所示：



增加多个不同优先级的消息队列

观察上图，我们实现了三个不同优先级的消息队列，然后可以使用任务调度器来统一调度这三个不同消息队列中的任务。

好了，现在我们引入了多个消息队列，结合任务调度器我们就可以灵活地调度任务了，这样我们就可以让高优先级的任务提前执行，采用这种方式似乎解决了消息队列的队头阻塞问题。

不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，甚至有可能出现还未处理输入事件，就合成了该事件要显示的图片。因此我们需要让一些相同类型的任务保持其相对执行顺序。

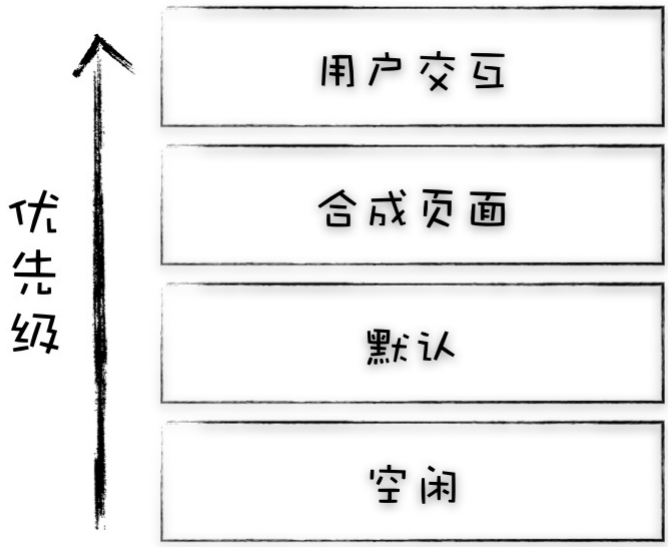
2. 第二次迭代：根据消息类型来实现消息队列

要解决上述问题，我们可以为不同类型的任务创建不同优先级的消息队列，比如：

- 可以创建输入事件的消息队列，用来存放输入事件。

- 可以创建合成任务的消息队列，用来存放合成事件。
- 可以创建默认消息队列，用来保存如资源加载的事件和定时器回调等事件。
- 还可以创建一个空闲消息队列，用来存放V8的垃圾自动垃圾回收这一类实时性不高的事件。

最终实现效果如下图所示：



根据消息类型实现不同优先级的消息队列

通过迭代，这种策略已经相当实用了，但是它依然存在着问题，那就是这几种消息队列的优先级都是固定的，任务调度器会按照这种固定好的静态的优先级来分别调度任务。那么静态优先级会带来什么问题呢？

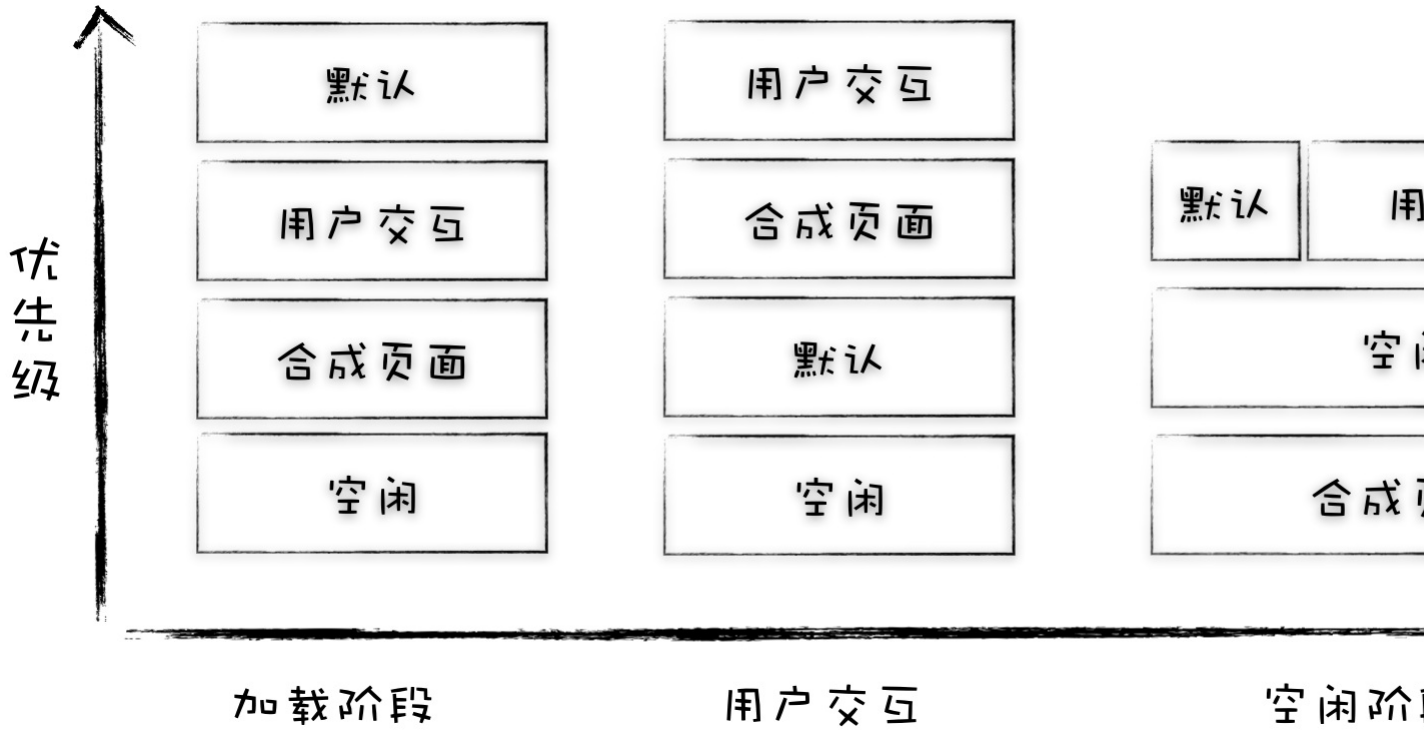
我们在《25 | 页面性能：如何系统地优化页面？》这节分析过页面的生存周期，页面大致的生存周期大体分为两个阶段，加载阶段和交互阶段。

虽然在交互阶段，采用上述这种静态优先级的策略没有什么太大问题的，但是在页面加载阶段，如果依然要优先执行用户输入事件和合成事件，那么页面的解析速度将会被拖慢。Chromium团队曾测试过这种情况，使用静态优先级策略，网页的加载速度会被拖慢14%。

3. 第三次迭代：动态调度策略

可以看出，我们所采用的优化策略像个跷跷板，虽然优化了高优先级任务，却拖慢低优先级任务，之所以会这样，是因为我们采取了静态的任务调度策略，对于各种不同的场景，这种静态策略就显得过于死板。

所以我们还得根据实际场景来继续平衡这个跷跷板，也就是说在不同的场景下，根据实际情况，动态调整消息队列的优先级。一图胜过千言，我们先看下图：



动态调度策略

这张图展示了Chromium在不同的场景下，是如何调整消息队列优先级的。通过这种动态调度策略，就可以满足不同场景的核心诉求了，同时这也是Chromium当前所采用的任务调度策略。

上图列出了三个不同的场景，分别是加载过程，合成过程以及正常状态。下面我们就结合这三种场景，来分析下Chromium为何做这种调整。

首先我们来看看**页面加载阶段**的场景，在这个阶段，用户的最高诉求是在尽可能短的时间内看到页面，至于交互和合成并不是这个阶段的核心诉求，因此我们需要调整策略，在加载阶段将页面解析，JavaScript脚本执行等任务调整为优先级最高的队列，降低交互合成这些队列的优先级。

页面加载完成之后就进入了**交互阶段**，在介绍Chromium是如何调整交互阶段的任务调度策略之前，我们还需要岔开一下，来回顾下页面的渲染过程。

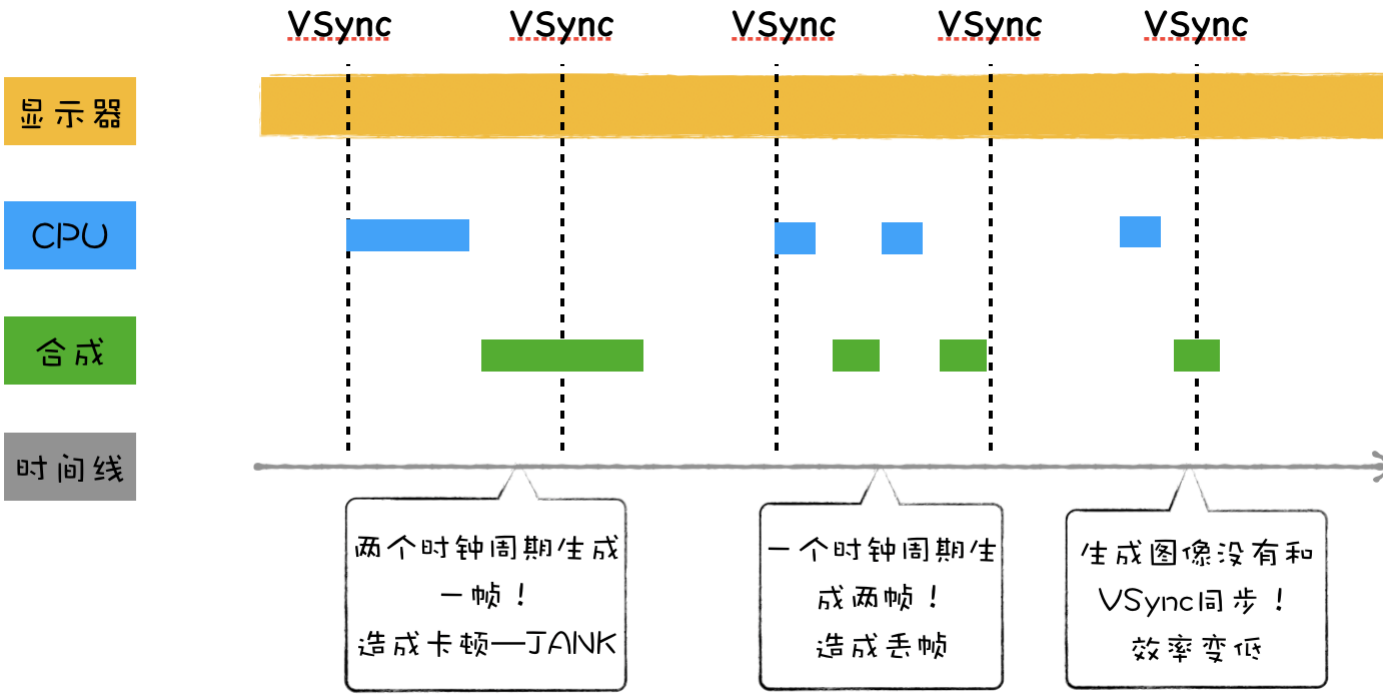
在《06 | 渲染流程（下）：HTML、CSS和JavaScript，是如何变成页面的？》和《24 | 分层和合成机制：为什么CSS动画比JavaScript高效？》这两节，我们分析了一个页面是如何渲染并显示出来的。

在显卡中有一块叫着**前缓冲区**的地方，这里存放着显示器要显示的图像，显示器会按照一定的频率来读取这块前缓冲区，并将前缓冲区中的图像显示在显示器上，不同的显示器读取的频率是不同的，通

常情况下是60HZ，也就是说显示器会每间隔1/60秒就读取一次前缓冲区。

如果浏览器要更新显示的图片，那么浏览器会将新生成的图片提交到显卡的**后缓冲区**中，提交完成之后，GPU会将**后缓冲区和前缓冲区互换位置**，也就是前缓冲区变成了后缓冲区，后缓冲区变成了前缓冲区，这就保证了显示器下次能读取到GPU中最新的图片。

这时候我们会发现，显示器从前缓冲区读取图片，和浏览器生成新的图像到后缓冲区的过程是不同步的，如下图所示：



VSync时钟周期和渲染引擎生成图片不同步问题

这种显示器读取图片和浏览器生成图片不同步，容易造成众多问题。

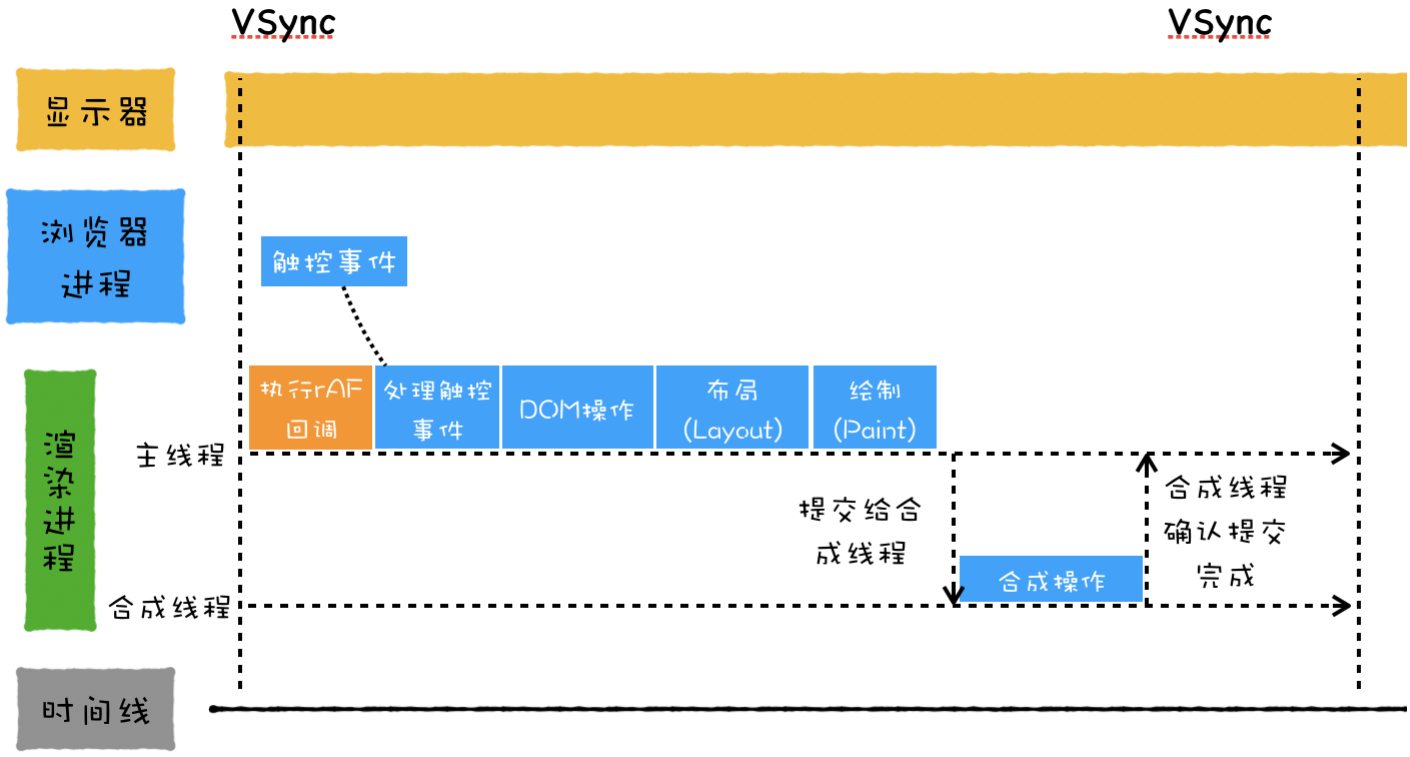
- 如果渲染进程生成的帧速比屏幕的刷新率慢，那么屏幕会在两帧中显示同一个画面，当这种断断续续的情况持续发生时，用户将会很明显地察觉到动画卡住了。
- 如果渲染进程生成的帧速率实际上比屏幕刷新率快，那么也会出现一些视觉上的问题，比如当帧速率在100fps而刷新率只有60Hz的时候，GPU所渲染的图像并非全都被显示出来，这就会造成丢帧现象。
- 就算屏幕的刷新频率和GPU更新图片的频率一样，由于它们是两个不同的系统，所以屏幕生成帧的周期和VSync的周期也是很难同步起来的。

所以VSync和系统的时钟不同步就会造成掉帧、卡顿、不连贯等问题。

为了解决这些问题，就需要将显示器的时钟同步周期和浏览器生成页面的周期绑定起来，Chromium也是这样实现，那么下面我们就来看看Chromium具体是怎么实现的？

当显示器将一帧画面绘制完成后，并在准备读取下一帧之前，显示器会发出一个垂直同步信号（vertical synchronization）给GPU，简称 VSync。这时候浏览器就会充分利用好VSync信号。

具体地讲，当GPU接收到VSync信号后，会将VSync信号同步给浏览器进程，浏览器进程再将其同步到对应的渲染进程，渲染进程接收到VSync信号之后，就可以准备绘制新的一帧了，具体流程你可以参考下图：



绑定VSync时钟同步周期和浏览器生成页面周期

上面其实是非常粗略的介绍，实际实现过程也是非常复杂的，如果感兴趣，你可以参考这篇文章。

好了，我们花了很大篇幅介绍了VSync和页面中的一帧是怎么显示出来，有了这些知识，我们就可以回到主线了，来分析下渲染进程是如何优化交互阶段页面的任务调度策略的？

从上图可以看出，当渲染进程接收到用户交互的任务后，接下来大概率是要进行绘制合成操作，因此我们可以设置，**当在执行用户交互的任务时，将合成任务的优先级调整到最高。**

接下来，处理完成DOM，计算好布局和绘制，就需要将信息提交给合成线程来合成最终图片了，然后合成线程进入工作状态。现在的场景是合成线程在工作了，**那么我们就可以把下个合成任务的优先级调整为最低，并将页面解析、定时器等任务优先级提升。**

在合成完成之后，合成线程会提交给渲染主线程提交完成合成的消息，如果当前合成操作执行的非常快，比如从用户发出消息到完成合成操作只花了8毫秒，因为VSync同步周期是16.66（1/60）毫秒，那么这个VSync时钟周期内就不需要再次生成新的页面了。那么从合成结束到下个VSync周期内，就进入了一个空闲时间阶段，那么就可以在这段空闲时间内执行一些不那么紧急的任务，比如V8的垃圾回收，或者通过window.requestIdleCallback()设置的回调任务等，都会在这段空闲时间内执行。

4. 第四次迭代：任务饿死

好了，以上方案看上去似乎非常完美了，不过依然存在一个问题，那就是在某个状态下，一直有新的高优先级的任务加入到队列中，这样就会导致其他低优先级的任务得不到执行，这称为任务饿死。

Chromium为了解决任务饿死的问题，给每个队列设置了执行权重，也就是如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样就缓解了任务饿死的情况。

总结

好了，本节的内容就介绍到这里，下面我来总结下本文的主要内容：

首先我们分析了基于单消息队列会引起队头阻塞的问题，为了解决队头阻塞问题，我们引入了多个不同优级的消息队列，并将紧急的任务添加到高优先级队列，不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，所以我们又迭代了第二个版本。

在第二个版本中，按照不同的任务类型来划分任务优先级，不过由于采用的静态优先级策略，对于其他一些场景，这种静态调度的策略并不是太适合，所以接下来，我们又迭代了第三版。

第三个版本，基于不同的场景来动态调整消息队列的优先级，到了这里已经非常完美了，不过依然存在着任务饿死的问题，为了解决任务饿死的问题，我们给每个队列一个权重，如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样我们就完成了Chromium的任务改造。

通过整个过程的分析，我们应该能理解，在开发一个项目时，不要试图去找最完美的方案，完美的方案往往是不存在的，我们需要根据实际的场景来寻找最适合我们的方案。

思考题

我们知道CSS动画是由渲染进程自动处理的，所以渲染进程会让CSS渲染每帧动画的过程与VSync的时钟保持一致,这样就能保证CSS动画的高效率执行。

但是JavaScript是由用户控制的，如果采用setTimeout来触发动画每帧的绘制，那么其绘制时机是很难和VSync时钟保持一致的，所以JavaScript中又引入了window.requestAnimationFrame，用来和VSync的时钟周期同步，那么我留给你的问题是：你知道requestAnimationFrame回调函数的执行时机吗？

参考资料

下面是我参考的一些资料：

- Blink Scheduler
- Blink Scheduler PPT
- Chrome的消息类型
- Chrome消息优先级
- 无头浏览器

欢迎在留言区分享你的想法。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

你好，我是李兵。

我们都知道，要想利用JavaScript实现高性能的动画，那就得使用requestAnimationFrame这个API，我们简称rAF，那么为什么都推荐使用rAF而不是setTimeOut呢？

要解释清楚这个问题，就要从渲染进程的任务调度系统讲起，理解了渲染进程任务调度系统，你自然就明白了rAF和setTimeOut的区别。其次，如果你理解任务调度系统，那么你就能将渲染流水线和浏览器系统架构等知识串起来，理解了这些概念也有助于你理解Performance标签是如何工作的。

要想了解最新Chrome的任务调度系统是怎么工作的，我们得先来回顾下之前介绍的消息循环系统，我们知道了渲染进程内部的大多数任务都是在主线程上执行的，诸如JavaScript执行、DOM、CSS、计算布局、V8的垃圾回收等任务。要让这些任务能够在主线程上有条不紊地运行，就需要引入消息队列。

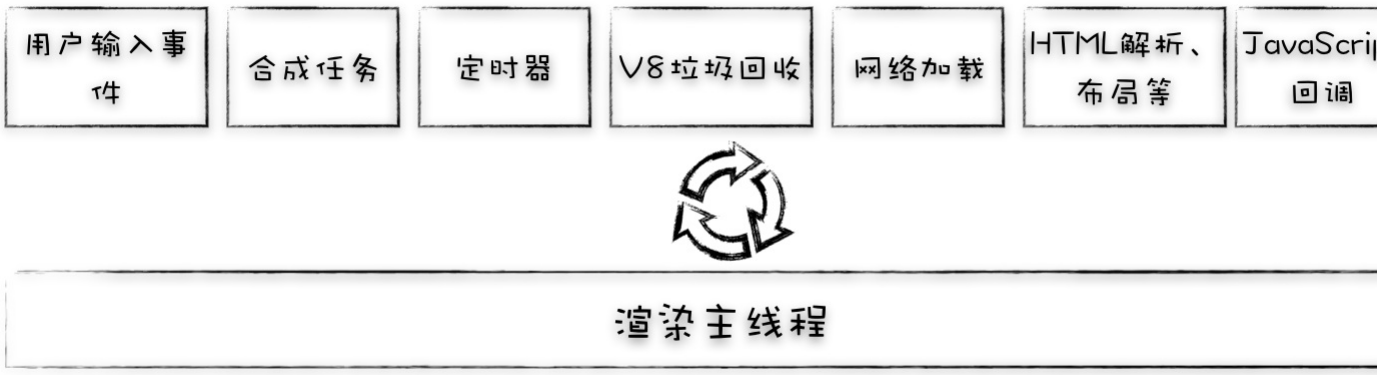
在前面的《16 | WebAPI：setTimeout是如何实现的？》这篇文章中，我们还介绍了，主线程维护了一个普通的消息队列和一个延迟消息队列，调度模块会按照规则依次取出这两个消息队列中的任务，并在主线程上执行。为了下文讲述方便，在这里我把普通的消息队列和延迟队列都当成一个消息队列。

新的任务都是被放进消息队列中去的，然后主线程再依次从消息队列中取出这些任务来顺序执行。这就是我们之前介绍的消息队列和事件循环系统。

单消息队列的队头阻塞问题

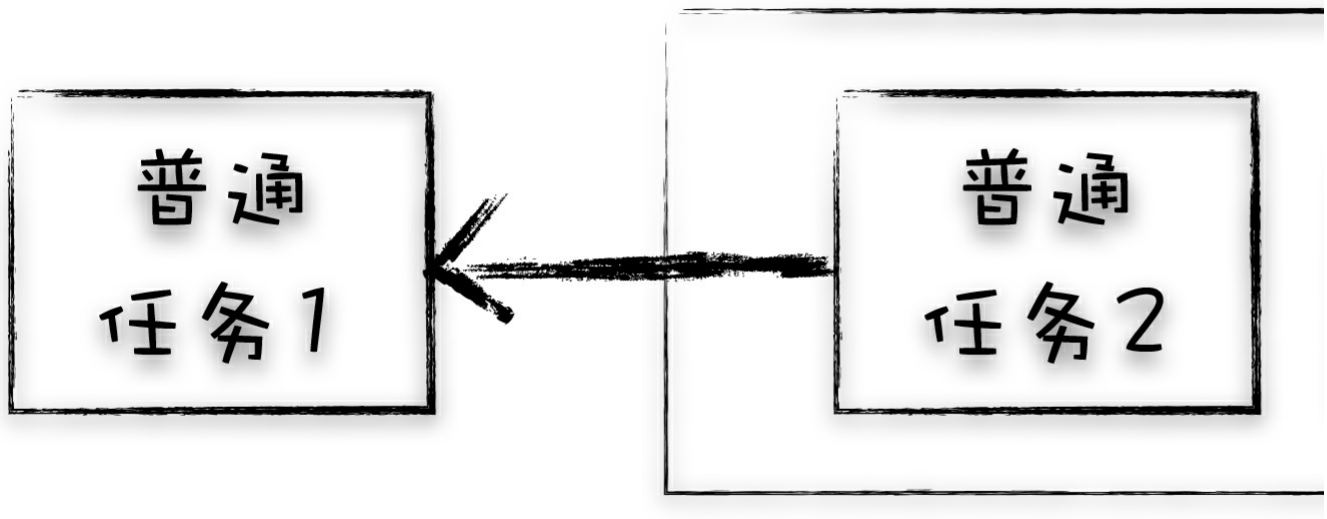
我们知道，渲染主线程会按照先进先出的顺序执行消息队列中的任务，具体地讲，当产生了新的任务，渲染进程会将其添加到消息队列尾部，在执行任务过程中，渲染进程会顺序地从消息队列头部取出任务并依次执行。

在最初，采用这种方式没有太大的问题，因为页面中的任务还不算太多，渲染主线程也不是太繁忙。不过浏览器是向前不停进化的，其进化路线体现在架构的调整、功能的增加以及更加精细的优化策略等方面，这些变化让渲染进程所需要处理的任务变多了，对应的渲染进程的主线程也变得越拥挤。下图所展示的仅仅是部分运行在主线程上的任务，你可以参考下：



任务和消息队列

你可以试想一下，在基于这种单消息队列的架构下，如果用户发出一个点击事件或者缩放页面的事件，而在此时，该任务前面可能还有很多不太重要的任务在排队等待着被执行，诸如V8的垃圾回收、DOM定时器等任务，如果执行这些任务需要花费的时间过久的话，那么就会让用户产生卡顿的感觉。你可以参看下图：



队头阻塞问题

因此，**在单消息队列架构下，存在着低优先级任务会阻塞高优先级任务的情况**，比如在一些性能不高的手机上，有时候滚动页面需要等待一秒以上。这像极了我们在介绍HTTP协议时所谈论的队头阻塞问题，那么我们也把这个问题称为消息队列的队头阻塞问题吧。

Chromium是如何解决队头阻塞问题的？

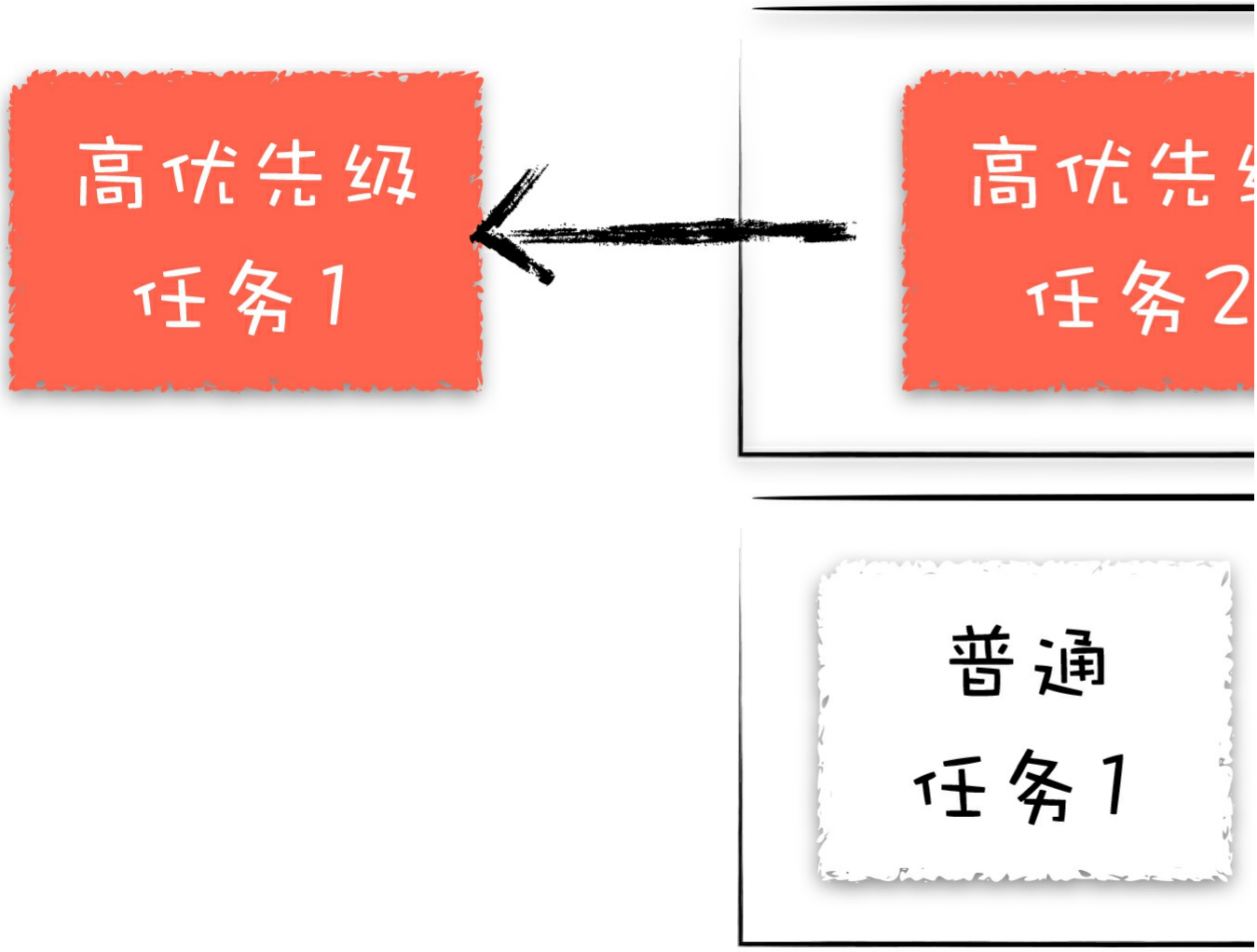
为了解决由于单消息队列而造成的队头阻塞问题，Chromium团队从2013年到现在，花了大量的精力在持续重构底层消息机制。在接下来的篇幅里，我会按照Chromium团队的重构消息系统的思路，来带你分析下他们是如何解决掉队头阻塞问题的。

1. 第一次迭代：引入一个高优先级队列

首先在最理想的情况下，我们希望能够快速跟踪高优先级任务，比如在交互阶段，下面几种任务都应该视为高优先级的任务：

- 通过鼠标触发的点击任务、滚动页面任务；
- 通过手势触发的页面缩放任务；
- 通过CSS、JavaScript等操作触发的动画特效等任务。

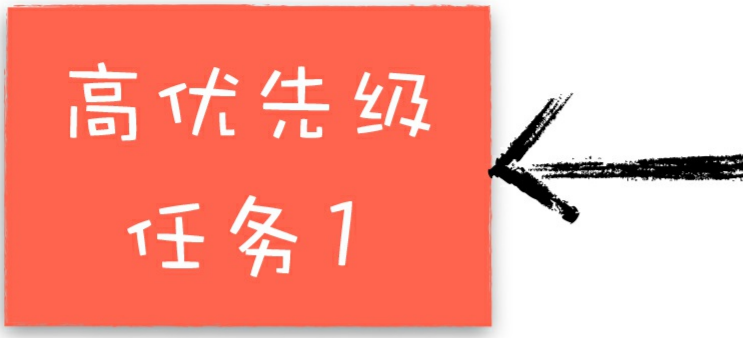
这些任务被触发后，用户想立即得到页面的反馈，所以我们需要让这些任务能够优先与其他的任务执行。要实现这种效果，我们可以增加一个高优级的消息队列，将高优先级的任务都添加到这个队列里面，然后优先执行该消息队列中的任务。最终效果如下图所示:



引入高优先级的消息队列

观察上图，我们使用了一个优先级高的消息队列和一个优先级低消息队列，渲染进程会将它认为是紧急的任务添加到高优先级队列中，不紧急的任务就添加到低优先级的队列中。然后我们再在渲染进程中引入一个**任务调度器**，负责从多个消息队列中选出合适的任务，通常实现的逻辑，先按照顺序从高优先级队列中取出任务，如果高优先级的队列为空，那么再按照顺序从低优级队列中取出任务。

我们还可以更进一步，将任务划分为多个不同的优先级，来实现更加细粒度的任务调度，比如可以划分为高优先级，普通优先级和低优先级，最终效果如下图所示：



增加多个不同优先级的消息队列

观察上图，我们实现了三个不同优先级的消息队列，然后可以使用任务调度器来统一调度这三个不同消息队列中的任务。

好了，现在我们引入了多个消息队列，结合任务调度器我们就可以灵活地调度任务了，这样我们就可以让高优先级的任务提前执行，采用这种方式似乎解决了消息队列的队头阻塞问题。

不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，甚至有可能出现还未处理输入事件，就合成了该事件要显示的图片。因此我们需要让一些相同类型的任务保持其相对执行顺序。

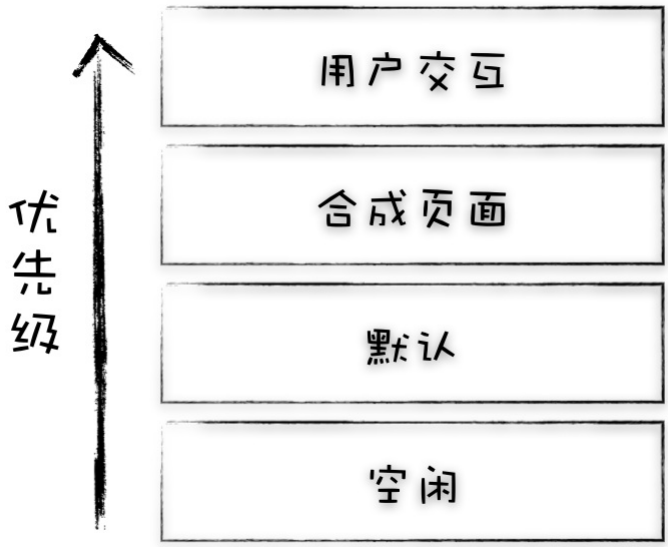
2. 第二次迭代：根据消息类型来实现消息队列

要解决上述问题，我们可以为不同类型的任务创建不同优先级的消息队列，比如：

- 可以创建输入事件的消息队列，用来存放输入事件。

- 可以创建合成任务的消息队列，用来存放合成事件。
- 可以创建默认消息队列，用来保存如资源加载的事件和定时器回调等事件。
- 还可以创建一个空闲消息队列，用来存放V8的垃圾自动垃圾回收这一类实时性不高的事件。

最终实现效果如下图所示：



根据消息类型实现不同优先级的消息队列

通过迭代，这种策略已经相当实用了，但是它依然存在着问题，那就是这几种消息队列的优先级都是固定的，任务调度器会按照这种固定好的静态的优先级来分别调度任务。那么静态优先级会带来什么问题呢？

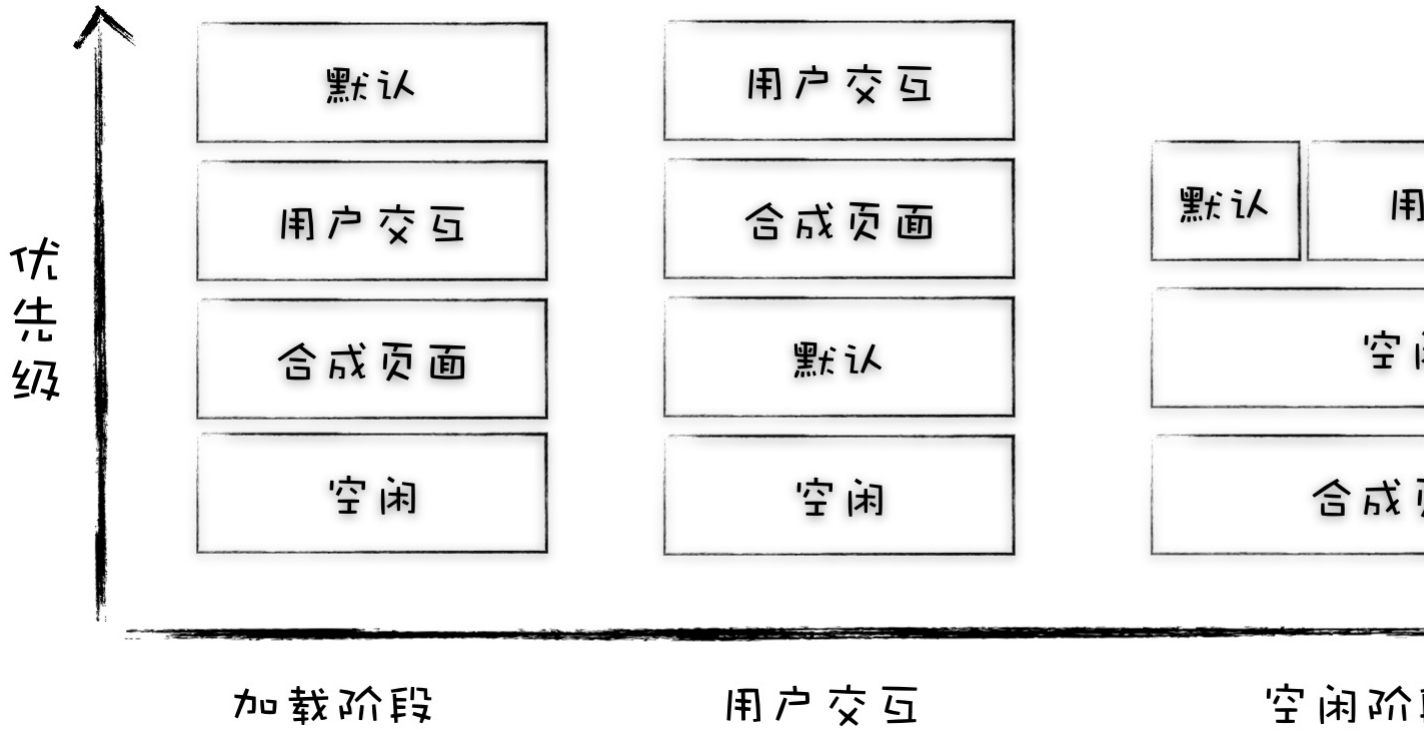
我们在《25 | 页面性能：如何系统地优化页面？》这节分析过页面的生存周期，页面大致的生存周期大体分为两个阶段，加载阶段和交互阶段。

虽然在交互阶段，采用上述这种静态优先级的策略没有什么太大问题的，但是在页面加载阶段，如果依然要优先执行用户输入事件和合成事件，那么页面的解析速度将会被拖慢。Chromium团队曾测试过这种情况，使用静态优先级策略，网页的加载速度会被拖慢14%。

3. 第三次迭代：动态调度策略

可以看出，我们所采用的优化策略像个跷跷板，虽然优化了高优先级任务，却拖慢低优先级任务，之所以会这样，是因为我们采取了静态的任务调度策略，对于各种不同的场景，这种静态策略就显得过于死板。

所以我们还得根据实际场景来继续平衡这个跷跷板，也就是说在不同的场景下，根据实际情况，动态调整消息队列的优先级。一图胜过千言，我们先看下图：



动态调度策略

这张图展示了Chromium在不同的场景下，是如何调整消息队列优先级的。通过这种动态调度策略，就可以满足不同场景的核心诉求了，同时这也是Chromium当前所采用的任务调度策略。

上图列出了三个不同的场景，分别是加载过程，合成过程以及正常状态。下面我们就结合这三种场景，来分析下Chromium为何做这种调整。

首先我们来看看**页面加载阶段**的场景，在这个阶段，用户的最高诉求是在尽可能短的时间内看到页面，至于交互和合成并不是这个阶段的核心诉求，因此我们需要调整策略，在加载阶段将页面解析，JavaScript脚本执行等任务调整为优先级最高的队列，降低交互合成这些队列的优先级。

页面加载完成之后就进入了**交互阶段**，在介绍Chromium是如何调整交互阶段的任务调度策略之前，我们还需要岔开一下，来回顾下页面的渲染过程。

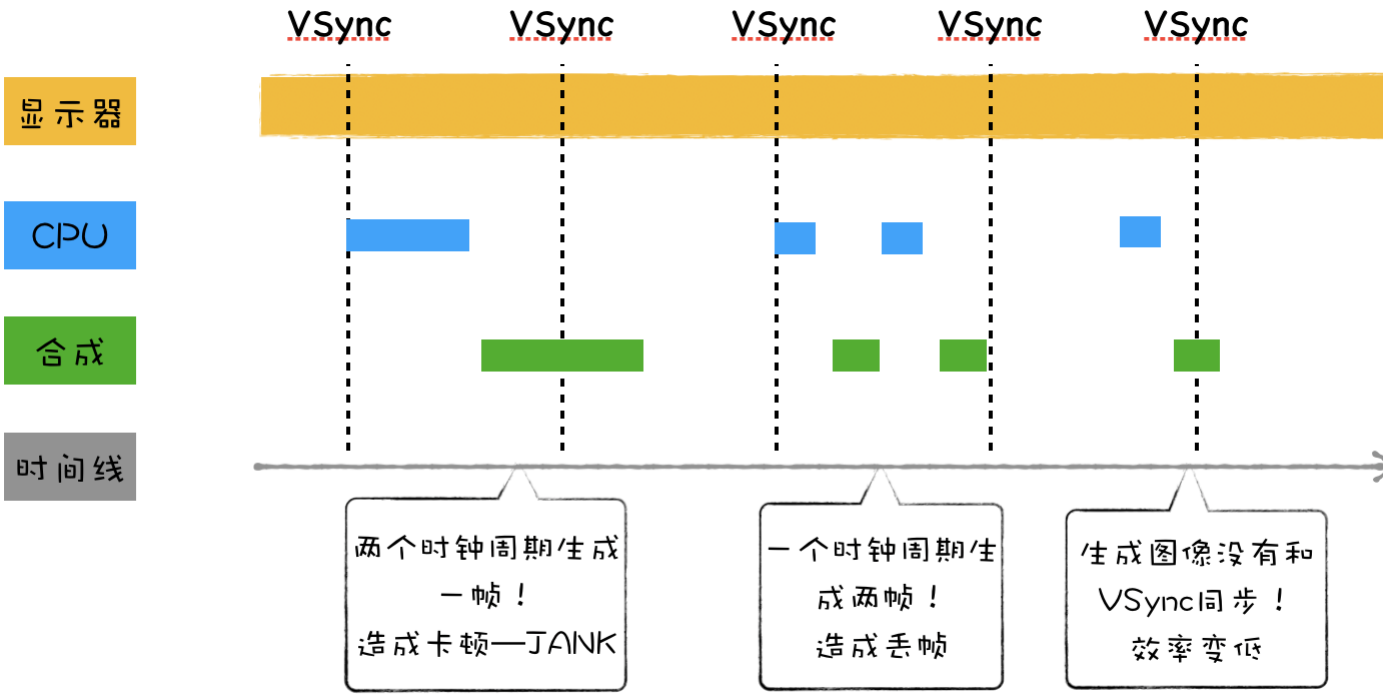
在《06 | 渲染流程（下）：HTML、CSS和JavaScript，是如何变成页面的？》和《24 | 分层和合成机制：为什么CSS动画比JavaScript高效？》这两节，我们分析了一个页面是如何渲染并显示出来的。

在显卡中有一块叫着**前缓冲区**的地方，这里存放着显示器要显示的图像，显示器会按照一定的频率来读取这块前缓冲区，并将前缓冲区中的图像显示在显示器上，不同的显示器读取的频率是不同的，通

常情况下是60HZ，也就是说显示器会每间隔1/60秒就读取一次前缓冲区。

如果浏览器要更新显示的图片，那么浏览器会将新生成的图片提交到显卡的**后缓冲区**中，提交完成之后，GPU会将**后缓冲区和前缓冲区互换位置**，也就是前缓冲区变成了后缓冲区，后缓冲区变成了前缓冲区，这就保证了显示器下次能读取到GPU中最新的图片。

这时候我们会发现，显示器从前缓冲区读取图片，和浏览器生成新的图像到后缓冲区的过程是不同步的，如下图所示：



VSync时钟周期和渲染引擎生成图片不同步问题

这种显示器读取图片和浏览器生成图片不同步，容易造成众多问题。

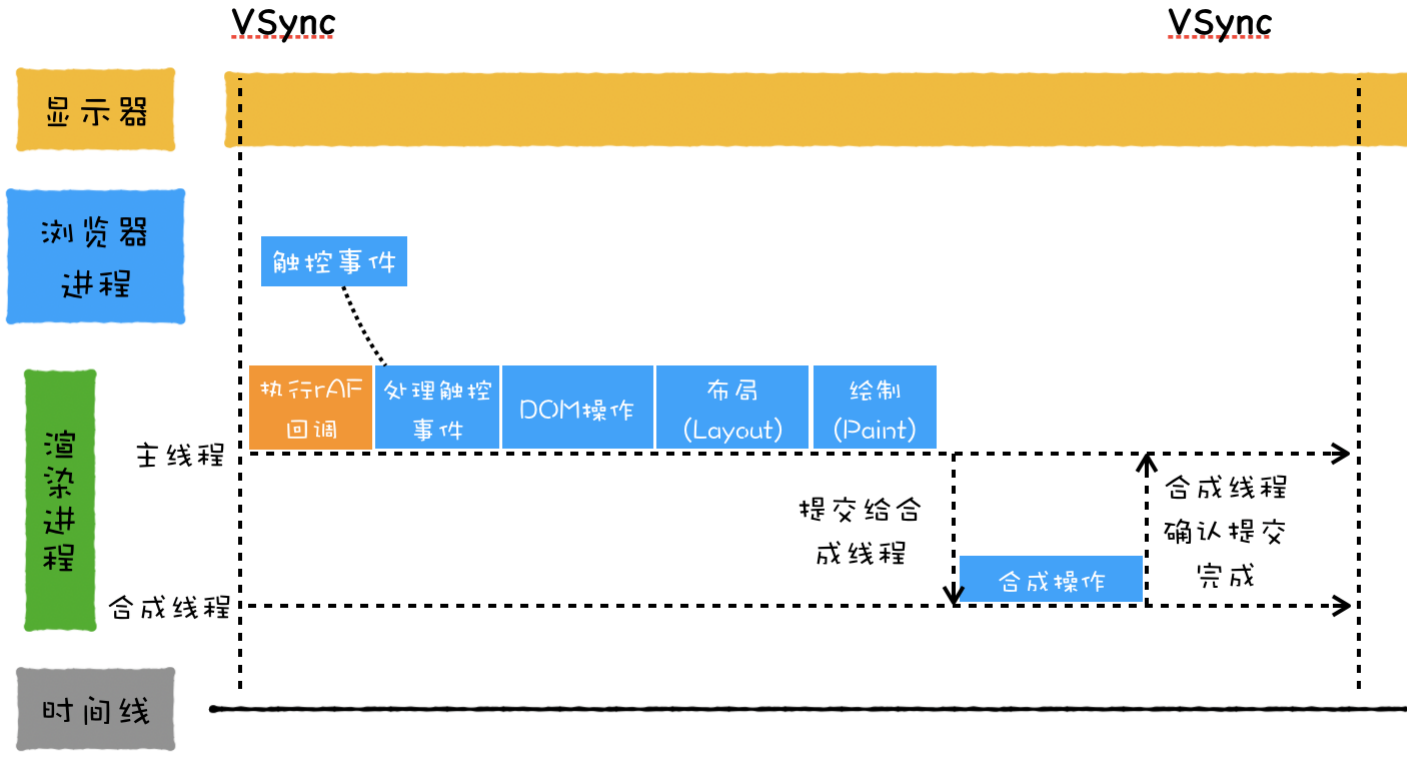
- 如果渲染进程生成的帧速比屏幕的刷新率慢，那么屏幕会在两帧中显示同一个画面，当这种断断续续的情况持续发生时，用户将会很明显地察觉到动画卡住了。
- 如果渲染进程生成的帧速率实际上比屏幕刷新率快，那么也会出现一些视觉上的问题，比如当帧速率在100fps而刷新率只有60Hz的时候，GPU所渲染的图像并非全都被显示出来，这就会造成丢帧现象。
- 就算屏幕的刷新频率和GPU更新图片的频率一样，由于它们是两个不同的系统，所以屏幕生成帧的周期和VSync的周期也是很难同步起来的。

所以VSync和系统的时钟不同步就会造成掉帧、卡顿、不连贯等问题。

为了解决这些问题，就需要将显示器的时钟同步周期和浏览器生成页面的周期绑定起来，Chromium也是这样实现，那么下面我们就来看看Chromium具体是怎么实现的？

当显示器将一帧画面绘制完成后，并在准备读取下一帧之前，显示器会发出一个垂直同步信号（vertical synchronization）给GPU，简称 VSync。这时候浏览器就会充分利用好VSync信号。

具体地讲，当GPU接收到VSync信号后，会将VSync信号同步给浏览器进程，浏览器进程再将其同步到对应的渲染进程，渲染进程接收到VSync信号之后，就可以准备绘制新的一帧了，具体流程你可以参考下图：



绑定VSync时钟同步周期和浏览器生成页面周期

上面其实是非常粗略的介绍，实际实现过程也是非常复杂的，如果感兴趣，你可以参考这篇文章。

好了，我们花了很大篇幅介绍了VSync和页面中的一帧是怎么显示出来，有了这些知识，我们就可以回到主线了，来分析下渲染进程是如何优化交互阶段页面的任务调度策略的？

从上图可以看出，当渲染进程接收到用户交互的任务后，接下来大概率是要进行绘制合成操作，因此我们可以设置，**当在执行用户交互的任务时，将合成任务的优先级调整到最高。**

接下来，处理完成DOM，计算好布局和绘制，就需要将信息提交给合成线程来合成最终图片了，然后合成线程进入工作状态。现在的场景是合成线程在工作了，**那么我们就可以把下个合成任务的优先级调整为最低，并将页面解析、定时器等任务优先级提升。**

在合成完成之后，合成线程会提交给渲染主线程提交完成合成的消息，如果当前合成操作执行的非常快，比如从用户发出消息到完成合成操作只花了8毫秒，因为VSync同步周期是16.66（1/60）毫秒，那么这个VSync时钟周期内就不需要再次生成新的页面了。那么从合成结束到下个VSync周期内，就进入了一个空闲时间阶段，那么就可以在这段空闲时间内执行一些不那么紧急的任务，比如V8的垃圾回收，或者通过window.requestIdleCallback()设置的回调任务等，都会在这段空闲时间内执行。

4. 第四次迭代：任务饿死

好了，以上方案看上去似乎非常完美了，不过依然存在一个问题，那就是在某个状态下，一直有新的高优先级的任务加入到队列中，这样就会导致其他低优先级的任务得不到执行，这称为任务饿死。

Chromium为了解决任务饿死的问题，给每个队列设置了执行权重，也就是如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样就缓解了任务饿死的情况。

总结

好了，本节的内容就介绍到这里，下面我来总结下本文的主要内容：

首先我们分析了基于单消息队列会引起队头阻塞的问题，为了解决队头阻塞问题，我们引入了多个不同优级的消息队列，并将紧急的任务添加到高优先级队列，不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，所以我们又迭代了第二个版本。

在第二个版本中，按照不同的任务类型来划分任务优先级，不过由于采用的静态优先级策略，对于其他一些场景，这种静态调度的策略并不是太适合，所以接下来，我们又迭代了第三版。

第三个版本，基于不同的场景来动态调整消息队列的优先级，到了这里已经非常完美了，不过依然存在着任务饿死的问题，为了解决任务饿死的问题，我们给每个队列一个权重，如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样我们就完成了Chromium的任务改造。

通过整个过程的分析，我们应该能理解，在开发一个项目时，不要试图去找最完美的方案，完美的方案往往是不存在的，我们需要根据实际的场景来寻找最适合我们的方案。

思考题

我们知道CSS动画是由渲染进程自动处理的，所以渲染进程会让CSS渲染每帧动画的过程与VSync的时钟保持一致,这样就能保证CSS动画的高效率执行。

但是JavaScript是由用户控制的，如果采用setTimeout来触发动画每帧的绘制，那么其绘制时机是很难和VSync时钟保持一致的，所以JavaScript中又引入了window.requestAnimationFrame，用来和VSync的时钟周期同步，那么我留给你的问题是：你知道requestAnimationFrame回调函数的执行时机吗？

参考资料

下面是我参考的一些资料：

- Blink Scheduler
- Blink Scheduler PPT
- Chrome的消息类型
- Chrome消息优先级
- 无头浏览器

欢迎在留言区分享你的想法。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

你好，我是李兵。

我们都知道，要想利用JavaScript实现高性能的动画，那就得使用requestAnimationFrame这个API，我们简称rAF，那么为什么都推荐使用rAF而不是setTimeOut呢？

要解释清楚这个问题，就要从渲染进程的任务调度系统讲起，理解了渲染进程任务调度系统，你自然就明白了rAF和setTimeOut的区别。其次，如果你理解任务调度系统，那么你就能将渲染流水线和浏览器系统架构等知识串起来，理解了这些概念也有助于你理解Performance标签是如何工作的。

要想了解最新Chrome的任务调度系统是怎么工作的，我们得先来回顾下之前介绍的消息循环系统，我们知道了渲染进程内部的大多数任务都是在主线程上执行的，诸如JavaScript执行、DOM、CSS、计算布局、V8的垃圾回收等任务。要让这些任务能够在主线程上有条不紊地运行，就需要引入消息队列。

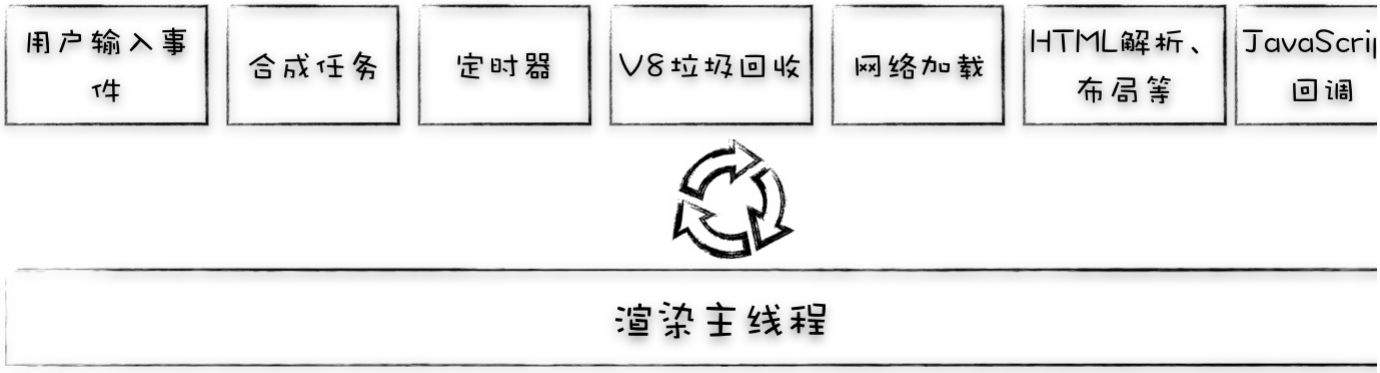
在前面的《16 | WebAPI：setTimeout是如何实现的？》这篇文章中，我们还介绍了，主线程维护了一个普通的消息队列和一个延迟消息队列，调度模块会按照规则依次取出这两个消息队列中的任务，并在主线程上执行。为了下文讲述方便，在这里我把普通的消息队列和延迟队列都当成一个消息队列。

新的任务都是被放进消息队列中去的，然后主线程再依次从消息队列中取出这些任务来顺序执行。这就是我们之前介绍的消息队列和事件循环系统。

单消息队列的队头阻塞问题

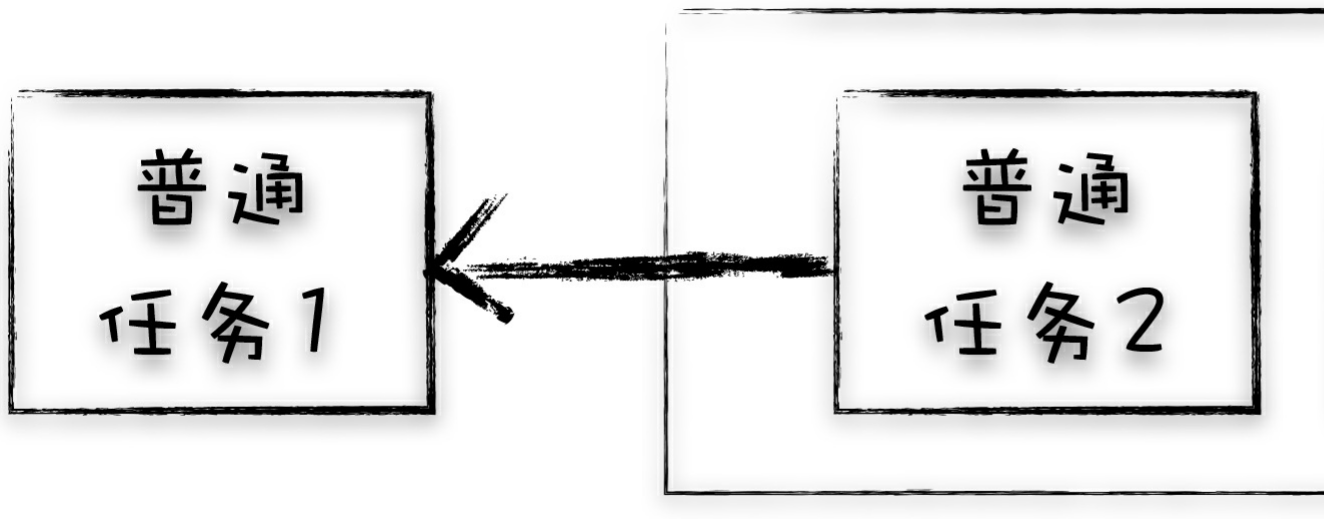
我们知道，渲染主线程会按照先进先出的顺序执行消息队列中的任务，具体地讲，当产生了新的任务，渲染进程会将其添加到消息队列尾部，在执行任务过程中，渲染进程会顺序地从消息队列头部取出任务并依次执行。

在最初，采用这种方式没有太大的问题，因为页面中的任务还不算太多，渲染主线程也不是太繁忙。不过浏览器是向前不停进化的，其进化路线体现在架构的调整、功能的增加以及更加精细的优化策略等方面，这些变化让渲染进程所需要处理的任务变多了，对应的渲染进程的主线程也变得越拥挤。下图所展示的仅仅是部分运行在主线程上的任务，你可以参考下：



任务和消息队列

你可以试想一下，在基于这种单消息队列的架构下，如果用户发出一个点击事件或者缩放页面的事件，而在此时，该任务前面可能还有很多不太重要的任务在排队等待着被执行，诸如V8的垃圾回收、DOM定时器等任务，如果执行这些任务需要花费的时间过久的话，那么就会让用户产生卡顿的感觉。你可以参看下图：



队头阻塞问题

因此，**在单消息队列架构下，存在着低优先级任务会阻塞高优先级任务的情况**，比如在一些性能不高的手机上，有时候滚动页面需要等待一秒以上。这像极了我们在介绍HTTP协议时所谈论的队头阻塞问题，那么我们也把这个问题称为消息队列的队头阻塞问题吧。

Chromium是如何解决队头阻塞问题的？

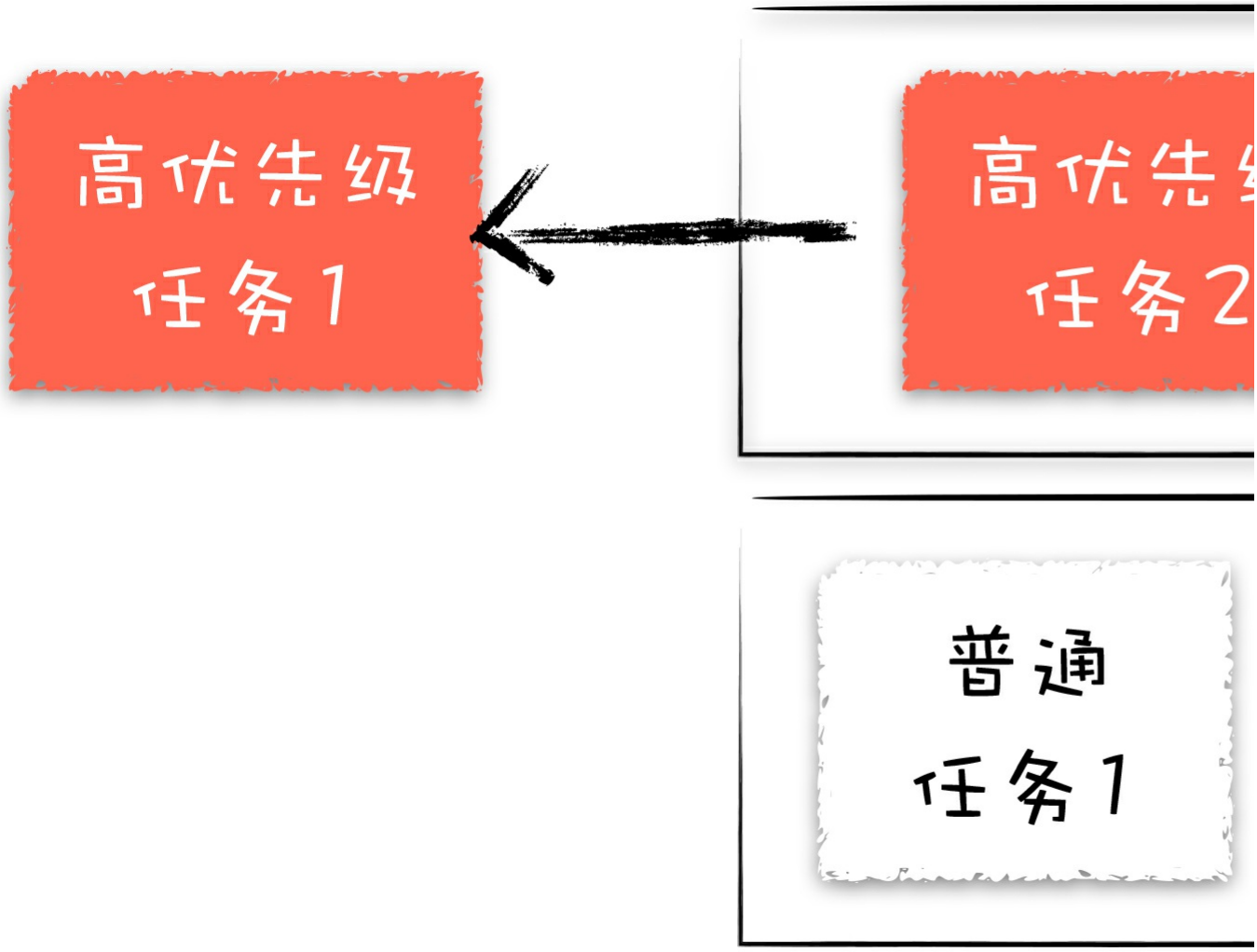
为了解决由于单消息队列而造成的队头阻塞问题，Chromium团队从2013年到现在，花了大量的精力在持续重构底层消息机制。在接下来的篇幅里，我会按照Chromium团队的重构消息系统的思路，来带你分析下他们是如何解决掉队头阻塞问题的。

1. 第一次迭代：引入一个高优先级队列

首先在最理想的情况下，我们希望能够快速跟踪高优先级任务，比如在交互阶段，下面几种任务都应该视为高优先级的任务：

- 通过鼠标触发的点击任务、滚动页面任务；
- 通过手势触发的页面缩放任务；
- 通过CSS、JavaScript等操作触发的动画特效等任务。

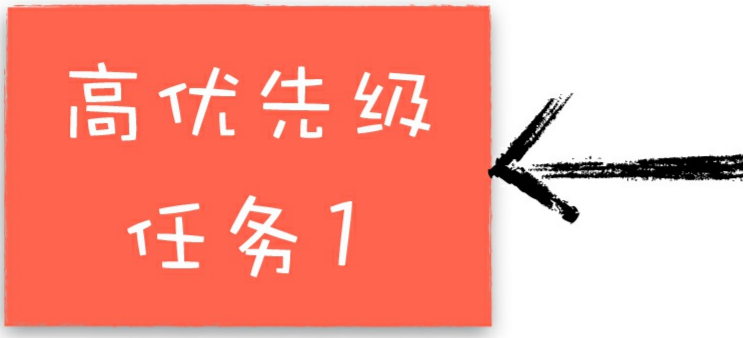
这些任务被触发后，用户想立即得到页面的反馈，所以我们需要让这些任务能够优先与其他的任务执行。要实现这种效果，我们可以增加一个高优级的消息队列，将高优先级的任务都添加到这个队列里面，然后优先执行该消息队列中的任务。最终效果如下图所示:



引入高优先级的消息队列

观察上图，我们使用了一个优先级高的消息队列和一个优先级低消息队列，渲染进程会将它认为是紧急的任务添加到高优先级队列中，不紧急的任务就添加到低优先级的队列中。然后我们再在渲染进程中引入一个**任务调度器**，负责从多个消息队列中选出合适的任务，通常实现的逻辑，先按照顺序从高优先级队列中取出任务，如果高优先级的队列为空，那么再按照顺序从低优级队列中取出任务。

我们还可以更进一步，将任务划分为多个不同的优先级，来实现更加细粒度的任务调度，比如可以划分为高优先级，普通优先级和低优先级，最终效果如下图所示：



增加多个不同优先级的消息队列

观察上图，我们实现了三个不同优先级的消息队列，然后可以使用任务调度器来统一调度这三个不同消息队列中的任务。

好了，现在我们引入了多个消息队列，结合任务调度器我们就可以灵活地调度任务了，这样我们就可以让高优先级的任务提前执行，采用这种方式似乎解决了消息队列的队头阻塞问题。

不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，甚至有可能出现还未处理输入事件，就合成了该事件要显示的图片。因此我们需要让一些相同类型的任务保持其相对执行顺序。

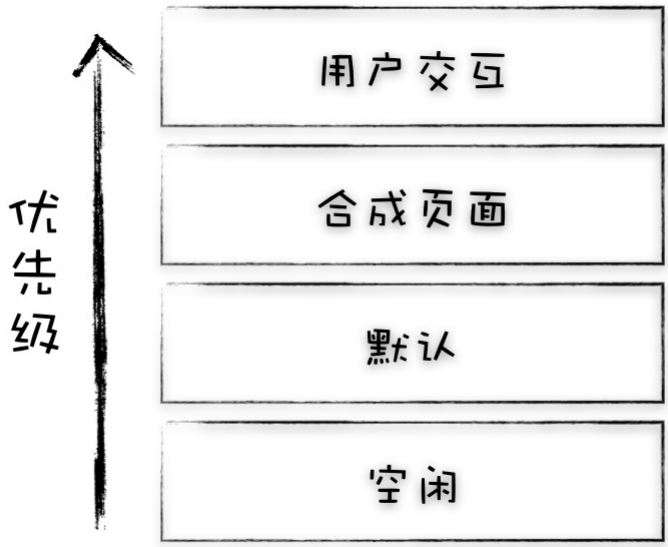
2. 第二次迭代：根据消息类型来实现消息队列

要解决上述问题，我们可以为不同类型的任务创建不同优先级的消息队列，比如：

- 可以创建输入事件的消息队列，用来存放输入事件。

- 可以创建合成任务的消息队列，用来存放合成事件。
- 可以创建默认消息队列，用来保存如资源加载的事件和定时器回调等事件。
- 还可以创建一个空闲消息队列，用来存放V8的垃圾自动垃圾回收这一类实时性不高的事件。

最终实现效果如下图所示：



根据消息类型实现不同优先级的消息队列

通过迭代，这种策略已经相当实用了，但是它依然存在着问题，那就是这几种消息队列的优先级都是固定的，任务调度器会按照这种固定好的静态的优先级来分别调度任务。那么静态优先级会带来什么问题呢？

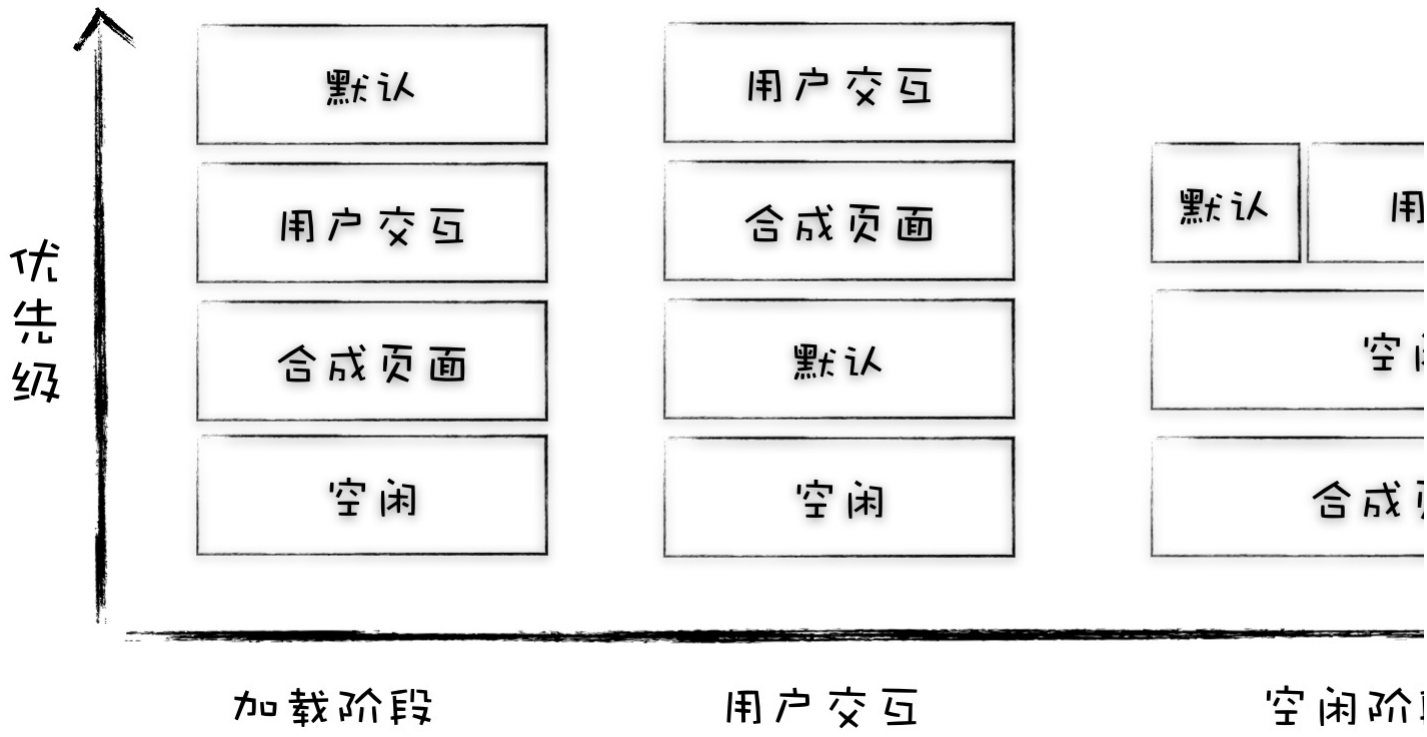
我们在《25 | 页面性能：如何系统地优化页面？》这节分析过页面的生存周期，页面大致的生存周期大体分为两个阶段，加载阶段和交互阶段。

虽然在交互阶段，采用上述这种静态优先级的策略没有什么太大问题的，但是在页面加载阶段，如果依然要优先执行用户输入事件和合成事件，那么页面的解析速度将会被拖慢。Chromium团队曾测试过这种情况，使用静态优先级策略，网页的加载速度会被拖慢14%。

3. 第三次迭代：动态调度策略

可以看出，我们所采用的优化策略像个跷跷板，虽然优化了高优先级任务，却拖慢低优先级任务，之所以会这样，是因为我们采取了静态的任务调度策略，对于各种不同的场景，这种静态策略就显得过于死板。

所以我们还得根据实际场景来继续平衡这个跷跷板，也就是说在不同的场景下，根据实际情况，动态调整消息队列的优先级。一图胜过千言，我们先看下图：



动态调度策略

这张图展示了Chromium在不同的场景下，是如何调整消息队列优先级的。通过这种动态调度策略，就可以满足不同场景的核心诉求了，同时这也是Chromium当前所采用的任务调度策略。

上图列出了三个不同的场景，分别是加载过程，合成过程以及正常状态。下面我们就结合这三种场景，来分析下Chromium为何做这种调整。

首先我们来看看**页面加载阶段**的场景，在这个阶段，用户的最高诉求是在尽可能短的时间内看到页面，至于交互和合成并不是这个阶段的核心诉求，因此我们需要调整策略，在加载阶段将页面解析，JavaScript脚本执行等任务调整为优先级最高的队列，降低交互合成这些队列的优先级。

页面加载完成之后就进入了**交互阶段**，在介绍Chromium是如何调整交互阶段的任务调度策略之前，我们还需要岔开一下，来回顾下页面的渲染过程。

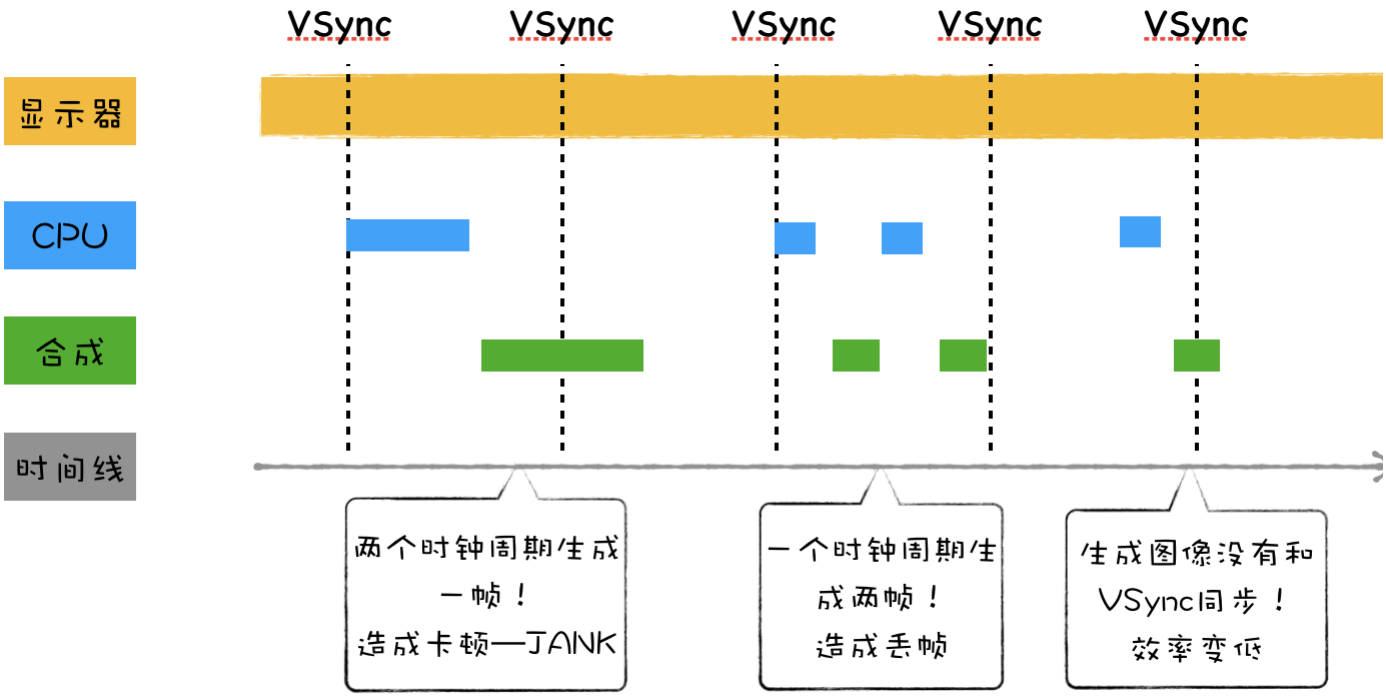
在《06 | 渲染流程（下）：HTML、CSS和JavaScript，是如何变成页面的？》和《24 | 分层和合成机制：为什么CSS动画比JavaScript高效？》这两节，我们分析了一个页面是如何渲染并显示出来的。

在显卡中有一块叫着**前缓冲区**的地方，这里存放着显示器要显示的图像，显示器会按照一定的频率来读取这块前缓冲区，并将前缓冲区中的图像显示在显示器上，不同的显示器读取的频率是不同的，通

常情况下是60HZ，也就是说显示器会每间隔1/60秒就读取一次前缓冲区。

如果浏览器要更新显示的图片，那么浏览器会将新生成的图片提交到显卡的**后缓冲区**中，提交完成之后，GPU会将**后缓冲区和前缓冲区互换位置**，也就是前缓冲区变成了后缓冲区，后缓冲区变成了前缓冲区，这就保证了显示器下次能读取到GPU中最新的图片。

这时候我们会发现，显示器从前缓冲区读取图片，和浏览器生成新的图像到后缓冲区的过程是不同步的，如下图所示：



VSync时钟周期和渲染引擎生成图片不同步问题

这种显示器读取图片和浏览器生成图片不同步，容易造成众多问题。

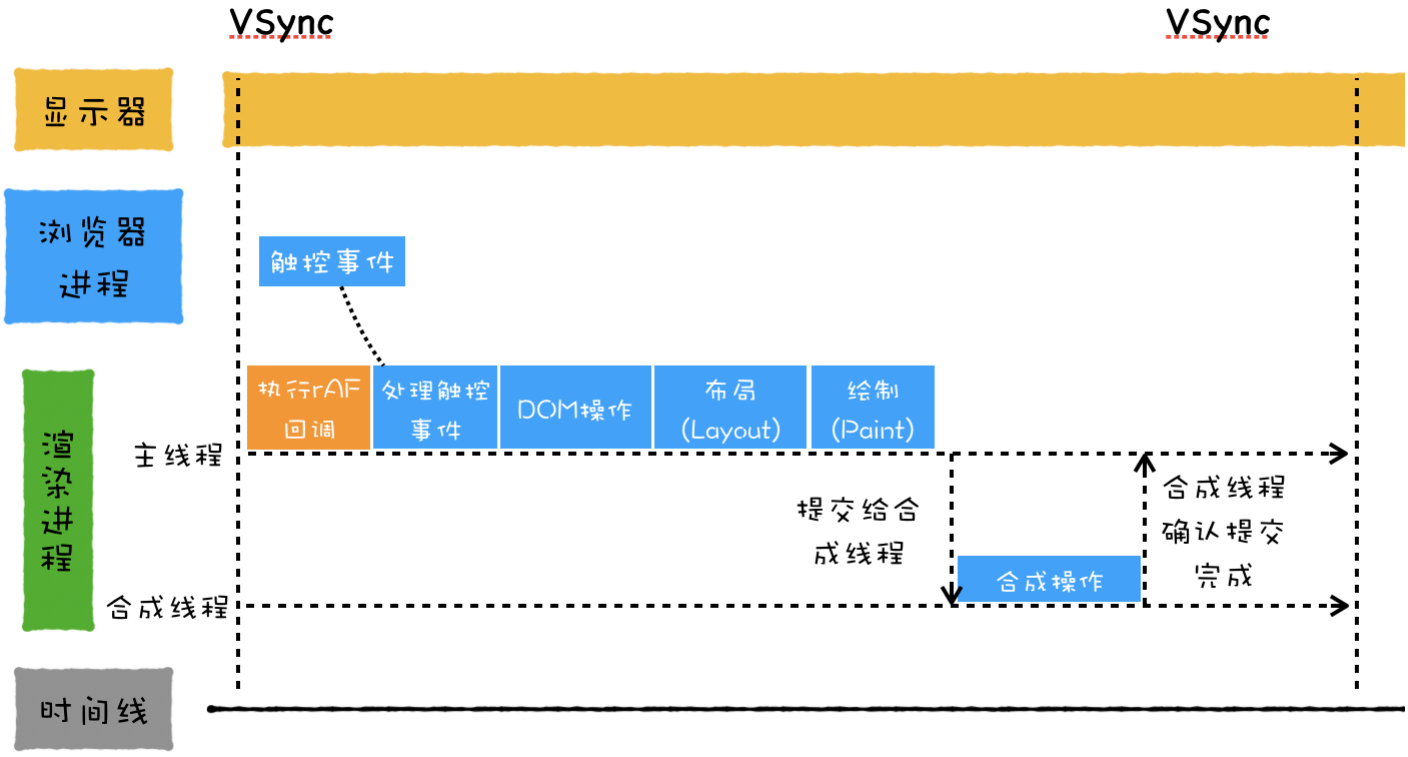
- 如果渲染进程生成的帧速比屏幕的刷新率慢，那么屏幕会在两帧中显示同一个画面，当这种断断续续的情况持续发生时，用户将会很明显地察觉到动画卡住了。
- 如果渲染进程生成的帧速率实际上比屏幕刷新率快，那么也会出现一些视觉上的问题，比如当帧速率在100fps而刷新率只有60Hz的时候，GPU所渲染的图像并非全都被显示出来，这就会造成丢帧现象。
- 就算屏幕的刷新频率和GPU更新图片的频率一样，由于它们是两个不同的系统，所以屏幕生成帧的周期和VSync的周期也是很难同步起来的。

所以VSync和系统的时钟不同步就会造成掉帧、卡顿、不连贯等问题。

为了解决这些问题，就需要将显示器的时钟同步周期和浏览器生成页面的周期绑定起来，Chromium也是这样实现，那么下面我们就来看看Chromium具体是怎么实现的？

当显示器将一帧画面绘制完成后，并在准备读取下一帧之前，显示器会发出一个垂直同步信号（vertical synchronization）给GPU，简称 VSync。这时候浏览器就会充分利用好VSync信号。

具体地讲，当GPU接收到VSync信号后，会将VSync信号同步给浏览器进程，浏览器进程再将其同步到对应的渲染进程，渲染进程接收到VSync信号之后，就可以准备绘制新的一帧了，具体流程你可以参考下图：



绑定VSync时钟同步周期和浏览器生成页面周期

上面其实是非常粗略的介绍，实际实现过程也是非常复杂的，如果感兴趣，你可以参考这篇文章。

好了，我们花了很大篇幅介绍了VSync和页面中的一帧是怎么显示出来，有了这些知识，我们就可以回到主线了，来分析下渲染进程是如何优化交互阶段页面的任务调度策略的？

从上图可以看出，当渲染进程接收到用户交互的任务后，接下来大概率是要进行绘制合成操作，因此我们可以设置，**当在执行用户交互的任务时，将合成任务的优先级调整到最高。**

接下来，处理完成DOM，计算好布局和绘制，就需要将信息提交给合成线程来合成最终图片了，然后合成线程进入工作状态。现在的场景是合成线程在工作了，**那么我们就可以把下个合成任务的优先级调整为最低，并将页面解析、定时器等任务优先级提升。**

在合成完成之后，合成线程会提交给渲染主线程提交完成合成的消息，如果当前合成操作执行的非常快，比如从用户发出消息到完成合成操作只花了8毫秒，因为VSync同步周期是16.66（1/60）毫秒，那么这个VSync时钟周期内就不需要再次生成新的页面了。那么从合成结束到下个VSync周期内，就进入了一个空闲时间阶段，那么就可以在这段空闲时间内执行一些不那么紧急的任务，比如V8的垃圾回收，或者通过window.requestIdleCallback()设置的回调任务等，都会在这段空闲时间内执行。

4. 第四次迭代：任务饿死

好了，以上方案看上去似乎非常完美了，不过依然存在一个问题，那就是在某个状态下，一直有新的高优先级的任务加入到队列中，这样就会导致其他低优先级的任务得不到执行，这称为任务饿死。

Chromium为了解决任务饿死的问题，给每个队列设置了执行权重，也就是如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样就缓解了任务饿死的情况。

总结

好了，本节的内容就介绍到这里，下面我来总结下本文的主要内容：

首先我们分析了基于单消息队列会引起队头阻塞的问题，为了解决队头阻塞问题，我们引入了多个不同优级的消息队列，并将紧急的任务添加到高优先级队列，不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，所以我们又迭代了第二个版本。

在第二个版本中，按照不同的任务类型来划分任务优先级，不过由于采用的静态优先级策略，对于其他一些场景，这种静态调度的策略并不是太适合，所以接下来，我们又迭代了第三版。

第三个版本，基于不同的场景来动态调整消息队列的优先级，到了这里已经非常完美了，不过依然存在着任务饿死的问题，为了解决任务饿死的问题，我们给每个队列一个权重，如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样我们就完成了Chromium的任务改造。

通过整个过程的分析，我们应该能理解，在开发一个项目时，不要试图去找最完美的方案，完美的方案往往是不存在的，我们需要根据实际的场景来寻找最适合我们的方案。

思考题

我们知道CSS动画是由渲染进程自动处理的，所以渲染进程会让CSS渲染每帧动画的过程与VSync的时钟保持一致,这样就能保证CSS动画的高效率执行。

但是JavaScript是由用户控制的，如果采用setTimeout来触发动画每帧的绘制，那么其绘制时机是很难和VSync时钟保持一致的，所以JavaScript中又引入了window.requestAnimationFrame，用来和VSync的时钟周期同步，那么我留给你的问题是：你知道requestAnimationFrame回调函数的执行时机吗？

参考资料

下面是我参考的一些资料：

- Blink Scheduler
- Blink Scheduler PPT
- Chrome的消息类型
- Chrome消息优先级
- 无头浏览器

欢迎在留言区分享你的想法。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

你好，我是李兵。

我们都知道，要想利用JavaScript实现高性能的动画，那就得使用requestAnimationFrame这个API，我们简称rAF，那么为什么都推荐使用rAF而不是setTimeOut呢？

要解释清楚这个问题，就要从渲染进程的任务调度系统讲起，理解了渲染进程任务调度系统，你自然就明白了rAF和setTimeOut的区别。其次，如果你理解任务调度系统，那么你就能将渲染流水线和浏览器系统架构等知识串起来，理解了这些概念也有助于你理解Performance标签是如何工作的。

要想了解最新Chrome的任务调度系统是怎么工作的，我们得先来回顾下之前介绍的消息循环系统，我们知道了渲染进程内部的大多数任务都是在主线程上执行的，诸如JavaScript执行、DOM、CSS、计算布局、V8的垃圾回收等任务。要让这些任务能够在主线程上有条不紊地运行，就需要引入消息队列。

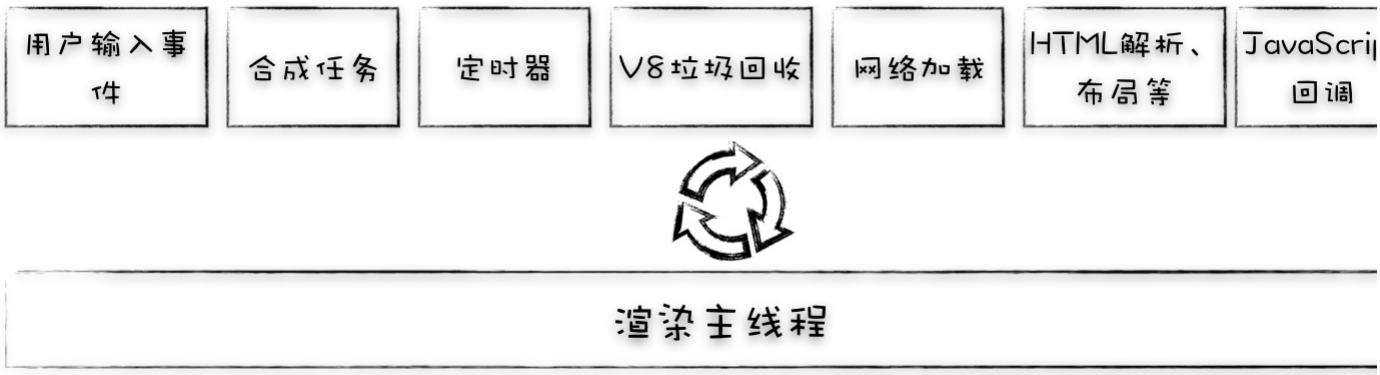
在前面的《16 | WebAPI：setTimeout是如何实现的？》这篇文章中，我们还介绍了，主线程维护了一个普通的消息队列和一个延迟消息队列，调度模块会按照规则依次取出这两个消息队列中的任务，并在主线程上执行。为了下文讲述方便，在这里我把普通的消息队列和延迟队列都当成一个消息队列。

新的任务都是被放进消息队列中去的，然后主线程再依次从消息队列中取出这些任务来顺序执行。这就是我们之前介绍的消息队列和事件循环系统。

单消息队列的队头阻塞问题

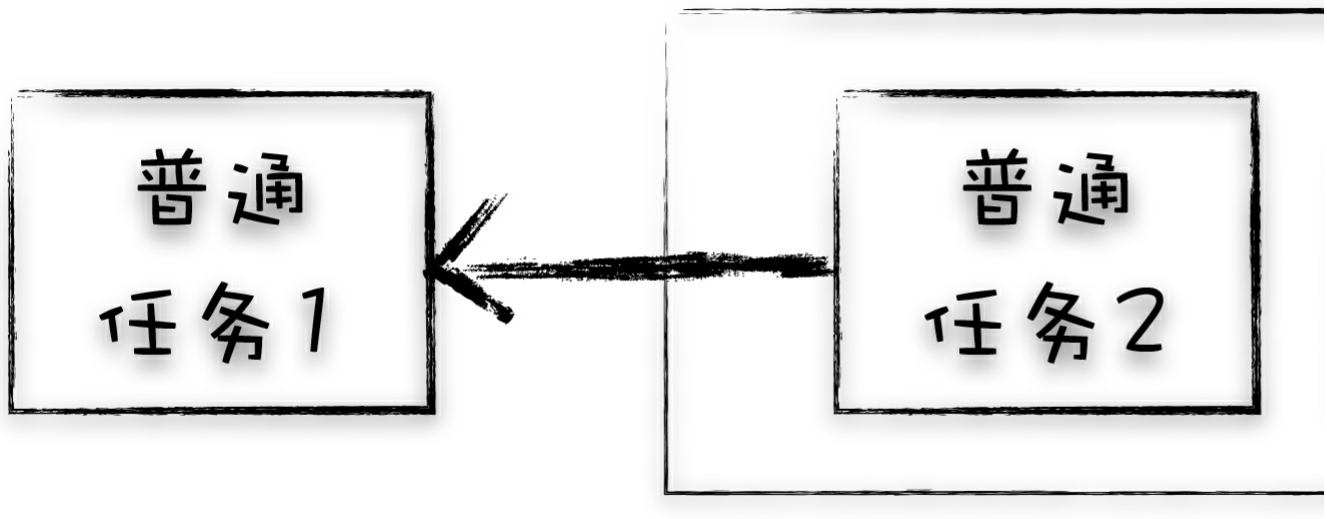
我们知道，渲染主线程会按照先进先出的顺序执行消息队列中的任务，具体地讲，当产生了新的任务，渲染进程会将其添加到消息队列尾部，在执行任务过程中，渲染进程会顺序地从消息队列头部取出任务并依次执行。

在最初，采用这种方式没有太大的问题，因为页面中的任务还不算太多，渲染主线程也不是太繁忙。不过浏览器是向前不停进化的，其进化路线体现在架构的调整、功能的增加以及更加精细的优化策略等方面，这些变化让渲染进程所需要处理的任务变多了，对应的渲染进程的主线程也变得越拥挤。下图所展示的仅仅是部分运行在主线程上的任务，你可以参考下：



任务和消息队列

你可以试想一下，在基于这种单消息队列的架构下，如果用户发出一个点击事件或者缩放页面的事件，而在此时，该任务前面可能还有很多不太重要的任务在排队等待着被执行，诸如V8的垃圾回收、DOM定时器等任务，如果执行这些任务需要花费的时间过久的话，那么就会让用户产生卡顿的感觉。你可以参看下图：



队头阻塞问题

因此，**在单消息队列架构下，存在着低优先级任务会阻塞高优先级任务的情况**，比如在一些性能不高的手机上，有时候滚动页面需要等待一秒以上。这像极了我们在介绍HTTP协议时所谈论的队头阻塞问题，那么我们也把这个问题称为消息队列的队头阻塞问题吧。

Chromium是如何解决队头阻塞问题的？

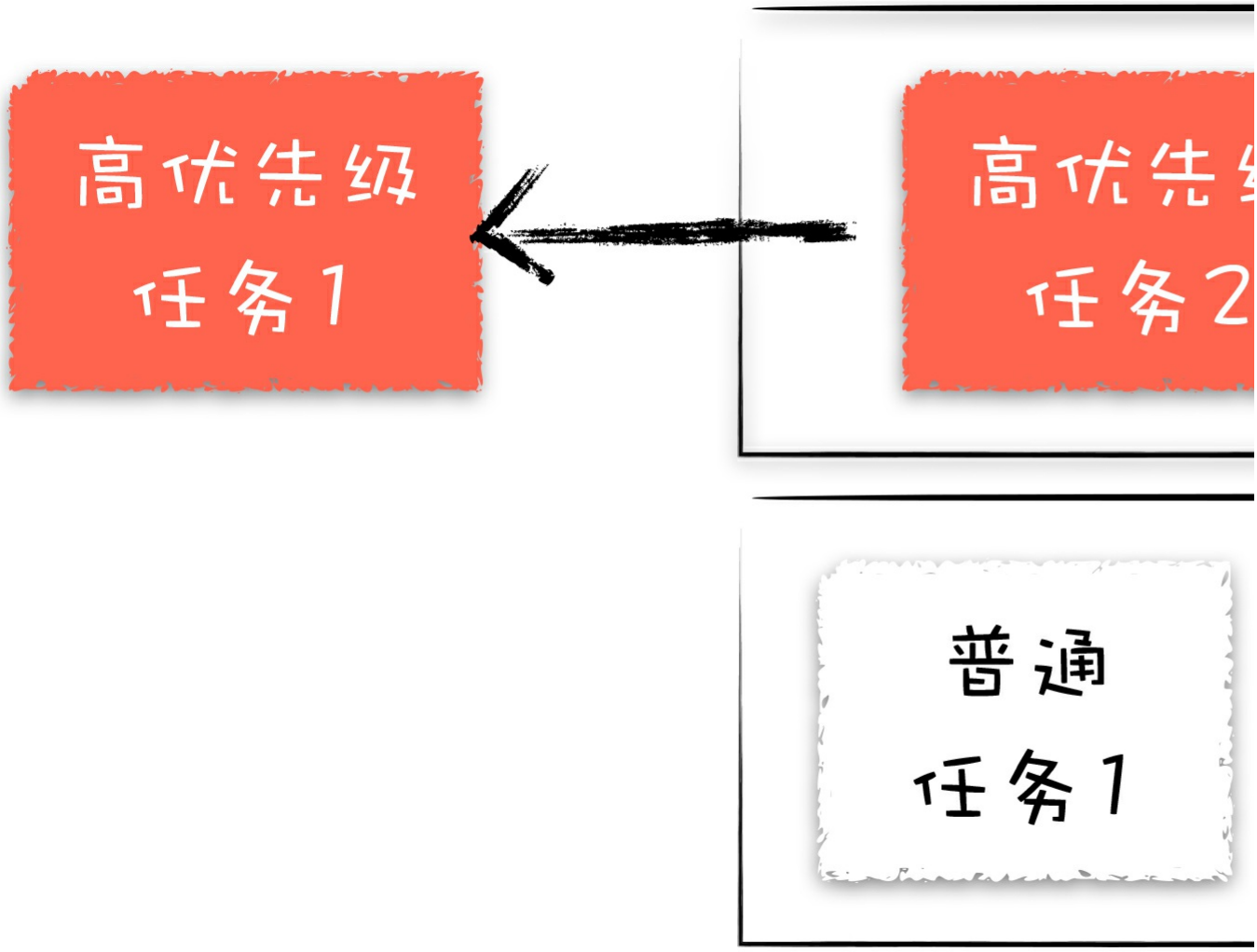
为了解决由于单消息队列而造成的队头阻塞问题，Chromium团队从2013年到现在，花了大量的精力在持续重构底层消息机制。在接下来的篇幅里，我会按照Chromium团队的重构消息系统的思路，来带你分析下他们是如何解决掉队头阻塞问题的。

1. 第一次迭代：引入一个高优先级队列

首先在最理想的情况下，我们希望能够快速跟踪高优先级任务，比如在交互阶段，下面几种任务都应该视为高优先级的任务：

- 通过鼠标触发的点击任务、滚动页面任务；
- 通过手势触发的页面缩放任务；
- 通过CSS、JavaScript等操作触发的动画特效等任务。

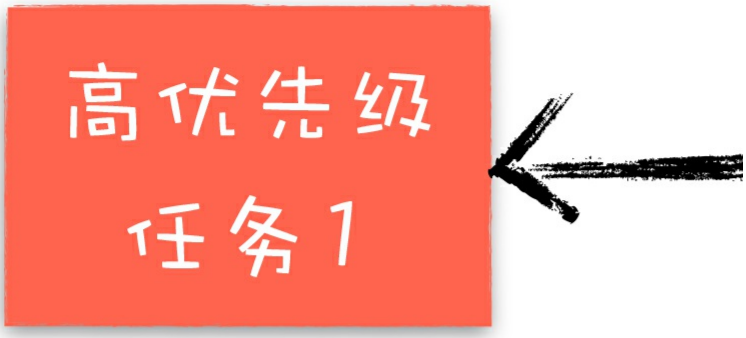
这些任务被触发后，用户想立即得到页面的反馈，所以我们需要让这些任务能够优先与其他的任务执行。要实现这种效果，我们可以增加一个高优级的消息队列，将高优先级的任务都添加到这个队列里面，然后优先执行该消息队列中的任务。最终效果如下图所示:



引入高优先级的消息队列

观察上图，我们使用了一个优先级高的消息队列和一个优先级低消息队列，渲染进程会将它认为是紧急的任务添加到高优先级队列中，不紧急的任务就添加到低优先级的队列中。然后我们再在渲染进程中引入一个**任务调度器**，负责从多个消息队列中选出合适的任务，通常实现的逻辑，先按照顺序从高优先级队列中取出任务，如果高优先级的队列为空，那么再按照顺序从低优级队列中取出任务。

我们还可以更进一步，将任务划分为多个不同的优先级，来实现更加细粒度的任务调度，比如可以划分为高优先级，普通优先级和低优先级，最终效果如下图所示：



增加多个不同优先级的消息队列

观察上图，我们实现了三个不同优先级的消息队列，然后可以使用任务调度器来统一调度这三个不同消息队列中的任务。

好了，现在我们引入了多个消息队列，结合任务调度器我们就可以灵活地调度任务了，这样我们就可以让高优先级的任务提前执行，采用这种方式似乎解决了消息队列的队头阻塞问题。

不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，甚至有可能出现还未处理输入事件，就合成了该事件要显示的图片。因此我们需要让一些相同类型的任务保持其相对执行顺序。

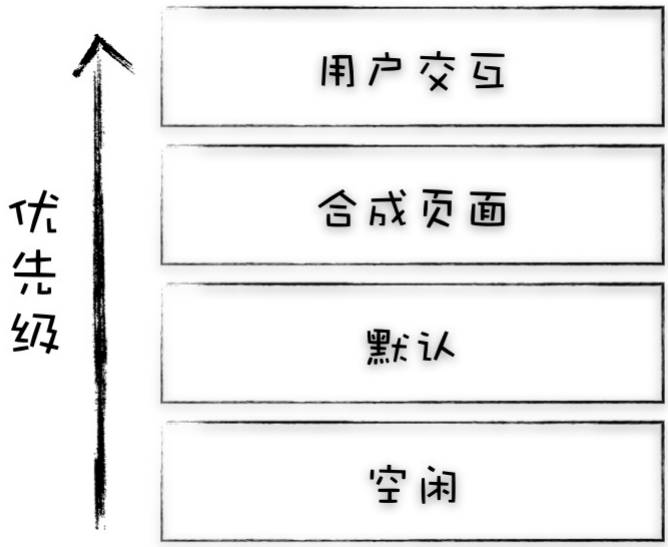
2. 第二次迭代：根据消息类型来实现消息队列

要解决上述问题，我们可以为不同类型的任务创建不同优先级的消息队列，比如：

- 可以创建输入事件的消息队列，用来存放输入事件。

- 可以创建合成任务的消息队列，用来存放合成事件。
- 可以创建默认消息队列，用来保存如资源加载的事件和定时器回调等事件。
- 还可以创建一个空闲消息队列，用来存放V8的垃圾自动垃圾回收这一类实时性不高的事件。

最终实现效果如下图所示：



根据消息类型实现不同优先级的消息队列

通过迭代，这种策略已经相当实用了，但是它依然存在着问题，那就是这几种消息队列的优先级都是固定的，任务调度器会按照这种固定好的静态的优先级来分别调度任务。那么静态优先级会带来什么问题呢？

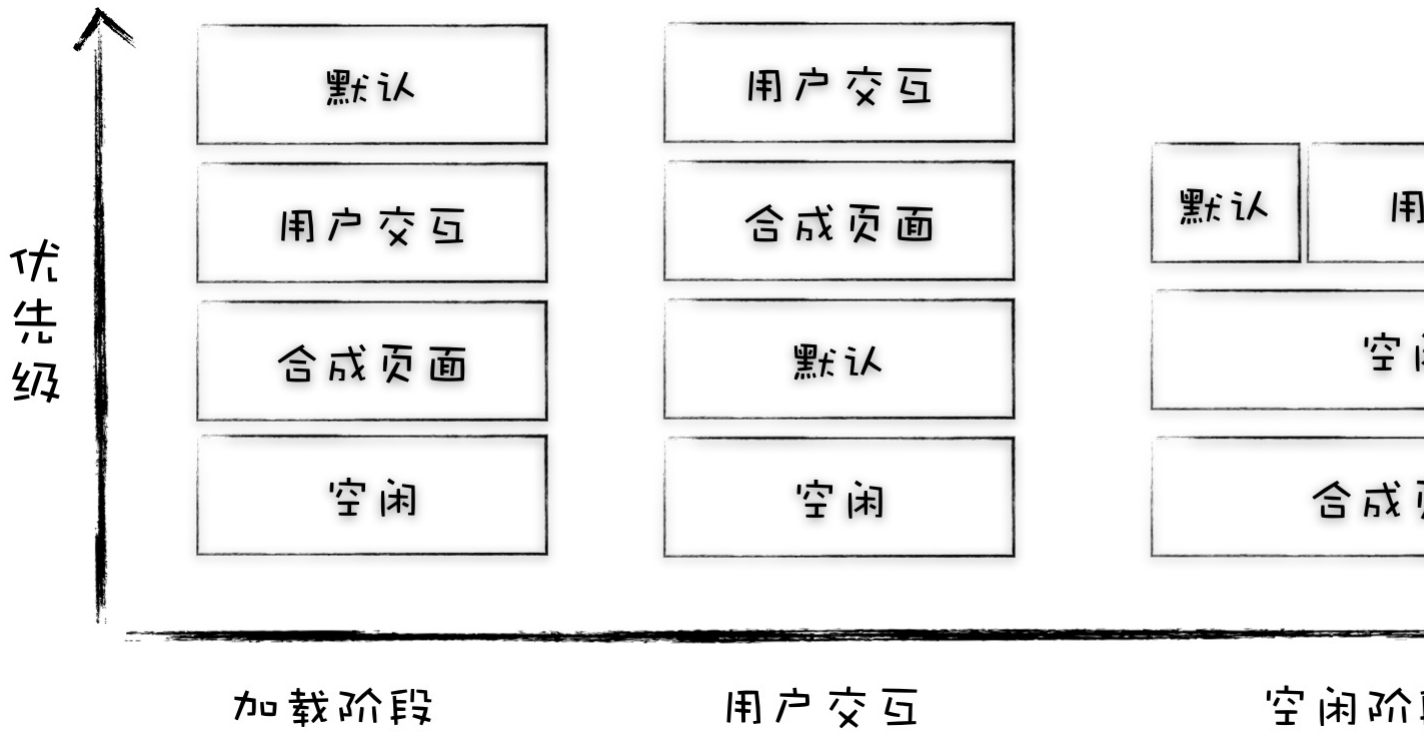
我们在《25 | 页面性能：如何系统地优化页面？》这节分析过页面的生存周期，页面大致的生存周期大体分为两个阶段，加载阶段和交互阶段。

虽然在交互阶段，采用上述这种静态优先级的策略没有什么太大问题的，但是在页面加载阶段，如果依然要优先执行用户输入事件和合成事件，那么页面的解析速度将会被拖慢。Chromium团队曾测试过这种情况，使用静态优先级策略，网页的加载速度会被拖慢14%。

3. 第三次迭代：动态调度策略

可以看出，我们所采用的优化策略像个跷跷板，虽然优化了高优先级任务，却拖慢低优先级任务，之所以会这样，是因为我们采取了静态的任务调度策略，对于各种不同的场景，这种静态策略就显得过于死板。

所以我们还得根据实际场景来继续平衡这个跷跷板，也就是说在不同的场景下，根据实际情况，动态调整消息队列的优先级。一图胜过千言，我们先看下图：



动态调度策略

这张图展示了Chromium在不同的场景下，是如何调整消息队列优先级的。通过这种动态调度策略，就可以满足不同场景的核心诉求了，同时这也是Chromium当前所采用的任务调度策略。

上图列出了三个不同的场景，分别是加载过程，合成过程以及正常状态。下面我们就结合这三种场景，来分析下Chromium为何做这种调整。

首先我们来看看**页面加载阶段**的场景，在这个阶段，用户的最高诉求是在尽可能短的时间内看到页面，至于交互和合成并不是这个阶段的核心诉求，因此我们需要调整策略，在加载阶段将页面解析，JavaScript脚本执行等任务调整为优先级最高的队列，降低交互合成这些队列的优先级。

页面加载完成之后就进入了**交互阶段**，在介绍Chromium是如何调整交互阶段的任务调度策略之前，我们还需要岔开一下，来回顾下页面的渲染过程。

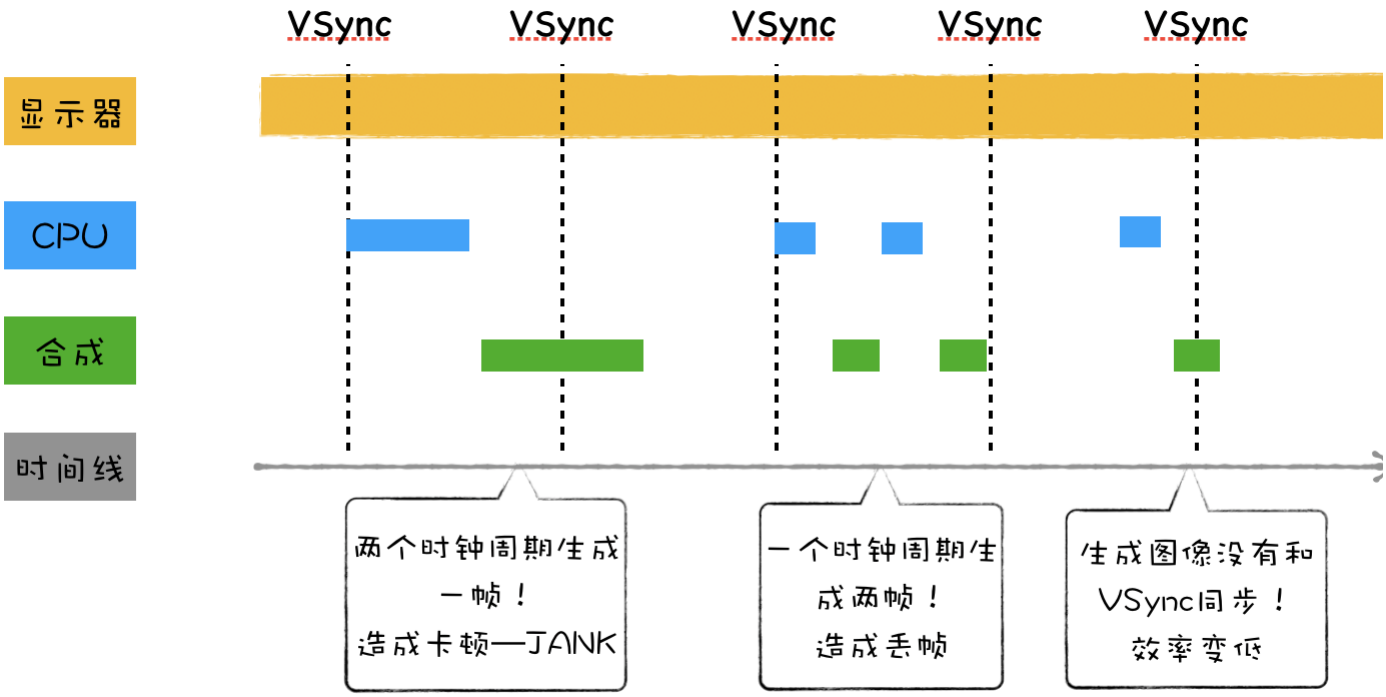
在《06 | 渲染流程（下）：HTML、CSS和JavaScript，是如何变成页面的？》和《24 | 分层和合成机制：为什么CSS动画比JavaScript高效？》这两节，我们分析了一个页面是如何渲染并显示出来的。

在显卡中有一块叫着**前缓冲区**的地方，这里存放着显示器要显示的图像，显示器会按照一定的频率来读取这块前缓冲区，并将前缓冲区中的图像显示在显示器上，不同的显示器读取的频率是不同的，通

常情况下是60HZ，也就是说显示器会每间隔1/60秒就读取一次前缓冲区。

如果浏览器要更新显示的图片，那么浏览器会将新生成的图片提交到显卡的**后缓冲区**中，提交完成之后，GPU会将**后缓冲区和前缓冲区互换位置**，也就是前缓冲区变成了后缓冲区，后缓冲区变成了前缓冲区，这就保证了显示器下次能读取到GPU中最新的图片。

这时候我们会发现，显示器从前缓冲区读取图片，和浏览器生成新的图像到后缓冲区的过程是不同步的，如下图所示：



VSync时钟周期和渲染引擎生成图片不同步问题

这种显示器读取图片和浏览器生成图片不同步，容易造成众多问题。

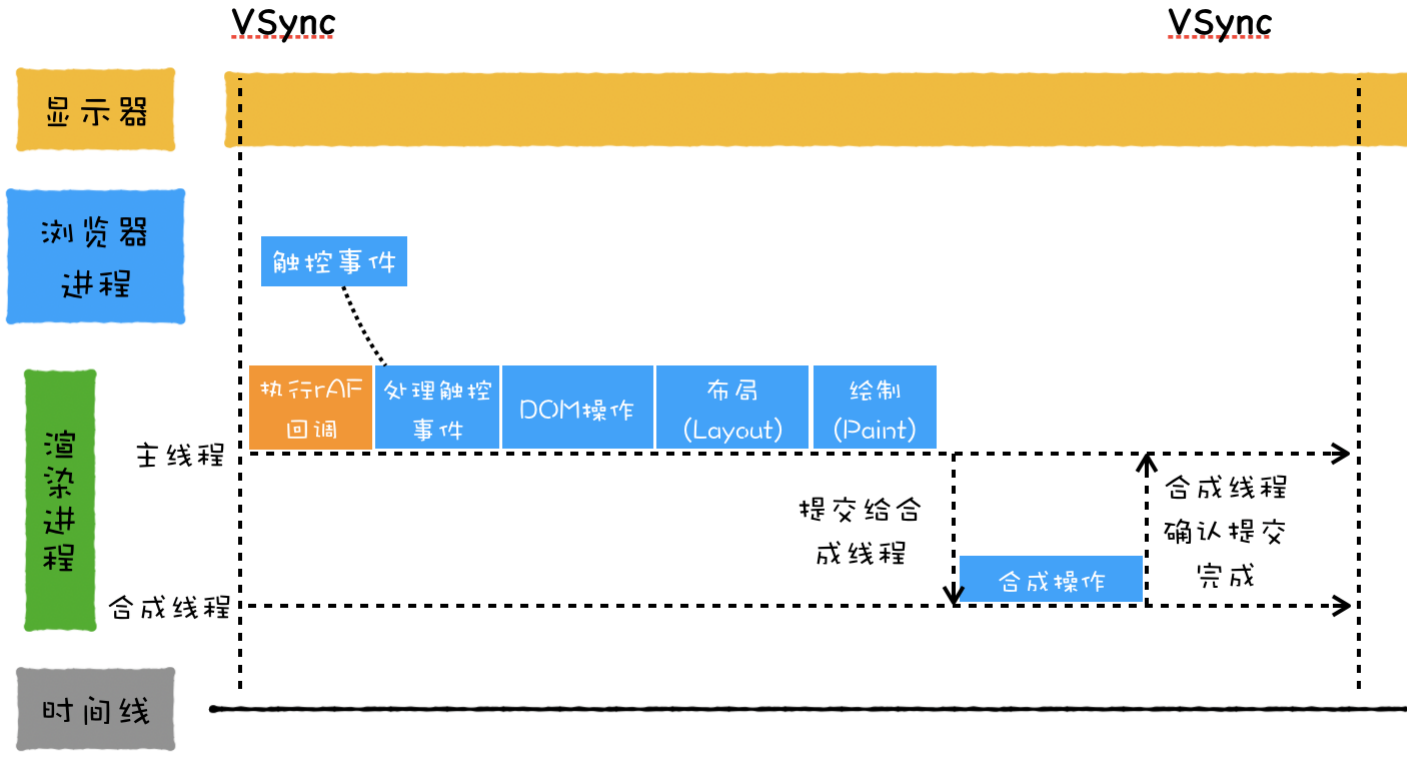
- 如果渲染进程生成的帧速比屏幕的刷新率慢，那么屏幕会在两帧中显示同一个画面，当这种断断续续的情况持续发生时，用户将会很明显地察觉到动画卡住了。
- 如果渲染进程生成的帧速率实际上比屏幕刷新率快，那么也会出现一些视觉上的问题，比如当帧速率在100fps而刷新率只有60Hz的时候，GPU所渲染的图像并非全都被显示出来，这就会造成丢帧现象。
- 就算屏幕的刷新频率和GPU更新图片的频率一样，由于它们是两个不同的系统，所以屏幕生成帧的周期和VSync的周期也是很难同步起来的。

所以VSync和系统的时钟不同步就会造成掉帧、卡顿、不连贯等问题。

为了解决这些问题，就需要将显示器的时钟同步周期和浏览器生成页面的周期绑定起来，Chromium也是这样实现，那么下面我们就来看看Chromium具体是怎么实现的？

当显示器将一帧画面绘制完成后，并在准备读取下一帧之前，显示器会发出一个垂直同步信号（vertical synchronization）给GPU，简称 VSync。这时候浏览器就会充分利用好VSync信号。

具体地讲，当GPU接收到VSync信号后，会将VSync信号同步给浏览器进程，浏览器进程再将其同步到对应的渲染进程，渲染进程接收到VSync信号之后，就可以准备绘制新的一帧了，具体流程你可以参考下图：



绑定VSync时钟同步周期和浏览器生成页面周期

上面其实是非常粗略的介绍，实际实现过程也是非常复杂的，如果感兴趣，你可以参考这篇文章。

好了，我们花了很大篇幅介绍了VSync和页面中的一帧是怎么显示出来，有了这些知识，我们就可以回到主线了，来分析下渲染进程是如何优化交互阶段页面的任务调度策略的？

从上图可以看出，当渲染进程接收到用户交互的任务后，接下来大概率是要进行绘制合成操作，因此我们可以设置，**当在执行用户交互的任务时，将合成任务的优先级调整到最高。**

接下来，处理完成DOM，计算好布局和绘制，就需要将信息提交给合成线程来合成最终图片了，然后合成线程进入工作状态。现在的场景是合成线程在工作了，**那么我们就可以把下个合成任务的优先级调整为最低，并将页面解析、定时器等任务优先级提升。**

在合成完成之后，合成线程会提交给渲染主线程提交完成合成的消息，如果当前合成操作执行的非常快，比如从用户发出消息到完成合成操作只花了8毫秒，因为VSync同步周期是16.66（1/60）毫秒，那么这个VSync时钟周期内就不需要再次生成新的页面了。那么从合成结束到下个VSync周期内，就进入了一个空闲时间阶段，那么就可以在这段空闲时间内执行一些不那么紧急的任务，比如V8的垃圾回收，或者通过window.requestIdleCallback()设置的回调任务等，都会在这段空闲时间内执行。

4. 第四次迭代：任务饿死

好了，以上方案看上去似乎非常完美了，不过依然存在一个问题，那就是在某个状态下，一直有新的高优先级的任务加入到队列中，这样就会导致其他低优先级的任务得不到执行，这称为任务饿死。

Chromium为了解决任务饿死的问题，给每个队列设置了执行权重，也就是如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样就缓解了任务饿死的情况。

总结

好了，本节的内容就介绍到这里，下面我来总结下本文的主要内容：

首先我们分析了基于单消息队列会引起队头阻塞的问题，为了解决队头阻塞问题，我们引入了多个不同优级的消息队列，并将紧急的任务添加到高优先级队列，不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，所以我们又迭代了第二个版本。

在第二个版本中，按照不同的任务类型来划分任务优先级，不过由于采用的静态优先级策略，对于其他一些场景，这种静态调度的策略并不是太适合，所以接下来，我们又迭代了第三版。

第三个版本，基于不同的场景来动态调整消息队列的优先级，到了这里已经非常完美了，不过依然存在着任务饿死的问题，为了解决任务饿死的问题，我们给每个队列一个权重，如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样我们就完成了Chromium的任务改造。

通过整个过程的分析，我们应该能理解，在开发一个项目时，不要试图去找最完美的方案，完美的方案往往是不存在的，我们需要根据实际的场景来寻找最适合我们的方案。

思考题

我们知道CSS动画是由渲染进程自动处理的，所以渲染进程会让CSS渲染每帧动画的过程与VSync的时钟保持一致,这样就能保证CSS动画的高效率执行。

但是JavaScript是由用户控制的，如果采用setTimeout来触发动画每帧的绘制，那么其绘制时机是很难和VSync时钟保持一致的，所以JavaScript中又引入了window.requestAnimationFrame，用来和VSync的时钟周期同步，那么我留给你的问题是：你知道requestAnimationFrame回调函数的执行时机吗？

参考资料

下面是我参考的一些资料：

- Blink Scheduler
- Blink Scheduler PPT
- Chrome的消息类型
- Chrome消息优先级
- 无头浏览器

欢迎在留言区分享你的想法。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

你好，我是李兵。

我们都知道，要想利用JavaScript实现高性能的动画，那就得使用requestAnimationFrame这个API，我们简称rAF，那么为什么都推荐使用rAF而不是setTimeOut呢？

要解释清楚这个问题，就要从渲染进程的任务调度系统讲起，理解了渲染进程任务调度系统，你自然就明白了rAF和setTimeOut的区别。其次，如果你理解任务调度系统，那么你就能将渲染流水线和浏览器系统架构等知识串起来，理解了这些概念也有助于你理解Performance标签是如何工作的。

要想了解最新Chrome的任务调度系统是怎么工作的，我们得先来回顾下之前介绍的消息循环系统，我们知道了渲染进程内部的大多数任务都是在主线程上执行的，诸如JavaScript执行、DOM、CSS、计算布局、V8的垃圾回收等任务。要让这些任务能够在主线程上有条不紊地运行，就需要引入消息队列。

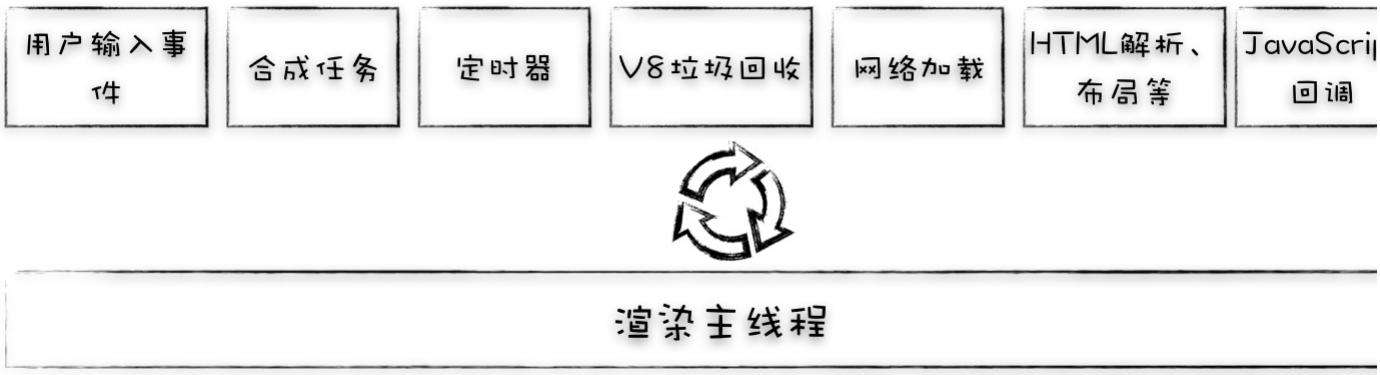
在前面的《16 | WebAPI：setTimeout是如何实现的？》这篇文章中，我们还介绍了，主线程维护了一个普通的消息队列和一个延迟消息队列，调度模块会按照规则依次取出这两个消息队列中的任务，并在主线程上执行。为了下文讲述方便，在这里我把普通的消息队列和延迟队列都当成一个消息队列。

新的任务都是被放进消息队列中去的，然后主线程再依次从消息队列中取出这些任务来顺序执行。这就是我们之前介绍的消息队列和事件循环系统。

单消息队列的队头阻塞问题

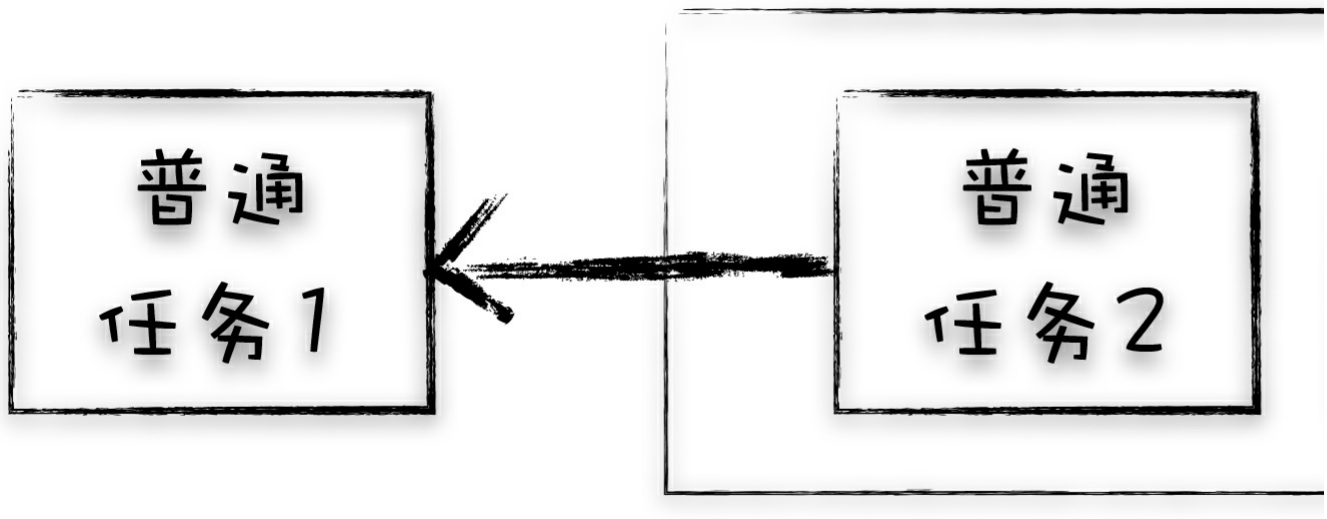
我们知道，渲染主线程会按照先进先出的顺序执行消息队列中的任务，具体地讲，当产生了新的任务，渲染进程会将其添加到消息队列尾部，在执行任务过程中，渲染进程会顺序地从消息队列头部取出任务并依次执行。

在最初，采用这种方式没有太大的问题，因为页面中的任务还不算太多，渲染主线程也不是太繁忙。不过浏览器是向前不停进化的，其进化路线体现在架构的调整、功能的增加以及更加精细的优化策略等方面，这些变化让渲染进程所需要处理的任务变多了，对应的渲染进程的主线程也变得越拥挤。下图所展示的仅仅是部分运行在主线程上的任务，你可以参考下：



任务和消息队列

你可以试想一下，在基于这种单消息队列的架构下，如果用户发出一个点击事件或者缩放页面的事件，而在此时，该任务前面可能还有很多不太重要的任务在排队等待着被执行，诸如V8的垃圾回收、DOM定时器等任务，如果执行这些任务需要花费的时间过久的话，那么就会让用户产生卡顿的感觉。你可以参看下图：



队头阻塞问题

因此，**在单消息队列架构下，存在着低优先级任务会阻塞高优先级任务的情况**，比如在一些性能不高的手机上，有时候滚动页面需要等待一秒以上。这像极了我们在介绍HTTP协议时所谈论的队头阻塞问题，那么我们也把这个问题称为消息队列的队头阻塞问题吧。

Chromium是如何解决队头阻塞问题的？

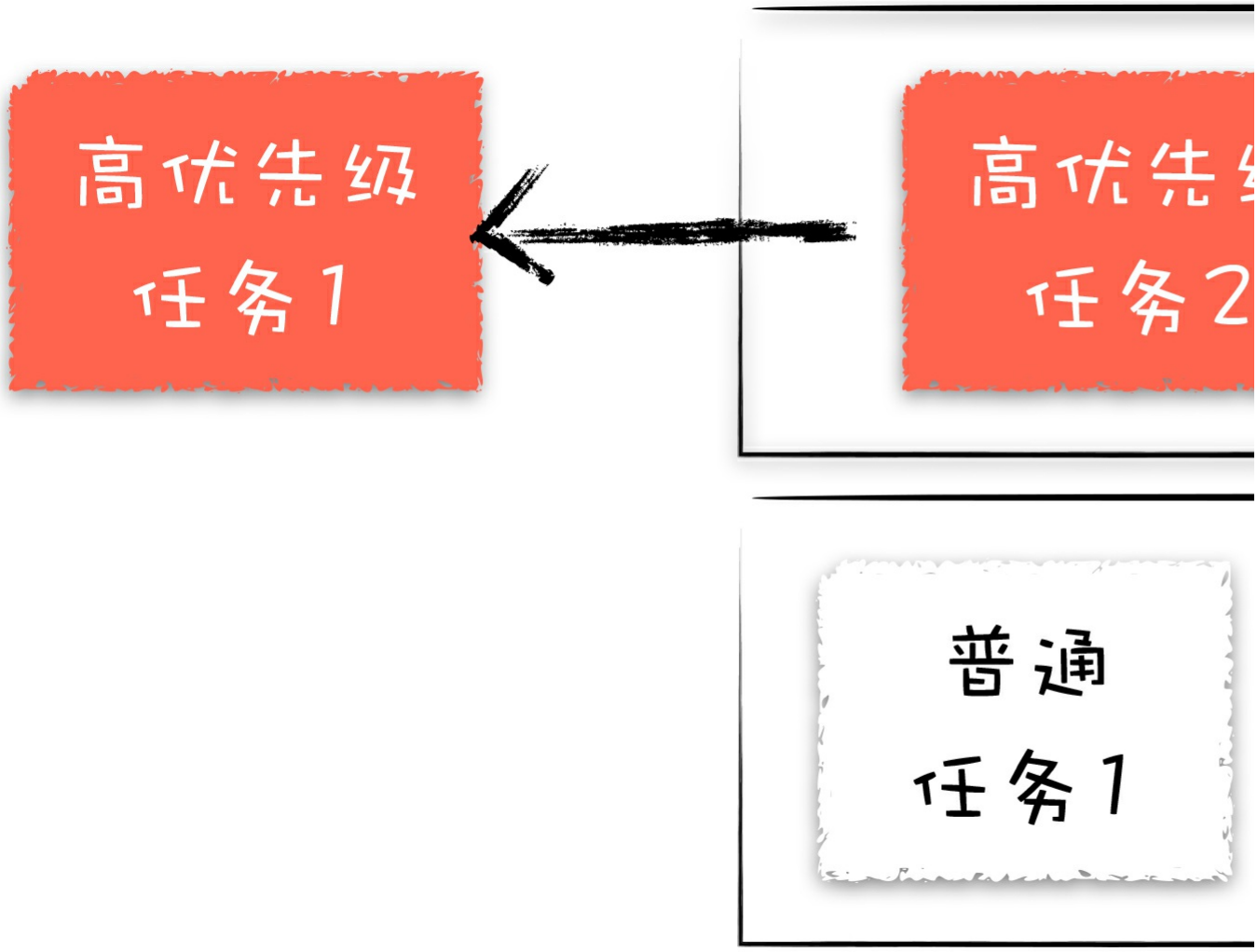
为了解决由于单消息队列而造成的队头阻塞问题，Chromium团队从2013年到现在，花了大量的精力在持续重构底层消息机制。在接下来的篇幅里，我会按照Chromium团队的重构消息系统的思路，来带你分析下他们是如何解决掉队头阻塞问题的。

1. 第一次迭代：引入一个高优先级队列

首先在最理想的情况下，我们希望能够快速跟踪高优先级任务，比如在交互阶段，下面几种任务都应该视为高优先级的任务：

- 通过鼠标触发的点击任务、滚动页面任务；
- 通过手势触发的页面缩放任务；
- 通过CSS、JavaScript等操作触发的动画特效等任务。

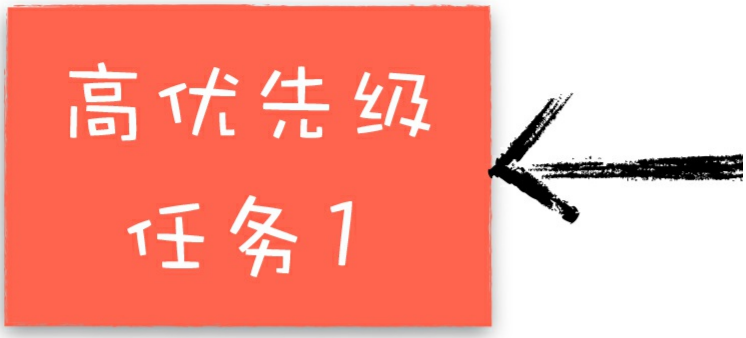
这些任务被触发后，用户想立即得到页面的反馈，所以我们需要让这些任务能够优先与其他的任务执行。要实现这种效果，我们可以增加一个高优级的消息队列，将高优先级的任务都添加到这个队列里面，然后优先执行该消息队列中的任务。最终效果如下图所示:



引入高优先级的消息队列

观察上图，我们使用了一个优先级高的消息队列和一个优先级低消息队列，渲染进程会将它认为是紧急的任务添加到高优先级队列中，不紧急的任务就添加到低优先级的队列中。然后我们再在渲染进程中引入一个**任务调度器**，负责从多个消息队列中选出合适的任务，通常实现的逻辑，先按照顺序从高优先级队列中取出任务，如果高优先级的队列为空，那么再按照顺序从低优级队列中取出任务。

我们还可以更进一步，将任务划分为多个不同的优先级，来实现更加细粒度的任务调度，比如可以划分为高优先级，普通优先级和低优先级，最终效果如下图所示：



增加多个不同优先级的消息队列

观察上图，我们实现了三个不同优先级的消息队列，然后可以使用任务调度器来统一调度这三个不同消息队列中的任务。

好了，现在我们引入了多个消息队列，结合任务调度器我们就可以灵活地调度任务了，这样我们就可以让高优先级的任务提前执行，采用这种方式似乎解决了消息队列的队头阻塞问题。

不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，甚至有可能出现还未处理输入事件，就合成了该事件要显示的图片。因此我们需要让一些相同类型的任务保持其相对执行顺序。

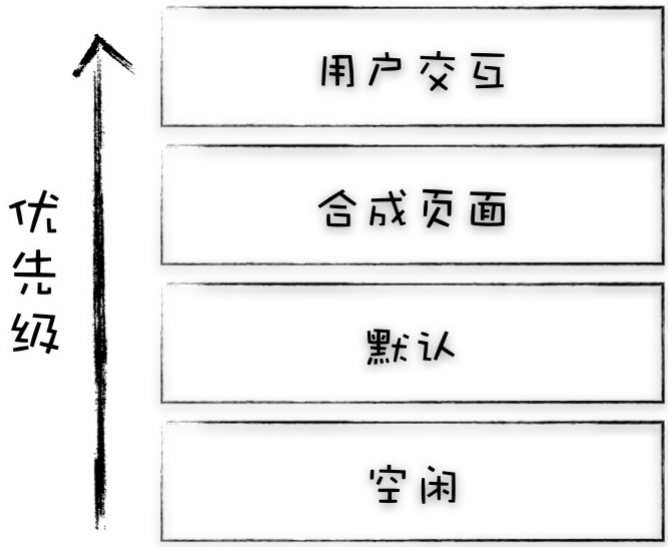
2. 第二次迭代：根据消息类型来实现消息队列

要解决上述问题，我们可以为不同类型的任务创建不同优先级的消息队列，比如：

- 可以创建输入事件的消息队列，用来存放输入事件。

- 可以创建合成任务的消息队列，用来存放合成事件。
- 可以创建默认消息队列，用来保存如资源加载的事件和定时器回调等事件。
- 还可以创建一个空闲消息队列，用来存放V8的垃圾自动垃圾回收这一类实时性不高的事件。

最终实现效果如下图所示：



根据消息类型实现不同优先级的消息队列

通过迭代，这种策略已经相当实用了，但是它依然存在着问题，那就是这几种消息队列的优先级都是固定的，任务调度器会按照这种固定好的静态的优先级来分别调度任务。那么静态优先级会带来什么问题呢？

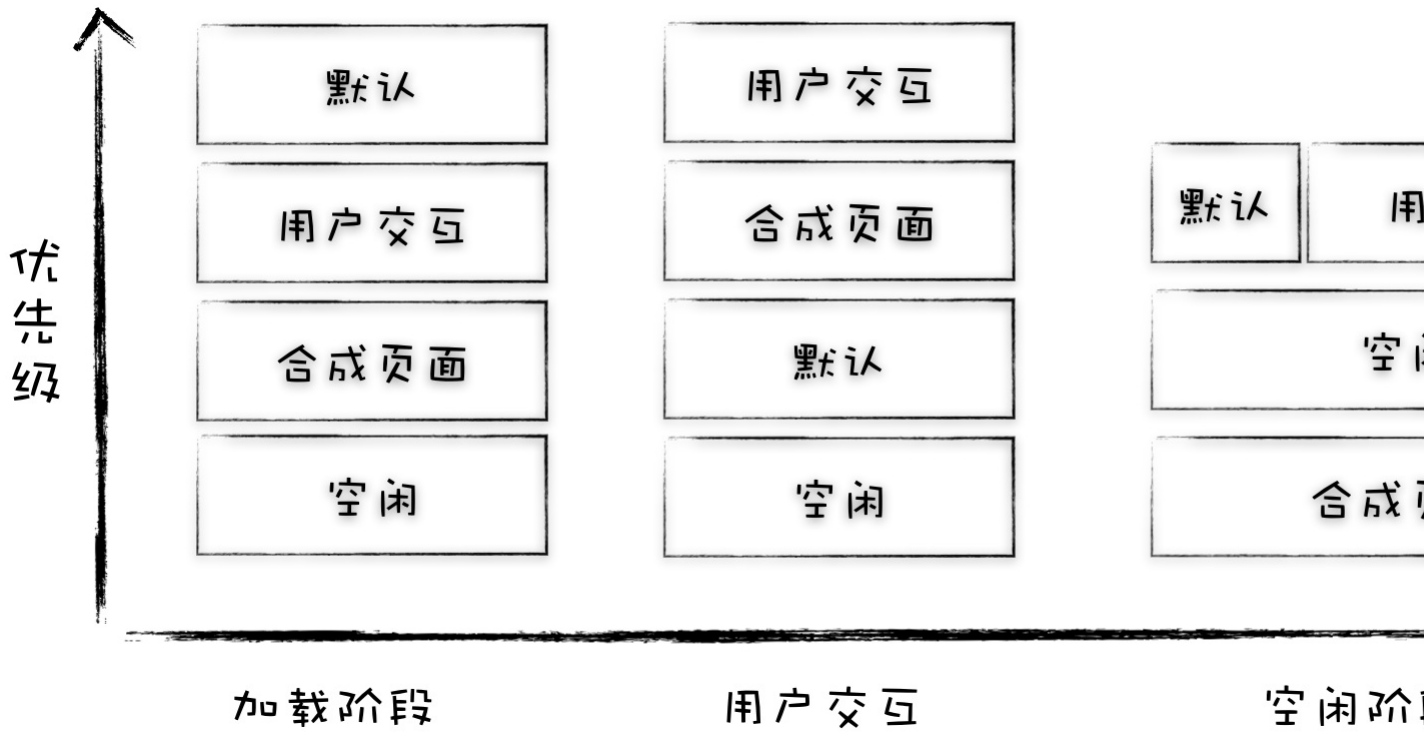
我们在《25 | 页面性能：如何系统地优化页面？》这节分析过页面的生存周期，页面大致的生存周期大体分为两个阶段，加载阶段和交互阶段。

虽然在交互阶段，采用上述这种静态优先级的策略没有什么太大问题的，但是在页面加载阶段，如果依然要优先执行用户输入事件和合成事件，那么页面的解析速度将会被拖慢。Chromium团队曾测试过这种情况，使用静态优先级策略，网页的加载速度会被拖慢14%。

3. 第三次迭代：动态调度策略

可以看出，我们所采用的优化策略像个跷跷板，虽然优化了高优先级任务，却拖慢低优先级任务，之所以会这样，是因为我们采取了静态的任务调度策略，对于各种不同的场景，这种静态策略就显得过于死板。

所以我们还得根据实际场景来继续平衡这个跷跷板，也就是说在不同的场景下，根据实际情况，动态调整消息队列的优先级。一图胜过千言，我们先看下图：



动态调度策略

这张图展示了Chromium在不同的场景下，是如何调整消息队列优先级的。通过这种动态调度策略，就可以满足不同场景的核心诉求了，同时这也是Chromium当前所采用的任务调度策略。

上图列出了三个不同的场景，分别是加载过程，合成过程以及正常状态。下面我们就结合这三种场景，来分析下Chromium为何做这种调整。

首先我们来看看**页面加载阶段**的场景，在这个阶段，用户的最高诉求是在尽可能短的时间内看到页面，至于交互和合成并不是这个阶段的核心诉求，因此我们需要调整策略，在加载阶段将页面解析，JavaScript脚本执行等任务调整为优先级最高的队列，降低交互合成这些队列的优先级。

页面加载完成之后就进入了**交互阶段**，在介绍Chromium是如何调整交互阶段的任务调度策略之前，我们还需要岔开一下，来回顾下页面的渲染过程。

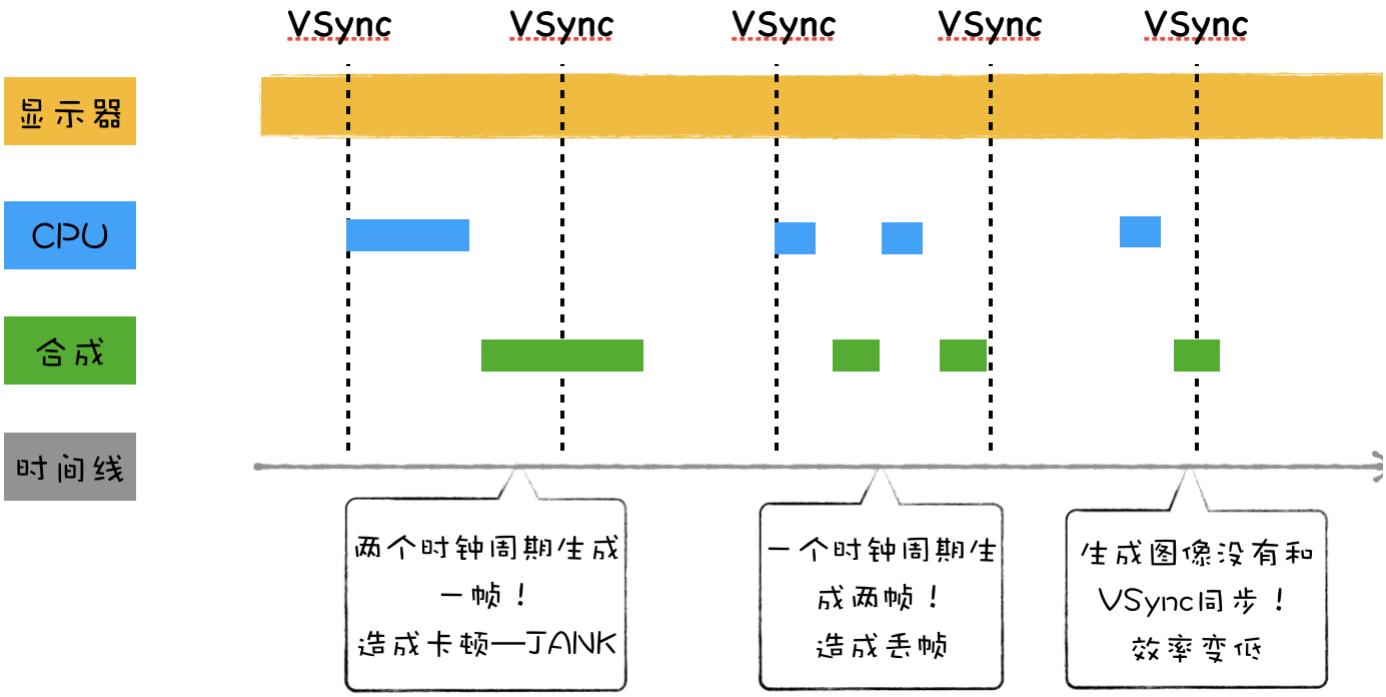
在《06 | 渲染流程（下）：HTML、CSS和JavaScript，是如何变成页面的？》和《24 | 分层和合成机制：为什么CSS动画比JavaScript高效？》这两节，我们分析了一个页面是如何渲染并显示出来的。

在显卡中有一块叫着**前缓冲区**的地方，这里存放着显示器要显示的图像，显示器会按照一定的频率来读取这块前缓冲区，并将前缓冲区中的图像显示在显示器上，不同的显示器读取的频率是不同的，通

常情况下是60HZ，也就是说显示器会每间隔1/60秒就读取一次前缓冲区。

如果浏览器要更新显示的图片，那么浏览器会将新生成的图片提交到显卡的**后缓冲区**中，提交完成之后，GPU会将**后缓冲区和前缓冲区互换位置**，也就是前缓冲区变成了后缓冲区，后缓冲区变成了前缓冲区，这就保证了显示器下次能读取到GPU中最新的图片。

这时候我们会发现，显示器从前缓冲区读取图片，和浏览器生成新的图像到后缓冲区的过程是不同步的，如下图所示：



VSync时钟周期和渲染引擎生成图片不同步问题

这种显示器读取图片和浏览器生成图片不同步，容易造成众多问题。

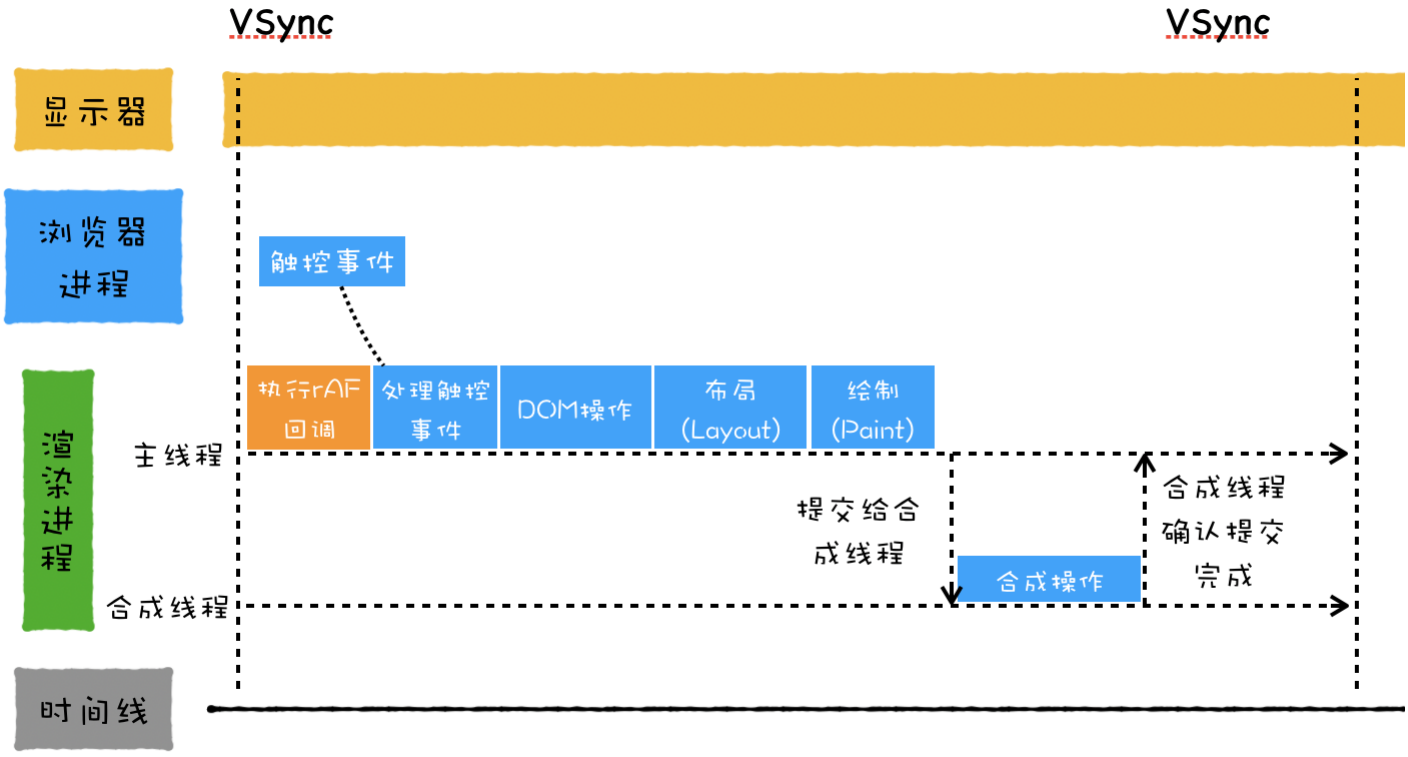
- 如果渲染进程生成的帧速比屏幕的刷新率慢，那么屏幕会在两帧中显示同一个画面，当这种断断续续的情况持续发生时，用户将会很明显地察觉到动画卡住了。
- 如果渲染进程生成的帧速率实际上比屏幕刷新率快，那么也会出现一些视觉上的问题，比如当帧速率在100fps而刷新率只有60Hz的时候，GPU所渲染的图像并非全都被显示出来，这就会造成丢帧现象。
- 就算屏幕的刷新频率和GPU更新图片的频率一样，由于它们是两个不同的系统，所以屏幕生成帧的周期和VSync的周期也是很难同步起来的。

所以VSync和系统的时钟不同步就会造成掉帧、卡顿、不连贯等问题。

为了解决这些问题，就需要将显示器的时钟同步周期和浏览器生成页面的周期绑定起来，Chromium也是这样实现，那么下面我们就来看看Chromium具体是怎么实现的？

当显示器将一帧画面绘制完成后，并在准备读取下一帧之前，显示器会发出一个垂直同步信号（vertical synchronization）给GPU，简称 VSync。这时候浏览器就会充分利用好VSync信号。

具体地讲，当GPU接收到VSync信号后，会将VSync信号同步给浏览器进程，浏览器进程再将其同步到对应的渲染进程，渲染进程接收到VSync信号之后，就可以准备绘制新的一帧了，具体流程你可以参考下图：



绑定VSync时钟同步周期和浏览器生成页面周期

上面其实是非常粗略的介绍，实际实现过程也是非常复杂的，如果感兴趣，你可以参考这篇文章。

好了，我们花了很大篇幅介绍了VSync和页面中的一帧是怎么显示出来，有了这些知识，我们就可以回到主线了，来分析下渲染进程是如何优化交互阶段页面的任务调度策略的？

从上图可以看出，当渲染进程接收到用户交互的任务后，接下来大概率是要进行绘制合成操作，因此我们可以设置，**当在执行用户交互的任务时，将合成任务的优先级调整到最高。**

接下来，处理完成DOM，计算好布局和绘制，就需要将信息提交给合成线程来合成最终图片了，然后合成线程进入工作状态。现在的场景是合成线程在工作了，**那么我们就可以把下个合成任务的优先级调整为最低，并将页面解析、定时器等任务优先级提升。**

在合成完成之后，合成线程会提交给渲染主线程提交完成合成的消息，如果当前合成操作执行的非常快，比如从用户发出消息到完成合成操作只花了8毫秒，因为VSync同步周期是16.66（1/60）毫秒，那么这个VSync时钟周期内就不需要再次生成新的页面了。那么从合成结束到下个VSync周期内，就进入了一个空闲时间阶段，那么就可以在这段空闲时间内执行一些不那么紧急的任务，比如V8的垃圾回收，或者通过window.requestIdleCallback()设置的回调任务等，都会在这段空闲时间内执行。

4. 第四次迭代：任务饿死

好了，以上方案看上去似乎非常完美了，不过依然存在一个问题，那就是在某个状态下，一直有新的高优先级的任务加入到队列中，这样就会导致其他低优先级的任务得不到执行，这称为任务饿死。

Chromium为了解决任务饿死的问题，给每个队列设置了执行权重，也就是如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样就缓解了任务饿死的情况。

总结

好了，本节的内容就介绍到这里，下面我来总结下本文的主要内容：

首先我们分析了基于单消息队列会引起队头阻塞的问题，为了解决队头阻塞问题，我们引入了多个不同优级的消息队列，并将紧急的任务添加到高优先级队列，不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，所以我们又迭代了第二个版本。

在第二个版本中，按照不同的任务类型来划分任务优先级，不过由于采用的静态优先级策略，对于其他一些场景，这种静态调度的策略并不是太适合，所以接下来，我们又迭代了第三版。

第三个版本，基于不同的场景来动态调整消息队列的优先级，到了这里已经非常完美了，不过依然存在着任务饿死的问题，为了解决任务饿死的问题，我们给每个队列一个权重，如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样我们就完成了Chromium的任务改造。

通过整个过程的分析，我们应该能理解，在开发一个项目时，不要试图去找最完美的方案，完美的方案往往是不存在的，我们需要根据实际的场景来寻找最适合我们的方案。

思考题

我们知道CSS动画是由渲染进程自动处理的，所以渲染进程会让CSS渲染每帧动画的过程与VSync的时钟保持一致,这样就能保证CSS动画的高效率执行。

但是JavaScript是由用户控制的，如果采用setTimeout来触发动画每帧的绘制，那么其绘制时机是很难和VSync时钟保持一致的，所以JavaScript中又引入了window.requestAnimationFrame，用来和VSync的时钟周期同步，那么我留给你的问题是：你知道requestAnimationFrame回调函数的执行时机吗？

参考资料

下面是我参考的一些资料：

- Blink Scheduler
- Blink Scheduler PPT
- Chrome的消息类型
- Chrome消息优先级
- 无头浏览器

欢迎在留言区分享你的想法。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

你好，我是李兵。

我们都知道，要想利用JavaScript实现高性能的动画，那就得使用requestAnimationFrame这个API，我们简称rAF，那么为什么都推荐使用rAF而不是setTimeOut呢？

要解释清楚这个问题，就要从渲染进程的任务调度系统讲起，理解了渲染进程任务调度系统，你自然就明白了rAF和setTimeOut的区别。其次，如果你理解任务调度系统，那么你就能将渲染流水线和浏览器系统架构等知识串起来，理解了这些概念也有助于你理解Performance标签是如何工作的。

要想了解最新Chrome的任务调度系统是怎么工作的，我们得先来回顾下之前介绍的消息循环系统，我们知道了渲染进程内部的大多数任务都是在主线程上执行的，诸如JavaScript执行、DOM、CSS、计算布局、V8的垃圾回收等任务。要让这些任务能够在主线程上有条不紊地运行，就需要引入消息队列。

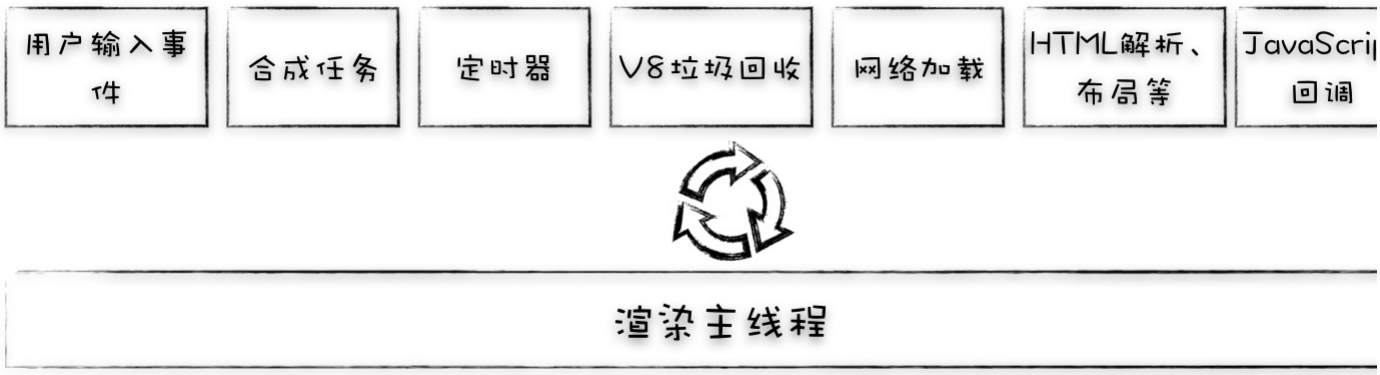
在前面的《16 | WebAPI：setTimeout是如何实现的？》这篇文章中，我们还介绍了，主线程维护了一个普通的消息队列和一个延迟消息队列，调度模块会按照规则依次取出这两个消息队列中的任务，并在主线程上执行。为了下文讲述方便，在这里我把普通的消息队列和延迟队列都当成一个消息队列。

新的任务都是被放进消息队列中去的，然后主线程再依次从消息队列中取出这些任务来顺序执行。这就是我们之前介绍的消息队列和事件循环系统。

单消息队列的队头阻塞问题

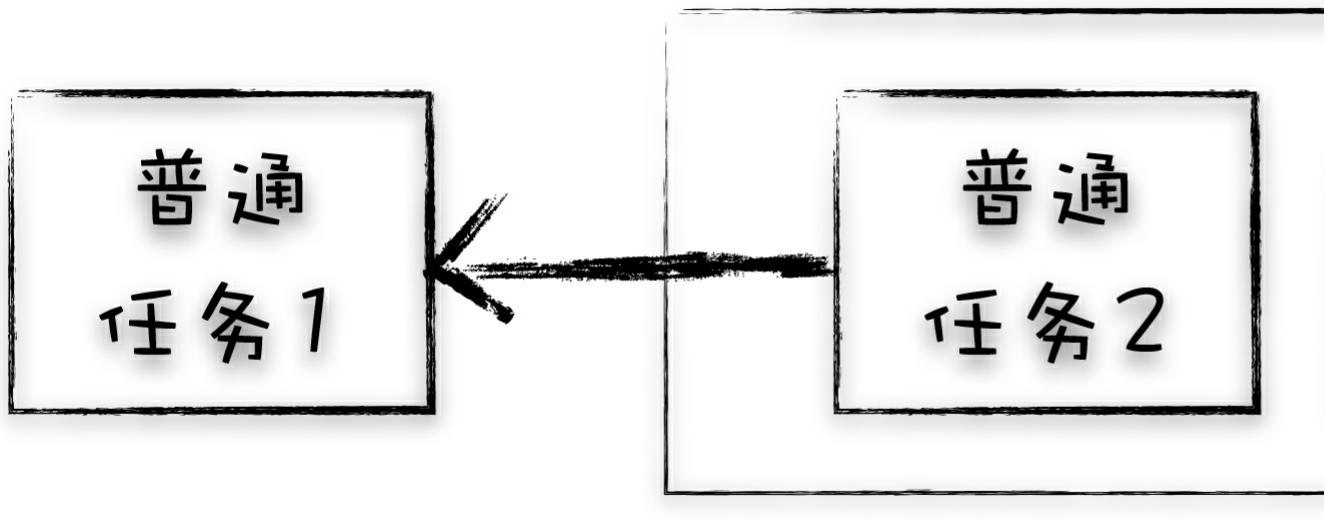
我们知道，渲染主线程会按照先进先出的顺序执行消息队列中的任务，具体地讲，当产生了新的任务，渲染进程会将其添加到消息队列尾部，在执行任务过程中，渲染进程会顺序地从消息队列头部取出任务并依次执行。

在最初，采用这种方式没有太大的问题，因为页面中的任务还不算太多，渲染主线程也不是太繁忙。不过浏览器是向前不停进化的，其进化路线体现在架构的调整、功能的增加以及更加精细的优化策略等方面，这些变化让渲染进程所需要处理的任务变多了，对应的渲染进程的主线程也变得越拥挤。下图所展示的仅仅是部分运行在主线程上的任务，你可以参考下：



任务和消息队列

你可以试想一下，在基于这种单消息队列的架构下，如果用户发出一个点击事件或者缩放页面的事件，而在此时，该任务前面可能还有很多不太重要的任务在排队等待着被执行，诸如V8的垃圾回收、DOM定时器等任务，如果执行这些任务需要花费的时间过久的话，那么就会让用户产生卡顿的感觉。你可以参看下图：



队头阻塞问题

因此，**在单消息队列架构下，存在着低优先级任务会阻塞高优先级任务的情况**，比如在一些性能不高的手机上，有时候滚动页面需要等待一秒以上。这像极了我们在介绍HTTP协议时所谈论的队头阻塞问题，那么我们也把这个问题称为消息队列的队头阻塞问题吧。

Chromium是如何解决队头阻塞问题的？

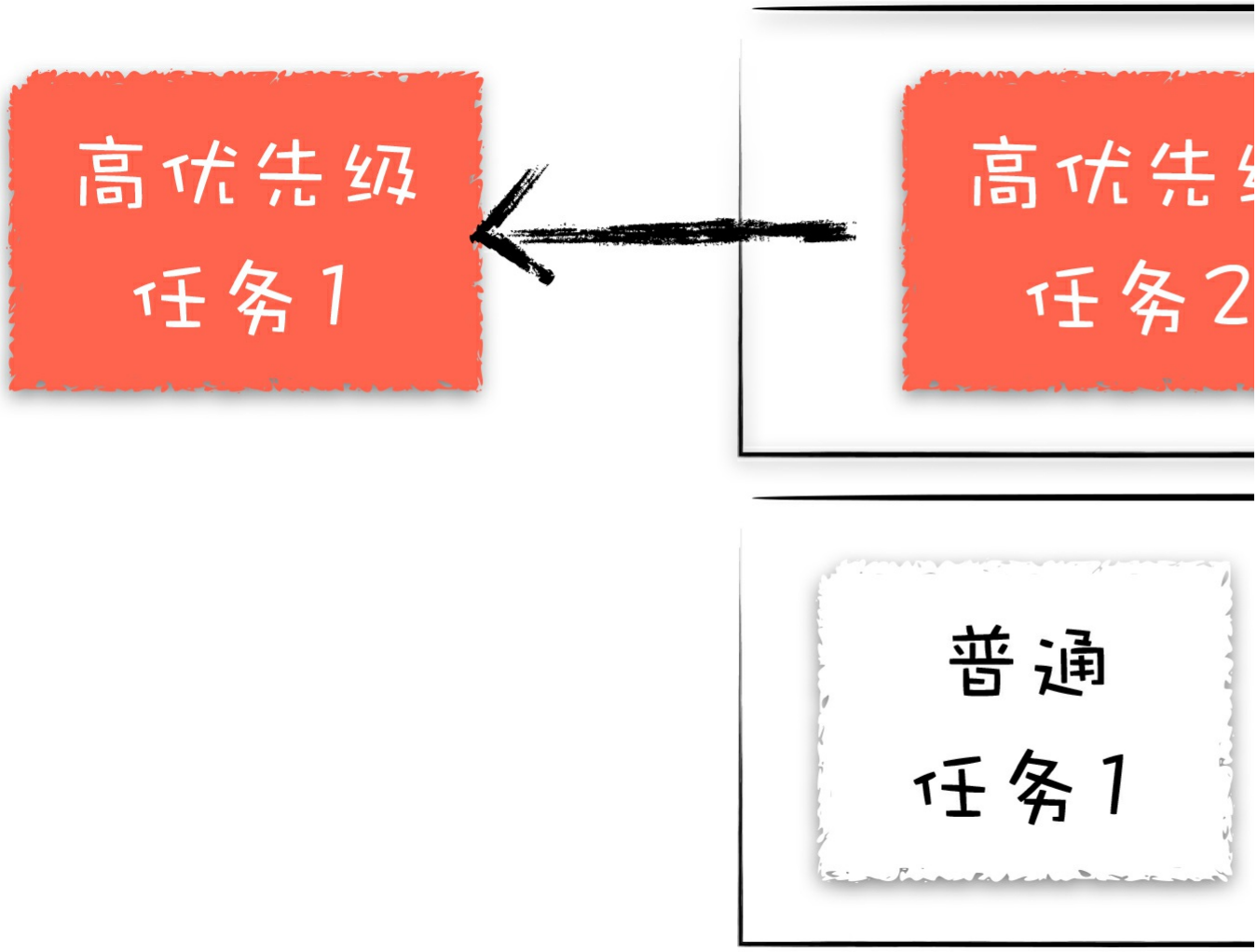
为了解决由于单消息队列而造成的队头阻塞问题，Chromium团队从2013年到现在，花了大量的精力在持续重构底层消息机制。在接下来的篇幅里，我会按照Chromium团队的重构消息系统的思路，来带你分析下他们是如何解决掉队头阻塞问题的。

1. 第一次迭代：引入一个高优先级队列

首先在最理想的情况下，我们希望能够快速跟踪高优先级任务，比如在交互阶段，下面几种任务都应该视为高优先级的任务：

- 通过鼠标触发的点击任务、滚动页面任务；
- 通过手势触发的页面缩放任务；
- 通过CSS、JavaScript等操作触发的动画特效等任务。

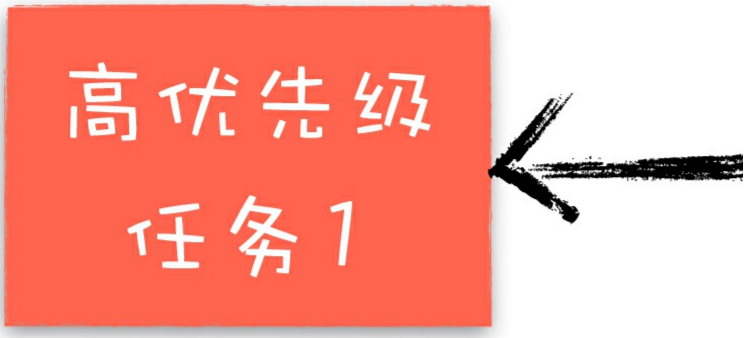
这些任务被触发后，用户想立即得到页面的反馈，所以我们需要让这些任务能够优先与其他的任务执行。要实现这种效果，我们可以增加一个高优级的消息队列，将高优先级的任务都添加到这个队列里面，然后优先执行该消息队列中的任务。最终效果如下图所示:



引入高优先级的消息队列

观察上图，我们使用了一个优先级高的消息队列和一个优先级低消息队列，渲染进程会将它认为是紧急的任务添加到高优先级队列中，不紧急的任务就添加到低优先级的队列中。然后我们再在渲染进程中引入一个**任务调度器**，负责从多个消息队列中选出合适的任务，通常实现的逻辑，先按照顺序从高优先级队列中取出任务，如果高优先级的队列为空，那么再按照顺序从低优级队列中取出任务。

我们还可以更进一步，将任务划分为多个不同的优先级，来实现更加细粒度的任务调度，比如可以划分为高优先级，普通优先级和低优先级，最终效果如下图所示：



增加多个不同优先级的消息队列

观察上图，我们实现了三个不同优先级的消息队列，然后可以使用任务调度器来统一调度这三个不同消息队列中的任务。

好了，现在我们引入了多个消息队列，结合任务调度器我们就可以灵活地调度任务了，这样我们就可以让高优先级的任务提前执行，采用这种方式似乎解决了消息队列的队头阻塞问题。

不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，甚至有可能出现还未处理输入事件，就合成了该事件要显示的图片。因此我们需要让一些相同类型的任务保持其相对执行顺序。

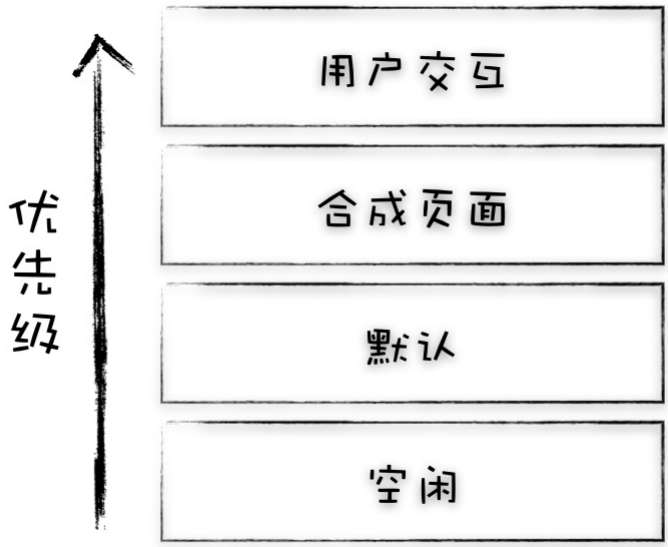
2. 第二次迭代：根据消息类型来实现消息队列

要解决上述问题，我们可以为不同类型的任务创建不同优先级的消息队列，比如：

- 可以创建输入事件的消息队列，用来存放输入事件。

- 可以创建合成任务的消息队列，用来存放合成事件。
- 可以创建默认消息队列，用来保存如资源加载的事件和定时器回调等事件。
- 还可以创建一个空闲消息队列，用来存放V8的垃圾自动垃圾回收这一类实时性不高的事件。

最终实现效果如下图所示：



根据消息类型实现不同优先级的消息队列

通过迭代，这种策略已经相当实用了，但是它依然存在着问题，那就是这几种消息队列的优先级都是固定的，任务调度器会按照这种固定好的静态的优先级来分别调度任务。那么静态优先级会带来什么问题呢？

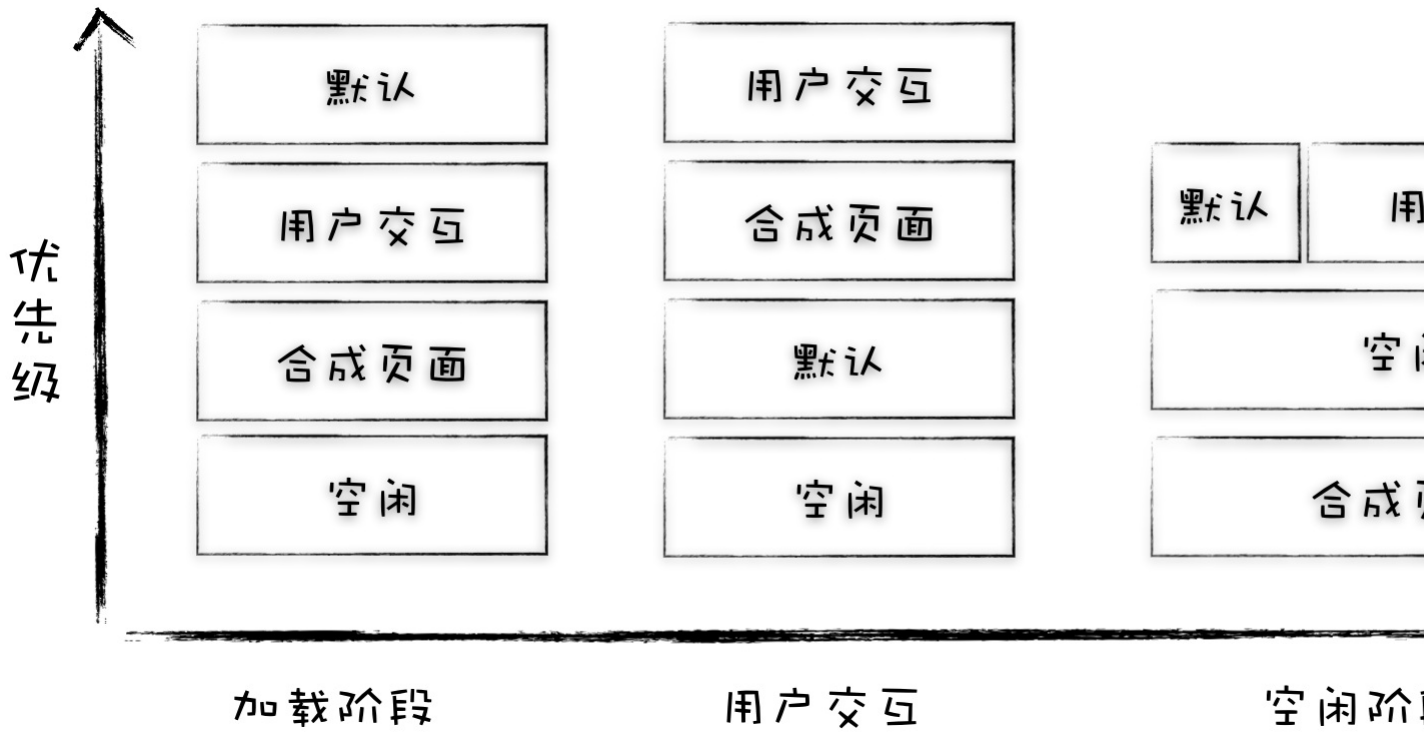
我们在《25 | 页面性能：如何系统地优化页面？》这节分析过页面的生存周期，页面大致的生存周期大体分为两个阶段，加载阶段和交互阶段。

虽然在交互阶段，采用上述这种静态优先级的策略没有什么太大问题的，但是在页面加载阶段，如果依然要优先执行用户输入事件和合成事件，那么页面的解析速度将会被拖慢。Chromium团队曾测试过这种情况，使用静态优先级策略，网页的加载速度会被拖慢14%。

3. 第三次迭代：动态调度策略

可以看出，我们所采用的优化策略像个跷跷板，虽然优化了高优先级任务，却拖慢低优先级任务，之所以会这样，是因为我们采取了静态的任务调度策略，对于各种不同的场景，这种静态策略就显得过于死板。

所以我们还得根据实际场景来继续平衡这个跷跷板，也就是说在不同的场景下，根据实际情况，动态调整消息队列的优先级。一图胜过千言，我们先看下图：



动态调度策略

这张图展示了Chromium在不同的场景下，是如何调整消息队列优先级的。通过这种动态调度策略，就可以满足不同场景的核心诉求了，同时这也是Chromium当前所采用的任务调度策略。

上图列出了三个不同的场景，分别是加载过程，合成过程以及正常状态。下面我们就结合这三种场景，来分析下Chromium为何做这种调整。

首先我们来看看**页面加载阶段**的场景，在这个阶段，用户的最高诉求是在尽可能短的时间内看到页面，至于交互和合成并不是这个阶段的核心诉求，因此我们需要调整策略，在加载阶段将页面解析，JavaScript脚本执行等任务调整为优先级最高的队列，降低交互合成这些队列的优先级。

页面加载完成之后就进入了**交互阶段**，在介绍Chromium是如何调整交互阶段的任务调度策略之前，我们还需要岔开一下，来回顾下页面的渲染过程。

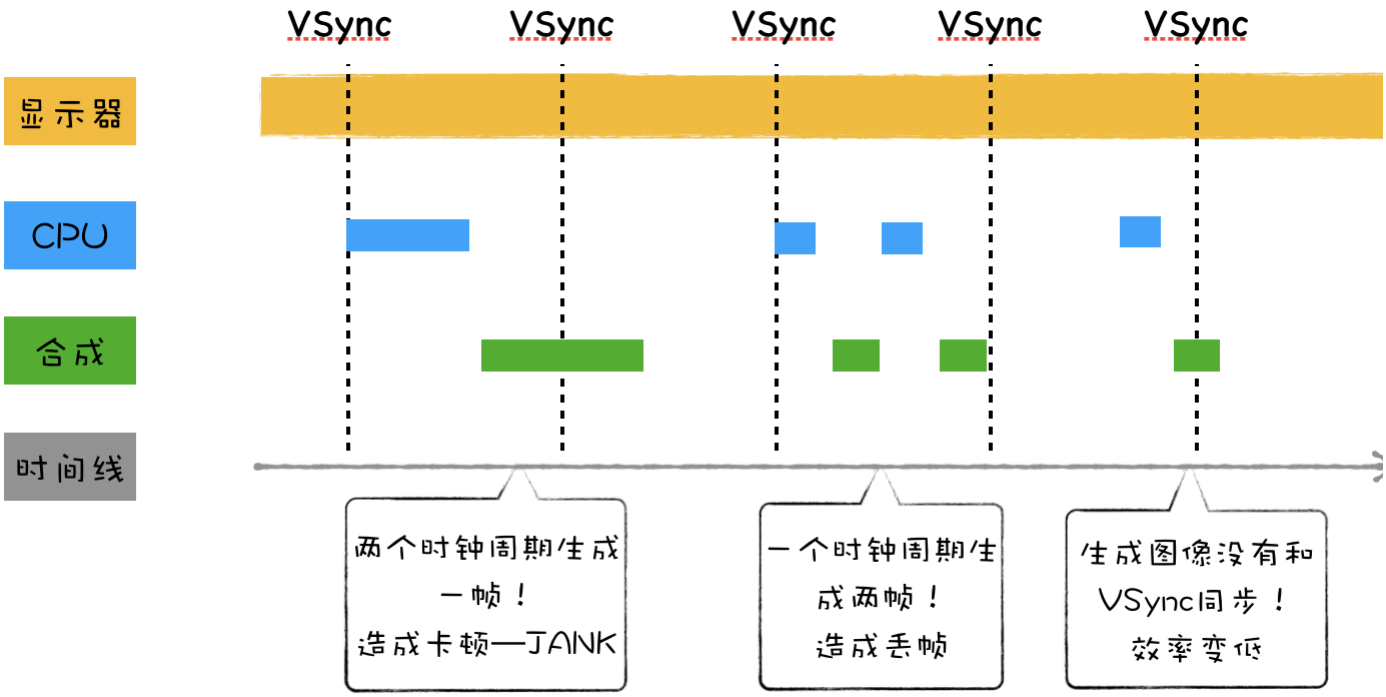
在《06 | 渲染流程（下）：HTML、CSS和JavaScript，是如何变成页面的？》和《24 | 分层和合成机制：为什么CSS动画比JavaScript高效？》这两节，我们分析了一个页面是如何渲染并显示出来的。

在显卡中有一块叫着**前缓冲区**的地方，这里存放着显示器要显示的图像，显示器会按照一定的频率来读取这块前缓冲区，并将前缓冲区中的图像显示在显示器上，不同的显示器读取的频率是不同的，通

常情况下是60HZ，也就是说显示器会每间隔1/60秒就读取一次前缓冲区。

如果浏览器要更新显示的图片，那么浏览器会将新生成的图片提交到显卡的**后缓冲区**中，提交完成之后，GPU会将**后缓冲区和前缓冲区互换位置**，也就是前缓冲区变成了后缓冲区，后缓冲区变成了前缓冲区，这就保证了显示器下次能读取到GPU中最新的图片。

这时候我们会发现，显示器从前缓冲区读取图片，和浏览器生成新的图像到后缓冲区的过程是不同步的，如下图所示：



VSync时钟周期和渲染引擎生成图片不同步问题

这种显示器读取图片和浏览器生成图片不同步，容易造成众多问题。

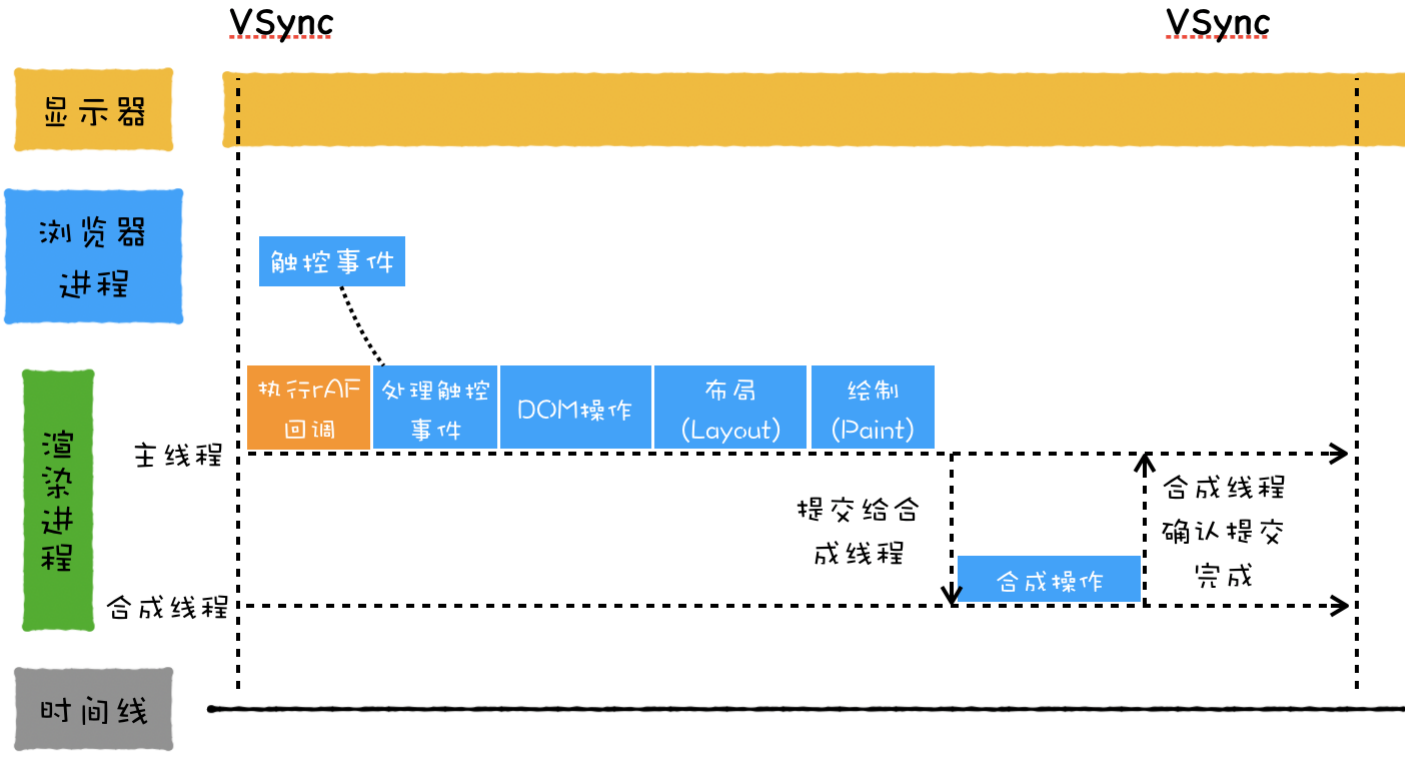
- 如果渲染进程生成的帧速比屏幕的刷新率慢，那么屏幕会在两帧中显示同一个画面，当这种断断续续的情况持续发生时，用户将会很明显地察觉到动画卡住了。
- 如果渲染进程生成的帧速率实际上比屏幕刷新率快，那么也会出现一些视觉上的问题，比如当帧速率在100fps而刷新率只有60Hz的时候，GPU所渲染的图像并非全都被显示出来，这就会造成丢帧现象。
- 就算屏幕的刷新频率和GPU更新图片的频率一样，由于它们是两个不同的系统，所以屏幕生成帧的周期和VSync的周期也是很难同步起来的。

所以VSync和系统的时钟不同步就会造成掉帧、卡顿、不连贯等问题。

为了解决这些问题，就需要将显示器的时钟同步周期和浏览器生成页面的周期绑定起来，Chromium也是这样实现，那么下面我们就来看看Chromium具体是怎么实现的？

当显示器将一帧画面绘制完成后，并在准备读取下一帧之前，显示器会发出一个垂直同步信号（vertical synchronization）给GPU，简称 VSync。这时候浏览器就会充分利用好VSync信号。

具体地讲，当GPU接收到VSync信号后，会将VSync信号同步给浏览器进程，浏览器进程再将其同步到对应的渲染进程，渲染进程接收到VSync信号之后，就可以准备绘制新的一帧了，具体流程你可以参考下图：



绑定VSync时钟同步周期和浏览器生成页面周期

上面其实是非常粗略的介绍，实际实现过程也是非常复杂的，如果感兴趣，你可以参考这篇文章。

好了，我们花了很大篇幅介绍了VSync和页面中的一帧是怎么显示出来，有了这些知识，我们就可以回到主线了，来分析下渲染进程是如何优化交互阶段页面的任务调度策略的？

从上图可以看出，当渲染进程接收到用户交互的任务后，接下来大概率是要进行绘制合成操作，因此我们可以设置，**当在执行用户交互的任务时，将合成任务的优先级调整到最高。**

接下来，处理完成DOM，计算好布局和绘制，就需要将信息提交给合成线程来合成最终图片了，然后合成线程进入工作状态。现在的场景是合成线程在工作了，**那么我们就可以把下个合成任务的优先级调整为最低，并将页面解析、定时器等任务优先级提升。**

在合成完成之后，合成线程会提交给渲染主线程提交完成合成的消息，如果当前合成操作执行的非常快，比如从用户发出消息到完成合成操作只花了8毫秒，因为VSync同步周期是16.66（1/60）毫秒，那么这个VSync时钟周期内就不需要再次生成新的页面了。那么从合成结束到下个VSync周期内，就进入了一个空闲时间阶段，那么就可以在这段空闲时间内执行一些不那么紧急的任务，比如V8的垃圾回收，或者通过window.requestIdleCallback()设置的回调任务等，都会在这段空闲时间内执行。

4. 第四次迭代：任务饿死

好了，以上方案看上去似乎非常完美了，不过依然存在一个问题，那就是在某个状态下，一直有新的高优先级的任务加入到队列中，这样就会导致其他低优先级的任务得不到执行，这称为任务饿死。

Chromium为了解决任务饿死的问题，给每个队列设置了执行权重，也就是如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样就缓解了任务饿死的情况。

总结

好了，本节的内容就介绍到这里，下面我来总结下本文的主要内容：

首先我们分析了基于单消息队列会引起队头阻塞的问题，为了解决队头阻塞问题，我们引入了多个不同优级的消息队列，并将紧急的任务添加到高优先级队列，不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，所以我们又迭代了第二个版本。

在第二个版本中，按照不同的任务类型来划分任务优先级，不过由于采用的静态优先级策略，对于其他一些场景，这种静态调度的策略并不是太适合，所以接下来，我们又迭代了第三版。

第三个版本，基于不同的场景来动态调整消息队列的优先级，到了这里已经非常完美了，不过依然存在着任务饿死的问题，为了解决任务饿死的问题，我们给每个队列一个权重，如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样我们就完成了Chromium的任务改造。

通过整个过程的分析，我们应该能理解，在开发一个项目时，不要试图去找最完美的方案，完美的方案往往是不存在的，我们需要根据实际的场景来寻找最适合我们的方案。

思考题

我们知道CSS动画是由渲染进程自动处理的，所以渲染进程会让CSS渲染每帧动画的过程与VSync的时钟保持一致,这样就能保证CSS动画的高效率执行。

但是JavaScript是由用户控制的，如果采用setTimeout来触发动画每帧的绘制，那么其绘制时机是很难和VSync时钟保持一致的，所以JavaScript中又引入了window.requestAnimationFrame，用来和VSync的时钟周期同步，那么我留给你的问题是：你知道requestAnimationFrame回调函数的执行时机吗？

参考资料

下面是我参考的一些资料：

- Blink Scheduler
- Blink Scheduler PPT
- Chrome的消息类型
- Chrome消息优先级
- 无头浏览器

欢迎在留言区分享你的想法。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

你好，我是李兵。

我们都知道，要想利用JavaScript实现高性能的动画，那就得使用requestAnimationFrame这个API，我们简称rAF，那么为什么都推荐使用rAF而不是setTimeOut呢？

要解释清楚这个问题，就要从渲染进程的任务调度系统讲起，理解了渲染进程任务调度系统，你自然就明白了rAF和setTimeOut的区别。其次，如果你理解任务调度系统，那么你就能将渲染流水线和浏览器系统架构等知识串起来，理解了这些概念也有助于你理解Performance标签是如何工作的。

要想了解最新Chrome的任务调度系统是怎么工作的，我们得先来回顾下之前介绍的消息循环系统，我们知道了渲染进程内部的大多数任务都是在主线程上执行的，诸如JavaScript执行、DOM、CSS、计算布局、V8的垃圾回收等任务。要让这些任务能够在主线程上有条不紊地运行，就需要引入消息队列。

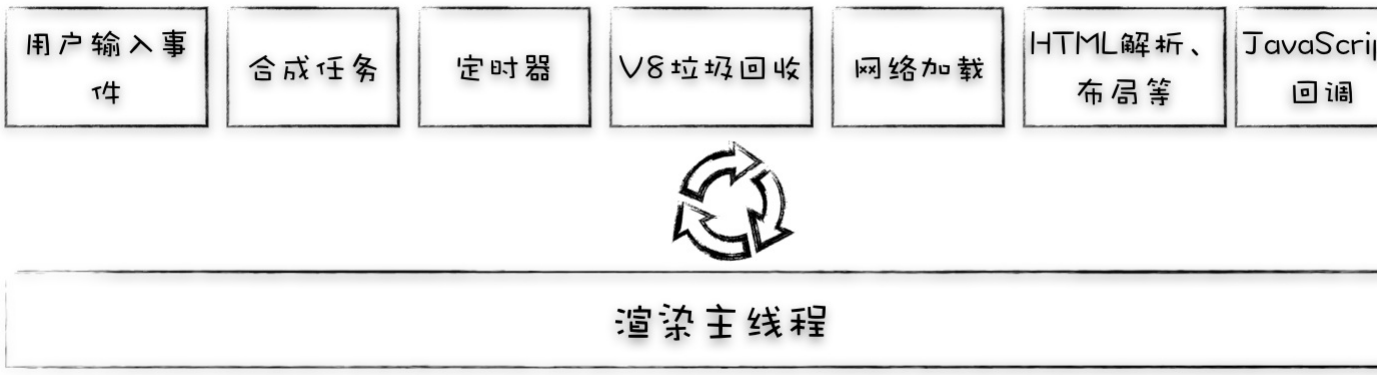
在前面的《16 | WebAPI：setTimeout是如何实现的？》这篇文章中，我们还介绍了，主线程维护了一个普通的消息队列和一个延迟消息队列，调度模块会按照规则依次取出这两个消息队列中的任务，并在主线程上执行。为了下文讲述方便，在这里我把普通的消息队列和延迟队列都当成一个消息队列。

新的任务都是被放进消息队列中去的，然后主线程再依次从消息队列中取出这些任务来顺序执行。这就是我们之前介绍的消息队列和事件循环系统。

单消息队列的队头阻塞问题

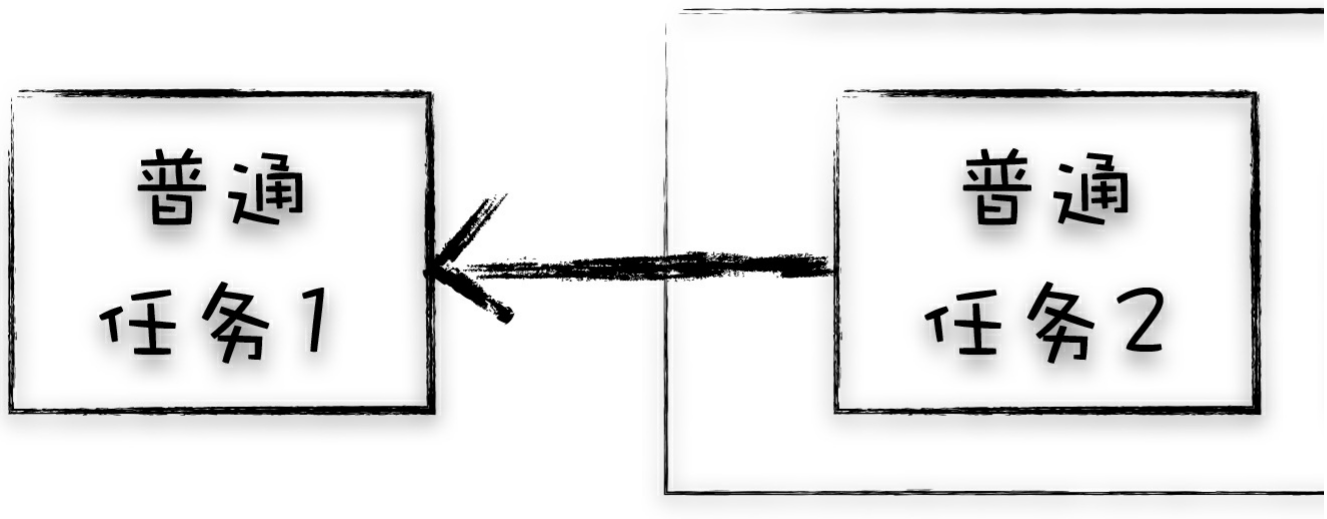
我们知道，渲染主线程会按照先进先出的顺序执行消息队列中的任务，具体地讲，当产生了新的任务，渲染进程会将其添加到消息队列尾部，在执行任务过程中，渲染进程会顺序地从消息队列头部取出任务并依次执行。

在最初，采用这种方式没有太大的问题，因为页面中的任务还不算太多，渲染主线程也不是太繁忙。不过浏览器是向前不停进化的，其进化路线体现在架构的调整、功能的增加以及更加精细的优化策略等方面，这些变化让渲染进程所需要处理的任务变多了，对应的渲染进程的主线程也变得越拥挤。下图所展示的仅仅是部分运行在主线程上的任务，你可以参考下：



任务和消息队列

你可以试想一下，在基于这种单消息队列的架构下，如果用户发出一个点击事件或者缩放页面的事件，而在此时，该任务前面可能还有很多不太重要的任务在排队等待着被执行，诸如V8的垃圾回收、DOM定时器等任务，如果执行这些任务需要花费的时间过久的话，那么就会让用户产生卡顿的感觉。你可以参看下图：



队头阻塞问题

因此，**在单消息队列架构下，存在着低优先级任务会阻塞高优先级任务的情况**，比如在一些性能不高的手机上，有时候滚动页面需要等待一秒以上。这像极了我们在介绍HTTP协议时所谈论的队头阻塞问题，那么我们也把这个问题称为消息队列的队头阻塞问题吧。

Chromium是如何解决队头阻塞问题的？

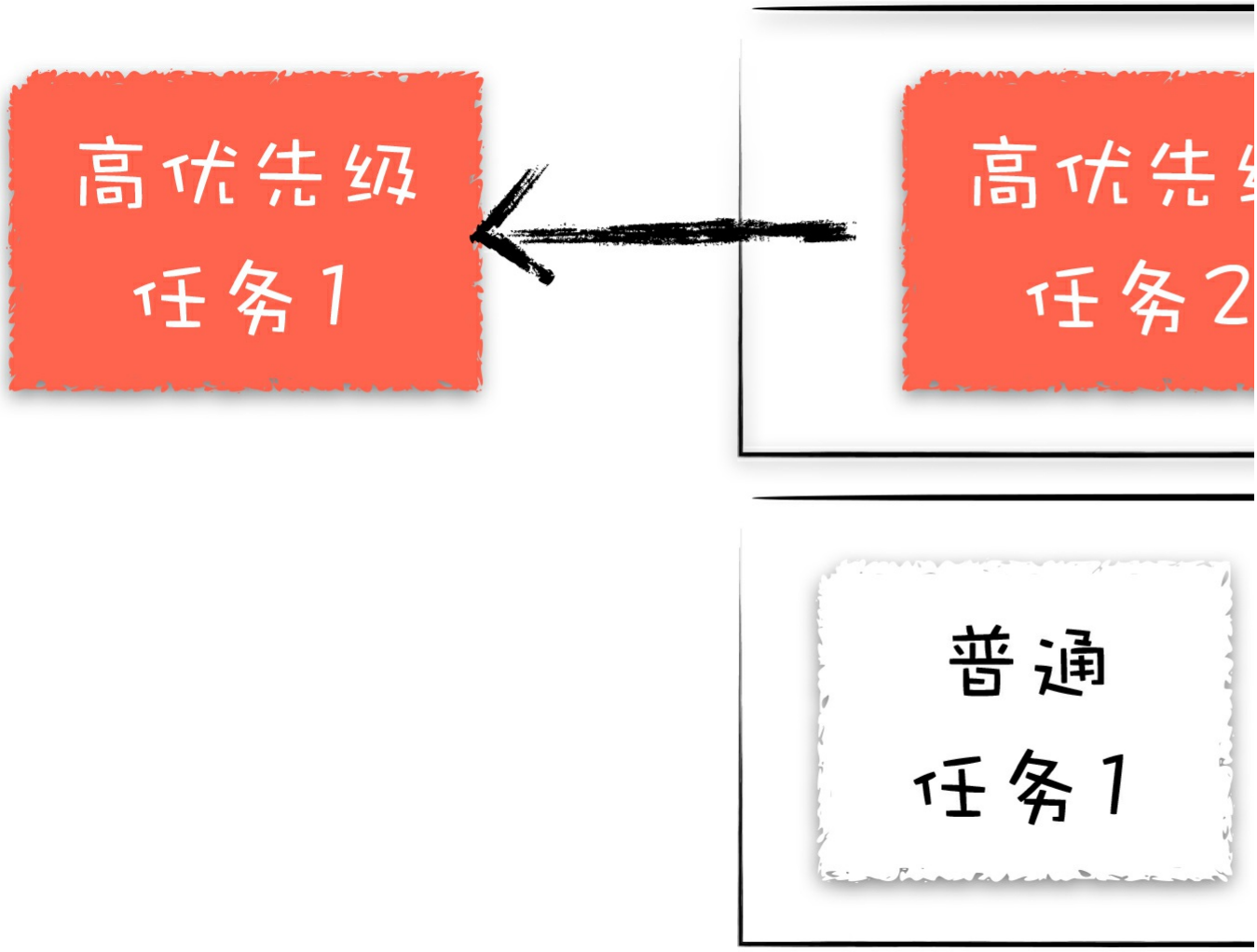
为了解决由于单消息队列而造成的队头阻塞问题，Chromium团队从2013年到现在，花了大量的精力在持续重构底层消息机制。在接下来的篇幅里，我会按照Chromium团队的重构消息系统的思路，来带你分析下他们是如何解决掉队头阻塞问题的。

1. 第一次迭代：引入一个高优先级队列

首先在最理想的情况下，我们希望能够快速跟踪高优先级任务，比如在交互阶段，下面几种任务都应该视为高优先级的任务：

- 通过鼠标触发的点击任务、滚动页面任务；
- 通过手势触发的页面缩放任务；
- 通过CSS、JavaScript等操作触发的动画特效等任务。

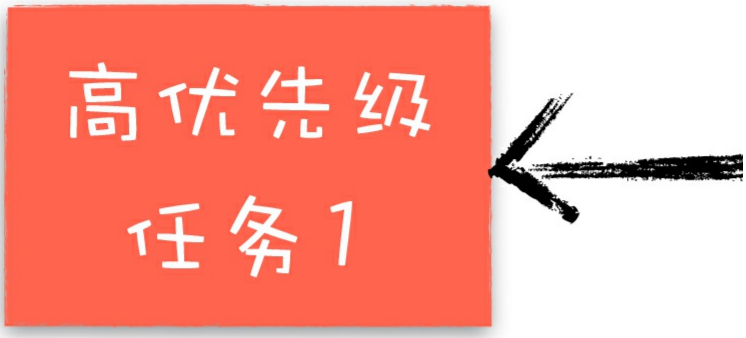
这些任务被触发后，用户想立即得到页面的反馈，所以我们需要让这些任务能够优先与其他的任务执行。要实现这种效果，我们可以增加一个高优级的消息队列，将高优先级的任务都添加到这个队列里面，然后优先执行该消息队列中的任务。最终效果如下图所示:



引入高优先级的消息队列

观察上图，我们使用了一个优先级高的消息队列和一个优先级低消息队列，渲染进程会将它认为是紧急的任务添加到高优先级队列中，不紧急的任务就添加到低优先级的队列中。然后我们再在渲染进程中引入一个**任务调度器**，负责从多个消息队列中选出合适的任务，通常实现的逻辑，先按照顺序从高优先级队列中取出任务，如果高优先级的队列为空，那么再按照顺序从低优级队列中取出任务。

我们还可以更进一步，将任务划分为多个不同的优先级，来实现更加细粒度的任务调度，比如可以划分为高优先级，普通优先级和低优先级，最终效果如下图所示：



增加多个不同优先级的消息队列

观察上图，我们实现了三个不同优先级的消息队列，然后可以使用任务调度器来统一调度这三个不同消息队列中的任务。

好了，现在我们引入了多个消息队列，结合任务调度器我们就可以灵活地调度任务了，这样我们就可以让高优先级的任务提前执行，采用这种方式似乎解决了消息队列的队头阻塞问题。

不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，甚至有可能出现还未处理输入事件，就合成了该事件要显示的图片。因此我们需要让一些相同类型的任务保持其相对执行顺序。

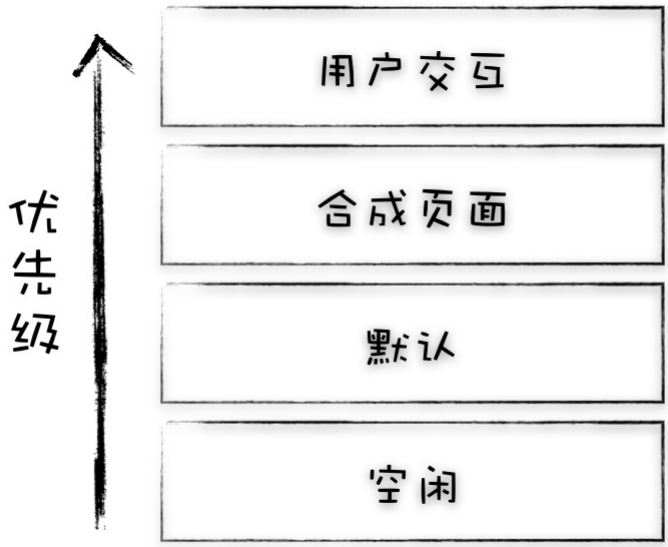
2. 第二次迭代：根据消息类型来实现消息队列

要解决上述问题，我们可以为不同类型的任务创建不同优先级的消息队列，比如：

- 可以创建输入事件的消息队列，用来存放输入事件。

- 可以创建合成任务的消息队列，用来存放合成事件。
- 可以创建默认消息队列，用来保存如资源加载的事件和定时器回调等事件。
- 还可以创建一个空闲消息队列，用来存放V8的垃圾自动垃圾回收这一类实时性不高的事件。

最终实现效果如下图所示：



根据消息类型实现不同优先级的消息队列

通过迭代，这种策略已经相当实用了，但是它依然存在着问题，那就是这几种消息队列的优先级都是固定的，任务调度器会按照这种固定好的静态的优先级来分别调度任务。那么静态优先级会带来什么问题呢？

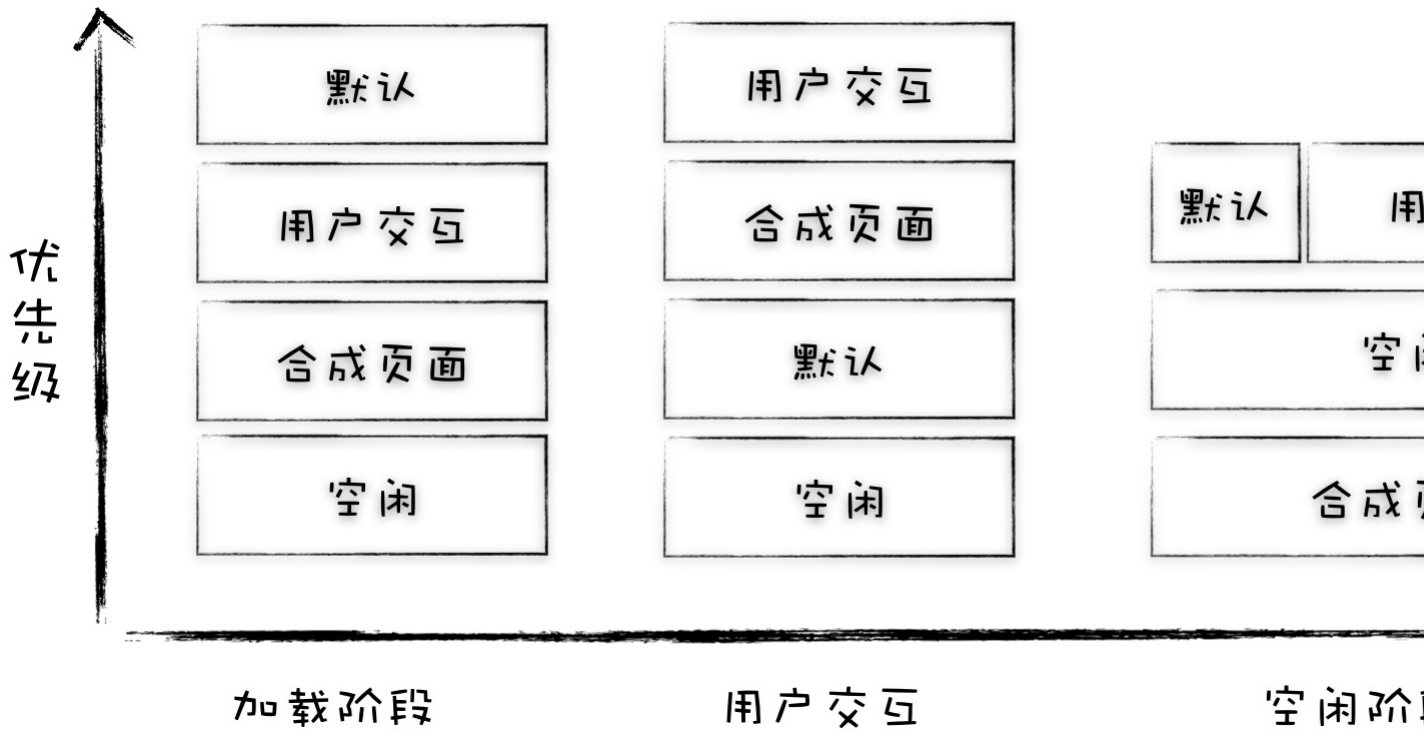
我们在《25 | 页面性能：如何系统地优化页面？》这节分析过页面的生存周期，页面大致的生存周期大体分为两个阶段，加载阶段和交互阶段。

虽然在交互阶段，采用上述这种静态优先级的策略没有什么太大问题的，但是在页面加载阶段，如果依然要优先执行用户输入事件和合成事件，那么页面的解析速度将会被拖慢。Chromium团队曾测试过这种情况，使用静态优先级策略，网页的加载速度会被拖慢14%。

3. 第三次迭代：动态调度策略

可以看出，我们所采用的优化策略像个跷跷板，虽然优化了高优先级任务，却拖慢低优先级任务，之所以会这样，是因为我们采取了静态的任务调度策略，对于各种不同的场景，这种静态策略就显得过于死板。

所以我们还得根据实际场景来继续平衡这个跷跷板，也就是说在不同的场景下，根据实际情况，动态调整消息队列的优先级。一图胜过千言，我们先看下图：



动态调度策略

这张图展示了Chromium在不同的场景下，是如何调整消息队列优先级的。通过这种动态调度策略，就可以满足不同场景的核心诉求了，同时这也是Chromium当前所采用的任务调度策略。

上图列出了三个不同的场景，分别是加载过程，合成过程以及正常状态。下面我们就结合这三种场景，来分析下Chromium为何做这种调整。

首先我们来看看**页面加载阶段**的场景，在这个阶段，用户的最高诉求是在尽可能短的时间内看到页面，至于交互和合成并不是这个阶段的核心诉求，因此我们需要调整策略，在加载阶段将页面解析，JavaScript脚本执行等任务调整为优先级最高的队列，降低交互合成这些队列的优先级。

页面加载完成之后就进入了**交互阶段**，在介绍Chromium是如何调整交互阶段的任务调度策略之前，我们还需要岔开一下，来回顾下页面的渲染过程。

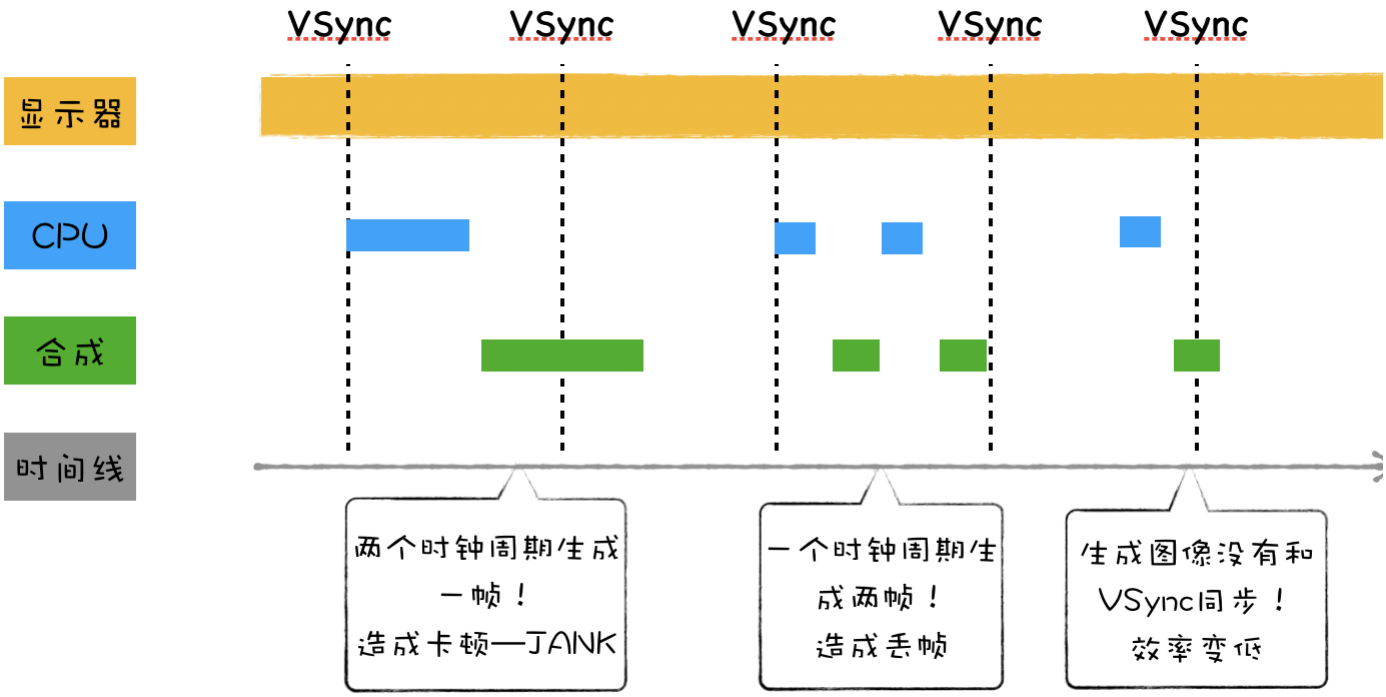
在《06 | 渲染流程（下）：HTML、CSS和JavaScript，是如何变成页面的？》和《24 | 分层和合成机制：为什么CSS动画比JavaScript高效？》这两节，我们分析了一个页面是如何渲染并显示出来的。

在显卡中有一块叫着**前缓冲区**的地方，这里存放着显示器要显示的图像，显示器会按照一定的频率来读取这块前缓冲区，并将前缓冲区中的图像显示在显示器上，不同的显示器读取的频率是不同的，通

常情况下是60HZ，也就是说显示器会每间隔1/60秒就读取一次前缓冲区。

如果浏览器要更新显示的图片，那么浏览器会将新生成的图片提交到显卡的**后缓冲区**中，提交完成之后，GPU会将**后缓冲区和前缓冲区互换位置**，也就是前缓冲区变成了后缓冲区，后缓冲区变成了前缓冲区，这就保证了显示器下次能读取到GPU中最新的图片。

这时候我们会发现，显示器从前缓冲区读取图片，和浏览器生成新的图像到后缓冲区的过程是不同步的，如下图所示：



VSync时钟周期和渲染引擎生成图片不同步问题

这种显示器读取图片和浏览器生成图片不同步，容易造成众多问题。

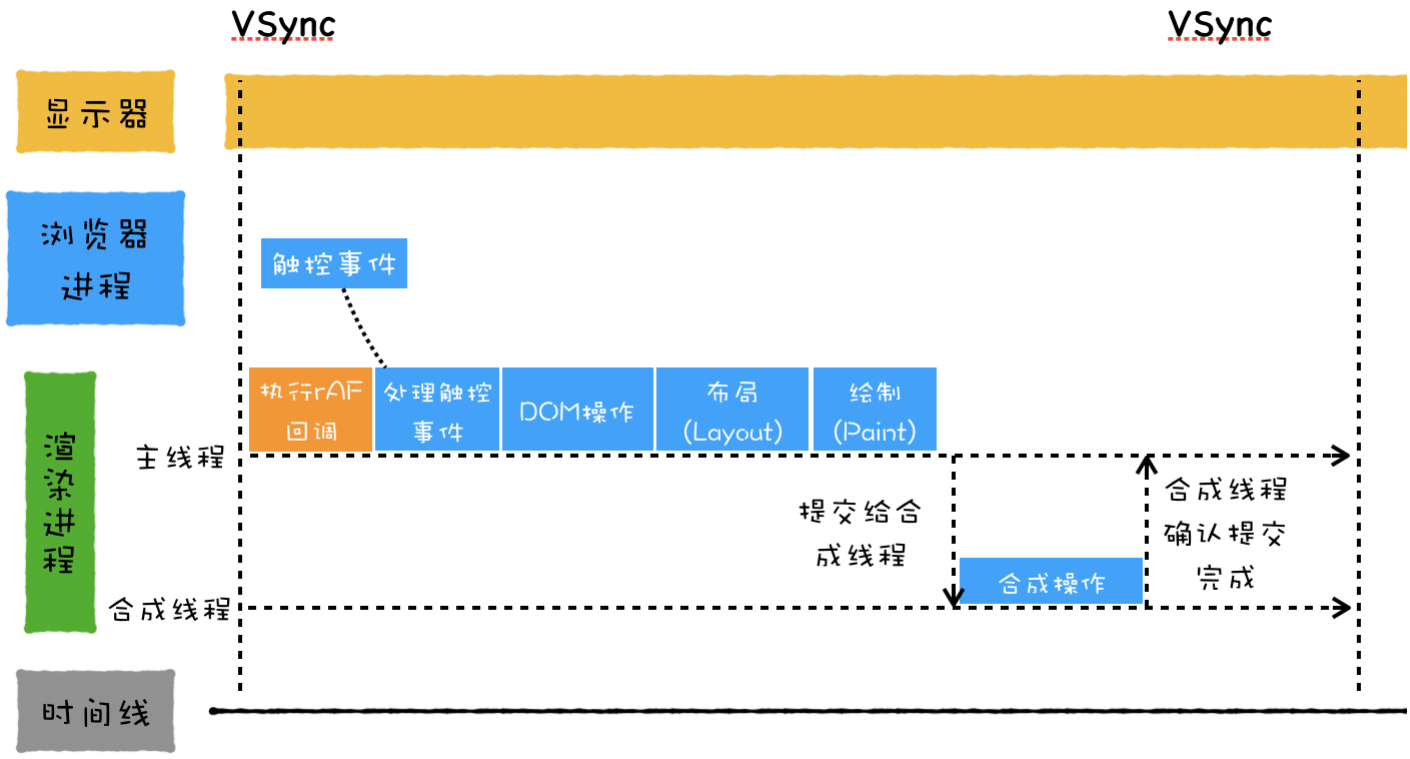
- 如果渲染进程生成的帧速比屏幕的刷新率慢，那么屏幕会在两帧中显示同一个画面，当这种断断续续的情况持续发生时，用户将会很明显地察觉到动画卡住了。
- 如果渲染进程生成的帧速率实际上比屏幕刷新率快，那么也会出现一些视觉上的问题，比如当帧速率在100fps而刷新率只有60Hz的时候，GPU所渲染的图像并非全都被显示出来，这就会造成丢帧现象。
- 就算屏幕的刷新频率和GPU更新图片的频率一样，由于它们是两个不同的系统，所以屏幕生成帧的周期和VSync的周期也是很难同步起来的。

所以VSync和系统的时钟不同步就会造成掉帧、卡顿、不连贯等问题。

为了解决这些问题，就需要将显示器的时钟同步周期和浏览器生成页面的周期绑定起来，Chromium也是这样实现，那么下面我们就来看看Chromium具体是怎么实现的？

当显示器将一帧画面绘制完成后，并在准备读取下一帧之前，显示器会发出一个垂直同步信号（vertical synchronization）给GPU，简称 VSync。这时候浏览器就会充分利用好VSync信号。

具体地讲，当GPU接收到VSync信号后，会将VSync信号同步给浏览器进程，浏览器进程再将其同步到对应的渲染进程，渲染进程接收到VSync信号之后，就可以准备绘制新的一帧了，具体流程你可以参考下图：



绑定VSync时钟同步周期和浏览器生成页面周期

上面其实是非常粗略的介绍，实际实现过程也是非常复杂的，如果感兴趣，你可以参考这篇文章。

好了，我们花了很大篇幅介绍了VSync和页面中的一帧是怎么显示出来，有了这些知识，我们就可以回到主线了，来分析下渲染进程是如何优化交互阶段页面的任务调度策略的？

从上图可以看出，当渲染进程接收到用户交互的任务后，接下来大概率是要进行绘制合成操作，因此我们可以设置，**当在执行用户交互的任务时，将合成任务的优先级调整到最高。**

接下来，处理完成DOM，计算好布局和绘制，就需要将信息提交给合成线程来合成最终图片了，然后合成线程进入工作状态。现在的场景是合成线程在工作了，**那么我们就可以把下个合成任务的优先级调整为最低，并将页面解析、定时器等任务优先级提升。**

在合成完成之后，合成线程会提交给渲染主线程提交完成合成的消息，如果当前合成操作执行的非常快，比如从用户发出消息到完成合成操作只花了8毫秒，因为VSync同步周期是16.66（1/60）毫秒，那么这个VSync时钟周期内就不需要再次生成新的页面了。那么从合成结束到下个VSync周期内，就进入了一个空闲时间阶段，那么就可以在这段空闲时间内执行一些不那么紧急的任务，比如V8的垃圾回收，或者通过window.requestIdleCallback()设置的回调任务等，都会在这段空闲时间内执行。

4. 第四次迭代：任务饿死

好了，以上方案看上去似乎非常完美了，不过依然存在一个问题，那就是在某个状态下，一直有新的高优先级的任务加入到队列中，这样就会导致其他低优先级的任务得不到执行，这称为任务饿死。

Chromium为了解决任务饿死的问题，给每个队列设置了执行权重，也就是如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样就缓解了任务饿死的情况。

总结

好了，本节的内容就介绍到这里，下面我来总结下本文的主要内容：

首先我们分析了基于单消息队列会引起队头阻塞的问题，为了解决队头阻塞问题，我们引入了多个不同优级的消息队列，并将紧急的任务添加到高优先级队列，不过大多数任务需要保持其相对执行顺序，如果将用户输入的消息或者合成消息添加进多个不同优先级的队列中，那么这种任务的相对执行顺序就会被打乱，所以我们又迭代了第二个版本。

在第二个版本中，按照不同的任务类型来划分任务优先级，不过由于采用的静态优先级策略，对于其他一些场景，这种静态调度的策略并不是太适合，所以接下来，我们又迭代了第三版。

第三个版本，基于不同的场景来动态调整消息队列的优先级，到了这里已经非常完美了，不过依然存在着任务饿死的问题，为了解决任务饿死的问题，我们给每个队列一个权重，如果连续执行了一定个数的高优先级的任务，那么中间会执行一次低优先级的任务，这样我们就完成了Chromium的任务改造。

通过整个过程的分析，我们应该能理解，在开发一个项目时，不要试图去找最完美的方案，完美的方案往往是不存在的，我们需要根据实际的场景来寻找最适合我们的方案。

思考题

我们知道CSS动画是由渲染进程自动处理的，所以渲染进程会让CSS渲染每帧动画的过程与VSync的时钟保持一致,这样就能保证CSS动画的高效率执行。

但是JavaScript是由用户控制的，如果采用setTimeout来触发动画每帧的绘制，那么其绘制时机是很难和VSync时钟保持一致的，所以JavaScript中又引入了window.requestAnimationFrame，用来和VSync的时钟周期同步，那么我留给你的问题是：你知道requestAnimationFrame回调函数的执行时机吗？

参考资料

下面是我参考的一些资料：

- Blink Scheduler
- Blink Scheduler PPT
- Chrome的消息类型
- Chrome消息优先级
- 无头浏览器

欢迎在留言区分享你的想法。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。