时间切分api比较

前置

console.log

1. console.log也就是process.stdout.write的执行时间也是要考虑的问题。
2. 为了使测试结果更准确，不要在执行过程中打印，使用数组存储过程中的记录，最后把数组打印出来。

统计时间

用performance.now()而不用Date.now()：

1. performance.now()返回当前页面的停留时间，Date.now()返回当前系统时间。但不同的是performance.now()精度更高，且比Date.now()更可靠。

2. performance.now()返回的是微秒级的，Date.now()只是毫秒级。
3. performance.now()一个恒定的速率慢慢增加的，它不会受到系统时间的影响。Date.now()受到系统时间影响，系统时间修改Date.now()也会改变。

task分割为什么不是微任务

微任务**无法真正达到**交还主线程控制权的要求。

因为一轮事件循环，是先执行一个宏任务，然后再清空微任务队列里面的任务，如果在清空微任务队列的过程中，依然有新任务插入到微任务队列中的话，还是把这些任务执行完毕才会释放主线程。所以微任务不合适。

拆分task 的api

既然浏览器可以自由调度的最小task是宏任务，那我们只需要将同步执行的代码使用宏任务拆分即可：

我们书写一个 yieldToMain 方法，其内部是使用setTimeout异步api来做到task拆分的作用，利用await/async 将此方法await 之后的代码推入到下一个事件循环中去。

```JavaScript
function yieldToMain () {
  return new Promise(resolve => {
    setTimeout(resolve, 0);
  });
}

while (tasks.length > 0) {
    if (navigator.scheduling.isInputPending()) {
      await yieldToMain();
    } else {
      const task = tasks.shift();
      task();
    }
  }
}
```

那我的异步宏任务api有

- setImmediate
- MessageChannel
- requestAnimationFrame

- requestIdleCallback
- setInterval
- setTimeout
- ...

以下我就逐个对各个api进行详细的讲解：

setImmediate

> 只在少量环境（比如 IE 的低版本、Node.js）可以使用

回调将在当前事件循环中的任何I/O操作之后以及为下一个事件循环安排的任何计时器之前执行。

所以setInmmediate应该在setTimeout之前执行。当然文档也说了在没有I/O的情况下，执行顺序是不确定的。

对于react 的调度器其：会优先使用 setImmediate，但它只在少量环境中存在。

MessageChannel

执行时机比setTimeout靠前。

requestAnimationFrame

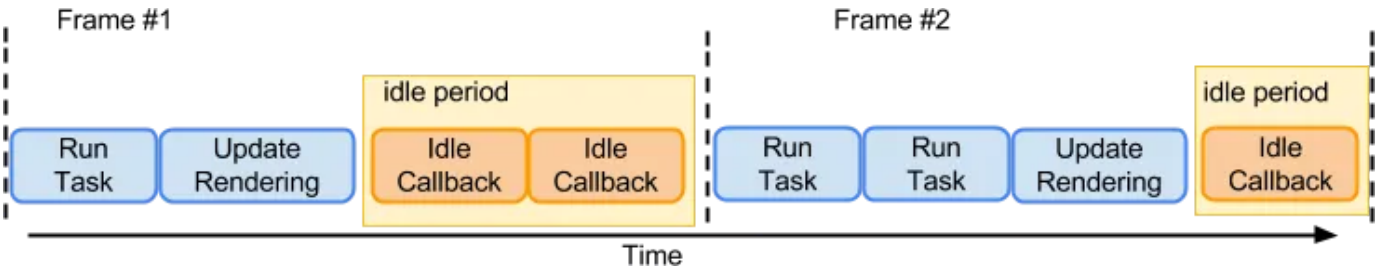
- 属于GUI引擎，raf是在微任务执行完之后，浏览器重排重绘之前执行。
- 执行的时机受屏幕刷新针影响，是不准确的，如果raf之前JS的执行时间过长，依然会造成延迟。
- 当在后台选项卡或隐藏选项卡中运行时，调用都会暂停<iframe>，以提高性能和电池寿命。
- 常用于更新dom，渲染动画。

requestIdleCallback

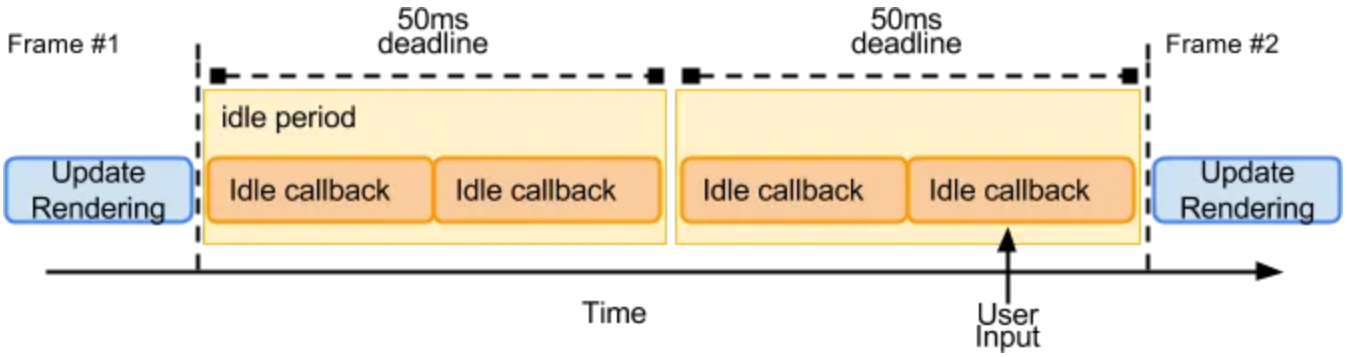
定位于执行后台和低优先级任务、执行频率。

在完成一帧中的输入处理、渲染和合成之后，线程会进入空闲时期（idle period），直到下一帧开始，或者队列中的任务被激活，又或者收到了用户新的输入。

requestIdleCallback 定义的回调就是在这段空闲时期执行。（Frame 渲染帧）



如果不存在屏幕刷新，浏览器会安排连续的长度为 50ms 的空闲时期



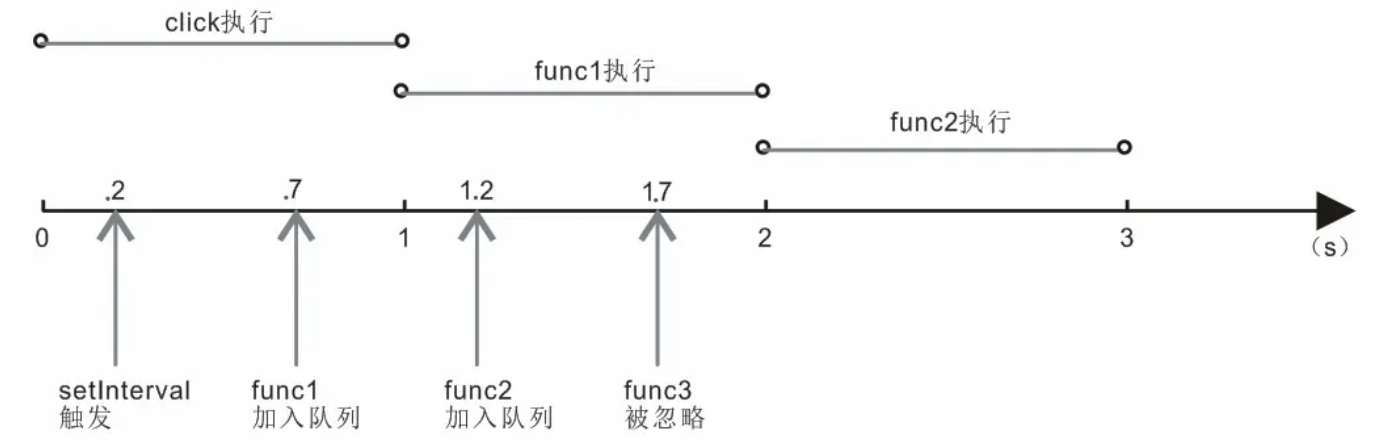
requestIdleCallback的执行时机是在浏览器重排重绘之后，也就是浏览器的空闲时间执行。其实执行的时机是不准确的，raf执行的JS代码耗时可能会过长。

避免在空闲回调中改变 DOM。 空闲回调执行的时候，当前帧已经结束绘制了，所有布局的更新和计算也已经完成。如果你做的改变影响了布局，你可能会强制停止浏览器并重新计算，而从另一方面来看，这是不必要的。如果你的回调需要改变 DOM，它应该使用Window.requestAnimationFrame()来调度它。

setInterval

- “丢帧”现象
- 不同定时器的代码的执行间隔比预期小

我们先来看下面这个例子：



1. click事件点击
2. 0.2s 设置定时器 时间间隔为500ms
3. 0.7s timer线程将定时器回调方法func1推入事件队列，等待主线程执行。
4. 1s click事件执行结束，浏览器从事件队列中取出func1执行
5. 1.2s timer线程将func2 回调方法推入事件触发线程，等待主线程执行。
6. 1.7s timer线程想推入第三次回调方法，但是事件队列中含有该定时器的回调，所以跳过。（丢帧）
7. 2s func1 执行完毕，在下一轮事件循环立即执行func2方法。

丢帧：

当使用 setInterval()时，仅当事件队列中没有该定时器的任何其他代码实例时，才将定时器代码添加到事件队列中。

间隔：

- 时间间隔不受回调函数影响，是由timer线程调度的
- fun是在每一个周期内被执行
- **func 函数的实际调用间隔要比代码中设定的时间间隔要短**

也可能出现这种情况，就是 func 的执行所花费的时间比我们间隔的时间更长。

在这种情况下，JavaScript 引擎会等待 func 执行完成，然后检查调度程序，如果时间到了，则 **立即** 再次执行它。

极端情况下，如果函数每次执行时间都超过 delay 设置的时间，那么每次调用之间将完全没有停顿。

setTimeout

setTimeout的递归层级过深的话，延迟就不是1ms，而是4ms，这样会造成延迟时间过长

浏览器

HTML5标准：如果嵌套的层级超过了 5 层，并且 timeout 小于 4ms，则设置 timeout 为 4ms。

chrome 中的 setTimeout 的行为基本和 HTML5 的标准一致。前 4 次，用的 timeout 都是 1ms以内延迟，后面的间隔时间都超过了 4ms；

nodejs

nodejs 中并没有最小延时 4ms 的限制，而是每次调用都会有 1ms 左右的延时(有时会是0.几毫秒，有时会是1.多毫秒)。

setTimeout和 setImmediate的比较

node v16.18.0 环境下

```JavaScript
→  js-api node  node.js
0.676934003829956 setTimeout
6.1175490617752075 setImmediate
→  js-api node  node.js
0.7656099796295166 setTimeout
7.105100989341736 setImmediate
→  js-api node  node.js
0.6082860231399536 setImmediate
5.74574601650238 setTimeout
```

两个人谁快表现出了一定的随机性，而且setTimeout的延迟也不一定都是1ms，有时会是小于1ms。

但是在I/O回调内，满足永远setImmediate在前

```JavaScript
var fs = require('fs')

fs.readFile(__filename, () => {
  setTimeout(() => {
    console.log('setTimeout', performance.now() - start);
  }, 0);
  setImmediate(() => {
    console.log('setImmediate', performance.now() - start);
  });
});
```

```JavaScript
→  js-api node  node.js
setImmediate 2.0537240505218506
setTimeout 7.27512800693512
→  js-api node  node.js
setImmediate 2.034590005874634
setTimeout 7.136919021606445
→  js-api node  node.js
setImmediate 1.9470280408859253
setTimeout 6.935880064964294
→  js-api node  node.js
setImmediate 2.1759870052337646
setTimeout 7.4798970222473145
→  js-api node  node.js
setImmediate 2.302621006965637
setTimeout 7.8137500286102295
```

参考资料

node 官网 https://cnodejs.org/topic/519b523c63e9f8a5429b25e3

CSDN-settimeout在各个浏览器的最小时间
https://blog.csdn.net/weixin_44730897/article/details/116797681

腾讯云开发者社区 https://cloud.tencent.com/developer/article/2136909

腾讯云开发者社区-「Nodejs进阶」一文吃透异步I/O和事件循环
https://cloud.tencent.com/developer/article/1873357

腾讯云开发者社区-setTimeout和setImmediate到底谁先执行，本文让你彻底理解Event Loop
https://cloud.tencent.com/developer/article/1717260

知乎-https://www.zhihu.com/question/56310675/answer/148502403

你真的了解 setTimeout 么？聊聊 setTimeout 的最小延时问题（附源码细节）
https://www.wangyulue.com/2023/03/%E4%BD%A0%E7%9C%9F%E7%9A%84%E4%BA%86%E8%A7%A3-settimeout-%E4%B9%88/

React 的调度系统 Scheduler https://www.51cto.com/article/741495.html

对于“不用setInterval，用setTimeout”的理解 https://segmentfault.com/a/1190000011282175