在上一篇文章中我们介绍了页面中的事件和消息队列，知道了**浏览器页面是由消息队列和事件循环系统来驱动的**。

那在接下来的两篇文章中，我会通过**setTimeout**和**XMLHttpRequest**这两个WebAPI来介绍事件循环的应用。这两个WebAPI是两种不同类型的应用，比较典型，并且在JavaScript中的使用频率非常高。你可能觉得它们太简单、太基础，但有时候恰恰是基础简单的东西才最重要，了解它们是如何工作的会有助于你写出更加高效的前端代码。

本篇文章主要介绍的是**setTimeout**。其实说起setTimeout方法，从事开发的同学想必都不会陌生，它就是一个**定时器，用来指定某个函数在多少毫秒之后执行**。它会返回一个整数，表示定时器的编号，同时你还可以通过该编号来取消这个定时器。下面的示例代码就演示了定时器最基础的使用方式：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,200);
```

执行上述代码，输出的结果也很明显，通过setTimeout指定在200毫秒之后调用showName函数，并输出“极客时间”四个字。

简单了解了setTimeout的使用方法后，那接下来我们就来看看浏览器是如何实现定时器的，然后再介绍下定时器在使用过程中的一些注意事项。

浏览器怎么实现setTimeout

要了解定时器的工作原理，就得先来回顾下之前讲的事件循环系统，我们知道渲染进程中所有运行在主线程上的任务都需要先添加到消息队列，然后事件循环系统再按照顺序执行消息队列中的任务。下面我们来看看那些典型的事件：

- 当接收到HTML文档数据，渲染引擎就会将“解析DOM”事件添加到消息队列中，
- 当用户改变了Web页面的窗口大小，渲染引擎就会将“重新布局”的事件添加到消息队列中。
- 当触发了JavaScript引擎垃圾回收机制，渲染引擎会将“垃圾回收”任务添加到消息队列中。
- 同样，如果要执行一段异步JavaScript代码，也是需要将执行任务添加到消息队列中。

以上列举的只是一小部分事件，这些事件被添加到消息队列之后，事件循环系统就会按照消息队列中的顺序来执行事件。

所以说要执行一段异步任务，需要先将任务添加到消息队列中。不过通过定时器设置回调函数有点特别，它们需要在指定的时间间隔内被调用，但消息队列中的任务是按照顺序执行的，所以为了保证回调函数能在指定时间内执行，你不能将定时器的回调函数直接添加到消息队列中。

那么该怎么设计才能让定时器设置的回调事件在规定时间内被执行呢？你也可以思考下，如果让你在消息循环系统的基础之上加上定时器的功能，你会如何设计？

在Chrome中除了正常使用的消息队列之外，还有另外一个消息队列，这个队列中维护了需要延迟执行的任务列表，包括了定时器和Chromium内部一些需要延迟执行的任务。所以当通过JavaScript创建一个定时器时，渲染进程会将该定时器的回调任务添加到延迟队列中。

如果感兴趣，你可以参考Chromium中关于队列部分的源码。

源码中延迟执行队列的定义如下所示：

```
 DelayedIncomingQueue delayed_incoming_queue;
```

当通过JavaScript调用setTimeout设置回调函数的时候，渲染进程将会创建一个回调任务，包含了回调函数showName、当前发起时间、延迟执行时间，其模拟代码如下所示：

```
struct DelayTask{
  int64 id;
  CallBackFunction cbf;
  int start_time;
  int delay_time;
};
DelayTask timerTask;
timerTask.cbf = showName;
timerTask.start_time = getCurrentTime(); //获取当前时间
timerTask.delay_time = 200;//设置延迟执行时间
```

创建好回调任务之后，再将该任务添加到延迟执行队列中，代码如下所示：

```
delayed_incoming_queue.push(timerTask);
```

现在通过定时器发起的任务就被保存到延迟队列中了，那接下来我们再来看看消息循环系统是怎么触发延迟队列的。

我们可以来完善上一篇文章中消息循环的代码，在其中加入执行延迟队列的代码，如下所示：

```
void ProcessTimerTask(){
  //从delayed_incoming_queue中取出已经到期的定时器任务
  //依次执行这些任务
}

TaskQueue task_queue;
void ProcessTask();
bool keep_running = true;
void MainTherad(){
  for(;;){
    //执行消息队列中的任务
    Task task = task_queue.takeTask();
    ProcessTask(task);

    //执行延迟队列中的任务
    ProcessDelayTask()

    if(!keep_running) //如果设置了退出标志，那么直接退出线程循环
        break;
  }
}
```

从上面代码可以看出来，我们添加了一个**ProcessDelayTask函数**，该函数是专门用来处理延迟执行任务的。这里我们要重点关注它的执行时机，在上段代码中，处理完消息队列中的一个任务之后，就开始执行ProcessDelayTask函数。ProcessDelayTask函数会根据发起时间和延迟时间计算出到期的任务，然后依次执行这些到期的任务。等到期的任务执行完成之后，再继续下一个循环过程。通过这样的方式，一个完整的定时器就实现了。

设置一个定时器，JavaScript引擎会返回一个定时器的ID。那通常情况下，当一个定时器的任务还没有被执行的时候，也是可以取消的，具体方法是调用**clearTimeout函数**，并传入需要取消的定时器的ID。如下面代码所示：

```
clearTimeout(timer_id)
```

其实浏览器内部实现取消定时器的操作也是非常简单的，就是直接从delayed_incoming_queue延迟队列中，通过ID查找到对应的任务，然后再将其从队列中删除掉就可以了。

使用setTimeout的一些注意事项

现在你应该知道在浏览器内部定时器是如何工作的了。不过在使用定时器的过程中，如果你不了解定时器的一些细节，那么很有可能掉进定时器的一些陷阱里。所以接下来，我们就来讲解一下在使用定时器过程中存在的那些陷阱。

1. 如果当前任务执行时间过久，会影响定时器任务的执行

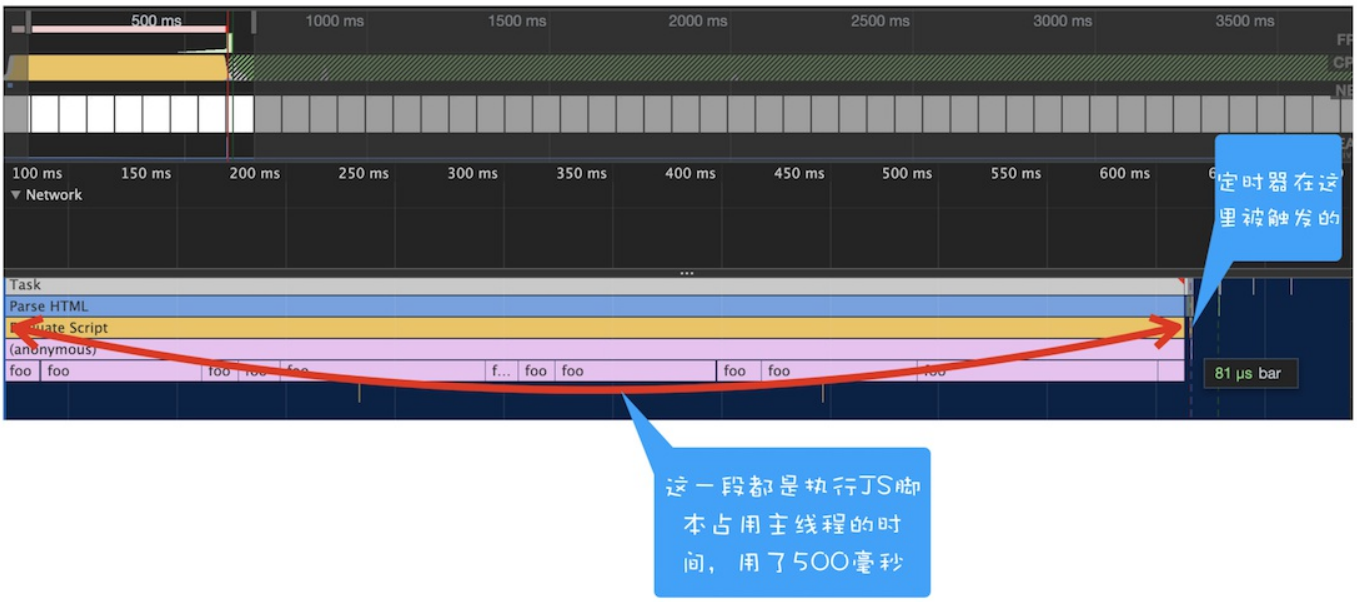
在使用setTimeout的时候，有很多因素会导致回调函数执行比设定的预期值要久，其中一个就是当前任务执行时间过久从而导致定时器设置的任务被延后执行。我们先看下面这段代码：

```
function bar() {
    console.log('bar')
}
function foo() {
    setTimeout(bar, 0);
    for (let i = 0; i < 5000; i++) {
        let i = 5+8+8+8
        console.log(i)
    }
}
foo()
```

这段代码中，在执行foo函数的时候使用setTimeout设置了一个0延时的回调任务，设置好回调任务后，foo函数会继续执行5000次for循环。

通过setTimeout设置的回调任务被放入了消息队列中并且等待下一次执行，这里并不是立即执行的；要执行消息队列中的下个任务，需要等待当前的任务执行完成，由于当前这段代码要执行5000次的for循环，所以当前这个任务的执行时间会比较久一点。这势必会影响到下个任务的执行时间。

你也可以打开Performance来看看其执行过程，如下图所示：



长任务导致定时器被延后执行

从图中可以看到，执行foo函数所消耗的时长是500毫秒，这也就意味着通过setTimeout设置的任务会被推迟到500毫秒以后再去执行，而设置setTimeout的回调延迟时间是0。

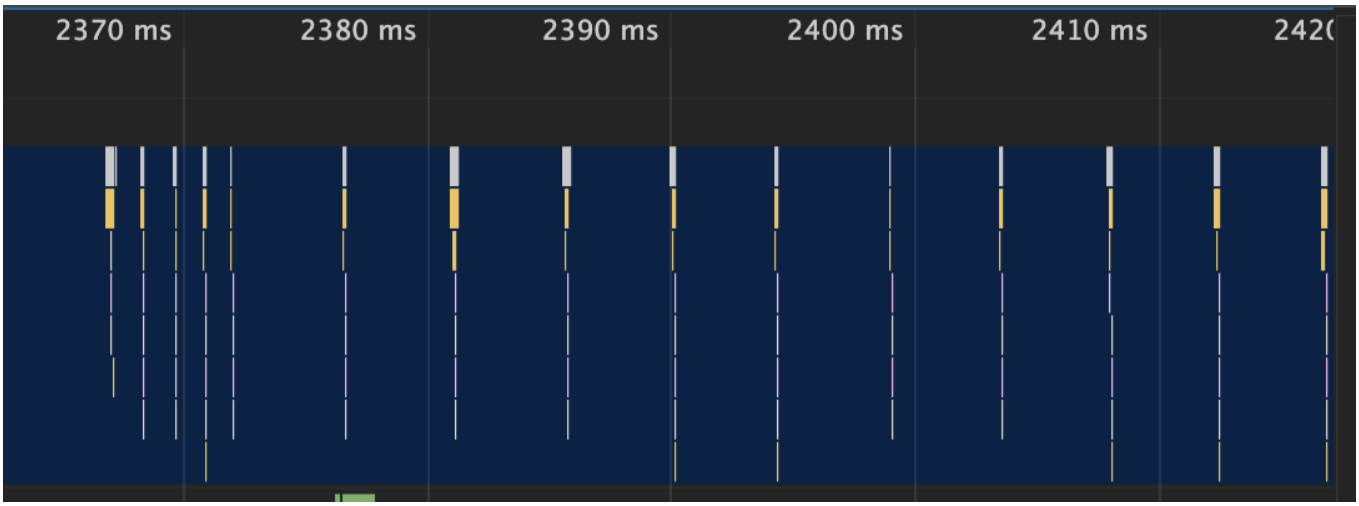
2. 如果setTimeout存在嵌套调用，那么系统会设置最短时间间隔为4毫秒

也就是说在定时器函数里面嵌套调用定时器，也会延长定时器的执行时间，可以先看下面的这段代码：

```
function cb() { setTimeout(cb, 0); }
setTimeout(cb, 0);
```

上述这段代码你有没有看出存在什么问题？

你还是可以通过Performance来记录下这段代码的执行过程，如下图所示：



循环嵌套调用setTimeout

上图中的竖线就是定时器的函数回调过程，从图中可以看出，前面五次调用的时间间隔比较小，嵌套调用超过五次以上，后面每次的调用最小时间间隔是4毫秒。之所以出现这样的情况，是因为在Chrome中，定时器被嵌套调用5次以上，系统会判断该函数方法被阻塞了，如果定时器的调用时间间隔小于4毫秒，那么浏览器会将每次调用的时间间隔设置为4毫秒。下面是Chromium实现4毫秒延迟的代码，你可以看下：

```
static const int kMaxTimerNestingLevel = 5;

// Chromium uses a minimum timer interval of 4ms. We'd like to go
// lower; however, there are poorly coded websites out there which do
// create CPU-spinning loops.  Using 4ms prevents the CPU from
// spinning too busily and provides a balance between CPU spinning and
// the smallest possible interval timer.
static constexpr base::TimeDelta kMinimumInterval = base::TimeDelta::FromMilliseconds(4);

base::TimeDelta interval_milliseconds =
      std::max(base::TimeDelta::FromMilliseconds(1), interval);

  if (interval_milliseconds < kMinimumInterval &&
      nesting_level_ >= kMaxTimerNestingLevel)
    interval_milliseconds = kMinimumInterval;

  if (single_shot)
    StartOneShot(interval_milliseconds, FROM_HERE);
  else
    StartRepeating(interval_milliseconds, FROM_HERE);
```

所以，一些实时性较高的需求就不太适合使用setTimeout了，比如你用setTimeout来实现JavaScript动画就不是一个很好的主意。

3. 未激活的页面，setTimeout执行最小间隔是1000毫秒

除了前面的4毫秒延迟，还有一个很容易被忽略的地方，那就是未被激活的页面中定时器最小值大于1000毫秒，也就是说，如果标签不是当前的激活标签，那么定时器最小的时间间隔是1000毫秒，目的是为了优化后台页面的加载损耗以及降低耗电量。这一点你在使用定时器的时候要注意。

4. 延时执行时间有最大值

除了要了解定时器的回调函数时间比实际设定值要延后之外，还有一点需要注意下，那就是Chrome、Safari、Firefox都是以32个bit来存储延时值的，32bit最大只能存放的数字是2147483647毫秒，这就意味着，如果setTimeout设置的延迟值大于 2147483647毫秒（大约24.8天）时就会溢出，那么相当于延时值被设置为0了，这导致定时器会被立即执行。你可以运行下面这段代码：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,2147483648);//会被理解调用执行
```

运行后可以看到，这段代码是立即被执行的。但如果将延时值修改为小于2147483647毫秒的某个值，那么执行时就没有问题了。

5. 使用setTimeout设置的回调函数中的this不符合直觉

如果被setTimeout推迟执行的回调函数是某个对象的方法，那么该方法中的this关键字将指向全局环境，而不是定义时所在的那个对象。这点在前面介绍this的时候也提过，你可以看下面这段代码的执行结果：

```
var name= 1;
var MyObj = {
  name: 2,
  showName: function(){
    console.log(this.name);
  }
}
setTimeout(MyObj.showName,1000)
```

这里输出的是1，因为这段代码在编译的时候，执行上下文中的this会被设置为全局window，如果是严格模式，会被设置为undefined。

那么该怎么解决这个问题呢？通常可以使用下面这两种方法。

第一种是将MyObj.showName放在匿名函数中执行，如下所示：

```
//箭头函数
setTimeout(() => {
    MyObj.showName()
}, 1000);
//或者function函数
setTimeout(function() {
```

```
  MyObj.showName();
}, 1000)
```

第二种是使用bind方法，将showName绑定在MyObj上面，代码如下所示：

```
setTimeout(MyObj.showName.bind(MyObj), 1000)
```

总结

好了，今天我们就介绍到这里，下面我来总结下今天的内容。

- 首先，为了支持定时器的实现，浏览器增加了延时队列。
- 其次，由于消息队列排队和一些系统级别的限制，通过setTimeout设置的回调任务并非总是可以实时地被执行，这样就不能满足一些实时性要求较高的需求了。
- 最后，在定时器中使用过程中，还存在一些陷阱，需要你多加注意。

通过分析和讲解，你会发现函数setTimeout在时效性上面有很多先天的不足，所以对于一些时间精度要求比较高的需求，应该有针对性地采取一些其他的方案。

思考时间

今天我们介绍了setTimeout，相信你现在也知道它是怎么工作的了，不过由于使用setTimeout设置的回调任务实时性并不是太好，所以很多场景并不适合使用setTimeout。比如你要使用JavaScript来实现动画效果，函数requestAnimationFrame就是个很好的选择。

那么今天留给你的作业是：你需要网上搜索了解下requestAnimationFrame的工作机制，并对比setTimeout，然后分析出requestAnimationFrame实现的动画效果比setTimeout好的原因。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。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。

在上一篇文章中我们介绍了页面中的事件和消息队列，知道了**浏览器页面是由消息队列和事件循环系统来驱动的**。

那在接下来的两篇文章中，我会通过**setTimeout**和**XMLHttpRequest**这两个WebAPI来介绍事件循环的应用。这两个WebAPI是两种不同类型的应用，比较典型，并且在JavaScript中的使用频率非常高。你可能觉得它们太简单、太基础，但有时候恰恰是基础简单的东西才最重要，了解它们是如何工作的会有助于你写出更加高效的前端代码。

本篇文章主要介绍的是**setTimeout**。其实说起setTimeout方法，从事开发的同学想必都不会陌生，它就是一个**定时器，用来指定某个函数在多少毫秒之后执行**。它会返回一个整数，表示定时器的编号，同时你还可以通过该编号来取消这个定时器。下面的示例代码就演示了定时器最基础的使用方式：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,200);
```

执行上述代码，输出的结果也很明显，通过setTimeout指定在200毫秒之后调用showName函数，并输出“极客时间”四个字。

简单了解了setTimeout的使用方法后，那接下来我们就来看看浏览器是如何实现定时器的，然后再介绍下定时器在使用过程中的一些注意事项。

浏览器怎么实现setTimeout

要了解定时器的工作原理，就得先来回顾下之前讲的事件循环系统，我们知道渲染进程中所有运行在主线程上的任务都需要先添加到消息队列，然后事件循环系统再按照顺序执行消息队列中的任务。下面我们来看看那些典型的事件：

- 当接收到HTML文档数据，渲染引擎就会将“解析DOM”事件添加到消息队列中，
- 当用户改变了Web页面的窗口大小，渲染引擎就会将“重新布局”的事件添加到消息队列中。
- 当触发了JavaScript引擎垃圾回收机制，渲染引擎会将“垃圾回收”任务添加到消息队列中。
- 同样，如果要执行一段异步JavaScript代码，也是需要将执行任务添加到消息队列中。

以上列举的只是一小部分事件，这些事件被添加到消息队列之后，事件循环系统就会按照消息队列中的顺序来执行事件。

所以说要执行一段异步任务，需要先将任务添加到消息队列中。不过通过定时器设置回调函数有点特别，它们需要在指定的时间间隔内被调用，但消息队列中的任务是按照顺序执行的，所以为了保证回调函数能在指定时间内执行，你不能将定时器的回调函数直接添加到消息队列中。

那么该怎么设计才能让定时器设置的回调事件在规定时间内被执行呢？你也可以思考下，如果让你在消息循环系统的基础之上加上定时器的功能，你会如何设计？

在Chrome中除了正常使用的消息队列之外，还有另外一个消息队列，这个队列中维护了需要延迟执行的任务列表，包括了定时器和Chromium内部一些需要延迟执行的任务。所以当通过JavaScript创建一个定时器时，渲染进程会将该定时器的回调任务添加到延迟队列中。

如果感兴趣，你可以参考Chromium中关于队列部分的源码。

源码中延迟执行队列的定义如下所示：

```
 DelayedIncomingQueue delayed_incoming_queue;
```

当通过JavaScript调用setTimeout设置回调函数的时候，渲染进程将会创建一个回调任务，包含了回调函数showName、当前发起时间、延迟执行时间，其模拟代码如下所示：

```
struct DelayTask{
  int64 id;
  CallBackFunction cbf;
  int start_time;
  int delay_time;
};
DelayTask timerTask;
timerTask.cbf = showName;
timerTask.start_time = getCurrentTime(); //获取当前时间
timerTask.delay_time = 200;//设置延迟执行时间
```

创建好回调任务之后，再将该任务添加到延迟执行队列中，代码如下所示：

```
delayed_incoming_queue.push(timerTask);
```

现在通过定时器发起的任务就被保存到延迟队列中了，那接下来我们再来看看消息循环系统是怎么触发延迟队列的。

我们可以来完善上一篇文章中消息循环的代码，在其中加入执行延迟队列的代码，如下所示：

```
void ProcessTimerTask(){
  //从delayed_incoming_queue中取出已经到期的定时器任务
  //依次执行这些任务
}

TaskQueue task_queue;
void ProcessTask();
bool keep_running = true;
void MainTherad(){
  for(;;){
    //执行消息队列中的任务
    Task task = task_queue.takeTask();
    ProcessTask(task);

    //执行延迟队列中的任务
    ProcessDelayTask()

    if(!keep_running) //如果设置了退出标志，那么直接退出线程循环
        break;
  }
}
```

从上面代码可以看出来，我们添加了一个**ProcessDelayTask函数**，该函数是专门用来处理延迟执行任务的。这里我们要重点关注它的执行时机，在上段代码中，处理完消息队列中的一个任务之后，就开始执行ProcessDelayTask函数。ProcessDelayTask函数会根据发起时间和延迟时间计算出到期的任务，然后依次执行这些到期的任务。等到期的任务执行完成之后，再继续下一个循环过程。通过这样的方式，一个完整的定时器就实现了。

设置一个定时器，JavaScript引擎会返回一个定时器的ID。那通常情况下，当一个定时器的任务还没有被执行的时候，也是可以取消的，具体方法是调用**clearTimeout函数**，并传入需要取消的定时器的ID。如下面代码所示：

```
clearTimeout(timer_id)
```

其实浏览器内部实现取消定时器的操作也是非常简单的，就是直接从delayed_incoming_queue延迟队列中，通过ID查找到对应的任务，然后再将其从队列中删除掉就可以了。

使用setTimeout的一些注意事项

现在你应该知道在浏览器内部定时器是如何工作的了。不过在使用定时器的过程中，如果你不了解定时器的一些细节，那么很有可能掉进定时器的一些陷阱里。所以接下来，我们就来讲解一下在使用定时器过程中存在的那些陷阱。

1. 如果当前任务执行时间过久，会影响定时器任务的执行

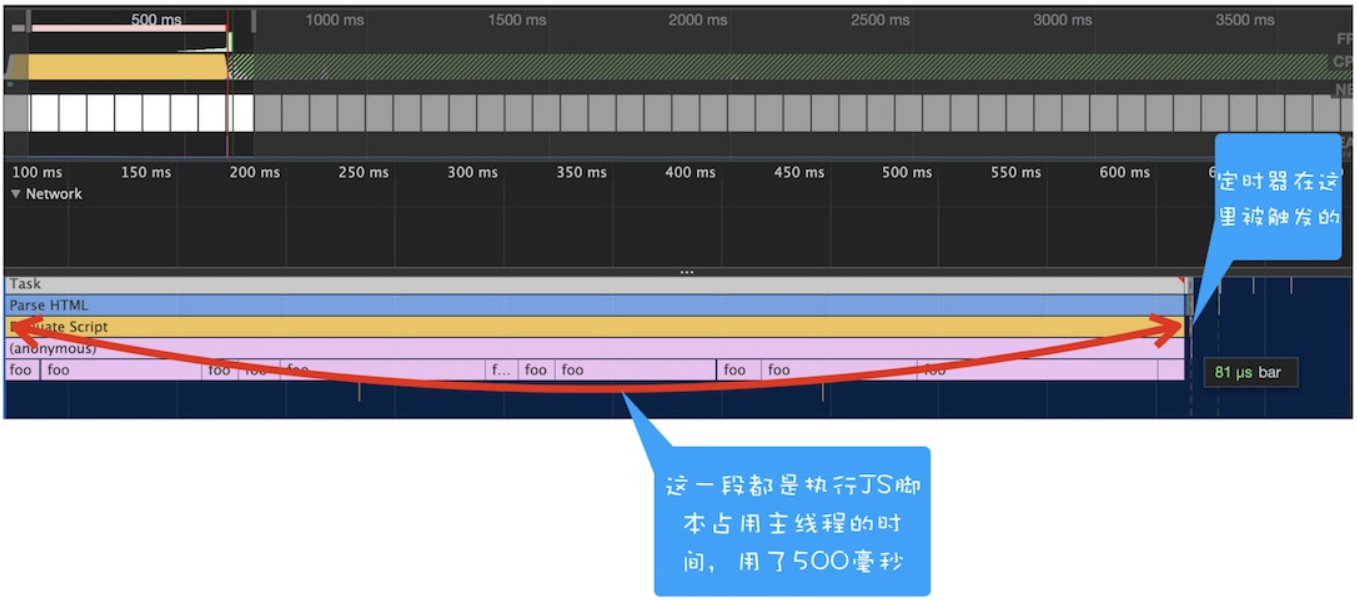
在使用setTimeout的时候，有很多因素会导致回调函数执行比设定的预期值要久，其中一个就是当前任务执行时间过久从而导致定时器设置的任务被延后执行。我们先看下面这段代码：

```
function bar() {
    console.log('bar')
}
function foo() {
    setTimeout(bar, 0);
    for (let i = 0; i < 5000; i++) {
        let i = 5+8+8+8
        console.log(i)
    }
}
foo()
```

这段代码中，在执行foo函数的时候使用setTimeout设置了一个0延时的回调任务，设置好回调任务后，foo函数会继续执行5000次for循环。

通过setTimeout设置的回调任务被放入了消息队列中并且等待下一次执行，这里并不是立即执行的；要执行消息队列中的下个任务，需要等待当前的任务执行完成，由于当前这段代码要执行5000次的for循环，所以当前这个任务的执行时间会比较久一点。这势必会影响到下个任务的执行时间。

你也可以打开Performance来看看其执行过程，如下图所示：



长任务导致定时器被延后执行

从图中可以看到，执行foo函数所消耗的时长是500毫秒，这也就意味着通过setTimeout设置的任务会被推迟到500毫秒以后再去执行，而设置setTimeout的回调延迟时间是0。

2. 如果setTimeout存在嵌套调用，那么系统会设置最短时间间隔为4毫秒

也就是说在定时器函数里面嵌套调用定时器，也会延长定时器的执行时间，可以先看下面的这段代码：

```
function cb() { setTimeout(cb, 0); }
setTimeout(cb, 0);
```

上述这段代码你有没有看出存在什么问题？

你还是可以通过Performance来记录下这段代码的执行过程，如下图所示：

循环嵌套调用setTimeout

上图中的竖线就是定时器的函数回调过程，从图中可以看出，前面五次调用的时间间隔比较小，嵌套调用超过五次以上，后面每次的调用最小时间间隔是4毫秒。之所以出现这样的情况，是因为在Chrome中，定时器被嵌套调用5次以上，系统会判断该函数方法被阻塞了，如果定时器的调用时间间隔小于4毫秒，那么浏览器会将每次调用的时间间隔设置为4毫秒。下面是Chromium实现4毫秒延迟的代码，你可以看下：

```
static const int kMaxTimerNestingLevel = 5;

// Chromium uses a minimum timer interval of 4ms. We'd like to go
// lower; however, there are poorly coded websites out there which do
// create CPU-spinning loops.  Using 4ms prevents the CPU from
// spinning too busily and provides a balance between CPU spinning and
// the smallest possible interval timer.
static constexpr base::TimeDelta kMinimumInterval = base::TimeDelta::FromMilliseconds(4);

base::TimeDelta interval_milliseconds =
      std::max(base::TimeDelta::FromMilliseconds(1), interval);

  if (interval_milliseconds < kMinimumInterval &&
      nesting_level_ >= kMaxTimerNestingLevel)
    interval_milliseconds = kMinimumInterval;

  if (single_shot)
    StartOneShot(interval_milliseconds, FROM_HERE);
  else
    StartRepeating(interval_milliseconds, FROM_HERE);
```

所以，一些实时性较高的需求就不太适合使用setTimeout了，比如你用setTimeout来实现JavaScript动画就不是一个很好的主意。

3. 未激活的页面，setTimeout执行最小间隔是1000毫秒

除了前面的4毫秒延迟，还有一个很容易被忽略的地方，那就是未被激活的页面中定时器最小值大于1000毫秒，也就是说，如果标签不是当前的激活标签，那么定时器最小的时间间隔是1000毫秒，目的是为了优化后台页面的加载损耗以及降低耗电量。这一点你在使用定时器的时候要注意。

4. 延时执行时间有最大值

除了要了解定时器的回调函数时间比实际设定值要延后之外，还有一点需要注意下，那就是Chrome、Safari、Firefox都是以32个bit来存储延时值的，32bit最大只能存放的数字是2147483647毫秒，这就意味着，如果setTimeout设置的延迟值大于 2147483647毫秒（大约24.8天）时就会溢出，那么相当于延时值被设置为0了，这导致定时器会被立即执行。你可以运行下面这段代码：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,2147483648);//会被理解调用执行
```

运行后可以看到，这段代码是立即被执行的。但如果将延时值修改为小于2147483647毫秒的某个值，那么执行时就没有问题了。

5. 使用setTimeout设置的回调函数中的this不符合直觉

如果被setTimeout推迟执行的回调函数是某个对象的方法，那么该方法中的this关键字将指向全局环境，而不是定义时所在的那个对象。这点在前面介绍this的时候也提过，你可以看下面这段代码的执行结果：

```
var name= 1;
var MyObj = {
  name: 2,
  showName: function(){
    console.log(this.name);
  }
```

```
}
setTimeout(MyObj.showName,1000)
```

这里输出的是1，因为这段代码在编译的时候，执行上下文中的this会被设置为全局window，如果是严格模式，会被设置为undefined。

那么该怎么解决这个问题呢？通常可以使用下面这两种方法。

第一种是将MyObj.showName放在匿名函数中执行，如下所示：

```
//箭头函数
setTimeout(() => {
    MyObj.showName()
}, 1000);
//或者function函数
setTimeout(function() {
  MyObj.showName();
}, 1000)
```

第二种是使用bind方法，将showName绑定在MyObj上面，代码如下所示：

```
setTimeout(MyObj.showName.bind(MyObj), 1000)
```

总结

好了，今天我们就介绍到这里，下面我来总结下今天的内容。

- 首先，为了支持定时器的实现，浏览器增加了延时队列。
- 其次，由于消息队列排队和一些系统级别的限制，通过setTimeout设置的回调任务并非总是可以实时地被执行，这样就不能满足一些实时性要求较高的需求了。
- 最后，在定时器中使用过程中，还存在一些陷阱，需要你多加注意。

通过分析和讲解，你会发现函数setTimeout在时效性上面有很多先天的不足，所以对于一些时间精度要求比较高的需求，应该有针对性地采取一些其他的方案。

思考时间

今天我们介绍了setTimeout，相信你现在也知道它是怎么工作的了，不过由于使用setTimeout设置的回调任务实时性并不是太好，所以很多场景并不适合使用setTimeout。比如你要使用JavaScript来实现动画效果，函数requestAnimationFrame就是个很好的选择。

那么今天留给你的作业是：你需要网上搜索了解下requestAnimationFrame的工作机制，并对比setTimeout，然后分析出requestAnimationFrame实现的动画效果比setTimeout好的原因。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。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。

在上一篇文章中我们介绍了页面中的事件和消息队列，知道了**浏览器页面是由消息队列和事件循环系统来驱动的**。

那在接下来的两篇文章中，我会通过**setTimeout**和**XMLHttpRequest**这两个WebAPI来介绍事件循环的应用。这两个WebAPI是两种不同类型的应用，比较典型，并且在JavaScript中的使用频率非常高。你可能觉得它们太简单、太基础，但有时候恰恰是基础简单的东西才最重要，了解它们是如何工作的会有助于你写出更加高效的前端代码。

本篇文章主要介绍的是**setTimeout**。其实说起setTimeout方法，从事开发的同学想必都不会陌生，它就是一个**定时器，用来指定某个函数在多少毫秒之后执行**。它会返回一个整数，表示定时器的编号，同时你还可以通过该编号来取消这个定时器。下面的示例代码就演示了定时器最基础的使用方式：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,200);
```

执行上述代码，输出的结果也很明显，通过setTimeout指定在200毫秒之后调用showName函数，并输出“极客时间”四个字。

简单了解了setTimeout的使用方法后，那接下来我们就来看看浏览器是如何实现定时器的，然后再介绍下定时器在使用过程中的一些注意事项。

浏览器怎么实现setTimeout

要了解定时器的工作原理，就得先来回顾下之前讲的事件循环系统，我们知道渲染进程中所有运行在主线程上的任务都需要先添加到消息队列，然后事件循环系统再按照顺序执行消息队列中的任务。下面我们来看看那些典型的事件：

- 当接收到HTML文档数据，渲染引擎就会将“解析DOM”事件添加到消息队列中，
- 当用户改变了Web页面的窗口大小，渲染引擎就会将“重新布局”的事件添加到消息队列中。
- 当触发了JavaScript引擎垃圾回收机制，渲染引擎会将“垃圾回收”任务添加到消息队列中。
- 同样，如果要执行一段异步JavaScript代码，也是需要将执行任务添加到消息队列中。

以上列举的只是一小部分事件，这些事件被添加到消息队列之后，事件循环系统就会按照消息队列中的顺序来执行事件。

所以说要执行一段异步任务，需要先将任务添加到消息队列中。不过通过定时器设置回调函数有点特别，它们需要在指定的时间间隔内被调用，但消息队列中的任务是按照顺序执行的，所以为了保证回调函数能在指定时间内执行，你不能将定时器的回调函数直接添加到消息队列中。

那么该怎么设计才能让定时器设置的回调事件在规定时间内被执行呢？你也可以思考下，如果让你在消息循环系统的基础之上加上定时器的功能，你会如何设计？

在Chrome中除了正常使用的消息队列之外，还有另外一个消息队列，这个队列中维护了需要延迟执行的任务列表，包括了定时器和Chromium内部一些需要延迟执行的任务。所以当通过JavaScript创建一个定时器时，渲染进程会将该定时器的回调任务添加到延迟队列中。

如果感兴趣，你可以参考Chromium中关于队列部分的源码。

源码中延迟执行队列的定义如下所示：

```
 DelayedIncomingQueue delayed_incoming_queue;
```

当通过JavaScript调用setTimeout设置回调函数的时候，渲染进程将会创建一个回调任务，包含了回调函数showName、当前发起时间、延迟执行时间，其模拟代码如下所示：

```
struct DelayTask{
  int64 id;
  CallBackFunction cbf;
  int start_time;
  int delay_time;
};
DelayTask timerTask;
timerTask.cbf = showName;
timerTask.start_time = getCurrentTime(); //获取当前时间
timerTask.delay_time = 200;//设置延迟执行时间
```

创建好回调任务之后，再将该任务添加到延迟执行队列中，代码如下所示：

```
delayed_incoming_queue.push(timerTask);
```

现在通过定时器发起的任务就被保存到延迟队列中了，那接下来我们再来看看消息循环系统是怎么触发延迟队列的。

我们可以来完善上一篇文章中消息循环的代码，在其中加入执行延迟队列的代码，如下所示：

```
void ProcessTimerTask(){
  //从delayed_incoming_queue中取出已经到期的定时器任务
  //依次执行这些任务
}

TaskQueue task_queue;
void ProcessTask();
bool keep_running = true;
void MainTherad(){
  for(;;){
    //执行消息队列中的任务
    Task task = task_queue.takeTask();
    ProcessTask(task);

    //执行延迟队列中的任务
    ProcessDelayTask()

    if(!keep_running) //如果设置了退出标志，那么直接退出线程循环
        break;
  }
}
```

从上面代码可以看出来，我们添加了一个**ProcessDelayTask函数**，该函数是专门用来处理延迟执行任务的。这里我们要重点关注它的执行时机，在上段代码中，处理完消息队列中的一个任务之后，就开始执行ProcessDelayTask函数。ProcessDelayTask函数会根据发起时间和延迟时间计算出到期的任务，然后依次执行这些到期的任务。等到期的任务执行完成之后，再继续下一个循环过程。通过这样的方式，一个完整的定时器就实现了。

设置一个定时器，JavaScript引擎会返回一个定时器的ID。那通常情况下，当一个定时器的任务还没有被执行的时候，也是可以取消的，具体方法是调用**clearTimeout函数**，并传入需要取消的定时器的ID。如下面代码所示：

```
clearTimeout(timer_id)
```

其实浏览器内部实现取消定时器的操作也是非常简单的，就是直接从delayed_incoming_queue延迟队列中，通过ID查找到对应的任务，然后再将其从队列中删除掉就可以了。

使用setTimeout的一些注意事项

现在你应该知道在浏览器内部定时器是如何工作的了。不过在使用定时器的过程中，如果你不了解定时器的一些细节，那么很有可能掉进定时器的一些陷阱里。所以接下来，我们就来讲解一下在使用定时器过程中存在的那些陷阱。

1. 如果当前任务执行时间过久，会影响定时器任务的执行

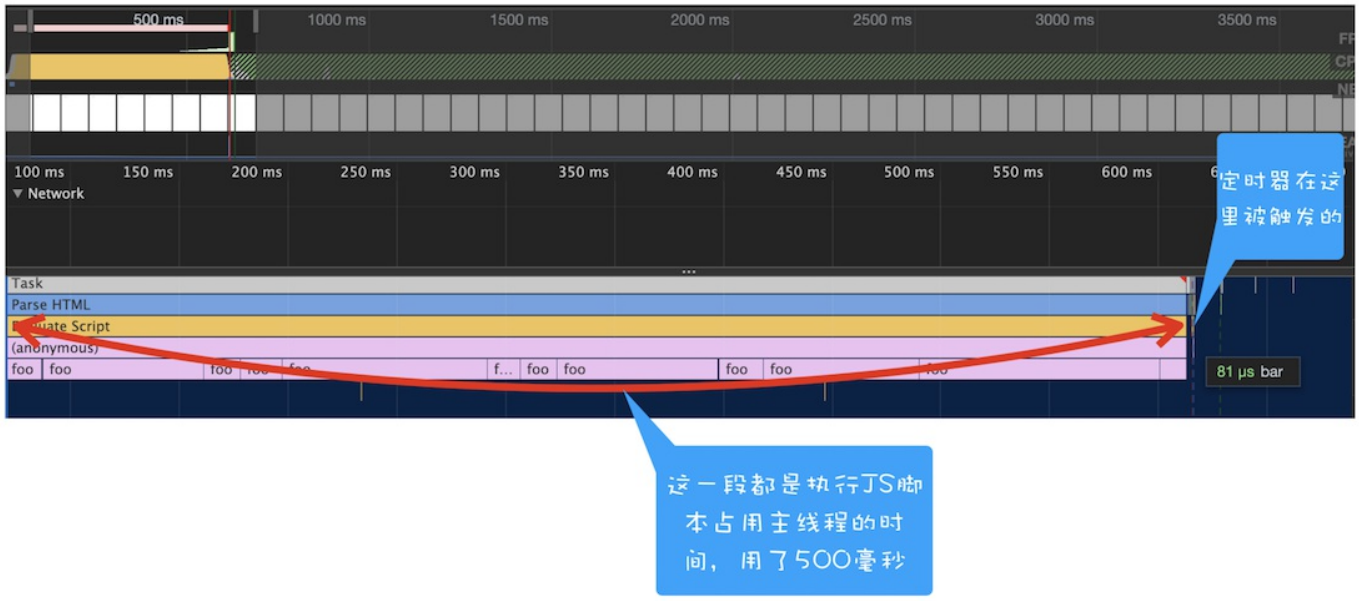
在使用setTimeout的时候，有很多因素会导致回调函数执行比设定的预期值要久，其中一个就是当前任务执行时间过久从而导致定时器设置的任务被延后执行。我们先看下面这段代码：

```
function bar() {
    console.log('bar')
}
function foo() {
    setTimeout(bar, 0);
    for (let i = 0; i < 5000; i++) {
        let i = 5+8+8+8
        console.log(i)
    }
}
foo()
```

这段代码中，在执行foo函数的时候使用setTimeout设置了一个0延时的回调任务，设置好回调任务后，foo函数会继续执行5000次for循环。

通过setTimeout设置的回调任务被放入了消息队列中并且等待下一次执行，这里并不是立即执行的；要执行消息队列中的下个任务，需要等待当前的任务执行完成，由于当前这段代码要执行5000次的for循环，所以当前这个任务的执行时间会比较久一点。这势必会影响到下个任务的执行时间。

你也可以打开Performance来看看其执行过程，如下图所示：



长任务导致定时器被延后执行

从图中可以看到，执行foo函数所消耗的时长是500毫秒，这也就意味着通过setTimeout设置的任务会被推迟到500毫秒以后再去执行，而设置setTimeout的回调延迟时间是0。

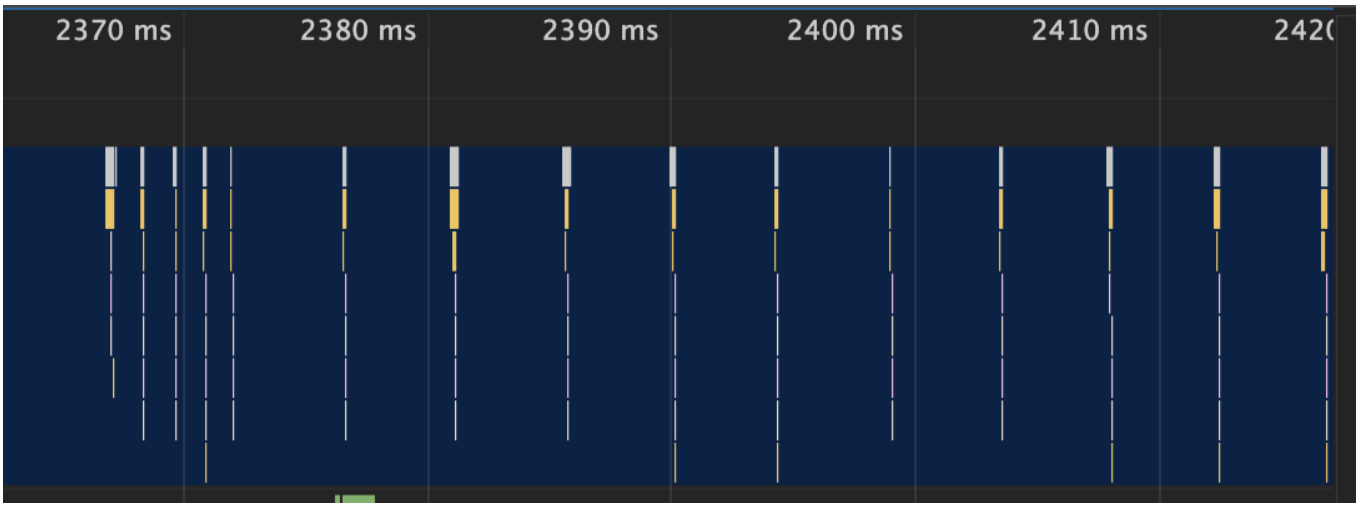
2. 如果setTimeout存在嵌套调用，那么系统会设置最短时间间隔为4毫秒

也就是说在定时器函数里面嵌套调用定时器，也会延长定时器的执行时间，可以先看下面的这段代码：

```
function cb() { setTimeout(cb, 0); }
setTimeout(cb, 0);
```

上述这段代码你有没有看出存在什么问题？

你还是可以通过Performance来记录下这段代码的执行过程，如下图所示：



循环嵌套调用setTimeout

上图中的竖线就是定时器的函数回调过程，从图中可以看出，前面五次调用的时间间隔比较小，嵌套调用超过五次以上，后面每次的调用最小时间间隔是4毫秒。之所以出现这样的情况，是因为在Chrome中，定时器被嵌套调用5次以上，系统会判断该函数方法被阻塞了，如果定时器的调用时间间隔小于4毫秒，那么浏览器会将每次调用的时间间隔设置为4毫秒。下面是Chromium实现4毫秒延迟的代码，你可以看下：

```
static const int kMaxTimerNestingLevel = 5;

// Chromium uses a minimum timer interval of 4ms. We'd like to go
// lower; however, there are poorly coded websites out there which do
// create CPU-spinning loops.  Using 4ms prevents the CPU from
// spinning too busily and provides a balance between CPU spinning and
// the smallest possible interval timer.
static constexpr base::TimeDelta kMinimumInterval = base::TimeDelta::FromMilliseconds(4);

base::TimeDelta interval_milliseconds =
      std::max(base::TimeDelta::FromMilliseconds(1), interval);

  if (interval_milliseconds < kMinimumInterval &&
      nesting_level_ >= kMaxTimerNestingLevel)
    interval_milliseconds = kMinimumInterval;

  if (single_shot)
    StartOneShot(interval_milliseconds, FROM_HERE);
  else
    StartRepeating(interval_milliseconds, FROM_HERE);
```

所以，一些实时性较高的需求就不太适合使用setTimeout了，比如你用setTimeout来实现JavaScript动画就不是一个很好的主意。

3. 未激活的页面，setTimeout执行最小间隔是1000毫秒

除了前面的4毫秒延迟，还有一个很容易被忽略的地方，那就是未被激活的页面中定时器最小值大于1000毫秒，也就是说，如果标签不是当前的激活标签，那么定时器最小的时间间隔是1000毫秒，目的是为了优化后台页面的加载损耗以及降低耗电量。这一点你在使用定时器的时候要注意。

4. 延时执行时间有最大值

除了要了解定时器的回调函数时间比实际设定值要延后之外，还有一点需要注意下，那就是Chrome、Safari、Firefox都是以32个bit来存储延时值的，32bit最大只能存放的数字是2147483647毫秒，这就意味着，如果setTimeout设置的延迟值大于 2147483647毫秒（大约24.8天）时就会溢出，那么相当于延时值被设置为0了，这导致定时器会被立即执行。你可以运行下面这段代码：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,2147483648);//会被理解调用执行
```

运行后可以看到，这段代码是立即被执行的。但如果将延时值修改为小于2147483647毫秒的某个值，那么执行时就没有问题了。

5. 使用setTimeout设置的回调函数中的this不符合直觉

如果被setTimeout推迟执行的回调函数是某个对象的方法，那么该方法中的this关键字将指向全局环境，而不是定义时所在的那个对象。这点在前面介绍this的时候也提过，你可以看下面这段代码的执行结果：

```
var name= 1;
var MyObj = {
  name: 2,
  showName: function(){
    console.log(this.name);
  }
}
setTimeout(MyObj.showName,1000)
```

这里输出的是1，因为这段代码在编译的时候，执行上下文中的this会被设置为全局window，如果是严格模式，会被设置为undefined。

那么该怎么解决这个问题呢？通常可以使用下面这两种方法。

第一种是将`MyObj.showName`放在匿名函数中执行，如下所示：

```
//箭头函数
setTimeout(() => {
    MyObj.showName()
}, 1000);
//或者function函数
setTimeout(function() {
  MyObj.showName();
}, 1000)
```

第二种是使用bind方法，将showName绑定在MyObj上面，代码如下所示：

```
setTimeout(MyObj.showName.bind(MyObj), 1000)
```

总结

好了，今天我们就介绍到这里，下面我来总结下今天的内容。

- 首先，为了支持定时器的实现，浏览器增加了延时队列。
- 其次，由于消息队列排队和一些系统级别的限制，通过setTimeout设置的回调任务并非总是可以实时地被执行，这样就不能满足一些实时性要求较高的需求了。
- 最后，在定时器中使用过程中，还存在一些陷阱，需要你多加注意。

通过分析和讲解，你会发现函数setTimeout在时效性上面有很多先天的不足，所以对于一些时间精度要求比较高的需求，应该有针对性地采取一些其他的方案。

思考时间

今天我们介绍了setTimeout，相信你现在也知道它是怎么工作的了，不过由于使用setTimeout设置的回调任务实时性并不是太好，所以很多场景并不适合使用setTimeout。比如你要使用JavaScript来实现动画效果，函数requestAnimationFrame就是个很好的选择。

那么今天留给你的作业是：你需要网上搜索了解下requestAnimationFrame的工作机制，并对比setTimeout，然后分析出requestAnimationFrame实现的动画效果比setTimeout好的原因。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。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。

在上一篇文章中我们介绍了页面中的事件和消息队列，知道了**浏览器页面是由消息队列和事件循环系统来驱动的**。

那在接下来的两篇文章中，我会通过**setTimeout**和**XMLHttpRequest**这两个WebAPI来介绍事件循环的应用。这两个WebAPI是两种不同类型的应用，比较典型，并且在JavaScript中的使用频率非常高。你可能觉得它们太简单、太基础，但有时候恰恰是基础简单的东西才最重要，了解它们是如何工作的会有助于你写出更加高效的前端代码。

本篇文章主要介绍的是**setTimeout**。其实说起setTimeout方法，从事开发的同学想必都不会陌生，它就是一个**定时器，用来指定某个函数在多少毫秒之后执行**。它会返回一个整数，表示定时器的编号，同时你还可以通过该编号来取消这个定时器。下面的示例代码就演示了定时器最基础的使用方式：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,200);
```

执行上述代码，输出的结果也很明显，通过setTimeout指定在200毫秒之后调用showName函数，并输出“极客时间”四个字。

简单了解了setTimeout的使用方法后，那接下来我们就来看看浏览器是如何实现定时器的，然后再介绍下定时器在使用过程中的一些注意事项。

浏览器怎么实现setTimeout

要了解定时器的工作原理，就得先来回顾下之前讲的事件循环系统，我们知道渲染进程中所有运行在主线程上的任务都需要先添加到消息队列，然后事件循环系统再按照顺序执行消息队列中的任务。下面我们来看看那些典型的事件：

- 当接收到HTML文档数据，渲染引擎就会将“解析DOM”事件添加到消息队列中，
- 当用户改变了Web页面的窗口大小，渲染引擎就会将“重新布局”的事件添加到消息队列中。
- 当触发了JavaScript引擎垃圾回收机制，渲染引擎会将“垃圾回收”任务添加到消息队列中。
- 同样，如果要执行一段异步JavaScript代码，也是需要将执行任务添加到消息队列中。

以上列举的只是一小部分事件，这些事件被添加到消息队列之后，事件循环系统就会按照消息队列中的顺序来执行事件。

所以说要执行一段异步任务，需要先将任务添加到消息队列中。不过通过定时器设置回调函数有点特别，它们需要在指定的时间间隔内被调用，但消息队列中的任务是按照顺序执行的，所以为了保证回调函数能在指定时间内执行，你不能将定时器的回调函数直接添加到消息队列中。

那么该怎么设计才能让定时器设置的回调事件在规定时间内被执行呢？你也可以思考下，如果让你在消息循环系统的基础之上加上定时器的功能，你会如何设计？

在Chrome中除了正常使用的消息队列之外，还有另外一个消息队列，这个队列中维护了需要延迟执行的任务列表，包括了定时器和Chromium内部一些需要延迟执行的任务。所以当通过JavaScript创建一个定时器时，渲染进程会将该定时器的回调任务添加到延迟队列中。

如果感兴趣，你可以参考Chromium中关于队列部分的源码。

源码中延迟执行队列的定义如下所示：

```
 DelayedIncomingQueue delayed_incoming_queue;
```

当通过JavaScript调用setTimeout设置回调函数的时候，渲染进程将会创建一个回调任务，包含了回调函数showName、当前发起时间、延迟执行时间，其模拟代码如下所示：

```
struct DelayTask{
  int64 id;
  CallBackFunction cbf;
  int start_time;
  int delay_time;
};
DelayTask timerTask;
timerTask.cbf = showName;
timerTask.start_time = getCurrentTime(); //获取当前时间
timerTask.delay_time = 200;//设置延迟执行时间
```

创建好回调任务之后，再将该任务添加到延迟执行队列中，代码如下所示：

```
delayed_incoming_queue.push(timerTask);
```

现在通过定时器发起的任务就被保存到延迟队列中了，那接下来我们再来看看消息循环系统是怎么触发延迟队列的。

我们可以来完善上一篇文章中消息循环的代码，在其中加入执行延迟队列的代码，如下所示：

```
void ProcessTimerTask(){
  //从delayed_incoming_queue中取出已经到期的定时器任务
  //依次执行这些任务
}

TaskQueue task_queue;
void ProcessTask();
bool keep_running = true;
void MainTherad(){
  for(;;){
    //执行消息队列中的任务
    Task task = task_queue.takeTask();
    ProcessTask(task);

    //执行延迟队列中的任务
    ProcessDelayTask()

    if(!keep_running) //如果设置了退出标志，那么直接退出线程循环
        break;
  }
}
```

从上面代码可以看出来，我们添加了一个**ProcessDelayTask函数**，该函数是专门用来处理延迟执行任务的。这里我们要重点关注它的执行时机，在上段代码中，处理完消息队列中的一个任务之后，就开始执行ProcessDelayTask函数。ProcessDelayTask函数会根据发起时间和延迟时间计算出到期的任务，然后依次执行这些到期的任务。等到期的任务执行完成之后，再继续下一个循环过程。通过这样的方式，一个完整的定时器就实现了。

设置一个定时器，JavaScript引擎会返回一个定时器的ID。那通常情况下，当一个定时器的任务还没有被执行的时候，也是可以取消的，具体方法是调用**clearTimeout函数**，并传入需要取消的定时器的ID。如下面代码所示：

```
clearTimeout(timer_id)
```

其实浏览器内部实现取消定时器的操作也是非常简单的，就是直接从delayed_incoming_queue延迟队列中，通过ID查找到对应的任务，然后再将其从队列中删除掉就可以了。

使用setTimeout的一些注意事项

现在你应该知道在浏览器内部定时器是如何工作的了。不过在使用定时器的过程中，如果你不了解定时器的一些细节，那么很有可能掉进定时器的一些陷阱里。所以接下来，我们就来讲解一下在使用定时器过程中存在的那些陷阱。

1. 如果当前任务执行时间过久，会影响定时器任务的执行

在使用setTimeout的时候，有很多因素会导致回调函数执行比设定的预期值要久，其中一个就是当前任务执行时间过久从而导致定时器设置的任务被延后执行。我们先看下面这段代码：

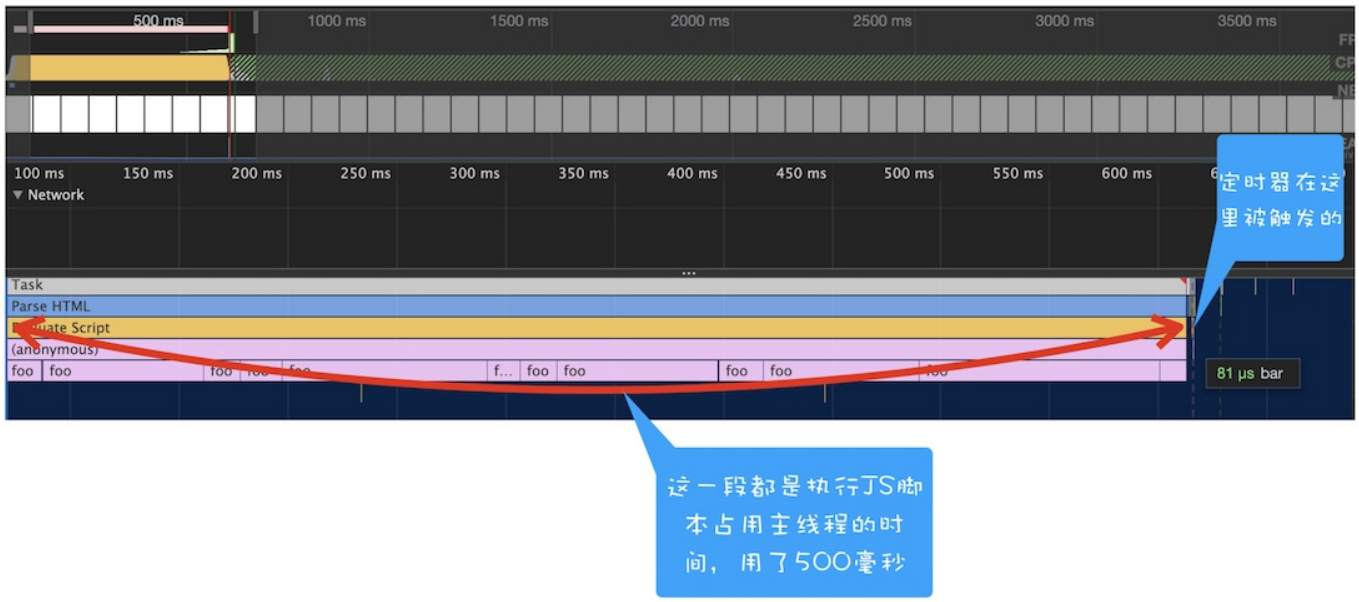
```
function bar() {
```

```
    console.log('bar')
}
function foo() {
    setTimeout(bar, 0);
    for (let i = 0; i < 5000; i++) {
        let i = 5+8+8+8
        console.log(i)
    }
}
foo()
```

这段代码中，在执行foo函数的时候使用setTimeout设置了一个0延时的回调任务，设置好回调任务后，foo函数会继续执行5000次for循环。

通过setTimeout设置的回调任务被放入了消息队列中并且等待下一次执行，这里并不是立即执行的；要执行消息队列中的下个任务，需要等待当前的任务执行完成，由于当前这段代码要执行5000次的for循环，所以当前这个任务的执行时间会比较久一点。这势必会影响到下个任务的执行时间。

你也可以打开Performance来看看其执行过程，如下图所示：



长任务导致定时器被延后执行

从图中可以看到，执行foo函数所消耗的时长是500毫秒，这也就意味着通过setTimeout设置的任务会被推迟到500毫秒以后再去执行，而设置setTimeout的回调延迟时间是0。

2. 如果setTimeout存在嵌套调用，那么系统会设置最短时间间隔为4毫秒

也就是说在定时器函数里面嵌套调用定时器，也会延长定时器的执行时间，可以先看下面的这段代码：

```
function cb() { setTimeout(cb, 0); }
setTimeout(cb, 0);
```

上述这段代码你有没有看出存在什么问题？

你还是可以通过Performance来记录下这段代码的执行过程，如下图所示：

循环嵌套调用setTimeout

上图中的竖线就是定时器的函数回调过程，从图中可以看出，前面五次调用的时间间隔比较小，嵌套调用超过五次以上，后面每次的调用最小时间间隔是4毫秒。之所以出现这样的情况，是因为在Chrome中，定时器被嵌套调用5次以上，系统会判断该函数方法被阻塞了，如果定时器的调用时间间隔小于4毫秒，那么浏览器会将每次调用的时间间隔设置为4毫秒。下面是Chromium实现4毫秒延迟的代码，你可以看下：

```
static const int kMaxTimerNestingLevel = 5;
```

```
// Chromium uses a minimum timer interval of 4ms. We'd like to go
// lower; however, there are poorly coded websites out there which do
// create CPU-spinning loops.  Using 4ms prevents the CPU from
// spinning too busily and provides a balance between CPU spinning and
// the smallest possible interval timer.
static constexpr base::TimeDelta kMinimumInterval = base::TimeDelta::FromMilliseconds(4);

base::TimeDelta interval_milliseconds =
      std::max(base::TimeDelta::FromMilliseconds(1), interval);

  if (interval_milliseconds < kMinimumInterval &&
      nesting_level_ >= kMaxTimerNestingLevel)
    interval_milliseconds = kMinimumInterval;

  if (single_shot)
    StartOneShot(interval_milliseconds, FROM_HERE);
  else
    StartRepeating(interval_milliseconds, FROM_HERE);
```

所以，一些实时性较高的需求就不太适合使用setTimeout了，比如你用setTimeout来实现JavaScript动画就不是一个很好的主意。

3. 未激活的页面，setTimeout执行最小间隔是1000毫秒

除了前面的4毫秒延迟，还有一个很容易被忽略的地方，那就是未被激活的页面中定时器最小值大于1000毫秒，也就是说，如果标签不是当前的激活标签，那么定时器最小的时间间隔是1000毫秒，目的是为了优化后台页面的加载损耗以及降低耗电量。这一点你在使用定时器的时候要注意。

4. 延时执行时间有最大值

除了要了解定时器的回调函数时间比实际设定值要延后之外，还有一点需要注意下，那就是Chrome、Safari、Firefox都是以32个bit来存储延时值的，32bit最大只能存放的数字是2147483647毫秒，这就意味着，如果setTimeout设置的延迟值大于 2147483647毫秒（大约24.8天）时就会溢出，那么相当于延时值被设置为0了，这导致定时器会被立即执行。你可以运行下面这段代码：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,2147483648);//会被理解调用执行
```

运行后可以看到，这段代码是立即被执行的。但如果将延时值修改为小于2147483647毫秒的某个值，那么执行时就没有问题了。

5. 使用setTimeout设置的回调函数中的this不符合直觉

如果被setTimeout推迟执行的回调函数是某个对象的方法，那么该方法中的this关键字将指向全局环境，而不是定义时所在的那个对象。这点在前面介绍this的时候也提过，你可以看下面这段代码的执行结果：

```
var name= 1;
var MyObj = {
  name: 2,
  showName: function(){
    console.log(this.name);
  }
}
setTimeout(MyObj.showName,1000)
```

这里输出的是1，因为这段代码在编译的时候，执行上下文中的this会被设置为全局window，如果是严格模式，会被设置为undefined。

那么该怎么解决这个问题呢？通常可以使用下面这两种方法。

第一种是将`MyObj.showName`放在匿名函数中执行，如下所示：

```
//箭头函数
setTimeout(() => {
    MyObj.showName()
}, 1000);
//或者function函数
setTimeout(function() {
  MyObj.showName();
}, 1000)
```

第二种是使用bind方法，将showName绑定在MyObj上面，代码如下所示：

```
setTimeout(MyObj.showName.bind(MyObj), 1000)
```

总结

好了，今天我们就介绍到这里，下面我来总结下今天的内容。

- 首先，为了支持定时器的实现，浏览器增加了延时队列。
- 其次，由于消息队列排队和一些系统级别的限制，通过setTimeout设置的回调任务并非总是可以实时地被执行，这样就不能满足一些实时性要求较高的需求了。
- 最后，在定时器中使用过程中，还存在一些陷阱，需要你多加注意。

通过分析和讲解，你会发现函数setTimeout在时效性上面有很多先天的不足，所以对于一些时间精度要求比较高的需求，应该有针对性地采取一些其他的方案。

思考时间

今天我们介绍了setTimeout，相信你现在也知道它是怎么工作的了，不过由于使用setTimeout设置的回调任务实时性并不是太好，所以很多场景并不适合使用setTimeout。比如你要使用JavaScript来实现动画效果，函数requestAnimationFrame就是个很好的选择。

那么今天留给你的作业是：你需要网上搜索了解下requestAnimationFrame的工作机制，并对比setTimeout，然后分析出requestAnimationFrame实现的动画效果比setTimeout好的原因。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。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。

在上一篇文章中我们介绍了页面中的事件和消息队列，知道了**浏览器页面是由消息队列和事件循环系统来驱动的**。

那在接下来的两篇文章中，我会通过**setTimeout**和**XMLHttpRequest**这两个WebAPI来介绍事件循环的应用。这两个WebAPI是两种不同类型的应用，比较典型，并且在JavaScript中的使用频率非常高。你可能觉得它们太简单、太基础，但有时候恰恰是基础简单的东西才最重要，了解它们是如何工作的会有助于你写出更加高效的前端代码。

本篇文章主要介绍的是**setTimeout**。其实说起setTimeout方法，从事开发的同学想必都不会陌生，它就是一个**定时器，用来指定某个函数在多少毫秒之后执行**。它会返回一个整数，表示定时器的编号，同时你还可以通过该编号来取消这个定时器。下面的示例代码就演示了定时器最基础的使用方式：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,200);
```

执行上述代码，输出的结果也很明显，通过setTimeout指定在200毫秒之后调用showName函数，并输出“极客时间”四个字。

简单了解了setTimeout的使用方法后，那接下来我们就来看看浏览器是如何实现定时器的，然后再介绍下定时器在使用过程中的一些注意事项。

浏览器怎么实现setTimeout

要了解定时器的工作原理，就得先来回顾下之前讲的事件循环系统，我们知道渲染进程中所有运行在主线程上的任务都需要先添加到消息队列，然后事件循环系统再按照顺序执行消息队列中的任务。下面我们来看看那些典型的事件：

- 当接收到HTML文档数据，渲染引擎就会将“解析DOM”事件添加到消息队列中，
- 当用户改变了Web页面的窗口大小，渲染引擎就会将“重新布局”的事件添加到消息队列中。
- 当触发了JavaScript引擎垃圾回收机制，渲染引擎会将“垃圾回收”任务添加到消息队列中。
- 同样，如果要执行一段异步JavaScript代码，也是需要将执行任务添加到消息队列中。

以上列举的只是一小部分事件，这些事件被添加到消息队列之后，事件循环系统就会按照消息队列中的顺序来执行事件。

所以说要执行一段异步任务，需要先将任务添加到消息队列中。不过通过定时器设置回调函数有点特别，它们需要在指定的时间间隔内被调用，但消息队列中的任务是按照顺序执行的，所以为了保证回调函数能在指定时间内执行，你不能将定时器的回调函数直接添加到消息队列中。

那么该怎么设计才能让定时器设置的回调事件在规定时间内被执行呢？你也可以思考下，如果让你在消息循环系统的基础之上加上定时器的功能，你会如何设计？

在Chrome中除了正常使用的消息队列之外，还有另外一个消息队列，这个队列中维护了需要延迟执行的任务列表，包括了定时器和Chromium内部一些需要延迟执行的任务。所以当通过JavaScript创建一个定时器时，渲染进程会将该定时器的回调任务添加到延迟队列中。

如果感兴趣，你可以参考Chromium中关于队列部分的源码。

源码中延迟执行队列的定义如下所示：

```
 DelayedIncomingQueue delayed_incoming_queue;
```

当通过JavaScript调用setTimeout设置回调函数的时候，渲染进程将会创建一个回调任务，包含了回调函数showName、当前发起时间、延迟执行时间，其模拟代码如下所示：

```
struct DelayTask{
  int64 id;
  CallBackFunction cbf;
  int start_time;
  int delay_time;
};
DelayTask timerTask;
timerTask.cbf = showName;
timerTask.start_time = getCurrentTime(); //获取当前时间
timerTask.delay_time = 200;//设置延迟执行时间
```

创建好回调任务之后，再将该任务添加到延迟执行队列中，代码如下所示：

```
delayed_incoming_queue.push(timerTask);
```

现在通过定时器发起的任务就被保存到延迟队列中了，那接下来我们再来看看消息循环系统是怎么触发延迟队列的。

我们可以来完善上一篇文章中消息循环的代码，在其中加入执行延迟队列的代码，如下所示：

```
void ProcessTimerTask(){
  //从delayed_incoming_queue中取出已经到期的定时器任务
  //依次执行这些任务
}

TaskQueue task_queue;
void ProcessTask();
bool keep_running = true;
void MainTherad(){
  for(;;){
    //执行消息队列中的任务
    Task task = task_queue.takeTask();
    ProcessTask(task);

    //执行延迟队列中的任务
    ProcessDelayTask()

    if(!keep_running) //如果设置了退出标志，那么直接退出线程循环
        break;
  }
}
```

从上面代码可以看出来，我们添加了一个**ProcessDelayTask函数**，该函数是专门用来处理延迟执行任务的。这里我们要重点关注它的执行时机，在上段代码中，处理完消息队列中的一个任务之后，就开始执行ProcessDelayTask函数。ProcessDelayTask函数会根据发起时间和延迟时间计算出到期的任务，然后依次执行这些到期的任务。等到期的任务执行完成之后，再继续下一个循环过程。通过这样的方式，一个完整的定时器就实现了。

设置一个定时器，JavaScript引擎会返回一个定时器的ID。那通常情况下，当一个定时器的任务还没有被执行的时候，也是可以取消的，具体方法是调用**clearTimeout函数**，并传入需要取消的定时器的ID。如下面代码所示：

```
clearTimeout(timer_id)
```

其实浏览器内部实现取消定时器的操作也是非常简单的，就是直接从delayed_incoming_queue延迟队列中，通过ID查找到对应的任务，然后再将其从队列中删除掉就可以了。

使用setTimeout的一些注意事项

现在你应该知道在浏览器内部定时器是如何工作的了。不过在使用定时器的过程中，如果你不了解定时器的一些细节，那么很有可能掉进定时器的一些陷阱里。所以接下来，我们就来讲解一下在使用定时器过程中存在的那些陷阱。

1. 如果当前任务执行时间过久，会影响定时器任务的执行

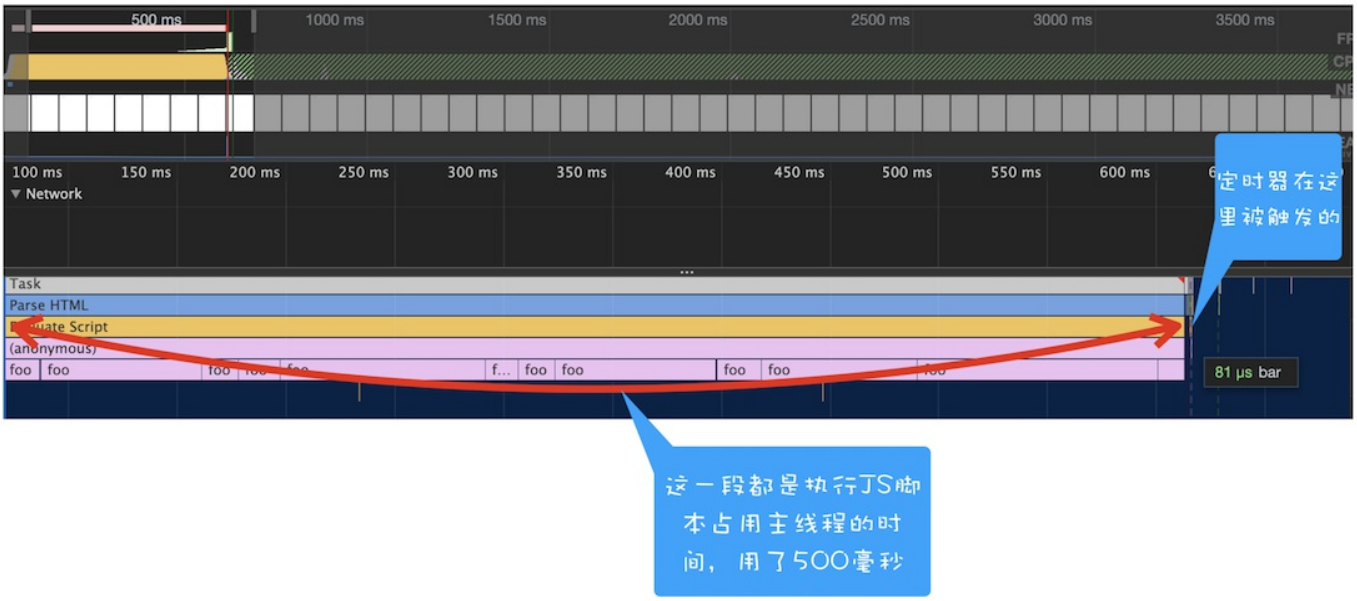
在使用setTimeout的时候，有很多因素会导致回调函数执行比设定的预期值要久，其中一个就是当前任务执行时间过久从而导致定时器设置的任务被延后执行。我们先看下面这段代码：

```
function bar() {
    console.log('bar')
}
function foo() {
    setTimeout(bar, 0);
    for (let i = 0; i < 5000; i++) {
        let i = 5+8+8+8
        console.log(i)
    }
}
foo()
```

这段代码中，在执行foo函数的时候使用setTimeout设置了一个0延时的回调任务，设置好回调任务后，foo函数会继续执行5000次for循环。

通过setTimeout设置的回调任务被放入了消息队列中并且等待下一次执行，这里并不是立即执行的；要执行消息队列中的下个任务，需要等待当前的任务执行完成，由于当前这段代码要执行5000次的for循环，所以当前这个任务的执行时间会比较久一点。这势必会影响到下个任务的执行时间。

你也可以打开Performance来看看其执行过程，如下图所示：



长任务导致定时器被延后执行

从图中可以看到，执行foo函数所消耗的时长是500毫秒，这也就意味着通过setTimeout设置的任务会被推迟到500毫秒以后再去执行，而设置setTimeout的回调延迟时间是0。

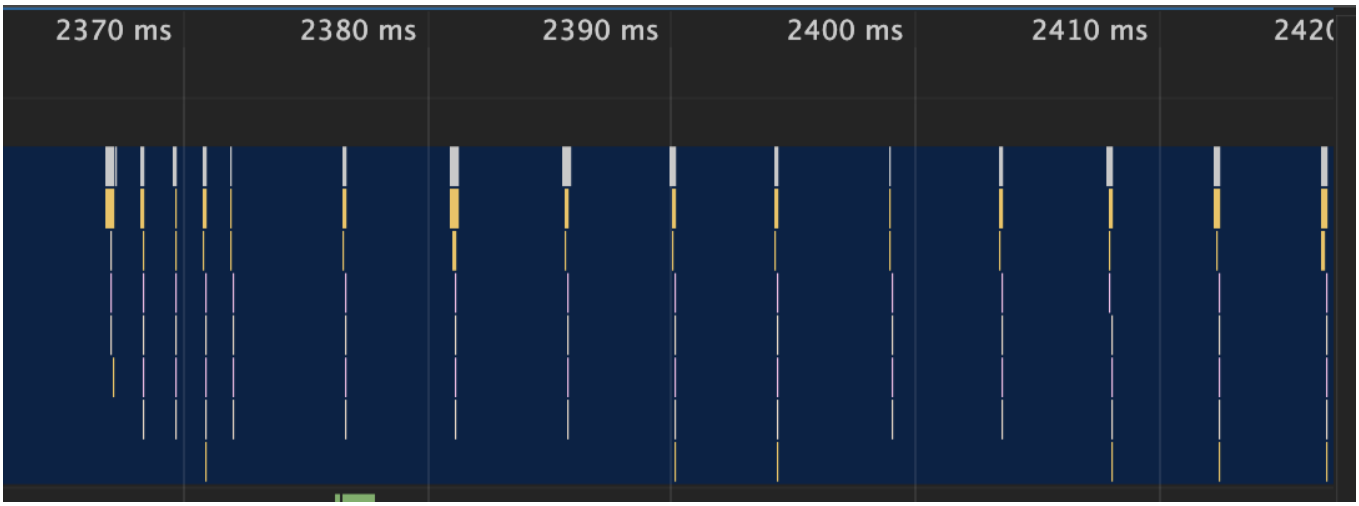
2. 如果setTimeout存在嵌套调用，那么系统会设置最短时间间隔为4毫秒

也就是说在定时器函数里面嵌套调用定时器，也会延长定时器的执行时间，可以先看下面的这段代码：

```
function cb() { setTimeout(cb, 0); }
setTimeout(cb, 0);
```

上述这段代码你有没有看出存在什么问题？

你还是可以通过Performance来记录下这段代码的执行过程，如下图所示：



循环嵌套调用setTimeout

上图中的竖线就是定时器的函数回调过程，从图中可以看出，前面五次调用的时间间隔比较小，嵌套调用超过五次以上，后面每次的调用最小时间间隔是4毫秒。之所以出现这样的情况，是因为在Chrome中，定时器被嵌套调用5次以上，系统会判断该函数方法被阻塞了，如果定时器的调用时间间隔小于4毫秒，那么浏览器会将每次调用的时间间隔设置为4毫秒。下面是Chromium实现4毫秒延迟的代码，你可以看下：

```
static const int kMaxTimerNestingLevel = 5;

// Chromium uses a minimum timer interval of 4ms. We'd like to go
// lower; however, there are poorly coded websites out there which do
// create CPU-spinning loops.  Using 4ms prevents the CPU from
// spinning too busily and provides a balance between CPU spinning and
// the smallest possible interval timer.
static constexpr base::TimeDelta kMinimumInterval = base::TimeDelta::FromMilliseconds(4);

base::TimeDelta interval_milliseconds =
      std::max(base::TimeDelta::FromMilliseconds(1), interval);

  if (interval_milliseconds < kMinimumInterval &&
      nesting_level_ >= kMaxTimerNestingLevel)
    interval_milliseconds = kMinimumInterval;

  if (single_shot)
    StartOneShot(interval_milliseconds, FROM_HERE);
  else
    StartRepeating(interval_milliseconds, FROM_HERE);
```

所以，一些实时性较高的需求就不太适合使用setTimeout了，比如你用setTimeout来实现JavaScript动画就不是一个很好的主意。

3. 未激活的页面，setTimeout执行最小间隔是1000毫秒

除了前面的4毫秒延迟，还有一个很容易被忽略的地方，那就是未被激活的页面中定时器最小值大于1000毫秒，也就是说，如果标签不是当前的激活标签，那么定时器最小的时间间隔是1000毫秒，目的是为了优化后台页面的加载损耗以及降低耗电量。这一点你在使用定时器的时候要注意。

4. 延时执行时间有最大值

除了要了解定时器的回调函数时间比实际设定值要延后之外，还有一点需要注意下，那就是Chrome、Safari、Firefox都是以32个bit来存储延时值的，32bit最大只能存放的数字是2147483647毫秒，这就意味着，如果setTimeout设置的延迟值大于 2147483647毫秒（大约24.8天）时就会溢出，那么相当于延时值被设置为0了，这导致定时器会被立即执行。你可以运行下面这段代码：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,2147483648);//会被理解调用执行
```

运行后可以看到，这段代码是立即被执行的。但如果将延时值修改为小于2147483647毫秒的某个值，那么执行时就没有问题了。

5. 使用setTimeout设置的回调函数中的this不符合直觉

如果被setTimeout推迟执行的回调函数是某个对象的方法，那么该方法中的this关键字将指向全局环境，而不是定义时所在的那个对象。这点在前面介绍this的时候也提过，你可以看下面这段代码的执行结果：

```
var name= 1;
var MyObj = {
  name: 2,
  showName: function(){
    console.log(this.name);
  }
}
setTimeout(MyObj.showName,1000)
```

这里输出的是1，因为这段代码在编译的时候，执行上下文中的this会被设置为全局window，如果是严格模式，会被设置为undefined。

那么该怎么解决这个问题呢？通常可以使用下面这两种方法。

第一种是将MyObj.showName放在匿名函数中执行，如下所示：

```
//箭头函数
setTimeout(() => {
    MyObj.showName()
}, 1000);
//或者function函数
setTimeout(function() {
```

```
  MyObj.showName();
}, 1000)
```

第二种是使用bind方法，将showName绑定在MyObj上面，代码如下所示：

```
setTimeout(MyObj.showName.bind(MyObj), 1000)
```

总结

好了，今天我们就介绍到这里，下面我来总结下今天的内容。

- 首先，为了支持定时器的实现，浏览器增加了延时队列。
- 其次，由于消息队列排队和一些系统级别的限制，通过setTimeout设置的回调任务并非总是可以实时地被执行，这样就不能满足一些实时性要求较高的需求了。
- 最后，在定时器中使用过程中，还存在一些陷阱，需要你多加注意。

通过分析和讲解，你会发现函数setTimeout在时效性上面有很多先天的不足，所以对于一些时间精度要求比较高的需求，应该有针对性地采取一些其他的方案。

思考时间

今天我们介绍了setTimeout，相信你现在也知道它是怎么工作的了，不过由于使用setTimeout设置的回调任务实时性并不是太好，所以很多场景并不适合使用setTimeout。比如你要使用JavaScript来实现动画效果，函数requestAnimationFrame就是个很好的选择。

那么今天留给你的作业是：你需要网上搜索了解下requestAnimationFrame的工作机制，并对比setTimeout，然后分析出requestAnimationFrame实现的动画效果比setTimeout好的原因。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。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。

在上一篇文章中我们介绍了页面中的事件和消息队列，知道了**浏览器页面是由消息队列和事件循环系统来驱动的**。

那在接下来的两篇文章中，我会通过**setTimeout**和**XMLHttpRequest**这两个WebAPI来介绍事件循环的应用。这两个WebAPI是两种不同类型的应用，比较典型，并且在JavaScript中的使用频率非常高。你可能觉得它们太简单、太基础，但有时候恰恰是基础简单的东西才最重要，了解它们是如何工作的会有助于你写出更加高效的前端代码。

本篇文章主要介绍的是**setTimeout**。其实说起setTimeout方法，从事开发的同学想必都不会陌生，它就是一个**定时器，用来指定某个函数在多少毫秒之后执行**。它会返回一个整数，表示定时器的编号，同时你还可以通过该编号来取消这个定时器。下面的示例代码就演示了定时器最基础的使用方式：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,200);
```

执行上述代码，输出的结果也很明显，通过setTimeout指定在200毫秒之后调用showName函数，并输出“极客时间”四个字。

简单了解了setTimeout的使用方法后，那接下来我们就来看看浏览器是如何实现定时器的，然后再介绍下定时器在使用过程中的一些注意事项。

浏览器怎么实现setTimeout

要了解定时器的工作原理，就得先来回顾下之前讲的事件循环系统，我们知道渲染进程中所有运行在主线程上的任务都需要先添加到消息队列，然后事件循环系统再按照顺序执行消息队列中的任务。下面我们来看看那些典型的事件：

- 当接收到HTML文档数据，渲染引擎就会将“解析DOM”事件添加到消息队列中，
- 当用户改变了Web页面的窗口大小，渲染引擎就会将“重新布局”的事件添加到消息队列中。
- 当触发了JavaScript引擎垃圾回收机制，渲染引擎会将“垃圾回收”任务添加到消息队列中。
- 同样，如果要执行一段异步JavaScript代码，也是需要将执行任务添加到消息队列中。

以上列举的只是一小部分事件，这些事件被添加到消息队列之后，事件循环系统就会按照消息队列中的顺序来执行事件。

所以说要执行一段异步任务，需要先将任务添加到消息队列中。不过通过定时器设置回调函数有点特别，它们需要在指定的时间间隔内被调用，但消息队列中的任务是按照顺序执行的，所以为了保证回调函数能在指定时间内执行，你不能将定时器的回调函数直接添加到消息队列中。

那么该怎么设计才能让定时器设置的回调事件在规定时间内被执行呢？你也可以思考下，如果让你在消息循环系统的基础之上加上定时器的功能，你会如何设计？

在Chrome中除了正常使用的消息队列之外，还有另外一个消息队列，这个队列中维护了需要延迟执行的任务列表，包括了定时器和Chromium内部一些需要延迟执行的任务。所以当通过JavaScript创建一个定时器时，渲染进程会将该定时器的回调任务添加到延迟队列中。

如果感兴趣，你可以参考Chromium中关于队列部分的源码。

源码中延迟执行队列的定义如下所示：

```
 DelayedIncomingQueue delayed_incoming_queue;
```

当通过JavaScript调用setTimeout设置回调函数的时候，渲染进程将会创建一个回调任务，包含了回调函数showName、当前发起时间、延迟执行时间，其模拟代码如下所示：

```
struct DelayTask{
  int64 id;
  CallBackFunction cbf;
  int start_time;
  int delay_time;
};
DelayTask timerTask;
timerTask.cbf = showName;
timerTask.start_time = getCurrentTime(); //获取当前时间
timerTask.delay_time = 200;//设置延迟执行时间
```

创建好回调任务之后，再将该任务添加到延迟执行队列中，代码如下所示：

```
delayed_incoming_queue.push(timerTask);
```

现在通过定时器发起的任务就被保存到延迟队列中了，那接下来我们再来看看消息循环系统是怎么触发延迟队列的。

我们可以来完善上一篇文章中消息循环的代码，在其中加入执行延迟队列的代码，如下所示：

```
void ProcessTimerTask(){
  //从delayed_incoming_queue中取出已经到期的定时器任务
  //依次执行这些任务
}

TaskQueue task_queue;
void ProcessTask();
bool keep_running = true;
void MainTherad(){
  for(;;){
    //执行消息队列中的任务
    Task task = task_queue.takeTask();
    ProcessTask(task);

    //执行延迟队列中的任务
    ProcessDelayTask()

    if(!keep_running) //如果设置了退出标志，那么直接退出线程循环
        break;
  }
}
```

从上面代码可以看出来，我们添加了一个**ProcessDelayTask函数**，该函数是专门用来处理延迟执行任务的。这里我们要重点关注它的执行时机，在上段代码中，处理完消息队列中的一个任务之后，就开始执行ProcessDelayTask函数。ProcessDelayTask函数会根据发起时间和延迟时间计算出到期的任务，然后依次执行这些到期的任务。等到期的任务执行完成之后，再继续下一个循环过程。通过这样的方式，一个完整的定时器就实现了。

设置一个定时器，JavaScript引擎会返回一个定时器的ID。那通常情况下，当一个定时器的任务还没有被执行的时候，也是可以取消的，具体方法是调用**clearTimeout函数**，并传入需要取消的定时器的ID。如下面代码所示：

```
clearTimeout(timer_id)
```

其实浏览器内部实现取消定时器的操作也是非常简单的，就是直接从delayed_incoming_queue延迟队列中，通过ID查找到对应的任务，然后再将其从队列中删除掉就可以了。

使用setTimeout的一些注意事项

现在你应该知道在浏览器内部定时器是如何工作的了。不过在使用定时器的过程中，如果你不了解定时器的一些细节，那么很有可能掉进定时器的一些陷阱里。所以接下来，我们就来讲解一下在使用定时器过程中存在的那些陷阱。

1. 如果当前任务执行时间过久，会影响定时器任务的执行

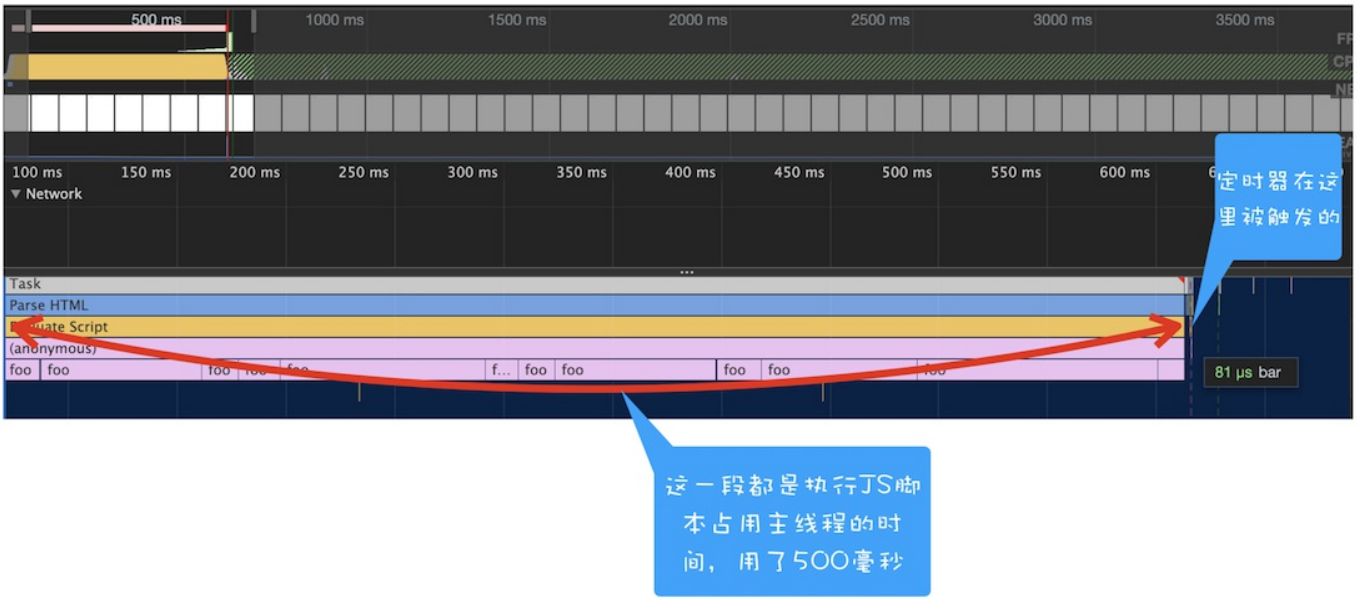
在使用setTimeout的时候，有很多因素会导致回调函数执行比设定的预期值要久，其中一个就是当前任务执行时间过久从而导致定时器设置的任务被延后执行。我们先看下面这段代码：

```
function bar() {
    console.log('bar')
}
function foo() {
    setTimeout(bar, 0);
    for (let i = 0; i < 5000; i++) {
        let i = 5+8+8+8
        console.log(i)
    }
}
foo()
```

这段代码中，在执行foo函数的时候使用setTimeout设置了一个0延时的回调任务，设置好回调任务后，foo函数会继续执行5000次for循环。

通过setTimeout设置的回调任务被放入了消息队列中并且等待下一次执行，这里并不是立即执行的；要执行消息队列中的下个任务，需要等待当前的任务执行完成，由于当前这段代码要执行5000次的for循环，所以当前这个任务的执行时间会比较久一点。这势必会影响到下个任务的执行时间。

你也可以打开Performance来看看其执行过程，如下图所示：



长任务导致定时器被延后执行

从图中可以看到，执行foo函数所消耗的时长是500毫秒，这也就意味着通过setTimeout设置的任务会被推迟到500毫秒以后再去执行，而设置setTimeout的回调延迟时间是0。

2. 如果setTimeout存在嵌套调用，那么系统会设置最短时间间隔为4毫秒

也就是说在定时器函数里面嵌套调用定时器，也会延长定时器的执行时间，可以先看下面的这段代码：

```
function cb() { setTimeout(cb, 0); }
setTimeout(cb, 0);
```

上述这段代码你有没有看出存在什么问题？

你还是可以通过Performance来记录下这段代码的执行过程，如下图所示：

循环嵌套调用setTimeout

上图中的竖线就是定时器的函数回调过程，从图中可以看出，前面五次调用的时间间隔比较小，嵌套调用超过五次以上，后面每次的调用最小时间间隔是4毫秒。之所以出现这样的情况，是因为在Chrome中，定时器被嵌套调用5次以上，系统会判断该函数方法被阻塞了，如果定时器的调用时间间隔小于4毫秒，那么浏览器会将每次调用的时间间隔设置为4毫秒。下面是Chromium实现4毫秒延迟的代码，你可以看下：

```
static const int kMaxTimerNestingLevel = 5;

// Chromium uses a minimum timer interval of 4ms. We'd like to go
// lower; however, there are poorly coded websites out there which do
// create CPU-spinning loops.  Using 4ms prevents the CPU from
// spinning too busily and provides a balance between CPU spinning and
// the smallest possible interval timer.
static constexpr base::TimeDelta kMinimumInterval = base::TimeDelta::FromMilliseconds(4);

base::TimeDelta interval_milliseconds =
      std::max(base::TimeDelta::FromMilliseconds(1), interval);

  if (interval_milliseconds < kMinimumInterval &&
      nesting_level_ >= kMaxTimerNestingLevel)
    interval_milliseconds = kMinimumInterval;

  if (single_shot)
    StartOneShot(interval_milliseconds, FROM_HERE);
  else
    StartRepeating(interval_milliseconds, FROM_HERE);
```

所以，一些实时性较高的需求就不太适合使用setTimeout了，比如你用setTimeout来实现JavaScript动画就不是一个很好的主意。

3. 未激活的页面，setTimeout执行最小间隔是1000毫秒

除了前面的4毫秒延迟，还有一个很容易被忽略的地方，那就是未被激活的页面中定时器最小值大于1000毫秒，也就是说，如果标签不是当前的激活标签，那么定时器最小的时间间隔是1000毫秒，目的是为了优化后台页面的加载损耗以及降低耗电量。这一点你在使用定时器的时候要注意。

4. 延时执行时间有最大值

除了要了解定时器的回调函数时间比实际设定值要延后之外，还有一点需要注意下，那就是Chrome、Safari、Firefox都是以32个bit来存储延时值的，32bit最大只能存放的数字是2147483647毫秒，这就意味着，如果setTimeout设置的延迟值大于 2147483647毫秒（大约24.8天）时就会溢出，那么相当于延时值被设置为0了，这导致定时器会被立即执行。你可以运行下面这段代码：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,2147483648);//会被理解调用执行
```

运行后可以看到，这段代码是立即被执行的。但如果将延时值修改为小于2147483647毫秒的某个值，那么执行时就没有问题了。

5. 使用setTimeout设置的回调函数中的this不符合直觉

如果被setTimeout推迟执行的回调函数是某个对象的方法，那么该方法中的this关键字将指向全局环境，而不是定义时所在的那个对象。这点在前面介绍this的时候也提过，你可以看下面这段代码的执行结果：

```
var name= 1;
var MyObj = {
  name: 2,
  showName: function(){
    console.log(this.name);
  }
```

```
}
setTimeout(MyObj.showName,1000)
```

这里输出的是1，因为这段代码在编译的时候，执行上下文中的this会被设置为全局window，如果是严格模式，会被设置为undefined。

那么该怎么解决这个问题呢？通常可以使用下面这两种方法。

第一种是将MyObj.showName放在匿名函数中执行，如下所示：

```
//箭头函数
setTimeout(() => {
    MyObj.showName()
}, 1000);
//或者function函数
setTimeout(function() {
  MyObj.showName();
}, 1000)
```

第二种是使用bind方法，将showName绑定在MyObj上面，代码如下所示：

```
setTimeout(MyObj.showName.bind(MyObj), 1000)
```

总结

好了，今天我们就介绍到这里，下面我来总结下今天的内容。

- 首先，为了支持定时器的实现，浏览器增加了延时队列。
- 其次，由于消息队列排队和一些系统级别的限制，通过setTimeout设置的回调任务并非总是可以实时地被执行，这样就不能满足一些实时性要求较高的需求了。
- 最后，在定时器中使用过程中，还存在一些陷阱，需要你多加注意。

通过分析和讲解，你会发现函数setTimeout在时效性上面有很多先天的不足，所以对于一些时间精度要求比较高的需求，应该有针对性地采取一些其他的方案。

思考时间

今天我们介绍了setTimeout，相信你现在也知道它是怎么工作的了，不过由于使用setTimeout设置的回调任务实时性并不是太好，所以很多场景并不适合使用setTimeout。比如你要使用JavaScript来实现动画效果，函数requestAnimationFrame就是个很好的选择。

那么今天留给你的作业是：你需要网上搜索了解下requestAnimationFrame的工作机制，并对比setTimeout，然后分析出requestAnimationFrame实现的动画效果比setTimeout好的原因。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。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。

在上一篇文章中我们介绍了页面中的事件和消息队列，知道了**浏览器页面是由消息队列和事件循环系统来驱动的**。

那在接下来的两篇文章中，我会通过**setTimeout**和**XMLHttpRequest**这两个WebAPI来介绍事件循环的应用。这两个WebAPI是两种不同类型的应用，比较典型，并且在JavaScript中的使用频率非常高。你可能觉得它们太简单、太基础，但有时候恰恰是基础简单的东西才最重要，了解它们是如何工作的会有助于你写出更加高效的前端代码。

本篇文章主要介绍的是**setTimeout**。其实说起setTimeout方法，从事开发的同学想必都不会陌生，它就是一个**定时器，用来指定某个函数在多少毫秒之后执行**。它会返回一个整数，表示定时器的编号，同时你还可以通过该编号来取消这个定时器。下面的示例代码就演示了定时器最基础的使用方式：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,200);
```

执行上述代码，输出的结果也很明显，通过setTimeout指定在200毫秒之后调用showName函数，并输出“极客时间”四个字。

简单了解了setTimeout的使用方法后，那接下来我们就来看看浏览器是如何实现定时器的，然后再介绍下定时器在使用过程中的一些注意事项。

浏览器怎么实现setTimeout

要了解定时器的工作原理，就得先来回顾下之前讲的事件循环系统，我们知道渲染进程中所有运行在主线程上的任务都需要先添加到消息队列，然后事件循环系统再按照顺序执行消息队列中的任务。下面我们来看看那些典型的事件：

- 当接收到HTML文档数据，渲染引擎就会将“解析DOM”事件添加到消息队列中，
- 当用户改变了Web页面的窗口大小，渲染引擎就会将“重新布局”的事件添加到消息队列中。
- 当触发了JavaScript引擎垃圾回收机制，渲染引擎会将“垃圾回收”任务添加到消息队列中。
- 同样，如果要执行一段异步JavaScript代码，也是需要将执行任务添加到消息队列中。

以上列举的只是一小部分事件，这些事件被添加到消息队列之后，事件循环系统就会按照消息队列中的顺序来执行事件。

所以说要执行一段异步任务，需要先将任务添加到消息队列中。不过通过定时器设置回调函数有点特别，它们需要在指定的时间间隔内被调用，但消息队列中的任务是按照顺序执行的，所以为了保证回调函数能在指定时间内执行，你不能将定时器的回调函数直接添加到消息队列中。

那么该怎么设计才能让定时器设置的回调事件在规定时间内被执行呢？你也可以思考下，如果让你在消息循环系统的基础之上加上定时器的功能，你会如何设计？

在Chrome中除了正常使用的消息队列之外，还有另外一个消息队列，这个队列中维护了需要延迟执行的任务列表，包括了定时器和Chromium内部一些需要延迟执行的任务。所以当通过JavaScript创建一个定时器时，渲染进程会将该定时器的回调任务添加到延迟队列中。

如果感兴趣，你可以参考Chromium中关于队列部分的源码。

源码中延迟执行队列的定义如下所示：

```
 DelayedIncomingQueue delayed_incoming_queue;
```

当通过JavaScript调用setTimeout设置回调函数的时候，渲染进程将会创建一个回调任务，包含了回调函数showName、当前发起时间、延迟执行时间，其模拟代码如下所示：

```
struct DelayTask{
  int64 id;
  CallBackFunction cbf;
  int start_time;
  int delay_time;
};
DelayTask timerTask;
timerTask.cbf = showName;
timerTask.start_time = getCurrentTime(); //获取当前时间
timerTask.delay_time = 200;//设置延迟执行时间
```

创建好回调任务之后，再将该任务添加到延迟执行队列中，代码如下所示：

```
delayed_incoming_queue.push(timerTask);
```

现在通过定时器发起的任务就被保存到延迟队列中了，那接下来我们再来看看消息循环系统是怎么触发延迟队列的。

我们可以来完善上一篇文章中消息循环的代码，在其中加入执行延迟队列的代码，如下所示：

```
void ProcessTimerTask(){
  //从delayed_incoming_queue中取出已经到期的定时器任务
  //依次执行这些任务
}

TaskQueue task_queue;
void ProcessTask();
bool keep_running = true;
void MainTherad(){
  for(;;){
    //执行消息队列中的任务
    Task task = task_queue.takeTask();
    ProcessTask(task);

    //执行延迟队列中的任务
    ProcessDelayTask()

    if(!keep_running) //如果设置了退出标志，那么直接退出线程循环
        break;
  }
}
```

从上面代码可以看出来，我们添加了一个**ProcessDelayTask函数**，该函数是专门用来处理延迟执行任务的。这里我们要重点关注它的执行时机，在上段代码中，处理完消息队列中的一个任务之后，就开始执行ProcessDelayTask函数。ProcessDelayTask函数会根据发起时间和延迟时间计算出到期的任务，然后依次执行这些到期的任务。等到期的任务执行完成之后，再继续下一个循环过程。通过这样的方式，一个完整的定时器就实现了。

设置一个定时器，JavaScript引擎会返回一个定时器的ID。那通常情况下，当一个定时器的任务还没有被执行的时候，也是可以取消的，具体方法是调用**clearTimeout函数**，并传入需要取消的定时器的ID。如下面代码所示：

```
clearTimeout(timer_id)
```

其实浏览器内部实现取消定时器的操作也是非常简单的，就是直接从delayed_incoming_queue延迟队列中，通过ID查找到对应的任务，然后再将其从队列中删除掉就可以了。

使用setTimeout的一些注意事项

现在你应该知道在浏览器内部定时器是如何工作的了。不过在使用定时器的过程中，如果你不了解定时器的一些细节，那么很有可能掉进定时器的一些陷阱里。所以接下来，我们就来讲解一下在使用定时器过程中存在的那些陷阱。

1. 如果当前任务执行时间过久，会影响定时器任务的执行

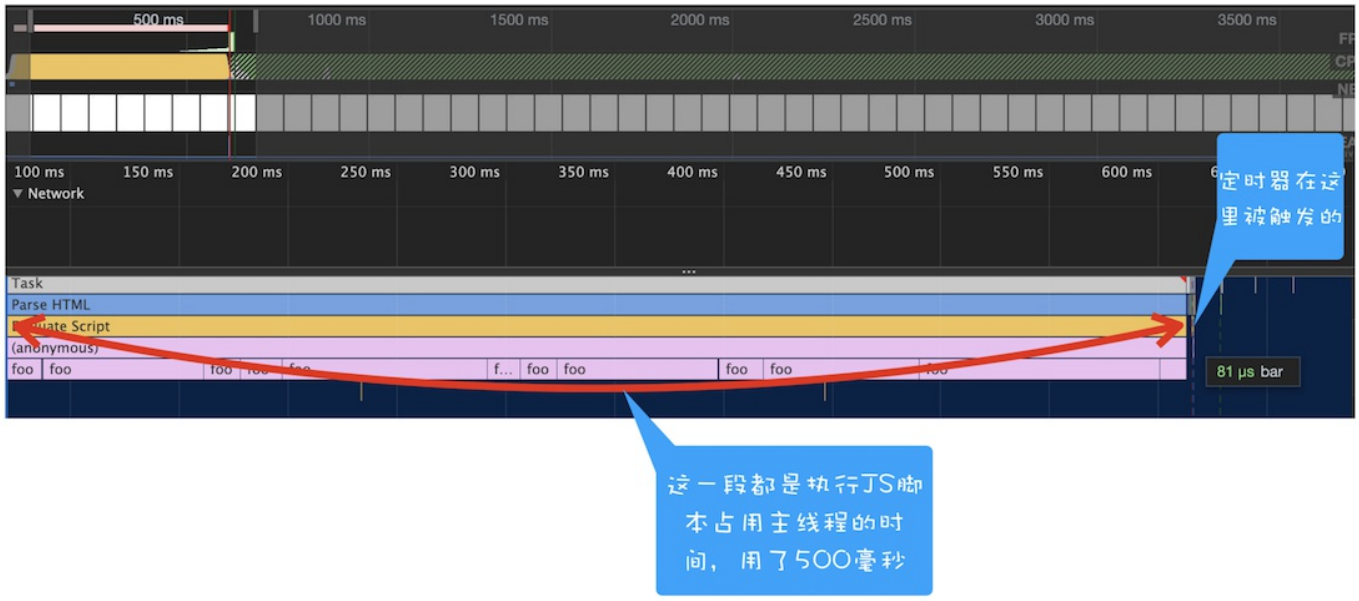
在使用setTimeout的时候，有很多因素会导致回调函数执行比设定的预期值要久，其中一个就是当前任务执行时间过久从而导致定时器设置的任务被延后执行。我们先看下面这段代码：

```
function bar() {
    console.log('bar')
}
function foo() {
    setTimeout(bar, 0);
    for (let i = 0; i < 5000; i++) {
        let i = 5+8+8+8
        console.log(i)
    }
}
foo()
```

这段代码中，在执行foo函数的时候使用setTimeout设置了一个0延时的回调任务，设置好回调任务后，foo函数会继续执行5000次for循环。

通过setTimeout设置的回调任务被放入了消息队列中并且等待下一次执行，这里并不是立即执行的；要执行消息队列中的下个任务，需要等待当前的任务执行完成，由于当前这段代码要执行5000次的for循环，所以当前这个任务的执行时间会比较久一点。这势必会影响到下个任务的执行时间。

你也可以打开Performance来看看其执行过程，如下图所示：



长任务导致定时器被延后执行

从图中可以看到，执行foo函数所消耗的时长是500毫秒，这也就意味着通过setTimeout设置的任务会被推迟到500毫秒以后再去执行，而设置setTimeout的回调延迟时间是0。

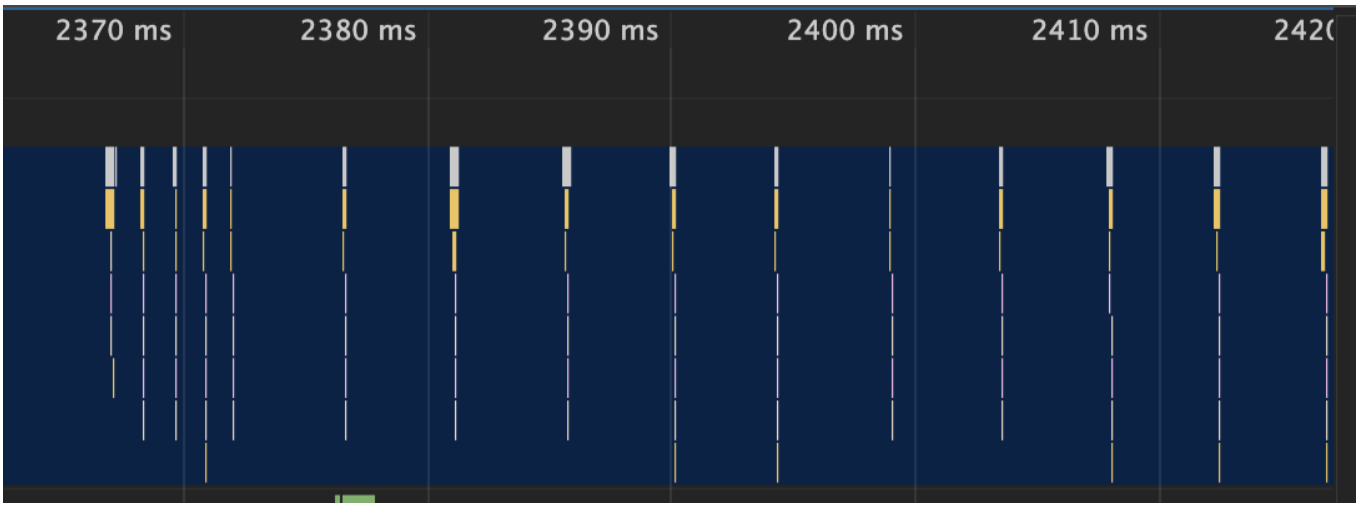
2. 如果setTimeout存在嵌套调用，那么系统会设置最短时间间隔为4毫秒

也就是说在定时器函数里面嵌套调用定时器，也会延长定时器的执行时间，可以先看下面的这段代码：

```
function cb() { setTimeout(cb, 0); }
setTimeout(cb, 0);
```

上述这段代码你有没有看出存在什么问题？

你还是可以通过Performance来记录下这段代码的执行过程，如下图所示：



循环嵌套调用setTimeout

上图中的竖线就是定时器的函数回调过程，从图中可以看出，前面五次调用的时间间隔比较小，嵌套调用超过五次以上，后面每次的调用最小时间间隔是4毫秒。之所以出现这样的情况，是因为在Chrome中，定时器被嵌套调用5次以上，系统会判断该函数方法被阻塞了，如果定时器的调用时间间隔小于4毫秒，那么浏览器会将每次调用的时间间隔设置为4毫秒。下面是Chromium实现4毫秒延迟的代码，你可以看下：

```
static const int kMaxTimerNestingLevel = 5;

// Chromium uses a minimum timer interval of 4ms. We'd like to go
// lower; however, there are poorly coded websites out there which do
// create CPU-spinning loops.  Using 4ms prevents the CPU from
// spinning too busily and provides a balance between CPU spinning and
// the smallest possible interval timer.
static constexpr base::TimeDelta kMinimumInterval = base::TimeDelta::FromMilliseconds(4);

base::TimeDelta interval_milliseconds =
      std::max(base::TimeDelta::FromMilliseconds(1), interval);

  if (interval_milliseconds < kMinimumInterval &&
      nesting_level_ >= kMaxTimerNestingLevel)
    interval_milliseconds = kMinimumInterval;

  if (single_shot)
    StartOneShot(interval_milliseconds, FROM_HERE);
  else
    StartRepeating(interval_milliseconds, FROM_HERE);
```

所以，一些实时性较高的需求就不太适合使用setTimeout了，比如你用setTimeout来实现JavaScript动画就不是一个很好的主意。

3. 未激活的页面，setTimeout执行最小间隔是1000毫秒

除了前面的4毫秒延迟，还有一个很容易被忽略的地方，那就是未被激活的页面中定时器最小值大于1000毫秒，也就是说，如果标签不是当前的激活标签，那么定时器最小的时间间隔是1000毫秒，目的是为了优化后台页面的加载损耗以及降低耗电量。这一点你在使用定时器的时候要注意。

4. 延时执行时间有最大值

除了要了解定时器的回调函数时间比实际设定值要延后之外，还有一点需要注意下，那就是Chrome、Safari、Firefox都是以32个bit来存储延时值的，32bit最大只能存放的数字是2147483647毫秒，这就意味着，如果setTimeout设置的延迟值大于 2147483647毫秒（大约24.8天）时就会溢出，那么相当于延时值被设置为0了，这导致定时器会被立即执行。你可以运行下面这段代码：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,2147483648);//会被理解调用执行
```

运行后可以看到，这段代码是立即被执行的。但如果将延时值修改为小于2147483647毫秒的某个值，那么执行时就没有问题了。

5. 使用setTimeout设置的回调函数中的this不符合直觉

如果被setTimeout推迟执行的回调函数是某个对象的方法，那么该方法中的this关键字将指向全局环境，而不是定义时所在的那个对象。这点在前面介绍this的时候也提过，你可以看下面这段代码的执行结果：

```
var name= 1;
var MyObj = {
  name: 2,
  showName: function(){
    console.log(this.name);
  }
}
setTimeout(MyObj.showName,1000)
```

这里输出的是1，因为这段代码在编译的时候，执行上下文中的this会被设置为全局window，如果是严格模式，会被设置为undefined。

那么该怎么解决这个问题呢？通常可以使用下面这两种方法。

第一种是将`MyObj.showName`放在匿名函数中执行，如下所示：

```
//箭头函数
setTimeout(() => {
    MyObj.showName()
}, 1000);
//或者function函数
setTimeout(function() {
  MyObj.showName();
}, 1000)
```

第二种是使用bind方法，将showName绑定在MyObj上面，代码如下所示：

```
setTimeout(MyObj.showName.bind(MyObj), 1000)
```

总结

好了，今天我们就介绍到这里，下面我来总结下今天的内容。

- 首先，为了支持定时器的实现，浏览器增加了延时队列。
- 其次，由于消息队列排队和一些系统级别的限制，通过setTimeout设置的回调任务并非总是可以实时地被执行，这样就不能满足一些实时性要求较高的需求了。
- 最后，在定时器中使用过程中，还存在一些陷阱，需要你多加注意。

通过分析和讲解，你会发现函数setTimeout在时效性上面有很多先天的不足，所以对于一些时间精度要求比较高的需求，应该有针对性地采取一些其他的方案。

思考时间

今天我们介绍了setTimeout，相信你现在也知道它是怎么工作的了，不过由于使用setTimeout设置的回调任务实时性并不是太好，所以很多场景并不适合使用setTimeout。比如你要使用JavaScript来实现动画效果，函数requestAnimationFrame就是个很好的选择。

那么今天留给你的作业是：你需要网上搜索了解下requestAnimationFrame的工作机制，并对比setTimeout，然后分析出requestAnimationFrame实现的动画效果比setTimeout好的原因。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。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。

在上一篇文章中我们介绍了页面中的事件和消息队列，知道了**浏览器页面是由消息队列和事件循环系统来驱动的**。

那在接下来的两篇文章中，我会通过**setTimeout**和**XMLHttpRequest**这两个WebAPI来介绍事件循环的应用。这两个WebAPI是两种不同类型的应用，比较典型，并且在JavaScript中的使用频率非常高。你可能觉得它们太简单、太基础，但有时候恰恰是基础简单的东西才最重要，了解它们是如何工作的会有助于你写出更加高效的前端代码。

本篇文章主要介绍的是**setTimeout**。其实说起setTimeout方法，从事开发的同学想必都不会陌生，它就是一个**定时器，用来指定某个函数在多少毫秒之后执行**。它会返回一个整数，表示定时器的编号，同时你还可以通过该编号来取消这个定时器。下面的示例代码就演示了定时器最基础的使用方式：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,200);
```

执行上述代码，输出的结果也很明显，通过setTimeout指定在200毫秒之后调用showName函数，并输出“极客时间”四个字。

简单了解了setTimeout的使用方法后，那接下来我们就来看看浏览器是如何实现定时器的，然后再介绍下定时器在使用过程中的一些注意事项。

浏览器怎么实现setTimeout

要了解定时器的工作原理，就得先来回顾下之前讲的事件循环系统，我们知道渲染进程中所有运行在主线程上的任务都需要先添加到消息队列，然后事件循环系统再按照顺序执行消息队列中的任务。下面我们来看看那些典型的事件：

- 当接收到HTML文档数据，渲染引擎就会将“解析DOM”事件添加到消息队列中，
- 当用户改变了Web页面的窗口大小，渲染引擎就会将“重新布局”的事件添加到消息队列中。
- 当触发了JavaScript引擎垃圾回收机制，渲染引擎会将“垃圾回收”任务添加到消息队列中。
- 同样，如果要执行一段异步JavaScript代码，也是需要将执行任务添加到消息队列中。

以上列举的只是一小部分事件，这些事件被添加到消息队列之后，事件循环系统就会按照消息队列中的顺序来执行事件。

所以说要执行一段异步任务，需要先将任务添加到消息队列中。不过通过定时器设置回调函数有点特别，它们需要在指定的时间间隔内被调用，但消息队列中的任务是按照顺序执行的，所以为了保证回调函数能在指定时间内执行，你不能将定时器的回调函数直接添加到消息队列中。

那么该怎么设计才能让定时器设置的回调事件在规定时间内被执行呢？你也可以思考下，如果让你在消息循环系统的基础之上加上定时器的功能，你会如何设计？

在Chrome中除了正常使用的消息队列之外，还有另外一个消息队列，这个队列中维护了需要延迟执行的任务列表，包括了定时器和Chromium内部一些需要延迟执行的任务。所以当通过JavaScript创建一个定时器时，渲染进程会将该定时器的回调任务添加到延迟队列中。

如果感兴趣，你可以参考Chromium中关于队列部分的源码。

源码中延迟执行队列的定义如下所示：

```
 DelayedIncomingQueue delayed_incoming_queue;
```

当通过JavaScript调用setTimeout设置回调函数的时候，渲染进程将会创建一个回调任务，包含了回调函数showName、当前发起时间、延迟执行时间，其模拟代码如下所示：

```
struct DelayTask{
  int64 id;
  CallBackFunction cbf;
  int start_time;
  int delay_time;
};
DelayTask timerTask;
timerTask.cbf = showName;
timerTask.start_time = getCurrentTime(); //获取当前时间
timerTask.delay_time = 200;//设置延迟执行时间
```

创建好回调任务之后，再将该任务添加到延迟执行队列中，代码如下所示：

```
delayed_incoming_queue.push(timerTask);
```

现在通过定时器发起的任务就被保存到延迟队列中了，那接下来我们再来看看消息循环系统是怎么触发延迟队列的。

我们可以来完善上一篇文章中消息循环的代码，在其中加入执行延迟队列的代码，如下所示：

```
void ProcessTimerTask(){
  //从delayed_incoming_queue中取出已经到期的定时器任务
  //依次执行这些任务
}

TaskQueue task_queue;
void ProcessTask();
bool keep_running = true;
void MainTherad(){
  for(;;){
    //执行消息队列中的任务
    Task task = task_queue.takeTask();
    ProcessTask(task);

    //执行延迟队列中的任务
    ProcessDelayTask()

    if(!keep_running) //如果设置了退出标志，那么直接退出线程循环
        break;
  }
}
```

从上面代码可以看出来，我们添加了一个**ProcessDelayTask函数**，该函数是专门用来处理延迟执行任务的。这里我们要重点关注它的执行时机，在上段代码中，处理完消息队列中的一个任务之后，就开始执行ProcessDelayTask函数。ProcessDelayTask函数会根据发起时间和延迟时间计算出到期的任务，然后依次执行这些到期的任务。等到期的任务执行完成之后，再继续下一个循环过程。通过这样的方式，一个完整的定时器就实现了。

设置一个定时器，JavaScript引擎会返回一个定时器的ID。那通常情况下，当一个定时器的任务还没有被执行的时候，也是可以取消的，具体方法是调用**clearTimeout函数**，并传入需要取消的定时器的ID。如下面代码所示：

```
clearTimeout(timer_id)
```

其实浏览器内部实现取消定时器的操作也是非常简单的，就是直接从delayed_incoming_queue延迟队列中，通过ID查找到对应的任务，然后再将其从队列中删除掉就可以了。

使用setTimeout的一些注意事项

现在你应该知道在浏览器内部定时器是如何工作的了。不过在使用定时器的过程中，如果你不了解定时器的一些细节，那么很有可能掉进定时器的一些陷阱里。所以接下来，我们就来讲解一下在使用定时器过程中存在的那些陷阱。

1. 如果当前任务执行时间过久，会影响定时器任务的执行

在使用setTimeout的时候，有很多因素会导致回调函数执行比设定的预期值要久，其中一个就是当前任务执行时间过久从而导致定时器设置的任务被延后执行。我们先看下面这段代码：

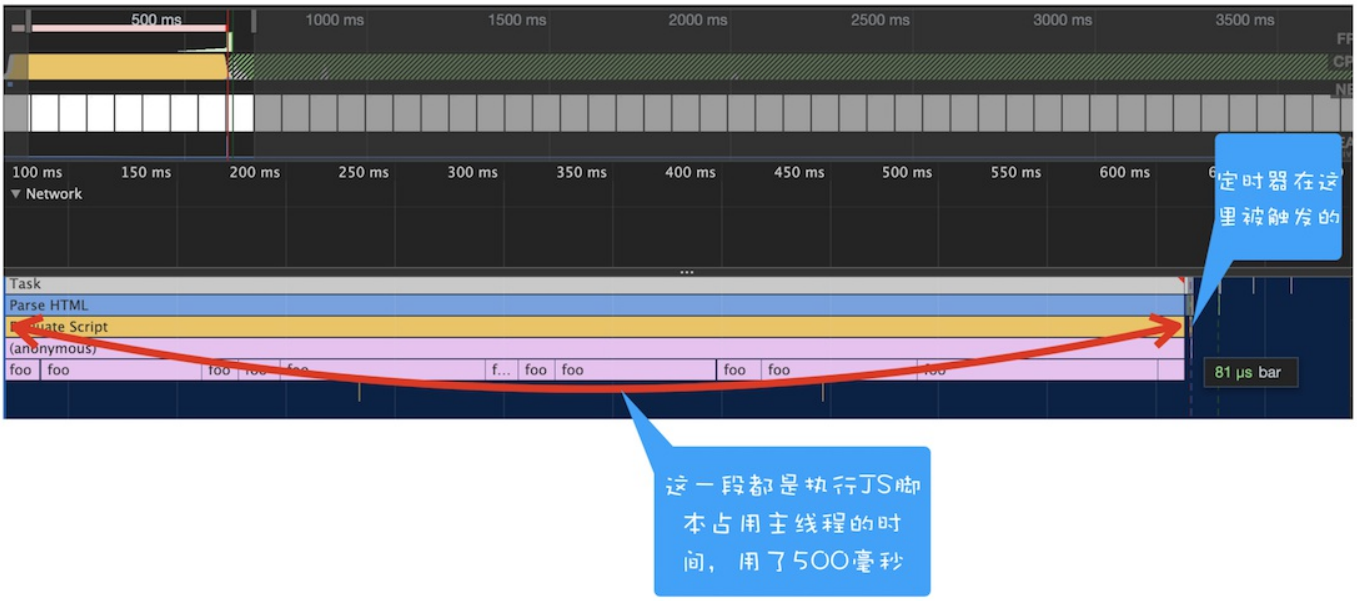
```
function bar() {
```

```
    console.log('bar')
}
function foo() {
    setTimeout(bar, 0);
    for (let i = 0; i < 5000; i++) {
        let i = 5+8+8+8
        console.log(i)
    }
}
foo()
```

这段代码中，在执行foo函数的时候使用setTimeout设置了一个0延时的回调任务，设置好回调任务后，foo函数会继续执行5000次for循环。

通过setTimeout设置的回调任务被放入了消息队列中并且等待下一次执行，这里并不是立即执行的；要执行消息队列中的下个任务，需要等待当前的任务执行完成，由于当前这段代码要执行5000次的for循环，所以当前这个任务的执行时间会比较久一点。这势必会影响到下个任务的执行时间。

你也可以打开Performance来看看其执行过程，如下图所示：



长任务导致定时器被延后执行

从图中可以看到，执行foo函数所消耗的时长是500毫秒，这也就意味着通过setTimeout设置的任务会被推迟到500毫秒以后再去执行，而设置setTimeout的回调延迟时间是0。

2. 如果setTimeout存在嵌套调用，那么系统会设置最短时间间隔为4毫秒

也就是说在定时器函数里面嵌套调用定时器，也会延长定时器的执行时间，可以先看下面的这段代码：

```
function cb() { setTimeout(cb, 0); }
setTimeout(cb, 0);
```

上述这段代码你有没有看出存在什么问题？

你还是可以通过Performance来记录下这段代码的执行过程，如下图所示：

循环嵌套调用setTimeout

上图中的竖线就是定时器的函数回调过程，从图中可以看出，前面五次调用的时间间隔比较小，嵌套调用超过五次以上，后面每次的调用最小时间间隔是4毫秒。之所以出现这样的情况，是因为在Chrome中，定时器被嵌套调用5次以上，系统会判断该函数方法被阻塞了，如果定时器的调用时间间隔小于4毫秒，那么浏览器会将每次调用的时间间隔设置为4毫秒。下面是Chromium实现4毫秒延迟的代码，你可以看下：

```
static const int kMaxTimerNestingLevel = 5;
```

```
// Chromium uses a minimum timer interval of 4ms. We'd like to go
// lower; however, there are poorly coded websites out there which do
// create CPU-spinning loops.  Using 4ms prevents the CPU from
// spinning too busily and provides a balance between CPU spinning and
// the smallest possible interval timer.
static constexpr base::TimeDelta kMinimumInterval = base::TimeDelta::FromMilliseconds(4);

base::TimeDelta interval_milliseconds =
      std::max(base::TimeDelta::FromMilliseconds(1), interval);

  if (interval_milliseconds < kMinimumInterval &&
      nesting_level_ >= kMaxTimerNestingLevel)
    interval_milliseconds = kMinimumInterval;

  if (single_shot)
    StartOneShot(interval_milliseconds, FROM_HERE);
  else
    StartRepeating(interval_milliseconds, FROM_HERE);
```

所以，一些实时性较高的需求就不太适合使用setTimeout了，比如你用setTimeout来实现JavaScript动画就不是一个很好的主意。

3. 未激活的页面，setTimeout执行最小间隔是1000毫秒

除了前面的4毫秒延迟，还有一个很容易被忽略的地方，那就是未被激活的页面中定时器最小值大于1000毫秒，也就是说，如果标签不是当前的激活标签，那么定时器最小的时间间隔是1000毫秒，目的是为了优化后台页面的加载损耗以及降低耗电量。这一点你在使用定时器的时候要注意。

4. 延时执行时间有最大值

除了要了解定时器的回调函数时间比实际设定值要延后之外，还有一点需要注意下，那就是Chrome、Safari、Firefox都是以32个bit来存储延时值的，32bit最大只能存放的数字是2147483647毫秒，这就意味着，如果setTimeout设置的延迟值大于 2147483647毫秒（大约24.8天）时就会溢出，那么相当于延时值被设置为0了，这导致定时器会被立即执行。你可以运行下面这段代码：

```
function showName(){
  console.log("极客时间")
}
var timerID = setTimeout(showName,2147483648);//会被理解调用执行
```

运行后可以看到，这段代码是立即被执行的。但如果将延时值修改为小于2147483647毫秒的某个值，那么执行时就没有问题了。

5. 使用setTimeout设置的回调函数中的this不符合直觉

如果被setTimeout推迟执行的回调函数是某个对象的方法，那么该方法中的this关键字将指向全局环境，而不是定义时所在的那个对象。这点在前面介绍this的时候也提过，你可以看下面这段代码的执行结果：

```
var name= 1;
var MyObj = {
  name: 2,
  showName: function(){
    console.log(this.name);
  }
}
setTimeout(MyObj.showName,1000)
```

这里输出的是1，因为这段代码在编译的时候，执行上下文中的this会被设置为全局window，如果是严格模式，会被设置为undefined。

那么该怎么解决这个问题呢？通常可以使用下面这两种方法。

第一种是将`MyObj.showName`放在匿名函数中执行，如下所示：

```
//箭头函数
setTimeout(() => {
    MyObj.showName()
}, 1000);
//或者function函数
setTimeout(function() {
  MyObj.showName();
}, 1000)
```

第二种是使用bind方法，将showName绑定在MyObj上面，代码如下所示：

```
setTimeout(MyObj.showName.bind(MyObj), 1000)
```

总结

好了，今天我们就介绍到这里，下面我来总结下今天的内容。

- 首先，为了支持定时器的实现，浏览器增加了延时队列。
- 其次，由于消息队列排队和一些系统级别的限制，通过setTimeout设置的回调任务并非总是可以实时地被执行，这样就不能满足一些实时性要求较高的需求了。
- 最后，在定时器中使用过程中，还存在一些陷阱，需要你多加注意。

通过分析和讲解，你会发现函数setTimeout在时效性上面有很多先天的不足，所以对于一些时间精度要求比较高的需求，应该有针对性地采取一些其他的方案。

思考时间

今天我们介绍了setTimeout，相信你现在也知道它是怎么工作的了，不过由于使用setTimeout设置的回调任务实时性并不是太好，所以很多场景并不适合使用setTimeout。比如你要使用JavaScript来实现动画效果，函数requestAnimationFrame就是个很好的选择。

那么今天留给你的作业是：你需要网上搜索了解下requestAnimationFrame的工作机制，并对比setTimeout，然后分析出requestAnimationFrame实现的动画效果比setTimeout好的原因。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。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。