第 5 章 元婴篇——JavaScript 异步编程

5.1 初识异步编程

众所周知 JavaScript 是一门单线程的语言，所以在 JavaScript 的世界中，默认情况下，同一时间节点系统只能做一件事情，这样的设定就造成了 JavaScript 这门语言的一些局限性。例如，在页面中加载一些远程数据时，如果按照单线程同步的方式运行，一旦有 HTTP 请求向服务器发送，就会出现等待数据返回之前，网页假死的效果。因为 JavaScript 在同一时间只能做一件事，这就导致了页面渲染和事件的执行在这个过程中无法进行，但是，显然在实际开发中，并没有遇见过这种页面假死的情况。

5.1.1 什么是同步和异步

基于以上的描述，在 JavaScript 的世界中，应该存在一种解决方案，来处理单线程造成的诟病，这就是同步（阻塞）和异步（非阻塞）执行模式。

1. 同步（阻塞式）

同步的意思是 JavaScript 会严格按照单线程（从上到下、从左到右的方式）的方式，进行代码的解释和运行，所以在运行代码时，不会出现先运行第 4 行和第 5 行的代码，再回头运行第 1 行和第 3 行的代码这种情况。接下来参考一个简单的同步执行案例，代码如下：

```
//第 5 章 5.1.1 一个简单的同步执行案例
var a = 1
var b = 2
var c = a + b
//这个例子总 c 一定是 3,不会出现先执行第 3 行,然后执行第 2 行和第 1 行的情况
console.log(c)
```

案例中的代码只会输出 3，不会出现其他执行结果。这是因为 JavaScript 默认执行代码的顺序是：从上到下、从左到右。这种执行顺序与人的阅读习惯类似，所以代码执行到 var c＝a＋b 时，a 和 b 分别代表 1 和 2，那么 c 的值恒定为 3。

接下来通过下列的案例升级一下代码的运行场景，若在顺序执行的代码中，加入一段循

环逻辑，该逻辑按照时间戳决定跳出条件，则最终会发生什么样的结果？代码如下：

```
//第 5 章 5.1.1 在顺序执行的代码中,加入一段循环逻辑
var a = 1
var b = 2
var d1 = new Date().getTime()
var d2 = new Date().getTime()
while(d2 - d1 < 2000){
  d2 = new Date().getTime()
}
//这段代码在输出结果之前网页会进入一个类似假死的状态
console.log(a + b)
```

按照顺序执行上面的代码，当代码在解释执行到第 5 行时，还是按正常的速度执行，但下一行，就会进入一个持续的循环中。d2 和 d1 在行级间的时间差仅仅是毫秒级的差别，所以在执行到 while 循环时，d2－d1 的值一定比 2000 小，那么这个循环会执行到什么时候？由于每次循环时，d2 都会获取一次最近的时间戳（时间戳的单位为毫秒，number 类型），直到 d2－d1＝＝2000 的情况，此时无论循环执行了多少次，恰好过了 2s 的时间，所以此代码无论计算机的硬件条件优劣情况如何，循环次数可能会不同，但循环消耗的时间一定是 2s，进而再输出 a＋b 的结果，那么这段程序的实际执行时间至少是 2s 以上。这就导致了程序阻塞的出现，也是将同步的代码运行机制叫作阻塞式运行的原因。

阻塞式运行的代码，在遇到消耗时间的代码片段时，之后的代码都必须等待耗时的代码运行完毕，才可以得到执行资源，这就是单线程同步的特点。

2．异步（非阻塞式）

在上文的阐述中，已经明白单线程同步模型中的问题所在，接下来引入单线程异步模型的介绍。

异步与同步对立，所以异步模式的代码是不会按照默认顺序执行的。JavaScript 执行引擎在工作时，仍然按照“从上到下，从左到右”的方式解释和运行代码。在解释时，如果遇到异步模式的代码，则引擎会将当前的任务“挂起”并略过（也就是先不执行这段代码）。继续向下运行非异步模式的代码。

那何时执行异步代码？直到同步代码全部执行完毕后，程序会将之前“挂起”的异步代码按照“特定的顺序”进行执行，所以异步代码并不会阻塞同步代码的运行，并且异步代码并不代表进入新的线程，与同步代码同时执行，而是等待同步代码执行完毕再进行工作。

异步代码的执行流程如图 5-1 所示。

接下来阅读下面的代码，理解异步代码的执行顺序，代码如下：

```
//第 5 章 5.1.1 理解异步代码的执行顺序
var a = 1
var b = 2
setTimeout(function(){
  console.log('输出了一些内容')
```

```
},2000)
//这段代码会直接输出 3 并且等待 2s 左右的时间再输出 function 内部的内容
console.log(a + b)
```

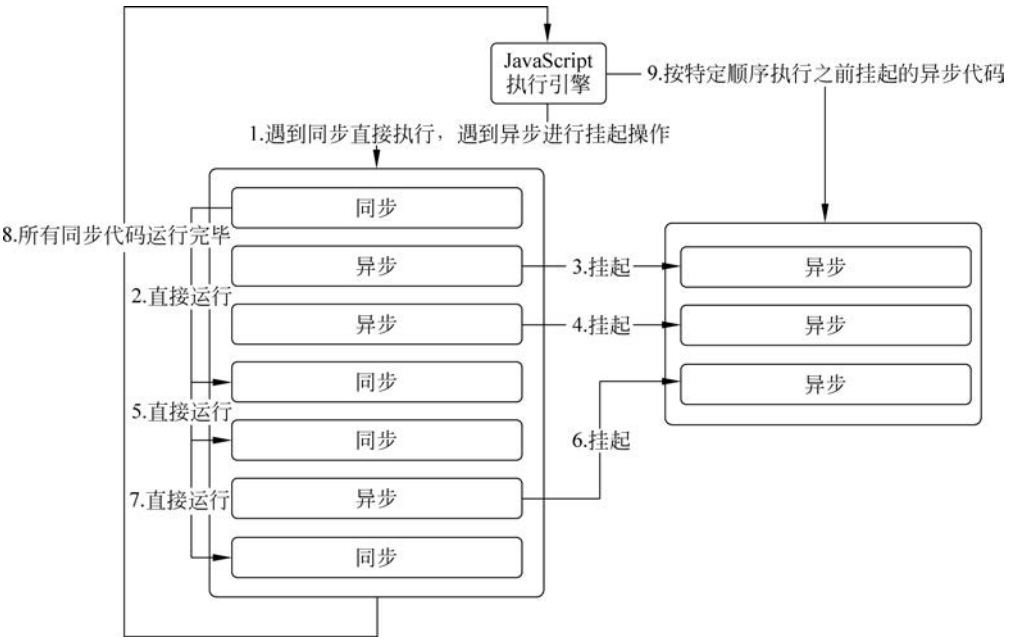

图 5-1　异步代码的执行流程

这段代码的 setTimeout()定时任务规定了 2s 后执行一些内容，在运行当前程序执行到 setTimeout()时，JavaScript 执行引擎并不会直接执行 setTimeout()内部的回调函数，而是会先将内部的函数在另外一个位置（具体是什么位置后面的内容会介绍）保存起来，然后继续执行下面的 console.log()进行输出。输出之后代码执行完毕，然后等待大概 2s，之前保存的函数会继续执行，所以无论怎么运行该代码，都会优先输出 3。

程序运行到异步（非阻塞）代码片段时，执行引擎会将异步代码的回调函数部分保存到一个暂存区，等待所有同步代码全部执行完毕后，非阻塞式的代码才会按照特定的执行顺序，分步执行，这就是单线程异步（非阻塞）程序的运行特点。

5.1.2　深入探索同步和异步

1. 结合生活理解同步和异步

艺术源于生活，程序的运行流程也源于生活，同步和异步的场景在生活中有很多实际的映射，参考下面的生活案例理解同步和异步。

1）同步的案例

在超市买完东西进行结账时，想要在同一时间节点结账的顾客都要在收银台排队，排队的顺序按照顾客到达收银台的顺序。顾客结账的流程就是一个非常完美的同步执行流程，

假设超市只有一个收银台，收银工作人员就相当于 JavaScript 执行引擎，每个结账的顾客就是 JavaScript 的一段代码：一个函数或一个循环。在同一时间点，收银员只能处理一个顾客的结账动作，这个流程也与 JavaScript 在同一时间节点只能做一件事是相同的。当某个顾客在结账时，若会员卡没有及时找到，或者其购买的蔬菜没有称重，则该顾客需要消耗时间来完成这些任务，以便结账流程可以顺利完成。在该顾客未完成结账拿走小票前，后面的所有顾客都需要等待，这就是单线程阻塞模型，也就是同步在生活中的映射。

2）异步的案例

当人们进餐馆吃饭时，这个场景就属于一个完美的异步流程场景。每一桌来的客人会按照他们来的顺序进行点菜，假设只有一个服务员的情况，点菜必须按照先后顺序，但是服务员不需要等第一桌客人点好的菜出锅，就可以直接去收集第二桌及第三桌客人的需求。这样可能在十分钟之内，服务员就将所有桌的客人点菜的菜单统计出来，并且发送给后厨。之后的上菜顺序，也不会按照点餐顾客的下单顺序，因为后厨收集到菜单后，可能有 1、2 和 3 号桌的客人都点了锅包肉，那么厨师会先一次出三份锅包肉，这样锅包肉在上菜时 1、2 和 3 号桌的客人都可以得到，并且其他的菜也会乱序地逐一上菜，这个过程就是异步的。如果按照同步的模式点餐，则默认在饭店点菜就会出现饭店在第一桌客人上满菜前，第二桌及其之后的客人就只能等待，连菜都不能点，两种上菜的流程如图 5-2 所示。

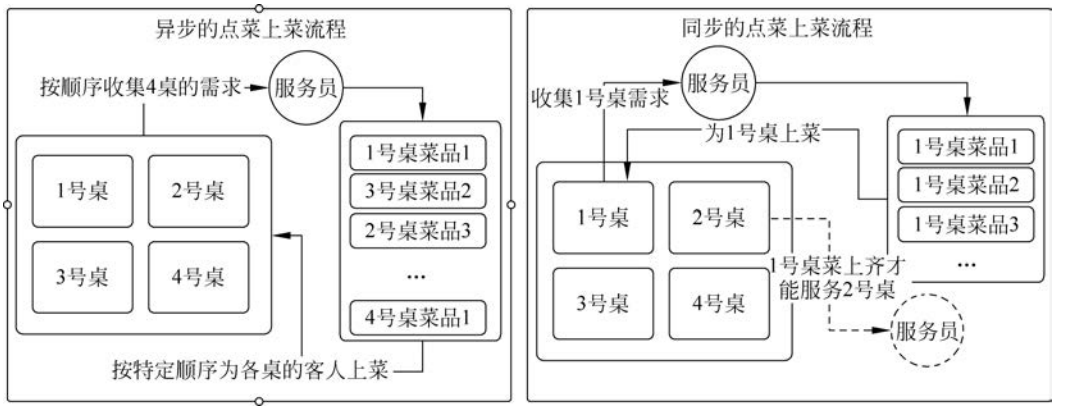

图 5-2　两种上菜的流程

根据图 5-2 所示，若采用同步的流程实现点菜上菜流程，这家餐馆的生意就会变得非常惨淡，因为在第一桌客人没有上齐菜前，第二桌客人连点菜都进行不了，所以使用异步流程进行点菜上菜可以最大化地利用餐馆资源，实现好的用户体验，这就是异步流程存在的意义。

2. 实际的例子

JavaScript 的运行顺序完全按照单线程的异步模型执行，即同步在前，异步在后。所有的异步任务都要等待当前的同步任务执行完毕后才能执行。参看下面的案例，代码如下：

```
//第 5 章 5.1.2 所有的异步任务都要等待当前的同步任务执行完毕后才能执行
var a = 1
var b = 2
var d1 = new Date().getTime()
var d2 = new Date().getTime()
setTimeout(function(){
  console.log('我是一个异步任务')
},1000)
while(d2 - d1 < 2000){
  d2 = new Date().getTime()
}
//这段代码在输出 3 之前会进入假死状态,'我是一个异步任务'一定会在 3 之后输出
console.log(a + b)
```

实际运行案例后,便会感受到单线程异步模型的执行顺序,运行案例会发现 setTimeout()设置的时间是 1000ms,但是在 while 阻塞 2000ms 的循环后,并没有等待 1s 而是直接输出“我是一个异步任务”。这是因为 setTimeout()的时间计算是从 setTimeout()这个函数执行时开始计算的。JavaScript 同一时间节点只能做一件事,所以在进入 while 循环时,JavaScript 执行引擎便无法运行其他代码。while 循环在消耗 2000ms 的过程中 setTimeout()的定时任务已经到时间了,但此时 JavaScript 执行引擎正在执行循环,所以定时器无法得到执行资源。待所有同步任务执行完毕后,setTimeout()的回调函数才会被触发。

可以结合生活场景分析该案例:张三某一天在公司上班,公司安排张三在上午完成一系列工作任务,张三在将工作任务执行到上午 10 点时,公司的小王跟张三说上午 11 点 30 分在会议室开会,此时张三手头有工作任务,便记录一下 11 点 30 分要开会的计划。当工作到 11 点 30 分时,张三手里的主要工作还没有做完,此时张三无法按时去会议室开会,只能继续完成手中的工作任务,直到上午的工作任务执行完毕,张三发现已经十二点了。虽然过了 11 点 30 分的开会时间,但张三还是按照约定去了会议室,将计划执行完毕。这个过程与上面案例的流程十分相似。

5.1.3 异步与多线程的区别

1. 通过 setInterval()理解异步与多线程的区别

很多人在学习异步时,会误以为异步与多线程是一回事,最直观的误会是由 setInterval()引起的,参考一个 setInterval()的案例,代码如下:

```
<!-- 第 5 章 5.1.3 一个 setInterval()的案例 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        setInterval(() => {
            console.log('第 1 个定时器')
        },1000)
        setInterval(() => {
            console.log('第 2 个定时器')
        },1000)
    </script>
</body>
</html>
```

运行该案例时，控制台上会每隔 1s 输出两行数据，如图 5-3 所示。

```
第1个定时器        第5章 5.1.3 一个setInterval()的案例.html:13
第2个定时器        第5章 5.1.3 一个setInterval()的案例.html:16
第1个定时器        第5章 5.1.3 一个setInterval()的案例.html:13
第2个定时器        第5章 5.1.3 一个setInterval()的案例.html:16
第1个定时器        第5章 5.1.3 一个setInterval()的案例.html:13
第2个定时器        第5章 5.1.3 一个setInterval()的案例.html:16
第1个定时器        第5章 5.1.3 一个setInterval()的案例.html:13
第2个定时器        第5章 5.1.3 一个setInterval()的案例.html:16
第1个定时器        第5章 5.1.3 一个setInterval()的案例.html:13
第2个定时器        第5章 5.1.3 一个setInterval()的案例.html:16
```

图 5-3　两种上菜的流程

该案例运行时，会让人感觉 JavaScript 在同一时间节点做两件事。这看似并行的代码实际上还是串行动作，在定时器中追加一个 for 循环便可以验证，代码如下：

```
<!-- 第 5 章 5.1.3 在定时器中追加一个 for 循环 -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        setInterval(() => {
            for( let i = 0 ;i < 10 ;i++){
                console.log(`第 1 个定时器 ${i}`)
            }
        },1000)
```

```
    setInterval(() => {
      for( let i = 0 ;i < 10 ;i++){
        console.log(`第 2 个定时器 ${i}`)
      }
    },1000)
  </script>
</body>
</html>
```

运行追加了 for 循环的案例便可直观地认识到异步与并行的区别。若程序为并行，两个定时器在 1s 时同时触发，则两个 for 循环的 10 次输出应该也是并列的。实际运行案例后会发现，每次到达定时器执行时机时，都会等待第 1 个定时器执行完毕后，才会执行第 2 个，如图 5-4 所示。

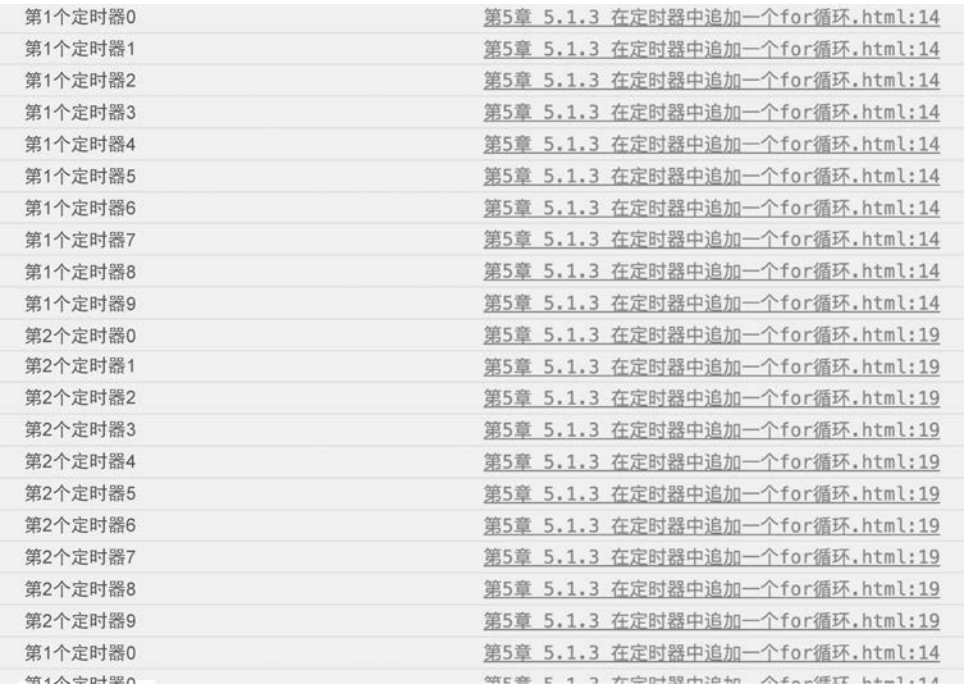

图 5-4　等待第 1 个定时器执行完毕后，才会执行第 2 个

2. 以 setTimeout()探究异步任务的执行规则

JavaScript 默认的执行顺序为同步先行，异步在后。异步任务间也存在执行顺序的规则，其具体规则如下。

（1）同一时间节点到期的异步任务，按照创建的顺序执行，代码如下：

```
<!-- 第 5 章 5.1.3 同一时间节点到期的异步任务,按照创建的顺序执行 -->
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <script>
    setTimeout(() => {
      console.log('第1个')
    }, 1000);
    setTimeout(() => {
      console.log('第2个')
    }, 1000);

    setTimeout(() => {
      console.log('第3个')
    }, 1000);
    console.log('同步先行')
  </script>
</body>
</html>
```

（2）不同时间节点到期的异步任务，按照时间顺序执行，代码如下：

```
<!-- 第5章 5.1.3 不同时间节点到期的异步任务,按照时间顺序执行 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <script>
    setTimeout(() => {
      console.log('第3个')
    }, 1000);
    setTimeout(() => {
      console.log('第2个')
    }, 500);

    setTimeout(() => {
      console.log('第1个')
    }, 300);
    console.log('同步先行')
  </script>
</body>
</html>
```

（3）没有设置时间的异步任务，也会等待同步任务执行完毕后执行，代码如下：

```
<!-- 第 5 章 5.1.3 没有设置时间的异步任务,也会等待同步任务执行完毕后执行 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <script>
    setTimeout(() => {
      console.log('第 1 个')
    });
    setTimeout(() => {
      console.log('第 2 个')
    });
    setTimeout(() => {
      console.log('第 3 个')
    });
    console.log('同步先行')
  </script>
</body>
</html>
```

（4）每个异步任务的回调函数内部，仍然会区分作用域内部的同步异步关系，代码如下：

```
<!-- 第 5 章 5.1.3 每个异步任务的回调函数内部,仍然会区分作用域内部的同步异步关系 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <script>
    setTimeout(() => {
      console.log(5)
      setTimeout(() => {
        console.log(7)
      });
      console.log(6)
    },10);
```

```
        setTimeout(() => {
          console.log(2)
          setTimeout(() => {
            console.log(4)
          });
          console.log(3)
        });

        console.log(1)
      </script>
    </body>
    </html>
```

5.2　初识异步编程

5.2.1　浏览器的线程组成

1. 什么是线程和进程

进程(Process) 是计算机中的程序关于某数据集合上的一次运行活动,是系统进行资源分配和调度的基本单位,是操作系统结构的基础。在当代面向线程设计的计算机结构中,进程是线程的容器。程序是指令、数据及其组织形式的描述,进程是程序的实体。是计算机中的程序关于某数据集合上的一次运行活动,是系统进行资源分配和调度的基本单位,是操作系统结构的基础。程序是指令、数据及其组织形式的描述,进程是程序的实体。

线程(thread) 是操作系统能够进行运算调度的最小单位。它被包含在进程之中,是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流,一个进程中可以并发多个线程,每条线程并行执行不同的任务。

线程是进程的一个执行流,是 CPU 调度和分派的基本单位,它是比进程更小的能独立运行的基本单位。

一个进程由几个线程组成(拥有很多相对独立的执行流的用户程序共享应用程序的大部分数据结构),线程与同属一个进程的其他的线程共享进程所拥有的全部资源。

进程有独立的地址空间,一个进程崩溃后,在保护模式下不会对其他进程产生影响,而线程只是一个进程中的不同执行路径。

线程有自己的堆栈和局部变量,但线程没有单独的地址空间,一个线程死掉就等于整个进程死掉,所以多进程的程序要比多线程的程序健壮,但在进行进程切换时,耗费资源较大,效率要差一些,但对于一些要求同时进行并且又要共享某些变量的并发操作,只能用线程,而不能用进程。

2. JavaScript 的运行环境是单线程吗?

5.1 节通过几个简单的例子了解了 JavaScript 代码的运行顺序,细心的读者会发现,若真的只存在一条线程,JavaScript 编程语言则无法实现异步能力。

回顾 5.1.2 节的代码案例，代码如下：

```
//第 5 章 5.1.2 所有的异步任务都要等待当前的同步任务执行完毕后才能执行
var a = 1
var b = 2
var d1 = new Date().getTime()
var d2 = new Date().getTime()
setTimeout(function(){
  console.log('我是一个异步任务')
},1000)
while(d2 - d1 < 2000){
  d2 = new Date().getTime()
}
//这段代码在输出 3 之前会进入假死状态,'我是一个异步任务'一定会在 3 之后输出
console.log(a + b)
```

该案例在执行 while 循环时，便超过了 setTimeout()的执行时间。若 JavaScript 真的在同一时间只能做一件事，则程序在运行到循环时，JavaScript 执行引擎并不会有任何资源供定时器计时使用，其结果应为 2s 后输出 3，再经过 1s 才输出“我是一个异步任务”。因为，若真的只有一条线程工作，则在代码没运行完前，计时器便无法工作，但实际情况却恰恰相反，执行到 setTimeout()时计时器便开始工作，while 循环在执行过程中，计时器便到达执行时间，所以实际参与 JavaScript 代码运行的线程不只一条。

虽然浏览器是以单线程执行 JavaScript 代码的，但是浏览器实际是以多个线程协助操作实现单线程异步模型的，具体线程组成如下：

(1) GUI 渲染线程。

(2) JavaScript 引擎线程。

(3) 事件触发线程。

(4) 定时器触发线程。

(5) HTTP 请求线程。

(6) 其他线程。

5.2.2 线程间的工作关系

按照真实的浏览器线程组成分析，会发现实际上运行 JavaScript 的线程并不只有一个，但是为什么说 JavaScript 是一门单线程的语言呢？因为在这些线程中实际参与代码执行的线程并不是所有线程，例如 GUI 渲染线程之所以单独存在，是为了防止在 HTML 网页渲染一半时，突然执行一段阻塞式的 JavaScript 代码，而导致网页卡在一半这种效果。在 JavaScript 代码运行的过程中，实际执行程序时，同时只存在一条活动线程，如图 5-5 所示。

这里实现同步和异步就是靠多线程切换的形式进行实现的。

以定时器为例，在 JavaScript 代码执行时，实际参与代码执行的至少有 JavaScript 引擎

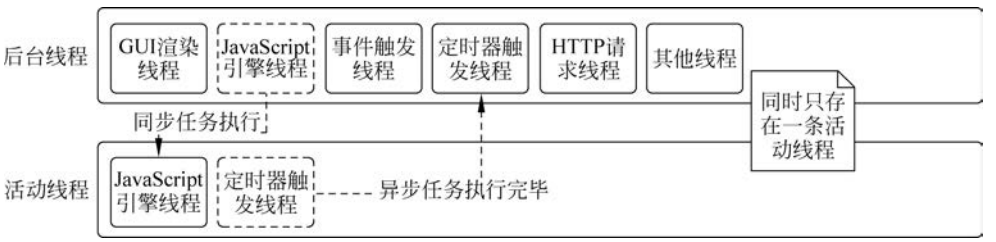

图 5-5　同时只存在一条活动线程

线程与定时器触发线程，可以通过画图的形式，了解实现定时器的线程模型，如图 5-6 所示。

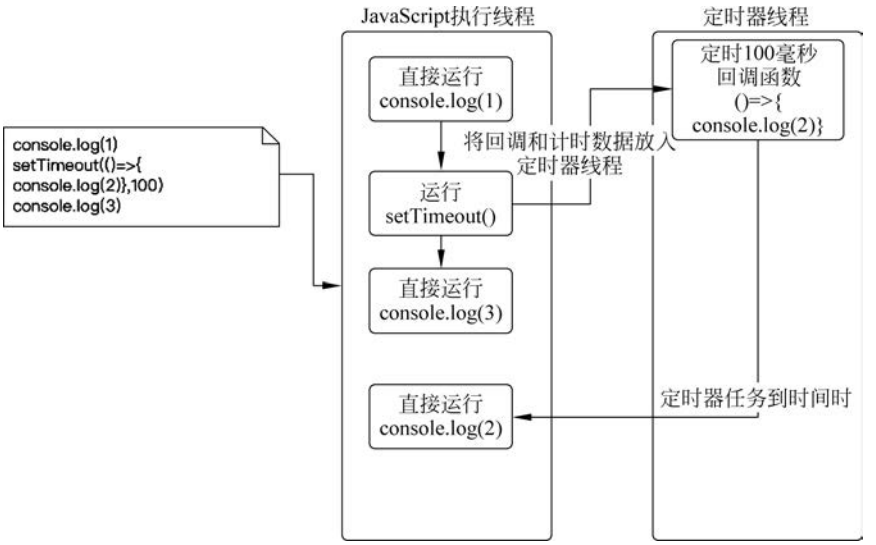

图 5-6　实现定时器的线程模型

根据图 5-5 的描述可以理解，JavaScript 在实际运行时，可能会有多个线程参与程序的运行，绝大多数场景的执行流程可理解为如下顺序：

（1）JavaScript 执行引擎以从上到下、从左至右的顺序执行代码。

（2）当遇到同步任务时，JavaScript 执行引擎直接运行当前代码。

（3）遇到类似 setTimeout()或 setInterval()的异步任务时，优先执行外层函数。这里需要了解的知识是，setTimeout()或 setInterval()函数的最外层函数本身属于同步代码，所以程序执行到该函数位置时，setTimeout()或 setInterval()函数体本身已经触发，其内部的回调函数才是异步任务的部分。setTimeout()或 setInterval()的功能是在定时器线程中创建一个计时的异步任务。

（4）异步任务创建后，进入异步任务对应的线程等待回调触发。例如，定时器任务会被发送到定时器线程中，定时器线程会按照定时任务设定的时间进行计时。定时器线程在计时过程中，JavaScript 执行引擎还会继续执行后续的同步任务，直到代码执行完毕。

（5）所有同步任务执行完毕后，异步线程中的任务才会陆续触发回调，直到所有异步任

务执行完毕。

鉴于参与程序运行的线程过多，通常将上面的细分线程归纳为下列两条线程。

1）主线程

这个线程用来执行页面的渲染、JavaScript 代码的运行和事件的触发等任务。

2）工作线程

这个线程是在幕后工作的，用来处理异步任务的执行，以实现非阻塞的运行模式。

5.2.3 JavaScript 的运行模型

在学习变量和数据类型时，就已经了解 JavaScript 存在各种内存空间，用于存储数据，所以实际上 JavaScript 有着非常复杂的运行模型。

可以以逻辑分区的方式对 JavaScript 的运行模型进行简化，如图 5-7 所示。

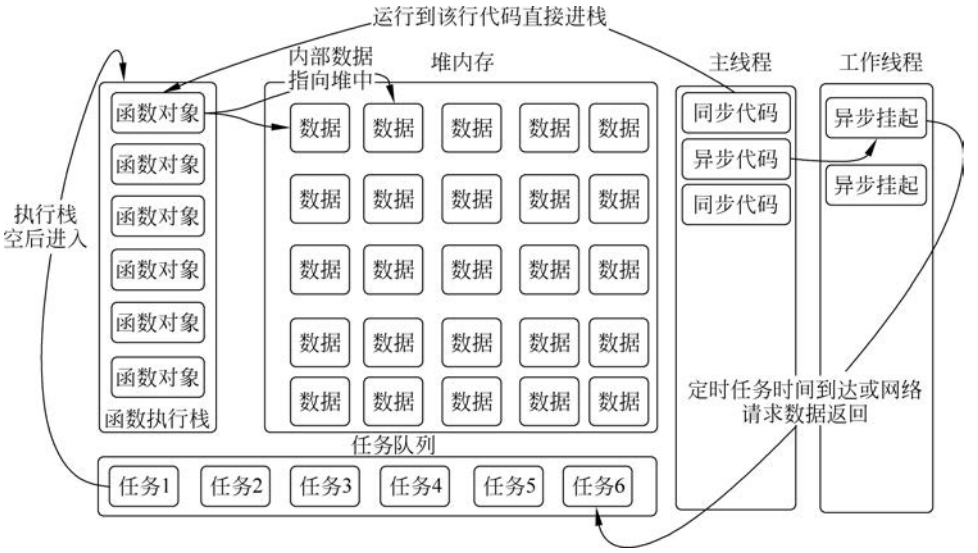

图 5-7 以逻辑分区的方式对 JavaScript 的运行模型进行简化

图 5-7 是 JavaScript 运行时的一个工作流程和内存划分的简要描述。根据图中内容可以得知，主线程就是 JavaScript 执行代码的线程，主线程代码在运行时，会按照同步和异步代码将其分成两个去处。

当遇到同步代码时，会直接将该任务放在一个叫作“函数执行栈”的空间进行执行。执行栈是典型的栈结构（先进后出），程序在运行时，会将同步代码按顺序入栈，将异步代码放到工作线程中暂时挂起。工作线程中保存任务的有定时任务函数、JavaScript 的交互事件及 JavaScript 的网络请求等耗时操作。

当主线程将代码块筛选完毕后，进入执行栈的函数会按照从外到内的顺序依次运行，运行中涉及的对象数据会在堆内存中进行保存和管理。当执行栈内的任务全部执行完毕后，执行栈就会被清空。执行栈被清空后，“事件循环”就会工作，“事件循环”会检测任务队列中

是否有要执行的任务，这个任务队列的任务源自工作线程。程序运行期间，工作线程会把到期的定时任务、返回数据的 HTTP 任务等异步任务，按照先后顺序插入任务队列中。等执行栈被清空后，事件循环会访问任务队列，将任务队列中存在的任务，按顺序（先进先出）放在执行栈中继续执行，直到任务队列被清空。

5.3　EventLoop 与异步任务队列

5.3.1　异步任务的去向与 EventLoop 的工作原理

对 5.2 节的学习，可能在大脑中很难形成图形界面，以此来帮助分析 JavaScript 的实际运行思路，接下来以一段简单的同步和异步混合任务案例为参考，开启更加细致的学习，代码如下：

```
//第 5 章 5.3.1 一段简单的同步和异步混合任务案例
function task1(){
    console.log('第 1 个任务')
}
function task2(){
    console.log('第 2 个任务')
}
function task3(){
    console.log('第 3 个任务')
}
function task4(){
    console.log('第 4 个任务')
}
task1()
setTimeout(task2,1000)
setTimeout(task3,500)
task4()
```

可以将案例中的 4 个函数看作 4 个要执行的任务，名为 task1、task2、task3 和 task4。task1 与 task4 任务按照同步任务执行，task2 与 task3 任务按照不同的时间节点以异步方式执行。

接下来结合图形分步拆解代码案例的运行过程，深入剖析代码的执行过程。该案例在运行前，相关的执行结构如图 5-8 所示。

在案例代码刚开始运行时，主线程即将工作，代码会按照顺序从上到下进行解释执行，此时执行栈、工作线程及任务队列都是空的，事件循环也没有工作。接下来让代码向下执行一步，如图 5-9 所示。

结合图 5-9 可以看出程序在主线程执行后，将任务 1、任务 4 和任务 2、任务 3 分别放进了两个方向，任务 1 和任务 4 都是立即执行任务，所以会按照 1 到 4 的顺序进栈出栈（这里

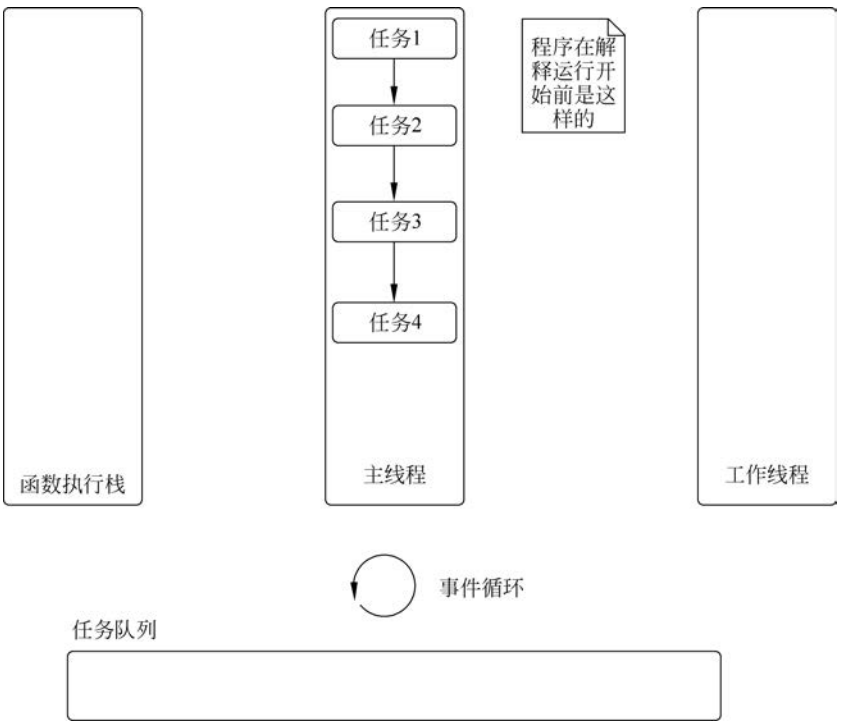

图 5-8　该案例在运行前，相关的执行结构

由于任务 1 和任务 4 是同级任务，所以会先执行任务 1 的进出栈，再执行任务 4 的进出栈），而任务 2 和任务 3 由于是异步任务，所以会进入工作线程挂起，并开始计时（这个过程并不会影响主线程的运行），完成以上步骤后，任务队列还是空的。该步骤运行后，各容器的内存结构如图 5-10 所示。

运行到此会发现，同步任务的执行速度是飞快的，瞬间执行栈已经清空，而任务 2 和任务 3 还没有到时间。这时事件循环便开始工作，来等待任务队列中的任务进入，此时工作线程的定时器时钟会计算任务 2 和任务 3 的到期时间，如图 5-11 所示。

参考图 5-11 的执行过程，会发现并不会直接将任务 2 和任务 3 一起放进任务队列，而是哪个计时器到时间，再将哪个任务放进任务队列。这样事件循环就会发现队列中的任务，并且将任务放入执行栈中进行消费，此时会输出任务 3 的内容。

最后到时间的任务 2，也会按照相同的方式，先进入任务队列，再进入执行栈，直到任务队列的任务清空，程序到此执行完毕，如图 5-12 所示。

通过图解之后，脑海里就会更清晰地记住异步任务的执行方式。这里采用最简单的任务模型进行描绘，复杂的任务在内存中的分配和走向是非常复杂的，有了这次的经验后便可以通过观察代码优先在大脑中模拟运行，这样可以更清晰地理解 JavaScript 的运行机制。

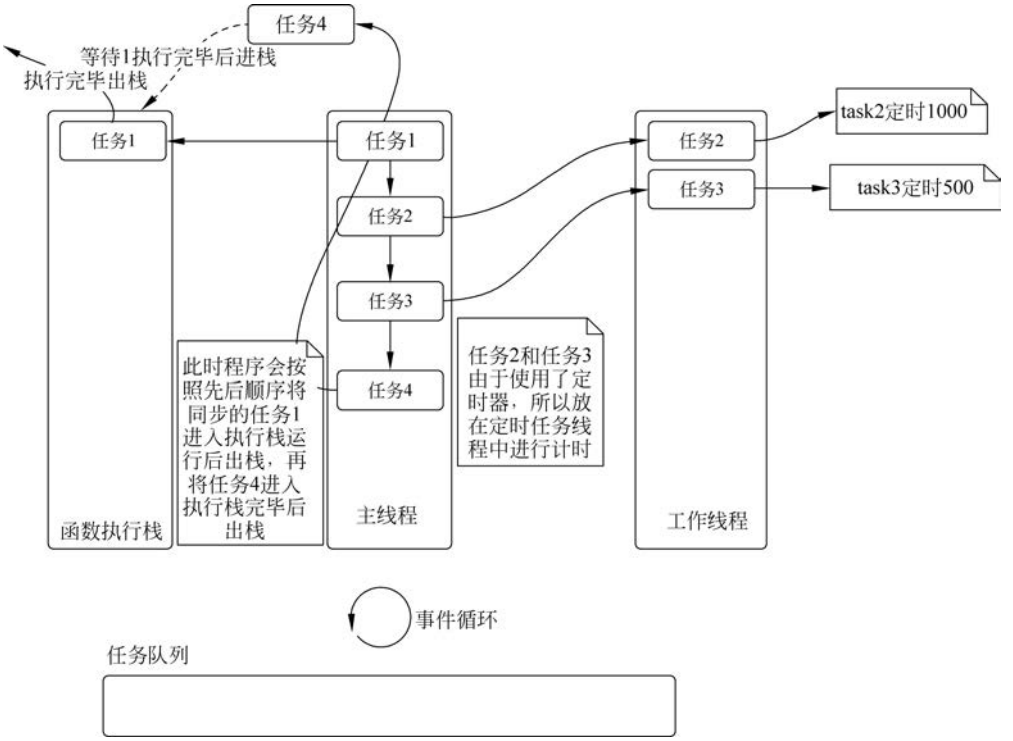

图 5-9　让代码向下执行一步

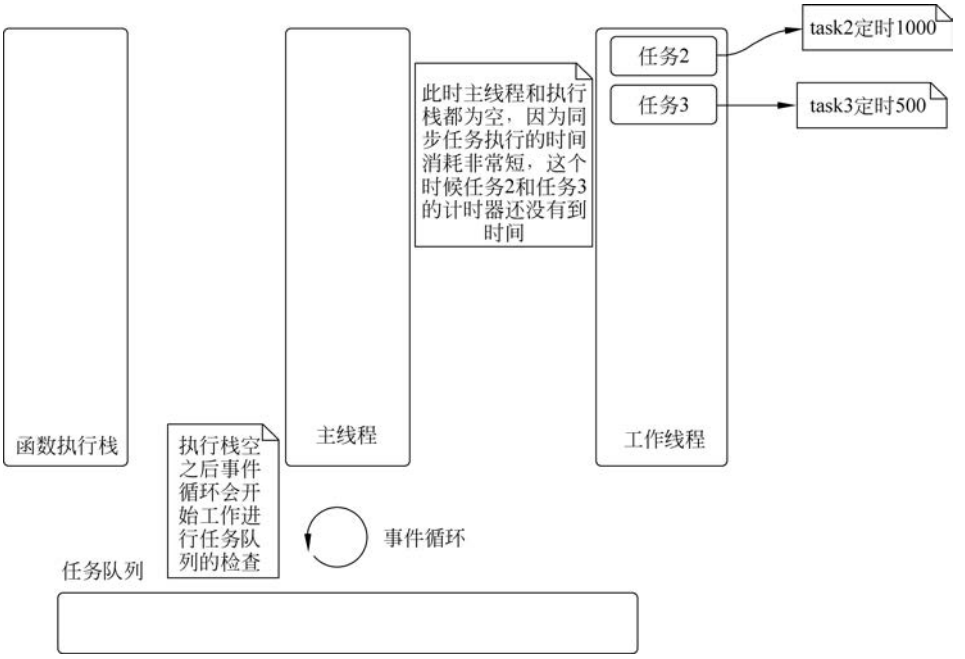

图 5-10　该步骤运行后，各容器的内存结构

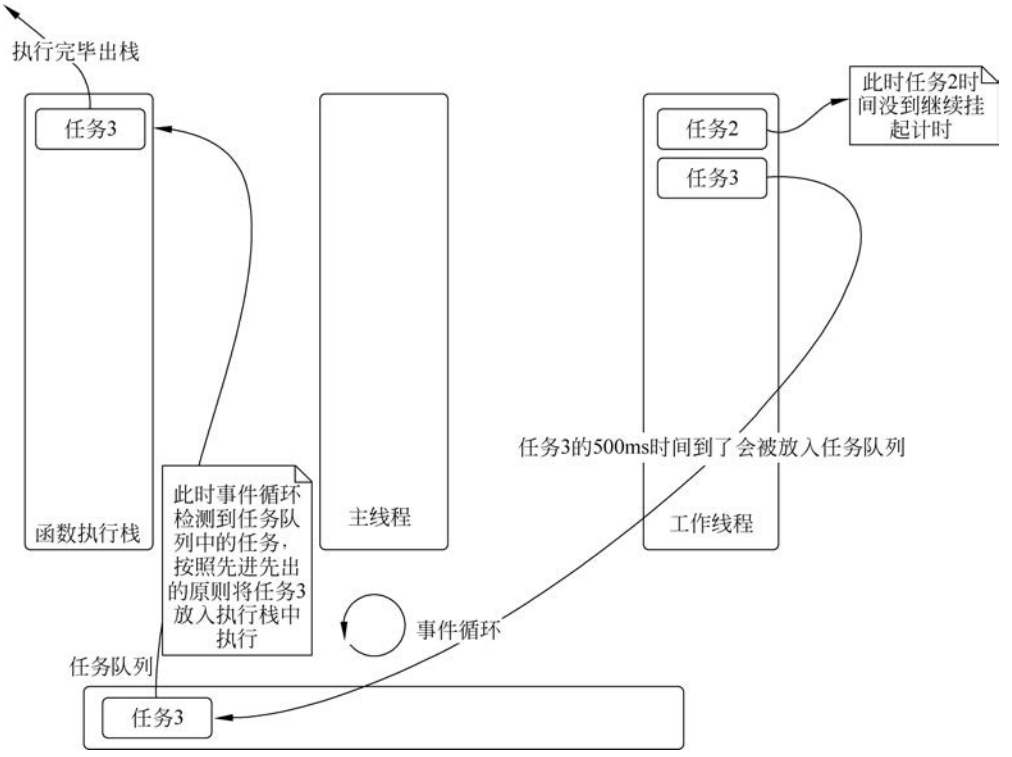

图 5-11　工作线程的定时器时钟会计算任务 2 和任务 3 的到期时间

5.3.2　关于函数执行栈

执行栈是一个栈的数据结构，当运行单层函数时，函数任务会进栈执行后出栈销毁，然后下一个函数任务才会进栈执行再出栈销毁，当有函数嵌套调用时，栈中才会堆积栈帧，接下来查看函数嵌套的例子，代码如下：

```
//第 5 章 5.3.2 函数嵌套的例子
function task1(){
  console.log('task1 执行')
  task2()
  console.log('task2 执行完毕')
}
function task2(){
  console.log('task2 执行')
  task3()
  console.log('task3 执行完毕')
}
function task3(){
  console.log('task3 执行')
}
task1()
console.log('task1 执行完毕')
```

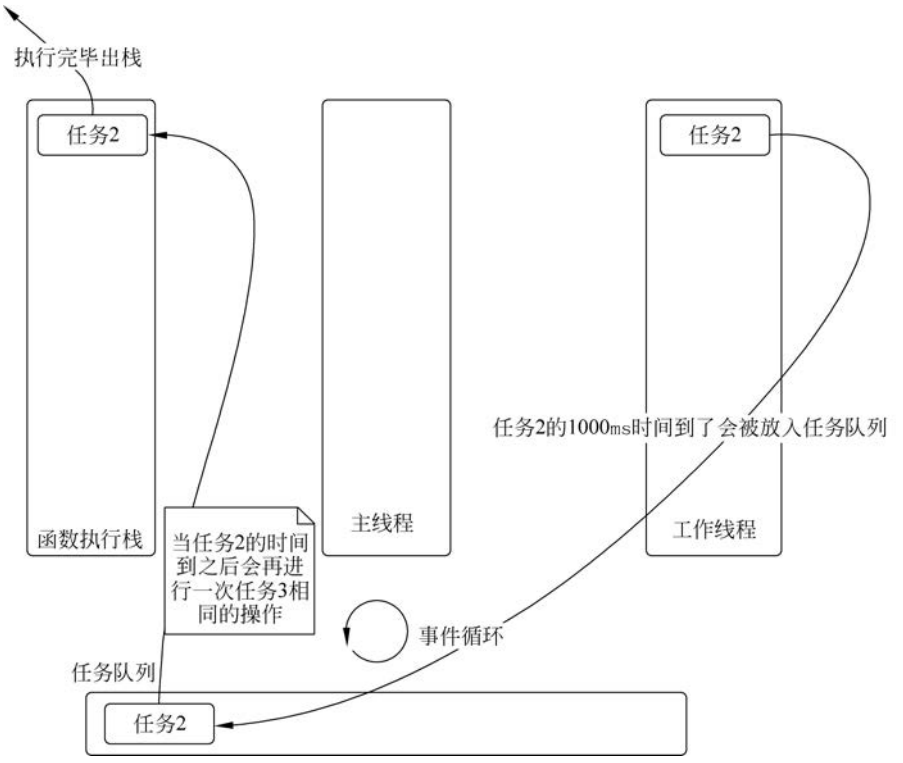

图 5-12　程序到此执行完毕

仅通过字面阅读，便能很快分析出该案例的输出结果，代码如下：

```
//第 5 章 5.3.2 该案例的输出的结果
/*
task1 执行
task2 执行
task3 执行
task3 执行完毕
task2 执行完毕
task1 执行完毕
*/
```

接下来仍然将案例中的函数看作任务，名为 task1、task2 及 task3，文字说明仍然按照代码流程进行描述。以图形分步描绘该案例的运行流程，案例的第 1 步的执行结果如图 5-13 所示。

第 1 次执行时，task1()函数执行到第 1 个 console.log()时会先进行输出，接下来会遇到 task2()函数的调用，task1()在未结束的情况下，主线程进入了 task2()函数，如图 5-14 所示。

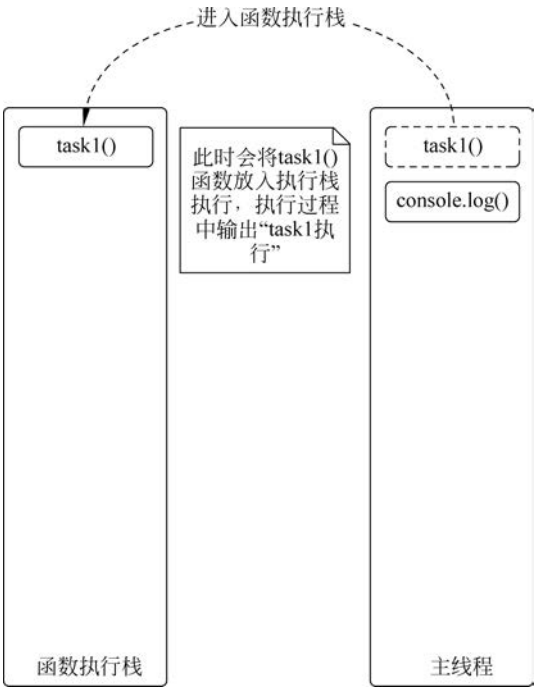

图 5-13　案例的第 1 步的执行结果

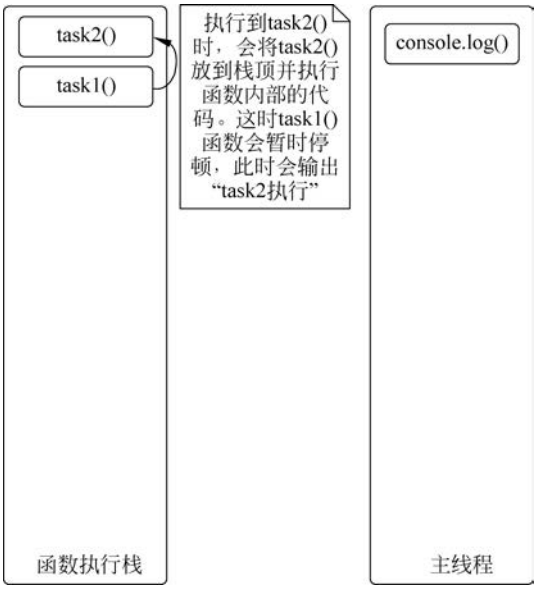

图 5-14　task1()在未结束的情况下，主线程进入了 task2()函数

执行到此时检测到 task2()中还有调用 task3()的函数，这样便会继续进入 task3()中执行，如图 5-15 所示。

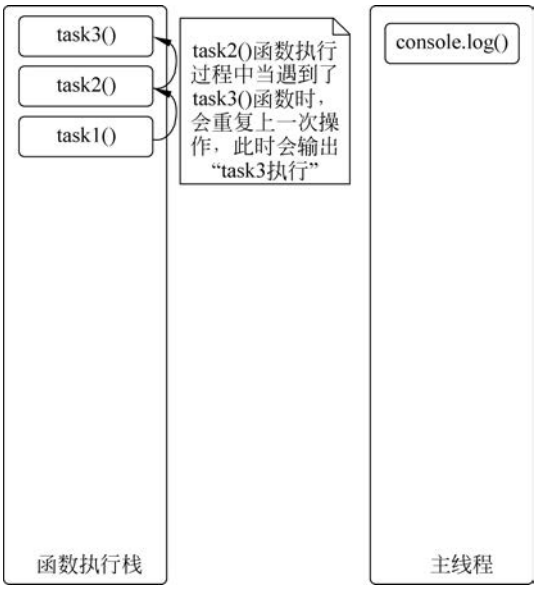

图 5-15　继续进入 task3()中执行

在执行完 task3()内的输出函数后，如果 task3()内部没有其他代码，则 task3()函数算执行完毕。接下来就会进行出栈工作，如图 5-16 所示。

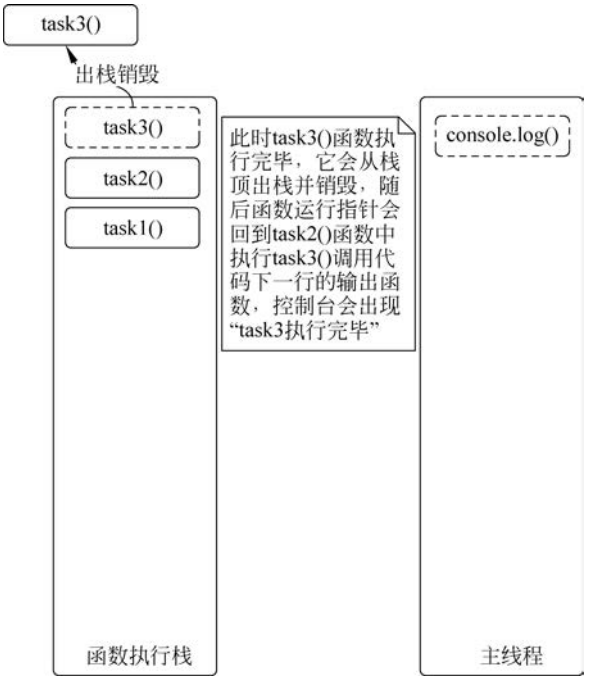

图 5-16　继续进入 task3()中执行

此时会发现 task3()出栈后，程序运行又会回到 task2()的函数中继续执行。接下来会发生与此步骤相同的事，如图 5-17 所示。

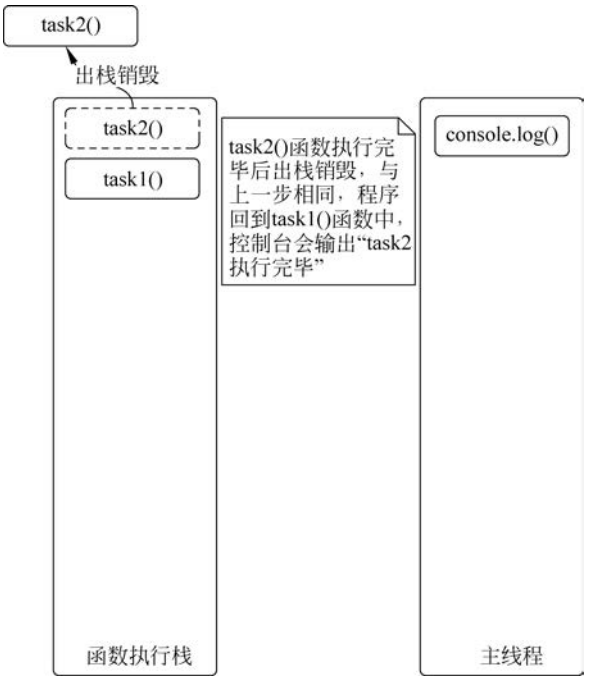

图 5-17　接下来会发生与此步骤相同的事

再之后剩下 task1()函数，随后会继续执行相同操作，直到函数执行栈清空，如图 5-18 所示。

当 task1()执行完毕后，最后一行输出，会进入执行栈执行并销毁，销毁后执行栈和主线程清空。整个过程就会体现出 1、2、3、3、2、1 这个顺序，打印输出时，也能通过打印的顺序来理解入栈和出栈的顺序和流程。

5.3.3　递归和栈溢出

理解了执行栈执行逻辑后，接下来深入学习递归函数。递归函数是项目开发时经常涉及的函数，在遍历未知深度的树形结构或其他合适的场景中需要大量使用递归。递归在面试中会经常被问到递归的风险问题，若了解了执行栈的执行逻辑后，则递归函数便可以看成一个嵌套了 $N(N>=2)$层的函数。这种函数在执行过程中，会产生大量的栈帧堆积。如果处理的数据过大，函数调用的层数过深，则会导致执行栈的高度不够放置新的栈帧，从而造成栈溢出的错误。这是在做海量数据递归时，一定要注意这个问题。

1. 关于执行栈的深度

关于执行栈的深度，不同的浏览器间存在差异，本节以 Chrome 浏览器为例，来尝试一下递归造成的栈溢出，代码如下：

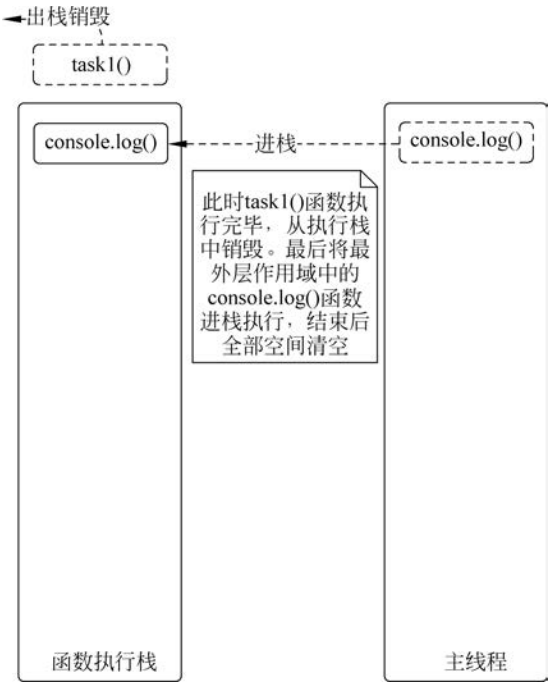

图 5-18　随后会继续执行相同操作，直到函数执行栈清空

```
//第 5 章 5.3.3 递归造成的栈溢出
var i = 0;
function task(){
  let index = i++
  console.log(`递归了 ${index}次`)
  task()
  console.log(`第 ${index}次递归结束`)
}

task()
```

栈溢出案例的运行结果如图 5-19 所示。

```
递归了11377次
递归了11378次
▸Uncaught RangeError: Maximum call stack size exceeded
    at task (test111.html:11)
    at task (test111.html:13)
    at task (test111.html:13)
    at task (test111.html:13)
    at task (test111.html:13)
    at task (test111.html:13)
    at task (test111.html:13)
    at task (test111.html:13)
    at task (test111.html:13)
    at task (test111.html:13)
    at task (test111.html:13)
```

图 5-19　栈溢出案例的运行结果

运行后发现，在递归执行 11 378 次后，会提示超过栈深度的错误，可以简单地将此数据看作 Chrome 浏览器中函数执行栈的最大深度。

2. 如何跨越执行栈的限制

发现问题后，考虑如何能通过技术手段跨越递归的限制。接下来将代码做如下更改，便不会出现栈溢出错误，代码如下：

```
//第 5 章 5.3.3 将代码做如下更改，便不会出现栈溢出错误
var i = 0;
function task(){
  let index = i++
  console.log(`递归了 ${index}次`)
  setTimeout(function(){
    task()
  })
  console.log(`第 ${index}次递归结束`)
}
task()
```

改造后的案例运行结果，如图 5-20 所示。

图 5-20　改造后的案例运行结果

仅做一个小改造，便不会出现栈溢出的错误。这个是因为改造后的案例使用了异步任务，以此去调用递归中的函数，这个函数在执行时，就不仅使用函数执行栈进行执行了。

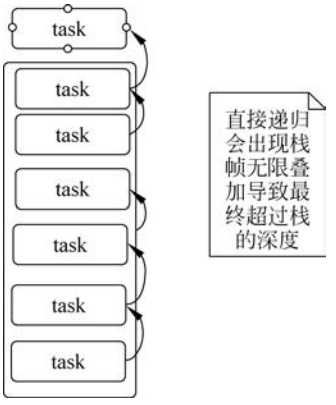

图 5-21　改造后的案例运行结果

接下来通过图形对比的方式加深对栈溢出的理解，当递归任务无休止执行时，函数执行栈的情况如图 5-21 所示。

当加入了异步调用递归函数代码后，递归的流程不仅利用了函数执行栈，还利用了事件循环，如图 5-22 所示。

有了异步任务后，递归便不会叠加栈帧了。因为放入工作线程后，该函数就结束了，可以出栈销毁，在执行栈中永远只有一个任务在运行。这样便防止了栈帧的无限叠加，从而解决了无限递归的问题。不过异步递归的过程是无法保证运行效率的，在实际的工作场景中，如果考虑性能问题，则需要使用 while 循环等

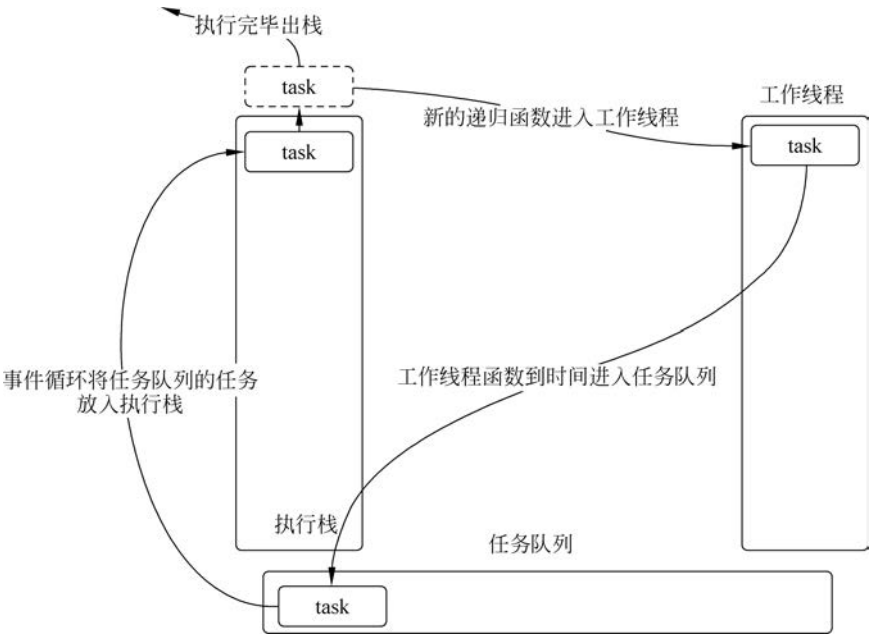

图 5-22　递归的流程利用函数执行栈和事件循环

解决方案，以此来保证运行效率。在实际工作场景中，应尽量避免递归循环，因为递归循环就算控制在有限栈帧的叠加，其性能也远远不及指针循环。

5.4　异步流程控制

5.4.1　宏任务与微任务

在明确事件循环模型及 JavaScript 的执行流程后，又认识了一个叫作任务队列的容器，它的数据结构为队列结构。所有除同步任务外的代码都会在工作线程中，按照到达的执行时机有序进入任务队列。

任务队列中的异步任务又分为宏任务和微任务。

1. 生活中的例子

在了解宏任务和微任务前，还是用生活中的实际场景举个例子：

在去银行办理业务时，每个人都需要在进入银行时，找到取票机进行取票，这个操作会把来办理业务的人，按照取票的顺序排成一个有序的队列（这个队列可以理解成异步任务队列）。

假设银行只开通了一个办事窗口，窗口的工作人员会按照排队的顺序进行叫号，到达号码的人就可以前往窗口办理业务（窗口可以理解为函数执行栈）。在第 1 个人办理业务的过程中，第 2 个以后的人都需要进行等待。这个场景与 JavaScript 的异步任务队列的执行场景是完全相同的。如果把每个办业务的人当作 JavaScript 中的每个异步的任务，则取号就

相当于将异步任务放入任务队列。银行的窗口就相当于函数执行栈，在叫号时代表将当前队列的第 1 个任务放入函数执行栈运行。

可能每个人在窗口办理的业务内容各不相同，例如，第 1 个人仅仅进行开卡操作，银行工作人员就会为其执行开卡流程，这就相当于执行异步任务内部的代码。在实际生活中，若第 1 个人的银行卡开通完毕，则银行的工作人员不会立即叫第 2 个人过来，而会询问第 1 个人："您是否需要为刚才开通的卡办理一些增值业务，例如活期储蓄"，这相当于在原开卡的业务流程中临时追加了一个新的任务。

若按照 JavaScript 的默认执行顺序，则这个人的新任务应该回到取票机取一张新的号码，再去队尾重新排队，但如果这样工作，办事效率就会急剧下降，所以银行实际的做法是在叫下一个人办理业务前，若前面的人临时有新的业务要办理，则工作人员会继续为其办理业务，直到这个人的所有事情都办理完毕。

从取号到办理追加业务完成的这个过程，就是微任务的实际体现。在 JavaScript 运行环境中，包括主线程代码在内，可以理解为所有的任务内部都存在一个微任务队列，在下一个宏任务执行前，事件循环系统都会先检测当前的代码块中是否包含已经注册的微任务，并将队列中的微任务优先执行完毕，进而执行下一个宏任务，实际的任务队列的结构如图 5-23 所示。

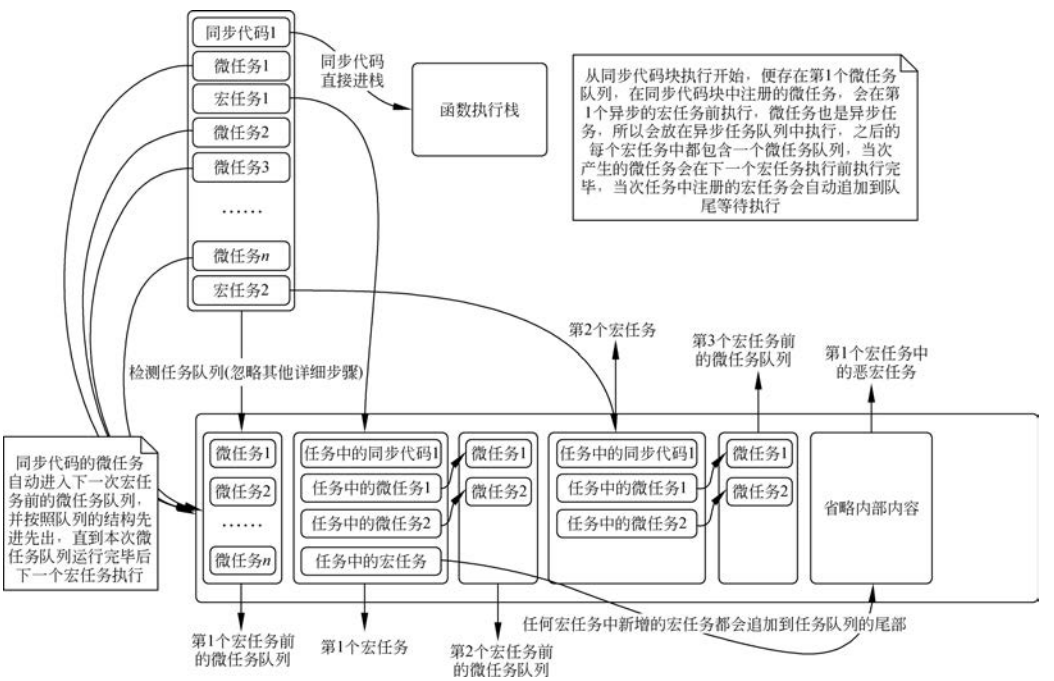

图 5-23　实际的任务队列的结构

2. 宏任务与微任务的介绍

JavaScript 中存在两种异步任务，一种是宏任务，另一种是微任务，它们的特点如下。

1）宏任务

宏任务是 JavaScript 中最原始的异步任务，包括 setTimeout()、setInterval()、AJAX 等，在代码执行环境中按照同步代码的顺序，逐个进入工作线程挂起，再按照异步任务到达的时间节点，逐个进入异步任务队列，最终按照队列中的顺序进入函数执行栈进行执行。

2）微任务

微任务是随着 ECMA 标准升级提出的新的异步任务，微任务在异步任务队列的基础上增加了微任务的概念，每个宏任务执行前，程序会先检测其中是否有当次事件循环未执行的微任务，优先清空本次的微任务后，再执行下一个宏任务，每个宏任务内部可注册当次任务的微任务队列，在下一个宏任务执行前运行，微任务也按照进入队列的顺序执行。

综上所述，在 JavaScript 的运行环境中，代码的执行流程如下：

（1）默认的同步代码按照顺序从上到下、从左到右运行，运行过程中注册本次的微任务和后续的宏任务。

（2）执行本次同步代码中注册的微任务，并向任务队列注册微任务中包含的宏任务和微任务。

（3）将下一个宏任务开始前的所有微任务执行完毕。

（4）执行最先进入队列的宏任务，并注册当次的微任务和后续的宏任务，宏任务会按照当前任务队列的队尾继续向下排列。

3. 常见的宏任务和微任务划分

常见的浏览器与 Node.js 环境下的宏任务列表如图 5-24 所示。

#	浏览器	Node
I/O	☑	☑
setTimeout	☑	☑
setInterval	☑	☑
setImmediate	✗	☑
requestAnimationFrame	☑	✗

图 5-24　常见的浏览器与 Node.js 环境下的宏任务列表

常见的浏览器与 Node.js 环境下的微任务列表如图 5-25 所示。

#	浏览器	Node
process.nextTick	✗	☑
MutationObserver	☑	✗
Promise.then catch finally	☑	☑

图 5-25　常见的浏览器与 Node.js 环境下的微任务列表

4. 一道经典的输出顺序笔试题

观察代码分析案例中代码的输出顺序，代码如下：

```
//第 5 章 5.4.1 观察代码分析案例中代码的输出顺序
setTimeout(function() {console.log('timer1')}, 0)

requestAnimationFrame(function(){
    console.log('UI update')
})

setTimeout(function() {console.log('timer2')}, 0)

new Promise(function executor(resolve) {
    console.log('promise 1')
    resolve()
    console.log('promise 2')
}).then(function() {
    console.log('promise then')
})

console.log('end')
```

按照同步先行，异步靠后的原则，在阅读代码时，先分析同步代码和异步代码。Promise 对象虽然是微任务，但是在执行语句 new Promise()时，回调函数是同步执行的，所以优先输出 promise 1 和 promise 2。

在 resolve 执行时，Promise 对象的状态变更为已完成，所以 then 函数的回调被注册到微任务事件中，此时并不执行，接下来应该输出 end。

同步代码执行结束后，观察异步代码的宏任务和微任务，在本次的同步代码块中注册的微任务会优先执行，参考上文中描述的列表，Promise 为微任务，setTimeout()和 requestAnimationFrame()为宏任务，所以 Promise 的异步任务会在下一个宏任务执行前执行，promise then 是第 4 个输出的结果。

接下来参考 setTimeout()和 requestAnimationFrame()两个宏任务，这里的运行结果有多种情况。如果 3 个宏任务都为 setTimeout()，则会按照代码编写的顺序执行宏任务，而中间包含了一个 requestAnimationFrame()，这里要回顾一下它们的执行时机了。setTimeout()是在程序运行到 setTimeout()时，立即注册一个宏任务，所以两个 setTimeout()的顺序一定是固定的，即 timer1 和 timer2 会先后输出，而 requestAnimationFrame()是请求下一次重绘事件，所以它的执行频率要参考浏览器的刷新率。

接下来参考一个计算 requestAnimationFrame()频率的案例，代码如下：

```
//第 5 章 5.4.1 计算 requestAnimationFrame()频率的案例
let i = 0;
let d = new Date().getTime()
let d1 = new Date().getTime()
```

```
function loop(){
  d1 = new Date().getTime()
  i++
  //当间隔时间超过 1s 时执行
  if((d1 - d)>=1000){
    d = d1
    console.log(i)
    i = 0
    console.log('经过了 1s')
  }
  requestAnimationFrame(loop)
}
loop()
```

该代码在浏览器运行时，控制台会每间隔 1s 进行一次输出，输出的 i 就是 loop 函数执行的次数，计算 requestAnimationFrame()频率的运行结果，如图 5-26 所示。

图 5-26　计算 requestAnimationFrame()频率的运行结果

该输出意味着 requestAnimationFrame()函数的执行频率是每秒 60 次左右，它按照浏览器的刷新率进行执行，即屏幕刷新一次，该函数就触发一次，运行间隔约为 16ms。

接下来参考一个计算 setTimeout()执行频率的案例，代码如下：

```
//第 5 章 5.4.1 计算 setTimeout()执行频率的案例
let i = 0;
let d = new Date().getTime()
let d1 = new Date().getTime()

function loop(){
  d1 = new Date().getTime()
  i++
  if((d1 - d)>=1000){
    d = d1
    console.log(i)
    i = 0
    console.log('经过了 1s')
```

```
    }
    setTimeout(loop,0)
}
loop()
```

该代码结构与上一个案例类似，循环采用 setTimeout()进行控制，setTimeout()执行频率的案例的运行结果如图 5-27 所示。

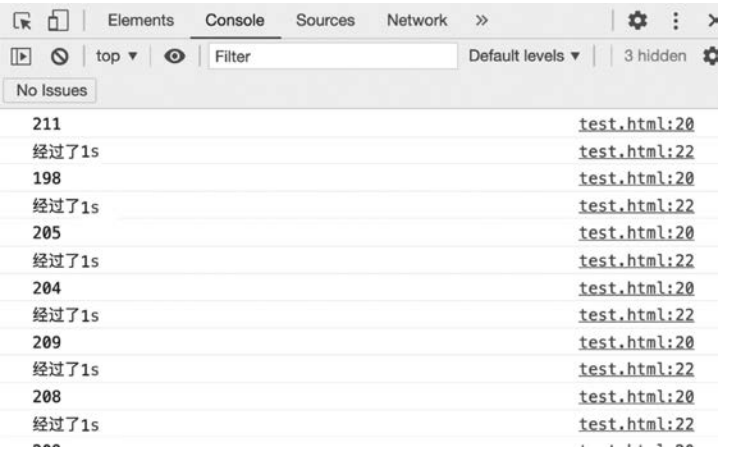

图 5-27　setTimeout()执行频率的案例的运行结果

根据运行结果得知，setTimeout(fn,0)的执行频率为每秒执行 200 次左右，所以它的间隔为 5ms 左右。

由于这两种异步的宏任务触发时机和执行频率不同，所以导致案例存在多种运行结果。若打开网页时，恰好 5ms 内执行了网页的重绘事件，则 requestAnimationFrame()在工作线程中就会到达触发时机优先进入任务队列，表达此时顺序的代码如下：

```
UI update->timer1->timer2。
```

而当打开网页时，上一次的重绘刚结束，下一次重绘的触发是 16ms 后，此时 setTimeout()注册的两个任务，在工作线程中会优先进入触发时机，这时输出的结果如下：

```
timer1->timer2->UI update。
```

极特殊的情况，上一次重绘导致本次 requestAnimationFrame()的执行时机恰好在网页打开 5ms 左右时，该时间极小概率会介于两个 setTimeout()之间，此种情况会得到下面的结果，代码如下：

```
timer1->UI update->timer2
```

这种情况出现的概率极低，但概率不为 0。

5.4.2　流程控制的银弹——Promise

1. Promise 简介

JavaScript 是一门典型的异步编程脚本语言，在编程过程中会大量出现异步代码，在 JavaScript 的整个发展历程中，对异步编程的处理方式经历了很多个时代，其中最典型也是现今使用最广泛的时代，便是 Promise 对象处理异步编程的时代。那什么是 Promise 对象呢？

Promise 是 ES6 版本提案中实现的异步处理方式，对象代表了未来将要发生的事件，用来传递异步操作的消息。

2. 为什么使用 Promise 对象

在过去的编程中，JavaScript 的主要异步处理方式是采用回调函数的方式进行处理，若要保证多个步骤的异步编程有序进行，则会出现下列情况，代码如下：

```
//第 5 章 5.4.2 若要保证多个步骤的异步编程有序进行
setTimeout(function(){
  //第 1 秒后执行的逻辑
  console.log('第 1 秒之后发生的事情')
  setTimeout(function(){
    //第 2 秒后执行的逻辑
      console.log('第 2 秒之后发生的事情')
    setTimeout(function(){
      //第 3 秒后执行的逻辑
      console.log('第 3 秒之后发生的事情')
    },1000)
  },1000)
},1000)
```

如案例中描述，若每间隔 1s 运行 1 个任务，则这 3 个任务必须按时间顺序执行，并且下一秒执行前，都要先获得上一秒运行的结果，所以不得不将代码编写为以上案例中的结构。该写法主要为了保证代码的执行顺序，这样避免不了在回调函数中嵌套大量的逻辑代码，这也是人们常说的“回调地狱”。

在实际编程中，上述案例的使用场景极少。在前端开发过程中，使用较多的异步流程为 AJAX 请求结构。当要求某个页面的多个接口保证有序调用时，开发者可能会采用嵌套结构实现，代码如下：

```
//第 5 章 5.4.2 当要求某个页面的多个接口保证有序调用时
//获取类型数据
$.ajax({
    url:'/***',
  success:function(res){
    var xxId = res.id
    //获取该类型的数据集合，必须等待回调执行才能进行下一步
```

```
      $.ajax({
        url:'/***',
        data:{
          xxId:xxId,                    //使用上一个请求结果作为参数调用下一个接口
        },
        success:function(res1){
          //得到指定类型集合
          ...

        }
      })
    }
})
```

这种情况在很多开发者的代码中都出现过。如果流程复杂化，在网络请求中继续夹杂其他异步流程，则这样的代码会变得难以维护。

其他异步场景，诸如 Node.js 文件中的原始 fs 模块等异步流程，在复杂业务场景中都避免不了这种嵌套结构。ECMA 提案中之所以出现 Promise 解决方案，便是为解决 JavaScript 在开发过程中遇到的实际问题，即“回调地狱”。其实解决“回调地狱”问题还有其他方案，本节不介绍中间的过渡方案，以 Promise 流程控制对象为主，因为它是解决“回调地狱”问题的银弹。

3. 使用 Promise 解决“回调地狱”问题

上文内容仅抛出问题，并没有针对问题做出合理的回答。接下来阐述如何使用 Promise 对象解决“回调地狱”问题。

在阐述前，先对 Promise 做一个简单的介绍：Promise 对象以链式调用的结构，将原本回调嵌套的异步处理流程，转化成“对象.then().then()…”的链式结构，虽然这种结构仍离不开回调函数，但将原本的回调嵌套结构，转换成连续调用结构，这样便可以采用“从上到下、从左至右”的方式进行阅读。

接下来仍然以 setTimeout()场景为例，改造上文的异步案例，代码如下：

```
//第 5 章 5.4.2 以 setTimeout()场景为例,改造上文的异步案例
//使用 Promise 拆解的 setTimeout 流程控制
var p = new Promise(function(resolve){
  setTimeout(function(){
    resolve()
  },1000)
})
p.then(function(){
  //第 1 秒后执行的逻辑
  console.log('第 1 秒之后发生的事情')
  return new Promise(function(resolve){
    setTimeout(function(){
      resolve()
```

```
    },1000)
  })
}).then(function(){
  //第 2 秒后执行的逻辑
  console.log('第 2 秒之后发生的事情')
  return new Promise(function(resolve){
    setTimeout(function(){
      resolve()
    },1000)
  })
}).then(function(){
  //第 3 秒后执行的逻辑
  console.log('第 3 秒之后发生的事情')
})
```

阅读案例会发现，使用 Promise 后的代码，将原来的 3 个 setTimeout()的回调嵌套，拆解成了 3 个 then()包裹的回调函数，按照上下顺序进行编写。这样从视觉上便可以按照人类“从上到下、从左到右”的线性思维来阅读代码，直观地查看这段代码的执行流程，其代价增加了接近 1 倍的代码量。

从以上案例得知，Promise 的作用是解决“回调地狱”问题，它的解决方式是将回调嵌套拆成链式调用，这样便可以按照上下顺序进行异步代码的流程控制。

5.4.3　回调函数与 Promise 对象

Promise 对象是一个 JavaScript 对象，在支持 ES6 语法的运行环境中，自动出现在全局对象中，它的初始化方式，代码如下：

```
//fn:是在初始化过程中调用的函数,是同步的回调函数
var p = new Promise(fn)
```

1. 重新理解回调函数

这里涉及一个概念：在 JavaScript 语言中，有一个特殊的函数叫作回调函数。回调函数的特点是把函数作为变量看待，由于 JavaScript 变量可以作为函数的形参，并且函数可以匿名创建，所以在定义函数时，可将一个函数的参数当作另一个函数来执行，代码如下：

```
//第 5 章 5.4.3 在定义函数时,可将一个函数的参数当作另一个函数来执行
//把 fn 当作函数对象就可以在 test 函数中使用()执行它了
function test(fn){
  fn()
}
//那么运行 test 时 fn 也会随着执行,所以向 test()中传入的匿名函数会运行
test(function(){
  ...
})
```

案例中的结构为 JavaScript 中典型的回调函数结构。按照事件循环中介绍的 JavaScript 函数运行机制，会发现其实回调函数本是同步代码，这是一个需要重点理解的知识点。

通常在编写 JavaScript 代码时，使用的回调嵌套的形式大多是异步流程，所以一些开发者可能会下意识地认为，凡是回调形式的函数都是异步流程。其实并不是这样的，真正的解释是：JavaScript 中的回调函数结构，默认为同步结构，由于 JavaScript 单线程异步模型的规则，若要编写异步代码，则必须使用回调嵌套的形式才能实现，所以回调函数结构不一定是异步流程，但是异步流程一定靠回调函数结构实现。

接下来通过一个简单的案例，理解回调函数与同步和异步的关系，代码如下：

```
//第 5 章 5.4.3 理解回调函数与同步和异步的关系
//同步的回调函数案例
function test(fn){
  fn()
}
console.log(1)
test(function(){
  console.log(2)
})
console.log(3)
//这段代码的输出顺序应该是 1、2、3，因为它属于直接进入执行栈的程序，会按照正常程序解析的
//流程输出

//异步的回调函数案例
function test(fn){
  setTimeout(fn,0)
}
console.log(1)
test(function(){
  console.log(2)
})
console.log(3)
//这段代码会输出 1、3、2，因为在调用 test()时 settimeout()会将 fn 放到异步任务队列挂起，
//等待主程序执行完毕后才会执行
```

2. 为什么异步流程要靠回调结构实现

思考一下，假设有一个变量 a 的值为 0，想要 1s 之后将 a 的值设置为 1，并且在这之后想要得到 a 的新结果。在这个逻辑中，若 1s 后将 a 设置为 1 采用的是 setTimeout()，则能否通过同步结构实现？参考下面的案例，代码如下：

```
//第 5 章 5.4.3 若 1s 后将 a 设置为 1 采用的是 setTimeout()，则能否通过同步结构实现
var a = 0
setTimeout(function(){
  a = 1
},1000)
console.log(a)
```

该代码块的输出结果一定为 0，由 JavaScript 单线程异步模型得知，当前代码块中 setTimeout()的回调函数是一个宏任务，会在本次的同步代码执行完毕后执行，所以声明 a=0 和输出 a 的值这两行代码会优先执行。这时对 a 赋值 1 的事件还没有发生，所以输出的结果就一定为 0。

接下来对代码做如下改造，试图使用阻塞的方式获取异步代码的结果，代码如下：

```
//第 5 章 5.4.3 试图使用阻塞的方式获取异步代码的结果
var a = 0
//依然使用 setTimeout 设置 1s 的延迟，以便设置 a 的值
setTimeout(function(){
  a = 1
},1000)
var d = new Date().getTime()
var d1 = new Date().getTime()
//采用 while 循环配合时间差来阻塞同步代码 2s
while(d1 - d < 2000){
  d1 = new Date().getTime()
}
console.log(a)
```

本案例的同步代码会在 while 循环中阻塞 2s，所以 console.log(a)这行代码会在 2s 后才能获得执行资源，但最终输出的结果仍然是 0，原因很简单：由 JavaScript 的运行模型进行理解，单线程异步模型的规则是严格的同步在前而异步靠后的顺序，虽然本案例的同步代码阻塞 2s，已经超过了 setTimeout()的等待时间，但 setTimeout()中的宏任务到时间后，仅仅会被从工作线程移动到任务队列中进行等待。在时间到达 1s 时，while 循环没有执行结束，所以函数执行栈会被继续占用，直到循环释放并输出 a 后，函数执行栈才被清空，任务队列中的宏任务才能被执行，所以这里就算 setTimeout()时间到了，也必须等待同步代码执行完毕。当输出 a 时，a=1 的行为仍然没有发生，所以默认的上下结构永远得不到异步回调中的结果，这也是异步流程都是回调函数结构的原因。

综上所述，想要真正地在 2s 后获取 a 的新结果，代码如下：

```
//第 5 章 5.4.3 想要真正地在 2s 后获取 a 的新结果
//只有在这个回调函数中才能获取 a 改造之后的结果
var a = 0
setTimeout(function(){
  a = 1
},1000)
//注册一个新的宏任务，让它在上一个宏任务后执行
setTimeout(function(){
  console.log(a)
},2000)
```

到这里，回调函数的意义及使用场景已经阐述得非常明确，深入研究回调函数是因为 Promise 对象是一个极特殊的存在，Promise 中既包含同步的回调函数，又包含异步的回调函数。

5.4.4 Promise 对象应用详细讲解

1. Promise 的执行顺序

参考一个 Promise 的应用案例，代码如下：

```
//第 5 章 5.4.4 一个 Promise 的应用案例
//实例化一个 Promise 对象
var p = new Promise(function(resolve,reject){

})
//通过链式调用控制流程
p.then(function(){
  console.log('then 执行')
}).catch(function(){
  console.log('catch 执行')
}).finally(function(){
  console.log('finally 执行')
})
```

参考案例中的 Promise 对象结构，一个 Promise 对象包含两部分回调函数，第一部分是执行语句 new Promise()时传入的对象，该回调函数是同步的，而 then()、catch()及 finally()中的回调函数是异步的，这里提前记好。接下来可以执行该程序，会发现这段程序并没有任何输出，继续改造 Promise 的案例，代码如下：

```
//第 5 章 5.4.4 继续改造 Promise 的案例
console.log('起步')
var p = new Promise(function(resolve,reject){
  console.log('调用 resolve')
  resolve('执行了 resolve')
})
p.then(function(res){
  console.log(res)
  console.log('then 执行')
}).catch(function(){
  console.log('catch 执行')
}).finally(function(){
  console.log('finally 执行')
})
console.log('结束')
```

这段程序的输出结果为起步、调用 resolve()、结束、执行 resolve()、执行 then()、执行 finally()。

接下来将 resolve()函数去掉，改成调用 reject()函数，代码如下：

```
//第 5 章 5.4.4 将 resolve()函数去掉,改成调用 reject()函数
console.log('起步')
```

```
var p = new Promise(function(resolve,reject){
  console.log('调用 reject')
  reject('执行了 reject')
})
p.then(function(res){
  console.log(res)
  console.log('then 执行')
}).catch(function(res){
  console.log(res)
  console.log('catch 执行')
}).finally(function(){
  console.log('finally 执行')
})
console.log('结束')
```

这段程序的输出结果为起步、调用 reject()、结束、执行 reject()、执行 catch()、执行 finally()。

经过对案例的学习，可以明确了解 Promise 的结构和运行流程。从运行流程上发现，语句 new Promise()中的回调函数的确是同步任务，如果这个回调函数内部没有执行 resolve()或 reject()，则 then()、catch()和 finally()回调函数均不会执行。运行 resolve()函数后，then()和 finally()会执行，而运行 reject()后，catch()和 finally()会执行。

2. Promise 结构

Promise 对象相当于一个未知状态的对象，相当于声明一个等待未来结果的对象：在结果发生前，它一直是初始状态；在结果发生后，它会变成其中一种目标状态。Promise 的中文翻译为保证，很多国外电影的台词都会出现 Promise 这个单词，Promise 在英文中代表非常强烈的语气词。在编程中 Promise 对象是一个非常严谨的对象，一定会按照约定执行，不会出现任何非预测结果(除使用不当外)。

Promise 自身具备以下 3 种状态。

(1) pending：初始状态，也叫就绪状态。这是在 Promise 对象定义初期的状态，这时 Promise 仅仅做了初始化，并注册对象上所有的任务。

(2) fulfilled：已完成，通常代表成功地执行了某个任务。当初始化函数中的 resolve()执行时，Promise 的状态就变更为 fulfilled，并且 then()函数注册的回调函数会开始执行，resolve()中传递的参数会进入回调函数作为形参。

(3) rejected：已拒绝，通常代表执行了一次失败任务，或者流程中断。当调用 reject()函数时，catch()中注册的回调函数会被触发，并且 reject()中传递的内容会变成回调函数的参数。

需要注意的是，处于 pending 状态时，Promise 会一直等待 resolve()或 reject()被执行，它们任意时候执行，then()或 catch()都会被触发。

Promise 约定，当对象创建后，同一个 Promise 对象，只能从 pending 状态变更为 fulfilled 或 rejected 状态的其中一种，状态一旦变更就不会再改变，此时 Promise 对象的流程执行完

成且执行 finally()函数。

3. 通过案例巩固理论

根据上文的分析,结合接下来的代码案例,继续学习 Promise 的规则,分析该对象的运行结果,代码如下:

```
//第 5 章 5.4.4 分析该对象的运行结果
//案例 1
new Promise(function(resolve,reject){
  resolve()
  reject()
}).then(function(){
  console.log('then 执行')
}).catch(function(){
  console.log('catch 执行')
}).finally(function(){
  console.log('finally 执行')
})
//结果顺序:then 执行 -> finally 执行

//案例 2
new Promise(function(resolve,reject){
  reject()
  resolve()
}).then(function(){
  console.log('then 执行')
}).catch(function(){
  console.log('catch 执行')
}).finally(function(){
  console.log('finally 执行')
})
//结果顺序:catch 执行 -> finally 执行

//案例 3
new Promise(function(resolve,reject){
}).then(function(){
  console.log('then 执行')
}).catch(function(){
  console.log('catch 执行')
}).finally(function(){
  console.log('finally 执行')
})
//不会产生任何结果
```

通过案例运行,再次巩固了 Promise 对象执行流程的印象。Promise 的异步回调部分如何执行,取决于初始化函数中的操作。一旦在初始化函数中调用 resolve(),再执行 reject()也不会影响 then()执行,此时 catch()也不会执行,反之同理,而在初始化回调函数中,如果不进行任何操作,Promise 的状态仍然是 pending,则所有注册的回调函数都不会执行。

5.4.5　链式调用及其他常用 API

链式调用这种编程方式最经典的使用，体现在 JQuery 框架中。很多语言到现在还在使用这种优雅的语法（不限前端或后台），接下来简单认识一下什么是链式调用。

为什么 Promise 对象可以“.then().catch()…”这样调用，甚至还能调用“.then().then()…”调用。其本质的链式调用原理，代码如下：

```
//第 5 章 5.4.5 其本质的链式调用原理
function MyPromise(){
  return this
}
MyPromise.prototype.then = function(){
  console.log('触发了 then')
  return this //new MyPromise()
}
new MyPromise().then().then().then()
```

其实，链式调用的本质是：在调用任意的函数执行到最后时，它又返回了一个调用对象或与调用对象相同的新实例对象，这两种方式都可以实现链式调用。

接下来，运行下面的案例，学习 Promise 对象的结构，代码如下：

```
//第 5 章 5.4.5 运行下面的案例,学习 Promise 对象的结构
var p = new Promise(function(resolve,reject){
  resolve('我是 Promise 的值')
})
console.log(p)
```

该案例运行后，会在控制台上得到以下内容，代码如下：

```
//第 5 章 5.4.5 该案例运行后,会在控制台上得到以下内容
Promise {<fulfilled>: '我是 Promise 的值'}
[[Prototype]]: Promise
[[PromiseState]]: "fulfilled"
[[PromiseResult]]: "我是 Promise 的值"
```

该结构的详细说明如下：

(1) [[Prototype]]代表 Promise 的原型对象。

(2) [[PromiseState]]代表 Promise 对象当前的状态。

(3) [[PromiseResult]]代表 Promise 对象的值，分别对应 resolve()或 reject()传入的结果。

1. 链式调用的注意事项

接下来通过 Promise 链式调用的程序案例，继续学习链式调用的特点，代码如下：

```
//第 5 章 5.4.5 Promise 链式调用的程序案例
var p = new Promise(function(resolve,reject){
```

```
    resolve('我是 Promise 的值')
  })
  console.log(p)
  p.then(function(res){
    //该 res 的结果是 resolve 传递的参数
    console.log(res)
  }).then(function(res){
    //该 res 的结果是 undefined
    console.log(res)
    return '123'
  }).then(function(res){
    //该 res 的结果是 123
    console.log(res)
    return new Promise(function(resolve){
      resolve(456)
    })
  }).then(function(res){
    //该 res 的结果是 456
    console.log(res)
    return '我是直接返回的结果'
  }).then()
    .then('我是字符串')
    .then(function(res){
    //该 res 的结果是"我是直接返回的结果"
    console.log(res)
  })
  /*
  该案例的输出结果
  Promise{<fulfilled>: '我是 Promise 的值'}
  ttt.html:16 我是 Promise 的值
  ttt.html:18 undefined
  ttt.html:21 123
  ttt.html:26 456
  ttt.html:31 我是直接返回的结果
   */
```

根据运行结果，可以分析出链式调用的基本规则如下：

（1）只要有 then()且触发了 resolve()，整个链条就会执行到结尾，这个过程中的第 1 个回调函数的参数是由 resolve()传入的值。

（2）Promise 对象的每个回调函数，都可以使用 return 返回一个结果，如果没有返回结果，则下一个 then()中回调函数的参数就是 undefined。

（3）Promise 的任意回调函数的返回结果，如果是普通类型的数据，则该值为下一个 then()中回调函数的参数。

（4）若 Promise 某个回调函数返回的内容是一个 Promise 对象，则这个 Promise 对象是 resolve()的参数，会成为下一个 then()中回调的函数的参数（可以暂时当作：返回 Promise 对象时，下一个 then()就是该对象的 then()，但内部代码并不是这样执行的）。

(5) 如果 then()中传入的不是函数或未传入任何内容,则 Promise 链条并不会中断 then 的链式调用,并且在这之前最后一次 then()中回调函数的返回结果,会直接进入离它最近的正确的 then()中的回调函数作为参数。

2. 中断链式调用

链式调用可以被中断吗? 答案是肯定的。有两种形式可以让 then()的链条中断,如果中断链式调用,则会触发一次 catch()中的回调函数执行。中断链式调用的案例的代码如下:

```
var p = new Promise(function(resolve,reject){
  resolve('我是 Promise 的值')
})
console.log(p)
p.then(function(res){
  console.log(res)
}).then(function(res){
  //有两种方式可以中断 Promise
  //throw('我是中断的原因')
  return Promise.reject('我是中断的原因')
}).then(function(res){
  console.log(res)

}).then(function(res){
  console.log(res)

}).catch(function(err){
  console.log(err)
})
/*结果如下:
Promise {<fulfilled>: '我是 Promise 的值'}
ttt.html:16 我是 Promise 的值
ttt.html:26 我是中断的原因
*/
```

运行案例会发现,中断链式调用后,会触发 catch()中的回调函数,并且从中断开始到 catch()中间的 then()的回调函数都不会执行,这样链式调用的流程便会结束。

中断的方式有两种:

(1) 抛出一个异常。

(2) 返回一个 rejected 状态的 Promise 对象。

3. 中断链式调用是否违背了 Promise 的精神

在介绍 Promise 时,强调了 Promise 是绝对保证的意思,并且 Promise 对象的状态一旦变更就不会再发生变化。当使用链式调用时,正常都是 then()中的回调函数连续,但触发中断时,catch()中的回调却执行了。按照约定规则 then()中的回调函数执行,就代表 Promise 对象的状态已经变更为 fulfilled 了,但是中断链式调用后,catch()中的函数却执行了。catch()中的回调函数执行,意味着 Promise 对象的状态变成了 rejected,这代表当前链

式调用时，Promise 的状态从 fulfilled 变成了 rejected。

按照上面的理解，中断链式调用恰恰违背了 Promise 的约定，若深入挖掘 Promise 对象的执行逻辑，则会发现上面的推断是不成立的。接下来通过一段简单的代码，了解 Promise 链式调用的细节，代码如下：

```
//第 5 章 5.4.5 了解 Promise 链式调用的细节
var p = new Promise(function(resolve,reject){
  resolve('我是 Promise 的值')
})
var p1 = p.then(function(res){

})
console.log(p)
console.log(p1)
console.log(p1 === p)
/*
运行结果:
Promise {<fulfilled>: '我是 Promise 的值'}
ttt.html:18 Promise {<pending>}
ttt.html:19 false
*/
```

运行案例会发现，返回的 p 和 p1 的状态不同，并且它们的比较结果是 false，这就代表它们在堆内存中并没有保存在同一个位置。p 和 p1 对象分别保存了两个 Promise 对象的引用地址，虽然 then()函数每次都返回一个 Promise 对象，实现链式调用，但 then()函数每次返回的都是一个新的 Promise 对象，这样便解释得通了。也就是说，每次 then()的回调函数在执行时都可以让本次的结果，在下一个异步步骤执行时变成不同的状态，并且不违背 Promise 对象最初的约定，因为每次 then()和 catch()的回调，都是异步执行且由不同的 Promise 对象控制的。

根据以上的分析，已经掌握了 Promise 在运行时的规则。这样就能解释得通，为什么最初通过 Promise 控制 setTimeout()每秒执行一次的功能可以实现，这是因为当使用 then()函数进行链式调用时，可以利用返回一个新的 Promise 对象，来执行下一次 then()的回调函数，而下一次 then()的回调函数的执行，必须等待其内部的 resolve()调用，这样在执行语句 new Promise()时，放入 setTimeout()进行延时，保证 1s 之后让状态变更，这样就能不编写回调嵌套便能实现连续地执行异步流程了。

4. Promise 常用 API 介绍

当代码中需要使用异步流程控制时，可以通过 then()的链式调用，实现异步流程按约定的顺序执行。假设在实际案例中，某个模块的页面需要同时调用 3 个服务器端接口：a、b 和 c，需要保证 3 个接口的数据全部返回后才能渲染页面。假设 a 接口耗时 1s，b 接口耗时 0.8s，c 接口耗时 1.4s，若只用 then()的链式调用来进行流程控制，虽然可以保证满足需求，但是通过 then()函数的异步控制，必须等待前一个接口回调执行完毕才能调用下一个接

口，这样总耗时为 1＋0.8＋1.4 ＝ 3.2s。这种累加显然增加了接口调用的时间消耗，所以 Promise 提供了 all()方法，以此来解决批量异步流程处理的问题，代码如下：

```
Promise.all([Promise 对象, Promise 对象,...]).then(回调函数)
```

回调函数的参数是一个数组，按照第 1 个参数的 Promise 对象的顺序，展示每个 Promise 的返回结果。

可以借助 Promise.all()实现，等最慢的接口返回数据后，一起得到所有接口的数据，那么总耗时将只会为最慢接口的消耗时间 1.4s，总共节省了 1.8s。Promise.all()的实际应用方式的代码如下：

```
//第 5 章 5.4.5 Promise.all()的实际应用方式
//promise.all 相当于统一处理了
//多个 promise 任务，保证处理的这些所有 promise
//对象的状态全部变成为 fulfilled 之后才会触发 all 的
//then()函数来保证将放置在 all 中的所有任务的结果返回
let p1 = new Promise((resolve,reject) => {
  setTimeout(() => {
    resolve('第 1 个 promise 执行完毕')
  },1000)
})
let p2 = new Promise((resolve,reject) => {
  setTimeout(() => {
    resolve('第 2 个 promise 执行完毕')
  },2000)
})
let p3 = new Promise((resolve,reject) => {
  setTimeout(() => {
    resolve('第 3 个 promise 执行完毕')
  },3000)
})
Promise.all([p1,p3,p2]).then(res => {
  console.log(res)
}).catch(function(err){
  console.log(err)
})
```

Promise.all()可以批量地处理异步的 Promise 执行流程，等待最慢的状态变更后统一做下一步的任务处理，所以 Promise 对象存在 race()方法，用来竞争异步流程中最快执行完毕的任务。race()与 all()方法的使用格式相同：

```
Promise.race([Promise 对象, Promise 对象,...]).then(回调函数)
```

回调函数的参数是前面数组中最快一种状态变更的 Promise 对象的值。

race()方法的主要使用场景是什么？举个例子，为了保证用户可以获得较低的延迟，通常网页中的流媒体模块会提供多个媒体数据源。网站运营商希望用户在进入网页时，流媒

体数据为用户提供最快的数据源，这时便可以使用 Promise.race()来让多个数据源进行竞赛。得到竞赛结果后，将延迟最低的数据源，用于用户播放视频的默认数据源，该场景便是race()的典型使用场景。

Promise.race()的经典使用案例，代码如下：

```
//promise.race()相当于将传入的所有任务
//进行了一个竞争，它们之间最先将状态变成 fulfilled 的
//那一个任务就会直接触发 race 的.then 函数并且将它的值
//返回，主要在多个任务之间竞争时使用
let p1 = new Promise((resolve,reject) => {
  setTimeout(() => {
    resolve('第 1 个 promise 执行完毕')
  },5000)
})
let p2 = new Promise((resolve,reject) => {
  setTimeout(() => {
    reject('第 2 个 promise 执行完毕')
  },2000)
})
let p3 = new Promise(resolve => {
  setTimeout(() => {
    resolve('第 3 个 promise 执行完毕')
  },3000)
})
Promise.race([p1,p3,p2]).then(res => {
  console.log(res)
}).catch(function(err){
  console.error(err)
})
```

5.4.6 异步代码同步化

Promise 的能力非常大，使用模式非常自由。Promise 的链式调用结构，将 JavaScript 一个时代的弊病从此解套。该解套虽然比较成功，但如果直接使用 then()函数进行链式调用，则开发时代码量仍然是非常大的，想要开发一个非常复杂的异步流程，依然需要大量的链式调用来进行支撑，开发者会感觉非常难受。

按照人类的线性思维，虽然 JavaScript 可分为同步和异步，但是在单线程模式下，若能完全按照同步代码的编写方式来处理异步流程，这才是开发者最期待的结果，那么有没有办法让 Promise 对象能更进一步地接近同步代码呢？

1. Generator 函数的介绍

在 JavaScript 中存在这样一种函数，即 Generator 函数结构，代码如下：

```
function * 函数名称(){
  yield                    //部分代码逻辑
}
```

ES6 新引入了 Generator 函数，可以通过 yield 关键字中断函数的执行，这为改变同步函数的执行流程提供了可能。这种人为干预函数运行流程的结构，让原本一次执行完毕的函数不仅能分步运行，还可以人为对其中插入代码，为异步代码同步化提供了可能。

接下来，参考一个 Generator 函数的基本案例，代码如下：

```
//第 5 章 5.4.6 一个 Generator 函数的基本案例
/* 该函数和普通函数不同，在调用函数体时，函数主体代码并不执行，只会返回一个分步执行对象，
该对象存在 next()方法，用来让程序继续执行，当程序遇到 yield 关键字时会停顿. next()返回的对
象中包含 value 和 done 两个属性，value 代表上一个 yield 返回的结果，done 代表程序是否执行完
毕. */
function * test(){

  var a = yield 1
  console.log(a)
  var b = yield 2
  console.log(b)
  var c = a + b
  console.log(c)
}
//获取分步执行对象
var generator = test()
//输出
console.log(generator)
//步骤 1，该程序从起点执行到第 1 个 yield 关键字后，step1 的 value 是 yield 右侧的结果 1
var step1 = generator.next()
console.log(step1)
//步骤 2，该程序从 var a 开始执行到第 2 个 yield 后，step2 的 value 是 yield 右侧的结果 2
var step2 = generator.next()
console.log(step2)
//由于没有 yield，所以该程序从 var b 开始执行到结束
var step3 = generator.next()
console.log(step3)
```

查看案例中的注释并运行该程序，上面案例的执行结果，代码如下：

```
//第 5 章 5.4.6 上面案例的执行结果
test { < suspended > } [[ GeneratorLocation ]]: ttt. html: 10 [[ Prototype ]]: Generator
[[GeneratorState]]: "closed"[[GeneratorFunction]]: ƒ *
test()[[GeneratorReceiver]]: Window
ttt.html:21 {value: 1, done: false}
ttt.html:12 undefined
ttt.html:23 {value: 2, done: false}
ttt.html:14 undefined
ttt.html:16 NaN
ttt.html:25 {value: undefined, done: true}
```

查看结果会发现 a 和 b 的值不见了，c 的值也是 NaN。虽然程序实现了分步执行，但流程却出现了问题。

这是因为在分步执行过程中，需要在程序中对运行的结果进行人为干预，也就是说yield返回的结果和它左侧变量的值都是可以被人为干预的。

接下来改造上面的案例内容，代码如下：

```
//第5章 5.4.6 改造上面的案例内容
function * test(){
  var a = yield 1
  console.log(a)
  var b = yield 2
  console.log(b)
  var c = a + b
  console.log(c)
}
var generator = test()
console.log(generator)
var step1 = generator.next()
console.log(step1)
var step2 = generator.next(步骤1:value)
console.log(step2)
var step3 = generator.next(步骤2:value)
console.log(step3)
```

将代码改造，在generator.next()函数中追加参数后，会发现控制台中的数据可以正常输出，代码如下：

```
//第5章 5.4.6 控制台中的数据可以正常输出
test {< suspended >}
ttt.html:21 {value: 1, done: false}
ttt.html:12 1
ttt.html:23 {value: 2, done: false}
ttt.html:14 2
ttt.html:16 3
ttt.html:25 {value: undefined, done: true}
```

也就是说，在next()函数执行的过程中，是需要传递参数的。目前一次next()执行时，如果不传递参数，则本次yield左侧变量的值会变成undefined。若想让yield左侧的变量有值，就必须在next()中传入需要的结果。

2. Generator函数能控制什么样的流程

创建一个Generator函数，在其中编写不同的同步和异步流程，代码如下：

```
//第5章 5.4.6 创建一个Generator函数,在其中编写不同的同步和异步流程
function * test(){
  var a = yield 1
  console.log(a)
  var res = yield setTimeout(function(){
    return 123
```

```
  },1000)
  console.log(res)
  var res1 = yield new Promise(function(resolve){
    setTimeout(function(){
      resolve(456)
    },1000)
  })
  console.log(res1)
}
var generator = test()
console.log(generator)
var step1 = generator.next()
console.log(step1)
var step2 = generator.next()
console.log(step2)
var step3 = generator.next()
console.log(step3)
var step4 = generator.next()
console.log(step4)
```

接下来查看案例代码的运行结果，代码如下：

```
//第 5 章 5.4.6 案例代码的运行结果
test {< suspended >}
ttt.html:27 {value: 1, done: false}
ttt.html:12 undefined
ttt.html:29 {value: 1, done: false}
ttt.html:16 undefined
ttt.html:31 {value: Promise, done: false}
ttt.html:22 undefined
ttt.html:33 {value: undefined, done: true}
```

根据调用情况发现，当结果输出时，并没有体现任何延迟。进一步观察打印输出，会发现yield 右侧的普通变量，可以直接在 step1 的 value 中获得，当 yield 的右侧为 setTimeout()时，结果中只可以得到 setTimeout()的定时器编号(并不能识别定时任务何时完成)，而当 yield 的右侧为 Promise 对象时，可以获得 Promise 对象本身。接下来查看案例中输出的 Promise 对象，代码如下：

```
//第 5 章 5.4.6 案例中输出的 Promise 对象
{value: Promise, done: false}
done: false
value: Promise
[[Prototype]]: Promise
[[PromiseState]]: "fulfilled"
[[PromiseResult]]: 456
[[Prototype]]: Object
```

阅读结果会发现，yield 可以得到 Promise 内部的结果，所以能确保在分步过程中，

Generator 函数可以对 Promise 实现的异步代码流程进行控制。

3. 用 Generator 将 Promise 的异步流程同步化

通过上文的学习，可以通过递归调用的方式，动态地执行一个 Generator 函数，以 done 属性识别函数是否执行完毕，通过 next()函数来推动函数向下执行。若在执行过程中遇到了 Promise 对象，就等待 Promise 对状态进行变更，再进入下一步。

接下来，排除出现异常和 reject()调用的情况，封装一个动态执行的 Generator 函数，代码如下：

```
//第 5 章 5.4.6 排除出现异常和 reject()调用的情况,封装一个动态执行的 Generator 函数
/**
 * fn:Generator 函数对象
 */
function generatorFunctionRunner(fn){
  //定义分步对象
  let generator = fn()
  //执行到第 1 个 yield
  let step = generator.next()
  //定义递归函数
  function loop(stepArg,generator){
    //获取本次的 yield 右侧的结果
    let value = stepArg.value
    //判断结果是不是 Promise 对象
    if(value instanceof Promise){
      //如果是 Promise 对象就在 then()函数的回调中获取本次程序结果
      //并且等待回调执行时进入下一次递归
      value.then(function(promiseValue){
        if(stepArg.done == false){
          loop(generator.next(promiseValue),generator)
        }
      })
    }else{
      //如果判断程序没有执行完就将本次的结果传入下一步,进入下一次递归
      if(stepArg.done == false){
        loop(generator.next(stepArg.value),generator)
      }
    }
  }
  //执行动态调用
  loop(step,generator)
}
```

有了 generatorFunctionRunner()函数后，可以将最初的 Promise 控制 3 个 setTimeout()的案例转换成基于 Generator 函数的流程控制，代码如下：

```
//第 5 章 5.4.6 将最初的 Promise 控制 3 个 setTimeout()的案例转换成基于 Generator
//函数的流程控制
```

```
function * test(){
  var res1 = yield new Promise(function(resolve){
    setTimeout(function(){
      resolve('第 1 秒运行')
    },1000)
  })
  console.log(res1)
  var res2 = yield new Promise(function(resolve){
    setTimeout(function(){
      resolve('第 2 秒运行')
    },1000)
  })
  console.log(res2)
  var res3 = yield new Promise(function(resolve){
    setTimeout(function(){
      resolve('第 3 秒运行')
    },1000)
  })
  console.log(res3)
}
generatorFunctionRunner(test)
```

通过案例中的 generatorFunctionRunner()函数处理后，可以在控制台发现，运行结果每隔 1s 输出一行，代码如下：

```
第 1 秒运行
ttt.html:22 第 2 秒运行
ttt.html:28 第 3 秒运行
```

经过 yield 修饰符之后可以惊喜地发现，若忽略 generatorFunctionRunner()函数，在 Generator 函数中，则可以将 then()回调成功地规避。程序运行到 yield 修饰的 Promise 对象所在的行时，便会进入挂起状态，直到 Promise 对象的状态变更为 fulfilled，才会向下一行执行。这样便通过 Generator 函数对象，成功地将 Promise 的异步流程同步化了。

Generator 函数实现的异步代码同步化方式是 JavaScript 异步编程的一个过渡期。通过该解决方案，只需提前准备好类似 generatorFunctionRunner()的工具函数，便可以很轻松地使用 yield 关键字实现异步代码同步化。

4. 终极解决方案——async 和 await

经过 Generator 方案的过渡后，异步代码同步化的需求逐渐成为主流。替代 Generator 的新方案在 ES7 版本中被提出，并且在 ES8 版本中得到实现，提案中定义了全新的异步控制流程，代码如下：

```
//第 5 章 5.4.6 提案中定义的函数使用成对的修饰符
async function test(){
  await ...
  await ...
```

```
}
test()
```

阅读案例发现，新提案的编写方式与 Generator 函数结构类似。提案中规定，可以使用 async 修饰一个函数，这样便可以在该函数的直接子作用域中，使用 await 来控制函数的流程。await 右侧可以编写任何变量或对象，当 await 右侧为同步结构时，await 左侧会得到返回右侧的结果并继续向下执行，而当 await 右侧为 Promise 对象时，若 Promise 对象状态为 pending，则函数会挂起等待。直到 Promise 对象变成 fulfilled，程序才再向下执行，Promise 的值会自动返回 await 左侧的变量中。async 和 await 需要成对出现，async 可以单独修饰函数，但是 await 只能在被 async 修饰的函数中使用。

有了 async 与 await，就相当于使用了自带自动执行函数的 Generator 函数，这样便无须单独针对 Generator 函数进行开发了。ES8 的规则落地后，async 和 await 逐渐成为主流异步流程控制的终极解决方案，而 Generator 结构则慢慢淡出了业务开发的舞台，不过 Generator 函数的流程控制方案，成为向下兼容过渡期版本浏览器的解决方案。

虽然在现今的大部分项目的业务代码中，使用 Generator 函数的场景非常少，但是查看脚手架项目的编译结果，还是能发现大量的 Generator 函数，这是脚手架为支持 ES8 提案落地前的浏览器版本提供的解决方案。

接下来，进一步认识 async 函数的执行流程，创建一个 async 修饰的函数，查看其执行特点，代码如下：

```
//第 5 章 5.4.6 创建一个 async 修饰的函数,查看其执行特点
async function test(){
  return 1
}
let res = test()
console.log(res)
/*
输出的结果如下:
Promise {<fulfilled>: 1}
[[Prototype]]: Promise
[[PromiseState]]: "fulfilled"
[[PromiseResult]]: 1

*/
```

根据输出结果会发现，其实 async 修饰的函数，本身就是一个 Promise 对象。虽然在函数中 return 的值是 1，但是在使用了 async 修饰后，test()函数运行时并没有直接返回 1，而是返回了一个值为 1 的 Promise 对象。

接下来，进一步剖析 async 函数的同步和异步特性，代码如下：

```
//第 5 章 5.4.6 进一步剖析 async 函数的同步和异步特性
async function test(){
```

```
  console.log(3)
  return 1
}
console.log(1)
test()
console.log(2)
```

案例输出的结果为 1、3、2。按照 Promise 对象的执行流程，test()函数被 async 修饰后，test()应该变成异步函数，那么应该在 1 和 2 输出完毕后输出 3，但是结果却出人意料，这难道打破了单线程异步模型的概念？答案是并没有。

回想 Promise 对象的结构，代码如下：

```
//第 5 章 5.4.6 回想 Promise 对象的结构
new Promise(function(){

}).then(function(){

})
```

介绍 Promise 对象时，特别介绍了回调函数与同步和异步的关系，并且强调 Promise 是一个极少数的既使用同步回调函数，又使用异步的回调函数的对象，所以在执行语句 new Promise()时，初始化函数是同步函数。

在揭开 async 修饰的函数的神秘面纱前，再参考一个完整的输出顺序案例，代码如下：

```
//第 5 章 5.4.6 一个完整的输出顺序案例
async function test(){
  console.log(3)
  var a = await 4
  console.log(a)
  return 1
}
console.log(1)
test()
console.log(2)
```

该案例的控制台输出顺序为 1、3、2、4。

按照一开始认为的 test()函数为同步函数的逻辑，3 和 4 应该是连续输出的，并不应该出现 3 在 2 之前，4 在 2 之后输出的情况，所以 test()函数单独按照同步逻辑和异步逻辑计算都不符合。

想要真正理解 test()函数的实际执行顺序，需要将当前的函数翻译一下。由于 async 修饰的函数会被解释成 Promise 对象，所以可将案例代码翻译成 Promise 对象结构，代码如下：

```
//第 5 章 5.4.6 将案例代码翻译成 Promise 对象结构
console.log(1)
```

```
new Promise(function(resolve){
  console.log(3)
  resolve(4)
}).then(function(a){
  console.log(a)
})
console.log(2)
```

阅读结果便豁然开朗，由于 Promise 初始化的回调函数是同步的，所以 1、3、2 都是由同步代码输出的，而 4 是在 resolve 中传入的，then()代表异步回调，所以 4 应该最后输出。

综上所述，async 函数的最大特点就是从第 1 个 await 作为分水岭。在第 1 个 await 的右侧和上面的代码，全部为同步代码区域，其相当于 new Promise()的回调函数内部。第 1 个 await 的左侧和下面的代码，则属于异步代码区域，相当于 then()的回调函数内部，所以会出现在同一个函数内，同时出现同步代码和异步代码的现象。

5. setTimeout()案例的最终解决方案

经过了两个时代的变革，可以使用同步化的方式，进行异步流程控制，不再依赖自定义的流程控制器函数，进行分步执行，这一切都是从 Promise 对象的规则定义开始的。

所以综合了多节的学习，setTimeout()案例的最终解决方案，代码如下：

```
//第 5 章 5.4.6 setTimeout()案例的最终解决方案
async function test(){
  var res1 = await new Promise(function(resolve){
    setTimeout(function(){
      resolve('第 1 秒运行')
    },1000)
  })
  console.log(res1)
  var res2 = await new Promise(function(resolve){
    setTimeout(function(){
      resolve('第 2 秒运行')
    },1000)
  })
  console.log(res2)
  var res3 = await new Promise(function(resolve){
    setTimeout(function(){
      resolve('第 3 秒运行')
    },1000)
  })
  console.log(res3)
}
test()
```

从“回调地狱”到 Promise 的链式调用，从 Generator 函数的分步执行到 async 和 await 的自动异步代码同步化，共经历了很多个年头，所以面试中经常会被问到 Promise 对象，并且沿着 Promise 对象深入地挖掘各种问题，主要为考察面试者对 Promise 对象及它的发展

历程是否有深入的了解，也是在考察面试者对 JavaScript 的事件循环系统和异步编程的基本功是掌握扎实。

Promise 和事件循环系统并不是 JavaScript 中的高级知识，而是真正的基础知识，所以所有人想要在行业中更好地发展下去，这些知识都是必备基础知识，必须扎实掌握。

5.5　手撕 Promise 对象

Promise 对象为 ES6 提案中实现的对象。在此提案前，浏览器内部并不存在 Promise 对象，也不支持 Promise 对象的异步控制，所以在不存在 Promise 对象的浏览器中，若运行包含了 Promise 的代码片段，则应如何保证代码能顺利执行？

在不支持 Promise 的浏览器中，存在 setTimeout()这种原始的异步流程控制解决方案，为了保证包含 Promise 对象的新代码能在老旧浏览器中顺利运行，需要程序员在充分了解 Promise 对象特性的前提下，以 setTimeout()为核心，徒手封装一个整体与 Promise 完全一致的伪 Promise 对象。徒手封装 Promise 对象不光是 ECMA 新特性的向下兼容方案，也是开发者在面试中经常遇到的手撕代码场景中的高频出现问题。

5.5.1　定义一个 Promise 对象

1. 分析 Promise 对象的结构

在仿写 Promise 对象前，需要对 Promise 对象本身有详细的了解，所以需要经过以下分析过程。

(1) 查看空 Promise 对象的结构和输出结果，代码如下：

```
//第 5 章 5.5.1 查看空 Promise 对象的结构和输出结果
var p = new Promise(function(resolve,reject){
console.log(resolve,reject)
})
console.log(p)

//输出的结果如下
/*
ƒ() { [native code] } ƒ () { [native code] }
Promise
[[Prototype]]: Promise
[[PromiseState]]: "pending"
[[PromiseResult]]: undefined
*/
```

(2) 查看 fulfilled 状态下的 Promise 对象结构，代码如下：

```
//第 5 章 5.5.1 查看 fulfilled 状态下的 Promise 对象结构
var p = new Promise(function(resolve,reject){
  resolve('已完成')
```

```
})
console.log(p)

//输出的结果如下
/*
Promise {<fulfilled>: '已完成'}
[[Prototype]]: Promise
[[PromiseState]]: "fulfilled"
[[PromiseResult]]: "已完成"
*/
```

(3) 查看 rejected 状态下的 Promise 对象,代码如下:

```
//第 5 章 5.5.1 查看 rejected 状态下的 Promise 对象
var p = new Promise(function(resolve,reject){
  reject('已拒绝')
})
console.log(p)

//输出的结果如下
/*
Promise {<rejected>: '已拒绝'}
[[Prototype]]: Promise
[[PromiseState]]: "rejected"
[[PromiseResult]]: "已拒绝"
Uncaught (in promise) 已拒绝
*/
```

2. Promise 对象的基本结构定义

根据 Promise 对象的特点分析,Promise 存在状态属性和值属性。初始化 Promise 时,需要传入一个回调函数,以便进行对象的基本设置。回调函数具备两个参数 resolve()和 reject(),两个参数均为函数。

综上所述,Promise 对象的初始化结构,代码如下:

```
//第 5 章 5.5.1 Promise 对象的初始化结构
function MyPromise(fn){
  //promise 的初始状态为 pending,可变成 fulfilled 或 rejected 其中之一
  this.promiseState = 'pending'
  this.promiseValue = undefined
  var resolve = function(){

  }
  var reject = function(){

  }
  if(fn){
    fn(resolve,reject)
```

```
  }else{
    throw('Init Error,Please use a function to init MyPromise!')
  }
}
```

根据对象特性，初始化 Promise 时的回调函数是同步执行的，所以此时的 fn()直接调用即可。

在调用 resolve()和 reject()时，需要将 Promise 对象的状态设置为对应的 fulfilled 和 rejected，其中需要传入 Promise 当前的结果，所以应该将 resolve()和 reject()修改为带参数的函数，代码如下：

```
//第 5 章 5.5.1 应该将 resolve()和 reject()修改为带参数的函数
//保存上下文对象
var _this = this
var resolve = function(value){
  if(_this.promiseState == 'pending'){
    _this.promiseState = 'fulfilled'
      _this.promiseValue = value
  }
}
var reject = function(value){
  if(_this.promiseState == 'pending'){
     _this.promiseState = 'rejected'
              _this.promiseValue = value
  }
}
```

定义完内部结构后，需要思考 Promise 在状态变更为 fulfilled 及 rejected 时，对应执行的 then()和 catch()中的回调函数。接下来初始化 Promise 对象的原型方法 then()和 catch()，代码如下：

```
//第 5 章 5.5.1 应该将 resolve()和 reject()修改为带参数的函数
MyPromise.prototype.then = function(callback){

}
MyPromise.prototype.catch = function(callback){

}
```

综上所述，自定义 Promise 对象的初始化结果，代码如下：

```
//第 5 章 5.5.1 自定义 Promise 对象的初始化结果
function MyPromise(fn){
  //promise 的初始状态为 pending,可变成 fulfilled 或 rejected 其中之一
  this.promiseState = 'pending'
  this.promiseValue = undefined
```

```
    var _this = this
    var resolve = function(value){
      if(_this.promiseState == 'pending'){
        _this.promiseState = 'fulfilled'
        _this.promiseValue = value
      }
    }
    var reject = function(value){
       if(_this.promiseState == 'pending'){
         _this.promiseState = 'rejected'
         _this.promiseValue = value
      }
    }
    if(fn){
      fn(resolve,reject)
    }else{
      throw('Init Error,Please use a function to init MyPromise!')
    }
  }
  MyPromise.prototype.then = function(callback){

  }
  MyPromise.prototype.catch = function(callback){

  }
```

5.5.2 实现 then()的回调函数

1. 让 then()的回调函数生效

接下来,使用 MyPromise 按照 Promise 的方式进行编程,实现它的流程控制功能。首先,需要让 then()函数的回调函数运行起来。

在定义 then()的回调函数流程前,先编写 MyPromise 对象的执行案例,代码如下:

```
//第 5 章 5.5.2 编写 MyPromise 对象的执行案例
var p = new MyPromise(function(resolve,reject){
  resolve(123)
})
console.log(p)
p.then(function(res){
  console.log(res)
})
//此时执行代码时控制台会输出以下内容
/*
MyPromise
promiseState: "fulfilled"
promiseValue: 123
[[Prototype]]: Object
*/
```

运行后会发现，自定义的 MyPromise 对象实例 p 的状态已经变更为 fulfilled，但是 then() 中的回调函数没有执行。

接下来，改造 resolve()函数的内容及 then()函数的内容，实现 then()中的回调函数触发功能，代码如下：

```
//第 5 章 5.5.2 实现 then()中的回调函数触发功能
//在 MyPromise 中改造该部分代码如下
//定义 then 的回调函数
this.thenCallback = undefined

var resolve = function(value){
  if(_this.promiseState == 'pending'){
    _this.promiseState = 'fulfilled'
    _this.promiseValue = value
    //异步地执行 then 函数中注册的回调函数
    setTimeout(function(){
      if(_this.thenCallback){
        _this.thenCallback(value)
      }
    })
  }
}

//在 then 中编写如下代码
MyPromise.prototype.then = function(callback){
  //then 第 1 次执行时将回调函数注册到当前的 Promise 对象
  this.thenCallback = function(value){
    callback(value)
  }
}
```

在两处改造完成后，会发现控制台上可以输出 then()函数中的回调执行的结果，并且回调函数参数就是 resolve()传入的值，代码如下：

```
MyPromise {promiseState: 'fulfilled', promiseValue: 123, thenCallback: undefined}
promise.html:51 123
```

至此，MyPromise 对象封装的完整结构，代码如下：

```
//第 5 章 5.5.2 MyPromise 对象封装的完整结构
function MyPromise(fn){
  //promise 的初始状态为 pending，可变成 fulfilled 或 rejected 其中之一
  this.promiseState = 'pending'
  this.promiseValue = undefined
  var _this = this
  //定义 then 的回调函数
  this.thenCallback = undefined
```

```
    var resolve = function(value){
      if(_this.promiseState == 'pending'){
        _this.promiseState = 'fulfilled'
        _this.promiseValue = value
        //异步地执行 then 函数中注册的回调函数
        setTimeout(function(){
          if(_this.thenCallback){
            _this.thenCallback(value)
          }
        })
      }
    }
    var reject = function(value){
      if(_this.promiseState == 'pending'){
        _this.promiseState = 'rejected'
        _this.promiseValue = value
      }
    }
    if(fn){
      fn(resolve,reject)
    }else{
      throw('Init Error,Please use a function to init MyPromise!')
    }
  }
  MyPromise.prototype.then = function(callback){
    //then 第 1 次执行时将回调函数注册到当前的 Promise 对象
    this.thenCallback = function(value){
      callback(value)
    }
  }
  MyPromise.prototype.catch = function(callback){

  }
  var p = new MyPromise(function(resolve,reject){
    resolve(123)
  })
  console.log(p)
  p.then(function(res){
    console.log(res)
  })
```

2. 实现 then()的异步链式调用

通过上文的编程,已经可以实现 then()中回调的自动触发,但是当前案例只能实现一个 then()的回调触发,并且无法链式调用,代码如下:

```
//第 5 章 5.5.2 当前案例只能实现一个 then()的回调触发,并且无法链式调用
var p = new MyPromise(function(resolve,reject){
  resolve(123)
```

```
})
console.log(p)
p.then(function(res){
  console.log(res)
}).then(function(res){
  console.log(res)
}).then(function(res){
  console.log(res)
})

//控制台信息如下
/*
MyPromise {promiseState: 'fulfilled', promiseValue: 123, thenCallback: undefined}
promise.html:52 Uncaught TypeError: Cannot read properties of undefined (reading 'then')
    at promise.html:52
(anonymous) @ promise.html:52
promise.html:51 123
*/
```

针对该情况，需要对 MyPromise 的流程控制代码进行进一步加强，以实现链式调用，并且需要确保，在链式调用的过程中将每次的结果顺利地向下传递。

根据 Promise 对象链式调用的特点，继续改造 resolve()和 then()，代码如下：

```
//第 5 章 5.5.2 根据 Promise 对象链式调用的特点，继续改造 resolve()和 then()
//resolve 部分代码实现
var resolve = function(value){
  if(_this.promiseState == 'pending'){
    _this.promiseValue = value
    _this.promiseState = 'fulfilled'
    //当传入的类型是 Promise 对象时
    if(value instanceof MyPromise){
      value.then(function(res){
        _this.thenCallback(res)
      })
    }else{
      //当传入的数据类型是普通变量时
      setTimeout(function(){
        if(_this.thenCallback){
          _this.thenCallback(value)
        }
      })
    }
  }
}
//then 函数代码实现
MyPromise.prototype.then = function(callback){
  var _this = this
  return new MyPromise(function(resolve,reject){
```

```
      _this.thenCallback = function(value){
        var callbackRes = callback(value)
        resolve(callbackRes)
      }
    })
  }
```

接下来,修改调用代码,向调用代码的 then()回调函数中加入不同的返回值,代码如下:

```
//第 5 章 5.5.2 向调用代码的 then()回调函数中加入不同的返回值
var p = new MyPromise(function(resolve){
  resolve(new MyPromise(function(resolve1){
    resolve1('aaa')
  }))
})
p.then(function(res){
  console.log(res)
  return 123
}).then(function(res){
  console.log(res)
  return new MyPromise(function(resolve){
    setTimeout(function(){
      resolve('Promise')
    },2000)
  })
}).then(function(res){
  console.log(res)
})
console.log(p)
```

运行调用代码会惊喜地发现,MyPromise 对象可以正常工作,并且可以实现 then()的回调函数的延时调用,结果如下:

```
//第 5 章 5.5.2 MyPromise 对象可以正常工作,并且可以实现 then()的回调函数的延时调用
MyPromise {promiseValue: MyPromise, promiseState: 'fulfilled', catchCallback: undefined,
thenCallback: ƒ}
test.html:57 aaa
test.html:60 123
test.html:67 Promise
```

至此,实现了链式调用的 MyPromise 对象的完整结构,代码如下:

```
//第 5 章 5.5.2 实现了链式调用的 MyPromise 对象的完整结构
function MyPromise(fn){
  var _this = this
  this.promiseValue = undefined
  this.promiseState = 'pending'
```

```
  this.thenCallback = undefined
  this.catchCallback = undefined
  var resolve = function(value){
    if(_this.promiseState == 'pending'){
      _this.promiseValue = value
      _this.promiseState = 'fulfilled'
      if(value instanceof MyPromise){

        value.then(function(res){
          _this.thenCallback(res)
        })
      }else{
        setTimeout(function(){
          if(_this.thenCallback){
            _this.thenCallback(value)
          }
        })
      }
    }
  }
  var reject = function(err){

  }
  if(fn){
    fn(resolve,reject)
  }else{
    throw('Init Error,Please use a function to init MyPromise!')
  }
}
MyPromise.prototype.then = function(callback){
  var _this = this
  return new MyPromise(function(resolve,reject){
    _this.thenCallback = function(value){
      var callbackRes = callback(value)
      resolve(callbackRes)
    }
  })
}
var p = new MyPromise(function(resolve){
  resolve(new MyPromise(function(resolve1){
    resolve1('aaa')
  }))
})
```

5.5.3　实现 catch()的完整功能

1. 实现 catch()的捕获功能

当 Promise 的对象触发 reject()函数时，它的状态会变更为 rejected，并且会触发 catch()中

的回调函数。

接下来，仿照 then()的实现方式，在 MyPromise 对象中定义好 reject()函数，代码如下：

```
//第 5 章 5.5.3 在 MyPromise 对象中定义好 reject()函数
//定义 catch 的回调函数
this.catchCallback = undefined
var reject = function(err){
  if(_this.promiseState == 'pending'){
    _this.promiseValue = err
    _this.promiseState = 'rejected'
    setTimeout(function(){
      if(_this.catchCallback){
        _this.catchCallback(err)
      }
    })
  }
}
```

然后，在 catch()函数中加入回调函数的处理，代码如下：

```
//第 5 章 5.5.3 在 catch()函数中加入回调函数的处理
MyPromise.prototype.catch = function(callback){
  var _this = this
  return new MyPromise(function(resolve,reject){
    _this.catchCallback = function(errValue){
      var callbackRes = callback(errValue)
      resolve(callbackRes)
    }
  })
}
```

最后，在案例中加入 catch()功能的调用流程，代码如下：

```
//第 5 章 5.5.3 在案例中加入 catch()功能的调用流程
var p = new MyPromise(function(resolve,reject){
  reject('err')
})
p.catch(function(err){
  console.log(err)
})
```

当运行此代码时，会发现 reject()可以直接触发 catch()的回调执行并输出对应的结果，代码如下：

```
//第 5 章 5.5.3 reject()可以直接触发 catch()的回调执行并输出对应的结果
MyPromise {promiseValue: 'err', promiseState: 'rejected', thenCallback: undefined,
catchCallback: f}
test.html:73 err
```

2. 跨越多个 then()的 catch()捕获

在上文的案例中，已经实现了 MyPromise 的 catch()函数功能，但当 catch()并不是 p 对象直接调用的函数时，catch()中的回调无法执行，代码如下：

```
//第 5 章 5.5.3 当 catch()并不是 p 对象直接调用的函数时,catch()中的回调无法执行
var p = new MyPromise(function(resolve,reject){
  reject(123)
})
console.log(p)
p.then(function(res){
  console.log(res)
}).catch(function(err){
  console.log(err)
})
```

按照已经封装好的功能，当 reject()触发时，MyPromise 对象的状态将自动变更为 rejected，此时 catch()并没有执行，所以 catch()的回调函数无法注册，MyPromise 的流程便断了。这时，需要追加判断代码，让 MyPromise 在 rejected()时，若没有 catchCallback()，则检测是否存在 thenCallback()，代码如下：

```
//第 5 章 5.5.3 若没有 catchCallback(),则检测是否存在 thenCallback()
var reject = function(err){
  if(_this.promiseState == 'pending'){
    _this.promiseValue = err
    _this.promiseState = 'rejected'
    setTimeout(function(){
      if(_this.catchCallback){
        _this.catchCallback(err)
      }else if(_this.thenCallback){
        _this.thenCallback(err)
      }else{
        throw('this Promise was reject,but can not found catch!')
      }
    })
  }
}
```

reject()函数部分改造后，需要将 then()函数中的逻辑更改，以配合新的逻辑，代码如下：

```
//第 5 章 5.5.3 then()函数中的逻辑更改,以配合新的逻辑
MyPromise.prototype.then = function(callback){
  var _this = this
  //实现链式调用并且每个节点的状态是未知的,所以每次都需要返回一个新的 Promise 对象
  return new MyPromise(function(resolve,reject){
    //then 第 1 次执行时将回调函数注册到当前的 Promise 对象
    _this.thenCallback = function(value){
```

```
        //判断如果进入该回调时 Promise 的状态为 rejected 就直接触发后续 Promise 的
        //catchCallback
        //直到找到 catch
        if(_this.promiseState == 'rejected'){
          reject(value)
        }else{
          var callbackRes = callback(value)
          resolve(callbackRes)
        }
      }
    })
  }
```

接下来,更改调用逻辑,在 catch()前加入更多的 then(),代码如下:

```
//第 5 章 5.5.3 在 catch()前加入更多的 then()
var p = new MyPromise(function(resolve,reject){
  reject('err')
})
p.then(function(res){
  console.log(res)
  return 111
}).then(function(res){
  console.log(res)
  return 111
}).then(function(res){
  console.log(res)
  return 111
}).catch(function(err){
  console.log(err)
})
console.log(p)
```

执行调用逻辑的输出结果,代码如下:

```
//第 5 章 5.5.3 执行调用逻辑的输出结果
MyPromise {promiseValue: 'err', promiseState: 'rejected', catchCallback: undefined,
thenCallback: f}
test.html:91 err
```

3. 实现链式调用的中断

本节仅介绍通过返回 Promise 对象来中断链式调用,接下来,在 MyPromise 的原型对象上增加静态 reject()方法,代码如下:

```
//第 5 章 5.5.3 在 MyPromise 的原型对象上增加静态 reject()方法
MyPromise.reject = function(value){
  return new MyPromise(function(resolve,reject){
    reject(value)
  })
}
```

然后，初始化调用代码，代码如下：

```
//第 5 章 5.5.3 初始化调用代码
var p = new MyPromise(function(resolve,reject){
  resolve(123)
})
console.log(p)
p.then(function(res){
  console.log('then1 执行')
  return 456
}).then(function(res){
  console.log('then2 执行')
  return MyPromise.reject('中断了')
}).then(function(res){
  console.log('then3 执行')
  return 789
}).then(function(res){
  console.log('then4 执行')
  return 666
}).catch(function(err){
  console.log('catch 执行')
  console.log(err)
})
```

最后，修改调试代码中的 then()函数逻辑，代码如下：

```
//第 5 章 5.5.3 修改调试代码中的 then()函数逻辑
MyPromise.prototype.then = function(callback){
  var _this = this
  return new MyPromise(function(resolve,reject){
    _this.thenCallback = function(value){
      if(_this.promiseState == 'rejected'){
        reject(value)
      }else{
        var callbackRes = callback(value)
        if(callbackRes instanceof MyPromise){
          if(callbackRes.promiseState == 'rejected'){
            callbackRes.catch(function(errValue){
              reject(errValue)
            })
          }else{
            resolve(callbackRes)
          }
        }else{
          resolve(callbackRes)
        }
      }
    }
  })
}
```

改造后的案例运行结果，代码如下：

```
//第 5 章 5.5.3 改造后的案例运行结果
MyPromise { promiseState: ' fulfilled ' , promiseValue: 123, thenCallback: undefined,
catchCallback: undefined}
promise.html:100 then1 执行
promise.html:103 then2 执行
promise.html:112 catch 执行
promise.html:113 中断了
```

经过改造会发现，在 then()的回调函数中返回 MyPromise.reject()后，then()的链式调用便会中断，并且会触发最近的 catch()的回调函数。

5.5.4 其他常用功能的实现

1. 实现 MyPromise.all()和 MyPromise.race()

根据 Promise.all()的特性，在 MyPromise 对象上创建静态方法 all()，通过 ES5 的语法融入闭包结构，实现 MyPromise.all()，代码如下：

```
//第 5 章 5.5.4 通过 ES5 的语法融入闭包结构，实现 MyPromise.all()
MyPromise.all = function(promiseArr){
  var resArr = []
  var errValue = undefined
  var isRejected = false
  return new MyPromise(function(resolve,reject){
    for(var i = 0;i < promiseArr.length;i++){
      (function(i){
        promiseArr[i].then(function(res){
          resArr[i] = res
          let r = promiseArr.every(item => {
            return item.promiseState == 'fulfilled'
          })
          if(r){
            resolve(resArr)
          }
        }).catch(function(err){
          isRejected = true
          errValue = err
          reject(err)
        })
      })(i)

      if(isRejected){
        break
      }
    }
  })
}
```

MyPromise.race()函数的实现流程与 MyPromise.all()类似，代码如下：

```
//第 5 章 5.5.4 MyPromise.race()函数的实现流程与 MyPromise.all()类似
MyPromise.race = function(promiseArr){
  var end = false
  return new MyPromise(function(resolve,reject){
    for(var i = 0;i < promiseArr.length;i++){
      (function(i){
        promiseArr[i].then(function(res){
          if(end == false){
            end = true
            resolve(res)
          }

        }).catch(function(err){
          if(end == false){
            end = true
            reject(err)
          }
        })
      })(i)
    }
  })
}
```

2．实现基于 Generator 对 MyPromise 对象的异步代码同步化

虽然徒手封装的 MyPromise 对象，并不是通过微任务系统实现的异步流程控制，但并不影响 Generator 函数对齐同步化。在实现 MyPromise 的异步代码同步化前，需要提前准备自动执行 Generator 的工具函数 generatorFunctionRunner()，代码如下：

```
//第 5 章 5.5.4 提前准备自动执行 Generator 的工具函数 generatorFunctionRunner()
/**
    * fn:Generator 函数对象
    */
function generatorFunctionRunner(fn){
  //定义分步对象
  let generator = fn()
  //执行到第 1 个 yield
  let step = generator.next()
  //定义递归函数
  function loop(stepArg,generator){
    //获取本次的 yield 右侧的结果
    let value = stepArg.value
    //判断结果是不是 Promise 对象
    if(value instanceof MyPromise|| value instanceof Promise){
      //如果是 Promise 对象，就在 then 函数的回调中获取本次程序结果
      //并且等待回调执行时进入下一次递归
      value.then(function(promiseValue){
```

```
            if(stepArg.done == false){
              loop(generator.next(promiseValue),generator)
            }
          })
        }else{
          //如果判断程序没有执行完,就将本次的结果进入下一次递归
          if(stepArg.done == false){
            loop(generator.next(stepArg.value),generator)
          }
        }
      }
      //执行动态调用
      loop(step,generator)
    }
```

接下来,编写针对 MyPromise 对象的同步化调用流程,代码如下:

```
//第 5 章 5.5.4 编写针对 MyPromise 对象的同步化调用流程
function * test(){
  let res1 = yield new MyPromise(function(resolve){
    setTimeout(function(){
      resolve('第 1 秒')
    },1000)
  })
  console.log(res1)
  let res2 = yield new MyPromise(function(resolve){
    setTimeout(function(){
      resolve('第 2 秒')
    },1000)
  })
  console.log(res2)
  let res3 = yield new MyPromise(function(resolve){
    setTimeout(function(){
      resolve('第 3 秒')
    },1000)
  })
  console.log(res3)
}
generatorFunctionRunner(test)
```

执行后会发现,MyPromise 对象也可以被 Generator 同步化,这完全归功于 MyPromise 对象所实现的 API 与原生 Promise 对象的 API 一致。

3. MyPromise 的完整源代码

通过简单的代码片段,便可以快速地实现一个微型的 Promise 对象,手写代码封装 Promise 对象,虽然对实际工作没有太大帮助,但是通过分析 Promise 的特性,并以原生 JavaScript 将其实现的过程,代表 JavaScript 异步编程水平近乎大成。最后附上自定义 MyPromise 对象的完整代码片段,代码如下:

```
//第 5 章 5.5.4 自定义 MyPromise 对象的完整代码片段
function MyPromise(fn){
  var _this = this
  this.promiseValue = undefined
  this.promiseState = 'pending'
  this.thenCallback = undefined
  this.catchCallback = undefined
  var resolve = function(value){
    if(_this.promiseState == 'pending'){
      _this.promiseValue = value
      _this.promiseState = 'fulfilled'
      if(value instanceof MyPromise){
        if(_this.thenCallback){
          value.then(function(res){
            _this.thenCallback(res)
          })
        }
      }else{
        setTimeout(function(){
          if(_this.thenCallback){
            _this.thenCallback(value)
          }
        })
      }
    }
  }
  var reject = function(err){
    if(_this.promiseState == 'pending'){
      _this.promiseValue = err
      _this.promiseState = 'rejected'
      setTimeout(function(){
        if(_this.catchCallback){
          _this.catchCallback(err)
        }else if(_this.thenCallback){
          _this.thenCallback(err)
        }else{
          throw('this Promise was reject,but can not found catch!')
        }
      })
    }
  }
  if(fn){
    fn(resolve,reject)
  }else{
    throw('Init Error,Please use a function to init MyPromise!')
  }
}
MyPromise.prototype.then = function(callback){
  var _this = this
```

```
    return new MyPromise(function(resolve,reject){
      _this.thenCallback = function(value){
        if(_this.promiseState == 'rejected'){
          reject(value)
        }else{
          var callbackRes = callback(value)
          if(callbackRes instanceof MyPromise){
            if(callbackRes.promiseState == 'rejected'){
              callbackRes.catch(function(errValue){
                reject(errValue)
              })
            }else{
              resolve(callbackRes)
            }
          }else{
            resolve(callbackRes)
          }

        }

      }
    })
  }
  MyPromise.prototype.catch = function(callback){
    var _this = this
    return new MyPromise(function(resolve,reject){
      _this.catchCallback = function(errValue){
        var callbackRes = callback(errValue)
        resolve(callbackRes)
      }
    })
  }
  MyPromise.reject = function(value){
    return new MyPromise(function(resolve,reject){
      reject(value)
    })
  }
  MyPromise.resolve = function(value){
    return new MyPromise(function(resolve){
      resolve(value)
    })
  }
  MyPromise.all = function(promiseArr){
    var resArr = []
    var errValue = undefined
    var isRejected = false
    return new MyPromise(function(resolve,reject){
      for(var i = 0;i < promiseArr.length;i++){
        (function(i){
```

```
        promiseArr[i].then(function(res){
          resArr[i] = res
          let r = promiseArr.every(item => {
            return item.promiseState == 'fulfilled'
          })
          if(r){
            resolve(resArr)
          }
        }).catch(function(err){
          isRejected = true
          errValue = err
          reject(err)
        })
      })(i)

      if(isRejected){
        break
      }
    }
  })
}

MyPromise.race = function(promiseArr){
  var end = false
  return new MyPromise(function(resolve,reject){
    for(var i = 0;i < promiseArr.length;i++){
      (function(i){
        promiseArr[i].then(function(res){
          if(end == false){
            end = true
            resolve(res)
          }

        }).catch(function(err){
          if(end == false){
            end = true
            reject(err)
          }
        })
      })(i)
    }
  })
}
```