《前端高手进阶》

朱德龙 前中兴软创主任工程师

— 拉勾教育出品 —

第09讲：为什么代码没有按照编写顺序执行？

前言

前端工程师算是最幸运的软件工程师

因为从一开始就可以接触到“异步”这种高级特性

比如 DOM 事件、AJAX 请求及定时器

同时也是最不幸的软件工程师

因为入门 JavaScript 的时候就要习惯异步这种高难度的开发方式

比如上一课时提到的那道经典的笔试题

就是异步造成的输出结果与预期不一致

了解异步

异步和同步

相比**异步**而言，大多数工程师可能更熟悉的是**同步**

要比较同步和异步

可以将调用函数的过程分成两部分：**执行操作和返回结果**

了解异步

拉勾教育
互联网人实战大学

同步调用函数的时候

会立即执行操作并等待得到返回结果后再继续运行

也就是说同步执行是**阻塞**的

异步会将操作和结果在时间上分隔开来

在当下执行操作，在未来某个时刻返回结果

在这个等待返回结果的过程中，程序将继续执行后面的代码

也就是说异步执行是**非阻塞**的

了解异步


```
function syncAdd(a, b){
 return a + b;
}
syncAdd(1, 2) //立即得到结果 3
function asyncAdd(a, b, cb){
 setTimeout(function(){
  cb(a + b);
 }, 1000)
}
asyncAdd(1, 2, console.log) // 1s后打印结果 3
```

了解异步

如果你经常调用 JavaScript 的异步函数可能会形成一个结论：

异步操作都采用回调函数的形式

```
var a = {
  counter: {
    index: 1
  }
};
console.log( a );//?
a.counter.index++;
```

了解异步

既然并非所有异步都回调，是否所有回调函数都是异步执行的呢？

答案也是**否定**的

上一课时中我们就提到过回调形式的同步函数

比如数组原型函数 forEach，又比如改变 this 指向的 call

异步原理

对于大多数语言而言

实现异步会通过启动额外的进程、线程或协程来实现

而我们在前面已经提到过，JavaScript 是**单线程**的

异步原理

为什么单线程还能实现异步呢 ？

把一些操作交给了其他线程处理

然后采用了一种称之为“**事件循环**”（也称“事件轮询”）的机制来处理返回结果

异步原理

```
var eventLoop = []; // 事件队列，先进先出
var event; // 事件执行成功的回调回调函数
while (true) {
  // 一次tick
  if (eventLoop.length > 0) {
    //队列中取出回调函数
    event = eventLoop.shift();
    try {
      event();
    } catch (err) {
      reportError(err);
    }
  }
}
```

异步原理

以 AJAX 请求为例

当我们发出一个 AJAX 请求时

浏览器会将请求任务分派给网络线程来进行处理

当对应的网络线程拿到返回的数据之后

就会把回调函数插入到事件队列中

异步原理

setTimeout 和 **setInterval** 也是同样的道理

当我们执行 setTimeout 的时候并不是直接把回调函数放入事件队列中

它所做的是交给定时器线程来处理

当定时器到时后

再把回调函数放在事件队列中

这样在未来的某轮 tick 中获取并执行这个回调函数

异步原理

这么做有一个**隐性的问题**

如果事件队列中已经有其他事件，那么这个回调就会排队等待

所以说 setTimeout/setInterval 定时器的精度并不高

准确地说，它只能确保回调函数不会在指定的时间间隔之前运行

但可能会在那个时刻运行

也可能在那之后运行

这就要根据事件队列的状态而定

异步原理

事件队列按照先进先出的顺序执行

那么如果队列较长时

排在后面的事件即使较为“紧急”

也得需要**等待**前面的任务先执行完成

设置多个队列，按照优先级来执行

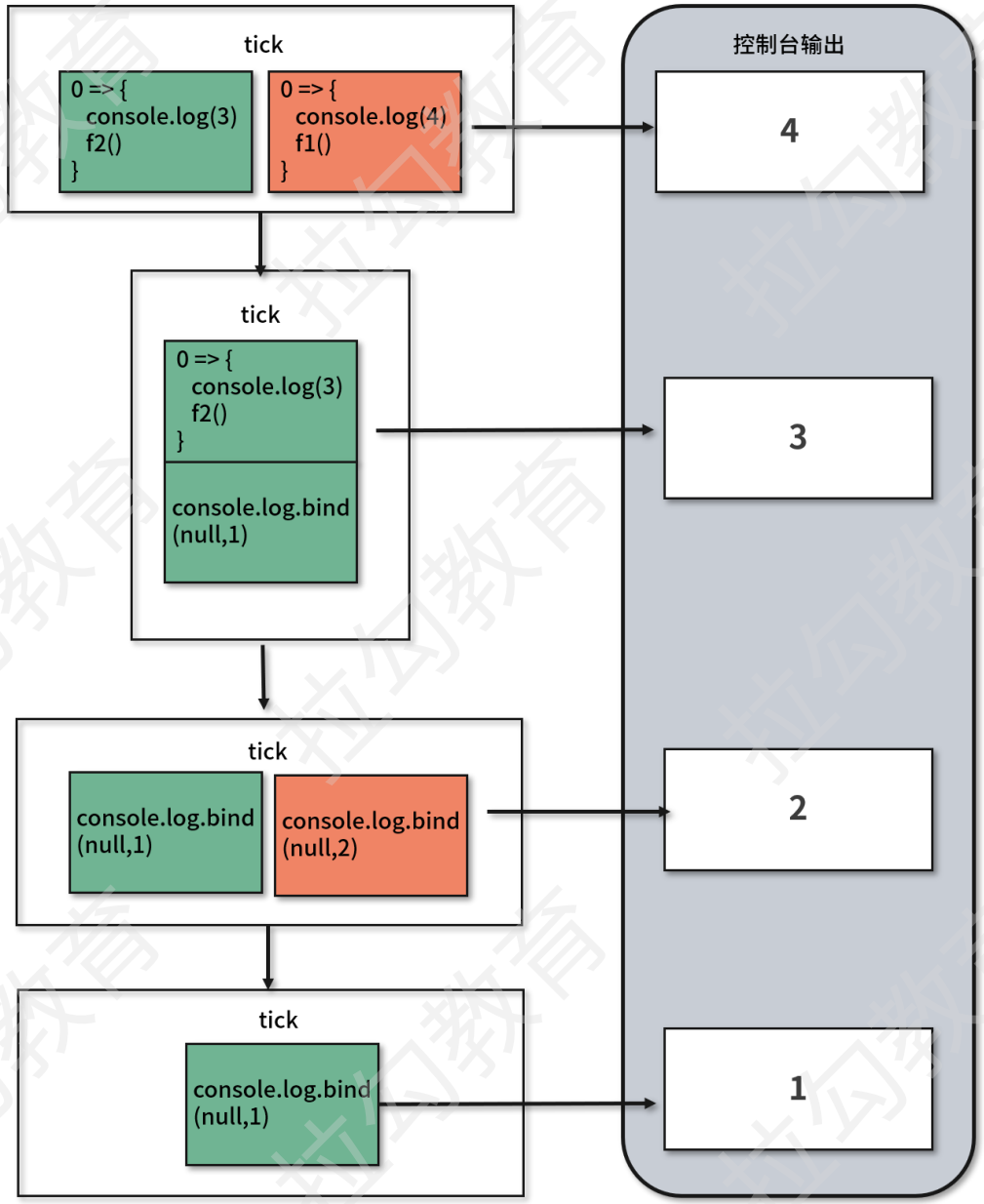
异步原理

```
function f1() {
  setTimeout(console.log.bind(null,1), 0)
}
function f2() {
  Promise.resolve().then(console.log.bind(null,2))
}
function f3() {
  setTimeout(() => {
    console.log(3)
    f2()
  }, 0)
}
```

异步原理

```
    console.log(3)
  f2()
 }, 0)
}
function f4(){
  Promise.resolve().then(()=>{
   console.log(4)
   f1()
  }, 0)
}
f3()
f4()
```

异步原理



异步原理

process.nextTick(Node.js) >
MutationObserver(浏览器)/promise.then(catch?finnally)>
setImmediate(IE) >
setTimeout/setIntervalrequestAnimationFrame >
其他 I/O 操作 / 浏览器 DOM 事件

处理异步

由于回调函数这种形式的代码**可读性非常差**

所以在编写代码的时候要尽量将回调形式**转化**成返回 Promise 对象的形式

一方面由于 ES6 标准下提供了原生 Promise 对象及方法

另一方面 Promise 的可操作性也更强

比如可以配合 async/await 关键字使用

也可以转换成 Observable 对象

所以越来越多的第三方库异步函数都开始返回 Promise 对象

处理异步


```
asyncF1()
.then(data => asyncF2(data))
.then(() => {
  ...
})
.catch(e => console.error(e))
```

处理异步

拉勾教育
互联网人实战大学

```
(async function() {
  try {
    const data = await asyncFn1()
    const result = await asyncFn2(data)
    ...
  } catch(e) {
    console.error(e)
  }
})()
```

处理异步

```
// 代码 1
var o = {
 fn() {
  console.log(this)
 }
}
o.fn() // o
// 代码 2
class A {
 fn() {
  console.log(this)
 }
}
var a = new A()
a.fn()// a
// 代码 3
function fn() {
 console.log(this)
}
fun() // 浏览器：Window；Node.js：global
```

处理异步


```
[1...n].reduce(async (lastPromise, i) => {
  try {
    await lastPromise
    console.log(await asyncF())
  } catch(e) {
    console.error(e)
  }
}, Promise.resolve())
```

处理异步

处理异步

```
const arr = [1, 2, 3, 4, 5]

arr.reduce(async (prs, cur, index) => {
  const t = await prs
  const time = index === 0 ? 0 : 1000 + (index - 1) * 500
  return new Promise((res) => {
    setTimeout(() => {
      console.log(cur);
      res(time)
    }, time)
  })
}, Promise.resolve(0))
```

处理异步

（1）Promise.all([promise1 ..... promiseN])

调用函数 Promise.all 会返回一个新的 Promise 实例

该实例在参数内所有的 **promise 都完成 (resolved) 时回调完成 (resolve)**

如果参数中 promise 有一个失败（rejected）

那么此实例返回第一个失败 promise 的结果

当执行的异步函数具有强一致性时可以使用它

比如要更新一个较大的表单数据

会发送多个请求分别更新不同的数据

如果一个请求更新失败则放弃本次提交

（2）Promise.allSettled([promise1.....promiseN])

调用函数 Promise.allsellted 会返回一个新的 Promise 实例

该实例会在所有给定的 promise **已经执行完成时返回一个对象数组**

每个对象表示对应的 promise 结果

这个函数适用于需要并发执行多个异步函数

这些异步函数的执行结果相互独立

比如同时发送多个 AJAX 请求来分别更新多条数据

（3）Promise.race([promise1.....promiseN])

调用函数 Promise.race 会返回一个新的 promise 实例

一旦参数中的某个 promise 执行完成

新的 promise 实例就会返回对应 promise 的**执行结果**

这个函数会让多个并发函数产生“竞争”

从而挑选出最先执行完成的

比如尝试从多个网址加载图片资源

处理异步

内部的异常**不能**在外部通过 try/catch 所捕获

当内部发生异常时，会**自动**进入失败状态（rejected）

```
new Promise((resolve, reject) => {
  throw new Error(0) // 等价于 reject(new Error(0))
})
```

处理异步


```
Promise.resolve(1)
.then(data => {
  const arr = data.split('')
  //...
}, error => {  // 这里捕获不到
  // ...
})
```

处理异步


```
Promise.resolve(1)
.then(data => {
 const arr = data.split('')
 // ...
})
.catch(error => { // 这里可以捕获
 // ...
})
```

补充：Promise 的局限性

立即执行

当一个 Promise 实例被创建时
内部的代码就会立即被执行，而且无法从外部停止
比如无法取消超时或消耗性能的异步调用
容易导致资源的浪费

补充：Promise 的局限性

单次执行

Promise 处理的问题都是“一次性”的

因为一个 Promise 实例只能 resolve 或 reject 一次

所以面对某些需要持续响应的场景时就会变得力不从心

比如上传文件获取进度时

默认采用的就是通过事件监听的方式来实现

总结

涉及了 JavaScript 的核心特性——**异步**

先从异步概念说起

然后深入异步原理讲述了事件循环和事件队列

最后列举了 3 个常见**异步场景的处理方法**

总结

尝试使用 RxJS 实现多个 Promise 的串行和并行

并说说它在处理异步方面的优缺点 ？

Next：第10讲《怎么复用你的代码？》

拉勾教育

—互联网人实战大学—

下载「拉勾教育App」
获取更多内容

L / A / G / O / U