JavaScript：async/await的基础用法

96 郭靖888 (/u/5400215b7272) ＋关注
(/u/5400215b7272) 2018.03* 字数 2753 阅读 1157 评论 2 喜欢 8

相对于回调函数来说， Promise 是一种相对优雅的选择。那么有没有更好的方案呢？答案就是 async/await。
优势主要体现在，级联调用，也就是几个调用依次发生的场景。
async/await。被称为到目前最优雅的异步过程解决方案，不知道你是否认同，反正我是信了。

相对于 Promise，async/await 有什么优点?

比较场景： 级联调用，也就是几个调用依次发生的场景

- Promise 主要用then函数的链式调用，一直点点点，是一种从左向右的横向写法。
  async/await 从上到下，顺序执行，就像写同步代码一样。这更符合人编写代码的习惯

- Promise 的then函数只能传递一个参数，虽然可以通过包装成对象，但是这会导致传递冗余信息，频繁的解析又重新组合参数，比较麻烦。
  async/await 没有这个限制，就当做普通的局部变量来处理好了，用let或者const定义的块级变量，想怎么用就怎么用，想定义几个就定义几个，完全没有限制，也没有冗余的工作。

- Promise 在使用的时候最好将同步代码和异步代码放在不同的then节点中，这样结构更加清晰。
  async/await 整个书写习惯都是同步的，不需要纠结同步和异步的区别。当然，异步过程需要包装成一个 Promise 对象，放在 await 关键字后面，这点还是要牢记的。

- Promise 是根据函数式编程的范式，对异步过程进行了一层封装。
  async/await 是基于协程的机制，是真正的“保存上下文，控制权切换 ... ... 控制权恢复，取回上下文”这种机制，是对异步过程更精确的一种描述。

> 进程、线程和协程的理解 (https://link.jianshu.com?t=http%3A%2F%2Fblog.csdn.net%2Fhairetz%2Farticle%2Fdetails%2F16119911)
> 上面的文章很好地解释了这几个概念的区别。
> 如果不纠结细节，可以简单地认为：进程 > 线程 > 协程；
> 协程可以独立完成一些与界面无关的工作，不会阻塞主线程渲染界面，也就是不会卡。
> 协程，虽然小一点，不过能完成我们程序员交给的任务。而且我们可以自由控制运行和阻塞状态，不需要求助于高大上的系统调度，这才是重点。

- async/await 是基于 Promise 的，是进一步的一种优化。不过再写代码的时候，Promise 本身的API出现得很少，很接近同步代码的写法。

await 关键字使用时有哪些注意点?

- 只能放在 async 函数内部使用，不能放在普通函数里面，否则会报错。

- 后面放 Promise 对象，在 Pending 状态时，相应的协程会交出控制权，进入等待状态。这个是本质。

- await 是 async wait 的意思， wait 的是 resolve(data) 消息，并把数据 data 返回。比如，下面代码中，当 Promise 对象由 Pending 变为 Resolved 的时候，变量 a 就等于 data ； 然后再顺序执行下面的语句 console.log(a);
  这真的是等待，真的是顺序执行，表现和同步代码几乎一模一样。

```
const a = await new Promise((resolve, reject) => {
    // async process ...
    return resolve(data);
});
console.log(a);
```

- await 后面也可以跟同步代码，不过系统会自动转化成一个 Promise 对象。
  比如
  `const a = await 'hello world';`
  其实就相当于
  `const a = await Promise.resolve('hello world');`
  这跟同步代码
  `const a = 'hello world';` 是一样的，还不如省点事，去掉这里的 await 关键字。

- await 只关心异步过程成功的消息 resolve(data) ，拿到相应的数据 data 。至于失败消息 reject(error) ，不关心，不处理。
  当然对于错误消息的处理，有以下几种方法供选择：
  （1）让 await 后面的 Promise 对象自己 catch
  （2）也可以让外面的 async 函数返回的 Promise 对象统一 catch
  （3）像同步代码一样，放在一个 try...catch 结构中

async 关键字使用时有哪些注意点?

- 有了这个 async 关键字，只是表明里面可能有异步过程，里面可以有 await 关键字。当然，全部是同步代码也没关系。当然，这时候这个 async 关键字就显得多余了。不是不能加，而是不应该加。

- async 函数，如果里面有异步过程，会等待；
  但是 async 函数本身会马上返回，不会阻塞当前线程。

> 可以简单认为， async 函数工作在主线程，同步执行，不会阻塞界面渲染。
> async 函数内部由 async 关键字修饰的异步过程，工作在相应的协程上，会阻塞等待异步任务的完成再返回。

- async 函数的返回值是一个 Promise 对象，这个是和普通函数本质不同的地方。这也是使用时重点注意的地方
  （1） return newPromise(); 这个符合 async 函数本意；
  （2） return data; 这个是同步函数的写法，这里是要特别注意的。这个时候，其实就相当于 Promise.resolve(data); 还是一个 Promise 对象。
  在调用 async 函数的地方通过简单的 = 是拿不到这个 data 的。
  那么怎么样拿到这个 data 呢?
  很简单，返回值是一个 Promise 对象，用 .then(data => { }) 函数就可以。
  （3）如果没有返回，相当于返回了 Promise.resolve(undefined);

- await 是不管异步过程的 reject(error) 消息的， async 函数返回的这个 Promise 对象的 catch 函数就负责统一抓取内部所有异步过程的错误。
  async 函数内部只要有一个异步过程发生错误，整个执行过程就中断，这个返回的 Promise 对象的 catch 就能抓到这个错误。

- async 函数执行和普通函数一样，函数名带个()就可以了，参数个数随意，没有限制；也需要有 async 关键字。
  只是返回值是一个 Promise 对象，可以用then函数得到返回值，用catch抓去整个流程中发生的错误。

基本套路

Step1：用 Promise 对象包装异步过程，这个和 Promise 的使用一样。只是参数个数随意，没有限制。

```
function sleep(ms) {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve('sleep for ' + ms + ' ms');
        }, ms);
    });
}
```

Step2：定义异步流程，可以将按照需要定制，就像写同步代码那样

```
async function asyncFunction() {
    console.time('asyncFunction total executing:');
    const sleep1 = await sleep(2000);
    console.log('sleep1: ' + sleep1);
    const [sleep2, sleep3, sleep4]= await Promise.all([sleep(2000), sleep(1000), sle
    console.log('sleep2: ' + sleep2);
    console.log('sleep3: ' + sleep3);
    console.log('sleep4: ' + sleep4);
    const sleepRace = await Promise.race([sleep(3000), sleep(1000), sleep(1000)]);
    console.log('sleep race: ' + sleepRace);
    console.timeEnd('asyncFunction total executing:');

    return 'asyncFunction done.'  // 这个可以不返回，这里只是做个标记，为了显示流程
}
```

Step3：像普通函数调用 async 函数，在 then 函数中获取整个流程的返回信息，在 catch 函数统一处理出错信息

```
asyncFunction().then(data => {
    console.log(data);       // asyncFunction return 的内容在这里获取
}).catch(error => {
    console.log(error);      // asyncFunction 的错误统一在这里抓取
});

console.log('after asyncFunction code executing....'); // 这个代表asyncFunction函数后的
                                                       // 显示asyncFunction本身会立即返
```

流程解析

上面的代码执行之后，输出的 log 如下，显示了代码执行流程

```
after asyncFunction code executing....
sleep1: sleep for 2000 ms
sleep2: sleep for 2000 ms
sleep3: sleep for 1000 ms
sleep4: sleep for 1500 ms
sleep race: sleep for 1000 ms
asyncFunction total executing:: 5006.276123046875ms
asyncFunction done.
```

1. `after asyncFunction code executing....` 代码位置在 `async` 函数 `asyncFunction()` 调用之后，反而先输出。这说明 `async` 函数 `asyncFunction()` 调用之后会马上返回，不会阻塞主线程。

2. `sleep1: sleep for 2000 ms` 这是第一个 `await` 之后的第一个异步过程，最先执行，也最先完成，说明后面的代码，不论是同步和异步，都在等他执行完毕。

3. `sleep2 ~ sleep4` 这是第二个 `await` 之后的 `Promise.all()` 异步过程。这是“比慢模式”，三个 `sleep` 都完成后，再运行下面的代码，耗时最长的是 `2000ms`；

4. `sleep race: sleep for 1000 ms` 这是第三个 `await` 之后的 `Promise.race()` 异步过程。这是“比快模式”，耗时最短 `sleep` 都完成后，就运行下面的代码。耗时最短的是 `1000ms`；

5. `asyncFunction total executing:: 5006.276123046875ms` 这是最后的统计总共运行时间代码。三个await之后的异步过程之和
   `1000（独立的） + 2000（Promise.all） + 1000（Promise.race） = 5000ms`
   这个和统计出来的5006.276123046875ms非常接近。说明上面的异步过程，和同步代码执行过程一致，协程真的是在等待异步过程执行完毕。

6. `asyncFunction done.` 这个是 `async` 函数返回的信息，在执行时的then函数中获得，说明整个流程完毕之后参数传递的过程。

log.png

异常处理

- `async` 标注过的函数，返回一个 `Promise` 对象，采用 `.then().catch()` 的方式来进行异常处理，是非常自然的方法，也推荐这么做。就像上面的 `step3` 那样做。

- 另外一种方法，就是对于异步过程采用 `await` 关键字，采用同步的 `try{} catch(){}` 的方式来进行异常处理。

- 这里要注意的是 `await` 关键字只能用在async标注的函数中，所以，原来的函数，不管以前是同步的还是异步的，都要加上 `async` 关键字，比如 `componentDidMount()` 就要变为 `async componentDidMount()` 才可以在内部使用 `await` 关键字，不过功能上没有任何影响。

- 另外，采用同步的 `try{} catch(){}` 的方式，可以把同步，异步代码都可以放在里面，有错误都能抓到，比如 `null.length` 这种，也能抓到。

```
async componentDidMount() { // 这是React Native的回调函数，加个async关键字，没有任何影响，
    // 将异步和同步的代码放在一个try..catch中，异常都能抓到
    try {
        let array = null;
        let data = await asyncFunction();  // 这里用await关键字，就能拿到结果值；否则，没有
        if (array.length > 0) {  // 这里会抛出异常，下面的catch也能抓到
            array.push(data);
        }
    } catch (error) {
        alert(JSON.stringify(error))
    }
}
```

> 这里模拟的是网络过程。一般情况， `array` 是一个数组，用if `(array.length > 0)` 判断一下长度，有值再处理，没有问题。但是，一旦网络出问题， `array` 就是一个 `null` ，平时工作很好的 `if (array.length > 0)` 判断就会抛异常， `JS` 代码就中断，停止工作，会带来意想不到的问题。
> 这里加了一个 `try..catch` 结构，这种异常就能捕获，（这是同步代码中的异常，不能用 `.then().catch()` 抓到）,根据异常信息，一般是 `null` 没有 `length` 属性，方便定位问题。这里的话用 `if (array && (array.length > 0))` 就会安全一点。

参考文章

本文只是介绍了 `async/await` 一种基础的用法。一个例子，将三种 `Promise` 使用中常用的场景模式都包括进去了，并且代码风格和同步代码非常相似。相比之下 `async/await` 这套异步代码编程方式确实比较优雅。

下面几篇文章都非常不错，介绍了 `async/await` 很多灵活的用法，建议好好看看。

理解 JavaScript 的 async/await (https://link.jianshu.com?t=https%3A%2F%2Fsegmentfault.com%2Fa%2F1190000007535316)

深入理解ES7的async/await (https://link.jianshu.com?t=http%3A%2F%2Fblog.csdn.net%2Fsinat_17775997%2Farticle%2Fdetails%2F60609498)

async 函数 (https://link.jianshu.com?t=http%3A%2F%2Fes6.ruanyifeng.com%2F%23docs%2Fasync)