Leon 1.1k

理解async/await

javascript node.js 异步编程 阅读约 16 分钟

首先明确一个问题，为什么 Node.js 需要异步编程?

JavaScript 是单线程的，在发出一个调用时，在没有得到结果之前，该调用就不返回，意思就是调用者主动等待调用结果，换句话说，就是必须等待上一个任务执行完才能执行下一个任务，这种执行模式叫：**同步**。
Node.js 的主要应用场景是处理高并发（单位时间内极大的访问量）和 I/O 密集场景（ps: I/O 操作往往非常耗时，所以异步的关键在于解决 I/O 耗时问题），如果采用同步编程，问题就来了，服务器处理一个 I/O 请求需要大量的时间，后面的请求都将排队，造成浏览器端的卡顿。异步编程能解决这个问题。
所谓**异步**，就是调用在发出后，这个调用就直接返回了，调用者不会立即得到结果，但是不会阻塞，可以继续执行后续操作，而被调用者执行得到结果后通过状态、事件来通知调用者使用回调函数（ callback ）来处理这个结果。Node在处理耗时的 I/O 操作时，将其交给其他线程处理，自己继续处理其他访问请求，当 I/O 操作处理好后就会通过事件通知 Node 用回调做后续处理。
有个例子非常好：

> 你打电话问书店老板有没有《分布式系统》这本书，如果是同步通信机制，书店老板会说，你稍等，"我查一下"，然后开始查啊查，等查好了（可能是5秒，也可能是一天）告诉你结果（返回结果）。而异步通信机制，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回结果）。然后查好了，他会主动打电话给你。在这里老板通过"回电"这种方式来回调。

下面几种方式是异步解决方案的进化过程：

CallBacks

回调函数就是函数A作为参数传递给函数B，并且在未来某一个时间被调用。callback的异步模式最大的问题就是，理解困难加回调地狱（callback hell），看下面的代码的执行顺序：

```
A();
ajax('url1', function(){
    B();
    ajax('url2', function(){
        C();
    }
    D();
});
E();
```

其执行顺序为：A => E => B => D => C，这种执行顺序的确会让人头脑发昏，另外由于由于多个异步操作之间往往会耦合，只要中间一个操作需要修改，那么它的上层回调函数和下层回调函数都可能要修改，这就陷入了回调地狱。而 Promise 对象就很好的解决了异步操作之间的耦合问题，让我们可以用同步编程的方式去写异步操作。

Promise

Promise 对象是一个构造函数，用来生成promise实例。Promise 代表一个异步操作，有三种状态：pending，resolved（异步操作成功由 pending 变为 resolved ），rejected（异步操作失败由 pending 变为 rejected ），一旦变为后两种状态将不会再改变。Promise 对象作为构造函数接受一个函数作为参数，而这个函数又接受 resolve 和 reject 两个函数做为参数，这两个函数是JS内置的，无需配置。resolve 函数在异步操作成功后调用，将pending状态变为resolved，并将它的参数传递给回调函数；reject 函数在异步操作失败时调用，将pending状态变为rejected，并将参数传递给回调函数。

- **Promise.prototype.then()**

Promise构造函数的原型上有一个then方法，它接受两个函数作为参数，分别是 resolved 状态和 rejected 状态的回调函数，而这两个回调函数接受的参数分别是Promise实例中resolve函数和reject**函数中的参数。** 另外rejected状态的回调函数是可省略的。

下面是一个使用示例：

```
const instance = new Promise((resolve, reject) => {
    // 一些异步操作
    if(/*异步操作成功*/) {
      resolve(value);
    } else {
      reject(error);
    }
  }
})
instance.then(value => {
  // do something...
}, error => {
  // do something...
})
```

注意Promise实例在生成后会立即执行，而 then 方法只有在所有同步任务执行完后才会执行，看看下面的例子：

```
const promise = new Promise((resolve, reject) => {
  console.log('async task begins!');
  setTimeout(() => {
    resolve('done, pending -> resolved!');
  }, 1000);
})

promise.then(value => {
  console.log(value);
})

console.log('1.please wait');
console.log('2.please wait');
console.log('3.please wait');
// async task begins!
// 1.please wait
// 2.please wait
// 3.please wait
// done, pending -> resolved!
```

上面的实例可以看出，Promise实例生成后立即执行，所以首先输出 'async task begins!'，随后定义了一个异步操作 setTimeout，1秒后执行，所以无需等待，向下执行，而then方法指定的回调函数要在所有同步任务执行完后才执行，所以先输出了3个'please wait'，最后输出'done, pending -> resolved!'。(此处省略了then方法中的reject回调，一般不在then中做rejected状态的处理，而使用catch方法专门处理错误，相当于.then(null, reject))

- **链式调用 then 方法**

  then 方法会返回一个新的 Promise 实例，可以分两种情况来看：

1. 指定返回值是新的 Promise 对象，如return new Promise(...)，这种情况没啥好说的，由于返回的是 Promise，后面显然可以继续调用then方法。
2. 返回值不是Promise， 如：return 1 这种情况还是会返回一个 Promise，并且这个Promise 立即执行回调 resolve(1)。所以仍然可以链式调用then方法。（注：如果没有指定return语句，相当于返回了undefined）

使用 then 的链式写法，按顺序实现一系列的异步操作，这样就可以用同步编程的形式去实现异步操作，来看下面的例子，实现隔两秒打一次招呼：

```
function sayHi(name) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(name);
    }, 2000)
  })
}

sayHi('张三')
  .then(name => {
    console.log(`你好， ${name}`);
    return sayHi('李四');    // 最终 resolved 函数中的参数将作为值传递给下一个then
  })
  // name 是上一个then传递出来的参数
  .then(name => {
    console.log(`你好， ${name}`);
    return sayHi('王二麻子');
  })
  .then(name => {
    console.log(`你好， ${name}`);
  })
// 你好， 张三
// 你好， 李四
// 你好， 王二麻子
```

可以看到使用链式then的写法，将异步操作变成了同步的形式，但是也带来了新的问题，就是异步操作变成了很长的then链，新的解决方法就是Generator，这里跨过它直接说它的语法糖：async/await。

async/await

- **async**

async/await实际上是Generator的语法糖。顾名思义，async关键字代表后面的函数中有异步操作，await表示等待一个异步方法执行完成。声明异步函数只需在普通函数前面加一个关键字async即可，如：

```
async function funcA() {}
```

async 函数返回一个Promise对象（如果指定的返回值不是Promise对象，也返回一个Promise，只不过立即 resolve ，处理方式同 then 方法），因此 async 函数通过 return 返回的值，会成为 then 方法中回调函数的参数：

```
async function funcA() {
  return 'hello!';
}

funcA().then(value => {
  console.log(value);
})
// hello!
```

单独一个 async 函数，其实与Promise执行的功能是一样的，来看看 await 都干了些啥。

- **await**

顾名思义， await 就是异步等待，它等待的是一个Promise，因此 await 后面应该写一个Promise对象，如果不是Promise对象，那么会被转成一个立即 resolve 的Promise。 async 函数被调用后就立即执行，但是一旦遇到 await 就会先返回，等到异步操作执行完成，再接着执行函数体内后面的语句。总结一下就是：async函数调用不会造成代码的阻塞，但是await会引起async函数内部代码的阻塞。看看下面这个例子：

```
async function func() {
  console.log('async function is running!');
  const num1 = await 200;
  console.log(`num1 is ${num1}`);
  const num2 = await num1+ 100;
  console.log(`num2 is ${num2}`);
  const num3 = await num2 + 100;
  console.log(`num3 is ${num3}`);
}

func();
console.log('run me before await!');
// async function is running!
// run me before await!
// num1 is 200
// num2 is 300
// num3 is 400
```

可以看出调用 `async func` 函数后，它会立即执行，首先输出了`'async function is running!'`，接着遇到了 `await` 异步等待，函数返回，先执行`func()`后面的同步任务，同步任务执行完后，接着await等待的位置继续往下执行。可以说，`async`函数完全可以看作多个异步操作，包装成的一个Promise 对象，而`await`命令就是内部`then`命令的语法糖。

值得注意的是， `await` 后面的 Promise 对象不总是返回 `resolved` 状态，只要一个 `await` 后面的Promise状态变为 `rejected` ，整个 `async` 函数都会中断执行，为了保存错误的位置和错误信息，我们需要用 `try...catch` 语句来封装多个 `await` 过程，如下：

```
async function func() {
  try {
    const num1 = await 200;
    console.log(`num1 is ${num1}`);
    const num2 = await Promise.reject('num2 is wrong!');
    console.log(`num2 is ${num2}`);
    const num3 = await num2 + 100;
    console.log(`num3 is ${num3}`);
  } catch (error) {
    console.log(error);
  }
}

func();
// num1 is 200
// 出错了
// num2 is wrong!
```

如上所示，在 `num2` 处 `await` 得到了一个状态为 `rejected` 的Promise对象，该错误会被传递到 `catch` 语句中，这样我们就可以定位错误发生的位置。

- **async/await比Promise强在哪儿?**

接下来我们用`async/await`改写一下Promise章节中关于`sayHi`的一个例子，代码如下：