用-Future-处理异步逻辑-傻瓜版

用 Future 处理异步逻辑（傻瓜版）

很多编程语言（JavaScript, Rust, C#）都将 async/await 作为语言内置的异步编程接口，虽然实现不同，对外表现也都大同小异。这里以 js 为例说明 async/await 的使用动机、使用方式。

> *为什么 JavaScript 没有 sleep 这样的函数?*

> *因为 JavaScript 最初仅被用于浏览器操作 DOM，是单线程的。首先单线程程序没有 sleep 的必要，其次，设想某个程序组件调用了 sleep，将导致整个浏览器卡死在这里，是不是很恐怖?*

这篇博客最初是用来在实验室小组内10分钟为同事们普及 JavaScript Promise 编程概念而写的，所以没有涉及任何有难度的 PL 知识和实现层面的细节。

Promise 动机

阻塞无处不在。假设我们需要请求一个网页的内容，并对这个内容进行打印输出，一个普通的同步 API 是这个样子的：

```
let resp = fetch('www.some.resource');
let data = resp.json();
console.log(data);
// do other stuff
```

但是网络请求是一个 IO 操作，很容易就发生阻塞。JavaScript 是单线程的，我们没有办法开一个线程来单独处理这部分逻辑，所以我们经常看见在 JavaScript 中，这种可能阻塞的 API 都被设计成这样：

```
fetch('www.some.resource', (resp) => {
  let data = resp.json();
  console.log(data);
});

// do other stuff
```

这样，我们不需要考虑这个网页的内容什么时候真正被请求到，我们通过传递一个回调函数来表达这样的逻辑：当这个内容被请求到后，它将作为参数执行以下逻辑。这样，fetch 操作不会阻塞当前线程，却完成了以往多线程才能完成的任务。

> *fetch 的第二个参数叫 continuation，意思是某个操作的后续操作。continuation 一般都以单参数的函数表达，参数就是表达式的求值结果。*

这样的代码有一个缺点，就是可能造成回调地狱，例如我希望在请求到网页结果后，根据结果内容再发出新的请求，代码可能变成这个样子：

```
fetch('someURL', (resp) => {
  let data = resp.json();
  fetch(data['img'], (resp) => {
    let data = resp.json();
    fetch(data['addr'], (resp) => {
      let data = resp.json();
      console.log(data);
    });
  });
});
```

这样的代码，想要换一下执行顺序，修改起来太麻烦了，读起来也很难受，错误处理也很困难。这也是为什么后来的 API 都尽量设计成 Promise 返回值：

```
let respPromise = fetch('someURL');
respPromise.then((resp) => {
  let data = resp.json();
  console.log(data);
});
```

Promise 对象的 `then` 方法允许我们定义 promise 的后续操作。`then` 的返回值也是一个 Promise，即包裹了回调函数的返回值的 Promise。所以前面的回调地狱就可以改写成如下形式：

```
fetch('someURL')
  .then((resp) => fetch(resp.json()['img']))
  .then((resp) => fetch(resp.json()['addr']))
  .then(console.log);
```

这样嵌套的函数就变成了顺序的，代码简洁多了。

await

JavaScript 提供了更加简单的操作 Promise 对象的语法糖。在上述例子中，使用 `await` 关键字可以让整个程序成为近乎命令式。例如，对于一个包含了字符串的 Promise，`await` 操作符能返回一个字符串：

```
let stringPromise = Promise.resolve('some string');
let s = await stringPromise;
console.log(s);
```

这段代码等价于：

```
Promise.resolve('some string')
  .then((s) ⇒ console.log(s));
// or more succinct
Promise.resolve('some string').then(console.log);
```

使用 `await` 来重写上述回调地狱：

```
let resp1 = await fetch('someURL');
let imgURL = resp.json()['img'];
let resp2 = await fetch(imgURL);
let dataURL = img.json()['addr'];
let resp3 = await fetch(dataURL);
let data = resp3.json();
console.log(data);
```

直觉上，`await` 操作就像阻塞，直到后续异步操作返回结果。但事实上，`await` 只是语法糖，这段代码和上述使用 `then` 的代码是完全等价的。

> *`await` 操作符后的表达式如果不是 Promise，就返回表达式的值本身。*

async

以上包含 `await` 的代码是不能直接执行的。在语言规范上，`await` 只能使用在 `async` 函数内（top-level await 因为诸多问题被单独标准化，而且引起了很激烈的后续讨论：Top-level await is a footgun）。上述代码要被包裹在一个函数内部：

```
async function main() {
  let resp1 = await fetch('someURL');
  let imgURL = resp.json()['img'];
  let resp2 = await fetch(imgURL);
  let dataURL = img.json()['addr'];
  let resp3 = await fetch(dataURL);
  let data = resp3.json();
  return data;
}
```

为了凸显 `async` 的某些特点，我们去掉了打印操作，而是将结果字符串返回。async 函数和普通的 JavaScript 函数没什么不同，可以当作普通函数直接使用。但是，有两点需要注意：

1. `await` 只能在 async 函数中使用
2. async 函数的返回值是 Promise，如果不是 Promise 会被自动包裹到 Promise
3. （其实 async 函数的类型和普通函数也有区别，并且不能使用 `new` 来创建对象。这已经不重要了，现在很少有人会用函数来做构造函数）

第二点很好理解，因为 async 函数是异步逻辑，返回值本来就应当是 Promise。但为什么 `await` 被限制在 async 函数中呢？因为“等待异步操作”的函数必须也是异步的。这也是异步函数的一个潜在问题：传染性，一旦一个地方用了异步操作，再想变回同步就不可能了。

> *async 仅仅只是一个给程序员看的标记，表示这是个异步函数，它可以等待其它异步函数的执行结果，让程序员清楚地知道这个函数在执行时需要让出线程，不能被当作同步函数使用。理论上来说 async 没有增强语言的表达能力。*

Promise 状态

Promise 对象有三个状态：Pending, Fulfilled, Rejected.

一个 Promise 被创建时是 pending 状态；执行成功后变成 fulfilled 状态，触发 `resolve` 回调；执行失败后变成 rejected 状态，触发 `reject` 回调。例子：

```
fetch('someURL').
  .then(resp ⇒ console.log('Success'))
  .catch(err ⇒ console.log('Failed'));
```

当 `fetch` 操作失败后，就会进入 rejected 状态并打印 `Failed`。

Promise.all

考虑这样一个场景：`urls` 是一个包含了10个 URL 的数组，我们想要对每一个 URL 都执行 fetch 操作，但又不希望它们相互等待。如果 fetch 一个资源的时间是0.1s，那么同时 fetch 10个资源的时间应当也是0.1s而不是1s。我们可以这样编写程序：

```
let results = await Promise.all(urls.map(fetch));
```

`Promise.all` 等待数组内所有 Promise 都变成 fulfilled 状态，就变成 fulfilled 状态，其值即为原数组内 Promise 对应的值构成的数组。如果其中任何一个 rejected，则该 Promise 也进入 rejected 状态。

`Promise.race` 也等待数组，行为是对偶的。只要数组内有任何一个元素 fulfiiled，就变成 fulfilled 状态并返回该元素的值；如果所有元素都 rejected，则进入 rejected.

await 解决 point free 的弊端

命令式编程的一个好处是几乎拥有一个无限平坦的 environment（变量名-值映射）。我们可以编写这样的程序：

```
async main() {
  let x = await f();
  let y = await g(x);
  let z = await h(x, y);
  return z;
}
```

如果没有 `await`，程序该怎么写?

```
function main() {
  f()
    .then(g)
    .<?>
 }
```

写到 `?` 处就卡住了，因为此时 `x` 的值已经丢失，只剩下 `g(x)`。要想把 `x` 的值保留下来只能利用数组和 `Promise.all`：

```
function main() {
  return f()
    .then(x ⇒ Promise.all([x, g(x)]))
    .then(arr ⇒ h(arr[0], arr[1]));
}
```

和 `await` 版本比起来非常地不方便。

模拟一个 sleep 函数

有了以上能力，写一个 sleep 函数就容易得多了。JavaScript 的运行时支持我们将任务塞到事件循环的队列，于是我们可以利用 `setTimeout` 来做定时任务：

```
setTimeout(function() {
  console.log("Hi");
}, 1000);
```

这行代码执行的1s后将会打印 `Hi` 字样。依据此功能，我们可以编写如下函数来模拟 `sleep` 的行为了：

```
async function sleep(time) {
  return new Promise((resolve) => setTimeout(resolve, time));
}

await sleep(1000);
console.log("Hi");
```

Promise monad

很容易看出 Promise 是一个 monad。其 `then` 操作正是 `map` 和 `flatMap`（在不同语言中也表示为 `bind` 或是 `>>=` 等名字）的合体。此处不做严格证明，注意到显然 `Promise.resolve` 就是 `pure`（也表示为 `return`）和 `then` 的类型就可以了。

> *什么是 monad?*

> *monad（单子）即自函子范畴上的幺半群（雾）。*

考虑到 `resolve` 回调的返回值如果不是 Promise 则会自动转为 Promise，所以分两种情况讨论。

当 resolve 回调的返回值不是 Promise：

```
then :
  {this : Promise<A>} →
  (resolve: A → B) →
  Promise<B>
```

和 `map` 的类型一致。

当 resolve 回调的返回值是 Promise：

```
then :
  {this : Promise<A>} →
  (resolve: A → Promise<B>) →
  Promise<B>
```

和 `flatMap` 的类型一致。

这个视角下，`await` 即是 do notation 中的 `←` 语法糖。

动脑筋

最后来一个思考题，需要结合 Continuation 相关的知识。请问以下代码的输出是什么?

```
function conditionalPass(n: number) {
  return new Promise<void>((resolve, _reject) => {
    if (n < 4 || n > 8) {
      resolve();
    }
  });
}

async function main() {
  for (let i = 0; i < 10; i++) {
    await conditionalPass(i);
    console.log(i);
  }
}

main();
```

答案？自己在电脑上跑一遍就知道了。

参考

https://tokio.rs/tokio/tutorial/async

（本文定价1元）