2017-04-03

Async/Await替代Promise的6个理由

译者按: Node.js的异步编程方式有效提高了应用性能；然而回调地狱却让人望而生畏，Promise让我们告别回调函数，写出更优雅的异步代码；在实践过程中，却发现Promise并不完美；技术进步是无止境的，这时，我们有了Async/Await。

原文: 6 Reasons Why JavaScript's Async/Await Blows Promises Away
译者: Fundebug

为了保证可读性，本文采用意译而非直译。

Node.js 7.6已经支持async/await了，如果你还没有试过，这篇博客将告诉你为什么要用它。

Async/Await简介

对于从未听说过async/await的朋友，下面是简介:

- async/await是写异步代码的新方式，以前的方法有**回调函数**和**Promise**。
- async/await是基于Promise实现的，它不能用于普通的回调函数。
- async/await与Promise一样，是非阻塞的。
- async/await使得异步代码看起来像同步代码，这正是它的魔力所在。

Async/Await语法

示例中，getJSON函数返回一个promise，这个promise成功resolve时会返回一个json对象。我们只是调用这个函数，打印返回的JSON对象，然后返回"done"。

使用Promise是这样的:

```
const makeRequest = () =>
  getJSON()
    .then(data => {
      console.log(data)
      return "done"
    })

makeRequest()
```

```
const makeRequest = async () => {
  console.log(await getJSON())
  return "done"
}

makeRequest()
```

它们有一些细微不同:

- 函数前面多了一个aync关键字。await关键字只能用在aync定义的函数内。async函数会隐式地返回一个promise，该promise的reosolve值就是函数return的值。(示例中reosolve值就是字符串”done”)
- 第1点暗示我们不能在最外层代码中使用await，因为不在async函数内。

```
// 不能在最外层代码中使用await
await makeRequest()

// 这是会出事情的
makeRequest().then((result) => {
  // 代码
})
```

await getJSON()表示console.log会等到getJSON的promise成功reosolve之后再执行。

为什么Async/Await更好?

1. 简洁

由示例可知，使用Async/Await明显节约了不少代码。我们不需要写.then，不需要写匿名函数处理Promise的resolve值，也不需要定义多余的data变量，还避免了嵌套代码。这些小的优点会迅速累计起来，这在之后的代码示例中会更加明显。

2. 错误处理

Async/Await让try/catch可以同时处理同步和异步错误。在下面的promise示例中，try/catch不能处理JSON.parse的错误，因为它在Promise中。我们需要使用.catch，这样错误处理代码非常冗余。并且，在我们的实际生产代码会更加复杂。

```
      .then(result => {
        // JSON.parse可能会出错
        const data = JSON.parse(result)
        console.log(data)
      })
      // 取消注释，处理异步代码的错误
      // .catch((err) => {
      //   console.log(err)
      // })
  } catch (err) {
    console.log(err)
  }
}
```

使用aync/await的话，catch能处理JSON.parse错误:

```
const makeRequest = async () => {
  try {
    // this parse may fail
    const data = JSON.parse(await getJSON())
    console.log(data)
  } catch (err) {
    console.log(err)
  }
}
```

3. 条件语句

下面示例中，需要获取数据，然后根据返回数据决定是直接返回，还是继续获取更多的数据。

```
const makeRequest = () => {
  return getJSON()
    .then(data => {
      if (data.needsAnotherRequest) {
        return makeAnotherRequest(data)
          .then(moreData => {
            console.log(moreData)
            return moreData
```

```
        return data
      }
    })
}
```

这些代码看着就头痛。嵌套（6层），括号，return语句很容易让人感到迷茫，而它们只是需要将最终结果传递到最外层的Promise。

上面的代码使用async/await编写可以大大地提高可读性:

```
const makeRequest = async () => {
  const data = await getJSON()
  if (data.needsAnotherRequest) {
    const moreData = await makeAnotherRequest(data);
    console.log(moreData)
    return moreData
  } else {
    console.log(data)
    return data
  }
}
```

4. 中间值

你很可能遇到过这样的场景，调用promise1，使用promise1返回的结果去调用promise2，然后使用两者的结果去调用promise3。你的代码很可能是这样的:

```
const makeRequest = () => {
  return promise1()
    .then(value1 => {
      return promise2(value1)
        .then(value2 => {
          return promise3(value1, value2)
        })
    })
}
```

```
const makeRequest = () => {
  return promise1()
    .then(value1 => {
      return Promise.all([value1, promise2(value1)])
    })
    .then(([value1, value2]) => {
      return promise3(value1, value2)
    })
}
```

这种方法为了可读性牺牲了语义。除了避免嵌套，并没有其他理由将value1和value2放在一个数组中。

使用async/await的话，代码会变得异常简单和直观。

```
const makeRequest = async () => {
  const value1 = await promise1()
  const value2 = await promise2(value1)
  return promise3(value1, value2)
}
```

5. 错误栈

下面示例中调用了多个Promise，假设Promise链中某个地方抛出了一个错误:

```
const makeRequest = () => {
  return callAPromise()
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => {
      throw new Error("oops");
    })
}

makeRequest()
  .catch(err => {
    console.log(err);
```

Promise链中返回的错误栈没有给出错误发生位置的线索。更糟糕的是，它会误导我们；错误栈中唯一的函数名为callAPromise，然而它和错误没有关系。(文件名和行号还是有用的)。

然而，async/await中的错误栈会指向错误所在的函数:

```
const makeRequest = async () => {
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  throw new Error("oops");
}

makeRequest()
  .catch(err => {
    console.log(err);
    // output
    // Error: oops at makeRequest (index.js:7:9)
  })
```

在开发环境中，这一点优势并不大。但是，当你分析生产环境的错误日志时，它将非常有用。这时，知道错误发生在makeRequest比知道错误发生在then链中要好。

6. 调试

最后一点，也是非常重要的一点在于，async/await能够使得代码调试更简单。2个理由使得调试Promise变得非常痛苦:

- 不能在返回表达式的箭头函数中设置断点

```
        .then(() => callAPromise())
        .then(() => callAPromise())
        .then(() => callAPromise())
        .then(() => callAPromise())
    }
```

- 如果你在.then代码块中设置断点，使用Step Over快捷键，调试器不会跳到下一个.then，因为它只会跳过异步代码。

使用await/async时，你不再需要那么多箭头函数，这样你就可以像调试同步代码一样跳过await语句。

```
const makeRequest = async () => {
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
}
```

结论

Async/Await是近年来JavaScript添加的最革命性的的特性之一。它会让你发现Promise的语法有多糟糕，而且提供了一个直观的替代方法。

忧虑

对于Async/Await，也许你有一些合理的怀疑：

- 它使得异步代码不在明显: 我们已经习惯了用回调函数或者.then来识别异步代码，我们可能需要花数个星期去习惯新的标志。但是，C#拥有这个特性已经很多年了，熟悉它的朋友应该知道暂时的稍微不方便是值得的。
- Node 7不是LTS（长期支持版本）：但是，Node 8下个月就会发布，将代码迁移到新版本会非常简单。

欢迎加入我们Fundebug的**Node.js技术交流群: 177654062**。

关于Fundebug

Fundebug是全栈JavaScript实时错误监控平台，能够及时发现您的应用错误，助您提升用户体验。

累计处理错误:

标签

React JavaScript 新闻 小程序 BUG Source Map Node.js 翻译 Docker Linux ES6
媒体报道 客户故事 Hexo 函数式 人工智能 随笔 历史 用户行为 教程

最新博客

Fundebug上线微信小游戏错误监控！支持自动截屏！
JavaScript正则表达式进阶指南
详解配置Visual Studio/Webstorm来调试JavaScript
Fundebug获邀参加优客工场三周年庆典
为什么选择SaaS?
RabbitMQ入门教程
Fundebug种子用户回馈活动

热门博客