async 函数

1. 基本用法

async 函数返回一个 Promise 对象，可以使用 then 方法添加回调函数。当函数执行的时候，一旦遇到 await 就会先返回，等到异步操作完成，再接着执行函数体内后面的语句。

```
function timeout(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}
async function asyncPrint(value, ms) {
  await timeout(ms);
  console.log(value);
}
asyncPrint('hello world', 500);
// 500ms 后输出 hello world
```

async 函数有多种使用形式。

```
async function foo() {} // 函数声明

const foo = async () => {}; // 箭头函数

const foo = async function () {}; // 函数表达式

// 对象的方法
let obj = { async foo() {} };
obj.foo().then(...)

// Class 的方法
class Storage {
  constructor() {
    this.cachePromise = caches.open('avatars');
  }
  async getAvatar(name) {
    const cache = await this.cachePromise;
    return cache.match(`/avatars/${name}.jpg`);
  }
}
const storage = new Storage();
storage.getAvatar('jake').then(…);
```

2. 语法

2.1. 返回 Promise 对象

async 函数返回一个 Promise 对象。**async 函数内部 return 语句返回的值，会成为 then 方法回调函数的参数。**

```
async function f() {
  return 'hello world'; // 这里返回的值会被 f 函数 then 方法接收
}
f().then(v => console.log(v)); // "hello world"
```

async 函数内部抛出错误，会导致返回的 Promise 对象变为 reject 状态。**抛出的错误对象会被 catch 方法回调函数接收到。**

```
async function f() {
  throw new Error('出错了'); // 这里抛出的错误会被 f 函数 catch 方法（then 方法第二个参数）接收
}
f().then(v => console.log('resolve', v), e => console.log('reject', e)); // reject Error: 出错了
```

2.2. Promise 对象的状态变化

async 函数返回的 Promise 对象，必须等到内部所有 await 命令后面的 Promise 对象执行完，才会发生状态改变，除非遇到 return 语句或者抛出错误。也就是说，只有 async 函数内部的异步操作执行完，才会执行 then 方法指定的回调函数。

```
async function getTitle(url) {
  let response = await fetch(url);
  let html = await response.text();
  return html.match(/<title>([\s\S]+)<\/title>/i)[1];
}
getTitle('https://tc39.github.io/ecma262/').then(console.log);
// "ECMAScript 2017 Language Specification"
```

上面代码中，函数 getTitle 内部有三个操作：抓取网页、取出文本、匹配页面标题。只有这三个操作全部完成，才会执行 then 方法里面的 console.log。

2.3. await 命令

如果 await 命令后面是一个 Promise 对象，返回该对象的结果。如果不是 Promise 对象，就直接返回对应的值。

```
async function f() {
  return await 123; // 等同于 return 123;
```

```
}
f().then(v => console.log(v)); // 123
```

如果 await 命令后面是一个 thenable 对象（即定义了 then 方法的对象），那么 await 会将其等同于 Promise 对象。

```
class Sleep {
  constructor(timeout) {
    this.timeout = timeout;
  }
  then(resolve, reject) {
    const startTime = Date.now();
    setTimeout(
      () => resolve(Date.now() - startTime),
      this.timeout
    );
  }
}
(async () => {
  const sleepTime = await new Sleep(1000);
  console.log(sleepTime); // 1000
})();
```

上例中，await 命令后面是一个 Sleep 对象的实例。这个实例不是 Promise 对象，但是因为定义了 then 方法，await 会将其视为 Promise 处理。

JavaScript 一直没有休眠的语法，但是借助 await 命令就可以让程序停顿指定的时间。简化的 sleep 实现：

```
function sleep(interval) {
  return new Promise(resolve => {
    setTimeout(resolve, interval);
  })
}
// 用法
async function one2FiveInAsync() {
  for(let i = 1; i <= 3; i++) {
    console.log(i);
    await sleep(1000);
  }
}
one2FiveInAsync(); // 首先输出 1，1s 后输出 2，再 1s 后输出 3
```

await 命令后面的 Promise 对象如果变为 reject 状态，则 reject 的参数会被 catch 方法的回调函数接收到。

```
async function f() {
  await Promise.reject('出错了');
```

```
}
f().then(v => console.log("v", v)).catch(e => console.log("e", e)); // e 出错了
```

任何一个 `await` 语句后面的 `Promise` 对象变为 `reject` 状态，那么整个 `async` 函数都会中断执行。

```
async function f() {
  await Promise.reject('出错了');
  await Promise.resolve('hello world'); // 不会执行
}
f().then(v => console.log("v", v)).catch(e => console.log("e", e)); // e 出错了
```

上例中，第二个 `await` 语句是不会执行的，因为第一个 `await` 语句状态变成了 `reject`。

如果希望即使前一个异步操作失败，也不要中断后面的异步操作。这时可以将第一个 `await` 放在 `try...catch` 结构里面，这样不管这个异步操作是否成功，第二个 `await` 都会执行。

```
async function f() {
  try {
    await Promise.reject('出错了');
  } catch(e) {
  }
  return await Promise.resolve('hello world');
}

f().then(v => console.log(v)); // hello world
```

另一种方法是 `await` 后面的 `Promise` 对象再跟一个 `catch` 方法，处理前面可能出现的错误。

```
async function f() {
  await Promise.reject('出错了').catch(e => console.log("e", e));
  return await Promise.resolve('hello world');
}
f().then(v => console.log("v", v));
// e 出错了
// v hello world
```

2.4. 错误处理

如果 `await` 后面的异步操作出错，那么等同于 `async` 函数返回的 `Promise` 对象被 `reject`。

```
async function f() {
  await new Promise(function (v, e) {
    throw new Error('出错了');
  });
}
```

```
f().then(v => console.log("v", v)).catch(e => console.log("e", e)); // e Error: 出
错了
```

防止出错的方法，也是将其放在 `try...catch` 代码块之中。

```
async function f() {
  try {
    await new Promise(function (resolve, reject) {
      throw new Error('出错了');
    });
  } catch(e) {
  }
  return await('hello world');
}
```

如果有多个 `await` 命令，可以统一放在 `try...catch` 结构中。

```
async function main() {
  try {
    const val1 = await firstStep();
    const val2 = await secondStep(val1);
    const val3 = await thirdStep(val1, val2);
    console.log('Final: ', val3);
  }
  catch (err) {
    console.error(err);
  }
}
```

下面的例子使用 `try...catch` 结构，实现多次重复尝试。

```
const superagent = require('superagent');
const NUM_RETRIES = 3;

async function test() {
  let i;
  for (i = 0; i < NUM_RETRIES; ++i) {
    try {
      await superagent.get('http://google.com/this-throws-an-error');
      break;
    } catch(err) {}
  }
  console.log(i); // 3
}

test();
```

上例中，如果 await 操作成功，就会使用 break 语句退出循环；如果失败，会被 catch 语句捕捉，然后进入下一轮循环。

2.5. 使用注意点

> 1. await 命令后面的 Promise 对象，运行结果可能是 rejected，所以最好把 await 命令放在 try...catch 代码块中。

```
async function myFunction() {
  try {
    await somethingThatReturnsAPromise();
  } catch (err) {
    console.log(err);
  }
}
// 另一种写法
async function myFunction() {
  await somethingThatReturnsAPromise().catch(function (err) {
    console.log(err);
  });
}
```

> 2. 多个 await 命令后面的异步操作，如果不存在继发关系，最好让它们同时触发。

```
function timeout(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}
(async function asyncPrint() {
  await timeout(3000);
  await timeout(5000);
  console.log("a"); // 8s 后输出 a
})();
```

上例中，asyncPrint 函数中 timeout(3000) 和 timeout(5000) 是两个独立的异步操作（即互不依赖），被写成继发关系。这样比较耗时，因为只有 timeout(3000) 完成以后，才会执行 timeout(5000)，timeout(5000) 完成以后才会输出 a，完全可以让它们同时触发。

```
// 写法一
await Promise.all([timeout(3000), timeout(5000)]);

// 写法二
let p1 = timeout(3000);
let p2 = timeout(5000);
await p1;
await p2;
```

上面两种写法，`timeout(3000)` 和 `timeout(5000)` 都是同时触发，这样就会缩短程序的执行时间。5s 后会输出 `a`。

> 3. `await` 命令只能用在 `async` 函数之中，如果用在普通函数，就会报错。

```
async function dbFuc(db) {
  let docs = [{}, {}, {}];

  // 报错
  docs.forEach(function (doc) {
    await db.post(doc);
  });
}
```

如果确实希望多个请求并发执行，可以使用 `Promise.all` 方法。当三个请求都会 `resolved` 时，下面两种写法效果相同。

```
async function dbFuc(db) {
  let docs = [{}, {}, {}];
  let promises = docs.map((doc) => db.post(doc));

  let results = await Promise.all(promises);
  console.log(results);
}
// 或者使用下面的写法
async function dbFuc(db) {
  let docs = [{}, {}, {}];
  let promises = docs.map((doc) => db.post(doc));

  let results = [];
  for (let promise of promises) {
    results.push(await promise);
  }
  console.log(results);
}
```

> 4. async 函数可以保留运行堆栈。

```
const a = () => {
  b().then(() => c());
};
```

上例中，函数 `a` 内部运行了一个异步任务 `b()`。当 `b()` 运行的时候，函数 `a()` 不会中断，而是继续执行。等到 `b()` 运行结束，可能 `a()` 早就运行结束了，`b()` 所在的上下文环境已经消失了。如果 `b()` 或 `c()` 报错，错误堆栈将不包括 `a()`。

现在将这个例子改成 `async` 函数。

```
const a = async () => {
  await b();
  c();
};
```

上例中，`b()` 运行的时候，`a()` 是暂停执行，上下文环境都保存着。一旦 `b()` 或 `c()` 报错，错误堆栈将包括 `a()`。

3. 顶层 await

允许在模块的顶层独立使用 `await` 命令，解决模块异步加载的问题。

```
// import() 方法加载
const strings = await import(`/i18n/${navigator.language}`);

// 数据库操作
const connection = await dbConnector();

// 依赖回滚
let jQuery;
try {
  jQuery = await import('https://cdn-a.com/jQuery');
} catch {
  jQuery = await import('https://cdn-b.com/jQuery');
}
```