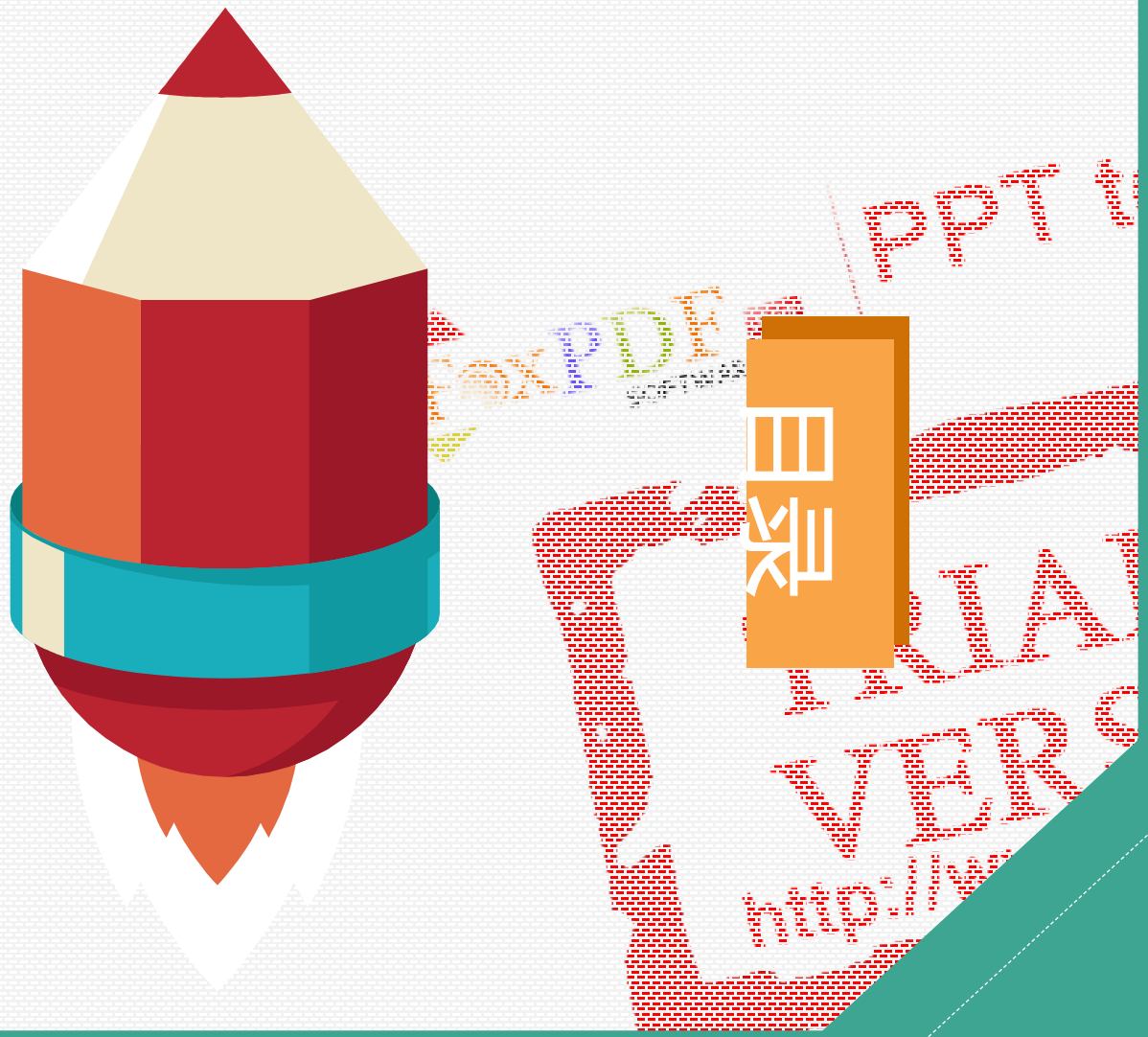

第43课：async函数
主讲老师：万章



- async函数的含义
- async函数的基本用法
- async函数的语法应用
- async函数的实现原理
- async函数的其他问题

async函数的含义

async函数的含义

async 函数是什么？一句话，它就是 Generator 函数的语法糖。

```
const fs = require('fs');

const readFile = function (fileName) {
  return new Promise(function (resolve, reject) {
    fs.readFile(fileName, function(error, data) {
      if (error) return reject(error);
      resolve(data);
    });
  });
};

const gen = function* () {
  const f1 = yield readFile('/etc/fstab');
  const f2 = yield readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

前文有一个 Generator 函数，依次读取两个文件。

```
const asyncReadFile = async function () {
  const f1 = await readFile('/etc/fstab');
  const f2 = await readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

上面代码的函数gen可以写成async函数，就是下面这样

一比较就会发现，async函数就是将 Generator 函数的星号（*）替换成async，将yield替换成await，仅此而已。

async函数的改进之处

（1）内置执行器。

Generator 函数的执行必须靠执行器，而async函数自带执行器。也就是说，async函数的执行，与普通函数一模一样，只要一行。

```
const asyncReadFile = async function () {
    await console.log("hello");
    await console.log("world");
};
```

```
> asyncReadFile();
  hello                                    demo.html:288
  world                                    demo.html:289
< ▶ Promise {<resolved>: undefined}
>
```

上面的代码调用了asyncReadFile函数，然后它就会自动执行，输出最后结果。这完全不像Generator 函数，需要调用next方法才能真正执行，得到最后结果。

async函数的改进之处

(2) 更好的语义。

async和await，比起星号和yield，语义更清楚了。**async表示函数里有异步操作，await表示紧跟在后面的表达式需要等待结果。**

(3) 更广的适用性。

async函数的await命令后面，可以是 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时会自动转成立即 resolved 的 Promise 对象）。

(4) 返回值是 Promise。

async函数的返回值是 Promise 对象，这比 Generator 函数的返回值是 Iterator 对象方便多了。你可以用then方法指定下一步的操作。

async函数的基本用法

async函数的基本用法

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', 5000);
```

```
Promise {<pending>}
  __proto__: Promise
  [[PromiseStatus]]: "resolved"
  [[PromiseValue]]: undefined
hello world
```

从上面的代码可以看出，async函数会先输出一个promise对象，然后会监听await后面promise对象的返回状态，当这状态为resolved的时候再往下执行其他的代码

async函数的基本用法

由于async函数返回的是 Promise 对象，可以作为await命令的参数。所以，上一页的例子也可以写成下面的形式。

```
async function timeout(ms) {
  await new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}

async function asyncPrint(value, ms) {
  await timeout(ms);
  console.log(value);
}

asyncPrint('hello world', 50);
```

```
// 函数声明
async function foo() {}

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

// 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(…);

// 箭头函数
const foo = async () => {};
```

async 函数有多种使用形式。

async函数的语法应用

async函数的语法应用

async函数的语法规则总体上比较简单，难点是错误处理机制。

1. async函数的返回值

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';
}

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

左侧代码中，函数f内部return命令返回的值，会被then方法回调函数接收到。

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');
}

f().then(
  v => console.log(v),
  e => console.log(e)
)
// Error: 出错了
```

async函数的语法应用

async函数的语法规则总体上比较简单，难点是错误处理机制。

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。

async函数的语法应用

async函数的语法规则总体上比较简单，难点是错误处理机制。

3. await 命令

正常情况下，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() {
  // 等同于
  // return 123;
  return await 123;
}

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

```
class Sleep {
    constructor(timeout) {
        this.timeout = timeout;
    }
    then(resolve, reject) {
        const startTime = Date.now();
        setTimeout(
            () => resolve("你好世界"),
            this.timeout
        );
    }
}

(async () => {
    const sleepTime = await new Sleep(1000);
    console.log(sleepTime);
})();
```

上面代码中，await命令的参数是数值123，这时等同于return 123。

另一种情况是，await命令后面是一个thenable对象（即定义then方法的对象），那么await会将其等同于

async函数的语法应用

async函数的语法规则总体上比较简单，难点是错误处理机制。

3. await 命令

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

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

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');
  await Promise.resolve('hello world'); // 不会执行
}
```

上面代码中，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。

async函数的语法应用

async函数的语法规则总体上比较简单，难点是错误处理机制。

3. await 命令

```
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。

有时，我们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() {
  await Promise.reject('出错了')
    .catch(e => console.log(e));
  return await Promise.resolve('hello world');
}

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

async函数的语法应用

async函数的语法规则总体上比较简单，难点是错误处理机制。

4．使用注意点

```
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);
  });
}
```

第一点，前面已经说过，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函数的语法应用

async函数的语法规则总体上比较简单，难点是错误处理机制。

4．使用注意点

第二点，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。

```
let foo = await getFoo();
let bar = await getBar();
```

上面代码中，getFoo和getBar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getFoo完成以后，才会执行getBar，完全可以让它们同时触发。

```
// 写法一
let [foo, bar] = await Promise.all([getFoo(), getBar()]);

// 写法二
let fooPromise = getFoo();
let barPromise = getBar();
let foo = await fooPromise;
let bar = await barPromise;
```

上面两种写法，getFoo和getBar都是同时触发，这样就会缩短程序的执行时间。

async函数的语法应用

async函数的语法规则总体上比较简单，难点是错误处理机制。

4．使用注意点

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

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

第三点，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函数的语法应用

async函数的语法规则总体上比较简单，难点是错误处理机制。

4．使用注意点

```
function dbFuc(db) { //这里不需要 async
  let docs = [{}, {}, {}];

  // 可能得到错误结果
  docs.forEach(async function (doc) {
    await db.post(doc);
  });
}
```

但是，如果将forEach方法的参数改成async函数，也有问题。

上面代码可能不会正常工作，原因是这时三个db.post操作将是并发执行，也就是同时执行，而不是继发执行(for each的特性)。正确的写法是采用for循环。

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

  for (let doc of docs) {
    await db.post(doc);
  }
}
```

async函数的语法应用

async函数的语法规则总体上比较简单，难点是错误处理机制。

4．使用注意点

```
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);
}
```

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

async函数的语法应用

async函数的语法规则总体上比较简单，难点是错误处理机制。

4．使用注意点

第四点，async 函数可以保留运行堆栈。

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

上面代码中，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()。

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

上面代码中，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()。

async函数的实现原理

async函数的实现原理

async 函数的实现原理，就是将 Generator 函数和自动执行器，包装在一个函数里。

```
async function fn(args) {
  // ...
}

// 等同于

function fn(args) {
  return spawn(function* () {
    // ...
  });
}
```

```
function spawn(genF) {
  return new Promise(function(resolve, reject) {
    const gen = genF();
    function step(nextF) {
      let next;
      try {
        next = nextF();
      } catch(e) {
        return reject(e);
      }
      if(next.done) {
        return resolve(next.value);
      }
      Promise.resolve(next.value).then(function(v) {
        step(function() { return gen.next(v); });
      }, function(e) {
        step(function() { return gen.throw(e); });
      });
    }
    step(function() { return gen.next(undefined); });
  });
}
```

所有的async函数都可以写成上面的第二种形式，其中的spawn函数就是自动执行器。

async函数的其他问题

async函数与其他异步处理方法的比较

```
function chainAnimationsPromise(elem, animations) {

  // 变量ret用来保存上一个动画的返回值
  let ret = null;

  // 新建一个空的Promise
  let p = Promise.resolve();

  // 使用then方法，添加所有动画
  for(let anim of animations) {
    p = p.then(function(val) {
      ret = val;
      return anim(elem);
    });
  }

  // 返回一个部署了错误捕捉机制的Promise
  return p.catch(function(e) {
    /* 忽略错误，继续执行 */
  }).then(function() {
    return ret;
  });

}
```

Promise 的写法

```
function chainAnimationsGenerator(elem, animations) {

  return spawn(function*() {
    let ret = null;
    try {
      for(let anim of animations) {
        ret = yield anim(elem);
      }
    } catch(e) {
      /* 忽略错误，继续执行 */
    }
    return ret;
  });

}
```

Generator 函数的写法

```
async function chainAnimationsAsync(elem, animations) {
  let ret = null;
  try {
    for(let anim of animations) {
      ret = await anim(elem);
    }
  } catch(e) {
    /* 忽略错误，继续执行 */
  }
  return ret;
}
```

async 函数的写法

假定某个 DOM 元素上面，部署了一系列的动画，前一个动画结束，才能开始后一个。如果当中有一个动画出错，就不再往下执行，返回上一个成功执行的动画的返回值。

实例：按顺序完成异步操作

```
function logInOrder(urls) {
  // 远程读取所有URL
  const textPromises = urls.map(url => {
    return fetch(url).then(response => response.text());
  });

  // 按次序输出
  textPromises.reduce((chain, textPromise) => {
    return chain.then(() => textPromise)
      .then(text => console.log(text));
  }, Promise.resolve());
}
```

Promise 的写法

```
async function logInOrder(urls) {
  for (const url of urls) {
    const response = await fetch(url);
    console.log(await response.text());
  }
}
```

继发版async写法

```
async function logInOrder(urls) {
  // 并发读取远程URL
  const textPromises = urls.map(async url => {
    const response = await fetch(url);
    return response.text();
  });

  // 按次序输出
  for (const textPromise of textPromises) {
    console.log(await textPromise);
  }
}
```

并发版async写法

实际开发中，经常遇到一组异步操作，需要按照顺序完成。比如，依次远程读取一组URL，然后按照读取的顺序输出结果。