ES 6

momo

ASYNC 函数

1 含义

- 就是 Generator 函数的语法糖。

GENERATOR 函数

1 简介

- ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同。
- 语法上：首先可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态。
- 执行 Generator 函数会返回一个遍历器对象，也就是说，Generator 函数除了状态机，还是一个遍历器对象生成函数。返回的遍历器对象，可以依次遍历 Generator 函数内部的每一个状态。
- 形式上：

```
function* helloWorldGenerator() {
  yield 'hello';
  yield 'world';
  return 'ending';
}

var hw = helloWorldGenerator();
```

1 简介

- 用法：

```
hw.next()
// { value: 'hello', done: false }

hw.next()
// { value: 'world', done: false }

hw.next()
// { value: 'ending', done: true }

hw.next()
// { value: undefined, done: true }
```

2 YIELD 表达式

- 暂停的标志。

```
function* gen() {
  yield  123 + 456;
}
```

YIELD & RETURN

- 相似：都能返回紧跟在语句后面的那个表达式的值
- 区别：多次 & 单次

- Generator 函数可以不用yield表达式，这时就变成了一个单纯的暂缓执行函数。

```
function* f() {
  console.log('执行了! ')
}

var generator = f();

setTimeout(function () {
  generator.next()
}, 2000);
```

3 NEXT 方法的参数

- yield表达式本身没有返回值，或者说总是返回undefined。next方法可以带一个参数，该参数就会被当作上一个yield表达式的返回值。

```
function* f() {
  for(var i = 0; true; i++) {
    var reset = yield i;
    if(reset) { i = -1; }
  }
}

var g = f();

g.next() // { value: 0, done: false }
g.next() // { value: 1, done: false }
g.next(true) // { value: 0, done: false }
```

```
function* foo(x) {
  var y = 2 * (yield (x + 1));
  var z = yield (y / 3);
  return (x + y + z);
}

var a = foo(5);
a.next() // Object{value:6, done:false}
a.next() // Object{value:NaN, done:false}
a.next() // Object{value:NaN, done:true}

var b = foo(5);
b.next() // { value:6, done:false }
b.next(12) // { value:8, done:false }
b.next(13) // { value:42, done:true }
```

- next方法的参数表示上一个yield表达式的返回值

```
function* dataConsumer() {
  console.log('Started');
  console.log(`1. ${yield}`);
  console.log(`2. ${yield}`);
  return 'result';
}

let genObj = dataConsumer();
genObj.next();
// Started
genObj.next('a')
// 1. a
genObj.next('b')
// 2. b
```

4 FOR ... OF

```
function* foo() {
  yield 1;
  yield 2;
  yield 3;
  yield 4;
  yield 5;
  return 6;
}

for (let v of foo()) {
  console.log(v);
}
// 1 2 3 4 5
```

- 上面代码使用for...of循环，依次显示 5 个yield表达式的值。这里需要注意，一旦next方法的返回对象的done属性为true，for...of循环就会中止，且不包含该返回对象，所以上面代码的return语句返回的6，不包括在for...of循环之中。

- 除了for...of循环以外，扩展运算符（...）、解构赋值和Array.from方法内部调用的，都是遍历器接口。这意味着，它们都可以将Generator 函数返回的 Iterator 对象，作为参数。

```
function* numbers () {
  yield 1
  yield 2
  return 3
  yield 4
}

// 扩展运算符
[...numbers()] // [1, 2]

// Array.from 方法
Array.from(numbers()) // [1, 2]

// 解构赋值
let [x, y] = numbers();
x // 1
y // 2

// for...of 循环
for (let n of numbers()) {
  console.log(n)
}
// 1
// 2
```

5 应用

- 异步操作的同步化表达

```
function* loadUI() {
  showLoadingScreen();
  yield loadUIDataAsynchronously();
  hideLoadingScreen();
}
var loader = loadUI();
// 加载UI
loader.next()

// 卸载UI
loader.next()
```

- Ajax 是典型的异步操作，通过 Generator 函数部署 Ajax 操作，可以用同步的方式表达。

```
function* main() {
  var result = yield request("http://some.url");
  var resp = JSON.parse(result);
    console.log(resp.value);
}

function request(url) {
  makeAjaxCall(url, function(response){
    it.next(response);
  });
}

var it = main();
it.next();
```

5 应用

- 控制流管理

```
step1(function (value1) {
  step2(value1, function(value2) {
    step3(value2, function(value3) {
      step4(value3, function(value4) {
        // Do something with value4
      });
    });
  });
});
```

```
Promise.resolve(step1)
  .then(step2)
  .then(step3)
  .then(step4)
  .then(function (value4) {
    // Do something with value4
  }, function (error) {
    // Handle any error from step1 through step4
  })
  .done();
```

```
function* longRunningTask(value1) {
  try {
    var value2 = yield step1(value1);
    var value3 = yield step2(value2);
    var value4 = yield step3(value3);
    var value5 = yield step4(value4);
    // Do something with value4
  } catch (e) {
    // Handle any error from step1 through step4
  }
}
```

```
scheduler(longRunningTask(initialValue));

function scheduler(task) {
  var taskObj = task.next(task.value);
  // 如果Generator函数未结束，就继续调用
  if (!taskObj.done) {
    task.value = taskObj.value
    scheduler(task);
  }
}
```

```
let steps = [step1Func, step2Func, step3Func];

function* iterateSteps(steps){
  for (var i=0; i< steps.length; i++){
    var step = steps[i];
    yield step();
  }
}
```

5 应用

- 部署 Iterator 接口

```
function* iterEntries(obj) {
  let keys = Object.keys(obj);
  for (let i=0; i < keys.length; i++) {
    let key = keys[i];
    yield [key, obj[key]];
  }
}

let myObj = { foo: 3, bar: 7 };

for (let [key, value] of iterEntries(myObj)) {
  console.log(key, value);
}

// foo 3
// bar 7
```

ASYNC 函数

1 含义

- 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());
};
```

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

1 含义

```
asyncReadFile();
```

- 改进
- 1 内置执行器。Generator 函数的执行必须靠执行器，所以才有了co模块，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。
- 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。co模块约定，yield命令后面只能是 Thunk 函数或 Promise 对象，而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，但这时等同于同步操作）。
- 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。

co 模块

- co模块是koa框架实现的关键技术，主要解决的是node.js的回调函数嵌套过多的问题。
  它用到了ES6的新特性generator函数, promise技术，以及thunk函数。
- **回调地狱问题**：异步函数因为其结束时间的不确定性，只能在其回调函数中处理其产生的数据。
  因此多个异步函数结果需要顺序执行时候，就只能通过回调函数一步步的嵌套执行，造成代码可读性很差。

2 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', 50);
```

- 由于async函数返回的是 Promise 对象，可以作为await命令的参数。所以，上面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);
```

3 语法

- **返回 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"
```

- 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
```

3 语法

- **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"
```

3 语法

- **await 命令**
- 正常情况下，await命令后面是一个 Promise 对象。如果不是，会被转成一个立即resolve的 Promise 对象。

```
async function f() {
  return await 123;
}

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

- 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))
.catch(e => console.log(e))
// 出错了
```

3 语法

- **await 命令**
- 只要一个await语句后面的 Promise 变为reject，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
}
```

- 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() {
  try {
    await Promise.reject('出错了');
  } catch(e) {
  }
  return await Promise.resolve('hello world');
}

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

4 与其他异步处理方法的比较

- async 函数与 Promise、Generator 函数的比较。
- 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。

PROMISE写法

```
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;
  });

}
```

GENERATOR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;
  });

}
```

ASYNC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;
}
```

4 与其他异步处理方法的比较

- async 函数与 Promise、Generator 函数的比较。
- **按顺序完成异步操作**
- 比如，依次远程读取一组 URL，然后按照读取的顺序输出结果。

PROMISE写法

```
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());
}
```

ASYNC写法

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

```
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);
  }
}
```