变量的解构赋值

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。

字符串解构赋值

```
const [a, b, c, d, e] = 'hello';
a // "h"
b // "e"
c // "l"
d // "l"
e // "o"
```

数组的解构赋值

以前，为变量赋值，只能直接指定值。

```
var a = 1;
var b = 2;
var c = 3;
//ES6允许写成下面这样。
let [a, b, c] = [1, 2, 3];
```

上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

本质上，这种写法属于“模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。下面是一些使用嵌套数组进行解构的例子。

```
let [foo, [[bar], baz]] = [1, [[2], 3]];
foo // 1
bar // 2
baz // 3

let [ , , third] = ["foo", "bar", "baz"];
third // "baz"

let [x, , y] = [1, 2, 3];
x // 1
y // 3

let [head, ...tail] = [1, 2, 3, 4];
head // 1
tail // [2, 3, 4]

let [x, y, ...z] = ['a'];
x // "a"
```

```
y // undefined
z // []
```

对象的解构赋值

解构不仅可以用于数组，还可以用于对象。

```
let { foo, bar } = { foo: "aaa", bar: "bbb" };
foo // "aaa"
bar // "bbb"
```

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

```
let { bar, foo } = { foo: "aaa", bar: "bbb" };
foo // "aaa"
bar // "bbb"

let { name } = { foo: "aaa", bar: "bbb" };
name // undefined
```

上面代码的第一个例子，等号左边的两个变量的次序，与等号右边两个同名属性的次序不一致，但是对取值完全没有影响。第二个例子的变量没有对应的同名属性，导致取不到值，最后等于`undefined`。

对象的扩展

属性的简洁表示法

ES6 允许直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

```
const foo = 'bar';
const baz = {foo};
baz // {foo: "bar"}

// 等同于
const baz = {foo: foo};
```

上面代码表明，ES6 允许在对象之中，直接写变量。这时，属性名为变量名, 属性值为变量的值。

方法的简洁表示法

除了属性简写，方法也可以简写。

```
function fun (){
    console.log(this.a + this.b);
}
```

```
var obj = {
    a:1,
    fun:fun,
    b:2,
};
obj.fun(); // 3
// 等同于
var obj = {
    a:3,
    fun,
    b:4,
};
obj.fun(); // 7
// 等同于
var obj = {
    a:5,
    fun(){console.log(this.a + this.b);},
    b:6,
};
obj.fun(); // 11
```

CommonJS 模块输出一组数据及方法，就非常合适使用简洁写法。

```
function getItem() {
}

function setItem() {
}

function clear() {
}

module.exports = { getItem, setItem, clear };
// 等同于
module.exports = {
    getItem: getItem,
    setItem: setItem,
    clear: clear
};
```

Promise 异步控制对象

> https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Using_promises

为什么要使用 Promise

以读取文件内容为例：

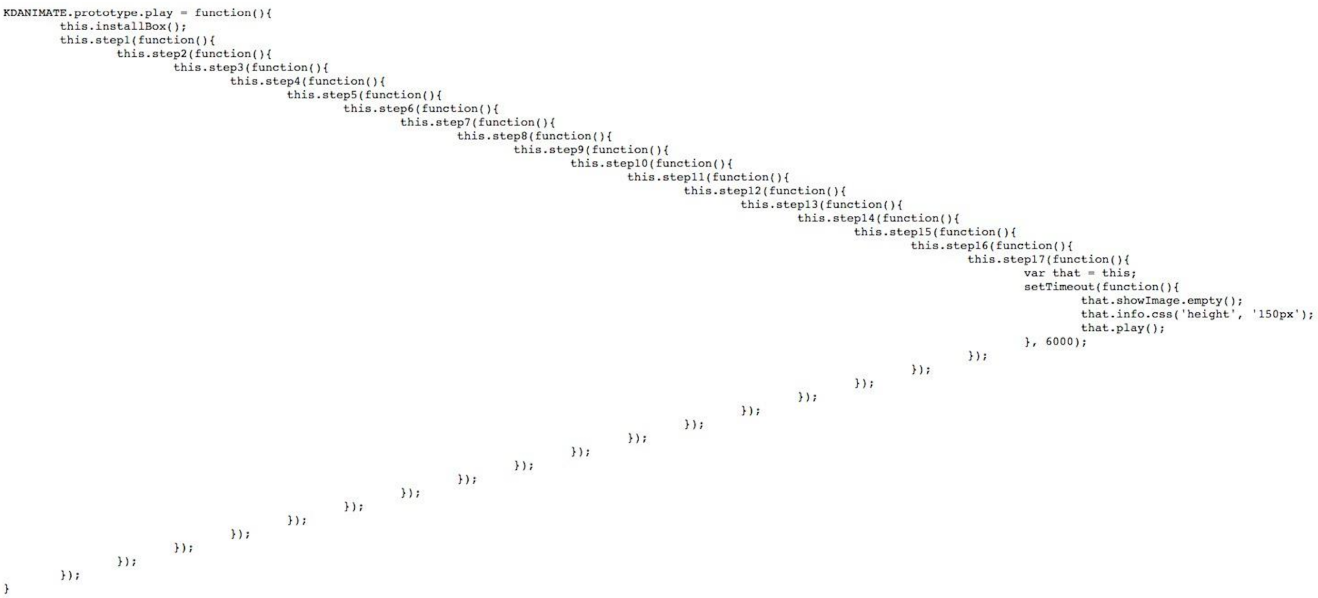
无法保证顺序的代码

```
var fs = require('fs')

fs.readFile('./data/a.txt', 'utf8', function (err, data) {
  if (err) {
    // return console.log('读取失败')
    // 抛出异常
    //    1. 阻止程序的执行
    //    2. 把错误消息打印到控制台
    throw err
  }
  console.log(data)
})

fs.readFile('./data/b.txt', 'utf8', function (err, data) {
  if (err) {
    // return console.log('读取失败')
    // 抛出异常
    //    1. 阻止程序的执行
    //    2. 把错误消息打印到控制台
    throw err
  }
  console.log(data)
})

fs.readFile('./data/c.txt', 'utf8', function (err, data) {
  if (err) {
    // return console.log('读取失败')
    // 抛出异常
    //    1. 阻止程序的执行
    //    2. 把错误消息打印到控制台
    throw err
  }
  console.log(data)
})
```

通过回调嵌套的方式来保证顺序：

```
var fs = require('fs')

fs.readFile('./data/a.txt', 'utf8', function (err, data) {
  if (err) {
    // return console.log('读取失败')
    // 抛出异常
    //    1. 阻止程序的执行
    //    2. 把错误消息打印到控制台
    throw err
  }
  console.log(data)
```

```
  fs.readFile('./data/b.txt', 'utf8', function (err, data) {
    if (err) {
      // return console.log('读取失败')
      // 抛出异常
      //    1. 阻止程序的执行
      //    2. 把错误消息打印到控制台
      throw err
    }
    console.log(data)
    fs.readFile('./data/c.txt', 'utf8', function (err, data) {
      if (err) {
        // return console.log('读取失败')
        // 抛出异常
        //    1. 阻止程序的执行
        //    2. 把错误消息打印到控制台
        throw err
      }
      console.log(data)
    })
  })
})
```



多层使用回调函数，就会进入 "**回调地狱**"

为了解决以上编码方式带来的问题（回调地狱嵌套），所以在 EcmaScript 6 中新增了一个 API：`Promise`。

Promise 的使用

```
var fs = require('fs')

// 在 EcmaScript 6 中新增了一个 API Promise
```

```
// Promise 是一个构造函数

// 创建 Promise 容器
// 1. 给别人一个承诺 I promise you.
//    Promise 容器一旦创建，就开始执行里面的代码
var p1 = new Promise(function (resolve, reject) {
  // console.log(2)
  fs.readFile('./data/aa.txt', 'utf8', function (err, data) {
    if (err) {
      // 失败了，承诺容器中的任务失败了
      // console.log(err)
      // 把容器的 Pending 状态变为 Rejected

      // 调用 reject 就相当于调用了 then 方法的第二个参数函数
      reject(err)
    } else {
      // console.log(3)
      // 承诺容器中的任务成功了
      // console.log(data)
      // 把容器的 Pending 状态改为成功 Resolved
      // 也就是说这里调用的 resolve 方法实际上就是 then 方法传递的那个 function
      resolve(data)
    }
  })
})

// console.log(4)

// p1 就是那个承若
// 当 p1 成功了 然后(then) 做指定的操作
// then 方法接收的 function 就是容器中的 resolve 函数
p1
  .then(function (data) {
    console.log(data)
  }, function (err) {
    console.log('读取文件失败了', err)
  })
```

封装 Promise 版本的 `readFile`：

```
var fs = require('fs')

function pReadFile(filePath) {
  return new Promise(function (resolve, reject) {
    fs.readFile(filePath, 'utf8', function (err, data) {
      if (err) {
        reject(err)
      } else {
        resolve(data)
      }
    })
```

```
  })
}

pReadFile('./data/a.txt')
  .then(function (data) {
    console.log(data)
    return pReadFile('./data/b.txt')
  })
  .then(function (data) {
    console.log(data)
    return pReadFile('./data/c.txt')
  })
  .then(function (data) {
    console.log(data)
  })
```

箭头函数

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/Arrow_functions

ES6 允许使用“箭头”（`=>`）定义函数。

箭头函数表达式的语法比 *函数表达式* 更短，并且没有自己的 this，arguments。这些函数表达式更适用于那些本来需要匿名函数的地方，并且它们不能用作构造函数。

```
var fun1 = function(str1,str2){
    return (str1 + str2);
}
// ↓

var fun2 = (str1,str2)=>{
    return (str1 + str2);
}
// ↓

var fun3 = (str1,str2) => str1 + str2;


console.log(fun1(1,2)); // 3
console.log(fun2(2,3)); // 5
console.log(fun1(4,5)); // 9
```

> 如果参数只有一个，可以将()省略 // arr.map(c=>c+1); 如果没有参数，则一定能要写上() // ()=> console.log('a') 如果多于一个参数，每个参数之间用逗号分隔 (x, y) => { ... } 如果方法体只有一句代码，可以省略{} 和分号，如果有返回可以省略return 如果方法体多于一句代码，则不能省略{} ,每句代码使用 分号分隔

注意：

> a. 箭头函数没有自己的this，函数体内部写的this，指向的是外层代码块的this b. 箭头函数内部的this是定义时所在的对象，而不是使用时所在的对象并且不会改变 c. 箭头箭头函数不能用作构造函数 d. 箭头函数内部不存在arguments，箭头函数体中使用的arguments其实指向的是外层函数的arguments