封装 AJAX

- ajax 使用起来太麻烦，因为每次都要写很多的代码
- 那么我们就封装一个 ajax 方法来让我们使用起来简单一些

确定一下使用的方式

- 因为有一些内容可以不传递，我们可以使用默认值，所以选择对象传递参数的方式

```
// 使用的时候直接调用，传递一个对象就可以
ajax({
  url: '', // 请求的地址
  type: '', // 请求方式
  async: '', // 是否异步
  data: '', // 携带的参数
  dataType: '', // 要不要执行 json.parse
  success: function () {} // 成功以后执行的函数
})
```

  - 确定好使用方式以后，就开始书写封装函数

封装

```
function ajax(options) {
  // 先准备一个默认值
  var defInfo = {
  url: '', // 地址不需要默认值
  type: 'GET', // 请求方式的默认值是 GET
  async: false, // 默认值是异步
  data: '', // 参数没有默认值
  dataType: 'string', // 默认不需要执行 json.parse
  success () {}, // 默认是一个函数
  }

  // 先来判断一下有没有传递 url，如果没有，直接抛出异常
  if (!options.url) {
  throw new Error('url 必须传递')
  }

```

```
17.   // 有了 url 以后就，我们就把用户传递的参数和我们的默认数据合并
18.   for (let key in options) {
19.   defInfo[key] = options[key]
20.   }
21.
22.   // 接下来的一切我们都是使用我们的 defInfo 就可以了
23.   // 第一步就是判断参数 data
24.   // data 可以不传递，可以为空
25.   // data 也可以是一个 key=value&key=value 格式的字符串
26.   // data 也可以是一个对象
27.   // 否则就抛出异常
28.   if (!(typeof defInfo.data === 'string' && /^(\w+=\w+&?)*$/.test(defInfo.data) ||
Object.prototype.toString.call(defInfo.data) === '[object Object]')) {
29.   throw new Error('请按照要求传递参数')
30.   }
31.
32.   // 参数处理完毕以后，在判断 async 的数据类型
33.   // 只能传递 布尔数据类型
34.   if (typeof defInfo.async !== 'boolean') {
35.   throw new Error('async 参数只接受布尔数据类型')
36.   }
37.
38.   // 在接下来就判断 type
39.   // 请求方式我们只接受 GET 或着 POST
40.   if (!(defInfo.type.toUpperCase() === 'GET' || defInfo.type.toUpperCase() === 'POST'))
{
41.   throw new Error('目前本插件只接受 GET 和 POST 方式，请期待更新')
42.   }
43.
44.   // 接下来就是判断 success 的判断，必须是一个函数
45.   if (Object.prototype.toString.call(defInfo.success) !== '[object Function]') {
46.   throw new Error('success 只接受函数数据类型')
47.   }
48.
49.   // 参数都没有问题了
50.   // 我们就要把 data 处理一下了
51.   // 因为 data 有可能是对象，当 data 是一个对象的时候，我们要把它转换成一个字符串
52.   var str = ''
53.   if (Object.prototype.toString.call(defInfo.data) === '[object Object]') {
54.   for (let attr in defInfo.data) {
55.   str += `${attr}=${defInfo.data[attr]}&`
```

```
56.   }
57.   str = str.slice(0, -1)
58.   defInfo.data = str
59.   }
60.
61.   // 参数全部验证过了以后，我们就可以开始进行正常的 ajax 请求了
62.   // 1. 准备一个 ajax 对象
63.   // 因为要处理兼容问题，所以我们准备一个函数
64.   function createXHR() {
65.   if (XMLHttpRequest) {
66.   return new XMLHttpRequest()
67.   } else {
68.   return new ActiveXObject('Microsoft.XMLHTTP')
69.   }
70.   }
71.
72.   // 2. 创建一个 ajax 对象
73.   var xhr = createXHR()
74.
75.   // 3. 进行 open
76.   xhr.open(defInfo.type, defInfo.url + (defInfo.type.toUpperCase() === 'GET' ?
`?${defInfo.data}&_=${new Date().getTime()}` : ''), defInfo.async)
77.
78.   if (defInfo.type.toUpperCase() === 'POST') {
79.   xhr.setRequestHeader('content-type', 'application/x-www-form-urlencoded')
80.   }
81.
82.   // 4. 进行 send
83.   xhr.send((defInfo.type.toUpperCase() === 'POST' ? `${defInfo.data}` : ''))
84.
85.   // 5. 接受响应
86.   xhr.onreadystatechange = function () {
87.   if (xhr.readyState === 4 && /2\d{2}/.test(xhr.status)) {
88.   // 表示成功，我们就要执行 success
89.   // 但是要进行 dataType 的判断
90.   if (defInfo.dataType === 'json') {
91.   defInfo.success(JSON.parse(xhr.responseText))
92.   } else {
93.   defInfo.success()
94.   }
95.   }
```

```
    }
  }
```

Promise

- `promise` 是一个 ES6 的语法
- 承诺的意思，是一个专门用来解决异步 **回调地狱** 的问题

回调函数

- 什么是回调函数？
- 就是把函数 A 当作参数传递到 函数 B 中
- 在函数 B 中以行参的方式进行调用

```
function a(cb) {
 cb()
}

function b() {
 console.log('我是函数 b')
}

a(b)
```

- 为什么需要回调函数
  - 当我们执行一个异步的行为的时候，我们需要在一个异步行为执行完毕之后做一些事情
  - 那么，我们是没有办法提前预知这个异步行为是什么时候完成的
  - 我们就只能以回调函数的形式来进行
  - 就比如我们刚刚封装过的那个 `ajax` 函数里面的 `success`
  - 我们并不知道 ajax 请求什么时候完成，所以就要以回调函数的形式来进行

回调地狱

- 当一个回调函数嵌套一个回调函数的时候
- 就会出现一个嵌套结构

- 当嵌套的多了就会出现回调地狱的情况
- 比如我们发送三个 ajax 请求
  - 第一个正常发送
  - 第二个请求需要第一个请求的结果中的某一个值作为参数
  - 第三个请求需要第二个请求的结果中的某一个值作为参数

```
ajax({
 url: '我是第一个请求',
 success (res) {
 // 现在发送第二个请求
 ajax({
 url: '我是第二个请求',
 data: { a: res.a, b: res.b },
 success (res2) {
 // 进行第三个请求
 ajax({
 url: '我是第三个请求',
 data: { a: res2.a, b: res2.b },
 success (res3) {
 console.log(res3)
 }
 })
 }
 })
 }
})
```

- **回调地狱，其实就是回调函数嵌套过多导致的**

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

- 当代码成为这个结构以后，已经没有维护的可能了
- 所以我们要把代码写的更加的艺术一些

PROMISE

- 为了解决回调地狱
- 我们就要使用 promise 语法
- 语法：

```
new Promise(function (resolve, reject) {
  // resolve 表示成功的回调
  // reject 表示失败的回调
}).then(function (res) {
  // 成功的函数
}).catch(function (err) {
  // 失败的函数
})
```

- promise 就是一个语法
  - 我们的每一个异步事件，在执行的时候
  - 都会有三个状态，执行中 / 成功 / 失败
- 因为它包含了成功的回调函数
- 所以我们就可以使用 promise 来解决多个 ajax 发送的问题

```
new Promise(function (resolve, reject) {
  ajax({
  url: '第一个请求',
  success (res) {
  resolve(res)
  }
  })
}).then(function (res) {
  // 准备发送第二个请求
  return new Promise(function (resolve, reject) {
```

```
    ajax({
    url: '第二个请求',
    data: { a: res.a, b: res.b },
    success (res) {
    resolve(res)
    }
    })
    })
  }).then(function (res) {
    ajax({
    url: '第三个请求',
    data: { a: res.a, b: res.b },
    success (res) {
    console.log(res)
    }
    })
  })
```

- 这个时候，我们的代码已经改观了很多了
- 基本已经可以维护了
- 但是对于一个程序员来说，这个样子是不够的
- 我们还需要更加的简化代码
- 所以我们就需要用到一个 es7 的语法了
- 叫做 async/await

ASYNC/AWAIT

- `async/await` 是一个 es7 的语法
- 这个语法是 **回调地狱的终极解决方案**
- 语法：

```
async function fn() {
  const res = await promise对象
}
```

- 这个是一个特殊的函数方式

- 这个是一个特殊的函数方式
- 可以 await 一个 promise 对象
- **可以把异步代码写的看起来像同步代码**
- 只要是一个 promiser 对象，那么我们就可以使用 `async/await` 来书写

```
async function fn() {
const res = new Promise(function (resolve, reject) {
ajax({
url: '第一个地址',
success (res) {
resolve(res)
}
})
})

// res 就可以得到请求的结果
const res2 = new Promise(function (resolve, reject) {
ajax({
url: '第二个地址',
data: { a: res.a, b: res.b },
success (res) {
resolve(res)
}
})
})

const res3 = new Promise(function (resolve, reject) {
ajax({
url: '第三个地址',
data: { a: res2.a, b: res2.b },
success (res) {
resolve(res)
}
})
})

// res3 就是我们要的结果
console.log(res3)
}
```

这样的异步代码写的就看起来像一个同步代码了

- 这样的异步代码与的就看起来像一个同步代码了