作者：前端小蜜蜂  
链接：https://zhuanlan.zhihu.com/p/288072115  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

我们在编写 JavaScript 代码时，经常会遇到逻辑判断复杂的情况。一般情况下，可以用 if/else 或 switch 来实现多个条件判断，但会出现一个问题：随着逻辑复杂度的增加，代码中的 if/else 和 switch 会越来越臃肿。本文将带你尝试写出更优雅的判断逻辑。

比如说下面这样一段代码：

const onButtonClick = (status) => {

if (status == 1) {

sendLog('processing')

jumpTo('IndexPage')

} else if (status == 2) {

sendLog('fail')

jumpTo('FailPage')

} else if (status == 3) {

sendLog('fail')

jumpTo('FailPage')

} else if (status == 4) {

sendLog('success')

jumpTo('SuccessPage')

} else if (status == 5) {

sendLog('cancel')

jumpTo('CancelPage')

} else {

sendLog('other')

jumpTo('Index')

}

}

你可以在代码中看到这个按钮的点击逻辑。根据活动状态的不同做两件事，发送日志埋点并跳转到相应的页面。很容易想到这段代码可以用 switch 重写如下：

const onButtonClick = (status) => {

switch (status) {

case 1:

sendLog('processing')

jumpTo('IndexPage')

break

case 2:

case 3:

sendLog('fail')

jumpTo('FailPage')

break

case 4:

sendLog('success')

jumpTo('SuccessPage')

break

case 5:

sendLog('cancel')

jumpTo('CancelPage')

break

default:

sendLog('other')

jumpTo('Index')

break

}

}

好吧，看起来比 if/else 层次结构更清晰一些，细心的读者可能也发现了一个小窍门：case 2 和 case 3 的逻辑一样时，可以把前面的逻辑处理代码省略，case 2 会自动执行与 case 3 的逻辑。

不过，还有一个更简单的写法：

const actions = {

'1': ['processing', 'IndexPage'],

'2': ['fail', 'FailPage'],

'3': ['fail', 'FailPage'],

'4': ['success', 'SuccessPage'],

'5': ['cancel', 'CancelPage'],

default: ['other', 'Index'],

}

const onButtonClick = (status) => {

let action = actions[status] || actions['default'],

logName = action[0],

pageName = action[1]

sendLog(logName)

jumpTo(pageName)

}

上面的代码看起来确实比较干净，这种方法的巧妙之处在于，它把判断条件作为对象的属性名，把处理逻辑作为对象的属性值。在点击按钮的时候，这种方法特别适用于单项条件判断的情况，即通过对象属性查找的方式进行逻辑判断。

这个方法很好，但是有没有其他的方法来编码呢？有的！

const actions = new Map([

[1, ['processing', 'IndexPage']],

[2, ['fail', 'FailPage']],

[3, ['fail', 'FailPage']],

[4, ['success', 'SuccessPage']],

[5, ['cancel', 'CancelPage']],

['default', ['other', 'Index']],

])

const onButtonClick = (status) => {

let action = actions.get(status) || actions.get('default')

sendLog(action[0])

jumpTo(action[1])

}

使用 Map 代替 Object 有很多优点，Map 对象和普通对象有的区别是：

* 一个对象通常有自己的原型，所以一个对象总是有一个“prototype”键
* 对象的键只能是一个字符串或符号，但 Map 的键可以是任何值
* 你可以通过使用 size 属性很容易得到 Map 中的键值对的数量，而一个对象中的键值对数量不能直接获取

现在我们来升级一下这个问题的难度。点击按钮时，不仅要判断状态，还要判断用户的身份。

const onButtonClick = (status, identity) => {

if (identity == 'guest') {

if (status == 1) {

//do sth

} else if (status == 2) {

//do sth

} else if (status == 3) {

//do sth

} else if (status == 4) {

//do sth

} else if (status == 5) {

//do sth

} else {

//do sth

}

} else if (identity == 'master') {

if (status == 1) {

//do sth

} else if (status == 2) {

//do sth

} else if (status == 3) {

//do sth

} else if (status == 4) {

//do sth

} else if (status == 5) {

//do sth

} else {

//do sth

}

}

}

从上面的例子中可以看到，当你的逻辑升级到双重判断的时候，你的判断力就会加倍，你的代码就会加倍。

如何才能让代码更干净利落呢？

这里有一个解决方案。

const actions = new Map([

['guest\_1', () => {}],

['guest\_2', () => {}],

['guest\_3', () => {}],

['guest\_4', () => {}],

['guest\_5', () => {}],

['master\_1', () => {}],

['master\_2', () => {}],

['master\_3', () => {}],

['master\_4', () => {}],

['master\_5', () => {}],

['default', () => {}],

])

const onButtonClick = (identity, status) => {

let action = actions.get(`${identity}\_${status}`) || actions.get('default')

action.call(this)

}

上述代码的核心逻辑是。将两个判断条件拼接成一个字符串作为 Map 的键，然后在查询时直接查询对应字符串的值。当然，我们也可以在这里把 Map 改成 Object。

const actions = {

guest\_1: () => {},

guest\_2: () => {},

//....

}

const onButtonClick = (identity, status) => {

let action = actions[`${identity}\_${status}`] || actions['default']

action.call(this)

}

如果读者觉得把查询拼成一个字符串有点尴尬，还有另一个解决办法，那就是用一个 Map 对象作为 key。

const actions = new Map([

[{ identity: 'guest', status: 1 }, () => {}],

[{ identity: 'guest', status: 2 }, () => {}],

//...

])

const onButtonClick = (identity, status) => {

let action = [...actions].filter(([key, value]) => key.identity == identity && key.status == status)

action.forEach(([key, value]) => value.call(this))

}

这里你也可以看到 Map 和普通对象的区别，其中 Map 可以用任何类型的数据作为键。现在让我们把它的难度再提高一点。如果对于 guest 身份来说，状态 1-4 的处理逻辑是一样的呢？

最坏的情况是这样的（代码大量重复）：

const actions = new Map([

[{ identity: 'guest', status: 1 }, () => {}],

[{ identity: 'guest', status: 2 }, () => {}],

[{ identity: 'guest', status: 3 }, () => {}],

[{ identity: 'guest', status: 4 }, () => {}],

[{ identity: 'guest', status: 5 }, () => {}],

//...

])

更好的方法是把处理逻辑函数分离出来：

const actions = () => {

const functionA = () => {}

const functionB = () => {}

return new Map([

[{ identity: 'guest', status: 1 }, functionA],

[{ identity: 'guest', status: 2 }, functionA],

[{ identity: 'guest', status: 3 }, functionA],

[{ identity: 'guest', status: 4 }, functionA],

[{ identity: 'guest', status: 5 }, functionB],

//...

])

}

const onButtonClick = (identity, status) => {

let action = [...actions()].filter(([key, value]) => key.identity == identity && key.status == status)

action.forEach(([key, value]) => value.call(this))

}

这对于日常需求来说已经足够了，但是说真的，函数 A 被引用了 4 次，还是有点烦人。

如果事情真的变得很复杂，比如身份有 3 种，状态有 10 种，你需要定义 30 个处理逻辑，其中很多处理逻辑都是一样的，这似乎让人无法接受。

而你可以这样做：

const actions = () => {

const functionA = () => {} // 逻辑处理 A

const functionB = () => {} // 逻辑处理 B

return new Map([

[/^guest\_[1-4]$/, functionA],

[/^guest\_5$/, functionB],

//...

])

}

const onButtonClick = (identity, status) => {

let action = [...actions()].filter(([key, value]) => key.test(`${identity}\_${status}`))

action.forEach(([key, value]) => value.call(this))

}

这时使用 Map 而不是 Object 的优势比较明显，因为可以用正则式作为键。

如果需求变成：所有的对 guest 操作都需要发送一个日志埋点，不同状态的 guest 可能有不同的逻辑处理，那么我们可以写成如下：

const actions = () => {

const functionA = () => {} // 逻辑处理 A

const functionB = () => {} // 逻辑处理 B

const functionC = () => {} // 发送日志 C

return new Map([

[/^guest\_[1-4]$/, functionA],

[/^guest\_5$/, functionB],

[/^guest\_.\*$/, functionC],

//...

])

}

const onButtonClick = (identity, status) => {

let action = [...actions()].filter(([key, value]) => key.test(`${identity}\_${status}`))

action.forEach(([key, value]) => value.call(this))

}

这样一来，公共逻辑和单个逻辑可以同时执行。

总结

本文讲到了八种 JS 逻辑判断的写法，包括：

1. if/else
2. switch
3. 单一判断：存储在 Object 中
4. 单一判断：存储在 Map 对象中
5. 多重判断：将条件串联成一个字符串，存储在 Object 中
6. 多重判断：将条件连成一个字符串，存储在 Map 对象中
7. 多重判断：把条件作为对象存储在 Map 中
8. 多重判断：把条件写成正则式存储在 Map 中