ECMAScript 6(二)

> 接上文

ES6的异步解决方案

在之前的ES5里面，我们总是会提到异步编程的概念，如定时器，如Ajax等。当我们在进行异步编程的时候，总会遇到一些问题。如我们经常所说的一个概念“同步等待，异步执行”

```
setTimeout(function(){
    console.log("hello");
},0);
console.log("biaogege");
```


上面的`setTimeout`是一个异步的，我们现在就使用这个函数来了解一下我们的异步编程

```
//模拟一个人去考驾照
function kemu1() {
    setTimeout(function () {
        //这就是你科目一得分
        let score = parseInt(Math.random() * 101);
        //我怎么样才能将这个score返回到外边去呢
        // return score;  这里是不能使用return的
    }, 3000);
}
```


在上面的代码当中我， 我们在函数的内部`kemu1()`里面写了一个定时器，3000毫秒以后就会执行一个随机数，问题的关键就在于，怎么样拿到这个随机数`score`。这里使用`return`关键是绝对不可以的，因为这个return如果返回，也仅仅只是返回给了回调函数，而没有返回给`kemu1`这个函数

为了解决上面的问题，我们之前在ES5里面，我们就有了一个新的思路【返回值不能出来，那我就进去呗】。所以我们就引入了**回调函数**

```
function kemu1(callBack) {
    setTimeout(function () {
        //这就是你科目一得分
        let score = parseInt(Math.random() * 101);
        //score出不来，那我就进去
        if (typeof callBack == "function") {
            callBack(score);
        }
    }, 3000);
}

kemu1(score => {
    console.log(`你的得分是${score}`);
});
```


我们现在使用了回调函数以后，我们就可以在这个函数的外部再去使用`score`的值了，这就是回调函数最经典的应用

在上面的应用里面，我们只有一个回调函数，现在我们完全去模似驾照的考试

```
function kemu1(successCallBack, failCallBack) {
    setTimeout(() => {
        let score = parseInt(Math.random() * 10) * 10;      //随机产生一个分数
        //科目1是90分及格
        if (score >= 50) {
            // 说明考试及格了
            if (typeof successCallBack == "function") {
                successCallBack({
                    msg: "考试通过",
                    value: score
                });
            }
        }
        else {
            //说明科目一考试挂了
            if (typeof failCallBack == "function") {
                failCallBack({
                    msg: "考试未通过",
                    value: score
                })
            }
        }
    }, 3000);
}
//现在调用上面的科目一的考试
kemu1(result => {
    //这个回调函数相当于successCallBack,也就是成功以后的回调
    console.log(`你的考试分数是${result.value},请准备下一场科目二的考试`);
}, result => {
    //这个回调相当于failCallBack,失败以后的回调
    console.log(`你的考试分数是${result.value}，考试没有通过,请在15天以后重新预约考试`);
})
```


在上面的代码当中，我们有两个回调过程，为什么会出现回调是因为`setTimeout`这个方法是异步的方法，它不会立即就把结果给我，我需要等待它产生结果，正是因为这种情况，所以我不能使用`return`返回，就只能使用回调

所以在以前的ES5里面，我们如果要解决异步编程，就只能是使用回调函数

在ES6里面，为了更好的解决异步编程，所以我们出了一个新的对象，这个对象叫`Promise`。这个英文的意思是**承诺**的意思

`Promise`承诺的内部是有三个状态的

1. `pending`等待状态
2. `resolve`成功状态
3. `reject`失败状态

情景一：

邓娜同学下午找标哥请求，说：“标哥，我下午要参回科目一考试，我要请一下午的假！望批准！”

标哥说：“没问题，考试完了以后，把结果告诉我就可以了”

邓娜开心的回道：“OK！好的标哥，没问题，我一考完就知道结果了，然后立刻告诉你！”

分析：

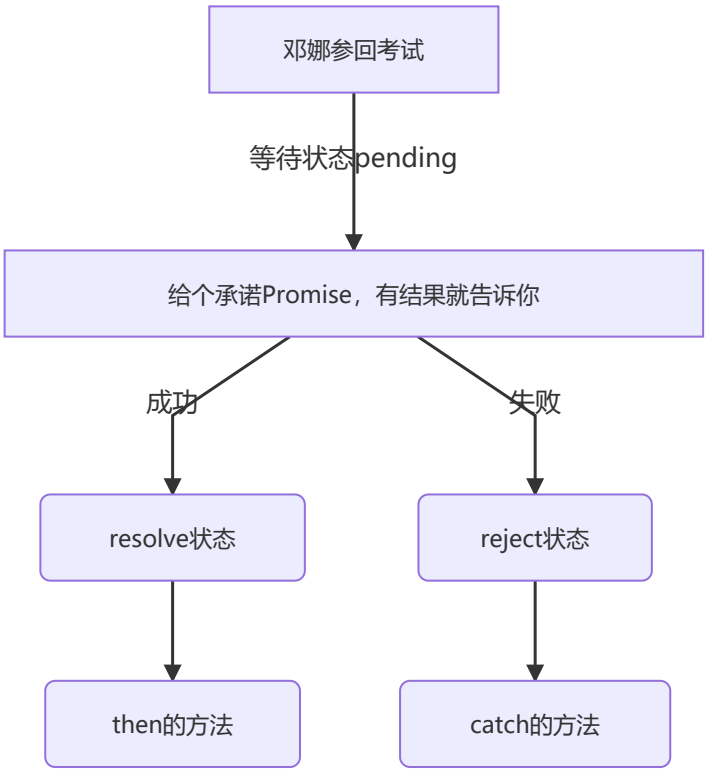
从邓娜的最后一句话开始，她就给了标哥一个承诺`Promise`

标哥得到邓娜的的承诺`Promise`以后，它是处于等待状态 `pending`状态（因为邓娜还没有考试完成）

最终这个等待状态一定会有结果（要么成功【邓娜考试通过`resolve`】，要么失败【邓娜考试挂了`reject`】）


```
function kemu1() {
    //邓娜去参回科目一的考试，你考完以后必须有个结果
    //承诺的内部有三个状态，等待状态，成功状态，失败状态
    //从邓娜参加科目一考试：我要等它给我结果
    //邓娜考完一瞬间，就已经有结果，这个结果可能考过了，也可能没考过
    //但是无论有没有考试通过，都要给我结果
    //在这里，邓娜要给标哥一个承诺
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let score = parseInt(Math.random() * 10) * 10; //随机产生一个分数
            if (score > 50) {
                //考试通过
                let obj = {
                    msg: "考试通过",
                    value: score
                }
                resolve(obj); //将原来的承诺中的等待状态变成 成功状态
            } else {
                //考试不通过
                let obj = {
                    msg: "考试未通过",
                    value: score
                }
                reject(obj); //将原来承诺中的等待状态变成失败状态
            }
        }, 3000);
    });
    return p;
}

//这样标哥就得到了kemu1()这个函数返回的一个承诺
//但是这个承诺是没有结果的
let result = kemu1(); //这个result就是一个Promise，默认是pending状态等待
//只有成功以后才会继续,then执行的就是resolve的结果
result.then(obj => {
    console.log(`恭喜邓娜，科目一考试通过，你的得分是${obj.value}`);
}).catch(obj => {
    //catch只有rejct会执行
    console.log(`邓娜哭丧着个脸对标哥说：“标哥，我考挂了，考了${obj.value}分”`);
});
```

在上面的代码当中`then`执行是`resolve`以后的结果,`catch`执行的则是`reject`的结果



> 由`pending`到`resolve`由代表成功，由`pending`到`reject`则代表失败，无论是转向了成功`resolve`还是转向了失败`reject`，这个状态都是不可逆转了

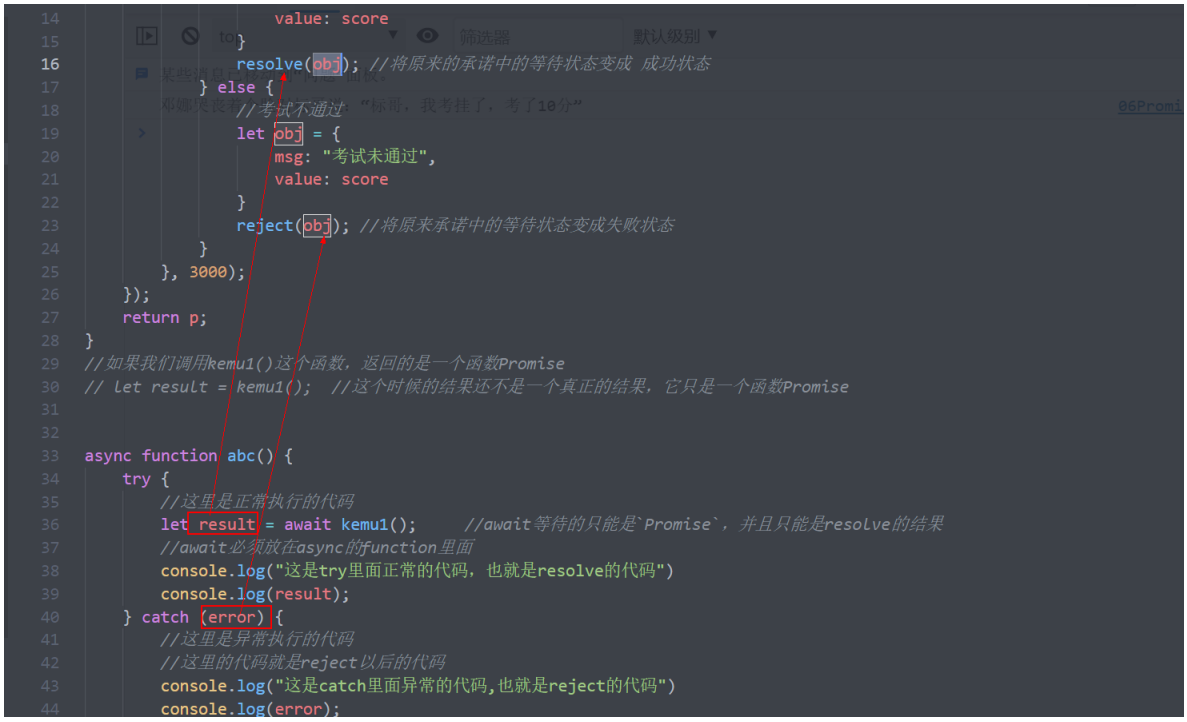
在上面的代码里面，我们使用`ES6`里面的`Promise`完成了异步编程，但是我们仍然在`then`与`catch`里面看到了回调函数，这样做不跟`ES5`没有什么区别吗?

`Promise`写成上面的样式，只是我们的第一步，我们还要再次借助于`ES7`里面的两个关键字`await/async`来完成，进一步简化异步编程的概念

下面的代码是典型的异步转同步

softeem · 杨标

```
async function abc() {
    try {
        //这里是正常执行的代码
        let result = await kemu1();      //await等待的只能是`Promise`，并且只能是resolve的结果
        //await必须放在async的function里面
        console.log("这是try里面正常的代码，也就是resolve的代码")
        console.log(result);
    } catch (error) {
        //这里是异常执行的代码
        //这里的代码就是reject以后的代码
        console.log("这是catch里面异常的代码,也就是reject的代码")
        console.log(error);
    }
}

abc();
```



```
                value: score
            }
            resolve(obj); //将原来的承诺中的等待状态变成 成功状态
        } else {
            //考试不通过
            let obj = {
                msg: "考试未通过",
                value: score
            }
            reject(obj); //将原来承诺中的等待状态变成失败状态
        }
    }, 3000);
});
return p;
}
//如果我们调用kemu1()这个函数，返回的是一个函数Promise
// let result = kemu1();  //这个时候的结果还不是一个真正的结果，它只是一个函数Promise


async function abc() {
    try {
        //这里是正常执行的代码
        let result = await kemu1();      //await等待的只能是`Promise`，并且只能是resolve的结果
        //await必须放在async的function里面
        console.log("这是try里面正常的代码，也就是resolve的代码")
        console.log(result);
    } catch (error) {
        //这里是异常执行的代码
        //这里的代码就是reject以后的代码
        console.log("这是catch里面异常的代码,也就是reject的代码")
        console.log(error);
```

在上面的代码当中，我们就使用了`await`的关键字去完成，它后面等等的一定是成功的`resolve`的结果，而失败的结果就会在`catch`里面

通过上面的代码，我们可以看到，我们已经将`setTimeout`这个里面的异步代码转换成了同步的代码

上面就是ES6里面，异步代码最经典的一种解决方法，使用了`async/await`去执行。

通过上面的例子，我们已经可以总结以下几个重点

1. `Promise`一定有三个状态
2. 成功的状态是`resolve`，失败的状态是`reject`，等待的状态是`pending`，必须这三个状态不可逆转，一但发生改变就不可能再更改
3. `await`必须与`async`同时使用
4. `await`等待的只能是`Promise`，不能是其它的东西，并且只能够等到的`resolve`的结果，失败的结果在`catch`里面
5. 一定要熟悉`Promise`的写法

```
let p = new Promise((resolve,reject)=>{
    //先写好了再说
});
return p;
```


ES6中Promise与Await的结合

我们现在使用`Promise`与`Await`去重新封装我们的`Ajax`方法试一下

```
var AjaxHelper = {
    get(url) {
        let p = new Promise((resolve, reject) => {
            let xhr = new XMLHttpRequest();
            xhr.open("get", url, true);
            xhr.onreadystatechange = function () {
                if (xhr.readyState == 4 && xhr.status == 200) {
                    //成功,这是异步代码，我怎么办呢？
                    resolve(xhr.responseText);
```

```
                }
            }
            //请求超时
            xhr.ontimeout = function (event) {
                reject(event);
            }
            //请求错误
            xhr.onerror = function (event) {
                reject(event);
            }
            xhr.send();
        });
        return p;
    }
}

let url = "http://www.softeem.xin:8888/public/musicData/musicData.json";

async function getMusicData() {
    //let result  = AjaxHelper.get(url);  //现在返回的是一个Promise
    //而Promise又可以直接使用await来等待结果
    try {
        let result = await AjaxHelper.get(url); //但是await又要与async结合起来一起使用
        console.log("请求成功");
        console.log(result);
    } catch (error) {
        console.log("请求失败");
        console.log(error);
    }
}
```

我们之前的Ajax封装使用的是回调函数，现在则使用了`Promise`和`await/async`来解决了

Promise的多次等待

其实在Promise的内部是可以多次等待的，我们只需要再内部再次使用Promise就可以了


```
function kemu1() {
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let score = parseInt(Math.random() * 101);
            if (score > 50) {
                //考试通过
                let obj = {
                    msg: "考试通过",
                    value: score
                }
                resolve(obj);           //由pending状态转向resolve成功状态
            }
            else {
                //考试不通过
                let obj = {
                    msg: "考试不通过",
                    value: score
                }
                reject(obj);            //由pending状态转向reject失败状态
```

```
            }
        }, 3000);
    })

    return p;
}
//考完科目一，就要考科目二
let result = kemu1();

result.then(obj => {
    //如果执行的是then必须是resolve的结果
    console.log(`你的科目一的考试成功是${obj.value}，考试通过了`);
    //邓娜的科目一考试通过以后，肯定要继续再去奋斗科目二，考试
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let obj = {
                value: 90, msg: "科目二考试通过"
            }
            resolve(obj);
        }, 3000);
    });
    return p;
}).then(obj => {
    console.log(`这是第二个then`);
    //这里执行科目一里面返回的Promise
    console.log(obj);
}).catch(obj => {
    //如果执行的是catch必然是reject的结果
    console.log(`你执行的是catch,你考试没有通过，你的分数是${obj.value}`);
})
```

在上面的代码当中，我们看到了代码中使用了两次`then`，因为有两个异步要进行。这就相当当于一个承诺结束了以后，开始了另一个承诺

在上面的代码里面，我们其实是可以把代码做进一步的简化过程


```
function kemu1() {
    let p = new Promise((resolve, reject) => {
        //这一个Promise默认是一个pending的等待状态
        setTimeout(() => {
            let score = parseInt(Math.random() * 100);
            if (score > 50) {//考试通过
                let obj = {
                    msg: "科目一考试通过", value: score
                }
                resolve(obj);           //由pending状态转向resolve成功状态
            }
            else { //考试不通过
                let obj = {
                    msg: "科目一考试不通过", value: score
                }
                reject(obj);            //由pending状态转向reject失败状态
            }
        }, 3000);
    });
    return p;
```

```
}

function kemu2() {
    let p = new Promise((resolve, reject) => {
        //这一个Promise默认是一个pending的等待状态
        setTimeout(() => {
            let score = parseInt(Math.random() * 100);
            if (score > 50) {//考试通过
                let obj = {
                    msg: "科目二考试通过", value: score
                }
                resolve(obj);
            }
            else { //考试不通过
                let obj = {
                    msg: "科目二考试不通过", value: score
                }
                reject(obj);
            }
        }, 3000);
    });
    return p;
}

//现在开始调用
// let result =kemu1();
// result.then();

kemu1().then(obj => {
    console.log(`科目一的考试通过了，你可以继续参回科目二考试，你的考试分数是${obj.value}`);
    let p = kemu2();          //科目2返回的也是一个Promise，所以它也可以直接返回
    return p;
}).then(obj => {
    console.log(`科目二的考试通过，你的考试成功为${obj.value}`);
    //科目二成功以后，就可以继续调科目三的方法呗
}).catch(obj => {
    //如果上面的任何一科考试不通过，都会进入到catch，同时要注意，catch只能有一次
    console.log(`${obj.msg},你的考试成绩为 ${obj.value}`);
});
```

上面的代码就是将之前的多个Promise放在了不同的方法里面，这样看起来更清晰一些。同时要注意一点，`catch`只可以有一次，而`then`则可以有多个

同理上面的代码也可以转换成`async/await`的写法


```
//科目一考试
function kemu1() {
    let p = new Promise((resolve, reject) => {
        //这一个Promise默认是一个pending的等待状态
        setTimeout(() => {
            let score = parseInt(Math.random() * 100);
            if (score > 50) {//考试通过
                let obj = {
                    msg: "科目一考试通过", value: score
                }
                resolve(obj);           //由pending状态转向resolve成功状态
```

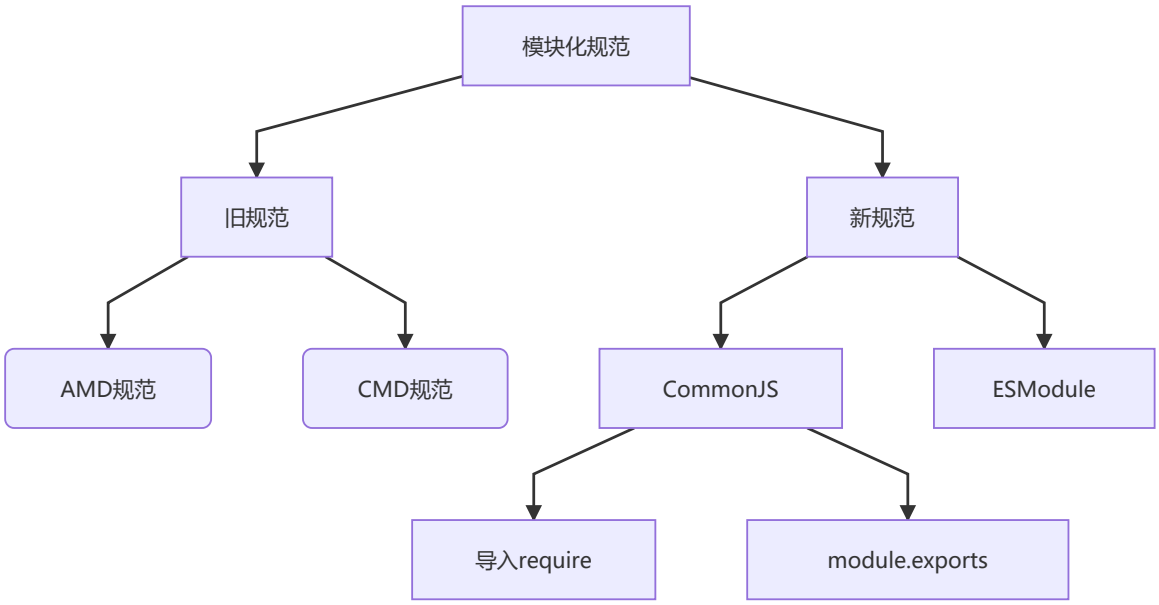
```
            }
            else { //考试不通过
                let obj = {
                    msg: "科目一考试不通过", value: score
                }
                reject(obj);            //由pending状态转向reject失败状态
            }
        }, 3000);
    });
    return p;
}
//科目二考试
function kemu2() {
    let p = new Promise((resolve, reject) => {
        //这一个Promise默认是一个pending的等待状态
        setTimeout(() => {
            let score = parseInt(Math.random() * 100);
            if (score > 50) {//考试通过
                let obj = {
                    msg: "科目二考试通过", value: score
                }
                resolve(obj);
            }
            else { //考试不通过
                let obj = {
                    msg: "科目二考试不通过", value: score
                }
                reject(obj);
            }
        }, 3000);
    });
    return p;
}

async function abc() {
    try {
        let result1 = await kemu1();    //科目一考试
        console.log(`你科目一考试的成绩为${result1.value}`);
        let result2 = await kemu2();    //科目二考试
        console.log(`你科目二的考试成绩为${result2.value}`);
    } catch (error) {
        console.log("我是catch里面的代码");
        console.log(`${error.msg}，你的考试成绩为${error.value}`);
    }
}

abc();
```

到目前为止，`Promise+Await`是解决JS异步编程最好的方法，无论是多少个异步，我们都可以通过它来完成。后期我们的`Express`框架当中会大量使用到Promise的使用方法

模块化

> 模块化指指的是JS文件之间的相互引用关系。在很久很久以前，JS也是有模块化的，它有两个模块化的规范（AMD/CMD规范），这两种规范功能非常少，并且实现起来也不方便，所以后期的时候慢慢的被另一种规范取代掉了，这个新规范就叫`CommonJS`规范

> 同时ES6出来以后也自带一个模块化的规范叫`ESModule`,但是这个规范目前在`Node.js`它不支持。Nodejs只支持`CommonJS`的规范，如果想让`NodeJS`支持`ESModule`的规范则需要借用于第三方平台如`webpack`或`glup`等



NodeJs是没有DOM的，所以它就是没有`<script>`这个标签。它现在没有这个标签以后JS文件之间怎么相互引用呢?

NodeJS里面虽然没有了`script`标签来进行引入，它是一套特殊的文件引入机制（`CommonJS`）以实现模块化，它使用的关键字是`module.exports`或`exports`来完成

`CommonJS`的模块化规范主要是为了`NodeJs`存在的规范，它只能在`NodeJs`平台下面使用

`ESModule`是ES6新出的模块化规范，它不能在`NodeJS`平台下面使用，可以借用第三方工具来使用（目前的浏览器则是直接支持）

模块导入

`CommonJS`使用的是`require()`去导入另一份JS文件，相当于把这个JS文件拿到了另一个JS文件里面去

Person.js

```
console.log("大家好，这是Perosn.js文件");
```


Student.js

```
require("./Person.js");
//这样就相当于把Person.js文件拿到了Student.js里面来
console.log("大家好，我是Student.js文件");
```


当我们使用`nodejs`去执行`Student.js`里面的代码的时候，`Person.js`祝导入进来同时执行了

在使用`require()`方法去执行导入的时候，它会形成一个缓存，如果多次导入它不会多次执行，所以当我们把`Student.js`的代码换成下面的代码

```
require("./Person.js");
//第一次是真正的导入，后期再导入就是从缓存里面拿值了
//这样就相当于把Person.js文件拿到了Student.js里面来
require("./Person.js");
require("./Person.js");
require("./Person.js");
require("./Person.js");
require("./Person.js");
console.log("大家好，我是Student.js文件");
```

softeem · 杨标

在上面的代码当中，我们在`Student.js`里面多次的导入了`Person.js`这个文件

只要通过`require()`导入一次以后就会形成缓存，后面就直接从这个缓存里面再拿值，不会再执行导入过程了，所以后面的导入其实是没有任何效果的

只有在第一次导入`require()`的时候是真正的导入，其它的时候都是从**缓存**里面直接拿值，所以在上面的代码运行以后的结果就是只会打印一次`大家好，这是Perosn.js文件`这个结果

既然有缓存，则必然会有一个清除缓存的方法

Person.js

```
console.log("大家好，这是Perosn.js文件");
//如果每次导入完成以后要清除缓存，怎么办呢？
delete require.cache[module.filename];
```

softeem · 杨标

当我们在`Person.js`的文件的最后面添加上面的代码以后，代表每次导入完成以后都从缓存当中删除这个模块的名称。这个时候，当我们在`Student.js`时面多次导入`Person.js`这个时候的文件，我们会看到在控制台就打印了多次【因为没有形成缓存，所以每次都相当于重新导入了】

当我们使用`require()`去导入JS文件的时候，里面的代码会执行，**但是在这个文件里面的变量，方法，对象等这些都不会拿过来**，这是受到了`CommonJS`规范的限制

Person.js

```
console.log("这是Person.js文件");

let a = "邓娜";
```

softeem · 杨标

Student.js

```
require("./Person.js");

console.log("这是Student.js");
//请问，我能不能在这里拿到变量a
console.log(a);      //这里就会报错，提示变量a没有被定义
```

softeem · 杨标

如果想拿到这些变量，方法或者对象 则需要被导入的文件(Person.js)**主动的去把这些东西给导出来**

模块导出

模块导入可以完美的解决上面的问题，模块导入使用的是对象，不是方法

在`NodeJS`的模块化规范`CommonJS`里面，每个文件内部都自带两个对象，分别是`module.exports`和`exports`

`module.exports`是专门负责导出，它可以导出任何东西，**默认是一个空的对象`{}`**

Person.js

```
console.log("这是Person.js文件");
let a = "邓娜";
module.exports = a;      //将变量a直接导出
```


Student.js

```
let a = require("./Student.js");
console.log("这是Student.js");
console.log(a);      //这样就可以了
```


在上面的代码里面，我们可以看到这个`Student.js`在导入`Person.js`的时候，顺便拿到了它导出的变量a,这样再去使用变量a就没有错误了！！！！

如果我们需要导出多个变量或者方法呢，或者对象呢？这个时候又应该怎么办呢？

这个时候我们可以借用于对象来完成

Person.js

```
console.log("我是Person.js");

let a = "邓娜";
let b = "蒋雨晴";

//现在怎么同时导出a与b呢
//module.exports 它是一个对象，对象就会有属性
let obj = {
    a: a,
    b: b
}

//现在导出的是一个对象了
module.exports = obj;
```


Student.js

```
let obj = require("./Person.js");
//这个时候obj就是模块Person所导了的对象了
console.log(obj.a);
console.log(obj.b);
console.log("这是Student.js");
```


上面的代码我们就导出了2个变量

上面的代码，我们完全可以使用解构的方式去简写，这样会更加方便一点

Person.js

```
console.log("我是Person.js");

let a = "邓娜";
let b = "蒋雨晴";
const sayHello = () => {
    console.log("大家好啊，我是你们的小可爱呀");
}

//现在怎么同时导出a与b呢
//现在导出的是一个对象了,使用到了对象里面的解构赋值
module.exports = {
    a, b
};
```

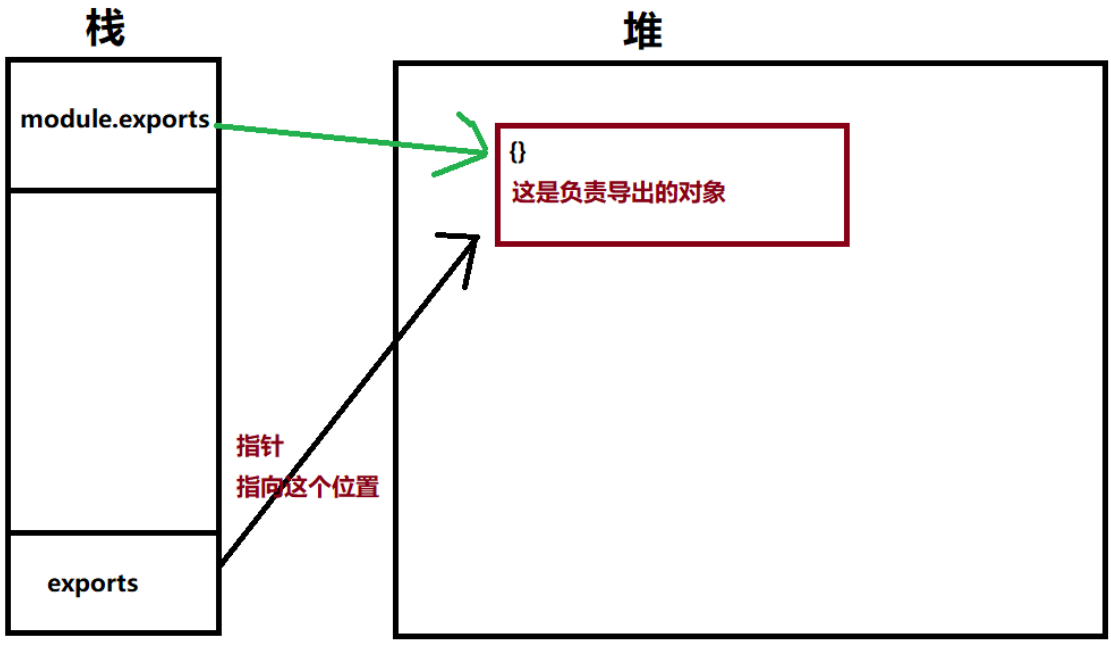

Student.js

```
// 使用解构取值
let { a, b ,sayHello}  = require("./Person.js");
console.log(a);
console.log(b);
sayHello();

console.log("这是Student.js");
```


在`Person.js`里面我们使用了解构的赋值，而在`Student.js`里面，我们使用了解构的取值，这样操作会更加简单一点

exports与module.exports的区别

在NodeJS平台下面的`CommonJS`规范里面，每个文件当中都会有一个`module.exports`，**这个对象主要是负责导出的**，同时还有一个`exports`，**它是一个指针，指向了module.exports最初的对象地址**，也就是这个最初的`{}`



最初的时候，它们两个人指向的是同一个内存地址，这个内存地址是一个对象`{}`

```
console.log(module.exports == exports);     //true
```

所以我们在导出多个数据的时候，也可以使用下面的方式

```
let a = "邓娜";
let b = "蒋雨晴";
let c = "易兰";

//现在我们可以直接在这个导出对象上面添加属性
module.exports.a = a;
module.exports.b = b;
exports.c = c;

console.log(module.exports);
console.log(exports);
console.log(module.exports == exports);
```


上面的结果执行如下

```
{ a: '邓娜', b: '蒋雨晴', c: '易兰' }
{ a: '邓娜', b: '蒋雨晴', c: '易兰' }
true
```


通过上面的例子，我们已经可以证实，`module.exports`与`exports`最初的时候是相同的，因为`exports`默认是指向了`module.exports`的

场景一

a.js

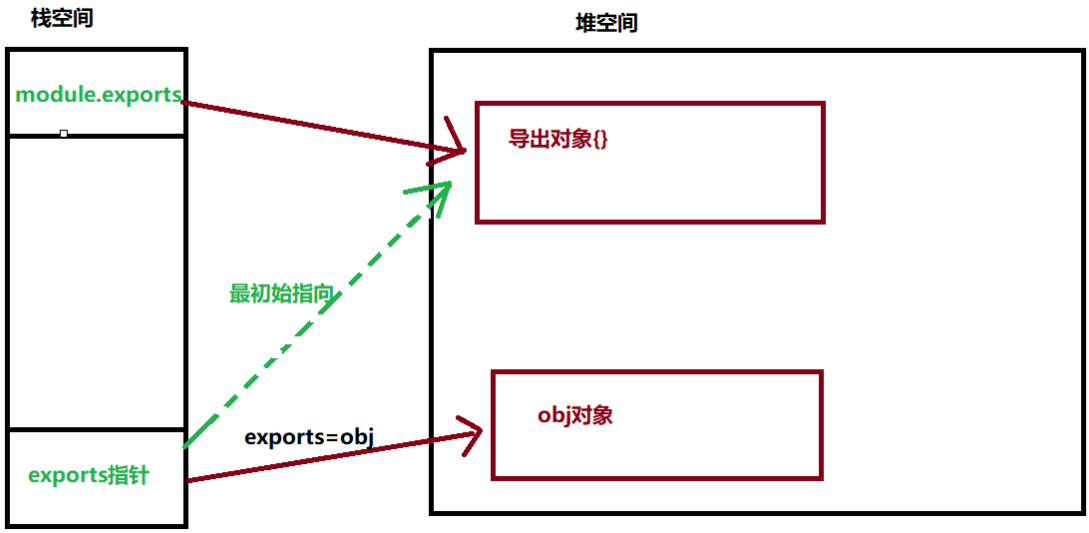
```
let userName = "邓娜";
let age = 18;
// 解构赋值
let obj = {
    userName,age
}
exports = obj;

/**
 * 真正负责导出的是module.exports
 * 现在exports已经不再指向module.exports
 */

/*

console.log(module.exports == exports);
console.log(module.exports);
console.log(exports);
*/
```


b.js

```
let obj = require("./a.js");
console.log(obj);
```

问题：请问，在`b.js`这个文件里面，能够成功的拿到`a.js`所导出的对象`obj`

分析：要解决上面的问题，我们就画内存图



`exports`在最初的时候是指向了`module.exports`的，但是在执行了`exports = obj`这个操作以后，它就不指向`module.exports`了，而是指向了`obj`这个对象。**但是真正负责导出的还是`module.exports`**，所以这个时候我们的`b.js`是拿不到`exports`是面面的`userName,age`的，它还是找的`module.exports`下面的空对象

场景二

a.js

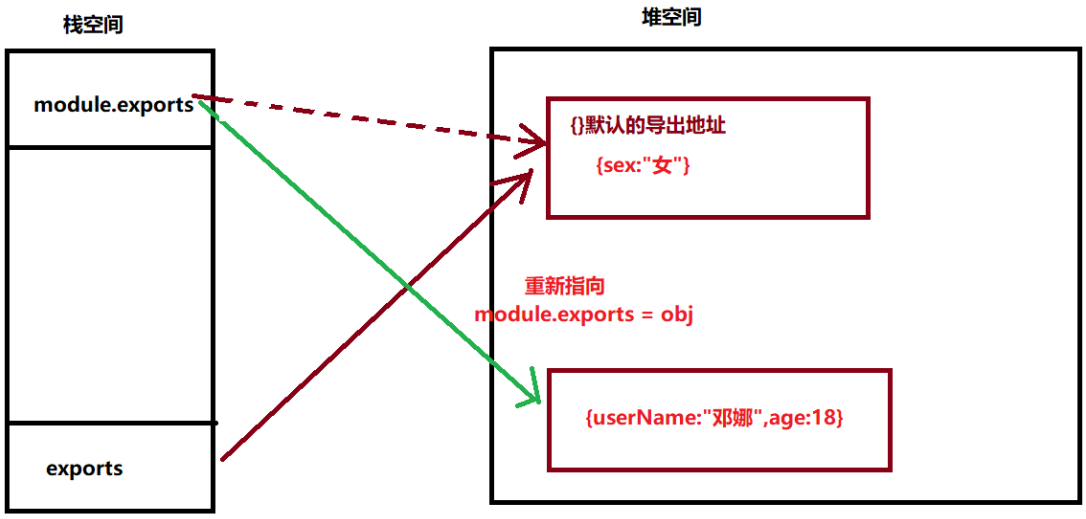
```
let userName = "邓娜";
let age = 18;
let obj = {
    userName, age
};

module.exports = obj;

exports.sex = "女";
```


b.js

```
let obj = require("./a.js");
console.log(obj);
//在这个obj上面，是否会有属性sex
```


问题：能够在`b.js`这个文件里的导入`a.js`的时候，这个`obj`对象上面是否有属性`sex`

分析：根据上面的过程，我们来画内存图



现在根据上面画的内存图，我们已经知道结果了，真正负责导出的只有`module.exports`,你们最初的时候是指向同一个内存的地址的，但是后来经过了`module.exports = obj`这个赋值以后，`module.exports`就不再指向原来的内存地址，而是指向了堆里了的一个新的对象内存地址，但是`exports`则还是指向原来的旧的对象的内存地址。所以最终`b.js`是没有`sex`这个属性的

总结： 所以通过上面的两个场景，我们都可以发现， 我们不能随便的去改变`exports`或`module.exports`在栈里面的地址，除非你只使用`module.exports`，否则，你应该是深入到堆里面去改变

正是因为有上面的情况，所以我们在导入多个值的时候应该是通过下面的方式来进行

a1.js

softeem · 杨标

```
let userName = "邓娜";
let age = 18;

module.exports.userName = userName;
exports.age = age;
//最初始的时候module.exports与exports都指向同一个对象，所以你无论是通过哪一个找到这个导入对象，只要在这个对象上面进行扩展，都没有问题

exports.sayHello = function () {
    console.log("大家好，我叫标哥");
}
```

b1.js

softeem · 杨标

```
let { userName, age, sayHello } = require("./a1.js");
console.log(userName,age);
sayHello();
```

但是有些时候我们也会主动的去改变`module.exports`,例如我只想导入**一个**对象，或者我只想使用`module.exports`这个就可以直接 使用`module.exports = obj`像这种方式完成【这种方式相当于放弃使用`exports`】

但是我要说明一点就是，这个东西既然存在了，必然会有人使用，你在工作当中要根据实际的情况来使用它们，因为有人喜欢使用`exports`，也人喜欢使用`module.exports`