JavaScript极速入门

同步与异步
promise/async/await

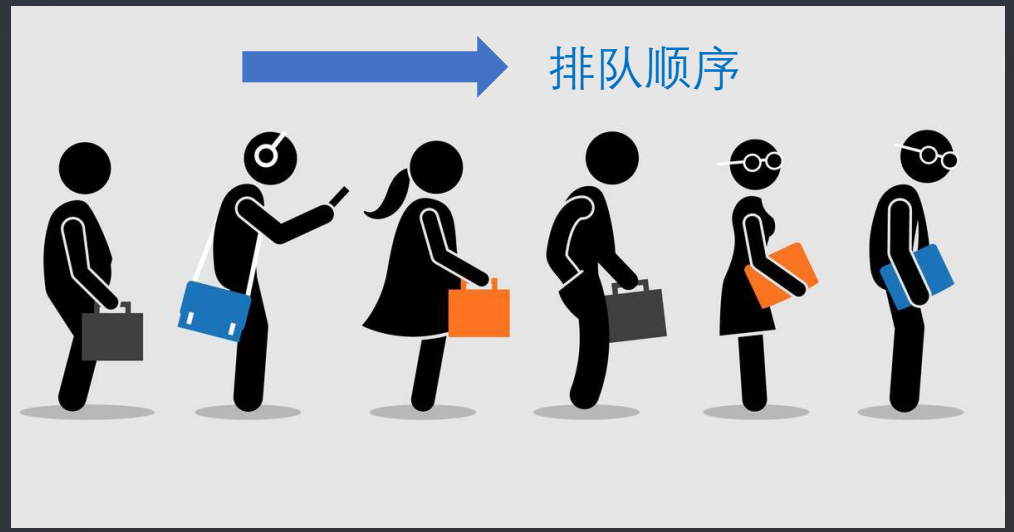
本节课提纲

1. 同步与异步概念
2. ajax （jquery）
3. promise
4. async / await

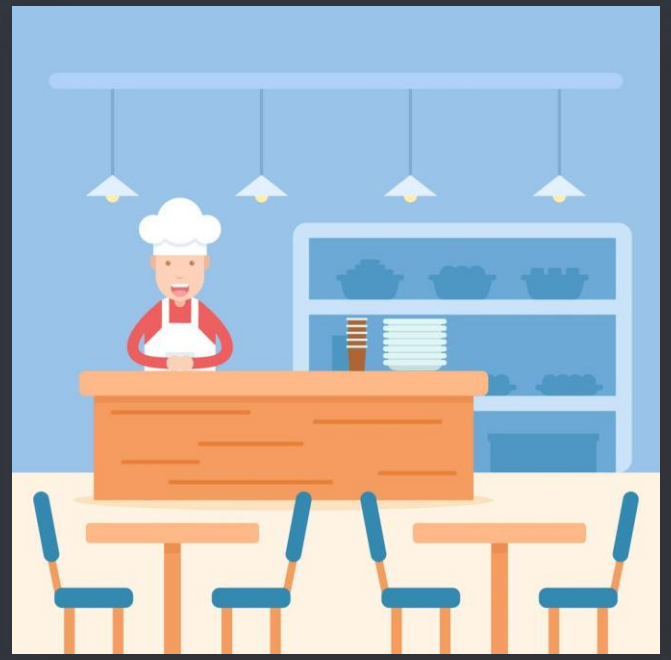
同步

同步就是后一个任务等待前一个任务执行完毕后，再执行，执行顺序和任务的排列顺序一致

按照排队的顺序，依次打饭，只有前面一个人打完饭后，才轮到后面一个人



食堂排队打饭



同步

同步就是后一个任务等待前一个任务执行完毕后，再执行，执行顺序和任务的排列顺序一致

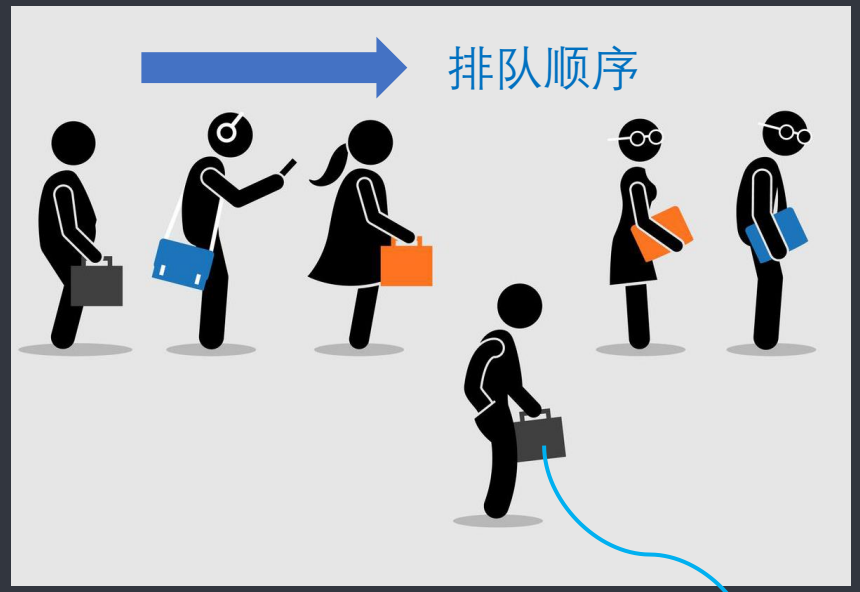
```
console.log('a')
console.log('b')
console.log('c')
```

异步

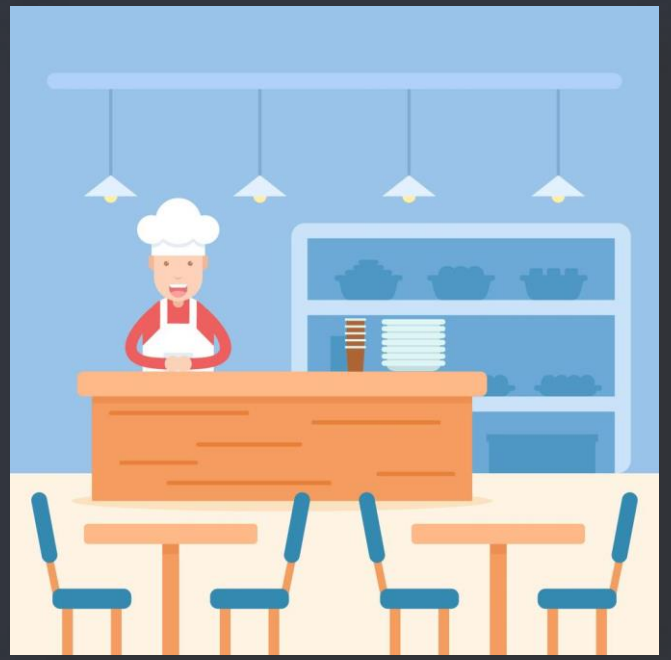
异步是非阻塞的，异步逻辑与主逻辑相互独立，主逻辑不需要等待异步逻辑完成，而是**可以立刻继续**下去

有同学忘记带饭卡了，怎么办？如果一直等着饭卡拿来，那是不是很花时间，后面同学就不能打饭了！

所以呢，我让出来，**让后面的人先打饭**，等我的饭卡到了，我再打饭



食堂排队打饭



异步

异步是非阻塞的，异步逻辑与主逻辑相互独立，主逻辑不需要等待异步逻辑完成，而是**可以立刻继续**下去

```
console.log('a')
console.log('b')
console.log('c')

setTimeout(() => { console.log('我饭卡拿到了！') }, 2000) //异步操作

console.log('d')
console.log('e')
```

网络异步请求

一般用jquery中ajax去做网络请求

```
console.log(1)

$.ajax({
    type: "get",
    url: "https://mockapi.eolinker.com/test",
    success: function (response) {
        console.log(response)
    }
});

console.log(2)
```

promise

异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理且更强大

有三个状态:
1、**pending** [待定] 初始状态
2、**fulfilled** [实现] 操作成功
3、**rejected** [拒绝] 操作失败

```
var promise = new Promise(传一个函数);
```

```
var promise = new Promise(function (resolve, reject) {
    if (/* 异步操作成功 */) {
        resolve(value);
    } else {
        reject(error);
    }
});
```

promise

当promise状态发生改变，就会触发**then()**里的响应函数处理后续步骤

状态改变，只有两种可能：
从**pending变为fulfilled**
从**pending变为rejected**

promise.**then(函数)**

```
var promise = new Promise(function (resolve, reject) {
    if (false) {
        resolve('success');
    } else {
        reject('fail');
    }
});

promise.then(res => {
    console.log(res) // 成功 resolve('success')
}).catch(err => {
    console.log(err) // 失败 reject('fail');
});
```

Promise.all

Promise.all可以将多个Promise实例包装成一个新的Promise实例。同时，成功和失败的返回值是不同的，**成功**的时候返回的是**一个结果数组**，而**失败**的时候则返回**最先被reject失败状态的值**

```
let p1 = new Promise((resolve, reject) => {
    resolve('成功了')
})

let p2 = new Promise((resolve, reject) => {
    resolve('success')
})

let p3 = Promise.reject('失败')

Promise.all([p1, p2]).then((result) => {
    console.log(result)                 //['成功了', 'success']
}).catch((error) => {
    console.log(error)
})

Promise.all([p1, p3, p2]).then((result) => {
    console.log(result)
}).catch((error) => {
    console.log(error)      // 失败了，打出 '失败'
})
```

Promise.race

顾名思义，Promse.race就是赛跑的意思，意思就是说，Promise.race([p1, p2, p3])里面哪个**结果获得的快**，就返回那个结果，不管结果本身是成功状态还是失败状态

```
let p1 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('success')
    }, 1000)
})

let p2 = new Promise((resolve, reject) => {
    setTimeout(() => {
        reject('failed')
    }, 500)
})

Promise.race([p1, p2]).then((result) => {
    console.log(result)
}).catch((error) => {
    console.log(error)  // 打开的是 'failed'
})
```

Async / await

也是用来处理异步的，其实是Generator 函数的改进，背后原理就是**promise**

情况1:

```
async function f1() {
    return "abc";
    // 自动包装成promise对象
    // 与下面两种等价
    // return Promise.resolve('abc');
    // return new Promise((resolve) => { resolve('abc') });
}
```

Async / await

也是用来处理异步的，其实是Generator 函数的改进，背后原理就是**promise**

情况2 – 与await结合：

```
async function f3() {
    return 'f3';
}

function f4() {
    return 'f4';
}

async function f5() {
    var c = await f3(); // await是异步函数
    var d = await f4(); // await是正常函数
    //看到await 会阻塞后面的代码，等主程序执行完，再回来执行
}
```

Async / await

```
async function f3() {
    return Promise.reject('sss');
}

async function f5() {
    try {
        var c = await f3();
    } catch (e) {
        console.log(e)
    }
    //如果await 是reject，后面代码就不会执行，要加上try catch才会执行
}
```