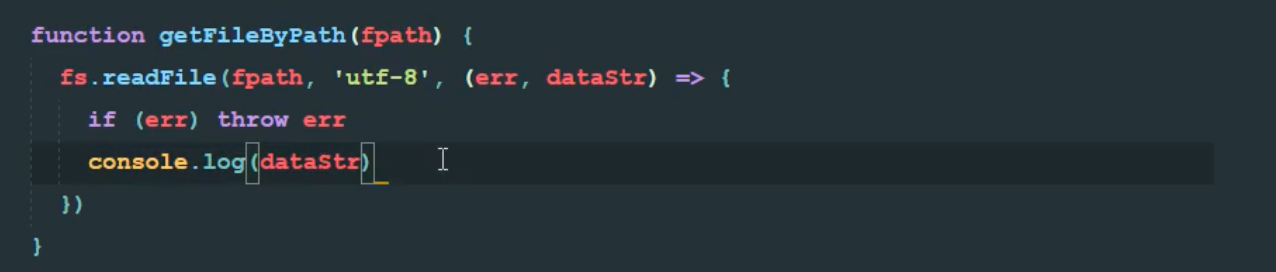
# Promise



定义一个方法: 从需求思考,路径就是不能写死

那么就给他一个不要写死的参数

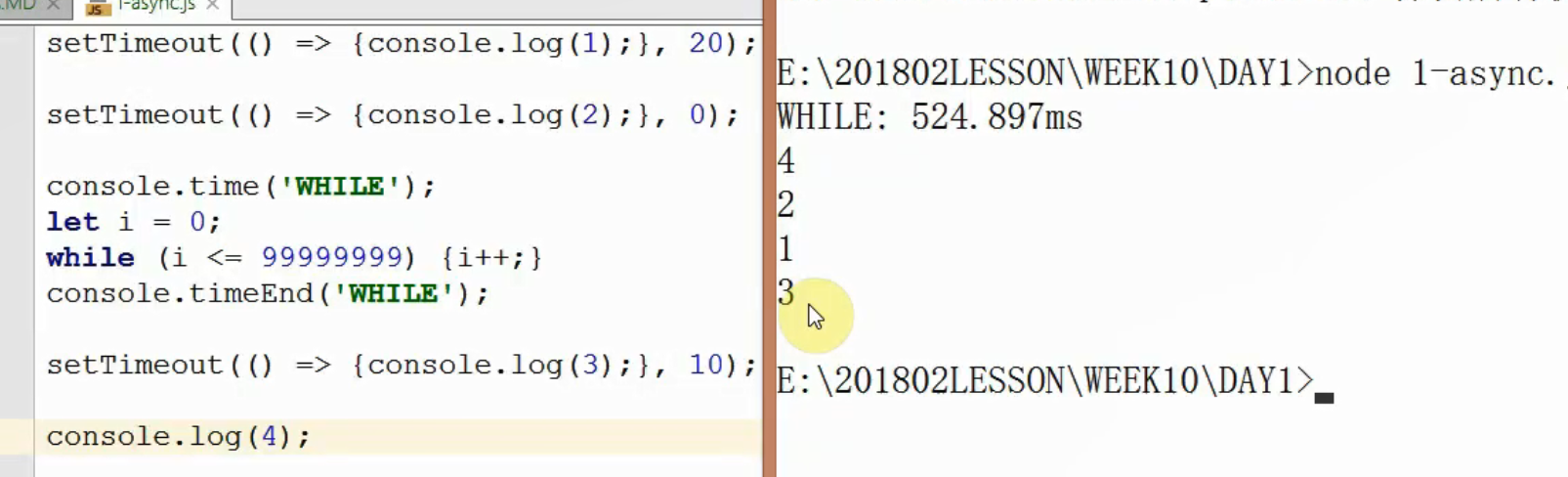


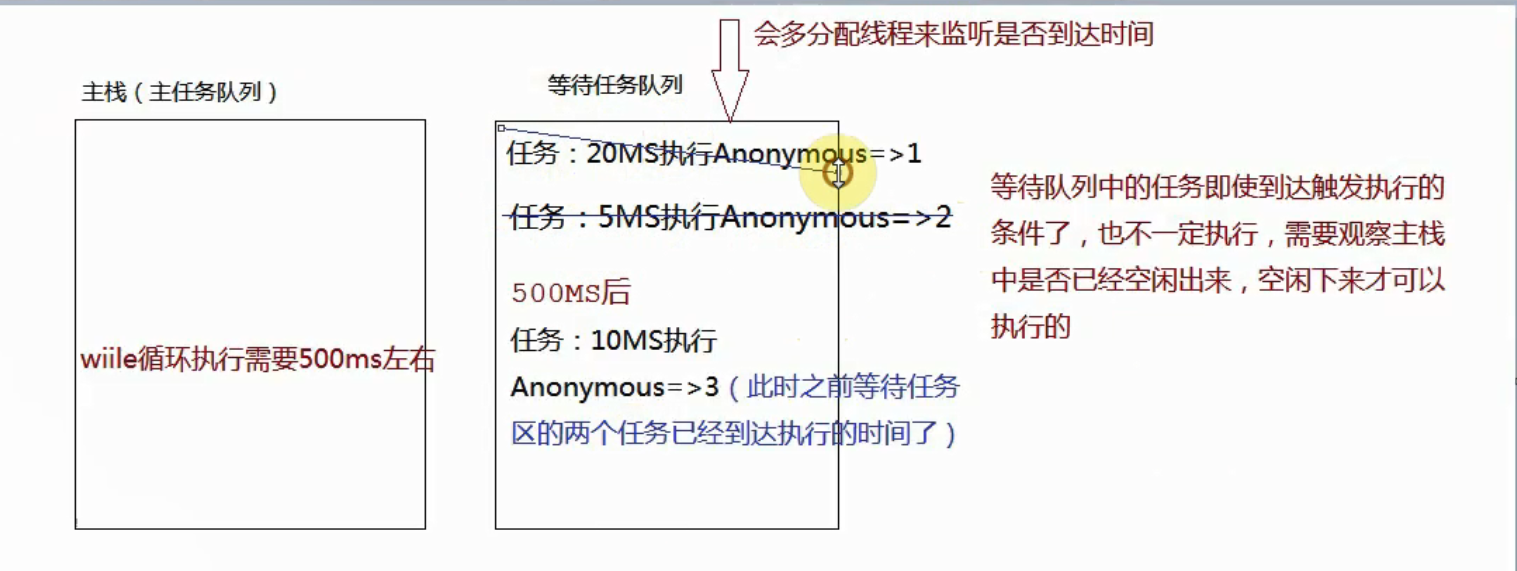
ruturn 没有,默认是undefined, 要设置一个回调函数





异步编程





等待区谁先到达时间就把谁放到主任务队列中执行 (特殊情况除外)

10 ms的和20ms 的有时候肯能会前,看while的循环时间.while如果只是6秒,那么就是 4 2 3 1

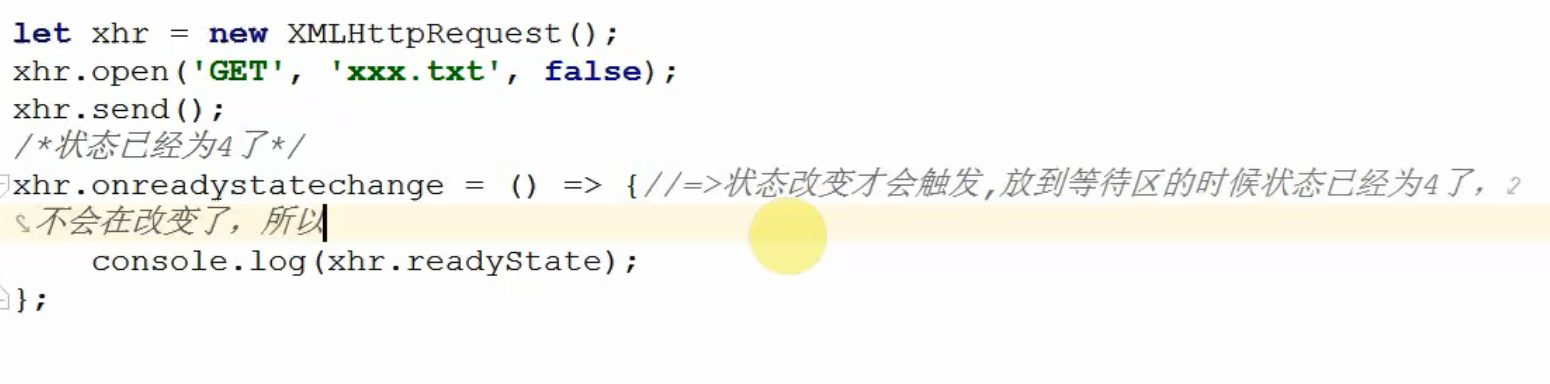
等待区中虽然 输出3 只要10ms[它=新发出请求,因为while消耗了不止20ms,是500ms],即前面20ms的早就到达时间了. 所以2先输出

同步的ajax和异步的ajax

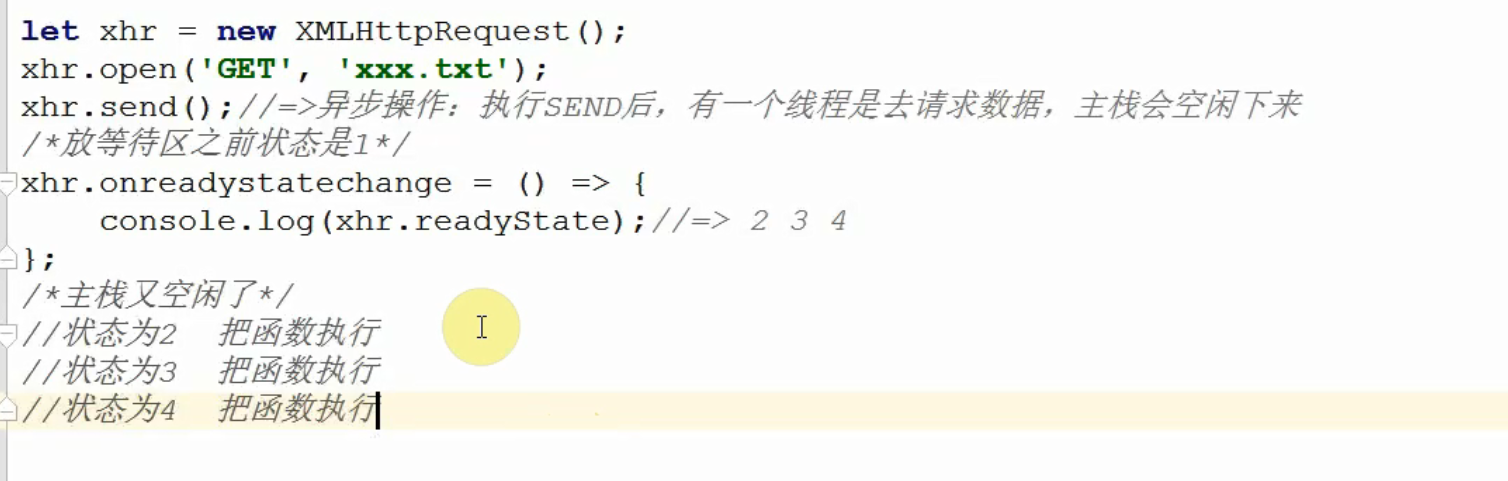
同步时:



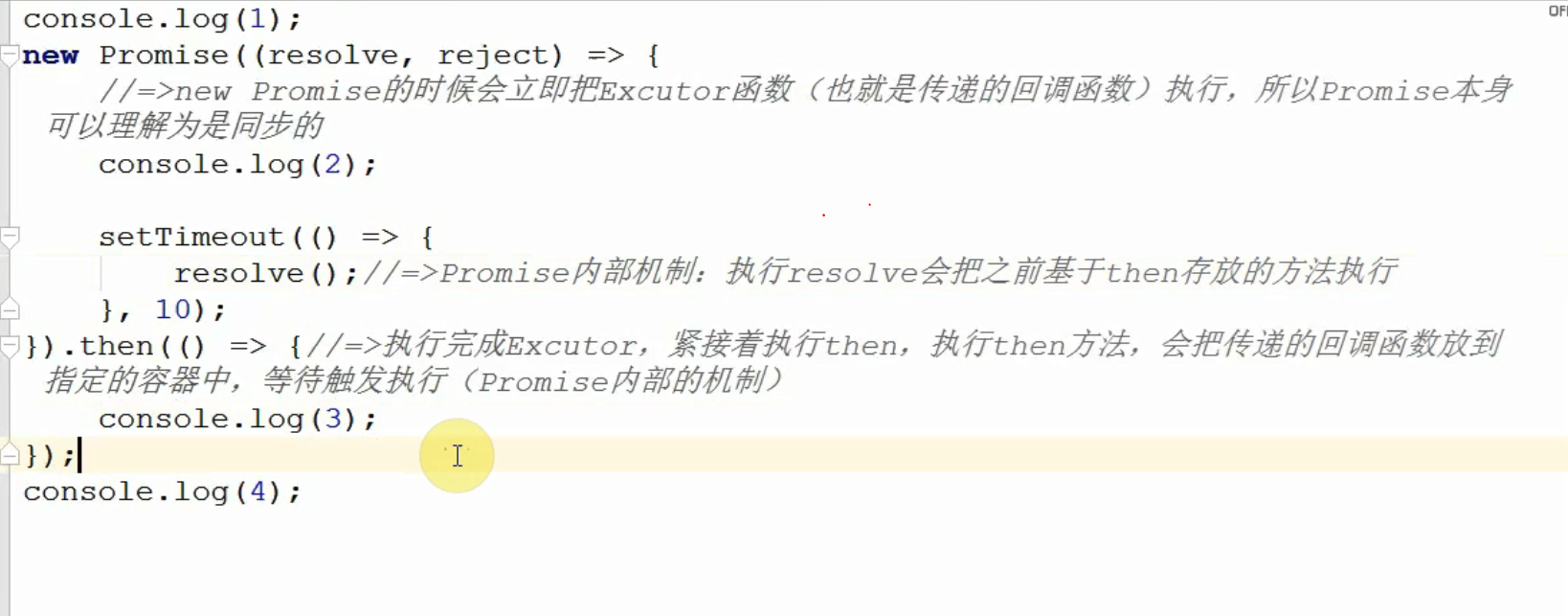
xhr.send( ) 和 xhr.onreadyChange() {} 调换位置时



异步操作时



async – promise



Promise 并不是完全的异步操作,resolve和reject是异步操作. new promise是一个同步操作,但resolve和reject不是同步而是异步的

new Promise ( ( *resolve*, *reject*) *=>* {

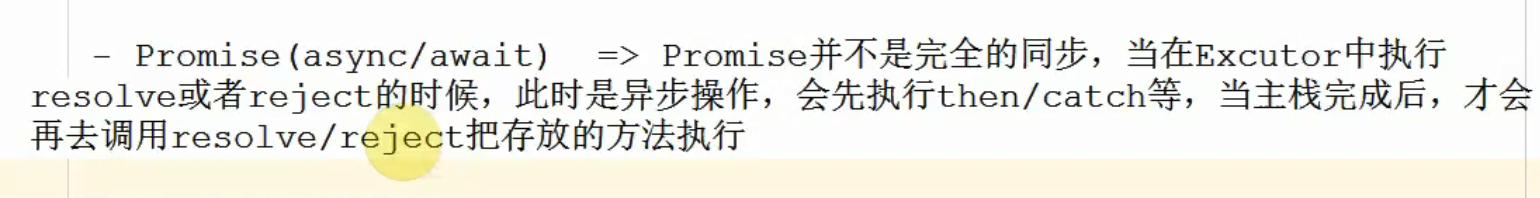
*console*.log(2);

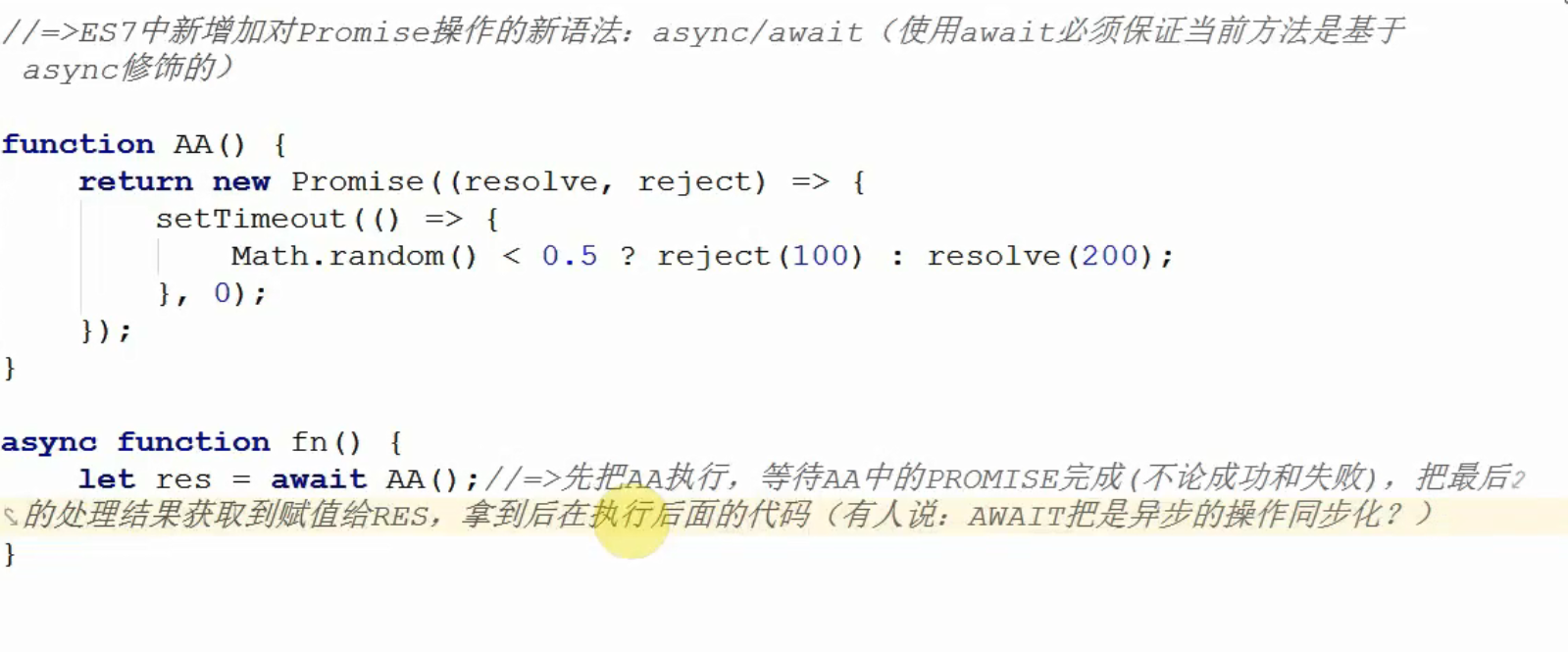
resolve()

}).then(

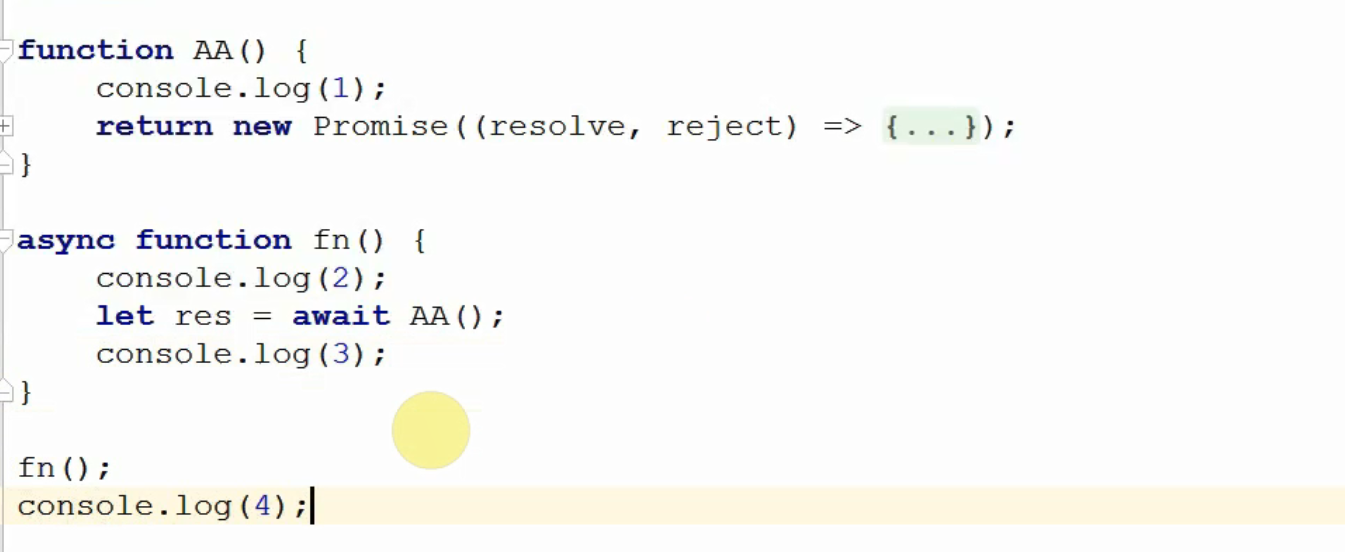
*console*.log(3)

)



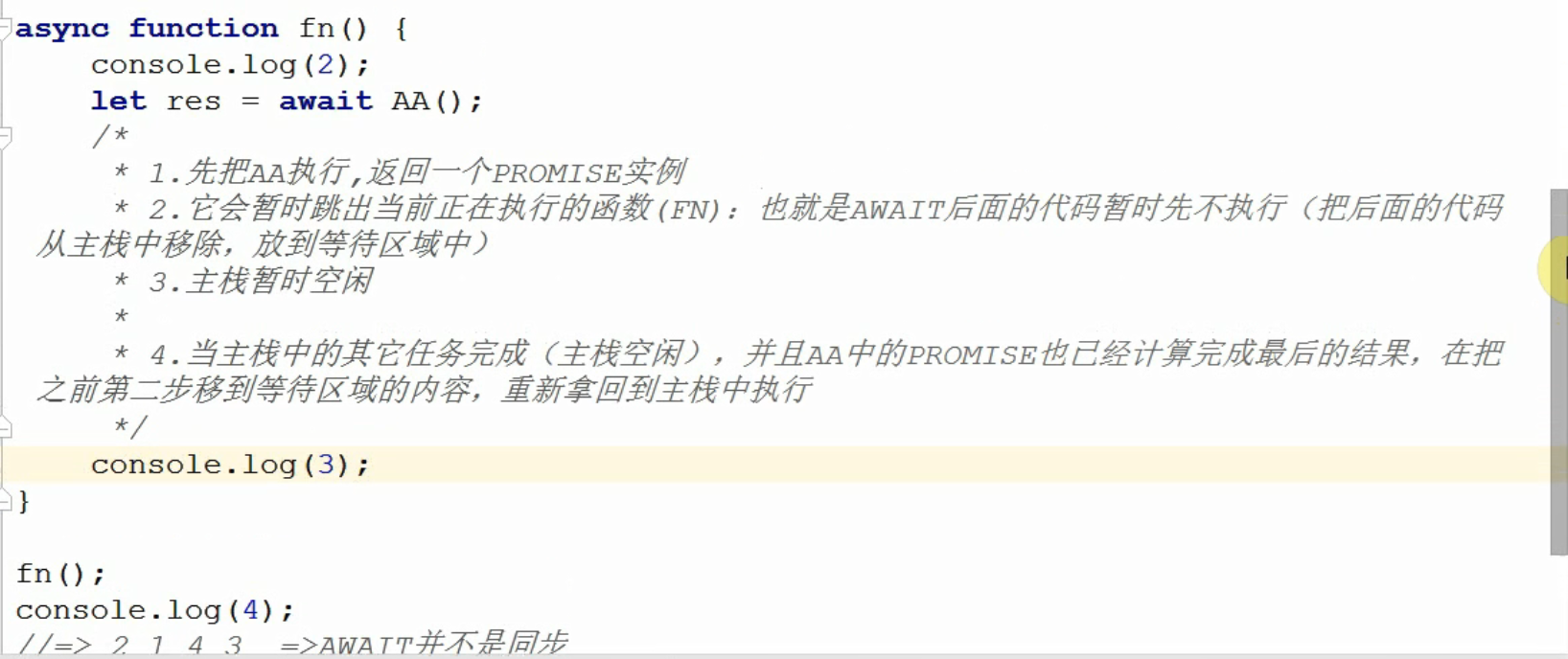


async await 是否是把异步操作同步化



// =>2 1 4 3

进来先执行 fn 输出2



经典案例

async *function* async1() {

*console*.log( 'async1 start');

await async2();

*console*.log('async1 end');

}

async *function* async2 (){

*console*.log('async2');

}

*console*.log('script start');

setTimeout ( *function*(){

*console*.log('setTimeout')

},0);

async1();

new Promise ( *function*(*resolve*){

*console*.log( 'promise1');

resolve;

}).then( () *=>* {

*console*.log('promise2')

});

*console*.log('script end')

宏任务: macro task

定时器

事件绑定

ajax

回调函数

node中fs可以进行异步的I/O操作

微任务: micro task

Promise(async/awiat) =>

process.nextTick

SYNC => micro(微任务) => macro(宏任务)

promise 在定时器之前

