**简述同步和异步,以及js的任务队列.**

javascript是单线程的一门语言，所以在执行任务的时候，所有任务必须排队，然后一个一个的执行.这就是同步模式

所以同步任务指的是，在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；

异步任务指的是，不进入主线程、而进入"任务队列"（task queue）的任务，只有等主线程任务执行完毕，"任务队列"开始通知主线程，请求执行任务，该任务才会进入主线程执行。

所以js的运行机制是这样的:

(1): 所有同步任务都在主线程上执行，形成一个执行栈.

(2): 主线程之外，还存在一个"任务队列"（task queue）。异步任务进入任务队列等待主线程执行完毕.

(3): 一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，对应的异步任务结束等待状态，进入执行栈，开始执行。

怎么知道主线程任务为空: JS引擎中存在monitoring process进程，它会持续不断的检查主线程执行栈是否为空，一旦为空，就会去Event Queue那里检查是否有等待被调用的函数。

(4) 所谓"回调函数"（callback），就是那些会被主线程挂起来的代码; 异步任务必须指定回调函数，当主线程开始执行异步任务，就是执行对应的回调函数。

**什么是js事件循环Event Loop轮询机制?**

(1): 所有同步任务都在主线程上执行，形成一个执行栈.

(2): 主线程之外，还存在一个"任务队列"（task queue）。异步任务进入任务队列等待主线程执行完毕.

(3): 一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，对应的异步任务结束等待状态，进入执行栈，开始执行。

（4）主线程不断重复上面的第三步。

只要主线程空了，就会去读取"任务队列"，这就是JavaScript的运行机制。这个过程会不断重复。该过程称为事件循环/轮询)

**什么是微任务和宏任务?**

除了广义的同步任务和异步任务，我们对任务有更精细的定义：

macro-task(宏任务)：包括整体代码script，setTimeout，setInterval;

micro-task(微任务)：Promise.then，Promise.catch process.nextTick.

宏任务的事件循环:宏任务执行,执行结束后判断是否有微任务执行,有的话执行所有微任务,再执行下一个宏任务.没有的话直接执行下一轮宏任务.如此循环.

**怎么理解js的多线程?**

为了利用多核CPU的计算能力，HTML5提出Web Worker标准，允许JavaScript脚本创建多个线程，但是子线程完全受主线程控制，且不得操作DOM。所以，这个新标准并没有改变JavaScript单线程的本质。

**输出结果:**

setTimeout(function(){

console.log('1')

},0);

new Promise(function(resolve){

console.log('2');

for(var i = 0; i < 10000; i++){

i == 99 && resolve();

}

}).then(function(){

console.log('3')

});

console.log('4');

打印结果顺序为:

2,4,3,1

解析:首先整个script代码作为宏任务执行,,首先遇到setTimeout是宏任务,跳过;接着遇到promise,立即执行;打印2;遇到promise.then微任务,推入任务队列;遇到console.log;立即执行,打印4;第一轮宏任务执行完毕,看看有哪些微任务, promise.then执行,打印3;没有微任务了;执行下一轮宏任务setTimeout,打印1.

注意: promise是立即执行的，它创建的时候就会执行，不存在将promise推入微任务中的说法； resolve()是用来表示promise的状态为fullfilled，相当于只是定义了一个有状态的Promise，但是并没有调用它； promise调用then的前提是promise的状态为fullfilled；只有promise调用then的时候，then里面的函数才会被推入微任务中.

**输出结果:**

new Promise(function(resolve,reject){

resolve();

}).then(function(){

console.log("1");

return new Promise(function(resolve,reject){

resolve();

console.log("2")

})

}).then(function(){ console.log("3");})

new Promise( function(resolve,reject){

console.log("4")

resolve();

}).then(function(){ console.log("5");})

//输出结果顺序:4,1,2,5,3

分析:嵌套的异步会加入到下次的任务队列中.

**输出结果:**

setTimeout(function () {

console.log('1');

},0);

new Promise(resolve => {

  console.log('2');

  resolve('3');

}).then(data => {console.log(data)});

function main(){

  console.log('4');

}

console.log('5');

console.log('6');

main();

//打印结果顺序:2,5,6,4,3,1

分析:先执行整体script的宏任务,遇到setTimeout是宏任务,跳过,遇到new promise,立即执行,打印2;遇到promise.then,推入微任务队列;遇到console.log(),打印5,6;遇到函数执行,打印4;

宏任务执行结束,查看是否有微任务,执行promise.then(),打印data即3;微任务执行完毕,开始下一轮宏任务setTimeout,打印1.

**异步编程有哪些方式?**

(1)回调函数是Javascript异步编程中最常见的, 回调代表着，当某个任务处理完，然后需要做的事，比如读取文件，连接数据库，等数据库连接成功执行编写的回调函数，又比如像一些动画处理，当动画走完，然后执行回调.

回调的好处就是容易编写，缺点就是过多的回调会产生回调地狱，代码横向扩展，代码可读性变差

(2) Promise对象

Promise对象有三种状态:pending(进行中)，fulfilled(已成功)，rejected(已失败)，当异步操作有结果时可以指定pending状态到fulfilled状态或pending状态到rejected状态的转换，状态一旦变为fulfilled，或rejected则这个Promise对象状态不会在改变。

let promise = new Promise((resolve, reject) => {//Promise对象接受一个函数

  try {

    setTimeout(() => {//模拟某异步操作 , 若操作成功返回数据

      resolve('hello world');

    }, 1000);

  }catch (error) {

    reject(error)

  }

});

promise.then((data) => {

  console.log(data)   //resolve()函数里面传的值

},(err) => {

  console.log(err) //reject()函数里传的值

});

(3) async/await(javascript异步的终极解决方案)

async修饰function,其返回一个Promise对象。

await必须放在async修饰的函数里面,并且放在异步请求的前面,表示等待异步结果,

解决异步的最好的方式了，让异步代码写起来跟同步代码一样，可读性和维护性都上来了。

async function main() {

    let [data1,data2] = await Promise.all([ajax.get('/api/test1', {name: 'dpf', age: 19}),ajax.post('/api/test2', {name: 'youname', age: 19})]);

    console.log(data1,data2)

}

main(); // main执行就可以拿到结果了,让异步代码用起来非常像同步代码