async/await和Promise的执行顺序

Promise

http://es6.ruanyifeng.com/#docs/promise

async/await

为什么使用async/await?

1.相比于 Promise，它能更好地处理 then 链

```
function takeLongTime(n) {
    return new Promise(resolve => {
        setTimeout(() => resolve(n + 200), n);
    });
}

function step1(n) {
    console.log(`step1 with ${n}`);
    return takeLongTime(n);
}

function step2(n) {
    console.log(`step2 with ${n}`);
    return takeLongTime(n);
}

function step3(n) {
    console.log(`step3 with ${n}`);
    return takeLongTime(n);
}
```

用 Promise 方式来实现这三个步骤的处理。

```
function doIt() {
    console.time("doIt");
    const time1 = 300;
    step1(time1)
        .then(time2 => step2(time2))
        .then(time3 => step3(time3))
        .then(result => {
            console.log(`result is ${result}`);
        });
}
doIt();
// step1 with 300
// step2 with 500
// step3 with 700
// result is 900
```

用 async/await 来实现。

```
async function doIt() {
    console.time("doIt");
    const time1 = 300;
    const time2 = await step1(time1);
    const time3 = await step2(time2);
    const result = await step3(time3);
    console.log(`result is ${result}`);
}
doIt();
```

2.中间值语义化

现在把业务要求改一下，仍然是三个步骤，但每一个步骤都需要之前每个步骤的结果。Pomise的实现看着很晕，传递参数太过麻烦。

```
function doIt() {
    console.time("doIt");
    const time1 = 300;
    step1(time1)
        .then(time2 => {
            return step2(time1, time2)
                .then(time3 => [time1, time2, time3]);
        })
        .then(times => {
            const [time1, time2, time3] = times;
            return step3(time1, time2, time3);
        })
        .then(result => {
            console.log(`result is ${result}`);
        });
```

```
}
doIt();
```

用 async/await 来写：

```
async function doIt() {
    console.time("doIt");
    const time1 = 300;
    const time2 = await step1(time1);
    const time3 = await step2(time1, time2);
    const result = await step3(time1, time2, time3);
    console.log(`result is ${result}`);
}
doIt();
```

一些概念

> Javascript是单线程的，所有的同步任务都会在主线程中执行。
> 主线程之外，还有一个任务队列。每当一个异步任务有结果了，就往任务队列里塞一个事件。
> 当主线程中的任务，都执行完之后，系统会“依次”读取任务队列里的事件。与之相对应的异步任务进入主线程，开始执行。
> 异步任务之间，会存在差异，所以它们执行的优先级也会有区别。大致分为 微任务（micro task，如：Promise、MutaionObserver等）和宏任务（macro task，如：setTimeout、setInterval、I/O等）。同一次事件循环中，微任务永远在宏任务之前执行。
> 主线程会不断重复上面的步骤，直到执行完所有任务。

JS执行顺序

例子：

```
function t1 () {
  console.log(1)
  var observer = new MutationObserver(() => console.log('ob'))
  observer.observe($0, { attributes: true })
  $0.style.height = '200px'
  console.log(2)
}

function t2() {
  console.log(3)
  Promise.resolve().then(() => console.log('ps'))
  console.log(4)
}

t1()
t2()

// 输出：
```

```
// 1
// 2
// 3
// 4
// ob
// ps
// undefined
```

上面这个例子说明，MutationObserver和Promise执行后，回调均被放置在本次事件循环tasks之后的Microtasks队列里，并依次执行。

async/await

await 的执行顺序：

```
async function async1() {
    console.log( 'async1 start' )
    await async2()
    console.log( 'async1 end' )
}
async function async2() {
    console.log( 'async2' )
}
async1()
console.log( 'script start' )
//async1 start
//async2
//script start
//async1 end
//undefined
```

从右向左的。先打印async2，后打印的script start

右侧表达式的结果，就是await要等的东西。 等到之后，对于await来说，分2个情况

- 不是promise对象
- 是promise对象

如果不是 **promise , await**会阻塞后面的代码，先执行**async**外面的同步代码，同步代码执行完，再回到**async**内部，把这个非**promise**的东西，作为 **await**表达式的结果

如果它等到的是一个 **promise** 对象，**await** 也会暂停**async**后面的代码，先执行**async**外面的同步代码，等着 **Promise** 对象 **fulfilled**，然后把 **resolve** 的参数作为 **await** 表达式的运算结果。

Chrome 提交了优化ECMAScript编辑性更改：
1、await 后面不一定会创建新的微任务，取决于await 后面是立即返回还是promise。立即返回则不创建。
2、await 执行之后不会强制创建新的微任务，而是继续执行。

以头条的一道面试题为例,分析下这段代码的执行顺序.

```
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( function () {
    console.log( 'promise2' )
} )
console.log( 'script end' )
```

> 直接打印同步代码 console.log('script start')
> 首先是2个函数声明，虽然有async关键字，但不是调用我们就不看。然后首先是打印同步代码 console.log('script start')

> 将setTimeout放入下一个宏任务队列

> 调用async1，打印 同步代码 console.log('async1 start')

> 分析下await async2()

```
前文提过await，1.它先计算出右侧的结果，2.然后看到await后，中断async函数
所以直接打印 console.log('async2')
```

被阻塞后，要执行async之外的代码

> 执行new Promise()，Promise构造函数是直接调用的同步代码，所以 console.log('promise1')

> 代码运行到promise.then()，发现这个是微任务，所以暂时不打印，只是推入当前宏任务的微任务队列中。

> 打印同步代码 console.log('script end')

执行完这个同步代码后，回到 await 表达式

> async2() 没有返回值，语义上就是 await undefined即立即返回，所以不创建微任务继续执行 console.log('async1 end')

宏任务中的同步任务执行完成后，开始执行微任务队列

> console.log('promise2')

至此整个宏任务都执行完成，开始下一个宏任务

> console.log('setTimeout')

最后结果：

```
script start
async1 start
async2
promise1
script end
async1 end
promise2
undefined
setTimeout
```

如果把async2 方法改成

```
async function async2() {
        return new Promise(function(resolve){
            console.log( 'async2' );
            resolve()
        })
}
```

那么在回到await表达式时 await 后面是promise，这个Promise.then()创建微任务加到队列中

结果就变成了：

```
script start
async1 start
async2
promise1
script end
promise2
async1 end
undefined
setTimeout
```