Async 和 Await

1. 基础语法

2. 使用场景

2.1 处理一般的成功结果

2.2 await管理异步

2.3 管理多个请求

3. JS 中的 宏任务 和 微任务

1. 基础语法

JavaScript | 复制代码

```
// 1. async 和 await
// 它俩的出现在es8中，组合出现，await不能单独用
// await 可以多次出现，跟 yield 一样，往外丢东西
// 一般用于操作基本promise封装的函数
/*
    a. 只能嵌套一级，await 一定要找最近的函数声明 async
    b. 不区分箭头函数或者普通函数，都是在函数体前面写
    c. await 可以获取到 resolve 传过来的内容
*/
// 2. 普通函数
// async function fn() {
//     await console.log(100);
// }
// function fn() {
//     (async () => {
//         await console.log(100);
//     })()
// }
// fn();

```

2. 使用场景

2.1 处理一般的成功结果

JavaScript | 复制代码

```javascript
// 3. 经典面试题
// let num = 0;
function getNum() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const num = 100;
            // 改变状态，通知then方法的回调1执行
            resolve(num);
        }, 1000)
    })
}
// 使用 async 和 await 的方式
; (async () => {
    // await getNum();
    const result = await getNum();
    // 等到promise的状态为 resolve 就会往下放行
    console.log(result);
    // console.log(num);
})()
// getNum()
//     .then(data => {
//         console.log(num);
//     })
```

2.2 await管理异步

JavaScript | 复制代码

```javascript

function printText01() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('1, 打印内容，天王盖地虎')
        }, 3000)
    })
}


function printText02() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('2, 打印内容，宝塔镇河妖')
        }, 500)
    })
}


function printText03() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('3, 打印内容，最后的结束内容')
        }, 4000)
    })
}
```

JavaScript | 复制代码

```javascript
// 正常的调用
// printText01().then(data => console.log(data));
// printText02().then(data => console.log(data));
// printText03().then(data => console.log(data));

// printText01()
//     .then(data => {
//         console.log(data)
//         return printText02()
//     })
//     .then(data => {
//         console.log(data)
//         return printText03()
//     })
//     .then(data => {
//         console.log(data)
//     })

// async 和 await
; (async () => {
    const text01 = await printText01()
    console.log(text01);
    const text02 = await printText02()
    console.log(text02);
    const text03 = await printText03()
    console.log(text03);
})()
console.log(100);
```

2.3 管理多个请求

JavaScript | 复制代码

```javascript
// 1. 发请求


// 1. 请求 美女图片
function getLady() {
    return new Promise((resolve, reject) => {
        $.ajax({
            method: 'get',
            url: 'https://www.mxnzp.com/api/image/girl/list/random',
            data: {
                app_id: 'qjwkhneqigkkujok',
                app_secret: 'VjNDVmIrZFkvaGhNdElPbmQxQzRhUT09'
            },
            success(res) {
                resolve(res);
            }
        })
    })
}


// 2. 请求 每日一句
function heros() {
    return new Promise((resolve, reject) => {
        $.ajax({
            url: 'http://api.xiaohigh.com/lol',
            success(res) {
                setTimeout(() => {
                    resolve(res);
                }, 2000)
            }
        })
    })
}
```

JavaScript | 复制代码

```
// 注册事件
$('button').click(async function () {
    // 按顺序返回请求的内容
    const res1 = await heros();
    console.log(res1.slice(0, 3));
    const res2 = await getLady();
    console.log(res2.data[0]);
    // heros()
    //     .then(res => {
    //         console.log(res.slice(0, 2));
    //         return getLady()
    //     })
    //     .then(res => {
    //         console.log(res.data[0]);
    //     });
})
```

3. JS 中的 宏任务 和 微任务

浏览器有排版引擎，和js解析引擎，js 是单线程的，每次只能处理一个任务

同步的代码优先执行，为了模拟多线程的感觉，才有了异步的方法

在js中，异步的方法总共分为3中：**dom的事件，ajax的回调，定时器**

那么，这些任务开启时，会自动去浏览器的任务队列中排队，也有一个新名字叫做宏队列

window对象，就会根据事件循环机制（Event Loop） 找对对应的代码执行

script 代码的读写也是宏任务，一般来说就没有算进去，if, for, whlie, 函数的调用，数组遍历...

除了宏任务之外，还有微任务，微任务指的就是通过 Promise 返回的那些回调

这些回调会后于同步的代码执行，但是先于下一个宏任务执行。

如果微任务比较多，也会进入微任务的队列。

学习和了解这些内容，并不会让封装代码或者业务逻辑多么高超，但是会让代码分析变得更有信心。

比如面试题，比如后期的一些代码报错和执行。

面试题1 JavaScript 复制代码

```javascript
setTimeout(() => {
    console.log("0")
}, 0)
new Promise((resolve, reject) => {
    console.log("1")
    resolve()
}).then(() => {
    console.log("2")
    new Promise((resolve, reject) => {
        console.log("3")
        resolve()
    }).then(() => {
        console.log("4")
    }).then(() => {
        console.log("5")
    })
}).then(() => {
    console.log("6")
})
    .then(() => {
        console.log('9');
    })
new Promise((resolve, reject) => {
    console.log("7")
    resolve()
}).then(() => {
    console.log("8")
})
```

最终的总结：

先是同步代码，

同步代码结束之后，再执行第一级的then

第一级的then执行完毕再执行第二级的then，以此类推

所有的微任务执行结束之后，再执行宏任务

如果宏任务中，内部继续含有同步代码，微任务和宏任务，那么顺序一样。