1. Promise.all()并发方法的特点：

         1、要么所有Promise实例都成功、要么都失败，

         2、如果都成功则返回所有Promise实例结果，并且该结果是数组，数组中的元素与all()方法中Promise实例一一对应;

        3、如果有一个Promise实例执行失败则都失败，并且只返回最先失败的那个Promise实例结果。

    <script>

        //并发：

        //热映电影

        function hotMovie() {

            $.get('热映电影地址', {}, function(d) {}, 'json');

        }

        //hotMovie();

        //推荐电影

        function recMovie() {

            $.get('推荐电影地址', {}, function(d) {}, 'json');

        }

        //recMovie()

        //Promise.all([promise对象1,promise对象2...])

        function demo1() {

            let p = new Promise((resolve, reject) => {

                setTimeout(() => {

                    resolve('1秒');

                    // reject('失败1');

                }, 1000);

            });

            return p

        }

        function demo2() {

            let p = new Promise((resolve, reject) => {

                setTimeout(() => {

                    resolve('3秒');

                    //  reject('失败3');

                }, 3000);

            });

            return p

        }

        let pm = Promise.all([demo2(), demo1()]);

        // let pm = Promise.all([demo1(), demo2()]);

        //  console.log(arr);  //Promise对象

        pm.then((d) => {

            console.log(d);

        }).catch((err) => {

            console.log(err);

        });

    </script>

1. Promise.race()方法的特点：

哪个promise实例最先执行完成返回哪个promise的结果，不管这个结果是resolve还是reject，只比谁最快。

    <script>

        //Promise.race([promise对象1,promise对象2...])

        function demo1() {

            let p = new Promise((resolve, reject) => {

                setTimeout(() => {

                    resolve('1秒');

                }, 1000);

            });

            return p

        }

        function demo2() {

            let p = new Promise((resolve, reject) => {

                setTimeout(() => {

                    resolve('2秒');

                }, 2000);

            });

            return p

        }

        let prm = Promise.race([demo1(), demo2()]);

        prm.then((d) => {

            console.log(d);

        }).catch((err) => {

            console.log(err);

        });

    </script>

1. 什么是iterator?

Iterator中文为迭代器或遍历器，它是一种接口机制，只要提供了这种接口机制的数据类型都可以使用for...of来遍历，这些数据类型都可以使用for...of来遍历：Array、String、arguments、Set、map.

数组原来可以使用for或forEach来遍历；

对象原来可以使用for...in来遍历；

Set、map原来可以使用forEach来遍历；

1. Iterator接口使用:

    <script>

        //iterator简介

        //iterator是一种接口机制，只要提供了这种接口机制的数据类型都可以统一使用for...of来遍历，这些数据类型提供了iterator接口机制：String、Array、arguments、set、map

        //Array数组原来可以使用forEach或for来遍历；

        //Object对象原来可以使用for..in来遍历

        let arr = [1, 3, 5, 9, 10];

        //  console.log(arr);

        let arr\_iterator = arr[Symbol.iterator]()

            // console.log(arr\_iterator);

            // console.log(arr\_iterator.next());

            // console.log(arr\_iterator.next());

            // console.log(arr\_iterator.next());

            // console.log(arr\_iterator.next());

            // console.log(arr\_iterator.next());

            // console.log(arr\_iterator.next());

//使用for...of遍历数组：

         for (var x of arr) {

             console.log(x);

         }

</script>

使用代码演示iterator接口工作过程(了解):

    <script>

     function demo() {

            // console.log(arr);

            var i = 0; //下标

            return {

                next: function() {

                    //判断数组是否遍历完

                    if (i < arr.length) {

                        return {

                            value: arr[i++],

                            done: false

                        };

                    } else { //数组遍历完成

                        return {

                            value: undefined,

                            done: true

                        };

                    }

                }

            };

        }

        let dmObj = demo();

        console.log(dmObj.next());

        console.log(dmObj.next());

        console.log(dmObj.next());

        console.log(dmObj.next());

        console.log(dmObj.next());

        console.log(dmObj.next());

    </script>

1. For...of与forEach、for...in的区别？

1、for...of与forEach的区别？

forEach中不能使用break、continue、return，但for...of中则可以使用。

 <script>

  for (var x of arr) {

            if (x > 5) {

                break;

            }

            console.log(x);

        }

 </script>

1. for...of与for...in的区别？

For...in通常遍历对象，for...of可以遍历数组、string、arguments、set、map.

1. 什么是generator?

Generator是一个函数，在内部可以定义多个不同状态，因此generator也称为状态机，它是异步编程解决方案之一。

1. Generator语法：

function\* 函数名() {

yield 异步操作1;

yield 异步操作2;

}

1. Generator函数特点:

function 与函数名之间有一个星号

内部用yield表达式来定义不同的状态

generator函数返回的是iterator对象，而不会执行函数内部逻辑

调用next方法函数内部逻辑开始执行，遇到yield表达式停止，返回{value: yield后的表达式结果/undefined, done: false/true}

再次调用next方法会从上一次停止时的yield处开始，直到最后

yield语句返回结果通常为undefined

**注意**：当给next()方法传参时会作为重启yield语句的返回值。

    <script>

        function demo() {

        }

        //  console.log(demo());

        //generator函数：

        function\* fn() {

        }

        // console.log(fn());

        // let gene = fn();

        //  console.log(gene.next());

        function\* demo() {

            console.log('开始执行...');

            let val = yield 123;

            console.log(val, '执行到这里');

            val = yield 'Hello';

            console.log(val, '执行完成....');

        }

        //注意：generator函数调用后函数体不会自动执行，需要手动调用next()方法

        //当给next()方法传参时会作为重启yield语句的返回值

        let genes = demo();

        let obj = genes.next();

        console.log(obj);

        obj = genes.next(66);

        console.log(obj);

        obj = genes.next('ok');

        console.log(obj);

    </script>

1. Generator应用：

**需求**：调用所有商品一级分类并展示到界面

     <script>

        //定义函数：获取所有商品一级分类数据

        function getCateData(url, data = {}, datatype = 'json') {

            $.get(url, data, function(d) {

                iter.next(d);

            }, datatype);

        }

        //定义generator函数：

        function\* showcate() {

            let data = yield getCateData('http://106.13.114.114:5000/api/firstCategory');

            //  console.log(data, 111);

            renderData(data.list)

        }

        let iter = showcate();

        iter.next();

        //定义函数：展示所有商品一级分类数据到界面上

        function renderData(arr) {

            console.log(arr, 777);

            let str = '';

            for (let i = 0; i < arr.length; i++) {

                // console.log(arr[i]);

                for (let s = 0; s < arr[i].length; s++) {

                    //  console.log(arr[i][s], 88);

                    str += `<li>${arr[i][s].firstName}</li>`

                }

            }

            // console.log(str);

            $("#content").html(`<ul>${str}</ul>`);

        }

    </script>

**作业**： 使用generator+ajax技术解决回调地狱问题？

1. async函数简介： Async函数是一个基于Promise的generator语法糖，以同步流程表达异步操作。

1. async函数语法：

async function 函数名(){

await 异步操作1;

await 异步操作2;

}

1. async函数特点：

1不需要像Generator去调用next方法，遇到await等待，当前的异步操作完成就往下执行

2返回的总是Promise对象，可以用then方法进行下一步操作

3async取代Generator函数的星号\*，await取代Generator的yield

4当给resolve()回调函数传参时会作为对应await语句的返回值

5语意上更为明确，使用简单

1. async用法注意:

async是用来修饰function函数的(放在function关键字前面)，被async修饰过的函数，返回值是一个promise对象(不管这个函数内部是否有return返回值)；这个promise对象的状态是resolved（成功状态）；除非在async函数内部出现异常，此时的promise状态是rejected；async函数返回的promise状态是自动的，我们不能像promise一样手动调用resolve和reject方法。

    <script>

        // async function demo() {

        //     return '成功...'

        // }

        // console.log(demo()); //返回的是promise对象

        // let prm = demo();

        // prm.then((d) => {

        //     console.log(d, 111)

        // }).catch((err) => {

        //     console.log(err, 333);

        // });

        //await必须与async结合使用，但async可以单独使用

        async function fn() {

            await '123';

        }

        // fn();

        async function test() {

            console.log('开始...');

            await 12;

            console.log('执行到这里');

            await 'Hi';

            console.log('完成...');

        }

        test();

    </script>