一、Generator函数

1、基本使用

Generator函数也称之为生成器函数，可以用来生成迭代器，。也就是可以通过for...of来遍历Generator函数.并且Generator函数提供了一种异步编程的解决方案。

生成器函数和普通函数不一样，普通函数是一旦调用就会执行完毕，但是生成器函数中间可以暂停，也就是执行一会歇一会。

Generator函数的创建

```
        function* go() {
                console.log(1);
                let a = yield 'a';
                console.log(2);

                let b = yield a;
                console.log(3);
                return b;
            }

        let it = go();
        let r1 = it.next();
        console.log(r1);
        let r2 = it.next('b的值');
        console.log(r2);
        let r4 = it.next('c的值');
        console.log(r4);
```

如果第一次执行next方法给变量a输入值应该怎样传值?

```
        function* go(str) {
                console.log(1);
                let a = yield str;
                console.log(2);
                // 这行代码实现了输入与输出，本次的输出放在了yield的后面，下次的输入放在了yield的前面。
                let b = yield a;
                console.log(3);
                return b;
            }
            // 在这里先去调用一下生成器函数，但是注意，调用它不会立即执行
            // 该函数在这里会返回一个迭代器
        let it = go('a的值'); // 调用的时候进行值的传递
        //  下面需要调用next()函数
        let r1 = it.next();

        console.log(r1);
        let r2 = it.next('b的值');
```

```
        console.log(r2);

        let r4 = it.next('c的值');
        console.log(r4);
```

2、next方法参数

在上一个案例中,给next方法添加了相应的参数，那么该参数会 被当作上一条yield语句的返回值。

下面看一下如下程序，判断其对应的输出结果。(直接看程序)

```
       function* test(num) {
            let x = 3 * (yield(num + 1));
            let y = yield(x / 3);
            return (x + y + num);
        }
        let n = test(6);
        console.log(n.next());
        console.log(n.next());
        console.log(n.next());
```

输出结果如下：

```
{value: 7, done: false}
{value: NaN, done: false}
{value: NaN, done: true}
```

现在将程序修改成如下的形式：

```
        function* test(num) {
            let x = 3 * (yield(num + 1));
            let y = yield(x / 3);
            return (x + y + num);
        }
        let n = test(6);
        console.log(n.next());
        console.log(n.next(3));
        console.log(n.next(3));
```

输出结果如下：

```
{value: 7, done: false}
{value: 3, done: false}
{value: 18, done: true}
```

注意：由于next方法的参数表示上一条yield语句的返回值，所以第一次使用next方法时不能带参数。

也就是第一次使用next方法时是用来启动遍历器对象的。

3、for...of循环

for...of循环可以自动遍历Generator函数，且此时不再需要调用next方法。

```
function* test() {
    yield 1;
    yield 2;
    yield 3;
    yield 4;
    yield 5;
    return 6;
}
for (let v of test()) {
    console.log(v);
}
```

注意：一旦next()方法返回的对象的done属性为true, for...of循环就会终止，且不包含该返回对象，所以上面的return语句不在for...of循环中。

在前面的课程中讲过，由于JavaScript对象没有遍历的接口，无法使用for...of进行遍历，那么现在可以通过Generator函数为它加上这个接口就可以了。

```
let user = {
    name: 'zs',
    age: 18
}

function* test(obj) {
    let keys = Reflect.ownKeys(obj);
    for (let key of keys) {
        yield [key, obj[key]];
    }
}
for (let item of test(user)) {
    console.log(item);
}
```

4、yield* 语句

如果在Generator函数内部调用一个Generator函数，默认情况下是没有效果的。

```
function* test() {
    yield 'a';
    yield 'b';
}

function* test1() {
    yield 'x';
    test();
    yield 'y';
}
for (let v of test1()) {
    console.log(v);
}
```

要解决这个问题，需要用到 yield* 语句，用来在一个Generator函数中执行另外一个Generator函数。

上面的程序，修改成如下的形式：

```
function* test() {
    yield 'a';
    yield 'b';
}

function* test1() {
    yield 'x';
    yield* test();
    yield 'y';
}
for (let v of test1()) {
    console.log(v);
}
```

其实上面的代码与下面的代码是等价的关系

```
function* test() {
    yield 'a';
    yield 'b';
}

function* test1() {
    yield 'x';
    for (let v of test()) {
        console.log(v);
    }
    yield 'y';
}
for (let v of test1()) {
    console.log(v);
}
```

所以 yield* 语句等同于在Generator函数内部部署了一个for...of循环。

看一下，下面的伪代码

```
function* test() {
    yield* it1;
    yield* it2;
}
```

上面的代码等同于下面的代码

```
function* test() {
    for (let value of it1) {
        yield value;
    }
    for (let value of it2) {
        yield value;
    }
}
```

如果 yield* 后面跟着一个数组，会出现什么情况呢?

由于数组原生支持遍历器，因此会遍历数组成员。

```
        function* test() {
            yield*[1, 2, 3, 4, 5, 6]
        }
        console.log(test().next())
```

上面的代码输出的结果为：

```
{value: 1, done: false}
```

通过上面的输出结果可以看出，加了星号后表示返回的是数组的遍历器对象。

如果不加星号，输出结果如下：

```
{value: Array(6), done: false}
```

不加星号返回的是整个数组。

所以，任何数据结构只要有了Iterator接口，就可以使用yield*来进行遍历。

下面再一段程序，看一下对应的输出结果

```
    function* test() {
            yield 1;
            yield 2;
            return 'test';
        }

        function* test1() {
            yield 3;
            let value = yield* test();
            console.log('value=', value);
            yield 4;
        }
        let it = test1();
        console.log(it.next());
        console.log(it.next());
        console.log(it.next());
        console.log(it.next());
        console.log(it.next());
```

输出的结果如下：

```
{value: 3, done: false}
{value: 1, done: false}
{value: 2, done: false}
value= test   {value: 4, done: false}
{value: undefined, done: true}
```

通过上面的代码可以，发现test函数中的return值，给了test1函数中的value这个变量。

5、关于Generator函数中的this问题

在讲解具体的this问题之前，先看一下下面的代码，是否有错误?

```
        function* Person() {
            yield this.name = 'zs';
            yield this.age = 18;
        }
        let person = new Person();
        console.log(person.name);
```

执行上面的代码后，发现是有错误的，因为Person既是构造函数，又是一个Generator函数，所以使用new命令就无法创建Person的对象。

怎样解决这个问题呢?

首先创建一个空对象，然后使用bind方法绑定Generator函数内部的this。这样，这个空对象就是Generator函数的实例对象了。

```
       function* Person() {
            yield this.name = 'zs';
            yield this.age = 18;
        }
        let person = {}
        let obj = Person.bind(person)();
        console.log(obj.next());
```

6、Generator函数应用场景

6.1 状态处理

单击按钮实现图片切换，这个案例如果按照以前的做法，如下:

```
   let button = document.getElementById('btn') //找到按钮
        let mm = document.getElementById('mv') //找到img标签
        let flag = 0
        button.onclick = function() {
            //将img标签的src属性的值，换成另外一张图片的地址.
            if (flag === 0) {
                mm.src = 'images/b.jpg';
                flag = 1;
            } else {
                mm.src = 'images/a.jpg';
                flag = 0;
            }

        }
```

使用 Generator函数处理

```
        let button = document.getElementById('btn') //找到按钮
        let mm = document.getElementById('mv') //找到img标签
        let it = f(0);
        button.onclick = function() {
            it.next();
```

```
        }

        function* f(flag) {
            while (true) {
                mm.src = 'images/b.jpg';
                yield flag;
                mm.src = 'images/a.jpg';
                yield flag;

            }
        }
```

使用Generator函数处理更加简单，并且更加符合函数的编程思想。（注意步骤的分析）

6.2 异步处理

前面讲过，Generator函数提供了一种异步处理的解决方案，而AJAX是典型的异步操作。

下面伪代码，直接看一下

```
function* main() {
  let result = yield request("http://xxx.com/api");
  let resp = JSON.parse(result);
    console.log(resp.value);
}

function request(url) {
  makeAjaxCall(url, function(response){
    it.next(response);
  });
}

let it = main();
it.next();
```

注意：在makeAjaxCall函数中的next( )方法，一定要把response作为它的参数。

因为该参数会给main函数中的result变量，最终对result进行处理。

并且，上面的写法几乎与同步操作的写法完全一样，写起来非常简单。

二、Promise对象

1、Promise定义

1.1 回调地狱问题

在讲解具体的Promise对象的定义前，先来讲解一下回调地狱的问题。

在开发中经常使用Ajax发送请求，那么就会出现如下的情况：

```
        $.ajax(url, success() {
            $.ajax(url2, success() {
                $.ajax(url3, success() {

                })
            })
        })
```

以上的代码反映了，在一个Ajax的回调中，又去发送了另外一个Ajax请求，依次类推，导致了多个回调函数的嵌套，导致代码不够直观并且难以维护，这就是常说的回调地狱。

所以在实际的开发中，不希望这种不断嵌套的回调，而是希望将这种多层变成一层。

要解决这个回调地狱的问题，就要用到Promise对象。

同步模式

同步模式指的就是代码中的任务依次执行。后一个任务必须等待前一个任务结束后，才能执行。程序的执行顺序与我们代码的编写顺序是完全一致的。

异步模式

异步模式对应的`API`是不会等待这个任务的结束才开始下一个任务，对于耗时操作，开启过后就立即往后执行下一个任务。

耗时任务的后续逻辑一般会通过回调函数的方式定义（例如ajax回调函数）。

Promise概念与基本使用

所谓的Promise就是一个对象，而Promise对象代表的是一个异步任务，也就是需要很长时间去执行的任务。

也就是通过Promise对象，可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数问题，也就是回调地狱的问题。

```
let promise = new Promise(function(resolve, reject) {
            setTimeout(function() {
                let num = Math.random();
                if (num > 0.3) {
                    resolve('成功了!')
                } else {
                    reject('失败了')
                }
            }, 3000)
        })
        promise.then(function(value) {
            console.log(value);
        }, function(reason) {
            console.log(reason);
        })
```

2 使用Promise封装AJAX操作

```
  let getJSON = function(url) {
```

```
            let p = new Promise(function(resolve, reject) {
                let xhr = new XMLHttpRequest();
                xhr.open('GET', url);
                xhr.onreadystatechange = handler;
                xhr.responseType = 'json';
                xhr.setRequestHeader('Accept', 'application/json');
                xhr.send();
                function handler() {
                    if (xhr.readyState === 4) {
                        if (this.status === 200) {
                            resolve(this.response)
                        } else {
                            reject(new Error(this.statusText));
                        }
                    }
                }
            });
            //返回Promise对象
            return p;
        }
        getJSON('http://localhost:3005/products').then(function(result) {
            console.log(result);
        }, function(error) {
            console.log('出错了:' + error)
        })
```

3、Promise链式调用

与传统回调函数处理异步任务相比，`Promise`最大的优势就是可以实现链式调用。

这样可以最大程度的避免回调地狱的问题。

`then`方法第一个参数是成功的回调，第二个参数是失败的回调，当然第二个参数是可以省略的。

`then`方法最大的特点就是可以返回一个`Promise`对象。

```
var promise=ajax('/api/users.json')
var promise2=promise.then(function onFulfilled(value){
    console.log('onFulfilled',value)
},function onRejected(error){
    console.log('onRejected',error)
})
console.log(promise2)//输出的是一个promise对象
console.log(promise2===promise)//返回值为false,所以这里的链式调用与前面我们学习的不一样，
以前是通过返回this的方式来实现。而这里的then方法
//返回的是一个全新的 Promise对象。
```

返回全新的`Promise`的目的，就是为了实现一个`Promise`的链条，也就是一个承诺结束后，返回一个新的承诺。每个承诺都可以负责一个异步任务，

相互之间没有什么影响，那么如果我们不断的链式调用`then`方法，然后这里每个`then`方法，都是为上一个`then`方法返回的`Promise`对象添加状态明确后的回调。

```
<!DOCTYPE html>
<html lang="en">
```

```
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Promise链式调用</title>
  </head>
  <body>
    <script>
      let getJson = function (url) {
        let p = new Promise(function (resolve, reject) {
          let xhr = new XMLHttpRequest();
          xhr.open("GET", url);
          xhr.onreadystatechange = handler;
          xhr.responseType = "json";
          xhr.setRequestHeader("Accept", "application/json");
          xhr.send();
          function handler() {
            if (xhr.readyState === 4) {
              if (xhr.status == 200) {
                resolve(this.response);
              } else {
                reject(new Error(this.statusText));
              }
            }
          }
        });
        return p;
      };
      var promise = getJson("http://localhost:3005/products");
      var promise2 = promise.then(
        function (result) {
          console.log(result);
        },
        function (err) {
          console.log("出错了:" + err);
        }
      );
      console.log("promise2=", promise2);
      console.log(promise === promise2);
    </script>
  </body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Promise链式调用</title>
  </head>
  <body>
    <script>
      let getJson = function (url) {
```

```
        let p = new Promise(function (resolve, reject) {
          let xhr = new XMLHttpRequest();
          xhr.open("GET", url);
          xhr.onreadystatechange = handler;
          xhr.responseType = "json";
          xhr.setRequestHeader("Accept", "application/json");
          xhr.send();
          function handler() {
            if (xhr.readyState === 4) {
              if (xhr.status == 200) {
                resolve(this.response);
              } else {
                reject(new Error(this.statusText));
              }
            }
          }
        });
        return p;
      };
      getJson("http://localhost:3005/products")
        .then(function (value) {
          console.log(value);
          console.log("111");
          return getJson("http://localhost:3005/cart");
        })
        .then(function (value) {
          console.log("then2=", value);
          console.log("222");
        })
        .then(function (value) {
          console.log(value);
          console.log("333");
        })
        .then(function (value) {
          console.log("444");
          return "abc";
        })
        .then(function (value) {
          console.log("555555");
          console.log("then5=", value);
        });
    </script>
  </body>
</html>
```

4、Promise异常处理

如果Promise执行结果失败，会调用我们所为其添加的onRejected回调函数。

```
var promise=ajax('/api/users.json')
var promise2=promise.then(function onFulfilled(value){
    console.log('onFulfilled',value)
},function onRejected(error){
    console.log('onRejected',error)
})
```

例如，我们请求了不存在地址，或者是我们在`ajax`方法内部出现了异常（`throw new Error()`），都会执行`onRejected`函数。

所以说`onRejected`就是处理`Promise`中的异常。当然关于异常处理，我们还有另外一种用户就是使用`Promise`对象的`catch`方法来完成。

下面，我们来实现以下

```
ajax('/api/users.json').then(function onFulfilled(value){
    console.log('onFulfilled',value)
}).catch(function onRejected(error){
    console.log('onRejected',error)
})
```

在上面的代码中，使用`then`注册了成功的回调，使用`catch`来处理异常。

其实这个`catch`方法就是`then`方法的别名。

5、Promise并行执行

例如，一个页面中有可能会与遇到多个请求服务端接口的情况，而这些请求之间没有相互的依赖关系。

那最好的选择就是同时请求服务端，避免一个一个的请求，而消耗过多的时间。

当然，你可能会说，这个实现起来非常的简单啊，把我们前面所写的`ajax`函数，多调用几次就可以了，如下所示：

```
ajax('/api/users.json')
ajax('/api/posts.json')
```

但是问题是，我们怎么知道所有的请求都结束了呢?

当然，你可能会说，我们定义一个计数器，每个请求结束后，让这个计数器累加一下，当累加的个数，与我们的任务数相同后，就表示所有的任务结束了。

这种方式比较麻烦。为了解决这个问题，`Promise`中提供了一个`all`方法。该方法接收的是一个数组，数组中的每个元素都是一个`Promise`

对象。我们可以把这些`Promise`对象，看作是一个一个的异步任务。`all`方法会返回一个全新的`Promise`对象。当`all`方法内部所有的`Promise`对象都执行完毕后，这时我们才会获取到`all`方法所返回的新的`Promise`对象。该`Promise`对象获取到的结果是一个数组。在这个数组中包含了每个异步任务执行的结果。

需要注意的就是`all`方法中所有`Promise`对象都执行成功了，才表示成功，只要有一个失败了，那么`all`方法的执行就失败了。

```
let promise1 = new Promise(function(resolve, reject) {
            setTimeout(function() {
                let num = Math.random();
                if (num > 0.3) {
                    resolve('成功了!')
                } else {
                    reject('失败了1')
                }
            }, 3000)
        })
        let promise2 = new Promise(function(resolve, reject) {
            setTimeout(function() {
                let num = Math.random();
                if (num > 0.3) {
                    resolve('成功了!')
                } else {
                    reject('失败了2')
                }
            }, 3000)
        })
      Promise.all([promise1, promise2]).then(function(data) {
            console.log(data);
        })
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Promise并行执行</title>
  </head>
  <body>
    <script>
      let getJson = function (url) {
        let p = new Promise(function (resolve, reject) {
          let xhr = new XMLHttpRequest();
          xhr.open("GET", url);
          xhr.onreadystatechange = handler;
          xhr.responseType = "json";
          xhr.setRequestHeader("Accept", "application/json");
          xhr.send();
          function handler() {
            if (xhr.readyState === 4) {
              if (xhr.status == 200) {
                resolve(this.response);
              } else {
                reject(new Error(this.statusText));
              }
            }
          }
        });
        return p;
      };
      Promise.all([
```

```
        getJson("http://localhost:3005/products"),
        getJson("http://localhost:3005/cart"),
        getJson("http://localhost:3005/ddd").catch(() => {}),
      ])
        .then((response) => {
          console.log(response);
        })
        .catch((err) => {
          console.log(err);
        });
    </script>
  </body>
</html>
```

6、Promise.race( )

与`all()`方法的区别是：

`Promise.all( )`是等待所有任务结束后才会结束。

‘Promise.race( )’只要有一个任务完成就结束。

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Promise中race方法</title>
  </head>
  <body>
    <script>
      let promise1 = new Promise(function (resolve, reject) {
        setTimeout(function () {
          let num = Math.random();
          if (num > 0.3) {
            resolve("成功了1!");
          } else {
            reject("失败了1");
          }
        }, 3000);
      });
      let promise2 = new Promise(function (resolve, reject) {
        setTimeout(function () {
          let num = Math.random();
          if (num > 0.3) {
            resolve("成功了2!");
          } else {
            reject("失败了2");
          }
        }, 3000);
      });
      let p = Promise.race([promise1, promise2])
        .then(function (data) {
```

```
          console.log(data);
        })
        .catch((err) => {
          console.log(err);
        });
    </script>
  </body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Promise中race方法</title>
  </head>
  <body>
    <script>
      let getJson = function (url) {
        let p = new Promise(function (resolve, reject) {
          let xhr = new XMLHttpRequest();
          xhr.open("GET", url);
          xhr.onreadystatechange = handler;
          xhr.responseType = "json";
          xhr.setRequestHeader("Accept", "application/json");
          xhr.send();
          function handler() {
            if (xhr.readyState === 4) {
              if (xhr.status == 200) {
                resolve(this.response);
              } else {
                reject(new Error(this.statusText));
              }
            }
          }
        });
        return p;
      };
      var promise = getJson("http://localhost:3005/products");
      const timeout = new Promise(function (resolve, reject) {
        setTimeout(() => reject(new Error("timeout")), 100);
      });
      Promise.race([promise, timeout])
        .then((value) => {
          console.log(value);
        })
        .catch((error) => {
          console.log(error);
        });
    </script>
  </body>
</html>
```

7、Promise静态方法

在Promise中还有几个静态方法也会使用到。

第一个 是Promise.resolve()

其作用就是将一个值，快速的转换成Promise对象。

```
Promise.resolve('foo').then(function(value){
    cosnole.log(value)
})//返回一个成功的Promise对象
```

第二个为Promise.reject()方法，该方法创建一个失败的Promise对象。

```
Promise.reject(new Error('rejected')).catch(function(error){
    console.log(error)
})
```

8、Promise执行顺序的问题

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Promise执行顺序问题</title>
  </head>
  <body>
    <script>
      console.log("start");
      setTimeout(() => {
        console.log("setTimeout");
      }, 0);
      Promise.resolve()
        .then(() => {
          console.log("promise");
        })
        .then(() => {
          console.log("promise2");
        })
        .then(() => {
          console.log("promise3");
        });
      console.log("end");
    </script>
  </body>
</html>
```

9、模拟Promise对象

1、搭建基本结构

```
<script>
          function MyPromise(task){
              let that = this;
              that.status='Pending';
              function resolve(){

              }
              function reject(){

              }
              task(resolve,reject);

          }
          let myPromise =new MyPromise(function(resolve,reject){


          })
      </script>
```

2、异常处理

```
<script>
      function MyPromise(task) {
        let that = this;
        that.status = "Pending";
        function resolve() {}
        function reject() {
          if (that.status === "Pending") {
            that.status = "Rejected";
            //状态修改完成后，调用的是then 方法中处理失败的回调函数
          }
        }
        try {
          task(resolve, reject);
        } catch (e) {
          reject(e);
        }
      }
      MyPromise.prototype.then = function (onFulfilled, onRejected) {};
      let myPromise = new MyPromise(function (resolve, reject) {});
    </script>
```

3、then方法处理与基本测试

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>构建自己的Promise对象</title>
```

```
</head>
<body>
  <script>
    function MyPromise(task) {
      let that = this;
      that.status = "Pending";
      that.value = undefined;
      that.onResolvedCallbacks = [];
      that.onRejectedCallbacks = [];
      function resolve(value) {
        if (that.status === "Pending") {
          that.status = "Resolved";
          that.value = value;
          that.onResolvedCallbacks.forEach((item) => item(that.value));
        }
      }
      function reject(reason) {
        if (that.status === "Pending") {
          that.status = "Rejected";
          that.value = reason;
          //状态修改完成后，调用的是then 方法中处理失败的回调函数
          that.onRejectedCallbacks.forEach((item) => item(that.value));
        }
      }
      try {
        task(resolve, reject);
      } catch (e) {
        reject(e);
      }
    }
    MyPromise.prototype.then = function (onFulfilled, onRejected) {
      let that = this;
      that.onResolvedCallbacks.push(onFulfilled);
      that.onRejectedCallbacks.push(onRejected);
    };
    let myPromise = new MyPromise(function (resolve, reject) {
      setTimeout(function () {
        let num = Math.random();
        if (num > 0.3) {
          resolve("成功了");
        } else {
          reject("失败了");
        }
      }, 3000);
    });
    myPromise.then(
      function (value) {
        console.log(value);
      },
      function (reason) {
        console.log(reason);
      }
    );
  </script>
</body>
```

```
</html>
```

4、完善操作

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>构建自己的Promise对象</title>
  </head>
  <body>
    <script>
      function MyPromise(task) {
        let that = this;
        that.status = "Pending";
        that.value = undefined;
        that.onResolvedCallbacks = [];
        that.onRejectedCallbacks = [];
        function resolve(value) {
          if (that.status === "Pending") {
            that.status = "Resolved";
            that.value = value;
            that.onResolvedCallbacks.forEach((item) => item(that.value));
          }
        }
        function reject(reason) {
          if (that.status === "Pending") {
            that.status = "Rejected";
            that.value = reason;
            //状态修改完成后，调用的是then 方法中处理失败的回调函数
            that.onRejectedCallbacks.forEach((item) => item(that.value));
          }
        }
        try {
          task(resolve, reject);
        } catch (e) {
          reject(e);
        }
      }
      MyPromise.prototype.then = function (onFulfilled, onRejected) {
        let that = this;
        if (that.status === "Resolved") {
          onFulfilled(that.value);
        }
        if (that.status === "Rejected") {
          onRejected(that.value);
        }
        that.onResolvedCallbacks.push(onFulfilled);
        that.onRejectedCallbacks.push(onRejected);
      };
      let myPromise = new MyPromise(function (resolve, reject) {
        // setTimeout(function () {
```

```
        //   let num = Math.random();
        //   if (num > 0.3) {
        //     resolve("成功了");
        //   } else {
        //     reject("失败了");
        //   }
        // }, 3000);
        resolve("成功了");
      });
      myPromise.then(
        function (value) {
          console.log(value);
        },
        function (reason) {
          console.log(reason);
        }
      );
    </script>
  </body>
</html>
```

三、async函数

1、常见异步编程方式

1.2.1 回调函数

JavaScript 语言对异步编程的实现，就是回调函数。**所谓回调函数，就是把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，就直接调用这个函数。**它的英语名字 callback，直译过来就是"重新调用"。

```
fs.readFile('/etc/passwd', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

上面代码中，readFile 函数的第二个参数，就是回调函数，也就是任务的第二段。等到操作系统返回了 /etc/passwd 这个文件以后，回调函数才会执行。

1.2.2 Promise对象

回调函数本身并没有问题，它的问题出现在多个回调函数嵌套。假定读取A文件之后，再读取B文件，代码如下。

```
fs.readFile(fileA, function (err, data) {
  fs.readFile(fileB, function (err, data) {
    // ...
  });
});
```

不难想象，如果依次读取多个文件，就会出现多重嵌套 .这样就产生了，我们前面讲解的回调地狱问题。

而Promise对象就是为了解决这个问题。

```
readFile(fileA)
.then(function(data){
  console.log(data.toString());
})
.then(function(){
  return readFile(fileB);
})
.then(function(data){
  console.log(data.toString());
})
.catch(function(err) {
  console.log(err);
});
```

通过Promise解决了回调地狱的问题。

1.2.3 Generator函数

Generator函数,就是一个封装的异步任务，或者说是异步任务的容器。异步操作需要暂停的地方，都用到yield语句。

下面的案例是前面用Generator函数封装的AJAX的异步操作。

```
function* main() {
  let result = yield request("http://xxx.com/api");
  let resp = JSON.parse(result);
    console.log(resp.value);
}

function request(url) {
  makeAjaxCall(url, function(response){
    it.next(response);
  });
}

let it = main();
it.next();
```

2、async函数

2.1 基本用法

```
        async function test() {
            let result = await Math.random();
            console.log(result);
        }
        test();
```

async: 表示函数中有异步操作,await 必须出现在 async 函数内部，不能单独使用。

await: 表示紧跟在后面的表达式需要等待结果。一般情况下await后面跟的是一个耗时的操作或者一个异步的操作。

2.2 使用方式

```
<script>
    function sleep(second) {
      return new Promise(function (resolve, reject) {
        setTimeout(function () {
          let num = Math.random();
          if (num > 0.8) {
            resolve("成功了");
          } else {
            reject("失败了");
          }
        }, second);
      });
    }
    async function awaitDemo() {
      let result = await sleep(3000);
      return result;
    }
    awaitDemo()
      .then(function (data) {
        console.log("data=", data);
      })
      .catch(function (err) {
        console.log("error=", err);
      });
    console.log("执行其它的代码");
  </script>
```

2.3 处理异步请求

```
<script>
      let getJson = function (url) {
        let p = new Promise(function (resolve, reject) {
          let xhr = new XMLHttpRequest();
          xhr.open("GET", url);
          xhr.onreadystatechange = handler;
          xhr.responseType = "json";
          xhr.setRequestHeader("Accept", "application/json");
          xhr.send();
          function handler() {
            if (xhr.readyState === 4) {
              if (xhr.status == 200) {
                resolve(this.response);
              } else {
                reject(new Error(this.statusText));
              }
            }
          }
        });
        return p;
      };
      async function getAjax() {
        try {
```

```
        let result = await getJson("http://localhost:3005/products");
        console.log(result);
      } catch (err) {
        console.log(err);
      }
    }
    getAjax();
  </script>
```

2.4 请求依赖关系的处理

```
  <script>
    function sleep(second, param) {
      return new Promise((resolve, reject) => {
        setTimeout(() => {
          resolve(param);
        }, second);
      });
    }
    async function test() {
      let result1 = await sleep(2000, "req01");
      let result2 = await sleep(1000, "req02" + result1);
      let result3 = await sleep(500, "req03" + result2);
      console.log(result1, result2, result3);
    }
    test();
  </script>
```

2.5 并且处理的问题

```
<script>
    let getJSON = function (url) {
      let p = new Promise(function (resolve, reject) {
        let xhr = new XMLHttpRequest();
        xhr.open("GET", url);
        xhr.onreadystatechange = handler;
        xhr.responseType = "json";
        xhr.setRequestHeader("Accept", "application/json");
        xhr.send();

        function handler() {
          if (xhr.readyState === 4) {
            if (this.status === 200) {
              resolve(this.response);
            } else {
              reject(new Error(this.statusText));
            }
          }
        }
      });
      //返回Promise对象
      return p;
    };
    async function getAJAX() {
```

```
      // try {
      //   let result = await getJSON("http://localhost:3005/products");
      //   let result1 = await getJSON("http://localhost:3005/products");
      //   let result2 = await getJSON("http://localhost:3005/products");
      //   console.log(result, result1, result2);
      //   console.log("clear the loading~"); //通过这一句代码模拟隐藏loading图片
      // } catch (e) {
      //   console.log(e);
      // }
      try {
        let result = getJSON("http://localhost:3005/products");
        let result1 = getJSON("http://localhost:3005/products");
        let result2 = getJSON("http://localhost:3005/products");
        let p = await Promise.all([result, result1, result2]);
        console.log(p);
        console.log("clear the loading~");
      } catch (e) {
        console.log(e);
      }
    }
    getAJAX();
  </script>
```