link: null
title: 珠峰架构师成长计划
description: 在需要多个操作的时候，会导致多个回调函数嵌套，导致代码不够直观，就是常说的回调地狱
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=73 sentences=167, words=1703

1. 异步回调

在需要多个操作的时候，会导致多个回调函数嵌套，导致代码不够直观，就是常说的回调地狱

如果几个异步操作之间并没有前后顺序之分,但需要等多个异步操作都完成后才能执行后续的任务，无法实现并行节约时间

2. Promise

Promise本意是承诺，在程序中的意思就是承诺我 **过一段时间后** 会给你一个结果。 什么时候会用到 **过一段时间**？答案是异步操作，异步是指可能比较长时间才有结果的才做，例如网络请求、读取本地文件等

3. Promise的三种状态

- Pending Promise对象实例创建时候的初始状态
- Fulfilled 可以理解为成功的状态
- Rejected 可以理解为失败的状态

> then 方法就是用来指定Promise 对象的状态改变时确定执行的操作，resolve 时执行第一个函数（onFulfilled），reject 时执行第二个函数（onRejected）

4. 构造一个Promise

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        if(Math.random()>0.5)
            resolve('This is resolve!');
        else
            reject('This is reject!');
    }, 1000);
});
promise.then(Fulfilled,Rejected)
```

- 构造一个Promise实例需要给Promise构造函数传入一个函数。
- 传入的函数需要有两个形参，两个形参都是function类型的参数。
    - 第一个形参运行后会让Promise实例处于resolve状态，所以我们一般给第一个形参命名为resolve,使 Promise 对象的状态改变成成功，同时传递一个参数用于后续成功后的操作
    - 第一个形参运行后会让Promise实例处于reject状态，所以我们一般给第一个形参命名为reject,将 Promise 对象的状态改变为失败，同时将错误的信息传递到后续错误处理的操作

```
function Promise(fn) {
    fn((data)=> {
        this.success(data);
    }, (error)=> {
        this.error();
    });
}

Promise.prototype.resolve = function (data) {
    this.success(data);
}

Promise.prototype.reject = function (error) {
    this.error(error);
}

Promise.prototype.then = function (success, error) {
    this.success = success;
    this.error = error;
}
```

```
class Promise {
    constructor(fn) {
        fn((data)=> {
            this.success(data);
        }, (error)=> {
            this.error();
        });
    }

    resolve(data) {
        this.success(data);
    }

    reject(error) {
        this.error(error);
    }

    then(success, error) {
        this.success = success;
        this.error = error;
        console.log(this);
    }
}
```

5. promise 做为函数的返回值

```
function ajaxPromise (queryUrl) {
  return new Promise((resolve, reject) => {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', queryUrl, true);
    xhr.send(null);
    xhr.onreadystatechange = () => {
      if (xhr.readyState === 4) {
        if (xhr.status === 200) {
          resolve(xhr.responseText);
        } else {
          reject(xhr.responseText);
        }
      }
    }
  });
}

ajaxPromise('http://www.baidu.com')
  .then((value) => {
    console.log(value);
  })
  .catch((err) => {
    console.error(err);
  });
```

6.promise的链式调用

- 每次调用返回的都是一个新的Promise实例
- 链式调用的参数通过返回值传递

then可以使用链式调用的写法原因在于，每一次执行该方法时总是会返回一个 Promise对象

```
readFile('1.txt').then(function (data) {
    console.log(data);
    return data;
}).then(function (data) {
    console.log(data);
    return readFile(data);
}).then(function (data) {
    console.log(data);
}).catch(function(err){
 console.log(err);
});
```

7.promise API

- **参数**：接受一个数组，数组内都是 Promise实例
- **返回值**：返回一个 Promise实例，这个 Promise实例的状态转移取决于参数的 Promise实例的状态变化。当参数中所有的实例都处于 resolve状态时，返回的 Promise实例会变为 resolve状态。如果参数中任意一个实例处于 reject状态，返回的 Promise实例变为 reject状态。

```
Promise.all([p1, p2]).then(function (result) {
  console.log(result);
});
```

> 不管两个promise谁先完成，Promise.all 方法会按照数组里面的顺序将结果返回

- **参数**：接受一个数组，数组内都是 Promise实例
- **返回值**：返回一个 Promise实例，这个 Promise实例的状态转移取决于参数的 Promise实例的状态变化。当参数中任何一个实例处于 resolve状态时，返回的 Promise实例会变为 resolve状态。如果参数中任意一个实例处于 reject状态，返回的 Promise实例变为 reject状态。

```
Promise.race([p1, p2]).then(function (result) {
  console.log(result);
});
```

7.3 Promise.resolve返回一个 Promise实例，这个实例处于 resolve状态。

根据传入的参数不同有不同的功能：

- 值(对象、数组、字符串等)：作为 resolve传递出去的值
- Promise实例：原封不动返回

返回一个 Promise实例，这个实例处于 reject状态。

- 参数一般就是抛出的错误信息。

8. q

Q是一个在Javascript中实现promise的模块

```
var Q = require('q');
var fs = require('fs');
function read(filename) {
    var deferred = Q.defer();
    fs.readFile(filename,'utf8', function (err, data) {
        if(err){
            deferred.reject(err);
        }else{
            deferred.resolve(data);
        }
    });
    return deferred.promise;
}

read('1.txt1').then(function(data){
    console.log(data);
},function(error){
    console.error(error);
});
```

```
module.exports = {
    defer(){
        var _success,_error;
        return {
            resolve(data){
                _success(data);
            },
            reject(err){
                _error(err);
            },
            promise:{
                then(success,error){
                    _success = success;
                    _error = error;
                }
            }
        }
    }
}
```

```
var defer = function () {
    var pending = [], value;
    return {
        resolve: function (_value) {
            if (pending) {
                value = _value;
                for (var i = 0, ii = pending.length; i < ii; i++) {
                    var callback = pending[i];
                    callback(value);
                }
                pending = undefined;
            }
        },
        promise: {
            then: function (callback) {
                if (pending) {
                    pending.push(callback);
                } else {
                    callback(value);
                }
            }
        }
    };
};
```

9. bluebird

实现 promise 标准的库是功能最全，速度最快的一个库

```
var Promise = require('./bluebird');

var readFile = Promise.promisify(require("fs").readFile);
readFile("1.txt", "utf8").then(function(contents) {
    console.log(contents);
})

var fs = Promise.promisifyAll(require("fs"));

fs.readFileAsync("1.txt", "utf8").then(function (contents) {
    console.log(contents);
})
```

```
module.exports = {
    promisify(fn){
        return function () {
            var args = Array.from(arguments);
            return new Promise(function (resolve, reject) {
                fn.apply(null, args.concat(function (err) {
                    if (err) {
                        reject(err);
                    } else {
                        resolve(arguments[1])
                    }
                }));
            })
        }
    },
    promisifyAll(obj){
        for(var attr in obj){
            if(obj.hasOwnProperty(attr) && typeof obj[attr] =='function'){
                obj[attr+'Async'] = this.promisify(obj[attr]);
            }
        }
        return obj;
    }
}
```

10. 动画

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>movetitle>
    <style>
        .square{
            width:40px;
            height:40px;
            border-radius: 50%;
        }
        .square1{
            background-color: red;
        }
        .square2{
            background-color: yellow;
        }
        .square3{
            background-color: blue;
        }
    style>
head>
<body>
<div class="square square1" style="margin-left: 0">div>
<div class="square square2" style="margin-left: 0">div>
<div class="square square3" style="margin-left: 0">div>
body>
<script>
  var square1 = document.querySelector('.square1');
  var square2 = document.querySelector('.square2');
  var square3 = document.querySelector('.square3');

  function move(element,target,resolve){
    let current = 0;
    let times = setInterval(function(){
      element.style.transform='translateX('+(++current)+'px)';
      if(current>=target){
        clearInterval(times);
        resolve();
      }
    },10);
  }
  function animate(element,target){
    return new Promise(function(resolve,reject){
      move(element,target,resolve);
    });
  }
  animate(square1,100)
    .then(function(){
      return animate(square2,100);
    })
    .then(function(){
      return animate(square3,100);
    });
script>
html>
```

11. co

```
let fs = require('fs');
function getNumber(){
  return new Promise(function (resolve,reject) {
    setTimeout(function(){
      let number = Math.random();
      if(number >.5){
        resolve(number);
      }else{
        reject('数字太小');
      }
    },1000);
  });
}
function *read(){
  let a = yield getNumber();
  console.log(a);
  let b = yield 'b';
  console.log(b);
  let c = yield getNumber();
  console.log(c);
}

function co(gen){
  return new Promise(function(resolve,reject){
    let g = gen();
    function next(lastValue){
      let {done,value} = g.next(lastValue);
      if(done){
         resolve(lastValue);
      }else{
        if(value instanceof Promise){
          value.then(next,function(val){
            reject(val);
          });
        }else{
          next(value);
        }
      }
    }
    next();
  });
}
co(read).then(function(data){
  console.log(data);
},function(reason){
  console.log(reason);
});
```

```
let fs = require('fs');
function readFile(filename){
  return new Promise(function (resolve,reject) {
    fs.readFile(filename,'utf8',function(err,data){
      if(err)
        reject(err);
      else
        resolve(data);
    })
  });
}
function *read(){
  let a = yield readFile('./1.txt');
  console.log(a);
  let b = yield readFile('./2.txt');
  console.log(b);
}

function co(gen){
  let g = gen();
  function next(val){
    let {done,value} = g.next(val);
    if(!done){
      value.then(next);
    }
  }
  next();
}
```

12. Promise/A+完整实现

```
function Promise(executor) {
  let self = this;
  self.status = "pending";
  self.value = undefined;
  self.onResolvedCallbacks = [];
  self.onRejectedCallbacks = [];
  function resolve(value) {
    if (value instanceof Promise) {
      return value.then(resolve, reject)
    }
    setTimeout(function () { // &#x5F02;&#x6B65;&#x6267;&#x884C;&#x6240;&#x6709;&#x7684;&#x56DE;&#x8C03;&#x51FD;&#x6570;
      if (self.status == 'pending') {
        self.value = value;
        self.status = 'resolved';
        self.onResolvedCallbacks.forEach(item => item(value));
      }
    });

  }

  function reject(value) {
    setTimeout(function () {
      if (self.status == 'pending') {
        self.value = value;
```

```
        self.status = 'rejected';
        self.onRejectedCallbacks.forEach(item => item(value));
      }
    });
  }

  try {
    executor(resolve, reject);
  } catch (e) {
    reject(e);
  }
}
function resolvePromise(promise2, x, resolve, reject) {
  if (promise2 === x) {
    return reject(new TypeError('&#x5FAA;&#x73AF;&#x5F15;&#x7528;'));
  }
  let then, called;

  if (x != null && ((typeof x == 'object' || typeof x == 'function'))) {
    try {
      then = x.then;
      if (typeof then == 'function') {
        then.call(x, function (y) {
          if (called)return;
          called = true;
          resolvePromise(promise2, y, resolve, reject);
        }, function (r) {
          if (called)return;
          called = true;
          reject(r);
        });
      } else {
        resolve(x);
      }
    } catch (e) {
      if (called)return;
      called = true;
      reject(e);
    }
  } else {
    resolve(x);
  }
}
Promise.prototype.then = function (onFulfilled, onRejected) {
  let self = this;
  onFulfilled = typeof onFulfilled == 'function' ? onFulfilled : function (value) {
    return value
  };
  onRejected = typeof onRejected == 'function' ? onRejected : function (value) {
    throw value
  };
  let promise2;
  if (self.status == 'resolved') {
    promise2 = new Promise(function (resolve, reject) {
      setTimeout(function () {
        try {
          let x = onFulfilled(self.value);
          resolvePromise(promise2, x, resolve, reject);
        } catch (e) {
          reject(e);
        }
      });

    });
  }
  if (self.status == 'rejected') {
    promise2 = new Promise(function (resolve, reject) {
      setTimeout(function () {
        try {
          let x = onRejected(self.value);
          resolvePromise(promise2, x, resolve, reject);
        } catch (e) {
          reject(e);
        }
      });
    });
  }
  if (self.status == 'pending') {
    promise2 = new Promise(function (resolve, reject) {
      self.onResolvedCallbacks.push(function (value) {
        try {
          let x = onFulfilled(value);
          resolvePromise(promise2, x, resolve, reject);
        } catch (e) {
          reject(e);
        }
      });
      self.onRejectedCallbacks.push(function (value) {
        try {
          let x = onRejected(value);
          resolvePromise(promise2, x, resolve, reject);
        } catch (e) {
          reject(e);
        }
      });
    });
  }
  return promise2;
}
Promise.prototype.catch = function (onRejected) {
  return this.then(null, onRejected);
}
Promise.all = function (promises) {
  return new Promise(function (resolve, reject) {
```

```
    let result = [];
    let count = 0;
    for (let i = 0; i < promises.length; i++) {
      promises[i].then(function (data) {
        result[i] = data;
        if (++count == promises.length) {
          resolve(result);
        }
      }, function (err) {
        reject(err);
      });
    }
  });
}

Promise.deferred = Promise.defer = function () {
  var defer = {};
  defer.promise = new Promise(function (resolve, reject) {
    defer.resolve = resolve;
    defer.reject = reject;
  })
  return defer;
}
/**
 * npm i -g promises-aplus-tests
 * promises-aplus-tests Promise.js
 */
try {
  module.exports = Promise
} catch (e) {
}
```

13. 资源

1.自己实现promise的all、race、resolve和reject方法