Promise

预习资料

名称	链接	备注
promise 基础	https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise	
promises A+规范	https://github.com/promises-aplus/promises-spec	
promises A+规范测试工具用例工具	https://github.com/promises-aplus/promises-tests	

一.promise规范

promise是什么?
promise对象表示一个异步操作的最终完成(或失败)及其结果值 (现在主流解决异步编程的方案)

promise是一个原生对象，下面我们就看下 promise的ECMA规范，查看promise的规范，能够更清楚的了解Promise原理

Promises/A+术语

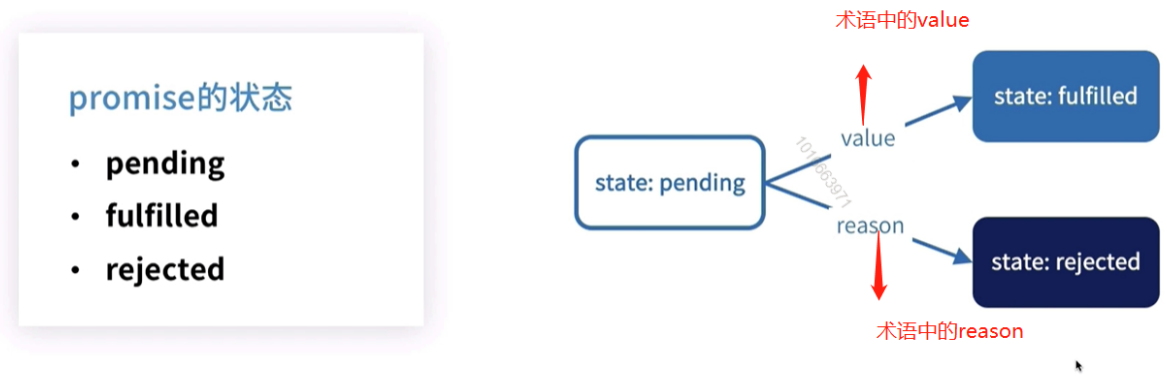
术语

- **promise** 一个有then方法的对象或函数，行为符合本规范
- **thenable** 一个定义了then方法的对象或函数
- **值，value** 任何JavaScript的合法值
- **异常，exception** throw语句抛出的值
- **拒绝原因，reason** 一个标示promise被拒绝原因的值

我们先知道有这些相关术语需要用到，先不对术语进行一一讲解了。
术语里面我们给promise下了一定义，promise是一个对象，或者一个函数。promise的状态做了一定规范，它有三个状态

1. pending
2. fulfilled
3. rejected

如图所示：



promise从pending状态转化成fulfilled状态需要术语中value值
promise从pending状态转化成rejected状态需要术语reason值
当promise转化成fulfilled或者rejected状态的时候，就稳定了，任何状态都不能改变这个promise的状态了。
所以我们得出的结论是：一个promise的状态一旦改变了就不能再改变了

Promises/A+要求
Promise需要有个then方法的

1. then方法有两个参数，onFulfilled和OnRejected 是两个函数。
2. onFulfilled在promise完成后被调用，onRejected在promise被拒绝执行后调用
3. 只被调用一次，要不是onFulfilled，要么是onRejected
4. then方法可以调用多次，onFulfilled和OnRejected就会多次注册，按照注册的顺序执行
5. then方法的返回值是一个promise对象，这样才能链式调用

现在问题来了，竟然现在then方法要返回一个promise对象,那么返回的promise对象的值和状态是怎么确定的了？

```
const promise1 = new Promise(function (resolve, reject) {})
 // Promise构造函数执行时立即调用executor 函数，
 // resolve 和 reject 两个函数作为参数传递给executor
 // (executor 函数在Promise构造函数返回所建promise实例对象前被调用)
```

```
const promise2 = promise1.then(onFulfilled,OnRejected)
```

我们假设then方法返回的是上图的promise2，规范分了三种情况，我们根据这三种情况来确定这个promise2的值和状态

第一种情况：

代码示例

```
let promise1 = new Promise((resolve, reject) => {
  // code
  resolve('success')
  // code
//   reject('failed')
})

let promise2 = promise1.then({a:1})  //不是一个函数，是一个对象的时候
undefined
promise2
▼Promise {<fulfilled>: "success"}
  ▶__proto__: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: "success"
```

value和promise1相同

第二种情况

代码示例

```
> let promise1 = new Promise((resolve, reject) => {
    // code
  //   resolve('success')
    // code
    reject('failed')
  })

  let promise2 = promise1.then({a:1})  //不是一个函数,是一个对象的时候
  promise2
◟ ▼Promise {<rejected>: "failed"}
    ▶__proto__: Promise
     [[PromiseState]]: "rejected"
     [[PromiseResult]]: "failed"
⊗ ▶Uncaught (in promise) failed
```

和promise1的reason一致

第三种情况 (这种情况是比较复杂的)

onFulfilled或者OnRejected是函数同时它 返回值是一个 x 变量

1. x 为undefined的时，不做任何处理 和promise1状态一样 值为underfind

```
let promise1 = new Promise((resolve, reject) => {
    resolve(1)
})
let promise2 = promise1.then(
    value => {
        return undefined //默认为undefined
        // return 1
        // return Promise.resolve(2)
        // return Promise.reject(3)
        // throw 4
    })
promise2
▼Promise {<fulfilled>: undefined}
  ▶__proto__: Promise
   [[PromiseState]]: "fulfilled"
   [[PromiseResult]]: undefined
```

2. x 为非promise的值时，则新promise的状态fulfilled，值为return 的值

```
let promise1 = new Promise((resolve, reject) => {
    resolve(1)
})
let promise2 = promise1.then(
    value => {
        // return undefined //默认为undefined
        return 1
        // return Promise.resolve(2)
        // return Promise.reject(3)
        // throw 4
    })
promise2
```

```
▼Promise {<fulfilled>: 1}
  ▶__proto__: Promise
   [[PromiseState]]: "fulfilled"
   [[PromiseResult]]: 1
```

3. x 通过throw主动抛出错误或者代码出现错误，则promise的状态为rejected，值为throw的值

```
let promise1 = new Promise((resolve, reject) => {
    resolve(1)
})
let promise2 = promise1.then(
    value => {
        // return undefined //默认为undefined
        // return 1
        // return Promise.resolve(2)
        // return Promise.reject(3)
        throw 4
    })
promise2
```

```
▼Promise {<rejected>: 4}
  ▶__proto__: Promise
   [[PromiseState]]: "rejected"
   [[PromiseResult]]: 4
▼Uncaught (in promise) 4
 Promise.then (async)
 (anonymous)          @ VM385:5
```

4. x 通过return 返回一个promise对象，则新promise就是return的promsie

```
let promise1 = new Promise((resolve, reject) => {
    resolve(1)
})
let promise2 = promise1.then(
    value => {
        // return undefined //默认为undefined
        // return 1
        //return Promise.resolve(2)
         return Promise.reject(3)
//          throw 4
    })
promise2
▼Promise {<rejected>: 3}
  ▶__proto__: Promise
   [[PromiseState]]: "rejected"
   [[PromiseResult]]: 3
Uncaught (in promise) 3
```

当x为以上四种情况都是当promise1是reslove fulfilled状态。promise2的相应的值的状态如果promise1是rejected状态的话，promise2与promise1的值和状态是一样的

Promise.resolve()快速获取一个成功状态的promise对象

Promise.reject()快速获取一个失败状态的promise对象

下面上代码示例：

```
let  promise = Promise.resolve('你好帅哥').then(2).then(Promise.resolve('错误')).then(console.log)
你好帅哥
undefined
```

1. Promise快速获取一个成功状态的promise对象，value值是'你好帅哥'
2. then方法调用，根据then方法的相关规范，第一个then参数不是函数，直接忽略，value还是 ‘你好帅哥’ ，同理第二个then，也忽略，value还是 ‘你好帅哥’ ，第三个then参数一个函数。直接调用console.log 打印出value值'帅哥'

二.ES6 Promise API

1.ES6有个Promise构造函数

构造函数	说明

new Promise(function(resolve, reject) {
// resolve(value)
// reject(reason)
})
	函数作为参数
	resolve函数将promise状态从pending变成resolved(fulfilled)，
	reject函数将promise状态从pending变成rejected

new Promise(func) 这个构造函数接收一个function函数作为参数。function函数同时也接收两个函数作为参数，这个两个方法是内部实现的，我们直接调用即可。。resolve函数将promise状态从pending编程resolved(fulfilled)，reject函数将promise状态从pending变成rejected

2.静态方法

方法	说明
Promise.resolve(param)	等同于 new Promise(function(resolve, reject){resolve(param)})
Promise.reject(reason)	等同于 new Promise(function(resolve, reject){reject(reason)})
Promise.all([p1,…,pn])	输入一组promise返回一个新的promise，全部promise都是fulfilled结果才是fulfilled状态
Promise.allSettled([p1,…,pn])	输入一组promise返回一个新的promise，所有的promise状态改变后结果promise变成fulfilled状态
Promise.race([p1,…,pn])	输入一组promise返回一个新的promise，结果promise的状态跟随第一个变化的promise状态

Promise.resolve(param)和Promise.reject(reason) 在promise规范已经讲解过了，分别是快速获取一个成功状态的promise对象,一个失败状态的promise状态

Promise.resolve(value)方法返回一个以给定值解析后的Promise 对象。如果这个值是一个 promise ，那么将返回这个 promise ；如果这个值是thenable（即带有"then" 方法），返回的promise会“跟随”这个thenable的对象，采用它的最终状态；否则返回的promise将以此值完成。

Promsie.all([p1,p2,....pn]) all方法接收一个promise的iterable类型，Array，Map，Set都属于ES6的iterable类型，也就是图表中得一组promise,只返回一个Promise实例。全部promise都是fulfilled结果才是fulfilled状态，同时所有输入的promise的resolve回调是一个数组(resolve函数返回的值，组成数组)，我们可以通过then方法获取到reslove回调，如下面代码

```
> const promise1 = Promise.resolve({a:2});
  const promise2 = 42;
  const promise3 = new Promise((resolve, reject) => {
    setTimeout(resolve, 100, 'foo');
  });
   Promise.all([promise1, promise2, promise3]).then((value)=>{
        console.log(value)
  })


◂ ▶ Promise {<pending>}

  ▼ (3) [{…}, 42, "foo"]
    ▶ 0: {a: 2}
      1: 42
      2: "foo"
      length: 3
    ▶ __proto__: Array(0)
```

所有promise都是fulfilled结果，promise状态才是fulfilled，否则都会出现错误，如下面代码所示

```
> const promise1 = Promise.reject(3);
  const promise2 = 42;
  const promise3 = new Promise((resolve, reject) => {
    setTimeout(resolve, 100, 'foo');
  });

  Promise.all([promise1, promise2, promise3]).then((values) => {
    console.log(values);
  });
◂ ▼ Promise {<rejected>: 3}
    ▶ __proto__: Promise
      [[PromiseState]]: "rejected"
      [[PromiseResult]]: 3
  ▶ Uncaught (in promise) 3                                    all:1
>
```

Promsie.allSettled([p1,p2...pn]) 接收参数跟all一致，也只是返回一个Promise实例，不同的是，只要promise状态改变后，返回的新promise状态就变成了fulfilled状态，如下面代码所示

```
> const promise1 = Promise.reject({a:1});
  const promise2 = new Promise((resolve, reject) => setTimeout(resolve, 100, 'foo'));
  const promise3 = 3
  const promises = [promise1, promise2,promise3];

  Promise.allSettled(promises).
    then((results) => console.log(results));
◂ ▸ Promise {<pending>}
  ▾ (3) [{…}, {…}, {…}]
    ▾ 0:
      ▸ reason: {a: 1}
        status: "rejected"
      ▸ __proto__: Object
    ▾ 1:
        status: "fulfilled"
        value: "foo"
      ▸ __proto__: Object
    ▾ 2:
        status: "fulfilled"
        value: 3
      ▸ __proto__: Object
      length: 3
    ▸ __proto__: Array(0)
>
```

Promsie.race([p1,p2...pn]),接收参数也同all一致，也是返回一个Promise对象，不同的是，race顾名思义，是赛跑的意思。返回的新的promise对象的状态跟随最新开始变化的那个promise状态(一旦迭代器中的某个promise解决或拒绝，返回的 promise就会解决或拒绝。)

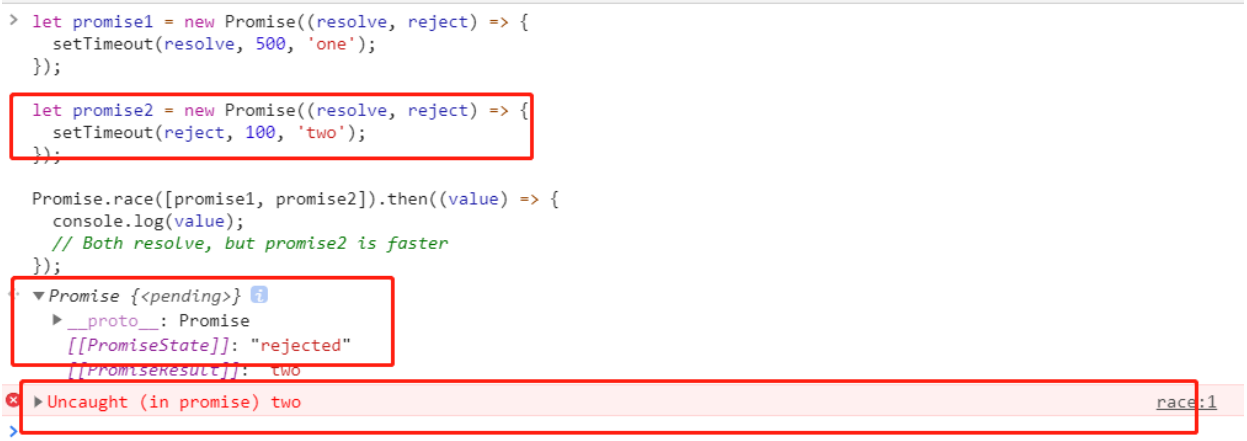
示例1：第一个开始变化的是 promise2 将变成fulfilled状态，返回的新promise状态也是fulfilled状态

```
> let promise1 = new Promise((resolve, reject) => {
    setTimeout(reject, 500, 'one');
  });

  let promise2 = new Promise((resolve, reject) => {
    setTimeout(resolve, 100, 'two');
  });

  Promise.race([promise1, promise2]).then((value) => {
    console.log(value);
    // Both resolve, but promise2 is faster
  });
◂ ▾ Promise {<pending>}
    ▸ __proto__: Promise
      [[PromiseState]]: "fulfilled"
      [[PromiseResult]]: undefined
  two
>
```

示例2:第一个开始变化的是promise2将变成reject状态，返回的新的promise状态也是reject状态。



3.实例方法

方法	说明
promise.then(onFulfilled,onRejected)	promise状态改变之后的回调，返回新的promise对象
promise.catch(function(reason){ })	同promise.then(null, onRejected)，promise状态为rejected的回调

promise.finally(function(reason){
// test
})
	同
	promise.then(function(){//test},function(){//test})，
	不管promise状态如何都会执行

上两点已经说过 Promise实例有then方法，它是promise状态改变之后的回调(onFulfilled,OnRejected)成功和失败的回调。第一个参数是成功函数的回调，第二个参数是失败函数的回调，回调只是执行两个中的一种，要么成功，要么失败。返回新的promise对象。

catch和finally 其实就是then方法在特殊场景下的使用
promise.catch((reason)=>{}) 如同promise.then(null,onRejected),promise状态为rejected的回调
promise.finally((reason)=>{})如同promise,then(function(){//test},function(){//test}),不管promise状态如何都会执行

使用ES6 Promise API要注意什么?

1. then和catch 返回的是新的promise对象，不是原来的promise对象
2. Promise对象的错误会'冒泡'，直到被获取为止，错误会被下一个catch捕获，所以我们在使用Promise链式调用的时候，我们只要在最后的一个使用catch捕获一下即

可

二.实践

我们在使用promise的过程中，有几点需要注意：

1. 不要忘记catch捕获错误(我在window注册onErr事件的时候window是可以监听到的，但是promise内部报错，不会触发window onErr,所以需要promise自己catch错误)
2. then方法中使用retrun(then方法返回的是一个新的promise对象，then中return就是新promsie对象的value值)
3. 传递函数给then方法(我们知道then方法可以接收非函数参数，在实际开发中传非函数参数没有多大的意义，所以的尽量传递函数给then方法，让其更加有意义)
4. 不要把promise写成嵌套，我们使用promise就是为了解决回调地狱，写成嵌套，让promise毫无意义了

用promise实现一下题目：

> 3秒之后亮红灯一次，再过2秒亮绿灯一次，再过1秒亮黄灯一次，用promise实现多次交替亮灯的效果（可以用console.log模拟亮灯）。

简单的一个有缺陷的代码一

```
1
2 //这是简单实现了 一组顺序的红灯 绿灯 黄灯 执行 这里只是执行了一次 ，没有达到多次执行。也不通用
3 Promise.resolve().then(()=>{
4  return new Promise((resolve,reject)=>{
5  setTimeout(()=>{
6  console.log('红色')
7  resolve()
8  },3000)
9  })
10 }).then(()=>{
11  return new Promise((resolve,reject)=>{
12  setTimeout(()=>{
13  console.log('绿色')
14  resolve()
15  },2000)
```

```
  })
}).then(()=>{
  return new Promise((resolve,reject)=>{
  setTimeout(()=>{
  console.log('黄色')
  resolve()
  },1000)
  })
})
```

核心思想: 通过Promise.reslove()静态方法，获取一个成功状态的promise对象。调用then方法，then返回一个 新的实例化Promise对象，并在其中写一个延迟三秒执行红灯的定时器，然后resolve()。确保红灯定期执行完毕后，再调用下一个then().返回新的promise对象。这样确保按照顺序执行相应的定时器,后面都以此类推.....

改装代码:封装成函数代码二

```
// 不同颜色的灯 执行间隔时间函数
function Lights (name,second){
   return new Promise((resolve,reject)=>{
         setTimeout(()=>{
            console.log(name)
            resolve()
          },second*1000)
    })
}
//顺序执行一组的 红 绿 黄
function orderLists(lights){
  let promise = Promise.resolve()
  lights.forEach(item =>{
         promise = promise.then(function(){
             return Lights(item.name,item.second)
         })
      })
}

orderLists([
    {name:'红',second:3},
    {name:'绿',second:2},
    {name:'黄',second:1}
])
```

```
undefined
红                                    VM12340:5
绿                                    VM12340:5
黄                                    VM12340:5
```

这只是顺序的执行的一次，要执行多次还得重新完善

完整代码

```
// 1.不同颜色的灯 执行间隔时间函数
function Lights (name,second){
    return new Promise((resolve,reject)=>{
          setTimeout(()=>{
             console.log(name)
             resolve()
           },second*1000)
    })
}
//2.顺序执行一组的 红 绿 黄
function orderLists(lights){
  let promise = Promise.resolve()
  lights.forEach(item =>{
          promise = promise.then(function(){
              return Lights(item.name,item.second)
          })
      })
//3.这里用到一个递归，实现多次执行 红绿黄灯
     promise.then(()=>{
        return orderLists(lights)
    })
}
orderLists([
    {name:'红',second:3},
    {name:'绿',second:2},
    {name:'黄',second:1}
])
```

代码流程分析:

1. 定义一个的灯的函数，返回时一个Promise实例，并在实例当中添加定时器，定时器执行完后后，resolve转化为成功状态。
2. 顺序执行相关定时器。这里的话，先通过Promise.resolve获取一个成功状态的promise对象，然后调用then方法，返回灯函数promise对象。这样就达到了，顺序执行各个定时器函数。
3. 这里要求不间断的灯交替，就得重复调用灯函数，得用到递归。

课后作业

根据Promise/A+规范实现promise，使用promises-aplus/promises-tests 插件验证。