函数对象、实例对象

- 函数对象：将函数作为对象使用时，简称为函数对象。
- 实例对象：new 构造函数或类产生的对象，我们称之为实例对象。

```
function Person (name,age){
    this.name = name
    this.age = age
}
Person.a = 1; //将Person看成一个对象
const p1 = new Person('老刘',18); //p1是Person的实例对象
console.log(p1);
//函数对象
function Person1 (){
}
Person1.name = 'Tom';
console.log(Person1.name );  //输出Person1 不是Tom 每个函数对象有个不可修改的属性name（函数名）
```

回调函数

- 什么是回调？——我们定义的，我们没有调用，最终执行了。
- 同步的回调函数
    - 理解：立即在主线程上执行, 不会放入回调队列中。
    - 例子：数组遍历相关的回调函数
- 异步的回调函数
    - 理解：不会立即执行, 会放入回调队列中，主线程代码执行完以后执行
    - 例子：定时器回调 / ajax回调

```
//同步的回调函数
let arr = [1,3,5,7,9]
arr.forEach((item)=>{
    console.log(item);
})
console.log('主线程的代码');
//异步的回调函数
setTimeout(()=>{
    console.log('@');
},2000)
console.log('主线程');
```

Error

- JS中的错误(Error)和错误处理
- mdn文档: https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Error
- 错误的类型

    - Error: 所有错误的父类型
        - ReferenceError: 引用的变量不存在
        - TypeError: 数据类型不正确
        - RangeError: 数据值不在其所允许的范围内--死循环
        - SyntaxError: 语法错误
- 错误处理
    - 捕获错误: try{}catch(){}
    - 抛出错误: throw error
- 错误对象
    - message属性: 错误相关信息
    - stack属性: 记录信息

```
//如何捕获一个错误
//try中放可能出现错误的代码，一旦出现错误立即停止try中代码的执行，调用catch，并携带错误信息
try {
    console.log(1);
    console.log(a);
    console.log(2);
} catch (error) {
    console.log('代码执行出错了,错误的原因是: ',error);
}
//如何抛出一个错误
function demo(){
    const date = Date.now()
    if(date % 2 === 0){
        console.log('偶数，可以正常工作');
    }else{
        throw new Error('奇数，不可以工作! ')
    }
}
try {
    demo()
} catch (error) {
    debugger;
    console.log('@',error);
}
```

Promise

简介

- Promise是什么?
    - 抽象表达:
        - Promise是一门新的技术（ES6提出的）
        - Promise是JS中异步编程的新方案（旧方案--纯回调）
    - 具体表达:
        - 从语法上来说：Promise是一个内置构造函数
        - 从功能上来说：Promise的实例对象可以用来封装一个异步操作，并可以获取其成功/失败的值

- Promise不是回调，是一个内置的构造函数，是程序员自己new调用的。
- new Promise的时候，要传入一个回调函数，它是同步的回调，会立即在主线程上执行，被称为executor函数
- 每一个Promise实例都有3种状态：初始化(pending)、成功(fulfilled)、失败(rejected)
- 每一个Promise实例在刚被new出来的那一刻，状态都是初始化(pending)
- executor函数会接收到2个参数，它们都是函数，分别用形参：resolve、reject接收
    - 调用resolve函数会：
        - 让Promise实例状态变为成功(fulfilled)
        - 可以指定成功的value
    - 调用reject函数会：
        - 让Promise实例状态变为失败(rejected)
        - 可以指定失败的reason。

```
//创建一个Promise实例对象
const p = new Promise((resolve, reject)=>{
    reject('ok')
})
console.log('@',p); //一般不把Promise实例做控制台输出
```

基本使用

- 重要语法
    - new Promise(executor)构造函数
    - Promise.prototype.then方法
- 基本编码流程
    - 创建Promise的实例对象(pending状态), 传入executor函数
    - 在executor中启动异步任务（定时器、ajax请求）
    - 根据异步任务的结果，做不同处理：
        - 如果异步任务成功了：
            - 我们调用resolve(value), 让Promise实例对象状态变为成功(fulfilled),同时指定成功的value
        - 如果异步任务失败了：
            - 我们调用reject(reason), 让Promise实例对象状态变为失败(rejected),同时指定失败的reason
    - 通过then方法为Promise的实例指定成功、失败的回调函数，来获取成功的value、失败的reason
        - 注意：then方法所指定的：成功的回调、失败的回调，都是异步的回调。
- 关于状态的注意点：
    - 三个状态:
        - pending: 未确定的------初始状态
        - fulfilled: 成功的------调用resolve()后的状态
        - rejected: 失败的-------调用reject()后的状态
    - 两种状态改变
        - pending ==> fulfilled
        - pending ==> rejected
    - 状态只能改变一次！！

- 一个promise指定多个成功/失败回调函数, 都会调用

```
const p = new Promise((resolve,reject)=>{
    //模拟一个异步任务
    /* setTimeout(()=>{
        resolve('我是成功的数据')
    },2000) */

    //真正开启一个异步任务
    const xhr = new XMLHttpRequest()
    xhr.onreadystatechange = ()=>{
        if(xhr.readyState === 4){
            //readyState为4代表接收完毕，接收的可能是：服务器返回的成功数据、服务器返回的错误
            if(xhr.status === 200) resolve(xhr.response)
            else reject('请求出错')
        }
    }
    xhr.open('GET','https://api.apiopen.top/getJoke')
    xhr.responseType = 'json'
    xhr.send()
})
p.then(
    (value)=>{console.log('成功了1',value);}, //成功的回调-异步
    (reason)=>{console.log('失败了1',reason);} //失败的回调-异步
)
console.log('@');
```

ajax封装

```
/*
    定义一个sendAjax函数，对xhr的get请求进行封装：
            1.该函数接收两个参数：url(请求地址)、data(参数对象)
            2.该函数返回一个Promise实例
                        (1).若ajax请求成功,则Promise实例成功,成功的value是返回的数据。
                        (2).若ajax请求失败,则Promise实例失败,失败的reason是错误提示。
*/
function sendAjax(url,data){
    return new Promise((resolve,reject)=>{
            //实例xhr
            const xhr = new XMLHttpRequest()
            //绑定监听
            xhr.onreadystatechange = ()=>{
                if(xhr.readyState === 4){
                    if(xhr.status >= 200 && xhr.status < 300) resolve(xhr.response);
                    else reject('请求出了点问题');
                }
            }
            //整理参数
            let str = ''
            for (let key in data){
                str += `${key}=${data[key]}&`
            }
```

```
            str = str.slice(0,-1)
            xhr.open('GET',url+'?'+str)
            xhr.responseType = 'json'
            xhr.send()
        })
}
const x = sendAjax('https://api.apiopen.top/getJoke',{page:1,count:2,type:'video'})
x.then(
    (data)=>{console.log('成功了',data);},
    (reason)=>{console.log('失败了',reason);}
)
```

- 纯回调

```
/*
    定义一个sendAjax函数，对xhr的get请求进行封装：
        1.该函数接收4个参数：url(请求地址)、data(参数对象)、success(成功的回调)、error(失败的回调)
*/
function sendAjax(url,data,success,error){
    //实例xhr
    const xhr = new XMLHttpRequest()
    //绑定监听
    xhr.onreadystatechange = ()=>{
        if(xhr.readyState === 4){
            if(xhr.status >= 200 && xhr.status < 300) success(xhr.response);
            else error('请求出了点问题');
        }
    }
    //整理参数
    let str = ''
    for (let key in data){
        str += `${key}=${data[key]}&`
    }
    str = str.slice(0,-1)
    xhr.open('GET',url+'?'+str)
    xhr.responseType = 'json'
    xhr.send()
}
sendAjax(
    'https://api.apiopen.top/getJoke',
    {page:1,count:2,type:'video'},
    response =>{console.log('成功了',response);},
    err =>{console.log('失败了',err);}
)
```

API

- Promise构造函数: new Promise (executor) {}
    - executor函数: 是同步执行的，(resolve, reject) => {}
    - resolve函数: 调用resolve将Promise实例内部状态改为成功(fulfilled)。
    - reject函数: 调用reject将Promise实例内部状态改为失败(rejected)。

    - 说明: excutor函数会在Promise内部立即同步调用，异步代码放在excutor函数中。
- Promise.prototype.then方法: Promise实例.then(onFulfilled,onRejected)
    - onFulfilled: 成功的回调函数 (value) => {}
    - onRejected: 失败的回调函数 (reason) => {}
    - 特别注意(难点)：then方法会返回一个新的Promise实例对象
- Promise.prototype.catch方法: Promise实例.catch(onRejected)
    - onRejected: 失败的回调函数 (reason) => {}
    - 说明: catch方法是then方法的语法糖，相当于: then(undefined（占位）, onRejected)

```
const p = new Promise((resolve,reject)=>{
        setTimeout(()=>{
            reject(100)
        },1000)
    })
p.catch(
    reason => {console.log('失败了2',reason);}
)
//相当于
p.then(
    undefined,
    reason => {console.log('失败了2',reason);}
)
```

- Promise.resolve方法: Promise.resolve(value)
    - 说明：用于快速返回一个状态为fulfilled或rejected的Promise实例对象
    - 备注：value的值可能是：(1)非Promise值 (2)Promise值

```
const p0 = Promise.reject(-100)
const p = Promise.resolve(p0)
p.then(
        value => {console.log('成功了',value);},
        reason => {console.log('失败了',reason);}
)
//------失败       只要调Promise.resolve不一定成功
```

- Promise.reject方法: Promise.reject方法(reason)
    - 说明：用于快速返回一个状态必为rejected的Promise实例对象

```
const p0 = Promise.resolve(100)
const p = Promise.reject(p0)
p.then(
        value => {console.log('成功了',value);},
        reason => {console.log('失败了',reason);}
)
//------失败     只要调Promise.reject一定失败
```

- Promise.all方法: Promise.all(promiseArr)
    - promiseArr: 包含n个Promise实例的数组

- 说明: 返回一个新的Promise实例, 只有所有的promise都成功才成功，打印全部成功的，只要有一个失败了就直接失败，只打印最快出失败结果的。

```
const p1 = Promise.resolve('a')
const p2 = new Promise((resolve,reject)=>{
        setTimeout(()=>{
                reject('b')
        },500)
})
const p3 = new Promise((resolve,reject)=>{
        setTimeout(()=>{
                reject('c')
        },2000)
})
const x = Promise.all([p1,p2,p3])
x.then(
        value => {console.log('成功了',value);},
        reason => {console.log('失败了',reason);}
)
```

- Promise.race方法: Promise.race(promiseArr)
  - promiseArr: 包含n个Promise实例的数组
  - 说明: 返回一个新的Promise实例, 成功还是很失败？以最先出结果的promise为准。

```
const p1 = Promise.reject('a')
const p2 = new Promise((resolve,reject)=>{
        setTimeout(()=>{
                resolve('b')
        },500)
})
const p3 = new Promise((resolve,reject)=>{
        setTimeout(()=>{
                reject('c')
        },2000)
})
const x = Promise.race([p3,p1,p2])
x.then(
        value => {console.log('成功了',value);},
        reason => {console.log('失败了',reason);}
)
```

改变状态

- 改变一个Promise实例的状态
  - 执行resolve(value): 如果当前是pending就会变为fulfilled
  - 执行reject(reason): 如果当前是pending就会变为rejected
  - 执行器函数(executor)抛出异常: 如果当前是pending就会变为rejected

```
const p  = new Promise((resolve,reject)=>{
    console.log(a); //引擎抛异常
    // throw 900 //编码抛异常    抛出异常后 就不继续往下执行了
})
p.then(
    value => {console.log('成功了',value);},
    reason => {console.log('失败了',reason);}
)
```

- 改变改变Promise实例的状态和指定回调函数谁先谁后?
  - 都有可能, 正常情况下是先指定回调再改变状态, 但也可以先改状态再指定回调
  - 如何先改状态再指定回调?
    - 延迟一会再调用then()
  - Promise实例什么时候才能得到数据?
    - 如果先指定的回调, 那当状态发生改变时, 回调函数就会调用, 得到数据

```
//先指定回调，后改变状态（最常见）
const p = new Promise((resolve,reject)=>{
    setTimeout(()=>{
        resolve('a')
    },4000)
})
p.then(
    //以下两个回调函数，在实例p自身中（类似于数组）缓存
    //调用时才推向微队列（需要执行的时候才会进队列）
    value => {console.log('成功了',value);},
    reason => {console.log('失败了',reason);
                )
```

    - 如果先改变的状态, 那当指定回调时, 回调函数就会调用, 得到数据

```
//先改状态，后指定回调
const p = new Promise((resolve,reject)=>{
        resolve(100)
})
setTimeout(()=>{
        p.then(
                //以下两个回调函数，直接推向微队列
                value => {console.log('成功了',value);},
                reason => {console.log('失败了',reason);}
        )
},2000)
```

新的Promise实例

- Promise实例.then()返回的是一个【新的Promise实例】，它的值和状态由什么决定?
  - 简单表达: 由then()所指定的回调函数执行的结果决定
  - 详细表达:

        - 如果then所指定的回调返回的是非Promise值a：
            - 那么【新Promise实例】状态为：成功(fulfilled), 成功的value为a
        - 如果then所指定的回调返回的是一个Promise实例p：
            - 那么【新Promise实例】的状态、值，都与p一致
        - 如果then所指定的回调抛出异常：
            - 那么【新Promise实例】状态为rejected, reason为抛出的那个异常

```
const p = new Promise((resolve,reject)=>{
        setTimeout(()=>{
                resolve('a')
        },1000)
})

p.then(
        value => {console.log('成功了1',value); return Promise.reject('a')},
        reason => {console.log('失败了1',reason);}
).then(
        value => {console.log('成功了2',value);return true},
        reason => {console.log('失败了2',reason); return 100}
).then(
        value => {console.log('成功了3',value);throw 900},
        reason => {console.log('失败了3',reason); return false}
).then(
        value => {console.log('成功了4',value);return -100},
        reason => {console.log('失败了4',reason);}
)
```

回调地狱

promise优势

- 指定回调函数的方式更加灵活：
    - 旧的：必须在启动异步任务前指定
    - promise：启动异步任务 => 返回promie对象 => 给promise对象绑定回调函数(甚至可以在异步任务结束后指定)
- 支持链式调用, 可以解决回调地狱问题
    - 什么是回调地狱：回调函数嵌套调用, 外部回调函数异步执行的结果是嵌套的回调函数执行的条件
    - 回调地狱的弊病：代码不便于阅读、不便于异常的处理
    - 一个不是很优秀的解决方案：then的链式调用
    - 终极解决方案：async/await（底层实际上依然使用then的链式调用）

```
/*
        定义一个sendAjax函数，对xhr的get请求进行封装：
                        1.该函数接收4个参数：url(请求地址)、data(参数对象)、success(成功的回调)、
error(失败的回调)
*/
function sendAjax(url,data,success,error){
        //实例xhr
```

```
        const xhr = new XMLHttpRequest()
        //绑定监听
        xhr.onreadystatechange = ()=>{
                if(xhr.readyState === 4){
                        if(xhr.status >= 200 && xhr.status < 300) success(xhr.response);
                        else error('请求出了点问题');
                }
        }
        //整理参数
        let str = ''
        for (let key in data){
                str += `${key}=${data[key]}&`
        }
        str = str.slice(0,-1)
        xhr.open('GET',url+'?'+str)
        xhr.responseType = 'json'
        xhr.send()
}
//纯回调引起的回调地狱
sendAjax(
        'https://api.apiopen.top/getJoke',
        {page:1,count:2,type:'video'},
        response =>{
                console.log('第1次成功了',response);
                sendAjax(
                        'https://api.apiopen.top/getJoke',
                        {page:1,count:2,type:'video'},
                        response =>{
                                console.log('第2次成功了',response);
                                sendAjax(
                                        'https://api.apiopen.top/getJoke',
                                        {page:1,count:2,type:'video'},
                                        response =>{
                                                console.log('第3次成功了',response);
                                        },
                                        err =>{console.log('第3次失败了',err);}
                                )
                        },
                        err =>{console.log('第2次失败了',err);}
                )
        },
        err =>{console.log('第1次失败了',err);}
)
```

then的链式调用（解决回调地狱）

```
function sendAjax(url,data){
        return new Promise((resolve,reject)=>{
                //实例xhr
                const xhr = new XMLHttpRequest()
                //绑定监听
                xhr.onreadystatechange = ()=>{
```

```
                        if(xhr.readyState === 4){
                                if(xhr.status >= 200 && xhr.status < 300)
resolve(xhr.response);
                                else reject('请求出了点问题');
                        }
                }
                //整理参数
                let str = ''
                for (let key in data){
                        str += `${key}=${data[key]}&`
                }
                str = str.slice(0,-1)
                xhr.open('GET',url+'?'+str)
                xhr.responseType = 'json'
                xhr.send()
        })
}
//发送第1次请求
sendAjax('https://api.apiopen.top/getJoke',{page:1})
.then(
        value => {
                console.log('第1次请求成功了',value);
                //发送第2次请求
                return sendAjax('https://api.apiopen.top/getJoke',{page:1})
        },
        reason => {console.log('第1次请求失败了',reason);}
)
.then(
        value => {
                console.log('第2次请求成功了',value);
                //发送第3次请求
                return sendAjax('https://api.apiopen.top/getJoke',{page:1})
        },
        reason => {console.log('第2次请求失败了',reason);}
)
.then(
        value => {console.log('第3次请求成功了',value);},
        reason => {console.log('第3次请求失败了',reason);}
)
```

中断promise链

- 当使用promise的then链式调用时, 在中间中断, 不再调用后面的回调函数。
- 办法: 在失败的回调函数中返回一个pendding状态的Promise实例。

```
function sendAjax(url,data){
        return new Promise((resolve,reject)=>{
                //实例xhr
                const xhr = new XMLHttpRequest()
                //绑定监听
                xhr.onreadystatechange = ()=>{
                        if(xhr.readyState === 4){
                                if(xhr.status >= 200 && xhr.status < 300)
```

```
resolve(xhr.response);
                                else reject('请求出了点问题');
                        }
                }
                //整理参数
                let str = ''
                for (let key in data){
                        str += `${key}=${data[key]}&`
                }
                str = str.slice(0,-1)
                xhr.open('GET',url+'?'+str)
                xhr.responseType = 'json'
                xhr.send()
        })
}
//发送第1次请求
sendAjax('https://api.apiopen.top/getJoke',{page:1})
.then(
        value => {
                console.log('第1次请求成功了',value);
                //发送第2次请求
                return sendAjax('https://api.apiopen.top/getJoke2',{page:1})
        },
        reason => {console.log('第1次请求失败了',reason);return new Promise(()=>{})}
)
.then(
        value => {
                console.log('第2次请求成功了',value);
                //发送第3次请求
                return sendAjax('https://api.apiopen.top/getJoke',{page:1})
        },
        reason => {console.log('第2次请求失败了',reason);return new Promise(()=>{})}
)
.then(
        value => {console.log('第3次请求成功了',value);},
        reason => {console.log('第3次请求失败了',reason);}
)
```

promise错误穿透

- 当使用promise的then链式调用时, 可以在最后用catch指定一个失败的回调
- 前面任何操作出了错误, 都会传到最后失败的回调中处理了
- 如果不存在then的链式调用，就不需要考虑then的错误穿透。

```
function sendAjax(url,data){
        return new Promise((resolve,reject)=>{
                //实例xhr
                const xhr = new XMLHttpRequest()
                //绑定监听
                xhr.onreadystatechange = ()=>{
                        if(xhr.readyState === 4){
                                if(xhr.status >= 200 && xhr.status < 300)
resolve(xhr.response);
```

```
                                else reject(`请求出了点问题`);
                        }
                }
                //整理参数
                let str = ''
                for (let key in data){
                        str += `${key}=${data[key]}&`
                }
                str = str.slice(0,-1)
                xhr.open('GET',url+'?'+str)
                xhr.responseType = 'json'
                xhr.send()
        })
}
//利用错误的穿透避免多次指定失败的回调
sendAjax('https://api.apiopen.top/getJoke2',{page:1})
.then(
        value => {
                console.log('第1次请求成功了',value);
                //发送第2次请求
                return sendAjax('https://api.apiopen.top/getJoke',{page:1})
        },
        // reason => {console.log('第1次请求失败了',reason);return new Promise(()=>{})}
)
.then(
        value => {
                console.log('第2次请求成功了',value);
                //发送第3次请求
                return sendAjax('https://api.apiopen.top/getJoke',{page:1},3)
        },
        // reason => {console.log('第2次请求失败了',reason);return new Promise(()=>{})}
)
.then(
        value => {console.log('第3次请求成功了',value);},
        // reason => {console.log('第3次请求失败了',reason);return new Promise(()=>{})}
)
.catch(
        reason => {console.log(reason);}
)
```

```
//简单例子演示错误的穿透
const p = new Promise((resolve,reject)=>{
        setTimeout(()=>{
                reject(-100)
        },1000)
})
p.then(
        value => {console.log('成功了1',value);return 'b'},
        //reason => {throw reason}    //底层帮我们补上的这个失败的回调
)
.then(
        value => {console.log('成功了2',value);return Promise.reject(-108)},
        //reason => {throw reason}    //底层帮我们补上的这个失败的回调
```

```
)
.catch(
        reason => {throw reason}
)
```

async与await（解决回调地狱）

- async修饰的函数
  - 函数的返回值为promise对象
  - Promise实例的结果由async函数执行的返回值决定
- await表达式
  - await右侧的表达式一般为Promise实例对象, 但也可以是其它的值
    - 如果表达式是Promise实例对象, await后的返回值是promise成功的值
    - 如果表达式是其它值, 直接将此值作为await的返回值
- 注意:
  - await必须写在async函数中, 但async函数中可以没有await
  - 如果await的Promise实例对象失败了, 就会抛出异常, 需要通过try...catch来捕获处理

```
//测试async
async   function demo(){
        const result = await p1
        console.log(result);
}
demo()
```

```
const p1 = new Promise((resolve,reject)=>{
        setTimeout(()=>{
                resolve('a')
        },1000)
})
const p2 = new Promise((resolve,reject)=>{
        setTimeout(()=>{
                resolve('一些错误')
        },2000)
})
const p3 = new Promise((resolve,reject)=>{
        setTimeout(()=>{
                resolve('c')
        },4000)
})

;(async()=>{
        try {
                const result1 = await p1
                console.log(result1);
                const result2 = await p2
                console.log(result2);
                const result3 = await p3
                console.log(result3);
```

```
        } catch (error) {
                console.log(error);
        }
})()
```

```
function sendAjax(url,data){
        return new Promise((resolve,reject)=>{
                //实例xhr
                const xhr = new XMLHttpRequest()
                //绑定监听
                xhr.onreadystatechange = ()=>{
                        if(xhr.readyState === 4){
                                if(xhr.status >= 200 && xhr.status < 300)
resolve(xhr.response);
                                else reject(`请求出了点问题`);
                        }
                }
                //整理参数
                let str = ''
                for (let key in data){
                        str += `${key}=${data[key]}&`
                }
                str = str.slice(0,-1)
                xhr.open('GET',url+'?'+str)
                xhr.responseType = 'json'
                xhr.send()
        })
}
(async()=>{
                try {
                        const result1 = await sendAjax('https://api.apiopen.top/getJoke',
{page:1})
                        console.log('第1次请求成功了',result1);
                        const result2 = await sendAjax('https://api.apiopen.top/getJoke',
{page:1})
                        console.log('第2次请求成功了',result2);
                        const result3 = await sendAjax('https://api.apiopen.top/getJoke',
{page:1})
                        console.log('第3次请求成功了',result3);
                } catch (error) {
                        console.log(error);
                }
})()
```

原理

- 使用async配合await这种写法：
    - 表面上不出现任何的回调函数
    - 但实际上底层把我们写的代码进行了加工，把回调函数“还原”回来了。
    - 最终运行的代码是依然有回调的，只是程序员没有看见。

```
const p = new Promise((resolve,reject)=>{
        setTimeout(()=>{
                resolve('a')
        },4000)
})
async   function demo(){
        //程序员“轻松”的写法
        const result = await p
        console.log(result);
        console.log(100);
        console.log(200);

        //浏览器翻译后的代码
        p.then(
                result => {
                        console.log(result);
                        console.log(100);
                        console.log(200);
                },
        )
}
demo()
console.log(1);
```

宏队列与微队列

- 宏队列:[宏任务1，宏任务2.....]
- 微队列:[微任务1，微任务2.....] Promise的异步回调是微任务
- 规则：每次要执行宏队列里的一个任务之前，先看微队列里是否有待执行的微任务
    - 如果有，先执行微任务
    - 如果没有，按照宏队列里任务的顺序，依次执行

面试题

```
setTimeout(()=>{
  console.log('timeout')
},0)

Promise.resolve(1).then(
  value => console.log('成功1',value)
)
Promise.resolve(2).then(
  value => console.log('成功2',value)
)
console.log('主线程')
//------主线程/成功1 1/成功2 2/timeout
```

```
setTimeout(()=>{
  console.log('timeout1')
```

```
})
setTimeout(()=>{
  console.log('timeout2')
})

Promise.resolve(1).then(
  value => console.log('成功1',value)
)
Promise.resolve(2).then(
  value => console.log('失败2',value)
)
//------成功1 1/失败2 2/timeout1/timeout2
```

```
setTimeout(()=>{
  console.log('timeout1')
  Promise.resolve(5).then(
    value => console.log('成功了5')
  )
})
setTimeout(()=>{
  console.log('timeout2')
})

Promise.resolve(3).then(
  value => console.log('成功了3')
)
Promise.resolve(4).then(
  value => console.log('失败了4')
)
//------成功了3/失败了4/timeout1/成功了5/timeout2
```