Promise深入 + 自定义Promise

1. 准备

1.1. 函数对象与实例对象

```
1. 函数对象: 将函数作为对象使用时, 简称为函数对象
2. 实例对象: new 函数产生的对象, 简称为对象
```

1.2. 回调函数的分类

```
1. 同步回调:
    理解: 立即执行, 完全执行完了才结束, 不会放入回调队列中
    例子: 数组遍历相关的回调函数 / Promise的excutor函数
2. 异步回调:
    理解: 不会立即执行, 会放入回调队列中将来执行
    例子: 定时器回调 / ajax回调 / Promise的成功|失败的回调
```

异步回调举例:

```
fs文件操作:
require('fs').readFile('/index.html',(err,data)+>{})

AJAX操作:
router.get('/info',(request,response)=>{})

定时器:
setTimeout(function(){},1000)
```

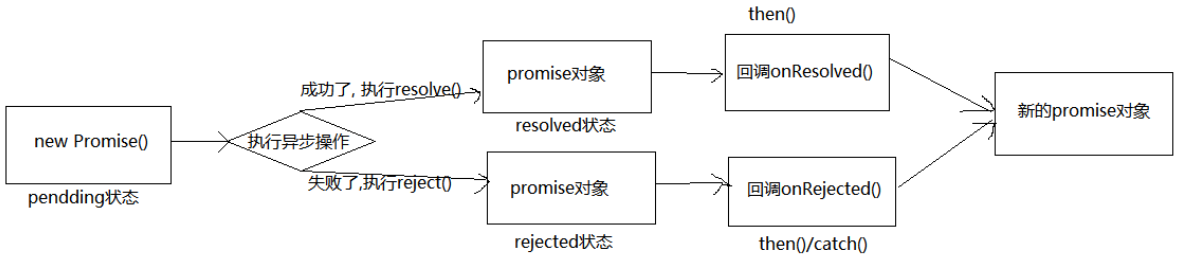
1.3. JS中的Error

```
1. 错误的类型
    Error: 所有错误的父类型
    ReferenceError: 引用的变量不存在
    TypeError: 数据类型不正确的错误
    RangeError: 数据值不在其所允许的范围内
    SyntaxError: 语法错误
2. 错误处理
    捕获错误: try ... catch
    抛出错误: throw error
3. 错误对象
    message属性: 错误相关信息
    stack属性: 函数调用栈记录信息
```

2. Promise的理解和使用

2.1. Promise是什么?

```
1.抽象表达:
    Promise是JS中进行异步编程的新的解决方案(旧的是谁?)
2.具体表达:
    从语法上来说: Promise是一个构造函数
    从功能上来说: promise对象用来封装一个异步操作并可以获取其结果
3. promise的状态改变(只有2种, 只能改变一次)
    pending变为resolved
    pending变为rejected
4. promise的基本流程
```



2.2 promise的模块化练习

1 定时器

```
// resolve和reject都是函数类型的参数
// 当异步任务成功时，调用resolve(),失败时自动调用reject()
const  p = new Promise((resolve,reject)=>{
    setTimeout(()=>{
        let n = rand(1,100);
        if(n<=30){
            // 将promise的状态改为成功，并且将n传递给成功的回调
            resolve(n);
        }else{
            // 将promise的状态改为失败，并且将n传递给失败的回调
            reject(n);
        }
    },1000)
})
// then方法传递两个函数参数，分别是成功和失败的回调
p.then((value)=>{
    alert('恭喜，中奖了,中奖号码为：'+value)
},(reason)=>{
    alert('谢谢惠顾，号码为：'+reason)
})
```

2 fs模块

```
const fs = require('fs');

// 不适用promise的形式
```

```
// fs.readFile('bkpp.txt',(err,data)=>{
//     if(err){
//         throw err;
//     }
//     console.log(data);
// })

// Promise形式

let p = new Promise((resolve,reject)=>{
    fs.readFile('bkp.txt',(err,data)=>{
        if(err){
                    reject(err);
        }
        resolve(data);
    })
})
p.then((data)=>{
    console.log(data.toString());
},(reason)=>{
    console.log(reason)
})
```

3 ajax请求

```
// 不用promise
const xhr = new XMLHttpRequest();
xhr.open('get','https://api.apiopen.top/getJok');
xhr.send();
xhr.onreadystatechange = function(){
    if(xhr.readyState == 4){
        if(xhr.status>=200 && xhr.status<=300){
            // 控制台输出响应状态码
            console.log(xhr.response)
        }else{
            // 控制天输出无响应状态码
            console.log(xhr.status)
        }
    }
}

// 使用promise
let p = new Promise((resolve,reject)=>{
    const xhr = new XMLHttpRequest();
    xhr.open('get','https://api.apiopen.top/getJoke');
    xhr.send();
    xhr.onreadystatechange = function(){
        if(xhr.readyState == 4){
            if(xhr.status>=200 && xhr.status<=300){
                // 控制台输出响应状态码
                resolve(xhr.response);
            }else{
                // 控制台输出无响应状态码
                reject(xhr.status)
            }
        }
    }
```

```
})
p.then((data)=>{
    console.log(data)
},(reason)=>{
    console.log(reason)
})
```

2.3 promise的封装练习

1 fs模块封装

```
function mineReadFile(path){
    return new Promise((resolve,reject)=>{
        // 读取文件
        require('fs').readFile(path,(err,data)=>{
            if(err){
                reject(err);
            }
            resolve(data);
        })
    })
}

mineReadFile('./bkpp.txt')   // 返回一个promise对象
.then((data)=>{
    console.log(data.toString());
},(reason)=>{
    console.log(reason)
})
```

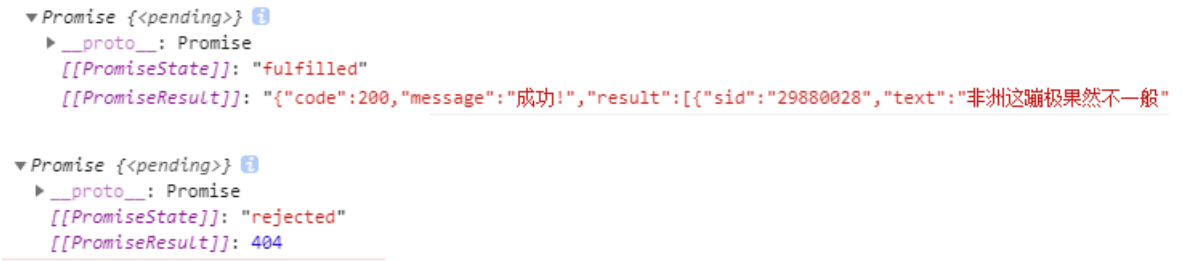
2 ajax封装

```
function sendAJAX(url){
    return new Promise((resolve,reject)=>{
        const xhr = new XMLHttpRequest();
        xhr.open('get',url);
        xhr.send();
        xhr.onreadystatechange = function(){
            if(xhr.readyState == 4){
                if(xhr.status>=200 && xhr.status<=300){
                    // 控制台输出响应状态码
                    resolve(xhr.response);
                }else{
                    // 控制台输出无响应状态码
                    reject(xhr.status)
                }
            }
        }
    })
}
// 调用
sendAJAX('https://api.apiopen.top/getJoke').then((data)=>{
    console.log(data)
},(reason)=>{
    console.log(reason)
})
```

2.4 Promise实例对象的两个属性

```
PromiseState：表示异步任务当前的状态
    取值：
        pending  未决定的，初始状态
        resolved(fullfilled)  成功
        rejected  失败，发生错误
    状态只能从pending到其他，并且只能改变一次

PromiseResult：存放异步任务成功/或者失败的结果
    resolve和reject函数可以修改这个值
    其他方法均不能修改
```



2.2. 为什么要用Promise?

```
1. 指定回调函数的方式更加灵活：可以在请求发出甚至结束后指定回调函数
    Promise.then(resolve);
2. 支持链式调用，可以解决回调地狱问题
    回调地狱问题：回调中嵌套包含另一个异步任务，该异步任务中又包含异步回调
```

2.3. 如何使用Promise?

2.3.1 主要API

```
Promise构造函数: Promise (excutor) {}
    (1) excutor 函数: 执行器 (resolve, reject) => {}
        在Promise内部同步调用，异步操作在执行器中执行
    (2) resolve 函数: 内部定义成功时我们调用的函数 value => {}
    (3) reject 函数: 内部定义失败时我们调用的函数 reason => {}

Promise.prototype.then方法: (onResolved, onRejected) => {}
    (1) onResolved 函数: 成功的回调函数 (value) => {}
    (2) onRejected 函数: 失败的回调函数 (reason) => {}
    说明: 指定用于得到成功 value 的成功回调和用于得到失败 reason 的失败回调
    返回一个新的 promise 对象

Promise.prototype.catch 方法: (onRejected) => {}
    (1) onRejected 函数: 失败的回调函数 (reason) => {}
    说明: then()的语法糖, 相当于: then(undefined, onRejected)
    只能指定失败的回调
    返回一个新的 promise 对象

Promise.resolve 方法: (value) => {}
    (1) value: 成功的数据或 promise 对象
    说明: 返回一个成功/失败的 promise 对象
    无论传递什么值，都会返回一个Promise对象，只是对象的状态有区别：
```

如果传入的值不是Promise实例对象，则返回一个状态为成功的Promise对象，结果是传入的参数
如果传入的值是Promise实例对象1，则该对象1的结果就决定了新的Promise对象的状态
成功，则新的Promise对象的状态也会成功，结果也是对象1的结果
失败，则新的Promise对象的状态也会失败，结果就是对象1的结果

```
var p = Promise.resolve('512');
console.log(p)
p.then((data)=>{
    console.log(data)
},(reason)=>{
    console.log(reason)
})
```

传入非Promise实例

```
▼Promise {<fulfilled>: "512"}
  ▶__proto__: Promise
   [[PromiseState]]: "fulfilled"
   [[PromiseResult]]: "512"
512
```

```
var p = Promise.resolve(new Promise((resolve,reject)=>{
    resolve('ok')
}));
console.log(p)
```

传入一个成功的Promise实例

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

```
var p = Promise.resolve(new Promise((resolve,reject)=>{
    reject('Error')
}));
console.log(p)
p.catch(err=>{
    console.log(err)
})
```

传入一个失败的Promise实例

```
▼Promise {<rejected>: "Error"}
  ▶__proto__: Promise
   [[PromiseState]]: "rejected"
   [[PromiseResult]]: "Error"
Error
```

Promise.reject 方法: (reason) => {}
(1) reason: 失败的原因
说明: 返回一个失败的 promise 对象,结果是传入的参数
参数:
非Promise对象：返回一个失败的 promise 对象,结果是传入的非Promise对象
成功的Promise对象：返回一个失败的 promise 对象,结果是传入的成功的Promise对象
失败的Promise对象：返回一个失败的 promise 对象,结果是传入的失败的Promise对象

```
var p = Promise.reject(521);
console.log(p)
p.catch(err=>{
    console.log(err)
})
```

传入非Promise实例

```
▼Promise {<rejected>: 521}
  ▶__proto__: Promise
   [[PromiseState]]: "rejected"
   [[PromiseResult]]: 521
521
```

```
var p = Promise.reject(new Promise((resolve,reject)=>{
    resolve(123)
}).then((value)=>{console.log(value)})
);
console.log('p:',p)
p.catch(err=>{
    console.log('p的结果:',err)
})
```

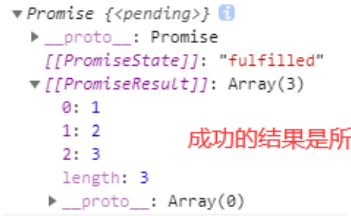
传入一个成功的Promise实例

```
p: ▼Promise {<rejected>: Promise}
    ▶__proto__: Promise
     [[PromiseState]]: "rejected"
    ▶[[PromiseResult]]: Promise
123
p的结果: ▼Promise {<fulfilled>: undefined}
    ▶__proto__: Promise
     [[PromiseState]]: "fulfilled"
     [[PromiseResult]]: undefined
```

所传入的实例

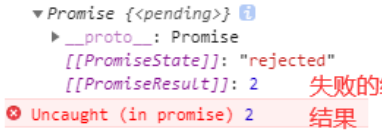
Promise.all 方法: (promises) => {}
(1) promises: 包含 n 个 promise 的数组
说明: 返回一个新的 promise, 只有所有的 promise 都成功才成功, 只要有一个失败了就直接失败
成功的结果是所有成功的promise实例的结果组成的一个数组
失败的结果是这个数组中失败的promise对象的结果

```
var p = Promise.all([Promise.resolve(1),Promise.resolve(2),Promise.resolve(3)]);
console.log(p)
```



成功的结果是所有promise对象的结果组成的数组

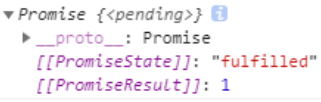
```
var p = Promise.all([Promise.resolve(1),Promise.reject(2),Promise.resolve(3),Promise.reject(4)]);
console.log(p)
```



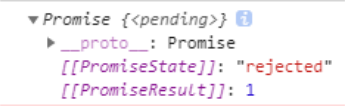
失败的结果是第一个失败的promise对象的结果

```
Promise.race 方法: (promises) => {}
    (1) promises: 包含 n 个 promise 的数组
    说明: 返回一个新的 promise, 第一个完成的 promise 的结果状态就是最终的结果状态
    不一定是按照参数顺序来的
```

```
var p = Promise.race([Promise.resolve(1),Promise.reject(2),Promise.resolve(3),Promise.reject(4)]);
console.log(p)
```




```
var p = Promise.race([Promise.reject(1),Promise.reject(2),Promise.resolve(3),Promise.reject(4)]);
console.log(p)
```



2.3.2 几个重要问题

> - 1 如何改变promise的状态?

```
    1.调用resolve函数，pending--->resolved
    2.调用reject函数，pending--->rejected
    3.抛出错误：throw 'Error'    pending--->rejected
```

```
var p = new Promise((resolve,reject)=>{
    resolve(123);
})
console.log(p)
```

```
▼Promise {<fulfilled>: 123}
  ▶__proto__: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: 123
```

```
var p = new Promise((resolve,reject)=>{
    reject(123);
})
console.log(p)
```

```
▼Promise {<rejected>: 123}
  ▶__proto__: Promise
    [[PromiseState]]: "rejected"
    [[PromiseResult]]: 123
▶Uncaught (in promise) 123
```

```
var p = new Promise((resolve,reject)=>{
    throw 'error'
})
console.log(p)
```

```
▼Promise {<rejected>: "error"}
  ▶__proto__: Promise
    [[PromiseState]]: "rejected"
    [[PromiseResult]]: "error"
▶Uncaught (in promise) error
```

- 2 一个promise指定多个成功/失败回调函数, 都会调用吗?

当 promise 改变为对应状态时都会调用

```
var p = new Promise((resolve,reject)=>{
    resolve('ok');
})
p.then((value)=>{
    console.log(1,value)
})
p.then((value)=>{
    console.log(2,value)
})
```

```
1 "ok"
2 "ok"
```

```
var p = new Promise((resolve,reject)=>{
    reject('error');
})
p.then(()=>{},(value)=>{
    console.log(1,value)
})
p.then(()=>{},(value)=>{
    console.log(2,value)
})
```

```
1 "error"
2 "error"
```

- 3 改变promise状态和指定回调函数谁先谁后?

(1) 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
(2) 如何先改状态再指定回调?
    1 在执行器中直接调用 resolve()/reject()
    2 延迟更长时间才调用 then()

(3) 什么时候才能得到数据?
    1 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
    2 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

先指定回调，再改变状态：

```
// setTimeout是宏任务，交给定时器模块处理，然后把回调交给任务队列
// 所以执行then，then是微任务，放入微任务队列，等待数据
// 等到状态改变，然后再执行微任务
var p = new Promise((resolve,reject)=>{
    setTimeout(function(){
        resolve('ok');
    },5000)
})
p.then((value)=>{
    console.log(1,value)
})
```

- 4 promise.then()返回的新promise的结果状态由什么决定?

(1) 简单表达：由 then()指定的回调函数执行的结果决定
(2) 详细表达：
    1 如果抛出异常，新 promise 变为 rejected，结果为抛出的异常
    2 如果返回的是非 promise 的任意值，新 promise 变为 resolved，结果为返回的值
    3 如果返回的是另一个新 promise，此 promise 的结果就会成为新 promise 的结果

- 5 promise如何串连多个操作任务?

(1) promise 的 then()返回一个新的 promise，可以开成 then()的链式调用
(2) 通过 then 的链式调用串连多个同步/异步任务

```
var p = new Promise((resolve,reject)=>{
    resolve(123)
})
var p2 = p.then((value)=>{
    return value+1;
}).then((value)=>{
    return value+1;
})
console.log(p2)
```

```
▼Promise {<pending>}
  ▶__proto__: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: 125
```

```
var p = new Promise((resolve,reject)=>{
    resolve(123)
})
var p2 = p.then((value)=>{
    console.log(value)
    // 这个函数中没有return 则then返回的promise对象中的结果就是undefined
}).then((value)=>{
    // 从而这里接收到的value就是undefined,value+1==>NaN
    return value+1;
    // 这个then有return语句，return的值是一个非promise对象，
    // 则它会返回一个成功的promise对象，结果是NaN
})
console.log(p2)
```

```
▼Promise {<pending>}
  ▶__proto__: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: NaN
123
```

- 6 promise异常传(穿)透?

(1) 当使用 promise 的 then 链式调用时，可以在最后指定失败的回调，
(2) 前面任何操作出了异常，都会传到最后失败的回调中处理

```
var p = new Promise((resolve,reject)=>{
    resolve(123)
})
var p2 = p.then((value)=>{
    return value+1;  fullfilled 124
}).then((value)=>{
    throw 'error'   rejected 'error'
}).then((value)=>{
    console.log(123)
    return value+'123';
},(err)=>{
    console.log(123,err);
    return err;  fullfilled 'error'
}).catch(err=>{
    console.log(err)   错误已经收集到，则catch不再执行
})
console.log(p2)
```

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

- 7 中断promise链

(1) 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
(2) 办法：在回调函数中返回一个 pendding 状态的 promise 对象
return new Promise(()=>{})
因为then方法的回调需要状态改变为resolved或者rejected后才可以执行，返回一个pending状态的promise,会导致then方法也是一个pending状态的promise对象，从而之后的then方法就不能执行

3. 自定义Promise

3.1 定义整体结构

3.2 Promise构造函数的实现

3.3 promise.then()/catch()的实现

3.4 Promise.resolve()/reject()的实现

3.5 Promise.all/race()的实现

3.6 Promise.resolveDelay()/rejectDelay()的实现

3.7 ES6 class版本

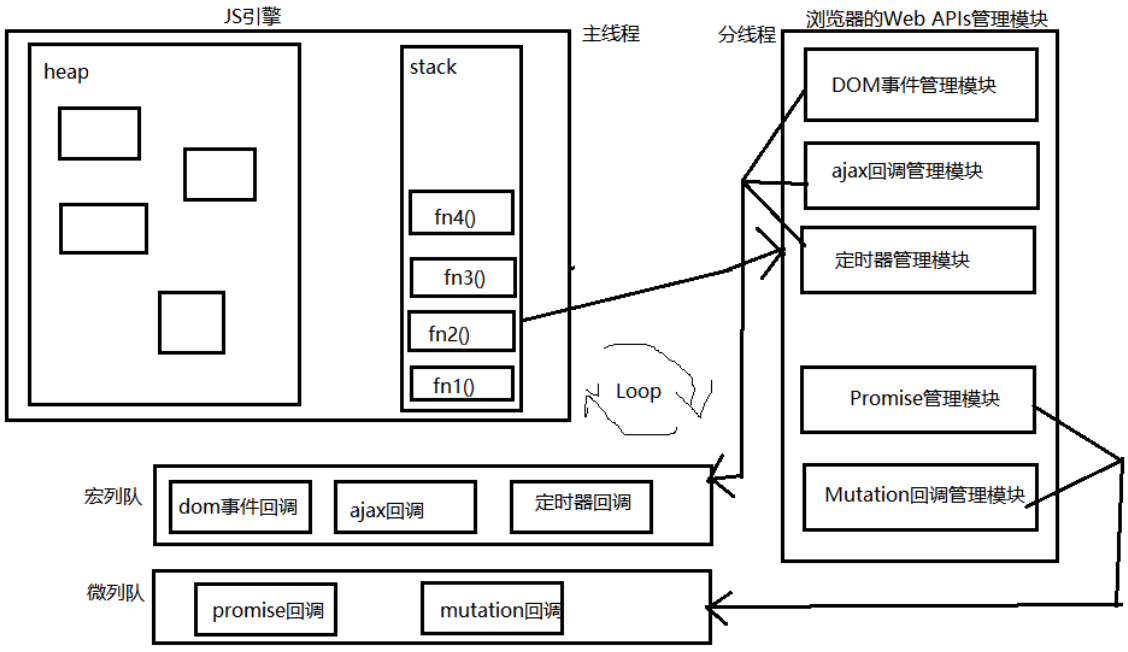
4. async与await

```
1. async 函数
    函数的返回值为promise对象
    promise对象的结果由async函数执行的返回值决定

2. await 表达式
    await右侧的表达式一般为promise对象, 但也可以是其它的值
    如果表达式是promise对象, await返回的是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

3. 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失败了, 就会抛出异常, 需要通过try...catch来捕获处理
```

5. JS异步之宏队列与微队列



1. 宏列队: 用来保存待执行的宏任务(回调), 比如: 定时器回调/DOM事件回调/ajax回调
2. 微列队: 用来保存待执行的微任务(回调), 比如: promise的回调/MutationObserver的回调
3. JS执行时会区别这2个队列
   JS引擎首先必须先执行所有的初始化同步任务代码
   每次准备取出第一个宏任务执行前, 都要将所有的微任务一个一个取出来执行