Promise

1.解决异步回调问题

1.1 如何同步异步请求

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

```
const fs = require('fs');
let school = {};
fs.readFile('./name.txt','utf8',function (err,data) {
    school.name = data;
});
fs.readFile('./age.txt','utf8',function (err,data) {
    school.age = data;
});
console.log(school);
```

1.2如何解决回调地狱

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

```
const fs = require('fs');
fs.readFile('./content.txt','utf8',function (err,data) {
    if(err)console.log(err);
    fs.readFile(data,'utf8',function (err,data) {
        if(err)console.log(err);
        console.log(data);
    })
});
```

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

例如媳妇说想买个包，这时候他就要"等待"我的回复，我可以过两天买，如果买了表示"成功"，如果我最后拒绝表示"失败"，当然我也有可能一直拖一辈子

- 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

4.1 promise的方法会立刻执行

```
let promise = new Promise(()=>{
    console.log('hello');
});
console.log('world');
```

4.2 promise也可以代表一个未来的值

```
const fs = require('fs');
let promise = new Promise((resolve,reject)=>{
    fs.readFile('./content.txt','utf8',function (err,data) {
        if(err)console.log(err);
        resolve(data);
    })
});
promise.then(data =>{
    console.log(data);
});
```

4.3 代表一个用于不会返回的值

```
const fs = require('fs');
let promise = new Promise((resolve,reject)=>{});
promise.then(data =>{
    console.log(data);
});
```

4.4 应用状态实现抛硬币

```
function flip_coin() {
    return new Promise((resolve,reject)=>{
        setTimeout(function () {
            var random =  Math.random();
            if(random > 0.5){
```

```
                resolve('正');
            }else{
                resolve('反');
            }
        },2000)
    })
}
flip_coin().then(data=>{
    console.log(data);
},data=>{
    console.log(data);
});
```

5.实现简单的Promise

```
function Promise(fn) {
    fn((data)=>{
        this.resolve(data)

    },(data)=>{
        this.reject(data);
    })
}
Promise.prototype.resolve = function (data) {
    this._success(data)
};
Promise.prototype.reject = function (data) {
    this._error(data);
};
Promise.prototype.then = function (success,error) {
```

```
    this._success = success;
    this._error = error;
};
```

6.Error会导致触发Reject

可以采用then的第二个参数捕获失败，也可以通过catch函数进行捕获

```
function flip_coin() {
    return new Promise((resolve,reject)=>{
        throw Error('没有硬币')
    })
}
flip_coin().then(data=>{
    console.log(data);
}).catch((e)=>{
    console.log(e);
})
```

7.Promise.all实现并行

接受一个数组，数组内都是Promise实例，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

```
const fs = require('fs');
let p1 =  new Promise((resolve,reject)=>{
    fs.readFile('./name.txt','utf8',function (err,data) {
```

```
        resolve(data);
    });
})
let p2 = new Promise((resolve,reject)=>{
    fs.redFile('./age.txt','utf8',function (err,data) {
        resolve(data);
    });
})
Promise.all([p1,p2]).then(([res1,res2])=>{
    console.log(res1);
})
```

> 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

8.Promise.race实现选择

接受一个数组，数组内都是Promise实例,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。

```
const fs = require('fs');
let p1 =  new Promise((resolve,reject)=>{
    fs.readFile('./name.txt','utf8',function (err,data) {
        resolve(data);
    });
})
let p2 = new Promise((resolve,reject)=>{
    fs.readFile('./age.txt','utf8',function (err,data) {
```

```
            resolve(data);
        });
    })
    Promise.race([p1,p2]).then(([res1,res2])=>{
        console.log(res1,res2);
    })
```

9.Promise.resolve

返回一个Promise实例，这个实例处于resolve状态。

```
Promise.resolve('成功').then(data=>{
    console.log(data);
})
```

10.Promise.reject

返回一个Promise实例，这个实例处于reject状态

```
Promise.reject('失败').then(data=>{
    console.log(data);
},re=>{
    console.log(re);
})
```

11.封装ajax

```
function ajax({url=new Error('url必须提供'),method='GET',async=true,dataType='json'}){
  return new Promise(function(resolve,reject){
     var xhr = new XMLHttpRequest();
     xhr.open(method,url,async);
     xhr.responseType = dataType;
     xhr.onreadystatechange = function(){
         if(xhr.readyState == 4){
             if(/^2\d{2}/.test(xhr.status)){
                resolve(xhr.response);
             }else{
                 reject(xhr.response);
             }
         }
     }
      xhr.send();
  });
}
```

12.chain中返回结果

```
Promise.resolve([1,2,3])
.then(arr=>{
    return [...arr,4]
}).then(arr=>{
    return [...arr,5]
}).then(arr=>{
    console.log(arr);
})
```

13.chain中返回promise

then中的结果是promise的resolve后的结果

```
Promise.resolve('user').then(data=>{
    return new Promise(function (resolve,reject) {
        fetch('/'+data).then(res=>res.json().then((json)=>{
            resolve(json)
        }))
    })
}).then(data=>{
    console.log(data);
});
```

改写的更简单些

```
Promise.resolve('user').then(data=>{
    return fetch('/'+data)
}).then(res=>{
    return res.json();
}).then(data=>{
    console.log(data);
})
```

14.async/await

本质是语法糖，await与async要连用，await后只能跟promise

```
async function getHello() {
    return new Promise((resolve,reject) => {
        setTimeout(function () {
            resolve('hello');
        },2000);
    })
}
async function getData () {
   var result = await getHello();
   console.log(result);
} ;
getData();
```