Promise的三种状态

- 挂起
- 已成功
- 已完成

其中后两种都是异步操作完成后的状态

Promise做保证

> Promise对象用于表示一个异步操作的最终状态（完成或失败），以及其返回的值。

MDN对Promise的定义如上，可以理解为此对象做了一些保证，告知我们异步操作的状态。具体如下：

- 当前事件队列执行完成之后，再调用回调函数
- 回调函数是通过then添加的
- 添加多个then，可以添加多个回调函数，依次执行

Promise链式调用

存在的需求：有时候我们需要连续调用多个异步操作，每一个操作都建立在得到上一部结果之后。以前有回调函数，这样会容易造成回调地狱。而采用Promise之后，每一步操作成功/失败之后都会带上其结果，执行下一个操作。

```
// 回调地狱
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
        console.log('Got the final result: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);

//采用Promise链式调用之后
doSomething().then(function(result) {
  return doSomethingElse(result);
})
.then(function(newResult) {
  return doThirdThing(newResult);
})
.then(function(finalResult) {
  console.log('Got the final result: ' + finalResult);
})
.catch(failureCallback);
```

错误处理

在上面的代码段1中，有三个错误回调函数。而在Promise链式调用中，只在结尾加上错误处理回调即可，当有错误抛出时，会返回一个带有错误原因的Promise到下一级catch，直到最底端catch函数。

```
// 依次输出： Initial Do that
new Promise((resolve, reject) => {
    console.log('Initial');
    resolve();
})
.then(() => {
    throw new Error('Something failed');
    console.log('Do this');
})
.then(() => {
    console.log('Do this whatever happened before');
})
.catch(() => {
    console.log('Do that');
})
```

此外，如果中途捕获了异常，依然可以接着then下去：

```
/*
 * 依次输出:
 * Initial
 * Do that
 * Do this whatever happened before
 */
new Promise((resolve, reject) => {
    console.log('Initial');
    resolve();
})
.then(() => {
    throw new Error('Something failed');
    console.log('Do this');
})
.catch(() => {
    console.log('Do that');
})
.then(() => {
    console.log('Do this whatever happened before');
})
```

原因在于catch(failureCallback)本身是then(null, failureCallback)的缩略形式，也是返回带有当前状态的Promise。下面这样咱们还能捕获到异常信息：

```
/*
```

```
 * 依次输出:
 * Initial
 * Something failed
 * Do that
 * Do this whatever happened before
 */
new Promise((resolve, reject) => {
    console.log('Initial');
    resolve();
})
.then(() => {
    throw new Error('Something failed');
    console.log('Do this');
})
.catch((e) => {
    console.log(e.message)
    console.log('Do that');
})
.then(() => {
    console.log('Do this whatever happened before');
})
```

使用async/await语法糖

一个栗子

```
// 使用Promise
doSth()
    .then(res => doSthElse(res))
    .then(newRes => doThirdTh(newRes))
    .then(finalRes => {
        console.log(`finalResult is ${finalRes}`)
    })
// 使用async/await将异步代码写成同步样式
async funtion foo () {
    let res = await doSth()
    let newRes = await doSthElse(res)
    let finalRes = await doThirdTh(newRes)
    console.log(`finalResult is ${finalRes}`)
}
```

Promise.resolve()、Promise.reject()妙用

使用这两种静态方法可以创建resolve或reject的保证，栗子如下：

```
getRecommend () {
    let today = new Date()
```

```
    let date = new Date(today.getFullYear(), today.getMonth() + 1,
today.getDate(), 9)

    return axios.get(`/api/getRecommend?date=${Number(date)}`
    ).then(response => {
      return Promise.resolve(response.data)
    }).catch(err => {
      console.log(err)
    })
}
```

当使用axios成功请求/api/getRecommend时，axios返回一个Promise对象，因为getRecommend()是要export出去的，这里直接返回一个状态完成的Promise，调用getRecommend()时，如果成功响应直接可以recommend.getRecommend().then(res => {})获取响应结果。

Promise.all()、Promise.race()并行执行多个Promise对象

- Promise.all()是所有Promise对象状态都是'已成功'
- Promise.race()是有一个Promise对象状态'已成功'