var p = new Promise(function(resolve, reject){

    //做一些异步操作

    setTimeout(function(){

        console.log('执行完成');

        resolve('随便什么数据');

    }, 2000);

});

Promise的构造函数接收一个函数参数。

resolve是将Promise的状态置为fullfiled，

reject是将Promise的状态置为rejected。

运行代码，会在2秒后输出“执行完成”。注意！我只是new了一个对象，并没有调用它，我们传进去的函数就已经执行了

**Promise使用**

我们用Promise的时候一般是包在一个函数中，在需要的时候去运行这个函数，如：

function runAsync(){

    var p = new Promise(function(resolve, reject){

        //做一些异步操作

        setTimeout(function(){

            console.log('执行完成');

            resolve('随便什么数据');

        }, 2000);

    });

    return p;

}

runAsync()

执行这个函数我们得到了一个Promise对象。看下面的代码：

runAsync().then(function(data){

    console.log(data);

    //后面可以用传过来的数据做些其他操作

    //......

});

直接调用then方法，then接收一个参数，是函数，其参数是resolve时传的的参数或者上一个then返回的参数

then返回一个新的Promise对象

## ****链式操作的用法****

使用Promise的正确场景是这样的：

runAsync1()

    .then(function (data) {

        console.log(data);

        return runAsync2();

    })

    .then(function (data) {

        console.log(data);

        return "data3";

    })

    .then(function (data) {

        console.log(data);

    });

## **Then中可以直接返回数据，或者返回一个Promise，**

**如果直接返回数据则会马上调用下一个then**

**如果返回Promise，则等待该Promise执行完毕，并接收该Promise的结果拿去调用下一个then**

## ****reject的用法****

reject的作用就是把Promise的状态置为rejected

then可以接受第二个参数，该参数是一个处理rejected状态的函数

function getNumber(){

    var p = new Promise(function(resolve, reject){

        //做一些异步操作

        setTimeout(function(){

            var num = Math.ceil(Math.random()\*10); //生成1-10的随机数

            if(num<=5){

                resolve(num);

            }

            else{

                reject('数字太大了');

            }

        }, 2000);

    });

    return p;

}

getNumber()

.then(

    function(data){

        console.log('resolved');

        console.log(data);

    },

    function(reason, data){

        console.log('rejected');

        console.log(reason);

    }

);

## ****catch的用法****

其实它和then的第二个参数一样，用来指定reject的回调，用法是这样：

getNumber()

    .then(function (data) {

        console.log('resolved');

        console.log(data);

    })

    .catch(function (reason) {

        console.log('rejected');

        console.log(reason);

    });

不过它还有另外一个作用：在catch之前的处理流程中，如果抛出异常了，那么并不会报错卡死js，而是会进到这个catch方法中。这与我们的try/catch语句有相同的功能。

**在catch中 中断其后的then**

promisefun()

    .then(

        value => {

            // ...

        }

    )

    .catch(

        errorData => {

            dispatch(error(errorData.message));

            // 如果发生错误，将中断其后then的执行

            return Promise.reject(errorData.message);

        }

    ).then(

        value => {

        }

    )

## ****all的用法****

Promise的all方法提供了并行执行异步操作的能力，并且在所有异步操作执行完后才执行回调。看下面的例子：

Promise

    .all([runAsync1(), runAsync2(), runAsync3()])

    .then(function (results) {

        console.log(results);

    });

等到它们都执行完后才会进到then里面

## ****race的用法****

**all方法的效果实际上是「谁跑的慢，以谁为准执行回调」，那么相对的就有另一个方法「谁跑的快，以谁为准执行回调」，这就是race方法，这个词本来就是赛跑的意思。**

Promise

    .race([runAsync1(), runAsync2(), runAsync3()])

    .then(function (results) {

        console.log(results);

    });

**ES7 Async/await使用**

Async/await是Promise的语法糖，如下示例

  return async function (dispatch:any) {

...

let token = await IEToken.getToken();

...

    return await fetch(fullUrl, {

      method: 'post',

      headers: headers,

      body: JSON.stringify(postData)

}).then(

...

    ).catch(

....

    ).then(

...

    )

  }