Promise是一个构造函数，自己身上有all、reject、resolve这几个眼熟的方法，原型上有then、catch等同样很眼熟的方法。

那就new一个

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参数，是函数，并且传入两个参数：resolve，reject，分别表示异步操作执行成功后的回调函数和异步操作执行失败后的回调函数。

在上面的代码中，我们执行了一个异步操作，也就是setTimeout，2秒后，输出“执行完成”，并且调用resolve方法。

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，这是需要注意的一个细节。所以我们用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()

在我们包装好的函数最后，会return出Promise对象，也就是说，执行这个函数我们得到了一个Promise对象。这是我们就可以应用promise的then和catch方法，进行扩展使用。

runAsync().then(function(data){

console.log(data);

//后面可以用传过来的数据做些其他操作

//......

});

在runAsync()的返回上直接调用then方法，then接收一个参数，是函数，并且会拿到我们在runAsync中调用resolve时传的的参数。运行这段代码，会在2秒后输出“执行完成”，紧接着输出“随便什么数据”。

这时候你应该有所领悟了，原来then里面的函数就跟我们平时的回调函数一个意思，能够在runAsync这个异步任务执行完成之后被执行。这就是Promise的作用了，简单来讲，就是能把原来的回调写法分离出来，在异步操作执行完后，用链式调用的方式执行回调函数。

function runAsync(callback){

setTimeout(function(){

console.log('执行完成');

callback('随便什么数据');

}, 2000);

}

runAsync(function(data){

console.log(data);

});

如上我们可以只通过callback来实现回调。但为什么还要使用promise呢。

因为当我们的程序需要多层回调时，我们不能在callback里传入callback2，但是我们可以在promise的then回调方法中重新定义一个promise对象然后继续调用then完成多层回调。

**链式操作的用法**

从表面上看，Promise只是能够简化层层回调的写法，而实质上，Promise的精髓是“状态”，用维护状态、传递状态的方式来使得回调函数能够及时调用，它比传递callback函数要简单、灵活的多。所以使用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 runAsync3();

})

.then(function(data){

console.log(data);

});

function runAsync1(){

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

//做一些异步操作

setTimeout(function(){

console.log('异步任务1执行完成');

resolve('随便什么数据1');

}, 1000);

});

return p;

}

function runAsync2(){

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

//做一些异步操作

setTimeout(function(){

console.log('异步任务2执行完成');

resolve('随便什么数据2');

}, 2000);

});

return p;

}

function runAsync3(){

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

//做一些异步操作

setTimeout(function(){

console.log('异步任务3执行完成');

resolve('随便什么数据3');

}, 2000);

});

return p;

}

输出结果为：



在then方法中，你也可以直接return数据而不是Promise对象，在后面的then中就可以接收到数据了，比如我们把上面的代码修改成这样：

runAsync1()

.then(function(data){

console.log(data);

return runAsync2();

})

.then(function(data){

console.log(data);

return '直接返回数据'; //这里直接返回数据

})

.then(function(data){

console.log(data);

});

## ****reject的用法****

我们光用了resolve，还没用reject呢，它是做什么的呢？事实上，我们前面的例子都是只有“执行成功”的回调，还没有“失败”的情况，reject的作用就是把Promise的状态置为rejected，这样我们在then中就能捕捉到，然后执行“失败”情况的回调。看下面的代码。

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);

}

);

getNumber函数用来异步获取一个数字，2秒后执行完成，如果数字小于等于5，我们认为是“成功”了，调用resolve修改Promise的状态。否则我们认为是“失败”了，调用reject并传递一个参数，作为失败的原因。

运行getNumber并且在then中传了两个参数，then方法可以接受两个参数，第一个对应resolve的回调，第二个对应reject的回调。所以我们能够分别拿到他们传过来的数据。多次运行这段代码，你会随机得到下面两种结果：

https://images2015.cnblogs.com/blog/520134/201603/520134-20160311004607960-1156803894.png或者https://images2015.cnblogs.com/blog/520134/201603/520134-20160311004616257-1024778840.png

**catch的用法**

我们知道Promise对象除了then方法，还有一个catch方法，它是做什么用的呢？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：在执行resolve的回调（也就是上面then中的第一个参数）时，如果抛出异常了（代码出错了），那么并不会报错卡死js，而是会进到这个catch方法中。请看下面的代码：

getNumber()

.then(function(data){

console.log('resolved');

console.log(data);

console.log(somedata); //此处的somedata未定义

})

.catch(function(reason){

console.log('rejected');

console.log(reason);

});

这与我们的try/catch语句有相同的功能。

**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。我们仍旧使用上面定义好的runAsync1、runAsync2、runAsync3这三个函数，看下面的例子：

Promise

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

.then(function(results){

console.log(results);

});

## https://images2015.cnblogs.com/blog/520134/201603/520134-20160311004843491-346782307.png****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。race的用法与all一样，我们把上面runAsync1的延时改为1秒来看一下：

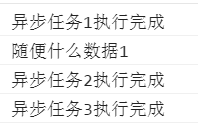
Promise

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

.then(function(results){

console.log(results);

});



这三个异步操作同样是并行执行的。结果你应该可以猜到，1秒后runAsync1已经执行完了，此时then里面的就执行了。

在then里面的回调开始执行时，runAsync2()和runAsync3()并没有停止，仍旧再执行。于是再过1秒后，输出了他们结束的标志。

//请求某个图片资源

function requestImg(){

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

var img = new Image();

img.onload = function(){

resolve(img);

}

img.src = 'xxxxxx';

});

return p;

}

//延时函数，用于给请求计时

function timeout(){

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

setTimeout(function(){

reject('图片请求超时');

}, 5000);

});

return p;

}

Promise

.race([requestImg(), timeout()])

.then(function(results){

console.log(results);

})

.catch(function(reason){

console.log(reason);

});

requestImg函数会异步请求一张图片，我把地址写为"xxxxxx"，所以肯定是无法成功请求到的。timeout函数是一个延时5秒的异步操作。我们把这两个返回Promise对象的函数放进race，于是他俩就会赛跑，如果5秒之内图片请求成功了，那么遍进入then方法，执行正常的流程。如果5秒钟图片还未成功返回，那么timeout就跑赢了，则进入catch，报出“图片请求超时”的信息。运行结果如下：

https://images2015.cnblogs.com/blog/520134/201603/520134-20160311005040272-341718790.png