* **Promise**

var promise = new Promise(function(resolve, reject) { ... });

*promise*

必需。承诺分配到的变量名称。

*resolve*

必需。一个函数，运行该函数可指示承诺已成功完成。

可以传递最多一个参数：如果不传递参数则是 undefined，参数类型也是灵活的

then 方法里对应的方法是第一个函数： onCompleted

*reject*

可选。一个函数，运行该函数可指示出现错误，承诺已被拒绝。

可以传递最多一个参数：如果不传递参数则是 undefined, 参数类型也是灵活的

then 方法里对应的方法是第二个函数： onRejected s

# Promise对象 then 方法

# promise.then(onCompleted, onRejected);

例如：

var mypromise = new Promise(

function (**ok, fail**) {

let msg = "";

if (false) {

msg = { status: "ok", message: "Good Case" };

**ok**(msg);

} else {

msg = "Bad Case";

**fail**(msg);

}

}

);

mypromise.then(

()=>console.log("Okey Okey"), -- 可以不带参数接收

()=>console.log("Failure")

);

mypromise.then(

a=>console.log(a), -- 也可以带参数接收

b=>console.log(b)

);

直接 resolve, reject

Promise

.resolve({name:"will", age:45})

.then(a => { console.log("Resolve"); console.log(a); })

Promise

.reject( 2888 )

.then(

a => { console.log("Resolve"); console.log(a); },

b => { console.log("Reject"); console.log(b); } -- 执行 onReject 函数

);

Promise.reject(6666)

.then(a => console.log(a)) -- 如果没有定义 onRejectCallBack, 则向下传递

.catch(err=>console.log("Catch: " + err)); -- 此时捕捉到

Promise.reject(6666)

.then(a => console.log(a)) -- 向下传递

.then(a => console.log(a), a=>console.log("then error: " + a)) -- 在此捕捉到reject

.catch(err => console.log("Catch: " + err)); -- 由于上面捕捉到， 所以此处不执行。

Promise

.reject(2888)

.catch(err => { console.log("Reject Catch Error: " + err); return Promise.reject(err); })

.then(

a => { console.log("Reject Resolve: " + a); },

b => { console.log("Reject Error: " + b); }

);

Promise

.reject(2888)

.catch(err => { console.log("Reject Catch Error: " + err); return Promise.reject(err); })

.then(

a => { console.log("Reject Resolve: " + a); },

b => { console.log("Reject Error: " + b); }

);

.catch回调函数，用于接收 reject 方法所返回的数据。 同时它本身也可以返回数据到下一个链式函数里。

Promise.reject(6666)

.then(a => console.log(a))

.then(a => console.log(a), a=>console.log("then error: " + a)) -- 在这里捕获reject 回调函数

.catch(err => console.log("Catch: " + err)); -- 由于上面已经捕获回调函数，此处不执行，除非前面有

return Promise.reject(xxx);

.finally(

-- 是不带参数的必须执行的函数，它的位置可以是随意的，执行的先后顺序也是按照所在位置， 并非一定放在最后。 可

以放在最前面， 则最先执行。

() => console.log("Finally: ")

);

注意： Promise 没有 when 方法, 但是 Promise 有 all 方法

function funcA(a) {

if (a >= 100)

return Promise.resolve({ status: "ok", data: a });

else

return Promise.reject({ status: "fail", errorCode:999, errorMessage:"Must>=100", data: a });

}

Promise.all( [funcA(99)] ) -- 必须是函数数组

-– 如果成功返回数据也是数组相对应各个函数。

.then(a => { console.log("Then:"); console.log(a[0]); })

-–如果是reject, 返回第一个reject，所以返回的数据不是数组，而是那个reject的数据

.catch(a => { console.log("catch error"); console.log(a); })

.finally(console.log("funcA exec finished!!!"));

同时执行多个函数：

function funcA(a) {

if (a >= 100)

return Promise.resolve({ status: "Aok", data: a });

else

return Promise.reject({ status: "Afail", errorCode:999, errorMessage:"Must>=100", data: a });

}

function funcB(a) {

if (a >= 1000)

return Promise.resolve({ status: "Bok", data: a });

else

return Promise.reject({ status: "Bfail", errorCode: 999, errorMessage: "Must>=100", data: a });

}

function funcC(a) {

if (a >= 2000)

return Promise.resolve({ status: "Cok", data: a });

else

return Promise.reject({ status: "Cfail", errorCode: 999, errorMessage: "Must>=100", data: a });

}

Promise.all([funcA(100), funcB(1000), funcC(2000)])

.then(a => { console.log("Then:"); console.log(a); })

.catch(a => { console.log("catch error"); console.log(a); })

.finally(console.log("funcA exec finished!!!"));

输出也是数组：

[…]

0: Object { status: "Aok", data: 100 }

​

1: Object { status: "Bok", data: 1000 }

​

2: Object { status: "Cok", data: 2000 }

​

length: 3

​

\_\_proto\_\_: Array []

Promise.all([

funcA(180),

funcB(500), --- reject case

funcC(800) --- reject case

]

).then(a => { console.log("Then:"); console.log(a); })

.catch(a => { console.log("catch error"); console.log(a); })

.finally(console.log("funcA exec finished!!!"));

对于多个 reject 的情况， 是只会返回最先的一个 reject case, 对于回调函数接收的数据不象resolve返回数组， 而只返回此reject 的数据。

Object { status: "Bfail", errorCode: 999, errorMessage: "Must>=100", data: 500 }

Promise.race – 捕获第一个最先成功的 resove.

Promise.race([

funcA(80), -- reject

funcB(1500), -- resolve

funcC(800)] -- reject

).then(

a => { console.log("Then:"); console.log(a); },

b => { console.log("Then Reject:"); console.log(b); } – 触发

);

由于第一个是 reject case，所以不会往下走，直接被触发

//异步的案例

function funcA(a) {

var prom = $.Deferred();

if (a >= 100)

setTimeout( ()=>prom.resolve({ status: "Aok", data: a }), 1200);

else

setTimeout(() => prom.reject({ status: "Afail", errorCode: 999, errorMessage: "Must>=100", data: a }), 1200);

return prom.promise();

}

function funcB(a) {

var prom = $.Deferred();

if (a >= 1000)

setTimeout(() => prom.resolve({ status: "Bok", data: a }), 1100);

else

setTimeout(() => prom.reject({ status: "Bfail", errorCode: 999, errorMessage: "Must>=100", data: a }), 1100);

return prom.promise();

}

function funcC(a) {

var prom = $.Deferred();

if (a >= 2000)

setTimeout(() => prom.resolve({ status: "Cok", data: a }), 1000);

else

setTimeout(() => prom.reject({ status: "Cfail", errorCode: 999, errorMessage: "Must>=100", data: a }), 1000);

return prom.promise();

}

Promise.race([

funcA(80), -- reject order by time: 3

funcB(1500), -- resolve order by time: 2

funcC(2800)] – resolve order by time: 1 - 此处触发

).then(

a => { console.log("Then:"); console.log(a); },

b => { console.log("Then Reject:"); console.log(b); }

);

执行结果：

Then: [good.js:51:5](webpack://src/good.js?)

Object { status: "Cok", data: 2800 }

Promise.race 是指按照时间顺序第一个被执行的函数， 不管是resolve 或者是 reject

定义Promise 函数的方法来使用：

function funcA(a) {

let prom = new Promise(function (resolve, reject) {

if (a >= 100)

resolve({ status: "ok", data: a });

else

reject({ status: "fail", errorCode:999, errorMessage:"Must>=100", data: a });

});

return prom; -- 函数的返回值是一个 Promise 对象

}

funcA(99).then(console.log)

.catch(console.log)

.finally(console.log("funcA exec finished"));

函数里面没有异步，直接返回的是 Promise.resolve(xxx) 或者是 Promise.reject(xxx) 函数的执行后的返回值。

function funcA(a) {

if (a >= 100)

return Promise.resolve({ status: "ok", data: a });

else

return Promise.reject({ status: "fail",

errorCode:999,

errorMessage:"Must>=100",

data: a });

}

funcA(89).then(a => { console.log("Then:"); console.log(a); })

.catch(a => { console.log("catch error"); console.log(a); })

.finally(console.log("funcA exec finished!!!"));

Promise 主要优势是处理异步，传统的异步处理主要是使用回调函数。总的来说，传统的回调函数写法使得代码混成一团，变得横向发展而不是向下发展。

如何把传统的 ajax 转成 Promise 的链式方法：

$.ajax – Ajax in Jquery , 则可以很方便使用 jquery 的 promise - $.Deferred()

function ajaxCall() {

let defer = $.Deferred();

$.ajax({

crossDomain: true,

dataType: 'json',

data: {

keys: [100,200]

},

contentType: "application/json",

error: function (xhr, tStatus, errorTh) {

defer.reject(tStatus + ": " + errorTh);

},

success: function (respData, tStatus) {

defer.resolve(respData);

},

type: "get",

url: "/persons.json"

});

return defer.promise(); - 特别注意,需要调用方法 promise()来获取

}

ajaxCall().then(

a => { console.log("Then Resolved: "); console.log(a) },

a => { console.log("Then Reject: "); console.log(a) }

)

.catch(

err => { console.log("Catch Error: "); console.log(err) }

)

.always(

()=>console.log("ajaxCall Completed!!!")

);

$.Deferred() 没有 .finally 而只有 .always

$.Deferred() -- promise() 支持的回调函数名称：

1. done ( success or resolve )
2. fail ( reject )
3. always ( finally )
4. catch (same as promise)

即如上所述

ajaxCall().done(

a => { console.log("Then Resolved: "); console.log(a) },

)

.fail(

err => { console.log("Catch Error: "); console.log(err) }

)

.always(

()=>console.log("ajaxCall Completed!!!")

);

其他 jquery 异步方法：

function ajaxCall() {

return $.get("/persons.json");

}

ajaxCall().done(

a => { console.log("Then Resolved: "); console.log(a) },

)

.fail( -- 注意如果此例子错误发生，则传进来的是 return $.get – ajax object

err => { console.log("Catch Error: "); console.log(err) }

)

.always(

()=>console.log("ajaxCall Completed!!!")

);

$http - Ajax in Angualr, 则可以很方便使用 jquery 的 promise - $.Deferred()

return $http({

url: api\_url,

method: p\_method,

headers: { 'Content-Type': 'application/json' },

params: p\_params

});

return $http({…}); 就相当于直接返回一个 Promise 对象，

$q - $q in AngularJS 的使用来创建 Promise

factory.postData = function (p\_url, p\_method, p\_params) {

var api\_url = API\_URL + p\_url;

var defer = $q.defer();

$http({

url: api\_url,

method: p\_method,

headers: { 'Content-Type': 'application/json' },

params: p\_params

}).then(

function (respData) {

defer.resolve(respData);

},

function (respData) {

defer.reject(respData);

}

);

return defer.promise; -特别注意: promise是属性而不是方法, 不需要加(),和jquery不同

}

// 传统写法

step1(function (value1) {

  step2(value1, function(value2){

    step3(value2,function(value3) {

      step4(value3,function(value4) {

        // ...

      });

    });

  });

});

按自定义的顺序来执行异步。 这个和

Promise.all([f1(), f2(), f3(),….]) – 执行的顺序和时间前后没有任何逻辑关系

下面的案例和使用方法，则是对异步方法按自定义的步骤来执行：

// 这个是一个通用的 Ajax 异步调用函数， 根据 url 不同获取不同数据

function ajaxCall(p\_url) {

let prom = new Promise(

(resolve, reject) => {

$.ajax({

crossDomain: true,

dataType: 'json',

data: {

keys: [100, 200]

},

contentType: "application/json",

error: function (xhr, tStatus, errorTh) {

reject(tStatus + ": " + errorTh);

},

success: function (respData, tStatus) {

resolve(respData);

},

type: "get",

url: p\_url

});

}

)

return prom;

}

ajaxCall("/persons.json").then(

a => { console.log("Persons JSON:"); console.log(a); return ajaxCall("/persons22.json"); },

a => { console.log("Persons Error:"); console.log(a); return ajaxCall("/persons22.json"); }

)

.then(

a => { console.log("Persons22 JSON:"); console.log(a); return ajaxCall("/persons33.json"); },

a => { console.log("Persons22 Error:"); console.log(a); return ajaxCall("/persons33.json"); }

)

.then(

a => { console.log("Persons33 JSON:"); console.log(a); return a; },

a => { console.log("Persons33 Error:"); console.log(a); return a; }

)

.then(

a => { console.log("Last Step:"); console.log(a); }

)

.finally(

() => console.log("Ajax Call Completed! Bye")

);

以上是在 第一个 ajax 完成以后，在 then 回调函数里，再对第二个 ajax 调用，以此类推，实现按自定义步骤先后执行。这只是用编程技巧去实现

[**https://blog.csdn.net/jasonzds/article/details/53717501**](https://blog.csdn.net/jasonzds/article/details/53717501)

**JavaScript再述promise的用法和常用方法**

原创 2016年12月17日 23:53:22

**1 Promise概述**

Promise对象是CommonJS工作组提出的一种规范，目的是为异步操作提供[统一接口](http://wiki.commonjs.org/wiki/Promises/A" \t "_blank)。

那么，什么是Promises？

首先，它是一个对象，也就是说与其他JavaScript对象的用法，没有什么两样；其次，它起到代理作用（proxy），充当异步操作与回调函数之间的中介。它使得异步操作具备同步操作的接口，使得程序具备正常的同步运行的流程，回调函数不必再一层层嵌套。

简单说，它的思想是，每一个异步任务立刻返回一个Promise对象，由于是立刻返回，所以可以采用同步操作的流程。这个Promises对象有一个then方法，允许指定回调函数，在异步任务完成后调用。

比如，异步操作f1返回一个Promise对象，它的回调函数f2写法如下。

(new Promise(f1)).then(f2);

这种写法对于多层嵌套的回调函数尤其方便。

// 传统写法

step1(function (value1) {

  step2(value1, function(value2){

    step3(value2,function(value3) {

      step4(value3,function(value4) {

        // ...

      });

    });

  });

});

// Promises的写法

(new Promise(step1))

  .then(step2)

  .then(step3)

  .then(step4);

从上面代码可以看到，采用Promises接口以后，程序流程变得非常清楚，十分易读。

注意，为了便于理解，上面代码的Promise对象的生成格式，做了简化，真正的语法请参照下文。

总的来说，传统的回调函数写法使得代码混成一团，变得横向发展而不是向下发展。Promises规范就是为了解决这个问题而提出的，目标是使用正常的程序流程（同步），来处理异步操作。它先返回一个Promise对象，后面的操作以同步的方式，寄存在这个对象上面。等到异步操作有了结果，再执行前期寄放在它上面的其他操作。

Promises原本只是社区提出的一个构想，一些外部函数库率先实现了这个功能。ECMAScript 6将其写入语言标准，因此目前JavaScript语言原生支持Promise对象。

前面说过，Promise接口的基本思想是，异步任务返回一个Promise对象。

Promise对象只有三种状态。

•   异步操作“未完成”（pending）

•   异步操作“已完成”（resolved，又称fulfilled）

•   异步操作“失败”（rejected）

这三种的状态的变化途径只有两种。

•   异步操作从“未完成”到“已完成”

•   异步操作从“未完成”到“失败”。

这种变化只能发生一次，一旦当前状态变为“已完成”或“失败”，就意味着不会再有新的状态变化了。因此，Promise对象的最终结果只有两种。

•   异步操作成功，Promise对象传回一个值，状态变为resolved。

•   异步操作失败，Promise对象抛出一个错误，状态变为rejected。

Promise对象使用then方法添加回调函数。then方法可以接受两个回调函数，第一个是异步操作成功时（变为resolved状态）时的回调函数，第二个是异步操作失败（变为rejected）时的回调函数（可以省略）。一旦状态改变，就调用相应的回调函数。

// po是一个Promise对象

po.then(

  console.log,

  console.error

);

上面代码中，Promise对象po使用then方法绑定两个回调函数：操作成功时的回调函数console.log，操作失败时的回调函数console.error（可以省略）。这两个函数都接受异步操作传回的值作为参数。

then方法可以链式使用。

po

  .then(step1)

  .then(step2)

  .then(step3)

  .then(

    console.log,

    console.error

  );

上面代码中，po的状态一旦变为resolved，就依次调用后面每一个then指定的回调函数，每一步都必须等到前一步完成，才会执行。最后一个then方法的回调函数console.log和console.error，用法上有一点重要的区别。console.log只显示回调函数step3的返回值，而console.error可以显示step1、step2、step3之中任意一个发生的错误。也就是说，假定step1操作失败，抛出一个错误，这时step2和step3都不会再执行了（因为它们是操作成功的回调函数，而不是操作失败的回调函数）。Promises对象开始寻找，接下来第一个操作失败时的回调函数，在上面代码中是console.error。这就是说，Promises对象的错误有传递性。

从同步的角度看，上面的代码大致等同于下面的形式。

try {

  var v1 = step1(po);

  var v2 = step2(v1);

  var v3 = step3(v2);

  console.log(v3);

} catch (error) {

  console.error(error);

}

ES6提供了原生的Promise构造函数，用来生成Promise实例。

下面代码创造了一个Promise实例。

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

  // 异步操作的代码

  if (/\* 异步操作成功 \*/){

    resolve(value);

  } else {

    reject(error);

  }

});

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由JavaScript引擎提供，不用自己部署。

resolve函数的作用是，将Promise对象的状态从“未完成”变为“成功”（即从Pending变为Resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；reject函数的作用是，将Promise对象的状态从“未完成”变为“失败”（即从Pending变为Rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

Promise实例生成以后，可以用then方法分别指定Resolved状态和Reject状态的回调函数。

po.then(function(value) {

  // success

}, function(value) {

  // failure

});

Promise的用法，简单说就是一句话：使用then方法添加回调函数。但是，不同的写法有一些细微的差别，请看下面四种写法，它们的差别在哪里？

// 写法一

doSomething().then(function () {

  return doSomethingElse();

});

// 写法二

doSomething().then(function () {

  doSomethingElse();

});

// 写法三

doSomething().then(doSomethingElse());

// 写法四

doSomething().then(doSomethingElse);

为了便于讲解，下面这四种写法都再用then方法接一个回调函数finalHandler。写法一的finalHandler回调函数的参数，是doSomethingElse函数的运行结果。

doSomething().then(function () {

  return doSomethingElse();

}).then(finalHandler);

写法二的finalHandler回调函数的参数是undefined。

doSomething().then(function () {

  doSomethingElse();

  return;

}).then(finalHandler);

写法三的finalHandler回调函数的参数，是doSomethingElse函数返回的回调函数的运行结果。

doSomething().then(doSomethingElse())

  .then(finalHandler);

写法四与写法一只有一个差别，那就是doSomethingElse会接收到doSomething()返回的结果。

doSomething().then(doSomethingElse)

  .then(finalHandler);

**2 promise.then**

所有的 promise 对象实例里都有一个then 方法，它是用来跟这个 promise 进行交互的。首先，then 方法会缺省调用 resolve() 函数:

new Promise(function(resolve, reject) {

     // Amock async action using setTimeout

     setTimeout(function(){ resolve(10); }, 3000);

})

.then(function(result) {

     console.log(result);

});

// From the console:

// 10

then 回调动作的触发时机是 promise 被执行完。我们还可以串联 then 方法执行回调操作：

new Promise(function(resolve, reject) {

     // Amock async action using setTimeout

     setTimeout(function(){ resolve(10); }, 3000);

})

.then(function(num) { console.log('first then:', num); return num \* 2; })

.then(function(num) { console.log('second then:', num); return num \* 2; })

.then(function(num) { console.log('last then: ',num);});

// From the console:

// first then: 10

// second then: 20

// lastthen:  40

你会发现，每次 then 调用都会以之前的 then 调用的返回值为参数。

如果一个 promise 已经执行完成，单then 被再次调用时，回调动作将会被再次执行。而如果这个 promise 里执行的是reject 回调函数，这是再调用 then 方法，回调函数将不会被执行。

**3 promise.all**

在我们的异步调用时经常有这样一种场景：我们需要同时调用多个异步操作，但希望只有等所有的操作都完成后，我们才去执行响应操作——这就是Promise.all 的作用。 Promise.all 方法可以接收多个 promise 作为参数，以数组的形式，当这些 promise 都成功执行完成后才调用回调函数。

Promise.all([promise1, promise2]).then(function(results){

     //Both promises resolved

})

.catch(function(error) {

     //One or more promises was rejected

});

一个很好的能演示 Promise.all 用法的例子是，执行多个 AJAX 操作(通过 fetch) 调用:

var request1 = fetch('/users.json');

var request2 = fetch('/articles.json');

Promise.all([request1, request2]).then(function(results) {

     // Both promises done!

});

一旦promise 里调用了reject函数，也就是执行被拒绝了，没有能够正常完成，情况会有些复杂。一旦promise 被拒绝，catch 方法会捕捉到首个被执行的reject函数：

var req1 = new Promise(function(resolve, reject){

     // Amock async action using setTimeout

     setTimeout(function(){ resolve('First!'); }, 4000);

});

var req2 = new Promise(function(resolve, reject){

     // Amock async action using setTimeout

     setTimeout(function(){ reject('Second!'); }, 3000);

});

Promise.all([req1, req2]).then(function(results){

     console.log('Then:', one);

}).catch(function(err) {

     console.log('Catch:', err);

});

// From the console:

// Catch:Second!

**4 promise.catch**

catch 当一个 promise 被拒绝(reject)时,catch 方法会被执行：

new Promise(function(resolve,reject) {

     // A mock async action using setTimeout

     setTimeout(function() { reject('Done!'); },3000);

})

.then(function(e){ console.log('done', e); })

.catch(function(e){ console.log('catch: ', e); });

// From theconsole:

// 'catch:Done!'

通常我们在 reject 方法里处理执行失败的结果，而在catch 里执行异常结果：

reject(Error('Data could not be found'));

**5 promise.race**

Promise.race 是一个有趣的函数——它不是等待所有的 promise 被resolve 或 reject，而是在所有的 promise 中只要有一个执行结束，它就会触发：

var req1 = newPromise(function(resolve, reject) {

     // A mock async action using setTimeout

     setTimeout(function() { resolve('First!'); },8000);

});

var req2 = newPromise(function(resolve, reject) {

     // A mock async action using setTimeout

     setTimeout(function() { resolve('Second!');}, 3000);

});

Promise.race([req1,req2]).then(function(one) {

     console.log('Then: ', one);

}).catch(function(one,two) {

     console.log('Catch: ', one);

});

// From theconsole:

// Then:Second!

一个有用的场景是，从多个镜像服务器下载资源，一旦有一个返回，其它的返回也就不用处理了。

**6 promise实际应用**

一个 XMLHttpRequest 调用转换为基于 Promises 的任务：

// From JakeArchibald's Promises and Back:

//http://www.html5rocks.com/en/tutorials/es6/promises/#toc-promisifying-xmlhttprequest

function get(url){

  // Return a new promise.

  return new Promise(function(resolve, reject) {

    // Do the usual XHR stuff

    var req = new XMLHttpRequest();

    req.open('GET', url);

    req.onload = function() {

      // This is called even on 404 etc

      // so check the status

      if (req.status == 200) {

        // Resolve the promise with theresponse text

        resolve(req.response);

      }

      else {

        // Otherwise reject with the statustext

        // which will hopefully be a meaningfulerror

        reject(Error(req.statusText));

      }

    };

    // Handle network errors

    req.onerror = function() {

      reject(Error("Network Error"));

    };

    // Make the request

    req.send();

  });

}

// Use it!

get('story.json').then(function(response){

  console.log("Success!", response);

}, function(error){

  console.error("Failed!", error);

});

Promise.resolve() 和 Promise.reject() 可以直接被调用。有时候，当判断出 promise 并不需要真正执行时，我们并不需要 使用 new 创建 Promise 对象，而是可以直接调用 Promise.resolve() 和 Promise.reject()。比如：

var userCache = {};

function getUserDetail(username) {

  // In both cases, cached or not, apromise will be returned

  if (userCache[username]) {

     //Return a promise without the "new" keyword

    return Promise.resolve(userCache[username]);

  }

  // Use the fetch API to get theinformation

  // fetch returns a promise

  return fetch('users/' + username + '.json')

    .then(function(result) {

      userCache[username] = result;

      return result;

    })

    .catch(function() {

      throw new Error('Could not finduser: ' + username);

});

}

因为 promise 肯定会返回，所以，我们可以使用then 和 catch 方法处理返回值！

**加载图片**

var preloadImage = function (path) {

  return new Promise(function(resolve, reject) {

    var image = newImage();

    image.onload  = resolve;

    image.onerror= reject;

    image.src = path;

  });

};