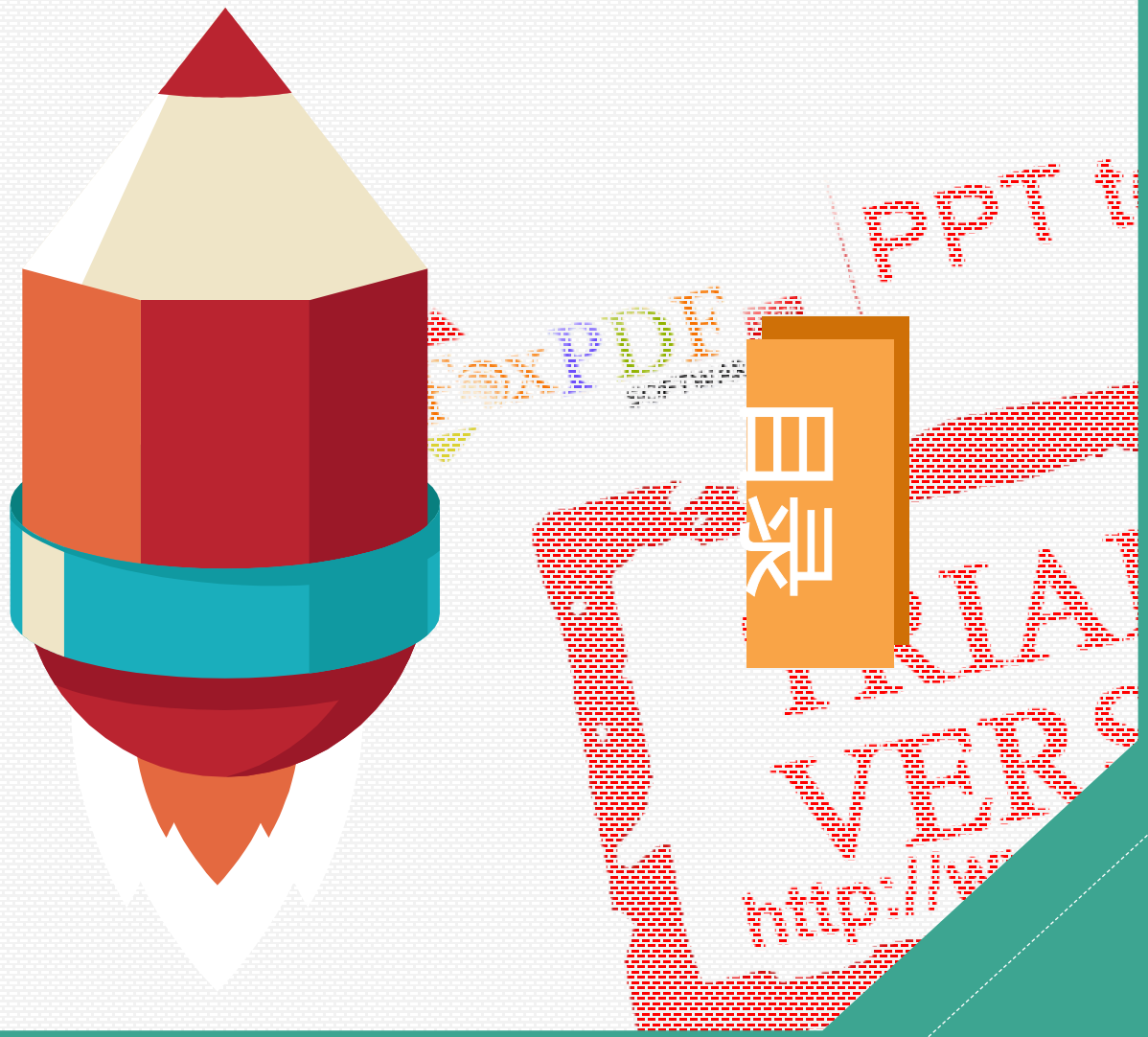
第39课：异步应用(generator和promise对象)

主讲老师：万章



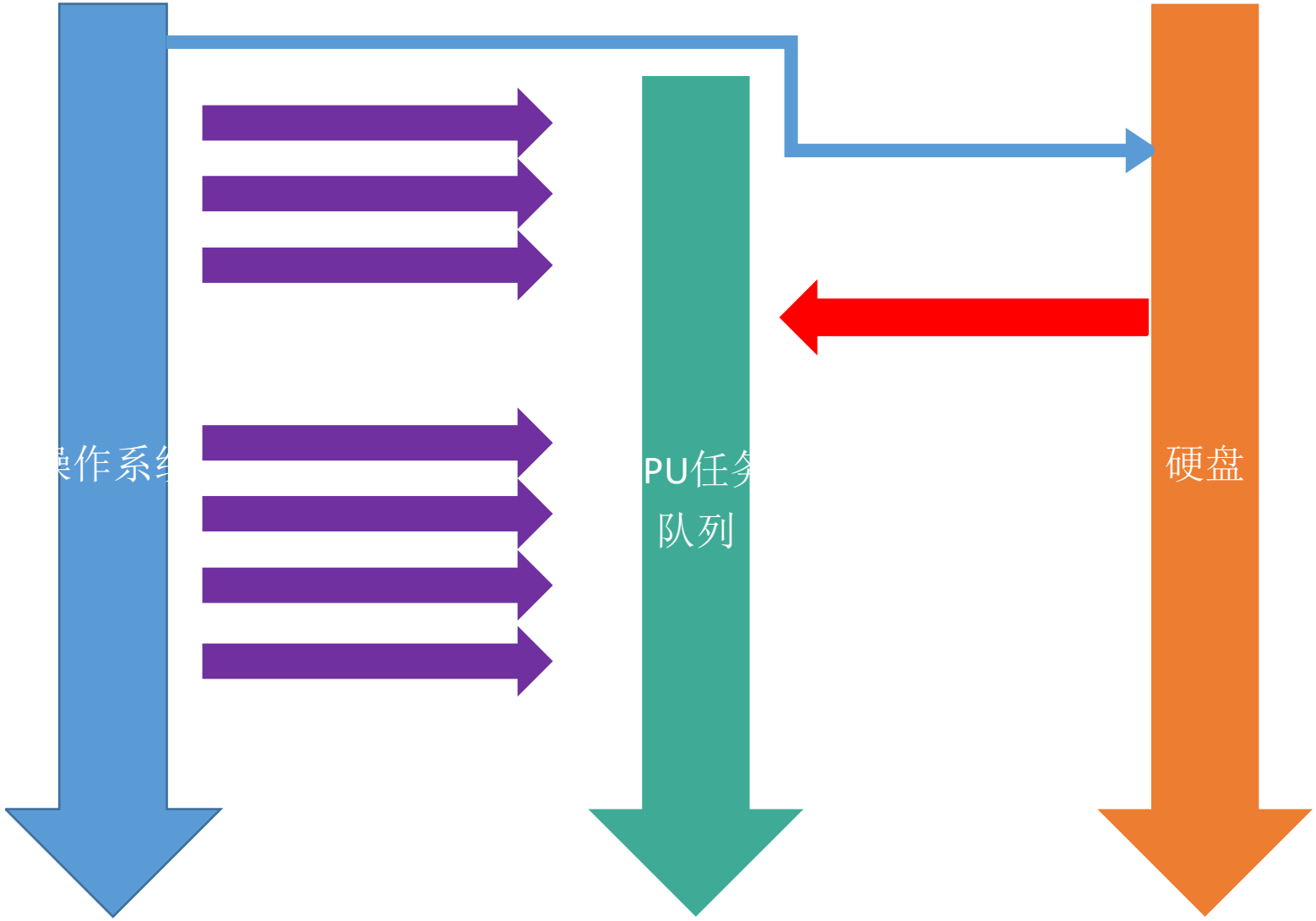
异步的基本概念

Thunk 函数

Promise对象

什么是异步？

什么是异步？



所谓“异步”，简单说就是一个任务不是连续完成的，可以理解成该任务被人为分成两段，先执行第一段，然后转而执行其他任务，等做好了准备，再回过头执行第二段

比如，有一个任务是读取文件进行处理，任务的第一段是向操作系统发出请求，要求读取文件。然后，程序执行其他任务，等到操作系统返回文件，再接着执行任务的第二段（处理文件）。这种不连续的执行，就叫做异步。

相应地，**连续的执行就叫做同步**。由于是连续执行，不能插入其他任务，所以操作系统从硬盘读取文件的这段时间，程序只能干等着

回调函数callback

JavaScript 语言对异步编程的实现，就是回调函数。所谓回调函数，就是把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，就直接调用这个函数。回调函数的英语名字callback，直译过来就是"重新调用"。

```
fs.readFile('/etc/passwd', 'utf-8', function (err, data) {
    if (err) throw err;
    console.log(data);
});
```

node的文件读取操作

上面代码中，readFile函数的第三个参数，就是回调函数，也就是任务的第二段。等到操作系统返回了/etc/passwd这个文件以后，回调函数才会执行

```
function  add(num1,num2,callback) {
    let sum = num1+num2;
    return callback(sum);//回调函数
}

function squa(num){
    return num*num;
}

add(5,5,squa);//100
```

上面代码中，add函数的第三个参数，就是回调函数，也就是任务的第二段。等到add函数计算完了sum的值，才会执行callback函数

Generator 函数协程

传统的编程语言，早有异步编程的解决方案（其实是多任务的解决方案）。其中有一种叫做"协程"（coroutine），意思是多个线程互相协作，完成异步任务。

协程有点像函数，又有点像线程。它的运行流程大致如下。

- 第一步，协程A开始执行。
- 第二步，协程A执行到一半，进入暂停，执行权转移到协程B。
- 第三步，（一段时间后）协程B交还执行权。
- 第四步，协程A恢复执行。

上面流程的协程A，就是异步任务，因为它分成两段（或多段）执行。

```
function* asyncJob() {
  // ...其他代码
  var f = yield readFile(fileA);
  // ...其他代码
}
```

举例来说，读取文件的协程写法如上

上面代码的函数asyncJob是一个协程，它的奥妙就在其中的yield命令。它表示执行到此处，执行权将交给其他协程。也就是说，yield命令是异步两个阶段的分界线。

协程遇到yield命令就暂停，等到执行权返回，再从暂停的地方继续往后执行。它的最大优点，就是代码的写法非常像同步操作，如果去除yield

Generator 函数协程

Generator 函数可以暂停执行和恢复执行，这是它能封装异步任务的根本原因。除此之外，它还有两个特性，使它可以作为异步编程的完整解决方案：

```
function* gen(x) {
    var y = yield x + 2;
    return y;
}

var g = gen(1);
g.next() // { value: 3, done: false }
g.next(2) // { value: 2, done: true }
```

- **函数体内外的数据交换**

  next返回值的 value 属性，是 Generator 函数向外输出数据；next方法还可以接受参数，向 Generator 函数体内输入数据。

- **错误处理机制**

  throw方法抛出的错误，可以被函数体内的try...catch代码块捕获。这意味着，出错的代码与处理错误的代码，实现了时间和空间上的分离，这对于异步编程无疑是很重要的

```
function* gen(x){
  try {
    var y = yield x + 2;
  } catch (e){
    console.log(e);
  }
  return y;
}

var g = gen(1);
g.next();
g.throw('出错了');
// 出错了
```

Generator 函数协程的异步实现

```
var fetch = require('node-fetch');

function* gen() {
    var url = 'https://api.github.com/users/github';
    var result = yield fetch(url);
    console.log(result.bio);
}
var g = gen();
var result = g.next();

result.value.then(function (data) {
    return data.json();
}).then(function (data) {
    g.next(data);
});
```

左侧代码中，首先执行 Generator 函数，获取遍历器对象，然后使用next方法执行异步任务的第一阶段。

由于Fetch模块返回的是一个 Promise 对象，因此要用then方法调用下一个next方法。

Thunk 函数

Thunk 函数的背景简介

背景小故事:参数的求值策略

```
var x = 1;

function f(m) {
  return m * 2;
}

f(x + 5)
```

上面代码先定义函数f，然后向它传入表达式x + 5。请问，这个表达式应该何时求值?

```
f(x + 5)
// 传值调用时，等同于
f(6)
```

“传值调用”（call by value），即在进入函数体之前，就计算x + 5的值（等于 6），再将这个值传入函数f;//比如C语言,JavaScript等

```
f(x + 5)
// 传名调用时，等同于
(x + 5) * 2
```

“传名调用”（call by name），即直接将表达式x + 5传入函数体，只在用到它的时候求值

Thunk 函数的含义

编译器的“传名调用”实现，往往是将参数放到一个临时函数之中，再将这个临时函数传入函数体。**这个临时函数就叫做 Thunk 函数。**

```
function f(m) {
  return m * 2;
}

f(x + 5);
```

等价于

```
var thunk = function () {
  return x + 5;
};

function f(thunk) {
  return thunk() * 2;
}
```

这就是 Thunk 函数的定义，它是“传名调用”的一种实现策略，用来替换某个表达式。

JavaScript 语言的 Thunk 函数

```
// 正常版本的readFile（多参数版本）
fs.readFile(fileName, callback);

// Thunk版本的readFile（单参数版本）
var Thunk = function (fileName) {
  return function (callback) {
    return fs.readFile(fileName, callback);
  };
};

var readFileThunk = Thunk(fileName);
readFileThunk(callback);
```

在 JavaScript 语言中，Thunk 函数替换的不是表达式，而是多参数函数，将其替换成一个只接受回调函数作为参数的单参数函数。

上面代码中，fs模块的readFile方法是一个多参数函数，两个参数分别为文件名和回调函数。经过转换器处理，它变成了一个单参数函数，只接受回调函数作为参数。

这个单参数版本，就叫做 Thunk 函数。

JavaScript 语言的 Thunk 函数

```
// ES5版本
var Thunk = function(fn){
  return function (){
    var args = Array.prototype.slice.call(arguments);
    return function (callback){
      args.push(callback);
      return fn.apply(this, args);
    }
  };
};

// ES6版本
const Thunk = function(fn) {
  return function (...args) {
    return function (callback) {
      return fn.call(this, ...args, callback);
    }
  };
};
```

任何函数，只要参数有回调函数，就能写成 Thunk 函数的形式。左侧是一个简单的 Thunk 函数转换器(使用了柯里化方法)

```
var readFileThunk = Thunk(fs.readFile);
readFileThunk(fileA)(callback);
```

目前有一个Thunkify 模块 可以用来作为函数的转换用，源码与上述基本一致,内部做了些兼容处理

Thunk 函数结合Generator函数的流程管理操作

```
var fs = require('fs');
var thunkify = require('thunkify');
var readFileThunk = thunkify(fs.readFile);

var gen = function* () {
    var r1 = yield readFileThunk('/etc/fstab');
    console.log(r1.toString());
    var r2 = yield readFileThunk('/etc/shells');
    console.log(r2.toString());
};

var g = gen();

var r1 = g.next();
r1.value(function (err, data) {
    if (err) throw err;
    var r2 = g.next(data);
    r2.value(function (err, data) {
        if (err) throw err;
        g.next(data);
    });
});
```

手动版流程管理

```
function run(fn) {
    var gen = fn();

    function next(err, data) {
        var result = gen.next(data);
        if (result.done) return;
        result.value(next);
    }

    next();
}

var g = function* () {
    var f1 = yield readFileThunk('fileA');
    var f2 = yield readFileThunk('fileB');
    // ...
    var fn = yield readFileThunk('fileN');
};

run(g);
```

自动版流程管理

Promise对象

Promise对象的基本概念

Promise 是异步编程的一种解决方案，比传统的解决方案—回调函数和事件更合理和更强大;

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

Promise对象的基本用法

```
const promise = new Promise(function (resolve, reject) {
    // ... some code

    /* 异步操作结束 得到异步操作的结果*/
    if (当操作的结果是xx, 则返回操作成功的状态) {
        resolve(value);
    } else { //当操作的结果不是xx,则返回操作失败的状态
        reject(error);
    }
});

promise.then(function fn_resolve(value) {
    // success
},function fn_reject(error) {
    // failure
});
```

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。

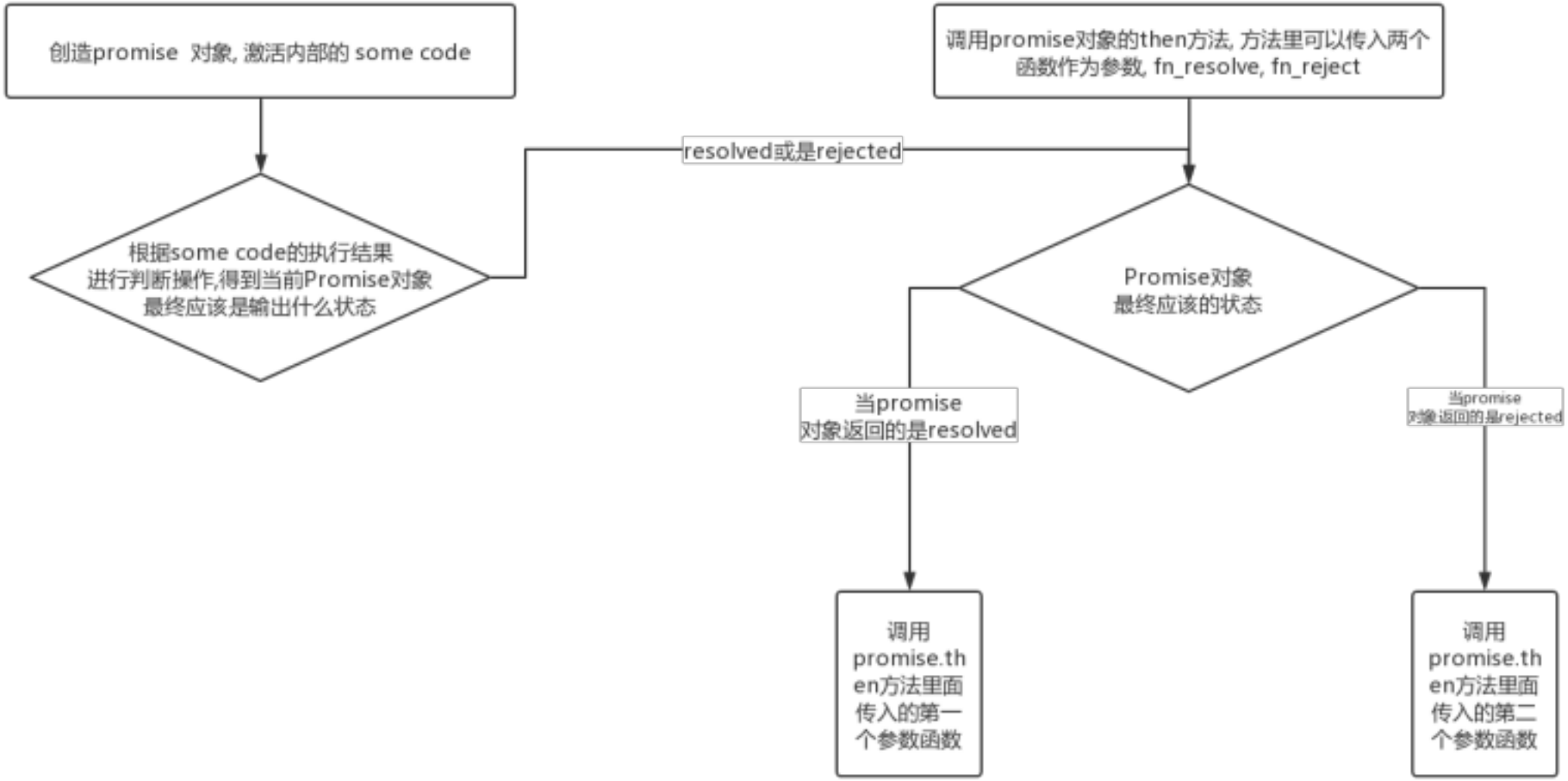
resolve函数的作用是:将Promise对象的状态从“未完成”变为“成功”（即从 pending 变为 resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；(就是传递给then方法，还可以传一些参数)

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。

then方法可以接受两个回调函数作为参数。**第一个回调函数是**Promise对象的状态变为resolved时调用，**第二个回调函数是**Promise对象的状态变为rejected时调用。

其中，第二个函数是可选的，不一定要提供。这两个函数都可以接受Promise对象传出的值作为参数。

Promise对象的基本用法的示意图



Promise对象的特点

（1）对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）。只有异步操作的结果(即在新建Promise对象时里面的some code的结果在通过判断之后执行的resolve或是reject函数)，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是Promise这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

（2）一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从pending变为fulfilled和从pending变为rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 resolved（已定型）

如果改变已经发生了，你再对Promise对象添加回调函数，也会立即得到这个结果。也就是说，你可以添加多个then,但是每个then都是要么resolved被激活或是rejected被激活

```
promise.then(function fn1_resolve(value) {
    // success
},function fn1_reject(error) {
    // failure
});
promise.then(function fn2_resolve(value) {
    // success
},function fn2_reject(error) {
    // failure
});
```

Promise对象的简易示例1

```
function timeout(ms) {
    return new Promise((resolve, reject) => {
        setTimeout(resolve, ms, 'done');
        //setTimeout在时间的参数后面还可以传入其他参数
        //这些其他参数会作为setTimeout里面的函数参数执行的参数
    });
}

timeout(100).then((value) => {
    console.log(value);//输出字符串 done
});
```

Promise对象的简易示例2

```
let promise = new Promise(function (resolve, reject) {
    console.log('Promise');
    resolve();
});

promise.then(function () {
    console.log('resolved.');
});

console.log('Hi!');

// Promise
// Hi!
// resolved
```

Promise对象的简易示例3

```
function loadImageAsync(url) {
    return new Promise(function (resolve, reject) {
        const image = new Image();

        image.onload = function () {
            resolve(image);
        };

        image.onerror = function () {
            reject(new Error('Could not load image at ' + url));
        };

        image.src = url;
    });
}
```

Promise对象的状态嵌套

```
34      const p1 = new Promise(function (resolve, reject) {
35          setTimeout(() => {
36              console.log("p1激活");
37              reject(new Error('fail'));
38          }, 3000)
39      })
40
41      const p2 = new Promise(function (resolve, reject) {
42          setTimeout(() => {
43              resolve(p1);
44          }, 1000)
45      })
46
47      p2
48          .then(result => {
49              console.log(result);
50          },err=>{
51              console.log(err);
52              console.log("p2激活");
53          })
```

```
p1激活                                    demo.html:36
Error: fail                               demo.html:51
    at demo.html:37
p2激活                                    demo.html:52
>
```

上面代码中，p1和p2都是 Promise 的实例，但是p2的resolve方法将p1作为参数，即一个异步操作的结果是返回另一个异步操作。

这时p1的状态就会传递给p2，也就是说，p1的状态决定了p2的状态。如果p1的状态是pending，那么p2的回调函数就会等待p1的状态改变；如果p1的状态已经是resolved或者rejected，那么p2的回调函数将会立刻执行。

Promise对象的状态嵌套的概念