Promise剖析

单开元

WHAT
WHY
HOW

WHAT

字面意思：允诺、许诺、给人以...的指望或希望

MDN：The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value

WHY

同步、异步

同步

同步任务指的是在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务。

同步模式编写简单。缺点是如果执行时间过长会堵塞后面代码的执行，导致用户体验较差。因此对于耗时操作，异步模式更佳。

异步

异步：异步任务指的是不进入主线程，而进入任务队列的任务。只有等主线程任务执行完毕，任务队列通知主线程，该任务才会排队进入主线程执行。

常见的异步模式：setTimeout、setInterval 、ajax

```
setTimeout(function() {
    console.log('asynchronousTaskA');
}, 0);
console.log('synchronizeTaskB');
//while(true)
```

异步任务会在当前脚本的所有同步任务执行完才会执行

WHY

回调函数: 简单理解为执行完回来调用的函数。

维基百科：In computer programming, a callback is a piece of executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at some convenient time.

回调函数使用场景：同步异步均可使用

WHY

```
function request(url, param, successFn, errorFn) {
    $.ajax({
        type: 'GET',
        url: url,
        param: param,
        success: successFn,
        error: errorFn
    });
}
request('testUrl', '', function(data) {
    console.log('请求成功数据:', data);
},function(error) {
    console.log('请求失败信息:', error);
});
```

WHY

异步回调便可以进行异步操作，为什么还要引入基于 异步的Promise?

WHY

```
function request(url, param, successFn, errorFn) {
    $.ajax({
        type: 'GET',
        url: url,
        param: param,
        success: successFn,
        error: errorFn
    });
}
request('testUrl', '', function(data) {
    console.log('请求成功数据:', data);
},function(error) {
    console.log('请求失败信息:', error);
});
```

```
function sendRequest(url, param) {
    return new Promise(function (resolve, reject) {
        request(url, param, resolve, reject);
    });
}

sendRequest('testUrl', '').then(function(data) {
    console.log('请求成功数据:', data);
}, function(error) {
    console.log('请求失败信息:', error);
});
```

略显复杂，需要先新建Promise，再定义回调

WHY

Promise的真正强大之处在于它的多重链式调用，可以避免层层嵌套回调。例如我们需要在第一次ajax请求之后，使用该次请求的结果再次请求呢?

WHY

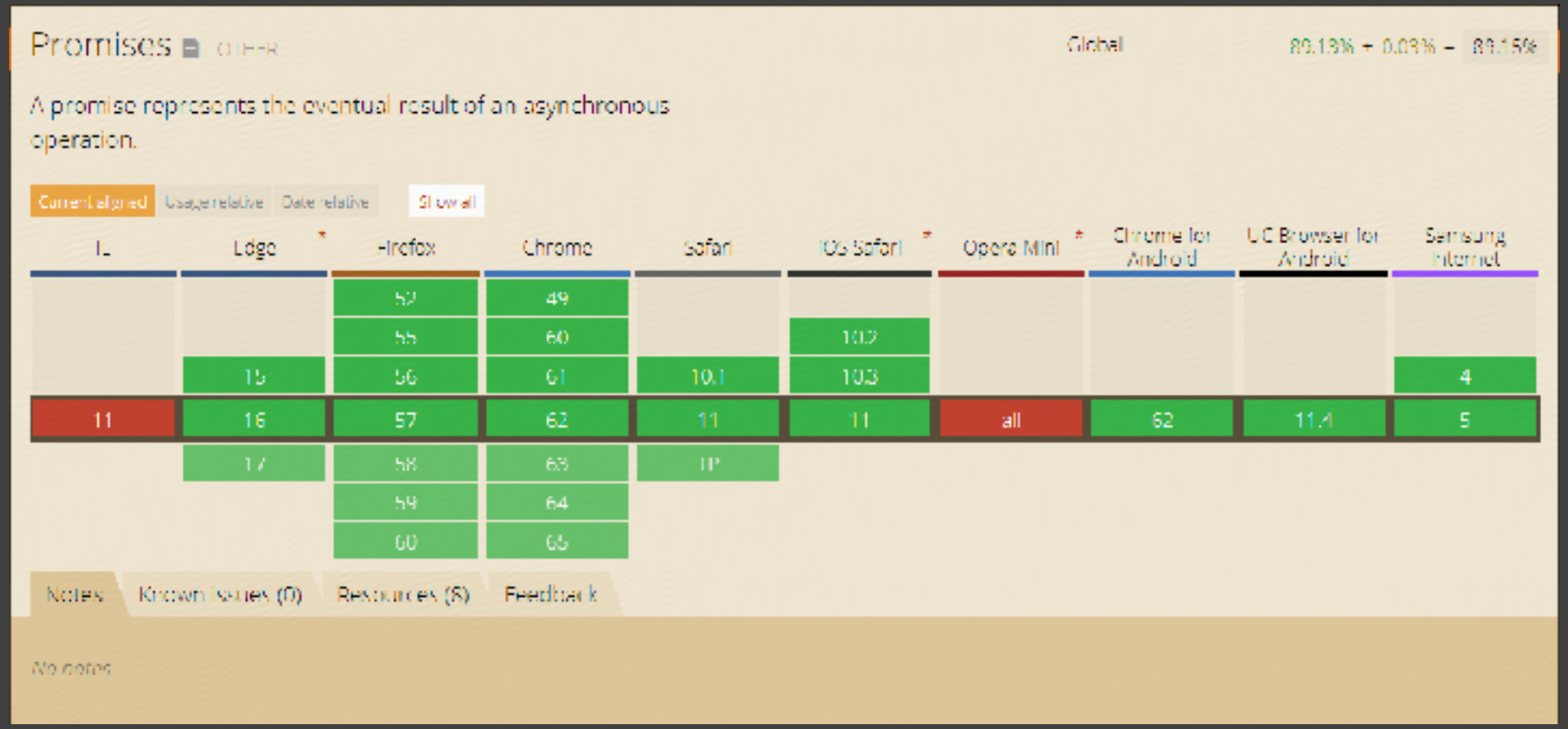
```
request('testUrl1', '', function(data1) {
    console.log('第一次请求成功的数据:', data1);
    request('testUrl2', data1, function (data2) {
        console.log('第二次请求成功的数据:', data2);
        request('testUrl3', data2, function (data3) {
            console.log('第三次的数据:', data3);
            //后续操作
        }, function(error3) {
            console.log('第三次请求失败信息:', error3);
        });
    }, function(error2) {
        console.log('第二次请求失败信息:', error2);
    });
}, function(error1) {
    console.log('第一次请求失败信息:', error1);
});
```

世界本没有回调，写的人多了，也就有了})})})})})。

WHY

```
sendRequest('testUrl1', '').then(function(data1) {
    console.log('第一次请求成功的数据:', data1);
    sendRequest('testUrl2', '');
}).then(function(data2) {
    console.log('第二次请求成功的数据:', data2);
    sendRequest('testUrl3', '');
}).then(function(data3) {
    console.log('第三次请求成功的数据:', data3);
}).catch(function(error) {
    console.log('sorry, 请求失败了, 这是失败信息:', error);
});
```

HOW

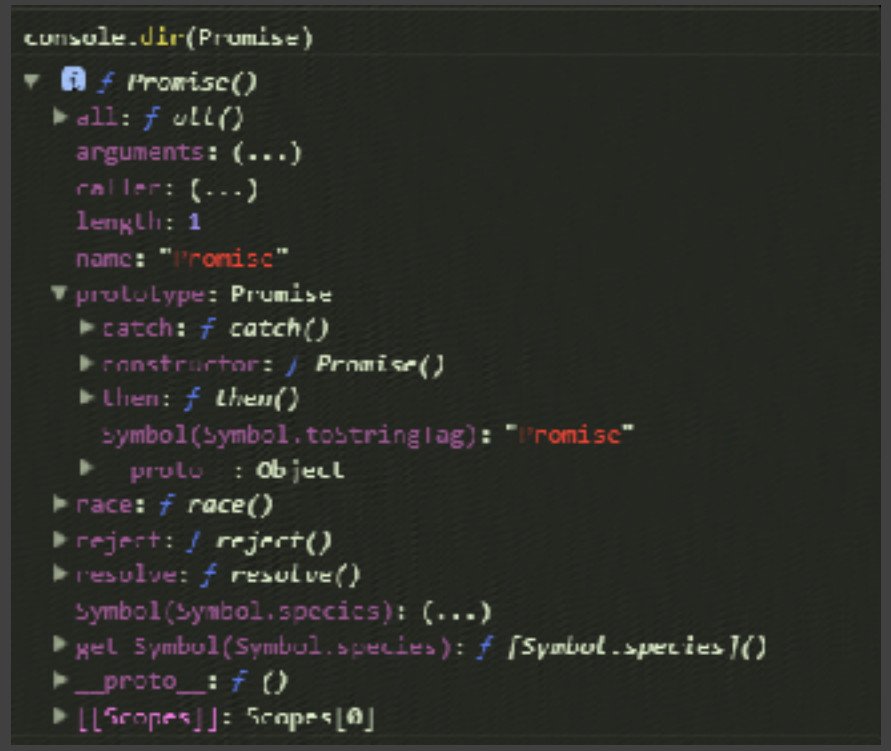

Promises
Global
A promise represents the eventual result of an asynchronous operation.
Current aligned
Slow all
IE
Edge
Firefox
Chrome
Safari
iOS Safari
Opera Mini
Chrome for Android
UC Browser for Android
Samsung Internet
11
16
57
62
11
11
all
62
11.4
5
Notes
Known issues (0)
Resources (8)
Feedback

HOW

1、通过Polyfill类库 如：es6-promise

2、通过扩展类库 如：bluebird

HOW



Promise是一个构造函数，自身拥有all、reject、resolve等方法，原型上有then、catch等方法。

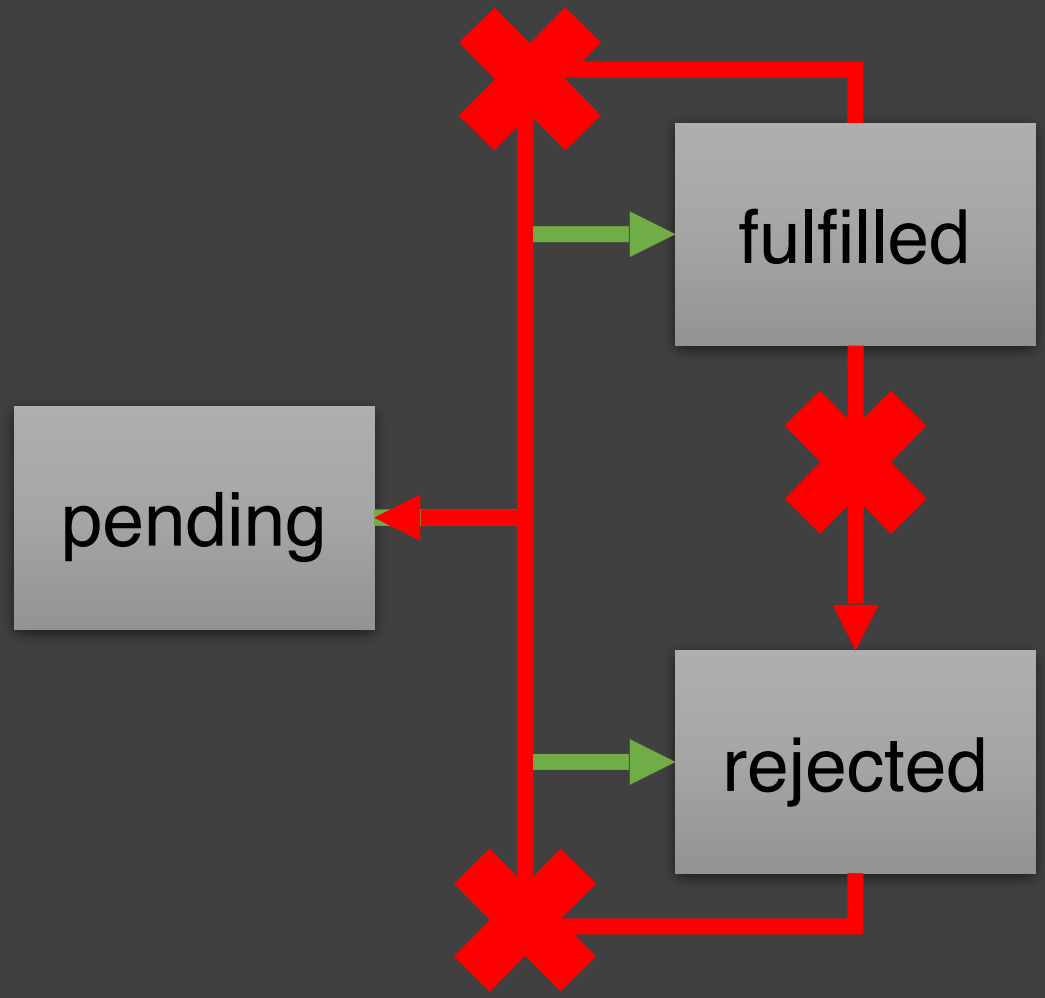
HOW

三种状态：

pending：初始状态

fulfilled：操作成功

rejected：操作失败



Promise实例

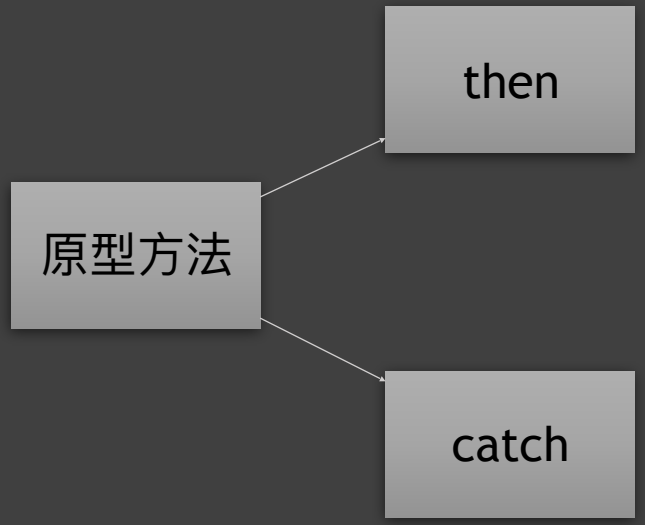
```
let promise = new Promise(function (resolve, reject) {
    if (/* 异步操作成功 */ ) {
        resolve(data);
    } else {
        /* 异步操作失败 */
        reject(error);
    }
});
```

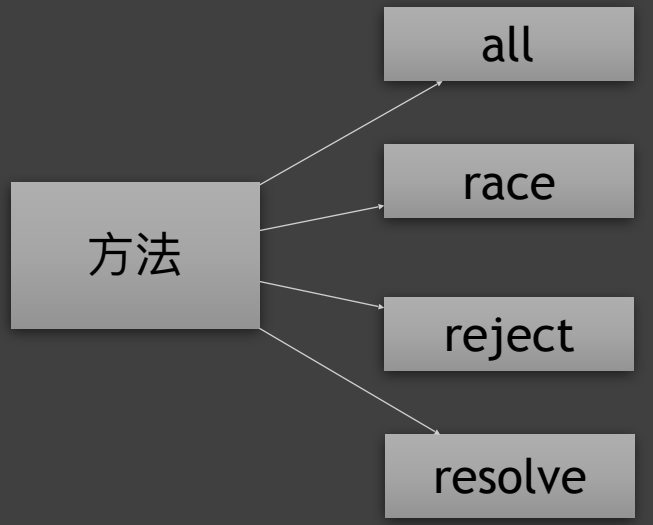
创建Promise实例的时候，需要传递一个函数作为参数，该函数提供两个参数作为回调函数。

resolve函数的作用：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：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。

HOW





THEN

```
promise.then(function(data) {
    // do something when success
}, function(error) {
    // do something when failure
});
```

Then方法有两个参数，分别为Promise从pending变为fulfilled和rejected时的回调函数。这两个回调函数，分别接收resolve和reject传出的值作为参数。

语法：Promise.prototype.then(onFulfilled, onRejected)

CATCH

Promise.prototype.catch(onRejected)

.catch(onRejected) === .then(undefined, onRejected)

```
promise.then(function(data) {
    console.log('success');
}).catch(function(error) {
    console.log('error', error);
});

/*******等同于*******/
promise.then(function(data) {
    console.log('success');
}).then(undefined, function(error) {
    console.log('error', error);
});
```

CATCH

```
sendRequest('testUrl').then(function(data1) {
    //do something
}).then(function (data2) {
    //do something
}).catch(function (error) {
    //处理前面二个Promise产生的错误
});
```

Promise对象的错误，会一直向后传递，直到被捕获。
then方法中的回调函数，如果抛出错误，则会被下一个catch捕获。
catch方法中也会抛出错误，则该方法后面的catch进行捕获。

ALL

语法：Promise.all(iterable)

```
let p1 = new Promise((resolve, reject) => {
    setTimeout(resolve, 1000, "one");
});
let p2 = new Promise((resolve, reject) => {
    setTimeout(reject, 2000, "two");
});
let p3 = new Promise((resolve, reject) => {
    reject("three");
});

Promise.all([p1, p2, p3]).then((value) => {
    console.log('resolve', value);
}, (error) => {
    console.log('reject', error);
});
```

RACE

语法：Promise.race(iterable)

```
let p3 = new Promise((resolve, reject) => {
    setTimeout(resolve, 500, "three");
});
let p4 = new Promise((resolve, reject) => {
    setTimeout(reject, 100, "four");
});

Promise.race([p3, p4]).then((value) => {
    console.log('resolve', value);
}, (error) => {
    console.log('reject', error);
});
```

RESOLVE

Promise.resolve(value);
Promise.resolve(promise);
Promise.resolve(thenable);

```
Promise.resolve('Success');

/*******等同于*******/
new Promise(function (resolve) {
    resolve('Success');
});
```

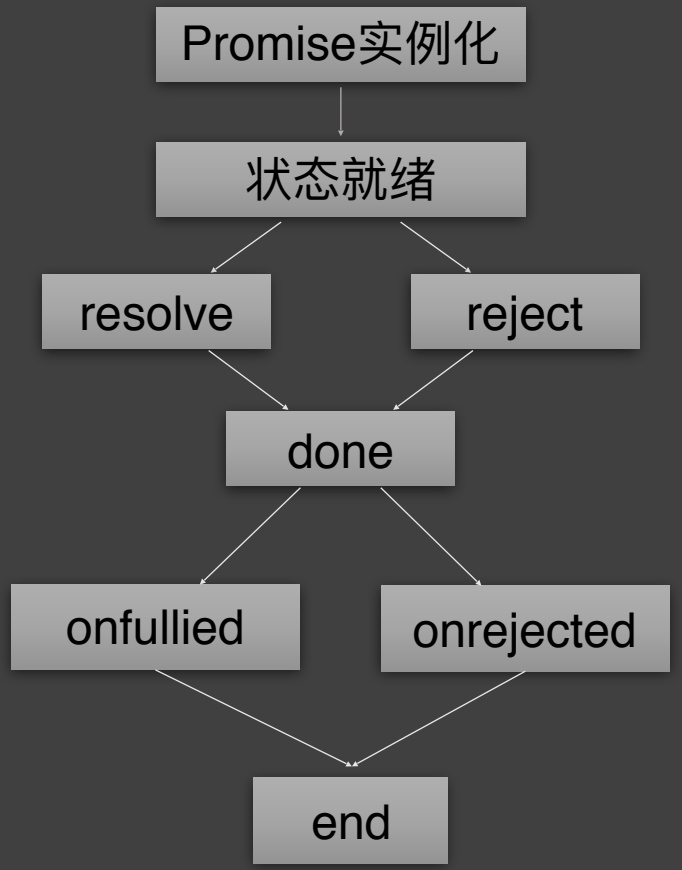
Promise.resolve相当于new Promise() 的快捷方式。
Resolve会让Promise对象立刻进入resolved状态，并将参数传递给后面then方法中的回调函数。
Promise.resolve()也可以把一个带有then方法的的对象立刻转为Promise对象。

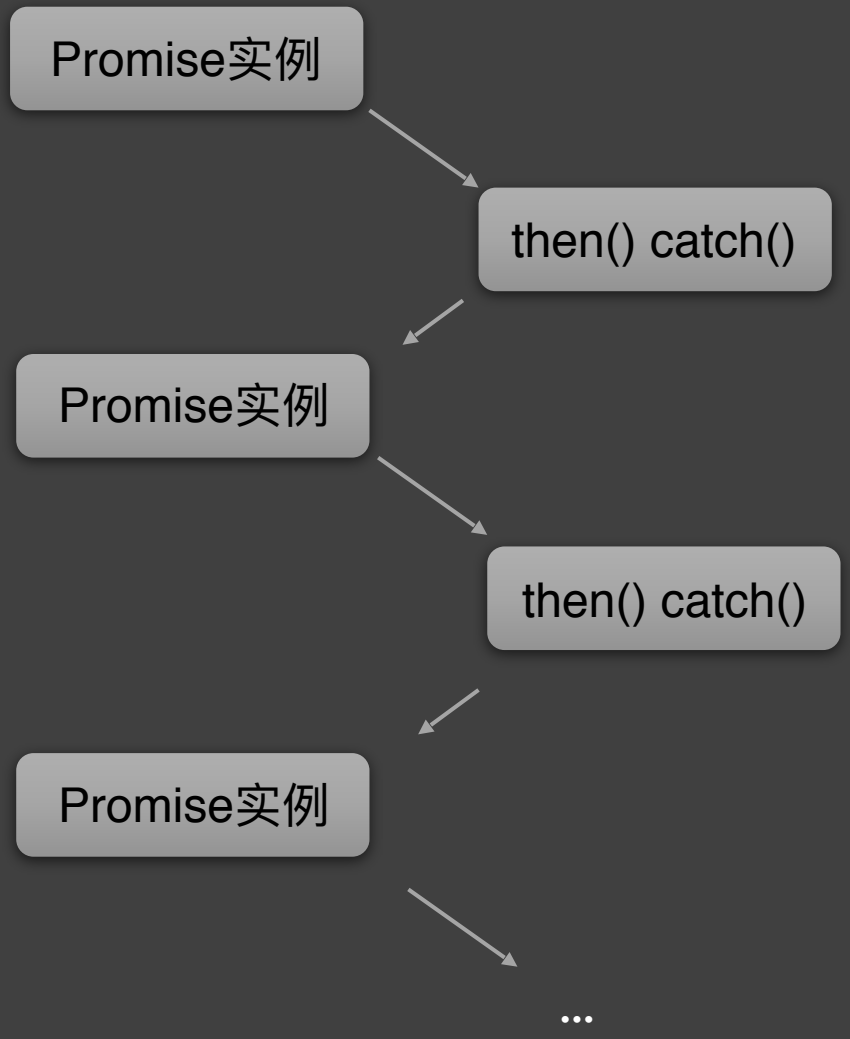
REJECT

语法：Promise.reject(reason)

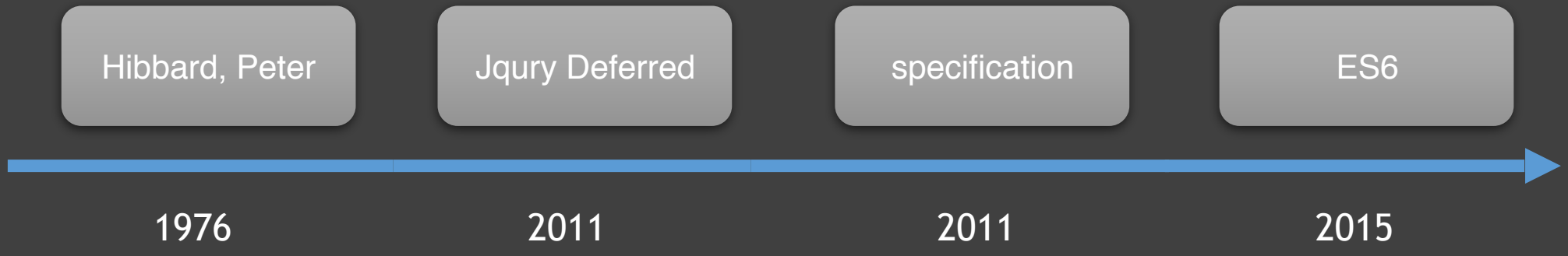
```
Promise.reject(new Error('error'));

/*******等同于*******/
new Promise(function (resolve, reject) {
    reject(new Error('error'));
});
```

Promise.reject会让Promise对象立即进入rejected状态，并将错误对象传递给then指定的onRejected回调函数


Promise实例化
状态就绪
resolve
reject
done
onfullied
onrejected
end


Promise实例
then() catch()
Promise实例
then() catch()
Promise实例
...

Promise不完整的历史



Promise/A+规范

Promise/A+规范是以Promises/A作为基础进行补充和修订，旨在提高promise实现之间的可互操作性。
Promises/A+ 对.then方法进行细致的补充，定义了细致的Promise Resolution Procedure流程，并且将.then方法作为promise的对象甄别方法。
ES6的Promise是Promise/A+规范的实现，，all和race属于ES6自己实现，规范中并没有。

常见问题

1、reject 和 catch 的区别

promise.then(onFulfilled, onRejected)

如果onFulfilled中发生异常，在onRejected中无法捕获该异常

promise.then(onFulfilled).catch(onRejected)

.then中的异常在.catch中可以捕获

常见问题

2、在Promise实例中的异步操作中抛错，不会被catch到

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        throw 'Uncaught Exception!';
    }, 1000);
});

promise.catch((error) => {
    console.log(error);
});
```

常见问题

3、promise状态变为resolve或reject，就凝固了，不会再改变

```
console.log(1);
new Promise(function (resolve, reject){
    reject();
    setTimeout(function (){
        resolve();
    }, 0);

    console.log(2);
}).then(function(){
    console.log(3);
}, function(){
    console.log(4);
});
console.log(5);
```

WHY

统一了同步和异步代码。

```
var price = () => {
    if (/*******价格存在*******/) {
        //渲染价格
    } else {
        getPrice().then((data) => {
            //渲染价格
        })
    }
}
```

```
var renderPrice = () => {
    if (/*******价格存在*******/) {
        Promise.resolve(data)
    } else {
        return getPrice()
    }
}
renderPrice.then( (data) => {
    //渲染价格
})
```

缺点

Promise也有一些缺点。
1、无法取消Promise，一旦新建它就会立即执行，无法中途取消。
2、如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。
3、当处于Pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）

谢谢