Promise

- Promise对象是对我们现在尚未得到的，但将来会得到的值的占位符。

三种状态

- pending 待定
- fulfilled 解决(有时候也会叫做 resolved)
- rejected 拒绝

Promise的特点

- 对象的状态不受外界的影响，只有异步结果可以决定当前是哪一种状态。任何其他的操作都无法改变。
- 一旦状态改变就不会再变。任何时候都会得到这个结果。
- promise对象只会有两种可能，从pending到resolved，从pending到rejected

Promise的缺点

- 无法取消promise，一旦创建就立即执行。如果不设置回调函数，promise内部的错误不会反映到外部

1. 在执行传入的回调函数时，会传入两个参数，resolve，reject这两个本身又是函数
2. 在回调函数中请求异步(只是请求，真正的异步代码的处理要去then中执行)

resolve 和 reject

```
new Promise((resolve, reject) => {
    console.log("in promise")
    resolve(console.log("in resolve"))
    console.log("in promise") //在new Promise中的代码都是同步执行的(resolve也是同步执
行的)
})
.then(res => {
    console.log(res) //只有这个处理程序then拿取参数的过程是异步执行的
})
```

- 这两个回调函数(resolve reject)直接执行的话，是同步的函数，但是这两个回调函数里面的参数(包括成功的值和错误的理由)的**抛出**就是异步的过程了。

Promise.resolve()

- `new Promise ((resolve, reject) => { resolve()}) === Promise.resolve()`
- 会实例化一个Promise并将状态转化为resolved
- 可以包装任何非期约值，包括错误对象。并将其转化为以解决的期约

```
new Promise((resolve, reject) => {
    resolve(new Error("xujie"))
```

```
}) //Promise <resolved> :Error:xujie
```

- 如果参数是一个期约的话， 那么外面的Promise.resolve()就相当于一个空包装
- 是一个幂等的方法

```
let p = Promise.resolve(3) // Promise <pending>
setTimeout(console.log, 0, p === Promise.resolve(p)) //true
setTimeout(console.log, 0, p === Promise.resolve(Promise.resolve(p))) //true
```

Promise.reject()

- 实例化一个Promise拒绝的期约， 并抛出一个错误。 (这个错误不能通过try catch捕获, 只能通过异步的处理程序catch或者then来捕获)(因为拒绝期约的错误并没有抛到同步代码的线程里面， 而是通过浏览器的异步消息队列来处理的)
- Promise.reject()没有实现和Promise.resolve()一样的幂等逻辑，如果一个期约被当做参数，那么这个被传入的期约就会变成参数。

```
let p = Promise.resolve(3)
Promise.reject(p) //这个Promise.resolve(3) 会成为错误的理由，而不是3
```

期约的实例方法

Promise.prototype.then()

- 其实then方法可以提供两个回调函数，第一个在上面的执行器promise resolve时执行回调，第二个在上面的执行器promise reject时执行回调(重要)
- p.then的返回值是一个**新的Promise实例**(p代指上面的执行器Promise)，then的返回值的值根据和p的执行状态的对应的then中的回调有关(也就是说如果p resolve，那么then的返回值就和then中的第一个回调有关， 如果p reject，那么then的返回值就和then中的第二个返回值有关)。

1. 如果then中第一个回调函数被定义，且有返回值那么就把这个返回值通过Promise.resolve()包装，然后then的返回值就等于这个被包装过的值
2. 如果then中的第一个回调函数被定义，但是没有返回值，那么then的返回值就是undefined经过Promise.resolve()包装
3. 如果then的中的第一个回调函数没有被定义， 那么then的返回值就是上面的p的解决之后的值。

实例中返回错误

```
let p = new Promise((resolve, reject) => {
    resolve("resolve p!")
})
let a = p.then(() => {
   return Error("xujie")
}) //返回的是错误对象的话， 不会爆出红色的异常，会把Error实例经过Promise.resolve()包装
```

```
当做then处理程序的返回值

let p = new Promise((resolve, reject) => {
    resolve("resolve p!")
})
let a = p.then(() => {
   throw("xujie")
})//如果是throw一个错误的话， 会爆出红色的异常，并且会把Promise.reject("xujie")经过
Promise.resolve()包装之后当做then处理器的返回值
```

Promise.prototype.catch()

- 其实就是个语法糖，`p.catch(() => {}) === p.then(null, () => {})`
- catch也有返回值，和上面介绍的行为一样。

Promise.prototype.finally()

- 无论p的状态如何都会执行,所以finally不接受任何参数。finally的返回值和状态无关，**大多数情况下**表现为父期约的传递。
- finally的返回值不和上面的执行器一样的场景是，finally处理程序中返回的是一个待定的期约，或者显示的返回一个拒绝的期约。
- 主要是为了解决冗余代码

非重入特性

- 在一个解决期约上调用then处理程序，会把then中的回调函数推荐消息队列。
- 因此then中的回调函数才是真正异步的函数。

拒绝期约和拒绝错误处理

- throw一个错误对象和在reject中返回一个Error对象，都会返回一个内容为错误对象的状态为reject的Promise。
- 期约可以使用任何理由拒绝，但是最好使用一个Error对象，因为Error对象会包含一些用于调试的信息，
- 这些错误都是通过异步抛出且未处理的，异步的错误只能通过异步的处理程序来捕获(注意只有经历了then等处理程序的错误才是异步的)

```
new Promise((resolve,reject) => {
     try{
         throw(Error("xujie")) //此时的throw的错误还没有经过then的异步调用，所以可以通
过try同步捕获
     }catch(e) {
         console.log(e);
     }
     resolve()
 })
```

期约的连锁和期约合成

期约连锁

- 可以使用连缀方法的调用的形式
- 实现的原理主要是then，catch，finally处理程序的返回值都是一个新的Promise。
- 这样的话每个后面的异步任务都会等待前面的异步任务执行完之后才被调用。简洁的将异步任务串行化，解决了回调依赖的难题。

期约合成

Promise.all()

Promise.race()

终止Promise(终止不是中断，Promise一旦被创建就没办法被中断)

终止Promise执行器函数

终止Promise的链式调用

```
new Promise((resolve,reject) => {
            setTimeout(() => {
                resolve()
            })
        }).then(() => {
            console.log("第一步的异步请求")

            return new Promise((resolve,reject) => {
                setTimeout(() => {
                    resolve()
                })
            })
        }).then(() => {
            console.log('第二次异步请求')

            return new Promise((resolve,reject) => {
                resolve()
            })
        }).then(() => {
            console.log("第三次异步请求");
        })
```

promise的链式调用和简写

```
new Promise((resolve, reject) => {
        setTimeout(() => {
          resolve('aaa');
        });
      }).then((res) => {
          console.log(res);
```

```
    return res + 111;
  }).then((res) => {
    console.log(res);

    return res + 222
  }).then((res) => {
    console.log(res);
  });
```

- 可以通过这种方式来进行链式调用，对于同一个数据进行多次处理 链式调用的基础是then和catch，finally的返回值都是一个Promise
- Promise.resolve()和Promise.reject()是Promise的一个简便的写法
- return res + 111,其实是Promise API内置的Promise.resolve()更加简便写法。其实这个语句在内部调用了一次 new Promise()

Promise.all()方法

```
Promise.all([
    new Promise(() => {
        ...
    }),
    new Promise(() => {
        ...
    })
]).then(results => {
    result[0] //里面是第一次执行的结果
    result[1] //里面是第二次执行的结果
})
```