Promise基础

三个状态

已兑现（fulfilled）：意味着操作成功完成；执行了resolve时，处于该状态

已拒绝（rejected）：意味着操作失败； 执行了reject时，处于该状态

一旦状态被确定下来，Promise的状态会被锁死，该Promise的状态是不可更改的

resolve

传入一个普通的值或者对象，那么这个值会作为then回调的参数

传入的是另外一个Promise，那么这个新Promise会决定原Promise的状态

```
const obj = {
 then(resolve, reject) {
  resolve('obj对象且有then方法,根据then方法结果来确定状态');
 },
};
new Promise((resolve, reject) => {
 resolve(obj);
}).then(
 (res) => {
  //obj对象且有then方法,根据then方法结果来确定状态
  console.log('res:', res);
 },
 (err) => {
  console.log('err:', err);
 }
);
```

传入的是一个对象，并且这个对象有实现then方法，那么会执行该then方法，并且根据 then方法的结果来决定Promise的状态

静态方法（类方法）

resolve

已经有一个现成的内容了，希望将其转成Promise来使用，这个时候我们可以使用 Promise.resolve 方法来完成

- 情况一：参数是一个普通的值或者对象
- 情况二：参数本身是Promise
- 情况三：参数是一个thenable

reject

Promise.reject传入的参数无论是什么形态，都会直接作为reject状态的参数传递到catch的。

all

将多个Promise包裹在一起形成一个新的Promise；新的Promise状态由包裹的所有Promise共同决定

- 所有的Promise状态变成fulfilled状态时，新的Promise状态为fulfilled，并且会将所有Promise的返回值组成一个数组
- 有一个Promise状态为reject时，新的Promise状态为reject，并且会将第一个reject的返回值作为参数

数据顺序由传入all方法的Promise顺序为准，而非异步调用得出结果速度决定

```
const p1 = new Promise((resolve, reject) =>
resolve('111'));
const p2 = new Promise((resolve, reject) =>
setTimeout(() => resolve('222'), 1500));
const p3 = new Promise((resolve, reject) =>
resolve('333'));

//按照传入的顺序进行输出
//普通对象会被转为Promise Promise.resolve('aaa')
Promise.all([p1, p2, p3, 'aaa']).then(
 // [ '111', '222', '333', 'aaa' ]
 (res) => console.log('res:',res),
 (err) => console.log(err)
);

//如果有一个reject 会停止且返回失败的那个promise
const p4 = new Promise((resolve, reject) =>
resolve('111'));
const p5 = new Promise((resolve, reject) =>
reject('error,p5'));
const p6 = new Promise((resolve, reject) =>
resolve('333'));
Promise.all([p4, p5, p6, 'aaa']).then(
 (res) => console.log(res),
 //error， p5
 (err) => console.log(err)
);
```

allSettled

- 该方法会在所有的Promise都有结果（settled），**无论是fulfilled，还是reject时，才会有最终的状态**
- 并且这个Promise的结果**一定是fulfilled**的；

与all方法的返回结果不同，allSettled的结果在一个数组的基础上，继续存放着每一个Promise的结果，并且是对应每一个对象的
这个对象中兑现状态的结果为status状态，以及对应的value值，拒绝状态的结果**为status状态，以及对应的reason值(错误理由)**，在此基础上我们就能够通过判断status状态进行针对性处理我们所需要的部分

rece

有一个Promise有了结果，我们就希望决定最终新Promise的状态，那么可以使用race方法

Promise.race() 非常适合用于实现超时机制。可以将一个异步操作的 Promise 和一个超时的 Promise 放在一起，如果原始操作在超时前完成，则继续处理结果；如果超时 Promise 先完成，则可以实施超时逻辑，如取消操作或返回超时错误

一旦其中一个Promise有了结果就会立刻返回，不管该结果是Promise的哪个状态

rece方法一旦报错就整个运行都结束了，如果我希望能够返回兑现的结果，只是为了筛选其中最快的部分，我又应该怎么做

any

ES12中新增的方法
any方法会等到一个fulfilled状态，才会决定新Promise的状态
如果所有的Promise都是reject的，那么也会等到所有的Promise都变成rejected状态

rece追求第一个结果
any方法追求第一个好的结果
只要追求到所需结果，就立刻返回内容。如果没有所需结果，比如any方法内的所有的Promise都是reject的，那么会报一个AggregateError的错误

```
const p1 = new Promise((resolve,reject)=>{
  setTimeout(()=>{
   reject("p1 reject error")
  },2000)
 })

 const p2 = new Promise((resolve,reject)=>{
  setTimeout(()=>{
   reject("p2 resolve")
  },1000)
 })

 const p3 = new Promise((resolve,reject)=>{
  setTimeout(()=>{
   reject("p3 resolve")
  },3000)
 })

//  //类方法，reve方法
//  Promise.race([p1,p2,p3]).then(res => {
//   console.log("res：",res)
//  }).catch(err =>{
//   console.log("err：",err)
//  })

Promise.any([p1,p2,p3]).then(res => {
  console.log("res：",res)
 }).catch(err =>{
// err： [AggregateError: All promises were rejected] {
 [errors]: [ 'p1 reject error', 'p2 resolve', 'p3 resolve' ]
}
  console.log("err：",err)
 })
```

实例方法

then

多次调用

- 每次调用都可以传入对应的fulfilled回调
- 当Promise的状态变成fulfilled的时候，**这些回调函数都会被执行**

```
const promise = new Promise((resolve, reject) => {
 resolve('完成');
});
promise.then((res) => {
 console.log(res);
});
promise.then((res) => {
 console.log(res);
});
promise.then((res) => {
 console.log(res);
});
promise.then((res) => {
 console.log(res);
});
// 打印四个完成
```

返回值

返回的是一个普通值(数值/字符串/普通对象/undefined), 那么这个普通的值被作为一个新的Promise的resolve值

返回值是一个Promise，所以可以进行链式调用

返回一个Promise

```
promise
 .then((res) => {
  return {
   then: function (resolve, reject) {
    resolve(222222);
   },
  };
 })
 .then((res) => {
  //res: 222222
  console.log('res:', res);
 });
```

返回一个thenable值

不返回值默认为undefined

当then方法抛出一个异常时，那么它处于reject状态

```
const promise = new Promise((resolve, reject) => {
 reject("失败");
});
promise
 .catch((err) => {
  //失败
  console.log(err);
 })
 .catch((err) => {
  console.log("err", err);
 })
 .then((res) => {
  //res undefined
  console.log("res", res);
 });
```

catch

catch传入的回调在执行完后，默认状态依然会是fulfilled的

return也分三种情况，跟then一样

```
// catch分情况
const promise = new Promise((resolve, reject) => {
//  reject('失败');
//  resolve("成功");
});

//首先打印promise的异常，如果promise没有异常，会打印then方法执行抛出的异常，以此类推
promise
 .then(() => {
  throw new Error("then里的异常");
 })
 .catch((err) => {

  //Error: then里的异常
  console.log(err);
 });
```

finally

ES9（ES2018）中新增的一个特性：表示无论Promise对象无论变成fulfilled还是reject状态，最终都会被执行的代码