Promise

1. 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。它由社区最早提出和实现，ES6 将其写进了语言标准，统一了用法，原生提供了`Promise`对象

2. Promise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，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。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

3.

Promise 承诺 构造函数

Promise 原型上的方法 then可以包含两个参数，失败和成功，(也可以使用一 个,另一个catch)，catch失败 constructor 构造函数本身

resolve 决定 成功回调函数 reject 拒绝 失败回调函数 all 所以异步完成

race 某一个异步完成,立即执行回调函数

Promise作为对象上的方法： all ， race，resolve，reject

pending(进行中状态) ===>> fulfilled(成功状态) / rejected(失败状态)

注意: then(fn1[,fn2]); 可以写俩个函数 success, error, fn1出错，会JS卡死

then 函数接收俩个参数，都是函数

then().catch(); 也可以分开写，利用catch这样then出错会执行到catch，JS不会卡死

4.异步处理:

1) 事件处理

2)回调函数（回调地狱）

3) setInterval/setTimeout

4) Promise

."异步模式"编程的几种方法:

（1）**回调函数**:优点是简单、容易理解和部署，缺点是不利于代码的阅读和维护，各个部分之间高度耦合（Coupling），使得程序结构混乱、流程难以追踪（尤其是回调函数嵌套的情况），而且每个任务只能指定一个回调函数。

（2）**采用事件驱动模式（事件监听）**：优点是比较容易理解，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以"去耦合"（Decoupling），有利于实现模块化。缺点是整个程序都要变成事件驱动型， 运行流程会变得很不清晰。

（3）**观察者模式（发布\订阅模式）**：这种方法的性质与"事件监听"类似，但是明显优于后者。因为我们可以通过查看"消息中心"，了解存在多少信号、每个信号有多少订阅者，从而监控程序的运行。

（4）**Promise对象解决**

Promise解决了什么问题？ ----->> ES5的回调地狱

```
function fn(callback) {
    let i = 10;
```

```
        callback && callback(i);
    }

    fn(function(val){
        console.log(val);
        fn1()
    });
```

```
// ==>> ES5的回调地狱(Ajax常出现这种情况)
let fn = (callback) => {
    let i = 10;
    callback && callback(i);
};

let fn1 = (callback) => {
  let name = '常连海';
  callback && callback(name);
};

let fn2 = callback => {
    let age = 25;
    callback && callback(age);
};

// fn ==>> fn1 ==>> fn2
fn(function (val) {
   console.log(val);
   fn1(function (val1) {
       console.log(val1);
       fn2(function (val2) {
           console.log(val2);
       });
   });
});
```

3

5.你要知道的:

console.dir(promise1.__proto__); // ===>> 类原型上的方法(catch, then, constructor)

console.dir(Promise); //构造函数 ,使用new // ===>> 类作为对象上的方法(all, prototype(原型), race,reject(失败), resolve(成功), __proto__(原型链))

console.log(promise1.__proto__ === Promise.prototype);

console.log(promise1.__proto__.__proto__ === Promise.prototype.__proto__ === Object.prototype

***then()返回Promise实例对象，可以链式写法， 可以写多个then函数，依次执行

6.图解说明:

1.Promise 的含义

2.基本用法

3.Promise.prototype.then()

4.Promise.prototype.catch()

5.Promise.all()

6.Promise.race()

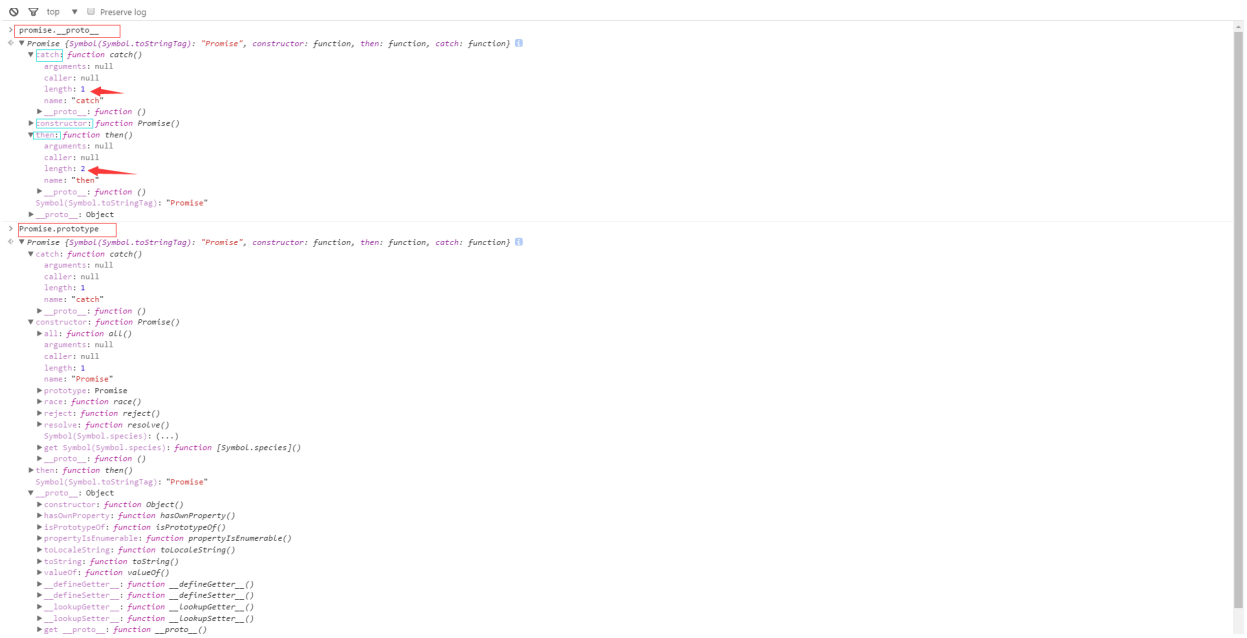
7.Promise.resolve()

8.Promise.reject()

9.两个有用的附加方法

10.应用

11.Promise.try()

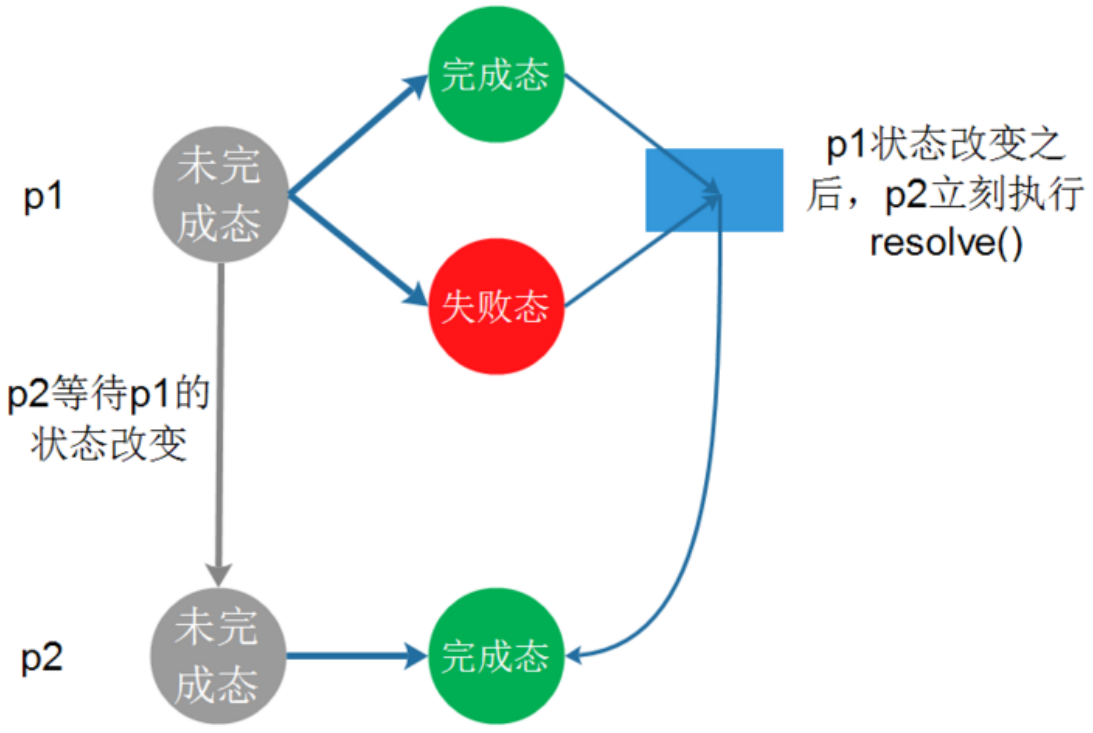


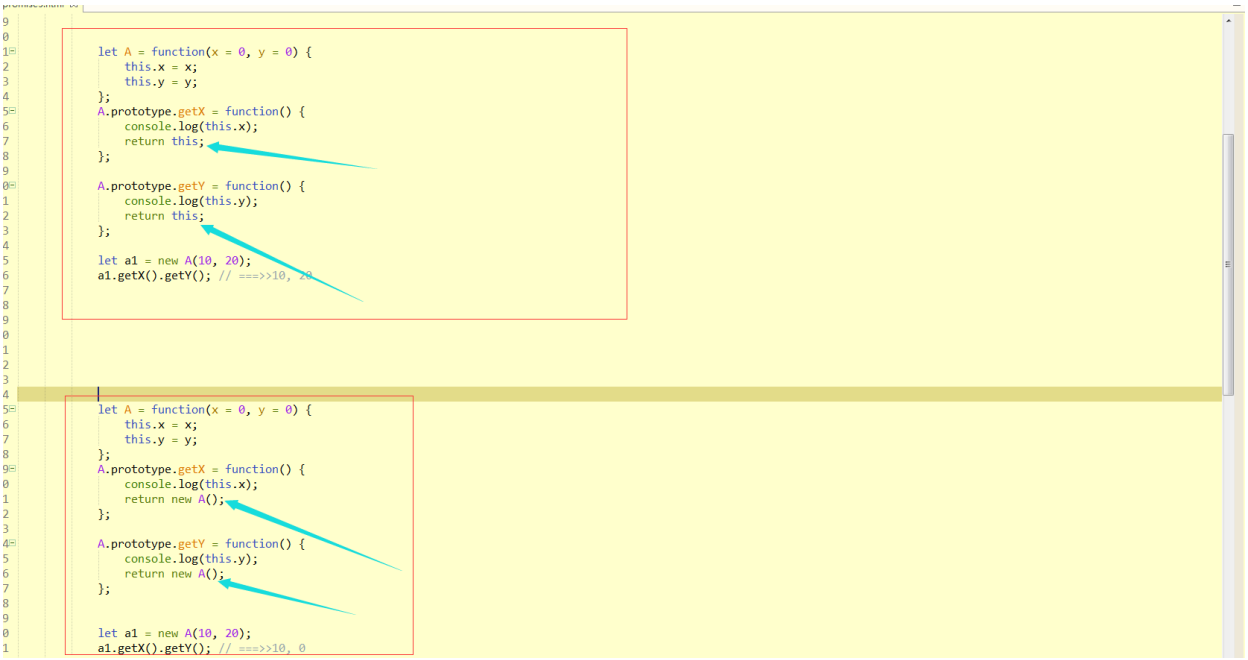
Javascript 中函数的 length 属性

每个函数都有一个 length属性 （ 函数名.length ） ， 表示期望接收的函数的个数（而不是实际接收的参数个数）

它与arguments不同， arguments.length 是表示函数实际接收的参数个数。

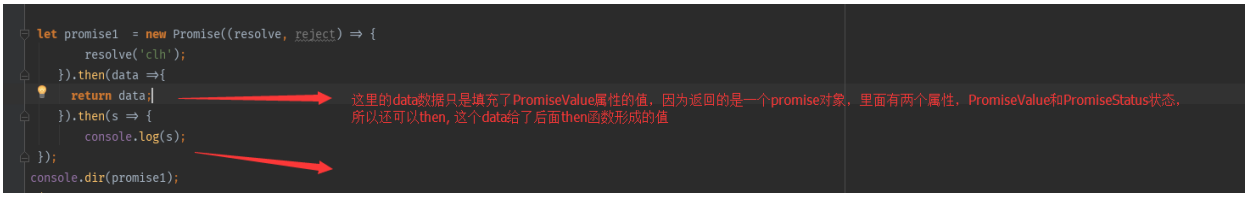
```
<html>
  <head>
      <script type="text/javascript">
          var add = function(num, num2, num3)
          {
              alert(num+20);
          }
          alert(add.length);
      </script>
  </head>
  <body>
  </body>
</html>
```





```
let A = function(x = 0, y = 0) {
    this.x = x;
    this.y = y;
};
A.prototype.getX = function() {
    console.log(this.x);
    return this;
};

A.prototype.getY = function() {
    console.log(this.y);
    return this;
};

let a1 = new A(10, 20);
a1.getX().getY(); // ===>>10, 20
```

```
let A = function(x = 0, y = 0) {
    this.x = x;
    this.y = y;
};
A.prototype.getX = function() {
    console.log(this.x);
    return new A();
};

A.prototype.getY = function() {
    console.log(this.y);
    return new A();
};

let a1 = new A(10, 20);
a1.getX().getY(); // ===>>10, 0
```



```
let promise1  = new Promise((resolve, reject) => {
      resolve('clh');
  }).then(data =>{
    return data;
  }).then(s => {
      console.log(s);
});
console.dir(promise1);
```

这里的data数据只是填充了PromiseValue属性的值，因为返回的是一个promise对象，里面有两个属性，PromiseValue和PromiseStatus状态，所以还可以then，这个data给了后面then函数形成的值

```
> p
< ▼Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: undefined}
    ▶__proto__: Promise
     [[PromiseStatus]]: "resolved"
     [[PromiseValue]]: undefined
>
```

7.实例

1. new Promise(); 接收一个参数是函数，里面有俩个参数是resolve成功回到和reject失败回调

```
let fn = function() {
        const promise1 = new Promise((resolve, reject)=>{
            window.setTimeout(function() {
                console.log('ok');
```

```
                        resolve('随便什么数据');
                },2000);
        });

        return promise1;
};

//===>>合起来写
fn().then((data) => {
        console.log(clh);
        console.log(data);
}, (error=> {
   console.log(error);
});

// ===>> 分开写
fn().then((data) => {
        console.log(clh);
        console.log(data);
}).catch((error) => {
        console.log(error);
});
```

2. 多个then链式写法

```
 // => https://www.cnblogs.com/lvdabao/p/es6-promise-1.html
// => https://www.cnblogs.com/lvdabao/p/jquery-deferred.html
// => http://es6.ruanyifeng.com/#docs/promise#Promise-race
let getData = function(url) {
         let flag = false;
         let data = {};
```

```
        $.ajax({
                        url: url,
                        type: 'get',
                        dataType: 'json',
                         success: function(data) {
                                if (data.code === 200)
{
                                        flag = true;
                                }
                         }
        });
        flag ? data: flag;
};



let fn1 = function(data) {
        let promise1 = new Promise((resolve, reject) =>
{
                                $.ajax({
                                url: './data/a.json',
                                type: 'get',
                                dataType: 'json',
                                success: function(data)
{
                                        if (data.code
=== 200) {

resolve(data);
```

```
                                        }
                                    }
                                });
        });
        return promise1;
};

let fn2 = function() {
        let     promise2 = new Promise((resolve,
reject) => {
                                $.ajax({
                                        url: './data/b.json',
                                        type: 'get',
                                        dataType: 'json',
                                         success:
function(data) {
                                                if (data.code
=== 200) {

resolve(data);
                                                }
                                        }
                                });
        });

        return promise2;
};

let fn3 = function() {
        let promise3 = new Promise((resolve, reject) =>
{
```

```
                                        $.ajax({
                                        url: './data/c.txt',
                                        type: 'get',
                                        success: function(data)
{
                                                resolve(data);
                                        },
                                         error: function() {

                                         }
                                });
        });
        return promise3;
}


fn1().then((data)=> {
        console.log(data);
        return fn2();

}).catch((error)=> {

}).then((data)=> {
        console.log(data);
        return fn3();
}).catch((error)=> {

}).then((data)=> {
        console.log(data);
});
```

3. all和race用法

all: `Promise.all`方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。

```
const p = Promise.all([p1, p2, p3]);
```

`p`的状态由`p1`、`p2`、`p3`决定，分成两种情况。

（1）只有`p1`、`p2`、`p3`的状态都变成`fulfilled`，`p`的状态才会变成`fulfilled`，此时`p1`、`p2`、`p3`的返回值组成一个数组，传递给`p`的回调函数。

（2）只要`p1`、`p2`、`p3`之中有一个被`rejected`，`p`的状态就变成`rejected`，此时第一个被`reject`的实例的返回值，会传递给`p`的回调函数。

all方法的效果实际上是「谁跑的慢，以谁为准执行回调」，那么相对的就有另一个方法「谁跑的快，以谁为准执行回调」，这就是race方法，这个词本来就是赛跑的意思。race的用法与all一样，我们把上面runAsync1的延时改为1秒来看一下：

```
    Promise.all([runAsync1(), runAsync2(), runAsync3()]).then(function(results){
        console.log(results);
}).catch(error => {
        console.log(error);
});


 //  ===>> Promise.all([]).then(fn).catch(fn);
       Promise.all([fn1(), fn2(), fn3()]).then((data) =>{
                    console.log(data);
```

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

race:`Promise.race`方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例

```
const p = Promise.race([p1, p2, p3]);
```

上面代码中，只要`p1`、`p2`、`p3`之中有一个实例率先改变状态，`p`的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给`p`的回调函数。

`romise.race`方法的参数与`Promise.all`方法一样，如果不是 Promise 实例，就会先调用下面讲到的`Promise.resolve`方法，将参数转为 Promise 实例，再进一步处理

1) Promise的all方法提供了并行执行异步操作的能力，并且在所有异步操作执行完后才执行回调。我们仍旧使用上面定义好的runAsync1、runAsync2、runAsync3这三个函数

2) 用Promise.all来执行，all接收一个数组参数，里面的值最终都算返回Promise对象。这样，三个异步操作的并行执行的，等到它们都执行完后才会进到then里面。那么，三个异步操作返回的数据哪里去了呢？都在then里面呢，all会把所有异步操作的结果放进一个数组中传给then，就是上面的results。所以上面代码的输出结果就是：

```
// ===>> Promise.race([]).then(fn).catch(fn);
        Promise.race([fn1(), fn2(), fn3()]).then((data) =>
{
```

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

```
Promise.race([runAsync1(), runAsync2(),
runAsync3()]).then(function(results){
    console.log(results);
}).catceh(error =>(){
    console.log(error);
});
```

4.Promise.prototype.then();

1) Promise 实例具有`then`方法，也就是说，`then`方法是定义在原型对象`Promise.prototype`上的。它的作用是为 Promise 实例添加状态改变时的回调函数。前面说　过，`then`方法的第一个参数是`resolved`状态的回调函数，第二个参数（可选）是`rejected`状态的回调函数。

2) 链式写法核心 --->> `then`方法返回的是一个新的`Promise`实例（注意，不是原来那个`Promise`实例）。因此可以采用链式写法，即`then`方法后面再调用另一个`then`方　法。

5.Promise.prototype.catch()

`Promise.prototype.catch`方法是`.then(null, rejection)`的别名，用于指定发生错误时的回调函数。

案例:

```
let promise = new Promise(resolve => {
        resolve('promise');
        console.log(33333)
}).then(data => {
        console.log(data);
});
```

3333, promise　　因为resolve是异步的,

因为立即 resolved 的 Promise 是在本轮事件循环的末尾执行，总是晚于本轮循环的同步任务

细节知识点:

1) Promise 新建后就会立即执行。

jQuery里面的Promise