06-第六章 Reflect 映射 异步的 Promise

ECMAscript 6

一、Reflect对象的设计目的

> (1)将`Object`对象的一些明显属于语言内部的方法（比如Object.defineProperty），放到Reflect对象上。现阶段，某些方法同时在Object和Reflect对象上部署，未来的新方法将只部署在`Reflect`对象上。也就是说，从Reflect对象上可以拿到语言内部的方法。

```
var obj = {};
            Reflect.defineProperty(obj,'index',{ value:88})
            console.log( obj);//{index: 88}
```

> (2)改某些Object方法的返回结果，让其变得更合理。比如，Object.defineProperty(obj, name, desc)在无法定义属性时，会抛出一个错误，而Reflect.defineProperty(obj, name, desc)则会返回false。

```
        var obj = {};

        if( Reflect.defineProperty(obj,'index',{ value:88}) ){
            console.log( 'ok' );//ok
        }
        console.log( obj);//{index: 88}
```

(3)让Object操作都变成函数行为。某些Object操作是命令式，比如name in obj和delete obj[name]，而Reflect.has(obj, name)和Reflect.deleteProperty(obj, name)让它们变成了`函数行为`

```
1.
2.
3.              var obj = {age:888};
4.              console.log( Reflect.has(obj,'age') );//true
5.
```

(4) `Reflect`对象的方法与`Proxy`对象的方法`一一对应`，只要是Proxy对象的方法，就能在Reflect对象上找到对应的方法

```
1. var obj = {};
2.          var prox = new Proxy({},{
3.              set(target,prop,self){
4.                  Reflect.set(target,prop,self);
5.              }
6.          })
7.          prox.index = 88;
8.          console.log(prox);//{index: 88}
```

二、Reflect的静态方法

（1）`Reflect.set(`target, name, value, receiver`)`

Reflect.set方法设置target对象的name属性等于value,如果name属性设置了赋值函数，则赋值函数的this绑定`receiver`。

```
var target = {
            foo:1,
            set bar(value){
                this.foo = value;
            }
        }

        Reflect.set(target,'bar',888);
        console.log( target.foo );//888
```

```
var target = {
            foo:1,
            set bar(value){
                this.foo = value;
            }
        }
        var receive = {
            foo:2,
        }
        Reflect.set(target,'bar',888,receive);
        console.log( target.foo );//1
        console.log( receive.foo );//888
```

(2) `Reflect.get(` target, name, receiver `)`

> Reflect.get方法查找并返回target对象的name属性，如果没有该属性，则返回undefined,如果name属性部署了读取函数（ `getter` ），则读取函数的 `this` 绑定receiver

```
var target = {
            foo:1,
            get bar(){
                return this.foo;
            }
        }
        var receive = {
            foo:2,
        }
       var r = Reflect.get(target,'bar');
        console.log( r );//1
```

```
var target = {
            foo:1,
            get bar(){
                return this.foo;
            }
        }
        var receive = {
            foo:2,
        }
       var r = Reflect.get(target,'bar',receive);
        console.log( r );//2
```

(3) `Reflect.defineProperty(` target, name, desc `)`

> Reflect.defineProperty方法基本等同于`Object.defineProperty`，用来为对象定义属性。未来，后者会被逐渐废除，请从现在开始就使用Reflect.defineProperty代替它

```
        var obj = {};
       Reflect.defineProperty(obj,'index',888);
```

(4) `Reflect.deleteProperty(` target, name `)`

Reflect.deleteProperty方法等同于delete obj[name]，用于删除对象的属性。返回一个布尔值

```
var obj = {age:999};
        Reflect.deleteProperty(obj,'age');
        console.log(obj);//{}
```

（5）`Reflect.has(target, name)`

Reflect.has方法对应name in obj里面的`in`运算符

```
var obj = {age:999};;
        console.log(  Reflect.has(obj,'age') );//true
```

（6）`Reflect.ownKeys(target)`

Reflect.ownKeys方法用于返回对象的所有属性，基本等同于`Object.getOwnPropertyNames`与`Object.getOwnPropertySymbols`之和

```
var obj = Object.create({},{
           age:{value:25},
           [Symbol('ok')]:{value:'二狗'}
        })
        console.log( Reflect.ownKeys(obj) )//["age", Symbol(ok)]
```

（7）`Reflect.setPrototypeOf(target, prototype)`

Reflect.setPrototypeOf方法用于设置对象的`__proto__`属性，返回第一个参数对象，对应`Object.setPrototypeOf(`obj, newProto`)`。

```
var proto = {
             show(){console.log('show');}
        }
        var obj = { age:20}
        Reflect.setPrototypeOf(obj,proto);
       obj.show();//show
```

（8）`Reflect.getPrototypeOf(`target`)`

Reflect.getPrototypeOf方法用于读取对象的`__proto__`属性，对应Object.getPrototypeOf(obj)。

```
var proto = {
            name:'二狗',
            show(){console.log('show');}
        }
        var obj = { age:20}
        Reflect.setPrototypeOf(obj,proto);
       //obj.show();//show
        var pro = Reflect.getPrototypeOf(obj);
        console.log( pro );//{name: "二狗"}
```

（9）`Reflect.getOwnPropertyDescriptor(`target, name`)`

eflect.getOwnPropertyDescriptor基本等同于`Object.getOwnPropertyDescriptor`

```
         var obj = { age:20}
        var pro = Reflect.getOwnPropertyDescriptor(obj,'age');
        console.log( pro );//{value: 20, writable: true, enumerabl
e: true, configurable: true}
```

（10）`Reflect.apply(`func, ctx, args`)`

> 一般来说，如果要绑定一个函数的this对象，可以这样写fn.apply(obj, args)，但是如果函数定义了自己的apply方法，就只能写成`Function.prototype.apply.call(`fn, obj, args`)`，采用Reflect对象可以简化这种操作

```
function fn(){
            console.log(this);//{age: 20}
            console.log(arguments);//[1, 2, 3]
        }
        fn.apply = function(value){
         console.log(value);
        }
        // fn.call('ok');
        obj = {age:20};
        Reflect.apply(fn,obj,[1,2,3]);
```

(11) `Reflect.construct(` target, args `)`

> Reflect.construct方法等同于new target(...args)，这提供了一种不使用 `new` ，来调用构造函数的方法

```
function Fn(...val){
            console.log(val);//[1, 2, 3]
        }
        var obj = Reflect.construct(Fn,[1,2,3]);
        console.log( obj.constructor === Fn);//true
```

(12) `Reflect.preventExtensions(` target `)`

```
Reflect.preventExtensions对应`Object.preventExtensions`方法，用于让一个对象变为不可扩展。它返回一个布尔值，表示是否操作成功
```

```
var obj = {age:25};
        Reflect.preventExtensions(obj);
        obj.index =888;
        console.log(obj);//{age: 25}
```

(13) `Reflect.isExtensible(` target `)` 可拓展的？

> Reflect.isExtensible方法对应Object.isExtensible，返回一个布尔值，表示当前对象是否可扩展

```
1.          var obj = {age:25};
2.         console.log( Reflect.isExtensible(obj) );//true
3.         Reflect.preventExtensions(obj);
4.         console.log( Reflect.isExtensible(obj) );//false
```

三、Reflect和Proxy实现观察者模式

> 观察者模式（Observer mode）指的是函数自动观察数据对象，一旦对象有变化，函数就会自动执行。

```
1.  var person = obserable({
2.          age:20,
3.          name:'二狗'
4.      });
5.      var print = function(){
6.          console.log( `person.age:${person.age}\nperson.name:${per
    son.name} `)
7.      }
8.      function obserable(obj){
9.          var proxy = new Proxy(obj,{
10.             set:function(target,prop,val,sel){
11.                 Reflect.set(target,prop,val,sel);
12.                 print();
13.             }
14.         });
15.         return proxy;
16.     }
17.     person.age = 888;
18.     //person.age:888
19.     //person.name:二狗
```

面代码中，数据对象person是观察目标，函数print是观察者。一旦数据对象发生变化，print就会自动执行

promise 诺言

07-第七章 Promise 对象

`ECMAscript 6`

一、Promise含义

> `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

Promise对象有以下两个特点:

> （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。

二、Promise基本用法

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表示成功和失败`fulfilled`和`reject`。它们是两个函数，由 JavaScript 引擎提供，不用自己部署。

```
1.          var promise = new Promise(function(){});
2.          console.log(promise);//{[[PromiseStatus]]: "pending", [[P
    romiseValue]]: undefined}
```

该函数的两个参数分别表示`fulfilled`和`reject`

```
1.  const promise = new Promise(function(success,err){
2.              if(true){
3.                  success('ok');
4.                  console.log('success');//success
5.              }else{
6.                  err('坏了');
7.                  console.log('err');
8.              }
9.          })
10.
```

Promise 新建后就会立即执行

```
1.  const promise = new Promise(function(success,err){
2.             console.log('Promise');
3.             success();
4.          });
5.
6.          promise.then(function(){
7.              console.log('success');
8.          });
9.          console.log('Hi');
10.
11.         //Promise
12.         //Hi
13.         //success
```

Promise 新建后立即执行，所以首先输出的是`Promise`。然后，`then`方法指定的回调函数，将在当前脚本所有同步任务执行完才会执行，所以`resolved`最后输出

三、then() 定义状态改变时的回调函数

> then方法的第一个参数是`success`状态的回调函数，第二个参数（可选）是`error`状态的回调函数

```
function loadImageAsny(url){
        return new Promise(function(suc,error){
            const image = new Imag();
            image.onload = function(){
                suc(image);
            }
            image.error = function(){
                error(new Error('ni error'));
            }
            image.src = url;
            console.log(url);
        })
    }

    loadImageAsny('hell.jpg').then(function(messa){
        console.log('success:'+messa);
    },function(messa){
        console.log('error=>'+messa);//error=>ReferenceError: Ima
g is not defined
    })
```

```
 function loadImageAsny(url){
            return new Promise(function(suc,error){
                const image = new Image();
                image.onload = function(){
                    suc(image);
                }
                image.error = function(){
                    error(new Error('ni error'));
                }
                image.src = url;
                console.log(url);
            })
        }

        var url ='https://gw.alicdn.com/bao/uploaded/i1/725677994/TB1GfBnedbJ8KJjy1zjXXaqapXa_!!0-item_pic.jpg';
        loadImageAsny(url).then(function(mess){
            console.log(mess);
        });
```

上面代码中，使用Promise包装了一个图片加载的`异步操作`。如果加载成功，就调用suc方法，否则就调用error方法

```
function getJson(url){
            const promise = new Promise(function(success,error){
                const xhr = new XMLHttpRequest();
                xhr.open('GET',url);
                xhr.onreadystatechange = handler;
                xhr.responseType = 'json';
                xhr.setRequetHeader('Accept','application/json');
                xhr.send();
                const handler = function(){
                    if(this.readState == 4 ){
                        if(this.status==200){
                            success(this.response);
                        }else{
                            error(new Error(this.statusText));
                        }
                    }
                }
            });
            return promise;
        }
        getJson('get.json').then(function(response){
            console.log('content:'+response);
        },function(error){
            console.log(error);
        })
```

四、catch() 捕获错误

> catch也能捕获error

```
    var promise = new Promise(function(succ,error){
            error('失败了');

        })
        promise.then(function(messa){
            console.log('succ:'+messa);
        }).catch(function(e){
            console.log('catch:'+e);//catch:失败了
        })
```

> promise抛出的一个错误，catch方法也能捕获

```
var promise = new Promise(function(succ,error){
            throw Error();
        })
        promise.catch(function(e){
            console.log('catch:'+e);//catch:Error
        })
```

比较上面两种写法，可以发现error方法的作用，等同于抛出错误。

```
var promise = new Promise(function(succ,error){
            succ('成功了');
            throw new Error;
        })
        promise.then(function(messa){
            console.log('succ:'+messa);
        }).catch(function(e){
            console.log('catch:'+e);/
        })
        //succ:成功了
```

上面代码中，Promise 在succ语句后面，再抛出错误，不会被捕获，等于没有抛出。因为 Promise 的状态一旦改变，就永久保持该状态，不会再变了

五、Promise.resolve()

> 有时需要将现有对象转为 Promise 对象，Promise.resolve方法就起到这个作用

Promise.resolve方法的参数分成四种情况：

> （1）参数是一个 Promise 实例，直接返回不做任何修改

```
var promise = new Promise(function(){});
        var pro = Promise.resolve(promise);
        console.log(pro);//{[[PromiseStatus]]: "pending", [[Promi
seValue]]: undefined}
```

（2）参数是一个thenable对象

Promise.resolve方法会将这个对象转为 Promise 对象，然后就`立即执行thenable对象的then方法`

```
var thenable = {
            then:function(resovle,reject){
                resovle('ok');
            }
        };
        var pro = Promise.resolve(thenable);
        pro.then(function(value){
            console.log(value);//ok
            console.log(pro);//{[[PromiseStatus]]: "resolved",
[[PromiseValue]]: "ok"}
        })
```

thenable对象的then方法执行后，对象p1的状态就变为resolved

（3）参数不是具有then方法的对象，或根本就不是对象

```
      var promise = Promise.resolve('ok');
        console.log(promise);//{[[PromiseStatus]]: "resolved",
[[PromiseValue]]: "ok"}
```

上面代码等同于

```
        var promise = new Promise(function(succ,sel){succ('ok')})
        console.log(promise);///{[[PromiseStatus]]: "resolved",
[[PromiseValue]]: "ok"}

```

> （4）不带有任何参数
> Promise.resolve方法允许调用时不带参数，直接返回一个`resolved`状态的Promise 对象

```
var promise = Promise.resolve();
        console.log(promise);//{[[PromiseStatus]]: "resolved", [[PromiseValue]]: undefined}
```

需要注意的是，`立即resolve`的 Promise 对象，是在`本轮“事件循环”`（event loop）的`结束时`，而不是在下一轮“事件循环”的开始时

```
setTimeout(function(){
            console.log('three');
        },0);
        Promise.resolve().then(function(){
            console.log('two');
        })
        console.log('one');
        //one
        //two
        //three
```

六、Promise.reject() 立即失败的

```
var promise = Promise.reject('出错了');
        promise.then(null,function(s){
            console.log(s);//出错了
        })
```

等同于

```
var promise = new Promise(function(resolve,reject){
            reject('出错了');
        })
        promise.then(null,function(s){
            console.log(s);//出错了
        })
```

注意，Promise.reject()方法的参数，会原封不动地作为reject的理由，变成后续方法的参数。这一点与Promise.resolve方法不一致

```
var thenable = {
            then:function(resolve,reject){
                reject('出错了');
            }
        }

        var promise = Promise.reject(thenable);
        promise.then(null,function(s){
            console.log(s === thenable);//true
        })
```

七、Promise.all()

> Promise.all方法接受一个 `数组` 作为参数,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。

```
var resolve = new Promise(function(resolve,reject){
            resolve('成功了');
        });
        var reject = new Promise(function(resolve,reject){
            setTimeout(function(){
                reject('失败了');
            },1000);
        });
        var promise = Promise.all([resolve,reject]);
        promise.then(function(message){
            console.log('success'+message);
        },function(message){
            console.log('error:'+message);//error:失败了
        })
```

注意，如果作为参数的 Promise 实例，自己定义了catch方法，那么它一旦被`rejected`，并不会触发`Promise.all()`的`catch`方法。

八、Promise.race()

Promise.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

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

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
