ECMAScript6 的 Promise 对象

1. 概念

   Promise 对象用于异步（asynchronouss）计算，一个 Promise 对象代表着一个还未完成，但预期完成的操作。

2. 出现原因：

   1） 如果你需要通过 ajax 发送多次请求，而每一次请求依赖上一次请求返回的结果作为参数来继续下次的请求，这样的话，就需要这么写代码：

```
/*-----请求A开始------*/
$.ajax({
    //...
    success:function(r1){

        /*-----请求B开始------*/
        $.ajax({
            //...
            success:function(r2){

                /*-----请求C开始------*/
                $.ajax({
                    //...
                    success:function(r3){

                    }

                });
                /*-----请求C结束------*/
            }
        });
        /*-----请求B结束------*/
    }
});
/*-----请求A结束------*/
```

   上面的例子，假设请求 C 需要依赖 B 返回的数据，那么 C 需要放在 B 的 success 函数里面。同样的，A 需要依赖 B 返回的数据，那么 A 也需要放在 B 的 success 函数里面。假设现在存在很多个请求，请求之前是相互的依赖关系，那么我们需要嵌套很多层，这样的话，代码的可读性就很差，不直观，调试起来也不方便，维护也不方便。

   2） 如果请求 C 需要依赖 A 和 B 的结果，而 A，B 是互相独立的，没有依赖关系，那么如果使用上面的实现方法，就使得 A（B）需要依赖 B（A）完成之后才能调用，这样需要更长的等待时间。

   所以，为了处理这种回调函数层层嵌套的问题（又称“回调地狱”），所以 Promise 就出现了。

3. 语法

```
new Promise(function(resolve,reject){
 //操作…
});
```

Promise 对象是全局对象，你也可以理解为一个类，创建 Promise 实例的时候，利用 new 关键字。其中，resolve 和 reject 两个参数是函数对象。resolve 是用于处理执行成功的场景，reject 是用在处理执行失败的场景，一旦操作完成就可以调用这两个函数。具体的调用是通过 then()方法来绑定操作后的处理程序，具体使用见之后的讲解。

4. 解析

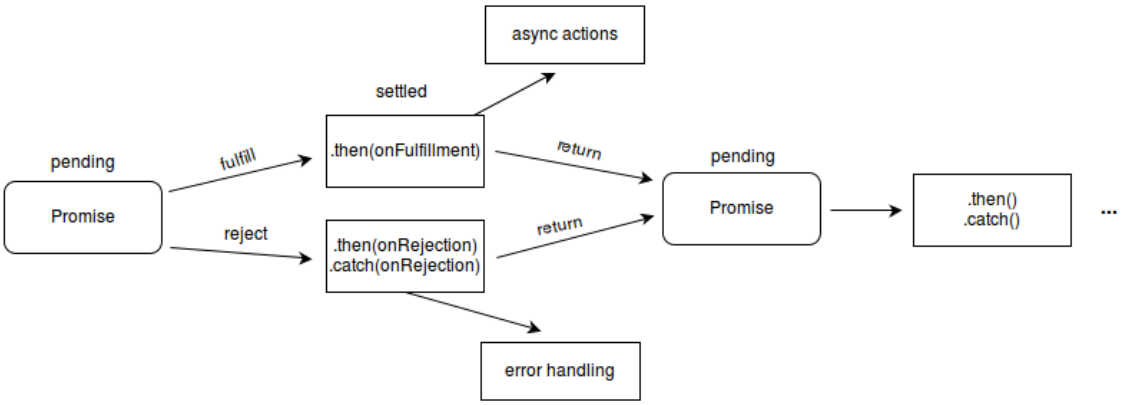
Promise 对象的三种状态：

1） pending：刚刚创建一个 Promise 实例的时候，表示初始化状态

2） fulfilled：resolve 方法调用的时候，表示操作成功

3） rejected：reject 方法调用的时候，表示操作失败

pending 状态的 promise 对象既可转换为带着一个成功值的 *fulfilled* 状态，也可变为带着一个失败信息的 *rejected* 状态。当状态发生转换时，promise.then 绑定的方法（函数句柄）就会被调用。(当绑定方法时，如果 promise 对象已经处于 fulfilled 或 rejected 状态，那么相应的方法将会被立刻调用， 所以在异步操作的完成情况和它的绑定方法之间不存在竞争条件。)



（图片来源：https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise）

5. 属性

Promise.length→长度属性，其值为 1（构造参数的数目）

Promise。prototype→Promise 构造器的原型

6. 方法

1） Promise.then()方法：用于绑定处理操作后的处理程序。

```
pro.then(function(res){
    //操作成功的处理程序
},function(error){
    //操作失败的处理程序
})
```

2） Promise.catch()方法：用于处理操作异常后的页面。

```
pro.catch(function(error){
    //操作失败的处理程序
})
```

3） then()和 catch()综合使用的例子：

```
let pro = new Promise(function(resolve,reject){
    if(true){
        resolve('操作成功');//此时resolve函数会返回值
    }
    else{
        reject('操作失败');
    }
});

pro.then(function(res){//then()函数绑定事件处理函数
    //这是操作成功的处理程序，此时res是resolve函数传过来的值
    console.log("test1 || success" + res);
}).catch(function(res){
    //这是操作失败的处理程序，此时res是reject函数传过来的值
    console.log("test1 || error" + res);
});
```

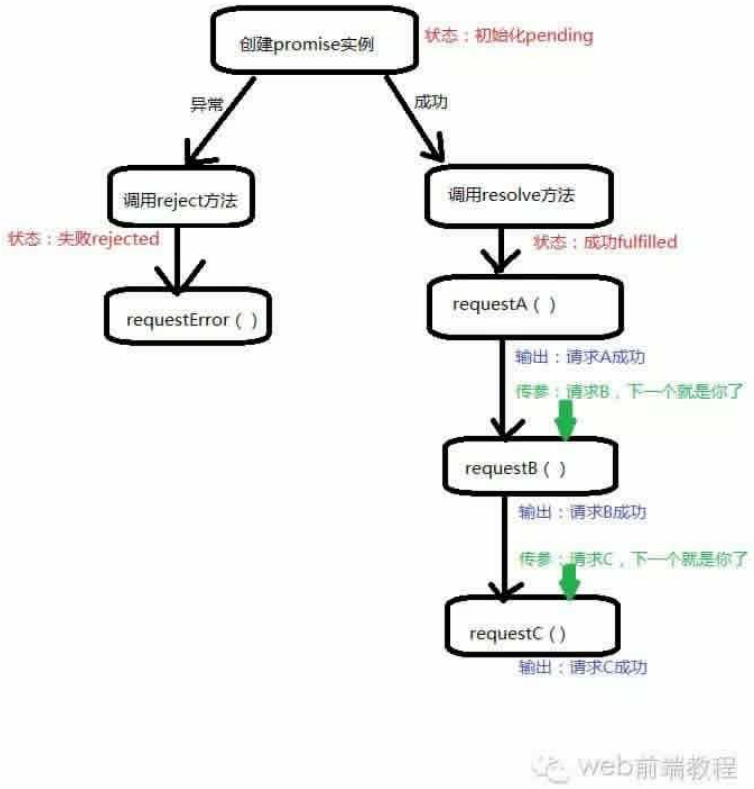
4）层层依赖用 Promise 处理：

```
let pro = new Promise(function(resolve,reject){
    if(true){
        resolve('操作成功');//此时resolve函数会返回值
    }
    else{
        reject('操作失败');
    }
});

pro.then(requestA)
   .then(requestB)
   .then(requestC)
   .catch(requestError);

function requestA(res){
    console.log(res);//输出resolve返回的值-->操作成功
    console.log("请求A成功");
    return "请求A，下一步就是B你了";
}

function requestB(res){
    console.log("上一步是"+res);
    console.log("请求B成功");
    return "请求B，下一步是C你了";
}

function requestC(res){
    console.log("上一步是"+res);
    console.log("请求C成功");
}
```

结果：

```
操作成功
请求A成功
上一步是请求A，下一步就是B你了
请求B成功
上一步是请求B，下一步是C你了
请求C成功
```

解析：上面的例子中：先创建一个实例，还声明了 4 个函数，其中三个是分别代表着请求 A、B、C 的；有了 then 方法，按照调用顺序，很直观的完成了三个操作的绑定。并且，如果请求 B 依赖请求 A 的结果，那么，可以在请求 A 的程序用 return 语句把需要的数据作为参数，传递给下一个请求。



（图片例子来源微信公众号：“web 前端教程”）

5） Promise.all()方法→接受一个数组作为参数，数组的元素是 Promise 实例对象，当参数中的实例对象的状态都为 fulfilled 时，Promise.all()才会有返回。

```
let pro1 = new Promise(function(resolve){
    setTimeout(function(){
        resolve("实例1操作成功");
    },5000);
});

let pro2 = new Promise(function(resolve){
    setTimeout(function(){
        resolve("实例2操作成功")
    },1000);
});

/*两个Promise实例都运行成功了才进行操作*/
Promise.all([pro1,pro2]).then(function(result){
    console.log(result);
}).catch(function(error){
    console.log(error + "error");
});
//返回结果: ["实例1操作成功", "实例2操作成功"]
```

解析：pro2 先进入成功 fulfilled 状态，才是 Promise.all()还不能进入 then 方法，等到 5s 之后，pro1 进入成功 fulfilled 状态，Promise.all()才进入 then 方法，才输出结果：["实例 1 操作成功", "实例 2 操作成功"]

6） Promise.race()→接受一个数组作为参数，数组的元素是 Promise 实例对象，当参数中的实例对象有一个状态（无论是 fulfilled 还是 rejected）发生变化的时候，Promise.race()就会进入 then 方法。

```
let pro1 = new Promise(function(resolve,reject){
    setTimeout(function(){
        resolve("实例1操作成功");
    },5000);
});

let pro2 = new Promise(function(resolve,reject){
    setTimeout(function(){
        reject("实例2操作失败")
    },1000);
});
/*两个Promise实例只要有一个运行成功才进行操作*/
Promise.race([pro1,pro2]).then(function(result){
    console.log(result);
}).catch(function(error){
    console.log("error:"+error);
});
//返回结果: error:实例2操作失败
```

调用reject方法

7. 原型

属性：

Promise.prototype.constructor→返回创建了实例原型的函数. 默认为 Promise 函数

方法：

Promise.prototype.catch(onRejected)→添加一个否定(rejection) 回调到当前 promise，返回一个新的 promise。如果这个回调被调用，新 promise 将以它的返回值来 resolve，否则如果当前 promise 进入 fulfilled 状态，则以当前 promise 的肯定结果作为新 promise 的肯定结果.

Promise.prototype.then(onFulfilled, onRejected)→添加肯定和否定回调到当前 promise, 返回一个新的 promise, 将以回调的返回值 来 resolve.

8. 例子

参考的例子：

```
<div id="log"></div>
    <script>
        'use strict';
        var promiseCount = 0;
        function testPromise() {
            var thisPromiseCount = ++promiseCount;
            var log = document.getElementById('log');
            log.insertAdjacentHTML('beforeend', thisPromiseCount + ') 开始(同步代码开始)<br/>');
            // 我们创建一个新的 promise: 然后用'result'字符串完成这个 promise (3 秒后)
            var p1 = new Promise(function (resolve, reject) {
                // 完成函数带着完成（resolve）或拒绝（reject）promise 的能力被执行
                log.insertAdjacentHTML('beforeend', thisPromiseCount + ') Promise 开始(异步代码开始)<br/>');

                // 这只是个创建异步完成的示例
                window.setTimeout(function () {
                    // 我们满足（fullfil）了这个 promise!
                    resolve(thisPromiseCount)
                }, Math.random() * 2000 + 1000);
            });
            // 定义当 promise 被满足时应做什么
            p1.then(function (val) {
                // 输出一段信息和一个值
                log.insertAdjacentHTML('beforeend', val + ') Promise 被满足了(异步代码结束)<br/>');
            });

            log.insertAdjacentHTML('beforeend', thisPromiseCount + ') 建立了 Promise(同步代码结束)<br/><br/>');
        }
        testPromise();
        testPromise();
        testPromise();
```

删除结果：

```
1) 开始(同步代码开始)
1) Promise开始(异步代码开始)
1) 建立了Promise(同步代码结束)

2) 开始(同步代码开始)
2) Promise开始(异步代码开始)
2) 建立了Promise(同步代码结束)

3) 开始(同步代码开始)
3) Promise开始(异步代码开始)
3) 建立了Promise(同步代码结束)

2) Promise被满足了(异步代码结束)
3) Promise被满足了(异步代码结束)
1) Promise被满足了(异步代码结束)
```

另一个比较好的例子：

https://github.com/mdn/promises-test/blob/gh-pages/index.html

9. 总结

   Promise 是一个让开发者更合理、更规范地用于处理异步操作的对象。

10. 参考资料

    https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise

    https://developer.mozilla.org/zh-CN/docs/Web/API/Element/insertAdjacentHTML

11. 接下来的安排

    ajax 源码运用

    结合 promise 与 ajax 源码运用

author：Triangel
林锦霞