你了解Promise吗

在进行异步编程中，Promise扮演了举足轻重的角色，它解决了ajax请求过程中的回调地狱的问题，令代码更具可读性。下面的介绍中，会通过一些片段代码，加上一些我自己的理解带大家一起重新温故一下Promise为编程所带来的便利。

> Promise是抽象异步处理对象以及对其进行各种操作的组件。
>
> Promise真的很重要很重要很重要，对，要强调三遍，一定要好好掌握。

实例：

```
var promise = new Promise((resolve,reject) => {
    if(true) { resolve(100) };
    if(false) { reject('error') };
});
//使用
promise.then(value => {
    console.log(value);   //100
}).catch(error => {
    console.error(error);
});
```

回调函数

在解释Promise之前，先来回顾一下什么是回调函数。

> 回调函数，也被称作高阶函数。
>
> 就是把一个函数作为参数传入“另一个函数”，然后这个函数在“另一个函数”中调用，那么这个函数，就叫做回调函数。
>
> **注意回调函数不是立即就执行。它是在包含的函数体中指定的地方“回头调用”。**
>
> 通俗来说，回调函数可以看成： A 让 B 做事，B 做着做着，信息不够，不知道怎么做了，就再让外面处理。

```
//回调函数举例1
$("#btn").click(() => {
  alert("点击后才出现");
});
//回调函数举例2
function runAsyncCallback(callback){
    setTimeout(() => {
        console.log('执行完成');
        callback('数据');
    }, 2000);
}
runAsync(data=>{
    console.log(data);  //2秒后先输出：执行完成，再输出：数据
});
```

为什么需要Promise

有非常多的应用场景我们不能立即知道应该如何继续往下执行。例如很重要的 `ajax请求` 的场景。通俗来说，由于网速的不同，可能你得到返回值的时间也是不同的，这个时候我们就需要等待，结果出来了之后才知道怎么样继续下去，例如下方的回调函数案例：

```
// 需求：当一个ajax结束后，得到的值，需要作为另外一个ajax的参数被使用（即该参数得从上一个ajax请求中获取）
var url = 'XXXXXX';
var result;

var XHR = new XMLHttpRequest();
XHR.open('GET', url, true);
XHR.send();

XHR.onreadystatechange = function() {
    if (XHR.readyState == 4 && XHR.status == 200) {
        result = XHR.response;
        console.log(result);
        // 伪代码
        var url2 = 'XXXXXX' + result.someParams;
        var XHR2 = new XMLHttpRequest();
        XHR2.open('GET', url2, true);
        XHR2.send();
        XHR2.onreadystatechange = function() {
            ...
        }
    }
}
```

当上述需求中出现第三个ajax（甚至更多）仍然依赖上一个请求的时候，代码就会变成一场灾难。也就是我们常说的 `回调地狱` 。

这时，我们可能会希望：

1. **让代码变得更具有可读性和可维护性**
2. **将请求和数据处理明确的区分开**

这时 `Promise` 就要闪亮登场了，Promise中有一个强大的then方法，可以解决刚刚遇到的回调地狱问题，并且让代码更优雅。

下面我们就一起来学习一下Promise，看一看它的强大之处。

Promise 的API

1、constructor （构造函数属性）

`Promise` 本身也是一个 `构造函数` ，需要通过这个构造函数创建一个新的 `promise` 对象作为接口，使用 `new` 来调用 `Promise` 的构造器来进行实例化，所以这个实例化出来的新对象：具有constructor属性，并且指针指向他的构造函数Promise。

```
var promise = new Promise((resolve, reject) => {
    // 异步处理
    // 处理结束后、调用resolve 或 reject
});
```

2、Instance Method （实例方法）

promise.then()

Promise对象中的 `promise.then(resolve, reject)` 实例方法，可以接收构造函数中处理的状态变化，并分别对应执行。

```
promise.then(onFulfilled, onRejected)
```

then方法有2个参数（都是可选参数）：

- resolve 成功时 `onFulfilled` 会被调用
- reject 失败时 `onRejected` 会被调用

`promise.then` 成功和失败时都可以使用，并且 **`then方法的执行结果也会返回一个Promise对象`** 。

promise.catch()

另外在只想对异常进行处理时可以采用 `promise.then(undefined, onRejected)` 这种方式，只指定reject时的回调函数即可。 不过这种情况下 `promise.catch(onRejected)` 应该是个更好的选择。

```
promise.catch(onRejected)
```

注意：在IE8及以下版本，使用 `promise.catch()` 的代码，会出现 **identifier not found** 的语法错误。（因为 `catch` 是ECMAScript的 **保留字** (Reserved Word)有关。在ECMAScript 3中保留字是不能作为对象的属性名使用的。）

解决办法：不单纯的使用 `catch` ，而是使用 `then` 来避免这个问题。

```
//then和catch方法 举例
function asyncFunction(value) {
    var p = new Promise((resolve, reject) => {
        if(typeof(value) == 'number'){
            resolve("数字");
        }else {
            reject("我不是数字");
        }
    });
    return p;
}

// 写法1：同时使用then和catch方法
asyncFunction('123').then(value => {
    console.log(value);
}).catch(error => {
    console.log(error);
});
//执行结果：数字

// 写法2：只使用 then方法，不使用catch 方法
```

```
// asyncFunction('abc').then(value => {
//     console.log(value);
// },(error) => {
//     console.log(error);
// });
//执行结果：我不是数字
```

3、Static Method （静态方法）

像 `Promise` 这样的全局对象还拥有一些静态方法。

Promise.all()

Promise.resolve()

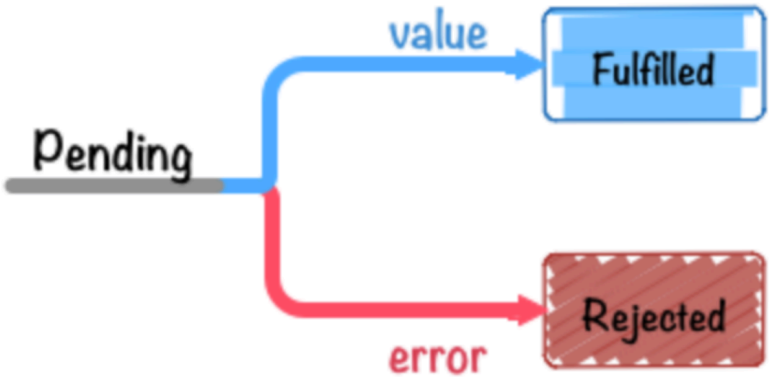
……

Promise 的状态 (Fulfilled、Rejected、Pending)

`Promise的精髓是“状态”，用维护状态、传递状态的方式来使得回调函数能够及时调用。`

用 `new Promise` 实例化的promise对象有以下三个状态。

- "unresolved" - `Pending` | 既不是resolve也不是reject的状态。等待中，或者进行中，表示Promise刚创建，还没有得到结果时的状态
- "has-resolution" - `Fulfilled` | resolve(成功)时。此时会调用 `onFulfilled`
- "has-rejection" - `Rejected` | reject(失败)时。此时会调用 `onRejected`



关于上面这三种状态的读法，其中 左侧为在 ES6 Promises 规范中定义的术语， 而右侧则是在 Promises/A+ 中描述状态的术语。

promise对象的状态，从 *Pending* 转换为 *Fulfilled* 或 *Rejected* 之后， 这个promise对象的状态就不会再发生任何变化。

当promise的对象状态发生变化时，用 `.then` 来定义只会被调用一次的函数。

Promise如何使用?

1、创建Promise对象

前面很多次强调，Promise本身就是一个构造函数，所以可以通过new创建新的Promise对象：

```
var p = new Promise((resolve, reject) => {
    //做一些异步操作
    setTimeout(() => {
        console.log('执行完成');
        resolve('我的数据');
    }, 1000);
});
//1秒之后输出：  执行完成
```

我们执行了一个异步操作，也就是setTimeout，2秒后，输出“执行完成”，并且调用resolve方法。但是只是new了一个Promise对象，并没有调用它，我们传进去的函数就已经执行了。为了避免这个现象产生，所以我们用Promise的时候一般是包在一个函数中，需要的时候去运行这个函数。

2、封装Promise对象

```
function asyncFunction(num) {
    var p = new Promise((resolve, reject) => {  //创建一个Promise的新对象p
        if (typeof num == 'number') {
            resolve();
        } else {
            reject();
        }
    });
    p.then(function() { //第一个function是resolve对应的参数
        console.log('数字');
    }, function() {   //第二个function是reject对应的参数
        console.log('我不是数字');
    })
    return p; //此处返回对象p
}

//执行这个函数我们得到了一个Promise构造出来的对象p，所以p.__proto__ === Promise.prototype，即p的指针指向了构造函数Promise，因此asyncFunction()能够使用Promise的属性和方法

//此种写法可以多次调用asyncFunction这个方法
asyncFunction('hahha'); //我不是数字
asyncFunction(1234);  //数字
```

我们刚刚讲到，**`then方法的执行结果也会返回一个Promise对象`**，得到一个结果。因此我们可以进行then的链式执行，接收上一个then返回回来的数据并继续执行，这也是`解决回调地狱`的主要方式。

3、Promise的链式操作和数据传递

下面我们就来看看如何确认then和catch两个方法返回的到底是不是新的promise对象。

```
var aPromise = new Promise(resolve => {
    resolve(100);
});
var thenPromise = aPromise.then(value => {
    console.log(value);
});
var catchPromise = thenPromise.catch(error => {
    console.error(error);
});
console.log(aPromise !== thenPromise); // => true
console.log(thenPromise !== catchPromise);// => true
```

`===` 是严格相等比较运算符，我们可以看出这三个对象都是互不相同的，这也就证明了 `then` 和 `catch` 都返回了和调用者不同的promise对象。我们通过下面这个例子进一步来理解：

```
// 1: 对同一个promise对象同时调用 `then` 方法
var aPromise = new Promise(resolve => {
    resolve(100);
});
aPromise.then(value => {
    return value * 2;
});
aPromise.then(value => {
    return value * 2;
});
aPromise.then(value => {
    console.log("1: " + value); // 1: 100
})

// vs

// 2: 对 `then` 进行 promise chain 方式进行调用
var bPromise = new Promise(resolve => {
    resolve(100);
});
bPromise.then(value => {
    return value * 2;
}).then(value => {
    return value * 2;
}).then(value => {
    console.log("2: " + value); // 2: 400
});
```

第1种写法中并没有使用promise的方法链方式，这在Promise中是应该极力避免的写法。这种写法中的 `then` 调用几乎是在同时开始执行的，而且传给每个 `then` 方法的 `value` 值都是 `100` 。

第2中写法则采用了方法链的方式将多个 `then` 方法调用串连在了一起，各函数也会严格按照 resolve → then → then → then 的顺序执行，并且传给每个 `then` 方法的 `value` 的值都是前一个promise对象通过 `return` 返回的值，实现了Promise的数据传递

4、通过Promise封装ajax 解决回调地狱问题

我们在开篇，通过一个ajax的例子，引出了回调地狱的概念，强调了通过回调函数方式解决 多级请求都依赖于上一级数据时 所引发的问题。下面我们通过刚刚学习过的Promise对上面的ajax数据依赖的案例进行重写：

```
var url = 'XXXXX';

// 封装一个get请求的方法
function getJSON(url) {
    return new Promise((resolve, reject) => {
        var XHR = new XMLHttpRequest();
        XHR.open('GET', url, true);
        XHR.send();

        XHR.onreadystatechange = function() {
            if (XHR.readyState == 4) {
                if (XHR.status == 200) {
                    try {
                        var response = JSON.parse(XHR.responseText);
                        resolve(response);
                    } catch (e) {
                        reject(e);
                    }
                } else {
                    reject(new Error(XHR.statusText));
                }
            }
        }
    })
}

getJSON(url)
    .then(resp => {
        console.log(resp);
        return url2 = 'http:xxx.yyy.com/zzz?ddd=' + resp;
    })
    .then(resp => {
        console.log(resp);
        return url3 = 'http:xxx.yyy.com/zzz?ddd=' + resp;
    });
```

new Promise写法的快捷方式

`1、Promise.resolve`

```
new Promise(resolve => {
    resolve(100);
});
// 等价于
Promise.resolve(100);   //Promise.resolve(100); 可以认为是上述代码的语法糖。

// 使用方法
Promise.resolve(100).then(value => {
    console.log(value);
});
```

```
另：`Promise.resolve` 方法另一个作用就是将 [thenable](http://liubin.org/promises-book/#Thenable) 对象转换为promise对象。

[ES6 Promises](http://liubin.org/promises-book/#es6-promises)里提到了[Thenable](http://liubin.org/promises-book/#Thenable)这个概念，简单来说它就是一个非常类似promise的东西。

就像我们有时称具有 `.length` 方法的非数组对象为Array like（类数组）一样，thenable指的是一个具有 `.then` 方法的对象。

将thenable对象转换promise对象
```

```
var promise = Promise.resolve($.ajax('/json/comment.json'));// => promise对象
promise.then(function(value){
   console.log(value);
});
```

`2、Promise.reject`

```
new Promise((resolve,reject) => {
    reject(new Error("出错了"));
});
// 等价于
 Promise.reject(new Error("出错了"));    // Promise.reject(new Error("出错了")) 就是上述代码的语法糖。

// 使用方法
Promise.reject(new Error("BOOM!")).catch(error => {
    console.error(error);
});
```

Promise.all()

> **`Promise.all`** 接收一个 promise对象的 数组作为参数 ，当这个数组里的所有promise对象全部变为resolve或reject状态的时候，它才会去调用 `.then` 方法。

> 也就是说：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。

```
// `delay`毫秒后执行resolve
function timerPromisefy(delay) {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve(delay);
        }, delay);
    });
}
var startDate = Date.now();
// 所有promise变为resolve后程序退出
Promise.all([
    timerPromisefy(1),
    timerPromisefy(32),
    timerPromisefy(64),
    timerPromisefy(128)
]).then(values => {
    console.log(Date.now() - startDate + 'ms');
    // 约128ms
    console.log(values);    // [1,32,64,128]
});
```

这说明 `timerPromisefy` 会每隔1, 32, 64, 128 ms都会有一个promise发生 `resolve` 行为，返回一个promise对象，状态为FulFilled，其状态值为传给 `timerPromisefy` 的参数，并且all会把所有异步操作的结果放进一个数组中传给then。

从上述结果可以看出，传递给 **`Promise.all`** 的promise并不是一个个的顺序执行的，而是 同时开始、并行执行 的。

Promise.race()

> 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，接收一个promise对象数组为参数。
>
> `Promise.all` 在接收到的所有的对象promise都变为 FulFilled 或者 Rejected 状态之后才会继续进行后面的处理， 与之相对的是 `Promise.race` 只要有一个promise对象进入 FulFilled 或者 Rejected 状态的话，就会继续进行后面的处理。

```
// `delay`毫秒后执行resolve
function timerPromisefy(delay) {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve(delay);
        }, delay);
    });
```

```
}
// 任何一个promise变为resolve或reject 的话程序就停止运行
Promise.race([
    timerPromisefy(1),
    timerPromisefy(32),
    timerPromisefy(64),
    timerPromisefy(128)
]).then(function (value) {
    console.log(value);    // => 1
});
```

上面的代码创建了4个promise对象，这些promise对象会分别在1ms，32ms，64ms和128ms后变为确定状态，即FulFilled，并且在第一个变为确定状态的1ms后， `.then` 注册的回调函数就会被调用，这时候确定状态的promise对象会调用 `resolve(1)` 因此传递给 `value` 的值也是1，控制台上会打印出 `1` 来。

---

promise的基本使用原理以及它在实际应用中为我们解决的问题，在上述过程中已经介绍完了，你是否理解了呢？学习是一个反复阅读，反复加深印象的过程，希望你能牢牢的掌握这一知识点，在vue、react等框架的使用中，也会频繁用到有关promise的知识，下面一起来检测一下我们的认知结果吧。

小练习

下面内容的输出结果应该是啥?

```
function test1() {
    console.log("test1");
}
function test2() {
    console.log("test2");
}
function onRejected(error) {
    console.log("捕获错误: test1 or test2", error);
}
function test3() {
    console.log("end");
}

var promise = Promise.resolve();
promise
    .then(test1)
    .then(test2)
    .catch(onRejected)
    .then(test3);
```

温馨提示：这里没有为 `then` 方法指定第二个参数(onRejected)

参考

JavaScript Promise迷你书

透彻掌握Promise的使用