30分钟，让你彻底明白Promise原理

2017-05-18

|

前言

前一阵子记录了promise的一些常规用法，这篇文章再深入一个层次，来分析分析promise的这种规则机制是如何实现的。ps:本文适合已经对promise的用法有所了解的人阅读,如果对其用法还不是太了解，可以移步我的上一篇博文。

本文的promise源码是按照Promise/A+规范来编写的（不想看英文版的移步Promise/A+规范中文翻译）

引子

为了让大家更容易理解，我们从一个场景开始讲解，让大家一步一步跟着思路思考，相信你一定会更容易看懂。

考虑下面一种获取用户id的请求处理

```
1
2
3   function getUserId() {
4       return new Promise(function(resolve) {
5           http.get(url, function(results) {
6               resolve(results.id)
7           })
8       })
9   }
10  getUserId().then(function(id) {
11  })
12
13
```

getUserId方法返回一个promise，可以通过它的then方法注册(注意注册这个

词)在promise异步操作成功时执行的回调。这种执行方式，使得异步调用变得十分顺手。

原理剖析

那么类似这种功能的Promise怎么实现呢？其实按照上面一句话，实现一个最基础的雏形还是很easy的。

极简promise雏形

```
function Promise(fn) {
    var value = null,
        callbacks = [];
    this.then = function (onFulfilled) {
        callbacks.push(onFulfilled);
    };
    function resolve(value) {
        callbacks.forEach(function (callback) {
            callback(value);
        });
    }
    fn(resolve);
}
```

上述代码很简单，大致的逻辑是这样的：

1. 调用then方法，将想要在Promise异步操作成功时执行的回调放入callbacks队列，其实也就是注册回调函数，可以向观察者模式方向思考；
2. 创建Promise实例时传入的函数会被赋予一个函数类型的参数，即resolve，它接收一个参数value，代表异步操作返回的结果，当一步操作执行成功后，用户会调用resolve方法，这时候其实真正执行的操作是将callbacks队列中的回调一一执行；

可以结合例1中的代码来看，首先new Promise时，传给promise的函数发送

异步请求，接着调用promise对象的then属性，注册请求成功的回调函数，然后当异步请求发送成功时，调用resolve(results.id)方法，该方法执行then方法注册的回调数组。

相信仔细的人应该可以看出来，then方法应该能够链式调用，但是上面的最基础简单的版本显然无法支持链式调用。想让then方法支持链式调用，其实也是很简单的：

```
this.then = function (onFulfilled) {
    callbacks.push(onFulfilled);
    return this;
};
```

see?只要简单一句话就可以实现类似下面的链式调用：

```


getUserId().then(function (id) {
}).then(function (id) {
});

```

加入延时机制

细心的同学应该发现，上述代码可能还存在一个问题：如果在then方法注册回调之前，resolve函数就执行了，怎么办？比如promise内部的函数是同步函数：

```

function getUserId() {
    return new Promise(function (resolve) {
        resolve(9876);
    });
}
getUserId().then(function (id) {
});

```

这显然是不允许的，Promises/A+规范明确要求回调需要通过异步方式执行，用以保证一致可靠的执行顺序。因此我们要加入一些处理，保证在resolve执行之前，then方法已经注册完所有的回调。我们可以这样改造下resolve函数:

```
function resolve(value) {
    setTimeout(function() {
        callbacks.forEach(function (callback) {
            callback(value);
        });
    }, 0)
}
```

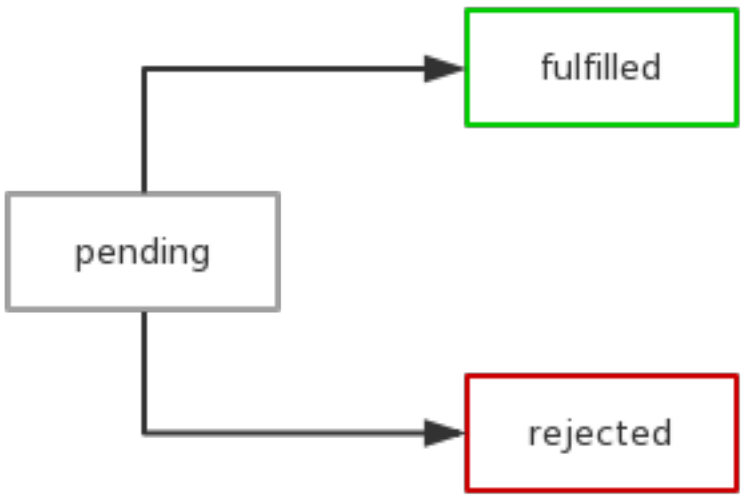
上述代码的思路也很简单，就是通过setTimeout机制，将resolve中执行回调的逻辑放置到JS任务队列末尾，以保证在resolve执行时，then方法的回调函数已经注册完成.

但是，这样好像还存在一个问题，可以细想一下：如果Promise异步操作已经成功，这时，在异步操作成功之前注册的回调都会执行，但是在Promise异步操作成功这之后调用的then注册的回调就再也不会执行了，这显然不是我们想要的。

加入状态

恩，为了解决上一节抛出的问题，我们必须加入状态机制，也就是大家熟知的pending、fulfilled、rejected。

Promises/A+规范中的2.1Promise States中明确规定了，pending可以转化为fulfilled或rejected并且只能转化一次，也就是说如果pending转化到fulfilled状态，那么就不能再转化到rejected。并且fulfilled和rejected状态只能由pending转化而来，两者之间不能互相转换。一图胜千言:



改进后的代码是这样的:

```
1
2
function Promise(fn) {
3
    var state = 'pending',
4
        value = null,
5
        callbacks = [];
6
    this.then = function (onFulfilled) {
7
        if (state === 'pending') {
8
            callbacks.push(onFulfilled);
9
            return this;
10
        }
11
        onFulfilled(value);
12
        return this;
13
    };
14
    function resolve(newValue) {
15
        value = newValue;
16
        state = 'fulfilled';
17
        setTimeout(function () {
18
            callbacks.forEach(function (callback) {
19
                callback(value);
20
            });
21
        }, 0);
22
    }
23
    fn(resolve);
24
}
25
26
```

上述代码的思路是这样的：resolve执行时，会将状态设置为fulfilled，在此之后调用then添加的新回调，都会立即执行。

这里没有任何地方将state设为rejected，为了让大家聚焦在核心代码上，这个问题后面会有一小节专门加入。

链式Promise

那么这里问题又来了，如果用户再then函数里面注册的仍然是一个Promise，该如何解决？比如下面的例4：

```
getUserId()
    .then(getUserJobById)
    .then(function (job) {
    });
function getUserJobById(id) {
    return new Promise(function (resolve) {
        http.get(baseUrl + id, function(job) {
            resolve(job);
        });
    });
}
```

这种场景相信用过promise的人都知道会有很多，那么类似这种就是所谓的链式Promise。

链式Promise是指在当前promise达到fulfilled状态后，即开始进行下一个promise（后邻promise）。那么我们如何衔接当前promise和后邻promise呢？（这是这里的难点）。

其实也不是辣么难，只要在then方法里面return一个promise就好啦。Promises/A+规范中的2.2.7就是这么说哒(微笑脸)~

下面来看看这段暗藏玄机的then方法和resolve方法改造代码：

```
function Promise(fn) {
    var state = 'pending',
        value = null,
        callbacks = [];
    this.then = function (onFulfilled) {
        return new Promise(function (resolve) {
            handle({
                onFulfilled: onFulfilled || null,
                resolve: resolve
            });
        });
    };
    function handle(callback) {
        if (state === 'pending') {
            callbacks.push(callback);
            return;
        }
        if(!callback.onFulfilled) {
            callback.resolve(value);
            return;
        }
        var ret = callback.onFulfilled(value);
        callback.resolve(ret);
    }
    function resolve(newValue) {
        if (newValue && (typeof newValue === 'object' || typeof newValue === 'function')) {
            var then = newValue.then;
            if (typeof then === 'function') {
                then.call(newValue, resolve);
                return;
            }
        }
        state = 'fulfilled';
        value = newValue;
        setTimeout(function () {
            callbacks.forEach(function (callback) {
                handle(callback);
            });
        }, 0);
    }
```

```
45      fn(resolve);
46  }
47
48
49
50
```

我们结合例4的代码，分析下上面的代码逻辑，为了方便阅读，我把例4的代码贴在这里：

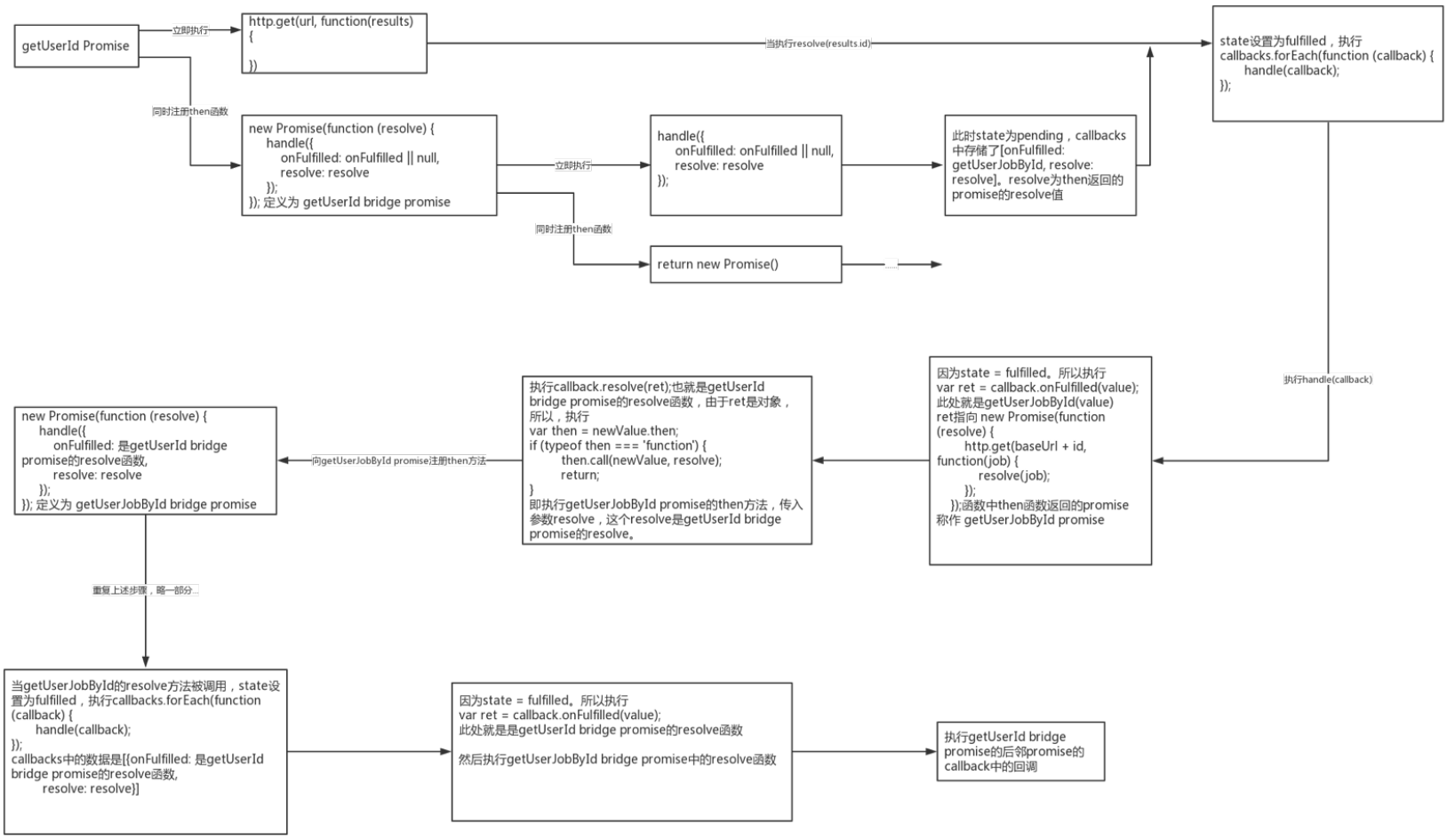
```
1
2
    getUserId()
3
        .then(getUserJobById)
4
        .then(function (job) {
5
        });
6
    function getUserJobById(id) {
7
        return new Promise(function (resolve) {
8
            http.get(baseUrl + id, function(job) {
9
                resolve(job);
10
            });
11
        });
12
    }
13
14
```

1. then方法中，创建并返回了新的Promise实例，这是串行Promise的基础，并且支持链式调用。
2. handle方法是promise内部的方法。then方法传入的形参onFulfilled以及创建新Promise实例时传入的resolve均被push到当前promise的callbacks队列中，这是衔接当前promise和后邻promise的关键所在（这里一定要好好的分析下handle的作用）。
3. getUserId生成的promise（简称getUserId promise）异步操作成功，执行其内部方法resolve，传入的参数正是异步操作的结果id
4. 调用handle方法处理callbacks队列中的回调：getUserJobById方法，生成新的promise（getUserJobById promise）
5. 执行之前由getUserId promise的then方法生成的新promise(称为bridge promise)的resolve方法，传入参数为getUserJobById

promise。这种情况下，会将该resolve方法传入getUserJobById promise的then方法中，并直接返回。

6. 在getUserJobById promise异步操作成功时，执行其callbacks中的回调：getUserId bridge promise中的resolve方法
7. 最后执行getUserId bridge promise的后邻promise的callbacks中的回调。

更直白的可以看下面的图，一图胜千言（都是根据自己的理解画出来的，如有不对欢迎指正）：



失败处理

在异步操作失败时，标记其状态为rejected，并执行注册的失败回调:

```
1
2
3   function getUserId() {
4       return new Promise(function(resolve) {
5           http.get(url, function(error, results) {
6               if (error) {
7                   reject(error);
8               }
9               resolve(results.id)
```

```
        })
    })
}
getUserId().then(function(id) {
}, function(error) {
    console.log(error)
})


```

有了之前处理fulfilled状态的经验，支持错误处理变得很容易,只需要在注册回调、处理状态变更上都要加入新的逻辑:

```


function Promise(fn) {
    var state = 'pending',
        value = null,
        callbacks = [];
    this.then = function (onFulfilled, onRejected) {
        return new Promise(function (resolve, reject) {
            handle({
                onFulfilled: onFulfilled || null,
                onRejected: onRejected || null,
                resolve: resolve,
                reject: reject
            });
        });
    };
    function handle(callback) {
        if (state === 'pending') {
            callbacks.push(callback);
            return;
        }
        var cb = state === 'fulfilled' ? callback.onFulfilled : callback.onRejected,
            ret;
        if (cb === null) {
            cb = state === 'fulfilled' ? callback.resolve : callback.reject;
            cb(value);
            return;

```

```
        }
        ret = cb(value);
        callback.resolve(ret);
    }
    function resolve(newValue) {
        if (newValue && (typeof newValue === 'object' || typeof newValue === 'function')) {
            var then = newValue.then;
            if (typeof then === 'function') {
                then.call(newValue, resolve, reject);
                return;
            }
        }
        state = 'fulfilled';
        value = newValue;
        execute();
    }
    function reject(reason) {
        state = 'rejected';
        value = reason;
        execute();
    }
    function execute() {
        setTimeout(function () {
            callbacks.forEach(function (callback) {
                handle(callback);
            });
        }, 0);
    }
    fn(resolve, reject);
}
```

上述代码增加了新的reject方法，供异步操作失败时调用，同时抽出了resolve和reject共用的部分，形成execute方法。

错误冒泡是上述代码已经支持，且非常实用的一个特性。在handle中发现没有指定异步操作失败的回调时，会直接将bridge promise(then函数返回的promise，后同)设为rejected状态，如此达成执行后续失败回调的效果。这

有利于简化串行Promise的失败处理成本，因为一组异步操作往往会对应一个实际功能，失败处理方法通常是一致的：

```
getUserId()
    .then(getUserJobById)
    .then(function (job) {
    }, function (error) {
        console.log(error);
    });
```

异常处理

细心的同学会想到：如果在执行成功回调、失败回调时代码出错怎么办？对于这类异常，可以使用try-catch捕获错误，并将bridge promise设为rejected状态。handle方法改造如下：

```
function handle(callback) {
    if (state === 'pending') {
        callbacks.push(callback);
        return;
    }
    var cb = state === 'fulfilled' ? callback.onFulfilled : callback.onRejected,
        ret;
    if (cb === null) {
        cb = state === 'fulfilled' ? callback.resolve : callback.reject;
        cb(value);
        return;
    }
    try {
        ret = cb(value);
        callback.resolve(ret);
    } catch (e) {
        callback.reject(e);
    }
}
```

如果在异步操作中，多次执行`resolve`或者`reject`会重复处理后续回调，可以通过内置一个标志位解决。

总结

刚开始看promise源码的时候总不能很好的理解then和resolve函数的运行机理，但是如果你静下心来，反过来根据执行promise时的逻辑来推演，就不难理解了。这里一定要注意的点是：promise里面的then函数仅仅是注册了后续需要执行的代码，真正的执行是在resolve方法里面执行的，理清了这层，再来分析源码会省力的多。

现在回顾下Promise的实现过程，其主要使用了设计模式中的观察者模式：

1. 通过Promise.prototype.then和Promise.prototype.catch方法将观察者方法注册到被观察者Promise对象中，同时返回一个新的Promise对象，以便可以链式调用。
2. 被观察者管理内部pending、fulfilled和rejected的状态转变，同时通过构造函数中传递的resolve和reject方法以主动触发状态转变和通知观察者。

参考文献

深入理解 Promise

JavaScript Promises … In Wicked Detail