剖析 Promise 之基础篇

spring ·2014-06-05 22:16

随着浏览器端异步操作复杂程度的日益增加，以及以 Evented I/O 为核心思想的 NodeJS 的持续火爆，Promise、Async 等异步操作封装由于解决了异步编程上面临的诸多挑战，得到了越来越广泛的应用。本文旨在剖析 Promise 的内部机制，从实现原理层面深入探讨，从而达到“知其然且知其所以然”，在使用 Promise 上更加熟练自如。如果你还不太了解 Promise，推荐阅读下 promisejs.org 的介绍。

是什么

Promise 是一种对异步操作的封装，可以通过独立的接口添加在异步操作执行成功、失败时执行的方法。主流的规范是 Promises/A+。

Promise 较通常的回调、事件/消息，在处理异步操作时具有显著的优势。其中最为重要的一点是：Promise 在语义上代表了异步操作的主体。这种准确、清晰的定位极大推动了它在编程中的普及，因为具有单一职责，而且将份内事做到极致的事物总是具有病毒式的传染力。分离输入输出参数、错误冒泡、串行/并行控制流等特性都成为 Promise 横扫异步操作编程领域的重要砝码，以至于 ES6 都将其收录，并已在 Chrome、Firefox 等现代浏览器中实现。

内部机制

自从看到 Promise 的 API，我对它的实现就充满了深深的好奇，一直有心窥其究竟。接下来，将首先从最简单的基础实现开始，由浅入深的逐步探索，剖析每一个 feature 后面的故事。

为了让语言上更加准确和简练，本文做如下约定：

- Promise：代表由 Promises/A+ 规范所定义的异步操作封装方式；
- promise：代表一个 Promise 实例。

基础实现

为了增加代入感，本文从最为基础的一个应用实例开始探索：通过异步请求获取用户id，然后做一些处理。在平时大家都是习惯用回调或者事件来处理，下面我们看下 Promise 的处理方式：

```
// 例1
```

```
function getUserId() {
    return new Promise(function (resolve) {
        // 异步请求
        Y.io('/userid', {
            on: {
                success: function (id, res) {
                    resolve(JSON.parse(res).id);
                }
            }
        });
    });
}

getUserId().then(function (id) {
    // do sth with id
});
```

JS Bin

`getUserId` 方法返回一个 promise，可以通过它的 `then` 方法注册在 promise 异步操作成功时执行的回调。自然、表意的 API，用起来十分顺手。

满足这样一种使用场景的 Promise 是如何构建的呢？其实并不复杂，下面给出最基础的实现：

```
function Promise(fn) {
    var value = null,
        deferreds = [];

    this.then = function (onFulfilled) {
        deferreds.push(onFulfilled);
    };

    function resolve(value) {
        deferreds.forEach(function (deferred) {
            deferred(value);
        });
    }

    fn(resolve);
}
```

代码很短，逻辑也非常清晰：

- 调用 `then` 方法，将想要在 Promise 异步操作成功时执行的回调放入 `deferreds` 队列；
- 创建 Promise 实例时传入函数被赋予一个函数类型的参数，即 `resolve`，用以在合适的时机触发异步操作成功。真正执行的操作是将 `deferreds` 队列中的回调一一执行；
- `resolve` 接收一个参数，即异步操作返回的结果，方便回调使用。

有时需要注册多个回调，如果能够支持 jQuery 那样的链式操作就好了！事实上，这很容易：

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

这个小改进带来的好处非常明显，当真是一个大收益的小创新呢：

```
// 例2

getUserId().then(function (id) {
    // do sth with id
}).then(function (id) {
    // do sth else with id
});
```

JS Bin

延时

如果 promise 是同步代码，`resolve` 会先于 `then` 执行，这时 `deferreds` 队列还空无一物，更严重的是，后续注册的回调再也不会被执行了：

```
// 例3

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

getUserId().then(function (id) {
    // do sth with id
});
```

JS Bin

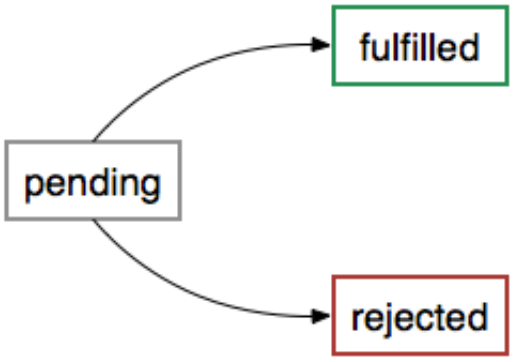
此外，Promises/A+ 规范明确要求回调需要通过异步方式执行，用以保证一致可靠的执行顺序。为解决这两个问题，可以通过 `setTimeout` 将 `resolve` 中执行回调的逻辑放置到 JS 任务队列末尾：

```
function resolve(value) {
    setTimeout(function () {
        deferreds.forEach(function (deferred) {
            deferred(value);
        });
    }, 0);
}
```

引入状态

Hmm，好像存在一点问题：如果 Promise 异步操作已经成功，之后调用 `then` 注册的回调再也不会执行了，而这是不符合我们预期的。

解决这个问题，需要引入规范中所说的 States，即每个 Promise 存在三个互斥状态：pending、fulfilled、rejected，它们之间的关系是：



经过改进后的代码：

```
function Promise(fn) {
    var state = 'pending',
        value = null,
        deferreds = [];

    this.then = function (onFulfilled) {
        if (state === 'pending') {
```

```
            deferreds.push(onFulfilled);
            return this;
        }
        onFulfilled(value);
        return this;
    };

    function resolve(newValue) {
        value = newValue;
        state = 'fulfilled';
        setTimeout(function () {
            deferreds.forEach(function (deferred) {
                deferred(value);
            });
        }, 0);
    }

    fn(resolve);
}
```

JS Bin

`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。

似乎少了点什么，哦，是的，没有任何地方将 state 设为 rejected，这个问题稍后会聊，方便聚焦在核心代码上。

串行 Promise

在这一小节，将要探索的是 Promise 的 Killer Feature：**串行 Promise**，这是最为有趣也最为神秘的一个功能。

串行 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）。例如获取用户 id 后，再根据用户 id 获取用户手机号等其他信息，这样的场景比比皆是：

```
// 例4

getUserId()
    .then(getUserMobileById)
    .then(function (mobile) {
        // do sth with mobile
```

```
    });

function getUserMobileById(id) {
    return new Promise(function (resolve) {
        Y.io('/usermobile/' + id, {
            on: {
                success: function (i, o) {
                    resolve(JSON.parse(o).mobile);
                }
            }
           });
    });
}
```

JS Bin

这个 feature 实现的难点在于：如何衔接当前 promise 和后邻 promise。

首先对 `then` 方法进行改造：

```
this.then = function (onFulfilled) {
    return new Promise(function (resolve) {
        handle({
            onFulfilled: onFulfilled || null,
            resolve: resolve
        });
    });
};

function handle(deferred) {
    if (state === 'pending') {
        deferreds.push(deferred);
        return;
    }

    var ret = deferred.onFulfilled(value);
    deferred.resolve(ret);
}
```

`then` 方法改变很多，这是一段暗藏玄机的代码：

- `then` 方法中，创建了一个新的 Promise 实例，并作为返回值，这类 promise，权且称作 bridge promise。这是串行 Promise 的基础。另外，因为返回类型一致，之前的链式执行仍

然被支持；

- `handle` 方法是当前 promise 的内部方法。这一点很重要，看不懂的童鞋可以去补充下闭包的知识。`then` 方法传入的形参 `onFullfilled`，以及创建新 Promise 实例时传入的 `resolve` 均被压入当前 promise 的 `deferreds` 队列中。所谓“巧妇难为无米之炊”，而这，正是衔接当前 promise 与后邻 promise 的“米”之所在。

新增的 `handle` 方法，相比改造之前的 `then` 方法，仅增加了一行代码：

```
deferred.resolve(ret);
```

这意味着当前 promise 异步操作成功后执行 `handle` 方法时，先执行 `onFulfilled` 方法，然后将其返回值作为实参执行 `resolve` 方法，而这标志着后邻 promise 异步操作成功，**接力**工作就这样完成啦！

以例 2 代码为例，串行 Promise 执行流如下：

这就是所谓的串行 Promise？当然不是，这些改造只是为了为最后的冲刺做铺垫，它们在重构底层实现的同时，兼容了本文之前讨论的所有功能。接下来，画龙点睛之笔--最后一个方法 `resolve` 是这样被改造的：

```
function resolve(newValue) {
    if (newValue && (typeof newValue === 'object' || typeof newValue === 'functio
n')) {
        var then = newValue.then;
        if (typeof then === 'function') {
            then.call(newValue, resolve);
            return;
        }
    }
    state = 'fulfilled';
    value = newValue;
    setTimeout(function () {
        deferreds.forEach(function (deferred) {
            handle(deferred);
        });
    }, 0);
}
```

啊哈，`resolve` 方法现在支持传入的参数是一个 Promise 实例了！以例 4 为例，执行步骤如

下：

1. `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 `userid`；
2. 调用 `handle` 方法处理 `deferreds` 队列中的回调：`getUserMobileById` 方法，生成新的 promise（简称 `getUserMobileById` promise）；
3. 执行之前由 `getUserId` promise 的 `then` 方法生成的 bridge promise 的 `resolve` 方法，传入参数为 `getUserMobileById` promise。这种情况下，会将该 `resolve` 方法传入 `getUserMobileById` promise 的 `then` 方法中，并直接返回；
4. 在 `getUserMobileById` promise 异步操作成功时，执行其 `deferreds` 中的回调：`getUserId` bridge promise 的 `resolve` 方法；
5. 最后，执行 `getUserId` bridge promise 的后邻 promise 的 `deferreds` 中的回调

上述步骤实在有些复杂，主要原因是 bridge promise 的引入。不过正是得益于此，注册一个返回值也是 promise 的回调，从而实现异步操作串行的机制才得以实现。

一图胜千言，下图描述了例 4 的 Promise 执行流：

失败处理

本节处理之前遗留的 rejected 状态问题。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：

```
// 例5

function getUserId() {
    return new Promise(function (resolve, reject) {
        // 异步请求
        Y.io('/userid/1', {
            on: {
                success: function (id, res) {
                    var o = JSON.parse(res);
                    if (o.status === 1) {
                        resolve(o.id);
                    } else {
                        // 请求失败，返回错误信息
                        reject(o.errorMsg);
                    }
                }
            }
        });
```

```
    });
}

getUserId().then(function (id) {
    // do sth with id
}, function (error) {
    console.log(error);
});
```

JS Bin

有了之前处理 fulfilled 状态的经验，支持错误处理变得很容易。毫无疑问的是，这将加倍 code base，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,
        deferreds = [];

    this.then = function (onFulfilled, onRejected) {
        return new Promise(function (resolve, reject) {
            handle({
                onFulfilled: onFulfilled || null,
                onRejected: onRejected || null,
                resolve: resolve,
                reject: reject
            });
        });
    };

    function handle(deferred) {
        if (state === 'pending') {
            deferreds.push(deferred);
            return;
        }

        var cb = state === 'fulfilled' ? deferred.onFulfilled : deferred.onRejected
,
            ret;
        if (cb === null) {
            cb = state === 'fulfilled' ? deferred.resolve : deferred.reject;
            cb(value);
            return;
        }
        ret = cb(value);
        deferred.resolve(ret);
```

```
    }

    function resolve(newValue) {
        if (newValue && (typeof newValue === 'object' || typeof newValue === 'fun
ction')) {
            var then = newValue.then;
            if (typeof then === 'function') {
                then.call(newValue, resolve, reject);
                return;
            }
        }
        state = 'fulfilled';
        value = newValue;
        finale();
    }

    function reject(reason) {
        state = 'rejected';
        value = reason;
        finale();
    }

    function finale() {
        setTimeout(function () {
            deferreds.forEach(function (deferred) {
                handle(deferred);
            });
        }, 0);
    }

    fn(resolve, reject);
}
```

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 `finale` 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 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：

```
// 例6

getUserId()
    .then(getUserMobileById)
```

```
    .then(function (mobile) {
        // do sth else with mobile
    }, function (error) {
        // getUserId或者getUerMobileById时出现的错误
        console.log(error);
    });
```

JS Bin

异常处理

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(deferred) {
    if (state === 'pending') {
        deferreds.push(deferred);
        return;
    }

    var cb = state === 'fulfilled' ? deferred.onFulfilled : deferred.onRejected,
        ret;
    if (cb === null) {
        cb = state === 'fulfilled' ? deferred.resolve : deferred.reject;
        cb(value);
        return;
    }
    try {
        ret = cb(value);
        deferred.resolve(ret);
    } catch (e) {
        deferred.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

Promise 作为异步操作的一种 Monad，魔幻一般的 API 让人难以驾驭。本文从简单的基础实现

起步，逐步添加内置状态、串行、失败处理/失败冒泡、异常处理等关键特性，最终达到类似由 Forbes Lindesay 所完成的一个简单 Promise 实现的效果。在让我本人更加深刻理解 Promise 魔力之源的同时，希望为各位更加熟练的使用这一实用工具带来一些帮助。

预告

下一篇关于 Promise 的文章中，将重点关注高阶应用的一些场景，例如并行 Promise、基于 Promise 的异步操作流封装、语法糖等。敬请期待。

参考

- Introduction to Promises
- JavaScript Promises ... In Wicked Detail
- A Gentle Introduction to Monads in JavaScript