混沌传奇 Lv3
2018年03月13日 阅读 31675

Promise原理讲解 && 实现一个Promise对象 (遵循Promise/A+规范)

> 提示：本篇文章的Promise代码实现部分讲解不够由浅入深， 推荐大家看我另一篇更细致的讲解Promise实现原理的分析文章： 《从零一步一步实现一个完整版的Promise》

1.什么是Promise?

> Promise是JS异步编程中的重要概念，异步抽象处理对象，是目前比较流行Javascript异步编程解决方案之一

2.对于几种常见异步编程方案

- 回调函数
- 事件监听
- 发布/订阅（深入了解发布/订阅，可以看我的文章用发布订阅模式编写一个可被其他对象拓展复用的自定义事件系统）
- Promise对象

这里就拿回调函数说说

(1) 对于回调函数 我们用Jquery的ajax获取数据时 都是以回调函数方式获取的数据

```js
$.get(url, (data) => {
    console.log(data)
)
```

(2) 如果说 当我们需要发送多个异步请求 并且每个请求之间需要相互依赖 那这时 我们只能 以嵌套方式来解决 形成 "回调地狱"

```
    $.get(data1.url, data2 => {
        console.log(data1)
    })
})
```

这样一来，在处理越多的异步逻辑时，就需要越深的回调嵌套，这种编码模式的问题主要有以下几个：

- 代码逻辑书写顺序与执行顺序不一致，不利于阅读与维护。
- 异步操作的顺序变更时，需要大规模的代码重构。
- 回调函数基本都是匿名函数，bug 追踪困难。
- 回调函数是被第三方库代码（如上例中的 ajax ）而非自己的业务代码所调用的，造成了 IoC 控制反转。

Promise 处理多个相互关联的异步请求

(1) 而我们Promise 可以更直观的方式 来解决 "回调地狱"

```js
const request = url => {
    return new Promise((resolve, reject) => {
        $.get(url, data => {
            resolve(data)
        });
    })
};

// 请求data1
request(url).then(data1 => {
    return request(data1.url);
}).then(data2 => {
    return request(data2.url);
}).then(data3 => {
    console.log(data3);
}).catch(err => throw new Error(err));
```

(2) 相信大家在 vue/react 都是用axios fetch 请求数据 也都支持 Promise API

```
import axios from 'axios';
axios.get(url).then(data => {
```

> Axios 是一个基于 promise 的 HTTP 库，可以用在浏览器和 node.js 中。

3.Promise使用

Promise 是一个构造函数， new Promise 返回一个 promise对象 接收一个excutor执行函数作为参数, excutor有两个函数类型形参resolve reject

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

promise相当于一个状态机

promise的三种状态

- pending
- fulfilled
- rejected

(1) promise 对象初始化状态为 pending

(2) 当调用resolve(成功)，会由pending => fulfilled

(3) 当调用reject(失败)，会由pending => rejected

> 注意promsie状态 只能由 pending => fulfilled/rejected, 一旦修改就不能再变

promise对象方法

(1) then方法注册 当resolve(成功)/reject(失败)的回调函数

```
// onFulfilled 是用来接收promise成功的值
// onRejected 是用来接收promise失败的原因
promise.then(onFulfilled, onRejected);
```

(2) resolve(成功) onFulfilled会被调用

```js
const promise = new Promise((resolve, reject) => {
   resolve('fulfilled'); // 状态由 pending => fulfilled
});
promise.then(result => { // onFulfilled
    console.log(result); // 'fulfilled'
}, reason => { // onRejected 不会被调用

})
```

(3) reject(失败) onRejected会被调用

```js
const promise = new Promise((resolve, reject) => {
   reject('rejected'); // 状态由 pending => rejected
});
promise.then(result => { // onFulfilled 不会被调用

}, reason => { // onRejected
    console.log(reason); // 'rejected'
})
```

(4) promise.catch

在链式写法中可以捕获前面then中发送的异常,

```js
promise.catch(onRejected)
相当于
promise.then(null, onRrejected);

// 注意
// onRejected 不能捕获当前onFulfilled中的异常
promise.then(onFulfilled, onRrejected);

// 可以写成:
promise.then(onFulfilled)
       .catch(onRrejected);
```

promise chain

```js
function taskA() {
    console.log("Task A");
}
function taskB() {
    console.log("Task B");
}
function onRejected(error) {
    console.log("Catch Error: A or B", error);
}

var promise = Promise.resolve();
promise
    .then(taskA)
    .then(taskB)
    .catch(onRejected) // 捕获前面then方法中的异常
```

Promise的静态方法

(1) Promise.resolve 返回一个fulfilled状态的promise对象

```js
Promise.resolve('hello').then(function(value){
    console.log(value);
});

Promise.resolve('hello');
// 相当于
const promise = new Promise(resolve => {
   resolve('hello');
});
```

(2) Promise.reject 返回一个rejected状态的promise对象

```js
Promise.reject(24);
new Promise((resolve, reject) => {
   reject(24);
});
```

(3) Promise.all 接收一个promise对象数组为参数

```js
const p1 = new Promise((resolve, reject) => {
    resolve(1);
});

const p2 = new Promise((resolve, reject) => {
    resolve(2);
});

const p3 = new Promise((resolve, reject) => {
    resolve(3);
});

Promise.all([p1, p2, p3]).then(data => {
    console.log(data); // [1, 2, 3] 结果顺序和promise实例数组顺序是一致的
}, err => {
    console.log(err);
});
```

(4) Promise.race 接收一个promise对象数组为参数

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。

```js
function timerPromisefy(delay) {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            resolve(delay);
        }, delay);
    });
}
var startDate = Date.now();

Promise.race([
    timerPromisefy(10),
    timerPromisefy(20),
    timerPromisefy(30)
]).then(function (values) {
    console.log(values); // 10
});
```

4.Promise 代码实现

```
 * Promise/A+规范译文:
 * https://malcolmyu.github.io/2015/06/12/Promises-A-Plus/#note-4
 */

// promise 三个状态
const PENDING = "pending";
const FULFILLED = "fulfilled";
const REJECTED = "rejected";

function Promise(excutor) {
    let that = this; // 缓存当前promise实例对象
    that.status = PENDING; // 初始状态
    that.value = undefined; // fulfilled状态时 返回的信息
    that.reason = undefined; // rejected状态时 拒绝的原因
    that.onFulfilledCallbacks = []; // 存储fulfilled状态对应的onFulfilled函数
    that.onRejectedCallbacks = []; // 存储rejected状态对应的onRejected函数

    function resolve(value) { // value成功态时接收的终值
        if(value instanceof Promise) {
            return value.then(resolve, reject);
        }

        // 为什么resolve 加setTimeout?
        // 2.2.4规范 onFulfilled 和 onRejected 只允许在 execution context 栈仅包含平台代码时运行.
        // 注1 这里的平台代码指的是引擎、环境以及 promise 的实施代码。实践中要确保 onFulfilled 和
onRejected 方法异步执行，且应该在 then 方法被调用的那一轮事件循环之后的新执行栈中执行。

        setTimeout(() => {
            // 调用resolve 回调对应onFulfilled函数
            if (that.status === PENDING) {
                // 只能由pending状态 => fulfilled状态 (避免调用多次resolve reject)
                that.status = FULFILLED;
                that.value = value;
                that.onFulfilledCallbacks.forEach(cb => cb(that.value));
            }
        });
    }

    function reject(reason) { // reason失败态时接收的拒因
        setTimeout(() => {
            // 调用reject 回调对应onRejected函数
            if (that.status === PENDING) {
                // 只能由pending状态 => rejected状态 (避免调用多次resolve reject)
                that.status = REJECTED;
                that.reason = reason;
                that.onRejectedCallbacks.forEach(cb => cb(that.reason));
            }
```

```
    // 捕获在excutor执行器中抛出的异常
    // new Promise((resolve, reject) => {
    //     throw new Error('error in excutor')
    // })
    try {
        excutor(resolve, reject);
    } catch (e) {
        reject(e);
    }
}

/**
 * resolve中的值几种情况:
 * 1.普通值
 * 2.promise对象
 * 3.thenable对象/函数
 */

/**
 * 对resolve 进行改造增强 针对resolve中不同值情况 进行处理
 * @param  {promise} promise2 promise1.then方法返回的新的promise对象
 * @param  {[type]} x         promise1中onFulfilled的返回值
 * @param  {[type]} resolve   promise2的resolve方法
 * @param  {[type]} reject    promise2的reject方法
 */
function resolvePromise(promise2, x, resolve, reject) {
    if (promise2 === x) {  // 如果从onFulfilled中返回的x 就是promise2 就会导致循环引用报错
        return reject(new TypeError('循环引用'));
    }

    let called = false; // 避免多次调用
    // 如果x是一个promise对象  (该判断和下面 判断是不是thenable对象重复 所以可有可无)
    if (x instanceof Promise) { // 获得它的终值 继续resolve
        if (x.status === PENDING) { // 如果为等待态需等待直至 x 被执行或拒绝 并解析y值
            x.then(y => {
                resolvePromise(promise2, y, resolve, reject);
            }, reason => {
                reject(reason);
            });
        } else { // 如果 x 已经处于执行态/拒绝态(值已经被解析为普通值)，用相同的值执行传递下去 promise
            x.then(resolve, reject);
        }
        // 如果 x 为对象或者函数
    } else if (x != null && ((typeof x === 'object') || (typeof x === 'function'))) {
        try { // 是否是thenable对象 (具有then方法的对象/函数)
            let then = x.then;
```

```
                    called = true;
                    resolvePromise(promise2, y, resolve, reject);
                }, reason => {
                    if(called) return;
                    called = true;
                    reject(reason);
                })
            } else { // 说明是一个普通对象/函数
                resolve(x);
            }
        } catch(e) {
            if(called) return;
            called = true;
            reject(e);
        }
    } else {
        resolve(x);
    }
}

/**
 * [注册fulfilled状态/rejected状态对应的回调函数]
 * @param  {function} onFulfilled fulfilled状态时 执行的函数
 * @param  {function} onRejected  rejected状态时 执行的函数
 * @return {function} newPromsie  返回一个新的promise对象
 */
Promise.prototype.then = function(onFulfilled, onRejected) {
    const that = this;
    let newPromise;
    // 处理参数默认值 保证参数后续能够继续执行
    onFulfilled =
        typeof onFulfilled === "function" ? onFulfilled : value => value;
    onRejected =
        typeof onRejected === "function" ? onRejected : reason => {
            throw reason;
        };

    // then里面的FULFILLED/REJECTED状态时 为什么要加setTimeout ?
    // 原因:
    // 其一 2.2.4规范 要确保 onFulfilled 和 onRejected 方法异步执行(且应该在 then 方法被调用的那一轮事件循环之后的新执行栈中执行) 所以要在resolve里加上setTimeout
    // 其二 2.2.6规范 对于一个promise，它的then方法可以调用多次.（当在其他程序中多次调用同一个promise的then时 由于之前状态已经为FULFILLED/REJECTED状态，则会走的下面逻辑),所以要确保为FULFILLED/REJECTED状态后 也要异步执行onFulfilled/onRejected

    // 其二 2.2.6规范 也是resolve函数里加setTimeout的原因
```

```
    // p1.then((value) => { // 此时p1.status 由pending状态 => fulfilled状态
    //     console.log(value); // resolve
    //     // console.log(p1.status); // fulfilled
    //     p1.then(value => { // 再次p1.then 这时已经为fulfilled状态 走的是fulfilled状态判断里的逻辑 所以我们也要确保判断里面onFuilled异步执行
    //         console.log(value); // 'resolve'
    //     });
    //     console.log('当前执行栈中同步代码');
    // })
    // console.log('全局执行栈中同步代码');
    //

    if (that.status === FULFILLED) { // 成功态
        return newPromise = new Promise((resolve, reject) => {
            setTimeout(() => {
                try{
                    let x = onFulfilled(that.value);
                    resolvePromise(newPromise, x, resolve, reject); // 新的promise resolve 上一个onFulfilled的返回值
                } catch(e) {
                    reject(e); // 捕获前面onFulfilled中抛出的异常 then(onFulfilled, onRejected)
                }
            });
        })
    }

    if (that.status === REJECTED) { // 失败态
        return newPromise = new Promise((resolve, reject) => {
            setTimeout(() => {
                try {
                    let x = onRejected(that.reason);
                    resolvePromise(newPromise, x, resolve, reject);
                } catch(e) {
                    reject(e);
                }
            });
        });
    }

    if (that.status === PENDING) { // 等待态
        // 当异步调用resolve/rejected时 将onFulfilled/onRejected收集暂存到集合中
        return newPromise = new Promise((resolve, reject) => {
            that.onFulfilledCallbacks.push((value) => {
                try {
                    let x = onFulfilled(value);
                    resolvePromise(newPromise, x, resolve, reject);
```

```
            });
            that.onRejectedCallbacks.push((reason) => {
                try {
                    let x = onRejected(reason);
                    resolvePromise(newPromise, x, resolve, reject);
                } catch(e) {
                    reject(e);
                }
            });
        });
    }
};

/**
 * Promise.all Promise进行并行处理
 * 参数: promise对象组成的数组作为参数
 * 返回值: 返回一个Promise实例
 * 当这个数组里的所有promise对象全部变为resolve状态的时候，才会resolve。
 */
Promise.all = function(promises) {
    return new Promise((resolve, reject) => {
        let done = gen(promises.length, resolve);
        promises.forEach((promise, index) => {
            promise.then((value) => {
                done(index, value)
            }, reject)
        })
    })
}

function gen(length, resolve) {
    let count = 0;
    let values = [];
    return function(i, value) {
        values[i] = value;
        if (++count === length) {
            console.log(values);
            resolve(values);
        }
    }
}

/**
 * Promise.race
 * 参数: 接收 promise对象组成的数组作为参数
 * 返回值: 返回一个Promise实例
```

```
Promise.race = function(promises) {
    return new Promise((resolve, reject) => {
        promises.forEach((promise, index) => {
           promise.then(resolve, reject);
        });
    });
}

// 用于promise方法链时 捕获前面onFulfilled/onRejected抛出的异常
Promise.prototype.catch = function(onRejected) {
    return this.then(null, onRejected);
}

Promise.resolve = function (value) {
    return new Promise(resolve => {
        resolve(value);
    });
}

Promise.reject = function (reason) {
    return new Promise((resolve, reject) => {
        reject(reason);
    });
}

/**
 * 基于Promise实现Deferred的
 * Deferred和Promise的关系
 * - Deferred 拥有 Promise
 * - Deferred 具备对 Promise的状态进行操作的特权方法（resolve reject）
 *
 *参考jQuery.Deferred
 *url: http://api.jquery.com/category/deferred-object/
 */
Promise.deferred = function() { // 延迟对象
    let defer = {};
    defer.promise = new Promise((resolve, reject) => {
        defer.resolve = resolve;
        defer.reject = reject;
    });
    return defer;
}

/**
 * Promise/A+规范测试
 * npm i -g promises-aplus-tests
```

```
try {
  module.exports = Promise
} catch (e) {
}
```

github源码地址：https://github.com/legend-li/Promise/blob/master/Promise.js

5.Promise测试

```
npm i -g promises-aplus-tests
promises-aplus-tests Promise.js
```

> Tip: 欢迎到QQ群583818653来交流前沿前端技术。我们是一个大前端交流中心，帮助大家快速进阶前端架构师
> Vue相关交流，请来QQ群366420656

6.相关知识参考资料

- ES6-promise
- Promises/A+规范-英文
- Promises/A+规范-翻译1
- Promises/A+规范-翻译-推荐
- JS执行栈
- Javascript异步编程的4种方法

> 备注：原文地址

> **关注下面的标签，发现更多相似文章**

React.js 前端 Promise