手把手教你实现一个完整的 Promise/A+

Promise是什么

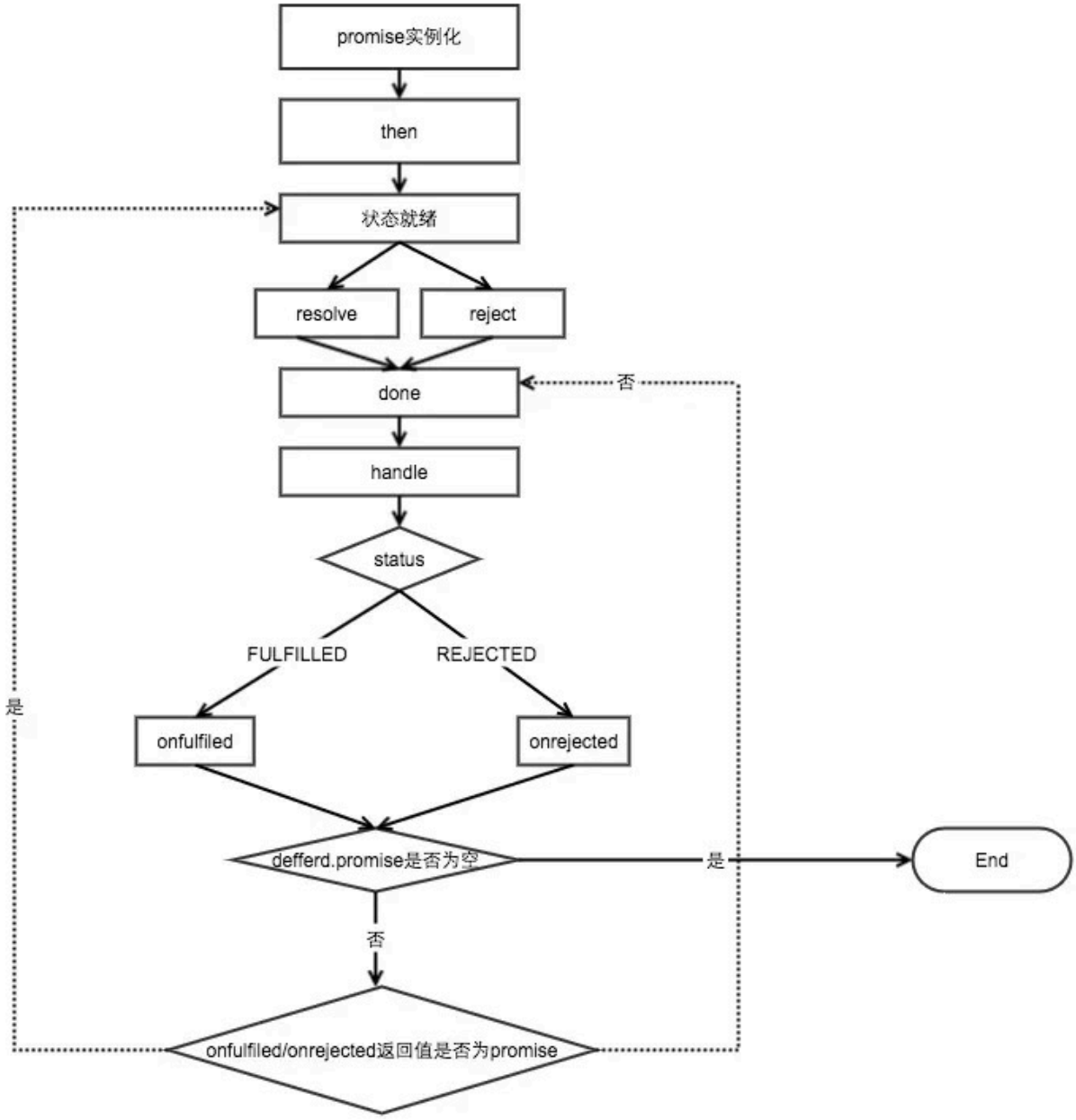
简单来说，Promise 主要就是为了解决异步回调的问题。用 Promise 来处理异步回调使得代码层次清晰，便于理解，且更加容易维护。其主流规范目前主要是 Promises/A+ 。对于 Promise 用法不熟悉的同学可以直接查看MDN.

在开始前，我们先写一个 promise 应用场景来体会下 promise 的作用。目前谷歌和火狐已经支持 es6 的 promise。我们采用 setTimeout 来模拟异步的运行，具体代码如下：

```
function fn1(resolve, reject) {
    setTimeout(function() {
        console.log('步骤一：执行');
        resolve('1');
    },500);
}

function fn2(resolve, reject) {
    setTimeout(function() {
        console.log('步骤二：执行');
        resolve('2');
    },100);
}

new Promise(fn1).then(function(val){
    console.log(val);
    return new Promise(fn2);
}).then(function(val){
    console.log(val);
    return 33;
}).then(function(val){
    console.log(val);
});
```



最终我们写的promise同样可以实现这个功能。

初步构建

下面我们来写一个简单的 promsie。Promise 的参数是函数 fn，把内部定义 resolve 方法作为参数传到 fn 中，调用 fn。当异步操作成功后会调用 resolve 方法，然后就会执行 then 中注册的回调函数。

```
function Promise(fn){
  //需要一个成功时的回调
  var callback;
  //一个实例的方法，用来注册异步事件
  this.then = function(done){
    callback = done;
  }
  function resolve(){
    callback();
  }
  fn(resolve);
}
```

加入链式支持

下面加入链式，成功回调的方法就得变成数组才能存储。同时我们给 resolve 方法添加参数，这样就不会输出 undefined。

```
function Promise(fn) {
    var promise = this,
        value = null;
        promise._resolves = [];

    this.then = function (onFulfilled) {
        promise._resolves.push(onFulfilled);
        return this;
    };

    function resolve(value) {
        promise._resolves.forEach(function (callback) {
            callback(value);
        });
    }

    fn(resolve);
}
```

- promise = this， 这样我们不用担心某个时刻 this 指向突然改变问题。
- 调用 then 方法，将回调放入 promise._resolves 队列；
- 创建 Promise 对象同时，调用其 fn, 并传入 resolve 方法，当 fn 的异步操作执行成功后，就会调用 resolve ，也就是执行`promise._resolves`队列中的回调；
- `resolve 方法`接收一个参数，即异步操作返回的结果，方便传值。
- then方法中的 return this 实现了链式调用。

但是，目前的 Promise 还存在一些问题，如果我传入的是一个不包含异步操作的函数，resolve就会先于 then 执行，也就是说 promise._resolves 是一个空数组。

为了解决这个问题，我们可以在 resolve 中添加 setTimeout，来将 `resolve` 中执行回调的逻辑放置到 JS 任务队列末尾。

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

引入状态

接着上面的步伐，引入状态：

```
function Promise(fn) {
    var promise = this,
        value = null;
        promise._resolves = [];
        promise._status = 'PENDING';

    this.then = function (onFulfilled) {
        if (promise._status === 'PENDING') {
            promise._resolves.push(onFulfilled);
            return this;
        }
        onFulfilled(value);
        return this;
    };


    function resolve(value) {
        setTimeout(function(){
            promise._status = "FULFILLED";
            promise._resolves.forEach(function (callback) {
                callback(value);
            })
        },0);
    }

    fn(resolve);
}
```

每个 Promise 存在三个互斥状态：pending、fulfilled、rejected。Promise 对象的状态改变，只有两种可能：从 pending 变为 fulfilled 和从 pending 变为 rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对 Promise 对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是

得不到结果的。

加上异步结果的传递

目前的写法都没有考虑异步返回的结果的传递，我们来加上结果的传递：

```
    function resolve(value) {
        setTimeout(function(){
            promise._status = "FULFILLED";
            promise._resolves.forEach(function (callback) {
                value = callback(value);
            })
        },0);
    }
```

串行 Promise

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）。例如我们先用ajax从后台获取用户的的数据，再根据该数据去获取其他数据。

这里我们主要对 then 方法进行改造：

```
    this.then = function (onFulfilled) {
        return new Promise(function(resolve) {
            function handle(value) {
                var ret = isFunction(onFulfilled) && onFulfilled(value) ||
value;
                resolve(ret);
            }
            if (promise._status === 'PENDING') {
                promise._resolves.push(handle);
            } else if(promise._status === FULFILLED){
                handle(value);
            }
        })

    };
```

then 方法该改变比较多啊，这里我解释下：

- 注意的是，new Promise() 中匿名函数中的 promise （promise._resolves 中的 promise）指向的都是上一个 promise 对象， 而不是当前这个刚刚创建的。
- 首先我们返回的是新的一个promise对象，因为是同类型，所以链式仍然可以实现。
- 其次，我们添加了一个 handle 函数，handle 函数对上一个 promise 的 then 中回调进行了处理，并且调用了当前的 promise 中的 resolve 方法。
- 接着将 handle 函数添加到 上一个promise 的 promise._resolves 中，当异步操作成功后就会执行 handle 函数，这样就可以 执行 当前 promise 对象的回调方法。我们的目的就达到了。

有些人在这里可能会有点犯晕，有必要对执行过程分析一下，具体参看以下代码：

```
new Promise(fn1).then(fn2).then(fn3)})
```

fn1, fn2, fn3的函数具体可参看最前面的定义。

1. 首先我们创建了一个 Promise 实例，这里叫做 promise1；接着会运行 fn1(resolve);
2. 但是 fn1 中有一个 setTimeout 函数，于是就会先跳过这一部分，运行后面的第一个 then 方法;
3. then 返回一个新的对象 promise2, promise2 对象的 resolve 方法和 then 方法的中回调函数 fn2 都被封装在 handle 中， 然后 handle 被添加到 promise1._resolves 数组中。
4. 接着运行第二个 then 方法，同样返回一个新的对象 promise3, 包含 promise3 的 resolve 方法和回调函数 fn3 的 handle 方法被添加到 promise2._resolves 数组中。
5. 到此两个 then 运行结束。 setTimeout 中的延迟时间一到，就会调用 promise1的 resolve方法。
6. resolve 方法的执行，会调用 promise1._resolves 数组中的回调，之前我们添加的 handle 方法就会被执行； 也就是 fn2 和 promsie2 的 resolve 方法，都被调用了。
7. 以此类推，fn3 会和 promise3 的 resolve 方法 一起执行，因为后面没有 then 方法了，promise3._resolves 数组是空的 。
8. 至此所有回调执行结束

但这里还存在一个问题，就是我们的 then 里面函数不能对 Promise 对象进行处理。这里我们需要再次对 then 进行修改，使其能够处理 promise 对象。

```
this.then = function (onFulfilled) {
        return new Promise(function(resolve) {
            function handle(value) {
                var ret = typeof onFulfilled === 'function' &&
onFulfilled(value) || value;
                if( ret && typeof ret ['then'] == 'function'){
                    ret.then(function(value){
                       resolve(value);
                    });
                } else {
                    resolve(ret);
                }
            }
            if (promise._status === 'PENDING') {
                promise._resolves.push(handle);
            } else if(promise._status === FULFILLED){
                handle(value);
            }
        })

    };
```

在 then 方法里面，我们对 ret 进行了判断，如果是一个 promise 对象，就会调用其 then 方法，形成一个嵌套，直到其不是promise对象为止。同时 在 then 方法中我们添加了调用 resolve 方法，这样链式得以维持。

失败处理

异步操作不可能都成功，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。

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，在注册回调、处理状态变更上都要加入新的逻辑：

```
    this.then = function (onFulfilled, onRejected) {
        return new Promise(function(resolve, reject) {
            function handle(value) {
                .......
            }
            function errback(reason){
                reason = isFunction(onRejected) && onRejected(reason) || reason;
                reject(reason);
            }
            if (promise._status === 'PENDING') {
                promise._resolves.push(handle);
                promise._rejects.push(errback);
            } else if(promise._status === 'FULFILLED'){
                handle(value);
            } else if(promise._status === 'REJECTED') {
               errback(promise._reason);
              }
        })

    };

    function reject(value) {
        setTimeout(function(){
            promise._status = "REJECTED";
            promise._rejects.forEach(function (callback) {
                promise._reason = callback( value);
            })
        },0);
    }
```

添加Promise.all方法

Promise.all 可以接收一个元素为 Promise 对象的数组作为参数，当这个数组里面所有的 Promise 对象都变为 resolve 时，该方法才会返回。

具体代码如下：

```
Promise.all = function(promises){
    if (!Array.isArray(promises)) {
        throw new TypeError('You must pass an array to all.');
      }
```

```
       // 返回一个promise 实例
      return new Promise(function(resolve,reject){
          var i = 0,
              result = [],
              len = promises.length,
               count = len;

          // 每一个 promise 执行成功后，就会调用一次 resolve 函数
          function resolver(index) {
              return function(value) {
                  resolveAll(index, value);
              };
            }

        function rejecter(reason){
            reject(reason);
        }

        function resolveAll(index,value){
            // 存储每一个promise的参数
            result[index] = value;
            // 等于0 表明所有的promise 都已经运行完成，执行resolve函数
            if( --count == 0){
                resolve(result)
            }
        }
          // 依次循环执行每个promise
          for (; i < len; i++) {
              // 若有一个失败，就执行rejecter函数
              promises[i].then(resolver(i),rejecter);
          }
      });
}
```

Promise.all会返回一个 Promise 实例，该实例直到参数中的所有的 promise 都执行成功，才会执行成功回调，一个失败就会执行失败回调。

日常开发中经常会遇到这样的需求，在不同的接口请求数据然后拼合成自己所需的数据，通常这些接口之间没有关联（例如不需要前一个接口的数据作为后一个接口的参数），这个时候 Promise.all 方法就可以派上用场了。

添加Promise.race方法

该函数和 Promise.all 相类似，它同样接收一个数组，不同的是只要该数组中的任意一个 Promise 对象的状态发生变化（无论是 resolve 还是 reject）该方法都会返回。我们只需要对 Promise.all 方法稍加修改就可以了。

```
Promise.race = function(promises){
    if (!Array.isArray(promises)) {
```

```
        throw new TypeError('You must pass an array to race.');
    }
    return Promise(function(resolve,reject){
        var i = 0,
            len = promises.length;

        function resolver(value) {
            resolve(value);
        }

        function rejecter(reason){
            reject(reason);
        }

        for (; i < len; i++) {
            promises[i].then(resolver,rejecter);
        }
    });
}
```

代码中没有类似一个 resolveAll 的函数，因为我们不需要等待所有的 promise 对象状态都发生变化，只要一个就可以了。

添加其他API以及封装函数

到这里，Promise 的主要API都已经完成了，另外我们在添加一些比较常见的方法。也对一些可能出现的错误进行了处理，最后对其进行封装。

完整的代码如下：

代码写完了，总要写几个实例看看效果啊，具体看下面的测试代码：

```
var getData100 = function(){
    return new Promise(function(resolve,reject){
        setTimeout(function(){
            resolve('100ms');
        },1000);
    });
}

var getData200 = function(){
    return new Promise(function(resolve,reject){
        setTimeout(function(){
            resolve('200ms');
        },2000);
    });
}
var getData300 = function(){
    return new Promise(function(resolve,reject){
```

```
        setTimeout(function(){
            reject('reject');
        },3000);
    });
}

getData100().then(function(data){
    console.log(data);      // 100ms
    return getData200();
}).then(function(data){
    console.log(data);      // 200ms
    return getData300();
}).then(function(data){
    console.log(data);
}, function(data){
    console.log(data);      // 'reject'
});

Promise.all([getData100(), getData200()]).then(function(data){
    console.log(data);      // [ "100ms", "200ms" ]
});

Promise.race([getData100(), getData200(), getData300()]).then(function(data){
    console.log(data);      // 100ms
});
Promise.resolve('resolve').then(function(data){
    console.log(data);      //'resolve'
})
Promise.reject('reject函数').then(function(data){
    console.log(data);
}, function(data){
    console.log(data);     //'reject函数'
})
```