```
function Promise(fn) {
    var state = 'pending',      // Promise 状态
        value = null,           // 当前 reolve 传入的原始变量值
        callbacks = [];         // 回调方法队列

    // then 方法返回一个新的 Promise, 新的 Promise 中调用一次 handle 方法, 要注意 handle 中传入的固定参数格式
    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') {              // 如果当前 Promise 的状态是 pedding
            callbacks.push(callback);           // 将 callback 函数添加到 callbacks 回调队列中
            return;                             // 直接返回
        }

        // 如果状态是不是 pedding，也就是状态发生了改变
        // 如果当前状态是 fulfilled, cd == callback.onFulfilled
        // 如果当前状态是 rejected,   cd == callback.onRejected
        var    cb = state === 'fulfilled' ? callback.onFulfilled : callback.onRejected,
               ret;

        // 如果 then 中传入的参数 onFulfilled(onRejected) 为假, 或者说没有传入参数, 那么 cd == callback.resolve(reject), 也就是 then 中返回的 Promise 中传入的 fn 的参数 resolve,然后 resolve(value),然后直接返回
        if (cb === null) {
            cb = state === 'fulfilled' ? callback.resolve : callback.reject;
            cb(value);
            return;
        }

        // 如果 callback.onFulfilled 或 callback.onRejected 不为空且不是一个函数(promise/A+规范规定,onFulfilled(onRejected)不是一个函数必须被忽略)
        if( !(typeof cb === 'function') ){
            state === 'fulfilled' ? callback.resolve(value) : callback.reject(value);
            return;
```

```
            }

            // 当 callback.onFulfilled 或 callback.onRejected 是一个函数，执行报错时，调用 callback.reject 将状态更改为 reject，将错误作为拒因传递
            try {
                ret = cb(value);
                callback.resolve(ret);
            } catch (e) {
                callback.reject(e);
            }
        }

        function resolve(newValue) {
            // 如果 newValue 是一个对象或方法
            if (newValue && (typeof newValue === 'object' || typeof newValue === 'function')) {
                var then = newValue.then;           // 获取 thenable 的 then
                // 如果 then 是一方法，也就是 newValue 是一个 thenable，then 递归调用 resolve、reject 方法,然后直接返回
                if (typeof then === 'function') {
                    // 一般来说 thenable 的 then 方法,都需要传入一个或两个方法(假装自己是一个 Promisede 的真正的 then 方法),所以 then.call(newValue,resolve,reject) 中的 resolve,reject 就是传入的两个方法,用来调用 resolve 或 reject,随后改变状态
                    then.call(newValue, resolve, reject);
                    return;
                }
            }
            state = 'fulfilled';            // 调整当前 Promise 的状态
            value = newValue;               // 如果 newValue 不是一个 thenable，设置当前 value 的值为 newValue
            execute();                      // 延迟调用回调队列中的函数
        }

        function reject(reason) {
            state = 'rejected';             // 调整当前 Promise 的状态
            value = reason;                 // 设置当前 value 的值为 reason
            execute();                      // 延迟调用回调队列中的函数
        }

        // 延迟调用函数，避免同步立即执行 callbacks 回调队列中的函数
        function execute() {
            setTimeout(function () {
                callbacks.forEach(function (callback) {
                    handle(callback);
                });
```

```
            }, 0);
        }

        fn(resolve, reject);
}




// 测试
var a = new Promise( function(resolve, reject){
        console.log(1);
        resolve(new Promise((resolve,reject)=>{
            reject('error');
        }));
} ).then(function(data){
        console.log(data);
        return new Promise(function(resolve,reject){
            resolve(4);
        });
}, function(err){
        console.log(err);
        return new Promise(function(resolve,reject){
            resolve(4);
        });
}).then(data => {
        console.log(data);
        a.a();
}).then(1)
.then( data=>{
        console.log(data);
}, err=>{
        console.log(err);
} );
```