Promise 学习笔记

第 1 章：Promise 的理解和使用

1.1. Promise 是什么?

1.1.1. 理解

1. 抽象表达:
    - Promise 是一门新的技术(ES6 规范)
    - Promise 是 JS 中进行异步编程的新解决方案
    - 备注：旧方案是单纯使用回调函数
2. 具体表达:
    - 从语法上来说: Promise 是一个 构造函数
    - 从功能上来说: promise 对象用来封装一个异步操作并可以获取其成功/失败的结果值

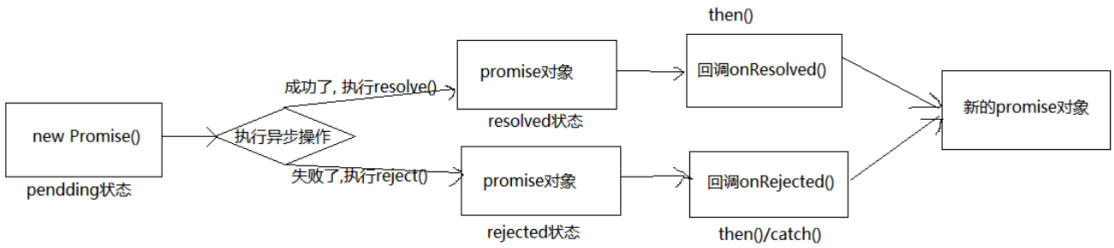
1.1.2. promise 的状态改变

1. pending 变为 `resolved` （成功）
2. pending 变为 `rejected` （失败）

说明:

- 只有这 2 种, 且 一个 promise 对象只能改变一次
- 无论变为成功还是失败, 都会有一个结果数据
- 成功的结果数据一般称为 `value`, 失败的结果数据一般称为 `reason`

1.1.3. promise 的基本流程



1.1.4. promise 的基本使用

1. 使用 1: **基本编码流程**

```
// 1) 创建 promise 对象(pending 状态), 指定执行器函数
const p = new Promise((resolve, reject) => {
    // 2) 在执行器函数中启动异步任务
    setTimeout(() => {
```

```
        const time = Date.now()
        // 3) 根据结果做不同处理
        // 3.1) 如果成功了, 调用 resolve(), 指定成功的 value, 变为 resolved 状
        态
        if (time%2===1) {
            resolve('成功的值 '+ time)
        } else { // 3.2) 如果失败了, 调用 reject(), 指定失败的 reason, 变为 rejected
状态
            reject('失败的值' + time)
        }
    }, 2000)
})
// 4) 能 promise 指定成功或失败的回调函数来获取成功的 vlaue 或失败的 reason
p.then(
    value => { // 成功的回调函数 onResolved, 得到成功的 vlaue
        console.log('成功的 value: ', value)
    },
    reason => { // 失败的回调函数 onRejected, 得到失败的 reason
        console.log('失败的 reason: ', reason)
    }
)
```

2. 使用 2: **使用 promise 封装基于定时器的异步**

```
function doDelay(time) {
    // 1. 创建 promise 对象
    return new Promise((resolve, reject) => {
        // 2. 启动异步任务
        console.log('启动异步任务')
        setTimeout(() => {
            console.log('延迟任务开始执行 ... ')
            const time = Date.now() // 假设: 时间为奇数代表成功, 为偶数代表失败
            if (time %2=== 1) { // 成功了
                // 3. 1. 如果成功了, 调用 resolve()并传入成功的 value
                resolve('成功的数据 ' + time)
            } else { // 失败了
                // 3.2. 如果失败了, 调用 reject()并传入失败的 reason
                reject('失败的数据 ' + time)
            }
        }, time)
    })
}
const promise = doDelay(2000)
promise.then(
value => {
        console.log('成功的 value: ', value)
    },
    reason => {
        console.log('失败的 reason: ', reason)
    },
)
```

3. 使用 3: **使用 promise 封装 ajax 异步请求**

```
/*
可复用的发 ajax 请求的函数: xhr + promise
```

```
*/
function promiseAjax(url) {
    return new Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest()
        xhr.onreadystatechange = () => {
            if (xhr.readyState!==4) return
            const {status, response} = xhr
            // 请求成功, 调用 resolve(value)
            if (status>=200 && status<300) {
                resolve(JSON.parse(response))
            } else { // 请求失败, 调用 reject(reason)
                reject(new Error('请求失败: status: ' + status))
            }
        }
        xhr.open("GET", url)
        xhr.send()
    })
}
promiseAjax('https://api.apiopen.top2/getJoke?page=1&count=2&type=video').then(
    data => {
        console.log('显示成功数据', data)
    },
    error => {
        alert(error.message)
    }
)
```

1.2. 为什么要用 Promise?

1.2.1. 指定回调函数的方式更加灵活

1. 旧的: 必须在启动异步任务前指定
2. promise: 启动异步任务 => 返回 promie 对象 => 给 promise 对象绑定回调函数(甚至可以在异步任务结束后指定/多个)

1.2.2. 支持链式调用, 可以解决回调地狱问题

1. 什么是回调地狱?
    - 回调函数嵌套调用, 外部回调函数异步执行的结果是嵌套的回调执行的条件
2. 回调地狱的缺点?
    - 不便于阅读
    - 不便于异常处理
3. 解决方案?
    - promise 链式调用
4. 终极解决方案?
    - async/await

```
/*
1. 指定回调函数的方式更加灵活:
旧的: 必须在启动异步任务前指定
promise: 启动异步任务 => 返回 promie 对象 => 给 promise 对象绑定回调函数
```

```
(甚至可以在异步任务结束后指定)

2. 支持链式调用, 可以解决回调地狱问题
什么是回调地狱? 回调函数嵌套调用, 外部回调函数异步执行的结果是嵌套的回调函
数执行的条件
回调地狱的缺点? 不便于阅读 / 不便于异常处理
解决方案? promise 链式调用
终极解决方案? async/await
*/
// 成功的回调函数
function successCallback(result) {
    console.log("声音文件创建成功: " + result);
}
// 失败的回调函数
function failureCallback(error) {
    console.log("声音文件创建失败: " + error);
}
/* 1.1 使用纯回调函数 */
createAudioFileAsync(audioSettings, successCallback, failureCallback)
/* 1.2. 使用 Promise */
const promise = createAudioFileAsync(audioSettings); // 2
setTimeout(() => {
    promise.then(successCallback, failureCallback);
}, 3000);
/*
2.1. 回调地狱
*/
doSomething(function(result) {
    doSomethingElse(result, function(newResult) {
        doThirdThing(newResult, function(finalResult) {
            console.log('Got the final result: ' + finalResult)
        }, failureCallback)
    }, failureCallback)
}, failureCallback)
/*
2.2. 使用 promise 的链式调用解决回调地狱
*/
doSomething().then(function(result) {
    return doSomethingElse(result)
})
.then(function(newResult) {
    return doThirdThing(newResult)
})
.then(function(finalResult) {
    console.log('Got the final result: ' + finalResult)
})
.catch(failureCallback)
/*
2.3. async/await: 回调地狱的终极解决方案
*/
async function request() {
    try {
        const result = await doSomething()
        const newResult = await doSomethingElse(result)
        const finalResult = await doThirdThing(newResult)
        console.log('Got the final result: ' + finalResult)
```

```
    } catch (error) {
        failureCallback(error)
    }
}
```

1.3. 如何使用 Promise?

1.3.1. API

1. Promise 构造函数: Promise (excutor) {}
    - executor 函数: 执行器 (resolve, reject) => {}
    - resolve 函数: 内部定义成功时我们调用的函数 value => {}
    - reject 函数: 内部定义失败时我们调用的函数 reason => {}
    - 说明: executor 会在 Promise 内部立即同步调用,异步操作在执行器中执行
2. Promise.prototype.then 方法: (onResolved, onRejected) => {}
    - onResolved 函数: 成功的回调函数 (value) => {}
    - onRejected 函数: 失败的回调函数 (reason) => {}
    - 说明: 指定用于得到成功 value 的成功回调和用于得到失败 reason 的失败回调，返回一个新的 promise 对象
3. Promise.prototype.catch 方法: (onRejected) => {}
    - onRejected 函数: 失败的回调函数 (reason) => {}
    - 说明: then()的语法糖, 相当于: then(undefined, onRejected)
4. Promise.resolve 方法: (value) => {}
    - value: 成功的数据或 promise 对象
    - 说明: 返回一个成功/失败的 promise 对象
5. Promise.reject 方法: (reason) => {}
    - reason: 失败的原因
    - 说明: 返回一个失败的 promise 对象
6. Promise.all 方法: (promises) => {}
    - promises: 包含 n 个 promise 的数组
    - 说明: 返回一个新的 promise, 只有所有的 promise 都成功才成功, 只要有一个失败了就直接失败
7. Promise.race 方法: (promises) => {}
    - promises: 包含 n 个 promise 的数组
    - 说明: 返回一个新的 promise, 第一个完成的 promise 的结果状态就是最终的结果状态

```
/*
1. Promise 构造函数: Promise (excutor) {}
excutor 函数: 同步执行 (resolve, reject) ⇒ {}
resolve 函数: 内部定义成功时我们调用的函数 value ⇒ {}
reject 函数: 内部定义失败时我们调用的函数 reason ⇒ {}
说明: excutor 会在 Promise 内部立即同步回调,异步操作在执行器中执行
2. Promise.prototype.then 方法: (onResolved, onRejected) ⇒ {}
onResolved 函数: 成功的回调函数 (value) ⇒ {}
onRejected 函数: 失败的回调函数 (reason) ⇒ {}
说明: 指定用于得到成功 value 的成功回调和用于得到失败 reason 的失败回调
返回一个新的 promise 对象
3. Promise.prototype.catch 方法: (onRejected) ⇒ {}
```

```
onRejected 函数: 失败的回调函数 (reason) => {}
说明: then()的语法糖, 相当于: then(undefined, onRejected)
4. Promise.resolve 方法: (value) => {}
value: 成功的数据或 promise 对象
说明: 返回一个成功/失败的 promise 对象
5. Promise.reject 方法: (reason) => {}
reason: 失败的原因
说明: 返回一个失败的 promise 对象
6. Promise.all 方法: (promises) => {}
promises: 包含 n 个 promise 的数组
说明: 返回一个新的 promise, 只有所有的 promise 都成功才成功, 只要有一
个失败了就直接失败
7. Promise.race 方法: (promises) => {}
promises: 包含 n 个 promise 的数组
说明: 返回一个新的 promise, 第一个完成的 promise 的结果状态就是最终的
结果状态
*/
/*
new Promise((resolve, reject) => {
    if (Date.now()%2===0) {
        resolve(1)
    } else {
        reject(2)
    }
}).then(value => {
    console.log('onResolved1()', value)
}).catch(reason => {
    console.log('onRejected1()', reason)
})
*/
const p1 = Promise.resolve(1)
const p2 = Promise.resolve(Promise.resolve(3))
const p3 = Promise.resolve(Promise.reject(5))
const p4 = Promise.reject(7)
const p5 = new Promise((resolve, reject) => {
    setTimeout(() => {
        if (Date.now()%2===0) {
            resolve(1)
        } else {
            reject(2)
        }
    }, 100);
})
const pAll = Promise.all([p1, p2, p5])
pAll.then(
    values => {console.log('all 成功了', values)},
    reason => {console.log('all 失败了', reason)}
)
// const pRace = Promise.race([p5, p4, p1])
const pRace = Promise.race([p5, p1, p4])
pRace.then(
    value => {console.log('race 成功了', value)},
    reason => {console.log('race 失败了', reason)}
)
```

1.3.2. promise 的几个关键问题

1. 如何改变 promise 的状态?
    - resolve(value): 如果当前是 pending 就会变为 resolved
    - reject(reason): 如果当前是 pending 就会变为 rejected
    - 抛出异常: 如果当前是 pending 就会变为 rejected
2. 一个 promise 指定多个成功/失败回调函数, 都会调用吗?
    - 当 promise 改变为对应状态时都会调用
3. 改变 promise 状态和指定回调函数谁先谁后?
    - 都有可能, 正常情况下是先指定回调再改变状态, 但也可以先改状态再指定回调
    - 如何先改状态再指定回调?
        - 在执行器中直接调用 resolve()/reject()
        - 延迟更长时间才调用 then()
    - 什么时候才能得到数据?
        - 如果先指定的回调, 那当状态发生改变时, 回调函数就会调用, 得到数据
        - 如果先改变的状态, 那当指定回调时, 回调函数就会调用, 得到数据
4. promise.then()返回的新 promise 的结果状态由什么决定?
    - 简单表达: 由 then()指定的回调函数执行的结果决定
    - 详细表达:
        - 如果抛出异常, 新 promise 变为 rejected, reason 为抛出的异常
        - 如果返回的是非 promise 的任意值, 新 promise 变为 resolved, value 为返回的值
        - 如果返回的是另一个新 promise, 此 promise 的结果就会成为新 promise 的结果
5. promise 如何串连多个操作任务?
    - promise 的 then() 返回一个新的 promise, 可以开成 then() 的链式调用
    - 通过 then 的链式调用串连多个同步/异步任务
6. promise 异常传透?
    - 当使用 promise 的 then 链式调用时, 可以在最后指定失败的回调,
    - 前面任何操作出了异常, 都会传到最后失败的回调中处理
7. 中断 promise 链?
    - 当使用 promise 的 then 链式调用时, 在中间中断, 不再调用后面的回调函数
    - 办法: 在回调函数中返回一个 pendding 状态的 promise 对象

第 2 章：自定义(手写)Promise

2.1. 定义整体结构

```
/*
自定义 Promise
*/
(function (window) {
    /*
    Promise 构造函数
    excutor: 内部同步执行的函数 (resolve, reject) => {}
    */
    function Promise(excutor) {
```

```
    }

    /*
    为 promise 指定成功/失败的回调函数
    函数的返回值是一个新的 promise 对象
    */
    Promise.prototype.then = function (onResolved, onRejected) {
    }

    /*
    为 promise 指定失败的回调函数
    是 then(null, onRejected)的语法糖
    */
    Promise.prototype.catch = function (onRejected) {
    }

    /*
    返回一个指定了成功 value 的 promise 对象
    */
    Promise.resolve = function (value) {
    }

    /*
    返回一个指定了失败 reason 的 promise 对象
    */
    Promise.reject = function (reason) {
    }

    /*
    返回一个 promise, 只有 promises 中所有 promise 都成功时, 才最终成功, 只要有一个失败就直接
    失败
    */
    Promise.all = function (promises) {
    }

    /*
    返回一个 promise， 一旦某个 promise 解决或拒绝， 返回的 promise 就会解决或拒绝。
    */
    Promise.race = function (promises) {
    }

    // 暴露构造函数
    window.Promise = Promise
})(window)
```

2.2. Promise 构造函数的实现

```
/*
Promise 构造函数
excutor: 内部同步执行的函数 (resolve, reject) => {}
*/
function Promise(excutor) {
```

```
    const self = this
    self.status = 'pending' // 状态值, 初始状态为 pending, 成功了变为resolved, 失败了变为 rejected
    self.data = undefined // 用来保存成功 value 或失败 reason 的属性
    self.callbacks = [] // 用来保存所有待调用的包含 onResolved 和 onRejected 回调函数的对象的数组
    /*
    异步处理成功后应该调用的函数
    value: 将交给 onResolve()的成功数据
    */
    function resolve(value) {
        if(self.status!=='pending') { // 如果当前不是 pending, 直接结束
            return
        }

        // 立即更新状态, 保存数据
        self.status = 'resolved'
        self.data = value

        // 异步调用所有待处理的 onResolved 成功回调函数
        if (self.callbacks.length>0) {
            setTimeout(() => {
                self.callbacks.forEach(obj => {
                    obj.onResolved(value)
                })
            })
        }
    }

    /*
    异步处理失败后应该调用的函数
    reason: 将交给 onRejected()的失败数据
    */
    function reject(reason) {
        if(self.status!=='pending') { // 如果当前不是 pending, 直接结束
            return
        }

        // 立即更新状态, 保存数据
        self.status = 'rejected'
        self.data = reason
        // 异步调用所有待处理的 onRejected 回调函数
        setTimeout(() => {
            self.callbacks.forEach(obj => {
                obj.onRejected(reason)
            })
        })
    }

    try {
        // 立即同步调用 excutor()处理
        excutor(resolve, reject)
    } catch (error) { // 如果出了异常, 直接失败
        reject(error)
    }
}
```

2.3. promise.then()/catch()的实现

```
/*
为 promise 指定成功/失败的回调函数
函数的返回值是一个新的 promise 对象
*/
Promise.prototype.then = function (onResolved, onRejected) {
    const self = this
    // 如果 onResolved/onRejected 不是函数, 可它指定一个默认的函数
    onResolved = typeof onResolved===='function' ? onResolved : value => value // 指定返回的 promise 为一个成功状态, 结果值为 value
    onRejected = typeof onRejected === 'function' ? onRejected : reason => {throw reason} // 指定返回的 promise 为一个失败状态, 结果值为 reason
    // 返回一个新的 promise 对象
    return new Promise((resolve, reject) => {

        /*
        专门抽取的用来处理 promise 成功/失败结果的函数
        callback: 成功/失败的回调函数
        */
        function handle(callback) {
            // 1. 抛出异常 ===> 返回的 promise 变为 rejected
            try {
                const x = callback(self.data)
                // 2. 返回一个新的 promise ===> 得到新的 promise 的结果值作为返回的 promise 的结果值
                if (x instanceof Promise) {
                    x.then(resolve, reject) // 一旦 x 成功了, resolve(value), 一旦 x 失败了: reject(reason)
                } else {
                    // 3. 返回一个一般值(undefined) ===> 将这个值作为返回的 promise 的成功值
                    resolve(x)
                }
            } catch (error) {
                reject(error)
            }
        }
        if (self.status === 'resolved') { // 当前 promise 已经成功了
            setTimeout(() => {
                handle(onResolved)
            })
        } else if (self.status === 'rejected') { // 当前 promise 已经失败了
            setTimeout(() => {
                handle(onRejected)
            })
        } else { // 当前 promise 还未确定 pending
            // 将 onResolved 和 onRejected 保存起来
            self.callbacks.push({
                onResolved(value) {
                handle(onResolved)
            },
            onRejected(reason) {
                handle(onRejected)
            }
```

```
            })
        }
    })
}

/*
为 promise 指定失败的回调函数
是 then(null, onRejected)的语法糖
*/
Promise.prototype.catch = function (onRejected) {
    return this.then(null, onRejected)
}
```

2.4. Promise.resolve()/reject()的实现

```
/*
返回一个指定了成功 value 的 promise 对象
value: 一般数据或 promise
*/
Promise.resolve = function (value) {
    return new Promise((resolve, reject) => {
        if (value instanceof Promise) {
            value.then(resolve, reject)
        } else {
            resolve(value)
        }
    })
}

/*
返回一个指定了失败 reason 的 promise 对象
reason: 一般数据/error
*/
Promise.reject = function (reason) {
    return new Promise((resolve, reject) => {
        reject(reason)
    })
}
```

2.5. Promise.all/race()的实现

```
/*
返回一个新的 promise 对象, 只有 promises 中所有 promise 都产生成功 value 时, 才
最终成功, 只要有一个失败就直接失败
*/
Promise.all = function (promises) {
    // 返回一个新的 promise
    return new Promise((resolve, reject) => {
        // 已成功的数量
        let resolvedCount = 0
        // 待处理的 promises 数组的长度
```

```
        const promisesLength = promises.length
        // 准备一个保存成功值的数组
        const values = new Array(promisesLength)
        // 遍历每个待处理的 promise
        for (let i = 0; i < promisesLength; i++) {
            // promises 中元素可能不是一个数组, 需要用 resolve 包装一下
            Promise.resolve(promises[i]).then(
                value => {
                    // 成功当前 promise 成功的值到对应的下标
                    values[i] = value
                    // 成功的数量加 1
                    resolvedCount++
                    // 一旦全部成功
                    if(resolvedCount===promisesLength) {
                        // 将所有成功值的数组作为返回 promise 对象的成功结果值
                        resolve(values)
                    }
                },
                reason => {
                    // 一旦有一个promise产生了失败结果值, 将其作为返回promise对象的失败结果值
                    reject(reason)
                }
            )
        }
    })
}

/*
返回一个 promise，一旦某个 promise 解决或拒绝， 返回的 promise 就会解决或拒绝。
*/
Promise.race = function (promises) {
    // 返回新的 promise 对象
    return new Promise((resolve, reject) => {
        // 遍历所有 promise
        for (var i = 0; i < promises.length; i++) {
            Promise.resolve(promises[i]).then(
                (value) => { // 只要有一个成功了, 返回的 promise 就成功了
                    resolve(value)
                },
                (reason) => { // 只要有一个失败了, 返回的结果就失败了
                    reject(reason)
                }
            )
        }
    })
}
```

2.6. Promise.resolveDelay()/rejectDelay()的实现

```
/*
返回一个延迟指定时间才确定结果的 promise 对象
*/
```

```
Promise.resolveDelay = function (value, time) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (value instanceof Promise) { // 如果 value 是一个 promise, 取这个
                promise 的结果值作为返回的 promise 的结果值
                value.then(resolve, reject) // 如果 value 成功, 调用
                resolve(val), 如果 value 失败了, 调用 reject(reason)
            } else {
                resolve(value)
            }
        }, time);
    })
}

/*
返回一个延迟指定时间才失败的 Promise 对象。
*/
Promise.rejectDelay = function (reason, time) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject(reason)
        }, time)
    })
}
```

2.7. ES5 function 完整版本

Promise.js

2.8. ES6 class 完整版

Promise_class.js

第 3 章：async 与 await

3.1. mdn 文档

- https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/async_function
- https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/await

3.2. async 函数

1. 函数的返回值为 promise 对象
2. promise 对象的结果由 `async` 函数执行的返回值决定

3.3. await 表达式

1. `await` 右侧的表达式一般为 promise 对象, 但也可以是其它的值
2. 如果表达式是 promise 对象, `await` 返回的是 promise 成功的值
3. 如果表达式是其它值, 直接将此值作为 `await` 的返回值

3.4. 注意

1. await 必须写在 async 函数中, 但 async 函数中可以没有 await
2. 如果 await 的 promise 失败了, 就会抛出异常, 需要通过 `try ... catch` 捕获处理

```
function fn1() {
    return Promise.resolve(1)
}

function fn2() {
    return 2
}

function fn3() {
    return Promise.reject(3)
    // return fn3.test() // 程序运行会抛出异常
}

function fn4() {
    return fn3.test() // 程序运行会抛出异常
}

// 没有使用 await 的 async 函数
async function fn5() {
    return 4
}

async function fn() {
    // await 右侧是一个成功的 promise
    const result = await fn1()
    // await 右侧是一个非 promise 的数据
    // const result = await fn2()
    // await 右侧是一个失败的 promise
    // const result = await fn3()
    // await 右侧抛出异常
    // const result = await fn4()
    console.log('result: ', result)
    return result+10
}
```

```
async function test() {
    try {
        const result2 = await fn()
        console.log('result2', result2)
    } catch (error) {
        console.log('error', error)
    }
    const result3 = await fn4()
    console.log('result4', result3)
}
// test()
```