Promise 学习笔记

1. 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 JavaScript 中进行异步编程的新解决方案
        - 异步编程：fs 文件操作、数据库操作、AJAX、定时器
    - 备注：旧方案时单纯使用回调函数
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或者失败的值
    - promise 中既可以异步任务，也可以时同步任务

1.1.2. promise 的状态改变

1.1.2.1. Promise 状态

实例对象中的一个属性：PromiseState，该属性有三种值：

- pending　未决定的
- resolved / fullfiled　成功
- rejected　失败

状态变化只会有两种情况:

1. pending 变为 resolved
2. pending 变为 rejected

说明:

- 只有这两种，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 结果属性

1.1.3.1. Promise 对象的值

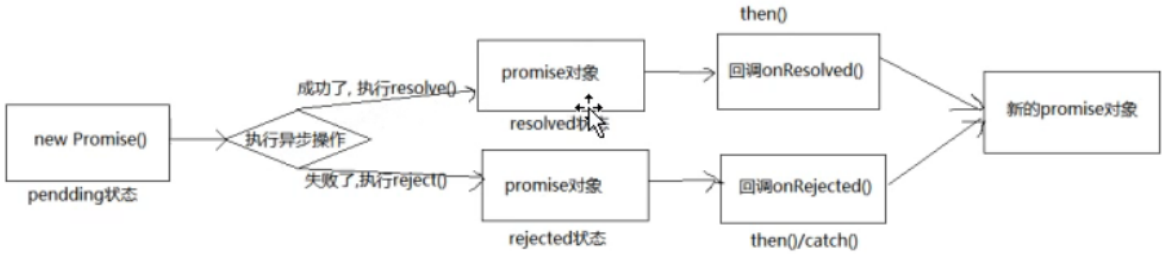
实例对象中的另一个属性：PromiseResult，保存着异步任务**成功/失败** 的结果

修改该属性的值:

- resolve 函数
- reject 函数

修改之后在后续的 then 方法中就可以把这个值取出来，进行操作

1.1.4. Promise 的工作流程



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 promise 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

1.3. Promise 封装

1.3.1. fs 模块

```
function mineReadFile (path) {
  return new Promise((resolve, reject) => {
    // 读取文件
    require('fs').readFile(path, (err, data) => {
      // 判断
      if (err) reject(err)
      // 成功
      resolve(data)
    })
  })
}
```

1.3.2. util.promisify方法

把回调函数风格的方法转变成 Promise 风格的方法

```
// 以读取文件为例

// 引入 util 模块
const util = require('util')
// 引入 fs 模块
const fs = require('fs')
```

```
7
8  // 返回一个新的函数
9  let minReadFile = util.promisify(fs.readFile);
10 minReadFile('./resource/content.txt').then(value => {
11   console.log(value.toString())
12 }, reason => {
13   console.log(reason)
14 })
```

1.3.3. 封装 Ajax 请求

```
1  function sendAJAX(url) {
2    return new Promise((resolve, reject) => {
3      const xhr = new XMLHttpRequest()
4      // 设置响应数据格式
5      xhr.responseType = 'json'
6      xhr.open('GET', url)
7      xhr.send();
8      // 处理结果
9      xhr.onreadystatechange = function () {
10       if (xhr.readyState === 4) {
11         // 判断成功
12         if (xhr.status >= 200 && xhr.status < 300) {
13           // 成功的结果
14           resolve(xhr.response)
15         } else {
16           reject(xhr.status)
17         }
18       }
19     }
20   })
21 }
22
23  // 调用
24 sendAJAX('https://api.apiopen.top/getJoke')
25 .then(value => {
26   console.log(value)
27 }, reason => {
28   console.warn(reason)
29 })
```

1.4. 如何使用 Promise?

1.4.1. API

1. Promise 构造函数：Promise(executor) {}
    - 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

then 方法返回一个新的 promise 对象

3. Promise.prototype.catch 方法：(onRejected) => {}
   - onRejected 函数：失败的回调函数 (reason) => {}
4. Promise.resolve 方法：(value) => {}
   - value：成功的数据或 promise 对象
   - 如果传递的参数为 非 promise 对象，则返回的结果为成功 promise 对象
   - 如果传入的参数为 Promise 对象，则参数的结果决定了 resolve 的结果

   说明：返回一个成功/失败的 promise 对象

```
let p1 = Promise.resolve(520)
let p2 = Promise.resolve(new Promise((resolve, reject) => {
  resolve('OK')
}))  // 这时 p2 状态为成功，成功的值为 'OK'
```

5. Promise.reject 方法：(reason) =>{}
   - reason：失败的原因

   说明：返回一个失败的 promise 对象

```
let p = Promise.reject(520)  // 无论传入的是什么，返回的都是一个失败的promise 对象
// 传入什么，失败的结果就是什么
```

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 都成功时才成功，只要有一个失败了就直接失败
   - 成功的结果时每一个 promise 对象成功结果组成的数组（有顺序）
   - 失败的结果是在这个数组中失败的那个 promise 对象失败的结果

```
let p1 = new Promise((resolve, reject) => {
  resolve('OK')
})
let p2 = Promise.resolve('Success')
let p3 = Promise.resolve('Success')

const result = Promise.all([p1, p2, p3])
```

7. Promise.race 方法：(promises) => {}
   - promises：包含 n 个 promise 的数组
   - race：赛跑/比赛

   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

```
let p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('OK')
  }, 1000)
})
let p2 = Promise.resolve('Success')
let p3 = Promise.resolve('Success')

const result = Promise.race([p1, p2, p3])  // =>结果为 p2 的结果，因为p2 先改变状态
```

1.4.2. 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（then方法）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

```
let p = new Promise((resolve, reject) => {
  resolve('ok')  // 这里状态改变了，所以下边两个回调都会执行，如果状态不改变，下面的回调都不执行
})

// 指定回调 - 1
p.then(value => {
  console.log(value)
})

// 指定回调 - 2
p.then(value => {
  alert(value)
})

```

3. 改变 promise 状态和指定回调函数谁先谁后?

   问题简单描述：promise 代码在运行时，resolve/reject改变状态先执行，还是 then 方法指定回调先执行?
   - 都有可能，正常情况下是先指定回调再改变状态，但也可以先改变状态再指定回调
     - 当执行器函数中的任务是一个同步任务(直接调 resolve()/reject()) 的时候，先改变 promise 状态，再去指定回调函数
     - 当执行器函数中的任务是一个异步任务的时候，then 方法先执行(指定回调)，改变状态后执行

```
// 这时是hen 方法先执行(指定回调)，改变状态后执行
let p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('OK')
  }, 1000)
})

p.then(value => {
  console.log(value)
})
```

- 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(回调函数什么时候执行)?
  - 如果先指定的回调，那当状态发生改变时(调用resolve()/reject()时)，回调函数就会调用，得到数据
  - 如果先改变的状态，那当指定函数时(then 方法)，回调函数就会调用，得到数据

4. promise.then() 返回的新 promise 的结果状态有什么决定?
   - 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
     - 如果返回的时另一个新的 promise，此 promise 的结果就会成为新 promise的结果
5. promise 如何串联多个操作任务?
   - promise 的 then() 返回一个新的promise，可以看成 then() 的链式调用
   - 通过 then 的链式调用串联多个同步/异步任务
6. promise异常穿透?
   - 当使用 promise 的 then 链式调用时，可以在最后指定失败的回调，
   - 前面任何操作除了异常，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 pending 状态的 promise 对象

```
let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('OK')
    }, 1000)
})

p.then(value => {
    console.log(111)
    return new Promise(() => {})
}).then(value => {
    console.log(222)
})
```

1.5. 手写 Promise

1.5.1. 函数方式

```
/*
自定义 Promise
 */

function Promise(executor) {
    // 添加属性
    this.PromiseState = 'pending'
    this.PromiseResult = null
    // 声明属性  因为实例对象不能直接调用onResolve跟onReject 所以下面then中需要先保存在callback里面
    this.callbacks = []
    // 保存实例对象的 this 的值
    const self = this //  常见的变量名有self _this that

    // resolve 函数
    function resolve(data) {
        // 判断状态
        if (self.PromiseState !== 'pending') return
        // console.log(this)  => 这里的this指向window，下面用this的话时直接修改的window
        // 1. 修改对象的状态 (PromiseState)
        self.PromiseState = 'fulfilled'
        // 2. 设置对象结果值 (PromiseResult)
        self.PromiseResult = data
        // 调用成功的回调函数
        setTimeout(() => {
            self.callbacks.forEach((item) => {
                item.onResolved(data)
            })
        })
    }

    // reject 函数
    function reject(data) {
        // 判断状态
        if (self.PromiseState !== 'pending') return
        // 1. 修改对象的状态 (PromiseState)
        self.PromiseState = 'rejected'
        // 2. 设置对象结果值 (PromiseResult)
        self.PromiseResult = data
        // 调用失败的回调函数
        setTimeout(() => {
            self.callbacks.forEach((item) => {
                item.onRejected(data)
            })
        })
    }
    try {
        // 同步调用【执行器函数】
        executor(resolve, reject)
    } catch (e) {
        // 修改 promise 对象状态
        reject(e)
```

```
    }
}

// 添加 then 方法
Promise.prototype.then = function (onResolved, onRejected) {
    const self = this
    // 判断回调函数参数
    if (typeof onRejected !== 'function') {
        onRejected = (reason) => {
            throw reason
        }
    }
    if (typeof onResolved !== 'function') {
        onResolved = (value) => value
    }
    return new Promise((resolve, reject) => {
        // 封装函数
        function callback(type) {
            try {
                // 获取回调函数的执行结果
                let result = type(self.PromiseResult)
                // 判断
                if (result instanceof Promise) {
                    result.then(
                        (v) => {
                            resolve(v)
                        },
                        (r) => {
                            reject(r)
                        }
                    )
                } else {
                    // 结果的对象状态为 【成功】
                    resolve(result)
                }
            } catch (e) {
                reject(e)
            }
        }
        // 调用回调函数  根据 PromiseState 去调用
        if (this.PromiseState === 'fulfilled') {
            setTimeout(() => {
                callback(onResolved)
            })
        }
        if (this.PromiseState === 'rejected') {
            setTimeout(() => {
                callback(onRejected)
            })
        }
        // 判断 pending 状态
        if (this.PromiseState === 'pending') {
            // 保存回调函数
            this.callbacks.push({
                onResolved: function () {
                    callback(onResolved)
                },
                onRejected: function () {
```

```
                    callback(onRejected)
                },
            })
        }
    })
}

// 添加 catch 方法
Promise.prototype.catch = function (onRejected) {
    return this.then(undefined, onRejected)
}

// 添加 resolve 方法
Promise.resolve = function (value) {
    return new Promise((resolve, reject) => {
        if (value instanceof Promise) {
            value.then(
                (v) => {
                    resolve(v)
                },
                (r) => {
                    reject(r)
                }
            )
        } else {
            // 状态设置为成功
            resolve(value)
        }
    })
}

// 添加 reject 方法
Promise.reject = function (reason) {
    return new Promise((resolve, reject) => {
        reject(reason)
    })
}

// 添加 all 方法
Promise.all = function (promises) {
    // 声明变量
    let count = 0 // 计数
    let arr = [] // 结果数组
    // 遍历
    return new Promise((resolve, reject) => {
        for (let i = 0; i < promises.length; i++) {
            promises[i].then(
                (v) => {
                    // 得知对象的状态是成功
                    // 每个promise对象成功都加 1
                    count++
                    // 将当前每个promise对象成功的结果都存入到数组中
                    arr[i] = v
                    // 判断
                    if (count === promises.length) {
                        // 修改状态
                        resolve(arr)
                    }
```

```
                },
                (r) => {
                    reject(r)
                }
            )
        }
    })
}

// 添加 race 方法
Promise.race = function (promises) {
    return new Promise((resolve, reject) => {
        for (var i = 0; i < promises.length; i++) {
            promises[i].then(
                (v) => {
                    // 修改返回对象的状态为成功
                    resolve(v)
                },
                (r) => {
                    // 修改返回对象的状态为成功
                    reject(r)
                }
            )
        }
    })
}

```

1.5.2. 类的方式（封装成class）

```
/*
自定义 Promise
 */

// 封装成类
class Promise {
    //构造方法
    constructor(executor) {
        // 添加属性
        this.PromiseState = 'pending'
        this.PromiseResult = null
        // 声明属性  因为实例对象不能直接调用onResolve跟onReject 所以下面then中需要先保存在callback里面
        this.callbacks = []
        // 保存实例对象的 this 的值
        const self = this //  常见的变量名有self _this that

        // resolve 函数
        function resolve(data) {
            // 判断状态
            if (self.PromiseState !== 'pending') return
            // console.log(this)  => 这里的this指向window，下面用this的话时直接修改的window
            // 1. 修改对象的状态 (PromiseState)
            self.PromiseState = 'fulfilled'
            // 2. 设置对象结果值 (PromiseResult)
            self.PromiseResult = data
```

```
            // 调用成功的回调函数
            setTimeout(() => {
                self.callbacks.forEach((item) => {
                    item.onResolved(data)
                })
            })
        }

        // reject 函数
        function reject(data) {
            // 判断状态
            if (self.PromiseState !== 'pending') return
            // 1. 修改对象的状态 (PromiseState)
            self.PromiseState = 'rejected'
            // 2. 设置对象结果值 (PromiseResult)
            self.PromiseResult = data
            // 调用失败的回调函数
            setTimeout(() => {
                self.callbacks.forEach((item) => {
                    item.onRejected(data)
                })
            })
        }
        try {
            // 同步调用【执行器函数】
            executor(resolve, reject)
        } catch (e) {
            // 修改 promise 对象状态
            reject(e)
        }
    }

    // then 方法封装
    then(onResolved, onRejected) {
        const self = this
        // 判断回调函数参数
        if (typeof onRejected !== 'function') {
            onRejected = (reason) => {
                throw reason
            }
        }
        if (typeof onResolved !== 'function') {
            onResolved = (value) => value
        }
        return new Promise((resolve, reject) => {
            // 封装函数
            function callback(type) {
                try {
                    // 获取回调函数的执行结果
                    let result = type(self.PromiseResult)
                    // 判断
                    if (result instanceof Promise) {
                        result.then(
                            (v) => {
                                resolve(v)
                            },
                            (r) => {
                                reject(r)
```

```
                            }
                        )
                    } else {
                        // 结果的对象状态为 【成功】
                        resolve(result)
                    }
                } catch (e) {
                    reject(e)
                }
            }
            // 调用回调函数  根据 PromiseState 去调用
            if (this.PromiseState === 'fulfilled') {
                setTimeout(() => {
                    callback(onResolved)
                })
            }
            if (this.PromiseState === 'rejected') {
                setTimeout(() => {
                    callback(onRejected)
                })
            }
            // 判断 pending 状态
            if (this.PromiseState === 'pending') {
                // 保存回调函数
                this.callbacks.push({
                    onResolved: function () {
                        callback(onResolved)
                    },
                    onRejected: function () {
                        callback(onRejected)
                    },
                })
            }
        })
    }

    // catch 方法
    catch(onRejected) {
        return this.then(undefined, onRejected)
    }

    // resolve 方法
    static resolve(value) {
        return new Promise((resolve, reject) => {
            if (value instanceof Promise) {
                value.then(
                    (v) => {
                        resolve(v)
                    },
                    (r) => {
                        reject(r)
                    }
                )
            } else {
                // 状态设置为成功
                resolve(value)
            }
        })
```

```
    }

    // reject 方法
    static reject(reason) {
        return new Promise((resolve, reject) => {
            reject(reason)
        })
    }

    // all 方法
    static all(promises) {
        // 声明变量
        let count = 0 // 计数
        let arr = [] // 结果数组
        // 遍历
        return new Promise((resolve, reject) => {
            for (let i = 0; i < promises.length; i++) {
                promises[i].then(
                    (v) => {
                        // 得知对象的状态是成功
                        // 每个promise对象成功都加 1
                        count++
                        // 将当前每个promise对象成功的结果都存入到数组中
                        arr[i] = v
                        // 判断
                        if (count === promises.length) {
                            // 修改状态
                            resolve(arr)
                        }
                    },
                    (r) => {
                        reject(r)
                    }
                )
            }
        })
    }

    //race 方法
    static race(promises) {
        return new Promise((resolve, reject) => {
            for (var i = 0; i < promises.length; i++) {
                promises[i].then(
                    (v) => {
                        // 修改返回对象的状态为成功
                        resolve(v)
                    },
                    (r) => {
                        // 修改返回对象的状态为成功
                        reject(r)
                    }
                )
            }
        })
    }
}

```

1.6. async 与 await

1.6.1. 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

1.6.2. 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

1.6.3. await 表达式

1. await 右侧的表达式一般为 promise 对象，但也可以时其它的值
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

1.6.4. 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 try...catch 捕获处理

1.6.5. async 与 await 结合

```
// resource 1.html 2.html 3.html

const fs = require('fs')

// 回调函数的方式
fs.readFile('./resousrce/1.html', (err, data1) => {
  if (err) throw err
  fs.readFile('./resousrce/2.html', (err, data2) => {
    if (err) throw err
    fs.readFile('./resousrce/3.html', (err, data3) => {
        if (err) throw err
      console.log(data1 + data2 + data3)
        })
    })
})
```

```
// resource 1.html 2.html 3.html

const fs = require('fs')
const util = require('util')
const mineReadFile = util.pomiseify(fs.readFile)

// async 与 await 结合
async function main() {
  try {
    // 读取第一个文件的内容
    let data1 = await mineReadFile('./resourse/1.html')
        let data2 = await mineReadFile('./resourse/2.html')
    let data3 = await mineReadFile('./resourse/3.html')
```

```
    console.log(data1 + data2 + data3)
  }catch(e) {
    console.log(e)
  }
}
main()
```

```
// async 与 await 结合发送 Ajax 请求
function sendAJAX(url) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest()
    // 设置响应数据格式
    xhr.responseType = 'json'
    xhr.open('GET', url)
    xhr.send();
    // 处理结果
    xhr.onreadystatechange = function () {
      if (xhr.readyState === 4) {
        // 判断成功
        if (xhr.status >= 200 && xhr.status < 300) {
          // 成功的结果
          resolve(xhr.response)
        } else {
          reject(xhr.status)
        }
      }
    }
  })
}

// 段子接口地址：https://api.apiopen.top/getJoke
let btn = document.querySelector('#btn')

btn.addEventListener('click', async function () {
  // 获取段子信息
  let duanzi = await sendAJAX('https://api.apiopen.top/getJoke')
  console.log(duanzi)
})
```