Promise从入门到自定义

第1章：准备

1.1. 区别实例对象与函数对象

- 实例对象: new 函数产生的对象, 称为实例对象, 简称为对象
- 函数对象: 将函数作为对象使用时, 简称为函数对象

```
function Fn() { // Fn是函数

}
const fn = new Fn() // Fn是构造函数   new返回的是实例对象
console.log(Fn.prototype) // Fn是函数对象
Fn.call({}) // Fn是函数对象

$('#test') // $是函数
$.ajax() // $是函数对象
/*
总结
  1. 点的左边是对象(可能是实例对象也可能是函数对象)
  2. ()的左边是函数
*/
```

1.2. 二种类型的回调函数

1.2.1. 同步回调

- 理解: 立即执行, 完全执行完了才结束, 不会放入回调队列中
- 例子: 数组遍历相关的回调函数 / Promise的excutor函数

1.2.2. 异步回调

- 理解: 不会立即执行, 会放入回调队列中将来执行
- 例子: 定时器回调 / ajax回调 / Promise的成功|失败的回调

```
// 1. 同步回调函数
const arr = [1, 2, 3]
arr.forEach(item => {// 同步执行的回调函数, forEach()内部在执行完所有回调函数后才结束, 不会放入回调队列中
  console.log('forEach callback()', item)
})
console.log('forEach()之后')

// 2. 异步回调函数
setTimeout(() => { // 异步执行的回调函数, setTimeout()在回调函数执行前就结束了, 回调会放入回调队列中在同步代码执行完后才会执行
  console.log('setTimeout callback()')
}, 0)
console.log('setTimeout()之后')
```

1.3. JS的error处理

1.3.1. 错误的类型

- Error: 所有错误的父类型
- ReferenceError: 引用的变量不存在
- TypeError: 数据类型不正确的错误
- RangeError: 数据值不在其所允许的范围内

- SyntaxError: 语法错误

1.3.2. 错误处理

- 捕获错误: try ... catch
- 抛出错误: throw error

1.3.3. error对象的结构

- message属性: 错误相关信息
- stack属性: 函数调用栈记录信息

```
/*
1. 常见内置错误
*/
// ReferenceError: 引用的变量不存在
// console.log(a) // ReferenceError: a is not define
d

// TypeError: 数据类型不正确的错误
var b = null
// console.log(b.xxx) // TypeError: Cannot read prop
erty 'xxx' of null
b = 3
// console.log(b.xxx())  // TypeError: b.xxx is not
a function

// RangeError: 数据值不在其所允许的范围内
function fn1() {
  fn1()
}
// fn1() // RangeError: Maximum call stack size exce
eded
// SyntaxError: 语法错误
// const c = """  // SyntaxError: Unexpected string
/*
```

```
. 错误处理
*/
// 2.1 捕获错误
try {
  var d = 3
  d()
} catch (error) {
  console.log(error.message)
  console.log(error.stack)
}
console.log('捕获错误后还可以继续执行')
// 2.2 抛出错误
function doThing() {
  const time = Date.now()
  if (time %2 === 1) {
    console.log('当前是奇数, 可以执行业务逻辑处理')
  } else {
    throw new Error('当前时间是偶数, 无法处理业务逻辑')
  }
}
try {
    doThing()
} catch (error) { // 捕获错误, 做相应的界面提示
  alert(error.message)
}
```

第2章：Promise的理解和使用

2.1. Promise是什么?

2.1.1. 理解

2.抽象表达:

- 1) Promise是一门新的技术(ES6规范)

- 2) 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:

- 1) 从语法上来说: Promise是一个构造函数
- 2) 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

2.1.2. promise的状态改变

- pending变为resolved
- pending变为rejected
- 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

2.1.3. promise的基本流程

2.1.4. promise的基本使用

1) 使用1: 基本编码流程

```
<script>

 // 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)
  }
 )
</script>
```

2) 使用2: 使用promise封装基于定时器的异步

```
<script>
 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)
```

```
  },

 )

</script>
```

3) 使用3: 使用promise封装ajax异步请求

```
<script>


 //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)
  }
```

```
  )

</script>
```

2.2. 为什么要用Promise?

2.2.1. 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)

2.2.2. 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

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

2.3. 如何使用Promise?

2.3.1. API

1.Promise构造函数: Promise (excutor) {}

- (1) executor函数: 执行器 (resolve, reject) => {}
- (2) resolve函数: 内部定义成功时我们调用的函数 value => {}
- (3) 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) => {}

- (1) onResolved函数: 成功的回调函数 (value) => {}
- (2) 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)}
```

```
)
```

2.3.2. promise的几个关键问题

1.如何改变promise的状态?

- (1) resolve(value): 如果当前是pending就会变为fulfilled
- (2) reject(reason): 如果当前是pending就会变为rejected
- (3) 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?

- (1) 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
- (2) 如何先改状态再指定回调?
  - ① 在执行器中直接调用resolve()/reject()
  - ② 延迟更长时间才调用then()
- (3) 什么时候才能得到数据?
  - ① 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
  - ② 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?

- (1) 简单表达: 由then()指定的回调函数执行的结果决定
- (2) 详细表达:

    - ① 如果抛出异常, 新promise变为rejected, reason为抛出的异常
    - ② 如果返回的是非promise的任意值, 新promise变为resolved, value为返回的值
    - ③ 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?

- (1) promise的then()返回一个新的promise, 可以开成then()的链式调用
- (2) 通过then的链式调用串连多个同步/异步任务

6.promise异常传透?

- (1) 当使用promise的then链式调用时, 可以在最后指定失败的回调,
- (2) 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?

- (1) 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
- (2) 办法: 在回调函数中返回一个pendding状态的promise对象