

Alt text

**Promise的含义**

Promise是异步编程的一种解决方案，比传统的解决方案（回调函数和事件）更合理更强大。

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件 (通常是一个异步操作)的结果。从语法上说，Promise是一个对象，从它可以获取异步操作的消息。

**为什么要用promise?**

如果说 当我们需要发送多个异步请求 并且每个请求之间需要相互依赖 那这时 我们只能 以嵌套方式来解决 形成 "回调地狱"

Promise对象有以下2个特点：

1.对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态：Pending(进行中)、Resolved(已完成)和Rejected(已失败)。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是Promise这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

2.一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从Pending变为Resolved；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对象田静回调函数，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。

有了Promise对象，就可以把异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供了统一的接口，使得控制异步操作更加容易。

**、Promise的作用**

在ajax请求数据的过程中，我们可以异步拿到我们想要的数据，然后在回调中做相应的数据处理。

这样做看上去并没有什么麻烦，但是如果这个时候，我们还需要做另外一个ajax请求，这个新的ajax请求的其中一个参数，得从上一个ajax请求中获取，这个时候我们就需要在回调函数中再写一个异步请求，然后在这个异步函数的回调函数里在写相应的数据处理。要是连续嵌套个三四层，往往就很恶心了。

写起来就像下面这样：

$.ajax({
type:'get',
url:'url\_1',
data: 'data'
success : function(res){
//相应的数据处理
var data = res.data
$.ajax({
type:'get',
url:'url\_2',
data: data
success : function(res){
//相应的数据处理
$.ajax({
type:'get',
url:'url\_3',
data: data
success : function(res){
//相应的数据处理
}
})
}
})
}
})

代码逻辑书写顺序与执行顺序不一致，不利于阅读与维护。

异步操作的顺序变更时，需要大规模的代码重构。

回调函数基本都是匿名函数，bug 追踪困难。

回调函数是被第三方库代码（如上例中的 ajax ）而非自己的业务代码所调用的，造成了 IoC 控制反转。

**3.Promise使用**

**1.Promise 是一个构造函数， new Promise 返回一个 promise对象 接收一个excutor执行函数作为参数, excutor有两个函数类型形参resolve reject**

**const promise = new Promise((resolve, reject) => {
*// 异步处理*
*// 处理结束后、调用resolve 或 reject*
});**

**2.promise相当于一个状态机**

promise的三种状态

pending

fulfilled

rejected

1.promise 对象初始化状态为 pending

2.当调用resolve(成功)，会由pending => fulfilled

3.当调用reject(失败)，会由pending => rejected

注意promsie状态 只能由 pending => fulfilled/rejected, 一旦修改就不能再变

**3.promise对象方法**

1.then方法注册 当resolve(成功)/reject(失败)的回调函数

// onFulfilled 是用来接收promise成功的值
// onRejected 是用来接收promise失败的原因
promise.then(onFulfilled, onRejected);

then方法是异步执行的

2.resolve(成功) onFulfilled会被调用

3.reject(失败) onRejected会被调用

注：通过回调函数传参

**实例**

const promise = new Promise((resolve, reject) => {

resolve('fulfilled'); // 状态由 pending => fulfilled

});

promise.then(result => { // onFulfilled

console.log(result); // 'fulfilled' }, reason => { // onRejected 不会被调用 })

promise.then方法每次调用 都返回一个新的promise对象 所以可以链式写法

**5.Promise的静态方法**

Promise.resolve('hello');
*// 相当于*
const promise = new Promise(resolve => {
resolve('hello');
});

3.Promise.all 接收一个promise对象数组为参数

只有全部为resolve才会调用 通常会用来处理 多个并行异步操作

面试题总结：1

const promise = new Promise((resolve, reject) => {
console.log(1);
resolve();
console.log(2);
});
promise.then(() => {
console.log(3);
});
console.log(4);

copycode.gif

复制代码

输出结果为：1，2，4，3。

　　解题思路：then方法是异步执行的。

copycode.gif

复制代码

const promise = new Promise((resolve, reject) => {
setTimeout(() => {
resolve('success')
reject('error')
}, 1000)
})
promise.then((res)=>{
console.log(res)
},(err)=>{
console.log(err)
})

copycode.gif

复制代码

输出结果：success

　　解题思路：Promise状态一旦改变，无法在发生变更。

3、

Promise.resolve(1)
.then(2)
.then(Promise.resolve(3))
.then(console.log)

输出结果：1

　　解题思路：Promise的then方法的参数期望是函数，传入非函数则会发生值穿透。

4、

copycode.gif

复制代码

setTimeout(()=>{
console.log('setTimeout')
})
let p1 = new Promise((resolve)=>{
console.log('Promise1')
resolve('Promise2')
})
p1.then((res)=>{
console.log(res)
})
console.log(1)

copycode.gif

复制代码

输出结果：

　　　　Promise1

　　　　1

　　　　Promise2

　　　　setTimeout

　　解题思路：这个牵扯到js的执行队列问题，整个script代码，放在了macrotask queue中，执行到setTimeout时会新建一个macrotask queue。但是，promise.then放到了另一个任务队列**microtask queue**中。script的执行引擎会取1个macrotask queue中的task，执行之。然后把所有**microtask queue**顺序执行完，再取setTimeout所在的macrotask queue按顺序开始执行。

const promise = new Promise((resolve, reject) => {
setTimeout(() => {
console.log('开始');
resolve('success');
}, 5000);
});
const start = Date.now();
promise.then((res) => {
console.log(res, Date.now() - start);
});
promise.then((res) => {
console.log(res, Date.now() - start);
});

copycode.gif

复制代码

输出结果：

　　　　开始

　　　　success 5002

　　　　success 5002

　　解题思路：promise 的**.then**或者**.catch**可以被调用多次，但这里 Promise 构造函数只执行一次。或者说 promise 内部状态一经改变，并且有了一个值，那么后续每次调用**.then**或者**.catch**都会直接拿到该值。

Promise的先进之处在于可以在then方法中继续写Promise对象并返回。

自己实现一个promise

function Promise(fn){

var status='pending'

function successNotify(){

status='fulfilled'//状态为fufilled

toDothen.apply(undefined,arguments)//执行回调

}

function failNotify

**源码**

**// 判断变量否为function
const isFunction = variable => typeof variable === 'function'
// 定义Promise的三种状态常量
const PENDING = 'PENDING'
const FULFILLED = 'FULFILLED'
const REJECTED = 'REJECTED'
class MyPromise {
constructor (handle) {
if (!isFunction(handle)) {
throw new Error('MyPromise must accept a function as a parameter')
}
// 添加状态
this.\_status = PENDING
// 添加状态
this.\_value = undefined
// 添加成功回调函数队列
this.\_fulfilledQueues = []
// 添加失败回调函数队列
this.\_rejectedQueues = []
// 执行handle
try {
handle(this.\_resolve.bind(this), this.\_reject.bind(this))
} catch (err) {
this.\_reject(err)
}
}
// 添加resovle时执行的函数
\_resolve (val) {
const run = () => {
if (this.\_status !== PENDING) return
// 依次执行成功队列中的函数，并清空队列
const runFulfilled = (value) => {
let cb;
while (cb = this.\_fulfilledQueues.shift()) {
cb(value)
}
}
// 依次执行失败队列中的函数，并清空队列
const runRejected = (error) => {
let cb;
while (cb = this.\_rejectedQueues.shift()) {
cb(error)
}
}
/\* 如果resolve的参数为Promise对象，则必须等待该Promise对象状态改变后,
当前Promsie的状态才会改变，且状态取决于参数Promsie对象的状态
\*/
if (val instanceof MyPromise) {
val.then(value => {
this.\_value = value
this.\_status = FULFILLED
runFulfilled(value)
}, err => {
this.\_value = err
this.\_status = REJECTED
runRejected(err)
})
} else {
this.\_value = val
this.\_status = FULFILLED
runFulfilled(val)
}
}
// 为了支持同步的Promise，这里采用异步调用
setTimeout(run, 0)
}
// 添加reject时执行的函数
\_reject (err) {
if (this.\_status !== PENDING) return
// 依次执行失败队列中的函数，并清空队列
const run = () => {
this.\_status = REJECTED
this.\_value = err
let cb;
while (cb = this.\_rejectedQueues.shift()) {
cb(err)
}
}
// 为了支持同步的Promise，这里采用异步调用
setTimeout(run, 0)
}
// 添加then方法
then (onFulfilled, onRejected) {
const { \_value, \_status } = this
// 返回一个新的Promise对象
return new MyPromise((onFulfilledNext, onRejectedNext) => {
// 封装一个成功时执行的函数
let fulfilled = value => {
try {
if (!isFunction(onFulfilled)) {
onFulfilledNext(value)
} else {
let res = onFulfilled(value);
if (res instanceof MyPromise) {
// 如果当前回调函数返回MyPromise对象，必须等待其状态改变后在执行下一个回调
res.then(onFulfilledNext, onRejectedNext)
} else {
//否则会将返回结果直接作为参数，传入下一个then的回调函数，并立即执行下一个then的回调函数
onFulfilledNext(res)
}
}
} catch (err) {
// 如果函数执行出错，新的Promise对象的状态为失败
onRejectedNext(err)
}
}
// 封装一个失败时执行的函数
let rejected = error => {
try {
if (!isFunction(onRejected)) {
onRejectedNext(error)
} else {
let res = onRejected(error);
if (res instanceof MyPromise) {
// 如果当前回调函数返回MyPromise对象，必须等待其状态改变后在执行下一个回调
res.then(onFulfilledNext, onRejectedNext)
} else {
//否则会将返回结果直接作为参数，传入下一个then的回调函数，并立即执行下一个then的回调函数
onFulfilledNext(res)
}
}
} catch (err) {
// 如果函数执行出错，新的Promise对象的状态为失败
onRejectedNext(err)
}
}
switch (\_status) {
// 当状态为pending时，将then方法回调函数加入执行队列等待执行
case PENDING:
this.\_fulfilledQueues.push(fulfilled)
this.\_rejectedQueues.push(rejected)
break
// 当状态已经改变时，立即执行对应的回调函数
case FULFILLED:
fulfilled(\_value)
break
case REJECTED:
rejected(\_value)
break
}
})
}
// 添加catch方法
catch (onRejected) {
return this.then(undefined, onRejected)
}
// 添加静态resolve方法
static resolve (value) {
// 如果参数是MyPromise实例，直接返回这个实例
if (value instanceof MyPromise) return value
return new MyPromise(resolve => resolve(value))
}
// 添加静态reject方法
static reject (value) {
return new MyPromise((resolve ,reject) => reject(value))
}
// 添加静态all方法
static all (list) {
return new MyPromise((resolve, reject) => {
/\*\*
\* 返回值的集合
\*/
let values = []
let count = 0
for (let [i, p] of list.entries()) {
// 数组参数如果不是MyPromise实例，先调用MyPromise.resolve
this.resolve(p).then(res => {
values[i] = res
count++
// 所有状态都变成fulfilled时返回的MyPromise状态就变成fulfilled
if (count === list.length) resolve(values)
}, err => {
// 有一个被rejected时返回的MyPromise状态就变成rejected
reject(err)
})
}
})
}
// 添加静态race方法
static race (list) {
return new MyPromise((resolve, reject) => {
for (let p of list) {
// 只要有一个实例率先改变状态，新的MyPromise的状态就跟着改变
this.resolve(p).then(res => {
resolve(res)
}, err => {
reject(err)
})
}
})
}
finally (cb) {
return this.then(
value => MyPromise.resolve(cb()).then(() => value),
reason => MyPromise.resolve(cb()).then(() => { throw reason })
);
}
}**