确认以下几个Promise特点：

Promise中有三种状态：pendding(进行中)、fulfilled(已完成)、rejected(已失败)

Promise状态一旦确定，将无法再改变

# 第一步：覆盖原生的Promise

function Promise(executor){

function resolve(){

}

function reject(){

}

executor(resolve,reject)

}

# 第二步：原生的Promise对象上有内置的属性:[[ PromiseState ]]和[[PromiseResult]]

function Promise(executor){

this.PromiseState = “pedding”

this.PromiseResult = null

// resolve函数和reject函数中this指向window,所以需要传递一个this给它们

const self = this

function resolve(data){

self.PromiseState = “fulfilled”

self.PromiseResult = data

}

function reject(data){

self.PromiseState = “rejected”

self.PromiseResult = data

}

executor(resolve,reject)

}

# 第三步：原生Promise特性 一旦状态确定就无法再改变

function Promise(executor){

this.PromiseState = “pedding”

this.PromiseResult = null

// resolve函数和reject函数中this指向window,所以需要传递一个this给它们

const self = this

function resolve(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “fulfilled”

self.PromiseResult = data

}

function reject(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “rejected”

self.PromiseResult = data

}

executor(resolve,reject)

}

# 第四步：改变promise状态的方式有三种:resolve(),reject(),throw “error”(抛出异常)

function Promise(executor){

this.PromiseState = “pedding”

this.PromiseResult = null

// resolve函数和reject函数中this指向window,所以需要传递一个this给它们

const self = this

function resolve(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “fulfilled”

self.PromiseResult = data

}

function reject(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “rejected”

self.PromiseResult = data

}

try{

executor(resolve,reject)

}catch(e){

reject(e)

}

}

# 第五步：手写then方法

function Promise(executor){

this.PromiseState = “pedding”

this.PromiseResult = null

// resolve函数和reject函数中this指向window,所以需要传递一个this给它们

const self = this

// 定义一个对象,保存then中的成功和失败的回调函数

const callback = {}

function resolve(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “fulfilled”

self.PromiseResult = data

if(self.callback.onResolved){

self.callback.onResolved(data)

}

}

function reject(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “rejected”

self.PromiseResult = data

if(self.callback.onRejected){

self.callback.onRejected(data)

}

}

try{

executor(resolve,reject)

}catch(e){

reject(e)

}

}

// 【then方法】

Promise.prototype.then = (onResolved,onRejected)=>{

// 根据Promise实例对象中返回的结果,调用then中不同的回调

if(this.Promisestate=== “fulfilled”){

onResolved(this.PromiseResult)

}

if(this.Promisestate=== “rejected”){

onRejected(this.PromiseResult)

}

// 如果Promise中是异步操作,按照代码执行顺序,会直接先指定then方法,此时Promise的状态还是pendding

if(this.Promisestate=== “pedding”){

// 等到Promise中的异步代码执行完毕,Promise的状态确定了,就会执行then方法.

this.callback = {

onResolved,

onRejected

}

}

}

# 第六步：在Promise中使用异步操作前提下。then方法可能同时调用多次,都要返回对应的结果,不能后一次的结果覆盖前一次的

function Promise(executor){

this.PromiseState = “pedding”

this.PromiseResult = null

// resolve函数和reject函数中this指向window,所以需要传递一个this给它们

const self = this

// 定义一个数组,保存then中的成功和失败的回调函数对象

const callback = []

function resolve(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “fulfilled”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onResolved(data)

})

}

function reject(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “rejected”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onRejected(data)

})

}

try{

executor(resolve,reject)

}catch(e){

reject(e)

}

}

// 【then方法】

Promise.prototype.then = (onResolved,onRejected)=>{

// 根据Promise实例对象中返回的结果,调用then中不同的回调

if(this.Promisestate=== “fulfilled”){

onResolved(this.PromiseResult)

}

if(this.Promisestate=== “rejected”){

onRejected(this.PromiseResult)

}

// 如果Promise中是异步的方法,按照代码执行顺序,会直接先指定then方法,此时Promise的状态还是pendding

if(this.Promisestate=== “pedding”){

// 等到Promise中的异步代码执行完毕,Promise的状态确定了,就会执行then方法.

this.callback.push({

onResolved,

onRejected

})

}

}

# 第七步：在Promise同步任务下，then方法返回结果的实现

then方法的返回结果由回调函数的结果决定

如果返回的是非promise对象,那么Promise状态就是成功的（resolved）

如果返回的是Promise对象

状态是rejected或throw “error”,那么就是失败的

状态是resolved,就是成功的

function Promise(executor){

this.PromiseState = “pedding”

this.PromiseResult = null

// resolve函数和reject函数中this指向window,所以需要传递一个this给它们

const self = this

// 定义一个数组,保存then中的成功和失败的回调函数对象

const callback = []

function resolve(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “fulfilled”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onResolved(data)

})

}

function reject(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “rejected”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onRejected(data)

})

}

try{

executor(resolve,reject)

}catch(e){

reject(e)

}

}

// 【then方法】

Promise.prototype.then = (onResolved,onRejected)=>{

return new Promise((resolve,reject)=>{

// 根据Promise实例对象中返回的结果,调用then中不同的回调

if(this.Promisestate=== “fulfilled”){

// 对返回的结果进行判断

let result = onResolved(this.PromiseResult)

if(result instanceof Promise){

result.then(res=>{

resolve(res)

},err=>{

reject(err)

})

}else{

// 如果不是Promise对象,状态都为成功

resolve(result)

}

}

if(this.Promisestate=== “rejected”){

// 对返回的结果进行判断

let result = onRejected(this.PromiseResult)

if(result instanceof Promise){

result.then(res=>{

resolve(res)

},err=>{

reject(err)

})

}else{

// 如果不是Promise对象,状态都为成功

resolve(result)

}

}

// 如果Promise中是异步的方法,按照代码执行顺序,会直接先指定 then方法,此时Promise的状态还是pendding

if(this.Promisestate=== “pedding”){

// 等到Promise中的异步代码执行完毕,Promise的状态确定了,就会执行then方法.

this.callback.push({

onResolved,

onRejected

})

}

})

}

# 第八步：在Promise异步任务下，then方法返回结果的实现

function Promise(executor){

this.PromiseState = “pedding”

this.PromiseResult = null

// resolve函数和reject函数中this指向window,所以需要传递一个this给它们

const self = this

// 定义一个数组,保存then中的成功和失败的回调函数对象

const callback = []

function resolve(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “fulfilled”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onResolved(data)

})

}

function reject(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “rejected”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onRejected(data)

})

}

try{

executor(resolve,reject)

}catch(e){

reject(e)

}

}

// 【then方法】

Promise.prototype.then = (onResolved,onRejected)=>{

const self = this

return new Promise((resolve,reject)=>{

// 根据Promise实例对象中返回的结果,调用then中不同的回调

if(this.Promisestate=== “fulfilled”){

// 对返回的结果进行判断

let result = onResolved(this.PromiseResult)

if(result instanceof Promise){

result.then(res=>{

resolve(res)

},err=>{

reject(err)

})

}else{

// 如果不是Promise对象,状态都为成功

resolve(result)

}

}

if(this.Promisestate=== “rejected”){

// 对返回的结果进行判断

let result = onRejected(this.PromiseResult)

if(result instanceof Promise){

result.then(res=>{

resolve(res)

},err=>{

reject(err)

})

}else{

// 如果不是Promise对象,状态都为成功

resolve(result)

}

}

// 如果Promise中是异步的方法,按照代码执行顺序,会直接先指定 then方法,此时Promise的状态还是pendding

if(this.Promisestate=== “pedding”){

// 等到Promise中的异步代码执行完毕,Promise的状态确定了,就会执行then方法.

this.callback.push({

onResolved:function(){

// 对返回的结果进行判断

// 注意:内部的this指向onResolve(),所以需要保存外部的this,在内部使用

let result = onResolved(self.PromiseResult)

if(result instanceof Promise){

result.then(res=>{

resolve(res)

},err=>{

reject(err)

})

}else{

// 如果不是Promise对象,状态都为成功

resolve(result)

}

}

onRejected:function(){

// 对返回的结果进行判断

let result = onRejected(self.PromiseResult)

if(result instanceof Promise){

result.then(res=>{

resolve(res)

},err=>{

reject(err)

})

}else{

// 如果不是Promise对象,状态都为成功

resolve(result)

}

}

})

}

})

}

# 第九步：then方法完善与优化

function Promise(executor){

this.PromiseState = “pedding”

this.PromiseResult = null

// resolve函数和reject函数中this指向window,所以需要传递一个this给它们

const self = this

// 定义一个数组,保存then中的成功和失败的回调函数对象

const callback = []

function resolve(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “fulfilled”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onResolved(data)

})

}

function reject(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “rejected”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onRejected(data)

})

}

try{

executor(resolve,reject)

}catch(e){

reject(e)

}

}

// 【then方法】

Promise.prototype.then = (onResolved,onRejected)=>{

const self = this

// 封装函数

function callback(type){

// 对返回的结果进行判断

let result = type(self .PromiseResult)

// 返回的结果是Promise对象

if(result instanceof Promise){

result.then(res=>{

resolve(res)

},err=>{

reject(err)

})

}else{

// 如果不是Promise对象,状态都为成功

resolve(result)

}

}

return new Promise((resolve,reject)=>{

// 根据Promise实例对象中返回的结果,调用then中不同的回调

// 成功的回调

if(this.Promisestate=== “fulfilled”){

callback(onResolved)

}

// 失败的回调

if(this.Promisestate=== “rejected”){

callback(onRejected)

}

// 异步的成功与失败回调

// 如果Promise中是异步的方法,按照代码执行顺序,会直接先指定 then方法,此时Promise的状态还是pendding

if(this.Promisestate=== “pedding”){

// 等到Promise中的异步代码执行完毕,Promise的状态确定了, 就会执行then方法.

this.callback.push({

onResolved:function(){

callback(onResolved)

},

onRejected:function(){

callback(onRejected)

}

})

}

})

}

# 第十步：完成catch方法

function Promise(executor){

this.PromiseState = “pedding”

this.PromiseResult = null

// resolve函数和reject函数中this指向window,所以需要传递一个this给它们

const self = this

// 定义一个数组,保存then中的成功和失败的回调函数对象

const callback = []

function resolve(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “fulfilled”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onResolved(data)

})

}

function reject(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “rejected”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onRejected(data)

})

}

try{

executor(resolve,reject)

}catch(e){

reject(e)

}

}

// 【then方法】

Promise.prototype.then = (onResolved,onRejected)=>{

const self = this

// catch方法可以链式穿透和值传递

// 意思就是

If( typeof onRejected !== “function” ){

onRejected = reason => { throw reason }

}

if( typeof onResolved !== “function”){

onResolved = value => value // 值传递

}

// 封装函数

function callback(type){

// 对返回的结果进行判断

let result = type(self .PromiseResult)

// 返回的结果是Promise对象

if(result instanceof Promise){

result.then(res=>{

resolve(res)

},err=>{

reject(err)

})

}else{

// 如果不是Promise对象,状态都为成功

resolve(result)

}

}

return new Promise((resolve,reject)=>{

// 根据Promise实例对象中返回的结果,调用then中不同的回调

// 成功的回调

if(this.Promisestate=== “fulfilled”){

callback(onResolved)

}

// 失败的回调

if(this.Promisestate=== “rejected”){

callback(onRejected)

}

// 异步的成功与失败回调

// 如果Promise中是异步的方法,按照代码执行顺序,会直接先指定 then方法,此时Promise的状态还是pendding

if(this.Promisestate=== “pedding”){

// 等到Promise中的异步代码执行完毕,Promise的状态确定了, 就会执行then方法.

this.callback.push({

onResolved:function(){

callback(onResolved)

},

onRejected:function(){

callback(onRejected)

}

})

}

})

}

// 【catch方法】

Promise.prototype.catch = function(onRejected){

// catch方法返回的是一个Promise对象,可以使用已有的then方法处理

return this.then(undefined,onRejected)

}

# 第十一步：完成resolve,reject方法

function Promise(executor){

this.PromiseState = “pedding”

this.PromiseResult = null

// resolve函数和reject函数中this指向window,所以需要传递一个this给它们

const self = this

// 定义一个数组,保存then中的成功和失败的回调函数对象

const callback = []

function resolve(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “fulfilled”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onResolved(data)

})

}

function reject(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “rejected”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onRejected(data)

})

}

try{

executor(resolve,reject)

}catch(e){

reject(e)

}

}

// 【then方法】

Promise.prototype.then = (onResolved,onRejected)=>{

const self = this

// catch方法可以链式穿透和值传递

// 意思就是

If( typeof onRejected !== “function” ){

onRejected = reason => { throw reason }

}

if( typeof onResolved !== “function”){

onResolved = value => value // 值传递

}

// 封装函数

function callback(type){

// 对返回的结果进行判断

let result = type(self .PromiseResult)

// 返回的结果是Promise对象

if(result instanceof Promise){

result.then(res=>{

resolve(res)

},err=>{

reject(err)

})

}else{

// 如果不是Promise对象,状态都为成功

resolve(result)

}

}

return new Promise((resolve,reject)=>{

// 根据Promise实例对象中返回的结果,调用then中不同的回调

// 成功的回调

if(this.Promisestate=== “fulfilled”){

callback(onResolved)

}

// 失败的回调

if(this.Promisestate=== “rejected”){

callback(onRejected)

}

// 异步的成功与失败回调

// 如果Promise中是异步的方法,按照代码执行顺序,会直接先指定 then方法,此时Promise的状态还是pendding

if(this.Promisestate=== “pedding”){

// 等到Promise中的异步代码执行完毕,Promise的状态确定了, 就会执行then方法.

this.callback.push({

onResolved:function(){

callback(onResolved)

},

onRejected:function(){

callback(onRejected)

}

})

}

})

}

// 【catch方法】

Promise.prototype.catch = function(onRejected){

// catch方法返回的是一个Promise对象,可以使用已有的then方法处理

return this.then(undefined,onRejected)

}

// 【resolve方法】

Promise.resolve = function(data){

return new Promise((resolve,reject)=>{

If(data instanceof Promise ){

data.then(value=>{

resolve(value)

},reason=>{

reject(reason)

})

}else{

resolve(data)

}

})

}

// 【reject方法】

Promise.reject = function(data){

return new Promise((resolve,reject)=>{

reject(data)

})

}

# 第十二步：完成all、race方法

function Promise(executor){

this.PromiseState = “pedding”

this.PromiseResult = null

// resolve函数和reject函数中this指向window,所以需要传递一个this给它们

const self = this

// 定义一个数组,保存then中的成功和失败的回调函数对象

const callback = []

function resolve(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “fulfilled”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onResolved(data)

})

}

function reject(data){

if(self.PromiseState !==”pedding”) return

self.PromiseState = “rejected”

self.PromiseResult = data

self.callback.forEach(item=>{

item.onRejected(data)

})

}

try{

executor(resolve,reject)

}catch(e){

reject(e)

}

}

// 【then方法】

Promise.prototype.then = (onResolved,onRejected)=>{

const self = this

// catch方法可以链式穿透和值传递

// 意思就是

If( typeof onRejected !== “function” ){

onRejected = reason => { throw reason }

}

if( typeof onResolved !== “function”){

onResolved = value => value // 值传递

}

// 封装函数

function callback(type){

// 对返回的结果进行判断

let result = type(self .PromiseResult)

// 返回的结果是Promise对象

if(result instanceof Promise){

result.then(res=>{

resolve(res)

},err=>{

reject(err)

})

}else{

// 如果不是Promise对象,状态都为成功

resolve(result)

}

}

return new Promise((resolve,reject)=>{

// 根据Promise实例对象中返回的结果,调用then中不同的回调

// 成功的回调

if(this.Promisestate=== “fulfilled”){

callback(onResolved)

}

// 失败的回调

if(this.Promisestate=== “rejected”){

callback(onRejected)

}

// 异步的成功与失败回调

// 如果Promise中是异步的方法,按照代码执行顺序,会直接先指定 then方法,此时Promise的状态还是pendding

if(this.Promisestate=== “pedding”){

// 等到Promise中的异步代码执行完毕,Promise的状态确定了, 就会执行then方法.

this.callback.push({

onResolved:function(){

callback(onResolved)

},

onRejected:function(){

callback(onRejected)

}

})

}

})

}

// 【catch方法】

Promise.prototype.catch = function(onRejected){

// catch方法返回的是一个Promise对象,可以使用已有的then方法处理

return this.then(undefined,onRejected)

}

// 【resolve方法】

Promise.resolve = function(data){

return new Promise((resolve,reject)=>{

If(data instanceof Promise ){

data.then(value=>{

resolve(value)

},reason=>{

reject(reason)

})

}else{

resolve(data)

}

})

}

// 【reject方法】

Promise.reject = function(data){

return new Promise((resolve,reject)=>{

reject(data)

})

}

// 【all方法】

Promise.all = function(promiseArr){

return new Promise((resolve,reject)=>{

let count = 0

let arr = []

for(let i = 0;i<promiseArr.length;i++){

promiseArr[i].then(value=>{

count++

arr[i] = value // 不能用push,不然会打乱顺序

},reason=>{

reject(reason)

})

}

})

}

// 【race方法】

Promise.all = function(promiseArr){

return new Promise((resolve,reject)=>{

for(let i = 0;i<promiseArr.length;i++){

promiseArr[i].then(value=>{

resolve(value)

},reason=>{

reject(reason)

})

}

})

}