参考：https://github.com/dslu7733/promise/blob/master/promise

0.1. 两种回调函数

- 同步回调：立即执行，直到执行结束。不会放到回调队列。例子：数组遍历相关的回调函数，Promise的excutor函数
- 异步回调：不立即执行，而是放到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 回调函数就是一个通过函数指针调用的函数，通俗理解就是参数是函数的函数，进一步地，根据函数参数执行的先后顺序划分出了同步和异步回调

```
//同步回调
const arr = [1,2,3]
arr.forEach( item => {  //遍历回调
  console.log(item)     //不进入回调队列，立刻执行
} )
console.log('forEach之后')  //先输出1 2 3,再输出forEach之后


//异步回调
setTimeout( () => {
  console.log('timeout callback()') //这个函数进入回调队列排队
}, 0)
//总是执行完下面的代码，最后才执行回调队列里面的函数
console.log('setTimeout()之后') //先输出setTimeout()之后，最后才输出 timeout
callback()

//判断是同步还是异步，再最后加一个console.log
```

0.2. Error

参考MDN https://developer.mozilla.org/en-US/docs/Web/JavaScript

- 错误类型

- Error: 所有错误的父类型

1. ReferenceError：引用类型错误
2. TypeError： 数据类型错误
3. SyntaxError: 语法错误
4. RangeErrot: 数据值不在其所允许的范围内

- 错误处理
    1. 捕获错误：try ... catch
    2. 抛出错误: throw error
- 错误对象
    1. message属性： 错误相关信息
    2. stack属性： 函数调用栈记录信息

0.2.1. 常见内置错误

```
//ReferenceError: 引用变量不存在
console.log(a) //ReferenceError: a is not defined

//TypeError: 数据类型不正确
var b = {}
b.xxx()  //TypeError: b.xxx is not a function

//RangeError: 数据值不在其所允许的范围内
function fn(){
  fn()
}
fn()  //RangeError: Maximum call stack size exceeded


//SyntaxError: 语法错误
const c = """  //SyntaxError: Unexpected string

/*
 vs code好用快捷键
 Alt+Shift+上/下键 : 快速复制一行
 Alt++上/下键 : 快速移动一行
 ctrl+shift+k : 删除一行
 ctrl+` : 回到终端
*/
```

0.2.2. 错误处理

```
//错误捕获
try {
  let d
  console.log(d.xxx)
} catch (error) { //可以通过debugger查看error对象的属性
  console.log(error.message)
  console.log(error.stack) // 默认是console.log(error.stack)
}
// 可以继续向下执行
console.log('出错之后')

//抛出错误
function something() {
  if (Date.now() % 2  ===  1 ){
    console.log('当前时间为奇数,可执行任务')
  } else {
    throw new Error('当前时间为偶数无法执行任务') // 一般都抛出Error类型
  }
}

//情况1 直接抛出异常
something() //Error: 当前时间为偶数无法执行任务
console.log('something之后')  //不会继续这句代码（没有对异常进行处理）
```

```
//情况2 捕获处理异常
try {
  something()
} catch(error) {
  console.log(error.message)
}
```

0.3. 什么是Promise

- Promise是JS中进行异步编程的新的解决方案
    1. 语法上: Promise是一个构造函数
    2. 功能上: Promise对象用来封装一个异步操作并可以获得其结果
- Promise状态
    1. 初始化状态pending，未确定的
    2. pending变成功:fulfilled
    3. pending变失败:rejected
    4. 一个promise对象只能改变一次，无论成功还是失败,都会有一个结果数据(成功的数据value or 错误的结果数据reason)

```
then()
                                成功,执行resolve() > Promise对象(resolved状态) > 回调onResolved()
  new Promise() > 执行异步操作 {
} 新的Promise对象
                                失败,执行reject()  > Promise对象(rejected状态)  > 回调onRejected()

then()/catch()
```

0.4. Promise基本使用

1. 创建一个新的promise对象
2. 执行异步操作任务
3. 3.1 成功,调用resolve(value) ; 3.2 失败,调用reject(reason)
4. then(value => { //接收得到成功的value}, reason => { //接收失败得到的reason})

```
//1.创建一个新的promise对象
//尽量使用const变量
const p = new Promise((resolve, reject) => { //执行器函数是同步回调!
  console.log('执行 excutor') //立刻执行
  //2.执行异步操作任务
  setTimeout(() => {
    const time = Date.now()
    //3.1 成功,调用resolve(value)
    if( time % 2 === 0 ){
```

```
      resolve('成功的数据,time=' + time)
    } else {
    //3.2 失败,调用reject(reason)
      reject('失败的数据,time=' + time)
    }
  }, 1000)
})
console.log('new Promise()之后')  //先输出执行 excutor再输出new Promise()之后

p.then(
  value => { //接收得到成功的value, onResolved()
    console.log('成功的回调', value)
  },
  reason => { //接收失败得到的reason, onRejected()
    console.log('失败的回调', reason)
  }
)
```

0.5. 为什么要用Promise

对比不同回调方式(伪代码)

1. 指定回调方式更加灵活，就是时间问题 - 旧的: 必须在启动异步任务前指定回调函数 - 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)
2. 支持链式调用,解决回调地狱 - 什么是回调地狱： 回调函数嵌套使用，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 / async await

```
function successCallback(result) {
  console.log('声音文件创建成功' + result)
}

function failureCallback(error) {
  console.log('声音文件创建失败' + error)
}

/* 1.1 纯回调函数 */
//启动任务(audioSettings)前必须指定回调函数(callback)
createAudioFileAsync(audioSettings, successCallback, failureCallback)

/* 1.2 promise */
//可在启动任务(audioSettings)后指定回调函数(callback)
const promise = createAudioFileAsync(audioSettings)
setTimeout(() => {
  promise.then(successCallback, failureCallback)
}, 1000)



/* 2.1 回调地狱 */
//回调函数的嵌套,串联执行，第二个的异步任务执行要以第一个的结果作为条件，第三个的异步任务
```

```
执行是以第二个的结果作为条件的
doSomething(function (result) { //第一个函数function就是sucessCallback
  doSomethingElse(result, function (newResult) {
    doThirdThing(newResult, function (finalResult) {
      console.log('Got the final result' + finalResult)
    }, failureCallback)
  }, failureCallback)
}, failureCallback)


/* 2.2 链式调用 */
doSomething().then(function(result) { //result是doSomething函数成功执行的返回值
  return doSomethingElse(result)      //执行器函数,同步回调
})
.then(function(newResult){  //newResult是doSomethingElse成功执行的返回值
  return doThirdThing(newResult)
})
.then(function(finalResult){
  console.log('Got the final result' + finalResult)
})
.catch(failureCallback) //无论是哪一部分的异常，最后异常穿透到统一的错误处理


/* 2.3 async/await : 回调地狱的终极解决方案 */
//根本上去掉回调函数
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)
  }
}
```

0.6. Promise的API

1. Promise构造函数： Promise(exscutor) { }

- executor函数（执行器函数）：同步执行 (resolve, reject) => {}
- resolve函数： 内部定义成功时调用resolve函数来将pending状态改成fulfilled value => {}
- reject函数： 内部定义失败时调用reject函数来将pending状态改为rejected reason => {}
- 说明： executor函数会在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： 指定用于一个成功value的成功回调和一个失败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: 成功的数据或Promise对象
- 返回一个成功/失败的promise对象

5. Promise.reject()方法

- reason: 失败的原因
- 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，其中一个失败就失败

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) => {
  setTimeout( () => {
    resolve('成功') //resolve就像是一个传递数据的运输机
  }, 1000 )
})
.then(
  value => {
    console.log('onResolved()1', value)
  }
)
.catch(
  reason => {
    console.log('onRejected()1', reason)
  }
)
// 产生一个成功值为1的promise对象
const p1 = new Promise((resolve, reject) => {
  resolve(1)
})
const p2 = Promise.resolve(2) // 和上面一句结果是一样
const p3 = Promise.reject(3)
// p1.then( value => {console.log(value)} )
// p2.then( value => {console.log(value)} )
// p3.catch( reason => {console.log(reason)} )

//const pAll = Promise.all([p1,p2,p3])
const pAll = Promise.all([p1,p2])
pAll.then(
  values => {
    console.log('all onResolved()', values) // 结果all onResolved() [1,2] values顺
```

```
序要和传入all()的数组一致，和完成的先后顺序没关系
  },
  reason => {
    console.log('all onRejected()', reason)  // const pAll =
Promise.all([p1,p2,p3])，结果all onRejected() 3
  }
)


const pRace = Promise.race([p1,p2,p3])
pRace.then(
  value => {
    console.log('race onResolved()', value)
  },
  reason => {
    console.log('race onResolved()', reason)
  }
)
```

0.7. promise的几个关键问题

0.7.1. error属于promise哪个状态

```
const p = new Promise((resolve, reject)=>{
  // throw new Error('出错了') //抛出异常， promse变为 rejected失败状态,reason为抛出
的error
  throw 3 // 地出异常,promise变为rejected状态， reason为抛出的3
})

p.then(
  value => {},
  reason => { console.log('reason', reason) } //reason Error: 出错了
)
```

0.7.2. 一个promise指定多个成功/失败回调函数

当promise改变为对应状态时都会调用

```
const p2 = new Promise((resolve, reject)=>{
  throw new Error('出错了') //属于reject状态
})

p2.then(
  value => {},
  reason1 => { console.log('reason1', reason1) } //reason1 Error: 出错了
).then(
  reason2 => { console.log('reason2', reason2) }  //reason2 undefined
)
```

0.7.3. 状态改变与指定回调函数的先后次序

- 都有可能，正常情况下是先指定回调再改变状态，但也可先改变状态再指定回调
- 如何先改变状态再指定回调？
  - 在执行器中直接调用resolve()/reject()
  - 延迟更长时间才调用then()
- 什么时候才能的得到数据？
  - 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

```
//常规：先指定回调函数，后改变的状态
new Promise((resolve, reject)=>{
  setTimeout(()=>{
    resolve(1)  //后改变状态(同时指定数据),异步执行回调函数
  },1000)
}).then(  //先指定回调函数,保存当前指定的回调函数
  value => {},
  reason => { console.log('reason', reason) }
)

//如何向改变状态，再指定回调函数
new Promise((resolve, reject)=>{
    resolve(1)  //先改变状态(同时指定数据)
}).then(  //后指定回调函数,异步执行回调函数
  value => { console.log('value', value) },
  reason => { console.log('reason', reason) }
)
console.log('-----')  //先输出----, 再输出value 1
```

0.7.4. promise.then()返回的新promise的结果状态由什么决定(重点)

- 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

```
new Promise((resolve, reject) => {
  resolve(1)
}).then(
  value => {
    console.log("onResolved()", value) // onResolved() 1
  },
  reason => {
    console.log("onRejected()", reason)
  }
).then(
  value => {
    console.log("onResolved1()", value) // undefined 因为第一个.then执行的value函数
```

```
它执行成功没有返回值，结果为undefined，因此第二个.then里的value为undefined
  },
  reason => {
    console.log("onRejected2()", reason)
  }
)
/*------------------------------*/
new Promise((resolve, reject) => {
  resolve(1)
}).then(
  value => {
    console.log("onResolved()", value) // onResolved() 1
    // return 2 // 如果返回的是非 promise的任意值，新 promise变为resolved，value为返回的值
    // return Promise.resolve(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
    // return Promise.reject(4)
    throw 5 // 如果抛出异常，新 promise变为 rejected， reason为抛出的异常
  },
  reason => {
    console.log("onRejected()", reason)
  }
).then(
  value => {
    console.log("onResolved1()", value) // 2  // 3
  },
  reason => {
    console.log("onRejected2()", reason) // 4 // 5
  }
)
```

0.7.5. 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

```
new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('执行任务1(异步)')
    resolve(1)
  }, 1000)
}).then(
  value => {
    console.log('任务1的结果:', value)
    console.log('执行任务2(同步)')
    // 返回同步
    return 2
  }
).then(
  value => {
    console.log('任务2的结果:', value)
```

```
    // 返回异步
    return new Promise((resolve, reject) => {
      //启动任务3(异步)
      setTimeout(() => {
        console.log('执行任务3(异步)')
        resolve(3)
      }, 1000)
    })
  }
).then(
  value => {
    console.log('任务3的结果:', value)
  }
)
/*
执行任务1(异步)
任务1的结果: 1
执行任务2(同步)
任务2的结果: 2
执行任务3(异步)
任务3的结果: 3
*/
new Promise((resolve, reject) => {
  reject(1)
}).then(
  value => {
    console.log("onResolved1()", value)
    throw 5
  },
  reason => {
    console.log("onRejected1()", reason)
  }
).then(
  value => {
    console.log("onResolved2()", value)
  },
  reason => {
    console.log("onRejected2()", reason)
  }
)
/*
onRejected1() 1
onResolved2() undefined
*/
```

0.7.6. 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
- 前面任何操作出现了异常，都会传到最后失败的回调中处理

```
new Promise((resolve, reject) => {
  //resolve(1)
  reject(1)
}).then(
  value => {
    console.log('onResolved1()', value)
    return 2
  }
).then(
  value => {
    console.log('onResolved2()', value)
    return 3
  }
).then(
  value => {
    console.log('onResolved3()', value)
  }
).catch(reason => { // 上面的then里都没有reason函数，相当于每个then隐藏了reason => {throw reason}，一直传递下去直到找到处理reason的函数
  console.log('onRejected1()', reason)
})
// reason => {throw reason}的{}不能省略，因为=>同时还包含return的意思，但是throw前面不能有return
```

0.7.7. 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

```
new Promise((resolve, reject) => {
  reject(1)
}).then(
  value => {
    console.log('onResolved1()', value)
    return 2
  }
).then(
  value => {
    console.log('onResolved2()', value)
    return 3
  }
).then(
  value => {
    console.log('onResolved3()', value)
  }
).catch(reason => {
  console.log('onRejected1()', reason)
  return new Promise(()=>{})  //返回一个pending的promise 中断promise链
}).then(
  value => { console.log('onResolved4()', value) },
```

```
  reason => { console.log('onRejected4()', reason)}
)
/*
onRejected1() 1
*/
```

0.8. async function和await expression

1. async function (function return Promise)

- async函数返回Promise对象
- promise对象的结果由async函数执行的返回值决定

2. await expression (value or Promise)

- expression一般是Promise对象，也可以是其他值
- 如果是Promise对象，await返回的是Promise成功的值
- 如果是其他值，直接将此值作为await的返回值

3. await必须写在async中，但async可以没有await

- 如果await的Promise失败，就会抛出异常，需通过try...catch...捕获处理

async函数返回一个promise对象,async函数返回的promise的结果是由函数执行的结果决定

```
async function fn1() {
  //return 1
  //throw 2
  //return Promise.resolve(3)
  return new Promise((resolve, reject)=>{
    setTimeout(()=>{
      resolve(4)
    }, 1000)
  })
}

const result = fn1()
//console.log(result) //Promise { 1 }

result.then(
  value => {
    console.log('onResolved()', value)
  },
  reason => {
    console.log('onRejected()', reason)
  }
)


/*------------------------*/
```

```
function fn2(){
  return new Promise((resolve, reject)=>{
    setTimeout(()=>{
      //resolve(4)
      reject(6)
    }, 1000)
  })
}

async function fn4(){
  return 5
}

async function fn3(){
  try {
    //const value = await fn2() //await右侧表达式如果是Promise，得到的结果就是promise成功的value
    const value = await fn4() //await右侧表达式如果不是Promise，得到的结果就是它本身，因为异步函数返回的结果Promise成功了,成功的结果就是1，才会返回1
    console.log('value', value)
  } catch {
    console.log('得到失败的结果', error)
  }

}

fn3()

async function fn5(){
  try{
    const value = await fn2()
    console.log('fn5 value', value)
  }catch(error){  //捕获失败promise的结果
    console.log('fn5 error', error)
  }
}

fn5()
```

0.9. 宏队列和微队列

- 宏队列： dom事件回调，ajax回调，定时器回调
- 微队列： promise回调，mutationObserver回调
- 步骤：

1. JS引擎首先先执行所有初始化同步任务代码
2. 每次准备去出第一个宏任务执行前，都要将所有的微任务一个个取出来执行

```
setTimeout(() => {
  console.log("timeout callback1()")
},0)
```

```
setTimeout(() => {
  console.log("timeout callback2()")
},0)
Promise.resolve(1).then(
  value => {
    console.log("Promise callback1()", value)
  }
)
Promise.resolve(2).then(
  value => {
    console.log("Promise callback2()", value)
  }
)
/*
Promise callback1() 1
test.js:9
Promise callback2() 2
test.js:14
timeout callback1()
test.js:2
timeout callback2()
*/
setTimeout(() => {
  console.log("timeout callback1()")
  Promise.resolve(1).then(
  value => {
    console.log("Promise callback3()", value)
  }
)
},0)
setTimeout(() => {
  console.log("timeout callback2()")
},0)
Promise.resolve(1).then(
  value => {
    console.log("Promise callback1()", value)
  }
)
Promise.resolve(2).then(
  value => {
    console.log("Promise callback2()", value)
  }
)
/*
Promise callback1() 1
Promise callback2() 2
timeout callback1()
Promise callback3() 1 // 每次做宏任务之前，先将微任务完成
timeout callback2()
 * /
```

0.10. 面试题1

考点： 同步->微队列->宏队列

```
setTimeout(() =>{
  console.log(1)
})
Promise.resolve().then(() => {
  console.log(2)
})
Promise.resolve().then(() => {
  console.log(3)
})
console.log(4)
/*4231*/
```

0.11. 面试题2

```
setTimeout(() => {
  console.log(1)
})
new Promise((resolve) => {
  console.log(2) // 这是同步的
  resolve()
}).then(() => {
  console.log(3)
}).then(() => {
  console.log(4)
})
console.log(5)
/*25341,3执行了才放4进微队列*/
```

0.12. 面试题3

```
const first = () => (new Promise((resolve, reject) => {
  console.log(3) // 同步
  let p = new Promise((resolve, reject) => {
    console.log(7) // 同步
    setTimeout(() => {
      console.log(5)
      resolve(6) // 此时p已经修改过状态了，不是pending状态不能再修改，因此这句话不打印
    }, 0)
    resolve(1) // p改变状态
  })
  resolve(2) // first改变状态
  p.then((arg) => { // p的回调先产生
    console.log(arg)
  })
}))
```

```
first().then((arg) => { // first的回调后产生
  console.log(arg)
})
console.log(4) // 同步
/*374125
宏:  [5]
微:  [1,2]
*/
```

0.13. 面试题4

```
setTimeout(() => {
  console.log("0")
}, 0)
new Promise((resolve, reject) => {
  console.log("1")
  resolve()
}).then(() => {
  console.log("2")
  new Promise((resolve, reject) => {
    console.log("3")
    resolve()
  }).then(() => {
    console.log("4")
  }).then(() => {
    console.log("5")
  })
}).then(() => {
  console.log("6")
})

new Promise((resolve, reject) => {
  console.log("7")
  resolve()
}).then(() => {
  console.log("8")
})
/*172384650
17
宏:  [0]
微: [2,8]
1723
宏:  [0]
微: [8,4]
console.log("2")
  new Promise((resolve, reject) => {
    console.log("3") // 同步
    resolve()
  }).then(() => { // 放进微队列
    console.log("4")
  }).then(() => { // 4还在微队列还没有执行，不能放5进微队列，但此时这语句已经结束，整段
```

```
返回undefined
    console.log("5")
  })
1723
宏： [0]
微: [8,4,6]
() => {
  console.log("6") // 6进入微队列
}
17238
宏： [0]
微: [4,6]
172384
宏： [0]
微: [6，5]// 4执行完就放5进微队列
172384650
*/
```