一、Promise -核心逻辑实现

```
// Promise 就是一个类，在执行这个类的时候 需要传递一个执行器进去 执行器会立即执行
// Promise中有三种状态，分别为成功 fulfilled 失败 rejected 等待 pending
// pending -> fulfilled
// pending -> rejected
// 一旦状态确定就不可更改
// resolve和reject函数是用来更改状态的
// resolve：fulfilled
// reject：rejected
// then方法内部做的事情就是判断状态 如果状态是成功 调用成功的回调函数 如果状态是失败 调用失败回调函数 then方法是被定义在原型对象中
// then成功回调有一个参数 表示成功之后的值 then失败回调有一个参数，表示失败后的原因
// then方法是可以被链式调用的，后面then方法的回调函数拿到值的是上一个then方法的回调函数的返回值

const PENDING = 'pending'; // 等待
const FULFILLED = 'fulfilled'; //成功
const REJECTED = 'rejected'; // 失败

class MyPromise {
 constructor (executor) {
 try {
 executor(this.resolve, this.reject)
 } catch (e) {
 this.reject(e);
 }
 }
 //promise状态
 status = PENDING;
 // 成功之后的值
 value = undefined;
 // 失败后的原因
 reason = undefined;
```

```
31  // 成功回调
32  successCallback = []
33  // 失败回调
34  failCallback = []
35
36  resolve = value => {
37  // 如果状态不是等待 阻止程序向下执行
38  if (this.status !== PENDING) return;
39  //将状态更改为成功
40  this.status = FULFILLED
41  // 保存成功之后的值
42  this.value = value;
43  // 判断成功回调是否存在 如果存在 调用
44  // this.successCallback && this.successCallback(this.value);
45  while(this.successCallback.length)
this.successCallback.shift()();
46  }
47  reject = reason => {
48  // 如果状态不是等待 阻止程序向下执行
49  if (this.status !== PENDING) return;
50  //将状态更改为失败
51  this.status = REJECTED
52  // 保存失败后的原因
53  this.reason = reason
54  // 判断失败回调是否存在 如果存在 调用
55  // this.failCalback && this.failCalback(this.value);
56  while(this.failCalback.length) this.failCalback.shift()();
57  }
58  then (successCallbak, failCalback) {
59  successCallback = successCallback ? successCallback :value =>
value;
60  failCalback = failCalback ? failCalback :reason => { throw rea
son };
61  let promise2 = new MyPromise((resolve, reject) => {
62  //判断状态
63  if (this.status === FULFILLED) {
```

```
64  setTimeout(() => {
65  try {
66  let x = successCallbak(this.value)
67  // 判断x的值是普通值还是promise对象
68  // 如果是普通值，直接调用resolve
69  // 如果是promise对象 查看promise对象返回的结果
70  // 再根据promise对象返回的结果 决定调用resolve 还是调用reject
71  resolvePromise(promise2, x, resolve, reject);
72  } catch (e) {
73  reject(e)
74  }
75  }, 0)
76  } else if (this.status === REJECTED) {
77  setTimeout(() => {
78  try {
79  let x = failCalback(this.reason)
80  // 判断x的值是普通值还是promise对象
81  // 如果是普通值，直接调用resolve
82  // 如果是promise对象 查看promise对象返回的结果
83  // 再根据promise对象返回的结果 决定调用resolve 还是调用reject
84  resolvePromise(promise2, x, resolve, reject);
85  } catch (e) {
86  reject(e)
87  }
88  }, 0)
89  } else {
90  // 等待
91  // 将成功回调和失败回调存储起来
92  this.successCallbak.push(() = > {
93  setTimeout(() => {
94  try {
95  let x = successCallbak(this.reason)
96  // 判断x的值是普通值还是promise对象
97  // 如果是普通值，直接调用resolve
98  // 如果是promise对象 查看promise对象返回的结果
```

```
// 再根据promise对象返回的结果 决定调用resolve 还是调用reject
resolvePromise(promise2, x, resolve, reject);
} catch (e) {
reject(e)
}
})
this.failCalback.push(() => {
setTimeout(() => {
try {
let x = failCalback(this.reason)
// 判断x的值是普通值还是promise对象
// 如果是普通值，直接调用resolve
// 如果是promise对象 查看promise对象返回的结果
// 再根据promise对象返回的结果 决定调用resolve 还是调用reject
resolvePromise(promise2, x, resolve, reject);
} catch (e) {
reject(e)
}
}, 0)
});
}
});
return promise2;
}
finally (callback) {
return this.then(value => {
return MyPromise.resolve(callback()).then(() => value);
}, reason => {
return MyPromise.resolve(callback()).then(() => {throw
reason});
})
}
catch (failCallback) {
return this.then(undefined, failCallback)
}
```

```
133  static all (array) {
134  let result = []
135  let index = 0;
136  return new MyPromise((resolve, reject) => {
137  function addData (key, value) {
138  result[key] = value
139  index++;
140  if (index === array.length) {
141  resolve(result)
142  }
143  }
144  for (let i = 0; i < array.length; i++) {
145  let current = array[i];
146  if (current instanceof MyPromise) {
147  // promise 对象
148  current.then(value => addData(i, value), reason => reject(rea
on))
149  } else {
150  // 普通值
151  addData(i, array[i])
152  }
153  }
154  resolve(result)
155  })
156  }
157  static resolve (value) {
158  if (value instanceof MyPromise) return value;
159  return new MyPromise(resolve => resolve(value))
160  }
161 }
162
163 function resolvePromise(promise2, x, resolve, reject) {
164  if (promise2 === x) {
165  reject(new TypeError('Chaining cycle detected for promise #<P
omise>'))
166  }
```

```
 if (x instanceof MyPromise) {
 // promise 对象
 // x.then(value => resolve(value), reason => reject(reason))
 x.then(resolve, reject)
 } else {
 // 普通值
 resolve(x);
 }
}

module.exports = MyPromise;

const MyPromise = requeir('./myPromise')
let promise = new MyPromise((resolve, reject) => {
 setTimeout(() => {
 resolve('成功')
 }, 2000)
 throw new Error('executor error')
 resolve('成功')
 reject('失败')
})

function other () {
 return new MyPromise((resolve, reject) => {
 resolve('other');
 });
}

promise.then(value => {
 console.log(value)
 throw new Error('then error');
}, reason => {
 console.log(value)
})

```

```
let p1 = promise.then(value => {
 console.log(value);
 return p1;
})
p1.then(value => {
 console.log(value);
 return 'aaa';
}, reason => {
 console.log(reason.message)
 return 10000;
}).then(() => {
 console.log(value)
})
```

二、在Promise类中加入异步逻辑

看上

三、实现then方法多次调用添加多个处理函数

看上上

四、实现then方法的链式调用（一）

五、实现then方法的链式调用（二）

六、then 方法链式调用识别Promise对象自返回

七、捕获错误及then链式调用其他状态代码补充

八、将then方法的参数变成可选参数

九、Promise.all方法

```
function p1 () {
```

```
 return new Promise(function (resolve, reject) {
 setTimeout(function () {
 resolve('p1')
 }, 2000)
 })
}

function p2 () {
  return new Promise(function (resolve, reject) {
  resolve('p2')
  })
}
Promise.all(['a', 'b', p1(), p2(), 'c']).then(function (result)
{
  //result -> ['a', 'b', 'p1', 'p2', 'c']
})
```

十、Promise.resolve方法的实现

```
function p1 () {
 return new Promise(function (resolve, reject) {
 resolve('hello');
 })
}
MyPromise.resolve(10).then(value => console.log(value));
MyPromise.resolve(p1()).then(value => console.log(value));
```

十一、finally方法的实现

```
function p1 () {
 return new Promise(function (resolve, reject) {
 setTimeout(function () {
 resolve('p1')
 }, 2000)
 })
```

```
}

function p2 () {
  return new Promise(function (resolve, reject) {
  resolve('p2')
  })
}

p2().finally(() => {
  console.log('finally')
  return p1()
}).then (value => {
  console.log(value)
}, reason => {
  console.log(reason)
})
```

十二、catch方法的实现

```
function p1 () {
 return new Promise(function (resolve, reject) {
 setTimeout(function () {
 resolve('p1')
 }, 2000)
 })
}

function p2 () {
  return new Promise(function (resolve, reject) {
  reject('p2')
  })
}

p2()
```

```
16  .then(value => console.log(value))
17  .catch(reson => console.log(reason))
```