1.首先，毫无疑问，Promise()是一个构造函数， 并且，存在3种状态，pending， fulfilled(也可以叫Resolved)， rejected。分别表示等待时，成功时，失败时
2.Promise实例化时，传了个参数，并且这个参数是个函数（并且是个立即执行函数），同时这个函数还有两个参数，且这两个参数，依然是函数，分别是resolve()，reject()
3.then()函数中的第一个参数（后文我们统称为then的resolve回调），是在调用then()方法的Promise对象的状态变为fulfilled时被执行的，而第二个参数（后文我们统称为rejected回调），是在Promise对象的状态变成rejected时被调用的。
4.通过第四代，我们还可以看出，在resolve()函数执行时，是将Promise对象的状态变更成了fulfilled,从而触发了then的resolve回调函数的执行。而reject()函数执行时，是将Promise对象的状态变更成了rejected，从而触发了then的reject回调的执行
5.resolve(res)时的值，就是then的resolve回调的参数，reject(err)的值就是thenreject回调的参数
6.then()函数和catch()函数可以被链式调用

Promise基础雏形

```
//定义了一个状态，执行了立即执行函数。并将resolve，reject传入到立即执行函数中。
class MyPromise {
    constructor (fun) {
         // 定义初始状态(3个状态分别是pending，fulfilled，rejected)
        this.status = 'pending'
        // 定义两个变量分别来存储成功时值和失败时的值
        this.resolveValue = null
        this.rejectValue = null

        // 定义resolve函数
        let resolve = (val) => {
            // 1、将状态变更为fulfilled，但是注意一点，Promise是有个特点的，就是状态只能由pending状态变更为fulfilled或者由pending状态变更为rejected。且，状态变化后，不会再变化。故，我们需要先判断当前是否是等待状态pending
            if (this.status === 'pending') {  // this指向实例化出来Promise对象
                this.status = 'fulfilled'
                // 2、保存resolve时的值，以便后面调用then()方法时使用
                this.resolveValue = val
            }
        }
        // 定义reject函数
        let reject = (val) => {
            // 1、状态变更为rejected
            if (this.status === 'pending') {
                this.status = 'rejected'
                // 2、保存reject()时的值
                this.rejectValue = val
            }
        }

        try {
            // 执行函数
            console.log("执行函数",fun,"resolve-->:",resolve)
            console.log("====reject-->:",reject);
            fun(resolve, reject)
        } catch (err) {
            reject(err)
        }
```

```
    }
    //步骤2
    then = (onFullFilled, onRejected) => {
        // onFullFilled, onRejected分别resolve()时的回调函数和reject()时的回调函数
        // 此时，判断状态，不同状态时，分别执行不同的回调
        if (this.status === 'fulfilled') {
            onFullFilled(this.resolveValue)
        }
        if (this.status === 'rejected') {
            onRejected(this.rejectValue )
        }
        // 上面的this指向的是调用then的promise实例，故可以直接拿到状态和返回值
    }
}
//测试代码：
const promise1 = new MyPromise((resolve, reject) => {
    resolve(123)
})
promise1.then((res) => {
    console.log(res)  // 123
}, (err) => {
    console.log(err)
})
/*
123
*/
```

此时，resolve, reject已经有了，new Promise()时，已经可以调用resolve()方法和reject()方法了。并且，resolve()和reject()时，promise状态也已经发生了改变。并保存了resolve和reject出来的值。

步骤2：在下一步，状态已经发生改变了，我们是不是要触发then的resolve回调，或者reject回调了。所以，我们来实现then()函数

```
MyPromise.prototype.then = (onFullFilled, onRejected) => {
    // onFullFilled, onRejected分别resolve()时的回调函数和reject()时的回调函数
    // 此时，判断状态，不同状态时，分别执行不同的回调
    if (this.status === 'fulfilled') {
        onFullFilled(this.resolveValue)
    }
    if (this.status === 'rejected') {
        onRejected(this.rejectValue )
    }
    // 上面的this指向的是调用then的promise实例，故可以直接拿到状态和返回值
}
```

步骤3，then()函数的完善

到此时，只是完成了基本,我们在then函数中，只判断了状态为fufilled时，调了onFullFilled，状态为rejected时，调了onRejected。但如果then()函数被调用时，promise的状态还并未发生改变（也就是还处于pending时），那then()函数内的代码是不是不会执行拉。因为我们并没有写pending状态时的处理代码。如以下情况

```
const promise1 = new MyPromise((resolve, reject) => {
    setTimeout(() => {
        resolve(123)
    }, 0)
})
promise1.then((res) => {
```

```
    console.log(res)
}, (err) => {
    console.log(err)
})
/*
没有输出

const promise1 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(123)
    }, 0)
})
promise1.then((res) => {
    console.log("输出2",res)
}, (err) => {
    console.log(err)
})
正常应该输出：输出2 123
*/

1、先定义两个变量用来保存then()的回调函数
this.onFullFilledList = []
this.onRejectedList = []

2.then()执行时，如果状态还未发生改变（还是pending时），那么就将回调函数先保存起来
MyPromise.prototype.then = function (onFullFilled, onRejected) {
    // onFulfilled, onRejected分别resolve()时的回调函数和reject()时的回调函数
    // 此时，判断状态，不同状态时，分别执行不同的回调
    if (this.status === 'fulfilled') {
        onFullFilled(this.resolveValue)
    }
    if (this.status === 'rejected') {
        onRejected(this.rejectValue )
    }
    if (this.status === 'pending') {
        // 保存的是一个函数，而函数内是回调的执行代码，当我们执行被保存的函数时，函数内的
onFullFilled和onRejected是不是也就跟着执行拉
        this.onFullFilledList.push(() => {
            onFullFilled(this.resolveValue)
        })
        this.onRejectedList.push(() => {
            onRejected(this.rejectValue )
        })
    }
    // 上面的this指向的是调用then的promise实例，故可以直接拿到状态和返回值
}

3、在resolve()和reject()的时候， 去取onFullFilledList，onRejectedList两个队列中的函数，
并依次执行
// 定义resolve函数
let resolve = (val) => {
     // 1、将状态变更为fulfilled，但是注意一点，Promise是有个特点的，就是状态只能由pending
状态变更为fulfilled或者由pending状态变更为rejected。且，状态变化后，不会再变化。故，我们需要
先判断当前是否是等待状态pending
     if (this.status === 'pending') {  // this指向实例化出来Promise对象
         this.status = 'fulfilled'
         // 2、保存resolve时的值，以便后面调用then()方法时使用
         this.resolveValue = val
```

```
            // 执行then的resolve回调
            this.onFullFilledList.forEach(funItem => funItem())
        }
}

// 定义reject函数
let reject = (val) => {
        // 1、状态变更为rejected
        if (this.status === 'pending') {
            this.status = 'rejected'
            // 2、保存reject()时的值
            this.rejectValue = val

            // 执行then的reject回调
            this.onRejectedList.forEach(funItem => funItem())
        }
}
```

到此，就不会再有因为异步代码而执行不了的问题了。看下完整代码，并验证下

```
class MyPromise {
    constructor (fun) {
            // 定义初始状态(3个状态分别是pending, fulfilled, rejected)
        this.status = 'pending'
        // 定义两个变量分别来存储成功时值和失败时的值
        this.resolveValue = null
        this.rejectValue = null
        //在resolve()和reject()的时候，去取onFullFilledList，onRejectedList两个队列中的函数，并依次执行
        this.onFullFilledList = []
        this.onRejectedList = []


        // 定义resolve函数
        let resolve = (val) => {
            // 1、将状态变更为fulfilled，但是注意一点，Promise是有个特点的，就是状态只能由pending状态变更为fulfilled或者由pending状态变更为rejected。且，状态变化后，不会再变化。故，我们需要先判断当前是否是等待状态pending
            if (this.status === 'pending') {  // this指向实例化出来Promise对象
                this.status = 'fulfilled'
                // 2、保存resolve时的值，以便后面调用then()方法时使用
                this.resolveValue = val

                // 执行then的resolve回调
                this.onFullFilledList.forEach(funItem => funItem())
            }
        }
        // 定义reject函数
        let reject = (val) => {
            // 1、状态变更为rejected
            if (this.status === 'pending') {
                this.status = 'rejected'
                // 2、保存reject()时的值
                this.rejectValue = val
```

```
                // 执行then的reject回调
                this.onRejectedList.forEach(funItem => funItem())
            }
        }

        try {
            // 执行函数
            console.log("执行函数",fun)
            // console.log("resolve-->:",resolve)
            // console.log("====reject-->:",reject);
            fun(resolve, reject)
        } catch (err) {
            reject(err)
        }
    }
    //步骤2
    then = (onFullFilled, onRejected) => {
        // onFullFilled, onRejected分别resolve()时的回调函数和reject()时的回调函数
        // 此时，判断状态，不同状态时，分别执行不同的回调
        if (this.status === 'fulfilled') {
            onFullFilled(this.resolveValue)
        }
        if (this.status === 'rejected') {
            onRejected(this.rejectValue )
        }

        //then()执行时，如果状态还未发生改变（还是pending时），那么就将回调函数先保存起来
        if (this.status === 'pending') {
            // 保存的是一个函数，而函数内是回调的执行代码，当我们执行被保存的函数时，函数内的
onFullFilled和onRejected是不是也就跟着执行拉
            this.onFullFilledList.push(() => {
                onFullFilled(this.resolveValue)
            })
            this.onRejectedList.push(() => {
                onRejected(this.rejectValue )
            })
        }
        // 上面的this指向的是调用then的promise实例，故可以直接拿到状态和返回值
    }
}
//测试代码
const promise1 = new MyPromise((resolve, reject) => {
    setTimeout(() => {
        resolve(123)
    }, 0)
})
promise1.then((res) => {
    console.log(res)
}, (err) => {
    console.log(err)
})
```

Promise.then()的链式调用

Promise之所以能够进行链式调用，是因为then()方法内部返回了一个Promise实例，而返回的这个Promise实例在继续调用了第二个then()方法。并且第二个then的resolve回调的参数，是上一个then的resolve回调函数的返回值。

我们来改造下then

```
MyPromise.prototype.then = function (onFullFilled, onRejected) {
    // 将then函数内部返回的Promise对象取名为promise2，后续文档中将直接以promise2来表示这个对象
    const promise2 = new MyPromise((resolve, reject) => {
        // onFulfilled resolve()时的回调函数, onRejected reject()时的回调函数
        // 此时，判断状态，不同状态时，分别执行不同的回调
        if (this.status === 'fulfilled') {
            // 定义一个变量来保存onFullFilled的返回值
            let result = onFullFilled(this.resolveValue)
            resolve(result)
        }
        if (this.status === 'rejected') {
            // 定义一个变量来保存onRejected的返回值
            let result = onRejected(this.rejectValue )
            reject(result)
        }
        if (this.status === 'pending') {
            this.onFullFilledList.push(() => {
                let result = onFullFilled(this.resolveValue)
                resolve(result)
            })
            this.onRejectedList.push(() => {
                let result = onRejected(this.rejectValue )
                resolve(result)
            })
        }
    })
}
```

可以看到，我们在调用then时，返回了一个新的Promise实例，并且将这个then(onFullFilled, onRejected)的resolve回调和reject回调的返回值resolve或者reject出去了。
这种方式，对于当onFullFilled返回的是一个普通值来说，是可行的，但如果onFullFilled返回的是一个Promise对象或者函数呢。
从原生Promise的功能上，我们是可以看出的:
1.当onFullFilled函数返回值是普通值时，下一个then的onFullFilled函数将会以这个返回值作为参数。
2.当onFullFilled函数返回值是一个函数时，下一个then的onFullFilled函数也会直接以这个函数当作参数

3.当onFullFilled函数返回值是一个Promise时，then()方法返回的Promise对象(下文统称为promise2)的状态就取决去这个onFullFilled函数所返回的Promise的状态。promise2 resolve()或者reject()的值，取决于onFullFilled函数返回的Prmise对象resolve()或者reject()的值
所以，此时，我们是不是需要一个函数来专门判断这个onFullFilled的返回值到底是普通值还是函数还是Promise对象。并且，当值不同时，处理方式就不一样。
如下:

```
// then函数中，返回值的处理函数，判断返回值的类型，并做处理
const formatPromise = (promise, result, resolve, reject) => {
```

```
    // 首先，先判断下promise是不是result，因为我们知道，我们的result是一个返回值，他可能是一个Promise，那如果他直接返回第一个参数中的promise的话，那么是会造成死循环的。
    if (promise === result) {
        return new Error('未知的result')
    }
    // 判断是否是对象
    if (typeof result === 'object' && result != null) {
        try {
            // 如果是对象，先看是否存在then函数
            let then = result.then
            // 如果result.then是一个函数，就说明是Promise对象，或者thenable对象
            if (typeof then === 'function') {
                // 利用.call将this指向result，防止result.then()报错
                then.call(result, res => {
                    resolve(res)
                }, err => {
                    reject(err)
                })
            } else {
                // 如果不是function,那么说明只是普通对象，并不是Promise对象，当普通值处理
                resolve(result)
            }
        } catch (err) {
            reject(err)
        }
    } else {
        // 不是对象，那就是普通值或者函数，直接resolve()
        resolve(result)
    }
}
MyPromise.prototype.then = function (onFullFilled, onRejected) {
    // 将then函数内部返回的Promise对象取名为promise2，后续文档中将直接以promise2来表示这个对象
    const promise2 = new MyPromise((resolve, reject) => {
        // onFulfilled, onRejected分别resolve()时的回调函数和reject()时的回调函数
        // 此时，判断状态，不同状态时，分别执行不同的回调
        if (this.status === 'fulfilled') {
            setTimeout(() => {
                try {
                    // 定义一个变量来保存onFullFilled的返回值
                    let result = onFullFilled(this.resolveValue)
                    formatPromise(promise2, result, resolve, reject)
                } catch (err) {
                    reject(err) // 捕捉上面代码执行的错误
                }
            }, 0) // 这里说明下为说明要用setTimeout，因为我这段代码要用到promise2，而如果是同步代码，promise2不可在自己的立即执行函数内调用自己
        }
        if (this.status === 'rejected') {
            setTimeout(() => {
                try {
                    // 定义一个变量来保存onRejected的返回值
                    let result = onRejected(this.rejectValue )
                    formatPromise(promise2, result, resolve, reject)
                } catch (err) {
                    reject(err) // 捕捉上面代码执行的错误
                }
            }, 0)
```

```
        }
        if (this.status === 'pending') {
            this.onFullFilledList.push(() => {
                setTimeout(() => {
                    try {
                        // 定义一个变量来保存onFullFilled的返回值
                        let result = onFullFilled(this.resolveValue)
                        formatPromise(promise2, result, resolve, reject)
                    } catch (err) {
                        reject(err) // 捕捉上面代码执行的错误
                    }
                }, 0)
            })
            this.onRejectedList.push(() => {
                setTimeout(() => {
                    try {
                        // 定义一个变量来保存onRejected的返回值
                        let result = onRejected(this.rejectValue )
                        formatPromise(promise2, result, resolve, reject)
                    } catch (err) {
                        reject(err) // 捕捉上面代码执行的错误
                    }
                }, 0)
            })
        }
    })
    return promise2
}
```

到这一步，我们已经可以处理return一个Promise对象时的情况了。我们来验证一下

```
const promise1 = new MyPromise((resolve, reject) => {
    setTimeout(() => {
        resolve(123)
    }, 0)
})
promise1.then((res) => {
    console.log(res)
    return new MyPromise((resolve, reject) => {
        resolve(234)
    })
}, (err) => {
    console.log(err)
}).then(res => {
    console.log(res)
})
// 输出
123
234
```

但此时，又引出一个问题了，上图中，return的Promise里面，如果我resolve的不是234，而是resolve了一个新的Promise呢看下面代码

```
const promise1 = new MyPromise((resolve, reject) => {
    setTimeout(() => {
        resolve(123)
    }, 0)
})
promise1.then((res) => {
    console.log(res)
    return new MyPromise((resolve, reject) => {
        const promise3 = new MyPromise((resolve, reject) => {
            resolve(234)
        })
        resolve(promise3)
    })
}, (err) => {
    console.log(err)
}).then(res => {
    console.log(res)
})
// 输出
123
MyPromise {
  status: 'fulfilled',
  resolveValue: 234,
  rejectValue: null,
  onFullFilledList: [],
  onRejectedList: []
}
```

可以看出，这个时候，就解析不出234了。更别说如果promise3内部resolve的又是一个Promise了。此时，如果出现这个层层嵌套的。我们是不是要进一步的去进行解析啊，直到解析出一个普通值。那么这个时候，我们是不是要用到递归啊，不断的利用fomatPromise去解析resolve的值，直到resolve的是一个普通值才停止。下面我们看代码，继续完善formatPromise

```
class MyPromise {
    constructor (fun) {
         // 定义初始状态(3个状态分别是pending, fulfilled, rejected)
        this.status = 'pending'
        // 定义两个变量分别来存储成功时值和失败时的值
        this.resolveValue = null
        this.rejectValue = null
        //在resolve()和reject()的时候，去取onFullFilledList，onRejectedList两个队列中的函数，并依次执行
        this.onFullFilledList = []
        this.onRejectedList = []
        // 定义resolve函数
        let resolve = (val) => {
            // 1、将状态变更为fulfilled，但是注意一点，Promise是有个特点的，就是状态只能由pending状态变更为fulfilled或者由pending状态变更为rejected。且，状态变化后，不会再变化。故，我们需要先判断当前是否是等待状态pending
            if (this.status === 'pending') {  // this指向实例化出来Promise对象
                this.status = 'fulfilled'
                // 2、保存resolve时的值，以便后面调用then()方法时使用
                this.resolveValue = val
```

```
                // 执行then的resolve回调
                this.onFullFilledList.forEach(funItem => funItem())
            }
        }
        // 定义reject函数
        let reject = (val) => {
            // 1、状态变更为rejected
            if (this.status === 'pending') {
                this.status = 'rejected'
                // 2、保存reject()时的值
                this.rejectValue = val

                // 执行then的reject回调
                this.onRejectedList.forEach(funItem => funItem())
            }
        }
        try {
            // 执行函数
            // console.log("执行函数",fun)
            // console.log("resolve-->:",resolve)
            // console.log("====reject-->:",reject);
            fun(resolve, reject)
        } catch (err) {
            reject(err)
        }
    }
}
const formatPromise = (promise, result, resolve, reject) => {
    // 首先，先判断下promise是不是result，因为我们知道，我们的result是一个返回值，他可能是一个Promise，那如果他直接返回第一个参数中的promise的话，那么是会造成死循环的。
    if (promise === result) {
        return new Error('未知的result')
    }
    // 判断是否是对象
    if (typeof result === 'object' && result != null) {
        try {
            // 如果是对象，先看是否存在then函数
            let then = result.then
            // 如果result.then是一个函数，就说明是Promise对象，或者thenable对象
            if (typeof then === 'function') {
                // 利用.call将this指向result，防止result.then()报错
                then.call(result, res => {
                    // 替换resolve(res)
                    formatPromise(promise, res, resolve, reject)
                }, err => {
                    reject(err)
                })
            } else {
                // 如果不是function,那么说明只是普通对象，并不是Promise对象，当普通值处理
                resolve(result)
            }
        } catch (err) {
            reject(err)
        }
    } else {
        // 不是对象，那就是普通值或者函数，直接resolve()
        resolve(result)
    }
```

```
}
MyPromise.prototype.then = function (onFullFilled, onRejected) {
    // 将then函数内部返回的Promise对象取名为promise2，后续文档中将直接以promise2来表示这个对象
    const promise2 = new MyPromise((resolve, reject) => {
        // onFulfilled, onRejected分别resolve()时的回调函数和reject()时的回调函数
        // 此时，判断状态，不同状态时，分别执行不同的回调
        if (this.status === 'fulfilled') {
            setTimeout(() => {
                try {
                    // 定义一个变量来保存onFullFilled的返回值
                    let result = onFullFilled(this.resolveValue)
                    formatPromise(promise2, result, resolve, reject)
                } catch (err) {
                    reject(err) // 捕捉上面代码执行的错误
                }
            }, 0) // 这里说明下为说明要用setTimeout，因为我这段代码要用到promise2，而如果是同步代码，promise2不可在自己的立即执行函数内调用自己
        }
        if (this.status === 'rejected') {
            setTimeout(() => {
                try {
                    // 定义一个变量来保存onRejected的返回值
                    let result = onRejected(this.rejectValue )
                    formatPromise(promise2, result, resolve, reject)
                } catch (err) {
                    reject(err) // 捕捉上面代码执行的错误
                }
            }, 0)
        }
        if (this.status === 'pending') {
            this.onFullFilledList.push(() => {
                setTimeout(() => {
                    try {
                        // 定义一个变量来保存onFullFilled的返回值
                        let result = onFullFilled(this.resolveValue)
                        formatPromise(promise2, result, resolve, reject)
                    } catch (err) {
                        reject(err) // 捕捉上面代码执行的错误
                    }
                }, 0)
            })
            this.onRejectedList.push(() => {
                setTimeout(() => {
                    try {
                        // 定义一个变量来保存onRejected的返回值
                        let result = onRejected(this.rejectValue )
                        formatPromise(promise2, result, resolve, reject)
                    } catch (err) {
                        reject(err) // 捕捉上面代码执行的错误
                    }
                }, 0)
            })
        }
    })
    return promise2
}
const promise1 = new MyPromise((resolve, reject) => {
```

```
    setTimeout(() => {
        resolve(123)
    }, 0)
})
promise1.then((res) => {
    console.log(res)
    return new MyPromise((resolve, reject) => {
        const promise3 = new MyPromise((resolve, reject) => {
            resolve(234)
        })
        resolve(promise3)
    })
}, (err) => {
    console.log(err)
}).then(res => {
    console.log(res)
})
// 输出
123
234
```

Promise.catch()

其实.catch()，和.then()的reject回调是一样的。只是使用位置不一样罢了。并且.catch()也支持链式调用。也就是说.catch和.then其实是一样的，也返回了一个Promise对象对不对，因为这样才能链式调用。所以其实catch很简单，他其实内部就是执行了一个只有reject回调的then函数。下面我们看下代码

现在，我们这个Promise核心代码好像是已经完成了是吧！ 但其实，这里还有bug。下面我们继续看下下面一段代码：

```
const promise1 = new MyPromise((resolve, reject) => {
    setTimeout(() => {
        resolve(123)
    }, 0)
})
promise1.then((res) => {
    console.log(res)
    return new MyPromise((resolve, reject) => {
        reject(new Error('345'))
    })
}, (err) => {
    console.log(1)
    console.log(err)
}).then((res) => {
    console.log('第二个then')
}).catch((err) => {
    console.log(2)
    console.log(err)
})
// 输出
123
2
TypeError: onRejected is not a function
```

当我们在触发错误的地方和catch函数之间插入了一个then的时候，发生了什么啊，我们发现，catch的回调是执行了，但是这个错误并没有被抛出来，then函数内部报错了。为什么啊。
这个时候我们就要想到一点，Promise的错误捕获是不是一层层捕获的啊，按理说第一个then内部抛出了错误，我们是不是优先在第一个then后面的函数内进行捕获啊（也就是第二个then内），但是，由于我们并没有给第二个then定义一个错误捕获的函数，所以这个时候是不是就报错了啊 ， 说onRejected（也就是then的第二个参数）不是一个函数。但是原生Promise功能是怎样的啊。当没有第二个参数的时候，错误是不是会继续往下传递啊。所以这个时候，我们需要判断一下第二个参数到底有没有。如果没有，或者不是函数，我们是不是要将错误，继续往下抛出啊。下面我们改造下then,继续看代码

```
const isFun = (fun) => typeof fun === 'function'
MyPromise.prototype.then = function (onFullFilled, onRejected) {
    // 判断onRejected是否存在或者是否是函数，如果不是函数或者不存在，我们让它等于一个函数，并且在函数内继续将err向下抛出
    onRejected = isFun(onRejected) ? onRejected : err => {
        throw err
    }

    // 将then函数内部返回的Promise对象取名为promise2，后续文档中将直接以promise2来表示这个对象
    const promise2 = new MyPromise((resolve, reject) => {
        // onFulfilled, onRejected分别resolve()时的回调函数和reject()时的回调函数
        // 此时，判断状态，不同状态时，分别执行不同的回调
        if (this.status === 'fulfilled') {
            setTimeout(() => {
                try {
                    // 定义一个变量来保存onFullFilled的返回值
                    let result = onFullFilled(this.resolveValue)
                    formatPromise(promise2, result, resolve, reject)
                } catch (err) {
                    reject(err) // 捕捉上面代码执行的错误
                }
            }, 0) // 这里说明下为说明要用setTimeout，因为我这段代码要用到promise2，而如果是同步代码，promise2不可在自己的立即执行函数内调用自己
        }
        if (this.status === 'rejected') {
            setTimeout(() => {
                try {
                    // 定义一个变量来保存onRejected的返回值
                    let result = onRejected(this.rejectValue)
                    formatPromise(promise2, result, resolve, reject)
                } catch (err) {
                    reject(err) // 捕捉上面代码执行的错误
                }
            }, 0)
        }
        if (this.status === 'pending') {
            this.onFullFilledList.push(() => {
                setTimeout(() => {
                    try {
                        // 定义一个变量来保存onFullFilled的返回值
                        let result = onFullFilled(this.resolveValue)
                        formatPromise(promise2, result, resolve, reject)
                    } catch (err) {
                        reject(err) // 捕捉上面代码执行的错误
                    }
                }, 0)
```

```
            })
            this.onRejectedList.push(() => {
                setTimeout(() => {
                    try {
                        // 定义一个变量来保存onRejected的返回值
                        let result = onRejected(this.rejectValue)
                        formatPromise(promise2, result, resolve, reject)
                    } catch (err) {
                        reject(err) // 捕捉上面代码执行的错误
                    }
                }, 0)
            })
        }
    })
    return promise2
}
//此时，我们再验证下
const promise1 = new MyPromise((resolve, reject) => {
    setTimeout(() => {
        resolve(123)
    }, 0)
})
promise1.then((res) => {
    console.log(res)
    return new MyPromise((resolve, reject) => {
        reject(new Error('345'))
    })
}, (err) => {
    console.log(1)
    console.log(err)
}).then((res) => {
    console.log('第二个then')
}).catch((err) => {
    console.log(2)
    console.log(err)
})
// 输出
123
2
Error: 345
```

好，catch的错误一步一步向下传递的问题我们解决了。那么then是不是也有这样的问题啊，then函数的值一步一步向下传递的问题我们是不是还没解决？大家还记得我们上篇文章中提到的值穿透的现象吗？当我们的then函数的第一个参数不存在，或者不是函数时，他的值是不是会穿透到第二个then的resolve回调中啊。怎么实现呢，原理和catch其实是一样的。话不多说，直接看代码

```
const isFun = (fun) => typeof fun === 'function'
MyPromise.prototype.then = function (onFullFilled, onRejected) {
    // 判断onRejected是否存在或者是否是函数，如果不是函数或者不存在，我们让它等于一个函数，并且在函数内继续将err向下抛出
    onRejected = isFun(onRejected) ? onRejected : err => {
        throw err
    }
    onFullFilled = isFun(onFullFilled)? onFullFilled : res => res

    // 将then函数内部返回的Promise对象取名为promise2，后续文档中将直接以promise2来表示这个对象
```

```
    const promise2 = new MyPromise((resolve, reject) => {
        // onFulfilled, onRejected分别resolve()时的回调函数和reject()时的回调函数
        // 此时，判断状态，不同状态时，分别执行不同的回调
        if (this.status === 'fulfilled') {
            setTimeout(() => {
                try {
                    // 定义一个变量来保存onFullFilled的返回值
                    let result = onFullFilled(this.resolveValue)
                    formatPromise(promise2, result, resolve, reject)
                } catch (err) {
                    reject(err) // 捕捉上面代码执行的错误
                }
            }, 0) // 这里说明下为说明要用setTimeout，因为我这段代码要用到promise2，而如果是同步代码，promise2不可在自己的立即执行函数内调用自己
        }
        if (this.status === 'rejected') {
            setTimeout(() => {
                try {
                    // 定义一个变量来保存onRejected的返回值
                    let result = onRejected(this.rejectValue)
                    formatPromise(promise2, result, resolve, reject)
                } catch (err) {
                    reject(err) // 捕捉上面代码执行的错误
                }
            }, 0)
        }
        if (this.status === 'pending') {
            this.onFullFilledList.push(() => {
                setTimeout(() => {
                    try {
                        // 定义一个变量来保存onFullFilled的返回值
                        let result = onFullFilled(this.resolveValue)
                        formatPromise(promise2, result, resolve, reject)
                    } catch (err) {
                        reject(err) // 捕捉上面代码执行的错误
                    }
                }, 0)
            })
            this.onRejectedList.push(() => {
                setTimeout(() => {
                    try {
                        // 定义一个变量来保存onRejected的返回值
                        let result = onRejected(this.rejectValue)
                        formatPromise(promise2, result, resolve, reject)
                    } catch (err) {
                        reject(err) // 捕捉上面代码执行的错误
                    }
                }, 0)
            })
        }
    })
    return promise2
}

//我们再来验证下值穿透的现象
const promise1 = new MyPromise((resolve, reject) => {
    setTimeout(() => {
        resolve(123)
```

```
    }, 0)
})
promise1.then('aaa', (err) => {
    console.log(1)
    console.log(err)
}).then((res) => {
    console.log('第二个then')
    console.log(res)
}).catch((err) => {
    console.log(2)
    console.log(err)
})
// 输出
第二个then
123
```

完整代码

```
// 是否是函数
const isFun = (fun) => typeof fun === 'function'

// 是否是Promise对象
const isPromise = (value) => {
    if ((value != null && typeof value === 'object') || typeof value === 'function') {
        if (typeof value.then == 'function') {
            return true
        }
    } else {
        return false
    }
}
// then函数中，返回值的处理函数，判断返回值的类型，并做处理
const formatPromise = (promise, result, resolve, reject) => {
    // 首先，先判断下promise是不是result，因为我们知道，我们的result是一个返回值，他可能是一个Promise，那如果他直接返回第一个参数中的promise的话，那么是会造成死循环的。
    if (promise === result) {
        return new Error('未知的result')
    }
    // 判断是否是对象
    if (typeof result === 'object' && result != null) {
        try {
            // 如果是对象，先看是否存在then函数
            let then = result.then
            // 如果result.then是一个函数，就说明是Promise对象，或者thenable对象
            if (typeof then === 'function') {
                // 利用.call将this指向result，防止result.then()报错
                then.call(result, res => {
                    formatPromise(promise, res, resolve, reject)
                }, err => {
                    reject(err)
                })
            } else {
                // 如果不是function,那么说明只是普通对象，并不是Promise对象，当普通值处理
                resolve(result)
            }
        } catch (err) {
```

```
            reject(err)
        }
    } else {
        // 不是对象，那就是普通值或者函数，直接resolve()
        resolve(result)
    }
}

class MyPromise {
    constructor (fun) {
        // 定义初始状态(3个状态分别是pending, fulfilled, rejected)
        this.status = 'pending'
        // 定义两个变量分别来存储成功时值和失败时的值
        this.resolveValue = null
        this.rejectValue = null
        this.onFullFilledList = []
        this.onRejectedList = []

        // 定义resolve函数
        let resolve = (val) => {
            // 1、将状态变更为fulfilled，但是注意一点，Promise是有个特点的，就是状态只能由pending状态变更为fulfilled或者由pending状态变更为rejected。且，状态变化后，不会再变化。故，我们需要先判断当前是否是等待状态pending
            if (this.status === 'pending') {  // this指向实例化出来Promise对象
                this.status = 'fulfilled'
                // 2、保存resolve时的值，以便后面调用then()方法时使用
                this.resolveValue = val

                // 执行then的resolve回调
                this.onFullFilledList.forEach(funItem => funItem())
            }
        }

        // 定义reject函数
        let reject = (val) => {
            // 1、状态变更为rejected
            if (this.status === 'pending') {
                this.status = 'rejected'
                // 2、保存reject()时的值
                this.rejectValue = val

                // 执行then的reject回调
                this.onRejectedList.forEach(funItem => funItem())
            }
        }

        try {
            fun(resolve, reject)
        } catch (err) {
            reject(err)
        }
    }

}

MyPromise.prototype.then = function (onFullFilled, onRejected) {
    // 判断onRejected是否存在或者是否是函数，如果不是函数或者不存在，我们让它等于一个函数，并且在函数内继续将err向下抛出
```

```
    onRejected = isFun(onRejected) ? onRejected : err => {
        throw err
    }
    onFullFilled = isFun(onFullFilled) ? onFullFilled : res => res

    // 将then函数内部返回的Promise对象取名为promise2，后续文档中将直接以promise2来表示这个对象
    const promise2 = new MyPromise((resolve, reject) => {
        // onFulfilled, onRejected分别resolve()时的回调函数和reject()时的回调函数
        // 此时，判断状态，不同状态时，分别执行不同的回调
        if (this.status === 'fulfilled') {
            setTimeout(() => {
                try {
                    // 定义一个变量来保存onFullFilled的返回值
                    let result = onFullFilled(this.resolveValue)
                    formatPromise(promise2, result, resolve, reject)
                } catch (err) {
                    reject(err) // 捕捉上面代码执行的错误
                }
            }, 0) // 这里说明下为说明要用setTimeout，因为我这段代码要用到promise2，而如果是同步代码，promise2不可在自己的立即执行函数内调用自己
        }
        if (this.status === 'rejected') {
            setTimeout(() => {
                try {
                    // 定义一个变量来保存onRejected的返回值
                    let result = onRejected(this.rejectValue)
                    formatPromise(promise2, result, resolve, reject)
                } catch (err) {
                    reject(err) // 捕捉上面代码执行的错误
                }
            }, 0)
        }
        if (this.status === 'pending') {
            this.onFullFilledList.push(() => {
                setTimeout(() => {
                    try {
                        // 定义一个变量来保存onFullFilled的返回值
                        let result = onFullFilled(this.resolveValue)
                        formatPromise(promise2, result, resolve, reject)
                    } catch (err) {
                        reject(err) // 捕捉上面代码执行的错误
                    }
                }, 0)
            })
            this.onRejectedList.push(() => {
                setTimeout(() => {
                    try {
                        // 定义一个变量来保存onRejected的返回值
                        let result = onRejected(this.rejectValue)
                        formatPromise(promise2, result, resolve, reject)
                    } catch (err) {
                        reject(err) // 捕捉上面代码执行的错误
                    }
                }, 0)
            })
        }
    })
```

```
    return promise2
}

MyPromise.prototype.catch = function (err) {
    return this.then(undefined, err)
}

MyPromise.resolve = (value) => {
    // 如果是一个promise对象就直接将这个对象返回
    if (isPromise(value)) {
        return value
    } else {
        // 如果是一个普通值就将这个值包装成一个promise对象之后返回
        return new MyPromise((resolve, reject) => {
            resolve(value)
        })
    }
}

MyPromise.reject = (value) => {
    return new MyPromise((resolve, reject) => {
        reject(value)
    })
}

MyPromise.all = (arr) => {
    // 返回一个promise
    return new MyPromise((resolve, reject) => {
        let resArr = [] // 存储处理的结果的数组
        // 判断每一项是否处理完了
        let index = 0
        function processData(i, data) {
            resArr[i] = data
            index += 1
            if (index == arr.length) {
                // 处理异步，要使用计数器，不能使用resArr==arr.length
                resolve(resArr)
            }
        }
        for (let i = 0; i < arr.length; i++) {
            if (isPromise(arr[i])) {
                arr[i].then((data) => {
                    processData(i, data)
                }, (err) => {
                    reject(err) // 只要有一个传入的promise没执行成功就走reject
                    return
                })
            } else {
                processData(i, arr[i])
            }
        }
    })
}

MyPromise.race = (arr) => {
    return new MyPromise((resolve, reject) => {
        for (let i = 0; i < arr.length; i++) {
            if (isPromise(arr[i])) {
```

```
                arr[i].then((data) => {
                    resolve(data)// 哪个先完成就返回哪一个的结果
                    return
                }, (err) => {
                    reject(err)
                    return
                })
            } else {
                resolve(arr[i])
            }
        }
    })
}

const promise1 = new MyPromise((resolve, reject) => {
    setTimeout(() => {
        resolve(123)
    }, 0)
})
promise1.then('aaa', (err) => {
    console.log(1)
    console.log(err)
}).then((res) => {
    console.log('第二个then')
    console.log(res)
}).catch((err) => {
    console.log(2)
    console.log(err)
})
```

核心代码我们已经实现了，剩下的几个Promise.Resolve(),Promise.Reject(),Promise.all(),Promise.race()就比较简单了。我们就不做过多的说明了。大家直接看代码吧

Promise.resolve()

```
const isPromise = (value) => {
    if ((value != null && typeof value === 'object') || typeof value === 'function') {
        if (typeof value.then == 'function') {
            return true
        }
    } else {
        return false
    }
}
MyPromise.resolve = (value) => {
    // 如果是一个promise对象就直接将这个对象返回
    if (isPromise(value)) {
        return value
    } else {
        // 如果是一个普通值就将这个值包装成一个promise对象之后返回
        return new MyPromise((resolve, reject) => {
            resolve(value)
        })
    }
```

```
}
```

Promise.reject()

```
MyPromise.reject = (value) => {
    return new MyPromise((resolve, reject) => {
        reject(value)
    })
}
```

Promise.all()

all的特点就是如果有其中一个返回了错误（reject），那么就立即返回错误。否则，必须等到所有的都成功之后才会返回

```
MyPromise.all = (arr) => {
    // 返回一个promise
    return new MyPromise((resolve, reject) => {
        let resArr = [] // 存储处理的结果的数组
        // 判断每一项是否处理完了
        let index = 0
        function processData(i, data) {
            resArr[i] = data
            index += 1
            if (index == arr.length) {
                // 处理异步，要使用计数器，不能使用resArr==arr.length
                resolve(resArr)
            }
        }
        for (let i = 0; i < arr.length; i++) {
            if (isPromise(arr[i])) {
                arr[i].then((data) => {
                    processData(i, data)
                }, (err) => {
                    reject(err) // 只要有一个传入的promise没执行成功就走reject
                    return
                })
            } else {
                processData(i, arr[i])
            }
        }
    })
}
```

使用：

```
//promise all
Promise.all([test(1),test(2)]).then((x)=>{console.log(x)},(y)=>{console.log(y)})
```

Promise.race()

race的特点是，哪个先返回状态，就立即返回这个的状态和值（和赛跑一样，哪个先到，我就用哪个）

```
MyPromise.race = (arr) => {
    return new MyPromise((resolve, reject) => {
        for (let i = 0; i < arr.length; i++) {
```

```
            if (isPromise(arr[i])) {
                arr[i].then((data) => {
                    resolve(data)// 哪个先完成就返回哪一个的结果
                    return
                }, (err) => {
                    reject(err)
                    return
                })
            } else {
                resolve(arr[i])
            }
        }
    })
}
```

使用:

```
function getData1(){
    return new Promise( (resolve, reject) => {
        setTimeout( () => {
            console.log('第一条数据加载成功')
            resolve('data1')
        },500)
    })
}

function getData2(){
    return new Promise( (resolve, reject) => {
        setTimeout( () => {
            console.log('第二条数据加载成功')
            resolve('data2')
        },1000)
    })
}


function getData3(){
    return new Promise( (resolve, reject) => {
        setTimeout( () => {
            console.log('第三条数据加载成功')
            resolve('data3')
        },1000)
    })
}

let p = Promise.race([getData1(), getData2(), getData3()]);

p.then(data => {
    console.log(data)  //打印结果为data1
})
```