

## JS的Promise实现原理

潭州教育前端学院 筑梦编程 2018-10-20

本篇文章主要在于探究 Promise 的实现原理，带领大家一步一步实现一个 Promise，不对其用法做说明。

接下来，带你一步一步实现一个 Promise

### 1. Promise 基本结构

```
new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve('FULFILLED')  
  }, 1000)  
})
```

构造函数Promise必须接受一个函数作为参数，我们称该函数为handle，handle又包含resolve和reject两个参数，它们是两个函数。

定义一个判断一个变量是否为函数的方法，后面会用到

```
// 判断变量否为function  
const isFunction = variable => typeof variable === 'function'
```

首先，我们定义一个名为 MyPromise 的 Class，它接受一个函数 handle 作为参数

```
class MyPromise {  
  constructor (handle) {  
    if (!isFunction(handle)) {  
      throw new Error('MyPromise must accept a function as a parameter')  
    }  
  }  
}
```

## 2. Promise 状态和值

Promise 对象存在以下三种状态：

Pending(进行中)

Fulfilled(已成功)

Rejected(已失败)

状态只能由 Pending 变为 Fulfilled 或由 Pending 变为 Rejected，且状态改变之后不会在发生变化，会一直保持这个状态。

Promise的值是指状态改变时传递给回调函数的值

上文中handle函数包含 resolve 和 reject 两个参数，它们是两个函数，可以用于改变 Promise 的状态和传入 Promise 的值

```
new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve('FULFILLED')  
  }, 1000)  
})
```

这里 resolve 传入的 "FULFILLED" 就是 Promise 的值

resolve 和 reject

resolve : 将Promise对象的状态从 Pending(进行中) 变为 Fulfilled(已成功)

reject : 将Promise对象的状态从 Pending(进行中) 变为 Rejected(已失败)

resolve 和 reject 都可以传入任意类型的值作为实参，表示 Promise 对象成功 (Fulfilled) 和失败 (Rejected) 的值

了解了 Promise 的状态和值，接下来，我们为 MyPromise 添加状态属性和值

首先定义三个常量，用于标记Promise对象的三种状态

```
// 定义Promise的三种状态常量
const PENDING = 'PENDING'
const FULFILLED = 'FULFILLED'
const REJECTED = 'REJECTED'
```

再为 MyPromise 添加状态和值，并添加状态改变的执行逻辑

```
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
    // 执行handle
    try {
      handle(this._resolve.bind(this), this._reject.bind(this))
    } catch (err) {
      this._reject(err)
    }
  }
}
```

```
}  
// 添加resolve时执行的函数  
_resolve (val) {  
  if (this._status !== PENDING) return  
  this._status = FULFILLED  
  this._value = val  
}  
// 添加reject时执行的函数  
_reject (err) {  
  if (this._status !== PENDING) return  
  this._status = REJECTED  
  this._value = err  
}  
}
```

这样就实现了 Promise 状态和值的改变。下面说一说 Promise 的核心: then 方法

### 3. Promise 的 then 方法

Promise 对象的 then 方法接受两个参数:

`promise.then(onFulfilled, onRejected)`

参数可选

onFulfilled 和 onRejected 都是可选参数。

如果 onFulfilled 或 onRejected 不是函数，其必须被忽略

onFulfilled 特性

如果 onFulfilled 是函数：

当 promise 状态变为成功时必须被调用，其第一个参数为 promise 成功状态传入的值（resolve 执行时传入的值）

在 promise 状态改变前其不可被调用

其调用次数不可超过一次

onRejected 特性

如果 onRejected 是函数：

当 promise 状态变为失败时必须被调用，其第一个参数为 promise 失败状态传入的值（reject 执行时传入的值）

在 promise 状态改变前其不可被调用

其调用次数不可超过一次

多次调用

then 方法可以被同一个 promise 对象调用多次

当 promise 成功状态时，所有 onFulfilled 需按照其注册顺序依次回调

当 promise 失败状态时，所有 onRejected 需按照其注册顺序依次回调

返回

then 方法必须返回一个新的 promise 对象

```
promise2 = promise1.then(onFulfilled, onRejected);
```

因此 promise 支持链式调用

```
promise1.then(onFulfilled1, onRejected1).then(onFulfilled2, onRejected2);
```

这里涉及到 Promise 的执行规则，包括“值的传递”和“错误捕获”机制：

1、如果 onFulfilled 或者 onRejected 返回一个值 x，则运行下面的 Promise 解决过程：

[[Resolve]](promise2, x)

若 x 不为 Promise，则使 x 直接作为新返回的 Promise 对象的值，即新的 onFulfilled 或者 onRejected 函数的参数。

若 x 为 Promise，这时后一个回调函数，就会等待该 Promise 对象(即 x)的状态发生变化，才会被调用，并且新的 Promise 状态和 x 的状态相同。

下面的例子用于帮助理解：

```
let promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve()
  }, 1000)
})
promise2 = promise1.then(res => {
  // 返回一个普通值
  return '这里返回一个普通值'
})
promise2.then(res => {
  console.log(res) //1秒后打印出： 这里返回一个普通值
})
```



```
let promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve()
  }, 1000)
})
promise2 = promise1.then(res => {
  // 返回一个Promise对象
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('这里返回一个Promise')
    }, 2000)
  })
})
promise2.then(res => {
  console.log(res) //3秒后打印出： 这里返回一个Promise
})
```

2、如果 onFulfilled 或者onRejected 抛出一个异常 e ，则 promise2 必须变为失败 (Rejected) ，并返回失败的值 e，例如：

```
let promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('success')
  }, 1000)
})
promise2 = promise1.then(res => {
  throw new Error('这里抛出一个异常e')
})
promise2.then(res => {
```

```
    console.log(res)
  }, err => {
    console.log(err) //1秒后打印出：这里抛出一个异常e
  })
```

3、如果onFulfilled 不是函数且 promise1 状态为成功 (Fulfilled) , promise2 必须变为成功 (Fulfilled) 并返回 promise1 成功的值, 例如:

```
let promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('success')
  }, 1000)
})
promise2 = promise1.then('这里的onFulfilled本来是一个函数, 但现在不是')
promise2.then(res => {
  console.log(res) // 1秒后打印出: success
}, err => {
  console.log(err)
})
```

4、如果 onRejected 不是函数且 promise1 状态为失败 (Rejected) , promise2必须变为失败 (Rejected) 并返回 promise1 失败的值, 例如:

```
let promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject('fail')
```

```
    }, 1000)
  })
  promise2 = promise1.then(res => res, '这里的onRejected本来是一个函数，但现在不是')
  promise2.then(res => {
    console.log(res)
  }, err => {
    console.log(err) // 1秒后打印出: fail
  })
}
```

根据上面的规则，我们来为 完善 MyPromise

修改 constructor : 增加执行队列

由于 then 方法支持多次调用，我们可以维护两个数组，将每次 then 方法注册时的回调函数添加到数组中，等待执行

```
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then方法

首先，then 返回一个新的 Promise 对象，并且需要将回调函数加入到执行队列中

```

// 添加then方法
then (onFulfilled, onRejected) {
  const { _value, _status } = this
  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(onFulfilled)
      this._rejectedQueues.push(onRejected)
      break
    // 当状态已经改变时，立即执行对应的回调函数
    case FULFILLED:
      onFulfilled(_value)
      break
    case REJECTED:
      onRejected(_value)
      break
  }
  // 返回一个新的Promise对象
  return new MyPromise((onFulfilledNext, onRejectedNext) => {
  })
}

```

那返回的新的 Promise 对象什么时候改变状态？改变为哪种状态呢？

根据上文中 then 方法的规则，我们知道返回的新的 Promise 对象的状态依赖于当前 then 方法回调函数执行的情况以及返回值，例如 then 的参数是否为一个函数、回调函数执行是否出错、返回值是否为 Promise 对象。

我们来进一步完善 then 方法:

```
// 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
    }
})
}

```

这一部分可能不太好理解，读者需要结合上文中 then 方法的规则来细细的分析。

接着修改 \_resolve 和 \_reject：依次执行队列中的函数

当 resolve 或 reject 方法执行时，我们依次提取成功或失败任务队列当中的函数开始执行，并清空队列，从而实现 then 方法的多次调用，实现的代码如下：

```

// 添加resolve时执行的函数
_resolve (val) {
  if (this._status !== PENDING) return
  // 依次执行成功队列中的函数，并清空队列
  const run = () => {
    this._status = FULFILLED
    this._value = val
    let cb;
    while (cb = this._fulfilledQueues.shift()) {
      cb(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)
}

```

这里还有一种特殊的情况，就是当 resolve 方法传入的参数为一个 Promise 对象时，则该 Promise 对象状态决定当前 Promise 对象的状态。

```

const p1 = new Promise(function (resolve, reject) {
  // ...
});

const p2 = new Promise(function (resolve, reject) {
  // ...
  resolve(p1);
})

```



上面代码中，p1 和 p2 都是 Promise 的实例，但是 p2 的 resolve 方法将 p1 作为参数，即一个异步操作的结果是返回另一个异步操作。

注意，这时 p1 的状态就会传递给 p2，也就是说，p1 的状态决定了 p2 的状态。如果 p1 的状态是 Pending，那么 p2 的回调函数就会等待 p1 的状态改变；如果 p1 的状态已经是 Fulfilled 或者 Rejected，那么 p2 的回调函数将会立刻执行。

我们来修改 \_resolve 来支持这样的特性

```
// 添加 resolve 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Promise的状态才会改变，且状态取决于参数Promise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)  
}
```

这样一个Promise就基本实现了，现在我们来加一些其它的方法

catch 方法

相当于调用 then 方法, 但只传入 Rejected 状态的回调函数

```
// 添加catch方法
catch (onRejected) {
  return this.then(undefined, onRejected)
}
```

静态 resolve 方法

```
// 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))
}
```

静态 reject 方法

```
// 添加静态reject方法
static reject (value) {
  return new MyPromise((resolve ,reject) => reject(value))
}
```

静态 all 方法

```
// 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)
      })
    }
  })
}

```

静态 race 方法

```

// 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)
      }, reject)
    }
  })
}

```

```
    }, err => {
      reject(err)
    })
  }
})
}
```

finally 方法

finally 方法用于指定不管 Promise 对象最后状态如何，都会执行的操作

```
finally (cb) {
  return this.then(
    value => MyPromise.resolve(cb()).then(() => value),
    reason => MyPromise.resolve(cb()).then(() => { throw reason })
  );
};
```

这样一个完整的 Promise 就实现了，大家对 Promise 的原理也有了解，可以让我们在使用 Promise 的时候更加清晰明了。

完整代码如下

```
// 判断变量是否为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resolve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Promise的状态才会改变，且状态取决于参数Promise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 })
    );
}
}

```