

手写 Promise

在上一章节中我们了解了 Promise 的一些易错点，在这一章节中，我们会通过手写一个符合 Promise/A+ 规范的 Promise 来深入理解它，并且手写 Promise 也是一道大厂常考题，在进入正题之前，推荐各位阅读一下 [Promise/A+ 规范](http://www.ituring.com.cn/article/66566) (<http://www.ituring.com.cn/article/66566>)，这样才能更好地理解这个章节的代码。

实现一个简易版 Promise

在完成符合 Promise/A+ 规范的代码之前，我们可以先来实现一个简易版 Promise，因为在面试中，如果你能实现出一个简易版的 Promise 基本可以过关了。

那么我们先来搭建构建函数的大体框架

```
const PENDING = 'pending'
const RESOLVED = 'resolved'
const REJECTED = 'rejected'

function MyPromise(fn) {
  const that = this
  that.state = PENDING
  that.value = null
  that.resolvedCallbacks = []
  that.rejectedCallbacks = []
  // 待完善 resolve 和 reject 函数
  // 待完善执行 fn 函数
}
```

- 首先我们创建了三个常量用于表示状态，对于经常使用的一些值都应该通过常量来管理，便于开发及后期维护
- 在函数体内部首先创建了常量 `that`，因为代码可能会异步执行，用于获取正确的 `this` 对象
- 一开始 `Promise` 的状态应该是 `pending`
- `value` 变量用于保存 `resolve` 或者 `reject` 中传入的值
- `resolvedCallbacks` 和 `rejectedCallbacks` 用于保存 `then` 中的回调，因为当执行完 `Promise` 时状态可能还是等待中，这时候应该把 `then` 中的回调保存起来用于状态改变时使用

接下来我们来完善 `resolve` 和 `reject` 函数，添加在 `MyPromise` 函数体内部

```
function resolve(value) {
  if (that.state === PENDING) {
    that.state = RESOLVED
    that.value = value
    that.resolvedCallbacks.map(cb =>
cb(that.value))
  }
}

function reject(value) {
  if (that.state === PENDING) {
    that.state = REJECTED
    that.value = value
    that.rejectedCallbacks.map(cb =>
cb(that.value))
  }
}
```

这两个函数代码类似，就一起解析了

- 首先两个函数都得判断当前状态是否为等待中，因为规范规定只有等待态才可以改变状态
- 将当前状态更改为对应状态，并且将传入的值赋值给 `value`
- 遍历回调数组并执行

完成以上两个函数以后，我们就该实现如何执行 `Promise` 中传入的函数了

```
try {  
  fn(resolve, reject)  
} catch (e) {  
  reject(e)  
}
```

- 实现很简单，执行传入的参数并且将之前两个函数当做参数传进去
- 要注意的是，可能执行函数过程中会遇到错误，需要捕获错误并且执行 `reject` 函数

最后我们来实现较为复杂的 `then` 函数

```

MyPromise.prototype.then = function(onFulfilled,
onRejected) {
  const that = this
  onFulfilled = typeof onFulfilled === 'function'
? onFulfilled : v => v
  onRejected =
    typeof onRejected === 'function'
    ? onRejected
    : r => {
      throw r
    }
  if (that.state === PENDING) {
    that.resolvedCallbacks.push(onFulfilled)
    that.rejectedCallbacks.push(onRejected)
  }
  if (that.state === RESOLVED) {
    onFulfilled(that.value)
  }
  if (that.state === REJECTED) {
    onRejected(that.value)
  }
}

```

- 首先判断两个参数是否为函数类型，因为这两个参数是可选参数
- 当参数不是函数类型时，需要创建一个函数赋值给对应的参数，同时也实现了透传，比如如下代码

```

// 该代码目前在简单版中会报错
// 只是作为一个透传的例子
Promise.resolve(4).then().then((value) =>
console.log(value))

```

- 接下来就是一系列判断状态的逻辑，当状态不是等待态时，就去执行相对应的函数。如果状态是等待态的话，就往回调函数中 push 函数，比如如下代码就会进入等待态的逻辑

```
new MyPromise((resolve, reject) => {  
  setTimeout(() => {  
    resolve(1)  
  }, 0)  
}).then(value => {  
  console.log(value)  
})
```

以上就是简单版 Promise 实现，接下来一小节是实现完整版 Promise 的解析，相信看完完整版的你，一定会对于 Promise 的理解更上一层楼。

实现一个符合 Promise/A+ 规范的 Promise

这小节代码需要大家配合规范阅读，因为大部分代码都是根据规范去实现的。

我们先来改造一下 resolve 和 reject 函数

```

function resolve(value) {
  if (value instanceof MyPromise) {
    return value.then(resolve, reject)
  }
  setTimeout(() => {
    if (that.state === PENDING) {
      that.state = RESOLVED
      that.value = value
      that.resolvedCallbacks.map(cb =>
cb(that.value))
    }
  }, 0)
}
function reject(value) {
  setTimeout(() => {
    if (that.state === PENDING) {
      that.state = REJECTED
      that.value = value
      that.rejectedCallbacks.map(cb =>
cb(that.value))
    }
  }, 0)
}

```

- 对于 resolve 函数来说，首先需要判断传入的值是否为 Promise 类型
- 为了保证函数执行顺序，需要将两个函数体代码使用 setTimeout 包裹起来

接下来继续改造 then 函数中的代码，首先我们需要新增一个变量 promise2，因为每个 then 函数都需要返回一个新的 Promise 对象，该变量用于保存新的返回对象，然后我们先来改造判断等待态的逻辑

```

if (that.state === PENDING) {
  return (promise2 = new MyPromise((resolve,
reject) => {
    that.resolvedCallbacks.push(() => {
      try {
        const x = onFulfilled(that.value)
        resolutionProcedure(promise2, x, resolve,
reject)
      } catch (r) {
        reject(r)
      }
    })

    that.rejectedCallbacks.push(() => {
      try {
        const x = onRejected(that.value)
        resolutionProcedure(promise2, x, resolve,
reject)
      } catch (r) {
        reject(r)
      }
    })
  })))
}

```

- 首先我们返回了一个新的 Promise 对象，并在 Promise 中传入了一个函数
- 函数的基本逻辑还是和之前一样，往回调数组中 push 函数
- 同样，在执行函数的过程中可能会遇到错误，所以使用了 try...catch 包裹
- 规范规定，执行 onFulfilled 或者 onRejected 函数时会返回一个 x，并且执行 Promise 解决过程，这是为了不同的 Promise 都可以兼容使用，比如 JQuery 的 Promise 能兼容

ES6 的 Promise

接下来我们改造判断执行态的逻辑

```
if (that.state === RESOLVED) {
  return (promise2 = new MyPromise((resolve,
  reject) => {
    setTimeout(() => {
      try {
        const x = onFulfilled(that.value)
        resolutionProcedure(promise2, x, resolve,
reject)
      } catch (reason) {
        reject(reason)
      }
    })
  })))
}
```

- 其实大家可以发现这段代码和判断等待态的逻辑基本一致，无非是传入的函数的函数体需要异步执行，这也是规范规定的
- 对于判断拒绝态的逻辑这里就不一一赘述了，留给大家自己完成这个作业

最后，当然也是最难的一部分，也就是实现兼容多种 Promise 的 resolutionProcedure 函数

```
function resolutionProcedure(promise2, x,
resolve, reject) {
  if (promise2 === x) {
    return reject(new TypeError('Error'))
  }
}
```


首先规范规定了 `x` 不能与 `promise2` 相等，这样会发生循环引用的问题，比如如下代码

```
let p = new MyPromise((resolve, reject) => {
  resolve(1)
})
let p1 = p.then(value => {
  return p1
})
```

然后需要判断 `x` 的类型

```
if (x instanceof MyPromise) {
  x.then(function(value) {
    resolutionProcedure(promise2, value,
      resolve, reject)
  }, reject)
}
```

这里的代码是完全按照规范实现的。如果 `x` 为 `Promise` 的话，需要判断以下几个情况：

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 被执行或拒绝
2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```

let called = false
if (x !== null && (typeof x === 'object' ||
typeof x === 'function')) {
  try {
    let then = x.then
    if (typeof then === 'function') {
      then.call(
        x,
        y => {
          if (called) return
          called = true
          resolutionProcedure(promise2, y,
resolve, reject)
        },
        e => {
          if (called) return
          called = true
          reject(e)
        }
      )
    } else {
      resolve(x)
    }
  } catch (e) {
    if (called) return
    called = true
    reject(e)
  }
} else {
  resolve(x)
}

```

- 首先创建一个变量 `called` 用于判断是否已经调用过函数

- 然后判断 `x` 是否为对象或者函数，如果都不是的话，将 `x` 传入 `resolve` 中
- 如果 `x` 是对象或者函数的话，先把 `x.then` 赋值给 `then`，然后判断 `then` 的类型，如果不是函数类型的话，就将 `x` 传入 `resolve` 中
- 如果 `then` 是函数类型的话，就将 `x` 作为函数的作用域 `this` 调用之，并且传递两个回调函数作为参数，第一个参数叫做 `resolvePromise`，第二个参数叫做 `rejectPromise`，两个回调函数都需要判断是否已经执行过函数，然后进行相应的逻辑
- 以上代码在执行的过程中如果抛错了，将错误传入 `reject` 函数中

以上就是符合 Promise/A+ 规范的实现了，如果你对于这部分代码尚有疑问，欢迎在评论中与我互动。

小结

这一章节我们分别实现了简单版和符合 Promise/A+ 规范的 Promise，前者已经足够应付大部分面试的手写题目，毕竟写出一个符合规范的 Promise 在面试中不大现实。后者能让你更加深入地理解 Promise 的运行原理，做技术的深挖者。