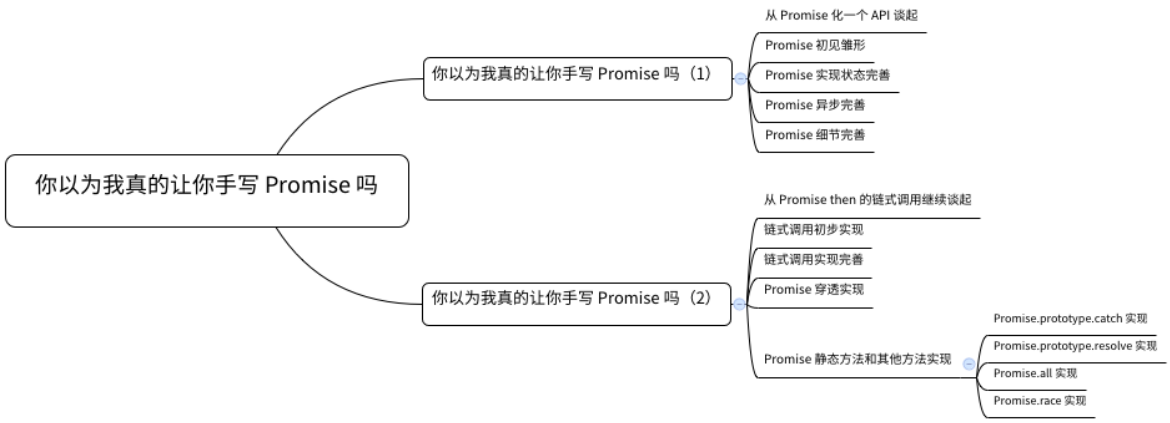
你以为我真的让你手写 Promise 吗（2）？

在上一讲中，我们渐进式地实现了一个貌似能工作的 Promise，并配以实例进行完善。如果你觉得已经接近「大功告成」了，其实这才刚刚开始。Promise 这个概念相对来说比较复杂，下面继续一边研究、一边实现吧。

先来回顾一下这两节课的相关知识点：



话不多说，让我们开始吧。

从 Promise then 的链式调用继续谈起

先来看一道题目：

```
const promise = new Promise((resolve, reject) => {
   setTimeout(() => {
     resolve('lucas')
 }, 2000)
})
```

```
promise.then(data => {
 console.log(data)
 return `${data} next then`
})
.then(data => {
 console.log(data)
})
```

这段代码执行后，将会在 2 秒后输出：lucas，紧接着输出：lucas next then。

我们看到，Promise 实例的 then 方法支持链式调用，输出 resolved 值后，如果在 then 方法体 onfulfilled 函数中同步显式返回新的值，将会在新 Promise 实例的 then 方法 onfulfilled 函数中输出新值。

如果在第一个 then 方法体 onfulfilled 函数中返回另一个 Promise 实例：

```
const promise = new Promise((resolve, reject) => {
 setTimeout(() => {
     resolve('lucas')
 }, 2000)
})

promise.then(data => {
 console.log(data)
 return new Promise((resolve, reject) => {
   setTimeout(() => {
       resolve(`${data} next then`)
   }, 4000)
 })
})
.then(data => {
 console.log(data)
})
```

将在 2 秒后输出：lucas，紧接着再过 4 秒后（第 6 秒）输出：lucas next then。

由此可知：

一个 Promise 实例的 then 方法体 onfulfilled 函数和 onrejected 函数中，是支持再次返回一个 Promise 实例的，也支持返回一个非 Promise 实例的普通值；并且返回的这个 Promise 实例或者这个非 Promise 实例的普通值将会传给下一个 then 方法 onfulfilled 函数或者 onrejected 函数中，这样就支持链式调用了。

那我们该怎么实现这种行为呢?

链式调用初步实现

让我们来分析一下：为了能够支持 then 方法的链式调用，那么每一个 then 方法的 onfulfilled 函数和 onrejected 函数都应该返回一个 Promise 实例。

我们一步一步来，先实现：

```
const promise = new Promise((resolve, reject) => {
 setTimeout(() => {
     resolve('lucas')
 }, 2000)
})

promise.then(data => {
 console.log(data)
 return `${data} next then`
})
.then(data => {
 console.log(data)
})
```

这种 onfulfilled 函数返回一个普通值的场景，这里 onfulfilled 函数指的是：

```
data => {
 console.log(data)
 return `${data} next then`
}
```

在我们上一讲实现的 then 方法中，就可以创建一个新的 promise2 用以返回：

```
Promise.prototype.then = function(onfulfilled, onrejected)
{
   onfulfilled = typeof onfulfilled === 'function' ?
onfulfilled : data => data
 onrejected = typeof onrejected === 'function' ?
onrejected : error => { throw error }
 // promise2 将作为 then 方法的返回值
 let promse2
 if (this.status === 'fulfilled') {
   return promse2 = new Promise((resolve, reject) => {
           setTimeout(() => {
               try {
                   // 这个新的 promse2 resolved 的值为
onfulfilled 的执行结果
                   let result = onfulfilled(this.value)
                   resolve(result)
               }
               catch(e) {
                   reject(e)
               }
           })
   })

 }
 if (this.status === 'rejected') {
   onrejected(this.reason)
 }
 if (this.status === 'pending') {
   this.onFulfilledArray.push(onfulfilled)
   this.onRejectedArray.push(onrejected)
 }
}
```

当然别忘了 this.status === 'rejected' 状态和 this.status === 'pending' 状态也要加入相同的逻辑：

```
Promise.prototype.then = function(onfulfilled, onrejected)
{
 // promise2 将作为 then 方法的返回值
 let promise2
 if (this.status === 'fulfilled') {
   return promise2 = new Promise((resolve, reject) => {
           setTimeout(() => {
               try {
                   // 这个新的 promise2 resolved 的值为
onfulfilled 的执行结果
                   let result = onfulfilled(this.value)
                   resolve(result)
               }
               catch(e) {
                   reject(e)
               }
           })
   })
 }
 if (this.status === 'rejected') {
   return promise2 = new Promise((resolve, reject) => {
           setTimeout(() => {
               try {
                   // 这个新的 promise2 reject 的值为
onrejected 的执行结果
                   let result = onrejected(this.value)
                   resolve(result)
               }
               catch(e) {
                   reject(e)
               }
           })
   })
 }
 if (this.status === 'pending') {
   return promise2 = new Promise((resolve, reject) => {
     this.onFulfilledArray.push(() => {
```

```
        try {
          let result = onfulfilled(this.value)
          resolve(result)
        }
        catch(e) {
          reject(e)
        }
      })

      this.onRejectedArray.push(() => {
        try {
          let result = onrejected(this.reason)
          resolve(result)
        }
        catch(e) {
          reject(e)
        }
      })
    })
  }
}
```

这里要重点理解 this.status === 'pending' 判断分支中的逻辑，这也最难理解的。我们先想想：当使用 Promise 实例，调用其 then 方法时，应该返回一个 Promise 实例，返回的就是 this.status === 'pending' 判断分支中返回的 promise2。那么这个 promise2 什么时候被 resolve 或者 reject 呢？应该是在异步结束，依次执行 onFulfilledArray 或者 onRejectedArray 数组中的函数时。

我们再思考，那么 onFulfilledArray 或者 onRejectedArray 数组中的函数应该做些什么呢？很明显，需要将 promise2 的状态切换，并 resolve onfulfilled 函数执行结果或者 reject onrejected 结果。

这也就是我们的改动，将 this.onFulfilledArray.push 的函数由：

```
this.onFulfilledArray.push(onfulfilled)
```

改为：

```
() => {
   setTimeout(() => {
       try {
           let result = onfulfilled(this.value)
           resolve(result)
       }
       catch(e) {
           reject(e)
       }
   })
}
```

的原因。 this. onRejectedArray.push 的函数的改动点同理。

这非常不容易理解，如果读者仍然想不明白，也不需要着急。还是应该先理解透 Promise，再返回来看，多看几次，一定会有所收获。

请注意，此时 Promise 实现的完整代码为：

```
function Promise(executor) {
 this.status = 'pending'
 this.value = null
 this.reason = null
 this.onFulfilledArray = []
 this.onRejectedArray = []

 const resolve = value => {
   if (value instanceof Promise) {
     return value.then(resolve, reject)
   }
   setTimeout(() => {
     if (this.status === 'pending') {
       this.value = value
       this.status = 'fulfilled'
```

```
        this.onFulfilledArray.forEach(func => {
          func(value)
        })
      }
    })
  }

  const reject = reason => {
    setTimeout(() => {
      if (this.status === 'pending') {
        this.reason = reason
        this.status = 'rejected'

        this.onRejectedArray.forEach(func => {
          func(reason)
        })
      }
    })
  }


    try {
        executor(resolve, reject)
    } catch(e) {
        reject(e)
    }
}

Promise.prototype.then = function(onfulfilled, onrejected)
{
  // promise2 将作为 then 方法的返回值
  let promise2
  if (this.status === 'fulfilled') {
    return promise2 = new Promise((resolve, reject) => {
            setTimeout(() => {
                try {
```

```
                    // 这个新的 promise2 resolved 的值为
onfulfilled 的执行结果
                    let result = onfulfilled(this.value)
                    resolve(result)
                }
                catch(e) {
                    reject(e)
                }
            })
    })
  }
  if (this.status === 'rejected') {
    return promise2 = new Promise((resolve, reject) => {
            setTimeout(() => {
                try {
                    // 这个新的 promise2 reject 的值为
onrejected 的执行结果
                    let result = onrejected(this.value)
                    resolve(result)
                }
                catch(e) {
                    reject(e)
                }
            })
    })
  }
  if (this.status === 'pending') {
    return promise2 = new Promise((resolve, reject) => {
      this.onFulfilledArray.push(() => {
        try {
          let result = onfulfilled(this.value)
          resolve(result)
        }
        catch(e) {
          reject(e)
        }
      })
```

```
      this.onRejectedArray.push(() => {
        try {
          let result = onrejected(this.reason)
          resolve(result)
        }
        catch(e) {
          reject(e)
        }
      })
    })
  }
}
```

链式调用实现完善

我们继续来实现 then 方法显式返回一个 Promise 实例的情况。对应场景：

```
const promise = new Promise((resolve, reject) => {
 setTimeout(() => {
     resolve('lucas')
 }, 2000)
})

promise.then(data => {
 console.log(data)
 return new Promise((resolve, reject) => {
   setTimeout(() => {
       resolve(`${data} next then`)
   }, 4000)
 })
})
.then(data => {
 console.log(data)
})
```

对比第一种情况（ onfulfilled 函数和 onrejected 函数返回一个普通值的情况），实现这种 onfulfilled 函数和 onrejected 函数返回一个 Promise 实例也并不困难。但是我们需要小幅度重构一下代码，在上面实现的 let result = onfulfilled(this.value) 语句和 let result = onrejected(this.reason) 语句中，变量 result 由一个普通值会成为一个 Promise 实例。换句话说就是：变量 result 既可以是一个普通值，也可以是一个 Promise 实例，为此我们抽象出 resolvePromise 方法进行统一处理。改动已有实现为：

```
const resolvePromise = (promise2, result, resolve, reject)
=> {

}

Promise.prototype.then = function(onfulfilled, onrejected)
{
 // promise2 将作为 then 方法的返回值
 let promise2
 if (this.status === 'fulfilled') {
   return promise2 = new Promise((resolve, reject) => {
           setTimeout(() => {
               try {
                   //这个新的 promise2 resolved 的值为
onfulfilled 的执行结果
                   let result = onfulfilled(this.value)
                   resolvePromise(promise2, result,
resolve, reject)
               }
               catch(e) {
                   reject(e)
               }
           })
   })
 }
 if (this.status === 'rejected') {
   return promise2 = new Promise((resolve, reject) => {
           setTimeout(() => {
               try {
```

```
                    //这个新的 promise2 reject 的值为
onrejected 的执行结果
                    let result = onrejected(this.value)
                 resolvePromise(promise2, result, resolve,
reject)
                }
                catch(e) {
                    reject(e)
                }
            })
    })
 }
 if (this.status === 'pending') {
   return promise2 = new Promise((resolve, reject) => {
     this.onFulfilledArray.push(value => {
       try {
         let result = onfulfilled(value)
         resolvePromise(promise2, result, resolve, reject)
       }
       catch(e) {
         reject(e)
       }
     })

     this.onRejectedArray.push(reason => {
       try {
         let result = onrejected(reason)
         resolvePromise(promise2, result, resolve, reject)
       }
       catch(e) {
         reject(e)
       }
     })
   })
 }
}
```

现在的任务就是完成 resolvePromise 函数，这个函数接受四个参数：

promise2: 返回的 Promise 实例

result: onfulfilled 或者 onrejected 函数的返回值

resolve: promise2 的 resolve 方法

reject: promise2 的 reject 方法

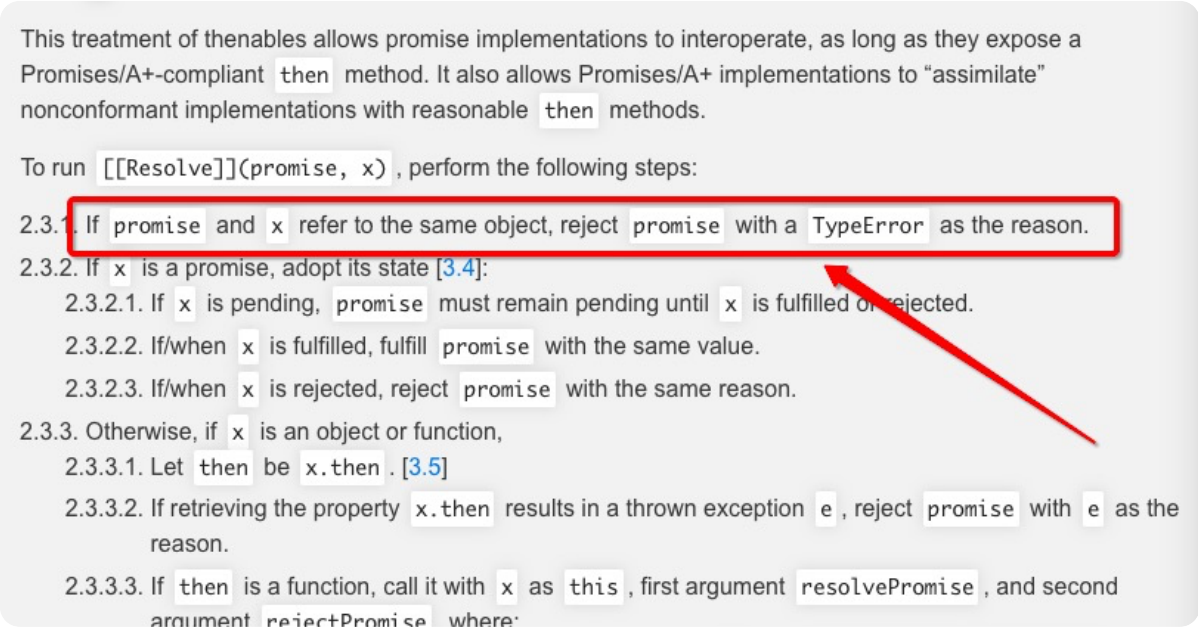
有了这些参数，我们就具备了抽象逻辑的必备条件。接下来就是动手实现：

```
const resolvePromise = (promise2, result, resolve, reject)
=> {
 // 当 result 和 promise2 相等时，也就是说 onfulfilled 返回
promise2 时，进行 reject
 if (result === promise2) {
   reject(new TypeError('error due to circular
reference'))
 }

 // 是否已经执行过 onfulfilled 或者 onrejected
 let consumed = false
 let thenable

 if (result instanceof Promise) {
   if (result.status === 'pending') {
     result.then(function(data) {
       resolvePromise(promise2, data, resolve, reject)
     }, reject)
   } else {
     result.then(resolve, reject)
   }
   return
 }
```

```
 let isComplexResult = target => (typeof target ===
'function' || typeof target === 'object') && (target !==
null)

 // 如果返回的是疑似 Promise 类型
 if (isComplexResult(result)) {
   try {
     thenable = result.then
     // 如果返回的是 Promise 类型，具有 then 方法
     if (typeof thenable === 'function') {
       thenable.call(result, function(data) {
         if (consumed) {
           return
         }
         consumed = true

         return resolvePromise(promise2, data, resolve,
reject)
       }, function(error) {
         if (consumed) {
           return
         }
         consumed = true

         return reject(error)
       })
     }
     else {
       resolve(result)
     }

   } catch(e) {
     if (consumed) {
       return
     }
     consumed = true
     return reject(e)
```

```
    }
  }
  else {
    resolve(result)
  }
}
```

我们看 resolvePromise 方法第一步进行了以「死循环」的处理。并在发生死循环是，reject 掉，错误信息为 new TypeError('error due to circular reference')。怎么理解这个处理呢，规范中为我们指出：



其实出现「死循环」的情况如下：

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
      resolve('lucas')
  }, 2000)
})

promise.then(onfulfilled = data => {
  console.log(data)
  return onfulfilled(data)
})
.then(data => {
```

```
 console.log(data)
})
```

接着，对于 onfulfilled 函数返回的结果 result：如果 result 非 Promise 实例，非对象，非函数类型，是一个普通值的话（上述代码中 isComplexResult 函数进行判断），我们直接将 promise2 以该值 resolve 掉。

对于 onfulfilled 函数返回的结果 result：如果 result 含有 then 属性方法，我们称该属性方法为 thenable，说明 result 是一个 Promise 实例，我们执行该实例的 then 方法（既 thenable），此时的返回结果有可能又是一个 Promise 实例类型，也可能是一个普通值，因此还要递归调用 resolvePromise。如果读者还是不明白这里为什么需要递归调用 resolvePromise，可以看代码例子：

```
const promise = new Promise((resolve, reject) => {
 setTimeout(() => {
     resolve('lucas')
 }, 2000)
})

promise.then(data => {
 console.log(data)
 return new Promise((resolve, reject) => {
   setTimeout(() => {
       resolve(`${data} next then`)
   }, 4000)
 })
 .then(data => {
   return new Promise((resolve, reject) => {
     setTimeout(() => {
         resolve(`${data} next then`)
     }, 4000)
   })
 })
})
.then(data => {
 console.log(data)
})
```

该段代码将会在 2 秒是输出：lucas，10 秒时输出：lucas next then next then。

此时我们的 Promise 实现的完整代码为：

```
function Promise(executor) {
 this.status = 'pending'
 this.value = null
 this.reason = null
 this.onFulfilledArray = []
 this.onRejectedArray = []

 const resolve = value => {
   if (value instanceof Promise) {
     return value.then(resolve, reject)
   }
   setTimeout(() => {
     if (this.status === 'pending') {
       this.value = value
       this.status = 'fulfilled'

       this.onFulfilledArray.forEach(func => {
         func(value)
       })
     }
   })
 }

 const reject = reason => {
   setTimeout(() => {
     if (this.status === 'pending') {
       this.reason = reason
       this.status = 'rejected'

       this.onRejectedArray.forEach(func => {
         func(reason)
       })
```

```
      }
    })
  }


    try {
        executor(resolve, reject)
    } catch(e) {
        reject(e)
    }
}

const resolvePromise = (promise2, result, resolve, reject)
=> {
  // 当 result 和 promise2 相等时，也就是说 onfulfilled 返回
promise2 时，进行 reject
  if (result === promise2) {
    reject(new TypeError('error due to circular
reference'))
  }

  // 是否已经执行过 onfulfilled 或者 onrejected
  let consumed = false
  let thenable

  if (result instanceof Promise) {
    if (result.status === 'pending') {
      result.then(function(data) {
        resolvePromise(promise2, data, resolve, reject)
      }, reject)
    } else {
      result.then(resolve, reject)
    }
    return
  }

  let isComplexResult = target => (typeof target ===
```

```
'function' || typeof target === 'object') && (target !==
null)

 // 如果返回的是疑似 Promise 类型
 if (isComplexResult(result)) {
   try {
     thenable = result.then
     // 如果返回的是 Promise 类型，具有 then 方法
     if (typeof thenable === 'function') {
       thenable.call(result, function(data) {
         if (consumed) {
           return
         }
         consumed = true

         return resolvePromise(promise2, data, resolve,
reject)
       }, function(error) {
         if (consumed) {
           return
         }
         consumed = true

         return reject(error)
       })
     }
     else {
       resolve(result)
     }

   } catch(e) {
     if (consumed) {
       return
     }
     consumed = true
     return reject(e)
   }
```

```
 }
 else {
   resolve(result)
 }
}

Promise.prototype.then = function(onfulfilled, onrejected)
{
 onfulfilled = typeof onfulfilled === 'function' ?
onfulfilled : data => data
 onrejected = typeof onrejected === 'function' ?
onrejected : error => { throw error }

 // promise2 将作为 then 方法的返回值
 let promise2

 if (this.status === 'fulfilled') {
   return promise2 = new Promise((resolve, reject) => {
     setTimeout(() => {
       try {
         // 这个新的 promise2 resolved 的值为 onfulfilled 的
执行结果
         let result = onfulfilled(this.value)
         resolvePromise(promise2, result, resolve, reject)
       }
       catch(e) {
         reject(e)
       }
     })
   })
 }
 if (this.status === 'rejected') {
   return promise2 = new Promise((resolve, reject) => {
     setTimeout(() => {
       try {
         // 这个新的 promise2 reject 的值为 onrejected 的执行
结果
```

```
          let result = onrejected(this.reason)
          resolvePromise(promise2, result, resolve, reject)
        }
        catch(e) {
          reject(e)
        }
      })
    })
  }
  if (this.status === 'pending') {
    return promise2 = new Promise((resolve, reject) => {
      this.onFulfilledArray.push(value => {
        try {
          let result = onfulfilled(value)
          resolvePromise(promise2, result, resolve, reject)
        }
        catch(e) {
          reject(e)
        }
      })

      this.onRejectedArray.push(reason => {
        try {
          let result = onrejected(reason)
          resolvePromise(promise2, result, resolve, reject)
        }
        catch(e) {
          reject(e)
        }
      })
    })
  }
}
```

Promise 穿透实现

到这里，读者可以松口气，我们的 Promise 基本实现除了静态方法以外，已经完成了 95%。为什么不是 100% 呢？其实还有一处细节，我们看代码：

```
const promise = new Promise((resolve, reject) => {
 setTimeout(() => {
     resolve('lucas')
 }, 2000)
})


promise.then(null)
.then(data => {
 console.log(data)
})
```

这段代码将会在 2 秒后输出：lucas。这就是 Promise 穿透现象：

给 .then() 函数传递非函数值作为其参数时，实际上会被解析成 .then(null)，这时候的表现应该是：上一个 promise 对象的结果进行「穿透」，如果在后面链式调用仍存在第二个 .then() 函数时，将会获取被穿透下来的结果。

那该如何实现 Promise 穿透呢?

其实很简单，并且我们已经做到了。想想在 then() 方法的实现中：我们已经对 onfulfilled 和 onrejected 函数加上判断：

```
Promise.prototype.then = function(onfulfilled =
Function.prototype, onrejected = Function.prototype) {
 onfulfilled = typeof onfulfilled === 'function' ?
onfulfilled : data => data
 onrejected = typeof onrejected === 'function' ?
onrejected : error => { throw error }

   // ...
}
```

如果 onfulfilled 不是函数类型，则给一个默认值，该默认值是返回其参数的函数。onrejected 函数同理。这段逻辑，就是起到了实现「穿透」的作用。

Promise 静态方法和其他方法实现

这一部分，我们将实现：

- Promise.prototype.catch
- Promise.resolve，Promise.reject
- Promise.all
- Promise.race

Promise.prototype.catch 实现

Promise.prototype.catch 可以进行异常捕获，它的典型用法：

```
const promise1 = new Promise((resolve, reject) => {
 setTimeout(() => {
     reject('lucas error')
 }, 2000)
})

promise1.then(data => {
 console.log(data)
}).catch(error => {
 console.log(error)
})
```

会在 2 秒后输出：lucas error。

其实在这种场景下，它就相当于：

```
Promise.prototype.catch = function(catchFunc) {
 return this.then(null, catchFunc)
}
```

因为我们知道 .then() 方法的第二个参数也是进行异常捕获的，通过这个特性，我们比较简单地实现了 Promise.prototype.catch。

Promise.prototype.resolve 实现

MDN 上对于 Promise.resolve(value) 方法的介绍：

Promise.resolve(value) 方法返回一个以给定值解析后的 Promise 实例对象。

请看实例：

```
Promise.resolve('data').then(data => {
 console.log(data)
})
console.log(1)
```

先输出 1 再输出 data。

那么实现 Promise.resolve(value) 也很简单：

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

顺带实现一个 Promise.reject(value)：

```
Promise.reject = function(value) {
 return new Promise((resolve, reject) => {
```

```
    reject(value)
  })
}
```

Promise.all 实现

MDN 关于 的解释：

Promise.all(iterable) 方法返回一个 Promise 实例，此实例在 iterable 参数内所有的 promise 都「完成（resolved）」或参数中不包含 promise 时回调完成（resolve）；如果参数中 promise 有一个失败（rejected），此实例回调失败（reject），失败原因的是第一个失败 promise 的结果。

还是看实例体会一下：

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
      resolve('lucas')
  }, 2000)
})

const promise2 = new Promise((resolve, reject) => {
  setTimeout(() => {
      resolve('lucas')
  }, 2000)
})

Promise.all([promise1, promise2]).then(data => {
  console.log(data)
})
```

将在 2 秒后输出：["lucas", "lucas"]。

实现思路也很简单：

```
Promise.all = function(promiseArray) {
 if (!Array.isArray(promiseArray)) {
     throw new TypeError('The arguments should be an
array!')
 }
 return new Promise((resolve, reject) => {
   try {
     let resultArray = []

     const length = promiseArray.length

     for (let i = 0; i         promiseArray[i].then(data =>
{
         resultArray.push(data)

         if (resultArray.length === length) {
           resolve(resultArray)
         }
       }, reject)
     }
   }
   catch(e) {
     reject(e)
   }
 })
}
```

我们先进行了对参数 promiseArray 的类型判断，对于非数组类型参数，进行抛错。Promise.all 会返回一个 Promise 实例，这个实例将会在 promiseArray 中的所有 Promise 实例 resolve 后进行 resolve，且 resolve 的值是一个数组，这个数组存有 promiseArray 中的所有 Promise 实例 resolve 的值。

整体思路依赖一个 for 循环对 promiseArray 进行遍历。同样按照这个思路，我们对 Promise.race 进行实现。

Promise.race 实现

还是先来看一下 Promise.race 的用法。

```
const promise1 = new Promise((resolve, reject) => {
 setTimeout(() => {
     resolve('lucas1')
 }, 2000)
})

const promise2 = new Promise((resolve, reject) => {
 setTimeout(() => {
     resolve('lucas2')
 }, 4000)
})

Promise.race([promise1, promise2]).then(data => {
 console.log(data)
})
```

将会在 2 秒后输出：lucas1，实现 Promise.race 为：

```
Promise.race = function(promiseArray) {
 if (!Array.isArray(promiseArray)) {
     throw new TypeError('The arguments should be an
array!')
 }
 return new Promise((resolve, reject) => {
   try {
         const length = promiseArray.length
     for (let i = 0; i
       promiseArray[i].then(resolve, reject)
     }
   }
   catch(e) {
     reject(e)
   }
 })
}
```

我们来简单分析一下，这里使用 for 循环同步执行 promiseArray 数组中的所有 promise 实例 then 方法，第一个 resolve 的实例直接会触发新 Promise（代码中新 new 出来的） 实例的 resolve 方法。

总结

这两节课，相信读者通过对 Promise 实现的学习，对 Promise 这个概念的理解大大加深。其实，实现一个 Promise 不是目的，并且这个 Promise 实现也没有完全 100% 遵循规范，我们更加应该掌握概念，融会贯通。另外，整体来看，这部分内容不好理解，如果暂时难以接受全部概念，也不要灰心。实现的代码就在那里，我们要有决心慢慢地掌握它。

最终把所有的实现放在一起：

```
function Promise(executor) {
 this.status = 'pending'
 this.value = null
 this.reason = null
 this.onFulfilledArray = []
 this.onRejectedArray = []

 const resolve = value => {
   if (value instanceof Promise) {
     return value.then(resolve, reject)
   }
   setTimeout(() => {
     if (this.status === 'pending') {
       this.value = value
       this.status = 'fulfilled'

       this.onFulfilledArray.forEach(func => {
         func(value)
       })
     }
   })
 }
```

```
 const reject = reason => {
   setTimeout(() => {
     if (this.status === 'pending') {
       this.reason = reason
       this.status = 'rejected'

       this.onRejectedArray.forEach(func => {
         func(reason)
       })
     }
   })
 }


   try {
       executor(resolve, reject)
   } catch(e) {
       reject(e)
   }
}

const resolvePromise = (promise2, result, resolve, reject)
=> {
 // 当 result 和 promise2 相等时，也就是说 onfulfilled 返回
promise2 时，进行 reject
 if (result === promise2) {
   return reject(new TypeError('error due to circular
reference'))
 }

 // 是否已经执行过 onfulfilled 或者 onrejected
 let consumed = false
 let thenable

 if (result instanceof Promise) {
   if (result.status === 'pending') {
     result.then(function(data) {
```

```
        resolvePromise(promise2, data, resolve, reject)
      }, reject)
    } else {
      result.then(resolve, reject)
    }
    return
  }

  let isComplexResult = target => (typeof target ===
'function' || typeof target === 'object') && (target !==
null)
  // 如果返回的是疑似 Promise 类型
  if (isComplexResult(result)) {
    try {
      thenable = result.then
      // 如果返回的是 Promise 类型，具有 then 方法
      if (typeof thenable === 'function') {
        thenable.call(result, function(data) {
          if (consumed) {
            return
          }
          consumed = true

          return resolvePromise(promise2, data, resolve,
reject)
        }, function(error) {
          if (consumed) {
            return
          }
          consumed = true

          return reject(error)
        })
      }
      else {
        return resolve(result)
      }
```

```
    } catch(e) {
      if (consumed) {
        return
      }
      consumed = true
      return reject(e)
    }
  }
  else {
    return resolve(result)
  }
}

Promise.prototype.then = function(onfulfilled, onrejected)
{
  onfulfilled = typeof onfulfilled === 'function' ?
onfulfilled : data => data
  onrejected = typeof onrejected === 'function' ?
onrejected : error => {throw error}

  // promise2 将作为 then 方法的返回值
  let promise2

  if (this.status === 'fulfilled') {
    return promise2 = new Promise((resolve, reject) => {
      setTimeout(() => {
        try {
          // 这个新的 promise2 resolved 的值为 onfulfilled 的
执行结果
          let result = onfulfilled(this.value)
          resolvePromise(promise2, result, resolve, reject)
        }
        catch(e) {
          reject(e)
        }
      })
```

```
    })
  }
  if (this.status === 'rejected') {
    return promise2 = new Promise((resolve, reject) => {
      setTimeout(() => {
        try {
          // 这个新的 promise2 reject 的值为 onrejected 的执行
结果
          let result = onrejected(this.reason)
          resolvePromise(promise2, result, resolve, reject)
        }
        catch(e) {
          reject(e)
        }
      })
    })
  }
  if (this.status === 'pending') {
    return promise2 = new Promise((resolve, reject) => {
      this.onFulfilledArray.push(value => {
        try {
          let result = onfulfilled(value)
          resolvePromise(promise2, result, resolve, reject)
        }
        catch(e) {
          return reject(e)
        }
      })

      this.onRejectedArray.push(reason => {
        try {
          let result = onrejected(reason)
          resolvePromise(promise2, result, resolve, reject)
        }
        catch(e) {
          return reject(e)
        }
```

```
      })
    })
  }
}

Promise.prototype.catch = function(catchFunc) {
  return this.then(null, catchFunc)
}

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

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

Promise.race = function(promiseArray) {
  if (!Array.isArray(promiseArray)) {
      throw new TypeError('The arguments should be an
array!')
  }
  return new Promise((resolve, reject) => {
    try {
      const length = promiseArray.length
      for (let i = 0; i
        promiseArray[i].then(resolve, reject)
      }
    }
    catch(e) {
      reject(e)
    }
  })
```

```
}

Promise.all = function(promiseArray) {
 if (!Array.isArray(promiseArray)) {
     throw new TypeError('The arguments should be an
array!')
 }
 return new Promise((resolve, reject) => {
   try {
     let resultArray = []

     const length = promiseArray.length

     for (let i = 0; i         promiseArray[i].then(data =>
{
         resultArray.push(data)

         if (resultArray.length === length) {
           resolve(resultArray)
         }
       }, reject)
     }
   }
   catch(e) {
     reject(e)
   }
 })
}
```

点击查看下一节

面向对象和原型——永不过时的话题