

16 起底 Promise/A+——决议程序（Resolution Procedure）

更新时间：2020-04-14 11:33:42



“

辛苦是获得一切的定律。——牛顿

”

现有链式调用缺陷分析

我们上一节写出来这个 Promise，最明显的一个缺陷就是下一个 then 拿不到上一个 then 的结果：

```
const cutePromise = new CutePromise(function (resolve, reject) {
  resolve('成了！');
});

cutePromise.then((value) => {
  console.log(value)
  console.log('我是第 1 个任务')
  return '第 1 个任务的结果'
}).then(value => {
  // 此处 value 期望输出 '第 1 个任务的结果'
  console.log('第二个任务尝试拿到第 1 个任务的结果是：',value)
});
```

这段代码里我们尝试在第 2 个 `then` 中拿到第 1 个 `then` 中的结果，然而实际的输出却是：

成了！

我是第 1 个任务

第二个任务尝试拿到第 1 个任务的结果是：成了！

第二个 `then` 好像无视了第一个 `then` 的结果，仍然获取到的是我们在 `Promise` 执行器中 `resolve` 出的那个最初的值——这显然是不合理的。

事实上，除了这个最明显的缺陷，我们现在实现出来这个 `Promise` 还有很多能力上的问题，比如说 `thenable` 对象的特殊处理缺失、比如异常处理缺失等等，这些问题可以用一句话来归纳——对 `then` 方法的处理过于粗糙。

重新审视 `then` 方法——理解 `Promise` 决议程序

前面我们说过，整个 `Promise` 规范，在方法层面，基本就是围绕着 `then` 打转。其中一个最需要引起大家注意的东西叫做 **Promise Resolution Procedure(Promise决议程序)**。这个名字翻译过来很绕，尤其是“决议”这个动作，看上去挺唬人的。其实这里的“决议”，描述的就是 `resolve` 这个动作。决议程序，约束的就是 `resolve` 应该如何表现。这个动作和 `then` 息息相关，所以要想把 `then` 方法完善起来，我们必须对决议程序的内容有细致的了解。我们一起来看看 `Promise/A+` 规范中的相关内容：

笔者注：这部分东西理论性较强，大家如果一下不太能消化，建议跳到后面写完代码再回来看。但不管是现在读还是等下回来读，这块东西都是绝不能跳过的。决议程序，是一个区分度非常高的考点。围绕决议程序展开钻研，不仅会帮你理解 `Promise` 的运作机制，更能深化你对 `then` 方法的认知，这些对你应对任何难度的面试都是非常有利的。

决议程序处理是以一个 `promise` 和一个 `value` 为输入的抽象操作，我们把它表示为

`[[Resolve]](promise, x)`

笔者注：别懵。这种形式看起来太高级了一点也不友好，但这种写法你肯定见过：

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

`[[Resolve]](promise, x)`。意思是说如果 `onFulfilled` 或 `onRejected` 返回了值 `x`，则执行 `Promise` 解析流程 `[[Resolve]](promise2, x)`。

只要都实现了 `promise/A+` 标准，那么不同的 `Promise` 都可以之间相互调用。

1. 如果 `x` 和 `promise` 都指向同一个对象，则以 `TypeError` 为 `reason` 拒绝执行 `promise`。
2. 如果 `x` 是 `Promise` 对象，则 `promise` 采用 `x` 当前的状态：
 - a. 如果 `x` 是 `pending` 状态，`promise` 必须保持 `pending` 状态直到 `x` 的状态变为 `resolved` 或者 `rejected`。
 - b. 如果 `x` 是 `resolved` 状态，用相同的值 `value` 执行 `promise`。
 - c. 如果 `x` 是 `rejected` 状态，则用相同的原因 `reason` 执行 `promise`。
3. 如果 `x` 是一个对象或者函数：
 - a. 将 `promise` 的 `then` 方法指向 `x.then`。
 - b. 如果 `x.then` 属性抛出异常 `error`，则以 `error` 为 `reason` 来调用 `reject`。
 - c. 如果 `then` 是是一个函数，那么用 `x` 为 `this` 来调用它，第一个参数为 `resolvePromise`，第二个参数为 `rejectPromise`

- i. 如果 `resolvePromise` 以值 `y` 为参数被调用，则运行 `[[Resolve]](promise, y)`。
 - ii. 如果 `rejectPromise` 以原因 `r` 为参数被调用，则用原因 `r` 执行 `promise` (`reject`)。
 - iii. 如果 `resolvePromise` 和 `rejectPromise` 均被调用，或者被同一参数调用了多次，则使用第一次调用并忽略剩下的调用。
 - iv. 如果 `then` 抛出了异常 `error`
4. 如果 `resolvePromise` 或 `rejectPromise` 已经被调用，则忽略它。
 5. 否则用 `error` 为 `reason` 拒绝 `promise`
 - d. 如果 `then` 不是 `function`，用 `x` 为参数执行 `promise`
 6. 如果 `x` 不是一个 `object` 或者 `function`，用 `x` 为参数执行 `promise`。

用决议程序完善 **CutePromise**

咱们主要的思路在于把上述的决议程序的逻辑给提出来，在此基础上完善 `then` 方法（因为决议程序我们会放到 `then` 方法里来调用）。

构造函数改造

构造函数侧的改造无需太多，我们主要是把 `setTimeout` 给拿掉。这是因为后续我们会把异步处理放到 `then` 方法中的 `resolveByStatus/ rejectByStatus` 里面来做。

```

function CutePromise(executor) {
  // value 记录异步任务成功的执行结果
  this.value = null;
  // reason 记录异步任务失败的原因
  this.reason = null;
  // status 记录当前状态，初始化是 pending
  this.status = 'pending';

  // 缓存两个队列，维护 resolved 和 rejected 各自对应的处理函数
  this.onResolvedQueue = [];
  this.onRejectedQueue = [];

  // 把 this 存下来，后面会用到
  var self = this;

  // 定义 resolve 函数
  function resolve(value) {
    // 如果是 pending 状态，直接返回
    if (self.status !== 'pending') {
      return;
    }
    // 异步任务成功，把结果赋值给 value
    self.value = value;
    // 当前状态切换为 resolved
    self.status = 'resolved';
    // 批量执行 resolved 队列里的任务
    self.onResolvedQueue.forEach(resolved => resolved(self.value));
  }

  // 定义 reject 函数
  function reject(reason) {
    // 如果是 pending 状态，直接返回
    if (self.status !== 'pending') {
      return;
    }
    // 异步任务失败，把结果赋值给 value
    self.reason = reason;
    // 当前状态切换为 rejected
    self.status = 'rejected';
    // 用 setTimeout 延迟队列任务的执行
    // 批量执行 rejected 队列里的任务
    self.onRejectedQueue.forEach(rejected => rejected(self.reason));
  }

  // 把 resolve 和 reject 能力赋予执行器
  executor(resolve, reject);
}

```

下面我们来编写决议程序！这个 `resolutionProcedure` 可以说是咱们这节的一个学习的关键，各位留心阅读逐行注释中的解析：

```

function resolutionProcedure(promise2, x, resolve, reject) {
  // 这里 hasCalled 这个标识，是为了确保 resolve、reject 不要被重复执行
  let hasCalled;
  if (x === promise2) {
    // 决议程序规范：如果 resolve 结果和 promise2 相同则 reject，这是为了避免死循环
    return reject(new TypeError('为避免死循环，此处抛错'));
  } else if (x !== null && (typeof x === 'object' || typeof x === 'function')) {
    // 决议程序规范：如果 x 是一个对象或者函数，则需要额外处理下
    try {
      // 首先是看它有没有 then 方法（是不是 thenable 对象）
      let then = x.then;
      // 如果是 thenable 对象，则将 promise 的 then 方法指向 x.then。
      if (typeof then === 'function') {
        // 如果 then 是一个函数，那么用 x 为 this 来调用它，第一个参数为 resolvePromise，第二个参数为 rejectPromise
        then.call(x, y => {
          // 如果已经被 resolve/reject 过了，那么直接 return
          if (hasCalled) return;
          hasCalled = true;
          // 进入决议程序（递归调用自身）
          resolutionProcedure(promise2, y, resolve, reject);
        }, err => {
          // 这里 hasCalled 用法和上面意思一样
          if (hasCalled) return;
          hasCalled = true;
          reject(err);
        });
      } else {
        // 如果 then 不是 function，用 x 为参数执行 promise
        resolve(x);
      }
    } catch (e) {
      if (hasCalled) return;
      hasCalled = true;
      reject(e);
    }
  } else {
    // 如果 x 不是一个 object 或者 function，用 x 为参数执行 promise
    resolve(x);
  }
}

```

这个决议程序会在 `then` 方法中被调用（`then` 方法同样伴随不小改动，大家留心注释解析）：

```

// then 方法接收两个函数作为入参（可选）
CutePromise.prototype.then = function(onResolved, onRejected) {

  // 注意，onResolved 和 onRejected 必须是函数；如果不是，我们此处用一个透传来兜底
  if (typeof onResolved !== 'function') {
    onResolved = function(x) {return x};
  }
  if (typeof onRejected !== 'function') {
    onRejected = function(e) {throw e};
  }

  // 依然是保存 this
  var self = this;
  // 这个变量用来存返回值 x
  let x

  // resolve 态的处理函数
  function resolveByStatus(resolve, reject) {
    // 包装成异步任务，确保决议程序在 then 后执行
    setTimeout(function() {
      try {
        // 返回值赋值给 x
        x = onResolved(self.value);
        // 进入决议程序

```

```

        resolutionProcedure(promise2, x, resolve, reject);
    } catch (e) {
        // 如果onResolved或者onRejected抛出异常error，则promise2必须被rejected，用error做reason
        reject(e);
    }
    });
}

// reject态的处理函数
function rejectByStatus(resolve, reject) {
    // 包装成异步任务，确保决议程序在 then 后执行
    setTimeout(function() {
        try {
            // 返回值赋值给 x
            x = onRejected(self.reason);
            // 进入决议程序
            resolutionProcedure(promise2, x, resolve, reject);
        } catch (e) {
            reject(e);
        }
    });
}

// 注意，这里我们不能再简单粗暴 return this 了，需要 return 一个符合规范的 Promise 对象
var promise2 = new CutePromise(function(resolve, reject) {
    // 判断状态，分配对应的处理函数
    if (self.status === 'resolved') {
        // resolve 处理函数
        resolveByStatus(resolve, reject);
    } else if (self.status === 'rejected') {
        // reject 处理函数
        rejectByStatus(resolve, reject);
    } else if (self.status === 'pending') {
        // 若是 pending，则将任务推入对应队列
        self.onResolvedQueue.push(function() {
            resolveByStatus(resolve, reject);
        });
        self.onRejectedQueue.push(function() {
            rejectByStatus(resolve, reject);
        });
    }
});

// 把包装好的 promise2 return 掉
return promise2;
};

```

如此一来，我们就实现了一个符合预期的 **Promsie** 了，它可以通过 这套[Promise/A+ 规范](#)的测试用例。

小建议

手写 **Promise**，在不同的面试官、不同的团队里，有着不同的答题标准。对一些团队来说，完成到我们上一节结束时那种程度，已经可以拿到全部的分。如果你是第一次阅读本专栏、第一次接触 **Promise** 底层原理，同时在阅读本节的过程中感到吃力，这是非常正常的事情。不必心急，如果时间充裕，试着去多读几遍、一行一行跟着敲下来；如果急于完成知识点扫盲，那么也可以先跳过本节。待整个知识体系的骨架搭建起来后，再回头来集中火力克服掉它^_^。

学习拓展

[Promises/A+规范-英文](#)

[Promises/A+规范-翻译](#)

}



15 起底 Promise/A+ —— 从实践到原理

17 变量提升与暂时性死区

