# 15 起底 Promise/A+ —— 从实践到原理

更新时间：2020-08-01 18:07:53

> 天才就是百分之二的灵感，百分之九十八的汗水。——爱迪生

## Promise/A+ 规范，咱们晚点在看

很多人会告诉你，写 Promise 的第一步，就是认真阅读Promise/A+规范（Promise/A+ 规范的原文大家可以点击https://promisesaplus.com/查看）。

为什么说要先读规范？因为规范就意味着标准，它是对你目标产物的特性的约束——必须得符合我这里说给你的这些特征，你才能是算是个 Promise。我们可以认为，开发 Promise 的这个过程，也像是在写一个需求，而Promise/A+ 规范，就是我们的需求文档。

然而对大多数同学来说，在学习这个阶段，你需要的本来就不是一份需求文档，而是一份学习指南。多数同学只要点开这份”需求文档“扫上一眼，他很可能就已经不想写 Promise 了——规范，本质上就是把对新手来说晦涩的知识，以生硬的形式表达了出来。对高手来说，照着规范做实现，不是啥难事。但是对学习者来说，这无疑于在没学完课本、没做过练习题的情况下就被要求去参加考试了，非常容易产生挫败感进而放弃。

其实这个顺序倒过来比较合理——先跟着我撸一个 Promise 出来，在写的过程中，我会一点一点告诉你，为什么要这样做，规范对此是如何描述的。在这个过程中，你对 Promise/A+ 规范的认知和理解会从无到有，从模糊到通透。写完之后，再自己回头去阅读规范原文，你便会发现那些原本看似晦涩的条条框框，一下子变得生动鲜活起来了。此时再去细细琢磨里面的每一句话，就会越读越有味道。“从实践到原理”的用意就在此。

## 快速上手：executor 与三种状态

我们现在回忆一下之前咱们用过的 Promise。从使用的感受上来说，一个 Promise 应该具备的最基本的特征，至少有以下两点：

- 可以接收一个 executor 作为入参
- 具备 pending、resolved 和 rejected 这三种状态

我们先从这最基本的轮廓入手（解析在逐行注释里，本节注释非常重要）：

```
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';

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

  // 定义 resolve 函数
  function resolve(value) {
    // 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';
  }

  // 定义 reject 函数
  function reject(reason) {
    // 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';
  }

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

## then 方法的行为

每一个 promise 实例一定有个 then 方法，由此我们不难想到，then 方法应该装在 Promise 构造函数的原型对象上（解析在逐行注释里，本节注释非常重要）：

```
// 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;
    // 判断是否是 resolved 状态
    if (self.status === 'resolved') {
        // 如果是 执行对应的处理方法
        onResolved(self.value);
    } else if (self.status === 'rejected') {
        // 若是 rejected 状态，则执行 rejected 对应方法
        onRejected(self.reason);
    }
};
```

## 先爽一把

把咱们的 CutePromise 丢进控制台跑跑看吧：

```
new CutePromise(function(resolve, reject){
    resolve('成了！ ');
}).then(function(value){
    console.log(value);
}, function(reason){
    console.log(reason);
});

// 输出 “成了！ ”

new CutePromise(function(resolve, reject){
    reject('错了！ ');
}).then(function(value){
    console.log(value);
}, function(reason){
    console.log(reason);
});

// 输出“错了！ ”
```

OK！各位如果没敲错字的话，咱们可爱版的 Promise 已经妥妥地跑起来了哈。现在骨骼有了，我们给它加点血肉、再画上眉毛眼睛，就是一个人模人样的 Promise 了~

## 链式调用

想必大家还记得，在 Promise 中，then 方法和 catch 方法都是可以通过链式调用这种形式无限调用下去的。这里先给大家透个底儿：Promise/A+ 规范里，其实压根儿没提 catch 的事儿，它只强调了 then 的存在、约束了 then 的行为。所以咱们此处，就是要实现 then 的链式调用。

要想实现链式调用，咱们考虑以下几个重大的改造点：

- then方法中应该直接把 this 给 return 出去（链式调用常规操作）；
- 链式调用允许我们多次调用 then，多个 then 中传入的 onResolved（也叫onFulFilled） 和 onRejected 任务，

我们需要把它们维护在一个队列里；

- 要想办法确保 then 方法执行的时机，务必在 onResolved 队列 和 onRejected 队列批量执行前。不然队列任务批量执行的时候，任务本身都还没收集完，就乌龙了。一个比较容易想到的办法就是**把批量执行这个动作包装成异步任务**，这样就能确保它一定可以在同步代码之后执行了。

OK，明确了改造点之后，咱们动手来完善构造函数这一侧的代码：

```
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) {
    // 如果不是 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';
    // 用 setTimeout 延迟队列任务的执行
    setTimeout(function(){
      // 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) {
    // 如果不是 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
    setTimeout(function(){
      // 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);
}
```

相应地，then 方法也需要进行改造。除了返回 this 以外，现在我们会把 resolved 和 rejected 任务没有完全被推入队列时的情况，全部视为 pending 状态。于是在 then 方法中，我们还需要对 pending 做额外处理：

```
// 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;
    // 判断是否是 resolved 状态
    if (self.status === 'resolved') {
        // 如果是 执行对应的处理方法
        onResolved(self.value);
    } else if (self.status === 'rejected') {
        // 若是 rejected 状态，则执行 rejected 对应方法
        onRejected(self.reason);
    } else if (self.status === 'pending') {
        // 若是 pending 状态，则只对任务做入队处理
        self.onResolvedQueue.push(onResolved);
        self.onRejectedQueue.push(onRejected);
    }
    return this
};
```

## 再爽一把

现在我们来验证链式调用是否能生效：

```
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')
}).then(value => {
    console.log('我是第 2 个任务')
});
// 依次输出“成了！ ” “我是第 1 个任务” “我是第 2 个任务”
```

输出结果如下：

```
成了！
我是第 1 个任务
我是第 2 个任务
```

可以看出，我们的链式调用生效了！

不过，想必细心的同学早已看出，我们现在实现的这个版本的链式调用，相比真实 Promise 的链式调用来说，还是非常单薄的。那么它到底单薄在哪？要想实现一个更完整的链式调用，咱还需要补哪些课？不急不急，先确保你充分理解并吸收了本节的知识（可不要小看我们现在手里这个 CutePromise，你拿着它已经可以干掉不少竞争对手了；甚至在一些面试官看来，他们想要的标准答案也不过就是这个）。更深入的探索，且听我们下节分解~

}

14 Promise 命题思路全解析

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