【翻译】Promises/A+规范

于明昊 (/space/111344) 发表于 2014-07-27 11:49 28843 阅读

英文原文：Promise/A+ (https://promisesaplus.com/)

我的博客：Promise A+ 规范 (http://malcolmyu.github.io/malnote/2015/06/12/Promises-A-Plus/)

译者序：一年前曾译过 Promise/A+ 规范，适时完全不懂 Promise 的思想，纯粹将翻译的过程当作学习，旧文译下来诘屈聱牙，读起来十分不顺畅。谁知这样一篇拙译，一年之间竟然点击数千，成为谷歌搜索的头条。今日在理解之后重译此规范，以飨读者。

一个开放、健全且通用的 JavaScript Promise 标准。由开发者制定，供开发者参考。

译文术语

- **解决（fulfill）**：指一个 promise 成功时进行的一系列操作，如状态的改变、回调的执行。虽然规范中用 `fulfill` 来表示解决，但在后世的 promise 实现多以 `resolve` 来指代之。
- **拒绝（reject）**：指一个 promise 失败时进行的一系列操作。
- **终值（eventual value）**：所谓终值，指的是 promise 被**解决**时传递给解决回调的值，由于 promise 有**一次性**的特征，因此当这个值被传递时，标志着 promise 等待态的结束，故称之终值，有时也直接简称为值（value）。
- **据因（reason）**：也就是拒绝原因，指在 promise 被**拒绝**时传递给拒绝回调的值。

Promise 表示一个异步操作的最终结果，与之进行交互的方式主要是 `then` 方法，该方法注册了两个回调函数，用于接收 promise 的终值或本 promise 不能执行的原因。

本规范详细列出了 `then` 方法的执行过程，所有遵循 Promises/A+ 规范实现的 promise 均可以本标准作为参照基础来实施 `then` 方法。因而本规范是十分稳定的。尽管 Promise/A+ 组织有时可能会修订本规范，但主要是为了处理一些特殊的边界情况，且这些改动都是微小且向下兼容的。如果我们要进行大规模不兼容的更新，我们一定会在事先进行谨慎地考虑、详尽的探讨和严格的测试。

从历史上说，本规范实际上是把之前 Promise/A 规范 (http://wiki.commonjs.org/wiki/Promises/A) 中的建议明确成为了行为标准：我们一方面扩展了原有规范约定俗成的行为，一方面删减了原规范的一些特例情况和有问题的部分。

最后，核心的 Promises/A+ 规范不设计如何创建、解决和拒绝 promise，而是专注于提供一个通用的 `then` 方法。上述对于 promises 的操作方法将来在其他规范中可能会提及。

术语

Promise

promise 是一个拥有 `then` 方法的对象或函数，其行为符合本规范；

thenable

是一个定义了 `then` 方法的对象或函数，文中译作“拥有 `then` 方法”；

值（value）

指任何 JavaScript 的合法值（包括 `undefined`，thenable 和 promise）；

异常（exception）

是使用 `throw` 语句抛出的一个值。

据因（reason）

表示一个 promise 的拒绝原因。

要求

Promise 的状态

一个 Promise 的当前状态必须为以下三种状态中的一种：**等待态（Pending）**、**执行态（Fulfilled）**和**拒绝态（Rejected）**。

等待态（Pending）

处于等待态时，promise 需满足以下条件：

- 可以迁移至执行态或拒绝态

执行态（Fulfilled）

处于执行态时，promise 需满足以下条件：

- 不能迁移至其他任何状态
- 必须拥有一个**不可变**的终值

拒绝态（Rejected）

处于拒绝态时，promise 需满足以下条件：

- 不能迁移至其他任何状态
- 必须拥有一个**不可变**的据因

这里的不可变指的是恒等（即可用 `===` 判断相等），而不是意味着更深层次的不可变（**译者注：**盖指当 value 或 reason 不是基本值时，只要求其引用地址相等，但属性值可被更改）。

Then 方法

一个 promise 必须提供一个 `then` 方法以访问其当前值、终值和据因。

promise 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` 不是函数，其必须被忽略
- 如果 `onRejected` 不是函数，其必须被忽略

`onFulfilled` 特性

如果 `onFulfilled` 是函数：

- 当 `promise` 执行结束后其必须被调用，其第一个参数为 `promise` 的终值
- 在 `promise` 执行结束前其不可被调用
- 其调用次数不可超过一次

`onRejected` 特性

如果 `onRejected` 是函数：

- 当 `promise` 被拒绝执行后其必须被调用，其第一个参数为 `promise` 的据因
- 在 `promise` 被拒绝执行前其不可被调用
- 其调用次数不可超过一次

调用时机

`onFulfilled` 和 `onRejected` 只有在执行环境 (http://es5.github.io/#x10.3)堆栈仅包含**平台代码**时才可被调用 注1

调用要求

`onFulfilled` 和 `onRejected` 必须被作为函数调用（即没有 `this` 值） 注2

多次调用

`then` 方法可以被同一个 `promise` 调用多次

- 当 `promise` 成功执行时，所有 `onFulfilled` 需按照其注册顺序依次回调
- 当 `promise` 被拒绝执行时，所有的 `onRejected` 需按照其注册顺序依次回调

返回

`then` 方法必须返回一个 `promise` 对象 注3

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

- 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)`
- 如果 `onFulfilled` 或者 `onRejected` 抛出一个异常 `e` ，则 `promise2` 必须拒绝执行，并返回拒因 `e`
- 如果 `onFulfilled` 不是函数且 `promise1` 成功执行， `promise2` 必须成功执行并返回相同的值
- 如果 `onRejected` 不是函数且 `promise1` 拒绝执行， `promise2` 必须拒绝执行并返回相同的据因

译者注：理解上面的“返回”部分非常重要，即：**不论** `promise1` **被 reject 还是被 resolve 时** `promise2` **都会被 resolve，只有出现异常时才会被 rejected。**

Promise 解决过程

Promise 解决过程是一个抽象的操作，其需输入一个 `promise` 和一个值，我们表示为 `[[Resolve]](promise, x)`，如果 `x` 有 `then` 方法且看上去像一个 Promise ，解决程序即尝试使 `promise` 接受 `x` 的状态；否则其用 `x` 的值来执行 `promise` 。

这种 thenable 的特性使得 Promise 的实现更具有通用性：只要其暴露出一个遵循 Promise/A+ 协议的 `then` 方法即可；这同时也使遵循 Promise/A+ 规范的实现可以与那些不太规范但可用的实现能良好共存。

运行 `[[Resolve]](promise, x)` 需遵循以下步骤：

`x` 与 `promise` 相等

如果 `promise` 和 `x` 指向同一对象，以 `TypeError` 为据因拒绝执行 `promise`

`x` 为 Promise

如果 `x` 为 Promise ，则使 `promise` 接受 `x` 的状态 注4：

- 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
- 如果 `x` 处于执行态，用相同的值执行 `promise`
- 如果 `x` 处于拒绝态，用相同的据因拒绝 `promise`

`x` 为对象或函数

如果 `x` 为对象或者函数：

- 把 `x.then` 赋值给 `then` 注5
- 如果取 `x.then` 的值时抛出错误 `e` ，则以 `e` 为据因拒绝 `promise`
- 如果 `then` 是函数，将 `x` 作为函数的作用域 `this` 调用之。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`：
  - 如果 `resolvePromise` 以值 `y` 为参数被调用，则运行 `[[Resolve]](promise, y)`
  - 如果 `rejectPromise` 以据因 `r` 为参数被调用，则以据因 `r` 拒绝 `promise`
  - 如果 `resolvePromise` 和 `rejectPromise` 均被调用，或者被同一参数调用了多次，则优先采用首次调用并忽略剩下的调用
  - 如果调用 `then` 方法抛出了异常 `e`：
    - 如果 `resolvePromise` 或 `rejectPromise` 已经被调用，则忽略之
    - 否则以 `e` 为据因拒绝 `promise`
  - 如果 `then` 不是函数，以 `x` 为参数执行 `promise`
- 如果 `x` 不为对象或者函数，以 `x` 为参数执行 `promise`

如果一个 promise 被一个循环的 thenable 链中的对象解决，而 `[[Resolve]](promise, thenable)` 的递归性质又使得其被再次调用，根据上述的算法将会陷入无限递归之中。算法虽不强制要求，但也鼓励施者检测这样的递归是否存在，若检测到存在则以一个可识别的 `TypeError` 为据因来拒绝 `promise` 注6。

注释

- **注1** 这里的**平台代码**指的是引擎、环境以及 promise 的实施代码。实践中要确保 `onFulfilled` 和 `onRejected` 方法异步执行，且应该在 `then` 方法被调用的那一轮事件循环之后的新执行栈中执行。这个事件队列可以采用“宏任务（macro-task）”机制或者“微任务（micro-task）”机制来实现。由于 promise 的实施代码本身就是平台代码（**译者注：**即都是 JavaScript），故代码自身在处理在处理程序时可能已经包含一个任务调度队列。

  译者注：这里提及了 macrotask 和 microtask 两个概念，这表示异步任务的两种分类。在挂起任务时，JS 引擎会将所有任务按照类别分到这两个队列中，首先在 macrotask 的队列（这个队列也被叫做 task queue）中取出第一个任务，执行完毕后取出 microtask 队列中的所有任务顺序执行；之后再取 macrotask 任务，周而复始，直至两个队列的任务都取完。

  两个类别的具体分类如下：

  - **macro-task:** script（整体代码）, `setTimeout`, `setInterval`, `setImmediate`, I/O, UI rendering
  - **micro-task:** `process.nextTick`, `Promises`（这里指浏览器实现的原生 Promise）, `Object.observe`, `MutationObserver`

  详见 stackoverflow 解答 (http://stackoverflow.com/questions/25915634/difference-between-microtask-and-macrotask-within-an-event-loop-context) 或 这篇博客 (http://wengeezhang.com/?p=11)

- **注2** 也就是说在**严格模式（strict）**中，函数 `this` 的值为 `undefined`；在非严格模式中其为全局对象。
- **注3** 代码实现在满足所有要求的情况下可以允许 `promise2 === promise1` 。每个实现都要文档说明其是否允许以及在何种条件下允许 `promise2 === promise1` 。
- **注4** 总体来说，如果 `x` 符合当前实现，我们才认为它是真正的 promise 。这一规则允许那些特例实现接受符合已知要求的 Promises 状态。
- **注5** 这步我们先是存储了一个指向 `x.then` 的引用，然后测试并调用该引用，以避免多次访问 `x.then` 属性。这种预防措施确保了该属性的一致性，因为其值可能在检索调用时被改变。
- **注6** 实现不应该对 thenable 链的深度设限，并假定超出本限制的递归就是无限循环。只有真正的循环递归才应能导致 `TypeError` 异常；如果一条无限长的链上 thenable 均不相同，那么递归下去永远是正确的行为。

javascript (/tag/6) promise (/tag/4118) deffer (/tag/31966)

(/space/275823) 讲一个故事，Promise是一个是一个美好的承诺，承诺本身会做一些异步的事情，做完之后会返回成功失败，以方便下一个承诺的执行，当成功的时候，下一个Promise的resolve 会承接这个状态，当失败的时候，下一个Promise的reject会承接这个状态，如果下边的Promise没人会解决，那么这个Promise的失败状态，就得不到结局。就会在后边的Promise去查找设置了解决失败的reject函数，来执行解决失败状态。 解决完了以后会接着返回一个pending（Promise）可以执行调用then继续执行，这样 就是 resolve => throw error ->reject执行=>pending.then() 执行。不知道我这么讲对不～明昊大神赐教-一只小蜜蜂

厉害了这个码 (/space/275823) 发表于 2017-10-18 21:20:10 1 推荐

对的，但是有一点，解决完以后会返回一个 resolved or rejected 的 Promise

于明昊 (/space/111344) 发表于 2017-10-26 22:24:05

(/space/286234) 在译者注中“在挂起任务时，JS 引擎会将所有任务按照类别分到这两个队列中，首先在 macrotask 的队列（这个队列也被叫做 task queue）中取出第一个任务，执行完毕后取出 microtask 队列中的所有任务顺序执行；之后再取 macrotask 任务，周而复始，直至两个队列的任务都取完。”