

## 14 Promise 命题思路全解析

更新时间：2020-04-13 09:17:30



“

合理安排时间，就等于节约时间。——培根

”

**Promise** 是大家的老朋友了，不管是面试还是日常开发，咱们都离不开它。**Promise** 面试题千千万，但命题角度基本都逃不过以下三条线：

- 考察 **Promise** 特性（问答题）
- 给出一段 **Promise** 代码，问输出结果（这类题的关键在于搞清楚 **Promise** 中不同任务的执行时机）
- 深度考察 **Promise** 原理（终极版本就是让你手写一个 **Promise**）

掌握了这三类问题的答法，你已经算是把 **Promise** 翻了个底朝天。日后面对再深入刁钻的 **Promise** 问题，你都能做到有迹可循。

### 命题点一：Promise 特性类问题

**Promise**特性问答题，主要是为了考察你对 **Promise** 基础知识的熟悉度。大家需要关注的是这“灵魂三问”：

问：说说你理解的 **Promise**

答题思路：这道题的表述方式有很多，大家注意答题时务必囊括以下几个要点：代理对象、三个状态、状态切换机制

课代表示范一下：

**Promise** 对象是一个代理对象。它接受你传入的 **executor**（执行器）作为入参，允许你把异步任务的成功和失败分别绑定到对应的处理方法上去。一个 **Promise** 实例有三种状态：

- **pending** 状态，表示进行中。这是 **Promise** 实例创建后的一个初始态；
- **fulfilled** 状态，表示成功完成。这是我们在执行器中调用 **resolve** 后，达成的状态；
- **rejected** 状态，表示操作失败、被拒绝。这是我们在执行器中调用 **reject** 后，达成的状态；

**Promise**实例的状态是可以改变的，但它只允许被改变一次。当我们的实例状态从 **pending** 切换为 **rejected** 后，就无法再扭转为 **fulfilled**，反之同理。当 **Promise** 的状态为 **resolved** 时，会触发其对应的 **then** 方法入参里的 **onfulfilled** 函数；当 **Promise** 的状态为 **rejected** 时，会触发其对应的 **then** 方法入参里的 **onrejected** 函数。

问：**Promise** 的出现是为了解决什么问题？

这个问题，现在要还是答不出来，说明上一个小节读得不够认真啊。。。

问：**Promise** 常见方法有哪些？各自是干嘛的？

答题思路：这道题需要答出 **all**、**race**、**reject** 和 **resolve**。

课代表示范一下：

**Promise**的方法有以下几种：

- **Promise.all(iterable)**：这个方法返回一个新的 **promise** 对象，该 **promise** 对象在 **iterable** 参数对象里所有的 **promise** 对象都成功的时候才会触发成功，一旦有任何一个 **iterable** 里面的 **promise** 对象失败则立即触发该 **promise** 对象的失败。

举个?：

```
var p1 = Promise.resolve("1号选手");
var p2 = '2号选手';
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, "3号选手");
});
Promise.all([p1, p2, p3]).then(values => {
  console.log(values); // ["1号选手", "2号选手", "3号选手"]
});
```

- **Promise.race(iterable)**：当 **iterable** 参数里的任意一个子 **promise** 被成功或失败后，父 **promise** 马上也会用子 **promise** 的成功返回值或失败详情作为参数调用父 **promise** 绑定的相应处理函数，并返回该 **promise** 对象。

举个?：

```
var p1 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 100, "1号选手");
});
var p2 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 50, "2号选手");
});

// 这里因为 2 号选手返回得更早，所以返回值以 2 号选手为准
Promise.race([p1, p2]).then(function(value) {
  console.log(value); // "2号选手"
});
```

- **Promise.reject(reason)**：返回一个状态为失败的**Promise**对象，并将给定的失败信息传递给对应的处理方法
- **Promise.resolve(value)**：它返回一个 **Promise** 对象，但是这个对象的状态由你传入的**value**决定，情形分以下

两种：

- 如果传入的是一个带有 `then` 方法的对象（我们称为 `thenable` 对象），返回的 `Promise` 对象的最终状态由 `then` 方法执行决定
- 否则的话，返回的 `Promise` 对象状态为 `fulfilled`，同时这里的 `value` 会作为 `then` 方法中指定的 `onfulfilled` 的入参

## 命题点二：看代码说答案类问题

能说出 `Promise` 是啥，不代表你就真正理解它、能用它干活了。为了看看你到底几斤几两，面试官此时往往会 `show you the code`。咱们现在就通过 3 道好题，一起看看这些 `code` 类题目的命题规律：

真题1：

```
const promise = new Promise((resolve, reject) => {
  console.log(1);
  resolve();
  console.log(2);
});

promise.then(() => {
  console.log(3);
});

console.log(4);
```

运行结果：

```
1
2
4
3
```

考点点拨：**Promise** 中的处理函数是异步任务

`then` 方法中传入的任务是一个异步任务。`resolve()` 这个调用，作用是将 `Promise` 的状态从 `pending` 置为 `fulfilled`，这个新状态会让 `Promise` 知道“我的 `then` 方法中那个任务可以执行了”——注意是“可以执行了”，而不是说“立刻就会执行”。毕竟作为一个异步任务，它的基本修养就是要等同步代码执行完之后再执行。所以说数字 3 的输出排在最后。

真题2：

```
const promise = new Promise((resolve, reject) => {
  resolve('第 1 次 resolve')
  console.log('resolve后的普通逻辑')
  reject('error')
  resolve('第 2 次 resolve')
})

promise
  .then((res) => {
    console.log('then: ', res)
  })
  .catch((err) => {
    console.log('catch: ', err)
  })
```

运行结果：

```
resolve后的普通逻辑
then: 第 1 次 resolve
```

考点点拨：**Promise** 对象的状态只能被改变一次

分析：这段代码里，**promise** 初始状态为 **pending**，我们在函数体第一行就用 **resolve** 把它置为了 **fulfilled** 态。这个切换完成后，后续所有尝试进一步作状态切换的动作全部不生效，所以后续的 **reject**、**resolve** 大家直接忽略掉就好；需要注意的是，我们忽略的是第一次 **resolve** 后的 **reject**、**resolve**，而不是忽略它身后的所有代码。因此 **console.log('resolve后的普通逻辑')** 这句，仍然可以正常被执行。至于这里为啥它输出在 “**then: 第 1 次 resolve**” 的前面，原因和真题1是一样一样的~

真题3:

```
Promise.resolve(1)
  .then(Promise.resolve(2))
  .then(3)
  .then()
  .then(console.log)
```

运行结果：

```
1
```

考点点拨：**Promise** 值穿透问题

分析：大家知道，**then** 方法里允许我们传入两个参数：**onFulfilled**（成功态的处理函数）和 **onRejected**（失败态的处理函数）。

你可以两者都传，也可以只传前者或者后者。但是无论如何，**then** 方法的入参只能是函数。万一你想塞给它一些乱七八糟的东西，它就会“翻脸不认人”。

具体到我们这个题里，第一个 **then** 方法中传入的是一个 **Promise** 对象，**then** 说：“我不认识”；第二个 **then** 中传入的是一个数字，**then** 继续说“我不认识”；第四个干脆啥也没穿，**then** 说“入参 **undefined** 了，拜拜”；直到第五个入参，一个函数被传了进来，**then** 哭了：“终于等到一个我能处理的！”，于是只有最后一个入参生效了。

在这个过程中，我们最初 **resolve** 出来那个值，穿越了一个又一个无效的 **then** 调用，就好像是这些 **then** 调用都是透明的、不存在的一样，因此这种情形我们也形象地称它是 **Promise** 的“值穿透”。

## 命题点三：**Promise** 底层原理考察

涉及到 **Promise** 的底层原理问题，有的面试官会让你描述 **Promise/A+** 方案的实现思路，更多的面试官会倾向于要求你手写一个简单的 **Promise** 看看——毕竟 **talk is cheap** 嘛。

不过这类问题也好解决，既然手写 **Promise** 是它的终极形态，咱们就来解决这个终极形态。在手写 **Promise** 的过程中，**Promise** 的底层原理、包括一些实现细节都会被你潜移默化地吸收掉，日后它即便是以问答题的形式出现，你也自然无所畏惧了。因此在下一节，我们会用一个完整的章节手把手教大家来写一个 **Promise**。

```
}
```

