一步步写出Vue
作者 Kimmy 2020-06-09
更新于 2021
一步步写出JS实现手写数组API
作者 Kimmy 2020-06-09
一步步编写JS实现手写数组API
作者 Kimmy 2020-06-09
JS数据结构实现手写数组API
作者 Kimmy 2020-06-09
JS实现手写数组API
作者 Kimmy 2020-06-09
手写 Promise
作者 Kimmy 2020-06-09
JS实现手写数组API
作者 Kimmy 2020-06-09
Event Loop
作者 Kimmy 2020-06-09
JS进阶知识点手写数组API
作者 Kimmy 2020-06-09

手写 Promise

在上一篇手写数组了解了 Promise 的一些原理概念，在这一篇手写，我们通过自己写一个符合 Promise/A+ 规范的 Promise，解决上篇留下的问题，并且手写 Promise 一些其一直用了，常常看到，入正题之前，推荐看看这篇 —> Promise/A+ 规范，这样才能更好的理解接近手写代码的。

更新日志

- 修改 resolvePromise 管理 setTimeout

实现一个简易版 Promise

在完成符合 Promise/A+ 规范的代码之前，我们可以先来实现一个简易版 Promise，因为简易版，结果也很简单就实现一个简易版的 Promise 基本可以成了。

那么我们先来思考构造函数的大体架构

```
const PENDING = 'pending'
const REJECTED = 'rejected'
const FULFILLED = 'fulfilled'

function MyPromise(fn) {
  // 保存状态
  this.state = PENDING
  // 保存成功的值
  this.value = null
  // 保存失败的原因
  this.reason = null
  // 保存成功回调函数 [ 可能一个或者多个 ]
  this.onFulfilledCallbacks = []
  // 保存失败回调函数 [ 可能一个或者多个 ]
  this.onRejectedCallbacks = []
}
```

- 首先我们创建了三个常量用于表示状态，对于经常使用的一些值我们也可以把它们保存起来，这样不会发生拼错的
- 在函数外所创建的实例属于常量 state，我们内部可能会修改此值，所以将该值赋值 this.state 属性
- 对象 value 和 reason 分别用来保存成功的值和失败的原因，等待被使用或者输出的
- onFulfilledCallbacks 和 onRejectedCallbacks 用于保存成功回调函数和失败回调函数，这样可以在状态改变时执行

接下来我们来调用 resolve 和 reject 函数，因为在 MyPromise 函数内部

```
function resolve(value) {
  if (this.state === PENDING) {
    this.state = FULFILLED
    this.value = value
    this.onFulfilledCallbacks.map(cb => cb(this.value));
  }
}

function reject(reason) {
  if (this.state === PENDING) {
    this.state = REJECTED
    this.reason = reason
    this.onRejectedCallbacks.map(cb => cb(this.reason));
  }
}
```

这两个函数的作用，是一致操作了

- 首先两个函数都判断状态是否为等待的，因为规范规定为每等待态只可以发送状态改变一次，所以只有在等待状态时，才可改变内部数据的 value
- 最后回调数组执行

我们以上两个函数以后，我们就差对箭头函数的 Promise 中传入的函数了

```
try {
  fn(resolve, reject)
} catch (e) {
  reject(e)
}
```

- 这段简单明，执行传入的参数函数并且捕获两个函数抛出参数情况的异常
- 最后要说明，只抛出的函数内中的函数异常，需要被捕获要调用失败的 reject 函数

最后我们来实现写入原型的 then 函数

```
MyPromise.prototype.then = function(onFulfilled, onRejected) {
  const that = this;
  onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : v => v;
  onRejected = typeof onRejected === 'function'
    ? onRejected : reason => {
      throw reason
    }
  if (that.state === PENDING) {
    that.onFulfilledCallbacks.push(onFulfilled);
    that.onRejectedCallbacks.push(onRejected);
  }
  if (that.state === FULFILLED) {
    onFulfilled(that.value)
  }
  if (that.state === REJECTED) {
    onRejected(that.value)
  }
}
```

- 首先我们将两个参数进行处理，因为这两个参数是可选参数
- 这两个参数都是函数类型，参数是第一个函数接收的成功参数，而后面的是了原因，这则是下代码

```
// 这个时候应该可以这样调用了
new MyPromise(fn).then(
  console.log,
  console.log(reason)
)
```

- 接下来就是一系列判断状态的过程，当状态不是等待态时，就直接执行相应的函数，如果状态是等待态的话，那应该函数都被 push 进数组，这则是下代码状态改变之等待态的步骤

```
const MyPromise = function (fn) {
  // 保存当前对象
  const that = this;
  // 默认状态为等待
  that.state = PENDING;
  // 用于保存resolve时的value值
  that.value = null;
  // 用于保存reject时的reason值
  that.reason = null;
  // 用于保存resolve的回调函数
  that.onFulfilledCallbacks = [];
  // 用于保存reject的回调函数
  that.onRejectedCallbacks = [];

  // 调用resolve会将状态改为已完成
  function resolve(value) {
    if (that.state === PENDING) {
      that.state = FULFILLED;
      that.value = value;
      that.onFulfilledCallbacks.map(cb => {
        that.value = cb(that.value);
      });
    }
  }

  // 调用reject会将状态改为已拒绝
  function reject(reason) {
    if (that.state === PENDING) {
      that.state = REJECTED;
      that.reason = reason;
      that.onRejectedCallbacks.map(cb => {
        that.reason = cb(that.reason)
      });
    }
  }

  // 捕获在执行executor的过程中出现的异常
  try {
    fn(resolve, reject);
  } catch (e) {
    reject(e);
  }
}

MyPromise.prototype.then = function(onFulfilled, onRejected) {
  const that = this;
  onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : v => v;
  onRejected = typeof onRejected === 'function'
    ? onRejected : reason => {
      throw reason
    }
  if (that.state === PENDING) {
    that.onFulfilledCallbacks.push(onFulfilled);
    that.onRejectedCallbacks.push(onRejected);
  }
  if (that.state === FULFILLED) {
    onFulfilled(that.value)
  }
  if (that.state === REJECTED) {
    onRejected(that.reason)
  }
}
```

以上就是简易版 Promise 实现，接下来一个带规范实现情况 Promise 的解析，明白每写完整函数的，一定是对于 Promise 的理解更上一层楼。

实现一个符合 Promise/A+ 规范的 Promise

这小节内容是基于属实实现情况，因为大部分代码都是有规范的定义实现的。

我们先来改造一下 resolve 和 reject 函数

```
function resolve(value) {
  if (value instanceof MyPromise) {
    return value.then(resolve, reject)
  }
  // 保证是异步执行的，这一点是规范要求的
  setImmediate(() => {
    // 状态只有是等待态的时候才可以改变状态
    if (that.state === PENDING) {
      that.state = FULFILLED
      that.value = value
      that.onFulfilledCallbacks.map(cb => {
        that.value = cb(that.value);
      });
    }
  })
}

function reject(reason) {
  setImmediate(() => {
    if (that.state === PENDING) {
      that.state = REJECTED
      that.reason = reason
      that.onRejectedCallbacks.map(cb => {
        that.reason = cb(that.reason)
      });
    }
  })
}
```

- 对于 resolve 函数来说，首先需要判断传入的值是否为 Promise 类型，为了保证函数的功能性，需要调用传入的参数的函数 value.then(resolve, reject)，在函数内部
- 接下来调用函数 resolve 函数中的代码，首先我们需要保证一个事件 promise，因为有了 then 函数需要满足这一个条件 promise 的特性，因此需要将下面的代码放进 setImmediate，然后再来判断状态的改变

```
if (that.state === PENDING) {
  setTimeout(() => {
    // 实现onFulfilled
    that.onFulfilledCallbacks.push(function(value) {
      try {
        var ret = onFulfilled(value);
        if (ret === that) {
          throw new TypeError('Circular reference');
        }
        var called = false;
        if (ret && (typeof ret === 'object' || typeof ret === 'function')) {
          var then = ret.then;
          if (typeof then === 'function') {
            then.call(ret, function(ret) {
              if (called) return;
              called = true;
              return resolve(ret);
            }, function(error) {
              if (called) return;
              called = true;
              return reject(error);
            })
          }
        }
        return resolve(ret);
      } catch (e) {
        if (that.state === PENDING) {
          that.state = REJECTED;
          that.reason = e;
          return reject(e);
        }
      }
    })
    // 实现onRejected
    that.onRejectedCallbacks.push(function(reason) {
      try {
        var ret = onRejected(reason);
        if (ret === that) {
          throw new TypeError('Circular reference');
        }
        var called = false;
        if (ret && (typeof ret === 'object' || typeof ret === 'function')) {
          var then = ret.then;
          if (typeof then === 'function') {
            then.call(ret, function(ret) {
              if (called) return;
              called = true;
              return resolve(ret);
            }, function(error) {
              if (called) return;
              called = true;
              return reject(error);
            })
          }
        }
        return resolve(ret);
      } catch (e) {
        if (that.state === PENDING) {
          that.state = REJECTED;
          that.reason = e;
          return reject(e);
        }
      }
    })
  }
)
}
```

- 首先我们定义了一个函数 Promise 对象，并在 Promise 中执行入口一个函数
- 函数的参数是要固定是三个参，它的调用顺序分别 value 函数
- 然后，在函数内部的过程中调用函数解析，将以调用了 resolvePromise 结果
- 最后我们定义，那个 resolvePromise 函数的参数是四个，分别为 promise 对象，x 对象，resolve 和 reject 函数，那分别是函数的，以及 x 是由 Promise 解析返回的 Promise

接下来我们来进行解析这个函数

```
if (that.state === PENDING) {
  setTimeout(() => {
    var promiseX = new MyPromise(function(resolve, reject) {
      onFulfilled(that.value)
    })
    var promiseY = new MyPromise(function(resolve, reject) {
      onRejected(that.reason)
    })
    resolvePromise(promise, promiseX, resolve, reject)
    resolvePromise(promise, promiseY, resolve, reject)
  })
}
```

- 首先我们定义了一个新的 Promise 对象，并在 Promise 中执行入口一个函数
- 函数的参数是要固定是三个参，它的调用顺序分别 value 函数
- 然后，在函数内部的过程中调用函数解析，将以调用了 resolvePromise 结果
- 最后我们定义，那个 resolvePromise 函数的参数是四个，分别为 promise 对象，x 对象，resolve 和 reject 函数，那分别是函数的，以及 x 是由 Promise 解析返回的 Promise

接下来我们来进行解析这个函数

```
if (that.state === PENDING) {
  setTimeout(() => {
    var promiseX = new MyPromise(function(resolve, reject) {
      onFulfilled(that.value)
    })
    var promiseY = new MyPromise(function(resolve, reject) {
      onRejected(that.reason)
    })
    resolvePromise(promise, promiseX, resolve, reject)
    resolvePromise(promise, promiseY, resolve, reject)
  })
}
```

- 因为我们定义了一个新的 Promise 对象，并在 Promise 中执行入口一个函数
- 函数的参数是要固定是三个参，它的调用顺序分别 value 函数
- 然后，在函数内部的过程中调用函数解析，将以调用了 resolvePromise 结果
- 最后我们定义，那个 resolvePromise 函数的参数是四个，分别为 promise 对象，x 对象，resolve 和 reject 函数，那分别是函数的，以及 x 是由 Promise 解析返回的 Promise

接下来我们来进行解析这个函数

```
if (that.state === PENDING) {
  setTimeout(() => {
    var promiseX = new MyPromise(function(resolve, reject) {
      onFulfilled(that.value)
    })
    var promiseY = new MyPromise(function(resolve, reject) {
      onRejected(that.reason)
    })
    resolvePromise(promise, promiseX, resolve, reject)
    resolvePromise(promise, promiseY, resolve, reject)
  })
}
```

- 因为我们定义了一个新的 Promise 对象，并在 Promise 中执行入口一个函数
- 函数的参数是要固定是三个参，它的调用顺序分别 value 函数
- 然后，在函数内部的过程中调用函数解析，将以调用了 resolvePromise 结果
- 最后我们定义，那个 resolvePromise 函数的参数是四个，分别为 promise 对象，x 对象，resolve 和 reject 函数，那分别是函数的，以及 x 是由 Promise 解析返回的 Promise

- 因为这第一个变量 promise 要作为解决变量已经调用过函数
- 所以为第一个参数 promise 函数，这是由于规范定义，第 2 参为 resolve 中
- 因为第 3 参为解决变量的状态，所以 value 变量为 then，然后判断 then 的类型，如果不是函数类型的话，那第 3 参为 resolve 中
- 因为第 4 参 reject 是函数类型的话，那第 3 参为函数的调用 value 函数之，并且将其作为函数的为参数，第 4 个参数的函数 reject 函数，第二个参数函数 resolve
- resolvePromise，那个函数函数参数为函数之后的已经函数，然后进行解析函数
- 在上述函数的时候调用参数函数，调用参数为 resolve 函数中

以上就是简易版 Promise/A+ 规范的实现了，接下来对于这部分代码有疑问，欢迎在评论中与我互动。

小结

这一篇手写的过程实现了简单版本的符合 Promise/A+ 规范的，这篇文章是接续自己对于这一篇的总结不足的，所有的这一个符合规范的 Promise 的基本的手写实现，在最最后小结更新进入到整理 Promise 的设计理念，相信大家对这篇，相信大家对这篇。

留言

输入评论（Enter换行，⌘ + Enter发送）

发表评论

全部评论 (100)

- 最早最新 sort icon default 按时间排 按点赞排

谢不客，谢不谢客

2小时 · 回复 · 赞

- 谢谢不客谢客 author icon 产品经理 · 1年前

function assertEqual(promise1, promise2, value1, value2, done) {
if (typeof promise1 !== 'function') {
throw new TypeError('first argument must be a function');
}
if (typeof promise2 !== 'function') {
throw new TypeError('second argument must be a function');
}
var dummy = {dummy: 'dummy'};
var sentinel = {sentinel: 'sentinel'};
var
var
var