《前端高手进阶》

朱德龙 前中兴软创主任工程师

— 拉勾教育出品 —

加餐02：手写 Promise、async/await

Promise/A+ 规范

https://promisesaplus.com/

Promise 是一个**对象或者函数**

对外提供了一个 then 函数，内部拥有 3 个状态

Promise/A+ 规范

then 函数

```
promise.then(onFulfilled, onRejected)
```

- 如果可选参数不为函数时应该被忽略
- 两个函数都应该是异步执行的，即放入事件队列等待下一轮 tick，而非立即执行
- 当调用 onFulfilled 函数时，会将当前 Promise 的值作为参数传入
- 当调用 onRejected 函数时，会将当前 Promise 的失败原因作为参数传入
- then 函数的返回值为 Promise

Promise/A+ 规范

Promise 状态

- **pending**: “等待”状态，可以转移到 fulfilled 或者 rejected 状态
- **fulfilled**: “执行”（或“履行”）状态，是 Promise 的最终态
  表示执行成功，该状态下不可再改变
- **rejected**: “拒绝”状态，是 Promise 的最终态
  表示执行失败，该状态不可再改变

Promise/A+ 规范

Promise 解决过程

Promise 解决过程是一个抽象的操作，即接收一个 promise 和一个值 x

目的就是对 Promise 形式的执行结果进行统一处理

Promise 解决过程

情况 1： x 等于 promise

抛出一个 TypeError 错误，拒绝 promise

Promise 解决过程

情况 2：x 为 Promise 的实例

如果 x 处于等待状态，那么 promise 继续等待至 x 执行或拒绝

否则根据 x 的状态执行/拒绝 promise

Promise 解决过程

情况 3：x 为对象或函数

该情况的核心是取出 x.then 并调用，在调用的时候将 this 指向 x

将 then 回调函数中得到结果 y 传入新的 Promise 解决过程中

形成一个递归调用

其中，如果执行报错，则以对应的错误为原因拒绝 promise

Promise/A+ 规范

Promise 解决过程

情况 4：如果 x 不为对象或函数

以 x 作为值，执行 promise

Promise 实现

Promise() 函数及状态

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

Promise 实现

Promise() 函数及状态

```
function Promise(execute) {
  var self = this;
  self.state = PENDING;
  function resolve(value) {
    if (self.state === PENDING) {
      self.state = FULFILLED;
      self.value = value;
    }
```

Promise 实现

Promise() 函数及状态

```
function reject(reason) {
  if (self.state === PENDING) {
    self.state = REJECTED;
    self.reason = reason;
  }
}
try {
  execute(resolve, reject);
} catch (e) {
```

Promise 实现

Promise() 函数及状态

```
    self.reason = reason;
  }

  try {
    execute(resolve, reject);
  } catch (e) {
    reject(e);
  }
```

Promise 实现

then() 函数

```
Promise.prototype.then = function (onFulfilled, onRejected) {
}
```

Promise 实现

then() 函数

```
onFulfilled = typeof onFulfilled === "function" ? onFulfilled : function (x) {
  return x
};
onRejected = typeof onRejected === "function" ? onRejected : function (e) {
  throw e
};
```

Promise 实现

then() 函数

```
var self = this;
switch (self.state) {
 case FULFILLED:
  setTimeout(function () {
   onFulfilled(self.value);
  })
  break;
 case REJECTED:
  setTimeout(function () {
   onRejected(self.reason);
```

Promise 实现

then() 函数

```
    break;
  case REJECTED:
    setTimeout(function () {
      onRejected(self.reason);
    })
    break;
  case PENDING:
    // todo
    break;
}
```

Promise 实现

then() 函数

```
case PENDING:
  self.onFulfilledFn = function () {
    onFulfilled(self.value);
  }
  self.onRejectedFn = function () {
    onRejected(self.reason);
  }
  break;
```

Promise 实现

then() 函数

```
case FULFILLED:
 promise = new Promise(function (resolve, reject) {
  setTimeout(function () {
   try {
    onFulfilled(self.value);
   } catch (e) {
    reject(e)
   }
  })
 });
 break;
```

Promise 实现

then() 函数

```
case PENDING:
  promise = new Promise(function (resolve, reject) {
   self.onFulfilledFn.push(function () {
    try {
     onFulfilled(self.value);
    } catch (e) {
     reject(e)
    }
   });
```

Promise 实现

then() 函数

```
self.onRejectedFn.push(function () {
  try {
    onRejected(self.reason);
  } catch (e) {
    reject(e)
  }
});
break;
```

Promise 实现

then() 函数

```
function Promise(execute) {
  ...
  self.onFulfilledFn = [];
  self.onRejectedFn = [];
  ...
  function resolve(value) {
   setTimeout(function() {
     ...
     self.onFulfilledFn.forEach(function (f) {
      f(self.value)
```

Promise 实现

then() 函数

```
    })
  })
}
function reject(reason) {
  setTimeout(function() {
    ...
    self.onRejectedFn.forEach(function (f) {
      f(self.reason)
    })
  })
}
```

Promise 实现

拉勾教育
互联网人实战大学

then() 函数

```
}
function reject(reason) {
 setTimeout(function() {
   ...
   self.onRejectedFn.forEach(function (f) {
    f(self.reason)
   })
 })
}
}
```

Promise 实现

resolvePromise() 函数

```
function resolvePromise(promise, x) {
  if (promise === x) {
    return reject(new TypeError("x ä??è??ä?? promise ??"));
  }
}
```

Promise 实现

怎么让 promise 接受 x 的状态 ?

直接改变 promise 状态肯定是不可取的

首先状态信息属于内部变量

其次也无法调用属性 onResolvedFn 和 onFulfilledFn 中的待执行函数

所以必须要通过调用 promise 在构造时的函数 resolve() 和 reject() 来改变

Promise 实现

怎么让 promise 接受 x 的状态

如果 x 处于等待状态，那么 promise 继续保持等待状态

等待解决过程函数 resolvePromise() 执行

否则应该用相同的值执行或拒绝 promise

我们无法从外部拒绝或执行一个 Promise 实例

只能通过调用构造函数传入的 resolve() 和 reject() 函数来实现

所以还需要把这两个函数作为参数传递到 resolvePromise 函数中

Promise 实现


```
function resolvePromise(promise, x, resolve, reject) {
  ...
  if (x instanceof Promise) {
    if (x.state === FULFILLED) {
      resolve(x.value)
    } else if (x.state === REJECTED) {
      reject(x.reason)
    } else {
```

Promise 实现


```
    reject(x.reason)
  } else {
    x.then(function (y) {
      resolvePromise(promise, y, resolve, reject)
    }, reject)
  }
 }
}
```

Promise 实现


```
if ((x !== null) && ((typeof x === 'object') || (typeof x === 'function'))) {
  var executed;
  try {
    var then = x.then;
    if (typeof then === "function") {
      then.call(x, function (y) {
        if (executed) return;
        executed = true;
        return resolvePromise(promise, y, resolve, reject)
      }, function (e) {
```

Promise 实现


```
      executed = true;
      return resolvePromise(promise, y, resolve, reject)
    }, function (e) {
      if (executed) return;
      executed = true;
      reject(e);
    })
  } else {
    resolve(x);
  }
} catch (e) {
```

Promise 实现


```
      reject(e);
    })
  } else {
    resolve(x);
  }
} catch (e) {
  if (executed) return;
  executed = true;
  reject(e);
}
}
```

Promise 实现


```
resolve(x);
```

Promise 测试

专门测试 Promise 规范性的模块 **promises-aplus-tests**

该模块内置了数百个测试案例，支持命令行**一键测试**

只是在导出模块的时候需要遵循 CommonJS 规范

并且按照要求导出对应的函数

https://github.com/yalishizhude/course/tree/master/plus2

Promise 测试


```
  872 passing (17s)

Done in 17.00s.
```

async/await

拉勾教育
互联网人实战大学

async 是 ES2017 标准推出的用于处理异步操作的关键字

从本质上来说就是 **Generator 函数的语法糖**

什么是 Generator 函数?

拉勾教育
互联网人实战大学

Generator 函数是 ES6 提出的除 Promise 之外的另一种**异步解决方案**

什么是 Generator 函数?


```
function* fn(){
  ...
}
```

什么是 Generator 函数?

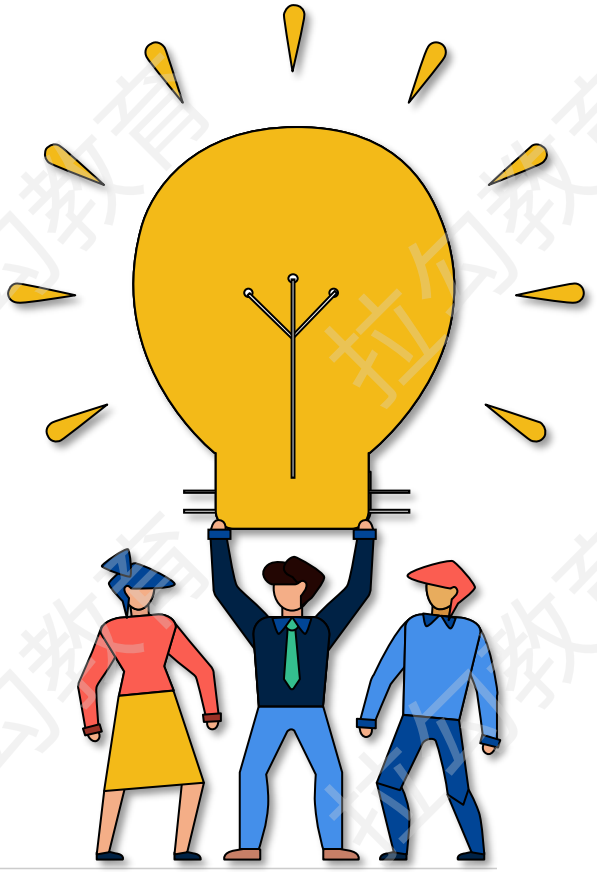
- 函数体内部使用 **yield** 表达式，定义不同的内部状态
- 当函数体外部调用迭代器的 next() 函数时

  函数会执行到下一个 yield 表达式的位置，并返回一个对象

  该对象包含属性 value 和 done

  value 是调用 next() 函数时传入的参数

  done 为布尔值表示是否执行完成



什么是 Generator 函数?

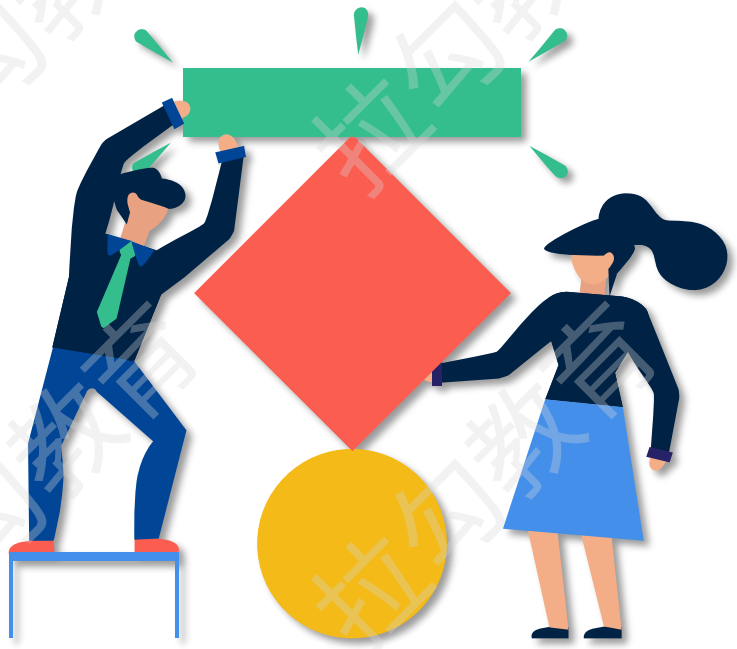

```
function asyncFn(cb) {
  setTimeout(cb, 1000, 1)
}
function* fn() {
  var result = yield asyncFn(function(data) {
    it.next(data);
  })
  console.log(result)  // 1
}
var it = fn()
it.next()
```

async/await 原理

Generator 函数带来了一些麻烦

相较之下 Promise 是更优秀的异步解决方案

async/await 做的事情就是将 Generator 函数转换成 Promise



async/await 原理


```
function generator2promise(generatorFn){
 return function (){
  var gen = generatorFn.apply(this, arguments);
  return new Promise(function (resolve, reject) {
   function step(key, arg) {
    try{
     var info = gen[key](arg);
     var value = info.value;
    } catch (error) {
     reject(error);
     return;
```

async/await 原理


```
    return;
  }
  if (info.done) {
    resolve(value);
  } else {
    return Promise.resolve(value).then(function
(value) {
      step("next", value);
    }, function (err) {
      step("throw", err);
    });
```

async/await 原理


```
(value){
        step("next", value);
      }, function (err) {
        step("throw", err);
      });
    }

    return step("next");
  });
 };
}
```

总结

通过代码实例深入分析了 Promise/A+ 规范以及 async/await 的实现原理

对于手写 Promise 的过程

先理解清楚规范，然后根据规范一步一步地去实现和优化

对于 async/await 语法糖

结合 Generator 函数，理解其封装原理即可

总结

拉勾教育
互联网人实战大学

试着自己动手写一个 Promise，看看能否通过测试用例

Next：第14讲《HTTP 协议和它的“补丁”们》

拉勾教育

—互联网人实战大学—

下载「拉勾教育App」
获取更多内容

L / A / G / O / U