

## 起步构建

本章来自自己开发一个 Promise 实现，提升异步编程的能力。

首先声明定义类并声明 Promise 状态与值，有以下几个细节需要注意。

- executor 为执行者
- 当执行者出现异常时触发拒绝状态
- 使用静态属性保存状态值
- 状态只能改变一次，所以在 resolve 与 reject 添加条件判断
- 因为

resolve或

rejected方法在 executor 中调用，作用域也是 executor 作用域，这会造成 this 指向 window，现在我们使用的是 class 定义，this 为 undefined。

```
1 class HD {
2   static PENDING = "pending";
3   static FULFILLED = "fulfilled";
4   static REJECTED = "rejected";
5   (executor) {
6     this.status = HD.PENDING;
7     this.value = null;
8     try {
9       executor(this.resolve.bind(this), this.reject.bind(this));
10    } catch (error) {
11      this.reject(error);
12    }
13  }
14  resolve(value) {
15    if (this.status === HD.PENDING) {
16      this.status = HD.FULFILLED;
17      this.value = value;
18    }
19  }
20  reject(value) {
21    if (this.status === HD.PENDING) {
22      this.status = HD.REJECTED;
23      this.value = value;
24    }
25  }
26 }
```

```
26 }  
27
```

下面测试一下状态改变

```
1 <script src="HD.js"></script>  
2 <script>  
3   let p = new HD((resolve, reject) => {  
4     resolve("后盾人");  
5   });  
6   console.log(p);  
7 </script>  
8
```

## #THEN

现在添加 then 方法来处理状态的改变，有以下几点说明

1. then 可以有两个参数，即成功和错误时的回调函数
2. then 的函数参数都不是必须的，所以需要设置默认值为函数，用于处理当没有传递时情况
3. 当执行 then 传递的函数发生异常时，统一交给 onRejected 来处理错误

## #基础构建

```
1 then(onFulfilled, onRejected) {  
2   if (typeof onFulfilled !== "function") {  
3     onFulfilled = value => value;  
4   }  
5   if (typeof onRejected !== "function") {  
6     onRejected = value => value;  
7   }  
8   if (this.status == HD.FULFILLED) {  
9     try {  
10      onFulfilled(this.value);  
11    } catch (error) {  
12      onRejected(error);  
13    }  
14  }  
15  if (this.status == HD.REJECTED) {
```

```

16     try {
17         onRejected(this.value);
18     } catch (error) {
19         onRejected(error);
20     }
21 }
22 }
23

```

下面来测试 then 方法的，结果正常输出后盾人

```

1 let p = new HD((resolve, reject) => {
2     resolve("后盾人");
3 }).then(
4     value => {
5         console.log(value);
6     },
7     reason => {
8         console.log(reason);
9     }
10 );
11 console.log("houdunren.com");
12

```

## #异步任务

但上面的代码产生的 Promise 并不是异步的，使用 setTimeout 来将 onFulfilled 与 onRejected 做为异步宏任务执行

```

1 then(onFulfilled, onRejected) {
2     if (typeof onFulfilled !== "function") {
3         onFulfilled = value => value;
4     }
5     if (typeof onRejected !== "function") {
6         onRejected = value => value;
7     }
8     if (this.status == HD.FULFILLED) {
9         setTimeout(() => {
10             try {

```

```

11         onFulfilled(this.value);
12     } catch (error) {
13         onRejected(error);
14     }
15 });
16 }
17 if (this.status === HD.REJECTED) {
18     setTimeout(() => {
19         try {
20             onRejected(this.value);
21         } catch (error) {
22             onRejected(error);
23         }
24     });
25 }
26 }
27

```

现在再执行代码，已经有异步效果了，先输出了[houdunren.com](http://houdunren.com)

```

1 let p = new HD((resolve, reject) => {
2     resolve("后盾人");
3 }).then(
4     value => {
5         console.log(value);
6     },
7     reason => {
8         console.log(reason);
9     }
10 );
11 console.log("houdunren.com");
12

```

## #PENDING 状态

目前 then 方法无法处理 promise 为 pending 时的状态

```

1 ...
2 let p = new HD((resolve, reject) => {

```

```

3   setTimeout(() => {
4     resolve("后盾人");
5   });
6 })
7 ...
8

```

为了处理以下情况，需要进行几点改动

1. 在构造函数中添加 callbacks 来保存 pending 状态时处理函数，当状态改变时循环调用

```

1 (executor) {
2   ...
3   this.callbacks = [];
4   ...
5 }
6

```

2. 将 then 方法的回调函数添加到 callbacks 数组中，用于异步执行

```

1 then(onFulfilled, onRejected) {
2   if (typeof onFulfilled !== "function") {
3     onFulfilled = value => value;
4   }
5   if (typeof onRejected !== "function") {
6     onRejected = value => value;
7   }
8   if (this.status === HD.PENDING) {
9     this.callbacks.push({
10      onFulfilled: value => {
11        try {
12          onFulfilled(value);
13        } catch (error) {
14          onRejected(error);
15        }
16      },
17      onRejected: value => {
18        try {

```

```

19         onRejected(value);
20     } catch (error) {
21         onRejected(error);
22     }
23 }
24 });
25 }
26 ...
27 }
28

```

### 3. resolve 与 reject 中添加处理 callback 方法的代码

```

1  resolve(value) {
2    if (this.status === HD.PENDING) {
3      this.status = HD.FULFILLED;
4      this.value = value;
5      this.callbacks.map(callback => {
6        callback.onFulfilled(value);
7      });
8    }
9  }
10 reject(value) {
11   if (this.status === HD.PENDING) {
12     this.status = HD.REJECTED;
13     this.value = value;
14     this.callbacks.map(callback => {
15       callback.onRejected(value);
16     });
17   }
18 }
19

```

## #PENDING 异步

执行以下代码发现并不是异步操作，应该先输出 [大叔视频](#) 然后是`后盾人`

```

1  let p = new HD((resolve, reject) => {

```

```

2   setTimeout(() => {
3       resolve("后盾人");
4       console.log("大叔视频");
5   });
6 }).then(
7   value => {
8       console.log(value);
9   },
10  reason => {
11      console.log(reason);
12  }
13 );
14

```

解决以上问题，只需要将 resolve 与 reject 执行通过 setTimeout 定义为异步任务

```

1  resolve(value) {
2      if (this.status == HD.PENDING) {
3          this.status = HD.FULFILLED;
4          this.value = value;
5          setTimeout(() => {
6              this.callbacks.map(callback => {
7                  callback.onFulfilled(value);
8              });
9          });
10     }
11 }
12 reject(value) {
13     if (this.status == HD.PENDING) {
14         this.status = HD.REJECTED;
15         this.value = value;
16         setTimeout(() => {
17             this.callbacks.map(callback => {
18                 callback.onRejected(value);
19             });
20         });
21     }
22 }
23

```

## #链式操作

Promise 中的 then 是链式调用执行的，所以 then 也要返回 Promise 才能实现

1. then 的 onReject 函数是对前面 Promise 的 rejected 的处理
2. 但该 Promise 返回状态要为 fulfilled，所以在调用 onRejected 后改变当前 promise 为 fulfilled 状态

```
1  then(onFulfilled, onRejected) {
2    if (typeof onFulfilled !== "function") {
3      onFulfilled = value => value;
4    }
5    if (typeof onRejected !== "function") {
6      onRejected = value => value;
7    }
8    return new HD((resolve, reject) => {
9      if (this.status === HD.PENDING) {
10         this.callbacks.push({
11           onFulfilled: value => {
12             try {
13               let result = onFulfilled(value);
14               resolve(result);
15             } catch (error) {
16               reject(error);
17             }
18           },
19           onRejected: value => {
20             try {
21               let result = onRejected(value);
22               resolve(result);
23             } catch (error) {
24               reject(error);
25             }
26           }
27         });
28       }
29       if (this.status === HD.FULFILLED) {
30         setTimeout(() => {
31           try {
32             let result = onFulfilled(this.value);
```



```

33         resolve(result);
34     } catch (error) {
35         reject(error);
36     }
37 });
38 }
39 if (this.status === HD.REJECTED) {
40     setTimeout(() => {
41         try {
42             let result = onRejected(this.value);
43             resolve(result);
44         } catch (error) {
45             reject(error);
46         }
47     });
48 }
49 });
50 }
51

```

下面执行测试后，链式操作已经有效了

```

1  let p = new HD((resolve, reject) => {
2      resolve("后盾人");
3      console.log("hdcms.com");
4  })
5  .then(
6      value => {
7          console.log(value);
8          return "大叔视频";
9      },
10     reason => {
11         console.log(reason);
12     }
13 )
14 .then(
15     value => {
16         console.log(value);
17     },

```

```
18   reason => {
19     console.log(reason);
20   }
21 );
22 console.log("houdunren.com");
23
```

## #返回类型

如果 then 返回的是 Promise 呢？所以我们需要判断分别处理返回值为 Promise 与普通值的情况

## #基本实现

下面来实现不同类型不同处理机制

```
1  then(onFulfilled, onRejected) {
2    if (typeof onFulfilled !== "function") {
3      onFulfilled = value => value;
4    }
5    if (typeof onRejected !== "function") {
6      onRejected = value => value;
7    }
8    return new HD((resolve, reject) => {
9      if (this.status === HD.PENDING) {
10        this.callbacks.push({
11          onFulfilled: value => {
12            try {
13              let result = onFulfilled(value);
14              if (result instanceof HD) {
15                result.then(resolve, reject);
16              } else {
17                resolve(result);
18              }
19            } catch (error) {
20              reject(error);
21            }
22          },
23          onRejected: value => {
24            try {
25              let result = onRejected(value);
```

```
26         if (result instanceof HD) {
27             result.then(resolve, reject);
28         } else {
29             resolve(result);
30         }
31     } catch (error) {
32         reject(error);
33     }
34 }
35 });
36 }
37 if (this.status == HD.FULFILLED) {
38     setTimeout(() => {
39         try {
40             let result = onFulfilled(this.value);
41             if (result instanceof HD) {
42                 result.then(resolve, reject);
43             } else {
44                 resolve(result);
45             }
46         } catch (error) {
47             reject(error);
48         }
49     });
50 }
51 if (this.status == HD.REJECTED) {
52     setTimeout(() => {
53         try {
54             let result = onRejected(this.value);
55             if (result instanceof HD) {
56                 result.then(resolve, reject);
57             } else {
58                 resolve(result);
59             }
60         } catch (error) {
61             reject(error);
62         }
63     });
64 }
65 });
```

```
66 }
```

```
67
```

## #代码复用

现在发现 pending、fulfilled、rejected 状态的代码非常相似，所以可以提取出方法 Parse 来复用

```
1  then(onFulfilled, onRejected) {
2    if (typeof onFulfilled !== "function") {
3      onFulfilled = value => value;
4    }
5    if (typeof onRejected !== "function") {
6      onRejected = value => value;
7    }
8    return new HD((resolve, reject) => {
9      if (this.status === HD.PENDING) {
10         this.callbacks.push({
11           onFulfilled: value => {
12             this.parse(onFulfilled(this.value), resolve, reject);
13           },
14           onRejected: value => {
15             this.parse(onRejected(this.value), resolve, reject);
16           }
17         });
18       }
19       if (this.status === HD.FULFILLED) {
20         setTimeout(() => {
21           this.parse(onFulfilled(this.value), resolve, reject);
22         });
23       }
24       if (this.status === HD.REJECTED) {
25         setTimeout(() => {
26           this.parse(onRejected(this.value), resolve, reject);
27         });
28       }
29     });
30   }
31   parse(result, resolve, reject) {
32     try {
```

```

33     if (result instanceof HD) {
34         result.then(resolve, reject);
35     } else {
36         resolve(result);
37     }
38 } catch (error) {
39     reject(error);
40 }
41 }
42

```

## #返回约束

then 的返回的 promise 不能是 then 相同的 Promise，下面是原生 Promise 的示例将产生错误

```

1 let promise = new Promise(resolve => {
2     setTimeout(() => {
3         resolve("后盾人");
4     });
5 });
6 let p = promise.then(value => {
7     return p;
8 });
9

```

解决上面的问题来完善代码，添加当前 promise 做为 parse 的第一个参数与函数结果比对

```

1 then(onFulfilled, onRejected) {
2     if (typeof onFulfilled !== "function") {
3         onFulfilled = value => value;
4     }
5     if (typeof onRejected !== "function") {
6         onRejected = value => value;
7     }
8     let promise = new HD((resolve, reject) => {
9         if (this.status == HD.PENDING) {
10             this.callbacks.push({
11                 onFulfilled: value => {
12                     this.parse(promise, onFulfilled(this.value), resolve, reject);

```

```

13         },
14         onRejected: value => {
15             this.parse(promise, onRejected(this.value), resolve, reject);
16         }
17     });
18 }
19 if (this.status == HD.FULFILLED) {
20     setTimeout(() => {
21         this.parse(promise, onFulfilled(this.value), resolve, reject);
22     });
23 }
24 if (this.status == HD.REJECTED) {
25     setTimeout(() => {
26         this.parse(promise, onRejected(this.value), resolve, reject);
27     });
28 }
29 });
30 return promise;
31 }
32 parse(promise, result, resolve, reject) {
33     if (promise == result) {
34         throw new TypeError("Chaining cycle detected for promise");
35     }
36     try {
37         if (result instanceof HD) {
38             result.then(resolve, reject);
39         } else {
40             resolve(result);
41         }
42     } catch (error) {
43         reject(error);
44     }
45 }
46

```

现在进行测试也可以得到原生一样效果了

```

1 let p = new HD((resolve, reject) => {
2     resolve("后盾人");

```

```
3 });  
4 p = p.then(value => {  
5   return p;  
6 });  
7
```

## #RESOLVE

下面来实现 Promise 的 resolve 方法

```
1 static resolve(value) {  
2   return new HD((resolve, reject) => {  
3     if (value instanceof HD) {  
4       value.then(resolve, reject);  
5     } else {  
6       resolve(value);  
7     }  
8   });  
9 }  
10
```

使用普通值的测试

```
1 HD.resolve("后盾人").then(value => {  
2   console.log(value);  
3 });  
4
```

使用状态为 fulfilled 的 promise 值测试

```
1 HD.resolve(  
2   new HD(resolve => {  
3     resolve("houdunren.com");  
4   })  
5 ).then(value => {  
6   console.log(value);  
7 });  
8
```

使用状态为 rejected 的 Promise 测试

```
1 HD.resolve(  
2   new HD((), reject) => {  
3     reject("reacted");  
4   })  
5 ).then(  
6   value => {  
7     console.log(value);  
8   },  
9   reason => {  
10    console.log(reason);  
11  }  
12 );  
13
```

## #REJECT

下面定义 Promise 的 reject 方法

```
1 static reject(reason) {  
2   return new HD((), reject) => {  
3     reject(reason);  
4   });  
5 }  
6
```

使用测试

```
1 HD.reject("rejected").then(null, reason => {  
2   console.log(reason);  
3 });  
4
```

## #ALL

下面来实现 Promise 的 all 方法



```

1 static all(promises) {
2   let resolves = [];
3   return new HD((resolve, reject) => {
4     promises.forEach((promise, index) => {
5       promise.then(
6         value => {
7           resolves.push(value);
8           if (resolves.length == promises.length) {
9             resolve(resolves);
10          }
11        },
12        reason => {
13          reject(reason);
14        }
15      );
16    });
17  });
18 }
19

```

来对所有 Promise 状态为 fulfilled 的测试

```

1 let p1 = new HD((resolve, reject) => {
2   resolve("后盾人");
3 });
4 let p2 = new HD((resolve, reject) => {
5   reject("后盾人");
6 });
7 let promises = HD.all([p1, p2]).then(
8   promises => {
9     console.log(promises);
10  },
11  reason => {
12    console.log(reason);
13  }
14 );
15

```

使用我们写的 resolve 进行测试

```

1 let p1 = HD.resolve("后盾人");
2 let p2 = HD.resolve("houdunren.com");
3 let promises = HD.all([p1, p2]).then(
4   promises => {
5     console.log(promises);
6   },
7   reason => {
8     console.log(reason);
9   }
10 );
11

```

其中一个 Promise 为 rejected 时的效果

```

1 let p1 = HD.resolve("后盾人");
2 let p2 = HD.reject("rejected");
3 let promises = HD.all([p1, p2]).then(
4   promises => {
5     console.log(promises);
6   },
7   reason => {
8     console.log(reason);
9   }
10 );
11

```

## #RACE

下面实现 Promise 的 race 方法

```

1 static race(promises) {
2   return new HD((resolve, reject) => {
3     promises.map(promise => {
4       promise.then(value => {
5         resolve(value);
6       });
7     });
8   });

```

```
9  }  
10
```

我们来进行测试

```
1  let p1 = HD.resolve("后盾人");  
2  let p2 = HD.resolve("houdunren.com");  
3  let promises = HD.race([p1, p2]).then(  
4    promises => {  
5      console.log(promises);  
6    },  
7    reason => {  
8      console.log(reason);  
9    }  
10 );  
11
```

使用延迟 Promise 后的效果

```
1  let p1 = new HD(resolve => {  
2    setInterval(() => {  
3      resolve("后盾人");  
4    }, 2000);  
5  });  
6  let p2 = new HD(resolve => {  
7    setInterval(() => {  
8      resolve("houdunren.com");  
9    }, 1000);  
10 });  
11 let promises = HD.race([p1, p2]).then(  
12   promises => {  
13     console.log(promises);  
14   },  
15   reason => {  
16     console.log(reason);  
17   }  
18 );
```

