

# Promise

JavaScript 中存在很多异步操作,Promise 将异步操作队列化,按照期望的顺序执行,返回符合预期的结果。可以通过链式调用多个 Promise 达到我们的目的。

Promise 在各种开源库中已经实现,现在标准化后被浏览器默认支持。

promise 是一个拥有 then 方法的对象或函数

## #问题探讨

下面通过多个示例来感受一下不使用 promise 时,处理相应问题的不易,及生成了不便阅读的代码。

## #定时嵌套

下面是一个定时器执行结束后,执行另一个定时器,这种嵌套造成代码不易阅读

```
1  <style>
2    div {
3      width: 100px;
4      height: 100px;
5      background: yellowgreen;
6      position: absolute;
7    }
8  </style>
9
10 <body>
11   <div></div>
12 </body>
13
14 <script>
15   function interval(callback, delay = 100) {
16     let id = setInterval(() => callback(id), delay);
17   }
18
19   const div = document.querySelector("div");
20   interval(timeId => {
21     const left = parseInt(window.getComputedStyle(div).left);
22     div.style.left = left + 10 + "px";
23     if (left > 200) {
```

```

24     clearInterval(timeId);
25     interval(timeId => {
26         const width = parseInt(window.getComputedStyle(div).width);
27         div.style.width = width - 1 + "px";
28         if (width <= 0) clearInterval(timeId);
29     }, 10);
30 }
31 }, 100);
32 </script>
33

```

## #图片加载

下面是图片后设置图片边框，也需要使用回调函数处理，代码嵌套较复杂

```

1  function loadImage(file, resolve, reject) {
2      const image = new Image();
3      image.src = file;
4      image.onload = () => {
5          resolve(image);
6      };
7      image.onerror = () => {
8          reject(new Error("load fail"));
9      };
10     document.body.appendChild(image);
11 }
12
13 loadImage(
14     "images/houdunren.png",
15     image => {
16         image.style.border = "solid 5px red";
17     },
18     error => {
19         console.log(error);
20     }
21 );
22

```

## #加载文件

下面是异步加载外部JS文件，需要使用回调函数执行，并设置的错误处理的回调函数

```
1 function load(file, resolve, reject) {
2   const script = document.createElement("script");
3   script.src = file;
4   script.onload = resolve;
5   script.onerror = reject;
6   document.body.appendChild(script);
7 }
8 load(
9   "js/hd.js",
10  script => {
11    console.log(`${script.path[0].src} 加载成功`);
12    hd();
13  },
14  error => {
15    console.log(`${error.srcElement.src} 加载失败`);
16  }
17 );
18
```

实例中用到的 `hd.js` 与 `houdunren.js` 内容如下

```
1 # hd.js
2 function hd() {
3   console.log("hd function run");
4 }
5
6 # houdunren.js
7 function houdunren() {
8   console.log("houdunren function run");
9   hd();
10 }
11
```

如果要加载多个脚本时需要嵌套使用，下面`houdunren.js` 依赖 `hd.js`，需要先加载`hd.js` 后加载`houdunren.js`

不断的回调函数操作将产生回调地狱，使代码很难维护

```

1 load(
2   "js/hd.js",
3   script => {
4     load(
5       "js/houdunren.js",
6       script => {
7         houdunren();
8       },
9       error => {
10        console.log(`${error.srcElement.src} 加载失败`);
11      }
12    );
13  },
14  error => {
15    console.log(`${error.srcElement.src} 加载失败`);
16  }
17 );
18

```

## #异步请求

使用传统的异步请求也会产生回调嵌套的问题，下在是获取向军的成绩，需要经过以下两步

### 1. 根据用户名取得

向军 的编号

### 2. 根据编号获取成绩

示例中用到的 php 文件请在 [版本库 \(opens new window\)](#)中查看  
 启动 PHP 服务器命令 `php -S localhost:8080`

```

1 function ajax(url, resolve, reject) {
2   let xhr = new XMLHttpRequest();
3   xhr.open("GET", url);
4   xhr.send();
5   xhr.onload = function() {
6     if (this.status == 200) {
7       resolve(JSON.parse(this.response));
8     } else {
9       reject(this);

```

```

10     }
11 };
12 }
13 ajax("
  http://localhost:8888/php/user.php?name=
  ajax("
14     ajax(
15         、
  http://localhost:8888/php/houdunren.php?id=
  、
16     response => {
17         console.log(response[0]);
18     }
19 );
20 });
21

```

## #肯德基

下面是模拟肯德基吃饭的事情，使用 `promise` 操作异步的方式每个阶段会很清楚

```

1 let kfc = new Promise((resolve, reject) => {
2     console.log("肯德基厨房开始做饭");
3     resolve("我是肯德基，你的餐已经做好了");
4 });
5 let dad = kfc.then(msg => {
6     console.log(`收到肯德基消息: ${msg}`);
7     return {
8         then(resolve) {
9             setTimeout(() => {
10                 resolve("孩子，我吃了两秒了，不辣，你可以吃了");
11             }, 2000);
12         }
13     };
14 });
15 let son = dad.then(msg => {
16     return new Promise((resolve, reject) => {
17         console.log(`收到爸爸消息: ${msg}`);
18         setTimeout(() => {
19             resolve("妈妈，我和向军爸爸吃完饭了");

```

```

20     }, 2000);
21 });
22 });
23 let ma = son.then(msg => {
24     console.log(`收到孩子消息: ${msg},事情结束`);
25 });
26

```

而使用以往的回调方式，就会让人苦不堪言

```

1  function notice(msg, then) {
2      then(msg);
3  }
4  function meal() {
5      notice("肯德基厨房开始做饭", msg => {
6          console.log(msg);
7          notice("我是肯德基，你的餐已经做好", msg => {
8              console.log(`收到肯德基消息: ${msg}`);
9              setTimeout(() => {
10                 notice("孩子，我吃了两秒了，不辣，你可以吃了", msg => {
11                     console.log(`收到爸爸消息: ${msg}`);
12                     setTimeout(() => {
13                         notice("妈妈，我和向军爸爸吃完饭了", msg => {
14                             console.log(`收到孩子消息: ${msg},事情结束`);
15                         });
16                     }, 2000);
17                 });
18             }, 2000);
19         });
20     });
21 }
22 meal();
23

```

## #异步状态

Promise 可以理解为承诺，就像我们去 KFC 点餐服务员给我们一引取餐票，这就是承诺。如果餐做好了叫我们这就是成功，如果没有办法给我们做出食物这就是拒绝。

- 一个

promise 必须有一个  
then 方法用于处理状态改变

## #状态说明

Promise 包含 `pending`、`fulfilled`、`rejected` 三种状态

- `pending` 指初始等待状态，初始化

promise 时的状态

- `resolve` 指已经解决，将

promise 状态设置为

`fulfilled`

- `reject` 指拒绝处理，将

promise 状态设置为

`rejected`

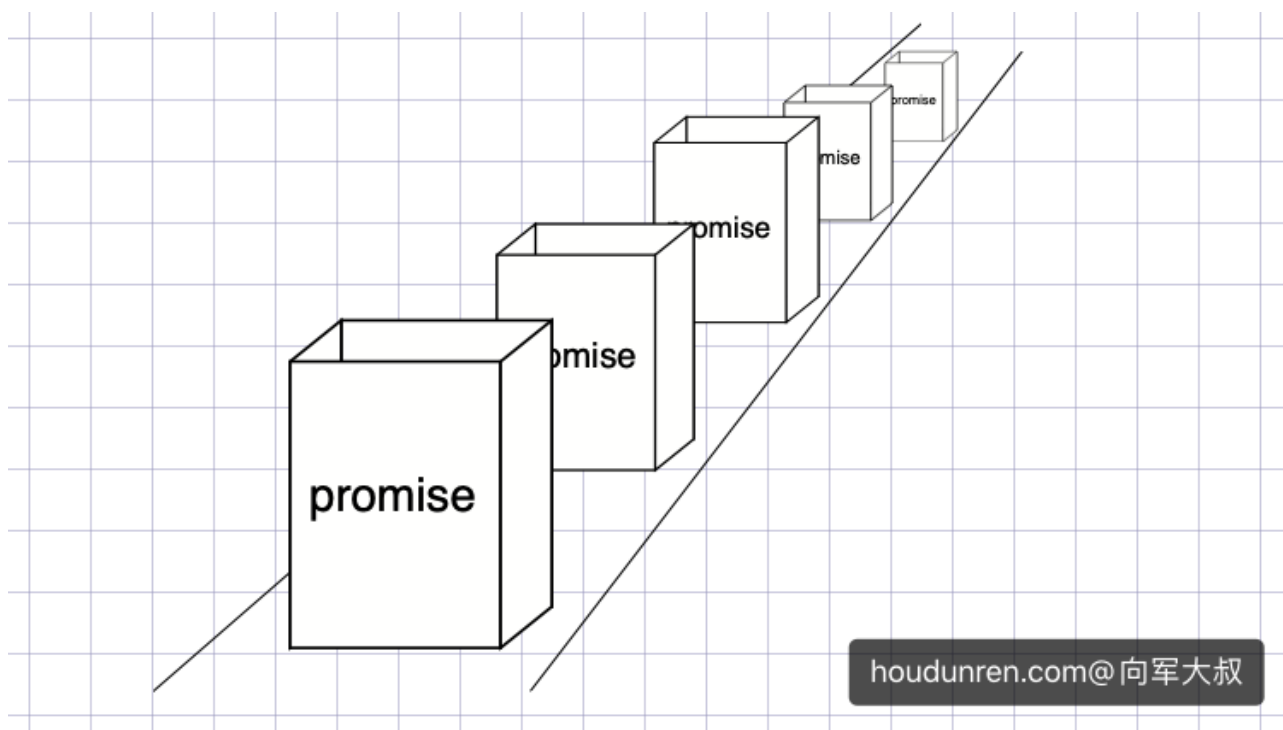
- promise 是生产者，通过

`resolve` 与

`reject` 函数告之结果

- promise 非常适合需要一定执行时间的异步任务
- 状态一旦改变将不可更改

promise 是队列状态，就像体育中的接力赛，或多米诺骨牌游戏，状态一直向后传递，当然其中的任何一个 promise 也可以改变状态。



promise 没有使用 `resolve` 或 `reject` 更改状态时，状态为 `pending`

```
1 console.log(  
2   new Promise((resolve, reject) => {  
3     });  
4   ); //Promise {<pending>}  
5
```

当更改状态后

```
1 console.log(  
2   new Promise((resolve, reject) => {  
3     resolve("fulfilled");  
4   })  
5 ); //Promise {<resolved>: "fulfilled"}  
6  
7 console.log(  
8   new Promise((resolve, reject) => {  
9     reject("rejected");  
10  })  
11 ); //Promise {<rejected>: "rejected"}  
12
```

`promise` 创建时即立即执行即同步任务，`then` 会放在异步微任务中执行，需要等同步任务执行后才执行。

```
1 let promise = new Promise((resolve, reject) => {  
2   resolve("fulfilled");  
3   console.log("后盾人");  
4 });  
5 promise.then(msg => {  
6   console.log(msg);  
7 });  
8 console.log("houdunren.com");  
9
```

`promise` 操作都是在其他代码后执行，下面会先输出 `houdunren.com` 再弹出 `success`

- `promise` 的 `then`、`catch`、`finally` 的方法都是异步任务
- 程序需要将主任务执行完成才会执行异步队列任务



```
1 const promise = new Promise(resolve => resolve("success"));
2 promise.then(alert);
3 alert("houdunren.com");
4 promise.then(() => {
5     alert("后盾人");
6 });
7
```

下例在三秒后将 `Promise` 状态设置为 `fulfilled`，然后执行 `then` 方法

```
1 new Promise((resolve, reject) => {
2     setTimeout(() => {
3         resolve("fulfilled");
4     }, 3000);
5 }).then(
6     msg => {
7         console.log(msg);
8     },
9     error => {
10        console.log(error);
11    }
12 );
13
```

状态被改变后就不能再修改了，下面先通过 `resolve` 改变为成功状态，表示 `promise` 状态已经完成，就不能使用 `reject` 更改状态了

```
1 new Promise((resolve, reject) => {
2     resolve("操作成功");
3     reject(new Error("请求失败"));
4 }).then(
5     msg => {
6         console.log(msg);
7     },
8     error => {
9         console.log(error);
10    }
11 );
```

```
11 );  
12
```

## #动态改变

下例中 p2 返回了 p1 所以此时 p2 的状态已经无意义了，后面的 then 是对 p1 状态的处理。

```
1  const p1 = new Promise((resolve, reject) => {  
2    // resolve("fulfilled");  
3    reject("rejected");  
4  });  
5  const p2 = new Promise(resolve => {  
6    resolve(p1);  
7  }).then(  
8    value => {  
9      console.log(value);  
10   },  
11   reason => {  
12     console.log(reason);  
13   }  
14 );  
15
```

如果 `resolve` 参数是一个 `promise`，将会改变 `promise` 状态。

下例中 `p1` 的状态将被改变为 `p2` 的状态

```
1  const p1 = new Promise((resolve, reject) => {  
2    resolve(  
3      //p2  
4      new Promise((s, e) => {  
5        s("成功");  
6      })  
7    });  
8  }).then(msg => {  
9    console.log(msg);  
10  });  
11
```

当 promise 做为参数传递时，需要等待 promise 执行完才可以继承，下面的 p2 需要等待 p1 执行完成。

- 因为

p2 的

resolve 返回了

p1 的 promise，所以此时

p2 的

then 方法已经是

p1 的了

- 正因为以上原因

then 的第一个函数输出了

p1 的

resolve 的参数

```
1  const p1 = new Promise((resolve, reject) => {
2      setTimeout(() => {
3          resolve("操作成功");
4      }, 2000);
5  });
6  const p2 = new Promise((resolve, reject) => {
7      resolve(p1);
8  }).then(
9      msg => {
10         console.log(msg);
11     },
12     error => {
13         console.log(error);
14     }
15 );
16
```

## #then

一个 promise 需要提供一个 then 方法访问 promise 结果，then 用于定义当 promise 状态发生改变时的处理，即 promise 处理异步操作，then 用于结果。

promise 就像 kfc 中的厨房，then 就是我们用户，如果餐做好了即 fulfilled，做不了拒绝即 rejected 状态。那么 then 就要对不同状态处理。

- then 方法必须返回 promise，用户返回或系统自动返回

- 第一个函数在

resolved 状态时执行，即执行  
resolve时执行

then第一个函数处理成功状态

- 第二个函数在

rejected状态时执行，即执行

reject 时执行第二个函数处理失败状态，该函数是可选的

- 两个函数都接收

promise 传出的值做为参数

- 也可以使用

catch 来处理失败的状态

- 如果

then 返回

promise，下一个

then 会在当前

promise 状态改变后执行

## #语法说明

then 的语法如下，onFulfilled 函数处理 fulfilled 状态， onRejected 函数处理 rejected 状态

- onFulfilled 或 onRejected 不是函数将被忽略
- 两个函数只会被调用一次
- onFulfilled 在 promise 执行成功时调用
- onRejected 在 promise 执行拒绝时调用

```
1 promise.then(onFulfilled, onRejected)
2
```

## #基础知识

then 会在 promise 执行完成后执行，then 第一个函数在 resolve成功状态执行

```
1 const promise = new Promise((resolve, reject) => {
2   resolve("success");
3 }).then(
4   value => {
5     console.log(`解决: ${value}`);
6   },
```

```
7   reason => {
8     console.log(`拒绝:${reason}`);
9   }
10 );
11
```

`then` 中第二个参数在失败状态执行

```
1  const promise = new Promise((resolve, reject) => {
2    reject("is error");
3  });
4  promise.then(
5    msg => {
6      console.log(`成功: ${msg}`);
7    },
8    error => {
9      console.log(`失败:${error}`);
10   }
11 );
12
```

如果只关心成功则不需要传递 `then` 的第二个参数

```
1  const promise = new Promise((resolve, reject) => {
2    resolve("success");
3  });
4  promise.then(msg => {
5    console.log(`成功: ${msg}`);
6  });
7
```

如果只关心失败时状态, `then` 的第一个参数传递 `null`

```
1  const promise = new Promise((resolve, reject) => {
2    reject("is error");
3  });
4  promise.then(null, error => {
5    console.log(`失败:${error}`);
6  });
```

promise 传向 then 的传递值，如果 then 没有可处理函数，会一直向后传递

```

1 let p1 = new Promise((resolve, reject) => {
2   reject("rejected");
3 })
4 .then()
5 .then(
6   null,
7   f => console.log(f)
8 );
9

```

如果 onFulfilled 不是函数且 promise 执行成功, p2 执行成功并返回相同值

```

1 let promise = new Promise((resolve, reject) => {
2   resolve("resolve");
3 });
4 let p2 = promise.then();
5 p2.then().then(resolve => {
6   console.log(resolve);
7 });
8

```

如果 onRejected 不是函数且 promise 拒绝执行, p2 拒绝执行并返回相同值

```

1 let promise = new Promise((resolve, reject) => {
2   reject("reject");
3 });
4 let p2 = promise.then(() => {});
5 p2.then(null, null).then(null, reject => {
6   console.log(reject);
7 });
8

```

## #链式调用

每次的 `then` 都是一个全新的 `promise`，默认 `then` 返回的 `promise` 状态是 `fulfilled`

```

1 let promise = new Promise((resolve, reject) => {
2   resolve("fulfilled");
3 }).then(resolve => {
4   console.log(resolve);
5 })
6 .then(resolve => {
7   console.log(resolve);
8 });
9

```

每次的 `then` 都是一个全新的 `promise`，不要认为上一个 `promise` 状态会影响以后 `then` 返回的状态

```

1 let p1 = new Promise(resolve => {
2   resolve();
3 });
4 let p2 = p1.then(() => {
5   console.log("后盾人");
6 });
7 p2.then(() => {
8   console.log("houdunren.com");
9 });
10 console.log(p1); // Promise {<resolved>}
11 console.log(p2); // Promise {<pending>}
12
13 # 再试试把上面两行放在 setTimeout里
14 setTimeout(() => {
15   console.log(p1); // Promise {<resolved>}
16   console.log(p2); // Promise {<resolved>}
17 });
18

```

`then` 是对上个 `promise` 的 `rejected` 的处理，每个 `then` 会是一个新的 `promise`，默认传递 `fulfilled` 状态

```

1 new Promise((resolve, reject) => {
2   reject();
3 })
4 .then(

```

```

5   resolve => console.log("fulfilled"),
6   reject => console.log("rejected")
7 )
8 .then(
9   resolve => console.log("fulfilled"),
10  reject => console.log("rejected")
11 )
12 .then(
13   resolve => console.log("fulfilled"),
14   reject => console.log("rejected")
15 );
16
17 # 执行结果如下
18   ejected
19   fulfilled
20   fulfilled
21

```

如果内部返回 `promise` 时将使用该 `promise`

```

1  let p1 = new Promise(resolve => {
2    resolve();
3  });
4  let p2 = p1.then(() => {
5    return new Promise(r => {
6      r("houdunren.com");
7    });
8  });
9  p2.then(v => {
10   console.log(v); //houdunren.com
11 });
12

```

如果 `then` 返回 `promise` 时，后面的 `then` 就是对返回的 `promise` 的处理，需要等待该 `promise` 变更状态后执行。

```

1  let promise = new Promise(resolve => resolve());
2  let p1 = promise.then(() => {
3    return new Promise(resolve => {

```



```

4     setTimeout(() => {
5         console.log(`p1`);
6         resolve();
7     }, 2000);
8 });
9 }).then(() => {
10    return new Promise((a, b) => {
11        console.log(`p2`);
12    });
13 });
14

```

如果 `then` 返回 `promise` 时，返回的 `promise` 后面的 `then` 就是处理这个 `promise` 的

如果不 `return` 情况就不是这样了，即外层的 `then` 的 `promise` 和内部的 `promise` 是独立的两个 `promise`

```

1  new Promise((resolve, reject) => {
2      resolve();
3  })
4  .then(v => {
5      return new Promise((resolve, reject) => {
6          resolve("第二个promise");
7      }).then(value => {
8          console.log(value);
9          return value;
10     });
11 })
12 .then(value => {
13     console.log(value);
14 });
15

```

这是对上面代码的优化，把内部的 `then` 提取出来

```

1  new Promise((resolve, reject) => {
2      resolve();
3  })
4  .then(v => {
5      return new Promise((resolve, reject) => {

```

```
6     resolve("第二个promise");
7   });
8 })
9 .then(value => {
10   console.log(value);
11   return value;
12 })
13 .then(value => {
14   console.log(value);
15 });
16
```

## #其它类型

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`。

## #循环调用

如果 `then` 返回与 `promise` 相同将禁止执行

```
1 let promise = new Promise(resolve => {
2   resolve();
3 });
4 let p2 = promise.then(() => {
5   return p2;
6 }); // TypeError: Chaining cycle detected for promise
7
```

## #promise

如果返回值是 `promise` 对象，则需要更新状态后，才可以继承执行后面的 `promise`

```
1 new Promise((resolve, reject) => {
2   resolve(
3     new Promise((resolve, reject) => {
4       setTimeout(() => {
5         resolve("解决状态");
6       });
7     })
8   );
9 })
```

```

6      }, 2000);
7    })
8  );
9 })
10 .then(
11   v => {
12     console.log(`fulfilled: ${v}`);
13     return new Promise((resolve, reject) => {
14       setTimeout(() => {
15         reject("失败状态");
16       }, 2000);
17     });
18   },
19   v => {
20     console.log(`rejected: ${v}`);
21   }
22 )
23 .catch(error => console.log(`rejected: ${error}`));
24

```

## #Thenables

包含 `then` 方法的对象就是一个 `promise`，系统将传递 `resolvePromise` 与 `rejectPromise` 做为函数参数

下例中使用 `resolve` 或在 `then` 方法中返回了具有 `then` 方法的对象

- 该对象即为

`promise` 要先执行，并在方法内部更改状态

- 如果不更改状态，后面的

`then promise` 都为等待状态

```

1  new Promise((resolve, reject) => {
2    resolve({
3      then(resolve, reject) {
4        resolve("解决状态");
5      }
6    });
7  })
8  .then(v => {
9    console.log(`fulfilled: ${v}`);

```

```

10  return {
11    then(resolve, reject) {
12      setTimeout(() => {
13        reject("失败状态");
14      }, 2000);
15    }
16  };
17 })
18 .then(null, error => {
19   console.log(`rejected: ${error}`);
20 });
21

```

包含 `then` 方法的对象可以当作 promise 来使用

```

1  class User {
2    (id) {
3      this.id = id;
4    }
5    then(resolve, reject) {
6      resolve ajax(`
7        http://localhost:8888/php/houdunren.php?id=
8        resolve ajax(`
9    }
10   }
11   new Promise((resolve, reject) => {
12     resolve ajax(`
13       http://localhost:8888/php/user.php?name=
14       resolve ajax(`
15   })
16   .then(user => {
17     return new User(user.id);
18   })
19   .then(lessons => {
20     console.log(lessons);
21   });
22 }
23

```

当然也可以是类

```

1 new Promise((resolve, reject) => {
2   resolve(
3     class {
4       static then(resolve, reject) {
5         setTimeout(() => {
6           resolve("解决状态");
7         }, 2000);
8       }
9     }
10  );
11 }).then(
12   v => {
13     console.log(`fulfilled: ${v}`);
14   },
15   v => {
16     console.log(`rejected: ${v}`);
17   }
18 );
19

```

如果对象中的 then 不是函数，则将对象做为值传递

```

1 new Promise((resolve, reject) => {
2   resolve();
3 })
4 .then(() => {
5   return {
6     then: "后盾人"
7   };
8 })
9 .then(v => {
10  console.log(v); //{then: "后盾人"}
11 });
12

```

## #catch

下面使用未定义的变量同样会触发失败状态

```

1 let promise = new Promise((resolve, reject) => {
2   hd;
3 }).then(
4   value => console.log(value),
5   reason => console.log(reason)
6 );
7

```

如果 onFulfilled 或 onRejected 抛出异常，则 p2 拒绝执行并返回拒因

```

1 let promise = new Promise((resolve, reject) => {
2   throw new Error("fail");
3 });
4 let p2 = promise.then();
5 p2.then().then(null, resolve => {
6   console.log(resolve + ",后盾人");
7 });
8

```

catch 用于失败状态的处理函数，等同于 `then(null, reject){}`

- 建议使用

catch 处理错误

- 将

catch 放在最后面用于统一处理前面发生的错误

```

1 const promise = new Promise((resolve, reject) => {
2   reject(new Error("Notice: Promise Exception"));
3 }).catch(msg => {
4   console.error(msg);
5 });
6

```

catch 可以捕获之前所有 promise 的错误，所以建议将 catch 放在最后。下例中 catch 也可以捕获到了第一个 then 返回的 promise 的错误。

```

1 new Promise((resolve, reject) => {
2   resolve();

```

```

3  })
4  .then(() => {
5      return new Promise((resolve, reject) => {
6          reject(".then ");
7      });
8  })
9  .then(() => {})
10 .catch(msg => {
11     console.log(msg);
12 });
13

```

错误是冒泡的操作的，下面没有任何一个`then` 定义第二个函数，将一直冒泡到 `catch` 处理错误

```

1  new Promise((resolve, reject) => {
2      reject(new Error("请求失败"));
3  })
4  .then(msg => {})
5  .then(msg => {})
6  .catch(error => {
7      console.log(error);
8  });
9

```

`catch` 也可以捕获对 `then` 抛出的错误处理

```

1  new Promise((resolve, reject) => {
2      resolve();
3  })
4  .then(msg => {
5      throw new Error("这是then 抛出的错误");
6  })
7  .catch(() => {
8      console.log("33");
9  });
10

```

`catch` 也可以捕获其他错误，下面在 `then` 中使用了未定义的变量，将会把错误抛出到 `catch`

```
1 new Promise((resolve, reject) => {
2   resolve("success");
3 })
4 .then(msg => {
5   console.log(a);
6 })
7 .catch(reason => {
8   console.log(reason);
9 });
10
```

## #使用建议

建议将错误要交给`catch`处理而不是在`then`中完成，不建议使用下面的方式管理错误

```
1 new Promise((resolve, reject) => {
2   reject(new Error("请求失败"));
3 }).then(
4   msg => {
5     console.log(msg);
6   },
7   error => {
8     console.log(error);
9   }
10 );
11
```

## #处理机制

在 `promise` 中抛出的错误也会被`catch` 捕获

```
1 const promise = new Promise((resolve, reject) => {
2   throw new Error("fail");
3 }).catch(msg => {
4   console.log(msg.()+"后盾人");
5 });
6
```



可以将上面的理解为如下代码，可以理解为内部自动执行 `try...catch`

```
1 const promise = new Promise((resolve, reject) => {
2   try {
3     throw new Error("fail");
4   } catch (error) {
5     reject(error);
6   }
7 }).catch(msg => {
8   console.log(msg.());
9 });
10
```

但像下面的在异步中 `throw` 将不会触发 `catch`，而使用系统错误处理

```
1 const promise = new Promise((resolve, reject) => {
2   setTimeout(() => {
3     throw new Error("fail");
4   }, 2000);
5 }).catch(msg => {
6   console.log(msg + "后盾人");
7 });
8
```

下面在 `then` 方法中使用了没有定义的 `hd` 函数，也会抛除到 `catch` 执行，可以理解为内部自动执行 `try...catch`

```
1 const promise = new Promise((resolve, reject) => {
2   resolve();
3 })
4 .then(() => {
5   hd();
6 })
7 .catch(msg => {
8   console.log(msg.());
9 });
10
```

在 `catch` 中发生的错误也会抛给最近的错误处理

```
1  const promise = new Promise((resolve, reject) => {
2    reject();
3  })
4  .catch(msg => {
5    hd();
6  })
7  .then(null, error => {
8    console.log(error);
9  });
10
```

## #定制错误

可以根据不同的错误类型进行定制操作，下面将参数错误与 404 错误分别进行了处理

```
1  class ParamError extends Error {
2    constructor(msg) {
3      super(msg);
4      this.name = "ParamError";
5    }
6  }
7  class HttpError extends Error {
8    constructor(msg) {
9      super(msg);
10     this.name = "HttpError";
11   }
12 }
13 function ajax(url) {
14   return new Promise((resolve, reject) => {
15     if (!/^http/.test(url)) {
16       throw new ParamError("请求地址格式错误");
17     }
18     let xhr = new XMLHttpRequest();
19     xhr.open("GET", url);
20     xhr.send();
21     xhr.onload = function() {
22       if (this.status == 200) {
```

```

23     resolve(JSON.parse(this.response));
24   } else if (this.status == 404) {
25     // throw new HttpError("用户不存在");
26     reject(new HttpError("用户不存在"));
27   } else {
28     reject("加载失败");
29   }
30 };
31 xhr.onerror = function() {
32   reject(this);
33 };
34 });
35 }
36
37 ajax(`
  http://localhost:8888/php/user.php?name=
  ajax(`
38 .then(value => {
39   console.log(value);
40 })
41 .catch(error => {
42   if (error instanceof ParamError) {
43     console.log(error.message);
44   }
45   if (error instanceof HttpError) {
46     alert(error.message);
47   }
48   console.log(error);
49 });
50

```

## #事件处理

**unhandledrejection**事件用于捕获到未处理的 Promise 错误，下面的 then 产生了错误，但没有 `catch` 处理，这时就会触发事件。该事件有可能在以后被废除，处理方式是对没有处理的错误直接终止。

```

1 window.addEventListener("unhandledrejection", function(event) {
2   console.log(event.promise); // 产生错误的promise对象
3   console.log(event.reason); // Promise的reason

```

```
4 });  
5  
6 new Promise((resolve, reject) => {  
7   resolve("success");  
8 }).then(msg => {  
9   throw new Error("fail");  
10 });  
11
```

## #finally

无论状态是 `resolve` 或 `reject` 都会执行此动作，`finally` 与状态无关。

```
1 const promise = new Promise((resolve, reject) => {  
2   reject("hdcms");  
3 })  
4 .then(msg => {  
5   console.log("resolve");  
6 })  
7 .catch(msg => {  
8   console.log("reject");  
9 })  
10 .finally(() => {  
11   console.log("resolve/reject状态都会执行");  
12 });  
13
```

下面使用 `finally` 处理加载状态，当请求完成时移除加载图标。请在后台 php 文件中添加 `sleep(2);` 设置延迟响应

```
1 <body>  
2   <style>  
3     div {  
4       width: 100px;  
5       height: 100px;  
6       background: red;  
7       color: white;  
8       display: none;  
9     }
```

```
10 </style>
11 <div>loading...</div>
12 </body>
13 <script>
14 function ajax(url) {
15     return new Promise((resolve, reject) => {
16         document.querySelector("div").style.display = "block";
17         let xhr = new XMLHttpRequest();
18         xhr.open("GET", url);
19         xhr.send();
20         xhr.onload = function() {
21             if (this.status == 200) {
22                 resolve(JSON.parse(this.response));
23             } else {
24                 reject(this);
25             }
26         };
27     });
28 }
29
30 ajax("
http://localhost:8888/php/user.php?name=
ajax\("
31     .then\(user => {
32         console.log\(user\);
33     }\)
34     .catch\(error => {
35         console.log\(error\);
36     }\)
37     .finally\(\(\) => {
38         document.querySelector\("div"\).style.display = "none";
39     }\)
40 </script>
41
```

## #实例操作

## #异步请求

下面是将 `ajax` 修改为 `promise` 后，代码结构清晰了很多

```

1 function ajax(url) {
2   return new Promise((resolve, reject) => {
3     let xhr = new XMLHttpRequest();
4     xhr.open("GET", url);
5     xhr.send();
6     xhr.onload = function() {
7       if (this.status == 200) {
8         resolve(JSON.parse(this.response));
9       } else {
10        reject(this);
11      }
12    };
13  });
14 }
15
16 ajax("
17 http://localhost:8888/php/user.php?name=
18 ajax("
19 .then(user =>ajax(`
20 http://localhost:8888/php/houdunren.php?id=
21 .then(user =>ajax(`
22 .then(lesson => {
23   console.log(lesson);
24 });
25

```

## #图片加载

下面是异步加载图片示例

```

1 function loadImage(file) {
2   return new Promise((resolve, reject) => {
3     const image = new Image();
4     image.src = file;
5     image.onload = () => {
6       resolve(image);
7     };
8     image.onerror = reject;
9     document.body.appendChild(image);

```

```

10   });
11 }
12
13 loadImage("images/houdunren.png").then(image => {
14   image.style.border = "solid 20px black";
15   console.log("宽度:" + window.getComputedStyle(image).width);
16 });
17

```

## #定时器

下面是封装的 `timeout` 函数，使用定时器操作更加方便

```

1  function timeout(times) {
2    return new Promise(resolve => {
3      setTimeout(resolve, times);
4    });
5  }
6
7  timeout(3000)
8    .then(() => {
9      console.log("3秒后执行");
10     return timeout(1000);
11   })
12   .then(() => {
13     console.log("执行上一步的promise后1秒执行");
14   });
15

```

封闭 `setInterval` 定时器并实现动画效果

```

1  <body>
2    <style>
3      div {
4        width: 100px;
5        height: 100px;
6        background: yellowgreen;
7        position: absolute;
8      }

```

```
9   </style>
10  <div></div>
11 </body>
12 <script>
13   function interval(delay = 1000, callback) {
14     return new Promise(resolve => {
15       let id = setInterval(() => {
16         callback(id, resolve);
17       }, delay);
18     });
19   }
20   interval(100, (id, resolve) => {
21     const div = document.querySelector("div");
22     let left = parseInt(window.getComputedStyle(div).left);
23     div.style.left = left + 10 + "px";
24     if (left >= 200) {
25       clearInterval(id);
26       resolve(div);
27     }
28   }).then(div => {
29     interval(50, (id, resolve) => {
30       let width = parseInt(window.getComputedStyle(div).width);
31       div.style.width = width - 10 + "px";
32       if (width <= 20) {
33         clearInterval(id);
34       }
35     });
36   });
37 </script>
38
```

## #链式操作

- 第个

then 都是一个 promise

- 如果

then 返回 promise, 只当

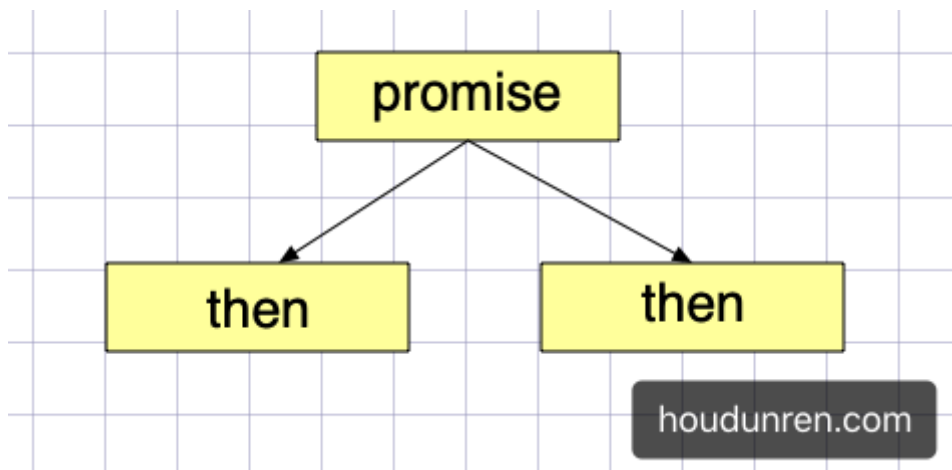
promise 结束后, 才会继承执行下一个

then



## #语法介绍

下面是对同一个 `promise` 的多个 `then`，每个 `then` 都得到了同一个 `promise` 结果，这不是链式操作，实际使用意义不大。



```
1  const promise = new Promise((resolve, reject) => {
2    resolve("后盾人");
3  });
4  promise.then(hd => {
5    hd += "-hdcms";
6    console.log(hd); //后盾人-hdcms
7  });
8  promise.then(hd => {
9    hd += "-houdunren";
10   console.log(hd); //后盾人-houdunren
11 });
12
```

第一个 `then` 也是一个 `promise`，当没接受到结果是状态为 `pending`

```
1  const promise = new Promise((resolve, reject) => {
2    resolve("后盾人");
3  });
4
5  console.log(
6    promise.then(hd => {
7      hd += "-hdcms";
8      console.log(hd);
```

```

9    })
10   ); //Promise {<pending>}
11

```

`promise` 中的 `then` 方法可以链接执行，`then` 方法的返回值会传递到下一个 `then` 方法。

- `then` 会返回一个

`promise`，所以如果有多个

`then` 时会连续执行

- `then` 返回的值会做为当前

`promise` 的结果

下面是链式操作的 `then`，即始没有 `return` 也是会执行，因为每个 `then` 会返回 `promise`

```

1  new Promise((resolve, reject) => {
2    resolve("后盾人");
3  })
4  .then(hd => {
5    hd += "-hdcms";
6    console.log(hd); //后盾人-hdcms
7    return hd;
8  })
9  .then(hd => {
10   hd += "-houdunren";
11   console.log(hd); //后盾人-hdcms-houdunren
12 });
13

```

`then` 方法可以返回一个 `promise` 对象，等 `promise` 执行结束后，才会继承执行后面的 `then`。后面的 `then` 方法就是对新返回的 `promise` 状态的处理

```

1  new Promise((resolve, reject) => {
2    resolve("第一个promise");
3  })
4  .then(msg => {
5    console.log(msg);
6    return new Promise((resolve, reject) => {
7      setTimeout(() => {
8        resolve("第二个promise");
9      }, 3000);
10   });

```

```
11 })
12 .then(msg => {
13   console.log(msg);
14 });
15
```

## #链式加载

使用 `promise` 链式操作重构前面章节中的文件加载，使用代码会变得更清晰

```
1 function load(file) {
2   return new Promise((resolve, reject) => {
3     const script = document.createElement("script");
4     script.src = file;
5     script.onload = () => resolve(script);
6     script.onerror = () => reject();
7     document.body.appendChild(script);
8   });
9 }
10
11 load("js/hd.js")
12 .then(() => load("js/houdunren.js"))
13 .then(() => houdunren());
14
```

## #操作元素

下面使用 `promise` 对元素事件进行处理

```
1 <body>
2   <div>
3     <h2>第九章 闭包与作用域</h2>
4     <button>收藏课程</button>
5   </div>
6 </body>
7
8 <script>
9   new Promise(resolve => {
10     document.querySelector("button").addEventListener("click", e => {
```

```

11     resolve();
12 });
13 })
14 .then(() => {
15     return new Promise(resolve => {
16         setTimeout(() => {
17             console.log("执行收藏任务");
18             resolve();
19         }, 2000);
20     });
21 })
22 .then(() => {
23     return new Promise(resolve => {
24         setTimeout(() => {
25             console.log("更新积分");
26             resolve();
27         }, 2000);
28     });
29 })
30 .then(() => {
31     console.log("收藏成功! 奖励10积分");
32 })
33 .catch(error => console.log(errro));
34

```

## #异步请求

下面是使用链式操作获取学生成绩

```

1  function ajax(url) {
2      return new Promise((resolve, reject) => {
3          let xhr = new XMLHttpRequest();
4          xhr.open("GET", url);
5          xhr.send();
6          xhr.onload = function() {
7              if (this.status == 200) {
8                  resolve(JSON.parse(this.response));
9              } else {
10                 reject(this);

```

```

11     }
12   };
13 });
14 }
15 ajax("
  http://localhost:8888/php/user.php?name=
  ajax("
16 .then(user => {
17   return ajax(`
    http://localhost:8888/php/houdunren.php?id=
    return ajax(`
18 })
19 .then(lesson => {
20   console.log(lesson);
21 });
22

```

## #扩展接口

### #resolve

使用 `promise.resolve` 方法可以快速的返回一个 promise 对象  
根据值返回 `promise`

```

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

```

下面将请求结果缓存，如果再次请求时直接返回带值的 `promise`

- 为了演示使用了定时器，你也可以在后台设置延迟响应

```

1 function query(name) {
2   const cache = query.cache || (query.cache = new Map());
3   if (cache.has(name)) {
4     console.log("走缓存了");
5     return Promise.resolve(cache.get(name));
6   }
7   return ajax(`
  http://localhost:8888/php/user.php?name=

```

```

    return ajax(`
8      response => {
9        cache.set(name, response);
10       console.log("没走缓存");
11       return response;
12     }
13   );
14 }
15 query("向军").then(response => {
16   console.log(response);
17 });
18 setTimeout(() => {
19   query("向军").then(response => {
20     console.log(response);
21   });
22 }, 1000);
23

```

如果是 `thenable` 对象，会将对象包装成 promise 处理，这与其他 promise 处理方式一样的

```

1  const hd = {
2    then(resolve, reject) {
3      resolve("后盾人");
4    }
5  };
6  Promise.resolve(hd).then(value => {
7    console.log(value);
8  });
9

```

## #reject

和 `Promise.resolve` 类似，`reject` 生成一个失败的 `promise`

```

1  Promise.reject("fail").catch(error => console.log(error));
2

```

下面使用 `Project.reject` 设置状态

```
1 new Promise(resolve => {
2   resolve("后盾人");
3 })
4 .then(v => {
5   if (v !== "houdunren.com") return Promise.reject(new Error("fail"));
6 })
7 .catch(error => {
8   console.log(error);
9 });
10
```

## #all

使用`Promise.all`方法可以同时执行多个并行异步操作，比如页面加载时同时获取课程列表与推荐课程。

- 任何一个

Promise 执行失败就会调用

catch方法

- 适用于一次发送多个异步操作
- 参数必须是可迭代类型，如 Array/Set
- 成功后返回

promise 结果的有序数组

下例中当 `hdcms`、`houdunren` 两个 Promise 状态都为 `fulfilled` 时，`hd` 状态才为 `fulfilled`。

```
1 const hdcms = new Promise((resolve, reject) => {
2   setTimeout(() => {
3     resolve("第一个Promise");
4   }, 1000);
5 });
6 const houdunren = new Promise((resolve, reject) => {
7   setTimeout(() => {
8     resolve("第二个异步");
9   }, 1000);
10 });
11 const hd = Promise.all([hdcms, houdunren])
12   .then(results => {
13     console.log(results);
14   });
```

```

14   })
15   .catch(msg => {
16     console.log(msg);
17   });
18

```

根据用户名获取用户，有任何一个用户获取不到时 `promise.all` 状态失败，执行 `catch` 方法

```

1  function ajax(url) {
2    return new Promise((resolve, reject) => {
3      let xhr = new XMLHttpRequest();
4      xhr.open("GET", url);
5      xhr.send();
6      xhr.onload = function() {
7        if (this.status == 200) {
8          resolve(JSON.parse(this.response));
9        } else {
10         reject(this);
11       }
12     };
13   });
14 }
15
16 const api = "
http://localhost:8888/php
const api = "
17 const promises = ["向军", "后盾人"].map(name => {
18   return ajax(`${api}/user.php?name=${name}`);
19 });
20
21 Promise.all(promises)
22   .then(response => {
23     console.log(response);
24   })
25   .catch(error => {
26     console.log(error);
27   });
28

```

可以将其他非 `promise` 数据添加到 `all` 中，它将被处理成 `Promise.resolve`



```

1  ...
2  const promises = [
3    ajax(`${api}/user.php?name=向军`),
4    ajax(`${api}/user.php?name=后盾人`),
5    { id: 3, name: "hdcms", email: "admin@hdcms.com" }
6  ];
7  ...
8

```

如果某一个`promise`没有 `catch` 处理，将使用`promise.all` 的 `catch` 处理

```

1  let p1 = new Promise((resolve, reject) => {
2    resolve("fulfilled");
3  });
4  let p2 = new Promise((resolve, reject) => {
5    reject("rejected");
6  });
7  Promise.all([p1, p2]).catch(reason => {
8    console.log(reason);
9  });
10

```

## #allSettled

`allSettled` 用于处理多个`promise`，只关注执行完成，不关注是否全部执行成功，`allSettled` 状态只会是`fulfilled`。

下面的 `p2` 返回状态为 `rejected`，但`promise.allSettled` 不关心，它始终将状态设置为 `fulfilled`。

```

1  const p1 = new Promise((resolve, reject) => {
2    resolve("resolved");
3  });
4  const p2 = new Promise((resolve, reject) => {
5    reject("rejected");
6  });
7  Promise.allSettled([p1, p2])
8  .then(msg => {
9    console.log(msg);
10 })

```

下面是获取用户信息，但不关注某个用户是否获取不成功

```

1  const api = "
   http://localhost:8888/php
   const api = "
2  const promises = [
3    ajax(`${api}/user.php?name=向军`),
4    ajax(`${api}/user.php?name=后盾人`)
5  ];
6  Promise.allSettled(promises).then(response => {
7    console.log(response);
8  });
9

```

## #race

使用 `Promise.race()` 处理容错异步，和 `race` 单词一样哪个 Promise 快用哪个，哪个先返回用哪个。

- 以最快返回的 promise 为准
- 如果最快返回的状态为

rejected 那整个

promise为

rejected执行 cache

- 如果参数不是 promise，内部将自动转为 promise

下面将第一次请求的异步时间调整为两秒，这时第二个先返回就用第二人。

```

1  const hdcms = new Promise((resolve, reject) => {
2    setTimeout(() => {
3      resolve("第一个Promise");
4    }, 2000);
5  });
6  const houdunren = new Promise((resolve, reject) => {
7    setTimeout(() => {
8      resolve("第二个异步");
9    }, 1000);
10 });
11 Promise.race([hdcms, houdunren])
12 .then(results => {

```

```
13 console.log(results);
14 })
15 .catch(msg => {
16   console.log(msg);
17 });
18
```

获取用户资料，如果两秒内没有结果 `promise.race` 状态失败，执行 `catch` 方法

```
1 const api = "
  http://localhost:8888/php
  const api = "
2 const promises = [
3   ajax(`${api}/user.php?name=向军`),
4   new Promise((a, b) =>
5     setTimeout(() => b(new Error("request fail")), 2000)
6   )
7 ];
8 Promise.race(promises)
9 .then(response => {
10   console.log(response);
11 })
12 .catch(error => {
13   console.log(error);
14 });
15
```

## #任务队列

### #实现原理

如果 `then` 返回 `promise` 时，后面的 `then` 就是对返回的 `promise` 的处理

```
1 let promise = Promise.resolve();
2 let p1 = promise.then(() => {
3   return new Promise(resolve => {
4     setTimeout(() => {
5       console.log(`p1`);
6       resolve();

```

```

7     }, 1000);
8   });
9 });
10 p1.then(() => {
11   return new Promise((a, b) => {
12     setTimeout(() => {
13       console.log(`p2`);
14     }, 1000);
15   });
16 });
17

```

下面使用 `map` 构建的队列，有以下几点需要说明

- `then` 内部返回的

promise 更改外部的

promise 变量

- 为了让任务继承，执行完任务需要将

promise 状态修改为

fulfilled

```

1 function queue(nums) {
2   let promise = Promise.resolve();
3   nums.map(n => {
4     promise = promise.then(v => {
5       return new Promise(resolve => {
6         console.log(n);
7         resolve();
8       });
9     });
10  });
11 }
12
13 queue([1, 2, 3, 4, 5]);
14

```

下面再来通过 `reduce` 来实现队列

```

1 function queue(nums) {
2   return nums.reduce((promise, n) => {

```

```

3     return promise.then(() => {
4         return new Promise(resolve => {
5             console.log(n);
6             resolve();
7         });
8     });
9 }, Promise.resolve());
10 }
11
12 queue([1, 2, 3, 4, 5]);
13

```

## #队列请求

下面是异步加载用户并渲染到视图中的队列实例

- 请在后台添加延迟脚本，以观察队列执行过程
- 也可以在任何

promise 中添加定时器观察

```

1 class User {
2     //加载用户
3     ajax(user) {
4         let url = `
5         http://localhost:8888/php/user.php?name=
6         let url = `
7
8         return new Promise(resolve => {
9             let xhr = new XMLHttpRequest();
10            xhr.open("GET", url);
11            xhr.send();
12            xhr.onload = function() {
13                if (this.status == 200) {
14                    resolve(JSON.parse(this.response));
15                } else {
16                    reject(this);
17                }
18            };
19        });
20    }
21    //启动
22    render(users) {
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

```

20     users.reduce((promise, user) => {
21         return promise
22             .then(() => {
23                 return this.ajax(user);
24             })
25             .then(user => {
26                 return this.view(user);
27             });
28     }, Promise.resolve());
29 }
30 //渲染视图
31 view(user) {
32     return new Promise(resolve => {
33         let h1 = document.createElement("h1");
34         h1.innerHTML = user.name;
35         document.body.appendChild(h1);
36         resolve();
37     });
38 }
39 }
40 new User().render(["向军", "后盾人"]);
41

```

## #高可用封装

上例中处理是在队列中完成，不方便业务定制，下面将 Promise 处理在剥离到外部后台请求处理类

```

1  export default function(url) {
2      return new Promise((resolve, reject) => {
3          let xhr = new XMLHttpRequest()
4          xhr.open('GET', url)
5          xhr.send()
6          xhr.onload = function() {
7              if (this.status === 200) {
8                  resolve(this.response)
9              } else {
10                 reject(this)
11             }

```

```
12     }
13   })
14 }
15
```

## 队列处理类

```
1 export default function(promises) {
2   promises.reduce((promise, next) => promise.then(next), Promise.resolve())
3 }
4
```

## 后台脚本

```
1 <?php
2 $users = [
3     1 => "小明",
4     2 => "李四",
5     3 => "张三"
6 ];
7 sleep(1);
8 echo $users[$_GET['id']];
9
```

## 使用队列

```
1 <script type="module">
2   import queue from './queue.js'
3   import axios from './axios.js'
4   queue(
5     [1, 2, 3].map(v => () =>
6       axios(`user.php?id=${v}`).then(user => console.log(user))
7     )
8   )
9 </script>
10
```

## #async/await

使用 `async/await` 是 promise 的语法糖，可以让编写 promise 更清晰易懂，也是推荐编写 promise 的方式。

- `async/await` 本质还是 promise，只是更简洁的语法糖书写
- `async/await` 使用更清晰的 promise 来替换 `promise.then/catch` 的方式

## #async

下面在 `hd` 函数前加上 `async`，函数将返回 promise，我们就可以像使用标准 Promise 一样使用了。

```
1 async function hd() {
2   return "houdunren.com";
3 }
4 console.log(hd());
5 hd().then(value => {
6   console.log(value);
7 });
8
```

如果有多个 `await` 需要排队执行完成，我们可以很方便的处理多个异步队列

```
1 async function hd(message) {
2   return new Promise(resolve => {
3     setTimeout(() => {
4       resolve(message);
5     }, 2000);
6   });
7 }
8 async function run() {
9   let h1 = await hd("后盾人");
10  console.log(h1);
11  let h2 = await hd("houdunren.com");
12  console.log(h2);
13 }
14 run();
15
```

## #await

使用 `await` 关键词后会等待 promise 完



- await 后面一般是 promise，如果不是直接返回
- await 必须放在 async 定义的函数中使用
- await 用于替代

then 使编码更优雅

下例会在 await 这行暂停执行，直到等待 promise 返回结果后才继续执行。

```

1  async function hd() {
2    const promise = new Promise((resolve, reject) => {
3      setTimeout(() => {
4        resolve("houdunren.com");
5      }, 2000);
6    });
7    let result = await promise;
8    console.log(result);
9  }
10 hd()
11

```

一般 await 后面是外部其它的 promise 对象

```

1  async function hd() {
2    return new Promise(resolve => {
3      setTimeout(() => {
4        resolve("fulfilled");
5      }, 2000);
6    });
7  }
8  async function run() {
9    let value = await hd();
10   console.log("houdunren.com");
11   console.log(value);
12 }
13 run();
14

```

下面是请求后台获取用户课程成绩的示例

```

1  async function user() {

```

```

2   let user = await ajax(`
  http://localhost:8888/php/user.php?name=
    let user = await ajax(`
3   let lessons = await ajax(
4     `
  http://localhost:8888/php/houdunren.php?id=
    `
5   );
6   console.log(lessons);
7 }
8

```

也可以将操作放在立即执行函数中完成

```

1 (async () => {
2   let user = await ajax(`
  http://localhost:8888/php/user.php?name=
    let user = await ajax(`
3   let lessons = await ajax(
4     `
  http://localhost:8888/php/houdunren.php?id=
    `
5   );
6   console.log(lessons);
7 })();
8

```

下面是使用 `async` 设置定时器，并间隔时间来输出内容

```

1 async function sleep(ms = 2000) {
2   return new Promise(resolve => {
3     setTimeout(resolve, ms);
4   });
5 }
6 async function run() {
7   for (const value of ["后盾人", "向军"]) {
8     await sleep();
9     console.log(value);
10  }
11 }
12 run();

```

## #加载进度

下面是请求后台加载用户并通过进度条展示的效果

```

1  <body>
2    <style>
3      div {
4        height: 50px;
5        width: 0px;
6        background: green;
7      }
8    </style>
9    <div id="loading"></div>
10 </body>
11 <script src="js/ajax.js"></script>
12 <script>
13   async function query(name) {
14     return ajax(`
15       http://localhost:8888/php/user.php?name=
16       return ajax(`
17   }
18   (async () => {
19     let users = ["后盾人", "向军", "李四", "王五", "赵六"];
20     for (let i = 0; i < users.length; i++) {
21       await query(users[i]);
22       let progress = (i + 1) / users.length;
23       loading.style.width = progress * 100 + "%";
24     }
25   })();
26 </script>

```

## #类中使用

和 promise 一样，await 也可以操作 `thenables` 对象

```

1  class User {

```

```

2   (name) {
3       this.name = name;
4   }
5   then(resolve, reject) {
6       let user = ajax(`
http://localhost:8888/php/user.php?name=
        let user = ajax(`
7       resolve(user);
8   }
9 }
10 async function get() {
11     let user = await new User("向军");
12     console.log(user);
13 }
14 get();
15

```

类方法也可以通过 `async` 与 `await` 来操作 promise

```

1 class User {
2     () {}
3     async get(name) {
4         let user = await ajax(
5             `
http://localhost:8888/php/user.php?name=
            `
6         );
7         user.name += "-houdunren.com";
8         return user;
9     }
10 }
11 new User().get("向军").then(resolve => {
12     console.log(resolve);
13 });
14

```

## #其他声明

函数声明

```
1 async function get(name) {  
2   return await ajax(`  
  http://localhost:8888/php/user.php?name=  
    return await ajax(`  
3 }  
4 get("后盾人").then(user => {  
5   console.log(user);  
6 });  
7
```

## 函数表达式

```
1 let get = async function(name) {  
2   return await ajax(`  
  http://localhost:8888/php/user.php?name=  
    return await ajax(`  
3 };  
4 get("后盾人").then(user => {  
5   console.log(user);  
6 });  
7
```

## 对象方法声明

```
1 let hd = {  
2   async get(name) {  
3     return await ajax(`  
      http://localhost:8888/php/user.php?name=  
        return await ajax(`  
4   }  
5 };  
6  
7 hd.get("后盾人").then(user => {  
8   console.log(user);  
9 });  
10
```

## 立即执行函数

```

1 (async () => {
2   let user = await ajax(`
   http://localhost:8888/php/user.php?name=
   let user = await ajax(`
3   let lessons = await ajax(
4     `
   http://localhost:8888/php/houdunren.php?id=
   `
5   );
6   console.log(lessons);
7 })();
8

```

类方法中的使用

```

1 class User {
2   async get(name) {
3     return await ajax(`
   http://localhost:8888/php/user.php?name=
   return await ajax(`
4   }
5 }
6 let user = new User().get("后盾人").then(user => {
7   console.log(user);
8 });
9

```

## #错误处理

async 内部发生的错误，会将必变 promise 对象为 rejected 状态，所以可以使用 `catch` 来处理

```

1 async function hd() {
2   console.log(houdunren);
3 }
4 hd().catch(error => {
5   throw new Error(error);
6 });
7

```

下面是异步请求数据不存在时的错误处理

```

1  async function get(name) {
2    return await ajax(`
    http://localhost:8888/php/user.php?name=
    return await ajax(`
3  }
4
5  get("向军小哥").catch(error => {
6    alert("用户不存在");
7  });
8

```

如果 `promise` 被拒绝将抛出异常，可以使用 `try...catch` 处理错误

```

1  async function get(name) {
2    try {
3      let user = await ajax(
4        `
    http://localhost:8888/php/user.php?name=
5      `);
6      console.log(user);
7    } catch (error) {
8      alert("用户不存在");
9    }
10 }
11 get("向军老师");
12

```

多个 `await` 时当前面的出现失败，后面的将不可以执行

```

1  async function hd() {
2    await Promise.reject("fail");
3    await Promise.resolve("success").then(value => {
4      console.log(value);
5    });
6  }
7  hd();
8

```

如果对前一个错误进行了处理，后面的 `await` 可以继续执行

```

1  async function hd() {
2    await Promise.reject("fail").catch(e => console.log(e));
3    await Promise.resolve("success").then(value => {
4      console.log(value);
5    });
6  }
7  hd();
8

```

也可以使用 `try...catch` 特性忽略不必要的错误

```

1  async function hd() {
2    try {
3      await Promise.reject("fail");
4    } catch (error) {}
5    await Promise.resolve("success").then(value => {
6      console.log(value);
7    });
8  }
9  hd();
10

```

也可以将多个 `await` 放在 `try...catch` 中统一处理错误

```

1  async function hd(name) {
2    const host = "
http://localhost:8888/php
    const host = "
3    try {
4      const user = await ajax(`${host}/user.php?name=${name}`);
5      const lessons = await ajax(`${host}/user.php?id=${user.id}`);
6      console.log(lessons);
7    } catch (error) {
8      console.log("用户不存在");
9    }
10  }
11  hd("后盾人教程");
12

```



## #并发执行

有时需要多个 `await` 同时执行，有以下几种方法处理，下面多个 `await` 将产生等待

```
1  async function p1() {
2    return new Promise(resolve => {
3      setTimeout(() => {
4        console.log("houdunren");
5        resolve();
6      }, 2000);
7    });
8  }
9  async function p2() {
10   return new Promise(resolve => {
11     setTimeout(() => {
12       console.log("hdcms");
13       resolve();
14     }, 2000);
15   });
16 }
17 async function hd() {
18   await p1();
19   await p2();
20 }
21 hd();
22
```

使用 `Promise.all()` 处理多个 promise 并行执行

```
1  async function hd() {
2    await Promise.all([p1(), p2()]);
3  }
4  hd();
5
```

让 promise 先执行后再使用 `await` 处理结果

```
1  async function hd() {
2    let h1 = p1();
```

```
3   let h2 = p2();  
4   await h1;  
5   await h2;  
6 }  
7 hd();
```