JS 异步编程

概述

1. 采用单线程模式的工作原因：早期的JS主要是用来做DOM操作的
2. 单线程模式意味着只有一个队列在执行任务
    a. 优点：更安全
    b. 缺点：当遇到长时间的任务时，后续的代码就需要等待很久才能执行

同步模式

1. 顺序执行，一步一步地执行

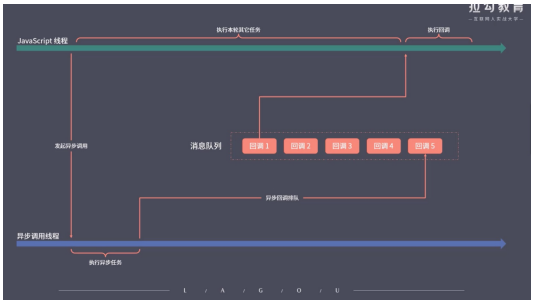
异步模式

1. 不会去等待这个任务的结束才开始执行下一个任务
2. 后续逻辑一般会通过回调函数的方式定义

3. 这个让单线程的JS语法可以同时处理大量耗时任务

三个关键字：Call stack、Web APIS、Event Loop的Queue;
JS线程执行

1. 遇到异步调用时，就向异步调用线程发起任务
2. 异步调用线程执行异步任务，将回调放入消息队列中
3. 当JS线程执行完毕后，再开始依次执行消息队列内的回调

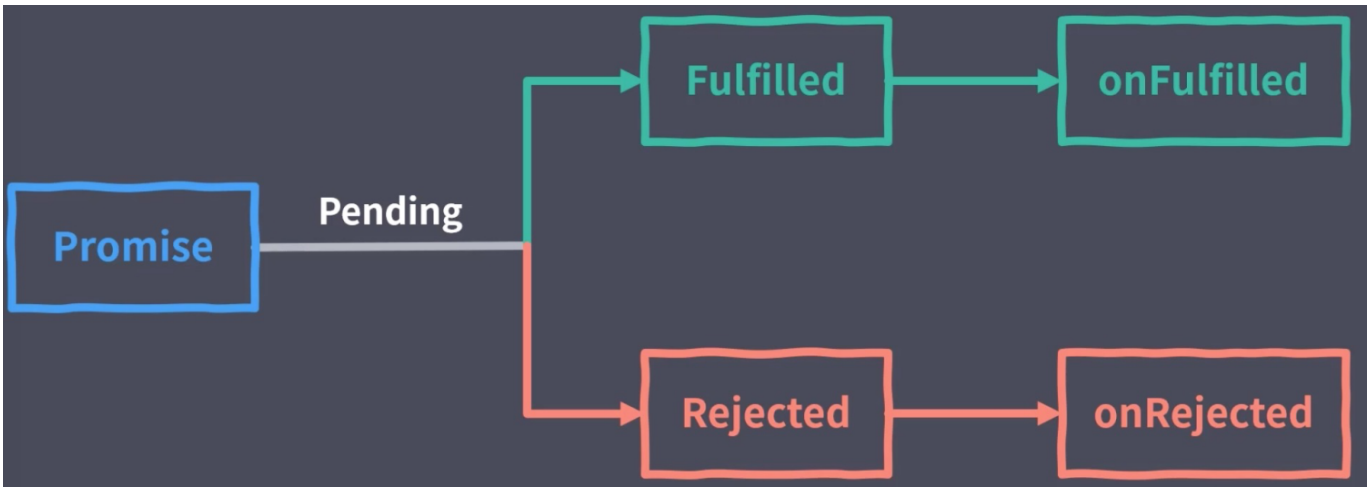


回调函数

1. 由调用者定义，交给执行者执行的函数，就叫做回调函数

```
function foo() {
  console.log('我是延迟1s被打印的');
}

setTimeout(foo, 1000);
```

Promise 概述



Promise 基本用法

```
const promise = new Promise(function (resolve, reject) {
  resolve(100);

  // reject(new Error('promise rejected'));
});

promise.then(function (value) {
  console.log('resolved', value); // resolve时打印
}, function (error) {
  console.log('rejected', error); // reject时打印
})
```

Promise 使用案例

webpack的安装

```
npm init
yarn add webpack webpack-cli webpack-dev-server html-webpack-plugin -D
# 开启服务命令
yarn webpack-dev-server xxx.js --open
```

webpakc.config.js的配置

```
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  mode: 'none',
  stats: 'none',
  devtool: 'source-map',
  plugins: [
    new HtmlWebpackPlugin()
  ]
}
```

JavaScript的代码

```
// Promise 方式的 AJAX
function ajax (url) {
  return new Promise(function (resolve, reject) {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', url);
    xhr.responseType = 'json';
    xhr.onload = function () {
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    }
    xhr.send();
  });
}

ajax('/api/users.json').then(function (res) {
  console.log(res);
}, function (error) {
  console.log(error);
})
```

Promise 常见误区

1. Promise本质上也是使用回调函数，来定义异步任务结束后需要执行的任务
2. 如果Promise.then也进行了Promise任务，请用链式调用来代替

Promise 链式调用

1. Promise对象的then方法会返回一个全新的Promise对象
2. 后面的then方法就是在为上一个then返回的Promise注册回调
3. 前面then方法中回调函数的返回值会作为后面then方法回调的参数
4. 如果回调中返回的是Promise，那后面then方法的回调会等待它的结束

```
let i = 0;
var promise = new Promise((resolve, reject) => {
  resolve(i++);
});

while(i++ < 5) {
```

```
  promise = promise.then(val => {
    console.log(val++);

    return val;
  });
}
// 打印：0、1、2、3、4
```

Promise 异常处理

有两种方式：

1. 第一个.then中，添加第二个回调函数来监听错误
2. 添加.catch(callback);

它们的区别是，第一种，当.then中也执行了Promise并且有错误，无法捕获到；第二种却可以，因为它就像是给整个Promise.then链条添加了一个捕获函数

```
ajax('/api/users.json').then(function (res) {
  return ajax('/error.json');
}, function (error) {
  console.log('error: ', error); // 无打印
})

ajax('/api/users.json').then(function (res) {
  return ajax('/error-url');
}).catch(error => {
  console.log('error: ', error); // error:  Error: Not Found
})
```

Promise 静态方法

1. Promise.resolve()
2. Promise.reject()

```
Promise.resolve('foo')
  .then(value => {
    console.log(value); // foo
  })

var promise = new Promise(function (resolve, reject) {
```

```
  resolve('foo2');
});
var promise2 = Promise.resolve(promise);
console.log(promise2 === promise); // true

Promise.resolve({
  then: function (onFulfilled, onRejected) {
    onFulfilled('foo3');
  }
})
.then(function (value) {
  console.log(value); // foo3
})

Promise.reject('foo4')
  .catch(function (error) {
    console.log('foo4'); // foo4
  })
```

Promise 并行执行

1. Promise.all(Array)：可以将多个promise合并为一个promise，等待所有任务结束后，.then就会触发
2. Promise.race(): 可以将多个promise合并为一个promise,，等待任意第一个任务结束后，.then就会触发

```
// ajax方法请参照“Promise 使用案例”中的代码
Promise.all([
  ajax('/api/users.json'),
  ajax('/api/posts.json')
]).then(value => {
  console.log(value); // [{name: "wubo", age: 27}, {post: 123}]
})
```

```
// 一个请求超时时，停止的方法
const timeout = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error('timeout'), 500));
})
Promise.race([
```

```
  ajax('/api/posts.json'),
  timeout,
]).then(value => {
  console.log(value);
}).catch(error => {
  console.log(error);
})
```

Promise 执行时序

宏任务 vs. 微任务

Promise.then是一个微任务（本轮调用的末尾直接执行，而不是进入队列等待），setTimeout是宏任务，这是JS为了提高整体的响应速度而做的特性

```
console.log('global start');

setTimeout(() => console.log('time'), 0);

Promise.resolve()
    .then(() => {
        console.log('r1');
    })
    .then(() => {
        console.log('r2');
    })

console.log('global end');
/*
global start
global end
r1
r2
time
*/
```

Generator 异步方案（上）

1. 更优秀的异步编写语法
2. .next()传入的参数，会作为yield的返回值

```
function * foo() {
  console.log('start');

  try {
    const res = yield 'foo';
    console.log(res);
  } catch (e) {
    console.log('error: ', e);
  }
}

const generator = foo();

const result = generator.next();
console.log(result);

// generator.next('bar'); // 第6行打印bar
generator.next(new Error('Generator error')); // 触发上面的catch
```

Generator 异步方案（中）

```
function * main () {
  const users = yield ajax('/api/users.json');
  console.log(users);

  const posts = yield aja('/api/posts.json');
  console.log(posts);
}

const g = main();

const result = g.next();

result.value.then(data => {
  const results2 = g.next(data); // 触发第3行的打印

  if (results2.done) return;

```

```
  results2.value.then(data => {
    const result3 = g.next(data); // 触发第6行的打印
  })
})
```

Generator 异步方案（下）

```
function * main () {
  try {
    const users = yield ajax('/api/users.json');
    console.log(users);

    const posts = yield ajax('/api/posts.json');
    console.log(posts);

    yiled ajax('/api/user-1.json');
  } catch (e) {
    console.log(e);
  }
}

const g = main();

function handleResult (result) {
  if (result.done) return;

  result.value.then(data => {
    handleResult(g.next(data));
  }, error => {
    g.throw(error);
  })
}

handleResult(g.next());
```

```
// co示例
function co (generator) {
```

```
  const g = generator();

  function handleResult (result) {
    if (result.done) return;

    result.value.then(data => {
      handleResult(g.next(data));
    }, error => {
      g.throw(error);
    })
  }

  handleResult(g.next());
}

co(main);
```

Async 函数

1. Async / Await 语法糖，语言层面的异步编程标准

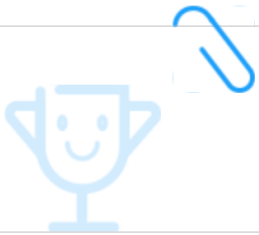
```
async function main () {
  try {
    const users = await ajax('/api/users.json');
    console.log(users);

    const posts = await aja('/api/posts.json');
    console.log(posts);

    const urls = await ajax('/api/urls.json');
    console.log(urls);
  } catch (e) {
    console.log(e);
  }
}
```

随堂测验

JavaScript 异步编程随堂测试

24
名次

战胜了75%的人！
参与人数：96



成绩单

答题人	您的得分	答对题数
吴博	50/55	10/11

答题解析

全部题目 | 错题集

您的姓名：*

您的回答：吴博

您的手机号：*

您的回答：17671686923

1.以下关于JavaScript的说法中正确的是：（） [分值：5]

您的回答：A.JavaScript是单线程语言，方便进行DOM操作。┇B.JavaScript的异步操作常见的有计时器、事件绑定、Ajax

✓ （得分：5）

2.下面哪些方法可以实现 JavaScript 异步编程？（） [分值：5]

您的回答：B.事件监听┇C.发布/订阅┇D.Promise对象 ✗

正确答案为：A.回调函数|B.事件监听|C.发布/订阅|D.Promise对象

3.关于Promise对象的状态，下列说法错误的是：（） [分值：5]

您的回答：D.Rejected失败可以改变成Fulfilled成功 ✓ （得分：5）

4.关于Generator函数的描述，错误的是：（） [分值：5]

您的回答：D.使用return语句使Generator函数暂停执行，直到next方法的调用 ✓ （得分：5）

5.下列代码的输出结果（）

```
1 console.log("1")
2 setTimeout(function () {
3   console.log("2")
```

```
}, 0)
console.log("3")
setTimeout(function () {
  console.log("4")
}, 1000)
console.log("5")
```

[分值：5]

您的回答：A.13524 ✓ （得分：5）

6.以下代码块的输出结果（）

```
const promise = new Promise((resolve, reject) => {
  console.log(1)
  resolve()
  console.log(2)
})
promise.then(() => {
  console.log(3)
})
console.log(4)
```

[分值：5]

您的回答：B.1243 ✓ （得分：5）

7.下面程序的正确输出结果是（）

```
Promise.resolve(1)
  .then((res) => {
    console.log(res)
    return 2
  })
  .catch((err) => {
    return 3
  })
  .then((res) => {
    console.log(res)
  })
```

[分值：5]

您的回答：C.12 ✓ （得分：5）

8.下列代码的执行结果为

```
const promise = new Promise((resolve, reject) => {
  resolve("success1")
  reject("error")
  resolve("success2")
})
promise
  .then((res) => {
```

[分值：5]

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

您的回答：A.then: success1 ✓ （得分：5）

9.下列代码的运行结果是（）

Promise.resolve(1).then(2).then(Promise.resolve(3)).then(console.log) [分值：5]

您的回答：A.1 ✓ （得分：5）

10.Generator函数的yield关键字的作用是： （） [分值：5]

您的回答：C.暂停执行，等待生成器对象的next( )方法调用 ✓ （得分：5）

11.下列叙述中不正确的是： （） [分值：5]

您的回答：B.next()方法执行的参数是不会当作yield表达式的返回值。┇C.next()方法的返回对象中的value是值，done是传递参数。 ✓ （得分：5）
