

实验 1 Javascript 基础快速回顾

一、实验目的

1. 学习并使用基本的 Javascript 语言
2. 理解 Promise 和 async/await 异步操作

二、实验原理

1. JavaScript
2. 异步操作 asynchronous operations

三、实验仪器材料

1. 计算机实训室电脑一台
2. VSCode
3. Chrome 浏览器

四、实验步骤

先在 VSCode 中打开一个空的目录，如 D:\learn\React，然后在菜单 Terminal > New Terminal... 打开一个新的终端，运行：

```
npm init -y
```



```
D:\learn\React>npm init -y
Wrote to D:\learn\React\package.json:

{
  "name": "react",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

此时会在目录下创建一个 package.json 文件。后续我们所有的 Javascript 例子就在这里新建 .js 文件，然后通过 node xxx.js 来运行。

4.1 Javascript 基础

4.1.1 变量声明 let 与 const

实例 example_01.js:

```

// 用 let 声明一个可能会改变的变量

let score = 10;

console.log("初始分数:", score); // 输出: 初始分数: 10

score = 15;

console.log("更新后分数:", score); // 输出: 更新后分数: 15


// 用 const 声明一个常量 (不会改变的值)

// 在 React 中, 我们优先使用 const, 除非我们知道这个变量需要被重新赋值。

const name = "张三";

console.log("姓名:", name); // 输出: 姓名: 张三


// 尝试修改 const 声明的变量会报错, 这能帮助我们避免意外修改数据

// name = "李四"; // 取消这行注释会看到错误

```

运行结果:

```

D:\learn\React\basicjs>node example_01.js
初始分数: 10
更新后分数: 15
姓名: 张三

```

4.1.2 核心数据类型: 对象 (Object) 与数组 (Array)

实例: example_02.js

```

// 对象: 用大括号 {} 创建, 存储键值对信息

const user = {

  id: 1,

  name: "Alice",

  isAdmin: true

};


// 访问对象的属性

console.log("用户名:", user.name); // 输出: 用户名: Alice

console.log("是管理员吗:", user.isAdmin); // 输出: 是管理员吗: true


// 数组: 用中括号 [] 创建, 存储一个有序的列表

const topics = ["React", "JavaScript", "CSS"];


// 访问数组的元素 (索引从 0 开始)

console.log("第一个主题:", topics[0]); // 输出: 第一个主题: React

console.log("数组长度:", topics.length); // 输出: 数组长度: 3

```

运行结果：

```
D:\learn\React\basicjs>node example_02.js
用户名: Alice
是管理员吗: true
第一个主题: React
数组长度: 3
```

4.1.3 函数，箭头函数 =>

实例：example_03.js

```
// 传统函数
function sayHello(name) {
  return `你好, ${name}!`;
}

console.log(sayHello("传统函数"));

// 箭头函数 (Arrow Function) - React 中最常用的写法，更简洁！
const sayHelloArrow = (name) => {
  return `你好, ${name}!`;
};

console.log(sayHelloArrow("箭头函数"));

// 如果函数体只有一行返回语句，可以更简洁：
const add = (a, b) => a + b;

console.log("2 + 3 =", add(2, 3)); // 输出: 2 + 3 = 5

// 在 React 中，你会这样写一个点击事件处理函数：
const handleClick = () => {
  console.log("按钮被点击了！");
};

handleClick();
```

运行结果：

```
D:\learn\React\basicjs>node example_03.js
你好, 传统函数!
你好, 箭头函数!
2 + 3 = 5
按钮被点击了!
```

4.1.4 数组的 .map() 方法

实例：example_04.js

```

const numbers = [1, 2, 3, 4];

// .map 会遍历数组的每一项，并根据你提供的函数返回一个“新的”数组
const doubledNumbers = numbers.map((number) => {
  return number * 2;
});

console.log("原数组:", numbers);           // 输出: 原数组: [1, 2, 3, 4]
console.log("新数组 (翻倍后):", doubledNumbers); // 输出: 新数组 (翻倍后): [2, 4, 6, 8]

// 在 React 中，你会这样做：
const userNames = ["Alice", "Bob", "Charlie"];

// 假设这是 React 的 JSX 写法，现在我们用字符串模拟
const userListItem = userNames.map((name) => {
  return `<li>${name}</li>`; // 把每个名字字符串，转换成一个 <li> 字符串
});

console.log(userListItem);

// 输出: ["<li>Alice</li>", "<li>Bob</li>", "<li>Charlie</li>"]
// 在 React 里，这些字符串就会被渲染成一个真正的列表！

```

运行结果：

```

D:\learn\React\basicjs>node example_04.js
原数组: [ 1, 2, 3, 4 ]
新数组 (翻倍后): [ 2, 4, 6, 8 ]
[ '<li>Alice</li>', '<li>Bob</li>', '<li>Charlie</li>' ]

```

4.1.5 对象解构（Destructuring）与展开语法（Spread Syntax）

实例：example_05.js

```

const user = {
  name: "David",
  age: 30,
  email: "david@example.com"
};

// 传统方式获取值
// const name = user.name;
// const age = user.age;

// 使用解构，一行搞定！
const { name, age } = user;
console.log(`姓名: ${name}, 年龄: ${age}`); // 输出: 姓名: David, 年龄: 30

```

```
// --- 展开语法 (...) ---
// 1. 用于复制对象/数组 (React 中更新 state 的核心技巧)
const numbers = [1, 2, 3];
const newNumbers = [...numbers, 4, 5]; // 复制 numbers, 并在后面加上 4, 5
console.log(newNumbers); // 输出: [1, 2, 3, 4, 5]

// 2. 更新对象的属性
const updatedUser = {
  ...user, // 复制 user 对象的所有属性
  age: 31   // 然后用新的 age 覆盖旧的
};
console.log(updatedUser); // 输出: {name: "David", age: 31, email: "david@example.com"}
```

运行结果:

```
D:\learn\React\basicjs>node example_05.js
姓名: David, 年龄: 30
[ 1, 2, 3, 4, 5 ]
{ name: 'David', age: 31, email: 'david@example.com' }
```

4. 1. 6 条件判断：三元运算符

实例: example_06. js

```
const isLoggedIn = false;

// 传统 if/else
let message;
if (isLoggedIn) {
  message = "欢迎回来! ";
} else {
  message = "请先登录。";
}
console.log(message); // 输出: 请先登录。

// 使用三元运算符: condition ? value_if_true : value_if_false
const ternaryMessage = isLoggedIn ? "欢迎回来! " : "请先登录。";
console.log(ternaryMessage); // 输出: 请先登录。

// 在 React 中会这样用:
// { isLoggedIn ? <UserProfile /> : <LoginForm /> }
```

运行结果：

```
D:\learn\React\basicjs>node example_06.js
请先登录。
请先登录。

D:\learn\React\basicjs>node example_06.js
欢迎回来!
欢迎回来!
```

4.1.7 综合以上知识点的小例子

实例：example_07.js

```
// 1. 数据：一个用户对象数组
const users = [
  { id: 1, name: "Alice", isActive: true },
  { id: 2, name: "Bob", isActive: false },
  { id: 3, name: "Charlie", isActive: true }
];

// 2. 目标：生成一个问候语列表，只包含活跃用户，并给每个人加上 "Mr." 前缀

// 3. 任务分解
const activeUsers = users.filter((user) => user.isActive === true); // .filter() 用于筛选

const greetings = activeUsers.map((user) => {
  // 4. 解构获取 name
  const { name } = user;
  // 5. 箭头函数 + 字符串模板
  return `Hello, Mr. ${name}!`;
});

// 6. 输出最终结果
console.log(greetings);
// 输出: ["Hello, Mr. Alice!", "Hello, Mr. Charlie!"]
```

运行结果：

```
D:\learn\React\basicjs>node example_07.js
[ 'Hello, Mr. Alice!', 'Hello, Mr. Charlie!' ]
```

4.1.8 Javascript 中的循环

实例：example_08.js

```
// for 循环

for (let i = 0; i < 5; i++) {
  console.log("数字是 " + i);
}


// for...of 循环:
// 用于遍历可迭代对象，例如数组、字符串等。它直接访问每个元素的值，而不是索引

const arr = [1, 2, 3, 4, 5];
for (const num of arr) {
  console.log(num);
}


const colors = ["red", "green", "blue"];
for (const color of colors) {
  console.log(color);
}


// for...in 循环:
// 用于遍历对象的属性，而不是遍历对象中的值。它返回对象的键（属性名）。

const person = {
  name: "张三",
  age: 30,
  city: "北京"
};

for (const key in person) {
  console.log(key + ": " + person[key]);
}


// while 循环:
// 循环条件为 true 时执行代码块。

let i = 0;
while (i < 5) {
  console.log("数字是 " + i);
  i++;
}


// do...while 循环:
// 和 while 循环类似，但 ensure 循环至少会执行一次。

let j = 0;
do {
  console.log("数字是 " + j);
  j++;
}
```

```
} while (j < 5);
```

运行结果：

```
D:\learn\React\basicjs>node example_08.js
数字是 0
数字是 1
数字是 2
数字是 3
数字是 4
1
2
3
4
5
red
green
blue
name: 张三
age: 30
city: 北京
数字是 0
数字是 1
数字是 2
数字是 3
数字是 4
数字是 0
数字是 1
数字是 2
数字是 3
数字是 4
```

4.1.9 类 (Class) 和面向对象编程 (OOP)

虽然 JavaScript 是一种基于原型的语言,但 ES6 引入的 `class` 语法让创建类变得更加直观,它本质上是现有原型继承模式的一个语法糖,但对于初学者来说,它更符合传统面向对象语言的习惯。

实例: `example_09.js`

```
class Person {
  // 构造函数,用于初始化对象的属性
  constructor(name, age) {
    this.name = name; // this 指向新创建的对象实例
    this.age = age;
  }

  // 类的方法
  sayHello() {
    console.log(`大家好,我叫 ${this.name}, 今年 ${this.age} 岁。`);
  }

  // 另一个方法
  getAge() {
    return this.age;
  }
}
```



```

}

// 使用 new 关键字创建 Person 类的实例
const person1 = new Person('张三', 30);
const person2 = new Person('李四', 25);

// 调用实例的方法
person1.sayHello(); // 输出: 大家好, 我叫 张三, 今年 30 岁。
console.log(person2.getAge()); // 输出: 25

```

运行结果:

```

D:\learn\React\basicjs>node example_09.js
大家好, 我叫 张三, 今年 30 岁。
25

```

类的继承 (Inheritance)

继承是一种面向对象编程的基本概念, 它允许一个类 (子类) 继承另一个类 (父类) 的属性和方法。这有助于代码重用和构建更复杂的类层次结构。

JS 中也是使用 `extends` 关键字来声明子类继承自哪个父类。在子类的 `constructor` 方法中, 必须先调用 `super()` 来调用父类的构造函数, 并传入父类需要的参数。在子类的方法中, `super` 还可以用来调用父类中被重写的方法。

实例: 创建一个继承自 `Person` 的 `Student` 类, 我们希望 `Student` 类除了有 `name` 和 `age` 属性外, 还能有 `studentId`。

在 `example_09.js` 后加上:

```

class Student extends Person {
  // 子类的构造函数
  constructor(name, age, studentId) {
    // 必须先调用 super() 来初始化父类的属性
    super(name, age);

    // 然后再初始化子类特有的属性
    this.studentId = studentId;
  }

  // 子类可以拥有自己的新方法
  study() {
    console.log(`${this.name} 正在努力学习...`);
  }
}

```

```

// 子类可以重写（Override）父类的方法
sayHello() {
  // 调用父类的 sayHello 方法，并增加自己的逻辑
  super.sayHello();
  console.log(`我的学号是 ${this.studentId}。`);
}
}

// 创建一个 Student 类的实例
const student1 = new Student('王五', 20, '2023001');

// 调用子类的方法
student1.sayHello();

/* 输出：
    大家好，我叫 王五，今年 20 岁。
    我的学号是 2023001。
*/

student1.study(); // 输出：王五 正在努力学习...

// 继承了父类的方法
console.log(student1.getAge()); // 输出：20

```

运行结果：

```

D:\learn\React\basicjs>node example_09.js
大家好，我叫 张三，今年 30 岁。
25
大家好，我叫 王五，今年 20 岁。
我的学号是 2023001。
王五 正在努力学习...
20

```

4. 1. 10 模块化

模块化是现代 JavaScript 开发中至关重要的一个概念。它帮助我们复杂的代码库分解成一个个独立、可复用的小块，从而提高代码的组织性和可维护性。

模块化就是将代码拆分到不同的文件中，每个文件都是一个模块。一个模块可以**导出（export）**变量、函数、类等，供其他模块使用；同时也可以**导入（import）**其他模块导出的内容。

这解决了以下几个问题：

- 命名冲突：模块内的变量和函数不会污染全局作用域。
- 代码复用：可以方便地在不同项目中导入和使用模块。
- 可维护性：代码逻辑清晰，易于管理。

export：导出模块成员

export 语句用于从一个模块中导出成员。有两种主要的导出方式：**命名导出**和**默认导出**。

命名导出 (Named Exports)：你可以导出多个成员，并在导入时使用相同的名称。

在 math.js 文件中：

```
// 导出常量
export const PI = 3.14159;

// 导出函数
export function add(a, b) {
  return a + b;
}

// 导出类
export class Calculator {
  multiply(a, b) {
    return a * b;
  }
}
```

默认导出 (Default Export)：每个模块只能有一个默认导出。默认导出通常用于导出模块的“主要”功能。

在 user.js 文件中：

```
// 导出一个默认的函数
export default function getUser() {
  return { name: "张三", id: 101 };
}

// 注意：同一个文件可以同时有命名导出和默认导出
export const userRole = "admin";
```

import: 导入模块成员

import 语句用于从其他模块中导入导出的成员。

导入命名导出: 导入时**需要使用**成员的**精确名称**，并用**大括号 {}** 包裹。

在 app.js 文件中:

```
import { PI, add, Calculator } from './math.js';
```

```
console.log(PI); // 3.14159
```

```
console.log(add(5, 3)); // 8
```

```
const calc = new Calculator();
```

```
console.log(calc.multiply(4, 5)); // 20
```

也可以使用 as 关键字为导入的成员指定别名，在后面加上代码:

```
import { PI as mathPI } from './math.js';
```

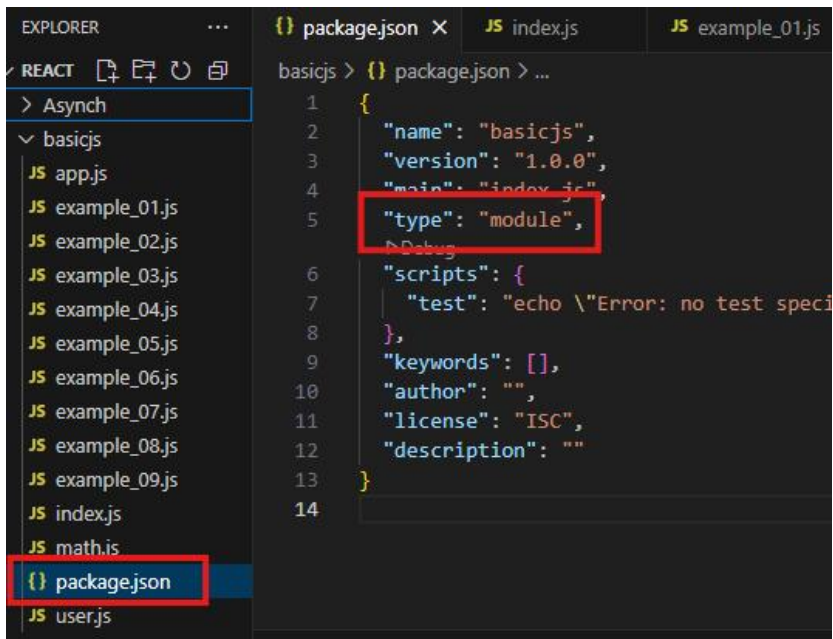
```
console.log(mathPI); // 3.14159
```

运行结果:

```
D:\learn\React\basicjs>node app.js
(node:67044) [MODULE_TYPELESS_PACKAGE_JSON] Warning: Module type of file:///D:/learn/React/basicjs/app.js is not specified and it doesn't parse as CommonJS.
Reparsing as ES module because module syntax was detected. This incurs a performance overhead.
To eliminate this warning, add "type": "module" to D:\learn\React\basicjs\package.json.
(Use `node --trace-warnings ...` to show where the warning was created)
3.14159
8
20
3.14159
```

对于这个 Warning, 我们需要在 package.json 中加上这句话:

"type": "module",



再次运行：

```
D:\learn\React\basicjs>node app.js
3.14159
8
20
3.14159
```

导入默认导出

导入默认导出的成员时，你可以给它任意的名称，不需要使用大括号。

在 app.js 文件中加上：

```
import userProfile from './user.js';
import { userRole } from './user.js';

const user = userProfile();
console.log(user); // { name: "张三", id: 101 }
console.log(userRole); // "admin"
```

运行结果：

```
D:\learn\React\basicjs>node app.js
3.14159
8
20
3.14159
{ name: '张三', id: 101 }
admin
```

导入所有成员

你可以使用 `*` 语法将一个模块的所有命名导出导入为一个对象。

在 `app.js` 文件中加上：

```
import * as MathUtils from './math.js';

console.log(MathUtils.PI); // 3.14159
console.log(MathUtils.add(10, 20)); // 30
```

运行结果：

```
D:\learn\React\basicjs>node app.js
3.14159
8
20
3.14159
{ name: '张三', id: 101 }
admin
3.14159
30
```

4.1.11 `==` vs `===`：强制类型转换的“陷阱”

`===` (严格相等)：它会比较值和类型。只有当两个操作数的值和类型都完全相同时，它才返回 `true`。这是进行比较时强烈推荐使用的运算符，因为它行为可预测，能有效避免类型转换带来的意外。

文件 `compare_01.js`：

```
console.log(10 === 10);    // true (值和类型都相同)
console.log('10' === 10); // false (值相同，但类型不同)
console.log(true === 1);   // false (值和类型都不同)
console.log(null === undefined); // false (类型不同)
```

运行结果：

```
D:\learn\React\basicjs>node compare_01.js
true
false
false
false
```

`==` (宽松相等)：它只比较值，如果两个操作数的类型不同，它会在比较前尝试将它们转换成相同的类型。这个过程被称为强制类型转换 (Type Coercion)。

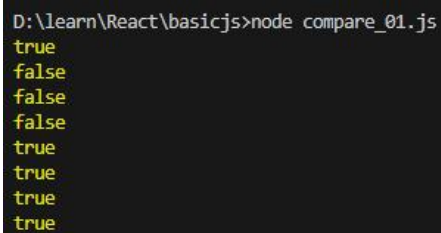
在 compare_01.js 后面加上代码：

```
console.log('10' == 10);    // true (JavaScript 悄悄地把 '10' 转换成了 10)
console.log(true == 1);     // true (JavaScript 把 true 转换成了 1)
console.log(false == 0);    // true (JavaScript 把 false 转换成了 0)
console.log(null == undefined); // true (这是一个特殊情况，它们之间互相相等)
```

注意： 尽管 `null == undefined` 返回 `true`，但 `null === undefined` 返回 `false`。

这再次证明了 `===` 的严谨性。

运行结果：



```
D:\learn\React\basicjs>node compare_01.js
true
false
false
false
true
true
true
```

类型转换：了解隐式和显式转换

显式类型转换 (Explicit Coercion)

这是你主动进行的转换，代码意图非常清晰。通常使用 `String()`、`Number()`、`Boolean()` 等函数。

在 coercoin_01.js 中：

```
let num = 10;
let str = '20';

// 将数字显式转换为字符串
let newStr = String(num); // "10"
console.log(typeof newStr); // "string"

// 将字符串显式转换为数字
let newNum = Number(str); // 20
console.log(typeof newNum); // "number"
```

运行结果：

```
D:\learn\React\basicjs>node coercion_01.js
string
number
```

隐式类型转换 (Implicit Coercion)

这是 JavaScript 在执行特定操作时自动进行的转换。这些转换往往是“陷阱”的来源。

- 字符串与数字相加：如果加法操作符 (+) 的一个操作数是字符串，JavaScript 会将另一个操作数也转换为字符串，然后进行字符串拼接。

在 coercion_02.js 中：

```
console.log(10 + '5');    // "105" (数字 10 转换成字符串 "10")
console.log('5' + 10);    // "510" (数字 10 转换成字符串 "10")
```

运行结果：

```
D:\learn\React\basicjs>node coercion_02.js
105
510
```

- 非加法运算：在减法、乘法、除法等操作中，JavaScript 会将字符串转换为数字。

在 coercion_02.js 后面加上：

```
console.log('10' - 5);    // 5 (字符串 "10" 转换成数字 10)
console.log('10' * 5);    // 50 (字符串 "10" 转换成数字 10)
console.log('10' / 5);    // 2 (字符串 "10" 转换成数字 10)
```

运行结果：

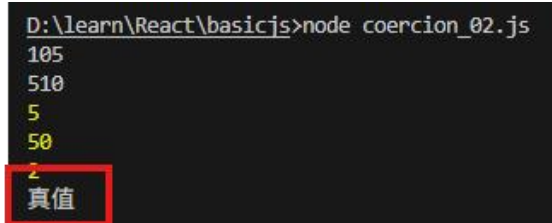
```
D:\learn\React\basicjs>node coercion_02.js
105
510
5
50
2
```

- 布尔值转换：在条件判断中，JavaScript 会将非布尔值转换为布尔值。
 - 假值 (Falsy values)：false, 0, '' (空字符串), null, undefined, NaN (非数字)。
 - 真值 (Truthy values)：除了上面的假值，其他所有值都是真值。

在 coercion_02.js 后面加上：

```
if ('hello') {  
  console.log('真值'); // 输出: 真值  
}  
  
if (0) {  
  console.log('假值'); // 不会执行  
}
```

运行结果：



```
D:\learn\React\basicjs>node coercion_02.js  
105  
510  
5  
50  
2  
真值
```

4.1.12 错误处理

错误处理是任何健壮应用程序不可或缺的一部分。在 JavaScript 中，try...catch 语句是处理运行时错误的主要机制，它可以帮助你优雅地捕获并响应程序执行过程中可能发生的异常，而不是让程序直接崩溃。

在程序运行过程中，可能会遇到各种意料之外的情况，比如：

- 网络请求失败：服务器无响应或返回错误。
- 无效的用户输入：用户输入的数据格式不正确。
- 代码逻辑错误：尝试访问未定义的变量或属性。
- 资源加载失败：图片或文件路径错误。

如果没有适当的错误处理，这些错误会导致程序中断，用户体验极差。错误处理就是通过特定的机制，识别、捕获并采取措施应对这些错误，从而保证程序的稳定性和用户友好性。

try...catch 语句

try...catch 是 JavaScript 中用于处理同步代码错误的标准方法。它由两个主要的代码块组成：

- **try 块:** 你认为可能会抛出错误（异常）的代码放在这里。如果 try 块中的代码执行时没有发生错误，那么 catch 块会被跳过。

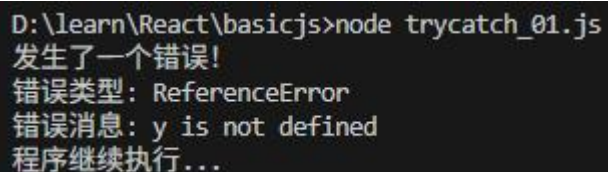
- **catch 块:** 如果 try 块中的任何代码抛出了错误，那么该错误会被 catch 块捕获。catch 块会接收到一个表示该错误的参数（通常命名为 error 或 err），你可以在这里对错误进行处理，比如：

- 向用户显示友好的错误消息。
- 记录错误信息以便调试。
- 尝试恢复程序状态。

举例说明，在 trycatch_01.js 中：

```
try {  
  // 这段代码可能会抛出错误  
  let x = y + 1; // y 未定义，会抛出 ReferenceError  
  console.log(x); // 这行代码不会被执行  
} catch (error) {  
  // 如果 try 块中发生错误，错误会被捕获到这里  
  console.error("发生了一个错误！");  
  console.error("错误类型:", error.name);    // 例如: ReferenceError  
  console.error("错误消息:", error.message); // 例如: y is not defined  
}  
  
console.log("程序继续执行..."); // 即使发生错误，程序也不会崩溃
```

运行结果：



```
D:\learn\React\basicjs>node trycatch_01.js  
发生了一个错误!  
错误类型: ReferenceError  
错误消息: y is not defined  
程序继续执行...
```

在 trycatch_02.js 中：

```
function divide(a, b) {  
  try {  
    if (b === 0) {  
      // 抛出一个自定义错误  
      throw new Error("除数不能为零！");  
    }  
  }  
}
```

```

        return a / b;
    } catch (error) {
        console.error("除法操作失败:", error.message);
        return NaN; // 返回一个特殊值表示操作失败
    }
}

console.log(divide(10, 2)); // 输出: 5
console.log(divide(10, 0)); // 输出: 除法操作失败: 除数不能为零! 然后是 NaN
console.log(divide(20, 4)); // 输出: 5

```

运行结果:

```

D:\learn\React\basicjs>node trycatch_02.js
5
除法操作失败: 除数不能为零!
NaN
5

```

`try...catch` 语句还可以包含一个可选的 `finally` 块。无论 `try` 块中的代码是否发生错误，`finally` 块中的代码总是会执行。它通常用于执行清理操作，比如关闭文件、释放资源等。

在 `trycatch_03.js` 中:

```

function readFile(filename) {
    try {
        console.log(`尝试读取文件: ${filename}`);
        // 模拟文件读取, 如果文件名不对就抛出错误
        if (filename !== "data.txt") {
            throw new Error("文件不存在或无权限访问!");
        }
        console.log("文件读取成功!");
        return "文件内容...";
    } catch (error) {
        console.error("读取文件时出错:", error.message);
        return null;
    } finally {
        // 无论是否出错, 都会执行这里的代码
        console.log("文件操作结束, 执行清理或关闭操作。");
    }
}

```

```

    }
}

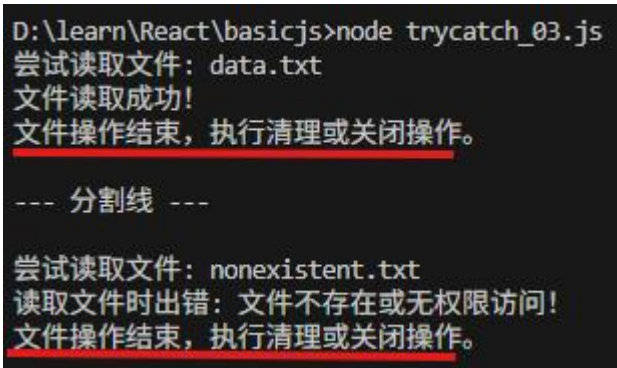
readFile("data.txt");
// 输出:
// 尝试读取文件: data.txt
// 文件读取成功!
// 文件操作结束, 执行清理或关闭操作。

console.log("\n--- 分割线 ---\n");

readFile("nonexistent.txt");
// 输出:
// 尝试读取文件: nonexistent.txt
// 读取文件时出错: 文件不存在或无权限访问!
// 文件操作结束, 执行清理或关闭操作。

```

运行结果:



```

D:\learn\React\basicjs>node trycatch_03.js
尝试读取文件: data.txt
文件读取成功!
文件操作结束, 执行清理或关闭操作。
--- 分割线 ---
尝试读取文件: nonexistent.txt
读取文件时出错: 文件不存在或无权限访问!
文件操作结束, 执行清理或关闭操作。

```

异步错误处理的注意事项

值得强调的是, `try...catch` 不能直接捕获异步操作 (如 `Promise`、`setTimeout` 中的错误)。对于异步操作, 你需要使用 `Promise` 的 `.catch()` 方法或 `async/await` 的 `try...catch` 来处理错误。

例如, 对于 `Promise`, 在 `trycatch_04.js` 中:

```

// Promise 错误处理
new Promise((resolve, reject) => {
  setTimeout(() => {
    reject(new Error("异步操作失败! ")); // 异步抛出错误
  }, 1000);
})

```

```
}).then(data => console.log(data))  
.catch(error => console.error("捕获到 Promise 错误:", error.message)); // Promise 的错误在这里捕获
```

运行结果：

```
D:\learn\React\basicjs>node trycatch_04.js  
捕获到 Promise 错误：异步操作失败！
```

对于 `async/await`，`try...catch` 则可以像处理同步代码一样处理异步错误：

在 `trycatch_04.js` 中加上：

```
async function fetchData() {  
  try {  
    const response = await fetch('https://invalid.url/data'); // 模拟一个会失败的请求  
    const data = await response.json();  
    console.log(data);  
  } catch (error) {  
    console.error("捕获到 Async/Await 错误:", error.message);  
  }  
}  
  
fetchData();
```

运行结果：

```
D:\learn\React\basicjs>node trycatch_04.js  
捕获到 Async/Await 错误：fetch failed  
捕获到 Promise 错误：异步操作失败！
```

通过学习 `try...catch`，你的程序将更具韧性，能够优雅地应对运行时可能出现的各种问题，从而提供更好的用户体验。

4.2 异步操作

我们先来理解一下什么是 异步操作（asynchronous operations）。我小时候学过数学家华罗庚的一篇文章，讲解时间管理。大致意思是，想象一下你正在厨房里做饭：

- **同步（Synchronous）**：你把水烧开，然后盯着壶，直到水沸腾，然后你再去做下一件事，比如切菜。在这个过程中，你不能同时做其他任何事情。这就是同步操作，

即一个任务必须完全完成后，下一个任务才能开始。

- **异步 (Asynchronous)**: 你把水放到炉子上烧，然后你告诉自己：“水烧开需要 10 分钟，我不用一直盯着它。这 10 分钟里，我可以去切菜、洗碗，或者准备其他的食材。” 当水烧开后，壶会发出响亮的哨声提醒你，你就可以回到烧水这个任务上，把它倒出来。这就是异步操作，即你发出一个任务（烧水），然后你可以去做其他事情，当任务完成时会给你一个“通知”。

在 JavaScript 中，许多操作都是异步的，比如：

- 从服务器获取数据 (fetch API)
- 读取文件
- 设置定时器 (setTimeout)
- 数据库操作

回调函数、Promise 和 async/await 就是解决这些异步任务的工具，它们让异步代码写起来更像同步代码，让代码变得更易读、更易于管理。

1) 回调函数

回调函数是 JavaScript 异步编程的基石，虽然现代代码更倾向于使用 Promise 和 async/await，但理解回调函数的工作原理对理解 JavaScript 的执行机制至关重要。

简单来说，**回调函数就是一个作为参数传递给另一个函数的函数**。在主函数执行完特定任务后，它会“回调”这个作为参数的函数。

创建一个简单的回调函数的例子 callback_01.js: processUser 函数接受两个参数：一个用户信息和一个**回调函数 callback**。当 processUser 完成它的内部逻辑后，它会调用传入的回调函数。

```
function processUser(username, callback) {  
  console.log('正在处理用户: ${username}');  
  
  // 模拟一个耗时的操作，比如数据处理或数据库查询  
  setTimeout(() => {
```

```

const data = {
  id: Math.random().toString(36).substring(7),
  username: username
};

// 操作完成后，调用回调函数，并把结果作为参数传给它
callback(data);
}, 1000);
}

// 定义一个回调函数，它接收处理后的数据
function displayUser(userData) {
  console.log("用户数据处理完成！");
  console.log("-----");
  console.log(`ID: ${userData.id}`);
  console.log(`用户名: ${userData.username}`);
}

// 调用主函数，并把 `displayUser` 作为回调函数传进去
console.log("开始处理...");
processUser("张三", displayUser);
console.log("处理函数已启动，代码将继续执行...");

```

运行结果：（注意中间的延迟过程）

```

D:\learn\React\Async>node callback_01.js
开始处理...
正在处理用户：张三
处理函数已启动，代码将继续执行...
用户数据处理完成！
-----
ID: sjuth4
用户名：张三

```

运行效果：

1. “开始处理...”
2. “正在处理用户：张三”
3. “处理函数已启动，代码将继续执行...”
4. （等待 1 秒后）
5. “用户数据处理完成！”
6. “-----”
7. “ID: ...”
8. “用户名：张三”

这个例子清楚地说明了**异步的非阻塞特性**：processUser 启动后，主程序会立即继续执行（**第 3 步**），而不会等待它完成。

在实际应用中，异步操作可能会失败。因此，一个健壮的回调函数模式通常会把错误作为回调函数的第一个参数。如果操作成功，第一个参数就是 null。

实例 callback_02.js:

```
function fetchData(url, callback) {
  console.log(`正在从 ${url} 获取数据...`);

  // 模拟网络请求
  setTimeout(() => {
    if (url === "http://api.example.com/data") {
      // 成功的情况
      const data = { message: "数据获取成功！" };
      callback(null, data); // 第一个参数为 null 表示没有错误
    } else {
      // 失败的情况
      const error = new Error("URL 不正确，获取失败。");
      callback(error, null); // 第一个参数为 Error 对象
    }
  }, 1500);
}

// 成功的回调处理
fetchData("http://api.example.com/data", (error, data) => {
  if (error) {
    console.error("发生错误:", error.message);
  } else {
    console.log("成功:", data.message);
  }
});

// 失败的回调处理
fetchData("http://api.example.com/invalid-url", (error, data) => {
  if (error) {
    console.error("发生错误:", error.message);
  } else {
    console.log("成功:", data.message);
  }
});
```



```
});
```

运行结果：

```
D:\learn\React\Async>node callback_02.js
正在从 http://api.example.com/data 获取数据...
正在从 http://api.example.com/invalid-url 获取数据...
成功：数据获取成功！
发生错误：URL 不正确，获取失败。
```

请注意执行的顺序以及中间的延迟。

这个例子是理解 **Promise** 的关键前置知识。 **Promise** 的 `.then()` 和 `.catch()` 实际上就是对这种回调模式的封装，使得异步代码链式调用时更易于管理，避免了所谓的“回调地狱”（**callback hell**）。

2) Promise（承诺）

Promise 的字面意思就是“承诺”，它代表了一个异步操作的最终完成（或失败）及其结果值。

你可以把 **Promise** 想象成一个餐厅的叫号器。

- 你去餐厅点餐，服务员给你一个叫号器（**Promise**）。
- 这个叫号器有三种状态：
 - ① 待定（**pending**）：你点完餐，拿着叫号器等待。这个任务正在进行中，结果还不确定。
 - ② 已兑现/已完成（**fulfilled**）：叫号器响了，你的餐做好了。任务成功完成，你拿到了结果（食物）。
 - ③ 已拒绝/已失败（**rejected**）：等了很久，服务员过来抱歉地说你点的菜卖完了。任务失败了，你得到一个失败的原因（“菜品售罄”）。

一个 **Promise** 一旦从 **pending** 变为 **fulfilled** 或 **rejected**，它的状态就不会再改变。

一个 **Promise** 对象有 `.then()` 和 `.catch()` 两个常用的方法来处理成功和失败的结果。如果承诺兑现了，`then` 下一步做什么；如果失败/被拒绝了，`catch` 失败原因然后根据原因做下一步。

我们创建一个文件 `promise_01.js`:

```
// 示例：模拟一个从服务器获取数据的 Promise
function fetchData() {
  return new Promise((resolve, reject) => {
    // 模拟网络请求需要 3 秒
    setTimeout(() => {
      const isSuccess = true; // 假设请求成功了

      if (isSuccess) {
        // 请求成功，调用 resolve() 并传递数据
        resolve('这是从服务器获取的数据！');
      } else {
        // 请求失败，调用 reject() 并传递错误信息
        reject('网络连接失败');
      }
    }, 3000);
  });
}

// 使用 Promise
console.log('开始请求数据...');
fetchData()
  .then((data) => {
    // .then() 处理成功的 Promise，接收 resolve() 传递的数据
    console.log('数据获取成功！');
    console.log('获取到的数据是:', data);
  })
  .catch((error) => {
    // .catch() 处理失败的 Promise，接收 reject() 传递的错误信息
    console.log('数据获取失败！');
    console.error('错误信息:', error);
  });
```

运行结果：

```
D:\learn\React\Asynch>node promise_01.js
开始请求数据...
数据获取成功！
获取到的数据是：这是从服务器获取的数据！
```

如果我们把 `isSuccess` 的初始值设为 `false`，运行结果：

```
D:\learn\React\Asynch>node promise_01.js
开始请求数据...
数据获取失败!
错误信息: 网络连接失败
```

Promise 的另一个强大之处在于可以进行链式调用，让多个异步任务按顺序执行。

想象你正在制作一份三明治：

1. 获取面包（任务 1）
2. 切面包（任务 2）
3. 在面包上涂酱（任务 3）

这些步骤必须按顺序完成。你可以用 Promise 链来模拟这个过程（promise_02.js）：

```
function getBread() {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log('1. 面包已拿到');
      resolve('新鲜的面包');
    }, 1000);
  });
}

function sliceBread(bread) {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log(`2. 正在切 ${bread}`);
      resolve('切好的面包片');
    }, 1000);
  });
}

function spreadJam(slicedBread) {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log(`3. 正在 ${slicedBread} 上涂果酱`);
      resolve('美味的三明治完成了！');
    }, 1000);
  });
}

// 链式调用
```

```
console.log('开始制作三明治...');
getBread()
  .then(bread => sliceBread(bread)) // 任务 1 完成后，将结果传递给任务 2
  .then(slicedBread => spreadJam(slicedBread)) //任务 2 完成后，将结果传递给任务 3
  .then(finalSandwich => {
    console.log(finalSandwich); // 最终的结果
  })
  .catch(error => {
    console.error('制作过程中出错:', error);
  });
```

这种方式可以清晰地表达任务的执行顺序，避免了 回调地狱（callback hell），即多层嵌套的回调函数。在 Promise 出现之前，异步操作主要依赖回调函数的方式，也就是说我执行完一个任务，然后根据设定好的回调函数，确定下一步做什么。Promise 大大改善了异步操作，但是还不是最优的方法。

运行结果：

```
D:\learn\React\Asynch>node promise_02.js
开始制作三明治...
1. 面包已拿到
2. 正在切 新鲜的面包
3. 正在 切好的面包片 上涂果酱
美味的三明治完成了！
```

3) async/await

async/await 是在 ES2017 中引入的，它是在 Promise 的基础上，提供了一种更简洁、更直观的语法来处理异步操作。它的核心思想是：让异步代码看起来和写起来都像同步代码一样。

- **async 关键字**：用于声明一个函数是异步的。这个函数会默认返回一个 Promise 对象。
- **await 关键字**：只能在 async 函数内部使用。它会“暂停”函数的执行，等待 Promise 得到解决（fulfilled），然后继续执行后续代码。如果 Promise 被拒绝（rejected），await 会抛出一个错误。

我们仍然用制作三明治的例子，这次使用 `async/await` (`async_01.js`)：

```
// 假设 getBread, sliceBread, spreadJam 函数都如上所示，返回 Promise
function getBread() {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log('1. 面包已拿到');
      resolve('新鲜的面包');
    }, 1000);
  });
}

function sliceBread(bread) {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log(`2. 正在切 ${bread}`);
      resolve('切好的面包片');
    }, 1000);
  });
}

function spreadJam(slicedBread) {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log(`3. 正在 ${slicedBread} 上涂果酱`);
      resolve('美味的三明治完成了！');
    }, 1000);
  });
}

async function makeSandwich() {
  try {
    console.log('开始制作三明治...');
    // await 关键字会“等待” getBread() 的 Promise 完成，然后将结果赋值给
    bread 变量
    const bread = await getBread();

    // 等待 sliceBread() 的 Promise 完成
    const slicedBread = await sliceBread(bread);

    // 等待 spreadJam() 的 Promise 完成
    const finalSandwich = await spreadJam(slicedBread);

    console.log(finalSandwich);
  } catch (error) {
```

```
    // try-catch 块用来捕获 await 抛出的错误
    console.error('制作过程中出错:', error);
  }
}

makeSandwich();
```

是不是感觉 `async/await` 的代码更直观，更易于阅读？它就像你在写同步代码，一步一步地执行，但实际上，每个 `await` 都在等待一个异步任务的完成。

运行结果：

```
D:\learn\React\Asynch>node async_01.js
开始制作三明治...
1. 面包已拿到
2. 正在切 新鲜的面包
3. 正在 切好的面包片 上涂果酱
美味的三明治完成了！
```

既然 `async/await` 如此方便，为什么 JavaScript 还要保留 Promise？这就像问：“有了汽车，为什么还需要保留自行车？”

答案是：`async/await` 是在 Promise 的基础上构建的，它只是 Promise 的一个语法糖。你可以把它们理解为两种不同的工具，用来解决同一个问题，但各有其适用场景和独特作用。

我们通过一个比喻来解释：

- Promise 是一个“异步任务的底层蓝图”。它定义了异步操作的三个基本状态（待定、完成、失败）和如何处理这些状态的方法（`.then()` 和 `.catch()`）。所有的异步操作，无论是网络请求还是定时器，都可以被封装成一个 Promise 对象。Promise 就像是汽车的发动机和变速箱，它们是汽车能够行驶的底层核心。
- `async/await` 是一个“让异步代码看起来更像同步代码的驾驶舱”。它提供了一个更直观、更线性的语法来操作这些异步任务。`async` 和 `await` 就像是汽车的方向盘和油门。有了它们，你不需要去理解发动机的内部工作原理，就能轻松地驾驶汽车。

保留 Promise 的必要性

以下是为什么 Promise 仍然至关重要的几个原因：

① 它是所有异步操作的基石

几乎所有现代的异步 API（如 `fetch()`、`setTimeout` 的 `Promise` 版本、数据库操作等）都默认返回一个 `Promise` 对象。如果你需要自己封装一个异步任务，比如读取文件或与第三方 API 交互，你必须使用 `new Promise()` 来创建它。`async/await` 无法自己创建异步任务，它只能等待那些已经返回 `Promise` 的任务。

举例：

- 你要编写一个函数来模拟网络请求，你必须使用 `Promise`：

```
function fetchData() {  
  return new Promise((resolve) => {  
    setTimeout(() => resolve('数据'), 2000);  
  });  
}
```

- 这里的 `fetchData` 函数就是异步操作的蓝图，而 `async/await` 则是用来“使用”这个蓝图的：

```
async function getDataAndUse() {  
  const data = await fetchData(); // await 正在等待 fetchData 返回的 Promise  
  console.log(data);  
}
```

② 处理并行任务

当你有多个互不依赖的异步任务需要同时执行时，`Promise.all()` 是一个非常有用的工具。`async/await` 无法直接并行处理这些任务，但它可以与 `Promise.all()` 结合使用。

举例：

- 你需要同时获取用户数据和产品数据，但它们之间没有依赖关系。
- 如果使用 `await`，它们会按顺序执行，浪费时间：

```
// 顺序执行，总耗时约 4 秒  
const user = await getUserData();  
const products = await getProductData();
```

- 如果使用 `Promise.all()`，它们会并行执行，总耗时只取决于最慢的那个任务（约 2 秒）：

```
// 并行执行，总耗时约 2 秒
const [user, products] = await Promise.all([
  getUserData(),
  getProductData()
]);
```

这种强大的并行处理能力是 Promise 独有的特性，async/await 需要借助它才能实现。

③ 更好的控制流

Promise 提供了更丰富的控制流方法，如 **Promise.race()**（在多个任务中，哪个先完成就用哪个结果）和 **Promise.allSettled()**（等待所有 Promise 都完成，无论成功或失败）。这些方法在复杂的异步场景中非常有用。

举例：

- **Promise.race()** 可以用于设置超时机制，哪个先完成就取哪个结果：

```
const result = await Promise.race([
  fetchData(),
  // 真正的请求
  new Promise((_, reject) => setTimeout(() => reject(new Error('请求超时')), 3000
))
]);
```

async/await 自身没有这样的能力，它必须依赖 Promise 的这些静态方法。

结论

async/await 并没有取代 Promise，而是依赖于 Promise。

你可以把 Promise 看作是异步编程的“基础语言”，而 async/await 则是这个语言的“高级方言”。学会 async/await 可以让你更高效地编写和阅读异步代码，但理解 Promise 的底层原理，以及它提供的那些强大方法（如 **Promise.all()**），才能让你真正掌握异步编程的精髓，并在面对复杂场景时游刃有余。

所以，两者是相辅相成的关系。掌握了 Promise 的“蓝图”，再使用 async/await 的“驾驶舱”，你就能在异步编程的世界里畅行无阻。

五、实验注意事项

1. 注意教师的操作演示。

2. 学生机与教师机内网连通，能接收和提交实验结果。
3. 按实验要求输入测试数据，并查看输出结果是否符合实验结果。

六、思考题