JS基础

Status	In progress
Summary	JS基本数据类型和输入输出
Project	JS
Tags	

JS基本数据类型

数据类型	类型说明	示例	内存占用
Undefined	表示变量未被初始化	`let a;`	通常是8字节
Null	表示变量的值为空	`let a = null;`	通常是8字节
Boolean	表示逻辑实体	`let a = true;`	通常是4字节
Number	表示双精度64位浮点数	`let a = 100;`	通常是8字节
BigInt	表示大于2^53 - 1的整数。	`let a = 12345678901234567890123456789012345678901234567890n;`	根据数值大小动态变化，可以占用多个字节
String	表示一系列字符	`let a = "hello";`	每个字符通常占用2字节（UTF-16编码）
Symbol	表示独一无二的值	`let a = Symbol('id');`	通常是8字节，但具体取决于实

			现

NaN和null

在JavaScript中，`NaN` 和 `null` 是两个特殊的值，它们分别代表了不同的概念和用途。理解这两个值的特性和行为对于编写健壮的JavaScript代码非常重要。

NaN

`NaN` 是一个代表“Not-A-Number”（非数字）的特殊值。它是一个在JavaScript中用来表示数学运算失败或错误结果的值。`NaN` 是 `Number` 对象的一个属性。以下是几个产生 `NaN` 的情况：

1. **无效的数学运算**：如 `0/0` 或 `Math.sqrt(-1)`。
2. **类型错误的运算**：如 `"abc" * 3`。
3. **函数返回失败**：如 `parseFloat("xyz")`。

`NaN` 有几个独特的特性：

- `NaN` 不等于任何值，包括它自己。即 `NaN !== NaN` 返回 `true`。
- 检测一个值是否为 `NaN`，应使用 `Number.isNaN()` 函数，而不是等号运算。

null

`null` 在JavaScript中代表“无”或“空值”。它是一个原始值，通常用来表示变量中故意存放的空或不存在的值。`null` 是一个字面量（不像 `undefined`，它是一个类型未定义的变量的默认值）。

`null` 的常见用途包括：

- 初始化一个预期会被后来赋予对象的变量。
- 明确一个对象目前没有有效的值。

`null` 的特性包括：

- `null` 是一个假值（falsy value），但它是一个对象类型（这是历史遗留的一个错误）。即 `typeof null` 返回 `"object"`。
- `null` 通常用于与 `undefined` 比较，两者在非严格比较 (`==`) 下相等，即 `null == undefined` 返回 `true`，但在严格比较 (`===`) 下不等，即 `null === undefined` 返回 `false`。

区别和联系

尽管 `NaN` 和 `null` 都表示某种“空”或“无效”的状态，它们的用途和含义有明显的区别：

- `NaN` 表示一个错误的数值结果。
- `null` 表示一个故意的空值。

在实际编程中，合理使用这两个值可以帮助你更准确地描述代码的意图，处理错误，以及管理数据的存在状态。

代码分析

```
/***
 * JS的基本数据类型
 */

let udf = undefined
let nll = null
let nan = NaN

let a = 24
let b = 8.8
let c = 'Leon'
let d = true

const add = (x, y) => {
  return x + y
}

const sub = (x, y) => x - y

console.log(`Number add undefined: ${add(udf, a)}`)  // NaN
console.log(`Number add null: ${add(nll, a)}`)       // 24
console.log(`Number add nan: ${add(nan, a)}`)        // NaN
console.log(`Number add boolean: ${add(b, d)}`)        // 9.8
console.log(`Number sub boolean: ${sub(b, d)}`)        // 7.8000000000

console.log(`String add Number: ${add(c, a)}`)       // Leon24
console.log(`String sub undefined: ${sub(c, a)}`)    // NaN
console.log(`String sub null: ${sub(c, nll)}`)       // NaN
console.log(`String add boolean: ${sub(c, d)}`)       // NaN

// 输出如下
/***
Number add undefined: NaN
Number add null: 24
```

```
Number add nan: NaN
Number add boolean: 9.8
Number sub boolean: 7.800000000000001
String add Number: Leon24
String sub undefined: NaN
String sub null: NaN
String add boolean: NaN
*/
```

这段JavaScript代码展示了如何使用基本数据类型进行不同的运算，并展示了JavaScript在处理类型转换时的行为。下面是对代码中的每个输出语句的分析，以及为什么会产生这样的结果。

分析

1. **Number add undefined: NaN**
   - 使用 `undefined` 和数字进行加法运算，结果为 `NaN`。这是因为 `undefined` 在数学运算中转换为 `NaN`，任何与 `NaN` 的运算结果都是 `NaN`。
2. **Number add null: 24**
   - 使用 `null` 和数字进行加法运算，结果为数字本身。这是因为 `null` 在数学运算中被转换为 `0`，因此 `0 + 24` 等于 `24`。
3. **Number add nan: NaN**
   - 使用 `NaN` 和数字进行加法运算，结果为 `NaN`。`NaN` 是一个特殊的数值，表示“不是一个数字”，任何与 `NaN` 的运算都会返回 `NaN`。
4. **Number add boolean: 9.8**
   - 使用数字和布尔值进行加法运算。布尔值 `true` 转换为 `1`，所以 `8.8 + 1` 等于 `9.8`。
5. **Number sub boolean: 7.800000000000001**
   - 使用数字和布尔值进行减法运算。同样，布尔值 `true` 转换为 `1`，因此 `8.8 - 1` 应该等于 `7.8`。但由于浮点数精度问题，结果显示为 `7.800000000000001`。
6. **String add Number: Leon24**
   - 字符串和数字进行加法运算，数字会转换为字符串，然后进行字符串连接。因此，`'Leon' + 24` 转换为 `'Leon24'`。
7. **String sub undefined: NaN**
   - 字符串和 `undefined` 进行减法运算，由于 `undefined` 转换为 `NaN`，任何与 `NaN` 的运算都是 `NaN`。

8. **String sub null: NaN**
    - 字符串和 `null` 进行减法运算。尽管 `null` 可以转换为 `0`，字符串到数字的转换失败（因为 `'Leon'` 不能转换为一个有效的数字），结果是 `NaN`。
9. **String add boolean: NaN**
    - 字符串和布尔值进行减法运算，布尔值 `true` 转换为 `1`，但由于字符串到数字的转换失败，结果是 `NaN`。

结论

这段代码有效地展示了JavaScript中的隐式类型转换和数据类型之间的交互。特别是 `undefined`, `null`, 和 `NaN` 在数学运算中的行为，以及数字与字符串或布尔值的交互。此外，代码也揭示了浮点数精度可能带来的问题。这些示例对于理解和预测JavaScript中变量和表达式的行为非常有帮助。

输入输出

这份代码使用了三种格式化输出，详细见注释。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="js demo" content="width=device-width, initial-scale=1.0
  <title>JS Demo</title>
</head>
<body>
  <div id="content"></div> <!-- 容器元素 -->
  <script type="module" src="src/data_and_io.js"></script>
</body>
</html>
```

```
/* data_and_io.js */
/* JS basic variable and io*/
import { sprintf} from './sprintf.js'

/***
 * 格式化输出
 */
```

```
let msg = "Liovale"


/***
 * 1. 使用字符串模板（Template Literals）
 * JavaScript ES6 引入了模板字符串，这是一种包含嵌入表达式的字符串字面量。你可以
 */
console.log(`Welcome to js, ${msg}`) // 注意

/***
 * 2. 使用 console.log 的高级特性
 * console.log 可以接受多个参数，并且可以使用类似于 C 语言中 printf 的格式化选
 */
console.log("Hello %s!", msg)

/***
 * 3. 对于更复杂的格式化需求，可以使用第三方库，如 sprintf-js，它提供了类似于 C
 * 首先，需要安装这个库：
 * nvm install sprintf-js
 */
// const sprintf = require('sprintf-js').sprintf // 这种方式是运行在node
// let info = sprintf("Thinking and coding, %s.", msg)
// console.log(info)


// 操作 DOM

// 这种方法过时了，不好维护
// document.write(`<h1>${msg}</h1>`)

// innerHTML
document.getElementById('content').innerHTML = `<h1>innerHTML ${msg}<


// appendChild
let contentDiv = document.getElementById('content')
let newElement = document.createElement('h1')
/***
```

```
 * 我这里使用的sprintf库过于古老，不想修改原始库，并且需要保持 ES6 模块的结构，可
 */
newElement.textContent = sprintf("appendChild %s", msg)
contentDiv.appendChild(newElement);

alert(`Welcome ${msg}!`)
```

模块化老JS库

上一节的示例代码使用了第三方提供的 `sprintf.js` 代码来看，这个库确实定义了 `sprintf` 和 `vsprintf` 函数，但并没有使用 ECMAScript 6 (ES6) 模块的 `export` 关键字进行导出（查看源代码发现的）。相反，它使用了一种较老的方法来兼容不同的 JavaScript 环境，包括 Node.js 和浏览器。

解决方案

要在使用 ES6 模块的环境中导入这个库，您可以采取以下几种方法之一：

1. 修改 `sprintf.js` 以添加 ES6 导出

您可以在文件的末尾添加 ES6 模块导出语句。这需要您能够修改 `sprintf.js` 文件。在文件的最后添加如下代码：

```
export { sprintf, vsprintf };
```

这样修改后，您就可以使用您最初的导入语句了：

```
import { sprintf } from '../lib/sprintf-js/src/sprintf.js';
```

2. 使用非模块脚本

如果您不希望修改原始文件，另一种方法是在 HTML 文件中通过 `<script>` 标签直接引入 `sprintf.js`，然后在全局作用域中使用 `sprintf` 和 `vsprintf` 函数。这适用于不支持模块化的传统网页项目。

```
<script src="../lib/sprintf-js/src/sprintf.js"></script>
<script type="module">
  // 您的其他模块化 JavaScript 代码
  console.log(sprintf("%02d", 5)); // 使用全局 sprintf 函数
</script>
```

3. 创建一个包装模块

如果您不想修改原始库，并且需要保持 ES6 模块的结构，您可以创建一个新的 JavaScript 文件来包装 `sprintf.js` 的功能，并导出所需的函数。例如，您可以创建一个名为 `sprintf-module.js` 的文件，内容如下：

```
import '../lib/sprintf-js/src/sprintf.js';

export const sprintf = window.sprintf;
export const vsprintf = window.vsprintf;
```

然后，您可以通过这个新模块来导入 `sprintf` 和 `vsprintf`：

```
import { sprintf, vsprintf } from './sprintf-module.js';
```

结论

根据您的项目需求和环境，选择最适合的方法。如果您正在构建一个现代的、模块化的前端应用，推荐修改 `sprintf.js` 来直接支持 ES6 模块导出，或者创建一个包装模块。如果是一个传统的网页项目，使用 `<script>` 标签可能是最简单的解决方案。

window对象

在上一节提到的包装模块中，使用 `window` 对象是为了访问由 `sprintf.js` 脚本定义并附加到全局 `window` 对象上的 `sprintf` 和 `vsprintf` 函数。这种方法通常用于浏览器环境，因为在浏览器中，全局作用域是通过 `window` 对象访问的。

解释

1. **引入脚本**：当您使用 `import '../lib/sprintf-js/src/sprintf.js';` 这行代码时，假设 `sprintf.js` 是一个传统的脚本，它可能不会导出任何模块，而是直接将一些变量或函数定义在全局作用域中。在浏览器中，这意味着它们被添加到了 `window` 对象。
2. **使用 `window` 对象**：通过 `window.sprintf` 和 `window.vsprintf`，您可以访问那些被 `sprintf.js` 添加到全局作用域的函数。这样，即使 `sprintf.js` 本身不是一个模块，您也可以在 ES6 模块环境中像使用模块一样使用这些函数。
3. **导出为模块**：通过将这些全局函数赋值给常量并导出它们，您创建了一个可以在其他 ES6 模块中使用的接口。这样，其他文件可以通过标准的 `import` 语句来导入 `sprintf` 和 `vsprintf`。

注意事项

- **环境限制**：这种方法假设您的代码运行在一个浏览器环境中，因为 `window` 对象是浏览器的全局对象。如果您在 Node.js 或其他非浏览器环境中运行代码，`window` 对象是不存在的。在

Node.js 中，全局对象是 `global`，但直接在模块间共享全局变量并不是一个好的实践。

- **模块化和封装**：虽然这种方法可以工作，但它违背了模块化和封装的原则。最好的做法是修改原始的 `sprintf.js`，使其成为一个真正的模块，直接导出所需的功能。

替代方法

如果您在 Node.js 或希望有更好封装的环境中工作，最好修改 `sprintf.js` 使其使用模块导出，或者如果您不能修改原始文件，您可以考虑使用构建工具（如 Webpack 或 Rollup）来包装这些全局依赖，使其可以更好地集成到现代 JavaScript 项目中。

在Web开发中，`window` 对象是一个非常重要的概念，它代表了浏览器窗口，并且作为全局对象，为运行在浏览器中的 JavaScript 提供了一个执行环境。`window` 对象拥有一系列属性和方法，可以控制浏览器的行为以及与浏览器窗口进行交互。下面是一些关于 `window` 对象的关键点：

1. 全局作用域

在浏览器中，`window` 对象同时也是全局作用域，这意味着在全局作用域中声明的所有变量、函数和对象都成为 `window` 对象的属性。例如：

```
var x = 1;
console.log(window.x); // 输出 1
```

2. 提供全局函数和属性

`window` 对象提供了许多内置的方法和属性，例如 `alert()`, `confirm()` 和 `prompt()` 这些用户交互函数，以及 `setTimeout()`, `setInterval()` 这些定时器函数。

3. 浏览器窗口控制

`window` 对象允许开发者控制浏览器窗口的行为，包括打开新窗口 (`window.open()`)，调整窗口大小 (`window.resizeTo()`) 等。

4. 访问浏览器的历史、位置和导航

`window` 对象还包含了其他重要的对象，如 `window.document` (代表加载在窗口中的文档的文档对象模型（DOM）)、`window.location` (提供当前URL的信息和方法来改变URL) 和 `window.history` (允许操作浏览器的会话历史)。

5. 事件处理

`window` 对象也是事件处理的一个重要对象，可以在这个对象上监听和处理诸如 `load`, `resize`, `scroll` 等事件。

6. 存储功能

`window` 提供了 `localStorage` 和 `sessionStorage` 对象，允许Web应用在用户的浏览器中存储数据。

7. 安全性

由于 `window` 对象包含了大量的功能和访问浏览器核心功能的能力，它也涉及到一些安全性问题，尤其是跨域访问和脚本之间的交互。

使用场景

- **开发者通常会利用 `window` 对象来进行全局变量的声明和使用**，尽管这种做法在现代JavaScript开发中越来越被局部作用域和模块化所取代。
- **在进行页面间通信或数据存储时，`window` 的属性和方法也非常有用**。

总结来说，`window` 对象是JavaScript在浏览器中的全局环境的核心，它不仅仅是全局作用域的代表，还是许多浏览器功能和API的访问入口。

JS定义函数

在JavaScript中，定义函数有几种不同的方法，每种方法都有其特定的用途和适用场景。下面是几种常见的方式来定义函数：

1. 函数声明（Function Declarations）

函数声明是最常见的定义函数的方式。它以关键字 `function` 开头，后跟函数名和函数体。

```
function sayHello() {
    console.log("Hello, world!");
}
sayHello(); // 调用函数输出 "Hello, world!"
```

函数声明的一个特点是**函数提升**（function hoisting），意味着你可以在声明函数之前调用它：

```
sayHello(); // 调用函数输出 "Hello, world!"
function sayHello() {
    console.log("Hello, world!");
}
```

2. 函数表达式（Function Expressions）

函数表达式允许你将函数赋值给变量。函数可以是匿名的，也可以是命名的。

```
const greet = function() {
    console.log("Hello, world!");
};
greet(); // 调用函数输出 "Hello, world!"
```

函数表达式不会提升，所以你必须在定义函数之后才能调用它。

3. 箭头函数（Arrow Functions）

箭头函数是ES6中引入的，提供了一种更简洁的方式来写函数。它们特别适合用于匿名函数表达式。

```
const greet = () => {
    console.log("Hello, world!");
};
greet(); // 调用函数输出 "Hello, world!"
```

箭头函数还有其他的语法简化，例如当只有一个表达式时可以省略花括号，并且自动返回表达式的结果：

```
const add = (a, b) => a + b;
console.log(add(5, 3)); // 输出 8
```

箭头函数还有不绑定自己的 `this` 的特性，使得它们非常适合用于需要维护外部 `this` 上下文的情况（如回调函数和方法）。

4. Generator 函数

Generator 函数是ES6引入的一种特殊类型的函数，它可以暂停执行并在稍后继续，非常适合用于执行异步操作。

```
function* generateSequence() {
    yield 1;
    yield 2;
    return 3;
}

const generator = generateSequence();
console.log(generator.next()); // { value: 1, done: false }
console.log(generator.next()); // { value: 2, done: false }
console.log(generator.next()); // { value: 3, done: true }
```

5. IIFE（立即调用的函数表达式）

IIFE是一种在定义后立即执行的函数，通常用于创建一个独立的作用域。

```
(function() {
    console.log("Hello, world!");
})();
```

这种方式在早期JavaScript中被广泛用于模拟块级作用域和避免污染全局命名空间。