Javascript

> - Javascript是一门**解释型**语言，无需编译，而是在程序运行时进行解释。解释型语言的运行速度通常不如编译型，而JS通过JIT（即时编译技术）提高了运行的速度，是一种比较快的解释型语言。JS通常作为网页开发的脚本与HTML、CSS配合使用，使网页具有各种各样的功能。
> - 作为一种脚本语言（或称为“动态语言”），JS的数据类型直到运行时才能确定。同时，JS弱化了类型，即在使用前不需要像C/C++那样进行**类型**声明或内存分配，在语法上也更加轻量化，大大减轻了开发人员的工作量，使他们可以把工作重心放在功能设计上。
> - JS的简便好用同时也会带来一些弊端。比如由于缺乏类型检查，在面对比较复杂的工程时，可能无法检查出错误，导致bug越积越多。

1. 基本语法

1.1 数据类型

> - 值：
>   数字(Number)、字符串(String)、布尔(Boolean)、空(Null)、未定义 (Undefined)、Symbol
> - 引用：
>   对象(Object)、数组(Array)、函数(Function)，以及特殊对象——正则(RegExp)、日期(Date)

数字

只有一种数字类型，即Number，没有int、float、double等。

> "NaN"是一种属性，用来指示“不是数字类型”。可调用 `isNaN()` 函数来判断是否为数字类型。

字符串

JS支持模版字符串——用反引号()括住字符串，并用占位符 `${}` 在字符串中插入表达式：

```
let number = 2; // 常量赋值
console.log(`There are ${number} books on the shelf`);
```

字符串转数字： `parseInt()` 、 `parseFloat()`

区分空和未定义

null本身就是一种值，可以把null赋给变量。而undefine代表该变量在声明后未被赋值。

对象

对象是属性和方法的容器，是一个比较广泛的概念，数组、函数等其实都可以作为对象，现在只关注其比较简单的含义：

```
// 定义一个对象
var student = {
    name: "Alice",
    id:"1234",
    print: () => {
        console.log(student.name);
        console.log(student.id);
    }
}
// 调用该对象的方法
student.print()
```

1.2 变量

类型不需要声明，但变量本身还是需要声明的：

```
let num = 10; // 声明一个块作用域的局部变量，并初始化（可选）
var num = 10; // 声明一个全局作用域变量，并初始化（可选）
const num = 10; // 声明一个块作用域的只读常量，并初始化（必须）
```

全局变量一般不会随便使用，声明变量时还是以let为主。

1.3 函数

函数的声明：function关键字、函数名、形参（如有）、返回值（如有）

```
function average(num1, num2) {
    return (num1 + num2) / 2
}
```

1.4 类

与C++类似，在JS中也可以声明一个类(class)，并通过类来实例化对象：

```
class Student {
    constructor(name,id) {
        this.name = name;
        this.id = id;
    }
    print() {
        console.log(this.name);
        console.log(this.id);
    }
}
var student1 = new Student("Alice",1234);
student1.print();
```

JS同样支持类的继承。

1.5 控制结构

条件判断

```
if (a == b) {
    console.log("equal");
}
else {
    console.log("unequal");
}
```

循环

也有while、do……while、switch，for循环除C/C++那种写法外，还有：

```
for (i in array) {

}
```

1.6 异常处理

```
try { // 检测异常
    if (isNaN(a) || isNaN(b)) {
        throw "not number"; // 抛出异常
    }
    else {
        if (b != 0) {
            return a/b;
        }
        else {
            throw "math error";
        }
    }
} catch(err) { // 接收异常，只有发生异常才会执行
    console.log(err);
} finally { // 无论有没有异常都会执行
    console.log("done");
}
```

1.7 模块

JS中可以导入导出模块。

- 导入：import

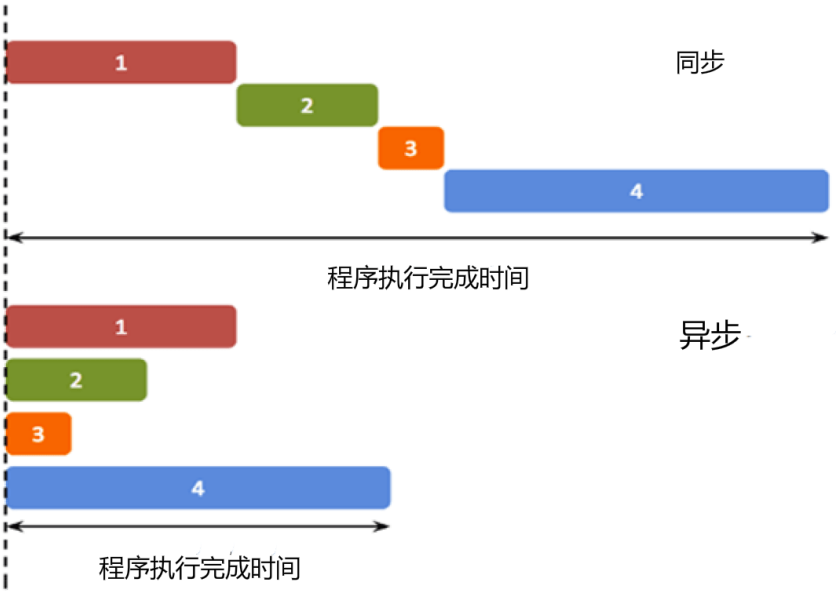
```
import {a,b,c} from " ";
```

- 导出：export/export default

```
// export
export class Student {
    constructor(name,id) {
        this.name = name;
        this.id = id;
    }
}
import {Student} from " "; // 花括号

// default export
export default class Student {
    constructor(name,id) {
        this.name = name;
        this.id = id;
    }
}
import Student from " ";
```

1.8 异步操作



为什么需要异步?

我们在网页中常见的按钮、复选框、输入窗口等交互功能都需要等待用户的反应，再执行下一步。如果把对应的操作写在主线程中，同步执行，那么在用户反应之前，网站的其它功能都无法执行，这肯定是不可以的。当我们使用异步操作，使这些事件在子线程里监听用户的反应，即使发生阻塞，也不会影响网站其它功能的执行。

1.8.1 回调函数

把这些事件挪到子线程中也是有代价的，比如我们无法再将其合并回主线程，无法得知它是否完成了。这个时候就需要一个回调函数，在异步事件完成时自动调用，返回事件的结果。
JS中的一些异步函数自带回调函数这一参数，比如 `setTimeout()`

```
// 1、两秒后打印"finish"
setTimeout(() => {
    console.log("finish")
}, 2000);
// 2、打印"start"
let sum=0;
console.log("start");
for(i=1;i<=10;i++) {
    sum+=i;
}
// 3、打印求和结果
console.log(sum);
```

如果 `setTimeout()` 不是异步执行的话，2、3都需要等1完成后才能执行。而异步执行后的结果为：

```
start
55
finish
```

也可以自己定义回调函数这一功能

```
function main(value, cb) {
    console.log("start");
    cb(value);
}
main(10,(value) => {
    let sum=0;
    for(i=1;i<=value;i++) {
        sum+=i;
    }
    console.log(sum);
    console.log("finish")
})
```

1.8.2 Promise

如果我们希望在一个异步事件结束后开启下一个异步事件，并这样接力下去，最容易想到的方法是将异步函数嵌套使用，把下一次的异步事件放进上一次的回调函数中执行，比如：

```
setTimeout(() => {
    console.log("1");
    setTimeout(() => {
        console.log("2");
        setTimeout(() => {
            console.log("3");
        }, 2000);
    }, 2000);
}, 2000);
```

但这样层层嵌套的可读性比较差，一旦异步事件变多就会比较混乱，故而不太建议这么写。JS中内置的Promise对象(Object)可以将嵌套的结构变成顺序，简化了多次执行异步操作时的代码复杂度，是一种好的编程风格。Promise使得异步方法能够像同步方法一样返回值。异步方法的返回值即为一个Promise对象，只会处在以下几种状态之一：

- 待定：Promise的初始状态，不知道有没有成功
- 已兑现：说明异步事件已成功完成
- 已拒绝：说明异步事件执行失败，需要调用异常处理

Promise的构造函数接受一个函数作为参数，称为起始函数。起始函数包含resolve和reject两个参数，分别作为返回值表示Promise成功和失败的状态，它们同时也是函数。调用 `resolve()` 后， `then()` 方法才能将Promise传下去。
Promise常用的方法有：

- `then()`
- `catch()`
- `finally()`

`resolve()` 中可以放置一个参数用于向当前Promise调用的 `then()` 方法传递一个值， `then()` 中的函数也可以将返回值传递给下一个 `then()` 。该函数返回的既可以是一个值，也可以是一个Promise对象。如果是后者，则会在之前的事件完成后建立一个新的异步事件，并通过继续调用 `then()` 方法，将若干异步事件像链表一样串联在一起。

```
new Promise((resolve, reject) => {
    setTimeout(() => {
        console.log(1);
        resolve(); // 如果成功，则调用resolve()
    }, 2000);
}).then(() => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log(2);
            resolve("OK"); // resolve传给下一个异步事件的值可以被then()方法接收
        }, 2000);
    });
}).then((x) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log(3);
            console.log(x);
        }, 2000);
    });
});

// 可以将重复的部分定义成一个函数
function count(x,time) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log(x);
            resolve();
        },time)
    });
}
// 原来的代码可以改写成：
count(1,2000).then(() => {
    return count(2,2000);
}).then(() => {
    count(3,2000);
});
```

其它可能用到的Promise方法

- `all()`：作为其参数的所有Promise对象都被兑现时，当前Promise才能执行成功
- `any()`：作为其参数的Promise对象中只要有一个能被兑现，就返回该Promise

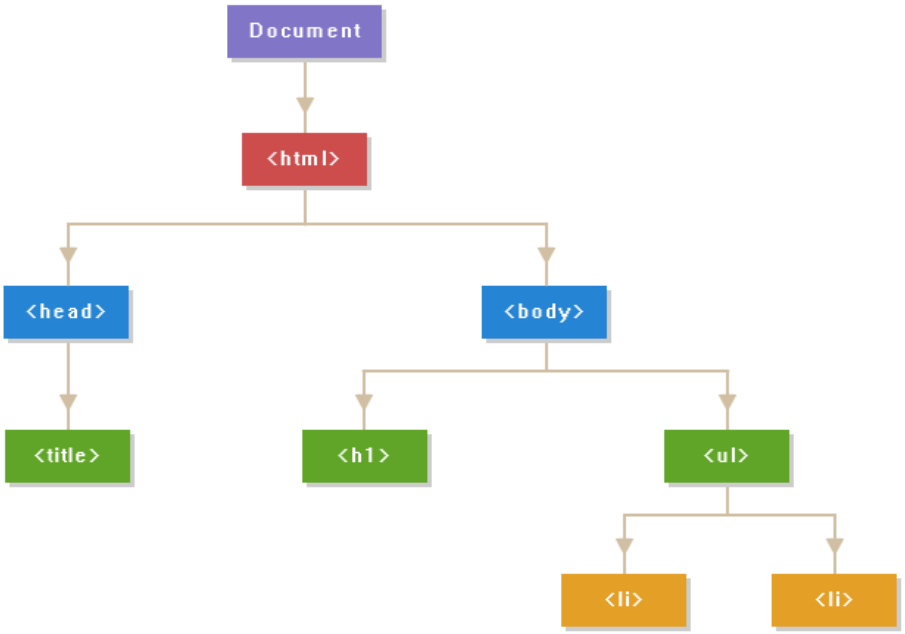
1.8.3 async/await

利用async/await关键字和Promise对象，我们可以很容易地编写异步函数。用await关键字修饰一个返回Promise的函数，使代码在该点上等待该Promise的完成。我们可以如此定义一个异步函数：

```
async function counter() {
    await count(1,2000);
    await count(2,2000);
    await count(3,2000);
}
```

2. 文档对象模型(DOM)

文档对象模型定义了文档的逻辑结构，使得Javascript可以控制HTML文档中的指定元素，是JS最重要的特点之一。当页面加载时，浏览器会自动创建当前页面的文档对象模型，生成该文档的逻辑结构树。文档对象模型通过document对象来调用相关功能。




```
<p id="content">Hello World! </p>
```

查找HTML中的元素

我们可以给HTML文档中元素的id、class属性赋任意值，并在JS中由此来查找它们。

- 通过元素的id属性查找：

```
let x = document.getElementById("content");
```

- 通过元素的class属性查找：

```
let x = document.getElementsByClassName("content");
```

添加/删除HTML元素

`appendChild()`：将新元素作为其它元素的子结点进行添加（添加到尾部）
`insertBefore()`：将新元素添加到其它元素的头部

```
let para = document.createElement("p"); // 创建一个段落
let text = document.createTextNode("abc"); // 创建一个文本结点
para.appendChild(text); // 将该段落与文本合成一个元素 <p>abc</p>
x.appendChild(para);
x.insertBefore(para);
```

删除元素时需要知道该结点的父结点
`removeChild()`：将某元素的子结点移除
一般来说是这样删除结点的：

```
child.parentNode.removeChild(child); // 先找父结点，再删除
```

修改HTML元素

为了修改元素，我们首先需要查找到该元素。其中，返回值x具有多个属性。可以对这些属性重新赋值以实现修改。

- 利用innerHTML属性修改元素的内容：

```
let x = document.getElementById("content"); // 找到对应的元素
x.innerHTML = "hello world";
```

- 利用style属性修改元素的CSS格式：
  style属性下还有color、fontSize等属性，代表颜色、字号等性质

```
document.getElementById("content").style.color="red"; // 使"Hello World! "变成红色的
```

3. 浏览器对象模型(BOM)

JS可以使用window对象及其相关属性，对浏览器进行控制。**DOM的document对象实际上是window对象的属性之一。**
window的常见属性有：

- `open()` / `close()`：打开/关闭浏览器窗口
- `alert()`：弹窗警告

在很多情况下，使用window的属性时都可以省略"window"

4. JS的使用

4.1 引入JS脚本的方法

JS用于描述HTML页面的功能，有两种常用的引入方式：
其一为把JS代码直接写在HTML `<script></script>` 元素里，比如：

```
<script>
    console.log("Hello World!")
</script>
```

其二是通过文件的路径将JS文件整体引入HTML中，比如：

```
<script src="___.js" defer></script>
```

4.2 运行环境

- node.js
- html
- google chrome -> 自定义及控制 -> 更多工具 -> 开发者工具