## 字符串

### 字符串的概念

字符串即我们通常所说的*文本*。它来自计算机深处，却拥有人类可读的形式。它是在人机之间传递信息的使者，一座字符堆垒起来的桥梁。它的定义形式是一对引号 "" 或 '' ，引号中的内容就是它的全部。我们可以使用 + 号来拼接两个字符串，使用 .length 属性来获得它的长度，使用下标 [] 来获得它在某个位置上的字符。

const s = "Hello ";  
alert(s.length); // 6  
alert(s + "world"); // "Hello world"  
alert(s + 'JavaScript'); // "Hello JavaScript"

一个字符串中的每一个字符都具有一个编号，这个编号从 0 开始，最后一个字符的编号就是它的长度减去 1。如果指定编号超出了限度，则会得到一个 undefined。

const s = "Hello world";  
alert(s[0]); // "H";  
alert(s.length); // 11  
alert(s[s.length]); // undefined  
alert(s[s.length - 1]); // "d"  
alert(s[-1]); // undefined

像数值一样，你可以使用 === 和 !== 来比较两个字符串是否完全相等。

let a = "aaa", b = "bbb";  
alert(a === b); // false  
alert(a !== b); // true  
alert(a === "aaa"); // true

练习 3.1.1

1. 编写一个程序，使用 prompt 函数得到用户输入的的名字，用 alert 对这个名字打招呼。
2. 编写一个程序，从用户输入的字符串中获得一个随机位置上的字符。（备注：你可能需要复习“数值”一节。）

### 转义字符

有时我们需要在字符串中使用一些特殊的字符，这些字符无法用通常的方式输入，例如换行符。我们可以使用一类称为*转义字符*的标记来表示这些字符。它们的原理是，通过在普通的字符前加上符号 \ ，来改变这个字符本来的含义或者作用。JavaScript 规定了一些转义字符，如下表。

|  |  |
| --- | --- |
| 转义字符 | 含义 |
| \\ | 反斜杠本身 |
| \n | 换行符 |
| \r | 回车符 |
| \t | 水平制表符 |
| \v | 垂直制表符 |
| \b | 退格符 |
| \f | 换页符 |
| \u | Unicode 码点 |

以下是一些使用示例：

alert("Hello\\world!"); // Hello\world!
  
alert("Hello\nworld!"); // Hello
  
 // world!
  
alert("Hello\rworld!"); // Hello
  
 // world!
  
alert("Hello\tworld!"); // Hello world!
  
// \v \b \f 三个字符在一般的文本编辑器中无法显示
  
alert("Hello\vworld!"); // Hello□world!
  
alert("Hello\bworld!"); // Hello□world!
  
alert("Hello\fworld!"); // Helloworld!

一般情况下我们只需要使用 \\ 来得到 \ 字符本身，或者 \n 作为换行符即可。\u 表示一个 Unicode 码点，关于它的详细内容见下文。

**转义引号**

假如你的字符串里面需要包含单引号和双引号，但是又不能与表示字符串开头结尾的引号冲突，可以用*转义引号*的办法来规避。你可以在字符串中出现的引号前加 \ 以进行转义，如果不会发生冲突，则不需要转义。

const s1 = "He said: \"Hello world!\""; // He said: "Hello world!"
  
const s2 = "He said: "Hello world!""; // SyntaxError: Unexpected identifier
  
const s3 = "He said: \'Hello world!\'"; // He said: 'Hello world!'
  
const s4 = "He said: 'Hello world!'"; // He said: 'Hello world!'

### Unicode

在深入探讨 JavaScript 中的字符串之前，我们首先要了解一下 **Unicode**。

Unicode 是世界上最为通用的字符集，它可以看做一切其他字符集（如 ASCII，GBK 等）的合体，涵盖了目前世界上几乎所有已知的现存书写系统，从欧洲的拉丁和西里尔字母，到远东的汉字、日文、韩文，再到东南亚圆润的字母文字、印第安人的奇特符号，甚至盲文、emoji ，都在 Unicode 这一字符集的涵盖范围内。它的开发与实现遵循 ISO 的国际标准，有许多具体的方式来处理遵循 Unicode 标准的文本内容，包括 UTF-8、UTF-16、UTF-32 等。JavaSciript 中的字符串使用 Unicode 作为处理依据，以便充分融入国际化的 Web 环境中。

Unicode 源于一个很简单的想法：将全世界所有的字符包含在一个集合里，计算机只要支持这一个字符集，就能显示所有的字符，再也不会有乱码了。

——阮一峰（2014年）

Unicode 的核心概念是，从 0 开始，为每一个包含在这个字符集中的字符分配一个独一无二的数字编号，称为“码点（code point）”，并将相应的字形和意义与这个编号一一对应。如：U+0000，U+0FE3，U+CFFF 等。

前缀 U+ 表示紧跟在后面的十六进制数是一个 Unicode 码点，我们一般习惯使用十六进制数来表达 Unicode 码点。

例如，中文“好”的码点是：U+597D。

Unicode 中不同的符号不是一次性全部定义的，而是分成多个区域，每个区域可以存放 216 （65536）个字符，称为一个平面。目前一共有 17 个平面，也就是说，整个 Unicode 字符集的大小现在是 221。

最前面的 65536 个字符位，称为*基本多文种平面*（缩写为 BMP），它的码点范围是从 0 一直到 216-1 ，写成 16 进制就是从U+0000 到 U+FFFF。所有最常见的字符都放在这个平面，这是 Unicode 最先定义和公布的一个平面。剩下的字符都放在辅助平面（缩写为 SMP），码点范围从 U+010000 一直到 U+10FFFF。

**1. UTF-32**

Unicode 只规定了每个字符的码点作为统一标准，而在实际应用中基于 Unicode 标准有多种具体实现方式，它们统称为*编码方法*。最直观的编码方法是，每个码点使用八个十六进制数（即四个字节）表示，字节内容完全对应码点。这种编码方法称为 **UTF-32** 。比如，码点 U+0000 就用四个字节的 0 表示，码点597D就在前面加两个字节的0。

使用 UTF-32 方式编码的“好”和 U+0000（空字符）如下：

U+0000 => 0x0000 0000

U+597D => 0x0000 597D

UTF-32 的优点在于：

1. 转换规则简单直观
2. 查找效率高。

缺点在于：**浪费空间**，同样一份英语文本，用它进行编码所占用的空间是原始的 ASCII 编码的四倍。这是致命的缺点，因此实际上没有人使用这种编码方法，目前的互联网页面标准（HTML 5）就明文规定，网页不能以 UTF-32 方式进行编码。

**2. UTF-8**

人们需要的是一种节省空间的编码方法，于是 **UTF-8** 应运而生。UTF-8 是一种*变长*的编码方法，一个字符的编码长度，从 1 个字节到 4 个字节不等。常用的字符编码较短，而最前面的 128 个字符，只使用 1 个字节表示，与 ASCII 的方式完全相同。

如下表：

|  |  |
| --- | --- |
| 编码范围 | 占用字节数 |
| 0x0000 - 0x007F | 1 |
| 0x0080 - 0x07FF | 2 |
| 0x0800 - 0xFFFF | 3 |
| 0x010000 - 0x10FFFF | 4 |

UTF-8 对存储空间的节省使得它成为互联网上最常用的编码方式。

**3. UTF-16**

**UTF-16** 编码方式介于 UTF-32 与 UTF-8 之间，同时结合了*定长*和*变长*两种编码方法的特点。

它的编码规则很简单：基本多文种平面的字符占用 2 个字节，其余的（不那么常用）的字符占用 4 个字节。也就是说，UTF-16 的编码长度要么是 2 个字节（U+0000 到 U+FFFF），要么是 4 个字节（U+010000 到 U+10FFFF）。根据字符的 Unicode 码点进行相应的 UTF-16 编码的时候，首先区分这是基本多文种平面的字符（码点 U+FFFF 以内），还是辅助平面字符。如果是前者，直接将码点转为对应的十六进制形式，长度为 2 字节。

如： U+597D => 0x597D

而辅助平面字符的编码则根据 Unicode 3.0 标准给出的公式，用 JavaScript 代码编写如下：

// 假设 c 是待编码的字符码点
  
const H = Math.floor((c - 0x10000) / 0x400) + 0xD800;
  
const L = (c - 0x10000) % 0x400 + 0xDC00;

其中 H 就是最终编码的左 4 位十六进制数，L 就是右 4 位。用一个码点为 U+1D306 的字符演示：

// const c = 0x1D306;
  
const H = Math.floor((0x1D306 - 0x10000) / 0x400) + 0xD800;
  
const L = (0x1D306 - 0x10000) % 0x400 + 0xDC00;
  
// H = 0xD834, L = 0xDF06
  
// U+1D306 的 UTF-16 编码即 0xD834 DF06

**4. UCS-2**

JavaScript 所采用的编码方式称为 *UCS-2*，它的出现是基于历史原因，可以看做 UTF-16 的子集。因此，JavaScript 中字符串的每一个字符都至少占用 2 字节空间，BMP 以外的字符则通过两个 2 字节的字符来表示这个字符的编码。这种实现方式称为*代理对*，相关细节可参见本书附录。

2015 年的 ECMAScript 6 标准提供了另一种更加方便的表示非 BMP 字符，可以使用一对大括号将字符码点包裹起来，写在 \u 前缀后面：

alert("\u{ed306}"); // "󭌆"

在 JavaScript 的字符串中，你可以直接打出一个字符并放在字符串中，也可以使用 *Unicode 转义标记*来输入 Unicode 中的其它字符。转义标记写作 \u ，后接字符码点的十六进制表示。如数字表示方式一样，JavaScript 中的十六进制数字不区分大小写。注意：**你至少要写 4 位十六进制数字**，否则会得到一个错误。如下：

alert("\u0041"); // "A"
  
alert("\u41"); // SyntaxError: Invalid Unicode escape sequence
  
alert("\u597D"); // "好"
  
alert("\u4f60\u597d"); // "你好"
  
alert("\u3053\u3093\u306b\u3061\u306f"); // "こんにちは"
  
alert("\uc5b4\ub5bb\uac8c 지내니"); // "어떻게 지내니"
  
alert("\u2600"); // "☀"
  
alert("\u2614"); // "☔"
  
alert("\u3a3\u222b\u221e"); // "Σ∫∞"
  
alert("我是\u5c0f\u53ef\u7231！"); // "我是小可爱"

### 字符串操作

JavaScript 提供了一组实用的字符串操作函数，以便于完成诸多常见的文本操作需求。

**toUpperCase 和 toLowerCase**

这两个函数用于转换一个字符串中的大小写，并返回转换后的结果。

const s = "Hello world";
  
alert(s.toUpperCase()); // "HELLO WORLD"
  
alert(s.toLowerCase()); // "hello world"
  
alert("Madam, I'm Adam.".toUpperCase()) // "MADAM, I'M ADAM."

**replace**

这个函数用于在给定字符串中替换第一个匹配的文本。

const s = "Hello world";
  
const s2 = s.replace("world", "JavaScript");
  
alert(s2); // "Hello JavaScript"
  
alert(s.replace("l", "k")); // "Heklo world"

replace 函数的第一个参数也可以是一个*正则表达式*，用于描述更加复杂的模式或者进行全局性的替换。关于正则表达式的概念和细节我们将在第八章讨论。

**indexOf** 和 **lastIndexOf**

indexOf 函数用于在一个字符串中从左往右搜索第一处匹配的位置，以 0 开始计数。如果没有找到，得到的值为 -1。可以接受第二个参数，用于查找指定出现次数的出现位置。

const s = "Hello world";
  
alert(s.indexOf("H")); // 0
  
alert(s.indexOf("l")); // 2
  
alert(s.indexOf("llo")); // 2
  
alert(s.indexOf("l", 3)); // 3
  
alert(s.indexOf("666")); // -1

lastIndexOf 函数与 indexOf 类似，但是搜索方向相反，从后往前查找。

const s = "Hello world!";
  
alert(s.lastIndexOf("llo")); //2
  
alert(s.lastIndexOf("l")); //9
  
alert(s.lastIndexOf("l", 4)); //3

### 转换规则

* 数字和字符串相加，会将数字转换为十进制形式，与字符串拼接，得到拼接后的结果。
* alert(1 + "1"); // "11"  
  alert("My number is " + 15 + "2"); // "My number is 152"  
  alert("My number is " + 15 + 2); // "My number is 152"
* 数字与字符串相减，会先尝试将字符串转换为数值，然后进行相减。
* alert(100 - "10"); // 90  
  alert("100" - 10); // 90  
  alert("100" - "10"); // 90
* 布尔值在参与数字运算的时候, true 会被转换为 1，false 会被转换为 0，然后进行运算。
* alert(1 + true); // 2  
  alert(1 + false); // 1
* 布尔值与字符串相加，会将布尔值直接转换为字符串。
* alert("I think it is " + true); // "I think it is true"  
  alert("Oh, it's " + false); // "Oh, it's false"
* 布尔值与字符串相减，则会依据上述转换规则，将布尔值和字符串分别转换为数字，然后进行运算。
* alert(true - "1"); // 0  
  alert(false - "10"); // -10

### 模板字符串

假如我们需要在一个字符串里写长长的一段话：

Lorem ipsum dolor sit amet, consectetur adipiscing elit.  
Sed eleifend vitae massa sed porttitor. Aliquam erat volutpat.  
Maecenas feugiat, urna sit amet feugiat gravida,  
felis ante lobortis tortor, vel dictum enim sem vitae eros.  
Vivamus mi eros, feugiat ut ex laoreet,  
commodo mattis nisi. Praesent mollis augue eu ligula scelerisque,  
et accumsan mauris pretium. Cras efficitur vel elit eu varius.  
Integer luctus facilisis dignissim.  
Duis pretium lorem nec risus posuere euismod.  
Quisque leo erat, suscipit eget metus vitae,  
accumsan accumsan ex. Curabitur mattis non neque at hendrerit.  
Praesent sollicitudin, nibh quis maximus vestibulum,  
risus ipsum tempus leo, nec imperdiet quam purus eget sem.  
Proin lectus nibh, viverra et vestibulum sed, lacinia ut ipsum.

这段话很长，中间夹杂着许多换行。而 JavaScript 中本来的字符串定义方式是不支持直接换行的，连成一段就失去了美感，该怎样解决呢？

一个最直观的办法是这样：

const s = "Lorem ipsum dolor sit amet, consectetur adipiscing elit." +  
"Sed eleifend vitae massa sed porttitor. Aliquam erat volutpat.\n" +  
"Maecenas feugiat, urna sit amet feugiat gravida,\n" +  
"felis ante lobortis tortor, vel dictum enim sem vitae eros.\n" +  
"Vivamus mi eros, feugiat ut ex laoreet,\n" +  
"commodo mattis nisi. Praesent mollis augue eu ligula scelerisque,\n" +  
"et accumsan mauris pretium. Cras efficitur vel elit eu varius,\n" +  
"Integer luctus facilisis dignissim.\n" +  
"Duis pretium lorem nec risus posuere euismod.\n" +  
"Quisque leo erat, suscipit eget metus vitae,\n" +  
"accumsan accumsan ex. Curabitur mattis non neque at hendrerit.\n" +  
"Praesent sollicitudin, nibh quis maximus vestibulum,\n" +  
"risus ipsum tempus leo, nec imperdiet quam purus eget sem.\n" +  
"Proin lectus nibh, viverra et vestibulum sed, lacinia ut ipsum.\n";

将字符串根据换行拆成许多小的字符串，每个字符串末尾使用 \n 标记换行，使用 + 一个一个进行拼接。很长时间里人们用的就是这种办法。

如果不考虑文本呈现出来的模样，只是在代码里美观一些，也有一种简便点的解决办法：

const s = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.\  
Sed eleifend vitae massa sed porttitor. Aliquam erat volutpat.\  
Maecenas feugiat, urna sit amet feugiat gravida,\  
felis ante lobortis tortor, vel dictum enim sem vitae eros.\  
Vivamus mi eros, feugiat ut ex laoreet,\  
commodo mattis nisi. Praesent mollis augue eu ligula scelerisque,\  
et accumsan mauris pretium. Cras efficitur vel elit eu varius.\  
Integer luctus facilisis dignissim.\  
Duis pretium lorem nec risus posuere euismod.\  
Quisque leo erat, suscipit eget metus vitae,\  
accumsan accumsan ex. Curabitur mattis non neque at hendrerit.\  
Praesent sollicitudin, nibh quis maximus vestibulum,\  
risus ipsum tempus leo, nec imperdiet quam purus eget sem.\  
Proin lectus nibh, viverra et vestibulum sed, lacinia ut ipsum.";

通过在每一行末尾加上 \ 来将换行*转义*，使其不被认为是语法上一行的截止，连成一整个字符串。

假如我们需要在字符串中插入其它一些运行时才确定的内容，比如一个表达式的运算结果，我们可以使用字符串拼接，将表达式的值拼接在两个字符串中，像这样：

alert("1 + 13 - 32 + 11 + 53 - 29 的结果是" + (1 + 13 - 32 + 11 + 53 - 29) + "。");

如果要拼接的表达式不多还好，多了写起来可就麻烦了。同时考虑到多行字符串写起来的种种不便，在 ECMAScript 6 的标准中规定了一种新的字符串定义法——**模板字符串**。

模板字符串能够扫除已有的问题，它使用 ` （反引号）来标记字符串的开始和结束。一个模板字符串可以像这样使用：

const s = `Lorem ipsum dolor sit amet, consectetur adipiscing elit.  
Sed eleifend vitae massa sed porttitor. Aliquam erat volutpat.  
Maecenas feugiat, urna sit amet feugiat gravida,  
felis ante lobortis tortor, vel dictum enim sem vitae eros.  
Vivamus mi eros, feugiat ut ex laoreet,  
commodo mattis nisi. Praesent mollis augue eu ligula scelerisque,  
et accumsan mauris pretium. Cras efficitur vel elit eu varius.  
Integer luctus facilisis dignissim.  
Duis pretium lorem nec risus posuere euismod.  
Quisque leo erat, suscipit eget metus vitae,  
accumsan accumsan ex. Curabitur mattis non neque at hendrerit.  
Praesent sollicitudin, nibh quis maximus vestibulum,  
risus ipsum tempus leo, nec imperdiet quam purus eget sem.  
Proin lectus nibh, viverra et vestibulum sed, lacinia ut ipsum.`;

如果你将 s 显示出来，就会发现它完全记录了*原文*。不但字符串中间可以直接换行，换行还可以被直接记录下来，不需要再使用单独的 \n 来标记换行。

但是模板字符串的方便之处不仅仅止于此。请看：

const a = 3, b = 2;  
const s = `I had ${a} apples, and Lily gave me ${b} apples.  
Now I have ${a + b} apples.`  
alert(s); // I had 3 apples, and Lily gave me 2 apples.  
 // Now I have 5 apples.

在模板字符串中，你可以使用 ${} 标记来插入一个表达式，这个表达式的值可以直接被插入最终的字符串中，省去了反复拼接的麻烦。如果你在字符串中需要使用 ${} 这三个字符本身，直接使用\进行转义即可：

const a = 3, b = 2;  
const s = `I had \${a} apples, and Lily gave me \${b} apples.  
Now I have \${a + b} apples.`  
alert(s); // I had ${a} apples, and Lily gave me ${b} apples.  
 // Now I have ${a + b} apples.

一个模板字符串和普通字符串在意义和使用上是基本一致的。

**标签模板**

模板字符串的功能不止于此。它可以紧跟在一个函数名后面，该函数将被调用来处理这个模板字符串。这被称为 “ *标签模板* ” 功能（ tagged template ）。

alert`123` // 123  
// 等同于  
alert(123) // 123

前面说过模板字符串和普通字符串在使用上基本一致，因此普通字符串中的转义字符在模板字符串中依然会被转义。如：

alert`Hello\nworld`; // Hello  
 // world

我们可以使用 String.raw 函数，来使模板字符串真正“如实”记录我们写下的内容。

alert(String.raw`Hello\nworld`); // Hello\nworld

String.raw 虽然是一个函数，但是不能使用括号进行调用，自然也不能用于普通字符串。

alert(String.raw(`Hello\nworld`)); // TypeError: Cannot convert undefined or null to object  
alert(String.raw("Hello\nworld")); // TypeError: Cannot convert undefined or null to object  
alert(String.raw"Hello\nworld"); // SyntaxError: missing ) after argument list