基本引用类型

原始值包装类型

为了方便操作原始值，提供了3种特殊的引用类型

- Boolean
- Number
- String

每当用到某个原始值的方法或属性时，后台都会创建一个相应的原始包装类型的对象，暴露出操作原始值的各种方法

```
let s1 = "some text";
let s2 = s1.substring(2);
```

s1是原始值，原始值不是对象，按理说不应该存在方法。经过了一些列后台的处理才变得合法

1. 创建一个String的实例
2. 调用实例上的方法
3. 销毁实例

```
let temp = new String(s1);
let s2 = s1.substring(2);
temp = null;
```

引用类型 与 原始值包装类型主要区别在于对象的生命周期
new实例化引用类型得到的实例会在离开作用域时被销毁，自动创建的原始值包装对象只存在忘问他的那行代码执行期间

```
let s1 = "some text";
s1.color = "red";
console.log(s1.color); // undefined
```

Object构造函数作为工厂方法，能根据传入值的类型返回相应原始值包装类型实例

```
let obj = new Object("some text");
console.log(obj instanceof String); // true
```

构造函数是用new调用的！与转型函数不一样！！

```
let value = "25";
let number = Number(value); // 转型函数
console.log(typeof number); // "number"
let obj = new Number(value); // 构造函数
console.log(typeof obj); // "object"
```

Boolean

Boolean是布尔值的引用类型

- valueOf() 重写，返回原始值true或false
- toString() 重写，返回字符串"true"或"false"

Boolean 是对象！ 在逻辑表达式里面是 true！

```
let falseObject = new Boolean(false);
let result = falseObject && true;
console.log(result); // true
let falseValue = false;
result = falseValue && true;
console.log(result); // false
```

Boolean与布尔值区别

```
console.log(typeof falseObject); // object
console.log(typeof falseValue); // boolean
console.log(falseObject instanceof Boolean); // true
console.log(falseValue instanceof Boolean); // false
```

永远不要去使用Boolean！

Number

Number是数值的引用类型

- valueOf() 重写，返回原始数值
- toString() 重写，返回字符串，可以传参，表示进制

除了继承的方法还有别的方法：

- toFixed() 参数表示返回数值字符串包含 多少位 小数 四舍五入
  有效表示20个小数位
- toExponential() 返回科学计数法表示的字符串
- toPrecision() 返回最合理的结果，参数表示位数
  有效表示1-21个小数位
- isInteger()方法 ES6新增，判断一个数是否保存为整数

```
console.log(Number.isInteger(1)); // true
console.log(Number.isInteger(1.00)); // true
console.log(Number.isInteger(1.01)); // false
```

- isSafeInteger() 判断是否是安全整数，有没有超过范围

String

String是字符串的引用类型
valueOf()、toLocaleString()和 toString()都返回原始字符串值
每个String都有length属性

JavaScript字符（暂时用不到的知识点）

JavaScript 字符串由 16 位码元（code unit）组成。每16位码元对应一个字符
charCodeAt()可以查看指定码元的字符编码
fromCharCode() 可以接收任意数值，返回数值对应的字符拼接成的字符串

```
console.log(message.charCodeAt(2)); // 99
console.log(String.fromCharCode(0x61, 0x62, 0x63, 0x64, 0x65)); // "abcde"
console.log(String.fromCharCode(97, 98, 99, 100, 101)); // "abcde"
```

Unicode 16位码元表示 基本多语言平面 （BMP）足够，但是更多的字符(冷门)得用另外16位去增补平面，这种策略称为代理对

```
let message = "ab😊de";
console.log(message.length); // 6
console.log(message.charAt(1)); // b
console.log(message.charAt(2)); // <?>
console.log(message.charAt(3)); // <?>
console.log(message.charAt(4)); // d
console.log(message.charCodeAt(1)); // 98
console.log(message.charCodeAt(2)); // 55357
console.log(message.charCodeAt(3)); // 56842
console.log(message.charCodeAt(4)); // 100
console.log(String.fromCodePoint(0x1F60A)); // 😊
console.log(String.fromCharCode(97, 98, 55357, 56842, 100, 101)); // ab😊de
```

normalize() 方法 暂时用不到的知识

```
// U+00C5：上面带圆圈的大写拉丁字母 A
console.log(String.fromCharCode(0x00C5)); // Å
// U+212B：长度单位“埃”
console.log(String.fromCharCode(0x212B)); // Å
// U+004：大写拉丁字母 A
// U+030A：上面加个圆圈
console.log(String.fromCharCode(0x0041, 0x030A)); // Å
```

3个一样的字符，不同的表达方式，但是ES认为这三个字符互不相等
需要规范化形式

```
let a1 = String.fromCharCode(0x00C5),
 a2 = String.fromCharCode(0x212B),
 a3 = String.fromCharCode(0x0041, 0x030A);
// U+00C5 是对 0+212B 进行 NFC/NFKC 规范化之后的结果
console.log(a1 === a1.normalize("NFD")); // false
console.log(a1 === a1.normalize("NFC")); // true
console.log(a1 === a1.normalize("NFKD")); // false
console.log(a1 === a1.normalize("NFKC")); // true
// U+212B 是未规范化的
console.log(a2 === a2.normalize("NFD")); // false
console.log(a2 === a2.normalize("NFC")); // false
console.log(a2 === a2.normalize("NFKD")); // false
console.log(a2 === a2.normalize("NFKC")); // false
// U+0041/U+030A 是对 0+212B 进行 NFD/NFKD 规范化之后的结果
console.log(a3 === a3.normalize("NFD")); // true
console.log(a3 === a3.normalize("NFC")); // false
console.log(a3 === a3.normalize("NFKD")); // true
console.log(a3 === a3.normalize("NFKC")); // false

let a1 = String.fromCharCode(0x00C5),
 a2 = String.fromCharCode(0x212B),
 a3 = String.fromCharCode(0x0041, 0x030A);
console.log(a1.normalize("NFD") === a2.normalize("NFD")); // true
console.log(a2.normalize("NFKC") === a3.normalize("NFKC")); // true
console.log(a1.normalize("NFC") === a3.normalize("NFC")); // true
```

字符串操作方法

- concat()，用于字符串拼接，返回新字符串，不改变原字符串

  ```
  let stringValue = "hello ";
  let result = stringValue.concat("world");
  console.log(result); // "hello world"
  console.log(stringValue); // "hello"

  let stringValue = "hello ";
  let result = stringValue.concat("world", "!");
  console.log(result); // "hello world!"
  console.log(stringValue); // "hello"
  ```

  一般还是用+吧

- slice、substr、substring
  提取子字符串
  返回一个子字符串
  参数1个或2个

  - 第一个表示起始位置
  - slice、substring第二个表示结束位置
  - substr第二个表示返回的子字符串字符数

- 第二个省略都代表提取到字符串尾
- slice所有负值都当成倒数第几个
- substr第一个负值当成倒数第几个，第二个负值=0
- substring所有负值=0

```
let stringValue = "hello world";
console.log(stringValue.slice(3)); // "lo world"
console.log(stringValue.substring(3)); // "lo world"
console.log(stringValue.substr(3)); // "lo world"
console.log(stringValue.slice(3, 7)); // "lo w"
console.log(stringValue.substring(3,7)); // "lo w"
console.log(stringValue.substr(3, 7)); // "lo worl"


let stringValue = "hello world";
console.log(stringValue.slice(-3)); // "rld"
console.log(stringValue.substring(-3)); // "hello world"
console.log(stringValue.substr(-3)); // "rld"
console.log(stringValue.slice(3, -4)); // "lo w"
console.log(stringValue.substring(3, -4)); // "hel"
console.log(stringValue.substr(3, -4)); // "" (empty string)
```

字符串位置方法

indexOf和lastIndexOf
一个从头找，一个从尾找，找不到返回-1
fn(sonStr:string,startPosition?:number)
不管是indexOf，还是lastIndexOf，第二个参数都代表在字符串的索引值

字符串包含方法

ES6新增3个判断字符串种是否包含另一个字符串的方法
严格来说并不能算包含

- startWith 是否以参数为起始
- endWith 是否以参数为结束
- includes 是否包含参数

startWith与include可以传第二个参数，表示起始位置
endWith第二个参数表示结束位置

trim()

创建一个字符串的副本，删除前后所有空格符

repeat()

字符串复制

```
let stringValue = "na ";
console.log(stringValue.repeat(16) + "batman");
```

```
// na na na na na na na na na na na na na na batman
```

padStart padEnd

复制字符串并填充，小于指定长度填充字符，默认空格

```
let stringValue = "foo";
console.log(stringValue.padStart(6)); // " foo"
console.log(stringValue.padStart(9, ".")); // "......foo"
console.log(stringValue.padEnd(6)); // "foo "
console.log(stringValue.padEnd(9, ".")); // "foo......"

let stringValue = "foo";
console.log(stringValue.padStart(8, "bar")); // "barbafoo"
console.log(stringValue.padStart(2)); // "foo"
console.log(stringValue.padEnd(8, "bar")); // "foobarba"
console.log(stringValue.padEnd(2)); // "foo"
```

字符串迭代与解构

字符串原型暴露@@iterator方法，表示迭代字符串的每个字符

```
let message = "abc";
let stringIterator = message[Symbol.iterator]();
console.log(stringIterator.next()); // {value: "a", done: false}
console.log(stringIterator.next()); // {value: "b", done: false}
console.log(stringIterator.next()); // {value: "c", done: false}
console.log(stringIterator.next()); // {value: undefined, done: true}
```

for-of和解构都是通过上面的迭代器实现的

```
for (const c of "abcde") {
 console.log(c);
}
// a
// b
// c
// d
// e

let message = "abcde";
console.log([...message]); // ["a", "b", "c", "d", "e"]
```

字符串大小写转换

toLowerCase()、toLocaleLowerCase()、toUpperCase()和toLocaleUpperCase()
toLocaleLowerCase()和toLocaleUpperCase()方法旨在基于特定地区实现（如土耳其语）

字符串模式匹配方法

String类型专门在字符串中实现模式匹配设计了几种方法

- match
  本质上跟 RegExp 对象的 exec()方法相同

```
let text = "cat, bat, sat, fat";
let pattern = /.at/;
// 等价于 pattern.exec(text)
let matches = text.match(pattern);
console.log(matches.index); // 0
console.log(matches[0]); // "cat"
console.log(pattern.lastIndex); // 0
```

- search
  返回第一个匹配项的位置，没找到返回-1

```
let text = "cat, bat, sat, fat";
let pos = text.search(/at/);
console.log(pos); // 1
```

- replace
  子字符串替换
  第一个参数：
  - RegExp对象
  - 字符串（这个字符串不会转换为正则表达式）
    只替换第一个匹配项，要全部替换得 是正则表达式且加全局标记

  第二个参数：一个字符串或者一个函数
  第二个参数如果是字符串也可以实现正则

字符序列	替换文本
$$	$
$&	匹配整个模式的子字符串。与 RegExp.lastMatch 相同
$'	匹配的子字符串之前的字符串。与 RegExp.rightContext 相同
$`	匹配的子字符串之后的字符串。与 RegExp.leftContext 相同
$n	匹配第 n 个捕获组的字符串，其中 n 是 0~9。比如，$1 是匹配第一个捕获组的字符串，$2 是匹配第二个捕获组的字符串，以此类推。如果没有捕获组，则值为空字符串
$nn	匹配第 nn 个捕获组字符串，其中 nn 是 01~99。比如，$01 是匹配第一个捕获组的字符串，$02 是匹配第二个捕获组的字符串，以此类推。如果没有捕获组，则值为空字符串

```
let text = "cat, bat, sat, fat";
result = text.replace(/(.at)/g, "word ($1)");
console.log(result); // word (cat), word (bat), word (sat), word (fat)
```

  如果第二个参数是函数：
  - 一个匹配项 获得参数 匹配的字符串，起始位置，整个字符串

- 多个匹配项每个匹配捕获组的字符串也会作为参数传给这个函数，但最后两个参数还是与整个模式匹配的开始位置和原始字符串。

返回值：替换成什么

```
function htmlEscape(text) {
 return text.replace(/[<>"&]/g, function(match, pos, originalText) {
 switch(match) {
 case "<":
 return "&lt;";
 case ">":
 return "&gt;";
 case "&":
 return "&amp;";
 case "\"":
 return "&quot;";
 }
 });
}
console.log(htmlEscape("<p class=\"greeting\">Hello world!</p>"));
// "&lt;p class=&quot;greeting&quot;&gt;Hello world!</p>"
```

- split
  会根据传入的分隔符将字符串拆分成数组
  分隔符可以是字符串也可以是RegExp对象

```
let colorText = "red,blue,green,yellow";
let colors1 = colorText.split(","); // ["red", "blue", "green", "yellow"]
let colors2 = colorText.split(",", 2); // ["red", "blue"]
let colors3 = colorText.split(/[^,]+/); // ["", ",", ",", ",", ""]
```

localeCompare

比较两个字符串，返回以下3个值中的一个

- 如果按照字母表顺序，字符串应该排在字符串参数前头，则返回负值。（通常是-1，具体还要看与实际值相关的实现。）
- 如果字符串与字符串参数相等，则返回 0。
- 如果按照字母表顺序，字符串应该排在字符串参数后头，则返回正值。（通常是 1，具体还要看与实际值相关的实现。）

看着很长，其实就是每个字母比

```
let stringValue = "yellow";
console.log(stringValue.localeCompare("brick")); // 1
console.log(stringValue.localeCompare("yellow")); // 0
console.log(stringValue.localeCompare("zoo")); // -1

console.log("y".localeCompare("y"))  // 0
console.log("y".localeCompare("Y"))  // -1
```

```
console.log("Y".localeCompare("y"))  // 1
```

HTML方法

满足早期js动态生成html，基本已经没人使用了

方　法	输　出
anchor(*name*)	*string*
big()	<big>*string*</big>
bold()	*string*
fixed()	<tt>*string*</tt>
fontcolor(*color*)	*string*
fontsize(*size*)	*string*
italics()	*string*
link(url)	*string*
small()	*string*
strike()	<strike>*string*</strike>
sub()	_{*string*}
sup()	^{*string*}