一、let和const命令

1、let命令

1.1 基本用法

ES6中新增了let命令，用于变量的声明，基本的用法和var类似。例如：

```
<script>
        // 使用var使用声明变量
        var userName = "bxg";
        console.log("userName=", userName);
        // 使用let声明变量
        let userAge = 18;
        console.log("userAge=", userAge);
    </script>
```

通过以上的代码，我们发现var和let的基本使用是类似的，但是两者还是有本质的区别，最大的区别就是：

使用let所声明的变量只在let命令所在的代码块中有效。

1.2 let与var区别

下面我们通过一个for循环的案例来演示一下let和var的区别,如下所示：

```
<script>
        for (var i = 1; i <= 10; i++) {
            console.log("i=", i)
        }
        console.log("last=", i)
    </script>
```

通过以上的代码，我们知道在循环体中i的值输出的是1--10，最后i的值为11.

但是如果将var换成let会出现什么问题呢？代码如下：

```
   for (let i = 1; i <= 10; i++) {
            console.log("i=", i)
        }
        console.log("last=", i)
```

在循环体中输出的i的值还是1--10,但是循环体外部打印i的值时出现了错误，错误如下：

出错的原因是：通过let声明的变量只在其对应的代码块中起作用，所谓的代码块我们可以理解成就是循环中的这一对大括号。

当然在这里我们通过这个提示信息，可以发现在ES6中默认是启动了严格模式的，严格模式的特征就是：变量未声明不能使用，否则报的错误就是变量未定义。

那么在ES5中怎样开启严格模式呢？我们可以在代码的最开始加上：**“use strict”**

刚才我们说到，let声明的变量只在代码块中起作用，其实就是说明了通过let声明的变量仅在**块级作用域**内有效

1.3 块级作用域

1.3.1 什么是块级作用域?

在这里告诉大家一个最简单的方法: **有一段代码是用大括号包裹起来的，那么大括号里面就是一个块级作用域**

也就是说，在我们写的如下的案例中：

```
for (let i = 1; i <= 10; i++) {
        console.log("i=", i)
    }
    console.log("last=", i)
```

i 这个变量的作用域只在这一对大括号内有效，超出这一对大括号就无效了。

1.3.2 为什么需要块级作用域?

ES5 只有全局作用域和函数作用域，没有块级作用域，这样就会带来一些问题，

```
第一：内层变量可能会覆盖外层变量
```

代码如下：

```
var temp = new Date();
  function show() {
      console.log("temp=", temp)
      if (false) {
          var temp = "hello world";
      }
```

```
        }
        show();
```

执行上面的代码，输出的结果为 *temp=undefined* ，原因就是变量由于提升导致内层的temp变量覆盖了外层的temp变量

第二： 用来计数的循环变量成为了全局变量

关于这一点，在前面的循环案例中，已经能够看到。在这里，可以再看一下

```
<script>
        for (var i = 1; i <= 10; i++) {
            console.log("i=", i)
        }
        console.log("last=", i)
    </script>
```

在上面的代码中，变量i的作用只是用来控制循环，但是循环结束后，它并没有消失，而是成了全局的变量，这不是我们希望的，我们希望在循环结束后，该变量就要消失。

以上两点就是，在没有块级作用域的时候，带来的问题。

下面使用let来改造前面的案例。

```
    let temp = new Date();
        function show() {
            console.log("temp=", temp)
            if (false) {
                let temp = "hello world";
            }

        }
        show();
```

通过上面的代码，可以知道let不像var那样会发生“变量提升”的现象。

第二个问题前面已经讲解过。

1.3.3 ES6块级作用域

let实际上为JavaScript新增了块级作用域，下面再看几个案例，通过这几个案例，巩固一下关于“块级作用域”这个知识点的理解，同时进一步体会块级作用域带来的好处

```
  <script>
        function test() {
```

```
            let num = 5;
            if (true) {
                let num = 10;
            }
            console.log(num)
        }
        test()
    </script>
```

上面的函数中有两个代码块，都声明了变量num,但是输出的结果是5.这表示外层的代码不受内层代码块的影响。如果使用var定义变量num,最后的输出的值就是10.

说一下，下面程序的输出结果是多少?

```
if (true) {
            let b = 20;
            console.log(b)
            if (true) {
                let c = 30;
            }
            console.log(c);
        }
```

输出的结果是：b的值是20，在输出c的时候，出现了错误。

导致的原因，两个if就是两个块级作用域，c这个变量在第二个if中，也就是第二个块级作用域中，所以在外部块级作用域中无法获取到变量c.

块级作用域的出现，带来了一个好处以前获得广泛使用的立即执行匿名函数不再需要了。

下面首先定义了一个立即执行匿名函数：

```
   ;(function text() {
            var temp = 'hello world';
            console.log('temp=', temp);
        })()
```

匿名函数的好处：通过定义一个匿名函数，创建了一个新的函数作用域，相当于创建了一个“私有”的空间，该空间内的变量和方法，不会破坏污染全局的空间 。

但是以上的写法是比较麻烦的，有了“块级作用域”后就编的比较简单了，代码如下：

```
        {
            let temp = 'hello world';
            console.log('temp=', temp);
```

```
    }
```

通过以上的写法，也是创建了一个“私有”的空间，也就是创建了一个封闭的作用域。同样在该封闭的作用域中的变量和方法，不会破坏污染全局的空间。

但是以上写法比立即执行匿名函数简单很多。

现在问你一个问题，以下代码是否可以：

```
let temp = '你好';
    {
        let temp = 'hello world';
    }
```

答案是可以的，因为这里有两个“块级作用域”，一个是外层，一个是内层，互不影响。

但是，现在修改成如下的写法：

```
let temp = '你好';
   {
      console.log('temp=', temp);
      let temp = 'hello world';
   }
```

出错了，也是变量未定义的错误，造成错误的原因还是前面所讲解的let 不存在“变量提升”。

块级作用域还带来了另外一个好处，我们通过以下的案例来体会一下：

该案例希望不同时间打印变量i的值。

```
for (var i = 0; i < 3; i++) {
        setTimeout(function() {
            console.log('i=', i);
        }, 1000)
   }
```

那么上面程序的执行结果是多少?

对了，输出的都是 i=3

造成的原因就是i为全局的。

那么可以怎样解决呢？相信这一点对你来说很简单，在前面ES5课程中也讲过。

```
for (var i = 0; i < 3; i++) {
```

```
            (function(i) {
                setTimeout(function() {
                    console.log('i=', i);
                }, 1000)
            })(i)
        }
```

通过以上的代码其实就是通过自定义一个函数，生成了函数的作用域,i变量就不是全局的了。

这种使用方式很麻烦，有了let命令后，就变的非常的简单了。

代码如下：

```
for (let i = 0; i < 3; i++) {
            setTimeout(function() {
                console.log('i=', i);
            }, 1000)
        }
```

1.4 let命令注意事项

1.4.1 不存在变量提升

let不像var那样会发生“变量提升”现象。所以，变量一定要在声明后使用，否则会出错。

关于这一点，前面的课程也多次强调。

```
console.log(num);
let num = 2;
```

1.4.2 暂时性死区

什么是暂时性死区呢?

先来看一个案例：

```
    var num = 123;
       if (true) {
           num = 666;
           let num;
       }
```

上面的代码中存在全局的变量num,但是在块级作用域内使用了let又声明了一个局部的变量num,导致后面的num绑定到这个块级作用域，所以在let声明变量前，对num进行赋值操作会出错。

所以说，只要在块级作用域中存在let命令，它所声明的变量就被“绑定”在这个区域中，不会再受外部的影响。

关于这一点，ES6明确规定，**如果在区域中存在let命令，那么在这个区域中通过let命令所声明的变量从一开始就生成了一个封闭的作用域，只要在声明变量前使用，就会出错。**

所以说，**所谓的“暂时性死区”指的就是，在代码块内，使用let命令声明变量之前，该变量都是不可用的。**

1.4.3 不允许重复声明

let 不允许在相同的作用域内重复声明一个变量，

如果使用var声明变量是没有这个限制的。

如下面代码所示：

```
function test() {
            var num = 12;
            var num = 20;
            console.log(num)
        }
        test()
```

以上代码没有问题，但是如果将var换成let,就会出错。如下代码所示：

```
function test() {
            let num = 12;
            let num = 20;
            console.log(num)
        }
        test()
```

当然，以下的写法也是错误的。

```
 function test() {
            var num = 12;
            let num = 20;
            console.log(num)
        }
        test()
```

同时，还需要注意，不能在函数内部声明的变量与参数同名，如下所示：

```
  function test(num) {
```

```
        let num = 20;
        console.log(num)
    }
    test(30)
```

2、const命令

2.1 基本用法

const用来声明常量，常量指的就是一旦声明，其值是不能被修改的。

这一点与变量是不一样的，而变量指的是在程序运行中，是可以改变的量。

```
    let num = 12;
    num = 30;
    console.log(num)
```

以上的代码输出结果为:30

但是通过const命令声明的常量，其值是不允许被修改的。

```
    const PI = 3.14;
    PI = 3.15;
    console.log(PI)
```

以上代码会出错。

在以后的编程中，如果确定某个值后期不需要更改，就可以定义成常量，例如:PI,它的取值就是3.14，后面不会改变。所以可以将其定义为常量。

2.2 const命令注意事项

2.2.1 不存在常量提升

以下代码是错误的

```
    console.log(PI);
    const PI = 3.14
```

2.2.2 只在声明的块级作用域内有效

const命令的作用域与let命令相同：只在声明的块级作用域内有效

如下代码所示：

```
if (true) {
    const PI = 3.14;
}
console.log(PI);
```

以上代码会出错

2.2.3 暂时性死区

const命令与let指令一样，都有暂时性死区的问题，如下代码所示：

```
if (true) {
    console.log(PI);
    const PI = 3.14;
}
```

以上代码会出错

2.2.4 不允许重复声明

```
let PI = 3.14;
const PI = 3.14;
console.log(PI);
```

以上代码会出错

2.2.5 常量声明必须赋值

使用const声明常量，必须立即进行初始化赋值，不能后面进行赋值。

如下代码所示：

```
const PI;
PI = 3.14;
console.log(PI);
```

以上代码会出错

二、解构赋值

1、数组解构赋值基本用法

所谓的解构赋值，就是从数组或者是对象中提取出对应的值，然后将提取的值赋值给变量。

首先通过一个案例，来看一下以前是怎样实现的。

```
        let arr = [1, 2, 3];
        let num1 = arr[0];
        let num2 = arr[1];
        let num3 = arr[2];
        console.log(num1, num2, num3);
```

在这里定义了一个数组arr,并且进行了初始化，下面紧跟着通过下标的方式获取数组中的值，然后赋值给对应的变量。

虽然这种方式可以实现，但是相对来说比较麻烦，ES6中提供了解构赋值的方式，代码如下：

```
        let arr = [1, 2, 3];
        let [num1, num2, num3] = arr;
        console.log(num1, num2, num3);
```

将arr数组中的值取出来分别赋值给了，num1,num2和num3.

通过观察，发现解构赋值等号两侧的结构是类似。

下面再看一个案例：

```
let arr = [{
                userName: 'zs',
                age: 18
            },
            [1, 3], 6
        ];
        let [{
                userName,
                age
            },
            [num1, num2], num3
        ] = arr;
        console.log(userName, age, num1, num2, num3);
```

定义了一个arr数组，并且进行了初始化，arr数组中有对象，数组和数值。

现在通过解构赋值的方式，将数组中的值取出来赋给对应的变量，所以等号左侧的结构和数组

arr的结构是一样的。

但是，如果不想获取具体的值，而是获取arr数组存储的json对象，数组，那么应该怎样写呢?

```
let arr = [{
                userName: 'zs',
                age: 18
            },
            [1, 3], 6
        ];
        let [jsonResult, array, num] = arr;
        console.log(jsonResult, array, num);
```

2、注意事项

2.1 如果解析不成功，对应的值会为undefined.

```
        let [num1, num2] = [6]
        console.log(num1, num2);
```

以上的代码中，num1的值为6，num2的值为undefined.

2.2 不完全解构的情况

所谓的不完全解构，表示等号左边只匹配右边数组的一部分。

代码如下：

```
        let [num1, num2] = [1, 2, 3];
        console.log(num1, num2);
```

以上代码的执行结果：num1=1,num2 = 2

也就是只取了数组中的前两个值。

```
        // 如果只取第一个值呢?
       let [num1] = [1, 2, 3];
      console.log(num1);
```

```
        //只取第二个值呢?
        let [, num, ] = [1, 2, 3];
        console.log(num);
```

```
        // 只取第三个值呢?
        let [, , num] = [1, 2, 3];
        console.log(num);
```

3、对象解构赋值基本使用

解构不仅可以用于数组，还可以用于对象。

```
  let {
          userName,
          userAge
      } = {
          userName: 'ls',
          userAge: 20
      };
      console.log(userName, userAge);
```

在对 对象进行解构赋值的时候，一定要注意：变量名必须与属性的名称一致，才能够取到正确的值。

如下所示：

```
      let {
          name,
          age
      } = {
          userName: 'ls',
          userAge: 20
      };
      console.log(name, age);
```

输出的结果都是undefined.

那么应该怎样解决上面的问题呢?

```
  let {
          userName: name,
          userAge: age
      } = {
          userName: 'ls',
          userAge: 20
      }
```

```
        console.log(name, age);
```

通过以上的代码解决了对应的问题，那么这种方式的原理是什么呢?

先找到同名属性，然后再赋值给对应的变量。

把上面的代码，改造成如下的形式，更容易理解:

```
        let obj = {
            userName: 'ls',
            userAge: 21
        };
        let {
            userName: name,
            userAge: age
        } = obj;
        console.log(name, age)
```

如果按照ES5的方式:

```
let name = obj.userName
let age = obj.userAge
```

4、对象解构赋值注意事项

4.1 默认解构

所谓的默认解构，指的是取出来值就用取出来的值，如果取不出来就用默认的值。

演示默认解构之前，先来看如下的代码:

```
        let obj = {
            name: 'zs'
        };
        let {
            name,
            age
        } = obj;
        console.log(name, age);
```

你想一下输出结果是什么呢?

输出的结果是：zs undefined

也就是name变量的值为:‘zs’, age变量的值为:'undefined'.

由于没有给age变量赋值所以该变量的值为'undefined'.

现在修改一下上面的程序

```
        let obj = {
            name: 'zs'
        };
        let {
            name,
            age = 20
        } = obj;
        console.log(name, age);
```

现在给age这个变量赋了一个默认值为20，所以输出的结果为：zs 20

这也就是刚才所说到的默认解构，也就是取出来值就用取出来的值，如果取不出来就用默认的值。

现在再问你一个问题：如果在对应中有age属性，那么对应的等号左侧的age这个变量的值是多少呢?

如下代码所示：

```
        let obj = {
            name: 'zs',
            age: 26
        };
        let {
            name,
            age = 20
        } = obj;
        console.log(name, age);
```

输出的结果为: zs 26

这就是，取出来值就用取出来的，取不出来就用默认值。

4.2 嵌套结构对象的解构

解构也可以用于对嵌套结构的对象，如下代码所示：

```
        let obj = {
            arr: [
```

```
            "Hello", {
                msg: 'World'
            }
        ]
    }
    let {
        arr: [str, {
            msg
        }]
    } = obj;
    console.log(str, msg);
```

在上面的代码中要注意的是：arr只是一种标志或者是一种模式，不是变量，因此不会被赋值。

再看一个案例:

```
    let obj = {
        local: {
            start: {
                x: 20,
                y: 30
            }
        }
    };
    let {
        local: {
            start: {
                x,
                y
            }
        }
    } = obj;
    console.log(x, y);
```

在该案例中创建了一个obj对象，在该对象中又嵌套了一个local对象，该对象可以认为是一个表示位置的坐标对象，在该对象中又嵌套了一个start对象，start对象可以认为是一个位置的起始坐标点，所以在该对象中有两个属性为x,y，分别表示横坐标和纵坐标。

所以说obj对象是一个比较复杂的嵌套结构的对象，现在对该对象进行解构，那么在等号的左侧的结构要和obj对象的结构一致，最后输出打印x,y的值。

问题：如果现在要打印等号左侧的local和start，那么输出的结果是什么呢?

会出错，原因就是在等号的左侧，只有x和y是变量，local和start都是一种标识，一种模式，所以不会被赋值。

5、字符串的解构赋值

字符串也可以进行解构赋值，这是因为字符串被转换成了一个类似于数组的对象。

```
let [a, b, c, d, e, f] = 'itcast';
console.log(a, b, c, d, e, f);
```

类似于数组的对象都有length属性，因此也可以对这个属性进行解构赋值。

```
let {
    length: len
} = 'itcast';
console.log('len=', len);
```

6、函数参数的解构赋值

函数的参数也能够进行解构的赋值，如下代码所示：

```
function test([x, y]) {
    return x + y;
}
console.log(test([3, 6]));
```

上面的代码中，函数test的参数不是一个数组，而是通过解构得到的变量x和y.

函数的参数的解构也可以使用默认的值。

```
function test({
    x = 0,
    y = 0
} = {}) {
    return [x, y];

}
console.log(test({
    x: 3,
    y: 6
}));
```

当然可以进行如下的调用

```
test({x:3})
test({})
```

7、解构赋值的好处

7.1 交换变量的值

```
        let num1 = 3;
        let num2 = 6;
        [num1, num2] = [num2, num1];
        console.log(num1, num2);
```

7.2 函数可以返回多个值

```
        function test() {
            return [1, 2, 3];
        }
        let [a, b, c] = test();
        console.log(a, b, c);
```

在上面的代码中，返回了三个值，当然在实际的开发过程中，你可以根据自己的实际情况确定返回的数据的个数。

如果，我只想接收返回中的一部分值呢?

```
// 接收第一个值
        function test() {
            return [1, 2, 3];
        }
        let [a] = test();
        console.log(a);

        // 接收前两个值
        function test() {
            return [1, 2, 3];
        }
        let [a, b] = test();
        console.log(a, b);
        // 只接收第一个值和第三个值。
        function test() {
            return [1, 2, 3];
```

```
}
let [a, , b] = test();
console.log(a, b);
```

7.3 函数返回一个对象

可以将函数返回的多个值封装到一个对象中。

```
function test() {
    return {
        num1: 3,
        num2: 6
    }
}
let {
    num1,
    num2
} = test();
console.log(num1, num2);
```

7.4 提取JSON对象中的数据

解构赋值对提取JSON对象中的数据也非常有用。

```
let userData = {
    id: 12,
    userName: 'zhangsan',
    userAge: 20
}
let {
    id,
    userName,
    userAge
} = userData;
console.log(id, userName, userAge);
```

以上的代码可以快速提取JSON中的数据。

三、字符串扩展

ES6对字符串的操作也进行了相应的扩展，在这一章节中，重点讲解关于字符串扩展的两部分内容。

第一部分：针对字符串操作扩展的方法

第二部分：模板字符串

1、字符串扩展方法

关于字符串扩展的方法，重点讲解如下方法：

includes( ), startsWith( ),endsWith( ),repeat( )

首先先讲解：includes( ), startsWith( ),endsWith( )这三个方法

传统上，JavaScript中只有indexOf方法可以用来确定一个字符串是否包含在另一个字符串中。但是在ES6中提供了，如下的方法也可以用来判断某个字符串是否包含在另一个字符串中。

`includes()` ：该方法返回结果为布尔值，表示是否找到了对应的字符串。如果找到返回true,否则

代码如下:

```
        let str = "itcast";
        console.log(str.includes("a"));
```

`startsWith()`： 该方法返回结果为布尔值，表示某个字符串是否在另外一个字符串的头部。如果是

代码如下:

```
        let str = "itcast";
        console.log(str.startsWith('i'));
```

`endsWith()`: 该方法返回结果为布尔值，表示某个字符串是否在另外一个字符串的尾部。如果是在尾

代码如下:

```
        let str = "itcast";
        console.log(str.endsWith('t'));
```

下面讲解一下repeat( ) 方法

该方法的作用是返回一个新字符串，表示将原来的字符串重复多少次。

代码如下:

```
        let str = "a";
        console.log(str.repeat(3));
```

注意：如果该方法的参数是一个小数，那么会被取整。如果是一个负数，会出错，但是这个负数如果是0到-1之间的小数，会先进行取整运算，那么得到的值为-0，而repeat( )方法会认为是0. 这时不会出错，但是也不会对字符串进行重复操作。

如果该方法的参数是一个数字的字符串，会先转换成数字。

2、模板字符串

2.1 模板字符串基本使用

在开发的过程中，经常进行字符串的拼接操作。例如：

```
        let userName = 'zs';
        let userAge = 21;
        let str = '大家好，我叫' + userName + ',今年' + userAge + '岁';
        console.log(str);
```

如果字符串结构比较复杂，那么采用以上的方式进行拼接会比较麻烦，例如：我们在开发中经常通过ajax,来获取服务端的数据，然后通过拼接的方式，将数据展示出来，如下面的伪代码所示：

```
$('#result').append("<td>用户编号"+userInfo.Id+"</td>"+"<td>用户姓名"+userInf
```

通过以上的伪代码，可以发现如果字符串的结构比较复杂，采用传统的方式来进行拼接处理，就会非常的麻烦。

为了解决这个问题ES6提供了模板字符串来进行处理。

模板字符串基本应用如下：

```
        let userName = 'zs';
        let userAge = 21;
        let str = `大家好,我叫${userName},今年${userAge}岁`;
        console.log(str);
```

注意：在构建模板字符串时，使用的是**“撇号”**， 也就是反引号。

现在使用模板字符串改造上面的伪代码，如下所示：

```
$('#result').append(`<td>用户编号${userInfo.Id}</td><td>用户姓名${userInfo.us
```

注意：如果在模板字符串中需要使用反引号，需要用到**反斜杠**进行转义。

```
        let userName = 'zs';
        let str = `大家好，我叫\`${userName}\``;
        console.log(str);
```

2.2 模板字符串原理

模板字符串本质上还是进行了相应的字符串的替换操作。使用正则表达式的方式，将模板中的内容用具体的数值替换掉。

下面来模拟一下这个过程。

```
        let userName = 'zs';
        let userAge = 21;
        let str = "我叫${userName},今年${userAge}岁了";
        let newStr = str.replace(/\$\{([^}]+)\}/g, function(matched, key) {
            // console.log(matched); 要替换的字符串也就是 ${userName}和${userA
            // console.log(key);分组的内容，也就是userName和userAge
            return eval(key);
        })
        console.log(newStr);
```

通过正则表达式来进行替换，然后通过eval执行对应的变量。

2.3 模板字符串特性

2.3.1 模板字符串换行

现在我们想要如下的输出效果：

```
        <ul>
            <li>1:zs</li><li>2:ls</li>
        </ul>
```

通过观察可以看出，

和

- 之间是有换行的。体现格式控制

具体的实现代码如下：

```
let users = [{
    userId: 1,
    userName: 'zs'
}, {
    userId: 2,
    userName: 'ls'
}];
let newUserList = [];
for (let i = 0; i < users.length; i++) {
    newUserList.push(`<li>${users[i].userId}:${users[i].userNam
}
let str = `
<ul>
    ${newUserList.join('')}
</ul>
`
console.log(str);
```

2.3.2 标签模板基本使用

什么是标签模板呢?

模板字符串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。这被称之为“标签模板”

```
let userName = 'zs';
let userAge = 21;
let str = showMsg `大家好,我叫${userName},今年${userAge}岁`;
function showMsg(strs, name, age) {
    // 第一个参数：是整个模板中的文本内容，是一个数组
    // 从第二个参数开始为对应的模板中变量的值，这里有两个变量，所以写两个参
    console.log(strs);
    console.log(name);
    console.log(age);
}
```

当然，showMsg这个函数的参数还有另外一种表现形式。如下代码所示：

```
let userName = 'zs';
let userAge = 21;
let str = showMsg `大家好,我叫${userName},今年${userAge}岁`;
function showMsg(strs, ...values) {
    console.log(strs);
    console.log(values);
```

```
    }
```

将模板中变量的值都给了values这个参数，而这个参数是一个数组。

2.3.3 标签模板应用场景

在讲解标签模板有哪些应用场景之前，先来看一个问题。

现在的问题是，前面写的showMsg这个函数，返回的是什么内容呢？这里可以打印str这个变量。

打印的结果为:undefined. 因为在showMsg这个函数中，没有返回值。如果想有返回值，必须通过return来完成。

但是问题是，应该返回什么内容呢？- 返回的是“替换后的内容”，也就是将模板中的变量用具体的数据替换掉后，返回。

怎样进行拼接呢?

可以通过循环的方式，对values这个参数中存储的数据进行遍历，然后与strs这个参数中存储的模板中文本进行拼接就可以了。但是要注意的是：strs这个参数中存储的文本的个数比values存储的数据要多一个。

具体实现如下：

```
let userName = 'zs';
let userAge = 21;
let str = showMsg `大家好,我叫${userName},今年${userAge}岁`;
console.log(str);

function showMsg(strs, ...values) {
    let strResult = '';
    for (let i = 0; i < values.length; i++) {
        strResult += strs[i] + values[i];
    }
    strResult += strs[strs.length - 1];
    return strResult;
}
```

那么这种实现方式，有什么好处呢?

可以在该函数中，定义自己想要的具体的转换格式。例如，将所有的用户名转换成大写进行展示。

```
let userName = 'zs';
let userAge = 21;
```

```
let str = showMsg `大家好,我叫${userName},今年${userAge}岁`;
console.log(str);

function showMsg(strs, ...values) {
    let strResult = '';
    for (let i = 0; i < values.length; i++) {
        strResult += strs[i] + values[i];
    }
    strResult += strs[strs.length - 1];
    return strResult.toUpperCase(); //转换大写
}
```

其实，标签模板还有一个比较重要的应用就是，过滤HTML中的危险字符。

例如，用户在发布评论的时候，如果输入了