TypeScript 循环 TypeScript Number

TypeScript 函数

函数是一组一起执行一个任务的语句。

您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。

函数声明告诉编译器函数的名称、返回类型和参数。函数定义提供了函数的实际主体。

函数定义

函数就是包裹在花括号中的代码块，前面使用了关键词 function：

语法格式如下所示：

```
function function_name()
{
    // 执行代码
}
```

实例

TypeScript

```
function () {
// 函数定义
console.log("调用函数")
}
```

调用函数

函数只有通过调用才可以执行函数内的代码。

语法格式如下所示：

```
function_name()
```

实例

TypeScript

```
function test() { // 函数定义
console.log("调用函数")
}
test() // 调用函数
```

函数返回值

有时，我们会希望函数将执行的结果返回到调用它的地方。

通过使用 return 语句就可以实现。

在使用 return 语句时，函数会停止执行，并返回指定的值。

语法格式如下所示：

```
function function_name():return_type {
    // 语句
    return value;
}
```

- return_type 是返回值的类型。
- return 关键词后跟着要返回的结果。
- 一个函数只能有一个 return 语句。
- 返回值的类型需要与函数定义的返回类型(return_type)一致。

实例

TypeScript

```
// 函数定义
function greet():string { // 返回一个字符串
return "Hello World"
}
function caller() {
var msg = greet() // 调用 greet() 函数
console.log(msg)
}
// 调用函数
caller()
```

- 实例中定义了函数 *greet()*，返回值的类型为 string。
- *greet()* 函数通过 return 语句返回给调用它的地方，即变量 msg，之后输出该返回值。。

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
// 函数定义
function greet() {
return "Hello World";
}
function caller() {
var msg = greet(); // 调用 greet() 函数
console.log(msg);
}
```

```
// 调用函数
caller();
```

带参数函数

在调用函数时，您可以向其传递值，这些值被称为参数。

这些参数可以在函数中使用。

您可以向函数发送多个参数，每个参数使用逗号 `,` 分隔：

语法格式如下所示：

```
function func_name( param1 [:datatype], ( param2 [:datatype]) {
}
```

- param1、param2 为参数名。
- datatype 为参数类型。

实例

TypeScript

```
function add(x: number, y: number): number {
return x + y;
}
console.log(add(1,2))
```

- 实例中定义了函数 *add()*，返回值的类型为 number。
- *add()* 函数中定义了两个 number 类型的参数，函数内将两个参数相加并返回。

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
function add(x, y) {
return x + y;
}
console.log(add(1, 2));
```

输出结果为：

```
3
```

可选参数和默认参数

可选参数

在 TypeScript 函数里，如果我们定义了参数，则我们必须传入这些参数，除非将这些参数设置为可选，可选参数使用问号标识 ？。

实例

TypeScript

```
function buildName(firstName: string, lastName: string) {
return firstName + " " + lastName;
}
let result1 = buildName("Bob"); // 错误，缺少参数
let result2 = buildName("Bob", "Adams", "Sr."); // 错误，参数太多了
let result3 = buildName("Bob", "Adams"); // 正确
```

以下实例，我么将 lastName 设置为可选参数：

TypeScript

```
function buildName(firstName: string, lastName?: string) {
if (lastName)
return firstName + " " + lastName;
else
return firstName;
}
let result1 = buildName("Bob"); // 正确
let result2 = buildName("Bob", "Adams", "Sr."); // 错误，参数太多了
let result3 = buildName("Bob", "Adams"); // 正确
```

可选参数必须跟在必需参数后面。 如果上例我们想让 firstName 是可选的，lastName 必选，那么就要调整它们的位置，把 first Name 放在后面。

如果都是可选参数就没关系。

默认参数

我们也可以设置参数的默认值，这样在调用函数的时候，如果不传入该参数的值，则使用默认参数，语法格式为：

```
function function_name(param1[:type],param2[:type] = default_value) {
}
```

注意：参数不能同时设置为可选和默认。

实例

以下实例函数的参数 rate 设置了默认值为 0.50，调用该函数时如果未传入参数则使用该默认值：

TypeScript

```
function calculate_discount(price:number,rate:number = 0.50) {
var discount = price * rate;
console.log("计算结果: ",discount);
}
calculate_discount(1000)
calculate_discount(1000,0.30)
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
function calculate_discount(price, rate) {
if (rate === void 0) { rate = 0.50; }
var discount = price * rate;
console.log("计算结果: ", discount);
}
calculate_discount(1000);
calculate_discount(1000, 0.30);
```

输出结果为：

```
计算结果:  500
计算结果:  300
```

剩余参数

有一种情况，我们不知道要向函数传入多少个参数，这时候我们就可以使用剩余参数来定义。

剩余参数语法允许我们将一个不确定数量的参数作为一个数组传入。

TypeScript

```
function buildName(firstName: string, ...restOfName: string[]) {
return firstName + " " + restOfName.join(" ");
}
let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

函数的最后一个命名参数 restOfName 以 ... 为前缀，它将成为一个由剩余参数组成的数组，索引值从0（包括）到 restOfNam e.length（不包括）。

TypeScript

```
function addNumbers(...nums:number[]) {
var i;
var sum:number = 0;
for(i = 0;i<nums.length;i++) {
sum = sum + nums[i];
}
console.log("和为: ",sum)
}
addNumbers(1,2,3)
addNumbers(10,10,10,10,10)
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
function addNumbers() {
var nums = [];
for (var _i = 0; _i < arguments.length; _i++) {
nums[_i] = arguments[_i];
}
var i;
var sum = 0;
```

```
for (i = 0; i < nums.length; i++) {
sum = sum + nums[i];
}
console.log("和为: ", sum);
}
addNumbers(1, 2, 3);
addNumbers(10, 10, 10, 10, 10);
```

输出结果为：

```
和为:  6
和为:  50
```

匿名函数

匿名函数是一个没有函数名的函数。

匿名函数在程序运行时动态声明，除了没有函数名外，其他的与标准函数一样。

我们可以将匿名函数赋值给一个变量，这种表达式就成为函数表达式。

语法格式如下：

```
var res = function( [arguments] ) { ... }
```

实例

不带参数匿名函数：

TypeScript

```
var msg = function() {
return "hello world";
}
console.log(msg())
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var msg = function () {
return "hello world";
};
console.log(msg());
```

输出结果为：

```
hello world
```

带参数匿名函数：

TypeScript

```
var res = function(a:number,b:number) {
return a*b;
};
console.log(res(12,2))
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var res = function (a, b) {
return a * b;
};
console.log(res(12, 2));
```

输出结果为：

```
24
```

匿名函数自调用

匿名函数自调用在函数后使用 () 即可：

TypeScript

```
(function () {
var x = "Hello!!";
console.log(x)
})()
<p>编译以上代码，得到以下 JavaScript 代码：</p>
<div class="example"><h2 class="example">JavaScript</h2><div class="example_code">[mycode3 type
="js"]
(function () {
var x = "Hello!!";
console.log(x)
})()
```

输出结果为：

```
Hello!!
```

构造函数

TypeScript 也支持使用 JavaScript 内置的构造函数 Function() 来定义函数：

语法格式如下：

```
var res = new Function( [arguments] ) { ... })
```

实例

TypeScript

```
var myFunction = new Function("a", "b", "return a * b");
var x = myFunction(4, 3);
console.log(x);
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var myFunction = new Function("a", "b", "return a * b");
var x = myFunction(4, 3);
console.log(x);
```

输出结果为：

```
12
```

递归函数

递归函数即在函数内调用函数本身。

> *举个例子：*
> *从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？"从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？'从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？.....'"*

实例

TypeScript

```
function factorial(number) {
if (number <= 0) { // 停止执行
return 1;
} else {
return (number * factorial(number - 1)); // 调用自身
}
};
console.log(factorial(6)); // 输出 720
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
function factorial(number) {
if (number <= 0) { // 停止执行
return 1;
}
else {
return (number * factorial(number - 1)); // 调用自身
}
}
;
console.log(factorial(6)); // 输出 720
```

输出结果为：

```
720
```

Lambda 函数

Lambda 函数也称之为箭头函数。

箭头函数表达式的语法比函数表达式更短。

函数只有一行语句：

```
( [param1, parma2,…param n] )=>statement;
```

实例

以下实例声明了 lambda 表达式函数，函数返回两个数的和：

TypeScript

```
var foo = (x:number)=>10 + x
console.log(foo(100)) //输出结果为 110
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var foo = function (x) { return 10 + x; };
console.log(foo(100)); //输出结果为 110
```

输出结果为：

```
110
```

函数是一个语句块：

```
( [param1, parma2,…param n] )=> {

    // 代码块
}
```

实例

以下实例声明了 lambda 表达式函数，函数返回两个数的和：

TypeScript

```
var foo = (x:number)=> {
x = 10 + x
console.log(x)
```

```
}
foo(100)
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var foo = function (x) {
x = 10 + x;
console.log(x);
};
foo(100);
```

输出结果为：

```
110
```

我们可以不指定函数的参数类型，通过函数内来推断参数类型:

TypeScript

```
var func = (x)=> {
if(typeof x=="number") {
console.log(x+" 是一个数字")
} else if(typeof x=="string") {
console.log(x+" 是一个字符串")
}
}
func(12)
func("Tom")
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var func = function (x) {
if (typeof x == "number") {
console.log(x + " 是一个数字");
}
else if (typeof x == "string") {
console.log(x + " 是一个字符串");
}
};
func(12);
func("Tom");
```

输出结果为：

```
12 是一个数字
Tom 是一个字符串
```

单个参数 `()` 是可选的：

TypeScript

```
var display = x => {
console.log("输出为 "+x)
}
display(12)
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var display = function (x) {
console.log("输出为 " + x);
};
display(12);
```

输出结果为：

```
输出为 12
```

无参数时可以设置空括号：

TypeScript

```
var disp =()=> {
console.log("Function invoked");
}
disp();
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var disp = function () {
console.log("调用函数");
};
disp();
```

输出结果为：

```
调用函数
```

函数重载

重载是方法名字相同，而参数不同，返回类型可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

参数类型不同：

```
function disp(string):void;
function disp(number):void;
```

参数数量不同：

```
function disp(n1:number):void;
function disp(x:number,y:number):void;
```

参数类型顺序不同：

```
function disp(n1:number,s1:string):void;
function disp(s:string,n:number):void;
```

如果参数类型不同，则参数类型应设置为 **any**。

参数数量不同你可以将不同的参数设置为可选。

实例

以下实例定义了参数类型与参数数量不同：

TypeScript

```
function disp(s1:string):void;
function disp(n1:number,s1:string):void;
function disp(x:any,y?:any):void {
console.log(x);
console.log(y);
}
disp("abc")
disp(1,"xyz");
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
function disp(x, y) {
console.log(x);
console.log(y);
}
disp("abc");
disp(1, "xyz");
```

输出结果为：

```
abc
undefined
1
xyz
```

← TypeScript 循环 TypeScript Number →

点我分享笔记