包装类型

包装类型概述

在 JavaScript 中，对应原始类型提供了包装类型。通过包装类型可以创建原始类型的对象（后面的课程学习）。

由于 JavaScript 是区分大小写的，从写法上来说，原始类型是全部小写，包装类型则是全部大写。

一般不建议使用包装类型定义对应的数据类型，但包装类型提供了操作相应值的方法。

> **值得注意的是:** 包装类型涉及到对象的概念，具体技术内容会在后面的课程学习。

Boolean 类型

Boolean 类型是原始类型 boolean 类型对应的包装类型。

```
1 | var bool = new Boolean(true);
```

boolean 类型与 Bollean 类型的区别:

- typeof 运算符对原始类型返回 "boolean"，而对包装类型为 "object"。
- instanceof 运算符测试 Boolean 类型返回 true，而测试 boolean 类型返回 false。

> **值得注意的是:** 不建议使用 Boolean 类型。

Number 类型

Number 类型是原始类型 number 类型对应的包装类型。

```
1 | var num = new Number(10);
```

number 类型与 Number 类型的区别:

- typeof 运算符对原始类型返回 "number"，而对包装类型为 "object"。
- instanceof 运算符测试 Number 类型返回 true，而测试 number 类型返回 false。

> **值得注意的是:** 不建议使用 Number 类型。

String 类型

String 类型是原始类型 string 类型对应的包装类型。

```
var str = new String("hello world");
```

string 类型与 String 类型的区别:

- typeof 运算符对原始类型返回 "string"，而对包装类型为 "object"。
- instanceof 运算符测试 String 类型返回 true，而测试 string 类型返回 false。

> **值得注意的是:** 不建议使用 String 类型。

instanceof 运算符

instanceof 运算符的左操作数是一个包装类型的变量，右操作数是对应的数据类型。如果左侧的变量是右侧的数据类型，则表达式返回 true；否则返回 false。例如下述代码:

```
var str = "this is message";
str instanceof String;// 计算结果为 true, str是String类型
str instanceof Object;// 计算结果为 true, 所有包装类型都是Object的实例
str instanceof Number;// 计算结果为 false
```

> **值得注意的是:** 所有对象都是 Object 类型的实例对象，通过 instanceof 运算符判断一个对象是否为具体数据类型，也包含"父类"。（后面课程会学习）

特殊类型

undefined

JavaScript 中有两个表示空的数据类型，undefined 和 null，其中比较有用的是 undefined。undefined 类型只有一个值，就是 undefined。

下列情况会返回undefined:

- 访问未修改的变量 undefined
- 没有定义 return 表达式的函数隐式返回 undefined
- return 表达式没有显式的返回任何内容
- 访问不存在的属性
- 任何被设置为 undefined 值的变量

null

null 类型是 JavaScript 中的一个特殊类型，用于表示一个不再指向任何内存空间地址的变量。

null 值多用于释放 JavaScript 中的资源（变量、数组和函数等）。

> **值得注意的是:** 使用 typeof 运算符计算 null 的话，返回的是 object。

```
var atguigu = null;
console.log(atguigu);// 输出 null
```

undefined 与 null

- 共同点: 都是原始类型，保存在栈中。
- 不同点
  - undefined: 表示变量声明但未被赋值，是所有未赋值变量的默认值。一般很少主动使用。
  - null: 表示一个没有指向任何内存地址的变量，将来可能指向某个具体内存地址。一般用于主动释放资源。

类型转换

隐式类型转换

由于 JavaScript 是弱类型/松散类型的，在任何情况下都可以强制转换。

- 转换为字符串: 将一个值加上空字符串可以轻松转换为字符串类型。

```
'' + 10 === '10'; // true
```

- 转换为数字: 使用一元的加号操作符，可以把字符串转换为数字。

```
+'10' === 10; // true
```

- 转换为布尔值: 使用否操作符两次，可以把一个值转换为布尔型。

```
!!'foo'; // true
```

显式类型转换

- 使用 JavaScript 的包装类型的构造函数进行类型转换。

构造函数	描述
Number()	将字符串或布尔值转换为数字，如果包含非法字符，则返回NaN。
String()	将数字或布尔值转换为字符串。
Boolean()	将字符串或数字转换为布尔值。

- 使用数据类型的转换函数进行类型转换。

构造函数	描述
toString()	将数字或布尔值转换为字符串。
parseInt()	将字符串或布尔值转换为整数类型。
parseFloat()	将字符串或布尔值转换为浮点类型。

运算符

JavaScript 提供了一组用于操作数据值的运算符。

- 算数运算符（+ - * / % ++ --）
- 比较运算符（> >= < <= == != === !==）
- 逻辑运算符（&& ll !）
- 赋值运算符（= += -= *= /= %= ）
- 字符串连接运算符（+）
- 三元运算符（? :）
- 特殊运算符（typeof instanceof delete）

算数运算符

给定 A=20 B=10 条件，下述表格描述算数运算符:

运算符	描述	例子
+	两个运算数相加	A + B = 30
-	第一个运算数减去第二个运算数	A – B = 10
*	两个运算数相乘	A * B = 200
/	第一个运算数除以第二个运算数	A / B = 2
%	求余运算符，计算整除后的余数	A % B = 0
++	增量运算符，整数值逐次加 1	A++ = 21
--	减量运算符，整数值逐次减 1	A-- = 19

算数运算符的基本操作比较简单，但下述情况需要特别注意:

- 如果运算数中的一个或两个是字符串类型，JavaScript 会自动转换为数字值，再进行计算。
- 如果运算数中的一个或两个是字符串类型，但其中的字符不是数字，JavaScript 会自动转换数字值失败，得到 NaN 结果。
- 任何一个运算数是 NaN，结果都是 NaN。
- 布尔值 false 和 true 会转换为 0 和 1 进行计算。

求余运算符

求余运算符，用于计算两个运算数整除后的余数。

```
console.log( 10 % 3 );// 输出 1

console.log( -10 % 3 );// 输出 -1

console.log( 10 % -3 );// 输出 1

console.log( -10 % -3 );// 输出 -1
```

自增运算符

自增运算符，用于整数值逐次加 1。分别具有两种用法:

- 前置型：自增运算符位于运算数之前。先加 1，再赋值。
- 后置型：自增运算符位于运算数之后。先赋值，再加 1。

```
var x = 3;
console.log( x++ );// 输出 3
console.log( x );// 输出 4

var y = 3;
console.log( ++y );// 输出 4
console.log( y );// 输出 4
```

自减运算符

自减运算符，用于整数值逐次减 1。分别具有两种用法:

- 前置型：自增运算符位于运算数之前。先减 1，再赋值。
- 后置型：自增运算符位于运算数之后。先赋值，再减 1。

```
var x = 3;
console.log( x-- );// 输出 3
console.log( x );// 输出 2

var y = 3;
console.log( --y );// 输出 2
console.log( y );// 输出 2
```

比较运算符

给定 A=20 B=10条件，下述表格描述比较运算符:

运算符	描述	例子
==	检查两个运算数的值是否相等，如果相等则结果为 true	A == B 为 false
!=	检查两个运算数的值是否不等，如果不等则结果为 true	A != B 为 true
>	检查左边运算数是否大于右边运算数，如果是则结果为 true	A > B 为 true
>=	检查左边运算数是否大于或等于右边运算数，如果是则结果为 true	A >= B 为 true
<	检查左边运算数是否小于右边运算数，如果是则结果为 true	A < B 为 false
<=	检查左边运算数是否小于或等于右边运算数，如果是则结果为 true	A <= B 为 false

全等与全不等

运算符	描述
===	两个运算数的值相等并且类型相同时，结果为 true
!==	两个运算数的值不等或者类型不同时，结果为 true

```
var x = 10;
var y = '10';

console.log( x == y );// 输出 true
console.log( x === y );// 输出 false
console.log( x != y );// 输出 false
console.log( x !== y );// 输出 true
```

isNaN 函数

isNaN() 函数用于判断其参数是否为 NaN（非数字值）。

多用于检测使用类型转换函数进行数据类型转换后的结果是否为合法的数字值。

> **值得注意的是:** NaN 与任何值（包括自身）进行比较，结果都是 false。不能使用 `==` 或者 `===` 运算符判断某个值是否是 NaN，而只能使用isNaN() 函数。

```
console.log(isNaN(parseInt('123.45a')));// 输出 true

console.log(isNaN('123.45a'));// 输出 true

console.log(isNaN(Number('123.45a')));// 输出 true
```

逻辑运算符

给定 A=20 B=10条件，下述表格描述比较运算符:

运算符	描述	例子
`&&`	逻辑与运算符。如果两个运算数都是 true，则返回 true	A && B 为 true
`||`	逻辑或运算符。如果两个运算数中任何一个是 true，则返回 true	A
`!`	逻辑非运算符。用于改变运算数的逻辑状态。如果逻辑状态为 true，则通过逻辑非运算符是逻辑状态改为 false	!(A && B) 为 false

逻辑与运算符

B1	B2	B1 && B2
false	false	false
false	true	false
true	false	false
true	true	true

```
console.log( false && true );// 输出 false
console.log( true && true );// 输出 true

// 数字值 1 和 0 转换为布尔值 true 和 false
console.log( 1 && 0 );// 输出 false

// 空字符串转换为布尔值 false，非空字符串转换为布尔值 true
console.log( "" && "atguigu" );// 输出 false
```

逻辑或运算符

B1	B2	B1 或 B2
false	false	false
false	true	true
true	false	true
true	true	true

```
console.log( false || true );// 输出 true
console.log( false || false );// 输出 false

// 数字值 1 和 0 转换为布尔值 true 和 false
console.log( 1 || 0 );// 输出 true

// 空字符串转换为布尔值 false，非空字符串转换为布尔值 true
console.log( "" || "atguigu" );// 输出 true
```

逻辑非运算符

B1	!B1
false	true
true	false

```
console.log( !true );// 输出 false

console.log( !1 );// 输出 false

console.log( !"atguigu" );// 输出 false
```

> **值得注意的是:** 能被转换为 false 的值有null, 0, NaN, 空字符串("") 和 undefined。

逻辑短路原则

所谓短路原则，就是只要确定运算符前面的运算数为 true 或 false，就可以确定返回结果为 true 或 false。

- 逻辑与运算符
  - 逻辑与运算符前面为false，结果都将返回逻辑与运算符前面的。
  - 逻辑与运算符前面为true，结果都将返回逻辑与运算符后面的值。

- 逻辑或运算符
  - 逻辑或运算符前面为false，结果都将返回逻辑或运算符后面的值。
  - 逻辑或运算符前面为true，结果都将返回逻辑或运算符前面的值。

赋值运算符

赋值运算符用于为变量或属性进行赋值操作。

```
var atguigu = "atguigu";// 将字符串 "atguigu" 赋值给变量 atguigu
var obj.x = 1;// 将数字值 1 赋值给 obj 对象的 x 属性
```

赋值运算符就是将右边运算数的值赋给左边运算数。

```
C = A + B;// 将A+B的值赋给C
```

运算符	描述	例子
+=	加等赋值运算符，将右边运算符与左边运算符相加并将运算结果赋给左边运算数	C += A 相当于 C = C + A
-=	减等赋值运算符，将左边运算数减去右边运算数并将运算结果赋给左边运算数	C -= A 相当于 C = C - A
*=	乘等赋值运算符，将右边运算数乘以左边运算数并将运算结果赋给左边运算数	C *= A 相当于 C = C * A
/=	除等赋值运算符， 将左边运算数除以右边运算数并将运算结果赋值给左边运算数	C /= A 相当于 C = C / A
%=	模等赋值运算符，用两个运算数做取模运算并将运算结果赋值给左边运算数	C %= A 相当于 C = C % A

> **值得注意的是:** C += A由于运行时可以进行优化，执行效率都要优于C = C + A。

字符串连接运算符

字符串连接运算符使用的是加法运算符（+）。

- 两个运算数都是数字值时，"+"用于两个运算数相加计算。
- 两个运算数中的一个是字符串时，"+"用于字符串连接计算。

```
var num1 = 1;
var num2 = 2;
var num3 = num1 + num2; // 加法计算

var num4 = "4";
var num5 = num1 + num4; //字符串拼接计算
```

版权说明

联系方式

可以问职业问题。
不要问技术问题。

可以问职业问题。
可以问技术问题。

可以问职业问题。
可以问技术问题。