第 1 章　类型

大多数开发者认为，像 JavaScript 这样的动态语言是没有**类型**（type）的。让我们来看看 ES5.1 规范（http://www.ecma-international.org/ecma-262/5.1/）对此是如何界定的：

> 本规范中的运算法则所操纵的值均有相应的类型。本节中定义了所有可能出现的类型。ECMAScript 类型又进一步细分为语言类型和规范类型。
>
> ECMAScript 语言中所有的值都有一个对应的语言类型。ECMAScript 语言类型包括 Undefined、Null、Boolean、String、Number 和 Object。

喜欢强类型（又称静态类型）语言的人也许会认为“类型”一词用在这里不妥。“类型”在强类型语言中的涵义要广很多。

也有人认为，JavaScript 中的“类型”应该称为“标签”（tag）或者“子类型”（subtype）。

本书中，我们这样来定义“类型”（与规范类似）：对语言引擎和开发人员来说，**类型**是值的内部特征，它定义了值的行为，以使其区别于其他值。

换句话说，如果语言引擎和开发人员对 `42`（数字）和 `"42"`（字符串）采取不同的处理方式，那就说明它们是不同的**类型**，一个是 `number`，一个是 `string`。通常我们对数字 `42` 进行数学运算，而对字符串 `"42"` 进行字符串操作，比如输出到页面。它们是不同的类型。

上述定义并非完美，不过对于本书已经足够，也和 JavaScript 语言对自身的描述一致。

1.1　类型

撇开学术界对类型定义的分歧，为什么说 JavaScript 是否有类型也很重要呢?

要正确合理地进行类型转换（参见第 4 章），我们必须掌握 JavaScript 中的各个类型及其内在行为。几乎所有的 JavaScript 程序都会涉及某种形式的强制类型转换，处理这些情况时我们需要有充分的把握和自信。

如果要将 `42` 作为 `string` 来处理，比如获得其中第二个字符 `"2"`，就需要将它从 `number`（强制类型）转换为 `string`。

这看似简单，但是强制类型转换形式多样。有些方式简明易懂，也很安全，然而稍不留神，就会出现意想不到的结果。

强制类型转换是 JavaScript 开发人员最头疼的问题之一，它常被诟病为语言设计上的一个缺陷，太危险，应该束之高阁。

全面掌握 JavaScript 的类型之后，我们旨在改变对强制类型转换的成见，看到它的好处并且意识到它的缺点被过分夸大了。现在先让我们来深入了解一下值和类型。

1.2　内置类型

JavaScript 有七种内置类型：

- 空值（`null`）
- 未定义（`undefined`）
- 布尔值（`boolean`）
- 数字（`number`）
- 字符串（`string`）
- 对象（`object`）
- 符号（`symbol`，ES6 中新增）

除对象之外，其他统称为“基本类型”。

我们可以用 `typeof` 运算符来查看值的类型，它返回的是类型的字符串值。有意思的是，这七种类型和它们的字符串值并不一一对应：

```
typeof undefined     === "undefined"; // true
typeof true          === "boolean";   // true
typeof 42            === "number";    // true
typeof "42"          === "string";    // true
typeof { life: 42 }  === "object";    // true

// ES6中新加入的类型
typeof Symbol()      === "symbol";    // true
```

以上六种类型均有同名的字符串值与之对应。符号是 ES6 中新加入的类型，我们将在第 3 章中介绍。

你可能注意到 `null` 类型不在此列。它比较**特殊**，`typeof` 对它的处理有问题：

```
typeof null === "object"; // true
```

正确的返回结果应该是 `"null"`，但这个 bug 由来已久，在 JavaScript 中已经存在了将近二十年，也许永远也不会修复，因为这牵涉到太多的 Web 系统，“修复”它会**产生**更多的 bug，令许多系统无法正常工作。

我们需要使用复合条件来检测 `null` 值的类型：

```
var a = null;

(!a && typeof a === "object"); // true
```

`null` 是基本类型中唯一的一个“假值”（falsy 或者 false-like，参见第 4 章）类型，`typeof` 对它的返回值为 `"object"`。

还有一种情况：

```
typeof function a(){ /* .. */ } === "function"; // true
```

这样看来，`function`（函数）也是 JavaScript 的一个内置类型。然而查阅规范就会知道，它实际上是 `object` 的一个“子类型”。具体来说，函数是“可调用对象”，它有一个内部属性 `[[Call]]`，该属性使其可以被调用。

函数不仅是对象，还可以拥有属性。例如：

```
function a(b,c) {
    /* .. */
}
```

函数对象的 `length` 属性是其声明的参数的个数：

```
a.length; // 2
```

因为该函数声明了两个命名参数，`b` 和 `c`，所以其 `length` 值为 `2`。

再来看看数组。JavaScript 支持数组，那么它是否也是一个特殊类型？

```
typeof [1,2,3] === "object"; // true
```

不，数组也是对象。确切地说，它也是 `object` 的一个“子类型”（参见第 3 章），数组的元素按数字顺序来进行索引（而非普通像对象那样通过字符串键值），其 `length` 属性是元素的个数。

1.3　值和类型

JavaScript 中的变量是没有类型的，**只有值才有**。变量可以随时持有任何类型的值。

换个角度来理解就是，JavaScript 不做“类型强制”；也就是说，语言引擎不要求**变量**总是持有与其**初始值同类型**的值。一个变量可以现在被赋值为字符串类型值，随后又被赋值为数字类型值。

`42` 的类型为 `number`，并且无法更改。而 `"42"` 的类型为 `string`。数字 `42` 可以通过**强制类型转换**（coercion）为字符串 `"42"`（参见第 4 章）。

在对变量执行 `typeof` 操作时，得到的结果并不是该变量的类型，而是该变量持有的值的类型，因为 JavaScript 中的变量没有类型。

```
var a = 42;
typeof a; // "number"

a = true;
typeof a; // "boolean"
```

`typeof` 运算符总是会返回一个字符串：

```
typeof typeof 42; // "string"
```

`typeof 42` 首先返回字符串 `"number"`，然后 `typeof "number"` 返回 `"string"`。

1.3.1 `undefined` 和 undeclared

变量在未持有值的时候为 `undefined`。此时 `typeof` 返回 `"undefined"`：

```
var a;

typeof a; // "undefined"

var b = 42;
var c;

// later

b = c;

typeof b; // "undefined"
typeof c; // "undefined"
```

大多数开发者倾向于将 undefined 等同于 undeclared（未声明），但在 JavaScript 中它们完全是两回事。

已在作用域中声明但还没有赋值的变量，是 undefined 的。相反，还没有在作用域中声明过的变量，是 undeclared 的。

例如：

```
var a;

a; // undefined
b; // ReferenceError: b is not defined
```

浏览器对这类情况的处理很让人抓狂。上例中，“b is not defined”容易让人误以为是“b is undefined”。这里再强调一遍，“undefined”和“is not defined”是两码事。此时如果浏览器报错成“b is not found”或者“b is not declared”会更准确。

更让人抓狂的是 `typeof` 处理 undeclared 变量的方式。例如：

```
var a;

typeof a; // "undefined"

typeof b; // "undefined"
```

对于 undeclared（或者 not defined）变量，`typeof` 照样返回 `"undefined"`。请注意虽然 `b` 是一个 undeclared 变量，但 `typeof b` 并没有报错。这是因为 `typeof` 有一个特殊的安全防范机制。

此时 `typeof` 如果能返回 undeclared（而非 undefined）的话，情况会好很多。

1.3.2 `typeof` Undeclared

该安全防范机制对在浏览器中运行的 JavaScript 代码来说还是很有帮助的，因为多个脚本文件会在共享的全局命名空间中加载变量。

很多开发人员认为全局命名空间中不应该有变量存在，所有东西都应该被封装到模块和私有 / 独立的命名空间中。理论上这样没错，却不切实际。然而这仍不失为一个值得为之努力奋斗的目标。好在 ES6 中加入了对模块的支持，这使我们又向目标迈近了一步。

举个简单的例子，在程序中使用全局变量 `DEBUG` 作为“调试模式”的开关。在输出调试信息到控制台之前，我们会检查 `DEBUG` 变量是否已被声明。顶层的全局变量声明 `var DEBUG = true` 只在 debug.js 文件中才有，而该文件只在开发和测试时才被加载到浏览器，在生产环境中不予加载。

问题是如何在程序中检查全局变量 `DEBUG` 才不会出现 `ReferenceError` 错误。这时 `typeof` 的安全防范机制就成了我们的好帮手：

```
// 这样会抛出错误
if (DEBUG) {
    console.log( "Debugging is starting" );
}

// 这样是安全的
if (typeof DEBUG !== "undefined") {
    console.log( "Debugging is starting" );
}
```

这不仅对用户定义的变量（比如 `DEBUG`）有用，对内建的 API 也有帮助：

```
if (typeof atob === "undefined") {
    atob = function() { /*..*/ };
}
```

如果要为某个缺失的功能写 polyfill（即衬垫代码或者补充代码，用来补充当前运行环境中缺失的功能），一般不会用 `var atob` 来声明变量 `atob`。如果在 `if` 语句中使用 `var atob`，声明会被提升（hoisted，参见《你不知道的 JavaScript（上卷）》[1] 中的“作用域和闭包”部分）到作用域（即当前脚本或函数的作用域）的最顶层，即使 `if` 条件不成立也是如此（因为 `atob` 全局变量已经存在）。在有些浏览器中，对于一些特殊的内建全局变量（通常称为“宿主对象”，host object），这样的重复声明会报错。去掉 `var` 则可以防止声明被提升。

[1]此书已由人民邮电出版社出版。——编者注

还有一种不用通过 `typeof` 的安全防范机制的方法，就是检查所有全局变量是否是全局对象的属性，浏览器中的全局对象是 `window`。所以前面的例子也可以这样来实现：

```
if (window.DEBUG) {
    // ..
}

if (!window.atob) {
    // ..
}
```

与 undeclared 变量不同，访问不存在的对象属性（甚至是在全局对象 `window` 上）不会产生 `ReferenceError` 错误。

一些开发人员不喜欢通过 `window` 来访问全局对象，尤其当代码需要运行在多种 JavaScript 环境中时（不仅仅是浏览器，还有服务器端，如 node.js 等），因为此时全局对象并非总是 `window`。

从技术角度来说，`typeof` 的安全防范机制对于非全局变量也很管用，虽然这种情况并不多见，也有一些开发人员不大愿意这样做。如果想让别人在他们的程序或模块中复制粘贴你的代码，就需要检查你用到的变量是否已经在宿主程序中定义过：

```
function doSomethingCool() {
    var helper =
        (typeof FeatureXYZ !== "undefined") ?
        FeatureXYZ :
        function() { /*.. default feature ..*/ };
```

```
    var val = helper();
    // ..
}
```

其他模块和程序引入 `doSomethingCool()` 时，`doSomethingCool()` 会检查 `FeatureXYZ` 变量是否已经在宿主程序中定义过；如果是，就用现成的，否则就自己定义：

```
// 一个立即执行函数表达式（IIFE，参见《你不知道的JavaScript（上卷）》“作用域和闭包”
// 部分的3.3.2节）
(function(){
    function FeatureXYZ() { /*.. my XYZ feature ..*/ }

    // 包含doSomethingCool(..)
    function doSomethingCool() {
        var helper =
            (typeof FeatureXYZ !== "undefined") ?
            FeatureXYZ :
            function() { /*.. default feature ..*/ };

        var val = helper();
        // ..
    }

    doSomethingCool();
})();
```

这里，`FeatureXYZ` 并不是一个全局变量，但我们还是可以使用 `typeof` 的安全防范机制来做检查，因为这里没有全局对象可用（像前面提到的 `window.___`）。

还有一些人喜欢使用“依赖注入”（dependency injection）设计模式，就是将依赖通过参数显式地传递到函数中，如：

```
function doSomethingCool(FeatureXYZ) {
    var helper = FeatureXYZ ||
        function() { /*.. default feature ..*/ };
    var val = helper();
    // ..
}
```

上述种种选择和方法各有利弊。好在 `typeof` 的安全防范机制为我们提供了更多选择。

1.4　小结

JavaScript 有七种内置类 型：`null`、`undefined`、`boolean`、`number`、`string`、`object` 和 `symbol`，可以使用 `typeof` 运算符来查看。

变量没有类型，但它们持有的值有类型。类型定义了值的行为特征。

很多开发人员将 undefined 和 undeclared 混为一谈，但在 JavaScript 中它们是两码事。`undefined` 是值的一种。undeclared 则表示变量还没有被声明过。

遗憾的是，JavaScript 却将它们混为一谈，在我们试图访问 `"undeclared"` 变量时这样报错：ReferenceError: a is not defined，并且 `typeof` 对 undefined 和 undeclared 变量都返回 `"undefined"`。

然而，通过 `typeof` 的安全防范机制（阻止报错）来检查 undeclared 变量，有时是个不错的办法。