apply(..) 是一个工具函数，适用于所有函数对象，它会以一种特殊的方式来调用传递给它的函数。

第一个参数是 this 对象（《你不知道的 JavaScript（上卷）》的“this 和对象原型”部分中有相关介绍），这里不用太过费心，暂将它设为 null。第二个参数则必须是一个数组（或者类似数组的值，也叫作类数组对象，array-like object），其中的值被用作函数的参数。

于是 Array.apply(..) 调用 Array(..) 函数，并且将 { length: 3 } 作为函数的参数。

我们可以设想 apply(..) 内部有一个 for 循环（与上述 join(..) 类似），从 0 开始循环到 length（即循环到 2，不包括 3）。

假设在 apply(..) 内部该数组参数名为 arr，for 循环就会这样来遍历数组：arr[0]、arr[1]、arr[2]。然而，由于 { length: 3 } 中并不存在这些属性，所以返回值为 undefined。

换句话说，我们执行的实际上是 Array(undefined, undefined, undefined)，所以结果是单元值为 undefined 的数组，而非空单元数组。

虽然 Array.apply(null, { length: 3 }) 在创建 undefined 值的数组时有些奇怪和繁琐，但是其结果远比 Array(3) 更准确可靠。

总之，永远不要创建和使用空单元数组。

3.4.2 Object(..)、Function(..) 和 RegExp(..)

同样，除非万不得已，否则尽量不要使用 Object(..)/Function(..)/RegExp(..)：

```
var c = new Object();
c.foo = "bar";
c; // { foo: "bar" }

var d = { foo: "bar" };
d; // { foo: "bar" }

var e = new Function( "a", "return a * 2;" );
var f = function(a) { return a * 2; }
function g(a) { return a * 2; }

var h = new RegExp( "^a*b+", "g" );
var i = /^a*b+/g;
```

在实际情况中没有必要使用 new Object() 来创建对象，因为这样就无法像常量形式那样一次设定多个属性，而必须逐一设定。

构造函数 Function 只在极少数情况下很有用，比如动态定义函数参数和函数体的时候。不

要把 Function(..) 当作 eval(..) 的替代品，你基本上不会通过这种方式来定义函数。

强烈建议使用常量形式（如 /^a*b+/g）来定义正则表达式，这样不仅语法简单，执行效率也更高，因为 JavaScript 引擎在代码执行前会对它们进行预编译和缓存。与前面的构造函数不同，RegExp(..) 有时还是很有用的，比如动态定义正则表达式时：

```
var name = "Kyle";
var namePattern = new RegExp( "\\b(?:" + name + ")+\\b", "ig" );

var matches = someText.match( namePattern );
```

上述情况在 JavaScript 编程中时有发生，这时 new RegExp("pattern","flags") 就能派上用场。

3.4.3 Date(..) 和 Error(..)

相较于其他原生构造函数，Date(..) 和 Error(..) 的用处要大很多，因为没有对应的常量形式来作为它们的替代。

创建日期对象必须使用 new Date()。Date(..) 可以带参数，用来指定日期和时间，而不带参数的话则使用当前的日期和时间。

Date(..) 主要用来获得当前的 Unix 时间戳（从 1970 年 1 月 1 日开始计算，以秒为单位）。该值可以通过日期对象中的 getTime() 来获得。

从 ES5 开始引入了一个更简单的方法，即静态函数 Date.now()。对 ES5 之前的版本我们可以使用下面的 polyfill：

```
if (!Date.now) {
    Date.now = function(){
        return (new Date()).getTime();
    };
}
```

如果调用 Date() 时不带 new 关键字，则会得到当前日期的字符串值。其具体格式规范没有规定，浏览器使用 "Fri Jul 18 2014 00:31:02 GMT-0500 (CDT)" 这样的格式来显示。

构造函数 Error(..)（与前面的 Array() 类似）带不带 new 关键字都可。

创建错误对象（error object）主要是为了获得当前运行栈的上下文（大部分 JavaScript 引擎通过只读属性 .stack 来访问）。栈上下文信息包括函数调用栈信息和产生错误的代码行号，以便于调试（debug）。

错误对象通常与 throw 一起使用：

```
function foo(x) {
    if (!x) {
        throw new Error( "x wasn't provided" );
    }
    // ..
}
```

通常错误对象至少包含一个 message 属性，有时也不乏其他属性（必须作为只读属性访问），如 type。除了访问 stack 属性以外，最好的办法是调用（显式调用或者通过强制类型转换隐式调用，参见第 4 章）toString() 来获得经过格式化的便于阅读的错误信息。

除 Error(..) 之外，还有一些针对特定错误类型的原生构造函数，如 EvalError(..)、RangeError(..)、ReferenceError(..)、SyntaxError(..)、TypeError(..) 和 URIError(..)。这些构造函数很少被直接使用，它们在程序发生异常（比如试图使用未声明的变量产生 ReferenceError 错误）时会被自动调用。

3.4.4 Symbol(..)

ES6 中新加入了一个基本数据类型 ——符号（Symbol）。符号是具有唯一性的特殊值（并非绝对），用它来命名对象属性不容易导致重名。该类型的引入主要源于 ES6 的一些特殊构造，此外符号也可以自行定义。

符号可以用作属性名，但无论是在代码还是开发控制台中都无法查看和访问它的值，只会显示为诸如 Symbol(Symbol.create) 这样的值。

ES6 中有一些预定义符号，以 Symbol 的静态属性形式出现，如 Symbol.create、Symbol.iterator 等，可以这样来使用：

```
obj[Symbol.iterator] = function(){ /*..*/ };
```

我们可以使用 Symbol(..) 原生构造函数来自定义符号。但它比较特殊，不能带 new 关键字，否则会出错：

```
var mysym = Symbol( "my own symbol" );
mysym;              // Symbol(my own symbol)
mysym.toString();   // "Symbol(my own symbol)"
typeof mysym;       // "symbol"

var a = { };
a[mysym] = "foobar";

Object.getOwnPropertySymbols( a );
// [ Symbol(my own symbol) ]
```

虽然符号实际上并非私有属性（通过 Object.getOwnPropertySymbols(..) 便可以公开获得对象中的所有符号），但它却主要用于私有或特殊属性。很多开发人员喜欢用它来替代有下划线（_）前缀的属性，而下划线前缀通常用于命名私有或特殊属性。

符号并非对象，而是一种简单标量基本类型。

3.4.5 原生原型

原生构造函数有自己的 .prototype 对象，如 Array.prototype、String.prototype 等。

这些对象包含其对应子类型所特有的行为特征。

例如，将字符串值封装为字符串对象之后，就能访问 String.prototype 中定义的方法。

根据文档约定，我们将 String.prototype.XYZ 简写为 String#XYZ，对其他 .prototypes 也同样如此。

- String#indexOf(..)
  在字符串中找到指定子字符串的位置。
- String#charAt(..)
  获得字符串指定位置上的字符。
- String#substr(..)、String#substring(..) 和 String#slice(..)
  获得字符串的指定部分。
- String#toUpperCase() 和 String#toLowerCase()
  将字符串转换为大写或小写。
- String#trim()
  去掉字符串前后的空格，返回新的字符串。

以上方法并不改变原字符串的值，而是返回一个新字符串。

借助原型代理（prototype delegation，参见《你不知道的 JavaScript（上卷）》的“this 和对象原型”部分），所有字符串都可以访问这些方法：

```
var a = " abc ";

a.indexOf( "c" ); // 3
a.toUpperCase();  // " ABC "
a.trim();         // "abc"
```

其他构造函数的原型包含它们各自类型所特有的行为特征，比如 Number#tofixed(..)（将数字转换为指定长度的整数字符串）和 Array#concat(..)（合并数组）。所有的函数都可以调用 Function.prototype 中的 apply(..)、call(..) 和 bind(..)。

然而，有些原生原型（native prototype）并非普通对象那么简单：

```
typeof Function.prototype;          // "function"
Function.prototype();               // 空函数!

RegExp.prototype.toString();        // "/(?:)/"——空正则表达式
"abc".match( RegExp.prototype );    // [""]
```

更糟糕的是，我们甚至可以修改它们（而不仅仅是添加属性）：

```
Array.isArray( Array.prototype );   // true
Array.prototype.push( 1, 2, 3 );    // 3
Array.prototype;                    // [1,2,3]

// 需要将Array.prototype设置回空,否则会导致问题!
Array.prototype.length = 0;
```

这里，Function.prototype 是一个函数，RegExp.prototype 是一个正则表达式，而 Array.prototype 是一个数组。是不是很有意思？

将原型作为默认值

Function.prototype 是一个空函数，RegExp.prototype 是一个“空”的正则表达式（无任何匹配），而 Array.prototype 是一个空数组。对未赋值的变量来说，它们是很好的默认值。

例如：

```
function isThisCool(vals,fn,rx) {
    vals = vals || Array.prototype;
    fn = fn || Function.prototype;
    rx = rx || RegExp.prototype;

    return rx.test(
        vals.map( fn ).join( "" )
    );
}

isThisCool();       // true
```

```
isThisCool(
    ["a","b","c"],
    function(v){ return v.toUpperCase(); },
    /D/
);                  // false
```

从 ES6 开始，我们不再需要使用 `vals = vals || ..` 这样的方式来设置默认值（参见第 4 章），因为默认值可以通过函数声明中的内置语法来设置（参见第 5 章）。

这种方法的一个好处是 `.prototypes` 已被创建并且仅创建一次。相反，如果将 `[]`、`function(){}` 和 `/(?:)/` 作为默认值，则每次调用 `isThisCool(..)` 时它们都会被创建一次（具体创建与否取决于 JavaScript 引擎，稍后它们可能会被垃圾回收），这样无疑会造成内存和 CPU 资源的浪费。

另外需要注意的一点是，如果默认值随后会被更改，那就不要使用 `Array.prototype`。上例中的 `vals` 是作为只读变量来使用，更改 `vals` 实际上就是更改 `Array.prototype`，而这样会导致前面提到过的一系列问题！

以上我们介绍了原生原型及其用途，使用它们时要十分小心，特别是要对它们进行更改时。详情请见本部分附录 A 中的 A.4 节。

3.5 小结

JavaScript 为基本数据类型值提供了封装对象，称为原生函数（如 `String`、`Number`、`Boolean` 等）。它们为基本数据类型值提供了该子类型所特有的方法和属性（如：`String#trim()` 和 `Array#concat(..)`）。

对于简单标量基本类型值，比如 `"abc"`，如果要访问它的 `length` 属性或 `String.prototype` 方法，JavaScript 引擎会自动对该值进行封装（即用相应类型的封装对象来包装它）来实现对这些属性和方法的访问。

第 4 章

强制类型转换

在对 JavaScript 的类型和值有了更全面的了解之后，本章旨在讨论一个非常有争议的话题：强制类型转换。

如第 1 章所述，关于强制类型转换是一个设计上的缺陷还是有用的特性，这一争论从 JavaScript 诞生之日起就开始了。在很多的 JavaScript 书籍中强制类型转换被说成是危险、晦涩和糟糕的设计。

秉承本系列丛书的一贯宗旨，对于不懂的地方我们应该迎难而上，知其然并且知其所以然，不会因为种种传言和挫折就退避三舍。

本章旨在全面介绍强制类型转换的优缺点，让你能够在开发中合理地运用它。

4.1 值类型转换

将值从一种类型转换为另一种类型通常称为*类型转换*（type casting），这是显式的情况；隐式的情况称为*强制类型转换*（coercion）。

JavaScript 中的强制类型转换总是返回标量基本类型值（参见第 2 章），如字符串、数字和布尔值，不会返回对象和函数。在第 3 章中，我们介绍过“封装”，就是为标量基本类型值封装一个相应类型的对象，但这并非严格意义上的强制类型转换。

也可以这样来区分：类型转换发生在静态类型语言的编译阶段，而强制类型转换则发生在

动态类型语言的运行时（runtime）。

然而在 JavaScript 中通常将它们统称为*强制类型转换*，我个人则倾向于用“隐式强制类型转换”（implicit coercion）和“显式强制类型转换”（explicit coercion）来区分。

二者的区别显而易见：我们能够从代码中看出哪些地方是显式强制类型转换，而隐式强制类型转换则不那么明显，通常是某些操作产生的副作用。

例如：

```
var a = 42;

var b = a + "";          // 隐式强制类型转换

var c = String( a );     // 显式强制类型转换
```

对变量 b 而言，强制类型转换是隐式的；由于 + 运算符的其中一个操作数是字符串，所以是字符串拼接操作，结果是数字 42 被强制类型转换为相应的字符串 "42"。

而 String(..) 则是将 a 显式强制类型转换为字符串。

两者都是将数字 42 转换为字符串 "42"。然而它们各自不同的处理方式成为了争论的焦点。

从技术角度来说，除了字面上的差别以外，二者在行为特征上也有一些细微的差别。我们将在 4.4.2 节详细介绍。

这里的“显式”和“隐式”以及“明显的副作用”和“隐藏的副作用”，都是相对而言的。

要是你明白 a + "" 是怎么回事，它对你来说就是“显式”的。相反，如果你不知道 String(..) 可以用来做字符串强制类型转换，它对你来说可能就是“隐式”的。

我们在这里以普遍通行的标准来讨论“显式”和“隐式”，而非 JavaScript 专家和规范的标准。如果你的理解与此有出入，请参照我们的标准。

要知道我们编写的代码大都是给别人看的。即便是 JavaScript 高手也需要顾及其他不同水平的开发人员，要考虑他们是否能读懂自己的代码，以及他们对于“显式”和“隐式”的理解是否和自己一致。

4.2 抽象值操作

介绍显式和隐式强制类型转换之前，我们需要掌握字符串、数字和布尔值之间类型转换的

基本规则。ES5 规范第 9 节中定义了一些“抽象操作”（即“仅供内部使用的操作”）和转换规则。这里我们着重介绍 ToString、ToNumber 和 ToBoolean，附带讲一讲 ToPrimitive。

4.2.1 ToString

规范的 9.8 节中定义了抽象操作 ToString，它负责处理非字符串到字符串的强制类型转换。

基本类型值的字符串化规则为：null 转换为 "null"，undefined 转换为 "undefined"，true 转换为 "true"。数字的字符串化则遵循通用规则，不过第 2 章中讲过的那些极小和极大的数字使用指数形式：

```
// 1.07 连续乘以七个 1000
var a = 1.07 * 1000 * 1000 * 1000 * 1000 * 1000 * 1000 * 1000;

// 七个1000一共21位数字
a.toString(); // "1.07e21"
```

对普通对象来说，除非自行定义，否则 toString()（Object.prototype.toString()）返回内部属性 [[Class]] 的值（参见第 3 章），如 "[object Object]"。

然而前面我们介绍过，如果对象有自己的 toString() 方法，字符串化时就会调用该方法并使用其返回值。

将对象强制类型转换为 string 是通过 ToPrimitive 抽象操作来完成的（ES5 规范，9.1 节），我们在此略过，稍后将在 4.2.2 节中详细介绍。

数组的默认 toString() 方法经过了重新定义，将所有单元字符串化以后再用 "," 连接起来：

```
var a = [1,2,3];

a.toString(); // "1,2,3"
```

toString() 可以被显式调用，或者在需要字符串化时自动调用。

JSON 字符串化

工具函数 JSON.stringify(..) 在将 JSON 对象序列化为字符串时也用到了 ToString。

请注意，JSON 字符串化并非严格意义上的强制类型转换，因为其中也涉及 ToString 的相关规则，所以这里顺带介绍一下。

对大多数简单值来说，JSON 字符串化和 toString() 的效果基本相同，只不过序列化的结果总是字符串：

```
JSON.stringify( 42 );   // "42"
JSON.stringify( "42" ); // ""42"" （含有双引号的字符串）
JSON.stringify( null ); // "null"
JSON.stringify( true ); // "true"
```

所有安全的 JSON 值（JSON-safe）都可以使用 JSON.stringify(..) 字符串化。安全的 JSON 值是指能够呈现为有效 JSON 格式的值。

为了简单起见，我们来看看什么是不安全的 JSON 值。undefined、function、symbol（ES6+）和包含循环引用（对象之间相互引用，形成一个无限循环）的对象都不符合 JSON 结构标准，支持 JSON 的语言无法处理它们。

JSON.stringify(..) 在对象中遇到 undefined、function 和 symbol 时会自动将其忽略，在数组中则会返回 null（以保证单元位置不变）。

例如：

```
JSON.stringify( undefined );      // undefined
JSON.stringify( function(){} );   // undefined

JSON.stringify(
   [1,undefined,function(){},4]
);                                // "[1,null,null,4]"
JSON.stringify(
   { a:2, b:function(){} }
);                                // "{"a":2}"
```

对包含循环引用的对象执行 JSON.stringify(..) 会出错。

如果对象中定义了 toJSON() 方法，JSON 字符串化时会首先调用该方法，然后用它的返回值来进行序列化。

如果要对含有非法 JSON 值的对象做字符串化，或者对象中的某些值无法被序列化时，就需要定义 toJSON() 方法来返回一个安全的 JSON 值。

例如：

```
var o = { };

var a = {
    b: 42,
    c: o,
    d: function(){}
};
```

```
// 在a中创建一个循环引用
o.e = a;

// 循环引用在这里会产生错误
// JSON.stringify( a );

// 自定义的JSON序列化
a.toJSON = function() {
    // 序列化仅包含b
    return { b: this.b };
};

JSON.stringify( a ); // "{"b":42}"
```

很多人误以为 `toJSON()` 返回的是 JSON 字符串化后的值，其实不然，除非我们确实想要对字符串进行字符串化（通常不会！）。`toJSON()` 返回的应该是一个适当的值，可以是任何类型，然后再由 `JSON.stringify(..)` 对其进行字符串化。

也就是说，`toJSON()` 应该“返回一个能够被字符串化的安全的 JSON 值”，而不是“返回一个 JSON 字符串”。

例如：

```
var a = {
    val: [1,2,3],

    // 可能是我们想要的结果!
    toJSON: function(){
        return this.val.slice( 1 );
    }
};

var b = {
    val: [1,2,3],

    // 可能不是我们想要的结果!
    toJSON: function(){
        return "[" +
            this.val.slice( 1 ).join() +
        "]";
    }
};

JSON.stringify( a ); // "[2,3]"

JSON.stringify( b ); // ""[2,3]""
```

这里第二个函数是对 `toJSON` 返回的字符串做字符串化，而非数组本身。

现在介绍几个不太为人所知但却非常有用的功能。

我们可以向 JSON.stringify(..) 传递一个可选参数 replacer，它可以是数组或者函数，用来指定对象序列化过程中哪些属性应该被处理，哪些应该被排除，和 toJSON() 很像。

如果 replacer 是一个数组，那么它必须是一个字符串数组，其中包含序列化要处理的对象的属性名称，除此之外其他的属性则被忽略。

如果 replacer 是一个函数，它会对对象本身调用一次，然后对对象中的每个属性各调用一次，每次传递两个参数，键和值。如果要忽略某个键就返回 undefined，否则返回指定的值。

```
var a = {
    b: 42,
    c: "42",
    d: [1,2,3]
};

JSON.stringify( a, ["b","c"] ); // "{"b":42,"c":"42"}"

JSON.stringify( a, function(k,v){
    if (k !== "c") return v;
} );
// "{"b":42,"d":[1,2,3]}"
```

如果 replacer 是函数，它的参数 k 在第一次调用时为 undefined（就是对对象本身调用的那次）。if 语句将属性 "c" 排除掉。由于字符串化是递归的，因此数组 [1,2,3] 中的每个元素都会通过参数 v 传递给 replacer，即 1、2 和 3，参数 k 是它们的索引值，即 0、1 和 2。

JSON.string 还有一个可选参数 space，用来指定输出的缩进格式。space 为正整数时是指定每一级缩进的字符数，它还可以是字符串，此时最前面的十个字符被用于每一级的缩进：

```
var a = {
    b: 42,
    c: "42",
    d: [1,2,3]
};

JSON.stringify( a, null, 3 );
// "{
//    "b": 42,
//    "c": "42",
//    "d": [
//       1,
//       2,
//       3
//    ]
// }"
```

```
JSON.stringify( a, null, "-----" );
// "{
// -----"b": 42,
// -----"c": "42",
// -----"d": [
// ----------1,
// ----------2,
// ----------3
// -----]
// }"
```

请记住，JSON.stringify(..) 并不是强制类型转换。在这里介绍是因为它涉及 ToString 强制类型转换，具体表现在以下两点。

(1) 字符串、数字、布尔值和 null 的 JSON.stringify(..) 规则与 ToString 基本相同。
(2) 如果传递给 JSON.stringify(..) 的对象中定义了 toJSON() 方法，那么该方法会在字符串化前调用，以便将对象转换为安全的 JSON 值。

4.2.2 ToNumber

有时我们需要将非数字值当作数字来使用，比如数学运算。为此 ES5 规范在 9.3 节定义了抽象操作 ToNumber。

其中 true 转换为 1，false 转换为 0。undefined 转换为 NaN，null 转换为 0。

ToNumber 对字符串的处理基本遵循数字常量的相关规则 / 语法（参见第 3 章）。处理失败时返回 NaN（处理数字常量失败时会产生语法错误）。不同之处是 ToNumber 对以 0 开头的十六进制数并不按十六进制处理（而是按十进制，参见第 2 章）。

数字常量的语法规则与 ToNumber 处理字符串所遵循的规则之间差别不大，这里不做进一步介绍，可参考 ES5 规范的 9.3.1 节。

对象（包括数组）会首先被转换为相应的基本类型值，如果返回的是非数字的基本类型值，则再遵循以上规则将其强制转换为数字。

为了将值转换为相应的基本类型值，抽象操作 ToPrimitive（参见 ES5 规范 9.1 节）会首先（通过内部操作 DefaultValue，参见 ES5 规范 8.12.8 节）检查该值是否有 valueOf() 方法。如果有并且返回基本类型值，就使用该值进行强制类型转换。如果没有就使用 toString() 的返回值（如果存在）来进行强制类型转换。

如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误。

从 ES5 开始，使用 Object.create(null) 创建的对象 [[Prototype]] 属性为 null，并且没有 valueOf() 和 toString() 方法，因此无法进行强制类型转换。详情请参考本系列的《你不知道的 JavaScript（上卷）》“this 和对象原型”部分中 [[Prototype]] 相关部分。

我们稍后将详细介绍数字的强制类型转换，在下面的示例代码中我们假定 Number(..) 已经实现了此功能。

例如：

```
var a = {
    valueOf: function(){
        return "42";
    }
};

var b = {
    toString: function(){
        return "42";
    }
};

var c = [4,2];
c.toString = function(){
    return this.join( "" ); // "42"
};

Number( a );                // 42
Number( b );                // 42
Number( c );                // 42
Number( "" );               // 0
Number( [] );               // 0
Number( [ "abc" ] );        // NaN
```

4.2.3 ToBoolean

下面介绍布尔值，关于这个主题存在许多误解和困惑，需要我们特别注意。

首先也是最重要的一点是，JavaScript 中有两个关键词 true 和 false，分别代表布尔类型中的真和假。我们常误以为数值 1 和 0 分别等同于 true 和 false。在有些语言中可能是这样，但在 JavaScript 中布尔值和数字是不一样的。虽然我们可以将 1 强制类型转换为 true，将 0 强制类型转换为 false，反之亦然，但它们并不是一回事。

1. 假值（falsy value）

我们再来看看其他值是如何被强制类型转换为布尔值的。

JavaScript 中的值可以分为以下两类：

(1) 可以被强制类型转换为 `false` 的值
(2) 其他（被强制类型转换为 `true` 的值）

JavaScript 规范具体定义了一小撮可以被强制类型转换为 `false` 的值。

ES5 规范 9.2 节中定义了抽象操作 `ToBoolean`，列举了布尔强制类型转换所有可能出现的结果。

以下这些是假值：

- `undefined`
- `null`
- `false`
- `+0`、`-0` 和 `NaN`
- `""`

假值的布尔强制类型转换结果为 `false`。

从逻辑上说，假值列表以外的都应该是真值（truthy）。但 JavaScript 规范对此并没有明确定义，只是给出了一些示例，例如规定所有的对象都是真值，我们可以理解为*假值列表以外的值都是真值*。

2. 假值对象（falsy object）

这个标题似乎有点自相矛盾。前面讲过规范规定所有的对象都是真值，怎么还会有假值对象呢？

有人可能会以为假值对象就是包装了假值的封装对象（如 `""`、`0` 和 `false`，参见第 3 章），实际不然。

这只是规范开的一个小玩笑。

例如：

```
var a = new Boolean( false );
var b = new Number( 0 );
var c = new String( "" );
```

它们都是封装了假值的对象（参见第 3 章）。那它们究竟是 `true` 还是 `false` 呢？答案很简单：