据。名称中的“type(类型)”是指看待一组位序列的“视图”，本质上就是一个映射，比如是把这些位序列映射为 8 位有符号整型数组还是 16 位有符号整型数组，等等。

如何构建这样的位集合呢？这称为一个“buffer”，最直接的方法是通过 `ArrayBuffer(..)` 构造器来构造：

```
var buf = new ArrayBuffer( 32 );
buf.byteLength;                         // 32
```

现在 `buf` 就是一个二进制 buffer，长为 32 字节（256 位），预先初始化全部为 `0`。一个 buffer 本身除了查看它的 `byteLength` 属性外，并不真正支持任何其他交互。

有些 Web 平台功能使用或者返回数组 buffer，比如 `FileReader#readAsArrayBuffer(..)`、`XMLHttpRequest#send(..)` 和 `ImageData`(canvas data)。

而在这个数组 buffer 之上，可以放置一个“视图”，这个视图以类型数组的形式存在。考虑：

```
var arr = new Uint16Array( buf );
arr.length;                             // 16
```

`arr` 是在这个 256 位 `buf` 上映射的一个 16 位无符号整型的类型数组，也就是说你得到了 16 个元素。

5.1.1 大小端（Endianness）

理解下面这点很重要：`arr` 的映射是按照运行 JavaScript 的平台的大小端设置（大端或小端）进行的。如果二进制数据的构造是基于某个大小端配置，而解释平台的大小端配置正相反，那么这就成了一个问题。

大小端的意思是多字节数字（比如前面代码片段中创建的 16 位无符号整型）中的低字节（8 位）位于这个数字字节表示中的右侧还是左侧。

举个例子，设想一个十进制数字 `3085`，我们需要用 16 位来表示它。如果只是用一个 16 位数字容器，不管大小端配置如何，它会被表示为二进制 `0000110000001101`（十六进制 `0c0d`）。

但是如果用两个 8 位数组表示数字 `3085`，那么大小端设置就会明显影响它在内存中的存储表示：

- `0000110000001101` / `0c0d`（大端）
- `0000110100001100` / `0d0c` (小端）

如果接收到一个来自于小端系统的表示 `3085` 的位序列 `0000110100001100`，而在大端系统中

为其建立视图，得到的值将是 3340（十进制）或者 0d0c（十六进制）。

目前 Web 上最常用的是小端表示方式，但是肯定存在不采用这种方式的浏览器。了解一块二进制数据生产方和消费方的大小端属性是很重要的。

根据 MDN，这里有一个快速检测 JavaScript 大小端的方法：

```
var littleEndian = (function() {
    var buffer = new ArrayBuffer( 2 );
    new DataView( buffer ).setInt16( 0, 256, true );
    return new Int16Array( buffer )[0] === 256;
})();
```

littleEndian 的结果可能是 true 也可能是 false，对于多数浏览器来说，它应该会返回 true。这个测试方法使用了 DataView(..)，此方法比在 buffer 上建立的视图访问（getting/setting）位提供了更底层、更小粒度的控制方法。前面代码片段中 setInt16(..) 方法的第三个参数是用来通知 DataView 要使用哪种大小端配置来操作。

不要把数组 buffer 的底层二进制存储的大小端和给定数字在 JavaScript 程序中如何表示搞混。比如，(3085).toString(2) 返回 "110000001101"，加上前面 4 个隐去的 "0" 看起来似乎是大端表示。实际上，这个表示法是基于 16 位视图，而不是两个 8 位字节的表示。要确定 JavaScript 环境的大小端，最好的方法就是前面的 DataView 测试。

5.1.2 多视图

单个 buffer 可以关联多个视图，比如：

```
var buf = new ArrayBuffer( 2 );

var view8 = new Uint8Array( buf );
var view16 = new Uint16Array( buf );

view16[0] = 3085;
view8[0];                       // 13
view8[1];                       // 12

view8[0].toString( 16 );        // "d"
view8[1].toString( 16 );        // "c"

// 交换（就像大小端变换一样！）
var tmp = view8[0];
view8[0] = view8[1];
view8[1] = tmp;

view16[0];                    // 3340
```

这个带类数组构造器有多个签名变体。目前我们展示的情况都是传入已经存在的 buffer。然而这种形式还接受两个额外参数：byteOffset 和 length。换句话说，可以从非 0 的位置开始带类型数组视图，也可以不消耗整个 buffer 长度。

如果二进制数据的 buffer 包含的数据格式大小 / 位置不均匀，这种技术是非常有用的。

举例来说，考虑一个二进制 buffer，在它的起始位置有一个 2 字节数字（也就是“字”），接着是两个 1 字节数字，然后是一个 32 位浮点数。这展示了如何通过在同一个 buffer 建立多个视图，从不同的位置，以不同长度访问这个数据：

```
var first = new Uint16Array( buf, 0, 2 )[0],
    second = new Uint8Array( buf, 2, 1 )[0],
    third = new Uint8Array( buf, 3, 1 )[0],
    fourth = new Float32Array( buf, 4, 4 )[0];
```

5.1.3 带类数组构造器

除了前面一节中展示的 (buffer,[offset, [length]]) 形式，带类数组构造器还支持以下这些形式。

- [constructor\](length)：在一个新的长度为 length 字节的 buffer 上创建一个新的视图。
- [constructor\](typedArr)：创建一个新的视图和 buffer，把 typeArr 视图的内容复制进去。
- [constructor\](obj)：创建一个新的视图和 buffer，并在类数组或对象 obj 上迭代复制其内容。

ES6 提供了下面这些带类数组构造器：

- Int8Array（8 位有符号整型），Uint8Array（8 位无符号整型）——Uint8ClampedArray（8 位无符号整型，每个值会被强制设置为在 0-255 内）；
- Int16Array（16 位有符号整型），Uint16Array（16 位无符号整型）；
- Int32Array（32 位有符号整型），Uint32Array（32 位无符号整型）；
- Float32Array（32 位浮点数，IEEE-754）；
- Float64Array（64 位浮点数，IEEE-754）。

带类数组构造器的实例几乎和普通原生数组完全一样。一些区别包括具有固定的长度以及值都属于某种“类型”。

然而，它们的 prototype 方法几乎完全一样。因此，很可能可以把它们当作普通数组使用而无需转换。

举例来说：

```
var a = new Int32Array( 3 );
a[0] = 10;
```

```
a[1] = 20;
a[2] = 30;

a.map( function(v){
    console.log( v );
} );
// 10 20 30

a.join( "-" );
// "10-20-30"
```

不能对 TypedArray 使用不合理的 `Array.prototype` 方法，比如修改器（`splice(..)`、`push(..)` 等）和 `concat(..)`。

要清楚 TypedArray 中的元素是限制在声明的位数大小中的。如果视图给一个 `Uint8Array` 的某个元素赋值为大于 8 位的值，这个值会被折回（wrap around）来适应其位宽。

这可能会引起问题，比如，如果你要把 TypedArray 中的值平方。考虑：

```
var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = a.map( function(v){
    return v * v;
} );

b;              // [100, 144, 132]
```

对于值 `20` 和 `30`，平方值就会溢出。要解决这样的局限，可以使用 `TypedArray#from(..)` 函数：

```
var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = Uint16Array.from( a, function(v){
      return v * v;
} );

b;              // [100, 400, 900]
```

关于与 TypedArray 共享的 `Array.from(..)` 方法的更多信息，参见 6.1.2 节，特别是“映射”那一小节解释了作为第二个参数的映射函数。

还有很有趣的一点要考虑，与普通数组一样，TypedArray 也有一个 `sort(..)` 方法，但是这个方法默认使用数字排序方法，而不是强制转换为字符串之后进行字母序比较 。举例来说：

```
var a = [ 10, 1, 2, ];
a.sort();                               // [1,10,2]

var b = new Uint8Array( [ 10, 1, 2 ] );
b.sort();                               // [1,2,10]
```

就像 `Array#sort(..)` 一样，`TypedArray#sort(..)` 接受了一个可选比较函数参数，它们的工作方式也是完全一样的。

5.2 Map

如果你的 JavaScript 经验丰富的话，应该会了解对象是创建无序键 / 值对数据结构 [也称为映射（map）] 的主要机制。但是，对象作为映射的主要缺点是不能使用非字符串值作为键。

举例来说，考虑：

```
var m = {};

var x = { id: 1 },
    y = { id: 2 };

m[x] = "foo";
m[y] = "bar";

m[x];               // "bar"
m[y];               // "bar"
```

这里发生了什么？ `x` 和 `y` 两个对象字符串化都是 `"[object Object]"`，所以 `m` 中只设置了一个键。

有些人通过维护平行的非字符串键数组和值数组来实现伪 map，比如：

```
var keys = [], vals = [];

var x = { id: 1 },
    y = { id: 2 };

keys.push( x );
vals.push( "foo" );

keys.push( y );
vals.push( "bar" );

keys[0] === x;              // true
vals[0];                    // "foo"

keys[1] === y;              // true
vals[1];                    // "bar"
```

当然，你不想自己维护这两个平行数组，所以可以定义一个数据结构，其中包含自动管理

低层的方法。除了必须要自己实现这些工作，这样做的主要缺点是访问的时间复杂度不再是 O(1)，而是 O(*n*)。

但在 ES6 中就不再需要这么做了！只需要使用 `Map(..)`:

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

m.get( x );                 // "foo"
m.get( y );                 // "bar"
```

这里唯一的缺点就是不能使用方括号 [ ] 语法设置和获取值，但完全可以使用 `get(..)` 和 `set(..)` 方法完美代替。

要从 map 中删除一个元素，不要使用 `delete` 运算符，而是要使用 `delete()` 方法：

```
m.set( x, "foo" );
m.set( y, "bar" );

m.delete( y );
```

你可以通过 `clear()` 清除整个 map 的内容。要得到 map 的长度（也就是键的个数），可以使用 `size` 属性（而不是 `length`）：

```
m.set( x, "foo" );
m.set( y, "bar" );
m.size;                         // 2

m.clear();
m.size;                         // 0
```

`Map(..)` 构造器也可以接受一个 iterable（参见 3.1 节），这个迭代器必须产生一列数组，每个数组的第一个元素是键，第二个元素是值。这种迭代的形式和 `entries()` 方法产生的形式是完全一样的，下一小节将会介绍。这使得创建一个 map 的副本很容易：

```
var m2 = new Map( m.entries() );

// 等价于：
var m2 = new Map( m );
```

因为 map 的实例是一个 iterable，它的默认迭代器与 `entries()` 相同，所以我们更推荐使用后面这个简短的形式。

当然，也可以在 `Map(..)` 构造器中手动指定一个**项目**（entry）列表（键 / 值数组的数组）：

```
var x = { id: 1 },
    y = { id: 2 };

var m = new Map( [
    [ x, "foo" ],
    [ y, "bar" ]
] );

m.get( x );                     // "foo"
m.get( y );                     // "bar"
```

5.2.1 Map 值

要从 map 中得到一列值，可以使用 `values(..)`，它会返回一个迭代器。在第 2 章和第 3 章中，我们介绍了几种顺序处理迭代器（就像数组）的方法，比如 spread 运算符 `...` 和 `for..of` 循环。另外，在 6.1.3 节我们将会详细介绍 `Array.from(..)` 方法。考虑：

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var vals = [ ...m.values() ];

vals;                           // ["foo","bar"]
Array.from( m.values() );       // ["foo","bar"]
```

前面一小节介绍过，可以在一个 map 的项目上使用 `entries()` 迭代（或者默认 map 迭代器）。考虑：

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var vals = [ ...m.entries() ];

vals[0][0] === x;               // true
vals[0][1];                     // "foo"

vals[1][0] === y;               // true
vals[1][1];                     // "bar"
```

5.2.2 Map 键

要得到一列键，可以使用 `keys()`，它会返回 map 中键上的迭代器：

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var keys = [ ...m.keys() ];

keys[0] === x;                // true
keys[1] === y;                // true
```

要确定一个 map 中是否有给定的键，可以使用 `has(..)` 方法：

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );

m.has( x );                    // true
m.has( y );                    // false
```

map 的本质是允许你把某些额外的信息（值）关联到一个对象（键）上，而无需把这个信息放入对象本身。

对于 map 来说，尽管可以使用任意类型的值作为键，但通常我们会使用对象，因为字符串或者其他基本类型已经可以作为普通对象的键使用。换句话说，除非某些或者全部键需要是对象，否则可以继续使用普通对象作为影射，这种情况下 map 才更加合适。

如果使用对象作为映射的键，这个对象后来被丢弃（所有的引用解除），试图让垃圾回收（GC）回收其内存，那么 map 本身仍会保持其项目。你需要从 map 中移除这个项目来支持 GC。在下一小节中，我们将会介绍作为对象键和 GC 的更好选择——WeakMap。

5.3 WeakMap

WeakMap 是 map 的变体，二者的多数外部行为特性都是一样的，区别在于内部内存分配（特别是其 GC）的工作方式。

WeakMap（只）接受对象作为键。这些对象是被**弱**持有的，也就是说如果对象本身被垃圾回收的话，在 WeakMap 中的这个项目也会被移除。然而我们无法观测到这一点，因为对象被垃圾回收的唯一方式是没有对它的引用了。但是一旦不再有引用，你也就没有对象引用来查看它是否还存在于这个 WeakMap 中了。

除此之外，WeakMap 的 API 是类似的，尽管要更少一些：

```
var m = new WeakMap();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );

m.has( x );                // true
m.has( y );                // false
```

WeakMap 没有 `size` 属性或 `clear()` 方法，也不会暴露任何键、值或项目上的迭代器。所以即使你解除了对 `x` 的引用，它将会因 GC 时这个条目被从 `m` 中移除，也没有办法确定这一事实。所以你就相信 JavaScript 所声明的吧！

和 Map 一样，通过 WeakMap 可以把信息与一个对象软关联起来。而在对这个对象没有完全控制权的时候，这个功能特别有用，比如 DOM 元素。如果作为映射键的对象可以被删除，并支持垃圾回收，那么 WeakMap 就更是合适的选择了。

需要注意的是，WeakMap 只是弱持有它的键，而不是值。考虑：

```
var m = new WeakMap();

var x = { id: 1 },
    y = { id: 2 },
    z = { id: 3 },
    w = { id: 4 };

m.set( x, y );

x = null;                  // { id: 1 } 可GC
y = null;                  // { id: 2 } 可GC
                           // 只因 { id: 1 } 可GC

m.set( z, w );

w = null;                  // { id: 4 } 不可GC
```

因此，我认为 WeakMap 更应该叫作“Weak-KeyMap”。

5.4 Set

set 是一个值的集合，其中的值唯一（重复会被忽略）。

set 的 API 和 map 类似。只是 `add(..)` 方法代替了 `set(..)` 方法（某种程度上说有点讽刺），没有 `get(..)` 方法。

考虑：

```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );
s.add( y );
s.add( x );

s.size;                         // 2

s.delete( y );
s.size;                         // 1

s.clear();
s.size;                         // 0
```

`Set(..)` 构造器形式和 `Map(..)` 类似，都可以接受一个 iterable，比如另外一个 set 或者仅仅是一个值的数组。但是，和 `Map(..)` 接受项目（entry）列表（键 / 值数组的数组）不同，`Set(..)` 接受的是值（value）列表（值的数组）：

```
var x = { id: 1 },
    y = { id: 2 };

var s = new Set( [x,y] );
```

set 不需要 `get(..)` 是因为不会从集合中取一个值，而是使用 `has(..)` 测试一个值是否存在：

```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );

s.has( x );           // true
s.has( y );           // false
```

除了会把 `-0` 和 `0` 当作是一样的而不加区别之外，`has(..)` 中的比较算法和 `Object.is(..)` 几乎一样（参见第 6 章）。

Set 迭代器

set 的迭代器方法和 map 一样。对于 set 来说，二者行为特性不同，但它和 map 迭代器的行为是对称的。考虑：