ArrayBuffer

□□

- ArrayBuffer □□
  - □□
  - ArrayBuffer.prototype.byteLength
  - ArrayBuffer.prototype.slice()
  - ArrayBuffer.isView()
- TypedArray □□
  - □□
  - □□□□
  - □□□□
  - □□□
  - BYTES_PER_ELEMENT □□
  - ArrayBuffer □□□□□□□□
  - □□
  - TypedArray.prototype.buffer
  - TypedArray.prototype.byteLength、TypedArray.prototype.byteOffset
  - TypedArray.prototype.length
  - TypedArray.prototype.set()
  - TypedArray.prototype.subarray()
  - TypedArray.prototype.slice()
  - TypedArray.of()
  - TypedArray.from()
- □□□□
- DataView □□
- □□□□□□□
  - AJAX
  - Canvas
  - WebSocket
  - Fetch API

- File API
- SharedArrayBuffer
- Atomics 对象

`ArrayBuffer` 对象、`TypedArray` 视图和 `DataView` 视图是 JavaScript 操作二进制数据的一个接口。这些对象早就存在，属于独立的规格（2011 年 2 月发布），ES6 将它们纳入了 ECMAScript 规格，并且增加了新的方法。它们都是以数组的语法处理二进制数据，所以统称为二进制数组。

这个接口的原始设计目的，与 WebGL 项目有关。所谓 WebGL，就是指浏览器与显卡之间的通信接口，为了满足 JavaScript 与显卡之间大量的、实时的数据交换，它们之间的数据通信必须是二进制的，而不能是传统的文本格式。文本格式传递一个 32 位整数，两端的 JavaScript 脚本与显卡都要进行格式转化，将非常耗时。这时要是存在一种机制，可以像 C 语言那样，直接操作字节，将 4 个字节的 32 位整数，以二进制形式原封不动地送入显卡，脚本的性能就会大幅提升。

二进制数组就是在这种背景下诞生的。它很像 C 语言的数组，允许开发者以数组下标的形式，直接操作内存，大大增强了 JavaScript 处理二进制数据的能力，使得开发者有可能通过 JavaScript 与操作系统的原生接口进行二进制通信。

二进制数组由三类对象组成。

（1）**`ArrayBuffer`** 对象：代表内存之中的一段二进制数据，可以通过“视图”进行操作。“视图”部署了数组接口，这意味着，可以用数组的方法操作内存。

（2）**`TypedArray`** 视图：共包括 9 种类型的视图，比如 `Uint8Array`（无符号 8 位整数）数组视图，`Int16Array`（16 位整数）数组视图，`Float32Array`（32 位浮点数）数组视图等等。

（3）**`DataView`** 视图：可以自定义复合格式的视图，比如第一个字节是 Uint8（无符号 8 位整数）、第二、三个字节是 Int16（16 位整数）、第四个字节开始是 Float32（32 位浮点数）等等，此外还可以自定义字节序。

简单说，`ArrayBuffer` 对象代表原始的二进制数据，`TypedArray` 视图用来读写简单类型的二进制数据，`DataView` 视图用来读写复杂类型的二进制数据。

`TypedArray` 视图支持的数据类型一共有 12种。

数据类型	字节长度	含义	对应的 C 语言类型
Int8	1	8 位带符号整数	signed char
Uint8	1	8 位不带符号整数	unsigned char
Uint8C	1	8 位不带符号整数（自动过滤溢出）	unsigned char
Int16	2	16 位带符号整数	short

数据类型	字节长度	含义	对应的 C 语言类型
Uint16	2	16 位不带符号整数	unsigned short
Int32	4	32 位带符号整数	int
Uint32	4	32 位不带符号的整数	unsigned int
BigInt64	8	64 位有符号整数	
BigUint64	8	64 位无符号整数	
Float16	2	16 位浮点数	
Float32	4	32 位浮点数	float
Float64	8	64 位浮点数	double

注意，二进制数组并不是真正的数组，而是类似数组的对象。

很多浏览器操作的 API，用到了二进制数组操作二进制数据，下面是其中的几个。

- Canvas
- Fetch API
- File API
- WebSockets
- XMLHttpRequest

ArrayBuffer 对象

概述

`ArrayBuffer`对象代表储存二进制数据的一段内存，它不能直接读写，只能通过视图（`TypedArray`视图和`DataView`视图）来读写，视图的作用是以指定格式解读二进制数据。

`ArrayBuffer`也是一个构造函数，可以分配一段可以存放数据的连续内存区域。

```
const buf = new ArrayBuffer(32);
```

上面代码生成了一段 32 字节的内存区域，每个字节的值默认都是 0。可以看到，`ArrayBuffer`构造函数的参数是所需要的内存大小（单位字节）。

为了读写这段内容，需要为它指定视图。`DataView`视图的创建，需要提供`ArrayBuffer`对象实例作为参数。

```
const buf = new ArrayBuffer(32);
const dataView = new DataView(buf);
dataView.getUint8(0) // 0
```

上面代码对一段 32 字节的内存，建立 `DataView` 视图，然后以不带符号的 8 位整数格式，从头读取 8 位二进制数据，结果得到 0，因为原始内存的 `ArrayBuffer` 对象，默认所有位都是 0。

另一种 `TypedArray` 视图，与 `DataView` 视图的一个区别是，它不是一个构造函数，而是一组构造函数，代表不同的数据格式。

```
const buffer = new ArrayBuffer(12);

const x1 = new Int32Array(buffer);
x1[0] = 1;
const x2 = new Uint8Array(buffer);
x2[0]  = 2;

x1[0] // 2
```

上面代码对同一段内存，分别建立两种视图：32 位带符号整数（ `Int32Array` 构造函数）和 8 位不带符号整数（ `Uint8Array` 构造函数）。由于两个视图对应的是同一段内存，一个视图修改底层内存，会影响到另一个视图。

`TypedArray` 视图的构造函数，除了接受 `ArrayBuffer` 实例作为参数，还可以接受普通数组作为参数，直接分配内存生成底层的 `ArrayBuffer` 实例，并同时完成对这段内存的赋值。

```
const typedArray = new Uint8Array([0,1,2]);
typedArray.length // 3

typedArray[0] = 5;
typedArray // [5, 1, 2]
```

上面代码使用 `TypedArray` 视图的 `Uint8Array` 构造函数，新建一个不带符号的 8 位整数视图。可以看到，`Uint8Array` 直接使用普通数组作为参数，对底层内存的赋值同时完成。

ArrayBuffer.prototype.byteLength

`ArrayBuffer` 实例的 `byteLength` 属性，返回所分配的内存区域的字节长度。

```
const buffer = new ArrayBuffer(32);
buffer.byteLength
// 32
```

如果要分配的内存区域很大，有可能分配失败（因为没有那么多的连续空余内存），所以有必要检查是否分配成功。

```
if (buffer.byteLength === n) {
  // 成功
} else {
  // 失败
}
```

ArrayBuffer.prototype.slice()

`ArrayBuffer`实例有一个`slice`方法，允许将内存区域的一部分，拷贝生成一个新的`ArrayBuffer`对象。

```
const buffer = new ArrayBuffer(8);
const newBuffer = buffer.slice(0, 3);
```

上面代码拷贝`buffer`对象的前 3 个字节（从 0 开始，到第 3 个字节前面结束），生成一个新的`ArrayBuffer`对象。`slice`方法其实包含两步，第一步是先分配一段新内存，第二步是将原来那个`ArrayBuffer`对象拷贝过去。

`slice`方法接受两个参数，第一个参数表示拷贝开始的字节序号（含该字节），第二个参数表示拷贝截止的字节序号（不含该字节）。如果省略第二个参数，则默认到原`ArrayBuffer`对象的结尾。

除了`slice`方法，`ArrayBuffer`对象不提供任何直接读写内存的方法，只允许在其上方建立视图，然后通过视图读写。

ArrayBuffer.isView()

`ArrayBuffer`有一个静态方法`isView`，返回一个布尔值，表示参数是否为`ArrayBuffer`的视图实例。这个方法大致相当于判断参数，是否为`TypedArray`实例或`DataView`实例。

```
const buffer = new ArrayBuffer(8);
ArrayBuffer.isView(buffer) // false

const v = new Int32Array(buffer);
```

```
ArrayBuffer.isView(v) // true
```

TypedArray 视图

概述

`ArrayBuffer` 对象作为内存区域，可以存放多种类型的数据。同一段内存，不同数据有不同的解读方式，这就叫做“视图”（view）。`ArrayBuffer` 有两种视图，一种是 `TypedArray` 视图，另一种是 `DataView` 视图。前者的数组成员都是同一个数据类型，后者的数组成员可以是不同的数据类型。

目前，`TypedArray` 视图一共包括 9 种类型，每一种视图都是一种构造函数。

- **`Int8Array`**：8 位有符号整数，长度 1 个字节。
- **`Uint8Array`**：8 位无符号整数，长度 1 个字节。
- **`Uint8ClampedArray`**：8 位无符号整数，长度 1 个字节，溢出处理不同。
- **`Int16Array`**：16 位有符号整数，长度 2 个字节。
- **`Uint16Array`**：16 位无符号整数，长度 2 个字节。
- **`Int32Array`**：32 位有符号整数，长度 4 个字节。
- **`Uint32Array`**：32 位无符号整数，长度 4 个字节。
- **`BigInt64Array`**：64 位有符号整数，长度 8 个字节。
- **`BigUint64Array`**：64 位无符号整数，长度 8 个字节。
- **`Float16Array`**：16 位浮点数，长度 2 个字节。
- **`Float32Array`**：32 位浮点数，长度 4 个字节。
- **`Float64Array`**：64 位浮点数，长度 8 个字节。

这 12个构造函数生成的数组，统称为 `TypedArray` 视图。它们很像普通数组，都有 `length` 属性，都能用方括号运算符（`[]`）获取单个元素，所有数组的方法，在它们上面都能使用。普通数组与 TypedArray 数组的差异主要在以下方面。

- TypedArray 数组的所有成员，都是同一种类型。
- TypedArray 数组的成员是连续的，不会有空位。
- TypedArray 数组成员的默认值为 0。比如，`new Array(10)` 返回一个普通数组，里面没有任何成员，只是 10 个空位；`new Uint8Array(10)` 返回一个 TypedArray 数组，里面 10 个成员都是 0。
- TypedArray 数组只是一层视图，本身不储存数据，它的数据都储存在底层的 `ArrayBuffer` 对象之中，要获取底层对象必须使用 `buffer` 属性。

构造函数

TypedArray 数组提供 12种构造函数，用来生成相应类型的数组实例。

构造函数有多种用法。

（1） TypedArray(buffer, byteOffset=0, length?)

同一个 `ArrayBuffer` 对象之上，可以根据不同的数据类型，建立多个视图。

```
// 创建一个8字节的ArrayBuffer
const b = new ArrayBuffer(8);

// 创建一个指向b的Int32视图，开始于字节0，直到缓冲区的末尾
const v1 = new Int32Array(b);

// 创建一个指向b的Uint8视图，开始于字节2，直到缓冲区的末尾
const v2 = new Uint8Array(b, 2);

// 创建一个指向b的Int16视图，开始于字节2，长度为2
const v3 = new Int16Array(b, 2, 2);
```

上面代码在一段长度为 8 个字节的内存（ `b` ）之上，生成了三个视图： `v1` 、 `v2` 和 `v3` 。

视图的构造函数可以接受三个参数：

- 第一个参数（必需）：视图对应的底层 `ArrayBuffer` 对象。
- 第二个参数（可选）：视图开始的字节序号，默认从 0 开始。
- 第三个参数（可选）：视图包含的数据个数，默认直到本段内存区域结束。

因此， `v1` 、 `v2` 和 `v3` 是重叠的： `v1[0]` 是一个 32 位整数，指向字节 0 ～字节 3； `v2[0]` 是一个 8 位无符号整数，指向字节 2； `v3[0]` 是一个 16 位整数，指向字节 2 ～字节 3 。只要任何一个视图对内存有所修改，就会在另外两个视图上反应出来。

注意， `byteOffset` 必须与所要建立的数据类型一致，否则会报错。

```
const buffer = new ArrayBuffer(8);
const i16 = new Int16Array(buffer, 1);
// Uncaught RangeError: start offset of Int16Array should be a multiple of 2
```

上面代码中，新生成一个 8 个字节的 `ArrayBuffer` 对象，然后在这个对象的第一个字节，建立带符号的 16 位整数视图，结果报错。因为，带符号的 16 位整数需要两个字节，所以 `byteOffset` 参数必须能够被 2 整除。

如果想从任意字节开始解读 `ArrayBuffer` 对象，必须使用 `DataView` 视图，因为 `TypedArray` 视图只提供 9 种固定的解读格式。

（2） TypedArray(length)

视图还可以不通过 `ArrayBuffer` 对象，直接分配内存而生成。

```
const f64a = new Float64Array(8);
f64a[0] = 10;
f64a[1] = 20;
f64a[2] = f64a[0] + f64a[1];
```

上面代码生成一个 8 个成员的 `Float64Array` 数组（共 64 字节），然后依次对每个成员赋值。这时，视图构造函数的参数就是成员的个数。可以看到，视图数组的赋值操作与普通数组的操作毫无两样。

（3）TypedArray(typedArray)

TypedArray 数组的构造函数，可以接受另一个 `TypedArray` 实例作为参数。

```
const typedArray = new Int8Array(new Uint8Array(4));
```

上面代码中，`Int8Array` 构造函数接受一个 `Uint8Array` 实例作为参数。

注意，此时生成的新数组，只是复制了参数数组的值，对应的底层内存是不一样的。新数组会开辟一段新的内存储存数据，不会在原数组的内存之上建立视图。

```
const x = new Int8Array([1, 1]);
const y = new Int8Array(x);
x[0] // 1
y[0] // 1

x[0] = 2;
y[0] // 1
```

上面代码中，数组 `y` 是以数组 `x` 为模板而生成的，当 `x` 变动的时候，`y` 并没有变动。

如果想基于同一段内存，构造不同的视图，可以采用下面的写法。

```
const x = new Int8Array([1, 1]);
const y = new Int8Array(x.buffer);
x[0] // 1
y[0] // 1

x[0] = 2;
y[0] // 2
```

（4）TypedArray(arrayLikeObject)

□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `TypedArray` □□□

```
const typedArray = new Uint8Array([1, 2, 3, 4]);
```

□□□□□ `TypedArray` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□ 8 □□□□□□□ `TypedArray` □□□

TypedArray □□□□□□□□□□□□□□

```
const normalArray = [...typedArray];
// or
const normalArray = Array.from(typedArray);
// or
const normalArray = Array.prototype.slice.call(typedArray);
```

□□□□

□□□□□□□□□□□□□□□ TypedArray □□□□□□□

- `TypedArray.prototype.copyWithin(target, start[, end = this.length])`
- `TypedArray.prototype.entries()`
- `TypedArray.prototype.every(callbackfn, thisArg?)`
- `TypedArray.prototype.fill(value, start=0, end=this.length)`
- `TypedArray.prototype.filter(callbackfn, thisArg?)`
- `TypedArray.prototype.find(predicate, thisArg?)`
- `TypedArray.prototype.findIndex(predicate, thisArg?)`
- `TypedArray.prototype.forEach(callbackfn, thisArg?)`
- `TypedArray.prototype.indexOf(searchElement, fromIndex=0)`
- `TypedArray.prototype.join(separator)`
- `TypedArray.prototype.keys()`
- `TypedArray.prototype.lastIndexOf(searchElement, fromIndex?)`
- `TypedArray.prototype.map(callbackfn, thisArg?)`
- `TypedArray.prototype.reduce(callbackfn, initialValue?)`
- `TypedArray.prototype.reduceRight(callbackfn, initialValue?)`
- `TypedArray.prototype.reverse()`
- `TypedArray.prototype.slice(start=0, end=this.length)`
- `TypedArray.prototype.some(callbackfn, thisArg?)`
- `TypedArray.prototype.sort(comparefn)`
- `TypedArray.prototype.toLocaleString(reserved1?, reserved2?)`
- `TypedArray.prototype.toString()`
- `TypedArray.prototype.values()`

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□ TypedArray □□□□ `concat`□□□□□□□□□□ TypedArray □□□□□□□□□□□□□

```
function concatenate(resultConstructor, ...arrays) {
  let totalLength = 0;
  for (let arr of arrays) {
    totalLength += arr.length;
  }
  let result = new resultConstructor(totalLength);
  let offset = 0;
  for (let arr of arrays) {
    result.set(arr, offset);
    offset += arr.length;
  }
  return result;
}

concatenate(Uint8Array, Uint8Array.of(1, 2), Uint8Array.of(3, 4))
// Uint8Array [1, 2, 3, 4]
```

□□□ TypedArray □□□□□□□□□□□□□ Iterator □□□□□□□□□□□

```
let ui8 = Uint8Array.of(0, 1, 2);
for (let byte of ui8) {
  console.log(byte);
}
// 0
// 1
// 2
```

□□□

□□□□□□□□□□□□□□□□□□

```
const buffer = new ArrayBuffer(16);
const int32View = new Int32Array(buffer);

for (let i = 0; i < int32View.length; i++) {
  int32View[i] = i * 2;
}
```

上面代码生成一个 16 字节的 `ArrayBuffer` 对象，然后在它的基础上，建立了一个 32 位整数的视图。由于每个 32 位整数占据 4 个字节，所以一共可以写入 4 个整数，依次为 0，2，4，6。

如果在这段数据上接着建立一个 16 位整数的视图，则可以读出完全不一样的结果。

```
const int16View = new Int16Array(buffer);

for (let i = 0; i < int16View.length; i++) {
  console.log("Entry " + i + ": " + int16View[i]);
}
// Entry 0: 0
// Entry 1: 0
// Entry 2: 2
// Entry 3: 0
// Entry 4: 4
// Entry 5: 0
// Entry 6: 6
// Entry 7: 0
```

由于每个 16 位整数占据 2 个字节，所以整个 `ArrayBuffer` 对象现在分成 8 段。然后，由于 x86 体系的计算机都采用小端字节序（little endian），相对重要的字节排在后面的内存地址，相对不重要字节排在前面的内存地址，所以就得到了上面的结果。

比如，一个占据四个字节的 16 进制数 `0x12345678`，决定其大小的最重要的字节是“12”，最不重要的是“78”。小端字节序将最不重要的字节排在前面，储存顺序就是 `78563412`；大端字节序则完全相反，将最重要的字节排在前面，储存顺序就是 `12345678`。目前，所有个人电脑几乎都是小端字节序，所以 TypedArray 数组内部也采用小端字节序读写数据，或者更准确的说，按照本机操作系统设定的字节序读写数据。

这并不意味大端字节序不重要，事实上，很多网络设备和特定的操作系统采用的是大端字节序。这就带来一个严重的问题：如果一段数据是大端字节序，TypedArray 数组将无法正确解析，因为它只能处理小端字节序！为了解决这个问题，JavaScript 引入 `DataView` 对象，可以设定字节序，下文会详细介绍。

下面是另一个例子。

```
// 假定某段buffer包含如下字节 [0x02, 0x01, 0x03, 0x07]
const buffer = new ArrayBuffer(4);
const v1 = new Uint8Array(buffer);
v1[0] = 2;
v1[1] = 1;
v1[2] = 3;
v1[3] = 7;
```

```
const uInt16View = new Uint16Array(buffer);

// 计算机采用小端字节序
// 所以头两个字节等于258
if (uInt16View[0] === 258) {
  console.log('OK'); // "OK"
}

// 赋值运算
uInt16View[0] = 255;    // 字节变为[0xFF, 0x00, 0x03, 0x07]
uInt16View[0] = 0xff05; // 字节变为[0x05, 0xFF, 0x03, 0x07]
uInt16View[1] = 0x0210; // 字节变为[0x05, 0xFF, 0x10, 0x02]
```

下面的函数可以用来判断，当前视图是小端字节序，还是大端字节序。

```
const BIG_ENDIAN = Symbol('BIG_ENDIAN');
const LITTLE_ENDIAN = Symbol('LITTLE_ENDIAN');

function getPlatformEndianness() {
  let arr32 = Uint32Array.of(0x12345678);
  let arr8 = new Uint8Array(arr32.buffer);
  switch ((arr8[0]*0x1000000) + (arr8[1]*0x10000) + (arr8[2]*0x100) + (arr8[3])) {
    case 0x12345678:
      return BIG_ENDIAN;
    case 0x78563412:
      return LITTLE_ENDIAN;
    default:
      throw new Error('Unknown endianness');
  }
}
```

总之，与普通数组相比，TypedArray数组的最大优点就是可以直接操作内存，不需要数据类型转换，所以速度快得多。

BYTES_PER_ELEMENT 属性

每一种视图的构造函数，都有一个`BYTES_PER_ELEMENT`属性，表示这种数据类型占据的字节数。

```
Int8Array.BYTES_PER_ELEMENT // 1
Uint8Array.BYTES_PER_ELEMENT // 1
Uint8ClampedArray.BYTES_PER_ELEMENT // 1
Int16Array.BYTES_PER_ELEMENT // 2
Uint16Array.BYTES_PER_ELEMENT // 2
Int32Array.BYTES_PER_ELEMENT // 4
Uint32Array.BYTES_PER_ELEMENT // 4
Float32Array.BYTES_PER_ELEMENT // 4
Float64Array.BYTES_PER_ELEMENT // 8
```

□□□□□ `TypedArray` □□□□□□□□□□ `TypedArray.prototype.BYTES_PER_ELEMENT` □

ArrayBuffer □□□□□□□□

`ArrayBuffer` □□□□□□□□□□□□□□□ `TextEncoder` □ `TextDecoder`
□□□□□□□□□□□□□□□□□□□□□□ TypeScript □□□□□□□□□□□□

```
/**
 * Convert ArrayBuffer/TypedArray to String via TextDecoder
 *
 * @see https://developer.mozilla.org/en-US/docs/Web/API/TextDecoder
 */
function ab2str(
  input: ArrayBuffer | Uint8Array | Int8Array | Uint16Array | Int16Array | Uint32Array |
Int32Array,
  outputEncoding: string = 'utf8',
): string {
  const decoder = new TextDecoder(outputEncoding)
  return decoder.decode(input)
}

/**
 * Convert String to ArrayBuffer via TextEncoder
 *
 * @see https://developer.mozilla.org/zh-CN/docs/Web/API/TextEncoder
 */
function str2ab(input: string): ArrayBuffer {
  const view = str2Uint8Array(input)
  return view.buffer
```

```
}

/** Convert String to Uint8Array */
function str2Uint8Array(input: string): Uint8Array {
  const encoder = new TextEncoder()
  const view = encoder.encode(input)
  return view
}
```

上面代码中，`ab2str()`的第二个参数`outputEncoding`给出了输出编码的编码，一般保持默认值（`utf-8`），其他可选值参见官方文档或 Node.js 文档。

溢出

不同的视图类型，所能容纳的数值范围是确定的。超出这个范围，就会出现溢出。比如，8 位视图只能容纳一个 8 位的二进制值，如果放入一个 9 位的值，就会溢出。

TypedArray 数组的溢出处理规则，简单来说，就是抛弃溢出的位，然后按照视图类型进行解释。

```
const uint8 = new Uint8Array(1);

uint8[0] = 256;
uint8[0] // 0

uint8[0] = -1;
uint8[0] // 255
```

上面代码中，`uint8`是一个 8 位视图，而 256 的二进制形式是一个 9 位的值`100000000`，这时就会发生溢出。根据规则，只会保留后 8 位，即`00000000`。`uint8`视图的解释规则是无符号的 8 位整数，所以`00000000`就是`0`。

负数在计算机内部采用“2 的补码”表示，也就是说，将对应的正数值进行否运算，然后加`1`。比如，`-1`对应的正值是`1`，进行否运算以后，得到`11111110`，再加上`1`就是补码形式`11111111`。`uint8`按照无符号的 8 位整数解释`11111111`，返回结果就是`255`。

一个简单转换规则，可以这样表示。

- 正向溢出（overflow）：当输入值大于当前数据类型的最大值，结果等于当前数据类型的最小值加上余值，再减去 1。
- 负向溢出（underflow）：当输入值小于当前数据类型的最小值，结果等于当前数据类型的最大值减去余值的绝对值，再加上 1。

上面的“余值”就是模运算的结果，即 JavaScript 里面的 `%` 运算符的结果。

```
12 % 4 // 0
12 % 5 // 2
```

上面代码中，12 除以 4 是没有余值的，而除以 5 会得到余值 2。

请看下面的例子。

```
const int8 = new Int8Array(1);

int8[0] = 128;
int8[0] // -128

int8[0] = -129;
int8[0] // 127
```

上面例子中，`int8` 是一个带符号的 8 位整数视图，它的最大值是 127，最小值是 -128。输入值为 `128` 时，相当于正向溢出 `1`，根据“最小值加上余值（128 除以 127 的余值是 1），再减去 1”的规则，就会返回 `-128`；输入值为 `-129` 时，相当于负向溢出 `1`，根据“最大值减去余值的绝对值（-129 除以 -128 的余值的绝对值是 1），再加上 1”的规则，就会返回 `127`。

`Uint8ClampedArray` 视图的溢出规则，与上面的规则不同。它规定，凡是发生正向溢出，该值一律等于当前数据类型的最大值，即 255；如果发生负向溢出，该值一律等于当前数据类型的最小值，即 0。

```
const uint8c = new Uint8ClampedArray(1);

uint8c[0] = 256;
uint8c[0] // 255

uint8c[0] = -1;
uint8c[0] // 0
```

上面例子中，`uint8C` 是一个 `Uint8ClampedArray` 视图，正向溢出时都返回 255，负向溢出都返回 0。

TypedArray.prototype.buffer

`TypedArray` 实例的 `buffer` 属性，返回整段内存区域对应的 `ArrayBuffer` 对象。该属性为只读属性。

```
const a = new Float32Array(64);
const b = new Uint8Array(a.buffer);
```

上面代码的 `a` 视图对象和 `b` 视图对象，对应同一个 `ArrayBuffer` 对象，即同一段内存。

TypedArray.prototype.byteLength, TypedArray.prototype.byteOffset

`byteLength` 属性返回 TypedArray 数组占据的内存长度，单位为字节。 `byteOffset` 属性返回 TypedArray 数组从底层 `ArrayBuffer` 对象的哪个字节开始。这两个属性都是只读属性。

```
const b = new ArrayBuffer(8);

const v1 = new Int32Array(b);
const v2 = new Uint8Array(b, 2);
const v3 = new Int16Array(b, 2, 2);

v1.byteLength // 8
v2.byteLength // 6
v3.byteLength // 4

v1.byteOffset // 0
v2.byteOffset // 2
v3.byteOffset // 2
```

TypedArray.prototype.length

`length` 属性表示 `TypedArray` 数组含有多少个成员。注意将 `length` 属性和 `byteLength` 属性区分，前者是成员长度，后者是字节长度。

```
const a = new Int16Array(8);

a.length // 8
a.byteLength // 16
```

TypedArray.prototype.set()

TypedArray 数组的 `set` 方法用于复制数组（普通数组或 TypedArray 数组），也就是将一段内容完全复制到另一段内存。

```
const a = new Uint8Array(8);
const b = new Uint8Array(8);

b.set(a);
```

上面代码复制 `a` 数组的内容到 `b` 数组，它是整段内存的复制，比一个个拷贝成员的那种复制快得多。

`set` 方法还可以接受第二个参数，表示从 `b` 对象的哪一个成员开始复制 `a` 对象。

```
const a = new Uint16Array(8);
const b = new Uint16Array(10);

b.set(a, 2)
```

上面代码的 `b` 数组比 `a` 数组多两个成员，所以从 `b[2]` 开始复制。

TypedArray.prototype.subarray()

`subarray` 方法是对于 TypedArray 数组的一部分，再建立一个新的视图。

```
const a = new Uint16Array(8);
const b = a.subarray(2,3);

a.byteLength // 16
b.byteLength // 2
```

`subarray`
方法的第一个参数是起始的成员序号，第二个参数是结束的成员序号（不含该成员），如果省略则包含剩余的
全部成员。所以，上面代码的 `a.subarray(2,3)`，意味着 b 只包含 `a[2]` 一个成员，字节长度为 2。

TypedArray.prototype.slice()

TypeArray 实例的 `slice` 方法，可以返回一个指定位置的新的 `TypedArray` 实例。

```
let ui8 = Uint8Array.of(0, 1, 2);
ui8.slice(-1)
// Uint8Array [ 2 ]
```

上面代码中，`ui8` 是 8 位无符号整数数组视图的一个实例。它的 `slice`
方法可以从当前视图之中，返回一个新的视图实例。

`slice` -1
-2

TypedArray.of()

TypedArray `of` `TypedArray`

```
Float32Array.of(0.151, -8, 3.7)
// Float32Array [ 0.151, -8, 3.7 ]
```

TypedArray

```
// 
let tarr = new Uint8Array([1,2,3]);

// 
let tarr = Uint8Array.of(1,2,3);

// 
let tarr = new Uint8Array(3);
tarr[0] = 1;
tarr[1] = 2;
tarr[2] = 3;
```

TypedArray.from()

`from` `TypedArray`

```
Uint16Array.from([0, 1, 2])
// Uint16Array [ 0, 1, 2 ]
```

`TypedArray`

```
const ui16 = Uint16Array.from(Uint8Array.of(0, 1, 2));
ui16 instanceof Uint16Array // true
```

`from` `map`

```
Int8Array.of(127, 126, 125).map(x => 2 * x)
// Int8Array [ -2, -4, -6 ]

Int16Array.from(Int8Array.of(127, 126, 125), x => 2 * x)
// Int16Array [ 254, 252, 250 ]
```

上面的例子中，`from`方法没有发生溢出，这说明遍历不是针对原来的 8 位整数数组。也就是说，`from`会将第一个参数指定的 TypedArray 数组，拷贝到另一段内存之中，处理之后再将结果转成指定的数组格式。

复合视图

由于视图的构造函数可以指定起始位置和长度，所以在同一段内存之中，可以依次存放不同类型的数据，这叫做“复合视图”。

```
const buffer = new ArrayBuffer(24);

const idView = new Uint32Array(buffer, 0, 1);
const usernameView = new Uint8Array(buffer, 4, 16);
const amountDueView = new Float32Array(buffer, 20, 1);
```

上面代码将一个 24 字节长度的`ArrayBuffer`对象，分成三个部分：

- 字节 0 到字节 3：1 个 32 位无符号整数
- 字节 4 到字节 19：16 个 8 位整数
- 字节 20 到字节 23：1 个 32 位浮点数

这种数据结构可以用如下的 C 语言描述：

```
struct someStruct {
  unsigned long id;
  char username[16];
  float amountDue;
};
```

DataView 视图

如果一段数据包括多种类型（比如服务器传来的 HTTP 数据），这时除了建立`ArrayBuffer`对象的复合视图以外，还可以通过`DataView`视图进行操作。

DataView ArrayBuffer
TypedArray
DataView

DataView ArrayBuffer

```
new DataView(ArrayBuffer buffer [, ] [, ]);
```

```
const buffer = new ArrayBuffer(24);
const dv = new DataView(buffer);
```

DataView TypedArray

- DataView.prototype.buffer ArrayBuffer
- DataView.prototype.byteLength
- DataView.prototype.byteOffset ArrayBuffer

DataView 11

- **getInt8** 1 8
- **getUint8** 1 8
- **getInt16** 2 16
- **getUint16** 2 16
- **getInt32** 4 32
- **getUint32** 4 32
- **getBigInt64** 8 64
- **getBigUint64** 8 64
- **getFloat16** 2 16
- **getFloat32** 4 32
- **getFloat64** 8 64

get

```
const buffer = new ArrayBuffer(24);
const dv = new DataView(buffer);

// 1 8
const v1 = dv.getUint8(0);

// 2 16
const v2 = dv.getUint16(1);
```

```
// 从第4个字节读取一个16位无符号整数
const v3 = dv.getUint16(3);
```

上面代码读取了 `ArrayBuffer` 对象的前 5 个字节，其中有一个 8 位整数和两个十六位整数。

如果一次读取两个或两个以上字节，就必须明确数据的存储方式，到底是小端字节序还是大端字节序。默认情况下，`DataView` 的 `get` 方法使用大端字节序解读数据，如果需要使用小端字节序解读，必须在 `get` 方法的第二个参数指定 `true`。

```
// 小端字节序
const v1 = dv.getUint16(1, true);

// 大端字节序
const v2 = dv.getUint16(3, false);

// 大端字节序
const v3 = dv.getUint16(3);
```

DataView 视图提供 11 个方法写入内存。

- **`setInt8`**：写入 1 个字节的 8 位整数。
- **`setUint8`**：写入 1 个字节的 8 位无符号整数。
- **`setInt16`**：写入 2 个字节的 16 位整数。
- **`setUint16`**：写入 2 个字节的 16 位无符号整数。
- **`setInt32`**：写入 4 个字节的 32 位整数。
- **`setUint32`**：写入 4 个字节的 32 位无符号整数。
- **`setBigInt64`**：写入 8 个字节的 64 位整数。
- **`setBigUint64`**：写入 8 个字节的 64 位无符号整数。
- **`setFloat16`**：写入 2 个字节的 16 位浮点数。
- **`setFloat32`**：写入 4 个字节的 32 位浮点数。
- **`setFloat64`**：写入 8 个字节的 64 位浮点数。

这一系列 `set` 方法，接受两个参数，第一个参数是字节序号，表示从哪个字节开始写入，第二个参数为写入的数据。对于那些写入两个或两个以上字节的方法，需要指定第三个参数，`false` 或者 `undefined` 表示使用大端字节序写入，`true` 表示使用小端字节序写入。

```
// 在第1个字节，以大端字节序写入值为25的32位整数
dv.setInt32(0, 25, false);

// 在第5个字节，以大端字节序写入值为25的32位整数
dv.setInt32(4, 25);
```

```
// 在第9个字节，以小端字节序写入值为2.5的32位浮点数
dv.setFloat32(8, 2.5, true);
```

如果不确定正在使用的计算机的字节序，可以采用下面的判断方式。

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

如果返回 `true`，就是小端字节序；如果返回 `false`，就是大端字节序。

二进制数组的应用

大量的 Web API 用到了 `ArrayBuffer` 对象和它的视图对象。

AJAX

传统上，服务器通过 AJAX 操作只能返回文本数据，即 `responseType` 属性默认为 `text`。`XMLHttpRequest` 第二版 `XHR2` 允许服务器返回二进制数据，这时分成两种情况。如果明确知道返回的二进制数据类型，可以把返回类型（`responseType`）设为 `arraybuffer`；如果不知道，就设为 `blob`。

```
let xhr = new XMLHttpRequest();
xhr.open('GET', someUrl);
xhr.responseType = 'arraybuffer';

xhr.onload = function () {
  let arrayBuffer = xhr.response;
  // ···
};

xhr.send();
```

如果知道传回来的是 32 位整数，可以像下面这样处理。

```
xhr.onreadystatechange = function () {
  if (req.readyState === 4 ) {
    const arrayResponse = xhr.response;
    const dataView = new DataView(arrayResponse);
    const ints = new Uint32Array(dataView.byteLength / 4);

    xhrDiv.style.backgroundColor = "#00FF00";
    xhrDiv.innerText = "Array is " + ints.length + "uints long";
  }
}
```

Canvas

网页 `Canvas` 元素输出的二进制像素数据，就是 TypedArray 数组。

```
const canvas = document.getElementById('myCanvas');
const ctx = canvas.getContext('2d');

const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
const uint8ClampedArray = imageData.data;
```

需要注意的是，上面代码的 `uint8ClampedArray` 虽然是一个 TypedArray 数组，但是它的视图类型是一种针对 `Canvas` 元素的专有类型 `Uint8ClampedArray`。这个视图类型的特点，就是专门针对颜色，把每个字节解读为无符号的 8 位整数，即只能取值 0 ~ 255，而且发生运算的时候自动过滤高位溢出。这为图像处理带来了巨大的方便。

举例来说，如果把像素的颜色值设为 `Uint8Array` 类型，那么乘以一个 gamma 值的时候，就必须这样计算：

```
u8[i] = Math.min(255, Math.max(0, u8[i] * gamma));
```

因为 `Uint8Array` 类型对于大于 255 的运算结果（比如 `0xFF+1`），会自动变为 `0x00`，所以图像处理必须要像上面这样算。这样做很麻烦，而且影响性能。如果将颜色值设为 `Uint8ClampedArray` 类型，计算就简化许多。

```
pixels[i] *= gamma;
```

`Uint8ClampedArray` 类型确保将小于 0 的值设为 0，将大于 255 的值设为 255。注意，IE 10 不支持该类型。

WebSocket

`WebSocket` □□□□ `ArrayBuffer` □□□□□□□□□□□□

```
let socket = new WebSocket('ws://127.0.0.1:8081');
socket.binaryType = 'arraybuffer';

// Wait until socket is open
socket.addEventListener('open', function (event) {
  // Send binary data
  const typedArray = new Uint8Array(4);
  socket.send(typedArray.buffer);
});

// Receive binary data
socket.addEventListener('message', function (event) {
  const arrayBuffer = event.data;
  // ...
});
```

Fetch API

Fetch API □□□□□□□□ `ArrayBuffer` □□□

```
fetch(url)
.then(function(response){
  return response.arrayBuffer()
})
.then(function(arrayBuffer){
  // ...
});
```

File API

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `ArrayBuffer` □□□

```
const fileInput = document.getElementById('fileInput');
const file = fileInput.files[0];
const reader = new FileReader();
reader.readAsArrayBuffer(file);
reader.onload = function () {
```

```
  const arrayBuffer = reader.result;
  // ···
};
```

□□□□□ bmp □□□□□□□ `file`□□□□□□□ bmp □□□□□□□□□□□□□□□□

```
const reader = new FileReader();
reader.addEventListener("load", processimage, false);
reader.readAsArrayBuffer(file);
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `DataView`□□□□□□□□ `bitmap`
□□□□□□□□□□□□□□□□□□□□□□□□ `Canvas`□□□□□

```
function processimage(e) {
  const buffer = e.target.result;
  const datav = new DataView(buffer);
  const bitmap = {};
  // □□□□□□□□
}
```

□□□□□□□□□□□□□□ bmp □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
bitmap.fileheader = {};
bitmap.fileheader.bfType = datav.getUint16(0, true);
bitmap.fileheader.bfSize = datav.getUint32(2, true);
bitmap.fileheader.bfReserved1 = datav.getUint16(6, true);
bitmap.fileheader.bfReserved2 = datav.getUint16(8, true);
bitmap.fileheader.bfOffBits = datav.getUint32(10, true);
```

□□□□□□□□□□□□□□

```
bitmap.infoheader = {};
bitmap.infoheader.biSize = datav.getUint32(14, true);
bitmap.infoheader.biWidth = datav.getUint32(18, true);
bitmap.infoheader.biHeight = datav.getUint32(22, true);
bitmap.infoheader.biPlanes = datav.getUint16(26, true);
bitmap.infoheader.biBitCount = datav.getUint16(28, true);
bitmap.infoheader.biCompression = datav.getUint32(30, true);
bitmap.infoheader.biSizeImage = datav.getUint32(34, true);
bitmap.infoheader.biXPelsPerMeter = datav.getUint32(38, true);
bitmap.infoheader.biYPelsPerMeter = datav.getUint32(42, true);
```

```
bitmap.infoheader.biClrUsed = datav.getUint32(46, true);
bitmap.infoheader.biClrImportant = datav.getUint32(50, true);
```

最后处理图像本身的像素信息。

```
const start = bitmap.fileheader.bfOffBits;
bitmap.pixels = new Uint8Array(buffer, start);
```

至此，图像文件的数据全部处理完成。下一步，可以根据需要，进行图像变形，或者转换格式，或者展示在`Canvas`网页元素之中。

SharedArrayBuffer

JavaScript 是单线程的，Web worker 引入了多线程：主线程用来与用户互动，Worker 线程用来承担计算任务。每个线程的数据都是隔离的，通过`postMessage()`通信。下面是一个例子。

```
// 主线程
const w = new Worker('myworker.js');
```

上面代码中，主线程新建了一个 Worker 线程。该线程与主线程之间会有一个通信渠道，主线程通过`w.postMessage`向 Worker 线程发消息，同时通过`message`事件监听 Worker 线程的回应。

```
// 主线程
w.postMessage('hi');
w.onmessage = function (ev) {
  console.log(ev.data);
}
```

上面代码中，主线程先发一个消息`hi`，然后在监听到 Worker 线程的回应后，就将其打印出来。

Worker 线程也是通过监听`message`事件，来获取主线程发来的消息，并作出反应。

```
// Worker 线程
onmessage = function (ev) {
  console.log(ev.data);
  postMessage('ho');
}
```

线程之间的数据交换可以是各种格式，不仅仅是字符串，也可以是二进制数据。这种交换采用的是复制机制，即一个进程将需要分享的数据复制一份，通过`postMessage`方法交给另一个进程。如果数据量比较大，这种通信的效率显然比较低。很容易想到，这时可以留出一块内存

区域，由主线程与 Worker 线程共享，两方都可以读写，那么就会大大提高效率，协作起来也会比较简单（不像 `postMessage` 那么麻烦）。

ES2017 引入 `SharedArrayBuffer`，允许 Worker 线程与主线程共享同一块内存。`SharedArrayBuffer` 的 API 与 `ArrayBuffer` 一模一样，唯一的区别是后者无法共享数据。

```
// 主线程

// 新建 1KB 共享内存
const sharedBuffer = new SharedArrayBuffer(1024);

// 主线程将共享内存的地址发送出去
w.postMessage(sharedBuffer);

// 在共享内存上建立视图，供写入数据
const sharedArray = new Int32Array(sharedBuffer);
```

上面代码中，`postMessage` 方法的参数是 `SharedArrayBuffer` 对象。

Worker 线程从事件的 `data` 属性上面取到数据。

```
// Worker 线程
onmessage = function (ev) {
  // 主线程共享的数据，就是 1KB 的共享内存
  const sharedBuffer = ev.data;

  // 在共享内存上建立视图，方便读写
  const sharedArray = new Int32Array(sharedBuffer);

  // ...
};
```

共享内存也可以在 Worker 线程创建，发给主线程。

`SharedArrayBuffer` 与 `ArrayBuffer` 一样，本身是无法读写的，必须在上面建立视图，然后通过视图读写。

```
// 分配 10 万个 32 位整数占据的内存空间
const sab = new SharedArrayBuffer(Int32Array.BYTES_PER_ELEMENT * 100000);

// 建立 32 位整数视图
const ia = new Int32Array(sab);  // ia.length == 100000
```

```
// 新建一个质数生成器
const primes = new PrimeGenerator();

// 将 10 万个质数，写入这段内存空间
for ( let i=0 ; i < ia.length ; i++ )
  ia[i] = primes.next();

// 向 Worker 线程发送这段共享内存
w.postMessage(ia);
```

Worker 线程收到数据后的处理如下。

```
// Worker 线程
let ia;
onmessage = function (ev) {
  ia = ev.data;
  console.log(ia.length); // 100000
  console.log(ia[37]); // 输出 163，因为这是第38个质数
};
```

Atomics 对象

多线程共享内存，最大的问题就是如何防止两个线程同时修改某个地址，或者说，当一个线程修改共享内存以后，必须有一个机制让其他线程同步。SharedArrayBuffer API 提供 `Atomics` 对象，保证所有共享内存的操作都是“原子性”的，并且可以在所有线程内同步。

什么叫“原子性操作”呢？现代编程语言中，一条普通的命令被编译器处理以后，会变成多条机器指令。如果是单线程运行，这是没有问题的；多线程环境并且共享内存时，就会出问题，因为这一组机器指令的运行期间，可能会插入其他线程的指令，从而导致运行结果出错。请看下面的例子。

```
// 主线程
ia[42] = 314159;  // 原先的值 191
ia[37] = 123456;  // 原先的值 163

// Worker 线程
console.log(ia[37]);
console.log(ia[42]);
// 可能的结果
```

```
// 123456
// 191
```

上面代码中，主线程的原始顺序是先对 42 号位置赋值，再对 37 号位置赋值。但是，编译器和 CPU 为了优化，可能会改变这两个操作的执行顺序（因为它们之间互不依赖），先对 37 号位置赋值，再对 42 号位置赋值。而执行到一半的时候，Worker 线程可能就会来读取数据，导致打印出 `123456` 和 `191`。

下面是另一个例子。

```
// 主线程
const sab = new SharedArrayBuffer(Int32Array.BYTES_PER_ELEMENT * 100000);
const ia = new Int32Array(sab);

for (let i = 0; i < ia.length; i++) {
  ia[i] = primes.next(); // 将质数放入 ia
}

// worker 线程
ia[112]++; // 错误
Atomics.add(ia, 112, 1); // 正确
```

上面代码中，Worker 线程直接改写共享内存 `ia[112]++` 是不正确的。因为这行语句会被编译成多条机器指令，这些指令之间无法保证不会插入其他进程的指令。请设想如果两个线程同时 `ia[112]++`，很可能它们得到的结果都是不正确的。

`Atomics` 对象就是为了解决这个问题而提出，它可以保证一个操作所对应的多条机器指令，一定是作为一个整体运行的，中间不会被打断。也就是说，它所涉及的操作都可以看作是原子性的单操作，这可以避免线程竞争，提高多线程共享内存时的操作安全。所以，`ia[112]++` 要改写成 `Atomics.add(ia, 112, 1)`。

`Atomics` 对象提供多种方法。

（1）Atomics.store()，Atomics.load()

`store()` 方法用来向共享内存写入数据，`load()` 方法用来从共享内存读出数据。比起直接的读写操作，它们的好处是保证了读写操作的原子性。

此外，它们还用来解决一个问题：多个线程使用共享内存的某个位置作为开关（flag），一旦该位置的值变了，就执行特定操作。这时，必须保证该位置的赋值操作，一定是在它前面的所有可能会改写内存的操作结束后执行；而该位置的取值操作，一定是在它后面所有可能会读取该位置的操作开始之前执行。`store()` 方法和 `load()` 方法就能做到这一点，编译器不会为了优化，而打乱机器指令的执行顺序。

```
Atomics.load(typedArray, index)
Atomics.store(typedArray, index, value)
```

store() typedArray SharedArrayBuffer
typedArray[index] load() typedArray SharedArrayBuffer
typedArray[index]

```
// main.js
ia[42] = 314159;  // 191
Atomics.store(ia, 37, 123456);  // 163

// Worker worker.js
while (Atomics.load(ia, 37) == 163);
console.log(ia[37]);  // 123456
console.log(ia[42]);  // 314159
```

Atomics.store() 42 37 37
163 Worker 37 42
Atomics.load()

```
// 
const worker = new Worker('worker.js');
const length = 10;
const size = Int32Array.BYTES_PER_ELEMENT * length;
// 
const sharedBuffer = new SharedArrayBuffer(size);
const sharedArray = new Int32Array(sharedBuffer);
for (let i = 0; i < 10; i++) {
  // 10 
  Atomics.store(sharedArray, i, 0);
}
worker.postMessage(sharedBuffer);
```

Atomics.store() Worker Atomics.load()

```
// worker.js
self.addEventListener('message', (event) => {
  const sharedArray = new Int32Array(event.data);
  for (let i = 0; i < 10; i++) {
    const arrayValue = Atomics.load(sharedArray, i);
    console.log(`The item at array index ${i} is ${arrayValue}`);
```

```
  }
}, false);
```

（2）Atomics.exchange()

Worker 线程如果要写入数据，可以使用上面的 `Atomics.store()` 方法，也可以使用 `Atomics.exchange()` 方法。它们的区别是，`Atomics.store()` 返回写入的值，而 `Atomics.exchange()` 返回被替换的值。

```
// Worker 线程
self.addEventListener('message', (event) => {
  const sharedArray = new Int32Array(event.data);
  for (let i = 0; i < 10; i++) {
    if (i % 2 === 0) {
      const storedValue = Atomics.store(sharedArray, i, 1);
      console.log(`The item at array index ${i} is now ${storedValue}`);
    } else {
      const exchangedValue = Atomics.exchange(sharedArray, i, 2);
      console.log(`The item at array index ${i} was ${exchangedValue}, now 2`);
    }
  }
}, false);
```

上面代码将共享内存的偶数位置的值改成 `1`，奇数位置的值改成 `2`。

（3）Atomics.wait()，Atomics.notify()

使用 `while` 循环等待主线程的通知，不是很高效，如果用在主线程，就会造成卡顿，`Atomics` 对象提供了 `wait()` 和 `notify()` 两个方法用于等待通知。这两个方法相当于锁内存，即在一个线程进行操作时，让其他线程休眠（建立锁），等到操作结束，再唤醒那些休眠的线程（解除锁）。

`Atomics.notify()` 方法以前叫做 `Atomics.wake()`，后来进行了改名。

```
// Worker 线程
self.addEventListener('message', (event) => {
  const sharedArray = new Int32Array(event.data);
  const arrayIndex = 0;
  const expectedStoredValue = 50;
  Atomics.wait(sharedArray, arrayIndex, expectedStoredValue);
  console.log(Atomics.load(sharedArray, arrayIndex));
}, false);
```

上面代码中，`Atomics.wait()`方法等同于告诉 Worker 线程，只要满足给定条件（`sharedArray`的`0`号位置等于`50`），就在这一行 Worker 线程进入休眠。

主线程一旦更改了指定位置的值，就可以唤醒 Worker 线程。

```
// 主线程
const newArrayValue = 100;
Atomics.store(sharedArray, 0, newArrayValue);
const arrayIndex = 0;
const queuePos = 1;
Atomics.notify(sharedArray, arrayIndex, queuePos);
```

上面代码中，`sharedArray`的`0`号位置改为`100`，然后就执行`Atomics.notify()`方法，唤醒在`sharedArray`的`0`号位置休眠队列里的一个线程。

`Atomics.wait()`方法的使用格式如下。

```
Atomics.wait(sharedArray, index, value, timeout)
```

它的四个参数含义如下。

- sharedArray: 共享内存的视图数组。
- index: 视图数据的位置（从 0开始）。
- value: 该位置的预期值。一旦实际值等于预期值，就进入休眠。
- timeout: 整数，表示过了这个时间以后，就自动唤醒，单位毫秒。该参数可选，默认值是`Infinity`，即无限期的休眠，只有通过`Atomics.notify()`方法才能唤醒。

`Atomics.wait()`的返回值是一个字符串，共有三种可能的值。如果`sharedArray[index]`不等于`value`，就返回字符串`not-equal`，否则就进入休眠。如果`Atomics.notify()`方法唤醒，就返回字符串`ok`；如果因为超时唤醒，就返回字符串`timed-out`。

`Atomics.notify()`方法的使用格式如下。

```
Atomics.notify(sharedArray, index, count)
```

它的三个参数含义如下。

- sharedArray: 共享内存的视图数组。
- index: 视图数据的位置（从 0开始）。
- count: 需要唤醒的 Worker 线程的数量，默认为`Infinity`。

`Atomics.notify()`方法一旦唤醒休眠的 Worker 线程，就会让它继续往下运行。

请看一个例子。

```
// 主线程
console.log(ia[37]);  // 163
Atomics.store(ia, 37, 123456);
Atomics.notify(ia, 37, 1);

// Worker 线程
Atomics.wait(ia, 37, 163);
console.log(ia[37]);  // 123456
```

上面代码中，视图数组 `ia` 的第 37 号位置，原来的值是 `163`。Worker 线程使用 `Atomics.wait()` 方法，指定只要 `ia[37]` 等于 `163`，就进入休眠状态。主线程使用 `Atomics.store()` 方法，将 `123456` 写入 `ia[37]`，然后使用 `Atomics.notify()` 方法唤醒 Worker 线程。

另外，基于 `wait` 和 `notify` 这两个方法的锁内存实现，可以看 Lars T Hansen 的 [js-lock-and-condition](#) 这个库。

注意，浏览器的主线程不宜设置休眠，这会导致用户失去响应。而且，主线程实际上会拒绝进入休眠。

（4）运算方法

共享内存上面的某些运算是不能被打断的，即不能在运算过程中，让其他线程改写内存上面的值。Atomics 对象提供了一些运算方法，防止数据被改写。

```
Atomics.add(sharedArray, index, value)
```

`Atomics.add` 用于将 `value` 加到 `sharedArray[index]`，返回 `sharedArray[index]` 旧的值。

```
Atomics.sub(sharedArray, index, value)
```

`Atomics.sub` 用于将 `value` 从 `sharedArray[index]` 减去，返回 `sharedArray[index]` 旧的值。

```
Atomics.and(sharedArray, index, value)
```

`Atomics.and` 用于将 `value` 与 `sharedArray[index]` 进行位运算 `and`，放入 `sharedArray[index]`，并返回旧的值。

```
Atomics.or(sharedArray, index, value)
```

`Atomics.or` 用于将 `value` 与 `sharedArray[index]` 进行位运算 `or`，放入 `sharedArray[index]`，并返回旧的值。

```
Atomics.xor(sharedArray, index, value)
```

Atomic.xor用于将 vaule与 sharedArray[index] 进行位运算 xor，放入 sharedArray[index]
，并返回旧的值。

（ 5）其他方法

Atomics 对象还有以下方法。

Atomics.compareExchange(sharedArray, index, oldval, newval) ：如果 sharedArray[index] 等于 oldval ，就写入 newval，返回 oldval。
Atomics.isLockFree(size) ：返回一个布尔值，表示 Atomics对象是否可以处理某个 size
的内存锁定。如果返回 false，应用程序就需要自己来实现锁定。

Atomics.compareExchange的一个用途是，从 SharedArrayBuffer
读取一个值，然后对该值进行某个操作，操作结束以后，检查一下 SharedArrayBuffer
里面原来那个值是否发生变化（即被其他线程改写过）。如果没有改写过，就将它写回原来的位置，否则读取
新的值，再重头进行一次操作。

版本号 #1
由 壳壳虫 创建于 2020-04-16 10:29:52 UTC
由 壳壳虫 更新于 2025-06-28 23:15:13 UTC