10th October 2019

ArrayBuffer, binary arrays

In web-development we meet binary data mostly while dealing with files (create, upload, download). Another typical use case is image processing.

That's all possible in JavaScript, and binary operations are high-performant.

Although, there's a bit of confusion, because there are many classes. To name a few:

- `ArrayBuffer`, `Uint8Array`, `DataView`, `Blob`, `File`, etc.

Binary data in JavaScript is implemented in a non-standard way, compared to other languages. But when we sort things out, everything becomes fairly simple.

The basic binary object is `ArrayBuffer` – a reference to a fixed-length contiguous memory area.

We create it like this:

```
let buffer = new ArrayBuffer(16); // create a buffer of length 16
alert(buffer.byteLength); // 16
```

This allocates a contiguous memory area of 16 bytes and pre-fills it with zeroes.

> **`ArrayBuffer` is not an array of something**
>
> Let's eliminate a possible source of confusion. `ArrayBuffer` has nothing in common with `Array`:
>
> - It has a fixed length, we can't increase or decrease it.
> - It takes exactly that much space in the memory.
> - To access individual bytes, another "view" object is needed, not `buffer[index]`.

`ArrayBuffer` is a memory area. What's stored in it? It has no clue. Just a raw sequence of bytes.

To manipulate an `ArrayBuffer`, we need to use a "view" object.

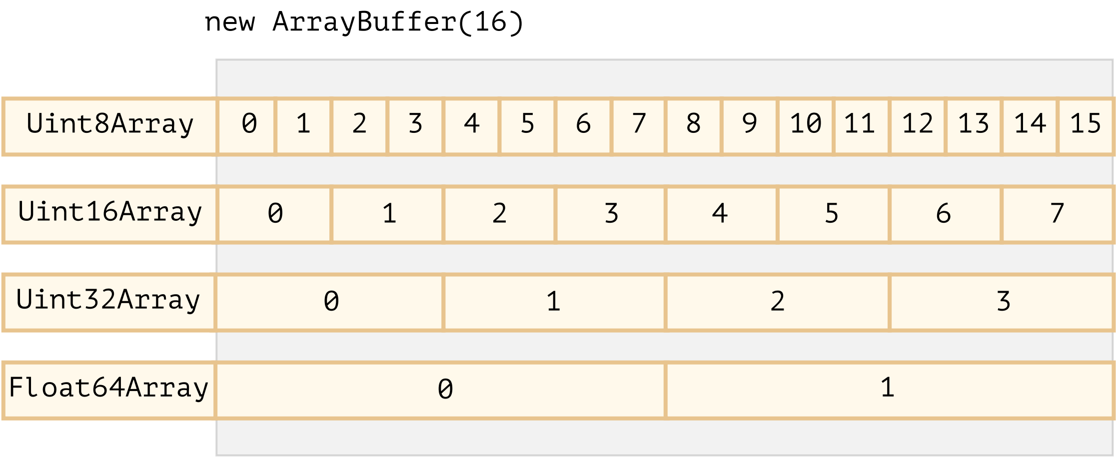
A view object does not store anything on it's own. It's the "eyeglasses" that give an interpretation of the bytes stored in the `ArrayBuffer`.

For instance:

- **`Uint8Array`** – treats each byte in `ArrayBuffer` as a separate number, with possible values are from 0 to 255 (a byte is 8-bit, so it can hold only that much). Such value is called a "8-bit unsigned integer".
- **`Uint16Array`** – treats every 2 bytes as an integer, with possible values from 0 to 65535. That's called a "16-bit unsigned integer".

- **`Uint32Array`** – treats every 4 bytes as an integer, with possible values from 0 to 4294967295. That's called a "32-bit unsigned integer".
- **`Float64Array`** – treats every 8 bytes as a floating point number with possible values from 5.0×10^{-324} to 1.8×10^{308}.

So, the binary data in an `ArrayBuffer` of 16 bytes can be interpreted as 16 "tiny numbers", or 8 bigger numbers (2 bytes each), or 4 even bigger (4 bytes each), or 2 floating-point values with high precision (8 bytes each).



`ArrayBuffer` is the core object, the root of everything, the raw binary data.

But if we're going to write into it, or iterate over it, basically for almost any operation – we must use a view, e.g:

```
let buffer = new ArrayBuffer(16); // create a buffer of length 16

let view = new Uint32Array(buffer); // treat buffer as a sequence of 32-bit i

alert(Uint32Array.BYTES_PER_ELEMENT); // 4 bytes per integer

alert(view.length); // 4, it stores that many integers
alert(view.byteLength); // 16, the size in bytes

// let's write a value
view[0] = 123456;

// iterate over values
for(let num of view) {
  alert(num); // 123456, then 0, 0, 0 (4 values total)
}
```

TypedArray

The common term for all these views (`Uint8Array` , `Uint32Array` , etc) is TypedArray. They share the same set of methods and properities.

They are much more like regular arrays: have indexes and iterable.

A typed array constructor (be it `Int8Array` or `Float64Array` , doesn't matter) behaves differently depending on argument types.

There are 5 variants of arguments:

```
new TypedArray(buffer, [byteOffset], [length]);
new TypedArray(object);
new TypedArray(typedArray);
new TypedArray(length);
new TypedArray();
```

1. If an `ArrayBuffer` argument is supplied, the view is created over it. We used that syntax already.

   Optionally we can provide `byteOffset` to start from (0 by default) and the `length` (till the end of the buffer by default), then the view will cover only a part of the `buffer` .

2. If an `Array` , or any array-like object is given, it creates a typed array of the same length and copies the content.

   We can use it to pre-fill the array with the data:

   ```
   let arr = new Uint8Array([0, 1, 2, 3]);
   alert( arr.length ); // 4, created binary array of the same length
   alert( arr[1] ); // 1, filled with 4 bytes (unsigned 8-bit integers) with
   ```

3. If another `TypedArray` is supplied, it does the same: creates a typed array of the same length and copies values. Values are converted to the new type in the process, if needed.

   ```
   let arr16 = new Uint16Array([1, 1000]);
   let arr8 = new Uint8Array(arr16);
   alert( arr8[0] ); // 1
   alert( arr8[1] ); // 232, tried to copy 1000, but can't fit 1000 into 8 bi
   ```

4. For a numeric argument `length` – creates the typed array to contain that many elements. Its byte length will be `length` multiplied by the number of bytes in a single item `TypedArray.BYTES_PER_ELEMENT` :

   ```
   let arr = new Uint16Array(4); // create typed array for 4 integers
   alert( Uint16Array.BYTES_PER_ELEMENT ); // 2 bytes per integer
   alert( arr.byteLength ); // 8 (size in bytes)
   ```

5. Without arguments, creates an zero-length typed array.

We can create a `TypedArray` directly, without mentioning `ArrayBuffer` . But a view cannot exist without an underlying `ArrayBuffer` , so gets created automatically in all these cases except the first one (when provided).

To access the `ArrayBuffer` , there are properties:

- `arr.buffer` – references the `ArrayBuffer`.
- `arr.byteLength` – the length of the `ArrayBuffer`.

So, we can always move from one view to another:

```
let arr8 = new Uint8Array([0, 1, 2, 3]);

// another view on the same data
let arr16 = new Uint16Array(arr8.buffer);
```

Here's the list of typed arrays:

- `Uint8Array`, `Uint16Array`, `Uint32Array` – for integer numbers of 8, 16 and 32 bits.
  - `Uint8ClampedArray` – for 8-bit integers, "clamps" them on assignment (see below).
- `Int8Array`, `Int16Array`, `Int32Array` – for signed integer numbers (can be negative).
- `Float32Array`, `Float64Array` – for signed floating-point numbers of 32 and 64 bits.

⚠ No `int8` or similar single-valued types

Please note, despite of the names like `Int8Array`, there's no single-value type like `int`, or `int8` in JavaScript.

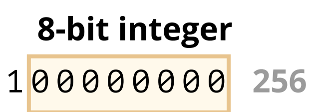
That's logical, as `Int8Array` is not an array of these individual values, but rather a view on `ArrayBuffer`.

Out-of-bounds behavior

What if we attempt to write an out-of-bounds value into a typed array? There will be no error. But extra bits are cut-off.

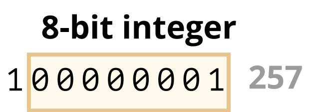
For instance, let's try to put 256 into `Uint8Array`. In binary form, 256 is `100000000` (9 bits), but `Uint8Array` only provides 8 bits per value, that makes the available range from 0 to 255.

For bigger numbers, only the rightmost (less significant) 8 bits are stored, and the rest is cut off:



So we'll get zero.

For 257, the binary form is `100000001` (9 bits), the rightmost 8 get stored, so we'll have `1` in the array:



In other words, the number modulo 2^8 is saved.

Here's the demo:

```
let uint8array = new Uint8Array(16);

let num = 256;
alert(num.toString(2)); // 100000000 (binary representation)

uint8array[0] = 256;
uint8array[1] = 257;

alert(uint8array[0]); // 0
alert(uint8array[1]); // 1
```

`Uint8ClampedArray` is special in this aspect, its behavior is different. It saves 255 for any number that is greater than 255, and 0 for any negative number. That behavior is useful for image processing.

TypedArray methods

`TypedArray` has regular `Array` methods, with notable exceptions.

We can iterate, `map`, `slice`, `find`, `reduce` etc.

There are few things we can't do though:

- No `splice` – we can't "delete" a value, because typed arrays are views on a buffer, and these are fixed, contiguous areas of memory. All we can do is to assign a zero.
- No `concat` method.

There are two additional methods:

- `arr.set(fromArr, [offset])` copies all elements from `fromArr` to the `arr`, starting at position `offset` (0 by default).
- `arr.subarray([begin, end])` creates a new view of the same type from `begin` to `end` (exclusive). That's similar to `slice` method (that's also supported), but doesn't copy anything – just creates a new view, to operate on the given piece of data.

These methods allow us to copy typed arrays, mix them, create new arrays from existing ones, and so on.

DataView

DataView is a special super-flexible "untyped" view over `ArrayBuffer`. It allows to access the data on any offset in any format.

- For typed arrays, the constructor dictates what the format is. The whole array is supposed to be uniform. The i-th number is `arr[i]`.

- With `DataView` we access the data with methods like `.getUint8(i)` or `.getUint16(i)`. We choose the format at method call time instead of the construction time.

The syntax:

```
new DataView(buffer, [byteOffset], [byteLength])
```

- **`buffer`** – the underlying `ArrayBuffer`. Unlike typed arrays, `DataView` doesn't create a buffer on its own. We need to have it ready.
- **`byteOffset`** – the starting byte position of the view (by default 0).
- **`byteLength`** – the byte length of the view (by default till the end of `buffer`).

For instance, here we extract numbers in different formats from the same buffer:

```
// binary array of 4 bytes, all have the maximal value 255
let buffer = new Uint8Array([255, 255, 255, 255]).buffer;

let dataView = new DataView(buffer);

// get 8-bit number at offset 0
alert( dataView.getUint8(0) ); // 255

// now get 16-bit number at offset 0, it consists of 2 bytes, together iterpr
alert( dataView.getUint16(0) ); // 65535 (biggest 16-bit unsigned int)

// get 32-bit number at offset 0
alert( dataView.getUint32(0) ); // 4294967295 (biggest 32-bit unsigned int)

dataView.setUint32(0, 0); // set 4-byte number to zero, thus setting all byte
```

`DataView` is great when we store mixed-format data in the same buffer. E.g we store a sequence of pairs (16-bit integer, 32-bit float). Then `DataView` allows to access them easily.

Summary

`ArrayBuffer` is the core object, a reference to the fixed-length contiguous memory area.

To do almost any operation on `ArrayBuffer`, we need a view.

- It can be a `TypedArray`:
  - `Uint8Array`, `Uint16Array`, `Uint32Array` – for unsigned integers of 8, 16, and 32 bits.
  - `Uint8ClampedArray` – for 8-bit integers, "clamps" them on assignment.
  - `Int8Array`, `Int16Array`, `Int32Array` – for signed integer numbers (can be negative).
  - `Float32Array`, `Float64Array` – for signed floating-point numbers of 32 and 64 bits.
- Or a `DataView` – the view that uses methods to specify a format, e.g. `getUint8(offset)`.

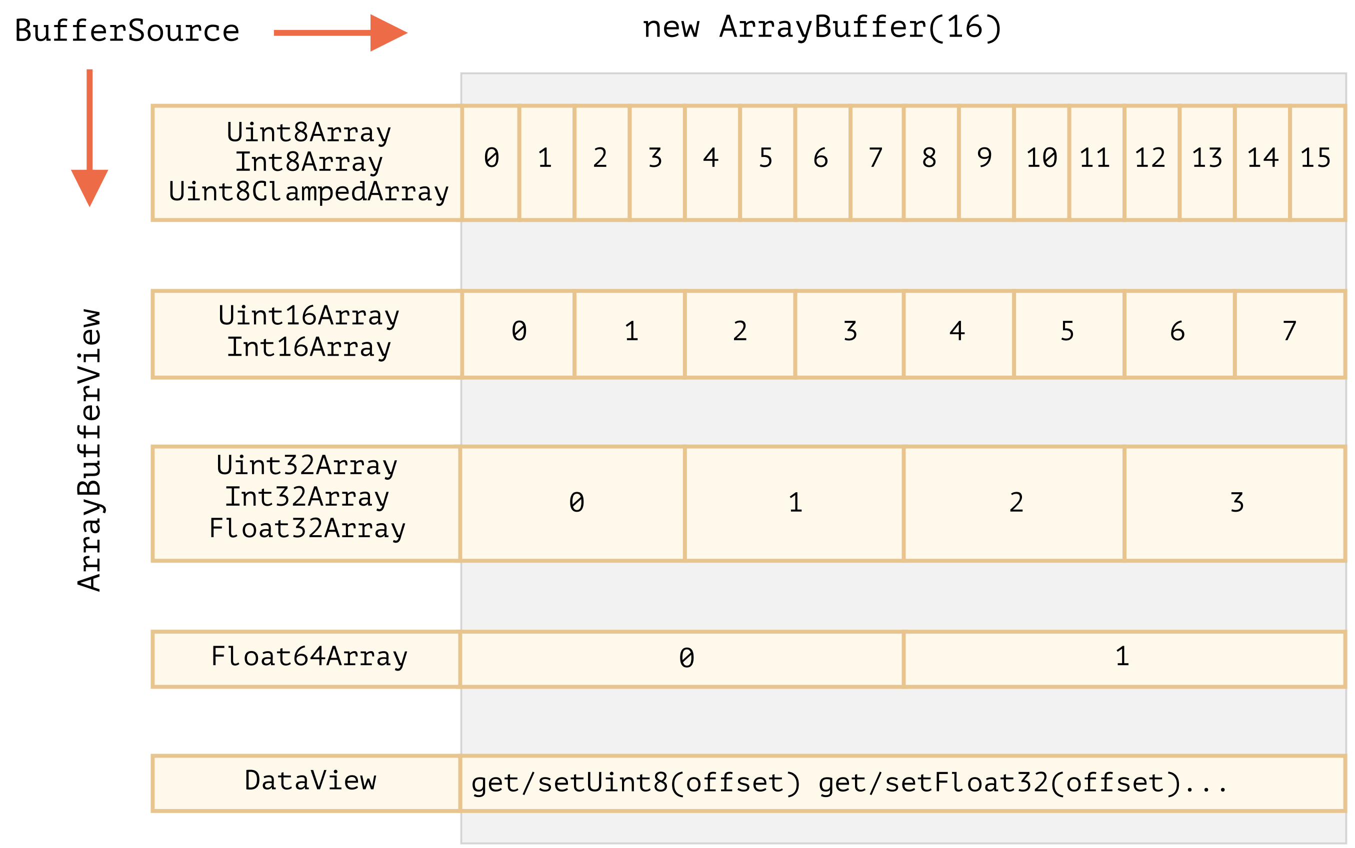
In most cases we create and operate directly on typed arrays, leaving `ArrayBuffer` under cover, as a "common discriminator". We can access it as `.buffer` and make another view if needed.

There are also two additional terms, that are used in descriptions of methods that operate on binary data:

- `ArrayBufferView` is an umbrella term for all these kinds of views.
- `BufferSource` is an umbrella term for `ArrayBuffer` or `ArrayBufferView`.

We'll see these terms in the next chapters. `BufferSource` is one of the most common terms, as it means "any kind of binary data" – an `ArrayBuffer` or a view over it.

Here's a cheatsheet:



Tasks

Concatenate typed arrays

Given an array of `Uint8Array`, write a function `concat(arrays)` that returns a concatenation of them into a single array.

Open a sandbox with tests.

solution

Previous lesson Next lesson

Share Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)

© 2007—2020 Ilya Kantor about the project contact us terms of usage privacy policy