Most popular (last 30 days)

ECMAScript 2017: the final feature set

ECMAScript 6 modules: the final syntax

ES proposal: import() – dynamically importing ES modules

Making transpiled ES modules more spec-compliant

ES proposal: Shared memory and atomics

Classes in ECMAScript 6 (final semantics)

Communicating between Web Workers via MessageChannel

Most popular (all time)

ECMAScript 6 modules: the final syntax

Classes in ECMAScript 6 (final semantics)

Iterating over arrays and objects in JavaScript

ECMAScript 6's new array methods

The final feature set of ECMAScript 2016 (ES7)

WebAssembly: a binary format for the web

Basic JavaScript for the impatient programmer

Google Dart to "ultimately ... replace JavaScript"

Six nifty ES6 tricks

Google's Polymer and the future of web UI frameworks

Blog archive

► 2017 (4)

► 2016 (38)

2015-09-05

Typed Arrays in ECMAScript 6

Labels: dev, esnext, javascript

Typed Arrays are an ECMAScript 6 API for handling binary data. This blog post explains how they work.

1. Overview

Code example:

```
let typedArray = new Uint8Array([0,1,2]);
console.log(typedArray.length); // 3
typedArray[0] = 5;
let normalArray = [...typedArray]; // [5,1,2]

// The elements are stored in typedArray.buffer.
// Get a different view on the same data:
let dataView = new DataView(typedArray.buffer);
console.log(dataView.getUint8(0)); // 5
```

Instances of `ArrayBuffer` store the binary data to be processed. Two kinds of *views* are used to access the data:

- Typed Arrays (`Uint8Array`, `Int16Array`, `Float32Array`, etc.) interpret the ArrayBuffer as an indexed sequence of elements of a single type.
- Instances of `DataView` let you access data as elements of several types (`Uint8`, `Int16`, `Float32`, etc.), at any byte offset inside an ArrayBuffer.

The following browser APIs support Typed Arrays (details are mentioned later):

- File API
- XMLHttpRequest
- Fetch API
- Canvas
- WebSockets
- And more

2. Introduction

For a long time, JavaScript was not very good at handling binary data. This changed with the introduction of the Typed Array API, whose main use cases are:

- Processing binary data: manipulating image data in HTML Canvas elements, parsing binary files, handling binary network protocols, etc.
- Interacting with native APIs: Native APIs often receive and return data in a binary format, which you could neither store nor manipulate well in traditional JavaScript. That meant that whenever you were communicating with such an API, data had to be converted from JavaScript to binary and back, for every call. Typed Arrays eliminate this bottleneck. One example of communicating with native APIs is WebGL, for which Typed Arrays were initially created. Section "History of Typed Arrays" of the article "Typed Arrays: Binary Data in the Browser" (by Ilmari Heikkinen for HTML5 Rocks) has more information.

Dr. Axel Rauschmayer

Free online books by Axel

Speaking JavaScript [up to ES5]

Exploring ES6

JavaScript training:
Ecmanauten

▼ 2015 (65)
► December (7)
► November (6)
► October (13)
▼ September (5)
Customizing ES6 via well-known symbols
__proto__ in ECMAScript 6
ECMAScript 6: holes in Arrays
The names of functions in ES6
Typed Arrays in ECMAScript 6
► August (6)
► July (3)
► June (3)
► April (4)
► March (4)
► February (8)
► January (6)
► 2014 (55)
► 2013 (98)
► 2012 (177)
► 2011 (380)
► 2010 (174)
► 2009 (68)
► 2008 (46)
► 2007 (12)
► 2006 (1)
► 2005 (2)

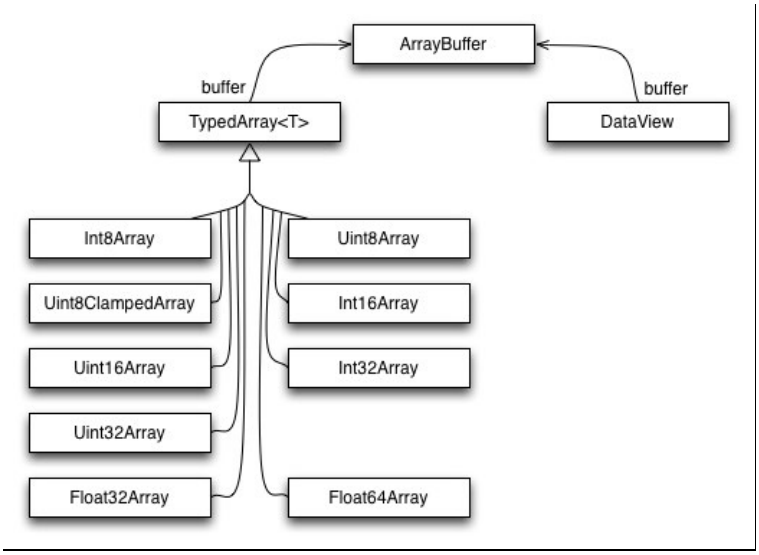
Labels

dev (592)
javascript (400)
computers (316)
life (194)
jslang (179)
esnext (156)
apple (107)
webdev (95)
mobile (83)
scitech (50)
hack (49)
mac (47)
google (39)
java (37)
ios (33)
business (32)
video (32)
clientjs (31)

Two kinds of objects work together in the Typed Array API:

- Buffers: Instances of `ArrayBuffer` hold the binary data.
- Views: provide the methods for accessing the binary data. There are two kinds of views:
  - An instance of a Typed Array constructor (`Uint8Array`, `Float64Array`, etc.) works much like a normal Array, but only allows a single type for its elements and doesn't have holes.
  - An instance of `DataView` lets you access data at any byte offset in the buffer, and interprets that data as one of several types (`Uint8`, `Float64`, etc.).

This is a diagram of the structure of the Typed Array API (notable: all Typed Arrays have a common superclass):



Typed Arrays were a separate specification before they became part of the ECMAScript 6 standard.

2.1. Element types

The following element types are supported by the API:

Element type	Bytes	Description	C type
Int8	1	8-bit signed integer	signed char
Uint8	1	8-bit unsigned integer	unsigned char
Uint8C	1	8-bit unsigned integer (clamped conversion)	unsigned char
Int16	2	16-bit signed integer	short
Uint16	2	16-bit unsigned integer	unsigned short
Int32	4	32-bit signed integer	int
Uint32	4	32-bit unsigned integer	unsigned int
Float32	4	32-bit floating point	float
Float64	8	64-bit floating point	double

The element type `Uint8C` is special: it is not supported by `DataView` and only exists to enable `Uint8ClampedArray`. This Typed Array is used by the `canvas` element (where it replaces `CanvasPixelArray`). The only difference between `Uint8C` and `Uint8` is how overflow and underflow are handled (as explained in the next section). It is recommended to avoid the former – quoting Brendan Eich:

> Just to be super-clear (and I was around when it was born), `Uint8ClampedArray` is *totally* a historical artifact (of the HTML5 canvas element). Avoid unless you really are doing canvas-y things.

hci (27)
entertainment (26)
nodejs (26)
society (26)
browser (25)
firefox (25)
html5 (24)
ipad (24)
movie (23)
psychology (22)
2ality (18)
tv (18)
android (17)
social (17)
chrome (16)
fun (16)
jsmodules (16)
tablet (16)
humor (15)
politics (15)
web (15)
cloud (14)
hardware (14)
microsoft (14)
software engineering (13)
blogging (12)
gaming (12)
eclipse (11)
gwt (11)
numbers (11)
programming languages (11)
app store (10)
media (10)
nature (10)
security (10)
semantic web (10)
software (10)
twitter (10)
webos (10)
12quirks (9)
education (9)
jstools (9)
photo (9)
webcomponents (9)
windows 8 (9)
async (8)
idea (8)
iphone (8)
itunes (8)
scifi-fantasy (8)
app (7)
babel (7)
bookmarklet (7)

2.2. Handling overflow and underflow

Normally, when a value is out of the range of the element type, modulo arithmetic is used to convert it to a value within range. For signed and unsigned integers that means that:

- The highest value plus one is converted to the lowest value (0 for unsigned integers).
- The lowest value minus one is converted to the highest value.

Modulo conversion for unsigned 8-bit integers:

```
> let uint8 = new Uint8Array(1);
> uint8[0] = 255; uint8[0] // highest value within range
255
> uint8[0] = 256; uint8[0] // overflow
0
> uint8[0] = 0; uint8[0] // lowest value within range
0
> uint8[0] = -1; uint8[0] // underflow
255
```

Modulo conversion for signed 8-bit integers:

```
> let int8 = new Int8Array(1);
> int8[0] = 127; int8[0] // highest value within range
127
> int8[0] = 128; int8[0] // overflow
-128
> int8[0] = -128; int8[0] // lowest value within range
-128
> int8[0] = -129; int8[0] // underflow
127
```

Clamped conversion is different:

- All underflowing values are converted to the lowest value.
- All overflowing values are converted to the highest value.

```
> let uint8c = new Uint8ClampedArray(1);
> uint8c[0] = 255; uint8c[0] // highest value within range
255
> uint8c[0] = 256; uint8c[0] // overflow
255
> uint8c[0] = 0; uint8c[0] // lowest value within range
0
> uint8c[0] = -1; uint8c[0] // underflow
0
```

2.3. Endianness

Whenever a type (such as `Uint16`) is stored as multiple bytes, *endianness* matters:

- Big endian: the most significant byte comes first. For example, the `Uint16` value 0xABCD is stored as two bytes – first 0xAB, then 0xCD.
- Little endian: the least significant byte comes first. For example, the `Uint16` value 0xABCD is stored as two bytes – first 0xCD, then 0xAB.

Endianness tends to be fixed per CPU architecture and consistent across native APIs. Typed Arrays are used to communicate with those APIs, which is why their endianness follows the endianness of the platform and can't be changed.

On the other hand, the endianness of protocols and binary files varies and is fixed across platforms. Therefore, we must be able to access data with either endianness. DataViews serve this use case and let you specify endianness when you get or set a value.

Quoting Wikipedia on Endianness:

- Big-endian representation is the most common convention in data networking; fields in the protocols of the Internet protocol suite, such as IPv4, IPv6, TCP,

chromeos (7)
english (7)
es proposal (7)
fringe (7)
html (7)
jsint (7)
jsshell (7)
thunderbolt (7)
webapp (7)
advancedjs (6)
blogger (6)
crowdsourcing (6)
latex (6)
lion (6)
promises (6)
ted (6)
book (5)
environment (5)
gadget (5)
googleio (5)
intel (5)
jsarrays (5)
jshistory (5)
layout (5)
light peak (5)
michael j. fox (5)
music (5)
pdf (5)
polymer (5)
shell (5)
tc39 (5)
template literals (5)
underscorejs (5)
vlc (5)
__proto__ (4)
coffeescript (4)
concurrency (4)
dart (4)
facebook (4)
gimp (4)
googleplus (4)
health (4)
howto (4)
hp (4)
javafx (4)
kindle (4)
leopard (4)
macbook (4)
motorola (4)
münchen (4)
occupy (4)
pl fundamentals (4)
presenting (4)
publishing (4)

and UDP, are transmitted in big-endian order. For this reason, big-endian byte order is also referred to as network byte order.

- Little-endian storage is popular for microprocessors in part due to significant historical influence on microprocessor designs by Intel Corporation.

You can use the following function to determine the endianness of a platform.

```
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');
    }
}
```

There are also platforms that arrange *words* (pairs of bytes) with a different endianness than bytes inside words. That is called mixed endianness. Should you want to support such a platform then it is easy to extend the previous code.

2.4. Negative indices

With the bracket operator `[ ]`, you can only use non-negative indices (starting at 0). The methods of ArrayBuffers, Typed Arrays and DataViews work differently: every index can be negative. If it is, it counts backwards from the length. In other words, it is added to the length to produce a normal index. Therefore `-1` refers to the last element, `-2` to the second-last, etc. Methods of normal Arrays work the same way.

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

Offsets, on the other hand, must be non-negative. If, for example, you pass `-1` to:

```
DataView.prototype.getInt8(byteOffset)
```

then you get a `RangeError`.

3. ArrayBuffers

ArrayBuffers store the data, *views* (Typed Arrays and DataViews) let you read and change it. In order to create a DataView, you need to provide its constructor with an ArrayBuffer. Typed Array constructors can optionally create an ArrayBuffer for you.

3.1. `ArrayBuffer` constructor

The signature of the constructor is:

```
ArrayBuffer(length : number)
```

Invoking this constructor via `new` creates an instance whose capacity is `length` bytes. Each of those bytes is initially 0.

3.2. Static `ArrayBuffer` methods

- `ArrayBuffer.isView(arg)`
  Returns `true` if `arg` is an object and a view for an ArrayBuffer. Only Typed Arrays and DataViews have the required internal property `[[ViewedArrayBuffer]]`. That means that this check is roughly equivalent to checking whether `arg` is an instance of a Typed Array or of `DataView`.

3.3. `ArrayBuffer.prototype` properties

- `get ArrayBuffer.prototype.byteLength`
  Returns the capacity of this ArrayBuffer in bytes.

series (4)
textbook (4)
web design (4)
amazon (3)
asmjs (3)
back to the future (3)
bitwise_ops (3)
css (3)
es2016 (3)
flattr (3)
fluentconf (3)
food (3)
foreign languages (3)
house (3)
icloud (3)
info mgmt (3)
jsfuture (3)
jsstyle (3)
linux (3)
mozilla (3)
python (3)
regexp (3)
samsung (3)
tizen (3)
traffic (3)
typedjs (3)
unix (3)
adobe (2)
angry birds (2)
angularjs (2)
astronomy (2)
audio (2)
comic (2)
design (2)
dom (2)
ecommerce (2)
eval (2)
exploring es6 (2)
facebook flow (2)
facets (2)
flash (2)
free (2)
futurama (2)
guide (2)
history (2)
hyena (2)
internet explorer (2)
iteration (2)
journalism (2)
jquery (2)
jsengine (2)
jslib (2)
law (2)
lightning (2)

- `ArrayBuffer.prototype.slice(start, end)`
  Creates a new ArrayBuffer that contains the bytes of this ArrayBuffer whose indices are greater than or equal to `start` and less than `end`. `start` and `end` can be negative (see Sect. “Negative indices”).

4. Typed Arrays

The various kinds of Typed Array are only different w.r.t. to the type of their elements:

- Typed Arrays whose elements are integers: `Int8Array`, `Uint8Array`, `Uint8ClampedArray`, `Int16Array`, `Uint16Array`, `Int32Array`, `Uint32Array`
- Typed Arrays whose elements are floats: `Float32Array`, `Float64Array`

4.1. Typed Arrays versus normal Arrays

Typed Arrays are much like normal Arrays: they have a `length`, elements can be accessed via the bracket operator `[ ]` and they have all of the standard Array methods. They differ from Arrays in the following ways:

- All of their elements have the same type, setting elements converts values to that type.
- They are contiguous. Normal Arrays can have *holes* (indices in the range [0, `arr.length`) that have no associated element), Typed Arrays can’t.
- Initialized with zeros. This is a consequence of the previous item:
  - `new Array(10)` creates a normal Array without any elements (it only has holes).
  - `new Uint8Array(10)` creates a Typed Array whose 10 elements are all 0.
- An associated buffer. The elements of a Typed Array `ta` are not stored in `ta`, they are stored in an associated ArrayBuffer that can be accessed via `ta.buffer`.

4.2. Typed Arrays are iterable

Typed Arrays implement a method whose key is `Symbol.iterator` and are therefore iterable (consult chapter “Iterables and iterators” in “Exploring ES6” for more information). That means that you can use the `for-of` loop and similar mechanisms in ES6:

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

ArrayBuffers and DataViews are not iterable.

4.3. Converting Typed Arrays to and from normal Arrays

To convert a normal Array to a Typed Array, you make it the parameter of a Typed Array constructor. For example:

```
> let tarr = new Uint8Array([0,1,2]);
```

The classic way to convert a Typed Array to an Array is to invoke `Array.prototype.slice` on it. This trick works for all Array-like objects (such as `arguments`) and Typed Arrays are Array-like.

```
> Array.prototype.slice.call(tarr)
[ 0, 1, 2 ]
```

In ES6, you can use the spread operator (`...`), because Typed Arrays are iterable:

```
> [...tarr]
[ 0, 1, 2 ]
```

Another ES6 alternative is `Array.from()`, which works with either iterables or Array-like objects:

markdown (2)
math (2)
meego (2)
month (2)
nike (2)
nokia (2)
npm (2)
programming (2)
raffle (2)
repl (2)
servo (2)
sponsor (2)
steve jobs (2)
travel (2)
typescript (2)
usb (2)
winphone (2)
wwdc (2)
airbender (1)
amdefine (1)
aol (1)
app urls (1)
architecture (1)
atscript (1)
basic income (1)
biology (1)
blink (1)
bluetooth (1)
canada (1)
clip (1)
coding (1)
community (1)
cross-platform (1)
deutsch (1)
diaspora (1)
distributed-social-network (1)
dsl (1)
dvd (1)
dzone (1)
emacs (1)
emberjs (1)
energy (1)
esnext news (1)
esprop (1)
example (1)
facetator (1)
feedback (1)
firefly (1)
firefoxos (1)
fritzbox (1)
german (1)
git (1)
guest (1)
guice (1)

```
> Array.from(tarr)
[ 0, 1, 2 ]
```

4.4. The Species pattern

Some methods create new instances that are similar to `this`. The species pattern lets you configure what constructor should be used to do so. For example, if you create a subclass `MyArray` of `Array` then the default is that `map()` creates instances of `MyArray`. If you want it to create instances of `Array`, you can use the species pattern to make that happen. Details are explained in Sect "The species pattern" in "Exploring ES6".

ArrayBuffers use the species pattern in the following locations:

- `ArrayBuffer.prototype.slice()`
- Whenever an ArrayBuffer is cloned inside a Typed Array or DataView.

Typed Arrays use the species pattern in the following locations:

- `TypedArray<T>.prototype.filter()`
- `TypedArray<T>.prototype.map()`
- `TypedArray<T>.prototype.slice()`
- `TypedArray<T>.prototype.subarray()`

DataViews don't use the species pattern.

4.5. The inheritance hierarchy of Typed Arrays

As you could see in the diagram at the beginning of this post, all Typed Array classes (`Uint8Array` etc.) have a common superclass. I'm calling that superclass `TypedArray`, but it is not directly accessible from JavaScript (the ES6 specification calls it *the intrinsic object %TypedArray%*). `TypedArray.prototype` houses all methods of Typed Arrays.

4.6. Static `TypedArray` methods

Both static `TypedArray` methods are inherited by its subclasses (`Uint8Array` etc.).

`TypedArray.of()`

This method has the signature:

```
TypedArray.of(...items)
```

It creates a new Typed Array that is an instance of `this` (the class on which `of()` was invoked). The elements of that instance are the parameters of `of()`.

You can think of `of()` as a custom literal for Typed Arrays:

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

`TypedArray.from()`

This method has the signature:

```
TypedArray<U>.from(source : Iterable<T>, mapfn? : T => U, thisArg?)
```

It converts the iterable `source` into an instance of `this` (a Typed Array).

For example, normal Arrays are iterable and can be converted with this method:

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

Typed Arrays are iterable, too:

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

The optional `mapfn` lets you transform the elements of `source` before they become elements of the result. Why perform the two steps *mapping* and *conversion* in one go?

h.264 (1)
home entertainment (1)
hosting (1)
htc (1)
ical (1)
jsdom (1)
jsmyth (1)
library (1)
location (1)
marketing (1)
mars (1)
meta-data (1)
middle east (1)
mpaa (1)
msl (1)
mssurface (1)
netflix (1)
nsa (1)
obama (1)
openoffice (1)
opinion (1)
oracle (1)
organizing (1)
philosophy (1)
pixar (1)
pnacl (1)
prism (1)
privacy (1)
proxies (1)
puzzle (1)
raspberry pi (1)
read (1)
rodney (1)
rust (1)
safari (1)
sponsoring (1)
star trek (1)
static generation (1)
talk (1)
technique (1)
theora (1)
thunderbird (1)
typography (1)
unicode (1)
v8 (1)
voice control (1)
webassembly (1)
webkit (1)
webm (1)
webpack (1)
yahoo (1)

Compared to performing the first step separately, via `source.map()`, there are two advantages:

1. No intermediate Array or Typed Array is needed.
2. When converting a Typed Array to a Typed Array whose elements have a higher precision, the mapping step can make use of that higher precision.

To illustrate the second advantage, let's use `map()` to double the elements of a Typed Array:

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

As you can see, the values overflow and are coerced into the `Int8` range of values. If map via `from()`, you can choose the type of the result so that values don't overflow:

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

According to Allen Wirfs-Brock, mapping between Typed Arrays was what motivated the `mapfn` parameter of `from()`.

4.7. `TypedArray.prototype` properties

Indices accepted by Typed Array methods can be negative (they work like traditional Array methods that way). Offsets must be non-negative. For details, see Sect. "Negative indices".

Methods specific to Typed Arrays

The following properties are specific to Typed Arrays, normal Arrays don't have them:

- `get TypedArray<T>.prototype.buffer : ArrayBuffer`
  Returns the buffer backing this Typed Array.
- `get TypedArray<T>.prototype.byteLength : number`
  Returns the size in bytes of this Typed Array's buffer.
- `get TypedArray<T>.prototype.byteOffset : number`
  Returns the offset where this Typed Array "starts" inside its ArrayBuffer.
- `TypedArray<T>.prototype.set(arrayOrTypedArray, offset=0)`
  Copies all elements of `arrayOrTypedArray` to this Typed Array. The element at index 0 of `arrayOrTypedArray` is written to index `offset` of this Typed Array (etc.).
  - If `arrayOrTypedArray` is a normal Array, its elements are converted to numbers who are then converted to the element type `T` of this Typed Array.
  - If `arrayOrTypedArray` is a Typed Array then each of its elements is converted directly to the appropriate type for this Typed Array. If both Typed Arrays have the same element type then faster, byte-wise copying is used.
- `TypedArray<T>.prototype.subarray(begin=0, end=this.length) : TypedArray<T>`
  Returns a new Typed Array that has the same buffer as this Typed Array, but a (generally) smaller range. If `begin` is non-negative then the first element of the resulting Typed Array is `this[begin]`, the second `this[begin+1]` (etc.). If `begin` in negative, it is converted appropriately.

Array methods

The following methods are basically the same as the methods of normal Arrays:

- `TypedArray<T>.prototype.copyWithin(target : number, start : number, end = this.length) : This`
  Copies the elements whose indices are between `start` (including) and `end` (excluding) to indices starting at `target`. If the ranges overlap and the former range comes first then elements are copied in reverse order to avoid overwriting source elements before they are copied.
- `TypedArray<T>.prototype.entries() : Iterable<[number,T]>`
  Returns an iterable over [index,element] pairs for this Typed Array.
- `TypedArray<T>.prototype.every(callbackfn, thisArg?)`
  Returns `true` if `callbackfn` returns `true` for every element of this Typed

Array. Otherwise, it returns `false`. `every()` stops processing the first time `callbackfn` returns `false`.

- `TypedArray<T>.prototype.fill(value, start=0, end=this.length) : void`
  Set the elements whose indices range from `start` to `end` to `value`.
- `TypedArray<T>.prototype.filter(callbackfn, thisArg?) : TypedArray<T>`
  Returns a Typed Array that contains every element of this Typed Array for which `callbackfn` returns `true`. In general, the result is shorter than this Typed Array.
- `TypedArray<T>.prototype.find(predicate : T => boolean, thisArg?) : T`
  Returns the first element for which the function `predicate` returns `true`.
- `TypedArray<T>.prototype.findIndex(predicate : T => boolean, thisArg?) : number`
  Returns the index of the first element for which `predicate` returns `true`.
- `TypedArray<T>.prototype.forEach(callbackfn, thisArg?) : void`
  Iterates over this Typed Array and invokes `callbackfn` for each element.
- `TypedArray<T>.prototype.indexOf(searchElement, fromIndex=0) : number`
  Returns the index of the first element that strictly equals `searchElement`. The search starts at `fromIndex`.
- `TypedArray<T>.prototype.join(separator : string = ',') : string`
  Converts all elements to strings and concatenates them, separated by `separator`.
- `TypedArray<T>.prototype.keys() : Iterable<number>`
  Returns an iterable over the indices of this Typed Array.
- `TypedArray<T>.prototype.lastIndexOf(searchElement, fromIndex?) : number`
  Returns the index of the last element that strictly equals `searchElement`. The search starts at `fromIndex`, backwards.
- `get TypedArray<T>.prototype.length : number`
  Returns the length of this Typed Array.
- `TypedArray<T>.prototype.map(callbackfn, thisArg?) : TypedArray<T>`
  Returns a new Typed Array in which every element is the result of applying `callbackfn` to the corresponding element of this Typed Array.
- `TypedArray<T>.prototype.reduce(callbackfn : (previousValue : any, currentElement : T, currentIndex : number, array : TypedArray<T>) => any, initialValue?) : any`
  `callbackfn` is fed one element at a time, together with the result that was computed so far and computes a new result. Elements are visited from left to right.
- `TypedArray<T>.prototype.reduceRight(callbackfn : (previousValue : any, currentElement : T, currentIndex : number, array : TypedArray<T>) => any, initialValue?) : any`
  `callbackfn` is fed one element at a time, together with the result that was computed so far and computes a new result. Elements are visited from right to left.
- `TypedArray<T>.prototype.reverse() : This`
  Reverses the order of the elements of this Typed Array and returns `this`.
- `TypedArray<T>.prototype.slice(start=0, end=this.length) : TypedArray<T>`
  Create a new Typed Array that only has the elements of this Typed Array whose indices are between `start` (including) and `end` (excluding).
- `TypedArray<T>.prototype.some(callbackfn, thisArg?)`
  Returns `true` if `callbackfn` returns `true` for at least one element of this Typed Array. Otherwise, it returns `false`. `some()` stops processing the first time `callbackfn` returns `true`.
- `TypedArray<T>.prototype.sort(comparefn? : (number, number) => number)`
  Sorts this Typed Array, as specified via `comparefn`. If `comparefn` is missing, sorting is done ascendingly, by comparing via the less-than operator (`<`).
- `TypedArray<T>.prototype.toLocaleString(reserved1?, reserved2?)`
- `TypedArray<T>.prototype.toString()`
- `TypedArray<T>.prototype.values() : Iterable<T>`
  Returns an iterable over the values of this Typed Array.

Due to all of these methods being available for Arrays, you can consult the following two sources to find out more about how they work:

- The following methods are new in ES6 and explained in chapter "New Array features" of "Exploring ES6": `copyWithin`, `entries`, `fill`, `find`, `findIndex`, `keys`, `values`.
- All other methods are explained in chapter "Arrays" of "Speaking JavaScript".

4.8. `«ElementType»Array` constructor

Each Typed Array constructor has a name that follows the pattern `«ElementType»Array`, where `«ElementType»` is one of the element types in the table at the beginning. That means that there are 9 constructors for Typed Arrays: `Int8Array`, `Uint8Array`, `Uint8ClampedArray` (element type `Uint8C`), `Int16Array`, `Uint16Array`, `Int32Array`, `Uint32Array`, `Float32Array`, `Float64Array`.

Each constructor has five *overloaded* versions – it behaves differently depending on how many arguments it receives and what their types are:

- `«ElementType»Array(buffer, byteOffset=0, length?)`
  Creates a new Typed Array whose buffer is `buffer`. It starts accessing the buffer at the given `byteOffset` and will have the given `length`. Note that `length` counts elements of the Typed Array (with 1–4 bytes each), not bytes.
- `«ElementType»Array(length)`
  Creates a Typed Array with the given `length` and the appropriate buffer (whose size in bytes is `length * «ElementType»Array.BYTES_PER_ELEMENT`).
- `«ElementType»Array()`
  Creates a Typed Array whose `length` is 0. It also creates an associated empty ArrayBuffer.
- `«ElementType»Array(typedArray)`
  Creates a new Typed Array that has the same length and elements as `typedArray`. Values that are too large or small are converted appropriately.
- `«ElementType»Array(arrayLikeObject)`
  Treats `arrayLikeObject` like an Array and creates a new TypedArray that has the same length and elements. Values that are too large or small are converted appropriately.

The following code shows three different ways of creating the same Typed Array:

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

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

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

4.9. Static `«ElementType»Array` properties

- `«ElementType»Array.BYTES_PER_ELEMENT`
  Counts how many bytes are needed to store a single element:

```
> Uint8Array.BYTES_PER_ELEMENT
1
> Int16Array.BYTES_PER_ELEMENT
2
> Float64Array.BYTES_PER_ELEMENT
8
```

4.10. `«ElementType»Array.prototype` properties

- `«ElementType»Array.prototype.BYTES_PER_ELEMENT`
  The same as `«ElementType»Array.BYTES_PER_ELEMENT`.

5. DataViews

5.1. `DataView` constructor

- `DataView(buffer, byteOffset=0, byteLength=buffer.byteLength-byteOffset)`
  Creates a new DataView whose data is stored in the ArrayBuffer `buffer`. By

default, the new DataView can access all of `buffer`, the last two parameters allow you to change that.

5.2. `DataView.prototype` properties

- `get DataView.prototype.buffer`
  Returns the ArrayBuffer of this DataView.
- `get DataView.prototype.byteLength`
  Returns how many bytes can be accessed by this DataView.
- `get DataView.prototype.byteOffset`
  Returns at which offset this DataView starts accessing the bytes in its buffer.
- `DataView.prototype.get«ElementType»(byteOffset, littleEndian=false)`
  Reads a value from the buffer of this DataView.
  - «ElementType» can be: `Float32`, `Float64`, `Int8`, `Int16`, `Int32`, `Uint8`, `Uint16`, `Uint32`
- `DataView.prototype.set«ElementType»(byteOffset, value, littleEndian=false)`
  Writes `value` to the buffer of this DataView.
  - «ElementType» can be: `Float32`, `Float64`, `Int8`, `Int16`, `Int32`, `Uint8`, `Uint16`, `Uint32`

6. Browser APIs that support Typed Arrays

Typed Arrays have been around for a while, so there are quite a few browser APIs that support them.

6.1. File API

The file API lets you access local files. The following code demonstrates how to get the bytes of a submitted local file in an ArrayBuffer.

```
let fileInput = document.getElementById('fileInput');
let file = fileInput.files[0];
let reader = new FileReader();
reader.readAsArrayBuffer(file);
reader.onload = function () {
    let arrayBuffer = reader.result;
    ···
};
```

6.2. `XMLHttpRequest`

In newer versions of the `XMLHttpRequest` API, you can have the results delivered in an ArrayBuffer:

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

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

xhr.send();
```

6.3. Fetch API

Similarly to `XMLHttpRequest`, the Fetch API lets you request resources. But it is based on Promises, which makes it more convenient to use. The following code demonstrates how to download the content pointed to by `url` as an ArrayBuffer:

```
fetch(url)
.then(request => request.arrayBuffer())
.then(arrayBuffer => ···);
```

6.4. Canvas

Quoting the HTML5 specification:

> The `canvas` element provides scripts with a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, art, or other visual images on the fly.

The 2D Context of `canvas` lets you retrieve the bitmap data as an instance of `Uint8ClampedArray`:

```
let canvas = document.getElementById('my_canvas');
let context = canvas.getContext('2d');
let imageData = context.getImageData(0, 0, canvas.width, canvas.height);
let uint8ClampedArray = imageData.data;
```

6.5. WebSockets

WebSockets let you send and receive binary data via ArrayBuffers:

```
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
    let typedArray = new Uint8Array(4);
    socket.send(typedArray.buffer);
});

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

6.6. Other APIs

- WebGL uses the Typed Array API for: accessing buffer data, specifying pixels for texture mapping, reading pixel data, and more.
- The Web Audio API lets you decode audio data submitted via an ArrayBuffer.
- Media Source Extensions: The HTML media elements are currently `<audio>` and `<video>`. The Media Source Extensions API enables you to create streams to be played via those elements. You can add binary data to such streams via ArrayBuffers, Typed Arrays or DataViews.
- Communication with Web Workers: If you send data to a Worker via `postMessage()`, either the message (which will be cloned) or the transferable objects can contain ArrayBuffers.
- Cross-document communication: works similarly to communication with Web Workers and also uses the method `postMessage()`.

7. Extended example: JPEG SOF0 decoder

The code of the following example is on GitHub. And you can run it online.

The example is a web pages that lets you upload a JPEG file and parses its structure to determine the height and the width of the image and more.

7.1. The JPEG file format

A JPEG file is a sequence of *segments* (typed data). Each segment starts with the following four bytes:

- Marker (two bytes): declares what kind of data is stored in the segment. The first of the two bytes is always 0xFF. Each of the standard markers has a human readable name. For example, the marker 0xFFC0 has the name “Start Of Frame (Baseline DCT)”, short: “SOF0”.
- Length of segment (two bytes): how long is this segment (in bytes, including the length itself)?

JPEG files are big-endian on all platforms. Therefore, this example demonstrates how important it is that we can specify endianness when using DataViews.

7.2. The JavaScript code

The following function `processArrayBuffer()` is an abridged version of the actual code; I've removed a few error checks to reduce clutter. `processArrayBuffer()` receives an ArrayBuffer with the contents of the submitted JPEG file and iterates over its segments.

```
// JPEG is big endian
var IS_LITTLE_ENDIAN = false;

function processArrayBuffer(arrayBuffer) {
    try {
        var dv = new DataView(arrayBuffer);
        ...
        var ptr = 2;
        while (true) {
            ...
            var lastPtr = ptr;
            enforceValue(0xFF, dv.getUint8(ptr),
                'Not a marker');
            ptr++;
            var marker = dv.getUint8(ptr);
            ptr++;
            var len = dv.getUint16(ptr, IS_LITTLE_ENDIAN);
            ptr += len;
            logInfo('Marker: '+hex(marker)+' ('+len+' byte(s))');
            ...

            // Did we find what we were looking for?
            if (marker === 0xC0) { // SOF0
                logInfo(decodeSOF0(dv, lastPtr));
                break;
            }
        }
    } catch (e) {
        logError(e.message);
    }
}
```

This code uses the following helper functions (that are not shown here):

- `enforceValue()` throws an error if the expected value (first parameter) doesn't match the actual value (second parameter).
- `logInfo()` and `logError()` display messages on the page.
- `hex()` turns a number into a string with two hexadecimal digits.

`decodeSOF0()` parses the segment SOF0:

```
function decodeSOF0(dv, start) {
    // Example (16x16):
    // FF C0 00 11 08 00 10 00 10 03 01 22 00 02 11 01 03 11 01
    var data = {};
    start += 4; // skip marker 0xFFC0 and segment length 0x0011
    var data = {
        bitsPerColorComponent: dv.getUint8(start), // usually 0x08
        imageHeight: dv.getUint16(start+1, IS_LITTLE_ENDIAN),
        imageWidth: dv.getUint16(start+3, IS_LITTLE_ENDIAN),
        numberOfColorComponents: dv.getUint8(start+5),
    };
    return JSON.stringify(data, null, 4);
}
```

More information on the structure of JPEG files:

- "JPEG: Syntax and structure" (on Wikipedia)
- "JPEG File Interchange Format: File format structure" (on Wikipedia)

8. Availability

Much of the Typed Array API is implemented by all modern JavaScript engines, but several features are new to ECMAScript 6:

- Static methods borrowed from Arrays: `TypedArray<T>.from()`, `TypedArray<T>.of()`
- Prototype methods borrowed from Arrays: `TypedArray<T>.prototype.map()` etc.
- Iterable Typed Arrays
- Support for the species pattern
- An inheritance hierarchy where `TypedArray<T>` is the superclass of all Typed Array classes

It may take a while until these are available everywhere. As usual, kangax' "ES6 compatibility table" describes the status quo.

Reply • Share ›

Christian Bankester • a year ago

From §2.2: "One minus the lowest value is converted to the highest value." Should be "lowest value minus one", yeah?

Reply • Share ›

Axel Rauschmayer Mod ➚ Christian Bankester • a year ago

That is correct. Fixed now, thanks!

Reply • Share ›

Serkan Sipahi • a year ago

@Dr. Axel Rauschmayer What is the advantage of working with TypedArray/Binary-Datas? Performance, Design, ..., etc... I have not yet understood it :(

Reply • Share ›

LionessLover ➚ Serkan Sipahi • a year ago

This sentence sounds too simple but that's really all: You work with binary data when you *have* binary data.

If all you have is text like in a "normal" web app then of course you have no use for this except for unusual use cases. But when you have binary APIs to connect to or when you have image data (canvas - games!!!) and need to do transformations then you obviously want to treat your binary data as binary data. It's really as simple as that.

What kind of software have you developed? Only text-processing, as in it uses strings for input and strings for output (e.g. html)? Then you never encountered a need for binary. Everything you give to and get from the usual APIs is strings. From DOM manipulation to taking mouse or touch screen events in Javascript, it's all text.

But that's the web and the browser world. Historically most APIs were *binary*, not text. And multimedia - from graphics to video and audio - is all binary too, so with HTML 5 and games and more and

see more

2 Reply • Share ›

MaxArt • a year ago

We've been knowing typed arrays for a while, but this is a nice recap. Thanks Axel.

Reply • Share ›

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)