EN

JS

EPUB/PDF

→ Binary data, files

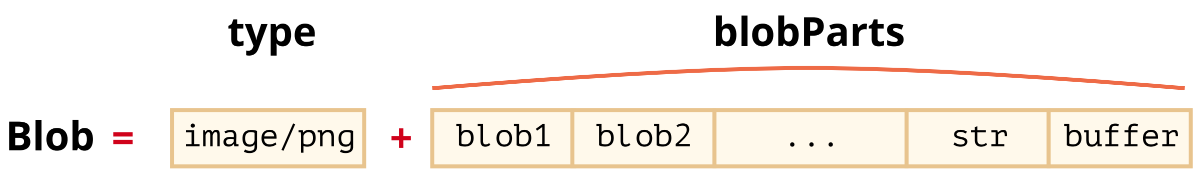
27th September 2019

Blob

`ArrayBuffer` and views are a part of ECMA standard, a part of JavaScript.

In the browser, there are additional higher-level objects, described in File API, in particular `Blob`.

`Blob` consists of an optional string `type` (a MIME-type usually), plus `blobParts` – a sequence of other `Blob` objects, strings and `BufferSource`.



The constructor syntax is:

```
new Blob(blobParts, options);
```

- **`blobParts`** is an array of `Blob`/`BufferSource`/`String` values.
- **`options`** optional object:
  - **`type`** – `Blob` type, usually MIME-type, e.g. `image/png`,
  - **`endings`** – whether to transform end-of-line to make the `Blob` correspond to current OS newlines (`\r\n` or `\n`). By default `"transparent"` (do nothing), but also can be `"native"` (transform).

For example:

```
// create Blob from a string
let blob = new Blob(["<html>…</html>"], {type: 'text/html'});
// please note: the first argument must be an array [...]
```

```
// create Blob from a typed array and strings
let hello = new Uint8Array([72, 101, 108, 108, 111]); // "Hello" in binary fo

let blob = new Blob([hello, ' ', 'world'], {type: 'text/plain'});
```

We can extract `Blob` slices with:

```
blob.slice([byteStart], [byteEnd], [contentType]);
```

- **`byteStart`** – the starting byte, by default 0.
- **`byteEnd`** – the last byte (exclusive, by default till the end).
- **`contentType`** – the `type` of the new blob, by default the same as the source.

The arguments are similar to `array.slice`, negative numbers are allowed too.

> **`Blob` objects are immutable**
>
> We can't change data directly in a `Blob`, but we can slice parts of a `Blob`, create new `Blob` objects from them, mix them into a new `Blob` and so on.
>
> This behavior is similar to JavaScript strings: we can't change a character in a string, but we can make a new corrected string.

Blob as URL

A Blob can be easily used as an URL for `<a>`, `<img>` or other tags, to show its contents.

Thanks to `type`, we can also download/upload `Blob` objects, and the `type` naturally becomes `Content-Type` in network requests.

Let's start with a simple example. By clicking on a link you download a dynamically-generated `Blob` with `hello world` contents as a file:

```
<!-- download attribute forces the browser to download instead of navigating
<a download="hello.txt" href='#' id="link">Download</a>

<script>
let blob = new Blob(["Hello, world!"], {type: 'text/plain'});

link.href = URL.createObjectURL(blob);
</script>
```

We can also create a link dynamically in JavaScript and simulate a click by `link.click()`, then download starts automatically.

Here's the similar code that causes user to download the dynamicallly created `Blob`, without any HTML:

```
let link = document.createElement('a');
link.download = 'hello.txt';

let blob = new Blob(['Hello, world!'], {type: 'text/plain'});

link.href = URL.createObjectURL(blob);

link.click();

URL.revokeObjectURL(link.href);
```

`URL.createObjectURL` takes a `Blob` and creates a unique URL for it, in the form `blob:<origin>/<uuid>`.

That's what the value of `link.href` looks like:

```
blob:https://javascript.info/1e67e00e-860d-40a5-89ae-6ab0cbee6273
```

The browser for each URL generated by `URL.createObjectURL` stores an the URL → `Blob` mapping internally. So such URLs are short, but allow to access the `Blob`.

A generated URL (and hence the link with it) is only valid within the current document, while it's open. And it allows to reference the `Blob` in `<img>`, `<a>`, basically any other object that expects an url.

There's a side-effect though. While there's a mapping for a `Blob`, the `Blob` itself resides in the memory. The browser can't free it.

The mapping is automatically cleared on document unload, so `Blob` objects are freed then. But if an app is long-living, then that doesn't happen soon.

So if we create a URL, that `Blob` will hang in memory, even if not needed any more.

`URL.revokeObjectURL(url)` removes the reference from the internal mapping, thus allowing the `Blob` to be deleted (if there are no other references), and the memory to be freed.

In the last example, we intend the `Blob` to be used only once, for instant downloading, so we call `URL.revokeObjectURL(link.href)` immediately.

In the previous example with the clickable HTML-link, we don't call `URL.revokeObjectURL(link.href)`, because that would make the `Blob` url invalid. After the revocation, as the mapping is removed, the URL doesn't work any more.

Blob to base64

An alternative to `URL.createObjectURL` is to convert a `Blob` into a base64-encoded string.

That encoding represents binary data as a string of ultra-safe "readable" characters with ASCII-codes from 0 to 64. And what's more important – we can use this encoding in "data-urls".

A data url has the form `data:[<mediatype>][;base64],<data>`. We can use such urls everywhere, on par with "regular" urls.

For instance, here's a smiley:

```
<img src="data:image/png;base64,R0lGODlhDAAMAKIFAF5LAP/zxAAAANyuAP/gaP///wAAA
```

The browser will decode the string and show the image:

To transform a `Blob` into base64, we'll use the built-in `FileReader` object. It can read data from Blobs in multiple formats. In the next chapter we'll cover it more in-depth.

Here's the demo of downloading a blob, now via base-64:

```
let link = document.createElement('a');
link.download = 'hello.txt';

let blob = new Blob(['Hello, world!'], {type: 'text/plain'});

let reader = new FileReader();
reader.readAsDataURL(blob); // converts the blob to base64 and calls onload

reader.onload = function() {
  link.href = reader.result; // data url
  link.click();
};
```

Both ways of making an URL of a `Blob` are usable. But usually `URL.createObjectURL(blob)` is simpler and faster.

URL.createObjectURL(blob)

- We need to revoke them if care about memory.
- Direct access to blob, no “encoding/decoding”

Blob to data url

- No need to revoke anything.
- Performance and memory losses on big `Blob` objects for encoding.

Image to blob

We can create a `Blob` of an image, an image part, or even make a page screenshot. That’s handy to upload it somewhere.

Image operations are done via `<canvas>` element:

1. Draw an image (or its part) on canvas using canvas.drawImage.
2. Call canvas method .toBlob(callback, format, quality) that creates a `Blob` and runs `callback` with it when done.

In the example below, an image is just copied, but we could cut from it, or transform it on canvas prior to making a blob:

```
// take any image
let img = document.querySelector('img');

// make <canvas> of the same size
let canvas = document.createElement('canvas');
canvas.width = img.clientWidth;
canvas.height = img.clientHeight;

```

```
let context = canvas.getContext('2d');

// copy image to it (this method allows to cut image)
context.drawImage(img, 0, 0);
// we can context.rotate(), and do many other things on canvas

// toBlob is async opereation, callback is called when done
canvas.toBlob(function(blob) {
  // blob ready, download it
  let link = document.createElement('a');
  link.download = 'example.png';

  link.href = URL.createObjectURL(blob);
  link.click();

  // delete the internal blob reference, to let the browser clear memory fr
  URL.revokeObjectURL(link.href);
}, 'image/png');
```

If we prefer `async/await` instead of callbacks:

```
let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png
```

For screenshotting a page, we can use a library such as https://github.com/niklasvh/html2canvas. What it does is just walks the page and draws it on `<canvas>`. Then we can get a `Blob` of it the same way as above.

From Blob to ArrayBuffer

The `Blob` constructor allows to create a blob from almost anything, including any `BufferSource`.

But if we need to perform low-level processing, we can get the lowest-level `ArrayBuffer` from it using `FileReader`:

```
// get arrayBuffer from blob
let fileReader = new FileReader();

fileReader.readAsArrayBuffer(blob);

fileReader.onload = function(event) {
  let arrayBuffer = fileReader.result;
};
```

Summary

While `ArrayBuffer`, `Uint8Array` and other `BufferSource` are “binary data”, a Blob represents “binary data with type”.

That makes Blobs convenient for upload/download operations, that are so common in the browser.

Methods that perform web-requests, such as XMLHttpRequest, fetch and so on, can work with `Blob` natively, as well as with other binary types.

We can easily convert betweeen `Blob` and low-level binary data types:

- We can make a Blob from a typed array using `new Blob(...)` constructor.
- We can get back `ArrayBuffer` from a Blob using `FileReader` , and then create a view over it for low-level binary processing.

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…)