Бинарные данные

ArrayBuffer, TypedArray, DataView

Базовый объект для работы с бинарными данными имеет тип **ArrayBuffer** и представляет собой ссылку на непрерывную область памяти фиксированной длины. Вот так мы можем создать его экземпляр:

```
let buffer = new ArrayBuffer(16); // создаётся буфер длиной 16 байт
alert(buffer.byteLength); // 16
```

ArrayBuffer – это не массив!

Для работы с ArrayBuffer нам нужен специальный объект, реализующий «**представление**» данных. Общий термин для всех таких представлений – это **TypedArray**, типизированный массив. У них имеется набор одинаковых свойств и методов. Они уже намного больше напоминают обычные массивы: элементы проиндексированы, и возможно осуществить обход содержимого. Типизированные массивы TypedArray, за некоторыми заметными исключениями, имеют те же методы, что и массивы Array. Мы можем обходить их, вызывать map, slice, find, reduce и т.д. Однако, есть некоторые вещи, которые нельзя осуществить

Список типизированных массивов:

- Uint8Array, Uint16Array, Uint32Array – целые беззнаковые числа по 8, 16 и 32 бита соответственно.
- Uint8ClampedArray – 8-битные беззнаковые целые, обрезаются по верхней и нижней границе при присвоении.
- Int8Array, Int16Array, Int32Array – целые числа со знаком (могут быть отрицательными).
- Float32Array, Float64Array – 32- и 64-битные числа со знаком и плавающей точкой.

Для доступа к **ArrayBuffer** в **TypedArray** есть следующие **свойства**:

- buffer – ссылка на объект ArrayBuffer.
- byteLength – размер содержимого ArrayBuffer в байтах.

Что если мы попытаемся записать в типизированный массив значение, которое превышает допустимое для данного массива? Ошибки не будет. Лишние биты просто будут отброшены (останутся правые биты)

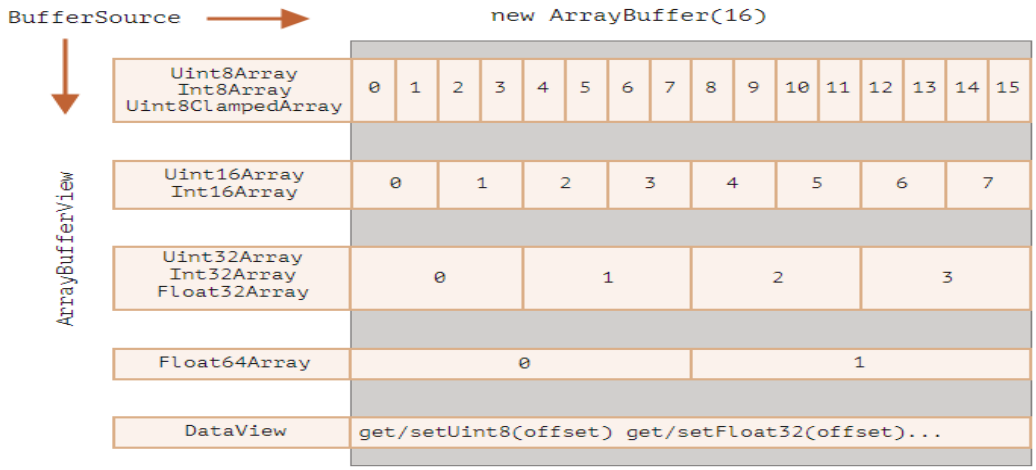
DataView – **представление**, использующее отдельные методы, чтобы уточнить формат данных при обращении, например, getUint8(offset). Представление DataView отлично подходит, когда мы храним данные разного формата в одном буфере. Синтаксис:

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

buffer – ссылка на бинарные данные ArrayBuffer. В отличие от типизированных массивов, DataView не создаёт буфер автоматически. Нам нужно заранее подготовить его самим.
byteOffset – начальная позиция данных для представления (по умолчанию 0).
byteLength – длина данных (в байтах), используемых в представлении (по умолчанию – до конца buffer).

```
// бинарный массив из 4х байт, каждый имеет максимальное значение 255
let buffer = new Uint8Array([255, 255, 255, 255]).buffer;
let dataView = new DataView(buffer);
// получим 8-битное число на позиции 0
alert( dataView.getUint8(0) ); // 255
// а сейчас мы получим 16-битное число на той же позиции 0, оно состоит из 2-х байт, вместе составляющих число 65535
alert( dataView.getUint16(0) ); // 65535 (максимальное 16-битное беззнаковое целое)
```

ArrayBufferView – это общее название для представлений всех типов.
BufferSource – это общее название для ArrayBuffer или ArrayBufferView.



TextDecoder и TextEncoder

Встроенный объект **TextDecoder** позволяет декодировать данные из бинарного буфера в обычную строку. Для этого прежде всего нам нужно создать сам декодер:

let decoder = new TextDecoder([label], [options]);

- **label** – тип кодировки, utf-8 используется по умолчанию, но также поддерживаются big5, windows-1251 и многие другие.
- **options** – объект с дополнительными настройками:
- **fatal** – boolean, если значение true, тогда генерируется ошибка для невалидных (не декодируемых) символов, в ином случае (по умолчанию) они заменяются символом \uFFFD.
- **ignoreBOM** – boolean, если значение true, тогда игнорируется BOM (дополнительный признак, определяющий порядок следования байтов), что необходимо крайне редко.

…и после использовать его метод decode:

let str = decoder.decode([input], [options]);

- **input** – бинарный буфер (BufferSource) для декодирования.
- **options** – объект с дополнительными настройками:
- **stream** – true для декодирования потока данных, при этом decoder вызывается вновь и вновь для каждого следующего фрагмента данных. В этом случае многобайтовый символ может иногда быть разделён и попасть в разные фрагменты данных. Это опция указывает TextDecoder запомнить символ, на котором остановился процесс, и декодировать его со следующим фрагментом.

```
let uint8Array = new Uint8Array([72, 101, 108, 108, 111]);
alert( new TextDecoder().decode(uint8Array) ); // Hello
```

TextEncoder поступает наоборот – кодирует строку в бинарный массив. Имеет следующий синтаксис:

let encoder = new **TextEncoder**();

Поддерживается **только кодировка «utf-8».**

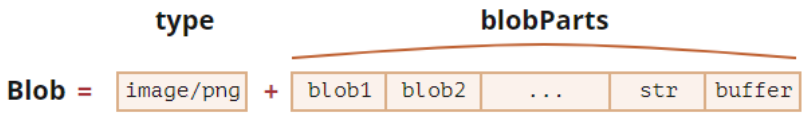
Кодировщик имеет следующие два **метода**:

- encode(str) – возвращает бинарный массив Uint8Array, содержащий закодированную строку.
- encodeInto(str, destination) – кодирует строку (str) и помещает её в destination, который должен быть экземпляром Uint8Array

```
let encoder = new TextEncoder();
let uint8Array = encoder.encode("Hello");
alert(uint8Array); // 72,101,108,108,111
```

Blob

Объект **Blob** состоит из необязательной строки type (обычно MIME-тип) и blobParts – последовательности других объектов Blob, строк и BufferSource.



Конструктор имеет следующий синтаксис:

new Blob(blobParts, options); где

blobParts – **массив** значений **Blob/BufferSource/String**.
options – необязательный объект с дополнительными настройками:

- **type** – тип объекта, обычно MIME-тип, например. image/png,
- **endings** – если указан, то окончания строк создаваемого Blob будут изменены в соответствии с текущей операционной системой (\r\n или \n). По умолчанию "transparent" (ничего не делать), но также может быть "native" (изменять).

Например:

```
// создадим Blob из строки
let blob = new Blob(["<html>…</html>"], {type: 'text/html'});
// обратите внимание: первый аргумент должен быть массивом [...]
```

Мы можем получить срез Blob, используя:
blob.slice([byteStart], [byteEnd], [contentType]);
byteStart – стартовая позиция байта, по умолчанию 0.
byteEnd – последний байт, по умолчанию до конца.
contentType – тип type создаваемого Blob-объекта, по умолчанию такой же, как и исходный.
Аргументы – как в array.slice, отрицательные числа также разрешены.

Мы **не можем изменять данные** напрямую в Blob, но мы можем делать срезы и создавать новый Blob на их основе, объединять несколько объектов в новый и так далее. Это поведение аналогично JavaScript-строке: мы не можем изменить символы в строке, но мы можем создать новую исправленную строку на базе имеющейся.

Blob как URL

Blob может быть использован как URL для <a>, <img> или других тегов, для показа содержимого.
Давайте начнём с простого примера. При клике на ссылку мы загружаем динамически генерируемый Blob с hello world содержимым как файл:

```
<!-- download атрибут указывает браузеру делать загрузку вместо навигации -->
<a download="hello.txt" href='#' id="link">Загрузить</a>

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

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

Также используется URL.revokeObjectURL(url) удаляет внутреннюю ссылку на объект, что позволяет удалить его (если нет другой ссылки) сборщику мусора, и память будет освобождена.

Blob to base64

Альтернатива URL.createObjectURL – конвертация Blob-объекта в строку с кодировкой **base64**. Эта кодировка представляет двоичные данные в виде строки с безопасными для чтения символами в ASCII-кодах от 0 до 64. И что более важно – мы можем использовать эту кодировку для «**data-urls**».

data url имеет форму data:[<mediatype>][;base64],<data>. Мы можем использовать такой url где угодно наряду с «обычным» url. Например, смайлик:

```
<img
src="
KBOaGcF35DhWHamZUW0K4mAbiwWtuf0uxFAgA7">
```

Браузер декодирует строку и показывает смайлик

Вот пример загрузки Blob при помощи base64:

```
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); // конвертирует Blob в base64 и вызывает onload

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

Изображение в Blob

Мы можем создать Blob для **изображения**, части изображения или даже создать скриншот страницы.
Операции с изображениями выполняются через элемент <**canvas**>:

- Для отрисовки изображения (или его части) на холсте (canvas) используется canvas.drawImage.
- Вызов canvas-метода .toBlob(callback, format, quality) создаёт Blob и вызывает функцию callback при завершении.

File и FileReader

Объект **File** наследуется от объекта Blob и обладает возможностями по взаимодействию с файловой системой.
Есть два способа его получить.

1. есть конструктор, похожий на Blob:

new File(fileParts, fileName, [options])

fileParts – массив значений Blob/BufferSource/строки.
fileName – имя файла, строка.
options – необязательный объект со свойством:

- **lastModified** – дата последнего изменения в формате таймстамп (целое число).

2. Чаще всего мы получаем файл из <input type="file"> или через перетаскивание с помощью мыши, или из других интерфейсов браузера. В этом случае файл получает эту информацию из ОС.

Так как File наследует от Blob, у объектов File есть те же свойства плюс:
name – имя файла,
lastModified – таймстамп для даты последнего изменения.

В этом примере мы получаем объект File из <input type="file">:

```
<input type="file" onchange="showFile(this)">

<script>
function showFile(input) {
  let file = input.files[0];
  alert(`File name: ${file.name}`); // например, my.png
  alert(`Last modified: ${file.lastModified}`); // например, 1552830408824
}
</script>
```

На заметку: Через <input> можно выбрать несколько файлов, поэтому input.files – псевдомассив выбранных файлов. Здесь у нас только один файл, поэтому мы просто берём input.files[0].

FileReader объект, цель которого читать данные из Blob (и, следовательно, из File тоже).

let reader = new FileReader(); // без аргументов

Основные методы:

- readAsArrayBuffer(blob) – считать данные как ArrayBuffer (для бинарных файлов, для низкоуровневой побайтовой работы с бинарными данными)
- readAsText(blob, [encoding]) – считать данные как строку (кодировка по умолчанию: utf-8)
- readAsDataURL(blob) – считать данные как base64-кодированный URL (когда мы хотим использовать данные в src для img или другого тега. Есть альтернатива – можно не читать файл, а вызвать **URL.createObjectURL(file)**).
- abort() – отменить операцию.

В процессе чтения происходят следующие **события**:

- loadstart – чтение начато.
- progress – срабатывает во время чтения данных.
- load – нет ошибок, чтение окончено.
- abort – вызван abort().
- error – произошла ошибка.
- loadend – чтение завершено (успешно или нет).

Когда чтение закончено, мы сможем получить **доступ к его результату** следующим образом:

- reader.result результат чтения (если оно успешно)
- reader.error объект ошибки (при неудаче).

Вот пример чтения файла:

```
<input type="file" onchange="readFile(this)">

<script>
function readFile(input) {
  let file = input.files[0];
  let reader = new FileReader();
  reader.readAsText(file);
  reader.onload = function() {
    console.log(reader.result);
  };
  reader.onerror = function() {
    console.log(reader.error);
  };
}
</script>
```

FileReader работает **для любых объектов Blob**, а не только для файлов. Поэтому мы можем использовать его для преобразования Blob в другой формат:
readAsArrayBuffer(blob) – в ArrayBuffer,
readAsText(blob, [encoding]) – в строку (альтернатива TextDecoder),
readAsDataURL(blob) – в формат base64-кодированного URL.