> Эта страница была переведена с английского языка силами сообщества. Вы тоже можете внести свой вклад, присоединившись к русскоязычному сообществу MDN Web Docs.

Типизированные массивы JavaScript

Типизированные массивы в JavaScript являются массивоподобными объектами, предоставляющими механизм доступа к сырым двоичным данным. Как вы уже можете знать, массив `Array` растёт и обрезается динамически, и может содержать элементы любого типа JavaScript. Благодаря оптимизациям JavaScript движков,

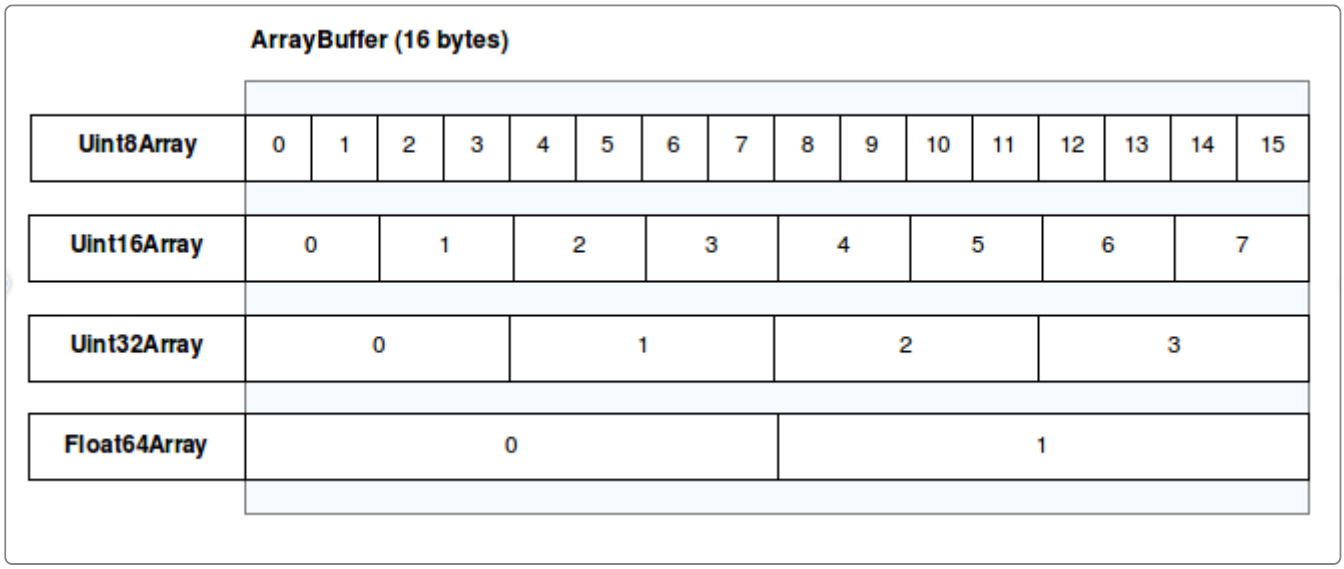
данными, требуется доступ к сырым данным WebSocket, и так далее. Становится очевидным, что возможность быстрой и эффективной работы с двоичными данными в JavaScript будет очень полезной, для чего типизированные массивы и предназначены.

Не следует путать типизированные массивы с обычными массивами: так, например, вызов `Array.isArray()` для типизированного массива вернёт `false`. Более того, не все методы, доступные для обычных массивов поддерживаются типизированными массивами (например, push и pop).

Буферы и представления: архитектура типизированных массивов

Для достижения максимальной гибкости и производительности, реализация типизированных массивов в JavaScript разделена на буферы и представления. Буфер (`ArrayBuffer`) — это объект, представляющий из себя набор данных. Он не имеет формата и не предоставляет возможности доступа к своему содержимому. Для доступа к памяти буфера вам нужно использовать представление. Представление

предоставляет контекст: тип данных, начальную позицию в буфере и количество элементов. Это позволяет представить данные в виде типизированного массива.



ArrayBuffer

Объект `ArrayBuffer` — это набор бинарных данных с фиксированной длиной. Вы не можете манипулировать содержимым `ArrayBuffer` напрямую. Вместо этого, необходимо создать типизированное представление `DataView`, которое будет отображать буфер в определённом формате, и даст доступ на запись и чтение его содержимого.

Типизированные представления

Название типизированного представления массива говорит само за себя. Оно представляет массив в распространённых числовых форматах, таких как `Int8`, `Uint32`, `Float64` и так далее. Среди прочих, существует специальное представление `Uint8ClampedArray`. Оно ограничивает значения интервалом от 0 до 255. Это полезно, например, при Обработке данных изображения в Canvas.

DataView

Объект `DataView` — это низкоуровневый интерфейс, предоставляющий API для записи/чтения произвольных данных в буфер. Это полезно при работе с различными типами данных, например. В то время как типизированные представления всегда имеют порядок байт (смотрите Endianness (en-US)) соответствующий используемому в вашей операционной системе, `DataView` позволяет контроллировать порядок байт (byte-order). По умолчанию это big-endian, но через API можно установить little-endian.

Веб API, использующие типизированные массивы

`FileReader.prototype.readAsArrayBuffer()`

Метод `FileReader.prototype.readAsArrayBuffer()` читает содержимое заданного `Blob` или `File`.

`XMLHttpRequest.prototype.send()`

Метод `send()` экземпляра `XMLHttpRequest` теперь поддерживает в качестве аргумента `ArrayBuffer`.

`ImageData.data`

Имеет тип `Uint8ClampedArray` и представляет изображение в виде одномерного массива, где цветовые компоненты расположены в порядке RGBA, и их значения принудительно ограничены диапазоном от 0 до 255.

Примеры

Использование представлений с буферами

Прежде всего, необходимо создать буфер с фиксированной длиной 16 байт:

JS

```
var buffer = new ArrayBuffer(16);
```

На данном этапе мы имеем область памяти в 16 байт, инициализированной нулевыми значениями. Всё, что мы можем сделать сейчас, это убедиться, что длина буфера действительно 16 байт:

JS

```
if (buffer.byteLength === 16) {
  console.log("Да, это 16 байт.");
} else {
  console.log("О нет, размер не наш!");
}
```

Прежде чем мы сможем приступить к полноценной работе с памятью, нам нужно создать представление. Давайте создадим представление, которое отображает буфер как массив из 32-битных целочисленных значений со знаком:

```
JS
var int32View = new Int32Array(buffer);
```

Теперь мы можем получить доступ к элементам представления как к элементам обычного массива:

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

Этот код поместит 4 элемента в буфер (4 элемента по 4 байта даст 16 байт) со следующими значениям: `0`, `2`, `4` и `6`.

Множество представлений для одних и тех же данных

Всё становится намного интереснее, если создать несколько разных представлений для одного и того же буфера. Например, приведённый выше код можно дополнить следующим образом:

```
JS
var int16View = new Int16Array(buffer);

for (var i = 0; i < int16View.length; i++) {
  console.log("Entry " + i + ": " + int16View[i]);
}
```

Здесь мы создаём 16-битное целочисленное представление, которое ссылается на тот же самый буфер, что и 32-битное представление, и затем выводим все 16-битные элементы этого представления. Мы получим следующий вывод: 0, 0, 2, 0, 4, 0, 6, 0.

Можно пойти дальше. Оцените этот код:

JS

```
int16View[0] = 32;
console.log("Элемент 0 в 32-битном представлении теперь равен " + int32View[0]);
```

Результатом выполнения станет текст: "Элемент 0 в 32-битном представлении теперь равен 32". Другими словами, два массива на самом деле являются лишь разными представлениями одного и того же буфера данных в разных форматах. Вы можете повторить это с представлениями любого типа.

Работа со сложными структурами данных

Комбинируя буфер и множество представлений разного формата, имеющих разные смещения относительно начала буфера, можно управляться с объектами содержащими разнородные данные. Это позволяет, к примеру, взаимодействовать со сложными структурам из WebGL, файлами данных или структурами языка C (сопоставление данных JS и C).

Рассмотрим следующую структуру из языка C:

CPP

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

Получить доступ к полям этой структуры можно следующим образом:

JS

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

// ... поместить данные структуры в буфер ...

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

Теперь получить или изменить значение поля `amountDue`, к примеру, можно путём обращения к `amountDueView[0]`.

> Примечание: Выравнивание данных в языке C является платформозависимым. Принимайте меры по вычислению правильных отступов в данных с учётом выравнивания.

Преобразование в обычные массивы

Иногда после обработки типизированного массива бывает полезно конвертировать его в обычный массив, чтобы получить доступ к методам прототипа `Array`. Для этих целей существует метод `Array.from`. А в тех случаях, когда `Array.from` не поддерживается, используйте следующий код:

```
JS
```

```
var typedArray = new Uint8Array([1, 2, 3, 4]),
  normalArray = Array.prototype.slice.call(typedArray);
normalArray.length === 4;
normalArray.constructor === Array;
```

Смотрите также

- Получение `ArrayBuffer`s и типизированных массивов из *Base64 кодировки* (en-US)
- `StringView` – библиотека для работы со строками в стиле языка C, основанная на типизированных массивах
- Быстрая работа с пикселями Canvas через типизированные массивы
- Типизированные массивы: Двоичные данные в браузере
- Endianness (en-US)

This page was last modified on 2 дек. 2023 г. by MDN contributors.