НЕТОЛОГИЯ

ARRAYBUFFER

АЛЕКСЕЙ СУДНИЧНИКОВ / ВЕБ-РАЗРАБОТЧИК

АЛЕКСЕЙ СУДНИЧНИКОВ

Веб-разработчик

@avsudnichnikov

ПЛАН ЗАНЯТИЯ

- Предисловие
- Типизированные массивы
- ArrayBuffer
- DataView
- Использование
- "Многопоточность" в JS

ПРЕДИСЛОВИЕ

ПРЕДИСЛОВИЕ

Эта лекция будет больше обзорной, поскольку для использования того, что в ней описано, нам будет необходимо разобрать соответствующие инструменты из Web API.

Но понимание материала, приведённого здесь, критично для следующей лекции и для всех лекций следующего курса, связанной с обработкой данных (а именно с загрузами файлов, работой с изображениями, записью аудио и видео).

Поэтому нам важно, чтобы вы поняли основные концепции.

ТИПИЗИРОВАННЫЕ МАССИВЫ

ТИПИЗИРОВАННЫЕ МАССИВЫ

JS используется не только для работы с объектами, но и для работы с данными, которые представляют из себя наборы байт - изображения, архивы и т.д.

ТИПИЗИРОВАННЫЕ МАССИВЫ

Поскольку Web API предоставляет возможность для выбора файлов с устройства пользователя и манипуляции ими, возникает желание строить приложения, умеющие:

- работать с графикой (фильтры, как в Instagram, ресайз, поворот и т.д.)
- работать с аудио/видео
- архивация/разархивация файлов
- криптография (преобразование на уровне байт)

Для этой цели в ES6 были добавлены специальные типы, которые, которые в общем называются Typed Arrays.

TYPED ARRAYS

Ключевые их характеристики:

- все данные должны быть одного типа (например, целые числа фиксированного размера (в байтах))
- все данные должны быть "примитивного типа" (но не примитивы в терминах JS)
- их размеры фиксированы (хотя есть некоторые исключения)
- данные идут последовательно (без пропусков: в обычных массивах можно сделать "дырки")
- данные расположены в памяти последовательно

Для этой цели в ES6 были добавлены специальные типы, которые, которые в общем называются Typed Arrays.

TYPED ARRAYS

Ключевое: в памяти компьютера есть только байты (8 бит), интерпретация этих байт зависит от того, как мы на них смотрим.

Например, мы можем их интерпретировать их как однобайтовые числа, а можем, как определённые

ТИПЫ TYPED ARRAYS

Доступны следующие представления:

`Int8Array()` - 8-битное число со знаком

`Uint8Array()` - беззнаковое 8-битное число

`Uint8ClampedArray()` - беззнаковое восьмибитное число ("зажатое")

`Int16Array()` - шестнадцитибитное число со знаком

`Uint16Array()` - беззнаковое шестнадцитибитное число

`Int32Array()` - 32-битное число со знаком

`Uint32Array()` - беззнаковое 32-битное число

`Float32Array()` - 32-битное вещественное число

`Float64Array()` - 64-битное вещественное число

`BigInt64Array()` - 64-битное число со знаком

`BigUint8Array()` - беззнаковое 64-битное число

СОЗДАНИЕ TYPED ARRAYS

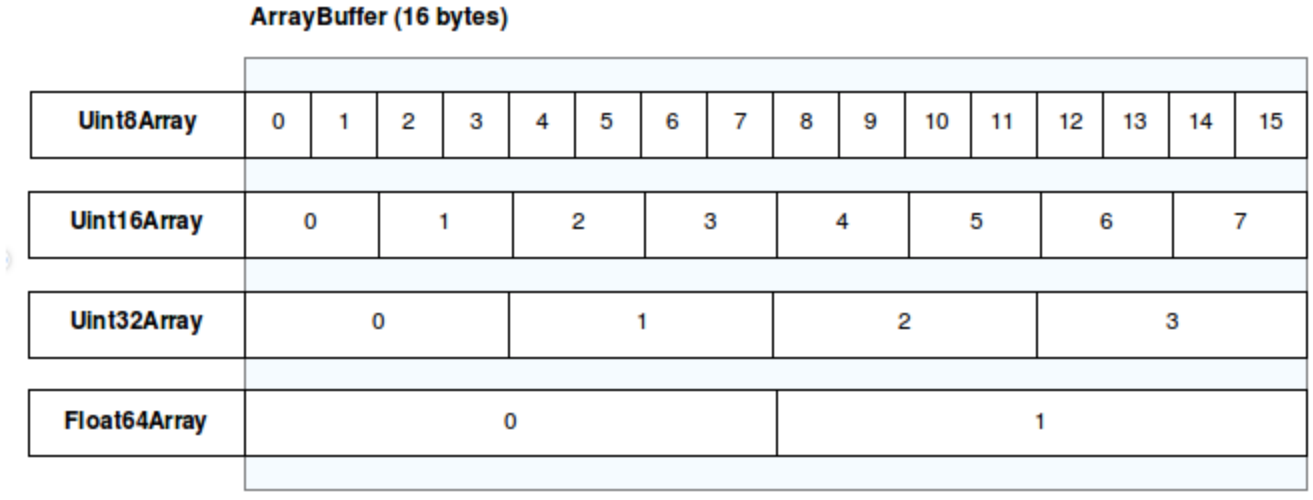
Мы можем создавать Typed Arrays различными способами:

```
// length = 0
const empty = new Uint8Array();
// length = 8
const sized = new Uint8Array(8);
// создание из массиво-подобного объекта*
const fromArrayLike = new Uint8Array([1, 2, 3]);
// так же можно создавать из другого типизированного массива и ArrayBuffer (об этом позже)
```

Под массиво-подобным объектом понимается либо объект, по которому можно итерироваться (пройдём на следующих лекциях), либо объект со свойством `length` и индексированными элементами.

TYPED ARRAYS

Typed Array используют буфферы для хранения данных, так что один и тот же буфер может быть использован разными массивами:



Изображение с MDN

Фактически, Typed Arrays являются представлениями этих данных (то, как мы их интерпретируем).

ARRAYBUFFER

ARRAYBUFFER

ArrayBuffer представляет собой ссылку на поток "сырых" данных.

Например, если создать `ArrayBuffer` длины 4:

```
1 const buffer = new ArrayBuffer(4);
```

Мы получим просто данные следующего вида (в байтах): `0000`

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Эти данные можно представить в разных вариантах: разделив по байту, 2 байта, или четыре.

Чтобы представить `ArrayBuffer` в каком-либо виде, требуется создать его представление:

```
1  const buffer = new ArrayBuffer(4);
2  const buffer8BitView = new Int8Array(buffer);
3
4  console.log(buffer8BitView);
5
6  // -> Int8Array(4)[0, 0, 0, 0];
```

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Этому же буферу можно задать и другое представление:

```
const buffer = new ArrayBuffer(4);
const buffer8BitView = new Int8Array(buffer);
const buffer16BitView = new Int16Array(buffer);

console.log(buffer8BitView);
console.log(buffer16BitView);

// -> Int8Array(4)[0, 0, 0, 0];
// -> Int16Array(2)[0, 0];
```

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

После создания представления можно в `ArrayBuffer` внести данные или изъять из него:

```
const buffer = new ArrayBuffer(4);
const buffer8BitView = new Int8Array(buffer);
const buffer16BitView = new Int16Array(buffer);

buffer16BitView[0] = 1000;

console.log(buffer16BitView[0] + 1);

// -> 1001
```

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Обратите внимание, что доступ к одному представлению не прекращается при использовании другого:

```
1  const buffer = new ArrayBuffer(4);
2  const buffer8BitView = new Int8Array(buffer);
3  const buffer16BitView = new Int16Array(buffer);
4
5  buffer16BitView[0] = 1000;
6
7  console.log(buffer8BitView);
8  console.log(buffer16BitView);
9
10 // -> Int8Array(4) [-24, 3, 0, 0]
11 // -> Int16Array(2) [1000, 0]
```

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Чтоб понять разницу между `Uint8Array` и `Uint8ClampedArray`, давайте проведем следующий опыт:

```
1  const buffer = new ArrayBuffer(2);
2  const notClampedBufferView = new Uint8Array(buffer);
3  const clampedBufferView = new Uint8ClampedArray(buffer);
4
5  console.log('Step 1');
6  notClampedBufferView[0] = 100;
7  clampedBufferView[1] = 100;
8  console.log(notClampedBufferView[0]);
9  console.log(clampedBufferView[1]);
10 console.log('-----');
```

```
console.log('Step 2');
notClampedBufferView[0] += 100;
clampedBufferView[1] += 100;
console.log(notClampedBufferView[0]);
console.log(clampedBufferView[1]);
console.log('-----');

console.log('Step 3');
notClampedBufferView[0] += 100;
clampedBufferView[1] += 100;
console.log(notClampedBufferView[0]);
console.log(clampedBufferView[1]);
console.log('-----');
```

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Что произошло?

Почему такая разница?

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Не стоит забывать, что в ArrayBuffer можно хранить и двоичное представление других (нечисловых) данных:

```
1  const helloStr = 'Hello, world!';
2
3  const buffer = new ArrayBuffer(helloStr.length);
4  const bufferView = new Uint8Array(buffer);
5
6
7  for (let i = 0; i < bufferView.length; i += 1) {
8    bufferView[i] = helloStr.charCodeAt(i);
9  }
10
11 for (let i = 0; i < bufferView.length; i += 1) {
12   console.log(String.fromCharCode(bufferView[i]));
13 }
```

ВАЖНО

Следующие темы этой лекции мы рассмотрим в обзорном режиме: сейчас нам они нужны больше для формирования общей картины.

К практическому их использованию мы перейдём уже на следующем курсе, когда будем манипулировать бинарными объектами в памяти.

DATAVIEW

BYTE ORDER (ENDIANESS)

Когда мы интерпретируем всего один байт (8 бит), то он интерпретируется в соответствии с простыми правилами: тот разряд, что слева - тот и является наименьшим:

```
10 (в двоичной) = 2 в десятичной системе счисления

1 (индекс 1) 0 (индекс 0) - считаем справа-налево, у первого индекс = 0

само преобразование
1 * 2^1 + 1 * 2^0 = 2
```

BYTE ORDER (ENDIANESS)

Но когда интерпретируется последовательность байт не всё так однозначно, есть целых две схемы:

1. big-endian: сначала больший байт, затем меньший байт
2. little-endian: сначала меньший байт, затем больший байт

BYTE ORDER (ENDIANESS)

Самое интересное, что в x86 - little-endian, а при передаче по сети (например, для некоторых данных в TCP) - big-endian.

В большинстве случаев мы этого не видим, но если читать спецификации, то это будет явно указано.

Например, спецификация формата PNG: *All integers that require more than one byte must be in network byte order: the most significant byte comes first, then the less significant bytes in descending order of significance*

DATAVIEW

Объект DataView предоставляет операции для доступа (чтение и запись) к байтам ArrayBuffer без учёта Byte Order:

```
const buffer = new ArrayBuffer(2);
new DataView(buffer).setInt16(0, 256, true); // little-endian - флаг true
console.log(new Int16Array(buffer)[0] === 256); // true
new DataView(buffer).setInt16(0, 256); // big-endian - флаг little-endian не
console.log(new Int16Array(buffer)[0] === 1); // true
```

ИСПОЛЬЗОВАНИЕ

ИСПОЛЬЗОВАНИЕ

В большинстве случаев, вы сами не будете использовать напрямую манипуляции на уровне байт (кроме получения и отправки данных).

Конечно же, если только целенаправленно не зададитесь такой целью (реализовать какие-то фильтры изображений на уровне байт).

Но достаточно много современных библиотек (zipjs, cryptojs и другие) используют это API, поэтому вы должны понимать, как это происходит.

"МНОГОПОТОЧНОСТЬ" В JS

"МНОГОПОТОЧНОСТЬ" В JS

Когда исполняется JS-код, он исполняется в одном потоке.

Что это значит? Это значит, что единовременно может исполняться только одна JS функция.

С одной стороны, это делает программирование на JS проще. С другой стороны накладывает на нас некоторые ограничения.

"МНОГОПОТОЧНОСТЬ" В JS

Например, если запустить какую-то тяжёлую функцию в коде JS (то же самое шифрование), то страница буквально "подвиснет".

Для простоты можно запустить скрипт подсчёта суммы от 0 до нескольких миллиардов в консоли любой загруженной страницы:

```
let sum = 0;
for (let i = 0; i <= 10000000000; i++) {
  sum++;
}
```

"МНОГОПОТОЧНОСТЬ" В JS

На время работы этого скрипта все элементы страницы перестанут реагировать на события (попробуйте покликать по ссылкам).

Всё дело в том, что реагирование на событие (перерисовка страницы) и выполнение JS скриптов происходит в одном и том же потоке: браузер не может перерисовать страницу, пока выполняется ваш код.

А теперь вспомните, сколько архивируются или шифруются большие файлы?

"МНОГОПОТОЧНОСТЬ" В JS

Подобные ограничения приводят к фактической бессмысленности того, что мы изучили, если нет возможности вынести из основного потока подобные вычисления (именно вычисления), а не работу со страницей.

Web Workers - API, позволяющие исполнять скрипты в фоновом режиме (в отдельном потоке) независимо от скриптов, работающих с пользовательским интерфейсом.

Реализация Web Workers есть в браузере, в Node.js есть похожее API, называемое Worker Threads.

WEB WORKERS

Ключевая область применения - выполнение ресурсоёмких/длительных задач, которые сложно выполнять в основном потоке без влияния на скорость отрисовки пользовательского интерфейса, например:

- обработка изображений;
- сортировка/обработка/манипуляция данными

WEB WORKERS

Web Worker’ы не имеют доступа к window, document и DOM, но при этом имеют доступ к:

- таймаутам и интервалам;
- сетевому взаимодействию;
- возможности создавать другие Worker’ы.

WEB WORKERS

"Общение" между основным скриптом и Web Worker'ом передачи сообщений (представьте, что вы общаетесь в Telegram с товарищем) - тут почти также.

Если при передаче сообщение передаются данные (например, файлы), то их передача производится посредством "копирования".

Т.е. изменение данных в Worker'е не повлечёт за собой изменения оригинальных данных (если вы скинули в Telegram коллеге документ, то это копия и изменения в ней не повлияют на ваш документ).

При передаче данных используется алгоритм «Structured Clone», который хоть и является достаточно быстрым, всё равно по факту является созданием копии.

SHAREDARRAYBUFFER

При этом иногда возникает необходимость не передавать данные, а организовать совместное использование: чтение и запись.

В этих случаях предоставляется специальный инструмент, который позволяет передавать блок памяти, доступный обеим сторонам: представьте, что вы с товарищем редактируете один документ в Google Docs (вы не пересылаете его друг другу, а редактируете один и тот же документ), при этом вы обмениваетесь сообщениями (например, через Telegram) по поводу того, что и как нужно сделать.

Примерно так же и работает SharedArrayBuffer.

ИТАК, ПОДВЕДЁМ ИТОГИ

На этой лекции мы с вами поговорили об типизированных массивах и возможностях их использования.

ИНТЕРЕСНОЕ ЧТИВО

ArrayBuffer:

- MDN - Типизованные массивы JavaScript
- Habr - ArrayBuffer и SharedArrayBuffer

НЕТОЛОГИЯ

Спасибо за внимание!!! Жду ваших вопросов 🙂

АЛЕКСЕЙ СУДНИЧНИКОВ

@avsudnichnikov