集合引用类型

定型数组

TODO:还得细看，暂时用不到的知识
ES6新增
typed array
目的是提升向原生库传输数据的效率

历史

想要开发一套JavaScript API，充分利用3D图形API和GPU加速在<canvas>上渲染复杂的图形

1. WebGL
   基于OpenGL ES (OpenGL for Embedded Systems)2.0规范
   因为 JavaScript 数组与原生数组之间不匹配，所以出现了性能问题
   图形API不需要JS默认的双精度浮点格式，WebGL都需要在目标环境分配新数组，迭代转型为适当的格式，花费很多时间。
2. 定型数组
   Mozilla为了解决这个问题实现了CanvasFloatArray
   提供JS接口，C语言风格浮点数值。
   可以直接给底层图形API工具
   CanvasFloatArray=====>Float32Array
   定型数组中可用的第一个"类型"

ArrayBuffer

Float32Array实际上是一种“视图”，可以允许JavaScript运行时访问一块名为ArrayBuffer的预分配内存。
ArrayBuffer是所有定型数组及视图引用的基本单位

SharedArrayBuffer是ArrayBuffer的一个变体，可以无须复制就在执行上下文间传递它

ArrayBuffer()是一个普通的JavaScript构造函数，可用于在内存中分配特定数量的字节空间

```
const buf = new ArrayBuffer(16); // 在内存中分配 16 字节
alert(buf.byteLength); // 16
```

ArrayBuffer一经创建不能调整大小，只能slice()复制全部或部分到新的实例中

```
const buf1 = new ArrayBuffer(16);
const buf2 = buf1.slice(4, 12);
alert(buf2.byteLength); // 8
```

ArrayBuffer特点

- 分配失败时会抛出错误
- 分配的内存不能超过 Number.MAX_SAFE_INTEGER（2^53 - 1）字节
- 声明会将所有二进制位初始化为 0
- ArrayBuffer分配的堆内存可以被当成垃圾回收，不用手动释放

DataView

- 可以读写ArrayBuffer的视图
- 专为文件I/O和网络I/O设计
- API支持缓冲数据的高度控制
- 对缓冲内容没有预设，不能迭代
- 只有在ArrayBuffer读写时才能创建DataView实例。这个实例可以使用全部或部分ArrayBuffer，且维护着对该缓冲实例的引用，以及视图在缓冲中开始的位置

```
const buf = new ArrayBuffer(16);
// DataView 默认使用整个 ArrayBuffer
const fullDataView = new DataView(buf);
alert(fullDataView.byteOffset); // 0
alert(fullDataView.byteLength); // 16
alert(fullDataView.buffer === buf); // true
// 构造函数接收一个可选的字节偏移量和字节长度
// byteOffset=0 表示视图从缓冲起点开始
// byteLength=8 限制视图为前 8 个字节
const firstHalfDataView = new DataView(buf, 0, 8);
alert(firstHalfDataView.byteOffset); // 0
alert(firstHalfDataView.byteLength); // 8
alert(firstHalfDataView.buffer === buf); // true
// 如果不指定，则 DataView 会使用剩余的缓冲
// byteOffset=8 表示视图从缓冲的第 9 个字节开始
// byteLength 未指定，默认为剩余缓冲
const secondHalfDataView = new DataView(buf, 8);
alert(secondHalfDataView.byteOffset); // 8
alert(secondHalfDataView.byteLength); // 8
alert(secondHalfDataView.buffer === buf); // true
```

要通过 DataView 读取缓冲，还需要几个组件。

- 首先是要读或写的字节偏移量。可以看成 DataView 中的某种“地址”。
- DataView 应该使用 ElementType 来实现 JavaScript 的 Number 类型到缓冲内二进制格式的转换。
- 最后是内存中值的字节序。默认为大端字节序。

ElementType

DataView对缓冲区内的数据类型没有预设，需要在读写时指定一个ElementType，然后DataView完成读写转换

ElementType	字　节	说　明	等价的 C 类型	值的范围
Int8	1	8 位有符号整数	signed char	–128~127
Uint8	1	8 位无符号整数	unsigned char	0~255
Int16	2	16 位有符号整数	short	–32 768~32 767
Uint16	2	16 位无符号整数	unsigned short	0~65 535
Int32	4	32 位有符号整数	int	–2 147 483 648~2 147 483 647
Uint32	4	32 位无符号整数	unsigned int	0~4 294 967 295
Float32	4	32 位 IEEE-754 浮点数	float	–3.4e+38~+3.4e+38
Float64	8	64 位 IEEE-754 浮点数	double	–1.7e+308~+1.7e+308

DataView为每种ElementType暴露了get和set方法，使用byteOffset（字节偏移量）定位读写位置

```
// 在内存中分配两个字节并声明一个 DataView
const buf = new ArrayBuffer(2);
const view = new DataView(buf);
// 说明整个缓冲确实所有二进制位都是 0
// 检查第一个和第二个字符
alert(view.getInt8(0)); // 0
alert(view.getInt8(1)); // 0
// 检查整个缓冲
alert(view.getInt16(0)); // 0
// 将整个缓冲都设置为 1
// 255 的二进制表示是 11111111（2^8 - 1）
view.setUint8(0, 255);
// DataView 会自动将数据转换为特定的 ElementType
// 255 的十六进制表示是 0xFF
view.setUint8(1, 0xFF);
// 现在，缓冲里都是 1 了
// 如果把它当成二补数的有符号整数，则应该是-1
alert(view.getInt16(0)); // -1
```

字节序

TODO: 暂时用不上的知识点

边界情况

DataView 完成读、写操作的前提是必须有充足的缓冲区,否则就会抛出 RangeError

```
const buf = new ArrayBuffer(6);
const view = new DataView(buf);
// 尝试读取部分超出缓冲范围的值
view.getInt32(4);
// RangeError
// 尝试读取超出缓冲范围的值
view.getInt32(8);
// RangeError
// 尝试读取超出缓冲范围的值
view.getInt32(-1);
// RangeError
// 尝试写入超出缓冲范围的值
view.setInt32(4, 123);
// RangeError
```

DataView 在写入缓冲里会尽最大努力把一个值转换为适当的类型，后备为 0。如果无法转换，则抛出错误

```
const buf = new ArrayBuffer(1);
const view = new DataView(buf);
view.setInt8(0, 1.5);
alert(view.getInt8(0)); // 1
view.setInt8(0, [4]);
alert(view.getInt8(0)); // 4
view.setInt8(0, 'f');
alert(view.getInt8(0)); // 0
view.setInt8(0, Symbol());
// TypeError
```

定型数组

定型数组是另一种形式的ArrayBuffer视图
概念上与DataView接近
区别:它特定于一种 ElementType 且遵循系统原生的字节序
提供了适用面更广的API 和更高的性能
提高与 WebGL 等原生库交换二进制数据的效率

创建定型数组的方式包括读取已有的缓冲、使用自有缓冲、填充可迭代结构，以及填充基于任意类型的定型数组。另外，通过<ElementType>.from()和<ElementType>.of()也可以创建定型数组

```
// 创建一个 12 字节的缓冲
const buf = new ArrayBuffer(12);
// 创建一个引用该缓冲的 Int32Array
const ints = new Int32Array(buf);
// 这个定型数组知道自己的每个元素需要 4 字节
// 因此长度为 3
alert(ints.length); // 3

// 创建一个长度为 6 的 Int32Array
const ints2 = new Int32Array(6);
// 每个数值使用 4 字节，因此 ArrayBuffer 是 24 字节
alert(ints2.length); // 6
// 类似 DataView，定型数组也有一个指向关联缓冲的引用
alert(ints2.buffer.byteLength); // 24
// 创建一个包含[2, 4, 6, 8]的 Int32Array
const ints3 = new Int32Array([2, 4, 6, 8]);
alert(ints3.length); // 4
alert(ints3.buffer.byteLength); // 16
alert(ints3[2]); // 6
// 通过复制 ints3 的值创建一个 Int16Array
const ints4 = new Int16Array(ints3);
// 这个新类型数组会分配自己的缓冲
// 对应索引的每个值会相应地转换为新格式
alert(ints4.length); // 4
alert(ints4.buffer.byteLength); // 8
alert(ints4[2]); // 6
// 基于普通数组来创建一个 Int16Array
const ints5 = Int16Array.from([3, 5, 7, 9]);
alert(ints5.length); // 4
alert(ints5.buffer.byteLength); // 8
alert(ints5[2]); // 7
```

```
// 基于传入的参数创建一个 Float32Array
const floats = Float32Array.of(3.14, 2.718, 1.618);
alert(floats.length); // 3
alert(floats.buffer.byteLength); // 12
alert(floats[2]); // 1.6180000305175781
```

定型数组的构造函数和实例都有一个 BYTES_PER_ELEMENT 属性，返回该类型数组中每个元素的大小：

```
alert(Int16Array.BYTES_PER_ELEMENT); // 2
alert(Int32Array.BYTES_PER_ELEMENT); // 4
const ints = new Int32Array(1),
 floats = new Float64Array(1);
alert(ints.BYTES_PER_ELEMENT); // 4
alert(floats.BYTES_PER_ELEMENT); // 8
```

如果定型数组没有用任何值初始化，则其关联的缓冲会以 0 填充：

```
const ints = new Int32Array(4);
alert(ints[0]); // 0
alert(ints[1]); // 0
alert(ints[2]); // 0
alert(ints[3]); // 0
```