

# WebAssembly 在 3D 模型解码中的应用

原创 王楠、陈泽槟 字节前端 ByteFE 2023年07月19日 18:15 北京

## 1. 前言

在 Web 3D 渲染场景中，使用建模软件构建好的 3D 模型文件，模型的细节越精细对应的文件体积也会越大，考虑到 Web 下网络传输的影响，往往会对 3D 模型文件应用压缩算法以减少文件体积，此时就需要在 Web 端对经过压缩的模型进行快速高效的解码。WebAssembly 在 CPU 计算密集的场景有着更优的性能，同时结合 Web Worker 一起使用的话，还不会阻塞渲染主线程，非常适合用来实现 3D 模型解码的解码器（以下简称 decoder）。

本文将以 glTF 格式的 3D 模型为例，介绍实际业务场景中是如何应用 WebAssembly 来实现 3D 模型的解码。首先，本文将简要介绍保时捷礼物的业务背景和 glTF 格式相关压缩算法；其次，分别介绍基于 WebAssembly 的 Draco 和 Meshopt 两种压缩算法的解码器实现；最后，在保时捷业务场景中尝试应用 glTF 两种压缩算法，并对 glTF 模型的不同压缩率和模型加载耗时进行了对比。

## 2. 背景

无论在抖音直播礼物的用户调研反馈，还是平台历史玩法礼物喜爱度调查中，自定义礼物的用户喜爱度较高。因此，保时捷自定义礼物由此产生，新增在直播间中允许用户“自定义某款礼物特效”的功能，同时，自定义的颗粒度，可根据特效模型做不同程度拆分，如颜色/组件/背景/模型等等。

保时捷礼物效果展示如下图 1 所示。

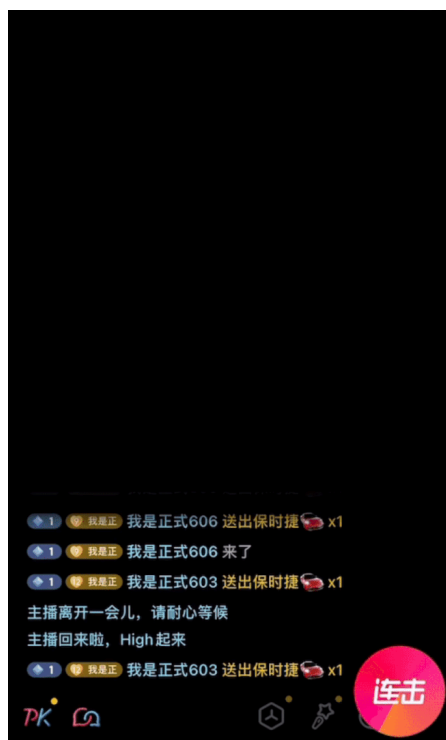


图 1.保时捷礼物特效展示

在抖音直播礼物中，包括“保时捷礼物”在内的多款 2D/3D 礼物均是使用自研渲染引擎进行开发渲染的，由于该引擎在初期尚未支持 glTF 模型格式的加载，当时采用的方案是将场景模型转换为 JSON 产

物，加载 JSON 来渲染 3D 模型。JSON 产物存在包体过大以及不能应用业界通用压缩算法的劣势；而 glTF 是通用的 3D 模型交付格式，并且可以应用通用的压缩算法，缩减资源包体。因此，自研渲染引擎提供 glTF 加载能力，从而使用 glTF 格式模型替换 JSON 模型是进一步提升用户体验的有效手段。

接下来，我们将从 glTF 相关的解码算法，glTF 3D 模型解码框架及收益等方面来进一步介绍基于 WebAssembly 的 3D 模型方案在保时捷场景中的应用。

### 3. glTF 文件及相关压缩算法介绍

#### 3.1 glTF 文件

glTF (GL Transmission Format) 是一种免版税的规范[1]，它主要应用于高效传输和加载 3D 场景和模型。常用 3D 模型建模工具，包括 3DMax、Maya、Blender 等，都是支持直接导出 glTF 格式的模型文件的。

glTF 文件通常包含两部分：JSON 格式的描述文件（. gltf ）和二进制格式的数据文件（. bin ）；其中， gltf 后缀的 JSON 描述文件的 bufferViews 字段指明了对应的 Buffer 数据来自 bin 后缀的二进制数据的哪一部分。此外，glTF 文件还允许引用一些贴图文件（如 . png 、 . jpg 等）用作 3D 模型的纹理，详细规则可参考文档[2]。

## glTF 2.0 API Reference Guide


# glTF - what the ?

An overview of the basics of the GL Transmission Format

**glTF** was designed and specified by the Khronos Group, for the efficient transfer of 3D content over networks.

The core of glTF is a **JSON** file that describes the structure and composition of a scene containing 3D models. The top-level elements of this file are:

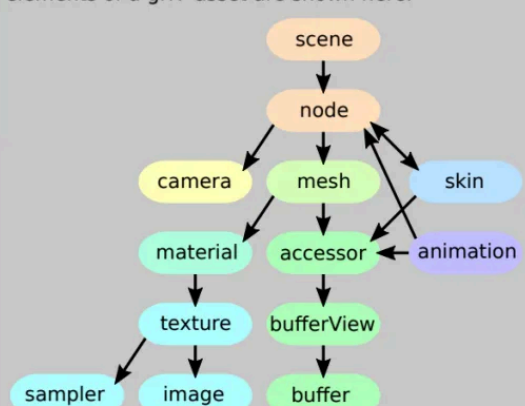
- scenes, nodes**  
Basic structure of the scene
- cameras**  
View configurations for the scene
- meshes**  
Geometry of 3D objects
- buffers, bufferViews, accessors**  
Data references and data layout descriptions
- materials**  
Definitions of how objects should be rendered
- textures, images, samplers**  
Surface appearance of objects
- skins**  
Information for vertex skinning
- animations**  
Changes of properties over time

These elements are contained in arrays. References between the objects are established by using their indices to look up  the objects in the arrays.

It is also possible to store the whole asset in a single binary glTF file. In this case, the JSON data is stored as a string, followed by the binary data of buffers or images.

## Concepts

The conceptual relationships between the top-level elements of a glTF asset are shown here:



```
graph TD
    scene --> node
    node --> camera
    node --> mesh
    node --> skin
    mesh --> material
    mesh --> accessor
    skin --> accessor
    skin --> animation
    material --> texture
    accessor --> bufferView
    texture --> sampler
    texture --> image
    bufferView --> buffer
```

## Binary data references

The images and buffers of a glTF asset may refer to external files that contain the data that are required for rendering the 3D content:

```
"buffers": [
  {
    "uri": "buffer01.bin"
    "byteLength": 102040,
  }
],
"images": [
  {
    "uri": "image01.png"
  }
],
```

The **buffers** refer to binary files (.BIN) that contain geometry- or animation data.

The **images** refer to image files (PNG, JPG...) that contain texture data for the models.

The data is referred to via URIs, but can also be included directly in the JSON using data URIs. The data URI defines the MIME type, and contains the data as a base64 encoded string:

**Buffer data:**

```
"data:application/glTF-buffer;base64,AAABAAIAAgA..."
```

**Image data (PNG):**

```
"data:image/png;base64,iVBORw0K..."
```

Page 1

图 2. glTF 规则示意图

glTF 规范还支持单一文件的二进制格式（. [glb](#)）。目前，glTF 规范最新的版本是 2.0.1，除了基础功能之外，规范里还定义了很多扩展[3]，可以支持更加灵活的数据格式。

针对 glTF 文件，常用的压缩算法有 Draco 和 Meshopt，它们均是以扩展的形式在规范进行定义。下面，我们分别对它们做简要的介绍。

### 3.2 Draco 压缩

Draco 是由 Google 提出的开源的压缩算法，可以针对 Mesh 的 Geometry 中相关的 Attributes 进行压缩，包括 Position、Normal、UV、Color、Joint 等，还可以对 PointCloud 进行压缩，有着非常不错的压缩率，被列入了 Khronos 官方的 glTF 2.0 的扩展列表中，扩展名为 KHR\_draco\_mesh\_compression[4]。Draco 的详细数据格式参考：Draco Bitstream Specification[5]。

Google 对 Draco 开源了 C++ 版的源码[6]，并默认提供了 JavaScript 版和 WebAssembly 版的 Encoder 和 Decoder 的预构建版本，其中 WebAssembly 的版本有着更优的性能（2 倍以上）。另外，也可以自行根据需求用源码去构建定制的版本（例如，JavaScript 版的 Decoder 可以预分配一定的内存以提高大约 2 倍的速度）。

### 3.3 Meshopt 压缩

Meshopt 是社区的产物，可以针对 Mesh 的 Geometry 和 Animation 等数据进行压缩，包括 Vertex、Index、Morph Target 以及 eeyframe 的 Times 和 Values 等，也是 glTF 2.0 的扩展之一，扩展名为：EXT\_meshopt\_compression[7]。

社区内也有 C++ 版本的开源项目：meshoptimizer[8]，可以对 glTF 进行 Meshopt 压缩，并提供了 WebAssembly 版本的 Encoder 和 Decoder；另外还提供了 WebAssembly 版本的 Simplifier，可以对模型进行简化（减面/减点），但要注意 Simplifier 是有损的。

glTF 的 Draco 和 Meshopt 算法及解码器如下表 1 所示。

	Draco	Meshopt
WebAssembly decoder	WebAssembly 版: <a href="#">draco_decoder.wasm(279KB)</a> [9] JavaScript 版: <a href="#">draco_decoder.js(702KB)</a> [10]	SIMD 版: <a href="#">wasm_simd_base64(10KB)</a> [11] Normal 版: <a href="#">wasm_base64(6.5KB)</a> [12]
Feature	压缩率通常更高，但解码消耗大（WebAssembly Decoder 文件体积大）	压缩率通常比 Draco 略低，但 Decoder 更轻量，解码速度更快

表 1. glTF Draco 和 Meshopt 解码器一览

## 4. 基于 glTF 的 3D 模型解码框架

### 4.1 3D 模型解码整体架构

glTF 模型文件对每个场景中的节点数据（Vertex、Index、Normal、UV 等）均采用 Buffer Views 表示。对 glTF 模型文件的压缩本质是对模型文件中 Buffer Views 里的二进制数据的压缩。对于每一个 Buffer View，都可以单独选择是否进行压缩以及应用哪种压缩算法进行压缩。

glTF Loader 对 glTF 模型文件解码过程的如下图 3 所示。

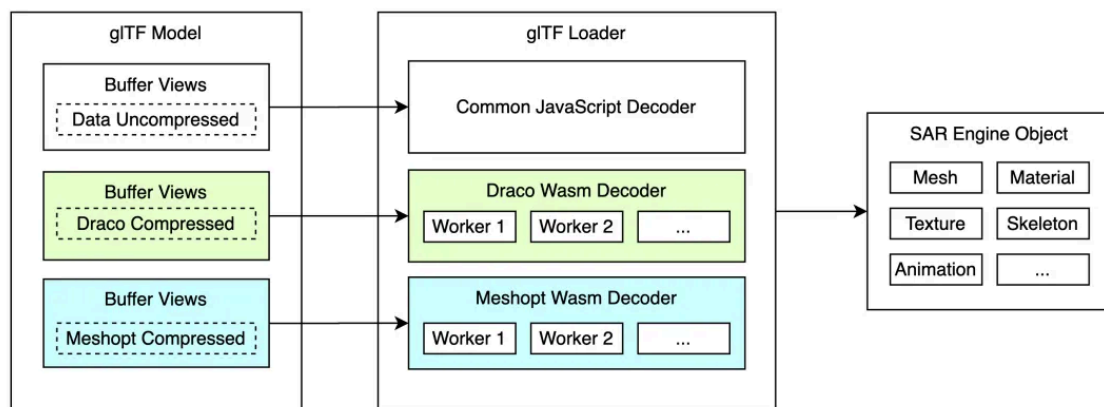


图 3. glTF 3D 模型解码整体架构

如上图 3 所示，glTF 的各个 Buffer View 需要首先识别其数据是否进行了压缩以及采用了进行了哪种压缩算法，然后选择对应的解码器 (Decoder) 进行解码。Buffer Views 往往是相互独立的，所以对 Buffer Views 的解码往往是可以并行进行的，故而复杂的 Decoder 可以放在多个 Worker 中并行进行解码。当完成解码后，glTF 模型文件里的所有数据经由 glTF Loader 转换成了渲染引擎支持的各种内置对象（如 Mesh、Material、Texture、Skeleton、Animation 等）并由渲染引擎进行渲染的。

接下来，我们分别介绍 Draco WebAssembly Decoder 和 Meshopt WebAssembly Decoder 的具体实现。

## 4.2 Draco WebAssembly Decoder

Draco 官方同时提供了 JavaScript 版本和 WebAssembly 版本的解码器，其中对 WebAssembly 的版本还提供了 WebAssembly 的 JavaScript Wrapper，也提供了完善的 API 文档和示例代码，直接引入 JavaScript Wrapper 即可非常方便地集成到项目中。

官方提供的 JavaScript Wrapper 包装了对 WebAssembly 版和 JavaScript 版 decoder 的异步加载能力，可以根据浏览器对 WebAssembly 的支持情况，自动切换所需要加载的 decoder 版本；同时还封装了一些诸如 DecodeBufferToMesh 等方法，只要把 Draco 的 WebAssembly 版、JavaScript 版以及 JavaScript Wrapper 三个文件放在一个目录下，就可以直接使用了。具体如下所示：

- draco\_decoder.js: JavaScript 版本的 decoder。
- draco\_decoder.wasm: WebAssembly 版本的 decoder。
- draco\_WebAssembly\_wrapper.js: Wrapper 详细解释见上。

```
<script src="./draco_WebAssembly_wrapper.js"></script>
<script>
  DracoDecoderModule().then((decoderModule) => {
    // Some buffer data to be decoded.
    const byteArray = new Uint8Array([0]);

    // Create the Draco decoder.
    const buffer = new decoderModule.DecoderBuffer();
    buffer.Init(byteArray, byteArray.length);

    // Create a buffer to hold the encoded data.
```

```

const decoder = new decoderModule.Decoder();
const geometryType = decoder.GetEncodedGeometryType(buffer);

// Decode the encoded geometry.
let outputGeometry;
let status;
if (geometryType == decoderModule.TRIANGULAR_MESH) {
    outputGeometry = new decoderModule.Mesh();
    status = decoder.DecodeBufferToMesh(buffer, outputGeometry);
} else {
    outputGeometry = new decoderModule.PointCloud();
    status = decoder.DecodeBufferToPointCloud(buffer, outputGeometry);
}

// You must explicitly delete objects created from the DracoDecoderModule
// or Decoder.
decoderModule.destroy(outputGeometry);
decoderModule.destroy(decoder);
decoderModule.destroy(buffer);
});
</script>

```

### 4.3 Meshopt WebAssembly Decoder

Meshopt 提供了 SIMD 和非 SIMD 版本的 WebAssembly Decoder。由于 SIMD 具有更好的性能，Meshopt 解码器优先使用 SIMD 版的 WebAssembly Decoder，因此，在解码过程中需要首先检测当前环境是否支持 SIMD，然后为 WebAssembly 解码器创建实例并完成初始化。

#### 4.3.1 检测 SIMD 环境

开源库 `wasm-feature-detect`[13] 里提供了对多种 WebAssembly 特性兼容性的检测方法，其基本原理是：尝试用使用了某个 WebAssembly 特性（如 SIMD、BigInt integration 等）的 WebAssembly 源码创建 WebAssembly 实例，如果创建失败则表示不支持该 WebAssembly 特性，也可以用 `WebAssembly.validate()` 方法来执行检测。

用来检测是否支持 SIMD 的 WebAssembly 文本格式的源码如下：

```

(module
  (func (result v128)
    i32.const 0
    i8x16.splat
    i8x16.popcnt
  )
)

```

WebAssembly 文本格式可以用 `wat2wasm`[14] 工具转换成二进制 WebAssembly 格式，最终简化后的检测代码如下所示。

```
const simd = async () => WebAssembly.validate(new Uint8Array([0, 97, 115, 109, 1, 0, 0, 0, 1, 5,
```

### 4.3.2 WebAssembly 解码器实例化

直接使用 `WebAssembly.instantiate()` 方法，或先 `new WebAssembly.Module()` 再 `new WebAssembly.Instance()`，均可创建 WebAssembly 实例，如下代码所示。

```
private async _createWebAssemblyInstance(WebAssembly: ArrayBuffer): Promise<WebAssembly.WebAssemblyInstance> {
    if (WebAssembly.instantiate instanceof Function) {
        return WebAssembly.instantiate(WebAssembly, {});
    }

    const module = new WebAssembly.Module(WebAssembly);
    const instance = new WebAssembly.Instance(module, {});

    return { module, instance };
}
```

### 4.3.3 Meshopt 异步解码

参照 Meshopt 文档[15], Meshopt decoder 的 API 如下代码所示。

```
decodeVertexBuffer: (target: Uint8Array, count: number, size: number, source: Uint8Array, filter?:
decodeIndexBuffer: (target: Uint8Array, count: number, size: number, source: Uint8Array) => void;
decodeIndexSequence: (target: Uint8Array, count: number, size: number, source: Uint8Array) => void;
decodeGltfBuffer: (target: Uint8Array, count: number, size: number, source: Uint8Array, mode: string,
useWorkers: (count: number) => void;
decodeGltfBufferAsync: (count: number, size: number, source: Uint8Array, mode: string, filter?: string)
```

实际业务中仅使用了 `decodeGltfBufferAsync()`，这个异步方法还包装了对 WebWorker 的支持，可满足所有业务需求，代码如下：



```

/**
 * 异步解析glTF meshopt buffer
 */
public async decodeGltfBufferAsync(count: number, size: number, source: Uint8Array, mode: string,
    if (this._workers.length > 0) {
        return this._decodeWorker(count, size, source, this._decoders[mode], filter === undefined ? undefined : filter);
    }
    const target = new Uint8Array(count * size);
    const func = this._instance.exports[this._decoders[mode]];
    const filterFunc = filter === undefined ? undefined : this._instance.exports[this._filters[filter]];
    this._decode(func, target, count, size, source, filterFunc);
    return target;
}

```

其中 `this._decode()` 方法包装了对 WebAssembly 实例上 `this._instance.exports` 上对应方法的实际调用和后处理逻辑，这里不再放出代码。

#### 4.3.4 Worker 初始化

考虑到对较大的 buffer 进行解码时，可能会消耗较长的时间，此时会阻塞浏览器渲染主线程，容易造成卡顿，此时可以使用 WebWorker 将整个解码的过程放到 worker 线程中去执行。创建执行 WebAssembly decoder 的 WebWorker 的代码如下所示。

```

private _initWorkers(count: number): void {
    const source = `
        let instance;

        const ready = WebAssembly.instantiate(new Uint8Array([${new Uint8Array(this._WebAssemblySource).then(function(result) { instance = result.instance; instance.exports.__WebAssembly_call_ctors;
        self.onmessage = workerProcess;

        function decode(fun, target, count, size, source, filter) {
            let { sbrk } = instance.exports;

            let count4 = (count + 3) & ~3;
            let tp = sbrk(count4 * size);
            let sp = sbrk(source.length);

            let heap = new Uint8Array(instance.exports.memory.buffer);
            heap.set(source, sp);

            let res = fun(tp, count, size, sp, source.length);

            if (res === 0 && filter) {
                filter(tp, count4, size);
            }

            target.set(heap.subarray(tp, tp + count * size));

            sbrk(tp - sbrk(0));

            if (res !== 0) {
                throw new Error("Malformed buffer data: " + res);
            }
        }
    `;
    const worker = new Worker('worker.js');
    worker.postMessage(source);
}

```

```

    }

    function workerProcess(event) {
        ready.then(function () {
            let { data } = event;
            try {
                let target = new Uint8Array(data.count * data.size);
                decode(instance.exports[data.mode], target, data.count, data.size, data.source, instance
                self.postMessage({ id: data.id, count: data.count, action: 'resolve', value: target },
            } catch (error) {
                self.postMessage({ id: data.id, count: data.count, action: 'reject', value: error });
            }
        });
    }
}

`.replace(/\n\s*/g, '');

const blob = new Blob([source], { type: 'text/javascript' });
const url = URL.createObjectURL(blob);

for (let i = 0; i < count; ++i) {
    this._workers[i] = this._createWorker(url);
}

URL.revokeObjectURL(url);
}

```

#### 4.3.5 Worker 异步解码

基于 WebWorker 的 3D 模型异步解码接口如下代码所示。

```

private _decodeWorker(count: number, size: number, source: Uint8Array, mode: string, filter?: str
    let worker = this._workers[0];

    for (let i = 1; i < this._workers.length; ++i) {
        if (this._workers[i].pending < worker.pending) {
            worker = this._workers[i];
        }
    }

    return new Promise((resolve, reject) => {
        const data = new Uint8Array(source);
        const id = this._requestId++;

        worker.pending += count;
        worker.requests[id] = { resolve, reject };
        worker.object.postMessage({ id, count, size, source: data, mode, filter }, [data.buffer]);
    });
}

```



在使用 WebWorker 的 `postMessage` API 进行通信时，传输 buffer（尤其是较大的 buffer）时需要使用 Transferable[16] 来进行传递，这样可以大大减少数据交换时带来额外的复制消耗，相关的原理可参考文档[17]。

## 5. glTF 3D 模型收益

上文我们介绍了 glTF 模型文件压缩算法及其对应解码器基于 WebAssembly 的实现，接下来介绍实际项目中对 3D 模型的收益。考虑到并行加载时性能更优，保时捷项目在上线时采用了分离式（一个模型文件由 .glTF、.bin 和 .png 等文件共同组成）的 glTF 模型文件。

### 5.1 资源大小

针对保时捷的模型，测试时准备了如下 4 种 3D 模型格式：

- 1. JSON+bin：当前线上环境使用的模型，一种自定义的 3D 模型格式，其中的 bin 是二进制数据；
- 2. glTF：原始未压缩的 glTF 模型；
- 3. glTF-draco：经过 draco 压缩后的模型；
- 4. glTF-meshopt：经过 meshopt 压缩后的模型；

具体的 3D 模型资源文件见下表 2。

	JSON+bin	glTF	glTF-draco	glTF-meshopt
bin	1.3 MB	2.4 MB	0.2 MB	0.4 MB
json	1.4 MB	0.05 MB	0.03 MB	0.03 MB
other(png etc.)	-	0.84 MB	0.84 MB	0.84 MB

表 2. 3D 模型资源文件对比一览表

参照上表 2 所示的 3D 模型资源文件对比数据，下图 4 以堆叠柱状图展示了各种模型资源包大小和组成，可以更直观地了解资源包间的差异。

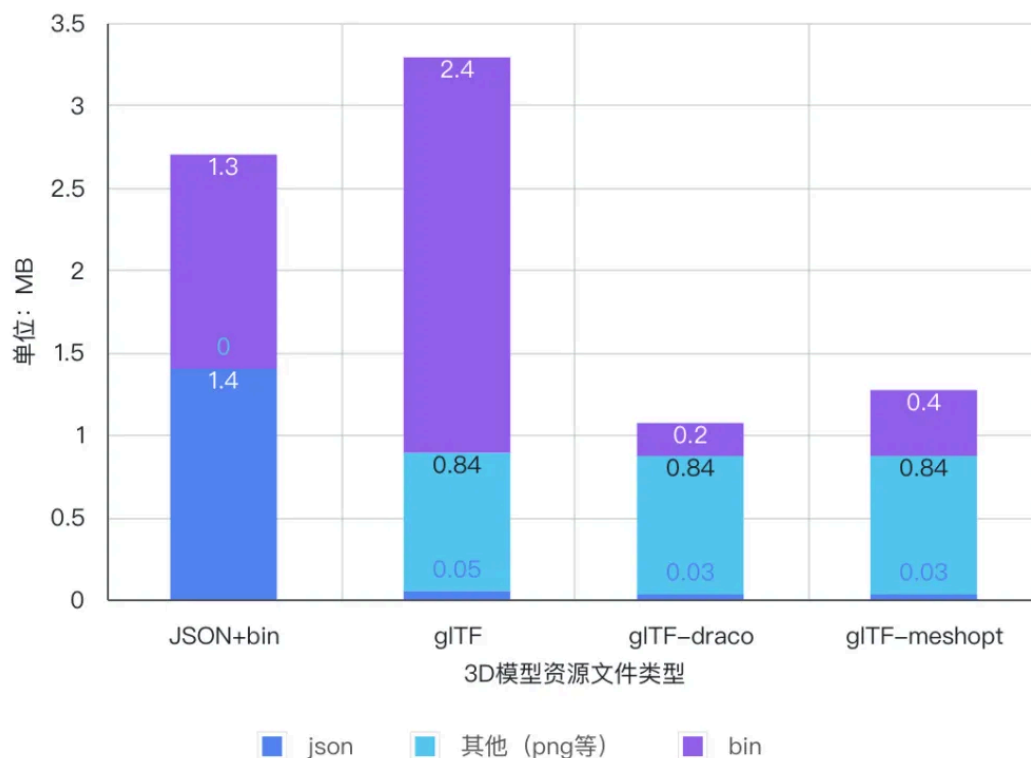


图 4. 3D 模型资源文件体积一览

从图 4 中看出，对 glTF 进行压缩，主要是压缩 bin 文件，而其他资源文件如 png 纹理文件等是没有压缩的。glTF-draco 和 glTF-meshopt 两个经过压缩后的 3D 模型文件的 zip 资源包体积相对 JSON+bin 和未压缩的 glTF 文件都有下降，但二者相差不大，结合表 2 可以计算出二者仅相差 0.1MB。

## 5.2 压缩率

由于线上环境对资源的下发采用的是对所有资源进行 zip 打包的形式，这里的压缩率在计算时是直接对 zip 包进行计算的。对应的压缩率的计算公式为：

压缩率 = glTF 整体资源 zip 包 / JSON+bin 整体资源 zip 包。

	JSON+bin	glTF	glTF-draco	glTF-meshopt
资源 zip 包	2.9 MB	3.3 MB	1.8 MB	1.9 MB

表 3. 资源 zip 包大小对比表

最终的压缩率为：

- glTF-draco 压缩率：1.8 MB / 2.9MB \* 100% = 62%。
- glTF-meshopt 压缩率：1.9 MB / 2.9MB \* 100% = 65.5%。

理论上，draco 压缩算法的压缩率通常比 meshopt 压缩算法更低；从实际的测试数据可以看出，看保时捷模型的 glTF-draco 压缩率略微小于 glTF-meshopt，差距不大。

除了模型包体积和压缩率之外，模型解码还需要评估其加载耗时指标，我们将在 5.3 小节中进行说明。

### 5.3 模型加载耗时

基于保时捷项目工程，最终打包产物在抖音直播测试环境下，使用 iPhone12 测试获取的保时捷礼物 3D 模型加载耗时如下图 5 所示。

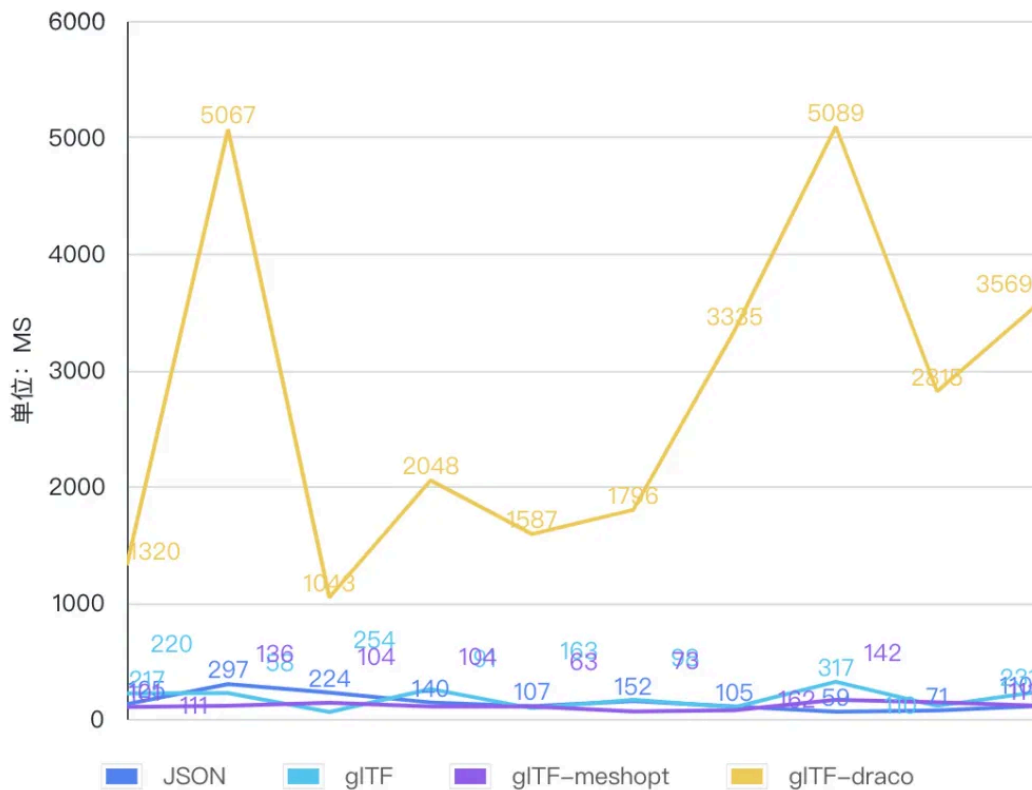


图 5. 3D 模型加载耗时对比

如上图 5 所示，glTF-draco 模型的加载耗时明显高于其他模型，且分布不均衡，方差较大，加载耗时在 1043 ms - 5089 ms 区间。经过分析，造成上述测试数据分布不均衡的原因很可能是测试时使用了抖音测试包在测试环境下进行的，其他可能的影响因素还有 draco 的 wasm decoder 的体积较大，对 wasm decoder 的解析也需要消耗不少时间。

由于，glTF-draco 模型加载耗时过高，造成图 5 中其他模型加载耗时对比不明显，将 glTF-draco 模型去除，如下图 6。其中，glTF、glTF-meshopt 加载耗时相差不大，而 glTF-meshopt 加载耗时波动区间整体低于其他两个模型。

结合 5.1 节中的资源大小对比，glTF-meshopt 分离式模型尺寸仅比 glTF-draco 分离式模型尺寸大 0.1MB，但是加载速度快了 2656ms。

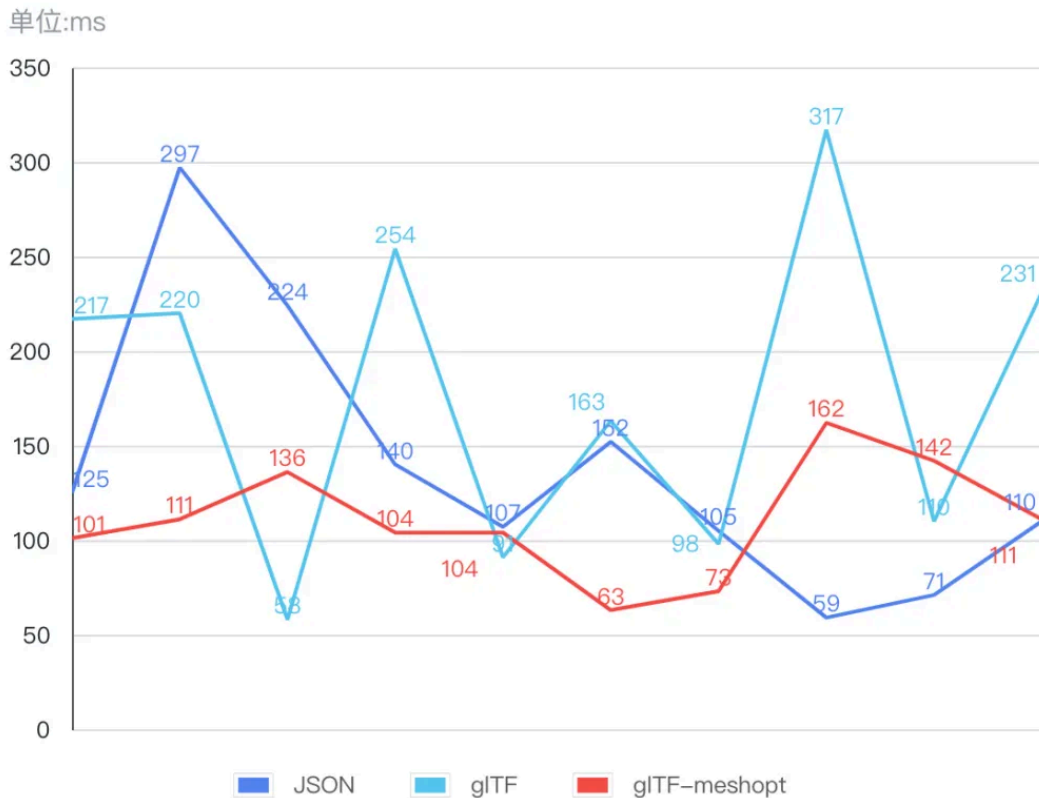


图 6. 模型加载耗时对比(简化)

综合考虑模型的资源大小、压缩率和模型加载耗时三个维度，虽然 glTF-meshopt 模型的压缩率没有 glTF-draco 的低（包体上仅相差 0.1MB），但其加载耗时较低且其 wasm decoder 的文件体积较小，glTF-meshopt 模型作为最终的压缩模型是一个综合最优的解决方案。

## 6. 总结

至此，我们已经介绍了保时捷礼物场景及其模型解码背后使用到的技术，包括 3D 模型加载，3D 模型格式选型，压缩算法选型；同时，基于保时捷礼物场景，深入分析了基于 WebAssembly 的 Draco 和 Meshopt 解码，以及在抖音直播保时捷礼物中可能的资源包体，解码等方面的收益。WebAssembly 仍然处于发展初期，随着其能力的不断提高，特别是 SIMD、多线程、bulk-memory 等性能友好的标准进一步成熟，未来在 3D 应用场景中必将发挥越来越重要的作用。

## 7. 参考文献

- [1]. glTF™ 2.0 Specification: <https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html>
- [2]. glTF 2.0 API Reference Guide: <https://www.khronos.org/files/gltf20-reference-guide.pdf>
- [3]. glTF Extension Registry: <https://github.com/KhronosGroup/glTF/tree/main/extensions>
- [4]. KHR\_draco\_mesh\_compression:  
[https://github.com/KhronosGroup/glTF/blob/main/extensions/2.0/Khronos/KHR\\_draco\\_mesh\\_compression/README.md](https://github.com/KhronosGroup/glTF/blob/main/extensions/2.0/Khronos/KHR_draco_mesh_compression/README.md)
- [5]. Draco Bitstream Specification: <https://google.github.io/draco/spec/>
- [6]. Draco: <https://github.com/google/draco>
- [7]. EXT\_meshopt\_compression:

[https://github.com/KhronosGroup/glTF/blob/main/extensions/2.0/Vendor/EXT\\_meshopt\\_compression/README.md](https://github.com/KhronosGroup/glTF/blob/main/extensions/2.0/Vendor/EXT_meshopt_compression/README.md)

[8]. Meshoptimizer: <https://github.com/zeux/meshoptimizer>

[9]. draco\_decoder.wasm: [https://www.gstatic.com/draco/versioned/decoders/1.5.6/draco\\_decoder.wasm](https://www.gstatic.com/draco/versioned/decoders/1.5.6/draco_decoder.wasm)

[10]. draco\_decoder.js: [https://www.gstatic.com/draco/versioned/decoders/1.5.6/draco\\_decoder.js](https://www.gstatic.com/draco/versioned/decoders/1.5.6/draco_decoder.js)

[11]. wasm\_simd\_base64:

[https://github.com/zeux/meshoptimizer/blob/f926b288264522e1b331a41b07ba40167f396913/js/meshopt\\_decoder.module.js#L9](https://github.com/zeux/meshoptimizer/blob/f926b288264522e1b331a41b07ba40167f396913/js/meshopt_decoder.module.js#L9)

[12]. wasm\_base64:

[https://github.com/zeux/meshoptimizer/blob/f926b288264522e1b331a41b07ba40167f396913/js/meshopt\\_decoder.module.js#L8](https://github.com/zeux/meshoptimizer/blob/f926b288264522e1b331a41b07ba40167f396913/js/meshopt_decoder.module.js#L8)

[13]. wasm-feature-detect : <https://github.com/GoogleChromeLabs/wasm-feature-detect>

[14]. Wabt : <https://github.com/webassembly/wabt>

[15]. meshopt\_decoder.js: <https://github.com/zeux/meshoptimizer/tree/master/js#decoder>

[16]. Transferable objects: [https://developer.mozilla.org/en-US/docs/Web/API/Web\\_Workers\\_API/Transferable\\_objects](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Transferable_objects)

[17]. Transferable objects - Lightning fast: <https://developer.chrome.com/blog/transferable-objects-lightning-fast/>



字节前端 ByteFE

技术前沿&文章分享&实践干货 尽在字节跳动前端团队

247篇原创内容

公众号

🌿 点击上方关注 · 我们下期再见 🌿

## 走进 WebAssembly 的世界



扫码进入专栏阅读完整版



点击左下方“阅读原文”，或扫描上方二维码，进入专栏阅读《走进 WebAssembly 的世界》完整版。

[原创集锦 · 目录](#)

[上一篇](#)

[WebGL大场景性能优化](#)

[下一篇](#)

[面向 WebAssembly 的 ByteReact 框架](#)

[阅读原文](#)