glTF 规范（中文）

CSDN：https://blog.csdn.net/u010447508
Github: https://github.com/ComeformPC

介绍(Introduction)

动机（Motivation）

本节内容是非规范性的

传统的三维模型格式被设计成存储数据进行离线使用，主要是支持桌面系统进行创出工作流程。工业标准的三维数据交换格式允许在不同建模工具之间以及通常在内容流水线中共享资产。但是，这些类型的数据格式均未针对下载速度和运行时快速加载进行优化。文件往往会变得很大，应用程序需要做大量的处理才能加载这些资产到 GPU 加速的应用程序中。

寻求高性能的应用程序很少直接加载建模格式的数据；相反的，他们把离线处理模型作为定制化内容流水线的一部分，将资产转换为针对其运行时程序优化的专有格式。这导致大量分散的不兼容的专有运行时格式，并且在内容创建流水线中重复工作。为一个应用程序导出的 3D 资产必须要还原成原始建模，并且执行另一个专有导出才能在另一个程序中重用。

随着基础移动端和 web 端的 3D 计算的出现，新型应用程序急需一种快速，动态加载的标准化 3D 资产。数字营销解决方案，电子商务产品可视化以及在线模型共享网站只是使用 WebGL 或者 OpenGL ES 构建的连接 3D 应用程序的一小部分。除了高效传输的需求外，许多这些在线应用程序可以受益于标准的可互操作的格式，从而能够在用户之间，应用程序之间以及异构的分布式内容流水线中共享和重用 3D 资产。

glTF 通过提供一种与供应商和运行时无关的格式来解决这些问题，该格式可以以最少的处理量进行加载和渲染。glTF 将易于解析的 JSON 场景描述与一个或多个表示几何，动画或者其他富文本数据的二进制文件结合在一起。二进制数据的存储方式可以将其直接加载到 GPU 缓冲区中，而无需进行额外解析或者其他操作。使用这种方法，glTF 如实地保留了包含节点、网格、相机、材质和动画的完整分层场景，同时实现高效的传输和快速的加载。

glTF 基础（glTF Basics）

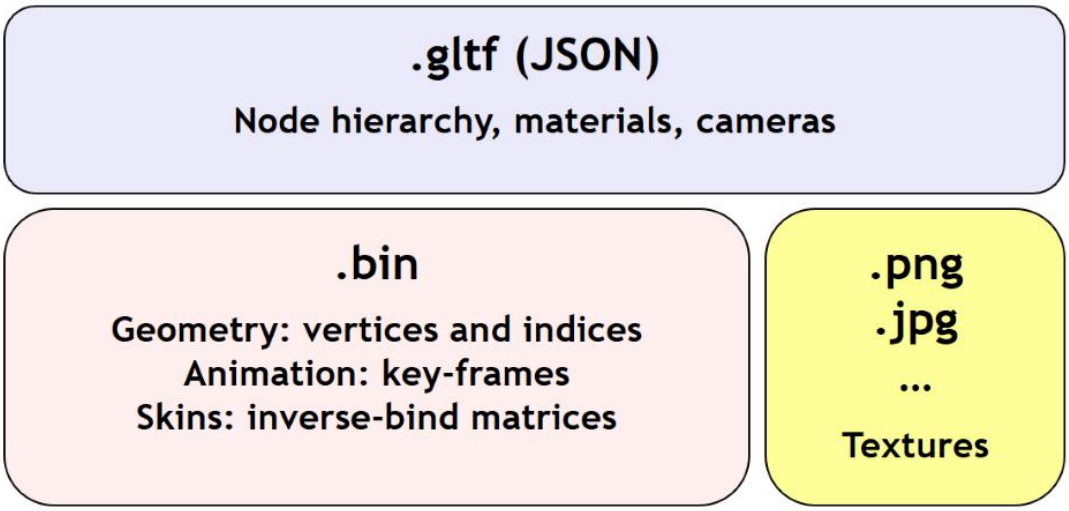
本节内容是非规范性的。

glTF 资产是 JSON 文件，另外还支持外部数据。具体来说，一个 glTF 资产表示为：

- 一个 JSON 格式的文件(.gltf)，包含完整的场景描述：节点层次结构，材质，相机以及网格，动画和其他构造的描述符信息
- 二进制文件（.bin），包含几何和动画数据以及其他基于缓冲区的数据
- 用于纹理的图像文件（.jpg，.png）

以其他格式定义的资产，例如图像，可以存储在通过 URL 引用的外部文件中，并排存储在 GLB 容器中，或者使用 data URIs 直接嵌入到 JSON 中。

有效的 glTF 资产必须指定其版本。



设计目标（Design Goals）

本节内容是非规范性的

glTF 旨在满足以下目标：

- *压缩文件大小*。尽管 Web 开发人员喜欢尽可能多地使用纯文本，但是由于文件过大，纯文本编码对于传输 3D 数据根本不可行。glTF JSON 文件本身是纯文本，但是它紧凑且能快速解析。所有的大数据，例如几何图形和动画，都存储在二进制文件中，它比等效的文本表示形式小得多。

- *快速加载*。glTF 数据结构被设计成尽可能接近 GPU API 数据，无论是 JSON 还是二进制文件，目的就是为了减少加载时间。例如，二进制网格数据可以被当作 JavaScript 数组查看，通过简单的复制就可以直接加载到 GPU 缓冲区中，无需解析和进一步处理。
- *运行时无关*。glTF 对目标应用程序或者 3D 引擎不做任何假设。除渲染和动画外 glTF 没有指定任何运行时行为。
- *完整的 3D 场景表示*。从建模包中导出单个模型不足以用于许多应用程序。通常，作者希望加载整个场景到他们的应用程序中，包括节点，变换，变换层次结构，网格，材质，相机和动画。glTF 努力保留所有的这些信息，以供下游应用程序使用。
- *可扩展性*。尽管最初的基本规范支持丰富的功能集，但是仍然有许多增长和改进的机会。glTF 定义了一种机制，该机制允许添加通用扩展和特定于供应商的扩展。

glTF 的设计采用务实的方法。该格式尽可能接近地匹配 GPU API，但如果仅这样做，就不会有建模工具和运行时系统通常存在的相机，动画或其他功能，并且在转译时会失去很多语义信息。通过支持这些通用结构，glTF 内容不仅能加载和渲染，还可以立即在更广泛的应用程序中使用，并且只需要在内容流水线中做很少的工作。

以下不在最初设计 glTF 的范围内:

- *glTF 不是流格式*。glTF 中的二进制数据本来就可以进行流式传输，并且缓冲区设置允许以增量方式获取数据。但是格式中没有其他流传输构造，并且对于实现流传输数据还是在渲染之前完整下载数据没有一致性要求。
- *glTF 并非人类可读*。尽管以 JSON 表示，但它对开发人员友好。

尽管 glTF 2.0 版本未定义几何图形和其他富文本数据的压缩，但 KHR_draco_mesh_compression 扩展提供了该选项。将来的扩展可能包括纹理和动画数据的压缩方法。

版本控制

次要版本中对 glTF 所做的任何更新都将向后向前兼容。向后兼容将确保任何支持加载 glTF 2.X 资产的客户端实现也能够加载 glTF 2.0 资产。向前兼容允许仅支持 glTF 2.0 的客户端实现加载 glTF 2.X 资产，同时忽略它不了解的任何新功

能。

次要版本更新可以引入新功能，但不会更改任何以前存在的行为。可以在次要版本更新中弃用现有功能，但不会将其删除。

主要版本更新不应与以前的版本兼容。

文件扩展名和 MIME 类型（File Extensions and MIME Types）

*.gltf 文件使用 model/gltf+json

*.bin 文件使用 application/octet-stream

纹理文件使用基础特定图像格式的官方 image/* 类型。为了与现代浏览器兼容，支持以下图像格式：image/jpeg,image/png

JSON 编码（JSON Encoding）

为了简化客户端实现，glTF 对 JSON 格式和编码有额外的限制。

1、JSON 必须使用没有 BOM 的 UTF-8 编码。

实现说明：glTF 导出器不得在 JSON 文本的开头添加字节顺序标记。为了互操作性，客户端实现可以忽略字节顺序标记的存在，而不是将其视为错误。更多信息查看 RFC8259,第八节。

2、本规范中定义的所有字符串（属性名称，枚举）只能使用 ASCII 字符集，并且必须以纯文本形式编写。

实现说明：这允许通用的 glTF 客户端实现不必具有完整的字符集支持。特定于应用程序的字符串（例如，“名称”属性的值或 Extras 字段的内容）可以使用任何符号。

3、JSON 对象中的名称（键）必须唯一，不允许重复的键。

统一资源标识符（URIs）

glTF 使用 URIs 引用缓冲区和图像资源。客户端必须至少支持以下两种 URI 类型：

- Data URIs，将资源嵌入 JSON 中。它们使用 RFC 2397 定义的语法。

*实现说明：*Data URIs 可以使用 JavaScript 进行解码，也可以由 Web 浏览器直接以 HTML 标签使用。

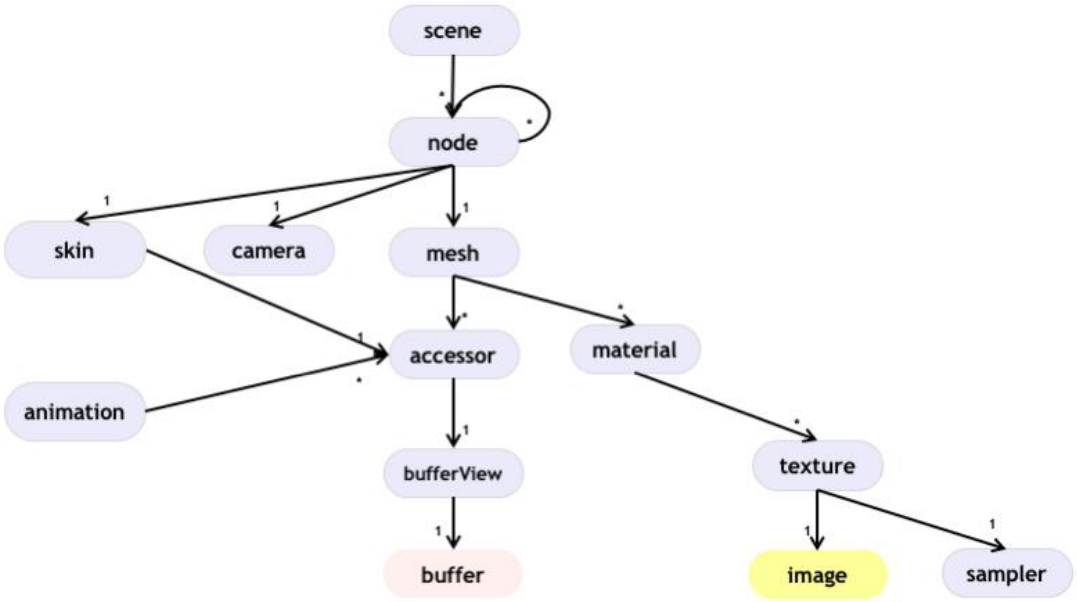
- 相对 URI 路径或者 RFC 3986 节第 4.2 节定义的路径编码-不带方案，权限或参数。保留字符必须按照 RFC 3986 第 2.2 节进行百分比编码。

*实现说明：*客户端可以选择支持其他 URI 组件。例如 http://或 file://结构，授权/主机名，绝对路径以及查询或片段参数。包含这些额外的 URL 组件的资产的可移植性可能较差。

*实现说明：*这允许应用程序决定最佳的传输方式：如果不同的资产共享许多相同的几何形状，动画或文件，则最好使用单独的文件以减少请求的数据总量。使用单独的文件，应用程序可以逐步加载数据，而无需为不可见的模型部分加载数据。如果应用程序更关心单文件部署，则嵌入数据可能是首选方法，即使由于 base64 编码而增加了文件大小并且不支持渐进式或按需加载。或者，资产可以使用 GLB 容器将 JSON 和二进制文件存储到一个文件中，而无需使用 base64 编码。有关详细内容，请参见 GLB 文件格式规范。

*实施说明:*尽管该规范并未明确禁止非标准化 URIs，但在某些平台上，可能不支持使用它们或者导致不希望的副作用，例如安全警告或缓存泄露。

概念（Concepts）



glTF 资产中的顶级结构

资产（Asset）

每个 glTF 资产都必须有 asset 属性。实际上，它是 JSON 成为有效的 glTF 唯

一需要的顶级属性。资产对象必须包含 glTF 版本，该版本信息指定资产的目标 glTF 版本。此外，可选的 minVersion 属性可用于指定加载资产所需的最低 glTF 版本。minVersion 属性允许资产创建器指定客户端实现加载资产必须支持的最低版本。这与 extensionsRequired 概念非常相似，只有客户端支持该指定的扩展时才能加载资产。可以将其他元数据存储在可选属性中，例如生成器或版权信息。例如：

```
   {
       "asset": {
           "version": "2.0",
           "generator":
"collada2gltf@f356b99aef8868f74877c7ca545f2cd206b9d3b7",
           "copyright": "2017 (c) Khronos Group"
       }
   }
```

实现说明: 客户端实现首先检查是否指定了 minVersion 属性并且保证主要和次要版本都能够支持。如果没有指定 minVersion，客户端就应检查 version 属性并保证支持主要版本。客户端加载 GLB 格式同样需要检查 JSON 块中的 minVersion 和 version 属性，因为 GLB 头中指定的版本仅指 GLB 容器版本。

索引和名称（Indices and Names）

glTF 资产的实体由其在对应数据中的索引引用，例如，bufferView 通过指定 buffers 数组中的缓冲区的索引来引用缓冲区。例如：

```
   {
       "buffers": [
           {
               "byteLength": 1024,
               "uri": "path-to.bin"
           }
       ],
       "bufferViews": [
           {
               "buffer": 0,
               "byteLength": 512,
               "byteOffset": 0
           }
       ]
```

```
}
```

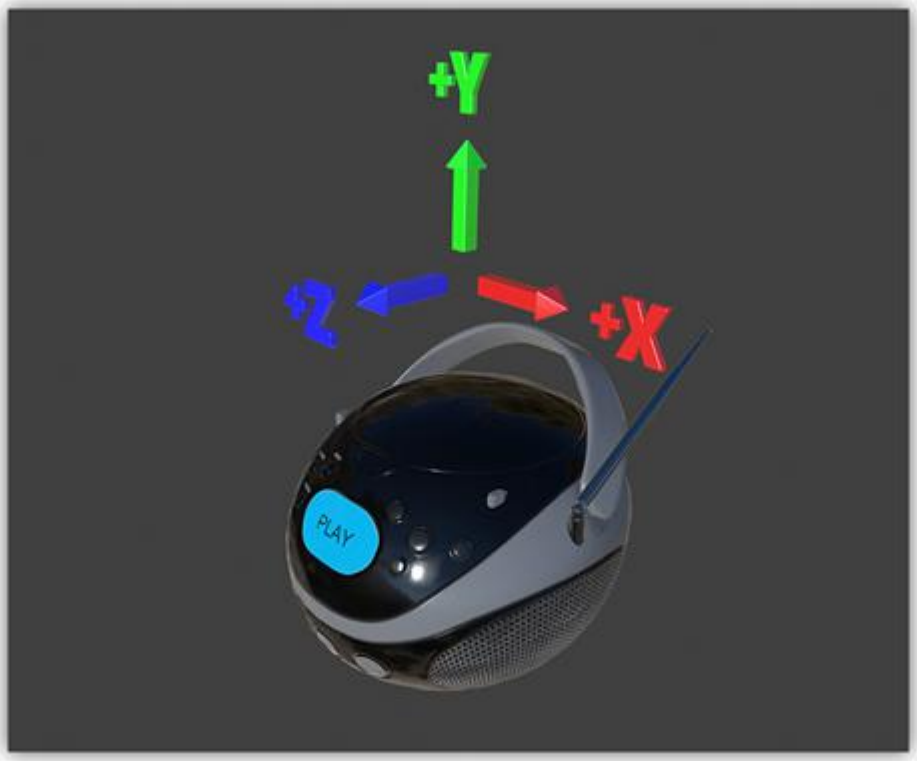
在此示例中，buffers 和 bufferViews 都只有一个元素。bufferView 通过使用 buffer 索引来引用 buffer：”buffer”:0。

索引用于 glTF 内部引用，names 则用于特定于应用程序的用途，例如显示。为此，任何顶级 glTF 对象都可以具有 name 字符串属性。这些属性值不保证唯一，因为他们旨在包含创作资产时创建的值。

对于属性名称，glTF 使用骆驼拼写法 likeThis。骆驼拼写法是 JSON 和 WebGL 中常见的命名约定。

坐标系和单位（Coordinate System and Units）

glTF 使用右手坐标系统，即 X 轴正方向和 Y 轴正方向的叉积就是 Z 轴正方向。glTF 定义 Y 轴正方向朝上。glTF 的前端面向 Z 轴正方向。



所有线性距离的单位均为米。
所有角度均为弧度。
旋转正方向为逆时针方向。
节点变换和动画通道路径是具有以下数据类型和语义的 3D 向量或四元数：

- 平移：包含沿 x,y,z 轴平移距离的 3D 向量
- 旋转：四元数（x,y,z,w），其中 w 是标量
- 缩放：包含沿 x,y,z 轴的缩放比例因子的 3D 向量

RGB 颜色值使用 sRGB 颜色原色。

实施说明：颜色原色定义了颜色模型中每个颜色通道的解释，特别是在 RGB 颜色模型方面。在典型的显示器中，原色描述红色、绿色和蓝色磷光体或滤光片的颜色。ITU-R BT.709 建议书中也定了相同的原色。由于绝大多数当前使用的显示器都使用与默认设置相同的原色，因此客户端实现通常不需要转换颜色值。未来的规范版本或扩展可能会允许其他颜色原色（例如 P3），甚至提供嵌入自定义颜色配置文件。

场景（Scenes）

glTF 资产包含零个或多个 *scenes*，即要渲染的一组视觉对象。场景定义在 scenes 数组中。scene(注意单数)是一个附加属性，用于标识在加载时要显示数组中的哪个场景。

scene.nodes 数组中列出的所有节点都必须是根节点（有关详细信息，请参见下一节）。

当 scene 未定义时，不需要运行时在加载时渲染任何内容。

实施说明：这允许应用程序将 glTF 资产用作单个实体（例如材质或网格）的库。

以下示例定义了具有一个场景的 glTF 资产，其中还包含一个节点。

```
{
    "nodes": [
        {
            "name": "singleNode"
        }
    ],
    "scenes": [
        {
            "name": "singleScene",
            "nodes": [
                0
            ]
        }
    ],
    "scene": 0
}
```

节点和层次（Nodes and Hierarchy）

glTF 资产可以定义节点，即包含要渲染的场景的对象。

节点有一个可选的 name 属性。

节点也具有变换属性，如下一节所述。

节点以父子层次结构组织，非正式地称为节点层次结构。没有父节点的节点称为根节点。

节点层次结构是使用节点的 children 属性定义的，如以下示例所示：

```
{
    "nodes": [
        {
            "name": "Car",
            "children": [1, 2, 3, 4]
        },
        {
            "name": "wheel_1"
        },
        {
            "name": "wheel_2"
        },
        {
            "name": "wheel_3"
        },
        {
            "name": "wheel_4"
        }
    ]
}
```

名为 Car 的节点有四个子节点。这些节点中的每一个都可以由自己的子节点，从而创建了节点层次结构。

说明：为了版本 2.0 一致性，glTF 节点层次结构不是有向无环图（DAG）或场景图，而是严格树的不相交联合。也就是说，没有节点可以是一个以上节点的直接后代。此限制旨在简化实现并促进一致性。

变换（Transformations）

任何节点都可以通过提供 matrix 属性或者任何平移、旋转和缩放属性（也称为 TRS 属性）来定义局部空间变换。平移和缩放是本地坐标系中的 FLOAT_VEC3 值。旋转是局部坐标系中的 FLOAT_VEC4 单位四元数值（x,y,z,w）。

当 maxtrix（矩阵）被定义了，必须将其分解为 TRS。这意味着变换矩阵不能倾斜或剪切。

TRS 属性被转换为矩阵，并以 T*R*S 的顺序相乘以构成变换矩阵；首先将缩放应用于顶点，然后是旋转，最后是平移。

当节点指向动画目标时（由 animation.channel.target 引用），只有 TRS 属性会存在，matrix 不会存在。

实施说明：如果变换的决定因素是负值，则网格三角形的朝向应该反转。这支持模仿几何图形的反向缩放。

实施说明：不可逆转变换（例如将一个轴缩放为零）可能导致照明可见度伪影。

在以下示例中，名为 box 的节点定义了非默认的旋转和平移。、

```
{
"nodes": [
    {
        "name": "Box",
        "rotation": [
            0,
            0,
            0,
            1
        ],
        "scale": [
            1,
            1,
            1
        ],
        "translation": [
            -17.7082,
            -11.4156,
            2.0922
        ]
    }
 ]
}
```

在下一个示例中使用 matrix 属性而不是单独的 TRS 值定义连接相机的节点的变换：

```
{
"nodes": [
    {
        "name": "node-camera",
        "camera": 1,
        "matrix": [
            -0.99975,
            -0.00679829,
            0.0213218,
            0,
```

```
            0.00167596,
            0.927325,
            0.374254,
            0,
            -0.0223165,
            0.374196,
            -0.927081,
            0,
            -0.0115543,
            0.194711,
            -0.478297,
            1
        ]
    }
  ]
 }
```

二进制数据存储（Binary Data Storage）

缓冲区和缓冲区视图（Buffer and Buffer Views）

缓冲区是存储为二进制大对象的数据。缓冲区可以包含几何图形、动画和蒙皮的组合。

二进制大对象允许高效的创建 GUP 缓冲区和纹理，因为它们不需要额外的转换，除了可能的解压缩之外。资产可以拥有任意数量的缓冲区文件，以实现多种应用程序的灵活性。

实施说明：尽管缓冲区大小没有上限，但实现时应注意 JSON 解析器在某些平台上运行时最多只能支持整数到 2^53。另外，将缓冲区存储为 GLB 二进制块时，隐含的限制为 2^32-1 字节。

缓冲区数据为小端模式（指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中）。

所有的缓冲区都存储在资产的 buffers 数组中。

下面的示例定义了一个缓冲区。byteLength 属性指定了缓冲区文件的字节大小。uri 属性指定缓冲区数据的资源地址。缓冲区数据也可以以 base64 编码数据格式存储在 glTF 文件中，并通过 data URI 引用。

```
  {
 "buffers": [
    {
        "byteLength": 102040,
```

```
        "uri": "duck.bin"
      }
     ]
    }
```

bufferView（缓冲区视图）表示缓冲区中的数据子集，它由缓冲区视图中的 byteOffset 属性指定在缓冲区中的字节偏移量，由 byteLength 属性指定字节长度。

当缓冲区视图包含顶点索引或 attribute 属性时，它们必须是缓冲区视图中的唯一内容。即在同一缓冲区视图中具有多种数据是无效的。

实现说明：这允许运行时将缓冲区视图数据上传到 GPU 而无需其他处理。当定义了 bufferView.target，运行时必须根据 target 来决定如何使用数据，或者根据网格访问器对象来推断。

下面的例子中定义了两个缓冲区视图：第一个是 ELEMENT_ARRAY_BUFFER(顶点索引)，其中包含了索引三角形集的顶点索引，第二个是 ARRAY_BUFFER，包含了三角形集的顶点数据。

```
{
"bufferViews": [
    {
        "buffer": 0,
        "byteLength": 25272,
        "byteOffset": 0,
        "target": 34963
    },
    {
        "buffer": 0,
        "byteLength": 76768,
        "byteOffset": 25272,
        "byteStride": 32,
        "target": 34962
    }
 ]
}
```

当缓冲区视图用于顶点属性数据时，它可能有 byteStride 属性。该属性定义了每个顶点之间的间距（以字节为单位）。

缓冲区和缓冲区视图不包含类型信息（在访问器中指定）。它们只需定义元素数据以从文件中检索出来。glTF 文件中的对象（网格，蒙皮，动画）通过访问器访问缓冲区和缓冲区视图。

GLB 存储缓冲区（GLB-stored Buffer）

glTF 资产可以使用 GLB 文件容器将所有资源打包到一个文件。glTF 缓冲区引用 GlB 存储的 BIN 块，必须具有未定义的 buffer.uri 属性，并且它必须是 buffer 数组的第一个元素；BIN 块的字节长度要比 JSON 定义的 buffer.byteLength 大 3 个字节，以满足 GLB 填充要求。任何未定义 buffer.uri 属性(不是 buffers 数组的第一个元素)的 gltf 缓冲区都不会引用 GLB 存储的 BIN 块，并且此类缓冲区的行为也未定义，以适应将来的扩展和规范版本。

实现说明：不需要严格相等的块长度和缓冲区长度，从 glTF 转换到 GLB：应用 GLB 填充后，实现方法不用更新 buffer.byteLength。

下面的例子中，第一个缓冲区对象引用 GLB 储存数据，而第二个则指向外部资源。

```
{
"buffers": [
    {
        "byteLength": 35884
    },
    {
        "byteLength": 504,
        "uri": "external.bin"
    }
 ]
}
```

有关详细信息，请参考 GLB 文件格式规范。

访问器（Accessors）

网格，蒙皮和动画的所有大数据都存储在缓冲区中，并通过访问器进行检索。

accessor(访问器)定义了一种方法用于从缓冲区视图中检索作为类型化数组的数据。访问器指定了分量类型（例如 5126 表示 FLOAT）,当组合在一起时，它们定义每个数组元素的完整数据类型。访问器还使用 byteOffset 和 count 指定 bufferView 中数据的位置。后者(count)指定 bufferView 中的元素数量，而不是字节数。元素可以是例如顶点索引，顶点属性，动画关键帧等。

所有的访问器都存储在资产的 accessors 数组中。

以下片段展示了两个访问器，第一个是用于检索基本图元索引的标量访问器，第二个是用于获取基本图元位置数据的 3 浮点数向量访问器。

```
{
```

```
    "accessors": [
        {
            "bufferView": 0,
            "byteOffset": 0,
            "componentType": 5123,
            "count": 12636,
            "max": [
                4212
            ],
            "min": [
                0
            ],
            "type": "SCALAR"
        },
        {
            "bufferView": 1,
            "byteOffset": 0,
            "componentType": 5126,
            "count": 2399,
            "max": [
                0.961799,
                1.6397,
                0.539252
            ],
            "min": [
                -0.692985,
                0.0992937,
                -0.613282
            ],
            "type": "VEC3"
        }
    ]
}
```


浮点数据（Floating-Point Data）

5126(FLOAT)分量数据类型必须使用 IEEE-754 单精度格式。
不允许使用 NaN,+Infinity,-Infinity 的值。

访问器元素大小（Accessor Element Size）

下表可以用于计算访问器可访问的元素大小。

componentType	Size in bytes
5120 (BYTE)	1
5121(UNSIGNED_BYTE)	1
5122 (SHORT)	2
5123 (UNSIGNED_SHORT)	2
5125 (UNSIGNED_INT)	4
5126 (FLOAT)	4

type	Number of components
"SCALAR"	1
"VEC2"	2
"VEC3"	3
"VEC4"	4
"MAT2"	4
"MAT3"	9
"MAT4"	16

元素大小，以字节为单位，等于（“componentType”的字节大小）*（“type”定义的分量数量）。

例如：

```
{
    "accessors": [
        {
            "bufferView": 1,
            "byteOffset": 7032,
            "componentType": 5126,
            "count": 586,
            "type": "VEC3"
        }
    ]
```

```
}
```

在这个访问器中，分量类型是 FLOAT，因此每个分量是 4 个字节。type 是“vec3”,因此包含 3 个分量。每个元素的大小为 12 个字节（4*3）。

访问器界限（Accessor Bounds）

accessor.min 和 accessor.max 属性是分别包含每个分量的最小值和最大值的数组。导出器和加载器必须将这些值视为与访问器中的 componentType 相同的数据类型，即对整数类型使用整数（不带小数部分的 JSON 数字），5126（FLOAT）使用浮点小数。

实现说明:当 componentType 为 5126（FLOAT）时，JavaScript 客户端实现应将 JSON 解析的双精度浮点数转换为单精度。这可以通过 Math.fround 函数来完成。

尽管并非所有的访问器都需要这些属性，但在某些情况下必须定义最大最小值。有关详细信息，请参阅规范的其他部分。

稀疏存取器（Sparse Accessors）（？）

当描述相对于参考数组的增量更改时，数组的稀疏编码通常比密集编码更节省内存。编码变形目标时通常时这种情况（通常，在变形目标中描述几个位移顶点比传输所有个变形目标顶点更有效）。

glTF2.0 扩展了访问器结构，以实现稀疏数组的有效传输。类似于标准访问器，稀疏访问器根据缓冲区视图中存储的数据初始化类型化元素的数组。最重要的是，稀疏访问器包括一个 sparse 字典，用于描述偏离其初始化值的元素。sparse 字典包含以下必需属性：

- count:位移元素的数量。
- Indices:严格增加 count 大小和特定 componentType 的整数数组，该数组储存偏离初始值的那些元素的索引。
- values:对应于索引数组中的位移元素数组

以下片段显示了一个稀疏访问器的示例，其中有 10 个元素偏离了初始化数组。

```
{
    "accessors": [
        {
            "bufferView": 0,
```

```
            "byteOffset": 0,
            "componentType": 5123,
            "count": 12636,
            "type": "VEC3",
            "sparse": {
                "count": 10,
                "indices": {
                    "bufferView": 1,
                    "byteOffset": 0,
                    "componentType": 5123
                },
                "values": {
                    "bufferView": 2,
                    "byteOffset": 0
                }
            }
        }
    ]
}
```

稀疏访问器与常规访问器的区别在于不需要 bufferView 属性。省略时，稀疏访问器将初始化为大小为零（访问器元素大小）*（访问器位移元素数量）字节的零数组。一旦应用了稀疏替换，稀疏访问器的 min 和 max 属性分别对应于最小和最大的分量值。

如果即没有定义 sparse 也没有 bufferView，则 min 和 max 属性可以具有任何值。这适用于通过外部方式（例如，通过扩展）提供二进制数据的用例。

数据对齐（Data Alignment）

访问器在 bufferView 中的偏移量（accessor.byteOffset）和访问器在缓冲区中的偏移量（accessor.byteOffset+bufferView.byteOffset）必须是访问器 componentType 大小的倍数。

当未定义引用的 bufferView 的 byteStride 时，这意味着访问器元素紧密包装，即有效步幅等于该元素的大小。定义 byteStride 时，它必须是访问器的分量类型大小的倍数。当两个或多个访问器使用同样的 bufferView 时，必须定义 byteStride。

每个访问器必须适配其 bufferView。即 accessor.byteOffset+STRIDE*(accessor.count-1)+SIZE_OF_ELEMENT 必须小于或等于 bufferView.length。

出于性能和兼容性的原因，顶点属性的每个元素必须与 bufferView 内的 4 个字节边界对齐（即 accessor.byteOffset 和 bufferView.byteStride 必须是 4 的倍数）。

矩阵类型的访问器中的数据以列主序存储；每列的开头必须与 4 字节边界对齐。为此，三种 type/componentType 组合需要特殊的布局。

MAT2（2X2 矩阵），1 字节分量

```
| 00| 01| 02| 03| 04| 05| 06| 07|

|===|===|===|===|===|===|===|===|

|m00|m10|---|---|m01|m11|---|---|
```

MAT3（3X3 矩阵），1 字节分量

```
| 00| 01| 02| 03| 04| 05| 06| 07| 08| 09| 0A| 0B|

|===|===|===|===|===|===|===|===|===|===|===|===|

|m00|m10|m20|---|m01|m11|m21|---|m02|m12|m22|---|
```

MAT3（3X3 矩阵），2 字节分量

```
| 00| 01| 02| 03| 04| 05| 06| 07| 08| 09| 0A| 0B| 0C| 0D| 0E| 0F| 10|
11| 12| 13| 14| 15| 16| 17|

|===|===|===|===|===|===|===|===|===|===|===|===|===|===|===|===|===
|===|===|===|===|===|===|===|===|===|===|===|===|===|===|===|===|===|

|m00|m00|m10|m10|m20|m20|---|---|m01|m01|m11|m11|m21|m21|---|---|m02
|m02|m12|m12|m22|m22|---|---|
```

对齐要求仅适用于每一列的开头，因此，如果没有更多数据，可以省略尾随字节。

实现说明：对于 JavaScript,这允许运行时从 glTF 缓冲区或每个缓冲区视图的 ArrayBuffer 有效地创建单个 ArrayBuffer,然后使用访问器将类型化地数组视图（例如 Float32Array）转换为 ArrayBuffer,而无需复制它，因为类型化数组的字节偏移量是该类型大小的倍数（例如，Float32Array 是 4 的倍数）。

考虑以下示例：

```
{
    "bufferViews": [
        {
            "buffer": 0,
            "byteLength": 17136,
```

```
            "byteOffset": 620,
            "target": 34963
        }
    ],
    "accessors": [
        {
            "bufferView": 0,
            "byteOffset": 4608,
            "componentType": 5123,
            "count": 5232,
            "type": "VEC2"
        }
    ]
}
```

通过上面的示例定义的访问二进制数据可以像这样完成:

```
const accessorTypeToNumComponentsMap = {
            'SCALAR': 1,
            'VEC2': 2,
            'VEC3': 3,
            'VEC4': 4,
            'MAT2': 4,
            'MAT3': 9,
            'MAT4': 16};
var typedView = new Uint16Array(buffer, accessor.byteOffset +
accessor.bufferView.byteOffset, accessor.count *
accessorTypeToNumComponentsMap[accessor.type]);
```

访问器分量类型大小为 2 个字节（componentType 是无符号整型）。访问器的 byteOffset 也可以被 2 整除。同样地，访问器针对 0 号缓冲区的偏移量为 5228（620+4068），可以被 2 整除。

几何图形（Geometry）

任何节点都可以包含一个在 mesh 属性中定义的网格。可以参考 skin 对象提供的信息对网格进行蒙皮。网格可以具有变形目标。

网格（Meshes）

在 glTF 中，网格被定义为图元数组。图元对应于 GPU 绘制调用所需的数据。图元指定一个或多个属性，这些属性对应于绘制调用中使用的顶点属性。具有索引的图元还定义了 indices 属性。属性和索引定义为对包含对应数据的访问器的引用。每个图元还指定了材质和 GPU 图元类型相对应的图元类型（例如三角形

集）。

实现说明: 将一个网格划分为多个图元可能有助于限制每个绘制调用的索引数。

如果 material 并未指定，那么将使用默认的材质。

以下示例定义了一个包含一个三角形集的网格：

```
{
"meshes": [
    {
        "primitives": [
            {
                "attributes": {
                    "NORMAL": 23,
                    "POSITION": 22,
                    "TANGENT": 24,
                    "TEXCOORD_0": 25
                },
                "indices": 21,
                "material": 3,
                "mode": 4
            }
        ]
    }
 ]
}
```

每个属性都定义为 attributes 对象中的一个属性。该属性的名称与标识顶点属性的枚举值相对应，例如 POSITION。该属性的值是包含对应数据的访问器的索引。

有效的语义属性名称包括 POSITION， NORMAL,TANGENT,TEXCOORD_0，TEXCOORD_1，COLOR_0，JOINT_0，WEIGHTS_0。特定于应用程序的语义必须以下划线开头，例如：_TEMPERATURE。

每个语义属性的有效访问器类型和分量类型在下方定义。

Name	Accessor Type(s)	Component Type(s)	Description
POSITION	"VEC3"	5126 (FLOAT)	XYZ vertex positions

Name	Accessor Type(s)	Component Type(s)	Description
`NORMAL`	`"VEC3"`	5126 (FLOAT)	Normalized XYZ vertex normals
`TANGENT`	`"VEC4"`	5126 (FLOAT)	XYZW vertex tangents where the *w* component is a sign value (-1 or +1) indicating handedness of the tangent basis
`TEXCOORD_0`	`"VEC2"`	5126 (FLOAT) 5121 (UNSIGNED_BYTE) normalized 5123 (UNSIGNED_SHORT) normalized	UV texture coordinates for the first set
`TEXCOORD_1`	`"VEC2"`	5126 (FLOAT) 5121 (UNSIGNED_BYTE) normalized 5123 (UNSIGNED_SHORT) normalized	UV texture coordinates for the second set
`COLOR_0`	`"VEC3"` `"VEC4"`	5126 (FLOAT) 5121 (UNSIGNED_BYTE) normalized 5123 (UNSIGNED_SHORT) normaliz	RGB or RGBA vertex color

Name	Accessor Type(s)	Component Type(s)	Description
		ed	
JOINTS_0	"VEC4"	5121 (UNSIGNED_BYTE) 5123 (UNSIGNED_SHORT)	See Skinned Mesh Attributes
WEIGHTS_0	"VEC4"	5126 (FLOAT) 5121 (UNSIGNED_BYTE) normalized 5123 (UNSIGNED_SHORT) normalized	See Skinned Mesh Attributes

POSITION 访问器必须定义 min 和 max 属性。

TEXCOORD,COLOR,JOINTS 和 WEIGHTS 属性的语义属性名称必须采用[语义]_[索引]的形式，例如 TEXCOORD_0,TEXCOORD_1,COLOR_0。

客户端实现必须支持至少两个 UV 纹理坐标集，一种顶点颜色和一种关节/权重集。扩展可以添加额外的属性名称，访问器类型或访问器分量类型。

用于索引属性语义的所有索引都必须以 0 开始，并且必须是连续的正整数：TEXCOORD_0,TEXCOORD_1 等。索引不得使用前导零填充数字的数量，并且不需要客户端支持比上述更多的索引语义。

实现说明：每个图元都对应一个 WebGL 绘制调用（当然，引擎可以自由的进行批量绘制调用）。当图元索引属性被定义后，它将引用用于索引数据的访问器，并且使用 GL 的 drawElements 函数。当未定义索引属性时，应使用 GL 的 drawArrays 函数，其 count（顶点）等于 attribute 属性引用的任何访问器的 count 属性（对于给定的图元，他们都相等）。

实现说明：当未指定法线数据时，客户端实现应该计算平面法线。

实现说明：当未指定切线数据时，客户端实现应使用默认的 MikkTSpace 算法计算切线。为了获取最佳结果，还应使用默认的 MikkTSpace 算法处理网格三角形。

实现说明：相同三角形的顶点应具有相同的 tangent.W 值。当相同三角形的顶点具有不同的 tangent.w 值时，切线空间被认为是未定义的。

实现说明：当指定了法线和切线时，客户端实现应通过获取法线和切线 xyz 向量的叉积并乘以切线的 W 分量来计算位切线：bitangent=cross(normal,tangent.xyz)*tangent.w。

变形目标（Morph Targets）

通过扩展网格概念定义变形目标。

变形目标是可变形的网格，其中通过将原始属性和目标属性的加权总和来获得图元的属性。

例如，以这种方式计算图元在索引 i 处的变形目标顶点 POSITION：

```
primitives[i].attributes.POSITION +
  weights[0] * primitives[i].targets[0].POSITION +
  weights[1] * primitives[i].targets[1].POSITION +
  weights[2] * primitives[i].targets[2].POSITION + ...
```

变形目标通过网格 primitives 中定义的 targets 属性实现。Targets 数组中的每个目标都是一个字典，将图元属性映射到包含变形目标位移数据的访问器中。当前，通常仅支持三个属性 POSITION，NORMAL,和 TANGENT。如果变形目标包含特定于应用程序的语义，其名称必须带有下划线（例如_TEMPERATURE）,类似于相关的属性语义。所有图元都必须以相同顺序列出相同数量的 targets。

每个属性语义属性的有限访问器类型和分量类型在下方定义。请注意，在以 TANGENT 数据为目标时，省略了用于惯用性的 w 分量，因为惯用性无法移动。

Name	Accessor Type(s)	Component Type(s)	Description
POSITION	"VEC3"	5126 (FLOAT)	XYZ vertex position displacements
NORMAL	"VEC3"	5126 (FLOAT)	XYZ vertex normal displacements
TANGENT	"VEC3"	5126 (FLOAT)	XYZ vertex tangent displacements

POSITION 访问器必须定义 min 和 max 属性。

所有变形目标的访问器都必须与原始图元的访问器有相同的 count。

变形目标还可以定义一个可选的 mesh.weights 属性，该属性存储默认的目标权重。在没有 node.weights 属性的情况下，使用这些权重来解析图元属性。缺少此属性时，默认目标权重假定为零。

下面的示例通过添加两个变形目标将上一个例子中定义的 Mesh 扩展为变形的网格：

```
{
    "primitives": [
        {
            "attributes": {
                "NORMAL": 23,
                "POSITION": 22,
                "TANGENT": 24,
                "TEXCOORD_0": 25
            },
            "indices": 21,
            "material": 3,
            "targets": [
                {
                    "NORMAL": 33,
                    "POSITION": 32,
                    "TANGENT": 34
                },
                {
                    "NORMAL": 43,
                    "POSITION": 42,
                    "TANGENT": 44
                }
            ]
        }
    ],
    "weights": [0, 0.5]
}
```

将变形目标应用于顶点位置和法线后，可能需要重新计算切线空间。有关详细信息，请参见附录 A。

实现说明：glTF 中并未限制变形目标的数量。一致的客户端实现必须支持至少 8 个变形属性。这意味着它必须支持至少八个包含 POSITION 属性的变形目标，或者四个包含 POSITION 和 NORMAL 属性的变形目标，或者两个包含 POSITION，NORMAL 和 TANGENT 属性的变形目标。对于包含更多变形属性的资产，渲染器可能选择完全支持他们，或仅使用权重最高的八个变形目标的属性。

实现说明：大量的创作和客户端实现将名称与变形目标相关联。尽管 glTF2.0 规范当前不支持提供指定名称的方法，但是大多数工具为此目标使用字符串数组 mesh.extras.targetName。

蒙皮（Skins）

所有的蒙皮信息都存储在资产的 skins 数组中。每个蒙皮都由 inverseBindMatrices 属性定义（该属性指向具有 IBM 数据的访问器），用于将要蒙皮的坐标带到与每个关节相同的空间中；joints 数组属性列出了用作关节以对皮肤进行动画处理的节点索引。关节的顺序在 skin.joints 数组中定义，并且必须与 inverseBindMatrices 数据的顺序匹配。skeletion 属性（如果存在）指向作为关节层次结构的公共根的节点或公共根的直接或间接父节点。

实施说明：定义如何摆放与关节一起使用的蒙皮的几何图形的矩阵（绑定形状矩阵）应先乘以网格数据或反向绑定矩阵。

实施说明：客户端实现应仅将骨架根节点的变换应用于蒙皮网格，而忽略蒙皮网格节点的变换。在下面的示例中，将应用 node_0 的平移和 node_1 的缩放，而忽略 node_3 的平移和 node_4 的旋转。

```
{
    "nodes": [
        {
            "name": "node_0",
            "children": [ 1 ],
            "translation": [ 0.0, 1.0, 0.0 ]
        },
        {
            "name": "node_1",
            "children": [ 2 ],
            "scale": [ 0.5, 0.5, 0.5 ]
        },
        {
            "name": "node_2"
        },
        {
            "name": "node_3",
            "children": [ 4 ],
            "translation": [ 1.0, 0.0, 0.0 ]
        },
        {
            "name": "node_4",
            "mesh": 0,
```

```
            "rotation": [ 0.0, 1.0, 0.0, 0.0 ],
            "skin": 0
        }
    ],
    "skins": [
        {
            "name": "skin_0",
            "inverseBindMatrices": 0,
            "joints": [ 1, 2 ],
            "skeleton": 1
        }
    ]
}
```

蒙皮网格属性（Skinned Mesh Attributes）

蒙皮网格是用顶点属性定义的，这些属性在顶点着色器的蒙皮计算中使用。JOINTS_0 属性数据包含了影响顶点的相应关节数组中关节的索引。WEIGHTS_0 属性数据定义权重，该权重指示关节应多大程度影响顶点。以下网格蒙皮定义了一个三角形网格图元的 JOINTS_0 和 WEIGHTS_0 顶点属性：

```
{
    "meshes": [
        {
            "name": "skinned-mesh_1",
            "primitives": [
                {
                    "attributes": {
                        "JOINTS_0": 179,
                        "NORMAL": 165,
                        "POSITION": 163,
                        "TEXCOORD_0": 167,
                        "WEIGHTS_0": 176
                    },
                    "indices": 161,
                    "material": 1,
                    "mode": 4
                }
            ]
        }
    ]
}
```

影响一个顶点的关节数限制为每组 4 个，因此引用的访问器必须具有 VEC4 类型和以下分量格式：

- JOINTS_0:UNSIGNED_BYTE or UNSIGNED_SHORT
- WEIGHTS_0:FLOAT ， or 归一化 UNSIGNED_BYTE ， or 归一化 UNSIGNED_SHORT

每个顶点的关节权重必须为非负值，并进行归一化以具有 1.0 的线性总和。对于给定的顶点，任何关节的权重均不得超过一个。

如果任何一个顶点受到四个以上关节的影响，则会在后续集合中找到其他关节和权重信息。例如，如果存在 JOINTS_1 和 WEIGHTS_1,它将引用访问器以获取最多 4 个影响顶点的附加关节。请注意，客户端实现仅需要支持最多四个权重和关节的单个集合，但是不支持文件中存在的所有权重和关节集合可能会对模型的动画产生影响。

所有的关节值必须在蒙皮的关节范围内。未使用的关节值（即权重为零的关节）应设置为零。

关节层次结构（Joint Hierarchy）

用于控制蒙皮网格姿势的关节层次结构只是 glTF 节点层次结构，每个节点都指定为关节。每个蒙皮的关节必须具有共同的根，该根可能是也可能不是关节节点本身。当场景中的节点引用外观时，公共根必须属于同一场景。

有关顶点蒙皮实现的更多详细信息，请参阅 glTF Overview。

实现说明：节点定义未指定是否应将节点视为关节。客户端实现可能希望首先遍历 skins 数组，编辑每个关节节点。

实现说明：关节可能具有相关联的规范节点，甚至是带有网格的完整节点子图。它通常用于将整个几何体附着到关节上，而不会被关节蒙皮。（例如附在手关节上的剑）。请注意，节点变换是节点相对于关节的局部变换，就像“变换”章节中描述的 glTF 节点层次结构中的任何其他节点一样。

实例化（Instantiation）

网格由 node.mesh 属性实例化。多个节点可以使用同一网格，而这些节点可能具有不同的变换。例如：

```
{
    "nodes": [
        {
            "mesh": 11
```

```
        },
        {
            "mesh": 11,
            "translation": [
                -20,
                -1,
                0
            ]
        }
    ]
}
```

变形目标是使用以下方式在节点内实例化的：

- mesh 属性中引用的变形目标。
- 变形目标权重会覆盖网格属性中引用的变形目标的权重。以下示例使用非默认权重来实例化变形目标。

```
{
    "nodes": [
        {
            "mesh": 11,
            "weights": [0, 0.5]
        }
    ]
}
```

使用节点的网格和蒙皮属性的组合在节点内实例化蒙皮。蒙皮示例的网格在 mesh 属性中定义。skin 属性包含实例化的蒙皮的索引。

```
{
    "skins": [
        {
            "inverseBindMatrices": 29,
            "joints": [1, 2]
        }
    ],
    "nodes": [
        {
            "name":"Skinned mesh node",
            "mesh": 0,
            "skin": 0
        },
        {
            "name":"Skeleton root joint",
            "children": [2],
            "rotation": [
                0,
```

```
                0,
                0.7071067811865475,
                0.7071067811865476
            ],
            "translation": [
                4.61599,
                -2.032e-06,
                -5.08e-08
            ]
        },
        {
            "name":"Head",
            "translation": [
                8.76635,
                0,
                0
            ]
        }
    ]
}
```

纹理数据（Texture Data）

glTF 将纹理读取分为三个不同的对象类型：纹理，图片和采样器。

纹理（Textures）

所有的纹理都存储在资产的 textures 数组中。纹理由图像资源定义，并由 source 属性（images 数组中的索引）和采样器索引（sampler）表示。

实现说明：glTF2.0 只支持二维纹理。

图片（Images）

纹理引用的图像存储在资产的 images 数组中。

每个图像包含以下其中一个：

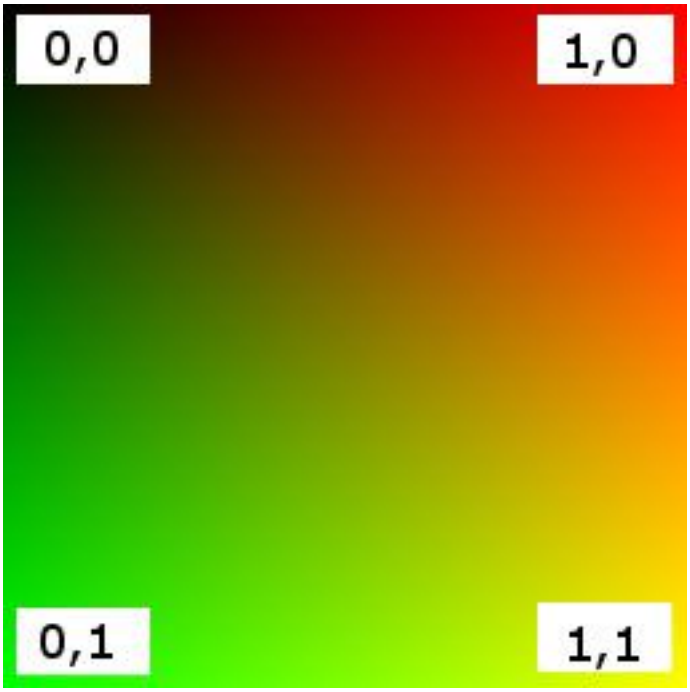
- 指向一个受支持的图像格式的外部文件 URL
- 嵌入 base64 编码数据的 URL
- 对 bufferview 的引用；同时还必须指定 mimeType

下面的示例展示一个指向外部 PNG 图片文件的图像，以及另一个引用带有图像数据的 bufferView 的图像。

```
{
    "images": [
        {
            "uri": "duckCM.png"
        },
        {
            "bufferView": 14,
            "mimeType": "image/jpeg"
        }
    ]
}
```

实现说明：当图像数据由 uri 和 mimeType 提供时，客户端实现应首先 JSON 定义的 MIME Type 而不是传输层提供的 MIME Type。

UV 坐标的原点（0,0）对应与纹理图像的左上角。下图对此进行了说明，其中显示了归一化 UV 空间的所有四个角各自的 UV 坐标。



必须忽略来自 PNG 或 JPEG 容器的任何色彩空间信息。有效的传输函数由引用图像的 glTF 对象定义。

实现说明：由于并非所有图像解码库都完全支持自定义颜色转换，因此这增加了资产的可移植性。为了实现正确的渲染，WebGL 运行时必须通过将 UNPACK_COLORSPACE_CONVERSION_WEBGL 标志设置为 NONE 来禁用此类转换。

采样器（Samplers）

采样器存储在资产的 samplers 数组中。每个采样器指定与 GL 类型相对应的

过滤器和包装选项。以下示例定义了具有线性 mag 滤波，线性 mipmap min 滤波以及 S（U）和 T（V）边界重复采样器。

```
{
    "samplers": [
        {
            "magFilter": 9729,
            "minFilter": 9987,
            "wrapS": 10497,
            "wrapT": 10497
        }
    ]
}
```

默认过滤实现说明：定义过滤选项后，运行时必须使用它们。否则，可以自由调整过滤以适应性能或质量目标。

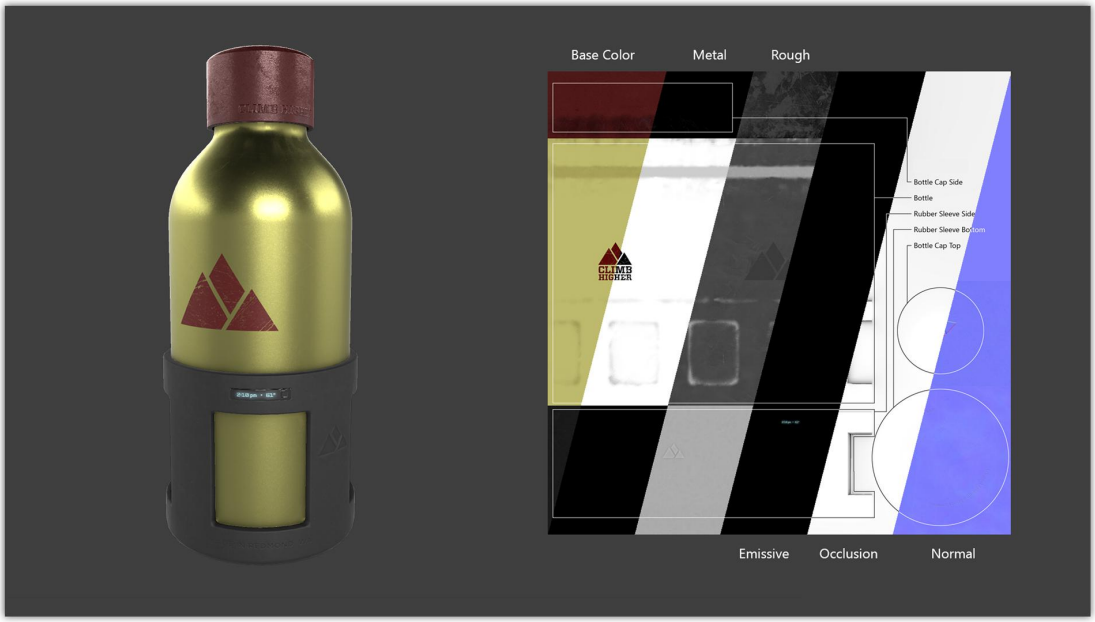
MipMapping 实现说明：当采样器的缩小过滤器（minFilter）使用纹理细分（NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, or LINEAR_MIPMAP_LINEAR）时，任何引用该采样器的纹理都需要进行纹理细分，例如，通过调用 GL 的 generateMipmap()函数。

非二次幂纹理实现说明：glTF 不保证纹理的尺寸是 2 的幂。在运行时，如果纹理的宽度或高度不是 2 的幂，则需要调整纹理的大小，将其尺寸调整为 2 的幂，以适应具有如下内容的采样器：

- 具有 REPEAT 或 MIRRORED_REPEAT 的包装模式（wrpS 或 wrapT）
- 使用缩小过滤器（minFilter），使用纹理细分（NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, or LINEAR_MIPMAP_LINEAR）

材质（Materials）

glTF 使用基于物理渲染（PBR）的广泛使用的材质表示的一组通用参数来定义材质。具体而言，glTF 使用金属粗糙度材质模型。使用这种材质的声明性表示，可以使 glTF 文件在各个平台之间一致地渲染。



金属粗糙度材质（Metallic-Roughness Material）

与金属粗糙度材质模型有关地所有参数均在 material 对象的 pbrMatallicRoughness 属性下定义。以下示例展示了如何使用金属粗糙度参数定义诸如金的材质。

```
{
    "materials": [
        {
            "name": "gold",
            "pbrMetallicRoughness": {
                "baseColorFactor": [ 1.000, 0.766, 0.336, 1.0 ],
                "metallicFactor": 1.0,
                "roughnessFactor": 0.0
            }
        }
    ]
}
```

金属粗糙度材质模型由以下属性定义：

- baseColor - 材质的基础颜色
- metallic - 材质的金属性
- Roughness - 材质的粗糙度

根据金属性的不同，基础颜色有两种不同的解释。当材质是金属时，基色是法向入射（F0）时特定的反射率测量值。对于非金属，基色表示材质的漫反射颜色。在此模型中，不可能为非金属指定 F0 值，而是使用 4%（0.04）的线性值。

可以使用因子或纹理定义每个属性（baseColor,metallic,roughness）的值。metallic 和 roughness 属性打包在单个纹理中，被称为 metallicRoughnessTexture。如果未提供纹理，则假定此材质模型中的所有各个纹理分量值均为 1.0。如果同时存在因子和纹理，则因子值充当相应纹理值的线性乘数。baseColorTexture 使用 sRGB 传输函数，并且在用于任何计算之前必须将其转换为线性空间。

例如，假设从 RGBA 格式的 baseColorTexture 中获取线性空间中的一个值为[0.9,0.5,0.3,1.0]，并建设 baseColorFactor 值为[0.2,1.0,0.7,1.0]。然后，结果将是：

```
[0.9 * 0.2, 0.5 * 1.0, 0.3 * 0.7, 1.0 * 1.0] = [0.18, 0.5, 0.21, 1.0]
```

以下等式显示了如何根据金属粗糙度材质的特性来计算双向反射率分布函数（BRDF）输入（Cdiff,FO,α）。除了材质属性外，如果图元使用属性语义属性 COLOR_0 指定了顶点颜色，则此值将作为 baseColor 的附加线性乘数。

```
const dielectricSpecular = rgb(0.04, 0.04, 0.04)
const black = rgb(0, 0, 0)
```

c_{diff} = lerp(baseColor.rgb * (1 - dielectricSpecular.r), black, metallic)

F_0 = lerp(dieletricSpecular, baseColor.rgb, metallic)

α = roughness ^ 2

所有实现都应对 BRDF 输入使用相同的计算。BRDF 本身的实现可能会根据设备性能和资源限制而有所不同。有关 BRDF 计算的更多详细信息，请参见附录 B。

附加贴图（Additional Maps）

材质定义还提供了其他贴图，这些贴图也可以与金属粗糙度材质模型以及通过 glTF 扩展提供的其他材料模型一起使用。

材质定义了以下其他贴图：

- normal:切线空间法线贴图。
- occlusion:遮挡贴图指示间接照明的区域。
- emissive:自发光贴图控制材质发射的光的颜色和强度。

以下示例展示了使用 pbrMetallicRoughness 参数及其他纹理贴图定义的材质：

```
{
    "materials": [
        {
```

```
            "name": "Material0",
            "pbrMetallicRoughness": {
                "baseColorFactor": [ 0.5, 0.5, 0.5, 1.0 ],
                "baseColorTexture": {
                    "index": 1,
                    "texCoord": 1
                },
                "metallicFactor": 1,
                "roughnessFactor": 1,
                "metallicRoughnessTexture": {
                    "index": 2,
                    "texCoord": 1
                }
            },
            "normalTexture": {
                "scale": 2,
                "index": 3,
                "texCoord": 1
            },
            "emissiveFactor": [ 0.2, 0.1, 0.0 ]
        }
    ]
}
```

实现说明：如果客户端实现资源受限，并且不能支持定义的所有贴图，则它应按以下优先级顺序支持这些其他映射。资源受限的实现应将贴图从底部拖放到顶部。

Map	Rendering impact when map is not supported
Normal	几何图形看起来不如创作的那么详细
Occlusion	在应该更暗的区域中，模型将显得更亮
Emissive	带灯的模型将不会点亮。例如，汽车模型的前灯将关闭而不是打开

Alpha 覆盖（Alpha Coverage）

alphaMode 属性定义如何解释主要因子和纹理的 alpha 值。alpha 值在针对金属粗糙度材质模型的 baseColor 中定义。

alphaMode 可以是以下值之一：

- OPAQUE - 渲染的输出是完全不透明的并且任何 alpha 值都将被忽略。
- MASK - 渲染的输出是完全不透明的还是完全透明的，具体取决于 alpha 值和指定的 alpha 截止值。此模式用于模拟几何形状，例如树叶或铁丝网。
- BLEND - 使用常规绘画操作（即，Porter 和 Duff 运算）将渲染的输出和背景进行融合。此模式用于模拟几何形状，例如纱布或者动物毛皮。

当 alphaMode 设置为 MASK 时，alphaCutoff 属性指定截止阈值。如果 alpha 值大于或等于 alphaCutoff 值，则将其渲染为完全不透明，否则将其渲染为完全透明。其他模式将忽略 alphaCutoff 值。

实时光栅化器实现说明: 实时光栅化器通常使用深度缓冲区和网格排序来支持 alpha 模式。下面介绍这些类型的渲染器的预期行为。

- OPAQUE - 为每个像素写入一个深度值，并且不需要网格排序即可正确输出。
- MASK - 不会为在 alpha 测试后丢弃的像素写入深度值。为所有其他像素写入深度值。不需要网格排序即可正确输出。
- BLEND - 对这种模式的支持各不相同。没有适用于所有情况的完美、快速的解决方案。客户端实现应尝试在尽可能多的情况下实现正确的混合输出。例如，实现可以丢弃具有零或接近零的 alpha 值的像素，以避免排序问题。

双面（Double Sided）

doubleSided 属性指定材料是否为双面。如果该值为 false，则启用背面剔除。如果该值为 true，则禁用背面剔除并启用双面照明。在计算照明方程之前，必须使背面的法线反转。

默认材质（Default Material）

当网格未指定材质时使用的默认材质被定义为未指定属性的材质。材质的所有默认值均适用。请注意，除非场景中存在某些照明，否则该材料不会发光，并且将是黑色的。

点和线材质（Point and Line Materials）

本节是非规范性的。

目前，该规范尚未定义非三角形图元（例如 POINTS 或 LINES）的大小和样式，应用程序可能使用各种技术适当地渲染这些图元。但是，为了保持一致性，提供了以下建议：

- POINTS 和 LINES 应在视口空间中的宽度应该为 1px。
- 对于具有 NORMAL 和 TANGENT 属性的 LINES，请使用包括法线贴图的标准照明进行渲染。
- 对于不具有 TANGENT 属性的 POINTS 或 LINES，使用标准照明进行渲染，带忽略材质上的任何法线贴图。
- 对于没有 NORMAL 属性的 POINTS 或 LINES，不要计算光照，而是为绘制的每个像素输出 COLOR 值。

相机（Cameras）

相机定义了从视图到裁剪坐标系的投影矩阵。投影可以是透视投影或正交投影。相机包含在节点中，因此可以进行变换。它们的世界空间变换矩阵用于计算视图空间变换。相机的定义是，局部正向 X 轴朝右，镜头朝 Z 轴负方向，相机顶部与 Y 轴正方向对齐。如果未指定任何变换，则相机位置位于原点。

相机存储在资产的 camera 数组中。每个相机都定义了一个 type 属性，该属性指定投影类型（透视或正交），以及一个 perspective 或 orthographic 属性，用于定义投影细节。

根据 zfar 属性的存在，透视相机可以使用有限或无限投影。

以下示例定义了两个透视相机，其中提供了 Y 视域，宽高比和裁剪信息。

```
{
    "cameras": [
        {
            "name": "Finite perspective camera",
            "type": "perspective",
            "perspective": {
                "aspectRatio": 1.5,
                "yfov": 0.660593,
                "zfar": 100,
                "znear": 0.01
            }
        },
        {
```

```
            "name": "Infinite perspective camera",
            "type": "perspective",
            "perspective": {
                "aspectRatio": 1.5,
                "yfov": 0.660593,
                "znear": 0.01
            }
        }
    ]
}
```

投影矩阵(Projection Matrices)

运行时应使用以下投影矩阵。

Infinite perspective projection（无限投影矩阵）

$$\begin{pmatrix} \frac{1}{a * \tan(0.5 * y)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(0.5 * y)} & 0 & 0 \\ 0 & 0 & -1 & -2n \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- a 等于 camera.perspective.aspectRatio(长宽比)
- y 等于 camera.perspective.yfov(Y 视域)
- n 等于 camera.perspective.znear(近裁剪面)

Finite perspective projection（有限投影矩阵）

$$\begin{pmatrix} \frac{1}{a * \tan(0.5 * y)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(0.5 * y)} & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- a 等于 camera.perspective.aspectRatio(长宽比)
- y 等于 camera.perspective.yfov(Y 视域)
- f 等于 camera.perspective.zfar(远裁剪面)
- n 等于 camera.perspective.znear(近裁剪面)

Orthographic projection（正交投影）

$$\begin{pmatrix} \frac{1}{r} & 0 & 0 & 0 \\ 0 & \frac{1}{t} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & \frac{f+n}{n-f} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- r 等于 camera.orthographic.xmag(水平放大率)
- t 等于 camera.orthographic.ymag(垂直放大率)
- f 等于 camera.orthographic.zfar(远裁剪面)
- n 等于 camera.orthographic.znear(近裁剪面)

动画（Animations）

glTF 通过节点变换的关键帧动画支持关节动画和蒙皮动画。关键帧数据存储在缓冲区中，并使用访问器在动画中引用。glTF2.0 还以类似方式支持实例化变形目标的动画。

说明：glTF2.0 仅支持动画节点变换和变形目标权重。规范的未来版本可能支持动画任意属性，例如材料颜色和纹理变换矩阵。

说明：glTF2.0 仅定义动画存储，因此规范并未定义任何特定的运行时行为，例如：播放顺序，自动启动，循环，时间轴映射等。

实现说明：glTF2.0 没有具体定义导入动画时的使用方式，但为了最佳实现，建议将每个动画动作作为动作自包含。例如，“行走”和“跑”动画可能各自包含针对模型各个骨骼的多个通道。客户端实现可以选择何时播放任何可用动画。

所有动画都存储在资产的 animations 数组中。动画定义了一组通道（channels 属性）和一组采样器，这些采样器指定包含关键帧数据和插值方法的访问器（samplers 属性）

以下示例展示了预期的动画使用方法。

```
{
    "animations": [
        {
            "name": "Animate all properties of one node with different samplers",
            "channels": [
                {
                    "sampler": 0,
                    "target": {
                        "node": 1,
```

```
                    "path": "rotation"
                }
            },
            {
                "sampler": 1,
                "target": {
                    "node": 1,
                    "path": "scale"
                }
            },
            {
                "sampler": 2,
                "target": {
                    "node": 1,
                    "path": "translation"
                }
            }
        ],
        "samplers": [
            {
                "input": 4,
                "interpolation": "LINEAR",
                "output": 5
            },
            {
                "input": 4,
                "interpolation": "LINEAR",
                "output": 6
            },
            {
                "input": 4,
                "interpolation": "LINEAR",
                "output": 7
            }
        ]
    },
    {
        "name": "Animate two nodes with different samplers",
        "channels": [
            {
                "sampler": 0,
                "target": {
                    "node": 0,
                    "path": "rotation"
```

```
                    }
                },
                {
                    "sampler": 1,
                    "target": {
                        "node": 1,
                        "path": "rotation"
                    }
                }
            ],
            "samplers": [
                {
                    "input": 0,
                    "interpolation": "LINEAR",
                    "output": 1
                },
                {
                    "input": 2,
                    "interpolation": "LINEAR",
                    "output": 3
                }
            ]
        },
        {
            "name": "Animate two nodes with the same sampler",
            "channels": [
                {
                    "sampler": 0,
                    "target": {
                        "node": 0,
                        "path": "rotation"
                    }
                },
                {
                    "sampler": 0,
                    "target": {
                        "node": 1,
                        "path": "rotation"
                    }
                }
            ],
            "samplers": [
                {
                    "input": 0,
```

```
                    "interpolation": "LINEAR",
                    "output": 1
                }
            ]
        },
        {
            "name": "Animate a node rotation channel and the weights of
a Morph Target it instantiates",
            "channels": [
                {
                    "sampler": 0,
                    "target": {
                        "node": 1,
                        "path": "rotation"
                    }
                },
                {
                    "sampler": 1,
                    "target": {
                        "node": 1,
                        "path": "weights"
                    }
                }
            ],
            "samplers": [
                {
                    "input": 4,
                    "interpolation": "LINEAR",
                    "output": 5
                },
                {
                    "input": 4,
                    "interpolation": "LINEAR",
                    "output": 6
                }
            ]
        }
    ]
}
```

Channels 将关键帧动画的输出值连接到层次结构中的特定节点。通道的 sampler 属性包含所在动画的 samplers 数组中的采样器的索引。target 属性是一个对象，该对用使用 node 属性标识进行动画的节点，以及使用 path 属性标识进行动画的节点属性。非动画属性必须在动画过程中保留其值。

当 node 未定义时，通道应该被忽略。有效的 path 值为“translation”，“rotation”，“scale”，“weights”。

每个动画采样器都定义输入/输出对：一组浮点标量值，以秒为单位表示线性时间；以及代表动画属性的一组矢量或标量。所有值都存储在缓冲区中，并可以通过访问器访问；有关输出访问器类型，请参考下表。关键帧之间的插值使用 interpolation 属性中指定的插值方法执行。支持的 interpolation 值包括 LINEAR，STEP，CUBICSPLINE。有关样条插值的更多信息，请参见附录 C。每个采样器的输入都相对于 t=0，定义为父级动画的开始。在提供的输入范围之前和之后，输出应“靠近”到输入范围的最近端。例如，如果动画的最早采样器输入为 t=10，则客户端实现应在 t=0 时开始播放该动画，并将输出钳位到第一个输出值。给定动画中的采样器不需要具有相同的输入。

channel.path	Accessor Type	Component Type(s)	Description
"translation"	"VEC3"	5126 (FLOAT)	XYZ translation vector
"rotation"	"VEC4"	5126 (FLOAT) 5120 (BYTE) normalized 5121 (UNSIGNED_BYTE) normalized 5122 (SHORT) normalized 5123 (UNSIGNED_SHORT) normalized	XYZW rotation quaternion
"scale"	"VEC3"	5126 (FLOAT)	XYZ scale vector
"weights"	"SCALAR"	5126 (FLOAT) 5120 (BYTE) normalized 5121 (UNSIGNED_BYTE) normalized 5122 (SHORT) normalized 5123 (UNSIGNED_SHORT) normalized	Weights of morph targets

实现必须使用以下等式从规范化的整数 c 中获取相应的浮点值 f，反之亦然：

accessor.componentType	int-to-float	float-to-int
5120 (BYTE)	f = max(c / 127.0, -1.0)	c = round(f * 127.0)
5121 (UNSIGNED_BYTE)	f = c / 255.0	c = round(f * 255.0)
5122 (SHORT)	f = max(c / 32767.0, -1.0)	c = round(f * 32767.0)
5123 (UNSIGNED_SHORT)	f = c / 65535.0	c = round(f * 65535.0)

动画采样器的 input 访问器必须定义 min 和 max 属性。

实现说明：带有非线性时间输入的动画（例如 Autodest 3ds Max 或 Maya 中的时间扭曲）不能直接用 glTF 动画表示。glTF 是一种运行时格式，非线性时间输入在运行时计算成本很高。导出工具应将非线性时间动画采样到线性输入和输出中，以进行准确表示。

变形目标动画帧由一系列标量定义，这些标量序列的长度等于动画变形目标中目标的数量。这些标量序列必须作为输出访问器中的单个流端对端防止，其最终大小将等于变形目标的数量乘以动画帧的数量。（？）

变形目标动画天生稀疏，请考虑使用稀疏访问器存储变形目标动画。与 CUBICSPLINE 插值一起使用时，切线（ak,bk）和值（vk）在关键帧内分组：

a1,a2,...an,v1,v2,...vn,b1,b2,...bn

有关样条插值的更多信息，请参见附录 C

glTF 动画可用于驱动关节动画或蒙皮动画。通过对蒙皮的关节层次中的关节进行动画处理，可以实现蒙皮动画。

指定扩展（Specifying Extensions）

glTF 定义了一种扩展机制，该机制允许使用新功能扩展基本格式。任何 glTF 对象都可以具有可选的 extensions 属性，如以下示例所示：

```
{
    "material": [
        {
```

```
            "extensions": {
                "KHR_materials_common": {
                    "technique": "LAMBERT"
                }
            }
        }
    ]
}
```

glTF 资产中使用的所有扩展名都必须在顶级 extensions 数组对象中列出，例如：

```
{
    "extensionsUsed": [
        "KHR_materials_common",
        "VENDOR_physics"
    ]
}
```

必须在顶层 extensionsRequired 数组中列出加载或渲染资产所需的所有 glTF 扩展,例如，

```
{
    "extensionsRequired": [
        "WEB3D_quantized_attributes"
    ]
}
```

ExtensionsRequired 是 extensionsUsed 的子集。extensionsRequired 中的所有值也必须存在于 extensionsUsed 中。

GIB 文件格式规范（GLB File Format Specification）

glTF 提供了两个数据传输选项，这些选项可以一起使用：

- glTF JSON 指向外部二进制数据（几何图形，关键帧，蒙皮）和图像。
- glTF JSON 嵌入 base64 编码的二进制数据，并使用 data URIs 内联图像。

对于这些资源，由于 base64 编码，glTF 需要单独的请求或额外的空间。Base64 编码需要额外的处理才能解码并会增加文件大小（对于编码后的资源，大约增加 33%）。尽管 gzip 减轻了文件大小，但解压和解码仍然会增加可观的加载时间。

为了解决这些问题，引入了一种容器格式，二进制 glTF。在二进制 glTF 中，

glTF 资产（JSON，.bin，images）可以存储在一个二进制大对象中。

该二进制大对象(可以是一个文件)具有以下结构：

- 占 12 个字节的表头
- 一个或多个 chunks，包含 JSON 内容和二进制数据

包含 JSON 的 chunks 可以引用外部资源，也可以引用其他 chunks 中存储的资源。

例如，想要按需下载纹理的应用程序可能将图像之外的所有内容嵌入二进制 glTF 中。仍支持嵌入式 base64 编码的资源，但是使用它们的效率很低。

文件扩展名（File Extension）

二进制 glTF 的文件扩展名是 .glb。

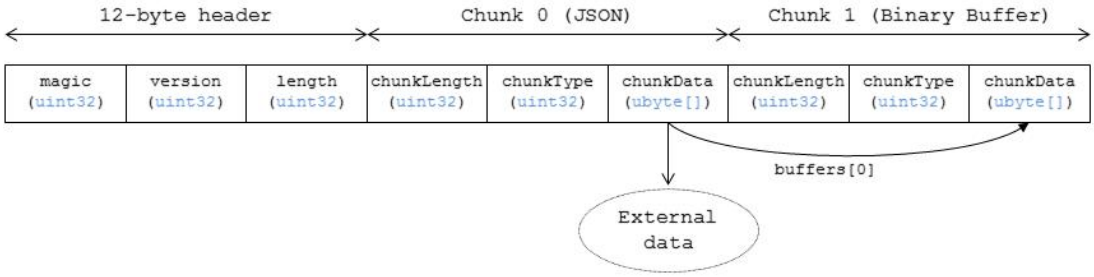
MIME 类型（MIME Type）

model/gltf-binary

二进制 glTF 结构（Binary glTF Layout）

二进制 glTF 是小端模式。图一显示了一个二进制 glTF 资产的示例。

图一：二进制 glTF 布局



以下各节将更详细地描述该结构。

头部（Header）

12 个字节的头部包含三个 4 字节条目：

```
uint32 magic

uint32 version

uint32 length
```

- magic 等于 0x46546C67，它是 glTF 的 ASCII 码，可用于将数据标识为

二进制 glTF。

- version 标识二进制 glTF 容器格式的版本。该规范定义了版本 2。
- length 表示二进制 glTF 总字节长度，包括头部和所有数据块。

实现说明：加载 GLB 格式的客户端实现也应检查 JSON 块中的资产版本属性，因为 GLB 头部中指定的版本仅指 GLB 容器版本。

数据块（Chunks）

每个数据块有以下结构：

```
uint32 chunkLength

uint32 chunkType

ubyte[] chunkData
```

- chunkLength 表示 chunkData 的字节长度。
- chunkType 表示数据块的类型。详情请见表 1。
- chunkData 是数据块的二进制有效载荷。

每个数据块的开始和结束必须与 4 字节边界对齐。有关填充方案，请参加块定义。数据块必须完全按照表 1 中给出的顺序出现。

表 1：块类型

	Chunk Type	ASCII	Description	Occurrences
1.	0x4E4F534A	JSON	Structured JSON content	1
2.	0x004E4942	BIN	Binary buffer	0 or 1

客户端实现必须忽略具有未知类型的块，以便 glTF 扩展能在前两个块之后引用具有新类型的其他块。

结构化 JSON 内容（Structured JSON Content）

该块保存结构化的 glTF 内容描述，就像在.gltf 文件中提供的那样。

实现说明：在 JavaScript 实现中，可以使用 TextDecoder API 从 ArrayBuffer 中提取 glTF 内容，然后可以使用 JSON.parse 解析 JSON。

该块必须是 Binary glTF 资产的第一个块。通过首先读取此块，实现可以逐步从后续块中检索资源。这样，还可以仅从 Binary glTF 资产中读取资源的选定子集（例如，网格的最简化的 LOD）。

必须用尾随的空格字符（0x20）填充此块，以满足对齐要求。

二进制缓冲区（Binary buffer）

该块包含用于几何图形，动画关键帧，蒙皮和图像的二进制有效载荷。有关从 JSON 引用此块的详细信息，请参见 glTF 规范。

该块必须是 Binary glTF 资产的第二个块。

必须用尾随零（0x00）填充此块，以满足对齐要求。