链滴

认识 deck.gl

作者：likaomer

原文链接：https://ld246.com/article/1605362989924

来源网站：链滴

认识deck.gl

来自官方的解释:

deck.gl的设计目的是简化大数据集的可视化。它使用户能够通过现有层的组合以有限的努力快速获令人印象深刻的可视化结果，同时为将基于WebGL的高级可视化打包为可重用的JavaScript层提供了整的体系结构。

简要概述:

deck.gl的基本思想是渲染层的叠加，通常是渲染在地图上,但不总是;

deck.gl面对的机遇与挑战:

1. 处理大数据集和性能更新优化;
2. 交互式处理 例如拾取事件(picking);
3. 地图投影和与基础地图的集成;
4. 经过验证的、经过良好测试的层的目录;
5. 易于创建新层或自定义现有层.

生态系统:

deck.gl是vis.gl框架的主要框架之一;与许多框架进行并行开发,如下;

● luma.gl - 一个通用的WebGL库，旨在与原始的WebGL API和其他WebGL库(尽可能)互操作。特是，luma.gl不声明WebGL上下文的所有权，并且可以使用任何提供的上下文，包括应用程序或其他ebGL库创建的上下文;

● react-map-gl - 一个React包装周围的Mapbox GL与deck.gl无缝工作(通俗点说就是mapbox-gl的ract版本);

● nebula.gl - 一个有着高性能的deck.gl的编辑框架;

如何使用:

1. 安装: npm install deck.gl --save
2. 也可以直接下载官方栗子:

git clone git@github.com:uber/deck.gl.git

cd deck.gl/examples/...(你想看的栗子目录下)

npm install

npm start ---- (运行)

这样栗子就运行起来啦!!!

纯js使用deck.gl:

● deck.gl的核心库和层对React或Mapbox gl没有依赖关系，可以被任何JavaScript应用程序使用。

●

```
import {Deck} from '@deck.gl/core'; // 引入deck的核心模块
```

原文链接：认识 deck.gl

```
import {GeoJsonLayer} from '@deck.gl/layers'; // 引入需要显示的层,有很多层,后续讲到

// source: Natural Earth http://www.naturalearthdata.com/ via geojson.xyz
const GEOJSON =
  'https://d2ad6b4ur7yvpq.cloudfront.net/naturalearth-3.3.0/ne_110m_admin_1_states_provinces_shp.geojson'; // 要展示的数据

const INITIAL_VIEW_STATE = {
  latitude: 40, // 经度
  longitude: -100, // 纬度
  zoom: 3, // 缩放级别
  bearing: 30, // 轴承指数
  pitch: 30 // 倾斜角度 0 - 90
};

export const deck = new Deck({
  width: '100%', // 设置deck范围
  height: '100%', // 高
  initialViewState: INITIAL_VIEW_STATE, // 定义初始化页面的deck在地图上的位置,视角
  controller: true, // 将控制器打开
  layers: [ // 需要显示的层
    new GeoJsonLayer({ // 将显示的层new出实例
      data: GEOJSON, // 展示的数据
      stroked: true, // 线
      filled: true, // 填充
      lineWidthMinPixels: 2, // 线的最小宽度
      opacity: 0.4, // 线的透明度
      getLineColor: () => [255, 100, 100], // 设置线的颜色
      getFillColor: () => [200, 160, 0, 180] // 设置填充的颜色
    })
  ]
});

// For automated test cases
/* global document */
document.body.style.margin = '0px';
```

@deck.gl/core是deck的一个子模块，不包含任何响应依赖项的gl.

- 上面的代码实例就是使用一个纯js的deck.gl模块库的完整示例. es6的写法;
- 在script中如何引入?

```
<script src='https://unpkg.com/deck.gl@latest/deckgl.min.js'></script>

<script src='https://api.tiles.mapbox.com/mapbox-gl-js/v0.44.1/mapbox-gl.js'></script>

<link href='https://api.tiles.mapbox.com/mapbox-gl-js/v0.44.1/mapbox-gl.css'></link>
```

不仅要引入deck.gl,而且还要引入mapbox-gl的js和css文件

结合React使用:

这也是官方推荐使用的方式;

虽然不是直接基于React, deck.gl从头设计到使用基于React的应用程序。deck.gl层自然适合React的件渲染流和基于flux/redux的应用。当您重新运行您的正常JSX或React组件时，gl层将被高效地重新载。

● 结合React使用Deck.gl，只需导入DeckGL React组件，并将其作为另一个组件的子组件呈现，然传入deck列表。deck.gl层作为属性。

●

```
/// app.js
import React from 'react'; // 结合React使用,先引入React
import DeckGL, {LineLayer} from 'deck.gl'; // 引入DeckGL 和需要使用的层

// Viewport settings
// 初始化的页面角度位置
const viewState = {
  longitude: -122.41669, // 经度
  latitude: 37.7853, // 纬度
  zoom: 13, // 缩放
  pitch: 0, // 倾斜角度
  bearing: 0 // 轴承
};

// Data to be used by the LineLayer
// 创建数据 源位置 --- 目标位置
const data = [{sourcePosition: [-122.41669, 37.7853], targetPosition: [-122.41669, 37.781]}];

// DeckGL react component
class App extends React.Component {
  render() {
    const layers = [
      new LineLayer({id: 'line-layer', data}) // 初始化层
    ];

    return (
      <DeckGL viewState={viewState} layers={layers} /> // 设置组件
    );
  }
}
```

deck.gl的重要伙伴 ---- react-map-gl。它是Mapbox的一个React包装器，可以共享相同的web mer ator viewport设置。

```
/// app.js
import React from 'react';
import DeckGL, {LineLayer} from 'deck.gl';
import {StaticMap} from 'react-map-gl'; // 引入react-map-gl

// Set your mapbox access token here
const MAPBOX_ACCESS_TOKEN = 'MAPBOX_ACCESS_TOKEN';

// Initial viewport settings
const initialViewState = {
  longitude: -122.41669,
```

```
  latitude: 37.7853,
  zoom: 13,
  pitch: 0,
  bearing: 0
};

// Data to be used by the LineLayer
const data = [{sourcePosition: [-122.41669, 37.7853], targetPosition: [-122.41669, 37.781]}];

class App extends React.Component {
  render() {
    const layers = [
      new LineLayer({id: 'line-layer', data})
    ];

    return (
      <DeckGL
        initialViewState={initialViewState}
        controller={true}
        layers={layers}
      >
        <StaticMap mapboxApiAccessToken={MAPBOX_ACCESS_TOKEN} /> ---- 在这加入react-
ap-gl的map组件
      </DeckGL>
    );
  }
}
```

- 关于react-map-gl:

如示例中所示，DeckGL React组件作为React组件的子组件(如使用类似于deck的参数显示地图的Rea t -map-gl)尤其有效。gl视口(即纬度，经度，缩放等)。在这个配置中，你的deck.gl层将在底层地图呈现一个完全同步的地理空间覆盖。

通过层可视化数据:

1. 渲染单个层

- deck.gl的设计允许您获取任何可以关联位置的数据，并使用deck轻松地在地图上呈现这些数据关于deck.gl层,只需实例化该层的类，并传入一组属性，其中包括数据本身，以及该层用于构建可视的一些访问器和属性;
- 属性是允许控制层如何呈现数据的值;
- 访问器是用来描述层应该如何提取各种值的函数。
-

```
<DeckGL layers={[
  new ArcLayer({data: ...})
]} />
```

2. 渲染多个层

- deck.gl允许使用相同或不同的数据集呈现多个层。只需提供层实例和deck的数组。deck.gl将顺序渲染它们(并处理悬停单击时的交互性等)。
-

```
<DeckGL layers={[
  new PathLayer({data: ...}),
  new LineLayer({data: ...}),
  new ArcLayer({data: ...}),
]} />
```

Deck.gl中主要的类:

1. Deck

- Deck是一个使用Deck的类。deck.gl层实例和viewport参数，将这些层呈现为透明覆盖，并处事件;
- 在使用React-map-gl中不需要此类的参加,而在纯js开发中,此类便是不可缺少的;
- 怎么用?
-

```
// Basic standalone use
import {Deck, ScatterplotLayer} from 'deck.gl';

const App = (viewState, data) => (
// 创建一个新的Deck类 --- 生成一个画布
const deck = new Deck({
// 加入层
layers: [new ScatterplotLayer({data})]
});
deck.setProps({viewState});
);
```

- Deck类的属性:
  1. width: 画布的宽(Number)
  2. height: 画布的高(Number)
  3. layers: 要渲染的层的集合 --- 将要渲染的层放到一个数组里面,进行逐个渲染(Array)
  4. layersFilter: 接收一个函数,函数的参数为layer, viewport, isPicking,用来过滤层,调整视口,制是否发生拾取事件,所有层在渲染之前都会走这个函数,所以层的显隐可以在这里实现(Function: ({layers, viewport, isPicking}) => {....})
  5. getCursor: 接收一个函数,来检索游标类型的自定义回调(Function: ({isDragging}) => isDragging ? 'grabbing' : 'grab')
  6. views: 单个视图或视图实例的数组(可选地与Viewport实例混合，尽管后者是不推荐的)。如没有提供，将创建一个MapView。如果提供了空数组，则不会显示任何视图。(Array)
  7. viewState: 地理空间视图状态(Object)
     - latitude: 经度(Number)
     - longitude: 纬度(Number)

- zoom: 缩放程度(Number)
    - bearing: 轴承程度(Number)
    - pitch: 倾斜角度(Number: 0 - 90)

可以通过改变viewState的参数值来改变画布实例的角度位置

8. initialViewState: 如果提供了initialViewState, Deck组件将使用内部状态跟踪来自任何附加制器的视图状态更改，initialViewState作为其初始视图状态。(Object: 属性和viewState一样)

9. controller: 视窗互动选项，例如:平移，旋转和缩放与鼠标，触摸和键盘。如果使用默认视图即单个MapView)，这是定义与视图道具交互的简写。

```
new Deck({
  ...
  views: [new MapView({controller: true})]
})
// 上面写法和下面是一样的
new Deck({
  ...
  // views: undefined
  controller: true
})
```

其值有以下几种类型:

- null or false: 默认不启动控制器
- true: 默认启动默认控制器
- Controller: 使用默认选项启动所提供的控制器。必须是MapController的子类。
- 默认为null

- Deck类的配置属性:

1. id(String): Canvas ID允许在CSS中定制样式;

2. style(Object): deckgl-canvas的Css样式;

3. pickingRadius(Number): 在选择时指针周围要包括的额外像素。当呈现的对象难以定位时这是很有帮助的，例如不规则形状的图标、小的移动圆圈或触摸交互。默认为0。

4. useDevicePixels(Boolean): 如果为真，设备的全分辨率将用于渲染，这个值可以在每帧中变，比如在屏幕之间移动窗口或改变浏览器的缩放级别。默认为真,但是一般都将其设置为false,因为个会非常消耗性能,除非机器的性能足够高.

5. gl(Object): gl的上下文,如果不提供将会自动创建;

- Deck类的回调函数:

1. onWebGLInitialized(Function: (gl) => {....})一旦webGL上下文被初始化,就会被调用;

2. onViewStateChange(Function: (viewState) -> {....}): 当用户改变画布角度时调用.例如使鼠标拖曳, 键盘控制等.

3. onLayerHover(Function:): 当指针下的对象发生变化时调用。在鼠标悬浮在物体之上发生回调;

- 参数: info, pickedInfos, event
- info: 在坐标中最顶层选择的info对象，在没有选择对象时为null;

- pickedInfos: 所有受影响的可选层的信息对象数组。
    - event: 原始MouseEvent对象

4. onLayerClick(Function): 当指针点击时发生的回调.参数同Hover回调

5. onDragStart, onDrag, onDragEnd(Function): 拖曳开始, 拖曳中, 拖曳结束

- 参数: info, event 同Hover事件的参数

6. onLoad(Function): 在创建gl上下文和Deck组件(ViewManager、LayerManager等)之后调一次。可用于触发视口转换;

- Deck类的方法:

1. finalize: 立即释放与此对象关联的资源(而不是等待垃圾收集)。

deck.finalize();

2. setProps: 更新属性.

deck.setProps({.....})

3. pickObject: 在给定的屏幕坐标上获取最近的可选可见对象.

deck.pickObject({x, y, radius, layerIds})

x: x位置(像素)

y: y位置(像素)

radius: 以像素为单位的公差半径

layerIds: 要查询的层id的列表。如果没有指定，则查询所有可选的和可见的层

2. Layer: Layer类是所有deck的基类。gl层，它提供了许多在所有层中可用的基本属性。

- 静态成员

1. layerName: 这个静态属性应该包含层的名称，通常是层的类的名称(不能在缩小的代码中可地自动推导)。它用作默认的层id以及调试和分析.

2. defaultProps: 所有deck.gl层定义一个defaultProps静态成员，列出它们的道具和默认值。层实例构建过程中，使用defaultProps可以提高代码的可读性和性能。

- 构造函数

new Layer(...props);

- 基础属性

1. id(String): 静态属性 --- 在给定的时间内，id必须在所有层之间是唯一的。id的默认值是层“name”。如果一个特定层类型的多个实例同时存在，它们必须拥有deck的不同id字符串。gl来正区分它们。

2. data(String|Iterable|Promise): deck.gl层通常期望数据支柱的值是一个JavaScript数组。可以使用任何实现iterable协议的对象.数组,伪数组都可以.

- 如果提供字符串，deck。gl将尝试将其作为URL加载，将其解析为JSON，然后使用预期结果JavaScript数组作为其数据支柱
- 如果是Promise函数，deck.gl将使用resolve解析值作为其数据支柱。

3. visible(Boolean): 层是否可见。在大多数情况下，建议使用可见道具来控制图层的可见性，

不是使用条件渲染

4. opacity(Number): 图层的不透明度;

- 交互属性

1. pickable(Boolean, default: false): 层是否响应鼠标指针选择事件。
2. onHover(Function): 当鼠标进入/离开此deck.gl层的对象时，将调用此回调,具有以下参数:
   - info: 此时鼠标下的信息
   - event: 事件对象
3. onClick(Function): 当鼠标点击此deck.gl层的对象时,将调用此回调,具有以下参数: 同onHovr
4. onDragStart, onDrag, onDragEnd同Deck回调事件;
5. highlightColor(Array, [0, 0, 128, 128])
   - RGBA颜色用于渲染突出显示的对象。当指定3 component (RGB)数组时，alpha使用默值255。
6. highlightedObjectIndex(Integer, -1)
   - 当提供一个有效值时，对应的对象(实例渲染中的一个实例或具有相同拾取颜色的一组原语将使用highlightColor高亮显示。
7. autoHighLight(Boolean, false)
   - 当为true时，鼠标指针指向的当前对象(悬停在其上时)用highlightColor高亮显示。要求picable为真。

- 坐标系统的属性 ---- 通常只在应用程序希望使用非Web Mercator投影经纬度的坐标时使用

1. coordinateSystem(Number): 指定如何对层位置和偏移量进行地理解释。
2. coordinateOrigin([Number, Number]): 将coordinateSystem设置为coordinate_system .meter_offset时需要。
3. wrapLongitude(Boolean, false): 自动将纵向缠绕在第180个反物质上，以获得当前视区的佳可视性。
4. modelMatrix(Number[16]):一个可选的4x4矩阵，它被乘进着色器项目GLSL函数和Viewpor的项目和非项目JavaScript函数使用的仿射投影矩阵

- 数据属性

1. dataComparator(Function): 这个支持使用自定义比较函数对数据支持进行比较。使用旧数和新数据对象调用compare函数，如果它们进行相等的比较，则期望返回true。
2. numInstances(Number): deck.gl通过计算数据中对象的数量，自动从数据支柱中获取绘图例的数量。但是，开发人员可能希望使用此道具手动覆盖它。
3. updateTriggers(Object): 调用getColor和getPosition等访问器来检索第一次添加层时的颜和位置。从那以后，为了最大化性能，deck。gl不会重新计算颜色或位置，除非通过浅比较更改数据柱。

3. 其他衍生Layer层的实例

1. ArcLayer: 弧线层呈现向上的弧线，连接源点和目标点对，指定为纬度/经度坐标。

- 怎么使用?

```
import DeckGL, {ArcLayer} from 'deck.gl';
// 从deck.gl里面引出弧线层类
const App = ({data, viewport}) => {
// 拿到数据,匹配格式
  /**
   * Data format:
   * [
   *   {
   *     inbound: 72633,
   *     outbound: 74735,
   *     from: {
   *       name: '19th St. Oakland (19TH)',
   *       coordinates: [-122.269029, 37.80787]
   *     },
   *     to: {
   *       name: '12th St. Oakland City Center (12TH)',
   *       coordinates: [-122.271604, 37.803664]
   *   },
   *   ...
   * ]
   */
    // 创建弧线层实例
  const layer = new ArcLayer({
    id: 'arc-layer', // 定义名称 -- 方便之后更容易找到此层并操作
    data, // 加载数据
    pickable: true, // 允许发生拾取事件
    getStrokeWidth: 12, // 设置弧线的宽度
    getSourcePosition: d => d.from.coordinates, // 设置来源点的位置
    getTargetPosition: d => d.to.coordinates, // 设置目标点的位置
    getSourceColor: d => [Math.sqrt(d.inbound), 140, 0], // 设置弧线来源点一边的弧线颜色
    getTargetColor: d => [Math.sqrt(d.outbound), 140, 0], // 设置弧线目标点的弧线颜色
    onHover: ({object, x, y}) => {
      const tooltip = `${object.from.name} to ${object.to.name}`; // 在鼠标悬浮的时候产生的回调
数
      /* Update tooltip
         http://deck.gl/#/documentation/developer-guide/adding-interactivity?                    secti
n=example-display-a-tooltip-for-hovered-object
      */
    }
  });

  return (<DeckGL {...viewport} layers={[layer]} />); // 最后在组件中渲染该层实例
};
```

- 属性: 从所有的基础层类(Layer)继承属性.
- 渲染时的配置项:

1. fp64(Boolean, false): 是否应该在高精度64位模式下渲染该层。注意自deck.gl v6.1，默认32位投影使用一种混合模式，这种模式与64位精度匹配，性能显著提高.

2. widthScale(Number, 1): 每个弧的宽度的比例因子。如果将属性设置为Math.pow(2,viewpot.zoom - 12)宽度保持不变，对应当前的zoom level和zoom level 12时1像素的宽度。还可以使用此性限制弧的最小大小宽度.

- 数据访存器:

    1. getSourcePosition(Function): 方法 --- 来检索每个对象的源位置。
    2. getTargetPosition(Function): 方法 --- 来检索每个对象的目的位置.
    3. getSourceColor(Function|Array, [0, 0, 0, 255]): 方法 --- 检索位置颜色.
        - 如果是Array,直接返回数组里面的rgba颜色值
        - 如果是Function函数值, 该函数返回一个rgba的数组值.
    4. getTargetColor(Function|Array, [0, 0, 0, 255]): 方法 --- 检索位置颜色.
        - 同上
    5. getStrokeWidth(Function|Number, 1): 每个对象的笔画宽度，以像素为单位。
        - 如果提供了数字，则将其用作所有对象的笔划宽度.
        - 如果提供了函数，则对每个对象调用该函数以检索其笔划宽度。

2.GeoJsonLayer: GeoJson层接收GeoJson格式的数据，并将其呈现为交互式的多边形、线和点。

- 怎么使用?

```
import DeckGL, {GeoJsonLayer} from 'deck.gl';
// 用法和创建和之前的栗子差不多,只是渲染时的一些属性不同
const App = ({data, viewport}) => {
  const {data, viewport} = this.props;

  /**
   * Data format:
   * Valid GeoJSON object
   */
  const layer = new GeoJsonLayer({
    id: 'geojson-layer',
    data,
    pickable: true,
    stroked: false,
    filled: true,
    extruded: true,
    lineWidthScale: 20,
    lineWidthMinPixels: 2,
    getFillColor: [160, 160, 180, 200],
    getLineColor: d => colorToRGBArray(d.properties.color),
    getRadius: 100,
    getLineWidth: 1,
    getElevation: 30,
    onHover: ({object, x, y}) => {
      const tooltip = object.properties.name || object.properties.station;
      /* Update tooltip
         http://deck.gl/#/documentation/developer-guide/adding-interactivity?section=example
display-a-tooltip-for-hovered-object
      */
    }
  });
```

```
  return (<DeckGL {...viewport} layers={[layer]} />);
};
```

● 渲染属性: 从所有基层属性继承而来的数据支柱被稍微灵活地解释了一下，以适应纯粹的GeoJso“有效负载”。

1. filled(Boolean, true): 是否绘制填充多边形(实体填充)。注意，对于每个多边形，只有外部边形和任何孔之间的区域将被填充。这个道具只有在多边形没有被挤压的情况下才有效。

2. stroked(Boolean, false): 是否绘制多边形的轮廓(固体填充)。注意，对于复杂的多边形，将制外部多边形以及任何孔的轮廓。

3. extruded(Boolean, false): 如果设置为true，沿z轴挤压多边形和多多边形特征。绘制的特的高度是通过getElevation访问器获得的。

4. wireframe(boolean, false): 是否生成六边形的线框。轮廓将有“水平”线关闭的顶部和底多边形和垂直线(“支柱”)的每个顶点的多边形.

● 只有当extruded设置为true是才有效

5. lineWidthScale(Boolean, 1): 行宽乘法器，用于乘到所有行，包括LineString和MultiLineStrng特性，如果描边属性为真，还包括多边形和多多边形特性的轮廓。

6. lineWidthMinPixels(Number, 0): 最小行宽，以像素为单位。

7. lineWidthMaxPixels(Number, Number.MAX_SAFE_INTEGER): 最大行宽, 像素为单位

8. lineJointRounded(Boolean, false): 两条geo线连接重叠部分的样式.

9. lineMiterLimit(Number, 4): 一个l连接点的最大宽度与行宽度之比。只有当linejointround假时才有效.

10. elevationScale(Number, 1): 海拔乘数。最终高程按海拔比例尺* getElevation(d)计算。elvationScale是一个处理的属性，可以在不更新数据的情况下缩放所有多边形的高程。

11. pointRadiusScale(Numbel, 1): 所有点的全局半径乘法器。

● 数据访问器

1. getLineColor(Function|Array, [0, 0,0,255]): 根据GeoJson特性的类型，行字符串的rgba颜和/或多边形的轮廓。格式为r, g, b， [a]。每个组件都在0-255范围内。

2. gitFillColor(Function|Array, [0,0,0,255]): GeoJson的多边形和点特征的纯色。格式为r, g, ， [a]。每个组件都在0-255范围内。

3. getRadius(Function|Number): 点的半径和多点特征，以米为单位。

4. getLineWidth(Function|Number): 根据GeoJson特性的类型，行字符串的宽度和/或多边的轮廓。单位是米。

5. getElevation(Function|Number): 多边形特征的高程(挤压时为真)。

3. HexagonLayer

● 六边形层根据点数组呈现一个六边形热图。它取六边形仓的半径，投影到六边形仓中。六边的颜色和高度由它包含的点的数量来决定。

● 怎么使用?

```
import DeckGL, {HexagonLayer} from 'deck.gl';

const App = ({data, viewport}) => {
```

```
  /**
   * Data format:
   * [
   *   {COORDINATES: [-122.42177834, 37.78346622]},
   *   ...
   * ]
   */
  const layer = new HexagonLayer({
    id: 'hexagon-layer',
    data,
    pickable: true,
    extruded: true,
    radius: 200,
    elevationScale: 4,
    getPosition: d => d.COORDINATES,
    onHover: ({object, x, y}) => {
      const tooltip = `${object.centroid.join(', ')}\nCount: ${object.points.length}`;
      /* Update tooltip
         http://deck.gl/#/documentation/developer-guide/adding-interactivity?section=example
display-a-tooltip-for-hovered-object
      */
    }
  });

  return (<DeckGL {...viewport} layers={[layer]} />);
};
```

- 渲染属性

1. radius(Number, 1000): 六角形料斗半径(米)。六边形是尖顶的(而不是平顶的)。

2. hexagonAggregator(Function, d3-hexbin): 六边形聚合器是将数据聚合到六边形容器的函数。六边形聚合器将层的道具和当前视图作为参数。输出应该是{六边形:[]，六边形顶点:[]}。六形是{centroid: []， points:[]}的数组，其中centroid是六边形的中心，points是六边形包含的点的数。六边形顶点(可选)是定义基本六边形几何形状的点数组.

3. colorRange(Array): 颜色值数组,用来创建六边形因密度不同而形成的颜色区分;

4. coverage(Number): 六角半径乘法器，夹在0 - 1之间。六边形的最终半径是用覆盖率*径来计算的。注意:覆盖率不影响点的绑定方式。容器的半径仅由半径属性决定。

5. elevationDomain(Array, [0, Max(count)]): 高程刻度输入域。高程比例尺是一种线性例尺，它将计数的数量映射到高程。默认情况下，它被设置为在每个六边形的点计数的0到最大值之。当您希望使用相同的高程比例呈现不同的数据输入以进行比较时，此属性非常方便。

6. elevationRange(Array, [0, 1000]): 高程刻度输出范围

7. elevationScale(Number, 1): 六角海拔乘数。实际高程按elevationScale* getElevation()计算。elevationScale是一个方便的属性，可以在不更新数据的情况下缩放所有六边形.

8. extruded(Boolean, false): 是否启用单元格提升。单元格高程按每个单元格中的点数计。如果设置为false，则所有单元格都是扁平的。

9. upperPercentile(Number, 100): 过滤箱和重新计算颜色的上位百分比。颜色值大于上分位的六边形将被隐藏(控制数据量密度的值,值越大,密度要更大才能更好的区分)

10. lowerPercentile(Number, 0): 过滤箱和重新计算颜色的较低百分位。颜色值小于最小分位数的六边形将被隐藏。(密度控制阀, 越低,密度越低的颜色有区分)

11. elevationUpperPercentile(Number, 100): 过滤箱和重新计算海拔高度的海拔上百分位。高程值于上百分位的六边形将被隐藏

12. elevationLowerPercentile(Number, 0): 过滤箱和重新计算海拔高度由海拔低百分位。高小于标高的六边形将被隐藏。

13. lightSettings(Object): 这个对象包含挤压多边形的灯光设置。请注意，这个道具可能在deck.gl的未来版本中更改.

- 数据访问器

1. getPosition(Function): 方法来检索每个点的位置。

2. getColorValue(Function): getColorValue是获取bin color所基于的值的访问器函数。将每个bin中的点数组作为参数，返回一个数字。例如，可以通过每个点的特定属性的avg/mean/ma，将getColorValue传递给颜色容器。默认情况下，getColorValue返回点数组的长度。

```
class MyHexagonLayer {
   getColorValue (points) {
      return points.length;
   }

   renderLayers() {
     return new HexagonLayer({
       id: 'hexagon-layer',
       getColorValue: this.getColorValue // instead of getColorValue: (points) => { return points
length; }
       data,
       radius: 500
     });
   }
 }
```

3. getElevationValue(Function): 类似于getColorValue, getElevationValue是获取bin elevation所基于的值的访问器函数。它将每个bin中的点数组作为参数，返回一个数字。默认情况下，getElevationValue返回点数组的长度

4. onSetColorDomain(Function): 计算bin颜色域时将调用此回调。

5. onSetElevationDomain(Function): 当bin海拔域计算完成时，将调用此回调。

注: 还有其他layer层的类,可以去官网上阅读,其实每个层的类的用法都一样,只是,每个层有其不一样的性,结合不一样的属性创建不一样的动画,或者物体.

4. shader modules --- 着色器模块 ---请自行查询官网