Report of the Treemap

文件结构和使用方法

使用方法:

请在联网条件下，用支持***html5***的浏览器打开 ***index.html*** 文件，点击**选择文件**按钮后，选择你想要转换成***Treemap***的***json***格式文件。 然后点击**呈现**按钮，即可获得***Treemap***。

（注意：代码中***colors***数组中颜色数目可能不够，倘若所给数据数目过多，请手动添加颜色）

功能:

当你把鼠标放置于某个矩形上时，会出现一个悬浮框显示其***name***和***size***属性的值。

当你点击某一颜色区域任意一个矩形区域时，该颜色区域会被放大显示。（注意只有点击矩形区域时才会有此效果，点击文本或矩形边框均不能实现此效果）

数据结构与算法

1、数据结构

以一个 ***data*** 对象存储获取的数据，后期数据的处理及存储同样放在***data***对象中。

其结构如下：

data{

children : [data1' , data2' , ...] ,

name: string ,

size: number,

x0,y0,x1,y1

}

2、函数列表

getSize(obj) //递归获取obj 对象中每一层的***size***值，并排序

squarifyEach(obj, x0, y0, x1, y1) //将***obj***对象中每一层元素排布在$[x0, y0; x1, y1]$矩形中，并记录坐标

squarify(obj, x0, y0, x1, y1) //将***obj***对象中***children***数组的所有成员排布在$[x0, y0; x1, y1]$矩形中

Dice(obj, x0, y0, x1, y1) //将***obj***中***children***数组的所有成员横向排布在$[x0, y0; x1, y1]$矩形中

Slice(obj, x0, y0, x1, y1) //将***obj***中***children***数组的所有成员纵向排布在$[x0, y0; x1, y1]$矩形中

work() //点击呈现按钮后，执行此函数，计算坐标并且渲染

render(obj, color, id, func) //将***obj***对象中元素，动态渲染成颜色为***color***，***id***属性值为***id***，点击后执行函数

名为***func***的矩形。

turn1(obj) //将***dom***元素***obj***所处的颜色区域放大显示

turn2(obj) //重新把***data***进行渲染，即返回***treemap***界面

3、函数解释

运行过程

用一个函数监听***input***节点，当用户选择文件时，执行***getSize***函数，以进行数据预处理

当用户点击呈现按钮时，触发***work()***函数，该函数会执行***squarifyEach()***函数，以使用***squrify***算法将数据排布成矩形，并记录坐标，方便后续渲染。然后为***svg***画布设置大小属性。并且循环地调用***render()***函数，将***data***最外层的***chilren***数组的成员渲染成同一颜色的矩形。

初始化数据

由于在初始的***data***中，只有最底层元素拥有***size***属性。故利用***getSize***函数，递归地预处理***data***，得到每一层对象地***size***属性。并且为每一层的***children***数组按***size***从大到小排序。在这个函数中，我们获取参数***obj***，并且对***obj***的类别进行讨论。当***obj***是数组的时候，计算返回其成员的***size***和。当***obj***不是数组时，我们得到其***children***的***size***之后，为其添加***size***属性。当***obj***没有***children***数组时，递归到达边界。

```
getSize (obj )
{
    if obj is Array
        for numbers in obj
            size + = size of number
        end
        sort obj
        return size
    else
        if obj has no chilren
            return
        else
            ownSize = getSize (chilrens)
            return ownSize
            addAttribute size to obj
        end if
    end if
}
```

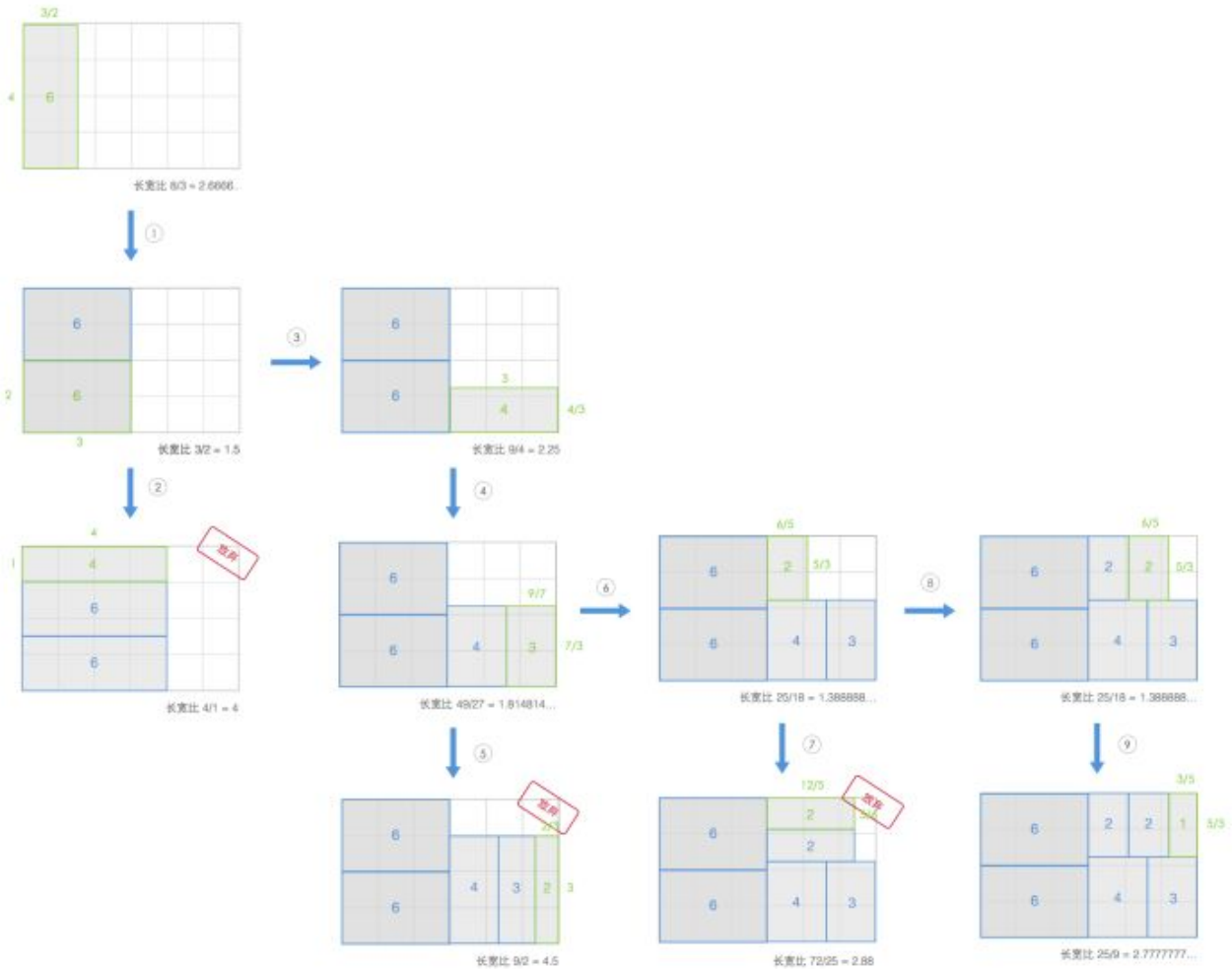
squarified算法

此算法旨在将层级数据依据相对***size***大小关系，动态创造矩形，排布在一个大的矩形之中。

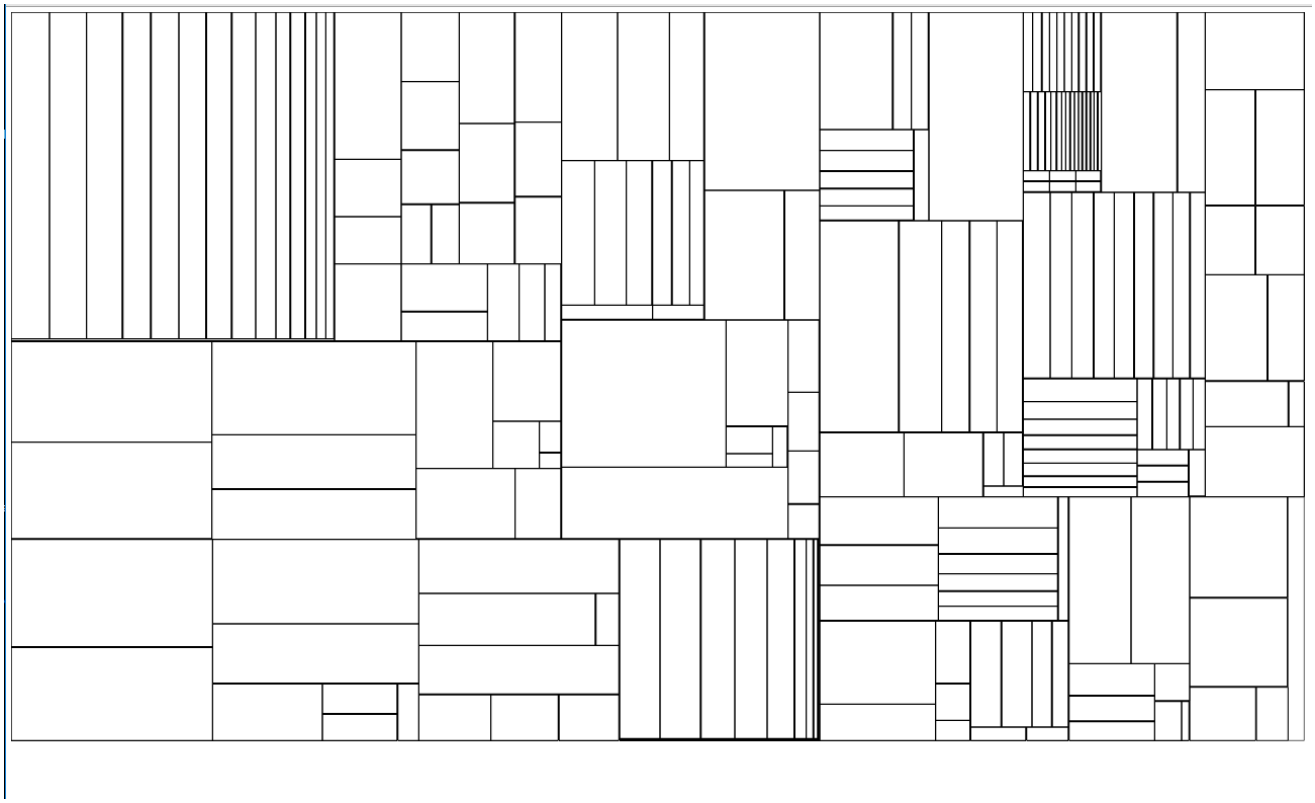
此类问题最初的算法是***slice-and-dice***算法。但是此类算法得到的矩形图随数据的变化过大，甚至生成长宽比过大的矩形，与用户的交互较差。

Bruls提出了新的***squrified*** 算法。 该算法运用贪心的思想。即获得尽可能接近1的长宽比以使得得到的矩形更像正方形。其具体实现方法为，先对矩形面积进行排序。然后选取矩形排列到当前空白区域上。显然，沿着空白区域短边进行排布获得的矩形长宽比更小。于是，我们总是把矩形紧靠空白区域短边进行排布。该算法的核心在于是否开始新的一列（一行）进行排布。判断依据如下：选择下一个待排布的矩形，将其加入到当前列（行）。我们把目前这一列（行）的总面积当作一个沿着短边进行排布，如果得到的新矩形长宽比比未加入时更小，说明加入该矩形使得矩形更好看，我们将它加入当前列（行）。反之，说明加入新矩形使得矩形更难看，于是我们把它放到下一列（行）。然后我们把每一列的元素在其区域内进行排布即可。

过程可以用下图表示（图片来源于知乎）



该算法在实现过程中得到了不好的结果。如下图（图片来源于自己之前的算法得到的图片）



于是我又参考了***d3***框架的***squarify***算法。这个算法旨在使得得到的矩形具有尽可能接近给定比例的长宽比。我选择了黄金分割比。

其算法大体框架与***Bruls***的***squarify***算法相似。区别在于判断是否将新矩阵加入当前列时的判断依据。

其实现公式如下：

$$\alpha = (dy * ratio)/(dx * S)$$

$$\beta = (S0 * S0 * dy * ratio)/(S * dx)$$

$$Ratio1 = (SM * S * dx)/(S0 * S0 * dy * ratio)$$

$$Ratio2 = (S0 * S0 * dy * ratio)/(S * Sm * dx)$$

$$minRatio = max(Ratio1, Ratio2)$$

其中 dx 表示短， dy表示长边，$ratio$ 是黄金分割比的倒数。S表示当前空白区域面积，$S0$表示当前列（行）的矩形的总面积，SM表示当前列中最大的矩形面积，Sm 表示当前最小的矩形面积。

以$Ratio1$为例分析其原理：

上面的$Ratio1$可以写成如下式子：

$$Ratio1 = ((SM/S0) * dx)/((S0/S) * dy)/ratio$$

其中，$(SM/S0) * dx$ 是最大的矩形的边，$(S0/S) * dy$ 表示当前列的矩形的边，同时也是最大矩形的另一条边。也就是说这个比值就是最大矩形的长宽之比。**故 $Ratio1$ 表示的是矩形长宽比与黄金分割比的接近程度。且值越小越接近。** $Ratio2$ 同理 。

我们取两者的最大值，即取两者中比例较差的一个。接着计算加入下一个矩阵后的***ratio***，如果矩形变得更优了，那么我们就把他加入到当前列中。否则，开始排布下一列。

以上算法在***squarify***函数中实现，用以下伪代码表述

```
for each child in obj
     minSize = maxSize = sumSize;
     alpha = max(dy / dx, dx / dy) / (size * ratio);
     beta = sumSize * sumSize * alpha;
     minRatio = max(maxSize / beta, beta / minSize);
     for each new child in obj
         sumSize += child.size
         get newRatio        //using the same method above
         if newRatio > minRatio
            break;
         else
             add this child into the row //row :{size ,dice, children}
          end if
     end
 end
```

接下来我们会调用***dice***或者***slice***函数，将***row***中元素，排布在当前列（行）中。至于选择哪一个函数，我们只需要比较当前列（行）的总矩形的水平方向和垂直方向的长度即可。事实上，这一点在***row***的***dice***属性中得以呈现。

dice和***slice***函数本质上是一样的，变化的仅仅是方向而已。现在以***dice***为例分析。其实现过程非常简单，让***row***中元素沿着水平方向排布即可。也就是说，垂直方向坐标沿袭***row***的坐标，水平方向按顺序排布。其中$dx = dx' * ownSize/size$ 。这个计算方式是很显然的。最后，我们将各个矩形坐标记录在***data***中。

SquarifyEach

通过上面的分析，我们已经可以实现将一个***children***数组中的元素排布在给定的矩形区域了。

接下来我们要做的就是逐层排布。这自然联想到运用递归

这个函数用以下伪代码表述

```
function squarifyEeach(obj,x0,y0,x1,y1)
{
    if obj has children
        squarify(obj,x0,y0,x1,y1)
        for each child in children
            squarifyEach(child, child.x0,child.y0,child.x1,child.y1);
        end
    end if
}
```

render

这个函数的核心在于动态创建并插入***dom***元素。这一部分用了***xml***的***svg***。为了能够更好的创建元素，还引用了***svg***的命名空间，故功能需在联网时实现。（这里声明一下自己引用命名空间的缘故。首先在我的理解下***xml***及其命名空间不算是某种图形框架，至少不算是***js***的图形框架。其本质和***html***一致，是一种标记语言。故我认为既然可以使用***html***，就也应当允许使用***xml***。另外要求用户联网使用并不过分，正如题目要求，写一个“网页”，故此应用本身就应当联网使用。除此之外，我试过不使用***svg***来实现此功能，如***h5***的***canvas***，但是后者画出的图形是单纯的图形而非***dom***元素，这不适合这个题目背景下的种种功能。我相信，我们应当鼓励选取恰当的方式来适应题目要求。）

在这个函数中，我们根据***data***中存好的坐标，动态向***svg***中插入***<g>***标签，其中包含一个***<rect>***标签和一个用来呈现***name*** 和 ***size*** 的***<text>***标签。这个函数中进行了大量的***dom***操作，事实上这繁而不难，没有必要过多描述。函数中运用了如下方法

```
document.createElementNS   //create new element
element.appenChild          //add element
element.setAttribute        //set attribute
element.innerHTML
```

函数的递归结构如下

```
function render(obj,color,id,func)
{
    if obj has children
        for each child in children
            render(child,color,id,func)
        end
     else
         create,set attribute and add newelement using values of obj
    end if
}
```

另外值得注意的是：由于第一层***children***中各个成员的***color***各不相同，我们需要循环第一层的***chilren***数组，逐个渲染。

Bonus

第一个***bonus***的实现其实很简单，根据功能很容易想到使用***<title>***标签。我们只需要在***render***函数中，向***<g>***中加入一个***<title>***即可。由于有两个信息，故我们用***<tspan>***标签进行换行。这里引用了，***xlink***命名空间。这个***bonus***可能有其他交互方式，可以监听矩形的***dom***，当鼠标放上去时在页面中央呈现一个包含矩形信息的元素。但我认为这样的方式反而臃肿，不如一个***<title>***来的简洁明了。

第二个***bonus***的实现，本质是对事件进行监听。由于时间关系，我没有来得及思考更优的交互方式。于是利用先前的函数，拿出部分***data***，重新***squarify*** 后 渲染到整个***svg*** 内。自己觉得这是一个相对偷懒的做法。同上，我们或许可以尝试制作弹窗等其他交互方式，但是时间不允许我做出更优的解答。

squarify算法部分参考如下文献：

https://zhuanlan.zhihu.com/p/19894525

https://www.win.tue.nl/~vanwijk/stm.pdf

解题过程及期间的心路历程

（意识流写法，旨在叙述心路历程及解题历程）

周五下午拿到题的时候一眼看到了***js***这道题。由于之前对***js***有过些许了解，于是我满怀喜悦的看完了这道题。看完之后，内心顿时有千万头***XXX***跑过。

在不住吐槽之余，内心却也不禁赞叹，此题妙绝，出题人别具匠心。

其一，妙在要求原生***js***。在当今一边看文档一边写前端的时代，越来越多的人离开文档 ***x*** 都不会。我也是这样的人之一。之前，***js***的框架我接触过一些，但这个时候提起原生***js***，我甚至分不清哪些方法是***jq***的，哪些方法是***js***原生的。于是，这样的要求更加直击根本——如何造轮子。事实证明，在解此题的过程中，我也确确实实对***js***对**dom**操作有了更加深刻的理解。

其二，妙在要求设计核心算法。我曾经一度认为***js***就是花架子，写一点前端，设计一下网页而已。然而，此题让我意识到，***js*** 也可以玩出很秀的操作。另外，这个算法同时也是一个很难的算法，这能够让我学到很多。

实际上，这两个精妙之处，也正是此题的困难之处。

周五下午，我就被卡在了第一步——读取文件。由于印象中，***nodejs***中可以直接操作***filesystem***，所以我试图直接操作***filesystem***来获取本地文件，在查阅大量资料之后，仍然没有找到相应的方法，我的心态逐渐爆炸。于是我想要去请教学长，当我把问题输入对话框时：“ 前端***js***不能直接获取本地文件嘛？”，我突然意识到，这个***js***是前端，是跑在浏览

器上的代码。何不直接通过浏览器获取文件呢。想到这里，我茅塞顿开，迅速写完了文件读取和数据预处理部分。

这时已经是晚上。我开始思考算法的实现。很显然，这是***d3*** 框架下的一个功能。而我只想出了一个类似于***slice-and-dice***的算法，并且在实现过程中出现了困难。

我突然想到了报名资料册上写到：“ 快速获取有效信息的能力尤为重要 ”。这么强烈的暗示，让我打开了百度。在查阅***Treemap***的时候，我获悉了这个核心算法的关键词：矩形树图。进而，便在知乎上找到了一篇科普文，其中对***Treemap***的各种算法有较为详细的解释。

周六，我首先实现了知乎所介绍的***squarifed***算法，但是实现后的结果不尽人意（如上图所示，当时还未染色）。又想到了***d3***的***treemap***，于是我花了将近一个小时研究了***d3***的算法。其中最困扰人的部分就是计算***newratio***的过程。思考很久之后我才明白这原来就是每一列内部矩形的长宽比。

至此，我也明白了为什么自己实现的那个算法存在问题。在我的算法中，只考虑了整列（行）的矩形的长宽比而未考虑内部矩形的长宽比，单纯粗暴的把小矩形塞进去，这就导致出现了长宽比很大的矩形。

在写代码时先后出现了几个***bug***，第一个就是***for***循环中的 ***i*** 未使用局部变量。循环体中的递归函数使得 ***i*** 的值不断变化，导致运行失败。在***debug***过程中才发现了***i***的奇异变化。第二个***bug***则是在实现算法的时候忘掉更新***dx***和***dy***的值，使得最后得到的矩形超出屏幕。在查看***console***的时候发现了数据中坐标值很大，检查代码才发现这一问题。

之后的一个困惑就是用什么来实现图形。在查资料的过程中，我发现了***svg***这一标签，觉得非常适合这个题目。然而在实现的时候发现常规的***dom***操作例如 ***document.createElement()***方法并不能够正常运行。我查找原因，知道了是***xml***命名空间的锅，但是我又不确定这个算不算所谓框架，就先放在那里去做其他题了。

周一晚，我决定做完它。考虑到***bonus*** 的功能，我最终还是选择了***svg***来实现绘制矩形。并且也想出了说服自己的理由。

在做这个题的整个过程中我都是精神抖擞的，除了在***debug***近一个小时后发现忘掉更新数据时的剁手的冲动以外，我一直都很享受这个过程。并且我也确确实实学到了很多东西。