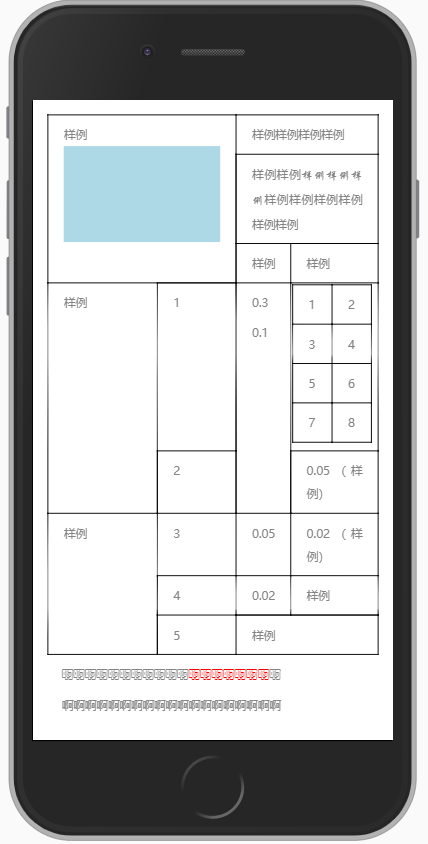
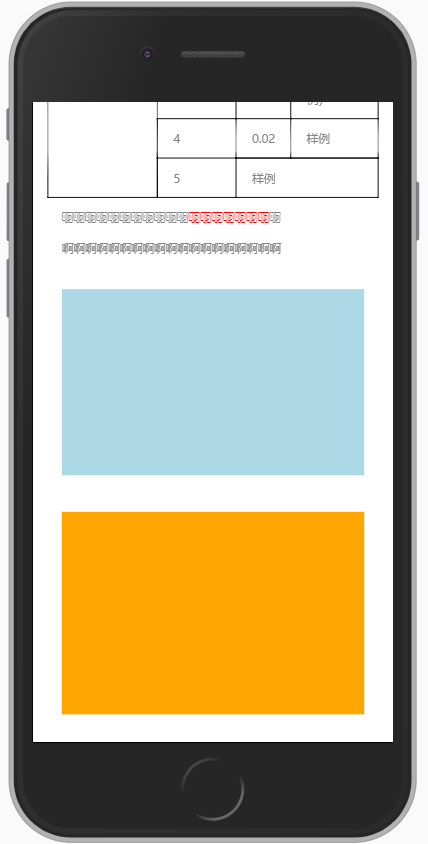
动机：

保险行业有各种各样的条款，如何展示这些文字内容巨多但又非常重要的条款是我们面临的一个关键问题，目前有两个方案：第一个以pdf的形式展示，这种方案是借用一些pdf插件来展示条款，由于目前项目中主要的场景是用户通过移动端来查看条款，因此是存在一些问题的，比如直接展示pdf文档无法随着移动端设备的变化自适应的进行排版以达到清晰的呈现。第二个以html静态文件的形式来展示这些条款，可想而知，在浏览器上渲染html文件必定优于在浏览器上展示pdf文件，因此第二种方案具有速度快、能自适应不同设备、体验好的优点。

问题：

给的文件是pdf文件或者是word文件，通常一个条款有四十多页，如果人工复制粘贴，可想而知时间成本巨大。并且更糟糕的是里面还有大量的表格，如下：





目标：

将word转成特定样式的html文件。

方案对比：

1. 在线转换工具 <https://wordhtml.com/> ：基本满足要求，但是在word较为复杂的时候，比如word中有列表，会表现不正常，需要手动调整，不支持字体字号与颜色的转换。
2. Mammoth包：基本满足要求，不支持字体字号与颜色的转换，未发现其他问题。
3. word-to-html：支持表格嵌套，表格合并。

word-to-html

使用方法：

step 1: 在node环境中执行下面代码，构建一个html文件用于生成最终结果

var path = require('path');

var word2html = require(word-to-html');

//Word document's absolute path

var absPath = path.join(\_\_dirname,'test.docx');

word2html(absPath,{tdVerticalAlign:'top'})

step 2: 打开上述构建的html代码，在浏览器的console面板中得到最终的字符串。

将.docx文件通过node插件转换成xml字符串，然后利用jsdom在node环境下将xml字符串解析成为xml节点树，或者利用在browser环境下的DOMpaser将xml字符串解析成xml节点树，最后深度遍历xml节点树构建html文档。在browser环境下支持字体字号与颜色的转换。

1、利用插件adm-zip将docx文件转换成xml字符串contentXml，然后利用包含script脚本的模板构建html并写入对应的html文件中。这里convert方法是核心。

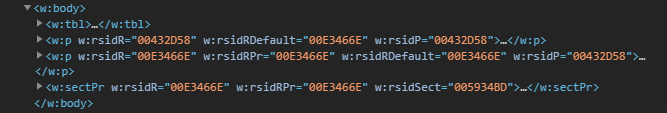
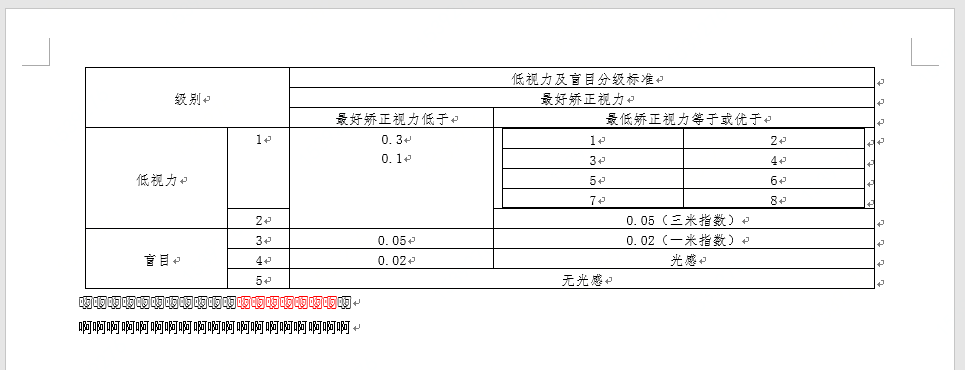


2、convert方法的代码如下，主要分两步，解析xml字符串生成xml节点树，处理xml节点树生成html字符串。

第一步：利用loadXML函数解析xml字符串生成xml节点树，loadXML做了兼容性处理，对于IE浏览器，利用ActiveXObject对象解析XML字符串。对于chrome浏览器（webkit）利用DOMpaser进行解析。



解析的结果如下：



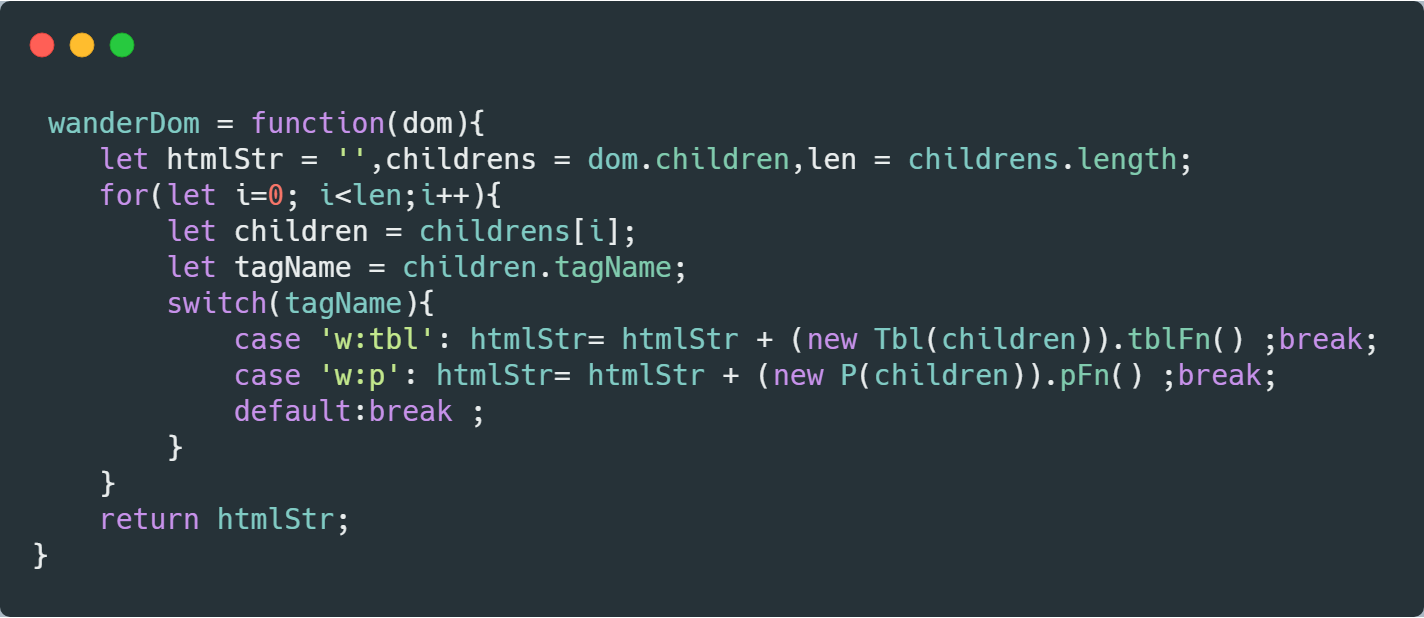
这里docx文件中从上到下分为三块：表格，文字，文字；节点树中分别对应<w:tbl>，<w:p>，<w:p>。再仔细观察可以找到规律，分为这三个标签的依据是回车换行符。

第二步：利用convert函数处理xml节点树生成html字符串

这里convert的逻辑很简单，获取上一步解析好的xml根节点的body节点，当然这里少了对null的处理，弱类型语言的缺点，下一步用ts进行类型检测。然后传入wanderDom函数，wanderDom的结果即最终的html字符串。



下面看对wanderDom函数对整个文档的body节点进行处理：



这里会对body节点进行深度遍历，首先循环遍历body节点的子节点，对于子节点的tagName为w:tbl的，生成一个Tbl实例，并调用实例上的tblFn开始处理表格节点，并将返回拼接到htmlStr。对于子节点的tagName为w:p的，生成一个P实例，并调用实例上的pFn开始处理文字节点，并将返回拼接到htmlStr。最终返回拼接好的htmlStr。接下来分析两种对象的处理逻辑。

P实例：对纯文本的处理

C:\Users\liaohuanyu\Documents\JD\office_dongdong\liaohuanyu\Temp\JdOnline20190731172200.png

上面的文本可以分为三部分，分别为黑色文字，红色文字，黑色文字。这三部分内容对应着三个标签<w:r>。每个<w:r>标签中的<w:rPr>包含了文字的信息，比如字体字号<w:rFonts>，颜色<w:color>。每个<w:r>标签中的<w:t>的内容textContent为文本内容。因此处理文字内容的逻辑如下：



这里构建一个P类用于处理纯文本相关的内容。其实例方法pFn逻辑很简单，利用getDirectDomsByTagName函数获取<w:p>标签下所有的<w:r>标签对应节点组成的数组，然后将这个数组传入rFn，将rFn返回的结果用html的<p>标签包裹起来并返回。这里的getDirectDomsByTagName只会获取根节点dom下第一层节点，保证递归正常进行下去。

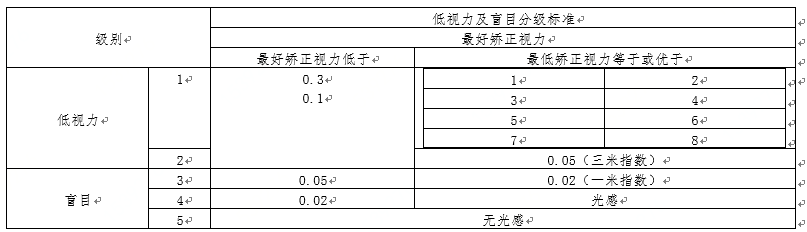


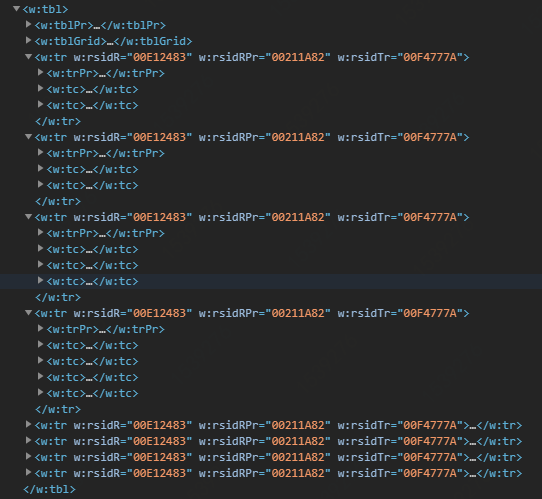
接下来，来到处理文字内容的逻辑函数rFn，这段逻辑主要是取出传入的<w:r>数组中每个部分的文字的字体字号颜色的值，然后用<span>标签包裹，最后返回html字符串。



这样递归的将文字进行了处理。

Tbl实例：对表格的处理





Docx文件中表格中的行对应的标签是w:tr，每一行中的列对应的标签是w:tc，这里表格有一个合并单元格的效果，因此可以发现每个w:tr中的w:tc的数量不相同。在html中控制表格单元格合并的操作是在<td colspan=${num} rowspan=${num}>，这两个属性分别控制着列方向的合并以及行方向的合并。上图中的w:tc中都有一个w:tcPr标签，包含描述这一列应该如何合并的信息。w:tcPr标签的子标签中描述列如何合并的标签为w:gridSpan,其属性w:val的值表示的是colspan的值。描述行如何合并的标签为w:vMerge,该标签上的w:val如果为restart则表示从本行开始该单元格合并下一行相同位置存在w:vMerge的单元格。



表格中存在文本，因此这里遇到w:p只需要调用上面对纯文本的逻辑。表格中也存在表格，因此遇到w:tbl，则调用这里将要分析的对表格的处理逻辑。

首先看Tbl类的主体实现：



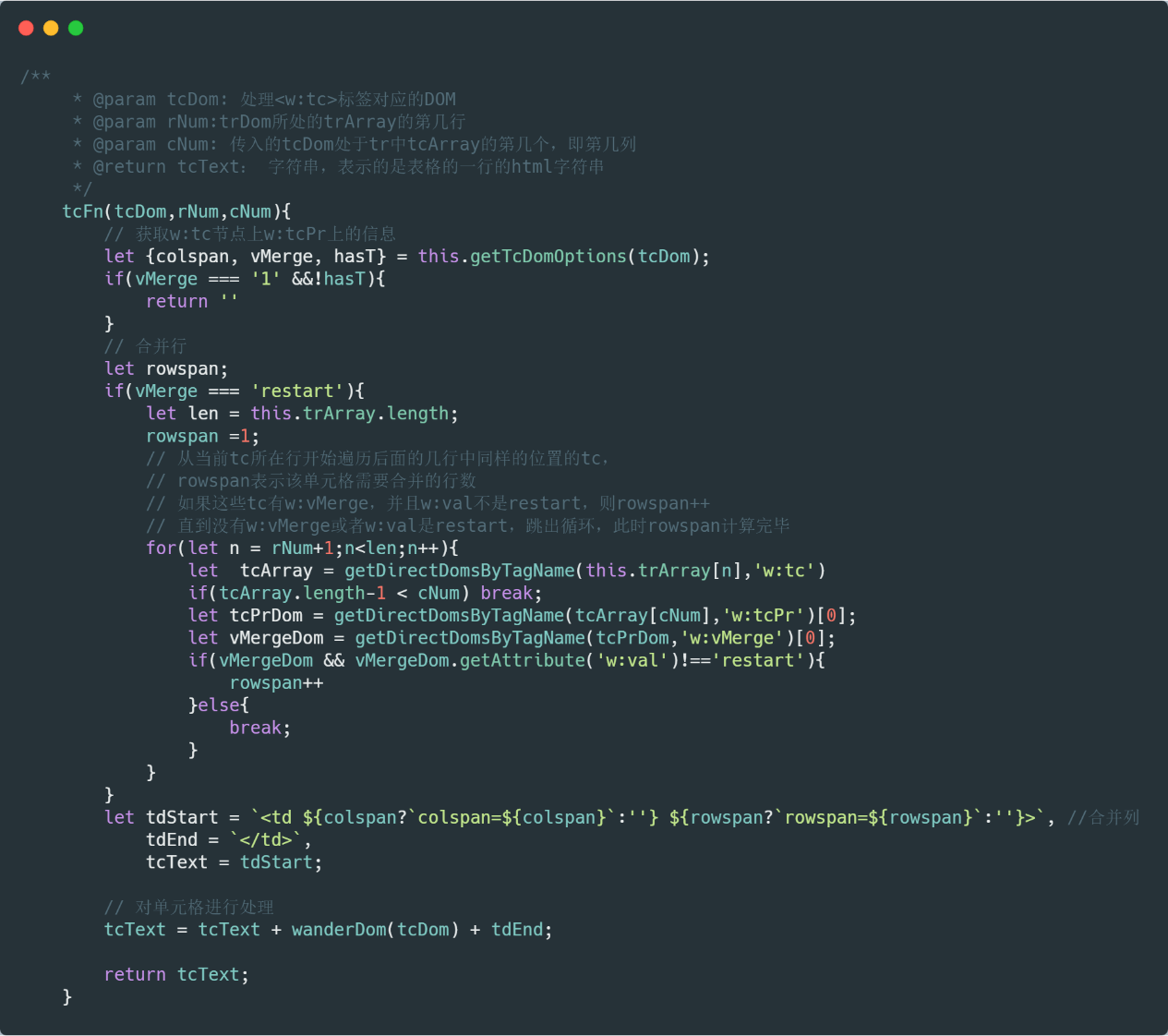
Tbl类有几个实例方法与一个用于存储表格节点的tblDom实例属性，一个用于存储<w:tr>节点组成的数组的trArray实例属性（实例方法都能访问到的公有属性），创建Tbl实例的同时，需要调用实例方法tblFn，该方法会遍历w:tbl下的子标签w:tr对应的节点，将节点被传入trFn进行处理，tblFn返回w:tbl标签对应的html字符串。



trFn会遍历其子节点w:tc并调用tcFn处理每个w:tc对应的节点，trFn返回的是表格值每一行的对应的html字符串，tr字符串内容在这里组装完成。接着，tcFn函数非常关键，包含处理嵌套表格，合并单元格的逻辑。

主要有以下的步骤：

1. 调用getTcDomOptions函数获取该单元格的w:tcPr的值并计算出该单元格需要合并多少行多少列
2. 然后调用wanderDom对该单元格的内容进行处理。



w:tc表示的单元格有下面几种情况：

1. 合并其他单元格（包括横向和纵向）：
   1. 单元格内容为纯文本，
   2. 单元格内容为表格，
   3. 单元格内容有表格也有文本
   4. 单元格没有内容

2、被合并



最终的结果：

