前言

解决方案

解析 HTML 结构

解决方案中涉及的 HTML 文档大体简单，有少量的表格和公式表达式以及图片等较复杂的元素，但大部分都是纯文本类型，第一步将 HTML 文档转化为 Json 字符串的步骤并不复杂，利用递归方法遍历 DOM 结构即可生成 Json 字符串。

HTML 整体被抽象成为三部分，Content，Header 以及 Footer，它们属于同一层级，是一种抽象类 Element 的数组对象 Elements[] ，Element 类还包括一些特殊元素，如分割线，表格等。

在 Element 中包含了由 Fragment 对象组成的 Segment[] 数组，下面的代码是 Element 和 Fragment 的类型定义和一个实例。

```
var ElementType = {
    段落: 0,
    表格: 1,
    线条: 2,
    列表: 3,
    分割线: 4,
}

var FragmentType = {
    图片: 0,
    文本: 1,
    表达式: 2,
    自定义表格: 3,
    录入框: 4,
    列表的标记: 5
}
```

```
{
    "Elements":[
        {
            "ElementType":0,
            "X":8.333333333333332,
            "Width":748.0145833333335,
            "TextAlign":"center",
            "Indent":0,
            "ParaIndent":0,
            "Segments":[
                {
                    "FragmentType":1,
                    "Content":"n",
                    "FontName":"宋体",
                    "FontStyle":0,
                    "FontSize":11,
                    "FontColor":"black"
                }
            ]
        }
    ],
    "Header":{
        "Elements":[
            {
                "ElementType":0,
                "X":8.333333333333332,
                "Width":747.9166666666667,
```

```
                "TextAlign":"right",
                "Indent":0,
                "ParaIndent":0,
                "Segments":[
                    {
                        "FragmentType":0,
                        "Width":73.95833333333334,
                        "Height":32.29166666666667,
                        "Content":"/upload/image/20210419/637544505136957073296388 8.jpg"
                    }
                ]
            },
        ]
    },
    "Footer":{
        "Elements":[
            {
                "ElementType":2,
                "Color":"rgb(0, 0, 0)",
                "Weight":1.0416666666666665,
                "LineStyle":"solid"
            }
        ]
    }
}
```


从实例中可以看出 `Element` 控制了位置信息，大小信息以及对齐方式等元素定位的要素，`Fragment` 则控制了具体节点的样式和类型。

C# 中数据类型的定义

初步的方案涉及几个关键的类

```
abstract class Element {
    // 元素类型
    ElementType ElementType { get; set; }
    // 元素基础行高
    float Height { get; set; }
    // 元素基础宽度
    float Width { get; set; }
    // 获取元素打印结果，若为 False 则进行分页处理
    abstract bool Print(Graphics graphics);
    // 初始化元素计算所需宽高，不同类型的元素具有不同的初始化方法
    abstract void Init(Graphics graphics);
    // 用于初始化 Header，Footer 时重置 ptr 的值
    abstract void Reset()
}
```

第一部分是 `Element` 接口，上文提到的几种类型的 `Element` 在 C# 代码中实现了 `Element` 接口，在 Json 反序列化到对象的过程中，声明对象的过程中完成了不同子类 `Init()` 函数的实现，并在后续打印文档的代码中统一递归执行 `Init()` 函数，完成初始化

```
// Json 对象对应的实体类
class EmrDocument {
    // 定义的 DOM 结构中的元素，有 Table，Line，Paragraph 等子类
    List<Element> Elments
    Header Header;
    Footer Footer;
    // 代表List中元素的下标，标出目前操作的元素
    int ptr;
}
```

第一个属性 `Elements` 是一个实现了接口 `Element` 的 `List` 集合，`Header` 和 `Footer` 和 `EmrDocumnet` 类类似，内部也有一个 `Element[]`

```
// 用于控制GDI绘图的坐标以及相关因素的抽象类
class Helper {
    // 初始化 Helper 类
    static void Init(Graphics graphics) {
        // 获取绘制面板引用
        gs = graphics;
        // PaperSize 是一个封装文档文档页面布局的类
        // 获取页面左边起始打印位置
        curx = Helper.PaperSize.left;
        // 从页面顶部开始打印
        cury = 0f;
        // 避免二次初始化
        // 通过计算在一个绘制面板上同样的字体大小不同字数字符串的宽度来计算字间距
        if (!isInit)
            TextMarginRatioWithFontsize = GetTextMarginRatioWithFontsize(graphics);
        isInit = true;
    }
};
```

绘图的实现

核心类 `SingleLineBuffer`，它实现了一个 `Print()` 方法用来最终输出到 Graphics 上

```
class SingleLineBuffer {
    // 行数
    // 主要用于判断首行缩进
    private int lineCount = 1;
    private float LineWidth = Helper.PaperSize.GetValidWidth();
    // 水平对齐方式
    private StringAlignment Alignment = StringAlignment.Near;
    // 垂直对齐方式
    private StringAlignment LineAlignment = StringAlignment.Near;
    // 首行缩进
    // </summary>
    public float Indent = 0;
    // 段落缩进
    public float ParaIndent = 0;
    // 行高
    public float LineHeight = 0;
    // 单行内的组成元素
    internal List<Fragment> fragments = new List<Fragment>();
    internal bool Print(Graphics gs) {
        // TLDT
    }
}
```

在打印文档的过程中完成了初始化，进入打印流程时，所有的 Element 都会借助 Helper 类来进行行的打印

```
Helper.CurLine = new SingleLineBuffer(this);
// 这个Print方法是SingleLineBuffer实现的
Helper.CurLine.Print(graphics);
```

在 `SingleLineBuffer` 的 `Print()` 方法中，首先判断行高和剩余页面高度的大小，如果无法在本页打印该行则返回 false

```
//打印之前先判断本行能否在本页绘制完毕
//若不能绘制完毕，则表示需要分页[页眉/页脚是个特例，不需要判断分页]
if (Helper.cury + lineSize.Height > Helper.PaperSize.top + Helper.PaperSize.GetValidHeight() &&
!Helper.IsDrawingHeaderOrFooter) {
    //需要分页
    return false;
}
```

确认了本页剩余空间后开始计算 `Helper.curx`，具体是通过判断计算首行缩进，段落缩进以及对齐方式来获取当前绘制起点的横坐标

完成了横坐标计算则标志本行的绘制初始化工作完成，然后开始计算下一行的起始位置，即 `Helper.cury`，和横坐标的计算方式类似，因为 `Fragment` 对象中包含了碎片的定位信息，直接调用即可

最后调用 `Fragment.Draw(Graphics gs, float beginx, float beginy)` 方法

这会先获取设置的字体信息，然后调用 GDI 的 `Graphics.DrawString(String s, Font font, Brush brush, float x, float y)` 完成 `Fragment` 的绘制

```
internal override void Draw(Graphics gs, float beginx, float beginy) {
    var font = GetFont();
    gs.DrawString(this.Content, font, Brushes.Black, beginx, beginy);
}
```