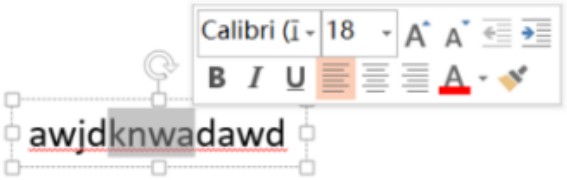
课件新版文本组件详细设计

引言

1. 功能概述

文本组件是课件的基础组件，包含文字排版，文字编辑，设置文字样式，设置段落样式，设置光标，设置选区等基础功能，示例如下：



2. 使用场景

文本组件在课件 pc 端，课件 android 端，互动 android

应用上均有使用，在设计时需要考虑跨端和跨应用的解决方案。

3. 设计目的

在之前的设计中，我们依据课件播放态和编辑态的两种技术选型——qt / web，采用了不同的方式来控制文本的样式和布局。播放态（qt）通过访问本地字体文件和使用相关文字库进行排版，编辑态（web）则基于浏览器默认提供的规则进行排版。尽管能满足多数普遍的应用场景，但也有一些难以被解决的问题，比如播放态能够控制文字的标点悬挂和字间距（跟 office ppt 一致），编辑态则缺少对应的能力（浏览器没有提供）。于是，我们开始调研新版文本组件的实现方式，旨在保持播放态和编辑态在文字排版上的统一（也包括 pc 端和 android 端的统一）。

逻辑架构概览

分层架构

为了更容易理解问题和更好的扩展性，我们对新版文本组件进行如下分层：

1. 视图分层

视图方面，我们将新版文本组件分为渲染和编辑两个组件：

- 渲染组件只负责渲染文本
- 编辑组件负责编辑文本

在多数场景下，渲染和编辑是分离的，比如课件播放态，课件 android 端，以及互动 android

应用，都是只需要渲染，不需要编辑。只在课件编辑态，才既需要渲染也需要编辑。

在旧版文本组件中，我们也是这样做的，渲染组件是一个名称叫 <TextPreview /> 的组件，编辑组件是一个名称叫 <TextEdit /> 的组件，通过判断是否可编辑来切换两个组件。示例如下：

Plain

```
<template>
  <TextEdit v-if="editable" />
  <TextPreview v-else />
</template>
```

看起来没啥问题，效率却很低，因为它是整体切换的，根据我们上面对场景的分析，可以发现我们只有：1. 只渲染；2. 既渲染又编辑两个场景，所以更合理的视图分层应该是这样的：

Plain

```
<template>
  <TextPreview />
  <TextEdit v-if="editable" />
</template>
```

当然，想要实现这样的分层并不容易，因为在传统意义上，我们没有办法将编辑行为从浏览器独立出来，我们一般都是需要借助浏览器的交互式控件来实现编辑的，比如 input，textarea，contenteditable，而这些交互式控件无一例外都将渲染和编辑绑

定了。我们在新版文本组件中提出了一种新的编辑思路：捕获鼠标事件，自绘光标和选区，同时在光标位置，隐藏一个交互式控件来捕获键盘输入，以此将渲染和编辑分离。于是，视图变成了这样：

Plain

```
<template>
  <TextPreview />
  <div v-if="editable" >
    <TextSelection />
    <Input style="opacity:0;" />
  </div>
</template>
```

技术选型

1. 模型

1.1. 文档模型

文档模型是指存储在课件文件（.ecw）中的数据模型，这里采用 TextBody 模型。

TextBody 是一个 3 层模型，它包含文档-段落-文本 3

层结构，最上层是文档，一个文档包含多个段落，每个段落中有至少一个文本。

参考如下：

文档：ITextBody

```
export interface ITextBody {
  props?: ITextBodyProps
  textStyleList?: ITextStyle[]
  paragraphList: IParagraph[]
}
```

段落：IParagraph

文本：IParagraphItem

TextBody 模型也是我们在旧版文本组件中使用的模型，新版保持不变。

在旧版中，文档模型在渲染时，需要按照一定的规则去外部动态继承属性，这是 ppt 的重要特性之一，新版也会体现。

1.2. 排版模型

排版模型是新引入的概念，它是指文档模型（TextBody）经文字引擎处理后的结果，通常包含渲染所需的排版信息，它也是一个多层结构。

补充公式的信息

参考如下：

```
Plain
```

```
{
    "data" : {
         "blocks" : [
              {
                   "lines" : [
                        {
                             "fragments" : [
                                  {
                                       "index" : 0,
                                       "words" : [
                                            {
                                                 "gcps" :
0,

    "paintPos" : {
                                                      "x"
: 0.00000000000000000,
                                                      "y"
: 61.06987951807228399
                                                 },
                                                 "pos" : {
                                                      "x"
: 19.82677165354327187,
                                                      "y"
: 61.50019922208518608
                                                 },
                                                 "size" :
{
                                                      "h"
: 76.79999999999999716,
                                                      "w"
: 76.00000000000000000
                                                 },
```

```
                                                                "text" :
"白"
                                                            }
                                                        ]
                                                    },
                ],
                "pos" :
                                        {
                                                    "x" :
14.17322834645669261,
                                                    "y" :
7.37007874015747966
                                        },
                                        "size" :
                                        {
                                                    "h" :
75.84000000000000341,
                                                    "w" :
390.65354330708657926
                                        }
              }
            ]
          }
      ]
    },
    "id" : "04393745"
}
```


排版模型和文档模型的结构并不完全一致，但也遵循了最基本的文档-段落-文本的分层结构，data 对应文档（ITextBody），blocks 对应段落（IParagraph），fragments

对应文本（IParagraphItem），同时，它还多了排版所需的 lines 和 words 的概念。

1.3. 渲染模型

排版模型并不能直接用于渲染，它仍缺少一些基本的样式属性（比如 color），于是，我们在内存中计算了一个新的渲染需要的数据模型，它依据排版模型的结构并按照一定的规则去文档模型中继承了必要的样式属性。

Plain

```
{
  "data" : {
    "props": ?, // 继承自 textbody 的 props
    "blocks" : [
      {
        "props": ?, // 继承自 paragraph 的 props
        "lines" : [
          {
            "fragments" : [
              {
                "props": ?, // 继承自 paragraphItem 的 props
                "index" : 0,
                "words" : [
                  {
                    "gcps" : 0,
                    "paintPos" : {
                      "x" : 0.00000000000000000,
                      "y" : 61.06987951807228399
                    },
                    "pos" : {
```

```
                    "x" : 19.82677165354327187,
                    "y" : 61.50019922208518608
                  },
                  "size" : {
                    "h" : 76.79999999999999716,
                    "w" : 76.00000000000000000
                  },
                  "text" : "白"
                }
              ],
            },
          ],
          "pos" :
              {
                "x" : 14.17322834645669261,
                "y" : 7.37007874015747966
              },
              "size" :
              {
                "h" : 75.84000000000000341,
                "w" : 390.65354330708657926
              }
        }
      ]
    }
  ]
},
"id" : "04393745"
}
```


继承的 props：

1. 文档样式属性

props	类型	说明	来源
writingMode	string	"hort"：横排、"eaVert"：竖排、"wordArtVert"：堆积从左到右	textVert

1. 段落样式属性

props	类型	说明	来源
bullent	Object	项目符号或编号	bullent、buNone

1. 文本样式属性

props	类型	说明	来源
fontStyle	string	'italic': 斜体 normal’ :正常	italic
fontWeight	string	'bold': 加粗 ‘normal’ :正常	bold
underlineColor	string	下划线颜色	underlineColor
fill	string	字体颜色	background
fontFamily	string	字体	lang、font
fontSize	string	字号	fontScale、size
shadow	object	文字阴影	**outerShadow**

部分 props 属性依赖多个值的计算结果。

1.4. 渲染模板

理论上，我们已经能够基于 1.3

章节的渲染模型去正常的渲染文本组件了，但考虑到性能问题，这里还需要做一个优化。

我们做了一个试验，分别测试动态渲染和静态渲染的效率。

- 动态渲染：基于 vue 的模板语法去渲染，比如

Plain

```
<template>
  <!-- 文本组件 -->
  <TextBody>
    <Paragraph>
      <TextLine>
        <Fragment />
      </TextLine>
    </Paragraph>
  </TextBody>
</template>
```

- 静态渲染：事先拼装好模板一次性渲染，比如

Plain

```
<template>
  <!-- 文本组件 -->
  <g v-html="" />
</template>
```

参考如下：

a. 同时渲染 20 个组件

动态渲染耗时 50 ms，静态渲染耗时 17 ms。

a. 同时渲染 10000 个组件

动态渲染耗时 38089 ms，静态渲染耗时 4027 ms。

很显然，我们应该采用更优的静态渲染的方案。

静态渲染的缺点是没有办法局部更新模板中的一个部分，但考虑到我们当前的排版方案，发生编辑行为时，往往需要整体更新，所以更契合静态渲染的方案。

1.5. 文档模型-排版模型-渲染模型-渲染模板的关系

我们在上面列举了新版文本组件用到的 4 个核心数据模型，它们的关系如下：

我们引入了文档 hash

的概念，可以保证我们在重复渲染一个文档模型时的效率最大化，比如以下几种场景：

a. 缩略图和舞台区重复渲染一个文档模型

b. 切页时重复渲染一个文档模型

c. 拷贝时重复渲染一个文档模型

d. 撤销重做时重复渲染历史文档模型

e. 编辑态和 web 播放态重复渲染一个文档模型

这里我们使用了 xxHash 算法，它的效率和其他 hash 算法的效率比较：

1.6. 选区模型

在分层架构章节，我们介绍了新的分层模型，它能够实现渲染态和编辑态在文字排版上的统一，却也带来了新的问题，比如，我们不得不放弃浏览器提供的 contenteditable 特性，自己去实现一套光标和选区的绘制逻辑。

选区模型就是我们描述光标和选区位置的模型。光标和选区本质上相同，起始位置等于结束位置就是光标，反之就是选区。

整个文本组件中，存在两个选区模型，一个叫选区渲染模型，一个叫选区编辑模型。

- 选区渲染模型是渲染光标和选区用的，包含了必要的坐标信息；
- 选区编辑模型是编辑文档用的，包含了必要的索引信息。

二者表现不同，却始终一致。

1.6.1. 选区渲染模型

Plain

```
selection = {
```

```
  range: [gcps1,gcps2], // gcps1:选区起始位置
gcps2：选区结束位置
  clients: [
    {
      x: 0,
      y: 0,
      width: 0,
      height: 0,
    },
    ....
  ]
}
```

- range：选区的起点和终点，主要是用来记录选区的路径信息
- clients：一个数组，数组中的每个对象是一个矩形，主要是用来绘制选区

在排版模型和选区模型中，我们都用到了 gcps 这个概念，它是一个 number 值，代表每个字在文档中的索引。我们在渲染页面的时候会把 gcps 挂载到 node 节点上，示例如下：

Plain

```
<text>
  <tspan gcps="0 1 2 3 4 5 6 7 8 9 10
11">awdawdawdawd</tspan>
</text>
```

选区交互参考：

1. 鼠标落下的点转换 range

```
Plain
// 鼠标落下的点转换range
nativeRangeToRange(startNode,startOffset, endNode,
endOffset) {
// 获取node上的gcps属性
....
return [gcps1,gcps2]
}
```

caretRangeFromPoint 的兼容性：

tips：鼠标位置转换 range 并不是只有这一种方法，如果我们发现 caretRangeFromPoint 的效率并不高或存在兼容性问题，我们也可以通过比如坐标值比较的方式来拿到 range。

1. range 转换 clients

2. 如何根据两个 gcps 找到 clients?

   c. 确认 gcps1 在渲染数据模型的所在行，判断 gcps2 与所在行行尾文本的 gcps3

   d. 如果 gcps2<=gcps3，则确定所在行，如果 gcps2>gcps3，代表跨行了，新增下一行的数据

   e. 重复 b 操作，直到 gcps2 处于所在行的行内

6. 如何计算详细的位置信息?

g. x,y: 取所在选区最左边的文字的x, y

h. width: 取所在选区的每一个文字的宽度和

i. height: 取所在选区的所在行的高度

Plain

```
rangeToClients(range) {
// 获取起始gcps
const [gcps1, gcps2] = range
// 确认gcps1,gcps2在内存模型的具体位置
...
// 获取选区clients
...
}
```

1.6.2. 选区编辑模型

Plain

```
selection = {
  anchor: { // anchor代表选区的起点
    path: [0,1], //
path代表在数据中查找路径，在textBody中即指代第1个paragraph的第2个it
em
    offset: 3, // offset代表在item中的偏移
  },
  focus: { // focus代表选区的终点
    path: [1,3],
    offset: 4,
  }
}
```

此模型在编辑数据的过程中使用，path 和 offset 的结构更有利于我们对 textBody 的节点进行插入/删除/合并/拆分等操作，这里看似使用了完全不同的数据结构，实际上它指向的 gcps 和 range 里面的一模一样。

选区渲染模型和选区编辑模型的关系如下：

2. 通信

2.1. 和文字引擎通信

我们使用 websocket 和文字引擎通信。

因为需要同时渲染多个文本组件，所以我们将和文字引擎通信的功能封装在 sdk-core 里面。

websocket

是异步的，天然可以实现我们现在对文字做的后渲染的处理效果，提升首屏体验。

2.1.1. 管理通信

在 sdk-core 里面提供管理通信的 ws

模块（包括建立通信，维持通信，取消通信等），通过 api 的方式对外暴露 send 接口，文本组件在渲染的时候调用 send 接口向 core 发送 textBody 数据和

Hash，如果缓存里没有, core 就会向文字引擎发送 textBody 数据，receive 的数据经过加工（排版模型 to 渲染模型 to 渲染模板），最终会放到全局状态机里管理，文本组件订阅状态机里的数据。

tips：这里还有一些细节就不赘述了，在编码的时候会体现，比如如果某个文档 Hash 不在缓存里，但在消息队列里（已经 send 等待 receive），也不会多次发送。另外，我们仍在考虑更多提高通信效率的方案，比如提前发送，压缩算法等。

2.1.2. 优化

和文字引擎的通信是高消耗的，在连续输入的时候，我们可能并不会每次都和引擎通信，也会考虑通过文字预排版的方案来提升编辑体验。比如，我们能够通过 wasm 的库获取到常见中英文字符的宽高，在连续输入的时候，可以进行预排版。

这是后期的优化方案。

模块设计

1. 文本组件

视图层采用 svg 进行排版。

基本示例：

Plain

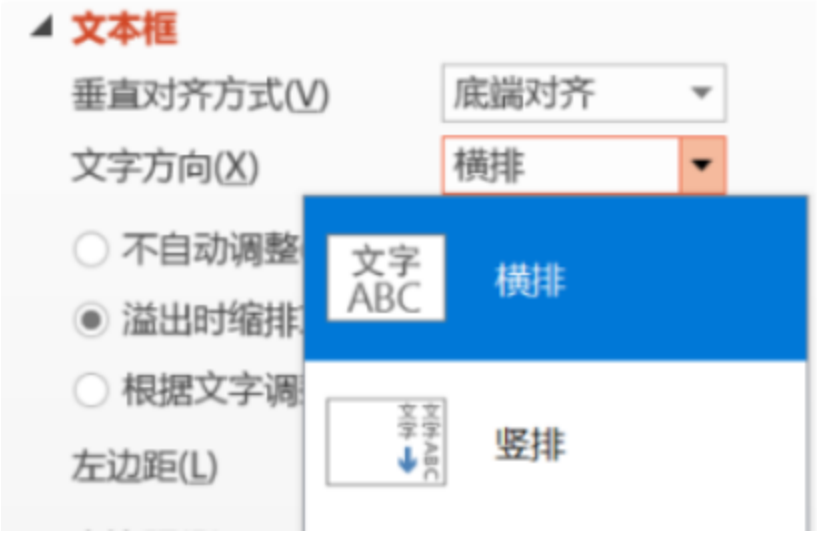
```
<template>
  <TextBody>
    <Paragraph
      v-for="(paragraph, i) of blocks"
      :key="i"
      :paragraph="paragraph"
      :textBlock="textBlocks[i]"
      >
      <TextLine
        v-for="(lineItem, i) of lines"
        :key="i"
        :lineItem="lineItem"
        :paragraph="paragraph"
        >
        <Fragment v-for="(item, i) of fragments" :key="i"
class="fragment" />
      </TextLine>
    </Paragraph>
  </TextBody>
<template>
```

<TextBody> 由多个<Paragraph> 组成，<Paragraph>由多个 <TextLine>

组成，一个 <TextLine> 含有多个<Fragment>。

1.1. 整体属性

1.1.1. 横排/竖排



文本竖排的属性在视图的表现上，有以下特点：

- 竖排文字方向从上到下，起始行在最右；横排是从左到右，起始行在最上方
- 西文字符旋转了 90°，中文字符不旋转

tips：关于文字旋转，语言场景比较多，如大部分亚洲国家的文字（中文、日语、韩语和蒙古语）不需要旋转，英语、阿拉伯数字需要旋转，后期根据需要具体调研。

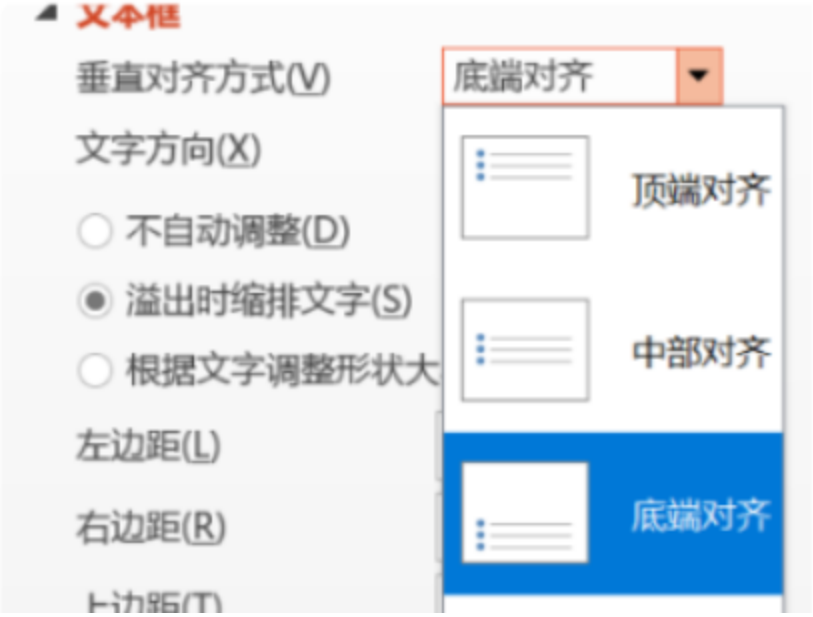
- 鼠标悬停时，鼠标样式为横向。

因此，对于竖排的场景：

- 渲染结构可复用基本示例；
- 文字可通过设置坐标以及旋转角度来排版（通过匹配特定字符来做旋转）；

- 鼠标样式可通过设置CSS样式 cursor: vertical-text 来处理。

1.1.2. 垂直对齐方式



由于 svg 的排版特性，垂直对齐方式仅需要设置整体文字容器的相应坐标即可。

1.1.3. 边距

文本内容与文本容器之间的边距，此特性直接使用文字引擎计算后的结果进行排版即可实现。

1.1.4. 自动换行

由于 svg

中没有控制自动换行的属性，因此需要计算出文字是否超出某个设置的宽度，并将之后的文本渲染到下一行。此特性直接使用文字引擎计算后的结果进行排版即可实现。

1.2. 段落属性

1.2.1. 水平对齐方式

水平对齐包含左对齐、居中对齐、右对齐、分散对齐、两端对齐。这些对齐方式依赖文本区域的大小，需要根据段落和文本区域的大小计算出文本的坐标，其中两端对齐和分散对齐还会影响字与字的间距。

Plain

```
<g>
  <g class="paragraph">
          <!--水平对齐会体现在line的x值上 -->
    <g class="line" transform="translate(x 0)">
      <g class="fragment">
          <text>
          <!--两端对齐、分散对齐还会体现在span的x、y属性上 -->
          <tspan x='x1 x2 ... xn' y='y1 y2 ...
yn'>水平对齐的文字</tspan>
          </text>
    </g>
    </g>
    </g>
</g>
```

1.2.2. 行间距

svg 中，行与行之间的间距通过控制行的 y 坐标实现。

Plain

```
<g>
  <g class="paragraph">
    <g class="line-row" transform="translate(0 0)">
      <g class="fragment">
        <text>水平对齐的文字</text>
      </g>
    </g>
    <!-- 通过 y 来控制行间距 -->
    <g class="line-row" transform="translate(0 y)">
      <g class="fragment">
        <text>水平对齐的文字</text>
      </g>
    </g>
  </g>
</g>
```

1.2.3. 段落缩进

svg 中，通过控制段落中每一个行的 x 坐标与 width 实现。

Plain

```
<g>
  <g class="paragraph">
    <!--段落缩进体现在每一行的x属性上 -->
    <g class="line-row" transform="translate(x 0)">
      <g class="fragment">
        <text>水平对齐的文字</text>
      </g>
    </g>
    <!--段落缩进体现在每一行的x属性上 -->
```

```
    <g class="line-row" transform="translate(x y)">
      <g class="fragment">
        <text>水平对齐的文字</text>
      </g>
    </g>
  </g>
</g>
```

1.2.4. 首行缩进、悬挂缩进

Plain

```
<g>
  <g class="paragraph">
        <!--首行缩进和悬挂缩进体现在段落第一行的x属性上 -->
    <g class="line-row" transform="translate(x1 0)">
      <g class="fragment">
        <text>水平对齐的文字</text>
      </g>
    </g>
    <g class="line-row" transform="translate(x2 y)">
      <g class="fragment">
        <text>水平对齐的文字</text>
      </g>
    </g>
  </g>
</g>
```

1.2.5. 符号、编号

Plain

```
<g>
  <g class="paragraph">
    <g class="line-row">
      <!-- 段落的第一行放置段落符号、编号 -->
```

```
      <text>a.</text>
      <!-- 其之后的文字需要相应的偏移 -->
      <g class="fragment" transform="translate(x1 0)">
        <text>水平对齐的文字</text>
      </g>
    </g>
  </g>
</g>
```

1.3. 文字属性

部分文字属性有现成的 svg 属性直接设置即可，与CSS属性基本一致。

属性	实现方式	备注
颜色	fill	
字体	font-family	
字号	font-size	
字体样式	font-style	应用的字体的样式（如：斜体）
加粗	font-weight	

部分属性没有直接可用的 svg

属性，或者可以的属性无法满足要求，比如字间距，下划线，删除线，上标，下标等

。

1.3.1. 字间距

通过 x 设置单个字符的坐标，两个字符坐标差值超过字符的宽度即可出现间距。

Plain

```
<g class="fragment">
  <text>
    <tspan x="0 20 40 60">text</tspan>
  </text>
</g>
```

1.3.2. 下划线、删除线

采用自定义 `<line>` 来绘制。

Plain

```
<g class="fragment">
  <text>
    <tspan x="0 20 40 60">text</tspan>
  </text>

  <g pointer-events="none">
    <!-- 下划线 -->
    <line x1="0" y1="4" x2="70" y2="4" stroke="#000"
stroke-width="2" />
    <!-- 删除线 -->
    <line x1="0" y1="-5" x2="70" y2="-5" stroke="#000"
stroke-width="2" />
  </g>
</g>
```

1.3.3. 上标、下标

不考虑上下标的嵌套场景，使用两个 `paragraphItem`

实现，并通过字号、坐标调整具体的上下标样式。

```
Plain
<g>
  <g class="fragment">
    <text>
      <tspan font-size="18">text</tspan>
    </text>
  </g>
  <g class="fragment">
    <text transform="translate(0 -10)">
      <tspan font-size="8">sup</tspan>
    </text>
  </g>
</g>
```

1.4. 效果属性

充分利用 svg

的绘图特性可以实现非常丰富的效果，目前课件支持的较少，后续待补充需求后完善

。

1.4.1. 阴影

阴影使用 svg 滤镜实现，相关的参数类似 CSS：

```
Plain
box-shadow: x y offset blur color;
```

svg 实现：

```
Plain
<defs>
```

```
  <filter id="filter_1">
    <!-- 阴影的偏移 -->
    <feOffset result="offOut_0" dx="7" dy="7"></feOffset>
    <!-- 阴影的模糊值 -->
    <feGaussianBlur stdDeviation="1"></feGaussianBlur>
    <!-- 基于转换矩阵对颜色进行变换 -->
    <feColorMatrix result="finalOut_0" type="matrix"
    values="0 0 0 0 0.2 0 0 0 0 0.2 0 0 0 0 0.2 0 0 0 0.4
0"
    color-interpolation-filters="sRGB"></feColorMatrix>
    <!--
允许同时应用滤镜效果而不是按顺序应用滤镜效果。利用result存储别的滤镜的
输出可以实现这一点，然后在一个 <feMergeNode> 子元素中访问它。 -->
    <feMerge>
      <feMergeNode in="finalOut_0"></feMergeNode>
      <feMergeNode in="SourceGraphic"></feMergeNode>
    </feMerge>
  </filter>
</defs>

<g filter="url(#filter_1)">
    <text>Text</text>
</g>
```

tips：阴影在横排、竖排场景下应该存在区别，暂不过多考虑竖排场景。金山竖排不支持阴影，zoho 不支持局部文字的阴影，仅支持整个文本元素的阴影。

1.5. 公式

在文字引擎的计算结果中，可以得知一块用于存放公式的区域。通过 katex.js 提供的方法，离屏创建公式Dom元素：

```
Plain
katex.render("c = \\pm\\sqrt{a^2 + b^2}", element, {
throwOnError: false
});
```

然后获取公式Dom的宽高，与文字引擎计算出的区域大小进行比较，比较后对公式Dom进行一定比例缩放，然后在svg中通过<foreignObject>嵌入html方式来实现公式的渲染：

```
Plain
<g>
<g class="fragment">
<foreignObject>
   <!--缩放公式Dom -->
<div style="transform:scale(0.5,0.5)">
    <!--katex.js生成的公式内容 -->
<span class="katex-display">
 ...
  </span>
</div>
</foreignObject>
</g>
</g>
```

tips：这是一种方案，当然我们也可以用其他方案来渲染公式，比如图片，不影响整体设计。

2. 编辑操作

2.1. 键盘操作

2.1.1. 键盘事件

针对键盘操作，使用一个隐藏的 input 组件进行劫持，每次点击文本时，需要使该

input 聚焦，从而可以监听DOM触发的输入类事件，整理如下：

事件名	参数	处理范围
keydown	{key,atlKey,ctrlKey,metaKey,shiftKey,isComposing...}	只处理isCompo的输入
compositionstart		不需要响应
beforeinput		不需要响应
compositionupdate		不需要响应
input	{data,isComposing,...}	只处理data!==
compositionend	{data,...}	处理data
copy、*cut*、*paste*		自定义复制粘贴
keyup		不需要响应

根据此表，可以得知针对 input 组件

，只需监听keydown、input、compositionend事件进行处理即可：

- keydown：所有按键都可触发，只处理其中特殊按键和组合键引起的且不是

  合成事件触发后的事件；

- input：可见字符输入触发，只处理data非空且不是合成事件触发后的输入（

  拼音输入过程不处理）；

- compositionend：合成事件结束事件，处理中文输入和键盘上键输入；

2.1.2. 输入处理

针对不同的键盘输入情况，进行如下的操作处理：

触发事件	输入	选中范围	处理方式
input	西文	选区	删除选中内容，收缩选区，在光标处插入西文（样式
		光标	在光标处插入西文（样式与光标样式保持一致）
compositionend	中文		处理方式同上
keydown	backspace	选区	• 跨段落：删除选中内容，合并选区设置的第落样式 • 同段落：删除选中内容
		光标	• 段落中间：删除光标前一个字符 • 含项目符号段落的段首：删除项目符号样式 • 普通段落的段首：合并光标前后段落
	delete	选区	同backspace
		光标	• 段落中间：删除光标后一个字符 • 段落末尾：合并光标前后的段落
	down	选区	选区收缩到选区终点
		光标	光标下移一位
	enter	选区	删除选中内容，分离当前段落，新段落与前一个段落
		光标	分离当前段落，新段落与前一个段落的段落样式相同
	left	选区	选区收缩到选区起点
		光标	光标前移一位
	right	选区	选区收缩到选区终点

		光标	光标后移一位
	tab	选区	• 跨段落：段落缩进级别+1 • 同段落：删除所选内容，插入一个\t
		光标	• 含项目符号段落的段首：段落缩进级别+1 • 其他：插入一个\t
	up	选区	选区收缩到选区起点，上移一位
		光标	光标上移一位
	ctrl+a		全选文字
	ctrl+b		设置文本样式（加粗）
	ctrl+c	选区	将所选内容的副本返回，保存在剪贴板中
		光标	无操作
	ctrl+i		设置文本样式（斜体）
	ctrl+u		设置文本样式（下划线）
	ctrl+v	选区	预处理：获取剪贴板数据，进行格式化处理，尽可能 删除选中内容，插入预处理后的剪贴板内容
		光标	插入预处理后的剪贴板内容
	ctrl+x	选区	删除选中内容，并将所删内容返回，保存在剪贴板中
		光标	无操作
	shift+enter	选区	删除选中内容，插入一个 break
		光标	• 含项目符号段落的段首：段落缩进级别+1 • 其他：插入一个\t

在某些情况下，集成文本组件的外部组件希望有自己独特的键盘响应行为，比如思维导图的 enter 键希望新建一个节点，而不是对文字进行换行。我们会提供一个 props 配置项，在组件初始化的时候禁用内部的按键操作行为。

2.2. 点击操作

在文本上进行点击行为，一般是设置选区。每次在视图组件上滑选或点击文本时，会根据点击位置拿到视图文字的

gcps，然后设置到选区编辑模型中，为接下来的操作做准备。

2.3. 其他操作

可能存在一些既不是键盘操作也不是点击操作的行为，比如侧边栏设置。编辑器提供一些 api 供外部调用，实现对应的功能。

3. 核心工具类

核心工具类是发生编辑行为时，我们修改 textBody 的方法，我们参考了 slate 的思路，设计了 transforms，interfaces 等模块去完成对应的操作。

3.1. 整体结构

3.2. 操作

3.2.1. GeneralTransform

内部只有一个transform函数，包含了9种基础操作，所有的修改数据的行为最终都会转化为这9种基础操作

Plain

```
transform:(editor,selection,op)=>{
  switch(op.type){
    case 'insert_node':{
      // 插入节点(paragraph或item)
    }
    case 'insert_text':{
      // 插入文本(在item的text中)
    }
    case 'merge_node':{
      // 合并节点
    }
    case 'move_node':{
      // 移动节点
    }
    case 'set_node':{
      // 设置节点属性
    }
    case 'set_selection':{
      // 设置选区
    }
    case 'split_node':{
      // 分离节点
    }
  }
  return selection
}
```

3.2.2. NodeTransform

Plain

```
NodeTransform = {
// 在指定位置插入节点
 insertNodes:(editor,nodes,options)=>void

 // 合并指定的节点与其前一兄弟节点，并在合并后删除空节点
 mergeNode:(editor,options?)=>void

 // 将指定位置的节点移动到新的位置
 moveNodes:(editor,options?)=>void

 // 删除指定位置的节点
 removeNodes:(editor,options?)=>void

 // 设置指定位置的节点的属性
 setNodes:(editor,options?)=>void

 // 分离指定位置的节点
 splitNodes:(editor,options?)=void

 // 删除指定位置的节点的指定属性
 unsetNodes:(editor,props,options?)=>void

 ...
}
```

3.2.3. SelectionTransform

Plain

```
SelectionTransform = {
// 收缩选区
 collapse:(editor,options?)=>void

 // 取消选区
```

```
 deselect:(editor)=>void

 // 设置选区
 select:(editor,target)=>void

 ...
}
```

3.2.4. TextTransform

Plain

```
TextTransform = {
 // 删除内容
 delete:(editor,options?)=>void

 // 在指定位置插入片段
 insertFragment:(editor,fragment,options?)=>void

 // 在指定位置插入文本
 insertText:(editor,text,options)=>void

...
}
```

3.3. 接口

围绕编辑器内部的一些概念，定义了一些接口，接口包含了相应概念的获取、比较、查找、修改等方法

3.3.1. NodeInterface

Plain

```
NodeInterface = {
// 节点的属性
extractProps:(node)=>NodeProps

 // 指定范围的片段
 fragment:(root,range)=>Descendant

 // 指定路径的节点
 get:(root,path)=>Node

 // 根据一组参数匹配节点
 matches:(node,props)=>boolean

 // 指定路径的节点的父亲节点
 parent:(root,path)=>Ancestor

 // 节点的纯文本内容
 string:(node)=>string

 ...
}
```

3.3.2. PathInterface

Plain

```
PathInterface = {
// 指定路径的祖先
 ancestors:(path,options?)=>Path[]

 // 寻找两个路径的共同祖先
 common:(path,another)=>Path

 // 比较两个路径的顺序
 compare:(path,another)=>-1|0|1
```

```
 // 检查一个路径是否存在前一个兄弟路径
 hasPrevious:(path)=>boolean

 // 指定路径的下一个兄弟路径
 next:(path)=>Path

 // 根据操作转化路径
 transform:(path,operation,options?)=>Path|null

 ...
}
```

3.3.3. RangeInterface

Plain

```
RangeInterface = {
// 按照在文档中顺序获取选区的起点和终点
edges:(range,options)=>[Point,Point]

// 检查一个选区是否包含另一个选区或路径或点
includes:(range,target)=>boolean

 // 获取两个选区相交的部分
 intersection:(range,another)=>Range|null

 // 根据操作转化选区
 transform:(range,op,options?)=>Range|null
 ...
}
```

3.3.4. TextInterface

Plain

```
TextInterface = {
// 判断两个文本节点是否相等
equals:(text,another,options)=>boolean

// 根据属性匹配文本节点
matches:(text,props)=>boolean

 // 获取一组range范围里的文本节点
 decorations:(node,decorations)=>Text[]

...
}
```

3.3.5. EditorInterface

Plain

```
EditorInterface = {
// 设置属性
addMark:(editor,key,value)=>void

// backspace删除
deleteBackward(editor,options?)=>void

// delete删除
deleteForward(editor,options?)=>void

// 删除文档片段
deleteFragemnet(editor,options?)=>void

// 指定位置的第一个节点
first:(editor,at)=>NodeEntry

// 插入段内换行
insertBreak:(editor)=>void
```

```
// 插入文档片段
insertFragment:(editor,fragment)=>void

// 插入节点
insertNode:(editor,node)=>void

// 插入文本
 insertText:(editor,text)=>void

// 获取将要被添加到当前选区的文本属性
marks:(editor)=>Omit<Text,'text'>|null

 // 根据脏路径规范化文档数据
normalize:(editor,options)=>

// 获取指定范围的纯文本内容
string:(editor,at,options)=>string

 // 执行完一个回调函数后进行normalize操作
 withoutNormalizing:(editor,fn)=>void
}
```

3.4. 工具

工具类函数：

- htmlToJSON：用于粘贴前的预处理，将html尽可能转化为textBody片段的数据
- gcpsToSelection：设置选区前，将选区渲染模型转换为选区编辑模型
- selectionToGcps：将选区编辑模型转换为选区渲染模型

...

3.5. create函数

createEditor函数创建一个编辑器对象返回，该对象内部保存所需修改数据的引用和一些其他辅助数据模型，并在其上挂载了可供调用的功能函数。

Plain

```
createEditor = (data)=>{
  const editor = {
    data:data,
    selection:null,
    marks:null,

    onChange:()=>{}
    apply:(op)=>{}

          addMark:(key,value)=>{}
          deleteBackward:(unit)=>{}
          deleteForward:(unit)=>{}
          deleteFragment:(direction?)=>{}
          getFragment:()=>{}
          insertBreak:()=>{}
          insetFragement:(fragment)=>{}
          insertNode:(node)=>{}
          insertText:(string)=>{}
          normalizeNode:(entry)=>{}
          removeMark:(key:string)=>{}
    ...
  }

  return editor
```

```
}
```

遗留问题

a. 在当前的设计方案中，撤销重做的时候，系统不会对光标撤销重做，跟用户预期可能不符。

b. 在光标位置双击的时候，通常会选中一个分词，在当前的设计方案中，英文文档我们能够借助前后空格选中一个单词，中文却没有办法选中一个词语，可能需要 c++ 在排版模型里面带上分词的信息，这块暂时没有。