链滴

Vditor 实现 Markdown 所见即所得

作者：88250

原文链接：https://ld246.com/article/1577370404903

来源网站：链滴

阶段性目标

大概在去年这个时候萌生了开发 Vditor 的念头，两个月后 Vditor 第一版完成。第一版基于 textarea 开发，该版本的主要目标是替换在用的第三方编辑器，实现 B3log 各产品线上编辑器的统一。

Vditor 的定位是“下一代的 Markdown 编辑器，为未来而构建”。第一版离这个目标相差还远，但过这大半年的迭代我们离目标已经越来越近了：

- 由基于 textarea 改为基于 contenteditable
- 实现 Markdown 引擎 Lute，并由 markdown-it 切换为 Lute
- 图表渲染、多媒体播放、语音阅读等功能
- 所见即所得模式

所见即所得

所见即所得（What you see is what you get，缩写即 WYSIWYG）在编辑器领域指的是看到啥就是，比如我想加粗某个字，那“加粗”只是个操作，操作结果就是得到加粗后的文本，而加粗的操作（记符）是不会出现在文本中的。

目前看来，大部分人习惯所见即所得的编辑方式，因为这种方式非常直观。但在程序员圈子里却恰恰反，我们更喜欢标记类排版编辑方式，其中最为流行的是 Markdown。显然这类标记语言不是所见所得的，但从程序员的角度理解的话，其实这才是真正的所见即所得，就像《黑客帝国》中的代码绿。

Markdown 实时渲染

在实现“Markdown 所见即所得”的方向上，Typora 作为先行者广受好评，但它到目前为止并不开。和它类似的应用并开源的是 Mark Text，不过它的编辑器似乎还不能方便地抽取出来应用于其他 Wb 应用。

通用的所见即所得 Markdown 编辑器也有项目，比如 HyperMD。它的编辑器内核用了 CodeMirro，依赖项较多。

除此之外，从标准 Markdown 规范（CommonMark/GFM）的支持上来看，它们都有或多或少的问。不遵循规范的话一旦渲染出现偏差，会给用户带来较大困扰，编辑发布后的渲染结果和编辑时的渲结果不一致，这个问题在内容发布多平台时尤为严重。

这三款编辑器设计的出发点是以 Markdown 为核心，所达到的效果是 Markdown 实时渲染，并且留标记符，光标如果在标记内部则展开整个标记节点。这个设计可以让 Markdown 用户更专注于内创作，减少标记符带来的视觉干扰，同时也基本兼顾 Markdown 的优势：排版不用记各种不同编辑的快捷键。但这样设计的不足之处是对非 Markdown 用户不太友好，展开的标记符会让普通用户有不知所措，并且还是会产生视觉干扰。

选择方向

综上，Vditor 的演进路线有两个方向：

1. 实现保留标记符的 Markdown 实时渲染
2. 所见即所得编辑器，支持 Markdown

起初我们主要在方向 1 上做出了努力，基本实现了支持 GFM 的 Markdown 实时渲染，但最后还是为产品思路转变而放弃了，因为我们觉得方向 2 更加普适。

方向 2 是实现一款所见即所得的富文本编辑器，并对 Markdown 提供支持。这样一来，Vditor 的受用户会变宽很多，同时 Markdown 用户也可以继续平滑地使用 Markdown 语法进行排版。知乎的辑器是支持简单的、类似 Markdown 语法排版的，不过稍微复杂一点的排版标记支持就不太好了。

另外，在表格的处理上实时渲染方案是不一致的。表格的 Markdown 表示稍微复杂，所以现有的实渲染方案也都是做成了所见即所得（弹框编辑）的，这导致整体设计不太一致。

这些因素促使我们决定将 Vditor 往所见即所得的方向演进，同时保留传统的 Markdown 分屏预览模。从此 Vditor 的定位也就变成了：“一款所见即所得的编辑器，支持 Markdown”。

技术实现

虽然前面提到的路线 1 已经放弃了，但还是分享一下技术实现。这个方案主要构成如下：

- 基于 AST 渲染带各种标记语义的 DOM，标记语义主要通过 span 属性实现
- 将 DOM 通过 text() 还原为 Markdown 原文
- 通过源码映射将 Markdown 原文和 AST 做关联，通过计算字符偏移量来实现光标位置重置

这个方案中最难的部分是实现光标重置，因为 CM 参考实现（基本所有实现了 CM 规范的解析器都用的参考实现的算法）只能做到块级元素的源码位置记录，做不到行级元素。要做到行级元素的源码置记录只能通过词法分析时记录每个 token 的字符位置来实现，但由于这样做会降低性能，所以大部解析器都没有实现源码映射，毕竟 Markdown 源码映射的应用场景太窄了。

Lute 引擎的源码映射是通过条件编译实现的，服务端渲染时编译的版本是不带源码映射的，Vditor 用的 JS 导出版本带了源码映射，并提供了 Vditor DOM 渲染器。

路线 2 的实现方案就简单很多，因为不走源码映射的思路：

- 基于 AST 渲染 Vditor DOM
- 解析 DOM 还原为 Markdown 原文
- 在 DOM 中记录“插入字符”实现光标位置重置

“插入字符”目前用的是 wbr 标签，解析时内部映射为 Caret \u2038 ，在解析过程中带着这个字符，渲染 DOM 时在替换为 wbr 。使用标签的原因是通过 DOM API 处理起来较为方便。

这个方案的弊端显而易见，输入 wbr 标签或者 Caret 字符将影响解析导致错误，不过实际使用时不可能出现，所以暂时就这样吧。

计划未来

功能实现好后就是性能优化了。大致的技术方案是局部解析渲染，这一步较为容易的是先做到根上的一层块级，一般来说一个块子树不会太大，所以估计这样做性能应该足够好了。如果追求极致的话可考虑将这个算法在块级子树上递归，通过控制深度来进行较细粒度的局部解析渲染。

在源码模式上可以基于 AST 做语法高亮，因为 Lute 实现了完整的 Markdown AST，“完整的”意就是所有源码都会被解析为节点，包括了标记符。这样就比较容易在渲染时带上 CSS 类来实现语法亮了。

更远一些的构想：

- 实现块风格编辑器，支持块复用
- 支持其他的标记语法