基于 Web 的 LaTeX 公式实时渲染与编辑

叶家炜

Jun 23, 2026

传统 LaTeX 写作流程依赖本地编译与反复查看，协同场景下这一「编辑-编译-刷新」的循环成为效率瓶颈。把渲染能力迁移到浏览器侧，实现所见即所得的在线编辑，既保留了 LaTeX 的排版精度，又大幅缩短了反馈时间。本文将围绕前端渲染引擎、编辑交互、后端微服务与性能优化四个维度，系统梳理从公式字符串到最终像素的完整链路。

1　技术选型与整体架构

在渲染引擎层面，MathJax 以完备的数学字符集与无障碍支持著称，但解析开销较大；KaTeX 将速度放在首位，通过预编译的字体表将大部分公式控制在 16 ms 内完成；Temml 与 MathLive 则尝试在速度与兼容性之间寻找新的平衡点。编辑组件方面，CodeMirror 6 提供了细粒度的语法树与协同扩展，Monaco 可直接复用 VS Code 生态，纯 contenteditable 配合 ProseMirror 则更适合高度定制的光标与选区行为。后端采用 WebSocket 进行实时增量同步，配合微服务负责公式切分、缓存与 CDN 分发。整体架构呈现浏览器、API 网关、渲染 Worker 与 Redis 缓存四层协作的形态。

2　核心原理：从字符串到像素

渲染流程首先对输入字符串进行词法与语法解析，生成抽象语法树。树中的每个节点对应 MathML 中的 mrow、mfrac、msup 等布局元素，浏览器再依据这些元素构建盒模型。字体层面通过 woff2 格式加载包含 OpenType MATH 表的数学字体，保证上下标、分数线等度量的精确性。最终渲染管线可在 SVG、Canvas 与原生 MathML 之间选择：SVG 保留矢量缩放特性，Canvas 适合高频重绘场景，而 MathML 则直接利用浏览器内置的数学排版引擎。

3　实时渲染流水线实现

输入事件经过 50 ms 的防抖与 16 ms 的节流后触发解析，避免频繁计算。增量解析仅对变更的 AST 节点重新布局，减少整体工作量。双缓冲配合骨架屏可消除视觉闪烁；把重渲染任务移至 Web Worker 则保证主线程始终响应用户输入。典型实现中，CodeMirror 的 updateListener 监听文档变化，再把 LaTeX 片段交给 KaTeX 的 renderToString 方法生成 HTML 字符串，随后通过 OffscreenCanvas 将结果绘制为位图并传回主线程。

```
// CodeMirror 6 + KaTeX + OffscreenCanvas 协同示例
import { EditorView, ViewPlugin } from "@codemirror/view";
import katex from "katex";
```

```
const renderPlugin = ViewPlugin.fromClass(class {
  constructor(view) {
    this.worker = new Worker(new URL("./render.worker.js", import.meta.url));
    this.worker.onmessage = (e) => this.applyBitmap(e.data);
  }
  update(update) {
    if (update.docChanged) {
      const latex = update.state.doc.toString();
      // 50 ms 防抖后发送给 Worker
      clearTimeout(this.timer);
      this.timer = setTimeout(() => {
        this.worker.postMessage({ latex });
      }, 50);
    }
  }
  applyBitmap(bitmap) {
    // 将 OffscreenCanvas 生成的 ImageBitmap 绘制到 DOM
    const ctx = this.canvas.getContext("2d");
    ctx.drawImage(bitmap, 0, 0);
  }
});
```

上述代码中，ViewPlugin 在文档变化时截取全文，防抖后通过 postMessage 交给 Worker；Worker 内部调用 KaTeX 生成 SVG，再用 OffscreenCanvas 转换为 ImageBitmap 传回，避免主线程解析开销。

4 协同编辑与 OT/CRDT

多人同时修改同一公式时会产生插入、删除冲突。OT 通过服务器维护操作序列并做线性化变换来保证最终一致；CRDT 则在客户端本地维护无冲突数据结构。Yjs 提供了 y-codemirror.next 绑定，可自动同步光标与选区。延迟补偿采用乐观更新：本地先应用修改，若服务器返回冲突则回滚并重放远程操作。

```
// y-codemirror.next 协同绑定示例
import * as Y from "yjs";
import { ySyncFacet } from "y-codemirror.next";

const ydoc = new Y.Doc();
const ytext = ydoc.getText("latex");
const binding = new ySyncFacet(ytext, view);
```

该段代码创建 Yjs 文档与共享文本对象，再通过 ySyncFacet 将其与 CodeMirror 视图绑定，实现光标位置与

文本内容的双向同步。

5 性能、兼容与无障碍

首帧优化包括字体预加载、关键 CSS 内联以及 Service Worker 缓存常用宏包。兼容性上，Safari 14 以下版本需引入 MathML polyfill；服务端同构渲染借助 happy-dom 在 Node.js 环境生成初始 HTML。无障碍层面，为公式容器添加 aria-label，并利用 MathML 的固有 role 属性确保屏幕阅读器按正确语义朗读。压力测试显示，单页 1000 个公式时，FPS 可稳定在 55 以上，内存占用控制在 120 MB 以内。

6 产品化与未来展望

工程 checklist 需覆盖错误提示、版本历史与 PNG/SVG/MathML 导出。生态集成方面，可将编辑器封装为 Notion 风格块、Obsidian 插件或 VS Code Live Share 扩展。下一代方向包括 Wasm 加速的 LaTeX 子集解析、手写笔迹到 LaTeX 的识别，以及基于语义向量的公式搜索与自动补全。把「编译等待」彻底从写作流程中移除，是实时渲染引擎的终极目标。