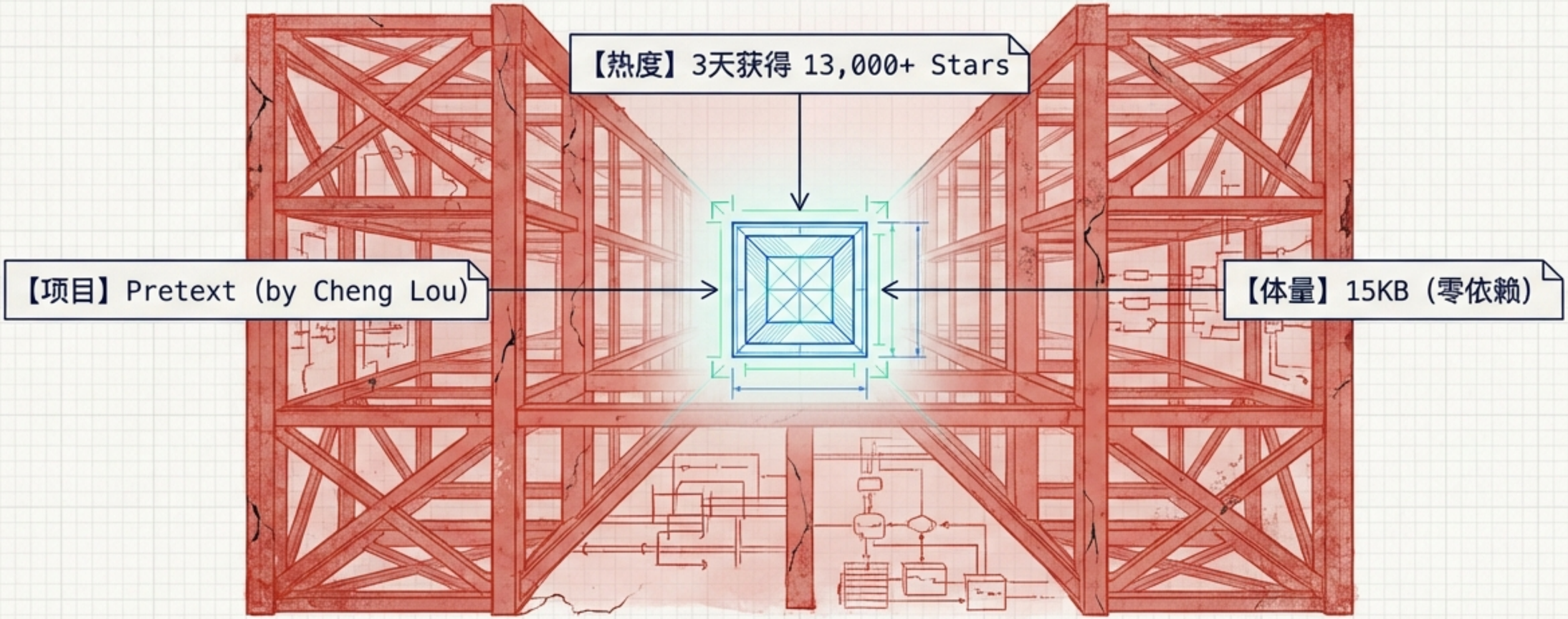
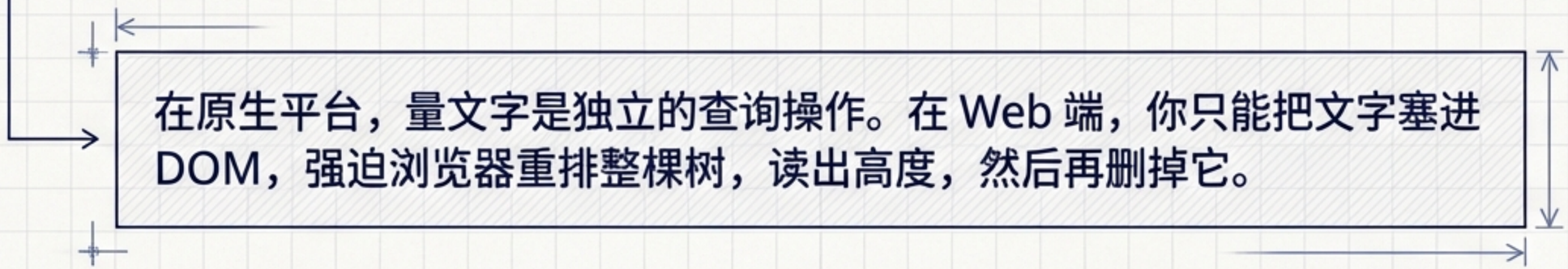
Web 做了三十年都没做到的事，被一个 15KB 的库解决了



仅仅为了解决一个看似简单的问题：告诉你一段文字渲染出来有多高，而不触碰 DOM。

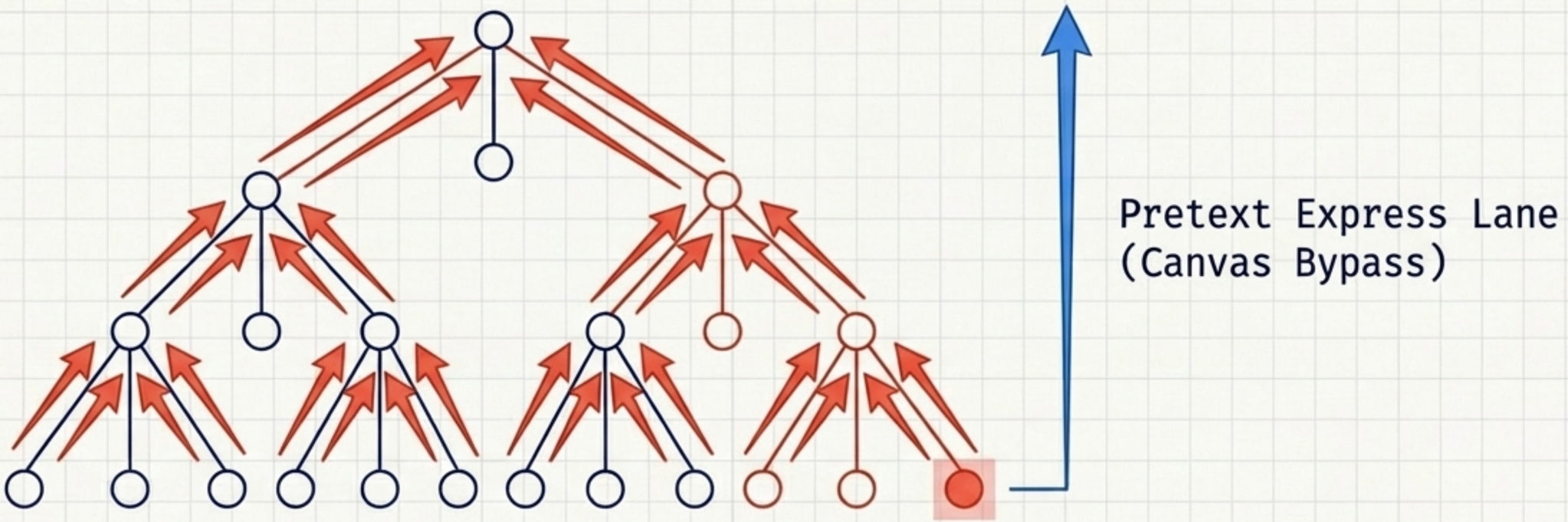
测量文本高度：原生平台的基操，Web 端的性能噩梦

平台	专用 API?	副作用（Reflow）	性能成本
iOS	sizeWithAttributes	无（一行代码）	极低
Android	StaticLayout	无	极低
Flutter / Compose	TextPainter / TextMeasurer	无	极低
Web	缺失	全局耦合（销毁并重建）	灾难级



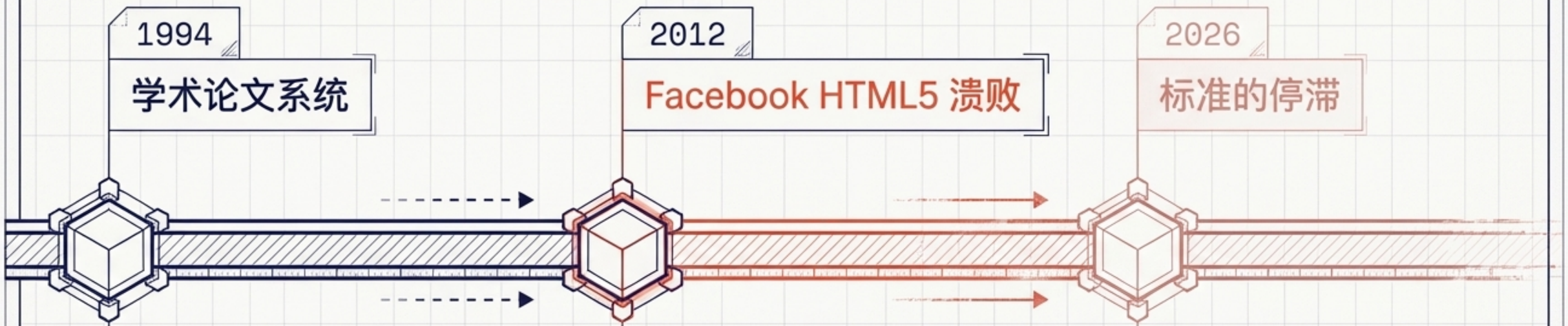
布局回流（Layout Reflow）：DOM 架构的全局交通堵塞

Global Coupling / 全局耦合



CSS 布局是全局粘合的。一次看似微小的测量，让整个页面的渲染树重新计算。

1994 年的合理设计，如何演变成跨越 30 年的技术债



1994

学术论文系统

- CSS 诞生。Web 被设计为纯文档交换。
- 独立查询布局结果的需求不存在，接口未被设计。

2012

Facebook HTML5 溃败

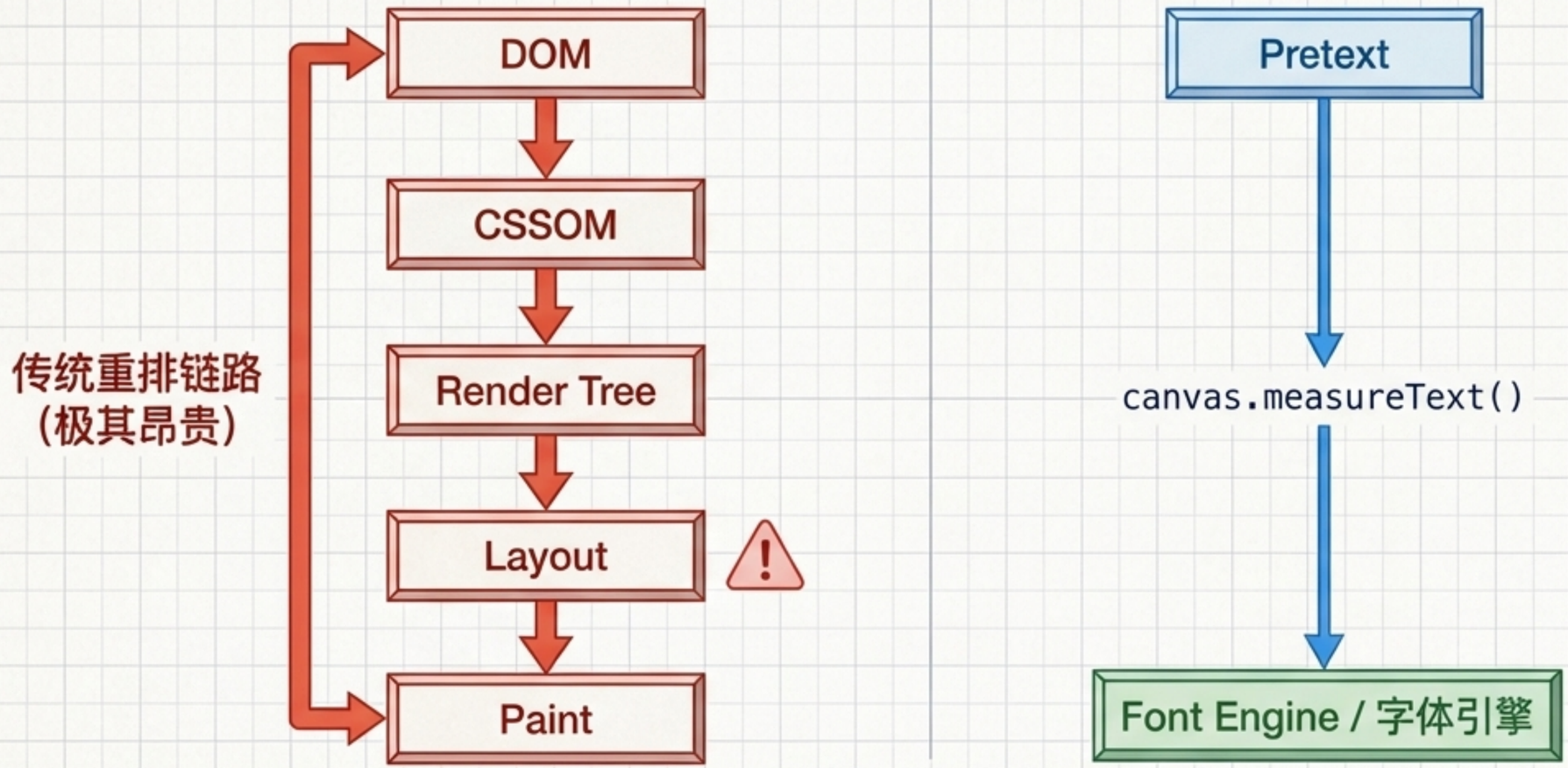
- DOM 全局 Reflow 扛不住 News Feed 性能要求。
- 被迫花 9 个月重写原生版。催生了绕过 CSS 的 React Native Yoga 引擎。

2026

标准的停滞

- W3C 提出 CSS Houdini API 试图还债。
- 至今没有主流浏览器实现。技术债变成天文数字。

Pretext 的优雅后门：绕过渲染树，直达底层字体引擎



Canvas 的文字渲染与 DOM 共用同一个底层字体引擎，但它完全独立于布局树之外。
通过 Canvas 量文字，永远不会触发 Reflow。

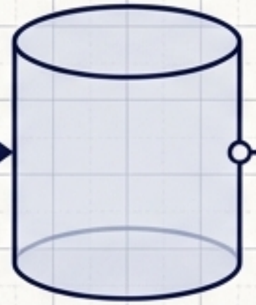
性能跃升的秘密：将昂贵的预检与廉价的计算物理分离

冷路径：一次性预检

```
prepare(text, font)
```

使用 Intl.Segmenter 处理语言感知分词（中日韩、emoji），通过 Canvas 测量每个片段并存入缓存。

500 段文字 ≈ 19ms



宽度数据缓存

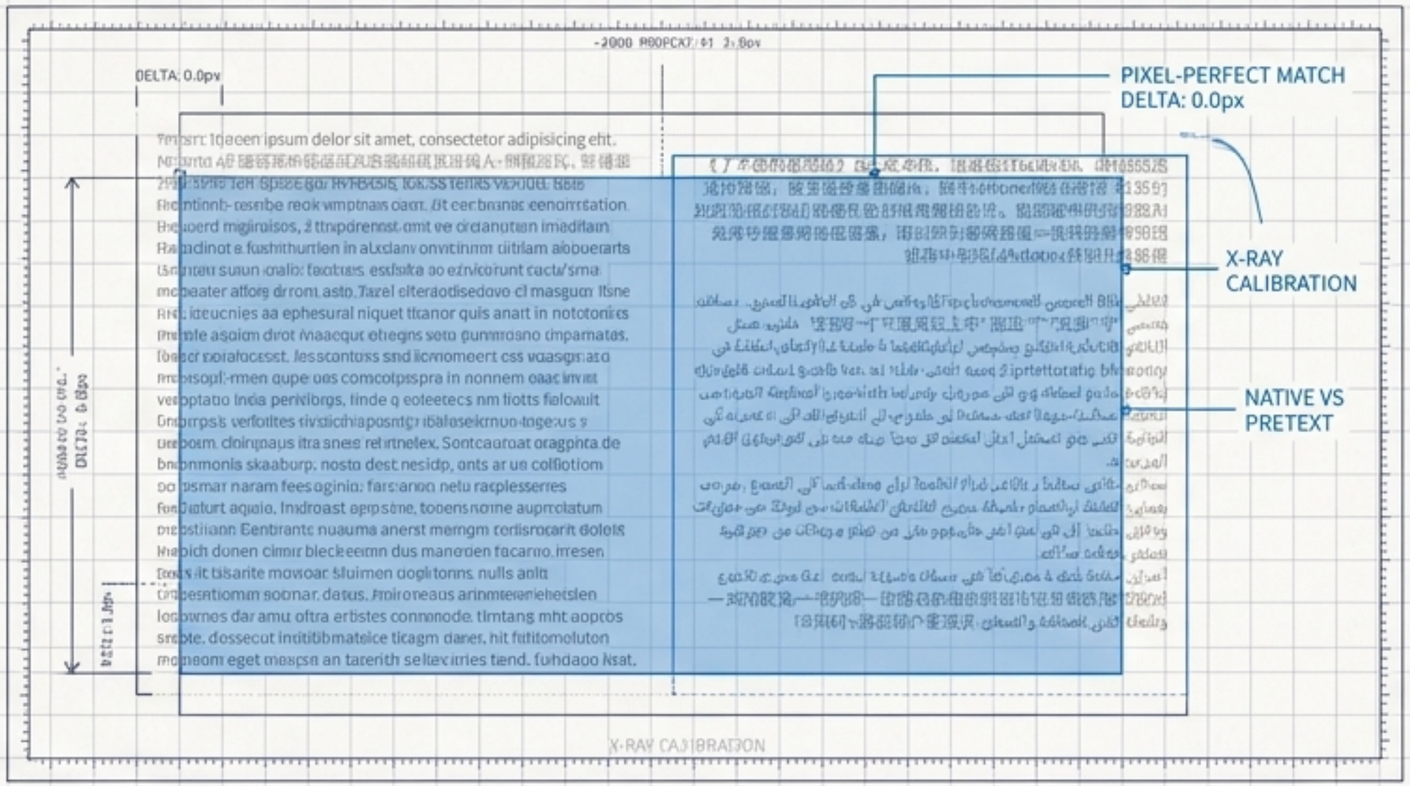
热路径：极速计算

```
layout(prepared, maxWidth,
       lineHeight)
```

纯算术运算。基于缓存数据模拟浏览器换行算法，直接返回高度数据，完全不碰 DOM。

500 段文字 ≈ 0.09ms

放弃 DOM 不等于放弃精度：像素级完美匹配



极限压力测试

将《了不起的盖茨比》全文本在 Chrome、Safari、Firefox 上的排版进行逐像素比对与算法迭代。

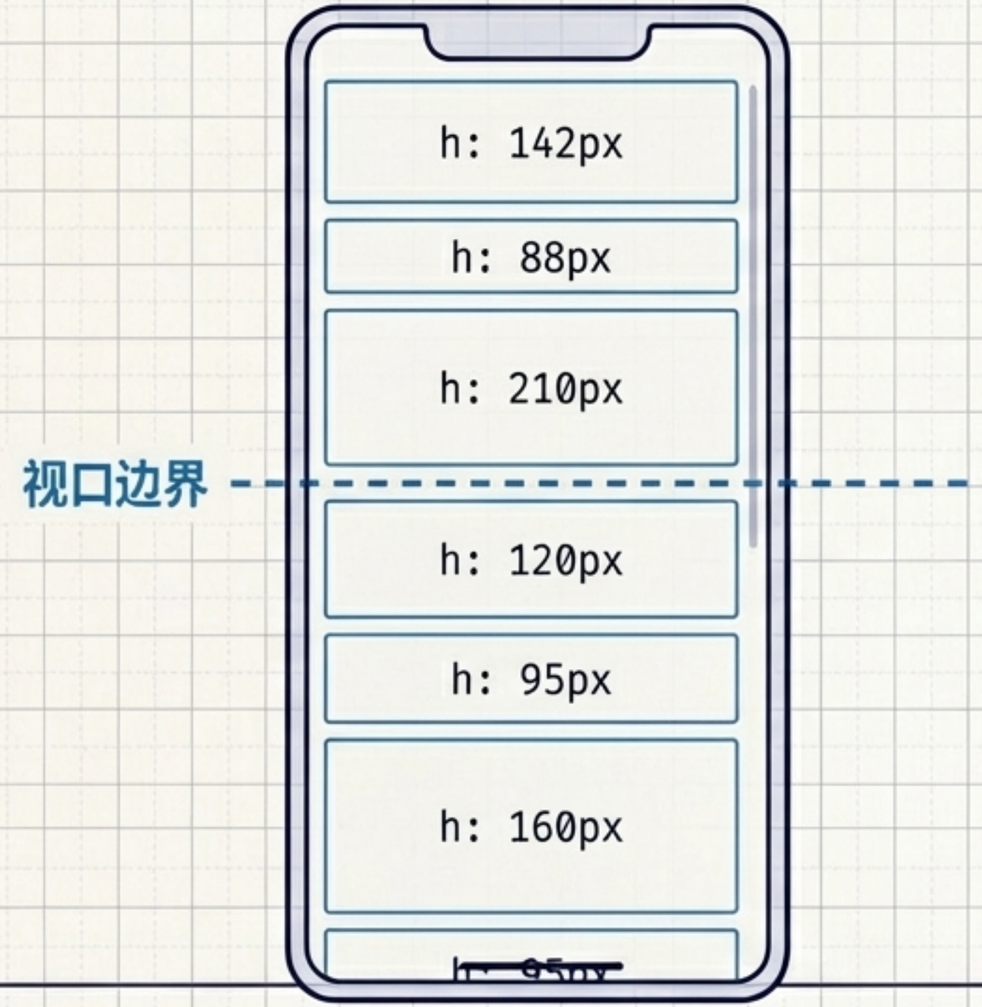
全语系完美支持

泰语 中文 韩语 日语 阿拉伯语

完美兼容双向文本（Bi-Di）与复杂软连字符。

在前端生态中，用户态的数学模拟完全可以达到甚至替代底层引擎的布局效果。

突破一：无需盲猜高度的真正虚拟列表（Virtual Lists）



传统现状

传统方案只能靠“猜”高度，或者先渲染再测量，导致滚动时出现肉眼可见的跳动和布局偏移（Layout Shift）。

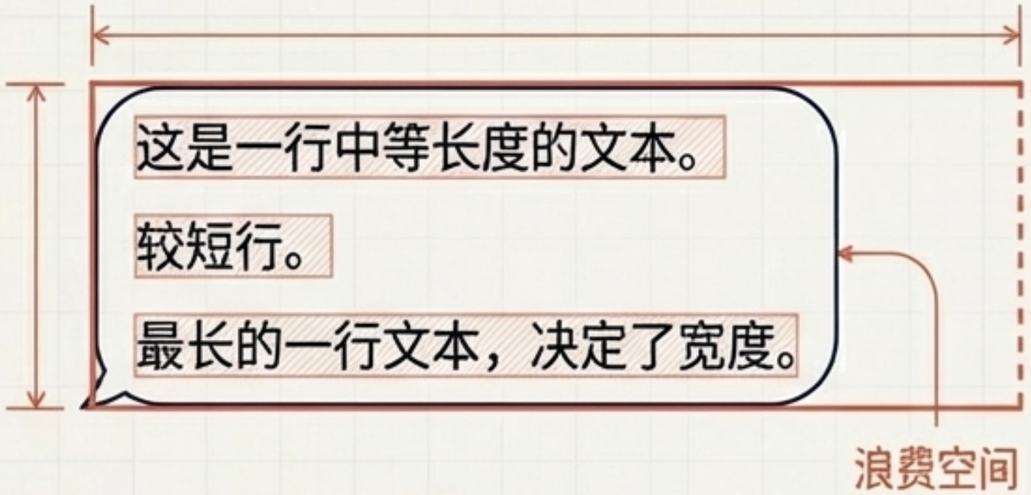
Pretext 现实

元素进入视口之前，即可通过预计算得知精确到像素的高度。实现零布局偏移，零 DOM 污染。

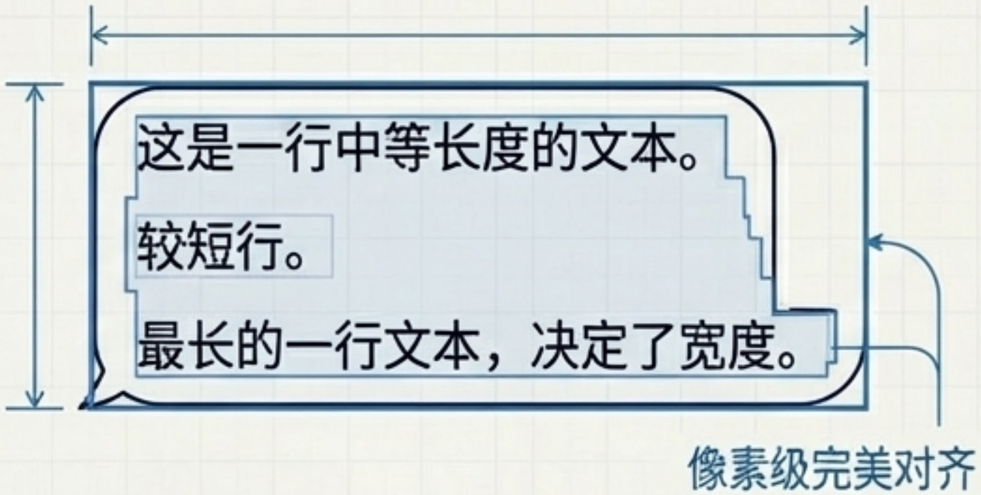
10 万个不同高度文本项瀑布流排版，稳定运行在 120fps。

突破二：打破矩形束缚的“收缩包裹”（Shrink-Wrap）

传统 CSS：容器比实际需要更宽

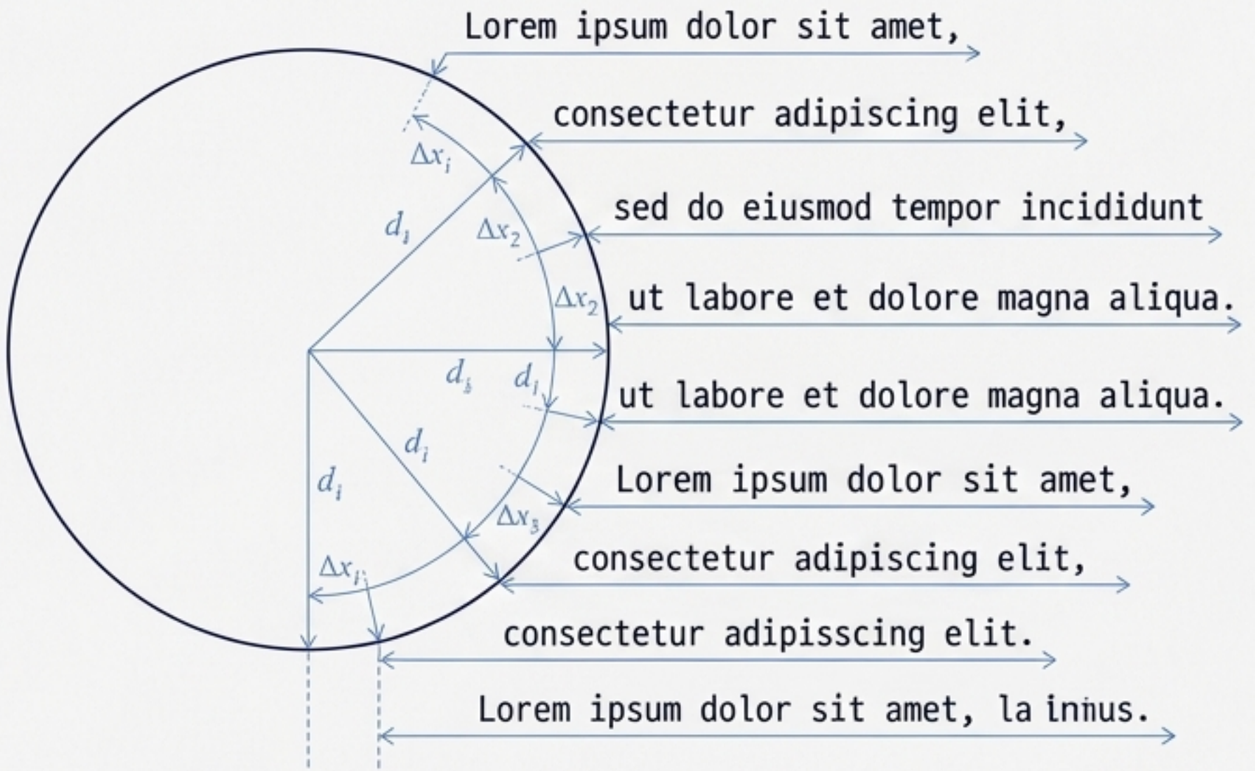


Pretext：像素级完美收缩包裹



CSS 无法表达「让容器宽度刚好等于最宽那行的宽度」。Pretext 的 `walkLineRanges()` API 通过二分搜索找出了数学上最紧凑的宽度——这是一个 Web 原生缺失的布局原语。

突破三：在用户态实现绕开障碍物的复杂排版



核心机制

通过 layoutNextLine() API，允许在计算时为每一行实时分配不同的最大宽度。

社区应用案例

- pretext-explosive：将每个字符转为独立物理粒子，点击产生爆炸交互。
- illustrated-manuscript：杂志级别的排版，文字能够动态环绕动画轨迹排布。

技术启示

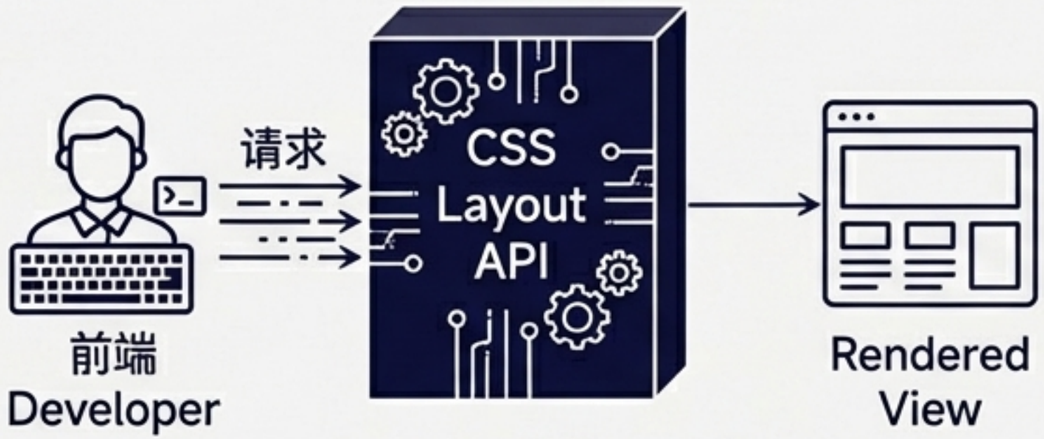
杂志式的复杂环绕排版，曾经因为算法复杂度而被原生 CSS 放弃；现在只需 15KB，即可在浏览器用户态轻松实现。

抽象的对决：全局黑盒机制 vs 独立可观测模型

维度	传统 Web CSS 布局	Pretext 架构模型
系统集成	全局耦合 （修改一处，影响全身）	隔离运行 （旁路计算，无副作用）
中间状态	隐藏/黑盒 （开发者无法介入）	可观测/可查询 （完全暴露给开发者）
首要服务对象	人类读者 （只看最终渲染结果）	算法与 AI （需要数据流与反馈循环）

隐藏中间状态：不仅是前端的死穴，更是 AI 架构的陷阱

前端（人类开发者隐喻）



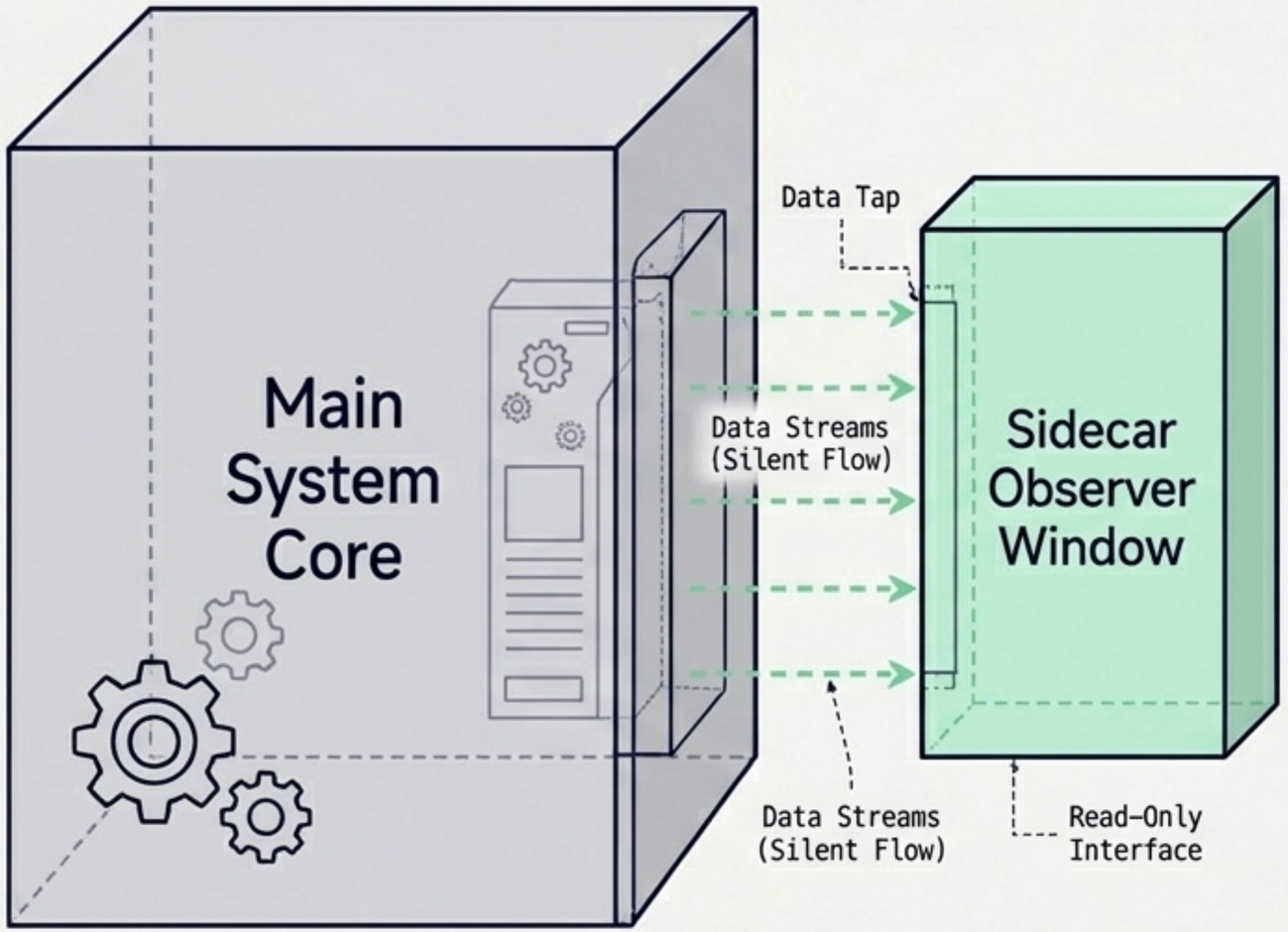
1994年的选择：隐藏布局中间状态，对人类而言是是干净优雅的抽象。

AI 架构（AI Agent 隐喻）

如今的陷阱：隐藏底层中间状态，彻底阻断了 AI 进行「尝试-反馈-修正」的闭环。

好的抽象，赋予你在不同层级工作的自由；坏的抽象，将所有层级死死粘合。我们们目前为 AI Agent 设计 API 时，正在重蹈 CSS 30年前的覆辙。—— via yage.ai

Sidecar Observer：为 AI 时代设计的可观测架构



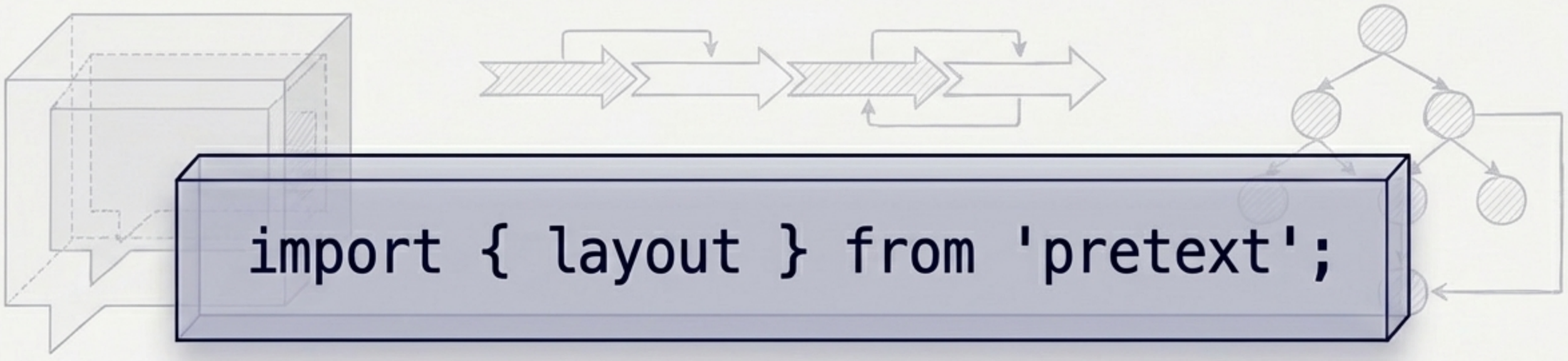
核心哲学

AI 及其衍生算法需要持续观测系统的中间状态（Intermediate States）。绝不能把计算过程封死在黑盒中。

Pretext 的终极启示

不要试图直接替换庞大且僵化的现有系统。在系统旁边开一个独立的观察窗口（旁路模式），让算法可以自由获取数据而无需触发全局运转。

结语：用 15KB 重塑 Web 底层基建



- 这不仅是一次技术的炫技，更是对"不可能"的 UI 模式的彻底解放。
- 一个人，借助 AI 辅助编码，在几周内完成了一个零依赖的基础设施奇迹。
- 有时候最重要的基础设施并不是最复杂的，而是精准补齐了那层早该存在、却一直缺席的抽象。

github.com/chenglou/pretext

yage.ai/web-layout-tradeoff.html