《实验二 文本编辑器背后的数据结构的研究学习》

实验报告

工作分工	姓名	学号
三个软件的概览		
深入探究 Scintilla 的 Gap Buffer 结构		

2025 年 11 月 20 日

一、三个软件的数据结构概览

1.1 Scintilla – Gap Buffer

通过阅读/src/SplitVector.h 文件，我们发现 Scintilla 采用了 Gap Buffer 数据结构，在内存中维护一个连续的缓冲区，并在缓冲区中预留一个 gap，这个 gap 位于当前编辑位置附近。当用户进行插入操作时，数据被插入到这个间隙中。Gap Buffer 的特性使得在光标附近操作时性能极佳，时间复杂度为$O(1)$，且连续内存访问，对缓存友好。

1.2 VS Code – Piece Table

通过阅读/src/vs/editor/common/model/pieceTreeTextBuffer 中的 4 个文件，我们了解到 VS Code 通过 Piece Table 储存文本。其核心思想是是将文本分解为多个不可变的片段，通过红黑树组织管理，实现高效的文本编辑和版本控制。

VS Code 的 Piece Table 由两大组件构成：字符串缓冲区数组和红黑树。缓冲区分为原始缓冲区（只读）和更改缓冲区（可写），所有编辑操作都在更改缓冲区中进行。红黑树的每个节点包含一个 Piece 对象，精确记录片段在缓冲区中的位置、长度和换行符数量。这种设计保证了原始文本的不可变性，所有修改都通过创建新片段实现。

当插入文本时，系统定位到对应树节点。如果插入点在节点中间，原节点被分割为新节点。新文本存入更改缓冲区并创建对应片段插入红黑树。删除操作通过调整树结构逻辑移除内容，不立即释放缓冲区空间。系统对批量编辑进行专门优化，自动合并操作以减少树重组开销。

Piece Table 避免了频繁内存移动，适合大规模文件编辑。

1.3 xi editor – Rope

xi editor 中的 Rope 实现基于 B 树变体结构，每个节点可以包含 4-8 个子节点，在树的高度和节点容量之间取得平衡。核心的 Node <N>结构采用泛型设计，通过 NodeInfo trait 定义节点行为，使得 Rope 可以灵活适应不同度量标准。

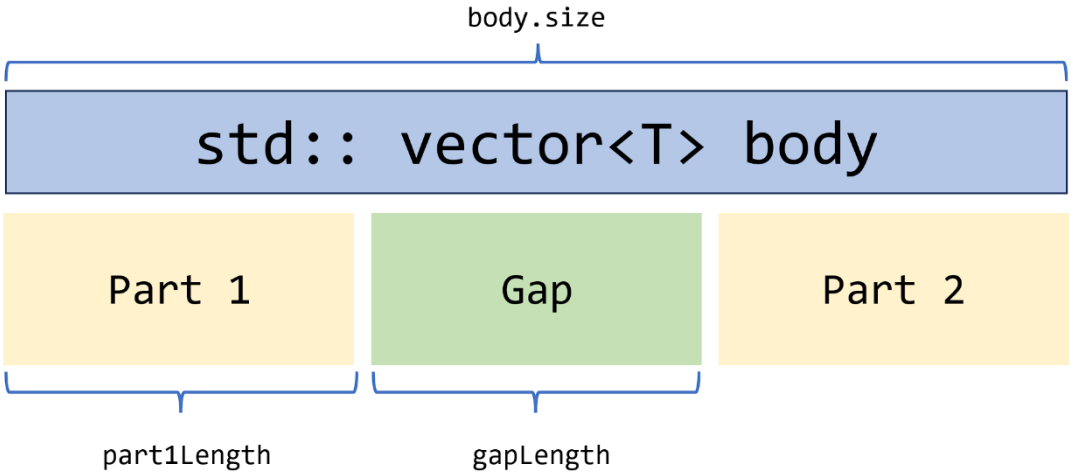
每个叶子节点存储实际的文本片段，内部节点则累积子树的元数据信息。这种设计使得 Rope 具有持久化特性——编辑操作仅修改受影响的部分节点，其他节点可以安全共享，极大优化了内存使用。

Rope 分块存储策略避免了单一大内存块的分配问题。通过维护子树长度和

行数等元数据，行列定位、区间提取等操作都能高效完成。

二、对 Scintilla 的 Gap Buffer 结构深入探究

2.1 数据结构逻辑图



Scintilla 使用的 Gap Buffer 结构如图所示，其使用了一个模板，在一个连续的内存块（即 std:: vector<T> body）中维护一个“间隙”（Gap）。文本内容被分为两部分，分别位于间隙的两侧。所有的编辑操作都发生在间隙处。

2.2 插入功能

假设要在逻辑位置 pos 插入字符串，需要进行以下操作：

1. 移动间隙 (GapTo): GapTo 操作是一个 protected 的操作，因此它不可以被外部实例调用，确保了安全性，达到了模块化封装的目的。这个函数检查当前间隙的位置 (part1Length) 是否就是目标插入点 pos。如果不是，它会通过内存移动 (std::move 或 std::move_backward) 将 pos 点之后的数据块移动到间隙之后，从而把间隙“挪”到 pos 位置。这个操作可以被理解为 C 语言中的 memcpy，只不过是一种更简洁更方便的 C++操作。

2. 检查空间 (RoomFor): RoomFor 同样是 protected 操作。它检查间隙的长度 (gapLength) 是否足够容纳要插入的内容。如果不够，会触发 ReAllocate，扩大底层 vector 的容量，从而增大间隙。

3. 插入数据: 以 Insert 开头的操作都是插入数据的操作，这些操作是 public 的。再间隙就位且空间足够后，直接将新数据拷贝到间隙的起始位置。

4. 更新数据: 最后，更新描述符 part1Length 增加插入的长度，gapLength 减

少相应的长度，lengthBody 也增加。

需要说明的是，当一次插入操作结束以后，根据 Gap Buffer 的特性，同一时刻只会同时存在一个 gap，而原先的 gap 既没有被删除、也没有被注释掉，而是被“移动”到了新的需要放置的位置。这就是 GapTo 存在的意义。

2.3 删除功能

假设要删除从逻辑位置 pos 开始的 len 个字符。需要进行以下操作：

1. 移动间隙 (GapTo): 与插入类似，首先调用 GapTo(pos) 将间隙移动到要删除内容的起始位置。

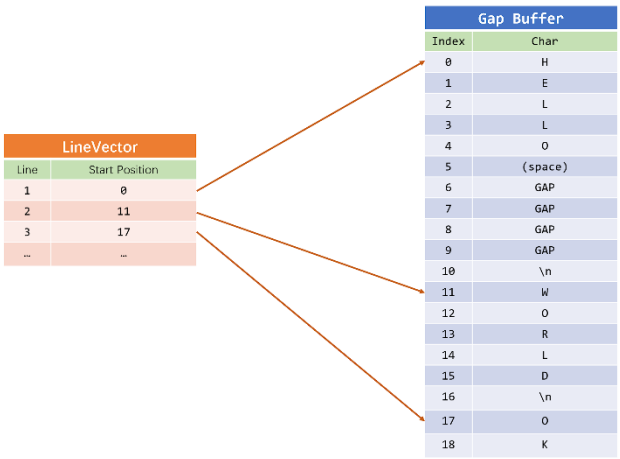
2. “删除”数据: 删除操作非常高效。它不进行任何内存擦除或移动，仅仅是增大间隙的长度，使其“吞噬”掉紧随其后的 len 个字符。

3. 更新数据: 更新描述符 gapLength 增加 len，lengthBody 减少 len。part1Length 保持不变。

2.4 按行号定位

Gap Buffer 数据结构定义的源码文件（即 /src/splitVector.h）中并没有记载按行号定位功能。这是因为按行号定位功能是比数据结构定义更高级的操作。在 /src/CellBuffer.cxx 中，实现了此功能。

具体而言，CellBuffer.cxx 中定义了一个名叫 LineVector 的辅助数据结构，其记录了每行开始的位置索引。这种索引表的结构用额外的空间换来了$O(1)$的时间复杂度，使得查找变得更加快捷。



2.5 Undo / Redo 功能

Undo 的具体过程如下：

1. 调用 CanUndo() 检查 currentAction > 0，确保有操作可撤销。

2. 获取步骤: 调用 GetUndoStep() 获取上一个操作（索引为 currentAction - 1）的详细信息。这包括操作类型（插入/删除）、位置、长度。被删除或插入的文本内容从 scraps 栈中获取（通过 scraps->CurrentText() - len）。

3. 执行逆操作: 编辑器（Document 类，虽然此处代码未显示调用，但逻辑如此）根据获取的信息执行逆操作：如果记录的是 Insert (插入)，则执行 Delete (删除)。如果记录的是 Remove (删除)，则执行 Insert (插入)（将 scraps 中的文本填回去）。

4. 更新状态: 调用 CompletedUndoStep()。scraps 指针后退（MoveBack），因为这部分文本现在被视为“未来”的 Redo 数据。currentAction 减 1，指针向左移动。

Redo 的具体过程如下：

1. 检查: 调用 CanRedo() 检查 currentAction < actions.SSize()，确保有操作可重做。

2. 获取步骤: 调用 GetRedoStep() 获取当前操作（索引为 currentAction）的详细信息。同样从 scraps 中获取对应的文本内容。

3. 执行正向操作: 编辑器执行与记录完全一致的操作：如果记录的是 Insert，则执行 Insert。如果记录的是 Remove，则执行 Remove。

4. 更新状态: 调用 CompletedRedoStep()。scraps 指针前进（MoveForward），跳过刚刚重做的文本。currentAction 加 1，指针向右移动。

2.7 Scintilla 使用 Gap Buffer 的优缺点评析

Gap Buffer 具有很多优点，首先，其局部编辑性能很高。在日常使用场景中，我们倾向于在同一个地方插入完所有文本之后再移动光标到另一个地方。而在 Gap Buffer 中，只要把 Gap 移动到合适位置，就可以连续写入或删除，这些操作时间复杂度为$O(1)$.

其次，Gap Buffer 的随机访问速度快。考虑到其本质就是一个数组，对给定的索引查找字符的操作也是$O(1)$的时间复杂度。

但 Gap Buffer 的缺点也不可忽视。首当其冲的就是其数据结构定义决定了其需要连续大块内存。当文件达到数十 GB 级别，操作系统可能很难分配一块连续的内存供读写使用。

此外，Gap 移动函数 GapTo 依赖 std::move 等操作，这些操作的时间复杂度为$O(n)$，是 Gap Buffer 最严重的性能瓶颈。在极端情况下，此操作可能会引发肉眼可见的延迟，使得用户体验感下降。

三、总结

本次实验我们通过 GitHub 等开源平台下载了 Scintilla, VS Code, xi editor 和 GNU Nano 的源码，进一步掌握了 git 用来切换 tag 的使用方法。

在研读代码方面，多语言环境带来了显著的挑战：VS Code (TypeScript)，xi editor (Rust)，Scintilla (C++)，只有 GNU Nano 使用的是 C 语言。但很遗憾的是，尽管 GNU Nano 是一款久负盛名的文本编辑器，但我们仔细研读后发现其并没有使用上述任何一种数据结构。为了减轻语言不同对我们带来的障碍，我们除了自己研读以外还使用了 GitHub Copilot 帮助我们分析代码。

在我们深入分析 Scintilla 的过程中，我们发现其虽然使用了 STL 和面向对象的特性，但并不影响我们分析其数据结构。我们通过研读 Scintilla 源码，意识到任何数据结构都具有局限性，选用什么样的数据结构不仅取决于数据结构本身特性，更取决于应用场景和环境。因此现代软件工程中，做出决策的关键在于对众多要素进行取舍。

综上所述，我们通过本次研究，对文本编辑器的数据结构有了更深层次的认识。