代码编辑器系列 #3 文本的存储 进化篇

2018, Sep 24 by Tesla Ice Zhang

在上上篇文章中我说过，

> 以后的方向主要是讲 JB 式编辑器的实现

在上一篇文章中我又说，

> 那么这篇文章先说点别的吧

简直是王镜泽定理的完美演绎啊。 为什么我要在半个月来第一篇博客开头说这个呢？因为这次讲的依然不是 JB 式编辑器的实现，真香。

关于上一篇博客在说 gap buffer 的时候提到的数据结构论文 Flexichain，我当时说读下来没学到什么东西。 实际上论文有两篇，内容应该基本上都在另一篇讲实现的里面。这是某活跃于 freenode #lisp 的 CL 厨告诉我的。

在我最近的新项目里，我使用了 Flexichain 论文里提到的 Hemlock 编辑器使用的数据结构（我的那个使用C++实现，改自一个单独的 Java 项目 text-sequence，前文提到的实验项目也换成了这个项目里的 `GapBuffer`）， 这个数据结构在 piece table

的论文里有一个类似物叫 Line Span，于是我就直接叫它 `LineSpan` 了。

本文包含这篇论文中的大部分知识。

前文说，

> 由于 Swing 的 API 看起来更低效

其实不是的， Swing 文本编辑器内部实现是一个 `GapContent`，也就是一个过度 OO 设计的 `GapBuffer`（可见写 Swing 的人并不是文盲，而且文化程度不低），这比我那时候用的更高效，所以我那时的想法还是太幼稚了。

既然本文是现代篇（对应上一篇的远古篇），那么讲的肯定是现代编辑器使用的数据结构啦。

介绍一个概念 end of line，通常是 `\n` 但考虑到有些情况下还有别的行分隔符就使用了 end of line 代表这个东西。

Code 早期使用的数据结构

应该很少有人使用可执行文件的名字来称呼这个全名叫 Visual Studio Code 的编辑器吧，正好可以显得很装逼（逃。 根据黑历史考据，我看到 Code 团队在 blog 里自述曾经使用按行存储的策略， 然后他们获得的好处是可以按行运行 Tokenizer，可以提高代码高亮的性能（意思就是直接不考虑包含 end of line 的 Token，很符合前端人员的编程思想）。

这历史应该是黑成碳了。根据上面那个链接里的博客来看，他们也没有使用优化 active line 的策略（比如我使用 GapBuffer ）。

不过呢，按行存储可以进行渲染上的优化（因为行可以被视为一个渲染单位，而且在屏幕移动时每行渲染出来的样子是不变的）， 这在某种意义上也是一种好处了（根据一个研究超算的

软粉的说法，Visual Studio 会缓存每行的代码渲染后得到的 texture）。

LineSpan 的实现相对 GapBuffer 较为繁琐，它在插入的时候需要检查是否有插入 end of line 来考虑是否要拆掉当前的 active line 、 需要在删除的时候检查是否删除了一个 end of line 来考虑是否要合并当前行和下一行。 好处是，可以对行进行批量操作。

目前我的编辑器也是临时使用的这种数据结构，而且比起他们这个还提高了当前行的编辑效率和获取行的效率。 我原本计划使用一个线段树维护每行的长度的前缀和（之所以没有选择我最喜欢的树状数组，是因为我还需要删点），这样可以获得 `O(log(n))` 的 `getLineInfoAt` 等函数。 但是在后来发现要获取对应行的迭代器还需要对链表进行高效的随机访问（或者存储迭代器到线段树里，但这意义已经没有那么大了，因为要换 Piece Table 实现），我就放弃了。 不过鄙视人家的黑历史也没什么意思，毕竟 Code 除了代码编辑的其他地方做的还是很不错的。

我做了一个 LineSpan 的数据结构可视化（非常妙），上传到了 bilibili 和 YouTube。

其他杂七杂八的数据结构

- Split-Merge Tree ，保证 Rope 的最小单位 Sequence 在一个长度范围内，超过就拆，小于就合并两个相邻的，是比平衡树更平衡的平衡树
- Fixed-Size-Buffer ，扁平版 Split-Merge Tree，你也可以叫它 Split-Merge LinkedList

文本序列数据结构的通用性质

看到这里，我们脑中应该已经有了一个文本序列数据结构的一个通用模型了。 在前文讲 Rope 的时候我曾经提到，它是将最小的文本表示作为了一种抽象结构并使用平衡树组合他们，然后举例了几种可能的实现—— 一个获取文本的函数、字符数组（C 风格字符串）、惰性读取的文件、另一颗平衡树等。 GapBuffer 其实也是一种实现。

我们可以尝试把这种思想套到 LineSpan 上，然后发现也完全适用——它其实只是提出了一种新的最小文本表示的实现——`Line` ，并使用链表或者平衡树去组合他们而已。 我们来给这个抽象里的概念起个名字吧。

- 一个 item 指文本序列的最小单位，通常是 `char` 或者 `wchar_t`
- 一个 sequence 指一系列以各种方式组合并在逻辑上是连续的 item ，即刚才提到的抽象结构，比如 `LinkedList<Character>` , `std::list<char>`
- 一个 buffer 指一段在物理上连续的 item ， 一般只在实现中涉及
    - 比如 `std::vector<char>::data` , `java.util.ArrayList<Character>` 里面的那个数组
    - 比如 mmap
- 一个 item 序列如果同时是 sequence 和 buffer 那么它是一个 span

    - 比如 `java.lang.String`, `std::string`
    - buffer 中的逻辑连续的一部分可以算作一个 span，比如 GapBuffer

- 一个 descriptor 指描述一段 sequence 的数据结构，比如 LineSpan 中通常需要一个 `LineInfo` 来保存 span 的位置、长度，那么这个 `LineInfo` 就是一个 descriptor
    - descriptor 通常持有一个 sequence 的指针
    - 这个定义和论文里说的不太一样，是我觉得更好的定义

然后我们来总结一下我们见过的各种东西吧：

- `java.lang.Character`, `char`, `wchar_t`, `ImWchar` 是 item
- Ruby/Lua/JavaScript/Dart/Perl 等语言中的 `string` 是 span
- Rope 是一个递归的 sequence（ `type Rope = BalancedTree (Either Span Rope)` ）
    - 平衡树的 `Node` 类可以看作 span 的 descriptor
- `java.lang.StringBuilder` 是一个大 buffer，最左边那部分是一个 span， descriptor 持有这个 `StringBuilder` 要 build 的 `String` 的长度
- GapBuffer 内部有一个大 buffer，它由两个中间有一个 gap 的不断变化的 span 组成

    - 第一个 span 的 descriptor 存储这个 span 的长度，第二个 span 的 descriptor 存储这个 span 在大 buffer 里的的起始点

- LineSpan 可以看作一个 `java.util.LinkedList<java.lang.CharSequence>` 或者 `std::list<std::string>` 的封装，属于 sequence
    - 链表的迭代器就是 descriptor，里面存的是 sequence，Code 早期的这些 sequence 全是 span，我的实现里 active line 是 sequence，其他行是 span

是不是一下子就搞懂这些名词的含义了？下面我们将使用这套名词介绍一个新数据结构。

Piece Table

这是我目前觉得最好的文本编辑器存储数据所使用的数据结构。Piece Table 由一个巨大的、immutable 的、最好是 lazy 的 buffer（mmap 很适合作为这个 buffer）和一系列指向 span 的 descriptor 组成。Descriptor 保存两个信息，头指针和长度。这一系列 descriptor 可以由链表保存（实现简单），也可以使用平衡树保存（更高效）。由于我们不需要修改或者删除这些 descriptor 指向的 span，我们可以把他们持有的 span 放进一个 buffer，我们称之为 add buffer，可以理解为一个 `ArrayList<Item>`。

在创建一个 Piece Table 的时候，我们需要初始化这个巨大的 buffer，比如文本编辑器打开一个文件的时候就可以使用文件的

mmap。 此时我们也初始化第一个 descriptor，头指针指向大 buffer 的开头，长度就是整个 buffer。

查询

如果给定 `offset` 要取一个 Item，就涉及存储 descriptor 的数据结构了—— 链表的话从头开始遍历 descriptor，找到这个 `offset` 所在的 descriptor 然后就可以从 buffer 里取值了，复杂度 `O(n)`。 如果是平衡树就可以直接从 `offset` 去找，复杂度 `O(log(n))`。

插入

插入任意内容时（假设输入了 `offset`（即 `index`）和 `sequence`），类似 LineSpan 插入 end of line 的情况，需要把这个 `offset` 所在的那个 descriptor 拆掉，变成两个分别描述原本的 descriptor 在 `offset` 前的那一半和后的那一半，然后把 `sequence` 添加到 add buffer 的尾部， 然后在刚才这两个 descriptor 的中间插入一个指向这个 `sequence` 在 add buffer 中位置的 descriptor。

举个例子，假设我们有一个文件，里面有 `Piece` 这几个字符。 我们用它创建一个 Piece Table 后，大 buffer 里就是 `Piece`，长度为 5。 Descriptor 序列是这样的：

Index	Buffer	Offset	Size
0	Big	0	5

我们对他进行 `insert(1, "Tb")`，那么先把 1 这个 `offset` 所在的 0 号（查找 descriptor 的方法同查询）descriptor 拆开，原本长度为 5 的变成一个长度为 1 的和一个长度为 4 的：

Index	Buffer	Offset	Size
0	Big	0	1
1	Big	1	4

然后向 add buffer 添加 `T` 和 `b` 这两个 Item，然后在刚才两个 descriptor 中间插入一个新的 descriptor：

Index	Buffer	Offset	Size
0	Big	0	1
1	Add	0	2
2	Big	1	4

删除

查找到 `offset` 所在的 descriptor，拆成两个并让左边那个的长度减一即可。

Piece Table 的优良性质

- 可持久化数据结构
- 背后的大 buffer 可以完全 lazy 化，内存中的数据量将会巨小
- descriptor 数量少
- 可以方便地使用平衡树优化
- 可以被翻译成『坨坨桌子』，很可爱
- 它可是 Code 使用的数据结构啊，搞懂了之后拿来吹的时候逼格挺高的

Benchmark

看论文去。

<end of blog>

Tweet this

Top

创建一个 issue 以申请评论

Create an issue to apply for commentary

协议/License