函数式编程思想在数据结构优化中的应用

罗翔宇
1200012779

December 24, 2012

Abstract

本文从函数式编程的思想出发，以线段树为例，尝试使用命令式语言模拟函数式语言的行为来实现线段树，保存线段树的所有历史版本，同时用他们之间的共用数据减少时间和空间消耗，从而获得了较传传统线段树更优秀的时间复杂度。

同时这个思想很容易的推广到其他数据结构中，比如：可持久化平衡树，可持久化块状链表，可持久化 Trie 树等。

keywords: 线段树，函数式编程思想，可持久化数据结构

1 传统数据结构

1.1 线段树简介

线段树 (segment tree)[1] 是一种优秀的数据结构，一般被用于解决与一个有限长度数列相关的区间问题。对于一个长度为 n 的数列，可以在 $O(n \lg n)$ 的时间复杂度内构建出线段树，并支持以下操作：

- 查询某段区间 $[l, r]$ 的所有数之和
- 查询某段区间的所有数的最大值
- 修改某个位置上的数字
- 查询整个数列的第 k 大值

[1] 中有详细的线段树的讲解，这不属于本文讨论的范畴之内。

1.2 传统数据结构的缺陷

下面让我们考虑这样一个问题：如果我们已经对数列进行了若干次操作，现在要撤销最近的若干次操作，从而获得这个数列的一个历史版本，这个操作应该如何维护。

一个朴素的想法就是，将每次的操作的数列都保存下来，然后每次撤销的话直接在保存的历史记录中查询就行了。但是注意到这个做法需要耗费大量的内存，大约是 $O(n^2)$ 级别，而当数列比较长的时候是无法实现的。

再考虑一个问题：对于一个序列，每次询问一段区间内数的 k 大值是多少。

对于这个问题，一个目前比较普遍的做法是对序列建立线段树，然后线段树每个节点是一棵平衡树，形成一种“树套树”的结构。对于每次询问二分答案，然后利用树套树进行查询从而获得最终答案。

而这种做法不仅代码实现烦琐，还用到了另一种数据结构——平衡树，而且单次操作的时间复杂度高达 $O(\lg^3 n)$。

下文就从这两个问题展开讨论，将函数式编程思想运用到线段树中，最终在 $O(n \lg n)$ 的时间复杂度，$O(n \lg n)$ 的空间复杂度内解决这个问题。

2 可持久化数据结构

2.1 函数式编程思想

当我们使用函数式编程语言（比如 haskell, scheme）来实现线段树的时候，是不能采用类似命令式语言的方式，每次修改节点的权值来实现的。

最主要的原因就是函数式语言中没有变量这个概念，也就是当前所有的状态是存在一个缓冲区中的，我们不能直接对缓冲区进行修改。而解决的方式就是，对于每次的修改操作，我们直接返回一棵新的线段树来取代当前的线段树。

这种方法听起来和朴素的做法没有什么差别，空间复杂度也是 $O(n^2)$ 级别。但是注意到一棵线段树的树高不会超过 $O(\lg n)$ ，也就是我们于我们一次修改操作相关的节点个数也不会超过 $O(\lg n)$ 。由于函数式编程惰性求值的特性，所以每次操作最多新建 $O(\lg n)$ 个节点，从而总的节点个数为 $O(m \lg n)$ ，其中 m 为操作的次数。

2.2 可持久化线段树

我们现在尝试在普通的命令式语言中来模仿函数式语言中的操作来实现线段树，即每次操作是返回一棵新的树。由于命令式语言没有惰性求值的特性，所以必须要求我们手动来模拟。

假设我们已经有一棵线段树 $\mathcal{T}$，然后当前操作为修改某个位置的数，我们按照传统的线段树的做法，自顶向下递归操作:

- Step 1. 首先新建一个线段树 $\mathcal{T}'$，其根节点为 $\mathcal{T}$ 的根节点，并令节点 u 为 $\mathcal{T}'$ 的根节点
- Step 2. 然后对于 u，不妨设要修改的位置在 u 的左儿子中，则新建一个节点 v 并将其设为 u 的左儿子。注意到此时 u 的右儿子与 $\mathcal{T}$ 中相同位置的节点的右儿子 (不妨设为 w) 是一样的，即此次操作并不影响 w ，于是将 u 的右儿子设为 w 即可
- Step 3. 如果 u 为叶子节点则退出循环，否则令 u 为 u 的左儿子重复 Step 2

当一系列的操作结束后，我们可以得到一些线段树的集合 $\mathcal{H}$，把第 i 次操作后得到的线段树记为 $\mathcal{T}_i$，$\mathcal{H}$ 即为可持久化线段树。

2.3 实际应用

现在回到本文开头提出的那两个问题。

当处理出 $\mathcal{H}$ 之后，问题 1 就变得异常简单了，要获得历史记录中第 i 次操作的结果，直接返回 $\mathcal{T}_i$ 即可。

而对于问题 2，我们以序列中数字的值为关键字来构建 $\mathcal{H}$ ，其中 $\mathcal{T}_i$ 表示序列 $\{a_1, a_2, \cdots .a_i\}$ 所形成的线段树，而线段树的每个节点中维护有多少个数的值落在在一段区间内。

对于查询区间 $[l, r]$ 中第 k 大值的操作，由于数字的出现次数满足区间减法，于是我们可以同时在 $\mathcal{T}_{l-1}$ 和 $\mathcal{T}_r$ 中递归，而数字出现的子树可以直接由 $\mathcal{T}_{l-1}$ 和 $\mathcal{T}_r$ 中对应节点相减而得，其余操作即和传统线段树相同。

2.4 时间空间复杂度分析

注意到线段树的树高不会超过 $O(\lg n)$，所以每次操作新增的节点个数不会超过 $O(\lg n)$，再加上初始线段树的节点个数 $O(n \lg n)$ ，所以总内存消耗为 $O(n \lg n + m \lg n)$。

而每次操作时，其递归准则和传统线段树相同，于是可以照搬传统线段树的时间复杂度分析方式，即每次操作的时间复杂度 $O(\lg n)$

2.5 扩展

我们可以将相同的思想拓展到块状链表中来构建可持久化块状链表。利用可持久化块状链表可以以 $O(n\sqrt{n})$ 的空间和 $O(m\sqrt{n})$ 时间复杂度来解决区间众数查询问题。

大大改善了区间众数查询问题一直一来的一种离线后利用曼哈顿最小生成树的时间复杂度为 $O(n \lg n + m\sqrt{n} \lg n)$ 的烦琐解法。

而我们如果将函数式编程思想运用到 AVL 平衡树上，可以使用十几 K 的内存来管理长达几百 M 的字符串，而且支持历史记录撤销操作，这为文本编辑器的实现提供了一个新的思路。

References

[1] ``Segment Tree", `http://en.wikipedia.org/wiki/Segment_tree`