持久化数据结构

汪直方

宁波市镇海蛟川书院

2024 年 2 月 3 日

通常的数据结构问题我们只需要数据结构的当前结构及信息，但有时我们需要之前的结构及信息，我们将这类需求称为持久化，满足该需求的数据结构称为持久化数据结构，可以进行持久化的数据结构称为可持久化数据结构。

通常的数据结构问题我们只需要数据结构的当前结构及信息，但有时我们需要之前的结构及信息，我们将这类需求称为持久化，满足该需求的数据结构称为持久化数据结构，可以进行持久化的数据结构称为可持久化数据结构。

我们可以将常见的对数据结构的操作分为三类：

通常的数据结构问题我们只需要数据结构的当前结构及信息，但有时我们需要之前的结构及信息，我们将这类需求称为持久化，满足该需求的数据结构称为持久化数据结构，可以进行持久化的数据结构称为可持久化数据结构。

我们可以将常见的对数据结构的操作分为三类：

1. 查询：给定一个或多个数据结构与查询参数，不对数据结构进行修改的情况下返回查询结果。

通常的数据结构问题我们只需要数据结构的当前结构及信息，但有时我们需要之前的结构及信息，我们将这类需求称为持久化，满足该需求的数据结构称为持久化数据结构，可以进行持久化的数据结构称为可持久化数据结构。

我们可以将常见的对数据结构的操作分为三类：

1. 查询：给定一个或多个数据结构与查询参数，不对数据结构进行修改的情况下返回查询结果。
2. 修改：给定一个数据结构与修改参数，返回一个修改后的数据结构。

通常的数据结构问题我们只需要数据结构的当前结构及信息，但有时我们需要之前的结构及信息，我们将这类需求称为持久化，满足该需求的数据结构称为持久化数据结构，可以进行持久化的数据结构称为可持久化数据结构。

我们可以将常见的对数据结构的操作分为三类：

1. 查询：给定一个或多个数据结构与查询参数，不对数据结构进行修改的情况下返回查询结果。
2. 修改：给定一个数据结构与修改参数，返回一个修改后的数据结构。
3. 合并：给定多个数据结构与合并参数，返回一个合并后的数据结构。

通常的数据结构问题我们只需要数据结构的当前结构及信息，但有时我们需要之前的结构及信息，我们将这类需求称为持久化，满足该需求的数据结构称为持久化数据结构，可以进行持久化的数据结构称为可持久化数据结构。

我们可以将常见的对数据结构的操作分为三类：

1. 查询：给定一个或多个数据结构与查询参数，不对数据结构进行修改的情况下返回查询结果。
2. 修改：给定一个数据结构与修改参数，返回一个修改后的数据结构。
3. 合并：给定多个数据结构与合并参数，返回一个合并后的数据结构。

我们往往会根据具体问题遇到不同难度的持久化需求。

常见的持久化需求可以分为四种：

常见的持久化需求可以分为四种：

1. 可撤销：支持撤销上一次操作，只在当前版本的数据结构上进行修改和查询。

常见的持久化需求可以分为四种：

1. 可撤销：支持撤销上一次操作，只在当前版本的数据结构上进行修改和查询。
2. 部分持久化：支持在历史版本上进行查询，只在最后一个版本的数据结构上进行修改。

常见的持久化需求可以分为四种：

1. 可撤销：支持撤销上一次操作，只在当前版本的数据结构上进行修改和查询。
2. 部分持久化：支持在历史版本上进行查询，只在最后一个版本的数据结构上进行修改。
3. 完全持久化：在部分持久化的基础上支持在一个历史版本上进行修改（一般不说明时不包含可合并的需求）。

常见的持久化需求可以分为四种：

1. 可撤销：支持撤销上一次操作，只在当前版本的数据结构上进行修改和查询。
2. 部分持久化：支持在历史版本上进行查询，只在最后一个版本的数据结构上进行修改。
3. 完全持久化：在部分持久化的基础上支持在一个历史版本上进行修改（一般不说明时不包含可合并的需求）。
4. 可合并：支持在任意历史版本上进行查询、修改、合并。

常见的持久化需求可以分为四种：

1. 可撤销：支持撤销上一次操作，只在当前版本的数据结构上进行修改和查询。
2. 部分持久化：支持在历史版本上进行查询，只在最后一个版本的数据结构上进行修改。
3. 完全持久化：在部分持久化的基础上支持在一个历史版本上进行修改（一般不说明时不包含可合并的需求）。
4. 可合并：支持在任意历史版本上进行查询、修改、合并。

一般提及持久化默认指完全持久化。

我们先考察当数据结构为单个变量的情况，任意修改与合并操作显然等效于查询与赋值：

我们先考察当数据结构为单个变量的情况，任意修改与合并操作显然等效于查询与赋值：

1. 可撤销：支持赋值操作和查询操作，以及将当前版本退回最后一次未被撤销的赋值操作前（即撤销最后一次未被撤销的赋值操作）。显然该问题等效于维护一个栈，每次操作压入、查询或弹出栈顶。

我们先考察当数据结构为单个变量的情况，任意修改与合并操作显然等效于查询与赋值：

1. 可撤销：支持赋值操作和查询操作，以及将当前版本退回最后一次未被撤销的赋值操作前（即撤销最后一次未被撤销的赋值操作）。显然该问题等效于维护一个栈，每次操作压入、查询或弹出栈顶。
2. 部分持久化、完全持久化、可合并：显然赋值操作不关心修改前版本的信息，相当于新建一个版本或在给定版本上查询。显然该问题等效于维护一个数组，支持尾部插入，查询任意位置的值。

我们先考察当数据结构为单个变量的情况，任意修改与合并操作显然等效于查询与赋值：

1. 可撤销：支持赋值操作和查询操作，以及将当前版本退回最后一次未被撤销的赋值操作前（即撤销最后一次未被撤销的赋值操作）。显然该问题等效于维护一个栈，每次操作压入、查询或弹出栈顶。
2. 部分持久化、完全持久化、可合并：显然赋值操作不关心修改前版本的信息，相当于新建一个版本或在给定版本上查询。显然该问题等效于维护一个数组，支持尾部插入，查询任意位置的值。

注意到数据结构的底层实现往往由变量组成，我们接下来对一般数据结构的持久化进行考察。

对于可撤销的情况，我们可以只维护当前版本的完整数据结构，并用一个修改栈维护每次修改操作在数据结构上修改的变量的位置与修改前的值，另用一个栈维护每个版本的修改栈的栈顶位置，每次撤销时先将维护修改栈的栈顶位置的栈弹出栈顶，再根据其新栈顶对修改栈不断根据栈顶还原数据结构并弹出栈顶直至修改栈还原。显然若时间复杂度不是均摊的，则单次操作时间复杂度不变，否则单次操作时间复杂度可能会达到单次操作的最坏时间复杂度（即均摊复杂度失效）。

对于可撤销的情况，我们可以只维护当前版本的完整数据结构，并用一个修改栈维护每次修改操作在数据结构上修改的变量的位置与修改前的值，另用一个栈维护每个版本的修改栈的栈顶位置，每次撤销时先将维护修改栈的栈顶位置的栈弹出栈顶，再根据其新栈顶对修改栈不断根据栈顶还原数据结构并弹出栈顶直至修改栈还原。显然若时间复杂度不是均摊的，则单次操作时间复杂度不变，否则单次操作时间复杂度可能会达到单次操作的最坏时间复杂度（即均摊复杂度失效）。

另外，若操作可逆，撤销操作也可以通过执行修改操作的逆操作完成，不会破坏均摊复杂度。

对于部分持久化的情况，我们可以视为有一个版本序列，一开始只有初始版本，每次修改操作会将序列末尾的版本进行修改并插入至序列末尾，每次查询操作给定一个或多个版本在序列中的编号与查询参数进行查询。

对于部分持久化的情况，我们可以视为有一个版本序列，一开始只有初始版本，每次修改操作会将序列末尾的版本进行修改并插入至序列末尾，每次查询操作给定一个或多个版本在序列中的编号与查询参数进行查询。

在可以离线且单次查询只在一个历史版本上进行的情况下，我们可以将查询预先存储在版本序列的对应位置上，只维护当前版本的数据结构，按版本序列枚举当前版本并对当前版本的询问进行查询，从当前版本到下一个版本只需进行一次修改即可。可以发现在这种情况下无需任何持久化。

对于完全持久化的情况，我们可以视为有一棵版本树，以初始版本为根结点，每次修改操作会给定一个结点与操作参数，将该结点对应版本根据参数进行修改并作为叶子插入到给定结点的儿子集合中。

对于完全持久化的情况，我们可以视为有一棵版本树，以初始版本为根结点，每次修改操作会给定一个结点与操作参数，将该结点对应版本根据参数进行修改并作为叶子插入到给定结点的儿子集合中。

显然每个非根结点的对应版本均由父结点经过一次修改操作得到。

对于完全持久化的情况，我们可以视为有一棵版本树，以初始版本为根结点，每次修改操作会给定一个结点与操作参数，将该结点对应版本根据参数进行修改并作为叶子插入到给定结点的儿子集合中。

显然每个非根结点的对应版本均由父结点经过一次修改操作得到。

在可以离线且单次查询只在一个历史版本上进行的情况下，我们可以离线处理出版本树并将询问存储在版本树的对应结点上，然后遍历版本树，维护当前结点对应版本的数据结构，相当于可撤销的持久化需求。

对于可合并的情况，我们可以视为有一个版本图，每个结点由其前驱合并或修改得到，每次修改或合并操作会新建一个结点并从给定版本对应结点向新结点连边。显然版本图是有向无环图。

Fat Node

Fat Node 是指对原先的每个变量存储一个二元组的序列，表示这个变量在某个时间是某个值。

Fat Node

Fat Node 是指对原先的每个变量存储一个二元组的序列，表示这个变量在某个时间是某个值。

Problem (部分持久化数组)

给定一个长度为 n 的数组 a，满足 $\forall i \in [1,n] \cap \mathbb{N}, a_i \in A$，$q$ 次操作：

1. 给定 $p \in [1,n] \cap \mathbb{N}, x \in A$，执行操作 $a_p \leftarrow x$。
2. 给定 $v \in [1,q'] \cap \mathbb{N}, p \in [1,n] \cap \mathbb{N}$，查询 a_p 在第 v 次操作前的值，当前为第 q' 次操作。

强制在线。

Solution

我们对数组的每个位置维护修改时间与修改值的序列，若直接使用动态扩容的数组维护则修改时间复杂度 $\Theta(1)$，查询时间复杂度 $\Theta(\log q_1)$。

Solution

我们对数组的每个位置维护修改时间与修改值的序列，若直接使用动态扩容的数组维护则修改时间复杂度 $\Theta(1)$，查询时间复杂度 $\Theta(\log q_1)$。

若使用 vEB 树或类似数据结构可以做到单次修改与查询时间复杂度 $\Theta(\log\log q_1)$。

Solution

我们对数组的每个位置维护修改时间与修改值的序列，若直接使用动态扩容的数组维护则修改时间复杂度 $\Theta(1)$，查询时间复杂度 $\Theta(\log q_1)$。

若使用 vEB 树或类似数据结构可以做到单次修改与查询时间复杂度 $\Theta(\log\log q_1)$。

Theorem 2.1

持久化数组问题在长度为 n 的持久化数组进行 q 次修改且值域 $|A| > q$，$q = n^{O(1)}, w = n^{O(1)}$ 的情况下若空间复杂度为 $O\left(q\log^{O(1)} q\right)$ 则在 cell probe 模型下单次查询时间复杂度为 $\Omega(\log\log q)$。

Solution

我们对数组的每个位置维护修改时间与修改值的序列，若直接使用动态扩容的数组维护则修改时间复杂度 $\Theta(1)$，查询时间复杂度 $\Theta(\log q_1)$。

若使用 vEB 树或类似数据结构可以做到单次修改与查询时间复杂度 $\Theta(\log\log q_1)$。

Theorem 2.1

持久化数组问题在长度为 n 的持久化数组进行 q 次修改且值域 $|A| > q$，$q = n^{O(1)}, w = n^{O(1)}$ 的情况下若空间复杂度为 $O\left(q\log^{O(1)} q\right)$ 则在 cell probe 模型下单次查询时间复杂度为 $\Omega(\log\log q)$。

证明见前驱查找问题处。

对于完全持久化数组，我们可以维护版本树的括号序列，左括号为该版本的值，右括号为该版本前一个版本的值，查询时找到不超过查询版本左括号位置的最后一个位置的值即可。

对于完全持久化数组，我们可以维护版本树的括号序列，左括号为该版本的值，右括号为该版本前一个版本的值，查询时找到不超过查询版本左括号位置的最后一个位置的值即可。

我们可以使用平衡树维护二元组，此时动态离散化可以单次操作 $\Theta(\log q_1)$ 次改变标号，无需用 w 或 $\Theta(\log n)$ 为块长分块，单次操作时间复杂度 $\Theta(\log q_1)$。

对于完全持久化数组，我们可以维护版本树的括号序列，左括号为该版本的值，右括号为该版本前一个版本的值，查询时找到不超过查询版本左括号位置的最后一个位置的值即可。

我们可以使用平衡树维护二元组，此时动态离散化可以单次操作 $\Theta(\log q_1)$ 次改变标号，无需用 w 或 $\Theta(\log n)$ 为块长分块，单次操作时间复杂度 $\Theta(\log q_1)$。

注意到动态离散化可以做到单次只改变均摊 $\Theta(1)$ 个标号，我们依然可以使用 vEB 做到单次操作均摊时间复杂度 $\Theta(\log\log q_1)$。

Solution

显然朴素的 Path Copy 是行不通的。例如当我们复制一个结点时，指向它的结点的出边会发生变化，我们此时需要对指向它的结点也进行递归复制，其时间复杂度显然是无法接受的。

Solution

显然朴素的 Path Copy 是行不通的。例如当我们复制一个结点时，指向它的结点的出边会发生变化，我们此时需要对指向它的结点也进行递归复制，其时间复杂度显然是无法接受的。

我们考虑进行 Fat Node，不妨先对部分持久化的情况进行讨论。

Solution

显然朴素的 Path Copy 是行不通的。例如当我们复制一个结点时，指向它的结点的出边会发生变化，我们此时需要对指向它的结点也进行递归复制，其时间复杂度显然是无法接受的。

我们考虑进行 Fat Node，不妨先对部分持久化的情况进行讨论。

我们对每个结点指向相邻结点的边维护在某个时间点被修改为某个值，这样我们每次复制结点只用在指向它的结点将当前时间及新结点的编号放入对应位置即可。例如部分持久化时我们每次复制结点要修改指向它的结点出边编号时只需要将当前时间和新结点编号放入这个动态数组的末尾即可，我们在历史版本中用到相邻结点的编号时只用定位到对应时间即可。

Solution

如果我们直接朴素实现的话再每个结点查询非最新版本的相邻结点编号时都需要进行二分，导致很多操作最坏情况下的时间复杂度会相比原先多乘上 $O(\log q_1)$，我们考虑类似分散层叠进行优化。

Solution

如果我们直接朴素实现的话再每个结点查询非最新版本的相邻结点编号时都需要进行二分，导致很多操作最坏情况下的时间复杂度会相比原先多乘上 $O(\log q_1)$，我们考虑类似分散层叠进行优化。

我们取常数 c，当一个结点的动态数组超过 c 个元素时将当前结点复制出一个新的结点维护从第 $c+1$ 个时间开始的状态。（显然此时由于 c 是常数，可以将动态数组改为静态数组。）

我们进行时间复杂度分析：

我们进行时间复杂度分析：

注意到入度最多为 d，我们一次只会在 d 个动态数组内插入当前结点的编号。我们考虑对每个动态数组 x 设一个 w 值作势能函数，满足当 $\lceil w(x) \rceil$ 增大时才可能对 x 所在的结点进行复制，这样复制结点的时间复杂度就可以从增加的 w 之中均摊支出。建立一个结点为动态数组的有向图 G，存在边 (x, y) 当且仅当 x 产生的复制会在 y 中插入一个元素。我们取 $c = d + 1$。

我们在出于其它原因向一个动态数组 x 插入一个元素时，对于任意 y，将 $w(y)$ 加上 $c^{-1-\mathrm{dist}(x,y)}$。我们考虑往 x 中插入一个元素时 $\sum w(u)$ 的增量：注意到满足 $\mathrm{dist}(x,y)=i$ 的 y 不超过 d^i 个，因此总和的增量不超过 $c^{-1}\sum_{i=0}^{+\infty}(d/c)^i=1/(c-d)=1$。因此我们可以均摊 $O(d)$ 的时间复杂度内修改一个结点，在严格 $\Theta(d)$ 的时间复杂度内得到任意历史版本相邻结点的编号。

我们在出于其它原因向一个动态数组 x 插入一个元素时，对于任意 y，将 $w(y)$ 加上 $c^{-1-\mathrm{dist}(x,y)}$。我们考虑往 x 中插入一个元素时 $\sum w(u)$ 的增量：注意到满足 $\mathrm{dist}(x,y)=i$ 的 y 不超过 d^i 个，因此总和的增量不超过 $c^{-1}\sum_{i=0}^{+\infty}(d/c)^i=1/(c-d)=1$。因此我们可以均摊 $O(d)$ 的时间复杂度内修改一个结点，在严格 $\Theta(d)$ 的时间复杂度内得到任意历史版本相邻结点的编号。

对于完全持久化的情况，我们可以类似地使用动态离散化维护版本树的括号序，将尾部插入改为任意位置插入。