技术路线报告

总体框架

入口

OpenHarmony 提供了全量备份工具与接口, 其增量备份仅有接口, 并没有实现. 鉴于算法差异以及冗余性等问题, 我们计划仅实现差异备份, 并用添加子系统的方式进行部署.

与 OpenHarmony 相同, 备份工具通过命令行方式进入. 即

```
backup_tool <subcommand> <parameter1> <parameter2>
```

方式, 具体是否使用与 OpenHarmony 备份工具同样的入口有待进一步确定.

进程组织策略

同样与 OpenHarmony 相同. 差异备份命令行 IO 系统与具体的备份服务模块作为独立的进程, 通过 IPC 通信. 调用备份工具后, 就启用对应的模块.

作为程序被命令行调用, 与作为备份服务被定时启动是两个不同的应用场景, 前者需要尽可能快的完成备份, 所以需要尽可能多的利用CPU和内存资源. 后者在后台静默运行, 不宜占用过高资源, 影响用户使用体验, 所以需要占用更少的内存和CPU.

为此, 入口提供一个可选参数, 用于控制备份所用策略, 也即选择所使用的线程数. 当进行多线程备份时, 主服务线程对需要备份的目录进行分块, 分发给线程池执行.

目录分块

多线程备份同一目录, 需要对目录分块, 使其可以被视为两个目录, 同时进行备份.

由于差异备份的特殊性, 目录分块需要保证在文件更新后

1. 已有文件所属的分块不会改变
2. 不会多出不属于任何分块的文件

这里使用文件路径哈希来解决上述问题. 假设最多支持 8 个线程, 则无论使用多少个线程, 都通过路径的某种哈希值将文件夹分成 8 块. 一旦有线程空闲, 它就获得其中的一块, 开始进行处理.

这种分类方法不能保证线程间工作负载均衡, 最糟糕的情况下, 所有文件都放在一个线程上, 导致并行执行的时间比串行还慢(因为有通信时间和一开始分类时间), 但它保证线程进行备份工作时不会遇到同步问题.

关于分块是否会降低局部性从而降低性能, 有待进一步测试.

全量备份恢复

全量备份

全量备份发生在第一次运行差异备份时.

OpenHarmony 的全量备份为了节约空间, 对小文件进行打包处理, 而大文件则直接原样复制.

此处全量备份的文件需要便于目录差异分析, 这使得打包小文件产生很多额外运算开销, 此处不打包, 所有文件原样复制.

对大文件的分割通常不能减少差异备份的数据量, 因为对一个文件的修改通常是插入/删除, 每种操作都导致二进制数据整体的偏移, 最终每个块都不一样, 需要将整个文件备份一遍.

采用 `diff` 算法可以分析文件差异, 并因此保留最少的数据, 然而其动态规划的内存需求可能不为备份运行所接受, 这点有待进一步实验, 此处就采取正常的差异更新策略, 有差异就整个复制.

备份文件存储

一块的文件存储在备份目录

```
.../backup/[version]/[slice_index]/file/
```

文件夹中.

为了便于差异分析, 一块同时还有一个文本文件 `manage.txt` 存储在 `file` 文件夹外, 记录了

1. 文件路径
2. 文件大小

`manage.txt` 中的文件组织成目录文件树的格式.

全量恢复

全量恢复发生在仅运行过全量备份就恢复时.

全量恢复将首先删除目标目录下的全部文件. 虽然可以比较差异再修改, 但应用目录的恢复多发生在卸载后重装的情形, 比较差异的用途较小, 这里不考虑.

全量恢复的过程较简单, 直接调用 OpenHarmony 相关接口进行目录复制.

差异备份恢复

差异备份

当第一次进行全量备份后, 发生的备份就是差异备份.

目录差异表示

之前提到的 `manage.txt` 文件可以被构造成一个目录树结构, 我们通过这样一个结构上的标记来代表对目录的修改.

对文件夹来说, 一个文件夹可以被

- 新建, 标记 `new`, 恢复该文件夹是从备份复制.
- 删除, 标记 `delete`, 这时该文件夹下没有文件, 自身也不存在.
- 新建后删除, 等于不存在
- 删除后新建, 标记 `new`, 符合其特征

还有特殊的标记

- `modify`, 子文件夹发生了修改
- `stay`, 完全不变, 默认标记

对文件来说, 一个文件可以被

- 新建, 标记 `new`, 恢复该文件是从备份复制.
- 删除, 标记 `delete`, 这时该文件不存在
- 删除后新建, 标记 `new`, 恢复该文件是从备份复制.
- 修改, 标记 `new`, 恢复该文件是从备份复制.

还有特殊的标记

- `stay`, 完全不变, 默认标记

如若支持 `diff` 算法分析差异, 则修改的标记有所变化, 此处不讨论那种标记系统.

目录差异分析

线程从主线程获得将要分析的文件目录, 这可以加载为目录树(新树). 读取 `manage.txt`, 加载全量备份所产生的目录树(旧树).

通过 BFS 遍历旧树和新树, 以层为单位, 可以将旧树和新树的节点进行比较. 以下比较未标记的节点并对新树做出更改

- 若旧树拥有该节点, 新树没有, 代表该节点被新树删除, 在新树创建该节点, 标记为 `delete`, 删除旧树该节点子节点
- 若旧书没有该节点, 新树有, 代表该节点为新树创造, 在新树标记为 `new`, 相应地复制文件夹/文件到备份目录

- 若两树都有一节点, 若为文件, 先对比文件大小后对比文件内容, 若发现不同则将其标记为 `new`, 复制文件到备份目录, 并且将其父节点都标记为 `modify`.

如此遍历分析整个目录. 此时新树存储了所有的更改. 释放旧树, 在

```
.../backup/[version]/[slice_index]/change.txt
```

中记录更改(非 `stay` 节点).

多次差异备份

实验文档中步骤

- 备份A1, 得到备份B1
- 分析A2与A1差异, 得到备份B21
- 分析A3与A2差异, 得到备份B32
- 合并B21与B32,得到B31

实际上应用在存在A3时不可能还存在A2, 一个应用只有一个时间的数据目录, 所以可以如下

- 备份A1, 得到备份B1
- 分析A2与B1差异, 得到备份B21
- 分析A3与B1的差距, 得到备份B31

或者使用增量备份的逻辑

- 备份A1, 得到备份B1
- 分析A2与B1差异, 得到备份B21
- 通过B1与B21复原临时系统T2
- 分析A3与T2差异, 得到备份B32

无论是哪种方法, 目前未讨论的步骤是

- 从全量备份和增量备份复原目录
- 合并两次的增量备份差异

前者是恢复的内容, 只讨论后者.

备份差异合并

从两个 `change.txt` 加载旧树和新树. 这些树每个节点存储

- 本级文件名
- 文件大小
- 标记, 是 `new`, `modify` 或 `delete`

这里添加标记 `null`, 代表该节点在树中不存在.

BFS 按层遍历, 根据节点在旧树和新树中的情况更新新树

- `new -> modify`, 这是添加并修改了文件夹, 改为 `new`
- `new -> delete`, 这是添加又删除了文件或文件夹, 在旧树删除其子节点.
- `new -> null`, 这是添加了文件(夹), 改为 `new`, 在新备份对应位置添加旧备份文件(夹)
- `modify -> modify`, 这是修改了文件夹, 不变
- `modify -> delete`, 这是修改又删除了文件夹, 在旧树删除其子节点.
- `modify -> null`, 这是修改了文件夹, 改为 `modify`, 在新备份对应位置添加旧备份文件(夹)
- `delete -> new`, 这是删除又添加了文件(夹), 不变, 不再遍历该节点子节点
- `delete -> null`, 这是删除了文件(夹), 改为 `delete`.
- `null -> new`, 这是添加了文件(夹), 不变
- `null -> modify`, 这是修改了文件夹, 不变
- `null -> delete`, 这是删除了文件夹, 不变

更新新树后, 新备份文件就是合并的备份文件, 重写 `change.txt`, 得到合并的备份.

对于目录树相关算法的剪枝, 留作进一步的工作.

差异恢复

从全量备份和差异恢复

从全量 `manage.txt` 与差异 `change.txt` 加载旧树和新树.

BFS 按层遍历, 根据节点在旧树和新树中的情况进行恢复, `x` 表示存在, `null` 表示不存在.

- `x -> new`, 这是更改了文件(夹), 将差异备份文件(夹)复制到恢复目录, 删除旧树子节点
- `x -> modify`, 这是更改了文件(夹), 将差异备份文件(夹)复制到恢复目录
- `x -> delete`, 这是删除了文件(夹), 删除旧树子节点
- `x -> null`, 这是不变, 将全量备份文件(夹)复制到恢复目录
- `null -> new`, 这是新建了文件(夹), 将差异备份文件(夹)复制到恢复目录
- `null -> delete`, 不变

这样就完成了恢复.

`x -> new` 和 `x -> modify` 存在差异, 例如一个文件夹先被删除后又创建, 则其子文件(夹)则与全量备份没有任何关联, 故删除全量备份的子节点. `modify` 状态的存在是必要的, 它让我们知道究竟哪些目录发生了更改, 便于差异 `change.txt` 的存储.

文件不会因为并行而产生访问冲突, 但是文件夹的创建具有此类可能, 因为一个文件夹下的内容可能被分到不同的块, 文件夹可能重复创建. 此种算法必须先复原文件夹, 然后才能并行进行文件读写.