第 8 章

データの atomic な永続化方法

ファイルに対する `write()` システムコールで書いたデータの永続化は `fsync()` か `fdatasync()` で，mmap されたファイルの変更データの永続化は `msync()` を使えば良いのでした．基本的な永続化の操作はこれだけしかなのですが，もう少し高い抽象度から見た操作として，atomic に書き込みを永続化するにはどうすれば良いかについて考えていきましょう．

- atomic にファイルに追記したい場合
- atomic にファイルの一部を上書きしたい場合
- atomic にファイルまるごと上書きしたい場合
- Copy-on-Write (CoW)
- その他

8.1 atomic にファイルに追記したい場合

この方法は追記ファイルである WAL ファイルの atomic 書き込みに使えます．前提として，データを書きたいのファイル内のオフセットは分かっているものとします．それまでも同様に atomic に追記していれば，ファイルの終端オフセットは分かるはずなので，それがこれから追記するデータの先頭オフセットとなります．

まずは，書きたいデータの checksum を計算します．アルゴリズムは crc32 など好きなものを選んでください．ただし，アルゴリズムの特性を良く理解してから選びましょう．不完全なデータなのに完全だと誤判定されてしまうリスクを認識してください，ということです．

データを実際に書くときは，checksum，データサイズ，データの中身を書き，永続化保証が必要なら直後に永続化命令を発行します．Checksum は典型的には固定サイズですが，そうでない場合は checksum データの終端が分かるようにします．データサイズも同様です．例えば 64bit little endian unsigend integer を使うのであれば，8 bytes 固定です．このフォーマットは一例ですが，それぞれのデータの位置が後で分かるようになっていることが必要です．データサイズが固定長の場合は，データサイズを記録しなくても良いでしょう．可変長のデータについては，別途固定長のサイズ情報を記録することで，区切りが分かるというわけです．C 言語の文字列のように，区切り文字を使うケースもありますが，データに区切り文字そのものを格納できないという制約があったり，頑張って格納しようとするとエスケープ処理が必要だったり，前から順番に読まないと区切り

が分からないというデメリットがありますので，データを永続ストレージに格納するというコンテクストでは多くの場合サイズ情報を別途記録する方が良いと考えらえます．特にエスケープ処理はセキュリティの穴が空きやすいので原則使わないようにしましょう．

Crash recovery するときには，まず checksum とデータサイズとデータの中身をメモリに読み込みます．データの checksum を再計算し，記録されているものと一致すれば，それは完全に書かれたものと信じて良く，そのデータは書かれたことにします．Checksum が不一致ならば，データは不完全と判断して，書かれなかったことにします．

「完全なデータが書かれているならば checksum が一致する」という命題は真です．しかしその逆である，「checksum が一致するならば完全なデータが書かれている」という命題は必ずしも真ではありません．だから，信じると書きました．完全なデータではないのに checksum が偶然一致してしまった，という稀な事象が起きていないと信じているということです．このリスクを少しでも減らすためには，checksum アルゴリズムの選定に気をつかうだけでなく，checksum が一致したとされるデータが (DBMS レベルで，さらにはアプリケーションレベルで) 正しいかどうかを確認するのが良いでしょう．

8.2 atomic にファイルの一部を上書きしたい場合

MySQL などでは double write という手段が使われています．Double write buffer という専用のファイルを用意し，そこにまず書いて永続化してから，本体ファイルを上書きし，最後に double write buffer から消します．Double write buffer に書くときは atomic に追記するテクニックを使います．Crash recovery 時に double write buffer に残っているデータは中途半端に書かれている可能性があるので，上書き操作を redo することで atomic 性を担保します．

実質的には double write buffer は WAL における redo log と似た働きをすると思って良いでしょう．WAL を使っているシステムの場合，WAL ファイルにログとして上書きしたいイメージを追記することで，Crash recovery 時に上書き操作を redo することで atomic 性を担保できます．毎回 WAL などに書くとログファイルが膨れあがってしまうので，checkpoint 後に初めて上書きする場合などに留めるなどの最適化が行われます．この方式は PostgreSQL で採用されています．

典型的なブロックストレージでは block ひとつの書き込みについては atomic 性を持っています．ファイルシステムが何か特別なことをしていない限り，昨今の Linux では 4KiB sector HDD が存在するので，実体としての atomic write 単位が 512B だったり 4KiB だったりしますが，小さい方に合わせて，512B alignment された領域に 512B の `write()` システムコールを 1 回のみ用いた書き込みであれば，atomic に書かれると思います．HDD に限らず flash memory で作られたブロックストレージも，この性質を満たすように作られているはずです．Linux のファイルシステムのスーパーブロックの書き込みはこの性質を仮定していると思います．これはシステム依存の挙動であることに十分注意してください．あなたがシステム全てをコントロールできる立場にあり，特定の書き込みが atomic に書かれることが確信できるならご自分の責任でそれに依存した設計をしても構いません．それを信じることができない環境では，atomic 追記のテクニックを使う必要があります．

8.3 atomic にファイルをまるごと上書きしたい場合

設定ファイルなどを更新するときに，ファイルまるごと atomic に変更したいときがあります．これは write-once の挙動なので，プログラムが単純になり，うれしいです．大きなファイルでも使えますが，変更点が少ない場合は無駄が多いですね．

まず temporary な名前をつけて対象と同一ディレクトリ内にファイルを作成し，新しい中身を書き込み，永続化します．(同じファイルシステムに属する場所でないと次の rename が失敗するので注意してください．) 次に，`rename()` システムコールを使って名前を対象のものに変えます (上書きします)．`rename()` は atomic 動作が保証されており，`rename()` の前後で対象ファイルを open した人は，以前のファイル内容か，新しいファイル内容のどちらかを必ず見ることになります．古いファイルは，既に open している人がいなくなったら実際に削除されます．以前は `rename()` によるファイルメタデータの変更を永続化するためにファイルの存在するディレクトリエントリを永続化する必要がある，という話がありましたが，最近のファイルシステムでは気にしなくて良いかも知れません．参考 (1): http://d.hatena.ne.jp/kazuhooku/20100202/1265106190 参考 (2): https://thunk.org/tytso/blog/2009/03/15/dont-fear-the-fsync/

8.4 Copy-on-Write (CoW)

上書きするときにコピーする，という名前通りの手法です．CoW というときは，メモリ断片とそれを指すポインタ (ポインタは atomic に書き換えられることが前提) の話と，ディスク上で Tree 構造を扱うときの話があるように思います．今回は後者の話です．Tree ノードの一部を上書きしたいとき (通常は leaf ノード)，新しいノードを確保し，ノードの中身をまるごとコピーして，必要な変更を新しいノードに加えます．新しい変更は atomic に実行する必要はありません．新しいノードはまだ root ノードから辿れない状態なので．ノードの位置情報を参照しているノード上で書き換える必要がありますが，その書き換えが atomic に出来るならそうして終わりです．atomic に書き換えできないなら，やっぱり位置情報を書き換えたノードのコピーを作って,,, という操作を再帰的に繰り返します．いずれ一番上の root ノードに到達します．root ノードが atomic に書けるならそれを atomic に書き換えて終わりです．そうでないなら，root ノードの変更されたコピーを用意して，新しい root ノードの位置情報を，何らかの方法でその tree 全体を管理するデータ (root の位置情報が記録されている場所) を atomic に書き換えます．CoW のメリットは，WAL が不要な点と，(root ノードまで CoW する場合に) 自然に過去の snapshot が作れる点です．(root ノードまで CoW する場合の) デメリットは，tree の深さと同じ数のノードの CoW を実行する必要がある点です．

8.5 その他

他にもあるかも知れません．興味があれば探求してみてください．