Code Gear、 Data Gear に基づく OS のプロトタイプ

仲 松 栞[†1] 照屋 のぞみ[†2] 河 野 真 治[†2]

Nakamatsu Shiori ,[†1] eruya Nozomi [†2] and Shinji KONO [†2]

1. 研究目的と背景

2. 非破壊的木構造データベース Jungle

Jungle は、当研究室で開発を行っている非破壊的木構造データベースで、Java を用いて実装されている。非破壊的木構造とは、データの編集を一度生成した木を上書きせず、ルートから編集を行う位置までのノードをコピーする特徴を持つ（図?? ）。これにより、読み込み中にデータが変更されないことが保証されているため、書き込みと読み込みを同時に行うことできる。 Jungle は名前付きの複数の木の集合からなり、木は複数のノードの集合でできている。ノードは自身の子のリストと属性名、属性値を持ち、データベースのレコードに相応する。通常のレコードと異なるのは、ノードに子供となる複数のノードが付くところである。Jungle では、子供からの親へのポインタは持たないため、親から子への片方向の参照しかできない。 通常の RDB と異なり、Jungle は木構造をそのまま読み込むことができる。例えば、XML や Json で記述された構造を、データベースを設計することなく読み込むことが可能である。また、この木を、そのままデータベースとして使用することも可能である。しかし、木の変更の手間は木の構造に依存する。特に非破壊木構造を採用している Jungle では、木構造の変更の手間は O(1) から O(n) となりえる。つまり、アプリケーションに合わせて木を設計しない限り、十分な性能を出すことはできない。逆に、正しい木の設計を行えば高速な処理が可能である。 Jungle は基本的に on memory で使用することを考えており、一度、木のルートを取得すれば、その上で木構造として自由にアクセスして良い。Jungle は分散データベースを構成するように設計されており、分散ノード間の通信は木の変更のログを交換することによって行われる。持続性のある分散ノードを用いることで Jungle の持続性を保証することができる。

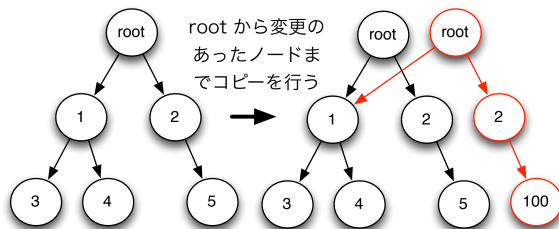

図 1 非破壊的木構造の編集

3. Index

Jungle は、非破壊的木構造というデータ構造上、過去の版の木構造を全て保持している。よって、すべての版に独立した Index が必要となるため、前の版の Index を破壊すること無く、Index を更新する必要がある。既存の TreeMap では、一度 Index の複製を行い、その後更新する必要があったため、Index の更新オーダーが O(n) となっていた。その問題を解決するため、Java 上で関数型プログラミングを行えるライブラリである、 Functional Java の TreeMap を使用し、それを用いて Index 実装を行なった。この TreeMap は、 Jungle と同じようにルートから変更を加えたノードまでの経路の複製を行い、データの更新を行なった際、前の版と最大限データを共有した新しい TreeMap を作成する。Jungle との違いは、木の回転処理が入ることである。これにより複数の版す

†1 琉球大学大学院理工学研究科情報工学専攻
Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University of the Ryukyus.
†2 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.

べてに対応した Index をサポートすることが可能になった。

4. 非破壊 TreeMap

Jungle の Index は、Functional Java の非破壊 TreeMap を用いて実装を行なっている。しかし、Functional Java の TreeMap は、木の変更の手間が大きい、並列実行処理速度が落ちるなど、実用的な性能を持っていなかった。そのため、Jungle の性能も、TreeMap 部分がネックとなっていた。その問題を解決するため、 Jungle で使用する非破壊 TreeMap を作成した。TreeMap は、Red Black Tree のアルゴリズムを用いている。RedBlackTree とは二分木探索の一つで、以下の条件を満たした木のことである。

- 各ノードは赤か黒の色を持つ。
- ルートノードは黒色である。
- 全ての葉は黒色である。
- 赤いノードの子は黒色である。
- 全ての葉からルートノードまでのパスには、同じ個数の黒いノードがある。

5. Index の差分 Update の実装

Jungle は木の編集を行う際に、編集を行うノードと、経路にあるノードの複製を行い新しい木構造を構築するため、 Index の中には、編集後の木には存在しない複製前のノードが残ってしまう。なので、Index の差分 Update を行う際には、それらのノードを Index から削除して、新しく複製されたノードを Index に登録する必要がある。 そのためには、編集を行なったノードを覚えておく必要がある。そこで、Jungle Tree Editor 内に、編集を加えたノードを覚えておくためのリストを定義した。Editor は木に編集を加えた際、リストに編集前のノードを保存する。そして、Commit 時にリストにあるノードを使って Index の中に残っている、編集後の木に存在しないノードを削除する。その後、新しく作られたノードを Index に登録して Update は終了する。

編集前ののノードの削除

ndex の Update を行う際、初めに、編集後の木に存在しないノードを Index から削除する。削除の対象は、変更を加えたノードと、ルートから変更を加えたノードまでの経路にあるノードである。ノードの削除は、以下の手順で行われる。

- 編集を行なったノードのリストからノードを取得する。
- 取得したノードが、保持している値を Index から削除する。
- 自身と子供のペアを Parent Index から削除する。
- ParentIndex から親を取得する。
- 2 - 4 をルートノードにたどり着くか、 ParentIndex から親を取得できなｋなるまで続ける。
- 1 - 5 をリストからノードが無くなるまで続ける。

Parent Index に現在のノードが登録されていない場合は、現在のノードからルートまでの経路にあるノードは Index から削除されているため、削除を終えて、リストに入っている次のノードの削除処理を行なっても構わない。

Index へのノードの挿入

Index から不要なノードを削除した後は、木の編集時新しく作られたノードを Index に挿入する。ノードの挿入は、以下の手順で行われる。

- 木からルートノードを取得する
- 取得したノードが Index に登録されているかを調べる。
- 登録されている場合、そのノード以下の Sub tree は、全て Index に登録されているので、次のノードに移動する。
- 登録されていなかった場合、自身が保持している値を Index に登録する。
- 自身と子ノードを Parent Index に登録する。
- 自身の子ノードを取得したノードとして 2 に戻る。
- 全てのノードを登録したら終了する。

6. Differential Jungle Tree の実装

Jungle は木の編集時、ルートから編集を行う位置までのノードの複製を行う。そのため、木の編集の手間は、木構造の形によって異なる。特に線形の木は、全てのノードの複製を行うため、変更の手間が O(n) になってしまう。そこで、Jungle は、線形の木を O(1) で変更する PushPop の機能を持つ。PushPop とは、ルートノードの上に新しいルートノードを付け加える API である (図??)。すると、木の複製を行う必要が無いため、木の変更の手間が O(1) でノードの追加を行える。 しかし、PushPop はルートノードを追加

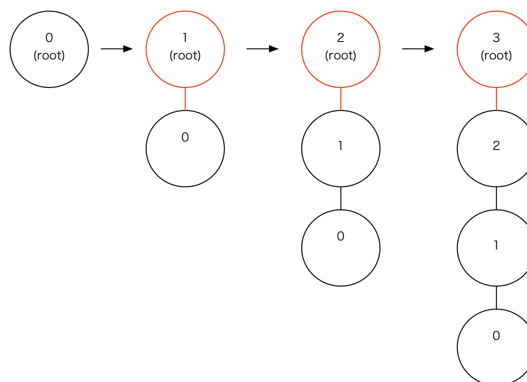

図 2 PushPop

していくため、図?? のようにノードの並びが逆順に

なってしまう。Log などの正順の木でなければデータを表現できない場合、木の編集時 PushPop を使用できない。そこで、木の編集の手間を O(1) で、正順の木を構築できる Differential Jungle Tree の実装を行なった。 Differential Jungle Tree は。木のバージョンごとに、自身の木の最後尾を表す末尾ノードを保持する。木の編集は、別途構築した Sub Tree に対して破壊的に更新を行い、Commit 時に末尾のノードに Sub Tree を Append することで行う。この場合は木が破壊的に変更されているように見えるが、前の版の末端部分を超えてアクセスすることがなければ複数の版を同時に使用することができる。

Differential Tree Context

Jungle の木は TreeContext というオブジェクトに自身の木の情報を保持している。Differential Jungle Tree では、現在の版の木構造の末尾ノードを保持することが可能な Differential Tree Context を作成した。

Differential Jungle Tree の作成

Differential Jungle Tree を作成するために Jungle に、新しい API を実装した。(表**??**)

createNewDifferenceTree(StringtreeName)	TJungle に新しくDifferential Jungle Tree を作成する。木の名前が重複した場合、生成に失敗し

表 1　Jungle に新しく実装した API

末尾ノードを使用した木の編集

Differential Jungle Tree の木の編集は、Differential Jungle Tree Editor を使用して行う。Differential Jungle Tree Editor は、Default Jungle Tree Editor と違い、生成時に新しい木構造 (Sub Tree) を自身の中に構築する。そして、木の編集は、Editor が保持している木構造に対して行う。編集後、Commit を行う際に構築した木構造を、 Differential Jungle Tree の末尾ノードに Append する。その際木の複製は行わない。　また、Differential Tree は自身が保持している木構造に対する変更しか行えないため、一度 Commit した木に対して変更は行えない。図**??**

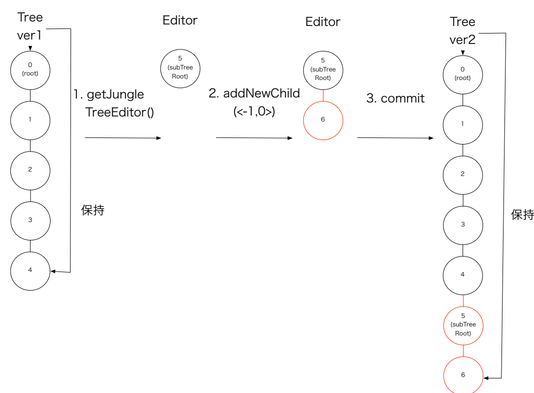

図 3　末尾ノードを使用した木の編集

- 木から getJungleTreeEditor で Editor を取得する。（このとき Editor は新しい木構造 (Sub Tree) を持つ）。
- Editor が保持している木構造に対して addNewChild(¡-1.0¿) を実行し、ノードの追加を行う。
- Commit を行い、Tree の末尾ノードに Editor が保持している木構造を Append する。

Editor が保持している木構造に最後に追加したノードが、新しい木の末尾ノードとなる。また、Differential Jungle Tree は、木の編集時複製を行わないため、Index のアップデートは、 Editor が保持している木構造のデータを Index に追加するだけで良い。

Differential Jungle Tree の検索

Differential Jungle Tree は、末尾ノードを使って、現在の木構造を表現している。なので、過去の木に対して全探索を行なった場合、その版には無いはずのノードが取得できてしまう。例として、編集前の木である Tree ver1 と編集後の木である Tree ver2 があるとする (図**??**)。ここで、Tree ver1 に対して、全探索を行なった場合、本来 Tree ver1 に存在しないノード 3・4 も検索対象に含まれてしまう。そこで、その版の木が持つ末尾ノード以下の Sub Tree を検索対象から除外する、Differential Interface Traverser を実装した。Differential Interface Traverser を用いて木の全探索を行なった場合、Tree ver1 に存在しないノード３・４は検索対象から省かれる。　Index を使用

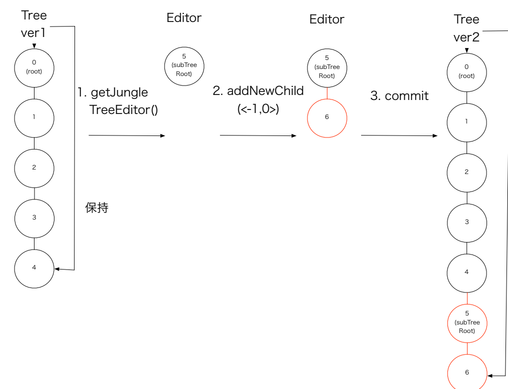

図 4　複数の版の木の表現

した検索を行う場合、各版の木に対応した Index があるため、Default Tree と検索のアルゴリズムは変わらない。これらの実装により Differential Jungle Tree は木構造の構築・検索を行う。

Differential Jungle Tree の検索

Differential Jungle Tree への Commit は、編集後の木のデータを持つ TreeContext を作り、編集前の木が持つ TreeContext と Atomic に入れ替えることで行われる。しかし、Differential Jungle Tree の Commit は、Default Jungle Tree の Commit と