ゲームエンジンにおける木構造データベースjungleの提案

135768K 氏名 武田和馬 指導教員：河野 真治

Abstract

There are some problems to Relational Databases and NoSQLs. One of them as Impedance mismatch. This problem is cased by a difference between the first normal form of database and the programs. There is a problem that NoSQL is unfit for parallel processing.

1 ゲームエンジンにおけるデータベース

Relational Database(RDB) は、列と行からなる2次元のテーブルにより実装されるデータベースである。データ型として文字列、数値、日付、Bool 型がある。RDB はスキーマの決まったデータを扱うことを長所としている。

RDB ではプログラムとデータベースとの間にミスマッチが発生する。プログラムではリストやネスト構造によりデータを持つことができる。しかし、データのネスト構造を許さない第一正規形を要求する RDB とは相容れない。これをインピーダンスミスマッチという。

この例として、ゲーム中のユーザが持つアイテムという単純なものでは RDB ではユーザとアイテムの組をキーとする巨大な表として管理する。

インピーダンスミスマッチの解決方法として ORMapper が挙げられる。これはデータベースのレコードをプログラム中のオブジェクトにマッピングし扱うことができる。オブジェクトに対する操作を行うと ORMapper が SQL を発行し、処理を行ってくれる。しかし、レコードをプログラム中のオブジェクトを対応させる ORMapper の技術でインピーダンスミスマッチの本質的な部分を解決することはできない。

NoSQL は Not Only SQL の略である。

通常 NoSQL データベースは非リレーショナル型であり、スキームの定義がない [1]。そのため、扱うデータの型が決まっていなくても気軽に扱える。

しかし、トランザクションとして Json の一括といった形で処理されている。そのため並列処理を必要とするアプリケーションには向かない。

2 Jungle Database の提案

この章の前半では RDB と NoSQL の利点と問題点を取り上げた。

非破壊的木構造データベースの Jungle を提案している [2]。Jungle はスケーラビリティのあるデータベースとして開発している。

ウェブサイトの構造は大体が木構造であるため、Jungle ではデータ構造として木構造を採用している。しかし、ウェブサイトだけでなくゲームにおいてもデータ構造が木構造になっている。

そこで、本研究では Jungle の木構造である特性を活かし、ゲームエンジン Unity で作成したゲームで使用する方法を提案する。

データベースとして Jungle Database を採用する。

Jungle は Java と Haskell によりそれぞれの言語で開発されている。本研究で扱うのは Java 版を C#で再実装したものである。

3 Jungle-Sharp の再実装

Jungle はもともと Java と Haskell で書かれていた。今回は Java 版をベースに C#で再実装する。エラーをチェックする Either の部分だけは Haskell の要素を取ってくる。

Jungle ではデータの編集を行った後、Either を用いてエラーのチェックを行う。エラーがあればエラーが包まれた Either が返される。エラーがない場合は指定した型のオブジェクトが Either に包まれて返される。

これは関数型プログラミング言語、Haskell から採用したものである。

編集を行うたび、Either のチェック bind で行うことにより、より関数型プログラミングに特化した書き方が可能になる。C#で実装した bind は以下に記述する。

DefaultEither.cs

```
public Either<A, B> bind (System.Func<B, Either<A, B>> f) {
    if (this.isA ()) {
      return this;
    }

    return f (this.b ());
}
```

bind での Either をチェックしつつデータを格納する例を以下に記述する。

DataSaveTest.cs

```
Item   apple = new Item("Apple");

either = either.bind ((JungleTreeEditor arg) => {
  return arg.addNewChildAt (rootPath, 0);
});

either = either.bind ((JungleTreeEditor arg) => {
  return arg.putAttribute (apple);
});
```

bind の実装により、ユーザ側で Either の Error チェックを行う必要がなくなる。

4 Unity で実装したアプリケーション

本論文では C#で再実装を行った Jungle を Unity で作られたゲームの上に構築する。例題のゲームとしては図 1 に記載した、マインクラフト [3] の簡易版を作成する。

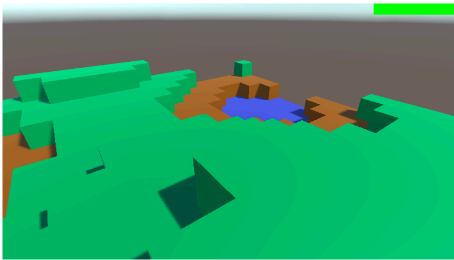

図 1: craft

プレイヤーは自由にマップを移動し、ステージの破壊や、生成を行うことができる。破壊や生成のオペレーションに合わせて Jungle のノードにも同期する。この同期も非破壊で行われる。

5 データの設計

Unity におけるゲームの構成は Object の親子関係、つまり木構造である。同じく Jungle Database は木構造である。

Jungle では複数の木を持つことができる。ゲームのシーンを構成する GameTree とアイテムを管理する ItemTree を Jungle 内に作る。

GameTree ではシーン内にある Player や Stage を構成する Cube などを格納している。図 2 では Jungle に格納する構造を示したものである。

ItemTree では Item の情報が格納されている。ItemTree はマスターデータとしている [4][5]。マスターデータとは、アイテムの名前や敵の出現確率などを示す。ゲーム開発者のみが更新できる。

図 3 では Jungle に格納している Item の構造を示したものである。

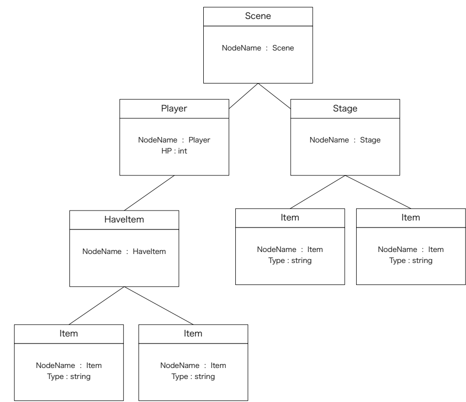

図 2: GameTree

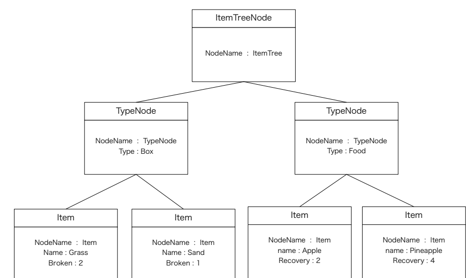

図 3: ItemTree

6 ゲームに特化したデータベース

C#の再実装を行った際に Java の Jungle に沿ってデータの型、つまり ByteArray で設計を行っていた。データの格納を行うたびに Byte Array へのキャストを行う必要がある。しかし、キャストの処理は軽くはない。

そこで、シーンを構成する Object をそのまま格納するに仕様を変更した。C#では Object クラスのエイリアスとして object 型が使える。

object 型を使うことによってユーザーが定義した任意の変数を代入することができる。以下にその使用例を記述する。

SaveData.cs

```
Player player = new Player ();
either = either.bind ((JungleTreeEditor arg) => {
  return arg.putAttribute ("Player", player);
});

Enemy enemy = new Enemy ();
either = either.bind ((JungleTreeEditor arg) => {
  return arg.putAttribute ("Enemy", enemy);
});
```

データを取り出すには Generic で型を指定する、もしくは as 演算子を用いてキャストを行う。以下に取り出す例を記述する。

SaveData.cs

```
Player player = attr.get<Player> ("Player");
Enemy enemy = attr.get ("Enemy") as Enemy;
```

データの型の再設計を行ったことによりシーン内のオブジェクトをそのまま格納が可能になった。格納の際に Byte

Array に変換する必要がない。

分散構造や、ネットワークで必要な時だけ変換する。

7　Jungle-Sharp の評価

本論文では Java で書かれた Jungle Database を C#で再実装した。同じオペレーションで Java と C#で計測を行った。なお、1 回目の処理はキャッシュを作り処理が遅くなるため、計測は行わず、2 回目以降から行う。計測時に使用したデータ挿入のオペレーションを以下に記述する。

BenchMarkmark.cs

```
for (int i = 0; i < trial; i++) {
  Either<Error, JungleTreeEditor> either = edt.addNewChildAt (path, i);
  either = either.bind ((JungleTreeEditor arg) => {
    return arg.putAttribute ("name", "Kazuma");
  });
}
```

計測に使用したマシンの環境を記述する。

表 1: 計測環境

OS	Mac OS Sierra 10.12.3
Memory	8 GB 2133 MHz LPDDR3
CPU	2.9 GHz Intel Core i5
Java	1.8.0111
.NET Runtimes (MonoDevelop-Unity)	Mono 4.0.5
.NET Runtimes (Xamarin)	Mono 4.6.2

計測結果を図 4 に示す。

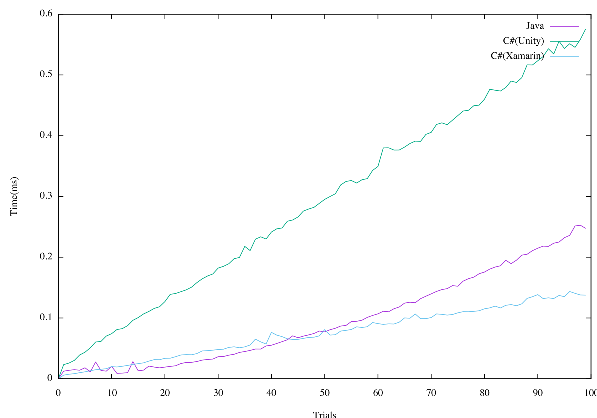

図 4: BenchMark

図 4 より、Unity で実行した結果では O(n) のグラフを示している。Unity ではレンダリングの機能も兼ねている。そのためプログラムを実行している間もレンダリングを行っているため、純粋な PutAttribute の計算時間ではないと考えられる。

そこで、純粋な速度を測定するため Xamarin で動かし測定した。C#で再実装した Jungle は Java 版とほぼ同じ計算量を示している。これにより、本来の Java と同じ、もしくはそれ以上のパフォーマンスを引き出すことができる。

8　まとめ

本研究では JungleDatabase を C#で再実装を行った。Java と C#は比較的似ている言語であるため移行は難しくはなかった。

性能としても Java 版に劣らない、もしくはそれ以上のパフォーマンスを出せる。

Either での bind の実装で、より関数型プログラミングを意識しながら記述することができる。これは Java 版にはない実装である。

Jungle Database はもともと Web 向けに作られたデータベースである。

Web ではリニアにデータが書き換わることは多くない。しかしゲームでは扱うデータが多くリニアに書き換わる。

そのため、Jungle の構成は保ちつつ、ゲームに合わせた Jungle の拡張を行った。

データの格納の際に ByteBuffer であったものを Object 型に変更した。これにより、シーンを構成する Object を手間なく格納することを可能にした。

Jungle は非破壊であるため、過去の変更を持っている。

ゲームにおいて過去の木を持ち続けることはパフォーマンスの低下につながる。そのため、過去の木をどこまで必要かを検討しなければならない。

現在 C#版の Jungle にはデータを永続化させる仕組みは備わっていない。実用的なゲームのデータベースとして使うためには永続化を実装する必要がある。

参考文献

[1] PETTER NASHOLM. Extracting data from nosql databases, jan 2012.

[2] Shinji Kono Shoshi Tamaki, Seiyu Tani. Cassandra を使ったスケーラビリティのある cms の設計, 2011.

[3] MicroSoft. https://minecraft.net/ja-jp/.

[4] Hitonishi Masaki. ゲームエンジニアのためのデータベース設計. http://www.slideshare.net/sairoutine/ss-62485460.

[5] Ryosuke Iwanaga. ソーシャルゲームのための mysql 入門.