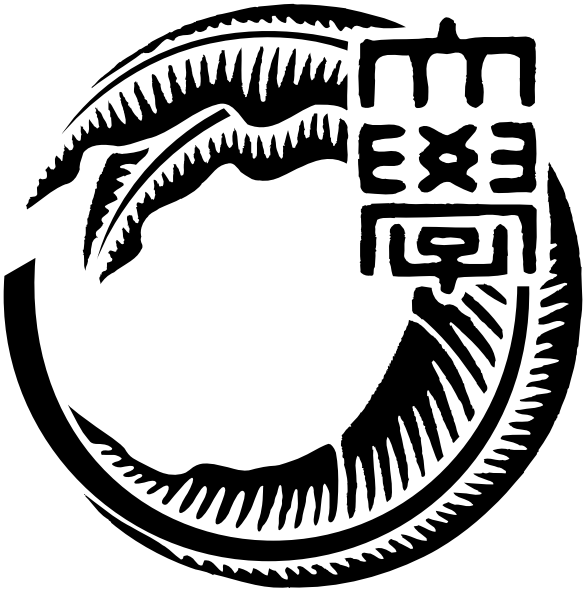
平成28年度 卒業論文

ゲームエンジンにおける木構造データベースJungleの提案



琉球大学工学部情報工学科

135768K 武田 和馬

指導教員 河野 真治

目 次

図 目 次

表 目 次

第1章　ゲームエンジンにおけるデータベース

この章ではデータベースの種類である Relational Database と NoSQL について述べる。

1.1　Relational Database

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 は主として使われているデータベースであるが、苦手としている事がある。

それは、スキーマレスなデータの扱いやマシン台数を増やし処理速度を上げることである。水平分割や垂直分割によりデータを分割できるが構造としては複雑化していく。

1.2　NoSQL

NoSQL は Not Only SQL の略で、SQL を必要としない非リレーショナル型のデータベースである。

Json や XML を扱えるデータベースでは通常スキームを必要としないため、特に扱うデータの型が決まっていなくても使うことができる。

しかし、不定形の構造の変更をトランザクションとして、Json の一括変更という形で処理されてしまっており、並列アプリケーションには向いていない。

1.3　Jungle の提案

非破壊的木構造データベースの Jungle を提案している [?]。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、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。

本研究では Jungle を Unity を用い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。

第2章　Jungle Database の概念

本章ではまずは Jungle Database の概念と構造について記述する。

2.1　木構造データベース Jungle

当研究室で開発している Jungle は過去の木を保存しつつ、新しい木を構成する手法を採用している。これを非破壊的木構造という。非破壊的木構造により、データベースを参照する側と更新する側のデータを安全に扱うことができる。

JungleDatabase は木の集合からなり、名前で管理される。木はノードの集合から出来ている。ノードには Key と Value の組からなるデータを持つことができる。これはデータベースのレコードに相当する。

通常のデータベースと違う点として子のノードを持つことである。

Jungle は、データの変更を一度生成した木を上書きせず、ルートから編集を行うノードまでのコピーを行い、新しく木構造を構築し、そのルートをアトミックに入れ替える図 2.1。これを非破壊的木構造と呼ぶ。非破壊木構造は新しい木を構築している時にも、現在の木を安全に読み出せるという大きな特徴がある。しかし、書き込みの手間は大きくなる。

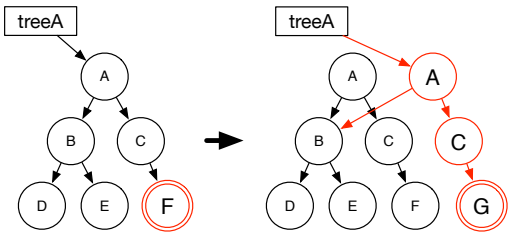


図 2.1: 非破壊的木構造の木の編集

2.2　Jungle Database の構造

非破壊木構造を採用している Jungle では、木の変更の手間は O(1) から O(n) となり得る。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充分な性能を出すことは出来ない。逆に木の設計を行えば高速な処理が可能である。

Jungle はオンメモリ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もよい。

Jungle は commit log を持ち、それを他のノードやディスクに転送することにより、分散構成と持続性を実現する。

第3章　JungleDatabaseのAPI

本章ではJungleDatabaseのAPIを記述する。

3.1　Jungleの木

Jungleは複数の木の名前を利用し、管理しており、名前により生成、編集を行う。以下にJungleクラスが提供している木の生成、管理を行うAPI(表3.1)に記述する。

表 3.1: Jungleに実装されているAPI

`JungleTree createNewTree(string treeName)`	Jungleに新しく木を生成する。木の名前が重複した場合、生成に失敗しnullを返す。
`JungleTree getTreeByName(string treeName)`	Jungleからと名前が一致するtreeを取得する。名前が一致するTreeがない場合取得は失敗しnullを返す

3.2　TreeNode

Jungleが保有する木は、複数のノードの集合で出来ている。ノードは、自身の子のList、属性名と属性値の組のデータを持つ。ノードに対するアクセスは表3.2に記述されているAPIを用いて行う。

表 3.2: TreeNodeに実装されているAPI

`Children getChildren()`	ノードの子供を扱うChildrenオブジェクトを返す。
`Attribute getAttribute()`	ノードが保持しているデータを扱うAttribteオブジェクトを返す。

3.3 Either

jungle では例外処理を投げる時に Either クラスを用いて行う。返って来た Either のオブジェクトに対して、`isA()` で `Error` かどうかをチェックする。`Error` でない場合は `b()` で対象のオブジェクトを取り出す事ができる。

以下にルートノードの 2 番目の子どもを取ってくるの Either のサンプルコードを記述する。

```
1 Either<Error,TreeNode> either = children.at(2);
2 if (either.isA())
3     return either.a();
4 TreeNode child = either.b();
```

3.4 Children と Attribute

Children クラスへのアクセスは表 3.3 に記述されている API を、Attribute クラスへアクセスは表 3.4 に記述されている API を用いて行う。

表 3.3: Children に実装されている API

`int size()`	子供の数を返す。
`<Either Error,TreeNode> at(int num)`	ノードが持つ子供の中から、 変数 `num` で指定された位置にある子ノードを返す。

表 3.4: Attribute に実装されている API

`T get<T>(string key)`	ノードが持つ値から、属性名 `key` とペアの属性値を `Generic` 型で返す。
`string getString(string key)`	ノードが持つ値から、属性名 `key` とペアの属性値を `string` 型で返す。

3.5 NodePath

Jungle では、木のノードの位置を `NodePath` クラスを使って表す。`NodePath` クラスはルートノードからスタートし、対象のノードまでの経路を、数字を用いて指し示すことで対象のノードの場所を表す。また、ルートノードは例外として-1 と表記される。`NodePath` クラスが`< -1,1,2,3>` を表している際の例を図 3.1 に記す。

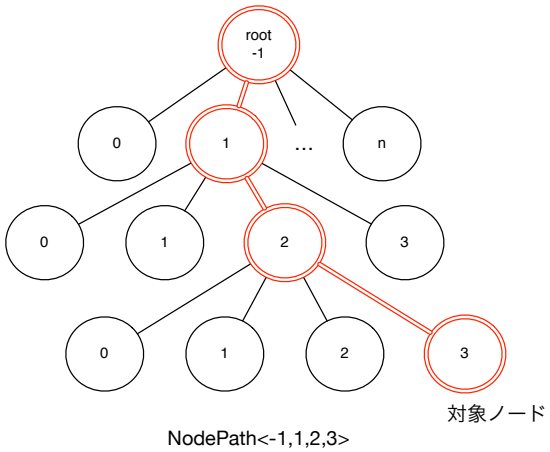


図 3.1: NodePath

3.6 木の編集

Jungle の木の編集は `JungleTreeEditor` クラスを用いて行われる。`JungleTreeEditor` クラスには編集を行うために、表 3.5 に記述されている API を用いて行う。

編集を行った後は、関数 `editor.commit()` で今までの編集をコミットすることができる。他の `JungleTreeEditor` クラスによって木が更新されていた場合はコミットは失敗し、`commit()` は `Error` を返す。その場合は、木の編集を最初からやり直す必要がある。

表 3.5: Editor に実装されている API

Either<Error, JungleTreeEditor> addNewChildAt(NodePath path, int pos)	変数 path で指定した場所にある、ノードの子供の変数 pos で指定した位置子ノードを追加する
Either<Error, JungleTreeEditor> deleteChildAt(NodePath path, int pos)	変数 path で指定した場所にある、ノードの子供の変数 pos で指定した位置の子ノードを削除する。
Either<Error, JungleTreeEditor> putAttribute(NodePath path, string key, byte[] value)	変数 path で指定した場所にあるノードに、属性名 変数 key 属性値 変数 value のペアで値を挿入する。
Either< Error, JungleTreeEditor> deleteAttribute(NodePath path, string key)	変数 path で指定した場所にあるノードが持つ、属性名 変数 key とペアで保存されているデータを削除する。
Either<Error, JungleTreeEditor> commit()	木へ行った変更をコミットする。自分が編集を行っていた間に、他の JungleTreeEditor クラスによって木が更新されていた場合、コミットは失敗する。

第4章　Unityでのデータベースの取扱い

この章ではゲームにおけるデータについて述べ、その後 Jungle との関係性や実装を記述する。

4.1　ゲームのデータ

ゲーム中のデータには幾つか考えられる。

シーンのオブジェクトが持つパラメータゲームのセーブデータネットワーク上で使用するデータ

4.2　ゲームデータとデータベース

ゲームのデータベースとして長年使われているのが RDB である。

プログラム中のデータ構造と RDB の表構造のズレによりインピーダンスミスマッチという問題がある。

例えば RPG ゲーム中のユーザが持つアイテムという単純なものでも、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ことになる。プログラム中では、ユーザが持つアイテムリストという簡単な構造を持つが、第一正規形を要求する RDB ではネスト構造を許さない。

ORMapper で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。

4.3　Unity と Jungle の関係

Unity は 3D ゲームエンジンで、ゲームを構成する要素 (Object) を C#で制御する。Object は一つのゲームのシーン (一画面の状況) の中で木構造を持つ。これをシーングラフと言う。シーングラフをそのまま Jungle に格納するという手法が考えられる。

4.4 Unityにおけるデータベース

Unityでのデータベースとして考えられるものとしてはSQLite3、PlayerPrefsが挙げられる。

PlayerPrefsとは、Unityに特化したバイナリ形式でKeyとValueのみで保存されるものである。セーブ機能に特化していてメモリ上にDBを展開するものではない。

SQLite3ではC#で利用できるORMapperが提供されている。プログラム中からデータのインサートやデリートを行う。

Unity5.3以降のバージョンでは標準でJsonが扱えるようになった。これにより、インスタンスをJson化することができる。しかし、変数名をKeyとしてValueを取り出すといったことは出来ない。

第5章　Jungle-Sharpの実装

Java と C#はよく似た言語であり、移行はそれほど難しくない。Jungle の中心部分である木構造と Index を構成する赤黒木のコードはほぼ変更なく移行できた。C#ではインナークラスが使えないので明示的なクラスに変換する必要があった。

5.1　AtomicRefarence の実装

Jungle の木の変更 (commit) は CAS(Check and Set) を用いて atomic に行われる。競合している書き込み中に自分の書き込みが成功した場合に関数 `success()` が成功する。

Java では AtomicRefarence が標準であるが C#にはなかったため AtomicRefarence のクラスを新たにつくった。

AtomicRefarence.cs

```
// C#
public bool CompareAndSet(T newValue, T prevValue) {
    T oldValue = value;
    return (oldValue
        != Interlocked.CompareExchange
                  (ref value, newValue, prevValue));
}

// Java
AtomicRefarence<T> atomic = new AtomicRefarence<T>();
atomic.compareAndSet(prevValue, newValue);
```

5.2　List の実装

木やリストをたどる時に Java では Iterator を用いる。Iterator は次の値があるかを返す boolean hasNext() と、T という型の次の値を取ってくる T next() を持つ Object である。C#では木やリストを辿りながら yeild で次の値を返す。Java では以下のように実装されている。

List.java

```
public Iterator<T> iterator() {
  return new Iterator<T>() {
    Node<T> currentNode = head.getNext();

    @Override
    public boolean hasNext() {
      return currentNode.getAttribute()
                                    != null;
    }

    @Override
    public T next() {
      T attribute
            = currentNode.getAttribute();
      currentNode
            = currentNode.getNext();
      return attribute;
    }
  };
}
```

C#ではIEnumeratorがあるのでそれを利用した。Listのforeachではを呼び出すために、一つずつ要素を返す必要がある。yield returnステートメントを利用することで位置が保持され、次に呼ばれた際に続きから値の取り出しが可能になる。以下にその実装例を示す。

List.cs

```
public IEnumerator<T> iterator() {
  Node<T> currentNode = head.getNext();
  while (currentNode.getAttribute() != null) {
    yield return (T)currentNode.getAttribute();
    currentNode = currentNode.getNext ();
  }
}
```

5.3 bindの実装

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 ());
}
```

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

```
System.Collection.Generic.List<BoxItemInfo> infoList = new System.Collection.Generic.List<BoxItemInfo> ();
    infoList.Add (new BoxItemInfo (1, 2, "Grass", "#019540FF"));
    infoList.Add (new BoxItemInfo (2, 4, "Wood", "#7F3C01FF"));
    infoList.Add (new BoxItemInfo (3, 1, "Sand", "#D4500EFF"));
    infoList.Add (new BoxItemInfo (4, 5, "Water", "#2432ADFF"));

    foreach (var info in infoList.Select((v, i) => new {v, i})) {
      either = either.bind ((JungleTreeEditor arg) => {
        return arg.addNewChildAt (path, info.i);
      });

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

bind の実装により、ユーザ側が Error のチェックする必要がなくなる。

第6章　Unityで実装したアプリケーション

本章では Unity で実際に作成したアプリケーションを示し、どのようにデータの設計を行ったかを述べる。

6.1　例題ゲーム

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 6.1 に記載した、マインクラフトの簡易版を作成する。

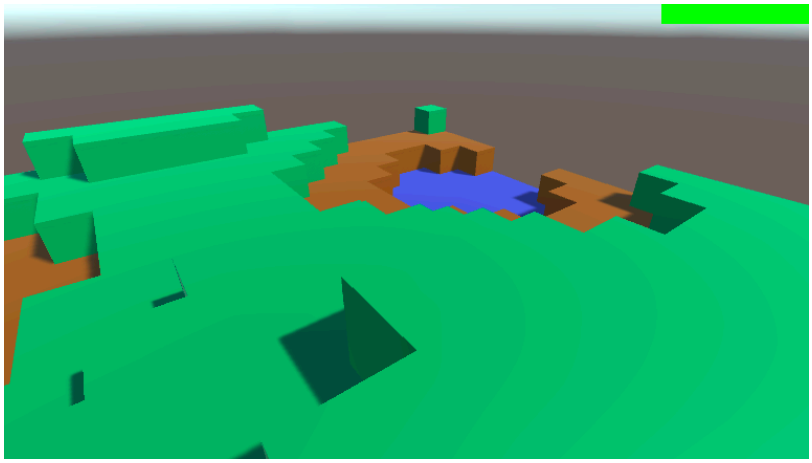


図 6.1: craft

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。

6.2　ゲームの要素

例題ゲームを構成するゲームの要素を記述する。

Unity ではオブジェクトに対してコンポーネントが紐付けられる。クラスも同様にコンポーネントして扱える。

ステージを構成するブロックのコンポーネントして、ItemBox クラスを紐付ける。ItemBox の持つ変数を表 6.1 に示す。

表 6.1: ItemBox クラスが持つ Attribute

`Broken`	Item の体力、0 になると自身の ItemBox のオブジェクトを破壊
`ColorCode`	自身のブロックの色

プレーヤーにはコンポーネントとして Player クラスを紐付ける。Player クラスの持つ変数を表 6.2 に示す。

表 6.2: Player クラスが持つ Attribute

`HP`	プレイヤーの体力、0 になるとゲームオーバー
`ItemList`	プレイヤーが持つアイテムのリスト

6.3 データ設計

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。Jungle Database は木構造型のデータベースであるので、そのまま格納するという手法が考えられる。

図 6.2 では Jungle に格納する構造を示したものである。

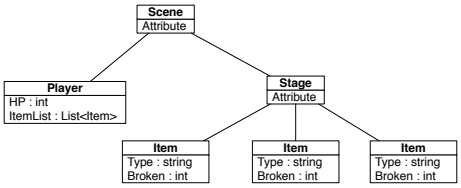


図 6.2: GameTree

6.4 Attributeの格納するデータの型

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Byte Array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。

第7章　ベンチマーク

本章では C#と Java の Jungle との比較、Jungle-Sharp と Unity 上で使われる SQLite3、PlayerPrefs との比較を行う。

7.1　Javaとの比較

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#で計測する。オペレーションは以下に記述する。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。

Benchmark.cs

```
public JungleTreeEditor createTree(JungleTreeEditor editor, int _curY, int _maxHeight, NodePath path) {

    if (_curY == _maxHeight) {
      return editor;
    }

    for (int i = 0; i < 3; i++) {
      Either<Error, JungleTreeEditor> either = editor.addNewChildAt (path, _curY);
      DebugCommon.Assert (either.isA (), "Error");
      editor = either.b ();
      string value = path.add (_curY).ToString ();
      either = editor.putAttribute (path.add (_curY), key, System.Text.Encoding.ASCII.GetBytes (value));
      DebugCommon.Assert (either.isA (), "Error");
      editor = either.b ();
      string value2 = value + "+ index";
      either = editor.putAttribute (path.add (_curY), indexKey, System.Text.Encoding.ASCII.GetBytes (value2));
      DebugCommon.Assert (either.isA (), "Error");
      editor = either.b ();
      editor = createTree (editor, _curY + 1, _maxHeight, path);
    }
    return editor;
  }
```

7.2　SQLite3とPlayerPrefsとの比較

Unity で使われているデータ保存として SQLite3 と PlayerPrefs がある。それぞれに対し、データの格納を行い、計測する。

第8章　結論

8.1　まとめ

本論文では 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。

Jungle はオンメモリで動作する。その為 SQLite3 や PlayerPrefs よりも速く動作する。

参考文献

[1] RICHARDSON, T., AND LEVINE, J. The remote framebuffer protocol. rfc 6143, mar 2011.

[2] TightVNC Software. `http://www.tightvnc.com`.

[3] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER,. A. virtual network computing, jan 1998.

[4] LOUP GAILLY, J., AND ADLER, M. zlib: A massively spiffy yet delicately unobtrusive compression library. `http://zlib.net`.

[5] Surendar Chandra, Jacob T. Biehl, John Boreczky, Scott Carter, Lawrence A. Rowe. Understanding screen contents for building a high performance, real time screen sharing system. *ACM Multimedia*, Oct 2012.

[6] Yu TANINARI and Nobuyasu OSHIRO and Shinji KONO. Vnc を用いた授業用画面共有システムの実装と設計. 日本ソフトウェア科学会第 28 回大会論文集, sep 2011.

[7] Yu TANINARI and Nobuyasu OSHIRO and Shinji KONO. Vnc を用いた授業用画面共有システムの設計・開発. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), may 2012.

[8] Tatsuki IHA and Shinji KONO. 有線 lan 上の pc 画面配信システム treevnc の改良. 第 57 回 プログラミング・シンポジウム, jan 2016.

[9] Nozomi TERUYA and Shinji KONO. 分散フレームワーク alice の pc 画面配信システムへの応用. 第 57 回 プログラミング・シンポジウム, jan 2016.

謝辞

本研究の遂行，また本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜わりました河野真治准教授に深く感謝致します。

数々の貴重な御助言と細かな御配慮を戴いた金川 竜己さん、比嘉健太さん、伊波立樹さん、並びに並列信頼研究室の皆様に深く感謝致します。

最後に、有意義な時間を共に過ごした情報工学科の学友、並びに物心両面で支えてくれた両親に深く感謝致します。

2017 年 3 月
武田和馬