平成29年度 卒業論文

Proposal of tree structured database Jungle in Game Engine.

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。次に分散システムにおいて重要な CAP 定理に触れる。

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。

テーブルを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。

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。これをインピーダンスミスマッチという。

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。

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。

1.2　ACID トランザクション

ACID(Atomicity,Consistency,Isolation,Durability) はデータベースのトランザクションの処理が確実に実行されることを保証するものである [1]。

ほとんどの RDB では ACID トランザクションを保証している。

- Atomicity(原子性)

  トランザクションを実行する際に、すべて成功するか、すべて失敗するか。

- Consistency(一貫性)

  トランザクション開始時と終了時にデータが一貫していなければならない。

- Isolation(独立性)

  他のトランザクションによる干渉を受けない。

- Durability(永続性)

  コミットしたトランザクションのデータは保存される。

1.3 NoSQL

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 [2]。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。

1.4 CAP 定理

分散システムにおいて、次の 3 つを同時に保証することは出来ない。

- Consistency(一貫性)

  すべてのノードはクエリが同じならば同じデータを返す。

- Availability(可用性)

  あるノードに障害が発生しても、機能しているノードにより常にデータの読み書きが行える。

- Partition-tolerance(分断耐性)

  ネットワーク障害によりノードの接続が切れてもデータベースは機能し続けることができる。

これは CAP 定理 [3] と呼ばれる。データベースを利用する場合は CAP 定理を意識しながら選択する。

一貫性と可用性を重視したデータベースが RDB である。分断耐性を必要とする場合は NoSQL データベースとなる。

NoSQL データベースでは一貫性を取るか、可用性を取るかによって選択するデータベースが変わる。以下にその 2 つの例を示す。

1.5 MongoDB

MongoDB は 2009 年に公開された NoSQL のデータベースである。ドキュメント指向型とされ、事前にテーブルの構造を決めておく必要がない。これをスキーマレスという。

MongoDB はマスター/スレーブ方式の Replication を採用している。保存したデータ (マスター) を複数のサーバー (スレーブ) に複製を取る。

スレーブを Read させることによって負荷分散も可能になる。

また、一台のサーバーにすべてのデータを持たず、複数のサーバーに分割して保持する。これを Sharding という。

MongoDB は Replocation と Sharding により、分断耐性と一貫性を持つ。

1.6 Cassandra

Cassandra[4] は 2008 年に Facebook によって公開された Key-Value 型のデータベースである。Amazon の Dynamo[1] と Google の BigTable[5] を合わせた特徴を持っている。Key-Value であるため、スキーマレスな NoSQL となる。

Bigtable から採用した、カラムファミリーと呼ばれる構造を基本としている。

カラムファミリーの行の部分は HashMap や連想配列のように Key-Value で複数格納している。

1 つの Key-Value の組をカラムと呼ぶ。RDB とは異なり、カラム名を事前に定義する必要がない。

Cassandra ではサーバーノードの配置に Consistent hasing アルゴリズムを用いる。また、これによりサーバー同士が理論上リング構造になっている。

Consistency Hashing によるリングの形成を図 1.1 に示す.

Cassandra はデータを最大どれだけ配置するかを示す Replication factor と, データの読み書きをいくつのノードから行うのかを決める Consistency Level の設定が行える。Consistency Level には主に ONE, QUORAM, ALL がある。

Replication factor の数値を N とした場合, ONE は 1 つのノード, QUORUM は $N/2 + 1$ のノード, ALL は N のノードへと読み書きを行う。Replication factor と Consistentcy Level の設定により, Cassandra は最新のデータを取得したいときとそうでないときで読み込みと書き込みの速度をあげることができる。一貫性が重要なデータに関しては QUORUM により書き込み読み込みを行うことで常に最新のデータを取得することができる。多少データが古くてもよい場合は ONE などを使用することでレスポンスを早くすることができる。Consistency Level QUORUM の時のデータ書き込みと読み込みについて図 1.2 と図 1.3 に示す。Consistency ハッシング, Replication factor と Consistency Level の設定により Cassandra は高い可用性と分断耐性を持つ。

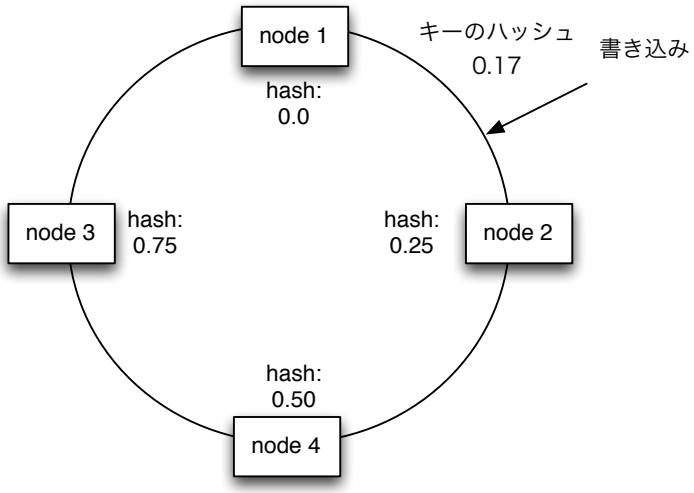

図 1.1: Consisteyncy hashing による ring 型トポロジーの形成

1.7 Jungle 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 [6]。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。

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。

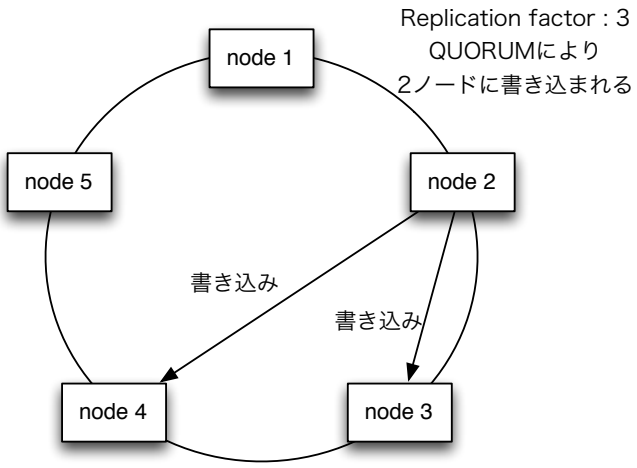

図 1.2: Consisteyncy Level QUORUM による書き込み

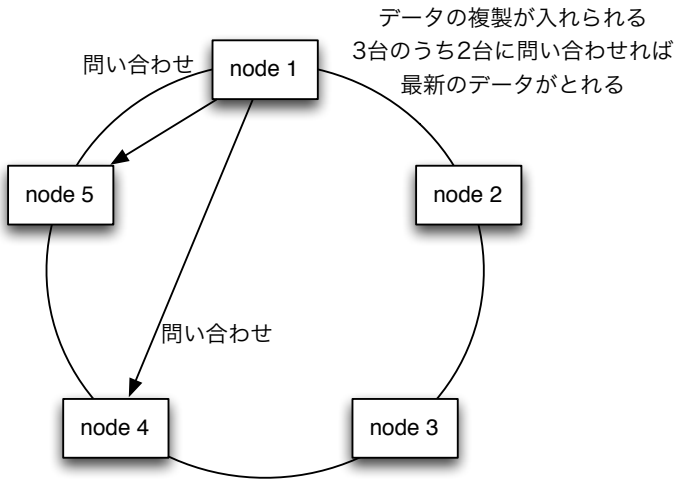

図 1.3: Consisteyncy Level QUORUM による読み込み

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。

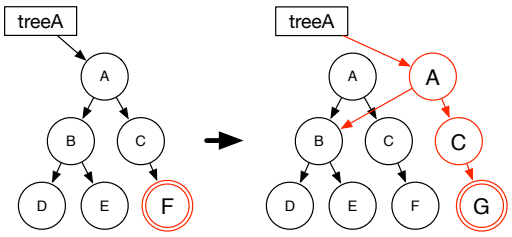

図 2.1: 非破壊的木構造の木の編集

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からの treeName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。

SaveData.cs

```
Either<Error,TreeNode> either = children.at(2);
if (either.isA())
    return either.a();
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。

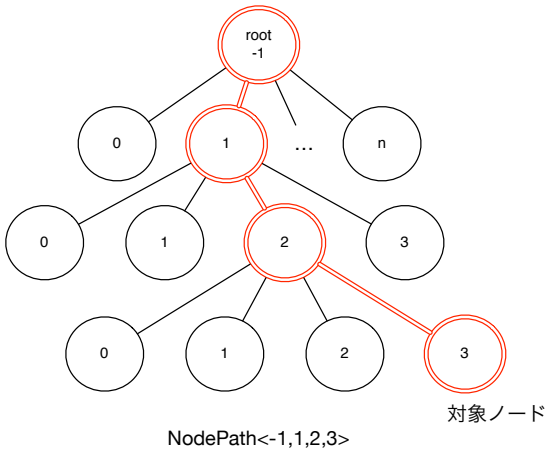

図 3.1: NodePath

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, object 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 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 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

ゲームを開発する際にデータの種類について知っておく必要がある。以下にゲームにおけるデータを記述する。

- オブジェクトが単体で持つデータ

  シーン内に存在するオブジェクトが持つパラメータ例えばプレイヤーの HP や経験値、位置座標などを示す。

- オブジェクト 1 つで複数持つデータ

  プレイヤーが持つアイテムデータなどを示す。

- マスタデータ (ReadOnly)[7][8]

  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。

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使われているのが RDB である。

1 章で述べたように RDB とプログラム間ではインピーダンスミスマッチという問題がある。

ここではゲームでのインピーダンスミスマッチの例を紹介する。

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。

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。

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`commit()`が成功する。失敗した場合ははじめからもう一度行う。

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 ());
}
```

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。

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　例題のゲーム

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 [9] の簡易版を作成する。

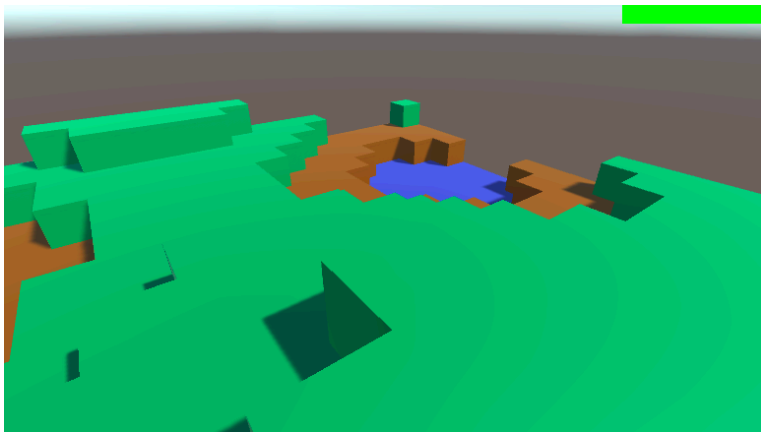

図 6.1: craft

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　ゲームを構成する要素

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。クラスは MonoBehaviour を継承している場合のみ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 クラスが持つパラメータ

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

ステージ上に回復アイテムをランダムに配置する。回復アイテムにはコンポーネントとして、ItemFood クラスを紐付ける。ItemFood が持つ変数を表 6.2 に示す。

表 6.2: ItemFood クラスが持つパラメータ

string `Name`	食べ物の名前
int `Recovery`	アイテムを取得時に回復する数

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 6.3 に示す。

表 6.3: Player クラスが持つパラメータ

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

6.3 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。

ItemTree は第 4 章で述べたマスターデータとして扱う。

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 6.2 では Jungle に格納する構造を示したものである。

Jungle ではオブジェクトが単体で持つデータとオブジェクトが複数持つデータを同時に表現できる。

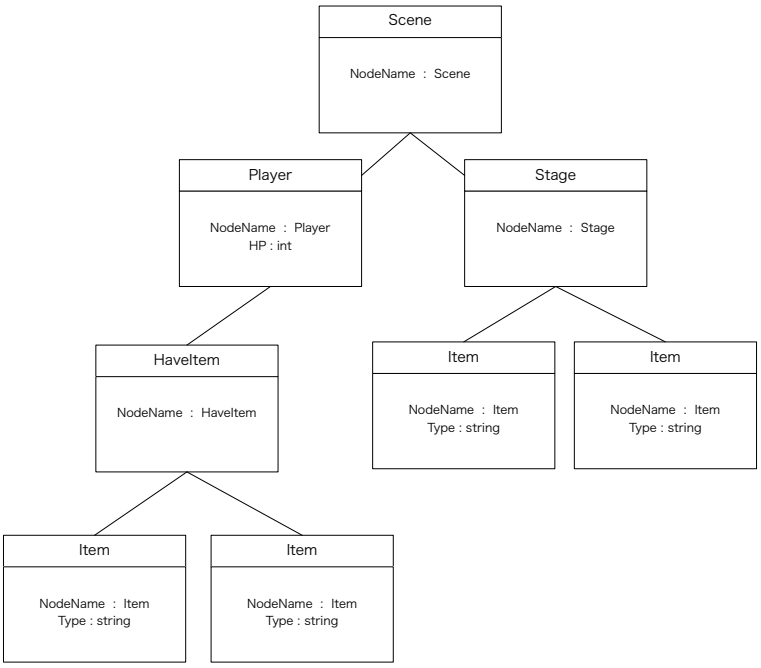

図 6.2: GameTree

ノード 1 つの Attribute に対してデータを格納する。図 6.2 のように Player が持つアイテムを表現したい場合は Player ノードの子として HaveItem ノードを作る。HaveItem ノードの子として持っているアイテムを子ノードとすればよい。

ItemTree では Item の情報が格納されている。図 6.3 では Jungle に格納している Item の構造を示したものである。

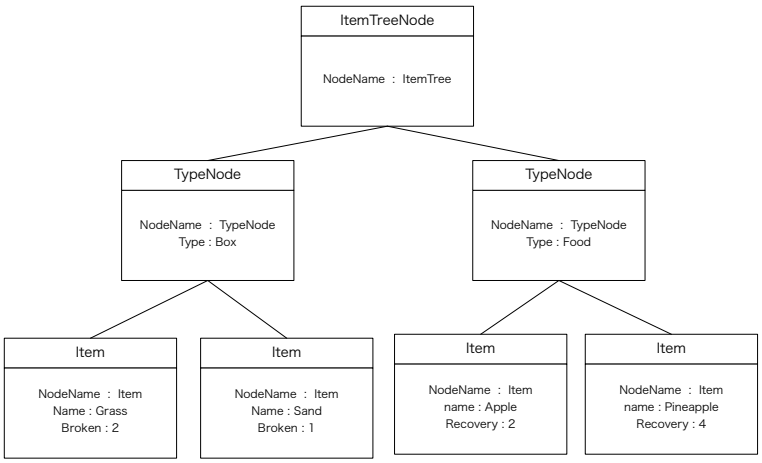

図 6.3: ItemTree

ItemTree では Root ノードは Item の Type が書かれた子ノードを持っている。子ノードは Stage を構成する Box Type、回復アイテムとする Food Type の 2 種類である。

6.4　ゲームエンジン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。

第7章　Jungle-Sharpの評価

本章ではC#とJava版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。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 7.1: 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7.1に示す。

Jungleでは木構造を変更する計算量として、O(1)からO(n)が期待される。

図7.1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。

XamarinはC#の性能を確認するために実行した。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。

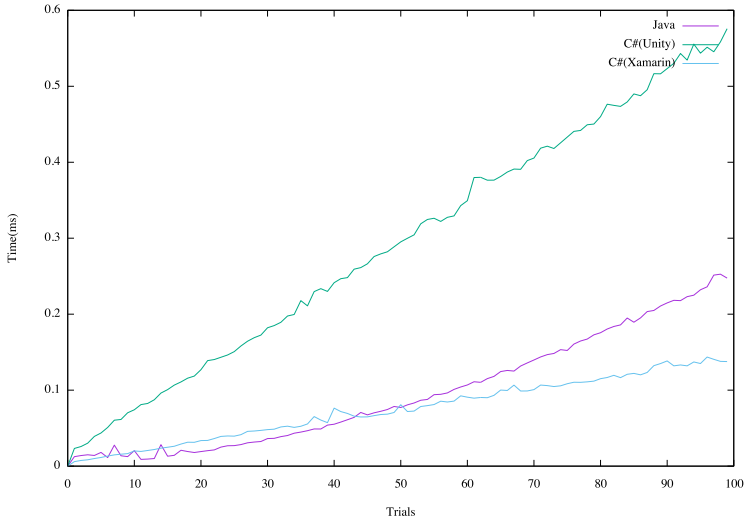

図 7.1: BenchMark

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 100 回行い、測定する。

以下に Jungle、SQLite3、PlayerPrefs でのデータ挿入を行うサンプルコードを記述する。

BenchMarkmark.cs

```
// Jungle
for (int i = 0; i < TrialCount; i++) {
    either.bind ((JungleTreeEditor arg) => {
      return arg.putAttribute (rootPath ,"Player_" + i, HP);
    });
}

either.bind ((JungleTreeEditor arg) => {
    return arg.commit();
});

// SQLite3
for (int i = 0; i < TrialCount; i++) {
    sql.ExecuteNonQuery ("insert into player values("+ i + "," + HP + " )");
}

// PlayerPrefs
for (int i = 0; i < TrialCount; i++) {
    PlayerPrefs.SetInt ("Player_" + i, HP);
}
PlayerPrefs.Save ();
```

Jungle は挿入後、`commit()` を行うまでを挿入とする。

`PlayerPrefs` は `Save()` を行うとバイナリに書き出される。そこまでを挿入とする。

計測結果を以下に記述する。

`Jungle` はデータを直接プログラム内部、つまりオンメモリに持っている。そのため通信を行わずにデータのやり取りができる。

表 7.2: 実行結果

Jungle	12.217ms
SQLite3	126.265ms
PlayerPrefs	985.131ms

SQLite3 ではデータを挿入の SQL を実行するたびデータベースとの通信を行うため遅くなっている。

PlayerPrefs はデータをまとめてセットすることができる。しかし、バイナリ形式で保存されるため、書き出す時間がかかってしまう。

第8章　結論

8.1　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] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels. Dynamo: Amazon's highly available key-value store. 2007.

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

[3] Seth Gilbert Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 2002.

[4] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. *LADIS*, Mar 2003.

[5] Fay Changand Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable : A distributed storege system for structured data.

[6] 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.

[7] 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.

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

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

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