Vol.2018-OS-143 No.12
2018/5/22

分散フレームワーク Christie と分散木構造データベース Jungle

河 野 真 治[†1]

現代のインターネットやアプリケーションで使われるデータは、複雑な構造を持っており、RDB の第一正規系には納まらないようになってきている。また、RDB では、編集中にロックがかけられ、複雑な構造に対する変更に向いていない。これらの問題は、不定形のデータ構造と RDB の表構造のミスマッチである。これを DB のインピーダンスミスマッチと言うことがある。そこで当研究室では、非破壊的木構造データベース Jungle を開発している。Jungle では、不定形のデータ構造を、直接木構造として格納することができる。木構造の変更は、過去の版を保存する非破壊で行われ、木のルートを Atomic に入れ替える操作がトランザクションになる。これにより、複雑な構造を DB として素直に取り扱うことができる。木構造の変更は、木の形に依存している。サービスやアプリケーションに適した木の形があり、それに対応した木の変更方法を採用する必要がある。本研究では Jungle の木と Index の編集機能の改善を行う。直線的なリスト構造に適した Push/Pop と差分リストを提案し、実装と評価を行なった。巨大な木が必要な場合は、木を特定の Key を用いて balance させることにより、変更を O(log n) にすることができる。Jungle は分散構造を取ることもできる。複数の木を複数の分散した Jungle ノード間で通信することにより、Jungle をスケールさせる。Jungle の木の変更 Log を当研究室で開発した分散フレームワーク Alice[1)] を用いて通信する。それぞれの木は、ルートノードに集約され、集約の過程で、競合する変更の Merge を行う。分散 Jungle の性能を測定する手法について述べる。

Shinji KONO [†1]

1. 分散アプリケーションとフレームワーク

現代のサービスは Internet と切り離せない形で提供されている。それらは比較的 ad-hoc に開発発展してきた HTTP 上に構築されている。一方で、分散データベースや分散プログラムをサポートするフレームワークの研究も古くから行われてきている。例えば、MySQL や Postgress などのオープンソースデータベースでもリプリケーションなどの分散技術を採用するようになってきている。HTTP もセキュリティを重視した HTTPS や、HTTP 上のファイル変更プロトコルである WebDAV などが導入されてきている。残念ながら、いまだに定番の分散フレームワークは存在せず、分散データベースも広く使われるようにはなっていない。古くに提案された分散フレームワーク CORBA や、KVS として導入された Cassandra あるいは HDFS なども企業内ネットワークで使われるに留まっている。

現代の OS には階層型ファイルシステムが必須なものとして備わっており、ファイルシステムの分散版が使われるようになると期待された時期もあった。Andrew File System などが設計されたが、広く使われるにはいたらなかった。本論文では、当研究室で開発してきた分散フレームワーク Alice と、それを用いて実装された分散木構造データベース Jungle を見直すことにより、幅広く使われる分散サービスを構築するためのフレームワークのあるべき姿について考察する。

†1 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.

2. 分散フレームワーク Alice

ネットワーク上のサービスには、利用者数の爆発的な増加を受け入れるスケールアウト (サーバの数を増やすことによりサービスの質を維持する手法) が必須である。安定したネットワークサービスを提供するためには、分散プログラムに信頼性とスケーラビリティが要求される。ここでいう信頼性とは、定められた環境下で安定して仕様に従った動作を行うことを指す。

分散プログラムには以下の 3 つの要素がある[2)]。

- ノード内の計算
- ノード間通信
- 地理的に分散したノード

本研究室で開発された分散フレームワーク Alice は、タスクを Code Segment、データを Data Segment と

ⓒ 2018 Information Processing Society of Japan

いう単位で記述し、Code Segment はインプットとなる Data Segment が全て揃うと並列に実行される。Data Segment は対になる key が存在し、Data Segment Manager というノードごとに存在する独自のデータベースによって管理されている。通信する各ノードに対応するラベル付きのプロキシである Remote Data Segment Manager を立て、ラベルと key を指定してデータを take/put する。Linda の Tuple space をノード毎に用意したような構造になっている。Linda と異なり、Tuple space が全世界で一つということはない。

Alice では TopologyManager という機構が分散ノードを管理しており、静的・動的なトポロジーを自動構成する。静的トポロジーではプログラマがトポロジーを図として記述できるため、より分かりやすく詳細な設定ができる。TopologyManager 内には KeepAlive という機能があり、常にノードが生きているか Heartbeat を送信して監視しており、どこかのノードに障害が起こればトポロジーを再構成するといった対応ができる。

3. 木構造分散データベース Jungle

Web 上の情報は複雑な木構造を持っていて、それを相互に参照する形式を持っている。Web 上の相互参照は URI によるものだが、URI の存在は保証されているわけではなく、リンク切れなどが存在する。広く使われるようになった RDB は、基本的にはネスト構造を持たない表の集まりであり、Web 上の情報には直接は対応しない。そこで、本研究室では木構造を直接扱うデータベース Jungle を提案している。Jungle では、木構造を扱えることによって、従来の RDB とは異なり、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。Jungle では非破壊的木構造を採用している。データの元の木を直接書き換えずに保存し、新しく構築した木のデータ構造を編集する方法である。木構造は、そのルートを Atomic に書き換えることにより、その変更の整合性を維持する。ルート複数存在し URI に相当するラベルを持っている。木に対する変更を log として記録し、それを Alice によって通信し、分散データベースを構成している。

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。また、ルートノードは例外として-1 と表記される。 NodePath クラスを用いて <-1,0,2,3 > を表している際の例を (図 1) に示す。

木のノードを他の木から参照することが必要な場合には、ノードに Index を張ることができる。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。そのために非破壊な Red Black Tree を Java 上に開発した。

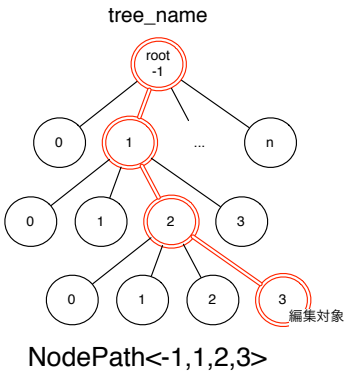

図 1 NodePath

Jungle は、読み込みは高速に行える反面、書き込みの手間は木の形・大きさに依存している。通常の変更では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。それに対処するために、木の変更アルゴリズムを三種提供している。

線形の木を O(1) で変更する PushPop 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、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していくため、ノードの並びが逆順になってしまう。Log などの正順を要求する場合に対処するために、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。

完全にランダムな変更が巨大な木に対して必要な場合には、Jungle の木そのものを Red Black Tree として構築する手法を使うことができる。この場合は木の構造を固定することはできないが、O(log n) での木の変更が可能になる。これは表構造そのものになる。これにより、Jungle を RDB して使うことも可能になった。Reb Black Jungle Tree 用には専用の Node Path が提供される。

ⓒ 2018 Information Processing Society of Japan

4. Jungle の分散構成

Jungle で木構造を分散させるために、Alice による通信を使用する。木への変更をコマンドログとして記録し、それを Alice を用いて他のノードに転送する。他のノードはログに従って自分の持っている木を変更する。複数のノードで変更を共有する手法は様々なものが開発されてきているが、現在は以下のような簡単なものを実装している。

まず、ノードを木構造に接続する。変更ログを自ノードに接続されているノードに向かって送信する。受けとったノードは自分の木を変更したのち、送られてきたノード以外の自分のノードに接続されているノードに変更ログを転送する (Split Horizon)。この方法では全部のノードに単純に変更が伝搬する。変更が衝突した場合は、Merge 処理を行う。Merge 処理は失敗しないと仮定する。Merge 処理による変更は他ノードに転送しない。失敗しない Merge は一般的な整合性を保証することはできないが、SNS への投稿などの場合では十分に機能する。今回は分散実装の確認のために簡単な方法を実装した。多数決などの複雑な分散機構を載せることも将来的には可能である。通信部分は Alice を用いて実装されているので、Jungle の分散構成は Alice の Toplogy Manager によって動的あるいは静的に構成される。

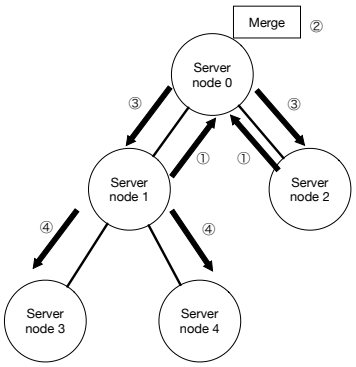

図 2 ツリー型のトポロジー

(図 2) の矢印の流れを以下に示す。

(1) servernode 1, servernode 2 からきたデータが servernode 0 で衝突。
(2) 衝突したデータの Merge が行われる。
(3) Merge されたデータが servernode 1,servernode 2 へ伝搬
(4) servernode1 から Merge されたデータが servernode 3、servernode 4 へ伝搬。全体でデータの整合性が取れる。

5. Jungle と Alice の問題点

Jungle の分散実装を行う時に、Alice で分散プログラムを記述させる時の問題点がいくつか明らかになった。

まず、Jungle と Alice の機能の重複がある。Alice を採用したのは簡単に Jungle の分散構成を実現するためだが、Jungle も Alice も、一種のデータベースであり、キーにより木構造あるいはオブジェクトを格納している。差は扱うデータが木構造かどうかと、トランザクションの取扱いの違いである。

Alice は DataSegment をキーにより通信するが、このキーはプログラムの任意の位置からアクセスすることができる。つまり大域変数的に扱うことが可能である。DataSegment へのアクセスはキーを取得した CodeSegment のみであり、変更はキューに格納され、読み出しはキューに沿って行われる。従って、通常の大域変数と違い、並列実行時に不整合が起きることはない。しかし、キーが大域的に見えているので、DataSegment へのアクセスをモジュール内などに制限することは難しい。

現状の Jungle の木のノードには属性と値の組があり、DB のレコードとして機能するが、値として使えるのは ByteBuffer であり型を持っていない。Jungle のトランザクションは Either という方法で実装されており、細かいエラーチェックを行う必要があり煩雑である。

Alice の DataSegment の管理は Singleton で実装されており、同一プロセス上で複数の Alice のインスタンスを立ち上げることができない。これにより分散アルゴリズムのテストを単一プロセスで行うことができないという問題がある。また、NAT 越えの実装を行う時にも複数のプロセスが必要になってしまう。

現状の Alice の記述方法が煩雑で、型の整合性をコンパイル時に指定できない。これに付いては次節でさらに詳しく述べる。

6. CodeSegment の記述方法

CS をユーザーが記述する際には CodeSegment クラスを継承して記述する (ソースコード 1 , 2)。継承することにより Code Segment で使用する Data Segment API を利用する事ができる。

Alice には、Start CS (ソースコード 1) という C の main に相当するような最初に実行される CS がある。Start CS はどの DS にも依存しない。つまり Input DS を持たない。この CS を main メソッド内で new し、execute メソッドを呼ぶことで実行を開始させることができる。

```
public class StartCodeSegment extends CodeSegment {
```

ⓒ 2018 Information Processing Society of Japan

```
    @Override
    public void run() {
        new TestCodeSegment();

        int count = 0;
        ods.put("local", "cnt", count);
    }

}
```

Code 1　StartCodeSegment

```
public class TestCodeSegment extends CodeSegment {
    private Receiver input1 = ids.create(CommandType.
        TAKE);

    public TestCodeSegment() {
        input1.setKey("local", "cnt");
    }

    @Override
    public void run() {
        int count = input1.asInteger();
        System.out.println("data␣=␣" + count);
        count++;
        if (count == 10){
            System.exit(0);
        }
        new TestCodeSegment();
        ods.put("local", "cnt", count);
    }
}
```

Code 2　CodeSegment

ソースコード 1 は、5 行目で次に実行させたい CS(ソースコード 2) を作成している。8 行目で Output DS API を通して Local DSM に対して DS を put している。Output DS API は CS の `ods` というフィールドを用いてアクセスする。`ods` は `put` と `update` と `flip` を実行することができる。TestCodeSegment はこの”cnt”という key に対して依存関係があり、8 行目で put が行われると TestCodeSegment は実行される。

CS の Input DS は、CS の作成時に指定する必要がある。指定は CommandType(PEEK か TAKE)、DSM 名、そして key よって行われる。Input DS API は CS の `ids` というフィールドを用いてアクセスする。Output DS は、`ods` が提供する put/update/flip メソッドをそのまま呼べばよかったが、Input DS の場合 `ids` に peek/take メソッドはなく、create/setKey メソッド内で CommandType を指定して実行する。

ソースコード 2 は、0 から 9 までインクリメントする例題である。2 行目では、Input DS API がもつ create メソッドで Input DS を格納する受け皿 (Receiver) を作っている。引数には `PEEK` または `TAKE` を指定する。

- `Receiver create(CommandType type)`

4 行目から 6 行目はコンストラクタである。コンストラクタはオブジェクト指向のプログラミング言語で新たなオブジェクトを生成する際に呼び出されて内容の初期化を行う関数である。

TestCodeSegment のコンストラクタが呼ばれた際には、

(1) CS が持つフィールド変数 `Receiver input` に `ids.create(CommandType.TAKE)` が行われ、`input` が初期化される。
(2) 5 行目にある TestCodeSegment のコンストラクタの TAKE が実行される。

5 行目は、2 行目の create で作られた Receiver が提供する setKey メソッドを用いて Local DSM から DS を取得している。

- `void setKey(String managerKey, String key)`

setKey メソッドは peek/take の実行を行う。どの DSM のどの key に対して peek または take コマンドを実行させるかを指定できる。コマンドの結果がレスポンスとして届き次第 CS は実行される。

ⓒ 2018 Information Processing Society of Japan

実行される run メソッドの内容は

(1) 10 行目で取得された DS を Integer 型に変換して count に代入する。
(2) 12 行目で count をインクリメントする。
(3) 16 行目で次に実行される CS を作る。run 内の処理を終えたら CS は破棄されるため、処理を繰り返したい場合はこのように新しいく CS を作る必要がある。この時点で次の CS は Input DS の待ち状態に入る。
(4) 17 行目で count を Local DSM に put する。Input DS が揃い待ち状態が解決されたため、次の CS が実行される。
(5) 13 行目が終了条件であり、count の値が 10 になれば終了する。

となっている。

1. で用いられている asInteger() は asClass メソッドの一部であり、asClass は take/peek で取得した DS を Object 型から任意の型で取得するための API である。

- `<T> T asClass(Class<T> clazz)`

CS 内で DS のデータを扱うには、正しい型を意識しながらこの asClass メソッドを使わなければならない。

このように、InputDS を記述するには、一度フィールドで Receiver を create して、その後 Reveiver に対して setKey で待ち合わせる key を指定しなければならない。このようにインプットの処理が分離されてしまっていては、記述が煩雑な上にコードを読んだ際にどの key に対して待ち合わせを行っているのか直感的に分からない。

さらに、setKey は明確な記述場所が決まっていないため、その DS を待ち合わせている CS 以外からも呼び出せてしまう (ソースコード 3)。

```
public class StartCodeSegment extends CodeSegment {
    @Override
    public void run() {
        TestCodeSegment cs = new TestCodeSegment();
        cs.input.setKey("data");
        ods.put("local", "data", 1);
    }
}
```

Code 3　setKey

```
public class TestCodeSegment extends CodeSegment {
    private Receiver input = ids.create(CommandType.
        TAKE);

    @Override
    public void run(){
        System.out.println("data␣=␣" + input.asInteger
            ());
    }
}
```

Code 4　Seprated setKey

このような書き方をされると、CS だけを見てどの key に対して待ち合わせを行っているのかわからないため、setKey を呼び出しているコードを辿る必要がある。これでは見通しが悪いため、どこで key を指定するのか明確にすべきである。

可読性の低いコードはプログラマの負担となるため、CS が何を待ち合わせているのかその CS を見ただけで理解できるように記述の分離問題を改善しなくてはならない。

setKey は CS のコンストラクタで指定することが多い。このとき、指定する key は引数などから動的に受け取り、セットすることができる。しかし、それでは実際にどんな処理が行われているのかわかりづらく、また、put する部分などの該当する key を扱う全てコードを変更しなければならない。このように、Alice では CS を使いまわすことを考慮して動的な setKey を可能にしてしまったせいで、慎重に書かなければプログラムの信頼性が保てないようになってしまっている。そのため、動的な setKey はできないように制限し、コードの見通しを良くする必要がある。CS に対してインプットとなる key が静的に決まれば、待ち合わせている key に対しての put のし忘れなどの問題をコンパイル時のモデル検査などで発見することができると考えられる。

inputDS を受け取る Receiver はデータを Object 型で持っており、そのデータを CS 内で扱うには正しい型にキャストする必要がある。しかし、inputDS で指定するのは key のみであり、そのデータの型までは分からない。そのため、DS の型を知るには put している部分まで辿る必要がある。辿っても flip されている可能性もあるため、最初にその DS を put している部分を見つけるのは困難である。従って、待ち合わせている key にどのような型のデータが対応しているのかをその CS を見ただけで分かるようにするべきと考える。

key 名とその key で待ち合わせた DS を受け取る Receiver 名は異なることがある。もしプログラマが適当に命名してしまえば後々混乱を招くため、待ち合わせる key 名と input DS の変数名一致を強制させたい。

7. 分散フレームワーク Christie の設計

以上の問題を踏まえ、新しい分散フレームワーク Christie の設計とプロトタイプの実装を行った。Christie に必要な要件は以下のように考えた。

- `create/setKey` のような煩雑な `API` をシンプルにし可読性を向上させる
- 並列分散環境下での型の整合性を保証する構文と仕組みを提供する
- 一つのプロセス内で複数のインスタンスを同時に立ち上げられる

ⓒ 2018 Information Processing Society of Japan

- 木構造などのデータ構造をネットワーク上でやりとりできる
- 通信に用いるキーの階層的管理を提供する
- これらの仕組みを実現するメタ計算機構を提供する

Chirstie では、CodeSegment/DataSegment と呼ばずに、CodeGear/DataGear と呼ぶ。これは将来的には Gears OS により実装するためである。

Alice の API の問題は、送信するオブジェクトの型を前もって指定できないことと、記述する場所に自由度があることが問題になっている。そこで、DataGear を Java の Annoation を用いて宣言することにした。これに関しては、実装を行ない、妥当な API を決定した。これについは次節で説明する。

Alice の Singleton は広範囲で使われており、Refactoring では修正できない。そこで、Christie では基本的な部分を再実装することにした。

Jungle との整合性では、Alice と Jungle の両方にオブジェクトのデータベースが存在することが良くない。Jungle では変更ログを Alice により送信しているが、これはリスト構造を送信することに相当する。Jungle では木の変更を伝搬させるが、これは木構造そのものを通信しているのと同等である。つまり、Alice の通信を木構造を持つオブジェクトにまで拡張することが可能だと考えられる。

Jungle の木構造と、Alice の Tuple 空間の違いは、変更時の同期方法にある。Jungle では木はラベルにより指定されて、その変更はトランザクションとして失敗と成功がある形で直列化される。Alice の同期機構は、Data Segment の待ち合わせによって行われる。単一の Data Segment を待ち合わせれば、それは単一のトランザクションと同等になる。複数の Data Segment を待ち合わせる形の同期は Jungle では実現できない。Jungle の木をキーに対応する Data Segment とすることよって、Alice により Jungle のトランザクションを実現できる。

Jungle の木のラベルは大域的な ID である必要があるが、特に構造を持たせていない。Alice のラベルは、接続された 2 点間でのみ有効になっている。従って、Jungle の木のラベルは分散システムの中で管理する必要がある。通常の URI のように木構造を持った構造として分散管理することが望ましい。つまり、Jungle の木のラベルは Jungle 自体で管理される木構造であるべきだと思われる。これはファイルシステムのパスに相当する。Christie は Gears OS での分散ファイルシステムとして使えるのが望ましい。

8. アノテーションの導入

InputAPI には Alice と同じく Take と Peek を用意した。Christie では Input DG の指定にはアノテーションを使う。アノテーションとは、クラスやメソッド、パッケージに対して付加情報を記述できる Java の Meta Computation である。先頭に@をつけることで記述でき、オリジナルのアノテーションを定義することもできる。

Alice では Input の受け皿である Receiver を作り後から key をセットしていたが、Christie では Input となる型の変数を直接宣言し、変数名として key を記述する。そして、その宣言の上にアノテーションで Take または Peek を指定する (ソースコード 5)。

```
@Take
public String name;
```

Code 5 Take の例

アノテーションで指定した InputDG は、CG を生成した際に CodeGear.class 内で待ち合わせの処理が行われる。これには Java の reflectionAPI を利用しており、アノテーションと同時に変数名も取得できるため、変数名による key 指定が実現した。

Christie のこのインプットアノテーションはフィールドに対してしか記述できないため、key の指定と Take/Peek の指定を必ず一箇所で書くことが明確に決まっている。そのため Alice のように外の CS からの key への干渉をされることがない。このように、アノテーションを用いたことで、Alice の記述の分離問題が解決された。また、key を変数名にしたことで、動的な key の指定や、key と変数名の不一致による可読性の低下を防ぐことができた。

リモートノードに対して Take/Peek する際は、TakeFrom/PeekFrom のアノテーションを用いる (ソースコード 6)。

```
@TakeFrom("remote")
public String name;
```

Code 6 TakeFrom

なお、圧縮の Meta Computation は Alice と同様で、指定する際に DGM 名の前に compressed をつける (ソースコード 7)。

```
@TakeFrom("compressedremote")
public String name;
```

ⓒ 2018 Information Processing Society of Japan

Code 7 Remote から圧縮して受け取る例

OutputAPI には put/flip を用意した。基本的なシンタックスは Alice と同様だが、Christie では put/flip のメソッドは CodeGear.class に用意されている。そのため CodeGear.class を継承する CG で直接 put メソッドを呼ぶことができる (ソースコード 8)。

```
put("remote", "count", 1);
```

Code 8 put

そのため、Christie には Alice の ODS にあたる部分がない。ODS を経由するより直接 DGM に書き込むような記述のほうが直感的であると考えたためである。圧縮を指定しての put も、Alice 同様 DGM 名の前に compressed をつける。

型の整合性の向上

Christie では Receiver 型ではなく直接変数を宣言する。そのため他の場所を辿らなくとも CG を見るだけでインプットされるデータの型が分かるようになった。また、変数を直接宣言するため、そもそも Alice のように asClass メソッドで型の取り出す必要がない。ソースコード 9 は InputDG のデータを扱うである。

```
public class GetData extends CodeGear{

    @Take
    public String name;

    @Override
    protected void run(CodeGearManager cgm) {
        System.out.println("this␣name␣is␣:␣" + name);
    }
}
```

Code 9 InputDG を扱う例

InputDG として宣言した変数の型は、reflection-API により内部で保存され、リモートノードと通信する際も適切な変換が行われる。このようにプログラマが指定しなくとも正しい型で取得できるため、プログラマの負担を減らし信頼性を保証することができる。

以下のコードは LocalDSM に put した DG を取り出して表示するのを 10 回繰り返す例題である。

```
public class StartTest extends StartCodeGear{

    public StartTest(CodeGearManager cgm) {
        super(cgm);
    }

    public static void main(String args[]){
        StartTest start = new StartTest(createCGM
              (10000));
    }

    @Override
    protected void run(CodeGearManager cgm) {
        cgm.setup(new TestCodeGear());
        put("count", 1);
    }
}
```

Code 10 StartCodeGear の例

```
public class TestCodeGear extends CodeGear {

    @Take
    public count;

    public void run(CodeGearManager cgm){
        System.out.println(hoge.getData());

        if (count != 10){
            cgm.setup(new TestCodeGear());
            put("count", count + 1);
        }
    }
}
```

Code 11 CodeGear の例

Alice 同様、Christie でも InputDG を持たない StartCG から処理を開始する。StartCG は StartCodeGear.class を継承することで記述できる。Alice では StartCS も CodeSegment.class を継承して書かれていたため、どれが StartCS なのか判別しづらかったが、Christie ではその心配はない。

StartCG を記述する際には createCGM メソッドで CGM を生成してコンストラクタに渡す必要がある。ソースコード 10 の 8 行目でそれが行われている。createCGM の引数にはリモートノードとソケット通信する際使うポート番号を指定する。CGM を生成した際に LocalDGM やリモートと通信を行うための Daemon も作られる。

CG に対してアノテーションから待ち合わせを実行する処理は setup メソッドが行う。そのためソースコード 10 の 13 行目、ソースコード 11 の 10 行目のように、new した CG を CGM の setup メソッドに渡す必要がある。Alice では new すれば CG が待ちに入ったが、Christie では一度 CG を new しないとアノテーションから待ち合わせを行う処理ができないため、new の後に setup を行う。そのため、CG の生成

ⓒ 2018 Information Processing Society of Japan

には必ず CGM が必要になる。run で CGM を受け渡すのはこのためである。なお、StartCG はインプットを持たないため、setup を行う必要がなく、new された時点で run が実行される。

9. Unix ファイルシステムとの比較

Christie を分散ファイルシステムとして使うのと、Unix ファイルシステム上の分散ファイルシステムとの違いについて考察する。

Christie に格納されるのはオブジェクトであり、型のないテキストファイルとは異なる。もちろん、文字列をそのまま格納することもできるが、巨大な文字列にはなんらかの構造を与えてランダムアクセスなどができるようにする。その意味で、Unix ファイルシステムでもファイルには必ず構造が入っている。

Unix ファイルにはディレクトリ構造があり、ディレクトリの構造にはトランザクションが導入されている。Christie の場合は、Christie の同期機構としてトランザクションが定義される。

Unix ファイル全体は i ノードを使った B-Tree 構造を持つ。これらはディレクトリ構造による木構造のパスでアクセスされる。Christie ではパスとディレクトリは木構造のデータベースとして定義される。

Unix ファイルの持続性はディスク上の i ノードの持続性によって実現される。Christie は持続性のあるノードに木構造を複製することによって持続性を実現する。

Unix ファイルは read/write によりアクセスするが、Christie ではメモリ上のオブジェクトであり、read-/write ではなく名前で指定された木に対する get/put で変更を行う。

持続性のあるノードに複製された Christie のオブジェクトはメモリ上から削除しても良い。再度必要な場合は、パスを用いて持続性のあるノードから複製する。

10. ま と め

分散木構造データベース Jungle と、分散フレームワーク Alice について考察し、両者を統一する形で、分散フレームワーク Christie を提案した。

Christie ではアノテーションを用いて、Alice の欠点であった記述の分離と型の整合性をコンパイル時に解決できない問題を解決した。

Jungle の構造を Christie に内蔵することにより、Christie の Data Gear は、分散環境で共有、あるいは、持続性を持つことができるようになる。その Data Gear へのアクセスに、大域的な木構造を持つラベルを用意することで、Christie の Data Gear をファイルシステムのように使えるようになる。

将来的には Gears OS でのファイルシステムとして Christie を使えるように、持続性、DataGear にアクセスするパスとしての木構造、分散環境での同期手法を研究していく予定である。

参 考 文 献

1) 照屋のぞみ，河野真治：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 (2016).
2) 安村 恭一 and 河野真治：動的ルーティングによりタプル配信を行なう分散タプルスペース Federated Linda，日本ソフトウェア科学会第 22 回大会論文集 (2005).

ⓒ 2018 Information Processing Society of Japan