

継続を使用する並列分散フレームワークのUnity実装

安田 亮^{1,a)} 河野 真治^{2,b)}

概要：FPS や MMORPG などのゲームにおける通信方式には、クライアントサーバ方式と p2p 方式の 2 つが考えられる。しかし、クライアントの負荷軽減やチート対策などを理由にクライアントサーバ方式が主流である。データの同期にはサーバを経由するため低速である。そこで本研究室で開発している分散フレームワーク Christie を用いることで、高速かつ、安全に、データの同期を行いたいと考えた。本研究では Christie をゲームエンジン Unity に対応するため、C#への書き換えを行う。

1. オンラインゲームにおけるデータ通信 2. Chrisite の C#への書き換えについて

Chrisite は Alice というプロジェクトで開発が行われていた。しかし Alice には様々な問題点があった。

3. Christie の基礎概念

Chrisite は当研究室で開発している分散通信フレームワークである。同じく当研究室で開発している GearsOS のファイルシステムに組み込まれる予定があるため、GearsOS を構成する言語 Continuation based C と似た概念を持っている。Chrisite に存在する概念として以下のようなものがある。

- CodeGear
- DataGear
- CodeGearManager
- DataGearManager

以下は java 版の Chrisite について解説を行う。CodeGear はクラスやスレッドに相当する。DataGear は変数データに相当し、CodeGear 内で annotation を用いて変数データを取得する。CodeGear 内に記述した全ての DataGear の中にデータが格納された際に、初めてその CodeGear が実行されるという仕組みになっている。CodeGearManager はノードであり、CodeGear、DataGear、DataGearManager を管理する。DataGearManager は DataGear を管理するものであり、put という操作により変数データ、つまり DataGear を格納できる。DataGearManager の put 操

作を行う際には Local と Remote のどちらかを選び、変数の key とデータを引数として渡す。Local であれば、Local の CodeGearManager が管理している DataGearManager に対し DataGear を格納していく。Remote であれば、接続した Remote 先の CodeGearManager が管理している DataGearManager に DataGear を格納できる。put 操作を行った後は、対象の DataGearManager の中に queue として保管される。DataGear を取り出す際には、CodeGear 内で宣言した変数データに annotation をつける。DataGear の annotation には Take、Peek、TakeFrom、PeekFrom の 4 つがある。

Take 先頭の DataGear を読み込み、その DataGear を削除する。DataGear が複数ある場合、この動作を用いる

Peek 先頭の DataGear を読み込むが、DataGear が削除されない。そのため、特に操作をしない場合は同じデータを参照し続ける。

TakeFrom (Remote DGM name) Take と似ているが、Remote DGM name を指定することで、その接続先 (Remote) の DataGearManager から Take 操作を行える。

PeekFrom (Remote DGM name) Peek と似ているが、Remote DGM name を指定することで、その接続先 (Remote) の DataGearManager から Peek 操作を行える。

4. プログラムの例

Code 1、Code 2、Code 3 は Chrisite の機能を使用して hello world を出力する例題である。

Code 1: StartHelloWorld

¹ 琉球大学大学院理工学研究科情報工学専攻

² 琉球大学工学部工学科知能情報コース

^{a)} riono210@cr.ie.u-ryukyu.ac.jp

^{b)} kono@ie.u-ryukyu.ac.jp

```
1 public class StartHelloWorld extends StartCodeGear {
2
3     public StartHelloWorld(CodeGearManager cgm) {
4         super(cgm);
5     }
6
7     public static void main(String[] args){
8         CodeGearManager cgm = createCGM(10000);
9         cgm.setup(new HelloWorldCodeGear());
10        cgm.setup(new FinishHelloWorld());
11        cgm.getLocalDGM().put("helloWorld", "hello");
12        cgm.getLocalDGM().put("helloWorld", "world");
13    }
14 }
```

Code 2: HelloWorldCodeGear

```
1 public class HelloWorldCodeGear extends CodeGear {
2
3     @Take String helloWorld;
4
5     @Override
6     protected void run(CodeGearManager cgm) {
7         System.out.print(helloWorld + "\n");
8         cgm.setup(new HelloWorldCodeGear());
9         cgm.getLocalDGM().put(helloWorld, helloWorld);
10    }
11 }
```

Code 3: FinishHelloWorld

```
1 public class FinishHelloWorld extends CodeGear {
2     @Take String hello;
3     @Take String world;
4
5     @Override
6     protected void run(CodeGearManager cgm) {
7         cgm.getLocalDGM().finish();
8     }
9 }
```

Code 1 では CodeGearManager を作り、setup(new CodeGear) を行うことで各 CodeGear に記述された DataGear の待ち合わせを行う。全ての DataGear が揃った場合に CodeGear が実行される。CodeGearManager の作成方法は StartCodeGear を継承したもから、createCGM(port) を実行することにより、CodeGearManager が作成できる。

Code 1 の 11、12 行目は put(key, data) を行うことで DataGearManager の queue にデータを格納することができる。key は string 型のみで格納したい変数名を指定する。11、12 行目の put では Code 2 のフィールド変数 helloWorld を指定し、データは”hello” と”world” を逐次的に格納している。

Code 2、Code 3 が CodeGear にあたる。それぞれのフィールド変数には@Take annotation が付いており、DataGearManager に格納された key を参照してデータを取得する。その後 DataGearManger に格納されたデー

タは破棄される。

Code 2 では最初にフィールド変数 helloWorld に string 型の”hello”を取得、print を行い、再び key hello、data ”hello” を DataGearManager に put している。また 8 行目で自らを setup しているため、再帰的に HelloWorld-CodeGear が再実行される。2 回目の実行ではフィールド変数 helloWorld に”world” が格納と出力がされ、key world、data ”world” が DataGearManager に格納される。Code 2 で put した”hello” と”world” は最終的に、Code 3 の同名のフィールド変数に格納される。2 回目の実行でも setup しているが、DataGearManager には key helloWorld のデータが無いため、3 回目以降は実行されない。

Code 2 の 2 回の実行後、Code 2 のローカル変数 hello と world が全て揃ったことにより Code 2 が実行されプログラムは終了する。

5. C# での Christie

Code 1、Code 2、Code 3 が C#ではこうなります

Code 4: C# StartHelloWorld

```
1 public class StartHelloWorld : StartCodeGear {
2
3     public StartHelloWorld(CodeGearManager cgm) : base
4         (cgm) { }
5
6     public static void Main(string[] args) {
7         CodeGearManager cgm = CreateCgm(10000);
8         cgm.Setup(new HelloWorldCodeGear());
9         cgm.Setup(new FinishHelloWorld());
10        cgm.GetLocalDGM().Put("helloWorld", "hello");
11        cgm.GetLocalDGM().Put("helloWorld", "world");
12    }
13 }
```

6. Unity

7. Unityでの動作

8. annotation の書き換え

java 版では DataGear を取得する際に、annotation という java の機能を用いて行った。C#には annotation はなく、代わりに attribute を利用して DataGear の取得を行っている。以下の Code 5、Code 6 は java と C# における Take の実装である。

Code 5: java における Take annotation の実装

```
1 @Target(ElementType.FIELD)
2 @Retention(RetentionPolicy.RUNTIME)
3 public @interface Take { }
```

Code 6: C# における Take attribute の実装

```
1 [AttributeUsage(AttributeTargets.Field)]
2 public class Take : Attribute { }
```

java で annotation を自作する際には、@interfaces で宣言する。また、Code 5 の 8 行目では annotation 情報をどの段階まで保持するかを指定しており、Take の場合 JVM によって保存され、ランタイム環境で使用できる。9 行目では annotation の適用可能箇所を指定しており、フィールド変数に対して適応可能となっている。

C# で attribute を作成する際には、System.Attribute を継承する必要がある。attribute の適用可能箇所については、Code 6 の 4 行目でフィールド変数を指定している。

9. MessagePack の相違点

Christie ではデータを送信する際に、MessagePack を使用してデータを圧縮し、送信している。java 版で使用している MessagePack はバージョンが古く現在はサポートされていない。そのため MessagePack の最新版とは記述方法が異なっている。Code 7 は MessagePack の使用方法を示したものである。

Code 7: java における MessagePack の使用方法

```
1 public class MessagePackExample {
2     @Message // Annotation
3     public static class MyMessage {
4         // public fields are serialized.
5         public String name;
6         public double version;
7     }
8
9     public static void main(String[] args) throws
10         Exception {
11         MyMessage src = new MyMessage();
12         src.name = "msgpack";
13         src.version = 0.6;
14
15         MessagePack msgpack = new MessagePack();
16         // Serialize
17         byte[] bytes = msgpack.write(src);
18         // Deserialize
19         MyMessage dst = msgpack.read(bytes, MyMessage.
20             class);
21     }
```

MessagePack を使用するには圧縮するクラスに対して @Message annotation をつける必要がある。これにより、クラス内で定義した public 変数が圧縮される。Code 7 の 17 - 21 行目は圧縮解凍の例であり、MessagePack のインスタンスを作成後、msgpack.write(data) を行うことで byte[] 型に data を圧縮できる。解凍には msgpack.read を使用し、圧縮された byte[] 型と圧縮対象のクラスを渡すことで解凍できる。

C# の MessagePack は複数存在しており、java と同様

な書き方をする MessagePack-CSharp を選択した。

Code 8: C# における MessagePack の使用方法

```
1 [MessagePackObject]
2 public class MyClass {
3     [Key(0)]
4     public int Age { get; set; }
5     [Key(1)]
6     public string FirstName { get; set; }
7     [Key(2)]
8     public string LastName { get; set; }
9
10    static void Main(string[] args) {
11        var mc = new MyClass {
12            Age = 99,
13            FirstName = "hoge",
14            LastName = "huga",
15        };
16
17        byte[] bytes = MessagePackSerializer.Serialize
18            (mc);
19        MyClass mc2 = MessagePackSerializer.
20            Deserialize<MyClass>(bytes);
21
22        // [99, "hoge", "huga"]
23        var json = MessagePackSerializer.ConvertToJson
24            (bytes);
25        Console.WriteLine(json);
26    }
```

MessagePack-CSharp では java 版と同様にクラスに対して圧縮を行うため Code 8 の 1 行目で MessagePackObject attribute を追加している。また、圧縮する変数に対して key を設定することができ、int や string を指定することができる。

データの圧縮には MessagePackSerializer.Serialize (data) を使用し、byte[] 型に圧縮される。解凍には MessagePackSerializer.Deserialize<T>(data) を使用する。Deserialize はジェネリクス関数であるため、<> 内に解凍するデータのクラスを指定する。Code 8 の 21 行目では、変数それぞれに key を設定していることで json に展開することが可能である。

10. CodeGear 実行時の ThreadPool から Task への変更

java 版では CodeGear の実行に ThreadPool を使用していた。C# では書き換えの際に ThreadPool よりも高機能な Task で書き換えを行った。

Code 9: java における CodeGear を処理する ThreadPool の実装の一部

```
1 public class PriorityThreadPoolExecutors {
2
3     public static ThreadPoolExecutor createThreadPool(
4         int nThreads, int keepAliveTime) {
5         return new PriorityThreadPoolExecutor(nThreads
```

```
        , nThreads, keepAliveTime, TimeUnit.  
        MILLISECONDS);  
5    }  
6    private static class PriorityThreadPoolExecutor  
        extends ThreadPoolExecutor {  
7        private static final int DEFAULT_PRIORITY = 0;  
8        private static AtomicLong instanceCounter =  
            new AtomicLong();  
9  
10       public PriorityThreadPoolExecutor(int  
            corePoolSize, int maximumPoolSize,  
11            int keepAliveTime,  
                TimeUnit unit) {  
12            super(corePoolSize, maximumPoolSize,  
                keepAliveTime, unit, (BlockingQueue)  
                new PriorityBlockingQueue<  
                    ComparableTask>(10,  
13            ComparableTask.  
                comparatorByPriorityAndSequentialOrder());  
14            }  
15  
16       @Override  
17       public void execute(Runnable command) {  
18           // If this is ugly then delegator pattern  
                needed  
19           if (command instanceof ComparableTask) //  
                Already wrapped  
20               super.execute(command);  
21           else {  
22               super.execute(new ComparableRunnableFor(  
                    command));  
23           }  
24       }  
25  
26       private Runnable newComparableRunnableFor(  
            Runnable runnable) {  
27           return new ComparableRunnable((  
                CodeGearExecutor) runnable);  
28       }  
29  
30       @Override  
31       protected <T> RunnableFuture<T> newTaskFor(  
            Runnable runnable, T value) {  
32           return new ComparableFutureTask<>((  
                CodeGearExecutor)runnable, value);  
33       }  
34     }  
35 }
```

Code 9 は java 版における CodeGear を実行する ThreadPool の実装の一部である。java では独自に ThreadPool を作成する際には ThreadPoolExecutor を継承する。また優先度の機構が実装されており、CodeGear 実行時に優先度を定めることが可能になっている。CodeGear の実行には 17 行目の execute を呼び出すことで、実行される。

Code 10: C# における CodeGear を処理する ThreadPool の実装

```
1 public class ThreadPoolExecutors {  
2  
3     public ThreadPoolExecutors() {  
4         int nWorkerThreads;
```

```
5         int nIOThreads;  
6         ThreadPool.GetMinThreads(out nWorkerThreads,  
            out nIOThreads);  
7         ThreadPool.SetMinThreads(nWorkerThreads,  
            nIOThreads);  
8     }  
9  
10    public ThreadPoolExecutors(int nWorkerThreads, int  
        nIOThreads) {  
11        ThreadPool.SetMinThreads(nWorkerThreads,  
            nIOThreads);  
12    }  
13  
14    public void Execute(CodeGearExecutor command) {  
15        Task.Factory.StartNew(() => command.Run());  
16    }  
17 }
```

Code 10 は Code 9 を C# に書き換えを行ったものである。CodeGear の実行には 14 行目の Execute を呼び出し、Task で実行を行っている。Task は C# の ThreadPool を拡張したもので、内部に ThreadPool と実行待ち Queue を持っている。スケジューラを自作することも可能である。実装の優先度が低かったため、今回は CodeGear の priority による実行順変更については実装を行わなかった。

11. チート対策について

12. 実装の現状

参考文献

- [1] RICHARDSON, T., AND LEVINE, J.: The remote framebuffer protocol. RFC 6143 (2011).
- [2] TightVNC Software: <http://www.tightvnc.com>.
- [3] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER,: A. Virtual Network Computing (1998).
- [4] LOUP GAILLY, J., AND ADLER, M.: zlib: A massively spiffy yet delicately unobtrusive compression library., <http://zlib.net>.
- [5] 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 (2011).
- [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的设计・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) (2012).