UnderStand Scala Comparing With Java
Java と比較することにより Scala を理解する

Masato Tsuchiya　2010/04

The attempt to understand Scala from implementing the same problem by Scala and Java is described. The problem is the simulation of digital circuit shown as sample [1]. We convert single and multi- thread case from Scala to Java. Next, we implement better parallel method where circuit elements directly communicate. And last, we implement this method using Scala and compare Scala and Java. The implemented programs are stored in simulation.zip. The directory of program is shown by […].

同じ問題を Scala と Java の両方で実装することにより Scala を理解する試みを述べる。ここで、問題はサンプルとして提示されたデジタル回路のシュミュレーション[1]とした。まず、シングルとマルチスレッドのケースを Scala から Java に変換した。次に回路要素が直接コミニュケーションする並列実行性が高い方法を実装した。 最後にこの方法を Scala で実装して Java と Scala を比較した。実装したプログラムは simulation.zip に収納してある。以下、プログラムのあるディレクトリーを[…]で示す。

Case 1.　Single Thread [simulation/java/single]

This case is single thread and conversion from Scala([1] 18.3 Case study: Discrete event simulation) to Java. The closure of delayed execution can replace by the command pattern in Java. The interface of command is Actor. Anonymous class which implements Actor is useful to avoid increment of tiny class.

ケーススタディー：離散イベントシミュレーション[1] 18.3 章 を Java に変換したプログラムである。遅延実行されるクロージャをコマンドパターンに置き換えることで、Scala プログラムを Java に変換することができた。ここで、コマンドは interface Actor を実装したクラスである。Actor を実装する無名クラスであるコマンドを使用して、小さなクラスが増えないようにした。

Case 2.　Multi Thread [simulation/java/multi/org]

This case is multi thread and conversion from Scala ([1] 30.6 A longer example: Parallel discrete event simulation) to Java. But thread is fixed to an instance and is not

recycled. The Actor of Scala is similar to Unix process communicating by message because Unix process is shared nothing model. All instances which are executed multi thread have an input queue. The queue must be thread safe. We use java.util.concurrent library.

大規模なサンプル：離散イベントシミュレーション[1] 30.6 章を Java に変換したプログラムである。ただし、Thread は特定のインスタンスに固定されていて、再利用されない。Scala の Actor はメッセージ通信を使用した Unix におけるマルチプロセスのプログラムモデルに似いている。なぜなら Unix のプロセスはシェアードナッシングであるから。マルチスレッドで動作するインスタンスには受信スレッドセーフな受信キューが必要である。ここでは java.concurrent ライブラリーを使用した。

Case 3 Direct communication [simulation/java/multi/adv]

This case is also multi thread. But, all circuit elements (Wire, Gate and Probe) directly communicate. In Case 2, Clock is a hub. And all circuit elements only communicate to Clock. The thread of Clock is heavy duty and may be bottleneck.

回路要素(Wire,Gate,Probe)が直接通信する Java による Multi Thread プログラムである。Case 2 は Clock が通信のハブになり、回路要素が直接通信することはない。この方法では Clock のスレッドの負荷が高くボトルネックになるかもしれない。

The analysis is important that three are two kind of message: control and signal. Circuit elements communicate using signal. Only Clock sends control to circuit elements and receives response. Clock is still a hub with control. But, Clock never send and receive signal. The concrete classes of control are Ping/Pong(synchronization), Start/Stop. The concrete classes of signal are SignalChanged and SetSignal. SignalChanged is sent from Gate to Wire. SetSignal is send from Wire to Gate and Probe. We introduce abstract class: Simulant which is able to receive signal and control. Clock and circuit elements are subclass of Simulant.

メッセージは制御と信号の２種類がある分析は重要である。回路要素は信号によりコミニュケーションする。Clock のみが制御を回路要素に発信し、回路要素は Clock に制御を返信する。つまり、制御に関しては、依然、 Clock がハブになる。制御の具象クラスは Ping/Pong(synchronization), Start/Stop である。信号の具象クラスは Wire から Gate と Probe に送られる SignalChanged と Gate から Wire に送られる SetSignal であ

る。ここでは、制御と信号を受信できる抽象クラス Simulant を導入して、Clock と回路要素を Simulant のサブタイプとした。

The behavior of input queue for signal and control is different. If there is no control in input queue, the thread accessing the queue must be blocked. If there is no signal in input queue, the thread accessing the queue should not be blocked. So, we use LinkedBlockingQueue for control and ConcurrentLinkedQueue for signal.

受信キューの動作において信号と制御は異なる。受信キューに制御のメッセージがなければスレッドはブロックされるべきである。一方、受信キューに信号のメッセージがない場合にはスレッドはブロックされるべきではない。このため、制御の受信キューには LinkedBlockingQueue、信号の受信キューには ConcurrentLinkedQueue を使用した。

Ping has current time. When circuit element receives Ping from Clock, it exhausts all signals which have time before or equal current time from ConcurrentLinkedQueue, and changes self state or sends signal to other circuit element discriminating signal. After all signals are done, it sends Pong to Clock to notify done.

Ping には現時刻が含まれている。Clock から Ping を受信した回路要素は受信キュー(ConcurrentLinkedQueue)にある信号の内、現時刻より以前の時刻を有する信号を全て取り出し、信号を識別して自分の状態を変化させるか他の回路要素に信号を発信する。全ての信号の処理が終了したら、Clock に Pong を返信して処理の終了を通知する。

Clock stops simulation. Pong has maximum delay at circuit element. Therefore
The time to stop is current time plus maximum delay of all Pong.

Clock がシミュレーションを終了させる。ここでは、回路要素の遅延時間を Pong に含めた。終了時間は遅延時間の最大値を現在時間に加えた時間である。

Clock is an infrastructure for circuit elements. All circuit elements need Clock.
We inject an instance of Clock to all circuit elements by Spring AOP(@Configurable).

Clock は回路要素のインフラストラクチャーに相当し、全ての回路要素が必要とする。ここでは、Spring AOP を使用して(特に @Configurable)、Clock を回路要素に注入した。

Case 4　Scala Direct Communication　[simulation/scala/multi/adv]

This is Scala version for Case 3. Control and signal are received by one queue. Signal is separated and saved to a list at reading the queue. Because multi threads access the list, we must synchronize the list. Next, we compare Case 3 and Case 4.

Case 3 を Scala に書き直したものである。ここでは、制御と信号を 1 つのキューで受信する。キューから読み出したとき、信号をリストに貯えた。このリストは複数のスレッドからアクセスされるので排他処理が必要である。Case 3, 4 を比較して考察する。

1. Code Length
1. コード量
   1) List operation
   1) List 操作

[Java]
```
if(control instanceof Ping) {
    Ping ping =(Ping)control;
    List<Signal> todo = new ArrayList<Signal>();
    for(Iterator<Signal> i = signals.iterator(); i.hasNext();) {
        Signal signal = i.next();
        if(signal.getTime() <= ping.getTime()) {
            todo.add(signal);
        }
    }
    int delay = 0;
    if( todo.size() > 0 ) {
         for(Signal signal: todo) {
              signals.remove(signal);
         }
         Collections.sort(todo);
         for(Signal signal: todo) {
              int t = receive(signal);
              if( t > delay ) delay = t;
         }
     }
     send(control.getSource(), new Pong(control.getTime(), delay));
```

[Scala]
```
case Ping(source, time) =>
     var todo: List[Signal] = List()
     signals.synchronized {
         val todo = signals.filter(_.time <= time)
         if(!todo.isEmpty) signals = signals.filter(_.time > time)
     }
     val delay = (0 /: todo.sort((a,b) => a.time < b.time))
                          ((m, signal) => max(m, receive(signal)))
     source ! Pong(this, time, delay)
```

   2) POJO

[Java]

```
protected abstract class Message implements Comparable<Message>  {
    private int time;
    Message(int time ) {
        this.time = time;
    }
    int getTime() {
        return time;
    }
    Simulant getSource() {
        return Simulant.this;
    }
    public int compareTo(Message that) {
        return this.time - that.time;
    }
}
```

[Scala]

```
sealed abstract class Message(val source:Simulant, val time: Int)
```

2. OOP vs. Functional programming

This problem is suitable to OOP. Because, it is message passing model between objects. Large scale structures (principal classes) are designed by OO. Functional programming is efficient at method and block level.

2. オブジェクト指向プログラムと関数型プログラム

この問題に関してはオブジェクト間でメッセージ通信するモデルなので、
アプリケーションの大枠(主要クラス)はオブジェクト指向により設計した。
メソッドやブロックレベルの実装には関数型プログラムが効果的であった。

[1] Programming in Scala: A Comprehensive Step-by-step Guide
Martin Odersky,Lex Spoon,Bill Venners, 2009, artima

[1]「Scala スケーラブル プログラミング」
Martin Odersky, Lex Spoon, Bill Venners
長尾訳、羽生田監修
インプレスジャパン 2009/09