Recovery アドバイスをもつアスペクト指向システム

熊原 奈津子　　石川 零　　西澤 無我　　光来 健一　　千葉 滋

東京工業大学大学院
情報理工学研究科 数理・計算科学専攻
{kumahara, rei, muga, kourai, chiba}@csg.is.titech.ac.jp

要 旨

本稿では、 GluonJR がもつ recovery アドバイスを用いることで、例外処理をプログラムロジックから分離して記述することができることを示す。recovery アドバイスで指定された範囲内で例外が生じた場合の処理を書くことができることと、指定した範囲を再度実行する特殊なメソッドをアドバイス内で呼ぶことができるため、例外が発生した時点でプログラムが終了されなくても済むようになることを示す。

1　はじめに

プログラムの実行環境の問題によって例外が発生した場合、その例外に対して適切な処理をすべきである。例外が発生したことを見落として正常時の動作を継続してしまうとより深刻で致命的な異常事態を招いてしまう恐れがある。

しかし、プログラムを記述する際には、例外処理の記述よりも、ロジックの記述に集中できた方がよい。また、ロジックを記述した後に必要に応じて例外処理を追加したい場合がある。本稿ではこの問題点の例として、サーバに負荷をかけて性能を測定するという大規模な実験を行うために複数クライアントを起動させる制御プログラム作成する場合を例にとって考えていく。

この問題点を解決するために、我々は例外処理を行うのに特化したアスペクト指向システム GluonJR を提案する。GluonJR がでは、 recovery アドバイスを用いることで、プログラム中の範囲を指定し、その範囲内で例外が発生した場合の処理をアドバイスとして記述できる。これにより、例外処理をプログラムロジックから分離して記述することができるようになる。また、アドバイスの中から、ポイントカットで指定した範囲の先頭に戻ってその範囲の処理を再実行するためのアドバイスの中で使える特殊なメソッドも提供している。

以下では、2 章で例外処理を行う際の問題点とその具体例を示し、3 章ではその問題を解決するために我々が提案する GluonJR について述べる。4 章では GluonJR を実装するにあたって鍵となる部分について述べる。5 章では関連研究を取り上げ、6 章で本稿をまとめる。

2　例外処理の記述

プログラムの実行環境の問題によって例外が発生した場合、その例外に対して適切な処理をすべきである。今、分散環境上で動くサーバマシンの負荷テストをしており、サーバに対して負荷を発生させるプログラムを複数のクライアント上で起動する制御プログラムを作成しているとする。各クライアントに必要なファイルを送信する部分のプログラムは次のようになるだろう。

```
 1 class Sender{
 2   public void sendFile(String host,
 3       String fileName) throws Exception {
 4     int n;
 5     int port = 9000;
 6     byte[] buff = new byte[1024];
 7
 8     Socket s = new Socket(host, port);
 9     DataOutputStream out
10         = new DataOutputStream(
11             s.getOutputStream());
12     RandomAccessFile file
13         = new RandomAccessFile(fileName, "r");
14
15     while((n = file.read(buff)) > 0){
16       out.write(buff, 0, n);
17     }
```

```
18
19      file.close();
20      out.close();
21      s.close();
22        :
23        :
24    }
25 }
```

この場合、サーバとクライアントをつなぐネットワーク障害に障害が発生したときや、クライアントマシンがそもそも起動していないときに、例外が発生する可能性がある。例えば、8 行目で UnknownHostException や IOException、11 行目で IOException、13 行目で FileNotFoundException など、起こり得る例外はたくさんある。これらの例外は、もし発生すると、sendFile(..) メソッドの実行を中断し、sendFile(..) メソッドの呼び出し側に投げられる。

このような実験プログラムのロジックを記述する際には、例外の処理に関しては後で記述したいことが多い。なぜなら、例外処理を書かなくても多くの場合はうまく動くので、最初の段階ではロジックを書く方に集中できた方がよいからである。例外処理は、後で必要になったとき、はじめて実験プログラムに追加できると望ましい。

しかし、プログラムに例外処理を不用意に後から加えると、既に動いているプログラムを変更することになるため、そのロジックを壊してしまう危険性がある。また追加した例外に修正や変更が加えられた場合には、対象となる例外処理を全て探し出して逐一変更しなければならなくなる。そのようなプログラムの追加は、面倒であるだけでなく、1 つでも変更し忘れるとプログラム全体の整合性がとれなくなる危険性がある。

追加する例外処理は、実験プログラムの例の場合、図 1 の四角で囲んだ部分に書くことになる。図 1 は例外処理について記述されていない、プログラムのロジックだけが書かれた実験プログラムの断片である。例えば図 1 の (1) に

```
try{
```

を追加し、(2) に

```
}catch(Exception e){
  e.printStackTrace();
}
```

などと追加することになるだろう。

```
class Sender{
  public void sendFile(String host,
      String fileName) throws Exception {
    int n;
    int port = 9000;
    byte[] buff = new byte[1024];
```

(1)

```
        Socket s = new Socket(host, port);
        DataOutputStream out
          = new DataOutputStream(
            s.getOutputStream());
        RandomAccessFile file
          = new RandomAccessFile(
            fileName, "r");

        while((n = file.read(buff)) > 0){
          out.write(buff, 0, n);
        }

        file.close();
        out.close();
        s.close();
```

(2)

```
      :
      :
  }
}
```

図 1: 実験プログラム

ところが実験プログラムの場合、実験の内容によって発生する例外をどのように処理したいかが変わる可能性がある。例えば、例外処理の内容を、エラーログを画面に出力することから、実験者にメールを送信するように変えるとする。その場合、プログラム中の全ての catch 節を修正しなければならないが、修正もれがあると、全ての例外がメールによって送信されず、例外に気づくのに遅れてしまう恐れがある。

さらには、例外が起こった時点でプログラムの実行を止めずに、処理全体をやり直したいときもある。大きな実験プログラムを実行する場合、全体の処理時間が長くなるので、途中で例外が発生したからといって、実験を途中で止めて最初からやり直すのは好ましいとはいえない。例えば、サーバとクライアント間でネットワーク障害が起こった場合、時間をおいて再試行すれば障害が解決してうまくいく場合がある。また、クライアントマシンからの応答がな

かった場合は代替のマシンに換えて再試行することで実験を続行できる場合もある。このように、プログラム全体を止めずに適切にリカバリ処理ができれば、それまでの実験の結果を無駄にせずに済む。

しかし、従来の Java 言語の範囲内ではリカバリ処理を記述するのは困難であった。つまり、プログラムの状態を、例外が起こらないような設定に変えて、もう一度同じ処理を繰り返させるような記述は、必ずしも容易ではなかった。例えば、try-catch 文の try ブロックの部分を catch 節の中から再試行したくても、そのような機能は Java 言語にはない。try ブロックの部分をメソッドにして catch 節の中でそのメソッドを呼ぶようにすれば、目的は達成できるが、catch 節の中にまた try-catch 文を書かなければならず、プログラムが見づらくなる。例えば図 1 の場合、

```
try {
  sendFileBody(host, fileName, port,
               buff, n);
} catch (Exception e) {
  host = getAnotherHost();
  try {
    sendFileBody(host, fileName, port,
                 buff, n);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

のようになり、見づらい。なお sendFileBody() は元の try ブロックの中身を実行するメソッドである。

try-catch 文を do-while 文で囲むことでリカバリ処理を実現する方法もある。つまり、図 1 の (1) に

```
Exception e = null;
do {
  try{
```

を追加し、(2) に

```
  } catch (Exception err) {
    e = err;
    host = getAnotherHost(host);
  }
} while (e != null);
```

を追加すれば、リカバリ処理を実装できる。しかし、先の方法と同様、プログラムが見づらくなる。

3 GluonJR

前章の問題点を解決するため、例外処理をアスペクトとして記述できるようにしたアスペクト指向システム *GluonJR* (GluonJ with Recover) を開発している。例外処理をアスペクトとすることで、プログラムロジックから、その中で起こった例外の処理を分離して記述することができるようになる。

3.1 Recovery アドバイス

GluonJR は、AspectJ のような一般的なアスペクト指向システムがもつ機能に加えて、**block ポイントカット** と、**recovery アドバイス**を提供する。block ポイントカットは、2 つのジョインポイントの組を選択するためのポイントカット指定子である。block ポイントカットによって選ばれたジョインポイントの組で囲まれた範囲で例外が発生したときに実行されるコードが、recovery アドバイスである。なお block ポイントカットで指定される範囲は、同一ブロックの中に含まれていなければならない。

例えば、2 章で示したファイルを送信するプログラムの例外処理を、recovery アドバイスを使ってアスペクトとして書くと、図 2 のようになる。実際の GluonJR は Java のライブラリとして実装されており、アスペクトは Java 言語で書くが、図 2 のプログラムは、説明のため AspectJ の文法を真似て書いた。

```
aspect FileSenderRecovery {
  pointcut region():
    block(call(Socket.new(..)),
          call(* Socket.close()))
    && withincode(void Sender.sendFile());

  recovery() throwing (UnknownHostException e):
        region() {
    host = getAnotherHost(host);
    retry();
  }
}
```

図 2: アスペクトの記述例

recovery アドバイスの中では、sendFile() メソッド内の局所変数 host を参照している。このような参照は、他のアスペクト指向システムでは一般的に許されていないが、GluonJR ではこれを許してい

る。また、recovery アドバイスの中では、retry() が呼ばれている。これを呼ぶと、block ポイントカットで指定された範囲の先頭に戻って、その範囲の処理が再実行される。retry() はリカバリ処理の記述のために、GluonJR が提供する特殊なメソッドである。

図 2 のアスペクトを、図 1 のプログラムに織り込む (weave する) と、以下の try-catch 文を使ったプログラムと同等の振る舞いをするプログラムが得られる。

```
class Sender{
  public void sendFile(String host,
      String fileName) throws Exception {
    int n;
    int port = 9000;
    byte[] buff = new byte[1024];

  try{

    Socket s = new Socket(host, port);
    DataOutputStream out
        = new DataOutputStream(
          s.getOutputStream());
    RandomAccessFile file
        = new RandomAccessFile(fileName, "r");
    while((n = file.read(buff)) > 0){
      out.write(buff, 0, n);
    }
    file.close();
    out.close();
    s.close();

  }catch(UnknownHostException e){
      //body of an advice
  }

      :
      :
  }
}
```

3.2 行アノテーション

現実には指定したい範囲の直前や直後に適当なジョインポイントがない場合がある。例えば、下のコードにあるように if 文の直後に for 文が書かれてある場合、if 文の直後すなわち for 文の直前には適当なジョインポイントは存在しないので、for 文の前後を範囲とする recover アドバイスはそのままでは定義できない。

```
if(){
    :
}else {
    :
}
for(){
    :
}
    :
```

このような場合は、GluonJR が提供する**行アノテーション**を用いて指定する。行アノテーションとはユーザ定義のジョインポイントであり、メソッド中の特定の行に対してつけられるアノテーションのことを指す。

将来例外を捕まえる範囲として指定されそうな箇所にあらかじめユーザが目印として行アノテーションを記述しておくことで後でジョインポイントとして利用できる。以下に、上の文の for 文の前後に行アノテーションを付加した例を示す。

```
if(){
    :
}else {
    :
}
@line(position = "begin")
for(){
    :
}
@line(position = "end")
    :
```

4 実装

現在、我々は GluonJR を Java のライブラリとして実装している。retry() や行アノテーションは未実装だが、recover アドバイスの実装の基本部分は完成している。図 2 の FileSenderRecovery アスペクトは、実際には次のように書く。

```
Pointcut pc = Pointcut.and(
  new Pointcut()
    .block(new Pointcut().call("Socket", "<init>"),
         new Pointcut().call("Socket", "close")),
  new Pointcut()
    .withincode("Sender", "sendFile"));

RecoveryAdvice ra = new RecoveryAdvice();
ra.setPointcut(pc);
ra.setThrowingException(
    "UnknownHostException", "e");
ra.setBody("e.printStackTrace();");

Weaver w = new Weaver(ra).weave();
```

このコードはアドバイスの本体を除いて、図 2 と同じことを表している。

現在の実装では recovery アドバイスで指定された 2 つのジョインポイントで囲まれた範囲を try ブロックとし、アドバイスとして書かれたコードを catch 節の中身とした try-catch 文を元のプログラムにバイトコード変換で埋め込む。バイトコード変換には Javassist[2] を利用する。

4.1 バイトコード変換

GluonJR はバイトコード変換によって try-catch 文を挿入することで recovery アドバイスを実現するが、素朴な変換ではうまくゆかない。2 章で示した図 2 の FileSenderRecovery アスペクトでは、call ポイントカットによって、Socket オブジェクトの生成時を表すジョインポイントを選択していた。これはプログラムの次の行に該当する。

```
Socket s = new Socket(host, port);
```

プログラムを Java コンパイラでコンパイルして得られるバイトコードのうち、上の行に対応するバイトコードは以下のようになる。

```
new Socket
dup
aload_1
iload 4
invokespecial Socket()
astore 5
   :
```

aload_1 と iload 4 は、コンストラクタの引数をスタックに積むための命令である。コンストラクタの呼び出しは invokespecial 命令である。

AspectJ に代表される通常のアスペクト指向システムでは、一般に、ジョインポイントを invokespecial 命令の実行時と解釈する。ところが、この命令を try ブロックの始点と考えると、retry() 命令の実現が困難になる。単純に goto 命令などで、invokespecial 命令から実行を再開しようとすると、コンストラクタの引数がスタックに積まれていない状態で、コンストラクタを呼ぼうとしてしまう。これは不正な実行なので、バイトコードを Java 仮想機械にロードする段階で、バイトコード検査器が不正なコードとしてロードを拒否し、VerifyError[1]が投げられてしまう。

この問題を回避するため、GluonJR では、ジョインポイントに該当するソースプログラムの行を構成するバイトコード列の先頭と末尾を、block ポイントカットによって選択される範囲の境界として用いる。先の例では、先頭の new 命令を境界として用いる。あるいは、このジョインポイントが範囲の末尾であるのなら、最後の astore 命令を境界として用いる。これによって、Java 仮想機械のスタックの状態が常に整合性に保った状態であることを保障する。一般的な Java コンパイラが生成するバイトコードでは、行の境界では必ずスタックが空になるので、整合性が保障できる。なお、実現にあたっては、Java クラスファイル内に記録されているソースプログラムの行と各バイトコード命令との対応関係を表す情報を利用している。

4.2 行アノテーション

行アノテーションは、現状では未実装であるが、ソースプログラムを変換して、行アノテーションを空の static メソッド呼び出しに置換することで実現する予定である。呼ばれる static メソッドとして、引数も返値もないメソッドを選べば、メソッドが呼ばれた前後（call の before, after）で自由にジョインポイントを指定することができる。また、メソッドの中身は空のため通常は何の影響もない。

4.3 動的なジョインポイントへの対応方法

ある条件を満たしている時のみアドバイスを実行して欲しい場合がある。静的な try-catch 文を埋め込むと条件に満たしていない場合もアドバイスに当たる catch 節のコードが実行されてしまうことになる。そのような場合、catch 節の冒頭で条件文を判断し、条件に一致しない場合は受け取った例外を再度投げ直す。つまり、

```
recovery() throwing (IOException e):
        region() && if(---){
    //body of this advice
}
```

[1] バイトコードが不正であると検証された場合に投げられるエラー

と書かれてあった場合、以下のようにアドバイスの冒頭で判断をするバイトコードに変換される。

```
try{
  :
}catch (IOException e){
    if (!---){throw e;}
    //body of this advice
}
```

5　関連研究

AspectJ を利用して、プログラム中の例外処理の分離を試みた研究がいくつか提案されている [4][3]。特に Lippert らは、既存のソフトウェア JWAM[1] 中に記述されている例外処理を AspectJ 0.4 で分離することを試みた。JWAM 内に記述されている例外処理はソフトウェア全体に散らばっており、それらの例外処理のコードは互いに類似している。Lippert らは散らばっているこれらの例外処理を、AspectJ を利用して一箇所にまとめることにより、ソフトウェア全体のコードサイズを減らすことができたと報告した。しかし、既存の AspectJ では、例外処理を分離して記述することは可能でも、GluonJR のようなリカバリー処理 (retry) を実現することは困難である。このため、本論文の 2 章で取り上げた実験プログラムの例外処理を AspectJ で記述するのは適切ではない。

また AspectJ のバージョン 0.6 以降から、例外処理に関連する言語機構が 2 つ提供された。それらは `handler` ポイントカット指定子と `after throwing` アドバイスである。`Handler` は、例外ハンドラの実行時、つまり `catch` 節の実行時をジョインポイントとして選択する。今回の実験プログラムのようにあらかじめ例外処理内容が `catch` 節内に定義されていない場合、このポイントカット指定子は有効ではない。一方、`after throwing` アドバイスは、選択されたジョインポイントが例外を投げて異常終了したときに実行される。ただし、このアドバイスは `catch` 節のように例外を捕まえるわけではなく、暗黙のうちに実行される。例外によるメソッドの異常終了の連鎖を止めるわけではない。それゆえ、`after throwing` を利用してリカバリー処理を実装することは困難である。

6　まとめ・今後の課題

6.1　まとめ

本稿において我々は例外処理をアドバイスとして扱えるようにしたアスペクト指向システム GluonJR を提案した。プログラムロジックから例外処理を分離して記述でき、例外処理の中で再試行できるように書ける事を示した。

GluonJR はバイトコード変換で例外処理をプログラムロジックに埋め込む。その時、バイトコード上のジョインポイントを選択するのではなくクラスファイルに含まれる行番号を利用して、つまり、ソースコード中のジョインポイントのある場所を境界としている。そうすることで、`VerifyError` が起こるのを回避することが出来ることを示した。

6.2　今後の課題

`recovery` アドバイスは現段階では、Java のインターフェースが、これを GluonJ と共に動作するように設計中である。また、`retry` メソッドの実装を進めることは今後の課題である。そして、定量的な実験をして有用性を示したい。

参考文献

[1] Jwam framework web site. `http://www.jwam.de/`.

[2] Shigeru Chiba. Load-time structural reflection in java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP '00)*, pages 313–336, London, UK, 2000. Springer-Verlag.

[3] A. Garci F. Filho, C. Rubira. A quantitative study on the aspectization of exception handling. workshop on exception handling in oo systems. In *International Conference on Software Engineering (ICSE)*, pages 418–427, July 2005.

[4] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *International Conference on Software Engineering (ICSE)*, pages 418–427, Jun 2000.