平成13年度学士論文

プログラム分散化のための
アスペクト指向言語

東京工業大学 理学部 情報科学科
学籍番号 98-2072-7
西澤 無我

指導教官
千葉 滋 講師

平成14年2月7日

概要

本稿は、プログラム分散化のためのアスペクト指向言語（AOP）である Addistant を拡張した、Addistant 2 を提案する。Addistant は、利用者が分散に関する記述を独自のアスペクト指向言語で指定することにより、単一の Java 仮想機械（JVM）上で実行することを目的として開発した既存プログラムを、利用者が望む形で機能分散化を導入する。この分散化はロード時に、バイトコードレベルで行われる。Addistant 2 は、分散アスペクトの記述力が十分ではなかった Addistant を、分散アスペクトの Join point の種類を増加することにより、大幅に分散アスペクトの記述力を強化した。また、本稿は Addistant 2 の典型的な利用方法の例も示す。

謝辞

本稿は以下の方々なくして、存在しえなかったでしょう。Addistant の開発、Addistant 2 の提案および本稿の編集になにかと心を砕いていただいた千葉滋講師、東京大学の光来健一氏、筑波大学の立堀道昭氏、横田大輔氏そして研究室のみなさん。心より感謝しています。

目 次

図 目 次

第1章　はじめに

今日、分散ソフトウェア、つまり複数の計算機上で動作するソフトウェアの必要性が高まる一方、その開発にかかるコストが問題になっている。これは、分散プログラムを作成する場合にネットワークなどの分散環境特有の問題に対処しなければならないためである。それらの処理の記述を含んだ分散プログラムは煩雑になり、非分散プログラムの作成に比べて、分散プログラムの作成や維持にかかる人的コストは飛躍的に大きくなりがちである。

分散プログラミングが煩雑であることの要因として、プログラムの分散に無関係な記述の中に分散に関わる処理が拡散して入り交じっている（crosscutting concerns）ことがあげられる。このようなプログラムは可読性が低く、変更も大変である。分散に無関係な記述が分かりにくくなる上、分散に関わる処理を変更するためにはプログラムのあちこちを修正しなければならなくなる。

我々は分散プログラミングをアスペクト指向言語（AOP）により支援する Addistant [1] を開発してきた。ここで、アスペクト指向 [2] とは、オブジェクト指向との相補性を意識した概念である。オブジェクト指向とは、機能性という点に着目して全体をオブジェクトと呼ばれるモジュールに分割していく概念である。しかし、個々のモジュール内には、同様の処理が股がる可能性がある。この個々のモジュールに股がった同様な処理を、オブジェクトとは別の側面から考慮し、それをモジュール化する概念を**アスペクト指向**といい、そのモジュールを**アスペクト**という。

Addistant は次の特徴をもつ。

- Addistant の利用者に、各クラスごとそのオブジェクトを分散環境中のどこに配置するかを指定させる。Addistant では、分散に関わる処理がプログラム全体に拡散して入り交じることを避けるため、利用者は、分散に無関係な記述である非分散プログラムとは別に、分散に関わる記述をまとめて記述する。このまとめて別に書かれた分散に関わる記述を**分散アスペクト**と呼ぶ。

- 対象プログラムのバイトコードを変換して、分散アスペクトによって指定されたクラスが、遠隔ホストで動作している Java 仮想機械（JVM）上で実行されるようにする。Addistant は変換のために、対象プログラムのソースコードを必要としない。変換されたバイトコードは正規の Java バイトコードであり、実行のために特別な JVM を必要としない。バイトコード変換には Javassist[?] を用いた。

- 変換されたバイトコードを Addistant の実行系により遠隔 JVM に自動的に配布される。

Addistant では、遠隔オブジェクト参照を、従来の分散プログラミング・ツールで使われてきたアイディアを組み合わせて実現した。この実装は、**プロキシ・マスタ方式**に基づいたもので、Addistant がバイトコード変換により自動的に行う。

しかしながら、Addistant では、開発が進むにつれ、その利点とともに問題点も明らかになってきた。その問題点とは、分散アスペクトの記述力が十分ではなかったため、利用者が望む機能分散をうまく実現できない場合があることである。そこで我々は、Addistant の分散アスペクトの記述力を大幅に高めた Addistant 2 を開発中である。本稿では Addistant 2 の典型的な利用方法の例をあげ、分散アスペクトの新しい記述方法を説明する。

本稿の残りは、次のような構成からなっている。（以下では、Addistant を Addistant 1 と呼ぶ。）第 2 章は 分散プログラミング用アスペクト言語の必要性と Addistant 1 の性能を述べる。第 3 章では、Addistant 2 の分散アスペクト記述を、第 4 章では、Addistant 2 のこれまでの実装、第 5 章では、Addistant 2 を利用したソフトウェアの機能分散例、第 6 章では、関連研究を、そして第 7 章でまとめを述べる。

第2章　分散プログラミング用アスペクト言語

分散プログラミングの難しさはよく知られており、これまでにも数多くの開発支援ツールが作成されてきた。しかしながら、分散処理に関する記述をどのように書くかについては、今だ研究課題である。

2.1　分離して記述することの必要性

分散プログラムを生成するとき、分散に関する記述は、分散と無関係な記述から完全に独立させ、一カ所にまとめるべきであると我々は考えている。例えば、分散化するオブジェクトのクラス名やネットワーク通信に必要なホスト名、ポート番号は、アプリケーションのロジックと独立して記述されるべきである。その理由として、記述が分離されているとプログラムの保守性が高まることがあげられる。分散環境の修正、変更があった場合には、まとめられた分散に関する記述だけを修正、変更すればよく、保守のための時間が大幅に削減できる。もし仮に、分散に関する記述がソースコード内に散らばっていると、わずかな分散環境の変更に対してでさえ、わざわざソースコード全体を細かく調べ、修正していかなくてはならない。この修正は非常に時間がかかり、間違いを犯しやすいため、細心の注意を払いながら行わなければならない。

しかしながら、分散に関する記述を独立に書くことは決して容易ではない。通常の分散プログラムでは、分散に関する記述と、分散とは無関係な記述が複雑に入り交じっている（crosscutting concerns）ためである。例えば、分散に関する記述を単純に分離しただけの差分パッチにするだけでは不適当である。例えば、次のような記述を考える。

```
<patch source="FrameCaller.java">
    line="312"
    FrameProxy fp = new FrameProxy();
```

```
    line="320"
    FrameProxy fp = new FrameProxy();
</patch>
```

ここで FrameCaller.java は分散とは無関係な処理を記述したファイルである。この記述は「FrameCaller.java ファイルの行番号 312 と 320 の Frame クラスのオブジェクト生成式を、分散処理用の FrameProxy クラスのオブジェクト生成式と置き換える」という意味である。このような記述では、もし FrameCaller.java を手直しした場合、行番号が変ってしまい分散に関する記述までも書き換えなければならなくなる。このように、わずかな変更で無効になる記述では実用にならない。

2.2　Addistant 1

我々は、分散アスペクト記述に従い、既存 Java プログラムを自動的に機能分散させることのできるシステム Addistant 1 を既に提案している。Addistant 1 の分散アスペクトは、分散に関わる記述を単純に分離しただけでなく、十分に抽象化された記述である。ここで、Addistant 1 の分散アスペクトの記述例を以下に示す。

```
<import proxy="rename" from="app">
    subclass@java.awt.Component              </import>
```

この例は、「Componet クラスとそのサブクラス全てのオブジェクトを、変数 app で指定されるホストに配置させる。また、遠隔オブジェクト参照を『rename』という手法を用いて実装する。」という意味の指定である。また、以下のような記述も可能である。

```
<import proxy="subclass" from="rmt">
    subclass@java.swing.*                    </import>
```

この例は、「Swing パッケージに含まれるクラスとそのサブクラス全てのオブジェクトを、変数 rmt で指定されるホストに配置させる。また、遠隔オブジェクト参照を『subclass』という手法を用いて実装する。」という意味の指定である。

Addistant 1 の分散アスペクト言語の要素は、以下の 2 種類である。

- 遠隔オブジェクト参照の実装方法を指定できる。Addistant 1 は遠隔オブジェクト参照をプロキシを使って実現するが、プロキシの実装方法には『replace』『rename』『subclass』『copy』の 4 種類がある。この中から、クラスごとに 1 つを選べる。その理由は、それぞれの手法には固有の適用可能な制約があり、単一の手法を選んでプログラム全体にその手法を用いることができないからである。
- オブジェクトを、ローカルと遠隔の 2 つのうちどちらのホストに配置するか、クラス単位で指定することができる。

Addistant 1 では分散アスペクトを XML 風に記述し、独自の解析器を用いて、分散に関するロジックを解析し、ソースプログラムを変換している。

2.3　Addistant 1 の限界

上で述べたように、Addistant 1 ではオブジェクトは、そのオブジェクトを定義しているクラス単位で、ローカルと遠隔のどちらかのホストに配置される。しかしながら、このようなオブジェクト配置の指定では、自動分散化が可能な既存プログラムに制限が生じる。

例えば、異なるホストを利用する 2 人に同じ画面を見せるアプリケーションを考える。このアプリケーションの実装としては、プログラム内に 2 つの JFrame[12] オブジェクトを生成する必要があるだろう。さらに、具体的に実装を考えると、createViewerFrame() メソッド内で JFrame のオブジェクト viewerFrame を、createEditorFrame() メソッド内で JFrame のオブジェクト editorFrame をそれぞれ生成し、表示させるようなプログラムになるだろう。そして、Addistant 1 はこのプログラムを、次のように機能分散させなければならない。

> 2 つのオブジェクトのうち viewerFrame は手元のホストで生成し、手元のディスプレイに表示し、editorFrame は遠隔のホストで生成し、遠隔のディスプレイに表示させる。

残念ながら、Addistant 1 では、このような機能分散をすることができない。なぜならば、Addistant 1 のオブジェクトの配置はクラス単位でしか指定することができないため、同じ JFrame クラスのオブジェクトである、viewerFrame オブジェクトと editorFrame オブジェクトはともに同一ホスト上にしか生成できないのである。

第3章　Addistant 2 - 記述力強化版

我々は 2.3 節の例のようなアプリケーションも分散化できるように、Addistant 1 を改良した Addistant 2 を開発中である。Addistant 1 は、分散に関するアスペクトと、分散とは無関係のアスペクトの『接点』の種類が少なかったため、分散アスペクトに記述できる事柄の範囲が狭かった。そこを Addistant 2 では改善し、アスペクト間の『接点』の種類を増加させ、分散アスペクトのより詳細な記述を可能にした。このアスペクトとアスペクトの『接点』をアスペクト指向では、一般に Join point と呼ぶ。

Addistant 2 で、最も大切な Join point の拡張は、より詳細なオブジェクト配置の指定を可能にした点である。2.2 節で述べたように、Addistant 1 のオブジェクト配置は、クラス単位でしか指定できなかった。Addistant 2 では、既存コード中の new 式のあるクラス、メソッドの種類といった、オブジェクトの生成文脈に応じた、オブジェクトの配置を指定することができる。

それ以外の追加された Join point の種類としては

- システムの実行環境の属性
- ORB 実装
- ネットワーク・ストリーム実装

があげられる。Addistant 2 の分散アスペクト記述では、これらの Join point を XML を使って細かく指定できる。以下では、上記の Addistant 2 の Join point を説明し、分散アスペクトの記述例を詳しく解説していく。

3.1　オブジェクトの配置

ローカルと遠隔のどちらのホストにオブジェクトを配置し動作させるか、という指示を、クラス単位の指定だけでなく、既存コード中の new 式のあるクラス、メソッドの種類といった、オブジェクトの生成文脈に応じて、細かく指定することができる。

Addistant 2 の、オブジェクト配置の記述の例を以下にあげる。

```
<import proxy="subclass" in="FrameClient.createViewerFrame()"
                         from="rmt">
    <subclass name="javax.swing.JFrame"/>
</import>
```

この分散アスペクトの記述は、「FrameClient クラス内の createViewer-Frame() メソッド中で生成される JFrame クラスとそのサブクラスは変数 rmt で指定されたホストに配置させる。また、遠隔オブジェクト参照は、『subclass』手法を用いて実装する」という事を意味している。Addistant 2 で新たに導入した in 属性は、その値にクラス名だけでなく、メソッド名まで指定することができ、in 属性値で指定された範囲が、オブジェクト配置の指定の有効範囲となる。

3.2　システムを実行する環境の属性

システムの実行環境とは、ネットワークでつながれたホスト名や、ネットワーク通信でソケットを張るためのポート番号、また手元にないクラスファイルをネットワーク越しにクラスローダが探してくる際の、検索場所（ディレクトリ）のことである。

以下では、具体的に分散アスペクト中で実行環境を記述する例を示す。

```
<start name="app">
    <host name="picard" rmiPort="14004" bytecodePort="14005"
          searchDir=apprun"/>
</start>
<remote name="rmt">
```

```
    <host name="taro" rmiPort="14006" bytecodePort="14007"
          searchDir="rmtrun"/>
</remote>
```

この記述は大きく分けて、ノード値 start のノードと、ノード値 remote のノードの 2 つに分割できる。前者は変数 app で指定されるホストの情報を記述するノードであり、後者は変数 rmt で指定されるホストの情報を記述するノードである。さらに、ホストの情報はノード値 host が表す。ノード値 host の中身を更に細かく見ると、ホスト名（name 属性)、ポート番号（rmiPort,bytecodePort 属性)、ディレクトリ（searchDir 属性）の 4 つの属性からなり、これらを分散アスペクト記述により指定することができる。

3.3　Object Request Broker の実装

Object Request Broker（ORB） とは、遠隔オブジェクトを生成したり遠隔メソッドを呼び出すなど、遠隔オブジェクトに対するネットワーク越しの要求の処理を定義しているクラスである。ユーザ定義の ORB クラスを分散アスペクトに記述することで、指定できる ORB の種類を拡張することができる。ただし、ユーザ定義の ORB クラスは RequestBrokerable インターフェイスを implements していなくてはならない。ORB クラスの指定が分散アスペクト内に記述されていないときは、Addistant 2 標準の ORB クラスを用いてシステムが実行される。

この機能を用いて、様々な ORB を利用することが可能である。例えば、システムの実行中にネットワークが切断されたときにも動作を続けられるよう、耐故障性を備えた ORB などが考えられる。例えば、変数 rmt で指定される複数のホストにオブジェクトのコピーを保持させる多重化 ORB を指定するには、次のように分散アスペクトを記述する。

```
<orb classname="RequestMultiplyingBroker"/>
```

RequestMultiplyingBroker は、Addistant 2 で提供される RequestBrokable インターフェイスを実装したクラスである。この ORB を利用する場合、変数 rmt には複数のホストを指定するが、これには次の 2 種類がある。

- 直接、変数 app で指定されるホストとデータのやり取りをしている、ただ 1 つのホスト。このホストは app で指定されるホストからの遠隔メソッド呼び出しを、オブジェクトのコピーをもつ他のホストへ、多重化して配送する役目をもつ。このホストを「遠隔リーダーホスト」と呼ぶ。

- 「遠隔リーダーホスト」から多重化された遠隔メソッド呼び出しを受け取って処理するホスト。

現在の実装では、最初に変数 rmt に代入されたホストを「遠隔リーダーホスト」とみなす。

3.4　ネットワーク・ストリームの実装

分散アスペクトの ORB と同様、ネットワーク・ストリームの実装クラスもユーザ定義のクラスで拡張することができる。この場合、ユーザが定義したクラスは CommChannel クラスを extends しなくてはならない。

分散用アスペクト言語で、ネットワーク・ストリームの実装の指定方法を記述する例を以下に示す。

```
<channel classname="SSLCommChannel"/>
```

これは通信チャンネルを暗号化させるクラスにより、ネットワーク・ストリームを拡張している。

第4章　これまでの実装

Addistant 2 の実装は、Addistant 1 の仕様に基づいているが、Join point の増加により、分散アスペクト記述を強化するため実装し直している。以下では Addistant 2 の実装で、特に重要だった技術について記述する。

4.1　遠隔ホスト上のクラスファイルをロードする

4.1.1　クラスローダによるプログラムの自動配布

Java のクラスは、デフォルトクラスローダがクラス情報を定義しているバイトコードから生成する。クラスローダのクラスは java.lang.ClassLoader である。アプリケーションでクラスが必要になったとき、デフォルトクラスローダがバイトコードを読み込みクラスをロードする。

ユーザ定義のクラスローダ（java.lang.ClassLoader のサブクラス）を記述することによって、任意のバイトコードからクラスをロードすることが可能になる。ユーザ定義のクラスローダを利用するには、クラス名を指定しユーザ定義のクラスローダからクラスを取得する。例えば、クラス MyClass を ユーザ定義のクラスローダ MyClassLoader からロードし、foo() メソッドを実行するには、以下のようなコードを実行する。

```
ClassLoader loader = new MyClassLoader();
Class c = loader.loadClass("MyClass");
Method m = c.getDeclaredMethod("foo", null);
m.invoke(null, null);
```

これは標準 Java reflectionAPI によるメソッド呼び出しの方法 [4] であるが、ユーザ定義のクラスローダは定義された処理に従いクラスをロードする。ユーザ定義のクラスローダにはクラス名が渡されるのでクラス名

から処理を判別し、ローカルのファイルシステムやネットワーク上からバイトコードを取得しクラスをロードすることができる。Addistant 2 ではオブジェクトの分散配置を実現するため、ユーザ定義のクラスローダを必要とする。

4.1.2　Javassist の概要

Javassist は、構造リフレクション [5]（structural reflection）の機能を提供するクラスライブラリである。構造リフレクションとは、クラスや関数などのデータ構造の定義を必要に応じて変更できるようにする機能である。Java は標準 Java API の一部としてリフレクションの機能を提供するプログラミング言語である。しかしながら、提供される機構はプログラム中で用いられるデータ構造、すなわちクラス、の定義を調べる機能（introspection）が主で、プログラムの振る舞いを変更する機能は非常に限られている。Java のリフレクション機能を強化するために、これまでいくつかのシステムが提案されてきたが、そのほとんどは動作リフレクション（behavioral reflection）の機能を提供している。これはメソッド呼び出しのような演算を横取りして、その動作を変更できるようにするものである。

我々はリフレクションの機能拡張を実装するには、動作リフレクションよりも構造リフレクションの方が適していると考える。構造リフレクションは、そのような言語拡張に必要な機能を直接提供するので、実装は容易に実現できる。また、動作リフレクションは構造リフレクションさえ提供していれば、その上に動作リフレクションを実現し、それを使って目的の言語拡張を実装することも可能である。

Javassist は構造リフレクションを実現するにあたり、クラスを JVM にロードする際にバイトコード変換を行なうことで実現する。従来知られている方法では、JVM の内部を変更する必要があったが、可搬性が重要な Java 言語ではこの方法は現実的でない。またソースコード変換器を使って実現する方法では、ソースコードなしでは構造リフレクションが利用できないという問題があった。Javassist で採用したバイトコード変換による方法では、このような問題を回避できる。

Addistant 2 では、構造リフレクションを実現するために、Javassist のバイトコード変換を利用している。

4.1.3　Javassist を用いたバイトコード変換

Addistant 2 では、クラスローダによりローカルのファイルシステムや遠隔ホスト上から取得したバイトコードにバイトコード変換を行ない、変換後のバイトコードからクラスを生成している。そのクラスは、バイトコード変換を行なった処理が反映される。バイトコード変換は、クラスに特定の処理を追加することや処理を変更すること、あるクラスを別のクラスとしてロードすることなどが可能になる。（以下の図を参照）

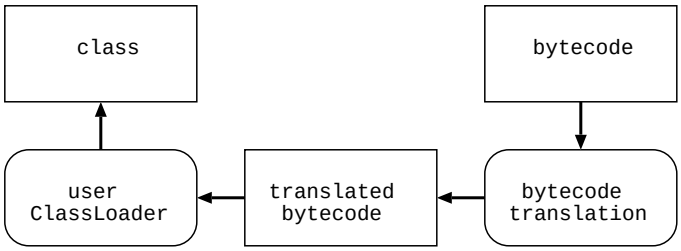


図 4.1: Bytecode acquisition and translation

Addistant 2 は、上記のような、バイトコード編集には Javassist を用いている。Javassist では、バイトコードからクラス情報を取得することやバイトコードを編集することができる。バイトコード編集の手順は、はじめにバイトコードから CtClass （compile-time class）オブジェクトを生成する。そして CtClass オブジェクトに変更を加える。変更を加えた CtClass オブジェクトからバイトコードを取得すれば、変更が反映されたバイトコードが取得できる。Javassist は CtClass オブジェクトを変更するための API を提供しているので、CtClass オブジェクトへの変更は容易に行なうことができる。

Javassist API の提供している処理の一部についてあげる。

- クラス情報の取得
- 新しいクラスの生成
- コンストラクタ、メソッド、フィールドの追加、変更
- クラス名、スーパークラス、実装インターフェイスの変更
- クラス内部で使用しているクラス名の変更

次の 4.2 節で Addistant 2 の遠隔オブジェクト参照の実装で Javassist を用いたバイトコード変換の具体例を説明する。

4.2　遠隔オブジェクト参照を実行する

Addistant 2 には、遠隔ホストで、オブジェクトの生成、メソッドの呼び出しをローカルホスト上のプロセス内部であるかのように対話する機能を提供している。これらの処理を実行するため、遠隔ホストへリクエストを送受信するには、標準 Java API の java.io パッケージや java.net パッケージを利用し通信プロトコルを Addistant 2 独自に生成する必要がある。

この節では、一般的な分散プログラミング支援ツールが遠隔オブジェクトを生成、参照するために必要なプロトコルと実装を示し、Addistant 2 の遠隔オブジェクト参照の実装で利用したプロキシ・マスタ方式 [6] について説明する。

4.2.1　遠隔オブジェクト生成と参照の実装

遠隔ホスト上にオブジェクトを生成するためには、クラスを参照し、そのクラスのコンストラクタ引数の型と値を送信する機能が必要である。これは遠隔ホスト上でオブジェクトの生成が、標準 Java の reflection API を利用しているためである。標準 Java の reflection API を利用すれば、任意のクラスに対してオブジェクトを生成することができる。もし逆に、Addistant 2 が reflection API を利用せずにオブジェクトの生成をしていたとすると、オブジェクト生成側のコードは、オブジェクトの元になるクラスに依存してしまう。このため、違うクラスのオブジェクトを生成するごとにオブジェクト生成側のコードを修正しなくてはならないため、プログラムの保守性が低下してしまうのである。

生成された遠隔オブジェクトの参照は、遠隔メソッド呼び出しの際に必要になるため、いつでも取り出せるようになっていた方が望ましい。そのため、オブジェクトの登録領域を用意しそこに格納しておき、クライアントには格納している場所を表すデータを返しておく必要がある。Addistant 2 では、この問題をオブジェクト参照テーブルを構築することにより、解決している。

具体的には、クライアントが遠隔ホスト側に新しいオブジェクトを生成

するよう要求を出すと、オブジェクトを生成するプログラムである RMIServer は、下記のような遠隔オブジェクト生成のための通信プロトコルを利用して遠隔オブジェクトを生成する。そのオブジェクト参照を RMIServer は、オブジェクト参照テーブルに登録し、登録に必要な objectID（key）をクライアントに送信する。

以下は、遠隔オブジェクト生成とその参照に必要な、クライアント側の通信プロトコルの実装と遠隔ホスト側のオブジェクト生成プログラムである、RMIServer の通信プロトコルの実装を記す。

```
/** Client **/
ObjectOutputStream out
    = new ObjectOutputStream(socket.getOutputStream());
out.writeObject(className);
out.writeObject(parameterTypes);
out.writeObject(parameterValues);

int objectID = readInt();
```

クライアントは、クラス名（String 値の className）、コンストラクタ引数の型（String[] 値の parameterTypes）、コンストラクタ引数の値（Object[] 値の parameterValues）を通信プロトコルとして送信すればよい。クライアントは RMIServer から遠隔オブジェクトを参照するために必要な objectID（key）を受信することにより、いつでもその遠隔オブジェクトを呼び出すことができる。

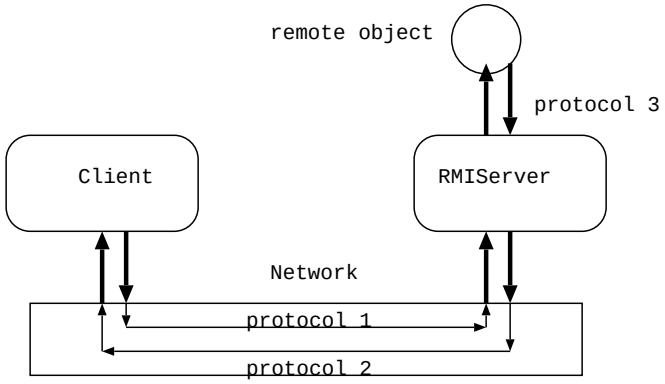
次に、遠隔ホスト上のオブジェクト生成プログラム（RMIServer）の通信プロトコルは以下となる。

```
/** RMIServer (remote object creater) **/
ObjectInputStream in
    = new ObjectInputStream(socket.getInputStream());
String className = (String) in.readObject();
String[] parameterTypes = (String[]) in.readObject();
Object[] parameterValues = (Object[]) in.readObject();
```

```
Class[] paramTypes = getClassArray(parameterTypes);
Class targetClass = loader.loadClass(className);
//loader is a value of ProxyClassLoader that extends ClassLoader.
Constructor cons = targetClass.getConstructor(paramTypes);
Object obj = cons.newInstance(parameterValues);

int objectID = ((Integer) table.registerObject(obj)).intValue;
out.writeInt(objectID);
```

RMIServer はクライアントから生成するオブジェクトのクラス名（String 値の className）、コンストラクタ引数の型（String[] 値の parameterTypes）、コンストラクタ引数の値（Object[] 値の parameterValues）を受け取って、標準 Java の reflection API により指定されたオブジェクトを生成している。その後、RMIServer はオブジェクト参照テーブル table に生成したオブジェクト参照を登録し、その objectID（key）をクライアントに送信している。以下の図は、クライアントが RMIServer を通じて遠隔オブジェクトを生成するための順序を表す。




```
protocol 1: Client sends "className", "paramTypes" and "paramValues"
            to RMIServer, then RMIServer receives those.
protocol 2: RMIServer sends objectID(key) to Client,
            then Client receives it from RMIServer.
protocol 3: The created object is put on ObjectReferenceTable
            by objectID(key).
```

図 4.2: remote object creation protocol

ここで オブジェクト参照テーブルとは、オブジェクト参照から objectID（key）を参照することができ、かつ objetID（key） からもオブジェクト参照を取り出すことができる実装のテーブルのことであるが、標準 Java API の java.util.Hashtable クラスや HashMap クラスをこのオブジェクト参照テーブルに利用することができない。なぜならば、Hashtable や HashMap では、そのオブジェクト参照がテーブルに登録されているかどうかを検索するため（例えば、Hashtable クラスの contains() メソッドや HashMap クラスの containsValues() メソッドなど）に java.lang.Object クラスの equals() メソッドを利用して、登録されているオブジェクト参照とそのオブジェクト参照を一つ一つ比べていくのである。

Hashtable クラスや HashMap クラスがオブジェクト参照テーブルに利用できない理由を述べる前に、以下の例について考える。

```
Integer ref1 = new Integer(1);
Integer ref2 = new Integer(1);

if (ref1.equals(ref2)) {
    System.out.println("equals: true");
} else {
    System.out.println("equals: false");
}
if (ref1 == ref2) {
    System.out.println("  ==  : true");
} else {
    System.out.println("  ==  : false");
}
```

これは、java.lang.Object クラスの equals() メソッドによるオブジェクト参照の等価と、== によるオブジェクト参照の等価を比較する例である。ref1 と ref2 ともに java.lang.Integer クラスの同じ引数をもつオブジェクト（これを同値なオブジェクト[1]という）参照であるが、このプログラムを実行すると結果は以下のようになる。

[1]同値なオブジェクトとは、同値関係、つまりオブジェクトの反射性、推移性、整合性を互いに満たすオブジェクトのことである。

```
equals: true
  ==  : false
```

この例の結果より、java.lang.Object クラスのメソッド equals() は、同値なオブジェクトを参照していれば true を返すのに対して、== は同値なオブジェクトであっても、参照が異なれば false になる、ということを意味している。先に Hashtable や HashMap クラスのオブジェクト参照の検索は java.lang.Object クラスの equals メソッドが利用されていると記述したが、Addistant 2 の仕様ではオブジェクト参照のテーブルには同値なオブジェクトが複数登録される可能性がある。例えば、同値なオブジェクト参照 A,B が同じオブジェクト参照テーブルに登録されていて、それぞれの key が Integer(1),Integer(2) であったとする。そこでオブジェクト参照 B から key を取り出したいとすると、equals() メソッドを利用して検索していたとしたら Integer(1) と Integer(2) のどちらかを返すことになってしまう。このため、オブジェクト参照の検索に、== を利用するテーブルを構築しなくては、オブジェクト参照のアイデンティティ、つまりオブジェクト参照とその key の一対一対応が保持できない。この問題を解決するため、Addistant 2 では標準 Java API の java.util.ArrayList クラスを利用してオブジェクト参照テーブルを構築した。

上記の実装によりクライアントは生成した遠隔オブジェクトの参照を保持、利用して、遠隔オブジェクトのメソッドを呼び出すことができる。

4.2.2　遠隔メソッド呼び出し

遠隔メソッドを呼び出すためには、メソッドの引数の型と値をネットワーク越しに送信する機能が必要である。この理由も、遠隔オブジェクトの生成と同様、メソッド呼び出しにも 標準 Java の reflectionAPI が利用されているためである。Addistant 2 は、メソッド呼び出しを reflection で記述することにより、任意のクラスのメソッドをオブジェクトの元になるクラスに依存することなく、呼び出すことができる。

以下は、遠隔メソッド呼び出しに必要な、クライアント側の通信プロトコルの実装を記す。

```
/** Client **/
ObjectOutputStream out
    = new ObjectOutputStream(socket.getOutputStream());
ObjectInputStream in
    = new ObjectInputStream(socket.getInputStream());
out.writeInt(objectID);
out.writeObject(className);
out.writeObject(methodName);
out.writeObject(parameterTypes);
out.writeObject(parameterValues);

Object result = in.readObject();
```

クライアントは、オブジェクト参照テーブルからオブジェクト参照を取り出す int 値の objectID（key）、String 値のクラス名 className、String 値のメソッド名 methodName、メソッド引数の型（String[] 値の paramTypes）、メソッド引数の値（Object[] 値の paramValues）をメッセージプロトコルとして送信し、そのメソッドの結果が戻ってくるのを待つ。一方、遠隔メソッドを呼び出すプログラム（RMIServer）は以下のようになる。

```
/** RMIServer (remote method invocation) **/
ObjectInputStream in
    = new ObjectInputStream(socket.getInputStream());
Integer objectID = new Integer(in.readObject());
String className = (String) in.readObject();
String methodName = (String) in.readObject();
String[] parameterTypes = (String[]) in.readObject();
Object[] parameterValues = (Object[]) in.readObject();

Class[] paramTypes = getClassArray(parameterTypes);
Object targetObj = table.referObject(objectID);
Class targetClass = loader.loadClass(className);
//loader is a value of ProxyClassLoader that extends ClassLoader.
Method method = targetClass.getMethod(methodName, paramTypes);
```

```
Object result = method.invoke(targetObj, parameterValues);
```

RMIServer は、クライアントからのメッセージプロトコルを受信し、標準 Java API の java.lang.reflect.Method クラスにより指定のメソッドを呼び出す。その後、メソッドの実行結果をクライアントに送信する。

4.2.3　プロキシ・マスタ方式

Addistant 2 では、上記の遠隔オブジェクト（マスタオブジェクト）へのアクセス（遠隔オブジェクトの生成や遠隔メソッドの呼び出し）を、ローカルホストに生成したプロキシオブジェクトにアクセスすることで実現している。プロキシオブジェクトはマスタオブジェクトの代理として働く。つまり、4.2.1 節や 4.2.2 節のような遠隔オブジェクト生成や遠隔メソッド呼び出しに必要なクライアントの操作をすべてプロキシオブジェクトに埋め込むのである。このプロキシ・マスタ方式により、ユーザはネットワークを意識することなく、分散オブジェクトを参照することが可能となる。

以下の図 4.3 は、非分散時のクライアントとオブジェクトのアクセスと、分散時のクライアントとオブジェクトのアクセスを示す。図に示すように、クライアント A とオブジェクト B があるとする。非分散時にはクライアント A は直接 オブジェクト B にアクセスすることが可能である。しかし、分散時オブジェクト B はネットワーク越しに存在するため、直接アクセスすることはできない。そのため、クライアント A はプロキシオブジェクト B へアクセスを要求する。アクセスされたプロキシオブジェクト B の内部では 4.2.1 節や 4.2.2 節で解説した、クライアントの役割をする Object Request Broker（ORB）が内蔵されており、プロキシオブジェクト B は ORB とネットワークを経由してマスタオブジェクト B へアクセスする。

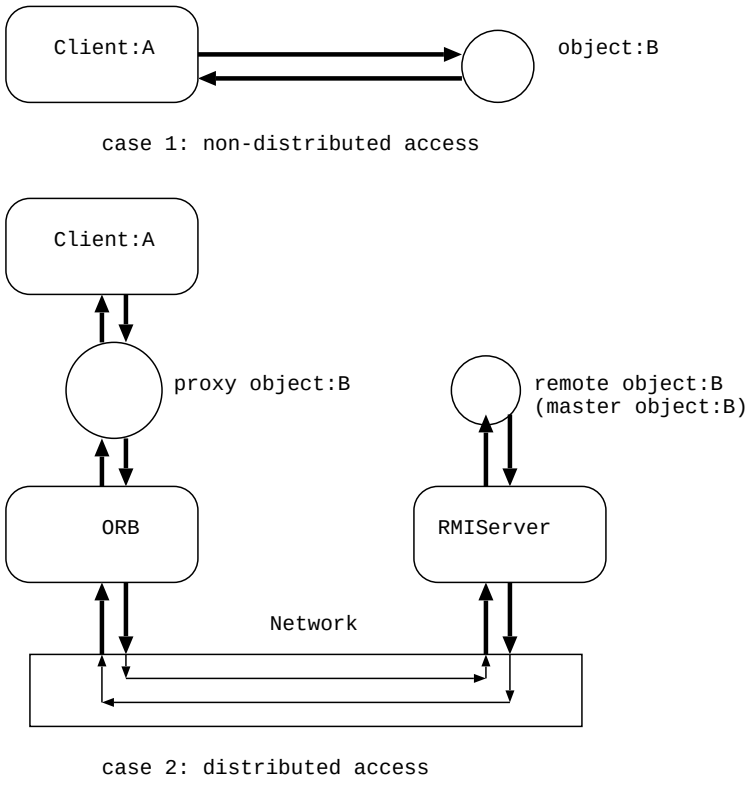


図 4.3: message protocol with proxy/master model

ここで、理想のプロキシクラスのプログラムを記述する。マスタオブジェクトの例として標準 Java API の java.awt.Frame クラスを取り上げる。既存の Frame クラスのコンストラクタとメソッド setTitle() の概要は、以下のコードである。

```
/** java.util.Frame.java **/
public Frame(String title) {
    this(title, ...);
}

public void setTitle(String title) {
    //sets title.
    this.title = title;
    ... ...
```

```
}
```

このマスタクラス Frame にアクセスするプロキシオブジェクト FrameProxy は、以下のようなコードになることが望ましい。

```
/** FrameProxy.java **/
RequestBroker orb = new RequestBroker();

public FrameProxy(String title) {
    int objectID = orb.createRemoteObject("Frame", "String", title);
    table.registerObject(this, objectID);
}

public void setTitle(String title) {
    int objectID = table.referObject(this);
    Object result = orb.invokeRemoteMethod(objectID, "Frame", "setTitle",
                                           "String", title);
    return ;
}
```

ここで RequestBroker とは、4.2.1 節や 4.2.2 節の遠隔オブジェクトの生成や遠隔メソッドの呼び出しのためのメッセージプロトコルを createRemoteObject(),invokeRemoteMethod() メソッド内に埋め込んだクラスである。プロキシオブジェクトは常に処理をマスタオブジェクトへ渡すため、クライアントはマスタオブジェクトが同一ホスト上にあるかのように扱うことができる。

プロキシ・マスタ方式を実装する際の問題

Addistant 2 では、上記のように遠隔オブジェクト参照の実装をプロキシ・マスタ方式に基づいて設計している。しかし、既存プログラムをプロキシ・マスタ方式の実装により、機能分散させるためには、いくつか問題点があげられる。以下にそれを記述する。

- プロキシオブジェクトを定義するクラスは、本来存在していない。どのようにプロキシクラスを生成すればよいか。
- 既存プログラムで、遠隔オブジェクト（マスタオブジェクト）を生成していたコードを、プロキシオブジェクトを生成するコードに変換しなくてはならない。

Addistant 2 は、これらの問題を Javassist を用いたバイトコード変換により、解決する。

プロキシクラスの生成

Addistant 2 では、プロキシオブジェクトの生成をロード時のバイトコード変換により実現している。バイトコード変換には Javassist を用いている。Addistant 2 は、以下の図 4.4 のように、プロキシ生成クラス（ProxyGenerator）を定義して、マスタクラスとテンプレートクラス（TemplateProxy）から、それぞれクラス情報とプロキシクラス情報を取得してバイトコード変換を行なっている。テンプレートクラスは Addistant 2 で、あらかじめ提供しているクラスで、プロキシクラスに必要な処理を格納している。

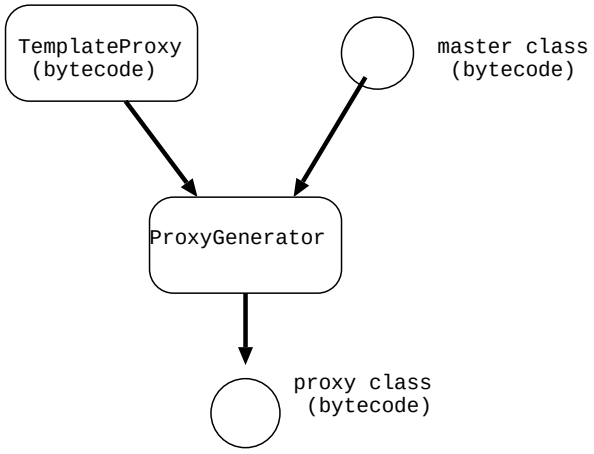


図 4.4: proxy object creation used to bytecode translation by Javassist

以下の図 4.5 では、Addistant 2 が先ほど例として取り上げた、Frame クラスのプロキシクラス FrameProxy のコードを生成する処理の流れを示

している。マスタの Frame クラスのメソッド setTile() は、引数の String オブジェクトをフレームタイトルとして設定する。それに対し、プロキシオブジェクトではフレームにタイトルを設定する処理は行なわない。ORB オブジェクトのメソッドを呼び出している。プロキシクラスに、マスタクラスのもつメンバメソッド名、引数の型とテンプレートクラスのもつメンバ本体を追加している。

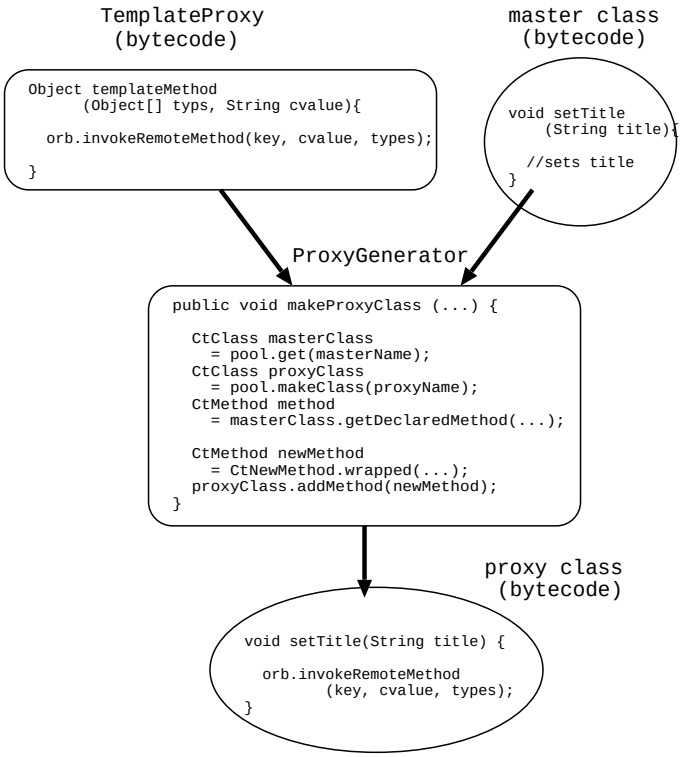


図 4.5: sample: FrameProxy creation by Addistant 2

上記のバイトコード変換により、遠隔のマスタオブジェクトを自動的に参照するプロキシオブジェクトを生成することができる。

クライアントクラスのオブジェクト生成式の置換

Addistant 2 では、クライアントクラス内部で使用している遠隔のマスタクラス名からローカルのプロキシクラス名への変更をロード時のバイトコード変換により実現している。バイトコード変換には Javassist を用

いている。Javassist では、オブジェクトの生成（new）式を異なるオブジェクトの生成式にバイトコード変換を可能にする replaceNew() メソッドが存在する。このメソッドは、マスタクラスのバイトコード、マスタオブジェクトを生成するクライアントクラスのバイトコード、さらにプロキシクラスのバイトコードが手に入っていれば、利用することができる。ただし前述していなかったが、プロキシクラスは、そのバイトコード生成時にマスタクラスをインターフェイスとして implements する必要がある。

このバイトコード変換により、既存クライアントプログラムのマスタオブジェクトの生成式を、自動的にプロキシオブジェクトの生成式に置き換えることが可能となる。

4.3　RMIServer のスレッドの同期

RMIServer は、遠隔オブジェクトの生成や参照の保持、遠隔メソッドの呼び出しを実行するためのプログラムであり、遠隔ホスト上で実行されるサーバとして実装すべきである。また、Addistant 2 内のプロセスで何度も呼ばれることになるため、RMIServer はマルチスレッド化することが望ましい。Java では、マルチスレッドプログラミングが可能である。スレッド間の同期に対しては ”synchronized” によって同期を取ることができる。

この節では、RMIServer が synchronized を伴ったメソッドを呼び出す際の問題点を提起し、その問題を解決した実装方法を記述する。

4.3.1　synchronaized とコールバック

スレッド間の同期に対しては ”synchronized” によって同期を取ることができる。synchronized とは、インスタンスをスレッド間で排他的に利用するための宣言である。他のスレッドがすでにそのインスタンスを利用しているときに、別のスレッドがこのメンバメソッドを実行しようとすると、前者の処理が終わるまで、後者は実行を待たされるようになる。排他の範囲はメソッドではなくインスタンスである。synchronized を利用する場合の例としては、オブジェクトに複数の関連のある値があった時に、これらをあるスレッドが書き換えている最中、別のスレッドが読み出してしまうと、変更途中の誤った値を読んでしまう事になる。このよ

うなときに、synchronized 宣言を利用して読み書きが同時に起らないようにするべきである。synchronized はメソッドを排他的に実行するための宣言ではない。インスタンスが異なれば待たされずに同時に実行されるし、synchronized を付けた別のメソッドであっても同じインスタンスに対するものならば同時に実行されない。また、synchronized 宣言を静的な（static な）メソッドに付けた場合は、そのクラスの Class オブジェクトが排他の範囲となる。つまり、クラスを排他的に利用する宣言になる。例えば、次のような synchronized を伴ったメソッド myMethod があるとする。

```
synchronized void mySyncMethod() {
    // ... do Something ...
}
```

以下の図 4.6 のように、このメソッドに対し異なるスレッド A,B 上から同時にアクセスすることはできない。スレッド A が mySyncMethod を実行すると、スレッド B はスレッド A の処理が終わるまで待ち状態となる。

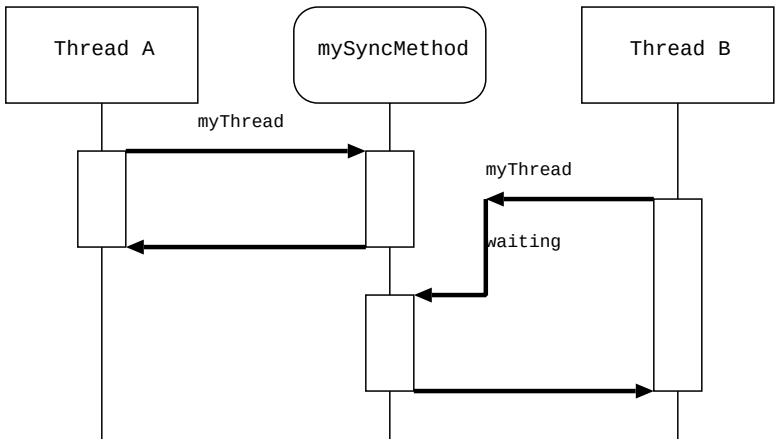


図 4.6: effect of synchronized keyword

コールバックとは、オブジェクト A からオブジェクト B のメソッドを呼び出したときに、オブジェクト B のメソッド内でオブジェクト A のメ

ソッドを呼び出しているような場合である。下図のよう、メソッドの返り値が返ってくる前にコールバックがくる。コールバックでは、オブジェク

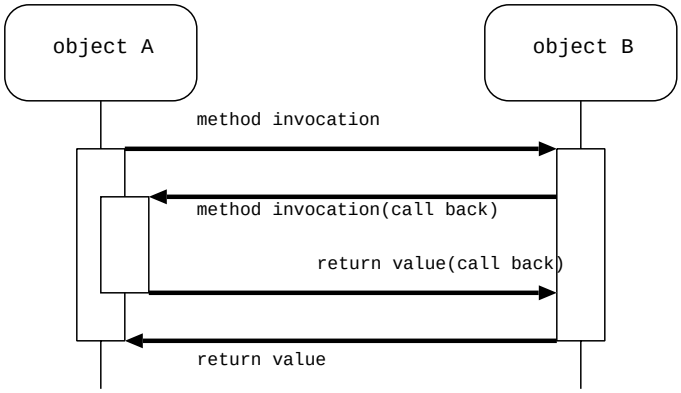


図 4.7: call back

ト A でメソッド処理中に他のメソッドが処理を行なう。synchronized があっても、同じスレッド上の処理であるため待ち状態にはならない。しかし、分散時にはスレッドの処理を正確に行なわないとコールバックの処理でデッドロックが発生することがある。

4.3.2　分散時のコールバック

分散時のコールバックでは、RMIServer を経由しコールバックが行なわれる。RMIServer ではソケット間通信により遠隔メソッド呼び出しなどを実現しているため、遠隔メソッド呼び出しの際はソケットを接続しなくてはならない。また、メソッドを呼ばれる側の RMIServer はサーバソケットを生成し、接続されたかどうか監視していなければならない。分散時のコールバックが発生した時の処理の流れを例を用いて表す。あるホスト A 上のスレッド 1 で synchronized 処理を行なっているクライアントから、遠隔ホスト B 上のマスタオブジェクトへの遠隔メソッド呼び出しが発生しているとする。このとき、コールバックが発生すると、ホスト A 上のスレッド 2 でソケットの監視をしているサーバソケットへコールバックの情報が渡される。サーバソケットはクライアントへ処理を渡そうとするが、synchronized 処理中であるためデッドロックが発生してしまう。

4.3.3　java.lang.ThreadLocal クラスを用いた解決

Addistant 2 では、コールバックが同じスレッド上へ返ってくるようにしている。スレッドごとにソケットを生成し、各ホスト上にある RMIServer 間はスレッド単位にコネクションを確立している。そのため、新たにスレッドを生成してサーバソケットによるソケットの監視を行なう必要がなくなる。コールバックに対しては、RMIServer 間でメソッドの返り値とコールバック識別することで、同じスレッド上でのコールバックを可能にしている。

スレッドごとのコネクションを実現するために、標準 Java API に含まれている java.lang.ThreadLocal クラスを用いている。このクラスはスレッドごとに固有な識別子を割り当てることが可能である。その識別子を利用してスレッドごとにソケットを割り当てている。新しいスレッドで、初めて遠隔メソッド呼び出しを実行するときにはソケットが存在しないため、ソケットを生成する必要がある。スレッドごとのコネクションを実現するため、リモートホスト上でスレッドを生成してからソケットを接続している。ローカルホスト上のスレッドと遠隔ホスト上のスレッドが一対一対応となっているため、非分散時と同じスレッド環境でアプリケーションを実行することができる。

4.4　RMIServer のマルチスレッド化

先述したように、RMIServer は、遠隔オブジェクトの生成や参照の保持、遠隔メソッドの呼び出しを実行するためのプログラムであり、パフォーマンスを向上させるために、マルチスレッド化するのが適切である。Java 言語は言語レベルでマルチスレッドを支援しているため、Java プログラムでは非常に簡単にマルチスレッドを使用することができる。

この節では、マルチスレッドプログラミングについて説明し、その記述の方法を解説する。

4.4.1　マルチスレッドプログラミング

マルチスレッドプログラミングは、一つのプログラムを論理的には独立に動くいくつかの部分に分けて、全体として調和して動くように組み上げるプログラミングのことである。

マルチスレッドを実現するためには、2 つの方法が存在する。

- マルチプロセス

  ほとんどのオペレーティング・システムで、マルチプロセスを生成することが可能である。 プログラムが起動すると、 それぞれのタスクのためのプロセスが生成され、それらが並行して実行される。ネットワーク・アクセスまたはユーザの入力を待つために、プログラムが停止するときにも別のプログラムが実行されて、リソースを効率的に使用する。しかし、このように個々のプロセスを生成する方法には不利な面が多い。プロセスを生成するたびにプロセッサは多くの処理時間を要し、 多くのメモリ・リソースも使用される。さらに、ほとんどのオペレーティング・システムでは、プロセスが他のプロセスのメモリ・スペースへアクセスすることを許可しない。そのため、プロセス間のやり取りは非常に厄介となり、プログラミングは一層複雑になります。

- スレッド

  スレッドはライト・ウェイト・プロセス (LWP) としても知られている。スレッドは 1 つのプロセス内でのみ機能するため、スレッドの作成は、プロセスを作成するよりもリソースを消費しない。さらに、マルチスレッドは、コンピュータ・リソースの消費を抑えるだけでなく、調整やデータ交換を可能にするため、マルチプロセスよりも有利と言える。

ここで、スレッドとプロセスは似た概念である。この 2 つは定義による区別は存在しないが、一般的にプロセスはスレッドよりもお互いに結び付きの少ない制御のことである。

4.4.2　Thread クラスと Runable インターフェイス

Java 言語では、マルチスレッドを記述するために 2 つの方法が存在する。

- java.lang.Thread クラスのサブクラスを生成する

  標準 Java API の java.lang.Thread クラスは、 スレッドの振る舞いが収められている具象クラス（つまり、抽象クラスではない具体的

なもの）である。ユーザは、スレッドを作成する場合に、Thread クラスから新しいクラスを作成する必要があり、目的の作業を行うように Thread クラスの メソッド run() をオーバーライドする。ユーザが直接このメソッドを呼び出すのではなく、ユーザはスレッドのメソッド start() を呼び出して、次に メソッド run() を呼び出します。以下の例は、その使用方法を示す。

```
class MyThread extends Thread {
    MyThread () {
        ... ...
    }
    public void run() {
        for ( ; ; ) {
            ... ...
            Thread.sleep(...);
            ... ...
        }
    }

    public static void main(String[] args) {
        MyThread mt1 = new MyTread();
        mt1.start();
        MyThread mt2 = new MyTread();
        mt2.start();
    }
}
```

しかし、この構造でスレッドを作成する方法は、スレッドとして実行させたいクラスがすでにクラス階層の一部となっている場合があるために、実現できない場合がある。Java 言語は、同じクラスに複数のインターフェイスが実装されていても、クラスが同じ 1 つの親を持つようにする。それにより、クラス階層を反映しないように、java.lang.Thread クラスからスレッドを作成しないこともある。

- java.lang.Runable インターフェイスを実装する

この Runable インターフェイスは 1 つのメソッド run() を用意しており、このインターフェイスを実装するクラスによって実装される必要がある。この Runable インターフェイスの使用方法を記述すると以下のようになる。

```
class MyThread2 implements Runable {
    MyThread2 () {
        ... ...
    }
    public void run() {
        for ( ; ; ) {
            ... ...
            Thread.sleep(...);
            ... ...
        }
    }

    public static void main(String[] args) {
        Thread t1 = new Thread(new MyThread2());
        t1.start();
        Thread t2 = new Thread(new MyThread2());
        t2.start();
    }
}
```

Runable インターフェイスを使用する場合、自分の望むクラスのオブジェクトを直接作成して実行することはできない。java.lang.Thread クラスのインスタンスの中から実行する必要がある。

4.4.3　内部クラス（inner class）

JDK 1.1 以降ネストしたクラスが宣言できるようになった。ここでは、クラスの宣言する場所の違いによる様々なクラスを説明する。

- トップクラス（top class）

  トップクラスとは、ファイル名と同じ名前をもつクラスであり、基本的なクラスである。

```
/** TopClass.java **/
public class TopClass {
    ... ...
}
```

- ネストトップクラス（nest top class）

  ネストトップクラスとは、ネストされてはいるが機能的にトップクラスと同じクラスのことである。ネストトップクラスの作成には、必ず static 修飾子を指定しなくてはならない。つまりインスタンスを生成する必要がなく、this をもたないクラスである。ネストトップクラスに他のクラスからアクセスすることも可能である。

```
/** TopClass2.java **/
public class TopClass2 {
    public static void main(String[] args) {
        TopClass2.NestTop nest = new TopClass2.NestTop();
        nest.write();
    }
    static class NestTop {
        void write() {
            ... ...
        }
    }
}
```

- メンバクラス（member class）

  ネストトップクラスは、スコープ内で宣言されるクラスであったが、トップクラスと同様な機能をもっていた。メンバクラスは、修飾子

として static を宣言することができないクラスであり、インスタンスを生成しなくてはならない。

```
/** TopClass3.java **/
public class TopClass3 {
    public static void main(String[] args) {
        TopClass3 tc = new TopClass3();
        tc.Member member = tc.getMember();
        member.write();
    }
    Member getMember() {
        Member obj = new Member();
        return obj;
    }
    class Member {
        void write() {
            ... ...
        }
    }
}
```

ネストトップレベルクラスでは、内部クラスから外側のメンバを直接参照することは不可能だった。しかし、メンバクラスの場合はトップクラスの一部であると考えられるため、内部クラスからトップクラスのメンバにアクセスすることが可能となる。

- ローカルクラス（local class）

  ローカルクラスとは、スコープ内部で宣言できるクラスである。つまり、クラスのメンバとしてではなく、クラスのメンバ内のクラスであるといえる。通常、ローカルクラスはメソッド内のクラスのことを意味する。

```
/** TopClass4.java **/
public class TopClass4 {
    public static void main(String[] args) {
        TopClass4 tc = new TopClass4();
        Foo foo = tc.getFoo();
        foo.write();
    }
    Foo getFoo()
        class Local extends Foo {
            void write() {
                ... ...
            }
        }
        return new Local();
    }
}
abstract class Foo {
    abstract void write();
}
```

- 匿名クラス（anonymous class）

  ローカルクラスでも、メンバの定義と生成を 1 つのメソッド内で行なうことができたが、その作業は別々で行なわれていた。匿名クラスは、定義とインスタンス生成を同時に行なうことのできるクラスである。

```
/** TopClass5.java **/
public class TopClass5 {
    public static void main(String[] args) {
        TopClass5 tc = new TopClass5();
        Foo foo = tc.getFoo();
        foo.write();
    }
```

```
        Foo getFoo() {
            return new Foo() {
                void write() {
                    ... ...
                }
            };
        }
    }
    abstract class Foo {
        abstract void write();
    }
```

4.4.4　RMIServer のマルチスレッド化

上記の java.lang.Thread クラスと匿名クラスを利用して RMIServer をマルチスレッド化する。以下のコードは、マルチスレッド化した RMIServer クラスの概要であるである。

```
public class RMIServer extends Thread {
    public RMIServer (int port, ...) {
        ServerSocket ssocket = new ServerSocket(port);
    }
    public void run() {
        for( ; ; ) {
            ... ...

                final Socket socket = ssocket.accept();
                new Thread() {
                    public void run() {
                        while (true) {
                            processRequest(socket);
                        }
                    }
                }.start();
```

```
            ... ...
        }
    }
}
```

匿名クラスを利用して RMIServer クラス中の run() メソッド内部に Thread オブジェクトの定義および生成をすることができる。ただし、匿名クラス内部で利用する外部変数は、修飾子 final を宣言していなくてはならない。これにより RMIServer をマルチスレッド化することができる。

4.5　オブジェクト入出力クラスの拡張

4.5.1　オブジェクト直列化と目的

オブジェクト直列化 [11] とは、コア Java API の入出力クラスを、オブジェクトを支援するように拡張したものである。オブジェクト直列化は、オブジェクトとそこから参照されているオブジェクトを、バイトストリームにコード化する。そして、そのストリームからオブジェクト構造の完全な復元を行われる。直列化は、簡易な持続性の実現や、ソケットや 遠隔メソッド呼び出しによる通信のために使用される。

バイトストリームへの Java オブジェクトの格納（writeObject）と取り出し（readObject）の能力は、アプリケーションを作成するために必要である。オブジェクトの直列化された形式の格納と取り出しで重要なことは、バイトストリームにコード化されたオブジェクトが、もとのオブジェクトの状態を再構築するのに十分な表現がされているかという事である。ストリームに保管されるオブジェクトでは、java.io.Serializable もしくは java.io.Externalizable インターフェイスのどちらかを実装していなくてはならない。オブジェクトでは、直列化された形式によって、オブジェクトの内容が保管されていたクラスを識別および検証し、その内容を新しいインスタンスに復元できなければならない。Serializable オブジェクトの場合、ストリームには、そのフィールドを互換性のあるバージョンのクラスに復元できるだけの十分なデータが含まれている。

もとオブジェクト内で他のオブジェクトを参照している場合、他のオブジェクトともとのオブジェクトの関係を維持するために、もとのオブ

ジェクトが格納と取り出しが行なわれたならば、同時に他のオブジェクトも格納と取り出しが行なわれなければならない。オブジェクトが格納されると、そのオブジェクトから参照関係にあるすべてのオブジェクトも格納される。

オブジェクトの直列化の目的として以下があげられる。

- シンプルで拡張性のある機構にする
- オブジェクトの型とその属性を、直列化された形式に維持する
- リモートオブジェクトに対し、必要に応じて整列化と非整列化を支援するだけの拡張性を持つ
- オブジェクトのシンプルな持続性を支援するだけの拡張性を持つ
- カスタマイズの場合だけ、クラスごとの実装を必要とさせる
- オブジェクトによってその外部形式を定義できる

4.5.2　直列化されるオブジェクトを置換する

Addistant 2 では、バイトストリームとしてコード化されるオブジェクトを置換したいときがある。例えば、手元のホスト上で生成したユーザ定義のクラスのオブジェクトを、遠隔メソッドの引数にしてはいけない。Addistant 2 では、手元のオブジェクトを生成した際に遠隔ホスト上にもそれと対をなすオブジェクトを生成[2]し、それぞれのホスト上の RMIServer のオブジェクト参照テーブルに登録しておく。この 2 つのオブジェクトは、登録した時の objectID（key）が同値であるため、遠隔メソッドの引数として手元で生成したオブジェクトをバイトストリームとして送信する必要はなく、その objectID（key）のみを送信する。そしてその objectID（key）が参照するオブジェクトを遠隔ホストのオブジェクト参照テーブルから取り出し、メソッドの引数にすればよい。一方、標準 Java API に組み込まれているクラス（String 型や java.util.Vector 型など）のオブジェクトは、置換することなくそのコピーをストリームにそのまま流したい。

[2] この場合、手元でプロキシオブジェクトが生成されれば、遠隔ホスト上にマスタオブジェクトが生成される。また、遠隔ホスト上にプロキシオブジェクトが生成されれば、マスタオブジェクトが生成される。

このように、マスタオブジェクトやプロキシオブジェクトならば、バイトストリームとして遠隔ホストに出力される前に置換し、またその他のオブジェクトならばそのままストリームに出力したい場合、java.io.ObjectOutputStream クラスの replaceObject() メソッドを利用して直列化する直前に条件により出力するオブジェクトを置き換えることが可能である。

4.5.3　replaceObject メソッド

ObjectOutputStream をスーパークラスにもつユーザ定義のクラスを作成し、内部で replaceObject() メソッドを記述することにより、直列化中のオブジェクトの代わりに代替オブジェクトを直列化させることができる。ただし、直列化の際のオブジェクトを置換する replaceObject() メソッドを呼ぶためには、enableRaplaceObject() メソッドを事前に呼ばなくてはならない。

この enableReplaceObject() メソッドは、信頼できる ObjectOutputStream のサブクラスが、直列化の際に、あるオブジェクトで別のオブジェクトを代用することを可能にするために呼び出される。オブジェクトの置換は、enableReplaceObject() メソッドが true 値で呼び出されるまでは、使用不可になっている。また、使用可能にしたあとで、false に設定して、使用不可にされる場合がある。enableReplaceObject() メソッド は、置換を要求するストリームを信頼できるかどうかを調べる。オブジェクトの private 状態が意図せずに公開されないことを保証するために、信頼できるストリームサブクラスだけが replaceObject() メソッドを使用することを許可する。

以下では、Addistant 2 で実装した ObjectOutputStream のサブクラス RMIObjectOutputStream クラスの概要を説明する。

```
public class RMIObjectOutputStream
    extends ObjectOutputStream {

    public RMIObjectOutputStream
          (OutputStream out, ...) {
        super(out);
        enableReplaceObject(true);
    }
```

```
    public final Object replaceObject(Object obj) {
        Object returnObj = obj;
        if (isProxyObject(obj)) {
              ... ...
            returnObj = new ProxyTag(objectID);
            return returnObj;
        }
        if (isMasterObject(obj)) {
                ... ...
            returnObj = new MasterTag(objectID);
            }
        }
        return returnObj;
    }
}
```

先述のようにオブジェクトの直列化の際に、そのオブジェクトがマスタオブジェクトやプロキシオブジェクトならば、オブジェクトの objectID（key）を取り出しその objectID を遠隔ホストに送信する。ところが、objectID をそのまま送信することは、それがマスタの objectID かプロキシの objectID か区別がつかないため、マスタの objectID であれば、それを MasterTag クラスに格納する。この MasterTag クラスとは、Addistant 2 特有のクラスであり、objectID を格納するためだけに存在するクラスである。当然、MasterTag クラスは Serializable インターフェイスを implements していなくてはならない。一方、プロキシの objectID ならば、Addistant 2 はそれを格納するために、ProxyTag クラスを用意している。また、オブジェクトがマスタオブジェクトやプロキシオブジェクトではないならば、そのまま直列化すればよい。

また、既存の ObjectOutputStream クラスの replaceObject() メソッドは以下のようになっている。

```
protected Object replaceObject(Object obj)
    throws IOException
{
    return obj;
}
```

これは引数に取ったオブジェクト obj をそのまま無変換で返すメソッドであるが直列化の際のオブジェクトを置換したいときは、この replaceObject() メソッドをオーバーライドする。さらに、オーバーライドしたユーザ定義の replaceObject() メソッドを有効にするために、enableReplaceObject() の引数を true として置かなくてはならない。なぜならば、ObjectOutputStream クラスの writeObject() メソッド内で引数が true でなければ、replaceObject() メソッドが実行されないのである。

4.5.4　バイトストリームから復元されたオブジェクトを置換する

Addistant 2 の仕様で、遠隔ホスト上から送信されたオブジェクトをバイトストリームからの復元後、他のオブジェクトに置換する場合がある。それは、4.5.2 節や 4.5.3 節で説明した MasterTag オブジェクトや ProxyTag オブジェクトが送信されてきた場合である。MasterTag オブジェクトや ProxyTag オブジェクトは、objectID を格納するための入れ物であり、その objectID がプロキシのものかマスタのものかを分別するために必要である。ところが、このオブジェクトをそのままメソッド引数の値に代入することは不可能である。そのため、MasterTag オブジェクトを受信したら、内蔵される objectID を取り出し、プロキシ専用のオブジェクト参照テーブルからオブジェクトを参照し、MasterTag オブジェクトと置き換えなくてはならない。ProxyTag オブジェクトの場合も同様である。

このように、オブジェクトをバイトストリームとして受信した側でも、復元後に他のオブジェクトと置換する必要にせまられる場合、java.io.ObjectInputStream クラスの resolveObject() メソッドを利用してバイトストリームから復元したオブジェクトを置き換えることが可能である。

4.5.5　resolveObject メソッド

ObjectInputStream クラスの resolveObject() メソッドは、4.5.2 節の replaceObject() メソッドと対になるクラスであり、バイトストリームから復元した後のオブジェクトを置き換える役割をもつ。標準の Java API で、resolveObject() は以下のコードである。

```
protected Object resolveObject(Object obj)
    throws IOException
{
    return obj;
}
```

この resolveObject() メソッドは Object 型の引数 obj を受け取って Object 型の obj を返すメソッドである。この resolveObject() メソッドをオーバーライドするためには、ObjectInputStream クラスのサブクラスをユーザ定義のクラスとして生成しなくてはならない。ただし、オーバーライドした resolveObject() メソッドを readObject() メソッドから呼ぶためには、enableResolbeObject() メソッドを事前に呼ばなくてはならない。この enableResolbeObject() メソッドは、ObjectOutputStream クラス内の enableReplaceObject() メソッドと同様、ObjectInputStream クラスのサブクラスが信頼できるクラスであるかをチェックするためのメソッドである。以下では、Addistant 2 が実装した ObjectInputStream クラスのサブクラスである RMIObjectInputStream クラスの概要を説明する。

```
public class RMIObjectInputStream
    extends objectInputStream {

    public RMIObjectInputStream(InputStream in, ...) {
        super(in);
        enableResolveObject(true);
    }

    public final Object resolveObject(Object obj) {
```

```
        Object returnObj = obj;
        if (isProxyTag(obj)) {
            int objectID = ((ProxyTag) obj).objctID;
            returnObj = masterTable.referObject(objectID);
        } else if (isMasterTag(obj)) {
            int objectID = ((MasterTag) obj).objctID;
            returnObj = proxyTable.referObject(objectID);
        }
        return returnObj;
    }
}
```

RMIObjectOutputStream クラスの解説と同様に、バイトストリームから復元されたオブジェクトが ProxyTag ならば、内蔵されている objectID を取り出し、マスタオブジェクトが登録されているオブジェクト参照テーブルからマスタオブジェクトを取り出し、復元されたオブジェクトが MasterTag ならば、内蔵されている objectID を取り出し、マスタオブジェクトが登録されているオブジェクト参照テーブルからその objectID のオブジェクトを取り出し置換する。

4.6　分散アスペクトの記述をシステムに埋め込む

Addistant 2 では、XML を用いて分散アスペクトを一つのファイルに記述している。この分散アスペクトファイルは Addistant 2 起動時にシステムに読み込まれるようになっている。この節では XML の必要性と XML で記述されたデータ構造を Addistant 2 のプログラム内に埋め込む方法を説明する。

4.6.1　XML の必要性

the eXtensible Markup Language（XML）[10] はメタレベルの言語を簡単に設計することができる。XML は単なるマークアップ言語にとどまらない。HTML と酷似しているが、HTML と XML は本質的に異なるものである。HTML の各タグにはそれぞれプレゼンテーション用の意味が

含まれており、その変更やタグの追加による拡張は不可能である。HTML は Web ブラウザ上に情報をグラフィカルに表示するために作成された、マークアップ言語 SGML の応用の一例である。一方、XML はマークアップ言語作成用のメタ言語である。XML を基にして、様々な応用分野におけるマークアップ言語が作成可能である。

XML はプラットフォームに独立しており、アプリケーションの記述に依存しないデータ構造を提供するのに適した技術である。しかし、XML で構築したデータ構造を理解し、それを使ってアプリケーションの記述を実行する手段が必要となる。その手段として、プラットフォームに依存しない実行環境である Java は最適である。

4.6.2　XML データを取り出す

XML で記述されたデータをアプリケーションに埋め込むためには、XML パーザを利用して XML データを一つ一つ解析しなくてはならない。Addistant 2 では XML パーザとして javax.xml.parsers パッケージのクラスを使用している。以下では、XML で記述されたデータを XML パーザで取り出すための実装を説明する。

XML で記述されたデータを下に示す。

```
/** policy.xml **/
<policy>
    <start name="app">
        <host name="picard" rmiPort="14000, 14001"
                            bytecodePort="14002" dir="apprun"/>
    </start>
    <remote name="rmt">
        <host name="taro" rmiPort="14003"
                          bytecodePort="14004" dir="rmtrun"/>
        <host name="yulian" rmiPort="14005"
                            bytecodePort="14006" dir="rmtrun2"/>
    </remote>
</policy>
```

XML は、HTML の記述方法と同様にデータをタグに格納している。XML

パーザはこの XML データを、タグの入れ子構造に従い、木のデータ構造を構築する。例えば上記の XML ファイル（policy.xml）を XML パーザが構築した木構造は以下のようになる。

```
root
 |
 |___ node policy
    |
    |__text \n
    |
    |___node start
    |  | \___attribute name = app
    |  |__text \n
    |  |
    |  |___node host
    |  |  | \___attribute name = picard
    |  |  | \___attribute rmiPort = 14000, 14001
    |  |  | \___attribute bytecodePort = 14002
    |  |  | \___attribute dir = apprun
    |  |  |
    |  |  |__text \n
    |  |
    |  |__text \n
    |
    |___node remote
    |  | \___attribute name = rmt
    |  |
    |  |__text \n
    |  |
    |  |___node host
    |  |  | \___attribute name = taro
    |  |  | \___attribute rmiPort = 14003
    |  |  | \___attribute bytecodePort = 14004
    |  |  | \___attribute dir = rmtrun
    |  |  |
```

```
    |  |  |__text \n
    |  |
    |  |__text \n
    |  |
    |  |___node host
    |  |  | \___attribute name = yulian
    |  |  | \___attribute rmiPort = 14005
    |  |  | \___attribute bytecodePort = 14006
    |  |  | \___attribute dir = rmtrun2
    |  |  |
    |  |  |__text \n
    |  |
    |  |__text \n
    |
    |__text \n
```

上記のように、XML パーザにより XML データ構造を木構造にすることができた。もし、remote タグ内の一番目の host タグ内の name 属性の値（taro）を取り出すのならば、以下のように実装する。

```
import java.io.File;
import java.io.IOException;
import javax.xml.parsers.*;
import org.xml.sax.SAXException;
import org.w3c.dom.*;

DocumentBuilderFactory factory
    = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse(new File("policy.xml"));

NodeList nodelist0 = document.getElementsByTagName("remote");
Node node0 = nodelist0.item(0);
NodeList nodelist = node0.getChildNodes();
Node node = nodelist.item(1);
```

```
NamedNodeMap attrs = node.getAttributes();
Node attr = attrs0.item(0);
String attr_value = attr.getNodeValue();
```

DocumentBuilder クラスの parse() メソッドは、引数にとった XML 文書を、木のデータ構造に変換し、Document オブジェクトとして生成する。メソッドの引数は、File, InputStream, InputSource 型をとることができ、手元の XML 文書だけでなく、遠隔のホスト上の文書も引数にとることが可能である。Document インターフェイスの getElementsByTagName() メソッドは、引数にとったタグのノード（Node オブジェクト）をリスト（NodeList オブジェクト）として格納する。例では、”remote” タグが引数であるので、”remote” タグのノードをリスト化する。NodeList インターフェイスのメソッド item() は、引数番目のノードを表す。例のコードでは、XML ファイルに ”remote” タグが一カ所しか存在していないため、item(0) と引数が 0 になっている。Node インターフェイスのメソッド getChildNodes() は、そのノードの子供のノードをリスト化する。この例で、リスト nodelist は {text, node(host), text, node(host), text} となる。ここで text とは、改行、空白を表す Text オブジェクトのことである。そのため、nodelist.item(1) としているのはリストの第一ノードが Text オブジェクトだからである。Node インターフェイスのメソッド getAttributes() は、そのタグの内部の属性をリスト化している。そのリストの第一属性が name であり、その値を取り出すために Node インターフェイスのメソッド getNodeValue() が存在する。

これにより、remote タグ内の一番目の host タグ内の name 属性の値（taro）を取り出すことができる。ただし、Addistant 2 で扱う XML 文書のデータの読み出しには、この実装は適切ではない。なぜならば、XML 文書から取り出したいデータは、remote タグだけではない。start タグや orb タグなど、Join point の分だけタグが存在しているため、十分抽象化して、実装すべきである。

4.6.3　分散アスペクトの記述をシステムに埋め込む

Addistant 2 では、分散に関する記述である分散アスペクトファイルの内容を ServerManager クラスに格納する。この ServerManager クラスとは、分散アスペクト内の記述を XML パーザで解析した後、分散アスペク

トファイルのデータをシステムに埋め込むためのクラスである。Addistant 2 は、この ServerManager クラスから、ホスト数、ポート番号やホスト名を参照する。

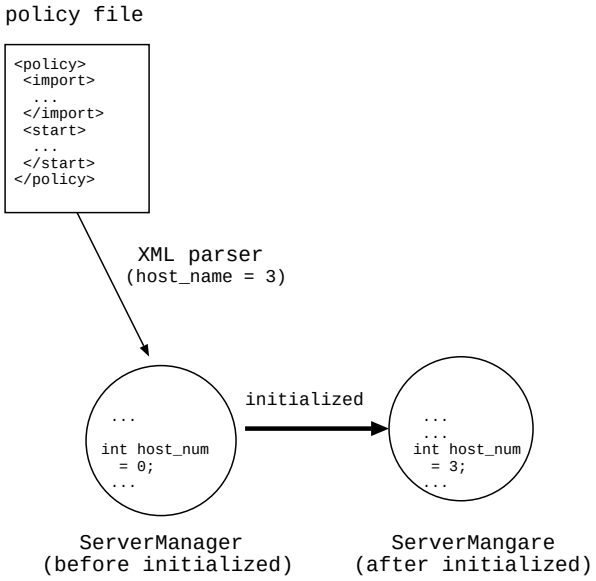


図 4.8: Analizes policy file by XML parser, then initializes the variables in ServerManager.

例えば、ServerManager クラスの記述として以下のものがある。

```
public class ServerManager {
    public static int HOST_NUMBER;
    public static void setHOST_NUMBER(int num) {
        HOST_NUMBER = num;
    }
    public static int getHOST_NUMBER() {
        return HOST_NUMBER;
    }
}
```

この変数 HOST_NUMBER は、Addistant 2 を実行するホスト数を表している。Addistant 2 を実行するホスト数は、個々のユーザが扱う環境によ

り異なるため、Join point としておくのが適切である。Addistant 2 では、ServerManager クラスの変数 HOST_NUMBER の初期化に DefaultConstructionSetter クラスを定義している。以下は、その DefaultConstructionSetter クラスの一部である。

```
public class DefaultConstructionSetter {
    XMLParser parser = new XMLParser("policy.xml");
    ServerManager.setHOST_NUMBER(parser.getHostNumber());
}
```

XMLParser クラスのメソッド getHostNumber() を呼ぶことで、分散アスペクトファイル（policy.xml）から、Addistant 2 を実行するホスト数を解析し、ServerManager クラスの変数 HOST_NUMBER にメソッド setHOST_NUMBER() を利用して代入する。これにより、Addistant 2 のシステムクラスは、ホスト数を ServerManager オブジェクトの getHOST_NUMBER() メソッドを通じて参照することが可能となる。DefaultConstructionSetter クラスは、ホスト数以外の分散アスペクトのデータも、同様な方法で ServerManager クラス内に代入している。

第5章　応用例

ここでは、Addistant 1 では分散化ができなかった 2.3 章の例を取り上げ、Addistant 2 を用いて分散化する例を示す。また、多重化 ORB を使う例を示す。

5.1　オブジェクト配置の指定

2.3 節の例は、JFrame クラスのオブジェクト viewerFrame と editorFrame のうち、veiwerFrame は手元のディスプレイに、editorFrame は遠隔のディスプレイに表示させるという機能分散を指定するものであった。

Addistant 2 では、オブジェクト配置の指定を、オブジェクトの生成文脈によって変えることができる。つまり、2 つのオブジェクト viewerFrame と editorFrame が、同一クラスで生成されていても、演算子 new が現れるメソッド createViewerFrame() と createEditorFrame() が異なれば、オブジェクトの配置を区別して指定することができる。

以下に、2.3 節の例のプログラムを分散化するための分散アスペクトを示す。

```
<import proxy="subclass" in="FrameClient.createViewerFrame()"
        from="rmt">
    <subclass name="javax.swing.JFrame"/>
</import>
<import proxy="subclass" in="FrameClient.createEditorFrame()"
        from="app">
    <subclass name="javax.swing.JFrame"/>
</import>
<start name="app">
    <host name="picard" rmiPort="14003" bytecodePort="14004"
          searchDir="apprun"/>
```

```
</start>
<remote name="rmt">
    <host name="taro" rmiPort="14005" bytecodePort="14006"
          searchDir="rmtrun"/>
</remote>
```

この記述を Addsitant 2 に与えると、2.3 節の例のプログラムは 3.1 節で述べたように、viewerFrame オブジェクトは変数 rmt で指定されるホストに、editorFrame オブジェクトは変数 app で指定されるホストに配置されるようになる。

5.2　ORB クラスの拡張

次に、Tic Tac Toe（マルバツゲーム）プログラムを分散化する例を示す。この例では、ユーザが定義した ORB クラスを用いる。

例えば、異なる 2 つのホストを利用して、2 人がマルバツゲームをしたいとする。このアプリケーションの実装には、それぞれのホスト上に同じ画面を表示する JFrame オブジェクトを生成する必要がある。このためには、片方の JFrame オブジェクトのメソッドが呼ばれたときには、他方のメソッドも同じ引数で呼び出してやり、2 つのオブジェクトの内部状態が常に同じになるようにすればよい。

5.2.1　RequestMultiplyingBroker クラスの実装

このような遠隔メソッド呼び出しを多重化する ORB を我々はユーザ定義の ORB として実装した。

RequestBrokable インターフェイス

第 3 章で記述したように、ORB を拡張するためには RequestBrokable インターフェイスを実装しなくてはならない。以下のコードは Request-Brokable インターフェイスである。

```
public interface RequestBrokable
{
    public Object invokeRemoteMethod(int id, String cvalue, Object[] args)
        throws InvocationTargetException;
    public int createRemoteObject(String cvalue, Object[] args)
        throws InvocationTargetException;
    public void registerNewProxyObject(int proxyID, RemoteProxy proxy);
}
```

コード内の 3 つのメソッドは、いずれも RMIServer とアクセスするためのメソッドである。簡単にそれぞれを解説する。

- invokeRemoteMethod() メソッド
  objectID（id）をもつ遠隔オブジェクトに対して、メソッドを呼び出す。String 値 cvalue にメソッド名とメソッド引数の型が格納されている。また、Object[] 値 args は、呼び出す遠隔メソッドの引数の値である。

- createRemoteObject() メソッド
  遠隔ホスト上にオブジェクトを生成し、RMIServer に登録された key（objectID）を返すメソッドである。String 値 cvalue には、クラス名とコンストラクタの引数の型が格納され、Object[] 値 args は、コンストラクタ引数の値である。

- registerNewProxyObject() メソッド
  新しく生成したプロキシオブジェクトを RMIServer に登録するためのメソッドである。RemoteProxy 値 の proxy は今回登録するプロキシオブジェクトであり、int 値の proxyID はそのプロキシオブジェクトの objectID である。

ORB をどのように拡張しようとしても、この 3 つのメソッドがなくては Addistant 2 は機能しないため、インターフェイス内に記述されているのである。

RequestMultiplyingBroker クラス

この例では、複数の遠隔ホストにオブジェクトのコピーを保持させる多重化 ORB を構築したい。つまり、変数 app で指定されたホスト上の ORB が特定の遠隔ホストの RMIServer とだけアクセスするのではなく、複数の遠隔ホストとアクセスするように ORB を拡張すればよい。

以下のコードは実装した ORB、RequestMultiplyingBroker クラスのコンストラクタと invokeRemoteMethod() メソッドである。

```
public class RequestMultiplyingBroker implements RequestBrokable
{
    RequestBrokable[] brokers;
    public RequestMultiplyingBroker(RequestBrokable[] brokers) {
        this.brokers = new RequestBrokable[brokers.length];
        for (int i = 0; i < brokers.length; ++i) {
            this.brokers[i] = brokers[i];
        }
    }

    public Object invokeRemoteMethod(int id, String cvalue, Object[] args)
        throws InvocationTargetException
    {
        Object result = null;
        for (int i = 0; i < brokers.length; ++i) {
            result = brokers[i].invokeRemoteMethod(id, cvalue, args);
        }
        return result;
    }
        ... ...
}
```

コンストラクタでは、複数の遠隔ホスト上の RMIServer とアクセスするための ORB（RequestBrokable 型）を初期化している。そして、invokeRemoteMethod() メソッド内部では複数の ORB に同じ命令を送り、それぞれの遠隔ホスト上の RMIServer を通じてメソッド呼び出しをしてい

る。他の 2 つのメソッドも同様に実装することができる。

5.2.2　ORB クラスの拡張を適用するための分散アスペクトの記述

この 多重化 ORB を使って、Tic Tac Toe を分散化するには、次のような分散アスペクトを記述すればよい。

```
<import proxy="subclass" in="TTTStarter" from="app">
    <subclass name="javax.swing.JFrame"/>
</import>
<import proxy="subclass" in="EventHandler" from="rmt">
    <subclass name="javax.swing.JFrame"/>
</import>
<orb classname="RequestMultiplyingBroker"/>
<start name="app">
    <host name="picard" rmiPort="14003" bytecodePort="14004"
          searchDir="apprun"/>
</start>
<remote name="rmt">
    <host name="taro" rmiPort="14005" bytecodePort="14006"
          searchDir="rmtrun"/>
    <host name="yulian" rmiPort="14007" bytecodePort="14008"
          searchDir="rmtrun2"/>
</remote>
```

この記述を Addistant 2 に与えると、3.2 節で述べたように ORB としてユーザ定義のクラス RequestMultiplyingBroker が用いられる。さらに、この場合「遠隔リーダーホスト」は、taro と呼ばれるホストとなる。

第6章　関連研究

これまでの分散処理プログラムの開発ツールは、プログラムを書く際に、初めから一定のスタイルに沿って書かなければならないものがほとんどであった。例えば、Java 標準 API に組み込まれている JavaRMI [7][8] では、遠隔オブジェクトは全てインタフェース型を通じて参照されなければならないという制約がある。この制約に沿ってプログラムを書かないと、JavaRMI が提供するスタブ生成コンパイラと ORB ライブラリを利用することができない。したがって、Addistant と異なり、初め分散を意識しないで書かれたプログラムを、後から分散対応にするような目的には JavaRMI は適さない。

D 言語 [9] もまた同様の問題がある。D 言語はアスペクト指向の分散プログラミングを支援する処理系であり、分散に関するアスペクトとして, 並列に動作するスレッド間の協調動作を扱う coordination アスペクトと、遠隔手続き呼び出しを扱う Interface Definition Language（IDL）を提供する。しかし、分散化の指示を、プログラム内に直接、修飾子の形で埋め込むため、分散を意識しないで書かれたプログラムを、後から分散対応にする場合には、元のソースファイルをあちこち修正しなければならない。とくに、Addistant と異なり、元のソースファイルが手に入らなければ分散対応にできないという問題がある。

AspectJ [2] は、Java 言語用の汎用的 AOP 言語である。AspectJ では、元のソースファイルからは独立したアスペクト記述を書くだけで、オブジェクトの挙動を変えるコードを join point に自動的に挿入することが可能である。これにより、ロギングやトランザクション管理を、元のソースファイルを修正せずに、アスペクト記述を書くだけで実装することができる。AspectJ と Addistant との違いは、前者が汎用的 AOP 言語であるのに対し、後者は分散処理に特化した AOP 言語であることである。AspectJ が提供する join point は、言語の基本機能に関係するものだけなので、AspectJ だけを使って既存プログラムを分散対応に拡張しようとすると、それは非常に困難になるだろう。

第7章　まとめ

本稿では、複数の JVM を利用して、機能分散を行うソフトウェアの開発を支援する分散用アスペクト指向言語、Addistant 2 を提案した。Addistant 2 の利用者は、分散に関する記述を、分散とは無関係な Java プログラムから、分離してまとめることができる。従来、分散環境を変更する際にはプログラム全体にわたって修正をおこなわなければならなかった。一方、Addistant 2 を使えば、我々が分散アスペクトと呼ぶ、独立したファイルの記述の修正のみで分散環境を変更できるようになった。Addistant 1 は分散アスペクトの記述力が十分ではなかったが、Addistant 2 では、Join point の種類を増やすことにより、分散アスペクトの記述力を強化することができた。これにより、Addistant 2 の利用者は、オブジェクトの生成文脈に従って、オブジェクトがどこに配置され、それらのオブジェクトの遠隔参照がどのように実装されているかを指定するファイルを与えるだけでよい。その他にも、分散アスペクトに記述するだけで、ユーザ定義の ORB クラスやネットワーク・ストリームの実装クラスを拡張したり、アプリケーションの実行環境を変更したりすることができる。Addistant 2 の典型的な利用例として、Swing ライブラリを用いたプログラムについて、同一クラスから生成された GUI オブジェクトが複数のホスト上で動作する例を示した。

Addistant 2 は Addistant 1 のコードを再利用せずに現在開発中である。現在までのところ、Addistant 1 相当の部分がほぼ完成している。ただし、遠隔オブジェクト参照は『subclass』しか指定できない。また、Addistant 2 での拡張はユーザ定義の ORB を指定する機能が完成している。

参考文献

[1] Tatsubori M,Sasaki T,Chiba S and Itano K, A Bytecode Translator for Distributed Execution of "Legacy" Java Software, In *Proccedings of ECOOP 2001, LNCS 2072*, Springer, pp.313–336, 2001.

[2] Gregor K,John L,Anurag M,Chris M, Aspect-Oriented Programming, In *Procedings of ECOOP 1997, LNCS 1241*, June, 1997.

[3] Shigeru Chiba, Load-time Structural Reflection in Java, In *Proccedings of ECOOP 2000, LNCS 1850*, Springer Verlag, pp.313–336, 2000.

[4] Liang, S. and Bracha, G., Dynamic Class Loading in the Java Virtual Machine, *Proccedings of ACM Conf. on OOPSLA 1998*, ACM SIGPLAN Notices, Vol.33, No. 10, pp. 36-44, .

[5] Shigeru Chiba, Michiaki Tatsubori, *Structural Reflection by Java Bytecode Instrumentation* .

[6] Rohnert, H., *The Proxy Design Pattern Revisited*, Addison-Wesley, pp. 105-118 1995 .

[7] Jim Farley, JAVA Distributed Computing, pp.47-79,

[8] Sun Microsystems, I, The Java Remote Method Invocation Specification, http://java.sun.com/products/jdk/rmi 1997.

[9] Nataraj Nagaratnam, Arvind Srinivasan, and Doug Lea, Remote Objects in Java, In *IASTED '96, International Conference on Networks*, 1996.

[10] Sun Microsystems, Java Technology and XML, http://java.sun.com/xml/docs.html.

[11] Sun Microsystems, Object Serialization, http://java.sun.com/j2se/1.3//docs//guide/serialization/index.html.

[12] Sun Microsystems, I, Java Foundation Classes, http://java.sun.com/products/jfc/ 1992.