

Java による分散プログラミング入門 ~ RMI まで ~

オブジェクト指向言語とオブジェクト指向設計の基礎

まず、はじめに、オブジェクト指向プログラミングについて簡単に解説する。

1つのプログラミング言語を知っていることと、その言語を正しくつかって実際にプログラムを書けることはおなじではない。特に、オブジェクト指向言語の場合には正しくつかえば、非常に効果的な保守性に優れたプログラムになるが、間違っつつかった場合には非常に醜いプログラムになってしまう。C++ や java のような非常に多機能なプログラミング言語の場合はその差は大きいものになってしまう。オブジェクト指向の考えをつかったオブジェクト指向設計については、Scott Meyers の "Effective C++" (岩谷訳、ソフトバンク、ISBN4 - 89052 - 401 - 0) の第6章「継承とオブジェクト指向設計」が非常に参考になるので、機会があったらみてほしい。

オブジェクト指向言語 C++

C++ は、C をベースにオブジェクト指向言語であり、1980年代半ばに Bjarne Stroustrup によって設計された言語である。C をベースにしているため、C を知っている人にはとっつきやすいが(たとえば、C のプログラムならば少々の変更でコンパイルできる)逆に C をベースにしているためにわかりにくくなっているところがある。オブジェクトは class で宣言する。下の例では、社員のデータをオブジェクトとして、定義している。この定義には、メンバー関数 print が定義されており、employee e に対して、e.print() でメンバー関数を呼び出す。右の例では、manager というオブジェクトを定義している。オブジェクト型 employee を「継承」しており、employee のメンバーに加えて、管理する社員へのポインター group を持つオブジェクト型であることを意味する。ここで、manager は employee から、導出された(derived)という。逆に、employee は manager の基本クラス(base class)であるという。個々で現れている public の意味は、このキーワード以降のメンバーは他のオブジェクトからアクセスできることを意味しメンバーの「可視性」を制御する。

```
class employee {
    char* name;
    short age;
    employee *next;
public:
    void print();
    ....
}
```

```
class manager : employee
{
    employee *group;
public:
    employee *getGroup()
    ....
}
```

以下に特徴をあげる：

- オブジェクトを定義するために class を導入。データ型に対し、その操作を定義するメンバー関数を宣言できる。ちなみに C の構造体である struct は、全メンバーが公開(public)な class と同値。
- クラス定義において、継承(inheritance)関係を定義でき、メンバーの可視性を制御できる。2つ以上のベースクラスも持つことができる。(Multiple inheritance)
- クラス定義においては、クラスを生成する構築子(constructor)と消滅子(destructor)を宣言でき、クラスが生成・消滅するときに呼び出される。
- new / delete 演算子
- 仮想メンバー関数(virtual function)
- オブジェクトに対し、演算子をできる(operator overloading)
- 多義関数名、int foo(int x)と int foo(double)は違う関数となる。ただし、「暗黙の型変換」が行われるので注意。
- default の引数が見える。
- 引数の Reference 渡しが見える。
- Template 機能。Generic なプログラミングができる。

オブジェクト指向言語 Java

ネットワーク向けのプログラミング言語として注目されている Java であるが、オブジェクト指向言語として C++ と比較されることが多い。

- すべてのプログラムはクラス定義の集まりで定義される。C のように、関数だけ、データ定義だけというのはない。
- オブジェクト指向言語。メンバー関数、メンバーの可視化制御、継承ができる。
- Constructor はあるが、destructor はない。参照されなくなったオブジェクトは自動的にガベージコレクションされる。
- ポインタはない。すべてのオブジェクトは、C++ でいえばポインターで表現されている。メンバー関数はすべて virtual メンバー関数。
- ひとつのオブジェクトからしか、継承できない。
- interface 定義。(C++ の仮想クラス定義に相当する)
- オブジェクト型に演算子は定義できない。Operator overloading なし。
- Template 機能もなし。

C++ と比較して議論されることもある java であるが、むしろ、その発想としては smalltalk に近い。プログラムは通常クラスファイルという java バイトコードからなる中間形式にコンパイルされ、java virtual machine と呼ばれるバイトコードインタプリタで実行される。この実行形式がネットワーク上の言語としての java の柔軟性を与えているといえる。

オブジェクト指向設計 (オブジェクト指向プログラミングの原則)

オブジェクト指向言語でプログラミングするときには、どれをオブジェクトにして、どのようなメンバー、メンバー関数を作るかを考えなくてはならない。プログラムを見通しよく作るには、プログラムする対象を反映したオブジェクトを設計、定義する必要がある。オブジェクト指向プログラミングに限らず、以下を考えることは重要である。

- 保守性：後から、見たとき、あるいはデバック中にも容易に理解できるようなプログラムを作る。他の人が見たときにわかりやすいこと (可読性) も重要である。
- 拡張性：プログラムの機能を加えるときに、なるべくほかのコードを変更せずに機能を加えることができることが望ましい。
- 再利用性：ほかのプログラムに転用できるような部品として設計しておけば、プログラムの価値は高まる。
- 効率：そして、プログラムは速くなくてはならない。

オブジェクト指向プログラミングをするときにオブジェクト設計の原則についていくつかあげておく。

public な継承が "is a" 関係であることをしっかり理解する (項目 35)

クラス A から public な継承をするクラス B は、タイプ B のオブジェクトはすべて、タイプ A であることを意味している。たとえば、

```
class Person { ... };          class Student : public Person { ... };
void dance(Person & p);       void study(Student& a);
```

を考えてみる。Person p; Student s; に対して、dance(p) でも、dance(s) でも OK であるが、study(s) は OK であるが、study(p) は NG である。つまり、public の継承は「特殊化」という意味を持つ。言い換えれば、public に継承するということは、ベースクラスは派生するクラスよりも一般的な概念であるということである。ベースクラスに特殊な public なメンバーを定義することは間違いを引き起こす。このことは、Java の public の継承にもいえる。

クラス間の関係としては、「has a」関係と「implemented in terms of」関係がある。

インタフェースの使い方、インタフェースと継承の違い

仮想メンバー関数の意味について考えてみる。C++ では、インタフェースのみを定義するためには純粋仮想関数というものをを用いる。java では、多重継承をさせない代わりに、C++ の仮想メンバー関数に相当する interface は別の定義で行う。

```
class Shape {
public:
    virtual void draw() const = 0; /* 純粋仮想関数 */
    int objectID();
```

```

    ...
}
class Rectangle: public Shape { ... };
class Oval : public Shape { ... };

```

Shape を継承する Rectangle も Oval も、メンバー関数 draw を定義しなくてはならない。インタフェースの継承とは、それを継承するメンバー関数は同じインタフェースを持っていることを強制することを意味する。純粹仮想関数を宣言する目的は、派生するクラスにインタフェースだけを継承させることである。純粹仮想関数だけを定義するクラスを定義する場合があります、これを C++ では抽象ベースクラス (Abstract Base Class, ABC) という。

これに対し、通常の間数では派生されたクラス側で仮想関数をオーバーライドすることができる。つまり、特殊化した側でメンバー関数を事情に合わせて変更できる。もしも、ない場合にはベースクラス側のメンバー関数が使われる。すなわち、通常の間数を用いる目的は、派生クラスに間数のインタフェースと間数のデフォルトの実装を継承させる。しかし、この機能は便利のように見えるが、デフォルトの実装が間違いを引き起こすもとなる可能性があるので注意。

Java の場合には C++ からみれば、仮想関数のみであるといえる。また、インタフェースのみを定義する場合には、interface 定義という別の仕組みが用意されており、extends でなく、implements で継承することになっており、これについては概念的に整理されている。

さて、仮想関数でない通常の間数、派生されるクラスにインタフェースと強制的な実装の両方を継承させるという意味になる。つまり、特殊化しても変わらない機能を定義するものであり、原則、継承するクラス側では定義してはならない。

層化によって”has a”関係や”is implemented in terms of”関係を表現する (項目 40)

層化(layering)とはクラス定義の中にデータメンバーとして別のクラスのオブジェクトを定義することである。たとえば、

```

class Name { ... }; class Address { ... };
class Person {
private:
    Name name;
    Address address;
    ... }

```

この上でわかるように、この関係は”has a”関係である。また、集合 Set をリスト List で表現する場合には、

```

class Set: List { ...なかには、Set 用のメンバー関数... };

```

で表現できる。しかし、このようにしてしまうと、Set のオブジェクトからは、List のメンバー関数も呼べてしまうことになる。これを避けるためには、継承関係を private にするか、

```

class Set {
private:
    List rep;
    ... };

```

とすれば、よい。すなわち、層化は...を用いて実装する、”is implemented in terms of”関係を定義することになる。

Private な継承は、正しくつかう (項目 41)

上の例でみたとおり、private な継承の意味は、”is implemented in terms of”関係を定義することである。Set を使う場合には、ほかからは List のメンバー関数にアクセスすることはできない。ソフトウェアの設計の間には意味がなく、実装の時にのみに意味がある。層化が使える時には層化を使うべきであるが、private の継承を使う理由はコードが単純化できる場合があるからである。しかし、コンストラクタの呼ばれる関係など、複雑な場合があるので注意。

Java による分散プログラミング

RMI とは Remote Method Invocation の略であり、Java の分散プログラミングのための仕掛けである。この仕掛けをつかうことによって、いろいろなマシンにオブジェクトのインスタンスを生成し、これらの中で RMI を使って他のマシンのオブジェクトのメソッドを呼び出すことによって、分散システムを構築することができる。基本的には分散システムをプログラミングするためには TCP/IP や UDP など低レベルの通信レイヤをつかう。しかし、いちいち、機能ごとにプロトコルを設計して、通信しなくてはならない。このプロトコルを関数呼び出しに抽象化したのが、RPC(remote procedure call)である。有名なものとして SUN RPC があるが、現在これを使って、Unix のシステムのいろいろな機能が実装されている。RMI は、オブジェクト指向言語での RPC であり、オブジェクト指向の概念で分散システムをプログラムできるようにする。C++などの言語については、CORBA などが有名であり、RMI のほかに Java に対しても、CORBA 実装もある。

ネットワーク上のオブジェクトの転送

プログラミングという観点からみれば、TCP/IP がもっとも基本的で低レベルの通信手段である。このレベルでは単なるバイナリのデータの転送が提供される。Java では、以下のようにしてプログラミングする。C のレベルの Socket よりもだいぶ簡略化されている。

サーバー側：

```
ServerSocket ss = new ServerSocket(port);
Socket s = ss.accept();
DataOutputStream out = new DataOutputStream(s.getOutputStream());
x = out.writeInt(); /* write ...*/
```

クライアント側：

```
Socket s = new Socket(host, port);
DataInputStream in = new DataInputStream(s.getInputStream());
y = in.readInt(); /* ... read ...*/
```

Java では、オブジェクトそのものを書き出す Serialization 機能を持っている。これをつかえば、Serializable インタフェースを実装しているオブジェクトそのものを転送することができる。

```
ObjectOutputStream out = new ObjectOutputStream(s.getOutputStream());
out.writeObject(obj);
ObjectInputStream in = new ObjectInputStream(s.getInputStream());
Object obj = in.readObject();
```

ここで、readObject から返されるのはすべてのオブジェクトの superClass である Object として返されるため、適当なクラスに cast して用いる。このオブジェクトの転送では「データ」のみがネットワークに送信されることに注意。異なるマシンの中で転送する場合には、転送されるオブジェクトのクラス情報（つまり、プログラム）は両方のマシンで同じプログラムをもっていないとてはならない。

オブジェクトを転送する場合、転送先では少なくともオブジェクトを利用するわけであるから、オブジェクトの詳しい内容を知らなくても、何のメソッドが使えるかは知っているはずである。Java では、このことは内部の実装は知らなくても、どのようなメソッドがあるか、つまり、インタフェースだけをしっていると考える。ここで、例として、時刻を返すオブジェクトを考えると、

```
public class ShowDateImpl implements Serializable, ShowDate {
    public ShowDateImp() { ... } /* constructor */
    public long getCurrentMillis() { ... } /* 現在の時刻を返すメソッド */
    public long getMillis() { ... } /* オブジェクトが生成された時刻を返すメソッド */
}
public interface ShowDate {
    public long getCurrentMillis();
    public long getMillis();
}
```

転送先では実際のプログラムである ShowDateImpl は知らなくてもよく、そのインタフェースである ShowDate のみをしっていればよいことになる。そこで、

```
送信側： ShowDateImpl obj = new ShowDate();
        out.writeObject(obj)
```

```
受信側： ShowDate obj = (ShowDate)in.readObject();
        obj.getMillis();
```

とすればいいはずである。しかし、これをすると、obj.getMillis()のところで、実際のプログラムがない (ClassNotFoundError ShowDateImpl) というエラーになってしまう。obj.getMillis()を受信側で実行するためにはインタフェースだけでは不十分で、実際のプログラム ShowDateImpl が必要となる。

クラス情報の転送

そこで、クラス情報の転送する方法を考える。まず、クラスを転送するサーバを作る。これは.class のファイルを送信するサーバである。これに接続して、受信側でクラス情報をもたらすプログラムが NetworkClassLoader である。このプログラムでは、転送されたクラスファイルを ClassLoader の defineClass を使って、転送された.class ファイルの内容をクラスとして使えるようにする。これによって、ShowDateImpl をつくっておけば、上のプログラムは動作するようになる。

実際、ObjectInputStream では、resolveClass というメソッドを定義してやれば、ここで不明なクラス (定義されていないクラス) について、NetworkClassLoader をつかってクラスをロードすることによって解決することができる。

RMI でのオブジェクトの転送

RMI では、MarshaledObject を使って、オブジェクトの転送をしている。プログラムに sun.rmi.sever にある MarshalOutputStream と MarshalInputStream を使えば、ObjectInputStream と ObjectInputStream でのプログラムと同じような方法で同様なことができる。ただし、ここで、セットアップとして以下のことをしなくてはならない。

1. まずあらかじめ、ネットワークのクラスサーバ (web サーバでもよい) を立ち上げておく。
(http://localhost:8081)
2. 送るべきプログラムを jar ファイルにしておく。(dl.jar)
3. 送信側のプログラムには、どこからクラスをロードするか (codebase) を指定する。
4. 双方のプログラムについて、セキュリティマネジャーを設定し、起動時にはセキュリティポリシーを指定する。

MarshaledOutputStream では、オブジェクトをネットワークに送り出すときに、オブジェクトの復元に利用すべきクラス情報を含んだホストとディレクトリ情報 (codebase) を URL 形式で、埋め込み、送り出す。受信側の MarshalledInputStream では、そこから必要なクラスをロードしてオブジェクトを復元することになる。

送り手側のプログラムでは、以下のように指定する。

```
java -Djava.rmi.sever.codebase=http://localhost:8081/dl.jar
-Djava.security.policy=policy ObjectSever
```

MarshaledObject を利用すれば、同じようなことができる。

```
送信側： ShowDateImpl obj = new ShowDate();
        out.writeObject( new MarshalledObject(obj))
```

```
受信側： MarshalledObject mo = (MarshalledObject)in.readObject();
        ShowDate obj = (ShowDate)mo.get();
        obj.getMillis();
```

これまで、java の分散環境でのオブジェクトの転送について説明した。その要点は、

- 転送先でオブジェクトを参照するためには、インタフェースのみを共有しておけばよい。これは、Java の interface を用いて実現されている。実際のコード (の実装) に関しては転送される側は知る必要はない。
- Java のオブジェクトの転送機構である ObjectOutputStream はオブジェクトのクラス名とデータのみを転送する。したがって、転送されたオブジェクトを実際に動作させる (例えば、メソッドを呼び出す) 場合にはコードを転送する必要がある。
- コードを転送するためにクラスファイルを転送する機構を用意する必要がある。通常、このために http サーバを用いる。これを自動的に行うクラスが MarshalledObjectStream である。

実行時に `java.rmi.server.codebase` に指定する。

これらの機構は、Java の特徴的な機構であり、オブジェクトをネットワーク中で自由に転送することを可能にしている。RMI の引数や結果の転送に利用されている。

RMI の概要

RMI とは Remote Method Invocation の略であり、Java の分散プログラミングのための仕掛けである。この仕掛けをつかうことによって、いろいろなマシンにオブジェクトのインスタンスを生成し、これらの間で RMI を使って他のマシンのオブジェクトのメソッドを呼び出すことによって、分散システムを構築することができる。

オブジェクトの転送では転送されたオブジェクトのメソッドを呼び出し、いろいろな操作をするものであるが、RMI はリモートにあるオブジェクトのメソッドを呼び出す。以下の手順で行う。

1. インタフェースを、Remote インタフェースを extend して定義する。これをクライアント、サーバ、双方に置く。
2. サーバ側にはリモートのオブジェクトを管理するプロセスである `rmiregistry` を起動しておく。
3. また、サーバ側に仲介するプログラムである `stub` を生成するプログラムである `rmic` をつかって、`stub` を生成しておく。このプログラムは、Remote インタフェースから、スタブをプログラムを生成する。スケルトン `_Skel.class` とスタブ `_Stub.class` が生成される。
4. サーバ側のオブジェクトは、`UnicastRemoteObject` を `super` クラスとして作成し、サーバ側ではリモートのオブジェクトを登録する。
5. クライアント側では登録されているオブジェクトを取り出し、インタフェースを使って呼び出す。サーバ側のプログラムでは、リモートのオブジェクトを登録するために、

```
ShowDateImpl sdi = new ShowDateImpl();
```

```
Naming.rebind("//localhost/TimeSever",sdi)
```

で、登録している。このプログラムでは、前の例のように `codebase` や `policy` を指定して、起動しなくてはならない。例えば、

```
java -Djava.rmi.sever.codebase=file:/home/msato/java/my-jini/  
-Djava.security.policy=policy.txt ShowDateImpl
```

というように、コードのベースを指定する。これは `http` を含む URL でもよい。

クライアントプログラムでは、

```
obj = (ShowDate) Naming.lookup("rmi://localhost/TimeSever");
```

として、登録されているオブジェクトへの参照を得ることができる。これに対し、`obj.getMills()` と呼び出すことによって、サーバ側に登録されているリモートのオブジェクトのメソッドが起動されて、これらの引数、結果はオブジェクトとして転送される。内部では、指定されているホスト（ここでは `localhost`）で実行されている `rmiregistry` に接続し、`TimeSever` という名前登録されているリモートオブジェクトから、スケルトンのクラスをクライアントに転送する。このスタブは同じインタフェースをもち、引数を `MarshallObject` としてリモートオブジェクトに転送する。その後に対応するスタブを通じて、オブジェクトのメソッドを呼び出している。

Activation

前の例では、サーバ側のプログラムが `rmiregistry` に登録されるとリモートの呼び出しをずっと待つために待機している。しかし、いろいろなサービスを考えるといろいろなプロセスを起動しておかなくてはならなくなり、不便である。そこで、`UnicastRemote` の代わりに `java.rmi.activation.Activatable` というクラスを使えば、デーモン `rmid` を通じて、呼び出し時に起動させることができる。以下の手順で作る。

- `java.rmi.activation.Activatable` を extends してクラスを作る。
- コンストラクタとして、ID と引数データを引数とするコンストラクターを定義する。
- `activationGroup` のインスタンスを生成する。これは、`policy` や実行環境を定義するものである。
- `activation group` に登録し、ID を取得し、これを使ってグループを生成する。デフォルトのグループに登録。
- `activation descriptor` を生成する。これには、クラスの名前、クラスがロードされるべき

codebase、コンストラクタに渡される引数を指定する。activationGroup が指定しない場合にはデフォルトの group が使われる。

- descriptor を rmid に登録する。ここに stub が返される。
- これを Name.bind で、rmiregistry に登録する。
- あとは、プログラムは終了してよい。

このプログラムでは rmid デーモンを用いるが、このデーモンが id との対応をとり、ファイルに登録されているオブジェクトを起動する。rmid にも policy をしてしておくことを忘れずに。