Java による分散プログラミング入門

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、間違って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 Stroustrop 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、empolyee　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(deriverd)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 の引数が使える。
- 引数の Reference 渡しが使える。
- 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ー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こと。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。 Persion 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 Retangle: public Shape {   …. };
class Oval : public Shape { … };
```

Shape を継承する Rectanle 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。純粋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。純粋仮想関数だけを定義するクラスを定義する場合があり、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、通常の関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、通常の仮想関数を用いる目的は、派生クラスに関数のインタフェースと関数のデフォー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、デフォールトの実装が間違いを引き起こすもとに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、inteface 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非仮想関数は、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”関係を定義するということになる。

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 のメンバー関数をアクセスすることはできない。ソフトウエ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。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使つかう。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.getInputSteram());
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。異なるマシンの間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くてはならない。

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しらなくても、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しらなくても、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いいことになる。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らうプログラムが 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、MarshalledObject 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。だだし、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ー（web サーバーでもよい）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。

MarhalOutputStream 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る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、送りだす。受信側の MarshallInputStream 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
```

MarshalledObject 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 ObjectStream 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. サーバー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 ShowDateImp();
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、サーバー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、それらの引数、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 という名前で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、サーバー側のプログラムが 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.Actvatable 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。デフォー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ールトの 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。

Java による分散プログラミング　～Jini,.～

Jini はこの分散オブジェクトプログラミングをベースに、いろいろなコンピュータ、家電に入っているプロセッサからスーパーコンピュータまで、ネットワーク上のあらゆる機器（コンピュータ）を「連合(federation)」させるための仕組みを提唱したものである。たとえば、いろいろな家電製品にはいまやプロセッサが入っているが、これをネットワークにつなぎ、RMI（というか、RMI で提供されている標準のプロトコルと Jini によって提供されるサービスの検索機能）でつなぐことによって、いろいろな家電を統一的に制御したり、利用したりできるようになる。Jini のもっとも重要な概念として「サービス」がある。ネットワーク上に接続されているコンピュータを単なるデータを交換する対象と考えるのではなく、なんらかのサービスを提供する対象と考える。そのサービスをお互いに交換することによって、分散システムはなんらかの仕事をする。これまで、いわゆるサーバはサービスを提供する担い手であり、クライアントはそのサーバからサービスを受ける形態が一般的であったが、Jini が想定しているのはネットワーク上の分散システムを構成するコンピュータがお互いにサービスを提供することによって協調作業をするシステムを想定している。

Jini でサービスをネットワーク上のどこからでも利用できる。サービスはネットワーク上を移動するオブジェクトによって提供される。いろいろなサービスがあるとする Jini では、そのサービスを見つけるための機構「Lookup サービス」が提供されている。これによって、ネットワーク上に提供されているサービスを検索し、そのサービスを利用できる。これについては、たとえば DHPC を考えるとわかりやすい。いまでは、ノート PC を単にケーブルを接続するだけで、ネットワークに参加できるが、ケーブルを接続したときにまず、ネットワークのアドレスを管理している DHPC サーバを検索し（これが Lookup、つまり DHCP のサービスの検索）、標準のプロトコルでアドレスやネットマスク、DNS などのアドレスを取得する。また、サービスを提供する側は、Lookup サービスに登録することを Join と呼んでいる。

RMI は個々のコンピュータで提供するオブジェクトを管理する（registry）機能を提供しているが、Jini はこれをネットワーク全体に拡張し、すべてのコンピュータで提供されている機能を検索する機能を提供するものということもできる。

Jini

Jini は以下の３つの部分からなっている。

1. JCP(The Jini Technology Core Platform)
2. JXP (The Jini Technology Extended Platform)
3. JSK(The Jini Technology Software Kit)
4. JSTK(The JavaSpace Technology Kit)

ここでは、上の３つを用いる。Jini を使う前に、以下の手順で Lookup サービスを立ち上げる。

- HTTP サーバの立ち上げ：Jini 自体のプログラムもいろいろなところで動かすことができるようにするために、httpd からロードできるように http サーバを立ち上げておく。
- RMI Activation Demon：　Lookup サービスは activation をつかっているので、rmid を立ち上げておく。
- Jini Lookup サービスの立ち上げ：Jini ではいくつかの lookup サービスのサーバがあるが、その一つである reggie を以下のようにして立ち上げる。
  `Java –jar reggie.jar http://hostname/reggie-dl.jar policy log_file myName`
  ここで、`http:`は上の `httpd` が立ち上がっているホストを指定する。

このほかに、`Lookup` サービスを立ち上げるための `GUI` や、状態を確認する `Browser` がある。

`Jini` の中核になるアイデアはいろいろなサービスを検索して、必要なサービスを利用できる環境を提供することである。さて、実際のプログラムをみてみることにしよう。最初の例は、`TimeService` オブジェクト（以前、`ShowDate` と読んでいたものと同じ）を登録し、それを利用する例である。まず、サーバ側では、

- 登録する `Lookup` サーバに対して、`LookupLocator` を作成し、ここから、`Registrar` を取得する。
- `TimeService` とプロパティから `ServiceItem` を作成し、この `registrar` に対し、登録する。

このサーバーを動かすためには、このプログラムをコンパイルし、起動する際に、このプログラムのクラスを供給する http サーバを用意しておかなくてはならない。このプログラムを Lookup サービスがもちいている http サーバと共用してもいいが、別でもよい。例えば、別にして 8081 にこのクラスのための http サーバーを立ち上げているとすると以下のようにして起動する。

```
Java –Djava.security.policy=policy.txt
–Djava.rmi.server.codebase=http:/HOSTNAME:8081/ setup
```

クライアント側は、

- まず、指定された URL から、LookupLocator をつくり、ここから、Registrar を取得する。
- 検索するクラス（インタフェース）を指定して、テンプレートを作成する。
- このテンプレートより、検索し、オブジェクトを取得する。

LookupLocator は、Lookup サービスが立ち上がっている場所を指定するオブジェクトである。しかし、Lookup サービスをそのものではない。Lookup サービスの本体は、ServiceRegistrar であり、これに対し、以下のようにして取得する。

```
LookupLocator locator = new LookupLocator(“jini://myhost”);
ServiceRegistrar registrar = locator.getRegistrar();
```

オブジェクトを登録するには、サービスのデータを作成して、登録する。

```
ServiceItem sit = new ServiceItem(…);
ServiceRegisteration sre = registrar.regsiter(sit,Lease.FOREVER);
```

ServiceItem は、サービス名、バージョン番号などを含んだ属性とオブジェクトを元に作る。
クライアント側では、検索するテンプレートを作成し、これを元に検索する。

```
ServiceTemplate tmpl = new ServiceTemplate(…);
Object service = registrar.lookup(tmpl);
```

テンプレートには、検索するクラスを指定する他、属性からサービスを探すこともできる。詳細は省略する。

また、これまでのプログラムに policy ファイルを指定して実行するが、これはどのような人がどのような権限で実行できるかを細かくしているものであるが、詳細は省略する。

Lookup サーバの検索

前述の LookupLocator の例は、Lookup サーバがわかっている場合を想定している。Lookup サーバから取得した ServiceRegistrar 内で、いろいろなタイプのサービスを検索できるようになっている点は、RMI と違うものの、RMI と大差がないとも言える。

Jini の想定する環境では、クライアントは Lookup サーバがどこにあるのかを情報を持っていないことがありうる。このためのクラスが、LookupDiscovery である。次の例はこのクラスを使う例である。どこにあるかわからないサーバーを探すために、Multicast のプロトコルを使っている。

Jini の Lookup サービスには「グループ」という概念がある。一つの Lookup サービスは複数のグループに属することができるし、複数の Lookup サービスが同一のグループに属することができる。グループについては、文字列の配列で表現している。但し、空の文字列の場合は特定のグループには属さないと解釈される。

LookupDiscovery が Mulitcast を使って discovery を行うドメインを Jini では、djinn(ジン)と呼んでいる。discovery には、３つのプロトコルが使われている。

multicast request プロトコル：自分が属する djinn に対して、マルチキャスト Lookup がないかを問い合わせる。

multicast announce プロトコル：Lookup サービスが、自分の存在をアナウンスするためのプロトコル。新しいサービスが加わったり、ネットワーク障害等で切断したり、復旧したりするときに用いる。

unicast discovery プロトコル：特定の Lookup サービスと通信するプロトコル。上の例。

LookupDiscovery オブジェクトは、グループを指定して作成される。

```
lookupDiscovery = new LookupDiscovery(groups);
```

Lookup サーバが見つかったときのインタフェースとして、イベントリスナーモデルを使っている。そのため、リスナーとして登録するオブジェクトは、DiscoveryListerner のインタフェースを定義していなくてはならない。

```
public class TimeServiceImpl implements DiscoveryListener …. {
```

```
  …..
  lookupDiscovery.addDiscoveryListerner(thisObj);
  ….
  public void discovered(DiscoveryEvent ev){ ….. /* 見つかった時の action */ }
  public void discarded(DiscoveryEvent ev) { … /* 切れた時の action */ }
…. }
```

discovered は Lookup サービスが見つかったときに呼び出されるメソッドである。ここで、引数になっているイベントから、ServiceRegistrar を取り出す。これについては、複数の Lookup サービスがありうるので、配列になっていることに注意。

```
   ServiceRegistrar[] regs = ev.getRegistrars();
```

サーバ側では、これを用いて、TimeServiceImpl を TimeService として登録している。登録の手順は前の Unicast のものと同等である。逆に、クライアント側ではどれか一つの ServiceRegistrar から、オブジェクトを取得している。

ちなみに、このプログラムでは Listener に設定してから、検索が終了し、リスナーが呼び出されるまでに時間がかかるため、設定後、sleep して待っている。

Jini: RMI の場合

次の例は、RMI をサービスとして登録するものである。違いは、サービスが Remote を extend して定義し、RMI できるようにしてある。この場合、Remote インタフェースを持つオブジェクトは、クライアント側に移動するのではなく、サーバ側にとどまって、サーバ側で実行される。そのため、このプログラムを実行する時には、rmic によって、スタブを生成しておくことが必要である。また、このプログラムではリスナーが見つかり登録するための部分を別のスレッドにして実行している。

リースの概念、その他

Jini のプログラミングの大きな特徴の一つに「リース」という考え方がある。つまり、あるオブジェクトが他のオブジェクトに貸し出す期間を設定し、その期限が過ぎると使えなくなるというものである。

RMI や RPC はリモートの呼び出しもあたかもローカルなもののように見せることによって、プログラミングを容易にするものである。しかし、Jini ではこの点をあえてユーザに見せるようにしている。例えば、分散環境ではリモートのコンピュータが壊れたり、ネットワークに障害が起こるかもしれない。このような環境ではあらかじめこのようなことを起こることを想定するプログラミングが必要となる。Jini のリースでは、あらかじめ決められた時間が来るとサービスは終了し、リソースは自動的に開放される。リースの期限が期限になる前に、更新するかどうかの対処を行う。リソースの管理を一定時間ごとにできることだけでなく、これにより障害等に強いソフトウエアを作成することができる。

このほかにも、Jini には Java でのイベントリスナーモデルを分散環境に拡張した分散イベント(distributed event)の仕組みがある。イベントも分散オブジェクトとして登録され、lookup サービスを通じてやり取りが行われる。また、データベースの更新などの同期を取るために、トランザクションをサポートする機構などもサポートされている。