解　説

Jini™による分散サービス環境

山口　浩

サン・マイクロシステムズ株式会社
Javaセンター技術企画室

Sun Microsystems社が1999年1月に発表したJiniテクノロジーは，電子機器を接続するための技術として注目を浴びているが，この技術自体はハードウェア接続のためだけのものではなく，もっと広範囲の分散環境を扱うことを目的として開発されたものである．その基盤にはJava™テクノロジー，特にRMIと呼ばれる分散オブジェクトの技術があり，これらを活用してネットワーク環境のさまざまな資源を柔軟かつ効果的に利用することを目指している．本稿では，Jiniテクノロジーを用いた分散オブジェクト環境について簡単な例をまじえて紹介する．

Jiniテクノロジー登場の背景

1995年にSun Microsystems社が発表したJava言語は，Java仮想マシンというソフトウェアのレイヤを用いることで，マシンアーキテクチャやオペレーティングシステムの違いを取り払い，一度コンパイルしたプログラムが，そのまま異なるプラットフォームで動作するというWrite Once, Run Anywhere™の実現を可能にした．このようなJavaのプログラムの実行環境をJavaプラットフォームと呼ぶ．これまで，ソフトウェアの開発者は1つのソフトウェアを種類の違ったハードウェアやオペレーティングシステムの上で動かすという問題を解決するためにさまざまな試みを行ってきた．Javaプラットフォームの登場は，このような問題に対する1つの解決方法，それも非常に単純な形の解決方法を提供するものである．もちろんJavaプラットフォームはマルチプラットフォームに関する万能薬を提供するものではないが，多くの場合，開発者に，より本質的な問題の解決に専念する時間を与えることになるであろう．

ここでネットワークに関係するアプリケーションプログラムに目を向けてみよう．近年，クライアント・サーバ型のアプリケーションやWWWブラウザにCGIプログラムを組み合わせたタイプのアプリケーションが基幹業務で用いられるプログラムとして盛んに開発されるようになっている．こういったネットワーク・アプリケーションは，いわゆるスタンドアローンのアプリケーションに比べると本質的に解決の難しい次のような問題を含んでいる．

1. 異種混合環境への対応
2. 部分的な障害への対応
3. 処理の遅延に対する対応
4. 安定性の確保
5. スケーラビリティへの対応
6. セキュリティ問題への対応

分散オブジェクト指向はこのようなネットワーク・アプリケーション環境における問題を一般的に解決するための重要な技術として開発されてきた．

OMGによるCORBAは，現在，分散オブジェクト技術の中で最も広く使われているものであるが，Javaプラットフォームは，これとは別の，より簡単なアプローチを可能にした．Java言語は元来オブジェクト指向であり，しかも最も広く利用されているTCP/IPベースのネ

ットワーク機能を利用するためにソケットの基本パッケージを備えている．また，CORBA等で分散オブジェクトの外部仕様を記述するために使用されるIDL（Interface Definition Language）に相当するinterfaceという概念も備えている．このような基盤があったためにJava言語で分散オブジェクトを扱うための機能拡張はきわめて自然に行われ，1997年にリリースされたJDK1.1にRMI（Remote Method Invocation）として登場した．

RMIを用いた分散オブジェクト環境

普通のJavaプログラムはJavaVM（Java Virtual Machine）と呼ばれる実行環境上のいくつかのオブジェクトとして実現されているが，RMIを用いると，あるJavaVM上のオブジェクトから別のJavaVMの上にあるオブジェクトの呼び出しを実現することができる．このような呼び出しには，実際はネットワークを用いるためのさまざまな処理が伴うが，それらは，ほとんどがRMIで提供されるクラスライブラリに隠蔽されてしまっているため，プログラマはまったく意識する必要がない．その結果，通常のJavaオブジェクトの操作とほとんど変わらないコードでネットワーク環境に対応するプログラムを書くことができる．

本稿では，例としてFAX送信を行うようなクライアント・サーバプログラムを取り上げる．このサーバプログラムはクライアントからFAXデータと送り先のFAX番号を受け取り，それを送り先のFAXに送信する機能を持つものとする．サーバの構成について詳しくは触れないが，ネットワークに接続されたコンピュータでFAXモデムが付属しているようなものを用いて実装されていると考えればよい．さてFAXサーバの利用者（クライアント）にとって重要なものは，FAXデータとFAX番号をどのようにして渡すかというインタフェースである．Java言語では，これをinterfaceというキーワードを用いて次のように記述する．

```
interface FaxSender {
  // 送り先の FAX 番号をセットする
  void setFaxNo(String faxNo);
  // 送信データをセットする
  void setFaxData(FaxData data);
  // 送信を開始する
  void send();
  // 送信を中止する
  void cancel();
  // 送信ステータスを調べる
  int getStatus();
}
```

従来の手法でクライアント・サーバ型のプログラムを書く場合には，サーバ側では，このインタフェースで定義されたメソッドを実装すると同時に，クライアントとのデータ交換を行う必要があり，この部分はエラー処理などを考慮するとかなり複雑である．また，クライアント・サーバ間のデータフォーマット（プロトコル）を設計し，それに従ってクライアントプログラムを実装する必要がある．ところがRMIを用いると，最初のインタフェースの実装部分を作成するだけで，クライアント・サーバ間のデータ交換については，一切考える必要がなくなる．以下に，例を示そう．

先に挙げたinterfaceはFAXサーバの機能を定義するためのインタフェースとしては十分であるが，RMIで用いるためには，次のような若干のコードの追加が必要である．

```
interface FaxSender extends Remote {
  void setFaxNo(String faxNo)
    throws RemoteException;
  void setFaxData(FaxData data)
    throws RemoteException;
  void send()
    throws RemoteException;
  void cancel()
    throws RemoteException;
  int getStatus()
    throws RemoteException;
}
```

まずextends Remoteは，FaxSenderインタフェースがRMIで用いられるリモートオブジェクトのインタフェースを表すことの宣言である．また各メソッドにthrows RemoteExceptionが追加されているのは，これらのメソッドがRMI下で呼び出される場合にRemoteExceptionという例外を発生する可能性があるためである．

RMIを用いたサーバのプログラムはこのインタフェースの実装を記述した通常のJavaプログラムで，特にネットワークを意識するようなコードはほとんど必要ない．ここでは実装クラスをFaxSenderImplとし，そのコードを次に示す．このクラスには，FaxSenderで定義されたすべてのメソッドの実装が含まれている．

```
class FaxSenderImpl
    extends UnicastRemoteObject
    implements FaxSender {
  private String faxNo;
  private FaxData data;

  FaxSender() throws RemoteException {
    super();
  }
  void setFaxNo(String faxNo)
      throws RemoteException {
```

```
    this.faxNo = faxNo;
  }
  void setFaxData(FaxData data)
      throws RemoteException {
    this.data = data;
  }
  ...
}
```

次に，このサーバのプログラムをネットワーク上の別のJavaVMで動くクライアントからアクセスできるようにするための準備を行う．RMIには，オブジェクトに名前をつけて，ネットワークから見えるようにするレジストリとしてrmiregistryが用意されている．いま，上記のサーバプログラムをFaxServerという名前で，レジストリに登録する．

```
try {
  System.setSecurityManager
    (new RMISecurityManager());
  FaxSender faxSender =
    new FaxSenderImpl();
  Naming.bind("FaxServer", faxSender);
} catch (Exception e) {...}
```

ここでsetSecurityManager(...)はRMIが後述するクラスのダイナミックローディングを行うためのセキュリティマネージャの設定である．このあとFaxSenderImplのオブジェクトを生成し，bind( )メソッドを使ってFaxServerという名前でレジストリに登録している．

一方クライアントのプログラムは次のようになる．

```
System.setSecurityManager
  (new RMISecurityManager());
// FAX データの作成
FaxData data = generateFaxData(...);
String faxNo = "03-4567-8901";
try {
  String url = "rmi://fax/FaxServer";
  FaxSender fax =
    (FaxSender)Naming.lookup(url);
  fax.setFaxNo(faxNo);
  fax.setData(data);
  fax.send();
  ...
} catch (RemoteException e) {...}
```

FAXデータとFAX番号を入手した後，lookup( )メソッドでレジストリからサーバオブジェクトのリモート参照を入手する．この時に指定するURLは

```
rmi://サーバホスト名/リモートオブジェクト名
```

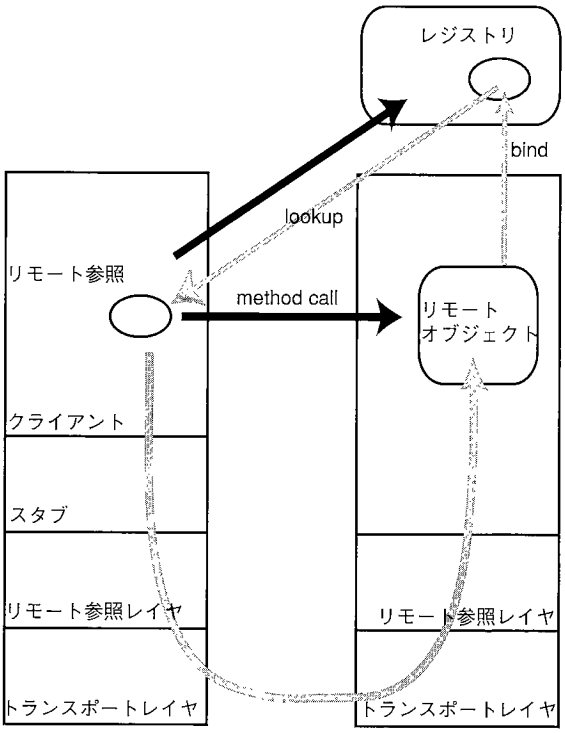

図-1　リモートオブジェクトの呼び出し

のような形式である．これで得られたリモート参照はFaxSenderインタフェースの型を持っており，通常のJavaのオブジェクトと同様にメソッド呼び出しが行える．ただし，これはリモート参照なので，setFaxNo( )やsetData( )，send( )等のメソッドの実行はサーバ側で行われる．

なお，これらのリモート参照を用いる処理は，実際にはネットワークを経由するため，通信時のエラーが発生する可能性がある．しかし，この例のようにtry-catchを用いることでエラーを通信時の例外として捕捉できるので，エラー処理はまとめてcatch節に書いておけばよい．

以上のように，RMIを用いることで，クライアント・サーバ型のプログラムを，ネットワークをほとんど意識しないスタンドアロンのプログラムのように書けることが理解できるであろう．実際にはクライアントからのリモート参照に対する呼び出しは，クライアント側にあるスタブを経由して図-1のように行われている．したがって，プログラム上はサーバ側を直接呼び出しているように見えるクライアントも，実はクライアント側にあるスタブを呼び出しているに過ぎない．

なお，実行時にクライアント側で必要なスタブは次のようにFaxSenderImplのクラスファイルからrmicコマンドを使って生成する．

```
% rmic -v1.2 FaxSenderImpl
```

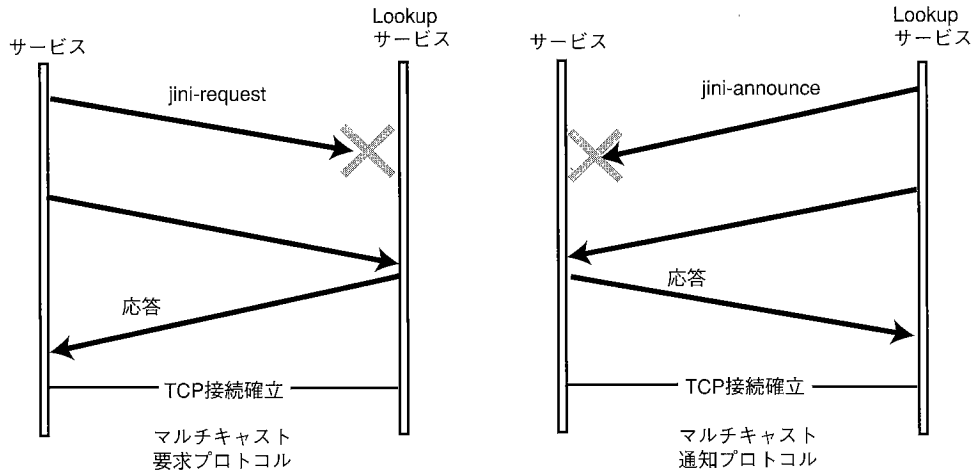

図-2　マルチキャスト要求／通知プロトコル

この結果，FaxSenderImpl_Stub.classというスタブファイルができるが，これは，RMIのダイナミックローディングの機能を用いることで，実行時にクライアント側にダウンロードすることができる．したがって，クライアントにはサーバの実装に関する情報は不要である．

サーバからサービスへ

前節で取り上げたFAXサーバには，いくつか改良すべき点がある．たとえばFAXの利用者が増加したためにFAXサーバを何台か増やすような場合を考えてみよう．このためには，別のFAXモデムを備えたサーバ機を用意する，あるいは現在のサーバ機にいくつかのFAXモデムを追加する，といった方法が考えられるが，いずれの場合にも，クライアントには複数のサーバのうちから，自分が利用するサーバを選択するような仕掛けを組み込む必要がある．また，サーバ側でも，新しく名前を決めて，レジストリに登録し，同時に利用者にアナウンスするといった作業が必要となる．このような問題はしばしば現実のシステムでも起こり得るが，その解決はそれほど簡単ではない．

ここで問題の解決のために，サーバを「クライアントに対してサービスを提供するもの」という形で捉えてみよう．これはクライアントがサーバオブジェクトを呼び出す時に，その名前ではなく，それが提供する機能によってオブジェクトを特定できるようにしようというアプローチである．

もし「FAXを送るサービス」を探すという形でクライアントを書くことができれば，サーバの数が増えてもクライアントの変更は不要である．また，サーバを追加しても，それを利用者にアナウンスする必要もなくなるであろう．このようにサービスという概念を導入することによって，分散サービス環境を提供しようというのがJiniテクノロジーの目指すパラダイムである．

Jiniテクノロジーにおいてサービスはネットワーク上のさまざまなリソースを表す基本的かつ重要な概念である．これにはデータベースへのアクセスや科学技術計算のソフトウェア，あるいはネットワークに接続されたFAXやプリンタ，ディジタルカメラを利用するためのソフトウェア等，一般にユーザが何らかの仕事を行うために必要となるコンポーネントが含まれる．またサービスは，それ自体が別のサービスのユーザとなることもでき，複数のサービスをグループ化することで，より複雑で高度な機能を実現することも可能である．Jiniテクノロジーによるソフトウェア・パラダイムは分散システムをこのような相互に関連するサービスの連合体によって組み立てようとするものである．

Lookupサービス

ネットワークで利用できるサービスの論理的な集合はdjinn（ジン）と呼ばれ，1つのネットワークに複数存在することができる．これらはグループ名を持っており，グループ名によってdjinnを特定できるようになっている．Jiniの世界はネットワークのユーザがグループ名を指定することで特定のdjinnを選択し，そこに自分のサービスを追加する，あるいはさまざまなサービスを探し出し，それらを用いて1つの仕事を行うように作られている．このようなサービスの登録や検索を行うために提供されているのがLookupと呼ばれるサービスである．

Jini環境ではすべてはLookupを利用することから始まるが，Lookup自体もJiniのサービスとして実現されているため，Lookupを探すためには別の仕組みが必要である．この仕組みはDiscoveryと呼ばれ，Jiniテクノロジーが提供する最も興味深い技術の1つである．なおLookupサービスの機能については，いくつかのAPIが定義されているが，実装方法は任意である．Sunから仕様書とともに提供されているJini Software Kitには，SunのJiniグループによる実装であるReggieというプログラムが含まれている．

DiscoveryによるLookupサービスの検索

たとえば，あるサービスがdjinnに参加したい場合，そのサービスは，まず自分自身のIPアドレスと適当なポート番号を含んだパケットをネットワークにマルチキャストする．ネットワーク上のdjinnのLookupサービスは，このようなパケットを常に監視していて，パケットを受信するとその中に書かれてあるIPアドレスとポート番号に対しTCP接続を試みる．これにサービスが応答することで両者の間にコネクションが成立しLookupサービスはこのコネクションを通じて，自分自身の参照をサービスに渡す．

通常は，このようにLookupサービスがすぐに応答するが，何らかの理由で応答しない場合がある．そのような場合には，サービスは同じパケットを5秒程度の間隔で，何回か再送しLookupサービスの応答を待つ．この一連のプロトコルをマルチキャスト要求プロトコルと呼ぶ．

マルチキャスト要求プロトコルでLookupサービスが応答しない場合は，次のマルチキャスト通知プロトコルに切り替える．これは，Lookupサービスが常に120秒程度の間隔で自分自身の存在を知らせるパケットをマルチキャストしているので，それをdjinnに参加したいサービスが待ち受けるというものである．パケットにはLookupサービスのホストとポート番号が書かれており，パケットを受け取ったサービスは，これを使ってLookupサービスにTCP接続し，Lookupサービスの参照を受け取る（図-2）．

Lookupサービスの存在するホストが分かっている場合には，これらのマルチキャストを利用する方法は不要であり，ユニキャストを用いる通常のTCP接続を用いて，Lookupサービスのリモート参照を得ることができる．この時に用いられるプロトコルはユニキャストDiscoveryプロトコルと呼ばれ，通常のLAN環境だけではなくマルチキャストが届かないような遠くのLookupサービスを見つけることもできる．

以上の3つのプロトコルがDiscoveryプロトコルと呼ばれる．Discoveryプロトコルはdjinnに参加しようとしているサービスで用いられるだけでなく，djinnのサービスを利用しようとしているユーザやクライアントプログラムも利用することができる．

ここで，マルチキャストのDiscoveryプロトコルを用いて，ネットワーク上にあるすべてのLookupサービスの参照を得るプログラムを見てみよう．

```
ServiceRegistrar registrars[];

LookupFinder() {
  ...
  LookupDiscovery ld;
  ld = new LookupDiscovery
    (LookupDiscovery.ALL_GROUPS);
  ld.addDiscoveryListener(this);
  ...
}

public void discovered(DiscoveryEvent e)
{
  registrars = e.getRegistrars();
}
```

ここでLookupDiscovery( )は先に述べた2種類のマルチキャストプロトコルによってLookupサービスを探すスレッドを生成するコンストラクタであり，Lookupサービスが見つかると，それらを分散イベント（DiscoveryEvent）としてdiscovered( )に渡すようになっている．そして，このイベントにgetRegistrars( )を用いることで，見つかったすべてのLookupサービスのサービスレジストラを得ることができる．

サービスをLookupサービスに登録するには，そのサービスIDとサービスを表すオブジェクト（プロキシ），それにサービスの利用者が検索で使用するための属性を記述したサービス項目を作成する必要がある．ただしサービスIDは，Lookupサービスに登録する際にLookupサービスによって割り当てられるサービス固有の情報である．Lookupサービスへの登録のプロセスは，Joinプロトコルと呼ばれ，次のようなステップから構成されている．

1. パケットストームの発生を防ぐためランダムな時間（最大15秒程度）待つ
2. Discovery プロトコルを用いてLookupサービスを見つける
3. サービス項目とリース期間を指定してLookupサービスに登録する
4. リースが切れる前に登録を更新する

ここで，登録にはリース期間を指定して行う必要があ

ることに注意しておく．Lookupサービスは指定されたリース期間が経過するとそのサービスの登録を抹消するようになっている．したがって，引き続き登録状態を保つためにはリースが切れる前に更新処理を行う必要がある．リースはネットワーク資源を有効利用するためJiniテクノロジーで提供されている重要な機能の1つである．

以下に，先にRMIのオブジェクトとして実現したFAXサーバをFaxServiceというサービスとしてLookupサービスに登録する例を示す．ここでは，簡単のためサービス項目に含まれる属性として，サービス名だけを用いる．また，以下に登場するregistrarsは，前述したregistrars配列の1つのエントリである．実際にはサービスを参加させたいdjinnのLookupサービスすべてに登録を行う必要がある．

```
Entry attribs[] = new Entry[]{
  new Name("Fax Service")};

FaxSender FaxSender
  = new FaxSenderImpl();
ServiceItem item = new ServiceItem
  (null, FaxSender, attribs);
registrar.register(item, Lease.ANY);
```

ここで，サービスの登録にはLookupサーバを明示的に指定する必要はない点に注意して欲しい．

今度はこのサービスを利用するクライアントを見てみよう．クライアントがdjinnに存在するサービスを見つけるためには，やはりLookupサービスが使われ，その流れはほとんど同じである．以下，異なる部分だけを簡単に示す．サービスの検索にはテンプレートと呼ばれるオブジェクトが使われる．これは，Lookupサービスに登録されているサービス項目をマッチングを使って検索するために利用される．いま，サービスの名前としてFax Serviceを持つものを検索するとしよう．この時のテンプレートは以下のようなものである．

```
Entry attribs[] = new Entry[]{
  Name("Fax Service")};
ServiceTemplate template =
  new ServiceTemplate(null,null,attribs);
```

ServiceTemplateのコンストラクタの最初の2つの引数は，サービスIDとサービスオブジェクトの型を表すものであるが，nullを指定するとすべてにマッチするワイルドカードとして働く．このテンプレートにマッチするサービスは，

```
FaxSender FaxSender =
  (FaxSender)registrar.lookup(template);
```

で得られる．ここでFaxSenderはFaxSenderオブジェクトのリモート参照であり，クライアントは，これを先に述べた方法とまったく同じように使用することができる．

ここで，最初の問題に戻ってみよう．FAXサーバをこのようなサービスの形で書くことにより，FAXサーバの追加は単なるFax Serviceを提供するエンティティの増加になる．またクライアントもサーバの実体を意識することなく何らかのサーバを使用することになる．これらのことから，サービスを利用することで安定性やスケーラビリティの問題も同時に解決できることが分かるであろう．

まとめ

本稿では，Jiniテクノロジーに含まれる基本技術のうちLookupサービスに焦点を絞って解説した．例として用いたFAXサービスを，他のネットワーク機器に置き換えれば，Jiniテクノロジーがネットワーク機器の接続技術として注目を浴びている理由が容易に推察できると思うが，ハードウェアが関係しないネットワーク上のサービスにもさまざまな形で適用できる技術であることも明白であろう．

実は，Jiniテクノロジーの基本技術には，Lookupサービス以外にもDiscoveryプロトコルで登場した分散イベントや，Lookupサービスで用いられるリースをはじめとして，いくつかの重要な概念が導入されているが，誌面の都合もあり十分紹介できなかった．本稿でJiniテクノロジーに興味を持たれた読者は，ぜひ，Jiniテクノロジーの他の技術についても調べていただきたい．本稿が，そのきっかけとなれば幸いである．

参考文献
1) Jini アーキテクチャの仕様,
http://www.sun.co.jp/java/software/jini/specs
2) Discovery と Join の仕様,
http://www.sun.co.jp/java/software/jini/specs
3) Jini Lookup サービスの仕様,
http://www.sun.co.jp/java/software/jini/specs
4) The Java Tutorial,
http://java.sun.com/docs/books/tutorial

(平成11年6月3日受付)