情報知能工学演習V

RMIによる分散オブジェクトプログラミング

担当教官：村尾 元 (murao@al.cs.kobe-u.ac.jp)
TA：稲元 勉 (inamoto@al.cs.kobe-u.ac.jp)

Last Updated: 2002年7月12日（金）

目 次

第 1 章　分散オブジェクト **5**
1.1 分散オブジェクトとは 5
1.2 Java における分散オブジェクト 6
1.3 分散オブジェクトの実現 8
1.3.1 インターフェイス 8
1.3.2 スタブとスケルトン 9
1.3.3 オブジェクトリファレンス 9

第 2 章　RMI を用いた分散オブジェクト **11**
2.1 インターフェイス 11
2.2 リモートオブジェクト 11
2.3 スタブとスケルトン 12
2.4 オブジェクトリファレンス 12
2.4.1 ネーミング・サービスの実行 12
2.4.2 オブジェクトリファレンスの登録 13
2.4.3 オブジェクトリファレンスの参照 13
2.5 オブジェクトの転送 14
2.6 セキュリティ 14
2.6.1 セキュリティマネージャ 14
2.6.2 セキュリティポリシーの設定 15
2.6.3 RMI で必要なセキュリティポリシー 15

第 3 章　RMI を用いた分散オブジェクトプログラムの作成と実行 **17**
3.1 簡単なプログラムの作成 17
3.2 簡単なプログラムの実行 20
3.3 セキュリティマネージャを用いたプログラムの作成と実行 22
3.4 RMI を用いたアプレットの作成と実行 25

第1章　分散オブジェクト

1.1　分散オブジェクトとは

分散システムでは，異なる計算機上で実行されているプログラムが相互に通信できる必要がある．先の演習で行ったように，Java 言語は通信のための基本メカニズムとしてソケットをサポートしている．これは柔軟性に富み，一般的通信には十分な機能を備えている．しかし，ソケットでは，通信に要する多くの部分をアプリケーション作成者が実装する必要があり，これは容易ではない（図 1.1）．

これを比較的容易に解決する方法として分散オブジェクトという技術がある．分散オブジェクトとは，一言で言うと「異なる計算機間でオブジェクト同士がメッセージ通信を行うための技術」である．これを用いるとネットワークを介して異なる計算機上にあるオブジェクトが，あたかも同一の計算機上にあるかのようにプログラムを行える．

その大ざっぱな仕組みは図 1.2 のようになっている．つまり，クライアント側で，サーバ側のオブジェクトの代理オブジェクト（proxy object）を生成し，これにメッセージを発行すると，代理オブジェクトは対応するサーバ側のオブジェクトにメッセージを送信し，サーバ側のオブジェクトがメッセージを処理するといった感じである．この代理オブジェクトがサーバ側のオブジェクトに発行するメッセージをリモートメッセージとも言う．

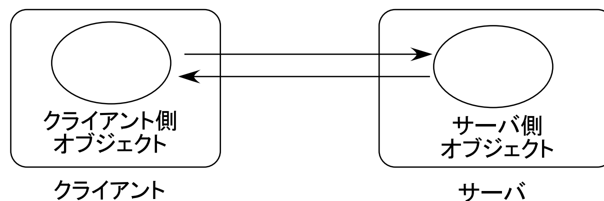

図 1.1: ソケットによるオブジェクト間通信：クライアント側オブジェクトとサーバ側オブジェクトの通信手順は全てユーザが作成する必要がある．

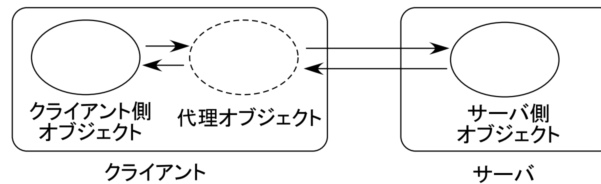

図 1.2: 分散オブジェクト : 代理オブジェクトとサーバ側オブジェクト間の通信はあらかじめ提供されるので，クライアント側オブジェクトは代理オブジェクトとメッセージのやりとりを行えばよい.

1.2　Java における分散オブジェクト

分散オブジェクトは，未だ発展中の分野であり，Java でも様々な分散オブジェクト技術が展開されている．その中でも有名なものとして，SUN による RMI（Remote Method Invocation : リモートメソッド呼び出し）と，電総研の平野氏による HORB（Hirano Object Request Broker) がある．また，分散オブジェクトの標準技術として仕様化が進められている CORBA（Common Object Request Broker）に対応した JavaIDL が Java1.1 より用意されている．

いずれの分散オブジェクト技術も，代理オブジェクトを介してリモートオブジェクト（サーバ側のオブジェクト）と通信するという基本は同じであるが，RMI や CORBA と HORB は，リモートオブジェクトの生成のタイミングが異なる．RMI や CORBA は，すでにサーバ側で用意されているオブジェクトに代理オブジェクトを介して接続するという形式であり，これは**接続モデル**と呼ばれる．HORB は接続モデルでの利用も可能であるが，代理オブジェクトが生成されると，対応するリモートオブジェクトも自動的に生成されるという**生成モデル**をサポートしている点が大きく異なる．

生成モデルでは，クライアント側で代理オブジェクトを生成すれば，対応するリモートオブジェクトが自動的に生成されるため，クライアント側では代理オブジェクトとリモートオブジェクトの対応を考えることなく，分散オブジェクトを使わない場合と同様のプログラミングが可能なことが大きな特徴である（図 1.3）．接続モデルは，もっとも多くの分散オブジェクト技術でサポートされている方法であるが，新たなリモートオブジェクトを生成するには，それを行うメソッドをサーバ側に用意して，これを明示的に呼び出す必要がある（図 1.4）．

本章では Java に標準で用意されている RMI を用いて分散オブジェクトプログラミングを体験してみる．Java RMI は Java 環境で動作させるために特別に設計された通信メカニズムである．これは，敢えて他の言語との相互運用性を犠牲にしながら，Java の特徴を生かすように設計された分散オブジェクトプログラム用のシステムである．

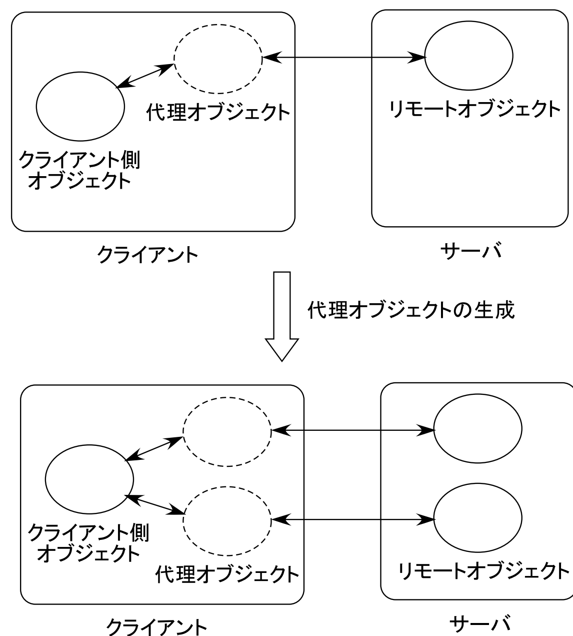

図 1.3: 生成モデル：代理オブジェクトが生成されると，対応するリモートオブジェクトも生成される．

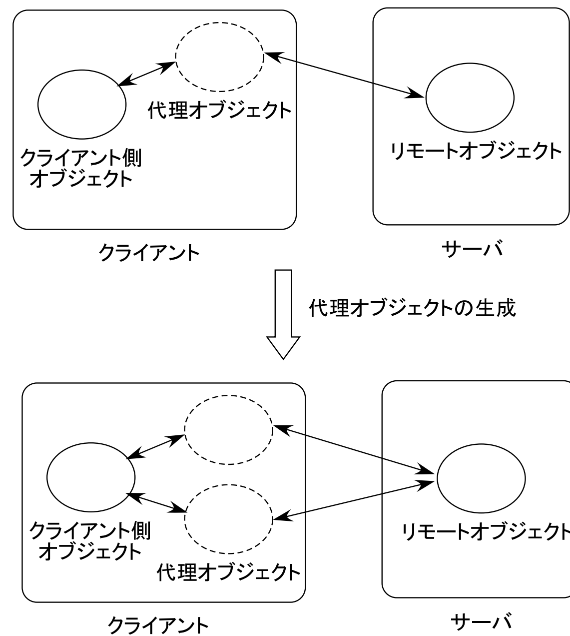

図 1.4: 接続モデル : 代理オブジェクトはすでに生成されているリモートオブジェクトに接続する.

1.3　分散オブジェクトの実現

1.3.1　インターフェイス

実際に分散オブジェクトはどのように実現されているのだろうか．少し詳細を見てみよう．

まず，代理オブジェクトにおいて実現されているメソッド，すなわち，リモートオブジェクトに実装されているうち，リモートメッセージで呼び出し可能なメソッドを，クライアントとサーバで共有するために**インターフェイス**の定義を行う．

インターフェイスは Java RMI や HORB においては Java の `interface` を用いて定義される．CORBA では IDL（Interface Definition Language）という専用の言語を用いてインターフェイスを定義する．

インターフェイス定義を専用のコンパイラを用いてコンパイルすることにより，クライアント側で用いられる**スタブ**と呼ばれるコードと，サーバ側で用いられる**スケルトン**と呼ばれるコードを得ることができる（図 1.5）．

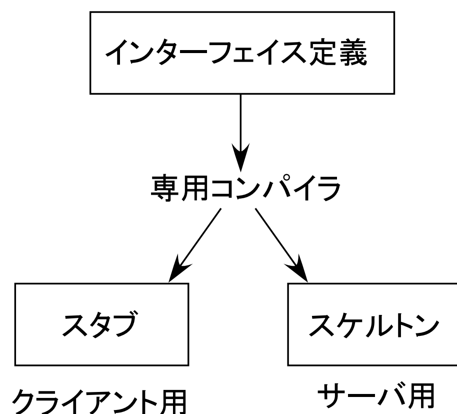

図 1.5: インターフェイスからスタブとスケルトンという二種類のコードを生成する．

1.3.2 スタブとスケルトン

スタブコードはクライアント側で代理オブジェクトを実現する部分である．ここにはクライアントから送られたメッセージをネットワークで送信可能な形に変換（マーシャリングと言う）しリモートオブジェクトに送信すると同時に，リモートオブジェクトから送信されたメッセージを解読（アンマーシャリングと言う）してクライアントに渡すという役割を果たす．

スケルトンはサーバ側で動作するオブジェクトとネットワークの間に入ってリモートオブジェクトを実現する部分であり，役割としてはクライアント側で言うところのスタブコードに似ている．

この様子を図 1.6 に示す．

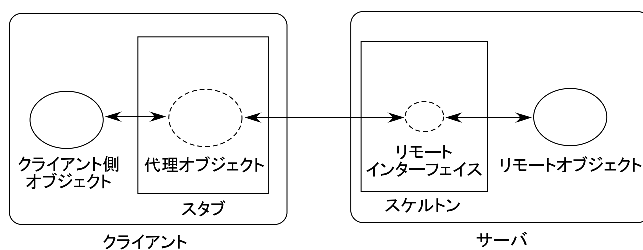

図 1.6: インターフェイスからスタブとスケルトンという二種類のコードを生成する．

1.3.3 オブジェクトリファレンス

前節までの説明で，代理オブジェクトと，これに対応するリモートオブジェクトがいったん接続されれば，分散オブジェクトプログラムがうまく動作するであろうことは納得して貰えただろうか．次に問題となるのは，代理オブジェクトが対応するリモートオブジェクトとどのように接続す

るのかということである．

これを解決するのが**オブジェクトリファレンス**である．オブジェクトリファレンスは，分散オブジェクト環境でオブジェクトを一意に特定するための情報である．実際にどのような情報を用いるかは分散オブジェクトの技術に依存しているが，一般的には 1) サーバに到達するためのネットワークプロトコルやアドレス情報と 2) サーバ内でオブジェクトを特定するための情報が含まれている．

オブジェクトリファレンスは，クライアント側で代理オブジェクトに対応するリモートオブジェクトを見つけるために利用されるが，このためにはサーバ側でリモートオブジェクトのオブジェクトリファレンスを登録する必要がある．このオブジェクトリファレンスの登録や参照を受け付けるためのサービスも分散オブジェクトの技術によって異なる．

Java RMI では JNDI（Java Naming and Directory Interface），もしくはその簡易版である RMI レジストリを用いてオブジェクトリファレンスのやりとりを行う．HORB では HORB の専用のサーバがその役割を司る．CORBA では様々な方法を用いてオブジェクトリファレンスのやりとりが可能で，プログラマが自分で使いやすい物を選ぶことができる．

いくつかの分散オブジェクト環境では，オブジェクトリファレンスを名前（＝文字列）で登録および参照する．この方式は，分かりやすい名前を用いてリモートオブジェクト（のオブジェクトリファレンス）をやりとりできるという利点がある．これは**ネーミング・サービス**と呼ばれる方式で，JNDI の名前を見て分かる通り Java RMI でも名前を用いてオブジェクトリファレンスのやりとりを行う．

第2章　RMIを用いた分散オブジェクト

2.1　インターフェイス

では実際にRMIを用いた分散オブジェクトプログラムを作成してみよう．RMIでは分散オブジェクトにおけるインターフェイスはinterfaceとして宣言する．これはクライアント側のオブジェクトが代理オブジェクトを通してリモートオブジェクトに依頼することができるメソッドを定義する．

Java RMIでは，インターフェイスはRemoteインターフェイスの拡張として宣言し，また全てのメソッドはRemoteException例外の送出を宣言しなくてはならない．これらはパッケージjava.rmiで宣言されているので，このパッケージ内のクラスをimportするのも忘れないでおこう．

以下にリモートインターフェイスの例を示す．これは文字列を送信するメソッドsay()を宣言している．

```
1: import java.rmi.*;
2:
3: public interface SayHello extends Remote {
4:     public void say(String str) throws RemoteException;
5: }
```

2.2　リモートオブジェクト

リモートオブジェクトは，サーバ側で動作するオブジェクトであり，代理オブジェクトを通して依頼されたメッセージを実際に処理するオブジェクトである．すなわち，処理すべきインターフェイスを実装するオブジェクトである．

また，リモートオブジェクトはUnicastRemoteObjectを拡張する必要がある．これにより，ネットワークを介してメッセージを受け取るための機構が暗黙のうちに組み込まれる．

以下にリモートオブジェクトの例を示す．これにより，文字列を送信するメソッドsay()は，受け取った文字列を画面に表示するように実現される．

7〜9行目のコンストラクタにも注意して欲しい．リモートオブジェクトの初期化に失敗した場合には，代理オブジェクトを経由してクライアント側にも例外を発生するため，コンストラクタでRemoteException例外を発生する必要がある．このため内部変数などの初期化がない場合にもコンストラクタは必要である．

```
 1: import java.rmi.*;
 2: import java.rmi.server.UnicastRemoteObject;
 3:
 4: public class SayHelloRemoteObject
 5:     extends UnicastRemoteObject implements SayHello
 6: {
 7:     public SayHelloRemoteObject() throws RemoteException {
 8:         super();
 9:     }
10:
11:     public void say(String hello) throws RemoteException {
12:         System.out.println(hello);
13:     }
14: }
```

2.3　スタブとスケルトン

Java RMI では，スタブとスケルトンはリモートインターフェイスを実装したオブジェクト，すなわちリモートオブジェクトを `rmic` という専用のコンパイラでコンパイルすることにより生成することができる．

先の `SayHelloRemoteObject` を例に取ると，具体的には以下のような手順でスタブとスケルトンを生成できる．

```
% javac SayHelloRemoteObject.java [Enter]
% rmic SayHelloRemoteObject [Enter]
```

2.4　オブジェクトリファレンス

2.4.1　ネーミング・サービスの実行

サーバ側でリモートオブジェクトを登録し，クライアント側でこれを取り出すには，ネーミング・サービスが動作している必要がある．サーバ側でネーミング・サービスが動作していない場合には自分でネーミング・サービスを実行する必要がある．ここでは簡易型のネーミング・サービスである RMI レジストリを用いた手法について説明しよう．

RMI レジストリは `rmiregistry` という名前のプログラムとして提供されており，これを実行することにより簡易型のネーミング・サービスを提供することができる．引数としてポートを指定することができる．ポート番号を省略した場合のデフォルトのポート番号は 1099 である．以下に

ポート番号 1100 で RMI レジストリを実行する場合の例を示す.

文末の「&」はプログラムをバックグラウンドで動作させることを意味するシェルコマンドで，これにより実行されたプログラムは背後で動作する（Windows 用語で言えば「常駐する」）.

```
% rmiregistry 1100 & Enter
```

2.4.2 オブジェクトリファレンスの登録

サーバ側のプログラムでリモートオブジェクトをネーミング･サービスに登録するには，java.rmi.Naming クラスの bind() または rebind() メソッドを用いる.

この二つのメソッドは，bind() ではすでに同じオブジェクトリファレンスで登録されたリモートオブジェクトがあった場合はエラーになるが，rebind() の場合はエラーにならずに新しいものに置き換えるという違いがある.

bind() や rebind() の第一引数には，オブジェクトの場所を URL 形式で以下のように指定する.

```
//host:port/name
```

ここで host はリモートオブジェクトが動作している計算機の名前であり，これを省略するとローカルホストが用いられる．port はネーミング・サービスがサービスしているポート番号で，2.4.1 章で RMI レジストリを実行する際に指定したポート番号などを指定する．これを省略した場合にはデフォルトのポート番号である 1099 が利用される．name はリモートオブジェクトに対応した任意の文字列で，プログラマが自由に指定することができる.

bind() や rebind() の第二引数には，登録するリモートオブジェクトを指定する.

例えばあるリモートオブジェクト obj を csux05 という名前の計算機のポート番号 1100 に，HelloObj という名前で登録するには，以下のように記述する.

```
Naming.rebind("//csux05:1100/HelloObj", obj);
```

2.4.3 オブジェクトリファレンスの参照

クライアント側のプログラムで，オブジェクトリファレンスを参照し，リモートオブジェクトに対応する代理オブジェクトを生成するには，java.rmi.Naming クラスの lookup() メソッドを用いる.

lookup メソッドの引数は一つで，登録の際に用いた URL 形式の名前を指定する．返り値は，名前で指定されたリモートオブジェクトに対応する代理オブジェクトそのものであり，これを**リモートインターフェイスの型にキャストして**用いる.

例えば csux05 という名前の計算機のポート番号 1100 に登録された HelloOb というリモートオブジェクトの代理オブジェクトを生成するには以下のように記述する.

```
SayHello obj = (SayHello)Naming.lookup("//csux05:1100/HelloObj");
```

2.5 オブジェクトの転送

ここで出てきた例では say() というメソッドを用いて String クラスのインスタンスをクライアント側からサーバ側に送信している．RMI では，このようなオブジェクトの送受信に Java の直列化（Serialization）の機構を利用している．

従って，独自に作られたクラスのインスタンスであっても，直列化可能であれば分散オブジェクト環境において，リモートメソッドの引数に用いることができる．逆に言えば，直列化できないクラスのインスタンスはリモートメソッドの引数に用いることはできない．

直列化についての詳細はここでは紹介しないが，多くの場合はインターフェイス Serializable を implements するだけで直列化可能になる．

2.6 セキュリティ

2.6.1 セキュリティマネージャ

Java RMI は，ネットワークを介してオブジェクトのやりとりを行うため，セキュリティマネージャの作成・インストールが必要である．セキュリティマネージャはプログラムがセキュリティ上問題のあるような動作を行う際に，それが実際に問題がないかを確認し，セキュリティ上問題がある場合には例外を発生する．

RMI では標準的なセキュリティマネージャとして RMISecurityManager が用意されているのでこれを用いることができる．RMISecurityManager を用いるには，プログラムの main() メソッドにおいて以下のように記述する．

```
System.setSecurityManager(new RMISecurityManager());
```

これを行うと，プログラムは標準のセキュリティポリシーに従ってネットワークの接続を行うことになる．標準のセキュリティポリシーは Java のインストールされたディレクトリ下の java.policy ファイルによって指定されており，ネットワークの接続に関しては

```
permission java.net.SocketPermission "localhost:1024-", "listen";
```

となっているが，これは「ローカルホストのポート番号 1024 以上へのソケット接続は listen（待機）のみ許可される」ということで，ほとんどの接続については許可されない．このため，RMI を用いる場合にはセキュリティポリシーを自分で設定する必要がある．

2.6.2 セキュリティポリシーの設定

セキュリティポリシーを設定するには，独自のセキュリティポリシーを書いたファイルを用意する．ネットワーク接続に関するセキュリティポリシーの書き方は以下の通りである．

```
grant codebase "パス" {
    permission java.net.SocketPermission
            "ホスト名もしくはIPアドレス:ポート番号", "アクション";
};
（一行で書いても良い）
```

「"パス"」にはURL形式で，このポリシーを適用するプログラムの所在を示す．ここに用いることのできるプロトコルは file もしくは http であり，例えば /home/John/java 以下のプログラムに許可を与えたければ,「codebase "file:///home/John/java/"」と書く．この時，パスの最後の "/" は省略できないことに注意すること.「codebase "パス"」の部分（下線部）を省略した場合には，全てのプログラムにこのポリシーを適用することになる．

「ホスト名もしくはIPアドレス」には，プログラムから接続したい計算機をホスト名かIPアドレスで指定し,「ポート番号」には接続したいポート番号を指定する.「ポート番号」には範囲指定も可能で「1024-」と書けばポート番号1024以降の全てを表し,「1024-2048」と書けばポート番号1024から2048までを表す.「:ポート番号」の部分を省略した場合は当該ホストの任意のポート番号への接続に対してこのポリシーが適用される．

「"アクション"」には許可したいアクションをコンマ（,）で区切って列挙する．例えば connect と resolve を許可したければ「"connect, resolve"」と書く．

実行時には，こうしてできたセキュリティポリシーを書いたファイルを，プログラムの実行時に -Djava.security.policy オプションで指定する．例えば ../security/policy というファイルを指定して Server というプログラムを実行するときは以下のように入力する．

```
% java -Djava.security.policy=../security/policy Server Enter
```

2.6.3 RMIで必要なセキュリティポリシー

RMIでは，サーバ側のプログラムでは accept, connect, resolve の3つのアクションに許可が，クライアント側のプログラムでは connect, resolve の2つのアクションに許可が必要である．よって，例えばサーバ側のプログラムでは，以下のように書いたファイルを実行時に指定すれば良い．

```
grant {
    permission java.net.SocketPermission
            "localhost", "accept,connect,resolve";
};
```

もし適切なポリシーが指定されていない場合には以下のような例外が発生する．

```
% java Server Enter
java.security.AccessControlException: access denied
    (java.net.SocketPermission 127.0.0.1:4675 accept,resolve)
at java.security.AccessControlContext.checkPermission(AccessContro
lContext.java:270)
at java.security.AccessController.checkPermission(AccessController
.java:401)
at java.lang.SecurityManager.checkPermission(SecurityManager.java:
542)
at java.lang.SecurityManager.checkConnect(SecurityManager.java:104
4)
at java.net.Socket.connect(Socket.java:419)
at java.net.Socket.connect(Socket.java:375)
at java.net.Socket.<init>(Socket.java:290)
（以下略）
```

この時，下線部を見ると，このプログラムの実行に必要なポリシーを知ることができる．この例では「IP アドレス `127.0.0.1` の計算機のポート番号 `4675` に `accept` と `resolve` の許可」が必要なことが分かる．この場合にはポリシーファイルに以下の行を追加してやれば良い．ちなみに `localhost` は IP アドレス `127.0.0.1` に対応する特殊な計算機名である．

```
grant {
    permission java.net.SocketPermission
                "localhost:4675", "accept,resolve";
};
```

第3章　RMIを用いた分散オブジェクトプログラムの作成と実行

3.1　簡単なプログラムの作成

では最後に，簡単なサンプルプログラムを実際に作ってみよう．このプログラムは，クライアントから送られた文字列をサーバ側のプログラムが画面に表示するという非常にシンプルなものである．また，まずは全体を理解するためにセキュリティマネージャは利用しないでプログラムを作成してみよう．

1. インターフェイス

クライアントが呼び出すリモートオブジェクトのメソッドを宣言する．以下のプログラムを SayHello.java という名前で保存しよう．

```
1: import java.rmi.*;
2:
3: public interface SayHello extends Remote {
4:     public void say(String str) throws RemoteException;
5: }
```

2. リモートオブジェクト

リモートインターフェイスを実装したリモートオブジェクトを作成する．もちろん，このオブジェクトはクライアントから呼び出されないメソッドを持っていても構わない．ここでは start() メソッドはリモートインターフェイスではない．これを SayHelloRemoteObject.java という名前で保存しよう．

```
 1: import java.rmi.*;
 2: import java.rmi.server.UnicastRemoteObject;
 3:
 4: public class SayHelloRemoteObject
 5:     extends UnicastRemoteObject implements SayHello
 6: {
 7:     public SayHelloRemoteObject() throws RemoteException {
 8:         super();
 9:     }
10:
11:     public void say(String hello) throws RemoteException {
12:         System.out.println(hello);
13:     }
14:
15:     public void start() {
16:         System.out.println("Start!");
17:     }
18: }
```

3. スタブとスケルトンの生成

リモートオブジェクトをコンパイルし，できた class ファイルからスタブとスケルトンを生成する．

```
% javac SayHelloRemoteObject.java [Enter]
% rmic SayHelloRemoteObject [Enter]
```

これにより

SayHelloRemoteObject_Stub.class（スタブ）
SayHelloRemoteObject_Skel.class（スケルトン）

という2つのファイルが生成される．

4. サーバ側プログラムの作成

続いてサーバ側のプログラムを作成しよう．7行目でリモートオブジェクトとなるオブジェクトを生成し，これを9行目で`//localhost/HelloObj`という名前で登録している．また，`SayHelloRemoteObject`のコンストラクタや`Naming`クラスの`rebind()`メソッドは例外を発生するため，全体を`try`〜`catch`で囲ってある．これを`SayHelloServer.java`という名前で保存し，コンパイルしよう．

```
 1: import java.rmi.*;
 2:
 3: public class SayHelloServer
 4: {
 5:     public static void main(String[] args) {
 6:         try {
 7:             SayHelloRemoteObject obj = new SayHelloRemoteObject();
 8:             obj.start();
 9:             Naming.rebind("//localhost/HelloObj",obj);
10:         } catch(Exception e) {
11:             e.printStackTrace();
12:             System.exit(1);
13:         }
14:     }
15: }
```

5. クライアント側プログラムの作成

7〜8行目で//localhost/HelloObjという名前の代理オブジェクトを取得し，これをSayHelloクラスの変数に代入している．

Namingクラスのlookup()メソッドの返り値はRemoteだが，リモートオブジェクトであるSayHelloRemoteObjectはRemoteの直接のサブクラスではないので，これにはキャストできない．そのため，リモートインターフェイスであるSayHelloにキャストしている点に注意しよう．

このプログラムをSayHelloClient.javaという名前で保存し，コンパイルしよう．

```
 1: import java.rmi.*;
 2:
 3: public class SayHelloClient
 4: {
 5:     public static void main(String[] args) {
 6:         try {
 7:             SayHello obj =
 8:                 (SayHello)Naming.lookup("//localhost/HelloObj");
 9:             obj.say("Hello World!");
10:         } catch( Exception e ) {
11:             e.printStackTrace();
12:             System.exit(1);
13:         }
14:     }
15: }
```

3.2 簡単なプログラムの実行

このプログラムの場合にはセキュリティマネージャを使っていない関係上，プログラムの実行はそれほど難しいものではない．以下の手順に従ってプログラムを実行してみよう．

なお，このサンプルプログラムでは`//localhost/HelloObj` という名前でリモートオブジェクトを登録していることで分かるように，サーバ側プログラムとクライアント側プログラムは同一の計算機上で動作させることを前提にしている．もちろん名前を変えれば異なる計算機上で動作させることもできるので試してみると良いだろう．

1. [サーバ側] ネーミング・サービスの実行

まず，サーバ側でネーミング・サービスを実行しておこう．RMI では，このために RMI レジストリ（`rmiregistry`）というプログラムを実行する．バックグラウンドで実行するために後ろに「`&`」を付けて実行する点に注意しよう．

```
% rmiregistry & Enter
```

なお，環境変数 `CLASSPATH` で示されたパスにスタブがない場合には，次にサーバを実行する際にスタブの場所を明示的に指定する必要がある．

2. [サーバ側] プログラムの実行

サーバ側プログラムの実行に必要なのは以下のファイルである．スタブも必要な点に注意しよう．

```
SayHello.class
SayHelloRemoteObject.class
SayHelloServer.class
SayHelloRemoteObject_Skel.class
SayHelloRemoteObject_Stub.class
```

これらのファイルを単一のディレクトリにまとめ，以下のようにサーバ側のプログラムを実行する．うまく実行できれば，画面には「`Start!`」と表示され，クライアントの接続待ちの状態になる．

```
% java SayHelloServer Enter
Start!
（ここでクライアントの接続待ち状態）
```

なお，ネーミング・サービス（ここでは rmiregistry）からスタブが見えない場合には，オブジェクトリファレンスの登録の際に以下のような例外が発生する．

```
% java SayHelloServer [Enter]
Start!
java.rmi.ServerException: RemoteException occurred in server thread;
nested exception is:
    java.rmi.UnmarshalException: error unmarshalling arguments; nes
ted exception is:
    java.lang.ClassNotFoundException: SayHelloRemoteObject_Stub
（以下省略）
```

この場合には-Djava.rmi.server.codebase オプションにより，スタブの場所を明示的に指定する必要がある．スタブの場所は URL 形式で指定する．

例えば，/home/murao/java/SayHello にスタブがあるならば以下のようにサーバ側のプログラムを実行する．なお，codebase を指定する際にはパスの最後の"/" は省略できないことに注意すること．

```
% java SayHelloServer
  -Djava.rmi.server.codebase="file:///home/murao/java/SayHello/"
（実際には一行）
```

3. [クライアント側] プログラムの実行

クライアント側プログラムの実行に必要なのは以下のファイルである．

```
SayHello.class
SayHelloClient.class
SayHelloRemoteObject_Stub.class
```

これらのファイルを単一のディレクトリにまとめ，以下のようにクライアント側のプログラムを実行する．うまく実行できれば，サーバ側の画面にクライアントから送信された文字列「Hello World!」が表示され，クライアント側のプログラムはすぐに終了する．

```
% java SayHelloClient [Enter]
% （すぐ終了）
```

4. [サーバ側] ネーミング・サービスの終了

プログラムの実行が終わったら，バックグラウンドで動作しているネーミング・サービスを終了させておこう．これには ps コマンドで rmiregistry のプロセス ID（PID）を調べ，kill コマンドで当該プロセスを停止させる．

3.3　セキュリティマネージャを用いたプログラムの作成と実行

次にセキュリティマネージャを用いたプログラムを作成する．と言っても，サーバ側プログラムとクライアント側プログラムに一行追加するだけである．

セキュリティマネージャのインストールに伴い，セキュリティポリシーの設定が必要になる．ここでは，サーバ側，クライアント側の各プログラムと同じディレクトリに RMI の実行に必要なポリシーだけを書いたファイルを置き，実行時にこれを指定することでセキュリティポリシーを設定する．

1. サーバ側

1.1 プログラム

下線部の 1 行を追加する．

```
 1: import java.rmi.*;
 2:
 3: public class SayHelloServer
 4: {
 5:     public static void main(String[] args) {
 6:         System.setSecurityManager(new RMISecurityManager());
 7:         try {
 8:             SayHelloRemoteObject obj = new SayHelloRemoteObject();
 9:             obj.start();
10:             Naming.rebind("//localhost/HelloObj",obj);
11:         } catch(Exception e) {
12:             e.printStackTrace();
13:             System.exit(1);
14:         }
15:     }
16: }
```

1.2 セキュリティポリシー

以下のように記述したファイルを server.policy という名前でサーバ側のプログラムと同じディレクトリに保存しよう．

```
grant {
    permission java.net.SocketPermission
        "localhost:1024-", "accept,connect,resolve";
};
```

ただし，この例では codebase を指定していない．分かるようであれば，サーバ側プログラムの場所（パス）を codebase に設定しよう．サーバ側プログラムの場所は，以下のようにそのプログラムがある場所で pwd コマンドを実行することで知ることができる．

```
% ls SayHelloServer.class Enter
SayHelloServer.class （←カレントディレクトリの確認）
% pwd Enter
/home/Teaching/Practice5/02_RMI/prog/server （←パス）
%
```

この場合には server.policy を以下のように記述する．

```
grant codebase "file:///home/Teaching/Practice5/02_RMI/prog/server/" {
    permission java.net.SocketPermission
        "localhost:1024-", "accept,connect,resolve";
};
```

2. クライアント側

2.1 プログラム

サーバ側プログラムと同様に，下線部の 1 行を追加する．

```
 1: import java.rmi.*;
 2:
 3: public class SayHelloClient
 4: {
 5:     public static void main(String[] args) {
 6:         System.setSecurityManager(new RMISecurityManager());
 7:         try {
 8:             SayHello obj =
 9:                 (SayHello)Naming.lookup("//localhost/HelloObj");
10:             obj.say("How are you?");
11:         } catch( Exception e ) {
12:             e.printStackTrace();
13:             System.exit(1);
14:         }
15:     }
16: }
```

2.2 セキュリティポリシー

以下のように記述したファイルを client.policy という名前でサーバ側のプログラムと同じディレクトリに保存しよう.

```
grant {
    permission java.net.SocketPermission
        "localhost:1024-", "connect,resolve";
};
```

3. [サーバ側] ネーミング・サービスの実行

プログラムの実行に先立ち，3.2 章の手順に従ってネーミング・サービスを起動しておこう.

4. [サーバ側] プログラムの実行

サーバ側プログラムの実行には以下のファイルが必要である.

```
SayHello.class
SayHelloRemoteObject.class
SayHelloServer.class
SayHelloRemoteObject_Skel.class
SayHelloRemoteObject_Stub.class
server.policy
```

これらのファイルを単一のディレクトリにまとめ，以下のようにサーバ側のプログラムを実行する．うまく実行できれば，画面には「Start!」と表示され，クライアントの接続待ちの状態になる．

```
% java -Djava.security.policy=./server.policy SayHelloServer [Enter]
Start!
（ここでクライアントの接続待ち状態）
```

5. [クライアント側] プログラムの実行

クライアント側プログラムの実行に必要なのは以下のファイルである．

```
SayHello.class
SayHelloClient.class
SayHelloRemoteObject_Stub.class
client.policy
```

これらのファイルを単一のディレクトリにまとめ，以下のようにクライアント側のプログラムを実行する．うまく実行できれば，サーバ側の画面にクライアントから送信された文字列「Hello World!」が表示され，クライアント側のプログラムはすぐに終了する．

```
% java -Djava.security.policy=./client.policy SayHelloClient [Enter]
% （すぐ終了）
```

6. [サーバ側] ネーミング・サービスの終了

プログラムの実行が終わったら，3.2 章の手順に従って，バックグラウンドで動作しているネーミング・サービスを終了させておこう．

3.4 RMI を用いたアプレットの作成と実行

アプレットにおいても，通常のアプリケーションと同様に，RMI を用いた分散オブジェクトを利用することができる．

この場合，アプレットは Web サーバを通してブラウザにダウンロードされると同時に，ネットワークを介して異なる計算機上のリモートオブジェクトと通信しながら動作することになる．また，Web サーバ経由でスタブを受け取ることができるため，クライアント側のプログラムと同一のディレクトリにスタブを置く必要はない．

サーバ側のプログラムは通信相手がアプレットであるか，通常のアプリケーションであるかは気にしなくても良い．実行の方法が異なるだけで，プログラムそのものはこれまでの章で説明して来たものと全く同じである．

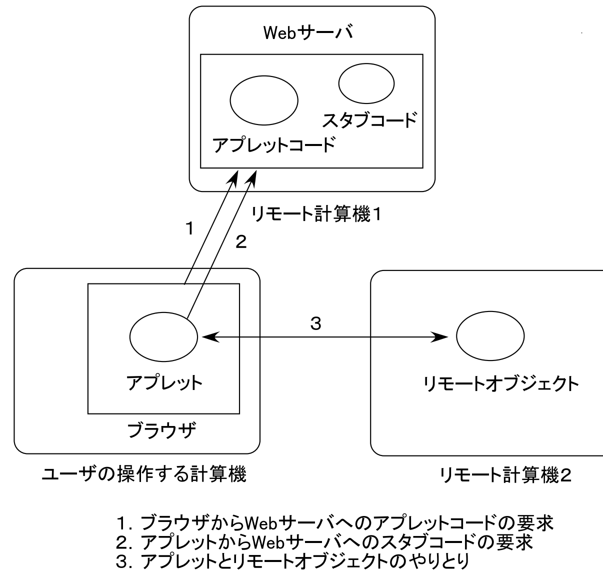

図 3.1: アプレットにおけるRMIを用いた分散オブジェクト

1. サーバ側プログラムの作成

サーバ側のプログラムは 3.3 章のものをそのまま用いる．

2. アプレットの作成

このアプレットは画面の真ん中に「`Send Message`」というボタンを置き，これを押すとリモートオブジェクトの `say()` メソッドを実行するというものである．アプレットの初期化時（`init()` メソッド内）に代理オブジェクトを生成し，ボタンのリスナー内でこの代理オブジェクトのメソッドを実行している．

なお，アプレットでは，ブラウザがセキュリティの管理を行うので，アプレットでセキュリティマネージャをインストールする必要はない．

```
 1: import java.applet.*;
 2: import java.awt.*;
 3: import java.awt.event.*;
 4:
 5: import java.rmi.*;
 6:
 7: public class SayHelloApplet extends Applet {
 8:     SayHello obj;
 9:
10:     class ButtonListener implements ActionListener {
11:         public void actionPerformed(ActionEvent ev) {
12:             try {
13:                 obj.say("Hello World!");
14:             } catch( Exception e ) {
15:                 stop();
16:             }
17:         }
18:     }
19:
20:     public void init() {
21:         Dimension d = getSize();
22:         Button button = new Button("Send Message");
23:
24:         button.setBounds((d.width-100)/2,(d.height-30)/2,100,30);
25:         setLayout(null);
26:         add(button);
27:
28:         try {
29:             obj = (SayHello)Naming.lookup("//localhost/HelloObj");
30:         } catch( Exception e ) {
31:             stop();
32:         }
33:     }
34: }
```

3. アプレットとスタブの Web サーバ上での配置

コンパイルしたアプレットは対応する HTML ファイルと同じディレクトリに，スタブも Web サーバ経由で取得できる任意のディレクトリに置く．もちろん，アプレットと同じディレクトリにスタブを置いても構わないが，スタブの場所はネーミング・サービスを通じてクライアント側のプログラム（ここではアプレット）に伝えられるため，アプレットと同じディレクトリである必要はない．

例えば /home/murao/public_html/ 以下が http://www.hoge.com/~murao/ という URL でアクセスできるとしよう．この時，図 3.2 のようにファイルを置くと，アプレットは

http://www.hoge.com/~murao/java/SayHello/index.html

で実行できる．スタブは

```
http://www.hoge.com/~murao/java/stubs/SayHelloRemoteObject_Stub.class
```

で取得可能である．

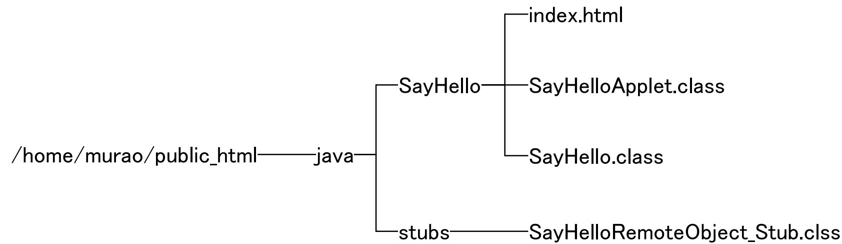

図 3.2: アプレットとスタブの Web サーバ上での配置

4. [サーバ側] ネーミング・サービスの実行

プログラムの実行に先立ち，3.2 章の手順に従ってネーミング・サービスを起動しておく．

5. [サーバ側] プログラムの実行

サーバ側プログラムを実行する際に，スタブの場所を`-Djava.rmi.server.codebase` オプションで指定する．この情報はネーミング・サービスに伝達され，これにより，クライアント側のプログラムは必要に応じてスタブを Web サーバから取得することができる．

今回の例では以下のように実行する．`codebase` オプションの最後は必ずスラッシュ（/）で終わることに注意しよう．

```
% java
    -Djava.rmi.server.codebase="http://www.hoge.com/~murao/java/stubs/"
    -Djava.security.policy=server.policy SayHelloServer
 （実際には一行）
```

6. アプレットの実行

クライアント側のプログラム，すなわちアプレットを実行するには，いつものように，Web ブラウザでアプレット起動用の HTML ファイルにアクセスすれば良い．この例では

```
http://www.hoge.com/~murao/java/SayHello/index.html
```

にアクセスすれば良い．