情報システム技術論'08 — ＃1

久野　靖*

2008.4.10

はじめに

こんにちは、久野です。今週から「情報システム技術論」を開始します。この科目は今年度は「現在の情報システムにおいてどのような技術が使われているのか」について、分散アプリケーション、とりわけ Web アプリケーションを中心に取り上げて行きます。とりあえず、最初の 3 回 (4 月ぶん) は次のように予定します。

- 分散アプリケーションと Java 言語における具体例
- Web アプリケーションとその基本技術
- ページ内スクリプトとデータベースの連携

残り 2 回ぶんの内容についてはそこまでの様子を見た上で決定します。

1　分散アプリケーション

1.1　なぜ分散アプリケーションか?

□ 分散システム→ネットワークで相互に接続された要素どうしが有機的に連携して動作するようなシステム。

- 例:　電子メール、DNS、WWW、…

□ 分散アプリケーション→アプリケーション (特定の用途のためのプログラム) が上述のような形で動作。

- 例:　グループウェア、オンライン会議、ファイル交換、…

□ 「銀行の預金管理システム」「JR の座席予約システム」などは?

- →オンラインシステムとは呼ばれるが分散とはあまり言わないかも…
- 理由:　「センター」がほとんどの業務を行い、「端末」は入力と出力
- 昔は末端の機器の処理能力が低かったことからこれが通常

□ なぜ今日、分散アプリケーションなのか?

- PC の普及→非常に多数の利用者→「センターが全部こなす」ではあまり大したことができない、アピールしない
- 末端の (利用者が接する) システムの能力が大きくなり仕事を分担可能
- 端末側とセンター側の分担/有機的な協力→魅力的な機能やインタフェースを実現可能に
- 例:　Google Maps、Yahoo!　Maps

□ 一方で難しさも

- 分散アプリケーションは「センター集中」より作りづらい (面倒、複雑、大変)
- そのため、ある程度のフレームワーク/サポートをできあいで用意することが行われている
- そのようなフレームワークもそれぞれ構造がややこしくマスターするのが面倒という面も。

1.2　分散アプリケーションの構造

□ 既存の分散システムの上に乗る形での構築

□ 独自プロトコル、独自技術によるもの (大変) ←今回

□ ある程度までフレームワークにたよる←今回

□ メールの上で動くもの…

- メーリングリスト、メールマガジン:　ユーザインタフェースはメールソフトや携帯端末 (インストール不要)
- それ以外にも、速度を要求しないなら、メールを通信のためのインフラとして使うような分散アプリケーションは十分あり得る

□ World Wide Web の上で動くもの (Web アプリケーション)

*筑波大学大学院経営システム科学専攻

- 今日では非常にメジャー
- ユーザインタフェースはブラウザ画面
- インストール不要
- サーバ側/クライアント側連携
- 一方で構築上の難しさ/セキュリティ上の難しさ (その構築技術は次回以降とりあげる)

□ 今回は独自プロトコルやフレームワークについて見てみる

□ Java 言語によるサンプルを題材に考えて行く

- Java 言語: ネットワークまわりの標準 API が充実→題材として適する
- とりあえず Java 言語は初めてでも大丈夫なように説明します

2 Java での基本的なネット接続

2.1 Java 言語の復習ないし入門

□ Java 言語の歴史と概観

- 1991 年に Sun Microsystems 社内で開発
- 基本的な設計→「複雑な/危ない機能を削除した C++ 言語」
- 1995 年 HotJava ブラウザ (アプレット) →一躍ブームに
- 「ブラウザ上で動く」機能は Flash などが主流に
- Java は「普通のソフトウェア開発用言語」としての利用がメインに

□ Java 言語の構造と機能を非常に簡単に説明すると…

- プログラムはすべて class
- 強い型の言語。型=class 型 + 基本型 (int, float, ...)
- 「static void main(String args[])」のメソッドから実行開始
- 「new クラス (…)」でオブジェクトを新規生成
- 「オブジェクト . メソッド (…)」でオブジェクトに付随するメソッド起動

□ Sample11.java:簡単な GUI プログラム

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Sample11 extends JFrame {
  JTextField f1 = new JTextField();
  JButton b1 = new JButton("Exec");
  public Sample11() {
    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container c = getContentPane(); c.setLayout(null);
    c.add(f1); f1.setBounds(40, 40, 180, 40);
    c.add(b1); b1.setBounds(240, 40, 80, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        f1.setText(f1.getText() + "*");
      }
    });
  }
  public static void main(String[] args) {
    new Sample11().setVisible(true);
  }
}
```

□ 解説:

- import は必要なクラス群を短く指定する機能
- クラス Sample11 は JFrame(ウィンドウ) の機能/データ構造を継承
- 画面内に入力欄とボタンを置く→これらのオブジェクトを用意
- 最後にある main() ではこの窓を生成して直ちに表示させるだけ
- 初期設定はクラス名と同じ名前のメソッド (コンストラクタ) で指定
- 窓のサイズは 400x400
- 窓を閉じるとこのプログラムも終了する
- 窓の中の部品を貼る場所を取り出し、自動配置機能をオフに
- 入力欄を貼って位置指定
- ボタンを貼って位置指定
- ボタンに動作をつける…「入力欄の文字列を取り出し、後ろに*を連結し、再び入力欄にセットする」

□ 演習:このプログラムを打ち込んで動かしなさい。動いたら部品の配置を変えたりボタンの動作を変えてみなさい。

2.2 C/S モデルとその得失

□ C/S(クライアントサーバモデル)

- 今日のネット上で主流の分散アプリの構造
- サーバは常時稼働状態にある

- クライアントは必要が生じたらユーザが動かす
- クライアントからサーバに接続してサービスを受ける
- サービスが終わったら切断する

□ C/S 構成の利点

- サーバ上に共有のデータを置いておける→情報の共有 (=ネットの利点) が行いやすい
- サーバ上の共有データを扱うのはサーバプログラムだけ→整合性の問題が置きにくい
- クライアントは好き勝手に起動/終了してかまわない→やりやすい

□ →今日のネット上のサービスアーキテクチャとして主流

□ C/S 構成の弱点

- サーバがボトルネックになりやすい← (性能、single point of failure)
- スケーラビリティに欠ける

□ 複数のサーバが互いに情報交換しあいながら動作するもののものある (メール、ニュース、IRC、…) →スケーラブル、ただし複雑

2.3 Java による簡単な C/S 通信

□ クライアント側は先のプログラムを多少手直し

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;

public class Sample12 extends JFrame {
  JTextField f1 = new JTextField();
  JButton b1 = new JButton("Exec");
  JLabel l1 = new JLabel("start...");
  public Sample12() {
    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container c = getContentPane(); c.setLayout(null);
    c.add(f1); f1.setBounds(40, 40, 180, 40);
    c.add(b1); b1.setBounds(240, 40, 80, 40);
    c.add(l1); l1.setBounds(20, 360, 360, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          Socket cs = new Socket("localhost", 4192);
          PrintWriter out = new PrintWriter(
            cs.getOutputStream(), true);
          BufferedReader in = new BufferedReader(
            new InputStreamReader(cs.getInputStream()));
          out.println(f1.getText());
          f1.setText(in.readLine());
        } catch(Exception ex) { l1.setText("!" + ex); }
      }
    });
  }
  public static void main(String[] args) {
    new Sample12().setVisible(true);
  }
}
```

□ 解説:

- GUI 部品としてラベル (表示欄) を増やした。
- ボタンの動作全体を try-catch で囲み、例外が発生したらその例外を受け止めて文字列としてラベルに表示するようにした。
- ボタン動作の内容は以下の通り。
- ソケットを localhost(このマシン) の 4192 番ポートに接続。
- そのソケットの書き込み口/読み出し口を用意。
- 書き込み口には入力欄の内容を書く。
- 読み出し口から受け取ったものを入力欄にセット。

□ 一方、サーバ側は GUI はなし。次のように。

```
import java.net.*;
import java.io.*;

public class Sample12Server {
  public static void main(String[] args)
      throws Exception {
    ServerSocket ss = new ServerSocket(4192);
    while(true) {
      Socket cs = ss.accept();
      PrintWriter out = new PrintWriter(
        cs.getOutputStream(), true);
      BufferedReader in = new BufferedReader(
        new InputStreamReader(cs.getInputStream()));
      String line = in.readLine();
      out.println(line + line);
      cs.close();
      if(line.equals("bye")) break;
    }
    ss.close();
  }
}
```

□ 解説:

- ポート 4192 番のサーバソケットを作る。以下ループ。
- クライアントが接続してくるのを待つ。接続してくると通常の通信用ソケットが返されて来る。
- これに対して書き込み口/読み出し口を用意する。
- まず 1 行読み込み、その文字列を 2 回連結したものを書き出す。
- 終わったらソケットは閉じる。

- 渡された 1 行が bye だったらループ終わり。
- 終わる時はサーバソケットも閉じる。

□ 演習:この 2 つのプログラムを打ち込んで動かしなさい。複数のクライアントでも動作することを確認すること。さらに、サーバ内部で次々に受け取った文字列を連結するようにして、複数のクライアント間のデータ共有ができることを確認しなさい。

3 分散システムの通信プロトコル

3.1 C/S 通信のさまざまな側面

□ ステートレスなプロトコル

- 先の例題では「接続→パラメタ送信→結果返送→すぐ切断」の反復。
- 1 つの要求が終わったらすぐ次の要求を処理。相互に関係がない (独立、状態を持たない→ステートレス)。
- サーバダウン時などの影響が少ない。サーバが簡単、効率的。
- Web サーバや NFS サーバはこのような方針。ただし接続を毎回張っているとオーバヘッドがあるので、接続は張ったままで動作可能。

□ ステートフルなプロトコル

- ステートレスの反対。接続している間「自分専用の担当者」に相手してもらう感じ。リモートログインを想像すればよい。
- たとえば:接続→パスワード要求→パスワード応答→複数の処理→最後に切断、など。ステートレスではこれはできない。
- プログラム的には、先の例題で in/out を用意した後、1 往復ではなく何往復もやりとりする。
- その場合、やりとりしている間は accept() が実行されないと次のサービスが受け付けられない→スレッドを生成して、そのスレッドがサービスを行い、本体はすぐ次ぎの accept() に進むのが普通。

□ 演習:先の例題を改造して、クライアントごとに数回に分けて文字列を送り、完了するとそれらがまとめて共有データに連結される、という風に直してみよ。(結構大変だと思います。ボタンも送信ボタンと完了ボタンの 2 つが必要になりますし。)

□ 演習:上記だと、1 つのクライアントが使用中は他のクライアントは接続しようとしても待たされるはず。これを各クライアントが 1 つのスレッドとして実行するように手直しし、待たされなくなることを確認してみよ。(これは上記ができてればそれほど大変ではないと思う。)

□ ステートレス vs ステートフル

- 「どちらがいい」というわけではなく、それぞれの利点と弱点がある。
- SMTP、FTP、TELNET、… →ステートフル
- HTTP、NFS、… →ステートレス
- ステートレスにするには「プロトコル側で状態を持たない」ことをどうやって補うかを考える必要がある。
- NFS であれば「何番のファイルの何バイト目から何バイト読め/書け」という形で要求を統一することでステートレスに。一番普通の「順番に読む」操作も要求時に上の形に直して要求。

□ テキストベースのプロトコル

- SMTP、HTTP などはテキストベースのプロトコル

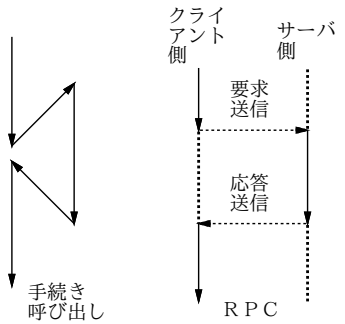
```
% telnet w3in 80
GET /index.html HTTP/1.1
Host: w3in
Connection: close

(ページ返送…)
%
```

- テストしてみようと思った時に手で打ち込んだりチェックできるという利点がある。
- その反面、コマンドや付加情報を生成したり受け側で解釈したりといった手間が必要。
- その反対→バイナリデータ表現を用いるもの。

3.2 RPC

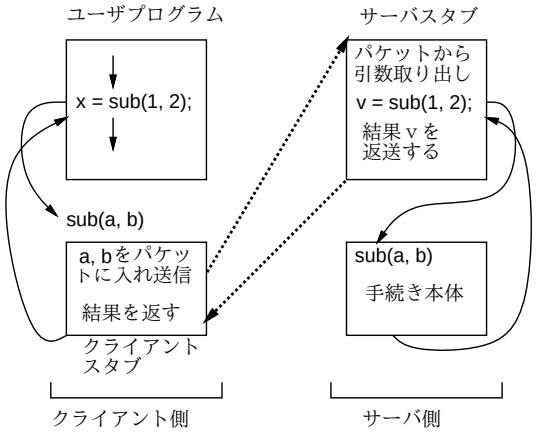
□ RPC(Remote Procedure Call)

- 通信路を張って往復しつつ通信→プロトコルが手作り、繁雑
- もっと多くのものを統一的に扱えプログラミングしやすい方法は?
- ネットワーク経由の往復は手続き呼び出し時の制御の往復に似ている→プログラム側からは手続き呼び出しに見えるようにしたらいい→ RPC のアイデア



□ RPC の構成…

- クライアント側…クライアントスタブ (普通の手続きのように見えて呼び出せるが、裏側ではパラメタをパケットに入れてサーバに送り、返されて来たパケットから結果を取り出して返送する。
- サーバ側…サーバスタブ (RPC システムからパケットを渡され、それを普通のパラメタの形に直して本来の手続きを呼び、戻って来たら結果をパケットに入れて返送する。



□ RPC ジェネレータ

- 上記のスタブ類を用意するのはすごく面倒に思えるが…
- 実際には「名前」「パラメタ個数とそれぞれの型」「返値の型」を指定してやれば、自動的に両側のスタブを生成してくれるジェネレータ (RPC ジェネレータ、スタブジェネレータ) が用意されている。
- パラメタや返値が複雑な型のときも、それを詰め込み (marshal)、取り出し (unmarshal) を行うコードが用意される。

□ RPC のバインディング

- このように RPC では一見手続き呼び出しのようにサーバ上のコードが呼び出せるが、実際には「ネットワークの向こう」への通信になっている。
- このため、クライアント側では「どのホスト宛接続する」という情報を供給する必要がある。
- サーバ側では RPC 呼び出し全体を管轄し、指定されたプログラムのスタブへの橋渡しを行うための「RPC サーバ」プログラムを動かしている必要がある。
- FreeBSD、Solaris などでは ONC RPC と呼ばれる RPC システムが動いていて、「rpcinfo」でサーバ側のプログラム登録状況が見られる。

3.3 ORB と CORBA

□ オブジェクト指向と ORB

- ここまでは「手続き指向」だったが、オブジェクト指向の普及とともに、呼び出される対象は「オブジェクトのメソッド」と考える方が素直に思えるようになってきた。
- サーバ側には「複数のオブジェクト」があり、それぞれが固有のデータを持っている。そのオブジェクトのメソッドを RPC で呼び出すことで、その固有データに間接的にアクセスできる。
- その遠隔呼び出しの橋渡し (バインディング) → ORB(Object Request Broker) と呼ばれるようになった。

□ CORBA --- ORB の標準仕様。1990 頃からある。

- ORB としては色々なものが作られたが、相互運用性のために共通のものを作ろうという話になり、OMG(Object Management Group) が作った標準仕様が CORBA(Common Object Request Broker Architecture)
- CORBA 仕様に従っているなら、どのクライアントとどの ORB を組み合わせても呼び出し利用できる (はず)
- ただし Microsoft はこれにくみせず COM/DCOM を採用 (Windows 用の CORBA ソフトももちろん作られたが)。
- CORBA IDL(Interface Definition Language) --- どんなメソッドがどんなパラメタを持っているか、を指定する標準的な記法として使われるようになった。(スタブジェネレータに食べさせるのが本来の目的だったがどちらかというと人間が読む仕様記述の記法として…)

□ CORBA の現在…

- 汎用的すぎて面倒であり遅かった。そしてあまり普及しないうちに Web アプリケーションの時代になってしまった。
- Web アプリでは遠隔呼び出しに CORBA よりは SOAP(Simple Object Access Protocol) など XML 系の技術を使う。Web アプリ側が接続先を指定できるので ORB はなくても済んでしまう。
- OMG は今は UML などの方に力を入れている (UML はメジャーになったのでそれなりに成功している)。
- それはそれとして、現在でも Java などから一応 CORBA は使えるようになっている。

4 Java RMI

4.1 Java RMI の概要

□ RMI(Remote Method Invocation) --- 遠隔オブジェクトに対してメソッド呼び出しを行うような機能全般。

□ Java では比較的初期からそのための機能が組み込まれている→ Java RMI。

- ORB、スタブジェネレータなども一通り揃っている。
- Java 言語の枠内で揃うようになっているので、CORBA より簡単。
- データだけでなく、コードも転送可能→大きな特徴。

□ 以下で先の例題をもとにした「文字列記憶サービス」を用いて RMI の必要事項を説明していく。

4.2 例題:文字列サービス

□ [1] インタフェース仕様

- Java RMI では CORBA IDL の代わりに Java 言語に備わっているインタフェース定義で遠隔オブジェクトの仕様を規定する。
- このインタフェース仕様は必ず「extends Remote」を指定する。これにより、このインタフェースは遠隔オブジェクトの仕様として使われることが宣言される。
- このインタフェースに現れるメソッドはすべて RemoteException という例外を発生させ得るように指定する。
- 実際、ネット経由で呼び出すのでネットワークが切断したらエラーになるから、そのような場合を必ず考慮するべき。(今回は例題を短くするためにさぼっている。)

```
import java.rmi.*;

interface Sample13IF extends Remote {
  public String exec(String s) throws RemoteException;
}
```

□ 解説:

- Remote インタフェースである。
- メソッドは文字列を 1 つ受け取り 1 つ返す exec() のみ。
- このメソッドは RemoteException を発生させ得る。

□ [2] サーバ側コード

- サーバ側に置かれたオブジェクトは extends UnicastRemoteObject と指定されている必要がある。
- サーバ側に置かれるオブジェクトは [1] で宣言したインタフェースを実装する必要がある。
- このクラス名を指定して Java 付属のスタブジェネレータ rmic を実行することでクライアントスタブ (Java ではスケルトンと呼ぶ)、サーバスタブ (Java では単にスタブと呼ぶ) が生成される。
- 呼び出し方は次の通り (☆のところにクラス名を指定)

```
% /compat/linux/usr/local/Java/jdk1.6.0_05/bin/rmic ☆
```

□ サーバ側コードはクラス本体部分と main() に分けて説明する。まずクラス本体部分から:

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.io.*;

public class Sample13Server
    extends UnicastRemoteObject implements Sample13IF {
  String data;
  public Sample13Server() throws RemoteException {
    data = "?";
  }
  public String exec(String s) {
    data = data + s; return data;
  }
  public String toString() {
    return data;
  }
```

□ 解説 (サーバクラス):

- extends UnicastRemoteObject の指定必要。
- implements Sample13IF の指定必要。

- 変数 data は記憶する文字列を保持。

```
  public static void main(String[] args)
      throws Exception {
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    Sample13Server srv = new Sample13Server();
    Registry reg = LocateRegistry.createRegistry(4918);
    reg.bind("str1", srv);
    System.out.print("command> ");
    while(!in.readLine().equals("bye")) {
      System.out.println(srv.getData());
      System.out.print("command> ");
    }
    System.exit(0);
  }
}
```

□ 解説 (main):

- コマンド入力のための入力ストリーム用意。
- サーバオブジェクトを作る。
- レジストリ (登録サーバ) をポート 4918 で作る。
- レジストリにサーバを str1 という名前で登録。
- コマンド入力ループで、bye という行が入力されるまで 1 行入力ごとに現在のサーバの文字列表現を書き出す。

□ [3] クライアント側コード

- クライアントはこれまでと同じく GUI を持つ。
- 初期設定のところでサーバのリモートオブジェクトを取得
- あとはこれに対するメソッド呼び出し→ RMI によりサーバで実行

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.rmi.*;

public class Sample13 extends JFrame {
  Sample13IF srv;
  JTextField f1 = new JTextField();
  JButton b1 = new JButton("Exec");
  JLabel l1 = new JLabel("start...");
  public Sample13() throws Exception {
    srv = (Sample13IF)
      Naming.lookup("rmi://localhost:4918/str1");
    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container c = getContentPane(); c.setLayout(null);
    c.add(f1); f1.setBounds(40, 40, 180, 40);
    c.add(b1); b1.setBounds(240, 40, 80, 40);
    c.add(l1); l1.setBounds(20, 360, 360, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          f1.setText(srv.exec(f1.getText()));
        } catch(Exception ex) { l1.setText("!" + ex); }
      }
    });
  }
  public static void main(String[] args)
      throws Exception {
    new Sample13().setVisible(true);
  }
}
```

□ 解説:

- Naming.lookup() でホスト、ポート、名前を指定→そのホストの RMI レジストリ (先に main() で起動したもの) に接続してリモートオブジェクトを取得
- ボタン押した時は単に srv.exec() を呼ぶだけでよい (簡単)。

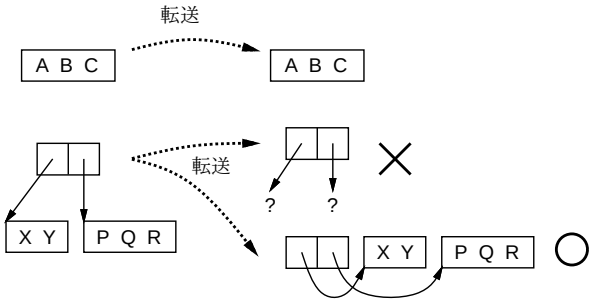
□ 演習: 文字列サービスを動かして前の演習と同様に確認してみよ。

5 オブジェクトとコードのモビリティ

5.1 モビリティ(可動性) とその内容

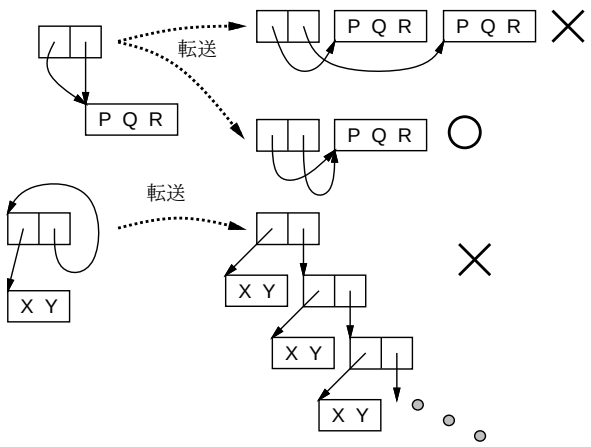
□ モビリティ(mobility) →ネットワークを通じて「××」が移動可能なこと。

- 通常のデータ (テキストとか) が移動するのは当り前だが…
- 「通常」とは? たとえばポインタは? →ポインタは「特定のマシンの上でのメモリアドレス→そのまま別のマシンに移動させても「とんでもない番地を指すポインタ」になるだけ。
- 正しくは「ポインタの指す先のデータ」もコピーする必要がある。



□ 構造のあるデータのコピーは必ずしも簡単ではない…

- 共有部分があったらどうするか? → その形を保ってコピーしたい。
- ループがあったらどうするか? → その形を保ってコピーしたい。



- ☆ポインタを順次たどっていくが、既に転送に着手したのと同じものに到達したら、新たに転送する代わりに「既に転送した何番目と同じ」という情報を送り、復元時にもそれに応じて復元。

□ コピー vs 参照セマンティクス

- コピーするということは、行った先と手元は別物になるということ。
- それでよい場合もあるが、オブジェクトは勝手に複製されてほしくないという場合もある。
- その場合、オブジェクトは動かずにずっとどこかに存在し、そのオブジェクトを指す参照はあちこちに渡して、それらの参照からオブジェクトを呼び出して利用、という考え方もある← CORBA など。
- しかしその方法しかないとなると、すべて RMI のようになるため遅いという問題がある。
- CORBA では言語が様々なのでどうにもならないが Java では「コピー」も「参照」も可能。参照は先にやったのがそれ。ではコピーは…

□ Java の場合は…

- Serializable インタフェースを implements で指定したクラスのオブジェクト→直列化可能 (ネットワーク経由での転送、ファイルへのそのままの格納が可能)
- そのときに前記☆のような制御は用意されている。さらに特別な処理をしたい場合はそれぞれのオブジェクトにメソッドを用意することで制御可能。
- 直列化可能なクラスのインスタンス変数は基本型 (単純なデータ) または再び直列化可能なクラス型でなければならない (送れる必要があるから)。
- インスタンス変数に transient というキーワードを指定しておくとその変数は送らない (復元したときは null になる)。そのため上記の制約もない。

□ コードモビリティ (モバイルコード)

- 受動的なデータだけならただ送ればよいが、オブジェクト指向言語では「データ+メソッド」がオブジェクト。
- このため、オブジェクトのデータを送ったが送った先にそのオブジェクトのメソッドのコードがなかった、では済まされない。
- 1 つの方法…あらかじめ必要なクラスのコードは用意させる。それがある状態でのみ送ることとする (普通のアプローチ)。
- 別の方法…オブジェクトを送る時、送り先にそのクラスのコードがまだなければコードも送る (Java のアプローチ) ← コードモビリティ

□ なぜ Java でこれができるか?

- もともと「アプレットのための言語」→コードをネット経由で転送することを前提として開発された言語+環境 ← i-アプリもそう
- 具体的には、Java バイトコードを送ってそれぞれの実行環境の JVM (バーチャルマシン) で実行→バイトコードは CPU 種別などに影響されずすべて共通→これを送れば OK。
- ただし、「コードが送られる」ことにはセキュリティ上の危うさがあるのでそのための工夫も必要 (今回は詳しくはやらない)。

5.2 例題:文字列サービス改訂版

□ [1] インタフェース仕様

```
import java.rmi.*;
import java.io.*;

interface Sample14IF extends Remote {
  public Data get() throws RemoteException;
  public void put(Data d) throws RemoteException;
  interface Data extends Serializable {
     public String exec(String s);
  }
}
```

□ 解説:

- インタフェース Sample14IF の中に Data というインタフェースを定義している。このインタフェースは extends Serializable と指定しているので、これを実装するオブジェクトは転送可能。
- Data オブジェクトにはこれまでサーバにあった exec() を持たせる。
- サーバは今度はこの Data オブジェクトを取り出したり戻したりするようなインタフェースに直した。

□ [2] サーバ側コード

- Data インタフェースを implements するクラス MyData を用意。このオブジェクトがクライアント側に転送され、そこで動作する。

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.io.*;

public class Sample14Server
    extends UnicastRemoteObject implements Sample14IF {
  Data data;
  public Sample14Server(Data d)
    throws RemoteException { data = d; }
  public Data get() { return data; }
  public void put(Data d) { data = d; }
  public String toString() { return data.toString(); }
  static class MyData implements Data {
    String str = "?";
    public String exec(String s) {
      str = str + s + "!"; return str;
    }
    public String toString() { return str; }
  }
  public static void main(String[] args)
      throws Exception {
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    Sample14Server srv = new Sample14Server(new MyData());
    Registry reg = LocateRegistry.createRegistry(4918);
    reg.bind("str1", srv);
    System.out.print("command> ");
    while(!in.readLine().equals("bye")) {
      System.out.println(srv.toString());
      System.out.print("command> ");
    }
    System.exit(0);
  }
}
```

□ 解説:

-
- サーバは最初 MyData オブジェクトを与えて初期化する。
- get()、put() はその Date オブジェクトをやりとり。
- toString() では Data オブジェクトの toString() を返す。
- クラス MyData はこれまでのサーバと類似しているが、通信機能はなく、単にデータを保持し計算。

□ [3] クライアント側コード

- フィールド、ボタンとも 2 つに増えている。
- 最初のフィールドには「rmi://sma:4918/str1」などの URI を与えて Conn ボタンを押すとサーバから Data オブジェクトを取得。その状態で再度 Conn ボタンを押すと取得してあった Data オブジェクトを書き戻す。
- 2 番目のフィールド/ボタンはこれまでの機能に類似。ただしサーバが処理を行うのではなく、手元に持って来た Data オブジェクトが処理を実行する。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.rmi.*;

public class Sample14 extends JFrame {
  Sample14IF srv;
  Sample14IF.Data data;
  JTextField f0 = new JTextField();
  JButton b0 = new JButton("Conn");
  JTextField f1 = new JTextField();
  JButton b1 = new JButton("Exec");
  JLabel l1 = new JLabel("start...");
  public Sample14() throws Exception {
    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container c = getContentPane(); c.setLayout(null);
    c.add(f0); f0.setBounds(40, 40, 180, 30);
    c.add(b0); b0.setBounds(240, 40, 80, 30);
    c.add(f1); f1.setBounds(40, 80, 180, 30);
    c.add(b1); b1.setBounds(240, 80, 80, 30);
    c.add(l1); l1.setBounds(20, 360, 360, 30);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          if(srv == null) { // connect & get
            srv = (Sample14IF)Naming.lookup(f0.getText());
            if(srv == null) l1.setText("Connect failed.");
            else { l1.setText("OK."); data = srv.get(); }
          } else { // save
            srv.put(data); srv = null; l1.setText("save.");
          }
        } catch(Exception ex) { l1.setText("!" + ex); }
      }
    });
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          if(data != null)
```

```
            f1.setText(data.exec(f1.getText()));
          } catch(Exception ex) { l1.setText("!" + ex); }
        }
      });
  }
  public static void main(String[] args)
      throws Exception {
    new Sample14().setVisible(true);
  }
}
```

□ 解説:

- 部品の配置などはこれまでと同様。
- ボタン b0 の動作では、srv が null であれば f0 に指定された文字列を使って Naming.lookup() を呼びサーバを取得。さらにサーバから Data オブジェクト取得。
- srv が null でないならそこに Data オブジェクトを書き込んで srv を null に。
- ボタン b1 の方はこれまでと同様ただし data に対して実行。

□ 演習: この例題を動かせ。文字列を加工してもサーバ側には反映されていなくて、最後に書き戻すと反映されることを確認すること。

□ 演習: クラス MyData の exec() の中身を変更して自分独自のものにしてみよ。続いて自分のサーバを動かし、動作を確認。OK なら、その rmi: URI をホワイトボードに書き、他人に使ってみてもらう (また自分も他人のものを動かしてみる)。

6 まとめ

□ 分散システムの技術として、C/S モデル、分散オブジェクトモデル、ORB、Java RMI、モビリティなどの話題をひととおり取り上げた。実際にお仕事に使うとなるとさらに大変だが、世の中は結構頑張っていると思いますよね? (しかし今日では Web アプリが主流に…)