情報システムとWeb技術＃3 — 業務モデリング/サーブレットとセッション管理

久野　靖*

2011.10.18

1　業務モデリング

1.1　業務モデリングとその位置づけ

ここまで、前々回にDFDによるシステムのモデル記述、前回にIDEF1Xによるデータモデル記述を取り上げたが、今回は業務モデリング、つまり業務そのものをモデル記述することについて取り上げる(実のところ、DFDで業務を記述することは十分可能だが、DFDだとデータの流れしか記述しないので、もう少し細かいことまで記述したモデルについて学ぶということ)。

具体的には、業務モデルとはどのようなものでしょう? たとえば営業担当者であれば、上司による客先の割り当てや売り上げ目標、重点商品などの指針に基づき、客先を回って売り込み、引き合いがあれば在庫や配送状況を調べて価格交渉を行い、契約を取りますが、それらの活動それぞれについて状況をデータとして蓄積したり、必要な参照データ(在庫、配送、価格現況)、生成データ(契約内容、商品の発送先や発送日時)を読み書きする必要があります。

営業なら大まかな枠組みは上のようかも知れませんが、細かい情報の動き方は商品や業界慣行、企業文化によってすべて違っています。そして、実際に行っている仕事に対応したデータがサポートされなければ困るわけです。これらをきちんと同定して「どこで何が起こってどんなデータが出入りする」ことを明確化したモデルが業務モデルです。

しかも、実際の仕事は営業だけでは済まず、製造、出荷、経理など多種の業務が相互に組み合わさって会社の仕事が達成されているわけですから、企業全体の業務モデルは極めて複雑になります。

いちばん望ましいのは、企業活動全体についての業務モデルがまず存在していて、それを持って来て必要な範囲の分析を行うことです。しかし現実には、システム設計者が必要な範囲の業務モデルをそのつど分析/構築してそれに基づいてデータ設計を行うのが普通、といったところでしょうか。

なお、**ERP**(Enterprise Resource Planning)とは、企業全体の業務モデルを「パッケージソフトウェア(SAPが有名)に一定範囲のカスタマイズを施したしたデキアイのものとして」システム側から提供してしまい、それに企業活動の方を合わせることで企業全体の業務のコンピュータサポートを行うことだ、と考えることができます。(むちゃくちゃ大変そうですが…)

全社的な業務モデルと関連して必要なのは、各所で使われているデータの呼び名を標準化し統一することです。なぜなら、同じものをA部門では「顧客レコード」、B部門では「顧客情報」と呼んでいたら、それが同じものであることがよく分からないからです。そこで、このような名前の統一を行い、さらにそのデータの内容についても同じになるように調整する作業が必要になります。このような仕事を専門に行う人を、**DA**(Data Administrator、ないしData Architect)と呼びます。DAは企業内の各部署に渡って使用する名前やデータ構造の内容を制定し従わせるという、強い権限を持つことになります。

*経営システム科学専攻

データモデルの記述については前回やったので、今回は取り上げませんが、実際には業務モデルを作成する過程において、どのようなデータが扱われているかも一緒に分析することになるので、データモデルも並行して作成していくことになるでしょう。そして、モデルが完成したらそれに基づいて今度はどのようなシステム化を行うかを検討して行くわけです (現状をまず分析してからシステムについて検討しないと、システム構築後が現状とどのように違うかが明確にならないので、問題点の検討や以降方法の検討などがうまく進められない危険があります)。

質問 1 あなたは業務分析や業務のモデル構築をやったことがありますか? そのときはどのような手法や記法を使いましたか? やってみてどうでしたか?

1.2 業務モデルの図法 IDEF0

情報システムの世界でモデルといった場合、さまざまなモノやそのつながり方を表すため、特定の図法を用いてあらわす、ということが多く行われています。たとえばプログラムの実行の流れを表すのには、過去においては**フローチャート**が多く使われて来ました (が、今では下火になっています。なぜか分かりますか?)。

業務モデルなど各種のモデル化のアプローチは、プロセス指向アプローチ、すなわち「どのような処理/プログラミングを行うか」に焦点を当てるものと、**データ主導アプローチ** (DOA, Data Oriented Approach) に大別できます。プロセス指向アプローチに使われる図法としては、今日では開発に広く使われているオブジェクト指向言語と親和性のよい **UML**(Universal Modeling Language) が一般的になっています。

データ主導アプローチでも UML は適用可能ですが、ここでは DOA で多く使われている業務モデルの図法として、米国空軍が情報システムの仕様書を標準化するために実施したプロジェクト (ICAM、Integrated Computer Aided Manufacturing program) のための定義用図法 **IDEF**(ICAM Definition) の中の **IDEF0** を紹介します。

IDEF0 では、業務の中の個々の活動を**アクティビティ** (activity) と呼び、長方形で表します。そして、アクティビティに関連する情報や制御やモノの流れを次の 4 種類に分類し、4 辺のそれぞれに矢線で記述します (図 1)。ちなみに、アクティビティを行う主体や、矢線上を流れる情報/モノなどをまとめて**エンティティ** (entity、「もの」という意味) と呼びます。

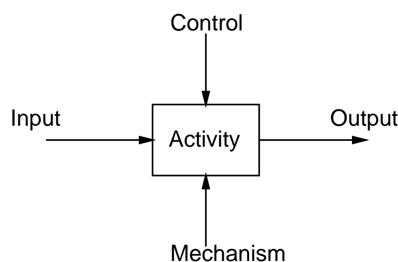

図 1: IDEF0 のアクティビティ

IDEF0 におけるアクティビティでは、上下左右に接続されるものはそれぞれ次のように決まっています。

- Input — 左。そのアクティビティの入力となる (主に処理される、また加工され得る) もの。「何を、どこから入力」を表現。
- Output — 右。そのアクティビティの出力となる (結果として出て来る) もの。「何を、どこへ出力」を表現。

- Control — 上。そのアクティビティが参照する (書き換えたり加工はしない) もの。「何を、どこから入力」を表現。
- Mechanism — 下。「誰が」「何を使って」「どういう方法で」「どういう要件」などの情報を記述。

たとえば、販売側が確定注文に応じて商品を配送業者に依頼して購入側に送るという業務フローをIDEF0 で記述すると図 2 のような感じになります。1)。

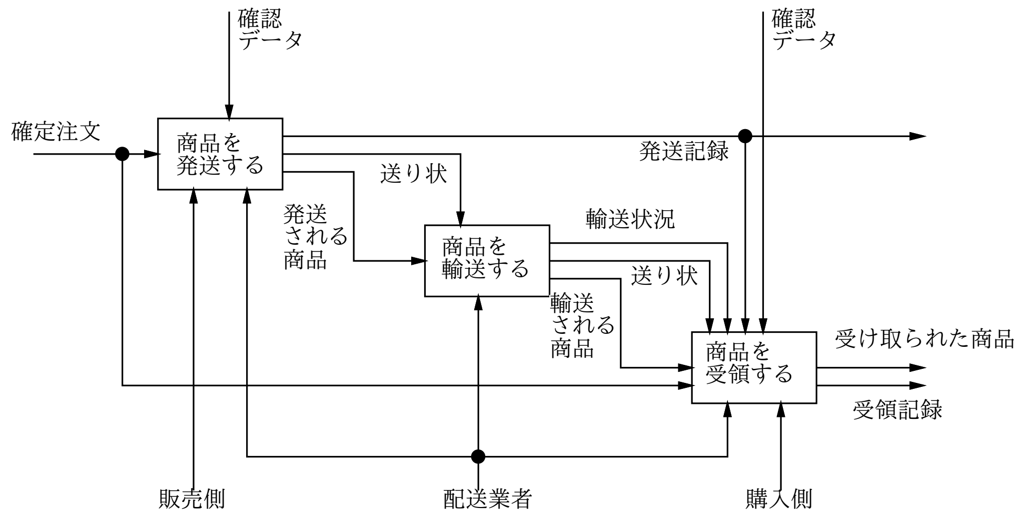

図 2: IDEF0 による記述のサンプル

具体的な各アクティビティの説明は次のとおりです。

- 「商品を発送する」では、販売側が、確定注文を確認データと突き合わせた上で、商品を送り状とともに配送業者に引き渡し、発送を記録する。
- 「商品を輸送する」では、配送業者が、発送される商品と送り状を輸送して受け取り側に届けますが、問い合わせがあれば輸送状況もそのつど返答する。
- 「商品を受領する」では、購入業者が配送業者から、商品と送り状を受け取り、確定注文の控え、および確認データと照合し、受領を記録する。

実際には、さらに「発送」「受領」「輸送」の中も複数のアクティビティから成っていると考えられ、それぞれの四角の中をさらに複数の四角に分解したモデル図も必要ならそれぞれ作るかも知れません。かなり面倒そうですが、このようにして業務の流れを**すべての**出入りする情報と一緒に記述できれば、それらの情報のどれとどれをデータベースに格納し、その中には何が含まれているかを検討する役に立ちそうだということはお分かり頂けると思います。

演習 1 あなたが普段よくやっているか、ないし自分が担当していなくてもよく知っている業務を 1 つ取り挙げ、その業務「近辺」の業務モデルを IDEF0 で記述してみなさい。どの範囲まで、については「切りのいい」範囲を判断して決めてください。

演習 2 あなたの会社が「個人顧客むけの引越し事業者」であるものとします。客から注文を受けて引越しを実施し代金を受け取る業務全体の業務モデルを IDEF0 で記述してみなさい。あなたが社長でこれから会社を起こすので、業務の構造は好きなように決めて構いません。

2　動的 Web ページとサーブレット

2.1　Web サーバのさまざまな役割

質問 2 前回までの Java で組んだ超お手軽な Web サーバと、実際にサイトの運用に使われている実用の Web サーバの違いは何と何だと思いますか?

先の質問には非常に沢山の答えがあると思いますが、思い付くものをとりあえず挙げてみましょう。

- HTTP プロトコルのより多くの機能に準拠 (コンテンツネゴシエーションなど)
- 性能向上の工夫 (接続の維持やサーバのプーリングなど)
- さまざまなサイトで運用するための柔軟性 (設定ファイルによるカスタマイズなど)
- 運用管理のための機能 (ログの保存、停止や再起動など)
- 安全性のための機能 (SSL のサポートや証明書の管理など)
- 動的コンテンツのための機能

どれにしても、原理的には「作ればいいだけ」ですが、実際に実用レベルで実装するとなると面倒なものばかりです。これを考えると、Apache など実用に供されている Web サーバが大規模で複雑なのも分かる気がします。こういうインフラは素人の手作りでは無理ですよね。

2.2　動的ページの必要性

ところでこの科目では情報システムの一環としての Web アプリケーションを取り上げるので、上の機能の中でも「動的コンテンツのための機能」がとりわけ重要となります。

歴史的に見ても、Web サーバが単に設置された HTML(静的な Web ページ) を世界に公開しているだけのときは、今から見たらあまり面白いことはできなかったように思われます。もちろん、世界中に情報発信ができて、自分が作成したページを世の中の多くの人が見てくれるようになった、というのは非常に大きな変化ではあったのですが…

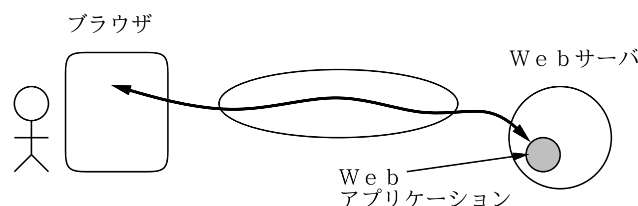

図 3: 動的ページによる Web アプリケーション

Web アプリケーションないし動的ページは、概念的には図 3 のように、ブラウザからの (ユーザの入力した) データを受け取って処理し、またその応答をブラウザに返すようなものを言います。では、それは一体どうやって作ったらいいのでしょう? もちろん、自分が Web サーバを全部書けば、ブラウザからやってきた情報を受け取るのも生成した HTML を送るのも全部自在です。しかしそれではさっきまでの「全部手作りで大変」に戻ってしまいます。

質問 3 サーバはプロがきっちり書くとして、その上であなたが上のような Web アプリケーション/動的ページを作成できるようになるためには、どのようになっていればいいと思いますか?

2.3 CGIとサーブレット

前節の問題を分かりやすく定式化すると、次のようになります。

> 基本的にはWebサーバがブラウザとやりとりしているのだが、その一部についてだけ、自分で書いたプログラムが動いてブラウザからのユーザ入力を受け取ったり、ブラウザに返すHTMLを出力したりできるようにする。

つまり、Webサーバの下でプログラムが動き、外部とやりとりできるような**インタフェース**が定義できればいいわけです。実際にそのようなインタフェースが作られ、**CGI**(Common Gateway Interface)と名付けられました。今でもPHPやPerlなど多くのサーバ側プログラムはCGIの上で動作しています。

CGIの基本的な仕組みは次の通りです (図4左)。

- サーバはブラウザからのHTTP要求を受け取り、そのURLが(設定ファイルやURL中の拡張子などにより)CGIプログラムだと分かったら、そのプログラムを起動する (どのURLがどのプログラムかの対応が設定ファイルから分かることもあるし、URLがファイルのありかを直接表していることもある)。
- プログラムが起動されるとき、サーバはHTTP要求ヘッダをはじめとするWebアプリにとって有用な情報を環境変数に入れてプログラムに渡す。また、ブラウザからの入力はプログラムの標準入力に接続される。
- プログラムの標準出力は、まず空行が来るまではサーバが読み取り、そこに書かれた応答ヘッダのひな型などをもとにサーバが正式な応答ヘッダを作成する。
- 空行から後はブラウザへの送信ストリームにそのままつながれ、そこに出力したものはブラウザに送られる。

CGIは確かに必要なことはすべてできるのですが、リクエスト毎にプログラムを別プロセスとして起動するためにのろく、また環境変数やストリームの冒頭部分 (空行までの部分) によるデータの受け渡しなど、いまひとつ美しくないところもあります (性能向上のためにいろいろ汚い工夫もされています)。

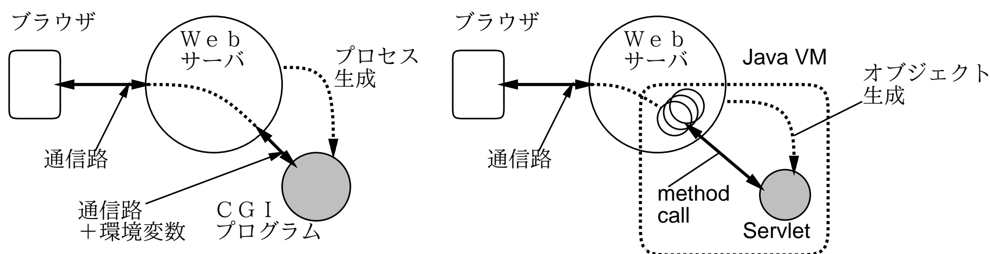

図4: CGIとサーブレット

そこでJava文明 (?) がCGIの代替案として作り出したのが、**サーブレット**という概念です (図4右)。サーブレットはプロセスではなくJavaのオブジェクトであり、必要とされた時にサーバの中に生成され、必要とされている間そこにとどまります。

サーブレットに対応するリクエストが来た時は、プロセスが新たに作られる代わりに、そのリクエストに対応するスレッドが割り当てられ、そのスレッドがサーブレットのコードを実行します。このため、プロセスを生成するよりもずっと軽量な処理ができます。

また、サーバとサーブレットの間のやりとりも、普通にオブジェクトのメソッドを呼び出すことで自由に行えます。このため、サーブレットの方がCGIよりも分かりやすい形でWebアプリケーションのためのさまざまな機能を活用することができるわけです。

2.4 サーブレット API

では具体的には、サーブレットはどのような形で作るのでしょうか。それは、サーブレットが持つべきさまざまな機能 (サーバとやりとりする等) を備えた土台のクラスが用意されていて、そのサブクラスとして自分が作りたい機能を持つサーブレットをコーディングします。つまり、次のような形になります。

```
public class MyServlet extends HttpServlet {
  自分が保持する変数の定義…
  protected void doGet(HttpServletReqest req, HttpServletResponse res)
      throws IOException, ServletException {
    GET リクエストに対する動作…
  }
  POST、PUT、DELETE 等も同様…
  下請けメソッドを追加してもよい…
}
```

doGet() 等のメソッドは HTTP のリクエストタイプ毎に一式用意されていて、土台のクラスでは何もしないような実装が提供されています。それに対してサブクラスを定義して自前のメソッド定義で差し替えることで、自分がやりたい処理をおこなうサーブレットクラスが作れるわけです。

リクエストに対応するメソッドに渡されて来るオブジェクト HttpServletRequest、HttpServletResponse はリクエストから情報を取り出したり応答を返すための機能を提供しています。HttpServletRequest については、次のメソッドをよく使うでしょう。

- String getParamter(String) — パラメタ名を渡すと対応する値を文字列値として返してくれる (その名前のパラメタが存在しないと null を返す)。
- HttpSession getSession() — セッションオブジェクト (後述) を返してくれる。

また、HttpServletResponse については次のメソッドをよく使うでしょう。

- void setContentType(String) — コンテンツ種別を設定する。getWriter() より先に呼び出す必要がある。
- void setContentEncoding(String) — 文字コードを設定する。getWriter() より先に呼び出す必要がある。
- PrintWriter getWriter() — コンテンツ本体を書き出すためのライタオブジェクトを返す。

2.5 簡単な掲示板サーブレット

では最初の例題として、簡単な掲示版を作ってみます。今回は JDBC は使用しないので、掲示板のデータはサーブレットのインスタンス変数内だけに蓄積されていて、サーブレットを止めると無くなります。まあ、例題ということで。

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class Sample31 extends HttpServlet {
  ArrayList<String> list = new ArrayList<String>();
  protected void doGet(HttpServletRequest req, HttpServletResponse res)
```

```
      throws IOException, ServletException {
    process(req, res);
  }
  protected void doPost(HttpServletRequest req, HttpServletResponse res)
      throws IOException, ServletException {
    process(req, res);
  }
  private void process(HttpServletRequest req, HttpServletResponse res)
      throws IOException, ServletException {
    res.setContentType("text/html");
    res.setCharacterEncoding("iso-2022-jp");
    PrintWriter pr = res.getWriter();
    pr.println("<html><head><title>Sample31</title><style type='text/css'>");
    pr.println(".main { padding: 1ex; background: rgb(200,255,120) }");
    pr.println(".form { padding: 1ex; background: rgb(255,220,200) }");
    pr.println(".mesg { padding: 1ex; background: rgb(200,220,155) }");
    pr.println("</style></head><body><h1>Sample31</h1>");
    if("post".equals(req.getParameter("cmd"))) {
      String n = req.getParameter("n"), m = req.getParameter("m");
      if(n == null || n.length() == 0 || m == null || m.length() == 0) {
        putMesg(pr, "name or message is empty.");
      } else {
        Calendar c = Calendar.getInstance();
        list.add(String.format("%d.%d %d:%02d %s: %s",
          c.get(Calendar.MONTH)+1, c.get(Calendar.DATE),
          c.get(Calendar.HOUR_OF_DAY), c.get(Calendar.MINUTE), n, m));
      }
    }
    putMain(pr); putForm(pr); pr.close();
  }
  private void putMesg(PrintWriter pr, String m) {
    pr.printf("<div class=mesg>%s</div>\n", m);
  }
  private void putMain(PrintWriter pr) {
    pr.println("<div class=main>");
    for(String s: list) { pr.printf("<p>%s</p>\n", s); }
    pr.println("</div>");
  }
  private void putForm(PrintWriter pr) {
    pr.println("<div class=form>");
    pr.println("<form method=post action='#'>");
    pr.println("<p>Name: <input type=text name=n size=10></p>");
    pr.println("<p>Mesg: <textarea name=m cols=40 rows=5></textarea></p>");
    pr.println("<p><button type=submit name=cmd value=post>Post</button></p>");
    pr.println("</div>");
  }
}
```

説明を書いておきます。

- インスタンス変数は文字列のリストだけです。ここに書き込みを蓄積します。
- このサーブレットは GET と POST でアクセスしますが、どちらも処理は共通なので下請けメソッド `process()` を作って `doGet()` と `doPost()` からそれを呼ぶようにしています。
- `process()` の中ではまずコンテンツ種別と文字コードを指定して結果を書き込むライタを作り、HTML の冒頭部分を作ります。
- 次に、`"cmd"`という名前のパラメタがあって渡された値が`"post"`であれば投稿があるので、その処理をします。
- 名前かメッセージが空ならエラーメッセージを出します。そうでなければ日時、名前、メッセージを組み合わせた文字列を作ってリストに追加します。
- あとは書き込み本体と書き込み用フォームを出力しておしまい。
- メッセージ、書き込みの表示、フォームとも単に出力するための下請けメソッドが定義してあります。

では、これを動かすためにコンパイルして WAR(Web ARchive) パッケージを作成します。以下の操作を 1 回だけ実行してください。

```
export CLASSPATH=.:/u1/kuno/work/servlet.jar  ←コンパイルに必要
cp -r /u1/kuno/work/WEB-INF WEB-INF           ←ファイルコピー
```

そして以下は修正のたびに実行します。

```
cd WEB-INF/classes                  ← Java コードはここに置いてある
javac Sample31.java                 ←コンパイル
cd                                  ←元の位置に戻る
jar cf ユーザ名.war WEB-INF
```

WAR ファイルの名前は自分の名前の後ろに「.war」をつけたものとしてください (他人と衝突しないため)。

2.6 サーブレットコンテナ Tomcat

サーブレットが動くためには、サーブレットを動かす仕組みを持った Web サーバ環境が必要です。これを**サーブレットコンテナ**と呼びます。ここではフリーソフトである Tomcat と呼ばれるコンテナを使用します。場所は次のところで動かしています (図 5 にトップ画面を示します)。

```
http://w3in:8080/
```

Tomcat では管理者がネット経由で自分のサーブレットを設置することができるので、皆様に管理者になって頂きます (どうせこの実習にしか使用しないし)。トップ画面で「Tomcat Manager」のリンクを選択するとユーザ名とパスワードを聞かれるので、両方とも「tomcat」と入力してください。するとマネージャ画面になります。ここで下にスクロールして「WAR ファイルを deploy」の入力欄まで行きます (図 6)。

ここで Browse ボタンでファイルダイアログを出し、先に作った WAR ファイルを選択してから Deploy ボタンを押すと、上の稼働サーブレットの表の中に自分のサーブレットが現れます。これで準備完了です。あとはブラウザの Location 欄で次の URL を開きます。

```
http://w3in:8080/ユーザ名/Sample31
```

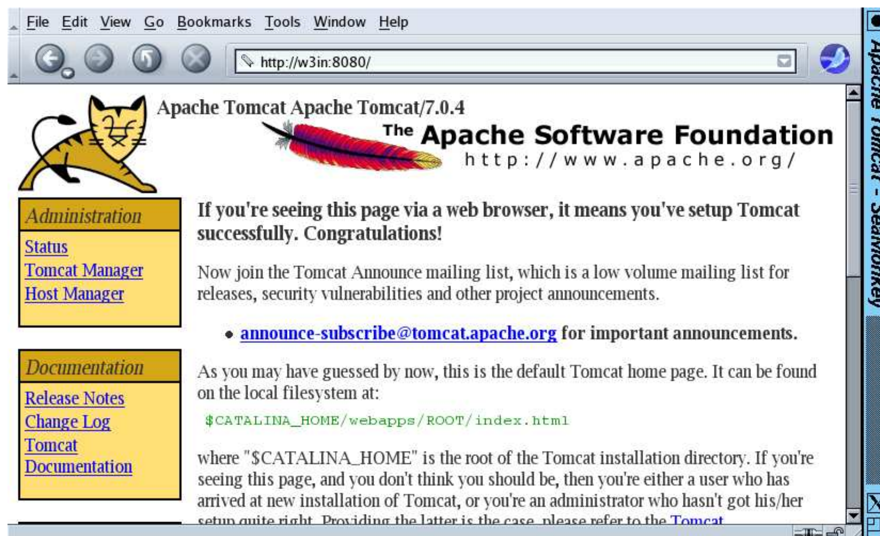

図 5: Tomcat のトップ画面

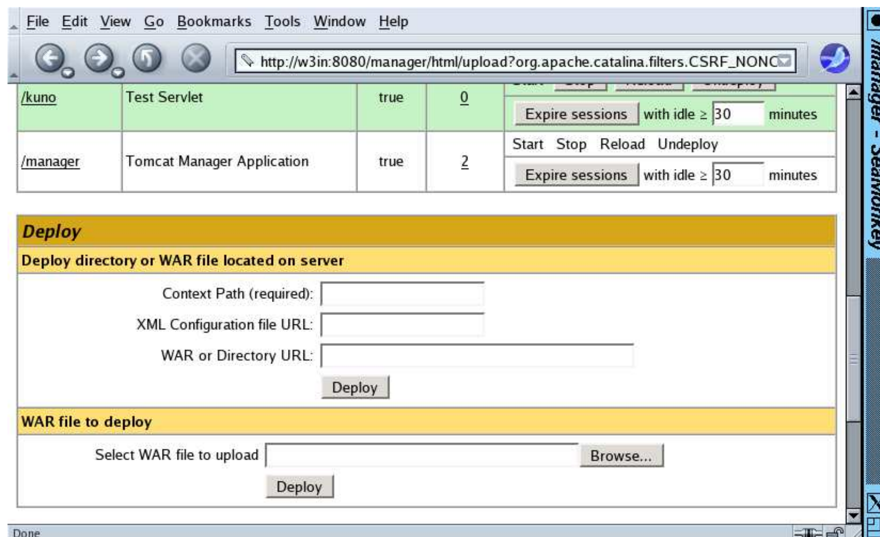

図 6: WAR ファイルを deploy

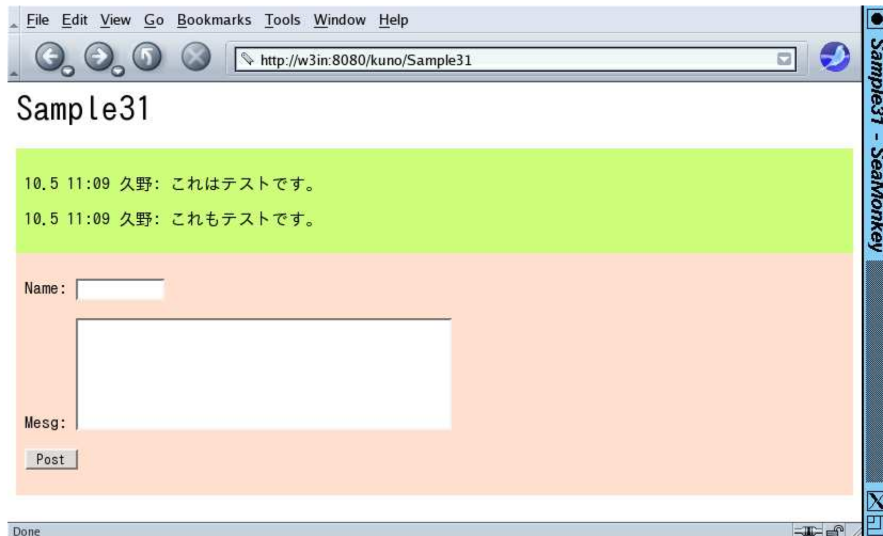

図 7: 簡単な掲示板サーブレット

これでサーブレット画面が現れます。掲示版なので、メッセージを書き込んだらそれが蓄積されて行きます。

演習 3 簡単な掲示板サーブレットをコピーしてきてそのまま動かしなさい。動いたら、次のような変更を行ってみなさい。

a. 色や表示の配置などを改良してみなさい。書き込みは表を使ってそろえた方がきれいかも知れない。他の人にも訪れてもらって、改良になっているかどうかコメントをもらうこと。
b. 表示方法を「時間の逆順」に切り替えられるようにしてみなさい。(ヒント: 逆順表示ボタンを追加する。ここでは 1 回だけ逆順になったら、すぐ元に戻ってしまうのでもよい。ずっと維持するようにもやればできる。)
c. 特定の人の書き込みだけを見るモードを提供してみなさい。または、検索モード (指定した文字列を含む書き込みだけが見える) でもよい (こちらの方がやや簡単)。
d. その他、自分の好きな改良を施してみなさい。

なお、サーブレットを増やす場合は、コピーした WEB-INF ディレクトリの下にある設定ファイル web.xml を適宜修正する必要があります。配布状態だと次のような感じです。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="2.4">
    <display-name>Test Servlet</display-name>
    <description>Test Servlet</description>
    <servlet>
        <servlet-name>Sample31</servlet-name>
        <servlet-class>Sample31</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Sample31</servlet-name>
        <url-pattern>/Sample31</url-pattern>
    </servlet-mapping>
</web-app>
```

ここで、`<servlet>`〜`</servlet>`、`<servlet-mapping>`〜`</servlet-mapping>`の 8 行をそのままコピーして増やし、`Sample31`という名前を新しく増やすサーブレットクラスの名前に変更してください。WAR ファイルに格納されているもののうち、ここで定義されているものだけがサーブレットとして直接呼べるようになります。

3 Web とセッション管理

3.1 HTTP のステートレス性

Web ページのアクセスに使われる HTTP の重要な特徴として、**ステートレス**であることが挙げられます。つまり、各 URL アクセスは、基本的にそれぞれが独立して扱われるということです。これがどういうことか分かりますか?

たとえば、伝統的なメールやリモートログインのプロトコルを考えてみます。これらでは、まずサーバに TCP 接続を張り、そして自分が誰であるかを告げて認証を行います。そして認証をパスしたら、自分のメールを読み書きしたり、自分のファイルを読み書きしたりできます。その途中で誰か別の人の指令が割り込んで来ることはありません (他人に私のメールを読み書きされたら困りますよね)。そして最後に接続を切って終わります。

Web でもこのようにすれば一見良さそうですが、TCP 接続を管理するというのは一定のコストが掛かります。たとえば、あるサーバに載っているユーザ数が 1000 人であれば、最大 1000 接続が同時に張られたままになっても何とか耐えられるでしょう。しかし Web サーバは世界中から何十万ものアクセスが来ることがあります。それだけの接続を張って保持することは基本的に無理です。

そこで、Web サーバでは基本的に、「この URL が欲しい」と言われたらその URL のページを返送し、それで終わりにし、何も覚えておかないことにしました。これを**ステートレス** (状態を持たないという意味) と呼び、これによって Web は大量の負荷に耐えられるようになったわけです。

3.2 Web にけるセッション機能

負荷の問題は解決しましたが、しかし単純な Web ページではない Web アプリケーションの場合は、やはり「私は誰々です」ということを教えた状態で次々に処理をしたいこともあります。

このとき、1 番最初のページで自分のユーザ名とパスワードを入れて送るところは問題ありません。しかし、2 番目以降のページで毎回、同じものを入れてもらうのは繁雑ですし、ページ毎に認証処理を行うのは重くもあります。

そこで、最初に認証を行ったら、そこで乱数を生成して、「鍵」としてブラウザに返します。ブラウザはその「鍵」を覚えておいて、同じサーバにアクセスするときには同じ「鍵」を渡します。サーバ側では「鍵」とユーザの対応を覚えてあるので、それをもとにそのユーザの処理の続きを実行できるわけです。

これは実質的に、先の TCP 接続を張るのと同じような効果をもたらすので、**セッション**と呼ばれます。ただ、TCP と何が違うかというと、ユーザのセッションの鍵情報はユーザ側のブラウザで記憶されていて、サーバは基本的にはそのための資源を保持しておかなくてよいことです (もちろん、データベース等に鍵とユーザ ID の対応表を覚えておくなどのことは必要ですが、これは単なるデータなので重くありません)。

「鍵」を覚えておく方法として、3 つのやり方が主に使われます。

(1) HTML フォームの隠しフィールド (input type=hidden) に鍵を覚えさせる。

(2) HTTP クッキー機能を用いて鍵を覚えさせる。

(3) 鍵を URL の一部としてエンコードして覚えさせる。

(1) はHTMLの設計当初からある方法で、全てのページがフォームを持つような設計になっていれば使用可能ですが、一時的にでもフォームを持たない別ページに移動されてしまうとアウトなので、今ではほとんど使われません。

(2) は便利なので今日もっとも多く使われている方法です (Netscape ブラウザがこのためにクッキー機能を導入していらい、それが事実上の標準として使われています)。クッキーの送受はHTTP要求ヘッダのCookie:ヘッダとHTTP応答ヘッダのSet-cookie:ヘッダによって行われます。

(3) はクッキーをOFFにしているブラウザのために考えられた機能ですが、今日ではほとんどの動的サイトがクッキーをONにしないと役に立たないため、あまり見かけなくなっています。

今回とりあげているJava Servletではセッション機能としてまず(2)、クッキーOFFの場合は(3)が使われるようになっています。その判別は自動で行われるので、Servletプログラマはいちいち気にしなくてもおまかせで大丈夫です。

3.3 Webアプリケーションのセキュリティ

ところで、前節の説明で乱数による「鍵」を生成するとしていましたが、そんな面倒なことをしないでもユーザIDをそのままブラウザに覚えさせておけばよくないでしょうか? そうすれば鍵からユーザIDを検索してくる手間が無くなります。

しかしこれは絶対にやってはいけません。というのは、ブラウザは各ユーザの制御下にあるプログラムなので、「他人のIDを勝手に送る」ブラウザを作られてしまうとその他人に簡単になりすまされてしまいます。

また、ユーザ名そのものでなくても、他人に知られ得るようなものであれば同じことです。実はまさにこのようなことが、スマートフォンと「かんたんログイン」がらみで問題にされています (ケータイアプリ開発者がこの問題をきちんと認識していない)。

`http://takagi-hiromitsu.jp/diary/20090802.html`

このほかにも、Webアプリケーション開発においては脆弱性を作り込まないための留意点が沢山あります。この話題ははじめると切りがないので、今わりと評判のいい参考文献だけ挙げておきます。

`http://www.amazon.co.jp/dp/4797361190`

3.4 サーブレットにおけるセッションオブジェクト

ではいよいよ、サーブレットでセッション機能を使う方法について説明します。それは非常に簡単で、既に延べた`HttpServletRequest`オブジェクトのメソッド`getSession()`を使用します。このメソッドはこのリクエストに対応するセッションオブジェクトを返します (この時点でまだセッション機能が使われていなければ新たなセッションオブジェクトが生成されます)。

セッションオブジェクトはクラス`HttpSession`のインスタンスであり、これに対して代表的なものとして次の2つのメソッドを使用します (これ以外にいろいろなメソッドありますがこの2つだけでも済みます)。

- `void setAttribute(String, Object)` — 任意のオブジェクトを指定した文字列を名前としてセッションに格納
- `String getAttribute(String)` — 名前を指定してセッションに格納してあったオブジェクトを取り出す

これを使うことにより、今回はユーザ名と「これまでどの画面 (状態) だったか」を記憶しておくようにします。より複雑なWebアプリケーションでは、もっと色々なものをセッションオブジェクトに保持することになるでしょう。

3.5　Web アプリケーションと状態遷移

状態遷移 (state transition) とは、システムを複数の状態とその間の行き来によって定式化するモデルないし考え方です。図法としては**状態遷移図** (state transition diagram) を用いて記述することが普通です。状態遷移図とは、複数の状態を円 (ないし矩形) で表し、その間の遷移 (行き来) を矢線で表します。矢線にはラベルが付してあり、どのような事象があった場合にその矢線をたどるかが明記されています。

状態遷移図は、計算機科学の中でよく出て来る概念ですが、Web アプリケーションの場合にも有用です。この場合は、状態は「どのような画面」に対応していると考えればよいでしょう。たとえば簡単な例として、ログインしてから複数の書き込みを行う掲示板の状態遷移図を図 8 に示します。

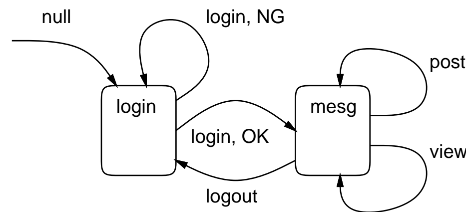

図 8: 簡単な Web アプリケーションの状態遷移図

この状態遷移図では、「何をすべきか」を表すパラメタ (後で `cm` という名前のパラメタを使うので、これをコマンドと呼びます) に応じて状態遷移が起こります。最初にアプリケーションが開始するときはコマンドは空です。この場合は、login 画面を表示します。次に、login コマンドを実行しますが、そのときはパスワードが OK の場合と NG の場合があります。NG の場合は再度 login 画面に戻ります。OK の場合はメッセージ表示画面に進みます。ここでは、logout コマンドが実行された場合は login 画面に戻ります。post コマンドが実行された場合は、書き込みを投稿してからメッセージ表示画面に戻ります。view コマンドが実行された場合は、単にメッセージ表示画面に戻ります。

このように、「現在の状態」と「コマンド」(およびその処理結果) に応じて状態遷移を明記することで、どのような順番で画面が遷移していくかが明確になり、間違いのないアプリケーション設計が可能となるわけです。

なお、セッション管理との関係ですが、これらの状態は「セッションごとに」維持されることに注意してください。たとえば隣の人が logout したからといって、自分も logout させられてしまうのでは困りますから。

演習 4 上の例をもとに、次のような画面 (状態) を持つように状態遷移図を拡張してみなさい。

a. 投稿をいきなり書き込むのでなく、確認画面で確認して投稿/取り止めを選択できるようにする。
b. メッセージの表示を「逆順」(新しい順) にも切り替えられるようにする。
c. 複数回 login することにより「連名」で投稿できるようにする。終わるときは全員ぶん一気に logout するだけでよい。
d. その他、面白いと思う機能 (画面) を追加。

3.6　セッションと状態遷移に基づくサーブレット

では、図 8 の状態遷移を持つサーブレットを作成してみます (図 9)。

基本的な構造ですが、GET でも POST でも区別なく扱うので、両方とも `process()` という共通の下請けメソッドを呼びます。この中ではまず、前の状態と現在の状態に基づいて、必要な処理を行い、次の状態 (画面) を変数 `st1` に入れます。そしてこれをセッションオブジェクトに記憶させ、その

図 9: セッションつき掲示板サーブレット

状態に対応した画面を出力します。実際の画面出力は、process() 本体では共通部分を出力し、あとはメッセージを出力する putNote()、書き込みを出力する putMesg()、書き込みフォームを出力する putForm()、ログイン画面を出力する putLogin() を適宜呼ぶことで各画面を生成しています。

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class Sample32 extends HttpServlet {
  ArrayList<String> list = new ArrayList<String>();
  protected void doGet(HttpServletRequest req, HttpServletResponse res)
      throws IOException, ServletException {
    process(req, res);
  }
  protected void doPost(HttpServletRequest req, HttpServletResponse res)
      throws IOException, ServletException {
    process(req, res);
  }
  private void process(HttpServletRequest req, HttpServletResponse res)
      throws IOException, ServletException {
    res.setContentType("text/html");
    res.setCharacterEncoding("iso-2022-jp");
    PrintWriter pr = res.getWriter();
    pr.println("<html><head><title>Sample32</title><style type='text/css'>");
    pr.println(".note { padding: 1ex; background: rgb(200,255,120) }");
    pr.println(".form { padding: 1ex; background: rgb(255,220,200) }");
    pr.println(".mesg { padding: 1ex; background: rgb(200,220,155) }");
    pr.println("</style></head><body><h1>Sample32</h1>");
    HttpSession ss = req.getSession();
    String st0 = (String)ss.getAttribute("State");
    String cmd = req.getParameter("cmd");
    String st1 = "login";
```

```
    if(st0==null || cmd==null) {
      st1 = "login";
    } else if(st0.equals("login") && cmd.equals("login")) {
      String user = req.getParameter("u"), pass = req.getParameter("p");
      if(pass.equals("hoge")) {
        st1 = "mesg"; ss.setAttribute("User", user);
      } else {
        putNote(pr, "invalid login."); st1 = "login";
      }
    } else if(st0.equals("mesg") && cmd.equals("logout")) {
      st1 = "login";
    } else if(st0.equals("mesg") && cmd.equals("view")) {
      st1 = "mesg";
    } else if(st0.equals("mesg") && cmd.equals("post")) {
      String user = (String)ss.getAttribute("User");
      String mesg = req.getParameter("m");
      if(user != null && !user.equals("") && !mesg.equals("")) {
        String yy = req.getParameter("yy"), mm = req.getParameter("mm"),
               dd = req.getParameter("dd");
        list.add(String.format("%s.%s.%s %s: %s", yy, mm, dd, user, mesg));
      } else {
        putNote(pr, "invalid user or empty message.");
      }
      st1 = "mesg";
    } else {
      putNote(pr, "unknown prev state or command.");
    }
    ss.setAttribute("State", st1);
    if(st1.equals("login")) {
      putLogin(pr);
    } else if(st1.equals("mesg")) {
      String user = (String)ss.getAttribute("User");
      putMesg(pr); putForm(pr, user);
    } else {
      putNote(pr, "unknown state.");
    }
    pr.close();
  }
  private void putNote(PrintWriter pr, String m) {
    pr.printf("<div class=note>%s</div>\n", m);
  }
  private void putMesg(PrintWriter pr) {
    pr.println("<div class=mesg>");
    for(String s: list) { pr.printf("<p>%s</p>\n", s); }
    pr.println("</div>");
  }
  private void putForm(PrintWriter pr, String user) {
```

```
    pr.println("<div class=form>");
    pr.println("<form method=post action='#'>");
    pr.println("<p><select name=yy><option value=2011>2011</option>");
    pr.println("<option value=2012>2012</option>");
    pr.println("<option value=2013>2013</option></select> ");
    pr.println("<select name=mm>");
    for(int i = 1; i <= 12; ++i) {
      pr.printf("<option value=%d>%d</option>", i, i);
    }
    pr.println("</select> ");
    pr.println("<select name=dd>");
    for(int i = 1; i <= 31; ++i) {
      pr.printf("<option value=%d>%d</option>", i, i);
    }
    pr.println("</select>");
    pr.printf("<p>Name: %s</p>", user);
    pr.println("<p>Mesg: <textarea name=m cols=40 rows=5></textarea></p>");
    pr.println("<p><button type=submit name=cmd value=post>Post</button>");
    pr.println("<button type=submit name=cmd value=view>View</button>");
    pr.println("<button type=submit name=cmd value=logout>Logout</button></p>");
    pr.println("</div>");
  }
  private void putLogin(PrintWriter pr) {
    pr.println("<div class=form>");
    pr.println("<form method=post action='#'>");
    pr.println("<p>Username: <input name=u type=text size=12></p>");
    pr.println("<p>Password: <input name=p type=password size=12></p>");
    pr.println("<p><button type=submit name=cmd value=login>Login</button></p>");
    pr.println("</div>");
  }
}
```

演習 5 この例題をそのまま動かしなさい。動いたら次のように修正してみなさい。

a. 演習 4 でやった状態の追加を行ってみなさい。
b. 書き込みに連番をつけなさい (ログイン時に 1 にセットされるとしてよい)。
c. 書き込みに「テーマ」をつけられるようにしなさい。一度書いたテーマはテーマ入力欄に現れるようにすること。できれば表示もテーマ別に並べるか、指定テーマのものだけが見えるかどちらかにするのがよい。
d. その他、セッションを使った面白い仕組みを入れてみなさい。

演習 6 次回はもちろん、JDBC を使ってデータベースにアクセスしつつサーブレットから Web ページを生成する共有日記 Web アプリケーションを作成していただきます。そこでは、第 2 回でやったような「強化した機能を持つ共有日記サイト」を作っていただきたいですが、大変ならとりあえず「基本機能」だけでもよいです。そのために、以下のドキュメントを作成して来て下さい。次回、各自の「成果物」を全員でレビューしたいと思います。

a. 掲示板にどのような独自な機能を持たせるか。それに対してどのような RDB 構成にするか。(データベース設計の IDEF1X)
b. 上記に対して、どのような画面遷移にするか。(状態遷移図)。