NAL

プログラミング
ガイド

ＷＥＢアプリケーション
サーバ・サイドＮＡＬ

版	日付
1.0	

Copyright 1999© ACT corporation.
All right reserved.

はじめに

ＮＡＬについて

ＮＡＬは（Network Abstractive Language）の頭文字をとったものです。
インターネットを含む、広域ネットワーク上のコンピュータにおいてアプリケーションを構築する場合、その手続きおよび概念を簡略化して一元的な取り扱いを記述する目的で開発した言語です。
プログラミングの前提として、オブジェクト指向概念をとっています。
本書をご覧になるにあたり、オブジェクト指向プログラミングの知識が必要です。

著作権について

本書に記述される内容について、そのすべての権利はアクト株式会社に属しています。
いかなる理由においても、権利者の許諾を得ない流用ならびに引用を禁じます。

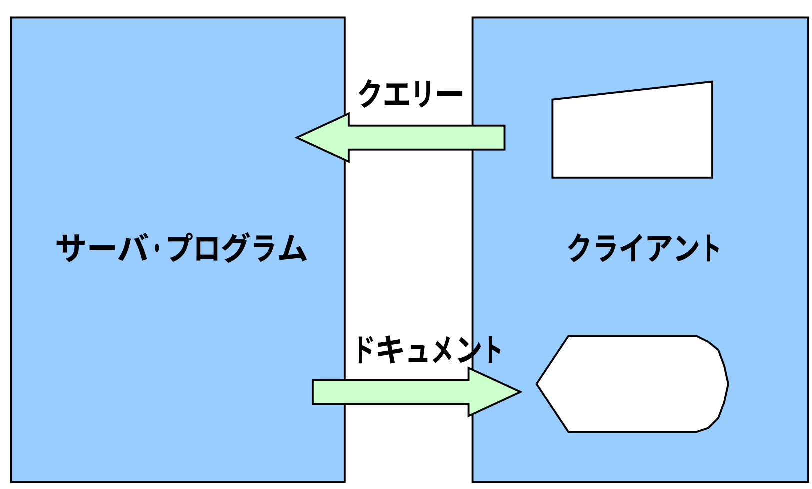
アクト株式会社

改訂履歴

1 ＷＥＢアプリケーションの基本 5
1.1 クエリー 5
1.1.1 サーバへのリクエストに直接記述する方法 5
1.1.2 ＦＯＲＭ部品を使ったクエリーの生成 5
1.1.3 クエリーのデコード 6
1.2 ドキュメント出力 6
1.2.1 pass 文による書き出し 6
1.2.2 write()関数による書き出し 7
1.2.3 式評価出力演算子による書き出し 7
1.2.4 タグの動的生成による書き出し 7
1.3 プログラム記述スタイル 7
1.3.1 文法的位置付けとプログラム記述 7
1.3.2 クエリーの扱い 8
1.4 サーバ・プログラミングへのアプローチ 9
1.4.1 静的ＨＴＭＬドキュメントによるＦＯＲＭ対応クエリーを調べる 9
1.4.2 静的ＨＴＭＬをＮＡＬに取り込む 9
1.4.3 ＮＡＬアプリケーションの動作イメージ 10
1.4.4 値の継承 11
2 サーバ・プログラミングの実際 12
2.1 仕様の決定 12
2.2 画面のデザイン 12
2.2.1 リスト画面 12
2.2.2 入力画面 12
2.3 ＮＡＬプログラムとしてまとめる 13
2.4 データベースの使い方 14
2.4.1 データベースの構成要素 14
2.4.2 データベースの設計 15
2.4.3 データの格納 16
2.4.4 データの取り出し 16
2.5 サーバ連携 18
2.5.1 サーバ連携の概念 18
2.5.2 サーバ連携の実際 18
2.6 多重登録の抑制 21

1　ＷＥＢアプリケーションの基本

ＷＥＢアプリケーションは、サーバ・プログラムがクライアントのブラウザから送付される要求（クエリー）を解釈し、その結果（ドキュメント）を出力することによって、クライアントのブラウザによって表示する。



1.1 クエリー

クエリーは、ブラウザから以下の２通りの方法によって発行される。

1.1.1サーバへのリクエストに直接記述する方法

ＨＴＭＬにおいてリンクタグ（例：<A HREF=″request?key1=value1&key2=value2″>）に固定的に記述する。
request には、サーバにあるＮＡＬスクリプトのロケーションを指定する。
この記述においては、該当するアンカーをユーザがクリックした時点で、
″GET request?key1=value1&key2=value2　HTTP/1.1″という文字列がＷＥＢサーバに届けられる。
ＷＥＢサーバによって request に指定されたＮＡＬスクリプトが起動され、このスクリプトにおいて′?′に続くクエリー″key1=value1&key2=value2″を取り出すことができる。

1.1.2 ＦＯＲＭ部品を使ったクエリーの生成

ＨＴＭＬにおいて、ＦＯＲＭタグとＦＯＲＭ部品を使用してクエリーを生成する。
この場合は、クエリーの key および value の値は、ユーザの操作によって可変となる。
ＦＯＲＭタグの action オプションには、サーバにあるＮＡＬスクリプトのロケーションを記述する。method オプションには″get″もしくは″post″を記述する。
以下に具体的な記述例を示す。

```
<html><body>
<form method="post" action="http://<ホスト>/<スプリプトパス>">
<input type="text" name="address">住所を記入してください。
<input type="submit" value="送信">
</form>
</body></html>
```

上記例において、action=″http://＜ホスト＞/＜スプリプトパス＞″の部分がＮＡＬスクリプトのロケーションを示し、name=″address″がクエリーの key として″address″を指定している。
ユーザがブラウザに表示されるテキスト枠に住所を xxxxxx と記入し、「送信」ボタンをクリックした時点で、ブラウザからサーバに対し、″address=xxxxxx″というクエリー文字列が送付される。
なお、xxxxxxx の部分はユーザが記入した文字列に依存した文字列データとなる。

1.1.3クエリーのデコード

ＦＯＲＭを使用する方法では、クエリーとして生成される″key=value″の関係が、実際に記述した key とも、また value に該当する入力値とも異なる場合がある。
これは、クエリー送信において「url-encode」という規約に従った文字列変換をおこなうためである。
ＮＡＬにおいてはクエリーを、Query という変数の値として取り出すことができ、このままでは url-encode された文字列であるが、
URL.decode(Query)という操作によって、デコードがおこなわれた結果の文字列として取り出すことができる。

1.2 ドキュメント出力

サーバからクラアントへの結果は、ドキュメント出力によっておこなわれる。
ＮＡＬにおいては、固定文字列による静的な書き出し、プログラムによる動的な書き出しがある。
固定文字列による静的な書き出しは pass 文が該当し、動的な書き出しにおいては、write()関数などによる方法と、タグ文の動的生成がある。

1.2.1pass 文による書き出し

pass 文は、スクリプト内の、演算子“%%”によって括られた範囲の文字列を無条件にドキュメントとして書き出すものであり、ドキュメントページを構成する文字列（タグを含む）が広範囲にわたって固定している場合などに使用する。
ＷＥＢアプリケーション開発では多くの場合、ホームページビルダなどのツールを使用して、代表的なドキュメントイメージを静的な手法で作成しておいてから、スクリプトを埋め込んで結果的にダイナミックなドキュメントページを構成する手法を用いる。
こうした場合、スクリプトを記述する部分が少ないとか、まとまった文字列を高速に出力する目的で pass 文を使用する。
以下に、pass 文のみで記述するＮＡＬスクリプトの例を示す。

```
%%<html><body>
こんにちは、このページはＮＡＬが生成しています。<br>
</body></html>%%
```

この例では、前後の“%%”を除いて、ファイル拡張子をそのまま“.html”に変えれば結果は同じである。
逆に、全てのＨＴＭＬドキュメントファイル（拡張子が“.html”のもの）は、拡張子を“.nal”に変更し、かつ記述の最初と最後に“%%”を記述すればＮＡＬスクリプトとなる。

pass 文を記述する上で注意すべきは、“%%”以降、“%%”までのテキスト（スクリプト内に記述された文字列）は改行コードを含めすべて書き出されるが、例外として各行頭にある連続するタブコードについては書き出しをおこなわない点である。
特に、“<pre>”と“</pre>”で囲まれた範囲の文字列（タグを含む）において、タブコードが出力されることを期待して記述するような場合は、意図した結果とならないことになる。

1.2.2write()関数による書き出し

write()文による方法は、可変要素をもとにドキュメントを書き出す一般的な方法であり、いわゆる「プログラムらしさ」を実感できる方法といえる。
以下に例を示す。

```
var s = 0;
for(var c = 1; c < 10; c++){
    s += c;
    write("Σ(1,%{ c }%) = %{ s }%<br>");
}
```

1.2.3式評価出力演算子による書き出し

「%{ 式 }%」という形式で記述することにより、<式>の値をそのままドキュメントに出力する。
以下に示す例を、先の write()文による方法と比較されたい。

```
var s = 0;
for(var c = 0; c < 10; c++){
    s += c;
    %{"Σ(1,%{ c }%) = %{ s }%<br>"}%;
}
```

式評価出力演算子による記述は、若干の手間の削減と、見通しのよいプログラムとする上で効果的である。

1.2.4タグの動的生成による書き出し

tag 文は“<”と“>”に括られた文字列（ＮＡＬにおいては内容を吟味しない）をＨＴＭＬタグもしくはＸＭＬタグその他マークアップランゲージとみなし、その範囲の文字列はそのままドキュメントとして出力する仕組みである。
ただし、tag 文内に“%{”と“}%”で括られた文字列がある場合、この文字列はＮＡＬの式として解釈し、その値を文字列化して埋め込み、タグを動的に生成することができる。
以下に例を示す。

```
const imgs = {"img1.gif","img2.gif","img3.gif"};
for(var id = 0; id < imgs.length; id++){
    <img src="%{ imgs[id] }%"><br>;
}
```

この例の出力結果は

```
<img src=”img1.gif”><br><img src=”img2.gif”><br><img src=”img3.gif”><br>
```

となることはいうまでもない。

1.3 プログラム記述スタイル

1.3.1文法的位置付けとプログラム記述

プログラマが、拡張子“.nal”としたＮＡＬスクリプトファイルを記述生成する場合、例えば以下のプログラムを書いたとする。

```
#!/usr/bin/nal
var s = 0;
for(var c = 1; c < 10; c++){
    s += c;
    write("Σ(1," , c , ") = " , s , "<br>");
}
```

もちろん、これはＮＡＬ文法として正しいし、またプログラマの意図した結果をクライアントのブラウザに表示する。
ＮＡＬにおいては記述したプログラムの前後に以下に示す暗黙の構文を追加して解釈している。
結果として、ＮＡＬ言語仕様におけるオブジェクト宣言文となり、記述されたプログラムはコンストラクタとして機能する。

```
object xxxxx(key1="value1" , key2="value2"){
    #!/usr/bin/nal
    var s = 0;
    for(var c = 1; c < 10; c++){
        s += c;
        write("Σ(1," , c , ") = " , s , "<br>");
    }
}
```

すなわち、“object xxxx(){”がプログラムの始まりであり、対応する“}”が終わりとなる。
ここで、xxxx はＮＡＬプログラムのファイル名（クライアントにとってはＵＲＬと同義）であり、ＮＡＬにおいてはオブジェクト名という。
クライアントがこのプログラムを呼び出すということは、すなわち object xxxx を生成して、その消滅までの**ふるまい**の結果をドキュメントとして得ることに他ならない。
ただ、ほとんどのプログラマは、このようなＮＡＬの論理的裏づけを意識することなくプログラミングできるはずであるが、この概念を理解しておけば更なる高度なシステム要求に対しても、どのようにプログラムを構築すればよいか、ひとつの指針となることは間違いない。

1.3.2クエリーの扱い

先に、クエリーを生成する過程で述べたように、クエリー文字列がＮＡＬプログラムに渡される。
しかしながら、単に文字列としてのクエリーを受けるのではなく、ＮＡＬはこれを「プリセット・フィールド」として引き渡す。
前節で、“object xxxxx(key1=”value1”,key2=”value2”){”が暗黙のうちに記述されたものとして扱う、と説明した。
実は、key1=value1,key2=value2 の部分がクエリー文字列から生成された実引数として、プログラム内からそのまま使用できるのである。

クライアントからのリクエストが、
“get http://nal.act.ne.jp/sample/query.nal?key1=value1&key2=value2”
であるとする。
実際のクエリーは“key1=value1&key2=value2”であり、これは２つのキー値、
key1=”value1”と key2=”value2”に分解される。
このキー値が各々プリセット・フィールドとなる。

上記の場合、プログラムにおいて
write(key1,key2)とした場合、結果が

```
value1value2
```

としてドキュメント出力される。
もちろん write(key1,”
”,key2)とすれば、

```
value1
value2
```

となる。
もし、クエリーとして届けられないフィールド変数 key3 を、write(key3)のように記述した場合、得られる結果は”null”となる。

プログラマは、プログラムが要求するプリセット・フィールドとクライアントから届けられるクエリーとの間で矛盾が生じないようにしなければならない。
また、クエリー中に、“key=value1&key=value2”のように、同一の key に対して複数の値を渡す場合、最後に渡される値が採用される点も忘れてはならない。

1.4 サーバ・プログラミングへのアプローチ

クライアントが要求するクエリーは、クライアントが勝手に送りつけてくる（サーバ側で予想つかないもの）ものではなく、あらかじめサーバ側が準備をしておくことが一般的である。

1.4.1静的ＨＴＭＬドキュメントによるＦＯＲＭ対応クエリーを調べる

一般的にＣＧＩとよばれる手法で対応するスタイルである。
クライアントにあらかじめ、サーバ・プログラムの要求するキー値の組み合わせをＦＯＲＭ部品として記述したＨＴＭＬファイルをアクセスさせ、ユーザの操作によって各キー値を記入させ、SUBMIT ボタンの操作によって、サーバに解析を依頼する。

ＨＴＭＬファイル、”calc.html”として以下を用意し、
http://nal.act.ne.jp/sample/calc.html として格納する。

```
<html><body>
<form method=get action=”http://nal.act.ne.jp/sample/calc.nal”>
<input type=”text” name=”数式”>
<input type=”submit” name=”func” value=”計算”>
</form>
</body></html>
```

ＮＡＬファイルとして”calc.nal”を以下の内容で記述し、

```
#!/usr/bin/nal
writeln(URL.decode(Query));
```

http://nal.act.ne.jp/sample/calc.nal に相当するディレクトリに格納する。
１行目の“#!/usr/bin/nal”という記述は、このＮＡＬスクリプトを解釈するためのＮＡＬエンジンへのパスを示している。
※Linux サーバの場合は、ファイルはＬＦのみの改行（ＣＲＬＦではない！）とし、かつこのファイルに実行属性（755）を与えておく必要がある。

ブラウザのＵＲＬ入力にて、http://nal.act.ne.jp/sample/calc.html を指定し、表示する。
テキスト入力欄に例えば“100+200”と入力し“計算”ボタンをクリックすると、ブラウザ画面には“数式=100+200&func=計算”と表示する。
すなわちブラウザは、<input type=”submit”>対応ボタンをクリックした時点で、フォーム部品各々の”name=”に指定した key と、”value=”で指定した値をクエリーとして送っていることがわかる。

1.4.2静的ＨＴＭＬをＮＡＬに取り込む

前節の方法では操作の継続性がなく、毎回ブラウザの「back」ボタンをクリックしなければならない。

そこで、今度は”calc.html”の内容を取り込んで、”calc.nal”を以下のように変更してみる。
※漢字を使用する場合、例え Linux サーバであっても必ずシフトＪＩＳ形式のテキストファイルでなければならない。

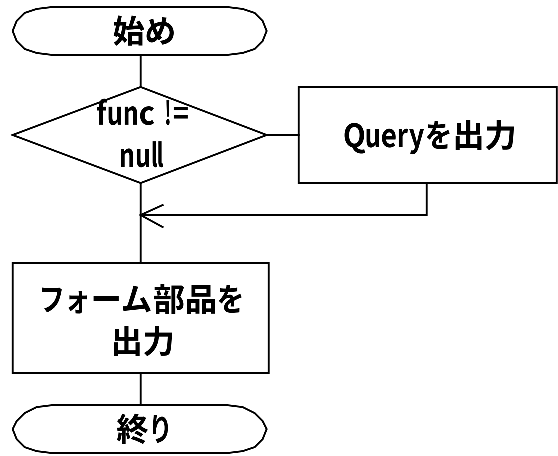
```
#!/usr/bin/nal

if(func != null){
        writeln(URL.decode(Query));
}
%%
<html><body>
<form method=get action="http://nal.act.ne.jp/sample/calc.nal">
<input type="text" name="数式">
<input type="submit" name="func" value="計算">
</form>
</body></html>
%%
```

としてから、ブラウザのＵＲＬ欄に”http://nal.act.ne.jp/sample/calc.nal”と入力してみる。

今度は、初期画面は同じであるが、テキスト入力欄に“100+200”を入力し“計算”ボタンをクリックするたびに先頭行に“数式=100+200&func=計算”と表示するはずである。
また、テキスト入力欄の内容を変更して“計算”ボタンをクリックするごとに処理を繰り返す対話型アプリケーションの形をとっている。

1.4.3ＮＡＬアプリケーションの動作イメージ

上に示したＮＡＬプログラムは、以下のフローチャートに相当する構成となっている。



クライアントのブラウザから“http://nal.act.ne.jp/sample/calc.nal”をＵＲＬ指定された場合、サーバはクエリー無しでこのプログラムを実行する。
つまり、“func==null”であるから“Query を出力”する部分は実行しない。
そして、“フォーム部品を出力”にて、出力されたＨＴＭＬドキュメントがクライアントに渡され、テキスト入力欄と、“計算”ボタンが表示される。

ユーザがテキスト入力欄に記入し、“計算”ボタンをクリックすれば、“func=計算”がクエリーとして渡される。したがって、この場合は“Query を出力”する。

ＮＡＬプログラムではクエリー構成要素としてのキー値（key=値）がステートメントとして解釈さ

れる。
すなわち、“数式=100+200&func=計算”というクエリーは

```
数式=”100+200”;
func=”計算”;
```

という二つの代入式として解釈される。

ここでキー値の値部分が文字列値となっていることに注意が必要である。
ブラウザは、すべてのクエリーを文字列として送り出すため、ＮＡＬでもキー値は文字列であるとして扱う。
もし、キー値が整数など、文字列でないことがわかっている、もしくはプログラム上文字列としてではなく整数などとして扱う必要がある場合は、“数式=eval(数式);”もしくは“数式=#数式;”のように記述する必要がある。両方とも、文字列をステートメントとして解釈するための手続きであるため、変数“数式”の内容が”100+200”という文字列であればこれを、100+200 という式として解釈し、結果は 300 が格納される。

1.4.4値の継承

ほとんどのアプリケーションでは、ユーザ操作の過程で値の決まる変数などを、次の操作への引数とか、またはその後の処理に使用するために保存する必要がある。
ＷＥＢアプリケーションでは、基本的にオンデマンド・バッチ処理（依頼された時点で処理を完結する）であるため、基本的な仕組みの範囲では変数の値を定常的に保持することができない。
そこで、以下の手法を使用して値を継承する。

フォーム部品中に“<input type=”hidden” name=xxxx value=yyyy>”がある。
これはブラウザ画面上には何も表示されないが、“value=yyyy”の記述が、yyyy 文字列として次のクエリーに反映されるという機能を持っている。
すなわち、中間値もしくは継承すべき変数の値を、この仕組みを使って以後の処理に引き渡すことができる。
また。各々のフォーム部品自体にも初期値を与えることができるため、ユーザ操作によって値を変えていくような場合に応用することもできる。

以下に“数式”という変数値を引き継ぐ例を示す。
“var ?数式=””;”は初期値リカバリとしての記述である。すなわちクエリー中に“数式”というキーがない場合には“数式”という変数を空文字列として定義する、という意味である。

```
#!/usr/bin.nal
var	?数式="";
if(func != null){
	数式 = #数式;
}
%%
<html><body>
<form method=post action="http://nal.act.ne.jp/sample/calc.nal">
<input type="text" name="数式" value="%{ 数式 }%">
<input type="submit" name="func" value="計算">
</form>
</body></html>
%%
```

2 サーバ・プログラミングの実際

サンプルとして、コメント追記可能な電子掲示板アプリケーションを取り上げる。

2.1 仕様の決定

ユーザがＵＲＬで示されたＮＡＬプログラムをアクセスすると、現在登録されている掲示データを一覧表示する。
登録可能な掲示は高々２０件とし、２０件を超える場合は登録の古いものから削除する。
掲示登録する場合の入力データは、タイトル、掲示メッセージ、記入者メールアドレスとする。
登録日付時刻をＵＴＣ１９７０年１月１日０時０分０秒からの経過秒にて記録し、登録順序はこれをもとに判断する。

2.2 画面のデザイン

仕様にしたがって、ＨＴＭＬエディタなどのツールによって基本画面を作成する。

2.2.1リスト画面

掲示板リスト画面を表示する基本的なＨＴＭＬファイルとその表示を以下に示す。なお、デザインなどについては考慮していない。
以下の内容のＨＴＭＬファイルを“disp.html”とする。

```
<html>
<body>
<h2>掲示板</h2>
<dl>
<dt><li>タイトル１
<dd>メッセージ１
<dt><li>タイトル２
<dd>メッセージ２
</dl>
<hr>
<a href="newpage.html">新規登録</a>
</body>
</html>
```

掲示板

- タイトル1
  メッセージ1
- タイトル2
  メッセージ2

新規登録

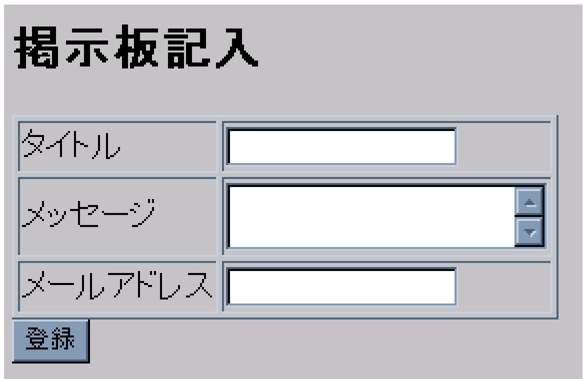
上記のＨＴＭＬにおいて、青字で示す<dt>
と<dd>に続く**タイトル**ｎと**メッセージ**ｎが掲示レコード数およびその内容による可変部分となり、これはあとで調整する。

2.2.2入力画面

以下の内容のＨＴＭＬファイルを“input.html”としてしておく

```
<html>
```

```
<body>
<h2>掲示板記入</h2>
<form method="post" action="board.nal">
<table border="1">
<tr><td>タイトル</td><td><input type="text" name="title"></td></tr>
<tr><td>メッセージ</td><td><textarea name="message"></textarea></td></tr>
<tr><td>メールアドレス</td><td><input type="text" name="address"></td></tr>
</table>
<input type="submit" name="cmd" value="登録">
</form>
</body>
</html>
```



2.3 ＮＡＬプログラムとしてまとめる

画面デザインとして作成したＨＴＭＬファイルをＮＡＬプログラムとしてまとめる。
各々の画面を表示するＨＴＭＬドキュメントを生成するメソッドとして構成し、ユーザ操作に応じて呼び出すメソッドを切り替える方法で実現する。
ただし、データベースに登録されている複数の掲示データを順に取りだしてタグ出力する部分は、メソッド（lists()）を呼び出すこととし、その具体的な内容については後で記述することとする。
したがって、先の“disp.html”を以下のように書き換えて、可変分を lists() メソッド呼び出しとして記述しておく。
また、「新規登録」のリンクをクリックした際には、″board.nal″にクエリーとして″cmd=新規登録″を送り、その解釈の結果として″input.html″が示されるとしておく。

```
<html>
<body>
<h2>掲示板</h2>
<dl>
%% lists(); %%
</dl>
<hr>
<a href="board.nal?cmd=新規登録">新規登録</a>
</body>
</html>
```

以下にＮＡＬプログラムの例を示す。プログラム名は“board.nal”とする。
これはまだ骨組みにしかすぎない。

```
#!/usr/bin/nal

/*
* 記入された内容をデータベースに登録し、一覧表示する。
```

```
*/
method  掲示登録(){
        /* この部分の実際の登録処理は後で記述します */
        掲示リスト();                          // 結果を一覧表示する
}

/*
* 掲示を一覧表示する
*/
method  掲示リスト(){
        /*
        * データベースからレコードを取り出して
        * <dl></dl>の組によってＨＴＭＬ化し出力する
        */
        method  lists(){
        /* この部分の実際の表示処理は後で記述します */
        }
        include "disp.html";    // 表示用ＨＴＭＬファイルを取り込む
}

/*
* 新たな掲示を記入し登録する
*/
method  掲示記入(){
        include "input.html";  // 入力用ＨＴＭＬファイルを取り込む
}

/*
* メイン
*/
select(cmd){
  case  "登録":          掲示登録();
  case  "新規登録":      掲示記入();
  default:               掲示リスト();
}
```

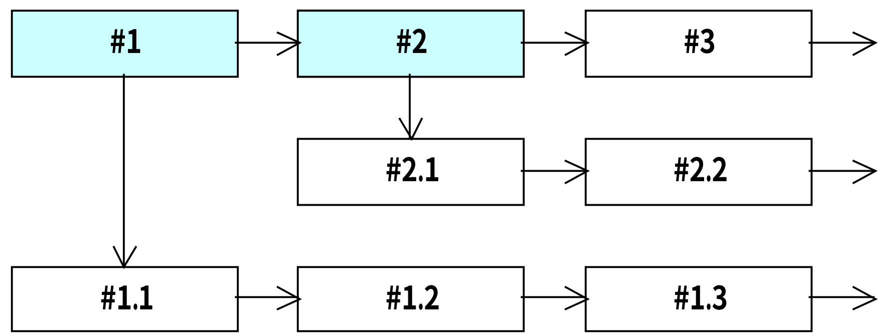
この段階で”http://＜ロケーション＞/board.nal”をアクセスして、画面遷移を確認しておくことを薦める。
ここで＜ロケーション＞はサーバ上で”board.nal”を配置したＷＥＢパスである。

2.4 データベースの使い方

掲示板に記入された個々の掲示を記録保存するためにデータベースを使うことを考える。
ＮＡＬには、自由度が高くかつプログラムの容易なデータベースエンジンを内部に、組み込みクラス（DBMAN）として持っている。
掲示板の用途であればこれで十分であるため、これを使用する。

2.4.1データベースの構成要素

ＮＡＬのデータベースエンジンは、そのレコード内容としてもＮＡＬのデータ型を踏襲する。
また各レコードは階層化順編成レコードとして管理される。
以下にイメージを示す。



各レコードは階層ごとにレコード番号およびキー（６３バイト以内の文字列）によってアクセスされ、あるレコードに属する下位階層レコードにおいても同じ構成となっている。

各々のレコードは下位層レコード群を持つもの（上記#1、#2）をフォルダレコードといい、下位層レコード群を持たないものをデータレコードという。
フォルダレコードとデータレコードの違いは下位層レコード群を持つか否かだけであり、すべてのデータレコードは下位層レコードを生成した時点でフォルダレコードとして、またすべてのフォルダレコードは下位層レコード群を削除した時点でデータレコードとなる。
データレコードであるかフォルダレコードであるかの判定は、count()メソッドの結果が 0 でないならフォルダレコードである。

レコードのアクセスはＤＢＭＡＮオブジェクトもしくはレコード・オブジェクトで管理され、その扱い方は同じである。
ＤＢＭＡＮオブジェクトは、ＤＢＭＡＮクラスの new によるコンストラクタで生成するオブジェクトであり、第１階層レコードを扱うためのメソッドのみ用意している。
レコードオブジェクトは、ＤＢＭＡＮオブジェクトもしくはレコードオブジェクト自身のもつメソッドの結果として生成され、レコードに関するすべてのメソッドを用意している。

2.4.2データベースの設計

仕様により、１レコードには以下の項目要素を含むことがわかる。
各々のデータ内容（規定）もあわせて記述する。

項目	内容	備考
登録日付時刻	UTC1970/1/1 0:0:0 からの経過秒	
タイトル	シフトＪＩＳ文字列	
メッセージ	シフトＪＩＳ文字列	
メールアドレス	半角英数および記号によるメールアドレス	

したがって、この内容をハッシュとしてレコードに記録する。
ハッシュのフィールド名も上表の項目名をそのまま使うことにする。
データベースファイル名は、“board.nlx”とする。
このデータベースファイルは、存在しなければ新規に作成し、存在すれば読み書きモードで排他ロックしてアクセスする。
データベースを開いたら、グローバル変数“db”に割り付けた DBMAN オブジェクトとしてアクセスするようにする。
具体的な記述は、 `var db = new DBMAN("board.nlx",DBMAN.CREATE);` となる。
もちろん終了時には `db.close();` によってデータベースを閉じておく。
※もしdb.close()を記述忘れした場合でもDBMANオブジェクトのデストラクタによって確実にデータベースを閉じるため、データが更新されないなどのトラブルは発生しない。

2.4.3データの格納

新しい掲示を登録する場合、データベース内の第１階層レコードに追加登録する。
仕様では、新たなレコードが先頭になるように登録し、またレコード総数が２０件を超える場合は最後のレコードを削除する。
登録するデータはハッシュであり、その内容は前もって構築されていることとする。
ハッシュの構築は以下のステートメントにて実現する。

```
var     掲示 = {
        登録日付時刻:   (new Date().getTime()/1000).toIntValue(),
        タイトル:       title,
        メッセージ:     message,
        メールアドレス: address
        };
```

登録日付時刻フィールドについては、現在時刻の UTC 基準時相対経過ミリ秒を 1000 で割って秒単位にし、これを整数にした値とする。
その他のフィールドは、登録者がフォームに記述した文字列をそのまま使用する。

こうして生成したハッシュを、

```
db.newRecord(掲示.登録日付時刻+””).put(掲示);
```

にて登録する。
ここで、newRecord()の引数はインデックスキーであり、登録時刻を文字列化した値を使用することを示している。（実際には使用しない）
ただし、レコード数の上限は２０件となっているので、レコード数が２０件ある場合は最後のレコードを削除してから登録するようにし、実際は以下のようにしなければならない。

```
while(db.count() >= 20) db.remove(db.count()-1);
db.newRecord(掲示.登録日付時刻+””).put(掲示);
```

2.4.4データの取り出し

掲示データを登録するごとに、登録レコード数は増加する。
現在の第１階層レコード数は db.count() メソッドの結果として得られる。
したがって、lists()として実現するメソッドの内容は、

```
method  lists(){
        <dl>
        for(var id = 0; id < db.length(); id++){
                var     recdata = db.get(id);
                <dt><li>%{ recdata.タイトル }%;
                <dd>%{ recdata.メッセージ }%;
        }
        </dl>
}
```

となる。

ここまでの”board.nal”プログラムを整理しておく（以下）

```
#!/usr/bin/nal

/*
* 記入された内容をデータベースに登録し、一覧表示する。
*/
method  掲示登録(){
var     掲示 = {
        登録日付時刻:   (new Date().getTime()/1000).toIntValue(),
```

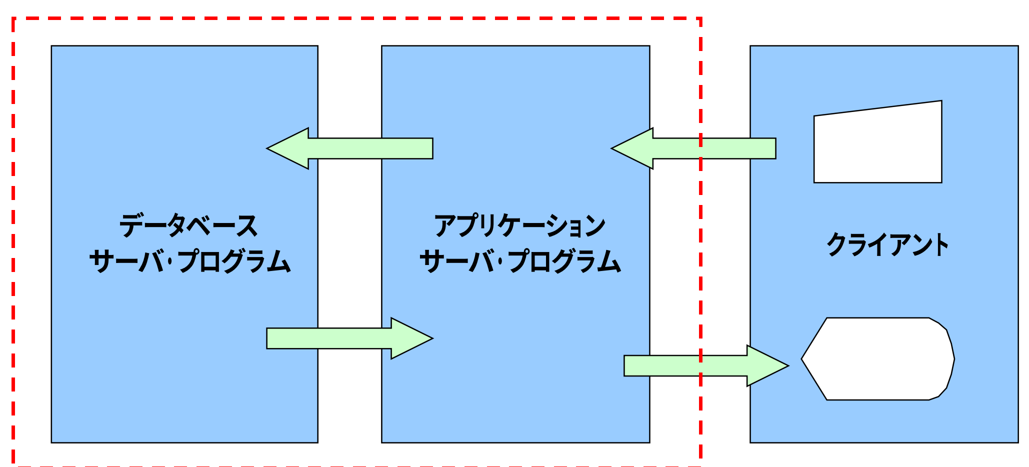
```
        タイトル:       title,
        メッセージ:     message,
        メールアドレス: address
        };
        while(db.length() >= 20) db.remove(db.length()-1);
        db.newRecord(掲示.登録日付時刻+"").put(掲示);

        掲示リスト();                       // 結果を一覧表示する
}

/*
* 掲示を一覧表示する
*/
method  掲示リスト(){
        /*
        * データベースからレコードを順次取り出して
        * <dl></dl>の組によってＨＴＭＬ化し出力する
        */
        method  lists(){
                <dl>
                for(var id = 0; id < db.count(); id++){
                        var     recdata = db.get(id);
                        <dt><li>%{ recdata.タイトル }%;
                        <dd>%{ recdata.メッセージ }%;
                }
                </dl>
        }
        include "disp.html";
}

/*
* 新たな掲示を記入し登録する
*/
method  掲示記入(){
        include "input.html";
}

/*
* メイン
*/
var     ?cmd = "";
var     db = new DBMAN("board.nlx",DBMAN.CREATE);
select(cmd){
  case  "登録":         掲示登録();
  case  "新規登録":     掲示記入();
  default:              掲示リスト();
}
db.close();
```

強調部分が、上記にて解説して追加した部分である。

2.5 サーバ連携

アプリケーションを複数のサーバによる連携として構築する場合を考える。
例えば、フロントエンド・サーバが操作受付を担当し、バックエンド・サーバがデータベースを担当するように構成する場合など、多くの場合負荷分散などを目的としたり、共通するデータベースを１ヶ所にまとめる場合などに用いられる。

2.5.1サーバ連携の概念



クライアントから見た場合、直接認識するサーバは１台だけであり、アプリケーションが複数台のサーバの連携によって構成されていたとしても、またその構成が随時変更されたとしても何ら影響も受けないし、また認識もされない。

アプリケーション・サーバは、アプリケーションを実行する場合、その処理の一部を他のサーバにゆだねることができる。すなわち、依頼するサーバを明確に管理していれば、その先のサーバでどのような処理がおこなわれるかについては関知する必要がない。
上図において、アプリケーション・サーバはクライアントにとってサーバではあるが、また一方でデータベース・サーバにとってはクライアントである。

2.5.2サーバ連携の実際

上記プログラムを２台のサーバで連携する場合を例に説明する。
サーバA（アプリケーション・サーバ）はクライアントからの受付のみを担当し、データベースアクセスはサーバB（データベース・サーバ）が担当するとする。
サーバAはサーバBの提供するクラスメソッドを利用することにする。
サーバBの提供するクラス（db クラスとする）のＵＲＬを
http://exec.act-is.co.jp/sample/boarddb.nal とする。

具体的なプログラムは以下のようになる。

サーバAのプログラム(“board.nal”)

```
#!/usr/bin/nal

/*
* データベースサーバ（サーバB）のクラスメソッドを使うための宣言
*/
extern  class   db():"http://exec.act-is.co.jp/sample/boarddb.nal";

/*
* 記入された内容をデータベースに登録し、一覧表示する。
*/
```

```
method  掲示登録(){
var     掲示 = {
        登録日付時刻:  (new Date().getTime()/1000).toIntValue(),
        タイトル:      title,
        メッセージ:    message,
        メールアドレス: address
        };
        db.add(掲示);
        掲示リスト();                     // 結果を一覧表示する
}

/*
* 掲示を一覧表示する
*/
method  掲示リスト(){
        /*
        * データベースからレコードを一括して取出し
        * <dl></dl>の組によってＨＴＭＬ化し出力する
        */
        method  lists(){
                var     data_array = db.getData();
                <dl>
                for(var id = 0; id < data_array.length(); id++){
                        var     recdata = data_array[id];
                        <dt><li>%{ recdata.タイトル }%;
                        <dd>%{ recdata.メッセージ }%;
                }
                </dl>
        }
        include "disp.html";
}

/*
* 新たな掲示を記入し登録する
*/
method  掲示記入(){
        include "input.html";
}

/*
* メイン
*/
select(cmd){
  case  "登録":         掲示登録();
  case  "新規登録":     掲示記入();
  default:              掲示リスト();
}
```

サーバＢのプログラム（“boarddb.nal”）

```
#!/usr/bin/nal

static  field   db = new DBMAN("board.nlx",DBMAN.CREATE);
```

```
/*
* データベースに新たなレコードを追加する
*/
method  add(掲示){
        while(db.length() >= 20) db.remove(db.length()-1);
        db.newRecord(掲示.登録日付時刻+"").put(掲示);
        db.close();
}

/*
* データベースの全てのデータを取り出す
*/
method  getData(){
        var     result = {};                    // 空の配列を用意
        for(var id = 0; id < db.length(); id++){
                result += db.get(id);
        }
        db.close();
        return  result;
}
```

サーバＡのプログラムでは、自身でデータベースを操作していた部分を、ＵＲＬで示す外部サーバＢが提供するクラスメソッドにゆだねるための変更をおこなうだけでよい。
すなわち、外部サーバＢが提供する class db()[1]のメソッドである add()を呼び出すために、db.add(掲示);とする。
データの取り出しは、パフォーマンスの観点から一括して取り出すメソッドとして db.getData()を使用する。
このメソッドから返される値は、各々のレコードをハッシュとして取り出し、それを１次元配列とした内容である。

サーバＢのプログラムは、他のサーバに対してパッシブなクラスメソッドを提供する。
このプログラム自身は起動ごとにクラスオブジェクト[2]を生成し、実行を終了すればオブジェクトを破棄するように動作する。
ここでは、add()と getData()というメソッドを呼び出されることを想定している。
利用者（サーバＡ）が、メソッド add()を呼び出す場合、先にクラス構築がなされ、その後でメソッドが呼び出される仕組みは、通常のオブジェクト生成の仕組みと変るところはない。
もちろん、クラス構築部分でデータベースを開いている処理をメソッド内で実行するようにしても問題がないし、またそのほうが見通しの良いプログラムになるのであれば推奨する。
例では、メソッドごとに db.close()を呼び出してデータベースを閉じているが、この記述自体は必要としない。なぜなら、new DBMAN()によって作り出されたオブジェクトは、そのオブジェクト自身のデストラクタ内で close()をおこなっているからである。

なお、サーバ連携に伴なう変更はサーバプログラムに限った変更であり、クライアントに対しては何ら影響を及ぼさない点は理解できると思う。

---

[1] クラス名は利用する側で自由に既定して構わない。
[2] クラスオブジェクトとは、new によるダイナミックオブジェクトを生成する機能を持たず、単独でオブジェクトとして振舞うものをいう。

2.6 多重登録の抑制

ブラウザの「更新」ボタンを押した場合に、同じフォームを再度送信してしまう特性によってデータベースに多重に登録されたり、また意図しないデータが削除されたりすることを防止する手法について述べる。
前節までに紹介したプログラムを使って、データを「登録」した直後にブラウザの「更新」ボタンを押すと、押した回数だけ同じデータが登録されることに気づく。

これはブラウザの特性上止むを得ない仕組みではあるが、実際問題としては具合がよくない。
そこで、登録フォーム画面ごとに１意的な識別を設け、データベースに登録する際にこれを検査して同じ識別があれば「再送」による２重登録であると判断するようにすればよい。

識別をつける手軽な方法は Math.random()関数による結果を用いる方法がある。
すなわち、hidden フォーム部品に Math.random()の値を埋め込んでおき、この識別をレコード識別として利用する。現在時刻値（秒単位）を利用する方法も考えられるが、不特定多数からの利用が前提となる場合は、同じ時刻に複数ユーザからアクセスされる可能性があることと、サーバの時計を修正した場合に問題となることを認識しなければならない。
対策を施したＨＴＭＬファイルとＮＡＬプログラムを以下に示す。

input.html

```
<html><body><h2>掲示板記入</h2>
<form method="post" action="board.nal">
<!--画面識別 -->
<input type="hidden" name="ident" value="%{ Math.random() }%">
<table border=1>
<tr><td>タイトル</td><td>
<input type="text" name="title"></td></tr>
<tr><td>メッセージ</td><td>
<textarea name="message"></textarea></td></tr>
<tr><td>メールアドレス</td><td>
<input type="text" name="address"></td></tr>
</table>
<input type=”submit” name=”cmd” value="登録">
</form></body></html>
```

board.nal

```
#!/usr/bin/nal

/*
* 記入された内容をデータベースに登録し、一覧表示する。
*/
method  掲示登録(){
var     掲示 = {
        登録日付時刻:   (new Date().getTime()/1000).toIntValue(),
        識別:           ident,          // 多重登録防止用レコード識別
        タイトル:       title,
        メッセージ:     message,
        メールアドレス: address
        };
        /*
        * 同じ識別をもつレコードが存在すれば多重登録とみなす
        */
        var     multi = false;
```

```
        for(var id = 0; id < db.length(); id++){
                if(db.get(id).識別 == ident){
                        multi = true; break;
                }
        }

        if(!multi){                     // 多重登録なら無視する
                while(db.length() >= 20) db.remove(db.length()-1);
                db.newRecord(掲示.登録日付時刻+"").put(掲示);
        }
        掲示リスト();                   // 結果を一覧表示する
}

/*
* 掲示を一覧表示する
*/
method  掲示リスト(){
        /*
        * データベースからレコードを順次取り出して
        * <dl></dl>の組によってＨＴＭＬ化し出力する
        */
        method  lists(){
                <dl>
                for(var id = 0; id < db.length(); id++){
                        var     recdata = db.get(id);
                        <dt><li>%{ recdata.タイトル }%;
                        <dd>%{ recdata.メッセージ }%;
                }
                </dl>
        }
        include "disp.html";
}

/*
* 新たな掲示を記入し登録する
*/
method  掲示記入(){
        include "input.html";
}

/*
* メイン
*/
var     db = new DBMAN("board.nlx",DBMAN.CREATE);
select(cmd){
  case  "登録":         掲示登録();
  case  "新規登録":     掲示記入();
  default:              掲示リスト();
}
db.close();
```