Microsoft

Web 開発 ガイドライン

～ ASP.NET プログラミング エッセンシャル ～

第 1 版 2010/04

マイクロソフト株式会社

目次

Web 開発の概要

私たちは日常的にインターネットを利用したさまざまなサービスを利用しています。

たとえば、どこかへ出かけたいときは電車の時刻表を検索し、週末の天気が知りたければいつでも調べることができます。また、お腹がすけばピザを注文し、書店へ足を運ぶ時間がなくても本の宅配を手配できるなど、1 日の中だけでもシチュエーションがたくさん挙げられます。

これらのインターネット上で動作するサービスを総称して、**Web アプリケーション**と呼びます。

Web アプリケーションは、基本的に Web サーバーとクライアント間のデータの送受信によって動作します。クライアントからの要求は、まずサーバー上のプログラムによって処理されます。サーバー側では、このとき、送信されたデータの登録や、指定されたキーワードなどの情報によってデータベースへの問い合わせなどを行うことになります。また、その結果から動的にコンテンツを生成し、クライアントに応答するのもサーバーの役割です。クライアント側では、Internet Explorer、Firefox のようなインターネット ブラウザーを利用してコンテンツを表示／実行します。

ここでは、このような情報のやりとりを行うために必要な Web 開発技術について、その概要を見ていきます。

クライアント サイド テクノロジ の概要

クライアント サイド テクノロジとは、名前の通り、クライアント上で動作するプログラムの総称です。複雑なビジネス ロジックの制御には不向きで、画面の制御や補助的なロジックの記述が主な用途です。

クライアント サイド テクノロジは、後述するサーバー サイド テクノロジと相互補完の関係にあります。たとえば、データ登録などの処理はサーバー サイド技術の役割ですが、その結果を視覚的な効果を伴いながらブラウザー上に表示させるのはクライアント サイド テクノロジの役割です。

クライアント サイド テクノロジは、主に **JavaScript** という言語を利用して実装します。

DHTML（Dynamic HTML）

DHTML（Dynamic HTML）とは、サーバー サイドから供給されたコンテンツを JavaScript などのスクリプトを使って動的に操作する技術のことです。DHTML を利用することで、たとえばエクスプローラーのように折りたたみ式のメニューや、データの入力時に誤りを指摘するというような対話的なしくみの実装が可能になります。JavaScript はこれらのしくみの実装に長い間利用されてきました。

AJAX

もっとも、DHTML による派手な視覚効果は、近年下火の傾向にありました。ブラウザー環境によって同じ挙動を保証しにくい、JavaScript を利用したウィルス被害が多い、あるいは、過度なアニメーションがユーザーによって飽きられた、など、原因はさまざまでしょう。

このため、Web アプリケーションはサーバー サイド テクノロジが主体、クライアント サイド テクノロジは補助的なしくみ、というサーバー偏重の状態がしばらく続いてきました。しかし、**AJAX 技術**の登場によって、その状況が大きく変わりつつあります。

AJAX は Asynchronous JavaScript + XML の略で、2005 年、Jesse James Garrett 氏によって名付けられました。AJAX は、JavaScript を利用してサーバー側と**非同期通信**を行い、受け取った結果を DHTML（DOM）などの技術を使ってページに反映するしくみです。AJAX 技術を利用することで、サーバー側との通信に際して画面全体をリフレッシュする必要がなくなり、スムーズな画面の更新が可能になります。より Windows アプリケーションに近い使い勝手を提供するための技術、と言い換えてもよいでしょう。

AJAX 技術は登場以来、急速に浸透しました。その結果、JavaScript の技術的な位置付けが大きく変わり、クライアント サイド テクノロジがユーザビリティを向上する技術として、再度見直されることになったのです。

RIA

RIA（Rich Internet Application）は、HTML だけでは表現できないリッチなユーザー インターフェイスを実現するための技術の総称です。AJAX も RIA 技術の一種と言えますが、その他にも、マイクロソフトの WPF、Silverlight、アドビ システムの Flash、Adobe Flex、Adobe AIR、サン マイクロシステムズ（開発時）の JavaFX などがあります。各技術の方向付けは 3 社それぞれです。マイクロソフトは既存の Windows アプリケーションに対しての RIA を目指し、アドビ システムはデザイナーによる利用に焦点をあて、サンマイクロシステムズは Java 資産の活用を狙っていると言えるでしょう。

WPF（Windows Presentation Foundation）にはスタンド アロンで実行可能なデスクトップアプリケーションである **WPF アプリケーション**と、Web ブラウザー上で動作する **WPF ブラウザー アプリケーション（XBAP : XAML Browser Application）**があります。

Silverlight は Web ブラウザーのプラグインです。Silverlight は当初、WPF のサブセットとして公開されました。WPF が .NET Framework 環境を前提とするのに対して、Silverlight はプラットフォームを選ばないという特長があります。開発言語としては、いずれも Visual Basic、C# などを利用できます。

Flash も、Silverlight と同じく、Web ブラウザーのプラグインです。Flash Player をインストールすることで、ベクタ グラフィックスのアニメーションや音声、音楽、効果音などを組み合わせた Web コンテンツを表現可能です。開発言語には ActionScript が用いられます。

Flex は Flash を基盤としたアプリケーション開発のためのフレームワークです。

Adobe AIR は、Flash をさらににデスクトップ環境で動作するためのフレームワークです。従来、ブラウザー上での動作に限定されてきた Flash の用途をより一層広げるための技術とも言えるでしょう。

サーバー サイド テクノロジ の概要

クライアント サイド テクノロジに対して、**サーバー サイド テクノロジ**は、サーバー サイドでプログラムを動作させる環境の総称です。サーバーの豊富な資源を活用できることから、複雑なビジネス ロジックの実行や、大量データの処理を得意とします。

初期のサーバー サイド テクノロジ - CGI -

サーバー サイド テクノロジの初期によく使われたのは、**Perl + CGI（Common Gateway Interface）**や **Java サーブレット**のような技術です。これらの技術では、プログラムからコンテンツを直接出力する方式を採用していることから、処理ロジックの記述に強い半面、出力結果がわかりづらく、開発生産性が低いという欠点がありました。

以下は、「Hello World」と出力させる、Perl＋CGI で書かれたプログラムです。

```
#!c:¥perl¥bin¥perl
print "Content-type: text/html¥n¥n";
print "<html>¥n";
print "<head>¥n<title>Hello world</title>¥n</head>¥n";
print "<body>¥n<h2>Hello world</h2>¥n";
print "</body>";
print "</html>"
```

中期のサーバー サイド テクノロジ - ASP／JSP／PHP -

CGI の欠点を解決するために登場したのが、**ASP（Active Server Pages）**や **JSP（JavaServer Pages）**、**PHP（Hypertext Preprocessor）**のような技術です。これらの技術では、HTML コードの内部にスクリプトを埋め込む、**HTML 埋め込み型**のスタイルを採用しています。これによって、レイアウト主体に動的なコードを記述できるようになり、出力結果をイメージしながら開発を進めることが可能になりました。

以下に、ASP、JSP、PHP によるプログラムの例を挙げてみましょう。

（1）ASP

以下は、ASP で書かれたプログラム例です。

```
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <% Response.Write("Hello World") %>
  </body>
</html>
```

ASP はコンテンツの動的な生成を実現するテクノロジの 1 つです。ASP の特長は以下の通りです。

- マイクロソフトの Web サーバーである Internet Information Services (IIS) で実行可能
- HTML の中へプログラムの埋め込みが可能
- JavaScript や VBScript で記述する

（2）JSP

以下は、JSP で書かれたプログラム例です。

```
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <%
      String s= "Hello World";
      out.println(s);
    %>
  </body>
</html>
```

JSP の最初の仕様は、サン マイクロシステムズによって 1998 年に発表されました。JSP の特長は以下の通りです。

- さまざまなプラットフォームで実行可能
- HTML の中へプログラムの埋め込みが可能
- Java 文法での記述が可能

Java アプリケーションの実行には仮想マシン、コンパイラやデバッガなどといった必要最低限のツールが提供されている **J2SE（Java 2 Standard Edition）**が必要です。また、この J2SE 上で動作するアプリケーションフレームワークとして、**J2EE（Java 2 Enterprise Edition）**があります。JSP & サーブレットは、この J2EE の中の一部として位置付けられます。

（3）PHP

以下は、PHP で書かれたプログラム例です。

```
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <?php
      echo 'Hello World!';
    ?>
  </body>
</html>
```

PHP は 1995 年に Rasmus Lerdorf 氏によって開発されたオープンソースのサーバー サイド スクリプト言語です。2004 年 7 月には PHP 5 が公開されました。PHP のプログラムはその Web サーバーにアクセス可能な Web クライアントさえあれば実行可能です。PHP の開発は C 言語 をベースに作られており、言語構造は簡単で理解しやすいことが特長です。

また、PHP は最近注目を集めている **LL (Lightweight Language)** と呼ばれる軽量言語の中のひとつでもあります。LL はインタプリタ言語で、変数の型が厳格ではなく、初学者でも簡単に利用できるのが特長です。他に代表的なものとして、Perl、Ruby、Python などがあります。

サーバー サイド テクノロジの現在

もっとも、HTML 埋め込み型のスタイルも、サーバー サイドで行うべき処理が複雑になるにつれ、限界が指摘されることも多くなってきました。ビジネス ロジックとレイアウトが複雑にからみ合うことで、プログラムのメンテナンスが難しくなってきたからです。

そこで最近では、ロジックとレイアウトを明確に分離するために、さまざまな枠組み（**フレームワーク**）を利用する機会が多くなっています。

Java 環境であれば Struts や JSF、PHP であれば symphony や cakePHP、Zend Framework などがそれです。ASP の後継である **ASP.NET** は、そうしたフレームワークの代表格と言えるでしょう。

ASP.NET では、**コード ビハインド（分離コード）**というしくみによって、デザインとコードとを明確に分離し、開発生産性とアプリケーションの保守性を大きく向上させています。また、サーバー コントロールに基づくイベント駆動型のプログラミング モデルを採用し、Windows アプリケーションを開発するのと同じ要領でアプリケーションを開発できるのも特長のひとつです。

たとえば、ボタンをクリックすると画面に「Hello World」と表示されるようにするには、以下のようなコードを記述します。

＜デザイン部分（*Hello.aspx*）＞

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeBehind="Default.aspx.cs" Inherits="HelloWorld._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <form id="form1" runat="server">
      <div>Hello World</div>
      <p>
        <asp:Button id="Button1" runat="server"
                    onclick="Button1_Click" Text="Button" />
        <asp:Label id="Label1" runat="server" Text="Label">
        </asp:Label>
      </p>
    </form>
  </body>
</html>
```

＜処理コード（*Hello.apsx.cs*）※*C#* を使用した場合＞

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace HelloWorld
{
  public partial class _Default : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
    }
    protected void Button1_Click(object sender, EventArgs e)
    {
      Label1.Text = "Hello World";
    }
  }
}
```

もっとも、いつも ASP.NET が最適の選択であるというわけではありません。たとえば、ASP.NET のようなユーザー インターフェイス（UI）中心の開発では、処理ロジック単体のテストが難しいという問題もあります。

これらの問題を解決するために、最近では **ASP.NET MVC** というフレームワークも提供されています。ASP.NET MVC は、MVC（Model-View-Controller）と呼ばれるデザインパターンに則って設計されたフレームワークです。MVC は、Java、PHP などの環境でも採用されているスタンダードなアーキテクチャのひとつです。ASP.NET MVC では UI と処理ロジックとの関係を薄くすることで、テストの自動化を容易にしています。

このように、なにが最適とは一概には言えませんが、今後はより一層、その場そのときに応じたフレームワークの選定が重要になっていくでしょう。

クライアント サイド テクノロジ

はじめに

Web アプリケーションの構成技術は、サーバー側で動作する**サーバー サイド テクノロジ**と、クライアント（ブラウザー）側で動作する**クライアント サイド テクノロジ**のふたつに大別されます。

クライアント サイド テクノロジは、必要なコンテンツとスクリプト、その他リソースをダウンロードした上で、クライアント側でプログラムを実行します。たとえば、マウス ポインタが重なると画像が入れ替わるようにしたり、フォームの入力値をチェックしたりするような処理を実装するのも、クライアント サイド テクノロジの役割です。**AJAX (Asynchronous JavaScript + XML)** の普及や **RIA (Rich Internet Applications)** への注目度が増すにつれ、最近ではクライアント サイド テクノロジの比重が高まりつつあります。

一般的な Web アプリケーションはクライアント サイドとサーバー サイドのテクノロジを組み合わせて構築されます。たとえば、ログイン画面を考えてみましょう。ユーザーはログイン ID とパスワードを入力し、ログイン ボタンをクリックします。このとき、クライアント側で未入力項目や桁数といった一次的な検証を行った後、サーバー サイドへデータを送信します。サーバー側では最終的なデータの検証を行ってから、ユーザー／パスワードの照合（認証処理）を実行します。クライアント サイド テクノロジとサーバー サイド テクノロジとを適切に分業させることは、Web アプリケーションを設計する上で重要なポイントのひとつです。

ここでは、クライアント サイド 開発においてポイントとなるテクノロジについて、詳しく見ていきます。

ダイナミック HTML の概要

ダイナミック HTML (DHTML) は、Microsoft Internet Explorer 4.0 で最初に導入された一連の革新的機能です。

DHTML を使用すると、これまでは実現が困難だった効果をページに簡単に追加できます。たとえば、次のようなことが可能になります。

- 一定時間が経過するか、ユーザーがページを操作するまで、コンテンツを非表示にできます。
- ドキュメント内のテキストとイメージをアニメーション表示できます。それぞれの要素は、事前設定された経路やユーザーによって選択された経路に沿って、任意の初期位置から任意の最終位置まで個別に移動させることができます。

- 最新ニュースや株価情報などのデータを表示および自動更新するティッカーを埋め込むことができます。
- form を使用してユーザー入力をキャプチャし、その入力データを即時に処理して応答できます。

DHTML は静的ページの不足点を補います。DHTML を使用すると、インタラクティブ性のためにパフォーマンスを犠牲にすることなく、インターネットやイントラネット上に革新的な Web サイトを作成できます。DHTML は、作成したドキュメントに対するユーザーの認知度を高めるだけでなく、サーバーへの要求数を減らすことでサーバーのパフォーマンスを向上させます。

以下、DHTML とその使用法を詳しく説明していきます。

ドキュメント オブジェクト モデル (DOM)

DHTML は、それ自体はテクノロジではなく、HTML、カスケーディング スタイル シート (CSS)、およびスクリプト (JavaScript) という相互に関連および補完する 3 つのテクノロジを組み合わせて開発されたものです。**ドキュメント オブジェクト モデル (DOM)** と呼ばれるプログラミング モデルでは、スクリプトとコンポーネントから HTML と CSS の機能にアクセス可能にするために、ドキュメントのコンテンツはオブジェクトとして表現されます。

DOM API は DHTML の基盤であり、ドキュメント内のほぼすべての要素にアクセスしてこれらを操作することを可能にする構造化インターフェイスを提供します。

動的スタイル

動的スタイルは DHTML の主要な機能です。CSS を使用すると、要素を追加または削除することなく、ドキュメント内の要素の外観と書式設定をすばやく変更できます。これにより、ドキュメントのサイズを小さく保てると共に、ドキュメントを操作するスクリプトを高速化できます。

オブジェクト モデルは、プログラムを介してスタイルにアクセスすることを可能にします。このため、シンプルなスクリプトベースのプログラミングを使用して、個別の要素のインライン スタイルやスタイル ルールを変更できます。これらのスクリプトは **JavaScript** を使用して記述することができます。

インライン スタイルは、style 属性を使用して要素に適用された CSS スタイル割り当てです。これらのスタイルを参照および設定するには、個別の要素の style オブジェクトを取得します。たとえば、マウスでポイントされた見出しのテキストを強調表示するには、次の例で示すように、style オブジェクトを使用して、フォントを拡大したり、その色を変更したりできます。

```
<html>
<head>
  <title>Dynamic Styles</title>
  <script type="text/javascript" language="javascript">
```

```
    function doChanges(e) {
      e.style.color = "green";
      e.style.fontSize = "20px";
    }
  </script>
</head>
<body>
  <h3 onmouseover="doChanges(this)"
    style="color:black;font-size:18px">Welcome to Dynamic HTML!</h3>
</body>
</html>
```

上記の例では、次のことが示されています。

- **HTML 要素** —— この例では H3 タグがターゲット要素です。
- **インライン スタイル** —— この要素は、最初は 18px の黒色のフォントで表示されます。
- **イベント属性** —— **onmouseover** 属性では、この要素がマウスでポイントされたときに実行される処理を定義しています。
- **イベント ハンドラ** —— このイベントに応じて実行される関数は、ドキュメントの head 内で宣言されています。ターゲット要素を表す DHTML オブジェクトは、**this** ポインタを使用して関数パラメーターとして渡されます。
- **style オブジェクト** —— style オブジェクトには、この要素が定義されたときにインライン スタイル内で設定された情報が含まれています。色とフォント サイズを変更するために、この関数は、この要素の **color** プロパティと **fontSize** プロパティを変更します。ブラウザーでは、画面上のテキストが即時に更新されて、これらの新しい属性値が表示されます。

次の例では、スタイルを使用して、ユーザーがマウスをクリックするまでページの一部を非表示にするように設定しています。このスクリプトでは、this ポインタを使用してターゲット要素を渡す代わりに、id によって HTML 要素を呼び出しています。

```
<html>
  <head>
    <title>Dynamic Styles</title>
    <script type="text/javascript" language="javascript">
      function showMe() {
        MyHeading.style.color = "red";
        MyList.style.display = "";
```

```
    }
  </script>
 </head>
 <body onclick="showMe()">
  <h3 id="MyHeading">Welcome to Dynamic HTML!</h3>

  <p>Just click and see!</p>

  <ul id="MyList" style="display:none">
    <li>Change the color, size, and typeface of text</li>
    <li>Show and hide text</li>
    <li>And much, much more</li>
  </ul>

 </body>
</html>
```

上記の例では、次のことに注目してください。

- リストの **display** 属性は、none に初期設定されています。このため、リストは非表示になります。
- **onclick** イベント属性は、body 内で設定されています。このため、ユーザーがページのどの場所をクリックしても、このイベントがトリガされます。
- イベント ハンドラは、id によってターゲット要素を呼び出し、display プロパティの値をクリアします。その直後に、リストに続くコンテンツが変化して新しいテキストが表示されます。

動的コンテンツ

DHTML を使用すると、ページが読み込まれた後にページのコンテンツを変更できます。DHTML DOM は、HTML ドキュメント内のすべての要素へのアクセスを可能にします。これにより、要素を作成、挿入、および削除したり、個別要素内のテキストや属性を変更したりできます。

標準の DOM メソッド

DOM プログラミング モデルは、任意のドキュメント タイプのコンテンツ、構造、およびスタイルに動的にアクセスしてこれらを更新できるように設計されています。このために、DOM では、ドキュメント階層がノードのツリーとして表現されます。ノードは、階層内の位置に応じて、**子ノード**や、**兄弟ノード**、**親ノード**を持つことができます。

標準の DOM メソッドは、HTML の文字列の解析や解釈ではなく、ドキュメント自体のツリー構造に重点を置いています。次の表では、コンテンツを動的に作成および操作するために使用できるプロパティとメソッドの一部について説明しています。

表. コンテンツを動的に作成および操作するために使用できるプロパティとメソッド（一部）

メソッド	説明
createElement	指定されたタイプの新しい要素 (ノード) を作成します。
createTextNode	プレーン テキスト ノード (非 HTML) を作成します。
appendChild	ノードを最後の子として親要素に追加します。
insertBefore	ノードを親の子ノードとしてドキュメントに挿入します。
replaceChild	既存の子要素を新しい子要素に置換します。

上記のメソッドに加えて、多くのブラウザーは **innerHTML** プロパティを完全にサポートしています。このプロパティを使用すると、要素の開始タグと終了タグの間に配置された HTML にアクセスできます。innerHTML はどの標準でも定義されていませんが、多くの場合は、その代替手法よりも高速で簡単に使用できます。

```
<html>
  <head>
    <title>Dynamic Content</title>
    <script type="text/javascript" language="javascript">
      function changeMe() {
        // Replace outerHTML with createElement and replaceChild.
        var oChild = document.getElementById("MyHeading");
        var oNewChild = document.createElement('H1');
        oNewChild.id = oChild.id;
        oNewChild.innerHTML = "Dynamic HTML!";
        oNewChild.style.color = "green";
        oChild.parentNode.replaceChild(oNewChild, oChild)

        // Use innerHTML instead of innerText.
        MyText.innerHTML = "Clicked. Thanks!";
        MyText.align = "center";
```

```
        // Change insertAdjacentHTML("BeforeEnd") to appendChild.
        var oPara = document.createElement('P');
        oPara.innerHTML = "Just give it a try!"
        oPara.align = "center";
        document.body.appendChild(oPara);
      }
    </script>
  </head>
  <body onclick="changeMe()">
    <h3 id="MyHeading">Welcome to Dynamic HTML!</h3>
    <p id="MyText">Click anywhere on this page.</p>
  </body>
</html>
```

上記のコード例は、その前のコード例よりも複雑になっていますが、現在使用されているブラウザーの多くで動作するという利点があります。

配置とアニメーション

配置機能を使用すると、別の要素やブラウザー ウィンドウ自体を基準としたページ上の相対位置に HTML 要素を配置できます。top と left の座標を指定することで、イメージ、コントロール、テキストなどの要素を希望の位置に正確に配置できます。z-index を割り当てることで、重なり合う要素の積み重ね順序を定義することもできます。

要素は、次のいずれかのキーワードを使用して配置できます。

表. キーワード

キーワード	説明
absolute	要素は、ドキュメントのレイアウト フローから除外され、そのコンテナーを基準として配置されます。
relative	要素は、そのコンテナーからのオフセット位置に配置されますが、その要素によって占有されるはずだったドキュメント内のスペースは保持されます。
fixed	Internet Explorer 7 以降でサポートされています。要素は、絶対配置の場合と同じように配置されますが、ドキュメントではなくブラウザー ウィンドウを基準として配置される点が異なります。

CSS を使用した配置

配置は、CSS のコンポーネントです。このため、要素の位置を設定するには、その要素の適切な CSS 属性を設定します。次の例では、イメージの絶対位置を設定する方法を示しています。

```
<html>
  <head>
    <title>Positioning</title>
  </head>
  <body>
    <h3>Welcome to Dynamic HTML!</h3>
    <img style="position:absolute;top:0;left:0;z-index:-1"
         src="clouds.gif" alt="clouds" />
  </body>
</html>
```

上記の例では、次のように処理しています。

- top と left を 0 に設定することで、イメージをドキュメントの左上隅に配置します。
- **z-index** 属性を -1 に設定することで、イメージをページ上のテキストの背後に配置します。

スクリプトを使用した配置

DOM はスタイルとスタイル シートへのアクセスを可能にするため、要素の色を設定および変更するのと同じくらい簡単に、任意の要素の位置を設定および変更できます。これによって、ユーザーがドキュメントを表示している方法に基づいて要素の位置を変更することが特に簡単になり、要素をアニメーション表示することも可能になります。アニメーションを実現するには、要素の位置を一定の時間間隔で少しずつ変化させるだけです。次の例では、イメージをページの右端から左端までスライドさせる方法を示しています。

```
<html>
  <head>
    <title>Dynamic Positioning</title>
    <script type="text/javascript" language="javascript">
      var id;
      function StartGlide()
      {
        Banner.style.pixelLeft = document.body.offsetWidth;
        Banner.style.visibility = "visible";
        id = window.setInterval(Glide,50);
```

```
    }
    function Glide()
    {
      Banner.style.pixelLeft -= 10;
      if (Banner.style.pixelLeft <= 0) {
        Banner.style.pixelLeft = 0;
        window.clearInterval(id);
      }
    }
  </script>
 </head>
 <body onload="StartGlide()">
  <h3>Welcome to Dynamic HTML!</h3>
  <img id="Banner"
       style="visibility:hidden;position:absolute;z-index:-1"
       src="eightball.gif" />
 </body>
</html>
```

上記の例では、次のように処理しています。

- イメージを絶対位置に配置し、最初は表示されないように設定します (visibility を hidden に設定)。
- **StartGlide** 関数によってイメージを表示し、イメージの位置をページの右端に設定して 50 ミリ秒の間隔で Glide の呼び出しを開始します。
- **Glide** 関数は 1 回呼び出されるごとにイメージを 10 ピクセルだけ左側に移動し、イメージが最終的に左端に達すると、指定された呼び出し間隔がキャンセルされます。

DHTML オブジェクト モデルについて

オブジェクト モデルとは

オブジェクト モデルは、DHTML をプログラム可能にするメカニズムです。オブジェクト モデルは、新しい HTML タグの習得や、新しい作成テクノロジを必要としません。

操作としては、特定の要素をマウスでポイントすること、キーを押すこと、フォームに情報を入力することなどが挙げられます。各イベントにスクリプトを関連付けることで、サーバーから新しいファイルを取得することなくコンテンツを動的に変更するようにブラウザーに指示できます。このこと

で得られる利点は、Web 作成者は従来よりも少ないページでインタラクティブな Web サイトを作成できること、ユーザーは新しいページが Web サーバーからダウンロードされるのを待つ必要がないこと、さらにその結果として、Web ブラウジングが高速化されてインターネット全体のパフォーマンスが向上することです。

スクリプトを使用した要素へのアクセス

オブジェクト モデルは、各要素が分類されるグループの階層である、要素のコレクションに重点を置いています。これらのコレクションのうちで最も重要なのは、**all コレクション**と **children コレクション**です。DHTML ドキュメントは、構造化されて配置された要素で構成されており、次の例では、各要素の有効範囲は、ドキュメントにおけるその要素タグの位置に応じて決まります。

```
<html>
  <body>
    <div>
      <p>Some text in a paragraph.</p>
      <img id="image1" src="mygif.gif">
    </div>
    <img id="image2" src="mygif.gif">
  </body>
</html>
```

上記の例では、div オブジェクトは p オブジェクトと image1 という名前の img オブジェクトを含んでいます (つまりこれらの親です)。逆に、image1 と p は div の子です。しかし、image2 という名前の img オブジェクトは、body オブジェクトの子です。

それぞれの要素オブジェクトは、階層内でその要素より下位にあるすべての要素が含まれた all コレクションと、その要素の直系の子孫である要素のみが含まれた children コレクションを持っています。上記の例では、b は、div オブジェクトの all コレクションに含まれますが、div オブジェクトの children コレクションには含まれません。同様に、div は body オブジェクトの children コレクションに含まれますが、p は含まれません。

各要素のこれらのコレクションに加えて、ドキュメント自体 (document オブジェクトで表現されます) は、いくつかの要素コレクションと非要素コレクションを持っています。最も重要なコレクションは、その Web ページ上のすべての要素が含まれた all コレクションです。このコレクションは、スクリプトを介して要素にアクセスするための主な手段です。

イベント ―― バブル、キャンセル、および処理

ボタンをクリックしたり、Web ページの一部をマウスでポイントしたり、ページのテキストを選択したりといった操作は、いずれも**イベント**を発生させます。DHTML ページ作成者は、そのイベントに応じて実行するコードを記述できます。このコードは、文字どおりイベントを処理するため、イベント ハンドラと呼ばれます。

このイベント モデルでは、ページ上のすべての HTML 要素は、あらゆるマウス イベントとキーボード イベントのソースになることができます。

次の表は、すべての HTML 要素が発行できる一般的なイベントを示しています。

表.*HTML* 要素が発行できる一般的なマウスイベント

マウス イベント	このイベントを発行させるユーザー操作
onmouseover	要素をマウスでポイントします (マウス ポインタを要素と重ねます)。
onmouseout	マウスを要素内から要素外に移動します (マウス ポインタを要素から離します)。
onmousedown	いずれかのマウス ボタンを押します。
onmouseup	いずれかのマウス ボタンを放します。
onmousemove	マウスを要素内で移動します。
onclick	マウスの左ボタンで要素をクリックします。
ondblclick	マウスの左ボタンで要素をダブルクリックします。

表. *HTML* 要素が発行できる一般的なキーボードイベント

キーボード イベント	このイベントを発行させるユーザー操作
onkeypress	キーを押してから放します (キーの押下から解放までの一連の操作)。キーを押したままにすると、複数の onkeypress イベントが発生します。
onkeydown	キーを押します。キーを押したままにしても、単一の onkeydown イベントしか発生しません。
onkeyup	キーを放します。

コンパクトでシンプルな保守しやすいコードを記述可能にするために、イベントを処理するための新しい方法としてイベント バブルが導入されました。イベント バブルは、HTML にとっては新しい機能であり、Web ドキュメントにイベント処理を組み込むための効率的なモデルを提供します。

イベント バブルは次のような利点をもたらします。

- 複数の一般的なアクションを一か所でまとめて処理できます。
- Web ページのコード総量を減らします。
- ドキュメントを更新するために必要なコード変更の数を減らします。

次のコードでは、実際のイベント バブルの簡単な例を示しています。

```
<html>
  <body>
    <div id="OuterDiv" style="background-color: red"
         onmouseover="alert(window.event.srcElement.id);">
      This is some text
      <img id="InnerImg" src="myImg.jpg" />
    </div>
  </body>
</html>
```

このページでは、ユーザーがマウス ポインタをテキスト上に移動すると、"OuterDiv" というテキストがダイアログ ボックスに表示されます。ユーザーがマウス ポインタをイメージ上に移動した場合は、"InnerImg" というテキストがダイアログ ボックスに表示されます。

ロールオーバー効果の処理

ロールオーバー効果とは、ユーザーがページの一部をマウスでポイントしたときに、その部分を変化させることです。

```
<html>
  <head>
    <title></title>
    <style type="text/css">
    .Item {
      cursor: hand;
      font-family: verdana;
      font-size: 20;
```

```
    font-style: normal;
    background-color: blue;
    color: white
  }
  .Highlight {
    cursor: hand;
    font-family: verdana;
    font-size: 20;
    font-style: italic;
    background-color: white;
    color: blue
  }
  </style>
</head>
<body>
  <span class="Item">Milk</span>
  <span class="Item">Cookies</span>
  <span class="Item">Eggs</span>
  <span class="Item">Ham</span>
  <span class="Item">Cheese</span>
  <span class="Item">Pasta</span>
  <span class="Item">Chicken</span>

  <script type="text/javascript">
  function rollon() {
    if (window.event.srcElement.className == "Item") {
      window.event.srcElement.className = "Highlight";
    }
  }
  document.onmouseover = rollon;

  function rolloff() {
    if (window.event.srcElement.className == "Highlight") {
      window.event.srcElement.className = "Item";
    }
```

```
    }
    document.onmouseout = rolloff;
    </script>
  </body>
</html>
```

上記の例では、7 つの **span** オブジェクトは、**Item** クラスを使用するように初期設定されています。これらの要素のいずれかがマウスでポイントされると、その要素は **Highlight** クラスを使用するように変更されます。

イベントバブルとロールオーバー効果の処理によって、次のことが可能になります。

- span オブジェクトからイベントを生成できるようになります。
- イベント バブルを使用すると、イベントを document オブジェクトレベルでキャプチャできます。このため、ロールオーバー効果の対象となる要素ごとに個別のイベント ハンドラを作成する必要はありません。

イベントのキャンセル

すべてのイベントは、そのイベントがキャンセルされない限り、その親要素にバブルアップします 。イベントをキャンセルするには、対応するイベント ハンドラで **window.event.cancelBubble** プロパティを true に設定する必要があります。イベントは、キャンセルされない限り、階層内をバブルアップし、そのイベントに対して登録されたすべての親要素によって処理されます。

最後の例では、イベント バブルを使用して、一般的な効果を一連の要素に適用する方法を示しています。特定の要素をその効果の対象から除外するには、次のコード行を変更するだけです。

<変更前のコード>

```
<span class="Item">Ham</span>
```

<変更後のコード>

```
<span class="Item" onmouseover="window.event.cancelBubble = true;"
    onmouseout="window.event.cancelBubble = true;">Ham</span>
```

Ham という単語をマウスで何回ポイントしても、そのスタイルは変化しません。その理由は、onmouseover イベントと onmouseout イベントは両方ともキャンセルされるからです。この結果として、これらのイベントは document オブジェクトまでバブルアップしなかったため、document オブジェクトでは、Ham という単語についてこれらのイベントを処理できませんでした。

特別な考慮事項

onmouseover イベントは、複数のオブジェクト上で同時に発生させることはできません。たとえば、次の例について考えてみましょう。

```
<div id="MyDiv">
  <img id="MyImg" src="myImg.jpg" />
</div>
```

マウス ポインタを img 上に移動した場合、イベントの順序は次のようになります。

```
MyDiv::onmouseover
MyDiv::onmouseout
MyImg::onmouseover
```

マウス ポインタを img から離すと、MyDiv::onmouseover イベントが再び発生します。

たとえば、マウス ポインタが div の外に移動したときに特殊な効果を実行するために、この移動のタイミングを検知する必要があるとします。このためには、**onmouseout** イベントをトラップするだけでは十分ではありません。このことを簡単にするために、Internet Explorer では、onmouseover イベントと onmouseout イベントのソース要素 (**fromElement**) とターゲット要素 (**toElement**) が示されます。これらのプロパティを **contains** メソッドと組み合わせて使用することで、マウス ポインタが特定の領域の外に移動したタイミングを検知できます。

次の例では、これらのプロパティとメソッドを使用する方法を示しています。

```
<html>
  <body id="Body">
    <div id="OuterDiv"
      style="width: 100px; height: 50px; background: red"
      onmouseover="over();" onmouseout="out();">
    <img id="Img1" src="img1.jpg" />
    <img id="Img2" src="img2.jpg" />
    <img id="Img3" src="img3.jpg" />
    </div>
  <script type="text/javascript">
  function over() {
    var s;
    s = "onmouseover: "+window.event.srcElement.id+" from: "+
    window.event.fromElement.id+" to: "+window.event.toElement.id;
    alert(s);
  }
  function out() {
    var s;
    s = "onmouseout: "+window.event.srcElement.id+" from: "+
```

```
    window.event.fromElement.id+" to: "+window.event.toElement.id;
    alert(s);

    if (!(OuterDiv.contains(window.event.toElement))) {
      alert("Out Now");
    }
  }
  </script>
  </body>
</html>
```

W3C ドキュメント オブジェクト モデルについて

World Wide Web コンソーシアム (W3C) の**ドキュメント オブジェクト モデル (DOM)** は、プラットフォームや言語に依存しないインターフェイスであり、ドキュメントのコンテンツ、構造、およびスタイルにスクリプトからアクセスしてこれらを更新することを可能にします。**W3C DOM** は、HTML ドキュメントと XML ドキュメントを表現する標準のオブジェクト セットの組み合わせ方法に関するモデルと、これらのオブジェクトにアクセスして操作するためのインターフェイスを備えています。

W3C DOM と DHTML オブジェクト モデルの比較

オブジェクト モデルは、ドキュメントやプログラムにアクセスしてこれらをプログラミングするためのメカニズムです。Internet Explorer 5 以降の DHTML オブジェクト モデルでは、すべての要素にアクセスできます。W3C DOM は、スクリプトですべての要素とすべての属性にアクセスできるという点で、DHTML オブジェクト モデルと一致しています。

W3C DOM の利点

W3C DOM を使用すると、ドキュメント ツリーを操作するためのさまざまな利点が得られます。コンテンツ作成者は、W3C DOM を使用して次のことができます。

- コンテンツを破棄して再作成することなく、ドキュメント ツリーの一部を別の部分に移動できます。
- 要素を作成し、ドキュメント ツリーの任意のポイントにアタッチできます。
- ドキュメント フラグメント内の新しいまたは既存のツリー分岐を編成および操作してから、これらのオブジェクトをツリーに挿入して戻すことができます。

コンテンツを破棄して再作成することなくドキュメント ツリーの一部を移動できることで、スクリプトのサイズを縮小できると共に作業効率が高まります。ul 要素と li 要素を使用して番号なしリストを作成する、次の HTML について考えてみましょう。

```
<ul id="oList" onclick="fnShuffleItem()">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
  <li>Item 5</li>
  <li id="oItem">Shuffle Item</li>
</ul>
```

DHTML オブジェクト モデルを使用して、直前のリストの項目を並べ替えるには、各項目を破棄して再作成する必要があります。li 要素が破棄される前に、これらの要素を再作成できるように outerHTML の値が記録されます。children コレクション内の要素のインデックスを常時監視することで、2 つのリスト項目の位置が入れ替えられます。次のサンプル コードでは、リスト項目を並べ替えます。

```
window.onload = fnInit;
var iShuffle;
function fnInit(){
  iShuffle = oList.children.length - 1;
}
function fnShuffleItem(){
  var oChildren = oList.children;
  if (iShuffle <= 0) {
    iShuffle = oChildren.length - 1;
    var sData = oChildren[0].outerHTML;
    oChildren[0].outerHTML = "";
    oList.innerHTML += sData;
  }
  else{
    var sSwap1 = oChildren[iShuffle-1].outerHTML;
    var sSwap2 = oChildren[iShuffle].outerHTML;
    oChildren[iShuffle-1].outerHTML = sSwap2;
    oChildren[iShuffle].outerHTML = sSwap1;
```

```
    iShuffle--;
  }
}
```

W3C DOM を使用すると、**swapNode** メソッドを使用してドキュメント ツリー階層内の 2 つのリスト項目の位置を入れ替えることで、同じ効果を実現できます。この例では、並べ替えられる項目は、前の項目と入れ替えられます。前の項目を特定するために、リストの children コレクションに対してクエリが実行される代わりに、**previousSibling** プロパティが使用されます。次のサンプル コードでは、W3C DOM の swapNode メソッドを使用してリスト項目を並べ替えます。

```
window.onload = fnInit;
var oShuffle;
function fnInit(){
   oShuffle = oList.lastChild;
}
function fnShuffleItem(){
   var oSwap = oShuffle.previousSibling;
   if (!oSwap) {
      oSwap = oList.lastChild
   }
   oShuffle.swapNode(oSwap);
}
```

W3C DOM の実装

W3C DOM インターフェイスを使用すると、ドキュメント ツリーのさまざまなノードにアクセスできます。W3C DOM のメンバを実装するには、ドキュメント ツリーの概念レイアウトと、ノード間の相互関係を理解する必要があります。

ここでは、階層関係に基づいたノードへのアクセス、ノードの作成、およびノードの操作によって W3C DOM を実装する方法について説明します。

階層関係に基づいたノードへのアクセス

要素の ID がわかっている場合は、特定のノードを識別することは簡単です。しかし、隣接する要素を検索することや、別の要素との関係に基づいて要素を検索することは難しいことがあります。W3C DOM では、ノード自体やノード間の相互関係を識別するためのいくつかのプロパティとコレクションが公開されています。

次の HTML では、ul 要素を使用して構築されたツリー構造を示しています。

```
<ul id="oParent">
```

```
   <li>Node 1</li>
   <li id="oNode">Node 2</li>
      <ul>
         <li>Child 1</li>
         <li id="oChild2">Child 2</li>
         <li>Child 3</li>
      </ul>
   <li>Node 3</li>
</ul>
```

ID が oNode であるリスト内のノードへの参照が得られれば、W3C DOM のメンバを使用して、隣接ノード、親ノード、および子ノードを識別できます。要素とテキストはノードとして存在するため、これらには W3C DOM を介してアクセスできます。

次のコードは、指定されたノードである oNode の基本構造を示しています。

```
<!-- oParent is the parent node of oNode. -->
<ul id="oParent">
  <!-- oNode is the childNode of oParent.-->
  <li id="oNode">
    <!-- Node 2 is a text node and is a child of oNode.-->
    Node 2
  </li>
</ul>
```

DHTML オブジェクト モデルでは、**ul** 要素と **li** 要素が公開されています。しかし、テキストにノードとして直接アクセスすることは、W3C DOM でのみ可能です。

最初の 3 つの li 要素は、それぞれ Node 1、Node 2、および Node 3 というラベルが割り当てられており、最初の ul 要素である oParent (親ノード) の子ノードです。子ノードのコレクションは、**childNodes** として **oParent** に公開されており、Node 1、Node 2、および Node 3 を含んでいます。これらの 3 つの li 要素は、子ノードとして、**parentNode** プロパティを通じて oParent を親ノードとして返します。Node 1、Node 2、および Node 3 は兄弟ノードであり、**previousSibling** プロパティと **nextSibling** プロパティを通じて相互に公開されます。Node 2 によって、ラベルのない 3 つの li 要素が含まれた childNodes コレクションが公開されます。

次のサンプルは、上記の HTML コードと同じ情報を提供するインタラクティブ ツリーです。このサンプルでは、選択したノードどうしの関係を確認したり、ツリーを操作したりできます。

◆ **親** —— oNode の親には、HTML コード内で oParent という ID が割り当てられています。

<DOM>

```
var oParent = oNode.parentNode
```

<DHTML オブジェクト モデル>

```
var oParent = oNode.parentElement
```

◆ **前の兄弟** —— oNode に隣接するノードには、Node 1 および Node 3 というラベルが割り当てられています。前の兄弟は Node 1 です。

<DOM>

```
var oPrevious = oNode.previousSibling
```

<DHTML オブジェクト モデル>

```
var oPrevious = fnGetSibling();
function fnGetSibling(){
   var oParent = oNode.parentElement;
   var iLength = oParent.children.length;
   for (var i = 0; i < iLength; i++) {
      if (oParent.children[i] == oNode) {
         return oParent.children[i - 1];
         break;
      }
   }
}
```

◆ **ノード値** —— oNode のノード値にアクセスするには、W3C DOM では nodeValue プロパティを使用し、DHTML オブジェクト モデルでは innerHTML プロパティまたは innerText プロパティを使用します。

<DOM>

```
oNode.childNodes[0].nodeValue = "The new label";
```

<DHTML オブジェクト モデル>

```
oNode.innerHTML = "The new label";
```

W3C DOM のクエリと DHTML オブジェクト モデルのクエリの類似点と相違点に注目してください。1 つ目に、**parentElement** プロパティと **parentNode** プロパティは、このサンプルでは同じ要素を返します。2 つ目に、DHTML オブジェクト モデルで前の兄弟を検索することは、W3C DOM で公開されている **previousSibling** プロパティを単に使用する場合よりも、手間がかかります。そして 3 つ目に、W3C DOM では、テキストはノードとして扱われ、childNodes コレクショ

ンを通じてアクセス可能です。**TextNode** オブジェクトの重要な特徴は、子ノードをまったく格納できないことです。

ノードの作成

ノードを作成するには、**createElement** メソッドや **createTextNode** メソッドを使用します。どちらのメソッドも document オブジェクトに対してのみ公開されており、HTML ドキュメント、DHTML ビヘイビアーなどで使用できます。DHTML オブジェクト モデルでは、要素をドキュメント階層に追加するには、**innerHTML** プロパティと **outerHTML** プロパティの値を変更するか、特定の要素に対して明示的なメソッドを使用します (table 要素用の **insertRow** メソッドや **insertCell** メソッドなど)。createElement メソッドでは、要素の名前のみが必要です。

次の構文を使用して新しい要素を作成します。

```
// Create an element
var oElement = document.createElement(sElementName);
```

createElement メソッドは Internet Explorer 5 以降では、すべての要素を独立要素として作成できます。また、id などの読み取り専用プロパティは、ドキュメント階層に挿入される前の独立要素に対しては、読み取り/書き込みプロパティです。一部の要素は、追加の手順を必要とする場合や、他の要素の存在に依存している場合があります。たとえば、**input** 要素の **type** 属性の既定値は text です。このため、ボタン コントロールを作成するには、**type** プロパティを button に設定し、**value** プロパティを設定してラベルを割り当てる必要があります。

次のサンプル コードでは、**fieldSet**、**legend**、および **input type="button"** コントロールをスクリプトで作成し、短いフォームを作成する方法を示しています。1 つの関数は要素を作成して末尾に付加するように設計されており、新しい要素への参照が返されるため、追加の要素を子として末尾に付加できる点に注目してください。

```
function fnCreate(sElement,sData,sType,oNode) {
   var oNewElement = document.createElement(sElement);
   if (sType) {
      oNewElement.type = sType;
      oNewElement.value = sData;
   }
   if (sData) {
      if (sElement.toLowerCase() != "input") {
         var oNewText = document.createTextNode(sData);
         oNewElement.appendChild(oNewText);
      }
```

```
   }
   if (oNode) {
      oNode.appendChild(oNewElement);
   }
   else {
      oNewForm.appendChild(oNewElement);
   }
   return oNewElement;
}

var oNode = fnCreate("fieldset");
fnCreate("legend", "My Form", "", oNode);
fnCreate("input", "Some Text", "text", oNode);
fnCreate("input", "A button", "button", oNode);
```

複数のノードを作成してスレッド化する際は、整形式の HTML を使用して、無効なツリーの作成を防止することが重要です。無効なツリーを作成してドキュメント階層に追加すると、予測不能な動作が発生する可能性があります。ほとんどの場合、Web 作成者は、このことを心配する必要はありませんが、table などの一部の要素には注意が必要です。

整形式の HTML テーブルは、table と tBody という、少なくとも 2 つのノードで構成されます。DHTML オブジェクト モデルでは、table (およびそこに含まれる行と列) を作成するには、**innerHTML** プロパティ、**insertRow** メソッド、**insertCell** メソッド、**rows** コレクション、および **cells** コレクションを使用します。tBody 要素は明示的には追加されませんが、テーブル オブジェクト モデルの一部として作成されます。たとえば、次のコードでは、DHTML オブジェクト モデルを使用して 2 つのセルから成るテーブルを作成します。

```
var sTable = "<table id='oTable1'></table>"
document.body.innerHTML += sTable;
oTable1.insertRow(oTable1.rows.length);
oTable1.insertRow(oTable1.rows.length);
oTable1.rows(0).insertCell(oTable1.rows(0).cells.length);
oTable1.rows(0).insertCell(oTable1.rows(0).cells.length);
oTable1.rows(0).cells(0).innerHTML = "Cell 1";
oTable1.rows(0).cells(1).innerHTML = "Cell 2";
```

次の HTML は、上記のコードの結果です。**tBody** 要素は、スクリプトで追加されなかったにもかかわらず存在していることに注目してください。

```
<table id="oTable">
  <tbody>
    <tr>
      <td>Cell 1</td>
      <td>Cell 2</td>
    </tr>
  </tbody>
</table>
```

W3C DOM で整形式のテーブルを作成するには、tBody 要素を明示的に作成し、ツリーに追加する必要があります。tBody 要素を組み込まないと、テーブル全体を構成しているノード群によって無効なツリーが形成され、予測不能な動作が生じます。次のスクリプトでは、W3C DOM を使用して、tBody 要素を組み込んでテーブルを作成する方法を示しています。

```
var oTable = document.createElement("table");
var oTBody = document.createElement("tbody");
var oRow = document.createElement("tr");
var oCell = document.createElement("td");
var oCell2 = oCell.cloneNode();
oRow.appendChild(oCell);
oRow.appendChild(oCell2);
oTable.appendChild(oTBody);
oTBody.appendChild(oRow);
document.body.appendChild(oTable);
oCell.innerHTML = "Cell 1";
oCell2.innerHTML = "Cell 2";
```

ノードの操作

ノードを簡単に操作できる W3C DOM のメソッドとしては、**cloneNode**、**removeNode**、**replaceNode**、**swapNode** メソッドなどが挙げられます。これらのメソッドを使用すると、ドキュメント ツリー全体にわたってノードのコピー、移動、および削除を実行できます。ノードが他のノードとどのように相互動作するのかを理解することで、ドキュメントをクリーンな状態に保って、クライアント側のスクリプトを安定させることができます。たとえば、次のリスト コードについて考えてみましょう。

```
<ul id="oParent">
  <li>Item 1</li>
  <li id="oNode2">Item 2</li>
```

```
  <ol id="oSubList">
    <li>Sub Item 1</li>
    <li>Sub Item 2</li>
    <li>Sub Item 3</li>
  </ol>
  <li>Item 3</li>
</ul>
```

無効なツリー構造は少しのミスで簡単に作成されてしまうため、ノードを操作する際は注意が必要です。たとえば、上記の HTML で表現されるリスト項目の 1 つをコピーしてリストの一番下に貼り付けるには、新しいノードは、最後のリスト項目の後ろではなく、親である oParent の後ろに追加する必要があります。新しいノードを最後のリスト項目の後ろに追加すると、このノードは他のノードの兄弟にはならずに、oParent の子になります。各リスト項目の後ろに子を追加する場合も、逆のことが当てはまります。新しいサブリストを作成して oParent の後ろに追加した場合は、そのサブリストは特定のリスト項目の子にはならずに、他のリスト項目の兄弟になります。

3 つの項目から成る新しいサブリストを作成するには、1 つの新しい ol 要素と 3 つの新しい li 要素を作成し、最後のリスト項目の後ろに追加します。ただし、この構造は既に存在しているため、cloneNode メソッドを使用してこの構造をコピーすることもできます。このメソッドに true をパラメーターとして渡すことで、子もコピーされます。次のコードでは、サブリスト全体をコピーし、最後のリスト項目の後ろに追加する方法を示しています。

```
var oClone = oSubList.cloneNode(true);
oParent.childNodes[oParent.childNodes.length-1].appendChild(oClone);
```

replaceNode メソッドや swapNode メソッドを使用すると、ドキュメント階層内のノードを別のノードに置き換えることができます。これらのメソッド間の違いは、replaceNode は、このメソッドの呼び出し元ノードをドキュメント階層から削除するのに対して、swapNode は、このメソッドの呼び出し元ノードを置き換え対象ノードの位置に移動するという点です。次のサンプル コードでは、両方のメソッドの使用方法を示しています。

```
<script type="text/javascript">
window.onload = fnInit;
function fnInit() {
   var oNewPara = document.createElement("p");
   var oText = document.createTextNode("This is the new paragraph");
   oNewPara.appendChild(oText);
   document.body.appendChild(oNewPara);
   // Paragraph 2 is replaced with the new paragraph
```

```
   oP2.replaceNode(oNewPara);
   // The position of paragraph 1 is exchanged with the new paragraph
   oNewPara.swapNode(oP1);
}
</script>

<p id="oP1">This is the first paragraph</p>
<p id="oP2">This is the second paragraph</p>
```

イベント モデルについて

イベントとは、状態の変化のようなアクションに応じて、またはドキュメントの表示中にマウスがクリックされたり、キーが押されたりした結果として発生する通知です。イベント ハンドラはコードです。これは通常、スクリプト言語で記述された関数やルーチンで、対応するイベントが発生すると制御を受け取ります。

イベントのライフサイクル

一般的なイベントの**ライフサイクル**は、次のステップから構成されます。

1. イベントに関連付けられているユーザー アクションまたは状態が発生します。
2. イベントの状態を反映するように event オブジェクトが直ちに更新されます。
3. イベントが発生します。これは、イベントに応える実際の通知です。
4. ソース要素に関連付けられているイベント ハンドラが呼び出され、そのアクションが実行され、制御が戻ります。
5. イベントは階層における次の要素にバブルアップし、その要素のイベント ハンドラが呼び出されます。
6. 既定の最終アクションが指定されている場合、ハンドラによってこのアクションがキャンセルされていない場合にのみ、アクションが実行されます。

次の例では、**onclick** イベントのイベント ハンドラ wasClicked を定義し、BODY 要素に関連付けます。このドキュメント内の任意の場所をユーザーがマウス ボタンでクリックすると、イベントが発生し、イベントが発生した要素のタグ名と "I was clicked" というメッセージがイベント ハンドラによって表示されます。

```
<html>
  <body onclick="wasClicked()">
    <h1>Welcome!</h1>
    <p>This is a very <b>short</b> document.</p>
```

```
    <script type="text/javascript" language="javascript">
      function wasClicked() {
        alert("I was clicked " + window.event.srcElement.tagName);
      }
    </script>
  </body>
</html>
```

event オブジェクトの **srcElement** プロパティは、イベントが発生した要素オブジェクトを表します。これは、イベントに応えてイベント ハンドラが実行するアクションを特定するうえで役立ちます。

別のイベント ハンドラ wasAlsoClicked を p 要素にアタッチすることで、ドキュメント要素の親子関係を説明します。

```
<html>
  <body onclick="wasClicked()">
    <h1>Welcome!</h1>
    <p onclick="wasAlsoClicked()">
      This is a very <b>short</b> document.</p>
    <script type="text/javascript" language="javascript">
      function wasClicked() {
        alert("I was clicked " + window.event.srcElement.tagName);
      }
      function wasAlsoClicked() {
        alert("You clicked me " + window.event.srcElement.tagName);
      }
    </script>
  </body>
</html>
```

前の例では、ユーザーが見出しの "Welcome!" をクリックすると、"I was clicked H1" というメッセージが表示されます。しかし、ユーザーが "short" をクリックした場合は、"You clicked me B"、"I was clicked B" という 2 つのメッセージがこの順番で表示されます。最初の "Welcome!" をクリックしたケースでは、onclick イベントが発生し、H1 要素がイベントのソース要素として設定されます。この要素にはイベント ハンドラがないため、イベントが階層内の親要素 BODY にバブルアップし、そのイベント ハンドラ wasClicked が呼び出されます。

2 つ目のケースで、ユーザーが "short" という単語をクリックしたとき、やはり onclick イベントが発生します。この場合、b 要素がソース要素として設定されます。b にはイベント ハンドラがないため、イベントは p 要素にバブルアップし、そのイベント ハンドラである wasAlsoClicked が呼び出されます。しかし、これでイベントが終わるわけではありません。wasAlsoClicked から制御が戻った後も、イベントは、階層の次の親要素である BODY に引き続きバブルアップするので、wasClicked が呼び出されます。

次の例では、**setBodyStyle** と **setParaStyle** という 2 つのイベント ハンドラを定義します。それぞれ、ユーザーがドキュメント内でクリックしたときと、パラグラフ内でクリックしたときに呼び出されます。ユーザーがパラグラフ内でクリックしたときに、パラグラフのスタイルのみを変化させるには、setParaStyle ハンドラで cancelBubble を使用して、イベントが BODY にバブルアップしないようにします。

```
<html>
  <body onclick="setBodyStyle()">
    <h1>Welcome!</h1>
    <p onclick="setParaStyle()">
      This is a very <b>short</b> document.</p>
    <script type="text/javascript" language="javascript">
      function setBodyStyle() { // Set all headings to green
        var coll = document.all.tags("H1");
        for (i = 0; i < coll.length; i++)
          coll.item(i).style.color = "green";
      }
      function setParaStyle() { // Underline the paragraph
        var el = window.event.srcElement;
        while ((el != null) && (el.tagName != "P")) {
          el = el.parentElement;
        }
        if (el != null)
          el.style.textDecoration = "underline";
        window.event.cancelBubble = true;
      }
    </script>
  </body>
</html>
```

イベント ハンドラのアタッチ

イベントが発生したときにハンドラが呼び出されるようにするには、**イベント ハンドラ**を特定の要素またはオブジェクトに関連付ける必要があります。イベント ハンドラを関連付けるには、次のいずれかに該当する方法を使用します。

イベント ハンドラ関数を宣言し、HTML タグの適切なインライン イベント属性でその関数への呼び出しを割り当てます。次の例では、flip という名前の JavaScript 関数を定義し、この関数を **onmouseover** イベントのイベント ハンドラとして img 要素に関連付けます。

```
<script type="text/javascript" language="javascript">
function flip() {
    // Carry out some work
}
</script>
    ...
<img src="sample.gif" onmouseover="flip()">
```

前の例は、イベント ハンドラ関数のバインド方法を示しています。ただし、イベントが発生するたびに評価に使用される式は何でもかまいません。これは式であるため、関数名の後にかっこが必要になることに注意してください。

イベント プロパティへのアタッチ

要素のインライン イベント属性は、プロパティとしても利用できます。つまり、プロパティを使用して、要素のイベント処理をいつでも動的に変更できるということです。

次の JavaScript の例では、**event** プロパティを使用してイベント ハンドラ **setHeadStyle** を H1 要素に関連付けます。**setHeadStyle** 関数へのポインタは、要素の **onclick** プロパティに割り当てられます。

```
<h1 id="MyHeading">Welcome!</h1>
    ...
<script type="text/javascript" language="javascript">
function setHeadStyle() {
  window.event.srcElement.style.color = "green";
}
document.all.MyHeading.onclick = setHeadStyle;
</script>
```

イベント バブルについての詳細

イベント バブルにより、イベントが発生するすべての要素のイベント ハンドラに対して、イベントに応答する機会が与えられます。次の例を考えてみましょう。

```
<p onclick="doPara()">
Jump to a <b>sample</b> document.
</p>
```

ユーザーが "sample" という単語をクリックしたとき、p 要素の他の単語をクリックしたときと同様のアクション、つまり doPara という名前のイベント ハンドラへの呼び出しが実行されると期待しても不思議ではありません。イベントがソース要素からバブルアップしなかった場合、"sample" をクリックしても、b 要素にはイベント ハンドラがバインドされていないため、何のアクションも起こりません。実際は、イベントが要素の階層をバブルアップするため、要素の親階層におけるすべてのイベント ハンドラに、応答の機会があります。このケースでは、ユーザーが "sample" をクリックすると、doPara が呼び出されます。

イベント バブルのことを忘れていた場合、最初は、一部の結果に困惑したのではないでしょうか。たとえば、**onmouseover** および **onmouseout** イベントを使用して、パラグラフのコレクションの表示/非表示の切り替えを行うと同時に、リスト内のイベントを使用してコレクション内のパラグラフを強調表示するドキュメントについて考えてみましょう。

```
<div onmouseover="showPara()" onmouseout="hidePara()">
My Menu
<p style="display:none" onmouseover="highlight()"
  onmouseout="unhighlight()">Item 1</p>
<p style="display:none" onmouseover="highlight()"
  onmouseout="unhighlight()">Item 2</p>
</div>
```

ユーザーがマウス ポインタをコレクション内のパラグラフに移動したり、そこからマウス ポインタを移動したりするたびに、div のイベント ハンドラが呼び出されます。これを考慮せずにハンドラが記述されていると、パラグラフを表示する必要があるときにそれが非表示になる可能性があります。

通常、このように入れ子になっているイベント ハンドラを使用する場合は、イベント ハンドラが呼び出しのたびにアクションを実行する必要があるかどうかを、慎重に検討する必要があります。アクションを制御する方法の 1 つとして、**cancelBubble** プロパティを使用する方法があります。前の例では、**highlight** および **unhighlight** イベント ハンドラは、cancelBubble を true に設定することで、イベントが div 要素にバブルアップされないようにすることができます。別の方法とし

ては、イベントのソース要素、つまり event オブジェクトの **srcElement** プロパティを確認し、その要素に適したアクションを選択する方法もあります。

要素がもともとイベントをサポートしていない場合でも、要素でイベントをトラップすることが可能です。たとえば、**onkeydown** イベントは **input type="text"** オブジェクトでサポートされています。ただし、イベント バブルのため、div の onkeydown イベントに対してイベント ハンドラを定義することができます。div に含まれる要素によって発生した onkeydown イベントは、div にバブルアップし、div のイベント ハンドラによって処理されます。

```
<div onkeydown="OnChangeHandler()">
  <input type="text" />
  <input type="text" />
</div>
```

戻り値と既定のアクションのキャンセル

イベント ハンドラは、JavaScript の return ステートメントのように、言語に対して定義されている戻り値のメカニズムを使用するか、event オブジェクトの returnValue プロパティを使用することで、イベントに値を返すことができます。

戻り値は、イベントに関連付けられている既定のアクションを制御するのに役立ちます。

たとえば、a 要素をクリックすると、href 属性で指定されたドキュメントがブラウザーによって読み込まれます。このような既定のアクションは、イベント ハンドラから false を返すことで、キャンセルすることができます。

すべてのイベント バブルが完了した後、returnValue が false の場合は、イベントに関連付けられている既定のアクションはキャンセルされます。このプロパティが設定されていない場合は、最後の関数の戻り値が false になります。一部のイベントでは、既定のアクションが関連付けられていないため、別の方法で returnValue (または関数の戻り値) を使用します。

event オブジェクト

window オブジェクトでは、event オブジェクトはすべてのイベント ハンドラにアクセスできます。また、言語に依存しないため、ハンドラがイベントに関する情報を取得して変更するのに役立つ方法です。たとえば、次の JavaScript の例では、いずれもオブジェクトを使用して、イベントが発生した要素のタグ名を表示します。

```
<script type="text/javascript" language="javascript">
function doClick() {
  alert(window.event.srcElement.tagName);
}
```

```
</script>
```

srcElement プロパティは、event オブジェクトで最も重要なプロパティの 1 つです。多くのイベント ハンドラがこのプロパティを使用して、イベントのソースに応じて実行するアクションを決定しています。

別の重要なプロパティとして **cancelBubble** があります。これは、所定のイベントに対するイベント バブルを制御するプロパティで、true に設定されている場合、イベントが要素階層をバブルアップするのを防ぎます。

returnValue プロパティは、イベントに値を返すための言語に依存しない方法をイベント ハンドラに提供します。ほとんどのイベントでは、戻り値を false に設定すると、イベントに対する既定のアクションがキャンセルされます。

キーボード イベント

キーボード イベントは、ユーザーがキーを押したり離したりしたときに発生します。**onkeydown** および **onkeyup** イベントは、それぞれキーが押されたり、離されたりしてキーの状態が変わると発生します。これらのイベントは、Shift、Ctrl、Alt キーなどのシフト状態キーを含むキーボード上のすべてのキーで発生します。

onkeypress イベントは、ユーザーのキーボード入力が文字に変換されたときに発生します。たとえば、ユーザーが文字や数字のキーを押したり、シフト キーと文字および数字の組み合わせを押したりしたときに発生します。

キーボード イベントが発生すると、event オブジェクトの **keyCode** プロパティには、対応するキーの **Unicode** キーコードが含められます。**altKey**、**ctrlKey**、および **shiftKey** プロパティは、Alt、Ctrl、および Shift キーの状態を表します。

keyCode プロパティの値を変更するか整数値を返すことで、イベントに関連付けられているキーを変更できます。ゼロまたは false を返すと、イベントをキャンセルできます。

マウス イベント

マウス イベントは、ユーザーがマウスを動かしたり、左ボタンをクリックしたりしたときに発生します。**onmousemove** イベントは、ユーザーがマウスを動かしたときに発生し、**onmouseover** および **onmouseout** は、マウスが要素内および要素外に移動した場合に発生します。**onmousedown** および **onmouseup** イベントは、それぞれ左マウス ボタンが押されたり離されたりして状態が変わると発生します。**onclick** および **ondblclick** イベントは、ボタンがシングルクリックおよびダブルクリックされると発生します。

マウス イベントが発生すると、event オブジェクトの **button** プロパティは、どのマウス ボタンが押されたのかを識別します。x および y プロパティは、イベント発生時のマウス ポインタの位

置を表します。onmouseover および onmouseout イベントの場合、**toElement** および **fromElement** プロパティは、マウスの移動先および移動元の要素を表します。

マウス クリック

onclick イベントは、ユーザーがボタンを押して離すと発生します。**ondblclick** イベントは、2 回の連続した onclick イベントの間隔が、システムで定義された時間内である場合に発生します。

マウス クリック イベントは、**onmousedown** と **onmouseup** イベントの間に発生します。たとえば、onclick イベントの後に onmouseup イベントが続きます。**ondblclick** イベントは、次の一連の操作の最後に発生します。

1. **onmousedown**
2. **onmouseup**
3. **onclick**
4. **onmouseup**
5. **ondblclick**

要素間の移動

onmouseover および **onmouseout** イベントは、マウス ポインタが、ある要素から別の要素に移動するときに発生します。次のようなドキュメントの断片について考えてみましょう。

```
<h1>Move from here</h1>
<p id="myP"><b><i>To here</i></b></p>
```

前の例で、マウス ポインタが H1 要素から単語 "To here" に移動すると、ポインタは p、b、および i 要素の上を通ります。ただし、イベントの順序は、次のように簡略化されます。H1 での onmouseout が階層をバブルアップし、i 要素の onmouseover がやはり階層をバブルアップします。

event オブジェクトの fromElement および toElement プロパティは、マウス ポインタの移動元と移動先の要素オブジェクトを返します。イベント バブルによって、マウスの移動で通過した各領域を特定することができます。

要素間を移動するとき、マウス ポインタが元の要素を離れたことを表すため、まず onmouseout イベントが発生します。次に、onmousemove イベントが発生し、マウス ポインタが移動したことを示します。最後に、onmouseover が発生し、マウス ポインタが新しい要素に入ったことを示します。

load および unload イベント

ドキュメントの現在の状態を示すイベントは、**onload**、および **onunload** の 2 つです。

onload イベントは、ドキュメントが読み込まれ、ページ上のすべての要素が完全にダウンロードされると発生します。onunload イベントは、別のドキュメントに移動するときなど、ドキュメントがアンロードされる直前に発生します。

フォームの概要

フォームは、情報を収集、表示、および配信するためのインターフェイスを提供する HTML の重要なコンポーネントです。フォームではテキストフィールド、ボタン、チェック ボックスなどのさまざまなコントロールを利用できるので、クライアントとサーバー間の情報交換手段による Web ページの機能拡張を図ることができます。

フォームの利点

フォームを使用する利点は次のとおりです。

- **実装の容易性** —— ユーザーと情報を交換する方法を提供するいくつかのコントロールが用意されています。
- **拡張性** —— ダイナミック HTML (DHTML) により、Web 作成者は必要に応じてフォームとコントロールを更新および修正できます。
- **アクセス性** —— カスケーディング スタイル シート (CSS) の各種属性を使用すると、特定のスタイルでフォーム コントロールをカスタマイズできます。

フォーム コントロール

フォーム コントロールを使用すると、Web 作成者は他のグラフィックベースのプログラミング言語と同様の利点を Web サイトで実現できます。次の表は、サポートされているフォームコントロールと、送信される属性、および説明を示しています。

表. フォーム コントロール

要素/コントロール	送信される属性	説明
button	NAME、VALUE	DHTML を組み込み可能で機能豊富なボタンコントロールをレンダリングします。
input type=button	NAME、VALUE	**input** 要素をボタンとしてレンダリングします。このボタンに **onclick** などのイベントを組み込み、クリックされたときに実行するアクションを設定できます。

input type=checkbox	NAME、VALUE	**input** 要素をチェック ボックスとしてレンダリングします。チェックボックスは、トピックから複数のグループ (好みの雑誌など) を選択する場合や、はい/いいえ形式の判断結果を指定する場合 (電子メールアドレスをメーリングリストに追加する場合など) に便利です。**onclick** などのイベントを **checked** プロパティと共に使用すると、状態 (非表示オブジェクトの表示など) を判断できます。
input type=file	NAME、VALUE (エンコードされて送信)	スクリプトで操作できないテキストフィールドおよび参照ボタンをレンダリングします。このコントロールが適切に機能するには、フォームの **enctype** 属性を multipart/form.data に設定する必要があります。セキュリティ上の理由により、クライアント上のスクリプトから **value** 属性にアクセスすることはできません。
input type=hidden	NAME、VALUE	非表示のコントロールをレンダリングします。ユーザーに表示する必要がない追加の情報を送信する場合に、非表示のコントロールが便利です。
input type=image	NAME	送信コントロールをイメージとしてレンダリングします。このイメージコントロールは、ボタンの代わりとして使用すると便利です。このイメージコントロールをクリックすると、これがそのフォームで送信される唯一のイメージコントロールとなります。また、イメージ コントロールは、**name** 属性を指定している場合にのみ送信されます。
input type=password	NAME、VALUE	内容をアスタリスクで置き換えたテキストフィールドをレンダリングします。クライアント上で同じ画面を見ている人に個人情報がわからないようにする場合に、このパスワード フィールドが便利です。
input type=radio	NAME、VALUE	ラジオコントロールがレンダリングされます。これは、一連の項目から 1 つを選択する場合に便利です。1 つのラジオコントロールを選択すると、同じ name 属性値を持つ他のラジオコントロールはすべてクリアされます。
input type=reset		フォームに入力されているすべての値をリセットするボタンをレンダリングします。
input type=submit	NAME、VALUE	特定のフォームで入力されたすべての情報を **action** 属性で指定されたスクリプトまたはプログラムに送信するボタンをレンダリングします。**method** 属性は、フォーム情報の配信

		方法を指定します。 送信コントロールは、name 属性を指定している場合にのみ送信されます。複数の送信コントロールが存在するフォームでは、クリックしたコントロールのみが送信されます。
input type=text	NAME、VALUE	単一行のテキストフィールドをレンダリングします。
select	NAME、option (選択されているものを送信)	内容を指定した **option** 要素を持つドロップダウンリスト ボックスまたはリストボックスを作成します。または、**options** コレクションと **add** メソッドを使用して動的にドロップダウン リスト ボックスまたはリスト ボックスを作成します。リスト ボックスは、**size** 属性に 1 より大きい値を指定することによって作成します。リスト ボックスでは、**multiple** 属性を指定すると複数のオプションを選択できるようになります。multiple 属性を指定すると、リスト ボックスが作成されます。**size** 属性および multiple 属性を指定していない場合は、ドロップダウン リスト ボックスが作成されます。
textArea	NAME、VALUE	複数行のテキスト フィールドをレンダリングします。 **textArea** 要素の **value** 属性は、**innerText** プロパティと同等であり、開始タグと終了タグの間に配置します。

フォームの実装

Web サイト上でフォームを実装する前に、**form** 要素の必要性を判断する必要があります。form 要素は、フォーム コントロールをサーバーに送信するために必要です。フォームの送信では、送信を指定した form 要素にあるコントロールの名前と値のペアのみが送信されます。

form 要素を使用する場合、**action** 属性および **method** 属性でフォーム コントロールのデータをどこにどのように配信するのかを指定します。ここでは、**get** メソッドと **post** メソッドを使用します。

- get メソッドでは、action 属性で指定した URL にフォーム データを付加しますが、データの最大文字数は実装によって制限があります (Internet Explorer における最大文字数は 2,083 文字)。このメソッドは、サイズ上の制限とドキュメントに置いた場合の可視性に問題があることから、フォームにおける使用には適しません。

◆ post メソッドでは、post トランザクションとしてフォーム データが送信され、そのサイズに制限はありません。したがって、データをサーバーに配信する場合にはこのメソッドが適しています。

フォーム アクションは、配信された情報を処理して HTTP サーバーに返信することを目的としたスクリプトです。これにより、処理結果をクライアントに返すことができます。次のサンプルは、フォーム データを post トランザクションとしてサーバーに送信する form 要素の構文を示しています。

```
<form action="process.aspx" method="post">
    ...
</form>
```

フォーム コントロールは、HTML 要素として Web ページに追加されます。たとえば、訪問者の名字用と名前用に 2 つのテキスト フィールドを追加するために必要なのは input type="text" コントロールのみです。

```
<form>
  First Name: <input type="text" name="FirstName" />
  <br />
  Last Name: <input type="text" name="LastName" />
</form>
```

コントロールを追加する際は、ID 属性と NAME 属性の違いを理解しておくことが重要です。どちらの属性もクライアント側スクリプトからアクセス可能ですが、フォームの送信で渡されるのは NAME のみです。さらに、ID はスクリプトで直接参照できますが、NAME は、該当の要素が form 要素の子である場合はその親フォームの NAME を指定して参照する必要があります。たとえば、Web 作成者がクライアント上で FirstName テキスト フィールドの VALUE 属性にアクセスしようとする場合は、次の点を考慮する必要があります。

```
<form name="Form1">
  <input type="text" id="oFirstName" name="FirstName" />
</form>
```

1. ID 属性を設定しておくと、その要素にはスクリプトから直接アクセスできます。ID は送信されないことに注意が必要です。

   ```
   var sFirstName = Form1.oFirstName.value;
   ```

2. テキスト フィールドが form 要素の子ではない場合は、NAME 属性を使用してテキスト フィールドを識別できます。

```
var sFirstName = FirstName.value;
```

3. テキスト フィールドが form 要素の子であり、ID 属性が設定されていない場合、そのテキスト フィールドにアクセスするには form の NAME 属性、または forms コレクションでの form の順序位置を使用する必要があります。また、item メソッドを使用すると、NAME や ID を指定せずにフォームの特定のコントロールを参照できます。

```
// Access the VALUE through the form NAME
var sFirstName = Form1.FirstName.value;

// Access the VALUE through the forms collection
// using the NAME attribute.
var sFirstName = document.forms[0].FirstName.value;

// Access the VALUE through the forms collection
// using the item method.
var sFirstName = document.forms[0].items[0].value;
```

input type="text"、input type="password"、および textArea の各コントロールは、名前と値のペアを送信する場合に便利です。

select、input type="checkbox"、および input type="radio" の各コントロールは、キーボードから入力しなくても (キーボードからの入力は可能です)、事前に設定された情報から選択できる理想的な方法です。

select コントロールは、事前に定義されたリストまたは動的に作成されたリストから 1 つ以上の項目をすばやく選択する場合にたいへん便利です。select コントロールでは、Web 作成者は SIZE 属性および MULTIPLE 属性を使用し、実装をいくつかの選択肢から選択できます。

select コントロールのこの 3 種類の実装は、ドロップダウン リスト ボックス、リストボックス、および複数選択リスト ボックスです。**option** オブジェクトは、コントロールにデータを読み込むために使用します。選択された値を取得するには、options コレクションに **selectedIndex** プロパティを使用して該当の項目を識別します。たとえば、次のコードには、サンプルの select コントロールと数行のスクリプトが記述されています。このスクリプトでは、**alert** メソッドを使用して、**onchange** イベントが発生したときに、選択された値を表示します。

```
<script type="text/javascript" language="javascript">
function fnShowText(){
   /* Use the selectedIndex property of the SELECT control
```

```
   to retrieve the text from the options collection. */

   var sText = oSel.options(oSel.selectedIndex).text;
   alert(sText);
}
</script>

<select id="oSel" onchange="fnShowText()">
  <option>Luna</option>
  <option>Saturn</option>
  <option>Europa</option>
  <option>Io</option>
  <option>Jupiter</option>
  <option>Monolith</option>
</select>
```

複数選択リストから選択された値を取得するには、少し異なる手法が必要です。すべてのオプションで **selected** プロパティを調べることで、どのオプションが選択されているかを判断できます。その場合、確認処理で扱っている現在のインデックスを使用すれば、その項目のテキストや値を取得できます。次のサンプルコードは、その方法を示しています。

```
<script type="text/javascript" language="javascript">
function fnShowText() {
  /* Look through all the options to find selected items. */
  for (var i = 0; i < oSel.options.length; i++) {
    /* Check to see if a particular option is selected */
    if (oSel.options(i).selected) {
      /* Return the selected value */
      alert(oSel.options(i).text);
    }
  }
}
</script>
<select id="oSel" multiple="multiple" size="5"
    onchange="fnShowText()">
  <option>Luna</option>
```

```
  <option>Saturn</option>
  <option>Europa</option>
  <option>Io</option>
  <option>Jupiter</option>
  <option>Monolith</option>
</select>
```

複数選択リストの送信では、選択された項目のコンマ区切りリストが送信されます。VALUE 属性がオプションとして提供されていない場合、代わりに **TEXT** 属性が送信されます。

input type="checkbox" コントロールと **input type="radio"** コントロールは、互いに似たようなインターフェイスを提供しますが、その機能は異なっています。同じ **NAME** 属性を持つ複数の input type="radio" コントロールはコレクションとして機能します。コレクションとして、一度にオンにできる input type="radio" コントロールは 1 つのみです。一度に選択できるオプションが 1 つのみなので、このコントロールは、二者択一の選択や 1 つの値のみ持つことができるオプションに適しています。

たとえば、小学校の歴史の教師が、複数の選択項目を持つ試験を作成する場合、その選択項目をラジオ コントロールにすることができます。次のサンプルコードは、input type="radio" コントロールの実装方法を示しています。**label** 要素を使用するには、各ラジオ コントロールに一意の ID 属性を指定し、NAME 属性はそのままにしてコレクションとして機能できるようにします。問題の正解を指定する input type="button" オブジェクトを記述し、コントロールの **checked** プロパティを検証するために **onclick** イベントを使用します。

```
<form name="Form1" action="Default.aspx" method="post">
  The first President of the United States was:
  <table>
    <tr>
      <td>
        <label for="oRadio1">George Washington</label>
      </td>
      <td>
        <input type="radio" name="radio1" id="oRadio1" />
      </td>
    </tr>
    <!-- Place additional choices here -->
  </table>
```

```
  <input type="button" value="Check Answer" onclick="fnGetAnswer()" />
</form>
```

そして、どのラジオ コントロールが選択されているのかを判別するために、コレクションにアクセスし、checked プロパティを確認します。

```
<script type="text/javascript" language="javascript">
var aRadio1Answers = new Array("Correct","Incorrect","Incorrect");
function fnGetAnswer() {
   for (var i = 0; i < Form1.radio1.length; i++) {
      if (Form1.radio1(i).checked) {
        alert(aRadio1Answers[i]);
      }
   }
}
</script>
```

一方、1 つのグループから複数項目を選択する必要がある場合は、input type="checkbox" コントロールを使用できます。たとえば、ある範囲の知識について教師が学生に試験問題を出す場合、複数の選択を可能にするには input type="checkbox" コントロールが適しています。この構文は、ラジオ コントロールとほとんど同じで、異なる点は TYPE 属性のみです。

```
Select the inventions created before 1500 AD.
<form name="Form1" action="Default.aspx" method="post">
  <table>
    <tr>
      <td>
        <label for="oInvention1">Crop Rotation (science)</label>
      </td>
      <td>
        <input type="checkbox" name="invention1" id="oInvention1" />
      </td>
    </tr>
    <!-- Place additional choices here -->
  </table>
  <input type="button" value="Check Answers" onclick="fnCheck()" />
</form>
```

この例では、正解の値が事前に決まっています。実際面では、正しい値であるかどうかを確認することは簡単です。ただし、正しくない値について確認し、さまざまな状況を処理するには少し工夫が必要です。

```
<script type="text/javascript" language="javascript">
// Preset binary markers for the correct answers.
var aAnswers = new Array(0,1,0,0,0,1,1,0,1,1);
// Total number of correct answers possible.
var iTotalCorrect = 5;
// Number of attempts to submit answers.
var iTries = 0;

function fnCheck() {
  // Increment tries and set correct and incorrect answers to zero.
  iTries++;
  var iCorrect = 0;
  var iIncorrect = 0;
  // Use an array of preset binaries and see
  // if the correct control is checked.
  for (var i = 0; i < Form1.invention1.length; i++) {
    // If the check box is checked and the binary is 1, it is a
    // correct answer.
    if ((Form1.invention1(i).checked) && (aAnswers[i] == 1)) {
      iCorrect++;
    }
    // If the check box is checked and the binary is 0,
    // it is an incorrect answer.
    if ((Form1.invention1(i).checked) && (aAnswers[i] == 0)) {
      iIncorrect++;
    }
  }
  // If there are fewer correct answers than the total, or there
  // are any incorrect answers, return the following message.
  if ((iCorrect < iTotalCorrect) || (iIncorrect > 0)) {
    alert("You have some correct and " + iIncorrect
```

```
          + " incorrect answers.");
  }
  // If there are no incorrect answers, and all the correct answers
  // are chosen, continue.
  if ((iCorrect == iTotalCorrect) && (iIncorrect == 0)) {
    // Proceed based on number of attempts.
    if (iTries == 1) {
      alert("Congratulations, you got them all right on the first try!");
    }
    else {
      alert("Congratulations. It took " + iTries + " tries.");
    }
  }
}
</script>
```

フォームの編成と機能の拡張

フォームは、すぐに大きくなり、編成や使用が困難になります。そこで、HTML では、Web 作成者がフォームを容易に管理できるように、またユーザーがフォームの特定の場所にすばやく移動できるようにいくつかの機能がサポートされています。

◆ 視覚的な機能の拡張により、ドキュメント上のフォームの配置を Web 作成者が効果的に編成できる方法が用意されています。

- **fieldset** 要素は、境界を持つ 1 つのフィールドに複数のフォーム コントロールをまとめます。**legend** 要素は、このフィールドのタイトルを指定します。
- **label** 要素を使用すると、Web 作成者はフォーム コントロールの目的を容易に識別でき、ユーザーはラベルをクリックすることでフォーカスを該当のコントロールに容易に設定できます。

◆ キーボードの拡張は、フォームの特定の領域にユーザーがすばやくアクセスできる論理的な手段を提供します。

- **tabindex** 属性では、ユーザーが Tab キーを押して次々にフォーム コントロール間を移動する場合の移動順序を Web 作成者が指定できます。
- **accesskey** 属性では、特定のフォーム コントロールにフォーカスを設定するキーの組み合わせを Web 作成者が指定できます。

前で説明した機能拡張を使用すると、より多くのユーザーが使用できるようにフォームを編成できます。次のコードは、この機能拡張の実装方法を示しています。

```
<form action="Default.aspx" method="post">
  <fieldset style="width: 200;">
    <!-- The LEGEND identifies the FIELDSET -->
    <legend>Accessibility Form</legend>
      <table>
        <tr>
          <td>
            <label for="oFirstName">
            <!-- The acceleration key is underlined
                 for easy identification -->
              <span style="text-decoration: underline;">F</span>
                irst Name
            </label>
          </td>
        <td>
        <!-- The ID is defined for the LABEL,
             the NAME is provided for form submission -->
        <input tabindex="1" type="text" id="oFirstName"
               name="FirstName1" accesskey="F" />
        </td>
      </tr>
    </table>
  </fieldset>
</form>
```

テキストの操作

テキストは、フォームで重要な役割を果たしています。特に、メッセージ、データベース クエリ、コンテンツの更新などで重要です。

ユーザーの直接入力が可能なフォーム コントロールは、**input type="text"** 要素と **textArea** 要素です。textArea 要素は、複数行のメッセージの入力に適しています。textArea 要素のサイズは、

COLS 属性と **ROWS** 属性を使用して調整できます。ワードラップは **WRAP** 属性を使用して指定します。次のサンプル コードは、COLS、ROWS、および WRAP の各属性を使用した textArea 要素の構文を示しています。

```
<textarea cols="50" rows="10" wrap="hard">
  ...
</textarea>
```

フォームの送信

form 要素の送信を指定すると、その要素にあるコントロールのみが送信されます。フォームを使用する理由は多数ありますが、情報の送信先となる場所は数か所のみです。フォームの送信先となるものは、次のとおりです。

- クライアント側スクリプト、Java アプレット、または Microsoft ActiveX コントロール
- サーバー側スクリプト、サーブレット、またはコントロール
- 電子メール

フォーム送信を処理するスクリプトやアプリケーションは、**ACTION** 属性で指定します。このスクリプトは、Web サーバーやオペレーティング システムでサポートされているスクリプトやアプリケーションと同様で、クライアントで認識可能な結果 (一般的には HTML) を生成します。フォーム送信を処理するスクリプトを **CGI (Common Gateway Interface) スクリプト**と呼びます。CGI スクリプトを記述する言語としては、Perl、Visual Basic、Microsoft Visual C++、Java などがあります。

Web 作成者は、Web ドキュメントに埋め込むサーバー側スクリプトを使用することもできます。これには **ASP (Active Server Pages)** などがあります。サーバー側スクリプトは、Web サーバーで前処理され、結果が HTML で生成されます。たとえば、ASP では、定義済みスクリプト オブジェクトと、クライアント側スクリプトの実装を使用すると、他の CGI 対応言語を使用した場合と同じ結果が得られます。

フォームで想定している処理がクライアントで実行するものかサーバーで実行するものかに関係なく、次のいずれかの方法でフォームを送信できます。

- **input type="submit"** コントロールを使用する。
- **input type="image"** コントロールを使用する。
- **TYPE** 属性を **submit** に設定した **button** オブジェクトを使用する。
- スクリプトで **submit** メソッドを使用する。

スクリプト言語、コントロール、またはアプリケーションを使用するほか、**mailto** プロトコルを使用してフォームを送信することもできます。mailto を使用する場合は、form 要素の **METHOD** 属性を post に設定し、**ENCTYPE** 属性を text/plain に設定することをお勧めします。このプロトコルを使用するには、電子メール メッセージを mailto プロトコルで作成します。つまり、電子メール フォーム名を使用して作成します。次の一覧は、電子メール フォーム名とそれぞれの説明を示しています。

- **Recipient** は、電子メールの送信先のプライマリ アドレスを指定します。
- **Subject** は、電子メール ドキュメントの件名を指定します。
- **CC** は、同報電子メールの受信者を指定します。
- **BCC** は、受信者を表示しない同報電子メールの受信者を指定します。BCC フィールドで指定したアドレスは、同報受信者とプライマリ アドレスの受信者には表示されません。
- **Body** は、電子メール ドキュメントの本文を指定します。

電子メール パスをスクリプトに記述していない場合にのみ、これらの特定のフィールド名を使用することが適切です。次の例は、**onsubmit** イベントが発生したときのフォーム アクションを構築する電子メール フォームの概要を示しています。

- この電子メール フォームは、4 つのテキスト フィールド、1 つの textArea 要素、1 つの送信ボタン、および 1 つのリセット ボタンで構成した基本的なフォームです。input type=submit コントロールと input type=reset コントロールの代わりに **button** オブジェクトを使用しています。これにより、**ACCESSKEY** で指定したキーをボタンの値に反映できます。

```
<form method="get" onsubmit="fnSetAction()" name="oMailForm">
  <label for="oTo"><span>A</span>ddress:</label>
  <input type="text" id="oTo" name="recipient" value=""
    accesskey="A" />
  <br />
  <label for="oSubject"><span>S</span>ubject:</label>
  <input type="text" id="oSubject" name="subject" value=""
    accesskey="S" />
  <br />
  <!-- Repeat for additional fields -->
  <label for="oBody"><span>M</span>essage Body:</label>
  <textarea id="oBody" name="body" cols="40" rows="5"
    accesskey="M" ></textarea>
```

```
  <br />
  <button accesskey="E" type="submit">
    Send <span>E</span>mail</button>
  <br />
  <button accesskey="R" type="reset">
    <span>R</span>eset E-mail Form</button>
</form>
```

- **onsubmit** ハンドラは、電子メール メッセージがどのように **mailto** プロトコルに追加されるのかを示しています。mailto プロトコルは、フォーム フィールドに基づいた値の文字列から作成され、**ACTION** 属性に追加されます。これは、フォーム フィールド名が電子メール フィールド名と同じ場合は不要です。

```
<script type="text/javascript" language="javascript">
function fnSetAction() {
   if (oMailForm.recipient.value != "") {
      var sAction="mailto:" + oMailForm.recipient.value;
      if (oMailForm.subject.value != "") {
         sAction += "&Subject=" + oMailForm.subject.value;
      }
      if (oMailForm.body.value != "") {
         sAction += "&body=" + oMailForm.body.value;
      }
      oMailForm.action = sAction;
   }
   else {
      alert("Address required.");
   }
}
</script>
```

ACTION 属性で指定した CGI スクリプトにフォームを送信する場合は、フォーム データの場所を **METHOD** 属性で指定します。**get** メソッド (非推奨) を使用する場合、フォーム データは URL に追加され、4K の制限が適用されます。一方、post メソッドを使用する場合、フォーム データは HTTP ヘッダーと共に送信されるので、サイズの制限はありません。

次のサンプルは、スクリプトを使用してこの情報にアクセスする方法を示しています。ここでは、**oFirstName**、**oLastName**、および **oPhrase** という名前の各コントロールを持つフォームを考えます。

```
var aData = new Array();
function fnInit() {
  var sLocation = location.href;
  var sData =
    sLocation.substring(sLocation.indexOf("?") + 1, sLocation.length);
  aData = sData.split("&");
  for (var i = 0; i < aData.length; i++) {
    var sName = aData[i].substring(0, aData[i].indexOf("="));
    var sValue =
      aData[i].substring(aData[i].indexOf("=") + 1,
      aData[i].length);
    alert(sName + ' = ' + unescape(sValue));
  }
}
```

フォームを送信すると、フォーム データは URL に付加されます。たとえば、ACTION 属性で指定した CGI スクリプトのファイル名拡張子の後に、フォーム データが次のように付加されます。

```
?oFirstName=Gertrude&oLastName=Spencer&oPhrase=My%20favorite%20color
%20is%20plaid.
```

このフォーム データの最初の文字には疑問符 (?) が記述され、それに続く情報がフォーム データであることを示します。

URL からフォーム データを復元する手順は次のとおりです。

- 名前と値のペアを互いに分離します。
- 名前を値から分割します。
- 値をデコードします。

値をデコードすると、Web 作成者はスクリプトを介してフォーム データを使用できるようになります。

CSS セレクターについて

カスケーディング スタイル シート (CSS) によるスタイル シートの基本構成ブロックは、スタイル ルールです。HTML ページの要素のスタイルを設定できるように、セレクターを使用してそれら

の要素を選択します。セレクターがないと、ルールをどのように適用するのかを決定できません。ここでは、CSS 宣言構文の基本を紹介し、セレクターの使用方法について説明します。

スタイル ルール

スタイル ルールは、HTML ページに特定の要素をどのようにレンダリングするかをブラウザーに指定するステートメントです。ルールは、セレクターとそれに続く宣言ブロックで構成します。宣言ブロックは、中かっこ ({}) で囲んだすべての部分を指します (中かっこ自体も含みます)。次のスタイル ルールでは、すべての H1 要素の既定のレンダリングを変更しています。

```
H1 { color:blue; }
```

スタイルの宣言は、単一のプロパティとその値をコロン (:) で区切って記述したペアで構成します。宣言ブロックの中では空白が無視されるので、空白を使用してルールを見やすいように記述できます。複数の宣言はセミコロン (;) で区切ります。

ドキュメント ツリー

ドキュメント ツリーは、ソース HTML にエンコードした要素の階層です。このツリーの各レベルは、先祖と子孫のような関係にあります。ツリーの各要素には必ず親が 1 つだけ存在し (ただし、ルート要素には親がありません)、子や兄弟は複数存在する場合があります。ドキュメント ツリーの要素には、その要素のタイプ、階層での相対的な位置、または ID やクラスなどの属性に基づいてスタイル ルールが適用されます。

セレクターと連結子

シンプルなセレクターは、**タイプ セレクター**または**ユニバーサル セレクター**です。シンプルなセレクターの直後には、ID またはクラスと疑似クラス (またはそのどちらか) を任意の順序で記述できます。セレクターが要素に一致すると、スタイル ルールが適用されます。次に、シンプルなセレクターの例を示します。

```
#myID { color: red; }
#myID:hover { color: orange; }
P.myClass { color: green; }
P.myClass:hover { text-decoration: underline; }
```

複数のシンプルなセレクターを**連結子**で結合できます。連結子は、子孫、子、兄弟など、シンプルなセレクター間のコンテキスト上の関係を指定します。連結子とそれを囲むシンプルなセレクターとの間には空白を挿入できます (連結子が単独で使用されている場合も該当)。次のスタイル ルールを使用すると、ドキュメント ツリー内で互いに隣接する最上位レベルの見出しどうしの間でスペースの量を少なくすることができます。

```
H1 + H2 { margin-top: -5mm; }
```

スタイル シートのサイズを小さくするために、複数のセレクターをコンマで区切ってグループ化できます。コンマ文字自体は連結子ではありませんが、1 つの宣言ブロックを複数の種類の要素に適用する簡潔なメカニズムとして使用できます。

```
H1, H2, H3 { font-family: helvetica; }
```

疑似クラスと疑似要素

スタイル ルールは、ドキュメント構造での位置に基づいて要素に関連付けることが普通ですが、CSS では**疑似クラス**と**疑似要素**の概念を使用して、ドキュメント ツリー外の情報に基づく書式を使用できます。疑似要素を使用して、要素の部分を指定します (:first-letter や :first-line など)。一方、疑似クラスは、名前、属性、およびコンテンツ以外の特性 (:first-child、:visited、:hover など) で要素を分類します。

```
P:first-child:first-line { text-transform: uppercase; }
```

カスケードの順序と特異性 (specificity)

1 つの HTML ドキュメントで複数のスタイル シートを使用できる CSS では、スタイルの競合がどうしても発生します。複数の宣言を同じ要素に適用すると、宣言どうしで競合が発生する可能性があります。その場合は、次のアルゴリズムに従って最終的に適用するルールを決定します。

1. ターゲットの **@media** タイプに適用される宣言を、継承される宣言も含め、すべて探し出します。
2. 重み付けと宣言の作成元で宣言を並べ替えて、その順番で適用します。ユーザーのカスタム スタイル シートはブラウザーの既定値より優先されますが、作成者のスタイル シートはユーザーのカスタム スタイル シートより優先されます (ユーザーの宣言に !important が指定されている場合を除きます)。**!important** を指定した宣言は、重み付けが大きくなります。
3. セレクターの特異性 (specificity) で宣言を並べ替えて、その順番で適用します。特異性の高いセレクターの方が、汎用的なセレクターより優先されます。特異性による重み付けは a, b, c, d の 4 つの数字で表され、優先度が決定されます。

```
*             {}  /* a=0 b=0 c=0 d=0 -> specificity = 0,0,0,0 */
li            {}  /* a=0 b=0 c=0 d=1 -> specificity = 0,0,0,1 */
li:first-line {}  /* a=0 b=0 c=0 d=2 -> specificity = 0,0,0,2 */
ul li         {}  /* a=0 b=0 c=0 d=2 -> specificity = 0,0,0,2 */
ul ol+li      {}  /* a=0 b=0 c=0 d=3 -> specificity = 0,0,0,3 */
li.class      {}  /* a=0 b=0 c=1 d=1 -> specificity = 0,0,1,1 */
#myID         {}  /* a=0 b=1 c=0 d=0 -> specificity = 0,1,0,0 */
li#myID       {}  /* a=0 b=1 c=0 d=1 -> specificity = 0,1,0,1 */
style=""          /* a=1 b=0 c=0 d=0 -> specificity = 1,0,0,0 */
```

4. 指定された順序で宣言を並べ替えて、その順番で適用します。2 つのルールが同じ重みと特異性を持つ場合、最後に解析された方のルールを適用します (**@import** ルールで指定したスタイル シートが先に解析されます)。スタイル シートで後の方にあるセレクターは、競合が発生したときに使用されます。そのため、リンクの疑似クラス セレクターは必ず次の順序で使用する必要があります。

```
A:link {}
A:visited {}
A:hover {}
A:active {}
```

カスケードの順序と特異性の詳細は http://www.w3.org/TR/CSS21/cascade.html をご参照ください。

印刷とスタイル シート

style 要素および **link** 要素は、スタイル シートの出力デバイスを定義する **MEDIA** 属性をサポートします。MEDIA に指定できる値は、all, screen, tty, tv, projection, print, handheld, braille, aural です。既定値は screen なので、MEDIA 属性を設定していない場合、スタイル シートは必ず画面上のページおよび印刷するページに適用されます。print は、ページを印刷するときにスタイル シートを使用することを示します。この値を指定した場合は、画面に表示するドキュメントの外観にはスタイル シートが影響しません。all は、画面に表示するドキュメントと印刷するドキュメントの両方で書式設定することを示します。

印刷用のページ区切りの設定

ドキュメントのページ区切りの配置を制御するには、**page-break-before** 属性および **page-break-after** 属性を使用します。これらの属性は、ドキュメントを印刷するときに改ページする場所および印刷を再開するページ (左または右) を示します。

たとえば、次のドキュメントではスタイル シートに "page" クラスを定義し、それを使用してドキュメントのページ区切りを設定しています。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
  <title>Dynamic Styles: Page Breaking</title>
  <style type="text/css" media="print">
```

```
  div.page {
    page-break-before: always;
  }
  </style>
</head>
<body>
  content on page 1
  <div class="page">
    content on page 2
  </div>
</body>
</html>
```

ショートカット アイコンを Web ページに追加する方法

ショートカット アイコンを使用すると、独自のロゴや他の小さな画像を Windows Internet Explorer の [お気に入り] メニュー、アドレス バー、およびページ タブ (Internet Explorer 7 以降) に表示できます。

ここでは、Internet Explorer でショートカット アイコンがどのように扱われるのか、ショートカット アイコンを Web ページに追加する方法、およびトラブルシューティング手順について明します。

ショートカット アイコンについて

ショートカット アイコンは、Microsoft Internet Explorer 5 以降でサポートされています。有効なアイコンが設定された Web ページにユーザーが初めてアクセスすると、Internet Explorer にショートカット アイコンがダウンロードされます。

Internet Explorer 7 では、ユーザーが [お気に入り] メニューにサイトを追加すると、そのサイトのショートカット アイコンは、作成されるショートカット (.url) ファイルの Microsoft Windows NT ファイル システム (NTFS) 代替データ ストリームに保存されます (FAT ファイル システムでフォーマットされたハード ディスクでは、アイコンは [Temporary Internet Files] フォルダに保存されます)。

Internet Explorer 6 では、アイコンはユーザーのコンピュータの [Temporary Internet Files] フォルダに保存され、このアイコンに関する追加のメタデータがユーザーの履歴情報に保存されます。

正しいサイズと形式のアイコンの作成

Internet Explorer 用のショートカット アイコンは、16 x 16 ピクセル以上の正方形サイズである必要があります。高 DPI ディスプレイでは、Internet Explorer ではアイコンが表示可能スペースに合わせて拡大されることがあるため、16 x 16 ピクセルと 32 x 32 ピクセルの 2 種類の (および帯域幅に余裕がある場合はさらに大きいサイズ) のアイコンを作成することをお勧めします。

Web ページへのショートカット アイコンの関連付け

アイコンを作成したら、そのアイコンを Web ページに関連付ける必要があります。それには、次の 2 つの方法があります。

1 つ目の方法は、アイコンを **favicon.ico** という既定のファイル名でドメインのルート ディレクトリに保存することです (例 : www.microsoft.com/favicon.ico)。ユーザーがその Web ページに初めてアクセスすると、Internet Explorer によってこのファイルが自動的に検索されて、このアイコンがアドレス バー、[お気に入り] メニュー内のそのサイト リンクの横、およびページ タブに表示されます。

ショートカット アイコンを Web ページに関連付けるための 2 つ目の方法は、ページの **head** 要素に HTML コード行を追加することです。このコード行に含まれる **link** タグでは、アイコン ファイルの場所と名前を指定します。この link タグは、ページごとに追加できます。まずアイコンを favicon.ico 以外のファイル名で保存してから、次のコードをページの head 要素に追加します。

```
<head>
  <link rel="shortcut icon"
    type="image/ico"
    href="http://www.mydomain.com/myicon.ico"/>
  <title>My Title</title>
</head>
```

ページのショートカット アイコンの変更

ドメインのルートにある既定の favicon.ico ファイルのみをショートカット アイコンとして使用している場合に、このアイコンを変更した場合は、ユーザーが各自の履歴とキャッシュをクリアするまで、ユーザーには変更後のアイコンは表示されません。Internet Explorer は、favicon.ico が変更されたかどうかを検知できないため、取得済みのアイコンがない場合にのみ新しいアイコンを読み込みます。

この問題を回避するには、link タグを使用して、新しいショートカット アイコン用に異なるファイル名を使用します。

ショートカット アイコンのトラブルシューティング

Web ページ用に不適切なショートカット アイコンや既定のショートカット アイコンが表示されている場合は、次のことを試してください。

- そのショートカット アイコンが正しいサイズと形式であることを確認します。
- Internet Explorer のキャッシュと履歴情報をクリアします。どちらかが破損している場合は、不適切なショートカット アイコンが表示される可能性があります。
- Internet Explorer がショートカット アイコンを [Temporary Internet Files] フォルダに保管できることを確認します。Internet Explorer がキャッシュを保持しないように設定している場合は、Internet Explorer はアイコンを保管できないため、代わりに既定の Internet Explorer ショートカット アイコンが表示されます。

DHTML を使用したアクセス可能な Web ページの作成

ここでは、ダイナミック HTML (DHTML) を使用して、アクセス可能な Web ページを作成する方法について説明します。

アクセス可能な Web ページとは

アクセシビリティの問題は、ユーザーが Web ページと対話するときに発生する可能性があります。たとえば、ユーザーに視覚障害がある場合は、ページのコンテンツを見ることができません。また、マウスを操作できない場合は、ページを操作できません。アクセス可能な Web ページには、これらの作業を行うための別の手段があります。

Web ページの 1 つ目のアクセシビリティ要件は、コンテンツを視覚的な方法と視覚に頼らない方法の両方で表すことです。一般的な Web ブラウザーでは、視覚以外の感覚、たとえば聴覚 (ボイス シンセサイザー) や触覚 (ブライユ点字) を通じてユーザーにページを提示する、スクリーン リーダーと呼ばれるプログラムが使用されています。2 つ目の要件としては、ユーザーがマウス以外の手段、たとえばキーボード入力などを使用して Web ページのすべての要素にアクセスできる必要があります。

Web ブラウザーは HTML の階層構造を利用しているため、ほとんどの Web ページが自動的にアクセス可能になります。たとえば、a オブジェクトはユーザーがクリックできるページ上の領域を示しているため、ユーザーは Tab キーで任意の a オブジェクトに移動して Enter キーを押すことで、リンクをたどることが可能になります。a 要素には、Web ブラウザーからスクリーン リーダーに提供されるテキストも含まれており、スクリーン リーダーでリンクの説明のテキストを音で表すことができます。

アクセス可能な Web ページの作成に関するヒント

アクセス可能な Web ページを作成する際、Internet Explorer で Web ページをアクセス可能な方法で示すことに関して、課題に直面することがあるかもしれません。ここで示すヒントを実装すると、こうした問題の回避に役立つ可能性があります。

◆ ページのすべてのテキスト領域に、マウス クリックに反応するアンカーを使用します。

DHTML を使用すれば、ほとんどすべてのテキスト要素にクリック イベントを関連付けることができますが、アクセシビリティの理由から、このイベントを a 要素にのみ関連付けることが考えられます。アンカーによって、キーボードユーザーは、キーボードを使用してページ上の領域を移動できるようになります。また、アンカーを使用してスクリーンリーダーが読み取れるヒントを提供し、Web ページ上のどの部分が対話可能であるかを識別できるようにすることもできます。

◆ テキストを組み込みコントロールに関連付けるには、**label** オブジェクトを使用します。

HTML 3.2 の仕様では、テキストをラジオ ボタンやテキスト ボックスなどの組み込みコントロールに関連付ける手段が手供されていません。a 要素は対応する終了要素 (/a) と中に含めるテキストを持ちますが、input 要素には終了要素がありません。終了要素がないことで、スクリーン リーダーでは、ユーザーにそのコントロールを説明するために使用されるテキストを見つけることが難しくなります。HTML 4.01 から label オブジェクトが導入されており、このオブジェクトを使用することでテキストを組み込みコントロールなどの別の HTML 要素に関連付けることができます。

label オブジェクトをラジオ ボタンに関連付けるには、次の HTML 構文を使用します。

```
<form ... >
  <input type="radio" id="FirstButton" name="radio1" />
  <label for="FirstButton">Text for the first radio button</label>
  <br />
  <input type="radio" id="SecondButton" name="radio1" />
  <label for="SecondButton">Text for the second radio button</label>
</form>
```

◆ カスケーディング スタイル シート (CSS) で配置機能を使用する場合は、座標を "em" 単位で指定します。

Internet Explorer で CSS 配置を使用すると、Web ページ上で要素を正確に配置することができます。ただし、ユーザーが既定のフォント サイズを変更したり、ページのフォントサイズを変更したりした場合は問題が発生する可能性があります。絶対位置指定のオブジェクトが適切に作成されないと、オブジェクトのサイズが大きくなり互いに重なることがあります。この問題を避けるには、位置指定するオブジェクトの座標を "em" 単位で指定します。絶対位置指定の div 要素を指定するには、次の構文を使用します。

```
<div style="position: absolute; left: 10em; top: 12em; height: 5em;
width: 5em">Here is some positioned text!</div>
```

◆ アクセシビリティに関して、位置指定の HTML コードをテストします。

たとえば、em 単位で位置指定された HTML 要素を使用したページを作成する場合、Microsoft Windows の表示設定を小さいフォントと大きいフォントで切り替えてみてください。また、[インターネット オプション] ダイアログ ボックスの [ユーザー補助] オプションをクリックし、[Web ページで指定されたフォント サイズを使用しない] をオンにします。次に、[表示] メニューで Internet Explorer のフォント サイズを [最大] に設定します。最後のテストとして、コントロール パネルの [ユーザー補助のオプション] の [画面] タブで、ハイ コントラスト モードを有効にします。これらの各設定により、Web ページの要素のフォント サイズが、障害を持つ人々が一般的に使用しているようなサイズになります。

◆ クライアント側イメージ マップを使用して、キーボードの操作を簡単にします。

クライアント側イメージ マップでは、定義された各領域をユーザーが Tab キーで移動できるため、サーバー側イメージマップよりもアクセスが容易です。キーボード操作をさらに簡単にするには、イメージマップ内の定義された各領域にアンカーを追加します。このようにすることで、どのユーザーも Tab キーを使用して目的のリンクに移動し、Enter キーを押すだけで操作できるようになります。

アクセス可能な HTML 要素

イメージ、テキスト、リンクなどの一部の HTML 要素にはアクセス可能ですが、アクセスできない要素もあります。HTML ドキュメント内のアクセス可能な各要素（タグ）は、ドキュメントのアクセシビリティの階層に表示されます。

アクセス可能な主な要素は次のとおりです。

a	**applet**	**area**	**body**	**button**	**embed**
frame	**frameset**	**iframe**	**img**	**input**	**object**
select	**table**	**td**	**th**	**textarea**	

要素の **TABINDEX** 属性の値を指定することで、アクセスできない要素をアクセス可能にすることもできます。

アクセスできない主な要素は次のとおりです。

div	**span**	**b**	**i**	**u**

JavaScript の概要

JavaScript は ネットスケープ コミュニケーションズが開発したインタプリタ方式のスクリプト言語です。まず、Netscape Navigator 2.0 に搭載され、その後、1996 年に Internet Explorer 3.0 へ搭載されて急速に一般に浸透しました。現在では、クライアント サイド スクリプトと言えば、JavaScript を指すと考えても良いでしょう。

JavaScript を使用することで、フォームの入力検証やテキストの加工、クッキーの読み込み、ウィンドウ操作など、ブラウザーに関わる操作の全般を制御できます。

JavaScript の動作

クライアント サイド スクリプトの動作を次の図に示します。クライアント（Web ブラウザー）からサーバーへ HTML の要求を送信すると、サーバーはクライアント（Web ブラウザー）へ HTML を送信します。クライアント（Web ブラウザー）では、受け取った HTML の中に記述されている JavaScript を解釈・実行し、その結果を表示します。

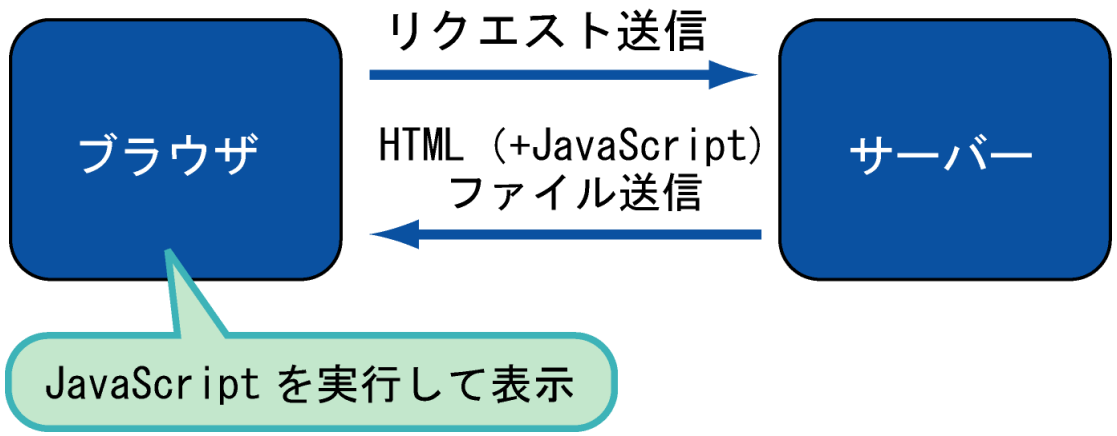

図. *JavaScript の動作*

JavaScript の記述

JavaScript のソースコードを記述するには、次の 2 種類があります。

- HTML のソースコードに直接記述する
- 外部のソースファイルをインポートする

HTML のソースコードに直接記述する場合は、<script></script>タグで囲み、type 属性を text/javascript に指定します。

ポップアップウィンドウに「Hello World」と表示させる例を以下に示します。

```
<script type="text/javascript">
  alert ("Hello World");
```

```
</script>
```

また、外部のソース ファイルを読み込ませるためには、以下のように src 属性にアドレスを指定します。

```
<script type="text/javascript"
        src="http://www.example.com/example.js">
</script>
```

DHTML（Dynamic HTML）

通常の静的な HTML 内にスクリプトを記述したりスタイル シートを適用したりすることで、ブラウザー上で動的なページを表現することができます。このように構築されたページは HTML に対して動的（Dynamic）な仕組みを付与するという意味で、**DHTML** と呼ばれます。

スタイル シートには **CSS（Cascading Style Sheets）**を用いるのが一般的です。スタイル シートを利用することで、HTML には文書の構造のみを記述し、文字のフォントやサイズの変更などといったデザイン部分の記述を切り離すことが可能になります。構造とデザインとを分離することで、レイアウトを複数のページで共有することができるようになり、ページごとのデザイン設定が不要になります。動的な文書の操作には、スタイル シートの併用は欠かせません。

もっとも DHTML は、ブラウザーやバージョンの違いによる互換性が低かったため、1998 年 10 月、W3C（World Wide Consortium）は HTML 文書や XML 文書をアプリケーションから利用するための標準的な API として **Document Object Model （DOM）** を勧告しました。DOM は、JavaScript から HTML 文書の要素や属性の取得、更新、追加、削除を行うための標準的な手段を提供します。最近では、クライアント上でのコンテンツ操作には DOM を利用するのが一般的です。

AJAX の概要

AJAX は、JavaScript の HTTP 通信機能を用いることで、より Windows アプリケーションライクな使い勝手の Web アプリケーションを実現できます。AJAX を利用することで、サーバーとの非同期通信が可能になるので、Web アプリケーションにありがちな、通信するたびに画面が切り替わることがなくなり、エンドユーザーの作業が中断されることが少なくなるというメリットがあります。たとえば、ページの再読み込みを行わずに地図上を移動できる Bing Maps（http://www.bing.com/maps/）のようなアプリケーションは、AJAX 技術を利用した好例です。

開発側のメリットもあります。まず、AJAX 技術は JavaScript、DOM を中心とした技術なので、プラグインなどを導入しなくても動作します。プラグインが不要ということは配布を容易にします。また、特定の開発環境に依存することなく、ブラウザーの標準機能を用いたシステム開発が可能です。

AJAX の動作

AJAX の動作を次の図に示します。クライアント（Web ブラウザー）からサーバーへ HTML の要求を送信すると、サーバーがクライアント（Web ブラウザー）に HTML を送信するという動作は通常のサイトにアクセスした場合と同様です。ここで、JavaScript は XMLHttpRequest というオブジェクトを用いてサーバーへデータ要求を送信します。サーバー側では処理を非同期に実行し、その結果をブラウザーに送信します。ブラウザーでは受信データを加工して、画面の必要な部分だけを適宜更新します。

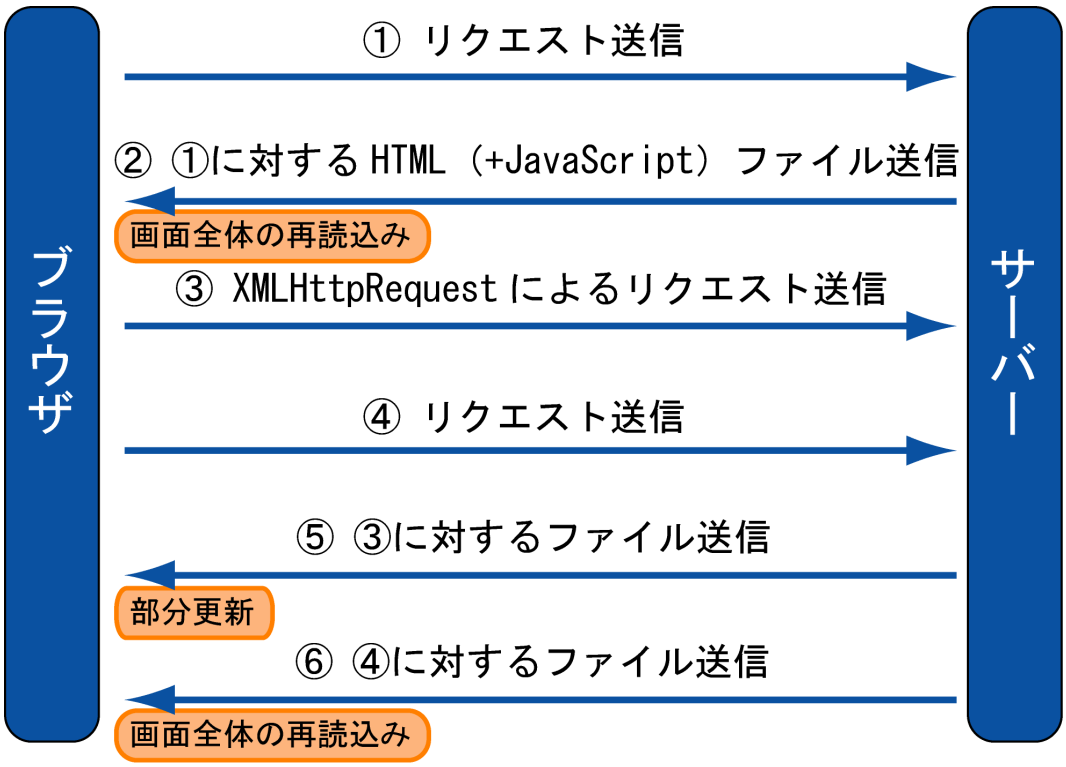

図. *AJAX の動作*

JavaScript におけるライブラリ

AJAX の普及に伴い、JavaScript の利用頻度が高まるにつれ、AJAX 対応の JavaScript ライブラリの活用は欠かせなくなりつつあります。JavaScript のライブラリはさまざまな機能を関数やクラスとしてまとめたものです。ライブラリにはいろいろありますが、共通した機能としては以下のようなものが挙げられます。

- 複数ブラウザー間の種類やバージョン差異の吸収（クロスブラウザー対応）
- AJAX 操作の簡略化

◆ 独自の GUI 構築（HTML、CSS、JavaScript の組合せ）によるサイトの表現力の向上

現在よく使われている JavaScript の主なライブラリは以下の通りです。

◆ Prototype （http://prototypejs.org/）

◆ jQuery （http://jquery.com/）

◆ Yahoo UI （http://developer.yahoo.com/yui/）

◆ Dojo （http://www.dojotoolkit.org/）

これらのライブラリを使用する際には、内容を書き換えても問題ないかどうかや、再配布についての規定（ライセンス）や複数バージョンの混在や同時利用の可否などにも注意してください。

jQuery の概要

上記で紹介したライブラリの中では、**jQuery** が最近注目を集めています。jQuery は以下のような特長があります。

◆ CSS セレクターを使用できる

◆ コア部分の容量が小さく、動作が軽い

◆ シンプルな記述により短いコードで多くの処理を簡単に実行できる

◆ プラグインが充実している

たとえば、test1 という id を指定して、その要素の文字の色を赤く変化させるには、次のようなコードを書きます。

```
$("#test1").css("color", "red");
```

セレクター構文を利用することで、要素の特定、操作が直感的に行えるのが jQuery の最大の特長です。

2010 年 2 月にはパフォーマンスが改善された jQuery 1.4.2 が公開されました。Safari 3.2、Safari 4、Firefox 2、Firefox 3、Firefox 3.5、IE 6、IE 7、IE 8、Opera 10.10、Chrome といった主なブラウザーでの検証が行われています。

マイクロソフトでも 2008 年 9 月から jQuery の正式サポートを開始しており、Visual Studio 2008（Visual Web Developer 2008 Express も含む）で Visual Studio 用のパッケージを適用させるとインテリセンス機能を利用できます。また、次期 ASP.NET AJAX 4.0 では、jQuery との統合が予定されており、ASP.NET 環境で JavaScript プログラミングを行う際に jQuery を使うための環境が整いつつあります。

また、**jQuery Plugins**（http://plugins.jquery.com/）では、jQuery の機能を拡張するための数多くのプラグインが公開されています。**jQuery UI**（http://jqueryui.com/）はカレンダーや、タブ、ドラッグ＆ドロップ操作などといったユーザー インターフェイスを持つコンポーネントです。これらは別途ダウンロードが必要で、Web ページから参照可能なディレクトリに展開して使用します。

利用方法

Web ページで jQuery を利用するには head 要素に以下のスクリプトを記述します。

```
<script type="text/javascript" src="jquery-x.x.x.js"></script>
```

※ src="jquery-x.x.x.js" の部分は jQuery のバージョンによって適宜変更してください

Visual Studio で jQuery を利用する

Visual Studio では、以下の手順で jQuery のインテリセンスを有効にできます。

1. ホット フィックスを適用する

以下のサイトから VS90SP1-KB958502-x86.exe をダウンロードし、実行します。

```
http://code.msdn.microsoft.com/KB958502/Release
/ProjectReleases.aspx?ReleaseId=1736
```

※ Visual Studio 2010 ではこのホットフィックスは不要です。

2. jquery-X.X.X.js と jquery-X.X.X-vsdoc.js をプロジェクトの同一の場所に配置する

jQuery を以下のサイトからダウンロードします。

http://docs.jquery.com/Downloading_jQuery

jQuery ではスクリプトからコメントや不要な改行、空白を除いた圧縮パッケージ（Minified 版）、コメントなどを含む非圧縮パッケージ（Uncompressed 版）の他に xxxx-vsdoc.js という Visual Studio 上で jQuery 用 IntelliSense 機能を有効にするためのファイルが用意されています。この中から、必要に応じて非圧縮パッケージかもしくは圧縮パッケージ（jquery-X.X.X.js）のどちらかと xxxx-vsdoc.js の 2 種類のファイルをプロジェクトに配置します。

3. 利用するページにスクリプト ファイルへの参照を追加する

以下のように .aspx ファイルに jQuery 用 IntelliSense の参照を追加します。

```
<script src="jquery-X.X.X.js" type="text/javascript"></script>
```

サーバー サイド テクノロジ

ASP.NET の概要

ASP.NET は、シンプルな Web サイトからエンタープライズクラスの Web アプリケーションまで、さまざまな規模の Web 構築を最小限のコーディングで実現するための統合 Web 開発モデルです。ASP.NET は **.NET Framework** の一部であり、ASP.NET アプリケーションのコードを作成する際は、.NET Framework 内のクラスにアクセスすることになります。アプリケーションのコードは、Microsoft Visual Basic や C# など、共通言語ランタイム (CLR) と互換性がある任意の言語で作成できます。これらの言語を使用して、共通言語ランタイム、タイプ セーフ、継承などのメリットを活用する ASP.NET アプリケーションを開発できます。

ここでは、ASP.NET と、ASP.NET Web アプリケーションの開発環境である Visual Studio および Visual Web Developer Express の機能について説明します。

Visual Studio および Visual Web Developer Express

Visual Studio と Visual Web Developer Express Edition (無償) は、ASP.NET Web アプリケーションを作成するための完全な機能を備えた開発環境です。これらは以下の機能を提供します。

- **Web ページ デザイン** —— HTML 検証機能を備えた HTML 編集モードと WYSIWYG 編集機能が組み込まれた強力な Web ページ エディターが用意されています。
- **ページ デザイン機能** —— マスター ページを使用した一貫性のあるサイト レイアウトと、テーマやスキンを使用した一貫性のあるページ外観を実現できます。
- **コード編集** —— Visual Basic や C# で動的 Web ページのコードを記述できる高機能なコードエディターを利用できます。
- **配置** —— テスト用ローカル Web サーバーや IIS (Internet Information Services) など、ホストしているサイトにサイトを公開するためのツールが用意されています。
- **マルチターゲット** —— Web アプリケーションのターゲットを特定のバージョンの .NET Framework で開発することができます。
- **IntelliSense** —— IntelliSense により、サーバー コードやクライアント コードを作成および変更する際に、簡単に言語リファレンスにアクセスできます。
- **デバッグ** —— 統合されたデバッガを使用して、ページをテストできます。
- **Web ページの個別化** —— ユーザー固有の設定を保存できるユーザー プロファイルを作成できます。
- **状態管理** —— ASP.NET の状態管理機能を使用することで、顧客情報やショッピング カートの中身など、複数のページ要求にまたがる情報を保存できます。

- **グローバリゼーション** —— ユーザーの使用言語およびロケールと一致するリソース ファイルからテキストが自動的に読み込まれるようにページを構成できます。
- **Web アプリケーション プロジェクト** —— Web アプリケーション プロジェクト モデルを使用することで、Web サイトを Bin フォルダ内の単一アセンブリにコンパイルし、プロジェクト リソースを明示的に定義できます。
- **WCF サービス アプリケーション プロジェクト** —— WCF サービス アプリケーション テンプレートは、サービス開発のための基本的なクラス構造を提供します。
- **クラス ライブラリ プロジェクト** —— クラス ライブラリ テンプレートを使用して、他のプロジェクトと共有できる再利用可能なクラスおよびコンポーネントをすばやく作成できます。
- **Web サービス アプリケーション プロジェクト** —— ASP.NET Web サービス アプリケーション テンプレートを使用して、Web サービスの内部動作のインフラストラクチャを作成できます。

Visual Web Developer Express について

Visual Web Developer Express は、さまざまな Web アプリケーションを容易に開発することができる使いやすい**無償の開発環境**です。Visual Web Developer Express は、主に Web アプリケーションの作成に必要なツールを提供する合理化されたインターフェイスを備えており、Visual Studio 製品版の Web 開発機能とほぼ同等の機能を使用することができます。

Visual Web Developer Express のインストール

Microsoft Web Platform Installer (Web PI) を使用して Visual Web Developer Express をインストールできます。Microsoft Web Platform Installer は、Microsoft Web Platform コンポーネントのダウンロード、インストール、サービスを簡単に行うことができる無料ツールです。これらのコンポーネントには、Visual Web Developer Express、インターネット インフォメーション サービス (IIS)、SQL Server Express、.NET Framework などがあります。

Web Platform Installer は以下のサイトからダウンロードできます。
Microsoft Web Platform: http://www.microsoft.com/web/

Web サイトと Web アプリケーション プロジェクトの作成と管理

Visual Studio や Visual Web Developer Express を使用して、Web サイト、Web アプリケーション、Web サービス、AJAX サーバー コントロールなど、さまざまなタイプの ASP.NET プロジェクトを作成できます。

Web サイトは、動的にコンパイルされるようにデザインされたプロジェクトです。Web サイト ファイルはそのままホスティング サーバーに配置でき、その Web サイトでページの 1 つが最初にアクセスされたときにコンパイルされます。Web サイト プロジェクトでは、クラス ファイルをソー

ス コード (.cs ファイルまたは .vb ファイル) として App_Code フォルダ内に保持し、これらのファイルも動的にコンパイルされます。サイトをプリコンパイルすることもできます。

Web アプリケーション プロジェクトは、プロジェクトが開発コンピュータでコンパイルされ、バイナリ形式でサーバーに配置されるようにデザインされています。IIS (Internet Information Services) での実行時、ASP.NET がバイナリをロードしてコードを実行します。

Web ページ と Web サーバー コントロール

Visual Studio や Visual Web Developer Express では、ASP.NET Web ページ (.aspx) と HTML ページ (.htm, .html) の両方を作成できます。ASP.NET Web ページは動的なページです。ASP.NET Web ページには、ASP.NET Web サーバー コントロールと、サーバー上の ASP.NET によって処理されるコードが含まれます。サーバーでの処理中、このコントロールとコードによってブラウザーに HTML (またはその他のマークアップ) として送信される出力が生成されます。Visual Studio や Visual Web Developer Express には、Web ページ デザイナーが含まれています。

カスタム レイアウトおよび外観

テンプレートのように機能するマスターページを使用して、**カスタム ページ レイアウト**を作成できます。レイアウト全体をマスターページで作成し、次にそのマスターページにマージするコンテンツ ページを作成します。Web サイトのページのカスタムな外観を作成するために、テーマを使用することができます。テーマを使用することで、コントロールやページの色、フォント、およびその他の特性を定義できます。

Web コントロール

Web ページ開発を容易にするために、**ASP.NET Web サーバー コントロール**を使用できます。Web サーバー コントロールは、テキスト ボックス、ボタン、チェック ボックス、メニューなど、使い慣れた機能をページに提供します。

ASP.NET は、Web ページで実行できるさまざまなタスク用の Web サーバー コントロールを備えています。次のコントロールが含まれます。

- **標準コントロール** —— ASP.NET Web ページに基本的な機能と複雑な機能の両方を追加できます。標準コントロールには、ボタン、画像、テキスト ボックス、チェック ボックス、ハイパーリンク、リスト ボックスなどがあります。
- **データ コントロール** —— Web ページをデータベースや XML ファイルなどのさまざまなデータ ソースに接続できます。データコントロールでは、ページのデータをテーブル形式やその他の形式で表示でき、ユーザーがデータを編集できます。
- **ナビゲーション コントロール** —— Web ページにさまざまな種類のメニューを追加できます。たとえば、静的メニューおよびフライアウト メニュー、ツリービュー、およびナビゲーション パス などがあります。

- **入力検査コントロール** —— ユーザー入力をチェックする方法を提供します。必須フィールド、値の範囲、最小値と最大値、および特定のパターンをチェックできます。
- **ログイン コントロール** —— 簡単にログオンフォームを作成したり、ユーザーを認証したりすることができます。また、ユーザーが Web サイトに登録したり、パスワードを回復または変更したりすることができるログインコントロールも使用できます。
- **Web パーツ コントロール** —— ユーザーがブラウザーで ASP.NET Web ページをカスタマイズできます。
- **AJAX 拡張コントロール** —— 非同期ポストバックなどの AJAX 機能を使用して、Web サイトを拡張できます。

ASP.NET コンパイラ

すべての ASP.NET コードはコンパイルされます。これには、厳密な型指定、パフォーマンス最適化、事前バインドなどのさまざまなメリットがあります。コンパイルされた ASP.NET コードは、共通言語ランタイム (CLR) によって、さらにネイティブコードにコンパイルされパフォーマンスの向上が図られます。

セキュリティ インフラストラクチャ

ASP.NET は、ユーザーアクセスを認証し、セキュリティ関連タスクを実行するための高度なセキュリティ インフラストラクチャを提供します。IIS から提供される Windows 認証を使用してユーザーを認証したり、ASP.NET フォーム認証と ASP.NET メンバーシップを使用して独自のユーザーデータベースで認証を管理したりすることができます。さらに、Windows グループを使用したり、ASP.NET の役割に基づく独自のカスタム ロール データベースを使用したりして、Web アプリケーションの機能や情報に対する許可を管理できます。

ASP.NET には、メンバーシップ、ロール、ログインなどのサーバーコントロールが用意されています。これらを使用して、ほとんどコードを記述せずに Web サイトに認証 (ログイン) および承認を追加できます。また、ユーザーがサイトに登録したり、ユーザーの資格情報を自動的にチェックしたりするログイン ページを作成できます。そして、ログインしたユーザーのみがページを表示できるように Web ページを保護できます。1 つのページで、ログインしたユーザーと匿名ユーザーにそれぞれ異なる情報を提供できます。

状態管理機能

ASP.NET には、顧客情報やショッピング カートの中身などの情報を異なるページ間の要求で格納しておくことができる**状態管理機能**が用意されています。アプリケーション固有、セッション固有、ページ固有、ユーザー固有、および開発者定義の情報を保存および管理できます。この情報は、ページ内のすべてのコントロールから切り離すことができます。

また、ASP.NET は**分散状態機能**を提供します。これは、1 台のコンピュータまたは複数のコンピュータ上にある同じアプリケーションの複数のインスタンスにまたがって状態情報を管理する機能です。

ASP.NET 構成

ASP.NET アプリケーションは、Web サーバー、Web サイト、または個別のアプリケーションに対して**構成設定**を定義できる構成システムを使用しています。構成設定は、ASP.NET アプリケーションの配置時に行うことができます。また、構成設定の追加や変更はいつでも可能で、稼働中の Web アプリケーションやサーバーへの影響は最小限です。

状態監視機能とパフォーマンス機能

ASP.NET には、ASP.NET アプリケーションの状態やパフォーマンスを監視するための機能が組み込まれています。**ASP.NET Health Monitoring** は、アプリケーションの動作状態やエラー状態に関する情報を重要なイベントとしてレポートします。これらのイベントは、診断と監視を兼ねており、何をどのようにログに記録するかをきわめて柔軟に設定できます。

アプリケーションから利用できるパフォーマンスカウンターについては、次の 2 つのグループが ASP.NET でサポートされています。

- ASP.NET システム パフォーマンス カウンター グループ
- ASP.NET アプリケーション パフォーマンス カウンター グループ

デバッグのサポート

ASP.NET は、実行時デバッグ インフラストラクチャを活用して、言語横断的/コンピュータ横断的なデバッグをサポートします。共通言語ランタイムによってサポートされるすべての言語とスクリプト言語の他に、管理オブジェクトと非管理オブジェクトの両方をデバッグできます。

XML Web サービス フレームワーク

ASP.NET は **XML Web サービス**をサポートしています。XML Web サービスは、アプリケーションどうしが HTTP/XML メッセージングなどの標準を使用してファイアウォールを越えて情報を交換するためのビジネス機能が入ったコンポーネントです。

拡張可能なホスティング環境とアプリケーション ライフサイクル管理

ASP.NET には、拡張可能なホスティング環境が組み込まれています。これは、ユーザーがアプリケーションのリソース (ページなど) に最初にアクセスしたときから、アプリケーションがシャットダウンされるまでのアプリケーションのライフサイクルを制御します。

拡張可能なデザイナー環境

ASP.NET には、Visual Studio などのビジュアル デザイン ツールで使用される Web サーバー コントロールのデザイナーを作成するためのサポートが強化されています。デザイナーは、コントロールのデザイン時ユーザーインターフェイスです。

データ駆動 Web ページ

ASP.NET は、さまざなま **データ ソース コントロール** をサポートします。これらのコントロールは、さまざまな種類のデータ ソースに接続して通信するために必要なすべてのタスクを実行します。データ ソース コントロールの利点は、すべての ASP.NET コントロールに対して、データ バインドのための一貫したインターフェイスを提供することです。

さらに、**統合言語クエリ (LINQ)** を使用してデータベースやインメモリ データ ソースにクエリを実行できます。LINQ は、クエリ操作を C# および Visual Basic で直接定義できるクエリ構文です。また Visual Studio や Visual Web Developer Express は、データ クラスをすばやく作成および編集できるオブジェクト リレーショナル マッピング デザイナーを備えています。

Visual Studio や Visual Web Developer Express は、ASP.NET Web ページに追加してデータを表示するためのさまざまなデータコントロールもサポートします。たとえば、GridView、DetailsView、FormView、ListView、DataList、および Repeater コントロールがあります。この各コントロールを使用することで、データをさまざまな方法で表示できます。さらに、データ テーブルを直接ページ内にドラッグすることができます。

また、SQL Server Express を使用することでデータをローカル データベースに格納できます。これはオプションのダウンロードです。

MVC パターンに基づいた Web アプリケーション

ASP.NET MVC は、保守が容易な標準ベースの Web アプリケーションを構築する Web 開発者にとって便利です。Model-View-Controller (MVC) パターンを使用することで、アプリケーション レイヤー間の依存性が減少するためです。また、ASP.NET MVC は、ページマークアップに対する完全な制御を提供します。ASP.NET MVC は、テスト駆動型開発 (TDD) をサポートすることで、Model と Controller のテストを容易におこなえます。

ASP.NET Dynamic Data

ASP.NET Dynamic Data は、データ駆動 ASP.NET Web アプリケーションを簡単に作成するためのフレームワークです。実行時にデータ モデル メタデータを自動的に発見し、そこから UI 動作を導出します。

ASP.NET Web ページのコード モデル

ASP.NET Web ページは 2 つの部分から構成されています。

- **ビジュアル要素** —— マークアップ、サーバー コントロール、静的テキストなどです。
- **ページのプログラミング ロジック** —— イベント ハンドラ、その他のコードなどです。

ASP.NET では、ビジュアル要素とコードを管理するために 2 つのモデル、単一ファイル ページ モデルと分離コード (コードビハインド) ページ モデルが用意されています。2 つのモデルは同じように機能するため、両方のモデルで同じコントロールとコードを使用できます。

単一ファイル ページ モデル

単一ファイル ページ モデルでは、ページのマークアップとそのプログラミングコードは同一の .aspx ファイル内にあります。プログラミングコードは、ASP.NET が実行する必要があるコードとしてマークするための **runat="server"** 属性を持つ **script** ブロックに記述します。

1 つの Button コントロールと 1 つの Label コントロールを含む単一ファイルページについて、コード例を次に示します。

<Visual Basic>

```
<%@ Page Language="VB" %>
<script runat="server">
  Protected Sub Button1_Click(ByVal sender As Object, _
      ByVal e As System.EventArgs)
    Label1.Text = "Clicked at " & DateTime.Now.ToString()
  End Sub
</script>
<html>
<head>
  <title>Single-File Page Model</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:Label ID="Label1" runat="server" Text="Label">
      </asp:Label>
      <br />
      <asp:Button ID="Button1" runat="server" Text="Button"
        OnClick="Button1_Click"></asp:Button>
```

```
    </div>
  </form>
</body>
</html>
```

<C#>

```
<%@ Page Language="C#" %>
<script runat="server">
void Button1_Click(Object sender, EventArgs e)
{
  Label1.Text = "Clicked at " + DateTime.Now.ToString();
}
</script>
<html>
<head>
  <title>Single-File Page Model</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:Label id="Label1" runat="server" Text="Label">
      </asp:Label>
      <br />
      <asp:Button id="Button1" runat="server" Text="Button"
        Onclick="Button1_Click"></asp:Button>
    </div>
  </form>
</body>
</html>
```

script ブロックには、そのページで必要とされるすべてのコードを含めることができます。コードは、ページ内のコントロールのイベント ハンドラ、メソッド、プロパティ、およびクラスファイル内で通常使用するその他のコードで構成することができます。実行時、単一ファイルページは、Page クラスから派生するクラスとして扱われます。ページには、明示的なクラス宣言は含まれません。代わりに、コントロールをメンバとして含む新しいクラスをコンパイラが生成します（すべてのコントロールがページメンバとして公開されるわけではありません。一部は他のコントロールの子です)。ペー

ジ内のコードはクラスの一部になります。たとえば、作成するイベントハンドラは、派生した Page クラスのメンバになります。

分離コード (コード ビハインド) ページ モデル

分離コード (コード ビハインド) ページ モデルでは、マークアップを 1 つのファイル (.aspx ファイル) に、プログラミングコードを別のファイルに記述します。コードファイルの名前は、使用するプログラミング言語によって異なります。

分離コード モデルでは、前のセクションで単一ファイルページ用に使用したサンプルは 2 つの部分に分けられます。マークアップが 1 つのファイル (この例では SamplePage.aspx) に格納され、次のコード例に示すように単一ファイルページと同様になります。

<Visual Basic>

```
<%@ Page Language="VB" CodeFile="SamplePage.aspx.vb"
    Inherits="SamplePage" AutoEventWire="false" %>
<html>
<head>
  <title>Code-Behind Page Model</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:Label id="Label1" runat="server" Text="Label">
      </asp:Label>
      <br />
      <asp:Button id="Button1" runat="server" Text="Button"
        OnClick="Button1_Click"></asp:Button>
    </div>
  </form>
</body>
</html>
```

<C#>

```
<%@ Page Language="C#" CodeFile="SamplePage.aspx.cs"
    Inherits="SamplePage" AutoEventWireup="true" %>
<html>
<head>
```

```
  <title>Code-Behind Page Model</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:Label id="Label1" runat="server" Text="Label">
      </asp:Label>
      <br />
      <asp:Button id="Button1" runat="server" Text="Button"
        OnClick="Button1_Click"></asp:Button>
    </div>
  </form>
</body>
</html>
```

単一ファイル モデルと分離コード モデルでは、.aspx ページに関して 2 つの相違点があります。分離コード モデルには、runat="server" 属性を持つ script ブロックはありません (ページにクライアント側のスクリプトを記述する場合は、ページに runat="server" 属性を持たない script ブロックを含めることができます)。2 つ目の違いは、分離コード モデル内の @Page ディレクティブが、外部ファイル (SamplePage.aspx.vb または SamplePage.aspx.cs) とクラスを参照する属性を含むという点です。これらの属性により、.aspx ページがページのコードにリンクされます。

コードは別のファイルに格納されています。単一ファイル ページの例と同じ Click イベント ハンドラを含む分離コード ファイルについて、コード例を次に示します。

<Visual Basic (.vb)>*

```
Partial Class SamplePage
  Inherits System.Web.UI.Page
  Protected Sub Button1_Click(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles Button1.Click
    Label1.Text = "Clicked at " & DateTime.Now.ToString()
  End Sub
End Class
```

<C# (.cs)>*

```
using System;
using System.Web;
using System.Web.UI;
```

```
using System.Web.UI.WebControls;
public partial class SamplePage : System.Web.UI.Page
{
  protected void Button1_Click(object sender, EventArgs e)
  {
    Label1.Text = "Clicked at " + DateTime.Now.ToString();
  }
}
```

分離コード ファイルでは、既定の名前空間にすべてのクラス宣言が含まれます。ただし、クラスは **partial** キーワードで宣言されます。これは、クラス全体が 1 つのファイルに含まれているわけではないことを示します。代わりに、コンパイラはページ実行時に、.aspx ページと、@Page ディレクティブで参照されるファイルを読み込み、それらを単一のクラスにアセンブルしてから、1 つのまとまりとして単一のクラスにコンパイルします。partial クラス ファイルは、ページの Page クラスから継承します。

ページ モデルの選択

単一ファイル モデルと分離コード モデルは、機能的には同じです。実行時、これらのモデルは同じように実行され、パフォーマンス上の違いはありません。したがって、ページ モデルの選択は別の要因に依存します。

単一ファイル ページの利点

原則として、単一ファイル モデルはコードがページ内のコントロールのイベント ハンドラを中心に構成されているページに適しています。

単一ファイル ページ モデルには、次のような利点があります。

- コードが多くないページの場合は、コードとマークアップを同じファイルに保持することの利便性が分離コード モデルの他の利点を上回ります。
- 単一ファイル モデルを使用して記述されたページは、ファイルが 1 つしかないため、配置や他のプログラマへの送信が多少簡単になります。
- 単一ファイル ページは、ファイル間に依存関係がないので名称の変更が簡単です。
- ページが単一のファイルで完結しているため、ソース コード管理システムでのファイル管理が多少簡単になります。

分離コード ページの利点

分離コード ページには、大量のコードを含む Web アプリケーションや、複数の開発者が Web サイトを作成している場合に適しているという利点があります。

分離コード モデルには、次のような利点があります。

- 分離コード ページでは、マークアップ (ユーザー インターフェイス) とコードが明確に分離されます。
- コードは、ページ マークアップのみを扱うページ デザイナーなどには公開されません。
- コードを複数のページで再利用できます。

コンパイルと配置

単一ファイルページと分離コードページのコンパイルと配置は、ほぼ同じです。最も単純な方法は、ターゲット サーバーにページをコピーすることです。分離コード ページを扱っている場合は、.aspx ページとコード ファイルをコピーします。ASP.NET は、ページが初めて要求されたときにページをコンパイルして実行します。どちらのシナリオでも、マークアップと共にソースコードを配置する点に注意してください。

または、Web サイトをプリコンパイルすることもできます。この場合、ASP.NET がページのオブジェクトコードを生成するので、それをターゲットサーバーにコピーします。プリコンパイルは、単一ファイルモデルと分離コードモデルのどちらでも使用でき、出力は両方のモデルで同じです。

Web アプリケーション プロジェクトのコンパイルの詳細

コンパイル モデル

Web アプリケーション プロジェクトの**コンパイルモデル**では、プロジェクト内のすべてのコードファイルが単一のアセンブリにプリコンパイルされます。既定では、このアセンブリは Bin フォルダに保存されます。このコンパイル モデルでは、単一のアセンブリが作成されるため、アセンブリ名やバージョンなどの属性を指定できます。出力アセンブリの場所も指定できます。

Web アプリケーション プロジェクトは、Web サイト プロジェクトと同様に、プロジェクトフォルダではなく**プロジェクトファイル**によって定義されます。プロジェクトファイルは、プロジェクトに含まれるファイル、およびアセンブリ参照と他のプロジェクトのメタデータ設定を格納します。プロジェクトフォルダ内のファイルでも、プロジェクトファイルに定義されていないファイルは、Web アプリケーション プロジェクトの一部としてコンパイルされることはありません。Visual Studio または Visual Web Developer Express によって追加および変更されたプロジェクト設定は、プロジェクトごとに生成されるプロジェクトファイル (.*proj) で参照されます。

ページの実行とデバッグを行うには、Web アプリケーションプロジェクト全体をコンパイルする必要があります。Visual Studio と Visual Web Developer Express は、変更したファイルのみをビルドするインクリメンタル ビルド モデルを使用しているため、Web アプリケーション プロジェクト全体のビルドを迅速に行うことができます。

クラス ファイルのプリコンパイル

Web アプリケーション プロジェクトは、MSBuild を使用してクラスファイルをプリコンパイルします。これらのクラスファイルは、単一のアセンブリにコンパイルされます。

次の表に、単一のアセンブリにコンパイルされる Web アプリケーションプロジェクトのクラスファイルの種類を示します。

表. Web アプリケーション プロジェクトのクラスファイルの種類

クラス ファイルの種類	説明
スタンドアロン	コンパイル後に Bin フォルダに追加できるクラスファイル
分離コード	コンテンツ ファイルに直接関連付けられるユーザー定義コード
デザイナー	デザイナーにより自動生成されるコード (.designer ファイルの変更は推奨されません)

コンパイル オプションのカスタマイズ

[プロジェクトのプロパティ] ウィンドウの [アプリケーション] タブで、出力アセンブリ名、バージョン、その他の詳細を指定できます。同じく [ビルド] タブで、プロジェクトのビルド構成を指定します。さらに [ビルド イベント] タブでは、コンパイル時にビルド前の処理とビルド後の処理を追加できます。

[ビルド アクション] プロパティの設定

クラスファイル (.aspx.cs など) の [ビルド アクション] プロパティが [コンパイル] に設定されている場合、既定では、Web アプリケーション プロジェクトのクラスファイルのみが **MSBuild** によってコンパイルされます。ただし、Web アプリケーション プロジェクトでクラスファイルが App_Code フォルダに格納されている場合、これらは **ASP.NET コンパイラ**によってコンパイルされます。

動的コンパイル

プロジェクト内のコードファイルは、MSBuild の使用によって単一のアセンブリにプリコンパイルされますが、Web アプリケーション プロジェクトの ASP.NET Web ページ (.aspx) とユーザー コントロール (.ascx) は、サーバー上で ASP.NET コンパイラによって動的にコンパイルされます。Web アプリケーション プロジェクトでは、Web ページとユーザー コントロールは、@Page ディレクティブまたは @Control ディレクティブで **CodeBehind** 属性および **Inherits** 属性を使用できます。CodeBehind 属性は、使用する分離コードファイルを参照します。Inherits 属性は、分離コードファイル内の名前空間とクラスをポイントします。

Web アプリケーション プロジェクトでは、サイト内の ASP.NET Web ページをコンパイルし、配置した後で、それらの Web ページに対して限定的な変更を加えることができます。

コントロールは、protected または public にしてあれば、分離コードファイル内で静的に宣言できます。コントロールの宣言を分離コードファイルに移動すると、次の場合に便利です。

- コントロールの型を標準の種類から派生する必要がある場合
- コントロールで既定のスコープ以外のスコープが必要な場合
- コントロールの宣言にメタデータ属性を追加する必要がある場合
- コントロールの宣言に XML コードのコメントを記述する必要がある場合

配置

すべてのクラスファイルが単一のアセンブリにコンパイルされるので、配置する必要があるのはそのアセンブリと .aspx ファイル、および .ascx ファイル、構成ファイル (Web.config)、その他の静的コンテンツファイルだけです。このモデルでは、.aspx ファイルはブラウザーから要求されない限りコンパイルされません。

ただし、.aspx ファイルをコンパイルして単一のアセンブリに含めて配置することもできます。それには、**Web Deployment Projects** を使用します。

Web サイト プロジェクト プリコンパイルの概要

機能

ASP.NET Web サイトを**プリコンパイル**することには、次の利点があります。

- ページやコードファイルが初めて要求されたときにそれらをコンパイルする必要がないため、ユーザーへの応答時間が短縮されます。
- ユーザーがサイトを表示する前に、コンパイル時のバグを特定できます。
- コンパイル済みのサイトを作成して、ソース コードを含めずに運用サーバーに配置できます。

背景

既定では、ASP.NET Web ページとコード ファイルは、ユーザーが Web サイトのページなどのリソースを初めて要求したときに動的にコンパイルされます。

ASP.NET では、ユーザーに Web サイトを提供する前に、サイト全体をプリコンパイルすることもできます。サイトをプリコンパイルには、次のオプションがあります。

- サイトの埋め込み先プリコンパイル
- サイトの配置用プリコンパイル

埋め込み先プリコンパイル

Web サイトをプリコンパイルすることで、パフォーマンスをある程度向上させることができます。これは、ASP.NET Web ページやコードファイルを頻繁に変更したり追加したりするサイトで特に効果的です。

サイトの埋め込み先プリコンパイルは、実質的にはユーザーがサイトのページを要求したときに実行されるコンパイルと同じコンパイルを実行します。このため、主なパフォーマンスの向上は初回の要求時にページをコンパイルする必要がなくなることによって図られます。

埋め込み先プリコンパイルを実行すると、すべての種類の ASP.NET ファイルがコンパイルされます。サイトを再度プリコンパイルすると、新しいファイルと変更されたファイル (またはそれらに依存するファイル) のみがコンパイルされます。このようにコンパイラが最適化されているため、小さな更新の後でもサイトをコンパイルする方が有利です。

配置用プリコンパイル

サイトのプリコンパイルを使用して、運用サーバーに配置できる実行可能バージョンのサイトを生成することもできます。配置用プリコンパイルでは、レイアウト形式で出力が作成されます。このレイアウトには、アセンブリ、構成情報、サイトのフォルダに関する情報、および静的ファイル (HTML ファイル、グラフィックなど) が含まれます。

サイトをコンパイルした後は、Windows XCopy コマンド、FTP、Windows インストールなどのツールを使用して、レイアウトを運用サーバーに配置できます。配置されたレイアウトはサイトとして機能し、ASP.NET はそのレイアウト内のアセンブリからページを取得して要求に応答します。

配置用プリコンパイルには、配置専用プリコンパイルと配置/更新用プリコンパイルの 2 種類があります。

配置専用プリコンパイル

配置専用プリコンパイルを実行すると、コンパイラは通常は実行時にコンパイルされるほとんどすべての ASP.NET ソース ファイルからアセンブリを生成します。これらのファイルには、ページ内のプログラムコード、.cs/.vb クラス ファイル、他のコードファイル、リソースファイルなどがあります。コンパイラは、この出力からすべてのソースとマークアップを削除します。生成されたレイアウトには、.aspx ファイルごとに、そのページの適切なアセンブリへのポインタを含むコンパイル済みのファイル (拡張子 .compiled) が生成されます。

ページのレイアウトを含む Web サイトを変更するには、元のファイルを変更し、サイトを再コンパイルして、そのレイアウトを再配置する必要があります。ただし、サイト構成は例外です。サイトを再コンパイルしなくても、運用サーバーの Web.config ファイルを変更できます。

配置/更新用プリコンパイル

配置/更新用プリコンパイルを実行すると、コンパイラは、すべてのソースコード (単一ファイル ページ内のページコードを除く) と、通常はアセンブリが生成されるその他のファイル (リソース ファイルなど) からアセンブリを生成します。コンパイラは、.aspx ファイルをコンパイル済み分離コード モデルを使用する単一ファイルに変換し、それらをレイアウトにコピーします。

このオプションを使用すると、サイト内の ASP.NET Web ページをコンパイルした後で、ページに一定の変更を加えることができます。たとえば、コントロールの配置、色、フォントなどのページの外観を変更できます。また、イベント ハンドラなどのコードが必要なければ、コントロールを追加することもできます。

サイトが初めて実行されると、ASP.NET は、マークアップから出力を生成するために追加のコンパイルを実行します。

プリコンパイルの実行

Aspnet_compiler.exe ツールを使用して、コマンドラインで Web サイトをプリコンパイルできます。Visual Studio には、IDE から Web サイトをプリコンパイルするためのコマンドが用意されています (Web サイトの発行)。

プリコンパイル出力の書き込み

プリコンパイル プロセスが終了すると、指定したフォルダに出力結果が書き込まれます。この出力は、ファイル転送プロトコル (FTP) または HTTP を使用してアクセスできるファイルシステム内の任意のフォルダに書き込むことができます。ターゲットサイトに書き込むには、適切なアクセス許可が必要です。

ステージングサーバーや運用サーバー上のターゲットフォルダを指定するか、ローカルコンピュータ上のフォルダに出力を書き込むことができます。運用サーバー上のフォルダを指定した場合は、プリコンパイルと配置を 1 つの手順で実行できます。Web サイトに含まれないフォルダに出力を書き込んだ場合は、別の手順で出力をサーバーにコピーすることができます。

コンパイル プロセスの出力には、コードまたはページをコンパイルしたアセンブリが含まれます。プリコンパイルされたサイトを更新可能にするオプションを選択した場合は、.aspx、.asmx、および .ashx ファイルの分離コード クラスがアセンブリにコンパイルされます。ただし、.aspx、.asmx、および .ashx ファイル自体は、ターゲットフォルダにそのままコピーされるため、サイトの配置後にそれらのレイアウトを変更することができます。プリコンパイルされたサイトが更新可能な場合、単一ファイル ページのコードはアセンブリにはコンパイルされず、ソース コードとして配置されます。

ASP.NET のプリコンパイル時のファイル処理

サイトを配置用にプリコンパイルすると、ASP.NET はレイアウトを作成します。レイアウトは、コンパイラ出力を含む構造です。

ソースコード (プログラムコードやリソースなど、アセンブリを生成するすべてのファイル) とマークアップ (.aspx ファイル) の両方をプリコンパイルすることも、ソースコードのみをプリコンパイルすることもできます。

コンパイルされるファイル

プリコンパイル プロセスは、ASP.NET Web アプリケーション内のさまざまな種類のファイルを処理します。アプリケーションが配置専用でプリコンパイルされるか、配置/更新用にプリコンパイルされるかによりファイルは異なる方法で処理されます。

次の表は、さまざまなファイルの種類とアプリケーションを配置専用でプリコンパイルする場合にそれらのファイルに対して行われる処理を示します。

表.ファイルの種類とアプリケーションを配置専用でプリコンパイルする場合の処理と出力場所

ファイルの種類	プリコンパイル処理
.aspx, .ascx, .master	アセンブリとそのアセンブリをポイントする .compiled ファイルを生成します。元のファイルは、要求に応答するためのプレースホルダーとして残されます。
.asmx, .ashx	アセンブリを生成します。元のファイルは、要求に応答するためのプレースホルダーとして残されます。
App_Code フォルダ内のファイル	Web.config 設定に基づいて、1 つまたは複数のアセンブリを生成します。
App_Code フォルダにない .cs ファイルまたは .vb ファイル	このファイルに依存するページまたはリソースと一緒にコンパイルします。
Bin フォルダ内の既存の .dll ファイル	ファイルをそのままコピーします。
リソース (.resx) ファイル	App_LocalResources または App_GlobalResources フォルダ内のすべての .resx ファイルに対して、1 つまたは複数のアセンブリと 1 つのカルチャー構造体を生成します。
App_Themes フォルダとサブフォルダ内のファイル	ターゲットにアセンブリを生成し、それらのアセンブリをポイントする .compiled ファイルを生成します。
静的ファイル (.htm, .html, .js, グラフィック ファイルなど)	ファイルをそのままコピーします。
ブラウザー定義ファイル	ファイルをそのままコピーします。

依存プロジェクト	依存プロジェクトの出力をアセンブリとして生成します。
Web.config ファイル	ファイルをそのままコピーします。
Global.asax ファイル	1 つのアセンブリを生成します。

次の表は、さまざまなファイルの種類と、アプリケーションを配置/更新用にプリコンパイルする場合にそれらのファイルに対して行われる処理を示します。

表. ファイルの種類とアプリケーションを配置/更新用にプリコンパイルする場合の処理と出力場所

ファイルの種類	プリコンパイル処理	出力場所
.aspx, .ascx, .master	分離コード クラス ファイルを持つファイルのアセンブリを生成します。アセンブリをポイントする .compiled ファイルを生成します。これらのファイルの単一ファイル バージョンは、ターゲットにそのままコピーされます。	アセンブリと .compiled ファイルが Bin フォルダに書き込まれます。
.asmx, .ashx	コンパイルしないでファイルをそのままコピーします。	コピー元と同じ構造です。
App_Code フォルダ内のファイル	Web.config 設定に基づいて、1 つまたは複数のアセンブリを生成します。	Bin フォルダ
App_Code フォルダにない .cs ファイルまたは .vb ファイル	このファイルに依存するページまたはリソースと一緒にコンパイルします。	Bin フォルダ
Bin フォルダ内の既存の .dll ファイル	ファイルをそのままコピーします。	Bin フォルダ
リソース (.resx) ファイル	App_GlobalResources フォルダ内のすべての .resx ファイルに対して、1 つまたは複数のアセンブリと 1 つのカルチャー構造体を生成します。 App_LocalResources フォルダ内の .resx ファイルに対しては、ファイルを出力場所の App_LocalResources フォルダにそのままコピーします。	アセンブリは Bin フォルダに置かれます。
App_Themes フォルダとサブフォルダ内のファイル	ファイルをそのままコピーします。	コピー元と同じ構造です。
静的ファイル (.htm, .html, .js, グラフィック ファイルなど)	ファイルをそのままコピーします。	コピー元と同じ構造です。

ブラウザー定義ファイル	ファイルをそのままコピーします。	App_Browsers
依存プロジェクト	依存プロジェクトの出力をアセンブリとして生成します。	Bin フォルダ
Web.config ファイル	ファイルをそのままコピーします。	コピー元と同じ構造です。
Global.asax ファイル	1 つのアセンブリを生成します。	Bin フォルダ

.compiled ファイル

ASP.NET Web アプリケーション内の実行可能ファイルに対して、コンパイラはアセンブリやファイルの名前に .compiled という拡張子を追加します。**.compiled** ファイルに、実行可能なコードは含まれません。ASP.NET が適切なアセンブリを見つけるために必要な情報のみが含まれます。

プリコンパイルされた Web サイトの更新

プリコンパイルされた Web サイトを配置した後は、そのサイト内のファイルに一定の変更を行うことができます。次の表に、ファイルに対して可能な変更をまとめます。

表. プリコンパイルされた *Web* サイトの更新

ファイルの種類	可能な変更 (配置専用)	可能な変更 (配置/更新用)
静的ファイル (.htm, .html, .js, グラフィック ファイルなど)	静的ファイルを変更、削除、または追加できます。ASP.NET Web ページが変更または削除されたページまたはページ要素を参照している場合は、エラーが発生します。	静的ファイルを変更、削除、または追加できます。ASP.NET Web ページが変更または削除されたページまたはページ要素を参照している場合は、エラーが発生します。
.aspx ファイル	既存のページを変更することはできません。新しい .aspx ファイルを追加することはできません。	.aspx ファイルのレイアウトを変更したり、コードを必要としない要素 (イベント ハンドラを持たない HTML 要素、ASP.NET サーバー コントロールなど) を追加したりすることができます。新しい .aspx ファイルを追加することもできます。これらのファイルは、初回の要求時にコンパイルされます。
.skin ファイル	変更と新しい.skin ファイルは無視されます。	変更と新しい.skin ファイルの追加が可能です。
Web.config ファイル	.aspx ファイルのコンパイルに影響する変更が可能です。デバッグまたはバッチ処理の	サイトまたはページのコンパイルに影響しない変更は可能です。これには、コンパイラ設定、信頼レベル、グローバリゼ

	コンパイル オプションは無視されます。 プロファイル プロパティまたはプロバイダ要素を変更することはできません。	ーションなどの変更があります。コンパイルに影響する変更やコンパイルされたページ内の動作に対する変更は無視されるか、エラーが発生します。それ以外の変更は可能です。
ブラウザー定義	変更と新しいファイルの追加が可能です。	変更と新しいファイルの追加が可能です。
リソース (.resx) ファイルからコンパイルされたアセンブリ	新しいリソース アセンブリ ファイルをグローバル リソースとローカル リソースの両方として追加できます。	新しいリソース アセンブリ ファイルをグローバル リソースとローカル リソースの両方として追加できます。

Web サイトのレイアウト

既定のページ

アプリケーションの既定のページを作成すると、ユーザーが簡単にサイトを訪問できるようになります。既定のページとは、ユーザーが特定のページを指定しないでサイトにアクセスしたときに表示されるページです。たとえば、Default.aspx という名前のページを作成して、サイトのルートフォルダ内に格納できます。ユーザーが特定のページを指定しないでサイトにアクセスした場合に (たとえば、http://www.contoso.com/)、自動的に Default.aspx ページが要求されるようにアプリケーションを構成できます。

アプリケーション フォルダ

ASP.NET は、特定の種類の内容に対して使用できる一定のフォルダ名を定めています。予約済みのフォルダ名と、そのフォルダに通常格納されるファイルの種類の一覧を次の表に示します。

表. 使用するフォルダと内容

フォルダ	説明
App_Browsers	個々のブラウザーの識別と、それぞれの機能の判断のために ASP.NET が使用するブラウザー定義ファイル (.browser ファイル) が格納されます。
App_Code	アプリケーションの一部としてコンパイルするユーティリティクラスおよびビジネスオブジェクトのソースコード (たとえば .cs ファイルや .vb ファイル) が格納されます。動的にコンパイルされるアプリケーションの場合、ASP.NET はアプリケーションに対する最初の要求時に App_Code フォルダ内のコードをコンパイルします。その後は、変更が検出されたときにこのフォルダ内のアイテムが再コンパイルされます。
App_Data	MDF ファイル、XML ファイルなどのアプリケーション データ ファイル、その他のデータ ストア ファイルが格納されます。App_Data フォルダは、メンバーシップ情報やロール情報を維持するアプリケーションのローカル データベースを格納するために、ASP.NET 2.0 が使用します。
App_GlobalResources	グローバル スコープのアセンブリにコンパイルされるリソース (.resx ファイルと .resources ファイル) が格納されます。App_GlobalResources フォルダ内のリソースは、厳密に型指定され、プログラムからアクセスできます。
App_LocalResources	アプリケーション内の特定のページ、ユーザーコントロール、またはマスターページに関連付けられているリソース (.resx ファイルお

	よび .resources ファイル) が格納されます。
App_Themes	ASP.NET Web ページとそのコントロールの外観を定義するファイルのコレクション (.skin ファイル、.css ファイル、イメージファイル、および汎用リソース) が格納されます。
App_WebReferences	アプリケーション内で使用される Web 参照を定義する参照コントラクトファイル (.wsdl ファイル)、スキーマ (.xsd ファイル)、および探索ドキュメントファイル (.disco ファイルと .discomap ファイル) が格納されます。
Bin	コントロール、コンポーネント、またはアプリケーションで参照するその他のコードをコンパイルしたアセンブリ (.dll ファイル) が格納されます。Bin フォルダ内のコードによって表されるクラスは、アプリケーションから自動的に参照されます。

サブフォルダの管理

サイトの構成設定は、そのサイトのルートフォルダ内にある Web.config ファイルで管理されます。サブフォルダ内にファイルがある場合、そのフォルダ内に Web.config ファイルを作成することにより、それらのファイルの構成設定を別々に維持できます。

サイト コンテンツに対するアクセスの制限

サイトの構成の一部として、個々のファイルまたはサブフォルダのいずれかに対するアクセスを制限する設定を構成できます。個別にコンテンツを制限することも、ロール (グループ) 別にコンテンツを制限することもできます。

Web サイトのパス

Web サイトのリソースを使用するときは、リソースのパスを指定する必要があります。ASP.NET には、リソースを参照したり、アプリケーションのページなどのリソースのパスを指定したりするための機能が用意されています。

リソースのパスの指定

多くの場合、ページの要素やコントロールは、ファイルなどの外部リソースを参照する必要があります。ASP.NET は、外部リソースを参照する多様なメソッドをサポートしています。使用する参照メソッドは、クライアント要素と Web サーバーコントロールのどちらを使用するかによって異なります。

クライアント要素

Web サーバーコントロールではないページの要素（**クライアント要素**）は、そのままブラウザーに渡されます。したがって、クライアント要素からリソースを参照する場合は HTML の URL の標準規則に従ってパスを指定します。完全修飾（絶対とも呼ばれます) URL パス、または各種の相対パスを使用できます。たとえば、ページにある img 要素の src 属性は、次のいずれかのパスを使用して設定できます。

- **絶対 URL パス** —— 外部 Web サイトなどの別の場所のリソースを参照する場合に便利です。

```
<img src="http://www.contoso.com/MyApp/Images/Sample.jpg" />
```

- **アプリケーションのルートではなくサイトのルートを基準にして解決されるサイトルート相対パス** —— イメージやクライアントスクリプトファイルなど、複数のアプリケーション間にまたがるリソースを Web サイトのルートの下にあるフォルダに格納している場合に便利です。

```
<img src="/Images/Sample.jpg" />
```

  Web サイトが http://www.contoso.com の場合、このパスは次のように解決されます。

```
http://www.contoso.com/Images/Sample.jpg
```

- **現在のページパスを基準にして解決される相対パス**

```
<img src="Images/Sample.jpg" />
```

- **現在のページ パスのピアとして解決される相対パス**

```
<img src="../Images/Sample.jpg" />
```

サーバー コントロール

リソースを参照する **ASP.NET サーバーコントロール**では、クライアント要素の場合と同様にして絶対パスと相対パスを使用できます。相対パスを使用すると、コントロールが含まれるページ、ユーザーコントロール、またはテーマのパスを基準にして相対的に解決されます。

このユーザーコントロールを実行すると、パスは /Controls/Images/Sample.jpg に解決されます。これは、ユーザーコントロールをホストするページの場所に関係ありません。

サーバーコントロールにおける絶対パス参照と相対パス参照には、次のような短所があります。

- 絶対パスは、アプリケーション間で移植できません。絶対パスで参照されているアプリケーションを移動すると、リンクは失われます。
- リソースやページを別のフォルダに移動すると、クライアント要素のスタイルに対する相対パスを維持することは困難になります。

このような短所を克服するために、ASP.NET には Web アプリケーションの**ルート演算子**（~）が用意されています。

次の例は、Image サーバー コントロールを使用するときに、~ 演算子を使用してイメージのルート相対パスを指定しています。

```
<asp:image runat="server" id="Image1"
  ImageUrl="~/Images/Sample.jpg" />
```

現在の Web サイトの物理ファイルパスの決定

アプリケーションで、サーバー上のファイルまたはその他のリソースのパスが必要になる場合があります。たとえば、アプリケーションがプログラムによってテキストファイルを読み書きする場合は、メソッドに対してファイルの完全な物理パスを提供する必要があります。ASP.NET では、アプリケーション内からプログラムを使用して任意の物理ファイルパスを取得することができます。これにより、ベースファイルパスを使用して、必要なリソースへの完全パスを作成できます。ファイルパスを決定する際に一般に使用される ASP.NET の機能には、パス情報を返す **HttpRequest** オブジェクトのプロパティと **MapPath** メソッドの 2 つがあります。

要求プロパティからのパスの決定

アプリケーションのリソースのパスを決定する際に使用する HttpRequest オブジェクトのプロパティを次の表に示します。

この表の例は、次の前提に基づいています。

- ブラウザーの要求は http://www.contoso.com/MyApp/MyPages/Default.aspx という URL を使用して行われました。
- "仮想パス" はサーバー識別子の後の要求 URL 部分を意味するものとします。この場合の仮想パスは /MyApplication/MyPages/Default.aspx です。
- Web サイトのルートの物理パスは、C:¥inetpub¥wwwroot¥MyApplication¥ です。
- 物理パスには MyPages というフォルダが含まれます。

表. HttpRequest オブジェクトのプロパティ

プロパティ	説明
ApplicationPath	行われる要求がアプリケーションのどこにあるかに関係なく、現在のアプリケーションのルートパスを取得します。この例では、プロパティは / を返します。
CurrentExecutionFilePath	現在の要求の仮想パスを取得します。要求がサーバーコードでリダイレクトされている場合、CurrentExecutionFilePath が正しいという点で FilePath プロパティとは異なります。この例では、プロパティは /MyApplication/MyPages/Default.aspx を返します。 Transfer または Execute の呼び出しの結果として実行中のコードでこのプロパティを受け取る場合、パスはコードの場所を反映します。
FilePath	現在の要求の仮想パスを取得します。この例では、プロパティは /MyApplication/MyPages/Default.aspx を返します。 **CurrentExecutionFilePath** プロパティとは異なり、**FilePath** は

	サーバー側転送を反映しません。
Path	現在の要求の仮想パスを取得します。この例では、プロパティは /MyApplication/MyPages/default.aspx を返します。
PhysicalApplicationPath	現在実行中のアプリケーションのルートディレクトリの物理ファイルシステムパスを取得します。この例では、プロパティは C:¥inetpub¥wwwroot¥ を返します。
PhysicalPath	要求された URL に対応する物理ファイルシステムパスを取得します。この例では、プロパティは C:¥inetpub¥wwwroot¥MyApplication¥MyPages¥default.aspx を返します。

MapPath メソッドの使用

MapPath メソッドは、渡される仮想パスの完全な物理パスを返します。たとえば、次のコードは Web サイトのルートのファイルパスを返します。

<Visual Basic>

```
Dim rootPath As String = Server.MapPath("~")
```

<C#>

```
String rootPath = Server.MapPath("~");
```

Web サイトのファイルの種類

Web サイト アプリケーションにはさまざまな種類のファイルが含まれます。一部のファイルは ASP.NET で、その他は IIS サーバーでサポート/管理されます。

ASP.NET が管理するファイルの種類

ASP.NET が管理するファイルの種類は、IIS の **Aspnet_isapi.dll** にマップされます。

表. *ASP.NET* が管理するファイルの種類

ファイルの種類	場所	説明
.asax	アプリケーション ルート	代表的な例として、**HttpApplication** クラスから派生したコードが格納された **Global.asax** ファイルがあります。このファイルは、アプリケーションを表し、アプリケーションの有効期間の開始時と終了時に実行されるオプションのメソッドが含まれます。

.ascx	アプリケーション ルートまたはサブディレクトリ	再利用可能なカスタムコントロールを定義する Web ユーザーコントロールファイルです。
.ashx	アプリケーション ルートまたはサブディレクトリ	**IHttpHandler** インターフェイスを実装するコードが格納された汎用ハンドラファイルです。
.asmx	アプリケーション ルートまたはサブディレクトリ	SOAP 経由で他の Web アプリケーションに提供するクラスとメソッドを格納した XML Web サービスのファイルです。
.aspx	アプリケーション ルートまたはサブディレクトリ	Web コントロール、プレゼンテーション ロジック、ビジネス ロジックを含めることができる ASP.NET Web フォーム ファイル (ページ) です。
.axd	アプリケーション ルート	Web サイトの管理に使用するハンドラファイルです。代表的なファイルとして **Trace.axd** があります。
.browser	App_Browsers サブディレクトリ	クライアントブラウザーの機能の識別に使用するブラウザー定義ファイルです。
.cd	アプリケーション ルートまたはサブディレクトリ	クラスダイアグラムのファイルです。
.compile	Bin サブディレクトリ	コンパイル済み Web サイト ファイルであるアセンブリを指す、プリコンパイルされたスタブファイルです。実行可能ファイル形式 (.aspx、ascx、.master、テーマファイル) はプリコンパイルされ、Bin サブディレクトリに保存されます。
.config	アプリケーション ルートまたはサブディレクトリ	ASP.NET 機能の設定の XML 要素が格納された構成ファイル (通常は Web.config) です。
.cs, .vb	App_Code サブディレクトリ。ASP.NET ページの分離コードファイルの場合は Web ページと同じディレクトリ	実行時にコンパイルされる、クラスのソース コード ファイルです。クラスは HTTP モジュール、HTTP ハンドラ、ASP.NET ページの分離

		コードファイル、またはアプリケーションロジックを含むスタンドアロン クラス ファイルを指します。
.csproj, .vbproj	Visual Studio のプロジェクトディレクトリ	Visual Studio のクライアントアプリケーションプロジェクトのプロジェクトファイルです。
.disco, .vsdisco	App_WebReferences サブディレクトリ	利用可能な Web サービスの場所の特定に使用する XML Web サービス検出ファイルです。
.dsdgm, .dsprototype	アプリケーション ルートまたはサブディレクトリ	分散サービスダイアグラム (DSD) ファイルです。Web サービスの通信のアーキテクチャ的な側面のリバースエンジニアリングを行う Web サービスを提供または使用する Visual Studio ソリューションに追加します。
.dll	Bin サブディレクトリ	コンパイル済みのクラス ライブラリ ファイル (アセンブリ) です。コンパイル済みアセンブリを Bin サブディレクトリに置く代わりに、クラスのソースコードを App_Code サブディレクトリに置くことができます。
.licx, .webinfo	アプリケーション ルートまたはサブディレクトリ	ライセンスファイルです。ライセンス管理によって、コントロール作成者は、ユーザーがそのコントロールの使用が許可されているかどうかを確認して、知的財産権を保護できます。
.master	アプリケーション ルートまたはサブディレクトリ	アプリケーションの他の Web ページのレイアウトを定義するマスターページです。
.mdb, .ldb	App_Data サブディレクトリ	Access データベースファイルです。
.mdf	App_Data サブディレクトリ	SQL Server Express で使用する SQL データベースファイルです。
.msgx, .svc	アプリケーション ルートまた	MFx (Indigo Messaging

	はサブディレクトリ	Framework) サービスファイルです。
.rem	アプリケーション ルートまたはサブディレクトリ	リモート処理ハンドラファイルです。
.resources, .resx	App_GlobalResources サブディレクトリまたは App_LocalResources サブディレクトリ	イメージ、ローカライズ可能テキスト、またはその他のデータを参照するリソース文字列を含むリソースファイルです。
.sdm, .sdmDocument	アプリケーション ルートまたはサブディレクトリ	システム定義モデル (SDM) ファイルです。
.sitemap	アプリケーション ルート	Web サイトの構造を含むサイトマップファイルです。ASP.NET には、サイトマップファイルを使用して Web ページに簡単にナビゲーション コントロールを表示するための既定のサイトマッププロバイダが付属します。
.skin	App_Themes サブディレクトリ	一貫した書式設定のために、Web コントロールに適用するプロパティ設定を含むスキンファイルです。
.sln	Visual Web Developer のプロジェクト ディレクトリ	Visual Web Developer プロジェクトのソリューションファイルです。
.soap	アプリケーション ルートまたはサブディレクトリ	SOAP の拡張ファイルです。

IIS が管理するファイルの種類

ASP.NET が管理するファイルの種類は、通常 IIS の **asp.dll** ハンドラにマップされます。

表. *ASP.NET* が管理するファイルの種類

ファイルの種類	場所	説明
.asa	アプリケーション ルート	ASP セッションまたはアプリケーションの有効期間の開始時と終了時に実行されるオプションのメソッドが格納されたファイルです。代表的なものとして **Global.asa** があります。
.asp	アプリケーション ルートまたはサブディレクトリ	@ ディレクティブおよび ASP の組み込みオブジェクトを使用したスクリプトコードが格納された ASP Web ページです。

.cdx	App_Data サブディレクトリ	Visual FoxPro の複合インデックスファイルの構造ファイルです。
.cer	アプリケーションルートまたはサブディレクトリ	Web サイトの認証に使用する証明書ファイルです。
.idc	アプリケーションルートまたはサブディレクトリ	httpodbc.dll にマップされたインターネットデータベースコネクタ (IDC) ファイルです。
.shtm, .shtml, .stm	アプリケーションルートまたはサブディレクトリ	ssinc.dll にマップされます。

静的なファイルの種類

IIS は、ファイル名の拡張子が MIME 形式のリストに登録されている場合に限り、静的ファイルにサービスを提供します。

登録されているファイルの種類の一部を、次の表に示します。

表. 登録されているファイルの種類（一部）

ファイルの種類	場所	説明
.css	アプリケーション ルートまたはサブディレクトリ、または App_Themes サブディレクトリ	HTML 要素の書式を設定するために使用するスタイルシートファイルです。
.js	アプリケーション ルートまたはサブディレクトリ	JavaScript (JScript) で記述されたスクリプトファイルです。
.htm, .html	アプリケーション ルートまたはサブディレクトリ	HTML コードで記述された静的 Web ファイルです。

ASP.NET 構成の概要

ASP.NET 構成システムの機能を使用することにより、サーバー全体のすべての ASP.NET アプリケーション、単一の ASP.NET アプリケーション、または個々のページやアプリケーションサブディレクトリを構成できます。認証モード、ページ キャッシュ、コンパイラ オプション、カスタムエラー、デバッグオプション、トレースオプションなどの機能を構成できます。

構成ファイル

ASP.NET 構成データは、どれも **Web.config** という名前の XML テキスト ファイルに格納されます。

構成ファイルの階層と継承

各 Web.config ファイルは、そのファイルが保存されているディレクトリと、その下位にあるすべての子ディレクトリに構成設定を適用します。子ディレクトリの設定は、親ディレクトリで指定されている設定をオプションで上書きまたは変更できます。**location** 要素にパスを指定して、Web.config ファイルの構成設定を個々のファイルまたはサブディレクトリにオプションで適用できます。

ASP.NET 構成階層のルートは、**%SystemRoot%¥Microsoft.NET¥Framework¥versionNumber¥CONFIG¥Web.config** ファイルです。このファイルには、特定バージョンの .NET Framework を実行するすべての ASP.NET アプリケーションに適用される設定が含まれています。各 ASP.NET アプリケーションは、ルート Web.config ファイルから既定の構成設定を継承するため、既定の設定を上書きする設定に対してのみ Web.config ファイルを作成する必要があります。

実行時には、ASP.NET は Web.config ファイルを使用して、受信 URL 要求ごとに構成設定の一意のコレクションを階層的に計算します。この設定は一度だけ計算され、サーバー上にキャッシュされます。ASP.NET は、構成ファイルに対して加えられたすべての変更を検出し、影響を受けるアプリケーションにこれらの変更を自動的に適用してから、通常はアプリケーションを再起動します。階層的な構成設定は、階層内の構成ファイルが変更されるたびに自動的に計算されキャッシュされます。IIS サーバーは、processModel セクションが変更されていない限り、変更を反映するために再起動する必要はありません。

構成ファイルの直接編集

構成ファイルは、テキストエディターまたは XML エディターを使用して直接編集できます。

ASP.NET 3.5 Web.config ファイルの追加の構成要素

.NET Framework バージョン 3.5 に対応する ASP.NET アプリケーションの Web.config ファイルには、以前のバージョンの Web.config ファイルには存在しない構成要素が含まれています。

次の表に、バージョン 3.5 の構成要素と、以前のバージョンの .NET Framework の構成要素に対して加えられた変更を示します。

表. 構成要素と変更点

構成要素	バージョン 3.5 での変更
system.codedom	(新しいセクション) 実行時に .NET Framework Code Document Object Model (CodeDOM) でソースコードをコンパイルする方法を指定します。
configSections	(新しいセクション) system.web.extensions セクションを定義します。このセクションは、クライアントスクリプトから Web サービスを呼び出す方法を定義するために、ASP.NET AJAX が使用します。
assemblies	(compilation 要素の新しいセクション) ASP.NET ページのコンパイル時に参照されるアセンブリのコレクションを指定します。ASP.NET バージョン 3.5 で新しく追加されたアセンブリは、このセクションに含まれます。
namespaces	(更新されたセクション) このセクションでは、既定でインポートされる名前空間を指定します。**System.Linq**、**System.Xml.Linq**、および **System.Collections.Generic** 名前空間が追加されています。
controls	(更新されたセクション) このセクションでは、ASP.NET Web ページのディレクティブ ページ ディレクティブが個々のページにコントロールを登録するのと類似した方法で、コントロールを含むアセンブリを登録し、そのコントロールを参照するためのプレフィックスを用意します。既定では、このセクションは、コントロールを **System.Web.Extensions** アセンブリに登録します。これには、**ListView** コントロールおよび AJAX 関連のコントロールが含まれます。
system.webServer	このセクションでは、httpHandlers セクションと httpModules セクションに追加される AJAX 関連の HTTP ハンドラおよびモジュールを置き換えます。このセクションは、統合モードで実行される IIS 7.0 でハンドラとモジュールが使用できるようにします。
assemblyBinding	(更新されたセクション) このセクションでは、以前のバージョンの ASP.NET AJAX Framework ではなく、ASP.NET バージョン 3.5 に含まれる ASP.NET AJAX Framework を使用するように、ランタイムに指示します。

構成ツール

ASP.NET 構成システムは、テキストエディターより簡単にアプリケーション構成を作成するツールを利用できます。

ASP.NET MMC スナップイン

ASP.NET 用の **Microsoft 管理コンソール (MMC: Microsoft Management Console)** スナップインは、ローカルまたはリモートの Web サーバー上のあらゆるレベルで ASP.NET 構成設定を操作できる便利な手段です。ASP.NET MMC スナップインは、ASP.NET 構成 API を使用しますが、グラフィカルユーザーインターフェイス (GUI) を用意することによって構成設定の編集プロセスを簡略化しています。さらに、Web アプリケーションが設定を継承できるかどうかを制御する ASP.NET 構成 API 機能をサポートし、構成階層のレベル間の依存関係を管理します。

Web サイト管理ツール

Web サイト管理ツールを使用すると、Web サイトの管理者特権を持つユーザー全員がその Web サイトの構成設定を管理できます。このツールを使用するには、Visual Studio の Web サイトプロジェクトにある [Web サイト] — [ASP.NET 構成] を選択します。

Web サイト管理ツールはタブ型のインターフェイスを備えており、関連する構成設定が次のタブにグループ化されています。

- **[セキュリティ] タブ** —— Web アプリケーションのリソースをセキュリティで保護する設定と、ユーザーのアカウントおよびロールを管理する設定を含みます。
- **[アプリケーション] タブ** —— ASP.NET アプリケーションに影響を与える構成要素を管理する設定を含みます。
- **[プロバイダ] タブ** —— アプリケーションプロバイダの追加、編集、削除、テスト、または割り当てを行う設定を含みます。

コマンドライン ツール

.NET Framework には、特定の構成操作を実行するコマンドラインツールが含まれています。たとえば、**Aspnet_regiis.exe** ツールでは ASP.NET アプリケーションに適用される .NET Framework のバージョンを指定できます。

ASP.NET 構成 API

ASP.NET 構成システムが備えている完全なマネージインターフェイスを使用すると、XML 構成ファイルを直接編集するのではなく、ASP.NET アプリケーションをプログラムから構成することができます。さらに、ASP.NET 構成 API は次のことを行います。

- データに関して、構成階層のあらゆるレベルからの統合ビューを提供して管理タスクを簡略化します。

- 構成の作成や複数のコンピュータの構成などの配置タスクを 1 つのスクリプトでサポートします。
- ASP.NET アプリケーション、コンソールのアプリケーションやスクリプト、Web ベースの管理ツール、MMC スナップインなどを作成する開発者に対して単一のプログラミングインターフェイスを提供します。
- 開発者や管理者が無効な構成設定を行うことを防ぎます。
- 構成スキーマを拡張できます。新しい構成パラメーターを定義して、それらを処理する構成セクションハンドラを記述できます。
- 現在実行しているアプリケーションから構成情報を取得する静的メソッドと、個々のアプリケーションから構成情報を取得する非静的メソッドを提供します。

セキュリティの構成

ASP.NET 構成システムを使用すると、承認されていないユーザーによるアクセスから構成ファイルを保護することができます。ASP.NET は、Machine.config ファイルや Web.config ファイルに対してアクセスを要求するすべてのブラウザーへのアクセスを拒否するように IIS を構成します。構成ファイルを直接要求しようとするすべてのブラウザーに対して HTTP アクセス エラー 403 (許可されていません) が返されます。

また、ASP.NET アプリケーション内の構成ファイルは、他のアプリケーションの構成ファイルを読み取るためのアクセス許可を持つアカウント下で構成アプリケーションが完全信頼で実行されている場合を除いて、別の ASP.NET アプリケーションの構成設定にアクセスすることは禁止されます。

ASP.NET 構成ファイルの階層と継承

ASP.NET 構成ファイルをアプリケーションディレクトリ全体に分散させて、複数の ASP.NET アプリケーションを 1 つの継承階層内に構成できます。この構造を使用すると、上位のディレクトリレベルの構成設定に影響を与えずに、特定のディレクトリレベルでアプリケーションが必要とする詳細度の構成を実現できます。

構成構造

ASP.NET 構成ファイルは、**Web.config** という名前で ASP.NET アプリケーション内の複数のディレクトリに置くことができます。ASP.NET 構成階層には次の特性があります。

- 構成ファイルがあるディレクトリ内のリソースとそのすべての子ディレクトリに適用される構成ファイルを使用します。
- 適切なスコープ (コンピュータ全体、すべての Web アプリケーション、個別のアプリケーション、またはアプリケーションのサブディレクトリ) を持つ場所に構成データを配置できます。

- 構成階層内の上位レベルから継承した構成設定を上書きできます。構成設定をロックして、下位レベルの構成設定による上書きを防ぐこともできます。
- 構成設定の論理グループをセクションに分けて整理します。

構成継承

すべての .NET Framework アプリケーションは、**%SystemRoot%¥Microsoft.NET¥Framework¥versionNumber¥CONFIG¥Machine.config** という名前のファイルから基本構成設定と既定値を継承します。**Machine.config** ファイルは、サーバー全体の構成設定に使用されます。この設定には、階層内の下位にある構成ファイルで上書きできないものがあります。

コンソールアプリケーションと Windows アプリケーションでは、<アプリケーション名>.config という名前の構成ファイルを使用して継承した設定を上書きします。ASP.NET アプリケーションは、**Web.config** という名前の構成ファイルを使用して継承した設定を上書きします。

ASP.NET 構成階層のルートは、**ルート Web.config ファイル**と呼ばれるファイルで、Machine.config ファイルと同じディレクトリにあります。ルート Web.config ファイルは、Machine.config ファイルのすべての設定を継承します。ルート Web.config ファイルには、特定のバージョンの .NET Framework を実行するすべての ASP.NET アプリケーションに適用される設定が格納されます。各 ASP.NET アプリケーションは、ルート Web.config ファイルから既定の構成設定を継承するため、既定の設定を上書きする設定に対してのみ Web.config ファイルを作成する必要があります。

コレクション要素内の継承

namespaces 要素や **customErrors** 要素などの一部の構成要素はコレクションです。

コレクション内の構成設定は、通常 **add** 子要素によって追加され **remove** 子要素によってキー名に基づいて削除されます。また、clear 子要素によってコレクション全体をクリアできます。子構成ファイルに追加された設定は、重複が許可されている場合を除き、親構成ファイルのキー名が同じ設定を上書きします。

構成設定のスコープ

構成設定にはさまざまなスコープがあります。一部の構成設定はグローバルスコープを持ちますが、アプリケーション、ルート Web.config ファイル、または Machine.config ファイルのスコープでのみ有効な構成設定もあります。

構成セクションのスコープは、ASP.NET に含まれるすべてのセクションについて Machine.config ファイルの **configSections** の **section** 要素 (全般設定スキーマ) の **allowDefinition** 属性で定義されます。

各ファイルが置かれる構成階層内のレベル、ファイル名、およびファイルの重要な継承特性の説明を次の表に示します。

表. 各ファイルが置かれる構成階層内のレベル、ファイル名、およびファイルの重要な継承特性

構成レベル	ファイル名	ファイルの説明
サーバー	Machine.config	Machine.config ファイルには、サーバー上のすべての Web アプリケーションの ASP.NET スキーマが含まれます。このファイルは、構成結合階層の最上位に位置します。
ルート Web	Web.config	サーバーの Web.config ファイルは、Machine.config ファイルと同じディレクトリに保存され、system.web 構成セクションの大部分の既定値が含まれます。実行時に、このファイルは構成階層内の最上位から 2 番目に結合されます。
Web サイト	Web.config	特定の Web サイトの Web.config ファイルには、Web サイトに適用される設定が含まれ、下位方向にあるサイトのすべての ASP.NET アプリケーションとサブディレクトリに継承されます。
ASP.NET アプリケーションのルート ディレクトリ	Web.config	特定の ASP.NET アプリケーションの Web.config ファイルは、アプリケーションのルート ディレクトリに置かれ、Web アプリケーションに適用される設定が含まれ、このブランチ内の下位方向にあるすべてのサブディレクトリに継承されます。
ASP.NET アプリケーションのサブディレクトリ	Web.config	アプリケーションのサブディレクトリの Web.config ファイルには、このサブディレクトリに適用される設定が含まれ、このブランチ内の下位方向にあるすべてのサブディレクトリに継承されます。
クライアント アプリケーション ディレクトリ	*ApplicationName*.config	<アプリケーション名>.config ファイルには、Windows クライアント アプリケーション (Web アプリケーションではなく) の設定が含まれます。

ProcessModel 要素

processModel 要素 (ASP.NET 設定スキーマ) は、サーバー上のすべての ASP.NET アプリケーションを含む、サーバーで使用されるプロセス モデルを構成します。そのため、processModel 設定は Machine.config ファイルにのみ置くことができ、どの Web.config ファイルの設定によっても上書きできません。

実行時の構成設定の計算

サーバーが特定の Web リソースに対する要求を受け取ると、ASP.NET は、要求された URL の仮想ディレクトリパスにあるすべての構成ファイルを使用して、リソースの構成設定を階層的に計算します。ローカル構成設定は、親構成ファイルの設定を上書きします。

この設定は一度計算され、それ以降の要求のためにキャッシュされます。構成が変更されると、アプリケーションは再起動されます。

1 つのファイルに構成される複数の ASP.NET リソース

多数の構成設定を管理する場合や ISP 設定でクライアント Web サイトを管理している場合は、多数の場所の設定を 1 つの Web.config ファイルに保存すると便利なことがあります。**location** 要素の **path** 属性を使用すると、アプリケーションのサブディレクトリに保存されている多数の固有 ASP.NET リソースを構成できます。

仮想ディレクトリと物理ディレクトリの設定の競合

仮想ディレクトリの構成設定は、物理ディレクトリ構造とは独立しています。また、構成に問題が発生しないように仮想ディレクトリは慎重に編成する必要があります。たとえば、次の物理ディレクトリ構造を持つ MyResource.aspx という名前の ASP.NET ファイルがあるとします。

```
C:¥Subdir1¥Subdir2¥MyResource.aspx
```

さらに、Subdir1 に構成ファイルがあり、vdir1 という名前の仮想ディレクトリが c:¥Subdir1 にマップされ、vdir2 という名前の仮想ディレクトリが c:¥Subdir1¥Subdir2 にマップされているとします。クライアントが http://localhost/vdir1/subdir2/MyResource.aspx を使用して物理的な場所 c:¥Subdir1¥Subdir2¥MyResource.aspx にあるリソースにアクセスする場合、リソースは vdir1 から構成設定を継承します。ただし、クライアントが http://localhost/vdir2/MyResource.aspx を使用して同じリソースにアクセスする場合、リソースは設定を vdir1 から継承しません。このように仮想ディレクトリを作成すると、予期しない結果になったりアプリケーションに障害が発生したりする場合もあります。仮想ディレクトリは入れ子にしないことをお勧めします。入れ子にする場合は Web.config ファイルを 1 つだけ使用してください。

ASP.NET 継承の制限

構成設定の継承を制限して、アプリケーションのパフォーマンスを向上させたり、高い信頼性を維持したり、管理を簡素化したりできる場合があります。制限は、**allowOverride**、**lockAttributes**、**lockAllAttributesExcept**、**lockAllElementsExcept**、**lockItem**、**lockElements** の各属性で制御します。

未処理の例外の構成設定

Aspnet.config ファイルに含まれる設定は、ASP.NET アプリケーションの作成中に**共通言語ランタイム (CLR)** によって処理されます。この設定は、特に未処理の例外の処理方法を CLR に指示します。この構成設定は次のようになります。

```
<legacyUnhandledExceptionPolicy enabled="false" />
```

ASP.NET の偽装

偽装を使用すると、要求を行うユーザーのクライアント側の Windows ユーザーアカウントを使用して ASP.NET アプリケーションを実行できます。偽装は、インターネット インフォメーション サービス (IIS) でユーザーを認証する ASP.NET Web アプリケーションでよく使用されます。偽装が有効である場合、偽装されるユーザーのコンテキストで実行されるのはアプリケーション コードだけです。

偽装は、**identity** 構成要素を使用して制御します。他の構成ディレクティブと同様に、このディレクティブは階層的に適用されます。アプリケーションの偽装を有効にするための最小の構成ファイルは次の例のようになります。

```
<configuration>
  <system.web>
    <identity impersonate="true"/>
  </system.web>
</configuration>
```

また、次の例に示すように、特定の名前とパスワードを追加できます。

```
<identity impersonate="true"
  userName="contoso¥Jane"
  password="********" />
```

※上の例に示された値は、正しいユーザー名およびパスワードに置き換えてください。

この構成を使用すると、要求者の ID とは関係なく、contoso¥Jane の ID を使用してアプリケーション全体を実行できます。このタイプの偽装は、別のコンピュータに委任できます。つまり、偽装されるユーザーのユーザー名とパスワードを指定することで、ネットワーク上の別のコンピュータに接続し、統合セキュリティを使用してファイルなどのリソースを要求したり、SQL Server へのアクセスを要求したりすることができます。

偽装された ID の読み取り

偽装されたユーザーの ID をプログラムから読み取る方法について、コード例を次に示します。

<Visual Basic>

```
Dim username As String = _
    System.Security.Principal.WindowsIdentity.GetCurrent().Name
```

<C#>

```
String username =
    System.Security.Principal.WindowsIdentity.GetCurrent().Name;
```

ロール管理の概要

ロール管理では、承認を管理することができ、アプリケーションのユーザーがアクセスできるリソースを指定できます。ロール管理では、ユーザーを管理者、営業、メンバなどのロールに割り当てることで、ユーザーのグループを 1 つの単位として扱うことができます。

ロールとアクセス ルール

ロールを作成する主な目的は、ユーザー グループのアクセス ルールを簡単に管理する方法を提供することです。ユーザーを作成し、ユーザーをロールに (Windows の場合はユーザーをグループに) 割り当て、次に特定のユーザーに制限するページセットを作成します。

ロール管理、ユーザー ID、およびメンバーシップ

ロールを使用するには、ユーザーが特定のロールに属しているかどうかを判断できるように、アプリケーションのユーザーを識別できる必要があります。アプリケーションを構成することで、**Windows 認証**または**フォーム認証**の 2 つの方法でユーザー ID を作成できます。アプリケーションがローカルエリアネットワーク (つまり、ドメインベースのイントラネットアプリケーション) で実行されている場合は、Windows ドメインアカウント名を使用してユーザーを識別できます。この場合、ユーザーのロールは、そのユーザーが属している Windows グループになります。

Windows アカウントの使用が実用的ではないインターネットアプリケーションやその他のシナリオでは、フォーム認証を使用してユーザー ID を作成できます。

Login コントロールまたはフォーム認証を使用してユーザー ID を作成する場合は、ロール管理とメンバーシップを組み合わせて使用することもできます。

ロール管理と ASP.NET ロール サービス

ASP.NET ロール サービスを使用することで、**Windows Communication Framework (WCF)** からロールにアクセスできます。ロールサービスを使用することで、WCF サービスを使用できるすべてのアプリケーションから、ユーザーのロールを確認できます。

ロール管理 API

ロール管理は、ページやフォルダに対する権限を制限するだけではありません。ロール管理は、ユーザーがロールに属しているかどうかをプログラムで判断するために使用できる API を提供します。これにより、ロールを使用するコードを記述して、ユーザーがだれであるかだけでなくユーザーが属しているロールに基づいてアプリケーション作業を実行できます。

アプリケーションでユーザー ID を作成した場合は、ロール管理 API メソッドを使用して、ロールを作成したり、ロールにユーザーを追加したりすることができます。また、どのユーザーがどのロールに属しているかについての情報を取得することもできます。

アプリケーションで Windows 認証を使用している場合は、ロール管理 API が提供するロール管理機能は少なくなります。

一方、ASP.NET ロール サービスを使用している場合は、ユーザーが特定のロールに属しているかどうかを確認したり、特定のユーザーのすべてのロールを取得したりできます。ただし、ロール サービス API を使用してロールを管理することはできません。

ASP.NET ロール管理のしくみ

ロール管理を使用するには、最初にロール管理を有効にし、オプションでロールを使用できるアクセスルールを構成します。これで、実行時にロール管理機能を使用してロールを操作できます。

ロール管理の構成

ASP.NET ロール管理を使用するには、アプリケーションの Web.config ファイルで次のように構成し、ASP.NET ロール管理を有効にします。

```
<roleManager
    enabled="true"
    cacheRolesInCookie="true" >
</roleManager>
```

ロールの一般的な用途は、ページやフォルダへのアクセスを許可または拒否できるルールを作成することです。このようなアクセスルールは、**Web.config** ファイルの **authorization** セクションで設定できます。次の例は、MemberPages という名前のフォルダ内のページをメンバのロールに属するユーザーのみが表示できるようにし、その他のユーザーのアクセスは拒否する方法を示しています。

```
<configuration>
  <location path="MemberPages">
    <system.web>
      <authorization>
        <allow roles="members" />
        <deny users="*" />
      </authorization>
    </system.web>
  </location>
  <!-- other configuration settings here -->
```

```
<configuration>
```

また、管理者やメンバなどのロールを作成し、そのロールにユーザー ID を割り当てる必要があります。アプリケーションで Windows 認証を使用している場合は、Windows のコンピュータの管理ツールを使用してユーザーおよびグループを作成します。

フォーム認証を使用している場合は、ASP.NET Web サイト管理ツールを使用してユーザーとロールを設定できます。必要に応じて、さまざまなロールマネージャーメソッドを呼び出すことで、この作業をプログラムで実行することもできます。

次の例は、ロールメンバを作成する方法を示しています。

<Visual Basic>

```
Roles.CreateRole("members")
```

<C#>

```
Roles.CreateRole("members");
```

次の例は、JoeWorden というユーザーを個別にロールマネージャーに追加する方法、および JillShrader および ShaiBassli というユーザーをロールメンバに一度に追加する方法を示しています。

<Visual Basic>

```
Roles.AddUserToRole("JoeWorden", "manager")
Dim userGroup(2) As String
userGroup(0) = "JillShrader"
userGroup(1) = "ShaiBassli"
Roles.AddUsersToRole(userGroup, "members")
```

<C#>

```
Roles.AddUserToRole("JoeWorden", "manager");
string[] userGroup = new string[2];
userGroup[0] = "JillShrader";
userGroup[1] = "ShaiBassli";
Roles.AddUsersToRole(userGroup, "members");
```

実行時のロールの操作

実行時にユーザーがサイトにアクセスする場合、ユーザーは Windows アカウント名として、またはアプリケーションにログインすることで ID を作成します。インターネットサイトでは、ユーザーがログインせずにサイトにアクセス (匿名アクセス) する場合、ユーザーにはユーザー ID がないため、どのロールにも属しません。ログインしたユーザーに関する情報は、アプリケーションでは、**User** プロパティで入手できます。ロールが有効になると、ASP.NET は現在のユーザーのロールを検索し、

それらを User オブジェクトに追加します。これで、ロールを確認できるようになります。次の例は、現在のユーザーがメンバのロールに属しているかどうかを判断する方法を示しています。このコードは、ユーザーがロールに属している場合、メンバ用のボタンを表示します。

<Visual Basic>

```
If User.IsInRole("members") Then
   buttonMembersArea.Visible = True
End If
```

<C#>

```
if (User.IsInRole("members"))
{
   buttonMembersArea.Visible = true;
}
```

ASP.NET は、**RolePrincipal** クラスのインスタンスも作成し、現在の要求コンテキストに追加します。これにより、ロール管理作業をプログラムで実行できます。たとえば、特定のロールに属しているユーザーを判断できます。次の例は、現在ログインしているユーザーのロールの一覧を取得する方法を示しています。

<Visual Basic>

```
Dim userRoles() as String = CType(User, RolePrincipal).GetRoles()
```

<C#>

```
string[] userRoles = ((RolePrincipal)User).GetRoles();
```

アプリケーションで LoginView コントロールを使用している場合、このコントロールはユーザーのロールを確認し、そのユーザーのロールに基づいてユーザーインターフェイスを動的に作成できます。

ロール情報のキャッシュ

ユーザーのブラウザーで Cookie が許可されている場合、ASP.NET は、オプションでユーザーのコンピュータの Cookie に暗号化されたロール情報を格納できます。

メンバーシップの概要

ASP.NET メンバーシップを使用すると、ユーザーの資格情報を検証して格納する機能を組み込むことができます。そのため、ASP.NET メンバーシップは Web サイトでのユーザー認証の管理に役立ちます。ASP.NET フォーム認証または ASP.NET ログイン コントロールを ASP.NET メンバーシップと併用することで、ユーザーを認証するための完全なシステムを作成できます。

ASP.NET メンバーシップは次の機能をサポートします。

- 新しいユーザーとパスワードを作成します。
- メンバーシップ情報を Microsoft SQL Server、Active Directory、または代替データストアに格納します。
- サイトを表示するユーザーを認証する。ユーザーはプログラムで認証できます。または、ASP.NET ログインコントロールを使用することで、コードをほとんど記述せずに完全な認証システムを作成できます。
- パスワードの作成、変更、リセットなどのパスワード管理を行います。
- 認証されたユーザーに対して独自のアプリケーションで使用でき、ASP.NET のパーソナル化システムおよびロール管理 (承認) システムとも統合できる一意の ID を公開します。
- カスタム メンバーシップ プロバイダを指定して、独自のコードによってメンバーシップを管理したり、カスタムデータストア内でメンバーシップデータを保守したりすることができるようにします。

メンバーシップ、ロール、およびユーザー プロファイル

メンバーシップは、ASP.NET 内の認証用の独立した機能ですが、ASP.NET ロール管理と統合してサイトの承認サービスとすることもできます。また、メンバーシップをユーザープロファイルと統合して、個々のユーザーに適合するようにアプリケーションごとにカスタマイズすることもできます。

メンバーシップのしくみ

メンバーシップを使用するには、まずメンバーシップをサイトに合わせて構成する必要があります。基本的な手順は次のとおりです。

1. メンバーシップオプションを Web サイト構成の一部として指定します。既定ではメンバーシップは有効になっています。
2. Windows 認証または Passport 認証ではなく、フォーム認証を使用するようにアプリケーションを構成します。
3. メンバーシップのユーザーアカウントを定義します。

これで、アプリケーション内でメンバーシップを使用してユーザーを認証できるようになります。ほとんどの場合、ログインフォームを用意する必要があります。

ユーザーが認証された後は、メンバーシップシステムにより、現在のユーザーの情報を格納したオブジェクトが使用できるようになります。

メンバーシップの構成と管理

メンバーシップシステムはアプリケーションの Web.config ファイルで構成します。最も簡単にメンバーシップを構成して管理するには、ウィザードベースのインターフェイスを備えた Web サイト管理ツールを使用します。メンバーシップを構成する際は、次の情報を指定します。

- 使用するメンバーシッププロバイダ（通常、メンバーシップ情報を格納するデータベースも指定する）。
- パスワードオプション (暗号化や、ユーザー固有の質問に基づくパスワードの復元をサポートするかどうかなど)。
- ユーザーとパスワード。Web サイト管理ツールを使用する場合は、ユーザーを直接作成し、管理できます。このツールを使用しない場合は、メンバーシップ関数を呼び出し、プログラムによってユーザーを作成および管理する必要があります。

ASP.NET マスター ページの概要

ASP.NET マスターページでは、アプリケーション内のページ用の一貫性のあるレイアウトを作成できます。1 つのマスターページで、アプリケーション内のすべてのページ (またはページのグループ) に適用する外観と標準動作を定義します。次に、表示内容が入った個別のコンテンツページを作成します。ユーザーがコンテンツページを要求すると、それらはマスターページとマージされマスターページのレイアウトとコンテンツページの内容を組み合わせた出力が生成されます。

マスター ページのしくみ

マスターページは、実際にはマスターページ自体と 1 つ以上のコンテンツページの 2 つの要素で構成されます。

マスター ページ

マスターページは、**.master** 拡張子を持つ ASP.NET ファイル (MySite.master など) で、静的テキスト、HTML 要素、およびサーバーコントロールを含めることができる定義済みのレイアウトが含まれます。マスターページは Master ディレクティブで識別されます。

<Visual Basic>

```
<%@ Master Language="VB" %>
```

<C#>

```
<%@ Master Language="C#" %>
```

@ Master ディレクティブには **@ Control** ディレクティブに含めることができるほとんどのディレクティブを含めることができます。たとえば、次のマスターページディレクティブには、分離コードファイルの名前が含まれ、クラス名がマスターページに割り当てられています。

<Visual Basic>

```
<%@ Master Language="VB" CodeFile="MasterPage.master.vb"
    Inherits="MasterPage" %>
```

<C#>

```
<%@ Master Language="C#" CodeFile="MasterPage.master.cs"
    Inherits="MasterPage" %>
```

@ Master ディレクティブの他に、マスターページには、html、head、form などのページのすべての最上位 HTML 要素も含まれます。

置き換え可能なコンテンツ プレースホルダー

すべてのページに表示される静的テキストとコントロールの他に、マスターページには 1 つ以上の **ContentPlaceHolder** コントロールも含まれます。これらのプレースホルダーコントロールは、置き換え可能なコンテンツが表示される領域を定義します。次に、置き換え可能なコンテンツがコンテンツページで定義されます。ContentPlaceHolder コントロールを定義すると、マスターページは次のようになります。

<Visual Basic>

```
<% @ Master Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
  <title>Master page title</title>
</head>
<body>
  <form id="form1" runat="server">
    <table>
      <tr>
        <td><asp:contentplaceholder id="Main" runat="server" /></td>
        <td><asp:contentplaceholder id="Footer" runat="server" /></td>
      </tr>
    </table>
  </form>
</body>
</html>
```

<C#>

```
<%@ Master Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
  <title>Master page title</title>
```

```
</head>
<body>
  <form id="form1" runat="server">
    <table>
      <tr>
        <td><asp:contentplaceholder id="Main" runat="server" /></td>
        <td><asp:contentplaceholder id="Footer" runat="server" /></td>
      </tr>
    </table>
  </form>
</body>
</html>
```

コンテンツ ページ

マスターページのプレースホルダーコントロールのコンテンツを定義するには、個々の**コンテンツページ**を作成します。このページは、特定のマスターページにバインドされている ASP.NET ページ (.aspx ファイル、および分離コードファイル) です。このバインドは、使用されるマスターページを指定する MasterPageFile 属性を含めることで、コンテンツ ページの @ Page ディレクティブで確立されます。たとえば、コンテンツページには、次のような @ Page ディレクティブを含めることができます。このディレクティブは、コンテンツページを Master1.master ページにバインドします。

<Visual Basic>

```
<%@ Page Language="VB" MasterPageFile="~/MasterPages/Master1.master"
    Title="Content Page" %>
```

<C#>

```
<%@ Page Language="C#" MasterPageFile="~/MasterPages/Master1.master"
    Title="Content Page" %>
```

このコンテンツページでは **Content** コントロールを追加して、それらをマスターページの **ContentPlaceHolder** コントロールにマップすることでコンテンツを作成します。たとえば、マスター ページに **Main** と **Footer** というコンテンツ プレースホルダーが含まれているものとします。このコンテンツ ページでは、2 つの Content コントロールを作成できます。次の図に示すように、1 つは ContentPlaceHolder コントロールの Main にマップされ、もう 1 つは、ContentPlaceHolder コントロールの Footer にマップされます。

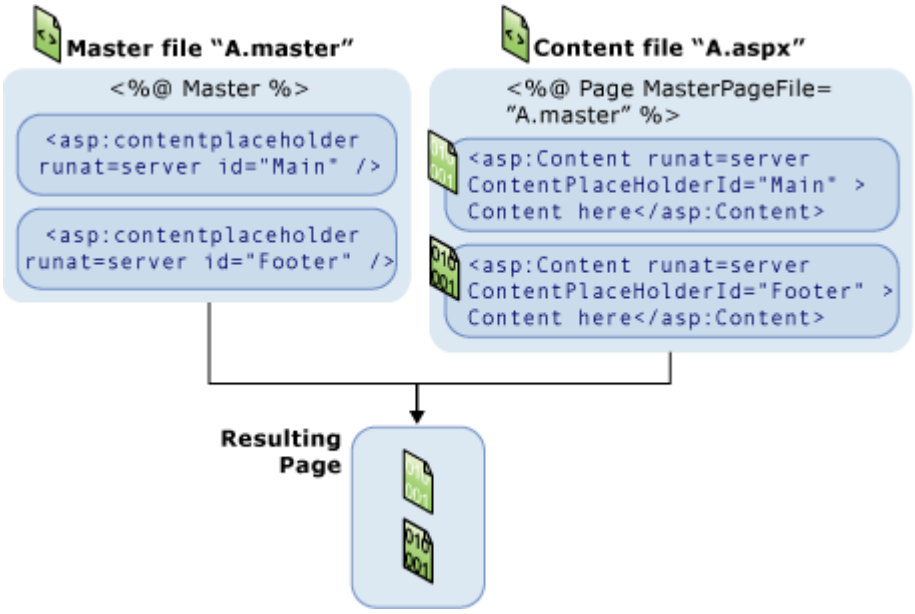

図.プレースホルダー コンテンツの置き換え

Content コントロールを作成したら、これらのコントロールにテキストとコントロールを追加します。コンテンツページでは、Content コントロール内に存在しないもの (サーバーコードのスクリプトブロックを除く) はすべてエラーになります。コンテンツページでは、ASP.NET ページで実行できるすべてのタスクを実行することができます。たとえば、サーバーコントロールとデータベースクエリ、またはその他の動的なメカニズムを使用して、Content コントロールのコンテンツを生成できます。

コンテンツページは、たとえば次のようになります。

<Visual Basic>

```
<% @ Page Language="VB" MasterPageFile="~/Master.master"
    Title="Content Page 1" %>
<asp:Content ID="Content1" ContentPlaceHolderID="Main" Runat="Server">
    Main content.
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="Footer"
    Runat="Server">
    Footer content.
</asp:content>
```

<C#>

```
<% @ Page Language="C#" MasterPageFile="~/Master.master"
    Title="Content Page 1" %>
<asp:Content ID="Content1" ContentPlaceHolderID="Main" Runat="Server">
    Main content.
```

```
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="Footer"
    Runat="Server">
    Footer content.
</asp:content>
```

マスター ページの利点

マスターページには、次のような利点があります。

- ページの共通機能を集中化できるため、1 か所だけで更新を行うことができます。
- 一連のコントロールとコードを作成し、その結果を簡単にページセットに適用できます。
- プレースホルダー コントロールのレンダリング方法を制御できるため、最終ページのレイアウトを細部まで制御できます。
- 個々のコンテンツページからマスターページをカスタマイズできるオブジェクトモデルが用意されています。

マスター ページの実行時の動作

実行時、マスターページは次の順序で処理されます。

1. ユーザーがコンテンツページの URL を入力してページを要求します。
2. ページがフェッチされると、@ Page ディレクティブが読み取られます。このディレクティブがマスターページを参照する場合は、そのマスターページも同様に読み取られます。今回が初めてのページ要求であった場合は、両方のページがコンパイルされます。
3. コンテンツが更新されているマスターページは、コンテンツページのコントロールツリーにマージされます。
4. 個々の **Content** コントロールのコンテンツは、マスターページ内の対応する **ContentPlaceHolder** コントロールにマージされます。
5. 結果のマージされたページがブラウザーにレンダリングされます。

このプロセスを次の図に示します。

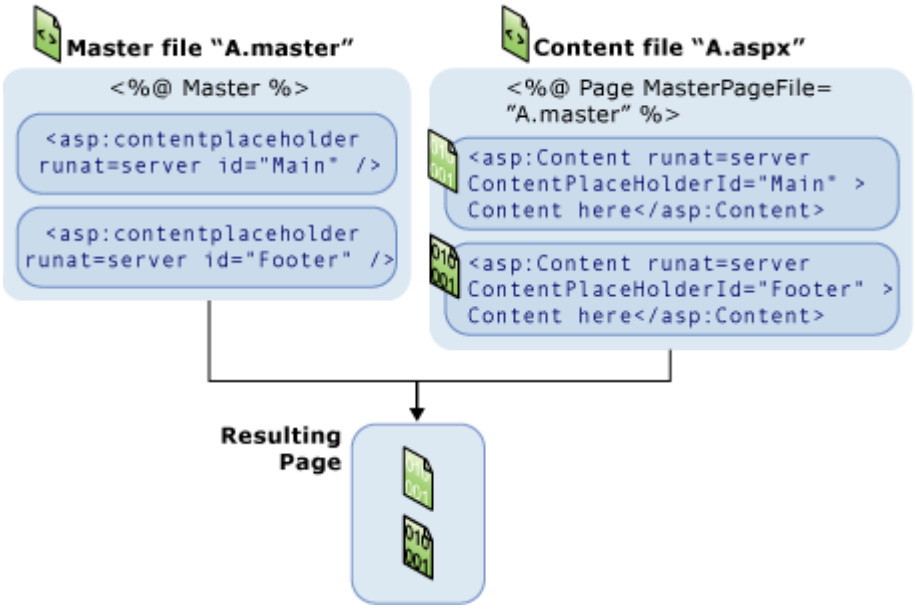

図. 実行時のマスターページ

クライアント側では、結合されたマスターページとコンテンツページは、単一の独立したページになります。このページの URL はコンテンツ ページの URL になります。

プログラミングの視点では、この 2 つのページは、各コントロールの個別のコンテナーとして動作します。コンテンツページは、マスターページのコンテナーとして動作します。ただし、次のセクションで説明するように、コンテンツページのコードからパブリック マスター ページ メンバを参照できます。

マスターページは、コンテンツページの一部になることに注意してください。実際、マスターページはコンテンツページの子として、またそのページ内のコンテナーとして、ユーザーコントロールとほとんど同じように動作します。ただし、この場合、マスターページはブラウザーにレンダリングされるすべてのサーバーコントロールのコンテナーです。たとえば、マージされたマスターページとコンテンツページのコントロールツリーは、次の図のようになります。

- ページ
    - マスター ページ
        - (マスター ページのマークアップとコントロール)
        - ContentPlaceHolder
            - コンテンツ ページのマークアップとサーバー コントロール
        - (マスター ページのマークアップとコントロール)
        - ContentPlaceHolder
            - コンテンツ ページのマークアップとサーバー コントロール
        - (マスター ページのマークアップとコントロール)

図. マージされたマスターページとコンテンツページのコントロール ツリー

マスター ページとコンテンツ ページのパス

コンテンツページが要求されると、そのコンテンツがマスターページにマージされ、ページはコンテンツページのコンテキストで実行されます。たとえば、**HttpRequest** オブジェクトの **CurrentExecutionFilePath** プロパティを取得する場合は、そのプロパティがコンテンツページコード内にあるかマスターページコード内にあるかに関係なく、そのパスはコンテンツページの場所を表します。

外部リソースの参照

コンテンツページとマスターページの両方に、**外部リソース**を参照するコントロールおよび要素を含めることができます。

マージされたコンテンツページとマスターページのコンテキストは、コンテンツページのコンテキストになります。これは、アンカー内のリソース (イメージファイルやターゲットページなど) の URL を指定する方法に影響することがあります。

サーバー コントロール

マスターページの**サーバーコントロール**では、ASP.NET は外部リソースを参照するプロパティの URL を動的に変更します。

ASP.NET は、次のような場合に URL を変更できます。

- URL が ASP.NET サーバーコントロールのプロパティである場合。
- プロパティがコントロール内で URL として内部的にマークされている場合（このプロパティは属性 **UrlPropertyAttribute** でマークされています）。

他の要素

ASP.NET は、サーバーコントロールではない要素の URL を変更することはできません。たとえば、マスターページで **img** 要素を使用しており、その **src** 属性を URL に設定している場合、ASP.NET はその URL を変更しません。この場合、URL はコンテンツページのコンテキストで解決され、URL を適切に作成します。

一般に、マスターページの要素を操作する場合は、サーバーコードを必要としない要素に対しても、サーバーコントロールを使用することをお勧めします。たとえば、img 要素を使用する代わりに、Image サーバーコントロールを使用します。これにより、ASP.NET は URL を正しく解決できるため、マスターページやコンテンツページを移動する際に生じる可能性があるメンテナンス上の問題を回避できます。

マスター ページとテーマ

ASP.NET テーマをマスターページに直接適用することはできません。テーマ属性を @ Master ディレクティブに追加すると、ページの実行時にエラーが発生します。

ただし、以下の状況ではテーマがマスターページに適用されます。

- テーマがコンテンツページで定義されている場合。
- pages 要素 (ASP.NET 設定スキーマ) 要素にテーマ定義を含めることで、サイト全体でテーマを使用するように設定されている場合。

マスター ページのスコープの設定

コンテンツページは、次の 3 つのレベルでマスターページに結合できます。

- **ページ レベル** —— 次のコード例に示すように、各コンテンツページのページディレクティブを使用して、コンテンツページをマスターページにバインドできます。

  <Visual Basic>

  ```
  <%@ Page Language="VB" MasterPageFile="MySite.Master" %>
  ```

  <C#>

  ```
  <%@ Page Language="C#" MasterPageFile="MySite.Master" %>
  ```

- **アプリケーション レベル** —— アプリケーションの構成ファイル (Web.config) の pages 要素で設定を行うことで、アプリケーション内のすべての ASP.NET ページ (.aspx ファイル) が自動的にマスターページにバインドされるように指定できます。この要素は次のようになります。

  この方法を使用すると、アプリケーション内の Content コントロールを持つすべての ASP.NET ページが指定されたマスターページにマージされます。

  ```
  <pages masterPageFile="MySite.Master" />
  ```

- **フォルダ レベル** —— この方法は、アプリケーションレベルのバインドと似ています。ただし、設定を行うのは、1 つのフォルダ内の Web.config ファイルのみです。マスターページのバインドは、そのフォルダ内の ASP.NET ページに適用されます。

ASP.NET のテーマとスキン

テーマは、ページとコントロールの外観を定義し、Web アプリケーション内の複数のページ、Web アプリケーション全体、またはサーバー上のすべての Web アプリケーションに統一した外観を適用できるプロパティ設定のコレクションです。

テーマとコントロール スキン

テーマは、スキン、カスケーディング スタイル シート (CSS: cascading style sheet)、イメージ、その他のリソースという要素のセットで構成されます。

スキン

スキンファイルのファイル名拡張子は **.skin** です。スキンファイルには各コントロールのプロパティ設定が含まれます。Button コントロールの**コントロールスキン**の例を次に示します。

```
<asp:button runat="server" BackColor="lightblue" ForeColor="black" />
```

コントロールには、**既定のスキン**と**名前指定スキン**の 2 種類のコントロールスキンがあります。

既定のスキンは、あるテーマがページに適用されるとき、同じ種類のすべてのコントロールに自動的に適用されます。コントロールスキンは、SkinID 属性が存在しない場合、既定のスキンになります。

名前指定スキンは、SkinID プロパティが設定されたコントロールスキンです。名前指定スキンを作成すると、アプリケーション内の同じコントロールのインスタンスごとに異なるスキンを設定できます。

カスケーディング スタイル シート

テーマには、**カスケーディング スタイル シート (.css ファイル)** も含めることができます。テーマフォルダ内でファイル名拡張子 .css を使用してスタイルシートを定義します。

テーマのグラフィックとその他のリソース

テーマには、グラフィックや、スクリプトファイルやサウンドファイルなど、その他のリソースも含めることができます。

通常、あるテーマのリソースファイルはそのテーマのスキンファイルと同じフォルダに置かれますが、テーマフォルダのサブフォルダなど、Web アプリケーション内のどこにでも配置できます。テーマフォルダのサブフォルダ内のリソースファイルを参照するには、この Image コントロールスキンのようなパスを使用します。

```
<asp:Image runat="server" ImageUrl="ThemeSubfolder/filename.ext" />
```

また、テーマフォルダの外部にリソースファイルを格納することもできます。次の例のように、~/SubFolder/filename.ext 形式のパスを使用してリソース ファイルを参照できます。

```
<asp:Image runat="server" ImageUrl="~/AppSubfolder/filename.ext" />
```

テーマのスコープの設定

テーマは、単一の Web アプリケーションに対して定義することも、Web サーバー上のすべてのアプリケーションが使用するグローバルなテーマとして定義することもできます。テーマを定義した後、**@ Page** ディレクティブの **Theme** 属性または **StyleSheetTheme** 属性を使用して個々のページにテーマを配置することも、アプリケーション構成ファイルで **<pages>** 要素を設定することにより、アプリケーション内のすべてのページに適用することもできます。

ページ テーマ

ページテーマは 1 つのテーマフォルダです。このフォルダでは、コントロール スキン、スタイル シート、グラフィックファイル、その他のリソースが Web サイトの ¥App_Themes フォルダのサブフォルダとして作成されます。各テーマは、¥App_Themes フォルダのそれぞれ個別のサブフォルダになります。次の例は、BlueTheme と PinkTheme という名前の 2 つのテーマを定義する一般的なページテーマを示しています。

```
- MyWebSite
  - App_Themes
    - BlueTheme
      - Controls.skin
      - BlueTheme.css
    - PinkTheme
      - Controls.skin
      - PinkTheme.css
```

グローバル テーマ

グローバルテーマとは、サーバー上のすべての Web サイトに適用できるテーマです。グローバルテーマを使用すると、同じサーバーに複数の Web サイトがあるときに、そのドメインの全体的な外観を定義できます。

グローバルテーマは、プロパティ設定、スタイルシート設定、およびグラフィックを含む点では、ページテーマと同じです。ただし、グローバルテーマは、Web サーバーにとってグローバルな Themes フォルダに保存されます。サーバー上のすべての Web サイトから、また、Web サイト内のすべてのページからグローバルテーマを参照できます。

テーマ設定の優先順位

テーマをどのように適用するかを指定することにより、ローカルのコントロール設定に対するテーマ設定の優先順位を指定できます。

ページの **Theme** プロパティを設定すると、テーマとページのコントロール設定がマージされて、そのコントロールに対する最終的な設定になります。コントロールとテーマの両方でコントロール設定が定義されている場合は、そのコントロールに対するどのページ設定よりも、テーマのコントロール設定が優先されます。そのため、ページのコントロールの各プロパティが既に設定されている場合でも、各ページにまたがって一貫した外観を作成できます。たとえば、以前のバージョンの ASP.NET で作成したページにテーマを適用できます。

また、ページの **StyleSheetTheme** プロパティを設定すると、テーマをスタイル シート テーマとして適用できます。この場合、設定が両方で定義されていると、テーマで定義されている設定よりもローカルのページ設定が優先されます。これがカスケーディング スタイル シートによって使用されるモデルです。ページ内の個別のコントロールのプロパティを設定しながら、全体的な外観としてのテーマも適用できるようにするには、テーマをスタイル シート テーマとして適用するという方法があります。

テーマを使用して定義できるプロパティ

一般に、テーマは、ページやコントロールの外観または静的コンテンツに関連するプロパティの設定に使用します。設定できるプロパティは、コントロールクラスで **ThemeableAttribute** 属性が true になっているものだけです。

コントロールの外観ではなく動作を明示的に指定するプロパティには、テーマの値を指定できません。たとえば、**Button** コントロールの **CommandName** プロパティは、テーマを使用して設定することはできません。また、GridView コントロールの AllowPaging プロパティや **DataSource** プロパティもテーマを使用して設定することはできません。

テーマとカスケード スタイル シート

テーマと**カスケーディング スタイル シート**は、任意のページに適用できる共通の属性セットを定義するという点で似ています。ただし、テーマは次の点でスタイルシートと異なります。

- テーマは、スタイル プロパティだけでなく、コントロールやページのさまざまなプロパティを定義できます。たとえば、テーマを使用すると、**TreeView** コントロールのグラフィックや **GridView** コントロールのテンプレートレイアウトなどを指定できます。
- テーマにはグラフィックを含めることができます。
- テーマには、スタイルシートとは異なり、優先順位がありません。**StyleSheetTheme** プロパティを使用してテーマを明示的に適用する場合を除き、既定では、ページの **Theme** プロパティで参照されるテーマ内で定義されているプロパティ値は、宣言によってコントロールに設定されているプロパティ値よりも優先されます。
- 各ページに適用できるテーマは 1 つだけです。

セキュリティに関する検討事項

Web サイトでテーマを使用すると、セキュリティ問題が発生する場合があります。また、悪意のあるテーマを使用することにより、次のようなことが行われる可能性があります。

- 目的どおりに動作しないようにコントロールの動作を変更する。
- クライアント側スクリプトを挿入して、クロス サイト スクリプティングのリスクを発生させる。
- 妥当性検査の内容を変更する。

- 機密情報を公開する。

このような一般的な脅威を緩和する方法を次に示します。

- グローバルテーマおよびアプリケーションテーマのディレクトリは、適切なアクセス制御設定で保護します。
- 信頼できないソースのテーマは使用を避けます。
- クエリデータにテーマ名を公開しないようにします。

ASP.NET ページ テーマを定義する

Visual Studio や **Visual Web Developer Express** では、ページテーマを定義して、アプリケーションの 1 つ以上のページに適用できます。また、マシンレベルでテーマを作成してサーバー上の複数のアプリケーションで使用することもできます。

テーマは、いくつかのサポートファイルで構成され、サポートファイルにはページの外観を決めるスタイル シート、サーバーコントロールの外観を定義するコントロールスキン、テーマを構成するその他のサポートイメージまたはファイルが含まれます。ページテーマかグローバルテーマかにかかわらずテーマの内容は同じです。

テーマを適用するためには、**@ Page** ディレクティブの **Theme** 属性または **StyleSheetTheme** 属性を使用するか、アプリケーション構成ファイルに **pages** 要素 (ASP.NET 設定スキーマ) 要素を設定します。

ページ テーマを作成するには

以下のような手順で行います。

1. [ソリューション エクスプローラー] で、ページテーマを作成する Web サイトの名前を右クリックし、[ASP.NET フォルダの追加] をクリックします。
2. [テーマ] をクリックします。
3. App_Themes フォルダが存在しない場合は、Visual Studio または Visual Web Developer Express によって作成されます。次に、App_Themes フォルダの子フォルダとしてテーマ用の新しいフォルダが作成されます。
4. 新しいフォルダの名前を入力します。
5. このフォルダ名は、ページテーマ名としても使用されます。たとえば、¥App_Themes¥FirstTheme という名前のフォルダを作成すると、テーマ名が FirstTheme となります。
6. 新しいフォルダに、テーマを構成するコントロール スキン、スタイル シート、およびイメージ用のファイルを追加します。

スキン ファイルとスキンをページ テーマに追加するには

以下のような手順で行います。

1. [ソリューション エクスプローラー] でテーマ名を右クリックし、[新しい項目の追加] をクリックします。
2. [新しい項目の追加] ダイアログボックスで [スキン ファイル] をクリックします。
3. [名前] ボックスに .skin ファイルの名前を入力し、[追加] をクリックします。
4. 通常は 1 つのコントロールに対して 1 つの **.skin** ファイル (Button.skin、Calendar.skin など) が作成されます。ただし、必要に応じて任意の数の .skin ファイルを作成できます。
5. .skin ファイルには、宣言構文を使用して通常のコントロール定義を追加し、テーマに設定するプロパティのみを含めます。コントロール定義には、必ず runat="server" 属性を含める必要があり、ID="" 属性を含めることはできません。
6. テーマの中のすべての Button コントロールの色とフォントを定義する Button コントロールの既定のコントロール スキンについて、コード例を次に示します。

```
<asp:Button runat="server"
  BackColor="Red"
  ForeColor="White"
  Font-Name="Arial"
  Font-Size="9px" />
```

6. 作成するコントロール スキン ファイルのそれぞれに対して、手順 2 と 3 を繰り返します。

ページ テーマにカスケーディング スタイル シート ファイルを追加するには

以下のような手順で行います。

1. [ソリューション エクスプローラー] で、テーマ名を右クリックし、[新しい項目の追加] をクリックします。
2. [新しい項目の追加] ダイアログ ボックスで、[スタイル シート] をクリックします。
3. [名前] ボックスに .css ファイルの名前を入力し、[追加] をクリックします。
4. テーマをページに適用すると、ASP.NET によりページの **head** 要素に、このスタイル シートへの参照が追加されます。

グローバル テーマの作成

グローバル テーマは、サーバー上のすべての Web サイトに適用されます。

グローバル テーマを作成するには

以下のような手順で行います。

1. 次のパスを使用して、Themes フォルダを作成します。

```
%windows%¥Microsoft.NET¥Framework¥version¥ASP.NETClientFiles¥Themes
```

2. Themes フォルダに、グローバル テーマ ファイルを格納するためのサブフォルダを作成します。サブフォルダ名は、テーマ名になります。たとえば、¥Themes¥FirstTheme という名前のフォルダを作成すると、テーマ名が FirstTheme となります。
3. 新しいフォルダに、グローバルテーマを構成するコントロール スキン、スタイル シート、およびイメージ用のファイルを追加します。
4. ファイルシステム Web サイトを ASP.NET 開発サーバーでテストしている場合は、テーマをそのままテストできます。
5. ローカルの IIS Web サイトを使用して Web サイトのテストを行っている場合は、コマンド ウィンドウを開き、aspnet_regiis -c を実行して、IIS を実行しているサーバーにテーマをインストールします。
6. テーマをリモート Web サイト、または FTP Web サイトでテストしている場合は、次のパスを使用して Themes フォルダを手動で作成する必要があります。

```
IISRootWeb¥aspnet_client¥system_web¥version¥Themes
```

ASP.NET テーマを適用する

テーマは、ページ、Web サイト、またはグローバルに適用できます。Web サイト レベルでテーマを設定すると、個々のページのテーマをオーバーライドしない限り、スタイルとスキンがサイト内のすべてのページとコントロールに適用されます。ページ レベルでテーマを設定すると、スタイルとスキンがページとページのすべてのコントロールに適用されます。

テーマを Web サイトに適用するには

次のような手順で行います。

1. アプリケーションの Web.config ファイルで、次の例のように <pages> 要素をグローバルテーマまたはページテーマの名前に設定します。

```
<configuration>
    <system.web>
        <pages theme="ThemeName" />
    </system.web>
</configuration>
```

2. テーマをスタイル シート テーマとして設定し、ローカル コントロールの設定に従属させる場合は、代わりに **styleSheetTheme** 属性を設定します。

```
<configuration>
    <system.web>
```

```
        <pages styleSheetTheme="Themename" />
    </system.web>
</configuration>
```

Web.config ファイルのテーマの設定は、アプリケーションのすべての ASP.NET Web ページに適用されます。Web.config ファイルのテーマの設定は、通常の構成階層の規則に従います。たとえば、テーマをページのサブセットにのみ適用する場合は、適用するページを専用の Web.config ファイルと共にフォルダに格納するか、またはルート Web.config ファイルに <location> 要素を作成してフォルダを指定します。

テーマを個々のページに適用するには

次の例に示すように、@ Page ディレクティブの Theme 属性または StyleSheetTheme 属性を使用するテーマの名前に設定します。

```
<%@ Page Theme="ThemeName" %>
<%@ Page StyleSheetTheme="ThemeName" %>
```

これで、テーマとそれに対応するスタイルおよびスキンが、テーマを宣言しているページにのみ適用されます。

コントロールへのスキンの適用

テーマで定義されているスキンは、テーマが適用されたアプリケーションまたはページのすべてのコントロールのインスタンスに適用されます。場合によっては、個々のコントロールに特定のプロパティセットを適用する必要があることがあります。それには、名前指定スキン (SkinID プロパティが設定されている .skin ファイルのエントリ) を作成し、ID を使用して個々のコントロールに適用します。

名前指定スキンをコントロールに適用するには

次の例に示すように、コントロールの **SkinID** プロパティを設定します。

```
<asp:Calendar runat="server" ID="DatePicker" SkinID="SmallCalendar" />
```

ページ テーマに SkinID プロパティに一致するコントロール スキンがない場合は、そのコントロール型の既定のスキンが使用されます。

ASP.NET Web サーバー コントロールとブラウザーの機能

種類の異なるブラウザー、または同一種類であってもバージョンの異なるブラウザーではサポートする機能がそれぞれ異なっています。ASP.NET サーバーコントロールは、ページを要求したブラウザーを自動的に判別し、ブラウザーに適合したマークアップをレンダリングします。

ブラウザーの種類の検出

既定では、ASP.NET は要求時にブラウザーからサーバーに渡されるユーザーエージェント情報を読み取ってブラウザーの機能を判定します。このとき、ブラウザーから受け取ったユーザーエージェント文字列と、ブラウザーの定義ファイルに保存されているユーザーエージェント文字列が比較されます。このブラウザーの定義ファイルには、さまざまなユーザーエージェントの機能に関する情報が格納されています。ASP.NET で、現在のユーザーエージェント文字列とブラウザー定義ファイルのユーザーエージェント文字列が一致したことが検出された場合は、対応するブラウザー機能が **HttpBrowserCapabilities** オブジェクトに読み込まれます。

HttpBrowserCapabilities オブジェクトは、**HttpRequest** オブジェクトの **Browser** プロパティで利用できます。

たとえば、現在のブラウザーの種類およびバージョンが特定のバージョンの JavaScript に対応しているかどうかを確認するには、次の例に示すように、**HttpBrowserCapabilitiesBase** ではなくブラウザーの定義ファイルだけで定義されているプロパティを使用します。

<Visual Basic>

```
Dim jsVersion as String
jsVersion = Request.Browser("JavaScriptVersion")
```

<C#>

```
string jsVersion = Request.Browser["JavaScriptVersion"];
```

要求を行っているブラウザーが Internet Explorer 8 である場合、**jsVersion** 文字列には値 "1.5" が含まれます。

要求がモバイル デバイスから行われたものであるかどうかを確認するには、次の例に示すように、**IsMobileDevice** プロパティを使用します。

<Visual Basic>

```
Dim isMobile as Boolean
isMobile = Request.Browser.IsMobileDevice
```

<C#>

```
bool isMobile = Request.Browser.IsMobileDevice;
```

ASP.NET では、次のフォルダに既定のブラウザー定義ファイルが保存されます。

```
%SystemRoot%¥Microsoft.NET¥Framework¥versionNumber¥Config¥Browsers
```

検出されたブラウザーの型のオーバーライド

ブラウザーを自動検出するのではなくページのレンダリング方法を明示的に制御する場合は、ページの **ClientTarget** プロパティを設定します。このプロパティは、ページの @ Page ディレクティブの属性として宣言して設定するか、またはプログラムによって設定します。

ClientTarget プロパティの値は、ページをレンダリングするブラウザーの種類を表すエイリアスです。追加のエイリアスを作成するには、ルート **Web.config** ファイルまたはアプリケーションの Web.config ファイルで定義します。

AJAX 対応の ASP.NET のコントロールと機能

ASP.NET の **AJAX** 対応の機能は、大部分の最新ブラウザーとの間に互換性があり、ブラウザーの既定のセキュリティ設定で実行できます。AJAX 対応のコントロールと機能を使用するには、クライアント スクリプトを実行する機能をブラウザーが備えている必要があります。**UpdatePanel** コントロールと **ScriptManager** コントロールは、AJAX 対応コントロールの例です。

クライアント スクリプト

ASP.NET サーバーコントロールの一部の機能は、クライアントスクリプトを実行できる環境を必要とします。ブラウザーがスクリプトの実行機能を備えている場合は、クライアントスクリプトが自動的に生成され、ページの一部として送信されます。

Web サーバー コントロールと CSS スタイル

ForeColor、**BackColor**、**Height**、**Width** などの各種の表示形式プロパティを設定することにより、**ASP.NET サーバー コントロール**の外観を制御できます。

ブラウザーへの表示形式プロパティのレンダリング

ページが実行されると、表示形式プロパティはユーザーが使用しているブラウザーの機能に応じてレンダリングされます。ユーザーのブラウザーがカスケーディング スタイル シート (CSS) をサポートしている場合、表示形式プロパティはコントロールを構成する HTML 要素のスタイル属性としてレンダリングされます。たとえば、サーバー コントロールを定義し、ForeColor プロパティを Red に設定し、Bold プロパティを true に設定し、Size プロパティを xx-small に設定しているときに、ユーザーのブラウザーがスタイル シートをサポートしている場合、コントロールは次のように描画されます。

```
<a id="hyperlink1"
    style="color: red; font-size: xx-small; font-weight: bold;">
  HyperLink
</a>
```

ブラウザーがスタイルをサポートしていない場合、コントロールは他の方法 (<font> 要素など) を使用してレンダリングされます。上記の例は、スタイルをサポートしないブラウザーでは次のようにレンダリングされます。

```
<a id="a1"><b><font color="red" size="1">HyperLink</font></b></a>
```

コントロールのスタイル オブジェクト

コントロールは、ForeColor、BackColor などの表示形式プロパティの他に、追加の表示形式プロパティをカプセル化した 1 つ以上のスタイル オブジェクトを公開します。1 つの例は **Font** スタイル プロパティで、**Size**、**Name**、**Bold** などのフォントに関する個々のプロパティを含む FontInfo 型のオブジェクトを公開します。

スタイル オブジェクトの優先順位と継承

複雑なコントロールでは、スタイルオブジェクトが他のスタイルオブジェクトの特性を継承することがよくあります。たとえば、**Calendar** コントロールの **SelectedDayStyle** オブジェクトは、**DayStyle** オブジェクトに基づいています。SelectedDayStyle のプロパティを明示的に設定しない場合は、DayStyle オブジェクトの特性が継承されます。

この継承は、設定するスタイルオブジェクトのプロパティに優先順位があることを意味しています。たとえば、Calendar コントロールのスタイルオブジェクトのプロパティについて、優先順位の一覧を示します。ここでは、優先順位が低い順に示しています。

1. 基本の Calendar コントロールの表示形式プロパティ
2. DayStyle スタイル オブジェクト
3. WeekendDayStyle スタイル オブジェクト
4. OtherMonthDayStyle スタイル オブジェクト
5. TodayDayStyle スタイル オブジェクト
6. SelectedDayStyle スタイル オブジェクト

コントロールのスタイルを設定する最善の方法は、コントロールによって定義されるスタイルプロパティを使用した上で、必要に応じてスタイルシートまたはインラインスタイルを使用して細かい調整を個々の要素に加えることです。コントロールのスタイルプロパティによって定義されるスタイルをオーバーライドするには、スタイルシートまたはインラインスタイルに **!important CSS 規則**を使用します。

次のコード例は、hovernodestyle 要素に CssClass プロパティを使用しています。このクラスは a:visited 定義をオーバーライドするために、myclass と a.myclass:visited の 2 回定義されます。

```
<%@ Page Language="C#" %>
<html>
<head runat="server">
  <asp:sitemapdatasource id="SiteMapSource" runat="server" />
  <style type="text/css">
    a:visited
    {
      color: #000066
    }
    myclass, a.myclass:visited {
      color: #FF0000
    }
  </style>
</head>
<body>
  <form runat="server">
    <a href="http://www.Contoso.com">Contoso</a>
    <asp:treeview id="treeview1" runat="server"
```

```
        initialexpanddepth="1"
        datasourceid="SiteMapSource"
        forecolor="#444444"
        font-names="Verdana"
        font-size="0.8em">
      <nodestyle font-bold="true" />
      <hovernodestyle cssclass=myclass />
    </asp:treeview>
  </form>
</body>
</html>
```

CSS スタイルおよびクラスの直接制御

コントロールは、表示形式プロパティとスタイルオブジェクトに加えて、CSS スタイルをより直接的に操作するために **CssClass** プロパティと **Style** プロパティの 2 つのプロパティを公開します。CssClass プロパティを使用すると、コントロールにスタイルシートクラスを割り当てることができます。Style プロパティを設定することにより、スタイル属性の文字列がコントロールにそのまま書き込まれるように設定できます。Style プロパティを使用すると、他のプロパティでは公開されないスタイル属性を設定できます。Style プロパティは、**Add**、**Remove** などのメソッドのコレクションを公開します。これらのメソッドを呼び出せば、スタイルを直接設定できます。

ASP.NET キャッシュの概要

アプリケーションのパフォーマンスを向上できるように、ASP.NET には、2 つの基本的なキャッシュ機構が用意されています。アプリケーションキャッシュとページ出力キャッシュです。

アプリケーション キャッシュ

アプリケーションキャッシュは、キー/値ペアを使用して任意のデータをプログラムによってメモリに格納する方法を提供します。アプリケーションキャッシュは、アプリケーション状態と同様に使用できます。ただし、アプリケーション状態とは異なり、アプリケーションキャッシュ内のデータは揮発性です。つまり、アプリケーションの起動中に常にメモリに保持されているわけではありません。アプリケーションキャッシュを使用する利点は、キャッシュが ASP.NET によって管理され、項目が期限切れまたは無効になるか、メモリが不足すると、項目が自動的に削除される点です。項目を削除するときにはアプリケーションに通知するようにアプリケーションキャッシュを構成することもできます。

ページ出力キャッシュ

ページ出力キャッシュは、処理された ASP.NET ページのコンテンツをメモリに格納します。これにより、ASP.NET は、ページ処理ライフサイクルを繰り返すことなく、ページの応答をクライアントに送信できます。ページ出力キャッシュは、頻繁には変更されないが作成に多くの処理を要するページに特に有効です。たとえば、トラフィック量は多いが更新頻度の低いデータを表示する Web ページを作成する場合は、ページ出力キャッシュによってページのパフォーマンスを大幅に向上できます。ページキャッシュは、ページごとに個別に構成できます。または、Web.config ファイルでキャッシュプロファイルを作成すると、キャッシュ設定を一度定義するだけで、その設定を複数のページで使用できます。

ページ出力キャッシュには、**フル ページ キャッシュ**と**部分ページ キャッシュ**の 2 つのモデルが用意されています。フル ページ キャッシュでは、ページの全コンテンツがメモリに保持されて、クライアントの要求を満たすために使用されます。部分ページキャッシュでは、ページの一部のみがキャッシュされ、他の部分は動的に処理されます。

部分ページ キャッシュは、**コントロール キャッシュ**と**キャッシュ後置換**の 2 つの方法で動作します。コントロールキャッシュは、**フラグメント キャッシュ**とも呼ばれます。これは、ユーザーコントロールに情報を格納してから、そのユーザーコントロールにキャッシュ可能のマークを付けるという方法でページ出力の一部をキャッシュします。

キャッシュ後置換は、コントロールキャッシュと逆です。ページ全体をキャッシュし、ページの一部分のみが動的に処理されます。

要求パラメーターに基づくページのキャッシュ

ASP.NET のページ出力キャッシュでは、ページの単一バージョンをキャッシュできる他に、さまざまな要求パラメーターに基づいて内容が異なる複数のページ バージョンを作成する機能が提供されています。

自動データ削除

ASP.NET は、次のいずれかの理由でキャッシュからデータを削除します。

- サーバーのメモリが不足し、クリーンアッププロセスが実行される場合。
- キャッシュ内の項目が期限切れになった場合。
- 項目の依存関係が変化した場合。

アプリケーションは、キャッシュされている項目を管理できるように、項目がキャッシュから削除されるときに ASP.NET から通知を受けることができます。

Scavenging (清掃)

Scavenging は、メモリが不足したときにキャッシュから項目を削除する処理です。項目は、一定時間アクセスされていない場合、またはキャッシュに追加されたときに低い優先順位が付けられた場合に削除されます。ASP.NET は、**CacheItemPriority** オブジェクトを使用して、項目を清掃する順番を決定します。

有効期限

ASP.NET は、項目が期限切れになった場合にも、キャッシュから項目を自動的に削除します。項目をキャッシュに追加する際に、次の表のように項目の有効期限を設定できます。

表. キャッシュの有効期限

有効期限の種類	説明
スライド式有効期限	項目が最後にアクセスされてから期限切れになるまでの期間を指定します。たとえば、キャッシュ内の項目が最後にアクセスされてから 20 分後に期限切れになるように設定できます。
絶対有効期限	アクセスされる頻度に関係なく、設定された時間に項目が期限切れになるように指定します。たとえば、項目が午後 6 時または 4 時間後に期限切れになるように設定できます。

依存関係

キャッシュ内の項目の有効期間が、ファイル、データベースなどの他のアプリケーション要素に依存するように構成できます。キャッシュ項目が依存する要素が変更されると、ASP.NET は項目をキャッシュから削除します。

ASP.NET のキャッシュでは、次の表に示す**依存関係**がサポートされています。

表. キャッシュの依存関係

依存関係	説明
キー依存関係	アプリケーション キャッシュ内の項目がキー/値のペアで格納されます。キー依存関係を使用すると、ある項目をアプリケーションキャッシュ内の別の項目のキーに依存させることができます。元の項目が削除されると、キー依存関係がある項目も削除されます。たとえば、**ReportsValid** というキャッシュ項目を追加した後で、ReportsValid キーに依存するいくつかのレポートをキャッシュしたとします。ReportsValid 項目が削除されると、依存関係にあってキャッシュされているすべてのレポートもキャッシュから削除されます。
ファイル依存関係	キャッシュ内の項目が外部ファイルに依存します。外部ファイルが変更または削除されると、キャッシュ内の項目も削除されます。
SQL 依存関係	キャッシュ内の項目が Microsoft SQL Server 2005、SQL Server 2000、または SQL Server 7.0 データベースのテーブルの変更に依存します。SQL Server 2005 の場合は、項目をテーブル内の行に依存させることができます。
集合依存関係	キャッシュ内の項目が **AggregateCacheDependency** クラスの使用によって複数の要素に依存します。いずれかの依存関係が変更されると、項目はキャッシュから削除されます。
カスタム依存関係	キャッシュ内の項目が独自コードで作成された依存関係によって構成されます。たとえば、Web サービスの呼び出し結果が特定の値になったときにキャッシュからデータを削除するように Web サービス キャッシュのカスタム依存関係を作成できます。

アプリケーション キャッシュ項目の削除通知

アプリケーション キャッシュから項目が削除されるときに通知を受け取ることができます。たとえば、作成にかなりの処理時間を要する項目がある場合は、その項目がキャッシュから削除されたときに直ちに再配置できるように通知を受け取ることができます。こうすれば、次にその項目を要求したユーザーが、項目が処理されるまで待たずに済みます。

アプリケーション データのキャッシュの詳細

ASP.NET には、強力で使いやすいキャッシュ機構が用意されています。これにより、作成時に大量のサーバーリソースを必要とするオブジェクトをメモリ内に格納することができます。このような

リソースをキャッシュすることにより、アプリケーションのパフォーマンスを大幅に向上させることができます。

キャッシュは、**Cache** クラスによって実装されます。Cache クラスは、使いやすさを重視して設計されています。Cache にアイテムを追加し、後から簡単なキー/値ペアを使用して追加したアイテムを取得できます。

Cache クラスは、項目がキャッシュされる方法とキャッシュされる期間をカスタマイズできる強力な機能を提供します。たとえば、システムメモリが不足してくると、キャッシュは頻繁に使用しない優先順位が低い項目を自動的に削除して、メモリを解放します。この方法は Scavenging (清掃) と呼ばれ、期限切れのデータが貴重なサーバーリソースを消費しないようにキャッシュが実行する処理の 1 つです。

Cache オブジェクトが清掃を行う際に、ある項目を他の項目よりも優先するように指定できます。項目の優先順位を示すには、**Add** メソッドまたは **Insert** メソッドを使用して項目を追加する際に **CacheItemPriority** 列挙値の 1 つを指定します。

Add メソッドまたは Insert メソッドを使用して項目をキャッシュに追加する際に、有効期限ポリシーも設定できます。項目の有効期限を定義するには、**DateTime** 値を使用して、項目が有効期限切れになる正確な時間 (絶対有効期限) を指定します。また、**TimeSpan** 値を使用して、スライド式有効期限を指定することもできます。この方法では、項目の前回のアクセス時間に基づいて有効期限が切れるまでの経過時間を指定できます。有効期限が切れた項目は、キャッシュから削除されます。この値を取得しようとすると、その項目が再度キャッシュに追加されない限り、null (Visual Basic の場合は Nothing) が返されます。

定期的にデータのリフレッシュが行われる項目や、指定した期間だけ有効な項目などがキャッシュに格納される場合、通常は、これらの揮発性の項目に有効期限ポリシーを設定して、そのデータが現在のデータである間だけキャッシュに保持されるようにします。たとえば、別の Web サイトからデータを取得してスポーツの得点経過を追跡するアプリケーションを作成する場合に、情報ソースの Web サイトでその試合の得点が変更されない限り、得点をキャッシュしておくことができます。この場合、有効期限ポリシーは、他の Web サイトで得点データが更新される間隔に基づいて設定できます。キャッシュに最新の得点データが保持されているかどうかを判断するためのコードを作成できます。得点データが最新でない場合は、そのコードによって情報ソースの Web サイトから得点データが読み取られ、新しい値がキャッシュされます。

最後に、ASP.NET では、キャッシュされている項目の有効性を外部のファイルやディレクトリ (ファイル依存関係)、またはキャッシュされているその他の項目 (キー依存関係) に基づいて定義できます。依存関係が関連付けられている項目が変更されると、キャッシュされている項目は無効になり、キャッシュから削除されます。この方法を使用すると、項目のデータ ソースが変更されたときに、そ

れらの項目をキャッシュから削除できます。たとえば、XML ファイルに格納された金融データを処理するアプリケーションを作成する場合に、その XML ファイル内での依存関係を維持したまま、ファイルのデータをキャッシュに挿入できます。このファイルが更新されると、項目はキャッシュから削除されます。アプリケーションは XML ファイルを再度読み込み、更新後のデータをキャッシュに挿入します。

宣言による ASP.NET ページのキャッシュの設定

デザイン時にページに必要なキャッシュの設定がわかっている場合は、宣言によってキャッシュを設定できます。この場合、ページは、すべての要求に同じキャッシュの設定を使用します。

ページのキャッシュの設定を行うには

1. ページに **@ OutputCache** ディレクティブを追加し、**Duration** 属性と **VaryByParam** 属性を定義します。
2. @ OutputCache ディレクティブに **Location** 属性を追加し、その値を **OutputCacheLocation** 列挙の Any、Client、Downstream、Server、ServerAndClient、None のいずれかの値に指定します。
3. ページのキャッシュを 60 秒に設定する方法について、コード例を次に示します。

```
<%@ OutputCache Duration="60" VaryByParam="None" %>
```

キャッシュプロファイルを使用してページのキャッシュを設定するには

1. アプリケーションの Web.config ファイルでキャッシュプロファイルを定義し、プロファイルに **duration** 設定と **varyByParam** 設定を追加します。次の **<caching>** 構成要素は、サーバーにページを 30 秒間キャッシュする **Cache30Seconds** というキャッシュプロファイルを定義しています。

```
<caching>
  <outputCacheSettings>
    <outputCacheProfiles>
      <add name="Cache30Seconds" duration="30"
        varyByParam="none" />
    </outputCacheProfiles>
  </outputCacheSettings>
</caching>
```

2. このプロファイルを使用する各 ASP.NET ページに **@ OutputCache** ディレクティブを追加し、CacheProfile 属性を Web.config ファイルで定義したキャッシュプロファイルの名前に設定します。Cache30Seconds というキャッシュプロファイルを使用するページのコード例を次に示します。

```
<%@ OutputCache CacheProfile="Cache30Seconds" %>
```

ASP.NET ページのキャッシュに有効期限値を設定する

ページを出力キャッシュに追加するには、そのページの**有効期限ポリシー**を設定する必要があります。これは、宣言またはプログラムによって実行できます。

ページの出力キャッシュの有効期限を宣言によって設定するには

応答をキャッシュする ASP.NET ページ (.aspx ファイル) に **@ OutputCache** ディレクティブを組み込みます。**Duration** 属性を正の数値に設定し、**VaryByParam** 属性を任意の値に設定します。たとえば、次の @ OutputCache ディレクティブはページの有効期限を 60 秒に設定します。

```
<%@ OutputCache Duration="60" VaryByParam="None" %>
```

ページの出力キャッシュの有効期限をプログラムによって設定するには

ページのコードで、**Response** オブジェクトの **Cache** プロパティにページの有効期限ポリシーを設定します。

次のコード例は、前の説明で @ OutputCache ディレクティブによって設定したのと同じキャッシュポリシーを設定します。

<C#>

```
Response.Cache.SetExpires(DateTime.Now.AddSeconds(60));
Response.Cache.SetCacheability(HttpCacheability.Public);
Response.Cache.SetValidUntilExpires(true);
```

<Visual Basic>

```
Response.Cache.SetExpires(DateTime.Now.AddSeconds(60))
Response.Cache.SetCacheability(HttpCacheability.Public)
Response.Cache.SetValidUntilExpires(True)
```

キャッシュされたページの有効期限が切れ、それ以降にそのページが要求された場合は、応答が動的に生成されます。この応答ページは、指定した存続期間だけキャッシュされます。

キャッシュされたページの有効性をチェックする

ユーザーがキャッシュされたページを要求すると、ASP.NET は、ページに定義されたキャッシュポリシーに基づいて、キャッシュされた出力がまだ有効かどうかを判断します。出力が有効な場合、キャッシュされた出力がクライアントに送信され、ページの再処理は行われません。ただし、ページが有効かどうかをチェックするカスタム ロジックを記述できるように、ASP.NET には、この検証チェック時に検証コールバックを使用してコードを実行する機能が用意されています。検証コールバックを使用すると、キャッシュの依存関係を使用する通常のプロセスの範囲外にあるキャッシュされたページを無効化できます。

キャッシュされたページの有効性をプログラムでチェックするには

1. **HttpCacheValidateHandler** 型のイベントハンドラを定義し、キャッシュされたページの応答の有効性をチェックするコードを組み込みます。
3. 検証ハンドラは、次のいずれかの **HttpValidationStatus** 値を返す必要があります。
   - **Invalid** —— キャッシュされたページが無効であることを示します。ページはキャッシュから追い出され、要求はキャッシュミスとして処理されます。
   - **IgnoreThisRequest** —— 要求をキャッシュミスとして扱います。このため、再度ページが処理されますが、キャッシュされたページは無効化されません。
   - **Valid** —— キャッシュされたページが有効であることを示します。
4. クエリ文字列変数 **status** に値 "invalid" または "ignore" が格納されているかどうかを判断する、**ValidateCacheOutput** という名前の検証ハンドラについて、コード例を次に示します。ステータス値が "invalid" の場合、メソッドは Invalid を返し、キャッシュ内のページが無効化されます。ステータス値が "ignore" の場合、メソッドは IgnoreThisRequest を返し、ページはキャッシュに残されますが、この要求に対して新しい応答が生成されます。

<C#>

```
public static void ValidateCacheOutput(HttpContext context,
    Object data, ref HttpValidationStatus status)
{
    if (context.Request.QueryString["Status"] != null)
    {
        string pageStatus = context.Request.QueryString["Status"];

        if (pageStatus == "invalid")
            status = HttpValidationStatus.Invalid;
        else if (pageStatus == "ignore")
            status = HttpValidationStatus.IgnoreThisRequest;
        else
            status = HttpValidationStatus.Valid;
    }
    else
        status = HttpValidationStatus.Valid;
}
```

<Visual Basic>

```
Public Shared Sub ValidatePage(ByVal context As HttpContext, _
       ByVal data As [Object], ByRef status As HttpValidationStatus)
```

```
    If Not (context.Request.QueryString("Status") Is Nothing) Then
        Dim pageStatus As String =
                            context.Request.QueryString("Status")
        If pageStatus = "invalid" Then
            status = HttpValidationStatus.Invalid
        ElseIf pageStatus = "ignore" Then
            status = HttpValidationStatus.IgnoreThisRequest
        Else
            status = HttpValidationStatus.Valid
        End If
   Else
       status = HttpValidationStatus.Valid
   End If
End Sub
```

5. ページのライフサイクルイベントの 1 つ (ページの Load イベントなど) から **AddValidationCallback** メソッドを呼び出し、手順 1 で定義したイベント ハンドラの最初の引数として渡します。
6. 次のコード例は、検証ハンドラにする **ValidateCacheOutput** メソッドを設定します。

<C#>

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Cache.AddValidationCallback(
        new HttpCacheValidateHandler(ValidateCacheOutput),
        null);
}
```

<Visual Basic>

```
Protected Sub Page_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Load

    Response.Cache.AddValidationCallback( _
        New HttpCacheValidateHandler(AddressOf ValidatePage), Nothing)
End Sub
```

キャッシュにアイテムを追加する

Cache オブジェクトを使用すると、アプリケーションキャッシュ内のアイテムにアクセスできます。Cache オブジェクトの **Insert** メソッドを使用すると、アプリケーションキャッシュにアイテムを追加できます。このメソッドは、キャッシュにアイテムを追加します。また、アイテムに依存関係、有効期限、および削除通知を設定するための各オプションを付加する複数のオーバーロードを備えています。Insert メソッドを使用してキャッシュにアイテムを追加するとき、同じ名前の項目が既に存在すると、キャッシュ内のアイテムは置き換えられます。

Add メソッドを使用して、キャッシュにアイテムを追加することもできます。このメソッドでは、Insert メソッドと完全に同じオプションを設定できます。ただし、Add メソッドは、キャッシュに追加されたオブジェクトを返します。また、Add メソッドを使用した場合は、同じ名前のアイテムが既にキャッシュに存在してもメソッドは既存のアイテムを置き換えず、例外も発生しません。

このトピックの手順では、アプリケーションキャッシュにアイテムを追加する次の方法について説明します。

- キーと値によってアイテムを直接設定してキャッシュにアイテムを追加する。
- Insert メソッドを使用してキャッシュにアイテムを追加する。
- キャッシュにアイテムを追加し、依存関係が変更されたときにキャッシュからアイテムを削除するように依存関係を追加する。キャッシュの他のアイテム、ファイル、および複数のオブジェクトに基づいて依存関係を設定できます。
- 有効期限ポリシーを付けてキャッシュにアイテムを追加する。アイテムの依存関係を設定できる以外に、経過時間の後（スライド式有効期限）や特定の時間（絶対有効期限）に有効期限が切れるようにアイテムを設定できます。スライド式有効期限または絶対有効期限のいずれかを定義できますが、同時に定義することはできません。
- キャッシュにアイテムを追加し、キャッシュされたアイテムの相対優先度を定義する。相対優先度を使用することにより、.NET Framework は、キャッシュされたアイテムの削除順序を決めることができます。低い優先度のアイテムは、高い優先度のアイテムよりも先にキャッシュから削除されます。
- Add メソッドを呼び出してアイテムを追加する。

キーと値によってアイテムを直接設定してキャッシュに追加するには

ディクショナリにアイテムを追加するときと同様に、項目のキーと値を指定してキャッシュにアイテムを追加します。

次のコード例は、CacheItem1 という名前のアイテムを Cache オブジェクトに追加します。

<C#>

```
Cache["CacheItem1"] = "Cached Item 1";
```

<Visual Basic>

```
Cache("CacheItem1") = "Cached Item 1"
```

Insert メソッドを使用してキャッシュにアイテムを追加するには

Insert メソッドを呼び出して、追加するアイテムのキーと値を渡します。

次のコード例は、CacheItem2 という名前の下に文字列を追加します。

<C#>

```
Cache.Insert("CacheItem2", "Cached Item 2");
```

<Visual Basic>

```
Cache.Insert("CacheItem2", "Cached Item 2")
```

依存関係を指定してキャッシュにアイテムを追加するには

Insert メソッドを呼び出して、CacheDependency オブジェクトのインスタンスを渡します。

次のコード例は、キャッシュ内の CacheItem2 という名前の別のアイテムに依存する CacheItem3 という名前のアイテムを追加します。

<C#>

```
string[] dependencies = { "CacheItem2" };
Cache.Insert("CacheItem3", "Cached Item 3",
    new System.Web.Caching.CacheDependency(null, dependencies));
```

<Visual Basic>

```
Dim dependencies As String() = {"CacheItem2"}
Cache.Insert("CacheItem3", "Cached Item 3", _
    New System.Web.Caching.CacheDependency( _
    Nothing, dependencies))
```

CacheItem4 という名前を持ち、XMLFile.xml という名前のファイルへのファイル依存関係を持つアイテムをキャッシュに追加する操作について、コード例を次に示します。

<C#>

```
Cache.Insert("CacheItem4", "Cached Item 4",
    new System.Web.Caching.CacheDependency(
    Server.MapPath("XMLFile.xml")));
```

<Visual Basic>

```
Cache.Insert("CacheItem4", "Cached Item 4", _
    New System.Web.Caching.CacheDependency( _
    Server.MapPath("XMLFile.xml")))
```

複数の依存関係を作成する方法について、コード例を次に示します。このコード例では、キャッシュ内の CacheItem1 という名前の別のアイテムへのキー依存関係を追加し、XMLFile.xml という名前のファイルへのファイル依存関係を追加します。

<C#>

```
System.Web.Caching.CacheDependency dep1 =
    new System.Web.Caching.CacheDependency(
                                Server.MapPath("XMLFile.xml"));
string[] keyDependencies2 = { "CacheItem1" };
System.Web.Caching.CacheDependency dep2 =
    new System.Web.Caching.CacheDependency(null, keyDependencies2);
System.Web.Caching.AggregateCacheDependency aggDep =
    new System.Web.Caching.AggregateCacheDependency();
aggDep.Add(dep1);
aggDep.Add(dep2);
Cache.Insert("CacheItem5", "Cached Item 5", aggDep);
```

<Visual Basic>

```
Dim dep1 As CacheDependency = _
    New CacheDependency(Server.MapPath("XMLFile.xml"))
Dim keyDependencies2 As String() = {"CacheItem1"}
Dim dep2 As CacheDependency = _
    New System.Web.Caching.CacheDependency(Nothing, _
    keyDependencies2)
Dim aggDep As AggregateCacheDependency = _
    New System.Web.Caching.AggregateCacheDependency()
aggDep.Add(dep1)
aggDep.Add(dep2)
Cache.Insert("CacheItem5", "Cached Item 5", aggDep)
```

有効期限ポリシーを付けてキャッシュにアイテムを追加するには

Insert メソッドを呼び出して、**絶対有効期限**または**スライド式有効期限**を渡します。

次のコード例は、1 分間の絶対有効期限を付けてキャッシュにアイテムを追加します。

<C#>

```
Cache.Insert("CacheItem6", "Cached Item 6",
    null, DateTime.Now.AddMinutes(1d),
```

```
    System.Web.Caching.Cache.NoSlidingExpiration);
```

<Visual Basic>

```
Cache.Insert("CacheItem6", "Cached Item 6", _
    Nothing, DateTime.Now.AddMinutes(1.0), _
    TimeSpan.Zero)
```

次のコード例は、10 分間のスライド式有効期限を付けてキャッシュに項目を追加します。

<C#>

```
Cache.Insert("CacheItem7", "Cached Item 7",
    null, System.Web.Caching.Cache.NoAbsoluteExpiration,
    new TimeSpan(0, 10, 0));
```

<Visual Basic>

```
Cache.Insert("CacheItem7", "Cached Item 7", _
    Nothing, System.Web.Caching.Cache.NoAbsoluteExpiration, _
    New TimeSpan(0, 10, 0))
```

優先度設定を付けてキャッシュに項目を追加するには

Insert メソッドを呼び出して、**CacheItemPriority** 列挙から値を指定します。

次のコード例は、High の優先値を付けてキャッシュに項目を追加します。

<C#>

```
Cache.Insert("CacheItem8", "Cached Item 8",
    null, System.Web.Caching.Cache.NoAbsoluteExpiration,
    System.Web.Caching.Cache.NoSlidingExpiration,
    System.Web.Caching.CacheItemPriority.High, null);
```

<Visual Basic>

```
Cache.Insert("CacheItem8", "Cached Item 8", _
    Nothing, System.Web.Caching.Cache.NoAbsoluteExpiration, _
    System.Web.Caching.Cache.NoSlidingExpiration, _
    System.Web.Caching.CacheItemPriority.High, _
    Nothing)
```

Add メソッドを使用してキャッシュにアイテムを追加するには

Add メソッドを呼び出します。このメソッドはアイテムを表すオブジェクトを返します。

次のコード例は、CacheItem9 という名前のアイテムをキャッシュに追加し、追加したアイテムを表すように変数 CachedItem9 に値を設定します。

<C#>

```
string CachedItem9 = (string)Cache.Add("CacheItem9",
    "Cached Item 9", null,
    System.Web.Caching.Cache.NoAbsoluteExpiration,
    System.Web.Caching.Cache.NoSlidingExpiration,
    System.Web.Caching.CacheItemPriority.Default,
    null);
```

<Visual Basic>

```
Dim CachedItem9 As String = CStr(Cache.Add("CacheItem9", _
    "Cached Item 9", Nothing, _
    System.Web.Caching.Cache.NoAbsoluteExpiration, _
    System.Web.Caching.Cache.NoSlidingExpiration, _
    System.Web.Caching.CacheItemPriority.Default, _
    Nothing))
```

ASP.NET のキャッシュの構成

ASP.NET には、ページ出力キャッシュおよびキャッシュ API を構成するための数多くのオプションが用意されています。

ページ出力キャッシュの構成

ページ出力キャッシュは、次の場所で構成できます。

- 構成ファイルページ出力キャッシュの構成は、コンピュータのすべての Web アプリケーションの設定を行う Machine.config ファイル、単一のアプリケーションの設定を行うアプリケーション固有の Web.config ファイルなど、アプリケーション構成階層の任意の構成ファイルの中で行うことができます。
- ページキャッシュのオプションは、個々のページで宣言またはプログラムを使用して設定できます。
- ユーザーコントロール キャッシュは、個々のユーザーコントロールで宣言またはプログラムを使用して設定できます。

Web.config のキャッシュの構成設定

Web.config ファイルのページ出力キャッシュには、**OutputCacheSection** と **OutputCacheSettingsSection** の 2 つのトップレベルの構成セクションがあります。

OutputCacheSection セクションは、ページ出力キャッシュの有効、無効など、アプリケーションスコープ設定を構成するために使用します。

OutputCacheSettingsSection は、個々のページで使用できるプロファイルおよび依存関係を構成するために使用します。

Machine.config のキャッシュの構成設定

Machine.config ファイルの構成セクションは、個々のアプリケーションがすべてのレベルでオーバーライドできないようにするために Machine.config ファイルの構成設定をロックできることを除いて、Web.config ファイルと同じです。

ページのキャッシュの構成設定

個々のページのキャッシュは、構成ファイルで定義されたキャッシュプロファイルを適用して構成できます。また、**@ OutputCache** ディレクティブで個々のキャッシュプロパティを構成するか、またはページのクラス定義に属性を設定して構成することもできます。

ユーザー コントロールのキャッシュの構成設定

ユーザーコントロール キャッシュは、ユーザーコントロール ファイルの **@ OutputCache** ディレクティブを設定するか、コントロールのクラス定義で **PartialCachingAttribute** 属性を設定することによって構成できます。

キャッシュ API の構成設定

アプリケーションの**キャッシュ API** は、Web.config ファイルで構成できます。ページ出力キャッシュと同様に、アプリケーションのホストは **Machine.config** ファイルで構成プロパティを設定し、キャッシュの構成設定をすべてのアプリケーションに対してロックできます。アプリケーションのキャッシュ API は、**CacheSection** で構成します。たとえば、次の構成要素によって項目の有効期限を無効にできます。

```
<cache disableExpiration="true" />
```

ページの複数バージョンのキャッシュ

ASP.NET では、出力キャッシュに同じページの複数のバージョンをキャッシュできます。出力キャッシュは、次の内容によって切り替えることができます。

- 初期要求 (HTTP GET) におけるクエリ文字列
- ポストバック (HTTP POST 値) 時に渡されるコントロール値
- 要求と共に渡される HTTP ヘッダー
- 要求元のブラウザーのメジャーバージョン番号
- ページ内のカスタム文字列

ページ出力の複数のバージョンを宣言によってキャッシュするには **@ OutputCache** ディレクティブの属性を使用し、プログラムによってキャッシュするには **HttpCachePolicy** クラスのプロパティとメソッドを使用します。

@ OutputCache ディレクティブには、ページ出力の複数のバージョンをキャッシュできる 4 つの属性があります。

- **VaryByParam** 属性を使用すると、キャッシュされる出力をクエリ文字列に基づいて切り替えることができます。
- **VaryByControl** 属性を使用すると、キャッシュされる出力をコントロールの値に基づいて切り替えることができます。
- **VaryByHeader** 属性を使用すると、キャッシュされる出力を要求の HTTP ヘッダーに基づいて切り替えることができます。
- **VaryByCustom** 属性を使用すると、キャッシュされる出力をブラウザーの種類または定義したカスタム文字列に基づいて切り替えることができます。

HttpCachePolicy クラスが提供する 2 つのプロパティと 1 つのメソッドを使用することにより、宣言で設定できるキャッシュ構成と同じ構成をプログラムで指定できます。VaryByParams プロパティと VaryByHeaders プロパティを使用すると、キャッシュポリシーを切り替えるためのクエリ文字列のパラメーターとヘッダー名をそれぞれ指定できます。SetVaryByCustom メソッドを使用すると、出力キャッシュを切り替える際の基準となるカスタム文字列を定義できます。

ASP.NET ページの一部だけのキャッシュ

要求のたびにページの一部のみを変更する必要があることが多い場合、ページ全体をキャッシュすることは効率的ではありません。このような場合は、ページの一部のみをキャッシュできます。それには、コントロールキャッシュとキャッシュ後置換の 2 つのオプションがあります。

コントロール キャッシュ

ユーザーコントロールを作成してコンテンツをキャッシュすることにより、データベース クエリのように作成に貴重なプロセッサ時間を消費するページの一部をページの他の部分と分離することができます。ページの構成要素のうち必要なサーバーリソースが少ない部分は、要求のたびに動的に生成できます。

キャッシュするページの各部分を特定し、その各部分を含むユーザーコントロールを作成したら、次はユーザーコントロールのキャッシュポリシーを決める必要があります。これらのポリシーを宣言によって設定するには、**@ OutputCache** ディレクティブを使用するか、ユーザーコントロールのコードで **PartialCachingAttribute** クラスを使用します。

たとえば、次のディレクティブをユーザーコントロール ファイル (.ascx ファイル) の先頭に含めると、コントロールの 1 つのバージョンは出力キャッシュに 120 秒間格納されます。

```
<%@ OutputCache Duration="120" VaryByParam="None" %>
```

コード内でキャッシュパラメーターを設定するには、ユーザーコントロールのクラス宣言内で属性を使用します。たとえば、次の属性をクラス宣言のメタデータに含めると、コンテンツの 1 つのバージョンは出力キャッシュに 120 秒間格納されます。

<C#>

```
[PartialCaching(120)]
public partial class CachedControl : System.Web.UI.UserControl
{
    // Class Code
}
```

<Visual Basic>

```
<PartialCaching(120)> _
Partial Class CachedControl
    Inherits System.Web.UI.UserControl
    ' Class Code
End Class
```

キャッシュされるユーザーコントロールのプログラムからの参照

キャッシュ可能なユーザーコントロールを宣言によって作成し、ID 属性を含めることで、そのユーザーコントロールのインスタンスをプログラムから参照できます。ただし、コードからユーザーコントロールを参照する前に、そのユーザーコントロールが出力キャッシュ内に存在するかどうかを確認する必要があります。キャッシュされるユーザーコントロールは、最初の要求に対してのみ動的に生成されます。引き続いて行われるすべての要求に対しては、指定された時間が切れるまで出力キャッシュが対応します。ユーザーコントロールがインスタンス化されていることを確認したら、そのコントロールを含むページからユーザーコントロールをプログラムで操作できます。たとえば、ユーザーコントロールに SampleUserControluser の ID を割り当てている場合は、次のコードでユーザーコントロールの存在を確認できます。

<C#>

```
protected void Page_Load(object sender, EventArgs e)
{
    if (SampleUserControl != null)
       // Place code manipulating SampleUserControl here.
}
```

<Visual Basic>

```
Protected Sub Page_Load(ByVal sender As Object, _
       ByVal e As System.EventArgs) Handles Me.Load
```

```
    If SampleUserControl <> Nothing Then
       ' Place code manipulating SampleUserControl here.
    End If
End Sub
```

ページとユーザー コントロールを異なる期間でキャッシュ

ページおよびそのページ内のユーザーコントロールに対して、それぞれ異なる出力キャッシュ期間を設定できます。ページの出力キャッシュ期間がユーザーコントロールの出力キャッシュ期間よりも長い場合は、ページの出力キャッシュ期間が優先されます。

ユーザーコントロールのキャッシュ期間よりもページのキャッシュ期間が長い場合の動作について、コード例を次に示します。ページは、100 秒間キャッシュされるように設定されています。

<C#>

```
<%@ Page language="C#" %>
<%@ Register tagprefix="SampleControl" tagname="Time"
    src="uc01.ascx" %>
<%@ OutputCache duration="100" varybyparam="none" %>
<SampleControl:Time runat="server" /><br /><br /><br />
    This page was most recently generated at:
  <p>
    <% DateTime t = DateTime.Now.ToString(); Response.Write(t); %>
  </p>
```

<Visual Basic>

```
<%@ Page language="VB" %>
<%@ Register tagprefix="SampleControl" tagname="Time"
    src="uc01.ascx" %>
<%@ OutputCache duration="100" varybyparam="none" %>
<SampleControl:Time runat="server" /><br /><br /><br />
    This page was most recently generated at:
  <p>
    <% Dim t As DateTime = DateTime.Now.ToString()
        Response.Write(t) %>
  </p>
```

ページに含まれるユーザーコントロールについて、コード例を次に示します。コントロールのキャッシュ期間は 50 秒に設定されます。

<C#>

```
<% @Control language="C#" %>
<% @OutputCache duration="50" varybyparam="none" %>
This user control was most recently generated at:
<p>
<% DateTime t = DateTime.Now.ToString();
    Response.Write(t); %>
</p>
```

<Visual Basic>

```
<% @Control language="VB" %>
<% @OutputCache duration="50" varybyparam="none" %>
This user control was most recently generated at:
<p>
<% Dim t As DateTime = DateTime.Now.ToString()
Response.Write(t) %>
</p>
```

ページの出力キャッシュ期間がユーザーコントロールよりも短い場合、ユーザーコントロール以外の部分が要求によって再生成された後でも、ユーザーコントロールはその出力キャッシュ期限が切れるまでキャッシュされています。

キャッシュ期間がページよりも長いユーザーコントロールを含むページのマークアップについて、コード例を次に示します。ページは、50 秒間キャッシュされるように設定されています。

<C#>

```
<%@ Page language="C#" %>
<%@ Register tagprefix="SampleControl" tagname="Time" src="uc2.ascx" %>
<%@ OutputCache duration="50" varybyparam="none" %>
<SampleControl:Time runat="server" /><br /><br /><br />
This page was most recently generated at:
<p>
 <% DateTime t = DateTime.Now.ToString();
    Response.Write(t); %>
</p>
```

<Visual Basic>

```
<%@ Page language="VB" %>
<%@ Register tagprefix="SampleControl" tagname="Time" src="Uc2.ascx" %>
```

```
<%@ OutputCache duration="50" varybyparam="none" %>
<SampleControl:Time runat="server" /><br /><br /><br />
This page was most recently generated at:
<p>
 <% Dim t As DateTime = DateTime.Now.ToString()
Response.Write(t) %>
</p>
```

ページに含まれるユーザーコントロールについて、コードを次に示します。コントロールのキャッシュ期間は 100 秒に設定されます。

<C#>

```
<% @Control language="C#" %>
<% @OutputCache duration="100" varybyparam="none" %>
This user control was most recently generated at:
<p>
<% DateTime t = DateTime.Now.ToString();
    Response.Write(t); %>
</p>
```

<Visual Basic>

```
<% @Control language="VB" %>
<% @OutputCache duration="100" varybyparam="none" %>

This user control was most recently generated at:
<p>
<% Dim t As DateTime = DateTime.Now.ToString()
Response.Write(t) %>
</p>
```

ASP.NET と Web フォームの概要

ASP.NET は、Web アプリケーション開発のための次世代のテクノロジです。Active Server Pages (ASP) の優れた点と、共通言語ランタイム (CLR) が提供するリッチなサービスおよび機能が組み合わされ、さらにその他のさまざまな新機能が追加されています。その結果として、わずかなコーディングで高い柔軟性が得られる、堅牢でスケーラブルかつ高速な Web 開発エクスペリエンスが実現されています。

Web フォームは ASP.NET の中核に位置し、Web アプリケーションの外観を与えるユーザーインターフェイス (UI) 要素です。Web フォームは、その中に配置されたコントロールのプロパティ、メソッド、およびイベントを提供するという点で Windows フォームに似ています。ただし、これらの UI 要素は、要求の中で指定された HTML などの適切なマークアップ言語を使って自分自身をレンダリングします。Visual Studio や Visual Web Developer Express を使用する場合には、従来から Web アプリケーションの UI の作成に使用されてきたドラッグアンドドロップインターフェイスも使用できます。

Web フォームは、ビジュアルな部分 (.aspx ファイル) と、別のクラスファイルに置かれるフォームの分離コードの 2 つの要素から構成されています。

Web フォームの目的

Web フォームと ASP.NET は、ASP のいくつかの制限を克服するために作成されたものです。新しい利点としては、次のようなものがあります。

- HTML インターフェイスのアプリケーションロジックからの分離。
- ブラウザーを検出し、HTML などの適切なマークアップ言語を送信することができるリッチなサーバー サイド コントロールのセットである。
- 新しいサーバー サイド .NET コントロールのデータバインディング機能により、書かなくてはならないコードの量が減っている。
- Visual Basic プログラマがよく慣れているイベントベースのプログラミングモデルである。
- VBScript または JScript をインタプリタ形式で処理していた ASP とは異なり、複数の言語をサポートしコンパイル済みのコードを使用する。
- サードパーティが追加の機能を提供するコントロールを作成できる。

表面上、Web フォームはコントロールを描画するためのワークスペースのように見えますが、実際にはそれよりもはるかに高度な機能が備わっています。しかし通常は、各種のコントロールを Web フォーム上に配置するだけで UI を作成することができます。使用するコントロールによって、各コントロールがどのプロパティ、イベント、およびメソッドを受け取るかが決定されます。ユーザーインターフェイスの作成に使用できるコントロールには、HTML コントロールと Web フォーム コントロールの 2 つのタイプがあります。

以下に、Web フォームと ASP.NET フレームワークで使用できるコントロールのタイプを示します。

HTML サーバー コントロール

HTML サーバー コントロールは、FrontPage やその他の HTML エディターを使って UI を描画したときに生成される実際の HTML 要素に似ています。Web フォームでは標準の HTML 要素も使用することができます。たとえば、テキストボックスを作成したい場合には、次のようにします。

```
<input type="text" id="txtFirstName" size="25" />
```

Visual Studio を使用している場合には、[ツールボックス] の [HTML] タブから **Input (Text)** コントロールを選択し、HTML ページ上の目的の位置にコントロールを描画します。

HTML 要素は、タグに "runat=server" を追加することで、Web フォームがサーバー上で処理されるときにのみ HTML サーバーコントロールとして動作させることができます。

```
<input type="text" id="txtFirstName" size="25" runat="server" />
```

HTML サーバーコントロールでは、タグに関連付けられたサーバーイベントを処理し (ボタンクリックなど)、Web フォームのコードの中でプログラム的に HTML タグを操作することができます。

コントロールがブラウザーに対してレンダリングされるときには、そのタグは Web フォームの内容から runat="server" を削除したものとしてレンダリングされます。この機能を使って、ブラウザーに送信される HTML を細かく制御することができます。

下の表は ASP.NET で利用できる HTML サーバーコントロールとコードの例です。

表. ASP.NET で利用できる HTML サーバーコントロールとコードの例

コントロール	説明	Web フォームのコード例
ボタン	クリックイベントに反応するために使用できる通常のボタン	<input type="button" runat="server" />
リセット ボタン	フォーム上の他のすべての HTML フォーム要素をデフォルト値にリセットする	<input type="reset" runat="server" />
送信 ボタン	フォームデータを、FORM タグの Action= 属性で指定されたページに自動的に POST する	<input type="submit" runat="server" />
テキスト フィールド	HTML フォーム上にユーザーのための入力領域を提供する	<input type="text" runat="server" />
テキスト エリア	HTML フォーム上での複数行の入力に使用される	<input type="textarea" runat="server" />

ファイル フィールド	テキストフィールドと [Browse] ボタンをフォーム上に配置し、[Browse] ボタンがクリックされたときに、ユーザーがローカルマシンからファイル名を選択できるようにする	<input type="file" runat="server" />
パスワード フィールド	HTML フォーム上の入力領域だが、このフィールドに入力された文字はアスタリスクとして表示される	<input type="password" runat="server" />
チェックボックス	ユーザーが選択またはクリアできるチェックボタンを提供する	<input type="checkbox" runat="server" />
ラジオ ボタン	フォーム上に複数配置され、ユーザーがいずれかの 1 つのコントロールを選択できるようにする	<input type="radio" runat="server" />
テーブル	情報を表形式で提示することができる	<table runat="server"></table>
イメージ	HTML フォーム上にイメージを表示する	
リスト ボックス	ユーザーに対して項目のリストを表示する。2 以上の値のサイズを設定して、表示する項目の数を指定することができる。この制限よりも多くの項目が存在する場合には、コントロールに自動的にスクロールバーが追加される	<select size="2" runat="server" ></select>
ドロップダウン	ユーザーに対して項目のリストを表示する。ただし、表示されるのは一度に 1 項目のみである。ユーザーは、このコントロールの端にある下向き矢印をクリックして、項目のリストを表示することができる	<select><option></option></select>
水平線	HTML ページに水平の線を表示する	<hr />

これらすべてのコントロールは、Web フォームに標準の HTML を書き込みます。オプションとして各コントロールに ID 属性を割り当て、このタイプのコントロールに共通するイベントを処理するクライアントサイドの JavaScript コードを作成することができます。

以下に、一般的なクライアント サイド イベントの例を下の表に示します。

表. 一般的なクライアント サイド イベントの例

イベント	説明
onblur	コントロールがフォーカスを失った
onchange	コントロールの内容が変化した
onclick	コントロールがクリックされた
onfocus	コントロールがフォーカスを受け取った
onmouseover	マウスがこのコントロールの上に置かれた

Web サーバー コントロール

Web サーバー コントロールは、HTML サーバーコントロールと同じようにサーバー上で作成され、実行されます。設計時に指定された操作を実行した後に、適切な HTML をレンダリングし、その HTML を出力ストリームに送信します。たとえば、DropDownList コントロールを使うとデータソースにバインドすることができますが、レンダリングされる出力は、ブラウザーに送信される場合には標準的な <SELECT> および <OPTION> タグとなります。これと同じ DropDownList コントロールが、ターゲットが携帯電話であるときには WML をレンダリングすることができます。これは、これらのコントロールが特定の 1 つのマークアップ言語にマッピングされているのではなく、適切なマークアップ言語をターゲットにできる柔軟性を備えているためです。

すべての Web サーバー コントロールは **System.Web.UI.WebControls** という共通の基本クラスを継承します。この基本クラスは、これらすべてのコントロールが持つことになる共通プロパティのセットをインプリメントしています。以下に、これらの共通プロパティの例を示します。

- BackColor
- Enabled
- Font-Size
- ForeColor
- TabIndex
- Visible
- Width

これ以外にも、Microsoft .NET Framework が提供するコントロールのカテゴリが数種類存在します。一部のコントロールは、対応する HTML とほぼ 1 対 1 の関係を持っています。一部のコントロールはサーバーにポストバックされるときに追加情報を提供し、一部のコントロールはデータをテーブルやリストタイプの形式で表示します。

下の表に、Web サーバー コントロールと、各コントロールで反応することができるサーバー サイドイベントのリストを示します。

表 . *ASP.NET と Web サーバー コントロール*

コントロール	説明	よく使われるサーバー サイド イベント	Web フォームのコード例
Label	テキストを HTML ページ上に表示する	なし	<asp:Label id="Label1" runat="server"> Label</asp:Label>
TextBox	ユーザーに HTML フォーム上の入力領域を提供する	TextChanged	<asp:TextBox id="TextBox1" runat="server"> </asp:TextBox>
Button	サーバー上のクリック イベントに応答するために使用される通常のボタン。CommandName および CommandArguments プロパティを設定することで、追加の情報を渡すことができる	Click, Command	<asp:Button id="Button1" runat="server" Text="Button"> </asp:Button>
LinkButton	サーバーへのポストバックを行うという点ではボタンに似ているが、ハイパーリンクのような外見をしている	Click, Command	<asp:LinkButton id="LinkButton1" runat="server"> LinkButton</asp:LinkButton>
ImageButton	グラフィカル イメージを表示することができ、クリックされると、クリック時のイメージ内でのマウス座標などのコマンド情報をサー	Click	<asp:ImageButton id="ImageButton1" runat="server"> </asp:ImageButton>

	バーにポスト バックする		
Hyperlink	クリック イベントに応答する通常のハイパーリンク コントロール	なし	<asp:HyperLink id="HyperLink1" runat="server"> HyperLink</asp:HyperLink>
DropDownList	HTML コントロールと似た通常のドロップダウン リスト コントロールだが、データ ソースにバインドすることができる	SelectedIndexChanged	<asp:DropDownList id="DropDownList1" runat="server"> </asp:DropDownList>
ListBox	HTML コントロールと似た通常の ListBox コントロールだが、データ ソースにバインドすることができる	SelectedIndexChanged	<asp:ListBox id="ListBox1" runat="server"> </asp:ListBox>
DataGrid	<TABLE> の強化版。データ ソースをこのコントロールにバインドすると、すべての列情報が表示される。また、このコントロールを使うと、改ページ、並べ替え、およびフォーマットをきわめて簡単に行える	CancelCommand, EditCommand, DeleteCommand, ItemCommand, SelectedIndexChanged, PageIndexChanged, SortCommand, UpdateCommand, ItemCreated, ItemDataBound	<asp:DataGrid id="DataGrid1" runat="server"> </asp:DataGrid>
DataList	テーブル以外のタイプのデータ形式を作成することができる。データをテンプレート項目にバインドすると、HTML の断片が一定の繰り返し形式に従って結合される	CancelCommand, EditCommand, DeleteCommand, ItemCommand, SelectedIndexChanged, UpdateCommand, ItemCreated, ItemDataBound	<asp:DataList id="DataList1" runat="server"> </asp:DataList>
Repeater	テーブル以外のタイプのデータ形式を作成することができる。デー	ItemCommand, ItemCreated, ItemDataBound	<asp:Repeater id="Repeater1" runat="server">

	タをテンプレート項目にバインドすると、HTML の断片が一定の繰り返し形式に従って結合される		</asp:Repeater>
CheckBox	ユーザーがチェックまたはアンチェックできるチェック ボックスを表示する、通常の HTML コントロールによく似たコントロール	CheckChanged	<asp:CheckBox id="CheckBox1" runat="server"> </asp:CheckBox>
CheckBoxList	連係して動作するチェック ボックスのグループを表示する	SelectedIndexChanged	<asp:CheckBoxList id="CheckBoxList1" runat="server"> </asp:CheckBoxList>
RadioButton	ユーザーがチェックまたはアンチェックできるボタンを表示する、通常の HTML コントロールによく似たコントロール	CheckChanged	<asp:RadioButton id="RadioButton1" runat="server"> </asp:RadioButton>
RadioButtonList	連係して動作するラジオ ボタンのグループを表示する	SelectedIndexChanged	<asp:RadioButtonList id="RadioButtonList1" runat="server"> </asp:RadioButtonList>
Image	ページ内にイメージを表示する、通常の HTML コントロールによく似たコントロール	なし	<asp:Image id="Image1" runat="server"> </asp:Image>
Panel	他のコントロールのグループ化に使用される	なし	<asp:Panel id="Panel1" runat="server"> Panel</asp:Panel>
PlaceHolder	他のサーバー サイド コントロールを実行時に動的に追加できる場所として機能する	なし	<asp:PlaceHolder id="PlaceHolder1" runat="server"> </asp:PlaceHolder>
Calendar	カレンダーの HTML	SelectionChanged,	<asp:Calendar

	バージョンを作成する。デフォルトの日付を設定したり、カレンダーの中で前方または後方に移動するといった操作を行うことができる	VisibleMonthChanged, DayRender	id="Calendar1" runat="server"> </asp:Calendar>
AdRotator	表示する広告のリストを指定することができる。ユーザーがページを再表示するたびに、一連の広告が順番に表示される	AdCreated	<asp:AdRotator id="AdRotator1" runat="server"> </asp:AdRotator>
Table	通常の HTML コントロールによく似たコントロール	なし	<asp:Table id="Table1" runat="server"> </asp:Table>
XML	HTML 内に XML ドキュメントを表示するために使用される。また、XML を表示する前に XSLT 変換を実行するためにも使用できる	なし	<asp:Xml id="Xml1" runat="server"> </asp:Xml>
Literal	リテラルを表示するという点でラベルに似ているが、実行時に新しいリテラルを作成し、それらをコントロールに格納することができる	なし	<asp:Literal id="Literal1" runat="server"> </asp:Literal>

フィールド検証コントロール

もう 1 つのタイプの Web フォーム コントロールは、ユーザーがバックエンドサーバーにデータを送信する前に、クライアント上でデータの**妥当性確認**を行います。

ユーザーが HTML ページ上のコントロールのグループに何らかのデータを入力したとき、通常は ASP または ASP.NET のコードで妥当性を確認できるように、すべてのデータをサーバーに返送しなくてはなりません。値が入力されたかどうかを確認する目的に、この余分なラウンドトリップを行

う代わりに、フィールド検証コントロールを使ってこのクライアント チェックを実行することができます。これらのコントロールは HTML ページにクライアント サイドの JavaScript コードを書き込み、値をラウンドトリップせずにチェックします。JavaScript はほとんどのブラウザー上で動作します。

フィールド検証コントロールを下の表にまとめます。これらのフィールド検証コントロールは、サーバーへラウンドトリップせずにユーザーの入力をチェックするための便利なツールです。

表. フィールド検証コントロール

コントロール	説明	HTML または JavaScript コードの例
RequiredFieldValidator	フォーム上のコントロールをチェックして、何らかの入力が行われているかどうかを確認する。入力がなければ、設定された ErrorMessage がこのコントロールに表示される。	<asp:RequiredFieldValidator id="RequiredFieldValidator1" runat="server" ErrorMessage="RequiredFieldValidator"></asp:RequiredFieldValidator>
CompareValidator	1 つのコントロールの内容を、フォーム上の別のコントロールの内容と比較して、両者が一致しているかどうかをチェックする。一致していなければ、設定された ErrorMessage がこのコントロールに表示される。	<asp:CompareValidator id="CompareValidator1" runat="server" ErrorMessage="CompareValidator"> </asp:CompareValidato>
RangeValidator	コントロールに入力された値が、指定された範囲内に収まっているかどうかをチェックする。収まっていなければ、設定された ErrorMessage がこのコントロールに表示される。	<asp:RangeValidator id="RangeValidator1" runat="server" ErrorMessage="RangeValidator"> </asp:RangeValidator>
RegularExpressionVali	コントロールの内容が、	<asp:RegularExpressionValidator

dator	定義された定型入力 (正規表現) と一致するかどうかをチェックする。一致していなければ、設定された ErrorMessage がこのコントロールに表示される。	id="RegularExpressionValidator1" runat="server" ErrorMessage="RegularExpressionValidator"> </asp:RegularExpressionValidator>
CustomValidator	特 定のコントロールの内容の妥当性を確認するために使用する、サーバー サイドまたはクライアント サイドのスクリプト関数を指定することができる。これらの関数からは True または False の値を返さなくてはならない。True の値が返された場合には、処理が続行される。False の値が返された場合には、このコントロールに対して指定された ErrorMessage が表示される。	<asp:CustomValidator id="CustomValidator1" runat="server" ErrorMessage="CustomValidator"> </asp:CustomValidator>
ValidationSummary	このフォーム上の他のすべてのバリデータ コントロールから、すべての ErrorMessage プロパティを自動的に収集し、それらを番号付きリスト、箇条書きリスト、または段落の形式で表示する。	<asp:ValidationSummary id="ValidationSummary1" runat="server"></asp:ValidationSummary>

カスタム コントロールの作成

以上の組み込みコントロールに加えて、独自のカスタム コントロールを作成することができます。たとえば、個々のメニュー項目がデータベースをもとにして構築されるメニューシステムを作成することができます。

Web フォームの動作

Windows フォームと同様に、Web フォームが初期化されロードされるときには、いくつかのイベントが一定の順序で発生します。また、ブラウザー内にレンダリングされたページとユーザーの相互作用に反応して発生するイベントもあります。標準的な ASP や HTML が作成され、ブラウザーに送信されるプロセスを考えると、すべてが直線的なトップダウンの形式で処理されると思うかもしれません。しかし、Web フォームではまったく違った形の処理が行われます。

Web フォームは Windows フォームと同様に、標準的なロード、描画 (レンダリング) およびアンロードのタイプのイベントを順番に発生します。このプロセスの中で、クラスモジュールの中のそれぞれ異なるプロシージャが呼び出されます。クライアントブラウザーからページが要求されると、.aspx ページの中のタグとページコードの中のタグの両方がロードされ処理されます。

最初に、**Init** イベントが、.aspx ファイルのタグの記述に従ってページを初期状態に設定します。ページが自分自身にポストバックする場合、Init は **"viewstate"** に格納されていたページ状態の復元も行います。このときコードは **Page_Init()** イベントを処理して、ページの初期状態をさらにカスタマイズすることもできます。

次に、**Load** イベントが発生します。Load イベントを使うと、このページが初めてロードされたのか、またはユーザーがページ上のボタンやその他のコントロールをクリックしたために起こったポストバックなのかを判断することができます。データのコントロールへのバインディングなどの一部の初期化操作は、最初のページロードでのみ行います。

その次に、ページがポストバックされていた場合に限り、コントロールイベントが発生します。最初に **"change"** イベントが発生します。これらのイベントはブラウザー内で蓄積され、ページがサーバーに送信されたときにのみ実行されます。例としては、テキストボックス内のテキストの変更や、リストの選択などがあります。

続いて、ページがポストバックされていた場合に限り、ページのポストバックを引き起こしたコントロールイベントが発生します。ポストバックイベントの例としては、ボタンのクリックやチェックボックスの **CheckedChanged** イベントのような **"autopostback"** 変更イベントがあります。

次に、ページがブラウザーに対してレンダリングされます。ページがポストバックを通して再び呼び出されたときに、ASP.NET がページを前の状態に戻せるようにページ内の隠されたフィールドにいくつかの状態情報 ("viewstate") が格納されます。

ページの破棄の前にコードが処理できる最後のページイベントが **Page_Unload()** です。ページはすでにレンダリングされているので、このイベントは一般にクリーンアップとログ記録のタスクの実行にのみ使用されます。最後に、実行中のページを表すクラスが破棄され、ページはサーバーメモリからアンロードされます。

.aspx ページまたはそのコードを変更した場合、そのページを表す動的に生成される DLL は、次にページが要求されたときに再生成されます。

Global.asax

Global.asax ファイルは ASP の Global.asa ファイルに似ていますが、ASP.NET には多数のイベントが追加されています。また、Global.asax ファイルは ASP のようにインタプリタ形式で解釈されるのではなく、コンパイルされています。Web サイト上で特定のイベントが発生したときに、Global.asax ファイル内でイベントプロシージャが発生するのはいままでと同じです。

下の表は、Global.asax ファイル内の利用可能なイベント プロシージャを示しています。

表. *Global.asax* 内の利用可能なイベントプロシージャ

イベント プロシージャ	説明
Application_Start	最初のユーザーが Web サイトを訪れたときに発生する。
Application_End	サイトのセッションの最後のユーザーがタイムアウトを起こしたときに発生する。
Application_Error	アプリケーション内で処理されないエラーが発生したときに発生する。
Session_Start	新しいユーザーが Web サイトを訪れたときに発生する。
Session_End	ユーザーのセッションがタイムアウトを起こすか終了したときに発生する。
Application_AcquireRequestState	ASP.NET が、現在の要求に関連付けられた現在の状態 (セッション状態など) を取得したときに発生する。
Application_AuthenticateRequest	セキュリティモジュールがユーザーのアイデンティティを確立したときに発生する。
Application_AuthorizeRequest	セキュリティモジュールがユーザー承認を確認したときに発生する。
Application_BeginRequest	ASP.NET が要求の処理を開始したときに、要求関連の他のイベントよりも前に発生する。

Application_Disposed	ASP.NET が、要求への応答の際に、実行のチェーンを完了したときに発生する。
Application_EndRequest	要求の処理中に、要求関連の他のイベントの後に、最後のイベントとして発生する。
Application_PostRequestHandlerExecute	ASP.NET ハンドラ (ページ、XML Web サービス) が実行を終了した直後に発生する。
Application_PreRequestHandlerExecute	ASP.NET が、ページや XML Web サービスなどのハンドラを実行する直前に発生する。
Application_PreSendRequestContent	ASP.NET がクライアントにコンテンツを送信する直前に発生する。
Application_PreSendRequestHeaders	ASP.NET がクライアントに HTTP ヘッダーを送信する直前に発生する。
Application_ReleaseRequestState	ASP.NET がすべての要求ハンドラの実行を終了した後に発生する。このイベントを受けて、状態モジュールは現在の状態データを保存する。
Application_ResolveRequestCache	ASP.NET が、承認イベントを完了した後に発生する。これを受けて、キャッシング モジュールは、ハンドラ (ページ、Web サービスなど) の実行をバイパスし、要求への応答にキャッシュを使用するようになる。
Application_UpdateRequestCache	ASP.NET がハンドラの実行を終了した後に発生する。これを受けて、キャッシング モジュールは、それ以降のキャッシュからの応答に使用される応答を格納する。

Web フォームの作成

ユーザーがファーストネームとラストネームを入力するための Web フォームを作成してみます。ユーザーが Web ページの 2 つのテキストフィールドにデータを入力した後に [Login] ボタンをクリックすると、[Login] ボタンの下のラベルにそのデータが「ラスト ネーム、ファースト ネーム」の形式で表示されるようにします。

下の図は、ここで使用するサンプルのログイン Web フォームを示しています。

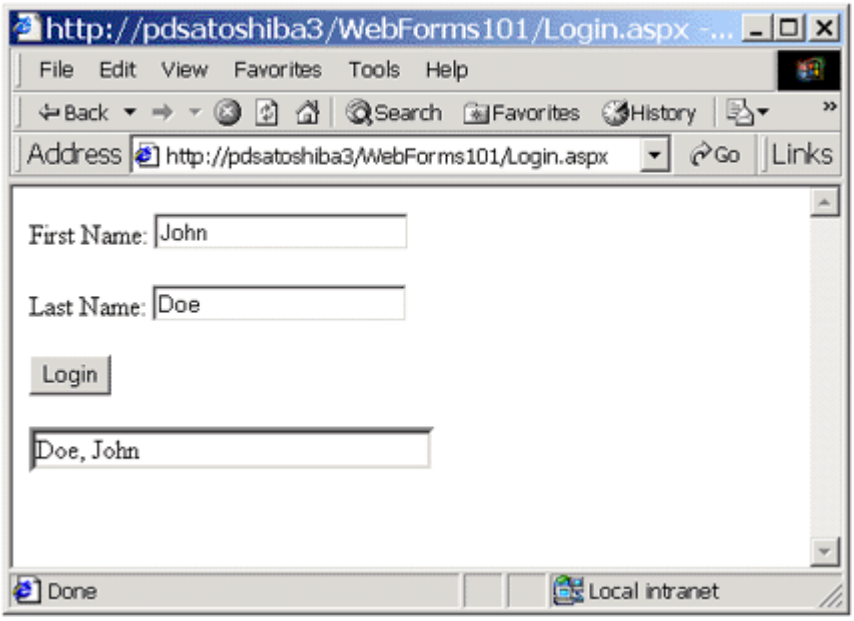

図. 入力されたデータをラベル コントロールに表示するサンプル

ログイン フォームを作成するためのステップ

最初に、新しい Web アプリケーション プロジェクトを作成します。

1. Visual Studio を起動し、[ファイル] – [新規作成] - [プロジェクト] をクリックします。
2. [新しいプロジェクト] ダイアログで [ASP.NET Web アプリケーション] クリックします。
3. このプロジェクトの名前を入力します（例：WebApplication1）。
4. [OK] をクリックして、新しい Web アプリケーション プロジェクトを作成します。
5. プロジェクトに既定で作成される Default.aspx を開きます。
6. ツールボックスから適切なコントロールを追加し、これらのコントロールのプロパティを下の従って設定することで、上記の図のようなフォームを作成します。

表. ログイン フォームの作成に使用するコントロール

コントロール タイプ	プロパティ	値
Label	Name	Label1
	Text	First Name
TextBox	Name	txtFirst
	Text	
Label	Name	Label2
	Text	Last Name
TextBox	Name	txtLast
	Text	
Button	Name	btnSubmit
	Text	Login
Label	Name	lblName
	BorderStyle	Insert
	Text	

この時点で、F5 キーを押してアプリケーションを実行することで、Web フォームをブラウザー内で表示することができます。このページにはまだ何の機能も含まれていませんが、現時点ですべてが正しく処理されていることを確認する良い機会となります。

ブラウザー内に Web フォームが表示され、2 つのテキストフィールドにデータを入力できるようになります。この時点では、[Login] ボタンをクリックしても何の処理も行われません。

次に、この [Login] ボタンに何らかの処理を行わせる方法を説明します。

ボタンへのコードの追加

ボタンにコードを追加して、このボタンがテキストボックスに入力されたデータをポストし、ボタンの下のラベルに適切なデータを格納するようにします。

1. Default.aspx ページをデザインモードで表示して Login ボタン コントロールをダブルクリックします。コード ウィンドウにイベント プロシージャ btnSubmit が表示されるはずです。カーソル位置でクリックを行います。
2. Click イベント プロシージャに次のコードを入力します。

<Visual Basic>

```
Protected Sub btnSubmit_Click(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles Button1.Click
    lblName.Text = txtLast.Text & ", " & txtFirst.Text
```

```
End Sub
```

<C#>

```
protected void Button1_Click(object sender, EventArgs e)
{
  lblName.Text = txtLast.Text + ", " + txtFirst.Text;
}
```

ここでは、txtLast と txtFirst のテキストボックスから Text プロパティの値を取得し、そのデータを [Login] ボタンの下のラベル コントロールに格納しています。Visual Basic プログラミングの経験がある人は、簡単に理解できるでしょう。実際、.NET の大きな利点は、Microsoft Windows アプリケーションと Web アプリケーションの両方で同じプログラミングモデルが使用できるという点にあります。

ASP.NET MVC の概要

Model-View-Controller (MVC) のアーキテクチャパターンでは、アプリケーションをモデル、ビュー、およびコントローラーの 3 つの主要コンポーネントに分離します。ASP.NET MVC フレームワークは、マスターページやメンバーシップベースの認証などの既存の ASP.NET 機能と統合できる、軽量で高度なテストが可能な Web アプリケーションフレームワークです。MVC フレームワークは、**System.Web.Mvc アセンブリ**で定義されます。

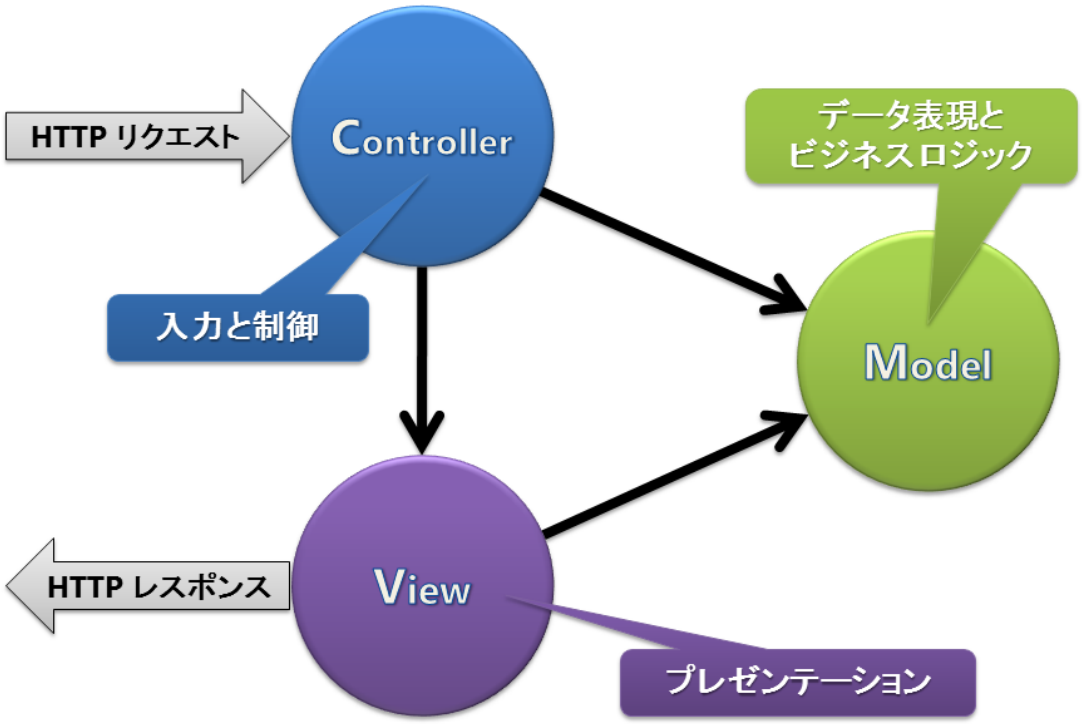

図. MVC 設計パターン

Web アプリケーションは、その種類により、MVC フレームワークを使用することでメリットが得られる場合があります。また、今までどおり、Web フォームとポストバックに基づく従来の ASP.NET アプリケーションパターンを使用することもできます。MVC フレームワークには、次のコンポーネントが含まれます。

- **モデル** —— モデル オブジェクトは、アプリケーションの中でデータドメインのロジックを実装する部分です。多くの場合、モデルオブジェクトでは、モデルの状態の取得と格納にデータベースを使用します。
- **ビュー** —— ビューは、アプリケーションのユーザーインターフェイス (UI) を表示するコンポーネントです。通常、この UI はモデル データから作成されます。

- **コントローラー** —— コントローラーは、ユーザーインタラクションを処理し、モデルを操作し、最後に、レンダリングするビューを選択して UI を表示するコンポーネントです。MVC アプリケーションでは、ビューには情報のみが表示されます。ユーザーの入力や操作を処理して対応するのは、コントローラーの役目です。

MVC パターンを使用すると、アプリケーションのさまざまな側面 (入力ロジック、ビジネスロジック、および UI ロジック) が分離されたアプリケーションを作成することができ、同時にこれらの要素間で疎結合を保つことができます。このパターンでは、各ロジックがアプリケーション内で配置される場所が指定されます。UI ロジックは、ビューに属します。入力ロジックは、コントローラーに属します。ビジネスロジックは、モデルに属します。このように分離されていることで、一度に実装の 1 つの側面に集中できるため、アプリケーション構築時の複雑さの管理が容易になります。たとえば、ビジネスロジックに依存することなく、ビューに集中することができます。

テスト駆動開発のサポート

MVC パターンでは、複雑さに対処できるだけでなく、Web フォームベースの ASP.NET Web アプリケーションをテストするよりも簡単にアプリケーションをテストできます。たとえば、Web フォームベースの ASP.NET Web アプリケーションでは、1 つのクラスが出力の表示とユーザー入力への応答の両方で使用されます。Web フォームベースの ASP.NET アプリケーション向けに、自動化されたテストを記述することは複雑になります。これは、個々のページをテストするためにアプリケーション内のページクラス、すべての子コントロール、および追加の依存クラスをインスタンス化する必要があるからです。ページを実行するために非常に多くのクラスがインスタンス化されるため、アプリケーションの個々のページだけを対象としたテストを記述することは困難な場合があります。したがって、Web フォームベースの ASP.NET アプリケーション向けにテストを実装することは、MVC アプリケーションのテストを実装するよりも難しくなります。さらに、Web フォームベースの ASP.NET アプリケーションのテストには、Web サーバーが必要です。MVC フレームワークでは、コンポーネントを分離し、インターフェイスを多用します。これにより、コンポーネントをフレームワークの他の部分から分離して、個別にテストすることが可能になります。

MVC アプリケーションを作成する場合

Web アプリケーションを実装するときに、ASP.NET MVC フレームワークを使用するか ASP.NET Web フォーム モデルを使用するかについては、慎重に検討する必要があります。MVC フレームワークは、Web フォーム モデルを置き換えるものではありません。Web アプリケーションに対しては、どちらのフレームワークも使用できます。

MVC ベースの Web アプリケーションの利点

ASP.NET MVC フレームワークには、次の利点があります。

- アプリケーションをモデル、ビュー、およびコントローラーに分割することにより、複雑さに対処しやすくなります。
- ビュー状態やサーバーベースのフォームは使用されません。このため、アプリケーションの動作を完全に制御しようとする開発者には、MVC フレームワークが適しています。
- 単一のコントローラーによって Web アプリケーション要求を処理するフロントコントローラー パターンを使用します。これにより、豊富なルーティングインフラストラクチャをサポートするアプリケーションを設計できます。
- テスト駆動開発 (TDD) のサポートが強化されています。
- 大規模な開発者チームによってサポートされる Web アプリケーション、およびアプリケーションの動作の高度な制御が必要な Web デザイナーに適しています。

Web フォーム ベースの Web アプリケーションの利点

Web フォームベースのフレームワークには、次の利点があります。

- HTTP を介して状態を維持するイベントモデルをサポートしています。そのため、基幹業務の Web アプリケーション開発に適しています。
- ページ コントローラー パターンを使用して個々のページに機能を追加します。
- サーバーベースのフォームでビュー状態を使用します。これにより、状態情報の管理が容易になります。
- 大量のコンポーネントを活用してアプリケーション開発を迅速に行う小規模な開発者チームまたはデザイナーのチームに適しています。
- 一般的に、コンポーネント (Page クラス、コントロールなど) 間は密接に統合されており、必要なコードは MVC モデルよりも少ないため、アプリケーション開発の複雑さが軽減されます。

ASP.NET MVC フレームワークの機能

ASP.NET MVC フレームワークでは、次のコンポーネントが提供されます。

- **アプリケーション作業 (入力ロジック、ビジネス ロジック、UI ロジック) の分離、テスト容易性、およびテスト駆動開発 (TDD)** —— MVC フレームワーク内のすべての中心的な規約は、インターフェイスを基準にしたものであり、モック オブジェクトを使用してテストできます。モック オブジェクトは、アプリケーションの実際のオブジェクトの動作を模倣するシミュレートオブジェクトです。
- **拡張可能および接続可能なフレームワーク** —— ASP.NET MVC フレームワークのコンポーネントは、簡単に置き換えたりカスタマイズしたりできるように設計されています。独自のビューエンジン、URL ルーティングポリシー、アクションメソッド パラメーターのシリアル化などのコンポーネントを組み込むことができます。ASP.NET MVC フレームワークは、依存関係の挿入 (DI) と制御の反転 (IOC) コンテナーモデルの使用もサポートします。DI を使用することで、オブジェクトそのものを作成するためにクラスに依存するのではなく、クラスにオブジェクトを

挿入できます。IOC は、オブジェクトが別のオブジェクトを必要とする場合に、最初のオブジェクトが構成ファイルなどの外部ソースから 2 番目のオブジェクトを取得するように指定します。

- **ASP.NET ルーティングの広範囲なサポート** —— ASP.NET ルーティングとは、検索可能でわかりやすい URL を持つアプリケーションを構築できる強力な URL マッピング コンポーネントです。URL にファイル名拡張子を含める必要はありません。URL は、検索エンジン最適化 (SEO) および REST (Representational State Transfer) アドレッシングに最適な URL 名前付けパターンをサポートします。
- **既存の ASP.NET ページ (.aspx ファイル)、ユーザーコントロール (.ascx ファイル)、およびマスターページ (.master ファイル) マークアップファイル内のマークアップをビューテンプレートとして使用するためのサポート** —— ASP.NET MVC フレームワークでは、入れ子になったマスターページ、インライン式 (<%= %>)、宣言型サーバーコントロール、テンプレート、データ バインド、ローカリゼーションなどの既存の ASP.NET 機能を使用できます。
- **既存の ASP.NET 機能のサポート** —— ASP.NET MVC では、フォーム認証および Windows 認証、URL 認証、メンバーシップおよびロール、出力およびデータキャッシュ、セッションおよびプロファイルの状態管理、状態監視、構成システム、プロバイダアーキテクチャなどの機能を使用できます。

MVC フレームワークとアプリケーションの構造

ASP.NET Web サイトでは、通常、URL はディスク上に保存されているファイルにマップされます (通常は .aspx ファイル)。これらの .aspx ファイルにはマークアップとコードが含まれており、これらが処理されることによって要求への応答が行われます。

ASP.NET MVC フレームワークは、ASP.NET Web フォーム ページとは異なり、URL をサーバーコードにマップします。URL を ASP.NET ページまたはハンドラにマップするのではなく、URL をコントローラークラスにマップします。コントローラークラスは、ユーザー入力やユーザー操作などの受信要求を処理し、ユーザー入力に基づいて適切なアプリケーションロジックとデータロジックを実行します。コントローラークラスは、通常、応答として HTML 出力を生成する個別のビューコンポーネントを呼び出します。

ASP.NET MVC フレームワークでは、モデル、ビュー、およびコントローラーの各コンポーネントが分離されています。モデルは、一般にデータベースに基づくデータを使用する、アプリケーションのビジネスロジックやドメインロジックを表します。ビューは、コントローラーによって選択され、適切な UI をレンダリングします。既定では、ASP.NET MVC フレームワークは、既存の ASP.NET ページ (.aspx)、マスター ページ (.master)、およびユーザー コントロール (.ascx) タイプを使用して、ブラウザーにレンダリングします。コントローラーは、コントローラー内の適切なアクションメソッドを見つけ、アクションメソッドの引数として使用する値を取得し、アクションメソッドが実行されるときに発生する可能性があるエラーを処理します。その後、コントローラーは要求されたビ

ューをレンダリングします。既定では、各コンポーネントセットは、MVC Web アプリケーションプロジェクトの個別のフォルダにあります。

URL ルーティング

ASP.NET MVC フレームワークは、URL をコントローラークラスにマップする柔軟性を備えた ASP.NET ルーティングエンジンを使用します。ASP.NET MVC フレームワークが受信 URL の評価と適切なコントローラーの選択を行うために使用するルーティング規則を定義できます。また、URL に定義されている変数をルーティングエンジンが自動的に解析したり、ASP.NET MVC フレームワークが値をパラメーター引数としてコントローラーに渡したりすることもできます。

MVC フレームワークとポストバック

ASP.NET MVC フレームワークは、サーバーとの通信に ASP.NET Web フォームのポストバックモデルを使用しません。代わりに、すべてのエンドユーザーインタラクションは、コントローラークラスにルーティングされます。これにより、UI ロジックとビジネスロジックの間の分離が保たれ、テストが容易になります。このため、ASP.NET ビュー状態と ASP.NET Web フォーム ページのライフサイクルイベントは、MVC ベースのビューと統合されません。

MVC プロジェクト テンプレート

ASP.NET MVC フレームワークには、MVC パターンをサポートするように構成された Web アプリケーションを作成するのに役立つ **Visual Studio プロジェクト テンプレート**が付属しています。このテンプレートから作成される新しい MVC Web アプリケーションは、必要なフォルダ、項目テンプレート、および構成ファイルエントリを備えるように構成されます。

新しい MVC Web アプリケーションを作成する場合、Visual Studio には、同時に 2 つのプロジェクトを作成するオプションが用意されています。1 つ目のプロジェクトは、アプリケーションの実装先の Web プロジェクトです。2 つ目のプロジェクトは、1 つ目のプロジェクト内の MVC コンポーネントに関する単体テストを作成できる単体テストプロジェクトです。

ASP.NET MVC アプリケーションをテストするには、.NET Framework と互換性がある単体テスト フレームワークを使用できます。Visual Studio Professional Edition 以上では、MSTest を使用できるテストプロジェクトをサポートします。

Web アプリケーションの MVC プロジェクトの構造

ASP.NET MVC Web アプリケーションプロジェクトを作成する場合、MVC コンポーネントは、次の図に示すプロジェクトフォルダに基づいて分類されます。

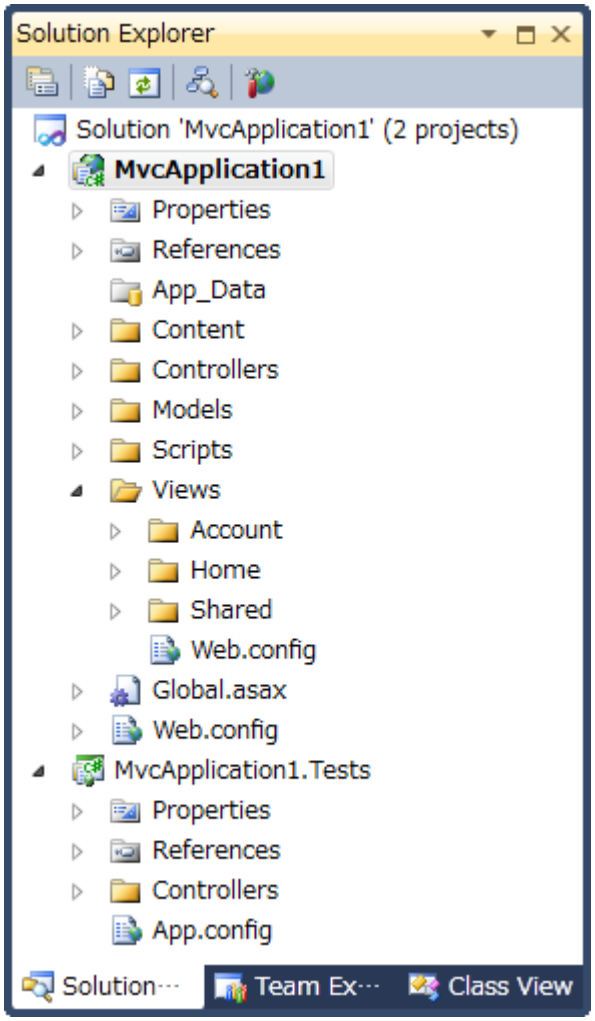

図. プロジェクトフォルダ

既定では、MVC プロジェクト内には次のフォルダがあります。

- **App_Data** —— データを格納する物理ストアです。
- **Content** —— カスケーディング スタイル シート (CSS)、イメージなどのコンテンツファイルの追加先として推奨されます。一般に、Content フォルダは静的ファイルの保存用です。
- **Controllers** —— コントローラーの保存先として推奨されます。
- **Models** —— MVC Web アプリケーションのアプリケーションモデルを表すクラスのために用意されています。このフォルダには、通常、オブジェクトを定義するコードと、データ ストアを操作するロジックを定義するコードが格納されています。
- **Scripts** —— アプリケーションをサポートするスクリプトファイルの保存先として推奨されます。既定では、このフォルダには、ASP.NET AJAX 基本ファイルと jQuery ライブラリが格納されています。
- **Views** —— ビューの保存先として推奨されます。Views は、ViewPage (.aspx)、ViewUserControl (.ascx)、および ViewMasterPage (.master) の各ファイルを使用するほか、

ビューのレンダリングに関するその他のファイルを使用します。Views フォルダ内には各コントローラー用のフォルダがあります。

グローバル URL ルーティングの既定

ルートは、**Global.asax** ファイルの **Application_Start** メソッドで初期化されます。次の例は、既定のルーティングロジックを格納している典型的な Global.asax ファイルを示しています。

<Visual Basic>

```
Public Class MvcApplication
  Inherits System.Web.HttpApplication

  Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

    ' MapRoute takes the following parameters, in order:
    ' (1) Route name
    ' (2) URL with parameters
    ' (3) Parameter defaults
    routes.MapRoute( _
      "Default", _
      "{controller}/{action}/{id}", _
      New With {.controller = "Home", .action = "Index", .id = ""} _
    )

  End Sub

  Sub Application_Start()
    RegisterRoutes(RouteTable.Routes)
  End Sub
End Class
```

<C#>

```
public class MvcApplication : System.Web.HttpApplication
{
  public static void RegisterRoutes(RouteCollection routes)
  {
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

```
    routes.MapRoute(
      "Default",                                  // Route name
      "{controller}/{action}/{id}",               // URL with parameters
      new { controller = "Home", action = "Index", id = "" }  // Parameter
defaults
    );

  }

  protected void Application_Start()
  {
    RegisterRoutes(RouteTable.Routes);
  }
}
```

ASP.NET Web フォームと MVC の互換性

経験豊富な ASP.NET 開発者であれば、ASP.NET MVC アプリケーションを作成する際、ASP.NET に関する知識の多くを適用できます。ASP.NET MVC は ASP.NET フレームワークの一部なので、ASP.NET の名前空間、クラス、およびインターフェイスのほぼすべてを MVC アプリケーションで使用できます。このトピックでは、ASP.NET Web フォームと ASP.NET MVC モデルの特長について説明します。また、ASP.NET フレームワークの機能についても、MVC アプリケーションで使用できる機能と使用を避ける必要がある機能に分けて説明します。

ASP.NET Web フォームの特長

次の一覧で、ASP.NET Web フォームの特長について説明します。

- **イベント モデル** —— Web フォームは、Windows アプリケーションに類似したイベント駆動型のプログラミングスタイルをサポートします。
- **状態管理** —— Web フォームでは、ビュー状態コントロールやサーバーベースのコントロールを使用することで、状態管理の複雑さを軽減しています。
- **ページ ベースのアーキテクチャ** —— Web フォームは、宣言型マークアップを含むページ (.aspx ファイル) と、機能を追加する分離コードファイルとを統合するアーキテクチャを提供します。
- **豊富なコントロールのセット** —— ASP.NET コミュニティは、開発時間の短縮に役立つ数百のサーバーコントロールとサーバーコンポーネントを用意しています。

ASP.NET MVC の特長

次の一覧で、ASP.NET MVC の特長について説明します。

- **関心の分離** —— ASP.NET MVC では、"関心の分離 (Separation of Concerns, SoC)" が適用されます。アプリケーションは、モデル、ビュー、コントローラーの各部分が疎結合の状態に分離されます。これにより、MVC アプリケーションはテストや保守が容易になります。
- **レンダリングされる HTML の詳細制御** —— MVC は、レンダリングされる HTML を詳細に制御します。
- **テスト駆動開発** —— MVC は、テスト駆動開発が容易になるように設計されています。MVC では、MVC プロジェクトとそのテストプロジェクトを同時に作成できます。したがって、アプリケーションのアクションメソッドごとに単体テストを作成し、Web アプリケーションの完全な要求サイクルを呼び出すことなく単体テストを実行できます。

MVC と互換性がある ASP.NET フレームワーク機能

Web フォームと **MVC** は大きく異なるテクノロジのように見えます。しかし、この 2 つのテクノロジはどちらも **ASP.NET フレームワーク** 上に構築されています。したがって、Web フォームに基づくアプリケーションの作成に使用した ASP.NET 機能のほとんどは、MVC アプリケーションの開発でも使用できます。

MVC と互換性がない ASP.NET フレームワーク 機能

ASP.NET MVC はビュー状態を使用した状態情報の維持を行わないため、状態情報を管理する必要がある場合は、他の方法を見つける必要があります。さらに、ビュー状態とポストバックに依存するサーバーコントロールは、ASP.NET MVC アプリケーションでの設計どおりには動作しません。したがって、GridView、Repeater、DataList などのコントロールは使用しないでください。

MVC アプリケーションのコントローラーとアクション メソッド

ASP.NET MVC フレームワークは、"**コントローラー**" と呼ばれるクラスに URL をマップします。コントローラーは、受信要求を処理し、ユーザーの入力や操作を処理し、適切なアプリケーションロジックを実行します。コントローラークラスは通常、個別のビューコンポーネントを呼び出して、要求の HTML マークアップを生成します。

すべてのコントローラーの基本クラスは、**ControllerBase** クラスです。このクラスは、一般的な MVC 処理を行います。**Controller** クラスは、ControllerBase を継承します。また、コントローラーの既定の実装です。Controller クラスは、次の処理段階を担当します。

- 呼び出す適切なアクションメソッドを探し、そのメソッドが呼び出し可能であるかどうかを検証する。
- アクションメソッドの引数として使用する値を取得する。

◆ アクションメソッドの実行中に発生するすべてのエラーを処理する。

◆ ASP.NET ページ（ビュー）の種類ごとに、そのページをレンダリングする既定の **WebFormViewEngine** クラスを提供する。

すべてのコントローラークラスには、"Controller" サフィックスを使用した名前を付ける必要があります。次の例は、**HomeController** という名前のサンプルのコントローラークラスを示しています。このコントローラークラスは、ビューページをレンダリングするアクションメソッドを含みます。

<Visual Basic>

```
<HandleError()> _
Public Class HomeController
  Inherits System.Web.Mvc.Controller

  Function Index() As ActionResult
    ViewData("Message") = "Welcome to ASP.NET MVC!"

    Return View()
  End Function

  Function About() As ActionResult
    Return View()
  End Function
End Class
```

<C#>

```
[HandleError]
public class HomeController : Controller
{
  public ActionResult Index()
  {
    ViewData["Message"] = "Welcome to ASP.NET MVC!";
    return View();
  }

  public ActionResult About()
  {
    return View();
```

```
  }
}
```

アクション メソッド

MVC フレームワークを使用しない ASP.NET アプリケーションでは、ユーザーインタラクションはページを中心に構成され、また、ページからのイベントおよびページ内のコントロールからのイベントの発生と処理を中心に構成されます。これに対して、ASP.NET MVC アプリケーションでのユーザーインタラクションは、コントローラーとアクションメソッドを中心に構成されます。コントローラーはアクションメソッドを定義します。コントローラーには、必要なアクションメソッドをいくつでも含めることができます。

アクションメソッドは一般に、ユーザーインタラクションと 1 対 1 の対応関係にあります。ユーザーインタラクションとは、たとえば、ブラウザーへの URL の入力、リンクのクリック、フォームの送信などです。このようなユーザーインタラクションの各々について、サーバーに要求が送信されます。それぞれの場合で、要求の URL には、MVC フレームワークがアクションメソッドの呼び出しに使用する情報が含まれます。

ユーザーがブラウザーに URL を入力すると、MVC アプリケーションは Global.asax ファイルに定義されているルーティング規則を使用して、URL の解析とコントローラーのパスの特定を行います。次に、コントローラーが適切なアクションメソッドを特定して要求を処理します。既定では、要求の URL は、コントローラー名とそれに続く操作名を含むサブパスとして処理されます。たとえば、ユーザーが http://contoso.com/MyWebSite/Products/Categories という URL を入力した場合、サブパスは、/Products/Categories の部分です。既定のルーティング規則は、"Products" をコントローラーの名前、"Categories" を操作の名前として処理します。したがって、ルーティング規則は、要求を処理するために Products コントローラーの Categories メソッドを呼び出します。URL が /Products/Detail/5 で終わる場合、既定のルーティング規則では、"Detail" を操作の名前として処理し、Products コントローラーの Detail メソッドを呼び出して要求の処理を行います。既定では、URL に含まれる値 "5" は、Detail メソッドにパラメーターとして渡されます。

次の例は、HelloWorld アクションメソッドを持つコントローラークラスを示しています。

<Visual Basic>

```
Public Class MyController
  Inherits System.Web.Mvc.Controller
  Function HelloWorld() As ActionResult
    ViewData("Message") = "Hello World!"
    Return View()
  End Function
End Class
```

<C#>

```
public class MyController : Controller
{
  public ActionResult HelloWorld()
  {
    ViewData["Message"] = "Hello World!";
    return View();
  }
}
```

ActionResult の戻り値の型

ほとんどのアクションメソッドは、**ActionResult** から派生するクラスのインスタンスを返します。ActionResult クラスは、すべての操作結果の基本です。ただし、アクションメソッドが実行するタスクに応じて、操作結果にはさまざまな型があります。たとえば、最もよく使用される操作は、 メソッドの呼び出しです。View メソッドは、ActionResult から派生する **ViewResult** クラスのインスタンスを返します。

文字列値、整数値、ブール値などの任意の型のオブジェクトを返すアクションメソッドを作成できます。これらの戻り値の型は、適切な ActionResult 型でラップされてから、応答ストリームにレンダリングされます。

次の表に、操作結果のビルトイン型とそれらの型を返す操作ヘルパーメソッドを示します。

表. 操作結果のビルトイン型とそれらの型を返す操作ヘルパーメソッド

操作結果	ヘルパー メソッド	説明
ViewResult	View	ビューを Web ページとしてレンダリングします。
PartialViewResult	PartialView	別のビュー内にレンダリングできるビューの 1 つのセクションとして定義される部分的なビューとしてレンダリングします。
RedirectResult	Redirect	URL を使用して、別のアクションメソッドにリダイレクトします。
RedirectToRouteResult	RedirectToAction RedirectToRoute	別のアクションメソッドにリダイレクトします。
ContentResult	Content	ユーザー定義のコンテンツの種類を返します。
JsonResult	Json	シリアル化された JSON オブジェクトを

		返します。
JavaScriptResult	JavaScript	クライアントで実行できるスクリプトを返します。
FileResult	File	応答に書き込むためのバイナリ出力を返します。
EmptyResult	(なし)	アクション メソッドが null の結果 (void) を返す必要がある場合に使用される戻り値を表します。

パブリック メソッドを非アクション メソッドとしてマークする

既定では、MVC フレームワークは、コントローラークラスのすべてのパブリックメソッドをアクションメソッドとして扱います。コントローラークラスがパブリックメソッドを含み、そのパブリック メソッドをアクションメソッドとして使用しない場合は、そのメソッドを **NonActionAttribute** 属性でマークする必要があります。

次の例は、**NonAction** 属性でマークされたメソッドを示しています。

<Visual Basic>

```
<NonAction()> _
Private Sub DoSomething()
  ' Method logic.
End Sub
```

<C#>

```
[NonAction]
private void DoSomething()
{
  // Method logic.
}
```

アクション メソッド パラメーター

既定では、**アクション メソッド パラメーター** の値は、要求のデータコレクションから取得されます。このデータコレクションには、フォームデータ、クエリ文字列値、および cookie 値の名前/値のペアが含まれています。

コントローラークラスは、**RouteData** インスタンスとフォームデータに基づいて、アクションメソッドを見つけ、そのアクションメソッドのパラメーター値を特定します。パラメーター値の解析が不可能で、パラメーターの型が参照型または NULL 可能な値型である場合は、null がパラメーター値として渡されます。そうでない場合は、例外がスローされます。

コントローラークラスのアクションメソッドを使用して URL パラメーター値にアクセスする方法は、いくつか存在します。**Controller** クラスは、アクションメソッドからアクセスできる **Request** プロパティと **Response** プロパティを公開します。これらのプロパティのセマンティクスは、既に ASP.NET の一部になっている **HttpRequest** オブジェクトや HttpResponse オブジェクトと同じです。ただし、Controller クラスの Request オブジェクトと Response オブジェクトは、シールクラスにならずに、HttpRequestBase 抽象クラスと HttpResponseBase 抽象クラスを実装するオブジェクトを受け付けます。これらの基本クラスによってモックオブジェクトを簡単に作成でき、それによって同様に、コントローラークラスの単体テストの作成が容易になります。

次の例は、Request オブジェクトを使用して id という名前のクエリ文字列を取得する方法を示しています。

<Visual Basic>

```
Public Sub Detail()
  Dim id As Integer = Convert.ToInt32(Request("id"))
End Sub
```

<C#>

```
public void Detail()
{
  int id = Convert.ToInt32(Request["id"]);
}
```

アクション メソッド パラメーターの自動マッピング

ASP.NET MVC フレームワークは、URL のパラメーター値をアクションメソッドのパラメーター値に自動的にマップできます。既定では、アクションメソッドがパラメーターを取る場合、MVC フレームワークは、受信要求データを調べて、同じ名前を持つ HTTP 要求値が要求に含まれているかどうかを判断します。含む場合は、要求値が自動的にアクションメソッドに渡されます。

次の例は、前の例を少し変更しています。変更したこの例では、id パラメーターが同じ id という名前の要求値にマップされることを前提としています。この自動マッピングのために、アクションメソッドでは要求からパラメーター値を取得するためのコードが不要になるため、パラメーター値の使用が容易になります。

<Visual Basic>

```
Public Function Detail(ByVal id As Integer)
  ViewData("DetailInfo") = id
  Return View()
End Function
```

<C#>

```
public ResultAction Detail(int id)
{
  ViewData["DetailInfo"] = id;
  return View();
}
```

クエリ文字列値の代わりにパラメーター値を URL の一部に埋め込むこともできます。たとえば、/Products/Detail?id=3 などのクエリ文字列を含む URL を使用する代わりに、/Products/Detail/3 のような URL を使用できます。既定のルートマッピング規則の形式は、**/{controller}/{action}/{id}** です。URL 中のコントローラー名と操作名の後に URL サブパスがある場合、そのサブパスは、id という名前のパラメーターとして扱われ、パラメーター値として自動的にアクションメソッドに渡されます。

MVC フレームワークは、アクションメソッドの省略可能な引数もサポートします。MVC フレームワークの省略可能なパラメーターは、**MVC 1.0** では、コントローラーアクションメソッドに対して NULL 可能な型の引数を使用することによって処理されます。たとえば、メソッドがクエリ文字列の一部に日付を受け取り、クエリ文字列パラメーターの指定がないときは既定で今日の日付を設定するには、次の例に示すようなコードを使用できます。

<Visual Basic>

```
Public Function ShowArticles(ByVal date As DateTime?)
  If Not date.HasValue Then
    date = DateTime.Now
  End If
  ' ...
End Function
```

<C#>

```
public ActionResult ShowArticles(DateTime? date)
{
  if (!date.HasValue)
  {
    date = DateTime.Now;
  }
  // ...
}
```

要求が date パラメーターの値を含む場合は、その値が ShowArticles メソッドに渡されます。要求がこのパラメーターの値を含まない場合、引数は null になり、コントローラーは存在しないパラメーターを扱うために必要な処理を行うことができます。

一方、**MVC 2** では、**System.ComponentModel.DefaultValueAttribute** クラスを使用して、アクションメソッドの引数に対してデフォルト値を設定することができます。

<Visual Basic>

```
Imports System.ComponentModel
Public Function Details(<DefaultValue(0)> ByVal id As Integer)
  ' ...
End Function
```

<C#>

```
using System.ComponentModel;
public ActionResult Details([DefaultValue(0)] int id)
{
  // ...
}
public ActionResult Details2(int id = 0)
{
  // ...
}
```

HTML ヘルパーを使用したフォームのレンダリング

ASP.NET MVC フレームワークは、HTML を簡単にビューに表示できるヘルパーメソッドを備えています。このトピックでは、最もよく利用される HTML ヘルパーの使用方法について説明します。最後のセクションでは、このトピックで説明した HTML ヘルパーの組み込み例を示します。

使用可能な HTML ヘルパー

次の一覧に、現在使用できるいくつかの HTML ヘルパーを示します。一覧の中でアスタリスク (*) が付いているヘルパーについては、このトピックで説明します。

- **ActionLink** —— アクションメソッドにリンクします。
- **BeginForm** * —— フォームの開始をマークし、そのフォームをレンダリングするアクションメソッドにリンクします。
- **CheckBox** * —— チェックボックスをレンダリングします。
- **DropDownList** * —— ドロップダウンリストをレンダリングします。

- **Hidden** —— ユーザーに対しては表示されないフォームに情報を埋め込みます。
- **ListBox** —— リストボックスをレンダリングします。
- **Password** —— パスワードを入力するためのテキストボックスをレンダリングします。
- **RadioButton** * —— ラジオボタンをレンダリングします。
- **TextArea** —— テキスト領域 (複数行テキストボックス) をレンダリングします。
- **TextBox** * —— テキストボックスをレンダリングします。

BeginForm ヘルパーの使用

BeginForm ヘルパーは、HTML フォームの開始をマークし、HTML の **form** 要素をレンダリングします。BeginForm ヘルパー メソッドには、いくつかのオーバーライドがあります。次の例で示す BeginForm ヘルパーのバージョンは、アクションメソッドの名前とフォームを送信するためのコントローラーの 2 つのパラメーターを使用します。BeginForm ヘルパーは、**IDisposable** インターフェイスを実装します。これにより、ASP.NET AJAX の場合と同様に、**using** キーワード (Visual Basic の場合は Using) を使用できます。

次の例は、using のパターンで BeginForm ヘルパーを使用する方法を示しています。

<Visual Basic>

```
<% Using Html.BeginForm("HandleForm", "Home") %>
  <!-- Form content goes here -->
<% End Using %>
```

<C#>

```
<% using(Html.BeginForm("HandleForm", "Home")) %>
<% { %>
  <!-- Form content goes here -->
<% } %>
```

BeginForm ヘルパーは、宣言的に使用することもできます。BeginForm を宣言的に使用することと HTML の form タグを使用することの違いは、BeginForm ではアクションメソッドとアクション属性に既定値が割り当てられ、このためにマークアップが簡素化されるということです。次の例は、宣言型マークアップを使用して、フォームの開始と終了をマークしています。

<Visual Basic>

```
<% Html.BeginForm() %>
  <!-- Form content goes here -->
<% Html.EndForm() %>
```

<C#>

```
<% Html.BeginForm(); %>
```

```
  <!-- Form content goes here -->
<% Html.EndForm(); %>
```

CheckBox ヘルパーの使用

CheckBox ヘルパー メソッドは、指定された名前を持つチェックボックスをレンダリングします。レンダリングされたコントロールは、ブール値 (true または false) を返します。次の例は、CheckBox ヘルパーメソッドのマークアップを示しています。

```
<%= Html.CheckBox("bookType") %>
```

DropDownList ヘルパーの使用

DropDownList ヘルパーは、ドロップダウンリストをレンダリングします。最も単純な形態では、DropDownList は 1 つのパラメーターを取ります。このパラメーターは、SelectList 型の値を持ち、ドロップダウンリストのオプション値を含む ViewData キーの名前です。MVC フレームワークは、ViewData の ModelState プロパティを使用して、選択された値を決定します。ModelState プロパティが空の場合、このフレームワークは Selected プロパティが設定されている項目を検索します。

次の例は、DropDownList ヘルパーメソッドのマークアップを示しています。

```
<%= Html.DropDownList("pets") %>
```

次のコードは、値が List オブジェクトに追加される Index アクションメソッドの一部です。List オブジェクトが SelectList のインスタンスに渡されてから、そのインスタンスが ViewData オブジェクトに追加されます。

<Visual Basic>

```
Dim petList As List(Of String) = New List(Of String)
petList.Add("Dog")
petList.Add("Cat")
petList.Add("Hamster")
petList.Add("Parrot")
petList.Add("Gold fish")
petList.Add("Mountain lion")
petList.Add("Elephant")

ViewData("pets") = New SelectList(petList)
```

<C#>

```
List<string> petList = new List<string>();
petList.Add("Dog");
petList.Add("Cat");
```

```
petList.Add("Hamster");
petList.Add("Parrot");
petList.Add("Gold fish");
petList.Add("Mountain lion");
petList.Add("Elephant");

ViewData["Pets"] = new SelectList(petList);
```

RadioButton ヘルパーの使用

RadioButton ヘルパー メソッドは、ラジオボタンをレンダリングします。最も単純な形態では、このメソッドは 3 つのパラメーターを取ります。このパラメーターは、コントロールグループの名前、オプション値、および最初からラジオボタンを選択しておくかどうかを決定する Boolean 値です。次のマークアップは、RadioButton ヘルパーメソッドのマークアップを示しています。

```
Select your favorite color:<br />
<%= Html.RadioButton("favColor", "Blue", true) %> Blue <br />
<%= Html.RadioButton("favColor", "Purple", false)%> Purple <br />
<%= Html.RadioButton("favColor", "Red", false)%> Red <br />
<%= Html.RadioButton("favColor", "Orange", false)%> Orange <br />
<%= Html.RadioButton("favColor", "Yellow", false)%> Yellow <br />
<%= Html.RadioButton("favColor", "Brown", false)%> Brown <br />
<%= Html.RadioButton("favColor", "Green", false)%> Green
```

TextBox ヘルパーの使用

TextBox ヘルパー メソッドは、指定された名前を持つテキストボックスをレンダリングします。次のマークアップは、TextBox ヘルパーメソッドのマークアップを示しています。

```
Enter your name: <%= Html.TextBox("name") %>
```

サンプル アプリケーション

次の例は、前の例を抽出する元になった完全なサンプルです。Index ページには、HTML ヘルパーメソッドを実装するフォームが表示されます。ユーザーがフォームを送信すると、フォームは **HandleForm** アクションメソッドによって処理され、このメソッドはユーザーが送信した情報が表示されるビューを生成します。

次の例は、HomeController クラスを示しています。

<Visual Basic>

```
<HandleError()> _
```

```
Public Class HomeController
  Inherits System.Web.Mvc.Controller

  Function Index() As ActionResult
    ViewData("Message") = "Welcome to ASP.NET MVC!"

    Dim petList As List(Of String) = New List(Of String)
    petList.Add("Dog")
    petList.Add("Cat")
    petList.Add("Hamster")
    petList.Add("Parrot")
    petList.Add("Gold fish")
    petList.Add("Mountain lion")
    petList.Add("Elephant")

    ViewData("pets") = New SelectList(petList)

    Return View()
  End Function

  Function About() As ActionResult
    Return View()
  End Function

  Public Function HandleForm(ByVal name As String, ByVal favColor As
String, ByVal bookType As Boolean, ByVal pets As String) As ActionResult
    ViewData("name") = name
    ViewData("favColor") = favColor
    ViewData("bookType") = bookType
    ViewData("pet") = pets

    Return View("FormResults")
  End Function
End Class
```

<C#>

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcHtmlHelpers.Controllers
{
  [HandleError]
  public class HomeController : Controller
  {
    public ActionResult Index()
    {
      ViewData["Message"] = "Welcome to ASP.NET MVC!";

      List<string> petList = new List<string>();
      petList.Add("Dog");
      petList.Add("Cat");
      petList.Add("Hamster");
      petList.Add("Parrot");
      petList.Add("Gold fish");
      petList.Add("Mountain lion");
      petList.Add("Elephant");

      ViewData["Pets"] = new SelectList(petList);

      return View();
    }

    public ActionResult About()
    {
      return View();
    }

    public ActionResult HandleForm(string name, string favColor,
```

```
                                    Boolean bookType, string pets)
    {
      ViewData["name"] = name;
      ViewData["favColor"] = favColor;
      ViewData["bookType"] = bookType;
      ViewData["pet"] = pets;

      return View("FormResults");
    }
  }
}
```

次の例は、Index ビューを示しています。

<Visual Basic>

```
<h2><%= Html.Encode(ViewData("Message")) %></h2>
<br /><br />
<% Using Html.BeginForm("HandleForm", "Home") %>
  Enter your name: <%= Html.TextBox("name") %>
  <br /><br />
  Select your favorite color:<br />
  <%= Html.RadioButton("favColor", "Blue", true) %> Blue <br />
  <%= Html.RadioButton("favColor", "Purple", false) %> Purple <br />
  <%= Html.RadioButton("favColor", "Red", false) %> Red <br />
  <%= Html.RadioButton("favColor", "Orange", false) %> Orange <br />
  <%= Html.RadioButton("favColor", "Yellow", false) %> Yellow <br />
  <%= Html.RadioButton("favColor", "Brown", false) %> Brown <br />
  <%= Html.RadioButton("favColor", "Green", false) %> Green
  <br /><br />
  <%= Html.CheckBox("bookType") %>
    I read more fiction than non-fiction.<br />
  <br /><br />
  My favorite pet: <%= Html.DropDownList("pets") %>
  <br /><br />
  <input type="submit" value="Submit" />
<% End Using%>
```

<C#>

```
<h2><%= Html.Encode(ViewData["Message"]) %></h2>
<br /><br />
<% using(Html.BeginForm("HandleForm", "Home")) %>
<% { %>
  Enter your name: <%= Html.TextBox("name") %>
  <br /><br />
  Select your favorite color:<br />
  <%= Html.RadioButton("favColor", "Blue", true) %> Blue <br />
  <%= Html.RadioButton("favColor", "Purple", false) %> Purple <br />
  <%= Html.RadioButton("favColor", "Red", false) %> Red <br />
  <%= Html.RadioButton("favColor", "Orange", false) %> Orange <br />
  <%= Html.RadioButton("favColor", "Yellow", false) %> Yellow <br />
  <%= Html.RadioButton("favColor", "Brown", false) %> Brown <br />
  <%= Html.RadioButton("favColor", "Green", false) %> Green
  <br /><br />
  <%= Html.CheckBox("bookType") %>
    I read more fiction than non-fiction.<br />
  <br /><br />
  My favorite pet: <%= Html.DropDownList("pets") %>
  <br /><br />
  <input type="submit" value="Submit" />
<% } %>
```

次の例は、FormResults ビューを示しています。

<Visual Basic>

```
<h2>FormResults</h2>
<p>
Your name: <b><%= Html.Encode(ViewData("name")) %></b>
</p>
<p>
Your favorite color: <b><%= Html.Encode(ViewData("favColor")) %></b>
</p>
<%  If (ViewData("bookType").Equals(True)) Then %>
<p>You read more <b>fiction</b> than non-fiction.</p>
```

```
<% Else %>
<p>You read more <b>non-fiction</b> than fiction.</p>
<% End If %>
Your favorite pet: <b><%= Html.Encode(ViewData("pet")) %></b>
```

<C#>

```
<h2>FormResults</h2>
<p>
Your name: <b><%= Html.Encode(ViewData["name"]) %></b>
</p>
<p>
Your favorite color: <b><%= Html.Encode(ViewData["favColor"]) %></b>
</p>
<% if (ViewData["bookType"].Equals(true))
   { %>
<p>You read more <b>fiction</b> than non-fiction.</p>
<% }
   else
   { %>
<p>You read more <b>non-fiction</b> than fiction.</p>
<% } %>
Your favorite pet: <b><%= Html.Encode(ViewData["pet"]) %></b>
```

AJAX スクリプトの追加

ASP.NET AJAX を使用すると、Web アプリケーションでサーバーからデータを非同期で取得し、既存のページの一部を更新できます。これによって Web アプリケーションの応答性が高まり、ユーザーエクスペリエンスが向上します。

このチュートリアルでは、初めて ASP.NET MVC アプリケーションに ASP.NET AJAX 機能を追加する際の手順を示します。

必要条件

このチュートリアルを実行するには、以下が必要です。

- **Microsoft Visual Studio 2008** —— このチュートリアルでは、Microsoft Visual Web Developer Express は使用できません。

◆ **ASP.NET MVC 2** —— このフレームワークは、マイクロソフトダウンロードセンターの「ASP.NET MVC 2 ページ」からダウンロードできます。

新しい MVC プロジェクトを作成する

初めに、新しい ASP.NET MVC プロジェクトを作成します。このチュートリアルを簡単にするために、ASP.NET MVC プロジェクトのオプションであるテストプロジェクトは作成しません。

新しい MVC プロジェクトを作成するには

以下の手順で行います。

1. [ファイル] メニューの [新しいプロジェクト] をクリックします。
2. [プロジェクトの種類] の [新しいプロジェクト] ダイアログ ボックスで、[Visual Basic] または [Visual C#] を展開し、[Web] をクリックします。
3. [Visual Studio にインストールされたテンプレート] で、[ASP.NET MVC Web アプリケーション] を選択します。
4. [名前] ボックスに「MvcAjaxApplication」と入力します。
5. [場所] ボックスに、プロジェクトフォルダの名前を入力します。
6. [ソリューション用のディレクトリを作成] を選択します。
7. [OK] をクリックします。
8. [テストプロジェクトの作成] ダイアログボックスで、[いいえ、単体テストプロジェクトを作成しません] を選択します。
9. [OK] をクリックします。新しい MVC アプリケーションプロジェクトが生成されます。

ASP.NET AJAX スクリプト ライブラリへの参照

ASP.NET AJAX のクライアント機能は、2 つのスクリプト ライブラリ **MicrosoftAjax.js** と **MicrosoftMvcAjax.js** によってサポートされます。これらのスクリプトのリリースバージョンとデバッグバージョンは、プロジェクトの Scripts フォルダにあります。クライアントスクリプトでこれらのライブラリにアクセスするには、現在のプロジェクトの MVC ビューにライブラリへの参照を追加する必要があります。

ASP.NET AJAX スクリプト ライブラリを参照するには

以下の手順で行います。

1. [ソリューション エクスプローラー] で、Views フォルダを展開し、Shared フォルダを展開します。
2. Site.Master ファイルをダブルクリックして開きます。
3. head 要素の末尾に、以下のマークアップを追加します。

<C#>

```
<script type="text/javascript"
    src="<%= Url.Content("~/Scripts/MicrosoftAjax.debug.js") %>">
</script>
<script type="text/javascript"
    src="<%= Url.Content("~/Scripts/MicrosoftMvcAjax.debug.js") %>">
</script>
```

HomeController クラスへのアクション メソッドの追加

次に、クライアントスクリプトから非同期で呼び出せるアクションメソッドを 2 つ追加します。**GetStatus** メソッドは、単に "OK" の状態と現在の時刻を返します。**UpdateForm** メソッドは、HTML フォームからの入力を受け取り、現在の時刻を表示するメッセージを返します。

HomeController クラスにアクション メソッドを追加するには

以下の手順で行います。

1. [ソリューション エクスプローラー] で、Controllers フォルダを展開し、**HomeController** クラスをダブルクリックして開きます。
2. **About** メソッドの後に次のコードを追加します。

<Visual Basic>

```
Public Function GetStatus() As String
  Return "Status OK at " + DateTime.Now.ToLongTimeString()
End Function

Public Function UpdateForm(ByVal textBox1 As String) As String
  If textBox1 <> "Enter text" Then
    Return "You entered: """ + textBox1.ToString() + """ at " +
DateTime.Now.ToLongTimeString()
  End If

  Return [String].Empty
End Function
```

<C#>

```
public string GetStatus()
{
  return "Status OK at " + DateTime.Now.ToLongTimeString();
}
```

```
public string UpdateForm(string textBox1)
{
  if (textBox1 != "Enter text")
  {
    return "You entered: ¥"" + textBox1.ToString() + "¥" at " +
      DateTime.Now.ToLongTimeString();
  }

  return String.Empty;
}
```

Index ページの再定義

最後に、ASP.NET MVC の Visual Studio プロジェクトに自動的に追加されている Index.aspx ページのコンテンツを再配置します。新しい Index.aspx ページには、ページがレンダリングされた時刻、メッセージを非同期で更新するためのリンク付きの状態メッセージ、テキスト文字列を送信するフォームが表示されます。

Index ページを再定義するには

以下の手順で行います。

1. [ソリューション エクスプローラー] で、Views フォルダを展開し、Home フォルダを展開し、Index ビューを開きます。
2. Content コントロールの内容を以下のマークアップで置き換えます。

<Visual Basic>

```
<h2><%= Html.Encode(ViewData("Message")) %></h2>
<p>
  Page Rendered: <%= DateTime.Now.ToLongTimeString() %>
</p>
<span id="status">No Status</span>
<br />
<%= Ajax.ActionLink("Update Status", "GetStatus",
    New AjaxOptions With {.UpdateTargetId = "status"}) %>
<br /><br />
<% Using (Ajax.BeginForm("UpdateForm",
    New AjaxOptions With {.UpdateTargetId = "textEntered"})) %>
  <%= Html.TextBox("textBox1","Enter text") %>
```

```
  <input type="submit" value="Submit"/>
  <br />
  <span id="textEntered">Nothing Entered</span>
<% End Using %>
```

<C#>

```
<h2><%= Html.Encode(ViewData["Message"]) %></h2>
<p>
    Page Rendered: <%= DateTime.Now.ToLongTimeString() %>
</p>
<span id="status">No Status</span>
<br />
<%= Ajax.ActionLink("Update Status", "GetStatus",
    new AjaxOptions{UpdateTargetId="status" }) %>
<br /><br />
<% using(Ajax.BeginForm("UpdateForm",
    new AjaxOptions{UpdateTargetId="textEntered"})) { %>
  <%= Html.TextBox("textBox1","Enter text") %>
  <input type="submit" value="Submit"/><br />
  <span id="textEntered">Nothing Entered</span>
<% } %>
```

この例では、**Ajax.ActionLink** メソッドを呼び出して非同期のリンクを作成します。このメソッドには、いくつかのオーバーライドがあります。この例では、3 つのパラメーターを受け取ります。1 番目のパラメーターは、リンクのテキストです。2 番目のパラメーターは、呼び出す MVC アクションメソッドです。3 番目のパラメーターは、呼び出しの目的を定義する **AjaxOptions** オブジェクトです。この場合、コードは、ID が status である DOM 要素を更新します。

フォームは Ajax.Form メソッドを使用して定義されます。これにもいくつかのオーバーライドがあります。この例では、2 つのパラメーターを受け取ります。3 番目のパラメーターは、呼び出すアクションメソッドです。2 番目のパラメーターは、別の AjaxOptions オブジェクトです。これは、ID が textEntered の DOM 要素を更新するように指定します。

アプリケーションの実行

これで、アプリケーションを実行しで動作を確認できます。

MVC アプリケーションを実行するには

以下の手順で行います。

1. Ctrl キーを押しながら F5 キーを押します。
2. レンダリングされた時刻がページに表示されます。
3. [状態の更新] のリンクをクリックします。
4. 状態メッセージが更新時刻で更新されます。状態メッセージのみが更新されたことに注意してください。
5. テキストボックスにテキストを入力し、[送信します] ボタンをクリックします。
6. 入力したテキストと入力時刻のメッセージが表示されます。この場合も、処理されたのはフォームだけです。

ASP.NET MVC アプリケーションを展開する

ホスティングプロバイダが既に ASP.NET MVC をホスティングサーバーにインストールしている場合、MVC アプリケーションの配置方法は、ASP.NET Web アプリケーションを配置する場合とまったく同じです。しかし、ホスティングプロバイダが現在 ASP.NET MVC をサポートしていない場合は、配置済みのアプリケーションの Bin フォルダに必要な MVC アセンブリをアップロードする必要があります。

ASP.NET MVC をインストールすると、次のアセンブリがコンピュータ上のグローバルアセンブリキャッシュ (GAC) に配置されます。

- **System.Web.Mvc** (ASP.NET MVC アセンブリ)
- **System.Web.Routing** (ASP.NET MVC が必要とする .NET Framework アセンブリ)
- **System.Web.Abstractions** (ASP.NET MVC が必要とする .NET Framework アセンブリ)

ホスティングプロバイダが ASP.NET バージョン 3.5 Server Pack 1 をインストールしている場合、アップロードする必要があるのは System.Web.Mvc アセンブリだけです。ホスティングプロバイダが ASP.NET バージョン 3.5 またはそれ以前のバージョンを使用している場合は、上に示したすべてのアセンブリを配置する必要があります。

MVC アプリケーションの配置

ASP.NET MVC アプリケーションを配置するには

以下の手順で行います。

1. Visual Studio で、配置するプロジェクトを開きます。
2. [ソリューション エクスプローラー] で、[参照設定] ノードを展開します。
3. 次のアセンブリを選択します。
   - ✓ System.Web.Mvc
   - ✓ System.Web.Routing

- ✓ System.Web.Abstractions

4. [プロパティ] ウィンドウで [ローカル コピー] を [True] に設定します。
5. [ソリューションエクスプローラー] でプロジェクトを右クリックし、[発行] をクリックします。[Web の発行] ダイアログボックスが表示されます。
6. [ターゲットの場所: (http:、ftp:、またはディスク パス)] で、ローカル フォルダを参照し、[開く] をクリックします。
7. [一致するファイルをローカル コピーに置き換える] または [発行の前に既存のファイルをすべて削除する] を選択します。
8. [コピー] の下の選択肢から、必要に応じて [このアプリケーションの実行に必要なファイルのみ]、[すべてのプロジェクト ファイル]、[ソース プロジェクト フォルダのすべてのファイル] のいずれかを選択します。
9. データベースファイルなど App_Data フォルダ内のファイルがアプリケーションに含まれる場合は、[App_Data フォルダのファイルを含める] を選択します。
10. 10.[発行] をクリックします。アプリケーションの配置に必要なすべてのファイルが発行先フォルダにコピーされます。
11. MVC がインストールされていないステージングサーバーまたは仮想マシンにファイルを配置して、アプリケーションをテストします。ステージングサーバーまたは仮想マシンにアクセスできない場合は、MVC をアンインストールして、アプリケーションをローカルでテストすることもできます。
12. アプリケーションをホスティングプロバイダにアップロードします。

URL ルーティング

URL ルーティングを使用すると、Web サイトの特定のファイルにマップする必要がない URL を使用できます。URL をファイルにマップする必要がないため、ユーザーの操作を表すわかりやすい URL を使用できます。

ASP.NET 3.5 SP1 では、ASP.NET MVC フレームワークおよび ASP.NET Dynamic Data によってルーティングが拡張され、MVC アプリケーションと ASP.NET Dynamic Data で URL ルーティング機能が使用できます。一方、ASP.NET 4 からは ASP.NET Web フォームでも URL ルーティング機能が使用できるようになっています。

ルーティングを使用しない ASP.NET アプリケーションでは、URL の受信要求は、通常、その要求を処理する物理ファイル (.aspx ファイルなど) にマップされます。たとえば、**http://server/application/Products.aspx?id=4** への要求は、ブラウザーへの応答をレンダリングするためのコードとマークアップを含む Products.aspx という名前のファイルにマップされます。この Web ページは、id=4 というクエリ文字列値を使用して、表示するコンテンツの種類を決定します。

URL ルーティングでは、要求ハンドラ ファイルにマップされる URL パターンを定義できますが、このパターンに URL 内のファイルの名前が含まれるとは限りません。また、URL パターンにプレースホルダーを含めることにより、クエリ文字列を使用しなくても要求ハンドラに変数データを渡すことができます。

たとえば、**http://server/application/Products/show/beverages** という要求により、ルーティング パーサーからページ ハンドラに Products、show、および beverages という値を渡すことができます。この例では、URL パターン server/application/{area}/{action}/{category} を使用してルートが定義されている場合に、ページハンドラは、主要領域に関連付けられている値が Products、主要操作の値が show、および主要カテゴリの値が beverages であるディクショナリコレクションを受け取ります。URL ルーティングによって管理されない要求の場合、アプリケーションは /Products/show/beverages というフラグメントをファイルのパスとして解釈します。

ルート

ルートは、ハンドラにマップされる URL パターンです。ハンドラには、Web フォーム アプリケーション内の物理ファイル (.aspx ファイルなど) を指定できます。また、要求を処理するクラス (MVC アプリケーション内のコントローラーなど) も指定できます。ルートを定義するには、URL パターン、ハンドラ、およびルートの名前 (必要な場合) を指定して、**Route** クラスのインスタンスを作成します。

ルートをアプリケーションに追加するには、Route オブジェクトを **RouteTable** クラスの静的な Routes プロパティに追加します。Routes プロパティは、アプリケーションのすべてのルートを格納する **RouteCollection** オブジェクトです。

URL パターン

URL パターンには、リテラル値、および URL パラメーターと呼ばれる可変プレースホルダーを含めることができます。リテラル値とプレースホルダーは、URL のセグメントに配置され、スラッシュ (/) 文字で区切られます。

要求が実行されると、URL は解析されてセグメントとプレースホルダーに分けられ、変数値が要求ハンドラに渡されます。このプロセスは、クエリ文字列内のデータが解析され、要求ハンドラに渡されるプロセスと同様です。どちらの場合も、変数情報は URL に含まれ、キー/値ペアの形式でハンドラに渡されます。クエリ文字列では、キーと値の両方が URL に含まれます。ルートでは、キーは URL パターンで定義されたプレースホルダー名であり、URL には値のみが含まれます。

URL パターンでは、プレースホルダーを中かっこ ({ および }) で囲んで定義します。1 つのセグメントに複数のプレースホルダーを定義できますが、その場合はリテラル値で区切る必要があります。たとえば、**{language}-{country}/{action}** は有効なルートパターンです。ただし、

{language}{country}/{action} は、プレースホルダー間にリテラル値や区切り記号がないため、有効なパターンではありません。そのため、ルートの language プレースホルダーの値と country プレースホルダーの値をどこで区切ればよいかを判断できません。

次の表に、有効なルートパターンと、各パターンに一致する URL 要求の例を示します。

表. ルート定義と一致する *URL* の例

ルート定義	一致する URL の例
{controller}/{action}/{id}	/Products/show/beverages
{table}/Details.aspx	/Products/Details.aspx
blog/{action}/{entry}	/blog/show/123
{reporttype}/{year}/{month}/{day}	/sales/2008/1/5
{locale}/{action}	/US/show
{language}-{country}/{action}	/en-US/show

MVC アプリケーションにおける一般的な URL パターン

通常、MVC アプリケーションにおけるルートの URL パターンには、{controller} と {action} の各プレースホルダーが含まれます。

受け取った要求は **UrlRoutingModule** オブジェクトにルーティングされ、その後 **MvcHandler HTTP** ハンドラにルーティングされます。MvcHandler HTTP ハンドラは、"Controller" サフィックスを URL のコントローラー値に付加して、要求を処理するコントローラーの種類名を決定することにより、呼び出すコントローラーを決定します。また、URL の action の値から、呼び出すアクションメソッドを決定します。

たとえば、URL パス "/Products" を含む URL は、ProductsController という名前のコントローラーにマップされます。action パラメーターの値は、呼び出すアクションメソッドの名前です。URL パス "/Products/show" を含む URL は、**ProductsController** クラスの **Show** メソッドを呼び出します。

次の表に、既定の URL パターン、および既定のルートで処理される URL 要求の例を示します。

表. 既定の *URL* パターンと一致する *URL* の例

既定の URL パターン	一致する URL の例
{controller}/{action}/{id}	http://server/application/Products/show/beverages
{resource}.axd/{*pathInfo}	http://server/application/WebResource.axd?d=...

IIS 7.0 以降では、ファイル名拡張子は不要です。IIS 6.0 では、次の例に示すように、ファイル名拡張子 ".mvc" を URL パターンに追加する必要があります。

```
{controller}.mvc/{action}/{id}
```

Web フォーム アプリケーションへのルートの追加

Web フォーム アプリケーションでは、**RouteCollection** クラスの **MapPageRoute(String, String, String)** メソッドを使用してルートを作成します。MapPageRoute メソッドは、Route オブジェクトを作成し、それを RouteCollection オブジェクトに追加します。この Route オブジェクトのプロパティは、MapPageRoute メソッドに渡すパラメーターで指定します。

次の例は、Global.asax ファイルのコードを示しています。このコードでは、action と categoryName という名前の 2 つの URL パラメーターを定義する Route オブジェクトが追加されます。このパターンを含む URL は、Categories.aspx という名前の物理ページに誘導されます。

<Visual Basic>

```
Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
  RegisterRoutes(RouteTable.Routes)
End Sub

Shared Sub RegisterRoutes(routes As RouteCollection)
  routes.MapPageRoute("",
    "Category/{action}/{categoryName}",
    "~/categoriespage.aspx")
End Sub
```

<C#>

```
protected void Application_Start(object sender, EventArgs e)
{
  RegisterRoutes(RouteTable.Routes);
}

public static void RegisterRoutes(RouteCollection routes)
{
  routes.MapPageRoute("",
    "Category/{action}/{categoryName}",
    "~/categoriespage.aspx");
}
```

MVC アプリケーション へのルートの追加

MVC 規則に従ってコントローラーを実装した場合、つまり ControllerBase クラスから派生したクラスを作成し、"Controller" で終わる名前を指定した場合は、MVC アプリケーションでルートを手動で追加する必要はありません。あらかじめ設定されたルートにより、コントローラークラスに実装したアクションメソッドが呼びされます。

次の例は、既定の MVC ルートを作成する Global.asax ファイル内のコードを示しています。このコードは、MVC アプリケーション用の Visual Studio プロジェクトテンプレートで定義されています。

<Visual Basic>

```
Public Class MvcApplication
  Inherits System.Web.HttpApplication

  Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

    ' MapRoute takes the following parameters, in order:
    ' (1) Route name
    ' (2) URL with parameters
    ' (3) Parameter defaults
    routes.MapRoute( _
      "Default", _
      "{controller}/{action}/{id}", _
      New With {.controller = "Home", .action = "Index", .id = ""} _
    )

  End Sub

  Sub Application_Start()
    RegisterRoutes(RouteTable.Routes)
  End Sub
End Class
```

<C#>

```
public class MvcApplication : System.Web.HttpApplication
{
```

```
  public static void RegisterRoutes(RouteCollection routes)
  {
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
      "Default",                              // Route name
      "{controller}/{action}/{id}",           // URL with parameters
      new { controller = "Home", action = "Index", id = "" }  // Parameter
defaults
    );

  }

  protected void Application_Start()
  {
    RegisterRoutes(RouteTable.Routes);
  }
}
```

URL パラメーターの既定値の設定

ルートを定義するときに、パラメーターの既定値を割り当てることができます。既定値は、そのパラメーターの値が URL に含まれていない場合に使用されます。ルートの既定値を設定するには、**Route** クラスの Defaults プロパティにディクショナリオブジェクトを割り当てます。次の例は、MapPageRoute(String, String, String, Boolean, RouteValueDictionary) メソッドを使用して、既定値が設定されたルートを追加する方法を示しています。

<Visual Basic>

```
Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
  RegisterRoutes(RouteTable.Routes)
End Sub

Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
  routes.MapPageRoute("",
    "Category/{action}/{categoryName}",
    "~/categoriespage.aspx",
    True,
```

```
    New RouteValueDictionary(New With _
      {.categoryName = "food", _
       .action = "show"} ))
End Sub
```

<C#>

```
void Application_Start(object sender, EventArgs e)
{
  RegisterRoutes(RouteTable.Routes);
}

public static void RegisterRoutes(RouteCollection routes)
{
  routes.MapPageRoute("",
    "Category/{action}/{categoryName}",
    "~/categoriespage.aspx",
    true,
    new RouteValueDictionary
      {{"categoryName", "food"}, {"action", "show"}});
}
```

URL ルーティングによって URL 要求が処理されるとき、この例のルート定義（categoryName の既定値は food、action の既定値は show）から、次の表に示す結果が生成されます。

表. *URL* とパラメーター値

URL	パラメーター値
/Category	*action* = "show" (既定値) *categoryName* = "food" (既定値)
/Category/add	*action* = "add" *categoryName* = "food" (既定値)
/Category/add/beverages	*action* = "add" *categoryName*= "beverages"

MVC アプリケーションでは、**RouteCollectionExtensions.MapRoute** メソッドをオーバーロード（**MapRoute(RouteCollection, String, String, Object, Object)** など）することによって既定値を指定できます。

URL パターン内での可変数のセグメントの処理

場合によっては、可変数の URL セグメントを含む URL 要求を処理する必要があります。ルートを定義するときに、パターンに含まれるセグメント数を超えるセグメントが URL に設定された場合、追加セグメントを最後のセグメントの一部として扱うかどうかを指定できます。追加セグメントを最後のセグメントの一部として処理するには、最後のパラメーターにアスタリスク (*) を付加します。これは汎用的なパラメーターと呼ばれます。汎用的なパラメーターを含むルートは、最後のパラメーターに該当する値を含まない URL にも一致します。次の例は、可変数のセグメントに一致するルート パターンを示しています。

```
query/{queryname}/{*queryvalues}
```

URL ルーティングによって URL 要求が処理されるとき、この例のルート定義から、次の表に示す結果が生成されます。

表. URL とパラメーター値

URL	パラメーター値
/query/select/bikes/onsale	*queryname* = "select" *queryvalues* = "bikes/onsale"
/query/select/bikes	*queryname* = "select" *queryvalues* = "bikes"
/query/select	*queryname* = "select" *queryvalues* = 空の文字列

ルートへの制約の追加

URL 要求とルート定義が URL 内のパラメーター数の点で一致することに加えて、パラメーター内の値が特定の制約に従うように指定することもできます。ルートの制約を満たさない値が URL に含まれている場合、そのルートは要求の処理に使用されません。制約を追加することにより、URL パラメーターに含まれる値を確実にアプリケーションにとって有効な値とすることができます。

制約は、正規表現を使用するか、または IRouteConstraint インターフェイスを実装するオブジェクトを使用して定義します。ルート定義を Routes コレクションに追加する場合は、検証テストを含む **RouteValueDictionary** オブジェクトを作成して制約を追加します。制約の適用対象となるパラメーターは、ディクショナリのキーによって識別されます。ディクショナリの値には、正規表現を表す文字列か、または **IRouteConstraint** インターフェイスを実装するオブジェクトを使用できます。

文字列を指定した場合、ルーティングはその文字列を正規表現として扱い、**Regex** クラスの **IsMatch** メソッドを呼び出して、パラメーター値が有効であるかどうかを確認します。この正規表現では、大文字と小文字は常に区別されません。

次の例は、locale パラメーターおよび year パラメーターが格納できる値を制限する制約を含むルートを、**MapPageRoute** メソッドを使用して作成する方法を示しています。MVC アプリケーションでは、**MapRoute** メソッドを使用します。

<Visual Basic>

```
Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
  routes.MapPageRoute("DetailRoute ",
    "Detail/{locale}/{year}",
    "~/detail.aspx",
    True,
    New RouteValueDictionary(New With _
      {.locale = "en-US", _
       .year = DateTime.Now.Year.ToString()}),
    New RouteValueDictionary(New With _
      {.locale = "[a-z]{2}-[a-z]{2}", .year = "¥d{4}"}))
End Sub
```

<C#>

```
public static void RegisterRoutes(RouteCollection routes)
{
  routes.MapPageRoute("DetailRoute",
    "Detail/{locale}/{year}",
    "~/detail.aspx",
    true,
    new RouteValueDictionary
      {{"locale", "en-US"}, {"year", DateTime.Now.Year.ToString()}},
    new RouteValueDictionary
      {{"locale", "[a-z]{2}-[a-z]{2}"}, {"year", @"¥d{4}"}}
  );
}
```

ルーティングによって URL 要求が処理されるとき、前の例のルート定義から、次の表に示す結果が生成されます。

表. 結果

URL	結果
/en-US	一致せず。*locale* と *year* の両方が必要です。

/en-US/08	一致せず。*year* の制約により、4 桁の数字が必要です。
/en-US/2008	*locale* = "en-US" *year* = "2008"

ルーティングが適用されない場合のシナリオ

URL ルーティングでは、Web サイトで有効な要求であっても処理されないことがあります。

URL パターンに一致する物理ファイルが見つかる場合

既定では、ルーティングは、Web サーバー上の既存の物理ファイルにマップされる要求は処理しません。たとえば、http://server/application/Products/Beverages/Coffee.aspx という要求は、Products/Beverages/Coffee.aspx という物理ファイルが存在すると、ルーティングによって処理されることはありません。この場合、定義されているパターン ({controller}/{action}/{id} など) に要求が一致しても、ルーティングによる処理は行われません。

要求がファイルを指す場合も含めて、すべての要求をルーティングで処理するには、**RouteCollection** オブジェクトの **RouteExistingFiles** プロパティを true に設定して、既定の動作を上書きします。この値を true に設定すると、定義されているパターンに一致するすべての要求がルーティングによって処理されます。

URL パターンのルーティングを明示的に無効化する場合

特定の URL 要求がルーティングで処理されないように指定することもできます。特定の要求がルーティングで処理されることを防ぐには、ルートを定義し、**StopRoutingHandler** クラスを使用してそのパターンを処理するように指定します。StopRoutingHandler オブジェクトによって要求が処理される場合、StopRoutingHandler オブジェクトは、要求がそれ以上ルートとして処理されるのをブロックします。代わりに、その要求は、ASP.NET ページ、Web サービス、またはその他の ASP.NET エンドポイントとして処理されます。**RouteCollection.Ignore** メソッド (MVC アプリケーションでは RouteCollectionExtensions.IgnoreRoute) を使用して、StopRoutingHandler クラスを使用するルートを作成できます。次の例は、WebResource.axd ファイルへの要求がルーティングによって処理されるのを防ぐ方法を示しています。

<Visual Basic>

```
Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
  routes.Ignore ("{resource}.axd/{*pathInfo}")
End Sub
```

<C#>

```
public static void RegisterRoutes(RouteCollection routes)
{
```

```
  routes.Ignore("{resource}.axd/{*pathInfo}");
}
```

URL とルートの照合

ルーティングは URL 要求を処理するとき、要求の URL をルートに一致させようと試みます。URL 要求をルートに一致させる操作は、次のすべての条件に基づいて決まります。

- ユーザーが定義したルートパターン、またはプロジェクトの種類に含まれている既定のルートパターン (存在する場合)
- Routes コレクションに追加した順序
- ルートに指定した既定値
- ルートに指定した制約
- 物理ファイルに一致する要求を処理するようにルーティングが定義されているかどうか

不適切なハンドラによって要求が処理されることを避けるために、ルートを定義する際は、これらすべての条件を考慮する必要があります。Routes コレクション内での Route オブジェクトの順序は重要です。ルートの照合は、コレクション内の最初のルートから最後のルートへの順に試行されます。一致が見つかると、それ以降のルートは評価されません。通常、Routes プロパティにルートを追加する際は、最も特定性の高いルート定義から最も特定性の低いルート定義の順に追加します。

たとえば、次のパターンでルートを追加するとします。

- **ルート 1** —— {controller}/{action}/{id} に設定
- **ルート 2** —— products/show/{id} に設定

最初にルート 1 が評価されますが、ルート 2 に一致する要求はすべてルート 1 にも一致するため、ルート 2 による要求の処理は行われません。http://server/application/products/show/bikes への要求は、ルート 2 の方が一致度が高いと考えられますが、ルート 1 によって次の値として処理されます。

- controller —— products
- action —— show
- id —— bikes

要求にパラメーターが含まれていない場合は、既定値が使用されます。その結果、ルートが予想外の要求に一致する場合があります。たとえば、次のパターンでルートを追加するとします。

- Route 1: {report}/{year}/{month} (year と month に既定値を設定)
- Route 2: {report}/{year} (year に既定値を設定)

この場合、ルート 2 で要求が処理されることはありません。ルート 1 は月次レポートを想定し、ルート 2 は年次レポートを想定しているとします。ルート 1 に既定値を設定すると、ルート 2 に一致するすべての要求はルート 1 にも一致するようになります。

annual/{report}/{year} や monthly/{report}/{year}/{month} のように定数を含めると、パターンのあいまいさをなくすことができます。

RouteTable コレクションに定義されている、どの Route オブジェクトにも URL が一致しない場合、その要求は ASP.NET ルーティングでは処理されません。代わりに、ASP.NET ページ、Web サービス、またはその他の ASP.NET エンドポイントに処理が渡されます。

ルートからの URL の作成

サイト内のページへのハイパーリンクを作成する場合は、URL パターンを使用することにより、ルートに対応する URL をプログラムで作成することができます。パターンを変更すると、新しいパターンに一致する URL が自動的に生成されます。

ルーティングされたページ内の URL パラメーターへのアクセス

ルーティングされたページ要求のハンドラでは、コードまたはマークアップを使用して、URL プレースホルダーで渡された値にアクセスすることができます。

MVC アプリケーションでは、URL プレースホルダーで渡された値は MVC フレームワークによって自動的に処理されます。

URL ルーティングとセキュリティ

認証規則は、ルート URL にのみ適用することも、ルート URL とそのマップ先である物理的な URL の両方に適用することもできます。たとえば、認証規則では、すべてのユーザーが Category から始まる URL にアクセスできるように指定する一方で、管理者のみが Categories.aspx ページにアクセスできるように指定することもできます。ルート URL パターン "contoso.com/Category/{controller}/{action}" が物理的な URL "contoso.com/Categoriespage.aspx" にマップされており、認証規則はルート URL にのみ適用されているとします。この場合、ルート URL を使用して Categoriespage.aspx を要求したユーザーには、そのファイルへのアクセスが常に許可されます。ただし、物理的な URL を使用して Categoriespage.aspx を要求した場合、そのファイルへのアクセスは管理者しか許可されません。

既定では、認証規則はルート URL と物理的な URL の両方に適用されます。

クラス参照

URL ルーティングの主要なサーバークラスを次の表に示します。

表. ASP.NET ルーティングの主要なサーバークラス

クラス	説明
Route	Web フォームまたは MVC アプリケーションでルートを表します。

DynamicDataRoute	Dynamic Data アプリケーションでルートを表します。
RouteBase	ASP.NET ルートを表す全クラスの基本クラスとして機能します。
RouteTable	アプリケーションのルートを格納します。
RouteCollection	ルートのコレクションの管理に使用できるメソッドを提供します。
RouteCollectionExtensions	MVC アプリケーションでルートのコレクションの管理に使用できる追加メソッドを提供します。
RouteData	要求されたルートの値を格納します。
RequestContext	ルートに対応する HTTP 要求についての情報を格納します。
StopRoutingHandler	ASP.NET ルーティングで URL パターンの要求が処理されないように指定することができます。
PageRouteHandler	Web フォーム アプリケーションのルートを定義することができます。
RouteValueDictionary	ルートの Constraints、Defaults、および DataTokens の各プロジェクトを格納できます。
VirtualPathData	ルート情報から URL を生成することができます。

URL ルーティングと URL の書き換え

URL ルーティングは、**URL の書き換え**とは異なります。URL の書き換えでは、受信要求を Web ページに送信する前に、URL が実際に変更されます。たとえば、URL の書き換えを使用するアプリケーションでは、/Products/Widgets/ という URL が /Products.aspx?id=4 に変更されることがあります。さらに、通常、URL の書き換えでは、パターンに基づいて URL を作成するための API は用意されません。URL の書き換えの場合、URL パターンを変更するには、元の URL を含むすべてのハイパーリンクを手動で更新する必要があります。

URL ルーティングでは、URL から値を抽出できるので、受信要求が処理されるときに URL は変更されません。URL を作成する必要がある場合は、URL を生成するメソッドにパラメーター値を渡します。URL パターンを変更する際は、1 か所を変更するだけで、そのパターンに基づいてアプリケーション内で作成されるすべてのリンクに、新しいパターンが自動的に反映されます。

Dynamic Data の概要

ASP.NET Dynamic Data は、データ駆動の ASP.NET Web アプリケーションの迅速な作成を可能にする**スキャフォールディング フレームワーク**です。実行時にデータ モデル メタデータを自動的に検出し、それに基づいて UI 動作を派生することで、この動作が可能になります。

スキャフォールディングフレームワークは、要素を変更したり、既定の動作をオーバーライドして新しい要素を作成したりすることで簡単にカスタマイズできます。また、既存のアプリケーションは、スキャフォールディング要素を ASP.NET ページに簡単に統合できます。

Dynamic Data の機能

ASP.NET Dynamic Data を使用すると、最小限のコードで Dynamic Data フレームワークの機能を最大限に活用する新しい**データ駆動 Web サイト**を作成できます。特定の Dynamic Data 機能を選択して既存の Web サイトに追加することもできます。

Dynamic Data は、次の機能を提供します。

- 基になるデータベーススキーマを読み取って Web アプリケーションを実行できる Web スキャフォールディング。Dynamic Data スキャフォールディングを使用すると、データモデルから標準的な UI を生成できます。
- 完全なデータアクセス操作 (作成、更新、削除、表示)、関係演算、およびデータ検証ができます。
- 外部キーリレーションシップの自動サポート。Dynamic Data は、テーブル間のリレーションシップを検出し、ユーザーが関連テーブルのデータを容易に表示できるようにする UI を作成します。
- 特定のデータフィールドを表示および編集するための UI のカスタマイズができます。
- 特定のテーブルのデータフィールドを表示および編集するための UI のカスタマイズができます。
- データフィールドの検証のカスタマイズ。プレゼンテーション層を使用することなく、ビジネスロジックをデータ層に保持できます。

Dynamic Data の特性

Dynamic Data は、実行時に基になるデータベーススキーマからデータフィールドの外観と動作を判断できるという動的な特性を持ちます。この機構に加えて、用意されている既定のページテンプレートやフィールドテンプレート、柔軟なカスタマイズ性により、さまざまなデザイン上の選択肢が提供されます。

次に例を示します。

- スキャフォールディングを使用した Web サイトの構築
- 既存の Web サイトへの動的データの追加

- データフィールド検証ビジネスロジックの追加
- 特定のデータフィールドまたは特定のテーブルを表示および編集するための UI のカスタマイズ

データ モデル

データ モデルは、データベースに含まれている情報と、データベース内の項目がどのように関連付けられているかを表します。

Dynamic Data では、**LINQ to SQL** データモデルと **ADO.NET Entity Framework** データモデルがサポートされています。1 つの Web アプリケーションには、データモデルの複数のインスタンスを含めることができますが、Dynamic Data で使用されるモデルの種類は同一である必要があります。

Dynamic Data で使用するデータモデルは、Web アプリケーションの Global.asax ファイルに登録します。Dynamic Data にデータモデルを登録すると、データフィールドの自動検証を実行でき、データ層レベルでデータの外観と動作を制御することが可能になります。

スキャフォールディング

スキャフォールディングは、データモデルに基づいてページを動的に表示することで、既存の ASP.NET ページフレームワークを強化する機構です。スキャフォールディングには、次の利点があります。

- 最小限のコードまたはコードなしでデータ駆動 Web アプリケーションを作成できる
- 短時間で開発できる
- データベーススキーマに基づく組み込みのデータ検証を利用できる
- それぞれの外部キーまたはブール値フィールドに対して自動的にデータが選択される

<ページ テンプレート>

Dynamic Data スキャフォールディングでは、ページテンプレートを使用してデータテーブルの既定のビューを提供します。ページテンプレートは、Dynamic Data で利用できる任意のテーブルのデータを表示するように構成された ASP.NET Web ページです。Dynamic Data には、テーブルの一覧表示 (リスト ビュー)、マスター/詳細テーブルの表示 (詳細ビュー)、データの編集 (編集ビュー) など、データの異なるビューのためのページテンプレートが用意されており、既定ではリストビューページテンプレートのみが使用するように構成されています。

また、既定のページテンプレートを変更したり、目的によってページテンプレートを使い分けられるように Dynamic Data を変更したりすることもできます。

<フィールド テンプレート>

Dynamic Data では、フィールドテンプレートを使用して、個々のデータフィールドを表示および編集するための UI をレンダリングします。適切なフィールドテンプレートは、データフィールドの型に基づいて決定されます。Dynamic Data には、データフィールドの表示および編集のための個別のフィールドテンプレートが含まれています。

たとえば、**DateTime** データフィールドには、次のフィールドテンプレートが使用されます。

- **DateTime.ascx** フィールド テンプレートは、**DateTime** データ型をテキスト (文字列) として表示し、**Literal** コントロールとしてレンダリングします。
- **DateTime_Edit.ascx** フィールド テンプレートは、**TextBox** コントロールをレンダリングします。データベース内のフィールドを null にすることができない場合、または入力を要求するようにデータモデルがカスタマイズされている場合は、**RequiredFieldValidator** コントロールもレンダリングされます。また、このテンプレートは、データモデルからスローされたあらゆる例外を処理する **DynamicValidator** コントロールを備えています。Regex クラスもサポートされます。

Dynamic Data Web プロジェクトを作成すると、Visual Studio によって、**DynamicData¥FieldTemplates** フォルダがプロジェクトに追加されます。このフォルダには、既定のフィールドテンプレートが格納されます。

個々のデータフィールドをどのようにレンダリングするかを指定するために、組み込みのフィールド テンプレートをカスタマイズしたり、新しいフィールドテンプレートを作成したりできます。たとえば、電話番号や電子メールアドレスの表示と編集を行う UI をレンダリングするフィールドテンプレートを作成できます。または、別の方法 (スライダなど) でユーザーが数値データを指定できるようにする UI をレンダリングするフィールドテンプレートを作成することもできます。

既存のデータ コントロールの拡張

Dynamic Data は、次のように既存のデータ コントロールを拡張して、動的な動作を実現します。

- **DetailsView** コントロールと **GridView** コントロールでは、定義済みの Dynamic Data テンプレートを使用してデータを動的に表示できます。各ページのデータコントロールに対して、同じマークアップとコードを何度も作成する必要はありません。これらのテンプレートをカスタマイズして、使用するコントロールや、データフィールドの表示用と編集用の UI のレンダリング方法を変更できます。1 か所に変更を加えるだけで、Web アプリケーション全体のデータコントロールの外観や動作に反映させることができます。これは特定のテーブルに関連付けられていないため、データベース内のどのテーブルでもページテンプレートを使用できます。
- **FormView** コントロールと **ListView** コントロールでは、DetailsView コントロールや GridView コントロールに似た動作を実装できます。そのためには、テンプレート内で DynamicControl コントロールを使用し、行のどのフィールドを表示するかを指定します。Dynamic Data は、指定されたテンプレートに基づいて、これらのコントロールの UI を自動的

に作成します。DynamicControl コントロールはフィールドの UI を自動的にはレンダリングしないため、コントロールを特定のデータフィールドにバインドする必要があります。

- Dynamic Data は、**LINQ to SQL** データモデルまたは **Entity Framework** データモデルのデータモデルメタデータを調べ、メタデータに基づいて自動検証を提供します。たとえば、データベースのある列が null 非許容とマークされている場合、その列には自動的に **RequiredFieldValidator** コントロールがレンダリングされます。データフィールドのレンダリング方法や検証方法をさらにカスタマイズするために、カスタムメタデータを適用することもできます。

クラス リファレンス

ASP.NET Dynamic Data クラスを含む名前空間の一覧を次の表に示します。

表. 名前空間の一覧

名前空間	説明
System.ComponentModel.DataAnnotations	Dynamic Data コントロールのメタデータを定義するために使用される属性クラスを提供します。
System.Web.DynamicData	ASP.NET Dynamic Data のコア機能を提供するクラスが含まれます。Dynamic Data の動作をカスタマイズするための機能拡張も用意されています。

Dynamic Data のガイドライン

ASP.NET Dynamic Data では、ニーズに合ったカスタマイズのレベルを選択できます。ここでは、タスクの実行に役立つガイドラインおよび提案事項を紹介します。

Dynamic Data を使用すると、完全な動的機能を持つ完成された Web アプリケーションをすばやく作成できます。さらに、簡単にデータベースを Web サイトに統合したり、必要な特定の動的機能を選択したりできます。また、幅広いカスタマイズも可能で、プレゼンテーション層からデータ層のニーズに Web サイトを適合させることができます。

スキャフォールディングを使用した Web サイトの作成

Dynamic Data は、Web スキャフォールディングをサポートしています。これにより、データモデルに基づくアプリケーションを最小限のコードで実行でき、目標を達成できます。この Web スキャフォールディングは標準の UI を持ちますが、テーブルに対する作成 (Create)、読み取り (Read)、更新 (Update)、および削除 (Delete) という CRUD 操作に対応しています。さらに、リレーションシップが完全にサポートされます。スキャフォールディングを使用して基本的なアプリケーションを作成し、後で適切なカスタマイズを適用できます。

既存の Web サイトへの動的データの追加

ASP.NET Dynamic Data では、必要な特定の動的機能を選択することで、動的データコントロール動作を既存の Web サイトに統合できます。そのためには、次の手順を実行する必要があります。

- Dynamic Data Web サイトを作成します。
- フィールドテンプレートを作成します。
- データモデルをカスタマイズします。テーブルを表示するカスタムページを作成します。
- 動的な動作を持つデータコントロールを使用します。

検証属性の使用

System.ComponentModel.DataAnnotations 属性を使用してメタデータをデータモデルのデータフィールドに適用することで、追加の情報を Dynamic Data に提供できます。Dynamic Data は、この情報を使用して、たとえばデータフィールドの表示および編集用の UI のレンダリング方法をカスタマイズすることができます。

検証属性を適用する場合は、次の使用上の制約に従う必要があります。

- 属性は、プロパティまたはフィールドに適用できます。
- 属性は 1 回しか適用できません。

検証属性の適用

任意の System.ComponentModel.DataAnnotations 属性をデータフィールドに適用する場合に従う必要がある手順を次に示します。

- Web アプリケーションの **App_Code** フォルダに、データコンテキスト部分クラスを含むクラスファイルを実装します。このクラスは、属性を適用するデータフィールドが含まれているテーブルを表します。
- **関連メタデータ クラス**として動作する別のクラスを作成します。このクラスには任意の名前を使用できますが、クラス名は、次に説明するように、部分クラスに適用される **MetadataTypeAttribute** 属性内で参照する名前と一致している必要があります。前の手順で作成したクラスと同じクラスファイルにこのクラスを入れます。
- 関連メタデータクラス内で、パブリックプロパティまたはフィールドを作成し、検証属性を適用するデータフィールドと同じ名前を付けます。
- 部分クラス定義に MetadataTypeAttribute 属性を適用します。属性のパラメータは、関連メタデータクラスの名前です。

検証エラーの生成

System.ComponentModel.DataAnnotations 属性を使用すると、検証が失敗したときにカスタムエラーを作成したり、組み込みのエラーを使用したりできます。このため、検証属性は、次のいずれかの名前付きエラーパラメータを受け取ることができます。

- **ErrorMessage** パラメータは、検証コントロールに関連付けられているエラーメッセージを指定します。このパラメータを使用すると、ローカライズされないカスタムエラーメッセージを指定し、既定のローカライズ可能なメッセージを (多くの場合) オーバーライドできます。
- **ErrorMessageResourceName** パラメータは、検証コントロールに関連付けられているエラーメッセージリソースを指定します。このパラメータを使用して、ローカライズされないエラーメッセージが含まれているリソースファイルを指定します。
- **ErrorMessageResourceType** パラメータは、検証コントロールに関連付けられているエラーメッセージの種類を指定します。このパラメータを使用して、リソース ファイルに定義されているエラーメッセージを識別します。リソースファイルは、前のパラメータを使用して指定します。

検証属性エラーメッセージを使用する場合、次のような選択肢があります。

- 必ずローカライズされる既定のエラーメッセージを使用できます。この場合は、前に示したどのパラメータも指定する必要はありません。
- **ErrorMessage** パラメータを使用して、既定のメッセージをオーバーライドする、ローカライズされないカスタムエラーメッセージを提供できます。
- **ErrorMessageResourceName** パラメータを使用して、ローカライズされないリソースエラーメッセージファイルを提供できます。次に、ErrorMessageResourceType パラメータを使用して、リソースファイルに含まれているエラーメッセージを指定します。

データ モデルの選択

LINQ to SQL と Entity Framework には共通する機能が多くありますが、それぞれ異なるシナリオを対象にした機能を持ちます。LINQ to SQL は、既存の Microsoft SQL Server スキーマに対してアプリケーションを迅速に開発することを主な目標としています。Entity Framework は、既存のリレーショナルスキーマへの疎結合の柔軟なマッピングを介して、Microsoft SQL Server およびサードパーティのデータベースへのオブジェクトおよびストレージ層アクセスを提供します。

LINQ to SQL

LINQ to SQL は、Microsoft SQL Server データベースを対象とする機能を備えています。既存のデータベーススキーマの厳密に型指定されたビューを実現します。

LINQ to SQL は、既存のデータベース スキーマと .NET Framework クラスとの直接の一対一のマッピングをサポートしています。1 つのテーブルは 1 つのクラスにマップでき、外部キーは厳密に型指定されたリレーションシップとして公開できます。

テーブル、ビュー、テーブル値関数に対する LINQ クエリを作成し、厳密に型指定されたオブジェクトとして結果を返し、厳密に型指定されたメソッドを介して厳密に型指定された結果を返すストアドプロシージャを呼び出すことができます。LINQ to SQL の重要なデザイン上の原則は、ほとんどの一般的なケースに対応することです。したがって、たとえば、顧客の Orders プロパティを介して注文のコレクションにアクセスしたときにその顧客の注文が以前に取得されていない場合、LINQ to SQL は注文を自動的に取得します。

ADO.NET Entity Framework

ADO.NET Entity Framework は、エンタープライズシナリオを対象とする機能を備えています。通常、企業のデータベースは、データベース管理者によって管理されています。このスキーマは、通常、優れたアプリケーションモデルを示す代わりに、ストレージに関する検討事項 (パフォーマンス、一貫性、パーティション分割) に最適化され、使用データや使用パターンがやがて変化するのに合わせて変更されます。

Entity Framework は、疎結合のアプリケーション指向のデータモデルの公開を対象に設計されており、既存のデータベーススキーマとは大きく異なります。

たとえば、単一のクラス (またはエンティティ) を複数のテーブル／ビューにマップしたり、複数のクラスを同じテーブル/ビューにマップしたりできます。継承階層を (LINQ to SQL の場合と同様に) 単一のテーブル/ビューや複数のテーブル／ビューにマップできます。この柔軟なマッピングは、時間の経過と共に変化するデータベースのスキーマに、アプリケーションを再コンパイルする必要なく対応するために、宣言的に指定されます。

Entity Framework には、アプリケーションデータモデルに対して多くの LINQ to SQL と同じ機能を公開する LINQ to Entities が含まれます。

ASP.NET AJAX の概要

ASP.NET の **AJAX 機能**を使用することにより、応答性に優れたユーザー インターフェイス (UI) を持つ Web ページをすばやく作成できます。

AJAX 機能には、ブラウザー間の **ECMAScript (JavaScript) テクノロジ**と**ダイナミック HTML (DHTML) テクノロジ**を組み込んだクライアントスクリプトライブラリ、ASP.NET サーバーベース開発プラットフォームとの統合機能などが含まれます。

ASP.NET AJAX 機能を使用する理由

ASP.NET の AJAX 機能を使用すると、サーバーベースの Web アプリケーションよりも多くの利点を備えた高機能 Web アプリケーションを構築できます。AJAX 対応アプリケーションは、次の機能を備えています。

- **効率の向上** —— これは、Web ページの処理の重要な部分がブラウザーで実行されるためです。
- **使い慣れた UI 要素** —— プログレス インジケータ、ツールヒント、ポップアップ ウィンドウなどがあります。
- **部分ページ更新** —— Web ページの変更された部分だけを更新します。
- **ASP.NET アプリケーションサービスとのクライアント統合** —— フォーム認証、ロール、およびユーザープロファイルに使用されます。
- **自動生成されたプロキシクラス** —— クライアントスクリプトからの Web サービスメソッドの呼び出しを簡略化します。
- **サーバーコントロールをカスタマイズできるフレームワーク** —— クライアント機能を追加するために使用されます。
- **一般に普及しているブラウザーのサポート** —— Microsoft Internet Explorer、Mozilla Firefox、Apple Safari などがサポートされています。

ASP.NET における AJAX 機能のアーキテクチャ

ASP.NET の AJAX 機能のアーキテクチャは、クライアントスクリプトライブラリとサーバーコンポーネントで構成されます。この 2 つの要素は堅牢な開発フレームワークを提供するために組み込まれています。

また、これらに加えて、ASP.NET AJAX Control Toolkit という、リッチな AJAX コントロールを提供する追加コンポーネントも使用できます。

クライアントスクリプトライブラリとサーバーコンポーネントに含まれる機能を次の図に示します。

図. ASP.NET AJAX サーバーおよびクライアントのアーキテクチャ

この図は、クライアントベースの Microsoft AJAX Library の機能を示しています。次に、この図にあるそれぞれの機能について詳しく説明します。

AJAX クライアント アーキテクチャ

クライアントアーキテクチャには、コンポーネントのサポート、ブラウザーの互換性、ネットワーク、およびコアサービス用のライブラリが含まれています。

コンポーネント

クライアント コンポーネントを使用すると、ポストバックを使用せずに、ブラウザーでさまざまな動作を行うことが可能になります。コンポーネントは 3 つのカテゴリに分類されます。

- **コンポーネント** —— コードをカプセル化する、タイマーオブジェクトなどの非ビジュアルオブジェクトです。
- **ビヘイビア** —— 既存の DOM 要素の基本動作を拡張します。
- **コントロール** —— カスタムの動作を持つ新しい DOM 要素を表します。

使用するコンポーネントの種類は、必要なクライアント動作の種類によって異なります。たとえば、既存のテキストボックスの透かしは、テキストボックスに関連付けられたビヘイビアを使用して作成できます。

ブラウザーの互換性

ブラウザーの互換性レイヤには、最も頻繁に使用されるブラウザー（Microsoft Internet Explorer、Mozilla Firefox、Apple Safari など）に対する AJAX スクリプト互換性が用意されています。このため、サポート対象であれば、どのブラウザーに対しても同じスクリプトを記述できます。

ネットワーキング

ネットワーク レイヤは、ブラウザー内のスクリプトと Web ベースのサービスおよびアプリケーションの間の通信を処理します。非同期のリモートメソッド呼び出しも管理します。UpdatePanel コントロールを使用する部分ページ更新など、一般的なシナリオの多くでは、ネットワーク レイヤは自動的に使用され、コードを記述する必要はありません。

ネットワークレイヤは、サーバーベースのフォーム認証、ロール情報、およびクライアントスクリプト内のプロファイル情報へのアクセスもサポートしています。Web アプリケーションが Microsoft AJAX Library にアクセスできる限り、ASP.NET を使用して作成されていないアプリケーションでもこのサポートを利用できます。

コア サービス

ASP.NET の AJAX クライアントスクリプトライブラリは、オブジェクト指向開発向けの機能を提供する JavaScript (.js) ファイルで構成されます。ASP.NET AJAX クライアントスクリプトライブラリに含まれるオブジェクト指向機能により、クライアントスクリプトの高度な一貫性とモジュール性が実現されます。次の**コアサービス**は、クライアントアーキテクチャの一部です。

- **JavaScript のオブジェクト指向拡張機能** —— クラス、名前空間、イベント処理、継承、データ型、オブジェクトのシリアル化などがあります。
- **基本クラス ライブラリ** —— 文字列ビルダ、拡張エラー処理などのコンポーネントが含まれます。
- **JavaScript ライブラリのサポート** —— JavaScript ライブラリは、アセンブリに埋め込まれるか、スタンドアロンの JavaScript (.js) ファイルとして提供されます。JavaScript ライブラリをアセンブリに埋め込むと、アプリケーションの配置が容易になり、バージョン管理の問題の解決に役立ちます。

<デバッグとエラー処理>

コアサービスには **Sys.Debug** クラスが含まれます。このクラスは、Web ページの末尾に読み取り可能な形式でオブジェクトを表示するためのメソッドを提供します。また、トレースメッセージも表示され、ユーザーはアサーションを使用し、デバッガを中断できます。拡張された **Error** オブジェクト API により、リリースモードとデバッグモードをサポートする有用な例外詳細が提供されます。

<グローバリゼーション>

ASP.NET の AJAX サーバーおよびクライアントのアーキテクチャには、クライアントスクリプトをローカライズおよびグローバライズするためのモデルが用意されています。これにより、単一のコードベースを使用するアプリケーションをデザインし、多くのロケール (言語およびカルチャ) に UI を提供できます。たとえば、AJAX アーキテクチャを使用すると、JavaScript コードはサーバーへのポストバックを行わなくても、ユーザーのブラウザーのカルチャ設定に応じて **Date** オブジェクトまたは **Number** オブジェクトの書式を自動的に指定できます。

AJAX サーバー アーキテクチャ

AJAX 開発をサポートするサーバー部分は、アプリケーションの UI とフローを管理する ASP.NET Web サーバーコントロールおよびコンポーネントで構成されます。サーバー部分は、シリアル化、検証、コントロールの拡張機能なども管理します。フォーム認証、ロール、およびユーザー プロファイル用の ASP.NET アプリケーションサービスにアクセスできる ASP.NET Web サービスもあります。

スクリプト サポート

ASP.NET の AJAX 機能は、サーバーからクライアントに送信される**サポートスクリプト**を使用することで実装されます。ブラウザーに送信されるスクリプトは、有効にする AJAX 機能によって異なります。

ASP.NET アプリケーション用のカスタムクライアントスクリプトを作成することもできます。その場合、AJAX 機能を使用して、カスタムスクリプトを静的な .js ファイル (ディスク上) またはアセンブリ内にリソースとして埋め込まれた .js ファイルとして管理できます。

ASP.NET AJAX 機能には、リリースモードとデバッグモード用のモデルがあります。

リリースモードでは、エラーチェックとパフォーマンスを最適化するための例外処理が行われ、スクリプトサイズは最小限に抑えられます。デバッグモードには、型チェックや引数チェックなど、より厳密なデバッグ機能が用意されています。

アプリケーションがデバッグモードの場合、ASP.NET はデバッグバージョンを実行します。これによりデバッグスクリプトで例外をスローできますが、リリースコードのサイズは最小限に抑えられます。

ASP.NET の AJAX のスクリプトサポートを使用して、2 つの重要な機能が提供されます。

- Microsoft AJAX Library は型システムであり、名前空間、継承、インターフェイス、列挙型、リフレクション、およびその他の機能を提供する一連の JavaScript 拡張機能です。
- 部分ページレンダリングは、非同期ポストバックを使用してページの領域を更新します。

<ローカリゼーション>

ASP.NET AJAX アーキテクチャは、**ASP.NET 2.0 ローカリゼーションモデル**の基盤に基づいています。アセンブリ内に埋め込まれる、またはディスク上にあるローカライズされた .js ファイルの追加サポートを提供します。ASP.NET は、特定の言語および地域にローカライズされたクライアントスクリプトとリソースを自動的に提供できます。

Web サービス

ASP.NET Web ページの AJAX 機能により、クライアントスクリプトを使用して、**ASP.NET Web サービス (.asmx)** と **WCF (Windows Communication Foundation) サービス (.svc)** の両方を呼び出すことができます。必要なスクリプト参照はページに自動的に追加され、クライアントスクリプトを使って Web サービスを呼び出すための Web サービスプロキシクラスが自動的に生成されます。

ASP.NET AJAX サーバーコントロールを使用せずに、ASP.NET Web サービスにアクセスすることもできます (別の Web 開発環境を使用している場合など)。そのためには、ページで Microsoft AJAX Library、スクリプトファイル、および Web サービス自体への参照を手動で追加します。実行時に、ASP.NET により、サービスの呼び出しに使用できるプロキシクラスが生成されます。

アプリケーション サービス

ASP.NET の**アプリケーションサービス**は、ASP.NET フォーム認証、ロール、およびユーザープロファイルに基づく組み込みの Web サービスです。これらのサービスは、AJAX 対応の Web ページ内のクライアントスクリプト、Windows クライアントアプリケーション、または WCF 互換のクライアントによって呼び出すことができます。

サーバー コントロール

ASP.NET AJAX サーバー コントロールは、リッチクライアント動作を作成するために組み込まれるサーバーおよびクライアントコードで構成されます。ASP.NET Web ページに AJAX コントロールを追加すると、AJAX 機能のブラウザーにサポートクライアントスクリプトが自動的に送信されます。コントロールの機能をカスタマイズするクライアントコードを追加することもできますが、これは必須ではありません。

最も頻繁に使用される ASP.NET AJAX サーバーコントロールを次の表に示します。

表. ASP.NET AJAX サーバー コントロール

コントロール	説明
ScriptManager	クライアントコンポーネント、部分ページレンダリング、ローカリゼーション、グローバリゼーション、およびカスタムユーザースクリプトのスクリプトリソースを管理します。ScriptManager コントロールは、UpdatePanel、

	UpdateProgress、および Timer の各コントロールを使用するために必要です。
UpdatePanel	同期ポストバックを使用してページ全体を最新の情報に更新するのではなく、ページの選択した部分を最新の情報に更新できます。
UpdateProgress	UpdatePanel コントロールでの部分ページ更新に関するステータス情報を表示します。
Timer	定義された間隔でポストバックを実行します。Timer コントロールを使用してページ全体をポストするか、UpdatePanel コントロールと共に使用して、定義された間隔で部分ページ更新を実行します。

AJAX クライアント動作を含む、カスタム ASP.NET サーバーコントロールを作成することもできます。他の ASP.NET Web コントロールの機能を強化するカスタムコントロールは、エクステンダコントロールと呼ばれます。

ASP.NET AJAX Control Toolkit

ASP.NET AJAX Control Toolkit は、ASP.NET AJAX コントロールとエクステンダを使用して作成できるエクスペリエンスを示す、サンプルとコンポーネントのコレクションです。この Control Toolkit には、カスタム コントロールおよびエクステンダを簡単に作成して再利用できるようにするサンプルと強力な SDK が用意されています。

ASP.NET AJAX Control Toolkit は、ASP.NET AJAX の Web サイト (http://www.asp.net/ajax/) からダウンロード可能で、サポートはコミュニティで行われています。

部分ページ レンダリングの概要

部分ページレンダリングにより、ポストバックの結果としてページ全体ではなく、変更された個々のページ領域のみが更新されます。

つまり、ポストバックのたびにページ全体の再読み込みが表示されることがなくなり、Web ページとのやり取りがよりシームレスになります。

ASP.NET を使用すると、クライアントスクリプトを記述しなくても、新しい ASP.NET Web ページおよび既存の ASP.NET Web ページに部分ページ レンダリングを追加できます。

部分ページ レンダリングの目的

既存の ASP.NET アプリケーションを拡張して、**AJAX (Asynchronous JavaScript and XML) 機能**が組み込まれた新しいアプリケーションを開発できます。以下の目的で、AJAX 機能を使用します。

- 機能が豊富で、ユーザーの操作への応答が良く、従来からのクライアントアプリケーションと同様に動作する Web ページを開発することにより、ユーザーエクスペリエンスを向上させるため。
- ページ全体の再表示を削減し、ページのちらつきを回避するため。
- クライアントスクリプトの作成を排除し、ブラウザー間に互換性を持たせるため。
- クライアントスクリプトを記述の作成を排除し、AJAX スタイルのクライアント/サーバー通信を実行するため。
- ASP.NET AJAX Control Toolkit のコントロールおよびコンポーネントを使用するため。
- カスタムの ASP.NET AJAX コントロールを開発するため。

部分ページ レンダリングの機能

部分ページレンダリングは、**ASP.NET のサーバーコントロールおよび Microsoft AJAX Library のクライアント機能**に依存しています。部分ページレンダリングを有効にするために、Microsoft AJAX Library を使用する必要はありません。この機能は、ASP.NET AJAX サーバーコントロールを使用するときに自動的に提供されるためです。ただし、クライアントライブラリに公開されている API を使用すると、追加の AJAX 機能を利用できます。

部分ページレンダリングをサポートする ASP.NET の主な機能は次のとおりです。

- ASP.NET サーバーコントロールに類似した動作の宣言モデル。多くの場合、宣言マークアップを使用するのみで、部分ページレンダリングを指定できます。
- 部分ページ更新に必要な基になるタスクを実行するサーバーコントロール。これには、**ScriptManager** コントロールおよび **UpdatePanel** コントロールが含まれます。
- 共通タスクのための、ASP.NET AJAX サーバーコントロールと Microsoft AJAX Library との統合。このタスクには、ユーザーがポストバックを取り消すことができるようにすること、非同期のポストバック中にカスタムの進捗メッセージを表示すること、および同時に実行されている非同期ポストバックの処理方法を決定することが含まれます。
- 部分ページレンダリングのエラー処理オプション。これにより、ブラウザーでのエラーの表示方法をカスタマイズできます。
- ASP.NET の AJAX 機能に組み込まれている、ブラウザー間での互換性。サーバーコントロールを使用するだけで、自動的に正しいブラウザー機能が起動されます。

部分ページ レンダリングの背景

ASP.NET Web サーバーコントロールを使用して構築された通常の Web ページでは、ページでのユーザーによる操作 (ボタンをクリックするなど) で開始されたポストバックを実行します。その応答として、サーバーが新しいページを表示します。この場合、ポストバック間で変更されなかったコントロールおよびテキストも頻繁に再表示されます。

部分ページレンダリングを使用すると、ページの個々の領域を非同期に再表示できるため、ユーザーへのページの応答が速くなります。

部分ページレンダリングは **ASP.NET AJAX Web サーバーコントロール**を使用すると実装できます。オプションで、Microsoft AJAX Library 内の API を使用するクライアントスクリプトを記述することもできます。

部分ページ更新用のサーバー コントロール

ASP.NET Web ページに AJAX 機能を追加するには、ページ内の更新する個々のセクションを指定します。次に、そのセクションのコンテンツを **UpdatePanel** コントロールに含めます。UpdatePanel コントロールのコンテンツは、HTML でも、その他の ASP.NET コントロールでもかまいません。UpdatePanel コントロールは、他のコントロールと同様にページに追加できます。たとえば、Visual Studio で、ツールボックスから Web ページにドラッグすることも、ページの宣言マークアップを使用して追加することもできます。

UpdatePanel コントロールのマークアップの例を次に示します。

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
    <ContentTemplate>
        <!-- Place updatable markup and controls here. -->
    </ContentTemplate>
</asp:UpdatePanel>
```

既定では、更新パネル内のコントロール (子コントロール) から発生するポストバックは、自動的に非同期のポストバックを開始し、部分ページ更新を実行します。更新パネル外のコントロールで非同期ポストバックを実行し、UpdatePanel コントロールのコンテンツを再表示するように指定することもできます。非同期ポストバックを実行するコントロールは、**トリガ**と呼ばれます。

非同期ポストバックは、同期ポストバックに類似した動作をします。サーバーページのライフサイクルイベントはすべて発生し、ビューステートおよびフォーム データは保持されます。ただし、描画フェーズでは UpdatePanel コントロールのコンテンツのみがブラウザーに送信されます。ページの他の部分は変更されません。

部分ページレンダリングをサポートするには、ページに **ScriptManager** コントロールを含める必要があります。ScriptManager コントロールにより、ページのすべての更新パネルおよびそのトリガが追跡されます。サーバー間で部分ページレンダリングの動作が調整され、非同期ポストバックの結果、表示するページのセクションが決定されます。

ポストバックがパネル内から発生するたびにコンテンツが再表示される UpdatePanel コントロールの例を次に示します。

<Visual Basic>

```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
<head>
    <title>Partial-Page Rendering Server-Side Syntax</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
      <asp:ScriptManager ID="ScriptManager1" runat="server" />
      <asp:UpdatePanel ID="UpdatePanel1" runat="server">
      <ContentTemplate>
        <fieldset>
        <legend>UpdatePanel</legend>
        Content to update incrementally without a full
        page refresh.
        <br />
        Last update:  <%= DateTime.Now.ToString() %>
        <br />
        <asp:Calendar ID="Calendar" runat="server"/>
        </fieldset>
      </ContentTemplate>
      </asp:UpdatePanel>
      <script type="text/javascript" language="javascript">
      var prm = Sys.WebForms.PageRequestManager.getInstance();
      prm.add_pageLoaded(PageLoadedEventHandler);
      function PageLoadedEventHandler() {
         // custom script
      }
      </script>
    </div>
```

```
    </form>
</body>
</html>
```

<C#>

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
<head>
    <title>Partial-Page Rendering Server-Side Syntax</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
      <asp:ScriptManager ID="ScriptManager1" runat="server" />
      <asp:UpdatePanel ID="UpdatePanel1" runat="server">
      <ContentTemplate>
        <fieldset>
        <legend>UpdatePanel</legend>
        Content to update incrementally without a full
        page refresh.
        <br />
        Last update:  <%= DateTime.Now.ToString() %>
        <br />
        <asp:Calendar ID="Calendar" runat="server"/>
        </fieldset>
      </ContentTemplate>
      </asp:UpdatePanel>
      <script type="text/javascript" language="javascript">
      var prm = Sys.WebForms.PageRequestManager.getInstance();
      prm.add_pageLoaded(PageLoadedEventHandler);
      function PageLoadedEventHandler() {
         // custom script
```

```
      }
      </script>
    </div>
    </form>
</body>
</html>
```

部分ページ更新に対するクライアント スクリプトの使用方法

Microsoft AJAX Library の **PageRequestManager** クラスは、部分ページ更新をサポートします。これはブラウザーで実行され、非同期ポストバックへの応答を管理し、個々の領域のコンテンツを更新します。この機能を有効にするために必要な作業はありません。ページに 1 つ以上の UpdatePanel コントロールと 1 つの ScriptManager コントロールを追加すると自動的に発生します。

JavaScript および PageRequestManager クラスを使用して、ページの部分ページ更新をカスタマイズすることもできます。たとえば、複数のポストバックが実行中の場合に特定の非同期ポストバックを優先するスクリプトを記述できます。また、進行中のポストバックをユーザーが取り消すことができるようにすることもできます。

ページの読み込みが完了したときに呼び出されるイベントハンドラを提供するクライアントスクリプトの例を次に示します。

```
<script type="text/javascript" language="javascript">
var prm = Sys.WebForms.PageRequestManager.getInstance();
prm.add_pageLoaded(PageLoadedEventHandler);
function PageLoadedEventHandler() {
   // custom script
}
</script>
```

部分ページ レンダリング サポートの有効化

ScriptManager コントロールの **EnablePartialRendering** プロパティを設定することにより、ページの部分ページレンダリングを有効または無効にできます。ScriptManager コントロールの SupportsPartialRendering プロパティを設定することにより、ページに対して部分ページレンダリングがサポートされているかどうかを指定することもできます。SupportsPartialRendering プロパティを設定せず、さらに EnablePartialRendering プロパティが既定値の true の場合、部分ページレンダリングがサポートされているかどうかはブラウザーの機能を使用して判別されます。

ページに対して部分ページレンダリングが有効になっていない場合、無効にされた場合、またはブラウザーでサポートされていない場合は、フォールバック動作がページで使用されます。通常は非同期ポストバックを実行する操作で、代わりに同期ポストバックが実行され、ページ全体が更新されます。ページの UpdatePanel コントロールはすべて無視され、そのコンテンツは UpdatePanel コントロール内にないかのように表示されます。

クラス リファレンス

部分ページ レンダリングの主要なサーバークラスの一覧を表に示します。

表. 部分ページレンダリングの主要なサーバークラス

クラス	説明
UpdatePanel	部分ページレンダリングで更新されるページの領域を指定します。
ScriptManager	ASP.NET 内の AJAX コンポーネント、部分ページレンダリング、クライアント要求、および ASP.NET Web ページでのサーバー応答を管理します。
ScriptManagerProxy	ScriptManager コントロールが既に親要素に含まれるページに、入れ子になったコンポーネントがスクリプト参照およびサービス参照を追加できるようにします。

部分ページレンダリングの主要なクライアントクラスの一覧を表に示します。

表. 部分ページレンダリングの主要なクライアントクラス

クラス	説明
Sys.WebForms.PageRequestManager	クライアントの部分ページレンダリングを管理し、カスタムのクライアントスクリプトにメンバを公開します。

ASP.NET AJAX での Web サービスの使用

ここでは、AJAX 対応 ASP.NET Web ページのクライアントスクリプトから Web サービスにアクセスする方法について説明します。対象となるサービスは、ユーザーが作成するカスタムサービス、組み込みのアプリケーションサービスのいずれかです。アプリケーションサービスは ASP.NET AJAX の一部として提供され、認証、ロール、およびプロファイルの各サービスを含みます。

カスタム Web サービスは、**ASP.NET Web サービス (.asmx)**、**Windows Communication Foundation (WCF) サービス (.svc)** のいずれかの形式になります。

シナリオ

次の場合、WCF と ASP.NET を使用します。

- WCF サービスを既に作成してある場合は、AJAX 対応 Web ページのスクリプトがサービスにアクセスできるようにするエンドポイントを追加できます。
- ASP.NET Web サービス (.asmx) を既に作成してある場合は、AJAX 対応 Web ページのスクリプトが同じサービスにアクセスできるよう、そのサービスを変更できます。
- ASP.NET AJAX Web ページからアクセスするカスタム Web ページを作成するには、WCF サービスまたは ASP.NET Web サービス (.asmx) として実装します。
- 組み込みの ASP.NET アプリケーションサービスを使用すると、AJAX 対応 Web ページで実行されるクライアントスクリプトからユーザーの認証、ロール、およびプロファイルの各情報にアクセスできます。

背景

ASP.NET を使用すると、Web ページ内のクライアントスクリプトからアクセスできる Web サービスを作成できます。ページは、AJAX テクノロジを使用して Web サービス呼び出しを行う Web サービス通信レイヤを通じてサーバーと通信します。クライアントとサーバーの間では、データは非同期 (通常は JSON 形式) で交換されます。

AJAX クライアントのクライアント サーバー間の通信

AJAX 対応 Web ページでは、ブラウザーはサーバーに対してページの初期要求を行った後、Web サービスに対してデータの**非同期要求**を行います。クライアント通信要素は、サーバーおよびコアクライアントスクリプトライブラリからダウンロードされたプロキシクラスの形式です。サーバー通信要素は、ハンドラとカスタムサービスです。クライアントとサーバーの間の通信に含まれる要素を次の図に示します。

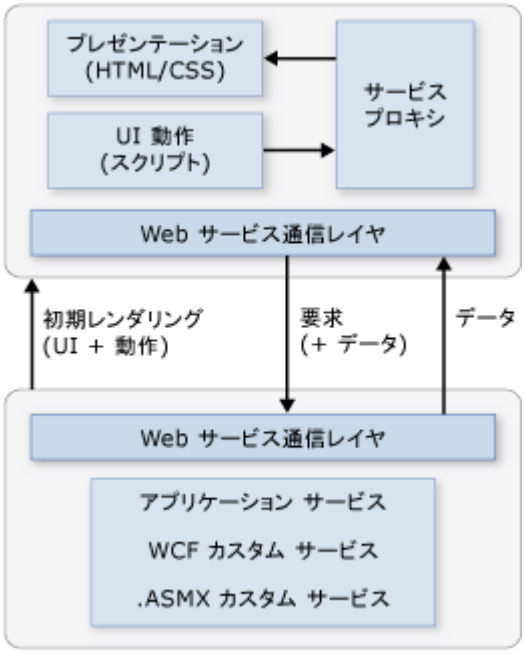

図. クライアントとサーバーの通信

AJAX クライアント アーキテクチャ

ブラウザーは、**プロキシクラス**を使用して Web サービスメソッドを呼び出します。プロキシクラスはサーバーによって自動的に生成され、ページの読み込み時にブラウザーにダウンロードされるスクリプトです。プロキシクラスには、Web サービスの公開メソッドを表すクライアントオブジェクトが用意されています。

Web サービス メソッドを呼び出すために、クライアントスクリプトはプロキシクラスの対応するメソッドを呼び出します。呼び出しは、**XMLHTTP** オブジェクト (XMLHttpRequest) を通じて非同期で行われます。

Web サービス通信レイヤには、プロキシクラスがサービス呼び出しを実行できるようにするライブラリスクリプトが含まれています。

プロキシクラスとコア Web サービス通信レイヤ内のコードにより、XMLHTTP の複雑さとブラウザー間の違いが隠ぺいされます。これにより、Web サービスの呼び出しに必要なクライアントスクリプトが簡略化されます。

Web サービス要求を行うには、次の 2 つの手法があります。

- HTTP POST メソッドを使用して Web サービスを呼び出します。POST 要求には、ブラウザーがサーバーに送信するデータを含む本体があります。POST 要求にはサイズ制限はありません。そのため、データのサイズが GET 要求固有のサイズ制限を超えた場合でも、POST 要求を使用できます。クライアントは要求を JSON 形式にシリアル化し、POST データとしてサーバーに送信します。サーバーは JSON データを .NET Framework 型に逆シリアル化して、実際の Web サービス呼び出しを行います。応答中に、サーバーは戻り値をシリアル化し、クライアントに返します。クライアントはそれを処理するために JavaScript オブジェクトに逆シリアル化します。
- HTTP GET メソッドを使用して Web サービスを呼び出します。これは POST 要求の機能に似ていますが、次のような違いがあります。
  - ✓ クライアントはクエリ文字列を使用して、パラメータをサーバーに送信します。
  - ✓ GET 要求は、**ScriptMethodAttribute** 属性を使用して構成された Web サービスメソッドのみを呼び出すことができます。
  - ✓ データサイズは、ブラウザーが許可する URL の長さに制限されます。

ASP.NET AJAX クライアントアーキテクチャを次の図に示します。

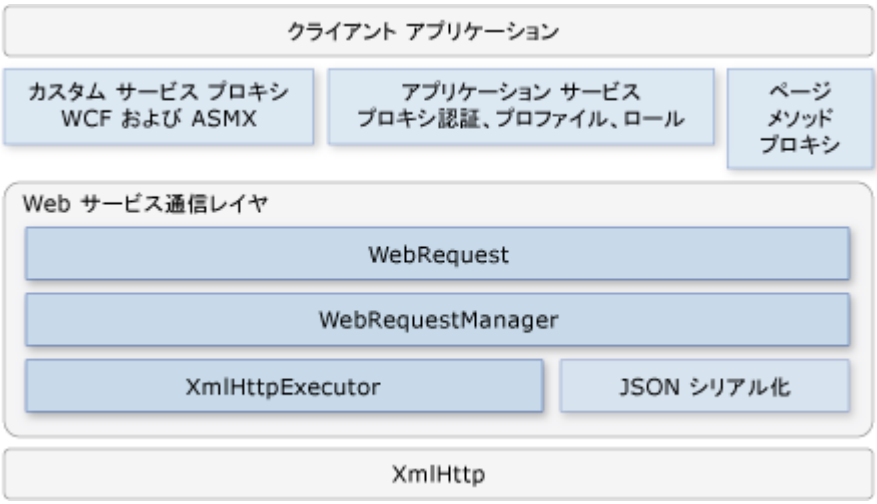

図. *AJAX クライアント アーキテクチャ*

クライアントアーキテクチャの要素には、コアライブラリ内の Web サービス通信レイヤ、およびページで使用されるサービス用のダウンロードされたプロキシクラスが含まれます。図に示されている個別の要素は次のとおりです。

- **カスタム サービス プロキシ クラス** —— サーバーによって自動的に生成され、ブラウザーにダウンロードされるクライアントスクリプトで構成されます。プロキシクラスには、ページで使用される WCF サービスまたは ASMX サービスごとにオブジェクトが用意されています (つまり、ページの **ScriptManager** コントロールの ServiceReferences 要素に含まれる項目ごとにオブジェクトが用意されています)。クライアントスクリプトでプロキシメソッドを呼び出すと、サーバー上の対応する Web サービスメソッドへの非同期要求が作成されます。
- **認証プロキシ クラス** —— **AuthenticationService** プロキシクラスは、サーバー認証アプリケーションサービスによって生成されます。このプロキシクラスにより、ユーザーはサーバーへのラウンドトリップを行わなくても、ブラウザーの JavaScript を使用してログオンまたはログアウトできます。
- **ロール プロキシ クラス** —— **RoleService** プロキシクラスは、サーバーロールアプリケーションサービスによって生成されます。このプロキシクラスにより、サーバーへのラウンドトリップを行わなくても、ユーザーをグループ化し、JavaScript を使用して各グループを 1 つの単位として扱うことができます。これは、サーバー上のリソースへのアクセスを有効にする、または拒否する場合に便利です。
- **プロファイル プロキシ クラス** —— **ProfileService** プロキシクラスは、サーバープロファイルアプリケーションサービスによって生成されます。このプロキシクラスは、サーバーへのラウンドトリップを行わなくても、JavaScript を使用して、現在のユーザーのプロファイル情報をクライアントで使用できるようにします。
- **ページ メソッド プロキシ クラス** —— ASP.NET ページで、静的メソッドを Web サービスメソッドであるかのように呼び出すことができるスクリプトインフラストラクチャをクライアントスクリプトに提供します。

- **Web サービス通信レイヤ** —— これは、クライアントスクリプトの種類を含むライブラリです。これらの種類により、ブラウザー (クライアント) はサーバー上のサービスと通信できます。また、クライアントとサーバーの間の非同期通信を確立および保持するための複雑な手続きが、クライアントアプリケーションで不要になります。非同期機能を提供するブラウザーの **XMLHTTP** オブジェクトをカプセル化し、クライアントアプリケーションがブラウザーに依存しないようにします。Web サービス通信レイヤの主な要素を次に示します。
  - ✓ **WebRequest** —— Web 要求を行うためのクライアントスクリプト機能を提供します。詳細については「WebRequest」クラスを参照してください。
  - ✓ **WebRequestManager** —— WebRequest オブジェクトによって関連するエグゼキュータオブジェクトに発行される Web 要求のフローを管理します。詳細については「WebRequestManager」クラスを参照してください。
  - ✓ **XmlHttpExecutor** —— ブラウザーの XMLHTTP サポートを使用して、非同期ネットワーク要求を行います。
  - ✓ **JSON シリアル化** —— JavaScript オブジェクトを JSON 形式にシリアル化します。JavaScript eval 関数を使用することで、逆シリアル化を使用できます。

既定のシリアル化形式は JSON ですが、Web サービスと ASP.NET Web ページ内の個別のメソッドは、XML などの別の形式を返すことができます。メソッドのシリアル化形式は属性で指定できます。たとえば ASMX サービスの場合、次の例に示すように、Web サービスメソッドが XML データを返すように **ScriptMethodAttribute** 属性を設定できます。

<C#>

```
[ScriptMethod(ResponseFormat.Xml)]
```

<Visual Basic>

```
<ScriptMethod(ResponseFormat.Xml)>
```

AJAX サーバー アーキテクチャ

クライアントアプリケーションとの通信を可能にする要素を含む、AJAX サーバーアーキテクチャを次の図に示します。

図. AJAX サーバー アーキテクチャ

サーバーアーキテクチャの要素には、HTTP ハンドラとシリアル化クラス、カスタムサービス、ページメソッド、およびアプリケーションサービスを持つ Web サービス通信レイヤが含まれます。図に示されている個別の要素は次のとおりです。

- **カスタム Web サービス** —— クライアントへの適切な応答を実装して返すことができるサービス機能を提供します。カスタム Web サービスは、ASP.NET サービスと WCF サービスのいずれかです。Web サービス通信レイヤは、クライアントスクリプトから非同期で呼び出すことができるクライアントスクリプトプロキシクラスを自動的に生成します。
- **ページ メソッド** —— このコンポーネントを使用すると、ASP.NET ページ内のメソッドを Web サービスメソッドであるかのように呼び出すことができます。ページメソッドは、ページメソッド呼び出しを実行しているページで定義する必要があります。
- **認証サービス** —— 認証サービスは、ユーザーがクライアント JavaScript を使用してログオンまたはログアウトできる認証プロキシクラスを生成します。このアプリケーションサービスはいつでも使用でき、インスタンス化する必要はありません。
- **ロール サービス** —— ロールサービスは、クライアント JavaScript が現在認証されているユーザーのロール情報にアクセスできるようにするロールプロキシクラスを生成します。このアプリケーションサービスはいつでも使用でき、インスタンス化する必要はありません。
- **プロファイル サービス** —— プロファイルサービスは、クライアント JavaScript が現在の要求に関連付けられたユーザーのプロファイルプロパティを取得および設定できるようにするプロファイルプロキシクラスを生成します。このアプリケーションサービスはいつでも使用でき、インスタンス化する必要はありません。
- **JSON シリアル化** —— サーバー JSON シリアル化コンポーネントにより、一般的な .NET Framework の型について、カスタマイズ可能な JSON 形式へのシリアル化と JSON 形式からの逆シリアル化を実行できます。
- **XML シリアル化** —— Web サービス通信レイヤは、Web サービスへの SOAP 要求の XML シリアル化、および Web サービスへの JSON 要求から XML 型を返すための XML シリアル化をサポートします。

ASP.NET サービスと WCF サービスの呼び出し例

クライアントスクリプトから ASP.NET サービスと WCF サービスを呼び出す方法の例を次に示します。

AJAX での Web サービス メソッドの呼び出し

.NET Framework により、クライアントスクリプトを使用してブラウザーから ASP.NET Web サービス (.asmx) メソッドを非同期で呼び出すことができます。ブラウザーとサーバーの間ではデータのみが転送されるため、ポストバックとページ全体の更新を行わなくても、サーバーベースのメソッドを呼び出すことができます。

ASP.NET Web ページで Web サービスメソッドを公開する方法を次の例に示します。

<Visual Basic>

```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
  <head id="Head1" runat="server">
    <style type="text/css">
      body { font: 11pt Trebuchet MS;
             font-color: #000000;
             padding-top: 72px;
             text-align: center }
      .text { font: 8pt Trebuchet MS }
    </style>
    <title>Simple Web Service</title>
    <script type="text/javascript">
      // This function calls the Web Service method.
      function GetServerTime()
      {
        Samples.AspNet.ServerTime.GetServerTime(OnSucceeded);
      }

      // This is the callback function that
      // processes the Web Service return value.
      function OnSucceeded(result)
      {
        var RsltElem = document.getElementById("Results");
        RsltElem.innerHTML = result;
      }
    </script>
  </head>

  <body>
    <form id="Form1" runat="server">
```

```
      <asp:ScriptManager runat="server" ID="scriptManager">
        <Services>
          <asp:ServiceReference path="ServerTime.asmx" />
        </Services>
      </asp:ScriptManager>
      <div>
        <h2>Server Time</h2>
        <p>Calling a service that returns the current server time.</p>
        <input id="EchoButton" type="button"
               value="GetTime" onclick="GetServerTime()" />
      </div>
    </form>
    <hr/>
    <div>
      <span id="Results"></span>
    </div>
  </body>
</html>
```

<C#>

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
  <head id="Head1" runat="server">
    <style type="text/css">
      body { font: 11pt Trebuchet MS;
             font-color: #000000;
             padding-top: 72px;
             text-align: center }
      .text { font: 8pt Trebuchet MS }
    </style>
    <title>Simple Web Service</title>
    <script type="text/javascript">
```

```
    // This function calls the Web Service method.
    function GetServerTime()
    {
      Samples.AspNet.ServerTime.GetServerTime(OnSucceeded);
    }

    // This is the callback function that
    // processes the Web Service return value.
    function OnSucceeded(result)
    {
      var RsltElem = document.getElementById("Results");
      RsltElem.innerHTML = result;
    }
  </script>
 </head>
 <body>
  <form id="Form1" runat="server">
    <asp:ScriptManager runat="server" ID="scriptManager">
      <Services>
        <asp:ServiceReference path="ServerTime.asmx" />
      </Services>
    </asp:ScriptManager>
    <div>
      <h2>Server Time</h2>
      <p>Calling a service that returns the current server time.</p>
      <input id="EchoButton" type="button"
             value="GetTime" onclick="GetServerTime()" />
    </div>
  </form>
  <hr/>
  <div>
    <span id="Results"></span>
  </div>
 </body>
</html>
```

ページスクリプトによって呼び出される Web ページと関連する Web サービスの例を次に示します。

<Visual Basic>

```
<%@ WebService Language="VB" Class="Samples.AspNet.ServerTime" %>

Imports System.Web
Imports System.Web.Services
Imports System.Xml
Imports System.Web.Services.Protocols
Imports System.Web.Script.Services

Namespace Samples.AspNet
  <WebService(Namespace:="http://tempuri.org/")> _
  <WebServiceBinding(ConformsTo:=WsiProfiles.BasicProfile1_1)> _
  <ScriptService()> _
  Public Class ServerTime
    Inherits System.Web.Services.WebService
    <WebMethod()> _
    Public Function GetServerTime() As String
      Return String.Format("The current time is {0}.", _
                           DateTime.Now)
    End Function
  End Class
End Namespace
```

<C#>

```
<%@ WebService Language="C#" Class="Samples.AspNet.ServerTime" %>

using System;
using System.Web;
using System.Web.Services;
using System.Xml;
using System.Web.Services.Protocols;
using System.Web.Script.Services;
```

```
namespace Samples.AspNet
{
  [WebService(Namespace = "http://tempuri.org/")]
  [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
  [ScriptService]
  public class ServerTime : System.Web.Services.WebService
  {
    [WebMethod]
    public string GetServerTime()
    {
      return String.Format("The server time is {0}.",
                           DateTime.Now);
    }
  }
}
```

AJAX クライアントからの HTTP 要求

前の例は、Web サービスの自動的に生成されたプロキシクラスを呼び出すことで、クライアントスクリプトから Web サービスを呼び出す方法を示しています。クライアントスクリプトから Web サービスへの低レベルの呼び出しを行うこともできます。この方法は、通信レイヤを管理したり、サーバーとの間で送信されているデータを調べたりする必要がある場合に使用します。この方法で Web サービスを呼び出すには「**WebRequest**」クラスを使用します。

WebRequest オブジェクトを使用して、指定した URL (HTTP エンドポイント) に接続する HTTP GET リクエストと HTTP POST リクエストを実装する方法を次の例に示します。

<JavaScript>

```
// ConnectingEndPoints.js
var resultElement;
function pageLoad()
{
    resultElement = $get("ResultId");
}

// This function performs a GET Web request.
function GetWebRequest()
{
```

```
    alert("Performing Get Web request.");

    // Instantiate a WebRequest.
    var wRequest = new Sys.Net.WebRequest();

    // Set the request URL.
    wRequest.set_url("getTarget.htm");
    alert("Target Url: getTarget.htm");

    // Set the request verb.
    wRequest.set_httpVerb("GET");

    // Set the request callback function.
    wRequest.add_completed(OnWebRequestCompleted);

    // Clear the results area.
    resultElement.innerHTML = "";

    // Execute the request.
    wRequest.invoke();
}

// This function performs a POST Web request.
function PostWebRequest()
{
    alert("Performing Post Web request.");

    // Instantiate a WebRequest.
    var wRequest = new Sys.Net.WebRequest();

    // Set the request URL.
    wRequest.set_url("postTarget.aspx");
    alert("Target Url: postTarget.aspx");

    // Set the request verb.
```

```
    wRequest.set_httpVerb("POST");

    // Set the request handler.
    wRequest.add_completed(OnWebRequestCompleted);

    // Set the body for he POST.
    var requestBody =
        "Message=Hello! Do you hear me?";
    wRequest.set_body(requestBody);
    wRequest.get_headers()["Content-Length"] =
        requestBody.length;

    // Clear the results area.
   resultElement.innerHTML = "";

    // Execute the request.
    wRequest.invoke();
}

// This callback function processes the
// request return values. It is called asynchronously
// by the current executor.
function OnWebRequestCompleted(executor, eventArgs)
{
    if(executor.get_responseAvailable())
    {
        // Clear the previous results.
        resultElement.innerHTML = "";

        // Display Web request status.
        resultElement.innerHTML +=
          "Status: [" + executor.get_statusCode() + " " +
                    executor.get_statusText() + "]" + "<br/>";

        // Display Web request headers.
```

```
        resultElement.innerHTML +=
            "Headers: ";

        resultElement.innerHTML +=
            executor.getAllResponseHeaders() + "<br/>";

        // Display Web request body.
        resultElement.innerHTML +=
            "Body:";

        if(document.all)
            resultElement.innerText +=
              executor.get_responseData();
        else
            resultElement.textContent +=
              executor.get_responseData();
    }
}
if (typeof(Sys) !== "undefined") Sys.Application.notifyScriptLoaded();
```

AJAX での WCF サービス操作の呼び出し

基本的に .asmx ベースのサービスを呼び出すのと同じように、クライアントスクリプトから **Windows Communication Foundation (WCF) サービス (.svc)** を非同期で呼び出すことができます。ASP.NET Web ページで WCF サービス操作を公開して呼び出す方法を次の例に示します。

<Visual Basic>

```
<%@ Page Language="VB" AutoEventWireup="true" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
<head runat="server">
    <style type="text/css">
        body {  font: 11pt Trebuchet MS;
                font-color: #000000;
```

```
                padding-top: 72px;
                text-align: center }
        .text { font: 8pt Trebuchet MS }
    </style>
    <title>Simple WCF Service Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:ScriptManager ID="ScriptManager1" runat="server">
            <Services>
                <asp:ServiceReference
                    Path="SimpleService.svc/ws"/>
            </Services>
            <Scripts>
                <asp:ScriptReference Path="service.js" />
            </Scripts>
        </asp:ScriptManager>

        <div>
            <h2>Simple WCF Service</h2>
            <input type='button' name="clickme" value="Greetings"
                onclick="javascript:OnClick()" />    
            <input type='button' name="clickme2" value="Greetings2"
                onclick="javascript:OnClick2()" />
            <hr/>
            <div>
                <span id="Results"></span>
            </div>
        </div>

    </form>
</body>
</html>
```

<C#>

```
<%@ Page Language="C#" AutoEventWireup="true"%>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
<head runat="server">
    <style type="text/css">
        body {  font: 11pt Trebuchet MS;
                font-color: #000000;
                padding-top: 72px;
                text-align: center }
        .text { font: 8pt Trebuchet MS }
    </style>
    <title>Simple WCF Service Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:ScriptManager ID="ScriptManager1" runat="server">
            <Services>
                <asp:ServiceReference
                    Path="SimpleService.svc/ws"/>
            </Services>
            <Scripts>
                <asp:ScriptReference Path="service.js" />
            </Scripts>
        </asp:ScriptManager>

        <div>
            <h2>Simple WCF Service</h2>
            <input type='button' name="clickme" value="Greetings"
                onclick="javascript:OnClick()" />    
            <input type='button' name="clickme2" value="Greetings2"
                onclick="javascript:OnClick2()" />
            <hr/>
            <div>
```

```
                <span id="Results"></span>
            </div>
        </div>

    </form>
</body>
</html>
```

<JavaScript>

```
var ServiceProxy;

function pageLoad()
{
    ServiceProxy = new ISimpleService();
    ServiceProxy.set_defaultSucceededCallback(SucceededCallback);
}

function OnClick()
{
    // var myService = new ISimpleService();
    ServiceProxy.HelloWorld1("George");
}

function OnClick2()
{
    var dc = new DataContractType();
    dc.FirstName = "George";
    dc.LastName = "Washington";
    ServiceProxy.HelloWorld2(dc);
}

// This is the callback function that
// processes the Web Service return value.
function SucceededCallback(result, userContext, methodName)
{
    var RsltElem = document.getElementById("Results");
```

```
    RsltElem.innerHTML = result + " from " + methodName + ".";
}
if (typeof(Sys) !== "undefined") Sys.Application.notifyScriptLoaded();
```

<Visual Basic>

```
Imports System
Imports System.Web
Imports System.Collections
Imports System.Collections.Generic
Imports System.Threading
Imports System.Xml
Imports System.Xml.Serialization
Imports System.Text
Imports System.IO
Imports System.Runtime.Serialization
Imports System.ServiceModel
Imports System.ServiceModel.Description
Imports System.ServiceModel.Dispatcher
Imports System.ServiceModel.Channels
Imports System.ServiceModel.Activation

' This a WCF service which consists of a contract,
' defined below as ISimpleService, and DataContractType,
' a class which implements that interface, see SimpleService,
' and configuration entries that specify behaviors associated with
' that implementation (see <system.serviceModel> in web.config)
Namespace Aspnet.Samples.SimpleService

    <ServiceContract()> _
    Public Interface ISimpleService
        <OperationContract()> _
        Function HelloWorld1(ByVal value1 As String) As String
        <OperationContract()> _
        Function HelloWorld2(ByVal dataContractValue1 _
        As DataContractType) As String
    End Interface 'ISimpleService
```

```
<ServiceBehavior(IncludeExceptionDetailInFaults:=True), _
AspNetCompatibilityRequirements(RequirementsMode:= _
AspNetCompatibilityRequirementsMode.Allowed)> _
Public Class SimpleService
    Implements ISimpleService

    Public Sub New()

    End Sub 'New

    Public Function HelloWorld1(ByVal value1 As String) As String _
    Implements ISimpleService.HelloWorld1
        Return "Hello " + value1
    End Function 'HelloWorld1

    Public Function HelloWorld2(ByVal dataContractValue1 _
    As DataContractType) As String _
    Implements ISimpleService.HelloWorld2
        Return "Hello " + dataContractValue1.FirstName + " " + _
        dataContractValue1.LastName
    End Function 'HelloWorld2
End Class 'SimpleService

<DataContract()> _
Public Class DataContractType
    Private _firstName As String
    Private _lastName As String

    <DataMember()> _
    Public Property FirstName() As String
        Get
            Return _firstName
        End Get
        Set(ByVal value As String)
```

```
                _firstName = value
            End Set
        End Property

        <DataMember()> _
        Public Property LastName() As String
            Get
                Return _lastName
            End Get
            Set(ByVal value As String)
                _lastName = value
            End Set
        End Property
    End Class 'DataContractType
End Namespace
```

<C#>

```
using System;
using System.Web;
using System.Collections;
using System.Collections.Generic;
using System.Threading;
using System.Xml;
using System.Xml.Serialization;
using System.Text;
using System.IO;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
using System.ServiceModel.Channels;
using System.ServiceModel.Activation;

// This a WCF service which consists of a contract,
// defined below as ISimpleService, and DataContractType,
// a class which implements that interface, see SimpleService,
```

```
// and configuration entries that specify behaviors associated with
// that implementation (see <system.serviceModel> in web.config)

namespace Aspnet.Samples
{
    [ServiceContract()]
    public interface ISimpleService
    {
        [OperationContract]
        string HelloWorld1(string value1);
        [OperationContract]
        string HelloWorld2(DataContractType dataContractValue1);
    }

    [ServiceBehavior(IncludeExceptionDetailInFaults = true)]
    [AspNetCompatibilityRequirements(RequirementsMode =
        AspNetCompatibilityRequirementsMode.Allowed)]
    public class SimpleService : ISimpleService
    {
        public SimpleService() { }

        public string HelloWorld1(string value1)
        {
            return "Hello " + value1;
        }
        public string HelloWorld2(DataContractType dataContractValue1)
        {
            return "Hello " + dataContractValue1.FirstName +
                                " " + dataContractValue1.LastName;
        }
    }

    [DataContract]
    public class DataContractType
    {
```

```
        string firstName;
        string lastName;

        [DataMember]
        public string FirstName
        {
            get { return firstName; }
            set { firstName = value; }
        }
        [DataMember]
        public string LastName
        {
            get { return lastName; }
            set { lastName = value; }
        }
    }
}
```

クラス リファレンス

クライアントスクリプトから呼び出すことができる Web サービスに関連する主なクラスの一覧を次の表に示します。

表. クライアントの名前空間

名前	説明
Sys.Net 名前空間	ASP.NET AJAX クライアントアプリケーションとサーバー上の Web サービスの間の通信を管理するクラスを含みます。**Sys.Net** 名前空間は、Microsoft AJAX Library に属します。
Sys.Serialization 名前空間	ASP.NET AJAX クライアントアプリケーションのデータシリアル化に関連するクラスを含みます。
Sys.Services 名前空間	ASP.NET AJAX クライアントアプリケーションで、ASP.NET 認証サービス、プロファイルサービス、およびその他のアプリケーションサービスへのスクリプトアクセスを提供する型が含まれます。**Sys.Services** 名前空間は、Microsoft AJAX Library に属します。

表. サーバーの名前空間

名前	説明
System.Web.Script.Serialization	マネージ型の JSON (JavaScript Object Notation) シリアル化

	と逆シリアル化を提供するクラスを含みます。シリアル化動作をカスタマイズするための機能拡張も提供します。

AJAX クライアント ライフ サイクル イベント

AJAX 対応の ASP.NET ページでは、ASP.NET Web ページと同じサーバーライフサイクルイベントが発生するだけでなく、**クライアント ライフ サイクル イベント**も発生します。クライアントイベントを使用すると、ポストバックと非同期ポストバックの両方の UI をカスタマイズできます (部分ページ更新)。クライアントイベントにより、ブラウザー内のページの有効期間中にカスタムスクリプトコンポーネントを管理することもできます。

クライアントイベントは、Microsoft AJAX Library 内のクラスによって発生します。

ページに ASP.NET AJAX サーバーコントロールが含まれている場合、これらのクラスは自動的にインスタンス化されます。クライアントクラスが提供する API により、イベントにバインドし、そのイベントのハンドラを提供できます。Microsoft AJAX Library はブラウザーに依存しないため、ハンドラ用に作成するコードは、サポート対象のすべてのブラウザーで同じように機能します。

最初の要求 (GET リクエスト) と同期ポストバックのキーイベントは、**Application** インスタンスの **load** イベントです。load イベントハンドラ内のスクリプトを実行すると、すべてのスクリプトとコンポーネントが読み込まれ、使用できるようになります。**UpdatePanel** コントロールを使用した部分ページレンダリングを有効にすると、キークライアントイベントは **PageRequestManager** クラスのイベントになります。これらのイベントにより、多数の一般的なシナリオを処理できます。たとえば、ポストバックのキャンセル、ポストバックの優先順位の設定、コンテンツが更新されたときの UpdatePanel コントロールのアニメーション化などの機能があります。

クライアント クラス

ASP.NET AJAX Web ページのクライアントライフサイクル中にイベントを発生させる 2 つのメイン Microsoft AJAX Library クラスは、Application クラスと PageRequestManager クラスです。

Application クラスは、ページに **ScriptManager** コントロールが含まれる場合、ブラウザーでインスタンス化されます。Application クラスは Page サーバーコントロールに似ています。このサーバーコントロールは Control クラスから派生しますが、サーバーイベントを発生させるための追加機能を提供します。同様に、Application クラスは **Sys.Component** クラスから派生しますが、ユーザーが処理できるクライアントライフサイクルイベントを発生させます。

ページに ScriptManager コントロールと 1 つ以上の **UpdatePanel** コントロールが含まれる場合、そのページは部分ページ更新を実行できます (部分ページレンダリングが有効で、ブラウザー

でサポートされている場合)。その場合、**PageRequestManager** クラスのインスタンスはブラウザーで自動的に有効になります。PageRequestManager クラスは、非同期ポストバックに固有のクライアントイベントを発生させます。

クライアント イベントのハンドラの追加

Application クラスと **PageRequestManager** クラスによって発生したイベントのハンドラを追加または削除するには、そのクラスの **add_eventname** メソッドと **remove_eventname** メソッドを使用します。Application オブジェクトの **init** イベントに **MyLoad** というハンドラを追加する方法の例を次に示します。

<JavaScript>

```
Sys.Application.add_init(MyInit);
function MyInit(sender) {
}
Sys.Appplication.remove_init(MyInit);
```

アプリケーションの load イベントと unload イベントの処理

Application オブジェクトの **load** イベントと **unload** イベントを処理する場合、ハンドラを明示的にイベントにバインドする必要はありません。その代わりに、予約名 pageLoad と pageUnload を使用する関数を作成します。この方法を使用して Application オブジェクトの load イベントのハンドラを追加する方法の例を次に示します。

<JavaScript>

```
function pageLoad(sender, args) {
}
```

その他のクライアント クラスのイベント

ここでは、**Application** クラスと **PageRequestManager** クラスによって発生するイベントについてのみ説明します。Microsoft AJAX Library にも、DOM 要素イベントのハンドラを追加、消去、および削除するためのクラスがあります。次のようなクラスがあります。

- **Sys.UI.DomEvent.addHandler** メソッドまたはショートカット **$addHandler**
- **Sys.UI.DomEvent.clearHandlers** メソッドまたはショートカット **$clearHandlers**
- **Sys.UI.DomEvent.removeHandler** メソッドまたはショートカット **$removeHandler**

Application クラスと PageRequestManager クラスのクライアント イベント

AJAX ASP.NET 対応のページで処理できる **Application** クラスと **PageRequestManager** クラスのクライアントイベントを次の表に示します。

表. Application クラスと PageRequestManager クラスのクライアントイベント

イベント	説明
Sys.Application.init イベント	すべてのスクリプトが読み込まれた後、オブジェクトが作成される前に発生します。コンポーネントを記述している場合、**init** イベントにより、ページにコンポーネントを追加するためのライフサイクル内のポイントが提示されます。これにより、後続のページライフルサイクルで、他のコンポーネントまたはスクリプトがコンポーネントを使用できます。ページ開発者の場合、ほとんどのシナリオでは、**init** イベントではなく **load** イベントを使用する必要があります。 **init** イベントは、ページが最初にレンダリングされたときに 1 回だけ発生します。後続の部分ページ更新では、**init** イベントは発生しません。
Sys.Application.load イベント	すべてのスクリプトが読み込まれ、$create を使用して作成されたアプリケーション内のすべてのオブジェクトが初期化された後に発生します。**load** イベントは、非同期ポストバックを含むサーバーへのすべてのポストバックに対して発生します。ページ開発者の場合、pageLoad という名前の関数を作成できます。この関数は、**load** イベントにハンドラを自動的に提供します。pageLoad ハンドラは、**add_load** メソッドによって **load** イベントに追加されたハンドラの後に呼び出されます。 **load** イベントは、Sys.ApplicationLoadEventArgs オブジェクトである eventargs パラメータを使用します。イベント引数を使用すると、部分ページ更新の結果としてページが更新されるかどうか、および前の **load** イベントの発生後に作成されたコンポーネントを確認できます。
Sys.Application.unload イベント	すべてのオブジェクトが破棄される前、およびブラウザーウィンドウの window.unload イベントが発生する前に発生します。 ページ開発者の場合、pageUnload という名前の関数を作成できます。この関数は、**unload** イベントにハンドラを自動的に提供します。pageUnload イベントは、ページがブラウザーからアンロードされる直前に呼

	び出されます。このイベント中で、コードが保持しているリソースを解放する必要があります。
Sys.Component.propertyChanged イベント	コンポーネントのプロパティが変更されたときに発生する可能性があります。このイベントが発生するのは、コンポーネント開発者がプロパティの set アクセサで Sys.Component.raisePropertyChange メソッドを呼び出した場合だけです。**propertyChanged** イベントは、Sys.applicationLoadEventArgs オブジェクトである eventargs パラメータを使用します。
Sys.Component.disposing イベント	**Application** インスタンスが破棄されたときに発生します。
Sys.WebForms.PageRequestManager の initializeRequest イベント	非同期要求が開始される前に発生します。このイベントを使用すると、別の非同期ポストバックを優先するなど、ポストバックをキャンセルできます。**initializeRequest** イベントは、Sys.WebForms.InitializeRequestEventArgs オブジェクトである eventargs パラメータを使用します。このオブジェクトにより、ポストバックの原因となった要素と基の要求オブジェクトが使用できるようになります。**InitializeRequestEventArgs** も cancel プロパティを公開します。**cancel** を true に設定すると、新しいポストバックがキャンセルされます。
Sys.WebForms.PageRequestManager の beginRequest イベント	非同期ポストバックが開始され、ポストバックがサーバーに送信される前に発生します。既に処理中のポストバックがある場合は停止されます (abortPostBack メソッドを使用)。このイベントを使用して、要求ヘッダーを設定したり、要求が処理中であることを示すページでアニメーションを開始したりできます。**beginRequest** イベントは、Sys.WebForms.BeginRequestEventArgs オブジェクトである eventargs パラメータを使用します。このオブジェクトにより、ポストバックの原因となった要素と基の要求オブジェクトが使用できるようになります。
Sys.WebForms.PageRequestManager の pageLoading イベント	サーバーから非同期ポストバックへの応答が受信された後、ページ上のコンテンツが更新される前に発生します。このイベントを使用すると、更新されたコンテンツにカスタム遷移効果を適用できます。**pageLoading** イベントは、Sys.WebForms.PageLoadingEventArgs オブジェクトである eventargs パラメータを使用します。このオブジェクトにより、最新の非同期ポストバックの結果として

	削除および更新されるパネルに関する情報が使用できるようになります。
Sys.WebForms.PageRequestManager の pageLoaded イベント	同期または非同期ポストバックの結果として、ページ上のすべてのコンテンツが更新された後に発生します。同期ポストバックではパネルが作成されるだけです が、非同期ポストバックではパネルが作成および更新されます。このイベントを使用すると、更新されたコンテンツのカスタム遷移効果を管理できます。**pageLoaded** イベントは、Sys.WebForms.PageLoadedEventArgs オブジェクトである eventargs パラメータを使用します。このオブジェクトにより、最新のポストバックで更新および作成されたパネルに関する情報が使用できるようになります。
Sys.WebForms.PageRequestManager の endRequest イベント	非同期ポストバックに対する応答が処理されページが更新された後、または応答の処理中 (エラーがある場合) に発生します。エラーが発生すると、ページは更新されません。このイベントを使用して、ユーザーへのカスタマイズされたエラー通知の提供、またはエラーの 記録を行います。**endRequest** イベントは、Sys.WebForms.EndRequestEventArgs オブジェクトである eventargs パラメータを使用します。このオブジェクトにより、発生したエラーおよびエラーが処理されたかどうかに関する情報が使用できるようになります。応答オブジェクトも使用できるようになります。

イベント順序の例

入れ子関係にある 2 つの **UpdatePanel** コントロールを含むページで発生したクライアントイベントの例を次に示します。親パネル内のボタンをクリックしたときと、入れ子になったパネル内のボタンをクリックしたときの違いがわかります。親パネル内のボタンをクリックすると、親パネルが更新され、入れ子になったパネルが削除および再作成されます。入れ子になったパネル内のボタンをクリックすると、入れ子になったパネルだけが更新されます。

<Visual Basic>

```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
```

```
<head>
    <title>Client Event Example</title>
    <style type="text/css">
    #OuterPanel { width: 600px; height: 200px; border: 2px solid blue; }
    #NestedPanel { width: 596px; height: 60px; border: 2px solid green;
                margin-left:5 px; margin-right:5px;
                margin-bottom:5px;}
    </style>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:ScriptManager ID="ScriptManager1" runat="server">
        <Scripts>
           <asp:ScriptReference Path="ClientEventTest.js" />
        </Scripts>
        </asp:ScriptManager>
        <asp:UpdatePanel ID="OuterPanel" UpdateMode="Conditional"
                         runat="server">
        <ContentTemplate>
            Postbacks from inside the outer panel and inner panel are
            asynchronous postbacks.
            PRM = Sys.WebForms.PageRequestManager.
            APP = Sys.Application.
            <br /><br />
            <asp:Button ID="OPButton1" Text="Outer Panel Button"
                        runat="server" />
            Last updated on
            <%= DateTime.Now.ToString() %>
            <br /><br />

            <asp:UpdatePanel ID="NestedPanel" UpdateMode="Conditional"
                             runat="server">
            <ContentTemplate>
                <asp:Button ID="NPButton1"
```

```
                Text="Nested Panel 1 Button" runat="server" />
            Last updated on
            <%= DateTime.Now.ToString() %>
            <br />
        </ContentTemplate>
        </asp:UpdatePanel>
    </ContentTemplate>
    </asp:UpdatePanel>

    <input type="button" onclick="Clear();" value="Clear" />
    <asp:Button ID="FullPostBack" runat="server"
      Text="Full Postback" />
    <a href="http://www.microsoft.com">Test Window Unload</a>
    <br />
    <span id="ClientEvents"></span>
  </div>
  </form>
</body>
</html>
```

<C#>

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
<head>
  <title>Client Event Example</title>
  <style type="text/css">
  #OuterPanel { width: 600px; height: 200px; border: 2px solid blue; }
  #NestedPanel { width: 596px; height: 60px; border: 2px solid green;
                 margin-left:5 px; margin-right:5px;
                 margin-bottom:5px;}
  </style>
</head>
```

```
<body>
    <form id="form1" runat="server">
    <div>
        <asp:ScriptManager ID="ScriptManager1" runat="server">
        <Scripts>
           <asp:ScriptReference Path="ClientEventTest.js" />
        </Scripts>
        </asp:ScriptManager>
        <asp:UpdatePanel ID="OuterPanel" UpdateMode="Conditional"
                         runat="server">
        <ContentTemplate>
            Postbacks from inside the outer panel and inner panel are
            asynchronous postbacks.
            PRM = Sys.WebForms.PageRequestManager.
            APP = Sys.Application.
            <br /><br />
            <asp:Button ID="OPButton1" Text="Outer Panel Button"
                        runat="server" />
            Last updated on
            <%= DateTime.Now.ToString() %>
            <br /><br />

            <asp:UpdatePanel ID="NestedPanel" UpdateMode="Conditional"
                             runat="server">
            <ContentTemplate>
                <asp:Button ID="NPButton1" Text="Nested Panel 1 Button"
                            runat="server" />
                Last updated on
                <%= DateTime.Now.ToString() %>
                <br />
            </ContentTemplate>
            </asp:UpdatePanel>
        </ContentTemplate>
        </asp:UpdatePanel>
```

```
        <input type="button" onclick="Clear();" value="Clear" />
        <asp:Button ID="FullPostBack" runat="server"
          Text="Full Postback" />
        <a href="http://www.microsoft.com">Test Window Unload</a>
        <br />
        <span id="ClientEvents"></span>
    </div>
    </form>
</body>
</html>
```

<JavaScript>

```
// Hook up Application event handlers.
var app = Sys.Application;
app.add_load(ApplicationLoad);
app.add_init(ApplicationInit);
app.add_disposing(ApplicationDisposing);
app.add_unload(ApplicationUnload);

// Application event handlers for component developers.
function ApplicationInit(sender) {
  var prm = Sys.WebForms.PageRequestManager.getInstance();
  if (!prm.get_isInAsyncPostBack())
  {
      prm.add_initializeRequest(InitializeRequest);
      prm.add_beginRequest(BeginRequest);
      prm.add_pageLoading(PageLoading);
      prm.add_pageLoaded(PageLoaded);
      prm.add_endRequest(EndRequest);
  }
  $get('ClientEvents').innerHTML += "APP:: Application init. <br/>";
}
function ApplicationLoad(sender, args) {
  $get('ClientEvents').innerHTML += "APP:: Application load. ";
  $get('ClientEvents').innerHTML += "(isPartialLoad = " +
    args.get_isPartialLoad() + ")<br/>";
```

```
}
function ApplicationUnload(sender) {
  alert('APP:: Application unload.');
}
function ApplicationDisposing(sender) {
  $get('ClientEvents').innerHTML += "APP:: Application disposing.
<br/>";
}
// Application event handlers for page developers.
function pageLoad() {
  $get('ClientEvents').innerHTML += "PAGE:: Load.<br/>";
}

function pageUnload() {
  alert('Page:: Page unload.');
}

// PageRequestManager event handlers.
function InitializeRequest(sender, args) {
  $get('ClientEvents').innerHTML += "<hr/>";
  $get('ClientEvents').innerHTML +=
    "PRM:: Initializing async request.<br/>";
}
function BeginRequest(sender, args) {
  $get('ClientEvents').innerHTML +=
    "PRM:: Begin processing async request.<br/>";
}
function PageLoading(sender, args) {
  $get('ClientEvents').innerHTML +=
    "PRM:: Loading results of async request.<br/>";
  var updatedPanels =
    printArray("PanelsUpdating", args.get_panelsUpdating());
  var deletedPanels =
    printArray("PanelsDeleting", args.get_panelsDeleting());
  var message = "-->" + updatedPanels + "<br/>-->" +
```

```
    deletedPanels + "<br/>";
  document.getElementById("ClientEvents").innerHTML += message;
}
function PageLoaded(sender, args) {
  $get('ClientEvents').innerHTML +=
    "PRM:: Finished loading results of async request.<br/>";
  var updatedPanels =
    printArray("PanelsUpdated", args.get_panelsUpdated());
  var createdPanels =
    printArray("PaneslCreated", args.get_panelsCreated());
  var message = "-->" + updatedPanels + "<br/>-->" +
    createdPanels + "<br/>";
  document.getElementById("ClientEvents").innerHTML += message;
}
function EndRequest(sender, args) {
  $get('ClientEvents').innerHTML += "PRM:: End of async request.<br/>";
}

// Helper functions.
function Clear()
{
  $get('ClientEvents').innerHTML = "";
}
function printArray(name, arr)
{
    var panels = name + '=' + arr.length;
    if (arr.length > 0)
    {
        panels += "(";
        for (var i = 0; i < arr.length; i++)
        {
            panels += arr[i].id + ',';
        }
        panels = panels.substring(0, panels.length - 1);
        panels += ")";
```

```
    }
    return panels;
}
```

一般的なシナリオのイベント順序

イベントの順序は、ページで使用されるコンテンツ、および行われるリクエストの種類 (初期リクエスト、ポストバック、または非同期ポストバック) によって異なります。ここでは、いくつかの一般的なシナリオのイベント順序について説明します。

初期リクエスト

ページの初期リクエストでは、限られた数のクライアントイベントが発生します。初期リクエストの次のシナリオを想定します。

- ページには **ScriptManager** コントロールが含まれ、コントロールの **SupportsPartialRendering** プロパティと **EnablePartialRendering** プロパティは両方とも true です。
- リクエストは HTTP GET です。
- サーバーからのレスポンスが正常に返されました。

この場合、次のクライアントイベントが次の順序で発生します。

1. 初期リクエストがサーバーに送信されます。
2. クライアントがレスポンスを受信します。
3. Application インスタンスが init イベントを発生させます。
4. Application インスタンスが load イベントを発生させます。

ブラウザー内のページの有効期間中に、**Application** インスタンスの **init** イベントが 1 回のみ発生します。後続の非同期ポストバックに対しては発生しません。初期リクエスト中に、PageRequestManager イベントは発生しません。

非同期ポストバック

非同期ポストバックは、サーバーにページデータを送信し、応答を受信してページの一部を更新します。非同期ポストバックの次のシナリオを想定します。

- ページには **ScriptManager** コントロールが含まれ、コントロールの **SupportsPartialRendering** プロパティと **EnablePartialRendering** プロパティは両方とも true です。
- ページには **UpdatePanel** コントロールが含まれ、コントロールの **ChildrenAsTriggers** プロパティは true です。
- **UpdatePanel** コントロール内のボタンにより、非同期ポストバックが開始されます。

◆ サーバーからのレスポンスが正常に返されました。

この場合、次のクライアントイベントが次の順序で発生します。

1. **UpdatePanel** 内のボタンがクリックされ、非同期ポストバックが開始されます。
2. **PageRequestManager** インスタンスが **initializeRequest** イベントを発生させます。
3. **PageRequestManager** インスタンスが **beginRequest** イベントを発生させます。
4. リクエストがサーバーに送信されます。
5. クライアントがレスポンスを受信します。
6. **PageRequestManager** インスタンスが **pageLoading** イベントを発生させます。
7. **PageRequestManager** インスタンスが **pageLoaded** イベントを発生させます。
8. **Application** インスタンスが **load** イベントを発生させます。
9. **PageRequestManager** インスタンスが **endRequest** イベントを発生させます。

Application インスタンスの **load** イベントは、**PageRequestManager pageLoaded** イベントの後、**endRequest** イベントの前に発生することに注意してください。

複数の非同期ポストバック

サーバー上またはブラウザー内で以前に開始されたリクエストが処理を完了する前に、ユーザーが新しいリクエストを開始すると、**複数の非同期ポストバック**が発生する可能性があります。複数の非同期ポストバックの次のシナリオを想定します。

◆ ページには **ScriptManager** コントロールが含まれ、コントロールの **SupportsPartialRendering** プロパティと **EnablePartialRendering** プロパティは両方とも true です。

◆ ページには、**UpdatePanel** コントロールが含まれています。

◆ 非同期ポストバックを開始する **UpdatePanel** コントロール内のボタンが 2 回クリックされます。2 回目のクリックは、1 回目のクリックによって開始されたリクエストをサーバーが処理している間に発生します。

◆ 最初の要求へのレスポンスがサーバーから正常に返されました。

この場合、次のクライアントイベントが次の順序で発生します。

1. **UpdatePanel** 内のボタンがクリックされ、非同期ポストバックが開始されます。
2. **PageRequestManager** インスタンスが **initializeRequest** イベントを発生させます。
3. **PageRequestManager** インスタンスが **beginRequest** イベントを発生させます。
4. リクエストがサーバーに送信されます。
5. ボタンが再度クリックされ、2 回目の非同期ポストバックが開始されます。
6. **PageRequestManager** インスタンスが 2 回目のボタンクリックの **initializeRequest** イベントを発生させます。

7. **PageRequestManager** インスタンスが 1 回目のボタンクリックの **endRequest** イベントを発生させます。
8. **PageRequestManager** インスタンスが 2 回目のボタンクリックの **beginRequest** イベントを発生させます。
9. 2 回目のクリックによって開始されたリクエストがサーバーに送信されます。
10. 2 回目のクリックに対してレスポンスが受信されます。
11. **PageRequestManager** インスタンスが **pageLoading** イベントを発生させます。
12. **PageRequestManager** インスタンスが **pageLoaded** イベントを発生させます。
13. **Application** インスタンスが **load** イベントを発生させます。
14. **PageRequestManager** インスタンスが **endRequest** イベントを発生させます。

非同期ポストバックの既定の動作では、最新の非同期ポストバックが優先されます。2 つの非同期ポストバックが順に発生し、最初のポストバックがブラウザーで処理中の場合、最初のポストバックはキャンセルされます。最初のポストバックがサーバーに送信された場合、サーバーは 2 番目のリクエストを受信したときに処理を開始し、最初のリクエストを返しません。

ページからの移動

ユーザーがページから移動すると、現在のページがブラウザーからアンロードされ、**unload** イベントを処理してリソースを解放できます。ページからの移動の次のシナリオを想定します。

- ページには **ScriptManager** コントロールが含まれ、コントロールの **SupportsPartialRendering** プロパティと **EnablePartialRendering** プロパティは両方とも true です。
- ターゲットページが存在します。

この場合、次のクライアントイベントが次の順序で発生します。

1. 新しいページのリクエストが開始されます。
2. ブラウザーが新しいページに対するレスポンスを受信します。
3. **Application** インスタンスが **unload** イベントを発生させます。
4. 新しいページが表示されます。

新しいページのリクエストでエラーが発生しても、**unload** イベントは発生しますが、新しいページは表示されません。

Microsoft AJAX CDN とは

CDN（コンテンツ デリバリー ネットワーク）とは Web コンテンツを配信するためのネットワークです。商用 CDN として有名なものに **Akamai**、**LimelightNetworks**、**EdgeCast** などがあります。CDN を利用すると、エッジ キャッシュによって画像、動画といった比較的容量の大きいコンテンツのパフォーマンスを改善できます。

2009 年 9 月より **jQuery** や **ASP.NET AJAX** などといった AJAX ライブラリに対してキャッシュのサポートを提供するサービスが開始されました。このサービスは無償で商業・非商業用ともに利用可能です。登録も必要ありません。

現在提供されているライブラリ

2010 年 1 月現在、**Microsoft Ajax Content Delivery Network** で提供される CDN キャッシュにロードされている JavaScript ライブラリ一覧は以下の通りです。この Ajax CDN で利用できるライブラリは今後、ASP.NET AJAX のバージョンアップと共に更新されます。

- ASP.NET Ajax Library version 0911（Beta）
- ASP.NET Ajax Library version 0910（Preview 6）
- ASP.NET Ajax Library version 0909（Preview 5）
- ASP.NET Ajax Library version 3.5
- jQuery version 1.4.2
- jQuery version 1.4.1
- jQuery version 1.4
- jQuery version 1.3.2
- jQuery Validation 1.6
- jQuery Validation 1.5.5
- ASP.NET MVC 1.0

各スクリプトファイルへの URL は、下記の Microsoft Ajax CDN サイトを参照してください。
Microsoft Ajax Content Delivery Network: http://www.asp.net/ajax/cdn/

使用方法（スクリプトタグへの追加）

Microsoft AJAX CDN から jQuery を使用したい場合は、以下のようなスクリプトタグをページに追加します。

```
<script src="http://ajax.microsoft.com/ajax/jquery/jquery-1.4.2.js"
    type="text/javascript"></script>
```

使用方法（ASP.NET 4.0 のスクリプト マネージャから）

ASP.NET 4.0 で使用する場合には、上記のスクリプトタグへの追加の他に、<asp:ScriptManager/> サーバー コントロールに追加された **EnableCdn** プロパティを **true** に設定します。

```
<asp:ScriptManager id="ScriptManager1" runat="server"
    EnableCdn="true" />
```

SSL サポートの提供

Microsoft CDN からスクリプトライブラリを参照する際に SSL のサポートを有効にするには以下の例のようにスクリプトの参照時に **https** を使用します。

```
<script src="https://ajax.microsoft.com/ajax/jquery/jquery-1.4.2.js"
    type="text/javascript"></script>
```

ASP.NET の状態管理の概要

ページがサーバーにポストされるたびに Web ページクラスの新しいインスタンスが作成されます。従来の Web プログラミングでは、通常、ページが再作成されるということは、ページとページ上のコントロールに関連付けられたすべての情報がラウンドトリップのたびに失われることを意味します。たとえば、ユーザーがテキスト ボックスに情報を入力した場合、その情報は、ブラウザーまたはクライアントデバイスからサーバーへのラウンドトリップにおいて失われます。

従来の Web プログラミングで継承されているこのような制限を克服するために、ASP.NET には、データをページ単位でもアプリケーション全体でも保存できるいくつかのオプションが用意されています。ビューステート、コントロールステート、隠しフィールド、Cookie、およびクエリ文字列などです。これらのデータはいずれもさまざまな方法でクライアントに格納されます。ただし、アプリケーション状態、セッション状態、およびプロファイルプロパティのデータはいずれもサーバーのメモリに格納されます。シナリオによって、各オプションには長所も短所もあります。

クライアント ベースの状態管理オプション

ここでは、ページ内またはクライアントコンピュータ上に情報を格納する状態管理オプションについて説明します。これらのオプションでは、ラウンドトリップ間でサーバーに情報が格納されることはありません。

ビューステート

ViewState プロパティは、同じページに対する複数の要求間で値を保持するためのディクショナリオブジェクトを提供します。これは、ラウンドトリップ間でページとコントロールのプロパティ値を保持するためにページが使用する既定の方法です。

ページが処理されると、ページとコントロールの現在の状態が文字列にハッシュされ、1 つの隠しフィールドとしてページに保存されます。また、**ViewState** プロパティに格納されているデータ量が MaxPageStateFieldLength プロパティに指定された値を超えた場合は、複数の隠しフィールドとして保存されます。ページがサーバーにポストバックされると、ページの初期化処理でそのビューステート文字列が解析され、プロパティ情報がそのページに復元されます。

ビューステートに値を格納することもできます。

コントロールステート

コントロールが正しく機能するように、コントロールステートデータを格納することが必要になる場合があります。たとえば、それぞれ異なる情報を表示する複数のタブを持つカスタムコントロールを作成した場合、そのコントロールは、正常に機能するためにラウンドトリップ間でどのタブが選択されたかを認識する必要があります。そのために、**ViewState** プロパティを使用できますが、開発者がページレベルでオフにできるため、コントロールが実質的に機能しなくなる可能性があります。

これを解決するために、ASP.NET ページ フレームワークは、コントロールステートという機能を ASP.NET に公開します。

ControlState プロパティは、コントロールに固有のプロパティ情報を保持できるようにします。ViewState プロパティと違ってオフにできません。

隠しフィールド

ASP.NET では、HTML の標準の**隠しフィールド**として保持される **HiddenField** コントロールに情報を格納できます。隠しフィールドはブラウザーに表示されませんが、標準のコントロールと同じようにプロパティを設定できます。ページがサーバーに送信されるときに、隠しフィールドの内容は他のコントロールの値と一緒に HTTP フォームコレクション内で送信されます。隠しフィールドは、ページに直接格納する必要があるページ固有の情報のためのリポジトリとして機能します。

HiddenField コントロールは、その Value プロパティに 1 つの変数を格納します。このコントロールは、ページに明示的に追加する必要があります。

ページの処理中に隠しフィールドの値を使用できるようにするには、HTTP の POST コマンドを使用してページを送信する必要があります。隠しフィールドを使用しているときにページがリンクや HTTP GET コマンドへの応答として処理される場合、隠しフィールドは使用できません。

Cookie

Cookie は、クライアントのファイルシステム上のテキストファイルに格納されるか、またはクライアントブラウザーセッション内でメモリに格納される、小さなデータです。Cookie には、サーバーがページ出力と共にクライアントに送信する、サイト固有の情報が含まれています。Cookie は (特定の有効期限を付与して) 一時的に保存することも、永続的に保存することもできます。

Cookie を使用すると、特定のクライアント、セッション、またはアプリケーションに関する情報を保存できます。Cookie はクライアントデバイスに保存されます。ブラウザーがページを要求すると、クライアントは、要求情報と一緒に Cookie の情報も送信します。サーバーが Cookie を読み取り、値を抽出します。Cookie の一般的な用途には、ユーザーがアプリケーションで既に認証されていることを示すための (通常は暗号化される) トークンの格納があります。

クエリ文字列

クエリ文字列は、ページの URL の最後に追加される情報です。一般的なクエリ文字列の例を次に示します。

```
http://www.contoso.com/listwidgets.aspx?category=basic&price=100
```

上記の URL パスのクエリ文字列は、疑問符 (?) で始まり、"category" と "price" という 2 つの属性値を含んでいます。

クエリ文字列は、状態情報を保持するための単純な方法ですが、いくつかの制限があります。たとえば、あるページから他のページに製品番号を渡して処理する場合など、ページ間で情報を渡すためにはクエリ文字列を使用すると簡単です。ただし、ブラウザーやクライアントデバイスには、URL の長さが 2,083 文字に制限されているものがあります。

ページの処理中にクエリ文字列の値を使用できるようにするには、HTTP の GET コマンドを使用してページを送信する必要があります。つまり、ページが HTTP の POST コマンドへの応答として処理される場合は、クエリ文字列を利用できません。

サーバー ベースの状態管理オプション

ASP.NET には、状態情報をクライアントではなくサーバーに保持するためのいくつかの方法が用意されています。サーバーベースの状態管理では、状態を保存するためにクライアントに送信する情報の量を削減できますが、サーバー上のリソースが大量に使用される可能性があります。ここでは、アプリケーション状態、セッション状態、プロファイルプロパティの 3 つのサーバーベース状態管理機能について説明します。

アプリケーション状態

ASP.NET では、アクティブな Web アプリケーションごとに、アプリケーション状態 (**HttpApplicationState** クラスのインスタンス) を使用して値を保存できます。アプリケーション状態は、Web アプリケーションのすべてのページからアクセスできるグローバルなストレージ機構です。そのため、アプリケーション状態は、サーバーへのラウンドトリップ間やページへの各要求間で維持する必要のある情報を格納するのに役立ちます。

アプリケーションの状態は、特定の URL への各要求時に作成される、キーと値のディクショナリに格納されます。この構造体にアプリケーション固有の情報を追加することにより、ページ要求間で情報を保持できます。

アプリケーション状態にアプリケーション固有の情報が追加された後は、サーバーがそれを管理します。

セッション状態

ASP.NET では、アクティブな Web アプリケーションのセッションごとに、**セッション状態** (**HttpSessionState** クラスのインスタンス) を使用して値を保存できます。

セッション状態はアプリケーション状態と似ていますが、スコープが現在のブラウザーセッションに限られる点で異なります。複数のユーザーがアプリケーションを使用している場合は、ユーザーセッションごとにセッション状態が異なります。また、同じユーザーがアプリケーションの使用をいったん中断し、その後で再度アプリケーションを使用した場合も、ユーザーセッションは、最初のセッションとは別のセッション状態になります。

セッション状態は、サーバーへのラウンド トリップ間およびページへの要求間で保持する必要のあるセッション固有の情報を格納するために使用される、キーと値のディクショナリとして構成されます。

セッション状態を使用して、次のタスクを実行できます。

- ブラウザーまたはクライアントデバッグからの要求を一意に識別し、それをサーバー上の個別のセッションインスタンスに割り当てる。
- セッション固有のデータをサーバー上に格納して、同じセッション内でブラウザーまたはクライアントデバイスからの複数の要求に対して使用する。
- 適切なセッション管理イベントを発生させる。また、それらのイベントを利用するアプリケーションコードを記述する。

セッション状態にアプリケーション固有の情報が追加された後は、サーバーがこのオブジェクトを管理します。セッション情報は、指定したオプションに従って、 Cookie、アウトプロセス サーバー、または Microsoft SQL Server を実行しているコンピュータに格納できます。

プロファイル プロパティ

ASP.NET には、ユーザー固有のデータを格納できる、**プロファイル プロパティ**という機能が用意されています。この機能はセッション状態とよく似ていますが、ユーザー セッションの有効期限が切れても、プロファイルデータは失われません。プロファイルプロパティ機能は、永続的な形式で格納され、個々のユーザーに関連付けられる ASP.NET プロファイルを使用します。ASP.NET プロファイルを使用すると、独自のデータベースを作成したり管理したりする手間をかけずに、ユーザー情報を容易に管理できます。また、このプロファイルでは、アプリケーション内のどこからでもアクセスできる、厳密に型指定された API を使ってユーザー情報を利用できます。このプロファイルには、任意の型のオブジェクトを格納できます。ASP.NET プロファイル機能は、ほとんどの種類のデータを定義および管理できると同時に、データをタイプ セーフな形で利用できるようにする汎用ストレージシステムを提供します。

プロファイルプロパティを使用するには、プロファイルプロバイダを設定する必要があります。ASP.NET には、プロファイルデータを SQL データベースに格納できるようにする **SqlProfileProvider** クラスが含まれていますが、プロファイルデータをカスタム形式で格納したり、XML ファイルなどのカスタムストレージ メカニズムや Web サービスに格納したりする独自のプロファイルプロバイダ クラスを作成することもできます。

プロファイルプロパティに配置されるデータは、アプリケーションメモリに格納されないため、インターネットインフォメーションサービス (IIS) やワーカープロセスを再起動してもデータが失われずに保護されます。また、プロファイルプロパティは、Web ファームや Web ガーデンなどの複数のプロセスでも保持できます。

Cookie の概要

Cookie とは

Cookie は、要求およびページと共に Web サーバーとクライアント間でやり取りされる少量のテキストです。Cookie には、ユーザーがサイトを訪問するたびに Web アプリケーションが読み込むことができる情報が格納されます。

Web アプリケーションは Cookie を使用してユーザー固有情報を格納できます。たとえば、サイトを訪問したユーザーに Cookie を使用してユーザー設定などの情報を格納できます。ユーザーが再び Web サイトを訪問すると、アプリケーションは以前に格納した情報を取得できます。

たとえば、ユーザーがサイトのページを要求したときにアプリケーションがページと共に日時を含む Cookie を送信すると、ブラウザーはページと共に Cookie を取得してユーザーのハードディスクのフォルダに格納します。

その後に、ユーザーが同じサイトのページを再び要求すると、URL を入力した時点でブラウザーはローカルのハード ディスクを確認して URL に関連付けられている Cookie を探します。Cookie がある場合、ブラウザーはページの要求と共に Cookie をサイトに送信します。これによって、アプリケーションはユーザーが前回サイトを訪問した日時を確認できます。この情報を使用してユーザーにメッセージを表示したり、または有効期限をチェックしたりできます。

Cookie は特定のページではなく Web サイトに関連付けられるため、ユーザーがサイト内のどのページを要求するかに関係なく、ブラウザーとサーバーは Cookie 情報を交換できます。ユーザーがさまざまなサイトを訪問するに従って、サイトが Cookie をブラウザーに送信するため、ブラウザーはすべての Cookie を個別に格納します。

Cookie によって、Web サイトはビジターに関する情報を格納できます。一般に、Cookie は Web アプリケーションで連続性を維持するもので、状態を管理する方法の 1 つです。実際に情報を交換している短い時間を除いて、ブラウザーと Web サーバーは切断されています。ユーザーが Web サーバーに行う各要求は、それぞれ独立して扱われます。ただし、多くの場合、Web サーバーにとってページを要求しているユーザーを認識できると便利です。たとえば、ショッピングサイトの Web サーバーは個々の顧客の行動を追跡して、ショッピングカートおよびその他のユーザー固有情報を管理します。したがって、Cookie はアプリケーションが処理を進めるために必要な適切な識別情報を提供する名刺のような役割を果たします。

Cookie の制限

ほとんどのブラウザーは、4,096 バイトまでの Cookie をサポートします。このサイズの制限により、Cookie はユーザー ID など少量のデータを格納するのに適しています。Cookie に格納された

ユーザー ID を使用して、ユーザーを識別してデータベースやその他のデータストアからユーザー情報を読み取ることができます。

ブラウザーにも、ユーザーのコンピュータに格納できるサイトあたりの Cookie の数の制限があります。ほとんどのブラウザーはサイトあたり 20 までの Cookie を許可します。それを超える数の Cookie を保存すると、古い Cookie から破棄されます。すべてのサイトから受け入れる Cookie の総数に対する絶対限界 (通常は 300) を設定するブラウザーもあります。

実際に影響を受ける Cookie の制限は、ブラウザーが Cookie を拒否するようにユーザーが設定できるということです。P3P プライバシー ポリシーを定義して Web サイトのルートに置くと、より多くのブラウザーがサイトから Cookie を受け入れるようになります。ただし、ユーザー固有情報を格納する場合に、Cookie の使用を避けて別の機構を使用する必要がある場合もあります。ユーザー情報を格納するための一般的な方法はセッション状態ですが、セッション状態は Cookie に依存します。

Cookie の書き込み

ブラウザーは、ユーザーのシステムで Cookie を管理します。Cookie は、Cookies というコレクションを公開する **HttpResponse** オブジェクトを使用してブラウザーに送信されます。HttpResponse オブジェクトには、**Page** クラスの **Response** プロパティとしてアクセスできます。ブラウザーに送信するすべての Cookie は、このコレクションに追加する必要があります。Cookie を作成するときは、**Name** と **Value** を指定します。各 Cookie には、ブラウザーが後で読み込むときに識別できるように一意の名前が付けられます。Cookie は名前によって格納されるため、2 つの Cookie に同じ名前を付けると上書きされます。

Cookie には、有効期限の日時も設定できます。期限切れの Cookie は、ユーザーが Cookie を書き込んだサイトを訪問したときにブラウザーによって削除されます。Cookie の有効期限は、アプリケーションにとって Cookie の値を有効にしておく必要がある期間に設定する必要があります。Cookie が期限切れにならないようにするには、有効期限を現在から 50 年に設定できます。

有効期限を設定しない場合も Cookie は作成されますが、ユーザーのハードディスクには保存されません。代わりに、Cookie はユーザーのセッション情報の一部として維持されます。ユーザーがブラウザーを閉じると、Cookie は破棄されます。このような非永続的な Cookie は、短時間だけ保存する必要がある情報またはセキュリティ上の理由からクライアントコンピュータのディスクに書き込むことができない情報に適しています。たとえば、非永続的な Cookie はユーザーが公共のコンピュータを使用する場合に便利です。この場合は、Cookie をディスクに書き込むべきではありません。

コード例

Cookie を Cookies コレクションに追加するには、いくつかの方法があります。Cookie を書き込む 2 つの方法の例を次に示します。

<Visual Basic>

```
Response.Cookies("userName").Value = "patrick"
Response.Cookies("userName").Expires = DateTime.Now.AddDays(1)

Dim aCookie As New HttpCookie("lastVisit")
aCookie.Value = DateTime.Now.ToString()
aCookie.Expires = DateTime.Now.AddDays(1)
Response.Cookies.Add(aCookie)
```

<C#>

```
Response.Cookies["userName"].Value = "patrick";
Response.Cookies["userName"].Expires = DateTime.Now.AddDays(1);

HttpCookie aCookie = new HttpCookie("lastVisit");
aCookie.Value = DateTime.Now.ToString();
aCookie.Expires = DateTime.Now.AddDays(1);
Response.Cookies.Add(aCookie);
```

この例では、userName と lastVisit という 2 つの Cookie を Cookies コレクションに追加します。最初の Cookie では、Cookies コレクションの値を直接設定します。このようにして値をコレクションに追加できるのは、Cookies が NameObjectCollectionBase 型の特殊なコレクションから派生しているからです。

2 番目のコード例では、HttpCookie 型のオブジェクトのインスタンスを作成し、プロパティを設定してから Add メソッドを使用して Cookies コレクションに追加します。HttpCookie オブジェクトをインスタンス化するときは、Cookie の名前をコンストラクタの一部として渡す必要があります。

この 2 つの例は共にブラウザーに Cookie を書き込むという同じタスクを実行します。どちらの方法でも、有効期限の値は DateTime 型にする必要があります。ただし、lastVisited 値も日時の値です。すべての Cookie 値は文字列として格納されるため、日時の値も String 型に変換する必要があります。

複数の値を含む Cookie

Cookie には、ユーザー名や前回の訪問などの 1 つの値を格納できます。1 つの Cookie に複数の名前と値のペアを格納することもできます。名前と値のペアは**サブキー**と呼ばれます。サブキーは、URL のクエリ文字列のようにレイアウトされます。たとえば、userName と lastVisit という 2 つの Cookie を個別に作成する代わりに、userName と lastVisit という 2 つのサブキーを含む userInfo という単一の Cookie を作成することもできます。

サブキーを使用する理由はいくつかあります。まず、関連する情報または類似した情報は 1 つの Cookie にまとめると便利です。さらに、すべての情報が 1 つの Cookie にあるため、有効期限などの Cookie の属性をすべての情報に適用できます。逆に、異なる種類の情報にそれぞれの有効期限を割り当てる場合は、個別の Cookie に情報を格納する必要があります。

サブキーを含む Cookie を作成する場合は、単一の Cookie を記述するための一連の構文を使用できます。2 つのサブキーを含む同じ Cookie を記述する 2 つの方法の例を次に示します。

<Visual Basic>

```
Response.Cookies("userInfo")("userName") = "patrick"
Response.Cookies("userInfo")("lastVisit") = DateTime.Now.ToString()
Response.Cookies("userInfo").Expires = DateTime.Now.AddDays(1)

Dim aCookie As New HttpCookie("userInfo")
aCookie.Values("userName") = "patrick"
aCookie.Values("lastVisit") = DateTime.Now.ToString()
aCookie.Expires = DateTime.Now.AddDays(1)
Response.Cookies.Add(aCookie)
```

<C#>

```
Response.Cookies["userInfo"]["userName"] = "patrick";
Response.Cookies["userInfo"]["lastVisit"] = DateTime.Now.ToString();
Response.Cookies["userInfo"].Expires = DateTime.Now.AddDays(1);

HttpCookie aCookie = new HttpCookie("userInfo");
aCookie.Values["userName"] = "patrick";
aCookie.Values["lastVisit"] = DateTime.Now.ToString();
aCookie.Expires = DateTime.Now.AddDays(1);
Response.Cookies.Add(aCookie);
```

Cookie のスコープの制御

既定では、サイトのすべての Cookie はクライアントにまとめて格納され、サイトへの要求と共にすべての Cookie がサーバーに送信されます。言い換えれば、サイトのすべてのページは、そのサイトのすべての Cookie を受け取ります。ただし、Cookie のスコープを次の 2 つの方法で設定できます。

- Cookie のスコープをサーバーの特定のフォルダに制限します。これによって、Cookie をサイトの特定のアプリケーションに限定できます。

◆ ドメインに対してスコープを設定します。これによって、Cookie にアクセスできるサブドメインを指定できます。

フォルダまたはアプリケーションに *Cookie* を制限する

Cookie をサーバーの特定のフォルダに制限するには、次の例のように Cookie の **Path** プロパティを設定します。

<Visual Basic>

```
Dim appCookie As New HttpCookie("AppCookie")
appCookie.Value = "written " & DateTime.Now.ToString()
appCookie.Expires = DateTime.Now.AddDays(1)
appCookie.Path = "/Application1"
Response.Cookies.Add(appCookie)
```

<C#>

```
HttpCookie appCookie = new HttpCookie("AppCookie");
appCookie.Value = "written " + DateTime.Now.ToString();
appCookie.Expires = DateTime.Now.AddDays(1);
appCookie.Path = "/Application1";
Response.Cookies.Add(appCookie);
```

パスには、サイトのルートの下の物理パスまたは仮想ルートを指定できます。その結果、Cookie は Application1 フォルダまたは仮想ルートのページのみが使用できるようになります。たとえば、www.contoso.com サイトでは、前の例で作成された Cookie は http://www.contoso.com/Application1/ というパスおよびそのフォルダの下のすべてのページで使用できます。ただし、Cookie は http://www.contoso.com/Application2/、http://www.contoso.com/ などの他のアプリケーションのページでは使用できません。

Cookie ドメインのスコープの制限

既定では、Cookie は特定のドメインに関連付けられます。たとえば、サイトが www.contoso.com で、ユーザーがこのサイトのページを要求したときに以前に書き込んだ Cookie をサーバーに送信するとします。これには、特定のパス値を指定した Cookie が含まれない場合もあります。サイトに contoso.com、sales.contoso.com、support.contoso.com などのサブドメインがある場合、Cookie を特定のサブドメインに関連付けることができます。そのためには、次の例のように Cookie の **Domain** プロパティを設定します。

<Visual Basic>

```
Response.Cookies("domain").Value = DateTime.Now.ToString()
Response.Cookies("domain").Expires = DateTime.Now.AddDays(1)
Response.Cookies("domain").Domain = "support.contoso.com"
```

<C#>

```
Response.Cookies["domain"].Value = DateTime.Now.ToString();
Response.Cookies["domain"].Expires = DateTime.Now.AddDays(1);
Response.Cookies["domain"].Domain = "support.contoso.com";
```

このようにドメインを設定すると、Cookie は指定されたサブドメインのページのみで使用できます。次の例のように Domain プロパティを使用すると、複数のサブドメインで共有できる Cookie を作成することもできます。

<Visual Basic>

```
Response.Cookies("domain").Value = DateTime.Now.ToString()
Response.Cookies("domain").Expires = DateTime.Now.AddDays(1)
Response.Cookies("domain").Domain = "contoso.com"
```

<C#>

```
Response.Cookies["domain"].Value = DateTime.Now.ToString();
Response.Cookies["domain"].Expires = DateTime.Now.AddDays(1);
Response.Cookies["domain"].Domain = "contoso.com";
```

これで、Cookie はプライマリ ドメインおよび sales.contoso.com ドメインと support.contoso.com ドメインで使用できるようになります。

Cookie の読み込み

ブラウザーがサーバーに要求するときは、要求と共にサーバーに対する Cookie を送信します。ASP.NET アプリケーションでは、**Page** クラスの **Request** プロパティで使用できる **HttpRequest** オブジェクトを使用して Cookie を読み込みます。HttpRequest オブジェクトの構造は基本的に HttpResponse オブジェクトと同じなため、HttpResponse オブジェクトに Cookie を書き込んだ場合と同様にして HttpRequest オブジェクトから Cookie を読み込むことができます。username という Cookie の値を取得して Label コントロールに表示する 2 つの方法のコード例を次に示します。

<Visual Basic>

```
If Not Request.Cookies("userName") Is Nothing Then
    Label1.Text = _
      Server.HtmlEncode(Request.Cookies("userName").Value)
End If

If Not Request.Cookies("userName") Is Nothing Then
    Dim aCookie As HttpCookie = Request.Cookies("userName")
    Label1.Text = Server.HtmlEncode(aCookie.Value)
```

```
End If
```

<C#>

```
if (Request.Cookies["userName"] != null)
    Label1.Text =
      Server.HtmlEncode(Request.Cookies["userName"].Value);

if (Request.Cookies["userName"] != null)
{
    HttpCookie aCookie = Request.Cookies["userName"];
    Label1.Text = Server.HtmlEncode(aCookie.Value);
}
```

Cookie が存在しない場合は NullReferenceException 例外が発生するため、Cookie の値を取得する前に Cookie が存在することを確認する必要があります。Cookie の内容をページに表示する前に **HtmlEncode** メソッドを呼び出してエンコードしていることにも注意してください。これによって、悪意のあるユーザーが Cookie に実行可能なスクリプトを追加することを防止できます。

Cookie のサブキーの値の読み込む方法は、値の設定とほとんど同じです。サブキーの値を取得する 1 つの方法のコード例を次に示します。

<Visual Basic>

```
If Not Request.Cookies("userInfo") Is Nothing Then
    Label1.Text = _
        Server.HtmlEncode(Request.Cookies("userInfo")("userName"))
    Label2.Text = _
        Server.HtmlEncode(Request.Cookies("userInfo")("lastVisit"))
End If
```

<C#>

```
if (Request.Cookies["userInfo"] != null)
{
    Label1.Text =
        Server.HtmlEncode(Request.Cookies["userInfo"]["userName"]);

    Label2.Text =
        Server.HtmlEncode(Request.Cookies["userInfo"]["lastVisit"]);
}
```

前のコード例では、以前に DateTime 値の文字列表現に設定した lastVisit サブキーの値を読み込みます。Cookie は値を文字列として格納するため、lastVisit 値を日付として使用する場合は、次の例のように適切な型に変換する必要があります。

<Visual Basic>

```
Dim dt As DateTime
dt = DateTime.Parse(Request.Cookies("userInfo")("lastVisit"))
```

<C#>

```
DateTime dt;
dt = DateTime.Parse(Request.Cookies["userInfo"]["lastVisit"]);
```

Cookie のサブキーは、NameValueCollection 型のコレクションになります。したがって、個々のサブキーを取得するもう 1 つの方法は、サブキーのコレクションを取得し、次の例のように名前によってサブキーの値を抽出することです。

<Visual Basic>

```
If Not Request.Cookies("userInfo") Is Nothing Then
    Dim UserInfoCookieCollection As _
        System.Collections.Specialized.NameValueCollection
    UserInfoCookieCollection = Request.Cookies("userInfo").Values
    Label1.Text = _
        Server.HtmlEncode(UserInfoCookieCollection("userName"))
    Label2.Text = _
        Server.HtmlEncode(UserInfoCookieCollection("lastVisit"))
End If
```

<C#>

```
if (Request.Cookies["userInfo"] != null)
{
    System.Collections.Specialized.NameValueCollection
        UserInfoCookieCollection;

    UserInfoCookieCollection = Request.Cookies["userInfo"].Values;
    Label1.Text =
        Server.HtmlEncode(UserInfoCookieCollection["userName"]);
    Label2.Text =
        Server.HtmlEncode(UserInfoCookieCollection["lastVisit"]);
}
```

Cookie の有効期限の変更

Cookie はブラウザーが管理します。ブラウザーは Cookie の有効期限の日時を使用して Cookie の保存を管理します。したがって、Cookie の名前と値を読み込むことはできますが、Cookie の有効期限の日時を読み込むことはできません。ブラウザーは、有効期限の情報を含めずに Cookie 情報をサーバーに送信します。Cookie の Expires プロパティは、常にゼロの日時の値を返します。Cookie の有効期限が問題になる場合は有効期限を再設定する必要があります。

Cookie コレクションの読み込み

ページで使用できるすべての Cookie を読み込む必要がある場合もあります。ページで使用できるすべての Cookie の名前と値を読み込むには、次のようなコードを使用して Cookies コレクションにループを実行します。

<Visual Basic>

```
Dim i As Integer
Dim output As System.Text.StringBuilder = New System.Text.StringBuilder
Dim aCookie As HttpCookie
For i = 0 to Request.Cookies.Count - 1
    aCookie = Request.Cookies(i)
    output.Append("Cookie name = " & Server.HtmlEncode(aCookie.Name) _
        & "<br />")
    output.Append("Cookie value = " & _
        Server.HtmlEncode(aCookie.Value) & "<br /><br />")
Next
Label1.Text = output.ToString()
```

<C#>

```
System.Text.StringBuilder output = new System.Text.StringBuilder();
HttpCookie aCookie;
for (int i=0; i<Request.Cookies.Count; i++)
{
    aCookie = Request.Cookies[i];
    output.Append("Cookie name = " + Server.HtmlEncode(aCookie.Name)
        + "<br />");
    output.Append("Cookie value = " + Server.HtmlEncode(aCookie.Value)
        + "<br /><br />");
}
Label1.Text = output.ToString();
```

前の例の制限は、Cookie にサブキーがある場合、サブキーが単一の名前/値の文字列として表示されることです。Cookie の **HasKeys** プロパティを確認すると、Cookie にサブキーがあるかどうかを判定できます。サブキーがある場合は、サブキーコレクションを読み込んで個々のサブキーの名前と値を取得できます。サブキーの値は、インデックス値を使用して **Values** コレクションから直接読み込むこともできます。対応するサブキーの名前は、配列文字列を返す Values コレクションの **AllKeys** メンバから取得できます。Values コレクションの Keys メンバを使用することもできます。ただし、AllKeys プロパティは最初にアクセスされたときにキャッシュされます。これとは対照的に、Keys プロパティはアクセスされるたびに配列を構築します。そのために、AllKeys プロパティは同じページの要求における 2 回目以降のアクセスがはるかに高速になります。

前の例を変更した例を次に示します。この例では、HasKeys プロパティを使用してサブキーをテストし、サブキーが検出されると、Values コレクションからサブキーを取得します。

<Visual Basic>

```
Dim i As Integer
Dim j As Integer
Dim output As System.Text.StringBuilder = New StringBuilder()
Dim aCookie As HttpCookie
Dim subkeyName As String
Dim subkeyValue As String
For i = 0 To Request.Cookies.Count - 1
    aCookie = Request.Cookies(i)
    output.Append("Name = " & aCookie.Name & "<br />")
    If aCookie.HasKeys Then
        For j = 0 To aCookie.Values.Count - 1
            subkeyName = Server.HtmlEncode(aCookie.Values.AllKeys(j))
            subkeyValue = Server.HtmlEncode(aCookie.Values(j))
            output.Append("Subkey name = " & subkeyName & "<br />")
            output.Append("Subkey value = " & subkeyValue & _
                "<br /><br />")
        Next
    Else
        output.Append("Value = " & Server.HtmlEncode(aCookie.Value) & _
            "<br /><br />")
    End If
Next
```

```
Label1.Text = output.ToString()
```

<C#>

```
for (int i=0; i<Request.Cookies.Count; i++)
{
    aCookie = Request.Cookies[i];
    output.Append("Name = " + aCookie.Name + "<br />");
    if(aCookie.HasKeys)
    {
        for(int j=0; j<aCookie.Values.Count; j++)
        {
            subkeyName = Server.HtmlEncode(aCookie.Values.AllKeys[j]);
            subkeyValue = Server.HtmlEncode(aCookie.Values[j]);
            output.Append("Subkey name = " + subkeyName + "<br />");
            output.Append("Subkey value = " + subkeyValue +
                "<br /><br />");
        }
    }
    else
    {
        output.Append("Value = " + Server.HtmlEncode(aCookie.Value) +
            "<br /><br />");
    }
}
Label1.Text = output.ToString();
```

次の例のように、サブキーを **NameValueCollection** オブジェクトとして抽出することもできます。

<Visual Basic>

```
Dim i As Integer
Dim j As Integer
Dim output As System.Text.StringBuilder = New StringBuilder()
Dim aCookie As HttpCookie
Dim subkeyName As String
Dim subkeyValue As String
For i = 0 To Request.Cookies.Count - 1
```

```
    aCookie = Request.Cookies(i)
    output.Append("Name = " & aCookie.Name & "<br />")
    If aCookie.HasKeys Then
        Dim CookieValues As _
            System.Collections.Specialized.NameValueCollection = _
                aCookie.Values
        Dim CookieValueNames() As String = CookieValues.AllKeys
        For j = 0 To CookieValues.Count - 1
            subkeyName = Server.HtmlEncode(CookieValueNames(j))
            subkeyValue = Server.HtmlEncode(CookieValues(j))
            output.Append("Subkey name = " & subkeyName & "<br />")
            output.Append("Subkey value = " & subkeyValue & _
                "<br /><br />")
        Next
    Else
        output.Append("Value = " & Server.HtmlEncode(aCookie.Value) & _
            "<br /><br />")
    End If
Next
Label1.Text = output.ToString
```

<C#>

```
System.Text.StringBuilder output = new System.Text.StringBuilder();
HttpCookie aCookie;
string subkeyName;
string subkeyValue;

for (int i = 0; i < Request.Cookies.Count; i++)
{
    aCookie = Request.Cookies[i];
    output.Append("Name = " + aCookie.Name + "<br />");
    if (aCookie.HasKeys)
    {
        System.Collections.Specialized.NameValueCollection
          CookieValues = aCookie.Values;
        string[] CookieValueNames = CookieValues.AllKeys;
```

```
        for (int j = 0; j < CookieValues.Count; j++)
        {
            subkeyName = Server.HtmlEncode(CookieValueNames[j]);
            subkeyValue = Server.HtmlEncode(CookieValues[j]);
            output.Append("Subkey name = " + subkeyName + "<br />");
            output.Append("Subkey value = " + subkeyValue +
                "<br /><br />");
        }
    }
    else
    {
        output.Append("Value = " + Server.HtmlEncode(aCookie.Value) +
            "<br /><br />");
    }
}
Label1.Text = output.ToString();
```

Cookie の変更と削除

Cookie を直接変更することはできません。Cookie を変更するには、新しい値を使用して新しい Cookie を作成し、ブラウザーに送信してクライアントにある古いバージョンを上書きします。ユーザーのサイト訪問の回数を格納する Cookie の値を変更するコード例を次に示します。

<Visual Basic>

```
Dim counter As Integer
If Request.Cookies("counter") Is Nothing Then
    counter = 0
Else
    counter = Int32.Parse(Request.Cookies("counter").Value)
End If
counter += 1
Response.Cookies("counter").Value = counter.ToString
Response.Cookies("counter").Expires = DateTime.Now.AddDays(1)
```

<C#>

```
int counter;
if (Request.Cookies["counter"] == null)
    counter = 0;
```

```
else
{
    counter = int.Parse(Request.Cookies["counter"].Value);
}
counter++;

Response.Cookies["counter"].Value = counter.ToString();
Response.Cookies["counter"].Expires = DateTime.Now.AddDays(1);
```

Cookie の削除

Cookie はユーザーのコンピュータにあるため、直接削除することはできません。ただし、ブラウザーに Cookie を削除させることはできます。その方法は、削除する Cookie と同じ名前で新しい Cookie を作成し、Cookie の有効期限を今日の日付より前に設定することです。ブラウザーが Cookie の有効期限をチェックする際に、期限切れになっている Cookie が削除されます。アプリケーションが使用するすべての Cookie を削除する 1 つの方法のコード例を次に示します。

<Visual Basic>

```
Dim aCookie As HttpCookie
Dim i As Integer
Dim cookieName As String
Dim limit As Integer = Request.Cookies.Count - 1
For i = 0 To limit
    cookieName = Request.Cookies(i).Name
    aCookie = New HttpCookie(cookieName)
    aCookie.Expires = DateTime.Now.AddDays(-1)
    Response.Cookies.Add(aCookie)
Next
```

<C#>

```
HttpCookie aCookie;
string cookieName;
int limit = Request.Cookies.Count;
for (int i = 0; i < limit; i++)
{
    cookieName = Request.Cookies[i].Name;
    aCookie = new HttpCookie(cookieName);
    aCookie.Expires = DateTime.Now.AddDays(-1);
```

```
    Response.Cookies.Add(aCookie);
}
```

サブキーの変更または削除

次の例に示すように、個々のサブキーの変更は作成する場合と同じです。

<Visual Basic>

```
Response.Cookies("userInfo")("lastVisit") = DateTime.Now.ToString()
Response.Cookies("userInfo").Expires = DateTime.Now.AddDays(1)
```

<C#>

```
Response.Cookies["userInfo"]["lastVisit"] = DateTime.Now.ToString();
Response.Cookies["userInfo"].Expires = DateTime.Now.AddDays(1);
```

個々のサブキーを削除するには、サブキーを保持する Cookie の Values コレクションを操作します。まず、Cookies オブジェクトから Cookie を取得して再作成します。次に Values コレクションの **Remove** メソッドを呼び出して、Remove メソッドに削除するサブキーの名前を渡します。次に、Cookie が変更された状態でブラウザーに送信されるように Cookies コレクションに追加します。サブキーを削除する方法の例を次に示します。この例では、削除するサブキーの名前を変数で指定しています。

<Visual Basic>

```
Dim subkeyName As String
subkeyName = "userName"
Dim aCookie As HttpCookie = Request.Cookies("userInfo")
aCookie.Values.Remove(subkeyName)
aCookie.Expires = DateTime.Now.AddDays(1)
Response.Cookies.Add(aCookie)
```

<C#>

```
string subkeyName;
subkeyName = "userName";
HttpCookie aCookie = Request.Cookies["userInfo"];
aCookie.Values.Remove(subkeyName);
aCookie.Expires = DateTime.Now.AddDays(1);
Response.Cookies.Add(aCookie);
```

Cookie とセキュリティ

Cookie のセキュリティの問題は、クライアントからデータを取得する場合と同様です。アプリケーションでは、Cookie はユーザー入力のもう 1 つの形式であるため、チェックとなりすましの対象

となります。Cookie はユーザーのローカルコンピュータに格納されるため、ユーザーは少なくとも Cookie に格納されているデータを参照できます。ユーザーはブラウザーが Cookie を送信する前に変更することもできます。

Cookie には、ユーザー名、パスワード、クレジットカードの番号などの重要情報を決して保存しないでください。ユーザーまたは Cookie を盗む可能性がある人物に入手されては困る情報は、Cookie に保存しないでください。

同様に、Cookie から取得する情報を安易に信用しないようにしてください。Cookie のデータは、それを書き込んだ時点と同じであると仮定しないでください。Cookie の値には、ユーザーが Web ページに入力したデータと同様の安全対策を適用してください。このトピックの以前の例では、ユーザーから取得した情報を表示する前にエンコードする場合と同様に、Cookie の値をページに表示する前に内容を HTML エンコードする例を示しました。

Cookie はブラウザーとサーバー間をプレーンテキストで送信されるため、Web トラフィックの横取りによって Cookie が読み取られる可能性があります。接続が SSL (Secure Sockets Layer) を使用する場合のみ Cookie を転送するように Cookie のプロパティを設定することもできます。SSL はユーザーのコンピュータにある Cookie を読み取りや変更から保護しませんが、Cookie が暗号化されるので転送中の読み取りは防止できます。

ブラウザーが Cookie を受け入れるかどうかの判定

ブラウザーは Cookie を拒否するように設定できます。Cookie を書き込むことができない場合にもエラーは発生しません。同様に、ブラウザーは現在の Cookie の設定に関する情報をサーバーに送信しません。

ブラウザーが Cookie を受け入れるかどうかを判定する 1 つの方法は、Cookie を書き込んでから再び読み込んでみることです。書き込んだ Cookie を読み込むことができない場合は、ブラウザーで Cookie が無効になっていると見なすことができます。

ブラウザーが Cookie を受け入れるかどうかをテストするコード例を次に示します。この例は、2 ページで構成されます。最初のページでは Cookie を書き込み、ブラウザーを 2 ページ目にリダイレクトします。2 ページ目では、Cookie を読み込みます。次に、ブラウザーを最初のページにリダイレクトし、テストの結果を含むクエリ文字列変数を URL に追加します。

最初のページのコードは次のようになります。

<Visual Basic>

```
Protected Sub Page_Load(ByVal sender As Object, _
        ByVal e As EventArgs) Handles Me.Load
    If Not Page.IsPostBack Then
        If Request.QueryString("AcceptsCookies") Is Nothing Then
```

```
            Response.Cookies("TestCookie").Value = "ok"
            Response.Cookies("TestCookie").Expires = _
                DateTime.Now.AddMinutes(1)
            Response.Redirect("TestForCookies.aspx?redirect=" & _
                Server.UrlEncode(Request.Url.ToString))
        Else
            Label1.Text = "Accept cookies = " & _
             Server.UrlEncode(Request.QueryString("AcceptsCookies"))
        End If
    End If
End Sub
```

<C#>

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        if (Request.QueryString["AcceptsCookies"] == null)
        {
            Response.Cookies["TestCookie"].Value = "ok";
            Response.Cookies["TestCookie"].Expires =
                DateTime.Now.AddMinutes(1);
            Response.Redirect("TestForCookies.aspx?redirect=" +
                Server.UrlEncode(Request.Url.ToString()));
        }
        else
        {
            Label1.Text = "Accept cookies = " +
                Server.UrlEncode(
                Request.QueryString["AcceptsCookies"]);
        }
    }
}
```

このページでは、まずポストバックかどうかを判別し、ポストバックではない場合は、テストの結果を含むクエリ文字列変数名 **AcceptsCookies** を探します。クエリ文字列変数がない場合はテスト

が完了していないため、TestCookie という Cookie を書き込みます。Cookie を書き込んだ後、Redirect を呼び出して、テストページ TestForCookies.aspx に転送します。テストページの URL に、現在のページの URL が含まれる redirect という名前のクエリ文字列変数が追加されます。これによって、テストの実行後、このページにリダイレクトして戻ることができます。

テストページにはコードのみを記述し、コントロールを含める必要はありません。テストページのコード例を次に示します。

<Visual Basic>

```
Sub Page_Load()
    Dim redirect As String = Request.QueryString("redirect")
    Dim acceptsCookies As String
    If Request.Cookies("TestCookie") Is Nothing Then
        acceptsCookies = "no"
    Else
        acceptsCookies = "yes"
        ' Delete test cookie.
        Response.Cookies("TestCookie").Expires = _
            DateTime.Now.AddDays(-1)
    End If
    Response.Redirect(redirect & _
                      "?AcceptsCookies=" & acceptsCookies, True)
End Sub
```

<C#>

```
protected void Page_Load(object sender, EventArgs e)
{
    string redirect = Request.QueryString["redirect"];
    string acceptsCookies;
    if (Request.Cookies["TestCookie"] == null)
        acceptsCookies = "no";
    else
    {
        acceptsCookies = "yes";
        // Delete test cookie.
        Response.Cookies["TestCookie"].Expires =
            DateTime.Now.AddDays(-1);
```

```
    }
    Response.Redirect(redirect + "?AcceptsCookies=" + acceptsCookies,
    true);
}
```

リダイレクト用クエリ文字列変数を読み込んだ後に、コードは Cookie の読み込みを試みます。ハウスキーピングのために、Cookie がある場合はすぐに削除します。テストが完了すると、コードは redirect クエリ文字列変数に含まれる URL を使用して新しい URL を作成します。新しい URL には、テストの結果を含むクエリ文字列変数も含まれます。最後に、新しい URL を使用してブラウザーを元のページにリダイレクトします。

この例を改善して、データベースなどの永続的な格納場所に Cookie のテスト結果を保存すると、ユーザーが元のページを表示するたびにテストを繰り返す必要がなくなります。既定でテストの結果をセッション状態に格納するには Cookie が必要です。

Cookie とセッション状態

ユーザーがサイトを訪問すると、サーバーはユーザーの訪問中存続する一意のセッションを確立します。ASP.NET は、各セッションに対してアプリケーションがユーザー固有情報を格納できるセッションステータス情報を維持します。

ASP.NET は、ユーザーをサーバーのセッション状態の情報にマップできるように、各ユーザーのセッション ID を追跡する必要があります。既定では、ASP.NET は非永続的な Cookie にセッション状態を格納します。ただし、ユーザーがブラウザーの Cookie を無効にしている場合は、セッション状態の情報を Cookie に格納できません。

ASP.NET には、Cookie なしのセッションのための代替手段があります。アプリケーションは、セッション ID を Cookie ではなく、サイトのページの URL に格納するように構成できます。アプリケーションがセッション状態に依存する場合は、Cookie なしのセッションを使用するように構成することを考慮します。ただし、ユーザーのセッションがアクティブな状態で同僚に URL を送信するなどの特別の状況で他のユーザーと URL を共有する場合は、両方のユーザーが同じセッションを共有することになるため、予測できない結果になることがあります。

ビューステートの概要

ASP.NET Web フォームで使用される**ビューステート**は、ページとコントロールの値をラウンドトリップ間で保持する方法です。ページの HTML マークアップを表示するときに、ポストバック間で保持する必要のある現在のページの状態と値を Base64 でエンコードした文字列にシリアル化します。次に、この情報がビューステートの隠しフィールドに出力されます。

また、ビューステートは、ポストバック間で維持する必要がある情報を保持するために、ASP.NET ページフレームワークで自動的に使用されます。この情報には、コントロールの既定値以外の値が含まれます。ビューステートを使用して、ページに固有のアプリケーションデータを格納することもできます。

ビューステートの機能

ビューステートは、ポストバック時に保持する必要のある値を格納できる、ASP.NET ページ内のリポジトリです。ページフレームワークはビューステートを使用して、ポストバック間でコントロール設定を維持します。

独自のアプリケーションでビューステートを使用すると、次の操作を実行できます。

- セッション状態またはユーザープロファイルに値を格納することなくポストバック間で値を保持します。
- ユーザーが定義するページプロパティまたはコントロールプロパティの値を格納します。
- SQL Server データベースまたは別のデータストアにビューステート情報を格納できる、カスタムビューステートプロバイダを作成します。

たとえば、ページ読み込みイベントで次回ページをサーバーに送信するときにコードがアクセスする情報をビューステートに格納できます。

Web アプリケーションには状態がありません。サーバーからページを要求されるたびに Web ページクラスの新しいインスタンスが作成されます。これは、通常、1 回のラウンドトリップごとにページとそのコントロールですべての情報が失われることを意味します。たとえば、既定では、ユーザーが HTML Web ページのテキストボックスに情報を入力すると、その情報はサーバーに送信されます。しかし、その情報は応答でブラウザーに返送されません。

Web プログラミング固有のこの制限に対処するために、ASP.NET ページフレームワークには、Web サーバーへのラウンドトリップ間でページとコントロールの値を保持するための状態管理機能がいくつか用意されています。この機能の 1 つがビューステートです。

既定では、ASP.NET ページフレームワークは、ビューステートを使用して、ページとコントロールの値をラウンドトリップ間で保持します。ページの HTML を表示するときに、ポストバック間で

保持する必要のある現在のページの状態と値を Base64 でエンコードした文字列にシリアル化します。次に、この情報がページの隠しフィールドに出力されます。

ページの **ViewState** プロパティを使用して、コード内のビューステートにアクセスできます。ViewState プロパティは、ビューステートデータを含むキーと値のペアが格納されたディクショナリです。

ページのデータを格納するカスタムの PageStatePersister クラスを実装すると、既定の動作を変更して別の場所にある SQL Server データベースなどにビューステートを格納できます。

ビューステートへの値の保存

ディクショナリオブジェクトを公開するページの **ViewState** プロパティを使用することにより、ビューステート情報にアクセスできます。このディクショナリを使用してカスタム値を格納できます。一般的な使用法としては、ページで定義したカスタムプロパティの値を格納します。

ビューステートは隠しフィールドとして送信されるため、ページの **PreRenderComplete** イベントが発生するまでビューステートを変更できます。ブラウザーにページが表示されると、ビューステートに対する変更は保存されません。

Web ページのソースを表示し、Base64 エンコードされた文字列をデコードできる場合は、隠しビューステートフィールド内の情報を確認することができます。これにより、セキュリティ上の問題が発生する場合があります。

値をビューステートに保存するには、保存する値を含む新しい項目を作成し、その項目をビューステートディクショナリに追加します。文字列と整数値をビューステートに保存するコードを含む ASP.NET Web ページの例を次に示します。

<Visual Basic>

```
<%@ Page Language="VB" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
  ' Sample ArrayList for the page.
  Dim PageArrayList As ArrayList

  Function CreateArray() As ArrayList
    ' Create a sample ArrayList.
    Dim result As ArrayList
    result = New ArrayList(4)
    result.Add("item 1")
```

```
    result.Add("item 2")
    result.Add("item 3")
    result.Add("item 4")
    Return result
  End Function

  Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs)
    If (Me.ViewState("arrayListInViewState") IsNot Nothing) Then
      PageArrayList = CType(Me.ViewState("arrayListInViewState"),
ArrayList)
    Else
      ' ArrayList isn't in view state, so it must be created and populated.
      PageArrayList = CreateArray()
    End If
    ' Code that uses PageArrayList.
  End Sub

  Sub Page_PreRender(ByVal sender As Object, ByVal e As EventArgs)
    ' Save PageArrayList before the page is rendered.
    Me.ViewState.Add("arrayListInViewState", PageArrayList)
  End Sub
</script>

<html>
<head>
    <title>View state sample</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    </div>
    </form>
</body>
</html>
```

<C#>

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
  // Sample ArrayList for the page.
  ArrayList PageArrayList;

  ArrayList CreateArray()
  {
    // Create a sample ArrayList.
    ArrayList result = new ArrayList(4);
    result.Add("item 1");
    result.Add("item 2");
    result.Add("item 3");
    result.Add("item 4");
    return result;
  }

  void Page_Load(object sender, EventArgs e)
  {
    if (ViewState["arrayListInViewState"] != null)
    {
      PageArrayList = (ArrayList)ViewState["arrayListInViewState"];
    }
    else
    {
      // ArrayList isn't in view state, so it must be created and
populated.
      PageArrayList = CreateArray();
    }
    // Code that uses PageArrayList.
  }

  void Page_PreRender(object sender, EventArgs e)
  {
```

```
    // Save PageArrayList before the page is rendered.
    ViewState.Add("arrayListInViewState", PageArrayList);
  }
</script>

<html>
<head>
    <title>View state sample</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    </div>
    </form>
</body>
</html>
```

ビューステートに格納できるデータ型

ビューステートに格納できるオブジェクトの型は次のとおりです。

- 文字列 (String)
- 整数 (Integer)
- Boolean 値
- Array オブジェクト
- ArrayList オブジェクト
- ハッシュ テーブル
- カスタムの型コンバータ

他の型のデータも格納できますが、ビューステートがその値をシリアル化できるように、**Serializable** 属性でクラスをコンパイルする必要があります。

ビューステートからの値の読み取り

ビューステートから値を読み取るには、ページの **ViewState** プロパティを取得し、ビューステート ディクショナリからその値を読み取ります。

ビューステートから arrayListInViewState という名前の ArrayList オブジェクトを取得し、GridView コントロールをデータ ソースとしてオブジェクトにバインドする方法の例を次に示します。

<Visual Basic>

```
Dim arrayList As ArrayList
arrayList = CType(ViewState("arrayListInViewState"), ArrayList)

Me.GridView1.DataSource = arrayList
Me.GridView1.DataBind()

arrayList = new ArrayList();
arrayList = (ArrayList)ViewState["arrayListInViewState"];

this.GridView1.DataSource = arrayList;
this.GridView1.DataBind();
```

ビューステートの値は String 型として型指定されます。Visual Basic で Option Strict On を設定する場合、前の例で示したように、ビューステート値は、適切な型にキャストしてから使用する必要があります。C# では、ビューステート値を読み取るときは常に適切な型にキャストする必要があります。

ビューステートから存在しない値を取得しようとしても、例外はスローされません。値がビューステートにあることを確認するには、まずオブジェクトが存在するかどうかを確認します。ビューステート エントリを確認する方法の例を次に示します。

<Visual Basic>

```
If ViewState("color") Is Nothing Then
    ' No such value in view state, take appropriate action.
End If
```

<C#>

```
if (ViewState["color"] == null)
    // No such value in view state, take appropriate action.
```

ビューステートのセキュリティ保護

既定では、ビューステートデータは、ページの隠しフィールドに格納され、Base-64 エンコーディングを使用してエンコードされます。また、ビューステートデータのハッシュは、MAC (Machine Authentication Code) キーを使用してデータから作成されます。ハッシュ値がエンコードされたビューステートに追加され、この文字列がページに格納されます。ページがサーバーにポストバックされると、ASP.NET ページフレームワークはハッシュ値を再計算し、そのハッシュ値と、ビューステートに格納されている値を比較します。ハッシュ値が一致しない場合、例外が発生し、ビューステートデータが無効である可能性が示されます。

ハッシュ値を作成することで、ASP.NET ページフレームワークは、ビューステートデータが破損または改ざんされていないかどうかをテストできます。ただし、改ざんされていない場合でも、ビューステートデータは悪意のあるユーザーによって傍受され、読み取られる可能性があります。

ビューステート ハッシュ値を計算するための *MAC* の使用

ビューステートハッシュ値の計算に使用される MAC キーは、自動生成されるか、Machine.config ファイルで指定されます。キーが自動生成される場合、コンピュータの MAC アドレスに基づいて作成されます。MAC アドレスは、そのコンピュータにインストールされているネットワーク アダプタの一意の GUID 値です。

悪意のあるユーザーが、ビューステートのハッシュ値に基づいて MAC キーをリバースエンジニアリングするのは困難です。したがって、MAC エンコーディングは、ビューステート データが変更されていないかどうかを確認するための適度に信頼性の高い方法と言えます。

通常、ハッシュの生成に使用する MAC キーが大きいほど、異なる文字列のハッシュ値が同じになる確率は低くなります。キーを自動生成する場合、ASP.NET は SHA-1 エンコーディングを使用して大型のキーを作成します。ただし、Web ファーム環境では、すべてのサーバーで同一の MAC キーを使用する必要があります。MAC キーが同一でないときに、ページを作成したサーバーとは別のサーバーにページがポストバックされると、ASP.NET ページフレームワークは例外を発生します。そのため、Web ファーム環境では、ASP.NET がキーを自動生成できるようにするのではなく、Machine.config ファイルでキーを指定する必要があります。その場合は、ハッシュ値に十分なセキュリティを提供できる長さのキーを作成するようにします。ただし、キーが長くなればなるほど、ハッシュの作成にかかる時間も長くなります。そのため、セキュリティ上のニーズとパフォーマンス上のニーズを比較検討する必要があります。

ビューステートの暗号化

MAC エンコーディングは、ビューステート データの改ざんを防止する上では役立ちますが、ユーザーによるデータの参照は防止できません。このデータは、2 とおりの方法でユーザーが参照できないようにできます。1 つは、ページを SSL 経由で送信する方法で、もう 1 つは、ビューステートデータを暗号化する方法です。ページが SSL 経由で送信されるようにすると、データパケットの盗聴や、ページが対象としていないユーザーによる不正なデータ アクセスを防止できます。

ただし、SSL はページをブラウザーに表示するためにページを復号化するため、ページを要求したユーザーは、ビューステートデータを参照できます。これは、ビューステートデータにアクセスできる承認済みユーザーについては心配する必要がない場合には問題ありません。ただし、どのユーザーにもアクセスを許可できないような情報をビューステートに格納するようなコントロールもあります。たとえば、ビューステート内にアイテム ID (データ キー) を格納するデータバインド コントロールがページに含まれている場合です。これらの ID に、顧客 ID などの重要情報が含まれている場合は、

SSL 経由でページを送信するだけでなく、または SSL 経由でページを送信する代わりに、ビューステート データを暗号化する必要があります。

ビューステートデータを暗号化するには、ページの **ViewStateEncryptionMode** プロパティに true を設定します。情報をビューステートに格納すると、通常の読み取りおよび書き込み方法を利用でき、暗号化と復号化がページによってすべて処理されます。ビューステートデータを暗号化すると、アプリケーションのパフォーマンスに影響する可能性があります。そのため、暗号化は必要な場合以外には使用しないでください。

コントロールステートの暗号化

コントロールステートを使用するコントロールには、**RegisterRequiresViewStateEncryption** メソッドを呼び出して、ビューステートを暗号化する必要があるものもあります。ページのコントロールがビューステートの暗号化を必要とする場合、ページのすべてのビューステートが暗号化されます。

ユーザーごとのビューステートのエンコーディング

Web サイトがユーザーを認証する場合、Page_Init イベント ハンドラに **ViewStateUserKey** プロパティを設定して、ページのビューステートを特定のユーザーと関連付けることができます。これはワンクリック攻撃の防止に役立ちます。この攻撃では、悪意のあるユーザーが、以前作成したページからあらかじめビューステートが埋め込まれている有効な Web ページを作成します。攻撃者は、ユーザーに、ユーザー ID を使用してサーバーにページを送信するリンクをクリックするように促します。

ViewStateUserKey プロパティが設定されていると、元のページのビューステートのハッシュを作成するのに攻撃者の ID が使用されます。何も知らないユーザーがページを再送信してしまった場合でも、ユーザーキーが異なるのでハッシュ値が異なります。ページは検証に失敗し、例外がスローされます。

ViewStateUserKey プロパティを、各ユーザーに対して一意の値 (ユーザー名やユーザー ID など) に関連付ける必要があります。

共有ホスト環境での構成のセキュリティ保護

共有ホスト環境では、状態管理プロパティが悪意のあるユーザーによって変更される可能性があり、コンピュータ上の他のアプリケーションに影響する場合があります。このような変更は、Machine.config ファイルを直接変更したり、構成クラスを介して変更したり、その他の管理ツールや構成ツールを使用したりして行われる可能性があります。アプリケーション構成の変更は、構成ファイルのセクションを暗号化することによって防止できます。

ビューステートの使用に関する考慮事項

ビューステートが提供する状態情報は、特定の ASP.NET ページに関するものです。複数のページの情報を使用する必要がある場合や、Web サイトへの訪問以外の目的で情報を保持する必要がある場合は、別の方法で状態を維持する必要があります。アプリケーション状態、セッション状態、またはプロファイルプロパティを使用できます。

ビューステート情報は XML にシリアル化された後で Base64 エンコーディングを使用してエンコードされます。このため、大量のデータが生成される可能性があります。ページをサーバーにポストすると、ビューステートの内容がページのポストバック情報の一部として送信されます。ビューステートに大量の情報が格納されている場合、ページのパフォーマンスに影響を与える可能性があります。アプリケーションの通常のデータを使用してページのパフォーマンスをテストし、ビューステートのサイズがパフォーマンス上の問題を発生させるかどうかを確認します。

個別のコントロールのコントロール情報を格納する必要がない場合は、コントロールのビューステートを無効にできます。各ポストバックのデータストアからページのコントロールが更新される場合、ビューステートのサイズを小さくするために、そのコントロールのビューステートをオフにできます。たとえば、GridView コントロールなどのコントロールのビューステートをオフにできます。

そのほかに、隠しフィールドに格納するデータが大量になると、プロキシやファイアウォールによってはデータを含むページにアクセスできなくなる場合があることに注意する必要があります。許可されるデータの最大量はファイアウォール実装やプロキシ実装によって異なるため、サイズの大きい隠しフィールドを使用すると断続的な問題が発生する可能性があります。**ViewState** プロパティに格納されているデータの容量がページの **MaxPageStateFieldLength** プロパティで指定されている値を超えると、ページはビューステートを複数の隠しフィールドに分割します。これにより、ファイアウォールが拒否するサイズを下回るように個々の隠しフィールドのサイズが小さくなります。

一部のモバイルデバイスは、隠しフィールドをまったく使用できません。このため、これらのデバイスではビューステートは機能しません。

コントロールステート

ビューステート以外に、ASP.NET は**コントロールステート**をサポートしています。ページまたはコントロールに対してビューステートが無効の場合でも、ページはコントロールステートを使用して、ポストバックの間維持する必要のあるコントロール情報を保持します。ビューステートと同様に、コントロールステートは 1 つ以上の隠しフィールドに格納されます。

セッション状態の概要

ASP.NET セッション状態を使用すると、ユーザーが Web アプリケーションの ASP.NET ページ間を移動するときに、ユーザーの値の格納と取得を行うことができます。HTTP は状態のないプロトコルです。つまり、Web サーバーはページに対する HTTP 要求をそれぞれ独立した要求として処理します。サーバーは以前の要求中に使用された変数値の情報を保持しません。ASP.NET セッション状態は、一定の時間枠に同じブラウザーから受け取った要求を 1 つのセッションとして認識し、そのセッションの間、変数値を保持できるようにします。

セッション状態の使用

ASP.NET セッション状態は、すべての ASP.NET アプリケーションで、既定で有効になります。セッション状態の代わりに、次を使用することもできます。

- ASP.NET アプリケーションのすべてのユーザーがアクセスできる変数を格納するアプリケーション状態
- ユーザー値を有効期限切れにすることなく、データストアに永続化するプロファイルプロパティ
- すべての ASP.NET アプリケーションが使用できるメモリに値を格納する ASP.NET キャッシュ
- ページの値を永続化するビューステート
- Cookie
- HTTP 要求から利用できる、HTML フォームのクエリ文字列とフィールド

セッション変数

セッション変数は **HttpContext** の **Session** プロパティを介して公開される **SessionStateItemCollection** オブジェクトに格納されます。ASP.NET ページでは、現在のセッション変数は Page オブジェクトの Session プロパティを介して公開されます。

セッション変数のコレクションは、変数名別または整数インデックス別にインデックス処理されます。セッション変数を作成するには、名前を指定してセッション変数を参照します。セッション変数を宣言したり、明示的にコレクションに追加したりする必要はありません。次の例では、ユーザーの名と姓を表すセッション変数を ASP.NET ページで作成し、それぞれに TextBox コントロールから取得した値を設定します。

<Visual Basic>

```
Session("FirstName") = FirstNameTextBox.Text
Session("LastName") = LastNameTextBox.Text
```

<C#>

```
Session["FirstName"] = FirstNameTextBox.Text;
Session["LastName"] = LastNameTextBox.Text;
```

セッション変数には任意の有効な .NET Framework 型を指定できます。StockPicks という名前のセッション変数に ArrayList オブジェクトを格納する例を次に示します。StockPicks セッション変数によって返される値は、SessionStateItemCollection からの取得時に適切な型にキャストする必要があります。

<Visual Basic>

```
' When retrieving an object from session state, cast it to
' the appropriate type.
Dim stockPicks As ArrayList = CType(Session("StockPicks"), ArrayList)

' Write the modified stock picks list back to session state.
Session("StockPicks") = stockPicks
```

<C#>

```
// When retrieving an object from session state, cast it to
// the appropriate type.
ArrayList stockPicks = (ArrayList)Session["StockPicks"];

// Write the modified stock picks list back to session state.
Session["StockPicks"] = stockPicks;
```

セッション ID

セッションは、**SessionID** プロパティを使用して読み込むことができる一意識別子によって識別されます。ASP.NET アプリケーションでセッション状態が有効になっている場合、アプリケーション内のページへの各要求について、ブラウザーから送信された SessionID 値がチェックされます。SessionID 値が提供されていない場合、ASP.NET は新しいセッションを開始し、そのセッションの SessionID 値が応答と共にブラウザーに送信されます。

既定では、SessionID 値は Cookie に格納されます。ただし、"Cookie なし" のセッションについては SessionID 値を URL に格納するようにアプリケーションを設定することもできます。

セッションは、要求が同じ SessionID 値で行われ続ける限りはアクティブと見なされます。あるセッションの要求から次の要求までの時間が、指定されているタイムアウト値 (分) を超えると、セッションは有効期限切れと見なされます。有効期限切れの SessionID 値で要求が行われると、新しいセッションが開始されます。

Cookie なしのセッション *ID*

既定では、SessionID 値はブラウザーの無期限セッション Cookie に格納されます。ただし、セッション ID が Cookie に格納されないようにすることもできます。このためには、Web.config ファイルの sessionState セクションで、cookieless 属性を true に設定します。

次の例は、Cookie なしのセッション ID を使用するように ASP.NET アプリケーションを構成する **Web.config** ファイルを示しています。

```
<configuration>
  <system.web>
    <sessionState cookieless="true"
      regenerateExpiredSessionId="true" />
  </system.web>
</configuration>
```

ASP.NET は、一意のセッション ID をページの URL に自動的に挿入して、Cookie なしのセッション状態を保持します。たとえば、次の URL は ASP.NET によって変更され、lit3py55t21z5v55vlm25s55 という一意のセッション ID が挿入されています。

```
http://www.example.com/(S(lit3py55t21z5v55vlm25s55))/orderform.aspx
```

ブラウザーへのページの送信時、ASP.NET は、ページ内でアプリケーション相対パスを使用しているリンクにセッション ID 値を埋め込み、リンクを変更します (絶対パスを使用しているリンクは変更されません)。この方法で変更されたリンクをユーザーがクリックする限り、セッション状態は保持されます。ただし、アプリケーションから提供された URL をクライアントが書き直した場合、ASP.NET は、セッション ID を解決して要求を既存のセッションに関連付けることができなくなります。この場合、要求に対して新しいセッションが開始されます。

セッション ID は、URL 内のアプリケーション名の後のスラッシュと、残りのファイル ID または仮想ディレクトリ ID の間に埋め込まれます。これにより、ASP.NET は、アプリケーション名を解決してから、SessionStateModule を要求に含めることができます。

期限切れのセッション *ID* の再生成

既定では、Cookie なしのセッションで使用されるセッション ID 値は再利用されます。つまり、期限切れのセッション ID によって要求が行われた場合、その要求に含まれている SessionID 値を使用して新しいセッションが開始されます。この結果、Cookie なしの SessionID 値を含むリンクが複数のブラウザーで使用される場合には、セッションデータが意図に反して共有される可能性があります。このような状況が発生する可能性があるのは、検索エンジン、電子メールメッセージ、または他のプログラムを通じてリンクが渡された場合です。セッション データが共有される可能性を低減するには、セッション ID の再利用を行わないようにアプリケーションを構成します。これを行うには、

sessionState 構成要素の **regenerateExpiredSessionId** 属性を true に設定します。このようにすると、期限切れのセッション ID によって Cookie なしのセッション要求が行われたときに、新しいセッション ID が生成されるようになります。

カスタム セッション *ID*

SessionID 値の提供と検証を行うために、カスタムクラスを実装できます。このためには、**SessionIDManager** クラスを継承するクラスを作成し、**CreateSessionID** メソッドおよび **Validate** メソッドを独自の実装でオーバーライドします。

ISessionIDManager インターフェイスを実装するクラスを作成すると、SessionIDManager クラスを置き換えることができます。たとえば、Web アプリケーションで、ISAPI フィルタを使用して ASP.NET 以外のページ (HTML ページやイメージなど) に一意識別子を関連付けることができます。この一意識別子を ASP.NET セッション状態で使用するには、カスタム SessionIDManager クラスを実装します。カスタムクラスが Cookie なしのセッション ID をサポートしている場合は、URL 内のセッション ID を送信および取得するためのソリューションを実装する必要があります。

セッション モード

ASP.NET セッション状態では、セッション変数の格納オプションがいくつかサポートされています。各オプションは、セッション状態 Mode の種類として識別されます。既定の動作では、セッション変数は ASP.NET ワーカープロセスのメモリ空間に格納されます。ただし、別のプロセス、SQL Server データベース、またはカスタムデータソースにセッション状態が格納されるように指定することもできます。アプリケーションでセッション状態を有効にしない場合は、セッションモードを Off に設定します。

セッション イベント

ASP.NET には、ユーザーセッションの管理に役立つ 2 つのイベントが用意されています。1 つは新しいセッションが開始されたときに発生する **Session_OnStart** イベントで、もう 1 つはセッションが放棄されるか有効期限が切れたときに発生する **Session_OnEnd** イベントです。セッションイベントは、ASP.NET アプリケーションの Global.asax ファイルで指定されます。

セッションの Mode プロパティが既定のモードである InProc 以外の値に設定されている場合、Session_OnEnd イベントはサポートされません。

セッション状態の設定

セッション状態は、system.web 構成セクションの **sessionState** 要素を使用して設定されます。また、@ Page ディレクティブの **EnableSessionState** 値を使用してセッション状態を設定することもできます。

sessionState 要素を使用すると、次のオプションを指定できます。

- セッションのデータの格納モード
- クライアントとサーバー間でのセッション ID 値の送信方法
- セッションの Timeout 値
- セッションの Mode 設定に基づくサポート値

アプリケーションを SQLServer セッション モードに構成する sessionState 要素の例を次に示します。Timeout 値を 30 分に設定し、セッション ID を URL に格納することを指定しています。

```
<sessionState mode="SQLServer"
  cookieless="true "
  regenerateExpiredSessionId="true "
  timeout="30"
  sqlConnectionString="Data Source=MySqlServer;Integrated
Security=SSPI;"
  stateNetworkTimeout="30"/>
```

セッション状態モードを Off に設定することによって、アプリケーションのセッション状態を無効にできます。アプリケーションの特定ページだけのセッション状態を無効にする場合は、@ Page ディレクティブの EnableSessionState 値を false に設定します。また、EnableSessionState 値を ReadOnly に設定して、セッション変数に読み取り専用アクセスを指定することもできます。

現在の要求とセッション状態

ASP.NET セッション状態へのアクセスは、セッションごとに排他的です。つまり、2 人のユーザーが同時に要求を行った場合、別々のセッションへの同時アクセスは許可されます。ただし、2 つの同時要求が (同じ SessionID 値を使用して) 同じセッションに対して行われた場合、最初の要求がセッション情報への排他的アクセスを取得します。2 つ目の要求は、最初の要求が完了した時点で実行されます。最初の要求がロックタイムアウトを経過したことによって情報への排他的なロックが解除された場合も、2 つ目のセッションがアクセスを取得します。@ Page ページディレクティブの EnableSessionState 値が ReadOnly に設定されている場合、読み取り専用のセッション情報を要求しても、セッションデータへの排他的ロックは生じません。ただし、セッションデータに対する読み取り専用要求は、セッションデータの読み取り/書き込み要求によるロックが解除されるまで、待機する必要があります。

アプリケーション状態の概要

アプリケーション状態は、ASP.NET アプリケーションのすべてのクラスが使用できるデータリポジトリです。アプリケーション状態はサーバーのメモリに格納され、データベースに情報を格納して取得するより高速です。個々のユーザーセッションに適用されるセッション状態とは異なり、アプリケーション状態はすべてのユーザーとセッションに適用されます。したがって、アプリケーション状態はユーザー間で変化しない頻繁に使用される少量のデータの格納に便利な場所です。

アプリケーション状態の使用

アプリケーション状態は、**HttpApplicationState** クラスのインスタンスに格納されます。このクラスは、オブジェクトのキーと値の辞書を公開します。

HttpApplicationState インスタンスは、ユーザーがアプリケーションの任意の URL リソースに最初にアクセスする際に作成されます。多くの場合、HttpApplicationState クラスには、**HttpContext** クラスの **Application** プロパティを使用してアクセスします。

アプリケーション状態は、次の 2 つの方法で使用できます。Contents コレクションに対しては、コードを使用して直接値を追加、アクセス、または削除できます。HttpApplicationState クラスには、アプリケーションのライフサイクル中にいつでもアクセスできます。ただし、アプリケーション状態のデータは、一般にアプリケーションの起動時に読み込むと便利です。そのためには、アプリケーション状態を読み込むコードを Global.asax ファイルの **Application_Start** メソッドに記述します。

Web アプリケーションの Global.asax ファイルの <object runat="server"> 宣言を使用してオブジェクトを **StaticObjects** コレクションに追加することもできます。この方法で定義したアプリケーション状態は、アプリケーションのすべてのコードからアクセスできます。アプリケーション状態の値のためのオブジェクト宣言の例を次に示します。

```
<object runat="server" scope="application" ID="MyInfo"
    PROGID="MSWC.MYINFO">
</object>
```

オブジェクトを StaticObjects コレクションに追加できるのは、Global.asax ファイル内だけです。コレクションは、コードを使用してオブジェクトを直接追加すると、**NotSupportedException** をスローします。

アプリケーション状態に格納されているオブジェクトのメンバには、Application コレクションを参照せずにアクセスできます。アプリケーション状態の StaticObjects コレクションで定義されているオブジェクトのメンバを参照する方法のコード例を次に示します。

<Visual Basic>

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs)
```

```
    Label1.Text = MyInfo.Title
End Sub
```

<C#>

```
protected void Page_Load(Object sender, EventArgs e)
{
    Label1.Text = MyInfo.Title;
}
```

アプリケーション状態の考慮事項

アプリケーション状態を使用する場合は、次のような重要な考慮事項に注意する必要があります。

- **リソース** —— アプリケーション状態はメモリに格納されるため、ディスクまたはデータベースに保存されるデータよりはるかに高速です。ただし、アプリケーション状態に大きなデータ ブロックを格納すると、サーバーのメモリが不足し、メモリがディスクにページングされることがあります。大量のアプリケーションデータを格納するために、アプリケーション状態の代わりに ASP.NET のキャッシュ機構を使用することもできます。ASP.NET のキャッシュもメモリにデータを格納するので非常に高速ですが、ASP.NET はキャッシュをアクティブに管理してメモリが不足すると項目を削除します。
- **揮発性** —— アプリケーション状態はサーバーのメモリに格納されるため、アプリケーションが停止または再起動すると消失します。たとえば、Web.config ファイルが変更されるとアプリケーションが再起動され、アプリケーション状態の値をデータベースなどの不揮発性の記憶媒体に保存していなければ、すべてのアプリケーション状態が消失します。
- **スケーラビリティ** —— アプリケーション状態は同じアプリケーションにサービスを提供する複数のサーバー間 (Web ファームなど)、または同じサーバーの同じアプリケーションにサービスを提供する複数のワーカープロセス間 (Web ガーデンなど) で共有されません。したがって、複数のサーバーまたはプロセスに同じアプリケーション状態のデータがあることを前提にすることはできません。アプリケーションがマルチプロセッサ環境またはマルチサーバー環境で実行される場合、アプリケーション全体で一貫性を保持する必要があるデータには、データベースなどのよりスケーラブルな方法を使用することを考慮してください。
- **同時実行** —— アプリケーション状態はフリースレッドのため、アプリケーション状態のデータには同時に多くのスレッドがアクセスできます。したがって、アプリケーション状態のデータを更新するタイミングが重要になります。組み込みの同期機能をサポートすると、スレッド セーフな方法でアプリケーション状態のデータを更新できます。**Lock** メソッドと **UnLock** メソッドを使用すると、一度に 1 つのソースだけが書き込むことができるようにデータをロックすることによってデータの整合性を維持できます。Global.asax ファイルの **Application_Start** メソッドでアプリケーション状態の値を初期化すると、同時実行の問題の発生を軽減できます。

プロファイル プロパティの概要

多くのアプリケーションでは、ユーザー固有の情報を格納して使用します。ユーザーがサイトを訪問したときは、格納した情報を使用してユーザーに Web アプリケーションのパーソナル化されたバージョンを表示できます。アプリケーションをパーソナル化するには一連の要素が必要です。一意のユーザー ID を使用して情報を格納し、再訪問したときにユーザーを認識し、必要に応じてユーザーの情報をフェッチする必要があります。**ASP.NET プロファイル機能**を使用すると、これらのすべてのタスクを実行できるため、アプリケーションを簡略化できます。

ASP.NET プロファイル機能は、情報を個々のユーザーに関連付けて永続的な形式で格納します。ASP.NET プロファイルを使用すると、ユーザー情報を管理するために独自のデータベースを作成および保守する必要がありません。また、ASP.NET プロファイル機能では、アプリケーション内の任意の場所からアクセスできる厳密に型指定された API を使用してユーザー情報を利用できます。

プロファイルを使用すると、どのような型のオブジェクトでも格納できます。このプロファイル機能には、ほぼすべての型のデータを定義および保持しつつ、タイプ セーフな方法でデータを使用できる汎用ストレージ機能が用意されています。

ASP.NET プロファイルの動作のしくみ

プロファイルを使用するには、まず ASP.NET Web アプリケーションの構成ファイルを変更してプロファイルを有効にします。構成の一環として、プロファイルデータの格納と取得を行う低レベルのタスクを実行する基本クラスであるプロファイルプロバイダも指定します。SQL Server にプロファイルデータを格納する .NET Framework のプロファイル プロバイダを使用するか、または、独自のプロファイル プロバイダを作成して使用します。選択したデータベースに接続する **SqlProfileProvider** のインスタンスを指定するか、またはローカルの Web サーバーにプロファイル データを格納する SqlProfileProvider の既定のインスタンスを使用します。

プロファイルの機能は、値を使用するプロパティの一覧を定義して構成します。たとえば、アプリケーションが天気予報などの地域固有情報を提供する場合は、ユーザーの郵便番号を格納します。その場合は、構成ファイルに **PostalCode** というプロファイルプロパティを定義します。構成ファイルの profile セクションは次のようになります。

```
<profile>
  <properties>
    <add name="PostalCode" />
  </properties>
</profile>
```

アプリケーションが実行されると、ASP.NET は **ProfileBase** クラスを継承し、動的に生成される **ProfileCommon** クラスを作成します。この動的な ProfileCommon クラスには、アプリケーシ

ョンの構成で指定するプロファイルプロパティの定義から作成されたプロパティが含まれます。この動的な ProfileCommon クラスのインスタンスは、現在の HttpContext の Profile プロパティの値に設定され、アプリケーションの各ページで使用できるようになります。

アプリケーションでは、格納する値を収集して、定義されているプロファイルプロパティに割り当てます。たとえば、アプリケーションのホームページにユーザーが郵便番号を入力するためのテキストボックスを作成します。次の例のように、ユーザーが郵便番号を入力したときに、現在のユーザーの値を格納するように Profile プロパティを設定します。

<Visual Basic>

```
Profile.PostalCode = txtPostalCode.Text
```

<C#>

```
Profile.PostalCode = txtPostalCode.Text;
```

Profile.PostalCode に値を設定すると、その値は自動的に現在のユーザーに対して格納されます。現在のユーザーを特定したり、明示的に値をデータベースに格納したりするためにコードを記述する必要はありません。これらのタスクはプロファイル機能が実行します。

値を使用する場合は、値を設定した場合とほとんど同じ方法で取得できます。GetWeatherInfo という架空の関数を呼び出し、プロファイルに格納されている現在のユーザーの郵便番号を渡すコード例を次に示します。

<Visual Basic>

```
weatherInfo = GetWeatherInfo( Profile.PostalCode )
```

<C#>

```
weatherInfo = GetWeatherInfo( Profile.PostalCode );
```

ユーザーを明示的に指定したり、データベースの検索を実行したりする必要はありません。プロファイルからプロパティ値を取得するだけで、ASP.NET は現在のユーザーを識別し、永続的なプロファイル ストアから値を探して必要なアクションを実行します。

ASP.NET の状態管理に関する推奨事項

状態管理は、同じページまたは異なるページに対する複数の要求にわたって状態およびページ情報を保持するプロセスです。すべての HTTP ベースのテクノロジについて言えることですが、Web フォームページは状態を持ちません。つまり、一連の要求がすべて同じクライアントからの要求かどうか、または 1 つのブラウザーインスタンスがページまたはサイトをまだアクティブに表示しているかどうかについて、自動的には示されません。また、ページはサーバーへのラウンドトリップごとに破棄されて再作成されるため、ページ情報が 1 つのページの有効期間を超えて存在することはありません。

ASP.NET には、サーバーへのラウンドトリップ間で状態を保持するための、いくつかの方法が用意されています。これらのオプションのどれを選択するかはアプリケーションに大きく依存するため、次の基準に基づいて選択する必要があります。

- どの程度の量の情報を格納する必要があるか
- クライアントが永続的な Cookie またはメモリ内の Cookie を受け入れるか
- クライアントとサーバーのどちらに情報を格納するか
- 情報の機密性は高いか
- アプリケーションにおけるパフォーマンスと帯域幅の基準は何か
- 対象とするブラウザーおよびデバイスの機能はどのようなものか
- ユーザーごとに情報を格納する必要があるか
- どれだけの期間、情報を格納する必要があるか
- Web ファーム (複数のサーバー)、Web ガーデン (1 台のコンピュータ上での複数のプロセス)、またはアプリケーションを提供する単一のプロセスのうちどれを使用するか

クライアント側の状態管理オプション

クライアント側のオプションを使用してページ情報を格納する場合は、サーバーリソースを使用しません。これらのオプションでは、通常、最小限のセキュリティしか提供されませんが、サーバーリソースに対する要求が大きくないため、サーバーのパフォーマンスは高速になります。ただし、クライアントに格納される情報を送信する必要があるため、この方法で格納できる情報の量には実質的な制限があります。

ASP.NET がサポートするクライアント側の**状態管理オプション**を順に説明していきます。

ビューステート

Web フォームページには、同じページに対する複数の要求の間で自動的に値を保持するための組み込み構造として、ViewState プロパティが用意されています。**ビューステート**はページの隠しフィールドとして保持されます。

ビューステートを使用することにより、ページがポストされて戻ってくるときに、ラウンドトリップ間で独自のページ固有の値を保持できます。たとえば、アプリケーションがユーザー固有の情報 (ページで使用されるが必ずしもコントロールの一部ではない情報) を保持している場合は、その情報をビューステートに格納できます。

ビューステートを使用する場合の長所

- サーバーリソースが不要
- 簡単な実装
- 強化されたセキュリティ機能

ビューステートを使用する場合の短所

- ビューステートはページ自体に格納されるため、大きな値を格納すると、ページの表示とポストにかかる時間が増大します。
- モバイルデバイスには、大量のビューステートデータを格納するためのメモリ容量がありません。
- ビューステートにはデータがハッシュ処理されて格納されますが、改変される可能性があります。

コントロールステート

ASP.NET のページフレームワークでは、サーバーへのトリップ間でカスタムコントロールデータを格納する方法として、**ControlState** プロパティが用意されます。たとえば、意図したとおりにコントロールを動作させるために、異なる情報を表示するさまざまなタブを持つカスタムコントロールを記述した場合、このコントロールは、ラウンドトリップ間でどのタブが選択されているかを知る必要があります。この目的にはビューステートを使用できますが、効果的にコントロールを中断するために、開発者がページレベルでビューステートをオフにする可能性があります。コントロールステートはビューステートのようにオフにできないため、この方法を使用するとコントロールステートデータを格納するときの信頼性が向上します。

コントロールステートを使用する場合の長所

- 既定では、コントロールステートはページ上の隠しフィールドに格納されます。
- コントロールステートはビューステートのようにオフにできないため、コントロールステートを管理するときの信頼性が向上します。
- カスタムアダプタを記述して、コントロールステートデータの格納方法と格納場所を制御できます。

コントロールステートを使用する場合の短所

- プログラミングが必要

隠しフィールド

ページの状態を維持する方法の 1 つとして、ページ固有の情報をページ上の**隠しフィールド**に格納できます。

隠しフィールドを使用する場合は、頻繁に変更される小量のデータだけをクライアント上に格納するのが最善です。

隠しフィールドを使用する場合の長所

- サーバーリソースが不要
- 幅広いサポート
- 簡単な実装

隠しフィールドを使用する場合の短所

- 改変される可能性があります。
- 多様なデータ型をサポートしません。
- ページ自体に格納されるため、大きな値を格納すると、ページの表示とポストにかかる時間が増大します。
- 格納するデータが大量になると、プロキシやファイアウォールによってはデータを含むページにアクセスできません。

Cookie

Cookie は、頻繁に変更される小量の情報をクライアント上に格納するときに便利です。この情報は、要求と一緒にサーバーに送信されます。

Cookie を使用する場合の長所

- ライアント上での有効期限規則に従って、ブラウザ セッションの終了時に期限切れにしたり、無期限に存在するようにしたりできます。
- サーバーリソースが不要です
- 単純なキーと値のペアを含む、サイズの小さなテキストベースの構造体です。
- クライアント上で最大のデータ永続性が得られます。

Cookie を使用する場合の短所

- ほとんどのブラウザでは Cookie のサイズが 4,096 バイトに制限されています（新しいバージョンのブラウザやクライアントデバイスでは、8,192 バイトの Cookie をサポートするのが一般的になっています）。
- 一部のユーザーは、ブラウザまたはクライアントデバイスで Cookie を受け入れる機能を無効にして、この機能を制限しています。
- 改変される可能性があります。Cookie に依存するアプリケーションで障害が発生することもあります。

クエリ文字列

クエリ文字列は、ページの URL の最後に追加される情報です。

クエリ文字列を使用して、ページにデータを返したり、URL を通じて他のページに送信したりできます。クエリ文字列は、いくつかの状態情報を保持するための単純な方法ですが、いくつかの制限があります。たとえば、製品番号を他のページに渡して処理する場合など、ページ間で情報を渡すためにはクエリ文字列を使用すると簡単です。

クエリ文字列を使用する場合の長所

- サーバーリソースが不要です。

- ほとんどすべてのブラウザおよびクライアントデバイスが、値を渡すためのクエリ文字列の使用をサポートしています。
- 実装が簡単です。

クエリ文字列を使用する場合の短所

- クエリ文字列に含まれる情報は、ブラウザーのユーザーインターフェイスを通して直接ユーザーに表示されます。
- 一部のブラウザーおよびクライアントデバイスでは、URL の長さが 2,083 文字に制限されています。

クライアント側での状態管理方法のまとめ

ASP.NET で使用できるクライアント側の状態管理オプションの一覧と各オプションを使用する際の推奨事項を次の表に示します。

表. サーバー側の状態管理オプション

状態管理オプション	推奨される使用法
ビューステート	ポストバックされるページに対して小量の情報を格納する必要がある場合に使用します。ViewState プロパティを使用すると、基本セキュリティを備えた機能が提供されます。
コントロールステート	サーバーへのラウンドトリップ間で、コントロールに対して小量の状態情報を格納する必要がある場合に使用します。
隠しフィールド	ポストバックされるページ、または他のページに返されるページに対して小量の情報を格納する必要があり、セキュリティが問題とならない場合に使用します。
Cookie	クライアント上に小量の情報を格納する必要があり、セキュリティが問題とならない場合に使用します。
クエリ文字列	1 つのページから別のページに小量の情報を送信する必要があり、セキュリティが問題とならない場合に使用します。

ページ情報を格納するためのサーバー側のオプションは、通常、クライアント側のオプションよりも高いセキュリティを提供します。しかし、より多くの Web サーバーリソースを使用するため、情報ストアのサイズが大きいと、スケーラビリティ上の問題につながる可能性があります。ASP.NET には、サーバー側での状態管理を実装するための、いくつかのオプションが用意されています。

サーバー側の状態管理オプション

ASP.NET がサポートするサーバー側の状態管理オプションを順に説明します。

アプリケーション状態

ASP.NET には、アプリケーション全体で使用できるグローバルなアプリケーション固有の情報を格納する方法として、**HttpApplicationState** クラスによるアプリケーション状態が用意されています。アプリケーション状態変数は、実際には、ASP.NET アプリケーションのグローバル変数です。

アプリケーション状態には、アプリケーション固有の値を格納できます。アプリケーション状態は、サーバーによって管理されます。

アプリケーション状態変数に挿入するデータとして理想的なデータは、複数のセッションによって共有され、頻繁に変更されないデータです。

アプリケーション状態を使用する場合の長所

- 実装が簡単です。
- アプリケーション状態には、アプリケーションのすべてのページからアクセスできます。

アプリケーション状態を使用する場合の短所

- アプリケーション状態に格納された変数は、アプリケーションが動作している特定のプロセスに対してだけグローバルであり、各アプリケーションプロセスで値が異なる場合があります。
- データの永続性の制限があります。
- アプリケーション状態では、サーバーのメモリが使用されます。

セッション状態

ASP.NET には、セッション内だけで使用できるセッション固有の情報を格納する方法として、セッション状態を利用できます。セッション状態は、**HttpSessionState** クラスによって実現されます。ASP.NET のセッション状態は、制限された時間ウィンドウ内での同一ブラウザーからの要求を 1 つのセッションとして識別し、変数値をそのセッションの存続期間中保持する機能を提供します。

セッション状態には、セッション固有の値およびオブジェクトを格納できます。セッション状態は、サーバーによって管理され、ブラウザーまたはクライアントデバイスで使用できます。セッション状態変数に格納するデータとして理想的なデータは、個別のセッションに固有の、存続期間が短い重要情報です。

セッション状態を使用する場合の長所

- 実装が簡単です。
- セッション管理イベントは、アプリケーションで発生させて使用できます。
- セッション状態変数に配置されたデータは、インターネットインフォメーションサービス (IIS: Internet Information Services) の再起動やワーカープロセスの再起動があっても失われずに保持されます。

- セッション状態は、マルチコンピュータ構成とマルチプロセス構成の両方で使用できるため、スケーラビリティを最適化できます。
- HTTP Cookie をサポートしないブラウザーでも動作します。

独自のセッション状態プロバイダを記述することによってカスタマイズおよび拡張できます。

セッション状態を使用する場合の短所

- セッション状態変数は、削除されるか置き換えられるまでメモリに保持されるため、サーバーのパフォーマンスを低下させる場合があります。

プロファイル プロパティ

プロファイルプロパティ機能では、プロファイルデータが永続的な形式で格納され、個々のユーザーに関連付けられる ASP.NET プロファイルを使用します。ASP.NET プロファイルを使用すると、独自のデータベースを作成したり管理したりする手間をかけずにユーザー情報を容易に管理できます。また、このプロファイルでは、アプリケーション内のどこからでもアクセスできる、厳密に型指定された API を使ってユーザー情報を利用できます。

プロファイル プロパティを使用する場合の長所

- プロファイルプロパティに配置されたデータは、IIS の再起動やワーカープロセスの再起動があっても失われずに保持されます。
- マルチコンピュータ構成とマルチプロセス構成の両方で使用できるため、スケーラビリティを最適化できます。
- 拡張性があります。

プロファイル プロパティを使用する場合の短所

- データをメモリに格納する代わりに、データストアに保持します。このため、一般に、セッション状態を使用する場合よりも遅くなります。
- セッション状態とは異なり、プロファイルプロパティ機能ではかなりの構成作業が必要になります。
- 一定の保守作業が必要です。プロファイルデータは不揮発性ストレージに保持されるため、データが古くなる前に、アプリケーションが適切なクリーンアップ機構を呼び出すように構成する必要があります。

データベースのサポート

Web サイト上で状態を管理するために、場合によってはデータベースのサポートが必要になります。通常、データベースのサポートは Cookie またはセッション状態を使って実施します。たとえば、次のような理由により、電子商取引 Web サイトではリレーショナルデータベースを使用して状態情報を保持するのが一般的です。

Cookie がサポートされたデータベース Web サイトの一般的な機能を次に示します。

- **セキュリティ** —— ユーザーは、サイトのログオンページでアカウント名とパスワードを入力します。サイトのインフラストラクチャは、データベース内でログオン値を照会し、そのユーザーがサイトを利用する権限を持っているかどうかを確認します。データベースでユーザー情報が確認されると、Web サイトはそのユーザーの一意な ID を含む有効な Cookie をクライアントコンピュータに配布します。これにより、サイトはユーザーにアクセスを許可します。
- **パーソナル化** —— セキュリティ情報を配置すると、サイトはクライアントコンピュータ上の Cookie を読み取って各ユーザーを識別できるようになります。サイトでは、通常、一意な ID によって識別されるユーザーの基本設定を記述したデータベースに情報を格納しています。このリレーションシップをパーソナル化と呼びます。サイトは、Cookie に含まれる一意な ID を使用してユーザーの好みを調べ、ユーザー固有の要求に関連したコンテンツおよび情報を配置してユーザーの好みに対応します。
- **一貫性** —— 商取引 Web サイトを作成したときに、サイト上の商品およびサービスに対する購買のトランザクションレコードを保持することが必要な場合があります。これにより、購買トランザクションが完了しているかどうかを確認したり、購買トランザクションに障害が発生した場合に実行するアクションを決定したりできます。
- **データ マイニング** —— サイトの使用状況、利用者、または製品トランザクションに関する情報をデータベースに確実に格納できます。たとえば、ビジネス開発部門はサイトで収集されたデータを使用して、翌年の製品ラインや販売ポリシーを決定できます。

データベースを使用した状態管理の長所

- データベースへのアクセスには、厳格な認証および承認が必要です。
- データベースには大量の情報を格納できます。
- データベースには情報を長期間にわたって格納でき、Web サーバーの可用性に左右されません。
- データベースには、トリガ、参照整合性、トランザクションなど、質の高いデータを維持するための各種の機能が備わっています。トランザクションに関する情報を (セッション状態ではなく) データベースに保持することにより、エラーの回復がより簡単になります。
- データベースに格納されたデータは、幅広い範囲の情報処理ツールによってアクセスできます。
- 幅広い範囲のデータベースツールが使用でき、多くのカスタム構成を使用できます。

データベースを使用した状態管理の短所

- データベースを使用して状態管理をサポートすると、ハードウェア構成およびソフトウェア構成の複雑さが増します。
- リレーショナルデータモデルの構造が良くないと、スケーラビリティ上の問題につながる場合があります。また、データベースに対して多くのクエリを使用すると、サーバーのパフォーマンスに悪影響を及ぼす場合があります。

サーバー側での状態管理方法のまとめ

ASP.NET で使用できるサーバー側の状態管理オプションの一覧と、各オプションを使用する際の推奨事項を次の表に示します。

表. 状態管理オプションの推奨される使用法

状態管理オプション	推奨される使用法
アプリケーション状態	多くのユーザーによって使用され、頻繁に変更されないグローバルな情報を格納する必要があり、セキュリティが問題とならない場合に使用します。アプリケーション状態には、大量の情報を格納しないでください。
セッション状態	個別のセッションに固有の存続期間の短い情報を格納する必要があり、セキュリティが問題となる場合に使用します。セッション状態には、大量の情報を格納しないでください。セッション状態オブジェクトは、アプリケーション内のすべてのセッションに対して作成され、その有効期間の間保持されることに注意してください。多くのユーザーが利用するアプリケーションでは、これによって多くのサーバーリソースが占有され、スケーラビリティに影響する場合があります。
プロファイル プロパティ	ユーザー セッションの有効期限が切れた後も保持され、それ以降アプリケーションにアクセスした際に再取得する必要があるユーザー固有の情報を格納する場合に使用します。
データベース サポート	大量の情報を格納する場合、トランザクションを管理する場合、または、アプリケーションおよびセッションの再起動後も情報を保持する必要がある場合や、データマイニングに関心があり、セキュリティが問題となる場合に使用します。

パフォーマンスとスケーラビリティ

ASP.NET パフォーマンスの向上とは

パフォーマンス目標値を達成する ASP.NET アプリケーションを構築するには、ボトルネックが発生しやすい場所とその原因、およびその予防策をとるための手順について把握しておく必要があります。また、健全なアーキテクチャと設計、ベストプラクティスに基づくコーディング、そして最適化したプラットフォームと .NET Framework 構成を組み合わせる必要があります。

ここでは、ASP.NET の設計に関する一連のガイドラインを提示します。このガイドラインに従うことにより、大きなコストを伴う再エンジニアリングでしか修正できないようなパフォーマンス上の問題をトップレベルの設計で回避することができます。さらに、ASP.NET のパフォーマンスに関する主要問題を説明します。これらの問題には、ページとコントロールの問題、キャッシング、リソース管理、セッション状態とビュー ステートの問題、スレッド処理、例外管理と文字列管理、COM 相互運用、などがあります。

パフォーマンスとスケーラビリティに関する問題

ASP.NET アプリケーションのパフォーマンスとスケーラビリティに悪影響を及ぼし得る主な問題について、以下で一通り取り上げます。この章の以降の節では、ここで掲げる問題を予防または解決するための方策や技術情報を提示します。

- **リソース アフィニティ** —— リソースアフィニティは、サーバーの追加を妨げることや、CPU やメモリの追加によるメリットを小さくすることがあります。リソースアフィニティは、コードが特定のスレッド、CPU、コンポーネント、インスタンス、またはサーバーを必要とする場合に発生します。
- **過度のアロケーション** —— メモリを要求ごとに過度に割り当てるアプリケーションは、メモリを消費し、必要以上のガベージコレクションを発生させます。ガベージコレクションの回数が増えることにより、CPU 利用度は増加します。このような過度のアロケーションは、一時アロケーションを原因としている場合があります。たとえば、タイトなループで + = 演算子を使う、過度の文字列連結が原因となる場合もあります。
- **希少なリソースを共有しない** —— Dispose メソッドや Close メソッドを呼び出してデータベースコネクションなどのリソースを開放しないと、リソース不足につながりかねません。リソースを閉じるか破棄することにより、リソースの使用を効率化することができます。
- **処理のブロック** —— ASP.NET 要求を処理する 1 つのスレッドは、下位呼び出しの戻り値を待つ間、ブロックされて新たなユーザー要求に対応できなくなります。所要時間の長いストアドプロシージャやリモートオブジェクトの呼び出しはスレッドを長時間ブロックする場合があります。

- **スレッドの誤使用** —— スレッドを要求ごとに作成すると、回避可能なスレッド初期化コストが発生します。また、シングル スレッド アパートメント (STA) COM オブジェクトを不適切に使うと、複数の要求がキューに入れられる場合があります。この場合、パフォーマンスは低下し、スケーラビリティ上の問題が発生します。
- **遅延バインディング** —— 遅延バインディングは、実行コードを特定し、ロードするために、実行時に追加的なインストラクションを要求します。対象コードがマネージコードでもアンマネージコードでも、これらのインストラクションは避けるべきです。
- **COM 相互運用の誤使用** —— COM 相互運用は通常、非常に効率的です。しかし、多くの要因が、そのパフォーマンスに影響を及ぼします。これらの要因には、マネージ / アンマネージ境界やアパートメント境界をまたいで渡る、引数のサイズと型などがあります。アパートメント境界をまたぐと、大きな負担を伴うスレッド切り替えが必要となる場合があります。
- **大きなページ** —— ページサイズは、ページ上のコントロールの数と型の影響を受けます。また、ページのレンダリングに使うデータと画像にも影響されます。ネットワークに多くのデータを送るほど、消費する帯域は大きくなります。高いレベルの帯域を消費するほど、ボトルネックは発生しやすくなります。
- **不適切なデータ キャッシュ** —— 静的データをキャッシュしない、アイテムを流し出すために過剰に多いデータをキャッシュする、アプリケーション全体で使われるデータでなくユーザーデータをキャッシュする、頻繁に使われないアイテムをキャッシュする、といったアプローチは、システムのパフォーマンスとスケーラビリティを制限することになりかねません。
- **不適切な出力キャッシュ** —— 出力キャッシュを使用していない場合、または不適切に使用している場合、Web サーバーに回避可能な負担がかかる場合があります。
- **非効率なレンダリング** —— HTML コードやサーバーコードを散在させる、ページポストバックで不必要な初期化コードを実行する、データの遅延バインディング、といったアプローチは、大きなレンダリングオーバーヘッドの原因となり得ます。このオーバーヘッドにより、感覚的および実質的なページパフォーマンスは低下しかねません。

設計上の考慮事項

高いパフォーマンスを備えた ASP.NET アプリケーションの構築は、設計時にパフォーマンスを考慮に入れれば非常に簡単になります。プロジェクト開始時からパフォーマンスプランを作成するようにしてください。決して、パフォーマンスをアプリケーション構築後に対処すべき問題としてとらえないでください。また、反復的な開発プロセスをとり、反復の間に定期的な計測を織り込むようにしてください。

ベストプラクティスに基づいた以下の設計ガイドラインに従えば、高いパフォーマンスを備えた Web アプリケーションを作成できる可能性は、きわめて高くなります。

以下に、設計ガイドラインを挙げていきましょう。

セキュリティとパフォーマンスを考慮する

認証スキームの選択は、アプリケーションのパフォーマンスとスケーラビリティに影響を及ぼす場合があります。以下の問題について考慮する必要があります。

- **ID** —— 使用する ID と、アプリケーションを通してそれを受け渡す方法について考慮してください。下位のリソースにアクセスするために、ASP.NET プロセス ID や、他のサービス ID を使用できます。または、偽装を許可し、オリジナルの呼び出し元の ID を渡すこともできます。Microsoft SQL Server にアクセスする場合は、SQL 認証を利用することも可能です。ただし、SQL 認証では、データベース接続文字列に資格情報を格納する必要があります。このアプローチは、セキュリティの観点からは望ましくありません。データベースなどの共有リソースに 1 つの ID を使って接続するとき、接続プーリングによるメリットを受けることができます。接続プーリングは、スケーラビリティを大きく向上させます。偽装を使ってオリジナルの呼び出し元の ID を受け渡す場合は、接続プーリングによる効率性のメリットを得られません。また、複数の個別ユーザーアカウントに対するアクセスコントロールを設定しなければなりません。これらの理由により、下位データベースへの接続には 1 つの信頼済み ID を使うのがベストです。
- **資格情報の管理** —— 資格情報の管理方法を考慮してください。アプリケーションで資格情報をデータベースに格納して検証するのか、資格情報を Active Directory ディレクトリサービスに格納する、オペレーションシステムの提供する認証メカニズムを使用するのかを、決めなければなりません。また、アプリケーションがサポートする同時アクセスユーザー数と、資格情報ストア (データベースまたは Active Directory) が対応するユーザー数も決めるべきです。アプリケーションのキャパシティプランニングを実行し、システムが予想負荷に対応できるかを判断する必要があります。
- **資格情報の保護** —— ネットワークへ送る視覚情報を暗号化および復号化すると、追加的な処理サイクルが発生します。Windows フォーム認証や SQL 認証などの認証スキームを使う場合、資格情報はクリアテキストで流れ、ネットワーク盗聴者からアクセス可能となってしまいます。これらのケースで、ネットワークへ送る資格情報の保護は、どの程度重要でしょうか？ NTLM や Kerberos プロトコルなど、オペレーティングシステムが提供する認証スキームを選択可能か、判断してください。これらの場合、暗号化オーバーヘッドを避けるため、資格情報はネットワークへ送られません。
- **暗号法** —— アプリケーションにおいて、送信中の情報を第三者による操作から保護することだけが必要なら、鍵付きハッシュを使用できます。これは、普通のハッシュに比べれば高くつきますが、暗号化は不要になります。ネットワークへ送るデータを隠さなければならない場合は、暗号化、そしてデータの有効性を確保するため、おそらく鍵付きハッシュが必要になります。送信側と受信側が鍵を共有できるなら、対称暗号化を採用することで非対称暗号化に比べて高いパフォーマンスを得ることができます。鍵のサイズが大きいほど、暗号化の強度は増しますが、パフォーマンスはサイズの小さい鍵を使う場合に比べて低下します。設計時には、このようなパフォーマンスとセキュリティのバランスに注意しなければなりません。

アプリケーションを論理的に分割する

レイヤリングにより、アプリケーションロジックをプレゼンテーション層、ビジネス層、データアクセス層の各論理階層に分けてください。これにより、コードの保守性を高めることができます。また、各論理階層のパフォーマンスを別々に監視し最適化することが可能になります。明確な論理パーティションを設けることにより、アプリケーションのスケーリングのための選択肢も広がります。コードビハインドファイルのコード量を減らすことを心掛け、保守性とスケーラビリティを向上させてください。

論理分割と物理展開を混同しないでください。論理分割により、プレゼンテーション ロジックとビジネスロジックを同じサーバー上に配置し、Web ファームのサーバーをまたいで同一ロジックが複数存在する状態にするか、あるいはロジックを物理的に分かれたサーバーにインストールするかを決められるようになります。注意すべきポイントは、リモート呼び出しは遅延コストを生じさせ、遅延の程度はレイヤ間の距離が長いほど大きくなるということです。

たとえば、プロセス内呼び出しが最も速く、次に同じコンピュータ内のプロセスをまたいだ呼び出し、そしてリモートネットワーク呼び出し、という順に続きます。可能な限り、論理パーティションで分けたロジック同士を、近接させてください。パフォーマンスを最適化するためには、ビジネスロジックとデータアクセスロジックを、Web サーバー上のアプリケーションの Bin ディレクトリに配置すべきです。

アフィニティを評価する

アフィニティにより、パフォーマンスを向上させることができます。しかし、アフィニティはスケーラビリティに悪影響を及ぼす場合があります。リソースアフィニティを発生させる主なコーディング手法には以下のものなどがあります。

- **インプロセス セッション状態を使う** —— サーバーアフィニティを避けるには、SQL Server データベースにおいて ASP.NET セッション状態をプロセス外で保守するか、リモートマシンで動作するアウトプロセスセッション状態サービスを使ってください。または、ステートレスアプリケーションを設計するか、セッション状態をクライアントに格納し、要求ごとにそれを渡すようにしてください。
- **コンピュータ独自の暗号化鍵を使う** —— データベースで、データの暗号化のためにコンピュータ独自の暗号化鍵を使うと、アプリケーションは Web ファームで作業できなくなります。これは、通常の暗号化データは、複数の Web サーバーからアクセス可能でなければならないためです。共有対称鍵の暗号化にコンピュータ独自鍵を使うのが、より望ましいアプローチです。この場合、暗号化データのデータベースへの格納には、共有対称鍵を使用します。

ラウンド トリップ数を減らす

ASP.NET の提供する以下のアプローチや機能を使い、Web サーバー・ブラウザー間、および Web サーバー・下位システム間の**ラウンドトリップ数**を減らしてください。

- **HttpResponse.IsClientConnected** —— 要求の処理および負荷の大きいサーバー側処理の実行の前に、クライアントがまだ接続されているかを確認するのに、HttpResponse.IsClientConnected プロパティを使うことを検討してください。ただし、IIS 5.0 では、この呼び出しはプロセス外へ発行しなければならない場合もあり、非常に大きな負担を伴うこともあります。このプロパティを使うなら、実質的なメリットを得られるかを計測してください。
- **キャッシング** —— アプリケーションが静的、またはほぼ静的なデータを取得、変換、レンダリングする場合、キャッシングによって余計なヒットを回避できます。
- **出力バッファリング** —— 可能な場合、出力をバッファすることによりラウンドトリップ数を減らしてください。このアプローチにより、サーバー側でのバッチ処理が可能になり、クライアントとのチャッティーな <処理規模が小さく対話数の多い> 通信を避けることができます。欠点は、ページのレンダリングは、完了するまでクライアント側から見られないという点です。対応策として、 **Response.Flush** メソッドを使用できます。このメソッドは、出力をクライアント側のポイントまで送ります。バッファ機能が無効な遅いネットワークにつながっているクライアントは、サーバーの応答時間に影響を与えることに注意してください。これは、サーバーがクライアントからの肯定応答を待たなければならないためです。クライアントからの肯定応答は、クライアントがサーバーからすべてのコンテンツを受信した後に発行されます。
- **Server.Transfer** —— 可能な限り、Response.Redirect メソッドの代わりに **Server.Transfer** メソッドを使ってください。Response.Redirect は、応答ヘッダーをクライアントへ送ります。これにより、クライアントは新しい URL により、再指定されたサーバーに新しい要求を送ります。Server.Transfer は、単純にサーバー側呼び出しをすることにより、この迂回を回避します。

Response.Redirect 呼び出しは、常に **Server.Transfer** 呼び出しに置き換え可能なわけではありません。これは、要求処理のハンドラ段階で、Server.Transfer は新しいハンドラを使用するためです。再指定中に認証チェックおよび承認チェックが必要となる場合は、Server.Transfer の代わりに **Response.Redirect** を使ってください。これら 2 つは、同等ではないためです。Response.Redirect 使用時は、ブール型の第 2 引数を受け入れる、オーバーロードしたメソッドを使うようにし、内部例外が発生しないように false を渡してください。

また、Server.Transfer は、コントロールを同じアプリケーションのページへ移す場合にのみ使えることにも注意してください。他のアプリケーションのページへ移す場合は、Response.Redirect を使わなければなりません。

所要時間の長いタスクをブロックしない

所要時間の長い処理、またはブロック処理を行う場合、以下の非同期メカニズムを用い、Web サーバーが他の受信要求を処理できるようにしてください。

- Web サービス呼び出しの間に追加的な並行処理を実行可能な場合、Web サービスやリモートオブジェクトを非同期で呼び出してください。できるだけ Web サービスへの同期 (ブロック) 呼び出しは避けてください。外への Web サービス呼び出しは、ASP.NET スレッドプールのスレッドによって実行されるためです。ブロック呼び出しは、他の受信要求の処理に利用可能なスレッドの数を減らします。
- 応答が必要でない場合は、Web メソッドやリモートオブジェクトメソッドでの OneWay 属性の使用を検討してください。この "ファイア アンド フォーゲット (一方的)" モデルにより、Web サーバーは呼び出しをした後、すぐに処理を継続できるようになります。シナリオによっては、これが最適なアプローチとなります。
- 作業をキューに入れ、完了するまでクライアントからポーリングするようにしてください。このアプローチでは、Web サーバーはコードを呼び出し、その後 Web クライアントが、作業完了の確認のためにサーバーへのポーリングを行います。

キャッシングを利用する

パフォーマンスに関連する設計上の考慮事項で最も重要なのは、おそらく、適切なキャッシングストラテジを採用することです。ASP.NET が提供する**キャッシング機能**には、出力キャッシュ、部分ページ キャッシュ、キャッシュ API などがあります。これらの機能を活かして、アプリケーションを設計してください。

キャッシングは、データアクセスやレンダリング出力のコストを減らすために利用できます。ページがデータをどのように使用し、レンダリングするかを知っておくことにより、効率的なキャッシングストラテジを設計できるようになります。キャッシングは、Web アプリケーションが常に、データベース、Web サービス、リモート アプリケーション サーバーなどのリモートリソースからのデータに依存している場合に、特に有効です。データベースに大きく依存するアプリケーションは、キャッシングによってデータベースへの負荷を小さくし、アプリケーションのスループットを向上させることで、メリットを得られる場合があります。原則として、キャッシングは同等の処理に比べて負担が小さいため、キャッシングを使うべきです。キャッシングストラテジを設計する際は、以下のことを考慮してください。

- **作成や取得が高くつくデータまたは出力を特定する** —— 作成や取得が高くつくデータまたは出力をキャッシングすれば、データ取得のためのコストを小さくすることができます。データのキャッシングにより、データベースサーバーへの負荷は小さくなります。
- **揮発性を評価する** —— キャッシングを効果的なものにするためには、静的なデータや出力、または頻繁に変更されないデータや出力を対象にするべきです。キャッシュの対象にふさわしいデータの簡単な例としては、国、都道府県、郵便番号のリストなどが挙げられます。通常、頻繁に

変更されるデータや出力は、キャッシングにあまり適していません。ただし、必要性の高さによっては、適用することも可能です。ユーザーデータのキャッシングは、原則として、ASP.NET のセッション状態ストアなどのキャッシュに精通している場合にのみ、推奨されます。

- **使用頻度を評価する** —— 頻繁に使用するデータや出力をキャッシュすれば、パフォーマンス上およびスケーラビリティ上の大きなメリットを得られるかもしれません。これらのメリットを得られるのは、やはり、静的なデータや出力、または頻繁に変更されないデータや出力をキャッシュする場合です。たとえば、定期的に変更され頻繁に使用されるデータを適切にキャッシュすれば、パフォーマンスとスケーラビリティは大きく向上する可能性があります。使用頻度が更新頻度より高いデータはキャッシング対象の候補となります。
- **揮発性データと不揮発性データを分ける** —— 操作支援やヘルプシステムなどの静的コンテンツをカプセル化し、揮発性の高いデータと別にするようにユーザーコントロールを設計してください。これにより、静的データのキャッシュが可能となり、サーバーへの負荷を小さくすることができます。
- **正しいキャッシング メカニズムを選択する** —— データのキャッシングには、さまざまな方法があります。正しい方法は、シナリオによって異なります。通常、ユーザー独自データは、Session オブジェクトに格納されます。静的ページや、多くのユーザーに提供されるパーソナライズされていないページなどの一部の動的ページは、ASP.NET の出力キャッシュや応答キャッシュによりキャッシュできます。ページの静的コンテンツは、出力キャッシュとユーザーコントロールの組み合わせにより、キャッシュできます。ASP.NET のキャッシング機能は、キャッシュの更新のための内蔵メカニズムを提供します。これは、アプリケーション状態、セッション状態、その他のキャッシング手法では提供されていません。

不要な例外を避ける

例外は、アプリケーションに大きなオーバーヘッドをもたらします。ロジックフローの制御に例外を使ってはなりません。可能な限り例外を避けるようにコードを設計してください。たとえば、ユーザー入力を検証し、例外の発生し得る既知の状況を確認してください。また、不要な処理を避けるため、早い段階で異常終了するようにコードを設計するようにしましょう。

アプリケーションが例外を処理しない場合、例外はスタックで上向きに伝達され、最終的に ASP.NET の例外ハンドラで処理されることになります。例外処理ストラテジを設計する際は、以下のことを考慮してください。

- **例外を避けるようにコードを設計する** —— ユーザー入力を検証し、例外の発生し得る既知の状況を確認してください。例外を避けるようにコードを設計してください。
- **ロジック フローを制御するために例外を使わないようにする** —— 通常のアプリケーションロジックフローを制御するために例外管理を使うことは避けてください。
- **すべての例外についてグローバル ハンドラに依存しないようにする** —— 例外が発生すると、ランタイムはスタックを操作し、探索します。ランタイムが例外ハンドラを求めてスタックを探索するほど例外処理に伴う負担は大きくなります。

- **例外は発生場所の近くでキャッチし処理する** —— 可能な限り、例外は発生場所の近くでキャッチし処理してください。これにより、スタックでの過剰かつ高くつく探索と操作を避けることができます。
- **処理できない例外をキャッチしない** —— コードが例外を処理できない場合、try / finally ブロックを使い、例外が発生しているかどうかにかかわらずリソースを閉じるようにしてください。try / finally ブロックを使うと、リソースは例外発生時に finally ブロックでクリーンアップされ、例外を適切なハンドラに入れられるようになります。
- **高くつく作業を避けるために早い段階で異常終了する** —— 依存するタスクが異常終了した場合に、高くつく、または所要時間の長い作業を回避できるようにコードを設計してください。
- **管理者のために例外を詳しくログする** —— 例外ログメカニズムを実装し、管理者や開発者が問題を特定して是正できるように、例外に関する詳しい情報を記録してください。
- **ユーザーに対する例外の詳細情報の表示を避ける** —— ユーザーに対し、詳しい例外情報を表示しないようにし、セキュリティを保つと共にクライアントへ送るデータ量を抑えてください。

実装に関する考慮事項

アプリケーションの設計からアプリケーションの展開に移る際に、ASP.NET アプリケーションの技術詳細に注意を払ってください。ASP.NET パフォーマンスの主な指標には、**応答時間**、**スループット速度**、**リソース管理**などがあります。

応答時間は、ページサイズを小さくする、サーバーコントロールへの依存度を小さくする、バッファを使ってクライアントとの不要な通信を減らす、といった方法で短縮することができます。リソースをキャッシュすることで不必要な作業を回避できます。

スループットは、スレッドを効果的に使うことによって向上させることができます。スレッドプールをチューニングし、競合を減らすと共に、スレッドをブロックしないようにしてください。スレッドをブロックすると、利用可能なワーカースレッドの数が減ってしまうためです。

不適切なリソース管理は、サーバーの CPU やメモリに対する負荷増加の原因となり得ます。リソース利用は、プールしたリソースを効果的に使う、開いたリソースを明示的に閉じる、または破棄する、文字列管理を効率的に行う、といった方法で向上させることができます。

ベストプラクティスに基づく実装ガイドラインに従えば、適切に改良したコードと、適切に設定したアプリケーションプラットフォームを使って、アプリケーションのパフォーマンスを向上させることができます。以下の各節では、ASP.NET の機能とシナリオについてのパフォーマンス上の考慮事項を説明します。

スレッド処理について

ASP.NET は、.NET スレッドプールのスレッドを使って要求を処理します。スレッドプールは、スレッド初期化コストを既に発生させたスレッド群を保守します。したがって、これらのスレッドは容易に再利用可能です。また、NET スレッドプールはセルフチューニングをします。CPU などについてリソース利用を監視するほか、新しいスレッドを加えたり必要に応じてプールのサイズを小さくしたりします。通常、スレッドをマニュアルで作成して作業を実行することは避けるべきです。代わりに、スレッドプールのスレッドを使ってください。同時に、アプリケーションが長期にわたってブロックするような操作を実行しないようにすることは重要です。このような長期にわたるブロックは、スレッドプールの枯渇や HTTP 要求の拒否に直結する場合があります。

競合を減らすための方策

競合を減らすための方策は、ASP.NET スレッドプールのチューニングを始めるための、経験に基づいた適切な手法を提供するものです。以下の条件が当てはまる場合、表で示す推奨の設定を用いることを検討してください。

- 利用可能な CPU がある。
- アプリケーションが、Web メソッドの呼び出しやファイルシステムへのアクセスなどの I/O 依存処理を実行する。
- ASP.NET Applications/Requests in Application Queue パフォーマンスカウンタが、キューに入った要求があることを示している。

表. 競合を減らすための推奨スレッド処理設定

コンフィギュレーション設定	デフォルト値 (.NET Framework 1.1)	推奨値
maxconnection	2	12 * CPU 数
maxIoThreads	20	100
maxWorkerThreads	20	100
minFreeThreads	8	88 * CPU 数
minLocalRequestFreeThreads	4	76 * CPU 数

この問題に対応するためには、Machine.config ファイルの以下のアイテムを設定する必要があります。なお、ここで掲げた推奨値は常に適用すべきものではありません。テストした上で、それぞれのシナリオに最適な設定を見つけてください。

- **maxconnection を 12 * CPU 数にセットする** —— この設定は、クライアントから初期化可能な外向きの HTTP 接続の最大数を管理するものです。この場合、ASP.NET がクライアントです。maxconnection は 12 * CPU 数にセットしてください。

- **maxIoThreads を 100 にセットする** —— この設定は、.NET スレッド プールの I/O スレッドの最大数を管理するものです。この数は、利用可能な CPU の数によって自動的に掛けられます。maxloThreads は 100 にセットしてください。
- **maxWorkerThreads を 100 にセットする** —— この設定は、スレッドプールのワーカースレッドの最大数を管理するものです。この数は、利用可能な CPU の数によって自動的に掛けられます。maxWorkerThreads は 100 にセットしてください。
- **minFreeThreads を 88 * CPU 数にセットする** —— ワーカープロセスは、スレッドプール内の利用可能なスレッド数がこの設定値を下回っている場合に、すべての受信要求をキューに入れます。この設定により、並行処理可能な要求の数を minFreeThreads から maxWorkerThreads までに効果的に制限できます。minFreeThreads は、88 * CPU 数にセットしてください。これにより、(maxWorkerThreads を 100 として) 並行処理要求数を 12 に制限します。
- **minLocalRequestFreeThreads を 76 * CPU 数にセットする** —— ワーカープロセスは、スレッドプール内の利用可能なスレッド数がこの設定値を下回っている場合に、(Web アプリケーションが要求をローカル Web サービスに送っていれば) ローカルホストからの要求をキューに入れます。この設定は minFreeThreads に似ていますが、ローカルコンピュータからのローカルホスト要求のみに適用されます。minLocalRequestFreeThreads は 76 * CPU 数に設定してください。

ASPX Web ページが Web サービスを要求ごとに複数回呼び出す場合は、以下の推奨事項を適用してください。

速い処理の求められるオペレーションには、受信要求の処理について、ASP.NET ランタイム を 12 スレッドに制限する、という推奨アプローチが最適です。この制限により、コンテキスト切り替え数も減らされます。アプリケーションが所要時間の長い呼び出しを実行する場合は、まず、「所要時間の長いタスクをブロックしない」の節で示した設計アプローチを検討してください。これを適用できない場合は、100 maxWorkerThreads から始め、minFreeThreads はデフォルトのままにしてください。このシナリオでは、要求はシリアル化されなくなります。次に、アプリケーションをテストしてみて、CPU 利用度が高く、コンテキスト切り替えが頻繁に発生している場合は、maxWorkerThreads を減らすか、minFreeThreads を増やしてテストしてください。

方策がうまくいっていれば、以下のようになります。

- CPU 利用度が上がる。
- ASP.NET Applications¥Requests/Sec パフォーマンスカウンタによって表されるスループットが向上する。
- ASP.NET Applications/Requests in Application Queue パフォーマンスカウンタによって表されるアプリケーションキューが減る。

スレッド処理に関するガイドライン

ここでは、ASP.NET におけるスレッド処理の効率性を高めるのに役立つガイドラインを順に説明します。

競合を減らすための方策を用いてスレッド プールをチューニングする

利用可能な CPU があって要求をキューに入れる場合、**ASP.NET スレッド プール**を設定してください。アプリケーションが**共通言語ランタイム（CLR）スレッド プール**を使っている場合、スレッドプールを適切にチューニングすることが重要です。正しくチューニングしなければ、競合問題、パフォーマンス上の問題、場合によってはデッドロックを経験することになるかもしれません。アプリケーションは以下の条件が当てはまる場合、CLR スレッド プールを使っている可能性があります。

- アプリケーションが Web サービス呼び出しをする。
- アプリケーションが WebRequest または HttpWebRequest クラスを使って外向きの Web 要求をする。
- アプリケーションが、QueueUserWorkItem メソッドを呼び出して明示的に作業をスレッド プールに入れる。

バースト負荷に対して minIoThreads と minWorkerThreads を考慮する

アプリケーションが、長い休止期間をはさみながらバースト負荷を経験する場合、スレッドプールは、スレッドの最適レベルに達するのに必要な時間を得られないかもしれません。バースト負荷は、多数のユーザーが突然、同時にアプリケーションに接続する場合に発生します。**minIoThreads** 設定と **minWorkerThreads** 設定により、負荷条件に対し、それぞれ I/O スレッド数とワーカースレッド数の下限を設定できます。

スレッドを要求ごとに作成しない

スレッドの作成は、マネージリソースとアンマネージリソースの両方の初期化を必要とする、大きな負担を伴う処理です。ASP.NET アプリケーションや Web サービスなどのサーバーベースのアプリケーションでは、スレッドを要求ごとにマニュアルで作成しないようにするべきです。

CPU 依存でなく、呼び出しと並行して処理できる作業がある場合は、非同期呼び出しを検討してください。例として、ファイル読み出しやファイル書き込みを含むディスク I/O 依存の処理、他の Web メソッドの呼び出しを含むネットワーク I/O 依存処理などがあります。

.NET Framework の提供するインフラストラクチャを利用し、**Beginsynchronous** メソッドと **Endsynchronous** メソッド（synchronous は同期メソッド名を表す）によって非同期処理を実行できます。この非同期呼び出しパターンが適切でない場合は、CLR スレッドプールのスレッドを使う

ことを検討してください。以下のコードでは、メッセージをスレッドプールの別スレッドで動作させるために、キューに入れる方法を示しています。

```
WaitCallback methodTarget = new WaitCallback(myClass.UpdateCache);
bool isQueued = ThreadPool.QueueUserWorkItem(methodTarget);
```

スレッドをブロックしない

現在の要求スレッドをブロックする ASP.NET ページから処理を実行すると、他の ASP.NET 要求の処理に利用可能なスレッドプール内のスレッドが、1 つ減ることになります。スレッドは、ブロックしないようにしてください。

追加的な並行作業がなければ非同期呼び出しを避ける

Web アプリケーションからの非同期呼び出しは、呼び出し完了待機中に実行すべき追加的な並行作業がアプリケーションにあり、その呼び出しによって実行される作業が CPU 依存でない場合にのみ行ってください。内部的には、非同期呼び出しは、スレッドプールのワーカースレッドを使います。実質的に、追加スレッドを使っていることになります。

Web メソッドの呼び出しやファイル処理の実行などの非同期 I/O 呼び出しをするのと同時に、呼び出しをしたスレッドは開放され、別の非同期呼び出しや他の並行作業など、追加的な作業を実行できるようになります。その後、これらすべてのタスクの完了を待つことができます。CPU 依存でない複数の非同期呼び出しを実行し、それらを同時に処理することによりスループットを向上させることができます。

リソース管理

ページやコントロールからの不適切なリソース管理は、Web アプリケーションのパフォーマンスを低下させる主な原因の 1 つです。不適切なリソース管理は、CPU に大きな負荷をかけ、大量のメモリを消費しかねません。

ガイドラインを順に説明しますので、効率的なリソース管理に役立ててください。

リソースをプールする

ADO.NET には、完全自動の**データベース コネクション プーリング**が組み込まれており、特別なコーディングは必要ありません。データベースへの各アクセス要求に対し、同じ接続文字列を使うようにしてください。

プールしたリソースは、できる限り速やかに開放し、プールに戻してください。プールしたリソースをキャッシュすることや、プールしたリソースの保持中に長いブロック呼び出しをすることは、避

けてください。その間、他のクライアントがそのリソースを使えないためです。また、複数の要求をまたいでオブジェクトを保持してはなりません。

開いたリソースについて明示的に Dispose または Close を呼び出す

IDisposable インターフェイスを実装したオブジェクトを使う場合は、そのオブジェクトが提供していれば、**Dispose** メソッド、または **Close** メソッドを呼び出すようにしてください。Close メソッドまたは Dispose メソッドを呼び出さないと、オブジェクトはクライアントが使用しなくなった後もメモリに長い間残り続けることになります。これにより、クリーンアップは遅れ、メモリ使用量は増加します。明示的に閉じるべき共有リソースの例としては、データベースコネクションやファイルなどがあります。Try / finally ブロックの finally クローズは、オブジェクトの Close メソッドや Dispose メソッドを呼び出すのに適した場所です。

<Visual Basic>

```
Try
  conn.Open()
  ...
Finally
  If Not(conn Is Nothing) Then
    conn.Close()
  End If
End Try
```

<C#>

```
SqlConnection conn = new SqlConnection(connString);
using (conn)
{
  conn.Open();
  . . .
} // ここで接続オブジェクト conn の Dispose を自動呼び出し.
```

プールしたリソースをキャッシュまたはブロックしない

アプリケーションがプールしたリソースを使っている場合、**リソースを開放**してプールに戻してください。プールしたリソースのキャッシュやプールしたリソースからのブロック呼び出しにより、他のユーザーが利用可能なプールしたリソースは減ってしまいます。プールしたリソースには、データ

ベースコネクション、ネットワークコネクション、Enterprise Services のプールしたオブジェクトなどがあります。

アプリケーションのアロケーション パターンを知っておく

不適切な**メモリ アロケーション パターン**により、ガベージコレクタがジェネレーション 2 のオブジェクトの回収に作業のほとんどを費やすようになってしまう場合があります。ジェネレーション 2 のオブジェクトの回収は、アプリケーションパフォーマンスの低下や、CPU への負荷の増加につながります。

リソースは遅く確保して早く手放す

重要で限りのある**共有リソース**は、必要となる直前に開き、できる限り速やかに開放してください。これらの共有リソースの例としては、データベースコネクション、ネットワークコネクション、トランザクションなどが挙げられます。

要求ごとに偽装しない

Web サーバーにおいて、呼び出し元を特定し、必要なら承認するようにしてください。システムリソースやアプリケーションをまたぐリソースには、Web アプリケーション プロセスの ID、または固定サービスアカウントを使ってアクセスしてください。システムリソースとは、イベントログなどのリソースをいいます。アプリケーションをまたぐリソースとは、データベースなどのリソースをいいます。**要求ごとの偽装**を避けることにより、セキュリティオーバーヘッドを最小限に抑え、リソースプーリングを最大化することができます。

ASP.NET ページ

ASP.NET ページと、コードビハインドのページロジックの効率性は、Web アプリケーションの全体的なパフォーマンスを大きく左右します。

順にガイドラインを示します。これらは個々の .aspx ファイル、および .ascx ファイルの開発に関連しています。

ページ サイズを小さくする

大きなサイズのページを処理すると、CPU 負荷やネットワーク帯域消費は増加しクライアントに対する応答時間は長くなります。複数のタスクをカバーする大きなページのデザインや作成は、特に各要求に対して実行するタスクが通常は数個しかない場合は避けてください。可能な場合は、ページを論理的に分割してください。

以下のうち 1 つ、またはすべてに従いページサイズを小さくしてください。

◆ ページで静的スクリプトを含むスクリプトを使い、クライアントが以後の要求のためにこれらのスクリプトをキャッシュできるようにする。

```
<script language="javascript" src="scripts/myscript.js">
```

◆ クライアントに応答を送る前に、空白を作るタブやスペースなどを削除する。空白を削ると、ページ サイズは大幅に小さくなります。以下は、空白を含んでいるテーブルの例です。

```
// 空白あり
<table>
  <tr>
    <td>hello</td>
    <td>world</td>
  </tr>
</table>
```

以下は、空白のないコードの例です。

```
// 空白なし
<table><tr><td>hello</td><td>world</td></tr></table>
```

ノートパッドを使ってこれらのテーブルを別々のテキストファイルに保存し、各ファイルのサイズを確認してみてください。2 つ目のテーブルは、空白をなくしただけで数バイトを節約していることがわかります。1,000 行のテーブルなら、空白をなくすだけで応答時間を短縮できます。インターネットシナリオでは、空白をなくしても大きな効果は得られないかもしれません。しかし、速度の遅いクライアントを含むインターネットシナリオなら、空白をなくすことによって応答時間を大幅に短縮できる場合もあります。また、HTTP 圧縮を検討することもできます。ただし、HTTP 圧縮は CPU 利用度に影響を与えます。

常にこのような方法でページを設計できるとは限りません。従って、**Internet Server API (ISAPI) フィルタ** または **HttpModule** オブジェクトを使うのが、空白をなくすための最も効果的な方法です。ISAPI フィルタは、HttpModule よりも高速です。ただし、ISAPI フィルタは作成が難しい上に、CPU 利用度を増加させます。また、IIS 圧縮を検討可能な場合もあります。IIS 圧縮は、メタベースエントリを使って追加できます。

ページサイズの調整にはこれらの他に、ビューステートを必要でなければ無効にする、グラフィックの使用を制限し圧縮グラフィックの使用を検討する、などの方法があります。

バッファリングを有効にする

バッファリングはデフォルトで有効となっているため、ASP.NET はサーバーで作業をバッチ化し、クライアントと不要な通信をしないようにします。このアプローチの欠点は、遅いページのレンダリングは完了するまでクライアント側から見られないという点です。この状況での次善策として、

Response.Flush の使用があります。Response.Flush は、出力をクライアント側のポイントまで送ります。バッファ機能が無効な遅いネットワークにつながっているクライアントは、サーバーの応答時間に影響を与えます。これは、サーバーがクライアントからの肯定応答を待たなければならないためです。最初の送信時にヘッダーを送っているため、待機を後回しにすることはできません。

バッファリングが無効になっている場合は、以下の方法で有効にできます。

- ページでプログラムによって有効にする。

```
Response.BufferOutput = true;
```

- @Page ディレクティブを使い、ページレベルで有効にする。

```
<%@ Page Buffer="true" %>
```

- Web.config ファイルまたは Machine.config ファイルの <pages> 要素を使い、アプリケーションレベルまたはコンピュータ レベルで有効にする。

```
<pages buffer="true" ...>
```

ASP.NET プロセスモデルを使って ASP.NET アプリケーションを動作させる場合、バッファリングを有効にすることは、さらに重要になります。ASP.NET ワーカープロセスはまず、応答バッファの形で IIS に応答します。これらの応答バッファのサイズは、31 KB です。IIS は、応答バッファを受信すると、実際の応答をクライアントへ返送します。バッファリングが無効になっていると、ASP.NET は 31 KB のバッファ全体を使う代わりに、バッファへ数個の文字を送ることしかできなくなります。この場合、ASP.NET でも IIS でも、追加的な CPU 処理が発生することになります。また、IIS プロセスにおけるメモリ消費が大幅に増加してしまう場合もあります。

Page.IsPostBack を使って冗長的な処理をなくす

Page.IsPostBack プロパティを使い、ページが最初にロードされたときにのみページ初期化を実行し、クライアントポストバックに応答して実行しないようにしてください。以下のコードは、Page.IsPostBack プロパティの使い方を示したものです。

```
if (Page.IsPostBack == false) {
  // 初期化ロジック
} else {
  // クライアント ポストバック ロジック
}
```

ページのコンテンツを分割してキャッシングを効率化すると共にレンダリングを減らす

ページのコンテンツを分割し、キャッシュしやすくしてください。ページのコンテンツを分割すると、それを取得、表示、キャッシュする上での選択肢が増えます。ユーザーコントロールを使い、操作アイテム、メニュー、広告、著作権表示、ページヘッダー、ページフッターなどの静的コンテンツを分割できます。また、動的コンテンツやユーザーの独自コンテンツについても、キャッシュ時の柔軟性を最大限に高められるよう分割するべきです。

ページがバッチ コンパイルされるようにする

プロセスにロードされるアセンブリの数が増えるにつれ、仮想アドレス空間が断片化していくことがあります。この場合は、メモリ不足状態に陥りやすくなります。プロセスに多数のアセンブリをロードしないように、ASP.NET は、同じディレクトリ内のすべてのページを 1 つのアセンブリにコンパイルしようとします。このコンパイルは、そのディレクトリ内のページに対する最初の要求があったときに発生します。以下のアプローチにより、バッチコンパイルされないアセンブリの数が少なくなるようにしてください。

- 複数の言語を同じディレクトリ内に混在させない。C# や Visual Basic .NET などの複数の言語が同じディレクトリ内のページで使われていると ASP.NET はアセンブリを言語ごとに分けてコンパイルします。
- コンテンツの更新によって追加アセンブリがロードされないようにする。
- 次節で説明するように、ページ レベルおよび Web.config ファイルで、debug 属性を false に設定する。

デバッグを false に設定する

debug が true に設定されていると、以下の状況が発生してしまいます。

- ページがバッチコンパイルされない。
- ページがタイムアウトしない。Web サービスエラーなどの問題が発生したとき、Web サーバーが要求をキューに入れ始め、応答しなくなってしまう場合があります。
- Temporary ASP.NET Files フォルダで追加的なファイルが生成される。
- 生成コードに System.Diagnostics.DebuggableAttribute 属性が追加される。これにより、生成コード情報に対する追加的なトラックが発生し、やはり確実な最適化を妨げられてしまいます。

パフォーマンステストやアプリケーションの生産開始の前に、Web.config ファイルやページレベルで debug が false に設定されていることを確認してください。ページレベルでは、debug はデフォルトで false に設定されています。開発期間中にこの属性を設定しなければならない場合は、以下のコードで示すように、Web.config ファイルレベルで設定しておくべきです。

```
<compilation debug="false" ... />
```

以下は、ページレベルで debug を false に設定する方法です。

```
<%@ Page debug="false" ... %>
```

負荷の大きいループを最適化する

いかなるアプリケーションでも、**負荷の大きいループ**はパフォーマンス上の問題の原因となり得ます。ループ内のコードに関連するオーバーヘッドを減らすには、以下の推奨事項に従うべきです。

- フィールドやプロパティへの反復的なアクセスを避ける。
- ループ内のコードを最適化する。
- 頻繁に呼び出すコードをループ内にコピーする。
- 再帰処理をループ処理に置き換える。
- パフォーマンス重視のコードパスでは ForEach の代わりに For を使う。

Response.Redirect の代わりに Server.Transfer の使用を検討する

Response.Redirect は、新しい URL によるサーバーへの新しい要求の送信をクライアントに求めるメタタグをクライアントへ送ります。**Server.Transfer** を使うと、サーバー側呼び出しにより、この迂回的なアプローチを回避できます。Server.Transfer 使用時は、ブラウザーで表示される URL は変わらず、負荷テストツールは、ページサイズを誤って報告するかもしれません。これは、同じ URL について異なるページがレンダリングされるためです。

Server.Transfer メソッド、Response.Redirect メソッド、Response.End メソッドはすべて、内部で Response.End を呼び出すため、ThreadAbortException 例外を発生させます。Response.End を呼び出すと、この例外が発生します。Response.End の内部呼び出しを避けるため、オーバーロードしたメソッドを使い、第 2 引数として false を渡すことを検討してください。

クライアント側検証を用いる

データの事前検証により、ユーザー要求の処理に必要なラウンドトリップ数を減らすことができます。ASP.NET では、検証コントロールを使い、ユーザー入力のクライアント側検証を実装できます。

サーバー コントロール

サーバー コントロールを利用し、よく使う機能のカプセル化と再利用を実現できます。簡潔なプログラミング抽象化をもたらすサーバーコントロールは、ASP.NET アプリケーション構築のための推奨方法とされています。サーバーコントロールを適切に使えば、出力キャッシュとコードの保守性を向上させることができます。パフォーマンスの最適化のために見直すべき主な範囲は、ビューステートとコントロール構成です。

順にガイドラインを示しますので、サーバーコントロール開発時は、これらのガイドラインに従ってください。

サーバー コントロールにおけるビュー ステートの使用状態を確認する

ビュー ステートは、サーバーでシリアル化され、シリアル化解除されます。CPU サイクルを節約するには、アプリケーションで使用するビューステートの数を減らしてください。不必要なビューステートは無効にします。以下のうち、1 つでも当てはまる場合は、ビューステートを無効にしてください。

- ユーザー入力のない読み出し専用ページを表示している。
- サーバーにポストバックしないページを表示している。
- ポストバックデータを確認することなく、ポストバックごとにサーバー コントロールを再構築している。

適切な場合はユーザー コントロールを使う

HTTP プロトコルはステートレスです。しかし、サーバーコントロールはビューステートによってページ要求間の状態情報を管理する、豊かなプログラミングモデルを提供します。サーバーコントロールは、自身と子コントロールを確立するために、固定量の処理を要求します。この結果、サーバーコントロールは、HTML コントロール、場合によっては静的テキストと比べても高くついてしまいます。サーバーコントロールが高くついてしまうシナリオの例を以下に掲げます。

- **狭帯域での大きなペイロード** —— ページ内のコントロール数が多いほど、ネットワークペイロードは大きくなります。従って、複数コントロールにより、クライアントへ送られた応答に対する TTLB (最終バイトまでの時間) と TTFB (先頭バイトまでの時間) は減ります。クライアントが低速のダイアルアップ接続を使用しているシナリオなど、クライアント―サーバー間の帯域幅が限られている場合、大きなビューステートペイロードを伴うページは、パフォーマンスに大きな悪影響を及ぼし得ます。
- **ビュー ステート オーバーロード** —— ビューステートは、サーバーでシリアル化、およびシリアル化解除されます。CPU にかかる負荷は、ビューステートのサイズに比例して大きくなります。ビューステートを使うサーバーコントロールに、プログラミング時にシリアル化可能なオブジェクトをビューステートプロパティに加えるのは簡単なことです。しかし、これによりオーバーヘッドは増加します。また、計算済みデータの格納や一般データの複数コピーの格納などの手法も、不要なオーバーヘッドの増加を招きます。
- **複合コントロールまたは多数コントロール** —— DataGrid などの複合コントロールを含むページは、ビューステートのフットプリント <メモリ占有スペース> を増加させる場合があります。多数のサーバーコントロールを含むページも、ビューステートによるメモリ使用量を増加させる場合があります。可能な場合は、この節で後述する別のアプローチを選択してください。

さまざまな機能を伴う対話を必要としていないなら、サーバーコントロールを提供したいユーザーインターフェイスをインライン化したものに置き換えてください。以下の状況では、サーバーコントロールの使用を避けることを検討すべきです。

- ポストバックをまたいで状態情報を保持する必要がない。
- コントロールに現れるデータが静的データである（たとえばラベルは静的データ）。
- サーバー側のコントロールへプログラム的にアクセスする必要がない。
- コントロールが読み出し専用データを表示している。
- ポストバック処理中にコントロールを必要としない。

サーバーコントロールに代わるものには、**シンプルなレンダリング**、**HTML 要素**、**Response.Write 呼び出しのインライン化**、**インライン アングル ブラケット (<% %>)** などがあります。パフォーマンスとの間でトレードオフを図ることが大切です。オーバーヘッドが許容範囲内でアプリケーションがパフォーマンス目標値を満たしているなら、過剰な最適化は避けてください。

コントロールを階層化しすぎないようにする

コントロールをあまりにも階層化しすぎると、サーバーコントロールとその子コントロールを作成するコストは大きくなってしまいます。過剰な階層化は、インラインコントロールを使う別の設計アプローチやサーバーコントロールの階層化の程度を抑えることによって回避可能な余計な処理を招きます。これは、コンテナ内に追加的な子コントロールを生成させる、Repeater、DataList、DataGrid などのリストコントロールを使っている場合、特に重要です。

たとえば、以下の Repeater コントロールについて考慮してください。

```
<asp:repeater id="r" runat="server">
<itemtemplate>
<asp:label runat="server"><%# Container.DataItem %><br/></asp:label>
</itemtemplate>
</asp:repeater>
```

データソースに 50 のアイテムがあると想定すると、Repeater コントロールを含むページのトレースが有効な場合、ページは実質的に 200 を超えるコントロールを含むことになります。

表. *Repeater* コントロール階層の一部

コントロール ID	型
Repeater	System.Web.UI.WebControls.Repeater
repeater:_ctl0	System.Web.UI.WebControls.RepeaterItem
repeater_ctl0:_ctl1	System.Web.UI.LiteralControl
repeater_ctl0:_ctl0	System.Web.UI.WebControls.Label

repeater_ctl0:_ctl2	System.Web.UI.LiteralControl
repeater:_ctl49	System.Web.UI.WebControls.RepeaterItem
repeater_ctl49:_ctl1	System.Web.UI.LiteralControl
repeater_ctl49:_ctl0	System.Web.UI.WebControls.Label
repeater_ctl49:_ctl2	System.Web.UI.LiteralControl

ASP.NET のリストコントロールは、さまざまなシナリオに対応できるように作られており、特定のシナリオには最適化されていません。パフォーマンス重視のシナリオでは、以下のうち、いずれかのアプローチを選択することができます。

あまり複雑でないデータを表示したいなら、Response.Write を呼び出して自らレンダリングすることも可能です。たとえば、以下のコードでは、この節で前述したのと同じ出力が得られます。

```
for(int i = 0; i < datasource.Count; i++)
{
  Response.Write(datasource[i] + "<br>");
}
```

より複雑なデータを表示したい場合は、レンダリングのためにカスタムコントロールを作成することができます。たとえば、以下のカスタムコントロールでは、この節で前述したのと同じ出力が得られます。

```
public class MyControl : Control
{
  private IList _dataSource;
  public IList DataSource
  {
    get { return _dataSource; }
    set { _dataSource=value; }
  }
  protected override void Render(HtmlTextWriter writer)
  {
    for(int i = 0; i < _dataSource.Count; i++)
    {
      writer.WriteLine(_dataSource[i] + "<br>");
    }
  }
}
```

データ連結

データ連結もまた、非効率的に用いるとパフォーマンス上の問題を招くことが多いアプローチの 1 つです。データ連結を用いる場合の推奨事項を挙げますので、これらを考慮するようにしてください。

Page.DataBind の使用を避ける

Page.DataBind の呼び出しにより、ページレベルのメソッドが呼ばれます。続いてこのメソッドがデータ連結をサポートするページの各コントロールの **DataBind** メソッドを呼び出します。ページレベルの DataBind を呼び出す代わりに、特定コントロールの DataBind を呼び出してください。この両方のアプローチのコード例を以下で示しています。

以下のコードでは、ページレベルの DataBind が呼び出されます。この DataBind が、今度は各コントロールの DataBind を呼び出します。

```
DataBind();
```

以下のコードでは、特定コントロールの DataBind が呼び出されます。

```
yourServerControl.DataBind();
```

DataBinder.Eval の呼び出しを最小限に抑える

DataBinder.Eval メソッドは、リフレクションを使用し、受け取った引数を調べて結果を返します。100 行 10 列のテーブルなら、各列で DataBinder.Eval を使うと DataBinder.Eval を 1,000 回呼び出すことになります。このシナリオでは、DataBinder.Eval の使用頻度は 1,000 倍となります。データ連結中の DataBinder.Eval の使用を制限するとページパフォーマンスは大きく向上します。DataBinder.Eval を使った **Repeater** コントロール内の **ItemTemplate** 要素について考慮してください。

```
<ItemTemplate>
  <tr>
    <td><%# DataBinder.Eval(Container.DataItem,"field1") %></td>
    <td><%# DataBinder.Eval(Container.DataItem,"field2") %></td>
  </tr>
</ItemTemplate>
```

このシナリオで DataBinder.Eval を使わない手法もあります。その例を以下に掲げます。

- **明示的にキャストする** —— 明示的にキャストすると、リフレクションのコストを避けられ、パフォーマンスは向上します。Container.DataItem を DataRowView にキャストしてください。

```
<ItemTemplate>
  <tr>
    <td><%# ((DataRowView)Container.DataItem)["field1"] %></td>
```

```
    <td><%# ((DataRowView)Container.DataItem)["field2"] %></td>
  </tr>
</ItemTemplate>
```

コントロールの連結に DataReader を使い、データの取得に特別なメソッドを使えば、明示的なキャストによるパフォーマンスの向上度は、さらに高まります。Container.DataItem を、DbDataRecord にキャストしてください。

```
<ItemTemplate>
  <tr>
    <td><%# ((DbDataRecord)Container.DataItem).GetString(0) %></td>
    <td><%# ((DbDataRecord)Container.DataItem).GetInt(1) %></td>
  </tr>
</ItemTemplate>
```

- **ItemDataBound イベントを使う** —— データ連結しようとするレコードが多くのフィールドを含んでいる場合は、ItemDataBound を使った方が効率的かもしれません。このイベントを使うと、型変換を 1 度実行するだけで済みます。以下は、DataSet オブジェクトを使ったコード例です。

```
protected void Repeater_ItemDataBound(Object sender,
                                      RepeaterItemEventArgs e)
{
  DataRowView drv = (DataRowView)e.Item.DataItem;
  Response.Write(string.Format("<td>{0}</td>",drv["field1"]));
  Response.Write(string.Format("<td>{0}</td>",drv["field2"]));
  Response.Write(string.Format("<td>{0}</td>",drv["field3"]));
  Response.Write(string.Format("<td>{0}</td>",drv["field4"]));
}
```

キャッシングについて

キャッシングにより、冗長的な作業を避けることができます。キャッシングを適切に使えば不要なデータベース検索などの負荷の大きい処理を回避できます。また、待ち時間の短縮にもつながります。

ASP.NET キャッシュは、ASP.NET アプリケーションに提供されるシンプルで拡張性のあるメモリ内キャッシングサービスです。これは、時間ベースの期限切れ機能を提供すると共に外部ファイル、ディレクトリなどのキャッシュキーへの依存性をトラックします。また、キャッシュ内のアイテムが期限切れとなったときにコールバック関数を呼び出すメカニズムも備えています。キャッシュは、LRU (最小使用頻度) アルゴリズム、設定済みメモリ制限、キャッシュ内のアイテムの CacheItemPriority

列挙値に基づき、アイテムを自動的に削除します。キャッシュデータは、アプリケーションまたはワーカープロセスのリサイクル時にも消滅します。

ASP.NET が提供するキャッシング技法は、キャッシュ API、出力キャッシュ、部分ページ / フラグメント キャッシュの 3 つです。

キャッシュ API

キャッシュ API を使用し、複数ユーザーによって共有 / アクセスされるアプリケーションにまたがるデータをプログラム的にキャッシュすべきです。キャッシュ API は、ユーザーに提示する前に、何らかの操作を施す必要のあるデータの置き場所としても適しています。これらのデータの例としては、文字列、配列、コレクション、データセットなどが挙げられます。

API キャッシュの使用が望ましい代表的なシナリオには、ニュース見出しや製品カタログなどがあります。

また、以下の状況ではキャッシュ API の使用を避けるべきです。

- キャッシュしようとするデータが、ユーザーの独自データである。この場合は、代わりにセッション状態の使用を検討してください。
- データがリアルタイムで更新される。
- アプリケーションが既に完成しており、コードベースの更新が望ましくない。この場合は、出力キャッシュの使用を検討してください。

キャッシュ API により、外部条件に依存するキャッシュへのアイテムの挿入が可能になります。キャッシュアイテムは、外部条件の変更時にキャッシュから自動的に除去されます。この機能は、**CacheDependency** クラスのインスタンスを受け入れる **Cache.Insert** メソッドの第 3 引数を使うことによって利用できます。CacheDependency クラスは、さまざまな依存シナリオをサポートする、8 種類のコンストラクタを備えています。これらのコンストラクタには、ファイルベース、時間ベース、優先順位ベースの依存性が、既存依存性に基づく依存性と共に含まれています。

キャッシュデータを提供する前に、コードを動作させることも可能です。たとえば、キャッシュデータを特定の顧客に提供したくても、他の顧客にはリアルタイム更新したデータを提供したくない場合もあるでしょう。このタイプのロジックも **HttpCachePolicy.AddValidationCallback** メソッドを使えば実行できます。

出力キャッシュ

出力キャッシュにより、ページ全体のコンテンツを一定期間内にキャッシュできるようになります。出力キャッシュは、クエリ文字列、ヘッダー、userAgent 文字列に基づく、さまざまな種類のページのキャッシングを可能にします。また、プロキシ、サーバー、クライアントなど、コンテンツをキャッシュすべき場所を判断可能になります。キャッシュ API を使った場合と同様に、データ取得に要

する時間は短縮されます。また、コンテンツのレンダリングに要する時間も短縮されます。要求ごとにビューを更新する必要がなく、ユーザーの独自データが含まれていないなら、動的に生成されるページでは出力キャッシュを利用可能にすべきです。

訪問頻度の高いページやレポートは出力キャッシュが望ましい代表的なシナリオです。

また、以下の状況では出力キャッシュの使用を避けてください。

- ページのデータにプログラム的にアクセスする必要がある。（代わりにキャッシュ API の使用を検討してください。）
- ページの種類が多くなりすぎた。
- 静的データ、動的データ、ユーザー独自データが、ページに混在している。（フラグメントキャッシングの使用を検討してください。）
- ビューごとにリフレッシュしなければならないコンテンツがページに含まれている。

部分ページ / フラグメント キャッシュ

部分ページ / フラグメント キャッシュは、出力キャッシュの派生アプローチです。これに含まれている追加属性により、ユーザー コントロール (.ascx ファイル) のプロパティに基づくバリエーションのキャッシュが可能になります。

フラグメントキャッシュは、ユーザーコントロールの使用により、@OutputCache ディレクティブと共に実装されます。ページのコンテンツ全体をキャッシュしたい場合にフラグメントキャッシュを使用するのは実用的ではありません。ページに静的データ、動的データ、ユーザー独自データが混在している場合は、ユーザーコントロールを作成し、ページを論理パーティションで区切ってください。これらのユーザーコントロールは、メインページから独立してキャッシュ可能で、キャッシングによって処理時間の短縮とパフォーマンスの向上を実現できます。

操作メニューやヘッダーとフッターなどがフラグメントキャッシュが望ましい代表的なシナリオになります。

また、以下の状況では、フラグメントキャッシュの使用を避けるべきです。

- ページの種類が多くなりすぎた。
- キャッシュしたユーザーコントロールに、ビューごとにリフレッシュしなければならないコンテンツが含まれている。

アプリケーションが複数のページ上で同じコントロールを使っている場合、ユーザーコントロール @OutputCache ディレクティブの Shared 属性を true に設定し、ページが同じインスタンスを共有するようにしてください。これにより、かなりのメモリスペースを節約できます。

キャッシングに関するガイドライン

キャッシング ストラテジを設計する際のガイドラインを挙げますので、これらに従うようにしてください。

ページで静的データと動的データを分ける

部分ページキャッシュにより、ユーザーコントロールを使ってページの一部分だけをキャッシュすることができます。ユーザーコントロールを使いページを分割してください。たとえば、静的情報、動的情報、ユーザー独自情報の混在した、以下のようなシンプルなコードを見てください。

<main.aspx>

```
<html>
<body>
  <table>
    <tr>
      <td colspan="3">Application Header ? Welcome John Smith</td>
    </tr>
    <tr>
      <td>Menu</td>
      <td>Dynamic Content</td>
      <td>Advertisments</td>
    </tr>
    <tr>
      <td colspan="3">Application Footer</td>
    </tr>
  </table>
</html>
```

このページを、以下のコードによって分割し、キャッシュすることができます。

<main.aspx>

```
<%@ Register TagPrefix="app" TagName="header" src="header.ascx" %>
<%@ Register TagPrefix="app" TagName="menu" src="menu.ascx" %>
<%@ Register TagPrefix="app" TagName="advertisements"
    src="advertisements.ascx" %>
<%@ Register TagPrefix="app" TagName="footer" src="footer.ascx" %>
<html>
<body>
```

```
  <table>
    <tr>
      <td colspan="3"><app:header runat="server" /></td>
    </tr>
    <tr>
      <td><app:menu runat="server" /></td>
      <td>Dynamic Content</td>
      <td><app:advertisements runat="server" /></td>
    </tr>
    <tr>
      <td colspan="3"><app:footer runat="server" /></td>
    </tr>
  </table>
</html>
```

<header.ascx>

```
<%@Control %>
Application Header ? Welcome <% GetName() %>
```

<menu.ascx>

```
<%@Control %>
<%@ OutputCache Duration="30" VaryByParam="none" %>
Menu
```

<advertisements.ascx>

```
<%@Control %>
<%@ OutputCache Duration="30" VaryByParam="none" %>
Advertisements
```

<footer.ascx>

```
<%@Control %>
<%@ OutputCache Duration="60" VaryByParam="none" %>
Footer
```

上記のサンプルで示したようにコードを区分けすることにより、ページの選択部分をキャッシュし、処理時間とレンダリング時間を短縮することができます。

メモリ制限を設定する

メモリ制限の設定とチューニングは、キャッシュの最適化に不可欠です。メモリ消費量が設定したメモリ制限の 20 パーセント未満になると、ASP.NET キャッシュは、まず LRU アルゴリズムと、

アイテムに割り当てられた CacheItemPriority 列挙値に基づき、キャッシュを小さくし始めます。メモリ制限の設定が高すぎると予期しないプロセスのリサイクルが発生しかねません。また、アプリケーションでメモリ不足例外が発生する場合もあります。逆にメモリ制限の設定が低すぎるとガベージコレクションの実行に要する時間が増加し、全体的なパフォーマンスが低下することになります。

実証的な検証により、プライベートバイトが 800 メガバイト (MB) を越えるとメモリ不足例外を受ける可能性が高くなることがわかっています。ただし、プライベートバイトをいつ増やすか、または減らすかについて判断する際に注意すべきことは、800 MB というのは .NET Framework 1.0 のみに関連しているということです。.NET Framework 1.1 を備えている場合や、/3 GB スイッチを使っている場合は 1,800 MB まで増やせます。

ASP.NET プロセスモデルを使用している場合、Machine.config ファイルのメモリ制限を以下のように設定します。

```
<processModel memoryLimit="50">
```

この値は、ワーカープロセスに消費することを認める物理メモリのパーセンテージを管理します。この値を超えると、プロセスはリサイクルされます。上記のコードサンプルでは、サーバーに 2 ギガバイト (GB) の RAM があるとすれば、利用可能な物理 RAM の総量が、RAM 全体の 50 パーセント未満、つまり 1GB 未満に落ちると、プロセスはリサイクルされます。ワーカープロセスメモリは、process パフォーマンスカウンタオブジェクトと private bytes カウンタを使って監視します。

適切なデータをキャッシュする

適切なデータをキャッシュするのは重要なことです。不適切なデータをキャッシュすると、パフォーマンスを向上させるどころか逆に低下させることになりかねません。

アプリケーション全体で使われるデータや、複数ユーザーに使われるデータをキャッシュしてください。また、静的データや、作成や取得に大きな負担を伴うデータをキャッシュしてください。取得に大きな負担を伴い、定期的に変更されるデータを適切に管理すればパフォーマンスとスケーラビリティを向上させることができます。データ量の大きいサイトでは、データを数秒間キャッシュしただけでもパフォーマンスは大きく向上する可能性があります。データ連結に最適化したシリアル化を用いたデータセットやカスタムクラスもまた、キャッシングの有力な候補となります。使用頻度が更新頻度より高いデータもキャッシングの有力な候補です。

データベースコネクションなどの希少な共有リソースをキャッシュしないでください。キャッシュすれば、競合が発生するためです。また、**DataReader** オブジェクトは、ベースとなる接続をオープンに保つためキャッシュに格納しないようにしてください。これらのリソースはプールすべきです。要求をまたぐユーザーごとのデータについては、キャッシュせずにセッション状態を使用してください。データが要求の独自データで同じ要求についてデータベースに繰り返しアクセスする代わりに、

その要求のライフタイムの間、そのデータを格納し、渡す必要がある場合は、データを **HttpContext**.Current.Cache オブジェクトに格納することを検討してください。

キャッシュを適切にリフレッシュする

データを 10 分ごとに更新するなら、キャッシュも 10 分ごとに更新する必要がある、というわけではありません。サービスレベル契約を満たすのに必要なデータの更新頻度を見極めてください。頻繁に変わるデータのためにキャッシュを再取得しないようにしてください。頻繁に変わるデータは、キャッシングの対象として好ましくありません。

適切なデータ形式でキャッシュする

レンダリングされた出力をキャッシュしたい場合は、出力キャッシュまたはフラグメントキャッシュを用いるべきです。レンダリングされた出力がアプリケーションの他の場所でも使われる場合は、その格納にはキャッシュ API を使ってください。データを操作する必要があるなら、キャッシュ API を使い、そのデータを格納してください。たとえば、データをコンボ ボックスに結び付けたい場合、取得したデータを、キャッシュする前に **ArrayList** オブジェクトに変換してください。

比較的静的なページのキャッシュには出力キャッシュを用いる

ページが、複数のユーザー要求の間で比較的静的な場合は**ページ出力キャッシュ**を使い、ページ全体を一定期間キャッシュすることを検討してください。キャッシュ期間は、ページのデータの性質に基づいて指定します。動的ページであっても、常に要求ごとに再構築しなければならないわけではありません。たとえば、作成に大きな負担を伴う Web ベースのレポートを一定期間キャッシュすることが可能な場合もあります。動的ページを 1、2 分間キャッシュしただけでもデータ量の多いページではパフォーマンスは大幅に向上し得ます。

キャッシュアイテムを、期限切れになる前に消去する必要がある場合は、**HttpResponse.RemoveOutputCacheItem** メソッドを使用できます。以下のコードで示すように、このメソッドは消去したいページへの絶対パスを受けます。

```
HttpResponse.RemoveOutputCacheItem("/Test/Test.aspx");
```

ここで注意すべきなのは、キャッシュは Web ファーム全体で共有されないため、パスはサーバーによって異なるという点です。また、ユーザーコントロールからは使用できません。

適切なキャッシュ場所を選択する

@OutputCache ディレクティブにより、**Location** 属性を使ってページのキャッシュ場所を指定できるようになります。Location 属性で選択可能な値は以下のとおりです。

- **Any** —— これがデフォルト値です。出力キャッシュは、要求を発行したブラウザークライアント、プロキシサーバー、要求処理にかかわっているその他のサーバー、または要求を処理するサーバーに配置可能です。
- **Client** —— 出力キャッシュは、要求を発行したブラウザークライアントに置かれます。
- **DownStream** —— 出力キャッシュは、元のサーバーを除く、あらゆる HTTP 1.1 キャッシュ可能デバイスに格納できます。これらのデバイスには、プロキシサーバーや要求元のクライアントも含まれます。
- **None** —— 出力キャッシュは、その要求ページについて無効となります。
- **Server** —— 出力キャッシュは、要求が処理された Web サーバーに置かれます。
- **ServerAndClient** —— 出力キャッシュは、元のサーバー、または要求元クライアントにのみ、格納可能です。プロキシサーバーは応答をキャッシュできません。

クライアントかプロキシサーバーが応答をキャッシュすることが確実でない場合は、Location 属性を Any、Server、ServerAndClient のいずれかに設定するべきです。その他に設定すると、下位に利用可能なキャッシュがなければ、出力キャッシュのメリットを得られなくなります。

選択的キャッシュには VaryBy 属性を使う

VaryBy 属性により、同じページの異なるバージョンをキャッシュすることが可能になります。ASP.NET は、4 つの VaryBy 属性を提供しています。

- **VaryByParam** —— クエリ文字列値に基づき、ページの異なるバージョンが格納されます。
- **VaryByHeader** —— 指定ヘッダー値に基づき、ページの異なるバージョンが格納されます。
- **VaryByCustom** —— ブラウザーの種類とメジャーバージョンに基づき、ページの異なるバージョンが格納されます。さらに、カスタム文字列を定義することにより出力キャッシュを拡張できます。
- **VaryByControl** —— ユーザーコントロールのプロパティ値に基づき、ページの異なるバージョンが格納されます。これは、ユーザーコントロールのみに適用可能です。

VaryBy 属性は、キャッシュするデータを特定します。以下のコードで、VaryBy 属性の使い方を示します。

```
<%@ OutputCache Duration="30" VaryByParam="a" %>
```

上記コードの設定により、以下のページについて同じバージョンをキャッシュすることになります。

- http://localhost/cache.aspx?a=1
- http://localhost/cache.aspx?a=1&b=1
- http://localhost/cache.aspx?a=1&b=2

VaryByParam 属性に b を加えると、1 つのバージョンでなく、異なる 3 つのバージョンが格納されることになります。キャッシュされるページのバリエーション数を知っておくのは重要なこと

です。2 つの変数 (a と b) があり、a に 5 種類の組み合わせ、b に 10 種類の組み合わせがあるなら、キャッシュされるページのバリエーションの総数は以下の式を使って計算できます。

```
(MAX a × MAX b) + (MAX a + MAX b) = 65 種類
```

VaryBy 属性を使う場合、バリエーションが多いほど Web サーバーでのメモリ消費量は増加するためバリエーション数を制限するようにしてください。

Windows Server 2003 ではカーネル キャッシュを使う

Windows Server 2003 および IIS 6.0 は、**カーネル キャッシュ**を提供しています。ASP.NET ページは、IIS 6.0 の カーネル キャッシュにより、自動的にメリットを得ることができます。カーネルキャッシュはパフォーマンスの大幅な向上をもたらします。これは、キャッシュ応答に対する要求が、ユーザーモードへの切り替えを経ずに提供されるためです。

Windows Server 2008 以降では出力キャッシュを使う

Windows Server 2008 以降、および IIS 7.0 以降では、**出力キャッシュ**を構成して、Web サーバー、サイト、またはアプリケーションのパフォーマンスを向上できます。ユーザーが Web ページを要求すると、IIS は要求を処理してクライアントブラウザにページを返します。出力キャッシュを有効にした場合は、処理された Web ページのコピーが Web サーバーのメモリに格納されます。その後、同じリソースが要求されると、その Web ページのコピーがクライアントブラウザに返されます。これにより、要求されるたびにページを再処理する必要がなくなります。

状態管理

Web アプリケーションでの状態管理には、固有の問題がつきまといます。この問題は、Web ファームに展開された Web アプリケーションで顕著に見られます。状態情報の格納場所と格納方法に関する選択は、アプリケーションのパフォーマンスとスケーラビリティに大きな影響を及ぼします。状態情報にはさまざまな種類があります。

- **アプリケーション状態** —— アプリケーション状態は、すべてのクライアントがアプリケーション全体で使う状態情報の格納に使われます。アプリケーション状態の使用は、サーバーアフィニティを発生させるため、スケーラビリティに影響します。Web シナリオにおいて、アプリケーション状態を変えたときに、サーバーをまたいで変更を複写するメカニズムはありません。そのため、同じユーザーからのその後の要求が別のサーバーへ行くと、その変更は適用されません。アプリケーション状態のデータは、以下のサンプルで示すように、キー / 値ペアを使って格納してください。

  ```
  Application["YourGlobalState"] = somevalue;
  ```

- **セッション状態** —— セッション状態は、サーバーにおけるユーザーごとの状態情報を格納するために使われます。この状態情報は、ファーム内の Web サーバーをまたいで、セッションクッキーまたは ASP.NET セッション状態スケールを使ってトラックされます。
- **ビュー ステート** —— ビューステートは、ページごとの状態情報を格納するために使われます。この状態情報は、HTTP POST 要求 / 応答のたびに流されます。
- **その他** —— 状態管理にはその他、クライアントクッキー、クエリ文字列、Hidden フィールドなどの手法があります。

以下は、状態管理全般についての広範な問題に対応するためのガイドラインを挙げます。

可能な限りクライアントでシンプルな状態情報を格納する

個人データなどと違って機密性の低い軽量なユーザー独自の状態情報を格納する場合は、クッキー、クエリ文字列、隠れたコントロールなどを使ってください。これらは、セキュリティ性の高い情報の格納には使用しないでください。情報の読み出しや操作が容易にできてしまうためです。

- **クライアント クッキー** —— クライアントクッキーはサーバーで作成され、クライアントブラウザーへ送られて格納されます。これらは、ドメインに固有のもので、そのセキュリティは完全ではありません。ブラウザーからのその後のすべての要求には、これらのクッキーが含まれるようになります。サーバーのコードは、これらを検査し、変更することができます。クッキーに入れることのできるデータの最大量は、4 KB です。
- **クエリ文字列** —— クエリ文字列は、URL に付加されるデータです。データはクリアテキストで、文字列の全長には制限があります。このデータは、ユーザーによって容易に操作可能です。そのため、認証や検証を経ることなく、クエリ引数に基づいて機密性の高いデータの取得や表示をしてはなりません。 ただし、匿名の Web サイトでは、それほど問題になりません。
- **Hidden コントロール** —— ページの Hidden コントロールは、要求と応答で行き来する状態情報を格納します。

シリアル化コストを考慮する

状態情報をシリアル化する必要がある場合は、**シリアル化コスト**を考慮してください。たとえば、リモートの状態ストアに状態情報を格納するために、シリアル化が必要となるかもしれません。この場合、絶対に必要なものだけを格納し、複雑なオブジェクトよりもシンプルな型を優先し、シリアル化による影響を減らすようにしてください。

アプリケーション状態

アプリケーション状態は、アプリケーション全体で使われる静的情報の格納のために使われます。ASP.NET がアプリケーション状態をサポートしているのは、主に Active Server Pages (ASP) テクノロジとの互換性を確保し、ASP.NET への移植を容易にするためにです。

以下にガイドラインを挙げます。ガイドラインをアプリケーションの最適化に役立ててください。

アプリケーション状態の格納には Applicaton オブジェクトでなく静的プロパティを使う

データは、**Application** オブジェクトでなく、アプリケーションクラスの静的メンバに格納すべきです。Application ディレクトリのアイテムよりも静的変数の方が速くアクセスできるため、これによってパフォーマンスは向上します。以下は、単純化したコード例です。

```
<%
private static string[] _states[];
private static object _lock = new object();
public static string[] States
{
  get {return _states;}
}
public static void PopulateStates()
{
  // スレッド セーフとすること
  if (_states == null)
  {
    lock (_lock)
    {
      // 状態情報を取得...
    }
  }
}
public void Application_OnStart(object sender, EventArgs e)
{
  PopulateStates();
}
%>
```

静的な読み出し専用データの共有にはアプリケーション状態を使う

アプリケーション状態は、アプリケーション全体で利用されサーバーに固有のものです。読み書き可能なデータも格納できますが、サーバーアフィニティを避けるためには、呼び出し専用データのみ

を格納すべきです。**Cache** オブジェクトの使用を検討してください。Cache オブジェクトは読み出し専用データよりも理想的です。

STA COM オブジェクトをアプリケーション状態に格納しない

STA COM オブジェクトをアプリケーション状態に格納すると、アプリケーションのボトルネックとなります。アプリケーションは、コンポーネントへのアクセスをシングルスレッドで実行するためです。アプリケーション状態には STA COM を格納しないようにしてください。

セッション状態

ASP.NET でセッション状態が必要な場合、3 つのセッション状態モードから選択可能です。以下で示すように選択するモードによってパフォーマンスとスケーラビリティは変わってきます。

- **InProc** —— インプロセスストアにより、セッション状態へのアクセスは最も速くなります。このモードでは、状態情報は ASP.NET プロセスのマネージメモリ内で保持されるため、シリアル化コストやマーシャリングコストは発生しません。ASP.NET プロセスは、Windows 2000 Server の Aspnet_wp.exe ファイルと、Windows Server 2003 の W3wp.exe ファイルです。プロセスがリサイクルされると状態データは失われます。ただし、プロセスのリサイクルは、アプリケーションに影響する場合、IIS 6 では無効にすることができます。インプロセスストアは、複数のワーカープロセスと共に利用できないため、アプリケーションのスケーラビリティを制限します。たとえば、インプロセスストアは、Web ファーム展開や Web ガーデン展開を妨げます。また、大型セッションや同時セッションが多くなるとメモリ不足が発生しかねません。
- **StateServer** —— ローカル Web サーバーや、Web ファームのすべての Web サーバーからアクセス可能なリモート サーバーに Win32 のセッション状態サービス (StateServer) をインストールできます。このアプローチによりスケーラビリティは向上しますが、パフォーマンスはインプロセスプロバイダの場合に比べて落ちます。これは、状態ストアとの間で状態情報を行き来させる際にシリアル化とマーシャリングが必要になるためです。
- **SQL Server** —— Microsoft SQL Server は、スケーラビリティに富み、簡単に利用可能なソリューションを提供します。SQL Server は、大量のセッション状態に最適なソリューションです。シリアル化コストとマーシャリングコストは、セッション状態サービスの場合と同じです。しかし、全体的なパフォーマンスは、わずかに落ちてしまいます。SQL サーバーは、フェールオーバーのためのクラスタ化を提供します。ただし、セッション状態のデフォルト設定では、クラスタ化はサポートされていません。有効にするには、コンフィギュレーションを変更する必要があります。また、セッションデータは、一時的でないテーブルに格納しなければなりません。

状態ストアの選択

インプロセス セッション状態ストアは、パフォーマンスとスケーラビリティの大幅な向上をもたらします。しかし、データ量の多い Web アプリケーションのほとんどは、Web ファームで動作します。スケールアウトを可能にするためには、セッション状態サービスか、SQL Server 状態ストアの

いずれかを選択しなければなりません。このとき、付随するネットワーク待ち時間とシリアル化の影響を考慮すると共に、それらを計測し、アプリケーションがパフォーマンス目標値を満たすことを確認する必要があります。以下の情報を、状態ストアの選択に役立ててください。

- **単体の Web サーバー** —— 単体の Web サーバーを備えているとき、セッション状態パフォーマンスを最適化したいとき、そして同時セッションの数がそれほど多くなく、限られているときは、インプロセスセッション状態ストアを使ってください。セッションの再構築が高くつくときや、ASP.NET 再起動時に持続性が必要な場合は、セッション状態サービスを使ってください。また、信頼性が最も重要な場合は、SQL Server 状態ストアを使ってください。
- **Web ファーム** —— ローカル Web サーバーでは、インプロセスセッション状態ストアを使うことやセッション状態サービスを動作させることは避けるようにしてください。これらは、サーバーアフィニティを発生させます。インターネットプロトコル (IP) アフィニティを使い、同じクライアントからのその後の要求を、同じサーバーが処理するようにすることができます。しかし、インターネットサービスプロバイダ (ISP) がリバースプロキシを使っていると、このアプローチによって問題が発生する場合があります。Web ファームシナリオでは、リモートセッション状態サービスか SQL Server を使ってください。
- **StateServer vs. SQLServer** —— SQL Server データベースを備えていない場合は、リモートセッション状態サービスを使ってください。エンタープライズアプリケーションやデータ量の多い Web アプリケーションでは、SQL Server を使ってください。リモートのセッション状態サービスと Web サーバーがファイアウォールによって隔離されている場合はポートを開く必要があります。デフォルトポートは 42424 ポートです。ポートは、以下のレジストリ キーによって変更できます。

```
HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥aspnet_state
¥Parameters
```

セッション状態パフォーマンス最適化のガイドライン

セッション状態パフォーマンスを最適化するためのガイドラインを以下に挙げます。

基本型を優先してシリアル化コストを減らす

StateServer または SQLServer の**アウトプロセス セッション状態ストア**を使うと、**シリアル化オーバーヘッド**が発生します。オブジェクトグラフがシンプルなほどシリアル化は速くなるはずです。シリアル化コストを最小限に抑えるために、Int、Byte、Decimal、String、 DateTime、TimeSpan、Guid、IntPtr、UintPrt などの基本型を使ってください。ASP.NET は、基本型のシリアル化には、内部の最適化したシリアル化メソッドを使います。複雑な型は、比較的遅い **BinaryFormatter** オブジェクトによってシリアル化されます。複雑な型については、**Serializable** 属性を使うか、**ISerializable** インターフェイスを実装することができます。このインターフェイスを使うと、より正確な制御が可能になり、シリアル化速度が向上する場合もあります。

シリアル化の対象を、最小限に抑えてください。不要ならシリアル化を無効にし、シリアル化可能クラスのうち、除外したいフィールドには **NonSerialized** 属性を付けてください。ISerializable インターフェイスを使い、シリアル化プロセスを制御してください。

使わなければセッション状態を無効にする

セッション状態を使わないのなら、無効にし、ASP.NET が実行する追加的なセッション処理をなくしてください。シンプルな状態情報をクライアントで格納し、それを要求ごとにサーバーへ渡す場合、セッション状態は必要となりません。また、以下で示すように、サーバーの全アプリケーション、特定のアプリケーション、または個々のページについて、セッション状態を無効にすることができます。

- サーバーの全アプリケーションについてセッション状態を無効にするには、以下で示す Machine.config ファイルの要素を使ってください。

```
<sessionState mode="Off" />
```

  セッション状態オーバーヘッドを完全になくすために、<httpModules> のセッション状態モジュールを取り除くこともできます。

- 特定のアプリケーションについてセッション状態を無効にするには、以下で示すアプリケーションの Web.config ファイルの要素を使ってください。

```
<sessionState mode="Off" />
```

- 特定の Web ページについてセッション状態を無効にするには以下のページ設定を使ってください。

```
<%@ Page EnableSessionState="false" . . .%>
```

セッション状態に STA COM オブジェクトを格納しないようにする

セッション状態に **STA COM** オブジェクトを格納すると、スレッドアフィニティが発生します。スレッドアフィニティは、パフォーマンスとスケーラビリティに大きな悪影響をもたらします。セッション状態で STA COM オブジェクトを使う場合は、必ず **@ Page** ディレクティブの **AspCompat** 属性をセットしてください。

可能な場合は ReadOnly 属性を使う

内部でセッション状態を使うページ要求は、セッションデータの管理のために **ReaderWriterLock** オブジェクトを使用します。この場合、ロックがかかっていなければ、同時に複数読み出しをすることが可能になります。書き込み側がセッション状態更新のためにロックを取得すると読み出し要求はすべてブロックされます。通常、各要求についてデータベースへ 2 つの呼び出しが送られます。1 つ目の呼び出しは、データベースへの接続を確保し、セッションをロック状態としページを実行します。2 つ目の呼び出しは、あらゆる変更について書き込み、セッションへのロッ

クを解除します。セッションデータを読み出すだけのページについては、以下のサンプルで示すように、**EnableSessionState** を ReadOnly にセットすることを検討してください。

```
<%@ Page EnableSessionState="ReadOnly" . . .%>
```

EnableSessionState を ReadOnly にセットすると、フレームを使っている場合に、特にメリットを得られます。フレーム使用時は、ReaderWriterLock が使われるため、デフォルト設定ではページの実行はシリアル化されます。EnableSessionState を ReadOnly にセットすると、ブロックを避け、データベースへの呼び出しを減らすことができます。前述したように、構成ファイルでセッションを無効にし、ページ単位で ReadOnly 属性を設定するのも 1 つの方法です。

ビュー ステート

ビューステートは主に、サーバーコントロールが、データを自身へポストバックするページのみで、状態情報を保持するために使います。情報はクライアントへ渡され、**_VIEWSTATE** と呼ばれる特定の隠れた変数で読み出されます。ASP.NET は、あらゆるシリアル化可能な型のビューステートへの格納を容易にしています。しかし、この機能は簡単に誤用され、パフォーマンス低下の原因となります。ビューステートは、それを必要としないページにとっては不要なオーバーヘッドとなります。ビューステートが大きくなるにつれてパフォーマンスは以下のようにして影響を受けます。

- ビューステートのシリアル化とシリアル化解除が、CPU サイクルの増加につながる。
- 大きくなったページのダウンロードにかかる時間が長くなる。
- 非常に大きいビューステートは、ガベージコレクションの効率性にも影響を及ぼし得る。

大量のビューステートの送信は、アプリケーションのパフォーマンスに大きな影響を与えかねません。Web クライアントが遅いダイアルアップ接続を使っている場合、パフォーマンスの低下度はさらに著しくなります。ビューステートを扱う場合は、さまざまな帯域条件でテストすることを検討してください。

また、以下に推奨事項を挙げますので、これに従ってアプリケーションによるビューステートの使用を最適化してください。

必要なければビューステートを無効にする

ASP.NET では、ビューステートはデフォルトで有効となっています。必要ない場合は、これを無効にしてください。たとえば、ページが出力のみの場合や、データを要求ごとに明示的に再ロードする場合は、ビューステートは必要ありません。その他、以下の状況では、ビューステートは必要ありません。

- **ページがポストバックしない** —— ページが情報を自身にポストバックしない場合、ページが出力のみに使われる場合、およびページが応答に依存しない場合。

- **サーバー コントロール イベントを処理しない** —— サーバーコントロールがイベントを処理しない場合、およびサーバーコントロールが動的、もしくはデータに依存するプロパティ値を持っていないか、プロパティ値が要求ごとにセットされる場合。
- **ページがリフレッシュされるたびにコントロールを再取得する** —— 古いデータを無視する場合、およびページがリフレッシュされるたびにサーバーコントロールを再取得する場合。

各レベルでビューステートを無効にするには、さまざまな方法があります。

- Web サーバーのすべてのアプリケーションについてビューステートを無効にするには、Machine.config ファイルの <pages> 要素を以下のように設定してください。

```
<pages enableViewState="false" />
```

これにより、@ Page ディレクティブの EnableViewState 属性を使い、必要とするページについてのみ、ビューステートを有効にすることができます。

- 1 つのページについてビューステートを無効にするには、@ Page ディレクティブを以下のように使ってください。

```
<%@ Page EnableViewState="false" %>
```

- ページの 1 つのコントロールについてビューステートを無効にするには、そのコントロールの EnableViewState プロパティを以下のコード例で示すようにして false に設定してください。

```
// 命令的
yourControl.EnableViewState = false;
// 宣言的
<asp:datagrid EnableViewState="false" runat="server" />
```

個別のコントロールのビューステートの有効化

ASP.NET 4 では、Web サーバー コントロールに **ViewStateMode** プロパティが追加されました。このプロパティを使用して、既定ではビューステートを無効にしておき、ページ内のビューステートが必要なコントロールでのみ有効にすることができます。

ViewStateMode プロパティは、Enabled、Disabled、および Inherit という 3 つの値から成る列挙値を使用します。Enabled では、対象のコントロールと、Inherit が設定されているか何も設定されていないその子コントロールのビューステートが有効になります。Disabled ではビューステートが無効になり、Inherit では親コントロールの ViewStateMode 設定を使用します。

```
<asp:PlaceHolder ID="PlaceHolder1" runat="server"
  ViewStateMode="Disabled">
  Disabled:
  <asp:Label ID="label1" runat="server" Text="[DeclaredValue]" />
```

```
  <br />
  <asp:PlaceHolder ID="PlaceHolder2" runat="server"
    ViewStateMode="Enabled">
    Enabled:
    <asp:Label ID="label2" runat="server" Text="[DeclaredValue]" />
  </asp:PlaceHolder>
</asp:PlaceHolder>
```

ビューステートに格納するオブジェクト数を最小化する

ビューステートに入れるオブジェクト数が増加するにつれ、ビューステートディクショナリのサイズは大きくなり、オブジェクトインスタンスのシリアル化とシリアル化解除に必要な処理時間も長くなります。ビューステートにオブジェクトを格納する場合は以下のガイドラインに従ってください。

- ビューステートは、文字列、整数、ブール値などのシリアル化可能な基本型、およびこれらの基本型を含む、配列、ArrayLists、Hashtables などのオブジェクトに最適化されています。上記以外の型をビューステートに格納する場合、ASP.NET は、内部で関連する型変換を実行しようとします。型変換ができない場合、これよりも大きな負担を伴う、バイナリシリアライザが使われることになります。
- ビューステートのサイズは、オブジェクトのサイズに直接的に比例します。大きなオブジェクトは格納しないようにしてください。

ビューステートのサイズを調べる

ページのトレースを可能にすることにより、各コントロールのビューステートのサイズを監視できるようになります。各コントロールのビューステートのサイズはトレース出力のコントロールツリーセクションの最左列に表示されます。この情報を参考にし、ビューステートの量を減らせるコントロールや、ビューステートを無効にできるコントロールがあるかを調べてください。

HTTP モジュール

HTTP モジュールは、ASP.NET パイプライン通過中の HTTP 要求 / 応答メッセージに対する事前処理および事後処理を可能にするものです。HTTP モジュールは通常、認証、承認、ログ、マシンレベルのエラー処理のために使われます。HTTP モジュールは、あらゆる要求に対して動作します。そのため、HTTP モジュールによるすべての処理は、登録場所に応じてアプリケーションかコンピュータのいずれかについて全体にわたる影響を及ぼします。

HTTP モジュールを開発する場合に考慮することを以下に挙げます。

パイプライン コードでは所要時間の長い呼び出しやブロック呼び出しを避ける

以下の理由により、HTTP モジュールに所要時間の長いコードを置かないようにしてください。

- ASP.NET ページは、同期処理される。
- HTTP モジュールは通常、同期イベントを使う。

所要時間の長い呼び出しやブロック呼び出しは、ASP.NET で動作可能な並行要求数を減らします。

非同期イベントを検討する

各同期イベントについて、非同期バージョンが存在します。非同期イベントは非同期作業の間、要求を論理的にブロックしてしまいますが、ASP.NET スレッドはブロックしません。

文字列管理

出力時に文字列の連結が必要となることは、よくあります。文字列連結は、メモリの一時アロケーションとその後のコレクションを必要とする、大きな負担を伴う処理です。そのため、文字列連結の実行は最小限に抑えるべきです。データのレンダリングのためにページで文字列を連結するための一般的な方法として以下の 3 つが挙げられます。

- **+= 演算子を使う** —— アペンド数が既知の場合、+= 演算子を使います。
- **StringBuilder** —— アペンド数が未知の場合は、StringBuilder オブジェクトを使います。StringBuffer は再利用可能なバッファとして扱ってください。
- **Response.Write <% %>** —— Response.Write メソッドを使います。出力を最も速くブラウザーへ返すことのできる方法の 1 つです。

上記のそれぞれについてパフォーマンスを計測してからアプローチを選ぶのが最も効果的です。アプリケーションが一時バッファに大きく依存している場合は、文字配列またはバイト配列のための再利用可能なバッファプールを実装することを検討してください。

文字列の管理に役立つガイドラインを以下に挙げます。

出力の書式設定には Response.Write を使う

可能な場合、ページレイアウトの書式設定のための文字列連結にループを使わないでください。代わりに、**Response.Write** を使うことを検討してください。このアプローチでは、ASP.NET 応答バッファに出力が書き込まれます。データセットまたは XML ドキュメントについてループを行っている場合、Response.Write を使うのが効率的なアプローチです。これは、クライアントに書き込む前に += 演算子を使って文字列を連結するよりも効率的です。Response.Write は、再利用可能なバ

ッファに内部で文字列を加えます。そのため、メモリのアロケーションやクリーンアップによるパフォーマンスオーバーヘッドを避けることができます。

StringBuilder を一時バッファとして使う

多くの場合で、Response.Write は適用できません。たとえば、ログファイルへの書き込みや XML ドキュメントの作成に文字列の作成が必要となる場合もあるでしょう。このような状況では、**StringBuilder** オブジェクトをデータ保持のための一時バッファとして使ってください。StringBuilder オブジェクトについて、初期キャパシティ設定をいろいろ変えてパフォーマンスを計測してみてください。

カスタム コントロール作成時には HttpTextWriter を使う

カスタムコントロール作成時は、**Render**、**RenderChildren**、**RenderControl** の各メソッドが、**HtmlTextWriter** オブジェクトへのアクセスを提供します。HtmlTextWriter は、**Response.Write** と同じ再利用可能なバッファに書き込みます。Response.Write の場合と同様に、HtmlTextWriter によりメモリのアロケーションやクリーンアップによるパフォーマンスオーバーヘッドを避けることができます。

例外管理

例外は、大きな負担を伴います。例外の原因を知っておき、例外を避け、発生しても効率的に処理するコードを作成することにより、アプリケーションのパフォーマンスとスケーラビリティを大幅に向上させることができます。

例外処理の設計と実装の際に使うガイドラインを以下に挙げますのでパフォーマンスを最適化してください。

Global.asax エラー ハンドラを実装する

例外管理の第一歩は、Global.asax ファイル、またはコードビハインドファイルでのグローバルエラーハンドラの実装です。グローバルエラーハンドラの実装により、アプリケーション内の処理されない例外はすべてトラップされます。ハンドラ内では、最低限、データベース、Windows イベントログ、ログファイルなどのデータストアに以下の情報をログするべきです。

- エラーが発生したページ
- 呼び出しスタック情報
- 例外名とメッセージ

エラーロジックの処理には、**Global.asax** ファイル、または以下のサンプルで示すようにコードビハインドで **Application_Error** イベントを使ってください。

```
public void Application_Error(object s, EventArgs ev)
{
  StringBuilder message = new StringBuilder();
  if (Server != null) {
    Exception e;
    for (e = Server.GetLastError(); e != null; e = e.InnerException)
    {
      message.AppendFormat("{0}: {1}{2}",
            e.GetType().FullName,
            e.Message,
            e.StackTrace);
    }
    // 例外情報と内部例外情報をログ
  }
}
```

アプリケーションの例外を監視する

アプリケーションで発生する例外の数を減らすためには、例外を効率的に監視する必要があります。そのためには以下を実行すべきです。

- 例外処理コードを実装しているなら、例外ログを定期的に見直す。
- .NET CLR Exceptions パフォーマンスモニタオブジェクトの # of Exceps Thrown / sec カウンタを監視する。この値は、1 秒当たりの平均要求数の 5 パーセント未満であるべきです。

破棄可能なリソースで try / finally を使う

例外発生時にリソースがクリーンアップされるようにするには、**try / finally** ブロックを使ってください。リソースは、finally クローズで閉じます。try / finally ブロックにより、例外発生時でもリソースが破棄されるようにしてください。以下はそのコード例です。

```
try
{
  conn.Open();
  ...
}
finally
{
  if (null != conn)
```

```
    conn.close;
}
```

例外を避けるコードを作成する

例外を避けるために用いることができる一般的なアプローチを以下に掲げます。

- **null 値をチェックする** —— オブジェクトを null に設定できる場合は、例外をスローする前に null となっていないかを確認してください。ビューステート、セッション状態、アプリケーション状態、またはキャッシュオブジェクトやクエリ文字列からアイテムを取得し、フィールド変数を作ると、オブジェクトが null となることがよくあります。たとえば、セッション状態情報へのアクセスに以下のコードを使わないでください。

```
try {
  loginid = Session["loginid"].ToString();
}
catch(Exception ex) {
  Response.Redirect("login.aspx", false);
}
```

セッション状態情報へのアクセスには、代わりに以下のコードを使ってください。

```
if (Session["loginid"] != null)
  loginid = Session["loginid"].ToString();
else
  Response.Redirect("login.aspx", false);
```

- **ロジックの制御に例外を使わない** —— 例外は、あくまでも例外です。開かないデータベースコネクションは例外です。一方、パスワードを打ち間違えたユーザーは処理が必要な状況にあるだけです。たとえば、ユーザーのログインに使う以下の関数プロトタイプを見てください。

```
public void Login(string UserName, string Password) {}
```

ログインの呼び出しには以下のコードが使われます。

```
try
{
  Login(userName,password);
}
catch (InvalidUserNameException ex)
{...}
catch (InvalidPasswordException ex)
{...}
```

とり得る値の列挙型を作り、その列挙型を返すために **Login** メソッドを以下のように変えた方が効果的です。

```
public enum LoginResult
{
  Success, InvalidUserName, InvalidPassword, AccountLockedOut
}
public LoginResult Login(string UserName, string Password) {}
```

Login の呼び出しには以下のコードが使われます。

```
LoginResult result = Login(userName, password);
switch (result)
{
  case Success:
    . . .
  case InvalidUserName:
    . . .
  case InvalidPassword:
}
```

- **Response.End の内部呼び出しをしないようにする** —— Server.Transfer、Response.Redirect、Response.End の各メソッドは例外を発生させます。各メソッドは、内部で Response.End を呼び出します。Response.End が呼び出されると、ThreadAbortException 例外が発生します。Response.Redirect を使用する場合は、オーバーロードしたメソッドを使い、第 2 引数として false を渡すことにより Response.End の内部呼び出しが発生しないようにしてください。
- **処理できない例外をキャッチしない** —— コードが例外を処理できない場合は、try / finally ブロックを用い、例外が発生するかどうかにかかわらずリソースを閉じるようにしてください。修復を試みることができない場合、例外はキャッチしないでください。例外は、その例外状況を処理できる適切なハンドラへ伝達するようにしてください。

タイムアウト時間を短めに設定する

ページ タイムアウト時間をあまりにも長めに設定すると、アプリケーションの一部の動作速度が遅い場合に問題が生じ得ます。具体的には以下の問題が発生する場合があります。

- プラウザが応答しなくなる。
- 受信要求がキューに入り始める。
- 要求キューが制限に達すると、IIS が要求を拒否する。
- ASP.NET が応答しなくなる。

デフォルトのページタイムアウト時間は、90 秒です。各シナリオに合わせて、この値を変えることができます。

ASP.NET フロントエンドアプリケーションがリモート Web サービスを呼び出す場合について考えてみてください。リモート Web サービスは、続いてメインフレームデータベースを呼び出します。このメインフレームへの呼び出しが何らかの理由でブロックされ始めると、フロントエンド ASP.NET ページはバックエンド呼び出しがタイムアウトするかページタイムアウト制限を越えるまで待機し続けます。この結果、現在の要求はタイムアウトし、ASP.NET は受信要求をキューに入れ始めます。これらの要求もまたタイムアウトすることになるかもしれません。これらの要求のタイムアウト時間を 90 秒より短くした方が効率は上がります。また、それにより、ユーザーの使い勝手も向上します。

ほとんどのインターネット / イントラネットシナリオでは、タイムアウト制限は 30 秒が妥当です。ホームページなどのトラフィック量の多いページではタイムアウト制限をさらに短くすべき場合もあるでしょう。レポートページなどアプリケーションが生成するまでに長い時間を要するページの場合、アプリケーションの場合はタイムアウト制限を長めに設定してください。

COM 相互運用

ASP.NET から **COM オブジェクト**を呼び出すと、パフォーマンスが影響を受ける場合があります。これは、スレッド処理の問題、データ型のマーシャリング、マネージコードとアンマネージコードの境界をまたいだ切り替え、などに対処しなければならなくなるためです。ASP.NET は、マルチスレッドアパートメント (MTA) スレッドで動作するため、STA COM コンポーネントとの共同作業では、特に問題が生じ得ます。

以下に COM 相互運用パフォーマンスのガイドラインを挙げます。

STA COM オブジェクトの呼び出しには AspCompat を使う

Visual Basic 6.0 コンポーネントなどの STA オブジェクトを ASP.NET ページから呼び出す場合は、ページレベルの AspCompat 属性を使ってください。ページのイベントが、デフォルトの MTA スレッドでなく、STA スレッドプールのスレッドを使って動作すべきことを表示するには、AspCompat 属性を以下のように使います。

```
<%@ Page AspCompat="true" language="c#" %>
```

STA オブジェクト呼び出しは、STA スレッドを必要とします。AspCompat 属性を使わない場合、すべての STA オブジェクト呼び出しは、ホスト STA スレッドでシリアル化され重大なボトルネックとなり得ます。

セッション状態やアプリケーション状態に COM ブジェクトを格納しない

セッション状態やアプリケーション状態などの状態コンテナに、COM オブジェクトを格納しないようにしてください。COM オブジェクトはシリアル化できません。シングル サーバー展開で呼び出すことは可能ですが、アフィニティとシリアル化の問題によりアプリケーションは Web ファームへ移されると作業できなくなります。

セッション状態に STA オブジェクトを格納しない

セッション状態に **STA オブジェクト**を格納することは技術的には可能ですが、スレッドアフィニティの問題を引き起こすため避けてください。格納すると、STA オブジェクトへの要求は、オブジェクトを作ったのと同じスレッドで動作しなければならず、ユーザー数が増えると、たちまちこれがボトルネックとなります。

ページ コンストラクタで STA オブジェクトを作成しない

ページコンストラクタで STA オブジェクトを作成しないでください。ホスト STA へのスレッド切り替えと、すべての呼び出しのシリアル化が発生することになるためです。**ASPCOMPAT** 属性により、**onload** や **button_click** などのページイベントに STA スレッドプールのスレッドを使うようにすることは可能ですが、コンストラクタなどのページの他の部分は、MTA スレッドによって動作します。

標準的な ASP Server.CreateObject を事前バインディングで補う

遅延バインディングは、COM クラスの場合でもメソッドを名前で実行する場合でも対象コードを見つけるための特別な指示が必要とします。**Server.CreateObject**、**Activator.CreateInstance**、**MethodInfo.Invoke** などのメソッドにより、コードは遅延バインディングを実行できます。ASP コードを移植する場合、new キーワードを使って**事前バインディング**を行ってください。

データ アクセス

ASP.NET アプリケーションのほとんどすべては、何らかの形で**データアクセス**を使っています。アプリケーション要求の大半は、データベースのデータを対象とします。このため、データアクセスは通常、パフォーマンスの向上を図る上で最も注意すべきポイントです。

以下にデータアクセス向上のためのガイドラインを挙げます。

大きな結果セットにはページングを用いる

大きなクエリ結果セットを**ページング**することにより、アプリケーションのパフォーマンスは大幅に向上します。大きな結果セットがある場合、ページングを実装して以下を実現してください。

◆ データベースでのバックエンドの作業を減らす。

◆ クライアントへ送られるデータのサイズを小さくする。

◆ クライアントの作業を制限する。

さまざまなページングソリューションが利用可能です。各ソリューションは、特定のシナリオに固有の問題を解決します。以下では、これらのソリューションについて簡単に説明します。

比較的すばやく簡単に利用できるソリューションは、**DataGrid** オブジェクトの提供する自動ページングです。ただし、このソリューションはインクリメントする一意の列数を備えるテーブルでのみ、利用できます。大きなテーブルには適していません。ページングをカスタマイズするには、**AllowPaging** プロパティと **AllowCustomPaging** プロパティを true にセットし、**PageSize** プロパティと **VirtualItemCount** プロパティをセットします。これにより、StartIndex (最終ブラウズ行) プロパティと **NextIndex** (**StartIndex** + **PageSize**) プロパティが計算されます。StartIndex 値と NextIndex 値は、ID 列が要求ページを取得し、表示する範囲として使用されます。このソリューションではデータをキャッシュできません。ネットワークの関連レコードのみを取得します。

インクリメントする一意の列数を備えていないテーブルについては、さまざまなソリューションが利用可能です。クラスタ化したインデックスを備え、サーバー側で特別なコーディングを必要としないテーブルについては、subquery を使い、スタートからのスキップ行数をトラックしてください。その結果のレコードについて、TOP キーワードを <pagesize> 要素と共に使い、行の次のページを取得してください。ネットワークから、関連ページレコードのみが取得されます。その他のソリューションでは、Table データ型かグローバル一時テーブルを追加的な IDENTITY 列と共に使い、クエリ結果を格納します。この IDENTITY 列は、取得して表示する行の範囲を制限するために使われます。この場合、サーバー側でのコーディングが必要となります。

高速かつ効率的なデータ結合のために DataReader を使う

データをキャッシュする必要がない場合、読み出し専用データ表示している場合、およびデータをできる限り速くコントロールに読み込む必要がある場合は、**DataReader** オブジェクトを使ってください。DataReader は、読み出し専用データを前方のみを参照して取得する場合に最適です。データを **DataSet** オブジェクトにロードしてから DataSet をコントロールに結合すると、データは 2 度移動することになります。また、このメソッドの場合、DataSet のコンストラクト時に DataReader に比べて大きなコストが発生します。さらに、DataReader を使うと、型指定の特別メソッドによるデータの取得が可能になりパフォーマンスは向上します。

ユーザーがデータを過剰に要求しないようにする

ユーザーに消費可能な量を超えるデータの要求および取得を認めると、アプリケーションリソースに不要な負担がかかることになります。この結果、CPU 利用度とメモリ消費量は増加し、応答時間は

長くなります。この状況は、接続速度の遅いクライアントに特に当てはまります。使い勝手の面で考えても、数千行のデータを 1 ユニットで見ることを望むユーザーは、ほとんどいません。

以下のうちのいずれかにより、ユーザーが取得可能なデータ量を制限してください。

- ページングを実装する。
- マスター / 詳細フォームをデザインする。各データについての情報をすべてユーザーに与える代わりに、ユーザーが自分にとって関心のあるデータだと認識するのに十分な量の情報のみを表示する。そして、そのデータを選択し詳細を取得することをユーザーに許可する。
- ユーザーがデータをフィルタできるようにする。

データのキャッシングを検討する

かなり静的で取得に大きな負担を伴うアプリケーション全体で使われるデータについては、**ASP.NET キャッシュ**に格納することを検討してください。

セキュリティに関する考慮事項

多くの場合で、セキュリティとパフォーマンスは、設計トレードオフを最も考慮すべき組み合わせです。これは、**セキュリティメカニズム**を追加するとパフォーマンスに悪影響が及ぶことが多いためです。しかし、不要、無効、または害のあるトラフィックをフィルタにかけることや、 Web サーバーに到達することを認められる要求の数を抑えることにより、サーバー負荷を軽減できます。不要なトラフィックを早めにブロックするほど多くの処理オーバーヘッドを回避できます。

以下に推奨事項を挙げていきますので、これに従うようにしてください。

不要な Web サーバー トラフィックを抑える

Web サーバーへのトラフィックを抑え、不要な処理を避けてください。たとえば、ファイアウォールで無効な要求をブロックし、Web サーバーにかかる負荷を制限してください。これに加え、以下のことを行ってください。

- サポートしていない拡張子を、IIS の 404.dll ファイルにマップする。
- UrlScan フィルタを使い、許可する命令と URL 要求を制御する。制御の対象となり得る命令には、Get、Post、SOAP などがあります。
- IIS ログを見直す。許可していないトラフィックでログが一杯になっている場合は、ファイアウォールでのそのトラフィックをブロックするか、リバース プロキシを使ってトラフィックをフィルタにかけることを検討してください。

匿名アクセスでは認証を無効にする

認証アクセスが必要なページと、匿名アクセスをサポートするページを分けてください。不要な認証オーバーヘッドを避けるために、匿名ページを含むディレクトリの Web.config ファイルで認証モードを None にセットしてください。以下のコードで、Web.config ファイルでの認証モードの設定方法を示します。

```
<authentication mode="None" />
```

ユーザー入力をクライアントで検証する

サーバーへのトラフィックの不要な増加を回避するために、クライアント側検証を用いることを検討してください。ただし、クライアント側検証は容易にパスできるため、これのみに頼らないでください。セキュリティ上の理由により、各クライアント側検証に対し同等のサーバー側検証を実装すべきです。

要求ごとの偽装を避ける

要求ごとの偽装により、オリジナル呼び出し元の ID を使ってデータベースにアクセスすると、アプリケーションのスケーラビリティは大幅に制限されます。要求ごとの偽装により、データベースコネクションプーリングは、効果的に使用できなくなってしまいます。信頼済みサブシステムモデルの方が望ましく、拡張性にも富んでいます。このモデルでは、固定サービスアカウントを使ってデータベースにアクセスします。また、オリジナル呼び出し元の ID を求められた場合は、やはり固定サービス アカウントにより、アプリケーションレベルでの ID を渡します。たとえば、ストアドプロシージャ引数によってオリジナル呼び出し元の ID を渡す場合もあるでしょう。

機密性の高いデータのキャッシングを避ける

機密性の高いデータは、キャッシュする代わりに必要時に取得してください。アプリケーションパフォーマンスの計測時に、要求ごとのデータ取得に非常に大きなコストがかかっていることが判明した場合は、データの暗号化、キャッシュ、取得、暗号化解除に要するコストを計測してください。要求ごとのデータ取得に伴うコストが暗号化と暗号化解除のコストを上回っている場合は、暗号データをキャッシュすることを検討してください。

保護コンテンツと非保護コンテンツを分ける

Web サイトのフォルダ構造を設計する際には、どこからでもアクセス可能なエリアと、認証アクセスとセキュアソケットレイヤ (SSL) が必要となる制限エリアを明確に分けてください。アプリケーションの仮想ルートフォルダの下で別のサブフォルダを用い、フォームログオンページ、チェックアウトページなどの HTTPS によって保護すべき機密性の高い情報をユーザーから受ける制限ページを確保します。これにより、サイト全体にパフォーマンスオーバーヘッドを及ぼすことなく、特定のページについてのみ HTTPS を使用できるようになります。

SSL は要求するページにのみ使う

SSL の使用には、大きな負担が伴います。SSL は、要求するページにのみ使用するようにしてください。これらのページにはクレジットカード番号やパスワードを受け付けるページなど、機密性の高いデータを取得または保管するページなどがあります。以下の場合にのみ SSL を使ってください。

- ページデータを暗号化したい。
- データが希望のサーバーに確実に送られるようにしたい。

SSL を使わなければならないページについては以下のガイドラインに従ってください。

- ページサイズをできる限り小さくする。
- ファイルサイズの大きいグラフィックの使用を避ける。グラフィックを使う場合は、ファイルサイズが小さく解像度が低いものを選んでください。または、非保護サイトのグラフィックを使ってください。ただし、この場合、Web ブラウザーはダイアログボックスを表示し非保護サイトのコンテンツを表示することについてユーザーに確認をとります。

ナビゲーションには絶対パスを使う

リダイレクトを使った HTTP－HTTPS 間のナビゲーションでは、対象ページでなく現在ページのプロトコルが使われます。HTTPS を使ったサイトから非保護サイトへ、相対パス (../publicpage.aspx) によってダイレクトすると、これらのパブリックなページは HTTPS プロトコルを使って提供されることになります。これにより、不要なオーバーヘッドが発生します。これを避けるには、リダイレクトの際に相対パスでなく、http://yourserver/publicpage.aspx のような**絶対パス**を使ってください。これは、HTTP を使用しているページから HTTPS を使うページへ移る場合についても同じです。以下のコードでは、HTTP を使用しているページから HTTPS を使うページへのリダイレクトの作成方法を示しています。

```
string serverName =
  HttpUtility.UrlEncode(Request.ServerVariables["SERVER_NAME"]);
string vdirName = Request.ApplicationPath;
Response.Redirect("https://" + serverName + vdirName +
  "/Restricted/Login.aspx", false);
```

SSL 処理のオフロードのために SSL ハードウェアの使用を検討する

SSL 処理について、ハードウェアソリューションを検討してください。ハードウェアアクセラレータを使ってロードバランサで SSL セッションを終了させると、一般的にパフォーマンスは向上します。このアプローチは、ユーザーの多いサイトでは特に有効です。

SSL セッション期限切れが起きないように SSL タイムアウトをチューニングする

SSL ハードウェアを使っていない場合、**ServerCacheTimer** プロパティをチューニングし、ブラウザークライアントとの SSL ハンドシェイク を再び交わさなくて済むようにしてください。SSL 使用時にリソース使用度が最も高くなるのは初期ハンドシェイク中です。このとき、公開鍵と秘密鍵を使った非対称の暗号化が実施されます。セキュリティ保護されたセッション鍵が生成され交付されると、アプリケーションデータの暗号化には、より高速な対称暗号化が用いられます。

SSL 接続を監視し、タイムアウト時間を長くすべきと判断すれば、ServerCacheTime レジストリのエントリ値を増加させてください。

展開上の考慮事項

物理的展開は、アプリケーションのパフォーマンス特性およびスケーラビリティ特性を決める上で重要な役割を果たします。リモート中間層を導入する特別な理由がなければ、Web アプリケーションのプレゼンテーション層、ビジネス層、データ アクセス層は Web サーバー上に展開すべきです。唯一のリモートホップは、データベースへのホップのみとすべきです。ここでは、展開上の主な考慮事項を挙げていきます。

不要なプロセス ホップを避ける

プロセス ホップは、マシンホップほど高くつきませんが、可能な限り避けるべきです。プロセスホップは、プロセス間通信 (IPC) とマーシャリングを必要とするため、追加的なオーバーヘッドを発生させます。たとえば、Enterprise Services を使うソリューションでは、Enterprise Services を中間リモート層に置く必要がある場合を除き、可能なら**ライブラリアプリケーション**を使ってください。

リモート中間層によるパフォーマンスへの影響を理解する

可能な場合、プロセス間通信やコンピュータ間通信によるオーバーヘッドを避けてください。ビジネス要件によってリモート中間層の使用が必須の場合を除き、プレゼンテーション、ビジネス、データ アクセスの各ロジックは、Web サーバーに置いてください。ビジネスアセンブリとデータアクセスアセンブリは、アプリケーションの Bin ディレクトリに展開してください。ただし、以下のような理由によりリモート中間層が必要となる場合もあります。

- インターネット向け Web アプリケーションと、その他の内部エンタープライズアプリケーションの間でビジネスロジックを共有したい。
- スケールアウト要件と障害耐性要件により、中間層クラスタまたは負荷分散サーバーの使用を義務付けられている。

◆ 会社のセキュリティポリシーにより、Web サーバーへのビジネスロジックの展開を禁止されている。

リモート中間層を使用してアプリケーションを展開する必要がある場合、同一環境での計測およびテストが可能となるように、その必要性を早い段階から認識してください。

HTTP パイプラインを短くする

HTTP パイプライン シーケンスは、Machine.config ファイルの設定によって決められます。不要なモジュールはコメントに入れてください。たとえば、フォーム認証を使用しない場合は、Machine.config ファイルのフォーム認証用エントリをコメントに入れるか、特定のアプリケーションについて、Machine.config ファイルのエントリを明示的に削除してください。以下のサンプルでは、エントリをコメントアウトする方法を示しています。

```
<httpModules>
<!-- <add name="FormsAuthentication"
type="System.Web.Security.FormsAuthenticationModule"/> -->
</httpModules>
```

以下の Web.config ファイルサンプルでは、特定のアプリケーションについてエントリを削除する方法を示しています。

```
<httpModules>
  <remove name="FormsAuthentication" />
</httpModules>
```

使用していない HTTP モジュールを使っているアプリケーションが Web サーバーに他にもある場合は、アプリケーションの Web.config ファイルから HTTP モジュールを削除してください。この場合は、Machine.config ファイルの HTTP モジュールをコメントアウトするのではなく、上記のようにしてください。

メモリ制限を設定する

アプリケーションを展開する前にメモリ制限を設定してください。これにより、ASP.NET キャッシュのパフォーマンスとサーバーの安定性を最適化することができます。

トレースとデバッグを無効にする

アプリケーションを展開する前に、トレースとデバッグを無効にしてください。トレースとデバッグは、パフォーマンス上の問題を発生させる場合があります。アプリケーションが生産段階に入っている間にトレースとデバッグを行うのは好ましくありません

以下のようにして、Machine.config ファイルと Web.config ファイルでトレースとデバッグを無効にしてください。

```
<configuration>
  <system.web>
    <trace enabled="false" pageOutput="false" />
    <compilation debug="false" />
  </system.web>
</configuration>
```

コンテンツの更新によって追加的なアセンブリがロードされないようにする

.aspx ページや .ascx ページを更新してアプリケーションを再起動しないと問題が発生する場合があります。以下のシナリオについて考えてみてください。ディレクトリに、以下のような 4 つのページがあるとします。

```
¥mydir
Page1.aspx
Page2.aspx
Page3.aspx
Page4.aspx
```

Mydir ディレクトリのページが最初に要求されると、以下で示すようにディレクトリ内のすべてのページが 1 つのアセンブリにコンパイルされます。

```
Assembly1.dll {page1.aspx, page2.aspx, page3.aspx, page4.aspx}
```

Page1.aspx が更新されると、Page1.aspx のために新しいアセンブリが 1 つ作成されます。これにより、以下で示すように 2 つのアセンブリができます。

```
Assembly1.dll {page1.aspx, page2.aspx, page3.aspx, page4.aspx}
Assembly2.dll {page1.aspx}
```

Page2.aspx が更新されると、Page2.aspx のために新しいアセンブリが 1 つ作成されます。これにより、以下で示すように 3 つのアセンブリができます。

```
Assembly1.dll {page1.aspx, page2.aspx, page3.aspx, page4.aspx}
Assembly2.dll {page1.aspx}
Assembly3.dll {page2.aspx}
```

このような問題により、複数のアセンブリが生成されるのを防ぐにはコンテンツ更新時に以下のステップに従ってください。

1. Web サーバーをローテーションから外す。
2. IIS を再起動する。
3. Temporary ASP.NET Files フォルダ内のファイルをすべて削除する。

4. 各ディレクトリについて 1 ページを要求し、各ディレクトリがバッチコンパイルされることを確認する。
5. Web サーバーをローテーションに戻す。

コンテンツの更新に対するこのアプローチにより、他の問題も解決します。バッチコンパイルの完了前にサーバーがローテーションに戻されると、一部のページが 1 つのアセンブリとしてコンパイルされる場合があります。バッチコンパイルされているのと同じディレクトリのページのバッチコンパイル中に別の要求が出されると、そのページは 1 つのアセンブリとしてコンパイルされます。Web サーバーをローテーションから外し後で戻すことにより、この問題を避けることができます。

負荷の大きい状況では XCOPY を避ける

XCOPY 展開では、アプリケーションや IIS を閉じる必要がないため、展開は容易になります。しかし、生産環境では、サーバーをローテーションから外し、IIS を閉じ、XCOPY 更新を実行してから、IIS を再起動し、サーバーをローテーションへ戻すべきです。

負荷の高い状況では、この手順に従うことは特に重要です。たとえば、仮想ディレクトリの 50 のファイルをコピーし、各ファイルのコピーに 100 ミリ秒を要する場合、ファイル全体をコピーするのに 5 秒かかります。この間、アプリケーションのアプリケーションドメインは、アンロードとロードを複数回繰り返すかもしれません。また、一定のファイルは、XCOPY プロセス (Xcopy.exe) によってブロックされる場合があります。あるファイルを XCOPY プロセスがロックした場合、ワーカー プロセスとコンパイラは、そのファイルにアクセスできません。

更新のために XCOPY 展開を使いたければ、.NET Framework には waitChangeNotification 設定と maxWaitChangeNotification 設定が含まれています。これらの設定を使い、ここで説明した XCOPY に関する問題を解消できます。

waitChangeNotification の設定値は、最も大きいファイルをコピーするのに XCOPY が要する、総時間に基づいて決めるべきです。maxWaitChangeNotification 設定の値は、すべてのファイルをコピーするのに XCOPY が要する総時間プラス少しの予備時間に基づいて決めるべきです。

ページのプリコンパイルを検討する

ユーザーが ASP.NET ファイルのバッチコンパイルをしなくて済むようにするためには、各ディレクトについて 1 ページに 1 つの要求を送り、Web サーバーをローテーションに戻す前にプロセッサが再びアイドル状態となるまで待機します。これにより、バッチコンパイルを開始できるようになります。その結果、ユーザーの体感パフォーマンスは向上し要求の処理と同時にディレクトリのバッチコンパイルを実行する負担は軽減されます。

Web ガーデン設定を考慮する

アプリケーションが STA オブジェクトを頻繁に使う場合や、アプリケーションがプロセス数の制限を受けるリソースプールにアクセスする場合は **Web ガーデン**の使用を検討してください。

アプリケーションにおける Web ガーデンの効力を判断するには、パフォーマンステストを行い、Web ガーデンを使った場合と使わなかった場合で結果を比べてみてください。

HTTP 圧縮の使用を検討する

ほとんどのブラウザー、および IIS は、**HTTP 圧縮**をサポートしています。クライアントが狭帯域接続で Web サーバーにアクセスする場合、通常は HTTP 圧縮によってパフォーマンスが向上します。

周辺キャッシュの使用を検討する

周辺ネットワークは、インターネットや他の大型ネットワークからのアクセスを制御することにより、イントラネットを外部侵入から保護します。周辺ネットワークは、プロキシ サーバー、パケットフィルタ、ゲートウェイなどの、2 つ以上のネットワークの間に境界を設けるシステムの組み合わせで成り立っています。

周辺ネットワークにプロキシサーバーが含まれている場合、プロキシサーバーでのキャッシングを可能にし、パフォーマンスを向上させることを検討してください。

おわりに

このドキュメントは、ASP.NET を中心に、クライアントサイドからサーバーサイドに至るまでの広範囲な Web 開発テクノロジの情報をまとめたガイドラインです。

一言で Web 開発といっても、多種多様のテクノロジとプラットフォームがあります。それらの中でも、マイクロソフトの Web テクノロジとプラットフォームに関する技術情報は膨大で、体系的に参照することはとても難しいものと思われていました。

そこで、マイクロソフト テクノロジをベースとした Web 開発で、欠かすことのできないトピックを一つのガイドラインとして再編成したものが、このドキュメントになります。

コンテンツのベースとして、MSDN ライブラリ（英語および日本語）にある Web 開発関連のドキュメントを取捨選択して翻訳や編集をおこなっています。さらに必要と思われる部分は加筆および修正をして、ASP.NET プログラミングのエッセンスをまとめました。

このドキュメントが、Web 開発に携わるエンジニアの皆様のお手元で少しでも役立つことを願っています。

マイクロソフト株式会社