1-6 .NET 5

.NET Frameworkは上記のとおりWindowsにおけるプログラミング言語共通のアプリケーションプラットフォームとして進化してきました。このため、当然ですがそのプラットフォームはWindowsに限定されています。そしてこれは、共通言語ランタイム（CLR）がWindows用の仮想マシンということでもあります。

.NET Frameworkが、Windows用の仮想マシンだけでなく、Linux用やmacOS用などの仮想マシンとしても用意されれば、ひとつのアプリケーションがクロスプラットフォームで動作することになります。

このような試みとして「.NET Core」、「Xamarin」（コラム参照）があり、これらが統合された「.NET 5」があります。.NET 5は、2020年11月に.NET Core3.1の後継バージョンとしてリリースされましたが、2021年以降も毎年11月に後継バージョンがリリースされる予定です。一方、.NET Frameworkは、2019年リリースの「.NET Framework 4.8」が最終バージョンとなり、.NET 5に統合されます。

1-6-1 .NET 5とは

さて、コラムで示したとおり、それぞれが特徴を持つアプリケーションプラットフォームですが、.NET 5は、.NET Coreの後継バージョンということだけでなく、.NET Framework、およびXamarinなども統合されたものとなります（Xamarinとの完全な統合は、.NET 5では見送られ、2021年リリースの.NET 6で実現されるようです）。

つまり今後のMicrosoftのアプリケーションプラットフォーム（アプリケーションフレームワーク）は、クロスプラットフォームを前提とした.NET 5とその後継バージョンにより提供されることになります。

この背景には、いうまでもなくスマートフォンの存在があります。PCでは圧倒的なシェアを持つWindowsですが、更に大きなマーケットを持つスマートフォンのOSとしては、Android、iOSなどが主流です。これに正面から対応するためのアプリケーションフレームワークとして.NET 5があり、その流れがあります。

ちなみに、.NET Core同様、.NET 5もオープンソースとなります。また、.NET 5のC#の規定バージョンは、C# 9.0になります。

C O L U M N

.NET 5以前のクロスプラットフォームと.NET Framework

■ .NET Core

.NET Coreのプラットフォームには Windowsの他、Linux、macOSなどがあります。.NET Coreは、マルチプラットフォーム共通のアプリケーションプラットフォームとして、また、.NET Frameworkのサブセットとして、サーバーサイドでの利用が想定されています。これにより、WindowsのVisual Studioで開発した.NET Coreアプリケーションは Windowsだけでなく、LinuxやmacOSなどでも動作するクロスプラットフォームのアプリケーションとなります。

■ Xamarin

クロスプラットフォームのアプリケーション開発はスマートフォンのプラットフォームであるAndroidやiOSなどでも求められます。これらのアプリケーションの多くはGUIを持つため、これらを含めた.NET Framework互換のアプリケーションプラットフォームが必要となります。これに対応するものとして「Xamarin」があり、対象のプラットフォームとしてAndroid、iOS、macOS、UWPなどがあります。ちなみに、Xamarinのベースは「Mono」といわれるもので、同じくMonoをベースとするものとしてUnityがあります。

■ .NET Standard

.NET Framework、.NET Core、およびXamarinはそれぞれ進化していますが、これらの基本クラスライブラリ（基本的な機能を提供するクラスライブラリ群）の共通仕様が「.NET Standard」といわれるものです。複数の環境で開発する場合、クラスライブラリの仕様が基本クラスライブラリだけでも共通化されていることは開発効率の向上や互換アプリケーションの開発のしやすさなど、大きなメリットがあります。

■ .NET Framework 4.8

.NET Frameworkは、.NET Framework 4.8が最終バージョンとなり、メンテナンス（バグやセキュリティ対応）ベースのサポートとなります。また、そのサポート期限は対象のOSのサポート期限までは継続されます。

上記のとおり.NET Framework 4.8のサポートは、OSのサポート期限以降は不明となるため、新規のアプリの開発は、.NET 5以降のバージョンでの開発が基本になります。また、既存のアプリも.NET 5以降のバージョンでのアップデートを検討する必要があります。

Frameworkクラスライブラリにおいてもリソースを使用している多くのクラスでこのIDisposableインターフェイス(Disposeメソッド)が実装されています。このため、これらのクラスを使用する場合などもusingステートメントを使用することで、より品質のよいコードを記述することができます。

リスト6.7　usingステートメント

```
try
{
    using (Example_Using example_Using = new Example_Using("Using"))
    {
        System.Threading.Thread.Sleep(3 * 1000);
    }
}
catch (Exception exception)
{
    Console.WriteLine(exception.StackTrace);
}
finally
{
    Console.WriteLine("---- end of Using");
}
```

6-7-7　using宣言

上記のusingステートメントでなく、単に変数の宣言にusingキーワードを付与することで、その変数のスコープベースでusingステートメントと同様にDisposeメソッドが呼び出されます(using宣言は、C#8.0からサポートされています)。

たとえば、上記のusingステートメントは、次のようにusing宣言を使用することもできます。

```
using Example_Using example_Using = new Example_Using("Using");
System.Threading.Thread.Sleep(3 * 1000);
```

6-8　メソッド

メソッドはクラスの機能を提供する基本的な関数メンバです。ちにみに、メソッドは構造体、およびインターフェイスのメンバとしても定義することができます。また、コンストラクタやデストラクタ、そしてプロパティなども広い意味ではメソッドのひとつです。

6-8-1　メソッドの定義

メソッドは、次の書式で定義されます(表6.9)。

```
【<属性>】…【private | internal | protected | public】
【extern】【static】【new】【virtual】【override】【abstract】【sealed】【async】
【ref】<返却データ型>  <メソッド名>(【<フォーマルパラメータリスト>】)
{
    【<ブロック>】
}
```

たとえば、次のように記述します。

```
private int Max(int x, int y)
{
    int max = Math.Max(x, y);
    return max;
}
```

表6.9　メソッドの書式

項目	説明	備考
<属性>	メソッドに対する注釈情報などを指定します	属性の詳細については、「11-3 属性」で解説します
<返却データ型>	メソッドから返却されるデータ型を指定します	
<フォーマルパラメータリスト>	それぞれのパラメータのデータ型などを指定します	フォーマルパラメータリストの詳細については、「型とメンバ」の「5-2-6 関数メンバとパラメータ」で解説しています

■ =>演算子による式形式の実装

メソッドの実装がひとつの式の場合、その定義の短縮形として「=>演算子」を使った記述が可能です。たとえば上記のメソッド(Max)は、=>演算子を使って次のように記述できます。

```
private static int Max(int x, int y) => Math.Max(x, y);
```

読みやすさは慣れと好みによりますが、より簡潔な記述が適している場面で有効です。

■ 値型返却値の参照渡し(ref返却値)

返却データが値型の場合、返却データの受け取りにおいてそのデータが複製されるため、refローカル変数と同様、大きな構造体などを繰り返し返却する場合などでは、性能面が課題となる場面があります。このような場面では、返却データ型の前に「ref」を付与することで返却値を参照で

6-9-4 プロパティパターン（名前付きパターンマッチング）

プロパティ名とその値によるパターンマッチングを次の書式で利用することができます。

```
{<マチングプロパティ>, …}
```

<マッチングプロパティ>は次のとおりです。

```
<プロパティ名>:【<マッチング値>｜<マッチング変数>】
```

たとえば、switch式のパターンマッチングにおいて次のように利用することができます。

```
string CheckSchedule(Schedule schedule) => schedule switch
{
    {UserName: "knowlbo", Subject: "打ち合わせ"} => "knowlbo：打ち合わせ",
    {UserName: var userName, Subject: _ } when (userName == "knowlbo") => "knowlbo：スケジュール",
    _ => "*：etc."
};
```

この名前付きのパターンマッチングのパターンは「プロパティパターン」とよばれます（プロパティパターンは、C# 8.0からサポートされています）。また、プロパティパターンは、is演算子においても同様に記述することができます。

```
schedule1 is { UserName: "knowlbo", Subject: "打ち合わせ" }
```

6-10 演算子

演算子はメソッド名を簡易な記号でサポートする特殊なメソッドと考えることができます。ちなみに、構造体のメンバとしても定義することができます。利用できる演算子は**表6.11**のとおりです。また、新たに定義された演算子は演算子のオーバーロード、あるいはユーザー定義演算子とよばれます。

表6.11 演算子メンバ

カテゴリ	単項演算子				二項演算子					
算術	+	-	++	--	+	-	*	/	%	
論理	true	false								
ビット	!	~			&	|	^	<<	>>	
比較					==	!=	>	<	>=	<=

6-10-1 ユーザー定義演算子の定義

ユーザー定義の単項演算子は、次の書式で定義されます。

```
【<属性>】…【extern】…
public static <データ型> operator <単項演算子>(<データ型> <フォーマルパラメータ>)
{
    【<ブロック>】
}
```

ユーザー定義の二項演算子は、次の書式で定義されます。

```
【<属性>】…【public】【extern】【static】
public static <データ型> operator
<二項演算子>(<データ型> <フォーマルパラメータ>,<データ型> <フォーマルパラメータ>)
{
    【<ブロック>】
}
```

暗黙の変換の変換演算子は、次の書式で定義されます。

```
【<属性>】…【public】【extern】【static】
public static implicit operator <変換先データ型>（<変換元データ型> <フォーマルパラメータ>）
{
    【<ブロック>】
}
```

実際は一行

明示的な変換の変換演算子は、次の書式で定義されます。

```
【<属性>】…【public】【extern】【static】
public static explicit operator <変換先データ型>（<変換元データ型> <フォーマルパラメータ>）
{
    【<ブロック>】
}
```

実際は一行

いコンストラクタを定義することもできません。しかし、パラメータを持たない構造体を生成することは可能です（すべてのフィールドが0初期化されます）。

■ readonly修飾子

readonly修飾子は、その構造体を読み取り専用（変更不可）とします。これにより構造体が読み取り専用となるため、そのフィールドすべてにも明示的にreadonly修飾子が必要となります(注1)。また、プロパティは<getアクセサ>のみ定義でき、<setアクセサ>は定義できません。

● 構造体のメソッド呼び出しにおけるコンパイラの防衛的コピー

あるクラスのフィールドとして構造体を考えます。そして、この構造体フィールドがreadonlyの場合、その構造体のメソッドの呼び出しにおいて、そのメソッドはその構造体のコピーに対して呼び出されます。これは、そのメソッド内で構造体の内容が変更されてもreadonlyの指定を保証するためのコンパイラによる処置で、「防衛的コピー」などといわれます。防衛的コピーは大きな構造体や、そのメソッドが数多く呼び出される場合などでは性能的な問題となりますが、対象の構造体が読み取り専用の場合は、その防衛的コピーは抑制されます。

● 構造体メンバへのreadonly修飾子

構造体単位ではなく、それぞれのメンバにreadonly修飾子を付与することもできます。これによりメンバごとに読み取り専用（変更不可）を指定することができます。そしてこのメンバがメソッドであれば、その呼び出しにおいて「防御的コピー」を抑制することができます（構造体のメンバのreadonly修飾子は、C#8.0からはサポートされています）。

■ ref修飾子

ref修飾子は、その構造体のインスタンスがスタックメモリに限定されることを宣言します(注2)。ref修飾子を持つ構造体は「ref 構造体」などとよばれ、ヒープメモリへの割り当てを防止することができます。スタックメモリはヒープメモリに比べメモリ確保がシンプルで低コストのため、パフォーマンスの向上が見込めます。また、ref 構造体はクラスや通常の構造体のフィールドとして利用することはできません。コンパイルエラーとなります。

ref 構造体は、参照型の構造体（クラスのようなもの）ではなく、通常の構造体と同様、値型です。また、代表的なref 構造体として、Span<T>構造体があります。Span<T>構造体の詳細については、「**8-2-3 Span<T>構造体**」を参照してください。

（注1） 構造体のreadonly修飾子は、C# 7.2からサポートされています。
（注2） 構造体のref修飾子は、C# 7.2からサポートされています。

7-1-2 構造体のメンバ

構造体に配置可能なメンバは**表7.2**のとおりです。

構造体はクラスと違い値型でガベージコレクションの対象とならないため、デストラクタを定義することはできません。構造体はクラスと同様にメンバとして更に構造体を定義することで構造体を入れ子（階層的な構成）にすることも可能です。

表7.2 構造体に配置可能なメンバ

メンバ		説明	備考
データメンバ	フィールド	構造体の状態値を持つ変数	
	定数	構造体に関連付けられた定数	
	イベント	構造体のイベント	
関数メンバ	静的コンストラクタ	最初のインスタンスが生成される前、または静的メンバが参照される前に呼び出されるメソッド	
	インスタンスコンストラクタ	構造体のインスタンスが生成される時に呼び出されるメソッド	
	メソッド	構造体のプロシジャ	
	プロパティ	フィールドへのアクセスメソッド	
	演算子	構造体の演算子のオーバーロードメソッド	
	インデクサ	構造体要素へのインデックスアクセスメソッド	
型メンバ	クラス	構造体の入れ子クラス	コンテナ型メンバ
	構造体	構造体の入れ子構造体	
	インターフェイス	構造体の入れ子インターフェイス	
	列挙型	構造体の入れ子列挙型	
	デリゲート	構造体の入れ子デリゲート	

■ 演算子メンバ

演算子メンバについてサンプルプログラムで解説します。**リスト7.1**はComplexNumberという複素数の構造体を定義しています。ひとつの単項演算子（①）と2つの二項演算子（②、③）を定義しています。ComplexNumberクラスはコンストラクタでその実数値（real）と素数値（imaginary）を設定します。また、ToStringメソッドでは「<実数値>±<素数値>i」のフォーマットで複素数を文字列に変換します。

リスト7.1 演算子メンバ

```
public struct ComplexNumber
{
    public ComplexNumber(double real, double imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }
    private double real;
    private double imaginary;
    public override string ToString()
```

```
var hitIndexes = checkNumbers.IndexOfAny(stackalloc[] { 17, 51, 55 });
```

C# 8.0からは、上記のようなstackallocの式中の記述が可能となりました。

11-2-6 アンセーフなジェネリック構造体

ジェネリック構造体のすべてのタイプパラメータがアンセーフの場合、その構造体はアンセーフな構造体となります（これは、C# 8.0からサポートされています）。

ここで、アンセーフの型は次のとおりです。

sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal、bool列挙型、ポインタ型

11-3 属性

CLR（共通言語ランタイム）ではクラスやメソッドなどに対する注釈情報をメタ情報としてアセンブリに埋め込み、任意のプログラムから参照できる仕組みを用意しています。この注釈情報は「属性」とよばれます。

C#のコンパイラも参照先アセンブリの属性を参照して必要なコードを生成しています。

11-3-1 属性の指定

属性の記述位置は基本的に対象となる定義の先頭になります。ただし、アセンブリとモジュールに対しては、usingディレクティブより後で、型または名前空間の定義の前に記述します。ちなみに、この位置に記述される属性は「グローバル属性」とよばれます。

属性は、次の書式で指定します。

```
[【<対象項目>:】<属性>【, <属性>】… ] …
```

また、<属性>の書式は次のとおりです。

```
<属性名>【(【<パラメータ値>【,<パラメータ値>】…【,<キーワード> = <パラメータ値>】…】)】
<属性名>【(【<キーワード> = <パラメータ値>】…】)】
```

表11.3に示す<対象項目>は、その属性の対象を明示的に指定するもので、たとえばメソッドに属性を記述する場合、<対象項目>を指定しないとメソッドが対象となりますが、「return」を指定することで戻り値に属性を指定することができます（**表11.4**）。

表11.3 属性の対象項目

対象項目	説明
assembly	アセンブリを対象項目とします
event	イベントを対象項目とします
field	フィールドを対象項目とします
method	メソッドを対象項目とします
module	モジュールを対象項目とします
param	パラメータを対象項目とします
property	プロパティを対象項目とします
return	戻り値を対象項目とします
type	型を対象項目とします

表11.4 属性の対象項目

記述対象	指定可能な対象項目						
アセンブリ	assembly						
モジュール	module						
class struct interface enum		type					
delegate		type					return
フィールド			field				
メソッド					method		return
プロパティGetアクセサ					method		return
プロパティSetアクセサ				param	method		return
プロパティインデクサ						property	
イベントフィールド		event	field		method		
イベントプロパティ		event				property	
イベントアド				param	method		
イベントリムーブ				param	method		
パラメータ				param			

11-3-2 属性の内部イメージ

属性はクラスやメソッドなどに対する注釈情報で、コンパイルとともにアセンブリに保持されます（**図11.2**）。

Webフォームではサーバコントロールの存在から、最終的に出力されるHTMLが制御しにくく、包括的なデザインが困難でした。

16-1-1 ASP.NETとASP.NET Core

ASP.NET MVCは、ASP.NETというWebフレームワークのなかのサブフレームワークです。2010年に初期バージョンのASP.NET MVC 1.0が登場し、その後は定期的にバージョンアップを繰り返してきました。登場からしばらくは低迷していましたが、フレームワークとして成熟するとともにWebフォームに取って代わって、ASP.NETにおけるWebフレームワークの主流となりました。ASP.NET MVCの最終バージョンは5.2.7となっています。

2016年、Webフレームワークに「ASP.NET Core」が登場しました。ポジションとしては、ASP.NETの直接的な後継でなく、新しい概念を取り込んだ新シリーズです（そのために「ASP.NET X」ではなく「ASP.NET Core X」という命名になっています）。ASP.NET Coreの一番の特徴は「.NET Core」上で動作するクロスプラットフォームなフレームワークだということです。

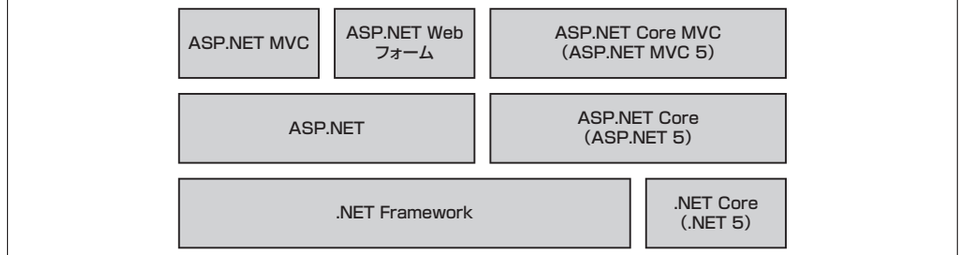
.NET Coreとは、クロスプラットフォームな（プラットフォームに依存しない）アプリケーション実行ランタイムです。フレームワークのレイヤーとしては.NET Frameworkと同じ位置ですが、WindowsOSでしか動作しない.NET Frameworkに対し、Mac、Linuxといった非WindowsのOS環境でも動作します。プロプロイエタリである.NET Frameworkに対し、オープンソースであることも特徴です。

ASP.NET Coreは、この.NET Coreランタイム上で動作します。よって、動作環境がWindows OSのみのASP.NETに対し、ASP.NET CoreではMac、LinuxOS上でも動作します。また使用できるWebサーバーについても、Windowsに組み込みのIIS（インターネット・インフォメーション・サービス）に依存していたASP.NETに対し、ASP.NET CoreではIIS以外にKestrelやNginxといったWebサーバーを選択できます。

またASP.NET Webフォームについては.NET Coreによっては動作しません（.NET Core上で動作するこれらの移植フレームワークは存在しません）が、ASP.NET MVCについては移植され、ASP.NET Core上で動作するASP.NET Core MVCというフレームワークが提供されています。

これらの関係を図に示したものが**図16.1**です。.NET Core 3.1の次のバージョンとしてリリースされた.NET Core系の後継である.NET 5は、.NET Coreの特徴をそのまま引き継いでいます。

図16.1 .NET Frameworkと.NET Core



16-1-2 ASP.NET MVCとASP.NET Core MVC

ASP.NET MVC Coreは、ASP.NET Coreを基盤として動作するWebアプリケーションフレームワークです。ASP.NET MVCに引き続いて、MVCパターンで設計されています。

開発においては、大枠ではASP.NET MVCと同じように実装可能です。ただしアプリケーションの観点では、おもに以下の点で異なります。

- 構成ファイルが「appsettings.json」というjson形式のファイルであること（ASP.NET MVCではXML形式の「Web.config」）。また構成ファイルの構成を変更できること。
- スタンドアロンプログラムで、アプリケーションのスタートポイントがProgram.Main()メソッドであること（ASP.NET MVCではスタート時にGlobal.asaxクラスのインスタンスが作成される）
- アプリケーションのライフサイクルに伴うイベント処理をIApplicationLifetimeクラスに設定すること（ASP.NET MVCではGlobal.asaxに定義する）

16-1-3 サンプルアプリケーション（ASP.NET MVC）

では早速、ASP.NET MVCアプリケーションをVisual Studioを使って作成してみましょう。一例として「Visual Studio 2019 Professional」を用います。

ここではサンプルとして、ページ上に文字列を表示する、シンプルなアプリケーションを作成します。

① プロジェクトの作成

Visual Studioを起動し、起動後のスタートメニューから「新しいプロジェクトの作成」を選択します（もしくはメニューの「ファイル」-「新規作成」-「プロジェクト」）（**図16.2**）。

リスト16.1　DefaultController.cs

```
using System.Web.Mvc;

namespace SampleMVC.Controllers
{
    public class DefaultController : Controller
    {
        // GET: Default
        public ActionResult Index()
        {
            //return View();
            return new ContentResult() { Content = "Hello World. I'm ASP.NET MVC" };
        }
    }
}
```

④ 実行

ではビルドを行い実行します。メニュー「デバッグ→デバッグなしで開始」を選択します。

デフォルトでは「https://localhost:xxxxx/」といったような開発環境のアドレスがブラウザでリクエストされます。この実行結果ではアプリケーションのHomeページが表示されます（**図16.9**）。

ここでは、ブラウザのアドレスを「https://localhost:xxxxx/Default/」としてアクセスしましょう。すると**図16.10**のような実行画面となります。「~/Default/」リクエストに対しては「DefaultController.Index()」がルーティングされ単純な文字列コンテンツを返却した、という実行結果となります。

URLとコントローラクラス（メソッド）のルーティングという概念が出てきていますが、現段階では「リクエストに対して、対応するコントローラクラス（メソッド）がマッピングされる仕組みがある」とだけ理解してください。以降、順次詳細の説明を行います。

図16.9　実行画面（https://localhost:xxxxx/）

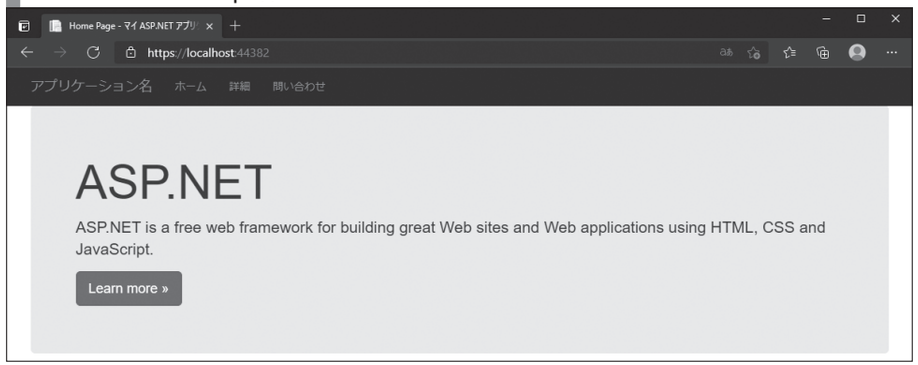


図16.10　実行画面（https://localhost:xxxxx/Default）

16-1-4　サンプルアプリケーション（ASP.NET Core MVC）

今度は、ASP.NET Core MVCアプリケーションの作成フローを解説しましょう。同じく「Visual Studio 2019 Professional」を用います。

基本的な流れは同じですが、ASP.NET Core MVCのパターンで流します。サンプルアプリケーションの内容自体はASP.NET MVCのものと同じになります。ページ上に文字列を表示する、シンプルなアプリケーションです。

① プロジェクトの作成

Visual Studioを起動し、起動後のスタートメニューから「新しいプロジェクトの作成」を選択します（もしくはメニューの「ファイル」-「新規作成」-「プロジェクト」）。

新しいプロジェクトの作成に移るので、プロジェクトテンプレートとして「ASP.NET Core Webアプリケーション」を選択し、「次へ」で次に進みます（**図16.11**）。

図16.11　プロジェクトテンプレート

リスト16.53　セッション機能の有効化

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    …
    app.UseSession();
    …
}
```

リスト16.54　TempDataを使用したアクションメソッド

```
namespace SampleMVC.Controllers
{
  public class EmployeeController : Controller
  {
    // GET: /Employee/UseTempDataFrom/
    public ActionResult UseTempDataFrom()
    {
      this.TempData["EmpData"] = CustomizeConverter.Serialize(new Employee() {
        Id = "001",
        FirstName = "Taro",
        LastName = "Yamada" };);

      return Redirect("UseTempDataTo");
    }

    //
    // GET: /Employee/UseTempDataTo/
    public ActionResult UseTempDataTo()
    {
        return View();
    }
  }
}
```

リスト16.55　TempDataを利用したビュー（UseTempDataTo.cshtml）

```
@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<body>
  Employee:
  @{ SampleMVC.Models.Employee employee = CustomizeConverter.Deserialize(TempData["EmpData"]); }
  @employee.Id @employee.FirstName @employee.LastName
</body>
</html>

public class CustomizeConverter
{
    public static object Serialize(Employee employee)
    {
        // シリアライズ
    }

    public static Employee Deserialize(object)
    {
        // デシリアライズ
    }
}
```

C O L U M N

Webアプリケーションにおける.NET Coreと.NET 5

2020年11月にリリースされた.NET 5は、世代としては.NET Core 3.1の次のメジャーリリースです。.NET Framework 系の後継ではなく、.NET Core系の後継にあたります。そのため、クロスプラットフォームや、オープンソースライセンスといった.NET Core系の特徴を引き継いでいます。

.NET Coreを使用しているWebアプリケーションにおいては、基本的な仕組みが変わらないため、使用するフレームワークの.NET 5へのバージョンアップに伴って、実装も大きく変わるところはありません。細かい点において、一部パッケージ名の変更や、Blazorの機能向上などの点で堅牢性、機能性が向上しています。Webアプリケーションでは、.NET CoreのWeb開発フレームワークであるASP.NET Coreで動作するASP.NET Core MVCなどが該当します。

一方、.NET Frameworkフレームワーク（ASP.NET）を使用しているWebアプリケーション、たとえばASP.NET MVCは.NET 5への移行にあたっては.NET Coreへの移行と同程度の差異が発生します。ただし、ASP.NET Webフォームは.NET Frameworkでしかサポートされておらず、.NET Coreまたその進化系である.NET 5でもサポートされていません。.NET 5への移行には、たとえばASP.NET MVCといった別の代替となるアーキテクチャへ乗り換える必要があります。

リスト17.22　_ViewStart.cshtml

```
@{
  Layout = "~/Views/Shared/_LayoutPage.cshtml";
}
```

リスト17.22の_ViewStart.cshtmlを使用した場合、前述の**リスト17.16**は**リスト17.23**のように書き換えることができます。実は_ViewStart.cshtmlの記述内容は、すべてのビューページに対して適用されます。そのため、各ビューページでLayoutプロパティを明示的に設定をする必要はありません。

特定のページで別のレイアウトを使用したい場合、もしくはいずれのレイアウトも使用したくない場合は、ビューページ側で明示的な記述を行うことで_ViewStart.cshtmlでの設定をオーバーライドすることができます。

リスト17.23　ビューページ

```
@{
  ViewBag.Title = "View1";
  Layout = "~/Views/Shared/_LayoutPage.cshtml";
}

<h2>Index</h2>
```

17-3 モデルバインディングと入力検証

多くのWebページでは画面からのデータの入力をサーバ側で受け取ります。マスタ編集画面は勿論のこと、データ一覧表示画面では検索条件の入力もあるでしょう。これらの入力データはモデルバインディングという機能によって効率的にサーバ側で取得することができます。

17-3-1　画面（ページ）入力値の取得

画面（ページ）でのユーザ入力は一般的に「<input> <select> <textarea>」等のタグによって構築されます。そして、これらの入力用タグを含んだフォームがsubmitされると、入力値を含むHTTPリクエストがサーバに対して送信されます。

このような仕組みはHTTPレベルのWeb共通事項であり、ASP.NET MVCでもASP.NET Webフォームでも、またRuby on Railsでも変わることはありません。

それぞれのWebフレームワークは、submitされたデータを受け取る仕組みを、各々に提供しています。ASP.NET MVCではHTTPリクエストの内容を以下の方法で取得できます。

- **薄いラッパークラス経由で取得する方法**
- **より高度な自動モデルバインディングにより取得する方法**

■ 薄いラッパークラス経由で取得する方法

この方法は、実装されるクラスの違いは有れど、ASP（Active Server Pages）のようなWeb技術聡明期から存在する方式です。ASP.NET MVCでは、「System.Web.HttpRequestBaseクラス」で実装されています。コントローラ クラスでは「Requestプロパティ」としてHttpRequestBaseクラスオブジェクトが提供されています。

ここでは**図17.7**のような「PCスペックを入力する画面」を例として使用する事とします。ページの構成は以下のとおりです。

- **ビュー（入力画面）**
  Views/DataTransSample/Index.cshtml（**リスト17.24**）
- **コントローラークラス**
  Controllers/DataTransController.cs（**リスト17.25**）
- **ビュー（入力内容表示画面）**
  Views/DataTransSample/ReceiveData.cshtml（**リスト17.26**）

図17.7　PCスペック入力画面

まず、入力画面は「http://localhost:xxxx/DataTrans/Index/」をリクエストすることで表示されます。

入力画面のビューは**リスト17.24**ですが、フォーム定義、および3つの項目の入力用テキストボックス、submit用ボタンから構成されています。BeginFrom()でアクション名は「SaveByRequest」、コントローラ名は「DataTransSample」と指定されている為、submit時にはDataTransSampleController.SaveByRequest()アクションメソッドがPOSTで呼び出されます。

18章 MSTestによるユニットテスト

ユニットテスト（単体テスト）は、テストコードをクラスやメソッド単位でコーディングして、プログラム実行結果の判定（テスト）を細かい粒度で行う自動化テストです。ここでは.Net Framework標準の「MSTest」について解説します。

18-1 MSTestとは

現在ではユニットテストのサポートとして、Visual Studio Unit Testing Frameworkの流れをくむ「MSTest」フレームワークがVisual Studioに統合されて提供されています（また、ほかにはサードパーティ製のテストフレームワークとして「NUnit」「MBUnit」「xUnit.NET」などがあり、Visual Studioに統合して使用できます）。本章では、おもにVisual Studio上での操作を通して、MSTestによるユニットテストコード実装から実行までを説明します。

18-1-1 Visual StudioにおけるMSTest

MSTestは「Microsoft Visual Studio 2012」シリーズから、Visual Studio標準搭載のテストフレームワークとして、.Net Framework開発環境で使用できるようになっています。ツールとして、テストをコードベースで記述できる.Net Frameworkライブラリや、記述したテストコードの検出・実行・結果の確認ができるGUIツールのテストエクスプローラーが用意されているため、Visual Studio上でテストコードの実装から実行までを一貫して行えます。

次節ではVisual Studioにおける、MSTestフレームワーク（MSTest.TestFramework 1.3.2（MSTest v2）.NET 4.5.0+）の操作方法を、一例として「Visual Studio 2019 Professional」を用いて説明します。

18-2 ユニットテストプロジェクト

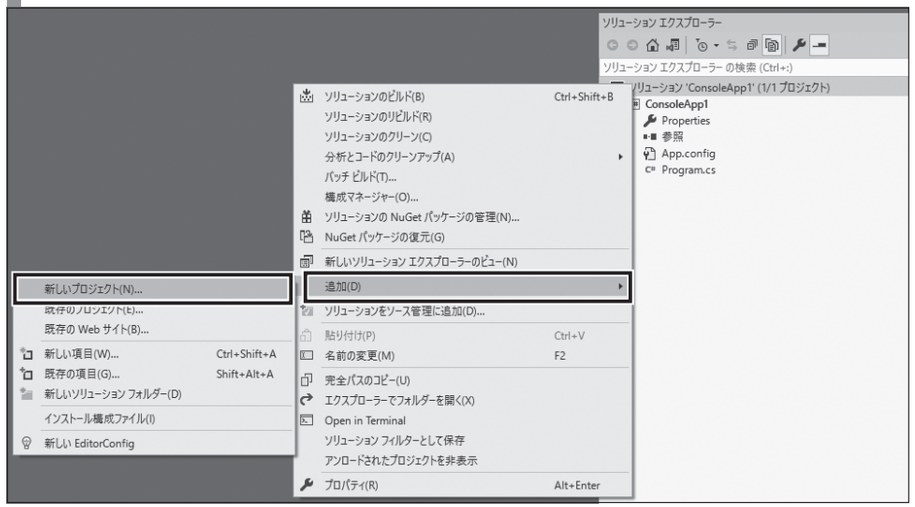
はじめに、Visual Studioにおけるテストプロジェクトの新規作成方法を解説します。既存のプロジェクトのコードに対するテストコードを実装する場合を想定し、既存のソリューションにテストプロジェクトを追加します。

18-2-1 ユニットテストプロジェクトの作成（追加）

まず、Visual Studioのソリューションエクスプローラー上からソリューションの操作で「追加」-「新しいプロジェクト」と選択します（もしくはメニュー「ファイル」-「新規作成」-「プロジェクト」）。

例として、「ConsoleApp1」プロジェクトを含む「ConsoleApp1」ソリューションにテストプロジェクトを追加します（**図18.1**）。

図18.1 ソリューションへのプロジェクトの追加



新しいプロジェクトの追加（プロジェクトテンプレートの選択）ダイアログが表示されます。コンソールアプリを対象とするので、ここでは「単体テスト プロジェクト（.Net Framework）」を選択して次へ進みます（**図18.2**）。