Visual Studio 2005 JScript

本文档中的内容已停用，将不再更新且不支持。某些链接可能无效。已停用的内容表示此内容的最近更新版本。

Visual Studio

Visual Studio 是一套完整的工具，用于生成桌面和基于团队的企业级 Web 应用程序。除了生成高性能的桌面应用程序外，还可以使用 Visual Studio 基于组件的强大开发工具和其他技术，简化基于团队的企业级解决方案的设计、开发和部署。

本节内容

Visual Studio 简介

找到有关 Visual Studio 新增功能的更多信息，了解有关 .NET Framework 的更多信息，并查找指向此版本 Visual Studio 入门的指针。

Visual Studio 集成开发环境

找到有关设计、开发、调试、测试、部署和管理用 Visual Studio 创建的应用程序的信息。

基于 Windows 的应用程序、组件和服务

确定生成应用程序和组件时使用的工具和技术，以及可使用 Visual Studio 创建的项。

Visual Studio 中的 .NET Framework 编程

了解在 Visual Basic、Visual C# 和 Visual J# 中开发应用程序时如何使用 .NET Framework。

Visual Basic

了解 Visual Basic 的新增功能，并研究如何使用 Visual Basic 开发应用程序。

Visual C#

了解 Visual C# 的新增功能，并研究如何使用 Visual C# 开发应用程序。

Visual C++

找到有关 Visual C++ 的新增功能的信息，以及发现如何使用 Visual C++ 开发应用程序。

Visual J#

找到有关 Visual J# 的信息，Visual J# 是一种工具，供 Java 语言程序员用于生成在 .NET Framework 上运行的应用程序和服务。

JScript

找到有关 JScript .NET（真正面向对象的脚本语言）的信息。

Visual Web Developer

了解 Visual Web Developer 并研究如何使用 Visual Web Developer 创建 Web 应用程序。

Visual Studio Team System

了解 Visual Studio 2005 Team System，这是一个高效、集成且可扩展的软件开发生命周期工具平台，可以帮助软件团队在整个软件开发过程中提高沟通和协作能力。

Visual Studio Tools for Office

了解如何创建商业应用程序，以利用 Microsoft Office 2003 的强大功能对信息进行收集、分析、操作或呈现。

智能设备开发

了解如何开发在基于 Windows CE 的智能设备（如 Pocket PC 和 Smartphone）上运行的软件。

工具和功能

了解 Crystal Reports、Windows Server 功能编程以及应用程序验证工具。

.NET Framework 示例

定位此版本的 Visual Studio 中提供的最新示例应用程序和示例。

快速入门

了解 .NET Framework SDK 附带的教程。

.NET Framework 词汇表

了解 .NET Framework 中使用的常用术语的定义。

相关章节

Visual Studio SDK

找到有关 Visual Studio 软件开发工具包 (SDK)（提供扩展性选项和工具包）的信息。

JScript

JScript 8.0 是一种有着广泛应用的现代脚本语言。它是一种真正面向对象的语言，不过仍保留其“脚本”特色。JScript 8.0 保持与 JScript 以前版本的完全向后兼容性，同时包含了强大的新功能并提供了对公共语言运行库和 .NET Framework 的访问。

以下主题介绍 JScript 8.0 的基本组件并提供有关如何使用该语言的信息。与任何现代编程语言一样，JScript 支持许多通用编程构造和语言元素。

若已经使用过其他语言编程，那么您可能熟悉本节中包含的很多资料。虽然大多数构造都与 JScript 的以前版本相同，但 JScript 8.0 引入了强大的新构造，这些构造类似于其他基于类的面向对象语言中的构造。

如果您是编程新手，本节中的资料可作为编写代码的基本构造块的介绍。当了解基础知识之后，您将能够使用 JScript 创建强大的脚本和应用程序。

本节内容

JScript 入门

介绍 JScript 8.0 的新增功能。

编写、编译、调试 JScript 代码

提供一组指向特定主题的链接，这些主题解释如何用 JScript 8.0 编写、编辑和调试代码。

使用 JScript 显示信息

包含一系列指向特定主题的链接，这些主题解释如何从命令程序、从 ASP.NET 以及在浏览器中显示信息。

正则表达式介绍

包含一个有关 JScript 8.0 中“正则表达式”的元素和过程的指南。其主题解释正则表达式的概念、正确的语法和适当的用法。

JScript 参考

列出“JScript 语言参考”所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

相关章节

Devenv 命令行开关

列出特定的语言参考主题，这些主题解释如何启动 Visual Studio 以及如何从命令提示处进行生成。

等效语言

比较 Visual Basic、C++、C#、JScript 和 Visual FoxPro 的关键字、数据类型、运算符和可编程对象（控件）。

.NET Framework 类库参考

包含指向一些主题（这些主题解释 .NET Framework 类库中的命名空间）的链接，并说明如何使用类库文档。

Visual Studio 命令和开关

包含特定的语言参考主题，这些主题解释如何从命令窗口和“查找/命令”框中使用命令与 IDE 进行交互。

Visual Studio 演练

提供指向特定主题的链接，这些主题讨论在开发特定应用程序时所涉及的步骤或如何使用主要的应用程序功能。

JScript 入门

JScript 8.0 在好几个方面代表了 JScript 语言的巨大进步。它与 Visual Studio 开发环境更紧密地集成在一起，有多种新功能，并可访问 .NET Framework 类，因此初看之下，从 JScript 转移到 JScript 8.0 似乎是一项骇人的工作。

事实上，几乎所有的更改都表现在附加功能上，而 JScript 的核心功能与先前版本是一样的。实际上所有 JScript 脚本都可无需更改即在 JScript 8.0 下运行（在关闭快速模式时）。若要在快速模式（ASP.NET 支持的模式）下运行，则某些脚本将需要稍作修改。

由于可以逐步地将新功能包含到您的代码中，因此完成从 JScript 到 JScript 8.0 的过渡并不困难。您可以按照自己的计划升级您的脚本，随着对 JScript 8.0 的了解增多，不断添加新的功能。

下面的文档有助于升级您的应用程序并快速了解 JScript 8.0 中所作的更改。

本节内容

什么是 JScript 8.0？

提供 JScript 8.0 的总体概述、它与 ECMAScript 的关系以及它从先前的 JScript 版本演变而来的过程。

JScript 8.0 的新增功能

列出 JScript 8.0 中的新功能，其中包括与 ECMAScript 版本 4 一起开发的功能和未在 ECMAScript 中指定的附加功能。

JScript 8.0 for JScript Programmers 简介

介绍 JScript 和 JScript 8.0 间的区别，为有经验的 JScript 程序员迅速提供背景情况。

Hello World! 的 JScript 版

阐释如何在 JScript 8.0 中编写熟悉的 Hello World! 程序。

升级先前的 JScript 版本中创建的应用程序

说明如何升级现有的 JScript 应用程序使之能够在 JScript 8.0 中运行。

在以前版本的公共语言运行库上运行 JScript 应用程序

解释如何将使用某个版本的运行库生成的 JScript 8.0 应用程序配置为在以前版本的运行库上运行。

为 JScript 程序员提供的附加资源

列出了一些为常见 JScript 编程问题提供答案的站点和新闻组。

相关章节

JScript 版本信息

列出 JScript 的所有功能以及引入每个功能的相应版本。

JScript 语言教程

介绍开发人员用来编写 JScript 代码的元素和过程，并提供链接，指向阐释语言元素和代码语法的详细背景信息的特定区域。

JScript 参考

列出 JScript 语言参考所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

什么是 JScript 8.0？

JScript 8.0 是 Microsoft 实现 ECMA 262 语言的新一代版本。通过将先前版本的 JScript 功能集与基于类的语言的最佳功能相结合，JScript 8.0 包含了这两个领域的精华。JScript 8.0 的改进内容（正在结合 ECMAScript 第 4 版开发研制）包括：真正的编译代码、类型化和无类型变量、后期和早期绑定、类（具有继承、函数重载、属性访问器，等等）、包、跨语言支持，以及对 .NET Framework 的完全访问。

新增功能

JScript 8.0 是一种真正的面向对象的脚本语言。尽管 JScript 8.0 现在可以使用类、类型和其他高级语言功能来编写可靠的应用程序，但它仍然保留着其"脚本"特色，支持无类型编程、expando 函数和类、动态代码执行（使用 **eval**）等。

JScript 8.0 不仅是一种无类型语言，现在它还可以用作一种强类型化的语言。在先前版本中，JScript 松散的类型化结构意味着变量采用赋给它们的值的类型。实际上，在先前版本中用户不能声明变量的数据类型。JScript 8.0 允许对变量进行类型批注，这与先前版本的 JScript 相比提供了更大的灵活性。这种方法将变量绑定到某个特定的数据类型，这样该变量就只能存储这种类型的数据。

在编程语言中，强类型有许多优势。除了当所使用的数据类型恰好适合正在使用的数据时获得的益处外，还有其他几点好处：

- 提高执行速度
- 运行时/编译时类型检查
- 自记录代码

最后，JScript 8.0 并不是另一种编程语言的简单版本，也不是对任何内容的简化。而是一种应用非常广泛的现代脚本语言。

注意
包含在许多 JScript 8.0 示例中的代码在一定程度上比实际脚本更明白更简洁。其目的是阐明概念，而不是表述最佳的编码简明程度和风格。无论如何，编写出在六个月后都能够轻松阅读并理解的代码是一种很不错的做法。

请参见

其他资源

JScript 参考

JScript 8.0 的新增功能

JScript 8.0 是新一代的 Microsoft JScript 语言，它是使用 Web 语言方便快捷地访问 Microsoft .NET 平台的一种方法。JScript 8.0 的主要作用是使用 ASP.NET 构造网站和使用 .NET Framework 脚本自定义应用程序。

JScript 8.0 与 ECMAScript 标准兼容，而且它还具有 ECMAScript 未提供的其他功能，例如，真正的编译代码、通过符合"通用语言规范"(CLS) 而实现的跨语言支持，以及对 .NET Framework 的访问。Visual Studio .NET 2002 中的 JScript .NET 版本充分利用了 .NET Framework 本身所具有的安全性，而 JScript .NET 2003 为 **eval** 方法添加了受限安全上下文，从而进一步增强了安全性。

在 JScript 8.0 中的几种新功能旨在充分利用 CLS，这是用来标准化数据类型、对象公开方式、对象互用方式等内容的一组规则。任何符合 CLS 的语言都可以使用在 JScript 8.0 中创建的类、对象和组件。作为 JScript 开发人员，您可以从其他符合 CLS 的编程语言访问类、组件和对象，而无需考虑语言特定的差异(比如数据类型)。JScript 7.0 程序使用的一些 CLS 功能包括命名空间、属性、引用参数和本机数组。

下面是 JScript .NET 和 JScript 8.0 中的一些新功能：

JScript 8.0 的新增功能

/platform 编译器选项

/platform 选项用于指定输出文件面向的处理器类型：x86 针对 32 位 Intel 兼容处理器，Itanium 针对 Intel 64 位处理器，x64 针对 AMD 64 位处理器。默认值 (anycpu) 允许输出文件在任何平台上运行。

JScript .NET 2003 的新增功能

eval 方法的受限安全上下文

为了维持安全性，不管调用方的权限是什么，内置的 **eval** 方法现在都默认在受限安全上下文中运行脚本。调用 **eval** 时如果将"unsafe"作为第二个可选参数，会导致脚本使用调用方的权限运行，这样就会允许访问文件系统、网络或用户界面。有关更多信息，请参见 eval 方法。

JScript .NET 2002 的新增功能

基于类的对象

JScript .NET(像 JScript 一样)通过基于原型的对象支持继承。JScript .NET 还允许声明定义对象数据和行为的类，从而支持基于类的对象。在 JScript .NET 中创建的类可由任何 .NET 语言使用和扩展。类可以继承基类的属性和方法。可以对类和类成员应用几种属性，修改它们的行为和可见性。有关更多信息，请参见基于类的对象。

JScript 数据类型

在 JScript .NET 中(像 JScript 一样)，可以在编写程序时不指定变量的数据类型。JScript .NET 也可用作一种强类型化语言，其中所有变量都绑定到特定数据类型，或者您还可以混合使用类型化和非类型化变量。JScript .NET 提供了许多新的数据类型。还可以将类和 .NET 类型用作数据类型。有关更多信息，请参见 JScript 数据类型。

条件编译

指令可控制 JScript .NET 程序的编译。例如，@debug 指令可以为脚本的特定部分打开或关闭调试信息的发布。有关更多信息，请参见 @debug 指令。@position 指令为调试器设置当前行的行号。有关更多信息，请参见 @position 指令。如果正在编写将合并到其他脚本中的代码，那么这两个指令都是有用的。有关更多信息，请参见条件编译。

JScript 命名空间

命名空间通过将类、接口和方法组织成层次结构来防止命名冲突。在 JScript .NET 中，您可以定义自己的命名空间。还可以使用 JScript .NET 访问任何 .NET Framework 命名空间，包括自己定义的那些命名空间。包语句允许打包相关的类以实现方便的部署和避免出现命名冲突。有关更多信息，请参见 package 语句。导入语句使得 .NET Framework 命名空间可用于某个脚本，这样该脚本就能访问命名空间中的类和接口。有关更多信息，请参见 import 语句。

JScript 变量和常数

JScript .NET 引入了一个 Const 语句，用来定义表示常数值的标识符。有关更多信息，请参见 JScript 变量和常数。

枚举

JScript .NET 引入了 Enum 语句，允许构造枚举数据类型。利用枚举，可以为您的数据类型值指定有用的名称。有关更多信息，请参见 enum 语句。

请参见

概念

JScript 8.0 for JScript Programmers 简介

此处提供的信息主要面向已经熟悉 JScript 并想了解 JScript 8.0 中引入的新功能的程序员。

常见任务

如何编译程序

JScript 8.0 命令行编译器从 JScript 程序创建可执行文件和程序集。有关更多信息，请参见 如何：从命令行编译 JScript 代码。

如何编写“Hello World”程序

编写“Hello World”的 JScript 8.0 版本非常容易。有关更多信息，请参见 Hello World! 的 JScript 版。

如何使用数据类型

在 JScript 8.0 中，冒号指定变量声明或函数定义中的类型。默认类型为 **Object**，它可以保存任何其他类型。有关更多信息，请参见 JScript 变量和常数和 JScript 函数。

JScript 8.0 具有几种内置数据类型（如 **int**、**long**、**double**、**String**、**Object** 和 **Number**）。有关更多信息，请参见 JScript 数据类型。还可以在导入适当的命名空间后使用任何 .NET Framework 数据类型。有关更多信息，请参见 .NET Framework 类库参考。

如何访问命名空间

访问命名空间可使用 **import**（导入）语句（当使用命令行编译器时）或 **@import** 指令（当使用 ASP.NET 时）。有关更多信息，请参见 import 语句。**/autoref** 选项（该选项在默认情况下为打开状态）自动尝试引用与 JScript .NET 程序中使用的命名空间相对应的程序集。有关更多信息，请参见 /autoref。

如何创建类型化（本机）数组

类型化数组数据类型可以通过在数据类型名称后面放置方括号 ([]) 来声明。您仍然可以使用 JScript 数组对象（即，用 **Array** 构造函数创建的对象）。有关更多信息，请参见数组概述。

如何创建类

在 JScript 8.0 中，可以定义您自己的类。类可以包含方法、字段、属性、静态初始值设定项和子类。可以编写一个全新的类，也可以从现有的类或接口继承。修饰符控制类成员的可见性、如何继承成员以及类的总体行为。也可以使用自定义属性。有关更多信息，请参见基于类的对象和 JScript 修饰符。

请参见

概念

升级先前的 JScript 版本中创建的应用程序

其他资源

JScript 入门

Hello World! 的 JScript 版

以下控制台程序是传统"Hello World!"程序的 JScript 版，该程序显示字符串 `Hello World!`。

示例

```
// A "Hello World!" program in JScript.
print("Hello World!");
```

本程序的要点如下所示：

- 注释
- 输出
- 编译和执行

注释

此示例的第一行包含一个注释。由于编译器忽略此注释，因此可以编写任何文本。此注释说明程序的目的。

```
// A "Hello World!" program in JScript.
```

双正斜杠 (`//`) 表示此行的其余部分是一个注释。可以将整行作为注释，或者可以在其他语句的结尾追加一个注释，如下所示：

```
var area = Math.PI*r*r;  // Area of a circle with radius r.
```

还可以使用多行注释。多行 JScript 注释以正斜杠和星号 (`/*`) 开头，以相反的顺序 (`*/`) 结束。

```
/*
Multiline comments allow you to write long comments.
They can also be used to "comment out" blocks of code.
*/
```

有关更多信息，请参见 JScript 注释。

输出

此示例使用 **print** 语句显示字符串 `Hello World!`：

```
print("Hello World!");
```

有关更多信息，请参见 print 语句。

程序与用户之间还有其他通信方式。System.Console 类公开了更便于与使用控制台的人进行交互的方法和属性。有关更多信息，请参见 **Console**。System.Windows.Forms.MessageBox 类公开了更便于与使用 Windows 窗体的用户进行交互的方法和属性。有关更多信息，请参见 System.Windows.Forms。

编译和执行

可使用命令行编译器编译"Hello World!"程序。

从命令行编译并运行程序

1. 使用任何文本编辑器创建源文件并使用一个文件名（如 Hello.js）来保存它。
2. 若要启动该编译器，请在命令提示处键入以下内容：

   ```
   jsc Hello.js
   ```

   注意

应当从 Visual Studio 命令提示处编译此程序。有关更多信息，请参见 如何：从命令行编译 JScript 代码。

若程序不包含编译错误，则编译器会创建一个 Hello.exe 文件。

3. 若要运行程序，请在命令提示处键入以下内容：

```
Hello
```

请参见

其他资源

编写、编译、调试 JScript 代码

使用 JScript 显示信息

JScript 参考

升级先前的 JScript 版本中创建的应用程序

大多数现有的 JScript 代码都可以很好地使用 JScript 8.0 中包含的增强功能，因为对于以前的版本，JScript 8.0 几乎是完全向后兼容的。JScript 8.0 的新功能开创了新的天地。

默认情况下，JScript 8.0 程序是在 *快速模式* 中编译的。由于快速模式对所允许的代码类型有一些限制，因此程序可能会更有效并执行得更快。但是，以前版本中可用的一些功能在快速模式下不可用。这些功能大部分与多线程应用程序不兼容，并会使代码效率低下。对于用命令行编译器编译的程序，可以关闭快速模式，而利用完全的向后兼容性。注意，用这种方法编译的代码运行得较慢，容错性也较差。在 ASP.NET 应用程序中不能关闭快速模式，因为会出现稳定性问题。有关更多信息，请参见 /fast。

快速模式

在快速模式中，会触发以下 JScript 行为：

- 必须声明所有变量。
- 函数变为常数。
- 内部对象不能有 expando 属性。
- 不能列出或更改内部对象的属性。
- arguments 对象不可用。
- 不能给只读变量、字段或方法赋值。
- eval 方法不能在封闭范围内定义标识符。
- eval 方法在受限安全上下文中执行脚本。

必须声明所有变量

先前的 JScript 版本不要求显式声明变量。尽管此功能使程序员节省了击键次数，但它也使跟踪错误变得困难。例如，您可能赋值给拼写错误的变量名，这将既不生成错误也不会返回需要的结果。而且，未声明的变量具有全局范围，还会引起其他混淆。

快速模式要求显示声明变量。这有助于避免出现前面提到的各种错误，并可产生运行得更快的代码。

JScript .NET 还支持经过类型批注的变量。这样就将每个变量与特定的数据类型绑定在一起，该变量只能存储那种类型的数据。尽管类型批注不是必须的，但使用它有助于避免与在变量中意外存储错误数据相关的那些错误，并可提高程序执行的速度。

有关更多信息，请参见 JScript 变量和常数。

函数变为常数

在以前的 JScript 版本中，用 **function** 语句声明的函数与保存 **Function** 对象的变量被同等对待。特别是，任何函数标识符都可用作变量，来存储任何类型的数据。

在快速模式中，函数变成了常数。因此，不能为函数赋新值或重新定义函数。这样可避免意外更改函数的意义。

如果您的脚本需要使函数发生更改，则可以显式使用某个变量以保存 **Function** 对象的实例。但是请注意，**Function** 对象运缓慢。有关更多信息，请参见 Function 对象。

内部对象不能有 expando 属性

在先前的 JScript 版本中，可以为内部对象添加 expando 属性。例如，此行为可用于为 **String** 对象添加方法以剪裁字符串前面的空格。

在快速模式中，这是不允许的。如果您的脚本使用了此功能，则必须修改脚本。可以在全局范围内定义函数，而不是将那些函数作为方法附加到对象上。然后，重写脚本中的每个实例（在该脚本中 expando 方法是从对象中调用的），以便将对象传递给适当的函数。

此规则的一个重要例外是 **Global** 对象，它仍然可具有 expando 属性。全局范围内的所有修饰符实际上都是 **Global** 对象的属性。显然，**Global** 对象必须能动态扩展以支持添加新的全局变量。

不能列出或更改内部对象的属性

在以前的 JScript 版本中，可以对内部对象的预定义属性进行删除、枚举或写入。例如，此种行为可用于更改 **Date** 对象的默认 **toString** 方法。

在快速模式中，这是不允许的。由于内部对象不能具有 expando 属性，因此不再需要此功能，而每个对象的属性则列在参考部分。有关更多信息，请参见对象。

arguments 对象不可用

先前的 JScript 版本在函数定义中提供了一个 **arguments** 对象，该对象允许函数接受任意个参数。该参数对象还可以引用当前函数和调用函数。

在快速模式中，**arguments** 对象不可用。但是，JScript 8.0 允许进行函数声明以便在函数参数列表中指定一个*参数数组*。这就允许函数接受任意多个参数，从而取代了 **arguments** 对象的部分功能。有关更多信息，请参见 function 语句。

在快速模式中没有办法直接访问和引用当前函数或调用函数。

不能给只读变量、字段或方法赋值

在先前的 JScript 版本中，语句似乎可以为只读标识符赋值。这种赋值将无提示地失败，而发现赋值失败的唯一方法是测试值是否实际发生了更改。为只读标识符赋值通常是某种差错引起的，因为它不会有任何效果。

在快速模式中，如果试图为只读标识符赋值，将生成编译时错误。要么可以移除该赋值，要么可以尝试为非只读的标识符赋值。

如果关闭快速模式，为只读标识符赋值将在运行时无提示地失败，但是会生成一个编译时警告。

eval 方法不能在封闭范围内定义标识符

在先前的 JScript 版本中，函数和变量可以通过调用 **eval** 方法在本地或全局范围内定义。

在快速模式中，函数和变量可以在对 **eval** 方法的调用中定义，但只能从这个特定的调用中对它们进行访问。一旦完成 **eval** 后，在 **eval** 内定义的函数和变量就不能再行访问。在 **eval** 内计算所得的结果可以赋给当前范围内可访问的任何变量。对 **eval** 方法的调用很慢，应考虑重写包含这些调用的代码。

当关闭快速模式时，可恢复 **eval** 方法的先前行为。

eval 方法在受限安全上下文中执行脚本

在以前版本的 JScript 中，传递至 **eval** 方法的代码将与调用代码在同一安全上下文中运行。

为了保护用户，传递至 **eval** 方法的代码会在受限安全上下文中执行，除非将字符串"unsafe"作为第二个参数传递。受限安全上下文禁止访问系统资源，如文件系统、网络或用户界面。如果代码试图访问这些资源，则会产生安全异常。

当 **eval** 的第二个参数为字符串"unsafe"时，传递给 **eval** 方法的代码在调用代码所在的安全上下文中执行。这样，可以还原 **eval** 方法以前的行为。

安全注意
以非安全模式使用 **eval** 只能执行从已知源获得的代码字符串。

请参见

参考

/fast

概念

JScript 8.0 for JScript Programmers 简介

其他资源

JScript 入门

在以前版本的公共语言运行库上运行 JScript 应用程序

除非另外指定，否则 JScript 应用程序会在编译器用于生成该应用程序的公共语言运行库版本上运行。但是，使用某个版本的运行库生成的 .exe 或 ASP.NET Web 应用程序可以在任何版本的运行库上运行。

适用其它运行库版本

要实现这一点，.exe 应用程序需要一个包含运行库版本信息的 app.config 文件（带有 supportedRuntime 标记）。其他 Visual Studio 语言提供了集成开发环境 (IDE) 支持，可用于通过使用它们的项目的属性页对话框修改 app.config 文件。例如，修改 Visual C# Windows 应用程序的 **SupportedRuntimes** 属性并在 JScript 应用程序中使用该更新的 app.config 文件。

在运行时，app.config 文件的名称必须是 *filename.ext*.config（其中，*filename.ext* 是启动该应用程序的可执行文件的名称），并且该文件与可执行文件必须位于同一目录中。例如，如果应用程序名为 TestApp.exe，则会将 app.config 文件命名为 TestApp.exe.config。

如果指定了多个运行库版本，并且运行应用程序所在的计算机上安装了多个运行库版本，那么该应用程序使用配置文件中指定的第一个与系统上安装的运行库匹配的版本。

有关更多信息，请参见 如何：使用应用程序配置文件指定 .NET Framework 的版本。

由于 JScriptASP.NET网页是单文件 Web 窗体页，因此不会将它们预编译为依赖于与编译器关联的 .NET Framework 程序集的 .dll。因而，这些页是在运行时编译的，并且 web.config 文件中不需要运行库版本信息。

请参见

参考

SupportedRuntimes 属性

概念

ASP.NET 网页代码模型

为 JScript 程序员提供的附加资源

以下站点和新闻组可为常见的 JScript 编程问题提供答案。

Web 上的 Microsoft 资源

Microsoft 产品支持服务 (http://support.microsoft.com/default.aspx?ln=zh-cn)

提供对知识库文章、下载和更新、支持 Web 广播以及其他服务的访问权。

MSDN 新闻组 (http://www.microsoft.com/china/community)

提供了一个交流园地，您可以通过它与世界各地的专家进行交流。

Microsoft ASP.NET (http://www.asp.net/)

提供与在 JScript 中进行 Web 开发有关的文章、演示、工具预览以及其他信息。

Microsoft Windows 脚本 (http://www.microsoft.com/china/scripting)

包含文章、示例以及 JScript 5.6 开发人员感兴趣的其他信息。

新闻组中的 Microsoft 资源

microsoft.public.dotnet.languages.jscript

为 JScript 的相关问题和一般讨论提供论坛。

microsoft.public.dotnet.scripting

为使用 .NET Framework 的脚本的相关问题和一般讨论提供论坛。

microsoft.public.vsnet.documentation

为 JScript 文档的相关问题和一般讨论提供论坛。

microsoft.public.scripting.jscript

为 JScript 5.*x* 的相关问题和一般讨论提供论坛。

microsoft.public.scripting.wsh

为在 Microsoft Windows 脚本宿主 (WSH) 中使用 JScript 5.*x* 的相关问题和一般讨论提供论坛。

Web 上的其它资源

4GuysFromRolla.com (http://www.4guysfromrolla.com/)

提供与在 JScript 中进行 Web 开发有关的文章、演示、工具预览以及其他信息。

DevX (http://www.devx.com/)

提供用于在 JScript 中进行开发的文章、演示、工具预览以及其他信息。

请参见

其他资源

JScript 入门

编写、编译、调试 JScript 代码

Visual Studio 集成开发环境 (IDE) 是所有语言的通用开发环境，它提供了帮助您编写可靠代码的工具和验证架构。IDE 还提供调试功能，用于消除不一致和解决代码错误。

本节内容

如何：从命令行编译 JScript 代码

描述如何使用命令行编译器生成编译的 JScript 程序。

使用 Visual Studio 编写 JScript 代码

解释如何使用 Visual Studio 集成开发环境 (IDE) 来编写和编辑 JScript 代码。

条件编译

描述如何以及何时使用条件编译，条件编译允许在编译时包含多种代码片段以进行调试，便于使用新的 JScript 功能，同时又保留了向后兼容性。

检测浏览器功能

描述如何通过使用脚本引擎功能和条件编译来确定 Web 浏览器引擎支持哪些版本的 JScript。

复制、传递和比较数据

阐释以传址方式和传值方式存储数据的区别，以及这两个替换选项与数据类型的关系。

JScript 如何重载方法

介绍如何使用相同的名称与不同的签名来重载方法。

如何：在 JScript 中处理事件

介绍如何通过将事件处理程序方法与现有事件链接来处理事件。

使用 Visual Studio 调试 JScript

列出对命令行程序或 ASP.NET 程序启用调试的过程。

使用公共语言运行库调试器调试 JScript

列出对命令行程序或 ASP.NET 程序启用公共语言运行库编译调试器的过程。

脚本疑难解答

提供一些提示，以避免在某些具体方面（如语法、脚本解释顺序、自动类型强制等等）出现常见脚本错误。

相关章节

从命令行生成

解释如何从命令行调用编译器，并提供返回特定结果的命令行语法的示例。

调试和分析应用程序

介绍调试 .NET Framework 应用程序的过程，以及未安装 Visual Studio 时如何使用 .NET Framework 来优化应用程序。

如何：从命令行编译 JScript 代码

若要生成可执行的 JScript 程序，必须使用命令行编译器 jsc.exe。该编译器可通过几种方法启动。

如果安装了 Visual Studio，可使用 Visual Studio 命令提示从计算机上的任何目录访问编译器。Visual Studio 命令提示位于 Microsoft Visual Studio 程序组的 Visual Studio 工具程序文件夹中。

另一种方法是从 Windows 命令提示符启动编译器。如果未安装 Visual Studio，这是典型的做法。

Windows 命令提示符

若要从 Windows 命令提示符启动编译器，则必须从该应用程序所在的目录运行它，或者在命令提示符下键入可执行文件的完全限定的路径。若要改写这种默认行为，必须修改 PATH 环境变量，这样只需键入编译器的名称即可从任何目录运行编译器。

修改 PATH 环境变量

1. 使用 Windows"搜索"功能在本地驱动器上查找 jsc.exe。jsc.exe 所在目录的确切名称取决于 Windows 目录的名称和位置以及安装的 .NET Framework 的版本。如果安装了多个 .NET Framework 版本，则必须确定使用哪个版本（通常使用最新的版本）。

   例如，编译器可能位于 `C:\WINNT\Microsoft.NET\Framework\v1.0.2914`。

2. 在桌面 (Windows 2000) 上右击"我的电脑"图标，然后从快捷菜单选择"属性"。

3. 选择"高级"选项卡，然后单击"环境变量"按钮。

4. 在"系统变量"窗格中，从列表中选择"Path（路径）"，然后单击"编辑"。

5. 在"编辑系统变量"对话框中，将光标移到"变量值"字段中的字符串的末尾，键入一个分号 (;)，然后键入在第 1 步中找到的完整目录名称。

   如果仍然使用第 1 步中的示例，则应键入：

   ```
   ;C:\WINNT\Microsoft.NET\Framework\v1.0.2914
   ```

6. 单击"确定"确认所做的编辑，然后关闭各对话框。

更改 PATH 环境变量之后，可以在 Windows 命令提示符下从计算机上的任何目录运行 JScript 编译器。

使用编译器

命令行编译器有一些内置的帮助。通过使用 **/help** 或 **/?** 命令行选项，或者通过在不提供任何选项的情况下使用编译器，就会显示帮助屏幕。例如：

```
jsc /help
```

有两种使用 JScript 的方式。可以编写从命令行编译的程序，也可以编写在 ASP.NET 中运行的程序。

使用 jsc 进行编译

- 在命令提示符下键入 jsc file.js

  此命令编译名为 file.js 的程序，以生成名为 file.exe 的可执行文件。

使用 jsc 生成 .dll 文件

- 在命令提示符下键入 jsc /target:library file.js

  此命令使用 **/target:library** 选项编译名为 file.js 的程序，以生成名为 file.dll 的库文件。

使用 jsc 生成具有不同名称的可执行文件

- 在命令提示符下键入 jsc /out:newname.exe file.js

  此命令使用 **/out:** 选项编译名为 file.js 的程序，以生成名为 newname.exe 的可执行文件。

使用 jsc 编译并给出调试信息

- 在命令提示符下键入 jsc /debug file.js

  此命令使用 **/debug** 选项编译名为 file.js 的程序，以生成名为 file.exe 的可执行文件以及名为 file.pdb 的文件（包含调试信息）。

JScript 命令行编译器还有很多其它命令行选项。有关更多信息，请参见 JScript 编译器选项。

请参见

其他资源

编写、编译、调试 JScript 代码

JScript 编译器选项

条件编译

使用 Visual Studio 编写 JScript 代码

Visual Studio 集成开发环境 (IDE) 为 JScript 开发提供了一些工具，其中一些工具与集成开发环境为其他语言提供的工具相同。“新建文件”对话框包含几个模板，这些模板为各种 JScript 文件提供框架。

IDE 过程

使用 Visual Studio 编写新的 JScript 代码

1. 启动 Microsoft Visual Studio。
2. 从“文件”菜单中，单击“新建文件”。
3. 在“类别”对话框中，单击“脚本”文件夹。
4. 在“模板”对话框中，选择“JScript 文件”或“JScript .NET Web 窗体”，然后单击“打开”。

使用 Visual Studio 编辑 JScript 代码

1. 启动 Microsoft Visual Studio。
2. 从“文件”菜单中单击“打开”，再单击“文件”。
3. 在“打开文件”对话框中，浏览到源文件，选择该文件并单击“打开”。

关键字会以不同的颜色显示，具体情况取决于每个文件中所使用的语言。当您编辑代码时，在“动态帮助”窗口中就会出现上下文相关的帮助。

在 Visual Studio 中保存 JScript 代码

- 从“文件”菜单中单击“保存<文件名>”或“另存<文件名>为”。

注意
您不能在 Visual Studio IDE 中编译 JScript 代码。必须从命令行或由 ASP.NET 页执行该步骤。

请参见

概念

从命令行生成
使用 Visual Studio 调试 JScript
使用公共语言运行库调试器调试 JScript

其他资源

编写、编译、调试 JScript 代码
Visual Studio 集成开发环境
使用 IntelliSense

条件编译

条件编译使 JScript 能够使用新的语言功能，同时又保留与不支持这些功能的旧版本之间的兼容性。条件编译的一些典型用途包括在 JScript 中使用新功能、将调试支持嵌入到一个脚本中以及跟踪代码执行。

本节内容

条件编译指令

列出控制如何编译脚本的指令以及指向特定信息的链接，这些信息解释每个指令的正确语法。

条件编译语句

列出并讨论根据条件编译变量的值控制脚本编译的语句。

条件编译变量

列出条件编译可用的预定义变量。

相关章节

编写、编译、调试 JScript 代码

指向特定主题的链接，这些主题解释如何使用集成开发环境 (IDE) 编写 JScript 代码。

如何：从命令行编译 JScript 代码

描述如何使用命令行编译器生成编译的 JScript 程序。

检测浏览器功能

描述如何通过使用脚本引擎功能和条件编译确定 Web 浏览器引擎支持 JScript 的哪些版本。

JScript 编译器选项

指向特定信息的链接，这些信息列出 JScript 命令行编译器可用的编译器选项。

条件编译指令

当编译代码进行调试时，可使用以下指令修改默认行为。

指令

指令	说明
@debug	开启或关闭调试符号的显示。
@position	在错误信息中给出有意义的位置信息。

这些指令是为设计代码的开发人员提供的，这些代码由宿主环境（如 ASP.NET）自动加入到 JScript 程序中。通常，编写在该环境中运行的脚本的作者对加入的这些代码并不感兴趣。当这些作者调试他们的代码时，只想查看他们所编写的程序部分，而不想查看开发工具编写的代码和附加部分。

这些条件编译指令可以通过关闭调试符号的显示并重置行位置来“隐藏”代码。这就使得脚本作者在调试脚本时只看到他们所编写的代码。

请参见

参考

/debug

概念

条件编译变量

条件编译语句

其他资源

条件编译

条件编译语句

JScript 可以使用以下语句根据条件编译变量的值控制脚本的编译。既可以使用 JScript 提供的变量，也可以使用 **@set** 指令或 **/define** 命令行选项定义自己的变量。

语句

语句	说明
@cc_on	激活条件编译支持。
@if	根据表达式的值，有条件地执行一组语句。
@set	创建使用条件编译语句的变量。

@cc_on、**@if** 或 **@set** 语句激活条件编译。条件编译的一些典型用途包括在 JScript 中使用新功能、将调试支持嵌入到一个脚本中以及跟踪代码执行。

当编写由 Web 浏览器运行的脚本时，总是将条件编译代码放在注释中。因此，不支持条件编译的宿主可以忽略该代码。这是一个示例。

```
/*@cc_on @*/
/*@if (@_jscript_version >= 5)
document.write("JScript Version 5.0 or better.<BR>");
@else @*/
document.write("You need a more recent script engine.<BR>");
/*@end @*/
```

此示例使用特殊的注释分隔符，仅当 **@cc_on** 语句激活条件编译之后才使用这些分隔符。不支持条件编译的脚本引擎显示一则消息，建议需要使用新的脚本引擎，而不会产生错误。支持条件编译的引擎根据引擎的版本编译第一个或第二个 `document.write`。请注意，7.x 版表示 JScript .NET。有关更多信息，请参见检测浏览器功能。

条件编译对于服务器端脚本和命令行程序也很有用。在这些应用程序中，可使用条件编译将其他函数编译到一个程序中，便于在调试模式下进行分析。

请参见

参考

/define

概念

条件编译变量

条件编译指令

检测浏览器功能

其他资源

条件编译

条件编译变量

以下预定义变量可用于条件编译。

变量

变量	说明
@_win32	如果在 Win32 系统上运行，并且没有指定 /platform 选项或指定了 /platform:anycpu 选项，则为 **true**；否则为 **NaN**。
@_win16	如果在 Win16 系统上运行，则为 **true**；否则为 **NaN**。
@_mac	如果在 Apple Macintosh 系统上运行，则为 **true**；否则为 **NaN**。
@_alpha	如果在 DEC Alpha 处理器上运行，则为 **true**；否则为 **NaN**。
@_x86	如果在 Intel 处理器上运行，并且没有指定 /platform 选项或指定了 /platform:anycpu 选项，则为 **true**；否则为 **NaN**。
@_mc680x0	如果在 Motorola 680x0 处理器上运行，则为 **true**；否则为 **NaN**。
@_PowerPC	如果在 Motorola PowerPC 处理器上运行，则为 **true**；否则为 **NaN**。
@_jscript	始终为 **true**。
@_jscript_build	JScript 脚本引擎的内部版本号。
@_jscript_version	以 major.minor 格式表示 JScript 版本号的数字。
@_debug	如果在调试模式下编译则为 **true**；否则为 **false**。
@_fast	如果在快速模式下编译则为 **true**；否则为 **false**。

注意

JScript .NET 报告的版本号为 7.*x*。JScript 8.0 报告的版本号为 8.*x*。

在使用条件编译变量之前，必须先打开条件编译。**@cc_on** 语句可打开条件编译。条件编译变量通常用于针对 Web 浏览器编写的脚本中。在为 ASP 或 ASP.NET 页或命令行程序编写的脚本中很少使用条件编译变量，这是因为可以使用其他方法确定编译器的兼容性。

当编写用于网页的脚本时，始终将条件编译代码放在注释中。这样，不支持条件编译的宿主就可以忽略该代码。这是一个示例。

```
/*@cc_on
  document.write("JScript version: " + @_jscript_version + ".<BR>");
  @if (@_win32)
     document.write("Running on 32-bit Windows.<BR>");
  @elif (@_win16)
     document.write("Running on 16-bit Windows.<BR>");
  @else
     document.write("Running on a different platform.<BR>");
  @end
@*/
```

条件编译变量可用于确定解释脚本的引擎的版本信息。这使脚本可以利用最新 JScript 版本中的功能，同时又保留向后兼容性。有关更多信息，请参见检测浏览器功能。

请参见

概念

条件编译指令

条件编译语句

检测浏览器功能

其他资源

条件编译

检测浏览器功能

尽管浏览器支持大多数 JScript 功能，但只有在服务器端才支持面向 .NET Framework、基于类的对象、数据类型、枚举、条件编译指令和 **const** 语句的那些新功能。因此，您应该在服务器端脚本中以独占方式使用这些功能。有关更多信息，请参见 JScript 版本信息。

JScript 脚本可检测对其进行解释或编译的引擎的能力。如果为服务器端应用程序（将运行在 ASP 或 ASP.NET 中）或命令行程序编写代码，则不必检测，因为可以很方便地发现所支持的 JScript 版本和相应的代码。然而，当在浏览器中运行客户端脚本时，为了确保脚本与浏览器中的 JScript 引擎相兼容，该检测就是很重要的。

有两种方法可以检查 JScript 的兼容性：使用脚本引擎函数或使用条件编译。同时使用这两种方法有一些优点。

脚本引擎函数

脚本引擎函数 (**ScriptEngine**、**ScriptEngineBuildVersion**、**ScriptEngineMajorVersion**、**ScriptEngineMinorVersion**) 返回有关脚本引擎当前版本的信息。有关更多信息，请参见 函数 (JScript)。

为了得到最大的兼容性，在检查受支持的 JScript 版本的页中，应该只使用在 JScript 版本 1 中能够找到的那些功能。如果某个引擎支持高于 1.0 的 JScript 版本，则可以重定向到包含高级功能的另一页。这就意味着，对应于您想要支持的每个 JScript 版本，您都必须有每个网页的一个单独版本。在大多数情况下，最有效的解决方案就是只有两个页，一个用于特定版本的 JScript，另一个不用 JScript 工作。

注意

使用高级功能的 JScript 代码必须放在某个单独的页中，该页不能由引擎不兼容的浏览器运行。这是强制性的，因为浏览器的脚本引擎将解释页中包含的所有 JScript 代码。对于较早的引擎，不能使用 **if...else** 语句在使用了最新 JScript 版本的代码块和 JScript 1 版代码块之间进行切换。

下面的示例阐释了脚本引擎函数的用法。由于这些函数是在 JScript 2.0 版中引入的，因此在试图使用它们之前必须首先确定引擎是否支持这些函数。如果引擎只支持 JScript 1.0 版或者不能识别 JScript，**typeof** 运算符将为每个函数名称返回字符串“undefined”。

```
if("undefined" == typeof ScriptEngine) {
   // This code is run if the script engine does not support
   // the script engine functions.
   var version = 1;
} else {
   var version = ScriptEngineMajorVersion();
}
// Display the version of the script engine.
alert("Engine supports JScript version " + version);
// Use the version information to choose a page.
if(version >= 5) {
   // Send engines compatible with JScript 5.0 and better to one page.
   var newPage = "webpageV5.htm";
} else {
   // Send engines that do not interpret JScript 5.0 to another page.
   var newPage = "webpagePre5.htm";
}
location.replace(newPage);
```

条件编译

条件编译的变量和语句可以将来自不支持条件编译的引擎的 JScript 代码隐藏起来。如果想在网页中直接包含少量替换代码，这种方法就是很有用的。

注意

不要在条件编译块中使用多行注释，因为不支持条件编译的引擎可能会对其作出错误解释。

```
<script>
/*@cc_on
```

```
@if(@_jscript_version >= 5 )
// Can use JScript Version 5 features such as the for...in statement.
// Initialize an object with an object literal.
var obj = {"a" : "Athens" , "b" : "Belgrade", "c" : "Cairo"};
var key;
// Iterate the properties.
for (key in obj) {
   document.write("The "+key+" property has value "+obj[key]+".<BR>");
}
@else
@*/
alert("Engine cannot interpret JScript Version 5 code.");
//@end
</script>
```

如果条件 **@if** 块包含大量代码，利用上述方法来使用脚本引擎函数可能更容易一些。

请参见

概念

JScript 版本信息

其他资源

编写、编译、调试 JScript 代码

函数 (JScript)

条件编译

复制、传递和比较数据

JScript 复制、传递和比较数据的方法取决于数据的存储方式，而数据的存储方式又取决于数据的类型。JScript 以传值方式或传址方式存储数据。

传值方式和传址方式

JScript 以"*传值方式*"复制、传递和比较数字和布尔值(**true** 和 **false**)。此过程在计算机内存中分配一个空间，然后将原始项的值复制到该处。对原始项的更改不会影响该副本(反之亦然)，这是因为两者是单独的实体。如果两个数字或布尔值具有相同的值，则认为它们相等。

JScript 以"*传址方式*"复制、传递和比较对象、数组和函数。实质上，此过程创建一个对原始项的引用，并将该引用当作副本一样使用。对原始项的更改会同时更改原始项和副本(反之亦然)。实际上只有一个实体；副本只是数据的另一个引用。

若要成功地以传址方式比较两个变量，这两个变量必须恰好引用同一个实体。例如，两个不同的 **Array** 对象的比较结果始终是不相等(即使它们包含相同的元素)。若要使比较成功，其中一个变量必须是对另一个变量的引用。若要检查两个数组是否包含相同的元素，请比较 **toString()** 方法的结果。

最后，JScript 以传址方式复制和传递字符串。字符串是否为对象决定了字符串的比较方式。对两个 **String** 对象(用 **new String**("something") 创建)以引用方式进行比较。如果其中一个(或两个)字符串是常值或基元字符串值，则以传值方式对它们进行比较。有关更多信息，请参见 JScript 赋值与相等。

注意

ASCII 和 ANSI 字符集均按将大写字母排在小写字母前面的序列顺序进行构造。例如，"Zoo"相比之下"小于""aardvark"。如果要执行不区分大小写的匹配，则可以对两个字符串调用 **toUpperCase()** 或 **toLowerCase()**。

函数参数

当 JScript 以传值方式向函数传递参数时，它就会制作一个单独的参数副本(仅存在于函数内部)。即使对象和数组以传址方式来传递，如果函数中的新值直接将其改写，则新值将不会在函数之外得到反映。只有对对象属性或数组元素的更改才会在函数外可见。

例如，以下程序有两个函数。第一个函数改写输入参数，它可进一步防止参数更改影响原始输入参数。第二个函数更改对象的属性而不改写该对象。

```
function clobber(param) {
   // Overwrite the parameter; this will not be seen in the calling code
   param = new Object();
   param.message = "This will not work.";
}

function update(param) {
    // Modify the property of the object; this will be seen in the calling code.
    param.message = "I was changed.";
}

// Create an object, and give it a property.
var obj = new Object();
obj.message = "This is the original.";

// Call clobber, and print obj.message.
clobber(obj);
print(obj.message);

// Call update, and print obj.message.
update(obj);
print(obj.message);
```

该代码的输出为：

```
This is the original.
I was changed.
```

数据比较

JScript 可以按传值方式或传址方式比较数据。若要以传值方式执行测试，JScript 比较两个截然不同的项以确定它们是否相等。通常，这种比较是逐字节进行的。当按传址方式测试时，它检查两个项是否引用同一个项。如果是，它们的比较结果则为相等；如果不是，即使它们包含完全相同的值（逐字节比较），它们的比较结果仍是不相等。

根据字符串是对象与否，可以按传值方式或传址方式比较字符串。如果两个字符串均为 **String** 对象，则以传址方式比较字符串；否则以传值方式比较。这样，如果两个字符串是单独创建的但具有相同的内容，则两者的比较结果为相等。若要比较两个 **String** 对象的值，先使用 **toString** 或 **valueOf** 方法将对象转换为非对象字符串，然后比较生成的字符串。有关更多信息，请参见 JScript 赋值与相等。

请参见

概念

JScript 赋值与相等

数据类型摘要

其他资源

JScript 语言教程

JScript 函数

JScript 如何重载方法

当类中两个或更多 JScript 成员(函数或属性)具有相同名称但不同签名时，被称为"重载"函数(属性)。函数的"签名"基于它接受的参数的数量、类型和顺序。如果两个函数以同样顺序接受相同类型相同数量的参数，则这两个函数具有相同的签名。以不同顺序接受相同类型参数的函数，或具有不同数量参数或者不同类型参数的函数具有不同签名。(注意，参数的名称不影响签名)。静态函数也可参与重载，但就返回类型而言，方法的静态状态不影响其签名。因此，一个静态方法如果具有与一个实例方法相同的名称，则必须具有一个不同的参数列表。

处理逻辑

调用重载的函数时，将调用其参数最密切匹配传递参数的重载函数，具体取决于传递给函数的实际参数类型。如果参数类型确切匹配特定重载，则调用该重载。如果参数类型不确切匹配任何重载，则通过一个排除过程来决定调用哪个重载。排除过程的运作是基于实际类型转换为可用重载中的类型的容易程度。有关更多信息，请参见 JScript 中的强制。在此示例中，类 MethodOverload 具有三个名为 Greetings 的重载方法。第一个重载不使用参数，第二个重载使用一个 **String** 类型的参数，第三个重载使用两个参数：**String** 类型和 **int** 类型。

```
var methodOverload = new MethodOverload();
methodOverload.Greetings();
methodOverload.Greetings("Mr. Brown");
methodOverload.Greetings(97, "Mr. Brown");

class MethodOverload
{
   function Greetings()
   {
     print("Hello, and welcome!");
   }
   function Greetings(name : String)
   {
     print("Hello, " + name + "!");
   }
   function Greetings(ticket : int, name : String)
   {
     print("Hello, " + name + "! Your ticket number is " + ticket + ".");
   }
}
```

该程序的输出为：

```
Hello, and welcome!
Hello, Mr.Brown!
Hello, Mr.Brown! Your ticket number is 97.
```

请参见

其他资源

JScript 语言教程

JScript 数据类型

如何：在 JScript 中处理事件

事件是用户通常执行的操作，例如单击鼠标按钮、按键、更改数据或者打开文档或窗体。此外，程序代码也可执行操作。事件处理程序是绑定到事件的方法。当引发事件时，执行事件处理程序内的代码。JScript .NET 事件处理程序可连接到任何类型的 .NET 应用程序（ASP.NET、Windows 窗体、控制台等）。但是，不能在 JScript 中声明新事件。JScript 代码只能使用已存在的事件。

为按钮控件的单击事件创建事件处理程序

- 添加下列代码：

```
// Events.js
import System;
import System.Windows.Forms;

class EventTestForm extends Form
{
  var btn : Button;

  function EventTestForm()
  {
    btn = new Button;
    btn.Text = "Fire Event";
    Controls.Add(btn);
    // Connect the function to the event.
    btn.add_Click(ButtonEventHandler1);
    btn.add_Click(ButtonEventHandler2);
  }

  // Add an event handler to respond to the Click event raised
  // by the Button control.
  function ButtonEventHandler1(sender, e : EventArgs)
  {
    MessageBox.Show("Event is Fired!");
  }

  function ButtonEventHandler2(sender, e : EventArgs)
  {
    MessageBox.Show("Another Event is Fired!");
  }
}

Application.Run(new EventTestForm);
```

注意

每个事件处理程序提供两个参数。第一个参数 sender 提供对引发事件的对象的引用。上面示例中的第二个参数 e 传递针对要处理的事件的对象。通过引用对象的属性（有时引用其方法）可获得一些信息，如鼠标事件中鼠标的位置或拖放事件中传输的数据。

编译代码

1. 使用 Visual Studio 提供的命令行编译器 jsc.exe。
2. 键入以下命令行指令创建名为 Events.exe 的 Windows 可执行 (EXE) 程序：

   jsc /target:winexe Events.js

注意

引发单个事件可通过链接尽可能多事件所需的函数来调用多个事件处理程序：

```
btn.add_Click(ButtonEventHandler1);
btn.add_Click(ButtonEventHandler2);
. . .
```

请参见

任务

如何：从命令行编译 JScript 代码

其他资源

编写、编译、调试 JScript 代码

使用 Visual Studio 调试 JScript

根据设计，某些 JScript 程序是用于从命令行运行的，而其他一些 JScript 程序则在 ASP.NET 页中运行。程序类型影响调试方法。

过程

为命令行程序设置调试

1. 使用 **/debug** 标志编译要调试的程序。有关更多信息，请参见 /debug。
2. 启动 Microsoft Visual Studio。
3. 从“文件”菜单中单击“打开”，然后单击“项目”。
4. 在“打开项目”对话框中，浏览到已编译的程序(扩展名为 .exe 的文件)，选择该程序，然后单击“打开”。
5. 从“文件”菜单中单击“打开”，再单击“文件”。
6. 在“打开文件”对话框中，浏览到源代码(扩展名为 .js 的文件)，选择该代码，然后单击“打开”。
7. 从“文件”菜单中单击“全部保存”。
8. 选择一个名称和位置来保存新项目。

完成此设置后，可以继续执行“使用 Visual Studio 进行调试”节中的操作。

为 ASP.NET 程序设置调试

1. 启动 Microsoft Visual Studio。
2. 打开要调试的 ASP.NET 文件。
3. 在 **@ page** 指令中，将调试标志设置为 true。例如：

```
<%@page Language=jscript debug=true %>
```

4. 在浏览器中打开该页以编译该页。
5. 从 Visual Studio“工具”菜单中单击“调试进程”。
6. 在“进程”对话框中，选择“显示系统进程”和“显示所有会话中的进程”选项。
7. 在“进程”对话框的“可用进程”窗格中，选择运行该 Web 应用程序的 ASP.NET 辅助进程，然后单击“附加”。

   默认情况下，对于 IIS 5.*x*，辅助进程是 aspnet_wp.exe(在 Windows 2000 和 Windows XP 上)，对于 IIS 6.0，辅助进程是 w3wp.exe(在 Windows Server 2003 上)。
8. 在“附加到进程”对话框中，选择“Common Language Runtime”，然后单击“确定”。
9. 在“进程”对话框中，单击“关闭”。

完成此设置后，可以继续执行“使用 Visual Studio 进行调试”节中的操作。

使用 Visual Studio 进行调试

1. 在 Visual Studio IDE 中，按照上面所述的两种设置过程中的任一种，打开要调试的文件。
2. 将光标移到文件中要设置断点的位置，然后按 F9 键。
3. 重复上面的步骤以添加更多断点。
4. 从“调试”菜单中，单击“启动”。

   程序在遇到断点或发生运行时错误时将停止运行。
5. 此时，将打开几个窗口，以便执行进一步的调试任务。有关更多信息，请参见 调试器指南。
6. 若要停止调试但保持程序运行，请从“调试”菜单上，选择“全部分离”。

   否则，停止调试后，程序也将终止。

备注

当调试从命令行编译的程序时，Visual Studio 在您每次开始调试时都会重新读取编译的程序。因此，您可以修改 JScript 代码并（在重新编译代码后）检查那些更改的效果。

请参见

任务

使用 Visual Studio 编写 JScript 代码

概念

从命令行生成

从命令行生成

使用公共语言运行库调试器调试 JScript

其他资源

调试器指南

调试 Web 应用程序

使用公共语言运行库调试器调试 JScript

根据设计，某些 JScript 程序是用于从命令行运行的，而其他一些 JScript 程序则在 ASP.NET 页中运行。程序类型影响调试方法。

公共语言运行库调试器 dbgclr.exe 位于 .NET Framework 安装的 GuiDebug 目录中。

若要使用 dbgclr.exe，必须使用路径名限定程序名，或者将该路径添加到搜索路径中。

过程

为命令行程序设置调试

1. 在任何编辑器中编写您的程序然后将它另存为文本。
2. 使用 /debug 标志编译程序。有关更多信息，请参见 /debug。
3. 启动公共语言运行库调试器 dbgclr。
4. 从 dgbclr 的“文件”菜单中单击“打开”，然后单击“文件”。
5. 在“打开文件”对话框中，打开要调试的源文件（扩展名为 .js 的文件）。
6. 从“调试”菜单中单击“要调试的程序”。
7. 在“要调试的程序”对话框中，单击“程序”窗格旁边的省略号 (...)。
8. 在“查找要调试的程序”窗口中，浏览到已编译的程序（扩展名为 .exe 的文件），选择该程序，然后单击“打开”。
9. 在“要调试的程序”对话框中，单击“确定”。

完成此设置后，可以继续执行“使用公共语言运行库调试器进行调试”节中的操作。

为 ASP.NET 程序设置调试

1. 在任何编辑器中编写您的程序然后将它另存为文本。
2. 为 ASP.NET 编写 HTML 包装。通过在代码中包括下面的一行确保指定要调试 JScript 代码：

   ```
   <%@page Language=jscript debug=true %>
   ```

3. 在浏览器中打开该页以编译该页。
4. 启动公共语言运行库调试器 dbgclr。
5. 从 dgbclr 的“工具”菜单中单击“调试进程”。
6. 在“进程”窗口中，选择“显示系统进程”和“显示所有会话中的进程”。
7. 在“可用进程”对话框中，选择运行该 Web 应用程序的 ASP.NET 辅助进程，单击“附加”，然后单击“关闭”。

   默认情况下，对于 IIS 5.*x*，辅助进程是 aspnet_wp.exe（在 Windows 2000 和 Windows XP 上），对于 IIS 6.0，辅助进程是 w3wp.exe（在 Windows Server 2003 上）。
8. 从“文件”菜单中单击“打开”，再单击“文件”。
9. 在“打开文件”窗口中，浏览到源代码，选择该代码，然后单击“打开”。

完成此设置后，可以继续执行“使用公共语言运行库调试器进行调试”节中的操作。

使用公共语言运行库调试器进行调试

1. 将光标移到文件中要设置断点的位置，然后按 F9 键。
2. 重复上面的步骤以添加更多断点。
3. 从“调试”菜单中，单击“启动”。

   程序在遇到断点或运行时错误时停止运行。此时，将打开几个窗口，以便执行进一步的调试任务。
4. 若要停止调试但保持程序运行，请从“调试”菜单上，选择“全部分离”。

   否则，停止调试后，程序也将终止。

备注

当调试从命令行编译的程序时，dgbclr 在您每次开始调试时都会重新读取编译的程序。因此，您可以修改 JScript 代码并（在重新编译代码后）检查那些更改的效果。

请参见

任务

使用 Visual Studio 编写 JScript 代码

概念

从命令行生成

使用 Visual Studio 调试 JScript

CLR 调试器 (DbgCLR.exe)

其他资源

调试器指南

脚本疑难解答

所有编程语言都包含一些潜在的陷井，无论是新手还是经验丰富的用户都会始料不及。以下是您编写 JScript 脚本时可能遇到的一些潜在麻烦。

语法错误

由于编程语言中的语法要比自然语言中的语法要严格得多，所以在编写脚本时务必要严格地注意细节。例如，如果希望某个参数为字符串，但却忘记用引号将它括起来，就会出现错误。

脚本解释的顺序

在网页中，JScript 解释取决于每个浏览器的 HTML 分析过程。<HEAD> 标记内的脚本在 <BODY> 标记内的文本之前解释。因此，当浏览器分析 <HEAD> 元素时，在 <BODY> 标记中创建的对象不存在，脚本不能对这些对象进行处理。

注意
此行为是 Internet Explorer 所特有的。ASP 和 WSH 具有不同的执行模型（像其他宿主一样）。

自动类型强制

JScript 是具有自动强制的松散类型化语言。因此，尽管具有不同类型的值不全等，但下面的示例中的表达式的计算结果为 **true**。

```
"100" == 100;
false == 0;
```

若要检查类型和值是否均相同，请使用全等运算符 ===。以下两个表达式的计算结果均为 false：

```
"100" === 100;
false === 0;
```

运算符优先级

在计算表达式时，运算执行的顺序取决于运算符优先级，而不是取决于表达式中运算符的顺序。因此，在下面的示例中，虽然表达式中减法运算符出现在乘法运算符之前，但是先计算相乘。

```
theRadius = aPerimeterPoint - theCenterpoint * theCorrectionFactor;
```

有关更多信息，请参见运算符优先级。

将 for...in 循环用于对象

当脚本使用 **for...in** 循环逐个通过对象的属性时，将对象字段赋给循环计数器变量的顺序不一定能预测或控制。此外，在不同的语言实现中，顺序可能会有所不同。有关更多信息，请参见 for...in 语句。

with 关键字

虽然 **with** 关键字为已存在于指定对象中的属性的寻址带来方便，但却不能用来为对象添加属性。若要在对象中创建新的属性，必须明确地引用该对象。有关更多信息，请参见 with 语句。

this 关键字

虽然 **this** 关键字存在于对象定义内，但是如果当前执行的函数不是对象定义，则一般不能使用 **this** 或类似的关键字来引用该函数。如果将函数作为方法赋给对象，则脚本可以在函数内使用 **this** 关键字来引用该对象。有关更多信息，请参见 this 语句。

编写在 Internet Explorer 或 ASP.NET 中编写脚本的脚本

当解释器遇到 </SCRIPT> 标记时，该标记将终止当前脚本。若要显示"</SCRIPT>"本身，请将其书写为两个或多个字符串（例如"</SCR"和"IPT>"），随后脚本可以在写出这两个字符串的语句中将它们串联在一起。

Internet Explorer 中的隐式窗口引用

因为可以同时打开多个窗口，所以任何隐式窗口引用均指向当前窗口。对于其他窗口，则必须使用显式引用。

请参见

任务

使用 Visual Studio 编写 JScript 代码

概念

使用 Visual Studio 调试 JScript

其他资源

编写、编译、调试 JScript 代码

使用 JScript 显示信息

程序一般要向用户显示信息。最基本的方法是显示文本字符串。JScript 程序可以通过不同的过程来显示信息，根据程序的用途而定。使用 JScript 有三种常见的方式：在 ASP.NET 页中使用、在客户端网页中使用以及从命令行使用。此处讨论每一种环境下的一些显示方法。

本节内容

从命令行程序显示

解释如何使用 JScript**print** 语句或 .NET FrameworkShow 方法从命令行程序显示数据。

从 ASP.NET 显示

演示如何使用 <%= %> 构造从 ASP.NET 程序显示数据。

在浏览器中显示信息

解释如何使用 **write** 或 **writeln** 方法直接在浏览器中显示数据。

使用消息框

描述如何使用窗口对象的 **alert**、**confirm** 和 **prompt** 方法来创建提示消息框（此消息框通常要求用户输入）。

相关章节

JScript 参考

列出组成 JScript 语言参考的元素，并为每个元素提供正确语法的示例。

System.Windows.Forms.MessageBox.Show

阐释 MessageBox.Show 方法的语法和返回不同结果的选项。

从 ASP.NET 显示

可以使用多种方法从 ASP.NET 程序显示信息。一种方法是使用 <%= %> 结构。另一种方法是使用 **Response.Write** 语句。

使用 <%= %>

从 ASP.NET 程序显示信息的最简单方法是使用 <%= %> 结构。在等号后面输入的值将写入当前页。下面的代码显示 name 变量的值。

```
Hello <%= name %>!
```

如果名称的值是“Frank”，此代码将在当前页中写入以下字符串：

```
Hello Frank!
```

<%= %> 结构在显示单条信息时最为有用。

Response.Write 语句

显示文本的另一种方法是使用 **Response.Write** 语句。可以将它放在 <% %> 块内。

```
<% Response.Write("Hello, World!") %>
```

Response.Write 语句还可以在脚本块内的函数或方法中使用。下面的示例显示了一个包含 **Response.Write** 语句的函数。

注意

在 ASP.NET 页中，函数和变量应当在 <script> 块内定义，而可执行代码必须括在 <% %> 块内。

```
<script runat="server" language="JScript">
   function output(str) {
      Response.Write(str);
   }
   var today = new Date();
</script>
Today's date is <% output(today); %>. <BR>
```

Response.Write 语句的输出被合并到正在处理的页中。这样就允许 **Response.Write** 的输出编写代码，而该代码又可显示文本。例如，下面的代码编写一个脚本块，该脚本块在正在访问该页的浏览器的警报窗口中显示当前日期（服务器上的）。<script> 标记被拆分开，因此服务器将不处理此标记。

```
<script runat="server" language="JScript">
   function popup(str) {
      Response.Write("<scr"+"ipt> alert('"+str+"') </scr"+"ipt>");
   }
   var today = new Date();
</script>
<% popup(today); %>
```

有关更多信息，请参见 Response。

请参见

概念

ASP.NET 概述

其他资源

使用 JScript 显示信息

从命令行程序显示

JScript 从命令行程序显示数据有三种方式。Microsoft JScript 命令行编译器提供 **print** 语句。Console 类提供其他方法，使得与使用控制台的用户之间的交互更加便利。

Show 方法显示来自弹出框的信息并接受输入。

print 语句

显示信息的最常用方式是使用 **print** 语句。它带有一个参数，即要显示的字符串，在命令行窗口中该字符串后面跟一个换行符。

单引号或双引号都可以包围字符串，这就允许引文中再包含引号或撇号。

```
          print("Pi is approximately equal to " + Math.PI);
print();
```

注意

只有用 JScript 命令行编译器编译的程序可以使用 **print** 语句。在 ASP.NET 页中使用 **print** 会导致编译器错误。

Console 类

Console 类公开一些方法和属性，以便利同控制台用户交互。**Console** 类的 **WriteLine** 方法提供类似于 **print** 语句的功能。**Write** 方法显示不带换行符的字符串。**Console** 类的另一个有用的方法是 **ReadLine** 方法，该方法读取从控制台输入的一行文本。

要使用 .NET Framework 的类和方法，首先使用 **import** 语句导入类所属的命名空间。要调用方法，可使用完全限定名，或者，如果当前范围内没有同名的方法，也可以只使用名称。

```
import System;
System.Console.WriteLine("What is your name: ");
var name : String = Console.Readline();
Console.Write("Hello ");
Console.Write(name);
Console.Write("!");
```

此程序要求从控制台输入一个名称。输入名称 Pete 后，此程序显示：

```
What is your name:
Pete
Hello Pete!
```

有关更多信息，请参见 **Console**。

Show 方法

Show 方法可以通用，因为它是重载的。最简单的重载有一个参数，即要显示的文本字符串。此消息框是模式化的。

注意

如果某个窗口或窗体在被显式关闭之前一直保留焦点，则该窗口或窗体就是模式化的。对话框和消息通常为模式化的。例如，在模式化的对话框中，只有在该对话框中选择“确定”以后才能访问另一个窗口。

```
import System.Windows.Forms;
System.Windows.Forms.MessageBox.Show("Welcome! Press OK to continue.");
MessageBox.Show("Great! Now press OK again.");
```

可以使用 **Show** 方法的其他重载以包含一个标题、其他按钮、一个图标或默认按钮。有关更多信息，请参见 **Show**。

请参见

参考

print 语句

import 语句

其他资源

使用 JScript 显示信息

在浏览器中显示信息

尽管浏览器支持大多数 JScript 功能，但只有在服务器端才支持面向 .NET Framework、基于类的对象、数据类型、枚举、条件编译指令和 **const** 语句的那些新功能。因此，您应该在服务器端脚本中以独占方式使用这些功能。有关更多信息，请参见 JScript 版本信息。

每当想要在浏览器（客户端）中运行一个脚本时，有经验的开发人员就会在代码中包括检测脚本引擎版本的代码。在脚本检测引擎版本后，它可以将浏览器重定向到具有与该浏览器的脚本引擎兼容的脚本的页。有关更多信息，请参见检测浏览器功能。

JScript 通过浏览器 document 对象的 **write** 和 **writeln** 方法在浏览器中显示信息。它还可以在浏览器的窗体中以及警报、提示和确认消息框中显示信息。有关更多信息，请参见使用消息框。

使用 document.write 和 document.writeln

显示信息的最常用方法是 **document** 对象的 **write** 方法。它带有一个参数，即它在浏览器中显示的字符串。该字符串要么是纯文本格式，要么是 HTML 格式。

由于字符串可以用单引号或双引号引起来，因此可以引用包含引号或撇号的内容。

```
document.write("Pi is approximately equal to " + Math.PI);
document.write();
```

注意

下面的简单函数使您在每次想要将文本显示在浏览器窗口中时，不必再键入 `document.write`。如果要写入的内容尚未定义，此函数不会通知您，但它允许您发出一个显示空行的 `w();` 命令。

```
function w(m) { // Write function.
   m = String(m); //  Make sure that the m variable is a string.
   if ("undefined" != m) { // Test for empty write or other undefined item.
      document.write(m);
   }
   document.write("<br>");
}

w('<IMG SRC="horse.gif">');
w();
w("This is an engraving of a horse.");
w();
```

writeln 方法几乎与 **write** 方法相同，它将一个换行符追加到提供的字符串的后面。在 HTML 中，这通常只能在项后面显示一个空格；在 <PRE> 和 <XMP> 标记内，换行符是按字面来解释的，浏览器可将其显示出来。

如果在调用 **write** 方法时文档不在已打开并进行分析的进程中，则 **write** 方法将打开并清除该文档。这样可能会导致意外的结果。下面的示例显示的脚本旨在一分钟显示一次时间，但它在第一次显示时间之后无法往下执行，因为它在进程中清除了自身。

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JScript">
function singOut()  {
   var theMoment = new Date();
   var theHour = theMoment.getHours();
   var theMinute = theMoment.getMinutes();
   var theDisplacement = (theMoment.getTimezoneOffset() / 60);
   theHour -= theDisplacement;
   if (theHour > 23)  {
      theHour -= 24
   }
   // The following line clears the script the second time it is run.
   document.write(theHour + " hours, " + theMinute + " minutes, Coordinated Universal Time.
");
   window.setTimeout("singOut();", 60000);
}
```

```
</SCRIPT>
</HEAD>
<BODY>
<SCRIPT>
singOut();
</SCRIPT>
</BODY>
</HTML>
```

如果使用 window 对象的 **alert** 方法而不用 **document.write**，那么该脚本就可以执行。

```
   // This line produced the intended result.
window.alert(theHour + " hours, " + theMinute + " minutes, Coordinated Universal Time.");
```

在 Internet Explorer 5 以及更高版本中，首选 **element.innerText** 或 **element.innerHTML**。

清除当前文档

document 对象的 **clear** 方法可清空当前文档。此方法还会清除您的脚本（以及文档的其余部分），因此一定要特别注意使用它的时机和方式。

```
document.clear();
```

请参见

概念

使用消息框

检测浏览器功能

其他资源

使用 JScript 显示信息

使用消息框

尽管浏览器支持大多数 JScript 功能，但只有在服务器端才支持面向 .NET Framework、基于类的对象、数据类型、枚举、条件编译指令和 **const** 语句的那些新功能。因此，您应该在服务器端脚本中以独占方式使用这些功能。有关更多信息，请参见 JScript 版本信息。

每当想要在浏览器（客户端）中运行一个脚本时，有经验的开发人员就会在代码中包括检测脚本引擎版本的代码。在脚本检测引擎版本后，它可以将浏览器重定向到具有与该浏览器的脚本引擎兼容的脚本的页。有关更多信息，请参见检测浏览器功能。

JScript 使用浏览器的 **alert**、**confirm** 和 **prompt** 消息框获取用户的输入。这些框是 **window** 对象的方法。由于 **window** 对象在对象层次结构的顶部，因此实际上不需要使用这些消息框中任何一个的全名（例如 `window.alert()`），但使用全名的好处在于可帮助您记住它们属于哪个对象。

Alert 消息框

alert 方法有一个参数，即要在警报消息框中显示的文本字符串。该字符串不是 HTML。此消息框提供了一个“确定”按钮用来关闭消息框，而且此消息框是模式，即用户必须在关闭此消息框后才能继续。

```
window.alert("Welcome! Press OK to continue.");
```

确认消息框

确认消息框包含“确定”和“取消”按钮，提出的问题将有两种可能的结果。**confirm** 方法返回 **true** 和 **false**。此消息框也是模式：用户必须响应（单击某个按钮）继而关闭它之后才能继续。

```
var truthBeTold = window.confirm("Click OK to continue. Click Cancel to stop.");
if (truthBeTold)
   window.alert("Welcome to our Web page!");
else
   window.alert("Bye for now!");
```

提示消息框

提示消息框包含“确定”和“取消”按钮，提供的文本字段接受响应提示的文本。如果提供了第二个字符串参数，则提示消息框的默认响应是在文本字段中显示第二个字符串。否则，默认文本为“undefined”。

像 **alert** 和 **confirm** 方法一样，**prompt** 显示一个模式消息框。用户必须关闭它以后才能继续。

```
var theResponse = window.prompt("Welcome?","Enter your name here.");
document.write("Welcome "+theResponse+".<BR>");
```

请参见

概念

在浏览器中显示信息

检测浏览器功能

其他资源

使用 JScript 显示信息

正则表达式介绍

这些章节介绍正则表达式的概念，并说明如何在 JScript 中创建和使用它们。

尽管每个主题都自成一体，但我们还是建议您按顺序细读这些主题，以便最好地理解本材料。许多主题依赖于对前面的主题所介绍的功能或概念的理解。

本节内容

正则表达式

通过与大多数读者已经熟悉的一些概念进行比较，说明正则表达式的概念。

正则表达式的用途

通过实例说明正则表达式如何扩展常规搜索条件。

正则表达式语法

说明构成正则表达式的字符、构成元字符的字符以及元字符的行为。

生成正则表达式

描述正则表达式的组件以及组件和分隔符之间的关系。

优先级顺序

说明如何计算正则表达式以及正则表达式的序列和语法对结果有何影响。

普通字符

区分普通字符与元字符，并说明如何将多个单字符正则表达式组合在一起，以创建更大的表达式。

JScript 中的特殊字符

说明转义符的概念以及如何创建与元字符匹配的正则表达式。

不可打印字符

列出用于表示正则表达式中非打印字符的转义序列。

字符匹配

说明正则表达式如何使用句点、转义符和中括号创建返回特定结果的序列。

JScript 中的限定符

说明当无法指定多少字符构成一个匹配时如何创建正则表达式。

定位点

说明如何将正则表达式固定到行首或行尾，以及如何创建在单词内、在单词的开头或者在单词的结尾出现的正则表达式。

替换和分组

说明替换如何使用"|"字符以允许在两个或更多替换选项之间进行选择，以及分组如何与替换配合工作以使结果更加精确。

JScript 中的反向引用

说明如何创建可以访问存储匹配模式的组成部分的正则表达式，而不必重新创建构成这一匹配模式的正则表达式。

相关章节

.NET Framework 正则表达式

阐明正则表达式的模式匹配表示法如何使开发人员能够快速分析大量的文本，以查找特定字符模式；提取、编辑、替换或删除文本子字符串；或将提取的字符串添加到集合中，以便生成报告。

正则表达式示例

提供指向代码示例的一组链接，这些示例说明正则表达式在常见的应用程序中的用法。

正则表达式

除非您以前使用过正则表达式，否则您可能不熟悉此术语。但是，毫无疑问，您已经使用过不涉及脚本的某些正则表达式概念。

正则表达式示例

例如，您很可能使用 ? 和 * 通配符来查找硬盘上的文件。通配符匹配文件名中的单个字符，而 * 通配符匹配零个或多个字符。像 data?.dat 这样的模式将查找下列文件：

data1.dat

data2.dat

datax.dat

dataN.dat

使用 * 字符代替 ? 字符扩大了找到的文件的数量。data*.dat 匹配下列所有文件：

data.dat

data1.dat

data2.dat

data12.dat

datax.dat

dataXYZ.dat

尽管这种搜索方法很有用，但它还是有限的。和 * 通配符的能力引入了正则表达式所依赖的概念，但正则表达式功能更强大，而且更加灵活。

请参见

其他资源

正则表达式介绍

正则表达式的用途

典型的搜索和替换操作要求您提供与预期的搜索结果匹配的确切文本。虽然这种技术对于对静态文本执行简单搜索和替换任务可能已经足够了，但它缺乏灵活性，若采用这种方法搜索动态文本，即使不是不可能，至少也会变得很困难。

示例方案

通过使用正则表达式，可以：

- 测试字符串内的模式。

  例如，可以测试输入字符串，以查看字符串内是否出现电话号码模式或信用卡号码模式。这称为数据验证。

- 替换文本。

  可以使用正则表达式来识别文档中的特定文本，完全删除该文本或者用其他文本替换它。

- 基于模式匹配从字符串中提取子字符串。

  可以查找文档内或输入域内特定的文本。

例如，您可能需要搜索整个网站，删除过时的材料，以及替换某些 HTML 格式标记。在这种情况下，可以使用正则表达式来确定在每个文件中是否出现该材料或该 HTML 格式标记。此过程将受影响的文件列表缩小到包含需要删除或更改的材料的那些文件。然后可以使用正则表达式来删除过时的材料。最后，可以使用正则表达式来搜索和替换标记。

正则表达式在 JScript 或 C 等语言中也很有用，这些语言的字符串处理能力还不为人们所知。

请参见

其他资源

正则表达式介绍

正则表达式语法

正则表达式是一种文本模式，包括普通字符（例如，a 到 z 之间的字母）和特殊字符（称为“*元字符*”）。模式描述在搜索文本时要匹配的一个或多个字符串。

正则表达式示例

表达式	匹配
/^\s*$/	匹配空行。
/\d{2}-\d{5}/	验证由两位数字、一个连字符再加 5 位数字组成的 ID 号。
/<\s*(\S+)(\s[^>]*)?>[\s\S]*<\s*\/\1\s*>/	匹配 HTML 标记。

下表包含了元字符的完整列表以及它们在正则表达式上下文中的行为：

字符	说明
\	将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如，“n”匹配字符“n”。“\n”匹配换行符。序列“\\”匹配“\”，“\(”匹配“(”。
^	匹配输入字符串开始的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，^ 还会与“\n”或“\r”之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，$ 还会与“\n”或“\r”之前的位置匹配。
*	零次或多次匹配前面的字符或子表达式。例如，zo* 匹配“z”和“zoo”。* 等效于 {0,}。
+	一次或多次匹配前面的字符或子表达式。例如，“zo+”与“zo”和“zoo”匹配，但与“z”不匹配。+ 等效于 {1,}。
?	零次或一次匹配前面的字符或子表达式。例如，“do(es)?”匹配“do”或“does”中的“do”。? 等效于 {0,1}。
{*n*}	*n* 是非负整数。正好匹配 *n* 次。例如，“o{2}”与“Bob”中的“o”不匹配，但与“food”中的两个“o”匹配。
{*n*,}	*n* 是非负整数。至少匹配 *n* 次。例如，“o{2,}”不匹配“Bob”中的“o”，而匹配“foooood”中的所有 o。“o{1,}”等效于“o+”。“o{0,}”等效于“o*”。
{*n*, *m*}	*M* 和 *n* 是非负整数，其中 *n* <= *m*。匹配至少 *n* 次，至多 *m* 次。例如，“o{1,3}”匹配“fooooood”中的头三个 o。'o{0,1}' 等效于 'o?'。注意：您不能将空格插入逗号和数字之间。
?	当此字符紧随任何其他限定符（*、+、?、{*n*}、{*n*,}、{*n*,*m*}）之后时，匹配模式是“非贪心的”。“非贪心的”模式匹配搜索到的、尽可能短的字符串，而默认的“贪心的”模式匹配搜索到的、尽可能长的字符串。例如，在字符串“oooo”中，“o+?”只匹配单个“o”，而“o+”匹配所有“o”。
.	匹配除“\n”之外的任何单个字符。若要匹配包括“\n”在内的任意字符，请使用诸如“[\s\S]”之类的模式。
(*pattern*)	匹配 *pattern* 并捕获该匹配的子表达式。可以使用 **$0…$9** 属性从结果“匹配”集合中检索捕获的匹配。若要匹配括号字符 ( )，请使用“\(”或者“\)”。
(?:*pattern*)	匹配 *pattern* 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。这对于用“or”字符 (|) 组合模式部件的情况很有用。例如，'industr(?:y|ies) 是比 'industry|industries' 更经济的表达式。

(?=*pattern*)	执行正向预测先行搜索的子表达式，该表达式匹配处于匹配 *pattern* 的字符串的起始点的字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?=95|98|NT|2000)' 匹配"Windows 2000"中的"Windows"，但不匹配"Windows 3.1"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
(?!*pattern*)	执行反向预测先行搜索的子表达式，该表达式匹配不处于匹配 *pattern* 的字符串的起始点的搜索字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?!95|98|NT|2000)' 匹配"Windows 3.1"中的 "Windows"，但不匹配"Windows 2000"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
x|*y*	匹配 *x* 或 *y*。例如，'z|food' 匹配"z"或"food"。'(z|f)ood' 匹配"zood"或"food"。
[*xyz*]	字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。
[^*xyz*]	反向字符集。匹配未包含的任何字符。例如，"[^abc]"匹配"plain"中的"p"。
[*a-z*]	字符范围。匹配指定范围内的任何字符。例如，"[a-z]"匹配"a"到"z"范围内的任何小写字母。
[^*a-z*]	反向范围字符。匹配不在指定的范围内的任何字符。例如，"[^a-z]"匹配任何不在"a"到"z"范围内的任何字符。
\b	匹配一个字边界，即字与空格间的位置。例如，"er\b"匹配"never"中的"er"，但不匹配"verb"中的"er"。
\B	非字边界匹配。"er\B"匹配"verb"中的"er"，但不匹配"never"中的"er"。
\c*x*	匹配 *x* 指示的控制字符。例如，\cM 匹配 Control-M 或回车符。*x* 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是"c"字符本身。
\d	数字字符匹配。等效于 [0-9]。
\D	非数字字符匹配。等效于 [^0-9]。
\f	换页符匹配。等效于 \x0c 和 \cL。
\n	换行符匹配。等效于 \x0a 和 \cJ。
\r	匹配一个回车符。等效于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等。与 [\f\n\r\t\v] 等效。
\S	匹配任何非空白字符。与 [^ \f\n\r\t\v] 等效。
\t	制表符匹配。与 \x09 和 \cI 等效。
\v	垂直制表符匹配。与 \x0b 和 \cK 等效。
\w	匹配任何字类字符，包括下划线。与"[A-Za-z0-9_]"等效。
\W	与任何非单词字符匹配。与"[^A-Za-z0-9_]"等效。

\x*n*	匹配 *n*，此处的 *n* 是一个十六进制转义码。十六进制转义码必须正好是两位数长。例如，“\x41”匹配“A”。“\x041”与“\x04”&“1”等效。允许在正则表达式中使用 ASCII 代码。
num	匹配 *num*，此处的 *num* 是一个正整数。到捕获匹配的反向引用。例如，“(.)\1”匹配两个连续的相同字符。
n	标识一个八进制转义码或反向引用。如果 *n* 前面至少有 *n* 个捕获子表达式，那么 *n* 是反向引用。否则，如果 *n* 是八进制数 (0-7)，那么 *n* 是八进制转义码。
nm	标识一个八进制转义码或反向引用。如果 *nm* 前面至少有 *nm* 个捕获子表达式，那么 *nm* 是反向引用。如果 *nm* 前面至少有 *n* 个捕获，则 *n* 是反向引用，后面跟有字符 *m*。如果两种前面的情况都不存在，则 *nm* 匹配八进制值 *nm*，其中 *n* 和 *m* 是八进制数字 (0-7)。
nml	当 *n* 是八进制数 (0-3)，*m* 和 *l* 是八进制数 (0-7) 时，匹配八进制转义码 *nml*。
\u*n*	匹配 *n*，其中 *n* 是以四位十六进制数表示的 Unicode 字符。例如，\u00A9 匹配版权符号 (©)。

请参见

其他资源

正则表达式介绍

生成正则表达式

正则表达式的结构与算术表达式的结构类似。即，各种元字符和运算符可以将小的表达式组合起来，创建大的表达式。

分隔符

通过在一对分隔符之间放置表达式模式的各种组件，就可以构建正则表达式。对于 JScript，分隔符是正斜杠 (/) 字符。例如：

```
/expression/
```

在上面的示例中，正则表达式模式 (*expression*) 存储在 **RegExp** 对象的 **Pattern** 属性中。

正则表达式的组件可以是单个字符、字符集、字符的范围、在几个字符之间选择或者所有这些组件的任何组合。

请参见

其他资源

正则表达式介绍

优先级顺序

正则表达式从左到右进行计算，并遵循优先级顺序，这与算术表达式非常类似。

运算符

下表从最高到最低说明了各种正则表达式运算符的优先级顺序：

运算符	说明
\	转义符
(), (?:), (?=), []	括号和中括号
*, +, ?, {n}, {n,}, {n,m}	限定符
^, $, *任何元字符*、*任何字符*	定位点和序列
|	替换

字符具有高于替换运算符的优先级，使得"m|food"匹配"m"或"food"。若要匹配"mood"或"food"，请使用括号创建子表达式，从而产生"(m|f)ood"。

请参见

其他资源

正则表达式介绍

普通字符

普通字符包括没有显式指定为元字符的所有可打印和不可打印字符。这包括所有大写和小写字母、所有数字、所有标点符号和一些其他符号。

简单表达式

正则表达式的最简单形式是在搜索字符串中匹配其本身的单个普通字符。例如，单字符模式，如 A，不论出现在搜索字符串中的何处，它总是匹配字母 A。下面是一些单字符正则表达式模式的示例：

```
/a/
/7/
/M/
```

可以将许多单字符组合起来以形成大的表达式。例如，以下正则表达式组合了单字符表达式：a、7 和 M。

```
/a7M/
```

请注意，没有串联运算符。只须在一个字符后面键入另一个字符。

请参见

其他资源

正则表达式介绍

JScript 中的特殊字符

许多元字符要求在试图匹配它们时特别对待。若要匹配这些特殊字符，必须首先使字符"*转义*"，即，将反斜杠字符 (\) 放在它们前面。下表列出了特殊字符以及它们的含义：

特殊字符表

特殊字符	注释
$	匹配输入字符串结尾的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，那么 $ 还匹配 \n 或 \r 前面的位置。若要匹配 $ 字符本身，请使用 \$。
( )	标记子表达式的开始和结束。可以捕获子表达式以供以后使用。若要匹配这两个字符，请使用 \(和 \)。
*	零次或多次匹配前面的字符或子表达式。若要匹配 * 字符，请使用 *。
+	一次或多次匹配前面的字符或子表达式。若要匹配 + 字符，请使用 \+。
.	匹配除换行符 \n 之外的任何单个字符。若要匹配 .，请使用 \。
[ ]	标记中括号表达式的开始。若要匹配这些字符，请使用 \[和 \]。
?	零次或一次匹配前面的字符或子表达式，或指示"非贪心"限定符。若要匹配 ? 字符，请使用 \?。
\	将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如，字符 n 匹配字符 n。\n 匹配换行符。序列 \\ 匹配 \，序列 \(匹配 (。
/	表示文本正则表达式的开始或结束。若要匹配 / 字符，请使用 \/。
^	匹配输入字符串开始处的位置，但在中括号表达式中使用的情况除外，在那种情况下它对字符集求反。若要匹配 ^ 字符本身，请使用 \^。
{ }	标记限定符表达式的开始。若要匹配这些字符，请使用 \{ 和 \}。
|	指出在两个项之间进行选择。要匹配 |，请使用 \\|。

请参见

其他资源

正则表达式介绍

不可打印字符

非打印字符也可以是正则表达式的组成部分。下表列出了表示非打印字符的转义序列：

转义序列

字符	含义
\c*x*	匹配 *x* 指示的控制字符。例如，\cM 匹配 Control-M 或回车符。*x* 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是"c"字符本身。
\f	换页符匹配。等效于 \x0c 和 \cL。
\n	换行符匹配。等效于 \x0a 和 \cJ。
\r	匹配一个回车符。等效于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等。与 [\f\n\r\t\v] 等效。
\S	匹配任何非空白字符。与 [^ \f\n\r\t\v] 等效。
\t	制表符匹配。与 \x09 和 \cI 等效。
\v	垂直制表符匹配。与 \x0b 和 \cK 等效。

请参见

其他资源

正则表达式介绍

字符匹配

句点 (.) 匹配字符串中的各种打印或非打印字符，只有一个字符例外。这个例外就是换行符 (\n)。下面的正则表达式匹配 aac、abc、acc、adc 等等，以及 a1c、a2c、a-c 和 a#c：

```
/a.c/
```

若要匹配包含文件名的字符串，而句点 (.) 是输入字符串的组成部分，请在正则表达式中的句点前面加反斜杠 (\) 字符。举例来说明，下面的正则表达式匹配 filename.ext：

```
/filename\.ext/
```

这些表达式只让您匹配"*任何*"单个字符。可能需要匹配列表中的特定字符组。例如，可能需要查找用数字表示的章节标题（Chapter 1、Chapter 2 等等）。

中括号表达式

若要创建匹配字符组的一个列表，请在方括号（[和]）内放置一个或更多单个字符。当字符括在中括号内时，该列表称为"*中括号表达式*"。与在任何别的位置一样，普通字符在中括号内表示其本身，即，它在输入文本中匹配一次其本身。大多数特殊字符在中括号表达式内出现时失去它们的意义。不过也有一些例外，如：

- 如果] 字符不是第一项，它结束一个列表。若要匹配列表中的] 字符，请将它放在第一位，紧跟在开始 [后面。
- \ 字符继续作为转义符。若要匹配 \ 字符，请使用 \\。

括在中括号表达式中的字符只匹配处于正则表达式中该位置的单个字符。以下正则表达式匹配 Chapter 1、Chapter 2、Chapter 3、Chapter 4 和 Chapter 5：

```
/Chapter [12345]/
```

请注意，单词 *Chapter* 和后面的空格的位置相对于中括号内的字符是固定的。中括号表达式指定的只是匹配紧跟在单词 *Chapter* 和空格后面的单个字符位置的字符集。这是第九个字符位置。

若要使用范围代替字符本身来表示匹配字符组，请使用连字符 (-) 将范围中的开始字符和结束字符分开。单个字符的字符值确定范围内的相对顺序。下面的正则表达式包含范围表达式，该范围表达式等效于上面显示的中括号中的列表。

```
/Chapter [1-5]/
```

当以这种方式指定范围时，开始值和结束值两者都包括在范围内。注意，还有一点很重要，按 Unicode 排序顺序，开始值必须在结束值的前面。

若要在中括号表达式中包括连字符，请采用下列方法之一：

- 用反斜杠将它转义：

  ```
  [\-]
  ```

- 将连字符放在中括号列表的开始或结尾。下面的表达式匹配所有小写字母和连字符：

  ```
  [-a-z]
  [a-z-]
  ```

- 创建一个范围，在该范围中，开始字符值小于连字符，而结束字符值等于或大于连字符。下面的两个正则表达式都满足这一要求：

  ```
  [!--]
  [!-~]
  ```

若要查找不在列表或范围内的所有字符，请将插入符号 (^) 放在列表的开头。如果插入字符出现在列表中的其他任何位置，则它匹配其本身。下面的正则表达式匹配编号大于 5 的章节标题：

```
/Chapter [^12345]/
```

在上面的示例中，表达式在第九个位置匹配 1、2、3、4 或 5 之外的任何数字字符。这样，例如，Chapter 7 就是一个匹配项，Chapter 9 也是一个匹配项。

上面的表达式可以使用连字符 (-) 来表示：

```
/Chapter [^1-5]/
```

中括号表达式的典型用途是指定任何大写或小写字母或任何数字的匹配。下面的表达式指定这样的匹配：

```
/[A-Za-z0-9]/
```

请参见

其他资源

正则表达式介绍

JScript 中的限定符

如果您不能指定构成匹配的字符的数量，那么正则表达式支持限定符的概念。这些限定符使您能够指定，为使匹配为真，正则表达式的某个给定组件必须出现多少次。

限定符含义

字符	说明
*	零次或多次匹配前面的字符或子表达式。例如，zo* 匹配 z 和 zoo。* 等效于 {0,}。
+	一次或多次匹配前面的字符或子表达式。例如，zo+ 匹配 zo 和 zoo，但不匹配 z。+ 等效于 {1,}。
?	零次或一次匹配前面的字符或子表达式。例如，do(es)? 匹配 do 或 does 中的 do。? 等效于 {0,1}。
{*n*}	*n* 是非负整数。正好匹配 *n* 次。例如，o{2} 不匹配 Bob 中的 o，但匹配 food 中的两个 o。
{*n*,}	*n* 是非负整数。至少匹配 *n* 次。例如，o{2,} 不匹配 Bob 中的 o，而匹配 foooood 中的所有 o。o{1,} 等效于 o+。o{0,} 等效于 o*。
{*n*, *m*}	*m* 和 *n* 是非负整数，其中 *n* <= *m*。匹配至少 *n* 次，至多 *m* 次。例如，o{1,3} 匹配 fooooood 中的头三个 o。o{0,1} 等效于 o?。注意：您不能将空格插入逗号和数字之间。

由于章节编号在大的输入文档中会很可能超过九，所以您需要一种方式来处理两位或三位章节编号。限定符给您这种能力。下面的正则表达式匹配编号为任何位数的章节标题：

```
/Chapter [1-9][0-9]*/
```

请注意，限定符出现在范围表达式之后。因此，它应用于整个范围表达式，在本例中，只指定从 0 到 9 的数字（包括 0 和 9）。

这里不使用 + 限定符，因为在第二个位置或后面的位置不一定需要有一个数字。也不使用？字符，因为它将章节编号限制到只有两位数。您需要至少匹配 Chapter 和空格字符后面的一个数字。

如果您知道章节编号被限制为只有 99 章，可以使用下面的表达式来至少指定一位但至多两位数字。

```
/Chapter [0-9]{1,2}/
```

上面的表达式的缺点是，大于 99 的章节编号仍只匹配开头两位数字。另一个缺点是 Chapter 0 也将匹配。只匹配两位数字的更好的表达式如下：

```
/Chapter [1-9][0-9]?/
```

或

```
/Chapter [1-9][0-9]{0,1}/
```

*、+ 和 ? 限定符都被称为"*贪心*的"，因为它们匹配尽可能多的文本。但是，有时您只需要最小的匹配。

例如，您可能搜索 HTML 文档，以查找括在 H1 标记内的章节标题。该文本在您的文档中如下：

```
<H1>Chapter 1 – Introduction to Regular Expressions</H1>
```

下面的表达式匹配从开始小于符号 (<) 到关闭 H1 标记的大于符号 (>) 之间的所有内容。

```
/<.*>/
```

如果您只需要匹配开始 H1 标记，下面的"非贪心"表达式只匹配 <H1>。

```
/<.*?>/
```

通过在 *、+ 或 ? 限定符之后放置 ?，该表达式从"贪心"表达式转换为"非贪心"表达式或者最小匹配。

请参见

其他资源

正则表达式介绍

定位点

本节前面的主题中的示例只涉及章节标题查找。字符串 Chapter 后面跟空格和数字的任何匹配项可能是实际章节标题，或者也可能是指向另一章的交叉引用。由于真正的章节标题总是出现在行的开始，所以设计一种方法只查找标题而不查找交叉引用可能很有用。

定位点工作方式

定位点提供该能力。定位点使您能够将正则表达式固定到行首或行尾。它们还使您能够创建这样的正则表达式，这些正则表达式出现在一个单词内、在一个单词的开头或者一个单词的结尾。下表包含正则表达式定位点以及它们的含义的列表：

字符	说明
^	匹配输入字符串开始的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，^ 还会与 \n 或 \r 之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，$ 还会与 \n 或 \r 之前的位置匹配。
\b	匹配一个字边界，即字与空格间的位置。
\B	非字边界匹配。

不能将限定符与定位点一起使用。由于在紧靠换行或者字边界的前面或后面不能有一个以上位置，因此不允许诸如 ^* 之类的表达式。

若要匹配一行文本开始处的文本，请在正则表达式的开始使用 ^ 字符。不要将 ^ 的这种用法与中括号表达式内的用法混淆。

若要匹配一行文本的结束处的文本，请在正则表达式的结束处使用 $ 字符。

若要在搜索章节标题时使用定位点，下面的正则表达式匹配一个章节标题，该标题只包含两个尾随数字，并且出现在行首：

```
/^Chapter [1-9][0-9]{0,1}/
```

真正的章节标题不仅出现行的开始处，而且它还是该行中仅有的文本。它即出现在行首又出现在同一行的结尾。下面的表达式能确保指定的匹配只匹配章节而不匹配交叉引用。通过创建只匹配一行文本的开始和结尾的正则表达式，就可做到这一点。

```
/^Chapter [1-9][0-9]{0,1}$/
```

匹配字边界稍有不同，但向正则表达式添加了很重要的能力。字边界是单词和空格之间的位置。非字边界是任何其他位置。下面的表达式匹配单词 *Chapter* 的开头三个字符，因为这三个字符出现字边界后面：

```
/\bCha/
```

\b 字符的位置是非常重要的。如果它位于要匹配的字符串的开始，它在单词的开始处查找匹配项。如果它位于字符串的结尾，它在单词的结尾处查找匹配项。例如，下面的表达式匹配单词 Chapter 中的字符串 *ter*，因为它出现在字边界的前面：

```
/ter\b/
```

下面的表达式匹配 Chapter 中的字符串 *apt*，但不匹配 aptitude 中的字符串 *apt*：

```
/\Bapt/
```

字符串 *apt* 出现在单词 *Chapter* 中的非字边界处，但出现在单词 *aptitude* 中的字边界处。对于 \B 非字边界运算符，位置并不重要，因为匹配不关心究竟是单词的开头还是结尾。

请参见

其他资源

正则表达式介绍

替换和分组

替换使用 | 字符来允许在两个或多个替换选项之间进行选择。例如，可以扩展章节标题正则表达式，以返回比章标题范围更广的匹配项。但是，这并不象您可能认为的那样简单。替换匹配 | 字符任一侧最大的表达式。

示例

您可能认为，下面的表达式匹配出现在行首和行尾、后面跟一个或两个数字的 Chapter 或 Section：

```
/^Chapter|Section [1-9][0-9]{0,1}$/
```

很遗憾，上面的正则表达式要么匹配行首的单词 *Chapter*，要么匹配行尾的单词 *Section* 及跟在其后的任何数字。如果输入字符串是 Chapter 22，那么上面的表达式只匹配单词 *Chapter*。如果输入字符串是 Section 22，那么该表达式匹配 Section 22。

若要使正则表达式更易于控制，可以使用括号来限制替换的范围，即，确保它只应用于两个单词 *Chapter* 和 *Section*。但是，括号也用于创建子表达式，并可能捕获它们以供以后使用，这一点在有关反向引用的那一节讲述。通过在上面的正则表达式的适当位置添加括号，就可以使该正则表达式匹配 Chapter 1 或 Section 3。

下面的正则表达式使用括号来组合 Chapter 和 Section，以便表达式正确地起作用：

```
/^(Chapter|Section) [1-9][0-9]{0,1}$/
```

尽管这些表达式正常工作，但 Chapter|Section 周围的括号还将捕获两个匹配字中的任一个供以后使用。由于在上面的表达式中只有一组括号，因此，只有一个被捕获的"*子匹配项*"。可以通过使用 **RegExp** 对象的 **$1-$9** 属性来引用此子匹配项。

在上面的示例中，您只需要使用括号来组合单词 *Chapter* 和 *Section* 之间的选择。若要防止匹配被保存以备将来使用，请在括号内正则表达式模式之前放置 ?:。下面的修改提供相同的能力而不保存子匹配项：

```
/^(?:Chapter|Section) [1-9][0-9]{0,1}$/
```

除 ?: 元字符外，两个其他非捕获元字符创建被称为"*预测先行*"匹配的某些内容。正向预测先行使用 ?= 指定，它匹配处于括号中匹配正则表达式模式的起始点的搜索字符串。反向预测先行使用 ?! 指定，它匹配处于与正则表达式模式不匹配的字符串的起始点的搜索字符串。

例如，假设您有一个文档，该文档包含指向 Windows 3.1、Windows 95、Windows 98 和 Windows NT 的引用。再进一步假设，您需要更新该文档，将指向 Windows 95、Windows 98 和 Windows NT 的所有引用更改为 Windows 2000。下面的正则表达式（这是一个正向预测先行的示例）匹配 Windows 95、Windows 98 和 Windows NT：

```
/Windows(?=95 |98 |NT )/
```

找到一处匹配后，紧接着就在匹配的文本（不包括预测先行中的字符）之后搜索下一处匹配。例如，如果上面的表达式匹配 Windows 98，将在 Windows 之后而不是在 98 之后继续搜索。

请参见

概念

JScript 中的反向引用

其他资源

正则表达式介绍

JScript 中的反向引用

正则表达式的最重要功能之一是存储匹配的模式的一部分以供以后重新使用的能力。您可能想起，若在正则表达式模式或模式的一部分两侧加上括号，就会导致表达式的一部分被存储到临时缓冲区中。可以通过使用非捕获元字符 ?:、?= 或 ?! 来重写捕获。

使用反向引用

每个捕获的子匹配项按照它们在正则表达式模式中从左到右出现的顺序存储。缓冲区编号从 1 开始，最多可存储 99 个捕获的子表达式。可以使用 *n* 来访问每个缓冲区，其中 *n* 是标识特定缓冲区的一位或两位十进制数字。

反向引用的最简单的、最有用的应用之一，是提供查找文本中两个相同的相邻单词的匹配项的能力。以下面的句子为例：

```
Is is the cost of of gasoline going up up?
```

上面的句子很显然有多个重复的单词。如果能设计一种方法定位该句子，而不必查找每个单词的重复出现，那该有多好。下面的正则表达式使用单个子表达式来实现这一点：

```
/\b([a-z]+) \1\b/gi
```

捕获的表达式，正如 [a-z]+ 指定的，包括一个或多个字母。正则表达式的第二部分是对以前捕获的子匹配项的引用，即，单词的第二个匹配项正好由括号表达式匹配。\1 指定第一个子匹配项。字边界元字符确保只检测整个单词。否则，诸如“is issued”或“this is”之类的词组将不能正确地被此表达式识别。

正则表达式后面的全局标记 (g) 指示，将该表达式应用到输入字符串中能够查找到的尽可能多的匹配。表达式的结尾处的不区分大小写 (i) 标记指定不区分大小写。多行标记指定换行符的两边可能出现潜在的匹配。

使用上面的正则表达式，下面的代码可以使用子匹配项信息，将文本字符串中的两个连续相同的单词的匹配项替换为同一单词的单个匹配项：

```
var ss = "Is is the cost of of gasoline going up up?.\n";
var re = /\b([a-z]+) \1\b/gim;       //Create regular expression pattern.
var rv = ss.replace(re,"$1");   //Replace two occurrences with one.
```

在 **replace** 方法内使用 **$1** 引用第一个保存的子匹配项。如果您有多个子匹配项，您将通过使用 **$2**、**$3** 等依次引用它们。

反向引用还可以将通用资源指示符 (URI) 分解为其组件。假定您想将下面的 URI 分解为协议（ftp、http 等等）、域地址和页/路径：

```
http://msdn.microsoft.com:80/scripting/default.htm
```

下面的正则表达式提供该功能：

```
/(\w+):\/\/([^/:]+)(:\d*)?([^# ]*)/
```

第一个括号子表达式捕获 Web 地址的协议部分。该子表达式匹配在冒号和两个正斜杠前面的任何单词。第二个括号子表达式捕获地址的域地址部分。子表达式匹配 / 或 : 之外的一个或多个字符。第三个括号子表达式捕获端口号（如果指定了的话）。该子表达式匹配冒号后面的零个或多个数字。只能重复一次该子表达式。最后，第四个括号子表达式捕获 Web 地址指定的路径和/或页信息。该子表达式能匹配不包括 # 或空格字符的任何字符序列。

将正则表达式应用到上面的 URI，各子匹配项包含下面的内容：

- **RegExp.$1** 包含“http”
- **RegExp.$2** 包含“msdn.microsoft.com”
- **RegExp.$3** 包含“:80”
- **RegExp.$4** 包含“/scripting/default.htm”

请参见

其他资源

正则表达式介绍

JScript 参考

JScript 编程语言包含各种属性、方法、对象、函数等。另外，JScript 可以利用来自 .NET Framework 类库的许多相应功能。以下各节解释这些功能的正确用法以及 JScript 内的正确语法。

本节内容

功能信息

提供了有关 JScript 中语言功能的版本信息，并将这些功能与 ECMAScript 标准指定的语言功能进行比较。

JScript 语言教程

介绍开发人员用来编写 JScript 代码的元素和过程，并提供链接，指向阐释语言元素和代码语法的详细背景信息的特定区域。

JScript .NET 语言参考

列出 JScript 面向对象的编程语言的基本组件，以及指向解释如何使用该语言的主题的链接。

JScript 编译器选项

列出可用于命令行编译器的选项，并提供链接，指向有关按字母顺序或按类别组织选项的主题。

相关章节

.NET Framework 参考

列出指向特定主题的链接，这些主题解释 .NET Framework 类库和其他基本元素的语法和结构。

功能信息

JScript 编程语言包含各种属性、方法、对象、函数等。另外，JScript 可以利用来自 .NET Framework 类库的许多相应功能。以下各节解释这些功能的正确用法以及 JScript 内的正确语法。

本节内容

Microsoft JScript 功能 - ECMA

标识 JScript 中的语言功能，这些功能是 ECMAScript 语言规范的一部分。

Microsoft JScript 功能 - 非 ECMA

标识 JScript 中的语言功能，这些功能未包括在 ECMAScript 语言规范中。

版本信息

提供不同版本的 JScript 之间的比较。

相关章节

JScript 参考

列出 JScript 语言参考中涉及的元素，并提供指向特定主题的链接，这些主题解释了有关正确使用这些语言元素的幕后详细信息。

Microsoft JScript 功能 - ECMA (JScript)

JScript 包含了 ECMAScript 版本 3 语言规范的几乎所有功能。另外，JScript 正在结合 ECMAScript 版本 4 进行开发，并且也包含了该语言的许多建议功能。下表列出了 ECMAScript 3 的功能以及 JScript 所支持的 ECMAScript 4 建议功能。

JScript 支持的功能

类别	ECMAScript 3 功能/关键字
数组处理	数组 concat、join、length、reverse、slice、sort
赋值	赋值 (=)、加法赋值 (+=)、按位"与"赋值 (&=)、按位"或"赋值 (|=)、按位"异或"赋值 (^=)、除法赋值 (/=)、左移赋值 (<<=)、取模赋值 (%=)、乘法赋值 (*=)、右移赋值 (>>=)、减法赋值 (-=)、无符号右移赋值 (>>>=)
布尔值	Boolean, true, false
注释	/*...*/ 或 //
常数/标识符	NaN null、Infinity undefined
控制流	break continue do...while for for...in if...else Labeled return switch while
日期和时间	Date getDate、getDay、getFullYear、getHours、getMilliseconds、getMinutes、getMonth、getSeconds、getTime、getTimezoneOffset、getYear、getUTCDate、getUTCDay、getUTCFullYear、getUTCHours、getUTCMilliseconds、getUTCMinutes、getUTCMonth、getUTC、setDate、setFullYear、setHours、setMilliseconds、setMinutes、setMonth、setSeconds、setTime、setYear、setUTCDate、setUTCFull、setUTCHours、setUTCMilliseconds、setUTCMinutes、setUTCMonth、setUTCSeconds、toGMTString、toLocaleString、toUTCString、parse、UTC
声明	Function new this var with
错误处理	Error, description, Number, throw, Try...Catch
函数创建	caller, Function arguments, length
全局方法	Global escape, unescape eval isFinite, isNaN parseInt, parseFloat
Math	Math abs、acos、asin、atan、atan2、ceil、cos、exp、floor、log、max、min、pow、random、round、sin、sqrt、tan、E、LN2、LN10、LOG、LOG10E、PI、SQRT1_2、SQRT2
数字	Number MAX_VALUE, MIN_VALUE NaN NEGATIVE_INFINITY, POSITIVE_INFINITY
对象创建	Object new constructor, instanceof, prototype, toString, valueOf
运算符	加法 (+), 减法 (-) 取模算法 (%) 乘法 (*), 除法 (/) 取反 (-) 相等 (==), 不等 (!=) 小于 (<), 小于或等于 (<=) 大于 (>) 大于或等于 (>=) 逻辑"与"(&&), 或 (||), 非 (!) 按位"与" (&), 或 (|), 非 (~), 异或 (^) 按位左移 (<<), 右移 (>>) 无符号右移 (>>>) 条件 (?:) 逗号 (,) delete, typeof, void 递减 (--), 增量 (++),严格相等 (===), 严格不等 (!==)
对象	Array Boolean Date Function Global Math Number Object RegExp Regular Expression String

正则表达式和模式匹配	RegExp index, input, lastIndex, $1...$9, source, compile, exec, test Regular Expression Syntax
字符串	String charAt, charCodeAt, fromCharCode indexOf, lastIndexOf split toLowerCase, toUpperCase lengthconcat, slice match, replace, search anchor, big, blink, bold, fixed, fontcolor, fontsize, italics, link, small, strike, sub, sup

类别	**建议的 ECMAScript 4 功能/关键字**
基于类的对象	class, extends, implements, interface,function get, function set, static,public, private, protected, internal,abstract, final,hide, override,static
声明	const
枚举	枚举

请参见

概念

Microsoft JScript 功能 -- 非 ECMA (JScript)

其他资源

JScript 参考

Microsoft JScript 功能 -- 非 ECMA (JScript)

JScript 包含了 ECMAScript 版本 3 中的几乎所有功能以及为 ECMAScript 版本 4 建议的许多功能。另外，JScript 还拥有许多 ECMAScript 语言未提供的独特功能。下表列出了 JScript 特有的功能。

JScript 特有的功能

类别	功能/关键字
数组处理	VBArray dimensions, getItem, lbound, toArray, ubound
基于类的对象	expando, super
条件编译	@cc_on,@if Statement,@set Statement,@debug,@position,Conditional Compilation Variables
数据类型	boolean, byte, char, decimal,double, float, int, long,Number, sbyte, short, String,uint, ulong, ushort
日期和时间	getVarDate
显示信息	print
枚举	Enumerator atEnd, item, moveFirst, moveNext
Namespaces	package, import
对象	Enumerator VBArray ActiveXObject GetObject
脚本引擎标识	ScriptEngine ScriptEngineBuildVersion ScriptEngineMajorVersion ScriptEngineMinorVersion

请参见

概念

Microsoft JScript 功能 - ECMA (JScript)

其他资源

JScript 参考

JScript 版本信息

JScript 是一种不断演变的语言，每个新版本都引入一些新的功能。若要充分利用该语言的特定版本提供的所有功能，必须有兼容版本的编译器或脚本引擎。

版本和宿主应用程序

当为服务器端应用程序或命令行程序编写代码时，它所支持的编译器版本和 JScript 版本通常是已知的。但是，当编写运行在浏览器脚本引擎中的客户端脚本时，运行的脚本将检测引擎版本。一旦知道了引擎版本，就可以运行使用兼容的 JScript 版本编写的脚本。有关更多信息，请参见检测浏览器功能。

下表列出了宿主应用程序所实现的 Microsoft JScript 版本。

宿主应用程序	1.0	2.0	3.0	4.0	5.0	5.1	5.5	5.6	.NET	8.0
Microsoft Internet Explorer 3.0	x									
Microsoft Internet Information Server 3.0		x								
Microsoft Internet Explorer 4.0			x							
Microsoft Internet Information Server 4.0			x							
Microsoft Internet Explorer 5.0					x					
Microsoft Internet Explorer 5.01						x				
Microsoft Windows 2000						x				
Microsoft Internet Explorer 5.5							x			
Microsoft Windows Millennium Edition							x			
Microsoft Internet Explorer 6.0								x		
Microsoft Windows XP								x		
Microsoft Windows Server 2003								x		
Microsoft .NET Framework 1.0									x	

注意

ScriptEngineMajorVersion 函数和 **@_jscript_version** 条件编译变量所报告的版本号始终是数值。这样就允许对版本号执行数值比较。对于 .NET 版本的应用程序，所报告的版本为 7.x，而不是 .NET。这就意味着，报告版本号为 7.*x* 或更高的引擎可以编译 JScript 8.0 代码。

下表列出了 JScript 语言功能和首次引入时的版本。

语言元素	1.0	2.0	3.0	4.0	5.0	5.5	.NET	8.0
0...n 属性						x		
$1...$9 属性			x					
abs 方法	x							

abstract 修饰符							x	
acos 方法	x							
ActiveXObject 对象			x					
加法运算符 (+)	x							
加法赋值运算符 (+=)	x							
anchor 方法	x							
apply 方法						x		
arguments 对象	x							
arguments 属性		x						
Array 对象		x						
asin 方法	x							
赋值运算符 (=)	x							
atan 方法	x							
atan2 方法	x							
atEnd 方法			x					
big 方法	x							
按位"与"运算符 (&)	x							
按位"与"赋值运算符 (&=)	x							
按位左移运算符 (<<)	x							
按位"取非"运算符 (~)	x							
按位"或"运算符 (|)	x							
按位"或"赋值运算符 (|=)	x							
按位右移运算符 (>>)	x							
按位"异或"运算符 (^)	x							
按位"异或"赋值运算符 (^=)	x							
blink 方法	x							
bold 方法	x							

boolean 数据类型							x	
Boolean 对象		x						
break 语句	x							
byte 数据类型							x	
call 方法						x		
callee 属性						x		
caller 属性		x						
catch 语句					x			
@cc_on 语句			x					
ceil 方法	x							
char 数据类型							x	
charAt 方法	x							
charCodeAt 方法						x		
class 语句							x	
逗号运算符 (,)	x							
//(单行注释语句)	x							
/*..*/(多行注释语句)	x							
比较运算符	x							
compile 方法			x					
concat 方法(数组)			x					
concat 方法(字符串)			x					
条件编译			x					
条件编译变量			x					
条件(三元)运算符 (?:)	x							
const 语句							x	
constructor 属性		x						
continue 语句	x							

cos 方法	x							
数据类型转换			x					
Date 对象	x							
@debug 指令							x	
debugger 语句			x					
decimal 数据类型							x	
decodeURI 方法						x		
decodeURIComponent 方法						x		
减量运算符 (--)	x							
delete 运算符			x					
description 属性					x			
dimensions 方法			x					
除法运算符 (/)	x							
除法赋值运算符 (/=)	x							
do...while 语句			x					
double 数据类型							x	
E 属性	x							
encodeURI 方法						x		
encodeURIComponent 方法						x		
enum 语句							x	
Enumerator 对象			x					
相等运算符 (==)	x							
错误对象					x			
escape 方法	x							
eval 方法	x							
exec 方法			x					

exp 方法	x							
expando 修饰符							x	
false 标识符	x							
final 修饰符							x	
fixed 方法	x							
float 数据类型							x	
floor 方法	x							
fontcolor 方法	x							
fontsize 方法	x							
for 语句	x							
for...in 语句					x			
fromCharCode 方法			x					
function get 语句							x	
Function 对象		x						
function set 语句							x	
function 语句	x							
getDate 方法	x							
getDay 方法	x							
getFullYear 方法			x					
getHours 方法	x							
getItem 方法			x					
getMilliseconds 方法			x					
getMinutes 方法	x							
getMonth 方法	x							
GetObject 函数			x					
getSeconds 方法	x							
getTime 方法	x							

getTimezoneOffset 方法	x							
getUTCDate 方法			x					
getUTCDay 方法			x					
getUTCFullYear 方法			x					
getUTCHours 方法			x					
getUTCMilliseconds 方法			x					
getUTCMinutes 方法			x					
getUTCMonth 方法			x					
getUTCSeconds 方法			x					
getVarDate 方法			x					
getYear 方法	x							
Global 对象			x					
global 属性						x		
大于运算符 (>)	x							
大于或等于运算符 (>=)	x							
hasOwnProperty 方法						x		
hide 修饰符							x	
@if 语句			x					
if...else 语句	x							
ignoreCase 属性						x		
import 语句							x	
in 运算符	x							
增量运算符 (++)	x							
index 属性			x					
indexOf 方法	x							
不等运算符 (!=)	x							
Infinity 属性			x					

input 属性 ($_)			x					
instanceof 运算符					x			
int 数据类型							x	
interface 语句							x	
internal 修饰符							x	
isFinite 方法			x					
isNaN 方法	x							
isPrototypeOf 方法						x		
italics 方法	x							
item 方法			x					
JScript 数据类型							x	
join 方法		x						
Labeled 语句			x					
lastIndex 属性			x					
lastIndexOf 方法	x							
lastMatch 属性 ($&)						x		
lastParen 属性 ($+)						x		
lbound 方法			x					
leftContext 属性 ($`)						x		
左移赋值运算符 (<<=)	x							
length 属性(参数)						x		
length 属性(数组)		x						
length 属性(函数)		x						
length 属性(字符串)	x							
小于运算符 (<)	x							
小于或等于运算符 (<=)	x							
link 方法	x							

LN2 属性	x							
LN10 属性	x							
localeCompare 方法						x		
log 方法	x							
LOG2E 属性	x							
LOG10E 属性	x							
逻辑"与"运算符 (&&)	x							
逻辑"非"运算符 (!)	x							
逻辑或运算符 (||)	x							
long 数据类型							x	
match 方法			x					
Math 对象	x							
max 方法	x							
MAX_VALUE 属性		x						
message 属性						x		
min 方法	x							
MIN_VALUE 属性		x						
取模运算符 (%)	x							
取模赋值运算符 (%=)	x							
moveFirst 方法			x					
moveNext 方法			x					
multiline 属性						x		
乘法运算符 (*)	x							
乘法赋值运算符 (*=)	x							
name 属性						x		
NaN 属性（全局）			x					

NaN 属性(数字)		x						
NEGATIVE_INFINITY 属性		x						
new 运算符	x							
非恒等运算符 (!==)	x							
null 标识符	x							
Number 数据类型							x	
Number 对象		x						
number 属性					x			
Object 对象			x					
运算符优先级	x							
override 修饰符							x	
package 语句							x	
parse 方法	x							
parseFloat 方法	x							
parseInt 方法	x							
PI 属性	x							
pop 方法						x		
@position 指令							x	
POSITIVE_INFINITY 属性		x						
pow 方法	x							
print 语句							x	
private 修饰符							x	
propertyIsEnumerable 属性						x		
protected 修饰符							x	
prototype 属性		x						
public 修饰符							x	
push 方法						x		

random 方法	x							
RegExp 对象			x					
正则表达式对象			x					
正则表达式语法			x					
replace 方法	x							
return 语句	x							
reverse 方法		x						
rightContext 属性 ($')						x		
右移赋值运算符 (>>=)	x							
round 方法	x							
sbyte 数据类型							x	
ScriptEngine 函数		x						
ScriptEngineBuildVersion 函数		x						
ScriptEngineMajorVersion 函数		x						
ScriptEngineMinorVersion 函数		x						
search 方法			x					
@set 语句			x					
setDate 方法	x							
setFullYear 方法			x					
setHours 方法	x							
setMilliseconds 方法			x					
setMinutes 方法	x							
setMonth 方法	x							
setSeconds 方法	x							
setTime 方法	x							
setUTCDate 方法			x					
setUTCFullYear 方法			x					

setUTCHours 方法			x					
setUTCMilliseconds 方法			x					
setUTCMinutes 方法			x					
setUTCMonth 方法			x					
setUTCSeconds 方法			x					
setYear 方法	x							
shift 方法						x		
short 数据类型							x	
sin 方法	x							
slice 方法(数组)			x					
slice 方法(字符串)			x					
small 方法	x							
sort 方法		x						
source 属性			x					
splice 方法						x		
split 方法			x					
sqrt 方法	x							
SQRT1_2 属性	x							
SQRT2 属性	x							
static 修饰符							x	
static 语句							x	
严格相等运算符 (===)	x							
strike 方法	x							
String 数据类型							x	
String 对象	x							
sub 方法	x							
substr 方法			x					

substring 方法	x							
减法运算符 (-)	x							
减法赋值运算符 (-=)	x							
sup 方法	x							
super 语句							x	
switch 语句			x					
tan 方法	x							
test 方法			x					
this 语句	x							
throw 语句					x			
toArray 方法			x					
toDateString 方法						x		
toExponential 方法						x		
toFixed 方法						x		
toGMTString 方法	x							
toLocaleDateString 方法						x		
toLocaleLowerCase 方法						x		
toLocaleString 方法	x							
toLocaleTimeString 方法						x		
toLocaleUpperCase 方法						x		
toLowerCase 方法	x							
toPrecision 方法						x		
toString 方法		x						
toTimeString 方法						x		
toUpperCase 方法	x							
toUTCString 方法			x					

true 标识符	x							
try...catch...finally 语句					x			
类型批注							x	
类型转换							x	
typeof 运算符	x							
ubound 方法			x					
uint 数据类型							x	
ulong 数据类型							x	
一元求非运算符 (-)	x							
undefined 属性						x		
unescape 方法	x							
unshift 方法						x		
无符号右移运算符 (>>>)	x							
无符号右移赋值运算符 (>>>=)	x							
ushort 数据类型							x	
UTC 方法	x							
valueOf 方法		x						
var 语句	x							
VBArray 对象			x					
void 运算符		x						
while 语句	x							
with 语句	x							

请参见

概念

JScript 8.0 的新增功能

其他资源

JScript 参考

JScript 语言教程

像许多其他编程语言一样，Microsoft JScript 脚本或程序以文本格式编写。通常，脚本或程序由许多语句和注释组成。在一个语句内，可以使用变量、表达式和标识符 (literal) 数据，比如字符串和数字。

本节内容

JScript 数组

解释数组的类型以及如何在 JScript 中使用它们。

JScript 赋值与相等

解释 JScript 如何将值分配给变量、数组元素和属性元素，并且解释 JScript 所使用的等式语法。

JScript 注释

阐释如何使用正确的 JScript 语法在代码中包含注释。

JScript 表达式

解释如何组合关键字、运算符、变量和标识符以产生新值。

JScript 标识符

解释如何创建 JScript 中标识符的有效名称。

JScript 语句

提供有关 JScript 中的两个基本指令单元（声明语句和可执行语句）的概述。

JScript 数据类型

包含指向特定主题的链接，这些主题解释如何使用 JScript 中的基元数据类型、引用数据类型和 .NET Framework 数据类型。

JScript 变量和常数

列出指向特定主题的链接，这些主题解释如何声明变量和常数以及如何使用它们引用对象。

JScript 对象

提供对象的概述并列出解释如何在 JScript 中创建和使用对象的链接。

JScript 修饰符

解释如何使用可见性修饰符、继承修饰符、版本安全修饰符、expando 修饰符和静态修饰符。

JScript 运算符

列出计算、逻辑、按位、赋值和杂项运算符并提供指向特定信息的链接，这些信息解释如何有效地使用这些运算符。

JScript 函数

描述函数的概念并提供指向特定主题的链接，这些主题解释如何使用和创建函数。

JScript 中的强制

解释 JScript 编译器如何针对不同数据类型的值执行操作。

复制、传递和比较数据

解释 JScript 如何在处理数组、函数、对象等内容时复制、传递和比较数据。

JScript 条件结构

描述 JScript 通常如何处理程序流程并提供指向特定信息的链接，这些信息解释如何控制程序的执行流程。

JScript 保留字 (JScript)

解释保留字的概念并列出 JScript 中的保留字。

JScript 的安全注意事项

解释如何避免 JScript 代码中的常见安全问题。

相关章节

语言参考

列出 JScript 语言参考中涉及的元素，并提供指向特定主题的链接，这些主题解释了有关正确使用这些语言元素的幕后详细信息。

.NET Framework 类库参考

包含指向特定主题的链接，这些主题解释 .NET Framework 类库中的类以及如何使用类库文档。

JScript 数组

数组将相关的数据项组合到一个变量中。与共享变量名合在一起的唯一编号（称为"索引"或"下标"）区分数组中的每个元素。在很多情况下，数组可使代码更短、更简单，这是因为通过使用索引号，循环可以有效地处理任意数量的元素。

JScript 提供了两种不同类型的数组，即 JScript 数组对象和类型化数组。在 JScript 数组对象（为稀疏数组）中，脚本可以动态地添加和移除元素，并且元素可以是任意的数据类型。在类型化数组（为稠密数组）中，数组大小是固定的，元素的类型必须与数组的基类型相同。

本节内容

数组概述

描述两种类型的 JScript 数组、这两种数组类型之间的差异以及如何选择适当的数组类型。

数组声明

解释声明数组的概念，以及使用 **new** 运算符声明数组与声明数组常值之间的区别。

数组使用

阐释如何访问一维数组、多维数组和嵌套数组中的元素。

嵌套数组

解释嵌套数组的概念、用途和使用方法。

多维数组

解释多维数组的概念、多维数组与嵌套数组之间的差别以及多维数组的使用方法。

相关章节

数组数据

阐释如何构成数组常值，以及如何在同一数组中组合多个数据类型。

Array 对象

描述 JScript **Array** 对象、该对象的使用方法以及该对象与 **System.Array** 数据类型之间如何交互操作的参考信息。

数组概述

数组将多个数据收集到一个变量中。单个索引号（用于一维数组）或者多个索引号（用于嵌套的数组或多维数组）引用数组中的数据。若要引用数组中的单个元素，可以使用后面跟数组索引（数组索引放在方括号 (**[]**) 中）的数组标识符。若要引用整个数组，则只需使用数组标识符。将数据收集到数组中可简化数据管理。例如，通过使用数组，方法只使用一个参数就可以将一组名称传递给函数。

JScript 中有两种类型的数组，即 JScript 数组和类型化数组。虽然这两种类型的数组很相似，但是它们之间也有一些差异。JScript 数组和类型化数组之间可以进行交互操作。因此，JScript **Array** 对象可以调用任何类型化数组的方法和属性，而类型化数组也可以调用 **Array** 对象的很多方法和属性。而且，接受类型化数组的函数也接受 **Array** 对象，反之亦然。有关更多信息，请参见 Array 对象。

类型化数组

类型化数组（也称为“本机数组”）与 C 和 C++ 等语言中使用的数组类似。类型化数组通过仅存储与数组类型声明所指定的类型对应的数据来提供类型安全。

注意
您可以定义类型为 **Object** 的类型化数组来存储任何类型的数据。

当脚本创建或初始化数组时，它就会设置类型化数组中元素的数量。更改元素数量的唯一方法是重新创建数组。对于创建的包含 *n* 个元素的类型化数组而言，其元素编号为 0 到 *n*-1。访问此范围外的元素就会产生错误。此外，类型化数组是“*稠密的*”，即在允许的范围内的每个索引都引用一个元素。

脚本可以将声明的类型化数组赋给变量或常数，或者将该数组传递给函数、运算符或语句。当将数组赋给变量（或常数）时，请确保变量的数据类型与数组的类型相匹配，并且数组的维数也匹配。

类型化数组是 .NET Framework **System.Array** 对象的一个实例。要访问 **System.Array** 对象的静态成员，或显式创建 **System.Array** 对象需要完全限定名称 **System.Array**。该语法将它与 **Array**（内部 **JScript** 对象）区分开来。

JScript 数组

JScript **Array** 对象提供比类型化数组更大的灵活性，当需要一般堆栈、需要项列表以及性能不是最关键因素时，使用这种对象是非常方便的。但是，由于类型化数组提供类型安全、改善的性能并且能与其他语言更好地进行交互，所以开发人员通常选择类型化数组，而不选择 JScript 数组。

JScript 数组可以存储任何类型的数据，因此可以方便而快捷地编写使用数组的脚本，而不必考虑类型冲突。因为它跳过了 JScript 提供的强类型检查，所以要慎用这种功能。

脚本可以动态地向 JScript 数组添加元素，也可以动态地移除 JScript 数组中的元素。若要添加数组元素，请给该元素赋一个值。**delete** 运算符可以移除元素。

JScript 数组是“*稀疏的*”。即，如果数组有 3 个元素，编号分别为 0、1 和 2，这时可以存在元素 50，而元素 3 到 49 不存在。每个 JScript 数组都具有 **length** 属性，当添加元素时，就会自动更新该属性。在前一个示例中，添加元素 50 就会使 length 变量的值更改为 51 而不是 4。

JScript **Array** 对象和 JScript **Object** 几乎是相同的。它们之间的两个主要差异是：**Object**（默认情况下）没有自动长度属性；而 JScript **Object** 没有 **Array** 的属性和方法。有关更多信息，请参见 JScript Array 对象。

请参见

参考

Array 对象

概念

数组数据

JScript Array 对象

其他资源

JScript 数组

数组声明

变量可以存储数组，与存储 JScript 中的所有其他数据一样。类型批注可以指定变量必须包含一个数组对象或者类型化数组，但是它不提供初始数组。若要在变量中存储数组，必须"*声明*"一个数组并将它赋给该变量。

声明 JScript 数组对象就会创建一个新的 **Array** 对象；而声明类型化数组则会保留一段足以存储每个数组元素的内存。这两种类型的数组都可以通过以下两种方式来声明：使用 **new** 运算符显式构造一个新的数组或者使用数组常值。

使用 new 运算符声明数组

若要声明新的 JScript **Array** 对象，可以将 **new** 运算符和 **Array** 构造函数一起使用。因为可以将成员动态地添加到 JScript 数组中，所以无须指定数组的初始大小。在本例中，将长度为零的数组赋给 `a1`。

```
var a1 = new Array();
```

若要给用 **Array** 构造函数创建的数组赋予初始长度，请将一个整数传递给数组构造函数。数组长度必须为零或正数。以下代码将长度为 10 的数组赋给 `a2`。

```
var a2 = new Array(10);
```

如果将多个参数或单个非数值参数传递给 **Array** 构造函数，则生成的数组将所有的参数包含为数组元素。例如，以下代码创建一个数组，其中元素 0 是数字 10，元素 1 是字符串"Hello"，而元素 2 则是当前日期。

```
var a3 = new Array(10, "Hello", Date());
```

new 运算符也可以声明类型化数组。因为类型化数组不能接受动态添加的元素，所以声明必须指定数组的大小。类型化数组的构造函数在数组大小的两边使用方括号，而不是使用圆括号。例如，以下代码声明包含 5 个整数的数组。

```
var i1 : int[] = new int[5];
```

new 运算符也可以声明多维数组。下面的示例声明一个 3×4×5 的整数数组。

```
var i2 : int[,,] = new int[3,4,5];
```

当声明嵌套的数组时，必须先声明基数组，然后声明子数组；不能同时声明这两种数组。这为确定子数组大小提供了更大的灵活性。在本例中，第一个子数组的长度为 1，第二个子数组的长度为 2，依此类推。

```
// First, declare a typed array of type int[], and initialize it.
var i3 : int[][] = new (int[])[4];
// Second, initialize the subarrays.
for(var i=0; i<4; i++)
   i3[i] = new int[i+1];
```

使用数组常值声明数组

另一种同时声明和初始化数组的方法是使用数组常值。数组常值表示一个 JScript **Array**。因为 JScript 数组与类型化数组交互操作，所以也可以使用常值初始化类型化数组。有关更多信息，请参见数组数据。

数组常值可以方便地初始化一维数组。请注意，当给类型化数组赋值时，编译器尝试将数组常值中的数据转换为正确的类型。在本例中，将数组常值赋给 JScript 数组和类型化数组。

```
var al1 : Array = [1,2,"3"];
var il1 : int[] = [1,2,"3"];
```

数组常值也可以初始化嵌套的数组。在下面的示例中，用包含两个整数数组的数组初始化 JScript 数组和类型化数组。

```
var al1 : Array = [[1,2,3],[4,5,6]];
var il1 : int[][] = [[1,2,3],[4,5,6]];
```

数组常值不能初始化多维类型化数组。

请参见

参考

new 运算符

概念

数组数据

JScript Array 对象

其他资源

JScript 数组

数组使用

JScript 中可使用一些类型数组。以下信息解释如何使用其中的某些数组，以及如何为特定的应用程序选择适当的数组。

一维数组

下面的示例显示如何访问 `addressBook` 数组的第一个和最后一个元素。它假定脚本的另一部分定义 `addressBook` 并给其赋值。因为在 JScript 中数组的索引是从零开始的，所以数组的第一个元素的索引是零，而最后一个元素的索引是数组的长度减 1。

```
var firstAddress = addressBook[0];
var lastAddress = addressBook[addressBook.length-1];
```

嵌套的数组与多维数组

可以将由多个索引引用的数据存储在嵌套的数组或多维数组中。这两种类型的数组都具有独特的功能。

嵌套的数组可用于在其中每一子数组具有不同长度的应用程序。可以方便地对子数组进行重组，这对数组元素的排序很有帮助。日历就是一个典型的例子，其中 `Year` 数组存储 12 个 `Month` 数组，而每个 `Month` 数组存储适当天数的数据。

多维数组用于以下场合：声明时每个维中的数组的大小是已知的。从速度和内存占用情况来看，多维数组比嵌套数组的效率更高。多维数组必须是类型化数组。用于数学计算的矩阵就是一个典型的例子，其中数组大小是固定的并且从一开始便是已知的。

循环处理 JScript 数组元素

因为 JScript 数组是稀疏的，所以数组的第一个元素和最后一个元素之间可能有大量未定义的元素。这意味着，如果使用 **for** 循环访问数组元素，则必须检查每个元素是否为 **undefined**（未定义）。

所幸，JScript 提供了 **for...in** 循环，可以方便地使用此循环只访问 JScript 数组中已定义的元素。下面的示例定义了一个稀疏 JScript 数组，并使用 **for** 循环和 **for...in** 循环显示其元素。

```
var a : Array = new Array;
a[5] = "first element";
a[100] = "middle element";
a[100000] = "last element";
print("Using a for loop. This is very inefficient.")
for(var i = 0; i<a.length; i++)
   if(a[i]!=undefined)
      print("a[" + i + "] = " + a[i]);
print("Using a for...in loop. This is much more efficient.");
for(var i in  a)
   print("a[" + i + "] = " + a[i]);
```

该程序的输出为：

```
Using a for loop. This is very inefficient.
a[5] = first element
a[100] = middle element
a[100000] = last element
Using a for...in loop. This is much more efficient.
a[5] = first element
a[100] = middle element
a[100000] = last element
```

请参见

参考

for...in 语句

概念

嵌套数组

多维数组 (Jscript)

其他资源

JScript 数组

嵌套数组

可以创建一个数组，并使用其他数组填充该数组。基数组可以是 JScript 数组或类型化数组。JScript 数组在存储数据类型方面具有更大的灵活性；而类型化数组则防止在数组中存储类型不适当的数据。

嵌套的数组可用于在其中每一子数组具有不同长度的应用程序。如果每个子数组都具有相同的长度，则多维数组更适用。有关更多信息，请参见多维数组。

类型化的嵌套数组

在下面的示例中，嵌套的字符串数组存储宠物的名字。因为每个子数组中元素的数量与其他数组无关（猫名字的数量可能与狗名字的数量不同），所以使用嵌套的数组而不使用多维数组。

```
// Create two arrays, one for cats and one for dogs.
// The first element of each array identifies the species of pet.
var cats : String[] = ["Cat","Beansprout", "Pumpkin", "Max"];
var dogs : String[] = ["Dog","Oly","Sib"];

// Create a typed array of String arrays, and initialze it.
var pets : String[][] = [cats, dogs];

// Loop over all the types of pets.
for(var i=0; i<pets.length; i++)
   // Loop over the pet names, but skip the first element of the list.
   // The first element identifies the species.
   for(var j=1; j<pets[i].length; j++)
      print(pets[i][0]+": "+pets[i][j]);
```

该程序的输出为：

```
Cat: Beansprout
Cat: Pumpkin
Cat: Max
Dog: Oly
Dog: Sib
```

也可以使用类型为 **Object** 的类型化数组来存储数组。

JScript 的嵌套数组

将 JScript 数组用作基数组，可在存储的子数组类型方面提供更大的灵活性。例如，以下代码创建一个 JScript 数组，该数组存储包含字符串和整数的 JScript 数组。

```
// Declare and initialize the JScript array of arrays.
var timecard : Array;
timecard = [ ["Monday", 8],
             ["Tuesday", 8],
             ["Wednesday", 7],
             ["Thursday", 9],
             ["Friday", 8] ];
// Display the contents of timecard.
for(var i=0; i<timecard.length; i++)
   print("Worked " + timecard[i][1] + " hours on " + timecard[i][0] + ".");
```

上面的示例显示：

```
Worked 8 hours on Monday.
Worked 8 hours on Tuesday.
Worked 7 hours on Wednesday.
Worked 9 hours on Thursday.
Worked 8 hours on Friday.
```

请参见

概念

数组数据

多维数组 (Jscript)

其他资源

JScript 数组

多维数组 (Jscript)

可以在 JScript 中创建多维类型化数组。多维数组使用多个索引来访问数据。当脚本声明数组时，它设定每个索引的范围。多维数组与嵌套的数组类似，但嵌套的数组中每个子数组可以具有不同的长度。有关更多信息，请参见嵌套数组。

讨论

一维数组的数据类型由后面带一对方括号 (**[]**) 的数据类型名称指定。用同一方法也可以指定多维数组的数据类型，但要在括号内使用逗号 (,)。数组的维数等于逗号的个数加 1。下面的示例阐释了定义一维数组和多维数组之间的区别。

```
// Define a one-dimensional array of integers. No commas are used.
var oneDim : int[];
// Define a three-dimensional array of integers.
// Two commas are used to produce a three dimensional array.
var threeDim : int[,,];
```

在下面的示例中，使用二维字符数组来存储 Tic-Tac-Toe（“零和叉”棋盘游戏）棋盘的状态。

```
// Declare a variable to store two-dimensional game board.
var gameboard : char[,];
// Create a three-by-three array.
gameboard = new char[3,3];
// Initialize the board.
for(var i=0; i<3; i++)
   for(var j=0; j<3; j++)
      gameboard[i,j] = " ";
// Simulate a game. 'X' goes first.
gameboard[1,1] = "X"; // center
gameboard[0,0] = "O"; // upper-left
gameboard[1,0] = "X"; // center-left
gameboard[2,2] = "O"; // lower-right
gameboard[1,2] = "X"; // center-right, 'X" wins!
// Display the board.
var str : String;
for(var i=0; i<3; i++) {
   str = "";
   for(var j=0; j<3; j++) {
      if(j!=0) str += "|";
      str += gameboard[i,j];
   }
   if(i!=0)
      print("-+-+-");
   print(str);
}
```

该程序的输出为：

```
O| |
-+-+-
X|X|X
-+-+-
 | |O
```

可以使用类型为 **Object** 的多维类型化数组来存储任何类型的数据。

请参见

概念

数组数据

嵌套数组

其他资源

JScript 数组

JScript 赋值与相等

在 JScript 中，赋值运算符给变量赋值。相等运算符比较两个值。

赋值

像许多编程语言一样，JScript 使用等号 (=) 给变量赋值：它是赋值运算符。= 运算符的左操作数必须是一个 *Lvalue*，这表示它必须是变量、数组元素或对象属性。

= 运算符的右操作数必须是一个 *Rvalue*。*Rvalue* 可以是任何类型的任意值，包括作为表达式结果的值。下面是 JScript 赋值语句的一个示例。

```
anInteger = 3;
```

JScript 将该语句解释为：

将值 3 赋给变量 `anInteger`，

或

"`anInteger` 得到值 3。

如果类型批注没有将语句中的变量绑定到特定的数据类型，则赋值将始终是成功的。否则，编译器将试图把 *Lvalue* 转换为 *Rvalue* 的数据类型。如果此转换始终不能成功执行，则编译器将生成错误。如果此转换对于某些值可成功执行，对于另一些值则会失败，那么编译器将生成警告，说明当运行代码时转换可能失败。

在本例中，当赋值给变量 `x` 时，存储在变量 `i` 中的整数值将转换成一个双精度值。

```
var i : int = 29;
var x : double = i;
```

有关更多信息，请参见类型转换。

相等

与其他一些编程语言不同，JScript 不使用等号作为比较运算符，而只作为赋值运算符。对于两个值的比较，可以使用相等运算符 (==) 或全等运算符 (===)。

相等运算符按值来比较基元字符串、数字和布尔值。如有必要进行类型转换之后，若两个变量具有相同的值，相等运算符将返回 **true**。对象（包括 **Array**、**Function**、**String**、**Number**、**Boolean**、**Error**、**Date** 和 **RegExp** 对象）是根据引用来进行比较的。即使两个对象变量具有相同的值，也只有当它们引用相同的对象时，比较结果才会返回 **true**。

全等运算符同时比较两个表达式的值和类型；只有当两个表达式通过相等运算符比较为相等并且两个操作数的数据类型相同时，才会返回 **true**。

注意

全等运算符不区分不同的数值数据类型。一定要确保理解赋值运算符、相等运算符和全等运算符之间的差异。

用户脚本中的比较始终有一个布尔值结果。请看下面的 JScript 代码行。

```
y = (x == 2000);
```

此处，测试变量 `x` 的值，看它是否等于数字 2000。如果是，则比较结果为布尔值 **true**，该值将赋给变量 `y`。如果 `x` 不等于 2000，则比较结果为布尔值 **false**，赋给 `y`。

相等运算符将进行类型转换，以检查值是否相同。在下面的 JScript 代码行中，字符串 "42" 将在与数字 42 比较之前被转换为一个数字。其结果为 **true**。

```
42 == "42";
```

对象使用不同的规则进行比较。相等运算符的行为取决于对象的类型。如果对象是用相等运算符定义的某个类的实例，返回值

将取决于相等运算符的实现。提供相等运算符的类不能在 JScript 中定义，尽管其他 .NET Framework 语言允许这样的类定义。

对于不具有已定义相等运算符的对象（比如基于 JScript **Object** 对象的对象或 JScript 类的实例），只有当两个对象都引用相同的对象时，比较结果才相等。这就意味着包含相同数据的两个不同对象的比较结果不同。下面的示例阐释了这一点。

```
// A primitive string.
var string1 = "Hello";
// Two distinct String objects with the same value.
var StringObject1 = new String(string1);
var StringObject2 = new String(string1);

// An object converts to a primitive when
// comparing an object and a primitive.
print(string1 == StringObject1);   // Prints true.

// Two distinct objects compare as different.
print(StringObject1 == StringObject2);   // Prints false.

// Use the toString() or valueOf() methods to compare object values.
print(StringObject1.valueOf() == StringObject2);   // Prints true.
```

在控制结构的条件语句中，相等运算符是非常有用的。您在这里组合相等运算符与使用它的语句。请看下面的 JScript 代码示例。

```
if (x == 2000)
   z = z + 1;
else
   x = x + 1;
```

如果 `x` 的值是 2000，则 JScript 中的 **if...else** 语句执行一个操作（本例中执行 `z = z + 1`），如果 `x` 的值不是 2000，则执行另一个操作 (`x = x + 1`)。有关更多信息，请参见 JScript 条件结构。

全等运算符 (===) 仅对数值数据类型执行类型转换。这意味着整数 42 被视为与双精度 42 相同，而这两者都与字符串"42"不同。下面的 JScript 代码说明了这一点。

```
var a : int = 42;
var b : double = 42.00;
var c : String = "42";
print(a===b); // Displays "true".
print(a===c); // Displays "false".
print(b===c); // Displays "false".
```

请参见

概念

布尔型数据

类型转换

其他资源

JScript 语言教程

JScript 条件结构

JScript 注释

单行 JScript 注释以两个正斜杠 (//) 开始。

代码中的注释

以下是单行注释（后跟一行代码）的一个示例。

```
// This is a single-line comment.
aGoodIdea = "Comment your code for clarity.";
```

多行 JScript 注释以正斜杠和星号 (/*) 开头，以相反的顺序 (*/) 结束。

```
/*
This is a multiline comment that explains the preceding code statement.
The statement assigns a value to the aGoodIdea variable. The value,
which is contained between the quote marks, is called a literal. A
literal explicitly and directly contains information; it does not
refer to the information indirectly. The quote marks are not part
of the literal.
*/
```

如果试图在一个多行注释中嵌入另一个多行注释，JScript 将以一种意想不到的方式解释生成的多行注释。标记嵌入的多行注释结尾的 */ 将被解释为整个多行注释的结尾。因此，在嵌入的多行注释后面的文本将被解释为 JScript 代码，并可能生成语法错误。

在下面的示例中，由于 JScript 将最里面的 */ 解释为最外面注释的结尾，因此第三行文本将被解释为 JScript 代码：

```
/* This is the outer-most comment
/* And this is the inner-most comment */
...Unfortunately, JScript will try to treat all of this as code. */
```

建议将所有注释编写为单行注释的块。这样就允许随后用一个多行注释来注释大段代码。

```
// This is another multiline comment, written as a series of single-line comments.
// After the statement is executed, you can refer to the content of the aGoodIdea
// variable by using its name, as in the next statement, in which a string literal is
// appended to the aGoodIdea variable by concatenation to create a new variable.
var extendedIdea = aGoodIdea + " You never know when you'll have to figure out what it does
.";
```

或者还可以使用条件编译安全有效地注释大段代码。

请参见

其他资源

JScript 参考
JScript 语言教程
条件编译

JScript 表达式

JScript 表达式是关键字、运算符、变量和标识符的组合形式，用来产生值。表达式可执行计算、操作数据、调用函数、测试数据或执行其他操作。

使用表达式

最简单的表达式是标识符。此处是 JScript 标识符表达式的一些示例。有关更多信息，请参见 JScript 中的数据。

```
3.9                              // numeric literal
"Hello!"                         // string literal
false                            // Boolean literal
null                             // literal null value
[1,2,3]                          // Array literal
var o = {x:1, y:2}               // Object literal
var f = function(x){return x*x;} // function literal
```

更复杂的表达式可包含变量、函数调用和其他表达式。可以使用运算符组合表达式并创建复杂的表达式。以下是使用运算符的示例：

```
4 + 5        // additon
x += 1       // addition assignment
10 / 2       // division
a & b        // bitwise AND
```

此处是 JScript 复杂表达式的一些示例。

```
radius = 10;
anExpression = 3 * (4 / 5) + 6;
aSecondExpression = Math.PI * radius * radius;
aThirdExpression = "Area is " + aSecondExpression + ".";
myArray = new Array("hello", Math.PI, 42);
myPi = myArray[1];
```

请参见

其他资源

JScript 参考
JScript 语言教程
JScript 运算符
JScript 变量和常数

JScript 标识符

在 JScript 中，标识符用于：

- 命名变量、常数、函数、类、接口和枚举
- 为循环提供标签（不常用）。

使用标识符

JScript 是一种区分大小写的语言。因此，名为 `myCounter` 的变量与名为 `MyCounter` 的变量不同。变量名可以是任意长度。创建有效变量名的规则如下：

- 第一个字符必须是 Unicode 字母（大写或小写）或下划线 (_) 字符。注意，数字不能用作第一个字符。
- 随后的字符必须是字母、数字或下划线。
- 变量名一定不能是保留字。

下面是有效标识符的一些示例：

```
_pagecount
Part9
Number_Items
```

下面是无效标识符的一些示例：

```
99Balloons // Cannot begin with a number.
Smith&Wesson // The ampersand (&) character is not a valid character for variable names.
```

在选择标识符时，应避免选择 JScript 保留字和已经是内部 JScript 对象或函数的名称，比如 **String** 或 **parseInt**。

请参见

概念

JScript 保留字 (JScript)

其他资源

JScript 变量和常数

JScript 函数

JScript 对象

JScript 语句

JScript 程序是语句的集合。JScript 语句(等效于自然语言中的完整句子)将可执行一个完整任务的表达式组合在一起。

使用语句

一个语句由一个或多个表达式、关键字或运算符(符号)组成。虽然可在同一行显示两个或更多个语句(用分号分隔),但通常是一行只包含一个语句。另外,大多数语句可以跨多个行。例外情况包括:

- 后缀递增和递减运算符必须与它们的参数显示在同一行。例如,`x++` 和 `i--`。
- 关键字 **continue** 和 **break** 必须与它们的标签显示在同一行。例如,`continue label1` 和 `break label2`。
- 关键字 **return** 和 **throw** 必须与它们的表达式显示在同一行。例如,`return (x+y)` 和 `throw "Error 42"`。
- 除非自定义属性前面带有修饰符,否则它必须与它要修饰的声明显示在同一行。例如 `myattribute class myClass`。

虽然不要求在行尾显式终止语句,但为了清楚起见,这里提供的大多数 JScript 示例都被显式终止。这是通过分号 (;) 完成的,分号是 JScript 语句的终止字符。这里有 JScript 语句的两个示例。

```
var aBird = "Robin"; // Assign the text "Robin" to the variable aBird.
var today = new Date(); // Assign today's date to the variable today.
```

由括号 ({}) 包围的一组 JScript 语句称为一个块。块中的语句通常可以视为一个语句。这就意味着可以在 JScript 要求使用单个语句的大多数地方使用块。需要引起注意的例外情况包括 **for** 和 **while** 循环的头。下面的示例阐释了典型的 **for** 循环:

```
var i : int = 0;
var x : double = 2;
var a = new Array(4);
for (i = 0; i < 4; i++) {
   x *= x;
   a[i] = x;
}
```

注意,块中的各个语句以分号结束,但块本身不是这样。

通常,函数、条件和类使用块。注意,与 C++ 和大多数其他语言不同,JScript 并不将块视为一个新范围;只有函数、类、静态初始值设定项和 catch 块创建新范围。

在下面的示例中,第一个语句开始定义一个函数,此函数由三个语句的 **if...else** 序列组成。在此块的后面有一个语句没有包含在函数块的括号内。因此,最后那一个语句不是函数定义的一部分。

```
function FeetToMiles(feet, cnvType) {
   if (cnvType == "NM")
      return( (feet / 6080) + " nautical miles");
   else if (cnvType == "M")
      return( (feet / 5280) + " statute miles");
   else
      return ("Invalid unit of measure");
}
var mradius = FeetToMiles(52800, "M");
```

请参见

参考

class 语句
function 语句
if...else 语句
static 语句

其他资源

JScript 参考
JScript 语言教程

JScript 数据类型

JScript 提供了 13 种基元数据类型和 13 种引用数据类型。此外，还可以声明新的数据类型或使用符合公共语言规范 (CLS) 的 .NET Framework 数据类型。本节包括有关内部数据类型的信息，并介绍如何扩展这些类型，如何定义自己的数据类型，如何输入数据，以及如何将数据从一种类型转换为另一种类型。

本节内容

JScript 中的数据

指向特定主题的链接，这些主题解释如何输入数组、布尔值、数值、字符串和对象数据。

数据类型摘要

包括一个表，其中列出了 JScript 中的基元和引用数据类型以及 .NET Framework 中相应的类。

用户定义的数据类型

解释如何使用类语句来定义新的数据类型。

类型化数组

阐释如何定义、初始化和使用类型化数组。

类型转换

解释类型转换的概念以及隐式和显式转换的不同。

相关章节

JScript 对象

提供对象的概述并列出解释如何在 JScript 中创建和使用对象的链接。

数据类型

列出指向特定参考主题的链接，这些参考主题解释内部数据类型及其关联属性和方法。

JScript 中的数据

像大多数语言一样，JScript 使用了几种基本类型的数据。其中有数值数据和字符串数据。字符串是一个文本块。在 JScript 程序中有几种方式可以输入这种数据。数据经常是通过标识符表达式输入的。

本节内容

数组数据

解释 JScript 中数组的概念以及如何在脚本中使用数组标识符输入数组数据。

布尔型数据

解释布尔型数据的概念以及如何在 JScript 代码中使用它的两个标识符值。

数值数据

描述整型数据与浮点数据之间的差异以及如何在脚本中输入数值数据。

字符串数据

解释字符串数据的概念、它的语法以及转义符的使用。

对象数据

描述 Jscript 中对象数据的概念、它的初始化和使用。

相关章节

JScript 数据类型

包含指向特定主题的链接，这些主题解释如何使用 JScript 中的基元数据类型、引用数据类型和 .NET Framework 数据类型。

数据类型

列出指向特定参考主题的链接，这些参考主题解释内部数据类型及其关联属性和方法。

数组数据

在 JScript 中，数组标识符 (literal) 可初始化一个数组。数组标识符（表示 JScript **Array** 对象）用一个以逗号分隔的列表表示，该列表由一对方括号 ([]) 包围。该列表的每个元素既可以是有效的 JScript 表达式，也可以为空（两个相连的逗号）。数组标识符列表中的第一个元素的索引号为零；列表中随后的每个元素都对应于数组中随后的元素。JScript **Array** 是稀疏排列的；如果数组标识符列表中的某个元素为空，则不初始化 JScript **Array** 中的对应元素。

使用数组数据

在本例中，变量 `arr` 被初始化为具有三个元素的数组。

```
var arr = [1,2,3];
```

可以使用 **Array** 标识符列表中的空元素创建稀疏数组。例如，下面的"数组"标识符表示一个只定义元素 0 和 4 的数组。

```
var arr = [1,,,,5];
```

数组标识符可包含任意类型的数据，包括其他数组。在以下数组的数组中，第二个子数组既有字符串又有数值数据。

```
var cats = [ ["Names", "Beansprout", "Pumpkin", "Max"], ["Ages", 6, 5, 4] ];
```

由于 JScript **Array** 对象与类型化数组相互作用，因此数组标识符在一些限制条件下也可以初始化类型化数组。数组标识符中的数据必须可转换为类型化数组的数据类型。数组标识符不能初始化多维类型化数组，但可以初始化类型化数组的类型化数组。当数组标识符初始化类型化数组时，需要两个步骤。首先，将数组标识符转换为类型化数组，后者用于初始化类型化数组。作为转换的一部分，数组标识符的每个空元素首先都被解释为 **undefined**，然后标识符的每个元素都被转换为类型化数组的相应数据类型。在下面的示例中，使用相同的数组标识符初始化一个 JScript 数组、一个整数数组和一个双精度数组。

```
var arr = [1,,3];
var arrI : int[] = [1,,3];
var arrD : double[] = [1,,3];
print(arr);   // Displays  1,,3.
print(arrI);  // Displays  1,0,3.
print(arrD);  // Displays  1,NaN,3.
```

数组标识符的空元素在整数数组中表示为 0，在双精度数组中表示为 **NaN**，这是由于 **undefined** 映射到这些值的缘故。

请参见

参考

Array 对象

概念

JScript 表达式

类型转换

其他资源

JScript 中的数据

数据类型 (JScript)

JScript 数组

内部对象

布尔型数据

虽然数值和字符串数据类型实际上可以有无限多个不同的值，但 **boolean** 数据类型只能有两个值。它们是标识符 **true** 和 **false**。布尔值表示条件的有效性（告知条件是真还是假）。

使用布尔值

可以使用布尔值（**true** 或 **false**）作为控制结构中的条件语句。例如，可以使用 **true** 作为 **while** 语句的条件，创建一个潜在的无限循环。

```
var s1 : String = "Sam W.";
var s2 : String = "";
while (true) {
   if(s2.Length<s1.Length)
      s2 = s2 + "*";
   else
      break;
}
print(s1);   // Prints Sam W.
print(s2);   // Prints ******
```

注意，跳出无限循环的条件可以移至循环控制中，使其成为显式有限循环。然而，对于某些循环来说，使用无限循环结构更容易编写。

在 **if...else** 语句中使用布尔型标识符能够使您方便地在程序中包括语句或在不同的语句之间进行选择。这种技术在开发程序时很有用。然而，直接包含语句（不用 **if** 语句）或使用注释避免包含语句则更为有效。

有关更多信息，请参见 JScript 条件结构。

请参见

参考

true 标识符
false 标识符
boolean 数据类型 (JScript)
Boolean 对象

概念

JScript 表达式

其他资源

JScript 中的数据
JScript 条件结构

数值数据

在 JScript 中，两种数值数据类型（整型数据和浮点数据）之间的选择取决于使用它们的特定环境。对整型数据和浮点数据的表示也有不同的方式。

正整数、负整数和数字零都是整数。它们可通过以 10 为基数（十进制）、以 8 为基数（八进制）和以 16 为基数（十六进制）来表示。JScript 中的大多数数字是十进制的。通过在整数前面加前导 0（零）来表示八进制整数。八进制整数只包含 0 到 7 的数字。前面有 0 而包含数字 8 和/或 9 的数字将被解释为十进制数字。一般不推荐使用八进制数字。

通过在整数前面加前导"0x"（零和 x|X）来表示十六进制 (hex) 整数。字母 A 到 F 以单个数字的形式表示以 10 为基数的 10 到 15。即 0xF 相当于 15，0x10 相当于 16。

八进制和十六进制数字都可以是负值，但不能有小数部分，也不能写成科学（指数）计数法。

浮点值是带有小数部分的整数。像整数一样，可以用数字后跟小数点和更多数字的形式来表示浮点值。此外，还可以用科学计数法来表示它们。即，使用大写或小写字母 *e* 来表示"10 的幂"。以单个 0 开头并包含小数点的数字被解释为十进制浮点标识符而不是八进制标识符。

另外，JScript 中的浮点数字可以表示整型数据类型所不能表示的特殊数值。这些是：

- **NaN**（不是数字）。当对不适当的数据（比如字符串或未定义值）执行数学运算时使用该值。
- **Infinity**。当一个正数太大以至于在 JScript 中无法表示时使用该值。
- -**Infinity**（负"无穷大"）。当一个负数的数值太大以至于在 JScript 中无法表示时使用该值。
- 正 0 和负 0。JScript 在某些条件下会区分正的零和负的零。

下面是 JScript 数字的一些示例。注意，以"0x"开头并包含小数点的数字将生成错误。

数字	说明	等效十进制数字
.0001、0.0001、1e-4、1.0e-4	四个等效的浮点数字。	0.0001
3.45e2	浮点数。	345
42	一个整数。	42
0378	一个整数。虽然此值看上去像一个八进制数字（因为以零开头），但 8 不是有效的八进制数字，因此将被视为十进制数字。这将产生 1 级警告。	378
0377	一个八进制整数。注意，虽然它看上去只比上面那个数字小 1，但它们的实际值相差甚远。	255
0.0001, 00.0001	一个浮点数字。即使此值以零开头，它也不是八进制数字，因为它有小数点。	0.0001
0Xff	十六进制整数。	255
0x37CF	十六进制整数。	14287
0x3e7	十六进制整数。注意，字母 *e* 不被视为幂。	999
0x3.45e2	这是一个错误。十六进制数字不能有小数部分。	N/A（编译器错误）

任何整型数据类型的变量只能表示一个有限范围的数字。如果试图将过大或过小的数值标识符分配给整型数据类型，将会在编译时生成类型不匹配错误。有关更多信息，请参见数据类型摘要。

标识符的数据类型

在大多数情况下，JScript 解释数值标识符的数据类型都是无关紧要的。但是，当数字很大或非常精确时，这些细节就很重要了。

根据其标识符大小及其使用情况，JScript 中的整数标识符可表示 **int**、**long**、**ulong**、**decimal** 或 **double** 类型的数据。在 **int** 类型范围(-2147483648 到 2147483647)内的标识符被解释为 **int** 类型。该范围以外但 **long** 类型范围以内(-9223372036854775808 至 9223372036854775807)的标识符解释为 **long**。该范围以外但 **ulong** 类型范围以内(9223372036854775807 至 18446744073709551615)的标识符解释为 **ulong**。其他所有整数标识符被解释为 **double** 类型，它必然伴有精度的降低。以上规则的例外情况是：如果标识符直接存储在类型为 **decimal** 的变量或常数中，或者如果标识符传递给接受类型为 **decimal** 的函数，则该标识符被解释为 **decimal**。

JScript 浮点标识符被解释为 **double** 数据类型，除非该标识符直接用作 **decimal**(如整数标识符)，在后一种情况下，该标识符被解释为 **decimal**。**decimal** 数据类型不能表示 **NaN**、正 **Infinity** 或负 **Infinity**。

请参见

参考

NaN 属性(全局)
Infinity 属性

概念

JScript 表达式

其他资源

JScript 中的数据
数据类型 (JScript)

对象数据

对象标识符可初始化 JScript **Object** 对象。对象标识符用逗号分隔的列表来表示，该列表由一对大括号 ({}) 包围。此列表的每个元素都是一个后面跟有冒号和属性值的属性。该值可以是任何有效的 JScript 表达式。

使用对象数据

在本示例中，变量 `obj` 初始化为具有两个属性(`x` 和 `y`)的对象，它们的值分别是 1 和 2。

```
var obj = { x:1, y:2 };
```

对象标识符可以嵌套。在本示例中，标识符 `cylinder` 指的是具有三个属性(`height`、`radius` 和 `sectionAreas`)的对象。而 `sectionAreas` 属性也是一个有着自己的属性(`top`、`bottom` 和 `side`)的对象。

```
var r = 3;
var h = 2;
var cylinder = { height : h, radius : r,
                 sectionAreas : { top : 4*Math.PI*r*r,
                                  bottom : 4*Math.PI*r*r,
                                  side : 2*Math.PI*r*h } };
```

注意

对象标识符不能用于初始化基于类的对象的实例。必须使用适当的构造函数来执行初始化。有关更多信息，请参见基于类的对象。

请参见

参考

Object 对象

概念

JScript 表达式

其他资源

JScript 中的数据

内部对象

字符串数据

字符串值是一个由零或多个相连的 Unicode 字符(字母、数字和标点符号)组成的链表。字符串数据类型表示 JScript 中的文本。若要在脚本中包含字符串标识符，请使用成对的单引号或双引号将它们引起来。被单引号引起的字符串内可以包含双引号，而被双引号引起的字符串内也可以包含单引号。下面是字符串的示例：

使用字符串数据

```
"The earth is round."
'"Come here, Watson. I need you." said Alexander.'
"42"
"15th"
'c'
```

JScript 提供了可以包含在字符串中的转义序列，用来创建不能直接键入的字符。这些序列都以反斜杠开头。反斜杠是一个转义字符，通知 JScript 解释器下一个字符是特殊字符。

转义序列	含义
\b	Backspace
\f	换页符(很少使用)
\n	换行符(换行)
\r	回车符。与换行符一起使用 (\r\n) 以设定输出格式。
\t	水平制表符
\v	垂直制表符。不符合 ECMAScript 标准，与 Microsoft Internet Explorer 6.0 不兼容。
\'	单引号 (')
\"	双引号 (")
\\	反斜杠 (\)
n	八进制数字 *n* 表示的 ASCII 字符。*n* 的值必须在 0 到 377 范围内(八进制)。
\x*hh*	两位十六进制数字 *hh* 表示的 ASCII 字符。
\u*hhhh*	四位十六进制数字 *hhhh* 表示的 Unicode 字符。

本表中未包含的任何转义序列只是表示转义序列中反斜杠后面的字符。例如，"\a"被解释为"a"。

由于反斜杠本身表示转义序列的开头，因此在脚本中不能直接键入。如果想包含反斜杠，则必须键入两个相连的字符 (\\)。

```
'The image path is C:\\webstuff\\mypage\\gifs\\garden.gif.'
```

单引号和双引号转义序列可用来在字符串标识符中包含引号。本示例显示了嵌入的引号。

```
'The caption reads, \"After the snow of \'97. Grandma\'s house is covered.\"'
```

JScript 使用内部 **char** 数据类型来表示单个字符。尽管字符串本身不属于 **char** 类型，但可以将包含一个字符或一个转义序列的字符串赋给 **char** 类型的变量。

包含零字符 ("") 的字符串是空(零长度)字符串。

请参见

参考

String 数据类型 (JScript)

String 对象

概念

JScript 表达式

其他资源

JScript 中的数据

数据类型摘要

JScript 提供了许多可以在您的程序中使用的数据类型。这些类型可分为两大类：值数据类型和引用数据类型（也称为 JScript 对象）。若要向 JScript 中添加类型，可导入包含新数据类型的命名空间或包，也可以定义能用作新数据类型的新类。

数据类型详细信息

下表显示了 JScript 支持的值数据类型。第二列描述了等效的 Microsoft .NET Framework 数据类型。可以声明一个 .NET Framework 类型或 JScript 值类型的变量，并获得完全相同的结果。还给出了每个类型的存储大小（适用时）和范围。第三列列出了给定类型的一个实例所要求的存储量（如果适用的话）。第四列提供了给定类型可存储的值的范围。

JScript 值类型	.NET Framework 类型	存储大小	范围
boolean	Boolean	N/A	true 或 false
char	Char	2 个字节	任何 Unicode 字符
float（单精度浮点）	Single	4 个字节	范围在大约 7 位准确度 -1038 至 1038。可以表示最小可为 10-44 的数字。
Number, double（双精度浮点）	Double	8 个字节	大致范围为 -10308 至 10308，准确度约 15 位。可以表示最小可为 10-323 的数字。
decimal	Decimal	12 个字节（整数部分）	大致范围为 -1028 至 1028，准确度约 28 位。可以表示最小可为 10-28 的数字。
byte（无符号）	Byte	1 个字节	0 到 255
ushort（无符号短整型）	UInt16	2 个字节	0 到 65,535
uint（无符号整数）	UInt32	4 个字节	0 到 4,294,967,295
ulong（无符号扩展整数）	UInt64	8 个字节	0 至约 1020
sbyte（有符号）	SByte	1 个字节	-128 到 127
short（有符号短整型）	Int16	2 个字节	-32,768 到 32,767
int（有符号整数）	Int32	4 个字节	-2,147,483,648 到 2,147,483,647
long（有符号扩展整数）	Int64	8 个字节	大约 -1019 至 1019
void	N/A	N/A	用作不返回值的函数的返回类型。

下表显示了 JScript 提供并可用作类型的引用数据类型（JScript 对象）。引用类型没有预定义的特定存储大小。

JScript 引用类型	.NET Framework 类型	引用
ActiveXObject	无直接等效项	自动对象。
数组	与 Array 和类型化数组互用	任何类型的数组。

Boolean	与 **Boolean** 进行互操作	布尔值，为 **true** 或 **false**。
Date	与 DateTime 进行互操作	日期是使用 JScript **Date** 对象实现的。范围是在 1970 年 1 月 1 日的前后都有大约 285,616 年。
Enumerator	无直接等效项	集合中项的枚举。仅用于向后兼容性。
Error	无直接等效项	**Error** 对象。
Function	无直接等效项	**Function** 对象。
Number	与 **Double** 进行互操作	数值，大约的范围是从 -10308 到 10308，准确度约为 15 位。可以表示最小可为 10-323 的数字。
Object	与 Object 进行互操作	**Object** 引用。
RegExp	与 Regex 进行互操作	一个正则表达式对象。
String 数据类型(长度可变)	String	0 到大约 20 亿个 Unicode 字符。每个字符是 16 位(两个字节)。
String 对象(长度可变)	与 **String** 进行互操作	0 到大约 20 亿个 Unicode 字符。每个字符是 16 位(两个字节)。
VBArray	无直接等效项	只读 Visual Basic 数组。仅用于向后兼容性。

请参见

参考

import 语句
package 语句
class 语句

概念

用户定义的数据类型
复制、传递和比较数据

其他资源

数据类型 (JScript)
对象 (JScript)
JScript 对象

用户定义的数据类型

有时可能需要 JScript 没有提供的数据类型。在这种情况下，可以导入定义新类的包，或使用 **class** 语句创建自己的数据类型。类可以用于类型批注，并可使类型化数组与 JScript 中预定义的数据类型有完全相同的行为方式。

定义数据类型

下面的示例使用 **class** 语句定义了一种新的数据类型 — `myIntVector`。此新类型用在函数声明中，表示函数的参数类型。而且还使用此新类型批注了一个变量的类型。

```
// Define a class that stores a vector in the x-y plane.
class myIntVector {
   var x : int;
   var y : int;
   function myIntVector(xIn : int, yIn : int) {
      x = xIn;
      y = yIn;
   }
}

// Define a function to compute the magnitude of the vector.
// Passing the parameter as a user defined data type.
function magnitude(xy : myIntVector) : double {
   return( Math.sqrt( xy.x*xy.x + xy.y*xy.y ) );
}

// Declare a variable of the user defined data type.
var point : myIntVector = new myIntVector(3,4);
print(magnitude(point));
```

该代码的输出为：

```
5
```

请参见

参考

class 语句

package 语句

概念

类型批注

其他资源

数据类型 (JScript)

JScript 对象

类型化数组

类型化数组是可以批注变量、常数、函数和参数的一种数据类型，就像内部数据类型一样。每一个类型化数组都有一个基数据类型，数组的每个元素都属于此基类型。此基类型本身可以是一种数组类型，允许用于数组的数组。

使用类型化数组

后面跟一组方括号的数据类型定义了一维的类型化数组。若要定义 *n* 维数组，基数据类型后面要跟一组方括号，用 *n*-1 个逗号将括号隔开。

对于类型化数组类型的变量，最初没有分配存储区，初始值为 **undefined**。若要初始化数组变量，可使用 **new** 运算符、数组标识符、数组构造函数或另一个数组。在声明类型化数组变量时，或以后用其他类型的变量声明时，会执行初始化。如果变量或参数的维数与分配给此变量或传递给此参数的数组的维数（或类型）不匹配，将产生类型不匹配错误。

使用数组构造函数，可创建具有指定（固定）大小的给定本机类型的数组。每个参数必须是计算结果为非负整数的表达式。每个参数的值决定数组每一维的大小；参数的数目决定数组的维数。

下面显示了一些简单的数组声明：

```
// Simple array of strings; initially empty. The variable 'names' itself
// will be null until something is assigned to it
var names : String[];

// Create an array of 50 objects; the variable 'things' won't be null,
// but each element of the array will be until they are assigned values.
var things : Object[] = new Object[50];
// Put the current date and time in element 42.
things[42] = new Date();

// An array of arrays of integers; initially it is null.
var matrix : int[][];
// Initialize the array of arrays.
matrix = new (int[])[5];
// Initialize each array in the array of arrays.
for(var i = 0; i<5; i++)
   matrix[i] = new int[5];
// Put some values into the matrix.
matrix[2][3] = 6;
matrix[2][4] = 7;

// A three-dimensional array
var multidim : double[,,] = new double[5,4,3];
// Put some values into the matrix.
multidim[1,3,0] = Math.PI*5.;
```

请参见

参考

var 语句

new 运算符

function 语句

概念

类型批注

其他资源

数据类型 (JScript)

类型转换

类型转换是将一个值从一种类型更改为另一个类型的过程。例如，可以将字符串"1234"转换为一个数字。而且，可以将任意类型的数据转换为 **String** 类型。某些类型转换永远不会成功。例如，**Date** 对象不能转换为 **ActiveXObject** 对象。

类型转换有可能扩大，也有可能收缩：扩大转换永远不会溢出，并且总是成功的；而收缩转换必然伴有信息的丢失，并有可能失败。

两种转换类型都可以是显式（使用数据类型标识符）或隐式（不使用数据类型标识符）的。有效的显式转换即使会导致信息的丢失，也始终是成功的。而隐式转换只有在转换过程不丢失数据的情况下才能成功；否则，它们就会失败并生成编译或运行时错误。

当原始数据类型在目标转换类型中没有明显的对应类型时，就会发生丢失数据的转换。例如，字符串"Fred"不能转换为一个数字。在这些情况下，类型转换函数将返回一个默认值。对于 **Number** 类型，默认值为 **NaN**；对于 **int** 类型，默认值为数字 0。

某些类型的转换（比如从字符串转换为数字）非常耗时。程序使用的转换越少，效率就越高。

隐式转换

大多数类型转换（比如给变量赋值）会自动发生。变量的数据类型决定了表达式转换的目标数据类型。

本示例说明如何在 **int** 值、**String** 值和 **double** 值之间进行数据的隐式转换。

```
var i : int;
var d : double;
var s : String;
i = 5;
s = i;  // Widening: the int value 5 coverted to the String "5".
d = i;  // Widening: the int value 5 coverted to the double 5.
s = d;  // Widening: the double value 5 coverted to the String "5".
i = d;  // Narrowing: the double value 5 coverted to the int 5.
i = s;  // Narrowing: the String value "5" coverted to the int 5.
d = s;  // Narrowing: the String value "5" coverted to the double 5.
```

在编译此代码时，编译时警告可能会声明收缩转换有可能失败或者速度很慢。

如果转换要求有信息丢失，则隐式收缩转换可能无法执行。例如，下面几行将无法执行。

```
var i : int;
var f : float;
var s : String;
f = 3.14;
i = f;  // Run-time error. The number 3.14 cannot be represented with an int.
s = "apple";
i = s;  // Run-time error. The string "apple" cannot be converted to an int.
```

显式转换

若要将某个表达式显式转换为特定数据类型，可使用数据类型标识符，后面跟要转换的表达式（在括号中）。显式转换比隐式转换需要更多键入，但使用户对结果更有把握。而且，显式转换可以处理有信息丢失的转换。

本示例说明如何在 **int** 值、**String** 值和 **double** 值之间进行数据的显式转换。

```
var i : int;
var d : double;
var s : String;
i = 5;
s = String(i);  // Widening: the int value 5 coverted to the String "5".
d = double(i);  // Widening: the int value 5 coverted to the double 5.
s = String(d);  // Widening: the double value 5 coverted to the String "5".
i = int(d);     // Narrowing: the double value 5 coverted to the int 5.
i = int(s);     // Narrowing: the String value "5" coverted to the int 5.
d = double(s);  // Narrowing: the String value "5" coverted to the double 5.
```

即使转换要求信息丢失，显式收缩转换通常也能够执行。显式转换不能用于在不兼容的数据类型之间执行转换。例如，不能在 **Date** 数据和 **RegExp** 数据之间相互转换。另外，转换对于某些值是不可能的，因为对于要转换的值没有敏感的值。例如，当试图将双精度值 **NaN** 显式转换为 **decimal** 时会引发错误。这是因为不存在能够用 **NaN** 标识的自然 **decimal** 值。

在本示例中，有小数部分的一个数字转换为一个整数，一个字符串转换为一个整数。

```
var i : int;
var d : double;
var s : String;
d = 3.14;
i = int(d);
print(i);
s = "apple";
i = int(s);
print(i);
```

输出为：

```
3
0
```

显式转换的行为取决于原始数据类型和目标数据类型。

请参见

参考

undefined 属性

概念

类型批注

其他资源

JScript 数据类型

JScript 变量和常数

在任何编程语言中，数据表示信息。例如，该字符串标识符就包含一个问题：

```
'How old am I?'
```

变量和常数存储着脚本利用变量名或常数名可方便引用的数据。由变量存储的数据在程序运行时可能会更改，而由常数存储的数据则不会。使用变量的脚本实际访问的是变量所表示的数据。下面是一个示例，其中将 `EndDate` 和 `TodaysDate` 的差所派生的值赋给名为 `NumberOfDaysLeft` 的变量。

```
NumberOfDaysLeft = EndDate - TodaysDate;
```

如果机械地理解，那么脚本使用变量来存储、检索和操作出现在脚本中的值。常数引用不会发生改变的数据。应始终创建有意义的变量名，以帮助您记住变量的目的并帮助其他人确认脚本的功能。

本节内容

JScript 变量和常数的类型

讨论如何为变量选择适当的数据类型以及正确选择变量数据类型的益处。

声明 JScript 变量和常数

解释如何声明类型化和非类型化变量与常数以及如何初始化它们。

变量和常数的范围

阐释 JScript 中全局和局部范围的差异以及局部范围如何隐藏全局范围。

未定义的值

解释未定义值的概念、如何确定变量或属性是否是未定义的，以及如何取消定义变量和属性。

相关章节

JScript 标识符

解释如何创建 JScript 中标识符的有效名称。

JScript 数据类型

包含指向特定主题的链接，这些主题解释如何使用 JScript 中的基元数据类型、引用数据类型和 .NET Framework 数据类型。

JScript 赋值与相等

解释 JScript 如何将值分配给变量、数组元素和属性元素，并且解释 JScript 所使用的等式语法。

JScript 参考

列出 JScript 语言参考中涉及的元素，并提供指向特定主题的链接，这些主题解释了有关正确使用这些语言元素的幕后详细信息。

JScript 变量和常数的类型

您可能会发现，JScript 中有许多数据类型在编写代码时很有用。有效地使用数据类型能够使程序比使用默认的 JScript 数据类型时更快地加载和运行。另外，编译器可以提供有关错误使用各种类型的有价值的错误信息和警告。

讨论

比如，如果变量存储的整数值始终不超过 1,000,000，那么就完全没有必要使用 8 字节 **double** 类型。实际上，对于此数据最有效的类型是 **int**，它是 4 字节的整数类型，可存储从 -2,147,483,648 到 2,147,483,647 范围内的数据值。

用类型批注声明变量或常数可确保此变量或常数只存储适当类型的数据。JScript 有许多其他的数据类型用于类型批注。有关更多信息，请参见数据类型摘要。若要向 JScript 中添加类型，要么导入包含类型的程序集，要么声明用户定义的类型（类）。

请参见

概念

声明 JScript 变量和常数

其他资源

JScript 变量和常数

JScript 数据类型

声明 JScript 变量和常数

JScript 程序必须指定程序将要使用的每个变量的名称。另外，此程序还可以指定每个变量将存储哪种数据类型。这两个任务都是用 **var** 语句完成的。

声明类型化变量和常数

在 JScript 中，可以声明变量，同时使用类型批注声明其类型。在下面的示例中，将变量 `count` 声明为 **int**（整数）类型。由于未提供初始值，`count` 具有默认值 **int**，即 0（零）。

```
var count : int; // An integer variable.
```

还可以给变量赋初始值：

```
var count : int = 1; // An initialized integer variable.
```

常数与变量的声明方式大致相同，它必须进行初始化。一旦定义了常数值，其值就不能更改。例如：

```
const daysInWeek : int = 7;            // An integer constant.
const favoriteDay : String = "Friday"; // A string constant.
const maxDaysInMonth : int = 31, maxMonthsInYear : int = 12
```

当然，在声明特定类型的变量时，所赋的值必须对该类型是有意义的。例如，将一个字符字符串值赋给整数变量就是没有意义的。这样做时，程序会引发 **TypeError** 异常，指示在代码中存在类型不匹配。**TypeError** 是一种可出现在运行脚本中的异常或错误。catch 块可以捕获 JScript 程序引发的异常。有关更多信息，请参见 try...catch...finally 语句。

虽然将每个声明放在单独一行时阅读代码更为方便，但也可以同时声明多个变量的类型和初始值。例如，此代码段读起来就很费劲：

```
var count : int = 1; amount : int = 12, level : double = 5346.9009
```

下面的代码段读起来更为轻松：

```
var count : int = 1;
var amount : int = 12;
var level : double = 5346.9009;
```

在一行上声明若干个变量时还要记住，类型批注只应用于它前面紧邻的变量。在下面的代码中，`x` 是一个 **Object**，因为它是默认的类型，并且 `x` 未指定值，而 *y* 是一个 **int**。

```
var x, y : int;
```

声明非类型化变量和常数

并非一定要使用类型化变量，但使用非类型化变量的程序运行更慢，并且容易出错。

下面的简单示例声明了一个名为 `count` 的变量。

```
var count;  // Declare a single declaration.
```

在没有指定数据类型的情况下，变量或常数的默认类型为 **Object**。在没有赋值的情况下，变量的默认值为 **undefined**。下面的代码说明了一个命令行程序的这些默认设置：

```
var count; // Declare a single declaration using default type and value.
print(count); //Print the value of count.
print(typeof(count)); // Prints undefined.
```

可以为变量分配初始值但不声明类型：

```
var count = 1; // An initialized variable.
```

下面的示例使用单个 **var** 语句声明了若干个变量：

```
var count, amount, level;  // multiple declarations with a single var keyword.
```

若要在不分配特定值的情况下声明一个变量并对其进行初始化，请为其分配 JScript 值 **null**。这是一个示例。

```
var bestAge = null;
```

存在已声明但没有分配值的变量，但它具有 JScript 值 **undefined**。这是一个示例。

```
var currentCount;
var finalCount = 1 * currentCount; // finalCount has the value NaN since currentCount is un
defined.
```

在 JScript 中，**null** 和 **undefined** 之间的主要差异在于：**null** 转换为零（尽管它不是零），而 **undefined** 转换为特殊值 **NaN**（不是一个数字）。有趣的是，在使用相等运算符 (==) 时，**null** 值和 **undefined** 值的比较结果总是相等。

声明非类型化常数的过程与声明变量的过程很类似，但必须为非类型化常数提供初始值。例如：

```
const daysInWeek  = 7;
const favoriteDay  = "Friday";
const maxDaysInMonth  = 31, maxMonthsInYear = 12
```

不用 var 声明变量

可以在声明中不使用 **var** 关键字并分配一个值来声明变量。这就是所谓的*隐式声明*，但不建议这样做。隐式声明用分配的名称创建一个全局对象的属性；该属性很像一个具有全局范围可见性的变量。当您在过程级别声明变量时，通常不希望它在全局范围内可见。在这种情况下，必须在变量声明中使用 **var** 关键字。

```
noStringAtAll = ""; // The variable noStringAtAll is declared implicitly.
```

不能使用从未声明过的变量。

```
var volume = length * width; // Error - length and width do not yet exist.
```

注意

不用 **var** 关键字声明变量将在运行于快速模式（JScript 的默认模式）下时生成编译时错误。若要从不使用 **var** 关键字的命令行编译程序，则必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

请参见

任务

如何：从命令行编译 JScript 代码

概念

JScript 标识符

其他资源

JScript 变量和常数

JScript 数据类型

变量和常数的范围

JScript 有三个范围：全局、局部和类。如果在函数或类定义之外声明变量或常数，则是全局变量，它的值可在整个程序中访问和修改。如果在函数定义内声明一个变量，则该变量为局部变量。每当执行函数时，都会创建和销毁该变量；在函数之外无法访问该变量。如果在类定义内声明一个变量，则该变量仅在类内部可用，不能从全局范围访问。有关更多信息，请参见基于类的对象。

讨论

像 C++ 这样的语言也有"块范围"；任何一组大括号 ({}) 都会定义一个新的范围。JScript 不支持块范围。

局部变量可以与全局变量具有相同的名称，但它们是完全不同且相互分离的。因此，更改一种变量的值对另一种变量不会产生影响。在声明局部变量的函数中，只有局部版本才具有意义。这就称为可见性。

```
// Define two global variables.
var name : String = "Frank";
var age : int = "34";

function georgeNameAge() {
   var name : String; // Define a local variable.
   name = "George";   // Modify the local variable.
   age = 42;          // Modify the global variable.
   print(name + " is " + age + " years old.");
}

print(name + " is " + age + " years old.");
georgeNameAge();
print(name + " is " + age + " years old.");
```

此程序的输出显示：不更改全局变量的值就可以修改局部变量。在函数内部对全局变量的更改会影响全局范围内的值。

```
Frank is 34 years old.
George is 42 years old.
Frank is 42 years old.
```

由于 JScript 会在执行任何代码之前处理变量和常数声明，因此是在条件块中声明还是在某一其他构造中声明并不重要。JScript 一旦找到所有变量和常数，就会执行函数中的代码。这就意味着在到达常数声明语句前局部常数值是未定义的，在函数中分配变量前局部变量是未定义的。

有时，这样会导致意外的行为。请看下面的程序。

```
var aNumber = 100;
var anotherNumber = 200;
function tweak() {
   var s = "aNumber is " + aNumber + " and ";
   s += "anotherNumber is " + anotherNumber + "\n";
   return s;
   if (false)  {
      var aNumber;                // This statement is never executed.
      aNumber = 123;              // This statement is never executed.
      const anotherNumber = 42;   // This statement is never executed.
   } // End of the conditional.
} // End of the function definition.

print(tweak());
```

该程序的输出为：

```
aNumber is undefined and anotherNumber is undefined
```

您可能想要 `aNumber` 为 100 或 123，`anotherNumber` 为 200 或 42，但这两个值都是 **undefined**。由于 `aNumber` 和

anotherNumber 都是用局部范围定义的，它们隐藏了具有相同名称的全局变量和常数。由于始终不运行初始化局部变量和常数的代码，因此它们的值为 **undefined**。

在快速模式下要求使用显式变量声明。当关闭快速模式时，要求使用隐式变量声明。函数内隐式声明的变量（即出现在赋值表达式的左边不带 var 关键字的变量）是全局变量。

请参见

概念

未定义的值

其他资源

JScript 变量和常数

未定义的值

在 JScript 中，可以声明一个变量而不为其赋值。一个经类型批注的变量假定该类型的默认值。例如，数值类型的默认值为零，**String** 数据类型的默认值为空字符串。然而，没有指定数据类型的变量有一个 **undefined** 初始值和一个 **undefined** 数据类型。同样，访问不存在的 expando 对象属性或数组元素的代码将返回一个 **undefined** 值。

使用 Undefined 值

若要确定变量或对象属性是否存在，请将它与 **undefined** 关键字（仅对声明变量或属性有效）作比较，或检查其类型是否为"undefined"（即使对未声明变量或属性也有效）。在下面的代码示例中，假定程序员试图测试是否已声明了变量 `x`：

```
// One method to test if an identifier (x) is undefined.
// This will always work, even if x has not been declared.
if (typeof(x) == "undefined"){
   // Do something.
}
// Another method to test if an identifier (x) is undefined.
// This gives a compile-time error if x is not declared.
if (x == undefined){
   // Do something.
}
```

检查是否未定义某个变量或对象属性的另一种方法是：将其值与 **null** 作比较。包含 **null** 的变量不包含值或对象。换句话说，它不包含有效的数字、字符串、布尔值、数组或对象。通过为变量赋 **null** 值可以清除变量的内容（不删除变量）。注意，使用相等运算符 (==) 时，**undefined** 和 **null** 值的比较结果为相等。

注意
在 JScript 中，使用相等运算符时，**null** 与 0 的比较结果不相等。此行为与其他语言（如 C 和 C++）不同。

在本示例中，测试 `obj` 对象，看它是否有 `prop` 属性。

```
// A third method to test if an identifier (obj.prop) is undefined.
if (obj.prop == null){
   // Do something.
}
```

该比较结果为 **true**

- 如果 `obj.prop` 属性包含值 **null**
- 如果 `obj.prop` 属性不存在

还有另外一种方法可以检查某个对象属性是否存在。如果指定的属性在所提供的对象中，则 **in** 运算符返回 **true**。例如，如果 `prop` 属性在 `obj` 对象中，下面的代码测试 **true**。

```
if ("prop" in someObject)
// someObject has the property 'prop'
```

若要从某个对象中移除属性，可使用 **delete** 运算符。

请参见

参考

null 标识符
undefined 属性
in 运算符
delete 运算符

其他资源

JScript 变量和常数
JScript 中的数据

JScript 数据类型
数据类型 (JScript)

JScript 对象

JScript 对象是数据和功能的封装。对象由属性(值)和方法(函数)组成。属性是对象的数据组件，而方法则提供了操作数据或对象的功能。JScript 支持 5 种对象：内部对象、基于原型的对象、基于类的对象、宿主对象(由宿主提供，如 ASP.NET 中的 **Response**)和 .NET Framework 类(外部组件)。

new 运算符与选定对象的构造函数相结合，可创建和初始化对象的实例。下面是使用构造函数的几个示例。

```
var myObject = new Object();           // Creates a generic object.
var birthday = new Date(1961, 5, 10);  // Creates a Date object.
var myCar : Car = new Car("Pinto");    // Creates a user-defined object.
```

JScript 支持两种类型的用户定义对象(基于类和基于原型)。两种类型都有独特的优点和缺点。基于原型的对象可动态扩展，但它们运行很慢，且不能与来自其他 .NET Framework 语言的对象有效地交互操作。另一方面，基于类的对象可扩展现有的 .NET Framework 类，提供类型安全，并确保有效的操作。通过使用 **expando** 修饰符定义类，基于类的对象可动态扩展(类似于基于原型的对象)。

本节内容

内部对象

列出 JScript 脚本中使用的一些公共对象，也列出了一些链接，指向说明如何使用这些对象的信息。

基于类的对象

提供如何使用 JScript 基于类的对象模型的指南，并且描述如何定义类(使用方法、字段和属性)、如何定义从其他类继承的类以及如何定义 expando 类。

基于原型的对象

提供有关如何使用 JScript 基于原型的对象模型的指南，并且提供指向特定信息的链接，这些信息描述基于原型的对象的自定义构造函数和继承。

相关章节

JScript 数据类型

包含指向特定主题的链接，这些主题解释如何使用 JScript 中的基元数据类型、引用数据类型和 .NET Framework 数据类型。

JScript 参考

列出"JScript 语言参考"所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

ASP.NET 介绍

介绍 ASP.NET，解释它如何能用于任何一种 .NET 兼容的语言，包括 JScript，来创建企业级 Web 应用程序，并提供一些指向参考信息的链接。

介绍 .NET Framework 类库

介绍 .NET Framework 类库，解释命名规则和系统命名空间，并提供指一些向参考信息的链接。

内部对象

JScript 提供了 16 种内部对象作为语言规范的一部分。每个内部对象都有关联的方法和属性，这些都在语言参考中有详细的描述。本节将讨论几种最常使用的对象，以阐释内部对象的基本语法和用法。

本节内容

JScript Array 对象

描述如何使用数组对象，如何利用它们的 expando 属性，以及它们如何与类型化数组进行比较。

JScript Date 对象

描述可接受的日期范围以及如何用当前日期和时间或任意日期和时间创建对象。

JScript Math 对象

阐释如何使用方法和属性以操作数值数据。

JScript Number 对象

解释数字对象的目的及其属性的意义。

JScript Object 对象

描述如何向对象添加 expando 属性和方法，并解释使用点运算符和索引运算符访问对象成员的差异。

JScript String 对象

解释字符串对象的目的以及字符串文本如何使用 **String** 对象的方法。

相关章节

JScript 对象

指向特定主题的链接，这些主题解释 JScript 中内部对象的语法和用法。

JScript 参考

列出“JScript 语言参考”所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

对象

列出 JScript 语言提供的所有对象，并提供指向语言参考信息的链接，这些信息解释每个对象的正确用法和语法。

JScript Array 对象

Array 对象是一个变量，它可以将相关的数据片段组合起来。一个唯一的数字（称为索引或下标）引用数组中的每一段数据。若要访问存储在数组中的数据，应使用索引运算符"[]"将数组标识符和索引组合起来，如 `theMonths[0]`。

创建数组

若要创建新的数组，可使用 **new** 运算符和 **Array** 构造函数。在本示例中，使用数组构造函数构造一个长度为 12 的数组。然后将数据输入到该数组中。

```
var theMonths = new Array(12);
theMonths[0] = "Jan";
theMonths[1] = "Feb";
theMonths[2] = "Mar";
theMonths[3] = "Apr";
theMonths[4] = "May";
theMonths[5] = "Jun";
theMonths[6] = "Jul";
theMonths[7] = "Aug";
theMonths[8] = "Sep";
theMonths[9] = "Oct";
theMonths[10] = "Nov";
theMonths[11] = "Dec";
```

当使用 **Array** 关键字创建数组时，JScript 将一个 **length** 属性包括进去，该属性记录项的数目。如果不指定一个数字，则长度设置为零，数组就没有任何项。如果指定一个数字，则长度就设置为该数字。如果指定多个参数，则这些参数就用作数组中的各项。此外，参数的个数要赋给长度属性，如下面的示例所示，此示例同前面的示例是等效的。

```
var theMonths = new Array("Jan", "Feb", "Mar", "Apr", "May", "Jun",
"Jul", "Aug", "Sep", "Oct", "Nov", "Dec");
```

数组文本提供了在数组中输入数据的另一种技术。有关更多信息，请参见数组数据。

Array 对象存储"稀疏"数组。即，如果某个数组有 3 个元素，编号为 0、1 和 2，那么即使不存在元素 3 到 49，也可以有元素 50。当您添加元素到 即，如果某个数组有 3 个元素，编号为 0、1 和 2，那么即使不存在元素 3 到 49，也可以有元素 50。当您添加元素到 **Array** 对象时，JScript 自动更改 <legacyBold>length</legacyBold> 属性的值。对象时，JScript 自动更改 **length** 属性的值。JScript 中的数组索引始终是从 0（而不是 1）开始，因此长度属性总是比数组中最大的索引大 1。

使用数组的 Expando 属性

数组对象像基于 JScript **Object** 对象的任何其他对象一样，也支持 expando 属性。Expando 属性是可以动态地添加到数组和从数组删除的新属性，类似于数组索引。与数组索引不同的是，数组索引必须是整数，而 expando 属性是字符串。此外，添加或删除 expando 属性并不更改 **length** 属性。

例如：

```
// Initialize an array with three elements.
var myArray = new Array("Hello", 42, new Date(2000,1,1));
print(myArray.length); // Prints 3.
// Add some expando properties. They will not change the length.
myArray.expando = "JScript";
myArray["another Expando"] = "Windows";
print(myArray.length); // Still prints 3.
```

类型化数组

创建如上所示的 `theMonths` 数组的另一种更快的方法是创建一个类型化（本机）数组，在本例中是一个字符串数组：

```
var theMonths : String[] = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
"Oct", "Nov", "Dec"];
```

访问类型化数组的元素比访问 JScript 数组对象中的元素更快。类型化数组与其他 .NET Framework 语言中的数组相兼容，并提供了类型安全。

JScript **Arrays** 对象在列表、队列、堆栈等方面是非常灵活而好用的，但本机数组在存储相同类型的固定大小项目时更具优势。一般来说，除非需要 **Array** 对象的特殊功能（如动态调整大小等），否则应使用类型化数组。

所有非破坏性的 JScript **Array** 方法（不更改长度的方法）都可以按类型化数组的方式调用。

请参见

参考

Array 对象

其他资源

内部对象

JScript Date 对象

JScript **Date** 对象可用来表示任意日期和时间，获取当前系统日期，以及计算日期差。它有几种预定义的属性和方法。**Date** 对象存储一周的某一天；月、日、年；以及以小时、分钟、秒、毫秒计的时间。此信息建立在自从协调通用时间 (UTC)（以前称为格林尼治标准时间）1970 年 1 月 1 日 00:00:00.000 以来的毫秒数的基础上。JScript 可以处理大约范围从公元前 250,000 年到公元 255,000 年之间的日期，尽管某些格式设置功能只在公元 0 年到公元 9999 年的日期范围内受到支持。

创建 Date 对象

若要创建新的 **Date** 对象，可使用 **new** 运算符。下面的示例计算当年已过去的天数和剩余的天数。

```
// Get the current date and read the year.
var today : Date = new Date();
// The getYear method should not be used. Always use getFullYear.
var thisYear : int = today.getFullYear();

// Create two new dates, one for January first of the current year,
// and one for January first of next year. The months are numbered
// starting with zero.
var firstOfYear : Date = new Date(thisYear,0,1);
var firstOfNextYear : Date = new Date(thisYear+1,0,1);

// Calculate the time difference (in milliseconds) and
// convert the differnce to days.
const millisecondsToDays = 1/(1000*60*60*24);
var daysPast : double = (today - firstOfYear)*millisecondsToDays;
var daysToGo : double = (firstOfNextYear - today)*millisecondsToDays;

// Display the information.
print("Today is: "+today+".");
print("Days since first of the year: "+Math.floor(daysPast));
print("Days until the end of the year: "+Math.ceil(daysToGo));
```

此程序的输出类似于：

```
Today is: Sun Apr 1 09:00:00 PDT 2001.
Days since first of the year: 90
Days until the end of the year: 275
```

请参见

参考

Date 对象

其他资源

内部对象

JScript Math 对象

Math 对象有许多内部属性和方法。这些属性是特定的数字。

使用 Math 对象

这些特定数字之一就是 pi 的值(大约为 3.14159...)。这是 **Math.PI** 属性的值，下面的示例中就使用了该属性。

```
// A radius variable is declared and assigned a numeric value.
var radius = 4;
var area = Math.PI * radius * radius;
// Note capitalization of Math and PI.
```

Math 对象的一种内置方法为幂方法，或 **pow**，它将数字提升到指定的幂次。下面的示例使用了 pi 和幂的计算。

```
// This formula calculates the volume of a sphere with the given radius.
var volume = (4/3)*(Math.PI*Math.pow(radius,3));
```

另一个示例如下：

```
var x = Math.floor( Math.random()*10 ) + 1;
```

不能显式构造 **Math** 对象；此对象始终对程序可用。

请参见

参考

Math 对象

其他资源

内部对象

JScript Number 对象

Number 对象的主要目的是收集用于默认的数值数据类型(即 **Number** 数据类型)的属性和方法。下表中列出了 **Number** 对象的属性所提供的数值常数。

Number 对象的属性

属性	说明
MAX_VALUE	最大可能值，大约为 1.79E+308；可以是正值或负值。(值在不同系统之间略有不同。)
MIN_VALUE	最小可能值，大约为 2.22E-308；可以是正值或负值。(值在不同系统之间略有不同。)
NaN	特殊的非数字值，"不是数字"(not a number)。
POSITIVE_INFINITY	大于最大正数 (**Number.MAX_VALUE**) 的任何正值都被自动转换为该值；表示无穷大。
NEGATIVE_INFINITY	比最大的负数 (-**Number.MAX_VALUE**) 还要负的任何值都被自动转换为该值；表示负无穷大。

Number.NaN 是一个特殊的属性，定义为"不是数字"(*not a number*)。当数值上下文中使用了不能表示为数字的表达式时，返回 **Number.NaN**。例如，当把字符串"Hello"或 0/0(零除以零)用作一个数字时，将返回 **NaN**。**NaN** 比较为与任何数字或本身不等。若要测试 **NaN** 结果，不用与 **Number.NaN** 比较；而要使用 **Global** 对象的 **isNaN** 方法。

Number 对象的 **toLocaleString** 方法产生一个字符串值，它表示该数字按宿主环境的当前区域设置适当格式化后的值。所用格式化通过用一个(与区域设置有关的)字符将小数点左边的数字按组分隔开，使得大数更容易阅读。有关更多信息，请参见 toLocaleString 方法。

请参见

参考

Number 对象

toLocaleString 方法

其他资源

内部对象

JScript Object 对象

JScript 中所有基于 **Object** 对象的对象都支持 *expando* 属性，即可以在程序运行时添加和移除的属性。

使用 Object 对象

这些属性可以有任何名称，包括数字。属性的名称如果是简单的标识符，则可以写在对象名称后的句点之后，比如：

```
var myObj = new Object();
// Add two expando properties, 'name' and 'age'
myObj.name = "Fred";
myObj.age = 53;
```

您也可以使用索引运算符"[]"来访问对象的属性。如果属性的名称不是简单的标识符，或者在编写脚本时还不知道属性的名称，就需要使用这种方法。方括号内的任意表达式（包括简单的标识符）都可以作为该属性的索引。JScript 中所有 expando 属性的名称在添加到对象之前都被转换为字符串。

在使用索引运算符时，对象被看作是一个"相关数组"。相关数组是一种数据结构，可动态地将任意数据值与任意字符串相关联。在本示例中，添加没有简单标识符的 expando 属性。

```
var myObj = new Object();
// This identifier contains spaces.
myObj["not a simple identifier"] = "This is the property value";
// This identifier is a number.
myObj[100] = "100";
```

虽然索引运算符更常见于与访问数组元素相关联，但在用于对象时，索引始终是以字符串来表达的属性名称。

Array 对象有一个特殊的 **length** 属性，当添加新元素时，该属性会发生更改；一般说来，对象都没有长度属性，即使在使用索引运算符添加属性时也是如此。

请注意两种对象属性访问方式之间的重要差异。

运算符	将属性名称当作	表示属性名称
句点 (.)	标识符	"不能"当作数据来进行操作
索引 ([])	字符串	"可以"当作数据来进行操作

如果您在运行时之前不知道属性名称将是什么（例如，当您基于用户输入构造对象时），此差异将显得十分有用。若要从关联数组中提取所有属性，必须使用 **for ... in** 循环。

请参见

参考

Object 对象

其他资源

内部对象

JScript String 对象

JScript 中的 **String** 对象表示文本数据，比如字、句等等。很少使用 **new** 运算符显式创建字符串对象，因为它们通常是通过将字符串赋给变量而隐式创建的。有关更多信息，请参见 String 对象。

使用 String 对象

String 对象有许多内置方法。其中有一种是 **substring** 方法，此方法返回字符串的一部分。它使用两个数字作为其参数。第一个数字是从零开始的索引，指示子字符串的开头，第二个数字指示子字符串的结尾。

```
var aString : String = "0123456789";
var aChunk : String = aString.substring(4, 7);  // Sets aChunk to "456".
```

String 对象还有一个 **length** 属性。该属性包含字符串中的字符数(0 表示空字符串)。这是一个数值，可直接用于计算中。下面示例获取字符串的长度。

```
var howLong : int = "Hello World".length  // Sets the howLong variable to 11.
```

请参见

参考

String 对象

概念

字符串数据

其他资源

内部对象

基于类的对象

由于 JScript 是基于类的、面向对象的编程语言，因此可以定义可从其他类继承的类。定义的类可以有方法、字段、属性和子类。继承能使类建立在现有类的基础上，并重写选定的基类方法和属性。JScript 中的类类似于 C++ 和 C# 中的类，它与基于原型的对象有很大的不同。

本节内容

创建自己的类

描述如何定义具有字段、方法和构造函数的类。

高级类创建

描述如何定义具有属性的类，如何从类继承以及如何创建支持 expando 属性的类。

相关章节

JScript 对象

包括指向特定主题的链接，这些主题解释内部 JScript 对象的语法和用法。

JScript 修饰符

描述可用于控制类成员可见性的修饰符，并描述类是如何继承的以及类如何操作。

基于原型的对象

提供有关如何使用 JScript 基于原型的对象模型的指南，并且提供指向特定信息的链接，这些信息描述基于原型的对象的自定义构造函数和继承。

JScript 参考

列出 JScript 语言参考中涉及的元素，并提供指向特定主题的链接，这些主题解释了有关正确使用这些语言元素的幕后详细信息。

创建自己的类

class 语句定义类。默认情况下，类成员都可以公开访问，这就意味着任何可访问类的代码都可以访问类成员。有关更多信息，请参见 JScript 修饰符。

具有字段的类

字段定义对象所使用的数据，类似于基于原型的对象中的属性。下面是一个有两个字段的简单类的示例，使用 **new** 运算符来创建该类的一个实例：

```
class myClass {
   const answer : int = 42; // Constant field.
   var distance : double;   // Variable field.
}

var c : myClass = new myClass;
c.distance = 5.2;
print("The answer is " + c.answer);
print("The distance is " + c.distance);
```

该程序的输出为：

```
The answer is 42
The distance is 5.2
```

具有方法的类

类还可以包含方法，方法是类中所包含的函数。方法定义操作对象数据的功能。可以重新定义前面已定义的 `myClass` 类，使之包含一个方法。

```
class myClass {
   const answer : int = 42;         // Constant field.
   var distance : double;           // Variable field.
   function sayHello() :String {    // Method.
      return "Hello";
   }
}

var c : myClass = new myClass;
c.distance = 5.2;
print(c.sayHello() + ", the answer is " + c.answer);
```

该程序的输出为：

```
Hello, the answer is 42
```

具有构造函数的类

可以为类定义构造函数。构造函数是与类有着相同名称的方法，当使用 **new** 运算符创建类时就运行构造函数。可以不指定构造函数的返回类型。在本示例中，向 `myClass` 类添加一个构造函数。

```
class myClass {
   const answer : int = 42;         // Constant field.
   var distance : double;           // Variable field.
   function sayHello() :String {    // Method.
      return "Hello";
   }
   // This is the constructor.
   function myClass(distance : double) {
      this.distance = distance;
   }
```

```
}

var c : myClass = new myClass(8.5);
print("The distance is " + c.distance);
```

该程序的输出为:

```
The distance is 8.5
```

请参见

概念

高级类创建

其他资源

基于类的对象

JScript 对象

高级类创建

在定义 JScript 类时，可以为属性赋值，已定义的类可以随后从其他类继承这些属性。属性（像字段一样也是类成员）对数据的访问方式提供了更多控制。通过使用继承，一类可以扩展另一类（或在另一类上添加行为）。

可以定义一个类，使该类的实例支持 expando 属性。这就意味着基于类的对象可以有动态添加到此对象的属性和方法。基于类的 expando 对象可提供一些与基于原型的对象所具有的相同的功能。

具有属性的类

JScript 使用 **function get** 和 **function set** 语句来指定属性。可以指定这两种访问器的任何一种或同时指定两种来创建只读、只写或读写属性，尽管只写属性极为少见，且可能指示类的设计中有问题。

执行调用的程序访问属性的方式与访问字段的方式相同。主要的不同之处在于：对于属性，要使用 getter 和 setter 来执行访问，而对于字段是直接进行访问的。属性使类能够执行诸如检查只输入有效信息、跟踪读取或设置属性的次数、返回动态信息等功能。

属性通常常用来访问类的私有字段或受保护字段。私有字段用 **private** 修饰符来标记，只有该类的其他成员才能访问它们。受保护字段用 **protected** 修饰符来标记，只有该类或派生类的其他成员才能访问它们。有关更多信息，请参见 JScript 修饰符。

在本示例中，属性用于访问受保护的字段。该字段受到保护，以防止外部代码更改其值，同时允许派生类访问它。

```
class Person {
   // The name of a person.
   // It is protected so derived classes can access it.
   protected var name : String;

   // Define a getter for the property.
   function get Name() : String {
      return this.name;
   }

   // Define a setter for the property which makes sure that
   // a blank name is never stored in the name field.
   function set Name(newName : String) {
      if (newName == "")
         throw "You can't have a blank name!";
      this.name = newName;
   }
   function sayHello() {
      return this.name + " says 'Hello!'";
   }
}

// Create an object and store the name Fred.
var fred : Person = new Person();
fred.Name = "Fred";
print(fred.sayHello());
```

该代码的输出为：

```
Fred says 'Hello!'
```

如果为 `Name` 属性分配空名称，则会生成错误。

从类继承

当以另一个类为基础来定义类时，使用 **extends** 关键字。JScript 可扩展大多数符合公共语言规范 (CLS) 的类。使用 **extends** 关键字定义的类称为派生类，从中扩展的类称为基类。

在本示例中，定义了新的 `Student` 类，它扩展了前面示例中的 `Person` 类。Student 类重新使用了基类中定义的 `Name` 属性，但却定义了新的 `sayHello` 方法，重写了基类的 `sayHello` 方法。

```
// The Person class is defined in the code above.
```

```
class Student extends Person {
   // Override a base-class method.
   function sayHello() {
      return this.name + " is studying for finals.";
   }
}

var mary : Person = new Student;
mary.Name = "Mary";
print(mary.sayHello());
```

该代码的输出为:

```
Mary is studying for finals.
```

在派生类中重新定义方法不会更改基类中的相应方法。

Expando 对象

如果只是想使一般对象成为 expando，可使用 **Object** 构造函数。

```
// A JScript Object object, which is expando.
var o = new Object();
o.expando = "This is an expando property.";
print(o.expando);  // Prints This is an expando property.
```

如果想使某一个类成为 expando，可使用 **expando** 修饰符定义该类。只能使用索引 (**[]**) 表示法来访问 expando 成员；不能使用句点 (**.**) 表示法来访问它们。

```
// An expando class.
expando class MyExpandoClass {
   function dump() {
      // print all the expando properties
      for (var x : String in this)
         print(x + " = " + this[x]);
   }
}
// Create an instance of the object and add some expando properties.
var e : MyExpandoClass = new MyExpandoClass();
e["answer"] = 42;
e["greeting"] = "hello";
e["new year"] = new Date(2000,0,1);
print("The contents of e are...");
// Display all the expando properites.
e.dump();
```

该程序的输出为:

```
The contents of e are...
answer = 42
greeting = hello
new year = Sat Jan 1 00:00:00 PST 2000
```

请参见

概念

JScript 修饰符

创建自己的类

其他资源

基于类的对象

JScript 对象

基于原型的对象

由于 JScript 是面向对象的编程语言，因此可以用它来定义自定义构造函数和继承。构造函数提供了一种能力，使用户能设计和实现自己的基于原型的对象。继承能使基于原型的对象共享一组可动态添加或移除的通用属性和方法。

在许多情况下，应使用基于类的对象而不是基于原型的对象。基于类的对象可传递给用其他 .NET Framework 语言编写的方法。而且，基于类的对象还可提供类型安全并产生高效率的代码。

本节内容

用构造函数创建自己的对象

解释如何使用构造函数创建具有属性和方法的对象。

高级对象创建 (JScript)

阐释如何使用继承将一组通用属性和方法添加到用给定构造函数创建的对象。

相关章节

JScript 对象

包括指向特定主题的链接，这些主题解释内部 JScript 对象的语法和用法。

基于类的对象

提供如何使用 JScript 基于类的对象模型的指南，并且描述如何定义类（使用方法、字段和属性）、如何定义从其他类继承的类以及如何定义 expando 类。

JScript 参考

列出 JScript 语言参考中涉及的元素，并提供指向特定主题的链接，这些主题解释了有关正确使用这些语言元素的幕后详细信息。

用构造函数创建自己的对象

JScript 的一个强大功能是能够定义构造函数，以创建自定义的基于原型的对象，以便在您的脚本中使用。要创建基于原型的对象的实例，首先必须定义一个构造函数。此过程将创建一个新对象并将它初始化（创建属性并赋初始值）。当完成后，构造函数将返回对所构造对象的引用。在构造函数内部，创建的对象是通过 **this** 语句引用的。

具有属性的构造函数

下面的示例为 `pasta` 对象定义了一个构造函数。**this** 语句允许构造函数初始化该对象。

```
// pasta is a constructor that takes four parameters.
function pasta(grain, width, shape, hasEgg) {
   this.grain = grain;    // What grain is it made of?
   this.width = width;    // How many centimeters wide is it?
   this.shape = shape;    // What is the cross-section?
   this.hasEgg = hasEgg;  // Does it have egg yolk as a binder?
}
```

在定义对象构造函数后，使用 **new** 运算符创建对象的实例。此处使用 `pasta` 构造函数创建 `spaghetti` 和 `linguine` 对象。

```
var spaghetti = new pasta("wheat", 0.2, "circle", true);
var linguine = new pasta("wheat", 0.3, "oval", true);
```

可以动态地为对象的某个实例添加属性，但这些更改只影响这一个实例。

```
// Additional properties for spaghetti. The properties are not added
// to any other pasta objects.
spaghetti.color = "pale straw";
spaghetti.drycook = 7;
spaghetti.freshcook = 0.5;
```

如果想为对象的所有实例都添加额外的属性而不修改构造函数，则可以将该属性添加到构造函数的原型对象。有关更多信息，请参见 高级对象创建 (JScript)。

```
// Additional property for all pasta objects.
pasta.prototype.foodgroup = "carbohydrates";
```

具有方法的构造函数

可以在对象的定义中包含方法（函数）。做到这一点的一种方法是，在构造函数中包含一个属性，而此属性引用其他地方定义的函数。像构造函数一样，这些函数也用 **this** 语句引用当前的对象。

下面的示例扩展前面定义的 `pasta` 构造函数以包含 **toString** 方法，当函数显示对象的值时将会调用此方法。（通常，在要求字符串的条件下使用对象时，JScript 将使用对象的 **toString** 方法。很少需要显式调用 **toString** 方法。）

```
// pasta is a constructor that takes four parameters.
// The properties are the same as above.
function pasta(grain, width, shape, hasEgg) {
   this.grain = grain;    // What grain is it made of?
   this.width = width;    // How many centimeters wide is it?
   this.shape = shape;    // What is the cross-section?
   this.hasEgg = hasEgg;  // Does it have egg yolk as a binder?
   // Add the toString method (defined below).
   // Note that the function name is not followed with parentheses;
   // this is a reference to the function itself, not a function call.
   this.toString = pastaToString;
}

// The function to display the contents of a pasta object.
function pastaToString() {
   return "Grain: " + this.grain + "\n" +
```

```
            "Width: " + this.width + " cm\n" +
            "Shape: " + this.shape + "\n" +
            "Egg?:  " + Boolean(this.hasEgg);
}

var spaghetti = new pasta("wheat", 0.2, "circle", true);
// Call the method explicitly.
print(spaghetti.toString());
// The print statement takes a string as input, so it
//  uses the toString() method to display the properties
// of the spaghetti object.
print(spaghetti);
```

这样将显示下面的输出。

```
Grain: wheat
Width: 0.2 cm
Shape: circle
Egg?:  true
Grain: wheat
Width: 0.2 cm
Shape: circle
Egg?:  true
```

请参见

其他资源

基于原型的对象

JScript 对象

高级对象创建 (JScript)

JScript 支持对自定义的基于原型的对象的继承。通过继承，基于原型的对象可以共享一组可动态添加或移除的通用属性和方法。而且，各个对象都可以重写默认的行为。

创建基于原型的对象

要创建基于原型的对象的实例，首先必须定义一个构造函数。有关更多信息，请参见用构造函数创建自己的对象。一旦编写了此构造函数，就可以使用 **prototype** 对象（它本身就是每个构造函数的一项属性）的属性来创建继承属性和共享方法。构造函数可以向对象提供实例特定的信息，而 **prototype** 对象可以向对象提供对象特定的信息和方法。

注意

若要影响对象的所有实例，必须对构造函数的 **prototype** 对象进行更改。更改对象的一个实例的 **prototype** 属性对同一对象的其他实例没有影响。

由于 **prototype** 对象的属性和方法是通过引用对象的每个实例的内容而复制的，因此所有的实例都可以访问同样的信息。可以更改一个实例中一个原型属性的值，以重写默认值，但此更改只影响这一个实例。下面的实例使用了自定义的构造函数 `Circle`。**this** 语句使此方法能够访问该对象的各个成员。

```
// Define the constructor and add instance specific information.
function Circle (radius) {
    this.r = radius;  // The radius of the circle.
}
// Add a property the Circle prototype.
Circle.prototype.pi = Math.PI;
function ACirclesArea () {
   // The formula for the area of a circle is pi*r^2.
   return this.pi * this.r * this.r;
}
// Add a method the Circle prototype.
Circle.prototype.area = ACirclesArea;
// This is how you would invoke the area function on a Circle object.
var ACircle = new Circle(2);
var a = ACircle.area();
```

利用这一原则，可以为现有的构造函数（它们都具有原型对象）定义其他属性。只有在关闭快速模式时这才有效。有关更多信息，请参见 /fast。

例如，如果想要移除字符串前面和后面的空格（类似于 Visual Basic 中的 **Trim** 函数），可以在 **String** 原型对象上创建您自己的方法，则脚本中的所有字符串都将自动继承此方法。下面示例使用正则表达式来移除这些空格。有关更多信息，请参见正则表达式对象。

```
// Add a function called trim as a method of the prototype
// object of the String constructor.
String.prototype.trim = function() {
   // Use a regular expression to replace leading and trailing
   // spaces with the empty string
   return this.replace(/(^\s*)|(\s*$)/g, "");
}

// A string with spaces in it
var s = "    leading and trailing spaces    ";
print(s + " (" + s.length + ")");

// Remove the leading and trailing spaces
s = s.trim();
print(s + " (" + s.length + ")");
```

当使用 **/fast-** 标志编译该程序后，该程序的输出为：

```
     leading and trailing spaces     (35)
```

```
  leading and trailing spaces (27)
```

请参见

其他资源

基于原型的对象

JScript 对象

JScript 修饰符

JScript 修饰符可更改类、接口或者类或接口的成员的行为和可见性。修饰符可以在定义类和接口时使用，但它们通常并不是必需的。

可见性修饰符

可见性修饰符对外部代码如何访问类、接口以及它们的成员加以限制。可以使用一些限制条件，通过避免调用专用的内部方法和字段，来促进建立良好的面向对象的编程习惯。

默认情况下，可访问某个类的任何代码都可以访问该类的任何成员。使用可见性修饰符可以有选择地阻止外部代码访问特定的类成员，只允许同一个包的类访问成员，或只允许派生类访问类成员。

可见性修饰符不能应用于全局函数或变量。只有 **protected** 和 **internal** 可见性修饰符可以在一起使用。

可见性修饰符	有效范围	含义
public	类、类成员、接口、接口成员或枚举	成员对于任何可访问类而对可见性没有限制的代码都是可见的。默认情况下，在 JScript 中，类、接口以及它们的成员都是公共的。
private	类成员	成员仅在声明它的类内可见。对派生类不可见。当前类之外的代码无法访问私有成员。
protected	类成员	成员仅在声明它的类内可见，并对该类的任何派生类可见。受保护的修饰符不能对包范围的类使用，但可以对嵌套的类使用。
internal	类、类成员、枚举	类、类成员或枚举在声明它们的包内任何地方都可见。在此包外不可见。

继承修饰符

继承修饰符控制派生类的方法和属性如何重写基类中的方法和属性。通过使用此控制，可以对派生类的方法是否将重写您所创建的类进行管理。

默认情况下，派生类的方法将重写基类方法，除非在派生类中使用了版本安全的 **hide** 属性。该属性可阻止重写。使用继承修饰符使您能控制某些特定的方法总是可以重写还是永远不能重写。

在某些条件下，可能需要确保不重写某种基类方法。例如，如果在包内定义了一个类，就可以使用 **final** 修饰符确保派生类将不能更改此类的方法和属性。

另一方面，您也可能希望让您的类重写某些方法。例如，可以创建一个类来提供一些基本功能，但对某些方法则使用 **abstract** 修饰符。实现这些抽象方法是派生类编写器的任务。

版本安全修饰符也管理重写，但它是从派生类一边（而不是从基类一边）来管理的。只有当版本安全修饰符所要重写的基类方法没有继承修饰符时，版本安全修饰符才有效果。

不能将两种继承修饰符组合在一起，也不能将继承修饰符与 **static** 修饰符组合在一起。

继承修饰符	有效范围	含义
abstract	类、方法或属性	用于方法或属性，此修饰符指示成员没有实现。用于类，此修饰符指示有一个或多个未实现的方法。不能使用 **new** 关键字实例化抽象类或包含抽象成员的类，但它可以用作基类。
final	类、方法或属性	用于不能被扩展的类或不能被重写的方法。使用 **final** 可防止派生类通过重写重要的函数而更改此类的行为。具有 **final** 修饰符的方法可以被隐藏或重载，但不能被重写。

版本安全修饰符

版本安全修饰符控制可重写基类中的方法的派生类方法。通过使用这种控制，您就可以管理您所创建的类是否将重写基类中的方法。

默认情况下，派生类的方法将重写基类中的方法，虽然派生类的定义中的继承修饰符也可以阻止重写。使用版本安全修饰符使您能控制某些特定的方法是否会被重写。

在某些条件下，可能需要确保不重写基类方法。例如，可以扩展一个类以更改基类方法的行为。如果不希望这些方法在基类中被重写，就可以使用 **hide** 修饰符对这些方法作出声明。

另一方面，您也可能想重写某些基类方法。例如，您可能想更改类的某些方法而不修改该类。通过扩展该类并使用 **override** 修饰符作为方法声明，可以使新方法重写基类。

版本安全修饰符的成功使用取决于基类方法的声明是否使用了继承修饰符。不能重写用 **final** 修饰符标记的基类方法；也不能隐藏用 **abstract** 修饰符标记的基类方法，除非为抽象的基类方法提供了显式实现。

不能将两种版本安全修饰符组合在一起，也不能将版本安全修饰符与 **static** 修饰符组合在一起。当您以版本安全模式运行时，对每个重写基类方法的方法，仅可使用一个版本安全修饰符。

版本安全修饰符	有效范围	含义
hide	方法或属性	成员不重写基类中有相同名称的成员。
override	方法或属性	默认情况下，成员可重写基类中有相同名称的成员。

expando 修饰符

expando 修饰符能使得基于类的对象的行为就像 JScript 对象一样。可以将方法和属性动态添加到 expando 对象。有关更多信息，请参见基于原型的对象。

可以独立于其他修饰符使用 **expando** 修饰符。

修饰符	有效范围	含义
expando	类或方法	用于类，给类提供一个默认的和带索引的属性，该属性能够存储和检索动态属性 (expando)。用于方法，指示它是一个 expando 对象的构造函数。

static 修饰符

static 修饰符指明类的成员属于类本身而不属于类的实例。因此，类特定的数据和方法可能不与任何特定的实例相关联。

不能将 **static** 修饰符与任何版本安全修饰符或继承修饰符组合。

修饰符	有效范围	含义
static	方法、属性、字段或类	用于方法，指示可以在没有类实例的情况下调用它。用于属性和字段，指定所有实例共享一个副本。不应将 **static** 修饰符与 **static** 语句相混淆，后者表示初始化类的代码。

请参见

参考

class 语句
interface 语句
function 语句
function get 语句
function set 语句
var 语句
const 语句
static 语句

其他资源

修饰符

JScript 运算符

JScript 具有丰富多样的运算符，包括算术运算符、逻辑运算符、位运算符、赋值运算符以及一些其他运算符。运算符将简单的表达式组合在一起构成更复杂的表达式。

本节内容

运算符摘要

提供几个关于 JScript 运算符的表，这些运算符是按运算符的类型分组的。

运算符优先级

提供一个列出运算符及其对应优先级的表，并提供一个示例说明运算符优先级如何起作用。

按位运算符强制

描述控制位运算符的操作数强制的规则。若要使二进制格式的操作数彼此兼容并且与位运算符兼容，必须使用强制。

相关章节

JScript 赋值与相等

解释如何使用赋值、相等和全等运算符。

JScript 中的强制

解释强制的概念、如何使用强制以及强制的限制。

运算符 (JScript)

列出指向有关 JScript 中运算符的参考主题的链接。

运算符摘要

以下各表列出了各种 JScript 运算符。说明栏的每个名称都链接到解释正确语法和用法的相应主题。

算术运算符

说明	符号
加法	+
递减	--
除法	/
递增	++
取模算法	%
乘法	*
减法	-
一元求反	-

所有算术运算符都对数值数据执行计算。当任一操作数为字符串时，加法运算符还执行字符串串联操作。

逻辑运算符

说明	符号
相等	==
大于或等于	>=
大于	>
恒等	===
包含在其中	in
不相等	!=
小于或等于	<=
小于	<
逻辑"与"	&&
逻辑"非"	!
逻辑"或"	||
不全等	!==

逻辑运算符返回**Boolean**值。根据运算符的不同，值可以表示比较、测试或组合的结果。

位运算符

说明	符号
按位"与"	&
按位左移	<<
按位"非"	~
按位"或"	|
按位右移	>>
按位"异或"	^
无符号右移	>>>

位操作符对操作数的二进制表示形式执行操作。如果操作数相互不兼容，将强迫使它们成为适当的类型。有关更多信息，请参见按位运算符强制。

赋值运算符

说明	符号
赋值	=
复合加法赋值	+=
复合按位"与"赋值	&=
复合按位"或"赋值	|=
复合按位"异或"赋值	^=
复合除法赋值	/=
复合左移赋值	<<=
复合取模赋值	%=
复合乘法赋值	*=
复合右移赋值	>>=
复合减法赋值	-=
复合无符号右移赋值	>>>=

所有的赋值运算符都返回赋给左操作数的值。

其他运算符

说明	符号
逗号	,
条件（三元）	?:
删除	delete

确定实例	instanceof
新建	new
Typeof	typeof
返回空	void

请参见

概念

运算符优先级

其他资源

JScript 运算符

运算符优先级

运算符优先级是 JScript 中的一套规则，用于控制编译器计算表达式时执行运算的顺序。先执行具有较高优先级的运算，然后执行较低优先级的运算。例如，先执行相乘，再执行相加。

优先级表

下表列出了 JScript 运算符，并按优先级顺序从高到低排列。

优先级	计算顺序	运算符	说明
15	从左到右	., [], ()	字段访问、数组索引、函数调用和表达式分组
14	从右到左	++, --, -, ~, !, delete, new, typeof, void	一元运算符、返回数据类型、对象创建、未定义的值
13	从左到右	*, /, %	相乘、相除、求余数
12	从左到右	+, -	相加、字符串串联、相减
11	从左到右	<<, >>, >>>	移位
10	从左到右	<, <=, >, >=, instanceof	小于、小于或等于、大于、大于或等于、是否为特定类的实例
9	从左到右	==, !=, ===, !==	相等、不相等、全等，不全等
8	从左到右	&	按位"与"
7	从左到右	^	按位"异或"
6	从左到右	|	按位"或"
5	从左到右	&&	逻辑"与"
4	从左到右	||	逻辑"或"
3	从右到左	?:	条件
2	从右到左	=, *OP*=	赋值、复合赋值
1	从左到右	,（逗号）	多个计算

表达式中的圆括号改变由运算符优先级确定的计算顺序。这就是说，先计算完圆括号内的表达式，然后再将它的值用于表达式的其余部分。

例如：

```
z = 78 * (96 - 3 + 45)
```

在上面的表达式中有 5 个运算符：=、*、()、- 和 +。按照运算符优先级规则，按以下顺序计算：()、-、+、*、=。

1. 最先计算圆括号内的表达式。在圆括号内，有加法和减法运算符。这两个运算符的优先级相同，按从左向右的顺序计算它们。先从 96 减去数字 3，结果是 93。然后将数字 45 与 93 相加，得出的值是 138。
2. 然后计算相乘。用数字 138 乘以数字 78，得出的值为 10764。
3. 最后进行赋值。将数字 10764 赋给 z。

请参见

概念

运算符摘要

其他资源

JScript 运算符

按位运算符强制

此版本的 JScript 中的位运算符与以前版本的 JScript 中的位运算符完全兼容。此外，JScript 运算符还可以用于新的数值数据类型。位运算符的行为取决于数据的二进制表示形式，因此了解运算符如何强制数据类型是非常重要的。

可以将三种类型的参数传递到位运算符：早期绑定的变量、后期绑定的变量以及常值数据。早期绑定的变量是使用显式类型批注定义的变量。后期绑定的变量是包含数值数据的、类型为 **Object** 的变量。

按位与 (&)、或 (|) 非 XOR (^) 运算符

如果一个操作数是后期绑定的变量或者两个操作数均是常值数据，则将两个操作数被强制为 **int** (**System.Int32**)，然后执行运算，返回的值为 **int**。

如果两个操作数均是早期绑定的变量，或者其中一个操作数是常值数据，而另一个操作数是早期绑定的变量，则执行更多的步骤。将两个操作数强制为由以下两个条件确定的类型：

- 如果操作数均非整型，则将两个操作数强制为 **int**。
- 如果只有一个操作数是整型，则将非整型操作数强制为整型或 **int**（取决于哪种类型更长）。
- 如果一个操作数更长，则操作数强制后的类型与较长操作数具有相同的长度。
- 如果其中一个操作数是无符号的，则操作数强制后的类型也是无符号的。否则，强制的类型是有符号的。

然后，将操作数强制为适当的类型，执行位运算并返回结果。结果的数据类型与被强制的操作数的类型相同。

当将整数用于位运算符和早期绑定的变量时，将整数解释为 **int**、**long**、**ulong** 或 **double**（取决于哪种类型是表示该数字的最小类型）。**decimal** 常值被强制为 **double**。可能需要根据上述规则对常值数据的类型进行进一步的强制。

按位“非”(~) 运算符

如果操作数是后期绑定的变量、早期绑定的浮点变量或者是常值数据，则将它强制为 **int** (**System.Int32**)，然后执行“非”运算，返回值为 **int**。

如果操作数是早期绑定的整型数据类型，则执行“非”运算，返回的类型与操作数的类型相同。

按位左移 (<<)、右移 (>>) 运算符

如果左操作数是后期绑定的变量、早期绑定的浮点变量或者是常值数据，则将它强制为 **int** (**System.Int32**)。否则，左操作数是早期绑定的整型数据类型，并且不执行强制。右操作数始终被强制为整型数据类型。然后，对被强制的值执行移位运算，返回的结果与左操作数（如果是早期绑定的变量）或类型 **int** 具有相同的类型。

无符号右移 (>>>) 运算符

如果左操作数是后期绑定的变量、早期绑定的浮点变量或者是常值数据，则将它强制为 **uint** (**System.UInt32**)。否则，左操作数是早期绑定的整型数据类型，并将它强制为相同大小的无符号类型。例如，将 **int** 强制为 **uint**。右操作数始终被强制为整型数据类型。然后，对被强制的值执行移位运算，返回的结果与被强制的左操作数（如果是早期绑定的变量）或类型 **uint** 具有相同的类型。

无符号右移的结果始终很小，可以将它存储在有符号的返回类型版本中，而不会产生溢出。

请参见

概念

运算符优先级
类型转换
JScript 中的强制
数值数据

其他资源

JScript 运算符

JScript 函数

JScript 函数可以执行操作、返回值或者执行操作并返回值。例如，函数可以显示当前的时间，并返回表示时间的字符串。函数也称为“全局方法”。

函数将几个运算组合到一起并使用一个名称，这使代码更加简单并且可以重复使用。可以编写一组语句，对它进行命名，然后通过调用其名称并给它传递必要的信息来执行整组语句。

若要向函数传递信息，请将信息放在函数名称后面的圆括号内。传递给函数的信息项也称为“参数”。有些函数不带参数，而另外一些函数则带一个或多个参数。在某些函数中，参数的数量取决于使用该函数的方式。

JScript 支持两种函数，即该语言内置的函数以及用户创建的函数。

本节内容

类型批注

描述类型批注的概念，以及如何在函数定义中使用它来控制输入和输出数据类型。

用户定义的 JScript 函数

阐释如何在 JScript 中定义新的函数，以及如何使用这些函数。

递归

解释递归的概念，并阐释如何编写递归函数。

相关章节

JScript 运算符

列出计算、逻辑、按位、赋值和杂项运算符并提供指向特定信息的链接，这些信息解释如何有效地使用这些运算符。

JScript 数据类型

包含指向特定主题的链接，这些主题解释如何使用 JScript 中的基元数据类型、引用数据类型和 .NET Framework 数据类型。

JScript 中的强制

解释强制的概念、如何使用强制以及强制的限制。

function 语句

描述声明函数的语法。

类型批注

函数中的类型批注指定函数参数的所需类型、返回数据的所需类型或者这两者的所需类型。如果没有给出函数的参数的类型批注，则参数的类型将为 **Object**。另外，如果没有指定函数的返回类型，则编译器将推断出适当的返回类型。

使用类型批注

使用函数参数的类型批注有助于确保函数只接受它能处理的数据。显式地声明函数的返回类型可提高代码的可读性，因为函数返回的数据的类型一目了然。

下面的示例阐释函数参数和函数返回类型的类型批注的使用。

```
// Declare a function that takes an int and returns a String.
function Ordinal(num : int) : String{
   switch(num % 10) {
   case 1: return num + "st";
   case 2: return num + "nd";
   case 3: return num + "rd";
   default: return num + "th";
   }
}

// Test the function.
print(Ordinal(42));
print(Ordinal(1));
```

该程序的输出为：

```
42nd
1st
```

如果传递到 `Ordinal` 函数的参数不能强制为整型，就会产生类型不匹配错误。例如，`Ordinal(3.14159)` 将失败。

请参见

参考

function 语句

其他资源

JScript 函数

数据类型 (JScript)

用户定义的 JScript 函数

虽然 JScript 包含很多内置函数，但是您也可以创建自己的函数。函数定义由函数语句和 JScript 语句块组成。

定义自己的函数

下面的示例中的 checkTriplet 函数将三角形的边长作为其参数。该函数通过检查三个数是否构成毕达哥拉斯三元数组来计算三角形是否为直角三角形，毕达哥拉斯三元数组的构成条件为：直角三角形斜边长度的平方等于其余两边长度的平方和。checkTriplet 函数调用另外两个函数之一来进行实际测试。

注意，在浮点测试版本中，将一个很小的数 (epsilon) 用作测试变量。由于浮点计算的不确定性和舍入误差，所以无法直接测试三个数是否构成毕达哥拉斯三元数组，除非已知所有这三个值均为整数。因为直接测试更准确，所以此示例中的代码确定它是否适于直接测试，如果适合，则使用直接测试。

当定义这些函数时，没有使用类型批注。对于此处的情况，checkTriplet 函数采用整型和浮点数据类型很有用。

```
const epsilon = 0.00000000001; // Some very small number to test against.

// Type annotate the function parameters and return type.
function integerCheck(a : int, b : int, c : int) : boolean {
   // The test function for integers.
   // Return true if a Pythagorean triplet.
   return ( ((a*a) + (b*b)) == (c*c) );
} // End of the integer checking function.

function floatCheck(a : double, b : double, c : double) : boolean {
   // The test function for floating-point numbers.
   // delta should be zero for a Pythagorean triplet.
   var delta = Math.abs( ((a*a) + (b*b) - (c*c)) * 100 / (c*c));
   // Return true if a Pythagorean triplet (if delta is small enough).
   return (delta < epsilon);
} // End of the floating-poing check function.

// Type annotation is not used for parameters here. This allows
// the function to accept both integer and floating-point values
// without coercing either type.
function checkTriplet(a, b, c) : boolean {
   // The main triplet checker function.
   // First, move the longest side to position c.
   var d = 0; // Create a temporary variable for swapping values
   if (b > c) { // Swap b and c.
      d = c;
      c = b;
      b = d;
   }
   if (a > c) { // Swap a and c.
      d = c;
      c = a;
      a = d;
   }

   // Test all 3 values. Are they integers?
   if ((int(a) == a) && (int(b) == b) && (int(c) == c)) { // If so, use the precise check.
      return integerCheck(a, b, c);
   } else { // If not, get as close as is reasonably possible.
      return floatCheck(a, b, c);
   }
} // End of the triplet check function.

// Test the function with several triplets and print the results.
// Call with a Pythagorean triplet of integers.
print(checkTriplet(3,4,5));
// Call with a Pythagorean triplet of floating-point numbers.
print(checkTriplet(5.0,Math.sqrt(50.0),5.0));
// Call with three integers that do not form a Pythagorean triplet.
print(checkTriplet(5,5,5));
```

该程序的输出为:

```
true
true
false
```

请参见

参考

function 语句

其他资源

JScript 函数

JScript 数据类型

递归

递归是一项非常重要的编程技巧，它使函数调用其本身。示例之一就是阶乘的计算。0 的阶乘被明确地定义为 1。*n*（大于 0 的整数）的阶乘是 1 到 *n* 之间所有整数的乘积。

使用递归

以下段落是用文字定义的一个阶乘计算函数。

如果数字小于零，则将其拒绝。如果数字不是整数，则将其拒绝。如果数字为零，其阶乘则为一。如果数字大于零，则将其乘以下一个更小数字的阶乘。

若要计算任一个大于零的数字的阶乘，必须至少计算另外一个数字的阶乘。函数在对当前数字执行计算之前，必须先对小于当前数字的相邻数字调用其自身。这就是递归的示例。

递归和迭代（循环）密切相关，即函数可以使用递归或迭代返回相同的结果。通常，某个计算适用于一种技巧或另一种技巧，您只须选择最自然或最理想的方法。

虽然递归的用处很大，但是如果使用不慎，创建的递归函数就可能从不返回结果并且不能到达终点。这种递归导致计算机执行“*无限*”循环。下面是一个示例：忽略阶乘计算文字描述中的第一项规则（有关负数的规则），然后计算任意负数的阶乘。这种计算失败的原因是：若要计算 -24 的阶乘，必须计算 -25 的阶乘，要计算 -25 的阶乘，必须先计算 -26 的阶乘，依此类推。显然，这种计算永远得不出结果。

递归可能出现的另一个问题是：递归函数可能用尽所有可用的资源（如系统内存、堆栈空间等等）。每次递归函数调用自身（或调用另一个函数，而另一个函数又调用原来的函数），递归函数就会占用一些资源。当递归函数退出时，就会释放这些资源，但是函数的递归层次过多，就会用尽所有可用的资源。发生这种情况时，就会引发异常。

因此，谨慎设计递归函数是非常重要的。如果怀疑可能出现递归过多（或无限递归）的情况，则设计函数时就应加入计算函数调用其自身的次数的功能，并设置调用次数限制。如果函数调用自身的次数超过阈值，则函数可以自动退出。迭代的最大次数的最佳取值取决于递归函数。

下面又是一个阶乘函数，这一次是用 JScript 代码编写的。还使用了类型批注，使函数只接受整数。如果传递的数字无效（即小于零的数），则 throw 语句就会产生一个错误。否则，使用递归函数计算阶乘。递归函数采用两个参数，一个用作阶乘参数，另一个用作跟踪当前递归层次的计数器。如果计数器没有达到最大递归层次，则返回原始数字的阶乘。

```
function factorialWork(aNumber : int, recursNumber : int ) : double {
   // recursNumber keeps track of the number of iterations so far.
   if (aNumber == 0) {  // If the number is 0, its factorial is 1.
      return 1.;
   } else {
      if(recursNumber > 100) {
         throw("Too many levels of recursion.");
      } else {  // Otherwise, recurse again.
         return (aNumber * factorialWork(aNumber - 1, recursNumber + 1));
      }
   }
}

function factorial(aNumber : int) : double {
   // Use type annotation to only accept numbers coercible to integers.
   // double is used for the return type to allow very large numbers to be returned.
   if(aNumber < 0) {
      throw("Cannot take the factorial of a negative number.");
   } else {  // Call the recursive function.
      return  factorialWork(aNumber, 0);
   }
}

// Call the factorial function for two values.
print(factorial(5));
print(factorial(80));
```

该程序的输出为：

```
120
```

```
7.156945704626378e+118
```

请参见

概念

类型批注

其他资源

JScript 函数

JScript 中的强制

JScript 可以对不同类型的值执行运算，而不会导致编译器引发异常。JScript 编译器在执行运算之前，自动将一种数据类型更改（强制）为另一种数据类型。其他语言则使用更为严格的控制强制的规则。

强制详细信息

除非证明强制永远不会成功，否则编译器允许使用所有的强制。任何 *可能* 失败的强制在编译时生成一个警告，并且很多强制在失败时会产生运行时错误。例如：

运算	结果
将数字与字符串相加	将数字强制为字符串。
将布尔值与字符串相加	将布尔值强制为字符串。
将数字与布尔值相加	将布尔值强制为数字。

请看下面的示例。

```
var x = 2000;      // A number.
var y = "Hello";   // A string.
x = x + y;         // the number is coerced into a string.
print(x);          // Outputs 2000Hello.
```

若要将字符串显式转换为整数，可以使用 **parseInt** 方法。有关更多信息，请参见 parseInt 方法。若要将字符串显式转换为数字，可以使用 **parseFloat** 方法。有关更多信息，请参见 parseFloat 方法。请注意，用于比较时，字符串被自动转换为等效的数字，但在相加（串联）时，仍保留为字符串。

因为 JScript 也是一种强类型的语言，所以也可以使用另一种强制机制。新机制使用目标类型名称，就好像它是一个将要转换的表达式作为参数的函数。这对于所有 JScript 基元类型、JScript 引用类型和 .NET Framework 类型都适用。

例如，以下代码将一个整型值转换为布尔值：

```
var i : int = 23;
var b : Boolean;
b = i;
b = Boolean(i);
```

因为 `i` 的值是一个非零值，所以 `b` 为 **true**。

新强制机制还适用于许多用户定义的类型。但是，由于 JScript 在转换不相似的类型时可能曲解用户的意图，所以以用户定义的类型作为强制源或强制目标的某些强制可能不能正常工作。当被转换的类型包含几个值时，尤其如此。例如，以下代码创建两个类（类型）。一个类包含单个变量 `i`（一个整数）。另一个类包含 3 个变量（`s`、`f` 和 `d`），每个变量具有不同的类型。在最后一条语句中，JScript 无法确定如何将第一种类型的变量转换为第二种类型。

```
class myClass {
   var i : int = 42;
}
class yourClass {
   var s : String = "Hello";
   var f : float = 3.142;
   var d : Date = new Date();
}
// Define a variable of each user-defined type.
var mine : myClass = new myClass();
var yours : yourClass;

// This fails because there is no obvious way to convert
// from myClass to yourClass
yours = yourClass(mine);
```

请参见

概念

类型转换

按位运算符强制

其他资源

JScript 函数

JScript 条件结构

通常，JScript 中的语句是按它们在脚本中出现的顺序执行的。这称为"顺序执行"，它是程序流程的默认方向。

顺序执行的一种替代方法是，当脚本遇到某种条件时就会将程序流程转移到脚本的另一部分。即，不是按顺序执行下一条语句，而是执行另一位置的一条语句。另一种替代方法称为"*迭代*"，它多次重复执行相同的语句序列。通常，迭代是使用循环实现的。

本节内容

条件语句

描述条件语句的概念，并解释用作条件语句的几种常见表达式类型。

控制结构

描述 JScript 提供的两种类型的控制结构，即选择控制结构和重复控制结构，以及两者的使用方法。

条件语句的使用

通过提供 **if** 语句和 **do**...**while** 循环的示例，阐释某些条件语句的用法。

条件运算符

描述如何使用条件运算符以及条件运算符与 **if**...**else** 语句之间的关系。

JScript 中的循环

介绍 JScript 中循环的概念，并列出指向特定信息的链接，这些信息解释在 JScript 代码中如何使用循环结构。

相关章节

JScript 数据类型

包含指向特定主题的链接，这些主题解释如何使用 JScript 中的基元数据类型、引用数据类型和 .NET Framework 数据类型。

JScript 参考

列出 JScript 语言参考中涉及的元素，并提供指向特定主题的链接，这些主题解释了有关正确使用这些语言元素的幕后详细信息。

条件语句

默认情况下，JScript 代码中的指令是按顺序执行的。但是，根据特定的条件改变逻辑顺序并将控制转移到代码的非顺序部分可能很有用。控制结构根据条件语句测试是 true 还是 false，将程序控制转移到两个位置之一。可以将可强制为布尔值的任何表达式用作条件语句。下面给出了一些常用的条件语句。

相等和全等

条件语句中的相等运算符 (==) 检查传递给它的两个参数是否具有相同的值，如有必要，进行类型转换以进行比较。全等运算符 (===) 比较两个表达式的值和类型；仅当两个操作数的值和数据类型都相同时，才返回 true。注意，全等运算符不区分不同的数值数据类型。

以下 JScript 代码将相等运算符与使用它的 **if** 语句结合起来。有关更多信息，请参见控制结构。

```
function is2000(x) : String {
   // Check if the value of x can be converted to 2000.
   if (x == 2000) {
      // Check is the value of x is strictly equal to 2000.
      if(x === 2000)
         print("The argument is number 2000.");
      else
         print("The argument can be converted to 2000.");
   } else {
      print("The argument is not 2000.");
   }
}
// Check several values to see if they are 2000.
print("Check the number 2000.");
is2000(2000);
print("Check the string \"2000\".");
is2000("2000")
print("Check the number 2001.");
is2000(2001);
```

下面是该代码的输出。

```
Check the number 2000.
The argument is number 2000.
Check the string "2000".
The argument can be converted to 2000.
Check the number 2001.
The argument is not 2000.
```

不相等和不全等

不相等运算符 (**!=**) 返回与相等运算符相反的结果。如果操作数具有相同的值，则不相等运算符返回 **false**；否则返回 **true**。相似地，不全等运算符 (**!==**) 返回与全等运算符相反的结果。

请看以下 JScript 代码示例，其中使用不相等运算符控制 **while** 循环。有关更多信息，请参见控制结构。

```
var counter = 1;
// Loop over the print statement while counter is not equal to 5.
while (counter != 5) {
   print(counter++);
}
```

下面是该代码的输出。

```
1
2
3
4
```

比较

如果一条数据具有特定的值，则可以使用相等或不相等运算符。但是，在某些情况下，代码可能需要检查某个值是否在特定范围内。在这些情况下，应该使用关系运算符：小于 (<)、大于 (>)、小于或等于 (<=) 以及大于或等于 (>=)。

```
if(tempInCelsius < 0)
   print("Water is frozen.")
else if(tempInCelsius > 100)
   print("Water is vapor.");
else
   print("Water is liquid.);
```

短路

如果要将几个条件放在一起测试，并且知道其中一个条件比其他条件更可能会通过或失败，则可以使用一种称为“短路计算”的功能加快脚本的执行速度，并避免出现可能产生错误的不利情况。当 JScript 计算某个逻辑表达式时，它只计算为得出结果而必须计算的子表达式。

逻辑“与”(**&&**) 运算符先计算传递给它的左侧表达式。如果该表达式转换为 **false**，则无论右侧表达式的值是多少，逻辑“与”运算符都不可能为 **true**。因此，不需要对右侧表达式进行计算。

例如，在表达式 `((x == 123) && (y == 42))` 中，JScript 先检查 `x` 是否为 123。如果不是，则不对 `y` 进行测试，JScript 返回值 **false**。

同样，逻辑或运算符 (**||**) 先计算左侧的表达式，如果它转换为 **true**，则不再计算右侧的表达式。

当要测试的条件涉及执行函数调用或其他复杂表达式时，短路是很有用的。对于逻辑“或”运算符，若要最有效地运行脚本，先测试最可能为 **true** 的条件。对于逻辑“与”运算符，先测试最可能为 **false** 的条件。

以下是说明采用这种方式设计脚本的优点的一个例子，如果 **runfirst()** 的返回值转换为 **false**，就不会执行 **runsecond()**。

```
if ((runfirst() == 0) || (runsecond() == 0)) {
   // some code
}
```

以下是说明采用这种方式设计脚本的优点的另一个例子，如果 **runfirst()** 的返回值转换为 **false**，就不会执行 **runsecond()**。

```
if ((x == 0) && (y/x == 5)) {
   // some code
}
```

其他

任何可以转换为布尔值的表达式都可以用作条件语句。例如，可以使用类似下面的表达式：

```
if (x = y + z) // This may not do what you expect - see below!
```

注意，以下代码并“*不*”检查 `x` 是否与 `y + z` 相等，这是因为此语法只使用单个等号(赋值)。相反，以上代码将值 `y + z` 赋给变量 `x`，然后检查是否可以将整个表达式的结果(`x` 的值)转换为值 **true**。若要检查 `x` 是否等于 `y + z`，可使用以下代码。

```
if (x == y + z) // This is different from the code above!
```

请参见

概念

布尔型数据

其他资源

JScript 条件结构

JScript 数据类型

JScript 参考

运算符 (JScript)

控制结构

对于除 **switch** 语句之外的所有控制结构，程序控制的转移是基于判定进行的，判定的结果是一个真实性语句（返回布尔值 **true** 或 **false**）。创建一个表达式，然后测试其结果是否为 **true**。有两种主要的程序控制结构。

选择控制结构

选择结构通过在程序中创建一个交叉点（类似道路的分岔），来指定程序流程的可能方向。JScript 中有 4 种选择结构。

- 单选结构 (**if**)
- 双选结构 (**if...else**)
- 多选结构 (**switch**)
- 内联条件运算符 **?:**

重复控制结构

重复结构指定当某种条件保持为 true 时重复执行某个操作。当满足控制语句的条件时（通常，经过特定次数的迭代后），控制转到重复结构之外的下一条语句。JScript 中有 4 种重复结构。

- 在循环顶部测试表达式 (**while**)
- 在循环底部测试表达式 (**do...while**)
- 对对象属性或数组元素进行操作 (**for...in**)
- 由计数器控制的重复 (**for**)

复合控制结构

复杂的脚本嵌套和堆叠选择控制结构和重复控制结构。

异常处理提供了另一种控制程序流程的方法，但此处不讲述它。有关更多信息，请参见 try...catch...finally 语句。

请参见

其他资源

JScript 条件结构

JScript 参考

条件语句的使用

JScript 支持 **if** 和 **if...else** 条件语句。**if** 语句测试一个条件。如果条件的计算结果为 true，则执行相应的 JScript 代码。**if...else** 语句测试一个条件，并根据条件语句的结果执行两个代码块之一。最简单的 **if** 语句形式可以写在一行内，但是通常使用多行 **if** 和 **if...else** 语句。

条件语句示例

以下示例阐释了 **if** 和 **if...else** 语句可以使用的语法。第一个示例显示最简单一种的布尔测试。当（且仅当）圆括号内的项计算结果为（或可以强制为）**true** 时，执行 **if** 后面的语句或语句块。

在下面的示例中，如果 `newUser` 的值可以转换为 true，则调用 `registerUser` 函数。

```
if (newUser)
   registerUser();
```

在本例中，除非两个条件均为 true，否则测试失败。

```
if (rind.color == "deep yellow " && rind.texture == "wrinkled") {
   theResponse = ("Is it a Crenshaw melon?");
}
```

在本例中，直到变量 `quit` 为 **true** 时，才停止执行 **do...while** 循环体中的代码。

```
var quit;
do {
   // ...
   quit = getResponse()
}
while (!quit)
```

请参见

参考

if...else 语句

其他资源

JScript 条件结构

JScript 参考

条件运算符

JScript 支持隐式条件格式，即条件运算符。它使用三个操作数。前两个操作数之间用问号分隔，第二个操作数和第三个操作数之间用冒号分隔。第一个操作数是条件表达式。第二个操作数是当条件表达式计算结果为 true 时执行的语句。如果条件为 false，则执行第三个操作数。有关更多信息，请参见条件（三元）运算符 (?:)。条件运算符类似于 **if...else** 语句。

使用条件运算符

在本例中，条件运算符确定 24 小时制的某个小时是在上午 ("AM") 还是在下午 ("PM")。

```
var hours : String = (the24Hour >= 12) ? " PM" : " AM";
```

一般来说，当在要执行的语句之间进行选择时，**if ... then ... else** 结构较为适用；而当在两个表达式之间进行选择时，则条件运算符 (?:) 较为适用。当在两个以上的选项之间选择或者当要执行语句块时，不要使用条件运算符。在这些情况下，请使用 **if...then...else** 结构。

请参见

其他资源

JScript 条件结构

JScript 参考

JScript 中的循环

JScript 包含几种重复执行语句或语句块的方法。一般，重复执行称为"*循环*"或"*迭代*"。迭代就是循环的单次执行。通常，它是由变量测试控制的，每次执行循环时，变量值都会发生变化。JScript 支持 4 种类型的循环：**for** 循环、**for...in** 循环、**while** 循环和 **do...while** 循环。

本节内容

for 循环

讨论 JScript 如何使用 **for** 循环，并提供一些实际的例子。

for...in 循环

描述 **for...in** 循环的概念，并解释在 JScript 中如何使用这种循环。

while 循环

讨论两种类型的 **while** 循环，并解释这两种循环与 **for** 循环之间的区别。

break 和 continue 语句

描述如何使用 **break** 和 **continue** 语句来改写循环的行为。

相关章节

JScript 条件结构

描述 JScript 通常如何处理程序流程并提供指向特定信息的链接，这些信息解释如何控制程序的执行流程。

JScript 参考

列出 JScript 语言参考中涉及的元素，并提供指向特定主题的链接，这些主题解释了有关正确使用这些语言元素的幕后详细信息。

for 循环

for 语句指定一个计数器变量、一个测试条件以及一个更新计数器的操作。在每次循环迭代之前，先测试条件。如果测试成功，则执行循环内的代码。如果测试失败，则不执行循环内的代码，程序继续执行紧靠循环后面的第一行代码。在循环执行后和下一次迭代开始之前，先更新计数器变量。

用于循环

如果循环条件始终不满足，则不执行该循环。如果始终满足测试条件，则产生无限循环。在某些情况下，可能希望出现前一种情况，但几乎从不希望出现后一种情况，因此编写循环条件时一定要谨慎。在本例中，使用 **for** 循环用前面的元素之和来初始化数组元素。

```
var sum = new Array(10); // Creates an array with 10 elements
sum[0] = 0;              // Define the first element of the array.
var iCount;

// Counts from 0 through one less than the array length.
for(iCount = 0; iCount < sum.length; iCount++) {
   // Skip the assignment if iCount is 0, which avoids
   // the error of reading the -1 element of the array.
   if(iCount!=0)
      // Add the iCount to the previous array element,
      // and assign to the current array element.
      sum[iCount] = sum[iCount-1] + iCount;
   // Print the current array element.
   print(iCount + ": " + sum[iCount]);
}
```

该程序的输出为：

```
0: 0
1: 1
2: 3
3: 6
4: 10
5: 15
6: 21
7: 28
8: 36
9: 45
```

在下一示例中有两个循环。第一个循环的代码块从不执行，而第二个循环则是无限循环。

```
var iCount;
var sum = 0;
for(iCount = 0; iCount > 10; iCount++) {
   // The code in this block is never executed, since iCount is
   // initially less than 10, but the condition checks if iCount
   // is greater than 10.
   sum += iCount;
}
// This is an infinite loop, since iCount is always greater than 0.
for(iCount = 0; iCount >= 0; iCount++) {
   sum += iCount;
}
```

请参见

参考

for 语句

其他资源

JScript 中的循环

JScript 条件结构

for...in 循环

JScript 提供一种特殊的循环，用于迭代对象的所有用户定义的属性、数组的所有元素或者集合中的所有元素。**for...in** 循环中的循环计数器是字符串或对象，而不是数字。它包含当前属性的名称、当前数组元素的索引或者集合中的当前元素。

使用 for...in 循环

下列代码阐释如何使用 **for...in** 构造。

```
// Create an object with some properties.
var prop, myObject = new Object();
myObject.name = "James";
myObject.age = 22;
myObject.phone = "555 1234";
// Loop through all the properties in the object.
for (prop in myObject){
   print("myObject." + prop + " equals " + myObject[prop]);
}
```

该程序的输出为：

```
myObject.name equals James
myObject.age equals 22
myObject.phone equals 555 1234
```

请注意，JScript 中 **for...in** 循环构造的新行为使得不再需要使用 **Enumerator** 对象来循环集合中的元素。

请参见

参考

for...in 语句

其他资源

JScript 中的循环

JScript 条件结构

JScript 参考

while 循环

while 循环允许重复执行语句块，这一点与 **for** 循环类似。但是，**while** 循环没有内置计数器变量或更新表达式。若要使用更复杂的规则来控制语句或语句块的重复执行，而不是仅“运行此代码 n 次”，则使用 **while** 循环。

使用 while 循环

下面的示例阐释了 **while** 语句的使用方法：

```
var x = 1;
while (x < 100) {
   print(x);
   x *= 2;
}
```

该程序的输出为：

```
1
2
4
8
16
32
64
```

注意

因为 **while** 循环没有显式的内置计数器变量，所以此循环比其他类型的循环更容易出现无限循环。再者，因为有时很难发现循环条件是在何时或何处更新的，所以如果使用不慎，编写的 **while** 循环可能从不更新其条件。因此，设计 **while** 循环时要十分谨慎。

正如上面提到的，JScript 中的 **do...while** 循环与 **while** 循环很相似。由于条件是在循环结束而不是在循环开始时测试的，所以可以保证至少执行一次 **do...while** 循环。例如，可以将上面的循环重写为：

```
var x = 1;
do {
   print(x);
   x *= 2;
}
while (x < 100)
```

该程序的此次输出与上面显示的输出相同。

请参见

参考

while 语句

do...while 语句

其他资源

JScript 中的循环

JScript 条件结构

JScript 参考

break 和 continue 语句

如果满足某种条件时，JScript 中的 **break** 语句就会停止执行循环。(请注意，**break** 也用于退出 **switch** 块)。如果循环是 **for** 或 **for...in** 循环，可以使用 **continue** 语句直接跳到下一个迭代，跳过代码块的其余部分，同时更新计数器变量。

使用 break 和 continue 语句

下面的示例阐释如何使用 **break** 和 **continue** 语句控制循环。

```
for(var i = 0;i <=10 ;i++) {
   if (i > 7) {
      print("i is greater than 7.");
      break; // Break out of the for loop.
   }
   else {
      print("i = " + i);
      continue; // Start the next iteration of the loop.
      print("This never gets printed.");
   }
}
```

该程序的输出为：

```
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i is greater than 7.
```

请参见

参考

break 语句

continue 语句

其他资源

JScript 中的循环

JScript 条件结构

JScript 参考

JScript 保留字 (JScript)

JScript 有许多保留字，它们在 JScript 语言语法中有特定的意义。因此，这些字不宜作为脚本中的函数、变量或常数的名称。总共有三类保留字。

受保护的保留字

受保护的保留字不能用作标识符。将受保护的保留字用作标识符会在加载脚本时引起编译错误。

break	case	catch	class	const
continue	debugger	default	delete	do
else	export	extends	false	finally
for	function	if	import	in
instanceof	new	null	protected	return
super	switch	this	throw	true
try	typeof	var	while	with

注意

尽管"export"是一个受保护的保留字，但它没有实现方法。

新保留字

JScript 还有一系列新保留字。像受保护的保留字一样，这些关键字在当前版本的 JScript 内有着特殊的意义。由于向后兼容的原因，新保留字可用作标识符。一旦将新保留字用作标识符，它就失去了作为脚本中关键字的意义。将新保留字用作标识符会引起混淆，应予以避免。

abstract	boolean	byte	char	decimal
double	enum	final	float	get
implements	int	interface	internal	long
package	private	protected	public	sbyte
set	short	static	uint	ulong
ushort	void			

未来保留字

JScript 有一系列未来保留字，这些保留字将被建议用作 JScript 的未来扩展中的关键字。像新保留字一样，这些保留字也可以在当前版本的 JScript 中用作标识符。然而，若避免使用这些字，则在更新脚本以利用未来版本的 JScript 中的功能时会更为方便。

在选择标识符时，避免选择已经是内部 JScript 对象或函数的名称(比如 **String** 或 **parseInt**)也是非常重要的。

assert	ensure	event	goto	invariant
namespace	native	require	synchronized	throws

transient	use	volatile		

请参见

其他资源

JScript 参考

JScript 语言教程

JScript 的安全注意事项

以任何语言编写安全代码都是一个挑战。JScript 的一些部分中开发人员可能在不知情的情况下以非安全方式使用语言，因为该语言不强制开发人员使用最有效的做法，虽然在设计 JScript 时将安全作为其中的一个目标，但是主要目标是促进有用应用程序的快速开发。在某些情况下，这两个目标是对立的。

如果您知道下面列出的一些主要区域中的潜在问题，则可以避免安全问题。除 **eval** 方法外，这些安全注意事项都是由于 .NET Framework 引入的新功能产生的。

eval 方法

最容易误用的 JScript 功能是 **eval** 方法，它允许动态执行 JScript 源代码。因为使用 **eval** 方法的 JScript 应用程序可以执行程序传递给它的任何代码，所以对 **eval** 方法的每个调用都会造成安全风险。除非您的应用程序需要灵活执行任何代码，否则请考虑显式编写应用程序传递至 **eval** 方法的代码。

为增加需要 **eval** 方法提供的完整灵活性的应用程序的安全，默认情况下传递给 **eval** 的代码在受限上下文中运行。受限安全上下文有助于禁止对系统资源的所有访问，如文件系统、网络或用户界面。如果代码试图访问这些资源，则会产生安全异常。但是，**eval** 方法运行的代码仍可以修改局部变量和全局变量。有关更多信息，请参见 eval 方法。

用以前版本的 JScript 编写的代码可能需要 **eval** 方法将代码与调用代码在同一安全上下文中运行。要实现这种行为，您可以将字符串"unsafe"作为第二个可选参数传递至 **eval** 方法。由于在"不安全"模式下代码字符串会与调用代码使用相同权限执行，所以您应该仅执行从已知来源获取的代码字符串。

安全属性

.NET Framework 的安全属性可用于显式重写 JScript 中的默认安全设置。但是，除非您知道修改默认安全设置的后果，否则请不要修改。特别是，由于不受信任的调用方通常无法安全地调用 JScript 代码，因此您不应应用 **AllowPartiallyTrustedCallers** 属性 (APTCA) 的自定义属性。如果您使用 APTCA 创建受信任的程序集，之后程序集被某个应用程序加载，则部分受信任的调用方可以访问该应用程序中完全受信任的程序集。有关更多信息，请参见代码安全维护指南。

部分受信任的代码和寄宿的 JScript 代码

承载 JScript 的引擎允许调用的任何代码修改引擎的组成部分，例如，全局变量、局部变量和任何对象的原型链。另外，任何函数都可以修改传递至它的任何 expando 对象的 expando 属性或方法。因此，如果 JScript 应用程序调用部分受信任的代码，或者如果它在某个应用程序内与其他代码（如 Visual Studio for Applications [VSA] 宿主中的代码）一起运行，则可以修改它的行为。

所以，应用程序（或 **AppDomain** 类的实例）中的任何 JScript 代码都应使用较低的信任级别，不得高于应用程序中其余代码使用的信任级别。否则，其他代码可能会操作 JScript 类的引擎，而引擎又会修改数据来影响应用程序中的其他代码。有关更多信息，请参见 _AppDomain。

程序集访问

JScript 可以引用同时使用强名称和简单文本名称的程序集。强名称引用包括程序集的版本信息以及确认程序集的完整性和标识的加密签名。虽然在引用程序集时使用简单名称较为容易，但是，如果您的系统上的另一个程序集具有相同的简单名称但功能不同，则强名称可以保护您的代码。有关更多信息，请参见 如何：引用具有强名称的程序集。

线程处理

JScript 运行库未被设计为线程安全的。因此，多线程 JScript 代码可能会具有不可预知的行为。如果您使用 JScript 开发程序集，请记住，它可以用于多线程上下文。您应使用 **System.Threading** 命名空间中的类（例如，**Mutex** 类），以确保程序集中的 JScript 代码使用正确的同步运行。

因为使用任何语言都很难编写正确的同步代码，因此您不应尝试使用 JScript 编写通用程序集，除非您非常熟悉如何实现必需的同步代码。有关更多信息，请参见 System.Threading。

注意
您无需为使用 JScript 编写的 ASP.NET 应用程序编写同步代码，因为 ASP.NET 管理其生成的所有线程的同步。但是，因为用 JScript 编写的 Web 控件的行为与程序集类似，所以它们必须包含同步代码。

运行时错误

因为 JScript 是一种松散类型化语言，所有它比一些其它语言更能容忍潜在的类型不匹配，例如 Visual Basic 和 Visual C#。因为

类型不匹配可以导致应用程序中发生运行时错误，所以在开发代码时查找潜在的类型不匹配十分重要。在 ASP.NET 页中，您可以将 **/warnaserror** 标志与命令行编译器或 **@ Page** 指令的 **warninglevel** 属性一起使用。有关更多信息，请参见 /warnaserror 和 @ Page。

兼容性模式

与那些在快速模式（默认模式）下编译的程序集相比，在兼容性模式下编译的程序集（使用 **/fast-** 选项）的安全性较低。**/fast-** 选项可启用默认情况下无法使用、但要与为 JScript 5.6 版及更早版本编写的脚本兼容又需要使用的语言功能。例如，在兼容性模式下，可以将 expando 属性动态地添加到内部对象，例如 **String** 对象。

提供兼容性模式的目的是帮助开发人员从旧式 JScript 代码生成独立的可执行文件。开发新的可执行文件或库时，请使用默认模式。这不仅有助于确保应用程序的安全，而且可以确保性能得到改进以及与其他程序集更好地交互。有关更多信息，请参见 /fast。

请参见

概念

升级先前的 JScript 版本中创建的应用程序

其他资源

本机代码和 .NET Framework 代码的安全性

语言参考

JScript 语言的元素组成了开发应用程序和脚本的基础。

本节内容

数据类型

指令

错误

函数

标识符 (Literal)

方法

修饰符

对象

运算符

属性

语句

相关章节

.NET Framework 参考

列出指向特定主题的链接，这些主题解释 .NET Framework 类库和其他基本元素的语法和结构。

数据类型 (JScript)

数据类型指定变量、常数或函数可接受的值的类型。变量、常数和函数的类型批注可通过限制数据的适当类型来帮助减少编程错误。而且，类型批注还可以产生更快更有效的代码。

本节内容

相关章节

boolean 数据类型 (JScript)

boolean 类型的值（真或假）取决于是否给该类型分配真或假关键字。

相应的 .NET Framework 数据类型为 **System.Boolean**。**Boolean** 数据类型等同于 **boolean** 数据类型。

备注

boolean 数据类型的属性和方法与 System.Boolean 属性和方法相同。

JScript 还定义了 **Boolean** 对象。**boolean** 数据类型与 **Boolean** 对象互用。因此，**Boolean** 对象可以调用 **boolean** 数据类型的方法和属性，而 **boolean** 数据类型可以调用 **Boolean** 对象的方法和属性。有关附加信息，请参见 Boolean 对象属性和方法。此外，Boolean 对象被采用 Boolean 数据类型的函数所接受，反之亦然。

大多数情况下，应使用 **boolean** 数据类型来代替 **Boolean** 对象。

属性和方法

AllMembers.T:System.Boolean

要求

.NET 版本

请参见

参考

Boolean 对象
Boolean Structure
true 标识符
false 标识符

概念

数据类型摘要

其他资源

数据类型 (JScript)

byte 数据类型 (JScript)

byte 类型存储为一个无符号的字节。

byte 类型可以表示 0 至 255 范围内（包括这两个数）的整数。

相应的 .NET Framework 数据类型为 Byte。**byte** 数据类型的属性和方法与 **Byte** 属性和方法相同。

属性和方法

AllMembers.T:System.Byte

要求

.NET 版本

请参见

概念

数据类型摘要

其他资源

数据类型 (JScript)

char 数据类型 (JScript)

char 类型存储为双字节的 Unicode 字符。

char 类型可以表示 65,536 个 Unicode 字符中的任意字符。

相应的 .NET Framework 数据类型为 Char。**char** 数据类型的属性和方法与 **Char** 的属性和方法相同。

属性和方法

AllMembers.T:System.Char

要求

.NET 版本

请参见

参考

String 数据类型 (JScript)

概念

数据类型摘要

其他资源

数据类型 (JScript)

decimal 数据类型 (JScript)

decimal 类型存储为一个 12 字节的整数部分、一个 1 位的符号和一个刻度因子。

decimal 类型可以精确地表示非常大或非常精确的小数。大至 1028（正或负）以及有效位数多达 28 位的数字可以作为 **decimal** 类型存储而不失其精确性。该类型对于必须避免舍入错误的应用程序（如记账）很有用。

相应的 .NET Framework 数据类型为 Decimal。**decimal** 数据类型的属性和方法与 **Decimal** 属性和方法相同。

属性和方法

AllMembers.T:System.Decimal

要求

.NET 版本

请参见

概念

数据类型摘要

其他资源

数据类型 (JScript)

double 数据类型 (JScript)

double 类型存储为八字节的双精度浮点数字。它表示双精度 64 位 IEEE 754 值。

double 类型能够以约 15 位的准确度表示大至 10308(正或负)的数字和小至 10-323 的数字。**double** 类型还可以表示 **NaN**(非数字)、正负无穷和正负零。

该类型对于需要大数但不需要很高的准确度的应用程序很有用。如果需要非常准确的数字，请考虑使用 **Decimal** 数据类型。

相应的 .NET Framework 数据类型为 Double。**double** 类型等效于 **Number** 类型。

备注

double 数据类型的属性和方法与 System.Double 属性和方法相同。

JScript 定义了 Number 对象。double 数据类型与 Number 对象互用。因此，Number 对象可以调用 double 数据类型的方法和属性，而 double 数据类型可以调用 Number 对象的方法和属性。有关其他信息，请参见 Number 对象属性和方法。此外，Number 对象被采用 double 数据类型的函数所接受，反之亦然。

大多数情况下，应使用 double 数据类型来代替 Number 对象。

属性和方法

AllMembers.T:System.Double

要求

.NET 版本

请参见

参考

Number 数据类型

decimal 数据类型 (JScript)

概念

数据类型摘要

其他资源

数据类型 (JScript)

float 数据类型

float 类型存储为四字节的单精度浮点数。它表示单精度 32 位 IEEE 754 值。

float 类型能够以约 7 位的准确度表示大至 1038（正或负）的数字和小至 10-44 的数字。**float** 类型还可以表示 **NaN**（非数字）、正负无穷和正负零。

该类型对于需要大数但不需要很高的准确度的应用程序很有用。如果需要非常准确的数字，请考虑使用 **Decimal** 数据类型。

相应的 .NET Framework 数据类型为 Single。**float** 数据类型的属性和方法与 **Single** 属性和方法相同。

属性和方法

AllMembers.T:System.Single

要求

.NET 版本

请参见

参考

decimal 数据类型 (JScript)

概念

数据类型摘要

其他资源

数据类型 (JScript)

int 数据类型

int 数据类型存储为四字节整数。

int 类型可以表示负 2,147,483,648 至正 2,147,483,647 范围内（包括这两个数）的整数。

相应的 .NET Framework 数据类型为 Int32。**int** 数据类型的属性和方法与 **Int32** 的属性和方法相同。

属性和方法

AllMembers.T:System.Int32

要求

.NET 版本

请参见

概念

数据类型摘要

其他资源

数据类型 (JScript)

long 数据类型 (JScript)

long 类型存储为八字节整数。

long 类型可以表示大约从 -1019 至 1019 的范围内的整数。

相应的 .NET Framework 数据类型为 Int64。**long** 数据类型的属性和方法与 **Int64** 属性和方法相同。

属性和方法

AllMembers.T:System.Int64

要求

.NET 版本

请参见

概念

数据类型摘要

其他资源

数据类型 (JScript)

Number 数据类型

Number 类型存储为八字节的双精度浮点数。它表示双精度 64 位 IEEE 754 值。

Number 类型能够以约 15 位的准确度来表示大至 1E+308（正或负）的数字和小至 1E-323 的数字。**Number** 类型也可以表示 **NaN**（非数字）、正负无穷和正负零。

该类型对于需要大数但不需要很高的准确度的应用程序很有用。如果需要非常准确的数字，请考虑使用 **Decimal** 数据类型。

相应的 .NET Framework 数据类型为 Double。**Number** 类型等效于 **double** 类型。

备注

Number 数据类型的属性和方法与 **Double** 属性和方法相同。

JScript 还定义了 Number 对象。Number 数据类型与 Number 对象互用。因此，Number 对象可以调用 Number 数据类型的方法和属性，而 Number 数据类型可以调用 Number 对象的方法和属性。有关其他信息，请参见 Number 对象属性和方法。此外，Number 对象被采用 Number 数据类型的函数所接受，反之亦然。

大多数情况下，应使用 Number 数据类型来代替 Number 对象。

属性和方法

AllMembers.T:System.Double

要求

.NET 版本

请参见

参考

double 数据类型 (JScript)
decimal 数据类型 (JScript)
Number 对象

概念

数据类型摘要

其他资源

数据类型 (JScript)

sbyte 数据类型 (JScript)

sbyte 类型存储为一个有符号的字节。

sbyte 类型可以表示负 128 至正 127 范围内（包括这两个数）的整数。

相应的 .NET Framework 数据类型为 SByte。**sbyte** 数据类型的属性和方法与 **SByte** 的属性和方法相同。

属性和方法

AllMembers.T:System.SByte

要求

.NET 版本

请参见

概念

数据类型摘要

其他资源

数据类型 (JScript)

short 数据类型 (JScript)

short 数据类型存储为双字节整数。

short 类型可以表示负 32,768 至正 32,767 范围内（包括这两个数）的整数。

相应的 .NET Framework 数据类型为 Int16。**short** 数据类型的属性和方法与 **Int16** 属性和方法相同。

属性和方法

AllMembers.T:System.Int16

要求

.NET 版本

请参见

概念

数据类型摘要

其他资源

数据类型 (JScript)

String 数据类型 (JScript)

String 的长度可以是零个字符至大约二十亿个字符。每个字符是一个 16 位的 Unicode 值。

相应的 .NET Framework 数据类型为 String。

备注

String 数据类型的属性和方法与 **String** 的属性和方法相同。

JScript 也定义了 **String** 对象，它提供与 **String** 数据类型不同的属性和方法。不能为 **String** 数据类型的变量创建属性或向其添加方法，但是可以为 **String** 对象的实例创建属性或向其添加方法。

String 对象与 **String** 数据互用。因此，**String** 对象可以调用 **String** 数据类型的方法和属性，而 **String** 数据类型可以调用 **String** 对象的方法和属性。有关其他信息，请参见 String 对象属性和方法。此外，**String** 对象被采用 **String** 数据类型的函数所接受，反之亦然。

在字符串中可以使用转义序列来表示不能直接在字符串中使用的特殊字符，如换行符或 Unicode 字符。当编译脚本时，字符串中的每个转义序列都会转换为它所表示的字符。有关其他信息，请参见字符串数据。

JScript 不解释特殊的 Unicode 序列（如代理项对），在比较字符串时也不将其正常化。

注意
表示单个字符并且只有组合在一起才有意义的 Unicode 字符对称作代理项对。

某些字符可以由多个 Unicode 字符序列来表示。如果单独的正常化序列表示相同的字符，则以相同的方式解释它们。

属性和方法

AllMembers.T:System.String

要求

.NET 版本

请参见

参考

String 对象

char 数据类型 (JScript)

概念

数据类型摘要

字符串数据

其他资源

数据类型 (JScript)

uint 数据类型

uint 类型存储为四字节的无符号整数。

uint 类型可以表示 0 至 4,294,967,295 范围内(包括这两个数)的整数。

相应的 .NET Framework 数据类型为 UInt32。**uint** 数据类型的属性和方法与 **UInt32** 的属性和方法相同。

属性和方法

AllMembers.T:System.UInt32

要求

.NET 版本

请参见

概念

数据类型摘要

其他资源

数据类型 (JScript)

ulong 数据类型 (JScript)

ulong 类型存储为八字节的无符号整数。

ulong 类型可以表示 0 至大约 1020 范围内的整数。

相应的 .NET Framework 数据类型为 UInt64。**ulong** 数据类型的属性和方法与 **UInt64** 的属性和方法相同。

属性和方法

AllMembers.T:System.UInt64

要求

.NET 版本

请参见

概念

数据类型摘要

其他资源

数据类型 (JScript)

ushort 数据类型 (JScript)

ushort 数据类型存储为双字节的无符号整数。

ushort 类型可以表示 0 至 65,635 范围内（包括这两个数）的整数。

相应的 .NET Framework 数据类型为 UInt16。**ushort** 数据类型的属性和方法与 **UInt16** 的属性和方法相同。

属性和方法

AllMembers.T:System.UInt16

要求

.NET 版本

请参见

概念

数据类型摘要

其他资源

数据类型 (JScript)

指令

JScript 指令控制特定的编译器、调试器和错误信息选项。

本节内容

@debug 指令

开启或关闭调试符号的显示。

@position 指令

在错误信息中给出有意义的位置信息。

相关章节

JScript 参考

列出"JScript 语言参考"所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

.NET Framework 参考

列出指向特定主题的链接，这些主题解释 .NET Framework 类库和其他基本元素的语法和结构。

@debug 指令

开启或关闭调试符号的显示。

```
@set @debug(on | off)
```

参数

on

默认值。开启调试的关键字。

off

可选项。关闭调试的关键字。

备注

JScript 作者编写的程序代码有时与正在编译和运行的实际代码不同。宿主环境（如 ASP.NET）或开发工具可能会生成其自己的代码并将其添加到程序中。此代码在调试过程中通常对于作者不具有意义。因此，当调试代码时，代码作者通常只想看到程序中他们自己编写的部分，而不想包括由开发工具生成的部分。包作者可能会出于类似原因关闭调试。

只有在使用带有 **/debug** 选项的命令行编译时，或使用 **@page** 指令中的调试标志集编译 ASP.NET 页时，编译器才发出调试符号。在这些情况下，**debug** 指令默认为打开。当 **debug** 指令出现时，它将一直有效，直至达到文件的末尾或找到下一个 **debug** 指令。

当 **debug** 指令关闭时，编译器不会为局部变量（在函数或方法中定义的变量）发出调试信息。但是，**debug** 指令不阻止为局部变量发出调试信息。

示例

从命令行使用 **/debug** 选项编译下面的代码时，这些代码会为局部变量 `debugOnVar` 但不为 `debugOffVar` 发出调试符号：

```
function debugDemo() {
   // Turn debugging information off for debugOffVar.
   @set @debug(off)
   var debugOffVar = 42;
   // Turn debugging information on.
   @set @debug(on)

   // debugOnVar has debugging information.
   var debugOnVar = 10;

   // Launch the debugger.
   debugger;
}

// Call the demo.
debugDemo();
```

要求

.NET 版本

请参见

参考

@set 语句
@position 指令
/debug
debugger 语句

其他资源

编写、编译、调试 JScript 代码

@position 指令

在错误信息中给出有意义的位置信息。

```
@set @position(end | [file = fname ;] [line = lnum ;] [column = cnum])
```

参数

fname

如果使用 **file**，则是必需的。它是表示文件名的字符串，其中可以包含也可以不包含驱动器或路径信息。

lnum

如果使用 **line**，则是必需的。它是任何表示一个创作代码行的非负整数。

cnum

如果使用 **column**，则是必需的。它是任何表示一个创作代码列的非负整数。

备注

JScript 作者编写的程序代码有时与正在编译和运行的实际代码不同。宿主环境（如 ASP.NET）或开发工具可能会生成其自己的代码并将其添加到程序中。此代码通常对于作者不具有意义，但它可能会在出错时给作者造成混淆。

编译器可能会错误地识别根本不存在于初始创作代码中的错误行，而不是在出错处正确地识别作者的代码行。这是因为所生成的其他代码已更改了作者的初始代码的相对位置。

示例

在下面的示例中，文件中的行号被更改，以提供由 JScript 宿主插入作者代码中的代码。左列中的行号表示用户所见的初始源。

```
01  ..  // 10 lines of host-inserted code.
..  ..  //...
10  ..  // End of host-inserted code.
11  ..  @set @position(line = 1)
12  01  var i : int = 42;
13  02  var x = ; // Error reported as being on line 2.
14  03  //Remainder of file.
```

要求

.NET 版本

请参见

参考

@set 语句

@debug 指令

错误

错误信息有助于对脚本中的意外结果或行为进行疑难解答。以下各节解释如何解决在运行时或语法不一致时发生的错误。

本节内容

JScript 运行时错误

JScript 语法错误

相关章节

JScript 参考

列出“JScript 语言参考”所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

错误信息

提供一个链接列表，这些链接指向有关 Visual Studio 集成开发环境 (IDE) 和其他开发语言的错误信息。

JScript 运行时错误

当 JScript 脚本试图执行系统不能执行的操作时，将导致 JScript 运行时错误。JScript 运行时错误在脚本执行过程中计算变量表达式并且动态地分配内存时出现。

捕获错误

运行时错误可以由 JScript 程序来捕获和检查。通过在 **try** 块中包含产生错误的代码，可以用 **catch** 块捕获所引发的任何错误。JScript 所引发的错误为 **Error** 对象。JScript 代码可以使用 **throw** 语句来生成任意数据类型的自定义错误（包括 **Error** 对象）。程序可以显示所捕获的 **Error** 对象的错误号和消息，以帮助您识别错误。如果未捕获错误，脚本则将终止。

有几种获取特定错误信息的帮助的方法：

- 在 JScript 运行时错误节点所包含的目录中查找错误号和消息。
- 在"索引"中的"查找"框中键入错误号。错误号的格式为 JSxxxx，其中 xxxx 是四位数的错误代码。
- 在"搜索"中的"查找"框中键入错误信息。记住，某些错误信息包括用单引号引起来的词。引起来的词引用代码中的标识符，它们不是错误信息的一部分。请不要将引起来的词用作搜索的一部分。

请参见

参考

错误对象

try...catch...finally 语句

throw 语句

概念

JScript 语法错误

JS5000: 无法给“this”赋值

已经给 **this** 赋值，此 JScript 关键字引用：

- 当前正在执行方法的对象，或
- 全局对象（如果没有当前方法或方法不属于其他任何对象）。

方法是通过对象调用的 JScript 函数。在方法中，**this** 关键字是对通过其调用方法的对象（正好是通过调用带有 **new** 运算符的类构造函数所创建的对象）的引用。

在方法中，您可以使用 **this** 来引用当前对象，但不能将新的值赋给 **this**。

更正此错误

- 不要给 **this** 赋值。若要访问实例化对象的属性或方法，请使用点运算符（例如 circle.radius）。

注意
不能将用户创建的变量命名为 **this**；它是 JScript 的保留字。

请参见

参考

this 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS5001: 应为数字

代码对类型不是 **Number** 的对象调用了 **Number.prototype.toString** 或 **Number.prototype.valueOf** 方法。这种调用的对象必须属于 **Number** 类型。

更正此错误

- 仅对类型为 **Number** 的对象调用 **Number.prototype.toString** 或 **Number.prototype.valueOf** 方法。

请参见

参考

Number 对象
toString 方法
valueOf 方法
prototype 属性

其他资源

对象 (JScript)

JS5002: 应为函数

代码对不是 **Function** 对象的对象调用了 **Function prototype** 方法之一，或者它使用了函数调用上下文中的对象。例如，由于 **mysample** 不是函数，以下代码会产生错误。

```
var mysample = new Object();  // Create a new object called "mysample".
var x = mysample();           // Try and call mysample as if it were a function.
```

更正此错误

1. 仅对 **Function** 对象调用 **Function prototype** 方法。
2. 请确保函数调用运算符 **()** 仅调用函数。

请参见

参考

Function 对象
prototype 属性

其他资源

对象 (JScript)

JS5003：无法给函数结果赋值

代码试图给函数结果赋值。可以函数的结果赋给变量，但不能将它用作变量。如果要将新值赋给函数本身，请省略小括号（函数调用运算符）。

更正此错误

1. 不要将函数调用结果的值用作可以“赋值”的对象。不过，可以将函数调用的结果赋给*变量*。

```
myVar = myFunction(42);
```

2. 或者，可以将函数本身（而不是其返回值）赋给变量。

```
myFunction = new Function("return 42;");
```

请参见

参考

Function 对象

其他资源

JScript 语言教程

JScript 函数

Visual Basic 和 Visual C# 项目扩展性方法

JS5005: 应为字符串

代码对类型不是 **String** 的对象调用了 **String.prototype.toString** 或 **String.prototype.valueOf** 方法。这种调用的对象必须属于 **String** 类型。

更正此错误

- 仅对类型为 **String** 的对象调用 **String.prototype.toString** 或 **String.prototype.valueOf** 方法。

请参见

参考

String 对象
toString 方法
valueOf 方法
prototype 属性

其他资源

对象 (JScript)

JS5006: 应为 Date 对象

代码对类型不是 **Date** 的对象调用了 **Date.prototype.toString** 或 **Date.prototype.valueOf** 方法。这种调用的对象必须属于 **Date** 类型。

更正此错误

- 仅对类型为 **Date** 的对象调用 **Date.prototype.toString** 或 **Date.prototype.valueOf** 方法。

请参见

参考

Date 对象
toString 方法
valueOf 方法
prototype 属性

其他资源

对象 (JScript)

JS5007: 应为对象

代码对类型不是 **Object** 的对象调用了 **Object.prototype.toString** 或 **Object.prototype.valueOf** 方法。这种调用的对象必须属于 **Object** 类型。

更正此错误

- 仅对类型为 **Object** 的对象调用 **Object.prototype.toString** 或 **Object.prototype.valueOf** 方法。

请参见

参考

Object 对象
toString 方法
valueOf 方法
prototype 属性

其他资源

对象 (JScript)

JS5008: 非法赋值

代码试图给只读标识符赋值。只读标识符不能赋值。例如，由宿主定义的对象和外部 COM 对象是只读标识符。

更正此错误

- 不要给只读标识符赋值。

请参见

参考

赋值运算符 (=)

JS5009: 未定义的标识符

JScript 编译器不识别某个标识符。当被引用的变量不存在或 **with** 块用于访问不存在的对象属性时，就可能出现这种错误。

更正此错误

1. 用 **var** 语句（像在 **var** x; 中一样）声明变量。
2. 确保仅在 **with** 块中引用了有效的对象成员。
3. 显式地引用对象成员，而不是使用 **with** 语句。

请参见

参考

var 语句

with 语句

概念

变量和常数的范围

其他资源

JScript 变量和常数

JScript 参考

JS5010: 应为 Boolean

代码对类型不是 **Boolean** 的对象调用了 **Boolean.prototype.toString** 或 **Boolean.prototype.valueOf** 方法。这种调用的对象必须属于 **Boolean** 类型。

更正此错误

- 仅对类型是 **Boolean** 的对象调用 **Boolean.prototype.toString** 或 **Boolean.prototype.valueOf** 方法。

请参见

参考

Boolean 对象
toString 方法
valueOf 方法
prototype 属性

其他资源

对象 (JScript)

JS5013: 应为 VBArray

向 **VBArray** 提供的对象不是 Visual Basic safeArray，但应为 Visual Basic safeArray。Visual Basic safeArrays 无法直接在 JScript 中创建，而必须通过从现有 ActiveX 或其他对象（或从同一网页上的 Visual Basic 脚本）检索值来导入。

更正此错误

1. 确保只向 **VBArray** 构造函数传递 Visual Basic safeArray 对象。
2. 使用 **System.Array** 对象。它允许任何 .NET 语言（包括 JScript 和 Visual Basic）访问和修改数组。

请参见

参考

VBArray 对象

概念

数组使用

JS5015: 应为 Enumerator 对象

代码对类型不是 **Enumerator** 的对象调用了 **Enumerator.prototype.atEnd**、**Enumerator.prototype.item**、**Enumerator.prototype.moveFirst** 或 **Enumerator.prototype.moveNext** 方法。这种调用的对象必须属于 **Enumerator** 类型。

更正此错误

- 仅对类型是 **Enumerator** 的对象调用 **Enumerator.prototype.atEnd**、**Enumerator.prototype.item**、**Enumerator.prototype.moveFirst** 或 **Enumerator.prototype.moveNext** 方法。若要确定对象是否为 **Enumerator** 对象，请使用：

```
if(x instanceof Enumerator)
```

请参见

参考

Enumerator 对象
atEnd 方法
item 方法 (JScript)
moveFirst 方法
moveNext 方法
prototype 属性

其他资源

对象 (JScript)

JS5016: 应为正则表达式对象

代码对类型不是 **RegExp** 的对象调用了 **RegExp.prototype.toString** 或 **RegExp.prototype.valueOf** 方法。这种调用的对象必须属于 **RegExp** 类型。

更正此错误

- 仅对类型是 **RegExp** 的对象调用 **RegExp.prototype.toString** 或 **RegExp.prototype.valueOf** 方法。

请参见

参考

正则表达式对象
toString 方法
valueOf 方法
prototype 属性

概念

正则表达式语法

其他资源

对象 (JScript)

JS5017: 正则表达式中有语法错误

正则表达式搜索字符串的语法违反一项或多项 JScript 正则表达式的语法规则。

更正此错误

- 确保正则表达式搜索字符串遵循 JScript 正则表达式语法。

请参见

参考

正则表达式对象

compile 方法 (JScript)

概念

正则表达式语法

JS5022: 异常被引发且未被捕获

代码包含一个未包含在 **try** 块内的 **throw** 语句，或者代码不包含用于捕获错误的关联 **catch** 块。使用 **throw** 语句从 **try** 块中引发的异常将在 **try** 块外由 **catch** 语句捕获。

更正此错误

1. 将可以引发异常的代码放在 **try** 块中，并确保存在相应的 **catch** 块。
2. 确保 catch 语句接受格式正确的异常。
3. 如果重新引发异常，则确保存在另一个相应的 catch 语句。

请参见

参考

错误对象
throw 语句
try...catch...finally 语句

JS5023：函数没有有效的原型对象

代码使用 **instanceof** 来确定对象是否是从特定函数类派生的，但是代码将对象的 **prototype** 属性重新定义为 **null** 或外部对象类型（都不是有效的 JScript 对象）。外部对象可以是宿主对象模型中的对象（例如，Internet Explorer 的文档或窗口对象）或外部 COM 对象。

更正此错误

- 确保函数的 **prototype** 属性引用有效的 JScript 对象。

请参见

参考

Function 对象

prototype 属性

其他资源

对象 (JScript)

JS5024: 要编码的 URI 包含无效字符

编码为统一资源标识符 (URI) 的字符串包含无效字符。虽然大多数字符在要转换为 URI 的字符串中都有效，但有些 Unicode 字符序列在此上下文中无效。

更正此错误

- 确保要编码的字符串只包含有效的 Unicode 序列。

  完整的 URI 由一系列组件和分隔符组成。常规格式为：

  ```
  <Scheme>:<first>/<second>;<third>?<fourth>
  ```

  尖括号中的名称表示组件，而":"、"/"、";"和"?"是用作分隔符的保留符号。

请参见

参考

encodeURI 方法
encodeURIComponent 方法

JS5025: 要解码的 URI 不是有效编码

代码试图解码格式不正确的统一资源标识符 (URI)。URI 具有特殊的语法；大多数非字母字符必须在 URI 中使用之前进行解码。**encodeURI** 和 **encodeURIComponent** 方法不能从常规 JScript 字符串中创建 URI。

完整的 URI 由一系列组件和分隔符组成。常规格式为：

```
<Scheme>:<first>/<second>;<third>?<fourth>
```

尖括号中的名称表示组件，而":"、"/"、";"和"?"是用作分隔符的保留符号。

更正此错误

- 确保代码只尝试对有效的 URI 进行解码。例如，常规的 JScript 字符串可能包含无效字符，因此可能不是有效的 URI。

请参见

参考

decodeURI 方法
decodeURIComponent 方法

JS5026: 小数位数超出范围

Number.prototype.toExponential 函数不能接受无效参数。**toExponential()** 函数的参数必须介于 0 和 20（包括这两个数）之间。

更正此错误

- 确保 **toExponential()** 的参数既不太大也不太小。

请参见

参考

Number 对象
toExponential 方法
prototype 属性

其他资源

对象 (JScript)

JS5027: 精度超出范围

Number.prototype.toPrecision 函数不能接受无效参数。**toPrecision** 函数的参数必须介于 1 和 21（包括这两个数）之间。

更正此错误

- 确保 **toPrecision** 的参数既不太大也不太小。

请参见

参考

Number 对象
toPrecision 方法
prototype 属性

其他资源

对象 (JScript)

JS5029: 数组长度必须为零或正整数

调用 **Array** 构造函数的参数为负或不是数字 (**NaN**)。注意，JScript 自动将小数转换为整数。

更正此错误

- 在创建新的 **Array** 对象时只应使用正整数或数字零。若要创建包含单个元素的数组，请采用一个两步过程。首先，创建包含一个元素的数组。然后，将值放入第一个元素 (array[0])。

  下面的示例将展示正确指定包含单个数值元素的数组的方式。

  ```
  var piArray = new Array(1);
  piArray [0] = 3.14159;
  ```

  除最大整数值（约为 40 亿）外，数组大小没有上限。

请参见

概念

数组使用

其他资源

JScript 参考

JS5030: 必须为数组长度分配正整数或者零

赋给 **Array** 对象 **length** 属性的值为负或不是数字 (**NaN**)。注意，JScript 自动将小数转换为整数。

更正此错误

- 将正整数或零赋给长度属性。下面的示例展示正确设置 **Array** 对象 **length** 属性的方式。

```
var my_array = new Array();
my_array.length = 99;
```

除最大整数值（约为 40 亿）外，数组大小没有上限。

请参见

概念

数组使用

其他资源

JScript 参考

JS5031: 应为 Array 对象

该程序试图在需要 Array 对象的上下文中使用非 **Array** 对象的内容。

更正此错误

- 确保在这种上下文中使用 **Array** 对象。

请参见

参考

Array 对象

概念

脚本疑难解答

其他资源

JScript 参考

JS5032: 没有这种构造函数

new 运算符应用于某个标识符，但是该标识符既不与构造函数对应也不与类构造函数对应。

更正此错误

- 确保 new 运算符应用于构造函数或类构造函数。

请参见

概念

脚本疑难解答

用构造函数创建自己的对象

创建自己的类

其他资源

JScript 参考

JS5033：不能通过别名调用 Eval

该程序(它的某个变量被设置为与 **eval** 方法相同)定义别名，然后将该别名用作函数。别名不能用于 **eval** 方法。

更正此错误

- 直接调用 **eval** 方法。

请参见

参考

eval 方法 (JScript)

概念

脚本疑难解答

其他资源

JScript 参考

JS5034: 尚未实现

该程序试图使用尚未实现的功能。

更正此错误

- 移除对未实现的功能的引用。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS5035: 无法提供空值或空的命名参数名

命名参数调用 JScript 函数或方法，而某个命名参数名称为空或 null。这是不允许的，因为所有参数都有非空的名称。

注意
当使用 JScript 调用函数和方法时，将不能使用命名参数。不过，可以从支持命名参数的其他语言（例如 Visual Basic）来调用 JScript 函数和方法。有关更多信息，请参见通过地址和名称传递参数。

更正此错误

- 为每个命名参数名称提供一个参数名称。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS5036: 重复的命名参数名

命名参数调用 JScript 函数或方法，而某个命名参数名称被使用了两次。这是不允许的，因为每个参数的名称必须唯一。

注意
当使用 JScript 调用函数和方法时，将不能使用命名参数。不过，可以从支持命名参数的其他语言（例如 Visual Basic）来调用 JScript 函数和方法。有关更多信息，请参见通过地址和名称传递参数。

更正此错误

- 为每个命名参数名称提供唯一的名称。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS5037: 指定的名称不是参数的名称

命名参数调用 JScript 函数或方法，而某个命名参数名称与参数名称不对应。这是不允许的，因为每个命名参数名称必须引用参数名称。

注意
当使用 JScript 调用函数和方法时，将不能使用命名参数。不过，可以从支持命名参数的其他语言（例如 Visual Basic）来调用 JScript 函数和方法。有关更多信息，请参见通过地址和名称传递参数。

更正此错误

- 为每个命名参数名称提供一个参数名称。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS5038: 指定的参数太少

指定的参数太少。参数名的个数不能超过传入的参数的个数。

命名参数调用 JScript 函数或方法，但是传入的参数数量超过了函数或方法指定的参数数量。这是不允许的，因为必须至少丢弃一个传入的参数。

注意
当使用 JScript 调用函数和方法时，不能使用命名参数。不过，可以从支持命名参数的其他语言（例如 Visual Basic）来调用 JScript 函数和方法。有关更多信息，请参见通过地址和名称传递参数。

更正此错误

- 确保传入的参数数量不超过参数名称的数量。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS5039: 无法在调试程序中计算表达式的值

在调试 JScript 程序时，向命令窗口中输入了无法计算的表达式。

更正此错误

- 确保只向命令窗口中输入有效的 JScript 表达式。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

编写、编译、调试 JScript 代码

JS5040: 给只读字段或属性赋值

该代码向只读标识符赋值。这是不允许的，因为代码不能向只读标识符写入。

const 语句定义只读字段或常数。不带匹配 **function set** 语句的 **function get** 语句定义只读属性。

更正此错误

1. 确保代码不向只读标识符赋值。
2. 用 **var** 语句定义字段或变量以便可向其赋值。
3. 在属性中添加匹配的 **function set** 语句以便可向其赋值。

请参见

参考

const 语句
function get 语句
function set 语句

概念

脚本疑难解答

其他资源

JScript 参考
基于类的对象

JS5041: 此属性只能分配给

该代码读取只写属性的值。这是不允许的，因为代码不能从只写标识符中读取值。

不带匹配 **function get** 语句的 **function set** 语句定义只写属性。

更正此错误

1. 确保该代码不从只写属性读取。
2. 向属性中添加匹配的 **function get** 语句以便使其可读。

请参见

参考

function get 语句

function set 语句

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS5042: 索引数与数组的维数不匹配

该代码访问数组的元素，但是索引的数量与数组的维数不匹配。

更正此错误

- 确保用于访问数组元素的索引数与在定义该数组时指定的维数相匹配。

请参见

概念

类型化数组

脚本疑难解答

其他资源

JScript 参考

JS5043: 不能在调试器中调用带 ref 参数的方法

在调试 JScript 程序时，向命令窗口中输入了对一个方法的调用，此方法通过引用接受参数。这是不允许的，因为在引用参数的过程中可能会更改变量的值。

更正此错误

- 确保只从命令窗口调用不通过引用接受参数的方法。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

编写、编译、调试 JScript 代码

JS5044: 不能使用后期绑定调用 Deny、PermitOnly 和 Assert 安全方法

该代码调用后期绑定的 **CodeAccessPermission** 对象的 **PermitOnly**、**Assert** 或 **Deny** 方法。出于安全原因，这是不允许的。若要使用 **PermitOnly**、**Assert** 和 **Deny** 方法，存储 **CodeAccessPermission** 对象的变量必须被显式类型化（早期绑定）以便只存储 **CodeAccessPermission** 对象。

更正此错误

- 在定义存储了 **CodeAccessPermission** 对象的变量时，使用类型批注。

请参见

参考

CodeAccessPermission Class

概念

脚本疑难解答

其他资源

JScript 参考

JS5045: JScript 不支持声明性安全属性

从 **CodeAccessSecurityAttribute** 继承的自定义属性应用于方法、类或程序集的定义。这是不允许的。必须使用动态安全（而非声明性安全属性）控制对部分代码的访问。

注意
只有早期绑定代码可在 .NET Framework 内部调用 **Assert**、**Deny** 或 **PermitOnly** 安全性方法。这意味着必须使用类型批注变量来存储权限对象，这是由于类型批注允许编译器生成早期绑定代码。而且，在运行时（使用 **eval** 方法或通过 **new** 运算符创建的 **Function** 对象）生成的代码是后期绑定代码，这将阻止它调用 **Assert**、**Deny** 或 **PermitOnly** 方法。

在下面的示例中，动态安全用于拒绝通过方法访问特定文件。

```
import System;
import System.IO;
import System.Security;
import System.Security.Permissions;
class Alpha{
   function Bravo() {
      var fileioperm : FileIOPermission;
      fileioperm = new FileIOPermission(FileIOPermissionAccess.AllAccess, 'd:\\temp\\myfile
.txt');
      fileioperm.Deny();
      // Any additional code in this method will be
      // denied access to d:\temp\myfile.txt.
   }
}
```

更正此错误

- 使用动态安全（而非声明性安全）声明安全方法或程序集。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JScript 语法错误

当 JScript 语句的结构违反一项或多项 JScript 脚本语言的语法规则时，将导致 JScript 语法错误。JScript 语法错误在语法编译阶段中、程序开始执行之前发生。

获取有关错误信息的帮助

有几种获取特定错误信息的帮助的方法：

- 在 JScript 语法错误节点所包含的目录中查找错误号和消息。
- 在“索引”中的“查找”框中键入错误号。错误号的格式为 JSxxxx，其中 xxxx 是四位数的错误代码。
- 在“搜索”中的“查找”框中键入错误信息。记住，某些错误信息包括用单引号引起来的词。引起来的词引用代码中的标识符，它们不是错误信息的一部分。引起来的词不应用作搜索的一部分。

请参见

概念

JScript 运行时错误

JS0005: 无效的过程调用或参数

该代码调用一个过程，但是未定义与此调用相匹配的过程。

更正此错误

1. 确保传递给该过程的数据类型与定义该过程所要采用的数据类型相匹配。
2. 确保传递给该过程的参数数量与该过程所期望的数量相匹配。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS0007: 内存不足

该程序已占用了所有可用内存，这可能会在处理大量数据时发生。避免出现此错误的一种方法是有效地利用可用内存。这可通过确保该程序不保留不再需要的内存（数组、对象等形式）来实现。

另一种减少程序使用的内存的方法是帮助垃圾回收例程动态释放内存。JScript 使用的垃圾回收例程负责释放未使用的内存。该例程释放程序不再使用的数据。当变量中的数据被新数据替换，或者当范围发生变化并且有一个变量不再可访问时，数据可能变得不可用。

若要释放程序中的内存，请在不再需要占用大量内存的变量（如大数组或其他对象）时将其设置为 **null**。这使得垃圾回收器能够释放内存。

更正此错误

1. 确保代码有效地使用内存。
2. 在需要占用大量内存的对象之前立即声明。
3. 当变量不再需要时将其设置为 **null**。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS0013: 类型不匹配

该代码尝试在上下文中使用某种与期望的数据类型不兼容的数据。这可能会在向变量赋值或者将参数传递给具有类型批注参数的函数时发生。

更正此错误

- 确保代码传递的数据的类型可被强制为期望的数据类型。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS0028: 堆栈空间不足

该程序已占用所有可用的堆栈空间。这可在递归函数永不显式终止时发生。

更正此错误

- 确保递归函数显式终止。

请参见

概念

脚本疑难解答

递归

其他资源

JScript 参考

JS0051: 内部错误

该代码已在脚本引擎中显示了一个内部错误。

更正此错误

1. 重写您的代码以便按照不同但等效的方式生成所需的结果。
2. 单击本页底部的“Microsoft 产品支持知识库链接”寻找解决方案的建议。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS0053: 未找到文件

该程序无法找到它尝试访问的文件。

更正此错误

1. 确保路径和文件名正确无误。
2. 确保该文件存在。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS0424: 需要对象

该程序尝试在需要使用对象的上下文中使用不是对象的内容。

更正此错误

- 确保对象存在于此上下文中。

请参见

参考

Object 对象

ActiveXObject 对象

概念

脚本疑难解答

其他资源

JScript 参考

JS0429: 无法创建对象

尽管该程序尝试创建新对象，但是它无法创建该对象。这可能会在提供该对象的应用程序不可用或者应用程序不提供该特定对象时发生。

更正此错误

1. 确保应用程序可用。

2. 确保使用适当的应用程序提供该对象。

请参见

参考

ActiveXObject 对象

概念

脚本疑难解答

其他资源

JScript 参考

JS0438: 对象不支持此属性或方法

该程序尝试访问的属性或方法不是对象的成员。

更正此错误

- 确保只访问对象的有效属性和方法。

请参见

参考

ActiveXObject 对象

概念

脚本疑难解答

其他资源

JScript 参考

JS0445: 对象不支持此操作

对象用于它不支持的操作。

更正此错误

- 确保只使用该对象的有效操作。

请参见

参考

ActiveXObject 对象

概念

脚本疑难解答

其他资源

JScript 参考

JS0451: 对象不是一个集合

该程序正尝试创建新的 **Enumerator** 对象，但是传递给构造函数的参数不是一个集合。

注意
可直接在 JScript 中访问集合的元素。有关更多信息，请参见 Enumerator 对象。

更正此错误

- 确保只使用集合构造 **Enumerator** 对象。

请参见

参考

Enumerator 对象

概念

脚本疑难解答

其他资源

JScript 参考

JS1002: 语法错误

一个 JScript 语句违反一项或多项 JScript 语法规则。

更正此错误

1. 复查指定行号上程序的语法。
2. 查找方向错误的小括号或大括号。

请参见

参考

错误对象

其他资源

JScript 参考

JS1003: 应输入“:”

一个表达式使用三元条件运算符，但在第二个和第三个操作数之间没有冒号。三元（三个操作数）条件运算符在第二个（真）和第三个（假）操作数之间需要一个冒号。

更正此错误

- 在第二个和第三个操作数之间插入一个冒号。

请参见

参考

条件（三元）运算符 (?:)

其他资源

运算符 (JScript)

JScript 参考

JS1004: 应为“;i±

位于一行的多个语句没有用分号分隔，或者位于一行的 **for** 语句没有用分号分隔其标头中的初始化、测试和递增表达式。

分号用于终止语句。虽然几个语句可以位于一行，但必须用分号将每个语句与下一个语句分隔开。

分号也用于分隔 **for** 循环的标头中的初始化、测试和递增表达式。

更正此错误

1. 用分号标记每个语句的末尾。
2. 确保函数调用正确地使用小括号。
3. 确保 **for** 循环的标头中存在分号。
4. 确保赋值表达式中有一个 =。

请参见

参考

for 语句

其他资源

JScript 语言教程

JScript 参考

JS1005: 应为“(”

一个应该用一组括号括起的表达式缺少左括号。有些表达式必须用一组左括号和右括号括起。例如，下面的 **for** 语句具有正确的括号位置：

```
for (initialize; test; increment) {
   statement;
}
```

更正此错误

- 为该计算表达式添加左括号。

请参见

其他资源

JScript 参考

JS1006: 应为“)”

一个应该用一组括号括起的表达式缺少右括号。有些表达式必须用一组左括号和右括号括起。例如，下面的 **for** 语句具有正确的括号位置：

```
for (initialize; test; increment) {
   statement;
}
```

更正此错误

- 为该计算表达式添加右括号。

请参见

其他资源

JScript 参考

JS1007: 应为“]”

一个对数组元素的引用未包含右中括号。任何引用数组元素的表达式都必须包含左中括号和右中括号。

更正此错误

- 为引用数组元素的表达式添加右中括号。

请参见

参考

Array 对象

概念

数组使用

JS1008: 应为“{”

缺少标记函数体、类成员块、接口成员块或枚举块的开头的左大括号。由函数体组成的代码即使是在一行，仍必须在两端加上左右大括号。

请注意，大括号在循环中的使用不如在函数体和成员块中严格。

更正此错误

- 添加标记函数体开头的左大括号。

请参见

参考

Function 对象
function 语句
class 语句
interface 语句
enum 语句

其他资源

JScript 参考

JS1009: 应为“}”

缺少标记函数体、类成员块、接口成员块、枚举块、循环、代码块或对象初始值设定项的末尾的右大括号。此错误的示例之一就是一个 **for** 循环只有标记循环体的左大括号。

更正此错误

- 添加标记函数、类、接口、枚举、循环、块或对象初始值设定项的末尾的右大括号。

请参见

参考

Function 对象
function 语句
class 语句
interface 语句
enum 语句

其他资源

JScript 条件结构
JScript 参考

JS1010: 应为标识符

在需要标识符的上下文中缺少标识符。标识符可能是：

- 变量，
- 属性，
- 数组，
- 函数名称。

更正此错误

- 更改表达式，使一个标识符出现在等号的左侧。

请参见

概念

数组使用

其他资源

JScript 参考

JS1011: 应为“=”

一个用于条件编译语句的变量在变量和所赋的值之间没有等号。

更正此错误

- 添加一个等号。例如：

```
@set @myvar1 = 1
```

请参见

概念

条件编译变量

其他资源

条件编译

JS1014: 无效字符

一个标识符包含被 JScript 编译器识别为无效的字符。有效字符符合下列规则：

- 第一个字符必须是 ASCII 字母（大写或小写）、下划线 (_) 或美元符号 ($)。
- 后继字符可以是 ASCII 字母、数字、下划线或美元符号。
- 标识符名称不能是保留字。

更正此错误

- 避免使用 JScript 语言定义之外的字符。

请参见

概念

字符串数据

其他资源

JScript 变量和常数

数据类型 (JScript)

JS1015: 未终止的字符串常数

一个字符串常数缺少右引号。字符串常数必须用一对引号引起。

注意
您可以使用匹配的单引号或双引号对。被单引号引起的字符串内可以包含双引号，而被双引号引起的字符串内也可以包含单引号。

更正此错误

- 在字符串末尾添加右引号。

请参见

参考

String 对象
toString 方法

JS1016: 未终止的注释

一个多行注释块未正确终止。多行注释必须以"/*"组合开头，以反序的"*/"组合结尾。

下面是多行注释正确用法的示例：

```
/* This is a comment
This is another part of the same comment.*/
```

更正此错误

- 确保多行注释以"*/"终止。

请参见

参考

Comment 语句

JS1018: “return”语句在函数范围外

return 语句位于代码的全局范围内或类体或包体之外。**return** 语句只应出现在函数体中。

更正此错误

- 移除 **return** 语句。

请参见

参考

return 语句
Function 对象
caller 属性

JS1019: “break”不能位于循环之外

break 关键字出现在循环之外。**break** 关键字用于终止循环或 **switch** 语句。它必须嵌入循环体或 **switch** 语句体内。

更正此错误

- 确保 **break** 关键字出现在封闭循环或 switch 语句内。

请参见

参考

break 语句

概念

脚本疑难解答

其他资源

JScript 条件结构

JScript 参考

JS1020: “continue”不能位于循环之外

continue 语句位于循环之外。**continue** 语句只能在下列对象体中使用：

- **do-while** 循环，
- **while** 循环，
- **for** 循环，
- **for/in** 循环。

更正此错误

- 确保 **continue** 语句出现在下列对象体中：
    - **do-while** 循环，
    - **while** 循环，
    - **for** 循环，
    - **for/in** 循环。

请参见

参考

continue 语句

概念

脚本疑难解答

其他资源

JScript 条件结构

JScript 参考

JS1023: 应为十六进制数字

代码包含不正确的 Unicode 转义序列，或者非十六进制字符是十六进制字符串中的第一个字符。

Unicode 转义序列以 \u 开头，后接恰好四个十六进制位。十六进制文字值以 0x 开头，后接任意数目的十六进制位。十六进制位只能包含数字 0-9、大写字母 A-F 和小写字母 a-f。下面的示例将展示一个格式正确的 Unicode 转义序列。

```
z = "\u1A5F";
```

下面的示例将展示一个格式正确的十六进制文字值。

```
k = 0x3E8;
```

更正此错误

1. 确保十六进制数字只包含数字 0-9、大写字母 A-F 和小写字母 a-f。
2. 确保 Unicode 转义序列包含四位。

注意
如果要在字符串中使用 \u 文本，则请使用两个反斜杠 (\\u)，其中一个反斜杠用于将第一个反斜杠转义。

请参见

其他资源

JScript 参考

数据类型 (JScript)

JS1024: 应为“while”

do while 循环不包含 **while** 条件。**do** 语句必须在代码块的末尾具有相应的 **while** 测试语句。

更正此错误

- 在右大括号后添加 **while** 测试语句。

请参见

参考

while 语句

其他资源

JScript 条件结构

JScript 参考

JS1025: 标签已重定义

新标签使用现有标签的名称。在指定范围内，标签必须唯一。

更正此错误

- 确保所有标签名称在其各自的范围内是唯一的。

请参见

参考

Labeled 语句
switch 语句
break 语句
continue 语句

JS1026: 未找到标签

代码引用不存在的标签。在指定范围内，标签必须唯一。

更正此错误

1. 检查标签名称的拼写。
2. 确保所有标签引用所引用的标签都已经在当前范围内定义（包括向前的定义）。

请参见

参考

Labeled 语句
switch 语句
break 语句
continue 语句

JS1027: “default”在“switch”语句中只能出现一次

switch 语句多次使用了 **default** case 语句。default case 语句必须始终是 switch 语句中的最后一个 case 语句（如果是贯穿的 case 语句）。

更正此错误

- 在 switch 语句中只使用一个 default case 语句。

请参见

参考

switch 语句

概念

JScript 保留字 (JScript)

其他资源

JScript 条件结构

JS1028: 应为标识符或字符串

声明对象文本时使用了不正确的文本语法。对象文本的属性必须是标识符或字符串。对象文本(也称作"对象初始值设定项")由逗点分隔的"属性:值"对列表构成，它们都包含在括号内。例如:

```
var point = {x:1.2, y:-3.4};
```

更正此错误

- 确保文本语法正确。

请参见

参考

逗号运算符 (,)

JS1029: 应为“@end”

条件编译代码块未以 **@end** 语句结尾。JScript 语句可以通过在两端添加 **@if/@end** 块来进行条件编译。

更正此错误

- 添加相应的 **@end** 语句。

请参见

参考

@if...@elif...@else...@end 语句

概念

条件编译变量

其他资源

条件编译

JS1030: 条件编译已关闭

代码使用条件编译变量，但条件编译未开启。开启条件编译会指示 JScript 编译器将以 @ 开头的标识符解释为条件编译变量。要实现此目的，请让条件代码以此语句开头：

```
/*@cc_on @*/
```

更正此错误

- 将以下语句添加到条件代码的开头：

  ```
  /*@cc_on @*/
  ```

请参见

参考

@cc_on 语句
@if...@elif...@else...@end 语句
@set 语句

概念

条件编译变量

其他资源

条件编译

JS1031: 应为常数

变量位于 @if(条件编译)测试语句中。只有文本和条件编译变量(它们在编译时都是常数)才能出现在条件编译测试语句中。

更正此错误

1. 用文本替换该变量。
2. 用条件编译变量替换该变量。

请参见

参考

@cc_on 语句
@if...@elif...@else...@end 语句
@set 语句

概念

条件编译变量

其他资源

条件编译

JS1032: 应为“@”

要用于条件编译语句的变量使用了 **@set** 语句，但在变量名称前却没有“@”符号。

更正此错误

- 紧接在变量名称之前添加一个“@”符号。例如：

```
@set @myvar = 1
```

请参见

参考

@set 语句

概念

条件编译变量

其他资源

条件编译

JS1033: 应为“catch”

异常处理 **try** 块不包含关联的 **catch** 语句。为了正确执行其功能，异常处理机制要求将可能失败的代码以及在发生异常时将不执行的代码包在 **try** 块内。异常将从 **try** 块中使用 **throw** 语句来引发，然后在 **try** 块外用一个或多个 **catch** 语句来捕获。

更正此错误

1. 添加关联的 **catch** 块。
2. 尝试使用 **finally** 块来替代 **catch** 块。

请参见

参考

try...catch...finally 语句
错误对象

JS1034: 不匹配的“else”; 未定义“if”

else 语句缺少一个与之匹配的 **if** 语句。**if** 语句后跟一个语句或复合语句，然后跟可选的 **else** 语句。这是 else 语句可出现的唯一上下文。

复合语句用大括号显式包围。编译器忽略跨栏约定，这有助于提高代码的可读性。

更正此错误

1. 用大括号将 **if** 语句后面的代码括起来。
2. 在 **else** 语句前面添加 **if** 语句。

请参见

参考

if...else 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1100: 应为“,”

在函数声明中参数之间缺少必需的逗号。

更正此错误

- 在函数声明中用逗号隔开每个参数。

请参见

参考

function 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1101: 已定义可见性修饰符

可见性修饰符在一个表达式中被多次应用。修饰符的其他应用均无效。

更正此错误

- 将可见性修饰符的应用次数减少到 1。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1102: 无效的可见性修饰符

可见性修饰符存在于不适当的上下文中。这可能会在以下情况下发生：将修饰符应用于不能使用修饰符的表达式时，或者将修饰符用于已经具有不兼容的修饰符的类或接口的成员。

更正此错误

1. 移除可见性修饰符或使用其他修饰符。

2. 确保应用于类成员的可见性修饰符与该类的可见性匹配。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1103: 缺少“case”或“default”语句

switch 语句块中的第一行代码不以 **case** 或 **default** 关键字开头。

更正此错误

- 在 **switch** 语句后添加 **case** 或 **default** 关键字。

请参见

参考

switch 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1104: 不匹配的“@end”; 未定义“@if”

@end 语句缺少一个与之匹配的 **@if** 语句。因为 **@end** 语句终止 **@if...@end** 块，所以每个 **@if** 语句必须有一个与之匹配的 **@end** 语句，反之亦然。

更正此错误

- 确保每个 **@end** 语句之前都有一个 **@if** 语句。

请参见

参考

@if...@elif...@else...@end 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1105: 不匹配的“@else”; 未定义“@if”

@else 语句缺少一个与之匹配的 **@if** 语句。**@else** 语句只能出现在 **@if**...**@end** 块中。

更正此错误

- 确保 **@else** 语句只出现在 **@if**...**@end** 块中。

请参见

参考

@if...@elif...@else...@end 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1106: 不匹配的“@elif”;未定义“@if”

@elif 语句缺少一个与之匹配的 **@if** 语句。**@elif** 语句只能出现在 **@if**...**@end** 块中。

更正此错误

- 确保 **@elif** 语句只出现在 **@if**...**@end** 块中。

请参见

参考

@if...@elif...@else...@end 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1107: 需要更多的源字符

某行代码在表达式关闭之前结束。此错误将发生于以下情况：需要您在同一行上具有开始和结束表达式并且您已经将该行分成两行或多行时。

更正此错误

- 在表达式开始的同一行上为其提供匹配的结束代码。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1108: 不兼容的可见性修饰符

可见性修饰符应用于不能使用该修饰符的类或接口，或者应用于用不兼容的整体修饰符定义的类或接口的成员。

更正此错误

1. 移除可见性修饰符或使用其他修饰符。
2. 确保应用于类成员的可见性修饰符与该类的可见性匹配。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

基于类的对象

JS1109: 此上下文中不允许有类定义

类声明存在于不适当的上下文中。只允许在主程序块、其他类、包或函数内部使用类声明。

更正此错误

- 在主程序块、另一个类、包或函数内部定义类。

请参见

参考

class 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1110: 表达式必须是一个编译时常数

在编译时未定义的表达式存在于只允许使用编译时常数的上下文中。

更正此错误

- 在此上下文中只使用编译时常数。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1111: 标识符已在使用中

标识符的名称在表达式中重复。

更正此错误

1. 如果两个实例引用同一个标识符，则移除该标识符的第二个定义。
2. 如果这些标识符应该不同，则赋予它们唯一的名称。

请参见

概念

脚本疑难解答

其他资源

JScript 变量和常数

JScript 参考

JS1112: 应为类型名称

所需的类型名称有拼写错误或被省略。在变量、常数、函数或参数的定义中，类型名称后面需要有一个分号。

更正此错误

- 确保在期望出现类型名称的位置使用有效的数据类型标识符。

请参见

概念

类型批注

脚本疑难解答

其他资源

JScript 参考

JS1113: 仅在类定义内有效

只在类定义中有效的指令或关键字存在于类定义外部。

更正此错误

1. 移除该项。
2. 确保代码处在类定义内部。

请参见

概念

脚本疑难解答

其他资源

基于类的对象

JScript 参考

JS1114: 未知的位置指令

该代码将无效的参数传递给位置指令。有效的参数是 **end**、**file=**、**line=** 和 **column=**。

更正此错误

- 确保在位置指令中只使用有效的参数。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

指令

JS1115: 在同一行上，指令后面不能有其他代码

指令后面有多余的代码。

更正此错误

- 将指令后面的行分成两行。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

指令

JS1118: 调试器指令错误或指令的位置错误

代码将无效的参数传递给 **@debug** 指令。有效的参数是 **on** 和 **off**。

- 或 -

代码包含不允许的指令。

更正此错误

1. 确保传递给 **@debug** 指令的参数为 **on** 或 **off**。
2. - 或 -
3. 将该指令移到另一个位置。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

指令

JS1119: 不能嵌套位置指令

代码包含嵌套的 **@position** 指令。

更正此错误

- 在开始一个 **@position** 指令之前关闭另一个。

请参见

参考

@position 指令

概念

脚本疑难解答

其他资源

JScript 参考

JS1120: 循环定义

该代码用类或接口本身定义类或接口。这可能会在以下情况下发生：代码包含两个类，每个类都是对另一个类的扩展。

更正此错误

- 确保在扩展类或接口时，基类或接口不依赖于正被定义的类或接口。

请参见

概念

脚本疑难解答

其他资源

基于类的对象

JScript 参考

JS1121: 已否决

该代码包含被否决的表达式。首选的替换表达式执行相同的任务。使用新方法，因为在该语言的以后版本中可能会取消对被否决表达式的支持。

更正此错误

1. 使用 encodeURI 方法而不使用 escape 方法。
2. 使用 getFullYear 方法而不使用 getYear 方法。
3. 使用 setFullYear 方法而不使用 setYear 方法。
4. 使用 substring 方法而不使用 substr 方法。
5. 使用 toUTCString 方法而不使用 toGMTString 方法。
6. 使用 decodeURI 方法而不使用 unescape 方法。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1122: 在当前上下文中使用“this”无效

该代码在 **static** 类或成员函数中使用 **this**。

更正此错误

1. 用对类的特定实例的限定引用替代 **this**。
2. 移除 **static** 修饰符。

请参见

概念

脚本疑难解答

其他资源

基于类的对象

JScript 参考

JS1123: 无法从此范围访问

某条语句尝试访问对象的成员，但是该成员具有防止从当前范围访问的可见性修饰符。例如，无法从类的外部访问私有字段。

更正此错误

1. 用其他方法（如通过公共方法）访问该成员。
2. 更改该代码访问的成员的修饰符。

请参见

概念

脚本疑难解答

其他资源

基于类的对象

JScript 参考

JS1124: 只有构造函数可以与包含它的类同名

非构造函数方法与它的类同名。

更正此错误

1. 通过消除返回类型和 **return** 语句使该方法成为构造函数。
2. 重命名该方法。

请参见

概念

脚本疑难解答

其他资源

基于类的对象

JScript 参考

JS1128: 类必须提供实现

final 方法或 **final** 类中的某种方法没有关联的主体。

更正此错误

1. 为该方法提供主体。
2. 移除 **final** 修饰符。

请参见

概念

JScript 修饰符

脚本疑难解答

其他资源

基于类的对象

JScript 参考

JS1129: 应为接口名称

定义的类实现一个不存在的接口。

更正此错误

- 在类定义中的 **implements** 关键字后面提供一个有效接口名称。

请参见

概念

脚本疑难解答

其他资源

基于类的对象

JScript 参考

JS1133: 永远不会到达 catch 子句

一个 **catch** 块跟在另一个捕捉所有错误的 **catch** 块后面。当 **catch** 语句的参数不具备特定类型时，该语句将捕捉所有错误。

更正此错误

- 检查所有 **catch** 语句，以确保 **catch** 语句每次只捕捉一个类型，最后一个 **catch** 语句捕捉非类型化错误。

请参见

参考

try...catch...finally 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1134: 无法扩展类型

表达式尝试扩展具有 **final** 修饰符的类型或类。

更正此错误

1. 不要尝试扩展类型或类。
2. 从类中移除 **final** 修饰符。

请参见

概念

JScript 修饰符

脚本疑难解答

其他资源

基于类的对象

JScript 参考

JS1135: 未声明变量

表达式中包括未用 **var** 语句定义的变量或拼错名称的变量，而且代码是在快速模式下编译的。在快速模式下编译的程序必须显式声明所有变量。

可为用命令行编译器编译的程序关闭快速模式。

更正此错误

1. 确保定义所有变量。
2. 确保所有标识符的拼写正确。
3. 用 **/fast-** 选项编译该程序以关闭快速模式。(仅限于命令行编译。)

请参见

参考

/fast

概念

脚本疑难解答

其他资源

JScript 变量和常数

JScript 参考

JS1136: 将变量保留为未初始化状态是危险的，而且使用起来很慢。

将变量保留为未初始化状态是危险的，而且使用起来很慢。您打算将此变量保留为未初始化状态吗？

用 **var** 语句定义的变量不具备指定的类型或在使用之前未被初始化。

更正此错误

1. 使用这些变量的类型批注。
2. 在使用前初始化所有变量。

请参见

概念

脚本疑难解答

其他资源

JScript 变量和常数

JScript 参考

JS1137: 这是一个新的保留字，不应用作标识符

该代码将新保留字用作标识符的名称。

更正此错误

- 更改标识符的名称以排除保留字。

请参见

概念

JScript 保留字 (JScript)

脚本疑难解答

其他资源

JScript 参考

JS1140: 在基类构造函数调用中不允许

该代码尝试将当前类的属性传递给基类构造函数。这是不允许的，因为当前类的属性只有在基类构造之后才存在。

更正此错误

- 确保不将正被构造的类的属性传递给基类构造函数。

请参见

概念

脚本疑难解答

其他资源

基于类的对象

JScript 参考

JS1141: 此构造函数或属性的 getter/setter 方法并不用于直接调用

表达式直接调用了类的构造函数方法。

- - 或 -

表达式直接调用了属性的 getter 或 setter 方法。

- 这两种方法不能被直接调用。

更正此错误

1. 不调用构造函数方法。

   - 或 -

2. 使用“.”语法访问该属性。

请参见

参考

function get 语句

function set 语句

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1142: 此属性的 get 和 set 方法相互不匹配

该代码为属性定义 **get** 和 **set** 访问器。但是，**get** 访问器的返回数据类型与 **set** 访问器的参数类型不同。

更正此错误

- 确保 **get** 访问器的返回类型与 **set** 访问器的参数类型匹配。

请参见

参考

function get 语句

function set 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1143: 必须从 System.Attribute 派生自定义属性类

用作自定义属性的类不是从 **System.Attribute** 派生。只有将 **System.Attribute** 作为基类的类可以用作自定义属性。

更正此错误

- 确保自定义属性类将 **System.Attribute** 用作基类。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1144: 在自定义属性中只允许使用基元类型

表达式尝试将基元类型以外的内容传递给自定义属性构造函数，或者表达式在需要属性的位置使用非属性的内容。

更正此错误

- 确保表达式使用属性，并确保它只将基元类型传递给自定义属性构造函数。

请参见

概念

编写自定义属性

脚本疑难解答

其他资源

JScript 参考

JS1146: 未知的自定义属性类或构造函数

该代码在需要属性的位置使用不是属性或自定义属性构造函数的内容。

更正此错误

- 确保该代码在此上下文中使用属性或自定义属性构造函数。

请参见

概念

编写自定义属性

脚本疑难解答

其他资源

JScript 参考

JS1148: 参数太多

参数太多。额外的参数将被忽略。

表达式向函数或方法传递的参数数量多于在该函数或方法的定义中指定的数量。

更正此错误

1. 对函数或方法使用正确数量的参数。
2. 检查以保证表达式没有多余的参数。

请参见

参考

function 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1149: 此 with 语句使该名称的使用不明确

表达式使用 **with** 语句访问某个类，该类的某个成员与当前范围中的标识符同名。编译器无法区分要访问哪个标识符。

更正此错误

1. 重命名当前范围中类或标识符的成员。
2. 避免使用 **with** 语句。

请参见

参考

with 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1150: eval 的存在使该名称的使用不明确

该程序使用 **eval** 语句并在关闭快速模式时编译。

关闭快速模式时，**eval** 语句允许在运行时用局部范围声明新变量。这些新变量可隐藏全局变量，如果某个变量未在局部范围中显式定义，则这样做可能会使对它的任何引用变得不明确。

更正此错误

1. 在 **fast** 选项处于打开状态时编译。
2. 避免使用 **eval** 语句。
3. 只使用当前的局部范围中可用的变量。

请参见

参考

eval 方法 (JScript)

/fast

概念

脚本疑难解答

其他资源

JScript 参考

JS1151: 对象没有这样的成员

表达式引用基于类的对象的成员，但是该对象不具备该名称的成员。

如果代码包含在 ASP.NET 页中，则该代码可在使用 **this** 语句的 **<script runat="server">** 块中定义构造函数。**expando** 修饰符应当用于 **<script runat="server">** 块的每个构造函数定义中。

例如，ASP.NET 页的以下代码使用以 **expando** 修饰符标记的构造函数。

```
<script runat="server">
expando function Person(name) {
   // If the expando modifier was not applied to the definition of Person,
   // the this statment in the following line of code would generate error
   // JS1151

   this.name = name;
}
</script>

<%
var fred = new Person("Fred");
Response.Write(fred.name);
%>
```

更正此错误

1. 确保表达式引用基于类的对象的现有成员并且成员名称拼写正确。
2. 确保将 **expando** 修饰符应用于 **<script runat="server">** 块中的每个构造函数声明中。

请参见

参考

this 语句

expando 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1152: 不能在 Expando 类上定义 Item 属性

不能在 Expando 类上定义 Item 属性。Item 保留给 expando 字段使用。

expando 类的成员名称为 **Item**。这是不允许的，因为它将导致与 **Item** 属性发生冲突，该属性是为 expando 类隐式定义的。

更正此错误

- 对名为 **Item** 的类成员进行重命名。

请参见

参考

expando 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1153: 不能在 Expando 类上定义 get_Item 或 set_Item

不能在 Expando 类上定义 get_Item 或 set_Item。这些方法保留给 expando 字段使用。

expando 类的成员名称为 **get_Item** 或 **set_Item**。这是不允许的，因为它将导致与 **get_Item** 或 **set_Item** 属性发生冲突，这些属性是为 expando 类隐式定义的。

更正此错误

- 对名为 **get_Item** 或 **set_Item** 的类成员进行重命名。

请参见

参考

expando 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1155: 基类定义 get_Item 或 set_Item，不能创建 expando 类

基类定义 get_Item 或 set_Item，不能创建 expando 类。这些方法保留给 expando 字段使用。

expando 类扩展具有名为 **get_Item** 或 **set_Item** 的成员的基类。这是不允许的，因为它将导致与 **get_Item** 或 **set_Item** 属性发生冲突，这些属性是为 expando 类隐式定义的。

更正此错误

1. 对名为 **get_Item** 或 **set_Item** 的基类成员进行重命名。
2. **不要从基类继承。**
3. **从类定义中移除 expando 修饰符。**

请参见

参考

expando 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1156: 某个基类已标记为 expando; 当前规范将被忽略

用 **expando** 修饰符标记的类扩展 expando 基类。但是，从 expando 基类派生的类将自动成为 expando; 不必显式添加 **expand** 修饰符。

更正此错误

- 从派生类中移除 **expando** 修饰符。

请参见

参考

expando 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1157: 抽象方法不能是私有的

某个方法同时拥有 **abstract** 和 **private** 修饰符。这是不允许的，因为私有方法是从类内部访问的，但抽象方法是从类的外部继承的。

更正此错误

- 从方法声明中移除 **abstract** 或 **private** 修饰符。

请参见

参考

abstract 修饰符

private 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1158: 此类型的对象是不可建立索引的

该代码尝试索引对象的元素，但是该对象的数据类型不支持索引。

可建立索引的对象（如数组和 JScript 对象）的元素是使用 [] 表示法访问的。

更正此错误

1. 更改该对象的数据类型。
2. 移除索引访问器。

请参见

概念

脚本疑难解答

类型化数组

其他资源

JScript 参考

JScript 对象

JS1159: 语法错误。使用“static classname {...}”定义类初始值设定项

代码块紧跟类中的修饰符。JScript 修饰符只能用于类成员。在以下两种情形下可能会产生此结果：

- 您打算定义类初始值但是省略了类名称。
- 您打算定义方法或属性访问器但是省略了 **function**、**function get** 或 **function set** 语句。

更正此错误

1. 如果您打算定义类初始值，请使用 **static** 语句的正确语法。
2. 如果您打算定义方法或属性访问器，请使用 **function**、**function get** 或 **function set** 语句的正确语法。

请参见

参考

static 语句
function 语句
function get 语句
function set 语句

概念

脚本疑难解答
JScript 修饰符

其他资源

JScript 参考

JS1160: 属性列表不适用于当前上下文

该代码指定的属性列表不能应用于当前上下文中。

更正此错误

- 确保只使用应用于当前上下文中的属性。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1161: 包内只允许使用类

该代码在包内部声明函数、变量或常数。只有类和接口才能在包内定义。

更正此错误

- 移除包中不是针对类或接口的所有定义。

请参见

参考

package 语句

概念

脚本疑难解答

其他资源

JScript 参考

JScript 对象

JS1162: Expando 类不应实现 IEnumerable

Expando 类不应实现 IEnumerable。此接口在 expando 类上隐式定义。

expando 类显式实现 **IEnumerable**。这是不必要的，因为带有 **expando** 修饰符的类隐式实现 **IEnumerable**。

更正此错误

- 不要显式实现 **IEnumerable**。

请参见

参考

expando 修饰符

IEnumerable Interface

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1163: 指定的成员不符合 CLS

该程序中包含 **CLSCompliantAttribute** 属性，并且编译器已检测到不符合公共语言规范 (CLS) 的类成员。该错误的一些可能的原因是：

- 成员名称不符合 CLS。符合 CLS 的名称不能以下划线 (_) 开头，不能包含美元符号 ($)，也不能与另一个公共成员的名称仅在大小写上不同。
- 如果该成员是公共方法，则公共语言运行库中不可用的数据类型用于对参数或返回类型进行类型批注。
- 如果该成员是字段或属性，则公共语言运行库中不可用的数据类型用于对字段或属性进行类型批注。

数据类型在公共语言运行库中不可用的原因有几个。

- 该类型在类中定义但是不能被公开访问。
- 该类型已经定义但未标记为符合 CLS。
- 该类型是不符合 CLS 的基元类型。例如，**uint** 是不符合 CLS 的基元类型。符合 CLS 的相应系统类型是 **System.UInt32**。
- 该类型是一个内部 JScript 对象，它们都不符合 CLS。常用的 JScript 对象（**Array**、**Date**、**Error**、**RegExp** 和 **Function**）与符合 CLS 的系统类型（**System.Array**、**System.DateTime**、**System.Exception**、**System.Text.RegularExpressions.RegEx** 和 **System.EventHandler**）相对应。

更正此错误

1. 确保成员名称不以下划线 (_) 开头，不包含货币符号 ($)，或者与另一个成员名称只是大小写不同。
2. 对于公共方法的参数或返回类型以及公共字段和属性的类型来说，确保它们是公共语言运行库数据类型或已标记为符合 CLS 的可公开访问的类。

请参见

参考

CLSCompliantAttribute Class

概念

脚本疑难解答

编写符合 CLS 的代码

公共语言规范

其他资源

JScript 参考

JS1164: 成员是不可删除的

该代码尝试"删除"不能被删除的对象成员。只有 expando 属性(已动态添加到对象中的属性)才能被删除。

更正此错误

- 不要尝试"删除"该对象成员。

请参见

参考

delete 运算符
expando 修饰符

概念

脚本疑难解答
JScript Object 对象

其他资源

JScript 参考

JS1165: 应为包名称

import 语句后面没有有效的包名称。

更正此错误

- 在 **import** 语句后面包括有效的包名称。

请参见

参考

import 语句

package 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1169: 表达式不起任何作用

表达式返回从未使用的值。

更正此错误

1. 移除该表达式。
2. 将该表达式的值用作函数或运算符的参数。

请参见

概念

脚本疑难解答

JScript 表达式

其他资源

JScript 参考

JS1170: 隐藏基类中声明的另一个成员

派生类中的成员隐藏(重新定义其含义)在基类中定义的字段、类、接口或枚举。这是不允许的;只能隐藏方法和属性。

更正此错误

- 确保任何派生类成员都不隐藏基于类的字段、类、接口或枚举定义。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1171: 无法更改基方法的可见性规范

派生类中的方法重写基类方法，而且这两种方法的可见性修饰符是不同的。这是不允许的，因为基类成员的可见性不能更改。

更正此错误

- 对派生类方法的可见性修饰符进行修改，使其与基类方法的可见性相匹配。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1172: 方法隐藏基类中的抽象方法

派生类中用 **hide** 修饰符定义的方法与基类中的抽象方法同名。这是不允许的，因为抽象方法需要从派生类实现，隐藏方法将会阻止实现。

更正此错误

1. 从基类方法中移除 **abstract** 修饰符并给该方法提供实现。
2. 从派生类方法中移除 **hide** 修饰符。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1173: 方法与基类中的某个方法匹配

方法与基类中的某个方法匹配。指定“override”或“hide”取消显示此消息。

派生类中未用版本安全修饰符定义的方法与基类方法相匹配。而且，该程序是用 **/versionsafe** 选项进行编译的。用 **/versionsafe** 选项进行编译时，与基类方法匹配的每个方法必须使用版本安全修饰符（**hide** 或 **override**）。

更正此错误

- 将适当的版本安全修饰符应用于方法声明。

请参见

参考

/versionsafe

hide 修饰符

override 修饰符

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1174: 方法与基类中某个不可重写的方法匹配

方法与基类中某个不可重写的方法匹配。指定“hide”取消显示此消息。

带有 **final** 修饰符的基类方法与派生类中的方法相匹配。另外，派生类方法具有 **override** 修饰符或代码正在用 **/versionsafe** 选项进行编译。final 方法不能被重写，如果使用的是 **/versionsafe** 选项，则必须为派生类方法显式指定 **hide** 修饰符。

更正此错误

- 将 **hide** 修饰符用于派生类中的方法。

请参见

参考

/versionsafe
hide 修饰符
override 修饰符
final 修饰符

概念

脚本疑难解答
JScript 修饰符

其他资源

JScript 参考

JS1175: 基类中没有要隐藏的成员

派生类方法具有 **hide** 修饰符，但是基类中没有与之匹配的方法。不能隐藏不存在的方法。

更正此错误

1. 从方法声明中移除 **hide** 修饰符。

   - 或 -

2. 向基类添加匹配方法。

请参见

参考

hide 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

JS1176: 基类中的方法具有不同的返回类型

派生类实现接口或扩展基类。派生类的某个方法具有与接口或基类中的方法相同的名称和参数列表，但是返回类型却不同。除了返回类型不同之外，两个方法不能具有相同的名称和参数列表。

更正此错误

1. 对派生类中的函数进行重命名。
2. 更改返回类型或方法，以便它们在派生类和接口或基类中相匹配。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1177: 与属性冲突

在派生类中定义的字段或方法与基类中的属性同名。这会在使用该名称引用派生类的成员时产生二义性，因此是不允许的。

更正此错误

- 对基类属性或派生类成员进行重命名。

请参见

参考

function set 语句

function get 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1178: “override”和“hide”不能一起使用

某个方法用 **override** 和 **hide** 两个修饰符定义。但是，每个类成员只能有一个版本安全的修饰符。

更正此错误

- 从方法声明中移除 **override** 或 **hide**。

请参见

参考

hide 修饰符

override 修饰符

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1179: 无效的选项

使用 **@option** 指令的表达式不包括有效选项。目前不支持 **@option** 指令。

更正此错误

- 不要使用 **@option** 指令。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1180: 基类中没有要重写的匹配方法

基类中没有与使用 **override** 修饰符的派生类方法相匹配的方法。不能重写不存在的方法。

更正此错误

1. 从方法声明中移除 **override** 修饰符。

   - 或 -

2. 向基类添加匹配方法。

请参见

参考

override 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1181: 对构造函数无效

类构造函数使用无效的修饰符。只能将可见性修饰符（**public**、**private**、**protected** 和 **internal**）、版本安全的修饰符（**hide** 和 **override**）或 **final** 和 **expando** 修饰符应用于构造函数。

更正此错误

- 使用该构造函数的有效修饰符。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

基于类的对象

JS1182: 无法从构造函数或 void 函数返回值

表达式尝试从构造函数或 **void** 函数返回值。构造函数自动返回指向构造函数的指针，它们不返回值。用 **void** 返回类型定义的函数无法返回值。

更正此错误

1. 移除 return 语句。
2. 为函数指定非 **void** 的返回类型。

请参见

参考

function 语句

return 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1183: 不止一个方法或属性与此参数列表匹配

正在调用重载方法或属性的表达式找不到与传递的参数类型完全匹配的参数。在这种情况下，编译器尝试确定哪个重载函数要求最少数量的参数数据类型强制。此错误表示编译器找到了多个与具有相同数量的数据类型强制的参数相匹配的函数。

更正此错误

- 检查该重载函数接受的数据类型，并确保参数的数据类型只与一个重载函数相匹配。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1184: 不止一个构造函数与此参数列表匹配

正在调用重载构造函数的表达式未找到与传递的参数的类型完全匹配的参数。在这种情况下，编译器尝试确定哪个重载构造函数要求最少数量的参数数据类型强制。此错误表示编译器找到了多个与具有相同数量的数据类型强制的参数相匹配的构造函数。

更正此错误

- 检查该重载构造函数接受的数据类型，并确保参数的数据类型只与一个重载构造函数相匹配。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JScript 参考

JS1185: 无法从此范围访问基类构造函数

某个类扩展基类，并且该基类的构造函数不可从当前范围访问。这会在可见性修饰符用于基类或基类的构造函数时发生。

更正此错误

1. 使用基类构造函数的其他可见性修饰符。

2. 如果该基类是用 **internal** 修饰符定义的，请在与基类相同的包中定义派生类。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1186: 八进制文本被否决

八进制文本表示代码中的数字。在八进制文本中，整数之前有一个或多个零 (0)。应使用小数或十六进制文本而不应使用八进制文本。

更正此错误

1. 如果数字是小数，请从该数字中移除前导零。
2. 将八进制数字转换为十进制或十六进制数字。

请参见

概念

脚本疑难解答

数值数据

其他资源

JScript 参考

JS1187: 变量可能未初始化

表达式正在访问某个变量的值，该变量既未初始化又未定义为特殊的数据类型。

更正此错误

1. 在使用变量之前先对其进行初始化。
2. 使用类型批注声明变量。

请参见

参考

var 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1188: 从此位置调用基类构造函数无效

表达式正在从构造函数定义内第一行之外的位置调用基类构造函数 **super**。

更正此错误

- 确保只从构造函数声明内的第一行调用基类构造函数。

请参见

参考

super 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1189: 以这种方式使用 super 关键字无效

表达式使用静态类成员中的 **super** 语句。这是不允许的，因为静态成员与类本身相关联，而 **super** 语句用于访问类的当前实例的基类成员。

更正此错误

1. 将 **super** 替换为对基类的特定实例的限定引用。
2. 移除 **static** 修饰符。

请参见

参考

super 语句

static 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

JS1190: 这样的 finally 块的运行速度会很慢并且可能导致混乱

这样的 finally 块的运行速度会很慢并且可能导致混乱。这是有意的吗？

一条语句(**return** 或 **break**)会导致对程序的控制离开 **finally** 块。如果在 **try** 或 **catch** 块中有 **return** 或 **break** 语句，这可能会产生意想不到的结果。

finally 块中的代码总是在 **try** 块中的代码之后运行，如果有错误的话，则在 **catch** 块中的代码之后运行。例如，如果在 **try** 块中遇到 **return** 语句，则在执行 **return** 语句之前先执行 **finally** 块。如果在 finally 块中有另一个 **return** 语句，那么将执行该语句，而不执行最初的 return 语句。若要避免这种可能会导致混淆的情况，请不要在 **finally** 块中使用 **return** 语句。

```
function test() {
   try {
      return(5);   // Attempt to return 5.
   } catch(e) {
      print(e);
   } finally {
      return(10);  // This gets run first, returning 10 instead of 5.
   }
}
print(test());     // Prints 10, not 5.
```

更正此错误

1. 确保不在 **finally** 块中使用 **return** 和 **break** 语句。
2. 如果打算在 **try** 和 **catch** 块之后执行 **return** 或 **break** 语句，则将它们移到紧跟 **finally** 块之后的位置。

请参见

参考

try...catch...finally 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1191: 应为“,”。写入“identifier : Type”而不是“Type identifier”来声明类型化参数

函数声明中包括未用逗号隔开的参数，或者包括被指定为 `Type identifier` 而非 `identifier : Type` 的类型批注参数。

更正此错误

1. 确保所有参数都用逗号隔开。
2. 用 `identifier : Type` 语法指定类型批注参数。

请参见

参考

function 语句

概念

脚本疑难解答

其他资源

JScript 参考

JScript 函数

JS1192: 抽象函数不能有函数体

函数体与方法或属性关联，但是该方法或属性用 **abstract** 修饰符标记或者处在接口中。

更正此错误

1. 移除函数体。
2. 更改修饰符。
3. 使用类而不使用接口。

请参见

参考

class 语句

interface 语句

概念

JScript 修饰符

脚本疑难解答

其他资源

JScript 参考

JS1193: 应为“,”或“)”

对函数、方法或构造函数的调用缺少逗号或右括号。

更正此错误

- 添加逗号或右括号。

请参见

概念

脚本疑难解答

其他资源

JScript 参考
JScript 函数
JScript 对象

JS1194: 应为“,”或“]”

对数组元素的引用缺少逗号或右方括号。

更正此错误

- 添加逗号或右方括号。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JScript 数组

JS1195: 应为表达式

在此上下文中，需要值或引用(表达式的结果)，但是尚未提供表达式。该代码可能包括不返回值的语句。

更正此错误

- 确保使用的是表达式。

请参见

概念

脚本疑难解答

JScript 表达式

其他资源

JScript 参考

JS1196: 不应为“;”

在此上下文中有不适当的分号，或者用分号终止的语句有错误。

更正此错误

1. 移除分号。
2. 确保用分号终止的语句没有任何其他错误。

请参见

概念

脚本疑难解答

JScript 语句

其他资源

JScript 参考

JS1197: 错误太多

错误太多。该文件可能不是 JScript .NET 文件。

该代码已生成太多的错误。最常见的原因是尝试编译的文件不是 JScript .NET 文件。或者，代码可能只是有几个编译器无法恢复的错误，从而导致了许多其他错误。

更正此错误

1. 确保编译器正在编译的文件是 JScript .NET 文件。

2. 修复前面的几个错误，然后重新编译。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1198: 语法错误。写入“var identifier : Type”而不是“Type identifier”来声明类型化变量

您的代码包含形式为 `Type identifier` 的 C 样式字段声明。若要在 Jscript 中声明字段，请使用 `var identifier : Type` 语法。

更正此错误

- 使用 `var identifier : Type` 语法声明字段。

请参见

参考

var 语句

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1199: 语法错误。写入“function identifier(...) : Type{”而不是“Type identifier(...){”来声明类型化函数

该代码包含形式为 `Type identifier(...)` 的 C 样式方法声明。若要在 Jscript 中声明方法，请使用 `function identifier(...) : Type` 语法。

更正此错误

- 使用 `function identifier(...) : Type` 语法声明方法。

请参见

参考

function 语句

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1200: 属性声明无效

属性声明无效。getter 不能有参数，而 setter 必须有一个参数。

该代码定义带有一个或多个参数的属性 getter 函数，或者定义不带参数或带有多个参数的属性 setter 函数。getter 函数的定义必须不带任何参数，而 setter 函数必须只带一个参数。

更正此错误

1. 定义不带任何参数的属性 getter 函数。
2. 定义只带一个参数的属性 setter 函数。

请参见

参考

function get 语句
function set 语句

概念

脚本疑难解答

其他资源

JScript 参考
基于类的对象

JS1203: 表达式没有地址

代码中的"and"符 (**&**) 后跟不包含地址的表达式。"&"符只应位于变量名（包含地址）之前，并且只应用于通过引用将变量传递给通过引用接受该参数的函数。

通过引用传递变量时，允许函数更改该变量的值。

注意
JScript 不允许使用引用参数来定义函数。JScript 提供了"&"符，它允许调用采用引用参数的外部对象。

更正此错误

- 确保 (**&**) 符位于对函数的调用中的变量名称的前面，并确保函数接受通过引用传递的参数。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1204: 并未提供所有必需的参数

该代码调用函数或方法时使用的参数少于定义该函数或方法要接受的参数。尽管该函数或方法将为缺少的参数提供默认值，但是最好为期望的所有参数提供值。

更正此错误

- 确保将所有必需的参数都传递给函数或方法。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1205: 此赋值创建一个立刻被丢弃的 expando 属性

该代码向某个对象不存在的属性赋值，但是该对象不支持 expando 属性。编译器尝试创建 expando 属性，但是这些属性因不能被添加到对象中而被丢弃。

更正此错误

1. 使用支持 expando 属性的对象。
2. 不要尝试将 expando 属性添加到不支持它们的对象中。

请参见

概念

脚本疑难解答
高级类创建
JScript Object 对象

其他资源

JScript 参考

JS1206: 您打算在此处写一条赋值语句吗?

该代码将赋值运算符用作条件语句的条件表达式。您可能打算使用相等运算符或全等运算符。

更正此错误

1. 将赋值运算符 (=) 更改为相等运算符 (==) 或全等运算符 (===)。
2. 将该赋值运算符移到紧邻条件语句之前，然后将赋值运算符的左操作数用作条件表达式。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JScript 条件结构

JS1207: 您是否打算将一条空语句用于 if 语句的这个分支？

该代码有一个后跟分号的 **if** 语句。分号被解释为空语句的结束符，该语句在 **if** 语句中的条件表达式为真时执行。

更正此错误

1. 移除分号。
2. 使 **if** 语句后跟空块 ({})。

请参见

参考

if...else 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1208: 指定的转换或强制不可能实现

该代码有一个无法执行的类型转换或强制。这表示原始数据类型与目标转换类型中的数据没有明显的相似之处。

更正此错误

- 确保提供的数据与转换的数据类型或它被强制成的数据类型兼容。

请参见

概念

脚本疑难解答

类型转换

JScript 中的强制

其他资源

JScript 参考

JS1209: final 和 abstract 不能一起使用

该代码包含一个用 **final** 和 **abstract** 修饰符标记的类或类成员。这些修饰符不能结合使用。

更正此错误

- 使用 **final** 或 **abstract** 修饰符。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1210: 必须是一个实例

该代码尝试使用类名称来访问非静态类成员。只有静态类成员才与类本身相关联；非静态成员与特定的类实例相关联并通过该特定类实例来进行访问。

更正此错误

- 确保通过类实例来访问非静态类成员。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1212: 除非声明类被标记为抽象的，否则不能是抽象的

该代码包含一个用 **abstract** 修饰符标记的成员，但是它所属的类未被标记为 **abstract**。如果某个类至少有一个成员是 **abstract**，则该类必须被标记为 **abstract**。

更正此错误

- 确保具有 **abstract** 成员的所有类都被标记为 **abstract**。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1213: 枚举的基类型必须是基元整型

该代码有一个枚举被声明为具有一个非基元整数类型的基础基类型。枚举的有效基类型为整型数据类型：**int**、**short**、**long**、**byte**、**uint**、**ushort**、**ulong** 和 **sbyte**。

更正此错误

- 确保每个枚举的基类型都是有效的整型数据类型。

请参见

参考

enum 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1214: 不可能构造抽象类的实例

该代码尝试用 **new** 运算符构造抽象类的实例。用 **abstract** 修饰符标记的类无法实例化。

更正此错误

1. 从该类中移除 **abstract** 修饰符。
2. 定义一个扩展抽象类的类，并为每个抽象方法和属性提供实现。
3. 不要尝试实例化抽象类。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1215: 将 JScript 数组转换为 System.Array 将导致内存分配和数组复制

该代码将 JScript **Array** 对象转换为类型化数组 (**System.Array**)。

注意
这可通过以下步骤来完成：分配足够的内存以存储类型化数组的副本；将 JScript 数组的元素复制到类型化数组中。

因此，除非该代码在修改类型化数组后将其复制回 JScript 数组，否则这些修改将不在 JScript 数组中反映出来。

更正此错误

- 使用显式类型转换将 JScript 数组转换为类型化数组。

请参见

概念

脚本疑难解答

类型转换

其他资源

JScript 参考

JScript 数组

JS1216: 静态方法不能是抽象的

某个方法同时具有 **static** 和 **abstract** 修饰符。这是不允许的，因为静态方法与类本身相关联，但是抽象方法是从类的外部继承的。

更正此错误

- 从方法声明中移除 **static** 或 **abstract** 修饰符。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1217: 静态方法不能为 final

某个方法同时具有 **static** 和 **final** 修饰符。这是不允许的，因为静态方法已经是 final 方法。静态方法与类本身相关联，因此不能被重写。

更正此错误

- 从方法声明中移除 **static** 或 **final** 修饰符。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1218: 静态方法不能重写基类方法

某个方法同时具有 **static** 和 **override** 修饰符。这是不允许的，因为静态方法与当前类相关联，重写只适用于类实例。

更正此错误

- 从方法声明中移除 **static** 或 **override** 修饰符。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1219: 静态方法不能隐藏基类方法

某个方法同时具有 **static** 和 **hide** 修饰符。这是不允许的，因为静态方法与当前类相关联，而且 **hide** 只适用于与类实例关联的成员。

更正此错误

- 从方法声明中移除 **static** 或 **hide** 修饰符。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1220: Expando 方法不能重写基类方法

某个方法同时具有 **expando** 和 **override** 修饰符。这是不允许的，因为 expando 方法与当前类相关联，而且 **override** 只适用于与类实例关联的成员。

更正此错误

- 从方法声明中移除 **expando** 或 **override** 修饰符。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1221: 变量参数列表必须为数组类型

函数的定义中有一个类型未批注为类型化数组的参数数组（或变量参数列表）。参数数组必须是函数声明中的最后一个参数，它的前面有一个省略号 (...)，其类型被批注为类型化数组。参数数组不能是 JScript **Array** 对象。

更正此错误

- 将变量参数列表的类型批注为类型化数组。

请参见

参考

function 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1222: Expando 方法不能是抽象的

某个方法同时具有 **expando** 和 **abstract** 修饰符。Expando 方法永远不能为 abstract。

更正此错误

- 从方法声明中移除 **expando** 或 **abstract** 修饰符。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1223: 没有函数体的函数应该是抽象的

类内的函数（方法或属性）没有主体，并且未用 **abstract** 修饰符加以标记。用 **abstract** 修饰符标记的函数不得有主体，而带有主体的函数不得用 **abstract** 修饰符加以标记。

更正此错误

1. 用 **abstract** 修饰符标记该函数。
2. 向函数中添加一个主体。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1224: 此修饰符不能在接口成员上使用

接口成员有一个不允许使用的修饰符。接口成员只允许使用 **public** 修饰符。

更正此错误

- 确保对接口成员只应用 **public** 修饰符。

请参见

参考

interface 语句

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1226: 不能在接口中声明变量

变量是在接口中声明的，这是不允许的。可在类中声明变量以创建字段。

更正此错误

- 确保变量声明不在接口声明中出现。

请参见

参考

interface 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1227: 不能在接口中声明接口

接口声明嵌套在接口声明中，这是不允许的。接口声明只允许出现在全局范围或包中。

更正此错误

- 确保不出现嵌套的接口声明。

请参见

参考

interface 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1228: 枚举成员声明不应使用“var”关键字

枚举声明中包含 **var** 关键字，但是不允许在枚举声明中使用变量声明。

更正此错误

- 移除 **var** 关键字或变量声明。

请参见

参考

enum 语句

var 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1229: import 语句在此上下文中无效

该代码中出现的 **import** 语句未处在全局范围中。import 语句只能出现在全局范围中。

更正此错误

- 将 **import** 语句从当前位置移到主程序块(全局范围)中。

请参见

参考

import 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1230: 此上下文中不允许枚举声明

枚举声明存在于不适当的上下文中。只允许在主程序块、类、包或函数内部使用枚举声明。

更正此错误

- 在主程序块、类、包或函数内部定义枚举。

请参见

参考

enum 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1231: 属性对于此类型的声明无效

该代码将属性应用于不接受属性的声明中。

更正此错误

- 确保不将该属性应用于此类型的声明中。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1232: 此上下文中不允许包声明

包声明出现在全局范围以外的上下文中。这是不允许的，包只能在全局范围中声明。

更正此错误

- 确保所有的包都在全局范围中定义。

请参见

参考

package 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1233: 构造函数不能有返回类型

类的构造函数指定了一个返回类型。然而，构造函数会自动返回对构造类实例的引用；它不返回值。

更正此错误

1. 从构造函数中移除返回类型规范。
2. 通过用不同于类名称的名称重命名方法，将构造函数更改为方法。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1234: 在库的内部只允许类型和包定义

正在编译代码以创建库，但是它包含不允许使用的声明。用于创建库的代码只能包含类、接口和包。

更正此错误

- 确保该代码只包含类、接口和包。

请参见

参考

/target:library
class 语句
interface 语句
package 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1235: 无效的调试指令

@debug 指令和无效的选项一起使用。有效的选项是 `on` 和 `off`。

更正此错误

- 对于 **@debug** 指令只使用 `on` 和 `off` 选项。

请参见

参考

@debug 指令

概念

脚本疑难解答

其他资源

JScript 参考

JS1236: 此类型的属性必须是唯一的

该代码将某个属性多次应用于某标识符，但是该属性只能应用一次。

更正此错误

- 确保该属性对于每个标识符只应用一次。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1237: 非静态嵌套类型只能由嵌套在同一类中的非静态类型扩展

类包含嵌套的类定义，并且该嵌套类未用 **static** 修饰符加以标记。另一个已定义的类扩展该嵌套类，但是此扩展类要么不包含正确的修饰符，要么不是在与该嵌套类相同的类中定义的。只有另一个嵌套在同一类中的非静态类可以扩展非静态嵌套类。

更正此错误

1. 确保只用非静态嵌套类扩展嵌套在同一类中的非静态类。

2. 将 **static** 修饰符应用于要被扩展的嵌套类。这使得非嵌套类和静态嵌套类都能够扩展该嵌套类。

请参见

参考

static 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1238: 如果有针对该属性 (property) 的属性 (attribute)，则必须在 getter 属性 (property) 上指定它

用 setter（**functionset** 声明）和 getter（**functionget** 声明）定义的属性 (Property) 有一个应用于 setter 的属性 (Attribute)，这是不允许的。所有属性 (Attribute) 必须显式应用于 getter（如果 getter 存在的话）。编译器将这些属性 (Attribute) 隐式应用于 setter。

更正此错误

- 将属性 (Attribute) 应用于 getter 属性 (Property)。

请参见

参考

function set 语句

function get 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1239: 在 try 语句的 catch 块之外使用 throw 时必须带参数

在 **catch** 块外使用了不带参数的 **throw** 语句。可以使用不带参数的 **throw** 语句的唯一的地方是 **catch** 块内，在这种情况下，它再次引发已捕获的错误。

更正此错误

1. 向 **throw** 语句传递一个参数。
2. 将 **throw** 语句移到 catch 块中。

请参见

参考

try...catch...finally 语句

throw 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1240: 变量参数列表必须是最后一个参数

函数的定义中有后跟其他参数的参数数组(或变量参数列表)。这是不允许的，因为参数数组必须是最后一个参数。

更正此错误

- 确保参数数组是函数定义中的最后一个参数。

请参见

参考

function 语句

概念

脚本疑难解答

其他资源

JScript 参考

JS1241：未能找到类型，是否缺少程序集引用?

对类型的限定引用使用外观类似包名称的限定符。但在包中找不到该类型，或者找不到包。这可能会在为包提供类型的程序集未被引用的情况下发生。

更正此错误

1. 确保该类型存在于所提供的包中。
2. 确保已打开 **/autoref** 选项或者已使用 **/reference** 选项显式引用了程序集。

请参见

参考

import 语句

/reference

/autoref

概念

脚本疑难解答

其他资源

JScript 参考

JS1242: 格式错误的八进制文本被作为十进制文本处理

文本数字的开头是前导零，并且没有小数点，这表示它是一个八进制（基数为 8）文本。然而，它还包含非八进制的数字 8 或 9。编译器会将该数字解释为十进制（基数为 10）数字。

更正此错误

1. 如果文本应该是十进制文本，请移除前导零。

2. 如果文本应该是八进制文本，请确保它只使用 0 到 7 的数字。

请参见

概念

脚本疑难解答

数值数据

其他资源

JScript 参考

JS1243：从静态范围无法访问非静态成员

静态方法或属性访问了类的非静态成员。静态类成员与类本身相关联，并且确实拥有与特定实例的成员有关的信息，而非静态成员与特定实例相关联。这意味着静态方法和属性无法访问非静态成员。

当类的实例作为参数传递给静态方法时，该方法可间接访问非静态成员。该静态方法可访问此类实例的所有成员，包括用 **private** 修饰符标记的成员。

更正此错误

1. 更改这些修饰符，以便被访问的成员和访问成员都是静态的或都是非静态的。
2. 将类的实例传递给静态方法。

请参见

参考

static 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1244: 必须使用类名访问静态成员

代码试图通过类实例访问静态类成员。静态类成员与类本身相关联，不能从类实例访问；它们必须用限定符形式的类名称直接访问。

更正此错误

- 确保用类名称访问静态成员。

请参见

参考

static 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1245: 无法使用类名访问非静态成员

该代码尝试使用类名称来访问非静态类成员。只有静态类成员才与类本身相关联；非静态成员与特定的类实例相关联并通过该特定类实例来进行访问。

更正此错误

- 确保通过类实例来访问非静态类成员。

请参见

参考

static 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1246: 类型没有这种静态成员

该代码尝试使用类名称访问成员(只访问静态类成员)，但是未找到匹配的静态成员。

更正此错误

- 当使用类名称访问成员时，请确保该成员是静态的。

请参见

参考

static 修饰符

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1247: 循环条件为函数引用

循环条件为函数引用。是要调用该函数吗？

在循环语句的条件表达式部分中，函数名后面没有一组将函数的参数括起来的括号。函数名本身指函数的 **Function** 对象；它并不引用函数返回的值。

尽管在某些情况下（如函数本身不断变化时），将 **Function** 对象用作循环条件可能会有所帮助，但通常这很可能是个错误。

更正此错误

- 使用函数调用语法，用括号将函数的参数括起来以便计算函数值。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1248: 应为“assembly”

该代码似乎要定义程序集的全局属性，但未使用程序集标识符。定义程序集属性的正确语法如下：

```
[assembly: attribute]
```

`attribute` 应是程序集有效的全局属性，它由 **System.Reflection** 命名空间提供。有关更多信息，请参见 System.Reflection 命名空间。

更正此错误

- 确保使用正确的语法声明全局属性。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1249: 程序集自定义属性可能不是另一构造的部分

程序集的自定义属性只能用在全局范围中。

更正此错误

- 确保程序集的自定义属性用在全局范围中。

请参见

概念

脚本疑难解答

变量和常数的范围

其他资源

JScript 参考

JS1250: Expando 方法不能为静态

某个方法同时具有 **expando** 和 **static** 修饰符。这是不允许的。

更正此错误

- 从方法声明中移除 **expando** 或 **static** 修饰符。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1251: 此方法与该类中的另一方法具有相同的名称和参数类型

在类中有多个具有相同名称和参数类型的方法。这是不允许的，因为无法在调用时区分它们。

更正此错误

1. 如果有重复的方法，请移除多余的方法。
2. 更改一个或多个方法的名称或参数类型。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

基于类的对象

JS1252: 应将用作构造函数的类成员标记为 expando 函数

new 运算符被应用于类成员。仅当类成员是用 expando 修饰符标记的方法或属性时才允许这样做，用 expando 修饰符标记使该类成员能够用作构造函数。

更正此错误

1. 将 **expando** 修饰符应用于类成员的定义。
2. 对于非 expando 类成员，不要使用 **new** 运算符。

请参见

参考

expando 修饰符

new 运算符

概念

脚本疑难解答

其他资源

JScript 参考

JS1253: 不是有效的版本字符串

不是有效的版本字符串。期望的格式为 major.minor[.build[.revision]]

该代码为程序集定义全局的 **AssemblyVersion** 属性，但是传递给该属性的版本字符串的格式不正确。版本字符串的格式必须为“主版本.次版本[.内部版本[.修订版]]”。

更正此错误

- 确保版本字符串的格式为“主版本.次版本[.内部版本[.修订版]]”。

请参见

参考

AssemblyVersionAttribute Class

概念

脚本疑难解答

其他资源

JScript 参考

JS1254: 可执行文件不能本地化，区域性应始终为空

该代码为程序集定义全局 **AssemblyCulture** 属性，由于可执行文件无法本地化，因此这是不允许的。

更正此错误

- 确保不为可执行文件指定 **AssemblyCulture** 属性。

请参见

参考

AssemblyCultureAttribute Class

概念

脚本疑难解答

其他资源

JScript 参考

JS1255: 加号运算符是较慢的字符串串联方法

加号运算符是较慢的字符串串联方法。请改用 System.Text.StringBuilder。

加号 (+) 运算符连接字符串。在许多情况下(如将许多小字符串追加到另一个字符串)，**System.Text.StringBuilder** 生成运行速度快得多的代码。

例如，考虑以下生成字符串"0123456789"的代码。它在编译时生成该警告。

```
var a : String = "";
for(var i=0; i<10; i++)
   a += i;
print(a);
```

它在运行时显示字符串"0123456789"。

当上述示例使用 **System.Text.StringBuilder** 时，程序的运行速度更快并且不生成警告。

```
var b : System.Text.StringBuilder;
b = new System.Text.StringBuilder();
for(var i=0; i<10; i++)
   b.Append(i);
print(b);
```

与前面的程序一样，它也显示"0123456789"。

防止出现警告的另一种方法是使用非类型化变量存放要向其追加其他字符串的字符串。

更正此错误

1. 对于要向其追加其他字符串的字符串类型，使用 **System.Text.StringBuilder**，并重写具有 **+=** 运算的语句，以改用 **Append** 方法。
2. 对于要向其追加其他字符串的字符串，使用非类型化变量。(该解决方案不会提高代码的运行速度，但它将不显示警告。)

请参见

参考

StringBuilder Class

概念

脚本疑难解答

其他资源

JScript 参考

JS1256: 条件编译指令和变量不能在调试器中使用

在调试 JScript 程序时，在命令窗口输入了条件编译指令或变量。条件编译指令和变量只能在程序的编译过程中使用；它们在编译完成后不可用。

更正此错误

- 确保不在命令窗口中输入条件编译指令和变量。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

编写、编译、调试 JScript 代码

JS1257: Expando 方法必须为公共的

具有 **expando** 修饰符的方法还有一个非 **public** 的可见性修饰符。这是不允许的。

更正此错误

- 从方法声明中移除 **expando** 或可见性修饰符。

请参见

概念

脚本疑难解答

JScript 修饰符

其他资源

JScript 参考

JS1258: 委托不应显式构造，只需使用方法名称

该代码从函数构造委托，这是不必要的。函数名本身引用委托。

更正此错误

- 使用不带括号的函数名来引用委托。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JScript 函数

JS1259: 引用的程序集依赖于另一个未被引用或无法找到的程序集

该程序用 **import** 语句隐式或用 **/reference** 选项显式导入一个依赖另一个程序集的程序集。而另一个程序集或者未被引用，或者不在指定位置，因此找不到该程序集。

出现该错误的一个可能原因是：在将一个程序集移到新位置时，未移动它所依赖的程序集。另一个原因是未引用其他程序集所依赖的程序集。

更正此错误

1. 确保程序所需的程序集存在。
2. 检查为每个所需的程序集指定的位置和名称是否正确。
3. 确保显式引用其他程序集需要但未被该程序导入的程序集。

请参见

参考

import 语句

/reference

概念

脚本疑难解答

其他资源

JScript 参考

JS1260: 此转换可能在运行时失败

该代码具有可能会在运行时失败的隐式类型转换。这表示原始数据类型中数据的一些值在目标转换类型中没有明显的相似类型。

如果使用允许转换损耗的显式类型转换，将会使代码更为可靠，并大大降低代码在运行时失败的可能性。

更正此错误

1. 确保所提供的数据与它要转换到的数据类型兼容。
2. 在将数据从一种类型转换为另一种类型时，请使用显式类型转换。

请参见

概念

脚本疑难解答

类型转换

其他资源

JScript 参考

JS1261：将字符串转换为数字或布尔值的过程很慢，并且可能在运行时失败

该代码具有可能会在运行时失败的隐式类型转换。这表示一些字符串值与数字或布尔值没有明显的相似类型。

如果使用允许转换损耗的显式类型转换，将会使代码更为可靠，并大大降低代码在运行时失败的可能性。

更正此错误

1. 确保所提供的字符串与它要转换到的数据类型兼容。
2. 在将数据从一种类型转换为另一种类型时，请使用显式类型转换。

请参见

概念

脚本疑难解答

类型转换

其他资源

JScript 参考

JS1262: 不是有效的 .resources 文件

该程序是用 **/resource** 选项编译的，但是所指定的资源文件的格式不正确。该文件可能已损坏，或者它可能不是资源文件。

更正此错误

- 确保用 **/resource** 选项指定的文件是有效的资源文件。

请参见

参考

/resource

概念

脚本疑难解答

其他资源

JScript 参考

JS1263: & 运算符只能用在参数列表中

该代码中的"and"符 (**&**) 用在了函数调用的外部。"&"符只应位于包含地址的变量名前，并且只应用于以下情况：通过引用将变量传递给接受引用参数的函数。

通过引用传递变量时，允许函数更改该变量的值。

注意
JScript 不允许使用引用参数来定义函数。JScript 提供了"&"符，它允许调用采用引用参数的外部对象。

更正此错误

- 确保 (**&**) 符位于对函数的调用中的变量名称的前面，并确保函数接受通过引用传递的参数。

请参见

概念

脚本疑难解答

其他资源

JScript 参考

JS1264: 指定的类型不符合 CLS

该程序包含 **CLSCompliantAttribute** 属性，而编译器检测到不符合公共语言规范 (CLS) 的数据类型定义。该错误的一些可能的原因是：

- 类型名不符合 CLS。符合 CLS 的名称不能以下划线 (_) 开头，不能包含美元符号 ($)，也不能与另一个公共成员的名称仅在大小写上不同。
- 枚举被定义为拥有一个不符合 CLS 类型的基础类型。例如，枚举可能基于基元类型 **uint**（它不符合 CLS），而不是基于相应的符合 CLS 的系统类型 **System.UInt32**。

更正此错误

1. 确保该数据类型名称不以下划线 (_) 开头，不包含货币符号 ($)，还要避免不同成员的名称只是大小写有区别。
2. 确保只将符合 CLS 的数据类型用作枚举的基础类型。

请参见

参考

CLSCompliantAttribute Class

概念

脚本疑难解答

编写符合 CLS 的代码

公共语言规范

其他资源

JScript 参考

JS1265: 此类成员不能标记为符合 CLS，因为该类没有标记为符合 CLS

该类包含用 **CLSCompliantAttribute** 属性标记的成员，但是类本身不是用 **CLSCompliantAttribute** 属性标记的。如果任何类成员标记为符合 CLS，则该类必须标记为符合 CLS。

更正此错误

- 确保将 **CLSCompliantAttribute** 属性应用于每个拥有用 **CLSCompliantAttribute** 属性标记的成员的类。

请参见

参考

CLSCompliantAttribute Class

概念

脚本疑难解答

编写符合 CLS 的代码

公共语言规范

其他资源

JScript 参考

JS1266: 此类型不能标记为符合 CLS，因为该程序集没有标记为符合 CLS

数据类型用 **CLSCompliantAttribute** 属性标记，但是包含该数据类型的程序集未用 **CLSCompliantAttribute** 属性标记。如果程序集包含任何标记为符合 CLS 的数据类型，则该程序集必须标记为符合 CLS。

更正此错误

- 如果程序集中的任何数据类型是用 **CLSCompliantAttribute** 属性标记的，请确保将 **CLSCompliantAttribute** 属性应用于该程序集。

请参见

参考

CLSCompliantAttribute Class

概念

脚本疑难解答

编写符合 CLS 的代码

公共语言规范

其他资源

JScript 参考

JS1267: 处理器无效

引用的程序集针对不同的处理器。

更正此错误

- 确保使用 **/platform:anycpu** 或者与生成程序集所用的相同 **/platform** 值生成引用的程序集。

请参见

参考

CLSCompliantAttribute Class

概念

脚本疑难解答

编写符合 CLS 的代码

公共语言规范

其他资源

JScript 参考

JS1268: 程序集密钥文件无效

找不到程序集密钥文件或包含无效数据。

更正此错误

- 确保程序集属性 **System.Reflection.AssemblyKeyFile** 指向包含有效密钥文件的文件。

请参见

参考

CLSCompliantAttribute Class

概念

脚本疑难解答

编写符合 CLS 的代码

公共语言规范

其他资源

JScript 参考

JS1269: 类型名称无效

完全限定类型名必须少于 1024 个字符。

更正此错误

- 确保使用包含少于 1024 个字符的完全限定类型名。

请参见

参考

CLSCompliantAttribute Class

概念

脚本疑难解答

编写符合 CLS 的代码

公共语言规范

其他资源

JScript 参考

JS2013: 无效的目标

目标类型无效。

更正此错误

- 确保用 **/target** 命令行选项指定“exe”、“library”或“winexe”。

请参见

参考

CLSCompliantAttribute Class

概念

脚本疑难解答

编写符合 CLS 的代码

公共语言规范

其他资源

JScript 参考

JS2039: 平台无效

平台类型无效。

更正此错误

- 确保为 **/platform** 命令行选项指定“x86”、“Itanium”、“x64”或“anycpu”。

请参见

参考

CLSCompliantAttribute Class

概念

脚本疑难解答

编写符合 CLS 的代码

公共语言规范

其他资源

JScript 参考

函数 (JScript)

这些函数内置在 JScript 返回值中，其他函数可以在后面的操作中使用它们。

本节内容

GetObject 函数

从文件中返回对自动化对象的引用。

ScriptEngine 函数

返回一个表示正在使用的脚本语言的字符串。

ScriptEngineBuildVersion 函数

返回正在使用的脚本引擎的内部版本号。

ScriptEngineMajorVersion 函数

返回正在使用的脚本引擎的主版本号。

ScriptEngineMinorVersion 函数

返回正在使用的脚本引擎的次版本号。

相关章节

JScript 参考

列出“JScript 语言参考”所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

.NET Framework 参考

列出指向特定主题的链接，这些主题解释 .NET Framework 类库和其他基本元素的语法和结构。

GetObject 函数 (JScript 8.0)

从文件中返回对自动化对象的引用。该函数有两种形式。

```
function GetObject(class : String)
function GetObject(pathname : String [, class : String])
```

参数

class

必选。*"appName.objectType"*形式的字符串，其中 *appName* 是提供该对象的应用程序的名称，*objectType* 是要创建的对象的类型或类。

pathname

必选。包含要检索对象的文件的完整路径和名称。如果省略 *pathname*，则必须提供 *class*。

备注

使用 **GetObject** 函数可以从文件中访问一个 Automation 对象。可以将由 **GetObject** 返回的对象赋给对象变量。例如：

```
var CADObject;
CADObject = GetObject("C:\\CAD\\SCHEMA.CAD");
```

在执行这段代码时，将启动与指定的 pathname 关联的应用程序，并且指定文件中的对象将被激活。如果 *pathname* 是零长度字符串 (""), 则 **GetObject** 返回指定类型的新对象实例。如果省略 *pathname* 参数，则 **GetObject** 返回指定类型的当前活动对象。如果指定类型的对象不存在，则发生错误。

某些应用程序使您得以激活文件的部分内容。若要实现此功能，请在文件名尾部加一个感叹号 (!)，然后在感叹号后加一个用来标识要激活文件部分的字符串。有关如何创建该字符串的信息，请参见创建对象的应用程序的相关文档。

例如，在绘图应用程序中，可将一幅图形的多个层存储在一个文件中。可以使用以下代码激活名为 `SCHEMA.CAD` 的图形中的某一层：

```
var LayerObject = GetObject("C:\\CAD\\SCHEMA.CAD!Layer3");
```

如果未指定对象的类，则 Automation 将根据所提供的文件名来确定要启动的应用程序和要激活的对象。但是某些文件可能支持多个类的对象。例如，一幅图形可能支持三种不同类型的对象：Application 对象、Drawing 对象和 Toolbar 对象，所有这些对象都是同一文件的组成部分。若要指定要在文件中激活的对象，请使用可选的 *class* 参数。例如：

```
var MyObject;
MyObject = GetObject("C:\\DRAWINGS\\SAMPLE.DRW", "FIGMENT.DRAWING");
```

在以上示例中，`FIGMENT` 是绘图应用程序的名称，而 `DRAWING` 是它支持的一种对象类型。对象激活后，在代码中使用已定义的对象变量引用此对象。在前面的示例中，可以使用对象变量 `MyObject` 访问新对象的属性和方法。例如：

```
MyObject.Line(9, 90);
MyObject.InsertText(9, 100, "Hello, world.");
MyObject.SaveAs("C:\\DRAWINGS\\SAMPLE.DRW");
```

注意

如果存在对象的当前实例，或者您希望使用已加载的文件创建对象，请使用 **GetObject** 函数。如果不存在当前实例，而且您也不希望使用已加载的文件来创建对象，则可以使用 **ActiveXObject** 对象。

如果对象本身已经注册为单实例对象，则无论执行多少次 **ActiveXObject**，也只创建一个对象实例。对于单实例对象，当使用零长度字符串 ("") 语法调用 **GetObject** 时，**GetObject** 始终返回同一实例，如果省略 *pathname* 参数，则会导致错误。

要求

版本 5

请参见

参考

ActiveXObject 对象

ScriptEngine 函数 (JScript 8.0)

返回一个表示正在使用的脚本语言的字符串。

```
function ScriptEngine() : String
```

备注

ScriptEngine 函数可以返回下列字符串：

字符串	说明
JScript	指示 Microsoft JScript 是当前的脚本引擎。
VBA	指示当前使用的脚本引擎是 Microsoft Visual Basic for Applications。
VBScript	指示 Microsoft Visual Basic Scripting Edition 是当前的脚本引擎。

示例

下面的示例阐释 **ScriptEngine** 函数的用法：

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

要求

版本 2

请参见

参考

ScriptEngineBuildVersion 函数 (JScript 8.0)
ScriptEngineMajorVersion 函数 (JScript 8.0)
ScriptEngineMinorVersion 函数 (JScript 8.0)

ScriptEngineBuildVersion 函数 (JScript 8.0)

返回正在使用的脚本引擎的内部版本号。

```
function ScriptEngineBuildVersion() : int
```

备注

返回值与包含在正在使用的脚本语言动态链接库 (DLL) 中的版本信息直接对应。

示例

下面的示例阐释 **ScriptEngineBuildVersion** 函数的用法：

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

要求

版本 2

请参见

参考

ScriptEngine 函数 (JScript 8.0)
ScriptEngineMajorVersion 函数 (JScript 8.0)
ScriptEngineMinorVersion 函数 (JScript 8.0)

ScriptEngineMajorVersion 函数 (JScript 8.0)

返回正在使用的脚本引擎的主版本号。

```
function ScriptEngineMajorVersion() : int
```

备注

返回值与包含在正在使用的脚本语言动态链接库 (DLL) 中的版本信息直接对应。

示例

下面的示例阐释 **ScriptEngineMajorVersion** 函数的用法：

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

要求

版本 2

请参见

参考

ScriptEngine 函数 (JScript 8.0)
ScriptEngineBuildVersion 函数 (JScript 8.0)
ScriptEngineMinorVersion 函数 (JScript 8.0)

ScriptEngineMinorVersion 函数 (JScript 8.0)

返回正在使用的脚本引擎的次版本号。

```
function ScriptEngineMinorVersion() : int
```

备注

返回值与包含在正在使用的脚本语言动态链接库 (DLL) 中的版本信息直接对应。

示例

下面的示例说明了 **ScriptEngineMinorVersion** 函数的用法。

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

要求

版本 2

请参见

参考

ScriptEngine 函数 (JScript 8.0)
ScriptEngineBuildVersion 函数 (JScript 8.0)
ScriptEngineMajorVersion 函数 (JScript 8.0)

标识符 (Literal)

标识符 (Literal) 是不变的程序元素，在 JScript 代码的上下文中具有特殊的意义。

本节内容

false 标识符

null 标识符

true 标识符

相关章节

JScript 参考

列出“JScript 语言参考”所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

.NET Framework 参考

列出指向特定主题的链接，这些主题解释 .NET Framework 类库和其他基本元素的语法和结构。

false 标识符

表示 false 的布尔值。

备注

布尔值为 **true** 或 **false**。与 **false** 相反的（即非 **false**）是 **true**。

要求

版本 1

请参见

参考

boolean 数据类型 (JScript)

Boolean 对象

true 标识符

其他资源

JScript 数据类型

null 标识符

表示"无对象"的对象。

备注

通过为变量赋 **null**值可以清除变量的内容(不删除变量)。

注意
在 JScript 中，**null** 与 0 不同(在 C 和 C++ 中与 0 相同)。此外，JScript 中的 **typeof** 运算符将空值报告为 **Object** 类型而不是 **null** 类型。保留了这种可能令人混淆的行为以便向后兼容。

要求

版本 1

请参见

参考

Object 对象

其他资源

JScript 数据类型

true 标识符

表示 true 的布尔值。

备注

布尔值为 **true** 或 **false**。与 **true** 相反的(即非 **true**)是 **false**。

要求

版本 1

请参见

参考

boolean 数据类型 (JScript)

Boolean 对象

false 标识符

其他资源

JScript 数据类型

Visual Basic 和 Visual C# 项目扩展性方法

方法是作为对象成员的函数。JScript 中的众多方法根据方法名称按字母进行分类。

本节内容

方法 (A-E)

方法 (F-I)

方法 (J-R)

方法 (S)

方法 (T-Z)

相关章节

JScript 参考

列出“JScript 语言参考”所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

对象

解释 JScript 中的对象的概念，说明对象与属性和方法的关系，并提供指向特定主题的链接，这些主题将详细介绍 JScript 所支持的对象。

.NET Framework 参考

列出指向特定主题的链接，这些主题解释 .NET Framework 类库和其他基本元素的语法和结构。

方法 (A-E)

方法是作为对象成员的函数。以下方法的名称以字母 a 到 e 开头。

本节内容

abs 方法

返回数字的绝对值。

acos 方法

返回数字的反余弦值。

anchor 方法

在对象中的指定文本两侧放置一个有 NAME 属性的 HTML 定位点。

apply 方法

返回某对象的一个方法，用另一个对象替换当前对象。

asin 方法

返回数字的反正弦值。

atan 方法

返回一个数的反正切值。

atan2 方法

返回从 X 轴到 (x,y) 点的角度（以弧度为单位）。

atEnd 方法

返回一个布尔值，通过该值指示枚举数是否位于集合的末尾。

big 方法

将 HTML <BIG> 标记放置到 **String** 对象中的文本两侧。

blink 方法

将 HTML <BLINK> 标记放置到 **String** 对象中的文本两侧。

bold 方法

将 HTML <B> 标记放置到 **String** 对象中的文本两侧。

call 方法

调用一个对象的方法，用另一个对象替换当前对象。

ceil 方法

返回大于或等于其数字参数的最小整数。

charAt 方法

返回指定索引位置处的字符。

charCodeAt 方法

返回指定字符的 Unicode 编码。

compile 方法

将正则表达式编译为内部格式。

concat 方法（数组）

返回由两个数组组合而成的新数组。

concat 方法（字符串）

返回由所提供的两个字符串串联而成的 **String** 对象。

cos 方法

返回数的余弦值。

decodeURI 方法

返回一个已编码的统一资源标识符 (URI) 的非编码形式。

decodeURIComponent 方法

返回统一资源标识符 (URI) 的一个已编码组件的非编码形式。

dimensions 方法

返回 VBArray 的维数。

encodeURI 方法

将文本字符串编码为有效的统一资源标识符 (URI)。

encodeURIComponent 方法

将文本字符串编码为统一资源标识符 (URI) 的一个有效组件。

escape 方法

对 **String** 对象编码以便它们能在所有计算机上可读。

eval 方法

计算 JScript 代码并执行。

exec 方法

在指定字符串中执行对匹配项的搜索。

exp 方法

返回 *e*（自然对数的底）的若干次幂。

相关章节

JScript 参考

列出“JScript 语言参考”所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

方法

按字母分类列出 JScript 中的可用方法，并列出指向每个方法类别的链接。

对象

解释有关 JScript 中的对象的概念，并说明对象与属性和方法的关系，并提供指向特定主题的链接，这些主题将详细介绍 JScript 所支持的对象。

abs 方法

返回数字的绝对值。

```
function abs( number : Number ) : Number
```

参数
number

必选。数值表达式。

备注

返回的值是 *number* 参数的绝对值。

示例

下面的示例阐释了 **abs** 方法的用法。

```
function ComparePosNegVal(n) {
   var s = "";
   var v1 = Math.abs(n);
   var v2 = Math.abs(-n);
   if (v1 == v2) {
      s = "The absolute values of " + n + " and "
      s += -n + " are identical.";
   }
   return(s);
}
```

要求

版本 1

应用于:

Math 对象

请参见
其他资源
Visual Basic 和 Visual C# 项目扩展性方法

acos 方法

返回数字的反余弦值。

```
function acos( number : Number ) : Number
```

参数
number

必选。数值表达式。

备注

返回值介于 0 和 *pi* 之间，表示 *number* 参数的反余弦值。如果 *number* 小于 -1 或 *number* 大于 +1，则返回值为 **NaN**。

要求

版本 1

应用于：

Math 对象

请参见
参考
asin 方法
atan 方法
cos 方法
sin 方法
tan 方法
PI 属性

anchor 方法

返回对象中指定文本周围带有 HTML 定位点（具有 NAME 属性）的字符串。

```
function anchor(anchorString : String ) : String
```

参数

anchorString

必选。要放在 HTML 定位点 NAME 属性中的文本。

备注

调用 **anchor** 方法在 **String** 对象外创建一个命名定位点。

未进行任何检查来查看此标记是否已应用于该字符串。

示例

下面的示例说明了 **anchor** 方法是如何实现此功能的：

```
var strVariable = "This is an anchor";
strVariable = strVariable.anchor("Anchor1");
```

最后一个语句后面的 `strVariable` 的值为：

```
<A NAME="Anchor1">This is an anchor</A>
```

要求

版本 1

应用于：

String 对象

请参见

参考

link 方法

JScript 8.0

apply 方法

返回某对象的一个方法，用另一个对象替换当前对象。

```
function apply([thisObj : Object [,argArray : { Array | arguments }]])
```

参数

thisObj

可选项。将作为当前对象使用的对象。

argArray

可选项。要传递给函数的参数数组或 **arguments** 对象。

备注

如果 *argArray* 既不是有效数组也不是 **arguments** 对象，则将导致 TypeError。

如果既未提供 *argArray* 参数也未提供 *thisObj* 参数，则 **global** 对象将被用作 *thisObj* 且不被传递任何参数。

要求

版本 5.5

应用于：

Function 对象

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

asin 方法

返回数字的反正弦值。

```
function asin(number : Number) : Number
```

参数
number

必选。数值表达式。

备注

返回值介于 *-pi/2* 和 *pi/2* 之间，表示 *number* 参数的反正弦值。如果 *number* 小于 -1 或 *number* 大于 +1，则返回值为 **NaN**。

要求

版本 1

应用于：

Math 对象

请参见
参考
acos 方法
atan 方法
cos 方法
sin 方法
tan 方法
PI 属性

atan 方法

返回一个数的反正切值。

```
function atan(number : Number ) Number
```

参数
number

必选。数值表达式。

备注

返回值介于 *-pi/2* 和 *pi/2* 之间，表示 *number* 参数的反正切值。

要求

版本 1

应用于：

Math 对象

请参见
参考
acos 方法
asin 方法
atan2 方法
cos 方法
sin 方法
tan 方法
PI 属性

atan2 方法

返回从 X 轴到 (x,y) 点的角度(以弧度为单位)。

```
function atan2(y : Number , x : Number ) : Number
```

参数

x

必选。一个数值表达式，表示点的笛卡尔 x 坐标。

y

必选。一个数值表达式，表示点的笛卡尔 y 坐标。

备注

返回值在 *-pi* 和 *pi* 之间，表示提供的 (*x*,*y*) 点的角度。

要求

版本 1

应用于:

Math 对象

请参见

参考

atan 方法
tan 方法
PI 属性

atEnd 方法

返回一个布尔值，通过该值指示枚举数是否位于集合的末尾。

```
function atEnd() : Boolean
```

备注

如果当前项是集合中的最后一个项，或者集合为空，或者当前项未定义，则 **atEnd** 方法返回 **true**。否则，返回 **false**。

示例

在下面的代码中，使用了 **atEnd** 方法来确定是否到达了一个驱动器列表的末尾：

```
function ShowDriveList(){
   var fso, s, n, e, x;
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);
   s = "";
   for (; !e.atEnd(); e.moveNext())
   {
      x = e.item();
      s = s + x.DriveLetter;
      s += " - ";
      if (x.DriveType == 3)
         n = x.ShareName;
      else if (x.IsReady)
         n = x.VolumeName;
      else
         n = "[Drive not ready]";
      s +=   n + "<br>";
   }
   return(s);
}
```

要求

版本 2

应用于：

Enumerator 对象

请参见

参考

item 方法 (JScript)
moveFirst 方法
moveNext 方法

big 方法

返回 **String** 对象中文本周围带有 HTML <BIG> 标记的字符串。

```
function big() : String
```

备注

未进行任何检查来查看此标记是否已应用于该字符串。

示例

下面的示例说明了 **big** 方法是如何工作的:

```
var strVariable = "This is a string object";
strVariable = strVariable.big();
```

最后一个语句后面的 `strVariable` 的值为:

```
<BIG>This is a string object</BIG>
```

要求

版本 1

应用于:

String 对象

请参见

参考

small 方法

blink 方法

返回 **String** 对象中文本周围带有 HTML <BLINK> 标记的字符串。

```
function blink() : String
```

备注

未进行任何检查来查看此标记是否已应用于该字符串。

在 Microsoft Internet Explorer 中不支持 <BLINK> 标记。

示例

下面的示例说明了 **blink** 方法是如何工作的：

```
var strVariable = "This is a string object";
strVariable = strVariable.blink();
```

最后一个语句后面的 `strVariable` 的值为：

```
<BLINK>This is a string object</BLINK>
```

要求

版本 1

应用于：

String 对象

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

bold 方法

返回 **String** 对象中文本周围带有 HTML <B> 标记的字符串。

```
function bold() : String
```

备注

未进行任何检查来查看此标记是否已应用于该字符串。

示例

下面的示例说明了 **bold** 方法是如何工作的:

```
var strVariable = "This is a string object";
strVariable = strVariable.bold();
```

最后一个语句后面的 `strVariable` 的值为:

```
<B>This is a string object</B>
```

要求

版本 1

应用于:

String 对象

请参见

参考

italics 方法

call 方法

调用一个对象的方法，用另一个对象替换当前对象。

```
function call([thisObj : Object [, arg1[, arg2[, ... [, argN]]]]])
```

参数

thisObj

可选项。将作为当前对象使用的对象。

arg1, arg2, ..., argN

可选项。将被传递到该方法的参数列表。

备注

call 方法可以用来代表另一个对象调用一个方法。**call** 方法使您得以将一个函数的对象上下文从初始的上下文改变为由 *thisObj* 指定的新对象。

如果没有提供 *thisObj* 参数，则 **global** 对象被用作 *thisObj*。

要求

版本 5.5

应用于：

Function 对象

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

ceil 方法

返回大于或等于其数字参数的最小整数。

```
function ceil(number : Number ) : Number
```

参数
number

必选。数值表达式。

备注

返回值为大于或等于其数字参数的最小整数值。

要求

版本 1

应用于:

Math 对象

请参见
参考
floor 方法

charAt 方法

返回 **String** 对象的指定索引处的字符。

```
function charAt(index : Number) : String
```

参数
index

必选。所需字符的从零开始的索引。有效值为 0 到字符串长度减 1 的数字。

备注

charAt 方法返回一个字符值，该字符值等于指定索引位置的字符。一个字符串中的第一个字符位于索引位置 0，第二个字符位于索引位置 1，依此类推。超出有效范围的 *index* 返回空字符串。

示例

下面的示例阐释了 **charAt** 方法的用法：

```
function charAtTest(n){
   var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; // Initialize variable.
   var s;                                  // Declare variable.
   s = str.charAt(n - 1);                  // Get correct character
                                           // from position n – 1.
   return(s);                              // Return character.
}
```

要求

版本 1

应用于：

String 对象

请参见
其他资源
Visual Basic 和 Visual C# 项目扩展性方法

charCodeAt 方法

返回一个整数，该整数表示 **String** 对象中指定位置处的字符的 Unicode 编码。

```
function charCodeAt(index : Number) : String
```

参数
index

必选。所需字符的从零开始的索引。有效值为 0 到字符串长度减 1 的数字。

备注

一个字符串中的第一个字符位于索引位置 0，第二个字符位于索引位置 1，依此类推。

如果指定 *index* 没有字符，将返回 **NaN**。

示例

下面的示例阐释了 **charCodeAt** 方法的用法。

```
function charCodeAtTest(n){
  var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; //Initialize variable.
  var n;                                  //Declare variable.
  n = str.charCodeAt(n - 1);              //Get the Unicode value of the
                                          // character at position n.
  return(n);                              //Return the value.
}
```

要求

版本 5.5

应用于:

String 对象

请参见
参考
fromCharCode 方法

compile 方法 (JScript)

将正则表达式编译为内部格式，从而执行得更快。

```
function compile(pattern : String [, flags : String] )
```

参数
pattern

必选。一个字符串表达式，包含要编译的正则表达式模式

flags

可选项。可以组合使用的可用标志有：

- g（全局搜索出现的所有 pattern）
- i（忽略大小写）
- m（多行搜索）

备注

compile 方法将 *pattern* 转换为内部的格式，从而执行得更快。例如，这允许在循环中更有效地使用正则表达式。当重复使用相同的表达式时，编译过的正则表达式使执行加速。然而，如果正则表达式发生更改，则这种编译毫无益处。

示例

下面的示例阐释了 **compile** 方法的用法：

```
function CompileDemo(){
   var rs;
   var s = "AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPp"
   // Create regular expression for uppercase only.
   var r = new RegExp("[A-Z]", "g");
   var a1 = s.match(r)              // Find matches.
   // Compile the regular expression for lowercase only.
   r.compile("[a-z]", "g");
   var a2 = s.match(r)              // Find matches.
   return(a1 + "\n" + a2);
}
```

要求

版本 3

应用于：

正则表达式对象

请参见
概念
正则表达式语法

concat 方法(数组)

返回由当前数组和任何其他项组合而成的新数组。

```
function concat([item1 : { Object | Array } [, ... [, itemN : { Object | Array }]]]) : Arr
ay
```

参数
item1, item2, ..., itemN

可选项。要添加到当前数组末尾处的其他项。

备注

concat 方法返回一个 **Array** 对象，它包含当前数组和提供的任何其他项的连接。

要添加的项 (*item1 ... itemN*) 会按照从左到右的顺序添加到数组。如果某一项为数组，则其内容将添加到当前数组的末尾。如果该项不是数组，则将其作为单个数组元素添加到数组末尾。

源数组元素按以下规则复制到结果数组：

- 对于从连接到新数组的任何数组中复制的对象引用，复制后仍然指向相同的对象。不论新数组和原始数组中哪一个有改变，都将引起对方的改变。
- 对于连接到新数组的数值或字符串，只复制其值。一个数组中值的改变并不影响另一个数组中的值。

示例

下面的示例阐释了使用数组时 **concat** 方法的用法：

```
function ConcatArrayDemo(){
   var a, b, c, d;
   a = new Array(1,2,3);
   b = "JScript";
   c = new Array(42, "VBScript");
   d = a.concat(b, c);
   //Returns the array [1, 2, 3, "JScript", 42, "VBScript"]
   return(d);
}
```

要求

版本 3

应用于：

Array 对象

请参见
参考
concat 方法(字符串)
join 方法
String 对象

concat 方法(字符串)

返回一个字符串值，该值包含当前字符串与提供的任何字符串的连接。

```
function concat([string1 : String [, ... [, stringN : String]]]]) : String
```

参数

string1, ..., stringN

可选。要连接到当前字符串末尾的 **String** 对象或文本。

备注

concat 方法的结果等同于：*result* = *curstring* + *string1* + ...+ *stringN*。*curstring* 是指对象中存储的用于提供 **concat** 方法的字符串。源字符串中或结果字符串中的值的变化都不会影响另一个字符串中的值。如果有不是字符串的参数，则它们在连接到 *curstring* 之前将首先被转换为字符串。

示例

下面的示例阐释了 **concat** 方法用于字符串时的用法：

```
function concatDemo(){
   var str1 = "ABCDEFGHIJKLM"
   var str2 = "NOPQRSTUVWXYZ";
   var s = str1.concat(str2);
   // Return concatenated string.
   return(s);
}
```

要求

版本 3

应用于：

String 对象

请参见

参考

加法运算符 (+)
Array 对象
concat 方法(数组)

JScript 8.0

cos 方法

返回数的余弦值。

```
function cos(number : Number) : Number
```

参数
number

必选。数值表达式。

备注

返回值为其数字参数的余弦值。

要求

版本 1

应用于:

Math 对象

请参见
参考
acos 方法
asin 方法
atan 方法
sin 方法
tan 方法

JScript 8.0

decodeURI 方法

返回一个已编码的统一资源标识符 (URI) 的非编码形式。

```
function decodeURI(URIstring : String) : String
```

参数
URIstring

必选。表示编码 URI 的字符串。

备注

使用 **decodeURI** 方法代替已经过时的 **unescape** 方法。

decodeURI 方法返回一个字符串值。

如果 *URIString* 无效，将发生 URIError。

要求

版本 5.5

应用于:

Global 对象

请参见
参考
decodeURIComponent 方法
encodeURI 方法

JScript 8.0

decodeURIComponent 方法

返回统一资源标识符 (URI) 的一个已编码组件的非编码形式。

```
function decodeURIComponent(encodedURIString : String) : String
```

备注

必选的 *encodedURIString* 参数是一个表示已编码的 URI 组件的值。

备注

URIComponent 是一个完整的 URI 的一部分。

如果 *encodedURIString* 无效，则将产生 URIError。

要求

版本 5.5

应用于:

Global 对象

请参见

参考

decodeURI 方法

encodeURI 方法

dimensions 方法

返回 VBArray 的维数。

```
function dimensions() : Number
```

备注

dimensions 方法提供了一个检索指定 VBArray 的维数的方法。

下面的示例由三部分组成。第一部分是创建 Visual Basic 安全数组的 VBScript 代码。第二部分是 JScript 代码，该代码确定安全数组中的维度数和每一维的上限。这两部分都位于 HTML 页的 <HEAD> 区域。第三部分是位于 <BODY> 区域内用于运行其他两个部分的 JScript 代码。

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(j, i) = k
         k = k + 1
      Next
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vba)
{
   var i, s;
   var a = new VBArray(vba);
   for (i = 1; i <= a.dimensions(); i++)
   {
      s = "The upper bound of dimension ";
      s += i + " is ";
      s += a.ubound(i)+ ".<BR>";
   }
   return(s);
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT language="jscript">
   document.write(VBArrayTest(CreateVBArray()));
</SCRIPT>
</BODY>
```

要求

版本 3

应用于:

VBArray 对象

请参见

参考

getItem 方法

lbound 方法

toArray 方法

ubound 方法

encodeURI 方法

返回编码为有效的唯一资源标识符 (URI) 的字符串。

```
function encodeURI(URIString : String) : String
```

参数
URIString

必选。表示编码 URI 的字符串。

备注

encodeURI 方法返回一个已编码的 URI。如果将编码结果传递给 **decodeURI**，则将返回初始的字符串。**encodeURI** 不对下列字符进行编码：":"、"/"、";"和"?"。请使用 **encodeURIComponent** 对这些字符进行编码。

要求

版本 5.5

应用于：

Global 对象

请参见
参考
decodeURI 方法
decodeURIComponent 方法

encodeURIComponent 方法

返回编码为统一资源标识符 (URI) 的有效组件的字符串。

```
function encodeURIComponent(encodedURIString : String) : String
```

参数
encodedURIString

必选。表示编码 URI 组件的字符串。

备注

encodeURIComponent 方法返回一个已编码的 URI。如果将编码结果传递给 **decodeURIComponent**，则将返回初始的字符串。因为 **encodeURIComponent** 方法编码所有字符，所以请注意该字符串是否表示路径，例如 */folder1/folder2/default.html*。斜杠字符将被编码，因此如果作为请求发送到 Web 服务器将无效。如果字符串中包含多个 URI 组件，请使用 **encodeURI** 方法进行编码。

要求

版本 5.5

应用于：

Global 对象

请参见
参考
decodeURI 方法
decodeURIComponent 方法

escape 方法

返回一个可在所有计算机上读取的编码 **String** 对象。

```
function escape(charString : String) : String
```

参数
charString

必选。要编码的任何 **String** 对象或文本。

备注

escape 方法返回一个包含 *charstring* 内容的字符串值(Unicode 格式)。所有空格、标点、重音符号以及任何其他非 ASCII 字符都用 **%***xx* 编码替换，其中 *xx* 等于表示该字符的十六进制数。例如，空格返回为"%20"。

字符值大于 255 的字符以 **%u***xxxx* 格式存储。

注意

escape 方法不能用来对唯一资源标识符 (URI) 进行编码。请改用 **encodeURI** 和 **encodeURIComponent** 方法。

要求

版本 1

应用于:

Global 对象

请参见
参考
encodeURI 方法
encodeURIComponent 方法
String 对象
unescape 方法

eval 方法 (JScript)

计算 JScript 代码并执行。

```
function eval(codeString : String [, override : String])
```

参数
codeString

必选。包含有效 JScript 代码的字符串。

override

可选项。确定要应用于 *codeString* 中代码的安全权限的字符串。

备注

eval 函数允许动态执行 JScript 源代码。

传递给 **eval** 方法的代码执行时所在的上下文和调用 **eval** 方法时的上下文一样。请注意，在 **eval** 语句中定义的新变量或类型对于封闭程序是不可见的。

除非将字符串"unsafe"传递为第二个参数，否则，传递至 **eval** 方法的代码在受限安全上下文中执行。受限安全上下文禁止访问系统资源，如文件系统、网络或用户界面。如果代码试图访问这些资源，则会产生安全异常。

当 eval 的第二个参数为字符串"unsafe"时，传递给 **eval** 方法的代码在调用代码所在的安全上下文中执行。第二个参数是区分大小写的，因此，字符串"Unsafe"或"UnSAfE"不会重写受限安全上下文。

安全注意
以非安全模式使用 **eval** 只能执行从可以信任的源获得的代码字符串。

示例

例如，下面的代码将变量 `mydate` 初始化为测试日期或当前日期，这取决于变量 `doTest` 的值：

```
var doTest : boolean = true;
var dateFn : String;
if(doTest)
   dateFn = "Date(1971,3,8)";
else
   dateFn = "Date()";

var mydate : Date;
eval("mydate = new "+dateFn+";");
print(mydate);
```

该程序的输出为：

```
Thu Apr 8 00:00:00 PDT 1971
```

要求

版本 1

应用于：

Global 对象

请参见
参考
String 对象

JScript 8.0

exec 方法

使用正则表达式模式对字符串执行搜索，并返回一个包含该搜索结果的数组。

```
function exec(str : String) : Array
```

参数

str

必选。执行搜索的 **String** 对象或字符串文本。

备注

如果 **exec** 方法没有找到匹配，将返回 **null**。如果找到匹配，则 **exec** 方法返回一个数组，并将更新全局 **RegExp** 对象的属性以反映匹配结果。数组元素 0 包含了完整的匹配，而元素 1 到 *n* 包含的是匹配中出现的任意一个子匹配。这相当于没有设置全局标志 (**g**) 的 **match** 方法的行为。

如果为正则表达式设置了全局标志，则 **exec** 从 **lastIndex** 值指示的位置开始搜索字符串。如果没有设置全局标志，则 **exec** 忽略 **lastIndex** 的值，从字符串的起始位置开始搜索。

exec 方法返回的数组有三个属性：**input**、**index** 和 **lastIndex.**。**Input** 属性包含整个被搜索的字符串。**Index** 属性包含了在整个被搜索字符串中匹配的子字符串的位置。**lastIndex** 属性中包含了匹配中最后一个字符的下一个位置。

示例

下面的示例阐释了 **exec** 方法的用法：

```
function RegExpTest() {
   var s = "";
   var src = "The rain in Spain falls mainly in the plain.";
   // Create regular expression pattern for matching a word.
   var re = /\w+/g;
   var arr;
   // Loop over all the regular expression matches in the string.
   while ((arr = re.exec(src)) != null)
      s += arr.index + "-" + arr.lastIndex + "\t" + arr + "\n";
   return s;
}
```

要求

版本 3

应用于：

正则表达式对象

请参见

参考

match 方法
RegExp 对象
search 方法
test 方法

概念

正则表达式语法

exp 方法

返回 e(自然对数的底)的若干次幂。

```
function exp(number : Number) : Number
```

参数
number

必选。数值表达式。

备注

返回值为自然对数。常数 *e* 是自然对数的底，它约等于 2.178，*number* 是提供的参数。

要求

版本 1

应用于:

Math 对象

请参见
参考
E 属性

方法 (F-I)

方法是作为对象成员的函数。以下方法的名称以字母 f 到 i 开头。

本节内容

fixed 方法

将 HTML <TT> 标记放置到 **String** 对象中的文本两侧。

floor 方法

返回小于或等于其数值参数的最大整数。

fontcolor 方法

将带有 COLOR 属性的 HTML <FONT> 标记放置到 **String** 对象中的文本两侧。

fontsize 方法

将带有 SIZE 属性的 HTML <FONT> 标记放置到 **String** 对象中的文本两侧。

fromCharCode 方法

从一些 Unicode 字符值中返回一个字符串。

getDate 方法

使用当地时间返回 **Date** 对象中一个月第几天的值。

getDay 方法

使用当地时间返回 **Date** 对象中该天的星期值。

getFullYear 方法

使用当地时间返回 **Date** 对象中的年份值。

getHours 方法

使用当地时间返回 **Date** 对象中的小时值。

getItem 方法

将位于指定位置的项返回。

getMilliseconds 方法

使用当地时间返回 **Date** 对象中的毫秒值。

getMinutes 方法

使用当地时间返回存储在 **Date** 对象中的分钟值。

getMonth 方法

使用当地时间返回 **Date** 对象中的月份值。

getSeconds 方法

使用当地时间返回存储在 **Date** 对象中的秒钟值。

getTime 方法

返回 **Date** 对象中的时间值。

getTimezoneOffset 方法

返回主机的时间与协调通用时间 (UTC) 之间的分钟差值。

getUTCDate 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的日期值。

getUTCDay 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中该天的星期值。

getUTCFullYear 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的年份值。

getUTCHours 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的小时值。

getUTCMilliseconds 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的毫秒值。

getUTCMinutes 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的分钟值。

getUTCMonth 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的月份值。

getUTCSeconds 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的秒钟值。

getVarDate 方法

将 **Date** 对象中的 VT_DATE 值返回。

getYear 方法

将 **Date** 对象中的年份值返回。(此方法已过时;请改用 **getFullYear** 方法。)

hasOwnProperty 方法

返回一个布尔值,该值指示一个对象是否具有指定名称的属性。

indexOf 方法

返回 **String** 对象内第一次出现子字符串的字符位置。

isFinite 方法

返回一个布尔值,该值指示所提供数字是否是有限值。

isNaN 方法

返回一个布尔值,该值指示某值是否为保留值 **NaN**(非数字)。

isPrototypeOf 方法

返回一个布尔值,该值指示对象是否存在于另一个对象的原型链中。

italics 方法

将 HTML <I> 标记放置到 **String** 对象中的文本两侧。

item 方法

返回集合中的当前项。

相关章节

JScript 参考

列出"JScript 语言参考"所包含的元素,并提供指向特定主题的链接,这些主题解释正确使用语言元素的详细背景信息。

方法

按字母分类列出 JScript 中的可用方法,并列出指向每个方法类别的链接。

对象

解释有关 JScript 中的对象的概念,并说明对象与属性和方法的关系,并提供指向特定主题的链接,这些主题将详细介绍 JScript 所支持的对象。

JScript 8.0

fixed 方法

返回 **String** 对象中文本周围带有 HTML <TT> 标记的字符串。

```
function fixed() : String
```

备注

未进行任何检查来查看此标记是否已应用于该字符串。

示例

下面的示例说明了 **fixed** 方法的工作方式：

```
var strVariable = "This is a string object";
strVariable = strVariable.fixed();
```

最后一个语句后面的 `strVariable` 的值为：

```
<TT>This is a string object</TT>
```

要求

版本 1

应用于：

String 对象

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

floor 方法

返回小于或等于其数值参数的最大整数。

```
function floor(number : Number) : Number
```

参数
number

必选。数值表达式。

备注

返回值为小于或等于其数值参数的最大整数值。

要求

版本 1

应用于:

Math 对象

请参见
参考
ceil 方法

fontcolor 方法

返回 **String** 对象中文本周围带有 HTML <FONT> 标记(具有 COLOR 属性)的字符串。

```
function fontcolor(colorVal : String) : String
```

参数

colorVal

必选。包含颜色值的字符串值。可以是颜色的十六进制值，也可以是颜色的预定义名。

备注

有效的预定义颜色名取决于 JScript 主机(浏览器、服务器等)。它们也可能随主机版本的不同而不同。更多信息请查阅主机文档。

未进行任何检查来查看此标记是否已应用于该字符串。

示例

下面的示例说明了 **fontcolor** 方法:

```
var strVariable = "This is a string";
strVariable = strVariable.fontcolor("red");
```

最后一个语句后面的 `strVariable` 的值为:

```
<FONT COLOR="RED">This is a string</FONT>
```

要求

版本 1

应用于:

String 对象

请参见

参考

fontsize 方法

fontsize 方法

返回 **String** 对象中文本周围带有 HTML <FONT> 标记(具有 SIZE 属性)的字符串。

```
function fontsize(intSize : Number) : String
```

参数
intSize

必选。用来指定文本大小的整数值。

备注

有效的整数值取决于 Microsoft JScript 主机。有关更多信息，请参见主机文档。

未进行任何检查来查看此标记是否已应用于该字符串。

示例

下面的示例说明了 **fontsize** 方法的用法：

```
var strVariable = "This is a string";
strVariable = strVariable.fontsize(-1);
```

最后一个语句后面的 `strVariable` 的值为：

```
<FONT SIZE="-1">This is a string</FONT>
```

要求

版本 1

应用于：

String 对象

请参见
参考
fontcolor 方法

fromCharCode 方法

从一些 Unicode 字符值中返回一个字符串。

```
function fromCharCode([code1 : Number [, ... [, codeN : Number]]]]) : String
```

参数

code1, ... , *codeN*

可选项。要转换为字符串的 Unicode 字符值序列。如果没有给出参数，结果为空字符串。

备注

fromCharCode 方法是从全局 String 对象中调用的。

示例

在下面的示例中，`test` 赋给“plain”字符串：

```
var test = String.fromCharCode(112, 108, 97, 105, 110);
```

要求

版本 3

应用于：

String 对象

请参见

参考

charCodeAt 方法

getDate 方法

使用当地时间返回 **Date** 对象中一个月第几天的值。

```
function getDate() : Number
```

备注

若要使用协调通用时间 (UTC) 获取日期值，请使用 **getUTCDate** 方法。

返回值是一个介于 1 到 31 之间的整数，该整数表示 **Date** 对象中的日期值。

示例

下面的示例阐释了 **getDate** 方法的用法。

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getYear();
   return(s);
}
```

要求

版本 1

应用于:

Date 对象

请参见

参考

getUTCDate 方法
setDate 方法
setUTCDate 方法

getDay 方法

使用当地时间返回 **Date** 对象中该天的星期值。

```
function getDay() : Number
```

备注

若要使用协调通用时间 (UTC) 获取星期数，请使用 **getUTCDay** 方法。

getDay 方法所返回的值是一个介于 0 到 6 之间的整数，该整数表示了星期数，返回值与一周中的星期数的对应关系如下：

值	**星期数**
0	星期日
1	星期一
2	星期二
3	星期三
4	星期四
5	星期五
6	星期六

下面的示例阐释了 **getDay** 方法的用法。

```
function DateDemo(){
   var d, day, x, s = "Today is: ";
   var x = new Array("Sunday", "Monday", "Tuesday");
   var x = x.concat("Wednesday","Thursday", "Friday");
   var x = x.concat("Saturday");
   d = new Date();
   day = d.getDay();
   return(s += x[day]);
}
```

要求

版本 1

应用于：

Date 对象

请参见

参考

getUTCDay 方法

getFullYear 方法

使用当地时间返回 **Date** 对象中的年份值。

```
function .getFullYear() : Number
```

备注

若要使用协调通用时间 (UTC) 获取年份值，请使用 **getUTCFullYear** 方法。

getFullYear 方法以绝对数的形式返回年份值。例如，1976 年的返回值为 1976。这样可以避免出现 2000 年问题，即不会把始于 2000 年 1 月 1 日的日期与始于 1900 年 1 月 1 日的日期混淆。

下面的示例阐释了 **getFullYear** 方法的用法。

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getFullYear();
   return(s);
}
```

要求

版本 3

应用于:

Date 对象

请参见

参考

getUTCFullYear 方法
setFullYear 方法
setUTCFullYear 方法

getHours 方法

使用当地时间返回 **Date** 对象中的小时值。

```
function getHours() : Number
```

备注

若要使用协调通用时间 (UTC) 获取小时值，请使用 **getUTCHours** 方法。

getHours 方法返回一个介于 0 到 23 之间的整数，该整数表示从午夜开始计算的小时数。在两种情况下会出现零：1:00:00 am 之前的时间，或创建 **Date** 对象时未将时间存储于其中。而要确定究竟是哪种情况，唯一的办法就是检查分钟和秒值是否也为零值。若它们均为零，则几乎可以肯定未将时间存储在 **Date** 对象中。

下面的示例阐释了 **getHours** 方法的用法。

```
function TimeDemo(){
   var d, s = "The current local time is: ";
   var c = ":";
   d = new Date();
   s += d.getHours() + c;
   s += d.getMinutes() + c;
   s += d.getSeconds() + c;
   s += d.getMilliseconds();
   return(s);
}
```

要求

版本 1

应用于：

Date 对象

请参见

参考

getUTCHours 方法
setHours 方法
setUTCHours 方法

getItem 方法

返回 VBArray 对象中指定位置处的项。

```
function getItem(dimension1 : Number [, ...], dimensionN : Number) : Object
```

参数

dimension1, ..., dimensionN

指定 VBArray 的所需元素的准确位置。参数的数量必须与 VBArray 中的维数相匹配。

示例

下面的示例由三部分组成。第一部分是创建 Visual Basic 安全数组的 VBScript 代码。第二部分是迭代 Visual Basic 安全数组并输出每个元素内容的 JScript 代码。这两部分都位于 HTML 页的 <HEAD> 区域。第三部分是位于 <BODY> 区域内用于运行其他两个部分的 JScript 代码。

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(i, j) = k
         document.writeln(k)
         k = k + 1
      Next
      document.writeln("<BR>")
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>
<SCRIPT LANGUAGE="JScript">
<!--
function GetItemTest(vbarray)
{
   var i, j;
   var a = new VBArray(vbarray);
   for (i = 0; i <= 2; i++)
   {
      for (j =0; j <= 2; j++)
      {
         document.writeln(a.getItem(i, j));
      }
   }
}-->
</SCRIPT>
</HEAD>
&ltBODY>
<SCRIPT LANGUAGE="JScript">
<!--
   GetItemTest(CreateVBArray());
-->
</SCRIPT>
</BODY>
```

要求

版本 1

应用于：

VBArray 对象

请参见

参考

dimensions 方法
lbound 方法
toArray 方法
ubound 方法

getMilliseconds 方法

使用当地时间返回 **Date** 对象中的毫秒值。

```
function getMilliseconds() : Number
```

备注

若要获取用协调通用时间 (UTC) 表示的毫秒数，请使用 **getUTCMilliseconds** 方法。

返回的毫秒值的范围是 0-999。

示例

下面的示例阐释了 getMilliseconds 方法的用法。

```
function TimeDemo(){
   var d, s = "The current local time is: ";
   var c = ":";
   d = new Date();
   s += d.getHours() + c;
   s += d.getMinutes() + c;
   s += d.getSeconds() + c;
   s += d.getMilliseconds();
   return(s);
}
```

要求

版本 3

应用于:

Date 对象

请参见

参考

getUTCMilliseconds 方法
setMilliseconds 方法
ssetUTCMilliseconds 方法

getMinutes 方法

使用当地时间返回 **Date** 对象中的分钟值。

```
function getMinutes() : Number
```

备注

若要使用协调通用时间 (UTC) 获取分钟值，请使用 **getUTCMinutes** 方法。

getMinutes 方法返回一个介于 0 到 59 之间的整数，该整数等于存储在 **Date** 对象中的分钟值。两种情况下会返回零：当小时后的时间不足一分钟时，或当创建 **Date** 对象时未将时间存储在该对象中。而要确定究竟属于哪种情况，唯一的方法就是检查小时和秒值是否也为零值。若它们均为零，则几乎可以肯定未将时间存储在 **Date** 对象中。

示例

下面的示例阐释了 **getMinutes** 方法的用法。

```
function TimeDemo(){
   var d, s = "The current local time is: ";
   var c = ":";
   d = new Date();
   s += d.getHours() + c;
   s += d.getMinutes() + c;
   s += d.getSeconds() + c;
   s += d.getMilliseconds();
   return(s);
}
```

要求

版本 3

应用于：

Date 对象

请参见

参考

getUTCMinutes 方法
setMinutes 方法
setUTCMinutes 方法

getMonth 方法

使用当地时间返回 **Date** 对象中的月份值。

```
function getMonth() : Number
```

备注

若要使用协调通用时间 (UTC) 获取月份值，请使用 **getUTCMonth** 方法。

getMonth 方法返回一个介于 0 到 11 之间的整数，该整数指示 **Date** 对象中的月份值。所返回的整数不是用于指示月份的传统数字。而是要比传统数字小 1。如果一个 **Date** 对象中存储的值为“Jan 5, 1996 08:47:00”，那么 **getMonth** 将返回 0。

示例

下面的示例阐释了 getMonth 方法的用法：

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getYear();
   return(s);
}
```

要求

版本 1

应用于：

Date 对象

请参见

参考

getUTCMonth 方法
setMonth 方法
setUTCMonth 方法

getSeconds 方法

使用当地时间返回 **Date** 对象中的秒值。

```
function getSeconds() : Number
```

备注

若要使用协调通用时间 (UTC) 获取秒值，请使用 **getUTCSeconds** 方法。

getSeconds 方法返回一个介于 0 到 59 之间的整数，该整数指示指定 **Date** 对象中的秒值。在下面两种情况下，将返回零。第一种情况是在当前一分钟内的时间还不足以增加一秒。另外一种情况是在创建 **Date** 对象时未将时间存储到其中。若要确定究竟属于哪种情况，唯一的方法是检查小时和分钟值是否也为零。若它们均为零，则几乎可以肯定未将时间存储在 **Date** 对象中。

示例

下面的示例阐释了 **getSeconds** 方法的用法。

```
function TimeDemo(){
   var d, s = "The current local time is: ";
   var c = ":";
   d = new Date();
   s += d.getHours() + c;
   s += d.getMinutes() + c;
   s += d.getSeconds() + c;
   s += d.getMilliseconds();
   return(s);
}
```

要求

版本 1

应用于:

Date 对象

请参见

参考

getUTCSeconds 方法
setSeconds 方法
setUTCSeconds 方法

getTime 方法

返回 **Date** 对象中的时间值。

```
function getTime() : Number
```

备注

getTime 方法返回一个整数值，该整数表示了介于 1970 年 1 月 1 日午夜和 **Date** 对象中的时间值之间的毫秒数。日期的范围大约是 1970 年 1 月 1 日午夜前后的 285,616 年。负数指示 1970 年之前的日期。

在进行多种日期和时间的换算时，通常有必要定义一些与一天、一小时或一分钟内的毫秒数相等的变量。例如：

```
var MinMilli = 1000 * 60
var HrMilli = MinMilli * 60
var DyMilli = HrMilli * 24
```

示例

下面的示例阐释了 **getTime** 方法的用法。

```
function GetTimeTest(){
   var d, s, t;
   var MinMilli = 1000 * 60;
   var HrMilli = MinMilli * 60;
   var DyMilli = HrMilli * 24;
   d = new Date();
   t = d.getTime();
   s = "It's been "
   s += Math.round(t / DyMilli) + " days since 1/1/70";
   return(s);
}
```

要求

版本 1

应用于：

Date 对象

请参见

参考

setTime 方法

JScript 8.0

getTimezoneOffset 方法

返回主机的时间与协调通用时间 (UTC) 之间的分钟差值。

```
function getTimezoneOffset() : Number
```

备注

getTimezoneOffset 方法返回一个整数值，该整数表示了当前计算机上的时间和 UTC 时间之间相差的分钟数。这些值适用于执行脚本的计算机。如果从一个服务器脚本调用该方法，返回值适用于服务器。若从一个客户端脚本调用该方法，则返回值适用于该客户端。

如果您的时间晚于 UTC 时间(例如太平洋夏时制)，则此值为正，而如果您的时间早于 UTC 时间(例如日本)，则此值为负。

例如，假设一个位于洛杉矶的客户端在 12 月 1 日与一个位于纽约市的服务器联系。若在客户端执行 **getTimezoneOffset**，则将返回 480，若在服务器端执行，则返回 300。

示例

下面的示例阐释了 **getTimezoneOffset** 方法的用法。

```
function TZDemo(){
   var d, tz, s = "The current local time is ";
   d = new Date();
   tz = d.getTimezoneOffset();
   if (tz < 0)
      s += tz / 60 + " hours before UTC";
   else if (tz == 0)
      s += "UTC";
   else
      s += tz / 60 + " hours after UTC";
   return(s);
}
```

要求

版本 1

应用于:

Date 对象

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

getUTCDate 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的日期。

```
function getUTCDate() : Number
```

备注

若要使用当地时间获取日期，请使用 **getDate** 方法。

返回值是一个介于 1 到 31 之间的整数，该整数表示 **Date** 对象中的日期值。

示例

下面的示例阐释了 **getUTCDate** 方法的用法。

```
function UTCDateDemo(){
   var d, s = "Today's UTC date is: ";
   d = new Date();
   s += (d.getUTCMonth() + 1) + "/";
   s += d.getUTCDate() + "/";
   s += d.getUTCFullYear();
   return(s);
}
```

要求

版本 3

应用于:

Date 对象

请参见

参考

getDate 方法
setDate 方法
setUTCDate 方法

getUTCDay 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中该天的星期值。

```
function getUTCDay() : Number
```

备注

若要使用当地时间获取星期数，请使用 **getDate** 方法。

getUTCDay 方法返回的值是一个介于 0 到 6 之间的整数，该整数表示一周中的星期数，返回值与星期数按如下方式对应：

值	星期数
0	星期日
1	星期一
2	星期二
3	星期三
4	星期四
5	星期五
6	星期六

示例

下面的示例阐释了 **getUTCDay** 方法的用法。

```
function DateDemo(){
   var d, day, x, s = "Today is ";
   var x = new Array("Sunday", "Monday", "Tuesday");
   x = x.concat("Wednesday","Thursday", "Friday");
   x = x.concat("Saturday");
   d = new Date();
   day = d.getUTCDay();
   return(s += x[day] + " in UTC");
}
```

要求

版本 3

应用于：

Date 对象

请参见

参考

getDay 方法

getUTCFullYear 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的年份值。

```
function getUTCFullYear() : Number
```

备注

若要使用当地时间获取年份，请使用 **getFullYear** 方法。

getUTCFullYear 方法以绝对数的形式返回年份。这样可以避免 2000 年问题，即不会把始于 2000 年 1 月 1 日的日期与始于 1900 年 1 月 1 日的日期混淆。

示例

下面的示例阐释了 **getUTCFullYear** 方法的用法。

```
function UTCDateDemo(){
   var d, s = "Today's UTC date is: ";
   d = new Date();
   s += (d.getUTCMonth() + 1) + "/";
   s += d.getUTCDate() + "/";
   s += d.getUTCFullYear();
   return(s);
}
```

要求

版本 3

应用于:

Date 对象

请参见

参考

getFullYear 方法
setFullYear 方法
setUTCFullYear 方法

getUTCHours 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的小时值。

```
function getUTCHours() : Number
```

备注

若要使用当地时间获取自午夜之后经过的小时数，请使用 **getHours** 方法。

getUTCHours 方法返回一个介于 0 到 23 之间的整数，此整数指示了午夜之后经过的小时数。两种情况下会出现零：在 1:00:00 A.M. 之前，或当创建 **Date** 对象时未将时间存储在该对象中。而要确定究竟是哪种情况，唯一的办法就是检查分钟和秒值是否也为零值。若它们均为零，则几乎可以肯定未将时间存储在 **Date** 对象中。

示例

下面的示例阐释了 **getUTCHours** 方法的用法。

```
function UTCTimeDemo(){
   var d, s = "Current Coordinated Universal Time (UTC) is: ";
   var c = ":";
   d = new Date();
   s += d.getUTCHours() + c;
   s += d.getUTCMinutes() + c;
   s += d.getUTCSeconds() + c;
   s += d.getUTCMilliseconds();
   return(s);
}
```

要求

版本 3

应用于：

Date 对象

请参见

参考

getHours 方法
setHours 方法
setUTCHours 方法

getUTCMilliseconds 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的毫秒值。

```
function getUTCMilliseconds() : Number
```

备注

若要获取用当地时间表示的毫秒数，请使用 getMilliseconds 方法。

返回的毫秒值的范围是 0-999。

示例

下面的示例阐释了 **getUTCMilliseconds** 方法的用法。

```
function UTCTimeDemo(){
   var d, s = "Current Coordinated Universal Time (UTC) is: ";
   var c = ":";
   d = new Date();
   s += d.getUTCHours() + c;
   s += d.getUTCMinutes() + c;
   s += d.getUTCSeconds() + c;
   s += d.getUTCMilliseconds();
   return(s);
}
```

要求

版本 3

应用于:

Date 对象

请参见

参考

getMilliseconds 方法
setMilliseconds 方法
ssetUTCMilliseconds 方法

getUTCMinutes 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的分钟值。

```
function getUTCMinutes() : Number
```

备注

若要获取用当地时间存储的分钟数，请使用 **getMinutes** 方法。

getUTCMinutes 方法返回一个介于 0 到 59 之间的整数，该整数等于包含在 **Date** 对象中的分钟数。两种情况下会出现零：小时之后的时间不足一分钟时，或当创建 **Date** 时未将时间存储在该对象中。而要确定究竟属于哪种情况，唯一的方法就是检查小时和秒值是否也为零值。若它们均为零，则几乎可以肯定未将时间存储在 **Date** 对象中。

示例

下面的示例阐释了 **getUTCMinutes** 方法的用法。

```
function UTCTimeDemo()
{
   var d, s = "Current Coordinated Universal Time (UTC) is: ";
   var c = ":";
   d = new Date();
   s += d.getUTCHours() + c;
   s += d.getUTCMinutes() + c;
   s += d.getUTCSeconds() + c;
   s += d.getUTCMilliseconds();
   return(s);
}
```

要求

版本 3

应用于：

Date 对象

请参见

参考

getMinutes 方法
setMinutes 方法
setUTCMinutes 方法

getUTCMonth 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的月份值。

```
function getUTCMonth(): Number
```

备注

若要获取用当地时间表示的月份，请使用 getMonth 方法。

getUTCMonth 方法返回一个介于 0 到 11 之间的整数，该整数指示 Date 对象中的月份值。所返回的整数不是用于指示月份的传统数字。而是要比传统数字小 1。如果一个 **Date** 对象中存储的值是"Jan 5, 1996 08:47:00.0"，**getUTCMonth** 将返回 0。

示例

下面的示例阐释了 **getUTCMonth** 方法的用法。

```
function UTCDateDemo(){
   var d, s = "Today's UTC date is: ";
   d = new Date();
   s += (d.getUTCMonth() + 1) + "/";
   s += d.getUTCDate() + "/";
   s += d.getUTCFullYear();
   return(s);
}
```

要求

版本 3

应用于:

Date 对象

请参见

参考

getMonth 方法
setMonth 方法
setUTCMonth 方法

getUTCSeconds 方法

使用协调通用时间 (UTC) 返回 **Date** 对象中的秒钟值。

```
function getUTCSeconds(): Number
```

备注

若要获取用当地时间表示的秒数，请使用 **getSeconds** 方法。

getUTCSeconds 方法返回一个介于 0 到 59 之间的整数，该整数指示指定 **Date** 对象中的秒值。两种情况下会出现零：当前一分钟内的时间不足以增加一秒，或当创建 **Date** 对象时未将时间存储在该对象中。而为了确定究竟属于哪种情况，唯一的方法就是检查分钟和小时值是否也为零值。若它们均为零，则几乎可以肯定未将时间存储在 **Date** 对象中。

示例

下面的示例阐释了 **getUTCSeconds** 方法的用法。

```
function UTCTimeDemo(){
   var d, s = "Current Coordinated Universal Time (UTC) is: ";
   var c = ":";
   d = new Date();
   s += d.getUTCHours() + c;
   s += d.getUTCMinutes() + c;
   s += d.getUTCSeconds() + c;
   s += d.getUTCMilliseconds();
   return(s);
}
```

要求

版本 3

应用于：

Date 对象

请参见

参考

getSeconds 方法
setSeconds 方法
setUTCSeconds 方法

getVarDate 方法

将 **Date** 对象中的 VT_DATE 值返回。

```
function getVarDate(): System.DateTime
```

备注

当同 COM 对象、ActiveX® 对象或其他用 VT_DATE 形式接受及返回日期值的对象（如 Visual Basic 和 VBScript）交互时，使用 **getVarDate** 方法。实际的格式取决于区域设置，而不应随 JScript 内部的变化而变化。

要求

版本 3

应用于：

Date 对象

请参见

参考

getDate 方法
parse 方法

getYear 方法

将 **Date** 对象中的年份值返回。

```
function getYear(): Number
```

备注

此方法已过时，之所以提供这个方法，只是为了向后兼容。请改用 **getFullYear** 方法。

对于 1900-1999 这段时间而言，年份是一个两位数整数值，该值作为所存储年份与 1900 年之间的差而被返回。而对于该段时间之外的年份，将返回一个四位数年份。例如，1996 年的返回值是 96，而 1825 和 2025 年则原样返回。

注意

对于 JScript 1.0 版，**getYear** 返回的值为从所提供 **Date** 对象中的年份值减去 1900 后的结果，无论该年份值是多少。例如，1899 年的返回值是 -1，而 2000 年的返回值是 100。

示例

下面的示例阐释了 getYear 方法的用法。

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getYear();
   return(s);
}
```

要求

版本 1

应用于:

Date 对象

请参见

参考

getFullYear 方法
getUTCFullYear 方法
setFullYear 方法
setUTCFullYear 方法
setYear 方法

hasOwnProperty 方法

返回一个布尔值，该值指示一个对象是否具有指定名称的属性。

```
function hasOwnProperty(proName : String) : Boolean
```

参数

proName

必选。一个属性名称的字符串值。

备注

如果该对象具有某指定名称的属性，则 **hasOwnProperty** 方法返回 **true**；否则将返回 **false**。此方法不会检查对象的原型链中是否存在该属性；该属性必须是对象本身的一个成员。

示例

在下面的示例中，所有 **String** 对象共享一个公共 split 方法。

```
var s = new String("JScript");
print(s.hasOwnProperty("split"));
print(String.prototype.hasOwnProperty("split"));
```

该程序的输出为：

```
false
true
```

要求

版本 5.5

应用于：

Object 对象

请参见

参考

in 运算符

indexOf 方法

返回 **String** 对象内第一次出现子字符串的字符位置。

```
function indexOf(subString : String [, startIndex : Number]) : Number
```

参数

subString

必选。在 **String** 对象中搜索的子字符串。

startIndex

可选项。该整数值指定在 **String** 对象内开始搜索的索引。若省略此参数，则搜索从字符串的起始处开始。

备注

indexOf 方法返回一个整数值，该值指示 **String** 对象内子字符串的起始位置。如果未找到子字符串，则返回 -1。

如果 startindex 为负，则将 startindex 视为零。如果它比最大字符位置索引还大，则将它视为可能的最大索引。

搜索将从左向右执行。否则，此方法与 lastIndexOf 相同。

示例

下面的示例阐释了 indexOf 方法的用法。

```
function IndexDemo(str2){
   var str1 = "BABEBIBOBUBABEBIBOBU"
   var s = str1.indexOf(str2);
   return(s);
}
```

要求

版本 1

应用于:

String 对象

请参见

参考

lastIndexOf 方法

JScript 8.0

isFinite 方法

返回一个布尔值，该值指示所提供数字是否是有限值。

```
function isFinite(number : Number) : Boolean
```

参数
number

必选。数值。

备注

若 *number* 是除 **NaN**、负无穷或正无穷之外的任意值，则 **isFinite** 方法将返回 **true**。在这三种情况中，函数返回 **false**。

要求

版本 3

应用于：

Global 对象

请参见
参考
isNaN 方法

JScript 8.0

isNaN 方法

返回一个布尔值，该值指示某值是否为保留值 **NaN**（非数字）。

```
function isNaN(number : Number) : Boolean
```

参数
number

必选。数值。

备注

如果值是 **NaN**，则 **isNaN** 函数返回 **true**，否则返回 **false**。通常使用此函数检测来自 **parseInt** 和 **parseFloat** 方法的返回值。

或者，将变量与其自身进行比较。如果比较的结果不相等，则该变量为 **NaN**。因为 **NaN** 是唯一的一个与其自身不等的值。

要求

版本 1

应用于：

Global 对象

请参见
参考
isFinite 方法
NaN 属性（全局）
parseFloat 方法
parseInt 方法

isPrototypeOf 方法

返回一个布尔值，该值指示对象是否存在于另一个对象的原型链中。

```
function isPrototypeOf(obj : Object) : Boolean
```

参数
obj

必选。一个对象，将对其原型链进行检查。

备注

如果 *obj* 在当前的原型链中有当前对象，则 **isPrototypeOf** 方法返回 **true**。原型链用于在同一个对象类型的不同实例之间共享功能。如果 *obj* 不是对象或者当前对象不出现在 *obj* 的原型链中，则 **isPrototypeOf** 方法返回 **false**。

示例

下面的示例阐释了 **isPrototypeOf** 方法的用法。

```
function test(){
   var re = new RegExp();                          //Initialize variable.
   return (RegExp.prototype.isPrototypeOf(re));  //Return true.
}
```

要求

版本 5.5

应用于:

Object 对象

请参见
其他资源
Visual Basic 和 Visual C# 项目扩展性方法

JScript 8.0

italics 方法

返回 String 对象中文本周围带有 HTML <I> 标记的字符串。

```
function italics() : String
```

备注

未进行任何检查来查看此标记是否已应用于该字符串。

示例

下面的示例说明了 **italics** 方法是如何工作的:

```
var strVariable = "This is a string";
strVariable = strVariable.italics();
```

最后一个语句后面的 `strVariable` 的值为:

```
<I>This is a string</I>
```

要求

版本 1

应用于:

String 对象

请参见

参考

bold 方法

item 方法 (JScript)

返回集合中的当前项。

```
function item() : Number
```

备注

item 方法返回 **Enumerator** 对象中的当前项。如果集合为空或如果当前项未被定义，它将返回 **undefined**。

示例

在下面的代码中，使用了 **item** 方法返回 **Drives** 集合中的一个成员。

```
function ShowDriveList(){
   var fso, s, n, e, x;
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);
   s = "";
   for (; !e.atEnd(); e.moveNext())
   {
      x = e.item();
      s = s + x.DriveLetter;
      s += " - ";
      if (x.DriveType == 3)
         n = x.ShareName;
      else if (x.IsReady)
         n = x.VolumeName;
      else
         n = "[Drive not ready]";
      s +=  n + "<br>";
   }
   return(s);
}
```

要求

版本 3

应用于:

Enumerator 对象

请参见

参考

atEnd 方法
moveFirst 方法
moveNext 方法

方法 (J-R)

方法是作为对象成员的函数。以下方法的名称以字母 j 到 r 开头。

本节内容

join 方法

返回由一个数组的所有元素串联而成的 **String** 对象。

lastIndexOf 方法

将 **String** 对象中子字符串最后出现的位置返回。

lbound 方法

将 VBArray 的指定维度中使用的最小索引值返回。

link 方法

将一个有 HREF 属性的 HTML 定位点放置在 **String** 对象中文本的两侧。

localeCompare 方法

返回一个值，指示两个字符串在当前区域设置中是否相等。

log 方法

返回一个数的自然对数。

match 方法

以数组形式返回使用所提供的 **Regular Expression** 对象对字符串执行搜索所得的结果。

max 方法

返回所提供两个的数值表达式中的较大者。

min 方法

返回所提供的两个数字中的较小者。

moveFirst 方法

在集合中重新将第一项设置为当前项。

moveNext 方法

将当前项移动到集合中的下一项。

parse 方法

分析一个包含日期的字符串，并返回该日期与 1970 年 1 月 1 日午夜之间相差的毫秒数。

parseFloat 方法

返回从字符串转换得到的浮点数。

parseInt 方法

返回从字符串转换得到的整数。

pop 方法

从数组中移除最后一个元素并将该元素返回。

pow 方法

返回基表达式的指定次幂的值。

push 方法

将新元素追加到一个数组中，并返回新的数组长度。

random 方法

返回介于 0 和 1 之间的伪随机数。

replace 方法

返回用正则表达式替换文本的字符串的副本。

reverse 方法

将元素顺序被反转的 **Array** 对象返回。

round 方法

返回舍入到最近整数的指定数值表达式。

相关章节

JScript 参考

列出"JScript 语言参考"所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

方法

按字母分类列出 JScript 中的可用方法，并列出指向每个方法类别的链接。

对象

解释有关 JScript 中的对象的概念，并说明对象与属性和方法的关系，并提供指向特定主题的链接，这些主题将详细介绍 JScript 所支持的对象。

JScript 8.0

join 方法

返回一个字符串值，其中包含一个数组的连接在一起的、并由指定的分隔符分隔的所有元素。

```
function join(separator : String) : String
```

参数

separator

必选。一个字符串，用于在结果 **String** 对象中将数组的一个元素与下一个元素分开。若忽略此参数，则数组元素之间用逗号分隔。

备注

如果数组的任一元素为 **undefined** 或 **null**，则该元素将被视为是空字符串。

示例

下面的示例阐释了 join 方法的用法。

```
function JoinDemo(){
   var a, b;
   a = new Array(0,1,2,3,4);
   b = a.join("-");
   return(b);
}
```

要求

版本 2

应用于：

Array 对象

请参见

参考

String 对象

lastIndexOf 方法

返回 **String** 对象中某个子字符串的最后一个匹配项的索引。

```
function lastIndexOf(substring : String [, startindex : Number ]) : Number
```

参数

substring

必选。在 String 对象内将被搜索的子字符串。

startindex

可选项。该整数值指定在 **String** 对象内开始搜索的索引。若省略该参数，则搜索将从字符串的结尾开始。

备注

lastIndexOf 方法返回一个整数值，指示 **String** 对象内子字符串的开始位置。如果未找到子字符串，则返回 -1。

如果 startindex 为负，则将 startindex 视为零。如果它比最大字符位置索引还大，则将它视为可能的最大索引。

从右向左执行搜索。否则，该方法和 indexOf 相同。

示例

下面的示例阐释了 lastIndexOf 方法的用法。

```
function lastIndexDemo(str2) {
   var str1 = "BABEBIBOBUBABEBIBOBU"
   var s = str1.lastIndexOf(str2);
   return(s);
}
```

要求

版本 1

应用于:

String 对象

请参见

参考

indexOf 方法

lbound 方法

将 VBArray 的指定维度中使用的最小索引值返回。

```
function lbound([dimension : Number]) : Object
```

参数

dimension

可选项。VBArray 的要获知其索引下限的维度。如果忽略此参数，**lbound** 将该参数视为 1。

备注

若 VBArray 为空，**lbound** 方法将返回未定义。如果 *dimension* 大于 VBArray 的维度数或为负，该方法将生成"下标超出范围"错误。

示例

下面的示例由三部分组成。第一部分是创建 Visual Basic 安全数组的 VBScript 代码。第二部分是 JScript 代码，确定该安全数组的维度数和每一维度的下限。由于该安全数组是在 VBScript 中而不是在 Visual Basic 中创建的，因此下限将始终为零。这两部分都位于 HTML 页的 <HEAD> 区域。第三部分是位于 <BODY> 区域内用于运行其他两个部分的 JScript 代码。

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(j, i) = k
         k = k + 1
      Next
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vba){
   var i, s;
   var a = new VBArray(vba);
   for (i = 1; i <= a.dimensions(); i++)
   {
      s = "The lower bound of dimension ";
      s += i + " is ";
      s += a.lbound(i)+ ".<BR>";
      return(s);
   }
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT language="jscript">
   document.write(VBArrayTest(CreateVBArray()));
</SCRIPT>
</BODY>
```

要求

版本 3

应用于:

VBArray 对象

请参见

参考

dimensions 方法
getItem 方法
toArray 方法
ubound 方法

link 方法

返回 **String** 对象中文本周围带有 HTML 定位点和 HREF 属性的字符串。

```
function link(linkstring : String) : String
```

备注

调用 **link** 方法来创建 **String** 对象外部的超链接。

未进行任何检查来查看此标记是否已应用于该字符串。

示例

下面的示例说明了该方法是如何实现此功能的:

```
var strVariable = "This is a hyperlink";
strVariable = strVariable.link("http://www.microsoft.com");
```

最后一个语句后面的 `strVariable` 的值为:

```
<A HREF="http://www.microsoft.com">This is a hyperlink</A>
```

要求

版本 1

应用于:

String 对象

请参见

参考

anchor 方法

JScript 8.0

localeCompare 方法

返回一个值，指示两个字符串在当前区域设置中是否相等。

```
function localeCompare(stringExp : String) : Number
```

参数

stringExp

必选。要与当前字符串对象进行比较的字符串。

备注

localeCompare 对当前字符串对象和 *stringExp* 进行区分区域设置的字符串比较，并返回 -1、0 或 +1，这取决于系统中默认的区域设置的排序顺序。

如果当前字符串对象排在 *stringExp* 之前，则 **localeCompare** 返回 -1；如果当前字符串排在 *stringExp* 之后，则返回 +1。如果返回值为零，则说明这两个字符串是相等的。

要求

版本 5.5

应用于：

String 对象

请参见

参考

toLocaleString 方法

JScript 8.0

log 方法

返回一个数的自然对数。

```
function log(number : Number) : Number
```

参数
number

必选。数值。

备注

返回值是 *number* 参数值的自然对数。基数是 *e*。

要求

版本 1

应用于:

Math 对象

请参见
其他资源
Visual Basic 和 Visual C# 项目扩展性方法

match 方法

使用正则表达式模式对字符串执行搜索，并返回一个包含该搜索结果的数组。

```
function match(rgExp : RegExp) : Array
```

参数

rgExp

必选。包含正则表达式模式和适用标志的 **Regular Expression** 对象的实例。也可以是包含正则表达式模式和标志的变量名或字符串。

备注

如果 **match** 方法没有找到匹配，将返回 **null**。如果找到匹配，则 **match** 方法返回一个数组，并将更新全局 **RegExp** 对象的属性以反映匹配结果。

match 方法返回的数组有三个属性：**input**、**index** 和 **lastIndex**.。**Input** 属性包含整个被搜索的字符串。**Index** 属性包含了在整个被搜索字符串中匹配的子字符串的位置。**lastIndex** 属性包含了前一次匹配中最后一个字符的下一个位置。

如果没有设置全局标志 (**g**)，数组元素 0 包含整个匹配，而元素 1 到元素 *n* 包含了匹配中曾出现过的任一个子匹配。此行为与没有设置全局标志的 **exec** 方法的行为相同。如果设置了全局标志，则元素 0 到 *n* 中包含所有出现的匹配。

示例

下面的示例阐释了 **match** 方法的用法。

```
function MatchDemo(){
   var r, re;         //Declare variables.
   var s = "The rain in Spain falls mainly in the plain";
   re = /ain/i;    //Create regular expression pattern.
   r = s.match(re);   //Attempt match on search string.
   return(r);         //Return first occurrence of "ain".
}
```

本示例阐释设置了 **g** 标志的 match 方法的用法。

```
function MatchDemo(){
   var r, re;         //Declare variables.
   var s = "The rain in Spain falls mainly in the plain";
   re = /ain/ig;      //Create regular expression pattern.
   r = s.match(re);   //Attempt match on search string.
   return(r);         //Return array containing all four
                      // occurrences of "ain".
}
```

下列代码行阐释了 **match** 方法如何用于字符串。

```
var r, re = "Spain";
r = "The rain in Spain".replace(re, "Canada");
```

要求

版本 3

应用于：

String 对象

请参见

参考

exec 方法

RegExp 对象

max 方法

返回零个或更多个所提供的数值表达式中较大的那个。

```
function max([number1 : Number [, ... [, numberN : Number]]]) : Number
```

参数
number1, ..., numberN

必选。数值表达式。

备注

如果未提供参数，则返回值等于 **NEGATIVE_INFINITY**。如果有名为 **NaN** 的参数，则返回值也为 **NaN**。

要求

版本 1

应用于:

Math 对象

请参见
参考
min 方法
NEGATIVE_INFINITY 属性

min 方法

返回零个或更多个所提供的数值表达式中较小的那个。

```
function min([number1 : Number [, ... [, numberN : Number]]]) : Number
```

参数
number1, ..., numberN

必选。数值表达式。

备注

如果未提供参数，则返回值等于 **POSITIVE_INFINITY**。如果有名为 **NaN** 的参数，则返回值也为 **NaN**。

要求

版本 1

应用于:

Math 对象

请参见
参考
max 方法
POSITIVE_INFINITY 属性

moveFirst 方法

将 **Enumerator** 对象中的当前项重置为第一项。

```
function moveFirst()
```

备注

如果集合中没有项，则当前项将被设置为未定义。

示例

在以下示例中，使用 **moveFirst** 方法从列表的开头对 **Drives** 集合的成员进行计算：

```
function ShowFirstAvailableDrive(){
   var fso, s, e, x;                //Declare variables.
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);  //Create Enumerator object.
   e.moveFirst();                   //Move to first drive.
   s = "";                          //Initialize s.
   do
   {
      x = e.item();                 //Test for existence of drive.
      if (x.IsReady)                //See if it's ready.
      {
         s = x.DriveLetter + ":";   //Assign 1<SUP>st</SUP> drive letter to s.
         break;
      }
      else
         if (e.atEnd())             //See if at the end of the collection.
         {
            s = "No drives are available";
            break;
         }
      e.moveNext();                 //Move to the next drive.
   }
   while (!e.atEnd());              //Do while not at collection end.
   return(s);                       //Return list of available drives.
}
```

要求

版本 3

应用于：

Enumerator 对象

请参见

参考

atEnd 方法
item 方法 (JScript)
moveNext 方法

JScript 8.0

moveNext 方法

将当前项移到 **Enumerator** 对象中的下一项。

```
function moveNext()
```

备注

如果枚举数位于集合的末尾，或集合为空，则当前项将被设置为未定义。

在以下示例中，使用 **moveNext** 方法在 **Drives** 集合中移动到下一个驱动器：

```
function ShowDriveList(){
   var fso, s, n, e, x;                 //Declare variables.
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);      //Create Enumerator object.
   s = "";                              //Initialize s.
   for (; !e.atEnd(); e.moveNext())
   {
      x = e.item();
      s = s + x.DriveLetter;            //Add drive letter
      s += " - ";                       //Add "-" character.
      if (x.DriveType == 3)
         n = x.ShareName;               //Add share name.
      else if (x.IsReady)
         n = x.VolumeName;              //Add volume name.
      else
         n = "[Drive not ready]";       //Indicate drive not ready.
      s +=   n + "\n";
   }
   return(s);                           //Return drive status.
}
```

要求

版本 3

应用于：

Enumerator 对象

请参见

参考

atEnd 方法
item 方法 (JScript)
moveFirst 方法

parse 方法

分析一个包含日期的字符串，并返回该日期与 1970 年 1 月 1 日午夜之间相差的毫秒数。

```
function parse(dateVal : {String | System.DateTime} ) : Number
```

参数
dateVal

必选。要么是包含诸如"Jan 5, 1996 08:47:00"格式的日期的字符串，要么是从 ActiveX® 对象或其他对象检索到的 VT_DATE 值。

备注

parse 方法返回一个整数值，此整数表示 *dateVal* 中所提供的日期与 1970 年 1 月 1 日午夜之间相差的毫秒数。

parse 方法是 **Date** 对象的一个静态方法。因为它是一个静态方法，它将按下面的示例中所显示的方法被调用，而不是作为一个已创建的 **Date** 对象中的一个方法被调用。

```
var datestring = "November 1, 1997 10:15 AM";
Date.parse(datestring)
```

下面这些规则控制着 **parse** 方法所能成功分析的字符串：

- 短日期可使用"/"或"-"日期分隔符，但是必须符合"月/日/年"的格式，例如"7/20/96"。
- 以"July 10 1995"形式表示的长日期中的年、月、日可以按任何顺序排列，而且年份可以用 2 位数或 4 位数的形式表示。如果使用 2 位数的形式来表示年份，则该年份必须大于或等于 70。
- 括号中的任何文本都被视为注释。这些括号可以嵌套。
- 逗号和空格被视为分隔符。允许使用多个分隔符。
- 月和日的名称必须具有两个或两个以上的字符。如果两个字符所组成的名称不唯一，则该名称将解析为最晚的一个匹配日期。例如，"Ju"被解析为七月，而不是六月。
- 如果提供一个日期，该日期中所说明的星期数与根据此日期中其他部分所确定的星期数不相符，则此日期中的星期数将被忽略。例如，尽管 1996 年 11 月 9 日实际上是星期五，但"Tuesday November 9 1996"还是可以被接受并进行分析。但是结果的 **Date** 对象中包含的是"Friday November 9 1996"。
- JScript 处理所有的标准时区，以及协调通用时间 (UTC) 和格林威治标准时间 (GMT)。
- 用冒号分隔小时、分钟和秒，但所有这些内容并非都需要指定。"10:"、"10:11"和"10:11:12"都是有效的。
- 若使用的时钟是 24 小时计时的，那么将中午 12 点之后的时间指定为"PM"是错误的。例如，"23:15 PM"就是错误的。
- 包含无效日期的字符串是错误的。例如，一个包含两个年份或两个月份的字符串是错误的。

示例

下面的示例阐释了 **parse** 方法的用法。给函数提供一个日期，则该函数将返回所提供日期与 1970 年 1 月 1 日之间的差。

```
function GetTimeTest(testdate){
   var s, t;                    //Declare variables.
   var MinMilli = 1000 * 60;       //Initialize variables.
   var HrMilli = MinMilli * 60;
   var DyMilli = HrMilli * 24;
   t = Date.parse(testdate);       //Parse testdate.
   s = "There are "                //Create return string.
   s += Math.round(Math.abs(t / DyMilli)) + " days "
   s += "between " + testdate + " and 1/1/70";
   return(s);                      //Return results.
}
```

要求

版本 1

应用于:

Date 对象

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

parseFloat 方法

返回从字符串转换得到的浮点数。

```
function parseFloat(numString : String) : Number
```

参数
numString

必选。一个表示浮点数的字符串。

备注

parseFloat 方法返回一个与 numString 中包含的数字相等的数字值。如果没有 numString 前缀可以被成功地分析为浮点数，则返回 **NaN**（非数字）。

可使用 **isNaN** 方法测试是否为 **NaN**。

示例

在下面的示例中，**parseFloat** 方法用于将两个字符串转换为数字。

```
        parseFloat("abc")      // Returns NaN.
parseFloat("1.2abc")   // Returns 1.2.
```

要求

版本 1

应用于：

Global 对象

请参见
参考
isNaN 方法
parseInt 方法
String 对象

parseInt 方法

返回从字符串转换得到的整数。

```
function parseInt(numString : String [, radix : Number]) : Number
```

参数
numString

必选。要转换为数字的字符串。

radix

可选项。介于 2 和 36 之间的一个值，用于指示 numString 所包含的数字的进制。如果未提供，则前缀为 '0x' 的字符串被视为十六进制，前缀为 '0' 的字符串被视为八进制。所有其他字符串都被视为是十进制。

备注

parseInt 方法返回与包含在 *numString* 中的数字相等的整数值。如果没有 numString 的前缀可以被成功地分析为一个整数，则返回 **NaN**（非数字）。

可使用 **isNaN** 方法测试是否为 **NaN**。

示例

在下面的示例中，**parseInt** 方法用于将两个字符串转换为数字。

```
        parseInt("abc")     // Returns NaN.
parseInt("12abc")   // Returns 12.
```

要求

版本 1

应用于：

Global 对象

请参见
参考
isNaN 方法
parseFloat 方法
String 对象
valueOf 方法

pop 方法

从数组中移除最后一个元素并将该元素返回。

```
function pop() : Object
```

备注

如果该数组为空，则返回 **undefined**。

要求

版本 5.5

应用于：

Array 对象

请参见

参考

push 方法

JScript 8.0

pow 方法

返回基数表达式的指定次幂的值。

```
function pow(base : Number, exponent : Number) : Number
```

参数

base

必选。表达式的基数值。

exponent

必选。表达式的指数值。

备注

pow 方法返回与 baseexponent 相等的数值表达式。

示例

下面的示例阐释了 **pow** 方法的用法。

```
var x = Math.pow(10,3); // x is assigned the value 1000.
```

要求

版本 1

应用于:

Math 对象

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

push 方法

将新元素追加到一个数组中，并返回新的数组长度。

```
function push([item1 : Object [, ... [, itemN : Object]]]) : Number
```

参数
item1, ..., itemN

可选项。**Array** 的新元素。

备注

push 方法将新元素按出现的顺序追加。如果参数之一是一个数组，则该数组将作为单个元素添加到数组中。如果要合并来自两个或多个数组中的元素，请使用 **concat** 方法。

要求

版本 5.5

应用于：

Array 对象

请参见
参考
concat 方法（数组）
pop 方法

JScript 8.0

random 方法

返回介于 0 和 1 之间的伪随机数。

```
function random() : Number
```

备注

生成的伪随机数介于 0（包括 0）和 1（不包括）之间，即返回值可以是 0，但它永远小于 1。在第一次加载 JScript 时，自动给随机数发生器提供种子。

要求

版本 1

应用于：

Math 对象

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

replace 方法

返回一个字符串的副本，该字符串的文本已被使用正则表达式或搜索字符串替换。

```
function replace(rgExp : RegExp, replaceText : String) : String
```

参数

rgExp

必选。包含正则表达式模式和适用标志的 **Regular Expression** 对象的实例。也可以是 **String** 对象或文本。如果 *rgExp* 不是 **Regular Expression** 对象的实例，它将被转换为字符串，并对结果进行精确的搜索；字符串将不会被试图转化为正则表达式。

replaceText

必选。一个 **String** 对象或字符串文本，包含用于替换当前字符串对象中 *rgExp* 的每个成功匹配的文本。在 Jscript 5.5 或更高版本中，*replaceText* 参数也可是返回替换文本的函数。

备注

在完成指定的替换之后，**replace** 方法的结果是当前字符串对象的副本。

下面任意的匹配变量都可用于识别最新的匹配及匹配字符串。在需要动态确定替换字符串的文本替换中可以使用匹配变量。

字符	含义
$$	**$**（JScript 5.5 或更高版本）
$&	指定当前字符串对象中与整个模式相匹配的部分。（JScript 5.5 或更高版本）
$`	指定当前字符串对象中位于 **$&** 所描述的匹配前面的部分。（JScript 5.5 或更高版本）
$'	指定当前字符串对象中位于 **$&** 所描述的匹配后面的部分。（JScript 5.5 或更高版本）
$ *n*	第 *n* 个捕获到的子匹配，这里 *n* 为从 1 到 9 的十进制一位数。（JScript 5.5 或更高版本）
$ *nn*	第 *nn* 个捕获到的子匹配，这里 *nn* 为从 01 到 99 的十进制二位数。（JScript 5.5 或更高版本）

如果 *replaceText* 是一个函数，对于每个匹配的子字符串，调用该函数时带有下面的 *m* + 3 个参数，这里 *m* 是在 *rgExp* 中用于捕获的左括弧的个数。第一个参数是匹配的子字符串。接下来的 *m* 个参数是搜索中捕获到的全部结果。参数 *m* + 2 是当前字符串对象中发生匹配位置的偏移量，而参数 *m* + 3 是当前字符串对象。结果为将每一匹配的子字符串替换为函数调用的相应返回值后的字符串值。

Replace 方法更新全局 **RegExp** 对象的属性。

示例

下面的示例阐释了 **replace** 方法的用法，即用词"A"替换词"The"的第一个实例。请注意因为该模式区分大小写，所以只替换词"The"的第一个实例。

```
function ReplaceDemo(){
   var r, re;                    //Declare variables.
   var ss = "The man hit the ball with the bat.\n";
   ss += "while the fielder caught the ball with the glove.";
   re = /The/g;                 //Create regular expression pattern.
   r = ss.replace(re, "A");     //Replace "The" with "A".
   return(r);                   //Return string with replacement made.
}
```

另外，**replace** 方法也可以替换模式中的子表达式。下面的示例将字符串中的每对单词进行了交换：

```
function ReplaceDemo(){
```

```
    var r, re;                      //Declare variables.
    var ss = "The rain in Spain falls mainly in the plain.";
    re = /(\S+)(\s+)(\S+)/g;        //Create regular expression pattern.
    r = ss.replace(re, "$3$2$1");   //Swap each pair of words.
    return(r);                      //Return resulting string.
}
```

下面的示例在 JScript 5.5 及更高版本中运行，执行了一个从华氏到摄氏的转换，此例阐释了如何使用一个像 *replaceText* 那样的函数。若要知道该函数是如何运行的，请传递一个包含数字的字符串，该数字后要紧跟着一个"F"（例如，"Water boils at 212F"）。

```
function f2c(s) {
  var test = /(\d+(\.\d*)?)F\b/g;    //Initialize pattern.
  return(s.replace
    (test,
      function($0,$1,$2) {
        return((($1-32) * 5/9) + "C");
      }
    )
  );
}
document.write(f2c("Water freezes at 32F and boils at 212F."));
```

要求

版本 1

应用于:

String 对象

请参见

参考

exec 方法
match 方法
RegExp 对象
search 方法
test 方法

reverse 方法

将元素顺序被反转的 **Array** 对象返回。

```
function reverse() : Array
```

备注

reverse 方法将一个 **Array** 对象中的元素按所在位置进行反转。在执行过程中，此方法并不创建新 **Array** 对象。

如果数组是不连续的，**reverse** 方法将在数组中创建元素，这些元素将填充数组中的间隙。所创建的这些元素的值全部未定义。

示例

下面的示例阐释了 **reverse** 方法的用法。

```
function ReverseDemo(){
   var a, l;                  //Declare variables.
   a = new Array(0,1,2,3,4);  //Create an array and populate it.
   l = a.reverse();           //Reverse the contents of the array.
   return(l);                 //Return the resulting array.
}
```

要求

版本 2

应用于:

Array 对象

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

JScript 8.0

round 方法

返回所提供的舍入到最近整数的数值表达式。

```
function round(number : Number) : Number
```

参数
number

必选。数值表达式。

备注

如果 *number* 的小数部分大于等于 0.5，返回值为大于 *number* 的最小整数。否则，**round** 将返回小于或等于 number 的最大整数。

要求

版本 1

应用于：

Math 对象

请参见
其他资源
Visual Basic 和 Visual C# 项目扩展性方法

JScript 8.0

方法 (S)

方法是作为对象成员的函数。以下方法的名称以字母 s 开头。

本节内容

search 方法

返回正则表达式搜索中第一个子字符串匹配项的位置。

setDate 方法

使用当地时间设置 **Date** 对象的数值日期。

setFullYear 方法

使用当地时间设置 Date 对象中的年份值。

setHours 方法

使用当地时间设置 **Date** 对象中的小时值。

setMilliseconds 方法

使用当地时间设置 **Date** 对象中的毫秒值。

setMinutes 方法

使用当地时间设置 **Date** 对象中的分钟值。

setMonth 方法

使用当地时间设置 **Date** 对象中的月份值。

setSeconds 方法

使用当地时间设置 **Date** 对象中的秒钟值。

setTime 方法

设置 **Date** 对象中的日期和时间值。

setUTCDate 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的数值日期。

setUTCFullYear 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的年份值。

setUTCHours 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的小时值。

setUTCMilliseconds 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的毫秒值。

setUTCMinutes 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的分钟值。

setUTCMonth 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的月份值。

setUTCSeconds 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的秒钟值。

setYear 方法

设置 **Date** 对象中的年份值。

shift 方法

从数组中移除第一个元素并将返回该元素。

sin 方法

返回一个数的正弦值。

slice 方法(数组)

返回一个数组中的一部分。

slice 方法(字符串)

返回字符串的片段。

small 方法

将 HTML 的 <SMALL> 标记添加到 String 对象中的文本两端。

sort 方法

返回一个元素已经进行了排序的 **Array** 对象。

splice 方法

从一个数组中移除元素，如有必要，在所移除元素的位置上插入新元素，并返回所移除的元素。

split 方法

返回一个字符串拆分为若干子字符串时所产生的字符串数组。

sqrt 方法

返回一个数的平方根。

strike 方法

将 HTML <STRIKE> 标记放置到 **String** 对象中的文本两侧。

sub 方法

将 HTML <SUB> 标记放置到 **String** 对象中的文本两侧。

substr 方法

返回一个从指定位置开始，并具有指定长度的子字符串。

substring 方法

返回位于 **String** 对象中指定位置的子字符串。

sup 方法

将 HTML <SUP> 标记放置到 **String** 对象中的文本两侧。

相关章节

JScript 参考

列出"JScript 语言参考"所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

方法

按字母分类列出 JScript 中的可用方法，并列出指向每个方法类别的链接。

对象

解释有关 JScript 中的对象的概念，并说明对象与属性和方法的关系，并提供指向特定主题的链接，这些主题将详细介绍 JScript 所支持的对象。

search 方法

返回正则表达式搜索中第一个子字符串匹配项的位置。

```
function search(rgExp : RegExp) : Number
```

参数
rgExp

必选。包含正则表达式模式和适用标志的 **Regular Expression** 对象的实例。

备注

search 方法表示是否找到了匹配项。如果找到一个匹配项，则 search 方法将返回一个整数值，该值指示发生匹配处距字符串开头的偏移量。如果没有找到匹配项，则返回 -1。

示例

下面的示例阐释了 search 方法的用法。

```
function SearchDemo(){
   var r, re;                    //Declare variables.
   var s = "The rain in Spain falls mainly in the plain.";
   re = /falls/i;                //Create regular expression pattern.
   r = s.search(re);             //Search the string.
   return(r);                    //Return the index to the first match
                                 //or -1 if no match is found.
}
```

要求

版本 3

应用于:

String 对象

请参见
参考
exec 方法
match 方法
正则表达式对象
replace 方法
test 方法
概念
正则表达式语法

setDate 方法

在当地时区中设置 **Date** 对象的日期。

```
function setDate(numDate : Number)
```

参数
numDate

必选。一个与数值日期相等的数值。

备注

若要使用协调通用时间 (UTC) 设置日期值，请使用 **setUTCDate** 方法。

如果 *numDate* 的值大于 **Date** 对象中所存储的月份中的天数或为负数，那么日期将被设置为由 *numDate* 减去所存储月份中天数而得到的日期。例如，如果所存储的日期是 1996 年 1 月 5 日，并调用了方法 setDate(32)，那么日期将变为 1996 年 2 月 1 日。负数的处理方法与此相似。

示例

下面的示例阐释了 **setDate** 方法的用法。

```
function SetDateDemo(newdate){
   var d, s;                  //Declare variables.
   d = new Date();            //Create date object.
   d.setDate(newdate);        //Set date to newdate.
   s = "Current setting is ";
   s += d.toLocaleString();
   return(s);                 //Return newly set date.
}
```

要求

版本 3

应用于:

Date 对象

请参见
参考
getDate 方法
getUTCDate 方法
setUTCDate 方法

setFullYear 方法

使用当地时间设置 **Date** 对象中的年份值。

```
function setFullYear(numYear : Number [, numMonth Number [, numDate Number]])
```

参数

numYear

必选。一个等于年份的数值。

numMonth

可选项。一个等于月份的数值。如果提供了 numDate，那么必须提供此参数。

numDate

可选项。一个等于日期的数值。

备注

如果未指定可选参数，那么所有采用可选参数的 set 方法都将使用相应的 get 方法返回的数值。例如，如果 *numMonth* 参数是可选的，但是尚未指定，则 JScript 将使用 **getMonth** 方法的返回值。

此外，如果参数的值大于其取值范围或者是负数，则其他存储的值都将得到相应的修改。

若要使用协调通用时间 (UTC) 设置年份，请使用 **setUTCFullYear** 方法。

Date 对象中所支持的年份范围大约是 1970 年前后 285,616 年。

示例

下面的示例阐释了 **setFullYear** 方法的用法：

```
function SetFullYearDemo(newyear){
   var d, s;                  //Declare variables.
   d = new Date();            //Create Date object.
   d.setFullYear(newyear);    //Set year.
   s = "Current setting is ";
   s += d.toLocaleString();
   return(s);                 //Return new date setting.
}
```

要求

版本 3

应用于：

Date 对象

请参见

参考

getFullYear 方法
getUTCFullYear 方法
setUTCFullYear 方法

setHours 方法

使用当地时间设置 **Date** 对象中的小时值。

```
function setHours(numHours : Number [, numMin : Number [, numSec : Number [, numMilli : Num
ber ]]])
```

参数
numHours

必选。一个等于小时值的数值。

numMin

可选项。一个等于分钟值的数值。

numSec

可选项。一个等于秒值的数值。

numMilli

可选项。一个等于毫秒值的数值。

备注

如果您没有指定可选参数，那么所有采用可选参数的 **set** 方法都将使用相应的 **get** 方法返回的值。例如，如果 *numMinutes* 参数是可选的，但是没有被指定，则 JScript 将使用 **getMinutes** 方法的返回值。

若要使用协调通用时间 (UTC) 设置小时值，请使用 **setUTCHours** 方法。

如果参数值大于其取值范围或者为负，则会相应地修改存储的其他值。例如，如果存储的日期是"Jan 5, 1996 00:00:00"，并调用了 **setHours(30)**，则该日期将变为"Jan 6, 1996 06:00:00"。负数的处理方法与此相似。

示例

下面的示例阐释了 **setHours** 方法的用法。

```
function SetHoursDemo(nhr, nmin, nsec){
   var d, s;                     //Declare variables.
   d = new Date();               //Create Date object.
   d.setHours(nhr, nmin, nsec);  //Set hours, minutes, & seconds.
   s = "Current setting is " + d.toLocaleString()
   return(s);                    //Return new date setting.
}
```

要求

版本 3

应用于:

Date 对象

请参见
参考
getHours 方法
getUTCHours 方法
setUTCHours 方法

setMilliseconds 方法

使用当地时间设置 **Date** 对象中的毫秒值。

```
function setMilliseconds(numMilli : Number)
```

参数
numMilli

必选。一个等于毫秒值的数值。

备注

若要使用协调通用时间 (UTC) 设置毫秒值，请使用 **setUTCMilliseconds** 方法。

如果 *numMilli* 的值大于 999 或为负，则所存储的秒数（以及分钟、小时等，如有必要）将增加适当的数量。

示例

下面的示例阐释了 **setMilliseconds** 方法的用法。

```
function SetMSecDemo(nmsec){
   var d, s;                    //Declare variables.
   var sep = ":";               //Initialize separator.
   d = new Date();              //Create Date object.
   d.setMilliseconds(nmsec);    //Set milliseconds.
   s = "Current setting is ";
   s += d.toLocaleString() + sep + d.getMilliseconds();
   return(s);                   //Return new date setting.
}
```

要求

版本 3

应用于:

Date 对象

请参见
参考
getMilliseconds 方法
getUTCMilliseconds 方法
ssetUTCMilliseconds 方法

setMinutes 方法

使用当地时间设置 Date 对象中的分钟值。

```
function setMinutes(numMinutes : Number [, numSeconds : Number [, numMilli : Number]])
```

参数

numMinutes

必选。一个等于分钟值的数值。

numSeconds

可选项。一个等于秒值的数值。如果使用了 numMilli 参数，那么必须提供此参数。

numMilli

可选项。一个等于毫秒值的数值。

备注

如果您没有指定可选参数，那么所有采用可选参数的 **set** 方法都将使用相应的 **get** 方法返回的值。例如，如果 *numSeconds* 参数是可选的，但是没有被指定，则 JScript 将使用 **getSeconds** 方法的返回值。

若要使用协调通用时间 (UTC) 设置分钟值，请使用 **setUTCMinutes** 方法。

如果参数值大于其取值范围或者为负，则会相应地修改存储的其他值。例如，如果存储的日期是"Jan 5, 1996 00:00:00"，并调用 **setMinutes(90)**，则日期将变为"Jan 5, 1996 01:30:00"。负数的处理方法与此相似。

示例

下面的示例阐释了 **setMinutes** 方法的用法。

```
function SetMinutesDemo(nmin, nsec){
   var d, s;                     //Declare variables.
   d = new Date();               //Create Date object.
   d.setMinutes(nmin, nsec);     //Set minutes.
   s = "Current setting is " + d.toLocaleString()
   return(s);                    //Return new setting.
}
```

要求

版本 1

应用于:

Date 对象

请参见

参考

getMinutes 方法
getUTCMinutes 方法
setUTCMinutes 方法

setMonth 方法

使用当地时间设置 **Date** 对象中的月份值。

```
function setMonth(numMonth : Number [, dateVal : Number])
```

参数
numMonth

必选。一个等于月份的数值。

dateVal

可选项。一个表示日期的数值。如果没有提供此参数，则使用通过调用 **getDate** 方法得到的数值。

备注

若要使用协调通用时间 (UTC) 设置月份值，请使用 **setUTCMonth** 方法。

如果 *numMonth* 的值大于 11（数值 0 表示一月）或为负，那么所存储的年份将相应地得到修改。例如，如果所存储的日期是"Jan 5, 1996"，并且调用了 setMonth(14) 方法，那么该日期就变为"Mar 5, 1997"。

示例

下面的示例阐释了 **setMonth** 方法的用法。

```
function SetMonthDemo(newmonth){
   var d, s;               //Declare variables.
   d = new Date();         //Create Date object.
   d.setMonth(newmonth);   //Set month.
   s = "Current setting is ";
   s += d.toLocaleString();
   return(s);              //Return new setting.
}
```

要求

版本 1

应用于:

Date 对象

请参见
参考
getMonth 方法
getUTCMonth 方法
setUTCMonth 方法

setSeconds 方法

使用当地时间设置 **Date** 对象中的秒钟值。

```
function setSeconds(numSeconds : Number [, numMilli : Number])
```

参数

numSeconds

必选。一个等于秒值的数值。

numMilli

可选项。一个等于毫秒值的数值。

备注

如果您没有指定可选参数，那么所有采用可选参数的 **set** 方法都将使用相应的 **get** 方法返回的值。例如，如果 *numMilli* 参数是可选的，但是没有被指定，则 JScript 将使用 **getMilliseconds** 方法的返回值。

若要使用协调通用时间 (UTC) 设置秒值，请使用 **setUTCSeconds** 方法。

如果参数值大于其取值范围或者为负，则会相应地修改存储的其他值。例如，如果存储的日期是"Jan 5, 1996 00:00:00"，而且调用了函数 setSeconds(150)，日期将变为"Jan 5, 1996 00:02:30"。

示例

下面的示例阐释了 **setSeconds** 方法的用法。

```
function SetSecondsDemo(nsec, nmsec){
   var d, s;                   //Declare variables.
   var sep = ":";
   d = new Date();             //Create Date object.
   d.setSeconds(nsec, nmsec);  //Set seconds and milliseconds.
   s = "Current setting is ";
   s += d.toLocaleString() + sep + d.getMilliseconds();
   return(s);                  //Return new setting.
}
```

要求

版本 1

应用于:

Date 对象

请参见

参考

getSeconds 方法
getUTCSeconds 方法
setUTCSeconds 方法

JScript 8.0

setTime 方法

设置 **Date** 对象中的日期和时间值。

```
function setTime(milliseconds : Number)
```

参数
milliseconds

必选。一个数值，表示自 UTC 1970 年 1 月 1 日午夜之后经过的毫秒数。

备注

如果 milliseconds 是一个负值，那它就表示 1970 年之前的日期。可用的日期范围大约是 1970 年前后 285,616 年。

使用 **setTime** 方法对日期和时间所进行的设置与时区无关。

示例

下面的示例阐释了 **setTime** 方法的用法。

```
function SetTimeTest(newtime){
   var d, s;                  //Declare variables.
   d = new Date();            //Create Date object.
   d.setTime(newtime);        //Set time.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                 //Return new setting.
}
```

要求

版本 1

应用于:

Date 对象

请参见
参考
getTime 方法

setUTCDate 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的数值日期。

```
function setUTCDate(numDate : Number)
```

参数
numDate

必选。一个与数值日期相等的数值。

备注

若要使用当地时间设置日期，请使用 **setDate** 方法。

如果 *numDate* 的值大于 **Date** 对象中所存储的月份中的天数或为负数，那么日期将被设置为由 *numDate* 减去所存储月份中天数而得到的日期。例如，如果所存储的日期是 1996 年 1 月 5 日，并且调用了方法 setUTCDate(32)，那么日期将变为 1996 年 2 月 1 日。负数的处理方法与此相似。

示例

下面的示例阐释了 **setUTCDate** 方法的用法。

```
function SetUTCDateDemo(newdate){
   var d, s;                    //Declare variables.
   d = new Date();              //Create Date object.
   d.setUTCDate(newdate);       //Set UTC date.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                   //Return new setting.
}
```

要求

版本 3

应用于:

Date 对象

请参见
参考
getDate 方法
getUTCDate 方法
setDate 方法

setUTCFullYear 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的年份值。

```
function setUTCFullYear(numYear : Number [, numMonth : Number [, numDate : Number]])
```

参数

numYear

必选。一个等于年份的数值。

numMonth

可选项。一个等于月份的数值。

numDate

可选项。一个等于日期的数值。

备注

如果您没有指定可选参数，那么所有采用可选参数的 **set** 方法都将使用相应的 **get** 方法返回的值。例如，如果 *numMonth* 参数是可选的，但是没有被指定，则 JScript 将使用 **getUTCMonth** 方法的返回值。

此外，如果参数的值大于其取值范围或者是负数，则其他存储的值都将得到相应的修改。

若要使用当地时间设置年份，请使用 **setFullYear** 方法。

Date 对象中所支持的年份范围大约是 1970 年前后 285,616 年。

示例

下面的示例阐释了 **setUTCFullYear** 方法的用法。

```
function SetUTCFullYearDemo(newyear){
   var d, s;                    //Declare variables.
   d = new Date();              //Create Date object.
   d.setUTCFullYear(newyear);   //Set UTC full year.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                   //Return new setting.
}
```

要求

版本 3

应用于:

Date 对象

请参见

参考

getFullYear 方法
getUTCFullYear 方法
setFullYear 方法

setUTCHours 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的小时值。

```
function setUTCHours(numHours : Number [, numMin : Number [, numSec : Number [, numMilli :
Number]]])
```

参数

numHours

必选。一个等于小时值的数值。

numMin

可选项。一个等于分钟值的数值。

numSec

可选项。一个等于秒值的数值。

numMilli

可选项。一个等于毫秒值的数值。

备注

如果您没有指定可选参数，那么所有采用可选参数的 **set** 方法都将使用相应的 **get** 方法返回的值。例如，如果 *numMin* 参数是可选的，但是没有被指定，则 JScript 将使用 **getUTCMinutes** 方法的返回值。

若要使用当地时间设置小时值，请使用 **setHours** 方法。

如果参数值大于其范围或为负数，则其他存储的值都将得到相应的修改。例如，如果所存储的日期是"Jan 5, 1996 00:00:00"，并调用了 setUTCHours(30) 方法，那么日期将变为"Jan 6, 1996 06:00:00"。

示例

下面的示例阐释了 **setUTCHours** 方法的用法。

```
function SetUTCHoursDemo(nhr, nmin, nsec){
   var d, s;                        //Declare variables.
   d = new Date();                  //Create Date object.
   d.setUTCHours(nhr, nmin, nsec);  //Set UTC hours, minutes, seconds.
   s = "Current setting is " + d.toUTCString()
   return(s);                       //Return new setting.
}
```

要求

版本 3

应用于:

Date 对象

请参见

参考

getHours 方法
getUTCHours 方法
setHours 方法

ssetUTCMilliseconds 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的毫秒值。

```
function setUTCMilliseconds(numMilli : Number)
```

参数
numMilli

必选。一个等于毫秒值的数值。

备注

若要使用当地时间设置毫秒值，请使用 **setMilliseconds** 方法。

如果 *numMilli* 的值大于 999 或者是负数，那么存储的秒值（以及分钟、小时等，如有必要）将增加一个适当的数量。

示例

下面的示例阐释了 **setUTCMilliseconds** 方法的用法。

```
function SetUTCMSecDemo(nmsec){
   var d, s;                          //Declare variables.
   var sep = ":";                     //Initialize separator.
   d = new Date();                    //Create Date object.
   d.setUTCMilliseconds(nmsec);       //Set UTC milliseconds.
   s = "Current setting is ";
   s += d.toUTCString() + sep + d.getUTCMilliseconds();
   return(s);                         //Return new setting.
}
```

要求

版本 3

应用于:

Date 对象

请参见
参考
getMilliseconds 方法
getUTCMilliseconds 方法
setMilliseconds 方法

setUTCMinutes 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的分钟值。

```
function setUTCMinutes(numMinutes : Number [, numSeconds : Number [, numMilli : Number ]])
```

参数

numMinutes

必选。一个等于分钟值的数值。

numSeconds

可选项。一个等于秒值的数值。如果使用了 *numMilli* 参数，那么必须提供此参数。

numMilli

可选项。一个等于毫秒值的数值。

备注

如果您没有指定可选参数，那么所有采用可选参数的 **set** 方法都将使用相应的 **get** 方法返回的值。例如，如果 *numSeconds* 参数是可选的，但是没有被指定，则 JScript 将使用 **getUTCSeconds** 方法的返回值。

若要使用当地时间修改分钟值，请使用 **setMinutes** 方法。

如果参数值大于其范围或为负数，则其他存储的值都将得到相应的修改。例如，如果所存储的日期是"Jan 5, 1996 00:00:00"，并调用了 setUTCMinutes(70) 方法，那么日期将变为"Jan 5, 1996 01:10:00.00"。

示例

下面的示例阐释了 **setUTCMinutes** 方法的用法：

```
function SetUTCMinutesDemo(nmin, nsec){
   var d, s;                    //Declare variables.
   d = new Date();              //Create Date object.
   d.setUTCMinutes(nmin,nsec);  //Set UTC minutes.
   s = "Current setting is " + d.toUTCString()
   return(s);                   //Return new setting.
}
```

要求

版本 3

应用于：

Date 对象

请参见

参考

getMinutes 方法
getUTCMinutes 方法
setMinutes 方法

setUTCMonth 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的月份值。

```
function setUTCMonth(numMonth : Number [, dateVal : Number ])
```

参数

numMonth

必选。一个等于月份的数值。

dateVal

可选项。一个表示日期的数值。如果没有提供此参数，那么将使用调用 **getUTCDate** 方法得到的数值。

备注

若要使用当地时间设置月份值，请使用 **setMonth** 方法。

如果 *numMonth* 的值大于 11（月份 0 代表一月）或者是负数，那么所存储的年份也将相应地得到增加或减少。例如，如果所存储的日期是"Jan 5, 1996 00:00:00.00"，并调用了 setUTCMonth(14) 方法，那么日期将变为"Mar 5, 1997 00:00:00.00"。

示例

下面的示例阐释了 **setUTCMonth** 方法的用法。

```
function SetUTCMonthDemo(newmonth){
   var d, s;                        //Declare variables.
   d = new Date();                  //Create Date object.
   d.setUTCMonth(newmonth);         //Set UTC month.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                       //Return new setting.
}
```

要求

版本 3

应用于:

Date 对象

请参见

参考

getMonth 方法
getUTCMonth 方法
setMonth 方法

setUTCSeconds 方法

使用协调通用时间 (UTC) 设置 **Date** 对象中的秒钟值。

```
function setUTCSeconds(numSeconds : Number [, numMilli : Number ])
```

参数

numSeconds

必选。一个等于秒值的数值。

numMilli

可选项。一个等于毫秒值的数值。

备注

如果您没有指定可选参数，那么所有采用可选参数的 **set** 方法都将使用相应的 **get** 方法返回的值。例如，如果 *numMilli* 参数是可选的，但是没有被指定，则 JScript 将使用 **getUTCMilliseconds** 方法的返回值。

若要使用当地时间设置秒值，请使用 **setSeconds** 方法。

如果参数值大于其取值范围或者为负，则会相应地修改存储的其他值。例如，如果所存储的日期是"Jan 5, 1996 00:00:00"，并调用了 setSeconds(150) 方法，那么日期将变为"Jan 5, 1996 00:02:30.00"

示例

下面的示例阐释了 **setSeconds** 方法的用法。

```
function SetUTCSecondsDemo(nsec, nmsec){
   var d, s;                        //Declare variables.
   d = new Date();                  //Create Date object.
   d.setUTCSeconds(nsec, nmsec);    //Set UTC seconds and milliseconds.
   s = "Current UTC milliseconds setting is ";
   s += d.getUTCMilliseconds();     //Get new setting.
   return(s);                       //Return new setting.
}
```

要求

版本 3

应用于:

Date 对象

请参见

参考

getSeconds 方法
getUTCSeconds 方法
setSeconds 方法

setYear 方法

设置 **Date** 对象中的年份值。

```
function setYear(numYear : Number)
```

参数
numYear

必选。一个等于年份减去 1900 的数值。

备注

这个方法已经过时，之所以仍然保留，只是为了向后兼容。请改用 **setFullYear** 方法。

要将 Date 对象的年份设置为 1997，请调用 setYear(97)。而要将年份设置为 2010，请调用 setYear(2010)。最后，要将年份设置为 0-99 范围内的某一年，请使用 **setFullYear** 方法。

注意

对于 JScript 1.0 版本，无论 *numYear* 提供的年份值是多少，**setYear** 使用的值都为该年份值加上 1900。例如，要将年份设置为 1899，则 *numYear* 参数的值是 -1，而要将年份设置为 2000，则 *numYear* 参数的值是 100。

要求

版本 1

应用于:

Date 对象

请参见
参考
getFullYear 方法
getUTCFullYear 方法
getYear 方法
setFullYear 方法
setUTCFullYear 方法

JScript 8.0

shift 方法

从数组中移除第一个元素并返回该元素。

```
function shift() : Object
```

备注

shift 方法将数组中的第一个元素移除并返回它。

要求

版本 5.5

应用于:

Array 对象

请参见

参考

unshift 方法

JScript 8.0

sin 方法

返回一个数的正弦值。

```
function sin(number : Number) : Number
```

参数
number

必选。一个用于计算其正弦的数值表达式。

备注

返回值为数值参数的正弦。

要求

版本 1

应用于:

Math 对象

请参见
参考
acos 方法
asin 方法
atan 方法
cos 方法
tan 方法

slice 方法（数组）

返回一个数组中的一部分。

```
function slice(start : Number [, end : Number]) : Array
```

参数

start

必选。一个指向数组指定部分的开头的索引。

end

可选项。一个指向数组指定部分的结尾的索引。

备注

slice 方法返回一个 **Array** 对象，其中包含了数组的指定部分。

slice 方法一直复制到 *end* 所指示的元素，但是不包括该元素。如果 *start* 为负，则将其视为 *length* + *start*，此处 *length* 为数组的长度。如果 *end* 为负，就将其视为 *length* + *end*，此处 *length* 为数组的长度。如果省略 *end*，则将一直提取到数组的结尾。如果 *end* 出现在 *start* 之前，则不会将任何元素复制到新数组中。

示例

在下面的示例中，除了最后一个元素之外，myArray 中所有的元素都被复制到 newArray 中：

```
var myArray = new Array(4,3,5,65);
var newArray = myArray.slice(0, -1)
```

要求

版本 3

应用于：

Array 对象

请参见

参考

slice 方法（字符串）

String 对象

slice 方法（字符串）

返回字符串的片段。

```
function slice(start : Number [, end : Number]) : String
```

参数

start

必选。指向字符串指定部分的开头的索引。

end

可选项。指向字符串指定部分的结尾的索引。

备注

slice 方法返回一个包含字符串指定部分的 **String** 对象。

slice 方法一直复制到 *end* 所指示的元素，但是不包括该元素。如果 *start* 为负，则将其视为 *length* + *start*，此处 *length* 为字符串的长度。如果 *end* 为负，则将其视为 *length* + *end*，此处 *length* 为字符串的长度。如果省略 *end*，则将一直提取到字符串的结尾。如果 *end* 出现在 *start* 之前，则不会将任何字符复制到新字符串中。

示例

在下面的示例中，对 **slice** 方法的第一次调用返回包含 `str` 的前五个字符的字符串。对 **slice** 方法的第二次调用返回包含 `str` 的后五个字符的字符串。

```
var str = "hello world";
var firstfive = str.slice(0,5); // Contains "hello".
var lastfive = str.slice(-5);   // Contains "world".
```

要求

版本 3

应用于：

String 对象

请参见

参考

Array 对象
slice 方法（数组）

small 方法

返回 **String** 对象中文本周围带有 HTML <SMALL> 标记的字符串。

```
function small() : String
```

备注

未进行任何检查来查看此标记是否已应用于该字符串。

示例

下面的示例阐释了 small 方法的用法：

```
var strVariable = "This is a string";
strVariable = strVariable.small();
```

最后一个语句后面的 `strVariable` 的值为：

```
<SMALL>This is a string</SMALL>
```

要求

版本 1

应用于：

String 对象

请参见

参考

big 方法

sort 方法

返回一个元素已经进行了排序的 **Array** 对象。

```
function sort(sortFunction : Function ) : Array
```

参数
sortFunction

可选项。用来确定元素顺序的函数的名称。

备注

sort 方法对 **Array** 对象进行排序；在执行过程中不会创建新 **Array** 对象。

如果在 sortfunction 参数中提供了一个函数，那么该函数必须返回下列值之一：

- 如果所传递的第一个参数小于第二个参数，则返回负值。
- 如果两个参数相等，则返回零。
- 如果第一个参数大于第二个参数，则返回正值。

如果省略 *sortFunction* 参数，元素将按 ASCII 字符顺序的升序进行排列。

示例

下面的示例阐释了 sort 方法的用法。

```
function SortDemo(){
   var a, l;                        //Declare variables.
   a = new Array("X" ,"y" ,"d", "Z", "v","m","r");
   l = a.sort();                    //Sort the array.
   return(l);                       //Return sorted array.
}
```

要求

版本 2

应用于：

Array 对象

请参见
其他资源
对象 (JScript)

splice 方法

从一个数组中移除元素，如有必要，在所移除元素的位置上插入新元素，并返回所移除的元素。返回从数组中移除的元素。

```
function splice(start : Number, deleteCount : Number [, item1 : Object [, ... [, itemN : Ob
ject]]]]) : Array
```

参数

start

必选。数组中移除元素操作的起点，从 0 开始。

deleteCount

必选。将移除的元素数。

item1, ..., itemN

可选项。插入数组中代替已移除元素的元素。

备注

splice 方法通过移除从 *start* 位置开始的指定个数的元素并插入新元素来修改数组。返回值是一个由所移除的元素组成的新 **array** 对象。

要求

版本 5.5

应用于:

Array 对象

请参见

参考

slice 方法（数组）

split 方法

返回一个字符串拆分为若干子字符串时所产生的字符串数组。

```
function split([ separator : { String | RegExp } [, limit : Number]]) : Array
```

参数

separator

可选项。字符串或 **Regular Expression** 对象的实例，它标识用于分隔字符串的一个或多个字符。如果忽略该参数，将返回包含整个字符串的单元素数组。

limit

可选项。一个用于限制数组中返回的元素数的值。

备注

split 方法的结果是在字符串中出现 *separator* 的每个位置分隔字符串后产生的字符串数组。*separator* 将不作为任何数组元素的一部分返回。

示例

下面的示例阐释了 split 方法的用法。

```
function SplitDemo(){
   var s, ss;
   var s = "The rain in Spain falls mainly in the plain.";
   // Split at each space character.
   ss = s.split(" ");
   return(ss);
}
```

要求

版本 3

应用于:

String 对象

请参见

参考

concat 方法（字符串）

RegExp 对象

正则表达式对象

概念

正则表达式语法

sqrt 方法

返回一个数的平方根。

```
function sqrt(number : Number) : Number
```

参数
number

必选。一个用于计算其平方根的数值表达式。

备注

如果 *number* 为负，则返回值为 **NaN**。

要求

版本 1

应用于：

Math 对象

请参见
参考
SQRT1_2 属性
SQRT2 属性

JScript 8.0

strike 方法

返回 **String** 对象中文本周围带有 HTML <STRIKE> 标记的字符串。

```
function strike() : String
```

备注

未进行任何检查来查看此标记是否已应用于该字符串。

示例

下面的示例阐释了 **strike** 方法的用法：

```
var strVariable = "This is a string object";
strVariable = strVariable.strike();
```

最后一个语句后面的 `strVariable` 的值为：

```
<STRIKE>This is a string object</STRIKE>
```

要求

版本 1

应用于：

String 对象

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

sub 方法

返回 **String** 对象中文本周围带有 HTML <SUB> 标记的字符串。

```
function sub() : String
```

备注

未进行任何检查来查看此标记是否已应用于该字符串。

示例

下面的示例阐释了 sub 方法的用法：

```
var strVariable = "This is a string object";
strVariable = strVariable.sub();
```

最后一个语句后面的 `strVariable` 的值为：

```
<SUB>This is a string object</SUB>
```

要求

版本 1

应用于：

String 对象

请参见

参考

sup 方法

substr 方法

返回一个从指定位置开始，并具有指定长度的子字符串。

```
function substr(start : Number [, length : Number]) : String
```

参数

start

必选。所需的子字符串的起始位置。字符串中第一个字符的索引为 0。

length

可选项。返回的子字符串中包含的字符数。

备注

如果 length 为 0 或负数，将返回一个空字符串。如果没有指定该参数，则子字符串将延续到字符串的结尾。

示例

下面的示例阐释了 **substr** 方法的用法。

```
function SubstrDemo(){
   var s, ss;                //Declare variables.
   var s = "The rain in Spain falls mainly in the plain.";
   ss = s.substr(12, 5);  //Get substring.
   return(ss);               // Returns "Spain".
}
```

要求

版本 3

应用于:

String 对象

请参见

参考

substring 方法

substring 方法

返回位于 **String** 对象中的指定位置的子字符串。

```
function substring(start : Number, end : Number) : String
```

参数

start

必选。从 0 开始的索引整数，指示子字符串的起始位置。

end

必选。从 0 开始的索引整数，指示子字符串的结束位置。

备注

substring 方法将返回一个字符串，该字符串包含从 *start* 直到 *end*（不包含 *end*）的子字符串。

substring 方法使用 *start* 和 *end* 两者中的较小值作为子字符串的起始点。例如，strvar.substring(0, 3) 和 strvar.substring(3, 0) 将返回相同的子字符串。

如果 *start* 或 *end* 为 **NaN** 或负数，那么它将被替换为 0。

子字符串的长度等于 *start* 和 *end* 之差的绝对值。例如，在 strvar.substring(0, 3) 和 strvar.substring(3, 0) 中，返回的子字符串的长度为 3。

示例

下面的示例阐释了 substring 方法的用法。

```
function SubstringDemo(){
   var ss;                          //Declare variables.
   var s = "The rain in Spain falls mainly in the plain..";
   ss = s.substring(12, 17);   //Get substring.
   return(ss);                      //Return substring.
}
```

要求

版本 1

应用于:

String 对象

请参见

参考

substr 方法

sup 方法

返回 **String** 对象中文本周围带有 HTML <SUP> 标记的字符串。

```
function sup() : String
```

备注

未进行任何检查来查看此标记是否已应用于该字符串。

示例

下面的示例阐释了 **sup** 方法的用法。

```
var strVariable = "This is a string object";
strVariable = strVariable.sup();
```

最后一个语句后面的 `strVariable` 的值为:

```
<SUP>This is a string object</SUP>
```

要求

版本 1

应用于:

String 对象

请参见

参考

sub 方法

方法 (T-Z)

方法是作为对象成员的函数。以下方法的名称以字母 t 到 z 开头。

本节内容

tan 方法

返回一个数的正切。

test 方法

返回一个布尔值，它指示在所搜索的字符串中是否存在一种模式。

toArray 方法

返回一个从 VBArray 转换来的标准 JScript 数组。

toDateString 方法

以字符串值的形式返回一个日期。

toExponential 方法

返回一个字符串，其中包含一个以指数记数法表示的数字。

toFixed 方法

返回一个字符串，表示一个以定点表示法表示的数字。

toGMTString 方法

返回使用格林尼治标准时间 (GMT) 转换为字符串的日期。

toLocaleDateString 方法

将一个日期以字符串值的形式返回，该字符串应适合于宿主环境的当前区域设置。

toLocaleLowerCase 方法

返回一个字符串，其中所有的字母都被转换为小写，同时考虑到宿主环境的当前区域设置。

toLocaleString 方法

返回使用当前区域设置转换为字符串的日期。

toLocaleTimeString 方法

以字符串值的形式返回一个时间，此字符串值应与宿主环境的当前区域设置相适应。

toLocaleUpperCase 方法

返回一个字符串，其中所有字母都被转换为大写，同时考虑宿主环境的当前区域设置。

toLowerCase 方法

返回一个字符串，该字符串中的所有字母都被转换为小写字母。

toPrecision 方法

返回一个字符串，其中包含一个以指数记数法或定点记数法表示的，具有指定数字位数的数字。

toString 方法

返回表示对象的字符串。

toTimeString 方法

以字符串值形式返回时间。

toUpperCase 方法

返回一个字符串，该字符串中的所有字母都被转换为大写字母。

toUTCString 方法

返回使用协调通用时间 (UTC) 转换为字符串的日期。

ubound 方法

返回在 VBArray 的指定维中所使用的最大索引值。

unescape 方法

解码用 **escape** 方法进行编码的 **String** 对象。

unshift 方法

将指定的元素插入数组开头并返回该数组。

UTC 方法

返回协调通用时间 (UTC)(或 GMT)1970 年 1 月 1 日午夜与所提供的日期之间相差的毫秒数。

valueOf 方法

返回指定对象的原始值。

相关章节

JScript 参考

列出"JScript 语言参考"所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

方法

按字母分类列出 JScript 中的可用方法，并列出指向每个方法类别的链接。

对象

解释有关 JScript 中的对象的概念，并说明对象与属性和方法的关系，并提供指向特定主题的链接，这些主题将详细介绍 JScript 所支持的对象。

JScript 8.0

tan 方法

返回一个数的正切。

```
function tan(number : Number) : Number
```

参数
number

必选。一个用于计算其正切的数值表达式。

备注

返回值为 *number* 的正切。

要求

版本 1

应用于:

Math 对象

请参见
参考
acos 方法
asin 方法
atan 方法
atan2 方法
cos 方法
sin 方法

test 方法

返回一个布尔值，它指示在所搜索的字符串中是否存在正则表达式模式。

```
function test(str : String) : Boolean
```

参数

str

必选。将被执行搜索的字符串。

备注

test 方法检查字符串中是否存在某种模式，如果存在，则返回 true，否则将返回 false。如果找到匹配项，则会更新全局 **RegExp** 对象的属性以反映匹配的结果。

如果为正则表达式设置了全局标志，则 **test** 从 **lastIndex** 值指示的位置开始搜索字符串。如果没有设置全局标志，则 **test** 忽略 **lastIndex** 的值，从字符串的起始位置开始搜索。

示例

下面的示例阐释了 test 方法的用法。若要使用此示例，请给函数传递一个正则表达式模式和一个字符串。该函数会在字符串中检测正则表达式模式的匹配项并返回一个指示此搜索结果的字符串：

```
function TestDemo(re, s){
   var s1;                          //Declare variable.
   // Test string for existence of regular expression.
   if (re.test(s))                  //Test for existence.
      s1 = " contains ";            //s contains pattern.
   else
      s1 = " does not contain ";    //s does not contain pattern.
   return("'" + s + "'" + s1 + "'"+ re.source + "'"); //Return string.
}
```

要求

版本 3

应用于：

正则表达式对象

请参见

参考

RegExp 对象

概念

正则表达式语法

toArray 方法

返回一个从 VBArray 转换来的标准 JScript 数组。

```
function toArray() : Array
```

备注

该转换将多维 VBArray 转换成一个一维 JScript 数组。**toArray** 方法将每个连续的维度追加到上一个维度的末尾。例如，一个三维且每一维有三个元素的 VBArray 将被转换为如下所示的 JScript 数组：

假定 VBArray 包含：(1, 2, 3)、(4, 5, 6) 和 (7, 8, 9)。则转换之后，JScript 数组包含：1、2、3、4、5、6、7、8 和 9。

目前还没有将 JScript 数组转换为 VBArray 的方法。

示例

下面的示例由三部分组成。第一部分是创建 Visual Basic 安全数组的 VBScript 代码。第二部分是将 Visual Basic 安全数组转换为 JScript 数组的 JScript 代码。这两部分都位于 HTML 页的 <HEAD> 区域。第三部分是位于 <BODY> 区域内用于运行其他两个部分的 JScript 代码。

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(j, i) = k
         document.writeln(k)
         k = k + 1
      Next
      document.writeln("<BR>")
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vbarray)
{
   var a = new VBArray(vbarray);
   var b = a.toArray();
   var i;
   for (i = 0; i < 9; i++)
   {
      document.writeln(b[i]);
   }
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT LANGUAGE="JScript">
<!--
   VBArrayTest(CreateVBArray());
-->
</SCRIPT>
</BODY>
```

要求

版本 3

应用于:

VBArray 对象

请参见

参考

dimensions 方法
getItem 方法
lbound 方法
ubound 方法

toDateString 方法

以字符串值的形式返回一个日期。

```
function toDateString() : String
```

备注

toDateString 方法返回一个包含日期的字符串值，该日期位于当前时区中，以一种方便、易读的格式表示。

要求

版本 5.5

应用于：

Date 对象

请参见

参考

toTimeString 方法
toLocaleDateString 方法

toExponential 方法

返回一个字符串，其中包含一个以指数记数法表示的数字。

```
function toExponential( [fractionDigits : Number] ) : String
```

参数
fractionDigits

可选项。小数点后数字的位数。值必须介于 0 -20 之间（含 1 和 21）。

备注

toExponential 方法返回一个字符串，该字符串表示一个指数记数法的数字。该字符串中有效数小数点之前包含一位数字，而小数点之后可以包含 *fractionDigits* 位数字。

如果没有提供 *fractionDigits*，则 **toExponential** 方法将返回足够多位数字，以便唯一指定该数字。

要求

版本 5.5

应用于：

Number 对象

请参见
参考
toFixed 方法
toPrecision 方法

toFixed 方法

返回一个字符串，表示一个以定点表示法表示的数字。

```
function toFixed( [fractionDigits : Number] ) : String
```

参数
fractionDigits

可选项。小数点后数字的位数。值必须介于 0 -20 之间（含 1 和 21）。

备注

toFixed 方法返回一个字符串，该字符串表示一个以定点表示法表示的数字。该字符串中有效数的小数点之前包含一位数字，而小数点后必须包含 *fractionDigits* 位数字。

如果没有提供 *fractionDigits* 参数或该参数是 **undefined** 的，则 **toFixed** 方法假定该参数值为 0。

要求

版本 5.5

应用于：

Number 对象

请参见
参考
toExponential 方法
toPrecision 方法

toGMTString 方法

返回使用格林尼治标准时间 (GMT) 转换为字符串的日期。

```
function toGMTString() : String
```

备注

toGMTString 方法已经过时，之所以仍然提供这个方法，只是为了向后兼容。推荐改用 **toUTCString** 方法。

toGMTString 方法返回一个 **String** 对象，此对象中包含 GMT 惯例格式的日期。返回值的格式是这样的："05 Jan 1996 00:00:00 GMT"。

要求

版本 1

应用于：

Date 对象

请参见

参考

toUTCString 方法

toLocaleDateString 方法

将一个日期以字符串值的形式返回，该字符串应适合于宿主环境的当前区域设置。

```
function toLocaleDateString() : String
```

备注

toLocaleDateString 方法返回一个包含当前时区日期的字符串值，以一种易读的格式表示。该日期的格式为宿主环境的当前区域设置中的默认格式。因为此方法的返回值将随计算机的不同而不同，所以脚本撰写过程中不能依赖该返回值。**toLocalDateString** 方法应仅用于格式显示，绝不要作为计算的一部分。

要求

版本 5.5

应用于：

Date 对象

请参见

参考

toDateString 方法

toLocaleTimeString 方法

toLocaleLowerCase 方法

返回一个字符串，其中所有的字母都被转换为小写，同时考虑到宿主环境的当前区域设置。

```
function tolocaleLowerCase() : String
```

备注

toLocaleLowerCase 方法转换字符串中的字符，同时考虑到宿主环境的当前区域设置。在大多数情况下，其结果与使用 **toLowerCase** 方法所得到的结果相同。如果语言规则与常规的 Unicode 大小写映射冲突，则结果将会不同。

要求

版本 5.5

应用于：

String 对象

请参见

参考

toLocaleUpperCase 方法
toLowerCase 方法

toLocaleString 方法

以字符串值的形式返回一个值，该值适合于宿主环境的当前区域设置。

```
function toLocaleString() : String
```

备注

对于 **Array** 对象，将数组元素转换为字符串并将这些字符串连接起来后返回，每个字符串由宿主环境的当前区域设置所指定的列表分隔符分隔。

对于 **Date** 对象，**toLocaleString** 方法返回一个 **String** 对象，此对象包含以当前区域设置的长默认格式编写的日期。

- 对于公元 1601 和 9999 之间的日期，其格式将根据用户在"控制面板"中选择的"区域设置"确定。
- 对于此范围之外的日期，将使用 **toString** 方法的默认格式。

对于 **Number** 对象，**toLocaleString** 产生一个表示 **Number** 值的字符串值，其格式对应于宿主环境的当前区域设置。

对于 **Object** 对象，使用 **ToLocaleString** 为所有对象提供一个通用的 **toLocaleString** 功能，即使这些对象可能并不使用此功能。

注意

toLocaleString 应当仅用于向用户显示结果；决不可将它用作脚本中计算的基础，因为返回的结果因计算机而异。

示例

下面的客户端示例阐释了使用 **Array**、**Date** 和 **Number** 对象的 **toLocaleString** 方法的用法。

```
function toLocaleStringArray() {
   // Declare variables.
   var myArray = new Array(6);
   var i;
   // Initialize string.
   var s = "The array contains: ";
   // Populate array with values.
   for(i = 0;i < 7; i++)
   {
      // Make value same as index.
      myArray[i] = i;
   }
   s += myArray.toLocaleString();
   return(s);
}
function toLocaleStringDate() {
   // Declare variables.
   var d = new Date();
   var s = "Current date setting is ";
   // Convert to current locale.
   s += d.toLocaleString();
   return(s);
}
function toLocaleStringNumber() {
   var n = Math.PI;
   var s = "The value of Pi is: ";
   s+= n.toLocaleString();
   return(s);
}
```

要求

版本 1

应用于:

Array 对象 | Date 对象 | Number 对象 | Object 对象

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

toLocaleTimeString 方法

以字符串值的形式返回一个时间，此字符串值应与宿主环境的当前区域设置相适应。

```
function toLocaleTimeString() : String
```

备注

toLocaleTimeString 方法返回一个包含当前时区时间的字符串值，以一种易读的格式表示。该时间的格式为宿主环境的当前区域设置中的默认格式。因为此方法的返回值将随计算机的不同而不同，所以脚本撰写过程中不能依赖该返回值。**toLocalTimeString** 方法仅应用于格式显示，绝不要作为计算的一部分。

要求

版本 5.5

应用于：

Date 对象

请参见

参考

toTimeString 方法

toLocaleDateString 方法

toLocaleUpperCase 方法

返回一个字符串，其中所有字母都被转换为大写，同时考虑宿主环境的当前区域设置。

```
function tolocaleUpperCase() : String
```

备注

toLocaleUpperCase 方法转换字符串中的字符，同时考虑到宿主环境的当前区域设置。在大多数情况下，其结果与使用 **toUpperCase** 方法所得到的结果相同。如果语言规则与常规的 Unicode 大小写映射冲突，则结果将会不同。

要求

版本 5.5

应用于：

String 对象

请参见

参考

toLocaleLowerCase 方法

toUpperCase 方法

toLowerCase 方法

返回一个字符串，该字符串中的所有字母都被转换为小写字母。

```
function toLowerCase() : String
```

备注

toLowerCase 方法对非字母字符无效。

示例

下面的示例说明了 **toLowerCase** 方法的效果：

```
var strVariable = "This is a STRING object";
strVariable = strVariable.toLowerCase();
```

最后一个语句后面的 `strVariable` 的值为：

```
this is a string object
```

要求

版本 1

应用于：

String 对象

请参见

参考

toUpperCase 方法

JScript 8.0

toPrecision 方法

返回一个字符串，其中包含一个以指数记数法或定点记数法表示的，具有指定数字位数的数字。

```
function toPrecision ( [precision : Number] ) : String
```

参数
precision

可选项。有效位数。值必须介于 1 - 21 之间（含 1 和 21）。

备注

对于以指数记数法表示的数字，将返回小数点后的 *precision - 1* 位数字。对于以定点记数法表示的数字，将返回 *precision* 位有效位数。

如果没有提供参数 precision 或者它为 **undefined**，则将转而调用 **toString** 方法。

要求

版本 5.5

应用于：

Number 对象

请参见
参考
toFixed 方法
toExponential 方法

toString 方法

返回表示对象的字符串。

```
function toString( [radix : Number] ) : String
```

参数
radix

可选项。为将数字值转换为字符串指定一个基数。此值仅用于数字。

备注

toString 方法是一个所有内置的 JScript 对象的成员。它的行为取决于对象的类型：

对象	行为
Array	将 **Array** 的元素转换为字符串。结果字符串被连接起来，用逗号分隔。
Boolean	如果布尔值是 true，则返回"真"。否则，返回"假"。
Date	返回日期的文本表示。
Error	返回一个包含相关错误信息的字符串。
Function	返回如下格式的字符串，其中 functionname 是一个函数的名称，此函数的 toString 方法被调用： "function functionname() { [native code] }"
Number	返回数字的文字表示。
String	返回 **String** 对象的值。
Default	返回"[object objectname]"，其中 objectname 是对象类型的名称。

示例

下面的示例阐释了带有 radix 参数的 **toString** 方法的用法。下文显示的函数的返回值是一个基数转换表。

```
function CreateRadixTable (){
   var s, s1, s2, s3, x;                    //Declare variables.
   s = "Hex    Dec   Bin \n";               //Create table heading.
   for (x = 0; x < 16; x++)                 //Establish size of table
   {                                        // in terms of number of
      switch(x)                             // values shown.
      {                                     //Set intercolumn spacing.
         case 0 :
            s1 = "      ";
            s2 = "    ";
            s3 = "   ";
            break;
         case 1 :
            s1 = "      ";
            s2 = "    ";
            s3 = "   ";
            break;
         case 2 :
            s3 = "  ";
            break;
         case 3 :
            s3 = "  ";
```

```
            break;
         case 4 :
            s3 = " ";
            break;
         case 5 :
            s3 = " ";
            break;
         case 6 :
            s3 = " ";
            break;
         case 7 :
            s3 = " ";
            break;
         case 8 :
            s3 = "" ;
            break;
         case 9 :
            s3 = "";
            break;
         default:
            s1 = "     ";
            s2 = "";
            s3 = "    ";
      }                                    //Convert to hex, decimal & binary.
      s += " " + x.toString(16) + s1 + x.toString(10)
      s +=  s2 + s3 + x.toString(2)+ "\n";

   }
   return(s);                              //Return entire radix table.
}
```

要求

版本 2

应用于:

Array 对象 | Boolean 对象 | Date 对象 | 错误对象 | Function 对象 | Number 对象 | Object 对象 | String 对象

请参见

参考

function 语句

toTimeString 方法

以字符串值形式返回时间。

```
function toTimeString() : String
```

备注

toTimeString 方法返回一个包含当前时区时间的字符串值，以一种方便、易读的格式表示。

要求

版本 5.5

应用于：

Date 对象

请参见

参考

toDateString 方法
toLocaleTimeString 方法

toUpperCase 方法

返回一个字符串，该字符串中的所有字母都被转换为大写字母。

```
function toUpperCase() : String
```

备注

toUpperCase 方法对非字母字符无效。

示例

下面的示例说明了 **toUpperCase** 方法的效果：

```
var strVariable = "This is a STRING object";
strVariable = strVariable.toUpperCase();
```

最后一个语句后面的 `strVariable` 的值为：

```
THIS IS A STRING OBJECT
```

要求

版本 1

String 对象

请参见

参考

toLowerCase 方法

toUTCString 方法

返回使用协调通用时间 (UTC) 转换为字符串的日期。

```
function toUTCString() : String
```

备注

toUTCString 方法返回一个 **String** 对象，此对象中包含了 UTC 惯例格式的日期，以一种简便、易读的形式表示。

示例

下面的示例阐释了 **toUTCString** 方法的用法。

```
function toUTCStrDemo(){
   var d, s;                    //Declare variables.
   d = new Date();              //Create Date object.
   s = "Current setting is ";
   s += d.toUTCString();        //Convert to UTC string.
   return(s);                   //Return UTC string.
}
```

要求

版本 3

应用于：

Date 对象

请参见

参考

toGMTString 方法

ubound 方法

返回在 VBArray 的指定维中所使用的最大索引值。

```
function ubound( [dimension : Number] ) : Number
```

参数
dimension

可选项。希望获知其上限索引的 VBArray 的维度。如果忽略此参数，**ubound** 将该参数视为 1 进行处理。

备注

如果 VBArray 为空，则 **ubound** 方法将返回未定义。如果 *dimension* 大于 VBArray 的维度数或为负，该方法将生成"下标超出范围"错误。

示例

下面的示例由三部分组成。第一部分是创建 Visual Basic 安全数组的 VBScript 代码。第二部分是 JScript 代码，该代码确定安全数组中的维度数和每一维的上限。这两部分都位于 HTML 页的 <HEAD> 区域。第三部分是位于 <BODY> 区域内用于运行其他两个部分的 JScript 代码。

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(j, i) = k
         k = k + 1
      Next
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vba)
{
   var i, s;
   var a = new VBArray(vba);
   for (i = 1; i <= a.dimensions(); i++)
   {
      s = "The upper bound of dimension ";
      s += i + " is ";
      s += a.ubound(i)+ ".<BR>";
      return(s);
   }
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT language="jscript">
   document.write(VBArrayTest(CreateVBArray()));
</SCRIPT>
</BODY>
```

要求

版本 3

应用于:

VBArray 对象

请参见

参考

dimensions 方法
getItem 方法
lbound 方法
toArray 方法

unescape 方法

从用 **escape** 方法编码的 **String** 对象中返回已解码的字符串。

```
function unescape(charString : String) : String
```

参数
charString

必选。要解码的 **String** 对象或文本。

备注

unescape 方法返回一个包含 *charString* 内容的字符串值。所有以 %xx 十六进制形式编码的字符都用 ASCII 字符集当中等效的字符代替。

以 %uxxxx 格式(Unicode 字符)编码的字符用十六进制编码 xxxx 的 Unicode 字符代替。

注意

unescape 方法不应用于解码"统一资源标识符"(URI)。请改用 **decodeURI** 和 **decodeURIComponent** 方法。

要求

版本 1

应用于:

Global 对象

请参见
参考
decodeURI 方法
decodeURIComponent 方法
escape 方法
String 对象

unshift 方法

将指定的元素插到数组的开头。

```
function unshift([item1 : Object [, ... [, itemN : Object]]]) : Array
```

参数

item1, ..., itemN

可选项。用于插到 **Array** 开头的元素。

备注

unshift 方法将这些元素插入到一个数组的开头，以便它们按其在参数表中的次序排列。

要求

版本 5.5

应用于：

Array 对象

请参见

参考

shift 方法

UTC 方法

返回协调通用时间 (UTC)(或 GMT)1970 年 1 月 1 日午夜与所提供的日期之间相差的毫秒数。

```
function UTC(year : Number , month : Number , day : Number [, hours : Number [, minutes : N
umber [, seconds : Number [,ms : Number]]]]) : Number
```

参数

year

必选。为了确保跨世纪日期的精确性，需要指定完整的年份。如果 year 处于 0 到 99 之间，那么 year 就被假定为 1900 + year。

month

必选。月份，用从 0 到 11 的整数表示(1 月至 12 月)。

day

必选。日期，用从 1 到 31 的整数表示。

hours

可选项。如果提供了 minutes 则必须给出该参数。一个指定小时的，从 0 到 23 的整数(午夜到 11pm)。

minutes

可选项。如果提供了 seconds 则必须给出该参数。一个指定分钟的，从 0 到 59 的整数。

seconds

可选项。如果提供了 milliseconds 则必须给出该参数。一个指定秒的，从 0 到 59 的整数。

ms

可选项。一个指定毫秒的，从 0 到 999 的整数。

备注

UTC 方法返回从 UTC 1970 年 1 月 1 日午夜到所提供日期之间的毫秒数。此返回值可用在 **setTime** 方法和 **Date** 对象构造函数中。如果参数值大于其范围或为负数，则其他存储的值都将得到相应的修改。例如，如果指定 150 秒，则 JScript 将该数字重新定义为 2 分 30 秒。

UTC 方法和接受日期的 **Date** 对象构造函数之间的差别在于：**UTC** 方法采用 UTC，而 **Date** 对象构造函数采用当地时间。

UTC 方法是一个静态方法。因此，在使用 **Date** 对象之前无需创建它。

注意

如果 *year* 介于 0 到 99 之间，则年份为 *1900 + year*。

示例

下面的示例阐释了 **UTC** 方法的用法。

```
function DaysBetweenDateAndNow(yr, mo, dy){
   var d, r, t1, t2, t3;            //Declare variables.
   var MinMilli = 1000 * 60         //Initialize variables.
   var HrMilli = MinMilli * 60
   var DyMilli = HrMilli * 24
   t1 = Date.UTC(yr, mo - 1, dy)    //Get milliseconds since 1/1/1970.
   d = new Date();                  //Create Date object.
   t2 = d.getTime();                //Get current time.
   if (t2 >= t1)
      t3 = t2 - t1;
   else
      t3 = t1 - t2;
```

```
    r = Math.round(t3 / DyMilli);
    return(r);                          //Return difference.
}
```

要求

版本 1

应用于:

Date 对象

请参见

参考

setTime 方法

JScript 8.0

valueOf 方法

返回指定对象的原始值。

```
function valueOf() : Object
```

备注

每个内部 JScript 对象的 **valueOf** 方法被以不同方式定义。

对象	返回值
Array	数组实例。
Boolean	布尔值。
Date	以毫秒数存储的时间值，从 UTC 1970 年 1 月 1 日午夜开始计算。
Function	函数本身。
Number	数字值。
Object	对象本身。这是默认设置。
String	字符串值。

Math 和 **Error** 对象都没有 **valueOf** 方法。

要求

版本 2

应用于:

Array 对象 | Boolean 对象 | Date 对象 | Function 对象 | Number 对象 | Object 对象 | String 对象

请参见

参考

toString 方法

修饰符

JScript 修饰符用于影响类、接口、类或接口的成员的行为和可见性。修饰符可以在定义类和接口时使用，但它们并不是必需的。

本节内容

abstract 修饰符

允许定义类和类成员但不允许给定实现的继承修饰符。

expando 修饰符

将类标记为可动态扩展或将方法标记为 expando 对象构造函数的兼容性修饰符。

final 修饰符

防止类扩展或防止方法或属性重写的继承修饰符。

hide 修饰符

防止方法或属性重写基类中方法或属性的版本安全修饰符。

internal 修饰符

使类、接口或成员仅对于当前包可见的可见性修饰符。

override 修饰符

显式重写基类中方法的版本安全修饰符。

private 修饰符

使类成员仅对于同一类的成员可见的可见性修饰符。

protected 修饰符

使类或接口的成员仅对于当前类或接口和当前类的派生类可见的可见性修饰符。

public 修饰符

使类或接口的成员对于可访问该类或接口的所有代码都可见的可见性修饰符。

static 修饰符

将类成员标记为属于该类本身的修饰符。

相关章节

JScript 修饰符

从概念上概述 JScript 修饰符的用途和用法。

abstract 修饰符

声明类必须扩展或者方法或属性的实现必须由派生类来提供。

```
abstract statement
```

参数
statement

必选。类、方法或属性定义。

备注

abstract 修饰符用于类中不具有实现的方法或属性或者用于包含这些方法的类。具有抽象成员的类不能使用 **new** 运算符来实例化。您可以从抽象基类派生抽象和非抽象的类。

类中的方法和属性以及类可以使用 **abstract** 修饰符来标记。如果一个类包含任何 **abstract** 成员，则必须标记为 **abstract**。接口和接口的成员为隐式抽象，它们不能采用 **abstract** 修饰符。字段不能为 **abstract**。

不能将 **abstract** 修饰符与其他继承修饰符 (**final**) 组合。默认情况下，类成员既不是 **abstract** 也不是 **final**。继承修饰符不能与 **static** 修饰符组合。

示例

下面的示例阐释 **abstract** 修饰符的用法。

```
// CAnimal is an abstract base class.
abstract class CAnimal {
   abstract function printQualities();
}
// CDog and CKangaroo are derived classes of CAnimal.
class CDog extends CAnimal {
   function printQualities() {
      print("A dog has four legs.");
   }
}
class CKangaroo extends CAnimal {
   function printQualities() {
      print("A kangaroo has a pouch.");
   }
}

// Define animal of type CAnimal.
var animal : CAnimal;

animal = new CDog;
// animal uses printQualities from CDog.
animal.printQualities();

animal = new CKangaroo;
// animal uses printQualities from CKangaroo.
animal.printQualities();
```

该程序的输出为：

```
A dog has four legs.
A kangaroo has a pouch.
```

要求

.NET 版本

请参见
参考

final 修饰符
static 修饰符
var 语句
function 语句
class 语句
new 运算符

概念

变量和常数的范围

其他资源

修饰符

expando 修饰符

声明类的实例支持 expando 属性或者方法是 expando 对象构造函数。

```
expando statement
```

参数
statement

必选。类或方法定义。

备注

expando 修饰符用于将类标记为可动态扩展(支持 expando 属性的类)。**expando** 类实例的 expando 属性必须使用 **[]** 符号来进行访问;它们不能使用点运算符来进行访问。**expando** 修饰符还将方法标记为 **expando** 对象构造函数。

类和类中的方法可以使用 **expando** 修饰符来标记。字段、属性、接口和接口的成员不能采用 **expando** 修饰符。

expando 类具有一个名为 **Item** 的隐藏私有属性，它采用一个 **Object** 参数并返回 **Object**。不得使用 **expando** 类的这一签名来定义属性。

示例 1

下面的示例阐释 **expando** 修饰符在一个类上的用法。expando 类类似于 JScript **Object**，但是它们具有以下所示的一些差异。

```
          expando class CExpandoExample {
   var x : int = 10;
}

// New expando class-based object.
var testClass : CExpandoExample = new CExpandoExample;
// New JScript Object.
var testObject : Object = new Object;

// Add expando properties to both objects.
testClass["x"] = "ten";
testObject["x"] = "twelve";

// Access the field of the class-based object.
print(testClass.x);      // Prints 10.
// Access the expando property.
print(testClass["x"]);   // Prints ten.

// Access the property of the class-based object.
print(testObject.x);     // Prints twelve.
// Access the same property using the [] operator.
print(testObject["x"]);  // Prints twelve.
```

该代码的输出为

```
10
ten
twelve
twelve
```

示例 2

下面的示例阐释 **expando** 修饰符在一个方法上的用法。以通常方法调用 expando 方法时，它访问字段 *x*。当使用 **new** 运算符将该方法用作显式构造函数时，它将 expando 属性添加到新对象。

```
class CExpandoExample {
   var x : int;
```

```
    expando function constructor(val : int) {
       this.x = val;
       return "Method called as a function.";
    }
 }

 var test : CExpandoExample = new CExpandoExample;
 // Call the expando method as a function.
 var str = test.constructor(123);
 print(str);        // The return value is a string.
 print(test.x);     // The value of x has changed to 123.

 // Call the expando method as a constructor.
 var obj = new test.constructor(456);
 // The return value is an object, not a string.
 print(obj.x);      // The x property of the new object is 456.
 print(test.x);     // The x property of the original object is still 123.
```

该代码的输出为

```
Method called as a function.
123
456
123
```

要求

.NET 版本

请参见

参考

static 修饰符

var 语句

function 语句

class 语句

概念

变量和常数的范围

类型批注

其他资源

修饰符

final 修饰符

声明类不能扩展或者方法或属性不能重写。

```
final statement
```

参数
statement

必选。类、方法或属性定义。

备注

final 修饰符用于指定类不能扩展或者方法或属性不能重写。它将防止其他类通过重写重要的函数来更改该类的行为。带有 **final** 修饰符的方法可以由派生类中的方法来隐藏或重载。

类中的方法和属性以及类可以使用 **final** 修饰符来标记。接口、字段和接口的成员不能采用 **final** 修饰符。

不能将 **final** 修饰符与其他继承修饰符 (**abstract**) 组合。默认情况下，类成员既不是 **abstract** 也不是 **final**。继承修饰符不能与 **static** 修饰符组合。

示例

下面的示例阐释 **final** 修饰符的用法。**final** 修饰符防止基类方法被派生类中的方法重写。

```
class CBase {
   final function methodA() { print("Final methodA of CBase.") };
   function methodB() { print("Non-final methodB of CBase.") };
}

class CDerived extends CBase {
   function methodA() { print("methodA of CDerived.") };
   function methodB() { print("methodB of CDerived.") };
}

var baseInstance : CBase = new CDerived;
baseInstance.methodA();
baseInstance.methodB();
```

该程序的输出显示 final 方法未被重写：

```
Final methodA of CBase.
methodB of CDerived.
```

要求

.NET 版本

请参见
参考
abstract 修饰符
hide 修饰符
override 修饰符
var 语句
function 语句
class 语句
概念
变量和常数的范围
类型批注
其他资源
修饰符

hide 修饰符

声明方法或属性隐藏基类中的方法或属性。

```
hide statement
```

参数
statement

必选。方法或属性定义。

备注

hide 修饰符用于隐藏基类中方法的方法。当基类不包含具有相同签名的成员时，不得将 **hide** 修饰符用于方法。

类中的方法和属性可以使用 **hide** 修饰符来标记。类、字段、接口和接口的成员不能采用 **hide** 修饰符。

不能将 **hide** 修饰符与其他版本安全修饰符 (**override**) 组合。版本安全修饰符不能与 **static** 修饰符组合。默认情况下，除非基类方法具有 **final** 修饰符，否则方法将重写基类方法。除非为抽象的基方法提供了显式实现，否则不能隐藏 **abstract** 方法。在版本安全模式下运行时，每当重写基类方法时必须使用一个版本安全修饰符。

示例

下面的示例阐释 **hide** 修饰符的用法。派生类中用 **hide** 修饰符标记的方法不会重写基类方法。用 **override** 标记的方法会重写基类方法。

```
class CBase {
   function methodA() { print("methodA of CBase.") };
   function methodB() { print("methodB of CBase.") };
}

class CDerived extends CBase {
   hide function methodA() { print("Hiding methodA.") };
   override function methodB() { print("Overriding methodB.") };
}


var derivedInstance : CDerived = new CDerived;
derivedInstance.methodA();
derivedInstance.methodB();

var baseInstance : CBase = derivedInstance;
baseInstance.methodA();
baseInstance.methodB();
```

该程序的输出显示隐藏方法不重写基类方法。

```
Hiding methodA.
Overriding methodB.
methodA of CBase.
Overriding methodB.
```

要求

.NET 版本

请参见
参考
override 修饰符
static 修饰符
var 语句
function 语句
class 语句

JScript 8.0

internal 修饰符

声明类、类成员、接口或接口成员具有内部可见性。

```
internal statement
```

参数
statement

必选。类、接口或成员定义。

备注

internal 修饰符使类、接口或成员仅在当前包中可见。当前包之外的代码不能访问 **internal** 成员。

类和接口可以使用 **internal** 修饰符来标记。在全局范围内，**internal** 修饰符与 **public** 修饰符相同。类或接口的任何成员都可以使用 **internal** 修饰符来标记。

不能将 **internal** 修饰符与其他任何可见性修饰符（**public**、**private** 或 **protected**）组合。可见性修饰符相对于它们的定义范围。例如，不能公开访问 **internal** 类的 **public** 方法，但任何具有该类访问权的代码都可以访问该方法。

要求

.NET 版本

请参见
参考
public 修饰符
private 修饰符
protected 修饰符
var 语句
function 语句
class 语句
概念
变量和常数的范围
其他资源
修饰符

override 修饰符

声明方法或属性重写基类中的方法或属性。

```
override statement
```

参数
statement

必选。方法或属性定义。

备注

override 修饰符用于重写基类中方法的方法。当基类不包含具有相同签名的成员时，不得将 **override** 修饰符用于方法。

类中的方法和属性可以使用 **override** 修饰符来标记。类、字段、接口和接口的成员不能采用 **override** 修饰符。

不能将 **override** 修饰符与其他版本安全修饰符 (**hide**) 组合。版本安全修饰符不能与 **static** 修饰符组合。默认情况下，除非基类方法具有 **final** 修饰符，否则方法将重写基类方法。不能重写 **final** 方法。在版本安全模式下运行时，每当重写基类方法时必须使用一个版本安全修饰符。

示例

下面的示例阐释 **override** 修饰符的用法。派生类中用 **override** 修饰符标记的方法会重写基类方法。用 **hide** 修饰符标记的方法不会重写基类方法。

```
class CBase {
   function methodA() { print("methodA of CBase.") };
   function methodB() { print("methodB of CBase.") };
}

class CDerived extends CBase {
   hide function methodA() { print("Hiding methodA.") };
   override function methodB() { print("Overriding methodB.") };
}


var derivedInstance : CDerived = new CDerived;
derivedInstance.methodA();
derivedInstance.methodB();

var baseInstance : CBase = derivedInstance;
baseInstance.methodA();
baseInstance.methodB();
```

该程序的输出显示 **override** 方法重写基类方法。

```
Hiding methodA.
Overriding methodB.
methodA of CBase.
Overriding methodB.
```

要求

.NET 版本

请参见
参考
hide 修饰符
var 语句
function 语句
class 语句
概念

private 修饰符

声明类成员具有私有可见性。

```
private statement
```

参数
statement

必选。类成员定义。

备注

private 修饰符使一个类的成员仅在该类中可见。当前类（包括派生类）之外的代码不能访问 **private** 成员。

全局范围内的类和接口不能使用 **private** 修饰符来标记。类或接口（包括嵌套类和嵌套接口）的任何成员都可以使用 **private** 修饰符来标记。

不能将 **private** 修饰符与其他任何可见性修饰符（**public**、**protected** 或 **internal**）组合。

要求

.NET 版本

请参见
参考
public 修饰符
protected 修饰符
internal 修饰符
var 语句
function 语句
class 语句
概念
变量和常数的范围
其他资源
修饰符

protected 修饰符

声明类成员或接口成员具有受保护的可见性。

```
protected statement
```

参数
statement

必选。类成员或接口成员定义。

备注

protected 标识符使类或接口的成员仅在该类或接口以及当前类的所有派生类中可见。当前类之外的代码无法访问 **protected** 成员。

全局范围内的类和接口不能使用 **protected** 修饰符来标记。类或接口（包括嵌套类和嵌套接口）的任何成员都可以使用 **protected** 修饰符来标记。

不能将 **protected** 修饰符与其他任何可见性修饰符（**public**、**private** 或 **internal**）组合。

要求

.NET 版本

请参见
参考
public 修饰符
private 修饰符
internal 修饰符
var 语句
function 语句
class 语句
概念
变量和常数的范围
其他资源
修饰符

public 修饰符

声明类、接口或成员具有公共可见性。

```
public statement
```

参数
statement

必选。类、接口或成员定义。

备注

public 修饰符使类的成员对于可访问该类的所有代码都可见。

所有类和接口在默认情况下都为 **public**。类或接口的成员可以使用 **public** 修饰符来标记。

不能将 **public** 修饰符与其他任何可见性修饰符（**private**、**protected** 或 **internal**）组合。

要求

.NET 版本

请参见
参考
private 修饰符
protected 修饰符
internal 修饰符
var 语句
function 语句
class 语句
概念
变量和常数的范围
其他资源
修饰符

static 修饰符

声明类成员属于类，而不属于类的实例。

```
static statement
```

参数

statement

必选。类成员定义。

备注

static 修饰符指明成员属于类本身而不属于类的实例。即使创建了类的多个实例，给定应用程序中只存在 **static** 成员的一个副本。您只能通过对类的引用（而不是对实例的引用）来访问 **static** 成员。但是，在类成员声明中，可以通过 **this** 对象来访问 **static** 成员。

类的成员可以使用 **static** 修饰符来标记。类、接口和接口的成员不能采用 **static** 修饰符。

不能将 **static** 修饰符与任何继承修饰符（**abstract** 和 **final**）或版本安全修饰符（**hide** 和 **override**）组合。

不要将 **static** 修饰符同 **static** 语句混淆。**static** 修饰符表示属于类本身（而不属于任何类实例）的成员。

示例

下面的示例阐释 **static** 修饰符的用法。

```
class CTest {
   var nonstaticX : int;      // A non-static field belonging to a class instance.
   static var staticX : int;  // A static field belonging to the class.
}

// Initialize staticX. An instance of test is not needed.
CTest.staticX = 42;

// Create an instance of test class.
var a : CTest = new CTest;
a.nonstaticX = 5;
// The static field is not directly accessible from the class instance.

print(a.nonstaticX);
print(CTest.staticX);
```

该程序的输出为：

```
5
42
```

要求

.NET 版本

请参见

参考

expando 修饰符
var 语句
function 语句
class 语句
static 语句

概念

变量和常数的范围
类型批注

其他资源

修饰符

对象 (JScript)

JScript 对象是属性和方法的集合。以下各节链接到有关解释如何使用 JScript 对象的信息。

注意
JScript 运行库未被设计为线程安全的。因此，当 JScript 对象和方法在多线程应用程序中使用时，可能会具有不可预知的行为。

本节内容

ActiveXObject 对象

启用和返回对自动化对象的引用。

arguments 对象

提供对传递到当前函数的参数的访问。

Array 对象

支持创建任何数据类型的数组。

Boolean 对象

创建新的布尔值。

Date 对象

启用日期和时间的基本存储和检索。

Enumerator 对象

启用集合中项的枚举。

错误对象

此对象包含有关在运行 JScript 代码时所出现错误的信息。

Function 对象

创建新函数。

Global 对象

内部对象，其用途是将全局方法收集到一个对象中。

Math 对象

提供基本数学函数和常数的内部对象。

Number 对象

数字数据类型和数值常数占位符的对象表示形式。

Object 对象

提供所有 JScript 对象共有的功能。

RegExp 对象

存储有关正则表达式模式搜索的信息。

正则表达式对象

包含正则表达式模式。

String 对象

允许操作和格式化文本字符串以及确定和定位字符串中的子字符串。

VBArray 对象

提供对 Visual Basic 安全数组的访问。

相关章节

JScript 参考

列出“JScript 语言参考”所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

方法

按字母分类列出 JScript 中的可用方法，并列出指向每个方法类别的链接。

属性

提供一个 JScript 中可用的属性列表，并提供指向特定链接的主题，这些主题解释每个属性的正确语法的用法。

JScript 对象

解释有关 JScript 中的对象的概念，并说明对象与属性和方法的关系，并提供指向特定主题的链接，这些主题将详细介绍 JScript 所支持的对象。

ActiveXObject 对象

此对象提供自动化对象的接口。

```
function ActiveXObject(ProgID : String [, location : String])
```

参数

ProgID

必选。形式为"*serverName.typeName*"的字符串，其中 *serverName* 是提供对象的应用程序的名称，*typeName* 是要创建的对象的类型或类。

location

可选项。要在其中创建对象的网络访问器的名称。

备注

通常，自动化服务器会提供至少一种对象。例如，字处理应用程序可能会提供应用程序对象、文档对象和工具栏对象。

以下代码通过调用 **ActiveXObject** 对象构造函数来启动应用程序（在这种情况下为 Microsoft Excel 工作表）。**ActiveXObject** 允许您在代码中引用应用程序。使用下面的示例，您可以使用对象变量 `ExcelSheet` 和其他 Excel 对象（包括应用程序对象和 ActiveSheet.Cells 集合）来访问新对象的属性和方法。

```
// Declare the variables
var Excel, Book;

// Create the Excel application object.
Excel = new ActiveXObject("Excel.Application");

// Make Excel visible.
Excel.Visible = true;

// Create a new work book.
Book = Excel.Workbooks.Add()

// Place some text in the first cell of the sheet.
Book.ActiveSheet.Cells(1,1).Value = "This is column A, row 1";

// Save the sheet.
Book.SaveAs("C:\\TEST.XLS");

// Close Excel with the Quit method on the Application object.
Excel.Application.Quit();
```

若要在远程服务器上创建对象，只能在关闭 Internet 安全机制时完成。您可以通过将计算机的名称传递到 **ActiveXObject** 的 *servername* 参数在远程网络计算机上创建对象。该名称与共享名的计算机名部分相同。对于名为"\\MyServer\public"的网络共享，*servername* 为"MyServer"。此外，您可以使用 DNS 格式或 IP 地址来指定 *servername*。

以下代码返回在名为"MyServer"的远程网络计算机上运行的 Excel 实例的版本号：

```
function GetAppVersion() {
   var Excel = new ActiveXObject("Excel.Application", "MyServer");
   return(Excel.Version);
}
```

如果指定的远程服务器不存在或者找不到，则会出错。

属性和方法

ActiveXObject 对象不具有任何内部属性或方法；它允许您访问自动化对象的属性和方法。

要求

版本 1

请参见

参考

new 运算符

GetObject 函数 (JScript 8.0)

arguments 对象

此对象表示当前所执行的函数、其参数和调用它的函数。此对象不能显式构造。

属性

arguments 对象属性

方法

arguments 对象没有方法。

要求

版本 1

备注

当 **arguments** 对象开始执行时，它会为每个函数进行实例化。**arguments** 对象只能在其关联函数的范围内直接进行访问。

传递到函数的所有参数和参数的数目都存储在 **arguments** 对象中。**arguments** 对象不是数组，但访问各个参数与访问数组元素的方式相同，要使用 [ ] 符号。

您可以使用 **arguments** 对象来创建可接受任意个参数的函数。这一功能还可以通过在定义函数时使用参数数组结构来实现。有关更多信息，请参见 **function** 语句主题。

注意
arguments 对象在以快速模式（JScript 的默认模式）运行时不可用。若要从命令行编译使用 **arguments** 对象的程序，则必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

示例

下面的示例阐释 arguments 对象的用法：

```
function argTest(a, b) : String {
   var i : int;
   var s : String = "The argTest function expected ";
   var numargs : int = arguments.length; // Get number of arguments passed.
   var expargs : int = argTest.length;   // Get number of arguments expected.
   if (expargs < 2)
      s += expargs + " argument. ";
   else
      s += expargs + " arguments. ";
   if (numargs < 2)
      s += numargs + " was passed.";
   else
      s += numargs + " were passed.";
   s += "\n"
   for (i =0 ; i < numargs; i++){        // Get argument contents.
      s += "  Arg " + i + " = " + arguments[i] + "\n";
   }
   return(s);                            // Return list of arguments.
}

print(argTest(42));
print(argTest(new Date(1999,8,7),"Sam",Math.PI));
```

该程序的输出为：

```
The argTest function expected 2 arguments. 1 was passed.
  Arg 0 = 42

The argTest function expected 2 arguments. 3 were passed.
  Arg 0 = Tue Sep 7 00:00:00 PDT 1999
  Arg 1 = Sam
  Arg 2 = 3.141592653589793
```

请参见

参考

new 运算符

function 语句

/fast

arguments 对象属性

arguments 对象表示当前所执行函数的参数以及对其进行调用的函数。

属性

0...n 属性

arguments 属性

callee 属性

caller 属性

length 属性(参数)

请参见

其他资源

属性 (JScript)

JScript 参考

Array 对象

提供对任何数据类型的 expando 数组的支持。**Array** 构造函数有三种形式。

```
function Array( [size : int] )
function Array( [... varargs : Object[]] )
function Array( [array : System.Array )
```

参数

size

可选项。数组大小。当数组基于零时，所创建的元素将具有从零到 *size* -1 的索引。

varargs

可选项。包含传递到构造函数的所有参数的类型化数组。这些参数用作数组中的前几个元素。

array

可选项。要复制到所构造数组的数组。

备注

如果只将一个参数传递到 **Array** 构造函数并且参数为数字，则该参数必须是无符号的 32 位整数（任何小于约四十亿的整数）。所传递的值是数组的大小。如果该值是小于零的数字或者不是整数，则将出现运行时错误。

数据类型为 **System.Array** 的变量可以传递到 **Array** 构造函数。它将生成作为输入数组副本的 JScript 数组。**System.Array** 必须只有一维。

如果将单个值传递到 **Array** 构造函数而且该值不是数字或数组，数组的 **length** 属性将设置为 1，数组第一个元素（元素 0）的值将成为单个传入的参数。如果将几个参数传递到构造函数，数组的长度则设置为参数的数目，这些参数将成为新数组中的前几个元素。

请注意，JScript 数组是稀疏数组；也就是说，即使您为数组分配多个元素，仍会只存在实际包含数据的元素。这将减少数组所使用的内存量。

Array 对象与 **System.Array** 数据类型互用。因此，**Array** 对象可以调用 **System.Array** 数据类型的方法和属性，而 **System.Array** 数据类型可以调用 **Array** 对象的方法和属性。此外，采用 **System.Array** 数据类型的函数接受 **Array** 对象，反之亦然。有关更多信息，请参见 Array 成员。

当 **Array** 对象传递到采用 **System.Array** 的函数或者从 **Array** 对象调用 **System.Array** 方法时，将复制 **Array** 的内容。这样，原始 **Array** 对象无法由 **System.Array** 方法修改，或通过将其传递给接受 **System.Array** 的函数来修改。只有非破坏性 **Array** 方法可在 **System.Array** 上调用。

提示

当您需要一般堆栈或项列表并且性能不是您最关心的问题时，**Array** 对象会给您带来许多方便。在其他所有上下文中，应使用类型化的数组数据类型。类型化数组的许多功能与 **Array** 对象相同，它也提供类型安全性、性能改进以及与其他语言的更好交互。

注意

Array 对象可以与 JScript 中的 .NET Framework **System.Array** 数据类型互用。但是，其他公共语言规范 (CLS) 语言不能使用 **Array** 对象，因为只有 JScript 提供此对象；它不是从 .NET Framework 类型派生的。因此，当对参数进行类型批注并返回符合 CLS 的方法的类型时，请确保使用 **System.Array** 数据类型，而不是 **Array** 对象。然而，可以使用 **Array** 对象对标识符（而不是参数或返回类型）进行类型批注。有关更多信息，请参见编写符合 CLS 的代码。

示例

可以使用 [ ] 符号来访问数组的各个元素。例如：

```
var my_array = new Array();
for (var i = 0; i < 10; i++) {
   my_array[i] = i;
```

```
}
var x = my_array[4];
```

由于 Microsoft JScript 中的数组基于零，所在在以上示例的最后语句中访问数组元素为第五个元素。该元素包含值 4。

属性和方法

Array 对象属性和方法

要求

版本 2

请参见

参考

new 运算符

概念

类型化数组

Array 对象属性和方法

Array 对象支持创建任何数据类型的数组。

属性

constructor 属性

length 属性（数组）

prototype 属性

方法

concat 方法（数组）

join 方法

pop 方法

push 方法

reverse 方法

shift 方法

slice 方法（数组）

sort 方法

splice 方法

toLocaleString 方法

toString 方法

unshift 方法

valueOf 方法

请参见

其他资源

属性 (JScript)

Visual Basic 和 Visual C# 项目扩展性方法

JScript 参考

Boolean 对象

Boolean 对象引用布尔值。

```
function Boolean( [boolValue : boolean] )
```

参数
boolValue

可选项。新对象的初始布尔值。如果 *boolValue* 被省略或者为 **false**、0、**null**、**NaN** 或空字符串，Boolean 对象的初始值则为 **false**。否则，初始值为 **true**。

备注

Boolean 对象是 Boolean 数据的包装。**Boolean** 对象的主要用途是将其属性收集到一个对象中，并允许布尔值通过 **toString** 方法转换为字符串。**Boolean** 对象类似于 **boolean** 数据类型。但是，它们具有不同的属性和方法。

注意

您很少需要显式构造 Boolean 对象。在大多数情况下应使用 boolean 数据类型。由于 Boolean 对象与 boolean 数据类型互用，所有 Boolean 对象方法和属性都可用于类型为 boolean 的变量。有关更多信息，请参见 boolean 数据类型。

Boolean 对象的数据类型是 **Object** 而不是 **boolean**。

属性和方法

Boolean 对象属性和方法

要求

版本 2

请参见
参考
Object 对象
Boolean Structure
new 运算符
var 语句

Boolean 对象属性和方法

Boolean 对象创建新的布尔值。

属性

constructor 属性

prototype 属性

方法

toString 方法

valueOf 方法

请参见

其他资源

属性 (JScript)

Visual Basic 和 Visual C# 项目扩展性方法

JScript 参考

Date 对象

此对象启用日期和时间的基本存储和检索。**Date** 构造函数有两种形式。

```
function Date( [dateVal : { Number | String | System.DateTime } ] )
function Date( year : int, month : int, date : int[, hours : int [, minutes : int [, second
s : int [, ms : int]]]] )
```

参数

dateVal

可选项。如果是数值，*dateVal* 以协调通用时间表示指定日期与 1970 年 1 月 1 日午夜之间相差的毫秒数。如果是字符串，则按照 **parse** 方法中的规则对 *dateVal* 进行分析。*dateVal* 也可以是 .NET 日期值。

year

必选。年份全称，如 1976（而不是 76）。

month

必选。月份，用从 0 到 11 的整数表示（1 月至 12 月）。

date

必选。日期，用从 1 到 31 的整数表示。

hours

可选项。如果提供了 minutes 则必须给出该参数。一个指定小时的，从 0 到 23 的整数（午夜到 11pm）。

minutes

可选项。如果提供了 seconds 则必须给出该参数。一个指定分钟的，从 0 到 59 的整数。

seconds

可选项。如果提供了 milliseconds 则必须给出该参数。一个指定秒的，从 0 到 59 的整数。

ms

可选项。一个指定毫秒的，从 0 到 999 的整数。

备注

Date 对象包含表示特定时间实例（可以精确到毫秒）的数字。如果参数值大于其取值范围或者为负，则会相应地修改存储的其他值。例如，如果指定 150 秒，则 JScript 将该数字重新定义为 2 分 30 秒。

如果数字是 **NaN**，则对象不表示特定的时间实例。如果没有将任何参数传递到 **Date** 构造函数，该函数将初始化为当前时间 (UTC)。类型为 **Date** 的变量在使用之前必须进行初始化。

可在 **Date** 对象中表示的日期范围大约为 1970 年 1 月 1 日左右的 285,616 年。

Date 对象具有两种静态方法：**parse** 和 **UTC**，它们在不创建 **Date** 对象的情况下进行调用。

如果在不使用 **new** 的情况下调用 **Date** 构造函数，那么无论向该构造函数传递什么参数，所返回的 **Date** 对象都包含当前日期。

注意

Date 对象可以与 JScript 中的 .NET Framework **System.DateTime** 数据类型互用。但是，其他公共语言规范 (CLS) 语言不能使用 **Date** 对象，因为只有 JScript 提供此对象；它不是从 .NET Framework 类型派生的。因此，当对参数进行类型批注并返回符合 CLS 的方法的类型时，请确保使用 **System.DateTime** 数据类型，而不是 **Date** 对象。然而，可以使用 **Date** 对象对标识符（而不是参数或返回类型）进行类型批注。有关更多信息，请参见编写符合 CLS 的代码。

示例

下面的示例使用 **Date** 对象。

```
var s : String = "Today's date is: ";   // Declare variables.
var d : Date = new Date();              // Create Date object with today's date.
s += (d.getMonth() + 1) + "/";          // Get month
s += d.getDate() + "/";                 // Get day
s += d.getYear();                       // Get year.
print(s);                               // Print date.
```

如果该程序在 1992 年 1 月 26 日运行，输出本应该是：

```
Today's date is: 1/26/1992
```

属性和方法

Date 对象属性和方法

要求

版本 1

请参见

参考

new 运算符

var 语句

JScript 8.0

Date 对象属性和方法

Date 对象启用日期和时间的基本存储和检索。

属性

constructor 属性

prototype 属性

方法

getDate 方法

getDay 方法

getFullYear 方法

getHours 方法

getMilliseconds 方法

getMinutes 方法

getMonth 方法

getSeconds 方法

getTime 方法

getTimezoneOffset 方法

getUTCDate 方法

getUTCDay 方法

getUTCFullYear 方法

getUTCHours 方法

getUTCMilliseconds 方法

getUTCMinutes 方法

getUTCMonth 方法

getUTCSeconds 方法

getVarDate 方法

getYear 方法

parse 方法

setDate 方法

setFullYear 方法

setHours 方法

setMilliseconds 方法

setMinutes 方法

setMonth 方法

setSeconds 方法

setTime 方法

setUTCDate 方法

setUTCFullYear 方法

setUTCHours 方法

setUTCMilliseconds 方法

setUTCMinutes 方法

setUTCMonth 方法

setUTCSeconds 方法

setYear 方法

toDateString 方法

toGMTString 方法

toLocaleDateString 方法

toLocaleString 方法

toLocaleTimeString 方法

toString 方法

toTimeString 方法

toUTCString 方法

UTC 方法

valueOf 方法

请参见

其他资源

属性 (JScript)

Visual Basic 和 Visual C# 项目扩展性方法

JScript 参考

Enumerator 对象

启用集合中项的枚举。

```
varName = new Enumerator([collection])
```

参数
varName

必选。枚举数分配到的变量名称。

collection

可选项。任何实现 **IEnumerable** 接口的对象，如数组或集合。

备注

每个集合在 JScript 中自动具有可枚举性。因此，您不需要使用 **Enumerator** 对象来访问集合的成员。您可以使用 **for...in** 语句来直接访问任何成员。**Enumerator** 对象是为实现向后兼容性而提供的。

与数组不同，集合的成员不可以直接进行访问。您不能像对于数组一样使用索引，而只能将当前项指针移动到集合的第一个或下一个元素。

Enumerator 对象提供了访问任何集合成员的方法，它的行为方式类似于 VBScript 中的 **For...Each** 语句。

您可以通过定义实现 **IEnumerable** 的类在 JScript 中创建集合。也可以使用另一种语言（如 Visual Basic）或通过 **ActiveXObject** 对象来创建集合。

示例 1

以下代码使用 **Enumerator** 对象来输出可用驱动器的盘符及其名称（如果可用）：

```
// Declare variables.
var n, x;
var fso : ActiveXObject = new ActiveXObject("Scripting.FileSystemObject");
// Create Enumerator on Drives.
var e : Enumerator = new Enumerator(fso.Drives);
for (;!e.atEnd();e.moveNext()) {      // Loop over the drives collection.
   x = e.item();
   if (x.DriveType == 3)              // See if network drive.
      n = x.ShareName;                // Get share name
   else if (x.IsReady)                // See if drive is ready.
      n = x.VolumeName;               // Get volume name.
   else
      n = "[Drive not ready]";
   print(x.DriveLetter + " - " + n);
}
```

根据系统的不同，输出将类似于下面这样：

```
A - [Drive not ready]
C - DRV1
D - BACKUP
E - [Drive not ready]
```

示例 2

示例 1 中代码可以进行改写，以便在不使用 Enumerator 对象的情况下使用。在该示例中，将直接访问枚举的成员。

```
// Declare variables.
var n, x;
var fso : ActiveXObject = new ActiveXObject("Scripting.FileSystemObject");
// The following three lines are not needed.
//    var e : Enumerator = new Enumerator(fso.Drives);
```

```
//    for (;!e.atEnd();e.moveNext()) {
//       x = e.item();
// Access the members of the enumeration directly.
for (x in fso.Drives) {               // Loop over the drives collection.
   if (x.DriveType == 3)              // See if network drive.
      n = x.ShareName;                // Get share name
   else if (x.IsReady)                // See if drive is ready.
      n = x.VolumeName;               // Get volume name.
   else
      n = "[Drive not ready]";
   print(x.DriveLetter + " - " + n);
}
```

属性

Enumerator 对象没有属性。

方法

Enumerator 对象方法

要求

版本 3

请参见

参考

new 运算符
for...in 语句

Enumerator 对象方法

Enumerator 对象启用集合中项的枚举。

方法

atEnd 方法

item 方法

moveFirst 方法

moveNext 方法

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

JScript 参考

错误对象

包含有关错误的信息。**Error** 构造函数有两种形式。

```
function Error([description : String ])
function Error([number : Number [, description : String ]])
```

参数

number

可选项。分配到错误的数值，指定 **number** 属性的值。如果省略，则为零。

description

可选项。描述错误的简短字符串，指定 **description** 和 **message** 属性的初始值。如果省略，则为空字符串。

备注

Error 对象可以使用以上所示的构造函数来显式地创建。您可以在 **Error** 对象中添加属性，以扩展其功能。每当出现运行时错误，也会创建一个 **Error** 对象来对错误进行描述。

通常，Error 对象通过 **throw** 语句以及它将被 **try...catch** 语句捕获这一期望来引发。您可以使用 **throw** 语句将任何类型的数据当作错误来传递；**throw** 语句将不会隐式地创建 **Error** 对象。但是，通过引发 **Error** 对象，**catch** 块可以类似地处理 JScript 运行时错误和用户定义的错误。

Error 对象具有四项内部属性：错误说明（**description** 和 **message** 属性）、错误号（**number** 属性）和错误的名称（**name** 属性）。**description** 和 **message** 属性引用相同的消息；**description** 属性提供向后兼容性，而 **message** 属性符合 ECMA 标准。

错误号是一个 32 位的值。较高的 16 位字是设施代码，而较低的字才是真正的错误代码。若要读完实际的错误代码，请使用 **&**（按位与）运算符来将 number 属性与十六进制数字 `0xFFFF` 组合。

警告

尝试在 ASP.NET 页中使用 JScript **Error** 对象可能会产生意外的结果。这是因为 ASP.NET 页的 JScript **Error** 对象与 **Error** 事件之间可能存在二义性。在 ASP.NET 页中处理错误时，要使用 **System.Exception** 类，而不使用 **Error** 对象。

注意

只有 JScript 提供 **Error** 对象。由于它不是从 .NET Framework 类型派生的，所以其他公共语言规范 (CLS) 语言不能使用它。因此，当对参数和符合 CLS 的方法的返回类型进行类型批注时，请确保使用 **System.Exception** 数据类型，而不是 **Error** 对象。然而，可以使用 **Error** 对象对标识符（而不是参数或返回类型）进行类型批注。有关更多信息，请参见编写符合 CLS 的代码。

示例

下面的示例阐释 **Error** 对象的用法。

```
try {
   // Throw an error.
   throw new Error(42,"No question");
} catch(e) {
   print(e)
// Extract the error code from the error number.
   print(e.number & 0xFFFF)
   print(e.description)
}
```

该代码的输出为：

```
Error: No question
42
No question
```

属性和方法

Error 对象属性和方法

要求

版本 5

请参见

参考

new 运算符
throw 语句
try...catch...finally 语句
var 语句
AllMembers.T:System.Web.UI.Page

Error 对象属性和方法

Error 对象包含有关错误的信息。

属性

description 属性

message 属性

name 属性

number 属性

方法

toString 方法

请参见

其他资源

属性 (JScript)

Visual Basic 和 Visual C# 项目扩展性方法

JScript 参考

Function 对象

创建新函数。

```
function Function( [[param1 : String, [..., paramN : String,]] body : String ])
```

参数

param1, ..., paramN

可选项。函数的参数。每个参数都可以具有类型批注。最后一个参数可以是 *parameterarray*，它由三个句号 (**...**) 后接一个参数数组名称和一个类型化数组类型批注来表示。

body

可选项。包含在调用函数时要执行的 JScript 代码块的字符串。

备注

Function 构造函数允许脚本在运行时创建函数。传递到 **Function** 构造函数的参数（除最后一个参数之外的所有参数）将用作新函数的参数。传递到构造函数的最后一个参数解释为函数体的代码。

JScript 在调用 **Function** 构造函数时编译由该构造函数创建的对象。虽然它使您的脚本在运行时重新定义函数的情况下具有更大的灵活性，但它也会减慢代码的执行速度。为了避免减慢脚本速度，应尽可能少地使用 **Function** 构造函数。

当调用要计算的函数时，应始终包括小括号和必需的参数。如果在无小括号的情况下调用函数，将返回该函数的 **Function** 对象。函数的文本可以使用 **Function** 对象的 **toString** 方法来获取。

注意

只有 JScript 提供 **Function** 对象。由于它不是从 .NET Framework 类型派生的，所以其他公共语言规范 (CLS) 语言不能使用它。因此，当对参数和符合 CLS 的方法的返回类型进行类型批注时，请确保使用 **System.EventHandler** 数据类型，而不是 **Function** 对象。然而，可以使用 **Function** 对象对标识符（而不是参数或返回类型）进行类型批注。有关更多信息，请参见编写符合 CLS 的代码。

示例

下面的示例阐释 **Function** 对象的用法。

```
var add : Function = new Function("x", "y", "return(x+y)");
print(add(2, 3));
```

该代码输出：

```
5
```

属性和方法

Function 对象属性和方法

要求

版本 2

请参见

参考

function 语句
new 运算符
var 语句

Function 对象属性和方法

Function 对象创建新的函数。

属性

0...n 属性

arguments 属性

callee 属性

caller 属性

constructor 属性

length 属性（函数）

prototype 属性

方法

apply 方法

call 方法

toString 方法

valueOf 方法

请参见

其他资源

属性 (JScript)

Visual Basic 和 Visual C# 项目扩展性方法

JScript 参考

Global 对象

内部对象，其用途是将全局方法收集到一个对象中。

Global 对象没有语法。请直接调用其方法。

属性和方法

Global 对象属性和方法

要求

版本 5

备注

Global 对象从不直接使用，并且不能使用 **new** 运算符来创建。它在初始化脚本引擎时创建，从而使其方法和属性可以直接使用。

请参见

参考

Object 对象

Global 对象属性和方法

Global 对象是一种内部对象，其目的是将全局方法收集到一个对象中。

属性

Infinity 属性

NaN 属性（全局）

undefined 属性

方法

decodeURI 方法

decodeURIComponent 方法

encodeURI 方法

encodeURIComponent 方法

escape 方法

eval 方法

isFinite 方法

isNaN 方法

parseFloat 方法

parseInt 方法

unescape 方法

请参见

其他资源

属性 (JScript)

Visual Basic 和 Visual C# 项目扩展性方法

JScript 参考

JScript 8.0

Math 对象

提供基本数学函数和常数的内部对象。此对象不能显式构造。

属性和方法

Math 对象属性和方法

要求

版本 1

备注

new 运算符不能创建 **Math** 对象，如果您试图这样做，它将返回错误。当加载脚本引擎时，该引擎将创建 **Math** 对象。它的所有方法和属性在任何时候都可用于脚本。

下面的示例阐释 **Math** 对象的用法。请注意，由于浮点数的精度有限，涉及浮点数的计算可以累积次要的舍入错误。您可以使用 **Number** 对象的 **toFixed** 方法来显示无次要舍入错误的数字。

示例

```
var pi : double = Math.PI;         // Should be about 3.14.
print(pi);
var cosPi : double = Math.cos(pi); // Should be minus one.
print(cosPi);
var sinPi : double = Math.sin(pi); // Should be zero.
print(sinPi.toFixed(10));
```

该代码的输出为：

```
3.141592653589793
-1
0.0000000000
```

请参见

参考

Number 对象

JScript 8.0

Math 对象属性和方法

Math 对象是一种内部对象，它提供基本的数学函数和常数。

属性

E 属性

LN10 属性

LN2 属性

LOG10E 属性

LOG2E 属性

PI 属性

SQRT1_2 属性

SQRT2 属性

方法

abs 方法

acos 方法

asin 方法

atan 方法

atan2 方法

ceil 方法

cos 方法

exp 方法

floor 方法

log 方法

max 方法

min 方法

pow 方法

random 方法

round 方法

sin 方法

sqrt 方法

tan 方法

请参见

其他资源

属性 (JScript)

Visual Basic 和 Visual C# 项目扩展性方法

JScript 参考

JScript 8.0

Number 对象

数值数据和数值常数占位符的对象表示形式。

```
function Number( [value : Number] )
```

参数
value

必选。所创建的 **Number** 对象的数值。

备注

Number 对象是数值数据的包装。**Number** 对象的主要用途是将其属性收集到一个对象中，并允许数字通过 **toString** 方法转换为字符串。**Number** 对象类似于 **Number** 数据类型。但是，它们具有不同的属性和方法。

注意

您很少需要显式地构造 **Number** 对象。在大多数情况下应使用 **Number** 数据类型。由于 **Number** 对象与 **Number** 数据类型互用，所有 **Number** 对象方法和属性都可用于类型为 **Number** 的变量。有关更多信息，请参见 Number 数据类型。

Number 对象将数值数据存储为八字节、双精度的浮点数。它表示双精度 64 位 IEEE 754 值。**Number** 对象可以表示从负 1.79769313486231570E+308 到正 1.79769313486231570E+308（包括这两个数）的范围内的数字。可以表示的最小数字是 4.94065645841247E-324。**Number** 对象也可以表示 **NaN**（非数字）、正负无穷和正负零。

Number 对象的数据类型为 **Object**，而不是 **Number**。

属性和方法

Number 对象属性和方法

要求

版本 1

请参见
参考
Object 对象
Number 数据类型
Math 对象
new 运算符

Number 对象属性和方法

Number 对象是数字数据类型和数值常数占位符的对象表示形式。

属性

constructor 属性

MAX_VALUE 属性

MIN_VALUE 属性

NaN 属性

NEGATIVE_INFINITY 属性

POSITIVE_INFINITY 属性

prototype 属性

方法

toExponential 方法

toFixed 方法

toLocaleString 方法

toPrecision 方法

toString 方法

valueOf 方法

请参见

其他资源

属性 (JScript)

Visual Basic 和 Visual C# 项目扩展性方法

JScript 参考

Object 对象

提供所有 JScript 对象共有的功能。

```
function Object([value : { ActiveXObject | Array | Boolean | Date | Enumerator | Error | Fu
nction | Number | Object | RegExp | String | VBArray ])
```

参数
value

可选项。任何一种 JScript 基元数据类型。如果 *value* 不是对象，则将返回未修改的对象。如果 *value* 是 **null**、**undefined** 或未提供，则创建无内容的对象。

备注

Object 对象构成了其他所有 JScript 对象的基础；它的所有方法和属性在其他所有对象中都可用。其方法可以在用户定义的对象中重新定义，JScript 将在适当的时候调用这些方法。**toString** 方法是经常重新定义的 **Object** 方法的一个示例。

没有用类型批注定义的变量隐式地属于类型 **Object**。除了它自己的属性和方法之外，每个 JScript 对象具有 **Object** 对象的所有属性和方法。

属性和方法

Object 对象属性和方法

要求

版本 3

请参见
参考
new 运算符
Function 对象
Global 对象

Object 对象属性和方法

Object 对象提供所有 JScript 对象共有的功能。

属性

constructor 属性

prototype 属性

propertyIsEnumerable 属性

方法

isPrototypeOf 方法

hasOwnProperty 方法

toLocaleString 方法

toString 方法

valueOf 方法

请参见

其他资源

属性 (JScript)
Visual Basic 和 Visual C# 项目扩展性方法
JScript 参考

RegExp 对象

内部对象，它存储有关正则表达式模式匹配结果的信息。此对象不能显式构造。

属性

RegExp 对象属性

方法

RegExp 对象没有方法。

要求

版本 3

备注

RegExp 对象不能直接创建，但它始终可用。在完成成功的正则表达式搜索之前，RegExp 对象的各项属性具有如下初始值：

属性	简写	初始值
index		-1
input	$_	空字符串
lastIndex		-1
lastMatch	$&	空字符串。
lastParen	$+	空字符串。
leftContext	$`	空字符串。
rightContext	$'	空字符串。
$1 - $9		空字符串。

全局 **RegExp** 对象不应与 **Regular Expression** 对象混淆。虽然它们看起来相同，但它们实际上是相互分离，互不相同的。全局 **RegExp** 对象的属性包含有关所发生的每一匹配的不断更新的信息，而 **Regular Expression** 对象的属性只包含有关与 **Regular Expression** 单个实例发生的匹配的信息。

注意

以快速模式（JScript 的默认模式）运行时，**RegExp** 对象的属性不可用。若要从命令行编译使用这些属性的程序，必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

下面的示例阐释全局 **RegExp** 对象的用法。该示例必须使用 **/fast-** 选项来编译。

示例

```
var re : RegExp = new RegExp("d(b+)(d)","ig");
var arr : Array = re.exec("cdbBdbsbdbdz");
print("$1 contains: " + RegExp.$1);
print("$2 contains: " + RegExp.$2);
print("$3 contains: " + RegExp.$3);
```

该代码的输出为：

```
$1 contains: bB
$2 contains: d
$3 contains:
```

请参见

参考

正则表达式对象

String 对象

/fast

概念

正则表达式语法

RegExp 对象属性

RegExp 对象是一种内部全局对象，它存储有关正则表达式模式匹配结果的信息。

属性

$1...$9 属性

index 属性

input 属性 ($_)

lastIndex 属性

lastMatch 属性 ($&)

lastParen 属性 ($+)

leftContext 属性 ($`)

rightContext 属性 ($')

请参见

其他资源

属性 (JScript)

JScript 参考

正则表达式对象

此对象包含正则表达式模式以及标识如何应用该模式的标志。

```
// The explicit constructor for a Regular Expression object.
function RegExp(pattern : String [,flags : String])
function RegExp(regexObj : System.Text.RegularExpressions.Regex)

// The implicit constructor for a Regular Expression object.
/pattern/[flags]
```

参数

pattern

必选。要使用的正则表达式模式。如果使用语法 1，则模式必须为字符串。如果使用语法 2，模式将由"/"字符分隔。

flags

可选项。如果使用语法 1，则标志必须处于字符串中。如果使用语法 2，标志字符将紧接在最后一个"/"字符之后。可以组合使用的可用标志有：

- g（全局搜索出现的所有 pattern）
- i（忽略大小写）
- m（多行搜索）

regexObj

必选。包含要使用的正则表达式模式的 **Regex** 对象。

备注

正则表达式对象不应与全局 **RegExp** 对象混淆。它们虽然看起来相似，但也可以很容易地加以区分。正则表达式对象的属性只包含有关一个特定正则表达式实例的信息，而全局 **RegExp** 对象的属性则包含有关所发生的每一匹配的不断更新的信息。

正则表达式对象存储用于搜索字符组合的字符串的模式。当创建正则表达式对象后，会将该对象传递到字符串方法，或者给字符串传递正则表达式对象的方法。有关最近执行的搜索的信息存储在全局 **RegExp** 对象中。

当搜索字符串频繁更改或未知时（例如从用户输入导出的字符串），应使用语法 1。当您提前知道搜索字符串时，应使用语法 2。

在 JScript 中，*pattern* 参数将在使用之前编译为内部格式。对于语法 1，*pattern* 正好在使用之前进行编译或者在调用 **compile** 方法时进行编译。对于语法 2，*pattern* 在脚本加载时编译。

注意

正则表达式对象可以与 JScript 中的 .NET Framework **System.Text.RegularExpressions.Regex** 数据类型互用。但是，其他公共语言规范 (CLS) 语言不能使用正则表达式对象，因为只有 JScript 提供此对象；它不是从 .NET Framework 类型派生的。因此，当对参数和符合 CLS 的方法的返回类型进行类型批注时，请确保使用 **System.Text.RegularExpressions.Regex** 数据类型，而不是"正则表达式"对象。然而，可以使用"正则表达式"对象对标识符（而不是参数或返回类型）进行类型批注。有关更多信息，请参见编写符合 CLS 的代码。

示例

下面的示例阐释正则表达式对象的用法。它将创建对象 `re1` 和 `re2`，这两个对象包含带有关联标志的正则表达式模式。在这种情况下，**match** 方法将在随后使用所得到的正则表达式对象：

```
var s : String = "The rain in Spain falls mainly in the plain";
// Create regular expression object using Syntax 1.
var re1 : RegExp = new RegExp("Spain","i");
// Create regular expression object using Syntax 2.
var re2 : RegExp = /IN/i;

// Find a match within string s.
print(s.match(re1));
```

```
print(s.match(re2));
```

该脚本的输出为

```
Spain
in
```

要求

版本 3

属性和方法

Regular Expression 对象属性和方法

请参见

参考

new 运算符
RegExp 对象
String 对象
Regex

概念

正则表达式语法

Regular Expression 对象属性和方法

正则表达式对象是包含正则表达式模式以及标识如何应用模式的标志的对象。

属性

global 属性

ignoreCase 属性

multiline 属性

source 属性

方法

compile 方法

exec 方法

test 方法

请参见

其他资源

属性 (JScript)
Visual Basic 和 Visual C# 项目扩展性方法
JScript 参考

String 对象

允许操作和格式化文本字符串并确定和定位字符串中的子字符串。

```
function String([stringLiteral : String])
```

参数
stringLiteral

可选项。任何 Unicode 字符组。

备注

String 对象可使用字符串文本隐式创建。以这种方式创建的 **String** 对象(指"基元"字符串)与使用 **new** 运算符创建的 **String** 对象处理方式不同。虽然可以读取属性并对基元字符串调用方法，但不能创建新的属性或为其添加新的方法。

在字符串中可以使用转义序列来表示不能直接在字符串中使用的特殊字符，如换行符或 Unicode 字符。当编译脚本时，字符串中的每个转义序列都会转换为它所表示的字符串。有关其他信息，请参见字符串数据。

JScript 还定义了 **String** 数据类型，它提供与 **String** 对象不同的属性和方法。不能为 **String** 数据类型的变量创建属性或向其添加方法，但是可以为 **String** 对象的实例创建属性或向其添加方法。

String 对象与 **String** 数据类型(它与 **System.String** 数据类型相同)互用。这意味着 **String** 对象可以调用 **String** 数据类型的方法和属性，而 **String** 数据类型可以调用 **String** 对象的方法和属性。有关更多信息，请参见 AllMembers.T:System.String。此外，**String** 对象被采用 **String** 数据类型的函数所接受，反之亦然。

String 对象的数据类型为 **Object**，而不是 **String**。

示例 1

以下脚本说明虽然可以读取长度属性并调用 **toUpperCase** 方法，但不能在基元字符串上设置自定义属性 `myProperty`:

```
var primStr : Object = "This is a string";
print(primStr.length);           // Read the length property.
print(primStr.toUpperCase());    // Call a method.
primStr.myProperty = 42;         // Set a new property.
print(primStr.myProperty);       // Try to read it back.
```

该脚本的输出为:

```
16
THIS IS A STRING
undefined
```

示例 2

对于用 **new** 语句创建的 **String** 对象，可以设置自定义属性:

```
var newStr : Object = new String("This is also a string");
print(newStr.length);           // Read the length property.
print(newStr.toUpperCase());    // Call a method.
newStr.myProperty = 42;         // Set a new property.
print(newStr.myProperty);       // Try to read it back.
```

该脚本的输出为:

```
21
THIS IS ALSO A STRING
42
```

属性和方法

String 对象属性和方法

要求

版本 1

请参见

参考

Object 对象
String 数据类型 (JScript)
new 运算符

概念

字符串数据

JScript 8.0

String 对象属性和方法

String 对象允许操作和格式化文本字符串以及确定和定位字符串中的子字符串。

属性

constructor 属性

length 属性(字符串)

prototype 属性

方法

anchor 方法

big 方法

blink 方法

bold 方法

charAt 方法

charCodeAt 方法

concat 方法(字符串)

fixed 方法

fontcolor 方法

fontsize 方法

fromCharCode 方法

indexOf 方法

italics 方法

lastIndexOf 方法

link 方法

localeCompare 方法

match 方法

replace 方法

search 方法

slice 方法(字符串)

small 方法

split 方法

strike 方法

sub 方法

substr 方法

substring 方法

sup 方法

toLocaleLowerCase 方法

toLocaleUpperCase 方法

toLowerCase 方法

toString 方法

toUpperCase 方法

valueOf 方法

请参见

其他资源

属性 (JScript)

Visual Basic 和 Visual C# 项目扩展性方法

JScript 参考

VBArray 对象

提供对 Visual Basic 安全数组的访问。

```
varName = new VBArray(safeArray)
```

参数

varName

必选。VBArray 分配到的变量名称。

safeArray

必选。VBArray 值。

备注

safeArray 参数在传递到 VBArray 构造函数之前必须具有一个 VBArray 值。要获取该值，可以从现有的 ActiveX 或其他对象检索该值。

注意

在 JScript 中创建的数组和在 Visual Basic 中创建的数组都可以与 .NET Framework 数组互用。因此，可以在 JScript 中直接访问在 Visual Basic 中创建的数组的元素。**VBArray** 对象仅为实现向后兼容性而提供的。有关数组的更多信息，请参见数组对象、Dim 语句和 AllMembers.T:System.Array。

VBArray 可以具有多维。每一维的索引可以不同。**dimensions** 方法检索数组中的维数；**lbound** 和 **ubound** 方法检索每一维所使用的索引范围。

属性

VBArray 对象没有属性。

方法

VBArray 对象方法

要求

版本 3

请参见

参考

new 运算符
Array 对象
Array

VBArray 对象方法

VBArray 对象提供对 Visual Basic 安全数组的访问。

方法

dimensions 方法

getItem 方法

lbound 方法

toArray 方法

ubound 方法

请参见

其他资源

Visual Basic 和 Visual C# 项目扩展性方法

JScript 参考

运算符 (JScript)

JScript 包含大量运算符，它们属于算术、逻辑、按位、赋值和杂项类别。以下各节链接到有关解释如何使用这些运算符的信息。

本节内容

加法赋值运算符 (+=)

将两个数相加或串联两个字符串，然后将结果赋给第一个参数。

加法运算符 (+)

将两个数相加或串联两个字符串。

赋值运算符 (=)

给变量赋值。

按位"与"赋值运算符 (&=)

对两个表达式执行按位"与"运算，然后将结果赋给第一个参数。

按位"与"运算符 (&)

对两个表达式执行按位"与"运算。

按位左移运算符 (<<)

将一个表达式的位左移。

按位"取非"运算符 (~)

对一个表达式执行按位"取非"(求非)运算。

按位"或"赋值运算符 (|=)

对两个表达式执行按位"或"运算，然后将结果赋给第一个参数。

按位"或"运算符 (|)

对两个表达式执行按位"或"运算。

按位右移运算符 (>>)

将一个表达式的位右移，保留符号。

按位"异或"赋值运算符 (^=)

对两个表达式执行按位 XOR 运算，然后将结果赋给第一个参数。

按位"异或"运算符 (^)

对两个表达式执行按位 XOR 运算。

逗号运算符 (,)

顺序执行两个表达式。

比较运算符

各种运算符(==、>、>=、===、!=、<、<=、!==)，这些运算符返回一个布尔值指示比较的结果。

条件(三元)运算符 (?:)

根据条件从两个语句中选择一个要运行的语句。

delete 运算符

从对象中删除一个属性，或从数组中移除一个元素。

除法赋值运算符 (/=)

将两个数相除并返回一个数值结果，然后将结果赋给第一个参数。

除法运算符 (/)

将两个数相除并返回一个数值结果。

in 运算符

测试一个对象中是否存在一种属性。

递增 (++) 和递减 (--) 运算符

增量运算符 (++) 将某个变量加一；减量运算符 (--) 将某个变量减一。

instanceof 运算符

返回一个布尔值，该值指示一个对象是否为特定类的一个实例。

左移赋值运算符 (<<=)

将表达式的位左移，然后将结果赋给第一个参数。

逻辑"与"运算符 (&&)

对两个表达式执行逻辑合取操作。

逻辑"非"运算符 (!)

对一个表达式执行逻辑求反操作。

逻辑或运算符 (||)

对两个表达式执行逻辑析取操作。

取模赋值运算符 (%=)

将两个数相除，然后将余数赋给第一个参数。

取模运算符 (%)

将两个数相除并返回余数。

乘法赋值运算符 (*=)

将两个数相乘，然后将结果赋给第一个参数。

乘法运算符 (*)

将两个数相乘。

new 运算符

创建一个新对象。

引用运算符 (&)

允许将对变量的引用传递到使用引用或输出参数的方法。

右移赋值运算符 (>>=)

将表达式的位左移，保留符号，然后将结果赋给第一个参数。

减法赋值运算符 (-=)

从另一个数中减去一个数，然后将结果赋给第一个参数。

减法运算符 (-)

指示数值表达式的负值或者从另一个数中减去一个数。

typeof 运算符

返回一个用于标识表达式的数据类型的字符串。

无符号右移赋值运算符 (>>>=)

对一个表达式中的位执行无符号右移，然后将结果赋给第一个参数。

无符号右移运算符 (>>>)

对一个表达式中的位执行无符号右移。

void 运算符

禁止表达式返回值。

相关章节

JScript 参考

列出"JScript 语言参考"所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

JScript 运算符

提供 JScript 中使用的运算符的概念性概述，并提供指向特定主题的链接，这些主题解释每个运算符的正确语法和运算符优先级的意义。

运算符优先级

提供一个列表，该列表包含有关 JScript 运算符的执行优先级的信息。

运算符摘要

列出 JScript 运算符并提供指向特定主题的链接，这些主题解释这些运算符的正确用法。

加法赋值运算符 (+=)

将变量值与表达式值相加，并将结果赋给该变量。

```
result += expression
```

参数

result

任何变量。

表达式

任何表达式。

备注

使用此运算符的效果基本上与指定 `result = result + expression` 相同，不同的只是仅计算一次 *result*。

表达式的类型决定 += 运算符的行为。

结果	*表达式*	**Then**
char	**char**	Error
char	Numeric	Add
char	**String**	Error
Numeric	**char**	Add
Numeric	Numeric	Add
Numeric	**String**	连接
String	**char**	连接
字符串	Numeric	连接
String	**String**	连接

若进行串联，数字将被强迫转换为数值的字符串表示形式，字符将被当作长度为 1 的字符串。若将字符和数字相加，字符将被强迫转换为数值，而后将两个数字相加。某些类型的组合会因加法的结果类型无法强迫为所需的输出类型而出错。

示例

下面的示例阐释加法赋值运算符如何处理不同类型的表达式。

```
var str : String = "42";
var n : int = 20;
var c : char = "A";  // The numeric value of "A" is 65.
var result;
c += n;         // The result is the char "U".
n += c;         // The result is the number 105.
n += n;         // The result is the number 210.
n += str;       // The result is the number 21042.
str += c;       // The result is the string "42U".
str += n;       // The result is the string "42U21042".
str += str;     // The result is the string "42U2104242U21042".
c += c;         // This returns a runtime error.
c += str;       // This returns a runtime error.
```

```
n += "string";  // This returns a runtime error.
```

要求

版本 1

请参见

参考

加法运算符 (+)

赋值运算符 (=)

概念

运算符优先级

运算符摘要

加法运算符 (+)

将数值表达式的值与另一数值表达式相加，或连接两个字符串。

```
expression1 + expression2
```

参数

expression1

任何表达式。

expression2

任何表达式。

备注

表达式的类型决定 + 运算符的行为。

如果	则执行	结果类型
两个表达式都是字符	连接	**String**
两个表达式都是数字	相加	numeric
两个表达式都是字符串	连接	**String**
一个表达式为字符，而另一个表达式为数字	相加	**char**
一个表达式为字符，而另一个表达式为字符串	连接	**String**
一个表达式为数字，而另一个表达式为字符串	连接	**String**

若进行串联，数字将被强迫转换为数值的字符串表示形式，字符将被当作长度为 1 的字符串。若将字符和数字相加，字符将被强迫转换为数值，而后将两个数字相加。

注意

在没有使用类型批注的情况下，可将数值数据存储为字符串。使用显式类型转换或类型注释变量可确保加法运算符不会将数字视为字符串，反之亦然。

示例

下面的示例阐释加法运算符如何处理不同类型的表达式。

```
var str : String = "42";
var n : double = 20;
var c : char = "A";  // the numeric value of "A" is 65
var result;
result = str + str;  // result is the string "4242"
result = n + n;      // result is the number 40
result = c + c;      // result is the string "AA"
result = c + n;      // result is the char "U"
result = c + str;    // result is the string "A42"
result = n + str;    // result is the string "2042"
// Use explicit type coversion to use numbers as strings, or vice versa.
result = int(str) + int(str);    // result is the number 84
result = String(n) + String(n);  // result is the string "2020"
result = c + int(str);           // result is the char "k"
```

要求

版本 1

请参见

参考

加法赋值运算符 (+=)

概念

运算符优先级

运算符摘要

类型转换

赋值运算符 (=)

给变量赋值。

```
result = expression
```

参数
result

任何变量。

表达式

任何表达式。

备注

= 运算符返回 *expression* 的值并将该值赋给 *variable*。这就意味着可以用下面的方式将赋值运算连起来写：

```
j = k = l = 0;
```

执行完该示例语句后，`j`、`k` 和 `l` 的值都等于零。

expression 的数据类型必须可强迫为 *result* 的数据类型。

要求

版本 1

请参见
概念
运算符优先级
运算符摘要

按位“与”赋值运算符 (&=)

对变量值与表达式值执行按位“与”运算，并将结果赋给该变量。

```
result &= expression
```

参数
result

任何数值变量。

表达式

任何数值表达式。

备注

使用此运算符的效果基本上与指定 `result = result & expression` 相同，不同的只是仅计算一次 *result*。

&= 运算符将这些参数强迫为匹配的数据类型。&= 运算符查看 result 和 expression 的二进制表示形式的值，并对其执行按位“与”运算。

该运算的输出如下所示：

```
0101    (result)
1100    (expression)
----
0100    (output)
```

任何时候，只要两个表达式的某位都为 1，则结果中的该位为 1。否则，结果中的该位为 0。

要求

版本 1

请参见
参考
按位“与”运算符 (&)
赋值运算符 (=)
概念
运算符优先级
运算符摘要
按位运算符强制

按位“与”运算符 (&)

对两个表达式执行按位"与"运算。

```
expression1 & expression2
```

参数

expression1

任何数值表达式。

expression2

任何数值表达式。

备注

& 运算符将参数强制为匹配的数据类型。& 运算符查看两个表达式的二进制表示形式的值，并对它们执行按位"与"运算。参数的数据类型决定此运算符所返回的数据类型。

此运算的结果如下所示：

```
0101   (expression1)
1100   (expression2)
----
0100   (result)
```

任何时候，只要两个表达式的某位都为 1，则结果中的该位为 1。否则，结果中的该位为 0。

要求

版本 1

请参见

参考

按位"与"赋值运算符 (&=)

概念

运算符优先级

运算符摘要

按位运算符强制

按位左移运算符 (<<)

左移表达式的位。

```
expression1 << expression2
```

参数

expression1

任何数值表达式。

expression2

任何数值表达式。

备注

<< 运算符将 expression1 的所有位左移 expression2 指定的位数。*expression1* 的数据类型决定此运算符所返回的数据类型。

<< 运算符屏蔽 *expression2* 以免 *expression1* 的移位量太大。否则，如果移位量超出 *expression1* 的数据类型中的位数，则可能移走所有的初始位，从而会提供无意义的结果。为了确保每次移位保留至少一个初始位，移位运算符将使用以下公式来计算实际移位量：使用 *expression1* 中的位数减一所得的结果来屏蔽 *expression2*（使用按位"与"运算符）。

示例

例如：

```
var temp
temp = 14 << 2
```

变量 temp 的值为 56，因为 14（即二进制的 00001110）左移两位等于 56（即二进制的 00111000）。

要理解屏蔽的工作方式，请考虑下面的示例。

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
var y : byte = x << 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in y are 00111100
// The value of y is 60
print(y); // Prints 60
```

要求

版本 1

请参见

参考

左移赋值运算符 (<<=)
按位右移运算符 (>>)
无符号右移运算符 (>>>)

概念

运算符优先级
运算符摘要
按位运算符强制

按位“取非”运算符 (~)

对一个表达式执行按位“取非”(求非)运算。

```
~ expression
```

参数
表达式

任何数值表达式。

备注

~ 运算符查看表达式的二进制表示形式的值，并执行按位求非运算。此运算的结果如下所示：

```
0101   (expression)
----
1010   (result)
```

表达式中的任何一位为 1，则结果中的该位变为 0。表达式中的任何一位为 0，则结果中的该位变为 1。

当 ~ 运算符充当整型数据类型的操作数时，它不进行强制并返回与操作数具有相同数据类型的值。当操作数是非整型数据类型时，该值在运算执行之前被强制为 **int** 类型，该运算符的返回值为 **int** 类型。

要求

版本 1

请参见
参考
逻辑“非”运算符 (!)
概念
运算符优先级
运算符摘要

按位“或”赋值运算符 (|=)

对变量值与表达式值执行按位“或”操作，并将结果赋给该变量。

```
result |= expression
```

参数

result

任何数值变量。

expression

任何数值表达式。

备注

使用此运算符的效果基本上与指定 `result = result | expression` 相同，不同的只是仅计算一次 *result*。

|= 运算符将这些参数强迫为匹配的数据类型。**|=** 运算符查看 *result* 和 *expression* 的二进制表示形式的值，并对其执行按位“或”运算。该运算的结果如下所示：

```
0101    (result)
1100    (expression)
----
1101    (output)
```

任何时候，只要两个表达式中的一个表达式的某位为 1，则结果中的该位为 1。否则，结果中的该位为 0。

要求

版本 1

请参见

参考

按位“或”运算符 (|)

赋值运算符 (=)

概念

运算符优先级

运算符摘要

按位运算符强制

按位“或”运算符 (|)

对两个表达式执行按位“或”运算。

```
expression1 | expression2
```

参数

expression1

任何数值表达式。

expression2

任何数值表达式。

备注

| 运算符将参数强制为匹配的数据类型。然后 **|** 运算符查看两个表达式的二进制表示形式的值，并执行按位“或”运算。参数的数据类型决定此运算符所返回的数据类型。

此运算的结果如下所示：

```
0101   (expression1)
1100   (expression2)
----
1101   (result)
```

任何时候，只要任一表达式的一位为 1，则结果中的该位为 1。否则，结果中的该位为 0。

要求

版本 1

请参见

参考

按位“或”赋值运算符 (|=)

概念

运算符优先级

运算符摘要

按位运算符强制

按位右移运算符 (>>)

右移表达式的位，保持符号不变。

```
expression1 >> expression2
```

参数

expression1

任何数值表达式。

expression2

任何数值表达式。

备注

>> 运算符将 expression1 的所有位右移 expression2 指定的位数。用 expression1 的符号位填充右移后左边空出来的位。右移的位被丢弃。*expression1* 的数据类型决定此运算符所返回的数据类型。

>> 运算符屏蔽 *expression2* 以免 *expression1* 的移位量太大。否则，如果移位量超出 *expression1* 的数据类型中的位数，则可能移走所有的初始位，从而会提供无意义的结果。为了确保每次移位保留至少一个初始位，移位运算符将使用以下公式来计算实际移位量：使用 *expression1* 中的位数减一所得的结果来屏蔽 *expression2*（使用按位"与"运算符）。

示例

例如，计算完下列代码后，*temp* 的值为 -4：因为 -14（即二进制的 11110010）右移两位后等于 -4（即二进制的 11111100）。

```
var temp
temp = -14 >> 2
```

要理解屏蔽的工作方式，请考虑下面的示例。

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
var y : byte = x >> 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in y are 00000011
// The value of y is 3
print(y); // Prints 3
```

要求

版本 1

请参见

参考

按位左移运算符 (<<)
右移赋值运算符 (>>=)
无符号右移运算符 (>>>)

概念

运算符优先级
运算符摘要
按位运算符强制

按位“异或”赋值运算符 (^=)

对变量和表达式执行按位“异或”运算，并将结果赋给该变量。

```
result ^= expression
```

参数

result

任何数值变量。

表达式

任何数值表达式。

备注

使用此运算符的效果基本上与指定 `result = result ^ expression` 相同，不同的只是仅计算一次 *result*。

^= 运算符将这些参数强迫为匹配的数据类型。**^=** 运算符查看两个表达式的二进制表示形式的值，并对其执行按位“异或”运算。此运算的结果如下所示：

```
0101    (result)
1100    (expression)
----
1001    (result)
```

当且仅当只有一个表达式的某位为 1 时，结果中的该位才为 1。否则，结果中的该位为 0。

要求

版本 1

请参见

参考

按位“异或”运算符 (^)

赋值运算符 (=)

概念

运算符优先级

运算符摘要

按位运算符强制

按位“异或”运算符 (^)

对两个表达式执行按位 XOR 运算。

```
expression1 ^ expression2
```

参数

expression1

任何数值表达式。

expression2

任何数值表达式。

备注

^ 运算符将参数强制为匹配的数据类型。^ 运算符随后查看两个表达式的二进制表示形式的值，并执行按位“异或”运算。参数的数据类型决定此运算符所返回的数据类型。

此运算的结果如下所示：

```
0101   (expression1)
1100   (expression2)
----
1001   (result)
```

当且仅当只有一个表达式的某位为 1 时，结果中的该位才为 1。否则，结果中的该位为 0。

要求

版本 1

请参见

参考

按位“异或”赋值运算符 (^=)

概念

运算符优先级

运算符摘要

按位运算符强制

逗号运算符 (,)

顺序执行两个表达式。

```
expression1, expression2
```

参数

expression1

任何表达式。

expression2

任何表达式。

备注

, 运算符使其两侧的表达式按从左到右的顺序执行，并获得右侧表达式的值。, 运算符最常见的用法是在 **for** 循环的递增表达式中使用。例如：

```
var i, j, k;
for (i = 0; i < 10; i++, j++) {
   k = i + j;
}
```

每次通过循环的结尾时，**for** 语句只允许执行单个表达式。, 运算符用于使多个表达式得以被视为单个表达式，从而绕过此限制。

要求

版本 1

请参见

参考

for 语句

概念

运算符优先级

运算符摘要

比较运算符

返回指示比较结果的布尔值。

```
expression1 comparisonoperator expression2
```

参数

expression1

任何表达式。

comparisonoperator

任何比较运算符(<、>、<=、>=、==、!=、===、!==)

expression2

任何表达式。

备注

比较字符串时，JScript 使用字符串表达式的 Unicode 字符值。

下面描述根据 expression1 和 expression2 的类型和值，不同组的运算符是如何起作用的：

关系运算符(<、>、<=、>=)

- 尝试将 expression1 和 expression2 都转换为数字。
- 如果两个表达式均为字符串，则按字典序比较字符串。
- 如果任一表达式为 **NaN**，则返回 **false**。
- 负零等于正零。
- 负无穷小于包括其自身在内的任何数。
- 正无穷大于包括其自身在内的任何数。

相等运算符(==、!=)

- 如果两表达式的类型不同，则尝试将它们转换为字符串、数字或布尔值。
- **NaN** 与包括其自身在内的任何值都不相等。
- 负零等于正零。
- **null** 与 **null** 和 **undefined** 都相等。
- 以下情况被认为是相等的：相同的字符串，数值上相等的数字，同一对象，相同的布尔值，或者当类型不同时可以被强制转换为上述情况之一的值。
- 其他比较都被认为是不等的。

全等运算符(===、!==)

除了不进行类型转换，并且类型必须相同以外，这些运算符与相等运算符的作用是一样的。

要求

版本 1

请参见

概念

运算符优先级

运算符摘要

条件(三元)运算符 (?:)

视情况返回以下两个表达式之一。

```
test ? expression1 : expression2
```

参数

test

任何 Boolean 表达式。

expression1

test 为 **true** 时返回的表达式。可能是逗点表达式。

expression2

test 为 **false** 时返回的表达式。可能是逗点表达式。

备注

?: 运算符可以用作 **if...else** 语句的快捷方式。它通常用作较大表达式(使用 **if...else** 语句会很繁琐)的一部分。例如:

```
var now = new Date();
var greeting = "Good" + ((now.getHours() > 17) ? " evening." : " day.");
```

在此例中,如果晚于下午 6 时,则创建一个包含 "Good evening." 的字符串。使用 **if...else** 语句的等效代码如下:

```
var now = new Date();
var greeting = "Good";
if (now.getHours() > 17)
   greeting += " evening.";
else
   greeting += " day.";
```

要求

版本 1

请参见

参考

if...else 语句

概念

运算符优先级

运算符摘要

delete 运算符

从对象中删除属性，从数组中移除元素或从 IDictionary 对象中移除项。

```
delete expression
```

参数

表达式

必选。任何结果为属性引用、数组元素或 IDictionary 对象的表达式。

备注

如果 expression 的结果是一个对象，并且在 expression 中指定的属性存在，同时该对象不允许此属性被删除，那么将返回 **false**。

在所有其他情况下，返回 **true**。

示例

下面的示例阐释 **delete** 运算符的一种用法。

```
// Make an object with city names and an index letter.
var cities : Object = {"a" : "Athens" , "b" : "Belgrade", "c" : "Cairo"}

// List the elements in the object.
var key : String;
for (key in cities) {
   print(key + " is in cities, with value " + cities[key]);
}

print("Deleting property b");
delete cities.b;

// List the remaining elements in the object.
for (key in cities) {
   print(key + " is in cities, with value " + cities[key]);
}
```

该代码的输出为：

```
a is in cities, with value Athens
b is in cities, with value Belgrade
c is in cities, with value Cairo
Deleting property b
a is in cities, with value Athens
c is in cities, with value Cairo
```

要求

版本 3

请参见

参考

IDictionary Interface

概念

运算符优先级

运算符摘要

除法赋值运算符 (/=)

变量值除以表达式值，并将结果赋给该变量。

```
result /= expression
```

参数
result

任何数值变量。

表达式

任何数值表达式。

备注

使用此运算符的效果基本上与指定 `result = result / expression` 相同，不同的只是仅计算一次 *result*。

要求

版本 1

请参见
参考
除法运算符 (/)
赋值运算符 (=)
概念
运算符优先级
运算符摘要

除法运算符 (/)

将两个表达式的值相除。

```
number1 / number2
```

参数

number1

任何数值表达式。

number2

任何数值表达式。

备注

如果 *number1* 是有限的非零值并且 *number2* 是零，则除法的结果在 *number1* 是正数时为 **Infinity**，在 *number1* 是负数时为 -**Infinity**。如果 *number1* 和 *number2* 都是零，则结果为 **NaN**。

要求

版本 1

请参见

参考

除法赋值运算符 (/=)

概念

运算符优先级

运算符摘要

in 运算符

测试一个对象中是否存在一种属性。

```
property in object
```

参数

property

必选。计算结果为字符串的表达式。

object

必选。任意对象。

备注

in 运算符检查对象中是否有名为 property 的属性。它还检查对象的原型，以便知道 *property* 是否为原型链的一部分。如果 *property* 在对象或原型链中，则 **in** 运算符返回 **true**，否则返回 **false**。

in 运算符不应与 **for...in** 语句混淆。

注意
若要测试对象本身是否有属性，并且不从原型链中继承属性，请使用对象的 **hasOwnProperty** 方法。

示例

下面的示例阐释 **in** 运算符的一种用法。

```
function cityName(key : String, cities : Object) : String {
   // Returns a city name associated with an index letter.
   var ret : String = "Key '" + key + "'";
   if( key in cities )
      return ret + " represents " + cities[key] + ".";
   else  // no city indexed by the key
      return ret + " does not represent a city."
}

// Make an object with city names and an index letter.
var cities : Object = {"a" : "Athens" , "b" : "Belgrade", "c" : "Cairo"}

// Look up cities with an index letter.
print(cityName("a",cities));
print(cityName("z",cities));
```

该代码的输出为：

```
Key 'a' represents Athens.
Key 'z' does not represent a city.
```

要求

版本 1

请参见

参考

for...in 语句

hasOwnProperty 方法

概念

运算符优先级

运算符摘要

递增 (++) 和递减 (--) 运算符

变量值递增 1 或递减 1。

```
//prefix syntax
++variable
--variable
//postfix syntax
variable++
variable--
```

参数
variable

任何数值变量。

备注

递增和递减运算符用作修改存储在变量中的值并访问该值的快捷方式。任一运算符可用在前缀或后缀语法中。

如果	等效操作	返回值
++ *variable*	*variable* += 1	*variable* 在增加之后的值
variable ++	*variable* += 1	*variable* 在增加之前的值
-- *variable*	*variable* -= 1	*variable* 在减小之后的值
variable --	*variable* -= 1	*variable* 在减小之前的值

示例

下面的示例阐释 ++ 运算符的前缀和后缀语法之间的区别。

```
// Example of prefix increment operator
var j1 : int = 2;
var k1 : int;
k1 = ++j1;          // k1 is 3, the value of j1 after incrementing

// Example of postfix increment operator
var j2 : int = 2;
var k2 : int;
k2 = j2++;          // k2 is 2, the value of j2 before incrementing
```

要求

版本 1

请参见
概念
运算符优先级
运算符摘要

instanceof 运算符

返回一个布尔值，该值指示一个对象是否为特定类或构造函数的一个实例。

```
object instanceof class
```

参数

object

必选。任何对象表达式。

class

必选。任何对象类或构造函数。

备注

如果 *object* 是 *class* 或构造函数的实例，则 **instanceof** 运算符返回 **true**。如果 object 不是指定类或函数的实例，或者 object 为 **null**，则返回 **false**。

JScript **Object** 很特殊。当且仅当对象用 **Object** 构造函数构造时，对象才被视为 **Object** 的实例。

示例 1

下面的示例阐释使用 **instanceof** 运算符检查变量的类型。

```
// This program uses System.DateTime, which must be imported.
import System

function isDate(ob) : String {
   if (ob instanceof Date)
      return "It's a JScript Date"
   if (ob instanceof DateTime)
      return "It's a .NET Framework Date"
   return "It's not a date"
}

var d1 : DateTime = DateTime.Now
var d2 : Date = new Date
print(isDate(d1))
print(isDate(d2))
```

该代码的输出为：

```
It's a .NET Date
It's a JScript Date
```

示例 2

下面的示例阐释使用 **instanceof** 运算符检查构造函数的实例。

```
function square(x : int) : int {
   return x*x
}

function bracket(s : String) : String{
   return("[" + s + "]");
}

var f = new square
print(f instanceof square)
print(f instanceof bracket)
```

该代码的输出为:

```
true
false
```

示例 3

下面的示例阐释 instanceof 运算符如何检查对象是否为 Object 的实例。

```
class CDerived extends Object {
   var x : double;
}

var f : CDerived = new CDerived;
var ob : Object = f;
print(ob instanceof Object);

ob = new Object;
print(ob instanceof Object);
```

该代码的输出为:

```
false
true
```

要求

版本 5

请参见

概念

运算符优先级

运算符摘要

左移赋值运算符 (<<=)

变量值根据表达式值指定的位数左移，并将结果赋给该变量。

```
result <<= expression
```

参数
result

任何数值变量。

表达式

任何数值表达式。

备注

使用此运算符的效果基本上与指定 `result = result << expression` 相同，不同的只是仅计算一次 *result*。

<<= 运算符将 result 的所有位左移 expression 所指定的位数。该操作符屏蔽 *expression* 以避免将 *result* 移位太多。否则，如果移位量超出 *result* 的数据类型中的位数，则可能会移走所有的初始位，从而会提供无意义的结果。为了确保每次移位保留至少一个初始位，移位运算符将使用以下公式来计算实际移位量：使用 *result* 中的位数减一所得的结果来屏蔽 *expression*（使用按位"与"运算符）。

示例

例如：

```
var temp
temp = 14
temp <<= 2
```

变量 temp 的值为 56，因为 14（即二进制的 00001110）左移两位等于 56（即二进制的 00111000）。移动时用零填充右边空出的位。

要理解屏蔽的工作方式，请考虑下面的示例。

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
x <<= 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in x are 00111100
// The value of x is 60
print(x); // Prints 60
```

要求

版本 1

请参见
参考
按位左移运算符 (<<)
按位右移运算符 (>>)
无符号右移运算符 (>>>)
赋值运算符 (=)
概念
运算符优先级
运算符摘要

逻辑“与”运算符 (&&)

对两个表达式执行逻辑合取操作。

```
expression1 && expression2
```

参数

expression1

任何表达式。

expression2

任何表达式。

备注

当且仅当两个表达式均为 **true** 时，结果才为 **true**。如果任一表达式为 **false**，则结果为 **false**。下表阐释如何确定结果：

如果 **expression1** 强迫为	且 **expression2** 强迫为	结果为	结果强迫为
true	**true**	expression2	**true**
true	**false**	expression2	**false**
false	**true**	expression1	**false**
false	**false**	expression1	**false**

JScript 使用下列规则将非布尔值转换为布尔值：

- 所有对象都被认为是 **true**。
- 当且仅当字符串为空时才被认为是 **false**。
- **null** 和未定义被认为是假。
- 当且仅当数字为 0 时才为 **false**。

要求

版本 1

请参见

概念

运算符优先级

运算符摘要

逻辑“非”运算符 (!)

对一个表达式执行逻辑求反操作。

```
!expression
```

参数

表达式

任何表达式。

备注

下表阐释如何确定结果：

如果 `expression` 强迫为	则 `result` 为
true	**false**
false	**true**

所有一元运算符（如 ! 运算符）都按照下面的规则来计算表达式的值：

- 如果应用于未定义的表达式或 **null** 表达式，则会引发一个运行时错误。
- 将对象转换为字符串。
- 如果可能，将字符串转换为数字。否则，将引发运行时错误。
- 布尔值被视为数字（如果是**假**则为 0；如果是**真**则为 1）。

运算符将应用于结果数字。

对于 ! 运算符，如果 expression 不为零，则 result 为零。如果 expression 为零，则 result 为 1。

要求

版本 1

请参见

参考

按位“取非”运算符 (~)

概念

运算符优先级

运算符摘要

逻辑“或”运算符 (||)

对两个表达式执行逻辑析取操作。

```
expression1 || expression2
```

参数

expression1

任何表达式。

expression2

任何表达式。

备注

如果两个表达式中至少有一个或者都为 **true**，则结果为 **true**。下表阐释如何确定结果：

如果 expression1 强迫为	且 expression2 强迫为	结果为	结果强迫为
true	**true**	expression1	**true**
true	**false**	expression1	**true**
false	**true**	expression2	**true**
false	**false**	expression2	**false**

JScript 使用下列规则将非布尔值转换为布尔值：

- 所有对象都被认为是 **true**。
- 当且仅当字符串为空时才被认为是 **false**。
- **null** 和未定义被认为是假。
- 当且仅当数字为 0 时才为 **false**。

要求

版本 1

请参见

概念

运算符优先级

运算符摘要

取模赋值运算符 (%=)

变量值除以表达式值，并将余数赋给变量。

```
result %= expression
```

参数
result

任何数值变量。

表达式

任何数值表达式。

备注

使用此运算符的效果基本上与指定 `result = result % expression` 相同，不同的只是仅计算一次 *result*。

要求

版本 1

请参见
参考
取模运算符 (%)
赋值运算符 (=)
概念
运算符优先级
运算符摘要

取模运算符 (%)

一个表达式的值除以另一个表达式的值，并返回余数。

```
number1 % number2
```

参数

number1

任何数值表达式。

number2

任何数值表达式。

备注

取模或余数运算符用 *number2* 除 *number1* 并只返回余数。结果中的符号与 *number1* 的符号相同。结果值在 0 和 *number2* 的绝对值之间。

取模运算符的参数可以为浮点数字，因此 `5.6 % 0.5` 返回 `0.1`。

示例

下面的示例阐释取模运算符的一种用法。

```
var myMoney : int = 128;
var cookiePrice : int = 33;
// Calculate the change if the maximum number of cookies are bought.
var change : int = myMoney % cookiePrice;
// Calculate number of cookies bought.
var numCookies : int = Math.round((myMoney-change)/cookiePrice);
```

要求

版本 1

请参见

参考

取模赋值运算符 (%=)

概念

运算符优先级

运算符摘要

乘法赋值运算符 (*=)

变量值乘以表达式值，并将结果赋给该变量。

```
result *= expression
```

参数
result

任何数值变量。

表达式

任何数值表达式。

备注

使用此运算符的效果基本上与指定 `result = result * expression` 相同，不同的只是仅计算一次 *result*。

要求

版本 1

请参见
参考
乘法运算符 (*)
赋值运算符 (=)
概念
运算符优先级
运算符摘要

JScript 8.0

乘法运算符 (*)

将两个表达式的值相乘。

```
number1 * number2
```

参数
number1

任何数值表达式。

number2

任何数值表达式。

备注

乘法运算符用 *number2* 乘 *number1* 并返回结果。如果有一个数字为 **NaN**，则结果为 **NaN**。**Infinity** 与零相乘的结果是 **NaN**，而 **Infinity** 与任何非零数字（包括 **Infinity**）相乘的结果是 **Infinity**。

要求

版本 1

请参见
参考
乘法赋值运算符 (*=)
概念
运算符优先级
运算符摘要

new 运算符

创建一个新对象。

```
new constructor[( [arguments] )]
```

参数

constructor

必选。对象的结构。若构造函数没有参数，则可省略圆括号。

arguments

可选项。任意传递给新对象的构造函数的参数。

备注

new 运算符执行以下任务：

- 创建一个没有成员的对象。
- 它为该对象调用构造函数，给新创建的对象传递一个引用，作为 **this** 指针。
- 然后，构造函数根据传递给它的参数初始化该对象。

示例

这些示例演示 **new** 运算符的一些用法。

```
var myObject : Object = new Object;
var myArray : Array = new Array();
var myDate : Date = new Date("Jan 5 1996");
```

要求

版本 1

请参见

参考

function 语句

引用运算符 (&)

& 运算符用于将对变量的引用传递给使用引用或输出参数的方法。当控件传递回调用方法时，对方法参数的更改将反映在按引用传递的变量上。

```
&expression
```

参数

expression

传递给方法的变量。

备注

JScript 可以调用使用引用和输出参数的方法，但不能定义它们。

示例

下面的示例阐释 **引用 (&)** 运算符的一种用法。

```
// Define Compute method in C# code.
public class C
{
    public static void Compute(ref int sum, out int product, int a, int b)
    {
        sum = a + b;
        product = a * b;
    }
}

// Call Compute method from your JScript code.
var a : int, b: int;
C.Compute(&a, &b, 2, 3)
print(a);
print(b);
```

请参见

参考

ref（C# 参考）

out（C# 参考）

右移赋值运算符 (>>=)

变量值右移表达式值指定的位数，保持符号不变，并将结果赋给该变量。

```
result >>= expression
```

参数
result

任何数值变量。

表达式

任何数值表达式。

备注

使用此运算符的效果基本上与指定 `result = result >> expression` 相同，不同的只是仅计算一次 *result*。

>>= 运算符将 result 的所有位右移 expression 指定的位数。用 result 的符号位填充右移后左边空出的位。右移的位被丢弃。该操作符屏蔽 *expression* 以避免将 *result* 移位太多。否则，如果移位量超出 *result* 的数据类型中的位数，则可能会移走所有的初始位，从而会提供无意义的结果。为了确保每次移位保留至少一个初始位，移位运算符将使用以下公式来计算实际移位量：使用 *result* 中的位数减一所得的结果来屏蔽 *expression*（使用按位"与"运算符）。

示例

例如，计算完下列代码后，*temp* 的值为 -4：因为 14（即二进制的 11110010）右移两位后等于 -4（即二进制的 11111100）。

```
var temp
temp = -14
temp >>= 2
```

要理解屏蔽的工作方式，请考虑下面的示例。

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
x >>= 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in x are 00000011
// The value of x is 3
print(x); // Prints 3
```

要求

版本 1

请参见
参考
按位左移运算符 (<<)
按位右移运算符 (>>)
无符号右移运算符 (>>>)
赋值运算符 (=)
概念
运算符优先级
运算符摘要
按位运算符强制

减法赋值运算符 (-=)

从变量值中减去表达式值，并将结果赋给该变量。

```
result -= expression
```

参数
result

任何数值变量。

表达式

任何数值表达式。

备注

使用此运算符的效果基本上与指定 `result = result - expression` 相同，不同的只是仅计算一次 *result*。

要求

版本 1

请参见
参考
减法运算符 (-)
赋值运算符 (=)
概念
运算符优先级
运算符摘要

减法运算符 (-)

从一个表达式的值中减去另一个表达式的值，若只有一个表达式时则取其相反数。

```
number1 - number2
```

```
-number
```

参数

number1

任何数值表达式。

number2

任何数值表达式。

number

任何数值表达式。

备注

在语法 1 中，- 运算符是用于计算两个数差值的算术减法运算符。在语法 2 中，- 运算符用作一元求反运算符，指示表达式的负值。

对于语法 2，和所有一元运算符一样，按如下规则计算表达式：

- 如果应用于未定义的表达式或 **null** 表达式，则会引发一个运行时错误。
- 将对象转换为字符串。
- 如果可能，将字符串转换为数字。否则，将引发运行时错误。
- 布尔值被视为数字（如果是假则为 0；如果是真则为 1）。

运算符将应用于结果数字。在语法 2 中，如果结果数字不是零，则 result 与结果数字符号颠倒后的值相等。如果结果数字为零，则 result 为零。

要求

版本 1

请参见

参考

减法赋值运算符 (-=)

概念

运算符优先级

运算符摘要

typeof 运算符

返回一个用于标识表达式的数据类型的字符串。

```
typeof[(]expression[)] ;
```

参数
表达式

必选。任何表达式。

备注

typeof 运算符把类型信息以字符串形式返回。**typeof** 返回八种可能的值："数字类型"、"字符串类型"、"布尔型"、"对象类型"、"函数类型"、"日期类型"、"未定义类型"和"未知类型"。

typeof 语法中的圆括号是可选的。

注意

:JScript 中的所有表达式都有一个 **GetType** 方法。该方法返回表达式的数据类型（而不是表示数据类型的字符串）。**GetType** 方法比 **typeof** 运算符提供更多信息。

示例

下面的示例阐释 **typeof** 运算符的用法。

```
var x : double = Math.PI;
var y : String = "Hello";
var z : int[] = new int[10];

print("The type of x (a double) is " + typeof(x)  );
print("The type of y (a String) is " + typeof(y) );
print("The type of z (an int[]) is " + typeof(z) );
```

该代码的输出为：

```
The type of x (a double) is number

The type of y (a String) is string

The type of z (an int[]) is object
```

要求

版本 1

请参见
参考
GetType
概念
运算符优先级
运算符摘要

无符号右移赋值运算符 (>>>=)

变量值无符号右移表达式值指定的位数，并将结果赋给该变量。

```
result >>>= expression
```

参数
result

任何数值变量。

表达式

任何数值表达式。

备注

使用此运算符的效果基本上与指定 `result = result >>> expression` 相同，不同的只是仅计算一次 *result*。

>>>= 运算符将 result 的所有位右移 expression 指定的位数。用零填充右移后左边空出的位。右移的位被丢弃。该操作符屏蔽 *expression* 以避免将 *result* 移位太多。否则，如果移位量超出 *result* 的数据类型中的位数，则可能会移走所有的初始位，从而会提供无意义的结果。为了确保每次移位保留至少一个初始位，移位运算符将使用以下公式来计算实际移位量：使用 *result* 中的位数减一所得的结果来屏蔽 *expression*（使用按位"与"运算符）。

示例

例如：

```
var temp
temp = -14
temp >>>= 2
```

变量 temp 的值 -14（即二进制的 11111111 11111111 11111111 11110010），右移两位后等于 1073741820（即二进制的 00111111 11111111 11111111 11111100）。

要理解屏蔽的工作方式，请考虑下面的示例。

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
x >>>= 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in x are 00000011
// The value of x is 3
print(x); // Prints 3
```

要求

版本 1

请参见
参考
无符号右移运算符 (>>>)
按位左移运算符 (<<)
按位右移运算符 (>>)
赋值运算符 (=)
概念
运算符优先级
运算符摘要
按位运算符强制

无符号右移运算符 (>>>)

右移表达式的位，不保留符号。

```
expression1 >>> expression2
```

参数

expression1

任何数值表达式。

expression2

任何数值表达式。

备注

>>> 运算符将 expression1 的所有位右移 expression2 指定的位数。用零填充右移后左边空出的位。右移的位被丢弃。*expression1* 的数据类型决定此运算符所返回的数据类型。

>>> 运算符屏蔽 *expression2* 以免 *expression1* 的移位量太大。否则，如果移位量超出 *expression1* 的数据类型中的位数，则可能移走所有的初始位，从而会提供无意义的结果。为了确保每次移位保留至少一个初始位，移位运算符将使用以下公式来计算实际移位量：使用 *expression1* 中的位数减一所得的结果来屏蔽 *expression2*（使用按位"与"运算符）。

示例

例如：

```
var temp
temp = -14 >>> 2
```

变量 temp 的值 -14（即二进制的 11111111 11111111 11111111 11110010），右移两位后等于 1073741820（即二进制的 00111111 11111111 11111111 11111100）。

要理解屏蔽的工作方式，请考虑下面的示例。

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
var y : byte = x >>> 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in y are 00000011
// The value of y is 3
print(y); // Prints 3
```

要求

版本 1

请参见

参考

无符号右移赋值运算符 (>>>=)

按位左移运算符 (<<)

按位右移运算符 (>>)

概念

运算符优先级

运算符摘要

按位运算符强制

void 运算符

禁止表达式返回值。

```
void expression
```

参数

表达式

必选。任何表达式。

备注

void 运算符计算其表达式，并返回未定义。当希望计算表达式，但又不希望脚本的其他部分看见其结果时，该运算符最有用。

要求

版本 2

请参见

概念

运算符优先级

运算符摘要

属性 (JScript)

属性是作为对象成员的一个值或一组值(以数组或对象的形式)。以下各节链接到有关解释如何使用 JScript 中属性的信息。

本节内容

0...n 属性

返回 **arguments** 对象中的各个参数的实际值，该值是由一个正在执行的函数的 **arguments** 属性返回的。

$1...$9 属性

返回在模式匹配期间找到的，所存储的最近的九个部分。

arguments 属性

返回当前所执行的"函数"(Function) object.caller 属性的"参数"(arguments) 对象。

callee 属性

返回正被执行的 **Function** 对象，即指定的 **Function** 对象的正文。

caller 属性

返回一个对函数的引用，该函数调用了当前函数。

constructor 属性

指定创建一个对象的函数。

description 属性

返回或设置与特定错误相关联的描述性字符串。

E 属性

返回数学常数 *e*，即自然对数的底。

global 属性

返回布尔值，该值指示使用正则表达式的 global 标志 (**g**) 的状态。

ignoreCase 属性

返回布尔值，该值指示在正则表达式中使用的 ignoreCase 标志 (**i**) 的状态。

index 属性

返回字符位置，它是被搜索字符串中第一个成功匹配的开始位置。

Infinity 属性

返回 Number.POSITIVE_INFINITY 的初始值。

input 属性 ($_)

返回执行正则表达式搜索所针对的字符串。

lastIndex 属性

返回字符的位置，该位置是被搜索字符串中下一次匹配的开始位置。

lastMatch 属性 ($&)

返回来自任何正则表达式搜索过程的最后匹配的字符。

lastParen 属性 ($+)

如果有的话，返回来自任何正则表达式搜索过程中最后带括号的子匹配。

leftContext 属性 ($`)

返回从一个被搜索字符串的开头到最后一个匹配的开头之前的位置之间的字符。

length 属性(参数)

返回由调用者传递给一个函数的实际参数数目。

length 属性(数组)

返回一个整数值，此整数比数组中所定义的最高位元素的下标大 1。

length 属性(函数)

返回为一个函数定义的参数数目。

length 属性(字符串)

返回 **String** 对象的长度。

LN10 属性

返回 10 的自然对数。

LN2 属性

返回 2 的自然对数。

LOG10E 属性

返回 *e*(自然对数的底)的以 10 为底的对数。

LOG2E 属性

返回 *e*(自然对数的底)的以 2 为底的对数。

MAX_VALUE 属性

返回可用 JScript 表示的最大的数。约等于 1.79E+308。

message 属性

返回一个错误信息的字符串。

MIN_VALUE 属性

返回 JScript 能够表示的最接近零的数字。约等于 5.00E-324。

multiline 属性

返回布尔值，该值指示在正则表达式中使用的 multiline 标志 (**m**) 的状态。

name 属性

返回一个名称，该名称属于一个错误。

NaN 属性

一个特殊值，此值指示算术表达式返回了非数字值。

NaN 属性(全局)

返回特殊值 **NaN**，指示一个表达式不是数字。

NEGATIVE_INFINITY 属性

返回一个超过 JScript 可表示的最大负数 (**-Number.MAX_VALUE**) 范围的负值。

number 属性

返回一个超过 JScript 可表示的最大负数 (**-Number.MAX_VALUE**) 范围的负值。

PI 属性

返回圆的周长与其直径的比值，约等于 3.141592653589793。

POSITIVE_INFINITY 属性

返回一个大于 JScript 所能表示的最大数字 (**Number.MAX_VALUE**) 的值。

propertyIsEnumerable 属性

返回布尔值，该值指示指定属性是否为对象的一部分以及该属性是否是可枚举的。

prototype 属性

为对象的类返回原型的引用。

rightContext 属性 ($')

返回一些字符，即从最后一个匹配之后的位置直到被搜索字符串的结尾处的字符。

source 属性

返回正则表达式模式的文本的一个副本。

SQRT1_2 属性

返回 0.5 的平方根，即 2 的平方根分之一。

SQRT2 属性

返回 2 的平方根

undefined 属性

返回 **undefined** 的一个初始值。

相关章节

JScript 参考

列出"JScript 语言参考"所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

.NET Framework 参考

列出指向特定主题的链接，这些主题解释 .NET Framework 类库和其他基本元素的语法和结构。

0...n 属性

返回 **arguments** 对象中的各个参数的实际值，该值是由一个正在执行的函数的 **arguments** 属性返回的。

```
[function.]arguments[[n]]
```

参数

function

可选项。当前正在执行的 **Function** 对象的名称。

n

必选。在 0 到 **arguments.length-1** 范围内的非负整数，其中 0 表示第一个参数，**arguments.length-1** 表示最后一个参数。

备注

由 0...*n* 属性返回的值就是传递给正在执行的函数的值。虽然 **arguments** 对象不是数组，但访问组成 **arguments** 对象的各个元素的方法与访问数组元素的方法相同。

注意

arguments 对象在以快速模式(JScript 的默认模式)运行时不可用。若要从命令行编译使用 **arguments** 对象的程序，则必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。有关更多信息，请参见 arguments 对象。

示例

下面的示例阐释了 **arguments** 对象的 0...n 属性的用法。

```
function argTest(){
   var s = "";
   s += "The individual arguments are:\n"
   for (var n=0; n< arguments.length; n++){
      s += "argument " + n;
      s += " is " + argTest.arguments[n] + "\n";
   }
   return(s);
}
print(argTest(1, 2, "hello", new Date()));
```

当使用 **/fast-** 选项编译该程序后，该程序的输出为：

```
The individual arguments are:
argument 0 is 1
argument 1 is 2
argument 2 is hello
argument 3 is Sat Jan 1 00:00:00 PST 2000
```

要求

版本 5.5

应用于：

arguments 对象 | Function 对象

请参见

其他资源

属性 (JScript)

$1...$9 属性

返回在模式匹配期间找到的，所存储的最近的九个部分。只读。

```
RegExp.$n
```

参数

RegExp

必选。全局 **RegExp** 对象。

n

必选。1 至 9 之间的整数。

备注

每当产生一个带括号的成功匹配时，**$1...$9** 属性的值就被修改。可以在一个正则表达式模式中指定任意多个带括号的子匹配，但只能存储最新的九个。

注意

以快速模式（JScript 的默认模式）运行时，**RegExp** 对象的属性不可用。若要从命令行编译使用这些属性的程序，必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

示例

下面的示例阐释了 $1...$9 属性的用法：

```
var s : String;
var re : RegExp = new RegExp("d(b+)(d)","ig");
var str : String = "cdbBdbsbdbdz";
var arr : Array = re.exec(str);
s = "$1 contains: " + RegExp.$1 + "\n";
s += "$2 contains: " + RegExp.$2 + "\n";
s += "$3 contains: " + RegExp.$3;
print(s);
```

当使用 **/fast-** 选项编译后，该程序的输出为：

```
$1 contains: bB
$2 contains: d
$3 contains:
```

要求

版本 1

应用于：

RegExp 对象

请参见

概念

正则表达式语法

arguments 属性

为当前执行中的 **Function** 对象返回 arguments 对象。

```
[function.]arguments
```

参数
function

可选项。当前正在执行的 **Function** 对象的名称。

备注

arguments 属性允许函数处理可变数量的参数。arguments 对象的 length 属性包含了传递给函数的参数数目。arguments 对象中包含的各个参数的访问方式与数组元素的访问方式相同。

注意

arguments 对象在以快速模式(JScript 的默认模式)运行时不可用。若要从命令行编译使用 **arguments** 对象的程序，则必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。有关更多信息，请参见 arguments 对象。

示例

下面的示例阐释了 arguments 属性的用法：

```
function argTest(){
   var s = "";
   s += "The individual arguments are:\n"
   for (var n=0; n< arguments.length; n++){
      s += "argument " + n;
      s += " is " + argTest.arguments[n] + "\n";
   }
   return(s);
}
print(argTest(1, 2, "hello", new Date()));
```

当使用 **/fast-** 选项编译该程序后，该程序的输出为：

```
The individual arguments are:
argument 0 is 1
argument 1 is 2
argument 2 is hello
argument 3 is Sat Jan 1 00:00:00 PST 2000
```

要求

版本 2

应用于：

Function 对象

请参见
参考
arguments 对象
function 语句

callee 属性

返回正被执行的 **Function** 对象，即指定的 **Function** 对象的正文。

```
[function.]arguments.callee
```

参数

function

可选项。当前正在执行的 **Function** 对象的名称。

备注

callee 属性是 **arguments** 对象的一个成员，该属性仅当相关函数正在执行时才可用。

callee 属性的初始值是正被执行的 **Function** 对象。这将允许匿名函数成为递归的。

注意

arguments 对象在以快速模式(JScript 的默认模式)运行时不可用。若要从命令行编译使用 **arguments** 对象的程序，则必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。有关更多信息，请参见 arguments 对象。

示例

下面的示例阐释了 **callee** 属性的用法。

```
function factorial(n) {
   if (n <= 0)
      return 1;
   else
      return n * arguments.callee(n - 1)
}
print(factorial(3));
```

当使用 **/fast-** 选项编译该程序后，该程序的输出为：

```
6
```

要求

版本 5.5

应用于：

arguments 对象 | Function 对象

请参见

参考

function 语句

caller 属性

返回一个对函数的引用，该函数调用了当前函数。

```
function.caller
```

参数
function

必选。当前正在执行的 **Function** 对象的名称。

备注

caller 属性只有当函数正在执行时才被定义。如果函数是从 JScript 程序的顶层调用的，则 **caller** 包含 **null**。

如果在字符串上下文中使用 **caller** 属性，则其结果和 *functionName*.**toString** 相同，就是说，将显示函数的反编译文本。

注意

caller 属性在以快速模式(JScript 的默认模式)运行时不可用。若要从命令行编译使用 **caller** 属性的程序，必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

示例

下面的示例阐释了 **caller** 属性的用法。

```
function callLevel(){
   if (callLevel.caller == null)
      print("callLevel was called from the top level.");
   else {
      print("callLevel was called by:");
      print(callLevel.caller);
   }
}
function testCall() {
   callLevel()
}
// Call callLevel directly.
callLevel();
// Call callLevel indirectly.
testCall();
```

当使用 **/fast-** 选项编译该程序后，该程序的输出为：

```
callLevel was called from the top level.
callLevel was called by:
function testCall() {
   callLevel()
}
```

要求

版本 2

应用于：

arguments 对象 | Function 对象

请参见
参考
function 语句

constructor 属性

指定创建一个对象的函数。

```
object.constructor
```

参数
object

必选。对象或函数的名称。

备注

constructor 属性是每个具有原型的对象的原型成员。这包括除了 **arguments**、**Enumerator**、**Error**、**Global**、**Math**、**RegExp**、**Regular Expression** 和 **VBArray** 对象以外的所有内部 JScript 对象。**constructor** 属性包含了对某种函数的引用，此种函数构造了特定对象的实例。

基于类的对象没有 **constructor** 属性。

示例

下面的示例阐释了 **constructor** 属性的用法。

```
function testObject(ob) {
   if (ob.constructor == String)
      print("Object is a String.");
   else if (ob.constructor == MyFunc)
      print("Object is constructed from MyFunc.");
   else
      print("Object is neither a String or constructed from MyFunc.");
}
// A constructor function.
function MyFunc() {
   // Body of function.
}

var x = new String("Hi");
testObject(x)
var y = new MyFunc;
testObject(y);
```

该程序的输出为：

```
Object is a String.
Object is constructed from MyFunc.
```

要求

版本 2

应用于：

Array 对象 | Boolean 对象 | Date 对象 | Function 对象 | Number 对象 | Object 对象 | String 对象

请参见
参考
prototype 属性

JScript 8.0

description 属性

返回或设置与特定错误相关联的描述性字符串。

```
object.description
```

参数

object

必选。**Error** 对象的实例。

备注

description 属性是一个字符串，该字符串包含与特定错误相关的错误信息。使用该属性中包含的值向用户发出一个错误警报，指出脚本无法处理。

description 和 **message** 属性引用相同的消息；**description** 属性提供向后兼容性，而 **message** 属性符合 ECMA 标准。

示例

下面的示例导致引发了异常，并显示了错误的说明。

```
function getAge(age) {
   if(age < 0)
      throw new Error("An age cannot be negative.")
   print("Age is "+age+".");
}

// Pass the getAge an invalid argument.
try {
   getAge(-5);
} catch(e) {
   print(e.description);
}
```

该代码的输出为：

```
An age cannot be negative.
```

要求

版本 5

应用于：

错误对象

请参见

参考

number 属性
message 属性 (JScript)
name 属性

JScript 8.0

E 属性

返回数学常数 *e*，即自然对数的底。

```
Math.E
```

参数

Math

必选。全局 **Math** 对象。

备注

E 属性约等于 2.718。

要求

版本 1

应用于：

Math 对象

请参见

参考

exp 方法

global 属性

返回布尔值，该值指示使用正则表达式的 global 标志 (**g**) 的状态。

```
rgExp.global
```

参数

rgExp

必选。**Regular Expression** 对象的一个实例。

备注

global 属性是只读的，并且如果 global 标志是为正则表达式设置的，则返回 **true**，否则返回 **false**。默认值为 **false**。

当使用 global 标志时，它将指示一个搜索操作应找到被搜索字符串中的所有模式，而不仅仅是第一个。这也称为全局匹配。

示例

下面的示例阐释了 **global** 属性的用法。

```
function RegExpPropDemo(re : RegExp) {
   print("Regular expression: " + re);
   print("global:     " + re.global);
   print("ignoreCase: " + re.ignoreCase);
   print("multiline:  " + re.multiline);
   print();
};

// Some regular expression to test the function.
var re1 : RegExp = new RegExp("the","i");  // Use the constructor.
var re2 = /\w+/gm;                         // Use a literal.
RegExpPropDemo(re1);
RegExpPropDemo(re2);
RegExpPropDemo(/^\s*$/im);
```

该程序的输出为：

```
Regular expression: /the/i
global:     false
ignoreCase: true
multiline:  false

Regular expression: /\w+/gm
global:     true
ignoreCase: false
multiline:  true

Regular expression: /^\s*$/im
global:     false
ignoreCase: true
multiline:  true
```

要求

版本 5.5

应用于：

正则表达式对象

请参见

参考

ignoreCase 属性

multiline 属性
概念
正则表达式语法

ignoreCase 属性

返回布尔值，该值指示在正则表达式中使用的 ignoreCase 标志 (**i**) 的状态。

```
rgExp.ignoreCase
```

参数

rgExp

必选。**Regular Expression** 对象的一个实例。

备注

ignoreCase 属性是只读的，并且如果 ignoreCase 标志是为正则表达式设置的，则返回 **true**，否则返回 **false**。默认值为 **false**。

如果使用了 ignoreCase 标志，则该标志将指示被搜索字符串中执行模式匹配的一个搜索应该不区分大小写。

示例

下面的示例阐释了 **ignoreCase** 属性的用法。

```
function RegExpPropDemo(re : RegExp) {
   print("Regular expression: " + re);
   print("global:     " + re.global);
   print("ignoreCase: " + re.ignoreCase);
   print("multiline:  " + re.multiline);
   print();
};

// Some regular expression to test the function.
var re1 : RegExp = new RegExp("the","i");  // Use the constructor.
var re2 = /\w+/gm;                         // Use a literal.
RegExpPropDemo(re1);
RegExpPropDemo(re2);
RegExpPropDemo(/^\s*$/im);
```

该程序的输出为：

```
Regular expression: /the/i
global:     false
ignoreCase: true
multiline:  false

Regular expression: /\w+/gm
global:     true
ignoreCase: false
multiline:  true

Regular expression: /^\s*$/im
global:     false
ignoreCase: true
multiline:  true
```

要求

版本 5.5

应用于：

正则表达式对象

请参见

参考

global 属性

multiline 属性
概念
正则表达式语法

index 属性

返回字符位置，它是被搜索字符串中第一个成功匹配的开始位置。

```
{RegExp | reArray}.index
```

参数

RegExp

必选。全局 **RegExp** 对象。

reArray

必选。**Regular Expression** 对象的 **exec** 方法所返回的数组。

备注

index 属性是从零开始的。

RegExp.index 属性的初始值为 -1。它的值是只读的，而且每当成功实现匹配时就会更改。

注意

以快速模式（JScript 的默认模式）运行时，**RegExp** 对象的属性不可用。若要从命令行编译使用这些属性的程序，必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

示例

下面的示例阐释了 **index** 属性的用法。该函数重复一个字符串搜索，并打印出字符串中每一个词的 **index** 和 **lastIndex** 值。

```
var src : String = "The rain in Spain falls mainly in the plain.";
var re : RegExp = /\w+/g;
var arr : Array;
while ((arr = re.exec(src)) != null)
   print(arr.index + "-" + arr.lastIndex + "\t" + arr);
```

该程序的输出为：

```
0-3     The
4-8     rain
9-11    in
12-17   Spain
18-23   falls
24-30   mainly
31-33   in
34-37   the
38-43   plain
```

要求

版本 3

应用于：

RegExp 对象

请参见

参考

exec 方法

概念

正则表达式语法

JScript 8.0

Infinity 属性

返回 **Number.POSITIVE_INFINITY** 的值。

```
Infinity
```

备注

Infinity 属性是 Global 对象的成员，在 Scripting 引擎初始化时变为可用。

要求

版本 3

应用于:

Global 对象

请参见

参考

POSITIVE_INFINITY 属性
NEGATIVE_INFINITY 属性

input 属性 ($_)

返回执行正则表达式搜索所针对的字符串。

```
//Syntax 1
{RegExp | reArray}.input

//Syntax 2
RegExp.$_
//The $_ property may be used as shorthand for the input property
//for the RegExp object.
```

参数

RegExp

必选。全局 **RegExp** 对象。

reArray

必选。**Regular Expression** 对象的 **exec** 方法所返回的数组。

备注

input 属性的值是据此来执行正则表达式搜索的字符串。

RegExp.input 属性的初始值为空字符串 ""。它的值是只读的，并且每当执行成功的匹配时就会更改。

注意

以快速模式(JScript 的默认模式)运行时，**RegExp** 对象的属性不可用。若要从命令行编译使用这些属性的程序，必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

示例

下面的示例阐释了 input 属性的用法：

```
var str : String = "A test string.";
var re : RegExp = new RegExp("\\w+","ig");
var arr : Array = re.exec(str);
print("The string used for the match was: " + arr.input);
```

该程序的输出为：

```
The string used for the match was: A test string.
```

要求

版本 3

应用于：

RegExp 对象

请参见

参考

exec 方法

概念

正则表达式语法

lastIndex 属性

返回字符的位置，该位置是被搜索字符串中下一次匹配的开始位置。

```
{RegExp | reArray}.lastIndex
```

参数

RegExp

必选。全局 **RegExp** 对象。

reArray

必选。**Regular Expression** 对象的 **exec** 方法所返回的数组。

备注

lastIndex 属性是从零开始的，也就是说，第一个字符的索引是零。其初始值为 -1。无论何时产生一个成功匹配，其值都被修改。

RegExp 对象的 **lastIndex** 属性是由 **RegExp** 对象的 **exec** 和 **test** 方法以及 **String**对象的 **match**、**replace** 和 **split** 方法修改的。

下面的规则适用于 **lastIndex** 的值：

- 如果没有匹配，则 **lastIndex** 被设置为 -1。
- 如果 **lastIndex** 大于字符串的长度，则 **test** 和 **exec** 失败，并且 **lastIndex** 被设置为 -1。
- 如果 **lastIndex** 等于字符串的长度，且模式与空字符串匹配，则正则表达式匹配。否则，匹配失败并且 **lastIndex** 被重新设置为 -1。
- 否则，**lastIndex** 被设置为紧接最近的匹配的下一个位置。

RegExp.lastIndex 属性的初始值为 -1。它的值是只读的，并且每当成功实现匹配时就会更改。

注意

以快速模式（JScript 的默认模式）运行时，**RegExp** 对象的属性不可用。若要从命令行编译使用这些属性的程序，必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

示例

下面的示例阐释了 **lastIndex** 属性的用法。该函数重复一个字符串搜索，并打印出字符串中每一个词的 **index** 和 **lastIndex** 值。

```
var src : String = "The rain in Spain falls mainly in the plain.";
var re : RegExp = /\w+/g;
var arr : Array;
while ((arr = re.exec(src)) != null)
   print(arr.index + "-" + arr.lastIndex + "\t" + arr);
```

该程序的输出为：

```
0-3     The
4-8     rain
9-11    in
12-17   Spain
18-23   falls
24-30   mainly
31-33   in
34-37   the
38-43   plain
```

要求

版本 3

应用于:

RegExp 对象

请参见

参考

exec 方法

概念

正则表达式语法

lastMatch 属性 ($&)

返回来自任何正则表达式搜索过程的最后匹配的字符。只读。

```
RegExp.lastMatch
```

参数

RegExp

必选。全局 **RegExp** 对象。

备注

lastMatch 属性的初始值是空字符串。每当产生成功匹配时，**lastMatch** 属性的值就会相应更改。

注意

以快速模式(JScript 的默认模式)运行时，**RegExp** 对象的属性不可用。若要从命令行编译使用这些属性的程序，必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

lastMatch 属性的缩写是 **$&**。表达式 **RegExp["$&"]** 和 **RegExp.lastMatch** 可交换使用。

示例

下面的示例阐释了 **lastMatch** 属性的用法：

```
var s;                                //Declare variable.
var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
var str = "cdbBdbsbdbdz";             //String to be searched.
var arr = re.exec(str);               //Perform the search.
s = "$1 returns: " + RegExp.$1 + "\n";
s += "$2 returns: " + RegExp.$2 + "\n";
s += "$3 returns: " + RegExp.$3 + "\n";
s += "input returns : " + RegExp.input + "\n";
s += "lastMatch returns: " + RegExp.lastMatch + "\n";
s += "leftContext returns: " + RegExp.leftContext + "\n";
s += "rightContext returns: " + RegExp.rightContext + "\n";
s += "lastParen returns: " + RegExp.lastParen + "\n";
print(s);                            //Return results.
```

当使用 **/fast-** 选项编译该程序后，该程序的输出为：

```
$1 returns: bB
$2 returns: d
$3 returns:
input returns : cdbBdbsbdbdz
lastMatch returns: dbBd
leftContext returns: c
rightContext returns: bsbdbdz
lastParen returns: d
```

要求

版本 5.5

应用于：

RegExp 对象

请参见

参考

$1...$9 属性
index 属性

lastParen 属性 ($+)

如果有的话，返回来自任何正则表达式搜索过程中最后带括号的子匹配。只读。

```
RegExp.lastParen
```

参数

RegExp

必选。全局 **RegExp** 对象。

备注

lastParen 属性的初始值是空字符串。每当产生成功匹配时，**lastParen** 属性的值就会相应更改。

注意

以快速模式（JScript 的默认模式）运行时，**RegExp** 对象的属性不可用。若要从命令行编译使用这些属性的程序，必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

lastParen 属性的缩写是 **$+**。表达式 **RegExp["$+"]** 和 **RegExp.lastParen** 可交换使用。

示例

下面的示例阐释了 **lastParen** 属性的用法：

```
var s;                                //Declare variable.
var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
var str = "cdbBdbsbdbdz";             //String to be searched.
var arr = re.exec(str);               //Perform the search.
s = "$1 returns: " + RegExp.$1 + "\n";
s += "$2 returns: " + RegExp.$2 + "\n";
s += "$3 returns: " + RegExp.$3 + "\n";
s += "input returns : " + RegExp.input + "\n";
s += "lastMatch returns: " + RegExp.lastMatch + "\n";
s += "leftContext returns: " + RegExp.leftContext + "\n";
s += "rightContext returns: " + RegExp.rightContext + "\n";
s += "lastParen returns: " + RegExp.lastParen + "\n";
print(s);                            //Return results.
```

当使用 **/fast-** 选项编译该程序后，该程序的输出为：

```
$1 returns: bB
$2 returns: d
$3 returns:
input returns : cdbBdbsbdbdz
lastMatch returns: dbBd
leftContext returns: c
rightContext returns: bsbdbdz
lastParen returns: d
```

要求

版本 5.5

应用于：

RegExp 对象

请参见

参考

$1...$9 属性

index 属性

JScript 8.0

leftContext 属性 ($`)

返回从一个被搜索字符串的开头到最后一个匹配的开头之前的位置之间的字符。只读。

```
RegExp.leftContext
```

参数

RegExp

必选。全局 **RegExp** 对象。

备注

leftContext 属性的初始值是空字符串。每当产生成功匹配时，**leftContext** 属性的值就会相应更改。

注意

以快速模式(JScript 的默认模式)运行时，**RegExp** 对象的属性不可用。若要从命令行编译使用这些属性的程序，必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

leftContext 属性的缩写是 **$`**。表达式 **RegExp["$`"]** 和 **RegExp.leftContext** 可交换使用。

示例

下面的示例阐释了 **leftContext** 属性的用法：

```
var s;                                //Declare variable.
var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
var str = "cdbBdbsbdbdz";             //String to be searched.
var arr = re.exec(str);               //Perform the search.
s = "$1 returns: " + RegExp.$1 + "\n";
s += "$2 returns: " + RegExp.$2 + "\n";
s += "$3 returns: " + RegExp.$3 + "\n";
s += "input returns : " + RegExp.input + "\n";
s += "lastMatch returns: " + RegExp.lastMatch + "\n";
s += "leftContext returns: " + RegExp.leftContext + "\n";
s += "rightContext returns: " + RegExp.rightContext + "\n";
s += "lastParen returns: " + RegExp.lastParen + "\n";
print(s);                            //Return results.
```

当使用 **/fast-** 选项编译该程序后，该程序的输出为：

```
$1 returns: bB
$2 returns: d
$3 returns:
input returns : cdbBdbsbdbdz
lastMatch returns: dbBd
leftContext returns: c
rightContext returns: bsbdbdz
lastParen returns: d
```

要求

版本 5.5

应用于：

RegExp 对象

请参见

参考

$1...$9 属性

index 属性

length 属性（参数）

返回由调用者传递给一个函数的实际参数数目。

```
[function.]arguments.length
```

参数

function

可选项。当前正在执行的 **Function** 对象的名称。

备注

对象的 **length** 属性被脚本引擎初始化为该函数中开始执行时，传递给 **Function** 对象的实际参数数目。

示例

下面的示例阐释了 **arguments** 对象的 **length** 属性的用法。

```
function argTest(a, b) : String {
   var i : int;
   var s : String = "The argTest function expected ";
   var numargs : int = arguments.length; // Get number of arguments passed.
   var expargs : int = argTest.length;   // Get number of arguments expected.
   if (expargs < 2)
      s += expargs + " argument. ";
   else
      s += expargs + " arguments. ";
   if (numargs < 2)
      s += numargs + " was passed.";
   else
      s += numargs + " were passed.";
   s += "\n"
   for (i =0 ; i < numargs; i++){        // Get argument contents.
      s += "  Arg " + i + " = " + arguments[i] + "\n";
   }
   return(s);                            // Return list of arguments.
}

print(argTest(42));
print(argTest(new Date(1999,8,7),"Sam",Math.PI));
```

当使用 **/fast-** 选项编译后，该程序的输出为：

```
The argTest function expected 2 arguments. 1 was passed.
  Arg 0 = 42

The argTest function expected 2 arguments. 3 were passed.
  Arg 0 = Tue Sep 7 00:00:00 PDT 1999
  Arg 1 = Sam
  Arg 2 = 3.141592653589793
```

要求

版本 5.5

应用于：

arguments 对象

请参见

参考

arguments 属性
length 属性（数组）

length 属性（字符串）

length 属性（数组）

返回一个整数值，此整数比数组中所定义的最高位元素的下标大 1。

```
arrayObj.length
```

参数
arrayObj

必选。任意 **Array** 对象。

备注

由于 JScript 数组中的元素不必连续，因此 **length** 属性不一定是该数组中的元素数。

如果 length 属性被赋了一个比原先值小的数值，则该数组将被截断，所有数组下标等于或者大于新 length 属性值的元素都将丢失。

如果为 **length** 属性赋一个比先前值大的值，则该数组在形式上被扩展，但不创建新元素。

示例

下面的示例阐释了 length 属性的用法。声明一个数组，为其添加两个元素。由于该数组中的最大索引为 6，因此长度为 7。

```
var my_array : Array = new Array();
my_array[2] = "Test";
my_array[6] = "Another Test";
print(my_array.length); // Prints 7.
```

要求

版本 2

应用于：

Array 对象

请参见
参考
length 属性（函数）
length 属性（字符串）

length 属性（函数）

返回为一个函数定义的参数数目。

```
function.length
```

参数
function

必选。当前正在执行的 **Function** 对象的名称。

备注

当创建一个函数的实例时，函数的 length 属性由脚本引擎初始化为该函数定义中的参数数目。

当一个函数被调用，其参数数目不同于其 length 属性的值时，所发生的情况将依赖于函数本身。

示例

下面的示例阐释了 length 属性的用法：

```
function argTest(a, b) : String {
   var s : String = "The argTest function expected " ;
   var expargs : int = argTest.length;
   s += expargs;
   if (expargs < 2)
      s += " argument.";
   else
      s += " arguments.";
   return(s);
}
// Display the function output.
print(argTest(42,"Hello"));
```

该程序的输出为：

```
The argTest function expected 2 arguments.
```

要求

版本 2

应用于：

Function 对象

请参见
参考
arguments 属性
length 属性（数组）
length 属性（字符串）

JScript 8.0

length 属性（字符串）

返回字符串的长度。

```
str.length
```

参数
str

必选。一个字符串或 **String** 对象的名称。

备注

length 属性包含一个整数，该整数指示 **String** 对象中的字符数。String 对象中的最后一个字符的索引为 length - 1。

要求

版本 1

应用于：

String 对象

请参见
参考
length 属性（数组）
length 属性（函数）

LN10 属性

返回 10 的自然对数。

```
Math.LN10
```

参数
Math

必选。全局 **Math** 对象。

备注

LN10 属性约等于 2.302。

要求

版本 1

应用于:

Math 对象

请参见
参考
length 属性（数组）
length 属性（函数）

LN2 属性

返回 2 的自然对数。

```
Math.LN2
```

参数
Math

必选。全局 **Math** 对象。

备注

LN2 属性约等于 0.693。

要求

版本 1

应用于:

Math 对象

请参见
参考
length 属性（数组）
length 属性（函数）

JScript 8.0

LOG10E 属性

返回 *e*（自然对数的底）的以 10 为底的对数。

```
Math.LOG10E
```

参数
Math

必选。全局 **Math** 对象。

备注

LOG10E 属性为一个常数，约等于 0.434。

要求

版本 1

应用于：

Math 对象

请参见
参考
length 属性（数组）
length 属性（函数）

JScript 8.0

LOG2E 属性

返回 *e*（自然对数的底）的以 2 为底的对数。

```
Math.LOG2E
```

参数
Math

必选。全局 **Math** 对象。

备注

LOG2E 属性为一个常数，约等于 1.442。

要求

版本 1

应用于：

Math 对象

请参见
参考
length 属性（数组）
length 属性（函数）

MAX_VALUE 属性

返回可用 JScript 表示的最大的数。约等于 1.79E+308。

```
Number.MAX_VALUE
```

参数

Number

必选。全局 **Number** 对象。

备注

Number 对象并不是必须在可访问 **MAX_VALUE** 属性之前创建。

要求

版本 2

应用于:

Number 对象

请参见

参考

MIN_VALUE 属性
NaN 属性
NEGATIVE_INFINITY 属性
POSITIVE_INFINITY 属性
toString 方法

message 属性 (JScript)

返回一个错误信息的字符串。

```
errorObj.message
```

参数
errorObj

必选。**Error** 对象的实例。

备注

message 属性是一个包含与特定错误相关的错误信息的字符串。使用包含在此属性中的值向用户发出警报，指出发生了一个您不能或不想处理的错误。

description 和 **message** 属性引用相同的消息；**description** 属性提供向后兼容性，而 **message** 属性符合 ECMA 标准。

示例

下面的示例导致引发了异常，并显示了错误信息。

```
function getAge(age) {
   if(age < 0)
      throw new Error("An age cannot be negative.")
   print("Age is "+age+".");
}

// Pass the getAge an invalid argument.
try {
   getAge(-5);
} catch(e) {
   print(e.message);
}
```

该代码的输出为：

```
An age cannot be negative.
```

要求

版本 5.5

应用于：

错误对象

请参见
参考
description 属性
name 属性

MIN_VALUE 属性

返回可在 JScript 中表示的最接近零的数。约等于 5.00E-324。

```
Number.MIN_VALUE
```

参数

Number

必选。全局 **Number** 对象。

备注

Number 对象并不是必须在可访问 **MIN_VALUE** 属性之前创建。

要求

版本 2

应用于:

Number 对象

请参见

参考

MAX_VALUE 属性
NaN 属性
NEGATIVE_INFINITY 属性
POSITIVE_INFINITY 属性
toString 方法

multiline 属性

返回布尔值，该值指示在正则表达式中使用的 multiline 标志 (**m**) 的状态。

```
rgExp.multiline
```

参数

rgExp

必选。**Regular Expression** 对象的一个实例。

备注

multiline 属性是只读的，并且如果 multiline 标志是为正则表达式设置的，则返回 **true**，否则返回 **false**。如果使用 **m** 标志创建正则表达式对象，那么 **multiline** 属性就是 **true**。默认值为 **false**。

如果 **multiline** 为 **false**，那么“^”与字符串的开始位置相匹配，而“$”与字符串的结束位置相匹配。如果 **multiline** 为 **true**，那么“^”与字符串开始位置以及“\n”或“\r”之后的位置相匹配，而“$”与字符串结束位置以及“\n”或“\r”之前的位置相匹配。

示例

下面的示例阐释了 multiline 属性的用法：

```
function RegExpPropDemo(re : RegExp) {
   print("Regular expression: " + re);
   print("global:     " + re.global);
   print("ignoreCase: " + re.ignoreCase);
   print("multiline:  " + re.multiline);
   print();
};

// Some regular expression to test the function.
var re1 : RegExp = new RegExp("the","i");  // Use the constructor.
var re2 = /\w+/gm;                         // Use a literal.
RegExpPropDemo(re1);
RegExpPropDemo(re2);
RegExpPropDemo(/^\s*$/im);
```

该程序的输出为：

```
Regular expression: /the/i
global:     false
ignoreCase: true
multiline:  false

Regular expression: /\w+/gm
global:     true
ignoreCase: false
multiline:  true

Regular expression: /^\s*$/im
global:     false
ignoreCase: true
multiline:  true
```

要求

版本 5.5

应用于：

正则表达式对象

请参见

参考

global 属性

ignoreCase 属性

概念

正则表达式语法

name 属性

返回一个名称，该名称属于一个错误。

```
errorObj.name
```

参数
errorObj

必选。**Error** 对象的实例。

备注

name 属性返回错误名或异常类型。当发生运行时错误时，名称属性将被设置为下列本机异常类型之一：

异常类型	含义
Error	此错误是一个用户定义的错误，是使用 **Error** 对象构造函数创建的。
ConversionError	每当试图对一个对象进行它所无法完成的转换时，将发生此错误。
RangeError	当给一个函数提供一个超过其允许范围的参数时，将发生此错误。例如，当试图构造的 **Array** 对象的长度不是有效的正整数时，发生此错误。
ReferenceError	当检测到无效引用时，将发生此错误。例如，如果所需的引用为 **null**，将发生此错误。
RegExpError	当正则表达式产生编译错误时，将发生此错误。然而，一旦该正则表达式经过了编译，就不会发生此错误。例如，如果声明正则表达式的模式使用了无效的语法或 **i**、**g** 或 **m** 以外的标志，或者多次包含同一个标志，将发生此错误。
SyntaxError	当分析源文本后发现它不遵循正确的语法时，将发生此错误。例如，当调用 **eval** 函数的参数不是有效的程序文本时，将发生此错误。
TypeError	当操作数的实际类型与所期望的类型不符时，将发生此错误。例如，如果在不是对象的内容上进行函数调用或者该内容不支持该调用时，发生此错误。
URIError	当检测到非法的统一资源标识符 (URI) 时，将发生此错误。例如，在正编码或解码的字符串中发现非法字符时，发生此错误。

示例

下面的示例导致引发了异常，并显示了错误和错误说明。

```
function getAge(age) {
   if(age < 0)
      throw new Error("An age cannot be negative.")
   print("Age is "+age+".");
}

// Pass the getAge an invalid argument.
try {
   getAge(-5);
} catch(e) {
   print(e.name);
   print(e.description);
}
```

该代码的输出为:

```
Error
An age cannot be negative.
```

要求

版本 5.5

应用于:

错误对象

请参见

参考

description 属性
message 属性 (JScript)
number 属性

JScript 8.0

NaN 属性

一个特殊值，此值指示算术表达式返回了非数字值。

```
Number.NaN
```

参数

Number

必选。全局 **Number** 对象。

备注

Number 对象并不是必须在可访问 **NaN** 属性之前创建。

NaN 不与任何值相等，包括其本身。若要测试某值是否等效于 **NaN**，请使用 **Global** 对象的 **isNaN** 方法。

要求

版本 2

应用于：

Number 对象

请参见

参考

isNaN 方法
MAX_VALUE 属性
MIN_VALUE 属性
NEGATIVE_INFINITY 属性
POSITIVE_INFINITY 属性
toString 方法

JScript 8.0

NaN 属性（全局）

返回特殊值 **NaN**，指示一个表达式不是数字。

```
NaN
```

备注

NaN 属性（非数字）是 **Global** 对象的成员，当脚本引擎初始化时变为可用。

NaN 不与任何值相等，包括其本身。若要测试某值是否等效于 **NaN**，请使用 **Global** 对象的 **isNaN** 方法。

要求

版本 3

应用于：

Global 对象

请参见

参考

isNaN 方法

NEGATIVE_INFINITY 属性

返回比 JScript 可表示的最小负数 (-Number.MAX_VALUE) 更小的值。

```
Number.NEGATIVE_INFINITY
```

参数

Number

必选。全局 **Number** 对象。

备注

Number 对象并不是必须在可访问 **NEGATIVE_INFINITY** 属性之前创建。

JScript 将 **NEGATIVE_INFINITY** 值显示为负无穷大 (-Infinity)。此值在数学上与无穷大相同。

要求

版本 2

应用于:

Number 对象

请参见

参考

MAX_VALUE 属性
MIN_VALUE 属性
NaN 属性
POSITIVE_INFINITY 属性
toString 方法

number 属性

返回或设置与特定错误相关联的数字值。

```
object.number
```

参数
object

任意 **Error** 对象的实例。

备注

错误号是一个 32 位的值。较高的 16 位字是设施代码，而较低的字才是真正的错误代码。若要读完实际的错误代码，请使用 **&**（按位与）运算符来将 number 属性与十六进制数字 `0xFFFF` 组合。

示例

下面的示例导致引发了异常，并显示了错误号。

```
function getAge(age) {
   if(age < 0)
      throw new Error(100)
   print("Age is "+age+".");
}

// Pass the getAge an invalid argument.
try {
   getAge(-5);
} catch(e) {
// Extract the error code from the error number.
   print(e.number & 0xFFFF)
}
```

该代码的输出为：

```
100
```

要求

版本 5

应用于：

错误对象

请参见
参考
description 属性
message 属性 (JScript)
name 属性

JScript 8.0

PI 属性

返回数学常数 pi 的值。

```
Math.PI
```

参数

Math

必选。全局 **Math** 对象。

备注

PI 属性是一个常数，约等于 3.14159。它表示圆的周长与直径的比率。

要求

版本 1

应用于：

Math 对象

请参见

其他资源

属性 (JScript)

POSITIVE_INFINITY 属性

返回比在 JScript 中能够表示的最大的数 (Number.MAX_VALUE) 更大的值。

```
Number.POSITIVE_INFINITY
```

参数

Number

必选。全局 **Number** 对象。

备注

Number 对象并不是必须在可访问 **POSITIVE_INFINITY** 属性之前创建。

JScript 将 **POSITIVE_INFINITY** 值显示为无穷大 (infinity)。此值在数学上与无穷大相同。

要求

版本 2

应用于:

Number 对象

请参见

参考

MAX_VALUE 属性
MIN_VALUE 属性
NaN 属性
NEGATIVE_INFINITY 属性
toString 方法

propertyIsEnumerable 属性

返回布尔值，该值指示指定属性是否为对象的一部分以及该属性是否是可枚举的。

```
object.propertyIsEnumerable(propName)
```

参数

object

必选。对象的实例。

propName

必选。一个属性名称的字符串值。

备注

如果 *propName* 存在于 *object* 中且可以使用一个 **For...In** 循环枚举出来，则 **propertyIsEnumerable** 属性将返回 **true**。如果 *object* 不具有所指定名称的属性或者所指定的属性不是可枚举的，则 **propertyIsEnumerable** 属性将返回 **false**。通常，预定义的属性不是可枚举的，而用户定义的属性总是可枚举的。

propertyIsEnumerable 属性不考虑原型链中的对象。

示例

下面的示例阐释了 **propertyIsEnumerable** 属性的用法。

```
var a : Array = new Array("apple", "banana", "cactus");
print(a.propertyIsEnumerable(1));
```

该程序的输出为：

```
true
```

要求

版本 5.5

应用于：

Object 对象

请参见

其他资源

属性 (JScript)

prototype 属性

为对象的类返回原型的引用。

```
object.prototype
```

参数
object

必选。对象的名称。

备注

用 **prototype** 属性为对象的类提供一组基本功能。对象的新的实例"继承"了赋予该对象的原型的行为。

所有内部 JScript 对象都有一个只读的 **prototype** 属性。可以像该例中那样，为原型添加功能，但不可以向对象赋予另外一个原型。但是，可以向用户定义的对象赋予新的原型。

本语言参考中，每个内部对象的方法和属性列表都指示了哪些是对象原型的一部分，哪些不是。

注意

当以快速模式(JScript 的默认模式)运行时，不能修改内置对象的 **prototype** 属性。若要从命令行编译使用 **prototype** 属性的程序，必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

示例

例如，若希望为 **Array** 对象添加一种方法，使其可返回数组中最大元素的值。要完成这一点，声明该函数，将其添加到 **Array.prototype**，然后使用它。

```
function array_max() {
   var i, max = this[0];
   for (i = 1; i < this.length; i++) {
      if (max < this[i])
         max = this[i];
   }
   return max;
}
Array.prototype.max = array_max;
var x = new Array(1, 2, 3, 4, 5, 6);
print(x.max());
```

当使用 /fast- 选项编译后，该程序的输出为：

```
6
```

要求

版本 2

应用于：

Array 对象 | Boolean 对象 | Date 对象 | Function 对象 | Number 对象 | Object 对象 | String 对象

请参见
参考
constructor 属性

rightContext 属性 ($')

返回一些字符，即从最后一个匹配之后的位置直到被搜索字符串的结尾处的字符。只读。

```
RegExp.rightContext
```

参数

RegExp

必选。全局 **RegExp** 对象。

备注

rightContext 属性的初始值为一个空字符串。每当产生一个成功匹配时，**rightContext** 属性的值将更改。

注意

以快速模式（JScript 的默认模式）运行时，**RegExp** 对象的属性不可用。若要从命令行编译使用这些属性的程序，必须使用 **/fast-** 关闭快速选项。由于线程处理问题，在 ASP.NET 中关闭快速选项是不安全的。

rightContext 属性的缩写是 **$'**。表达式 RegExp**["$'"]** 和 RegExp**.rightContext** 可交换使用。

示例

下面的示例阐述了 **rightContext** 属性的用法：

```
var s;                                //Declare variable.
var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
var str = "cdbBdbsbdbdz";             //String to be searched.
var arr = re.exec(str);               //Perform the search.
s = "$1 returns: " + RegExp.$1 + "\n";
s += "$2 returns: " + RegExp.$2 + "\n";
s += "$3 returns: " + RegExp.$3 + "\n";
s += "input returns : " + RegExp.input + "\n";
s += "lastMatch returns: " + RegExp.lastMatch + "\n";
s += "leftContext returns: " + RegExp.leftContext + "\n";
s += "rightContext returns: " + RegExp.rightContext + "\n";
s += "lastParen returns: " + RegExp.lastParen + "\n";
print(s);                            //Return results.
```

当使用 **/fast-** 选项编译该程序后，该程序的输出为：

```
$1 returns: bB
$2 returns: d
$3 returns:
input returns : cdbBdbsbdbdz
lastMatch returns: dbBd
leftContext returns: c
rightContext returns: bsbdbdz
lastParen returns: d
```

要求

版本 5.5

应用于：

RegExp 对象

请参见

参考

$1...$9 属性
index 属性

source 属性

返回正则表达式模式的文本的一个副本。只读。

```
rgExp.source
```

参数

rgExp

必选。一个正则表达式对象。

备注

rgExp 可以是一个存储 **Regular Expression** 对象的变量，或者可以是一个正则表达式。

示例

下面的示例阐释了 **source** 属性的用法：

```
var src : String = "Spain";
var re : RegExp = /in/g;
var s1;
// Test string for existence of regular expression.
if (re.test(src))
   s1 = " contains ";
else
   s1 = " does not contain ";
// Get the text of the regular expression itself.
print("The string " + src + s1 + re.source + ".");
```

该程序的输出为：

```
The string Spain contains in.
```

要求

版本 3

应用于：

正则表达式对象

请参见

参考

正则表达式对象

概念

正则表达式语法

SQRT1_2 属性

返回 0.5 的平方根，即 2 的平方根分之一。

```
Math.SQRT1_2
```

参数
Math

必选。全局 **Math** 对象。

备注

SQRT1_2 属性为一个常数，约等于 0.707。

要求

版本 1

应用于：

Math 对象

请参见
参考
sqrt 方法
SQRT2 属性

JScript 8.0

SQRT2 属性

返回 2 的平方根。

```
Math.SQRT2
```

参数
Math

必选。全局 **Math** 对象。

备注

SQRT2 属性为一个常数，约等于 1.414。

要求

版本 1

应用于：

Math 对象

请参见
参考
sqrt 方法
SQRT1_2 属性

undefined 属性

返回 **undefined** 的值。

```
undefined
```

备注

undefined 属性是 **Global** 对象的一个成员，该属性在脚本引擎初始化后变得可用。如果已声明了一个变量但还未进行初始化，则其值为 **undefined**。

如果还没有声明变量，则不能将其与 **undefined** 进行比较，但是可将变量的类型与字符串"undefined"进行比较。未声明的变量不能用于快速模式。

当显式测试变量或将变量设置为未定义时，**undefined** 属性是很有用的。

示例

下面的示例阐释了 undefined 属性的用法：

```
var declared;                          //Declare variable.
if (declared == undefined)             //Test variable.
   print("The variable declared has not been given a value.");
```

该代码的输出为：

```
The variable declared has not been given a value.
```

要求

版本 5.5

应用于：

Global 对象

请参见

概念

未定义的值

语句

语句是一条执行操作的 JScript 代码。有些语句声明用户定义的元素，如变量、函数、类和枚举，而另一些语句则控制程序流程。以下各节链接到有关解释如何使用 JScript 中语句的信息。

本节内容

interface 语句

声明一个接口和该接口的成员。

Labeled 语句

为一条语句提供一个标识符。

package 语句

提供一种方法，用以将类和接口一起打包在命名组件中。

print 语句

提供一种方法，用以显示从命令行运行程序所得的信息。

return 语句

从当前函数退出，并从该函数返回一个值。

@set 语句

创建使用条件编译语句的变量。

static 语句

声明将类初始化的代码块。

super 语句

引用当前对象的基类。

switch 语句

当指定的表达式的值与一个标签匹配时，使一条或多条语句得以执行。

this 语句

指当前的对象。

throw 语句

生成一个可由 **try...catch** 语句处理的错误条件。

try...catch...finally 语句

为 JScript 实现错误处理。

var 语句

声明一个变量。

while 语句

执行一个语句，直到指定的条件为 **false**。

with 语句

为语句建立默认对象。

相关章节

JScript 参考

列出"JScript 语言参考"所包含的元素，并提供指向特定主题的链接，这些主题解释正确使用语言元素的详细背景信息。

.NET Framework 参考

列出指向特定主题的链接，这些主题解释 .NET Framework 类库和其他基本元素的语法和结构。

JScript 8.0

break 语句

终止当前循环，或者，当与标签一起使用时，终止相关联的语句。

```
break [label];
```

参数
label

可选项。指定正要从中中断的语句的标签。

备注

通常在 **switch** 语句和 **while**、**for**、**for...in** 或 **do...while** 循环中使用 **break** 语句。*label* 参数最常使用在 **switch** 语句中，但它可以用在任何简单或复合的语句中。

执行 **break** 语句将导致程序流退出当前的循环或语句。程序流将继续执行紧接着当前循环或语句的下一条语句。

示例 1

下面的示例阐释了 **break** 语句的用法。

```
function breakTest(breakpoint){
   var i = 0;
   while (i < 100) {
      if (i == breakpoint)
         break;
      i++;
   }
   return(i);
}
```

示例 2

下面的示例阐释了标记 **break** 语句的用法。

```
function nameInDoubleArray(name, doubleArray) {
   var i, j, inArray;
   inArray = false;
   mainloop:
   for(i=0; i<doubleArray.length; i++)
      for(j=0; j<doubleArray[i].length; j++)
         if(doubleArray[i][j] == name) {
            inArray = true;
            break mainloop;
         }
   return inArray;
}
```

要求

版本 1

请参见
参考
continue 语句
do...while 语句
for 语句
for...in 语句
Labeled 语句
while 语句

class 语句

声明一个类的名称以及组成该类的变量、属性和方法的定义。

```
[modifiers] class classname [extends baseclass] [implements interfaces]{
   [classmembers]
}
```

参数

modifiers

可选项。控制类的可见性和行为的修饰符。

classname

必选。class 的名称；遵循标准的变量命名规则。

extends

可选项。指示 *classname* 类扩展 *baseclass* 的关键字。如果未使用此关键字，则创建扩展 **System.Object** 的标准 JScript 基类。

baseclass

可选项。所扩展的类的名称。

implements

可选项。指示 *classname* 类实现一个或多个接口的关键字。

interfaces

可选项。接口名称的逗号分隔列表。

classmembers

可选。*classmembers* 可以是以下对象：用 **function** 语句定义的方法或构造函数声明，用 **function get** 和 **function set** 语句定义的属性声明，用 **var** 或 **const** 语句定义的字段声明，用 **static** 语句定义的初始值设定项声明，用 **enum** 语句定义的枚举声明，或者嵌套类声明。

备注

根据类的修饰符，类可用于创建实例或用作其他类的基类。如果一个类使用 **abstract** 修饰符来标记，该类则可用作其他要扩展的类的基类，但不能创建 **abstract** 类的实例。如果一个类使用 **final** 修饰符来标记，则可用 **new** 运算符来创建该类的实例，但该类不能用作基类。

方法和构造函数可能会在类中重载。因此，多个方法（或构造函数）可以具有相同的名称。重载的类成员通过它们的唯一签名来进行区分，签名包含成员的名称及其每个形参的数据类型。重载允许一个类将具有相似功能的方法归为一组。

类可以使用 **extends** 关键字来继承现有基类的功能。基类的方法可以通过声明与新的类方法具有相同签名的新方法来进行重写。新类中的方法可以使用 **super** 语句来访问基类的重写成员。

类可以使用 **implements** 关键字在一个或多个接口上进行模式化。由于接口不提供任何成员的实现，类不能从接口继承任何行为。接口会为类提供"签名"，签名可以在与其他类进行交互时使用。除非实现接口的类为 **abstract**，否则必须为接口中定义的每一个方法提供实现。

修饰符可用于使类实例的行为更像 JScript 对象的行为。若要允许类实例动态地处理所添加的属性，请使用 **expando** 修饰符，该修饰符会自动为类创建默认的索引属性。只有 expando 属性才能使用 JScript **Object** 对象的方括号来进行访问。

示例 1

下面的示例创建一个具有多个字段和方法的 `CPerson` 类，其细节已省略。`CPerson` 类用作下一个示例中 `CCustomer` 类的基类。

```
// All members of CPerson are public by default.
class CPerson{
   var name : String;
   var address : String;
```

```
   // CPerson constuctor
   function CPerson(name : String){
      this.name = name;
   };

   // printMailingLabel is an instance method, as it uses the
   // name and address information of the instance.
   function printMailingLabel(){
      print(name);
      print(address);
   };

   // printBlankLabel is static as it does not require
   // any person-specific information.
   static function printBlankLabel(){
      print("-blank-");
   };
}

// Print a blank mailing label.
// Note that no CPerson object exists at this time.
CPerson.printBlankLabel();

// Create a CPerson object and add some data.
var John : CPerson = new CPerson("John Doe");
John.address = "15 Broad Street, Atlanta, GA 30315";
// Print a mailing label with John's name and address.
John.printMailingLabel();
```

该代码的输出为:

```
-blank-
John Doe
15 Broad Street, Atlanta, GA 30315
```

示例 2

`CCustomer` 类从 `CPerson` 派生，它具有其他不适用于 `CPerson` 类一般成员的字段和方法。

```
// Create an extension to CPerson.
class CCustomer extends CPerson{
   var billingAddress : String;
   var lastOrder : String;

   // Constructor for this class.
   function CCustomer(name : String, creditLimit : double){
      super(name); // Call superclass constructor.
      this.creditLimit = creditLimit;
   };

   // Customer's credit limit. This is a private field
   // so that only member functions can change it.
   private var creditLimit : double;
   // A public property is needed to read the credit limit.
   function get CreditLimit() : double{
      return creditLimit;
   }
}

// Create a new CCustomer.
var Jane : CCustomer = new CCustomer("Jane Doe",500.);
// Do something with it.
Jane.billingAddress = Jane.address = "12 Oak Street, Buffalo, NY 14201";
Jane.lastOrder = "Windows 2000 Server";
// Print the credit limit.
print(Jane.name + "'s credit limit is " + Jane.CreditLimit);
// Call a method defined in the base class.
```

```
Jane.printMailingLabel();
```

该部分代码的输出为:

```
Jane Doe's credit limit is 500
Jane Doe
12 Oak Street, Buffalo, NY 14201
```

要求

.NET 版本

请参见

参考

interface 语句
function 语句
function get 语句
function set 语句
var 语句
const 语句
static 语句
new 运算符
this 语句
super 语句

其他资源

修饰符

@cc_on 语句

激活条件编译支持。

```
@cc_on
```

备注

@cc_on 语句激活脚本引擎中的条件编译。

强烈推荐在注释中使用 **@cc_on** 语句，以使不支持条件编译的浏览器将您的脚本视为有效语法而接受它：

```
/*@cc_on*/
// The remainder of the script.
```

或者，在注释外的一个 **@if** 或 **@set** 语句也将激活条件编译。

要求

版本 3

请参见

参考

@if...@elif...@else...@end 语句

@set 语句

概念

条件编译变量

其他资源

条件编译

Comment 语句

使 JScript 分析器忽略注释。

```
//single-line comment
// comment

//multiline comment
/*
comment
*/

//single-line conditional comment
//@CondStatement

//multiline conditional comment
/*@
condStatement
@*/
```

备注

condStatement 参数是条件编译被激活后使用的条件编译代码。如果使用语法 3，则在"//"和"@"字符间可以没有空格。

用注释来阻止 JScript 分析器读取脚本的某些部分。可使用注释在程序中加入解释性说明。

如果使用的是语法 1，则分析器将忽略在注释标记和行尾之间的任何文本。如果使用语法 2，则将忽略开始标记和结束标记之间的所有文本。

当与不支持该功能的浏览器之间保留兼容性时，语法 3 和 4 用来支持条件编译。这些浏览器把这些形式的注释分别视为语法 1 和语法 2。

示例

下面的示例阐释了 **comment** 语句的最常见用法。

```
function myfunction(arg1, arg2){
   /* This is a multiline comment that
      can span as many lines as necessary. */
   var r = 0;
   // This is a single line comment.
   r = arg1 + arg2; // Sum the two arguments.
   return(r);
}
```

要求

版本 1

请参见

参考

@cc_on 语句

@set 语句

概念

条件编译变量

其他资源

条件编译

const 语句

声明一个常数。

```
//Syntax for declaring a constant of global scope or function scope.
const name1 [: type1] = value1 [, ... [, nameN [: typeN] = valueN]]
//Syntax for declaring a constant field in a class.
[modifiers] const name1 [: type1] = value1 [, ... [, nameN [: typeN] = valueN]]
```

参数

modifiers

可选项。控制字段的可见性和行为的修饰符。

name1, ..., nameN

必选。所声明的常数的名称。

type1, ..., typeN

可选项。所声明的常数的类型。

value1, ..., valueN

赋给常数的值。

备注

使用 **const** 语句可声明常数。常数可以绑定到特定的数据类型来确保类型安全。当声明常数时，必须给这些常数赋值，这些值不能在随后从脚本中更改。

类中的常数字段类似于全局或函数常数，不同的只是它的作用范围是该类，而且它可以用多个修饰符来指定其可见性和用法。

注意

当常数绑定到引用数据类型（如 **Object**、**Array**、类实例或类型化数组）时，可以更改常数所引用的数据。之所以可以这样，是因为 **const** 语句只会使引用类型成为常数；它所引用的数据不是常数。

示例

以下示例阐释 **const** 语句的用法。

```
class CSimple {
   // A static public constant field. It will always be 42.
   static public const constantValue : int = 42;
}
const index = 5;
const name : String = "Thomas Jefferson";
const answer : int = 42, oneThird : float = 1./3.;
const things : Object[] = new Object[50];
things[1] = "thing1";
// Changing data referenced by the constant is allowed.
```

要求

.NET 版本

请参见

参考

var 语句

function 语句

class 语句

概念

变量和常数的范围

类型批注

其他资源

修饰符

continue 语句

停止循环的当前迭代，并开始新的迭代。

```
continue [label];
```

参数
label

可选项。指定应用 **continue** 的语句。

备注

只能在 **while**、**do...while**、**for** 或 **for...in** 循环内使用 **continue** 语句。执行 continue 语句会停止当前循环的迭代，并从循环的开始处继续程序流。这将对不同类型的循环有如下影响：

- **while** 和 **do...while** 循环将测试其条件，如果条件为真，则将再次执行循环。
- **for** 循环执行其增量表达式，如果测试表达式为真，则将再次执行循环。
- **for...in** 循环继续进行到指定变量的下一个字段，并将再次执行循环。

示例

下面的示例说明了 **continue** 语句的用法。

```
function skip5(){
   var s = "", i=0;
   while (i < 10) {
      i++;
      // Skip 5
      if (i==5) {
         continue;
      }
      s += i;
   }
   return(s);
}
```

要求

版本 1

请参见
参考
break 语句
do...while 语句
for 语句
for...in 语句
Labeled 语句
while 语句

debugger 语句

启动调试器。

```
debugger
```

备注

debugger 语句启动已安装的调试器。其效果类似于在使用调试器语句的程序中设置断点。

如果未安装调试器，则 **debugger** 语句无效。

要求

版本 3

请参见

其他资源

语句

编写、编译、调试 JScript 代码

do...while 语句

将一个语句块执行一次，然后重复该循环的执行，直到条件表达式为 **false**。

```
do
      statement
while (expression)
```

参数

statement

必选。*expression* 为 **true** 时要执行的语句。可以是复合语句。

表达式

必选。一个可以被强迫转换为布尔值 **true** 或 **false** 的表达式。如果 *expression* 为 **true**，则再执行一次循环。如果 expression 是 **false**，则循环终止。

备注

在循环的第一次迭代完成前，不检查 *expression* 的值，保证循环至少执行一次。此后，每次后继循环迭代之后都要检查 expression。

示例

下面的示例阐释了如何使用 **do...while** 语句来迭代 **Drives** 集合。

```
function GetDriveList(){
   var fso, s, n, e, x;
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);
   s = "";
   do {
      x = e.item();
      s = s + x.DriveLetter;
      s += " - ";
      if (x.DriveType == 3)
         n = x.ShareName;
      else if (x.IsReady)
         n = x.VolumeName;
      else
         n = "[Drive not ready]";
         s +=  n + "\n";
      e.moveNext();
   }
   while (!e.atEnd());
   return(s);
}
```

要求

版本 3

请参见

参考

break 语句
continue 语句
for 语句
for...in 语句
while 语句
Labeled 语句

enum 语句

声明枚举数据类型的名称和枚举成员的名称。

```
[modifiers] enum enumName [ : typeAnnotation]{
   enumValue1 [ = initializer1]
   [,enumValue2 [ = initializer2]
   [, ... [,enumValueN [ = initializerN ] ]]]
}
```

参数

modifiers

可选项。控制枚举的可见性和行为的修饰符。

enumName

必选。枚举类型的名称。

typeAnnotation

可选项。枚举的基础数据类型。必须为整型数据类型。默认值为 **int**。

enumValue1, enumValue2, ..., enumValueN

可选项。枚举类型成员。

initializer1, initializer2, ..., initializerN

可选项。重写枚举成员的默认数值的常数表达式。

备注

enum 声明在程序中引入新的枚举数据类型。**enum** 声明只能出现在类声明可以出现的上下文中(即全局范围)、包范围内或类范围内，而不能出现在函数或方法中。

可以将枚举的基础类型声明为任何整型数据类型(**int**、**short**、**long**、**byte**、**uint**、**ushort**、**ulong** 或 **sbyte**)。枚举成员隐式地强迫到/自基础数据类型，从而允许将数值数据直接赋给类型为 **enum** 的变量。默认情况下，枚举的基础数据类型为 **int**。

每个枚举类型成员都有一个名称和一个可选的初始值设定项。初始值设定项必须是与指定枚举具有相同类型(或可转换为该类型)的编译时常数表达式。第一个枚举类型成员的值为零或初始值设定项的值(如果已提供)。后继的每个枚举类型成员的值比前一个成员大一或者是初始值设定项的值(如果已提供)。

enum 值以类似于访问静态类成员的方法来进行访问。成员的名称必须用枚举的名称(例如 `Color.Red`)来进行限定。当给 **enum** 类型的变量赋值时，可以使用以下三项中的一项:完全限定名(例如 `Color.Red`)、名称的字符串表示形式(例如 `"Red"`)或数值。

如果给 **enum** 赋以在编译时已知的字符串，编译器将执行必要的转换。例如，`"Red"` 会替换为 `Color.Red`。如果字符串在编译时未知，则将在运行时执行转换。如果字符串不是枚举类型的有效成员，该转换则可能失败。由于转换需要时间并且可能生成运行时错误，所以应避免将 **enum** 赋给变量字符串。

枚举类型的变量可以接受声明值范围之外的值。此功能的用途之一是允许将成员组合用作位标志，如以下示例所示。将 **enum** 变量转换为字符串会产生成员名称的字符串表示形式。

示例 1

下面的示例显示枚举的行为。它声明一个名为 `CarType` 的简单枚举，该枚举的成员包括 `Honda`、`Toyota` 和 `Nissan`。

```
        enum CarType {
   Honda,    // Value of zero, since it is first.
   Toyota,   // Value of 1, the successor of zero.
   Nissan    // Value of 2.
}

// Declare a variable of type CarType, and give it the value Honda.
var myCar : CarType = CarType.Honda;
```

```
print(int(myCar) + ": " + myCar);

myCar = "Nissan"; // Change the value to "Nissan".
print(int(myCar) + ": " + myCar);

myCar = 1; // 1 is the value of the Toyota member.
print(int(myCar) + ": " + myCar);
```

该代码的输出为:

```
0: Honda
2: Nissan
1: Toyota
```

示例 2

下面的示例显示如何使用枚举来接受位标志，并显示 **enum** 变量必须能够接受不显式存在于成员列表中的值。它定义一个枚举 `FormatFlags`，该 `FormatFlags` 用于修改 `Format` 函数的行为。

```
// Explicitly set the type to byte, as there are only a few flags.
enum FormatFlags : byte {
   // Can't use the default values, since we need explicit bits
   ToUpperCase = 1,   // Should not combine ToUpper and ToLower.
   ToLowerCase = 2,
   TrimLeft    = 4,   // Trim leading spaces.
   TrimRight   = 8,   // Trim trailing spaces.
   UriEncode   = 16   // Encode string as a URI.
}

function Format(s : String, flags : FormatFlags) : String {
   var ret : String = s;
   if(flags & FormatFlags.ToUpperCase) ret = ret.toUpperCase();
   if(flags & FormatFlags.ToLowerCase) ret = ret.toLowerCase();
   if(flags & FormatFlags.TrimLeft)    ret = ret.replace(/^\s+/g, "");
   if(flags & FormatFlags.TrimRight)   ret = ret.replace(/\s+$/g, "");
   if(flags & FormatFlags.UriEncode)   ret = encodeURI(ret);
   return ret;
}

// Combine two enumeration values and store in a FormatFlags variable.
var trim : FormatFlags = FormatFlags.TrimLeft | FormatFlags.TrimRight;
// Combine two enumeration values and store in a byte variable.
var lowerURI : byte = FormatFlags.UriEncode | FormatFlags.ToLowerCase;

var str : String = "  hello, WORLD  ";

print(trim + ": " + Format(str, trim));
print(FormatFlags.ToUpperCase + ": " + Format(str, FormatFlags.ToUpperCase));
print(lowerURI + ": " + Format(str, lowerURI));
```

该代码的输出为:

```
12: hello, WORLD
ToUpperCase:   HELLO, WORLD
18: %20%20hello,%20world%20%20
```

要求

.NET 版本

请参见

概念

类型转换

类型批注

其他资源

修饰符

for 语句

只要指定条件为真，就执行一个语句块。

```
for (initialization; test; increment)
...statement
```

参数

initialization

必选。一个表达式。该表达式在执行循环前仅被执行一次。

test

必选。布尔表达式。如果 test 为 **true**，则执行 statement。如果 test 为 **false**，则结束循环。

increment

必选。一个表达式。在每次通过循环的结尾执行该增量表达式。

statement

可选项。test 为 **true** 时要执行的语句。可以是复合语句。

备注

当循环将执行已知的次数时，通常使用 **for** 循环。

示例

下面的示例说明了一个 for 循环。

```
/* i is set to 0 at start, and is incremented by 1 at the end
   of each iteration. Loop terminates when i is not less
   than 10 before a loop iteration. */
var myarray = new Array();
for (var i = 0; i < 10; i++) {
   myarray[i] = i;
}
```

要求

版本 1

请参见

参考

for...in 语句

while 语句

for...in 语句

为一个对象的每个属性，或一个数组或集合中的每个元素，执行一个或多个语句。

```
for ( [var] variable in {object | array | collection})
      statement
```

参数

variable

必选。一个变量，可以是 *object* 的任何属性名、*array* 的任何索引或 *collection* 的任何元素。

object

要对其进行循环的 JScript 对象。

array

要对其进行循环的数组。它可以是 JScript **Array** 对象或 .NET Framework 数组。

collection

要对其进行循环的集合。它可以是从 .NET Framework 实现 **IEnumerable** 或 **IEnumerator** 接口的任何类。

statement

可选项。要为 *object* 的每个属性或者 *array* 或 *collection* 的每个元素执行的语句。可以是复合语句。

备注

在循环的每一次循环之前，会为 *variable* 分配 *object* 的下一个属性名、*array* 的下一个索引或 *collection* 的下一个元素。您可以将 *variable* 用于循环中的任何语句，以便引用 *object* 的属性或 *array* 的元素。

当在一个对象上循环时，没有办法确定或控制将对象的成员名赋给 variable 的顺序。**for...in** 语句无法在非 JScript 对象的成员上循环，例如，.NET Framework 对象。

数组按元素顺序循环，从最小的索引开始，到最大的索引结束。因为 JScript **Array** 对象可能是稀疏的，所以 **for...in** 语句只访问数组中已定义的元素。JScript **Array** 对象还可以具有 expando 属性，在这种情况下，会为 *variable* 分配数组索引作为属性名。如果数组是多维 .NET Framework 数组，则只枚举第一维。

为了在集合上循环，会按元素在集合中的显示顺序将元素分配给 *variable*。

示例 1

下面的示例阐释了 for ... in 语句的用法，该语句将一个对象用作一个相关数组。

```
function ForInDemo1() {
   var ret = "";

   // Initialize the object with properties and values.
   var obj : Object = {"a" : "Athens" ,
                       "b" : "Belgrade",
                       "c" : "Cairo"};

   // Iterate over the properties.
   for (var key in obj)
      // Loop and assign 'a', 'b', and 'c' to key.
      ret += key + ":\t" + obj[key] + "\n";

   return(ret);
} // ForInDemo1
```

此函数返回包含下列内容的字符串：

```
a:      Athens
b:      Belgrade
```

```
c:      Cairo
```

示例 2

此示例阐释了配合使用 **for ... in** 语句和 JScript **Array** 对象（具有 expando 属性）的方法。

```
function ForInDemo2() {
   var ret = "";

   // Initialize the array.
   var arr : Array = new Array("zero","one","two");
   // Add a few expando properties to the array.
   arr["orange"] = "fruit";
   arr["carrot"] = "vegetable";

   // Iterate over the properties and elements.
   for (var key in arr)
      // Loop and assign 0, 1, 2, 'orange', and 'carrot' to key.
      ret += key + ":\t" + arr[key] + "\n";

   return(ret);
} // ForInDemo2
```

此函数返回包含下列内容的字符串：

```
0:      zero
1:      one
2:      two
orange: fruit
carrot: vegetable
```

示例 3

下面的示例阐释了配合使用 **for ... in** 语句和集合的方法。其中，**System.String** 对象的 **GetEnumerator** 方法提供了字符串中字符的集合。

```
function ForInDemo3() {
   var ret = "";

   // Initialize collection.
   var str : System.String = "Test.";
   var chars : System.CharEnumerator = str.GetEnumerator();

   // Iterate over the collection elements.
   var i : int = 0;
   for (var elem in chars) {
      // Loop and assign 'T', 'e', 's', 't', and '.' to elem.
      ret += i + ":\t" + elem + "\n";
      i++;
   }

   return(ret);
} // ForInDemo3
```

此函数返回包含下列内容的字符串：

```
0:      T
1:      e
2:      s
3:      t
4:      .
```

要求

版本 5

注意
在集合上循环需要 .NET 版本。

请参见

参考

for 语句

while 语句

String.GetEnumerator Method

其他资源

JScript 数组

function 语句

声明一个新函数。这可以用在几种上下文中：

```
// in the global scope
function functionname([parmlist]) [: type] {
   [body]
}

// declares a method in a class
[attributes] [modifiers] function functionname([parmlist]) [: type] {
   [body]
}

// declares a method in an interface
[attributes] [modifiers] function functionname([parmlist]) [: type]
```

参数

attributes

可选项。控制方法的可见性和行为的属性。

modifiers

可选项。控制方法的可见性和行为的修饰符。

functionname

必选。函数或方法的名称。

paramlist

可选项。一个用于函数或方法的以逗号分隔的参数列表。每个参数都可以包含一个类型规范。最后一个参数可以是 *parameterarray*，它由三个句号 (**...**) 后接一个参数数组名称和一个类型化数组的类型批注来表示。

type

可选项。方法的返回类型。

body

可选项。定义函数或方法如何操作的一个或多个语句。

备注

使用 function 语句来声明一个以后要使用的函数。在脚本的其他地方调用该函数前，*body* 中包含的代码不被执行。**return** 语句用于从函数返回值。您不必使用 **return** 语句，因为程序将在到达函数结尾时返回。

方法类似于全局函数，不同之处在于它们的范围限于定义它们的 **class** 或 **interface**，并且它们可以有各种修饰符来控制它们的可见性和行为。**interface** 中的方法不能有主体，而 **class** 中的方法必须有主体。此规则有一个例外情况；如果 **class** 中的某个方法为 **abstract**，或 **class** 为 **abstract**，则该方法就不能有主体。

可以用类型批注声明函数或方法返回何种数据类型。如何将返回类型指定为 **void**，则函数内的任何 **return** 语句都不返回值。如果指定了 **void** 以外的任何返回类型，则函数内的所有 **return** 语句都必须返回一个强迫为指定返回类型的值。如果指定了返回类型，但 **return** 语句没有带任何值或者当到达函数结尾时没有出现 **return** 语句，则返回 **undefined** 值。构造函数不能指定返回类型，因为 **new** 运算符自动返回创建的对象。

如果没有为函数指定显式返回类型，则返回类型将设置为 **Object** 或 **void**。只有当没有 **return** 语句或 **return** 语句在函数体中不带任何值的情况下，才选择 **void** 返回类型。

参数数组可以用作函数的最后一个参数。在必选参数之后传递给函数的任何其他参数（如果有的话）都将输入到该参数数组中。参数的类型批注不是可选项；它必须是一个类型化数组。若要接受任意类型的参数，请使用 **Object[]** 作为此类型化数组。当调用的函数可接受各种参数时，应使用一个所需类型的显式数组，而不是提供一个参数列表。

在调用函数时，请确保始终包括圆括号和任何必选参数。不带圆括号就调用函数将导致返回函数的文本，而不是函数的结果。

示例 1

下面的示例阐释了第一个语法中 **function** 语句的用法：

```
interface IForm {
   // This is using function in Syntax 3.
   function blank() : String;
}

class CForm implements IForm {
   // This is using function in Syntax 2.
   function blank() : String {
      return("This is blank.");
   }
}

// This is using function in Syntax 1.
function addSquares(x : double, y : double) : double {
   return(x*x + y*y);
}

// Now call the function.
var z : double = addSquares(3.,4.);
print(z);

// Call the method.
var derivedForm : CForm = new CForm;
print(derivedForm.blank());

// Call the inherited method.
var baseForm : IForm = derivedForm;
print(baseForm.blank());
```

该程序的输出为：

```
25
This is blank.
This is blank.
```

示例 2

在本示例中，`printFacts` 函数将输入内容视为 **String**，并使用一个参数数组来接受各种 **Objects**。

```
         function printFacts(name : String, ... info : Object[]) {
   print("Name: " + name);
   print("Number of extra information: " + info.length);
   for (var factNum in info) {
      print(factNum + ": " + info[factNum]);
   }
}

// Pass several arguments to the function.
printFacts("HAL 9000", "Urbana, Illinois", new Date(1997,0,12));
// Here the array is intrepeted as containing arguments for the function.
printFacts("monolith", [1, 4, 9]);
// Here the array is just one of the arguments.
printFacts("monolith", [1, 4, 9], "dimensions");
printFacts("monolith", "dimensions are", [1, 4, 9]);
```

该程序在运行时显示下列输出：

```
Name: HAL 9000
Number of extra information: 2
0: Urbana, Illinois
1: Sun Jan 12 00:00:00 PST 1997
Name: monolith
```

```
Number of extra information: 3
0: 1
1: 4
2: 9
Name: monolith
Number of extra information: 2
0: 1,4,9
1: dimensions
Name: monolith
Number of extra information: 2
0: dimensions are
1: 1,4,9
```

要求

版本 1（对于语法 1）.NET 版本（对于语法 2 和 3）

请参见

参考

new 运算符

class 语句

interface 语句

return 语句

概念

变量和常数的范围

类型批注

类型化数组

其他资源

修饰符

function get 语句

声明类或接口中的新属性的访问器。**function get** 通常与 **function set** 一起出现，它们允许对属性进行读/写访问。

```
// Syntax for the get accessor for a property in a class.
 [modifiers] function get propertyname() [: type] {
   [body]
}

// Syntax for the get accessor for a property in an interface.
[modifiers] function get propertyname() [: type]
```

参数

modifiers

可选项。控制属性的可见性和行为的修饰符。

propertyname

必选。要创建的属性的名称。在类中必须是唯一的，不过同一个 *propertyname* 可同时与 **get** 和 **set** 访问器一起使用，以标识可对其进行读写操作的属性。

type

可选项。返回 **get** 访问器的类型。它必须匹配 **set** 访问器的参数类型（如果已定义）。

body

可选项。一个或多个定义 **get** 访问器如何操作的语句。

备注

访问对象属性的方式与访问字段的方式基本相同，只是属性允许对存储在对象中和从对象返回的值有更多的控制。属性可以是只读的、只写的或读写的，具体取决于在类中定义的 **get** 和 **set** 属性访问器的组合。属性通常用于确保 **private** 或 **protected** 字段中只存储合适的值。不能给只读属性赋值，也不能从只写属性中读取值。

必须指定返回类型的 **get** 访问器不具有任何参数。一个 **get** 访问器可以与一个 **set** 访问器形成一对，**set** 访问器具有一个参数但没有返回类型。如果同时将这两个访问器用于一个属性，则 **get** 访问器的返回类型必须与 **set** 访问器的参数类型相匹配。

一个属性可以具有 **get** 访问器或 **set** 访问器，或者同时具有这两者。只有属性的 **get** 访问器（如果没有 **get** 访问器则为 **set** 访问器）可以具有作为一个整体应用于此属性的自定义属性。**get** 和 **set** 访问器都可以具有应用于各自的访问器的修饰符和自定义属性。属性访问器不能重载，但可以隐藏或重写。

可以在 **interface** 的定义中指定属性，但在此接口中不能提供任何实现。

示例

下面的示例显示了若干属性声明。`Age` 属性定义为读取和写入的属性。还定义了只读 `FavoriteColor` 属性。

```
class CPerson {
   // These variables are not accessible from outside the class.
   private var privateAge : int;
   private var privateFavoriteColor : String;

   // Set the initial favorite color with the constructor.
   function CPerson(inputFavoriteColor : String) {
      privateAge = 0;
      privateFavoriteColor = inputFavoriteColor;
   }

   // Define an accessor to get the age.
   function get Age() : int {
      return privateAge;
   }
   // Define an accessor to set the age, since ages change.
   function set Age(inputAge : int) {
```

```
        privateAge = inputAge;
    }

    // Define an accessor to get the favorite color.
    function get FavoriteColor() : String {
        return privateFavoriteColor;
    }
    // No accessor to set the favorite color, making it read only.
    // This assumes that favorite colors never change.
}

var chris: CPerson = new CPerson("red");

// Set Chris age.
chris.Age = 27;
// Read chris age.
print("Chris is " + chris.Age + " years old.");

// FavoriteColor can be read from, but not written to.
print("Favorite color is " + chris.FavoriteColor + ".");
```

该**程序在运行**时显**示下列内容**:

```
Chrisis 27 years old.
Favorite color is red.
```

要求

.NET 版本

请参见

参考

class 语句

interface 语句

function 语句

function set 语句

概念

类型批注

其他资源

修饰符

function set 语句

声明类或接口中的新属性的访问器。**function set** 通常与 **function get** 一起出现，它们允许对属性进行读/写访问。

```
// Syntax for the set accessor of a property in a class.
 [modifiers] function set propertyname(parameter [: type]) {
   [body]
}

// Syntax for the set accessor of a property in an interface.
[modifiers] function set propertyname(parameter [: type])
```

参数

modifiers

可选项。控制属性的可见性和行为的修饰符。

propertyname

必选。要创建的属性的名称。在类中必须是唯一的，不过同一个 *propertyname* 可同时与 **get** 和 **set** 访问器一起使用，以标识可对其进行读写操作的属性。

parameter

必选。**set** 访问器所接受的形参。

type

可选项。**set** 访问器的参数类型。它必须匹配 **get** 访问器的返回类型(如果已定义)。

body

可选项。一个或多个定义 **set** 访问器如何操作的语句。

备注

访问对象属性的方式与访问字段的方式基本相同，只是属性允许对存储在对象中和从对象返回的值有更多的控制。属性可以是只读的、只写的或读写的，具体取决于在类中定义的 **get** 和 **set** 属性访问器的组合。属性通常用于确保 **private** 或 **protected** 字段中只存储合适的值。不能给只读属性赋值，也不能从只写属性中读取值。

set 访问器必须恰好具有一个参数，但它不能指定返回类型。一个 **set** 访问器可以与一个 **get** 访问器形成一对，**get** 访问器不具有任何参数且必须指定返回类型。如果同时将这两个访问器用于一个属性，则 **get** 访问器的返回类型必须与 **set** 访问器的参数类型相匹配。

一个属性可以具有 **get** 访问器或 **set** 访问器，或者同时具有这两者。只有属性的 **get** 访问器(如果没有 **get** 访问器则为 **set** 访问器)可以具有作为一个整体应用于此属性的自定义属性。**get** 和 **set** 访问器都可以具有应用于各自的访问器的修饰符和自定义属性。属性访问器不能重载，但可以隐藏或重写。

可以在 **interface** 的定义中指定属性，但在此接口中不能提供任何实现。

示例

下面的示例显示了若干属性声明。`Age` 属性定义为读取和写入的属性。还定义了只读 `FavoriteColor` 属性。

```
class CPerson {
   // These variables are not accessible from outside the class.
   private var privateAge : int;
   private var privateFavoriteColor : String;

   // Set the initial favorite color with the constructor.
   function CPerson(inputFavoriteColor : String) {
      privateAge = 0;
      privateFavoriteColor = inputFavoriteColor;
   }

   // Define an accessor to get the age.
   function get Age() : int {
```

```
        return privateAge;
    }
    // Define an accessor to set the age, since ages change.
    function set Age(inputAge : int) {
        privateAge = inputAge;
    }

    // Define an accessor to get the favorite color.
    function get FavoriteColor() : String {
        return privateFavoriteColor;
    }
    // No accessor to set the favorite color, making it read only.
    // This assumes that favorite colors never change.
}

var chris : CPerson = new CPerson("red");

// Set Chris's age.
chris.Age = 27;
// Read Chris's age.
print("Chris is " + chris.Age + " years old.");

// FavoriteColor can be read from, but not written to.
print("Favorite color is " + chris.FavoriteColor + ".");
```

该程序在运行时显示下列内容:

```
Chris is 27 years old.
Favorite color is red.
```

要求

.NET 版本

请参见

参考

class 语句

interface 语句

function 语句

function get 语句

概念

类型批注

其他资源

修饰符

JScript 8.0

@if...@elif...@else...@end 语句

根据表达式的值，有条件地执行一组语句。

```
@if (
   condition1
)
   text1
[@elif (
   condition2
)
   text2]
[@else
   text3]
@end
```

参数

condition1 和 condition2

必选。一个可强迫转换为布尔表达式的表达式。

text1

可选项。*condition1* 为 **true** 时要分析的文本。

text2

可选项。*condition1* 为 **false** 且 *condition2* 为 **true** 时要分析的文本。

text3

可选项。*condition1* 和 *condition2* 均为 **false** 时要分析的文本。

备注

在写 **@if** 语句时，不必将每个子句放到独立的行。可使用多个 **@elif** 子句。但是，所有 **@elif** 子句必须在 **@else** 子句之前出现。

通常使用 **@if** 语句来确定应当使用若干选项中的哪个选项来进行文本输出。

示例

下面的示例阐释了 **@if...@else...@end** 语句的用法。

```
        @if (@_win32)
   print("Operating system is 32-bit.");
@else
   print("Operating system is not 32-bit.");
@end
```

要求

版本 3

请参见

参考

@cc_on 语句

@set 语句

概念

条件编译变量

其他资源

条件编译

if...else 语句

根据表达式的值，有条件地执行一组语句。

```
if (condition)
      statement1
[else
      statement2]
```

参数

condition

必选。布尔表达式。如果 *condition* 为空或未定义，则将 *condition* 视为 **false**。

statement1

必选。*condition* 为 **true** 时要执行的语句。可以是复合语句。

statement2

可选项。*condition* 是 **false** 时要被执行的语句。可以是复合语句。

备注

将 statement1 和 statement2 包含在大括号 ({}) 内通常是一个好的作法，这样就很清楚，并可以避免无意中造成的错误。

示例

在下面的示例中，您可能想将 else 和第一个 if 语句一同使用，但它实际上却是和第二个 if 语句一同使用的。

```
            if (x == 5)
   if (y == 6)
      z = 17;
else
   z = 20;
```

按如下方法更改代码可以消除任何含混不清之处：

```
            if (x == 5)
   {
   if (y == 6)
      z = 17;
   }
else
   z = 20;
```

同样，如果希望向 statement1 添加一个语句但不使用大括号，则可能会意外地产生错误：

```
            if (x == 5)
   z = 7;
   q = 42;
else
   z = 19;
```

在这种情况下，存在一个语法错误，因为在 if 和 else 语句之间有多条语句。在 if 和 else 之间的语句需要大括号。

要求

版本 1

请参见

参考

条件（三元）运算符 (?:)

import 语句

启用对包含在当前脚本或外部库中的命名空间的访问。

```
import namespace
```

参数
namespace

必选。要导入的命名空间的名称。

备注

import 语句在名称提供为 *namespace* 的全局对象上创建属性并将其初始化，以包含对应于所导入命名空间的对象。任何使用 **import** 语句创建的属性都不能赋给其他对象、删除或枚举。所有 **import** 语句都在脚本开始时执行。

import 语句为您的脚本提供命名空间。命名空间可以在脚本中使用 **package** 语句来定义，或者外部程序集可能会提供命名空间。如果没有在脚本中找到命名空间，JScript 将在指定的程序集目录中搜索与命名空间的名称匹配的程序集，除非您正在编译程序并关闭 /autoref 选项。例如，如果导入命名空间 `Acme.Widget.Sprocket` 但该命名空间没有在当前脚本中定义，JScript 将在下列程序集中搜索命名空间：

- `Acme.Widget.Sprocket.dll`
- `Acme.Widget.dll`
- `Acme.dll`

您可以显式地指定要包括的程序集的名称。如果关闭 /autoref 选项或命名空间的名称与程序集名称不匹配，则必须进行这种显式指定。命令行编译器使用 /reference 选项来指定程序集名称，而 ASP.NET 使用 **@ Import** 和 **@ Assembly** 指令来完成此任务。例如，若要显式地包含程序集 mydll.dll，请从命令行键入

```
jsc /reference:mydll.dll myprogram.js
```

若要包括 ASP.NET 页的程序集，需要使用

```
<%@ Import namespace = "mydll" %>
<%@ Assembly name = "mydll" %>
```

当在代码中引用一个类时，编译器首先在局部范围内搜索该类。如果编译器没有找到匹配类，编译器将按照导入顺序搜索每个命名空间中的类，并在找到匹配项时停止。您可以使用类的完全限定名来确定类所派生自的命名空间。

JScript 不会自动导入嵌套命名空间；每个命名空间必须使用完全限定命名空间来导入。例如，若要从一个名为 `Outer` 的命名空间和一个名为 `Outer.Inner` 的嵌套命名空间中访问类，则必须导入这两个命名空间。

示例

下面的示例定义了三个简单的包，并将命名空间导入到脚本中。通常，每个包都位于一个单独的程序集中以便维护和分发包的内容。

```
// Create a simple package containing a class with a single field (Hello).
package Deutschland {
   class Greeting {
      static var Hello : String = "Guten tag!";
   }
};
// Create another simple package containing two classes.
// The class Greeting has the field Hello.
// The class Units has the field distance.
package France {
   public class Greeting {
      static var Hello : String = "Bonjour!";
   }
```

```
   public class Units {
      static var distance : String = "meter";
   }
};
// Use another package for more specific information.
package France.Paris {
   public class Landmark {
      static var Tower : String = "Eiffel Tower";
   }
};

// Declare a local class that shadows the imported classes.
class Greeting {
   static var Hello : String = "Greetings!";
}

// Import the Deutschland, France, and France.Paris packages.
import Deutschland;
import France;
import France.Paris;

// Access the package members with fully qualified names.
print(Greeting.Hello);
print(France.Greeting.Hello);
print(Deutschland.Greeting.Hello);
print(France.Paris.Landmark.Tower);
// The Units class is not shadowed, so it can be accessed with or without a fully qualified
name.
print(Units.distance);
print(France.Units.distance);
```

该脚本的输出为:

```
Greetings!
Bonjour!
Guten tag!
Eiffel Tower
meter
meter
```

要求

.NET 版本

请参见

参考

package 语句
/autoref
/lib
@ Assembly
@ Import

interface 语句

声明接口的名称以及组成接口的属性和方法。

```
[modifiers] interface interfacename [implements baseinterfaces] {
   [interfacemembers]
}
```

参数

modifiers

可选项。控制属性的可见性和行为的修饰符。

interfacename

必选。**interface** 的名称；遵循标准的变量命名规则。

implements

可选项。关键字，指示命名接口实现（或将成员添加到）先前定义的接口。如果未使用此关键字，则将创建一个标准的 JScript 基接口。

baseinterfaces

可选项。由 *interfacename* 实现的接口名称的逗号分隔列表。

interfacemembers

可选。*interfacemembers* 可以是方法声明（用 **function** 语句定义）或属性声明（用 **function get** 和 **function set** 语句定义）。

备注

JScript 中 **interface** 声明的语法类似于 **class** 声明的语法。接口类似于其中的每个成员都为 **abstract** 的 **class**；它只能包含无函数体的属性和方法声明。**interface** 不能包含字段声明、初始值设定项声明或嵌套类声明。一个 **interface** 可以使用 **implements** 关键字来实现一个或多个 **interface**。

一个 **class** 只能扩展一个基 **class**，但一个 **class** 可以实现多个 **interfaces**。一个 **class** 实现多个 **interface** 的这种方式使您可以通过比其他面向对象的语言（如 C++）更简单的形式来实现多继承。

示例

以下代码显示多个接口如何继承一个实现。

```
            interface IFormA {
   function displayName();
}

// Interface IFormB shares a member name with IFormA.
interface IFormB {
   function displayName();
}

// Class CForm implements both interfaces, but only one implementation of
// the method displayName is given, so it is shared by both interfaces and
// the class itself.
class CForm implements IFormA, IFormB {
   function displayName() {
      print("This the form name.");
   }
}

// Three variables with different data types, all referencing the same class.
var c : CForm = new CForm();
var a : IFormA = c;
var b : IFormB = c;
```

```
// These do exactly the same thing.
a.displayName();
b.displayName();
c.displayName();
```

该程序的输出为:

```
This the form name.
This the form name.
This the form name.
```

要求

.NET 版本

请参见

参考

class 语句

function 语句

function get 语句

function set 语句

其他资源

修饰符

JScript 8.0

Labeled 语句

为一条语句提供一个标识符。

```
label :
   [statements]
```

参数

label

必选。在指向作标记的语句时使用的唯一标识符。

statements

可选项。与 label 相关联的一个或多个语句。

备注

标签由 break 和 continue 语句使用，用于指定 break 和 continue 应用于哪个语句。

示例

在下面的语句中，continue 语句使用一个 labeled 语句来创建一个数组，在该数组中，每行的第三列包含一个未定义的值：

```
function labelDemo() {
   var a = new Array();
   var i, j, s = "", s1 = "";
   Outer:
   for (i = 0; i < 5; i++) {
      Inner:
      for (j = 0; j < 5; j++) {
         if (j == 2)
            continue Inner;
         else
            a[i,j] = j + 1;
      }
   }
   for (i = 0;i < 5; i++) {
      s = ""
      for (j = 0; j < 5; j++) {
         s += a[i,j];
      }
      s1 += s + "\n";
   }
   return(s1)
}
```

要求

版本 3

请参见

参考

break 语句

continue 语句

package 语句

创建一个 JScript 包，利用它可以方便地将命名组件打包。

```
package pname {
   [[modifiers1] pmember1]
   ...
   [[modifiersN] pmemberN]
}
```

参数

pname

必选。所创建的包的名称。

modifiers1, ..., modifiersN

可选项。控制 pmember 的可见性和行为的修饰符。

pmember1, ..., pmemberN

可选项。类、接口或枚举定义。

备注

包中只允许有类、接口和枚举。包成员可以使用可见性修饰符来标记，以控制对成员的访问。具体来说，**internal** 修饰符可将成员标记为仅在当前包中可见。

当导入包后，就可以按名称直接访问包成员，除非成员与导入范围可见的另一声明具有相同的名称。当出现这种情况时，必须用成员的包名称来限定该成员。

JScript 不支持声明嵌套包；只有类、接口和枚举声明才能在包中出现。包名称可以包含"."字符来指示应将其视为嵌套在另一个包中。例如，一个名为 `Outer` 的包和一个名为 `Outer.Inner` 的包相互之间不需要具有特殊的关系；它们都是全局范围的包。但是，这些名称意味着 `Outer.Inner` 应视为嵌套在 `Outer` 中。

示例

下面的示例定义了三个简单的包，并将命名空间导入到脚本中。通常，每个包都位于一个单独的程序集中以便维护和分发包的内容。

```
// Create a simple package containing a class with a single field (Hello).
package Deutschland {
   class Greeting {
      static var Hello : String = "Guten tag!";
   }
};
// Create another simple package containing two classes.
// The class Greeting has the field Hello.
// The class Units has the field distance.
package France {
   public class Greeting {
      static var Hello : String = "Bonjour!";
   }
   public class Units {
      static var distance : String = "meter";
   }
};
// Use another package for more specific information.
package France.Paris {
   public class Landmark {
      static var Tower : String = "Eiffel Tower";
   }
};

// Declare a local class that shadows the imported classes.
class Greeting {
```

```
    static var Hello : String = "Greetings!";
}

// Import the Deutschland, France, and France.Paris packages.
import Deutschland;
import France;
import France.Paris;

// Access the package members with fully qualified names.
print(Greeting.Hello);
print(France.Greeting.Hello);
print(Deutschland.Greeting.Hello);
print(France.Paris.Landmark.Tower);
// The Units class is not shadowed, so it can be accessed with or without a fully qualified
name.
print(Units.distance);
print(France.Units.distance);
```

该脚本的输出为:

```
Greetings!
Bonjour!
Guten tag!
Eiffel Tower
meter
meter
```

要求

.NET 版本

请参见

参考

import 语句

internal 修饰符

其他资源

修饰符

print 语句

将字符串以后接一个换行符的形式发送到控制台。

```
function print(str : String)
```

参数

str

可选项。要发送到控制台的字符串。

备注

print 语句使您可以显示用 JScript 命令行编译器 (jsc.exe) 编译的 JScript 程序中的数据。**print** 语句将单个字符串当作一个参数，并且通过将其发送到控制台，以后接换行符的形式显示该字符串。

您可以使用传递到 **print** 语句的字符串中的转义序列对输出进行格式化。转义序列是由反斜杠 (\) 后接字母或位组合构成的字符组合。转义序列可用于指定诸如回车和制表符移动等操作。有关转义序列的更多信息，请参见 **String** 对象主题。当需要对控制台输出的格式进行细微的控制时，可以使用 **System.Console.WriteLine** 方法。

默认情况下，**print** 语句会在 JScript 命令行编译器 jsc.exe 中启用。**print** 语句会在 ASP.NET 中禁用，您可以使用 /print- 选项将其在命令行编译器中禁用。

当没有要输出到的控制台（例如在 Windows GUI 应用程序中），**print** 语句将失败，而不发出任何提示。

print 语句中的输出可以从命令行重定向到一个文件。如果您希望重定向程序的输出，则应在每个输出行的末尾包含 \r 转义符。它会使输出重定向到一个文件中以进行正确格式化，而不影响命令行在控制台上的显示方式。

示例

下面的示例展示 print 语句的用法。

```
var name : String = "Fred";
var age : int = 42;
// Use the \t (tab) and \n (newline) escape sequences to format the output.
print("Name: \t" + name + "\nAge: \t" + age);
```

该脚本的输出为：

```
Name:   Fred
Age:    42
```

请参见

参考

/print

String 对象

Console Class

概念

从命令行程序显示

return 语句

从当前函数退出，并从该函数返回一个值。

```
return[(][expression][)]
```

参数

表达式

可选项。将从该函数返回的值。如果省略，则该函数不返回值。

备注

使用 **return** 语句来终止一个函数的执行，并返回 *expression* 的值。如果 expression 被省略，或在函数内没有执行 **return** 语句，则把 **undefined** 值赋给调用当前函数的表达式。

当执行 **return** 语句时，即使函数体中仍然还有其他语句，此函数也会停止执行。此规则的例外情况是：如果 **return** 语句出现在 **try** 块内而且有一个相应的 **finally** 块，则 **finally** 块中的代码将在此函数返回之前执行。

如果某个函数因到达函数体的结尾时没有执行 **return** 语句而返回，那么返回的值为 **undefined** 值（这就意味着此函数结果不能用作更大表达式的一部分）。

注意

finally 块中的代码是在遇到 **try** 或 **catch** 块中的某个 **return** 语句之后，但在执行该 **return** 语句之前运行的。在这种情况下，**finally** 块中的 **return** 语句是在最初的 **return** 语句之前执行的，这样就允许有不同的返回值。若要避免这种可能会导致混淆的情况，请不要在 **finally** 块中使用 **return** 语句。

示例

下面的示例阐释了 **return** 语句的用法。

```
function myfunction(arg1, arg2){
   var r;
   r = arg1 * arg2;
   return(r);
}
```

要求

版本 1

请参见

参考

function 语句

try...catch...finally 语句

@set 语句

创建使用条件编译语句的变量。

```
@set @varname = term
```

参数

varname

必选。有效的 JScript 变量名。必须总在前面放置一个"@"字符。

term

必选。零个或多个一元运算符，后面有一个常数、条件编译变量或用圆括号括起来的表达式。

备注

在条件编译中支持数字类型和布尔型的变量。而不支持字符串变量。使用 **@set** 创建的变量通常在条件编译语句中使用，但也可在 JScript 代码的任何地方使用。

变量声明的示例如下所示：

```
          @set @myvar1 = 12
@set @myvar2 = (@myvar1 * 20)
@set @myvar3 = @_jscript_version
```

在圆括号括起来的表达式中，支持下面的运算符：

- `! ~`
- `* / %`
- `+ -`
- `<< >> >>>`
- `< <= > >=`
- `== != === !==`
- `& ^ |`
- `&& | |`

如果变量在定义前使用，则它的值为 **NaN**。**NaN** 可通过使用 **@if** 语句进行检查：

```
@if (@newVar != @newVar)
   // ...
```

之所以能这样做是因为 **NaN** 是唯一一个不等于其自身的值。

要求

版本 3

请参见

参考

@cc_on 语句

@if...@elif...@else...@end 语句

概念

条件编译变量

其他资源

条件编译

static 语句

在类声明中声明新的类初始值设定项。

```
static identifier {
   [body]
}
```

参数
identifier

必选。包含初始值设定项块的类的名称。

body

可选项。组成初始值设定项块的代码。

备注

static 初始值设定项用于在首次使用之前对 **class** 对象(而不是对象实例)进行初始化。此初始化只出现一次，可用来对带有 **static** 修饰符的类中的字段进行初始化。

一个 **class** 可以包含多个与 **static** 字段声明交错的 **static** 初始值设定项块。若要将 **class** 初始化，请按其在 **class** 体中出现的顺序执行所有 **static** 块和 **static** 字段初始值设定项。此初始化在首次引用 **static** 字段之前执行。

不要将 **static** 修饰符同 **static** 语句混淆。**static** 修饰符表示属于类本身但不属于任何类成员的成员。

示例

下面的示例显示一个简单的 **class** 声明，其中的 **static** 初始值设定项用于执行只需进行一次的计算。该示例将计算一次阶乘表。当需要阶乘时，将从该表中读取阶乘。如果程序中多次需要较大的阶乘，此方法则要比递归计算阶乘更快。

static 修饰符用于阶乘方法。

```
class CMath {
   // Dimension an array to store factorial values.
   // The static modifier is used in the next two lines.
   static const maxFactorial : int = 5;
   static const factorialArray : int[] = new int[maxFactorial];

   static CMath {
      // Initialize the array of factorial values.
      // Use factorialArray[x] = (x+1)!
      factorialArray[0] = 1;
      for(var i : int = 1; i< maxFactorial; i++) {
         factorialArray[i] = factorialArray[i-1] * (i+1);
      }
      // Show when the initializer is run.
      print("Initialized factorialArray.");
   }

   static function factorial(x : int) : int {
      // Should have code to check that x is in range.
      return factorialArray[x-1];
   }
};

print("Table of factorials:");

for(var x : int = 1; x <= CMath.maxFactorial; x++) {
   print( x + "! = " + CMath.factorial(x) );
}
```

该代码的输出为:

```
Table of factorials:
Initialized factorialArray.
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

要求

.NET 版本

请参见

参考

class 语句

static 修饰符

super 语句

引用当前对象的基对象。它可以在两种上下文中使用。

```
// Syntax 1: Calls the base-class constructor with arguments.
super(arguments)

// Syntax 2: Accesses a member of the base class.
super.member
```

参数

arguments

在语法 1 中为可选。基类构造函数的参数的逗号分隔列表。

member

在语法 2 中为必选。要访问的基类的成员。

备注

super 关键字通常在两种情况中的一种使用。您可以使用它来显式调用具有一个或多个参数的基类构造函数。您还可以使用它来访问已由当前类重写的基类成员。

示例 1

在下面的示例中，**super** 引用基类的构造函数。

```
class baseClass {
   function baseClass() {
      print("Base class constructor with no parameters.");
   }
   function baseClass(i : int) {
      print("Base class constructor. i is "+i);
   }
}
class derivedClass extends baseClass {
   function derivedClass() {
      // The super constructor with no arguments is implicitly called here.
      print("This is the derived class constructor.");
   }
   function derivedClass(i : int) {
      super(i);
      print("This is the derived class constructor.");
   }
}

new derivedClass;
new derivedClass(42);
```

该程序在运行时显示下列输出。

```
Base class constructor with no parameters.
This is the derived class constructor.
Base class constructor. i is 42
This is the derived class constructor.
```

示例 2

在下面的示例中，**super** 允许访问基类的一个重写成员。

```
class baseClass {
   function test() {
```

```
      print("This is the base class test.");
   }
}
class derivedClass extends baseClass {
   function test() {
      print("This is the derived class test.");
      super.test(); // Call the base class test.
   }
}

var obj : derivedClass = new derivedClass;
obj.test();
```

该程序在运行时显示下列输出。

```
This is the derived class test.
This is the base class test.
```

要求

.NET 版本

请参见

参考

new 运算符

this 语句

switch 语句

当指定的表达式的值与一个标签匹配时，使一条或多条语句得以执行。

```
switch (expression) {
      case label1 :
      [statementlist1]
      [break;]
 [ ...
 [ case labelN :
      [statementlistN]
      [break;] ] ]
 [ default :
      [statementlistDefault]]
}
```

参数

表达式

必选。要计算的表达式。

label1, ..., labelN

必选。要与 expression 相匹配的标识符。如果 label === expression，则从冒号后面紧接的语句列表处开始执行，并继续此执行，直到遇到一个可选的 **break** 语句，或到达 **switch** 语句的末尾为止。

statementlist1, ..., statementlistN, statementlistDefault

可选项。要被执行的一个或多个语句。

备注

使用 **default** 子句来提供一个语句，该语句只在没有任何一个标签值与 expression 相匹配时才被执行。它可以出现在 **switch** 代码块内的任何地方。

可以指定零个或多个 label 块。如果没有 label 与 expression 的值匹配，并且没有提供 **default** 子句，则不执行任何语句。

switch 语句将按如下流程执行：

- 计算 expression 的值并依次查看 label，直到找到一个匹配。
- 如果 label 的值等于 expression，则执行其相伴的语句列表。

  继续执行，直到遇到一个 **break** 语句，或直到 **switch** 语句结束。这意味着，如果没有使用 **break** 语句，则将执行多个 *label* 块。
- 如果没有 label 等于 expression，则跳转到 default 子句。如果没有 default 子句，则转到最后一步。
- 在 **switch** 代码块末尾之后的语句处将继续执行过程。

示例

下面的 ASP.NET 示例将测试一个对象的类型。在此例中仅使用了一种类型，但您应该可以清楚地看出该函数是如何使用其他对象类型的。

```
<%@ language="jscript" %>
<%
var d = new Number();
function MyObjectType(obj : Object) : String {
   switch (obj.constructor){
      case Date:
         return "Object is a Date.";
         break;
      case Number:
         return "Object is a Number.";
         break;
```

```
        case String:
           return "Object is a String.";
           break;
        default:
           return "Object is unknown.";
     }
  }
  Response.Write(MyObjectType(d));
  %>
```

要求

版本 3

请参见

参考

break 语句

if...else 语句

JScript 8.0

this 语句

指当前的对象。

```
this.property
```

参数

property

必选。当前对象的属性的标识符。

备注

this 关键字通常用于对象构造函数，用来指当前对象。

示例

在下面的示例中，this 指的是新创建的 Car 对象，并给三个属性赋值：

```
function Car(color, make, model){
   this.color = color;
   this.make = make;
   this.model = model;
}
```

对于 JScript 的客户版本，如果在其他任何对象的上下文之外使用 this，则它指的是 window 对象。

要求

版本 1

请参见

参考

new 运算符

throw 语句

生成一个可由 **try...catch...finally** 语句处理的错误条件。

```
throw [exception]
```

参数
exception

可选项。任何表达式。

备注

使用 **throw** 语句时可不带参数，但是只有在 **throw** 语句包含在 **catch** 块中时才可以这样做。在这种情况下，**throw** 语句会再次引发由封闭 **catch** 语句捕获的错误。提供参数时，**throw** 语句会引发 *exception* 的值。

示例

下面的示例基于传入的值引发一个错误，然后阐释了在 **try...catch...finally** 语句层次中如何处理该错误：

```
function TryCatchDemo(x){
   try {
      try {
      if (x == 0)                              // Evalute argument.
         throw "x equals zero";                // Throw an error.
      else
         throw "x does not equal zero";        // Throw a different error.
      }
      catch(e) {                               // Handle "x=0" errors here.
         if (e == "x equals zero")             // Check for a handled error.
            return(e + " handled locally.");   // Return error message.
         else                                  // Can't handle error here.
            throw e;                           // Rethrow the error for next
      }                                        // error handler.
   }
   catch(e) {                                  // Handle other errors here.
      return(e + " error handled higher up."); // Return error message.
   }
}
print(TryCatchDemo(0)+ "\n");
print(TryCatchDemo(1));
```

要求

版本 5

请参见
参考
try...catch...finally 语句
错误对象

try...catch...finally 语句

为 JScript 实现错误处理。

```
try {
   [tryStatements]
} catch(exception) {
   [catchStatements]
} finally {
   [finallyStatements]}
```

参数

tryStatements

可选项。可能发生错误的语句。

exception

必选。任何变量名称。exception 的初始值是引发的错误的值。

catchStatements

可选项。处理在相关联的 tryStatement 中发生的错误的语句。

finallyStatements

可选项。在所有其他的错误过程发生之后被无条件执行的语句。

备注

try...catch...finally 语句提供了一种方法，可处理给定代码块中可能会发生的一些或全部错误，同时仍保持代码的运行。如果发生了程序员没有处理的错误，JScript 只给用户提供它的一般错误信息，就好象没有错误处理一样。

tryStatements 参数包含可能发生错误的代码，而 *catchStatements* 则包含了可处理任何发生的错误的代码。如果在 *tryStatements* 中发生了一个错误，则将把程序控制传递给 *catchStatements* 来处理该错误。exception 的初始值是发生在 tryStatements 中发生的错误的值。如果不发生错误，则不执行 *catchStatements*。

如果在与发生错误的 *tryStatements* 相关联的 *catchStatements* 中不能处理该错误，则使用 **throw** 语句将这个错误传播或重新引发给更高级的错误处理程序。

在执行完 *tryStatements* 中的语句，并在 *catchStatements* 的所有错误处理发生之后，可无条件执行 *finallyStatements* 中的语句。

请注意，即使 **try** 或 **catch** 块中出现返回语句，或 **catch** 块中引发错误，都会执行 *finallyStatements* 中的代码。*finallyStatments* 一定会始终运行。

示例

下面的示例阐释了 JScript 异常处理是如何进行的。

```
          try {
    print("Outer try running...");
    try {
       print("Nested try running...");
       throw "an error";
    } catch(e) {
       print("Nested catch caught " + e);
       throw e + " re-thrown";
    } finally {
       print("Nested finally is running...");
    }
 } catch(e) {
    print("Outer catch caught " + e);
 } finally {
    print("Outer finally running");
```

```
}
```

将生成以下输出：

```
Outer try running..
Nested try running...
Nested catch caught an error
Nested finally is running...
Outer catch caught an error re-thrown
Outer finally running
```

要求

版本 5

请参见
参考
throw 语句
错误对象

var 语句

声明一个变量。

```
// Syntax for declaring a variable of global scope or function scope.
var name1 [: type1] [= value1] [, ... [, nameN [: typeN] [= valueN] ]]

// Syntax for declaring a variable field within a class.
 [attributes] [modifiers] var name1 [: type1] [= value1] [, ... [, nameN [: typeN] [= value
N].]]
```

参数

attributes

可选项。控制字段的可见性和行为的属性。

modifiers

可选项。控制字段的可见性和行为的修饰符。

name1, ..., nameN

必选。被声明的变量的名字。

type1, ..., typeN

可选项。所声明的变量的类型。

value1, ..., valueN

可选项。赋给变量的初始值。

备注

使用 **var** 语句来声明变量。变量可以绑定到特定的数据类型来确保类型安全。当声明变量时，可以给这些变量赋值，并且这些值可以在以后从脚本中更改。未经显式初始化的变量被赋予默认值 **undefined**（必要时强迫转换为此变量的类型）。

类中的变量字段类似于全局或函数变量，不同的只是它的作用范围是该类，而且它可以用多个修饰符来指定其可见性和用法。

示例

下面的示例阐释了 **var** 语句的某些用法。

```
class Simple {
   // A field declaration of the private Object myField.
   private var myField : Object;
   // Define sharedField to be a static, public field.
   // Only one copy exists, and is shared by all instances of the class.
   static public var sharedField : int = 42;
}
var index;
var name : String = "Thomas Jefferson";
var answer : int = 42, counter, numpages = 10;
var simpleInst : Simple = new Simple;
```

要求

版本 1

请参见

参考

const 语句

function 语句

new 运算符

概念

变量和常数的范围

类型批注
其他资源
修饰符

while 语句

执行一个语句，直到指定的条件为 **false**。

```
while (expression)
      statement
```

参数

表达式

必选。在循环的每次迭代前检查的布尔表达式。如果 expression 是 **true**，则执行循环。如果 expression 是 **false**，则循环终止。

statement

必选。*expression* 为 **true** 时要执行的语句。可以是复合语句。

备注

while 语句在循环第一次执行前检查 expression。如果 expression 此次是 **false**，则该循环永远不执行。

示例

下面的示例阐释了 **while** 语句的用法。

```
function BreakTest(breakpoint){
   var i = 0;
   while (i < 100) {
      if (i == breakpoint)
         break;
      i++;
   }
   return(i);
}
```

要求

版本 1

请参见

参考

break 语句
continue 语句
do...while 语句
for 语句
for...in 语句

with 语句

为语句建立默认对象。

```
with (object)
      statement
```

参数

object

必选。新的默认对象。

statement

必选。*object* 作为其默认对象的语句。可以是复合语句。

备注

with 语句通常用来缩短特定情形下必须写入的代码的数量。

示例

在下面的示例中，请注意 **Math** 的重复使用。

```
var x, y;
x = Math.cos(3 * Math.PI) + Math.sin(Math.LN10);
y = Math.tan(14 * Math.E);
```

当使用 **with** 语句时，代码变得更短且更易于读取：

```
var x, y;
with (Math){
   x = cos(3 * PI) + sin (LN10);
   y = tan(14 * E);
}
```

要求

版本 1

请参见

参考

this 语句

JScript 编译器选项

JScript 编译器生成可执行文件 (.exe) 和动态链接库 (.dll)。

每个编译器选项均以两种形式提供：*-option* 和 */option*。本文档只提供 */option* 形式。

本节内容

JScript 编译器选项（按字母顺序列出）

提供按照字母升序顺序列出的编译器选项列表。

JScript 编译器选项（按类别列出）

提供按照以下类别列出的编译器选项列表：输出文件、.NET Framework 程序集、调试/错误检查、预处理器、资源和杂项。

相关章节

从命令行生成

解释有关从命令行生成 JScript 应用程序的详细信息（如语法和结果）。

编写、编译、调试 JScript 代码

解释如何使用 Visual Studio 集成开发环境 (IDE) 来编写和编辑 JScript 代码。

JScript 编译器选项(按字母顺序列出)

下列编译器选项按字母顺序排序。

编译器选项

选项	用途
@(指定响应文件)	指定响应文件。
/autoref	当声明变量时，如果程序集与导入的命名空间或者与类型批注具有相同的名称，则自动引用程序集。
/codepage	指定要用于编译中所有源代码文件的代码页。
/debug	发出调试信息。
/define	定义预处理器符号。
/fast	生成输出文件，该文件为速度进行了优化，但不支持以前版本中的某些语言功能。
/help, /?	将编译器选项列出到 stdout。
/lcid	为编译器消息指定代码页。
/lib	指定通过 /reference 引用的程序集的位置。
/linkresource	创建到托管资源的链接。
/nologo	不显示编译器版权标志信息。
/nostdlib	不导入标准库 (mscorlib.dll)。
/out	指定输出文件名称。
/platform (JScript)	指定平台类型。
/print	指定 print 语句是否可用。
/reference	从包含程序集的文件中导入元数据。
/resource	将托管资源嵌入程序集。
/target	使用下列三个选项之一指定输出文件的格式:/target:exe/target:library/target:winexe
/utf8output	使用 UTF-8 编码显示编译器输出。
/versionsafe	帮助确保所有重写显式执行。
/warn	设置警告等级。
/warnaserror	将警告提升为错误。
/win32res	将 Win32 资源插入到输出文件中。

请参见

概念

JScript 编译器选项(按类别列出)

从命令行生成

其他资源

JScript 编译器选项

JScript 编译器选项(按类别列出)

下列编译器选项按类别排序。

编译器选项

输出文件

选项	用途
/out	指定输出文件名称。
/target	使用下列三个选项之一指定输出文件的格式:/target:exe/target:library/target:winexe。

.NET Framework 程序集

选项	用途
/autoref	当声明变量时,如果程序集与导入的命名空间或者与类型批注具有相同的名称,则自动引用程序集。
/lib	指定通过 /reference 引用的程序集的位置。
/nostdlib	不导入标准库 (mscorlib.dll)。
/reference	从包含程序集的文件中导入元数据。

调试/错误检查

选项	用途
/debug	发出调试信息。
/lcid	为编译器消息指定代码页。
/versionsafe	帮助确保所有重写显式执行。
/warn	设置警告等级。
/warnaserror	将警告提升为错误。

预处理器

选项	用途
/define	定义预处理器符号。

资源

选项	用途
/linkresource	创建到托管资源的链接。
/resource	将托管资源嵌入程序集。
/win32res	将 Win32 资源插入到输出文件中。

杂项

选项	用途
@(指定响应文件)	指定响应文件。
/codepage	指定要用于编译中所有源代码文件的代码页。
/fast	生成输出文件,该文件为速度进行了优化,但不支持以前版本中的某些语言功能。

/help, /?	将编译器选项列出到 stdout。
/nologo	不显示编译器版权标志信息。
/platform (JScript)	指定平台类型。
/print	指定是否应定义 print 语句。
/utf8output	使用 UTF-8 编码显示编译器输出。

请参见

概念

JScript 编译器选项(按字母顺序列出)

从命令行生成

其他资源

JScript 编译器选项

从命令行生成

可以通过在命令行上键入编译器的可执行文件 (jsc.exe) 的名称来在命令行上启动它。有关更多信息，请参见从命令行编译 JScript 代码。

命令行示例

- 编译 File.js 生成 File.exe：

```
jsc File.js
```

- 编译 File.js 生成 File.dll：

```
jsc /target:library File.js
```

- 编译 File.js 并创建 My.exe：

```
jsc /out:My.exe File.js
```

- 编译 `test.js` 并创建一个 .dll：

```
jsc /target:library test.js
```

请参见

任务

如何：从命令行编译 JScript 代码

其他资源

JScript 编译器选项

@(指定响应文件)

指定响应文件。

```
@response_file
```

参数
response_file

一个列出编译器选项或要编译的源代码文件的文件。

备注

@ 选项使您可以指定包含编译器选项和要编译的源代码文件的文件。编译器将处理这些编译器选项和源代码文件，就如同在命令行上发出命令一样。

若要在一次编译中指定多个响应文件，请指定多个响应文件选项。例如：

```
@file1.rsp @file2.rsp
```

在响应文件中，多个编译器选项和源代码文件可以出现在同一行中。单个编译器选项的指定必须出现在同一行中（不能跨行）。

响应文件可以带有以 # 符号开始的注释。

从响应文件内部指定编译器选项就如同在命令行上发出这些命令。有关更多信息，请参见从命令行生成。

当编译器遇到命令选项时就处理它们，就如同在命令行上发出命令一样。因此，一个响应文件中的选项可能与其他响应文件中的选项或命令行选项不兼容。这可能生成错误。

不能嵌套响应文件。不能将 *@response_file* 放在响应文件中。JScript 编译器遇到这种情况时将报告错误。

示例

以下几行来自一个示例响应文件：

```
# build the first output file
/target:exe /out:MyExe.exe source1.js source2.js
```

请参见
其他资源
JScript 编译器选项

/autoref

当声明变量时，如果程序集与导入的命名空间或者与类型批注具有相同的名称，则自动引用程序集。

```
/autoref[+ | -]
```

参数

+| -

在默认情况下为 on，除非指定 /nostdlib+。如果指定 /autoref+ 或只指定 /autoref，将导致编译器根据导入的命名空间和完全限定名称自动引用程序集。

备注

/autoref 选项命令编译器引用程序集，而不必将程序集传递给 /reference。在使用 import 导入命名空间时，或者在代码中使用完全限定类型名称时，JScript 编译器将搜索包含该类型的程序集。有关 JScript 编译器如何搜索程序集的讨论，请参见 /lib。

如果编译器与要生成的程序的输出文件同名，则它不尝试引用程序集。

示例

下面的程序将在 /autoref+ 有效时编译和运行；编译器将在声明变量时把 System.dll 作为类型批注的一个结果来引用。

```
var s: System.Collections.Specialized.StringCollection =
                    new System.Collections.Specialized.StringCollection();
print(s);
```

下面的程序将在 /autoref+ 有效时编译和运行；编译器将把 System.dll 作为 **import** 语句的一个结果来引用。

```
import System;
var s = new System.Collections.Specialized.StringCollection();
print(s);
```

这些示例还显示编译器如何根据类型批注或 **import** 语句查找程序集名称。当编译器找不到包含 StringCollection 的名为 System.Collections.Specialized.dll 的程序集时，它会查找 System.Collections.dll。如果找不到该文件，它会查找 System.dll，并发现它确实包含 StringCollection。

请参见

参考

import 语句

/reference

其他资源

JScript 编译器选项

/codepage

指定要用于编译中所有源代码文件的代码页。

```
/codepage:id
```

参数
id

代码页的 id，该代码页用于编译中的所有源代码文件。

备注

当要编译的一个或多个源代码文件在创建时没有指定使用计算机上的默认代码页时，可以使用 /codepage 选项指定应该使用的代码页。/codepage 适用于编译中的所有源代码文件。

如果源代码文件是用计算机中的同一有效代码页创建的，或者是用 UNICODE 或 UTF-8 创建的，则不需要使用 /codepage。

请参见
其他资源
JScript 编译器选项

/debug

发出调试信息。

```
/debug[+ | -]
```

参数

+| -

指定 /debug+ 或只指定 /debug 会使编译器生成调试信息并将其放在一个(或多个).pdb 输出文件中。/debug-(如果不指定 /debug，则默认情况下有效)不生成任何调试信息，也不创建包含调试信息的输出文件。

备注

有关如何配置应用程序的调试性能的信息，请参见令映像更易于调试。

示例

将 app.exe 的调试信息放置在 app.pdb 文件中。

```
jsc /debug /out:app.pdb test.js
```

请参见

其他资源

JScript 编译器选项

/define

定义预处理器符号。

```
/define:name1[=value1][,name2[=value1]]
```

参数

name1, *name2*

要定义的一个或多个符号的名称。

value1, *value2*

要采用的符号的值。它们可以是布尔值或数字。

备注

/define 选项在程序中将名称定义为符号。

通过使用逗号分隔符号名称，可以用 /define 定义多个符号。例如：

```
/define:DEBUG,trace=true,max_Num=100
```

有关更多信息，请参见条件编译。

/d 是 /define 的缩写形式。

示例

用 /define:xx 编译。

```
print("testing")
/*@cc_on @*/
/*@if (@xx)
print("xx defined")
@else @*/
print("xx not defined")
/*@end @*/
```

请参见

其他资源

JScript 编译器选项

/fast

提高程序的执行速度。

```
/fast[+ | -]
```

参数

+| -

默认情况下，打开 /fast。/fast 或 /fast+ 使编译器生成速度经过优化的输出文件，但是，如果使用该选项，将不支持以前版本中的某些语言功能。另一方面，指定 /fast- 将提供向后语言兼容性，但是编译器将产生没有经过速度优化的输出文件。

备注

/fast 有效时，

- 必须声明所有变量。
- 函数变成常数，无法给它们赋值或重定义它们。
- 内置对象的预定义属性被标记为 DontEnum、DontDelete、ReadOnly。
- 可能无法展开除 Global 对象（也是全局范围）以外的内置对象上的属性。
- 在函数调用内部不能使用 `arguments` 变量。
- 给只读变量、字段或方法赋值将产生错误。

> **注意**
>
> 提供 /fast- 编译模式的目的是帮助开发人员从旧式 JScript 代码生成独立的可执行文件。当开发新的可执行文件或库时，请使用 /fast+ 编译模式。这样，可确保改进性能以及更好地与其他程序集兼容。

> **安全注意**
>
> /fast- 编译模式允许使用在 /fast+ 模式下无法使用的以前版本中的语言功能。误用这些功能会降低程序的安全性。有关更多信息，请参见 JScript 的安全注意事项。

示例

以牺牲完全的向后语言兼容性为代价，创建经过速度优化的输出文件：

```
jsc test.js
```

请参见

概念

JScript 的安全注意事项

其他资源

JScript 编译器选项

/help, /?

显示编译器的命令行帮助。

```
/help

-or-
/?
```

备注

该选项使编译器显示编译器选项列表以及每个选项的简短说明。

请参见

其他资源

JScript 编译器选项

/lcid

为编译器消息指定代码页。

```
/lcid:id
```

参数
id

代码页的 id，该代码页用于显示来自编译器的消息。

请参见
其他资源
JScript 编译器选项

/lib

指定程序集的引用位置。

```
/lib:dir1[, dir2]
```

参数

dir1

如果引用的程序集未在当前工作目录(在其中调用编译器的目录)或公共语言运行库的系统目录中找到，则编译器将在此目录中进行查找。

dir2

用于搜索程序集引用的一个或多个附加目录。用逗号或分号分隔附加目录的名称。

备注

/lib 选项指定通过 /reference 选项引用的程序集的位置。

编译器按以下顺序搜索未完全限定的程序集引用：

1. 当前工作目录。该目录为从其调用编译器的目录。
2. 公共语言运行库系统目录。
3. 由 /lib 指定的目录。
4. 由 LIB 环境变量指定的目录。

使用 /reference 指定程序集引用。

/lib 是累加的；每一次指定的值都追加到以前的值中。

示例

编译 t2.js 以创建 .exe。编译器将在工作目录和驱动器 C 上根目录中查找程序集引用。

```
jsc /lib:c:\ /reference:t2.dll t2.js
```

请参见

其他资源

JScript 编译器选项

/linkresource

创建到托管资源的链接。

```
/linkresource:filename[,name[,public|private]]
-or-
/linkres:filename[,name[,public|private]]
```

参数

filename

要链接到程序集的资源文件。

name[,public|private]（可选）

资源的逻辑名称；用于加载此资源的名称。默认为文件的名称。还可以指定该文件在程序集清单中是公共的还是私有的。例如，/linkres:filename.res,myname.res,public。默认情况下，*filename* 在程序集中是公共的。

备注

/linkresource 选项不将资源文件嵌入到输出文件中。使用 /resource 选项将一个资源文件嵌入到输出文件中。

例如，如果 *filename* 是由资源文件生成器 (Resgen.exe) 创建或在开发环境中创建的 .NET Framework 资源文件，则可以使用 System.Resources 命名空间中的成员来访问它（有关更多信息，请参见 System.Resources.ResourceManager）。对于所有其他资源，请使用 System.Reflection.Assembly 类中的 GetManifestResource* 方法在运行时访问资源。

filename 可以为任何文件格式。例如，您可能想将本机 DLL 设置为程序集的一部分，以便可将其安装到全局程序集缓存中，并且可从程序集中的托管代码访问它。

/linkres 是 /linkresource 的缩写形式。

示例

编译 `in.js` 并链接到资源文件 `rf.resource`：

```
jsc /linkresource:rf.resource in.js
```

请参见

其他资源

JScript 编译器选项

/nologo

取消显示版权标志信息。

```
/nologo
```

备注

当编译器启动时，/nologo 选项取消版权标志的显示。

请参见

其他资源

JScript 编译器选项

/nostdlib

不导入标准库。

```
/nostdlib[+ | -]
```

参数

+| -

/nostdlib 或 /nostdlib+ 选项导致编译器不导入 mscorlib.dll。如果想定义或创建自己的 System 命名空间和对象，则使用该选项。如果不指定 /nostdlib，mscorlib.dll 将导入程序（与指定 /nostdlib- 相同）。

备注

指定 /nostdlib+ 也就指定了 /autoref-。

示例

如果有一个名为 System 的组件。则能够到达该组件的唯一方式是使用

```
/nostdlib /r:your_library,mscorlib
```

在搜索 mscorlib 之前搜索您的库。通常，不应在应用程序中定义称为 System 的命名空间。

请参见

其他资源

JScript 编译器选项

/out

设置输出文件名。

```
/out:filename
```

参数
filename

由编译器创建的输出文件的名称。

备注

/out 选项指定输出文件的名称。编译器希望在 /out 选项之后找到一个或多个源代码文件。

如果不指定输出文件的名称：

- .exe 将采用用于生成输出文件的第一个源代码文件的名称。
- .dll 将采用用于生成输出文件的第一个源代码文件的名称。

在命令行上，有可能为一次编译指定多个输出文件。所有 /out 选项之后指定的源代码文件都将编译为由该 /out 选项指定的输出文件。

指定要创建的文件的完整名称和扩展名。扩展名必须是 .exe 或 .dll。可以为 /t:exe 项目指定 .dll 扩展名。

示例

编译 `t2.js`，创建输出文件 `t2.exe`，生成 `t3.js`，创建输出文件 `t3.exe`：

```
jsc t2.js /out:t3.exe t3.js
```

请参见
其他资源
JScript 编译器选项

/platform (JScript)

指定公共语言运行库 (CLR) 的哪个版本可以运行程序集。

```
/platform:[string]
```

参数
string

x86、Itanium、x64 或 anycpu（默认值）。

- x86 将程序集编译为由与 x86 兼容的 32 位公共语言运行库运行。
- Itanium 将程序集编译为由采用 Itanium 处理器的计算机上的 64 位公共语言运行库运行。
- x64 将程序集编译为由支持 x64 或 EM64T 指令集的计算机上的 64 位公共语言运行库运行。
- anycpu（默认设置）将程序集编译为在任意平台上运行。

备注

在 64 位 Windows 操作系统上：

- 用 **/platform:x86** 编译的程序集将在运行于 WOW64 下的 32 位 CLR 上执行。
- 用 **/platform:anycpu** 编译的可执行文件将在 64 位 CLR 上执行。
- 用 **/platform:anycpu** 编译的 DLL 将在加载该进程的同一 CLR 上执行。

有关开发在 Windows 64 位操作系统上运行的应用程序的更多信息，请参见 64 位应用程序。

示例

下面的示例演示如何使用 **/platform** 选项来指定应用程序只应当由采用 Itanium 的 64 位 Windows 操作系统上的 64 位 CLR 运行。

```
jsc /platform:Itanium myItanium.js
```

请参见
其他资源
JScript 编译器选项

/print

启用 print 命令。

```
/print[+ | -]
```

参数

+| -

默认情况下，/print 或 /print+ 使编译器能够使用 print 语句。print 语句的一个示例是：

```
print("hello world");
```

指定 /print- 将禁用 print 命令。

备注

如果 .dll 将被加载到没有控制台的环境中，则应使用 /print-。

可以将 Microsoft.JScript.ScriptStream.Out 设置为 TextWriter 对象的实例，以使 print 命令能够将输出发送到其他地方。

示例

使编译器不定义 print 语句。

```
jsc /print- test.js
```

请参见

其他资源

JScript 编译器选项

/reference

导入元数据。

```
/reference:file[;file2]
```

参数
file, file2

一个或多个包含程序集清单的文件。若要导入多个文件，请用逗号或分号分隔文件名。

备注

/reference 选项指示编译器使指定文件中的公共类型信息对于当前正在编译的项目可用。

引用的文件必须是程序集。例如，引用的文件必须用 Visual C#、JScript 或 Visual Basic 中的 /target:library 编译器选项创建，或者用 Visual C++ 中的 /clr /LD 编译器选项创建。

/reference 不能将模块作为输入。

如果引用一个程序集(程序集 A)，而其本身又引用另一个程序集(程序集 B)，则在下列情况下您需要引用程序集 B:

- 使用来自程序集 A 的类型继承自程序集 B 中的类型或实现程序集 B 中的接口。
- 如果调用程序集 B 中具有返回类型或参数类型的字段、属性、事件或方法。

使用 /lib 指定一个或多个程序集引用所在的目录。

为了让编译器识别出程序集中的类型(而不是模块)，需要强制解析类型，例如可以通过定义此类型的实例来完成。也有其他方法来为编译器解析程序集中的类型名称，例如，如果从一个程序集的类型继承，编译器就会知道此类型名称。

/r 是 /reference 的缩写形式。

注意

JScript 编译器 jsc.exe 可引用使用编译器的同一版本或较早版本创建的程序集。但是，JScript 编译器在引用用编译器的较高版本创建的程序集时，可能会遇到编译错误。例如，JScript .NET 2003 编译器可引用用 JScript .NET 2002 编译器创建的任何程序集，但 JScript .NET 2002 编译器在引用用 JScript .NET 2003 创建的程序集时可能失败。

示例

编译源文件 `input.js` 并从 `metad1.dll` 和 `metad2.dll` 导入元数据以生成 `out.exe`:

```
jsc /reference:metad1.dll;metad2.dll /out:out.exe input.js
```

请参见
其他资源
JScript 编译器选项

/resource

将托管资源嵌入程序集。

```
/resource:filename[,name[,public|private]]
-or-
/res:filename[,name[,public|private]]
```

参数
filename

要在输出文件中嵌入的资源文件。

name[,public|private]（可选）

资源的逻辑名称；用于加载此资源的名称。默认为文件的名称。还可以指定该文件在程序集清单中是公共的还是私有的。例如，/res:filename.res,myname.res,public。默认情况下，*filename* 在程序集中是公共的。

备注

使用 /resource 选项将资源链接到程序集，而不将资源文件放入输出文件中。

例如，如果 *filename* 是由资源文件生成器 (Resgen.exe) 创建或在开发环境中创建的 .NET Framework 资源文件，则可以使用 System.Resources 命名空间中的成员来访问它（有关更多信息，请参见 System.Resources.ResourceManager）。对于所有其他资源，请使用 System.Reflection.Assembly 类中的 GetManifestResource* 方法在运行时访问资源。

/res 是 /resource 的缩写形式。

示例

编译 `in.js` 并附加资源文件 `rf.resource`：

```
jsc /res:rf.resource in.js
```

请参见
其他资源
JScript 编译器选项

/target

指定输出文件格式。

/target 编译器选项可以以下列三种形式之一指定：

/target:exe

创建一个控制台 .exe 文件。

/target:library

创建一个 (.dll) 代码库。

/target:winexe

创建一个 Windows 程序。

备注

/target 导致将 .NET Framework 程序集清单放入输出文件中。

如果创建了一个程序集，则可以用 CLSCompliantAttribute 类属性指示全部或部分代码是符合 CLS 的。

```
import System;
[assembly:System.CLSCompliant(true)]   // specify assembly compliance

System.CLSCompliant(true) class TestClass   // specify compliance for element
{
   var i: int;
}
```

请参见

其他资源

JScript 编译器选项

/target:exe

创建控制台应用程序。

```
/target:exe
```

备注

/target:exe 选项使编译器创建可执行 (EXE) 的控制台应用程序。默认情况下，/target:exe 选项有效。将创建带扩展名 .exe 的可执行文件。

除非用 /out 选项另外指定，否则输出文件名称在每个输出文件的编译中将采用第一个源代码文件的名称。

使用 /target:winexe 创建可执行的 Windows 程序。

当在命令行上指定时，直到下一个 /out 或 /target:library 选项之前的所有文件都用于创建 .exe。/target:exe 选项对自上一个 /out 或 /target:library 选项以后的所有文件起作用。

/t 是 /target 的缩写形式。

示例

下列每个命令行都编译 `in.js`，创建 `in.exe`：

```
jsc /target:exe in.js
jsc in.js
```

请参见

参考

/target

其他资源

JScript 编译器选项

/target:library

创建一个代码库。

```
/target:library
```

备注

/target:library 选项使编译器创建 DLL 而不是可执行文件 (EXE)。DLL 创建后会带有 .dll 扩展名。

除非使用 /out 选项另外指定，否则输出文件的名称采用第一个输入文件的名称。

在命令行上指定时，直至下一个 /out 或 /target:exe 选项之前的所有源文件都用于创建 .dll。

/t 是 /target 的缩写形式。

注意

JScript 编译器 jsc.exe 可引用使用编译器的同一版本或较早版本创建的程序集。但是，JScript 编译器在引用用编译器的较高版本创建的程序集时，可能会遇到编译错误。例如，JScript .NET 2003 编译器可引用用 JScript .NET 2002 编译器创建的任何程序集，但 JScript .NET 2002 编译器在引用用 JScript .NET 2003 创建的程序集时可能失败。

示例

编译 `in.js`，创建 `in.dll`：

```
jsc /target:library in.js
```

请参见

参考

/target

其他资源

JScript 编译器选项

/target:winexe

创建一个 Windows 程序。

```
/target:winexe
```

备注

/target:winexe 选项使编译器创建可执行 (EXE) 的 Windows 程序。将创建带扩展名 .exe 的可执行文件。Windows 程序是通过 .NET Framework 库提供用户界面的程序。

使用 /target:exe 创建控制台应用程序。

除非另外用 /out 选项指定，否则输出文件名称在输出文件的编译中采用第一个源代码文件的名称。

在命令行指定该选项时，直至下一个 /out 或 /target 选项之前的所有文件都将用于创建 Windows 程序。

/t 是 /target 的缩写形式。

示例

将 `in.cs` 编译成 Windows 程序：

```
jsc /target:winexe in.js
```

请参见

参考

/target

其他资源

JScript 编译器选项

/utf8output

使用 UTF-8 编码显示编译器输出。

```
/utf8output[+ | -]
```

参数

+| -

默认情况下，/utf8output- 直接在控制台上显示输出。指定 /utf8output 或 /utf8output+ 将编译器输出重定向到文件。

备注

在某些国际配置中，编译器输出无法在控制台上正确显示。在这些配置中，请使用 **/utf8output** 并将编译器输出重定向到文件。

该选项默认为 /utf8output-。

指定 /utf8output 与指定 /utf8output+ 的作用相同。

示例

编译 `in.js` 并使编译器使用 UTF-8 编码显示输出：

```
jsc /utf8output in.js
```

请参见

其他资源

JScript 编译器选项

/versionsafe

标志隐式重写。

```
/versionsafe[+ | -]
```

参数

+| -

默认情况下，/versionsafe- 有效，如果编译器找到隐式方法重写，将不会生成错误。/versionsafe+ 与 /versionsafe 相同，都会使编译器对隐式方法重写生成错误。

备注

使用 Hide 或 Override 关键字显式指示方法的重写状态。例如，当用 /versionsafe 编译时，下列代码将生成错误：

```
class c
{
function f()
{
}
}
class d extends c
{
function f()
{
}
}
```

示例

编译 `in.js` 并使编译器在找到隐式方法重写时生成错误：

```
jsc /versionsafe in.js
```

请参见

其他资源

JScript 编译器选项

/warn

指定警告等级。

```
/warn:option
```

参数

option

想要为生成显示的最小警告等级。有效值为 0 到 4：

警告等级	含义
0	关闭所有警告消息的显示，只显示错误。
1	显示错误和严重的警告消息。
2	显示所有错误、一级警告以及某些不太严重的警告（如有关隐藏类成员的警告）。
3	显示错误、一级和二级警告以及某些不太严重的警告（如有关计算结果总为真或假的表达式的警告）。
4	显示所有错误、1-3 级的警告以及信息性警告。这是命令行中的默认警告等级。

备注

/warn 选项指定编译器要显示的警告等级。

使用 /warnaserror 将低于指定警告等级的所有警告视为错误。忽略等级较高的警告。

编译器始终显示错误。

/w 是 /warn 的缩写形式。

示例

编译 `in.js` 并让编译器仅显示等级 1 警告：

```
jsc /warn:1 in.js
```

请参见

其他资源

JScript 编译器选项

/warnaserror

将警告视为错误。

```
/warnaserror[+ | -]
```

参数

+| -

/warnaserror+ 选项将所有警告视为错误。

备注

任何平常被报告为警告的消息都被报告为错误。不创建任何输出文件。生成过程将继续，以便标识尽可能多的错误/警告。

默认情况下启用 /warnaserror，这导致警告不会妨碍生成输出文件。/warnaserror 与 /warnaserror+ 相同，它使警告被视为错误。

使用 /warn 可指定您希望编译器显示的警告等级。

示例

编译 `in.js` 并且让编译器不显示警告：

```
jsc /warnaserror in.js
```

请参见

其他资源

JScript 编译器选项

/win32res

将一个 Win32 资源插入到输出文件中。

```
/win32res:filename
```

参数
filename

要添加到输出文件的资源文件。

备注

Win32 资源文件可以用资源编译器创建。

Win32 资源可以包含版本或位图(图标)信息，这些信息有助于在 Windows 资源管理器中标识您的应用程序。如果不指定 /win32res，编译器将根据程序集版本生成版本信息。

请参见 /linkresource(用于引用 .NET Framework 资源文件)或 /resource(用于附加 .NET Framework 资源文件)。

示例

编译 `in.js`，并附加 Win32 资源文件 `rf.res` 以生成 `in.exe`：

```
jsc /win32res:rf.res in.js
```

请参见
其他资源
JScript 编译器选项