

SignalR 2.0

—————**SignalR2.0** 开发实战

时间：2014-05-19

By will (黄晓柠)

QQ：137786893

邮箱：will.huangxiaoning@gmail.com

第一节、 入门 ASP.NET SignalR2.0

1、SignalR 简介

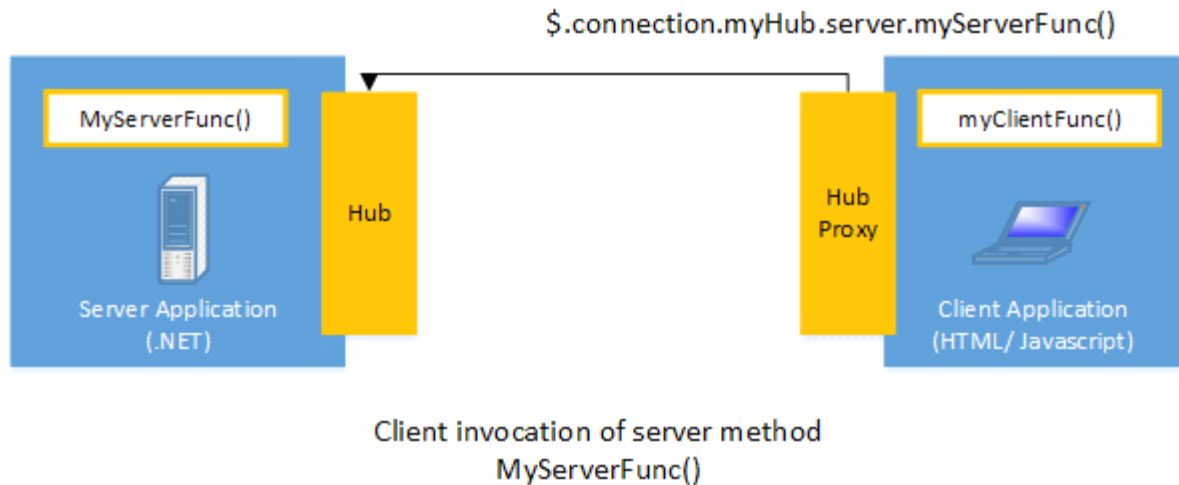
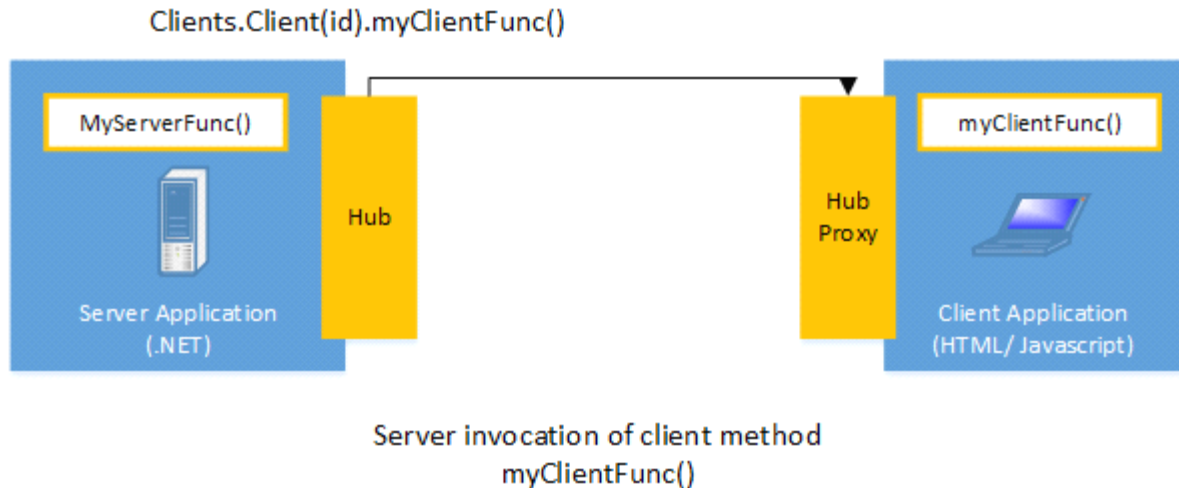
SignalR 是什么？

ASP.NET SignalR 是为 ASP.NET 开发人员提供的一个库，可以简化开发人员将实时 Web 功能添加到应用程序的过程。实时 Web 功能是指这样一种功能：当所连接的客户端变得可用时服务器代码可以立即向其推送内容，而不是让服务器等待客户端请求新的数据。

SignalR 可以用于将任何种类的“实时”Web 功能添加到您的 ASP.NET 应用程序。虽然我们经常把 Chat 应用作为最常用的一个例子，但实际上你可以利用它做很多事情。如果用户是通过刷新 web 页面，来查看新的数据，或者是通过页面实现长轮询来检索新的数据，那么就该考虑使用 SignalR 了。示例包括仪表板和监视应用程序、协作应用程序（例如同时编辑文档）、工作进度更新和实时表单等等。

SignalR 还适用于全新类型的 Web 应用程序，特别是需要从服务器高频率更新的应用程序，例如实时游戏。一个好的例子，请参阅 ShootR 游戏。

SignalR 提供一个简单的 API 用于创建服务器端到客户端的远程过程调用 (RPC)，以便从服务器端 .NET 代码中调用客户端浏览器（以及其他客户端平台）中的 JavaScript 函数。SignalR 还包括用于管理连接（例如，连接和断开连接事件）和为连接分组的 API。



SignalR 会自动管理连接，并允许您像 Chat 室那样向所有连接的客户端同时发送消息。您也可以向特定的客户端发送消息。客户端和服务器之间的连接是持久性的，不像传统的 HTTP 连接——每个通信都需要重新建立一个连接。

SignalR 支持“服务器推送”功能，即服务器代码可以使用远程过程调用（PRC）来调用浏览器中的客户端代码，而不使用目前在 Web 上常用的请求-响应模型。

SignalR 应用程序可以通过使用服务总线、SQL Server 或 Redis 扩展到数以千计的客户端。

SignalR 是开源的，可以通过 GitHub 访问。

SignalR 和 WebSocket

SignalR 会在可能的情况下使用新的 WebSocket 传输方式，并且在需要时回退到旧的传输方式。虽然您仍然可以直接使用 WebSocket 来编写应用程序，但是使用 SignalR 意味着您有许

多现成的额外功能可用，而无需自己实现这些功能。最重要的是，这意味着您可以使用 SignalR 编写应用程序以利用 WebSocket，而无需担心为旧的客户端单独创建代码。SignalR 还能够使您不必担心 WebSocket 的更新，因为 SignalR 会持续更新以支持基础传输方式的更改，为您的应用程序提供一致的接口以使用不同的 WebSocket 版本。

当然，您可以创建只使用 WebSocket 的解决方案，SignalR 为您提供了可能需要自行编写代码的所有功能，例如回退到其他的传输方式以及修订您的应用程序以更新到 WebSocket 实现。

传输和回退

SignalR 是对一组在构建客户端和服务端之间的 real-time 功能所需要使用的传输技术的抽象。SignalR 连接首先以 HTTP 发起请求，然后如果 WebSocket 可用的话，则升级到 WebSocket 连接。WebSocket 是 SignalR 的理想传输方式，因为它能够最高效地使用服务器的内存、有最低的延迟，而且有的最主要的功能（如客户端和服务端之间的全双工通信），但它也有最严格的环境需求：WebSocket 要求服务器是 Windows Server 2012 或 Windows 8 以及 .NET Framework 4.5。如果不满足这些要求，SignalR 将尝试使用其他传输方式来建立连接。

HTML5 传输

传输方式取决于是否支持 HTML 5。如果客户端浏览器不支持 HTML 5 标准，将使用较旧的传输方式。

WebSocket（如果服务器和浏览器都指明它们支持 WebSocket）。WebSocket 是唯一一个在客户端和服务端之间建立真正的持久双向连接的传输方式。然而，WebSocket 也有最严格的要求；只有最新版本的 Microsoft Internet Explorer、Google Chrome 和 Mozilla Firefox 完全支持，其他浏览器如 Opera 和 Safari 只有部分实现。

服务器发送事件，也称为 EventSource（如果浏览器支持服务器发送事件，基本上除了 Internet Explorer 之外其他的浏览器都支持此功能）。

Comet 传输

下列传输基于 Comet Web 应用程序模型，在该模型中有一个浏览器或其他客户端维护着一个长时间的 HTTP 请求，服务器可以在客户端没有明确请求的情况下使用此请求将数据推送到客户端。

Forever Frame（仅限 Internet Explorer）。Forever Frame 会创建一个隐藏的 IFrame，对服务器上的终结点发送一个不完整的请求。然后服务器不断地发送脚本到客户端，并且立即执行这些脚本，从而建立一个从服务器到客户端的单向实时连接。从客户端到服务器的连接使用的

是不同于服务器到客户端的连接，就像一个标准的 HTML 请求，对于每个需要发送的数据都会创建一个新连接。

Ajax 的长轮询。长轮询不会创建一个持久性连接，而是通过一个请求来轮询服务器，使连接保持打开状态直到服务器发出响应，这时候再关闭该连接，然后立即请求一个新的连接。这可能会在连接重置时产生一些延迟。

有关各种配置所支持的传输方式的详细信息，请参见支持的平台。

传输方式选择过程

下面的列表显示 SignalR 决定使用的传输方式的步骤。

如果浏览器是 Internet Explorer 8 或更早版本，则使用长轮询。

如果配置了 JSONP（即连接启动时 jsonp 参数设置为 true ），则使用长轮询。

如果正在使用跨域的连接（即 SignalR 终结点和宿主页不在相同的域中），并且符合以下的条件将使用 WebSocket：

客户端支持 CORS（Cross-Origin Resource Sharing）。哪种客户端支持 CORS 的详细信息，请参阅在 caniuse.com CORS。

客户端支持 WebSocket

服务器支持 WebSocket

如果这些标准中的任何一条不满足，将使用长轮询。跨域连接的详细信息，请参阅如何建立跨域的连接。

如果不配置为使用 JSONP、连接不跨域并且客户端和服务器都支持，将使用 WebSocket。

如果客户端或服务器不支持 WebSocket，则尽量使用服务器发送事件。

如果服务器发送事件不可用，则将尝试使用 Forever Frame。

如果 Forever Frame 不可用，则使用长轮询。

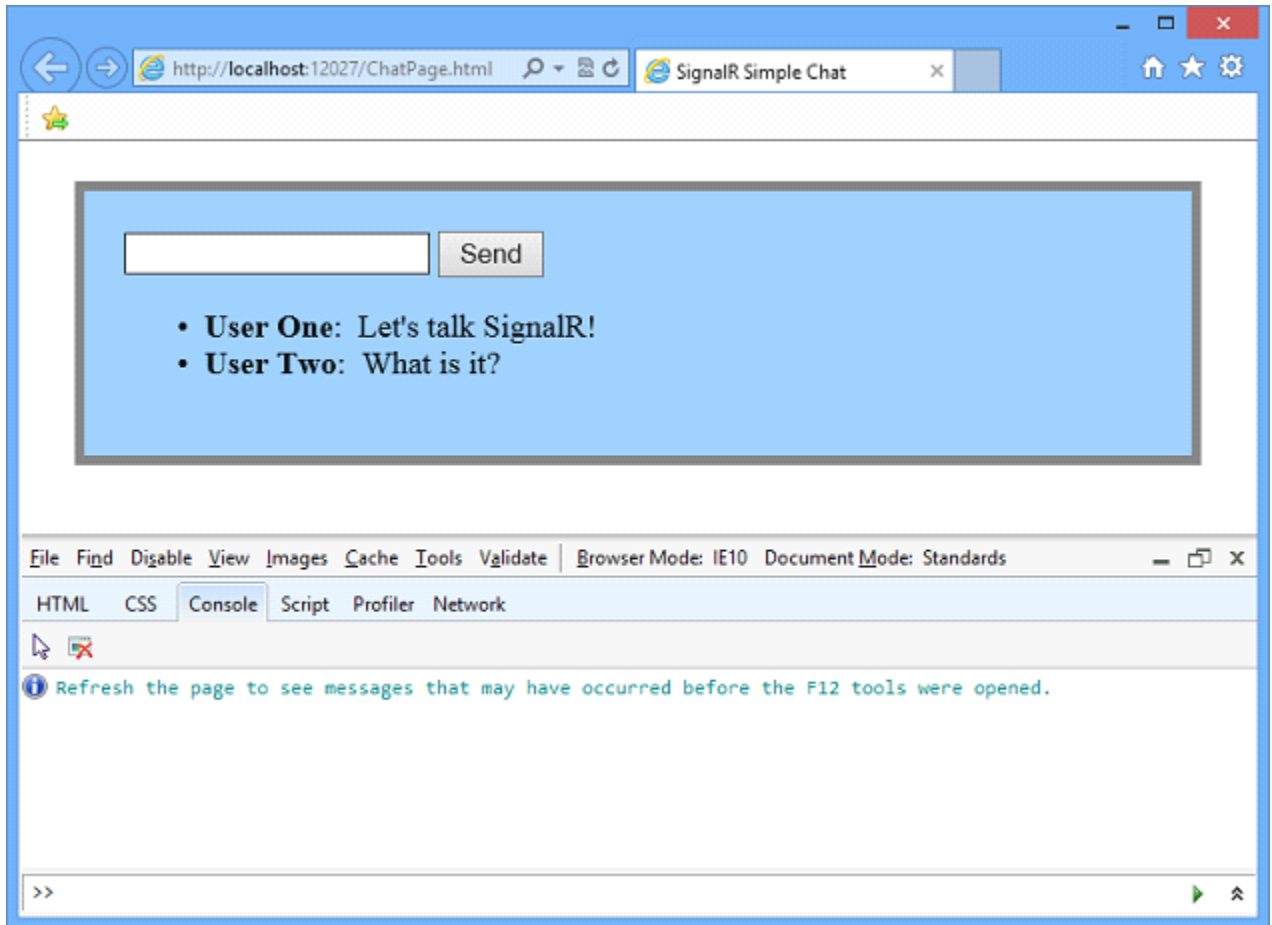
Monitoring 传输方式

您可以通过在您的应用程序 hub 启用日志记录，并在您的浏览器的控制台窗口中查看您的应用程序使用哪种传输协议。

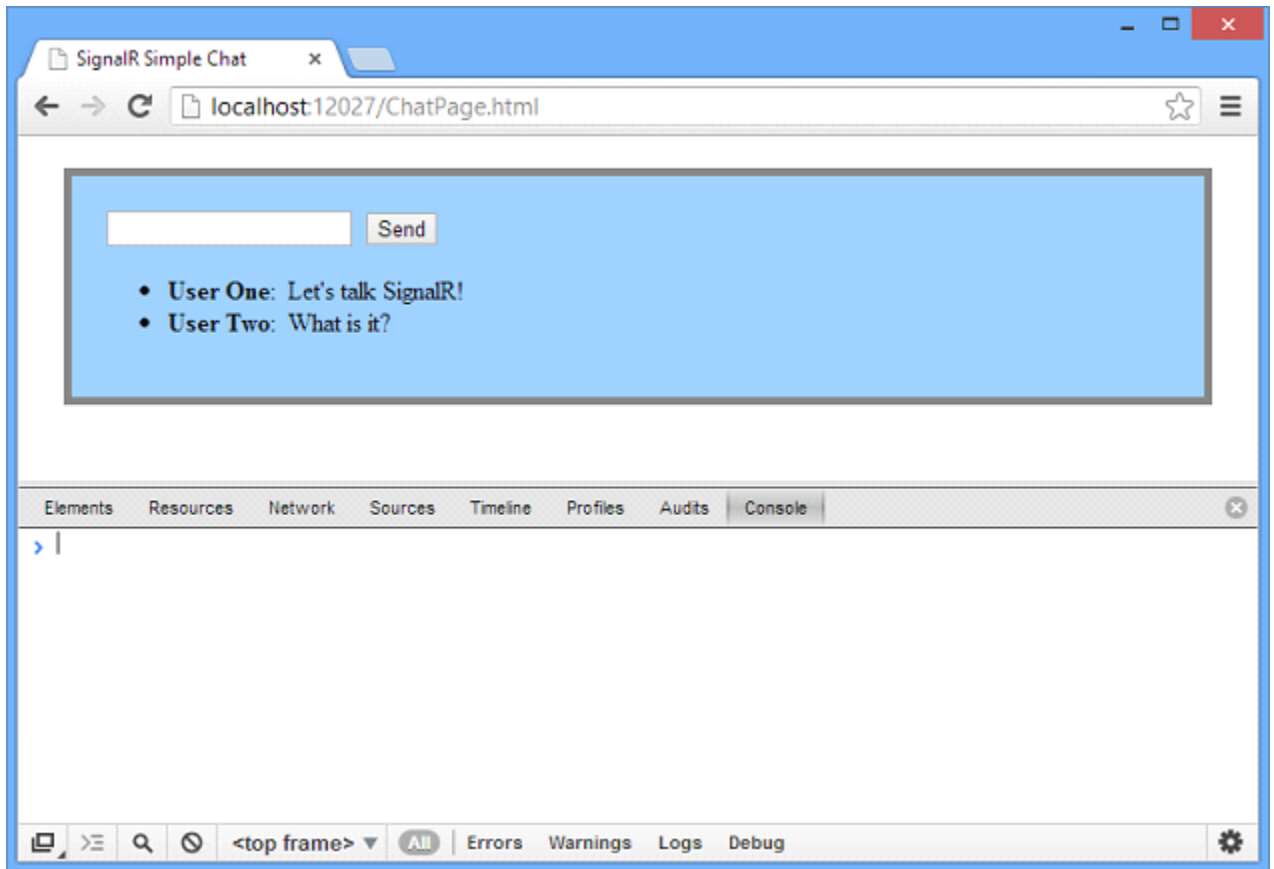
将下面的命令添加到您的客户端应用程序，以在浏览器中启用 hub 事件的日志记录：

```
$.connection.myHub.logging = true;
```

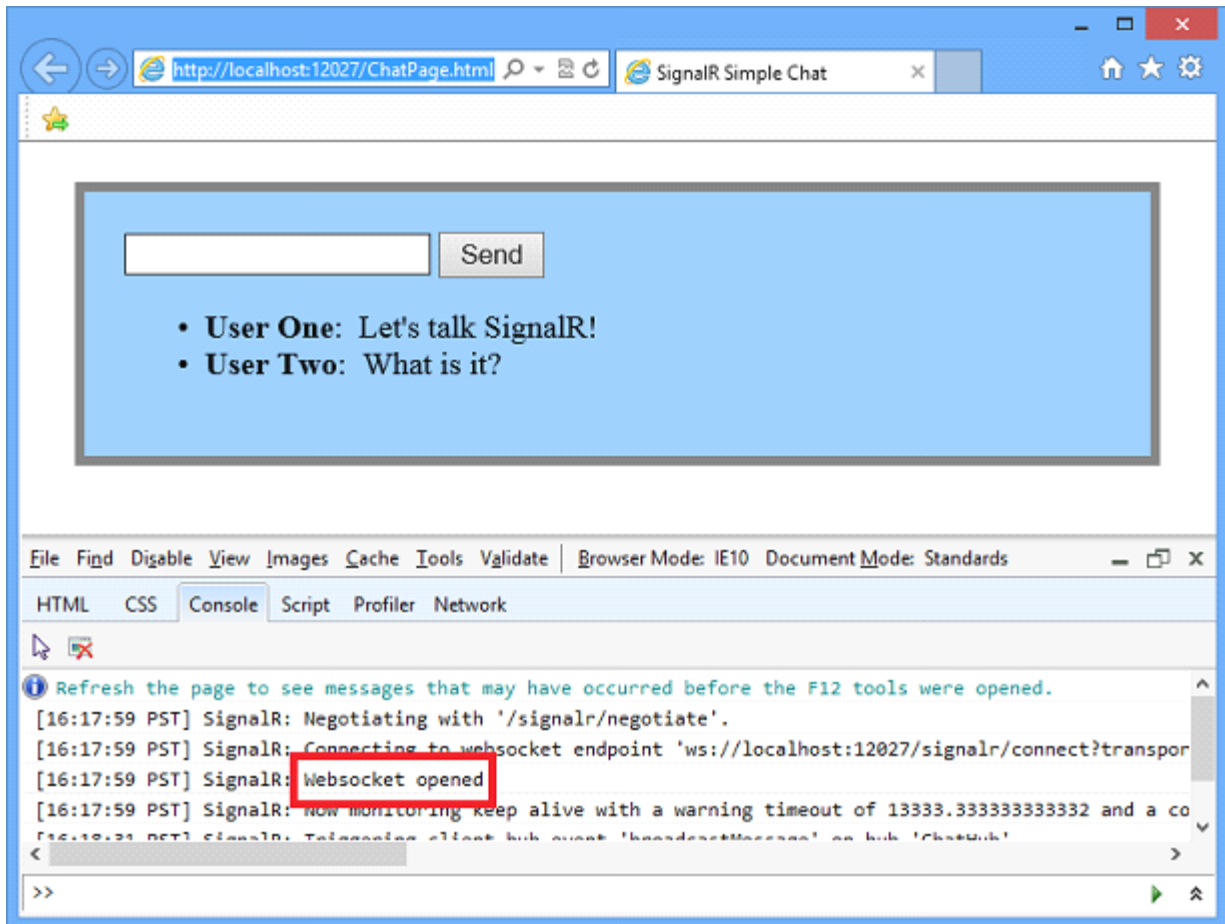
在 IE 中，通过按 f12 键，可以打开开发人员工具，然后单击控制台选项卡。



在 Chrome，按下 Ctrl + Shift + J 打开控制台。



通过控制台的日志记录，你就能够看到 SignalR 正在使用的传输协议。



指定传输协议

使用固定的传输协议需要一定的时间和客户端以及服务器的资源。如果客户端环境已知，那么当启动客户端连接时就可以指定传输协议。下面的代码段演示如何在已知客户端不支持任何其他协议时，直接在连接开始时就使用 Ajax 的长轮询：

```
connection.start({ transport: 'longPolling' });
```

如果你想要一个客户端按照特定顺序尝试传输方式，您可以指定尝试的顺序。下面的代码段演示如何尝试使用 WebSocket 并且在失败的时间去直接使用长轮询。

```
connection.start({ transport: ['webSockets', 'longPolling'] });
```

用于指定传输方式的字符串常量的定义如下：

```
webSockets
```

```
forverFrame
```

```
serverSentEvents
```


longPolling

连接和 Hubs

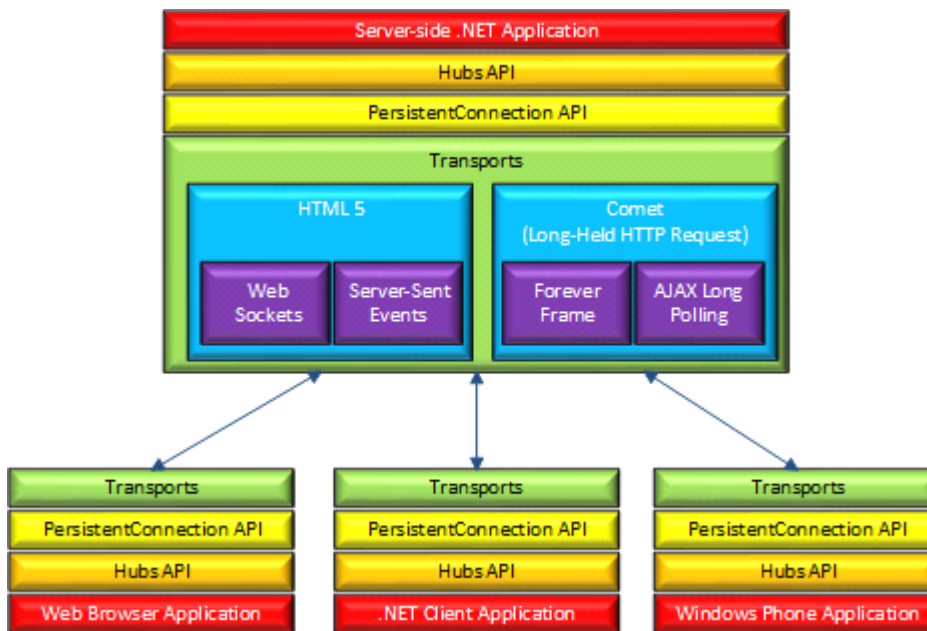
SignalR API 包含两种客户端和服务端之间进行通信的模型：永久连接和 Hubs。

连接表示为一个发送单个、编组或广播消息的简单终结点。开发人员可以使用持久性连接 API（在 .NET 代码中由 `PersistentConnection` 类表示）直接访问 SignalR 公开的底层通信协议的。使用过基于连接 Api 比如 wcf 的开发人员会更加熟悉连接通信模型。

Hub 是基于连接 API 的但是更高级别的通信管线，它允许客户端和服务端上彼此直接调用方法。SignalR 能够很神奇地处理跨机器的调度，使得客户端能够调用在服务器上的方法就像轻松地调用本地方法，反之亦然。使用过基于远程调用的 Api 比如 .NET Remoting 的开发人员会更加熟悉 Hubs 通信模型。使用 Hub 还允许您将强类型的参数传递给方法并且绑定模型。

体系结构关系图

下面的关系图显示了 Hub、持续连接和用于传输的基础技术之间的关系。



Hub 的工作原理

当服务器端代码调用客户端上的方法时，服务器发送一个数据包其中包含的要调用的方法的名称与参数（当被发送的方法参数包含对象时，它将被序列化为 JSON）。然后，客户端通过方法的名称在客户端代码中定义的方法中尝试匹配。如果匹配成功，则将执行客户端方法并且使用经过反序列化的参数数据。

可以使用 Fiddler 之类的工具监视方法的调用。下图显示了 Fiddler 的日志窗格中从 SignalR 的服务器发送到 web 浏览器客户端的方法调用。从 Hub 发起调用的方法称为 MoveShapeHub，被调用的方法是 updateShape。

```
10:38:03:1298 Session846.WebSocket'WebSocket #846' - Pushing 104 bytes from server WebSocket
81 66 7B 22 43 22 3A 22 42 2C 31 35 7C 43 2C 30 f{"C":"B,15|C,0
7C 44 2C 30 7C 45 2C 30 22 2C 22 4D 22 3A 5B 7B |D,0|E,0","M":[{"
22 48 22 3A 22 4D 6F 76 65 53 68 61 70 65 48 75 "H":"MoveShapeHu
62 22 2C 22 4D 22 3A 22 75 70 64 61 74 65 53 68 b","M":"updateSh
61 70 65 22 2C 22 41 22 3A 5B 7B 22 6C 65 66 74 ape","A":[{"left
22 3A 35 30 31 2E 30 2C 22 74 6F 70 22 3A 33 30 ":501.0,"top":30
32 2E 30 7D 5D 7D 5D 7D 2.0}}]}
```

在此示例中，Hub 名称使用参数 H 标识；方法名称使用参数 M 标识，同时正在发送给该方法的数据使用参数 A 标识。生成此消息的应用程序是在 High-Frequency Realtime 教程中创建的。

选择通信模型

大多数应用程序应使用 Hub 的 API。连接 API 可用于以下情况：

发送的实际消息需要指定的格式。

开发人员更喜欢使用消息传递和调度模型，而不是一个远程调用模型。

使用 SignalR 移植的现有的应用程序正在使用消息传递模型。

2、支持的平台

SignalR 在各种服务器和客户端配置下的支持。此外，每种传输体式格式都有自身请求限制；若是某种传输体式格式不被体系支持，SignalR 可以或许优雅地将故障转移到其他类型的传输体式格式。关于 SignalR 所支持的传输体式格式的具体信息，请参阅：[Transports and Fallbacks](#)。

服务器系统要求

SignalR 服务器组件可以驻留在不同的服务器配置。本节介绍了操作系统、.NET 框架、Internet 信息服务器和其他组件的受支持的版本。

支持的服务器操作系统

SignalR 服务器组件可以驻留在下列服务器或客户端操作系统。请注意使用 WebSockets 的 SignalR，为 Windows 服务器 2012 年或 Windows 8 需要（WebSocket 可以使用网站上的 Windows Azure，只要该网站的 .NET framework 版本设置为 4.5，并在该站点的配置页面中启用了 Web 套接字）。

- Windows 2012 Server
- Windows Server 2008 r2
- Windows 8
- Windows 7
- Windows Azure

支持的服务器 .NET 框架版本

在 .NET Framework 4.5 上只支持 SignalR 2.0。请参阅增强的可靠性、兼容性、稳定性和性能的更新推荐的更新部分。

受支持的服务器 IIS 版本

- IIS 8 或 IIS 8 Express。

-
- IIS 7 和 IIS 7.5, 须要 extensionless URLs 支持。
-
- IIS 必须在集成模式下运行, 不支持经典模式。当 IIS 运行在经典模式时, 应用办事器发送事务可能带来30秒的消息延迟。
-
- 托管应用法度必须运行在信赖模式下。

客户端系统要求

SignalR 可以使用在各种客户端平台。本节介绍在 web 浏览器, Windows 桌面应用程序、Silverlight 应用程序和移动设备中使用 SignalR 的系统要求。

- IE8 , 9 , 10 , 11 桌面及移动, Win8的 Modern, 版本都支持。
-
- 火狐: 当前版本-1, Win 及 Mac 版本。
-
- Chrome: 当前版本-1, Win 及 Mac 版本。
-
- Safari: 当前版本-1, Mac 及 iOS 版本。
-
- Opera: 当前版本-1, 仅限 Win 版本。
-
- 安卓浏览器。

除了要求某些浏览器, SignalR 使用的各种运输有他们自己的要求。下列运输支持下以下配置:

Web 浏览器的运输要求运输	互联网资源管理器	Chrome (Windows 或 iOS)	火狐浏览器	Safari (OSX 或 iOS)	Android 系统
WebSockets	10 +	当前值-1	当前值-1	当前值-1	不适用
服务器发送事件	不适用	当前值-1	当前值-1	当前值-1	不适用
ForeverFrame	8 +	不适用	不适用	不适用	4.1

长轮询	8 +	当前值-1	当前值 -1	当前值-1	4.1
-----	-----	-------	-----------	-------	-----

不支持的浏览器

固然在旧版本的浏览器中运行 SignalR 中可能不会有重大题目,但我们不会主动去测试 SignalR 在旧浏览器中的运行景象,也不会解决同旧浏览器的兼容题目。

第二节、教程：入门 SignalR 2.0 和 MVC5

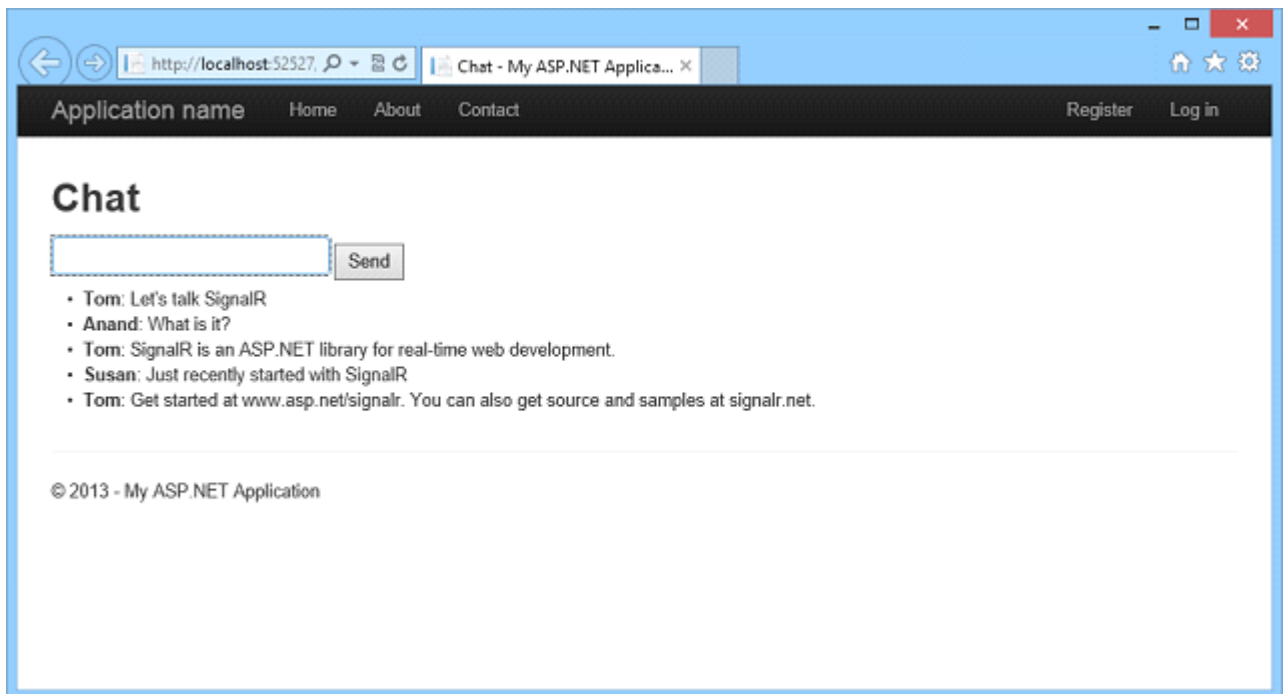
1、概述

本教程向您介绍与 ASP.NET SignalR 2.0 和 ASP.NET MVC 5 实时 web 应用程序开发。本教程使用相同的 Chat 应用程序代码作为 SignalR 入门教程，但显示了如何将它添加到 MVC 5 应用程序。

在本主题中，您将学习下面的 SignalR 开发任务：

- 向 MVC 5 应用程序添加 SignalR 类库。
- 创建 hub 和 OWIN 启动类，将内容推送到客户端。
- 在 web 页中使用 SignalR jQuery 库发送消息并显示。

下面的屏幕快照显示了在浏览器中运行的已完成的 Chat 应用程序。



章节：

- [创建项目](#)

- 运行示例
- 检查代码

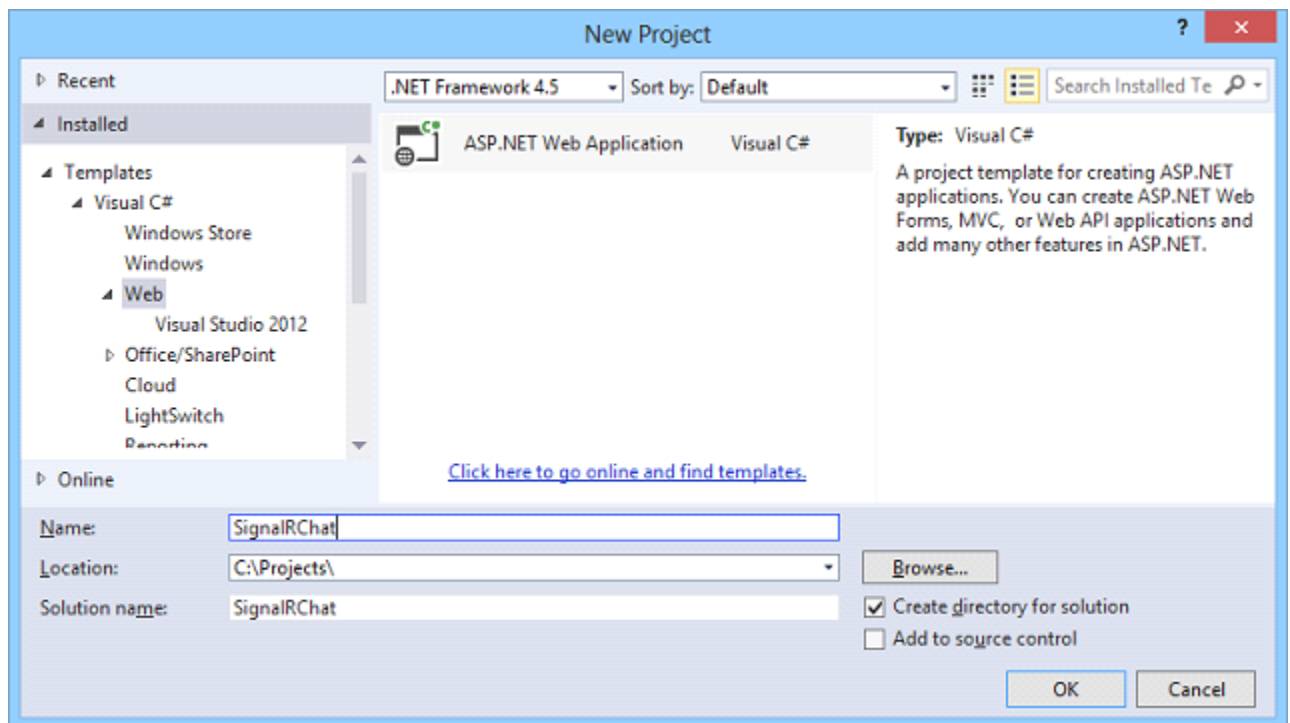
2、创建项目

先决条件:

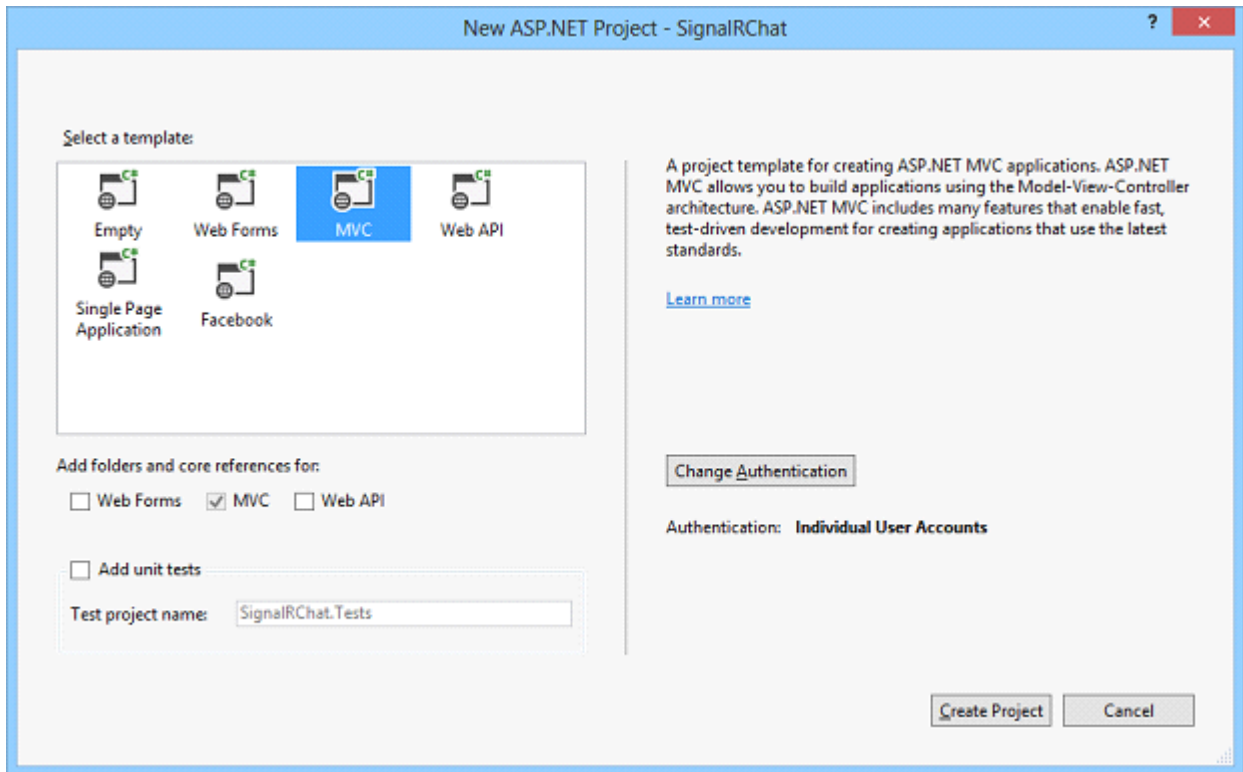
- Visual Studio 2013 年。如果您不具有 Visual Studio，请参见 [ASP.NET 下载](#) 获取免费 Visual Studio 2013 工具。

本节演示如何创建一个 ASP.NET MVC 5 应用程序、添加 SignalR 库中，并创建 Chat 应用程序。

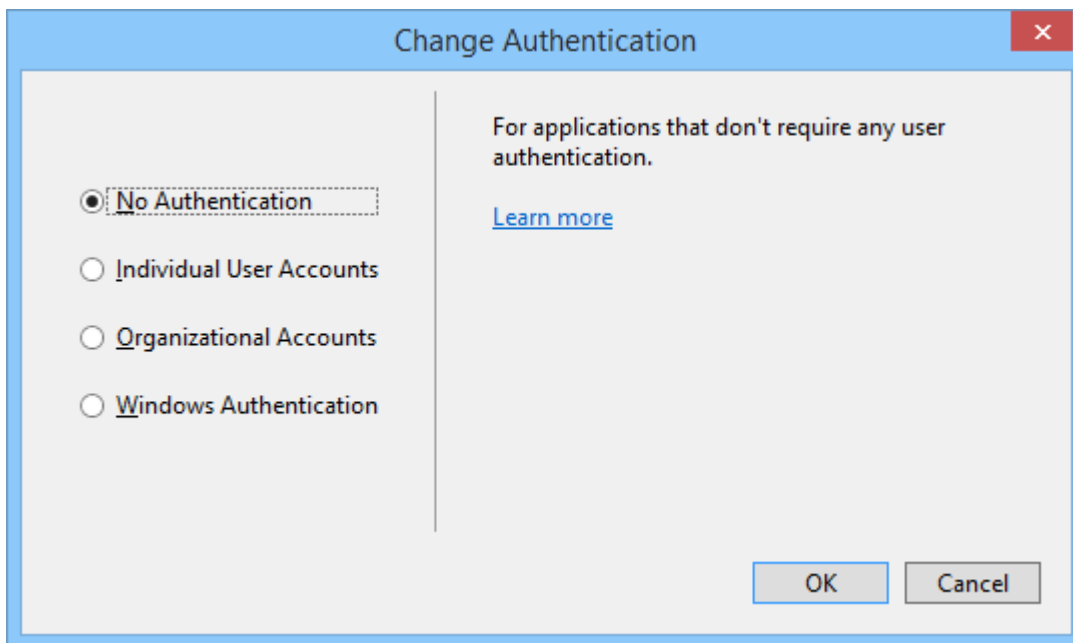
1. 在 Visual Studio 中创建一个 C# ASP.NET 应用程序的目标 .NET 框架 4.5，命名为 SignalRChat，然后单击确定。



2. 在 New ASP.NET Project 对话框的，和选择 MVC，单击更改身份验证。



3. 在更改身份验证对话框中，选择无身份验证，并单击确定。

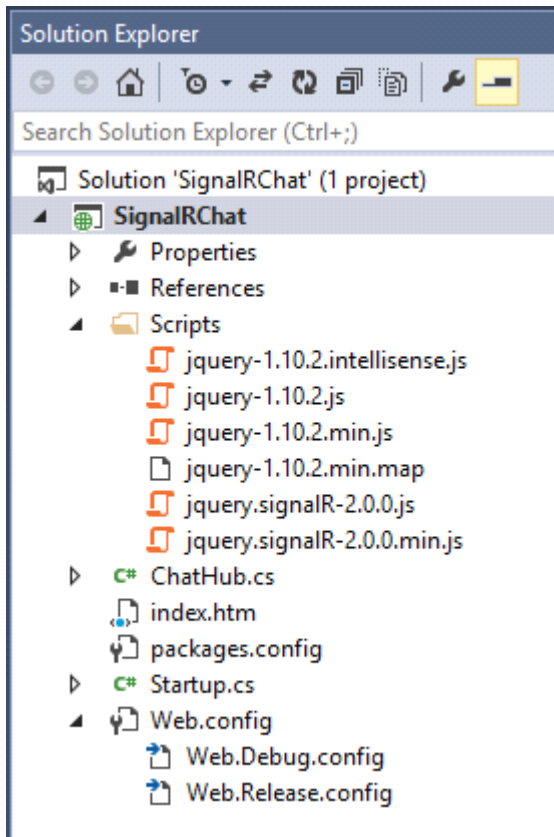


注：如果您选择一个不同的身份验证提供程序为您的应用程序，将会为您；创建一个 Startup.cs 类你不需要在下面的步骤 10 中创建您自己的 Startup.cs 类。

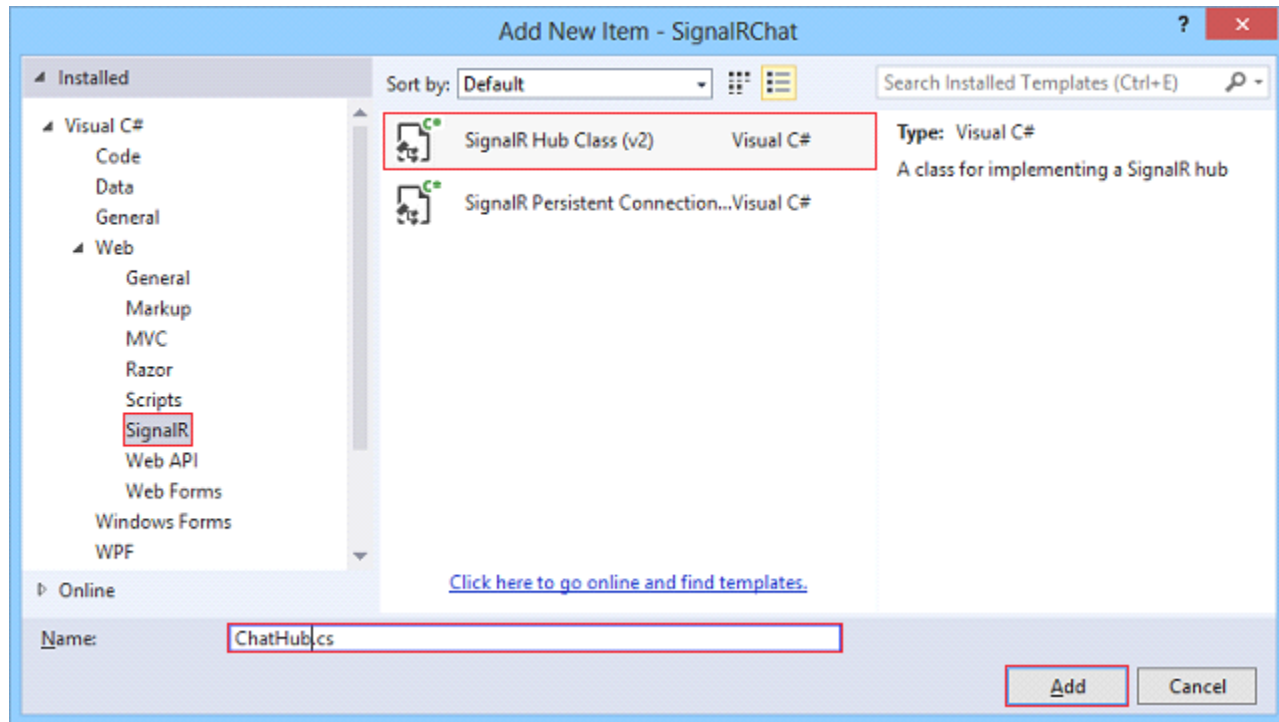
4. 在新的 **ASP.NET** 项目对话框中单击**确定**。
5. 打开工具 | **Library Package 管理器 | 程序包管理器控制台** 并运行下面的命令。此步骤向项目中添加一组脚本文件和启用 SignalR 功能的程序集引用。

```
install-package Microsoft.AspNet.SignalR
```

6. 在**解决方案资源管理器**中，展开脚本文件夹。请注意 SignalR 的脚本库已被添加到项目中。



7. 在**解决方案资源管理器**中，右键单击项目，选择添加**新文件夹**，并添加一个新的文件夹名为 **Hubs**。
8. 用鼠标右键单击**枢纽文件夹**，请单击添加**新项目**，选择 **Visual C# | Web | SignalR** 在已安装窗格中的节点从中心窗格中，选择 **SignalR Hub 类 (v2)**和创建一个名为 **ChatHub.cs** 的新 hub。您将使用此类将消息发送到所有客户端的 SignalR 服务器枢纽。



1

9. **ChatHub** 类中的代码替换为以下代码。

```
using System;
using System.Web;
using Microsoft.AspNet.SignalR;
namespace SignalRChat
{
    public class ChatHub : Hub
    {
        public void Send(string name, string message)
        {
            // Call the addNewMessageToPage method to update clients.
            Clients.All.addNewMessageToPage(name, message);
        }
    }
}
```

10. 创建一个称为 **Startup.cs** 的新类。将文件的内容更改为以下内容。

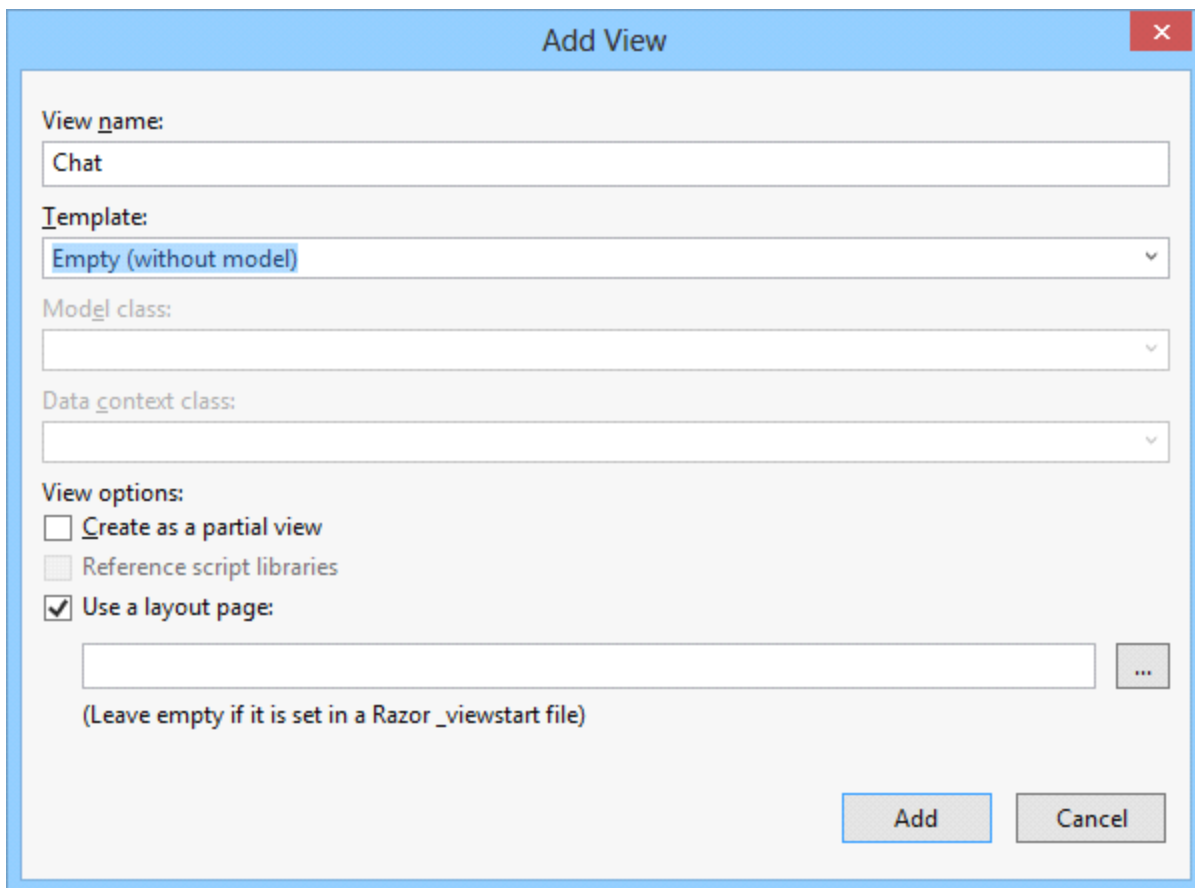
```
using Owin;
using Microsoft.Owin;
[assembly: OwinStartup(typeof(SignalRChat.Startup))]
namespace SignalRChat
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Any connection or hub wire up and configuration should go here
            app.MapSignalR();
        }
    }
}
```

11. 编辑 `HomeController` 类在 `Controllers/HomeController.cs` 中找到，也可将以下方法添加到类。此方法返回的 `Chat` 的视图，您将在稍后的步骤中创建。

```
public ActionResult Chat()
{
    return View();
}
```

12. 右键单击视图/主文件夹，并选择添加。...|查看。

13. 在添加视图对话框中，将新视图命名 `Chat`。



Add View

View name:
Chat

Template:
Empty (without model)

Model class:

Data context class:

View options:

Create as a partial view

Reference script libraries

Use a layout page:

...

(Leave empty if it is set in a Razor _viewstart file)

Add Cancel

14. Chat.cshtml 的内容替换为以下代码。

重要： 将 SignalR 和其他脚本库添加到您的 Visual Studio 项目时，程序包管理器可能会安装是显示本主题中的版本比更近的 SignalR 脚本文件的一个版本。请确保您的代码中的脚本引用匹配的版本安装在您的项目中的脚本库。

```

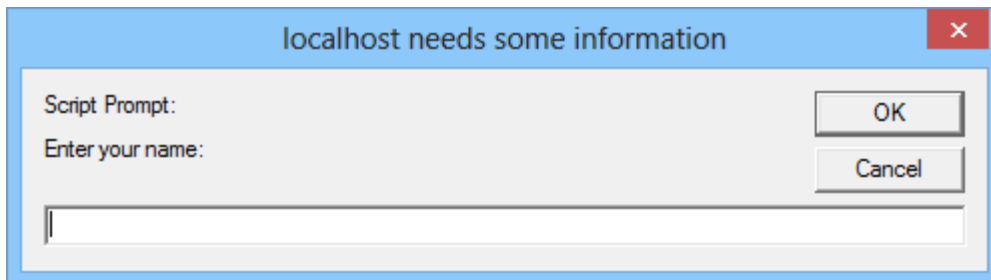
@{
    ViewBag.Title = "Chat";
}
<h2>Chat</h2>
<div class="container">
    <input type="text" id="message" />
    <input type="button" id="sendmessage" value="Send" />
    <input type="hidden" id="displayname" />
    <ul id="discussion">
    </ul>
</div>
@section scripts {
    <!--Script references. -->
    <!--The jQuery library is required and is referenced by default in _Layout.cshtml. -->
    <!--Reference the SignalR library. -->
    <script src="~/Scripts/jquery.signalR-2.0.3.min.js"></script>
    <!--Reference the autogenerated SignalR hub script. -->
    <script src="~/signalr/hubs"></script>
    <!--SignalR script to update the chat page and send messages.-->
    <script>
        $(function () {
            // Reference the auto-generated proxy for the hub.
            var chat = $.connection.chatHub;
            // Create a function that the hub can call back to display messages.
            chat.client.addNewMessageToPage = function (name, message) {
                // Add the message to the page.
                $('#discussion').append('<li><strong>' + htmlEncode(name)
                    + '</strong>: ' + htmlEncode(message) + '</li>');
            };
            // Get the user name and store it to prepend to messages.
            $('#displayname').val(prompt('Enter your name:', ''));
            // Set initial focus to message input box.
            $('#message').focus();
            // Start the connection.
            $.connection.hub.start().done(function () {
                $('#sendmessage').click(function () {
                    // Call the Send method on the hub.
                    chat.server.send($('#displayname').val(), $('#message').val());
                    // Clear text box and reset focus for next comment.
                    $('#message').val('').focus();
                });
            });
        });
        // This optional function html-encodes messages for display in the page.
        function htmlEncode(value) {
            var encodedValue = $('<div />').text(value).html();
            return encodedValue;
        }
    </script>
}

```

15. 保存所有的项目。

3、运行示例

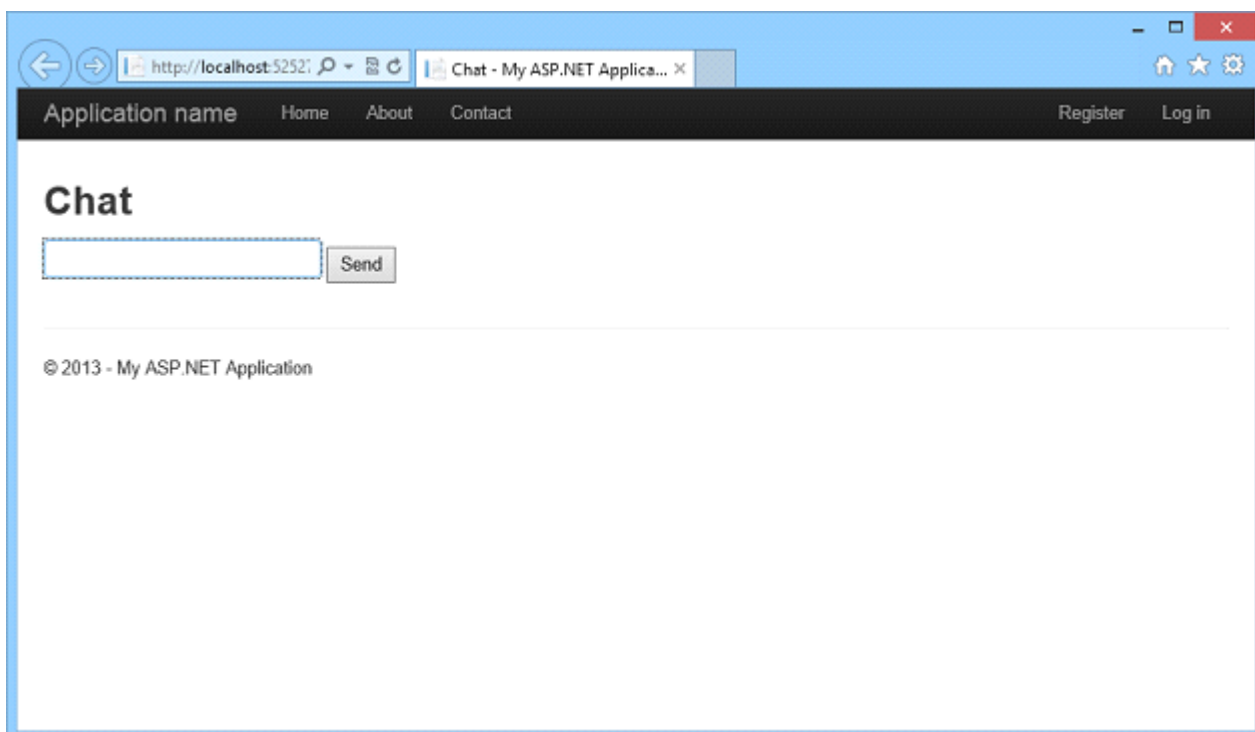
1. 按 F5 键在调试模式下运行该项目。
2. 在浏览器地址行中，将 **/home/chat** 追加到该项目的默认页的 URL。Chat 页加载在浏览器实例和用户名称的提示。



3. 输入用户的名称。
4. 从浏览器的地址行复制的 URL，并使用它来打开两个更多的浏览器实例。在每个浏览器实例，请输入一个唯一的用户名称。
5. 在每个浏览器实例中添加注释，单击发送。评论应在所有的浏览器实例中显示。

注：这个简单的 Chat 应用程序并不维护服务器上的讨论范围。Hub 广播到所有当前用户的评论。晚些时候加入这次 Chat 的用户将看到消息添加时间从他们加入。

6. 下面的屏幕快照显示了在浏览器中运行的 Chat 应用程序。



7. 在[解决方案资源管理器](#)，检查正在运行的应用程序的**脚本文档**节点。如果您正在使用 Internet Explorer 作为您的浏览器，该节点是在调试模式下可见。有一个名为 SignalR 库在运行时动态生成的 Hub 的脚本文件。此文件管理 jQuery 脚本和服务器端的代码之间的通讯。如果你使用 Internet Explorer 之外的浏览器，也可以通过它直接，例如 `http://mywebsite/signalr/hubs` 浏览访问动态 **Hub** 文件。

检查代码

SignalR Chat 应用程序演示了两个基本的 SignalR 开发任务：创建一个 Hub 作为主要协调对象在服务器上，并使用 SignalR jQuery 库发送和接收消息。

SignalR Hubs

在 **ChatHub** 的代码示例从 **Microsoft.AspNet.SignalR.Hub** 类派生的类。从 **Hub** 类派生的是生成一个 SignalR 应用程序的有用方法。可以创建您的 Hub 类的公共方法，然后通过调用它们从一个 web 页中的脚本访问这些方法。

在 Chat 代码中，客户端调用 **ChatHub.Send** 方法将发送一封新邮件。**Hub** 反过来将消息发送给所有的客户端通过调用 **Clients.All.addNewMessageToPage**。

发送方法演示中心的几个概念：

Hub 上声明的公共方法，这样客户端可以调用它们。

- 使用 **Microsoft.AspNet.SignalR.Hub.Clients** 属性来访问所有客户端连接到该集 **Hub**。在客户端（如 `addNewMessageToPage` 函数）上调用一个函数来更新客户端。

```
public class ChatHub : Hub
{
    public void Send(string name, string message)
    {
        Clients.All.addNewMessageToPage(name, message);
    }
}
```

SignalR 和 jQuery

Chat.cshtml 查看文件中的代码示例演示如何使用 SignalR jQuery 库与 SignalR **Hub** 进行通信。在代码中的基本任务在创建自动生成的代理为枢纽，声明一个函数，该服务器可以调用内容推送到客户端，并启动连接将消息发送到 Hub 的引用。

下面的代码声明一个 Hub 代理的引用。

```
var chat = $.connection.chatHub;
```

注：在 JavaScript 对服务器类和其成员的引用是在 camel 大小写。该代码示例作为 **chatHub** 引用在 JavaScript 中的 C# **ChatHub** 类。如果你想要引用的 **ChatHub** 类中与传统的 Pascal jQuery 套管如你将在 C# 中，编辑 **ChatHub.cs** 的类文件。添加 **using** 语句来引用 **Microsoft.Microsoft.AspNet.SignalR.Hubs** 命名空间。然后将添加 **HubName** 属性的 **ChatHub** 类，例如 **[HubName("ChatHub")]**。最后，您 jQuery 引用更新为 **ChatHub** 类。

下面的代码演示如何在脚本中创建一个回调函数。在服务器上的 Hub 类调用这个函数以将内容更新推送到每个客户端。可选调用 **htmlEncode** 函数显示一个 HTML 的方式显示它在页面中，作为一种方式来防止脚本注入前对消息内容进行编码。

```
chat.client.addNewMessageToPage = function (name, message) {  
    // Add the message to the page.  
    $('#discussion').append('<li><strong>' + htmlEncode(name)  
        + '</strong>: ' + htmlEncode(message) + '</li>');  
};
```

下面的代码演示如何使用 Hub 打开一个连接。代码启动连接，然后将它传递一个函数来处理 Chat 页面中的发送按钮上的单击事件。

注：此方法可确保该事件处理程序在执行之前建立连接。

```
$.connection.hub.start().done(function () {  
    $('#sendmessage').click(function () {  
        // Call the Send method on the hub.  
        chat.server.send($('#displayname').val(), $('#message').val());  
        // Clear text box and reset focus for next comment.  
        $('#message').val('').focus();  
    });  
});
```


第三节、 ASP.NET SignalR Hubs API

1、如何注册 SignalR 中间件

若要定义的路由的客户端将用于连接到您的 Hub，请调用 `MapSignalR` 方法，当应用程序启动时。

`MapSignalR` 是 `OwinExtensions` 类的扩展方法。

```
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(MyApplication.Startup))]
namespace MyApplication
{
    public class Startup
    {
        {
            public void Configuration(IAppBuilder app)
            {
                // Any connection or hub wire up and configuration should go here
                app.MapSignalR();
            }
        }
    }
}
```

2、如何创建和使用 Hub 类

若要创建一个 Hub，创建从 `Microsoft.AspNet.Signalr.Hub` 派生的类。下面的示例演示一个简单的 Hub 类为一个 Chat 应用程序。

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addNewMessageToPage(name, message);
    }
}
```

在此示例中，连接的客户端可以调用 `NewContosoChatMessage` 方法，以及当它，所收到的数据广播给所有连接的客户端。

Hub 中的名称的客户端 JavaScript 的 camel 大小写格式

默认情况下，客户端 JavaScript Hub 使用引用的类名称的缩写版本。SignalR 自动使这种更改，以便 JavaScript 代码可以符合 JavaScript 约定。前面的示例将称为 `contosoChatHub` 中的 JavaScript 代码。

服务器

```
public class ContosoChatHub : Hub
```

使用生成的代理的 **JavaScript** 客户端

```
var contosoChatHubProxy = $.connection.contosoChatHub;
```

如果你想要指定一个不同的名称为客户端使用，添加 `HubName` 属性。当您使用 `HubName` 属性时，对 JavaScript 客户端上的 camel 大小写没有更改。

服务器

```
[HubName("PascalCaseContosoChatHub")]public class ContosoChatHub : Hub
```

使用生成的代理的 **JavaScript** 客户端

```
var contosoChatHubProxy = $.connection.PascalCaseContosoChatHub;
```

3、如何在客户端可以调用的 Hub 类中定义的方法

若要公开上的 Hub 的你想要从客户端调用的方法，请声明一个公共方法，如下面的示例中所示。

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addNewMessageToPage(name, message);
    }
}
```

```
public class StockTickerHub : Hub
{
    public IEnumerable<Stock> GetAllStocks()
    {
        return _stockTicker.GetAllStocks();
    }
}
```

您可以指定返回类型和参数，其中包括复杂类型和数组，正如你将会在任何 C# 方法。您接收的参数或返回到调用方的任何数据在客户端和服务端之间传达的信息使用 JSON，和 SignalR 自动处理复杂对象的绑定和对象的数组。

Camel 大小写格式的方法名称的 JavaScript 客户端

默认情况下，客户端 JavaScript 引用 Hub 方法的使用方法名称的缩写版本。SignalR 自动使这种更改，以便 JavaScript 代码可以符合 JavaScript 约定。

服务器

```
public void NewContosoChatMessage(string userName, string message)
```

使用生成的代理的 **JavaScript** 客户端

```
contosoChatHubProxy.server.newContosoChatMessage(userName, message);
```

如果你想要指定一个不同的名称为客户端使用，添加 `HubMethodName` 属性。

服务器

```
[HubMethodName("PascalCaseNewContosoChatMessage")]
```

```
public void NewContosoChatMessage(string userName, string message)
```

使用生成的代理的 **JavaScript** 客户端

```
contosoChatHubProxy.server.PascalCaseNewContosoChatMessage(userName, message);
```

异步执行

如果该方法将长时间运行或要做的工作，将涉及等待，如数据库查询或 web 服务调用，使该 Hub 方法通过返回异步任务(代替 `void` 返回) 或任务 `<T>`(代替 `T` 返回类型) 的对象。当您从该方法返回一个 `Task` 对象时，SignalR 等待 `Task` 完成，然后它将解开的结果发送回客户端，所以 `t` 在这里是如何你的代码中，客户端的方法调用上无差异。

使得 Hub 方法异步可避免阻止连接，当它使用 WebSocket 运输。当一个 Hub 方法同步执行和运输是 WebSocket 时，Hub 从同一客户端上的方法的后续调用都被阻止，直到 Hub 方法完成。

下面的示例显示相同的方法编码来运行同步或异步，其次是用于调用任一版本的 JavaScript 客户端代码。

同步

```
public IEnumerable<Stock> GetAllStocks(){ // Returns data from memory.
return _stockTicker.GetAllStocks();}
```

异步

```
public async Task<IEnumerable<Stock>> GetAllStocks()
{
    // Returns data from a web service.
    var uri = Util.GetServiceUri("Stocks");
    using (HttpClient httpClient = new HttpClient())
    {
        var response = await httpClient.GetAsync(uri);
        return (await response.Content.ReadAsAsync<IEnumerable<Stock>>());
    }
}
```

使用生成的代理的 **JavaScript** 客户端

```
stockTickerHubProxy.server.getAllStocks().done(function (stocks) {
    $.each(stocks, function () {
        alert(this.Symbol + ' ' + this.Price);
    });
});
```

4、如何从 Hub 类调用客户端方法

若要从服务器调用客户端方法，在您 Hub 类中的方法中使用 **Clients** 属性。下面的示例演示对所有连接的客户端调用 **addNewMessageToPage** 的服务器代码和 JavaScript 客户端中定义的方法的客户端代码。

服务器

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addNewMessageToPage(name, message);
    }
}
```

使用生成的代理的 **JavaScript** 客户端

```
contosoChatHubProxy.client.addNewMessageToPage = function (name, message) {
    // Add the message to the page.
    $('#discussion').append('<li><strong>' + htmlEncode(name)
        + '</strong>: ' + htmlEncode(message) + '<li>');
};
```

您不能从客户端方法获取返回值如语法 `int x = Clients.All.add(1,1)` 无法运行。

您可以指定复杂类型和数组的参数。下面的示例将一个复杂类型传递给方法的参数中的客户端。

调用客户端方法使用一个复杂对象的服务器代码

```
public void SendMessage(string name, string message)

{

Clients.All.addContosoChatMessageToPage(new ContosoChatMessage()

{

    UserName = name,

    Message = message });

}
```

定义的复杂对象的服务器代码

```
public class ContosoChatMessage
{
    public string UserName { get; set; }
    public string Message { get; set; }
}
```

使用生成的代理的 **JavaScript** 客户端

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addMessageToPage = function (message) {
    console.log(message.UserName + ' ' + message.Message);
});
```

5、选择哪些客户端将会收到 RPC

客户端属性将返回一个 `HubConnectionContext` 对象，提供用于指定哪些客户端将收到 RPC 的几个选项：

- 所有连接的客户端。

```
Clients.All.addContosoChatMessageToPage(name, message);
```

- 仅调用客户端。

```
Clients.Caller.addContosoChatMessageToPage(name, message);
```

- 除了调用客户端的所有客户端。

```
Clients.Others.addContosoChatMessageToPage(name, message);
```

- 由连接 ID 来标识特定的客户端

```
Clients.Client(Context.ConnectionId).addContosoChatMessageToPage(name, message);
```

本示例对调用客户端调用 `addContosoChatMessageToPage`，并已使用 `Clients.Caller` 的效果相同。

- 所有连接的客户端除了指定的客户端，由连接 ID 标识。

```
Clients.AllExcept(connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
```

- 指定组中的所有已连接客户端。

```
Clients.Group(groupName).addContosoChatMessageToPage(name, message);
```

- 指定组中的所有连接的客户端除了指定的客户端，由连接 ID 标识。

```
Clients.Group(groupName, connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
```

- 指定组中的所有连接的客户端除了调用客户端。

```
Clients.OthersInGroup(groupName).addContosoChatMessageToPage(name, message);
```

- 特定用户，由用户 Id 标识。

```
Clients.User(userid).addContosoChatMessageToPage(name, message);
```

- 可以使用 IUserId 接口来确定用户 Id:

```
public interface IUserIdProvider{    string GetUserId(IRequest request);}
```

`IUserIdProvider` 的默认实现是 `PrincipalUserIdProvider`。若要使用此默认实现，在首次注册 `GlobalHost` 应用程序时启动它：

```
var idProvider = new PrincipalUserIdProvider();
```



```
GlobalHost.DependencyResolver.Register (typeof(IUserIdProvider), () => idProvider);
```

然后可以通过从客户端传递的 `Request` 对象中确定的用户名称。

- 所有客户端和组列表中的连接 Id。

```
Clients.Clients(ConnectionIds).broadcastMessage(name, message);
```

- 组的列表。

```
Clients.Groups(GroupIds).broadcastMessage(name, message);
```

6、如何从 Hub 类管理组成员身份

SignalR 中的组提供广播消息到连接的客户端的指定子集的方法。一个组可以有任意数量的客户端，而客户端可以任意数量的组的成员。

管理组成员，请使用添加和删除方法的 Hub 类 `Groups` 属性提供的。下面的示例演示在 Hub 方法调用的客户端代码，其次是 JavaScript 调用它们的客户端代码中使用的 `Groups.Add` 和 `Groups.Remove` 方法。

服务器

```
public class ContosoChatHub : Hub
{
    public Task JoinGroup(string groupName)
    {
        return Groups.Add(Context.ConnectionId, groupName);
    }

    public Task LeaveGroup(string groupName)
    {
        return Groups.Remove(Context.ConnectionId, groupName);
    }
}
```

使用生成的代理的 **JavaScript** 客户端

```
contosoChatHubProxy.server.joinGroup(groupName);  
  
contosoChatHubProxy.server.leaveGroup(groupName);
```

7、如何处理在 Hub 类连接生存期事件

处理连接生存期事件的典型原因是是否有用户连接或不，跟踪的并要跟踪的用户名称和连接 Id 之间的关联。若要运行您自己的代码，当客户端连接或断开连接时，重写的 **OnConnected**，**OnDisconnected** 和 **OnReconnected** 虚拟 Hub 的类的方法，如下面的示例中所示。

```
public class ContosoChatHub : Hub
{
    public override Task OnConnected()
    {
        // Add your own code here.
        // For example: in a chat application, record the association between
        // the current connection ID and user name, and mark the user as online.
        // After the code in this method completes, the client is informed that
        // the connection is established; for example, in a JavaScript client,
        // the start().done callback is executed.
        return base.OnConnected();
    }

    public override Task OnDisconnected()
    {
        // Add your own code here.
        // For example: in a chat application, mark the user as offline,
        // delete the association between the current connection id and user name.
        return base.OnDisconnected();
    }

    public override Task OnReconnected()
    {
        // Add your own code here.
        // For example: in a chat application, you might have marked the
        // user as offline after a period of inactivity; in that case
        // mark the user as online again.
        return base.OnReconnected();
    }
}
```

OnConnected、 OnDisconnected 和 OnReconnected 称为

浏览器导航到一个新页面，每次一个新的连接具有建立，这意味着 SignalR 将执行的 **OnDisconnected** 方法，其次是 **OnConnected** 方法。SignalR 总是会创建一个新的连接 ID 时建立新的连接。

SignalR 可以从自动恢复，如当电源是暂时断开连接和重新连接连接超时之前的连接已临时休息时调用 **OnReconnected** 方法。当客户端断开连接和 SignalR 不能自动重新连接，例如当浏览器导航到新的一页时调用 **OnDisconnected** 方法。因此，给定客户端的事件可能顺序是 **OnConnected**、**OnReconnected**、**OnDisconnected**；或 **OnConnected**，**OnDisconnected**。你看不到的序列 **OnConnected**、**OnDisconnected**、**OnReconnected** 为一个给定的连接。

`OnDisconnected` 调用方法，并不会在某些情况下，如当服务器停机或应用程序域得到回收利用。当另一台服务器来行上或应用程序域完成其循环时，某些客户端可能无法重新连接，然后触发 `OnReconnected` 事件。

更多的信息，请参阅[了解和处理连接生存期事件在 SignalR](#)。

8、如何从上下文属性获取有关客户端的信息

若要获取有关客户端的信息，请使用 Hub 类的 `Context` 属性。该 `Context` 属性返回一个 `HubCallerContext` 对象，提供对以下信息的访问：

调用客户端的连接 ID。

```
string connectionID = Context.ConnectionId;
```

连接 ID 是一个由 SignalR（你无法在您自己的代码中指定值）分配的 GUID。有一个连接和 ID 的每个连接，如果您有多个 Hub 在您的应用程序中所有 Hub 都使用 ID 相同的连接。

- HTTP 标头数据。

```
System.Collections.Specialized.NameValueCollection headers =  
Context.Request.Headers;
```

您还可以从 `Context.Headers` 的 HTTP 标头。同样的事情的多个引用的理由是 `Context.Headers` 第一次创建、以后，增加了 `Context.Request` 属性，`Context.Headers` 保留了向后兼容性。

- 查询字符串数据。

```
System.Collections.Specialized.NameValueCollection queryString =  
Context.Request.QueryString;
```

```
string parameterValue = queryString["parametername"];
```

您还可以从 `Context.QueryString` 获取查询字符串数据。

你在此属性中的查询字符串是在使用中的 HTTP 请求的 SignalR 连接建立的那个。通过配置的连接，这是一种方便的途径，关于客户端的数据从客户端传递到服务器，您可以在客户端中添加查询字符串参数。下面的示例演示一种方法在 JavaScript 客户端中添加一个查询字符串，当您使用生成的代理。

```
$.connection.hub.qs = { "version" : "1.0" };
```

- Cookie。

```
System.Collections.Generic.IDictionary<string, Cookie> cookies =  
Context.Request.Cookies;
```

您还可以从 `Context.RequestCookies` 获取 cookie。

- 用户信息。

```
System.Security.Principal.IPrincipal user = Context.User;
```

- 对请求的 `HttpContext` 对象：

```
System.Web.HttpContextBase httpContext = Context.Request.GetHttpContext();
```

使用此方法而不是 `HttpContext.Current` 的 `HttpContext` 对象获取 SignalR 的连接。

9、如何进行客户端和 Hub 类之间传递状态

客户端代理服务器提供，您可以在其中存储您想要传输到与每个方法调用服务器的数据的 `state` 对象。在服务器上可以访问此数据的 `Clients.Caller` 属性中 Hub 由客户端调用的方法。

`Clients.Caller` 属性不填充为连接生存期事件处理程序方法 `OnConnected`、`OnDisconnected`、`OnReconnected`。

创建或更新的 `state` 对象和 `Clients.Caller` 属性中的数据在两个方向的工作。您可以更新服务器中的值和它们传递回客户端。

下面的示例显示了存储状态传输到服务器，每个方法调用的 JavaScript 客户端代码。

```
contosoChatHubProxy.state.userName = "Fadi Fakhouri";
contosoChatHubProxy.state.computerName = "fadivm1";
```

下面的示例演示了在 .NET 客户端中的等效的代码。

```
contosoChatHubProxy["userName"] = "Fadi Fakhouri";

chatHubProxy["computerName"] = "fadivm1";
```

在 Hub 类中，您可以访问此数据在 `Clients.Caller` 属性中。下面的示例演示代码检索前面的示例中所述的状态。

```
public void NewContosoChatMessage(string data)
{
    string userName = Clients.Caller.userName;
    string computerName = Clients.Caller.computerName;
    Clients.Others.addContosoChatMessageToPage(message, userName, computerName);
}
```

10、如何在 Hub 类中处理错误

若要处理在 Hub 类方法中出现的错误，请使用一个或多个以下方法：

在 `try-catch` 块中换行方法代码和日志的异常对象。出于调试目的您可以向客户端发送异常但安全原因将详细的信息发送到客户端在生产中不建议使用。

创建 Hub 管道处理模块的 `OnIncomingError` 方法。下面的示例演示一个管道模块，记录错误，其次为 Hub 管道注入模块的 `Startup.cs` 中的代码。

```
public class ErrorHandlingPipelineModule : HubPipelineModule{

protected override void OnIncomingError(ExceptionContext exceptionContext,
IHubIncomingInvokerContext invokerContext)      {           Debug.WriteLine("=>
Exception " + exceptionContext.Error.Message);           if
(exceptionContext.Error.InnerException != null)
{           Debug.WriteLine("=> Inner Exception " +
exceptionContext.Error.InnerException.Message);           }
base.OnIncomingError(exceptionContext, invokerContext);

}}}
```

```
public void Configuration(IApplicationBuilder app)
{
// Any connection or hub wire up and configuration should go here
GlobalHost.HubPipeline.AddModule(new ErrorHandlingPipelineModule());
app.MapSignalR();
}
```

使用 **HubException** 类（在 2.0 中引入 SignalR）。可以从任何 Hub 调用会引发此错误。**HubError** 构造函数采用一个字符串消息和一个对象，用于存储额外的错误数据。SignalR 将自动序列化异常，并将其发送给客户端，将使用它来拒绝或失败的 Hub 方法调用。

下面的代码示例演示如何在一个 Hub 调用期间引发的 **HubException** 以及如何处理 JavaScript 和 .NET 客户端上的异常。

演示的 **HubException** 类的服务器代码

```
public class MyHub : Hub{

public void Send(string message)      {

if(message.Contains("<script>"))
```

```

{

    throw new HubException("This message will flow to the client", new { user =
Context.User.Identity.Name, message = message });

}

Clients.All.send(message);

}}

```

演示投掷 **HubException** 在一个 **Hub** 响应的 **JavaScript** 客户端代码

```

myHub.server.send("<script>")
    .fail(function (e) {
        if (e.source === 'HubException') {
            console.log(e.message + ' : ' + e.data.user);
        }
    });

```

演示投掷 **HubException** 在一个 **Hub** 响应的 **.NET** 客户端代码

```

try
{
    await myHub.Invoke("Send", "<script>");
}
catch(HubException ex)
{
    Console.WriteLine(ex.Message);
}

```

11、如何调用客户端方法和管理组从 **Hub** 类外

要从一个不同的类，比您的 Hub 类中调用客户端方法，获取对 Hub SignalR 上下文对象的引用并使用，可以在客户端上调用的方法或管理组。

下面的示例 `StockTicker` 类获取上下文对象、 将其存储在类的实例、 将类实例存储在静态属性，和使用上下文从单一类实例来上到名为 `StockTickerHub` 的 Hub 连接的客户端调用 `updateStockPrice` 方法。

```
// For the complete example, go to
// http://www.asp.net/signalr/overview/getting-started/tutorial-server-broadcast-with-aspnet-si
// This sample only shows code related to getting and using the SignalR context.
public class StockTicker
{
    // Singleton instance
    private readonly static Lazy<StockTicker> _instance = new Lazy<StockTicker>{
        () => new StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>());
    };

    private IHubContext _context;

    private StockTicker(IHubContext context)
    {
        _context = context;
    }

    // This method is invoked by a Timer object.
    private void UpdateStockPrices(object state)
    {
        foreach (var stock in _stocks.Values)
        {
            if (TryUpdateStockPrice(stock))
            {
                _context.Clients.All.updateStockPrice(stock);
            }
        }
    }
}
```

12、如何自定义 Hub 管道

SignalR 使您能够为中心管道注入您自己的代码。下面的示例显示了日志从客户端和传出的方法调用在客户端上收到的每个传入的方法调用的自定义 Hub 管道模块：

```
public class LoggingPipelineModule : HubPipelineModule
{
    protected override bool OnBeforeIncoming(IHubIncomingInvokerContext context)
    {
        Debug.WriteLine("=> Invoking " + context.MethodDescriptor.Name + " on hub " + context.M
        return base.OnBeforeIncoming(context);
    }
    protected override bool OnBeforeOutgoing(IHubOutgoingInvokerContext context)
    {
        Debug.WriteLine("<= Invoking " + context.Invocation.Method + " on client hub " + contex
        return base.OnBeforeOutgoing(context);
    }
}
```

下面的代码在 Startup.cs 文件中注册要在中心管道中运行的模块：

```
public void Configuration(IApplicationBuilder app)
{
    GlobalHost.HubPipeline.AddModule(new LoggingPipelineModule());
    app.MapSignalR();
}
```

第四节、Hub API 教程 - JavaScript 客户端

1、生成的代理，它为您做什么

例如，假设您有下面的 Hub 类服务器上：

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addContosoChatMessageToPage(name, message);
    }
}
```

下面的代码示例显示什么 JavaScript 代码看起来像调用 `NewContosoChatMessage` 方法在服务器上的和接收来自服务器的 `addContosoChatMessageToPage` 方法的调用。

与生成的代理

```

var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addContosoChatMessageToPage = function (name, message) {
    console.log(name + ' ' + message);
};
$.connection.hub.start().done(function () {
    // Wire up Send button to call NewContosoChatMessage on the server.
    $('#newContosoChatMessage').click(function () {
        contosoChatHubProxy.server.newContosoChatMessage($('#displayname').val(), $('#message')
        $('#message').val()).focus();
    });
});

```

没有生成的代理

```

var connection = $.hubConnection();
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
contosoChatHubProxy.on('addContosoChatMessageToPage', function(name, message) {
    console.log(name + ' ' + message);
});
connection.start().done(function() {
    // Wire up Send button to call NewContosoChatMessage on the server.
    $('#newContosoChatMessage').click(function () {
        contosoChatHubProxy.invoke('newContosoChatMessage', $('#displayname').val(), $('#message')
        $('#message').val()).focus();
    });
});

```

何时使用生成的代理

如果您想要注册，服务器将调用客户端方法的多个事件处理程序，则无法使用生成的代理。否则您可以选择使用生成的代理或不基于您编码的偏好。如果您选择不使用它，您不需要引用中的客户端代码的 `script` 元素中的“signalr/Hub”URL。

2、客户端安装程序

JavaScript 客户端需要对 jQuery 和 SignalR 核心 JavaScript 文件的引用。jQuery 版本必须为 1.6.4 或主要更新的版本，例如 1.7.2、1.8.2 或 1.9.1。如果您决定使用生成的代理，

您还需要对生成的 SignalR 代理 JavaScript 文件的引用。下面的示例演示什么引用可能看起来像在 HTML 页中，使用生成的代理。

```
<script src="Scripts/jquery-1.10.2.min.js"></script>
<script src="Scripts/jquery.signalR-2.0.0.min.js"></script>
<script src="signalr/hubs"></script>
```

3、如何建立连接

您可以在建立连接之前，您必须创建连接对象，创建的代理，并注册事件处理程序可以从服务器调用的方法。设置代理服务器和事件处理程序，当通过调用 `start` 方法建立连接。

(与生成的代理) 建立连接

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addContosoChatMessageToPage = function (name, message) {
    console.log(userName + ' ' + message);
};
$.connection.hub.start()
    .done(function(){ console.log('Now connected, connection ID=' + $.connection.hub.id); })
    .fail(function(){ console.log('Could not Connect!'); });
});
```

建立一个连接 (没有在生成的代理)

```
var connection = $.hubConnection();
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
contosoChatHubProxy.on('addContosoChatMessageToPage', function(userName, message) {
    console.log(userName + ' ' + message);
});
connection.start()
    .done(function(){ console.log('Now connected, connection ID=' + connection.id); })
    .fail(function(){ console.log('Could not connect'); });
```

`$.connection.hub` 是相同的 `$.hubConnection()` 创建的对象

4、如何配置连接

您建立连接之前，您可以指定查询字符串参数或指定的运输方法。

如何指定查询字符串参数

如果您想要将数据发送到服务器，当客户端连接时，您可以对连接对象添加查询字符串参数。下面的示例演示如何在客户端代码中设置一个查询字符串参数。

在（与生成的代理）调用 **start** 方法之前设置的查询字符串值

```
$.connection.hub.qs = { 'version' : '1.0' };
```

在调用 **start** 方法（不生成的代理）之前设置的查询字符串值

```
var connection = $.hubConnection();  
  
connection.qs = { 'version' : '1.0' };
```

下面的示例演示如何读取在服务器代码中的查询字符串参数

```
public class ContosoChatHub : Hub  
{  
    public override Task OnConnected()  
    {  
        var version = Context.QueryString['version'];  
        if (version != '1.0')  
        {  
            Clients.Caller.notifyWrongVersion();  
        }  
        return base.OnConnected();  
    }  
}
```

如何指定传输方法

指定的传输方法（与生成的代理）的客户端代码

```
$.connection.hub.start( { transport: 'longPolling' } );
```

```
$.connection.hub.start( { transport: ['webSockets', 'longPolling'] } );
```

指定的传输方法（不生成代理）的客户端代码

```
var connection = $.hubConnection();
connection.start({ transport: 'longPolling' });
```

```
var connection = $.hubConnection();
connection.start({ transport: ['webSockets', 'longPolling'] });
```

为指定的传输方法，可以使用下列值：

- "webSockets"
- "foreverFrame"
- "serverSentEvents"
- "longPolling"

显示（与生成的代理）的连接所使用的传输方法的客户端代码

```
$.connection.hub.start().done(function () {
    console.log("Connected, transport = " + $.connection.hub.transport.name);
});
```

显示（不生成代理）连接所使用的传输方法的客户端代码

```
var connection = $.hubConnection();
connection.hub.start().done(function () {
    console.log("Connected, transport = " + connection.transport.name);
});
```

5、如何在客户端，可以调用服务器上定义的方法

若要定义一个服务器可以从一个 Hub 中调用的方法，通过使用 `client` 属性，生成的代理，将事件处理程序添加到 Hub 代理或调用 `on` 方法，如果你不使用生成的代理。参数可以是复杂的

对象。方法的名称匹配是不区分大小写。例如， `Clients.All.addContosoChatMessageToPage` 在服务器上将在客户端上执行 `AddContosoChatMessageToPage`、`addContosoChatMessageToPage` 或 `addcontosochatmessagetopage`。

(与生成的代理) 的客户端上定义的方法

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addContosoChatMessageToPage = function (userName, message) {
    console.log(userName + ' ' + message);
};
$.connection.hub.start()
    .done(function(){ console.log('Now connected, connection ID=' + $.connection.hub.id); })
    .fail(function(){ console.log('Could not Connect!'); });
});
```

另一种方式 (与生成的代理) 的客户端上定义的方法

```
$.extend(contosoChatHubProxy.client, {
    addContosoChatMessageToPage: function(userName, message) {
        console.log(userName + ' ' + message);
    };
});
```

客户端 (无生成的代理, 或在调用 `start` 方法后添加时) 上定义的方法

```
var connection = $.hubConnection();
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
contosoChatHubProxy.on('addContosoChatMessageToPage', function(userName, message) {
    console.log(userName + ' ' + message);
});
connection.start()
    .done(function(){ console.log('Now connected, connection ID=' + connection.id); })
    .fail(function(){ console.log('Could not connect'); });
```

服务器代码, 调用客户端方法

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addContosoChatMessageToPage(name, message);
    }
}
```

下面的示例包括一个复杂对象作为方法的参数。

需要一个复杂的对象 (与生成的代理) 的客户端上定义的方法


```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addMessageToPage = function (message) {
    console.log(message.UserName + ' ' + message.Message);
};
```

定义的复杂对象的服务器代码

```
public class ContosoChatMessage{

    public string UserName { get; set; }

    public string Message { get; set; }

}
```

调用客户端方法使用一个复杂对象的服务器代码

```
public void SendMessage(string name, string message){

    Clients.All.addContosoChatMessageToPage(

new ContosoChatMessage() { UserName = name, Message = message });

}
```