


微信公众号：大内老A

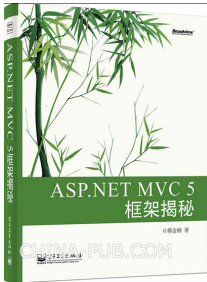


蒋金楠，网名Artech，知名IT博主，微软多领域MVP，畅销IT图书作者，著《WCF全面解析》、《ASP.NET MVC 4/5框架揭秘》、《ASP.NET Web API 2框架揭秘》等。请扫描此二维码关注Artech个人公众帐号，你将会得到及时的文章推送信息。）。

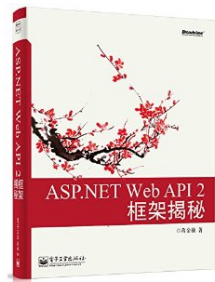
ASP.NET MVC 5 完全攻略[《ASP.NET MVC 5 框架揭秘》繁体版]



ASP.NET MVC 5 框架揭秘



ASP.NET Web API 2 框架揭秘



.NET Core下的日志（1）：记录日志信息

记录各种级别的日志是所有应用不可或缺的功能。关于日志记录的实现，我们有太多第三方框架可供选择，比如Log4Net、NLog、Loggr和Serilog等，当然我们还可以选择微软原生的诊断机制（相关API定义在命名空间“System.Diagnostics”中）实现对日志的记录。 .NET Core提供了独立的日志模型使我们可以采用统一的API来完成针对日志记录的编程，我们同时也可以利用其扩展点对这个模型进行定制，比如可以将上述这些成熟的日志框架整合到我们的应用中。本系列文章旨在从设计和实现的角度对.NET Core提供的日志模型进行深入剖析，不过在这之前我们必须对由它提供的日志记录编程模式具有一个大体的认识，接下来我们会采用实例的形式来演示如何相应等级的日志并最终将其写入到我们期望的目的地中。

目录

一、日志模型三要素

二、将日志写入不同的目的地

三、依赖注入

四、根据等级过滤日志消息

五、利用TraceSource记录日志

直接利用TraceSource记录追踪日志

利用TraceSourceLogger记录追踪日志

打赏

一、日志模型三要素

日志记录编程主要会涉及到三个核心对象，它们分别是Logger、LoggerFactory和LoggerProvider，这三个对象同时也是.NET Core日志模型中的核心对象，并通过相应的接口（ILogger、ILoggerFactory和ILoggerProvider）来体现。右图所示的UML揭示了日志模型的这三个核心对象之间的关系。



在进行日志记录编程时，我们直接调用Logger对象相应的方法写入日志，LoggerFactory是创建Logger对象的工厂。由LoggerFactory创建的Logger并不真正实现对日志的写入操作，真正将日志写入相应目的地的Logger是通过相应的LoggerProvider提供的，前者是对后者的封装，它将日志记录请求委托给后者来完成。

具体来说，在通过LoggerFactory创建Logger之前，我们会根据需求将一个或者多个LoggerProvider注册到LoggerFactory之上。比如，如果我们需要将日志记录到EventLog中，我们会注册一个EventLogLoggerProvider，后者会提供一个EventLogLogger对象来实现针对EventLog的日志记录。当我们利用LoggerFactory创建Logger对象时，它会利用注册其上的所有LoggerProvider创建一组具有真正日志写入功能的Logger对象，并采用“组合（Composition）”模式利用这个Logger列表创建并返回一个Logger对象。

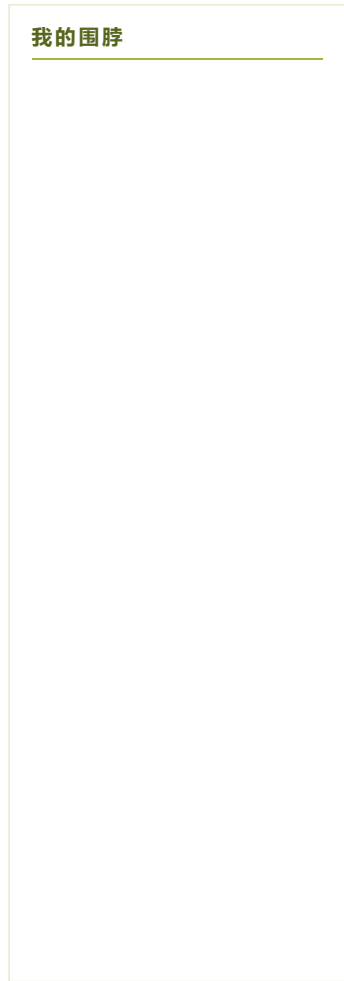
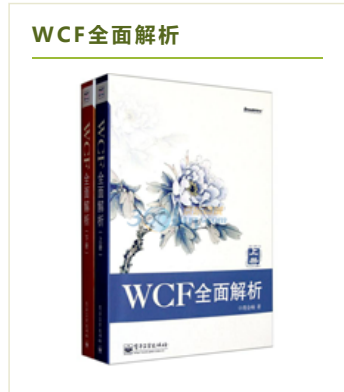
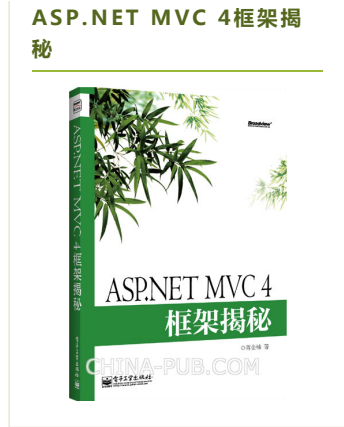
综上所述，LoggerFactory创建的Logger仅仅是一个“壳”，在它内部封装了一个或者多个具有真正日志写入功能的Logger对象。当我们调用前者实施日志记录操作时，它会遍历被封装的Logger对象列表，并委托它们将日志写入到相应的目的地。

二、将日志写入不同的目的地

接下来我们通过一个简单的实例来演示如何将具有不同等级的日志写入两种不同的目的地，其中一种是直接将格式化的日志消息输出到当前控制台，另一种则是将日志写入Debug输出窗口（相当于直接调用Debug.WriteLine方法），针对这两种日志目的地的Logger分别通过ConsoleLoggerProvider和DebugLoggerProvider来提供。

我们创建一个空的项目，并在其project.json文件中添加如下三个NuGet包的依赖，其中默认使用的LoggerFactory和Microsoft.Extensions.Logging之中，而上述的ConsoleLoggerProvider和DebugLoggerProvider则通过NuGet包来提供。由于在默认情况下，.NET Core并不支持中文编码，我们需要显式注册一个名为的针对相应的EncodingProvider，后者定义在NuGet包“System.Text.Encoding.CodePages”之中，所以我们需要添加这个NuGet包的依赖。

```
1: {
2:
3:   "dependencies": {
4:     ...
5:     "Microsoft.Extensions.Logging" : "1.0.0-rc2-final",
6:     "Microsoft.Extensions.Logging.Console" : "1.0.0-rc2-final",
7:     "Microsoft.Extensions.Logging.Debug" : "1.0.0-rc2-final",
```



- 推荐博文系列
- [01] WCF技术剖析

[02] WCF后续之旅

[03] WCF之旅

[04] WCF之绑定模型

[05] 谈谈分布式事务

.NET Core下的日志（1）：记录日志信息 - Artech - 博客园

```
8:
9:     "System.Text.Encoding.CodePages"           : "4.0.1-rc2-24027"
10:  },
11:  ...
12: }
```

我们在入口的Main方法中编写如下一段程序。我们首先创建一个LoggerFactory对象，并先后通过调用AddProvider方法在它上面注册一个ConsoleLoggerProvider对象和DebugLoggerProvider对象。创建它们调用的构造函数具有一个Func<string, LogLevel, bool>类型的参数旨在对日志消息进行写入前过滤（针对日志类型和等级），由于我们传入的委托对象总是返回True，意味着提供的所有日志均会被写入。

```
1: public class Program
2: {
3:     public static void Main(string[] args)
4:     {
5:         //注册EncodingProvider实现对中文编码的支持
6:         Encoding.RegisterProvider(CodePagesEncodingProvider.Instance);
7:
8:         Func<string, LogLevel, bool> filter = (category, level) => true;
9:
10:        ILoggerFactory loggerFactory = new LoggerFactory();
11:        loggerFactory.AddProvider(new ConsoleLoggerProvider(filter, false));
12:        loggerFactory.AddProvider(new DebugLoggerProvider(filter));
13:        ILogger logger = loggerFactory.CreateLogger("App");
14:
15:        int eventId = 3721;
16:
17:        logger.LogInformation(eventId, "升级到最新版本 ({version})", "1.0.0.rc2");
18:        logger.LogWarning(eventId, "并发量接近上限 ({maximum}) ", 200);
19:        logger.LogError(eventId, "数据库连接失败 (数据库: {Database}, 用户名: {User})", "TestDb",
"sa");
20:
21:        Console.Read();
22:    }
23: }
```

我们通过指定日志类型（“App”）调用LoggerFactory对象的CreateLogger方法创建一个Logger对象，并先后调用其LogInformation、LogWarning和LogError方法记录三条日志，这三个方法决定了写入日志的等级（Information、Warning和Error）。我们在调用这三个方法的时候指定了一个表示日志记录事件ID的整数（3721），以及具有占位符（“{version}”、“{maximum}”、“{Database}”和“{User}”）的消息模板和替换这些占位符的参数。

由于ConsoleLoggerProvider被事先注册到创建Logger的LoggerFactory上，所以当我们执行这个实例程序之后，三条日志消息会直接按照如下的形式打印到控制台上。我们可以看出格式化的日志消息不仅仅包含我们指定的消息内容，日志的等级、类型和事件ID同样包含其中。

```
1: info: App[3721]
2:     升级到最新版本 (1.0.0.rc2)
3: warn: App[3721]
4:     并发量接近上限 (200)
5: fail: App[3721]
6:     数据库连接失败 (数据库: TestDb, 用户名: sa)
```

由于LoggerFactory上还注册了另一个DebugLoggerProvider对象，由它创建的Logger会直接调用Debug.WriteLine方法写入格式化的日志消息。所以当我们以Debug模式编译并执行该程序时，Visual Studio的输出窗口会以右图所示的形式呈现出格式化的日志消息。



上面这个实例演示了日志记录采用的基本变成模式，即创建/获取LoggerFactory并注册相应的LoggerProvider，然后利用LoggerFactory创建Logger，并最终利用Logger记录日志。LoggerProvider的注册除了可以直接调用LoggerFactory的AddProvider方法来完成之外，对于预定义的LoggerProvider，我们还可以调用相应的扩展方法来将它们注册到指定的LoggerFactory上。比如在如下所示的代码片断中，我们直接调用针对ILoggerFactory接口的扩展方法AddConsole和AddDebug分别注册一个ConsoleLoggerProvider和DebugLoggerProvider。

```
1: ILogger logger = new LoggerFactory()
2:     .AddConsole()
3:     .AddDebug()
4:     .CreateLogger("App");
```

三、依赖注入

在我们演示的实例中，我们直接调用构造函数创建了一个LoggerFactory并利用它来创建用于记录日志的Logger，在一个.NET Core应用中，LoggerFactory会以依赖注入的方式注册到ServiceProvider之中。如果我们需要采用依赖注入的方式来获取注册的LoggerFactory，我们需要在project.json文件中添加针对“Microsoft.Extensions.DependencyInjection”这个NuGet包的依赖。

```
1: {
```

- [06] WCF中并发与限流体系深入解析
- [07] 深入剖析WCF的可靠会话
- [08] EnterLib深入解析与灵活应用
- [09] EnterLib PIAB深度剖析
- [10] 深度剖析C# 3.x新特性
- [11] 与VS集成的若干种代码生成机制
- [12] 深入剖析授权在WCF中的实现
- [13] WCF运行时框架的构建于扩展
- [14] WCF 4.0新特性
- [15] WCF REST系列
- [16] How ASP.NET MVC Works

版权声明

Artech 所有文章遵循创作共用版权协议，要求署名、非商业、保持一致。在满足创作共用版权协议的基础上可以转载，但请以超链接形式注明出处。

我的标签

- WCF(225)
- ASP.NET(81)
- MVC(68)
- ASP.NET Core(55)
- .NET Core(54)
- Security(35)
- IoC(28)
- EnterLib(27)
- Exception Handling(27)
- ASP.NET Web API(25)
- 更多

随笔分类 (716)

- [01] 技术剖析(372)
- [02] 编程技巧(173)
- [03] 设计模式(19)
- [04] 架构思想(25)
- [05] 开源框架(23)
- [06] 软件工程
- [07] 项目管理
- [08] 心情随笔(13)
- [09] 杂家杂谈(4)
- [10] 业界新闻(1)
- [11] 读书心得(1)
- [12] 它山之石(23)
- [13] 著作推广(15)
- [14] 框架设计(41)
- [15] 工具推荐(6)
- ASP.NET Core, IoC, DI

随笔档案 (651)

- 2016年12月 (14)
- 2016年11月 (10)
- 2016年10月 (4)
- 2016年9月 (5)

```
2:  "dependencies": {
3:    ...
4:    "Microsoft.Extensions.DependencyInjection"      : "1.0.0-rc2-final",
5:    "Microsoft.Extensions.Logging"                  : "1.0.0-rc2-final",
6:    "Microsoft.Extensions.Logging.Console"          : "1.0.0-rc2-final",
7:    "Microsoft.Extensions.Logging.Debug"            : "1.0.0-rc2-final",
8:  },
9:  ...
10: }
```

针对LoggerFactory的注册可以通过调用针对IServiceCollection接口的扩展方法AddLogging来完成。当我们调用这个方法的时候，它会创建一个LoggerFactory对象并以Singleton模式注册到指定的ServiceCollection之上。对于我们演示实例中使用的Logger对象，可以利用以依赖注入形式获取的LoggerFactory来创建，如下所示的代码片段体现了这样的编程方式。

```
1: ILogger logger = new ServiceCollection()
2:   .AddLogging()
3:   .BuildServiceProvider()
4:   .GetService<ILoggerFactory>()
5:   .AddConsole()
6:   .AddDebug()
7:   .CreateLogger("App");
```

四、根据等级过滤日志消息

对于通过某个LoggerProvider提供的Logger，它并总是会将提供给它的日志消息写入对应的目的地，它可以根据提供的过滤条件忽略无需写入的日志消息，针对日志等级是我们普遍采用的日志过滤策略。日志等级通过具有如下定义的枚举LogLevel来表示，枚举项的值决定了等级的高低，值越大，等级越高；等级越高，越需要记录。

```
1: public enum LogLevel
2: {
3:     Trace           = 0,
4:     Debug           = 1,
5:     Information     = 2,
6:     Warning         = 3,
7:     Error           = 4,
8:     Critical        = 5,
9:     None            = 6
10: }
```

在前面介绍ConsoleLoggerProvider和DebugLoggerProvider的时候，我们提到可以在调用构造函数时可以传入一个Func<string, LogLevel, bool>类型的参数来指定日志过滤条件。对于我们实例中写入的三条日志，它们的等级由低到高分别是Information、Warning和Error，如果我们选择只写入等级高于或等于Warning的日志，可以采用如下的方式来创建对应的Logger。

```
1: Func<string, LogLevel, bool> filter =
2:   (category, level) => level >= LogLevel.Warning;
3:
4: ILoggerFactory loggerFactory = new LoggerFactory();
5: loggerFactory.AddProvider(new ConsoleLoggerProvider(filter, false));
6: loggerFactory.AddProvider(new DebugLoggerProvider(filter));
7: ILogger logger = loggerFactory.CreateLogger("App");
```

针对ILoggerFactory接口的扩展方法AddConsole和AddDebug同样提供的相应的重载使我们可以通过传入的Func<string, LogLevel, bool>类型的参数来提供日志过滤条件。除此之外，我们还可以直接指定一个类型为LogLevel的参数来指定过滤日志采用的最低等级。我们演示实例中的使用的Logger可以按照如下两种方式来创建。

```
1: ILogger logger = new ServiceCollection()
2:   .AddLogging()
3:   .BuildServiceProvider()
4:   .GetService<ILoggerFactory>()
5:
6:   .AddConsole((c,l)=>l>= LogLevel.Warning)
7:   .AddDebug((c, l) => l >= LogLevel.Warning)
8:   .CreateLogger("App");
```

或者

```
1: ILogger logger = new ServiceCollection()
2:   .AddLogging()
3:   .BuildServiceProvider()
4:   .GetService<ILoggerFactory>()
5:   .AddConsole(LogLevel.Warning)
6:   .AddDebug(LogLevel.Warning)
7:   .CreateLogger("App");
```

由于注册到LoggerFactory上的ConsoleLoggerProvider和DebugLoggerProvider都采用了上述的日志过滤条件，所有由它们提供Logger都只会写入等级为Warning和Error的两条日志，至于等级为Information的那条则会自动忽略掉。所以我们的程序执行之后会在控制台上打印出如下所示的日志消息。

```
1: warn: App[3721]
```

- 2016年8月 (10)
- 2016年7月 (9)
- 2016年6月 (14)
- 2016年5月 (13)
- 2016年4月 (14)
- 2015年9月 (1)
- 2015年7月 (1)
- 2014年12月 (3)
- 2014年8月 (3)
- 2014年7月 (3)
- 2014年4月 (9)
- 2014年3月 (5)
- 2014年1月 (2)
- 2013年12月 (12)
- 2013年11月 (1)
- 2013年8月 (2)
- 2013年7月 (3)
- 2013年4月 (3)
- 2013年2月 (1)
- 2013年1月 (3)
- 2012年12月 (2)
- 2012年11月 (4)
- 2012年10月 (4)
- 2012年9月 (4)
- 2012年8月 (9)
- 2012年7月 (3)
- 2012年6月 (15)
- 2012年5月 (24)
- 2012年4月 (7)
- 2012年3月 (18)
- 2012年2月 (11)
- 2012年1月 (7)
- 2011年12月 (12)
- 2011年11月 (3)
- 2011年10月 (10)
- 2011年9月 (16)
- 2011年8月 (1)
- 2011年7月 (17)
- 2011年6月 (11)
- 2011年5月 (8)
- 2011年4月 (3)
- 2011年3月 (14)
- 2011年1月 (8)
- 2010年12月 (4)
- 2010年11月 (12)
- 2010年10月 (14)
- 2010年9月 (17)
- 2010年8月 (7)
- 2010年7月 (4)
- 2010年6月 (2)
- 2010年5月 (5)
- 2010年4月 (16)
- 2010年3月 (15)
- 2010年2月 (2)
- 2010年1月 (8)
- 2009年12月 (18)
- 2009年11月 (15)
- 2009年10月 (5)
- 2009年8月 (6)

```
2:         并发量接近上限 (200)
3: fail: App[3721]
4:         数据库连接失败 (数据库: TestDb, 用户名: sa)
```

五、利用TraceSource记录日志

从微软推出第一个版本的.NET Framework的时候，就在“System.Diagnostics”命名空间中提供了Debug和Trace两个类帮助我们完成针对调试和追踪信息的日志记录。在.NET Framework 2.0种，增强的追踪日志功能实现在新引入的TraceSource类型中，并成为我们的首选。.NET Core的日志模型借助TraceSourceLoggerProvider实现对TraceSource的整合。

直接利用TraceSource记录追踪日志

.NET Core 中的TraceSource以及相关类型定义在NuGet包“System.Diagnostics.TraceSource”，如果我们需要直接使用TraceSource来记录日志，应用所在的Project.json文件中需要按照如下的方式添加针对这个NuGet包的依赖。

```
1: {
2:   "dependencies": {
3:     ...
4:     "System.Diagnostics.TraceSource": "4.0.0-rc2-24027"
5:   },
6: }
```

不论采用Debug和Trace还是TraceSource，追踪日志最终都是通过注册的TraceListener被写入相应的目的地。在“System.Diagnostics”命名空间中提供了若干预定义的TraceListener，我们也可以自由地创建自定义的TraceListener。如下面的代码片断所示，我们通过继承抽象基类TraceListener自定义了一个ConsoleTranceListener类，它通过重写的Write和WriteLine方法将格式化的追踪消息输出到当前控制台。

```
1: public class ConsoleTraceListener : TraceListener
2: {
3:     public override void Write(string message)
4:     {
5:         Console.Write(message);
6:     }
7:
8:     public override void WriteLine(string message)
9:     {
10:        Console.WriteLine(message);
11:    }
12: }
```

我们可以直接利用TraceSource记录上面实例演示的三条日志。如下面的代码片断所示，我们通过指定名称（“App”）创建了一个TraceSource对象，然后在它的TraceListener列表中注册了一个ConsoleTraceListener对象。我们为此TraceSource指定了一个开关（一个SourceSwitch对象）让它仅仅记录等级高于Warning的追踪日志。我们调用TraceSource的TraceEvent方法实现针对不同等级（Information、Warning和Error）的三条追踪日志的记录。

```
1: public class Program
2: {
3:     public static void Main(string[] args)
4:     {
5:         TraceSource traceSource = new TraceSource("App");
6:         traceSource.Listeners.Add(new ConsoleTraceListener());
7:         traceSource.Switch = new SourceSwitch("LogWarningOrAbove", "Warning");
8:
9:         int eventId = 3721;
10:        traceSource.TraceEvent(TraceEventType.Information, eventId, "升级到最新版本({0})",
"1.0.0.rc2");
11:        traceSource.TraceEvent(TraceEventType.Warning, eventId, "并发量接近上限({0}) ", 200);
12:        traceSource.TraceEvent(TraceEventType.Error, eventId, "数据库连接失败 (数据库: {0}, 用户名: {1})", "TestDb", "sa");
13:    }
14: }
```

当我们执行该程序之后，满足TraceSource过滤条件的两条追踪日志（即等级分别为Warning和Error的两条追踪日志）将会通过注册的ConsoleTraceListner写入当前控制台，具体的内容如下所示。由于一个DefaultTraceListener对象会自动注册到TraceSource之上，在它的Write或者WriteLine方法中会调用Win32函数OutputDebugString或者Debugger.Log方法，所以如果我们采用Debug模式编译我们的程序，当程序运行后会在Visual Studio的输出窗口中看到这两条日志消息。

```
1: App Warning: 3721 : 并发量接近上限 (200)
2: App Error: 3721 : 数据库连接失败 (数据库: TestDb, 用户名: sa)
```

利用TraceSourceLoggerProvider记录追踪日志

NET Core的日志模型借助TraceSourceLoggerProvider实现对TraceSource的整合。具体来说，由于TraceSourceLoggerProvider提供的Logger对象实际上是对一个TraceSource的封装，对于提供给Logger的日志消息，后者会借助注册到TraceSource上面的TraceListener来完成对日志消息的写入工作。由于TraceSourceLoggerProvider定义在NuGet包“Microsoft.Extensions.Logging.TraceSource”，我们需要按照如下的方式将针对它的依赖定义在project.json中。

2009年7月 (18)
2009年6月 (9)
2009年5月 (3)
2009年4月 (4)
2008年12月 (3)
2008年11月 (5)
2008年10月 (2)
2008年9月 (8)
2008年8月 (11)
2008年7月 (9)
2008年2月 (2)
2008年1月 (2)
2007年12月 (3)
2007年11月 (4)
2007年10月 (1)
2007年9月 (6)
2007年8月 (4)
2007年7月 (9)
2007年6月 (6)
2007年5月 (9)
2007年4月 (9)
2007年3月 (16)
2007年2月 (2)

积分与排名

积分 - 3035072

排名 - 9

最新评论

1. Re:ASP.NET Core框架揭秘（持续更新中...）

@浮云也是种寂寞引用引用引用值得一提的是，这些文章将在个人公众帐号（Artech1984，大内老A）上发布，如果你希望采用这种阅读方式，或者希望得到及时的推送提醒，可以关注扫描左上方二维码关注。A大啊.....

--Artech

2. Re:ASP.NET Core框架揭秘（持续更新中...）

引用值得一提的是，这些文章将在个人公众帐号（Artech1984，大内老A）上发布，如果你希望采用这种阅读方式，或者希望得到及时的推送提醒，可以关注扫描左上方二维码关注。A大啊我关注你很久了哎 咋.....

--浮云也是种寂寞

3. Re:新作《ASP.NET MVC 5 框架揭秘》正式出版

大佬们，都在弄.net core。而我前几天才买这本书

--闲人不闲

4. Re:ASP.NET Core应用的错误处理[1]：三种呈现错误页面的方式

.NET Core下的日志（1）：记录日志信息 - Artech - 博客园

```
1: {
2:   "dependencies": {
3:     "Microsoft.Extensions.DependencyInjection" : "1.0.0-rc2-final",
4:     "Microsoft.Extensions.Logging" : "1.0.0-rc2-final",
5:     "Microsoft.Extensions.Logging.TraceSource" : "1.0.0-rc2-final"
6:   },
7:   ...
8: }
```

如果采用要利用日志模型标准的编程方式来记录日志，我们可以按照如下的方式来创建对应的Logger对象。如下面的代码片断所示，我们创建一个TraceSourceLoggerProvider对象并调用AddProvider方法将其注册到LoggerFactory对象上。创建TraceSourceLoggerProvider的构造函数接受两个参数，前者是一个SourceSwitch对象，用于过滤等级低于Warning的日志消息，后者则是我们自定义的ConsoleTraceListener对象。

```
1: ILoggerFactory loggerFactory = new ServiceCollection()
2:   .AddLogging()
3:   .BuildServiceProvider()
4:   .GetService<ILoggerFactory>();
5:
6: SourceSwitch sourceSwitcher = new SourceSwitch("LogWarningOrAbove", "Warning");
7: loggerFactory.AddProvider(new TraceSourceLoggerProvider(sourceSwitcher, new
ConsoleTraceListener()));
8:
9: ILogger logger = loggerFactory.CreateLogger("App");
```

我们可以调用针对ILoggerFactory的扩展方法AddTraceSource来实现对TraceSourceLoggerProvider的注册，该方法具有与TraceSourceLoggerProvider构造函数相同的参数列表。如下所示的代码片断通过调用这个扩展方法以更加精简的方式创建了日志记录所需的Logger对象。

```
1: ILogger logger = new ServiceCollection()
2:   .AddLogging()
3:   .BuildServiceProvider()
4:   .GetService<ILoggerFactory>()
5:   .AddTraceSource(new SourceSwitch("LogWarningOrAbove", "Warning"), new
ConsoleTraceListener())
6:   .CreateLogger("App");
```

作者：蒋金楠

微信公众号号：大内老A

微博：www.weibo.com/artech

如果你想及时得到个人撰写文章以及著作的消息推送，或者想看看个人推荐的技术资料，可以扫描左边二维码（或者长按识别二维码）关注个人公众号（原来公众帐号蒋金楠的自媒体将会停用）。

本文版权归作者和博客园共有，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接，否则保留追究法律责任的权利。

分类: [01] 技术剖析

好文要顶 关注我 收藏该文

Artech

关注 - 52 粉丝 - 7130

荣誉：推荐博客 +加关注

标签: .NET Core, Logging, Logger, LoggerFactory, LoggerProvider

« 上一篇: ASP.NET Core 1.0中实现文件上传的两种方式（提交表单和采用AJAX）

» 下一篇: .NET Core下的日志（2）：日志模型详解

posted @ 2016-06-03 06:23 Artech 阅读(3068) 评论(6) 编辑 收藏

评论列表

#1楼 2016-06-03 10:00 花儿笑弯了腰

支持 支持

支持(0) 反对(0)

#2楼 2016-06-03 13:16 Areyan

支持 支持

支持(0) 反对(0)

很有帮助~感谢~

--幻天芒

5. Re:ASP.NET Core框架揭秘（持续更新中...）

坐等Asp.Net Core框架揭秘，WCF API MVC对我帮助太大了

--灿若星辰の浩如烟海

6. Re:ASP.NET Core框架揭秘（持续更新中...）

看过蒋老师书籍的明显能发现书中的内容比博客里多 强烈期待新书早点出版啊！

--liamyu

7. Re:ASP.NET Core应用的错误处理[1]：三种呈现错误页面的方式

@幸运草thx！ ...

--Artech

8. Re:ASP.NET Core应用的错误处理[1]：三种呈现错误页面的方式

顶起来

--幸运草

9. Re:ASP.NET Core框架揭秘（持续更新中...）

@renjing引用蒋老师啥时候出.netcore的教程书籍呀？还不确定！ ...

--Artech

10. Re:ASP.NET Core框架揭秘（持续更新中...）

蒋老师啥时候出.netcore的教程书籍呀？

--renjing

推荐排行榜

1. 我的WCF之旅（1）：创建一个简单的WCF程序(1186)

2. 我的WCF之旅（2）： Endpoint Overview(262)

3. 我的WCF之旅（3）： 在WCF中实现双工通信(242)

4. How ASP.NET MVC Works?(233)

5. 我看周马，以及3Q大战背后的社会问题(170)

6. 新作《ASP.NET MVC 4框架揭秘》正式出版(153)

7. ASP.NET Core中的依赖注入（1）：控制反转（IoC）(146)

8. 《我的WCF之旅》博文系列汇总(141)

9. WCF技术剖析之二：再谈IIS与ASP.NET管道(138)

10. 关于CLR内存管理一些深层次的讨论[上篇](121)

11. ASP.NET MVC下的四种验证编程方式(112)

#3楼 2016-06-04 13:35 我有一颗四叶草

Hi 能否出个.NET CORE 1.0 如何做到前后端分离。前端只有web,后端提供restful api调用。Thanks, 目前我们有个电商网站我们是直接在views的cshtml里面写C#代码调用service的方法获取数据。感觉这种方式不好。而且还是同步的。Thanks

支持(0) 反对(0)

#4楼 2016-06-04 18:41 快乐的微笑

支持,最近一直关注.net core.

支持(0) 反对(0)

#5楼 2016-06-04 18:47 gleox

发现2个typo:

* 针对日子类型和等级 -> 针对日志类型和等级？

* 日志记录采用的基本变成模式 -> 日志记录采用的基本编程模式？

支持(0) 反对(0)

#6楼 2016-06-12 09:03 一树繁花

正准备要用呢

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问](#)网站首页。

最新IT新闻：

- 使用Actions on Google和API.AI构建会话式应用
- 微信开源软件列表
- 戴尔全球最薄27英寸显示器出镜：颜值无死角
- 犯上大企业病的零度无人机错在哪？
- 日本保险巨头启用AI替换30%理赔部员工

» 更多新闻...

最新知识库文章：

- 写给未来的程序媛
- 高质量的工程代码为什么难写
- 循序渐进地代码重构
- 技术的正宗与野路子
- 陈皓：什么是工程师文化？

» 更多知识库文章...

历史上的今天：
2012-06-03 ASP.NET MVC以ModelValidator为核心的Model验证体系：ModelValidatorProviders

12. 通过一个模拟程序让你明白ASP.NET MVC是如何运行的(111)
13. 我的WCF之旅（4）：WCF中的序列化[上篇](106)
14. 新作《ASP.NET MVC 5框架揭秘》正式出版(105)
15. WCF技术剖析之一：通过一个ASP.NET程序模拟WCF基础架构(101)
16. 从数据到代码——基于T4的代码生成方式(96)
17. 《WCF全面解析》（上、下册）正式出版(96)
18. 与VS集成的若干种代码生成解决方案[博文汇总(共8篇)](86)
19. 感恩回馈，《ASP.NET Web API 2框架揭秘》免费赠送(85)
20. C# 4.0新特性-"协变"与"逆变"以及背后的编程思想(82)
21. 在一个空ASP.NET Web项目上创建一个ASP.NET Web API 2.0应用(77)
22. 通过几个Hello World感受.NET Core全新的开发体验(73)
23. IoC+AOP的简单实现(72)
24. 如何编写没有Try/Catch的程序(71)
25. 我所理解的RESTful Web API [设计篇](71)
26. MVVM(Knockout.js)的新尝试：多个Page，一个ViewModel(71)
27. ASP.NET的路由系统：URL与物理文件的分离(71)
28. 如果没有Visual Studio 2015，我们如何创建.NET Core项目？(70)
29. 谈谈你最熟悉的System.DateTime[上篇](69)
30. 唐伯虎的垃圾(68)
31. ASP.NET MVC是如何运行的[1]：建立在"伪"MVC框架上的Web应用(66)
32. ASP.NET的路由系统：路由映射(64)
33. 谈谈IE针对Ajax请求结果的缓存(64)
34. 创建代码生成器可以很简单：如何通过T4模板生成代码？[上篇](64)
35. 你知道Unity IoC Container是如何创建对象的吗？(64)
36. 《WCF技术剖析》博文系列汇总[持续更新中](64)
37. ASP.NET MVC是如何运行的[2]：URL路由(63)
38. 我的WCF之旅（8）：WCF中的Session和Instancing Management(62)
39. ASP.NET Process Model之一：IIS 和 ASP.NET ISAPI(60)
40. 追踪记录每笔业务操作数据改变

的利器——SQLCDC(59)

Copyright ©2017 Artech