

CentoS dotNET 环境 部署

Geffzhang（张善友）

作者介绍.....	4
前言.....	5
部署 .NET Core Framework.....	7
部署 Mono 4 环境.....	16
源码安装.....	16
软件包安装.....	18
绿色环境 jws.mono.....	21
在线安装 JWS.Mono.....	21
jwsd 服务.....	22
激活 jws.mono 的图像处理.....	22
OWIN 服务器.....	23
OWIN 规范.....	23
微软 Katana 服务器.....	26
Jexus TinyFox.....	28
TinyFox 介绍.....	28
TinyFox 不依赖于 Mono “独立运行”.....	30
使用 Docker 部署.....	30
Docker 部署 ASP.NET 5 应用.....	30
创建运行环境.....	30
为你的 ASP.NET 5 应用创建一个 Docker 镜像.....	30
创建镜像.....	32
运行容器.....	32
Docker 部署 Mono.....	34
Web 服务器 Jexus.....	34
Jexus 是什么.....	34
安装 Jexus.....	36
将 Jexus 安装为系统服务.....	37
Jexus 的配置.....	39
Jexus 服务器配置.....	39
Jexus 网站配置.....	42
Jexus 的 NOFile 功能.....	46
Jexus 的 URL 重写.....	47
IP 访问控制.....	47
Jexus 反向代理.....	48
启用 HTTPS 进行 SSL 安全传输.....	49
安全保护策略.....	50
让 Jexus 支持高并发请求的优化技巧.....	51
提示 server busy.....	54
配置 PHP.....	54
PHP-FCGI 服务支持 PHP.....	54
PHP-FPM 服务支持 PHP.....	55
.NET(Phalanger)支持 PHP.....	57
Phalanger 简介.....	57
Phalanger 的组件.....	57

Jexus 下运行 Phalanger.....	57
配置 Perl.....	58
Ngnix+ fastcgi_mono.....	59
通过 yum 安装 Nginx:	59
配置 Nginx 和 fastcgi.....	59
启动 Nginx、fastcgi_server.....	59
测试是否能够执行 aspx.....	60
Apache +Mod_mono.....	60
安装 Apache 和 mod_mono 模块.....	61
配置 apache 和 mod_mono.....	61
测试是否能够执行 aspx.....	63
Mono 服务.....	63
mono-service 运行 windows 服务.....	64
使用 Topshelf 创建 Windows 服务.....	66
开发工具.....	72
Visual Studio 2015.....	72
Xamarin Studio.....	72
Visual Studio Code.....	77
Omnisharp.....	89
ASP.NET Linux 部署.....	91

作者介绍

张善友 2001 年开始他的职业生涯，他一直是一个微软技术的开发者，连续荣获 10 年的 ASP.NET MVP，热衷于开源，在社区积极推广开源技术 Mono。

张善友拥有 SUSE Linux 企业服务器, CentOS 以及 tLinux(腾讯自行研制的 Linux 发行版) 的专业经验，他主要是在 CentOS 上部署 Mono 平台，在业余时间喜欢教别人如何使用和利用 Linux 操作系统的力量，特别针对 Windows 开发人员收集编写了这本 Linux 简要。希望对 Windows 上的 .NET 开发人员顺利跨入 Linux 的 Mono 平台开发提供帮助。

业余时间运营微信公众号 dotNET 跨平台，微信号 opendotnet，欢迎关注。



这本书中资料来源于微信公众号 opendotnet、
<http://www.cnblogs.com/shanyou/archive/2012/07/28/2612919.html> 的文章
汇集和 <http://www.linuxdot.net> 的资料整理，以及在 QQ 群：102732979、
103810355 里群友的讨论，在这里为国内积极贡献 dotNET 跨平台实践的同仁表示感谢。

前言

传统.NET 框架

传统.NET 框架是这三个版本中特性最丰富也最完善的，Windows 操作系统就自带.NET 框架。围绕.NET 框架的生态系统也非常成熟所以.NET 框架具有极高的可用性和兼容性。

除了以上的优点，.NET 框架只能运行于 Windows 操作系统当中，而且体积较为庞大所以版本迭代速度较慢。现在我们虽然可以看到.NET 框架的源代码但它并不完全是传统意义上的开源项目。

Mono

Mono 是将.NET 框架迁移到非 Windows 平台所做出的努力。Mono 是一个开源项目，它也有一些类似的模块化理念，所以很多库已经支持 Mono。Mono 并不是由微软公司维护的但在.NET 核心框架逐步成熟的过程中它也起到的对跨平台应用开发进行探索的作用。

.NET 核心框架

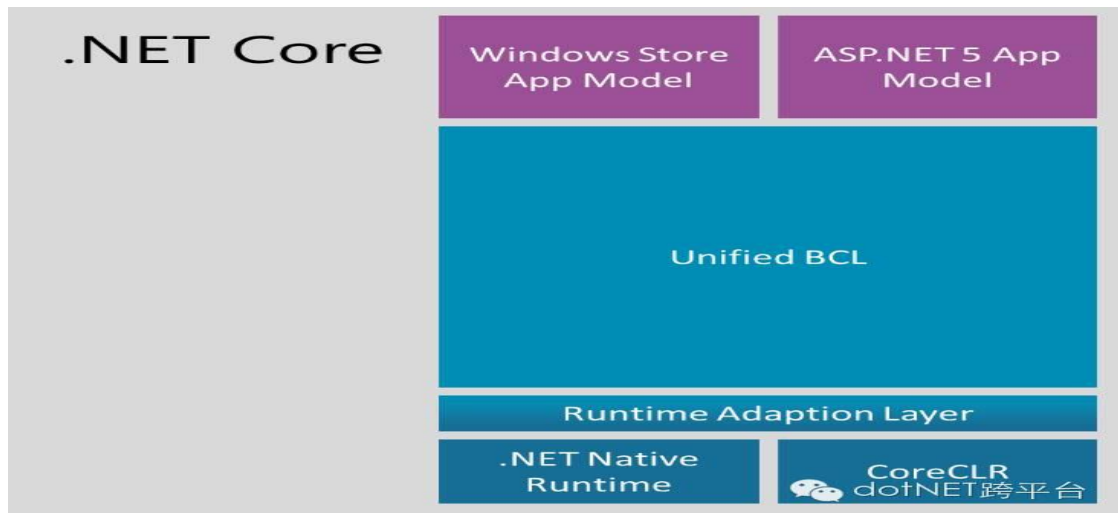
.NET Core 5 包含模块化的运行环境和库，但它并不包括.NET 框架的所有特性。Windows 操作系统中的.NET 核心框架所有的特性都已经开发完成而对于 Linux 和 OS X 还仍然在开发迭代中。.NET 核心框架包括名为 CoreFX 的库以及一个轻量级的运行环境 CoreCLR。.NET 核心框架是完全开源的，您可以在 GitHub 查看项目的进展情况。

我们通过 NuGet 来发布 CoreCLR 运行环境和 CoreFX 库。因为已经进行了模块化的分割，在开发应用时您只需加载所用到的特性相关的库。.NET 核心框架也可以运行于更轻量级的环境当中（比如 Windows Server Nano）。

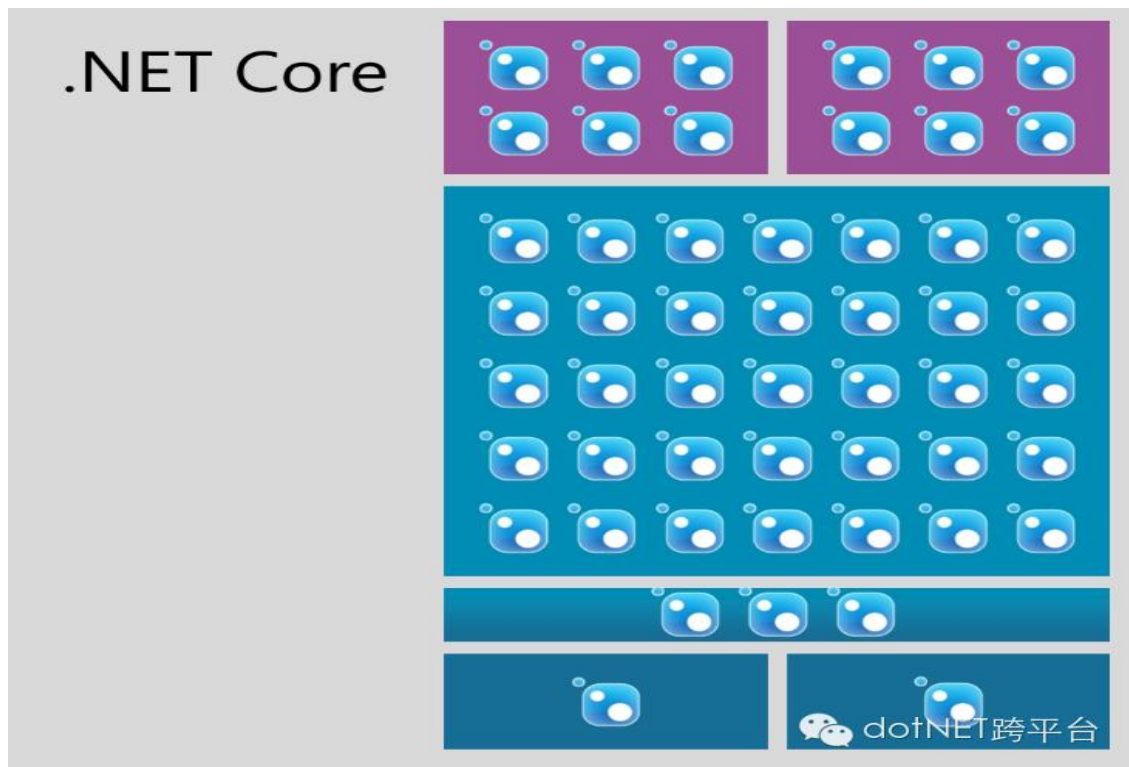
.NET 核心框架与传统.NET 框架所提供的 API 有所不同，所以基本上现有的针对.NET 框架所开发的程序都需要重新编译才能在.NET 核心框架上运行。相对而言.NET 核心框架还非常新，随着流行的.NET 库如 JSON.NET, AutoFac, xUnit.net 等加入对.NET 核心框架的支持，它正在快速的发展中。

针对.NET 核心框架进行开发允许开发人员只开发一次应用程序就运行在不同的平台上，但现在对于 Mac 和 Linux 系统的支持还不完全成熟，所以要用在生产环境中恐怕还有些勉强。

在 Linux 下运行.NET 环境现在有 2 个运行时环境 Mono 4 和 .NET Core Framework 5。.NET Core Framework 是针对云计算环境重新设计的.NET CLR，没有传统的.NET Framework 所要支持全方位编程环境（要能够符合 Desktop、WPF、Windows App、ASP.NET 等等）。.NET Core FX 简化了 Windows App 以及 ASP.NET 的 App 模型，并且基于此重新实现了与平台无关的基础类库（BCL - Base ClassLibrary），示意图像这样：



而在这个架构下，所有东西都被切成模块化，而这些模块都可以通过 NuGet 来各自更新或更换：



现有的项目无法这么轻易转换到新的架构下，还是可以继续在 full-stack 的 .NET Framework 上来建构各种应用程序，包括 ASP.NET 4.6 等等，在这条路线下，还是能够用一切熟悉的架构、工具来运行应用程序，这个对应的 Windows 版本是 .NET Framework 4.6, 那么 Linux 就是 Mono 4.2 版本了，Mono 4 版本整合了微软开源的代码，不仅是整合了 .NET CoreCLR 的开源版本代码，其默认语言是 C# 6, 还整合了微软的参考开源代码，Mono 还具备了很多 CoreCLR 没有的功能：LLVM、完整的提前编译（AOT）、原生客户端（NaCl）、微进程（tasklet）、跨虚拟机的垃圾回收（GC）桥接（bridge）、各种探查器（profile）模块等等。

本书简要介绍 Linux 上部署 .NET Core Framework 和 Mono 4 和 Web 服务器 Jexus 。

部署.NET Core Framework

.NET 运行环境(DNX)可以安装在 Linux 等多种平台上,在 Linux 系统中使用 Mono 安装 DNX。和 ASP.NET 5 的方法,安装 ASP.NET 5 建议使用 Mono 4.0.1 之后的版本。

ASP.NET 5 引入了一个新型的运行时,让我们可以现场交付模式组合式构建应用程序,而不依赖于宿主机上的.NET 框架。这种新模式为我们提供了命令行工具(DNVM、DNX、DNU)用于管理我们的.net 版本,依赖的库和运行环境,我们可以不需要 Visual Studio,只需要一个文本编辑器和命令行就可以开发一个应用程序。

了解.NET 版本管理器(DNVM)之间,.NET 执行环境(DNX)和.NET 开发实用程序(DNU)之间的关系是开发 ASP.NET 5 的根本。在这篇文章我们将看看在 CentOS 安装并使用 DNVM, DNX 和 DNU,从命令行和文本编辑器开发一个简单应用程序,如果你使用 docker,这些命令和 docker 还真的很像。

DNVM(.NET Version Manager): 由于要实现跨平台的目录,微软提供了 DNVM 功能, DNVM 是 ASP.NET 最底层的内容,它是一组 Powershell 脚本,用于启动指定版本的 ASP.NET 运行环境,并且可以在同一台机器的同一时间点上通过使用 Nuget 工具来管理各种版本的 ASP.NET 运行环境(DNX),以及进行相应的升级操作。

DNX(.NET Execution Environment): DNX 是 ASP.NET 程序的运行环境,用于启动并运行 ASP.NET 程序。该运行环境包括了编译系统、SDK 工具集、Native CLR 宿主环境。可以使用 DNVM 管理各种版本的 DNX,如 `dnvm list` 命令可以列出所有可用的 DNX 环境,而 `dnvm install 1.0.0-beta4` 则可以将指定版本的 DNX 安装到.dnx 文件夹,你可以在%USERPROFILE%\ .dnx\runtimes 目录下找到已安装所有版本的 DNX。不同的操作系统有不同的 DNX 版本。

dnx.exe: dnx.exe 是用于启动自宿主环境(Self-Hosting)的命令行工具,在使用命令行代码进行自宿主环境启动程序时, dnx 负责查找并调用 CLR Native Host, dnx 命令是整个运行环境的入口点,你可以使用 `dnx run` 来启动程序。

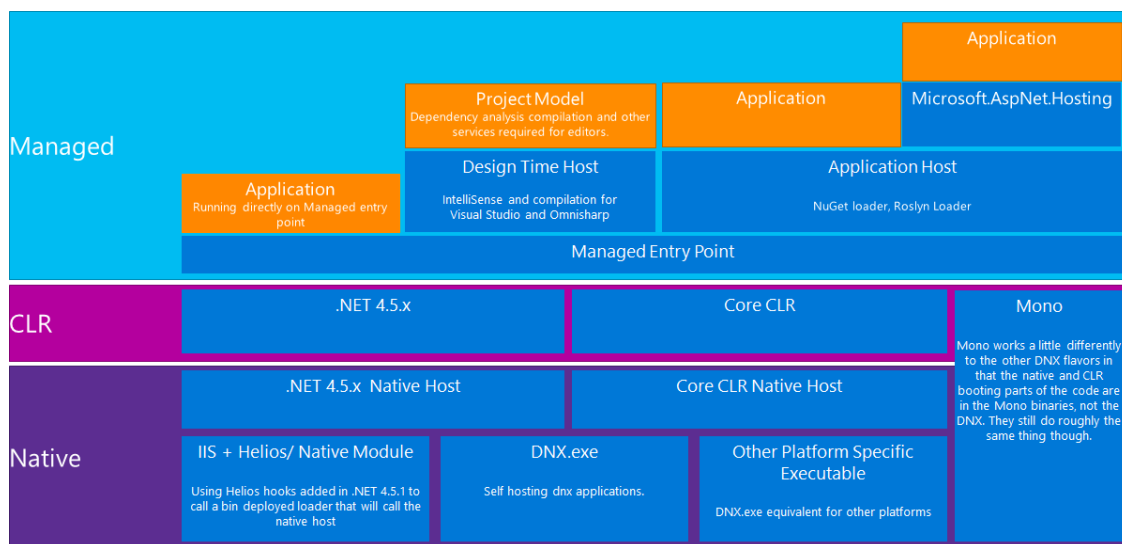
dnu(DNX Utility): 是一个命令行的包管理器,包含在 DNX 内,所以只要安装了 DNX,就可以使用 dnu 命令,其可以用于恢复程序包、安装程序包、部署程序包等等,比如把 project.json 里自定义的程序集自动下载下来进行使用。

DNX 架构及运行原理

DNX 是 ASP.NET 程序运行的核心,其遵循如下两个准则:

1. DNX 应该是自包含的，DNX 在解析完应用程序依赖树以后才能知道要使用哪个 Core CLR 包，所以在得到解析树之前，DNX 是无法加载任何 CLR 的，但 Roslyn 编译器除外。
2. 依赖注入（Dependency Injection，简称 DI）贯穿着整个系统栈，DI 是 DNX 的一个核心部分，所有 DNX 上的类库都构建在 DI 之上。

DNX 执行环境的分层架构如下：



Layer 0: Native Process

该层的功能非常简单，主要就是用于查找并调用 Layer 1 里的 CLR Native Host，并将系统相关的参数传递给 native host，以便后续使用。目前 Windows 下使用 DNX.exe 来处理这个事情，而 IIS 也提供了一个中介（网站 bin 目录下提供一个 `AspNet.Loader.dll`）可以将请求转发给 Native Host；而 Linux 和 Mac 则通过其相应版本的 dnx 来支持这项功能。

DNX 用法：

```
dnx.exe --lib {paths} --appbase {path} [ProgramName]
```

--lib {paths}: 程序集 dll 的保存地址（一般是引用的第三方案程序集和项目预编译程序集），该地址是 Layer 2 层的托管代码入口点可以加载程序集的地方。

--appbase {path}: 程序保存的目录，默认为 %CD%。

[ProgramName]: 程序名称，该程序所在的程序集（或者是含有 `Program::Main` 的 dll）保存在 `--lib` 路径下，默认值是 `appbase\project.json` 里的 `name`。大多数情况下，该名称都是包含着加载链的程序宿主（`Microsoft.Net.ApplicationHost`）。但是，如果你的程序包含了入口点（`Main` 方法），并被编译到 `--lib` 目录下的话，你就可以

使用该程序集的名称作为 [ProgramName]，这种方式将完全忽略加载链并直接启动你的程序。

Layer 1 : CLR Native Host

这一层的内容依赖于你所选择的 CLR 版本，该层有如下两个职责：

1. 启动 CLR，启动哪个 CLR 取决于你选择的 CLR 版本。如果是 Core CLR，该层会加载 `coreclr.dll`，配置并启动运行环境，然后创建应用程序域（AppDomain），以便运行所有的托管代码。
2. 调用托管代码的入口点（Layer 2），一旦 Native Host 的入口点返回了该线程，就会把 CLR 的线程清理干净并关闭，比如，卸载应用程序域（AppDomain）并停止运行环境。

Layer 2: Managed Entry Point

Layer 2 层（托管代码入口）是编写托管代码的第一层，其职责如下：

1. 创建 LoaderContainer（其包含需要的 ILoaders），ILoader 负责根据程序集的名称来加载程序集。CLR 需要一个程序集的话，LoaderContainer 就会使用其 ILoader 来解析所需要的程序集。
2. 从 `--lib` 的路径目录下，用根 ILoader 来加载程序集，并解析其依赖。
3. 调用程序的主入口点。

Layer 3: Application host/Application

如果开发人员将整个程序编译成程序集放在 `libpath` 目录下，那该层就是你的应用程序了。使用的时候，将含有程序入口点的程序集名称作为 [ProgramName] 的参数传入即可，Layer 2 层会直接调用该程序集。

不过，一般其它情况下，都会使用一个应用程序宿主（Application host）来解析程序的依赖内容并启动运行程序。Microsoft.Net.ApplicationHost 是运行环境提供的应用程序宿主，并拥有如下职责：

1. 解析 `project.json` 里定义的各种依赖程序集。
2. 将一个 ILoader 添加到 LoaderContainer，以便从各种地方（如源代码、NuGet、Roslyn 等）加载相应的程序集。

3. 调用程序集的入口点，将其作为下一个参数，传递给 DNX.exe。

Layer 4: Application

这一层，就是开发人员开发的程序，其运行在应用程序宿主之上。

环境配置：

要对 ASP.NET 5 程序的运行环境 DNX 进行配置，首先需要安装并配置 DNVM，CentOS 等 Linux 系统上需要先安装 Mono，可以参照文章 [CentOS 7 上部署 Mono 4 和 Jexus 5.6](#)。然后运行下面命令

```
curl -sSL  
https://raw.githubusercontent.com/aspnet/Home/master/dnvm  
install.sh | sh && source ~/.dnx/dnvm/dnvm.sh
```

```
[root@Mono ~]# curl -sSL  
https://raw.githubusercontent.com/aspnet/Home/dev/dnvminstall.sh |  
DNX_BRANCH=dev sh && source ~/.dnx/dnvm/dnvm.sh
```

Downloading dnvm as script to '/root/.dnx/dnvm'

Appending source string to /root/.bash_profile

Type 'source /root/.dnx/dnvm/dnvm.sh' to start using dnvm

运行命令 dnvm:

```

[root@Mono .dnx]# dnvm

  /-/-/ /-/-/ /-/-/ /-/-/
 /-/ /-/ /-/ /-/ /-/ /-/ /-/
/-/-/ /-/-/ /-/-/ /-/-/

.NET Version Manager - Version 1.0.0-beta6-10392
By Microsoft Open Technologies, Inc.

DNVM can be used to download versions of the .NET Execution Environment and
which version you are using.
You can control the URL of the stable and unstable channel by setting the
and DNX_UNSTABLE_FEED variables.

Current feed settings:
Default Stable: https://www.nuget.org/api/v2
Default Unstable: https://www.myget.org/F/aspnetvnext/api/v2
Current Stable Override: <none>
Current Unstable Override: <none>

Use dnvm [help|-h|--help|--help] to display help text.

[root@Mono .dnx]#

```

上述 DNVM 安装以后，系统会将 dnvm 文件复制到 /root/.dnvm 目录，并将 /root/.dnvm 目录添加到环境变量中，以便全局都可以使用。注意：这里只是安装了 DNVM，并没有安装任何版本的 DNX，要安装 DNX 的话，可以通过运行 dnvm 或 dnvm help 来查找相关的命令，具体命令如下：

```
dnvm upgrade [-x86][-x64] [-svr50][-svrc50] [-g|-global]
[-proxy <ADDRESS>]
```

1. 从 feed 源安装最新版的 DNX
2. 为已安装的 DNX 设置一个默认（default）别名
3. 将 DNX bin 添加的用户 PATH 环境变量中
4. **-g|-global** 在全局内进行安装（其它用户也可以使用）
5. **-f|-force** 强制更新成最新版（即便最新版已经安装过了）
6. **-proxy** 访问远程服务器的时候使用特定的地址作为代理

```
dnvm install <semver>|<alias>|<nupkg>|latest [-x86][-x64]
[-svr50][-svrc50] [-a|-alias <alias>] [-g|-global]
[-f|-force]
```

1. | 从 feed 源安装指定的 DNX
2. 从本地文件系统安装指定的 DNX
3. **latest** 从 feed 源安装最新版的 DNX
4. 将 DNX bin 添加到当前命令行的 path 环境变量中
5. **-p|-persistent** 将 DNX bin 添加到系统 PATH 环境变量中
6. **-a|-alias** 对指定安装的 DNX 设置别名
7. **-g|-global** 在全局内进行安装
8. **-f|-force** 强制安装指定的 DNX（即便该版本已经安装过了）

```
dnvm use <semver>|<alias>|none [-x86] [-x64] [-svr50] [-svrc50]  
[-p|-persistent] [-g|-global]
```

1. | 将 DNX bin 添加到当前命令行的 path 环境变量中
2. none 将 DNX bin 从当前命令行的 path 环境变量中删除
3. -p|-persistent 将 DNX bin 添加到系统 PATH 环境变量中
4. -g|-global 组合使用-p 将用户 PATH 修改成系统 PATH

```
dnvm list //列出所有已安装的 DNX 版本
```

```
dnvm alias //列出所有定义了别名的 DNX 版本
```

```
dnvm alias <alias> // 显示定义了别名的 DNX 名称
```

```
dnvm alias <alias> <semver> [-x86] [-x64] [-svr50] [-svrc50]  
//给指定的 DNX 版本设置别名
```

```
[root@Mono .dnx]# dnvm list
```

Active	Version	Runtime	Arch	Location	Alias
-----	-----	-----	----	-----	-----
*	1.0.0-beta4	mono		~/.dnx/runtimes	

```

[root@Mono .dnx]# dnvm install latest
Determining latest version
Latest version is 1.0.0-beta4
dnx-mono.1.0.0-beta4 already installed.
Adding /root/.dnx/runtimes/dnx-mono.1.0.0-beta4/bin to process PATH
[root@Mono .dnx]# dnvm upgrade
Determining latest version
Latest version is 1.0.0-beta4
dnx-mono.1.0.0-beta4 already installed.
Adding /root/.dnx/runtimes/dnx-mono.1.0.0-beta4/bin to process PATH
Setting alias 'default' to 'dnx-mono.1.0.0-beta4'
[root@Mono .dnx]#

```

管理程序集的 dnu 命令和 feed 源配置

通过 dnu 命令进行包管理的时候，通常使用如下命令：

dnu restore: 查询程序的所有依赖包，并将其全部下载到 **packages** 目录，该命令会下载整个依赖包以及这些依赖包所依赖的其它依赖包。

dnu install <package id>: 该 install 命令用于下载指定的程序包并添加到程序中。

dnu publish: 该命令会将你的程序打包到一个可以运行的自包含目录中。其会创建如下目录结构：

output/

output/packages

outpot/appName

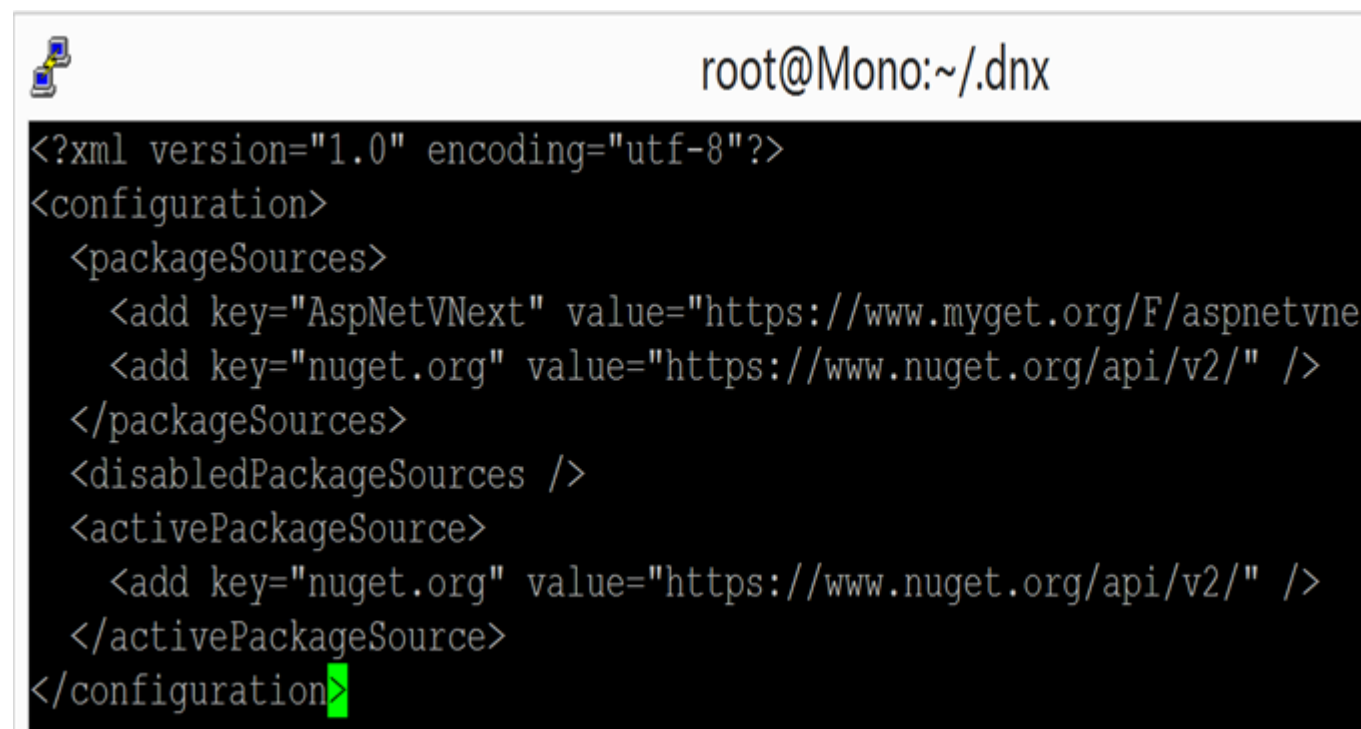
output/commandName.cmd

packages 目录包含所有应用程序需要的程序包。

appName 目录包含所有应用程序的代码，如果引用了其它项目，则在引用的其它项目也会创建各自项目的同级目录，即生成的目录会和 **AppName** 同级。

publish 命令，会将 project.json 中的 commands 节点中的各种命令，分别生成响应的命令行文件，如 commands 里的 web 命令，我们就可以通过 dnx web(格式: dnx <command>) 开运行它。

由于 dnu 在内部使用了 Nuget 命令，进行程序包的管理，所以使用的时候要正确配置 Nuget 的 feed 源，目前 ASP.NET 5 相关的包都在 myget feed 上，所以我们需要添加这个 feed 才能正常运行。这些配置信息在 *nix 下 Mono 使用的 ~/.config/NuGet/NuGet.config 文件中进行管理，示例如下：



```
root@Mono:~/.dnx
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="AspNetVNext" value="https://www.myget.org/F/aspnetvne
    <add key="nuget.org" value="https://www.nuget.org/api/v2/" />
  </packageSources>
  <disabledPackageSources />
  <activePackageSource>
    <add key="nuget.org" value="https://www.nuget.org/api/v2/" />
  </activePackageSource>
</configuration>
```

从命令行开始构建一个控制台程序

我们使用 vim 和 dnx/dnu 命令行构建一个简单的程序，创建一个目录 dnx_demo，在目录下创建一个 project.json 文件，包含以下内容：

```
{
  "version": "1.0.0-*",
  "description": "geffzhang demo project",
  "commands": {
    "runme": "dnx_demo"
  },
  "frameworks": {
```

```

"dnx451": { },

"dnxcore50": {

"dependencies": {

"System.Console": "4.0.0-beta-22816",

"Microsoft.CSharp": "4.0.0-beta-22816"

}

}

}

}

```

上面我们定义了一个命令"runme", 它指向的是工程名称:dnx_demo。我们可以通过命令行使用 dnx 运行我们的项目, 我们的项目指向传统的.NET Framework (dnx451) 和 .NET Core (dnxcore50), 所以我们可以用 dnx 和.net fx 运行。

然后在创建一个 *Program.cs* 文件, 内容如下:

```

using System;

namespace dnx_demo

{

public class Program

{

public void Main(string[] args)

{

Console.WriteLine("No Visual Studio Here!!");

Console.Read();

}

}

}

```

保存后, 我们先运行命令 dnu restore, 将我们依赖的程序包下载到 package 目录。

```
[root@Mono .dnx]# cd /root/dnx_demo/
[root@Mono dnx_demo]# vi project.json
[root@Mono dnx_demo]# dnu restore
Restoring packages for /root/dnx_demo/project.json
Writing lock file /root/dnx_demo/project.lock.json
Restore complete, 651ms elapsed
[root@Mono dnx_demo]# dnx . runme
No Visual Studio Here!!
```

目前我们运行的程序还仅仅是一个非常简单的控制台程序,还没有包括 EF, SignalR, Identity 等复杂组件,但从整个部署过程中,我们可以感觉到其实差距已经很小。首先运行和部署环境 DNVM 和 dnu, dnx 命令和 VS 2015 的环境是一致的,而且组件包都是从 Nuget 上获取,这和标准的 Windows 开发环境并没有太大区别。

参考内容:

[解读 ASP.NET 5 & MVC6 系列 \(4\) : 核心技术与环境配置](https://github.com/aspnet/Home/wiki/DNX-structure)

<https://github.com/aspnet/Home/wiki/Command-Line>

<https://github.com/aspnet/Home/wiki/Version-Manager>

<https://github.com/aspnet/Home/wiki/Package-Manager>

http://projectsaas-document.readthedocs.org/zh_CN/latest/getting-started/index.html

<https://alexanderzeitler.com/articles/Running-ASP.NET-5-beta4-in-Docker-with-DNX-runtime/>

部署 Mono 4 环境

源码安装

1、系统, 安装编译环境, 为编译 Mono 源码做准备。

```
yum -y update
```

2、安装 Mono 源码安装需要的库

```
yum -y install gcc gcc-c++ bison pkgconfig glib2-devel gettext make
libpng-devel libjpeg-devel libtiff-devel libexif-devel giflib-devel
libX11-devel freetype-devel fontconfig-devel cairo-devel
```

3、Mono 需要的 GDI+兼容 API 的库 Libgdiplus 支持 System.Drawing

```
cd /usr/local/src/
```



```
wget
http://download.mono-project.com/sources/libgdiplus/libgdiplus-3.8.ta
r.gz
tar -zxvf libgdiplus-3.8.tar.gz
cd libgdiplus-3.8
./configure --prefix=/usr
make
make install
```

4、源码安装 Mono 4

```
cd /usr/local/src/
wget
http://download.mono-project.com/sources/mono/mono-4.0.1.44.tar.bz2
tar -jxvf mono-4.0.1.44.tar.bz2
cd mono-4.0.1.44
./configure --prefix=/usr
make
make install
输入 mono -V 如有 mono 版本信息, 则安装成功.
[azureuser@mono mono-4.0.1]$ mono -V
Mono JIT compiler version 4.0.1 (tarball Mon May 11 09:53:17 CST 2015)
```

Copyright (C) 2002-2014 Novell, Inc, XamarinInc and Contributors.
www.mono-project.com

```
      TLS:          __thread
      SIGSEGV:      altstack
      Notifications: epoll
      Architecture: amd64
      Disabled:     none
Misc:      softdebug
      LLVM:         supported, not enabled.
      GC:           sgen
```

如果是 64 位版本的 CentOS, 在后续安装好 Jexus 5, 启动的时候会发生以下错误:

```
Sender: jws.exe, Sender TypeName: AppDomain
Exception Source: jws, TargetSite Name: A
Message is:
An exception was thrown by the type initializer for
Mono.Unix.Native.Stdlib
StackTrace is:
   at A.G.A (System.String[] A) [0x00000] in <filename unknown>:0
IsTerminating: True
```

建议大家在安装 mono 后，都 ldconfig 一下。（ldconfig 命令的用途，主要是在默认搜寻目录(/lib 和/usr/lib)以及动态库配置文件/etc/ld.so.conf 内所列的目录下，搜索出可共享的动态链接库(格式如前介绍,lib*.so*),进而创建出动态装入程序(ld.so)所需的连接和缓存文件.缓存文件默认为 /etc/ld.so.cache,此文件保存已排好序的动态链接库名字列表.）

另外有一点也非常重要，要在编译 Mono 的是指定安装到/usr（./configure --prefix=/usr），如果安装到一个系统根本“不了解”的文件夹，仅 ldconfig 都不行，还要在/etc/ld.so.conf 文件或/etc/ld.so.conf.d 中添加路径后再 ldconfig 才行呢。

软件包安装

CentOS 有一个 Yum 软件包管理，这极大地简化了安装 CentOS 的程序。只要你不需要最新的更新的软件程序包，通过 Yum 软件包管理是最简单，特别是对于新手 Linux/CentOS 用户的方法。

通过 Yum 包安装 Mono

参考官方文档：[Install Mono on Linux](#):

CentOS, Fedora, and derivatives

Add the Mono Project GPG signing key and the package repository **in a root shell** with:


```
rpm --import "http://keyserver.ubuntu.com/pks/lookup?op=get&search=0x3FA7E0328081BFF6A14DA29AA6A19B38D3D831EF"
yum-config-manager --add-repo http://download.mono-project.com/repo/centos/
```


Run a package upgrade to upgrade existing packages to the latest available. Then install Mono a


Users of CentOS or RHEL (or similar distributions) may need to add the [EPEL repository](#) to their

openSUSE and SLES

You can install using SUSE One-Click files (see below for descriptions):

 1-click Install **mono-devel**

 1-click Install **mono-complete**

 1-click Install **referenceassemblies-pcl**

Mono 包并没有包含在 CentOS 的仓库里，我们需要把 Mono 的仓库导入到包仓库里，在 root 用户权限下执行下面命令：

```
rpm --import
"http://keyserver.ubuntu.com/pks/lookup?op=get&search=0x3FA7E0328081BFF6A14DA29AA6A19B38D3D831EF"
```

```
yum-config-manager --add-repo http://download.mono-project.com/repo/centos/
```

有可能提示找不到 yum-config-manager ， 这个是因为系统默认没有安装这个命令，这个命令在 yum-utils 包里，可以通过命令 yum -y install yum-utils 安装。

```
[root@Mono ~]# yum -y install yum-utils
[root@Mono ~]# rpm -qa | grep yum
yum-utils-1.1.31-25.el7_0.noarch
yum-3.4.3-118.el7.centos.noarch
yum-metadata-parser-1.1.4-10.el7.x86_64
yum-plugin-fastestmirror-1.1.31-25.el7_0.noarch
```

然后执行

```
yum -y install mono-complete.x86_64 安装所有的软件包
```

```
mono-core.x86_64 0:4.0.1-4 mono-data.x86_64 0:4.0.1-4
```

```
mono-data-oracle.x86_64 0:4.0.1-4 mono-data-sqlite.x86_64 0:4.0.1-4
```

```
mono-devel.x86_64 0:4.0.1-4 mono-extras.x86_64 0:4.0.1-4
```

```
mono-locale-extras.x86_64 0:4.0.1-4 mono-mvc.x86_64 0:4.0.1-4
```

```
mono-nunit.x86_64 0:4.0.1-4 mono-reactive.x86_64 0:4.0.1-4
```

```
mono-wcf.x86_64 0:4.0.1-4 mono-web.x86_64 0:4.0.1-4
```

```
mono-winforms.x86_64 0:4.0.1-4 mono-winfxc core.x86_64 0:4.0.1-4
```

```
monodoc-core.x86_64 0:4.0.1-4
```

运行 `mono -V` 确认已经成功安装

```
[root@Mono ~]# mono -V
```

```
Mono JIT compiler version 4.0.1 (tarball Tue May 12 16:19:40 BST 2015)
```

```
Copyright (C) 2002-2014 Novell, Inc, Xamarin Inc and Contributors.
```

```
www.mono-project.com
```

```
TLS: __thread
```

```
SIGSEGV: altstack
```

```
Notifications: epoll
```

```
Architecture: amd64
```

```
Disabled: none
```

```
Misc: softdebug
```

```
LLVM: supported, not enabled.
```

```
GC: sgen
```

更新 Mono ， 执行下面的命令

```
yum -y update mono-complete.x86_64
```

绿色环境 jws.mono

为了降低 Linux 上 .NET 的学习难度, Jexus 的作者刘冰给我们制作了一款绿色版的 Linux .NET 环境包, 当中提供了 Jexus Web Server 以及 Mono 的运行环境, 带来了解压即可用, 删除即卸载便利, 同时还降低了 Linux 下 .NET 初学者的学习门槛。

使用绿色包具有以下的几项优点:

(1)、快速部署, 由于采用此方式部署仅仅需要执行一条解压命令(有需要的可自行注册环境变量), 没有编译过程, 大大节省了因为环境部署所需要消耗的宝贵时间。

(2)、针对性强: 由于每一款的绿色包都是针对其标注的 Linux 发行版进行编译, 因此绿色包具有比较强的发行版针对性。

(3)、精致而不失功能: 使用过绿色包的读者可能会发现, 它的打包文件大小甚至会比 Mono/Source 所发行的源码包还会小, 但功能却又没有减少。这个秘密就在于 Mono 与 MS.NET 不同, Mono 的库是向下兼容的, 因此, 在每款的绿色包中, 我们都会对“重复”的库进行剔除, 让包变得足够精致。

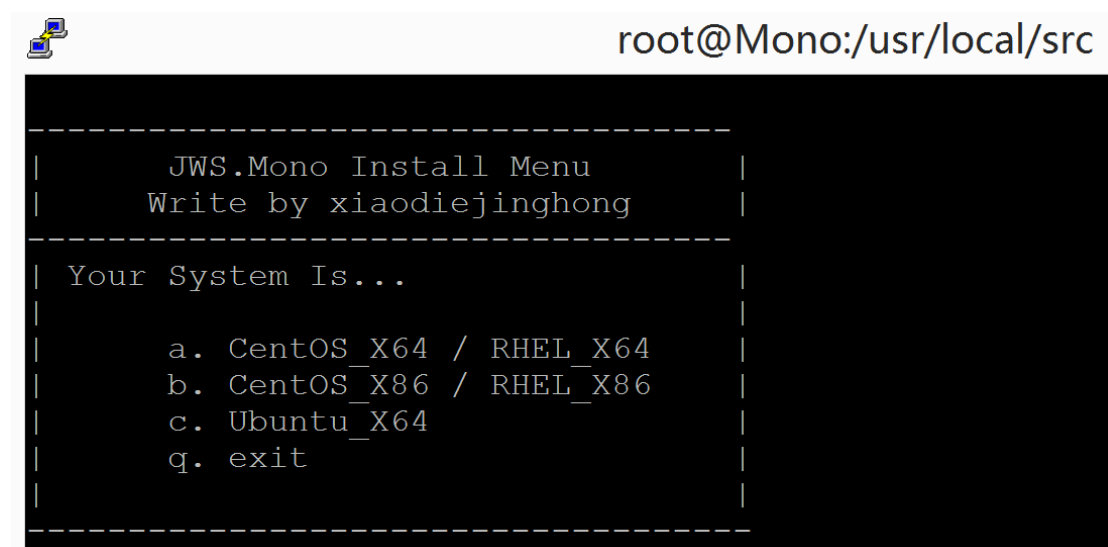
在线安装 JWS. Mono

```
wget http://jhonge.net/download/1413998270361/jwsmono\_net.sh
```

```
chmod a+x jwsmono_net.sh
```

```
./jwsmono_net.sh
```

通过以上三步, 我们可以进入到 JWS. Mono 的安装控制台, 我们只需要根据提示选择好操作系统:



```
root@Mono:/usr/local/src

-----
|           JWS.Mono Install Menu           |
|       Write by xiaodiejinghong             |
|-----|
| Your System Is...                          |
|                                             |
|   a. CentOS_X64 / RHEL_X64                |
|   b. CentOS_X86 / RHEL_X86                |
|   c. Ubuntu_X64                           |
|   q. exit                                  |
|-----|
```

然后再选择好想要的操作即可进行安装了, 然后电脑就会自动帮你下载所需要的包并且帮你安装好。

如果网络不好, 或者是内外网隔离, 可以单独下载整个包到本地。然后放入把链接放到迅雷离线、旋风离线或百度云离线等方式进行离线下载。

1. https://github.com/xiaodiejinghong/JWS_MONO_X64CentOS

2. https://github.com/xiaodiejinghong/JWS_MONO_X64CentOS

3. https://github.com/xiaodiejinghong/JWS_MONO_X64Ubuntu

选择好你的服务器操作系统版本，到上述地址，使用 download zip 下载。

使用此方式安装 jws.mono，我们只需进行以下几个步骤：

对于首次安装：

- a) 解压绿色包并切换到 data/ 目录
- b) 执行 `./install -i` 开始进行 jws.mono 的安装
- c) 提示 “Congratulations...All Install Complete~!” 方可完成安装（默认安装到 /jws.mono 中）

而对于需要升级旧版本的 jws.mono：

- a) 解压绿色包并切换到 data/ 目录
- b) 执行 `./install -u` 开始进行 jws.mono 的升级
- c) 待提示 “Congratulations...All Install Complete~!” 表示升级成功（注意：默认需要旧版本的 jws.mono 位于 /jws.mono 中）

这里还有一个新手慎用的小 Tips，执行 install 脚本的时候，还可以加入第二个参数 “--prefix”，通过 “--prefix=安装目录”，我们还可以指定 jws.mono 的安装路径，将 jws.mono 自动的安装到我们指定的目录中

jwsd 服务

使用脚本方式安装 jws.mono，我们除了可以通过进入 Jexus 目录直接操作 jws 来控制 Jexus 外，我们还可以通过另外一种方式来控制 Jexus，命令如下：

- 启动 Jexus: `service jwsd start`
- 关闭 Jexus: `service jwsd stop`
- 重启 Jexus: `service jwsd restart`
- 查看 Jexus 状态: `service jwsd status`
- 查看 Jexus 版本: `service jwsd version`
- 设置 Jexus 为开机启动: `chkconfig jwsd on`
- 取消 Jexus 开机启动: `chkconfig jwsd off`

激活 jws.mono 的图像处理

jws.mono 真的给我们带来了很大的便利，它免除了我们编译 Linux.NET 所带来的烦恼，节省了我们的时间。但是金无足赤人无完人，虽然 jws.mono 已经大致能够提供与我们自行编译相同的效果，不过它仍然有一点不足，那就是我们无法使用与图形处理相关的工作（System.Drawing）。造成这点不足的原因就是在于，我们的 jws.mono 没有内置与常见图像处理相关的库，澄清一点，这里所指的库是类似于“libpng 库”、“libjpeg 库”、“gd 库”之类的通用图形处理库，并非“libgdipplus 库”，“libgdipplus 库”是已经集成了的。

所以当有使用 jws.mono 的读者需要做一些与图像处理相关的操作时（验证码的生成之类的），页面会出现如下图的错误（大图，可以单独拖出来看）：

要解决这个问题也是比较简单的，我们只要把缺了的库补上即可。

对于 CentOS/Red Hat 操作系统的读者可在网络通畅的情况下执行以下命令：

```
yum install glib2-devel gettext make libpng-devel libjpeg-devel  
libtiff-devel libexif-devel giflib-devel libX11-devel freetype-devel  
fontconfig-devel cairo-devel
```

对于 Ubuntu/Debian 操作系统的读者则可以在网络畅通的情况下执行以下命令：

```
apt-get install libgif-dev libtiff4-dev  
apt-get install libpng12-dev libexif-dev libx11-dev  
apt-get install libxft-dev libjpeg62-dev
```

通过以上的命令，系统会自动的从 Linux 镜像服务器在线下载并安装相关的图形库。再这些库安装完毕之后，读者们只需要执行：

```
service jwsd stop  
service jwsd start
```

OWIN 服务器

OWIN 规范

OWIN 的全称是 Open Web Interface For .Net，OWIN 是 .Net 开源社区借鉴 Ruby 而制定的 .Net Web 开发架构，有着非常简单的规范定义，同时极度降低了模块间耦合。OWIN 并不是一个具体的实现，而只是一个规范，用来指导如何构建一个符合 OWIN 标准的 Web 生态环境。微软引入并推广 OWIN，同时依照 OWIN 规范，实现了 Katana。

OWIN 目的是在 web 服务器和应用程序之间隔离出一个抽象层，使它们之间解耦。OWIN 设计的 2 个目标：简单，以及尽量少的依赖其它的框架类型。这样就能够：

- 新的组件能够非常简单的开发和应用
- 程序能够简便地在 host 和 OS 上迁移

OWIN 核心定义

OWIN 将 web 应用中的 request, response, session, cookie 等所有相关信息都简化成下面的字典。本质上来说，这个字典就包含了一个 web 请求的所有上下文信息。

一个符合 OWIN 的 web 服务器，需要将请求信息包装成下面的字典类型，传递到下一层中。而下一层的组件或者应用程序，所要做的就是读取，修改这个字典的数据。最后，Web 服务器得到这个层层处理过的字典，然后输出网页到客户端。

IDictionary<string, object>

下面是具体的定义

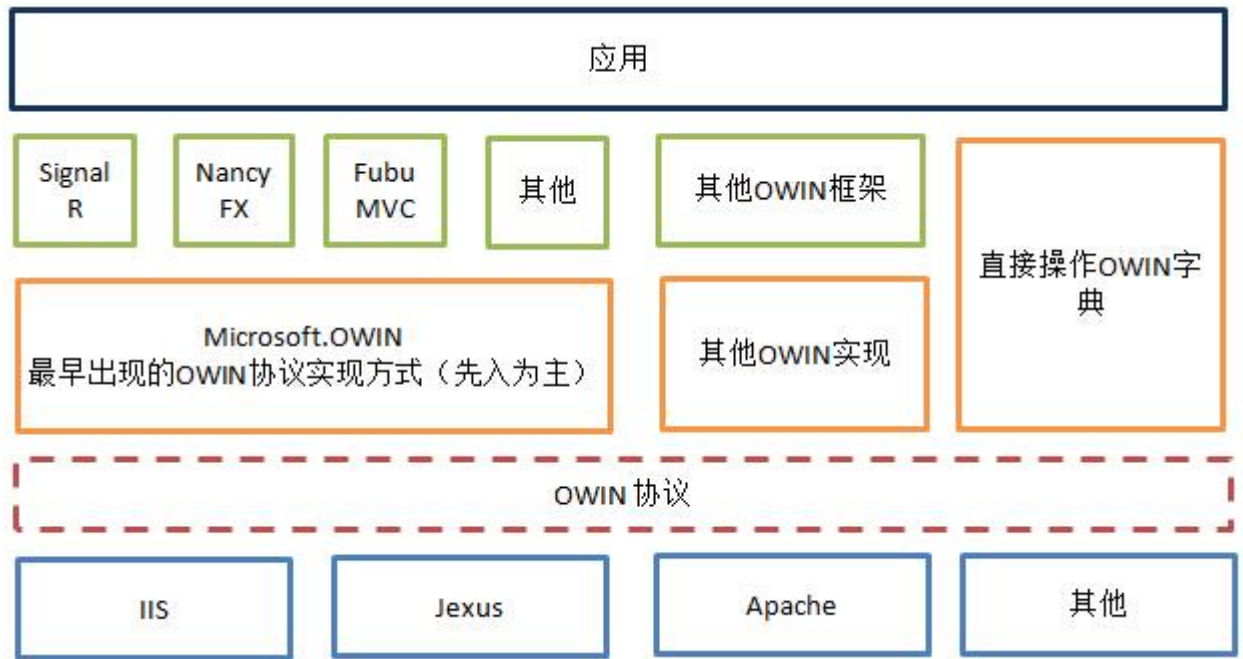
Key Name	Value Description
"owin.RequestBody"	A Stream with the request body, if any. Stream.Null MAY be used as a placeholder if there is no request body. See Request Body .
"owin.RequestHeaders"	An IDictionary<string, string[]><string, string[]=""> of request headers. See Headers .
"owin.RequestMethod"	A string containing the HTTP request method of the request (e.g., "GET", "POST").
"owin.RequestPath"	A string containing the request path. The path MUST be relative to the "root" of the application delegate; see Paths .
"owin.RequestPathBase"	A string containing the portion of the request path corresponding to the "root" of the application delegate; see Paths .
"owin.RequestProtocol"	A string containing the protocol name and version (e.g. "HTTP/1.0" or "HTTP/1.1").
"owin.RequestQueryString"	A string containing the query string component of the HTTP request URI, without the leading "?" (e.g., "foo=bar&baz=quux"). The value may be an empty string.
"owin.RequestScheme"	A string containing the URI scheme used for the request (e.g., "http", "https"); see URI Scheme .

另外一个核心是 **application delegate**, 这是所有运行在 OWIN 协议下的组件都需要遵循的接口

```
Func<IDictionary<string, object>, Task>;
```

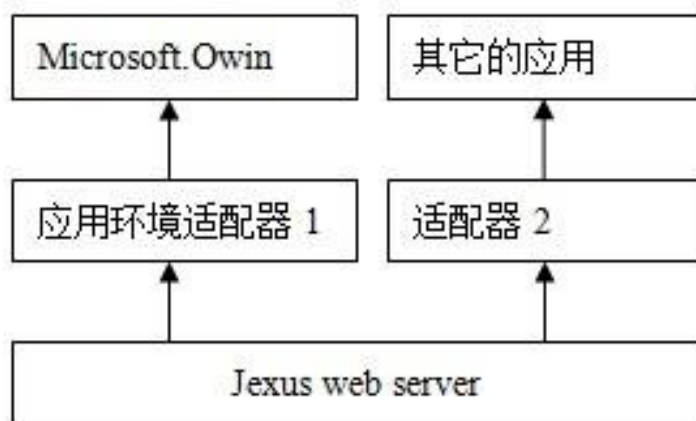
这样定义的原因是:

- 由于依赖少, 写一个 component 非常容易和简单
- 异步设计使得程序的运行效率更高, 特别是在遇到一些 I/O 密集的操作时
- application delegate 是可执行的最小单元, OWIN components 可以非常容易的互相连接组成一个 Http 处理管道



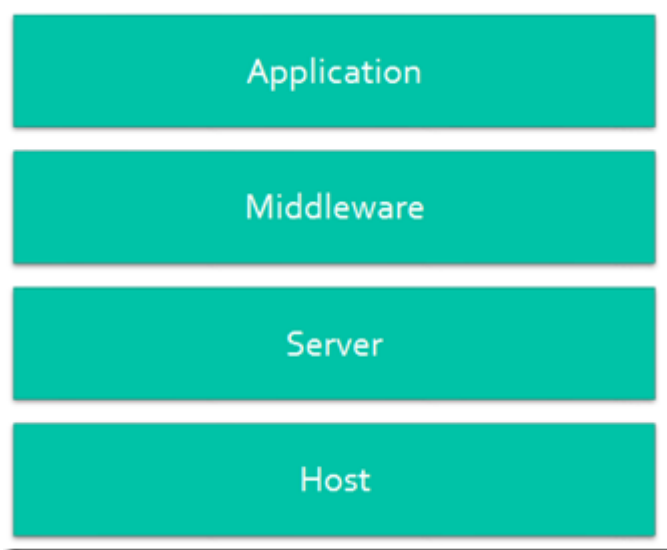
如图所示，虽然 OWIN 协议并不属于任何一方，但是出现得最早的关于 OWIN 协议的实现是微软的”Microsoft.OWIN“，因此目前许多的 OWIN 框架都是基于”Microsoft.OWIN“来实现。而”Startup“、“Configuration“、“Middleware“等组件是”Microsoft.OWIN“所提出的知识点，因此这些 OWIN 框架也就自然而然的会出现这一类的事物来。

任何公司和个人都可以有自己的实现，微软的实现 Microsoft.Owin.dll 的实现方式是很科学的，是我们开发 OWIN 服务器的一个很好的参考，目前现存的 OWIN 框架都是针对“Microsoft.OWIN”所缔造出来的框架，因此这些 OWIN 框架与 Katana 之间是无缝的连接，根本就无需适配器这一类型的玩意，而 Jexus 则不同，适配器这一说法是 Jexus 针对这些基于“Microsoft.OWIN”所实现出来的 OWIN 框架能够在 Jexus.OwinHost 中使用所诞生出的组件。国内的 MVP 刘冰也实现了一个 TinyFox 的 Owin 服务器，在后面会详细介绍。



微软 Katana 服务器

Katana 是 OWIN 规范的微软实现，Katana 项目的主要架构图如下：



1.1 Host

宿主只是一个进程，是整个 OWIN 程序的载体。这个宿主可以是 IIS, IIS Express, Console, Windows Service 等。

Host 的主要作用：

1. 管理底层进程
2. 当有请求过来的时候，选择相应的 Server 和构建 OWIN 管道处理请求。

我们最熟悉的 Host 就是 IIS/Asp.net。不过 IIS 既是 Host，也是 Server。在 IIS 中使用标准的 `HttpModule` 和 `HttpHandler` 类型来响应请求。Katana 通过在 IIS 上构建一个类似适配器，使得 OWIN 管道能够运行在 IIS 上。

我们还可以自己实现一个简单的 Host，这个 Host 可以在 Console 中，也可以是一个 Windows Service 进程。同时 Katana 也已经提供了一个可以直接运行 Host——`OwinHost.exe`

1.2 Server

负责绑定到 TCP 端口, 监听端口发送过来的请求, 同时将请求的信息依照 OWIN 规范, 包装成字典格式, 传递到下层的 Middleware. Katana 项目包含了 2 个 Server 实现:

- **Microsoft.Owin.Host.SystemWeb**

这个是用来对应 IIS 的, 由于 IIS 既是 Host, 又是 Server. 所以这个 Server 实现的作用是注册 ASP.NET HttpModule 和 HttpHandler 阻断原有的处理流程, 转而把请求发送到 OWIN 管道中处理。

Microsoft.AspNet.Host.SystemWeb 依赖于 System.Web, 和 IIS 耦合厉害, 所以无法迁移到非 IIS 服务器

- **Microsoft.Owin.Host.HttpListener**

使用 HttpListener 打开 Socket 端口, 监听请求, 然后将请求包装发送到 OWIN 管道中处理。

1.3 Middleware:

这是为组成 OWIN 管道中的组件。它可以是从简单压缩组件到 ASP.NET Web API 这样的完整框架。

当从客户端发送一个请求, 这个请求就会传到 OWIN 管道中处理。这个管道是在 startup code 中初始化的。组成管道的组件就是 Middleware. 要遵循 OWIN 标准, Middleware 应该要实现

```
Func<IDictionary<string, object>, Task>
```

Katana 提供了一个 **OwinMiddleware** 基类更加方便我们继承来实现 OWIN Middleware.

1.4 Application

这是您的程序代码。由于 Katana 并不取代 ASP.NET, 而是一种编写和托管组件的新方式, 因此现有的 ASP.NET Web API 和 SignalR 应用程序将保持不变, 因为这些框架可以参与 OWIN 管道。事实上, 对于这些类型的应用程序, Katana 组件只需使用一个小的配置类即可见。

OWIN 和 Katana 并不是一个全新的开发方式, 并不取代 ASP.NET, 是实现 Host, Server 和 Application 之间解耦, 是一种编写和托管组件的新方式。比如使用 OWIN 方式开发 Web API, 我们仍然还是使用 Asp.net Web API. 只是 Web API 变成了我们 OWIN 管道的一个组成部分了。正常开发中, 我们感觉不到 OWIN 的存在, 只是在 startup 代码中, 构建我们的 OWIN 管道。通常, 对于 Web API 和 SignalR 这种大型组件, 我们都是注册在 OWIN 管道的最后。但是像 authentication, cache 这样的组件, 我们通常会注册到管道前部。

<http://www.infoq.com/cn/news/2014/03/helios>

<https://github.com/aspnet/KestrelHttpServer>

Jexus TinyFox

TinyFox 介绍

TinyFox 是一款支持 OWIN 标准的 WEB 应用的高性能的 HTTP 服务器,是 Jexus Web Server 的“姊妹篇”。TinyFox 本身的功能是 html 服务器,所有的 WEB 应用,通过加载含有一个 OwinMain 方法的“应用程序适配器”或“插件”实现,TinyFox 与应用程序之间的数据交流格式是 OWIN 规范的字典。

一, TinyFox 有如下特点:

- 1, 跨平台: 支持 windows、linux 等常用操作系统;
- 2, 超轻量: 功能单一而明确: 除了静态文件由自身处理外, 其它的应用逻辑直接交给用户处理;
- 3, 高性能: 底层基于 libuv 开发, 是完全的异步、非阻塞、事件驱动模型, 上层代码也经过了高度优化; libuv 是 NodeJs 的基础库, libuv 是一个高性能事件驱动的程序库, 封装了 Windows 和 Unix 平台一些底层特性, 为开发者提供了统一的 API, libuv 采用了异步 (asynchronous), 事件驱动 (event-driven) 的编程风格, 其主要任务是为开发人员提供了一套事件循环和基于 I/O(或其他活动)通知的回调函数, libuv 提供了一套核心的工具集, 例如定时器, 非阻塞网络编程的支持, 异步访问文件系统, 子进程以及其他功能, 关于 libuv 的更多内容推荐参考电子书 <http://www.nowx.org/uvbook/>。
- 4, 高安全性和高稳定性: 本服务器由 Jexus Web Server 作者开发, 在很大程度上承接了 JWS 的高安全性和高稳定性素质;

二、如何使用

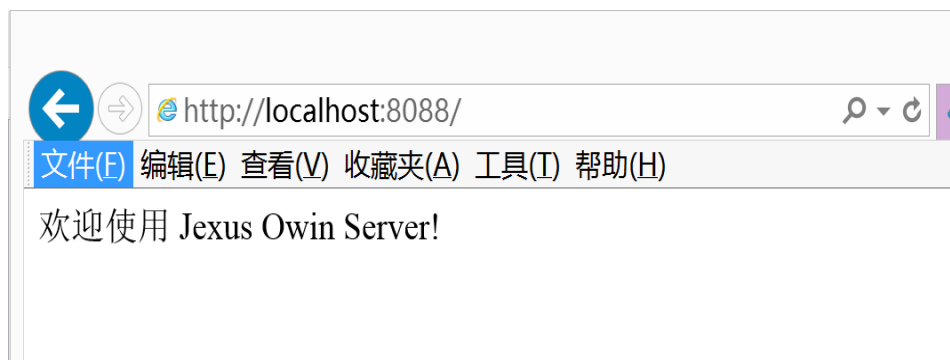
- 1, 下载安装包 <http://linuxdot.net/download/tinyfox-1.0.tar.gz>, 整个 Host 的大小用 7Z 打包了仅仅只有 1.26M 大小。拆开压缩包之后, 我们能够发现几个重要的组成, 如下图所示:

<input type="checkbox"/> 名称	修改日期	类型	大小
native	2015/6/28 19:23	文件夹	
site	2015/6/28 19:23	文件夹	
tools	2015/6/27 18:44	文件夹	
fox	2015/6/13 18:40	Windows 批处理...	1 KB
fox	2015/6/26 17:30	Shell Script	1 KB
ReadMe	2015/6/27 18:38	文本文档	2 KB
TinyFox	2015/6/27 18:15	应用程序	96 KB
TinyFox.exe	2015/6/8 9:15	CONFIG 文件	1 KB

- 2, windows 需要安装 .NET4 以及版本, 非 Windows 操作系统需要安装 mono 3.12 以上版本;
- 3, 在 windows 上, 点击 TinyFox 文件夹中的 fox.bat, 在 linux 上输入 `./fox.sh` 就能运行; 当前版本的默认端口是 8088, 可以修改 fox.bat (windows) 或 fox.sh (非 windows) 脚本, 在 TinyFox.exe 后边加上 “-p 端口号”。

自带一个测试项目，访问下面地址

<http://localhost:8088/>



<http://localhost:8088/info>



<http://localhost:8088/test>



- 4, 你的应用需要基于 OWIN 开发或者在具有 OWIN 接口的应用层框架上开发（比如 NancyFx），在此基本上，新建一个类（可称为“适配器”“接口”），这个类中，必须有一个叫“OwinMain”的方法（具体格式参见 demo），然后把编译后的 dll 放在网站 wwwroot 的 bin 文件夹或 approot 文件夹中。

惊鸿哥写了一篇文章“[OwinHost 再添新成员：TinyFox](#)”，文章详细介绍了如何部署用了 OWIN 模式的 NancyFX 的应用。

Tools 目录下 Jws.Owin-Adapters.zip 中有四个“应用”示例，base.demo 是最能体现 OWIN 原始定义，能达到“你想要什么就是什么”的境界，代码写得好的话，性能也是最高的。msowin.demo 是在插件中加载 microsoft.owin.dll 进入 ms 处理 OWIN 的流程，这个 demo 用了 ms 的 owin 管道，microsoft.owin.dll 是一个薄层，但它帮你把字典封装成了常见的 response 等对象，使用起来方便了很多，性能也很不错，是怎么利用 microsoft.owin.dll 的规则，做一个“中间件”。Nancy.demo 是在 ms owin 处理流程(管道)中加载 NancyFx，运行 NancyFx 应用层框架。

TinyFox 不依赖于 Mono “独立运行”

Tinyfxo 独立程序 linux 平台下分 32 位和 64 位的，windows 不需要。

使用 Docker 部署

Docker 部署 ASP.NET 5 应用

作为 ASP.NET 5 跨平台部署实现的一部分，微软发布了一个 ASP.NET 的 Docker 镜像文件 <https://github.com/aspnet/aspnet-docker>。当你下载了 ASP.NET 5 的 Docker 镜像，你就有了一个能够运行 ASP.NET 5 应用程序的 Linux 环境；现在你所要做的仅仅是在这个镜像中添加你的应用程序，然后启动一个容器，运行它，发布它。

创建运行环境

现在 Docker 还只能在 Linux 上运行，所以你必须找一台 Linux 机器或者装了 Linux 虚拟机的机器来运行 Docker；最新的 CentOS 和 Ubuntu 都支持 Docker。

为你的 ASP.NET 5 应用创建一个 Docker 镜像

为了能让 ASP.NET 应用程序在云端部署，你需要一个 Docker 镜像来承载你的应用。Docker 镜像的文件系统是层叠式的（AUFS 文件系统），可以这样形象的理解：你的应用程序只是“基础镜像”上层新加的一个层而已，而在我们的例子中“基础镜像”为 Docker Hub 中的 microsoft/aspnet。在 Docker 中镜像的层级是增量叠加起来的，这点跟 Git 版本控制类似，Docker 保存了每个层之间

的差异，所以当我们用 Docker 部署应用时，提交的更新不会包含 Linux 发行版内核或者 ASP.NET 的运行时，因为这些都已经存在于“基础镜像”中了，你只会提交基于此“基础镜像”构建的应用程序本身，所以 Docker 的这种差异化提交、部署机制能够确保应用程序以最快速、最小化的增量方式进行部署，为运维带来极大的便利。

可以通过 Dockerfile 创建 Docker 镜像。跟 Makefile 相似，Dockerfile 包含了供 Docker 用来构建一个镜像的所有步骤。

本教程所用到的 ASP.NET 源代码可以从 GitHub 的 [aspnet/Home](https://github.com/aspnet/Home) 仓库下的 HelloWeb 目录提取。首先，使用如下命令将源代码从 GitHub 上克隆下来：

```
[root@Mono ~]git clone git@github.com:aspnet/Home.git Home
```

切换到 aspnet-Home 的 sample 目录下：

```
[root@Mono ~]# cd Home/samples/1.0.0-beta4/HelloMvc/
```

完成后目录结构应该如下所示：

```
[root@Mono HelloMvc]# ls -la
total 248
drwxr-xr-x. 7 root root 4096 Jun 28 08:38 .
drwxr-xr-x. 5 root root 4096 Jun 21 07:23 ..
drwxr-xr-x. 2 root root 4096 Jun 21 07:23 Controllers
-rw-r--r--. 1 root root 1144 Jun 21 07:23 HelloMvc.xproj
drwxr-xr-x. 2 root root 4096 Jun 21 07:23 Models
-rw-r--r--. 1 root root 845 Jun 21 08:30 project.json
-rw-r--r--. 1 root root 209786 Jun 21 08:22 project.lock.json
drwxr-xr-x. 2 root root 4096 Jun 21 07:23 Properties
-rw-r--r--. 1 root root 550 Jun 21 07:23 Startup.cs
drwxr-xr-x. 4 root root 4096 Jun 21 07:23 Views
drwxr-xr-x. 2 root root 4096 Jun 21 07:23 wwwroot
```

然后我们在此目录下创建一个名叫 Dockerfile 的文件，并输入如下指令：

```
FROM microsoft/aspnet:1.0.0-beta4
```

```
COPY . /app
```

```
WORKDIR /app
```

```
RUN ["dnu","restore"]
```

```
EXPOSE 5004
```

```
ENTRYPOINT ["dnx","./","kestrel"]
```

让我们逐条解释此 Dockerfile 中命令的含义：

- 第一行中 FROM 命令后面的 ‘microsoft/aspnet’ 说明我们要下载 Docker Hub 中名为 ‘microsoft/aspnet:1.0.0-beta4’（此镜像也是 ASP.NET 在 Docker Hub 上的官方镜像）的镜像作为我们例子程序的“基础镜像”；
- COPY 命令告诉 Docker 在构建镜像的时候同时将当前目录下的所有文件拷贝到容器的 /app 目录下；紧接着，使用 WORKDIR 命令告诉 Docker 将容器启动目录设置为 /app 目录；
- RUN [dnu, restore] 命令告诉 Docker 运行 dnu restore 命令安装 ASP.NET 相关依赖项，这些都是 Docker 在第一次运行此容器之前要做的准备工作；
- EXPOSE 5004 命令会告诉 Docker 正在构建的镜像有个监听 5004 号端口的服务（可以查看 project.json 文件确认）也就是以此镜像为基础运行的容器需要向外暴露 5004 号端口）；

- 最后，ENTRYPOINT [dnx, ./, kestrel]命令说明每次用 Docker 启动此容器时都会自动执行 dnx kestrel 命令，同时通过运行此命令保证容器始终在运行不退出，其实 kestrel 命令就是启动了 ASP.NET 5 的服务器，启动此服务器后会启动一个监听 5004 号端口的进程，处理 HTTP 连接请求。

创建镜像

当我们编写完 Dockerfile 后，当前目录应该是如下结构，Dockerfile 和程序源代码在一起：

```
[root@Mono HelloMvc]# ls -la
total 252
drwxr-xr-x. 7 root root 4096 Jun 28 08:58 .
drwxr-xr-x. 5 root root 4096 Jun 21 07:23 ..
drwxr-xr-x. 2 root root 4096 Jun 21 07:23 Controllers
-rw-r--r--. 1 root root 124 Jun 28 08:58 Dockerfile
-rw-r--r--. 1 root root 1144 Jun 21 07:23 HelloMvc.xproj
drwxr-xr-x. 2 root root 4096 Jun 21 07:23 Models
-rw-r--r--. 1 root root 845 Jun 21 08:30 project.json
-rw-r--r--. 1 root root 209786 Jun 21 08:22 project.lock.json
drwxr-xr-x. 2 root root 4096 Jun 21 07:23 Properties
-rw-r--r--. 1 root root 550 Jun 21 07:23 Startup.cs
drwxr-xr-x. 4 root root 4096 Jun 21 07:23 Views
drwxr-xr-x. 2 root root 4096 Jun 21 07:23 wwwroot
```

现在我们来创建此 Docker 镜像。这个过程非常简单——运行 Docker 的 build 命令即可，命令如下：

```
[root@Mono HelloMvc]# docker build -t myapp .
```

这条命令运行结束后 Docker 就生成了一个名为 myapp 的镜像；同时，你对镜像所做的任何变化都能通过重新运行此命令来生成一个新的镜像。在你的 Linux 虚拟机或者开发环境中运行 docker images 命令可以看到我们刚刚创建的 myapp 镜像了：

```
[root@Mono HelloMvc]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
myapp                latest             1dbc7ed39eb0       4 minutes ago      810.9 MB
docker.io/geffzhang/centos/mono   latest             338ec2ba3ecd       2 weeks ago        555.9 MB
geffzhang/centos/mono             v4.0.1            338ec2ba3ecd       2 weeks ago        555.9 MB
docker.io/microsoft/aspnet        1.0.0-beta4       137f0562e2e0       2 weeks ago        729 MB
docker.io/centos                latest             fd44297e2ddb       9 weeks ago        215.7 MB
microsoft/aspnet                latest             16b1838c0b34       6 months ago       473.4 MB
```

你可以看到你的应用镜像以及 ASP.NET 镜像都存在于你的主机上。现在我们开始将 ASP.NET 应用程序通过 docker 部署。

运行容器

运行一个容器非常的简单，通过运行以下命令可以在你的开发机上启动 myapp 容器：

```
docker run -t -d -p 5004:5004 myapp
```

- -t 表示附加一个伪终端（tty）到容器；
- -d 表示以后台的方式运行容器，如果不使用此标识的话，ASP.NET 运行时后台的输入/输出流信息就会显示在你的终端上；

- -p 5004:5004 告诉 Docker 将容器的 5004 号端口映射到宿主机的 5004 号端口上。这样，宿主机 5004 号端口接收的数据包就会被转发到容器的 5004 号端口上了；
- 最后，myapp 是镜像的名称，我们可以用这个名称来启动之前 Dockerfile 构建的镜像。

容器启动之后（容器和镜像的区别就相当于 OO 中的对象和实例，对象是编译时的概念，实例是运行时的概念。以这样理解，容器是运行时的镜像。），可以通过以下命令查看容器运行的状态：

```
[root@Mono HelloMvc]# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
62d4b5dec406	myapp:latest	"dnx ./ kestrel"	2 minutes ago	Up 2 minutes	0.0.0.0:5004->5004/tcp	suspicious_lalande

```
[root@Mono HelloMvc]#
```

容器启动了，我们通过 <http://ip:5004> 就可以访问到 docker 里面运行的应用了。



Hello My name!

My address

© 2015 - My ASP.NET Application

相关连接：

- <http://blogs.msdn.com/b/webdev/archive/2015/01/14/running-asp-net-5-applications-in-linux-containers-with-docker.aspx>
- <http://blog.markrendle.net/fun-with-asp-net-5-and-docker/>

Docker 部署 Mono

<http://www.onegeek.com.au/articles/a-nancy-net-microservice-running-on-docker-in-under-20mb>

<https://github.com/seif/docker-mono-fastcgi-nginx>

<https://github.com/Misakai/mono-app>

<https://github.com/andrefernandes/docker-mono>

<https://github.com/azraelrabbit/monupw>

<https://github.com/catwithboots/monoconsul>

<http://blog.csdn.net/smallfish1983/article/details/38678187>

<https://github.com/neilcawse/dockernet>

Web 服务器 Jexus

Jexus 是什么

Jexus web server for linux 是一款基于.NET 兼容环境，运行于 Linux/unix 操作系统之上，以支持 ASP.NET 为核心功能的高性能 WEB 服务器，用于替换 Apache+mod_mono、Nginx+FastCgi 方案运行 ASP.NET。Jexus 不但支持 linux/freebsd 等多种操作系统这样的标志性特征，同时还拥有内核级的安全监控、入侵检测、URL 重写、无文件路由等一系列重要功能和专有特性。

Mono Xsp 和 Jexus 有什么区别：

- 速度方面：对于 ASP.NET 网页，大压力访问时 Jexus 处理速度更快；对于静态文件，Jexus 远快于 XSP，而且对磁盘的要求和影响小 N 倍；
- 功能方面：XSP 是以 ASP.NET 测试工作开发的，功能单调，而 Jexus 是作为生产环境使用的真实的 WEB 服务开发的，功能全面，因此，xsp 与 Jexus 在功能上可比性
- 稳定性方面：Jexus 有良好的容错和自动纠错能力，可以长期不间断运行，而 XSP 是单进程程序，没有任何自动纠错机制，无法保持不间断运行。
- 安全性方面：Jexus 具备快速有效的入侵检测和控制能力，XSP 没有任何安全检测功能，没有可比性；
- 多站点支持：XSP 支持一站，Jexus 支持任意多网站。

Jexus for linux 有如下特点：

- 支持多种操作系统：Jexus 最初的设计目标是“跨平台的 ASP.NET WEB 服务器”，由于在 Windows 系统上，IIS 已经是 ASP.NET 的优秀平台，所以，当前的 Jexus 以支持 Linux 和 FreeBSD 等非 Windows 系统为主要设计目标。正因为这个原因，就 Linux/FreeBSD 等平台的 ASP.NET 建设而言，Jexus 具有特别重大的意义。

- 高性能：“高性能”是 Jexus 的重要的设计目标。虽然 Jexus 基于 Mono 环境运行，但 Jexus 并非全部由 Mono 的 .NET 组件构成，对于涉及性能的关键代码，Jexus 非常巧妙地大量采用了 Linux/Unix 本身的优秀特性，从代码层保证了 Jexus 的性能优势。同时，Jexus 的框架特征也为高性能提供了基础保证。Jexus 不象 XSP 那样，纯 HTML 也需要经过 ASP.NET 处理，更不象 Apache 和 Nginx 等服务器，需要通过插件的形式间接地对 ASP.NET 进行支持，Jexus 把 HTML 静态文件处理模块、ASP.NET 处理模块、静态文件高速缓存机制、epoll/poll 数据传输机制等进行了高度集成，从架构本身入手，最大限度地提高 WEB 服务器的处理能力和传输速度。
- 安全性：Jexus 内核含有安全监控机制，绝大多数恶意访问在进入网站前就会被 Jexus 直接禁止，这是 Jexus 有别于其它 WEB 服务器的又一大特色，所以，Jexus 特别适合那些对安全要求较高的企业网站或政府网站使用。
- 稳定性：从运行机制而言，Jexus 系统中，有专门检测工作进程执行状态的管理单元，任何一个进程退出或者任何一个 ASP.NET 网站应用程序域退出，被会被管理单元发现并得到重启，从而保证了 Jexus 能够 7*24 小时不间断工作；从程序本身而言，Jexus 程序代码力求简洁，BUG 很少，同时，Jexus 的每个版本在正式发布之前，都要经过严格的压力测试，影响稳定性的因素，几乎在正式发布之前即已被全部排除。
- 功能强大：Jexus 支持 URL 重写，支持多目标服务器的反向代理，支持 PHP，支持 GZIP 压缩传输，并且，可以利用不同端口、不同虚拟路径、不同域名设置任意数量的网站，这些功能要素，表明了 Jexus 是一款功能完整而强劲的 WEB 服务器。
- 安装、配置、操作极为简单，服务社区化，各种问题能得到快速的处理，有良好的后续服务支撑能力。

Linux 软件包有一个做得非常好的地方，那就是对于这款软件的使用手册、帮助文档往往都会存在软件本身之中，同样关于 Jexus 的使用方法我们也可以通过 Jexus 软件包中的“readme”查阅得到。

我们先看一下 Jexus 目录中有些什么东西：

```
[root@Mono jexus]# ./jws -v
Jexus/5.6.4 Linux

[root@Mono jexus]# ls -la
.          jwsHttpd2.exe    jwsState.exe    jxHost2.dll.so  state2.conf
..         jwsHttpd2.exe.so jxAspx2.dll     jxHost.dll      state4.conf
def.py     jwsHttpd.exe     jxAspx2.dll.so  jxHost.dll.so
jws        jwsHttpd.exe.so  jxAspx.dll      log
jws.conf   jws.log          jxAspx.dll.so   os.def
jws.exe    jwsLog.exe       jxHost2.dll     siteconf
[root@Mono jexus]#
```

我们这里介绍的 Jexus 版本是 5.6.4, Jexus 主要由两个文件夹(log 和 siteconf), 一个脚本文件 jws 和一些的其他文件。

脚本文件 jws 提供了 5 个基本操作：

```
[azureuser@mono jexus]$ ./jws help
```

```
Usage: jws {start|stop|restart|regsvr|status|-v}
```

- jws start: 启动 Jexus，默认启动 Jexus 提供的 ASP.NET 状态服务；(如果需要开机自启动的，可以把脚本的全路径[包括脚本本身]添加到

/etc/rc.local 中) jws restart: 重启 Jexus, 如果命令后边加网站名作为参数, 那么就表示启动或重启指定的网站;

- jws stop: 停止 Jexus, 如果命令后边加网站名作为参数, 那么就表示停止指定的网站; 同时停止 Jexus 提供的 ASP.NET 状态服务;
- jws regsvr: 注册 jexus 所需要的全局程序集 (本命令只在安装或更新 jexus 后才用, 而且必须用;)。
- jws status: 检查 Jexus 的运行状态
- jws -v: 查看 Jexus 的版本

Jexus v6.0 架构有很大的变化。基于 v5.6.3.12 开发, v6.0 是每个网站拥有独立的工作进程, 不同网站可以使用不同的用户身份, 不同的网站可以使用不同的工作进程数量。

安装 Jexus

安装前的准备工作:

- * 需要 libc2.3.2 或更高版本的支持 (可用 `ldd --version` 查询版本情况), 如果需要启用 https, 系统中还需具备 libssl 库文件, 比如 libssl.so.0.9.8。
- * 系统已经安装好 mono 2.10.9 或更高版本 (当前最新正式版本是 mono 4.0.1)。

Mono 的官方网址是: www.go-mono.com。

Mono 的下载地址:

<http://www.go-mono.com/mono-downloads/download.html>。

Mono 的具体安装办法, 请前面章节的有关文章。

从 5.3.1 版本开始, Jexus 的安装过程简化了, 内置了两个新的脚本, 它们分别是 “install” 和 “upgrade”, 对应原来的 “安装” 和 “升级”, 将 Jexus 的安装过程自动化了。jexus 本身的安装和升级方式并没有发生更多的变化, 因此对于已经习惯于 5.3.1 之前版本的使用方式的用户, 你们仍然可以采用之前的安装方式。

```
cd /usr/local/src/  
wget http://linuxdot.net/down/jexus-5.6.4.tar.gz  
tar -zxvf jexus-5.6.4.tar.gz  
cd jexus-5.6.4  
sudo ./install
```

安装完成~!!! 在执行 install 脚本时, jexus 默认是安装到 /usr/jexus/ 中, 如果想安装到其他目录, 可以在执行该脚本时增加一个目录的参数 (比如 `sudo install /usr/local/jexus/`)。对于升级就是执行 upgrade 脚本。

启动 jexus 检查是否正常了

```
cd /usr/jexus  
sudo ./jws start
```

```
[azureuser@mono jexus-5.4.3]$ cd /usr/jexus/  
[azureuser@mono jexus]$ sudo ./jws start  
Start ... OK  
[azureuser@mono jexus]$ cat log/jws.log
```

06-28 04:19:16: * Jexus web server start-up success.

看到了 Jexus 已经成功启动的日志，我们还可以通过浏览器来检查下：

<http://127.0.0.1/info>

将 Jexus 安装为系统服务

Centos 6 可以使用下面的脚本将 Jexus 安装为服务，在目录/etc/init.d/创建一个脚本 jws：

```
[azureuser@mono jexus-5.4.3]cd /etc/init.d/  
[azureuser@mono jexus-5.4.3]vi jws  
按 i 编辑模式，粘贴下面内容  
#!/bin/bash  
#chkconfig: 2345 80 05  
#description:jws  
#  
. /etc/rc.d/init.d/functions
```

```
case "$1" in  
start)  
echo "Jexus Start.."  
/usr/jexus/jws start  
;;  
stop)  
echo "Jexus Stop.."  
/usr/jexus/jws stop  
;;  
restart)  
echo "Jexus Restart"  
/usr/jexus/jws restart  
;;  
status)  
/usr/jexus/jws status  
*)  
exit 1  
;;  
esac
```

```
exit $RETVAL
```

按 ESC 后，输入:wq 保存文件

把这个脚本作为“服务”加入

```
[azureuser@mono jexus-5.4.3]chmod 766 jws
```

```
[azureuser@mono jexus-5.4.3]chkconfig --add jws
```

```
[azureuser@mono jexus-5.4.3]service jws start #启动服务
```

Centos 7 使用 systemd 管理系统服务，我们把 Jexus 安装为系统服务，就是要写个 Systemd Service 脚本：

在目录/etc/systemd/system 下创建一个文件 jws.service, Systemd Service 的配置文件必须以.service 结尾的单元，用于控制由 systemd 控制或监视的进程，内容如下：

```
[root@TENCENT64 /etc/systemd/system]# vi jws.service
```

```
[Unit]
```

```
DefaultDependencies=no
```

```
Conflicts=shutdown.target
```

```
Description=Jexus web server daemon
```

```
[Service]
```

```
Type=oneshot
```

```
ExecStart=/usr/jexus/jws start
```

```
ExecStop=/usr/jexus/jws stop
```

```
RemainAfterExit=yes
```

```
[Install]
```

```
WantedBy=multi-user.target
```

Systemd 服务内容主要分为三部分，控制单元（unit）的定义，服务（service）的定义和安装（install）部分。

使用下面命令启用 jws.service 服务

```
[root@TENCENT64 /etc/systemd/system]# systemctl enable jws.service
```

使用命令启动服务

```
[root@TENCENT64 /etc/systemd/system]# systemctl start jws.service
```

使用命令查看下服务

```
[root@TENCENT64 /etc/systemd/system]# systemctl status jws.service
```

```
jws.service - Jexus web server daemon
```

```
Loaded: loaded (/etc/systemd/system/jws.service; enabled)
```

```
Active: active (exited) since Wed 2015-07-01 19:50:14 CST; 3min 7s ago
```

```
Main PID: 24441 (code=exited, status=0/SUCCESS)
```

```
Jul 01 19:50:14 TENCENT64.site jws[24441]: Starting ... OK
```

Jul 01 19:50:14 TENCENT64.site systemd[1]: Started Jexus web server daemon.

Jexus 的配置

Jexus 的配置分为两个方面，一个是 Jexus 全局配置（jws.conf），另外则是网站独立的配置（默认所有的网站配置文件都在 siteconf 文件夹中）。网站的独立配置可以调用很多 Jexus 的高级功能。

Jexus 服务器配置

用 vi 打开 Jexus 的安装目录下 jws.conf 文件就可以看到 Jexus 的配置项，和 IIS 不一样的：

```
[azureuser@mono jexus]$ vi jws.conf
SiteLogDir=log
SiteConfigDir=siteconf

Runtime=v4.0.30319
# LLVM=yes
# httpd.processes=2
# httpd.user=www-data

# php-fcgi.set=/usr/bin/php-cgi,6

# CertificateFile=/xxxx/xx.crt
# CertificateKeyFile=/xxxx/xx.key
```

下面表格列出各项配置：

配置	是否必须	描述	IIS 等效配置
SiteLogDir	是	该设置项设置 Jexus 的运行日志记录、各网站运行的日志记录都会存放到该文件夹中，默认在 Jexus 安装目录的 log 目录。此外，当记录的日志文件大小达到一定程度的时候，Jexus 会自动的把日志写到一个新的日志文件中，不会一直通过追加写的方式把所有的日志记录都写到	"system.applicati onHost/log" 和 "system.applicati onHost/sites/site/ logFile"

		同一文件中。	
SiteConfigDir	是	Web 网站配置文件的存放目录，可以使用和 jws.exe 所在目录的相对路径	没有
Runtime	否	该设置项用于配置 ASP.NET Runtime 的版本，默认是 v4.0.30319，除非有特殊需求需要改动 Runtime 的版本，一般情况下无需修改	"system.applicationHost/applicationPools/add/managedRuntimeVersion"
httpd.processes	否	Jexus 中默认的工作进程为 1 个，单个进程最大的并发数固定为 1 万个，通过开启多个工作进程可以提高 Jexus 处理并发的能力，开启多个进程的方式非常简单，只要修改 httpd.processes 的值并重启 Jexus 即可。不过这里有几点需要注意的地方： [1]Jexus 最大支持的工作进程为 8 个，因此这里最大只能填写 8；如果开启了多个 Jexus 的工作进程，请注意网站的 Session 的保存方式，请通过修改网站 config 文件来修改 Session 的保存方式，避免采用进程的方式保存 Session，否则将会造成 Session 的不同步，给网站的运作造成不必要的麻烦，Session 的保存可以借助第三方的方式来保存（比如数据库，或者 Jexus 提供的强大的 ASP.NET 服务状态保存）；Jexus 工作进程的设置跟服务器的 CPU 内核数和内存数有关，如果你的 Cpu 只有一个核，而你配置开启了两个进程，也只能达到一个容错的效果，能够承载的并发数是不会上去的。详细可以参考《让 Jexus 支持高并发请求的优化技巧》。	"system.applicationHost/applicationPools/add/processModel/maxProcesses"
httpd.user	否	应用程序池的运行账号，例如 httpd.user=www-data。默认是 root	"system.applicationHost/applicationPools/add/processModel/identity"

			yType” 和 ”system.appli cationHost/ap plicationPool s/add/process Model/userNam e”
httpd.MaxTotal Memory	否	这个是 5.5 之后的版本才具有的一个特性，回收工作进程开始工作之前所有的工作进程可以消耗的最大物理内存总理（单位为 MB），他的范围值从 256 到机器物理内存的 80%，默认值为 0，表示 Jexus 自动调整，请注意的是每个工作进程使用的最小内存为 128Mb	没有
httpd.MaxCpuTi me	否	这个是 5.5 之后的版本才具有的一个特性，表示单个工作进程在被回收之前可以消耗的 cpu 资源时间，他的范围是 600-14400，默认值是 3600	没有
LLVM	否	是否开启 LLVM 编译器功能。开启 LLVM 可以把中间语言的编译交给 LLVM 编译器，这可以加快编译的速度，编译出来的代码性能或许会更优秀一些。不过值得注意的地方：并不是所有的 Linux 都带有 LLVM 编译器，并且 mono 自带的编译器编译效率和编译生成的代码性能上也不差什么，因此如果没有特殊的需要，这一项可以让他保持默认关闭	没有
php-fcgi.set	否	此功能是为了让 Jexus 支持 PHP，由于这里与.NET 关系不大，这里就不做过多的介绍了，具体内容参照配置 PHP 和配置 Perl。	没有
CertificateFil e	否	服务器证书文件（x.509 格式）。默认值为空，目前版本的 Jexus 只支持一个服务器证书文件	没有
CertificateKey File	否	服务器证书的私钥文件（PEM 格式）。默认值为空	没有

Jexus 网站配置

Jexus 支持多站点，可以用不同的端口、域名、虚拟路径设置任意多的网站。必须把所有网站配置文件放到 jws.conf 指定的网站配置文件夹内，也就是配置项：“SiteConfigDir”，所定义的 Jexus 网站配置文件存放的目录地址（这个文件夹常常 jws 工作目录内的“siteconf”文件夹），这个文件夹除了网站配置文件，不能有其它任何文件，因为 jexus 会认为这儿的任何一个文件都代表着一个不同的网站。

每个网站有且只有一个配置文件，配置文件的文件名就是这个网站的名称，比如 www.mysite.com 这个网站，配置文件名可以写成“mysite”，当然也可以写成其它文件名，以便管理员容易记忆和识别，但要特别注意：文件名不能有空格。Jexus 中默认 SiteConfigDir 的配置为“siteconf”，为此我们进入到“siteconf”目录中，里面已经存在了一个缺省的配置文件“default”，我们用 vi 打开此文件。

```
[azureuser@mono siteconf]$ vi default
#####
# Web Site: Default
#####

port=80
root=/ /var/www/default
hosts=*      #OR your.com, *.your.com

# addr=0.0.0.0
# CheckQuery=false
# NoLog=true
# NoFile=/index.aspx
# Keep_Alive=false
# UseGZIP=true
# UseHttps=true
# DenyFrom=192.168.0.233, 192.168.1.*, 192.168.2.0/24
# AllowFrom=192.168.*.*
# DenyDirs=~/.cgi, ~/.upfiles

# rewrite=~/.+?\.(asp|php|cgi|pl|sh)$ /index.aspx

# reproxy=/bbs/ http://192.168.1.112/bbs/
```

该缺省文件已经包含了作者为我们提供的多项配置，其中包括三行处于运行状态（未注释的）和多行处于关闭状态（#号注释的）的配置项。

配置	是否必须	描述	IIS 等效配置
port	是	Jexus 监听的端口。这里注意了，	"system.applicati

		Jexus 不支持在同一网站配置文件同时设置两个或者两个以上的监听端口，因此如果需要设置监听计算机多个端口的话，需要通过设置多个配置文件的方式来设置并监听多个端口。	onHost/sites/site/binding"
addr	否	网站的 ip 地址，默认是 addr=0.0.0.0，注意目前尚不支持 IPV6 的地址	"system.applicationHost/sites/site/binding"
hosts	否	配置网站的域名，如果有的话；如果没有则配一个“*”。当这里配置了一个域名，譬如“mysite.com”，当用户访问“mysite.com”时，Jexus 会自动的把访问映射到此配置文件中（前提你有很多网站）。	"system.applicationHost/sites/site/binding"
root	是	该网站根目录的位置。该配置项有两个参数需要填写，分别是前面的“/”，和一个空格间隔后的网站的网站目录的物理地址。这里的作用跟磁盘挂载非常像，也是把某个文件目录挂载到 Jexus 中，成为一个网站目录，因此，第一个参数未必一定要填写代表根目录的“/”，是可以填写其他的，譬如“/admin/”，但是这会造成一个问题，假设你网站的域名是“mysite.com”，而你“host”中的第一个参数填写了“/admin/”，这时，你直接访问“majianle.com”是没办法访问到你的网站的，需要访问“mysite.com/admin/”才能访问到你挂载的网站。这里还望各位读者多加留意。	
indexes	否	默认文档列表，例如当 indexes=index.aspx,index.htm，当访问/ 如果存在 index.aspx 就会被解析到 index.aspx，然后是 index.htm，如果这两个都不存在则返回 404，如果没有设置这一项，Jexus 将使用它	"system.webServer/defaultDocument"

		内置的文档列表	
rewrite	否	Jexus 的 URLRewrite 使用的配置项是“rewrite”，它的重写是基于正则表达式的匹配重写，使用方法是“rewrite=正则匹配并替换的 URL 新生成的 URL”。	“system.webServer/rewrite/rules”
denyfrom	否	IP 访问地址限制，有时候我们需要一些特殊的功能，对一些特殊的 IP 进行访问的过滤，不允许它访问我们的网站。作者非常贴心的在 Jexus 中也加入了 ipdeny 的这么一个功能，使用方法也是很简单的，只要在“denyfrom”配置项中配置上具体的 IP 或者是网段，就可以实现对 IP 或某个网段进行 IP 的访问控制。	“system.webServer/security/ipSecurity”
allowfrom	否	IP 地址限制，有时候我们需要一些特殊的功能，对一些特殊的 IP 进行访问的过滤，只允许它访问我们的网站。作者非常贴心的在 Jexus 中也加入了 ipdeny 的这么一个功能，使用方法也是很简单的，只要在“allowfrom”配置项中配置上具体的 IP 或者是网段，就可以实现对 IP 或某个网段进行 IP 的访问控制。	“system.webServer/security/ipSecurity”
DenyDirs	否	隐藏的部分。当设置 DenyDirs = bin, App_code 时，对此类 URL 路径的访问被拒绝。	“system.webServer/security/requestFiltering/hiddenSegments”
checkquery	否	Query strings 限制，Jexus 使用内置的逻辑来执行 QueryString 的安全检查，默认值是 true，设置为 true 对 Jexus 的性能有一定的影响，关掉本项可以提高服务器速度，但就安全而言，不建议关掉它。	没有
nofile	否	类似于 IIS 的 404 自定义错误页。这是 Jexus 特有的功能，指的是如果服务器不存在用户要访问的文件，服务器将使用什么文件应答。路由后，原 RUL 路径	“system.webServer/httpErrors”

		<p>会存贮在 Jexus 特有一个服务器变量“X-Real-Uri”中。语法如下： nofile=/mvc/controller.aspx; nofile 对于 php 的 MVC 框架有重要意义，在 php 的 mvc 中，只需要写成 NoFile=/index.php，就会把请求转发给 index.php 处理。其实，有相当多的人，在用 jexus 跑 asp.net 的同时，也在用 jexus 跑 php 程序，而 php 网站，不少是 WordPress 的，它的 mvc 模式，在 jexus 上，只需要一句 nofile 就可以搞定。</p>	
nolog	否	<p>日志标记，默认值为 false，当设置为 true，Jexus 将停止生成这个站点的日志文件，用网站日志功能会提高 WEB 服务器系统的处理速度，但不足也是明显的，就是你无法详细了解网站的访问情况了。</p>	"system.applicationHost/sites/site/logFile"
keep_alive	否	<p>长连接开关，默认值是 true，即默认使用长连接，可以不填</p>	"system.webServer/httpProtocol/allowKeepAlive"
reproxy	否	<p>反向代理规则，参看后面的详细介绍</p>	没有
fastcgi.add	否	<p>对于 TCP 连接： fastcgi.add=需要 fast-cgi 处理的文件扩展名 tcp:fast-cgi 服务的 IP 地址:端口 例如： fastcgi.add=php,php3 tcp:127.0.0.1:9000 对于 unix sockets： fastcgi.add=需要 fcgi 处理的文件扩展名 socket:路径例如： fastcgi.add=php,php3 socket:/tmp/phpsvr</p>	没有
usegzip	否	<p>Gzip 压缩标志，默认值是 true，Jexus 提供了一个针对静态文件的压缩功能，使用的算法是目前比较流行的 Gzip 压缩算法。各位读者如果需要开启 Jexus 的 Gzip 压缩传输功能，则需要在相</p>	"system.webServer/urlCompression/doStaticCompression"

		应的网站配置文件中开启该功能。 usegzip=true #即 UseGzip 解释：启用这个功能后，当用户访问“.htm”“.js”等文件时，Jexus 会将这些文件进行 GZIP 压缩后发送给用户浏览器，这样，可以节约更多的网络带宽	
usehttps	否	SSL 标志，要启用 HTTPS，这个标志必须是 true，同时 port 必须被设置为 443. 默认值为 false，和服务器的证书配置搭配使用。	"system.applicationHost/sites/site/binding"

介绍完默认启用的三个配置项，接下来介绍一些高级的配置项，这些默认都是未启用的配置项。

Jexus 的 NOFile 功能

网站的用户在访问我们网站的时候很有可能会输入一个错误的 URL 地址，此 URL 或许是因为用户的输入错误，或许是网站的变更造成地址失效，又或者其他各种各样的原因。面对这种情况，我们一般都会通过定制一个错误页（404 或其他个性化的提示页面），当然这都是在写网站项目时通过.NET 来完成。Jexus 的第一项高级功能“NOFile”，它的功能跟我们自定义 404 页面是一样的，也是通过检测用户请求的资源是否存在，不存在则自动的跳转到一个我们默认好的页面去。我们来自定制一个 404 页面：

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;
charset=utf-8">
    <title>404</title>
  </head>
  <body>
    <h1>这里是一个自定义的 404 页面</是一个自定义的 404 页面 h1>
  </body>
</html>
```

然后我们在网站配置文件“default”中添加并设置“NOFILE”设置项：

```
#####
# Web Site: Default
#####

port=80
root=/ /var/www/default
```

```
hosts=*      #OR your.com,*.your.com
```

```
NoFile=/404.html
```

保存退出并重启 Jexus，我们在浏览器中输入一个错误的 URL 地址，就会以 404.html 页面替代。

Jexus 的 URL 重写

URL 重写也叫 URLRewrite，简单说，URLRewrite 就是对用户在浏览器中输入的 URL 地址（虚假的或者是经过改造的）在 Web 服务器中重新生成一个真实的 URL 资源路径。

在 Jexus 的 URLRewrite 使用的配置项是“rewrite”，它的重写是基于正则表达式的匹配重写，使用方法是“rewrite=正则匹配并替换的 URL 新生成的 URL”。比如，希望别人访问“.php”类型的文件时，服务器返回 /404.html 这个文件：

```
rewrite=^/.+?\.(asp|php|cgi)$ /404.html
```

“rewrite=”的后面是两部分组成，两部分之间由一个空格分开。

空格前是匹配的条件：用正则表达式描述 URL 的匹配条件。

空格后是匹配的目标：指的是如果用户访问的路径合乎前面的匹配条件，服务器将以哪个规则回应。

又如：

把“/bbs”解析为“/bbs/index.aspx”，把“/bbs/file-1”匹配为

“/bbs/show.aspx?id=1”：

```
rewrite=/bbs$ /bbs/index.aspx
```

```
rewrite=/bbs/file-([0-9]{1,6})$ /bbs/show.aspx?id=$1
```

格式解释：rewrite 的等号后含有两部分内容，用空格分开。前半部分是一个正则表达式，用于描述需要 URL 重写的（用户浏览器中的）url 路径样式，后半部分是当用户的 URL 合乎前面的正则表达式时，JWS 应该重写和访问的真实 URL 路径。

具体参考 <http://www.lexm.com/2013/10/jexus-series-url-rewrite/>

IP 访问控制

有时候我们需要一些特殊的功能，对一些特殊的 IP 进行访问的过滤，不允许它访问我们的网站。作者非常贴心的在 Jexus 中也加入了 ipdeny 的这么一个功能，使用方法也是很简单的，只要在“denyfrom”配置项中配置上具体的 IP 或者是网段，就可以实现对 IP 或某个网段进行 IP 的访问控制。

“阻止/允许”或“黑名单、白名单”

<http://www.lexm.com/2013/10/jexus-series-ip-and-domain-restrictions/>

Jexus 反向代理

反向代理是 Jexus 的一个重要功能，使用反向代理最大的一个优点就是可以实现负载均衡。当网站的访问量很高或者需要进行很复杂的计算时，单台的 Web 服务器往往是无法承载这巨大的压力的，我们需要多台 Web 服务器来并行的运作，这时我们就可以通过反向代理的负载均衡来获得服务器性能的堆叠。当大量的访问请求到达，反向代理服务器通过轮询，把请求分派到各台 Web 服务器处理，代理服务器本身并不做任何的请求应答，仅仅是做数据的转发，相对于应答处理，数据转发对服务器的性能消耗要小很多，我们的网站也可以获得更大的承载。

此外，反向代理还可以是我们的网站更安全，我们可以在反向代理服务器中设置更多的安全设置，在恶意的访问到达 Web 服务器之前先进行拦截和封杀，提高 Web 服务器的安全和网站的稳定性，这是其它服务器没有的。不但是反向代理，而且这时也把 jexus 当成了一台“入侵检测设备”（IDS）。

Jexus 的反向代理的配置关键字是 reproxy:

```
reproxy= /abc/ http://www.xxxx.com:890/abc/
```

参数的值由本站 RUL 根路径和目标网站 URL 根路径两部分组成，之间用空隔分开。

*技巧：反向代理的目标地址可以有多个，用英文逗号分隔，如：

```
reproxy=/abc/ http://192.168.0.3/abc/,http://192.168.0.4/abc/
```

这时，当用户访问/abc/时，jexus 就会随机选择一台服务器进行访问，达到负载均衡或服务器集群的效果。

利用 Jexus 的“多目标反向代理”功能，我们很容易实现多服务器的负载均衡，构成一个 WEB 服务器集群，大大提高网站的负载能力。Jexus 反向代理有一个特点：如果前端服务器本地网站中有内容，它就会直接使用前端服务器的本地内容，而不会将请求发送给后端服务器。

据我们所知，对于网站，一个 ASPX 网页上常常会含有图片、JS、CSS 等大量的静态文件，其比例甚至可以达到一比三十或更多，因此，如果你希望为静态文件提供更快的反应速度，你只需要把静态文件放到前端服务器对应网站的对应文件夹下就行。

Jexus 的反代很灵活，既可以以文件夹为单位对某个单一目标进行反代，也可以整站反代，关键是看你怎么设置。

比如：如果目标网站是：http://a.b.com/

那么，你就可以设：

```
reproxy=/ http://a.b.com/
```

这就是整站。甚至还可以把一个整站作为你的一个虚拟路径：

如：

```
reproxy=/blog/ http://blog.xy.com/
```

反向代理负载均衡具体的配置方法如下：

假设有一台面向用户的服务器作为前端服务器，IP 地址为 1.1.1.1，在这台服务器的后面，有二台工作服务器，分别为 1.1.1.2 和 1.1.1.3。1.1.1.2 由 80

端口提供服务，1.1.1.3 由 80、81 两个端口(更多也行)同时提供服务，那么，只需要一行文字就可以实现这两台服务器的负载均衡：

在 192.168.1.1 的网站配置文件中加一句：

```
rewrite=/ http://1.1.1.2/,http://1.1.1.3:80/,http://1.1.1.3:81/
```

配置文件生效后，当用户访问 1.1.1.1 时，Jexus 就会把用户的请求随机转发给 1.1.1.2 和 1.1.1.3 这两台工作服务器的三个服务端口，从而实现了负载均衡的目的。

注意：

多台服务器同时为同一个网站提供服务，存在着一个 SESSION 同步的部题。在 Jexus 上同步 SESSION 很方便：你只需要把每个网站的 State 服务器 IP 地址指向同一台服务器 IP 地址就行。具体方法是，修改网站的 web.config，在 system.web 节中加入：

```
<sessionState mode="StateServer"
stateConnectionString="tcpip=x.x.x.x:42424" timeout="60" />
```

Jexus 对 State 服务器的控制能力比较强，一旦 State 服务器的服务程序崩溃退出，Jws 就会在 10 秒之内自动重启它，所以，就机制而言，其稳定性是很高的，一般不会出现问題。

启用 HTTPS 进行 SSL 安全传输

第一步：登记 SSL 库。

首先查看“/lib”文件夹中 SSL 库文件名，该文件名应该是“libssl.so.版本号”，如果没有列出文件名，就证明你的系统还没有安装 OpenSSL，请安装后再操作。

（注：我的系统的 SSL 库文件名是：libssl.so.0.9.8）

用 VIM 打开 “/usr/etc/mono/config” 这个文件，在“<configuration>”节中，添加下面这一句：

```
<dllmap dll="libssl" target="libssl.so.0.9.8" os="!windows" />
```

（注意：dll 的值一定要填“libssl”，target 的值必须填你系统/lib 文件夹中的 ssl 库文件名）

第二步：生成服务器端 SSL 证书和私钥。

具体操作方法参见有关 APACHE 的 SSL 证书生成方法

第三步：把证书和私钥文件名填写到 JWS 的配置文件中。

在 Jexus 文件夹中，打开 “jws.conf”，添加下面两句：

CertificateFile=证书文件路径和文件名

CertificateKeyFile=私钥文件路径和文件名

第四步：开启网站的 HTTPS 功能。

在需要进行 https 加密传输的网站配置文件中添加一句：

```
UseHttps=true
```

如果使用的证书是自己制作的，可以加入下面的代码来解决。

```
ServicePointManager.ServerCertificateValidationCallback = new RemoteCertificateValidationCallback((object obj, X509Certificate x1, X509Chain x2, SslPolicyErrors sp) => {
    return true;
});
```

安全保护策略

Jexus web server 会主动对访问者提交的请求数据进行安全检测，即“入侵检测”，当它发现用户具有“SQL 注入”之类的恶意请求现象时就会自动终止向这个用户提供服务，并将它所发现的不安全访问情况写入日志以备进行更深入的处理。因此，从主动防卫而言，Jexus 较之其它 web 服务器具有更强的安全保护能力。

虽然 Jexus 能够主动防止很多外来的恶意访问，但是，Jexus 及其对应网站中的 ASP.NET 程序（.aspx/.dll/.cs/.asmx 等）在默认情况下是以 root 权限运行的，因而它们有能力对整过服务器的文件进行任意的操作。高权限是一把“双刀剑”，有利的一面是，我们因此可以利用服务器全部资源而写出功能极为强劲的 ASP.NET 网站程序，不利的一面是，如果这种强有力的权限一旦被人非法利用，那么这台服务器就没有安全性可言了。因此，当 Jexus 在默认配置的状态下工作时，我们强烈建议：

- 1、一定要对网站的 ASP.NET 程序（网页）的读写功能，进程操作功能进行限制和认真检查，一定要了解每个网站程序有些什么功能，可能会产生哪些不安全后果；

- 2、要严格禁止用户向网站上传 ASP.NET 程序或 perl 等脚本程序，比如.aspx、.dll、.cs、.pl、.sh 等等；

- 3、即使允许上传其它的在 ASP.NET 网站上不可执行的文件，也要严格限制用户对文件扩展名进行修改的权力，绝不允许用户把上传的文件更名为.aspx/dll/cs 等等 ASP.NET 可执行文件；

- 4、如果 asp.net 程序必须具有让用户上传文件的功能，那么，接受文件时一定要对文件扩展名进行严格的限制，保存时给文件更名，然后把文件存放到网站文件夹之外，或者把文件数据放到数据库中去；

- 5、禁止用 linux 服务器向用户提供 ASP.NET 虚拟空间，或者说禁止任何安全性不可信任的文件出现在 WEB 服务器上。

- 6、同时，对 Linux 服务器的其它功能的安全性也要认真限制，比如 FTP 服务器、SSH 服务器等等，努力提升服务器的整体安全性。

- 7、设置工作进程的用户身份：要想让网站更加安全，最重要方法就是给 Jexus 加个“金钟罩”：让 Jexus 以非 root 身份运行，Jexus 的主控进程任何时间都是 Root 权，而 listener 就在它上边，所以，不存在端口问题。用非 root，影响的是“工作进程”，当工作进程用非 root 权限运行时，工作进程就会因权限不足而被限制不能访问 linux 关键的文件，从而提高安全性。

步骤一：建立一个非 root 的普通用户

- 1、建立普通用户，比如：www-data

```
useradd www-data -d /home/www -s /sbin/nologin
```

（注：如果系统中已经有这个用户了，就不用再建）

2、按上面的指令内容，为这个用户建一个目标文件夹“/home/www”，再在/home/www中新建一个文件夹，名为“.wapi”

3、让“.wapi”文件夹的所有者为 www-data

chown www-data:www-data /home/www/.wapi

步骤二：修改 Jexus 操作脚本“jws”文件中的启动代码，使 Jexus 以 www-data 之类的身份工作

步骤三：让 Jexus 对日志文件和 PID 文件具有写入权。

Jexus 运行时，会自动产生两类文件，一是按用户设置的路径和文件名产生的日志文件，二是进程 ID 文件(在/tmp 中)，Jexus 必须对这些文件或文件夹拥有写入权，否则就无法启动。我们建议给 Jexus 设一个专用的 JEXUS 具有写入权的日志文件目录，把所有的日志都放在这里以便操作管理。

通过以上设置，网站中的 ASP.NET 程序就只具有 www-data 的权限了，这个权限非常小，即使“恶意访问者”有效突破了 Jexus 本身的检测防护关口而把恶意程序上传到网站中，也会因为没有相应权限而无法发挥其破坏作用。我们可以说，拥有“内外兼修”安全保护机制（即外围的权限设置和它本身的安全检测与防护能力）的 Jexus，其安全强度是非常高的。

让 Jexus 支持高并发请求的优化技巧

Jexus web server 5.4 每个工作进程的最大并发数固定为 1 万，最多可以同时开启 8 个工作进程，因此，每台 Jexus V5.4 服务器最多可以到支持 8 万个并发连接。但是，按照 linux 系统的默认设置，linux 是不能支持这么高的并发请求的，只有对 linux 进行一些必要的优化，才能达到让 Jexus 支持大并发的目的。

一、调整文件描述符数量限制

linux 默认文件描述符只有 1024 个，对于 Jexus 等一些服务来说，在大负载的情况下这点文件描述符是远远不够的，因为 Jexus 的工作方式，文件描述符的限制可能会极大的影响性能。当 Jexus 用完所有的文件描述符后，它不能接收用户新的连接。也就是说，用完文件描述符导致拒绝服务。直到一部分当前请求完成，相应的文件和 socket 被关闭，Jexus 不能接收新请求，这样就要扩大 linux 的文件描述符了。

编辑 /etc/security/limits.conf，更改或添加如下内容：

```
* soft nfile 20000
* hard nfile 20000
* root soft  nfile 20000
* root hard  nfile 20000
```

因为，有的 linux 会把“*”理解成非 root 之外的所有帐户，结果就把 root 排除在外了。而 jws 主进程始终是 root 控制的。

limits.conf 文件实际是 Linux PAM（插入式认证模块，Pluggable Authentication Modules）中 pam_limits.so 的配置文件，而且只针对于单个会话。

limits.conf 的格式如下：

username|@groupname type resource limit

username|@groupname：设置需要被限制的用户名，组名前面加@和用户名区别。也可以用通配符*来做所有用户的限制。

type: 有 soft, hard 和 -, soft 指的是当前系统生效的设置值。hard 表明系统中所能设置的最大值。soft 的限制不能比 hard 限制高。用 - 就表明同时设置了 soft 和 hard 的值。

resource:

core - 限制内核文件的大小

date - 最大数据大小

fsize - 最大文件大小

memlock - 最大锁定内存地址空间

nofile - 打开文件的最大数目

rss - 最大持久设置大小

stack - 最大栈大小

cpu - 以分钟为单位的最多 CPU 时间

noproc - 进程的最大数目

as - 地址空间限制

maxlogins - 此用户允许登录的最大数目

要使 limits.conf 文件配置生效,必须要确保 pam_limits.so 文件被加入到启动文件中。查看 /etc/pam.d/login 文件中有:

session required /lib/security/pam_limits.so

二、调整网络参数

Sysctl 是一个允许您改变正在运行中的 Linux 系统的接口。它包含一些 TCP/IP 堆栈和虚拟内存系统的高级选项, 这可以让有经验的管理员提高系统性能。用 sysctl 可以读取设置超过五百个系统变量。

查看所有可读变量:

```
% sysctl -a
```

读一个指定的变量, 例如 kern.maxproc:

```
% sysctl kern.maxproc kern.maxproc: 1044
```

要设置一个指定的变量, 直接用 variable=value 这样的语法:

```
# sysctl kern.maxfiles=5000
```

```
kern.maxfiles: 2088 -> 5000
```

您可以使用 sysctl 修改系统变量, 也可以通过编辑 sysctl.conf 文件来修改系统变量。sysctl.conf 看起来很像 rc.conf。它用 variable=value 的形式来设置值。指定的值在系统进入多用户模式之后被设置。并不是所有的变量都可以在这个模式下设置。

sysctl 变量的设置通常是字符串、数字或者布尔型。(布尔型用 1 来表示 'yes', 用 0 来表示 'no')。

编辑 “/etc/sysctl.conf”, 更改或添加如下内容:

```
net.core.somaxconn=8192
```

```
net.ipv4.tcp_syncookies=1
```

```
net.ipv4.tcp_tw_reuse=1
```

```
net.ipv4.tcp_tw_recycle=1
```

```
net.ipv4.tcp_fin_timeout=30
```

```
net.ipv4.tcp_keepalive_time=1200
```

```
net.ipv4.ip_local_port_range = 1024 65000
```

```
net.ipv4.tcp_max_tw_buckets = 5000
```

```
net.ipv4.tcp_max_syn_backlog=8192
```

```
net.ipv4.tcp_max_tw_buckets=10000
```

如果启用了 iptables 防火墙并加载了 ip_conntrack 模块，还需加入：

```
net.ipv4.ip_conntrack_max = 10240
```

注：ubuntu 是 “net.ipv4.netfilter.ip_conntrack_max”。

上述参数修改完成后，请用 “sysctl -p” 命令使其生效。

几个解释：

```
net.ipv4.tcp_syncookies = 1
```

#表示开启 SYN Cookies。当出现 SYN 等待队列溢出时，启用 cookies 来处理，可防范少量 SYN 攻击，默认为 0，表示关闭；

```
net.ipv4.tcp_tw_reuse = 1
```

#表示开启重用。允许将 TIME-WAIT sockets 重新用于新的 TCP 连接，默认为 0，表示关闭；

```
net.ipv4.tcp_tw_recycle = 1
```

#表示开启 TCP 连接中 TIME-WAIT sockets 的快速回收，默认为 0，表示关闭。

```
net.ipv4.tcp_fin_timeout = 30
```

#表示如果套接字由本端要求关闭，这个参数决定了它保持在 FIN-WAIT-2 状态的时间。

```
net.ipv4.tcp_keepalive_time = 1200
```

#表示当 keepalive 起用的时候，TCP 发送 keepalive 消息的频度。缺省是 2 小时，改为 20 分钟。

```
net.ipv4.ip_local_port_range = 1024 65000
```

#表示用于向外连接的端口范围。缺省情况下很小：32768 到 61000，改为 1024 到 65000。

```
net.ipv4.tcp_max_tw_buckets = 10000
```

#表示系统同时保持 TIME_WAIT 套接字的最大数量，如果超过这个数字，

#TIME_WAIT 套接字将立刻被清除并打印警告信息。默认为 180000，改为 10000。

#对于 Apache、Nginx、Jexus 等服务器，上几行的参数可以很好地减少 TIME_WAIT 套接字数量

三、整调 Jexus 配置参数

Jexus 默认工作进程数为 1，为了支持更大的并发数量，应根据服务器 CPU 内核数量及内存大小，合理调整工作进程数量。方法是，编辑 jws.conf，去掉

“httpd.processes” 项前边的 “#” 号，把进程数填写到等号右边（Jexus v5.4 版最大值不超过 8）。

注：

1) 进程数与 cpu 的总核数有关，建议 8 核以内设为 2，16 核设为 3，更多内核数就设为 4。

2) 进程数还与服务内存及 WEB 应用所占内存的多少有关，建议每个工作进程平均内存可用量不低于 256M，服务器总内存不低于 512M。

3) jexus 5.4.8 开始，可以自定义 jexus 工作进程对内存和 cpu 资料的消耗量了，超过限制，jexus 就会如 IIS 的进程池那么 “回收”。

jws.conf 中 httpd.maxTotalMemory=xxxx, 限制所有工作进程的内存使用总量, 单位 M, 任何工作进程超过这个总量的均值, 就会被重启。

httpd.maxCpuTime=yyyy, 限掉工作进程使用 cpu 的时间, 单位秒, 超过这个数, 这个工作进程就会被回收。

有了这两个参数, 对服务质量要求很高的用户, 就可以根据自己服务器硬件情况和应用大小、访问量大小等, 对 Jexus 回收工作进程的时间间隔作更精确的控制。

四、mono 版本选择

建议使用 mono 最新的成熟版本。

提示 server busy

JEXUS 会检测你的网站的处理速度, 三种情况会返回服务忙, 一是并发连接超过每进程 1 万个, 二是你的 asp.net 处理时间超过 9 秒, 三是, ASP.NET 队列长度超过 2500 个。

配置 PHP

Jexus 不仅支持 ASP.NET, 而且能够通个自带的 PHP-FCGI 服务以及 PHP-FPM 等方式灵活支持 PHP 而且还可以以 .NET (Phalanger) 方式支持 PHP。

PHP-FCGI 服务支持 PHP

1、安装 PHP-CGI:

```
[azureuser@mono ~]$ sudo yum -y install php-cgi
```

2、配置:

1) 修改 “/etc/php.ini” 文件:

找到 cgi.force_redirect=1 一行, 把前边的 “#” 号去掉, 把值从 1 改为 0, 如:

```
cgi.force_redirect=0
```

2) 修改 jws.conf。打开 jexus 文件夹中的 jws.conf, 作如下配置:

填写 PHP-CGI 程序路径和工作进程数。如:

“php-fcgi.set=/usr/bin/php-cgi,6”。

3) 修改网站配置。在需要使用 PHP 的网站的配置文件中添加:

```
fastcgi.add=php|socket:/var/run/jexus/phpsvr
```

1、以管理员身份重启 jexus。

在网站目录下创建一个 phpinfo 的页面 index.php

PHP Version 5.3.3



System	Linux mono 2.6.32-279.14.1.el6.openlogic.x86_64 #1 SMP Wed Dec 12 18:33:43 UTC 2012 x86_64
Build Date	Jul 3 2012 16:50:56
Configure Command	'./configure' '--build=x86_64-redhat-linux-gnu' '--host=x86_64-redhat-linux-gnu' '--target=x86_64-redhat-linux-gnu' '--program-prefix=' '--prefix=/usr' '--exec-prefix=/usr' '--bindir=/usr/bin' '--sbindir=/usr/sbin' '--sysconfdir=/etc' '--datadir=/usr/share' '--includedir=/usr/include' '--libdir=/usr/lib64' '--libexecdir=/usr/libexec' '--localstatedir=/var' '--sharedstatedir=/var/lib' '--mandir=/usr/share/man' '--infodir=/usr/share/info' '--cache-file=../config.cache' '--with-libdir=lib64' '--with-config-file-path=/etc' '--with-config-file-scan-dir=/etc/php.d' '--disable-debug' '--with-pic' '--disable-rpath' '--without-pear' '--with-bz2' '--with-exec-dir=/usr/bin' '--with-freetype-dir=/usr' '--with-png-dir=/usr' '--with-xpm-dir=/usr' '--enable-gd-native-ttf' '--without-gdgm' '--with-gettext' '--with-gmp' '--with-iconv' '--with-jpeg-dir=/usr' '--with-openssl' '--with-pcre-regex=/usr' '--with-zlib' '--with-layout=GNU' '--enable-exif' '--enable-ftp' '--enable-magic-quotes' '--enable-sockets' '--enable-sysvsem' '--enable-sysvshm' '--enable-sysvmsg' '--with-kerberos' '--enable-ucd-snmp-hack' '--enable-shmop' '--enable-calendar' '--without-sqlite' '--with-libxml-dir=/usr' '--enable-xml' '--with-system-tzdata' '--enable-force-cgi-redirect' '--enable-pcntl' '--with-imap=shared' '--with-imap-ssl' '--enable-mbstring=shared' '--enable-mbregex' '--with-gd=shared' '--enable-bcmath=shared' '--enable-dba=shared' '--with-db4=/usr' '--with-xmllrpc=shared' '--with-ldap=shared' '--with-ldap-sasl' '--with-mysql=shared,/usr' '--with-mysqli=shared,/usr/lib64/mysql/mysql_config' '--enable-dom=shared' '--with-pgsql=shared' '--enable-wddx=shared' '--with-snmp=shared,/usr' '--enable-soap=shared' '--with-xsl=shared,/usr' '--enable-xmlreader=shared' '--enable-xmlwriter=shared' '--with-curl=shared,/usr' '--enable-fastcgi' '--enable-pdo=shared' '--with-pdo-odbc=shared,unixODBC,/usr' '--with-pdo-mysql=shared,/usr/lib64/mysql/mysql_config' '--with-pdo-pgsql=shared,/usr' '--with-pdo-sqlite=shared,/usr' '--with-sqlite3=shared,/usr' '--enable-json=shared' '--enable-zip=shared' '--without-readline' '--with-libedit' '--with-pspell=shared' '--enable-phar=shared' '--with-tidy=shared,/usr' '--enable-sysvmsg=shared' '--enable-sysvshm=shared' '--enable-sysvsem=shared' '--enable-posix=shared' '--with-unixODBC=shared,/usr' '--enable-fileinfo=shared' '--enable-intl=shared' '--with-icu-dir=/usr' '--with- enchant=shared,/usr' '--with-recode=shared,/usr'
Server API	CGI/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc
Loaded	/etc/php.ini

PHP-FPM 服务支持 PHP

1、安装:

PHP-FPM 之前,你必须卸载系统中以前安装的 Apache 和 PHP。默认情况下,CentOS 的官方资源是没有 php-fpm 的,但我们可以从 Remi 的 RPM 资源中获得,它依赖于 EPEL 资源。我们可以这样增加两个资源库:

```
[azureuser@mono ~]$ sudo yum install yum-priorities -y
```

```
[azureuser@mono ~]$
```

```
sudo rpm -Uvh
```

```
http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm
```

```
[azureuser@mono ~]$
```

```
sudo rpm -Uvh
```

```
http://rpms.famillecollet.com/enterprise/remi-release-6.rpm
```

```
[azureuser@mono ~]$yum --enablerepo=remi install php php-fpm
```

2、配置

1)修改 fpm 配置（可以不用改）：打开/etc/php-fpm.d/www.conf 文件，把 pm.max_children 等的值没为你需要的值。

2)修改网站配置文件，在需要运行 PHP 的网站配置文件中添加：

fastcgi.add=php|tcp:127.0.0.1:9000

3、启动 FPM 服务：

```
[azureuser@mono siteconf]$ sudo service php-fpm start
```

如果你想在系统启动时自动运行 php-fpm，输入下列命令：

```
[azureuser@mono siteconf]$ sudo chkconfig --level 345 php-fpm on
```

PHP 仅安装了核心模块，你很可能需要安装其他的模块，比如 MySQL、XML、GD 等等，你可以输入下列命令：

```
[azureuser@mono siteconf]$ sudo yum --enablerepo=remi install php-gd  
php-mysql php-mbstring php-xml php-mcrypt
```

4、以管理员身份重启 jexus。

在网站目录下创建一个 phpinfo 的页面 index.php：

PHP Version 5.4.20



System	Linux mono 2.6.32-279.14.1.el6.openlogic.x86_64 #1 SMP Wed Dec 12 18:33:43 UTC 2012 x86_64
Build Date	Sep 18 2013 19:57:12
Server API	FPM/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc
Loaded Configuration File	/etc/php.ini
Scan this dir for additional .ini files	/etc/php.d
Additional .ini files parsed	/etc/php.d/curl.ini, /etc/php.d/fileinfo.ini, /etc/php.d/json.ini, /etc/php.d/phar.ini, /etc/php.d/zip.ini
PHP API	20100412
PHP Extension	20100525
Zend Extension	220100525
Zend Extension Build	API220100525,NTS
PHP Extension Build	API20100525,NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	disabled

.NET(Phalanger)支持 PHP

Phalanger 简介

Phalanger 是一种 PHP 语言编译器，也是针对 .NET 的 PHP 运行时。它可以用于把 PHP web 项目编译成 .NET 字节码，并在 Windows 中使用 IIS 或者在 Linux 上使用 Mono 和 Apache 作为 ASP.NET 应用程序来执行。然而，Phalanger 不仅仅是把已经存在的 PHP 应用编译到 .NET 中。

我们可以使用 Phalanger 创建组合 .NET 和 PHP 的解决方案，所采用的方式用标准的 PHP 解释器是不可能做到的。有了 Phalanger 扩展，PHP 程序可以直接使用 .NET 类，而 .NET 程序（比方说用 C# 编写的）也可以动态地调用 PHP 脚本，或者使用在 PHP 中实现的函数和类。

Phalanger 在很大程度上与 PHP 5 兼容，可以运行大量开源的 PHP 项目，包括 WordPress 和 MediaWiki。我们可以使用它把这些项目集成到 .NET 生态系统中，也可以开发新的项目，它会兼有 PHP 和 .NET 的优势。

Phalanger 的组件

Phalanger 包括多个部分独立的组件，可以用来开发运行在 .NET 上的 PHP 应用程序，并使用 .NET 或 Mono 来运行它们：

- Phalanger 编译器 Phalanger 会把 PHP 源代码编译成 .NET 程序集，它可以使用 .NET JIT（Just-in-time 编译器，它会为当前平台生成本地代码）执行。编译后的 PHP 代码会使用 Phalanger 运行时和动态语言运行时，从而提供了 PHP 语言动态特性的高效率实现。
- Phalanger 运行时和类库 Phalanger 运行时提供了对数组之类 PHP 特性的实现。Phalanger 还包含了针对 I/O、正则表达式以及其他标准 PHP 类库的 .NET 实现。
- Phalanger 可以通过托管的 PHP 4 扩展。
- 托管的扩展 通过包装 .NET 中提供的类似功能，PHP 扩展也可以重新实现。这些扩展可以是由任何 .NET 语言编写，并提供很好的性能。Phalanger 中包含多个扩展，包括 SPL、JSON、SimpleXML、MySQL 和 MS SQL 的提供程序。DEVSENSE【9】还提供了附加的扩展，像 Memcached、图像和 cURL 等。
- 与 Visual Studio 的集成 Phalanger 还与 Visual Studio 集成（最近的更新支持 Visual Studio 2010）。集成功能添加了针对 PHP 文件的颜色突出显示和智能提示功能，让我们可以调试使用 Phalanger 运行的 PHP 应用程序。

Jexus 下运行 Phalanger

Jexus 可以同时用普通方式 (PHP-CGI/PHP-FPM) 以及 .NET (Phalanger) 方式支持 PHP，为了不引起混淆，用 .NET 方式支持 PHP 时，要注意如下事项：

1、网站配置文件中的 UsePHP 的值为 false，或者不用这一句。

- 2、网站配置文件中添加一行 ASPNET_Exts=php,说明 php 网页按 ASP.NET 处理,如果已经有这一项,就在末尾添一个 php (用英文逗号与已有扩展名分隔)。
- 3、Jexus 启用 .NET4 工作模式(在 jws.conf 中添一行“Runtime=v4.0.30319”)。
- 4、在网站的 web.config 添加 Phalanger 有关配置。

<http://www.infoq.com/cn/articles/Phalanger>

配置 Perl

一、安装 Perl:

```
[azureuser@mono siteconf]$ sudo yum -y install perl
```

二、下载并安装必要的模块

在 Jexus 网站下载 “Perl FastCGI 支持包”

<http://linuxdot.net/down/Perl-FCGI.rar> , 解压后,把所有文件上传到服务器的某个目录,比如 “/tmp” 中,然后进入这个文件夹进行下面的操作:

1、安装 perl-FCGI:

```
tar -zxf FCGI-0.74.tar.gz
cd FCGI-0.74
perl Makefile.PL
make
make install
```

2、安装 perl-IO-ALL:

```
tar zxf IO-All-0.39.tar.gz
cd IO-All-0.39
perl Makefile.PL
make
make install
```

3、安装 perl-cgi:

```
将 perl-fcgi.txt 移动到/usr/bin 中,并更名为 perl-fcgi:
mv perl-fcgi.txt /usr/bin/perl-fcgi
添加可执行权:
chmod +x /usr/bin/perl-fcgi
```

三、启动 perl-fcgi:

```
perl-fcgi -l /tmp/perl.log -pid /tmp/perl.pid -S /tmp/perl.sock
注:建议把这个命令写到 /etc/rc.local 中,以便开机自启动
```

四、修改网站配置,使 Jexus 支持 perl cgi:

```
fastcgi.add=cgi|socket:/tmp/perl.sock
注:修改后,需重启 Jexus
```

五、停止 perl-fcgi:

```
kill -9 `cat /tmp/perl.pid`
rm -rf /tmp/perl.*
```

Ngnix+ fastcgi_mono

通过 yum 安装 Nginx:

```
[root@Mono /]# yum -y install nginx
```

配置 Nginx 和 fastcgi

```
[root@Mono html]# vi /etc/nginx/nginx.conf
```

修改 Server 如下:

```
server {
    listen      84 default_server;
    listen      [::]:84 default_server;
    server_name _;
    root        /usr/share/nginx/html;

    # Load configuration files for the default server block.
    include /etc/nginx/default.d/*.conf;

    location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
        fastcgi_index Default.aspx;
        fastcgi_pass 127.0.0.1:9900;
        include /etc/nginx/fastcgi_params;
    }
}
```

打开/etc/nginx/fastcgi_params, 在最后面添加以下两句:

```
[root@Mono html]# vi /etc/nginx/fastcgi_params
```

ASP.NET fastcgi configuration

```
fastcgi_param PATH_INFO      "";
```

```
fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
```

启动 Nginx、fastcgi_server

启动 nginx

```
[root@Mono html]# systemctl start nginx
```

启动 fastcgi_server

```
[root@Mono html]# fastcgi-mono-server4  
/applications=/:/usr/share/nginx/html /socket=tcp:127.0.0.1:9900  
/printlog=True &
```

测试是否能够执行 aspx:

在/usr/share/nginx/html/目录下建一个 test.aspx 页面，内容为
<%= "Hello World from nginx fastcgi!" %>
通过 wget http://localhost:84/test.aspx 来下载该页面的内容。

相关链接

<http://www.cnblogs.com/keyindex/archive/2012/06/11/2536843.html>
<http://www.cnblogs.com/wander1129/archive/2011/12/16/mono.html>
<http://www.cnblogs.com/bboy/archive/2012/10/08/2714626.html>
<http://www.cnblogs.com/yexuan/archive/2012/12/27/centos-mono-nginx.html>
<http://www.mono-project.com/docs/web/fastcgi/nginx/>

Apache +Mod_mono

由于CentOS自带的SELinux的原因,使得Apache无法连接到mod-mono-server,所以我们现在还需要配置SELinux。当然,这里为了简单就仅仅禁用SELinux,有兴趣的可以自己创建安全策略,使mod-mono-server可以生效。这里修改/etc/sysconfig/selinux文件。将“SELINUX=enforcing”修改为

“SELINUX=permissive”或“SELINUX=disabled”,保存重启系统即可。

关闭CentOS 7 防火墙和SELinux

1. 运行、停止、禁用firewalld

启动: # systemctl start firewalld

查看状态: # systemctl status firewalld 或者 firewall-cmd --state


停止: # systemctl disable firewalld

禁用: # systemctl stop firewalld

2. 关闭SELINUX

在文件/etc/sysconfig/selinux 修改

SELINUXTYPE = targeted 为 SELINUX = permissive



```
root@Mono:/etc/httpd/conf.d

# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#     enforcing - SELinux security policy is enforced.
#     permissive - SELinux prints warnings instead of enforcing
#     disabled - No SELinux policy is loaded.
SELINUX=enforcing
# SELINUXTYPE= can take one of these two values:
#     targeted - Targeted processes are protected,
#     minimum - Modification of targeted policy. Only selected
#               tected.
#     mls - Multi Level Security protection.
SELINUX=permissive
SELINUXTYPE=targeted

~
~
~
~
~
~
"/etc/sysconfig/selinux" 15L, 566C
```

#setenforce 0

命令所操作相关文件: /etc/grub.conf, /etc/selinux/config

安装 Apache 和 mod_mono 模块

安装 Apache 和 mod_mono 模块, 会把依赖的 apache 服务安装好。

```
[root@Mono ~]# yum install -y apache2-mod_mono.x86_64
```

配置 apache 和 mod_mono

#支持 dotnet4 framework, 在此我们添加自己网站的配置文件, 可在此处生成
<http://go-mono.com/config-mod-mono/>

```
[root@Mono conf.d]# vi /etc/httpd/conf.d/mod_mono.conf
```

#mod_mono.conf 添加内容的:

```
<VirtualHost *:82>
```

```
    ServerName html
```

```
    ServerAdmin web-admin@modmono
```

```
    DocumentRoot /var/www/html
```

MonoServerPath can be changed to specify which version of ASP.NET is hosted

```
# mod-mono-server2 = ASP.NET 2.0 / mod-mono-server4 = ASP.NET 4.0
```

```

MonoServerPath html "/usr/bin/mod-mono-server4"

# To obtain line numbers in stack traces you need to do two things:
# 1) Enable Debug code generation in your page by using the Debug="true"
#    page directive, or by setting <compilation debug="true" /> in the
#    application's Web.config
# 2) Uncomment the MonoDebug true directive below to enable mod_mono
debugging
MonoDebug html true

# The MONO_IOMAP environment variable can be configured to provide
platform abstraction
# for file access in Linux. Valid values for MONO_IOMAP are:
#   case
#   drive
#   all
# Uncomment the line below to alter file access behavior for the
configured application
MonoSetEnv html MONO_IOMAP=all
#
# Additional environment variables can be set for this server instance
using
# the MonoSetEnv directive. MonoSetEnv takes a string of 'name=value'
pairs
# separated by semicolons. For instance, to enable platform
abstraction *and*
# use Mono's old regular expression interpreter (which is slower, but
has a
# shorter setup time), uncomment the line below instead:
# MonoSetEnv html MONO_IOMAP=all;MONO_OLD_RX=1

MonoApplications html "/*:/var/www/html"
<Location "/">
    Allow from all
    Order allow,deny
    MonoSetServerAlias html
    SetHandler mono
</Location>
</VirtualHost>

#重启 apache 服务
[root@Mono conf.d]# systemctl restart httpd
以上设置 web 默认主页的位置: /var/www/html

```

测试是否能够执行 aspx

在/var/www/html/目录下建一个 test.aspx 页面，内容为

```
<%= "Hello World from apache mod_mono!" %>
```

通过 `wget http://localhost/test.aspx` 来下载该页面的内容。

相关链接

<http://www.cnblogs.com/mayswind/p/3189724.html>

<http://www.cnblogs.com/alsw/p/3550309.html>

Mono 服务

Windows 服务的运行可以在没有用户干预的情况下，在后台运行，没有任何界面。通过 Windows 服务管理器进行管理。服务管理器也只能做些简单的操作：开始，暂停，继续，停止。Windows 服务的特点：在后台运行，没有用户交互，可以随 Windows 启动而启动。

Mono 下的 Windows 服务叫做 `mono-service`，`mono-service` 也就是 Unix/Linux 的后台服务，也叫做 `Daemon`，在 Linux 系统中就包含许多的 `Daemon`。判断 `Daemon` 最简单的方法就是从名称上看。

CentOS 6 下自动启动的服务都在 `/etc/rc.d/init.d/` 目录下，比如说 `mysql`。如果不想让一个服务自动运行，把 `/etc/rc.d/init.d/` 目录下的这个服务脚本移除掉就可以。

`Daemon` 可以操作的状态：

- `start` 启动服务，等价于服务脚本里的 `start` 命令
- `stop` 停止服务，等价于副外长脚本 `stop` 命令
- `restart` 关闭服务，然后重新启动，等价于脚本 `restart` 命令
- `reload` 使服务不重新启动而重读配置文件，等价与服务脚本的 `reload` 命令
- `status` 提供服务的当前状态，等价于服务脚本的 `status` 命令
- `condrestart` 如果服务锁定，则这个来关闭服务，然后再次启动，等价于 `condrestart` 命令

一个 `service` 通常包含一个可执行的文件和一个 `service` 控制脚本。作为 `service` 程序本身的可执行程序一般存储在 `/usr/bin` 下；作为控制 `service` 的脚本一般存储在 `/etc/rc.d/init.d` 下，且控制 `service` 的脚本的格式相对固定，至少支持 `start`，`stop`，`status` 参数。

CentOS 7 使用 `Systemd` 管理服务，`Systemd` 服务则是要写个 `Systemd Service` 脚本，放在在目录 `/etc/systemd/system`。

mono-service 运行 windows 服务

我们可以写个脚本把 Mono service 包装成 Daemon。用 mono-service 来运行 windows 的服务程序。

命令格式: mono-service [options] program.exe

-d: 此选项可指定服务的工作目录。默认为当前目录。

-l: lockfile 文件，默认值是创建在 /tmp 基于承载该服务程序名称的文件名。

-m: 在 syslog 中显示的名称。

-n: 此处指定要启动（如果程序里包含多个服务）的服务名称。默认值是运行第一个定义的服务。

--debug: 使用此选项可防止 mono-service 重定向标准输入和标准输出和防止发送到后台程序。相当于--no-daemon

--no-daemon: 相当于--debug

例: mono-service -l:/var/run/MyService-lock.pid MyService.exe （这个-l 参数一定要加上）

控制服务（这几种操作的区别请参考 **windows** 的使用方式，这里我就不做过多解释了）：

暂停: kill -USR1 `cat <lock file>`

继续: kill -USR2 `cat <lock file>`

停止: kill `cat <lock file>`

这里的不是单引号，是数字 1 左边的那个点号

下面我们就以 mono-service 运行 SuperSocket 为例介绍下 mono-service 运行服务的脚本，这个脚本来自 [在 ubuntu 下用 mono-service 运行 SuperSocket:](#)

```
#!/bin/sh
```

```
# control supersocket like windows service. copy this to your Working directory  
then ./supersocket {start|stop|restart}
```

```
#custom your servicename
```

```
SERVICENAME="SuperSocket"
```

```
SERVICE_PID=""
```

```
SERVICE_PATH="/root/SuperSocket/" #your Working directory
```

```
export PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin
```



```

if [ -f "/var/run/${SERVICENAME}.pid" ]; then
    SERVICE_PID=$(cat /var/run/${SERVICENAME}.pid)
fi

case "$1" in
    start)
        if [ -z "${SERVICE_PID}" ]; then
            mono-service -d:${SERVICE_PATH} -l:/var/run/${SERVICENAME}.pid
            -m:supersocket.log SuperSocket.SocketService.exe

            echo "${SERVICENAME} running"
        else
            echo "${SERVICENAME} is already running!"
        fi
        ;;
    stop)
        if [ -n "${SERVICE_PID}" ]; then
            kill ${SERVICE_PID}

            echo "${SERVICENAME} stop success !"
        else
            echo "${SERVICENAME} is not running!"
        fi
        ;;
    restart)
        $0 stop

        sleep 1

        $0 start
        ;;
    *)
        echo "usage: $0 {start|stop|restart}"
esac

```

exit 0

使用 Topshelf 创建 Windows 服务

开发一个 Windows 服务通常也比较简单，在开发的时候我们期望以命令行方式运行，想对 Windows 服务有更多的控制，就有一个 Windows 服务框架 TopShelf 可以满足。Topshelf 是一个开源的跨平台的宿主服务框架，支持 Windows 和 Mono，只需要几行代码就可以构建一个很方便使用的服务宿主。

引用安装

- 1、官网：<http://topshelf-project.com/> 这里面有详细的文档及下载
- 2、Topshelf 的代码托管在 <http://github.com/topshelf/Topshelf/downloads>，可以在这里下载到最新的代码。
- 3、新建一个项目，只需要引用 Topshelf.dll 即可，为了日志输出显示，建议也同时引用 Topshelf.Log4Net。程序安装命令
 - Install-Package Topshelf
 - Install-Package Topshelf.Log4Net

使用

官网文档给过来的例子非常简单，直接使用即可以跑起来，官网文档地址：<http://docs.topshelf-project.com/en/latest/configuration/quickstart.html>

```
public class TownCrier
{
    readonly Timer _timer;
    public TownCrier()
    {
        _timer = new Timer(1000) {AutoReset = true};
        _timer.Elapsed += (sender, EventArgs) => Console.WriteLine("It
is {0} and all is well", DateTime.Now);
    }
    public void Start() { _timer.Start(); }
    public void Stop() { _timer.Stop(); }
}

public class Program
{
    public static void Main()
    {
        HostFactory.Run(x =>
```

//1

```

    {
        x.Service<TownCrier>(s => //2
        {
            s.ConstructUsing(name=> new TownCrier()); //3
            s.WhenStarted(tc => tc.Start()); //4
            s.WhenStopped(tc => tc.Stop()); //5
        });
        x.RunAsLocalSystem(); //6

        x.SetDescription("Sample Topshelf Host"); //7
        x.SetDisplayName("Stuff"); //8
        x.SetServiceName("Stuff"); //9
    }); //10
}
}
HostFactory.Run(x => //1
{
    x.Service<TownCrier>(s => //2
    {
        s.ConstructUsing(name => new TownCrier()); //
配置一个完全定制的服务,对 Topshelf 没有依赖关系。常用的方式。
        //the start and stop methods for the service
        s.WhenStarted(tc => tc.Start()); //4
        s.WhenStopped(tc => tc.Stop()); //5
    });
    x.RunAsLocalSystem(); //
服务使用 NETWORK_SERVICE 内置帐户运行。身份标识,有好几种方式,如:
    x.RunAs("username", "password"); x.RunAsPrompt();
    x.RunAsNetworkService(); 等

```

```

        x.SetDescription("Sample Topshelf Host 服务的描述");
//安装服务后, 服务的描述
        x.SetDisplayName("Stuff 显示名称");
//显示名称
        x.SetServiceName("Stuff 服务名称");
//服务名称
    });

```

扩展说明

Topshelf Configuration 简单配置

官方文档, 对 HostFactory 里面的参数做了详细的说明:

http://docs.topshelf-project.com/en/latest/configuration/config_api.html, 下面只对一些常用的方法进行简单的解释:

我们将上面的程序代码改一下:

```
HostFactory.Run(x => //1
```

```

        {
            x.Service<TownCrier>(s =>                                     //2
            {
                s.ConstructUsing(name => new TownCrier());           //
配置一个完全定制的服务,对 Topshelf 没有依赖关系。常用的方式。
                //the start and stop methods for the service
                s.WhenStarted(tc => tc.Start());                     //4
                s.WhenStopped(tc => tc.Stop());                       //5
            });
            x.RunAsLocalSystem();                                     //
服务使用 NETWORK_SERVICE 内置帐户运行。身份标识,有好几种方式,如:
x.RunAs("username", "password"); x.RunAsPrompt();
x.RunAsNetworkService(); 等

            x.SetDescription("Sample Topshelf Host 服务的描述");
//安装服务后,服务的描述
            x.SetDisplayName("Stuff 显示名称");
//显示名称
            x.SetServiceName("Stuff 服务名称");
//服务名称
        });

```

Service Configuration 服务配置

Topshelf 的服务一般主要有两种使用模式。

一、简单模式。继承 ServiceControl 接口,实现该接口即可。

Simple Service

To configure a simple service, the easiest configuration method is available.

```

HostFactory.New(x =>
{
    x.Service<MyService>();
});

// Service implements the ServiceControl methods directly and has a default constructor
class MyService : ServiceControl
{

```

实例:

namespace TopshelfDemo

```

{
    public class TownCrier : ServiceControl
    {

```

```

private Timer _timer = null;
readonly ILog _log = LogManager.GetLogger(typeof(TownCrier));
public TownCrier()
{
    _timer = new Timer(1000) { AutoReset = true };
    _timer.Elapsed += (sender, eventArgs) =>
_log.Info(DateTime.Now);

}

public bool Start(HostControl hostControl)
{
    _log.Info("TopshelfDemo is Started");
    _timer.Start();
    return true;
}

public bool Stop(HostControl hostControl)
{
    throw new NotImplementedException();
}
}

class Program
{
    public static void Main(string[] args)
    {
        var logCfg = new
FileInfo(AppDomain.CurrentDomain.BaseDirectory + "log4net.config");
        XmlConfigurator.ConfigureAndWatch(logCfg);

        HostFactory.Run(x =>
        {
            x.Service<TownCrier>();
            x.RunAsLocalSystem();

            x.SetDescription("Sample Topshelf Host 服务的描述");
            x.SetDisplayName("Stuff 显示名称");
            x.SetServiceName("Stuff 服务名称");
        });
    }
}

```

二、常用模式。

实例:

```
namespace TopshelfDemo
{
    public class TownCrier
    {
        private Timer _timer = null;
        readonly ILog _log = LogManager.GetLogger(
                                                    typeof(TownCrier));

        public TownCrier()
        {
            _timer = new Timer(1000) { AutoReset = true };
            _timer.Elapsed += (sender, eventArgs) =>
                _log.Info(DateTime.Now);
        }

        public void Start() { _timer.Start(); }
        public void Stop() { _timer.Stop(); }
    }

    class Program
    {
        public static void Main(string[] args)
        {
            var logCfg = new
                FileInfo(AppDomain.CurrentDomain.BaseDirectory + "log4net.config");
            XmlConfigurator.ConfigureAndWatch(logCfg);

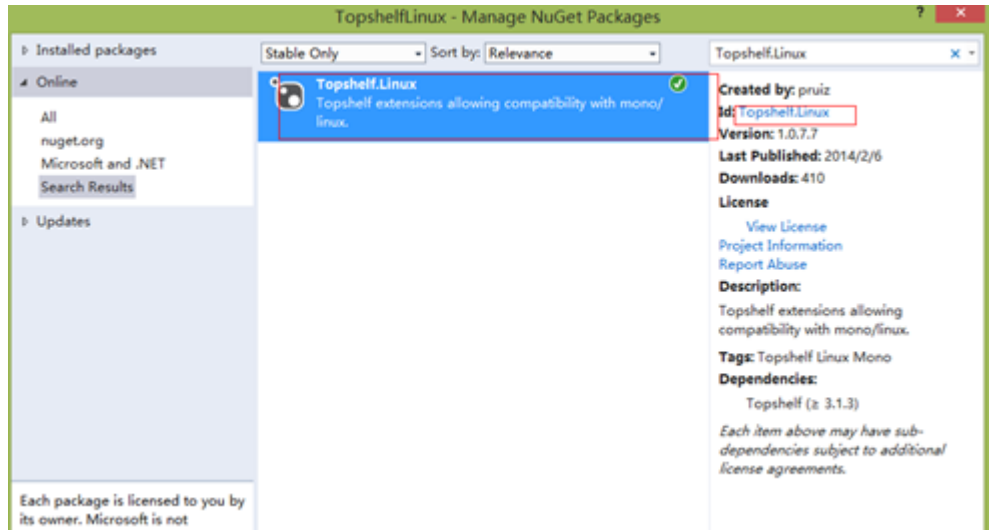
            HostFactory.Run(x =>
            {
                x.Service<TownCrier>(s =>
                {
                    s.ConstructUsing(name => new TownCrier());
                    s.WhenStarted(tc => tc.Start());
                    s.WhenStopped(tc => tc.Stop());
                });
                x.RunAsLocalSystem();

                x.SetDescription("Sample Topshelf Host 服务的描述");
                x.SetDisplayName("Stuff 显示名称");
                x.SetServiceName("Stuff 服务名称");
            });
        }
    }
}
```

Topshelf 支持 Mono 扩展 Topshelf.Linux

Topshelf 可以很好的支持 Windows 服务的开发，但是它和 Mono 不兼容，Github 上有一个扩展 <https://github.com/pruiz/Topshelf.Linux> 可以很好兼容 Linux/Mono，本文介绍使用方法：

1、在项目中添加 Topshelf.Linux， 通过 Nuget 很方便的添加引用：



2、在代码中加入下面一行代码 UseLinuxIfAvailable() :

```
class Program
{
    static ILog _log = LogManager.GetLogger(typeof(Program));

    static void Main(string[] args)
    {
```

```
        System.IO.Directory.SetCurrentDirectory(AppDomain.CurrentDomain.BaseDirectory);
```

```
        XmlConfigurator.ConfigureAndWatch(
            new FileInfo("log4net.config"));
```

```
        var host = HostFactory.New(x =>
        {
            x.Service<SampleService>(s =>
            {
                s.ConstructUsing(() => new SampleService());
                s.WhenStarted(v => v.Start());
                s.WhenStopped(v => v.Stop());
            });

            x.UseLinuxIfAvailable();
            x.RunAsLocalSystem();
            x.UseLog4Net();
        });
```

```

        x.SetDescription("SampleService Description");
        x.SetDisplayName("SampleService");
        x.SetServiceName("SampleService");
    });
    host.Run();
}

```

这样你的基于 Topshelf Windows 服务就完成了兼容 Mono 的改造工作。但是要注意的是在 Mono 下支持命令行运行,不能使用 Topshelf 的命令行 Start, Stop 控制服务等,这对于 Linux 环境来说足够了,可以通过 rc-scripts 来完成这些工作。

开发工具

Visual Studio 2015

Xamarin Studio

Xamarin 是基于 Mono 的平台,目前主要有以下产品(更具体请见:
<http://xamarin.com/products>):

- Xamarin Studio: IDE, 是从原来的 MonoDevelop 改名而来。现在从 MonoDevelop 官方网站下载的其实也是 Xamarin Studio: <http://monodevelop.com/>。(话说 MonoDevelop 也是 SharpDevelop 的一个分支发展而来)
- Xamarin.iOS: 原名 MonoTouch, 用于开发 iOS 应用程序, 并且可以发布到 app store 上。
- Xamarin.Mac: 用于开发 mac os x 应用程序, 类似于 windows 桌面应用。
- Xamarin.Android: 原名 MonoDroid/Mono for Android, 用于开发 Android 应用程序。
- Xamarin for Visual Studio: Visual Studio 的插件, 包括 iOS 和 Android, 不过目前只支持 vs2010/vs2012/Vs2013。
- Xamarin Test Cloud: 测试云, 可以把你的应用程序发布到 Xamarin 的云上面测试, 它可以自动帮你在数百种设备上测试你的应用程序。
- Component Store: 组件商店, 上面有各种收费/免费的控件提供下载。

Xamarin Studio 可以直接从 <http://www.monodevelop.com/download/> 下载, 目前最新的稳定版为 5.9.2。

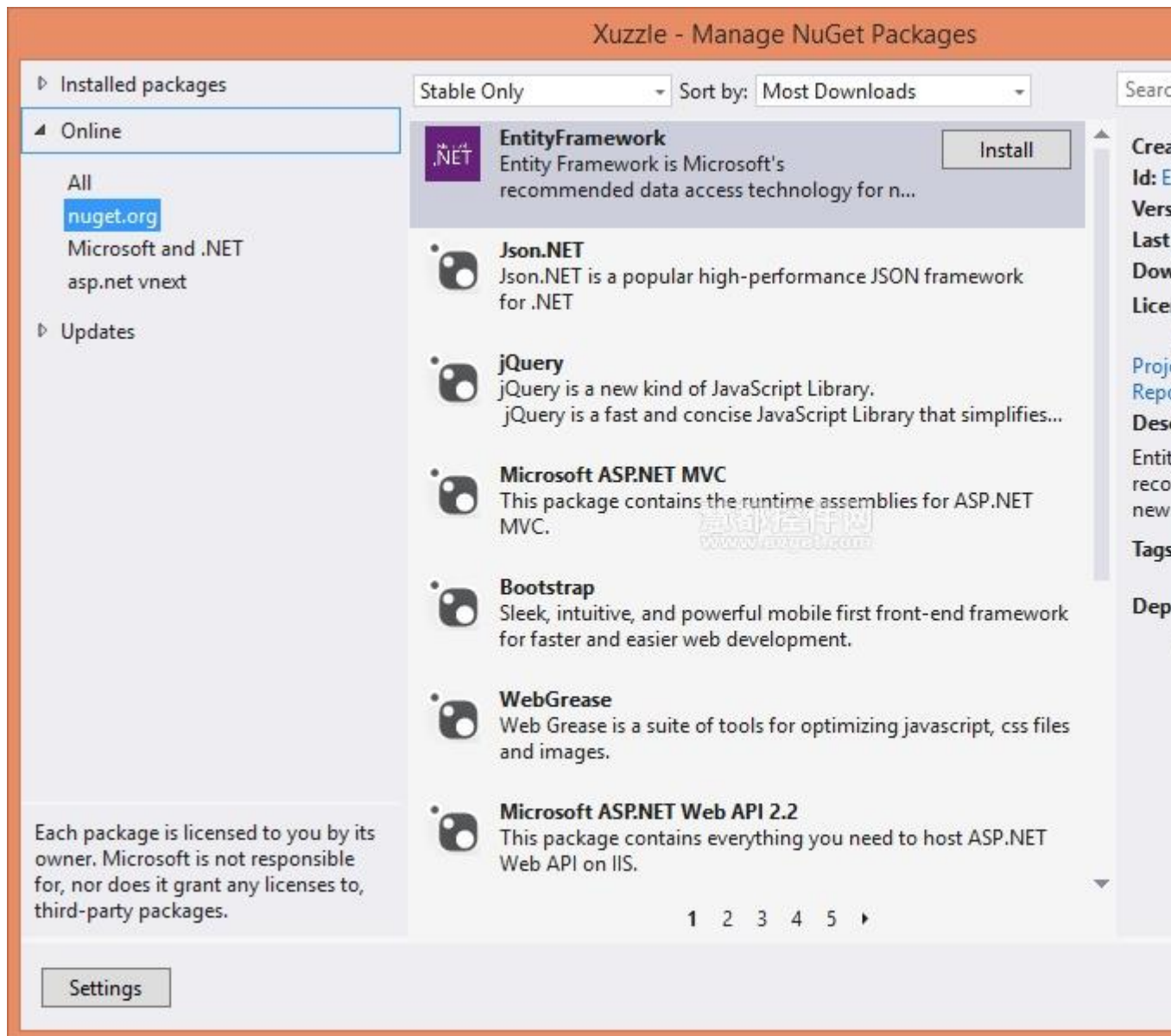
- Windows 版依赖于 .NET Framework 4.5 和 GTK# for .NET 2.12.25, 安装好两个依赖之后可以直接安装 Xamarin Studio 即可。

- Mac OSX 版依赖于 Mono + GTK#, 需要先从 <http://www.go-mono.com/mono-downloads/download.html> 下载 MRE (Mono Runtime) 或者 MDK (Mono DevelopmentKit) 安装。MRE 和 MDK 都包含 GTK# 和 MONO。

Xamarin Studio 5.2 版本, 带来了一些相当棒的特性, 其中有一些特性甚至超越了 Visual Studio 2013, 接下来就介绍我认为最棒的并且比 VS2013 要好用的三个功能:

NuGet 包管理

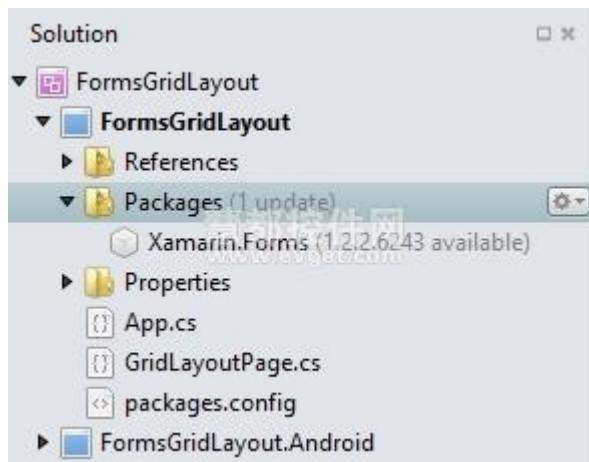
Visual Studio 对 NuGet 包的管理是通过对话框来管理的, 如下图所示:



对 NuGet 包的添加、更新、删除等操作都在这个对话框内完成, 不过缺点也是每次都得打开这个对话框。Xamarin Studio 提供一些更加人性化的管理方式, 一部分功能可以通过上右键下文菜单来管理, 不需要打开包管理对话框。

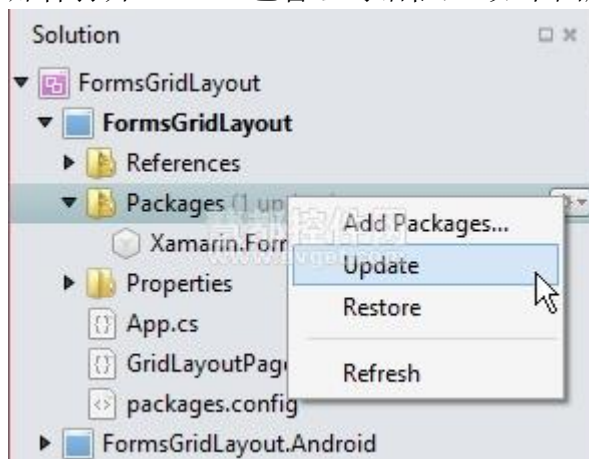
自动检查更新

打开一个带有 NuGet 包引用的项目时，Xamarin Studio 会自动检查项目引用的包有没有新版本，如果有新版本，则会在对应的节点上提示新版本，如下图所示：



一键更新还原

当引用的 nuget 包有新版本时，可以直接在包的节点上点击右键，选择更新，或者在包目录上右键，选择更新全部有新版本的包，而不必像 Visual Studio 那样打开 NuGet 包管理对话框，如下图所示：



如果需要的包需不是最新版本的，只要修改一下 packages.config 文件里的对应包的版本号，同样右键选择“还原 (Restore)”即可。

这一点与 Visual Studio 比起来还是方便很多的，得打开包管理控制台，输入这样一条命令才行：

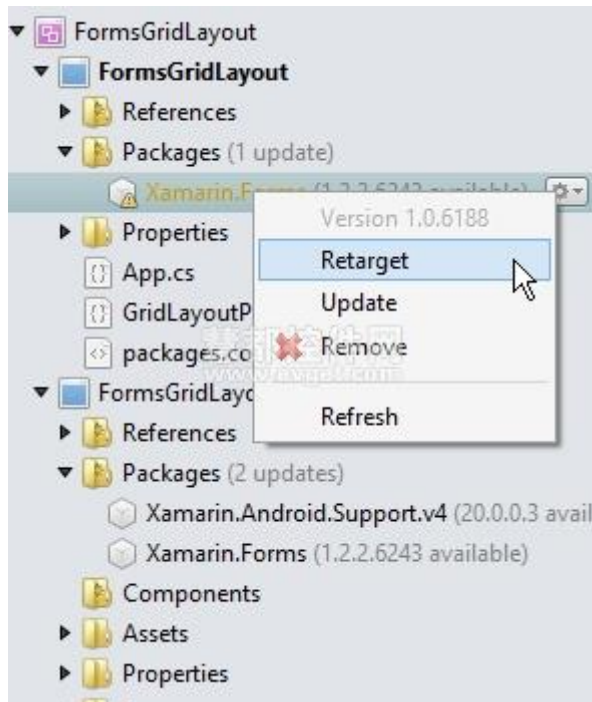
```
IPM> Install-Package package-id -Version package-version
```

重新指定目标

如果你在项目中使用了便携式类库项目 (PCL Library Project)，调整了类库的目标之后，对应的 nuget 包也要重新添加，在 Visual Studio 中，是通过删除重新添加来实现的，或者输入命令：

```
update-package -reinstall
```

但是在 Xamarin Studio 中，只需要点击一下右键，选择“ReTarget”即可，如下图所示：

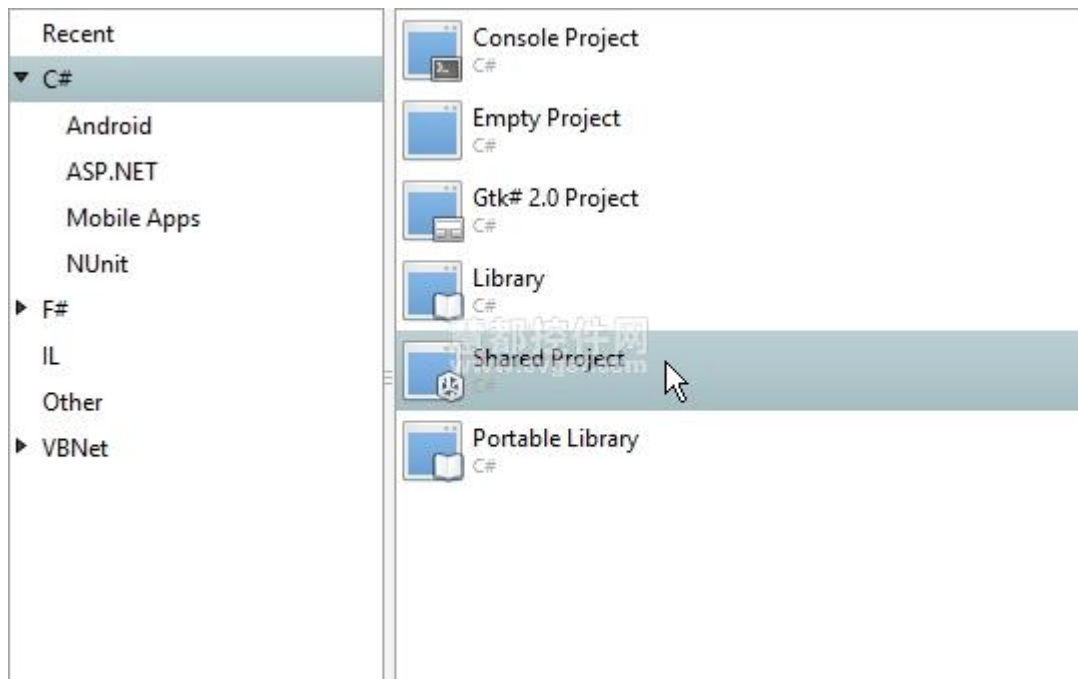


Shared Project 项目

Visual Studio 中的 Shared Project 让多项目中的文件链接成为历史，但是只支持 WinPhone 和 WinStore 两种项目类型，其它项目类型（Web, Library, PCL, Silverlight, WPF...）都不支持，真是让人不爽，不是不能支持，只是不让你用而已，其实就是一句 MSBuild 指令而已，手工编辑一下项目文件就行。

创建 Shared Project

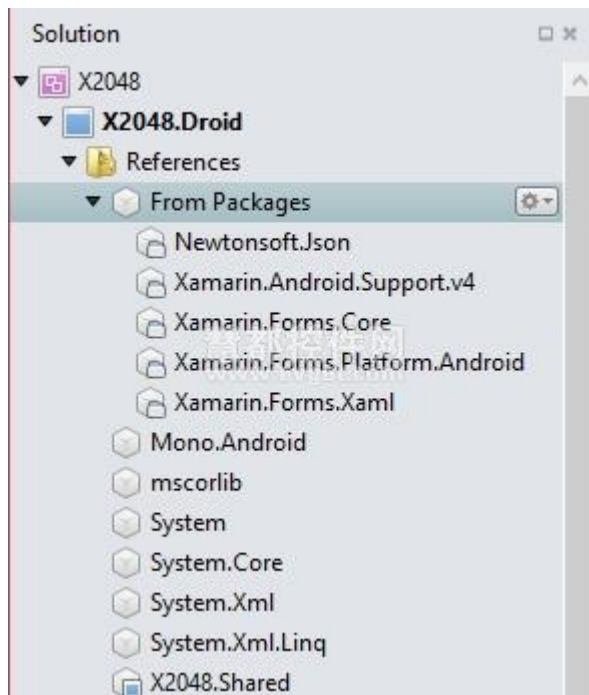
然而，Xamarin Studio 则提供了比较广泛的 Shared Project 项目支持，可以单独创建 Shared Project，所有项目类型都可以引用 Shared Project，如下图所示：



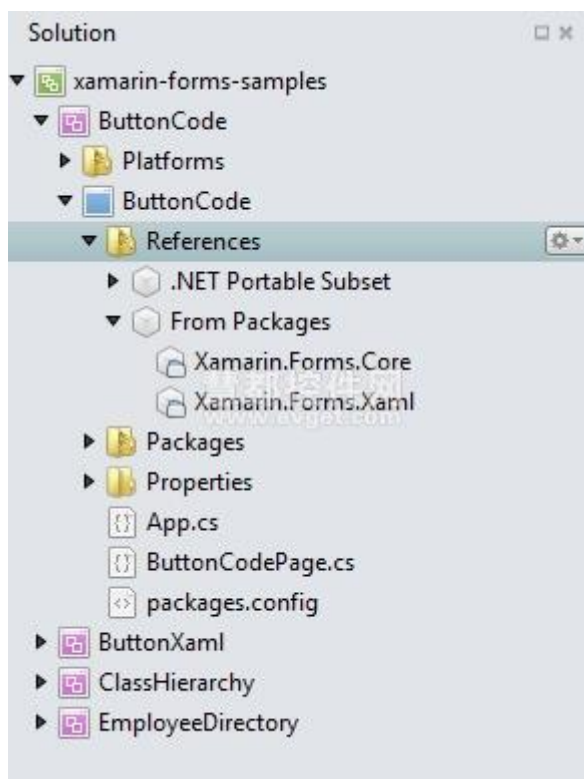
这一点还是非常赞的。

项目引用

Xamarin Studio 还有一点比较好的是对项目引用的分类，来自包的引用单独作为一组，这样看起来更加清晰：



特别是对于便携式类库项目，这样的分类看起来非常清楚：



以上三个功能是比较 Visual Studio 做的要好的三个特性，当然 Xamarin Studio 还有很多很好的特性，就不再列举了，希望这个开源的 IDE 能越来越好用！

<http://neutrofoton.com/tag/xamarin-studio/>

Visual Studio Code

Visual Studio Code 是一个跨平台，支持 30 多种语言的开箱代码编辑器。不管你是 .Net、Java、PHP、Python、TypeScript、Objective-C... 还是前端开发者，你都值得拥有。下面，让我们来看一看 Visual Studio Code 这个神器吧~

一、Visual Studio Code 的下载和安装

Visual Studio Code 最新版下载地址，<https://www.visualstudio.com/>，文件不到 60M，如图：



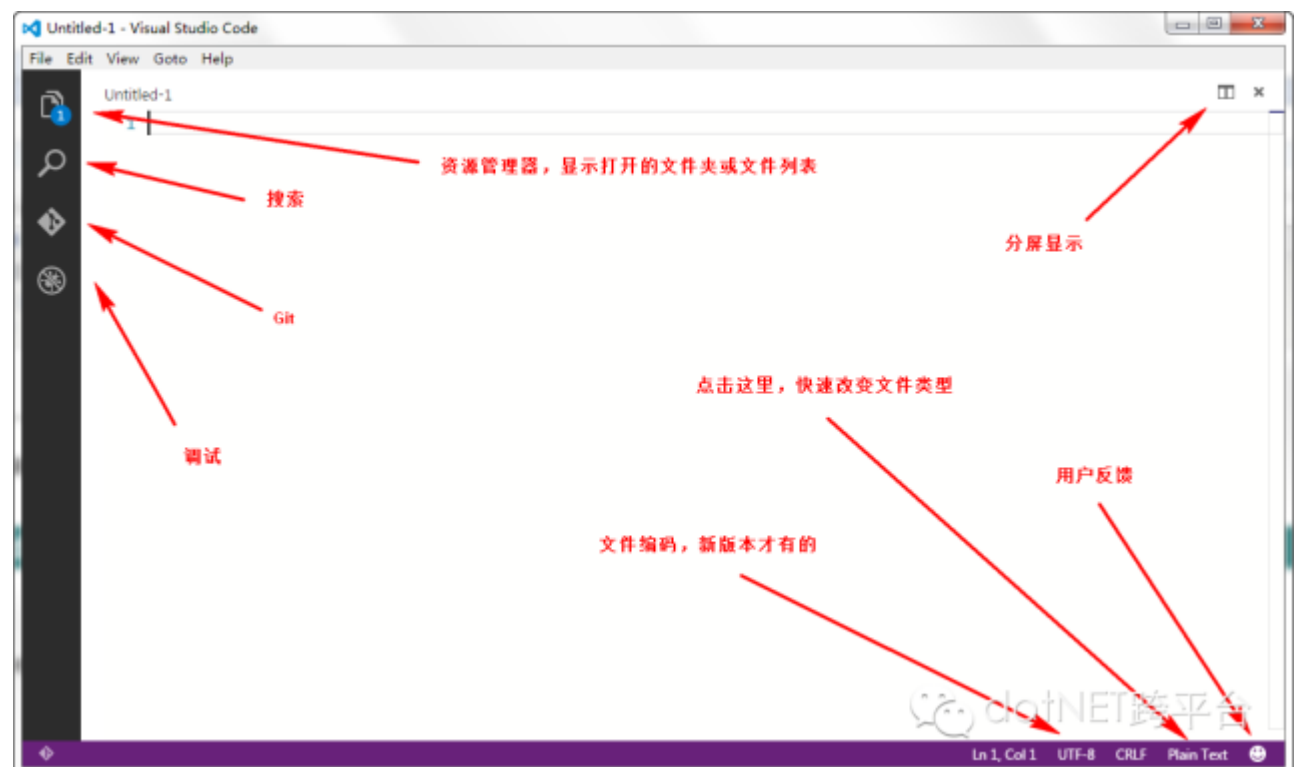
点击右边的下载，大家根据自己的系统下载对应的版本就好了。下载后，windows 双击 exe 文件会自动安装，其他系统我想大家都会的。下载安装完成后，我们打开 Visual Studio Code，点击 help > about, 会看到如下窗口：



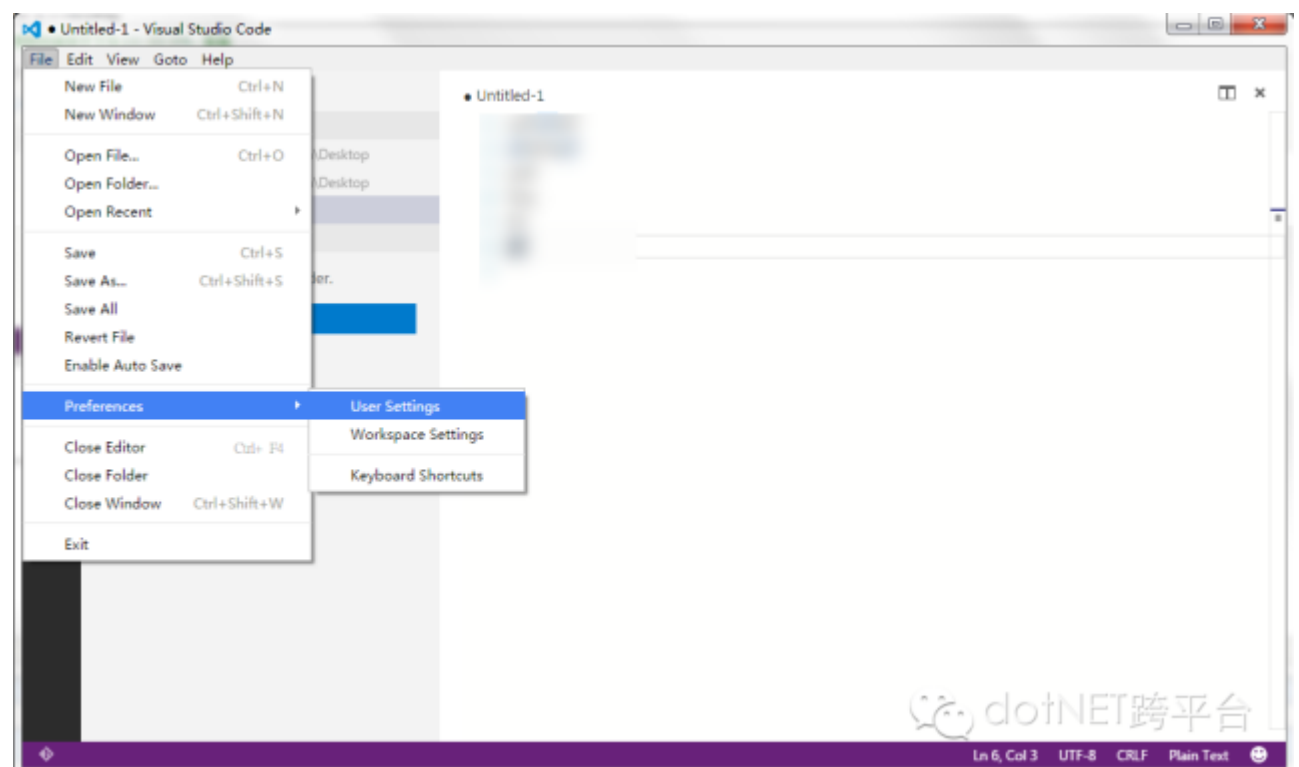
到这里，就说明你已经安装的是最新版本的 Visual Studio Code 了。

二、Visual Studio Code 简介

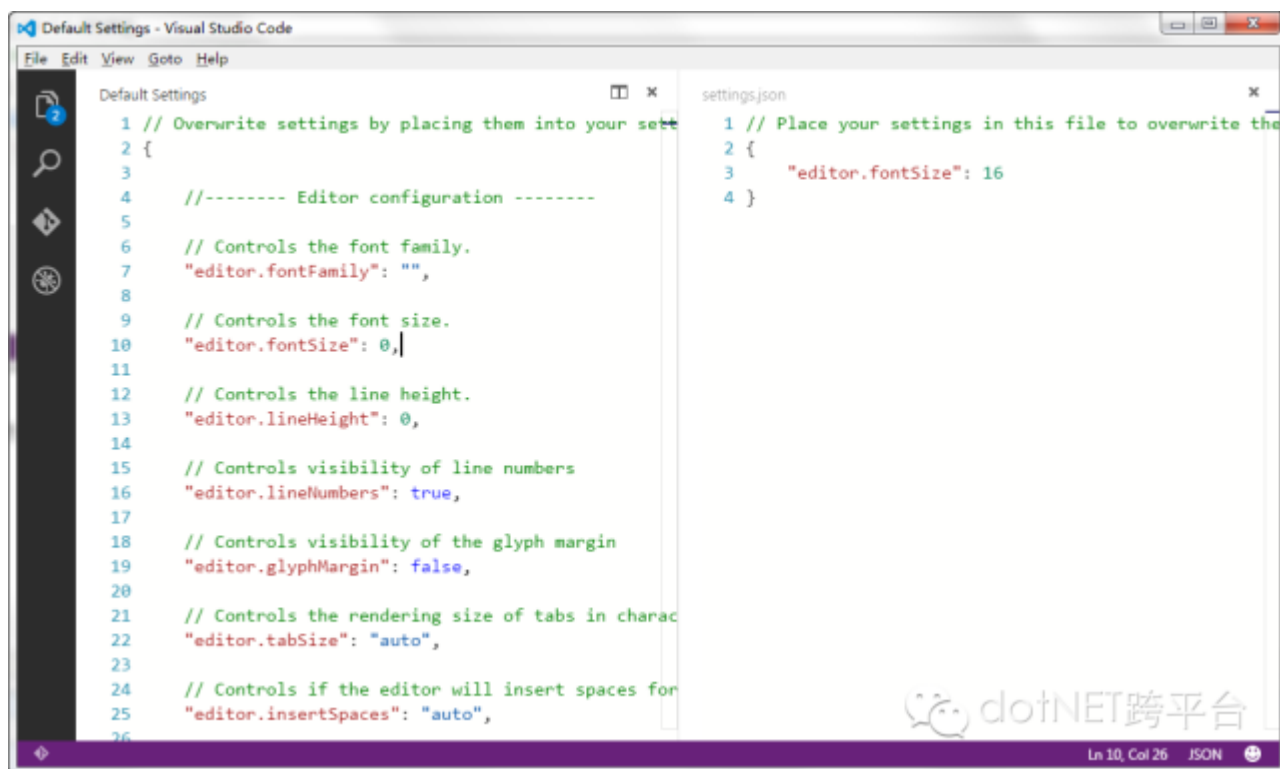
安装完成后，我们看到的 Visual Studio Code 界面如下，当然不同的系统界面边框略有不同，基本布局如图：



下面我们来进行一些简单的设置，下面就拿大家最关心的字体大小设置来说吧，其他的都类似，点击如图菜单：



我们会看到如下页面，VS Code 会打开两个文件：

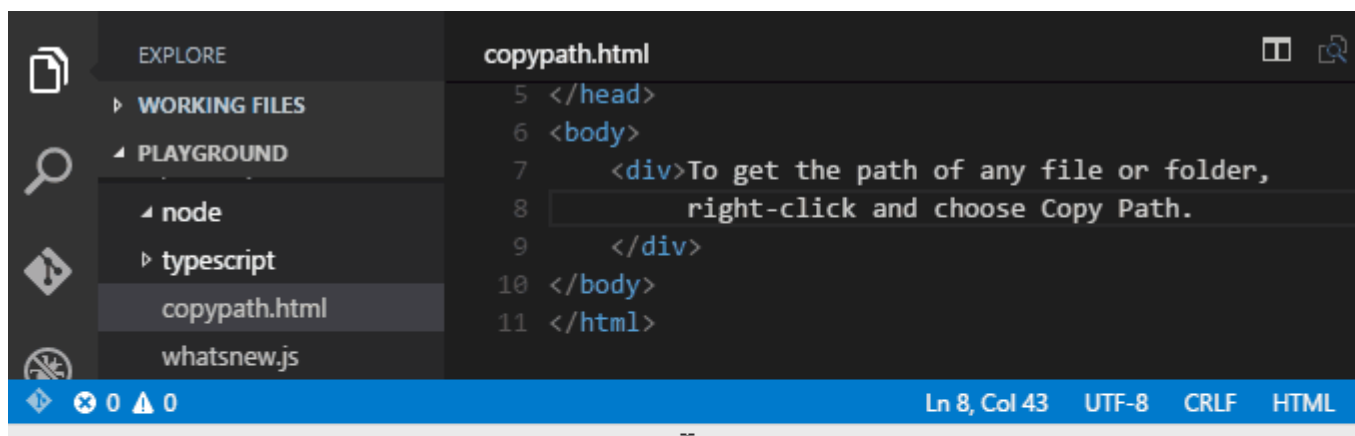


左边的是默认设置，右边的是用户设置，其他的一些设置都可以在左边的默认设置里面找到，我们要修改什么设置，拷贝到右边的窗口中修改即可。例如，我们要修改字体大小，如图修改就好了，注意这是个 json 文件。

工具栏简介

1、资源管理器

下面来看一下右边的工具栏，资源管理器就不用多说了很简单，只是这里有个功能值得注意一下，如图：

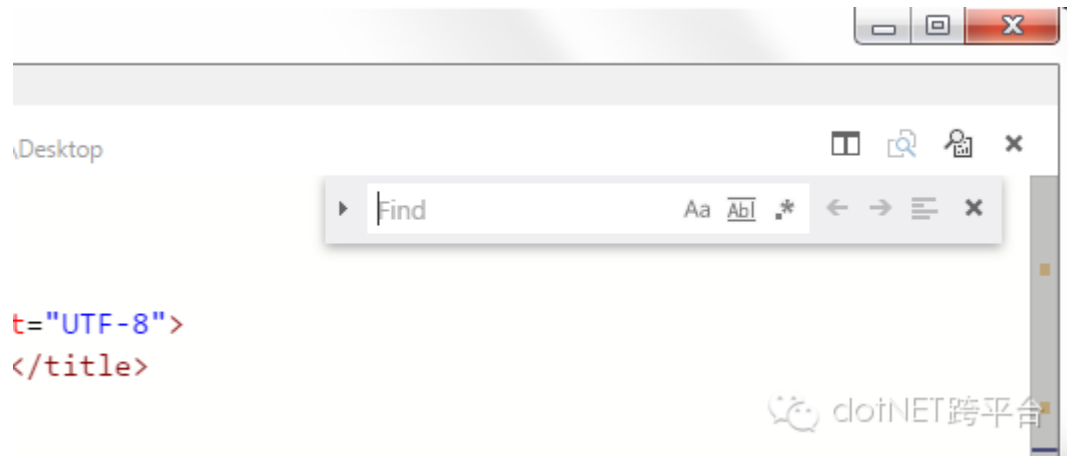


在文件上面点击鼠标右键，有个 Copy Path，可以复制文件的物理路径。当然，你会想为什么不是打开文件位置啊，其实这个功能已经有了，就是 Reveal in Explorer. 记住啦，不要再无知的吐槽了~

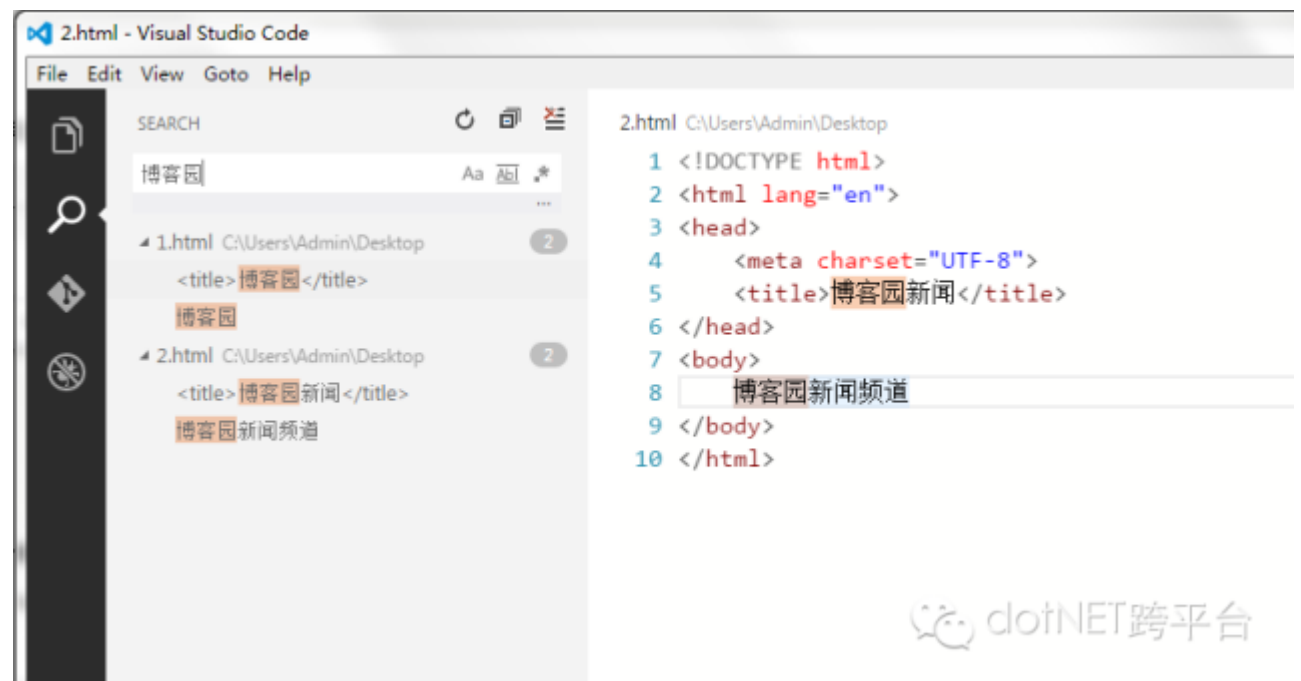
其他的菜单，自己点点就知道了，比较简单，我就不废话了。

2、搜索

下面来说说这个搜索，为什么要在这个地方加一个搜索的工具栏呢？如果你熟悉 Visual Studio 的话，你会发现按快捷键 Ctrl + F 会出现一个搜索框，如图：



细心的你也许会发现，当你按下 Ctrl + Shift + F 的时候，会激活这个工具栏的搜索功能，没错，这个功能就是类似 Visual Studio 中的全局搜索功能，如图：

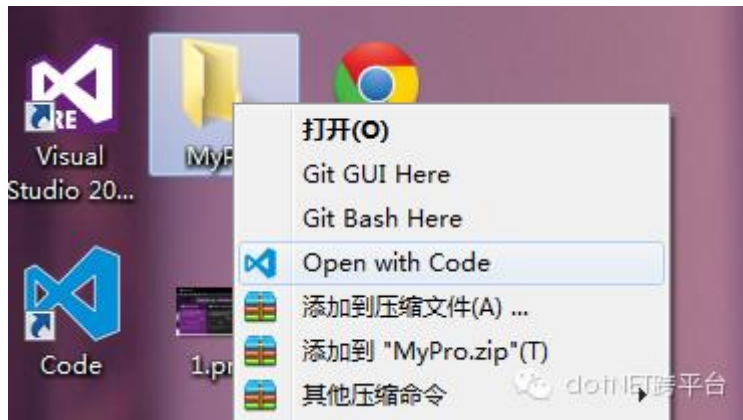


看到了吧，这里的搜索结果，在 1.html 中和 2.html 中都搜索出来了“博客园”这个关键字。看到这里，已经心动了吧~~

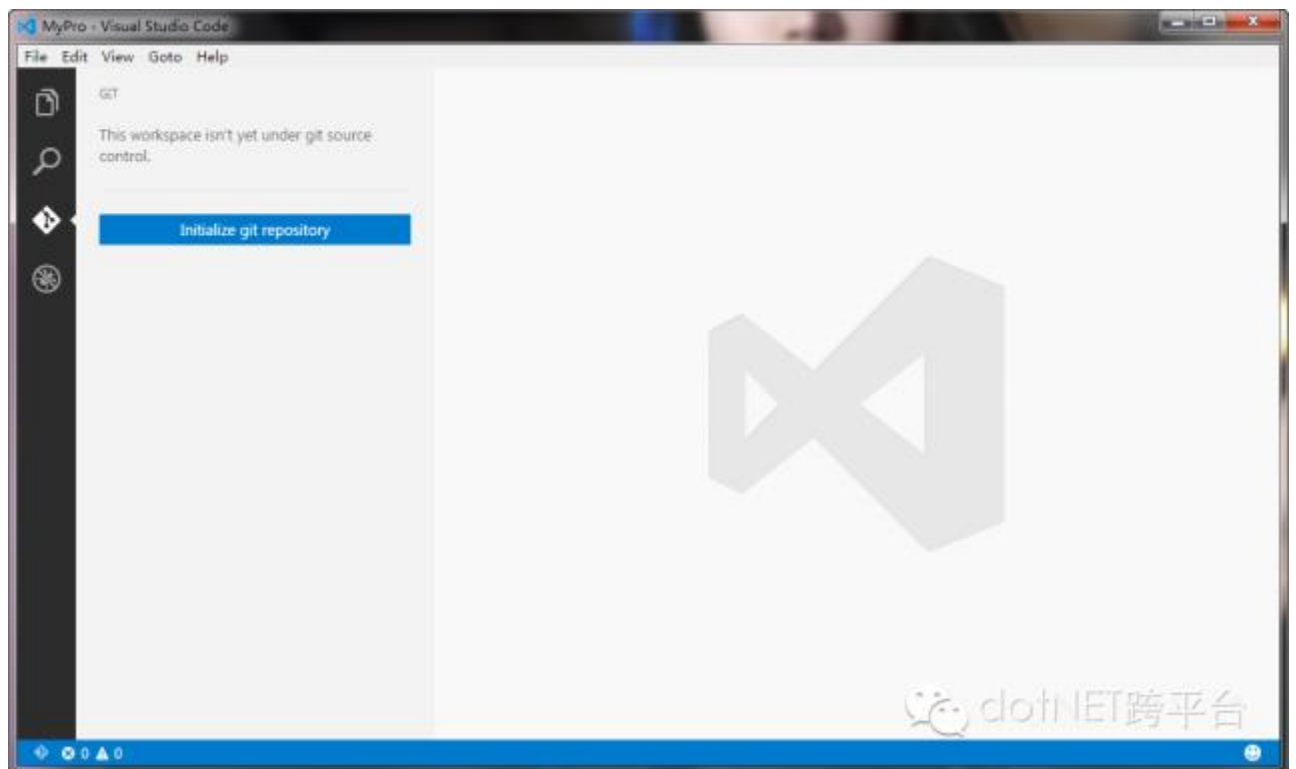
还有 Ctrl + G 键，输入行号可以跳转到指定的行！

3、Git 版本控制

这个功能，用过 Git 大家都清楚，Git 的初始化必须在一个空的文件夹里面，所以要使用这个功能，也是要先有一个文件夹的，下面我们来看看怎么操作，如图：



是不是感觉很流弊，右键文件夹都可以直接打开，打开后，我们继续看，如图：



到了这里，很明显了，点击那个 Initialize git repository 按钮，初始化 Git 仓库。然后，我们新建一个文件，名字为 index.html, 保存到这个文件夹下。接下来，注意啦，注意啦，我们要输入以下内容，如图：

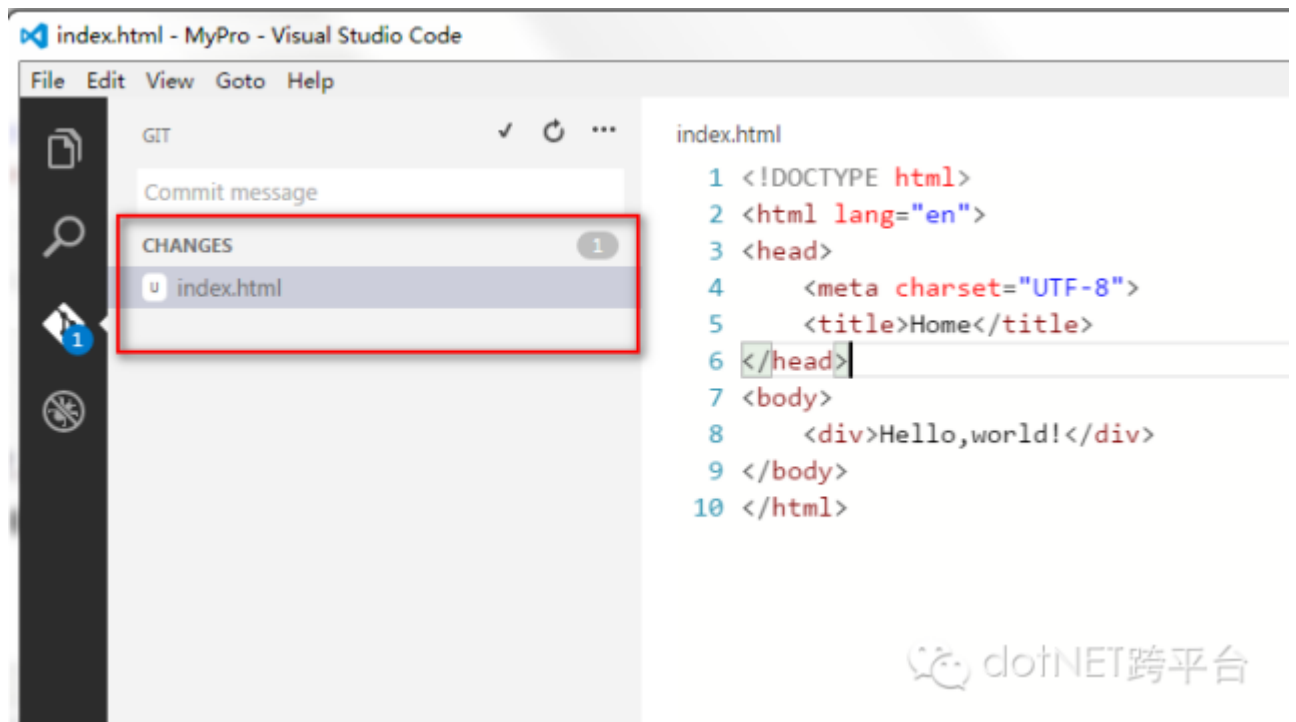
index.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Home</title>
6 </head>
7 <body>
8     <div>Hello,world!</div>
9 </body>
10 </html>
```

dotNET跨平台

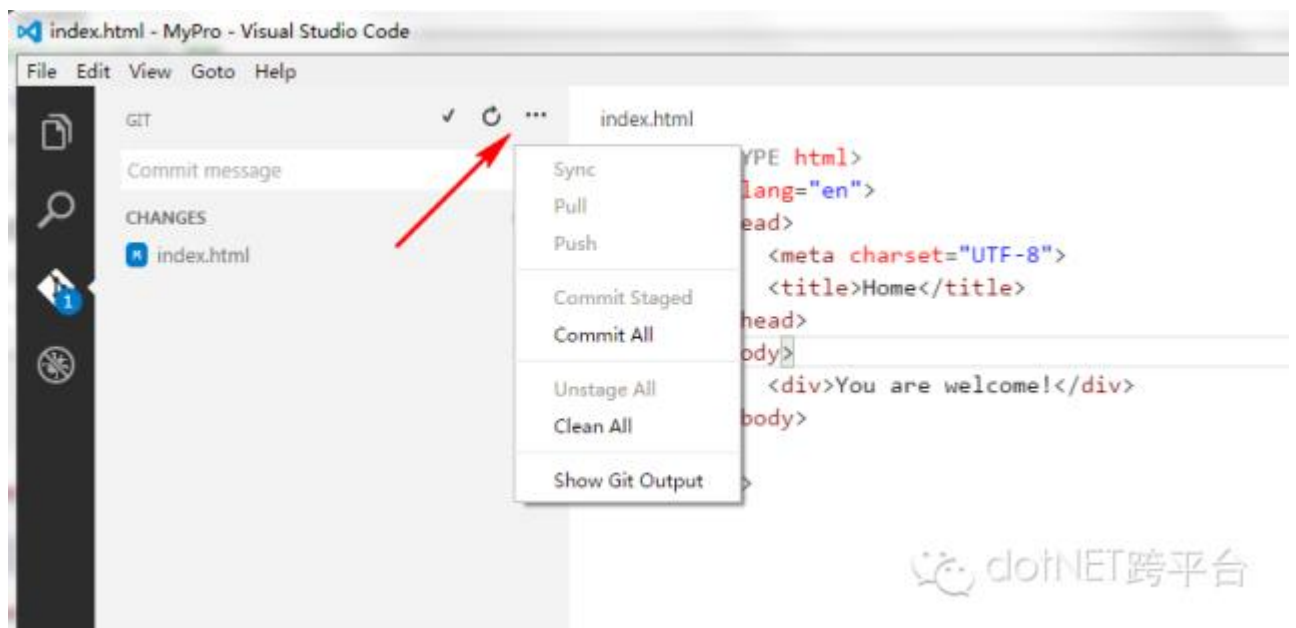
如果你还一行一行的敲，那就太 low 了，这里先交给大家一个方法，在空白的文档里面输入一个感叹号！，注意是英文的哦，然后，点击 Tab 键，看看发生了什么，是不是 duang 的以下就出来了~~

然后，我们按 Ctrl + S 保存文件，再来看 Git 这个工具栏的变化，如图：



dotNET跨平台

我们看到了，changes 里面出现了 index.html，然后，我们输入 commit message，点击上面的对勾提交，然后，我们会看到 changes 下面的文件都消失了，并且右边的 1 变成了 0。



点击那个... 按钮，会弹出菜单，这里有更多的 Git 的操作，我就不一一介绍了，相信使用过 Git 的都看的懂的！

4、调试

这个功能简单的说就是调试代码，如果要使用的话还要进行一些配置，具体大家看官方文档 Debugging, 我就不再细说了，大家有兴趣就自己去研究吧~~

三、Visual Studio Code 上手体验

VS Code 支持多种语言，我们先来看一下官方的说明：

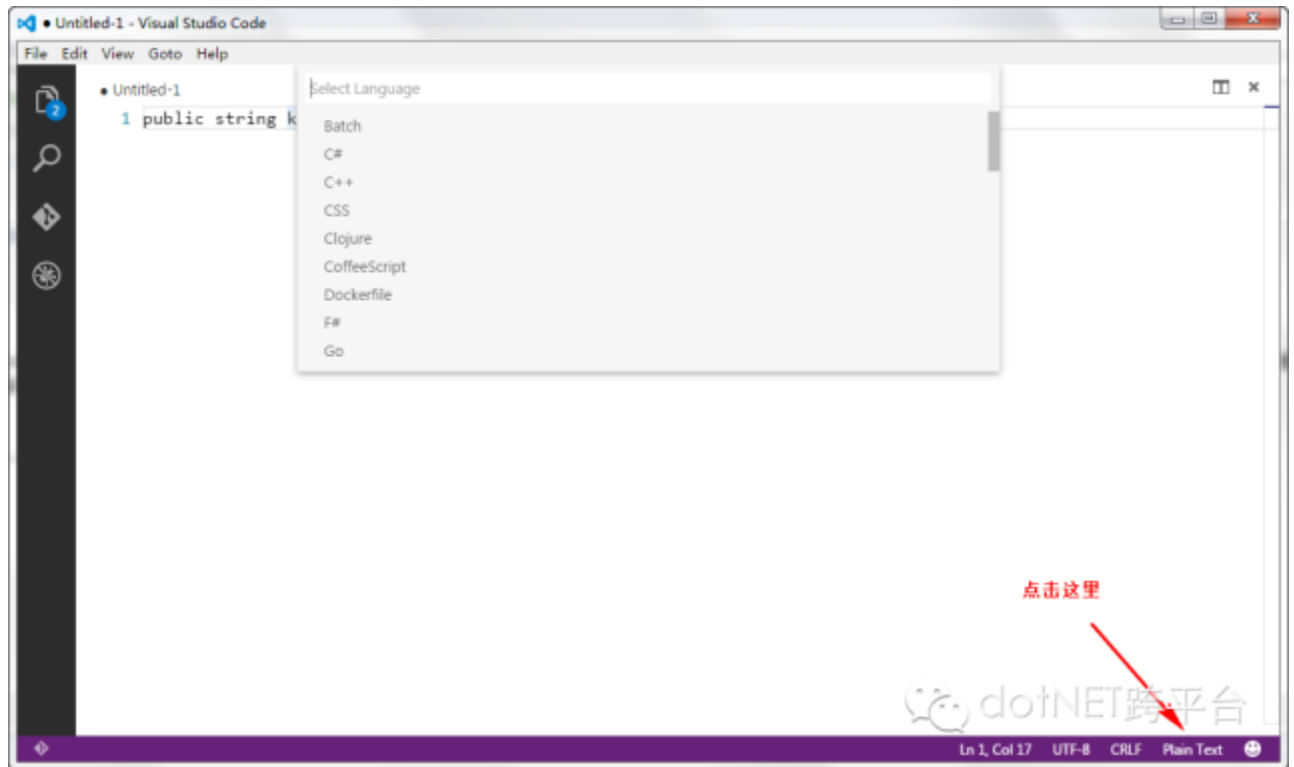
Features	Languages
Syntax coloring, bracket matching	Batch, C++, Clojure, Coffee Script, DockerFile, F#, Go, Jade, Java, HandleBars, Ini, Lua, Makefile, Markdown, Objective-C, Perl, PHP, PowerShell, Python, R, Razor, Ruby, Rust, SQL, Visual Basic, XML
+ IntelliSense, linting, outline	CSS, HTML, JavaScript, JSON, Less, Sass
+ Refactoring, find all references	C#, TypeScript

这里大概说的是对 CSS, HTML, JavaScript, JSON, Less, Sass 几种语言有智能提示，其他的语言都是语法高亮和重构。就是说你写 C#、Java 等一些代码都是没有智能提示的，但是都会有语法高亮。这些代码查看，大家可以打开一些相关的代码文件看看，我就不一一演示了。

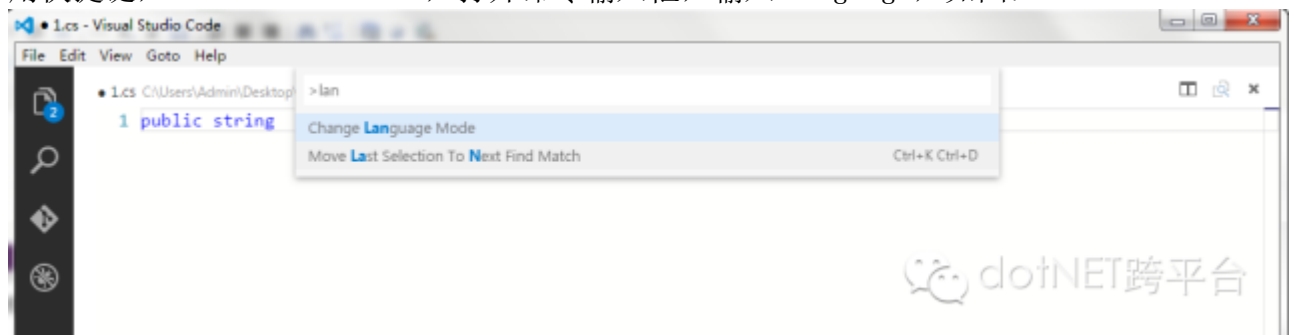
1、文件类型

下面我们来看看 VS Code 编写代码是否顺手，点击菜单新建文件或者使用快捷键 Ctrl + N，这个时候会新建一个 Untitled-1 的文件，默认为文本文件，不高亮和

提示任何代码，这个时候我们可以通过保存文件来改变文件类型，或者直接告诉 Vs Code 文件类型，如图：

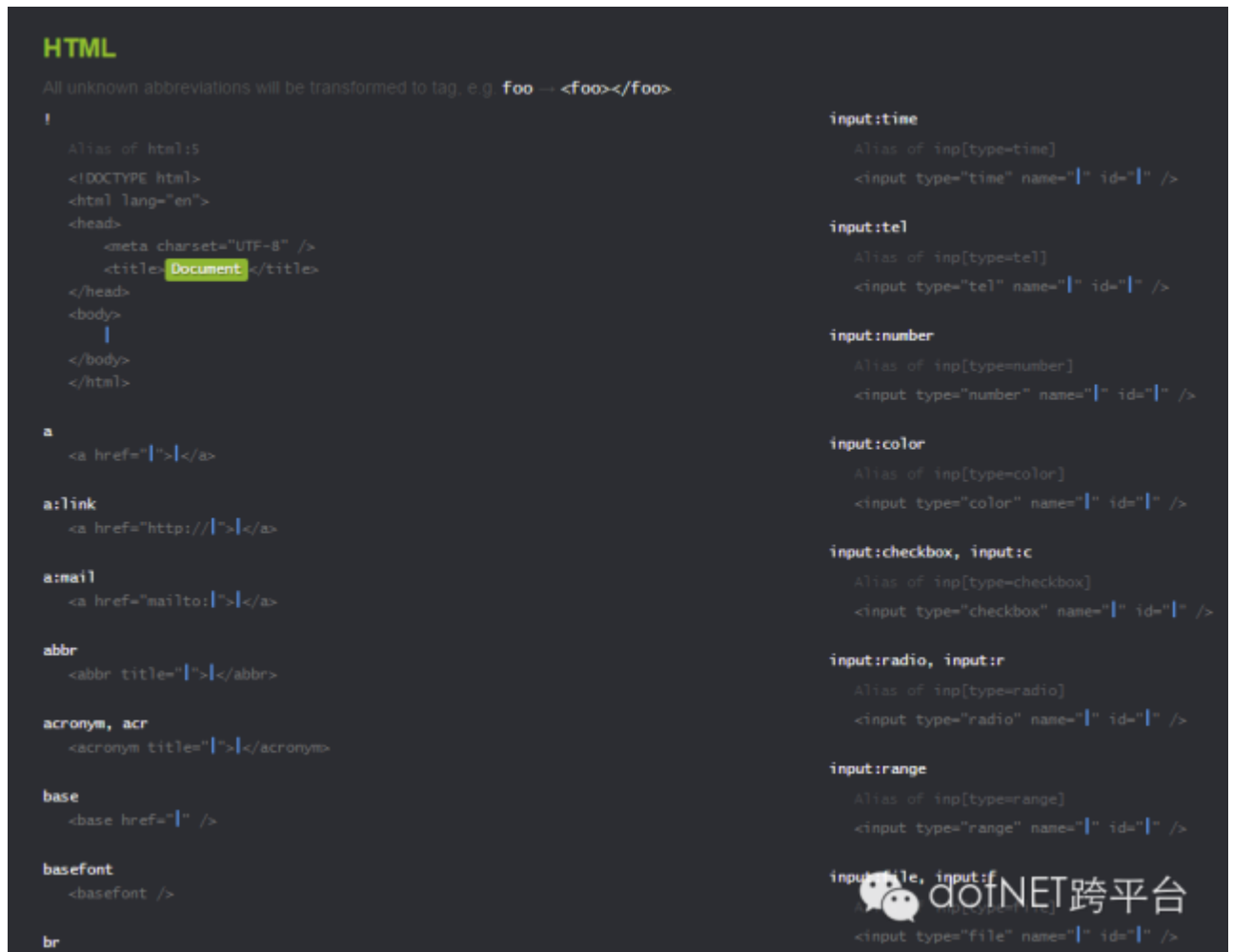


点击 Plain Text，然后在弹出的输入框输入你想要的语言就好了。或者我们使用快捷键，Ctrl + Shift + P，打开命令输入框，输入 language，如图：



选择第一个 Change Language Mode, 也可以打开这个语言选择框。

这里首先要说的是 VS Code 对 html 的支持，当然 VS Code 对 html 支持是非常好的，上面我们已经看到了，输入一个感叹号，然后按下 tab 键就一下子完成了 html5 文档的基本结构。其实，这是 VS Code 里面添加了 Emmet snippet expansion，官方文档地址 <http://docs.emmet.io/cheat-sheet/>，我们大概来看一下：

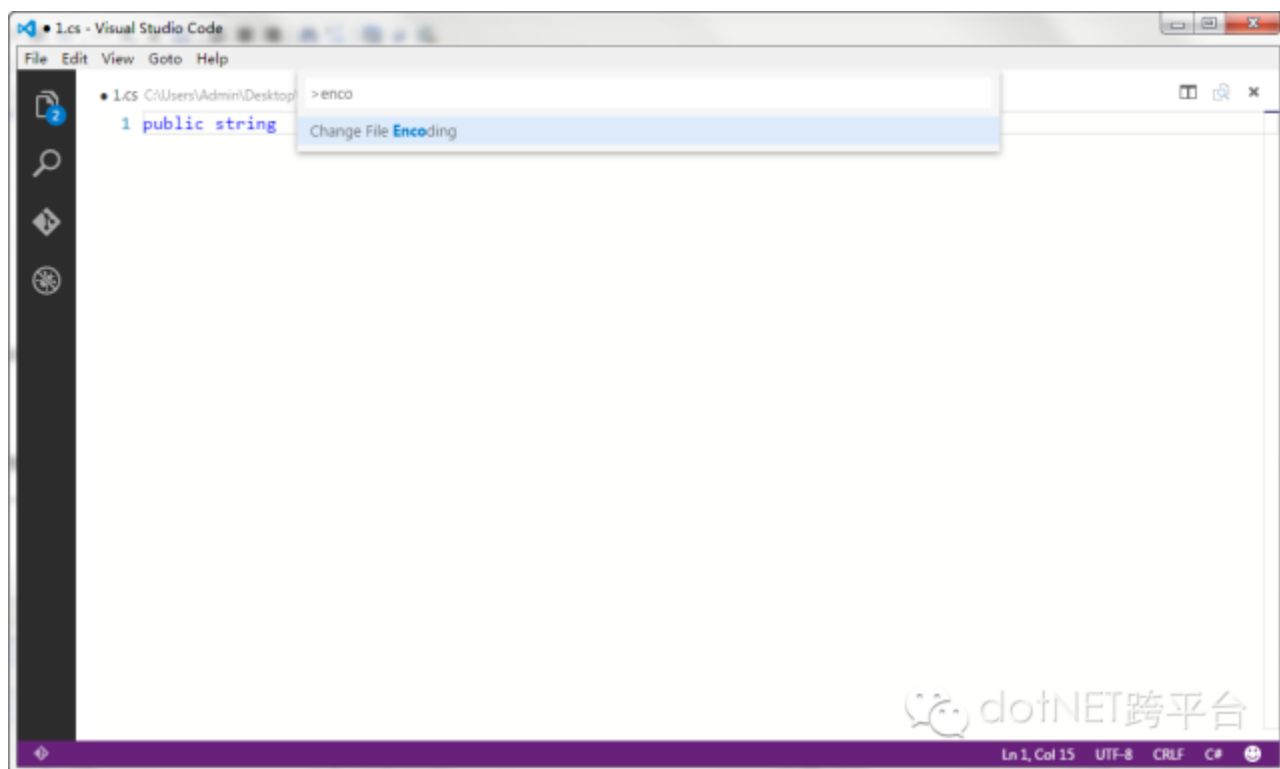


看到了吧，第一个就是我们上面使用的，同理如果我们输入<a>标签，我们只要输入 a，然后按 tab 键就可以了。这里就不再多说了，就留给大家去好好探索了~

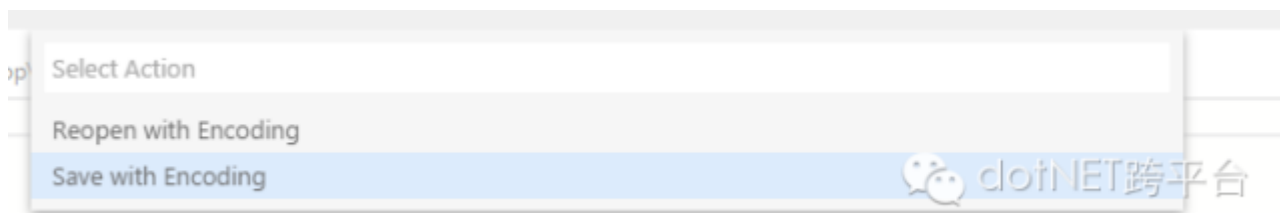
另外，VS Code 也支持 Html5 和 AngularJs 的智能提示，可以说是前端神器啊~~哈哈，目前我知道的也只有 VS2015 支持 AngularJS 的智能提示了。

2、文件编码

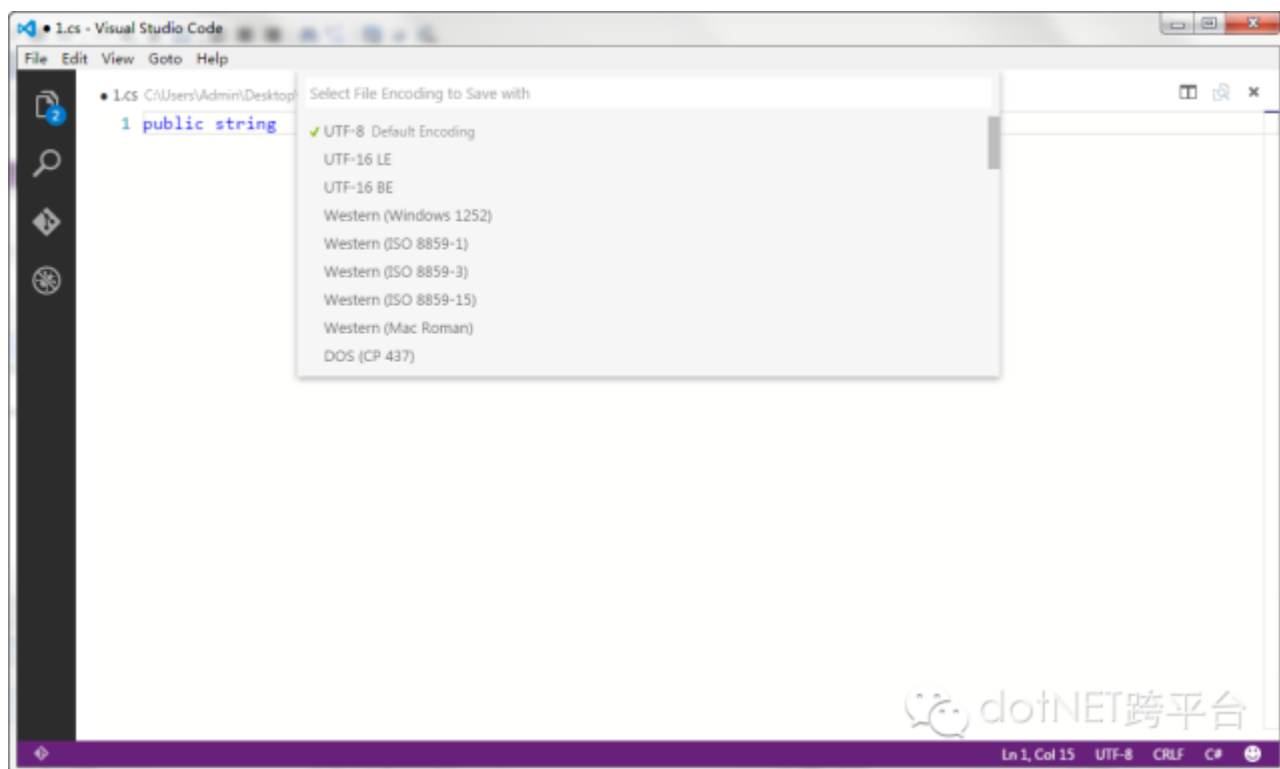
文件编码当然是很重要的，有时候我们要查看和修改文件编码，在介绍 VS Code 的时候，已经告诉大家，在哪里显示文件的编码了，下面我们来说一下如何修改文件编码，还是 Ctrl + Shift + P 打开命令面板，输入 encoding, 如图：



选择以后，会出现如下选项：

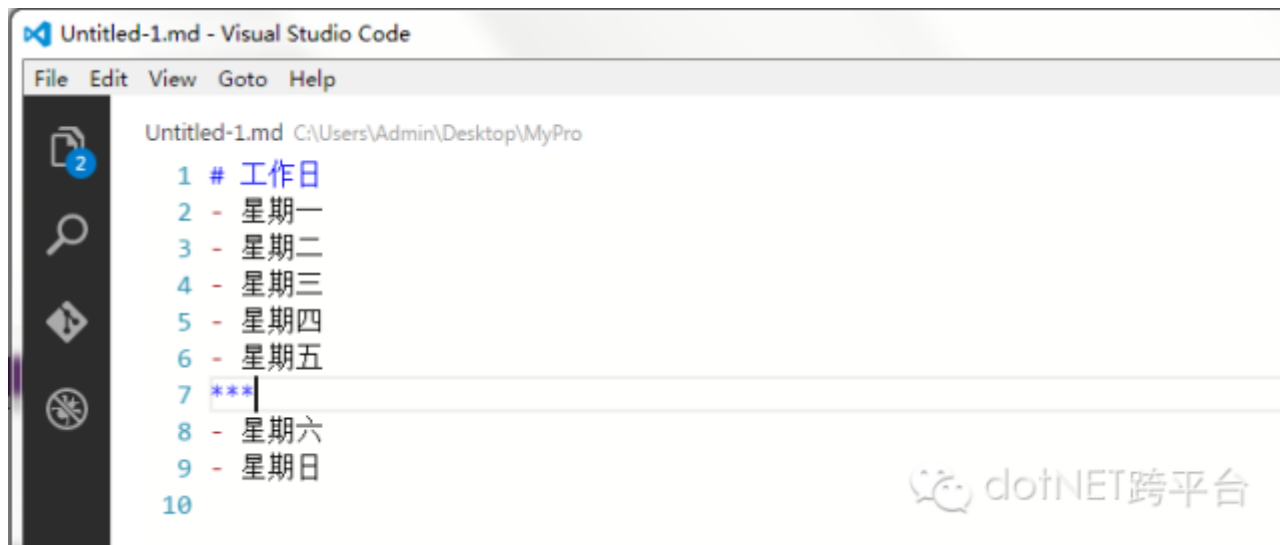


我们可以根据需要选择，这里我们选择保存的编码格式，如图：

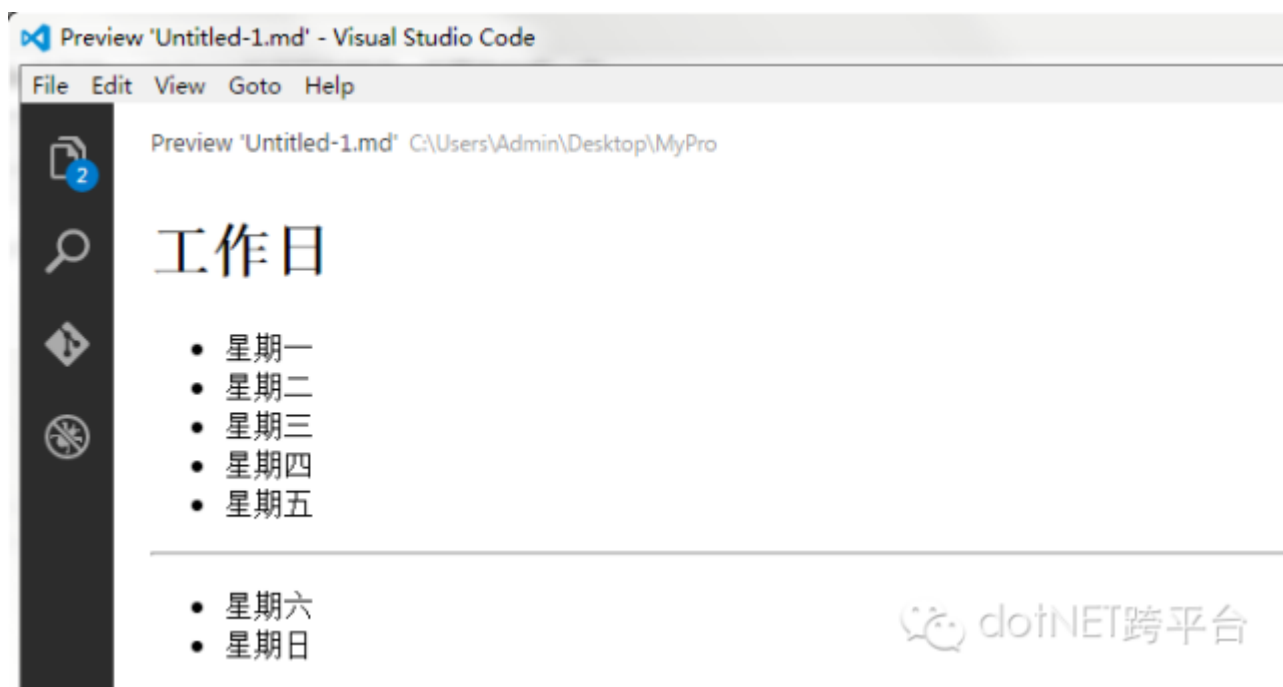


3、Markdown

还有一个比较让人喜欢的一点就是 VS Code 也支持 markdown 的书写和预览，下面我们看一下：

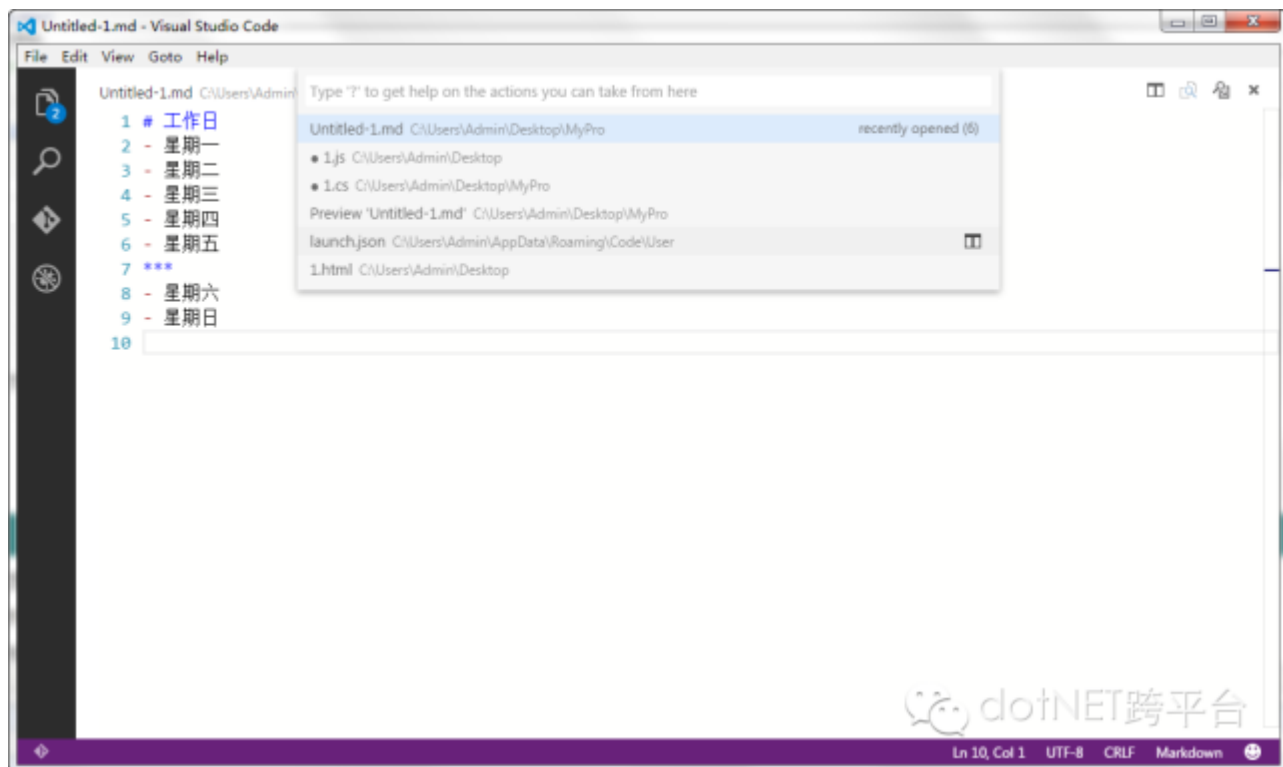


如果要预览的话，要先保存文件，然后按下快捷键 `Ctrl + Shift + V`，就可以预览了。如图：



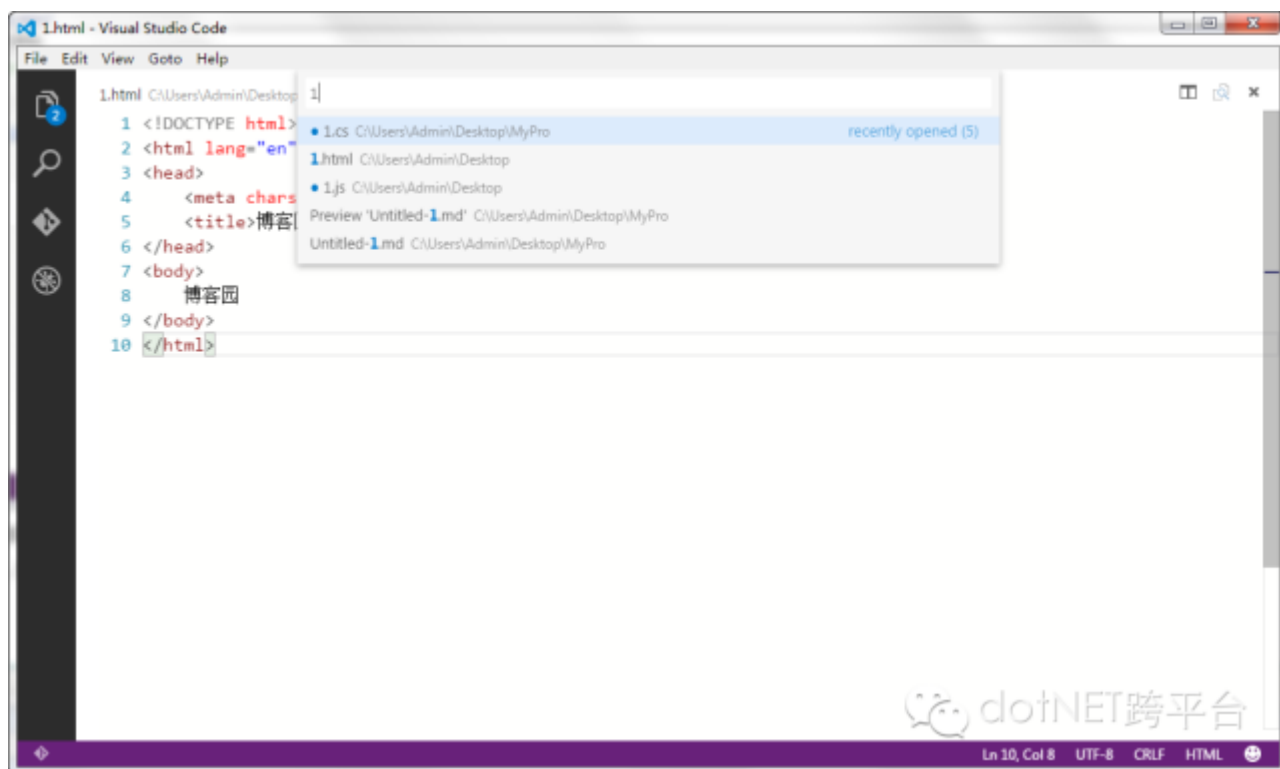
4、文件的切换

当我们编写多个文档时，经常需要在多个文件直接切换，用鼠标点击来切换是一个很奢侈的做法，我们通常都喜欢用快捷键。在 Vs Code 中，我们可以用 Ctrl + Tab 键来快速的切换文件，如图：



按住 Ctrl 键不要松，不断的按下 Tab 键来切换选择文件。当然，这只适用于比较少的文件时，文件多了，这样就不好用了。

文件多的时候，我们可以使用 Ctrl + p 快捷键，打开和上面一样的输入框，不过这个时候我们可以输入内容来搜索文件，然后选择打开。如图：



看到这里,现在感觉 VS Code 怎么样呢?当然,这只是 VS Code 的一小部分功能,还有更多的等待大家去发现!等到正式版出来的时候,估计会更加的好~~

四、总结

由于篇幅的关系,关于 VS Code 就先介绍到这里了,下面来简单的总结一下 Vs Code 的有点:

- 1、支持 30 多种常用语言的语法高亮,并对 html、js、css、Angular 等很好的语法支持,并且还支持 MarkDown 的预览!
- 2、体积小,功能强大,当然性能也是很好的,打开超大型的文本文件也不会卡死,大家可以和其他的一些文本编辑器对比一下。
- 3、支持命令操作(Ctrl + Shift + P)和鼠标操作,还有大量的快捷键,可以适应各种开发者的操作习惯。
- 4、支持 Git 版本控制器,可以完成创建分支、解决冲突、提交修改等操作;
- 5、强大的搜索功能,并且支持多文件搜索;
- 6、最大的有点,当然是跨平台、免费;

Omnisharp

Omnisharp 是一款用于 c#开发的 vim 插件。他是 NRefactory 的缩小版。

主要特性:

- * 代码补全
- * 跳转到定义(类型,变量,方法)
- * 查找类型/标识符(需要 [CtrlP](#) 插件支持)
- * 查看 接口实现/派生类
- * 查看调用
- * Contextual code actions

- * 语法检查（需要 [Syntastic](#) 插件支持）
- * 重命名（重构）
- * 语法高亮
- * 查看信息（包括类型，变量，方法）。并且有两种查看方式，状态栏和预览

- * 语法错误高亮
- * 集成编译功能（需要 vim-dispatch 支持）
- * 代码格式化
- * 添加当前文件到最近的工程文件（.csproj）
- * 添加引用。支持工程和文件引用

安装

```
cd ~/.vim/bundle
git clone https://github.com/nosami/Omnisharp.git
git submodule update --init --recursive
cd Omnisharp/server
xbuild
yum -y install python //安排 Python 2.7.5
```

使用

1、启动 OmniSharp 服务

当你安装 vim-dispatch 之后，用 vim 打开一个.cs 文件，服务就会自动开启。该服务会搜索.sln 文件位置，并启动 OmniSharp 服务，将.sln 文件路径传递给 OmniSharp 服务。

手动启动 OmniSharp 服务：

```
mono Omnisharp.exe -p (portnumber) -s (path\to\sln)
```

Omnisharp 监听端口 2000,所以要更改防火墙配置。

2、自动补全

在插入模式下，按 Ctrl-X Ctrl-O，可调出补全提示。如果安装了 SuperTab 插件，则点击 Table 就会弹出补全提示。如果想实现敲击字母就自动补全，可以研究一下这几个插件 NeoComplete, YouCompleteMe 和 NeoComplCache.

3、语法错误检测

当保存当前文件时，自动检测

4、其他特性需要绑定快捷键。具体参考下面的.vimrc 配置

<http://www.techrepublic.com/article/create-c-code-on-your-mac-using-a-tom-and-omnisharp/>

ASP.NET Linux 部署

ASP.NET 5 依然处于 RC 版本, 并不十分成熟. 但可以预见到的是, 就算本月 ASP.NET 5 RTM 版本如期推出, 其在 Linux 上面的开发和部署前景依然不是非常明朗: 特别令人困惑的是, MS 在 Linux 上至今仅仅推出了几个以开发为目的的简单服务器实现, 难以在其计划中寻觅到类似 IIS 的完整部署环境, 那么所谓的 ASP.NET 5 的跨平台开发是否只能停留在实验室水平? ASP.NET 5 要下半年才能 RC 和 RTW. 来月底 VS2015 只能完成 ASP.NET 4.6 的常规升级了. 今年之内想在 linux 下跑应用还得依靠 Mono。

ASP.NET	ASP.NET 是 .NET Framework 的一部分, 是一项微软公司的技术, 是一种使嵌入网页中的脚本可由因特网服务器执行的服务器端脚本技术, 本月即将发布的最新版本是版本 5, 又成为 vNext.
Linux	Linux 是一套免费使用和自由传播的类 Unix 操作系统, 是一个基于 POSIX 和 UNIX 的多用户、多任务、支持多线程和多 CPU 的操作系统. 本文中的 Linux 主要以 Ubuntu 作为样例.
Mono	mono 是指由 Novell 公司(由 Xamarin 发起, 并由 Miguel de Icaza 领导的, 一个致力于开创 .NET 在 Linux 上使用的开源工程. 就目前而言, 在 Linux 上的 .NET 应用还必须基于 Mono 来运行.
Jexus	Jexus 即 Jexus Web Server, 简称 JWS, 是 Linux 平台上的一款 ASP.NET WEB 服务器, 是目前唯一能够支持企业级 ASP.NET Linux 部署的一种方案(其他的服务器方案无类似定位).
OWIN	OWIN 在 .NET Web Servers 与 Web Application 之间定义了一套标准接口, OWIN 的目标是用于解耦 Web Server 和 Web Application. 基于此标准, 鼓励开发者开发简单、灵活的模块, 从而推进 .NET Web Development 开源生态系统的发展。
MS Owin	微软开发的基于 OWIN 规范的底层实现, 最新版本是 3.0.1, 其主项目名称为 Kanata
ASP.NET WebApi	ASP.NET MVC 4 包含了 ASP.NET Web API, 这是一个创建可以连接包括浏览器、移动设备等多种客户端的 Http 服务的新框架, ASP.NET Web API 也是构建 RESTful 服务的理想平台
RESTful	一种软件架构风格, 设计风格而不是标准, 只是提供了一组设计原则和约束条件。它主要用于客户端和服务端交互类的软件。基于这个风格设计的软件可以更简洁, 更有层次, 更易于实现缓存等机制。
NancyFx	Nancy 是一个基于 .NET 和 Mono 平台用于构建轻量级基于 HTTP 的 Web 服务。基于 .NET 和 Mono 平台, 框架的目标是保持尽可能多的方式, 并提供一个 super-duper-happy-path 所有交互。官方网站 http://nancyfx.org/

三种选择

就 .NET 路线的 Web 开发来看, 不管何种方式, 未来必然是基于 OWIN 开发的事实已经不可动摇了; 在这个基础上, 我认为目前在 Linux 上开发并部署 .NET Web 应用程序有 3 个路线可以选择:

1. **底层 Owin 路线**：选择 MS 的底层 OWIN 实现, 配合其他基于 OWIN 的独立组件, 形成以底层 OWIN 为核心的自行搭配的轻型构架, 这个方案目前已经可以完美部署到 Jexus.
2. **三方构架路线**：选择同样基于 OWIN 标准的, 非 MS 的三方构架实现, 目前最有潜力, 名气最大的是 NancyFx, Nancy 框架目前也同样能较好的部署到 Jexus 上去.
3. **正统 vNext 路线**：选择 MS 正统的下一代 ASP.NET 5 (vNext), 该版本的底层基于 OWIN 实现 (注意任何老的 ASP.NET 版本都不是基于 OWIN 的), 可谓是亲儿子; 但目前还没有发布正式版本, 其未来不可预期, 最关键的一点是, 在 Linux 上, 包括 Jexus 在内, 目前依然没有完美的基于商业环境的部署服务器环境支持.

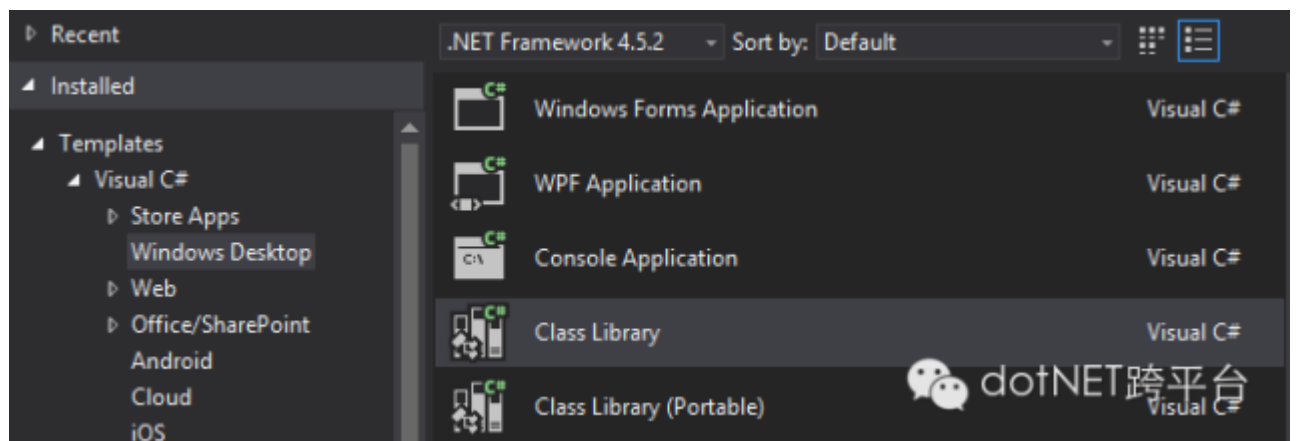
就这 3 个方案而言, 我觉得各有利弊: 底层方案需要更多的自行选择和组装, 但是与任何基于 Owin 的组件搭配自如; 三方方案面临生态环境的问题, 由于大部分高端的组件都来自 MS, 能否真正无缝连接需要考验, 自身的生存能力也是问题; 正统方案内容完整, 支持强大, 与 MS 各项技术融合度高, 但却面临成熟周期问题 (时不我待), 另外我最不爽的一点是, vNext 又一次搞成了铁索连环船, 连 WebApi 都和 MVC 融合了, 又给人一种整套推销的感觉, 有违当初 Owin 体系的初衷. 而最根本的问题是, 目前还没有任何方案来实现一个 Linux 上的 IIS 级服务器, 没有好的载体, 仅仅是把 Linux 定义为娱乐这个显然看不出太大的诚意.

综上, 我目前还是倾向于使用底层 Owin 方案, 目前商业化开发路线是: 基于 MS Owin 实现, 根据需要加入各种 MS 稳定组件, 比如 Web API 2.2 OWIN 5.2.3, Identity Owin 2.2.1, SignalR OWIN 1.2.2, OAuth 3.0.1, 和其他所有的通用型组件, 如 EF, Logging, IoC 等等; 最终通过 Mono 和 Jexus 架设到 Linux 环境. 下面我建一步演示如何组装 MS Owin 和 Web API 2.2, 并把它们部署到 Jexus 上去.

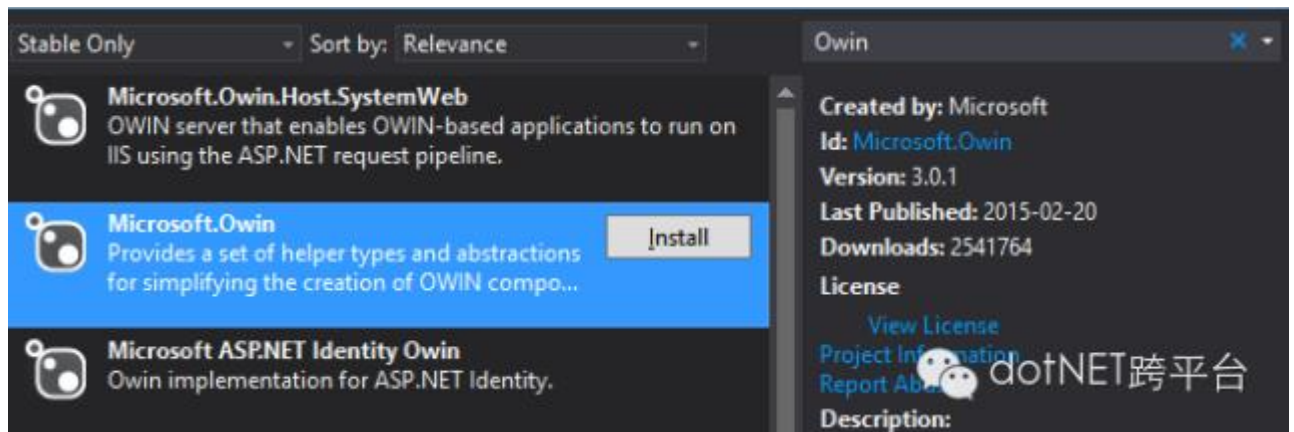
开发环境 VS 2013, Window 7 或 8; 部署环境 Ubuntu 15.

第一步: 建立项目

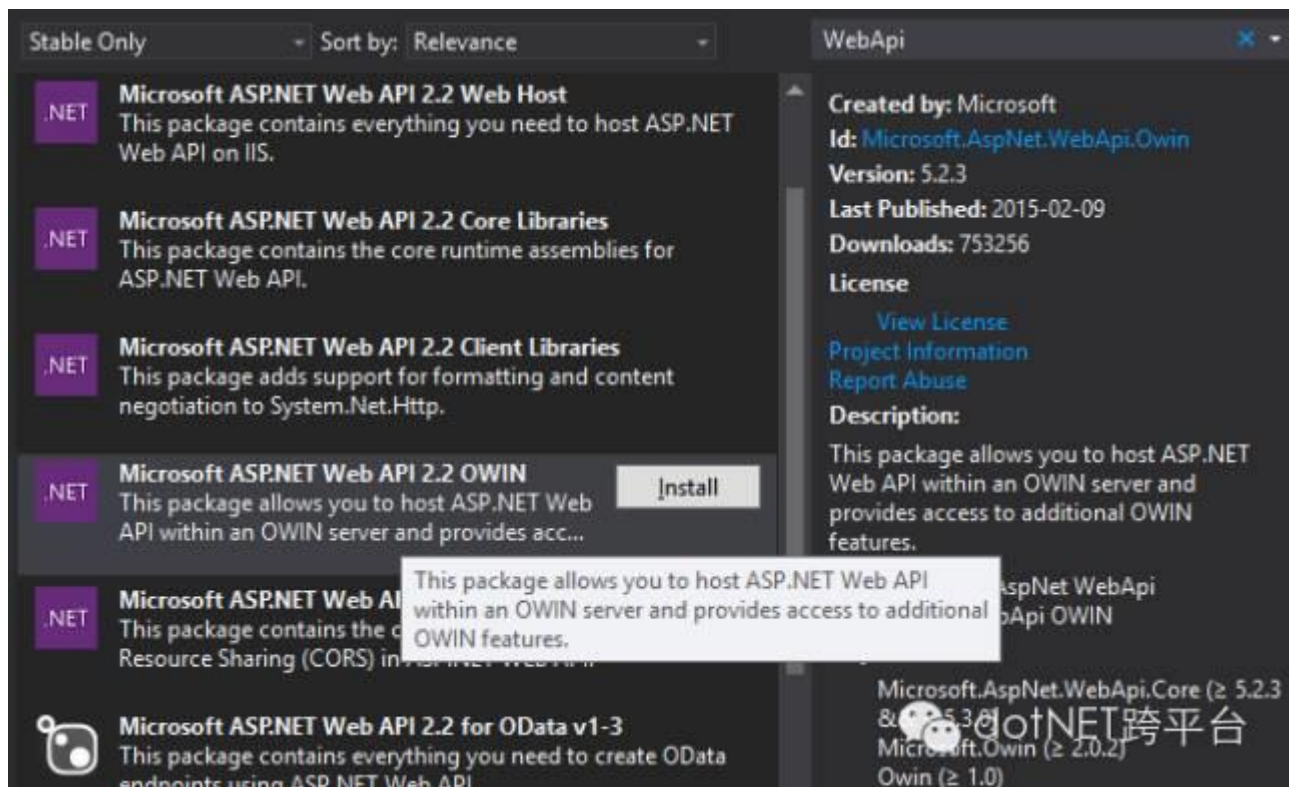
首先, 在 VS 2103 中建立一个 Class Library 项目, 注意只要 Library 项目, 这里可以选择 Framework 4.5.2 或者 4.5.1. 这个项目假设命名为 OwinExample.



然后, 我们加入这个项目必须的组件, 根据上面的描述, 我们需要 2 个组件: MS Owin 的核心实现 Microsoft Owin 和 ASP.NET WebApi 2.2 Owin
我们先加入 Microsoft Owin



然后加入 ASP.NET WebApi 2.2 Owin



第二步：建立 Owin 入口代码

首先,Owin 的传统入口类登场：Startup.cs

```
using Owin;
using System.Web.Http;
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        #region WebApi
        var httpConfig = new HttpConfiguration();
        httpConfig.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{action}/{id}",
```

```

defaults: new { id = RouteParameter.Optional }
);
//强制设定目前的 WebApi 返回格式为 json
httpConfig.Formatters.Remove(httpConfig.Formatters.XmlFormatter);
//加载 WebApi 中间件
app.UseWebApi(httpConfig);
#endregion
}
}

```

几个要点:

- 1 Startup 中的 Configuration 写成类成员方式, 而不是静态方式, 是为了和 Jexus 适配器配合, 其实差异不大.
- 1 WebApi 的配置写法和 MVC 基本类似.
- 1 Startup 和 Configuration 的命名并不是固定的, 只是预定俗成而已.

第三步: 建立 WebApi 代码

建立 **DefaultController.cs** 为一个默认的 WebApi, 里面包含一个最简单的 Hello 函数.

```

using System.Web.Http;
[AllowAnonymous]
public class DefaultController : ApiController
{
    [HttpGet]
    public string Hello()
    {
        return "Hello Owin!";
    }
}

```

自此, 简单的 MS Owin + WebApi 程序架设完毕. 在 Owin 体系下, 我们发现一切都变得非常简单和清晰.

第四步: 建立 Jexus 适配器代码

为了把项目部署到 Jexus 上去, 我们还需要一个非常简单的适配器类, 在项目中加入这个类以后, 就能无缝部署到 Jexus 服务器上去了, 我们把这个代码命名为 **Adapter.cs**:

/***** 加载

Microsoft.Owin.dll 进行 owin 编译的适配器 (插件) 示例

目的:

- * 演示如何将自己的处理方法 (中间件) 加入到 Microsoft.Owin.dll 的处理环节中

- *

- * 使用方法:

- * 将编译得到的 dll 连同 Owin.dll、Microsoft.Owin.dll 等文件一并放置到网站的 bin 文件夹中

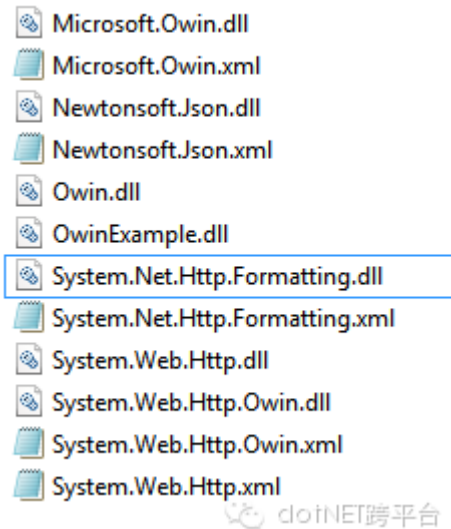
```

*****/region <USINGS>using System;using
System.Collections.Generic;using System.Threading.Tasks;using
Microsoft.Owin.Builder;#endregionnamespace OwinExample
{
    public class Adapter
    {
        static Func<IDictionary<string, object>, Task> _owinApp;
    }
    /// <summary>
    /// 默认构造函数          /// </summary>
    public Adapter()
    {
        ///创建默认的 AppBuilder
        var builder = new AppBuilder();
        ///创建用户定义的 Startup 类          ///这个类中必须有
        “Configuration” 方法
        var startup = new Startup();          ///调用
        Configuration 方法，把自己的处理函数注册到处理流程中
        startup.Configuration(builder);          ///生成 OWIN “入口” 函数
        _owinApp = builder.Build();
    }
    /// <summary>
    /// *** JWS 所需要的关键函数 ***          /// <para>每个请求到来，
    JWS 都把请求打包成字典，通过这个函数交给使用者</para>
    /// </summary>
    /// <param name="env">新请求的环境字典，具体内容参见 OWIN 标准
    </param>
    /// <returns>返回一个正在运行或已经完成的任务</returns>
    public Task OwinMain(IDictionary<string, object> env)
    {
        if (_owinApp == null) return null;          //
        将请求交给 Microsoft.Owin 对这个请求进行处理          //（你的处理方法
        已经在本类的构造函数中加入到它的处理序列中了）
        return _owinApp(env);
    }
}
}

```

这里再次感谢 Jexus 作者宇内流云提供的代码，出于对原作者的敬意这个代码除了命名空间以外我一个字母也没有改，其实也不需要改。其实大家可以看的出来，这么变态的注释应该不是我故意去写的。

自此我们的基于 MS Owin 和 WebApi 的迷你版应用开发完成，改为 Release 模式编译，我们可以得到如下图所示的一系列 DLL：



就这些 DLL 就能形成一个 WebApi 应用吗?事实就是如此, 而且这个应用能很好的部署到 Linux 环境上去.

第五步: 安装 Jexus 环境

这里先声明下, 基于个人的能力所限, 只能先给出 Ubuntu 最新版本的一个部署方案, 使用其他版本 Linux 的兄弟只能麻烦你们自寻门路了.

首先, 我们再 Ubuntu 上面安装 Mono 最新版本. 可以参考下面超链文章的指引:

<http://www.linuxdot.net/bbsfile-3090>

然后我们安装 Jexus 最新版本. (同样请参考下面的超链)

<http://www.linuxdot.net/bbsfile-3500>

第六步: 部署到 Jexus

部署 Jexus 网站的常规指导信息, 大家可以移步这里:

<http://www.linuxdot.net/bbsfile-3084>

下面说下我们的特殊部署步骤 (具体 Linux 命令我就不列举了):

1. 建立网站目录 `/var/www/owinexample`, 然后在这个目录下再建立一个 `bin` 目录.
2. 通过各种方式把上面自己开发产生的 `dll` 拷贝网站目录的 `bin` 目录下.
3. 建立 Jexus 网站的配置文件, 假设我们命名为 `owinexample`

其重要内容应该包括以下设置:

```
# For owinexample
```

```
port=88
```

```
root=/ /var/www/owinexample
```

```
hosts=* # or your.com, *.your.com
```

```
OwinMain=OwinExample.dll, OwinExample.Adapter
```

特别强调的是 `OwinMain` 这个必需配置, 并且需要对应正确的 DLL 文件名和 `Apdater` 类. 根据前面的描述, 我们可以知道我们应用的配置应该是 `OwinExample.dll`, `OwinExample.Adapter`.

另外, `Web.Config` 文件和其他任何文件在这种构架里面不是必须的.

1. 重启 Jexus 服务
2. 打开你的浏览器, 输入 `http://linuxserverip:88/api/default/hello` 就可以看到结果.

注意 linuxserverip 为部署服务器的 IP, 88 为我们再 Jexus 配置中设置的端口, api/default/hello 对应我们 WebApi 的路径映射, Controller 类名和方法名.

结束语

最后还是说下我们这种模式的优势, 劣势和意义:

优势: 基于 Owin 底层, 简单明了稳定, 可以融合任何基于 Owin 的相关技术, 扩展性强, 可以和 Mono, Jexus 完美结合, 性能最高.

劣势: 相当于自行组建构架, 搭建工作量大, 由于目前没有独立的 MVC 组件, 在 MVC 开发方面缺乏支持 (Nancy 的一部分 MVC 构架比如 Razor 引擎可以独立移入, 但这个方案有待验证).

这个方案的最终意义在于, 结合目前 .NET 和 Linux 方向上最具备稳定性和代表性的 MONO, MS Owin 和 Jexus, 在 ASP.NET vNext 最终能完美部署到 Linux 之前, 这是最接近于商业生产环境的方案之一.

最后拖一句, 这个方案的开发环境可以考虑用 TinyFox 或者 MS Owin Self Host 来做宿主. 都可以无缝连接, 代码不需要修改.