Трассировка и отладка в .NET

* [Трассировка и отладка в .NET](http://sharp-generation.narod.ru/ArchitectureNet/Glava14/trassiring.html#link1)
* [Пример TraceDemo](http://sharp-generation.narod.ru/ArchitectureNet/Glava14/trassiring.html#link2)
* [Разворачивание TraceDemo.exe.config](http://sharp-generation.narod.ru/ArchitectureNet/Glava14/trassiring.html#link3)
* [Использование классов Debug (Отладка) и Trace (Трассировка)](http://sharp-generation.narod.ru/ArchitectureNet/Glava14/trassiring.html#link4)
* [Использование переключателей для активизации диагностики](http://sharp-generation.narod.ru/ArchitectureNet/Glava14/trassiring.html#link5)
* [Активация и деактивация переключателей](http://sharp-generation.narod.ru/ArchitectureNet/Glava14/trassiring.html#link6)
* [Установка переключателей в файле конфигурации](http://sharp-generation.narod.ru/ArchitectureNet/Glava14/trassiring.html#link7)
* [Установка переключателей программным путем](http://sharp-generation.narod.ru/ArchitectureNet/Glava14/trassiring.html#link8)
* [Использование переключателей для управления выводом](http://sharp-generation.narod.ru/ArchitectureNet/Glava14/trassiring.html#link9)
* [Класс TraceListener](http://sharp-generation.narod.ru/ArchitectureNet/Glava14/trassiring.html#link10)
* [Коллекция слушателей](http://sharp-generation.narod.ru/ArchitectureNet/Glava14/trassiring.html#link11)
* [Резюме](http://sharp-generation.narod.ru/ArchitectureNet/Glava14/trassiring.html#link12)

Трассировка и отладка в .NET

Cложные приложения не всегда удается эффективно отлаживать под управлением отладчика, чтобы узнать, что пошло не так, как надо Дублирование, а также написание условий, необходимых для отладки, становится зачастую сложной задачей В пространстве имен System: : Diagnostics (Система Диагностика) есть определенные классы, которые помогают оснастить ваше приложение инструментальными средствами

Оснастив приложение необходимыми инструментальными средствами, в процессе отладки и трассировки вы сможете сделать его более устойчивым При этом также проясняется общий шаблон того, как каркас разбивает классы на отдельные задачи (запись вывода, управление выводом, назначение вывода) так, чтобы можно было настроить эти отдельные части, и в остальном полностью полагаться на классы Framework Механика оснащения приложения имеет три аспекта

Классы Trace (Трассировка) и Debug (Отладка) используются для генерации вывода трассировки и отладки Они имеют идентичные методы и свойства, которые позволяют выводить диагностику Однако эти классы не определяют назначение вывода

Классы Listeners (Слушатели) используются для направления вывода на различные устройства, хотя в то же время существует и назначение, принятое по умолчанию

Макроопределения Idef me DEBUG и tfaefine TRACE могут использоваться наряду с макросами lifdef и tendif для включения и отключения документирования Эти макро-флаги могут использоваться для того, чтобы различать отладочные и рабочие конструкции приложения Можно также поставить вывод классов Trace (Трассировка) и Debug (Отладка) в зависимость от значений условных выражений И наконец, можно управлять детализацией вывода, исходя из потребности в информации, с помощью классов BooleanSwitch и TraceSwitch

Пример TraceDemo

Пример TraceDemo иллюстрирует функции диагностики Если вы запустите этот пример на выполнение, то получите следующий вывод

Trace Listeners: Default

This was compiled with a DEBUG directive1 This was compiled with a TRACE directive1

Debug Boolean Switch disabled at startup.

Debug Boolean Switch enabled1 Trace Switch Startup Value = Warning TraceError' TraceWarning'

Trace Listeners:

Console::0ut Listener Output File Listener

File output.txt has been created

Вот перевод

Слушатели Трассировки. По умолчанию

Это было откомпилировано с директивой ОТЛАДКИ1 Это было откомпилировано с директивой

ТРАССИРОВКИ

Булев переключатель Отладка заблокирован при запуске. Булев переключатель Отладка разблокирован'

Значение переключателя Трассировка при запуске = Предупреждение TraceError' TraceWarning'

Слушатели Трассировки:

Консоль: : Слушатель Слушатель Выходного файла

Файл output.txt был создан

На этот вывод мы будем ссылаться при последующем изложении материала Кроме того, в папке проекта вы можете найти соответствующий файл output. txt

Разворачивание TraceDemo.exe.config

Проект TraceDemo имеет дополнительный шаг компоновки (рис. 14.1) для копирования файла конфигурации TraceDemo. ехе . conf ig выходной программы в папку отладки Debug (Отладка)2 Это проделывается для того, чтобы программа могла найти указанный файл конфигурации во время своего выполнения и определить начальные параметры трассировки и отладки, заданные в нем Вскоре вы увидите содержание этого конфигурационного файла

Рис. 14.1. Окно визуальной среды разработки Visual Studio для дополнительного шага компоновки

Использование классов Debug (Отладка) и Trace (Трассировка)

Все методы и свойства, содержащиеся в классах Debug (Отладка) и Trace (Трассировка), являются статическими Перегружаемые функции WriteLine и Write (Запись) используются для записи вывода отладки и трассировки Перегружаемые функции WriteLinelf и Writelf записывают вывод, если условие в их первом аргументе принимает значение true (истина)

Debug::WriteLine( // Отладка

"This was compiled with a DEBUG directive1");

// "Это было откомпилировано с директивой ОТЛАДКИ' ");

Trace::WriteLine(

"This was compiled with a TRACE directive1");

// "Это было откомпилировано с директивой ТРАССИРОВКИ1 ");

Debug::WriteLinelf( // Отладка

DebugBooleanSwitch->Enabled,

"Debug Boolean Switch enabled at startup.");

// "Булев переключатель Отладка разблокирован при запуске. ");

Debug::WriteLinelf( // Отладка

1DebugBooleanSwitch->Enabled,

"Debug Boolean Switch disabled at startup.");

// " Булев переключатель Отладка заблокирован при запуске. ");

Вывод форматируется с помощью методов Indent (Отступ) и Unindent (Неотступ) класса Trace (Трассировка) Размер отступа форматирования управляется свойством IndentSize, которое может быть изначально установлено в файле конфигурации и изменено программным путем в процессе выполнения программы Ниже показано, как Программно установить размер отступа

Trace::IndentSize=10;

Ниже показано, как установить размер отступа в файле конфигурации приложения:

<?xrr.I version="l. 0"?> <!-- xml версия = "1.0" -->

<configuration> <!-- конфигурация -->

<system.diagnostics>

<trace indentsize="15" />

</system.diagnostics>

</configuration> <!-- конфигурация -->

Метод Assert (Утверждение) может проверять утверждения. Свойство AutoFlush (Автоматическая дозапись) и метод Flush (Дозапись) управляют дозаписью буфера вывода.

Использование переключателей для активизации диагностики

Переключатели позволяют проводить мелкоструктурное управление выводом диагностики. Можно использовать класс BooleanSwitch для включения и отключения вывода, основываясь на его свойстве Enabled (Разблокировано).

Класс TraceSwitch предоставляет пять иерархических уровней управления для своего свойства Level (Уровень): TraceLevel: :Error (Ошибка), TraceLevel: -.Warning (Предупреждение), TraceLevel: :Info (Информация), TraceLevel: :Verbose (Подробно) и TraceLevel: :0f f (Выключено). Эти значения входят в состав перечисления TraceLevel. Установка одного из уровней подразумевает, что все более высокие уровни установлены. Например, если установлен уровень в TraceLevel: :Warning (Предупреждение), то считается, что активизированы уровни TraceLevel: :Error (Ошибка) и TraceLevel: :Warning (Предупреждение).

DebugBooleanSwitch->Enabled = true; // истина Debug::WriteLinelf(

DebugBooleanSwitch->Enabled,

"Debug Boolean Switch enabled!");

TraceLevelSwitch->Level = TraceLevel::Warning; // Уровень = Предупреждение

Trace::WriteLinelf(

TraceLevelSwitch->TraceError, "TraceError!"); Trace::WriteLinelf(

TraceLevelSwitch->TraceWarning, "TraceWarning!"); Trace::WriteLinelf(

TraceLevelSwitch->TraceInfo, "InfoMessage!"); Trace::WriteLinelf(

TraceLevelSwitch->TraceVerbose, "VerboseMessage!");

Конструкторы для этих переключателей принимают два параметра. Первый из них — это имя переключателя, а второй — текстовое описание переключателя. Классы BooleanSwitch и TraceSwitch являются производными от абстрактного класса Switch (Переключатель). Наследованием из класса Switch (Переключатель) можно создать и собственные классы переключателей. Заметим, что свойство Enabled (Разблокировано) класса BooleanSwitch и свойство Level (Уровень) класса TraceSwitch не являются членами класса Switch (Переключатель).

Активация и деактивация переключателей

Для того чтобы изначально активировать и деактивировать переключатели, можно использовать установки файла конфигурации приложения. Это можно также выполнить программным путем.

Установка переключателей в файле конфигурации

Начальные значения переключателей можно установить в файле конфигурации приложения следующим образом:

<configuration> <!-- конфигурация -->

<system.diagnostics

<switches> <!-- переключатели -->

<add name="DebugSwitch" value = "0" />

<!-- добавить имя = "DebugSwitch" значение = "О" -->

<add name="TraceSwitch" value = "2" />

<!-- добавить имя = "TraceSwitch" значение = "2" -->

</switches> <!-- переключатели -->

</system.diagnostics>

</configuration> <!-- конфигурация -->

Если в файле конфигурации эти начальные значения не указаны явно, то свойство Enabled (Разблокировано) класса DebugSwitch устанавливается равным false (ложь), а свойство Level (Уровень) класса TraceSwitch устанавливается равным Off (Выключено).

Установка переключателей программным путем

Свойство Enabled (Разблокировано) класса BooleanSwitch может иметь значение true (истина) или false (ложь). Свойство Level (Уровень) класса TraceSwitch может принимать одно из значений списка TraceLevel: TraceLevel: : Error (Ошибка), TraceLevel:: Warning (Предупреждение), TraceLevel:: Info (Информация), TraceLevel: :Verbose (Подробно), TraceLevel: :0ff (Выключено). Можно всегда получить уровень этих установок класса TraceSwitch с помощью просмотра свойств TraceError,TraceWarning,Tracelnfo и TraceVerbose.

Использование переключателей для управления выводом

Можно проверить установленные значения переключателей перед тем, как заниматься выводом отладки и трассировки. Это выполняется с помощью условного оператора if или путем передачи значения переключателя в качестве параметра одному из методов классов Trace (Трассировка) или Debug (Отладка).

Trace::WriteLinelf(

TraceLevelSwitch->TraceError, "TraceError!"); Trace::WriteLinelf(

TraceLevelSwitch->TraceWarning, "TraceWarning!"}; Trace::WriteLinelf(

TraceLevelSwitch->TraceInfo, "InfoMessage!"); Trace::WriteLineIf(

TraceLevelSwitch->TraceVerbose, "VerboseMessage!");

Так как эти значения могут устанавливаться не только программой, можно решать, при каких обстоятельствах устанавливать конкретный уровень детализации вывода трассировки или отладки. Например, можно включить уровень TraceVerbose для вывода в том случае, если требуется действительно наивысший уровень диагностики, и установить уровень равным Off (Выключено) после того, как проблема будет локализована.

Класс TraceListener

Классы, производные от абстрактного класса TraceListener, занимаются назначениями вывода диагностики. Класс TextWriterTraceListener разработан для прямого вывода в TextWriter, Stream (Поток) или в FileStream. Поток Console: :0ut является примером часто используемого потока вывода. Класс EventLogTraceLis-tener позволяет направлять вывод в журнал событий EventLog. Можно создать собственный журнал событий с помощью статического метода CreateEventSource класса EventLog. DefaultTraceListener посылает вывод в окно отладки. По умолчанию вывод отладки можно увидеть в окне Output (Вывод) среды разработки Visual Studio .NET, а также с помощью утилит, таких как DBMon, которая включена в настоящий проект. При желании можно настроить место появления вывода с помощью собственного класса, управляемого из TraceListener.

Коллекция слушателей

Классы Trace (Трассировка) и Debug (Отладка) имеют статическую коллекцию слушателей, называемую TraceListenerCollection. Эта коллекция представляет собой список объектов TraceListener, которые получают информацию, выводимую классами Debug (Отладка) и Trace (Трассировка). Слушатели добавляются в коллекцию и удаляются из нее подобно тому, как это делается для любой другой коллекции .NET.

// создать слушатель, который пишет на пульт (console)

TextWriterTraceListener \*ConsoleOutput = new

TextWriterTraceListener(

Console::0ut, "Console::Out Listener"); // Слушатель

Trace::Listeners->Add(ConsoleOutput); // Слушатели->Добавить

// создать слушатель, который пишет в текстовый файл

Stream \*OutputFile = File::Create("output.txt"); // Файл:: Создать

TextWriterTraceListener \*OutputFileListener = new

TextWriterTraceListener(

OutputFile,

"Output File Listener");

// "Слушатель Выходного файла");

Trace::Listeners->Add(OutputFileListener); // Слушатели->Добавить

// удалить заданный по умолчанию (default) слушатель

Trace::Listeners->Remove("Default"); // Слушатели->Удалить ("по умолчанию")

В этом фрагменте программы объект OutputFileListener будет посылать вывод трассировки в файл с именем output.txt. Слушатель по умолчанию Def aultTraceListener добавляется в коллекции Listener (Слушатель) автоматически. Любой из слушателей, в том числе и предлагаемый по умолчанию, может быть удален из коллекции с помощью вызова метода Remove (Удалить) этой коллекции. Чтобы составить список всех слушателей в коллекции, можно выполнить следующий фрагмент кода:

pEnum =

Trace::Listeners->GetEnumerator(); // Слушатели

while (pEnum->MoveNext())

{

TraceListener \*tr =

dynamic\_cast<TraceListener \*>(pEnum->Current);

Console::WriteLine(

String::Format("\t{0}", tr->Name)); // Строка:: Формат

}

Резюме

Добавление в приложения инструментальных средств для различных уровней детализации вывода отладки и диагностики является общей задачей программирования. Классы диагностики являются примером того, как же в среде .NET для решения стандартных задач профаммирования реализуются классы, чтобы программист мог сконцентрироваться именно на логике самой программы, а не на создании инфраструктуры отладки. С другой стороны, это также является примером продуманного разделения классов .NET, которое позволяет настраивать инфраструктуру таким образом, чтобы программист мог использовать минимум или максимум доступных функций, в зависимости от его потребностей.

м