DevLib User Guide

|  |  |
| --- | --- |
| Version | 1.4 |
| Date | 2014-09-23 |
| Author | Yu Guan |

**CHANGE HISTORY**

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Description |
| 1.0 | 2014-08-27 | Yu Guan | Initial first version of user guide document, based on DevLib v.1.20.14; finish DevLib.Logging chapter. |
| 1.1 | 2014-09-08 | Yu Guan | based on DevLib v.1.21.\*; add DevLib.Csv chapter; add DevLib.Diagnostics chapter; |
| 1.2 | 2014-09-11 | Yu Guan | add DevLib.IO chapter; add DevLib.Ioc chapter; add DevLib.Xml chapter; add DevLib.DesignPatterns; |
| 1.3 | 2014-09-17 | Yu Guan | add DevLib.ServiceModel chapter; add DevLib.ServiceProcess chapter; |
| 1.4 | 2014-09-23 | Yu Guan | based on DevLib v.1.22.\*; |

Contents

[Introduction 1](#_Toc399251016)

[1. Overview 2](#_Toc399251017)

[1.1 Why DevLib 2](#_Toc399251018)

[1.2 Added Value 2](#_Toc399251019)

[1.3 Features 2](#_Toc399251020)

[1.4 Current Restrictions 3](#_Toc399251021)

[2. Installation 4](#_Toc399251022)

[2.1 Use dlls directly 4](#_Toc399251023)

[2.2 Build source code 4](#_Toc399251024)

[Architecture Guide 5](#_Toc399251025)

[3. Architecture Guide 6](#_Toc399251026)

[3.1 Design Principles 8](#_Toc399251027)

[3.1.1 Single Responsibility Principle (SRP) 8](#_Toc399251028)

[3.1.2 Open Close Principle (OCP) 8](#_Toc399251029)

[3.1.3 Liskov Substitution Principle (LSP) 8](#_Toc399251030)

[3.1.4 Interface Segregation Principle (ISP) 8](#_Toc399251031)

[3.1.5 Dependency Inversion Principle (DIP) 8](#_Toc399251032)

[3.1.6 Don’t repeat yourself (DRY) 9](#_Toc399251033)

[3.1.7 You ain't gonna need it (YAGNI) 9](#_Toc399251034)

[Development Guide 10](#_Toc399251035)

[4. Development Guide 11](#_Toc399251036)

[4.1 DevLib.AddIn 11](#_Toc399251037)

[4.2 DevLib.Compression 11](#_Toc399251038)

[4.3 DevLib.Configuration 11](#_Toc399251039)

[4.4 DevLib.Csv 12](#_Toc399251040)

[4.4.1 Quick Start 12](#_Toc399251041)

[4.4.2 CsvDocument 12](#_Toc399251042)

[4.5 DevLib.DaemonProcess 14](#_Toc399251043)

[4.6 DevLib.Data 14](#_Toc399251044)

[4.7 DevLib.DesignPatterns 15](#_Toc399251045)

[4.7.1 Dispose Pattern 15](#_Toc399251046)

[4.7.2 Singleton Pattern 16](#_Toc399251047)

[4.7.3 Producer-Consumer Pattern 16](#_Toc399251048)

[4.8 DevLib.Diagnostics 21](#_Toc399251049)

[4.8.1 Benchmark 21](#_Toc399251050)

[4.8.2 RetryAction 22](#_Toc399251051)

[4.8.3 DiagnosticsUtilities 22](#_Toc399251052)

[4.9 DevLib.Dynamic 23](#_Toc399251053)

[4.10 DevLib.Expressions 23](#_Toc399251054)

[4.11 DevLib.ExtensionMethods 23](#_Toc399251055)

[4.12 DevLib.Input 23](#_Toc399251056)

[4.13 DevLib.IO 24](#_Toc399251057)

[4.13.1 SyncSerialPort 24](#_Toc399251058)

[4.14 DevLib.Ioc 29](#_Toc399251059)

[4.14.1 Create a container 29](#_Toc399251060)

[4.14.2 Register 29](#_Toc399251061)

[4.14.3 Resolve 29](#_Toc399251062)

[4.14.4 Unregister 30](#_Toc399251063)

[4.15 DevLib.Logging 32](#_Toc399251064)

[4.15.1 Quick Start 32](#_Toc399251065)

[4.15.2 Architecture 32](#_Toc399251066)

[4.15.3 LogManager 33](#_Toc399251067)

[4.15.4 Log levels 33](#_Toc399251068)

[4.15.5 Logger 33](#_Toc399251069)

[4.15.6 LoggerSetup 34](#_Toc399251070)

[4.15.7 Use case 35](#_Toc399251071)

[4.15.8 Benchmark and comparison 36](#_Toc399251072)

[4.15.9 Performance impact 39](#_Toc399251073)

[4.16 DevLib.Net 40](#_Toc399251074)

[4.17 DevLib.Options 40](#_Toc399251075)

[4.18 DevLib.Reflection 40](#_Toc399251076)

[4.19 DevLib.Serialization 40](#_Toc399251077)

[4.20 DevLib.ServiceModel 41](#_Toc399251078)

[4.20.1 Service Hosting 41](#_Toc399251079)

[4.20.2 Client Invoking 44](#_Toc399251080)

[4.20.3 WcfServiceType 48](#_Toc399251081)

[4.20.4 Sample 50](#_Toc399251082)

[4.21 DevLib.ServiceProcess 54](#_Toc399251083)

[4.21.1 Walkthrough 54](#_Toc399251084)

[4.21.2 Usage 57](#_Toc399251085)

[4.21.3 WindowsServiceSetup 58](#_Toc399251086)

[4.21.4 WindowsServiceBase 61](#_Toc399251087)

[4.21.5 WindowsServiceInstaller 63](#_Toc399251088)

[4.22 DevLib.TerminalServices 64](#_Toc399251089)

[4.23 DevLib.Timers 64](#_Toc399251090)

[4.24 DevLib.Utilities 64](#_Toc399251091)

[4.25 DevLib.WinForms 64](#_Toc399251092)

[4.26 DevLib.Xml 65](#_Toc399251093)

[4.26.1 XmlConverter 65](#_Toc399251094)

[5. Known Issues 66](#_Toc399251095)

[5.1 DevLib.DaemonProcess 66](#_Toc399251096)

[5.1.1 Get command line arguments 66](#_Toc399251097)

[Appendix 67](#_Toc399251098)

[6. References 68](#_Toc399251099)

Introduction

# Overview

DevLib is a collection of very reusable code and components. DevLib was initially designed through a lot of projects that I've worked on. Over time I accumulated a number of classes to handle various tasks and continue to add a number of items in my spare time. The library is written in C# and most of them support from .Net 2.0 to the latest .Net version.

## Why DevLib

Microsoft .Net Framework provides a number of built in classes and functions to help make a programmer's life easier, but in fact, they didn't think of everything. DevLib tries to fill in some of those gaps. It comes with a couple hundred extension methods, built in data types, components such as Logging system, etc. When you add it all up, DevLib is one of the largest set of utilities for .Net out there. It is time to boost your productivity.

## Added Value

DevLib has high-quality codes which exhibit the following characteristics:

1. **Understandable.** They must be clearly readable and straightforward. They must showcase the key things they’re designed to demonstrate. The relevant parts of codes should be easy to reuse. They should not contain unnecessary code. They must include appropriate documentation.
2. **Correct.** They must demonstrate properly how to perform the key things they are designed to teach. They must compile cleanly, run correctly as documented, and be tested.
3. **Consistent.** They should follow consistent coding style and layout to make the code easier to read. Likewise, codes should be consistent with each other to make them easier to use together. Consistency shows craftsmanship and attention to detail.
4. **Modern.**  They should demonstrate current practices such as use of Unicode, error handling, defensive programming, and portability. They should use current recommendations for runtime library and API functions. They should use recommended project & build settings.
5. **Safe.**  They must comply with legal, privacy, and policy standards. They must not demonstrate hacks or poor programming practices. They must not permanently alter machine state. All installation and execution steps must be reversible.
6. **Secure.** They should demonstrate how to use secure programming practices such as least privilege, secure versions of runtime library functions, and SDL-recommended project settings.

The proper use of programming practices, design, and language features determines how well codes can achieve these. DevLib source codes demonstrate “best practices” for projects.

## Features

The following list provides a rough overview of the features provided by DevLib:

|  |  |  |
| --- | --- | --- |
| Assembly | minimum support .Net Framework | Description |
| DevLib.AddIn | v2.0 | load and run objects in an isolated process |
| DevLib.Compression | v2.0 | zip/unzip files |
| DevLib.Configuration | v2.0 | deal with ini/xml/config configuration file |
| DevLib.Csv | v2.0 | deal with csv file and data |
| DevLib.DaemonProcess | v2.0 | protect process and prevent process been killed |
| DevLib.Data | v2.0 | sql helper; repository; |
| DevLib.DesignPatterns | v2.0 | design patterns implementation |
| DevLib.Diagnostics | v2.0 | performance measurement; retry action; |
| DevLib.Dynamic | v4.0 | dynamic feature for xml/json |
| DevLib.Expressions | v4.0 | more feature for Expressions |
| DevLib.ExtensionMethods | v3.5 | extension methods for .Net Framework |
| DevLib.Input | V2.0 | keyboard/mouse input |
| DevLib.IO | v2.0 | IO related like: serial port |
| DevLib.Ioc | v2.0 | inversion of control/container |
| DevLib.Logging | v2.0 | logging system |
| DevLib.Net | v2.0 | network related like: ftp/ async socket |
| DevLib.Options | v2.0 | parse command line arguments |
| DevLib.Reflection | v2.0 | reflection related |
| DevLib.Serialization | v2.0 | serialization/deserialization |
| DevLib.ServiceModel | v3.0 | wcf service host; wcf client proxy |
| DevLib.ServiceProcess | v2.0 | windows service related |
| DevLib.TerminalServices | v2.0 | terminal session related |
| DevLib.Timers | v2.0 | more feature for timer |
| DevLib.Utilities | v2.0 | utilities class |
| DevLib.WinForms | v3.5 | win form related |
| DevLib.Xml | v2.0 | more feature for xml |

## Current Restrictions

DevLib is currently available for:

* .NET Framework 2.0 and later
* C#

# Installation

DevLib dlls can be downloaded manually from CodePlex (https://devlib.codeplex.com/releases/) or may be built locally based on the source published on CodePlex (https://devlib.codeplex.com/).

## Use dlls directly

If you downloaded DevLib dlls from CodePlex (https://devlib.codeplex.com/releases/), you can use those dlls as reference in your projects. You can chose one or more to add, DevLib dlls have no dependency to each other.

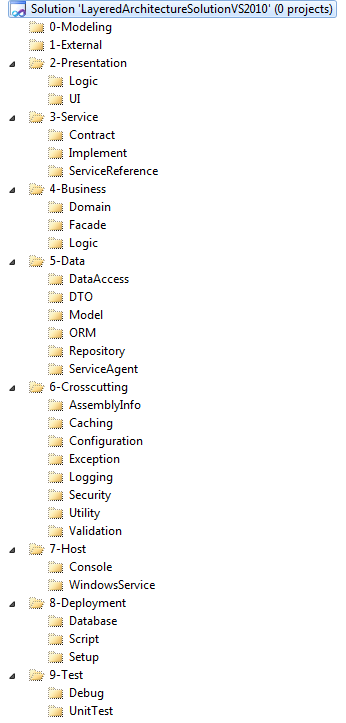
## Build source code

You can download DevLib source code from CodePlex (https://devlib.codeplex.com/SourceControl/latest/) and build it locally. The solution file is created by Visual Studio 2010, but the solution could be opened by Visual Studio later version like Visual Studio 2012/2013, etc. You can build the whole solution or just build each singular project you want. Then you can go to the build result folder and use the dll. Or you can use each singular project to your projects as reference.

Architecture Guide

# Architecture Guide

DevLib provides Visual Studio solution file template based on layered architecture, you can copy it and rename as your own solution. The solution folder structure show as below:





## Design Principles

Design principles describe fundamental ideas and are basic guidelines in software design.

The SOLID principles are the first five principles in object-oriented design (OOD) (Martin 2005). The following sections describe these and other principles in regards to DevLib.

### Single Responsibility Principle (SRP)

*A class should have one, and only one, reason to change.*

The “S” in SOLID requests for high cohesion at class level.

For more information see <http://en.wikipedia.org/wiki/Single_responsibility_principle>.

### Open Close Principle (OCP)

*You should be able to extend a classes behavior, without modifying it.*

The “O” in SOLID advocates extensible classes which allow for concrete or modified behavior in derived types. DevLib provides numerous interfaces and abstract classes which generated parts derive from. By providing partial classes to those generated parts you are able to override protected methods in order to implement custom requirements without the need to modify DevLib base classes.

For more information see <http://en.wikipedia.org/wiki/Open/closed_principle>.

### Liskov Substitution Principle (LSP)

*Derived classes must be substitutable for their base classes.*

The “L” in SOLID is commonly the most difficult principle in this series to understand. While at a first glance it seems to be natural that in place of a base class one of its derived types may be used in any place, it actually is not. When deriving a class one has to be very careful to make sure to not violate the base class’s invariants, which means not to strengthen any preconditions, and not to weaken and post conditions.

For more information see <http://en.wikipedia.org/wiki/Liskov_substitution_principle>.

### Interface Segregation Principle (ISP)

*Make fine grained interfaces that are client specific.*

The “I” in SOLID targets high cohesion and loos coupling by reducing interfaces to their minimum. Similar to the single responsibility principle for classes an interface should serve a single responsibility or purpose. If too many concerns get packet into a single interface, the interface becomes hard to implement and implementing classes are difficult to maintain.

For more information see <http://en.wikipedia.org/wiki/Interface_segregation_principle>.

### Dependency Inversion Principle (DIP)

*Depend on abstractions, not on concretions.*

The “D” in SOLID suggests relying on abstractions instead of concrete implementations in order to achieve looser coupling. Decoupling and reusability is improved by separating high-level and low-level components into separate components. Abstract types instead of concrete implementations are used when holding references to each other. Different patterns like plugin, service locator, dependency injection, adapter, and factory help to implement concrete solution following this principle.

DevLib applies this principle in many circumstances by providing interfaces and abstract types in its base library.

For more information see <http://en.wikipedia.org/wiki/Dependency_inversion_principle>.

### Don’t repeat yourself (DRY)

The DRY principle suggest in its purest form for any given aspect to have only one single place of truth. When or wherever an aspect is reused, it must not be re-implemented but has to rely on the single place of truth.

Applying this principle DevLib reaches a high degree of automaton during development by generating server, client, and common components.

For more information see <http://en.wikipedia.org/wiki/Don't_repeat_yourself>.

### You ain't gonna need it (YAGNI)

The YAGNI principle represents kind of a contrast to other principles and suggests to only add functionality if it is required. This principle has to be balanced against others to reach an optimal cost-benefit ratio. As this principle in its purest form doesn’t take mid-term and long-term architectural considerations into account, it should be used with caution when taken into consideration. While this principle may often not be applied entirely, it is still a reminder that every design decision should be taken according to requirements. Adherence to principles and implementation of patterns should not be done to fulfill a self-purpose but to provide business value.

For more information see <http://en.wikipedia.org/wiki/You_ain't_gonna_need_it>.

Development Guide

# Development Guide

This chapter explains step-by-step how to use DevLib components.

## DevLib.AddIn

## DevLib.Compression

## DevLib.Configuration

## DevLib.Csv

DevLib.Csv is a lightweight and fast Csv parser to parse .csv (delimited file) and write Csv. The primary features of DevLib.Csv are:

1. Load csv file/text/stream to DataTable/memory.
2. Add/Delete rows; Add/Delete columns; Modify cells.
3. Write data to csv file/text/stream.
4. Load/Write csv with customized delimiter and quote character.

### Quick Start

CsvDocument csv = new CsvDocument();

csv.Load(@"c:\test.csv");

……

csv[0, 0] = "hello";

csv.Save(@"c:\new.csv");

### CsvDocument

CsvDocument is a class represents a CSV document. You can load csv file/text/stream to the instance of CsvDocument. Then you can handle the csv data like a DataTable, you can add/delete rows or columns, you can modify cells’ contents. You can write the data to csv file/text/stream.

Load csv from file “c:\test.csv” with has header true, customized delimiter ‘;’ and customized quote ‘~’:

CsvDocument csv = new CsvDocument();

csv.Load(@"c:\test.csv", true, ';', '~');

Get the DataTable represents the csv data:

DataTable dataTable = csv.Table;

Delete the second row and add a column called “NewColumn”:

csv.Table.Rows.RemoveAt(1);

csv.Table.Columns.Add("NewColumn");

Get all columns name:

List<string> headers = csv.ColumnNames;

Get rows count and columns count:

int rowCount = csv.RowCount;

int columnCount = csv.ColumnCount;

Retrieve data:

string cellAtRow0Column1 = csv[0, 1];

string cellAtRow0ColumnNameA = csv[0, "A"];

DataRow row0 = csv[0];

DataColumn columnA = csv["A"];

Write data back to a new csv file “c:\new.csv”, with overwrite true, append content false, has header true, quote all cells false, customized delimiter ‘,’, customized quote ‘”’ and customized new line string "\r\n" for non-Unix platforms, or "\n" for Unix platforms:

csv.Save(@"c:\new.csv", true, false, true, false, ',', '"', Environment.NewLine);

## DevLib.DaemonProcess

## DevLib.Data

## DevLib.DesignPatterns

DevLib.DesignPatterns provide standardized and efficient solutions to software design and programming problems that are re-usable in your code. Software Architects and developers use them to build high quality robust applications.

### Dispose Pattern

The Dispose Pattern is intended to standardize the usage and implementation of finalizers and the IDisposable interface.

The main motivation for the pattern is to reduce the complexity of the implementation of the Finalize and the Dispose methods. The complexity stems from the fact that the methods share some but not all code paths (the differences are described later in the chapter). In addition, there are historical reasons for some elements of the pattern related to the evolution of language support for deterministic resource management.

/// <summary>

/// Sample code for dispose pattern.

/// </summary>

public class DisposePattern : IDisposable

{

/// <summary>

/// Field \_disposed.

/// </summary>

private bool \_disposed = false;

/// <summary>

/// Initializes a new instance of the <see cref="DisposePattern" /> class.

/// </summary>

public DisposePattern()

{

}

/// <summary>

/// Finalizes an instance of the <see cref="DisposePattern" /> class.

/// </summary>

~DisposePattern()

{

this.Dispose(false);

}

/// <summary>

/// Releases all resources used by the current instance of the <see cref="DisposePattern" /> class.

/// </summary>

public void Close()

{

this.Dispose();

}

/// <summary>

/// Releases all resources used by the current instance of the <see cref="DisposePattern" /> class.

/// </summary>

public void Dispose()

{

this.Dispose(true);

GC.SuppressFinalize(this);

}

/// <summary>

/// Releases all resources used by the current instance of the <see cref="DisposePattern" /> class.

/// protected virtual for non-sealed class; private for sealed class.

/// </summary>

/// <param name="disposing">true to release both managed and unmanaged resources; false to release only unmanaged resources.</param>

protected virtual void Dispose(bool disposing)

{

if (this.\_disposed)

{

return;

}

this.\_disposed = true;

if (disposing)

{

// dispose managed resources

////if (managedResource != null)

////{

//// managedResource.Dispose();

//// managedResource = null;

////}

}

// free native resources

////if (nativeResource != IntPtr.Zero)

////{

//// Marshal.FreeHGlobal(nativeResource);

//// nativeResource = IntPtr.Zero;

////}

}

/// <summary>

/// Method CheckDisposed.

/// </summary>

private void CheckDisposed()

{

if (this.\_disposed)

{

throw new ObjectDisposedException("DevLib.DesignPatterns.DisposePattern");

}

}

}

### Singleton Pattern

Often used for objects that need to provide global access with the constraint of only one single instance in the application. DevLib. DesignPatterns.Singleton<T> provides a thread-safe without using locks and generic type implementation.

MyClass globalObj = Singleton<MyClass>.Instance;

### Producer-Consumer Pattern

The producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time.

DevLib.DesignPatterns.ProducerConsumer is a thread-safe and concurrent solution for producer–consumer problem. It supports multi producers put data to queue at the same time and multi consumers to get data from the queue at the same time.

DevLib.DesignPatterns.ProducerConsumer has two implementations: **ProducerConsumer<T>** supports generic type. **ProducerConsumer** supports System.Object. Here take **ProducerConsumer<T>** for example, **ProducerConsumer** is similar.

#### Create instance

public ProducerConsumer(Action<T> consumerAction, int consumerThreads = 1, bool startNow = true)

**consumerAction** is a delegate that will be executed when the consumer thread processes a data item.

**consumerThreads** is the number of consumer threads.

**startNow** is whether to start the consumer thread immediately.

public ProducerConsumer(IProducerConsumerQueue<T> queue, Action<T> consumerAction, int consumerThreads = 1, bool startNow = true)

**queue** is an instance of IProducerConsumerQueue<T>.

**consumerAction** is a delegate that will be executed when the consumer thread processes a data item.

**consumerThreads** is the number of consumer threads.

**startNow** is whether to start the consumer thread immediately.

Once you have ProducerConsumer instance, it is ready to use.

#### Start the consumer thread

public void Start()

If in constructor you set startNow to true, the consumer thread already started and you don’t need to call Start again.

Note: the consumer thread start or not doesn’t affect producer, so even if ProducerConsumer instance didn’t start, the producer still can put data to the queue.

#### Stop the consumer thread

public void Stop(bool enableQueue = true, bool clearQueue = false)

**enableQueue** is producer still can add data to the queue when stopped or not.

**clearQueue** is whether to clear the queue when stop.

Note: if Stop with enableQueue false, the producer cannot put data to the queue until call Start again

#### Produce

Producer can call Enqueue method to put data to the queue.

public void Enqueue(T item)

item is the data item produced by producer.

public void Enqueue(IEnumerable<T> items)

items is a batch data items produced by producer.

#### Clear

Clear all data items from the queue.

public void Clear()

#### Properties

Gets the current number of items contained in the queue.

public long QueueCount { get }

Gets the accumulation count of produced items.

public long ProduceAccumulation { get }

Gets the accumulation count of consumed items.

public long ConsumeAccumulation { get }

Gets a value indicating whether all consumer threads are idle or not. Only when queue is empty and all threads are running and idle return true; otherwise, false.

public bool IsIdle { get }

Gets a value indicating whether the consumer thread is running.

public bool IsRunning { get }

#### IProducerConsumerQueue and ProducerConsumerQueue

**IProducerConsumerQueue** is an interface define a queue can be used for ProducerConsumer. It has both normal define and generic type define.

**ProducerConsumerQueue** is an implementation of **IProducerConsumerQueue** provided by DevLib. DesignPatterns namespace. It has both normal class and generic type class. It is based on .Net Framework Queue in memory.

You can use your own implementation of **IProducerConsumerQueue** to do some customized job, such as persistence queue based on file system or database; MSMQ;

For example:

public class MyDatabaseQueue<T> : IProducerConsumerQueue<T>

{

public MyDatabaseQueue()

{

// open connection to database.

}

public void Enqueue(T item)

{

// insert item to database.

}

public long Enqueue(IEnumerable<T> items)

{

// insert items to database, and return the number of items be inserted.

return items.LongCount();

}

public T Dequeue()

{

// query one item and remove from database.

throw new NotImplementedException();

}

public T Peek()

{

// query one item and keep it from database.

throw new NotImplementedException();

}

public bool Contains(T item)

{

// query item is in database or not.

throw new NotImplementedException();

}

public long Count()

{

// query total rows in database.

throw new NotImplementedException();

}

public void Enqueue(object item)

{

// insert item to database.

}

public long Enqueue(IEnumerable items)

{

// insert items to database, and return the number of items be inserted.

throw new NotImplementedException();

}

object IProducerConsumerQueue.Dequeue()

{

// query one item and remove from database.

throw new NotImplementedException();

}

object IProducerConsumerQueue.Peek()

{

// query one item and keep it from database.

throw new NotImplementedException();

}

public bool Contains(object item)

{

// query item is in database or not.

throw new NotImplementedException();

}

public void Clear()

{

// remove all rows in database.

throw new NotImplementedException();

}

}

Then you can use this class **MyDatabaseQueue<T>** as your customized queue for **ProducerConsumer<T>**

IProducerConsumerQueue<string> myQueue = new MyDatabaseQueue<string>();

ProducerConsumer<string> producerConsumer = new ProducerConsumer<string>(myQueue, item =>

{

// Consumer consumes item here.

}, 3, true);

## DevLib.Diagnostics

DevLib.Diagnostics namespace contains several diagnostic utilities such as: benchmark code snippet performance; retry mechanism wrap; get stack information utility, etc.

### Benchmark

Benchmark is a lightweight static class to provide benchmarking support to measure time used and GC used information to run code.

There is one method you can use to measure your code snippets.

public static BenchmarkResult Run(Action<int> action, int iteration = 1, string name = null, Action<string> outputAction = null)

**action** is the code snippets to run, it can be a delegate or anonymous method, parameter for the anonymous method is the index of iteration.

**iteration** is repeat times or your code snippets.

**name** is the name of current benchmark.

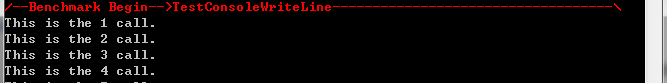
**outputAction** is the action to handle the benchmark test result string. by default is write to console.

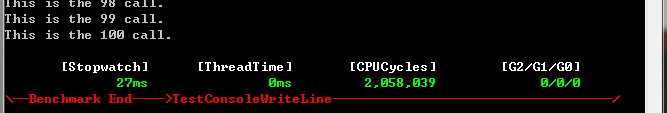
the return value is BenchmarkResult instance, include StopwatchElapsed, ThreadTimeElapsed, CPUCycles and GCCountArray.

For example, I want to measure the performance of running Console.WriteLine() 100 times:

Benchmark.Run(i => Console.WriteLine("This is the {0} call.", i), 100, "TestConsoleWriteLine");

and the output on the console is like this:





The lines in red color will be in a random color but matched.

Note: for more accurate measurement, you can run Initialize() just before your benchmark:

Benchmark.Initialize();

Benchmark.Run(i => Console.WriteLine("This is the {0} call.", i), 100, "TestConsoleWriteLine");

### RetryAction

RetryAction is a static class to provide retry mechanism to execute code snippets.

public static void Execute(Action<int> action, CatchAction<Exception, int> catchAction = null, int retryCount = 1, int retryInterval = 0)

**action** is the code snippets that will be invoked whenever a retry condition is encountered, parameter for the anonymous method is the index of retryCount.

**catchAction** is code snippets that will be invoked whenever an exception is caught, parameter for the anonymous method is the exception object and the index of retryCount.

**retryCount** is the number of subsequent retry attempts, not including the very first execution.

**retryInterval** is the time interval between retries in milliseconds.

For example, run DoSomthing(), this method may throw exception, if something is wrong, retry 3 times, between each retry wait 1 second.

RetryAction.Execute(i => DoSomthing(), (e, i) => Console.WriteLine("Count {0} failed: {1}", i.ToString(), e.ToString()), 3, 1000);

### DiagnosticsUtilities

DiagnosticsUtilities is a static class to provide several diagnostic method.

Write exception information to console.

public static void ConsoleOutputException(Exception exception)

Builds a readable representation of the method in which the frame is executing.

public static string GetStackFrameMethodInfo(StackFrame stackFrame)

Builds a readable representation of the stack trace.

public static string GetStackFrameInfo(int skipFrames)

## DevLib.Dynamic

## DevLib.Expressions

## DevLib.ExtensionMethods

## DevLib.Input

## DevLib.IO

DevLib.IO namespace contains components related to IO such as serial port.

### SyncSerialPort

SyncSerialPort is a class wraps the System.IO.Ports.SerialPort class and provides synchronous and asynchronous I/O mechanism to make reading/writing the data of serial port much easier.

#### Create instance

public SyncSerialPort(string portName = "COM1", int baudRate = 9600, Parity partity = Parity.None, int dataBits = 8, StopBits stopBits = StopBits.One)

**portName** is the port to use (for example, COM1).

**baudRate** is the baud rate.

**partity** is one of the System.IO.Ports.SerialPort.Parity values.

**dataBits** is the data bits value.

**stopBits** is one of the System.IO.Ports.SerialPort.StopBits values.

Example:

Create a SyncSerialPort instance by using serial port “COM3” with default serial port properties.

SyncSerialPort port = new SyncSerialPort("COM3");

#### Getting a List of Serial Ports

There is a static get only property you can call to get all serial ports on the local machine.

string[] ports = SyncSerialPort.PortNames;

#### Opening and Closing the SyncSerialPort

**Open**

Once you have a SyncSerialPort instance, you need to open it to be ready to read/write data.

SyncSerialPort port = new SyncSerialPort("COM3");

port.Open();

Open method will return true or false to indicate open operation succeeded or not. throwOnError indicates whether throw any exception that occurs or ignore any exception that occurs.

public bool Open(bool throwOnError = false)

**CurrentPort**

You can use the get only property CurrentPort to know which serial port is used by this instance.

string portName = port.CurrentPort;

**IsOpen**

You can use the get only property IsOpen to know if the serial port is already opened.

bool isOpen = port.IsOpen;

**Close**

Once you've finished working with the serial port you can Close it. Closing the serial port closes the handle associated with the driver and stops the I/O thread running in the background. You can no longer read/write data to the serial port and the write buffer is flushed. After closing the serial port, you may reopen it, you need to call Open again.

port.Close();

Close method will return true or false to indicate open operation succeeded or not. throwOnError indicates whether throw any exception that occurs or ignore any exception that occurs.

public bool Close(bool throwOnError = false)

**Dispose**

Disposing the SyncSerialPort instance will also close the serial port if it is currently open. Disposing the SyncSerialPort instance disallows it from being opened again in the future. All internal handles are cleaned up.

It is recommended that when you're finished using the serial port that you call Dispose.

port.Dispose();

or

using (SyncSerialPort port1 = new SyncSerialPort("COM1"))

{

port1.Open();

...

}

#### Reading and Writing Bytes

**Synchronous I/O**

Sync send a specified number of bytes to the serial port using data from a buffer.

public byte[] SendSync(byte[] buffer, bool waitTimeout = false, int timeout = 1000, bool throwOnError = false)

**buffer** is the byte array that contains the data to write to the port.

**waitTimeout** is whether read receive data after wait for timeout to expire or read on data received.

**timeout** is the number of milliseconds before a time-out occurs when a read operation does not finish.

**throwOnError** true to throw any exception that occurs.-or- false to ignore any exception that occurs.

return value is the byte array of received data.

Sync send a specified number of bytes to the serial port using data from a buffer.

public byte[] SendSync(byte[] buffer, int bytesToReceive, int timeout = 1000, bool throwOnError = false)

**buffer** is the byte array that contains the data to write to the port.

**bytesToReceive** is the number of bytes to read.

**timeout** is the number of milliseconds before a time-out occurs when a read operation does not finish.

**throwOnError** true to throw any exception that occurs.-or- false to ignore any exception that occurs.

return value is the byte array of received data.

Reads all bytes from the serial port input buffer.

public byte[] ReadSync(bool waitTimeout = false, int timeout = 1000, bool throwOnError = false)

**waitTimeout** is whether read receive data after wait for timeout to expire or read on data received.

**timeout** is the number of milliseconds before a time-out occurs when a read operation does not finish.

**throwOnError** true to throw any exception that occurs.-or- false to ignore any exception that occurs.

return value is the byte array of received data.

Reads all bytes from the serial port input buffer.

public byte[] ReadSync(int bytesToReceive, int timeout = 1000, bool throwOnError = false)

**bytesToReceive** is the number of bytes to read.

**timeout** is the number of milliseconds before a time-out occurs when a read operation does not finish.

**throwOnError** true to throw any exception that occurs.-or- false to ignore any exception that occurs.

return value is the byte array of received data.

**Asynchronous I/O**

Writes a specified number of bytes to the serial port using data from a buffer.

public void Send(byte[] buffer)

**buffer** is the byte array that contains the data to write to the port.

Writes a specified number of bytes to the serial port using data from a buffer.

public void Send(byte[] buffer, int offset, int count)

**buffer** is the byte array that contains the data to write to the port.

**offset** is the zero-based byte offset in the data parameter at which to begin copying bytes to the port.

**count** is the number of bytes to write.

#### Discard Buffer

Discards data from the serial driver's receive buffer.

public void DiscardInBuffer()

Discards data from the serial driver's transmit buffer.

public void DiscardOutBuffer()

#### Events

There are three events in the system, DataReceived, ErrorReceived and PinChanged. All events are programmed so they are serialized. Events do occur however on different threads to your main program, so if you are updating elements of your UI, you should be careful to marshal your UI updates via the Control.Invoke methods.

As a general guideline, you should not perform too much logic in your event handler code. While it will not stop reception of new data and you could theoretically spend a long period of time in your event (e.g. 2 seconds or so), no new event will be raised until your existing event handlers are finished.

**DataReceived**

When data is available to read in the read buffer, the DataReceived event is triggered.

**ErrorReceived**

On the reception of each character the serial port error status is checked. If there is an error detected, this event shall be raised.

**PinChanged**

If the driver detects a state change of the modem pins, an event is raised.

## DevLib.Ioc

DevLib.Ioc is a lightweight thread-safe Inversion of Control container. The primary features include:

1. Dependency resolution
2. Lifetime management (shared and non-shared copies)
3. Supports both constructor and property injection
4. Full control over registration - delete registrations and destroy shared copies as needed
5. Multiple resolution support through a simple label

### Create a container

To get started, you need to create an instance:

IocContainer container = new IocContainer();

or you can indicate whether to ignore case when resolve object by label or not.

IocContainer container = new IocContainer(true);

Default constructor doesn’t ignore case.

### Register

Register the implementation for a type. The registration can take instance of target type or take a delegate which is a function that returns an instance of the target type (this can be anything assignable to the base type, and may point to a factory method or even a static object if you want).

public IocContainer Register<T>(T instance, string label = null)

**T** is type of instance to register.

**instance** is object to return.

**label** is a unique label that allows multiple implementations of the same type.

return value is the current IocContainer instance.

public IocContainer Register<T>(CreationFunc creation, string label = null)

**T** is type of instance to register.

**creation** is delegate method to create a new instance.

**label** is a unique label that allows multiple implementations of the same type.

return value is the current IocContainer instance.

### Resolve

Resolve an instance of the requested type from the container.

public T Resolve<T>(bool createNew = false, string label = null)

**T** is the type to resolve.

**createNew** true to call delegate method to create a new instance; false to return a shared instance.

**label** is a unique label that allows multiple implementations of the same type.

return value is the retrieved object.

Check whether it is possible to resolve a type.

public bool CanResolve<T>(bool createNew = false, string label = null)

**T** is the type to resolve.

**createNew** true to check delegate method exists; false to check a shared instance exists.

**label** is a unique label that allows multiple implementations of the same type.

return true if can resolve; otherwise, false.

Try to resolve an instance of the requested type from the container.

public bool TryResolve<T>(out T instance, bool createNew = false, string label = null)

**T** is the type to resolve.

**instance** is the retrieved object, if it is possible to resolve one.

**createNew** true to call delegate method to create a new instance; false to return a shared instance.

**label** is a unique label that allows multiple implementations of the same type.

return true if resolve successfully; otherwise, false.

Resolve all from the container.

public bool CanResolve<T>(bool createNew = false, string label = null)

**T** is the type to resolve.

**createNew** true to check delegate method exists; false to check a shared instance exists.

**label** is a unique label that allows multiple implementations of the same type.

return true if can resolve; otherwise, false.

### Unregister

Unregister a type mapping with the container.

public bool Unregister<T>(bool instance = true, string label = null)

**T** is the type to unregister.

**instance** true to unregister the shared instance; false to unregister delegate method.

**label** is a unique label that allows multiple implementations of the same type.

return true if unregister successfully; otherwise, false.

Destroy a type mapping with the container.

public bool Destroy<T>()

**T** is the type to destroy.

return true if destroy successfully; otherwise, false.

Removes all values from container.

public void Clear()

## DevLib.Logging

DevLib.Logging is a lightweight logging component. The advantage is very limited impacting performance, very fast. DevLib.Logging support zero configuration or very simple configuration, also support emitting log to different file at runtime.

### Quick Start

LogManager.Open().Log("This is a log message.");

or

Logger logger = LogManager.Open();

logger.Log("This is a log message.");

logger.Log(LogLevel.ERRO, "This is an error!");

The log output format is:

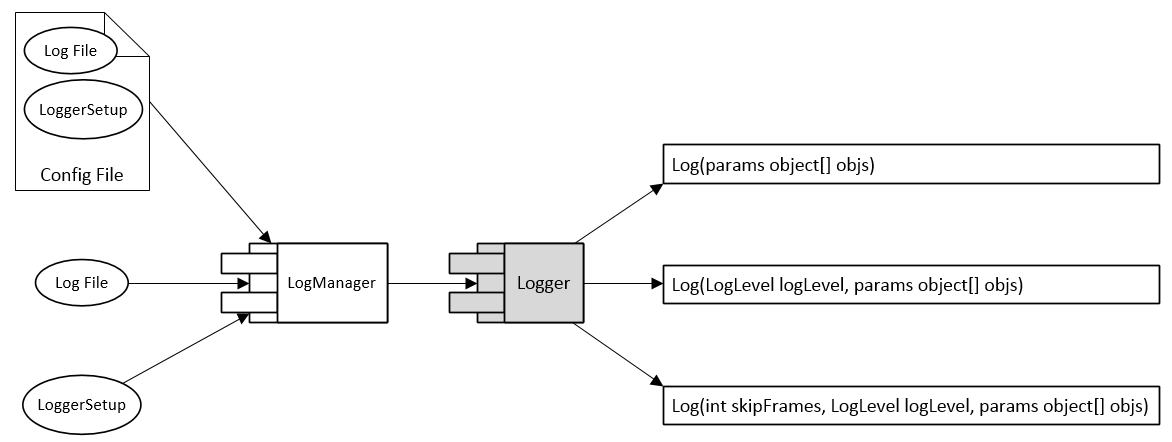
Timestamp|LogLevel|User|ThreadId| [Log Message] |StackInfo

Only [Log Message] is your own message, the rest info are fixed.

For example:

2014-08-28T11:21:57.2375349U-04:00|INFO|Administrator|001| [This is a log message.] |<TestLogging>b\_\_36 in Program.cs:234

### Architecture



In order to emit log messages from the application you need to use the logging API. There are two classes that you will be using the most: Logger and LogManager, both in the DevLib.Logging namespace. Logger has methods to emit log messages, and LogManager creates and manages instances of loggers.

### LogManager

LogManager is a factory class to create Logger instance, and Logger can only be created by this class.

There are several override methods to create Logger instance.

public static Logger Open(string logFile = null)

**logFile** specify which file you want Logger instance to write log message to. If it is null or empty, Logger will use current application process name + “.log” as default log file.

public static Logger Open(string logFile, LoggerSetup loggerSetup)

Logger instance will use **loggerSetup** settings to write log.

public static Logger Open(string logFile, string configFile)

Logger instance will use **logFile** as your log destination, and use loggerSetup settings defined in the **configFile**. If **configFile** also specify log file, it will be ignored.

public static Logger OpenConfig(string configFile = null)

Logger instance will use log file and loggerSetup defined in the **configFile**. If **configFile** doesn’t contain log file or loggerSetup, Logger will use default one.

### Log levels

Each log message has associated log level, which identifies how important/detailed the message is.

DevLib.Logging supports the following log levels:

**DBUG** - very detailed logs, which may include high-volume information such as protocol payloads. This log level is typically only enabled during development

**INFO** - information messages, which are normally enabled in production environment

**WARN** - warning messages, typically for non-critical issues, which can be recovered or which are temporary failures

**EXCP** - the thrown exception messages

**ERRO** - error messages

**FAIL** - very serious errors

### Logger

Logger is the class you can use to emit log messages. There are several override methods to write log.

public void Log(params object[] objs)

It will log the message as INFO level, and automatically call ToString() on each log objs.

public void Log(LogLevel logLevel, params object[] objs)

You can specify log level when write log.

public void Log(int skipFrames, LogLevel logLevel, params object[] objs)

**skipFrames** is the number of frames up the stack to skip. If you wrap Logger in your own class, you can use this to skip the wrapped class stack info.

### LoggerSetup

LoggerSetup defines the settings will be used by logger, defines logger’s behavior.

There are several setting properties:

public LogLevel Level {get; set;}

Indicating log level hierarchy write to log file. DBUG > INFO > WARN > EXCP > ERRO > FAIL  
Default value is DBUG, which means all log level message will be logged. If you set to WARN, it will only emit WARN, EXCP, ERRO and FAIL level messages.

public bool WriteToConsole {get; set;}

Indicating whether write log message to console.  
Default value is true

public bool WriteToFile {get; set;}

Indicating whether write log message to log file.  
Default value is true

public bool UseBracket {get; set;}

Indicating whether use square brackets ([ ]) around log message.  
Default value is true

public long RollingFileSizeMBLimit {get; set;}

Rolling log file size limit in MB. If less than or equal to zero, there is no limit.  
Default value is 10

public long RollingFileCountLimit {get; set;}

Rolling log file count limit. If less than zero, there is no limit. If equal to zero, there is no rolling log file.  
Default value is 10

bool RollingByDate {get; set;}

Indicating whether split log file by date.  
Default value is false

You can also define LoggerSetup settings in a configuration file with xml format. Below shows how to define in an xml file:

<LogConfig>

<LogFile>C:\test.log</LogFile>

<LoggerSetup>

<Level>DBUG</Level>

<WriteToConsole>true</WriteToConsole>

<WriteToFile>true</WriteToFile>

<UseBracket>true</UseBracket>

<RollingFileSizeMBLimit>10</RollingFileSizeMBLimit>

<RollingFileCountLimit>10</RollingFileCountLimit>

<RollingByDate>false</RollingByDate>

</LoggerSetup>

</LogConfig>

The node **<LogConfig />** can be putted anywhere in an xml file. The node **<LogFile />** could be null or empty.

### Use case

An application receives commands and do some following logic, those commands belong to a limited set of string. One requirement is according to different command string, write log to different log file, the log file name start with command string. At the same time, some log should go to a common log file.

For example: the commands set is {“openA”,”closeA”,”openB”,”closeB”,”doC”,”giveMeD”,…}, and if application receive command “openA”, then the log message should be wrote to “openA.MyApplication.log”, if application receive command “giveMeD”, then the log message will go to file “giveMeD.MyApplication.log”.

private Logger \_logger = LogManager.Open("Common.MyApplication.log");

private void DoLogicWhenReceiveCommand(string command)

{

this.\_logger.Log("DoLogicWhenReceiveCommand start");

string logFile = string.Format("{0}.MyApplication.log", command);

LogManager.Open(logFile).Log(LogLevel.DBUG, "receive command", command);

try

{

//Do the rest logic.

LogManager.Open(logFile).Log(LogLevel.INFO, "succeeded", command);

}

catch (Exception e)

{

LogManager.Open(logFile).Log(LogLevel.EXCP, "ops", e);

this.\_logger.Log(LogLevel.EXCP, "DoLogicWhenReceiveCommand error", e);

}

this.\_logger.Log("DoLogicWhenReceiveCommand end");

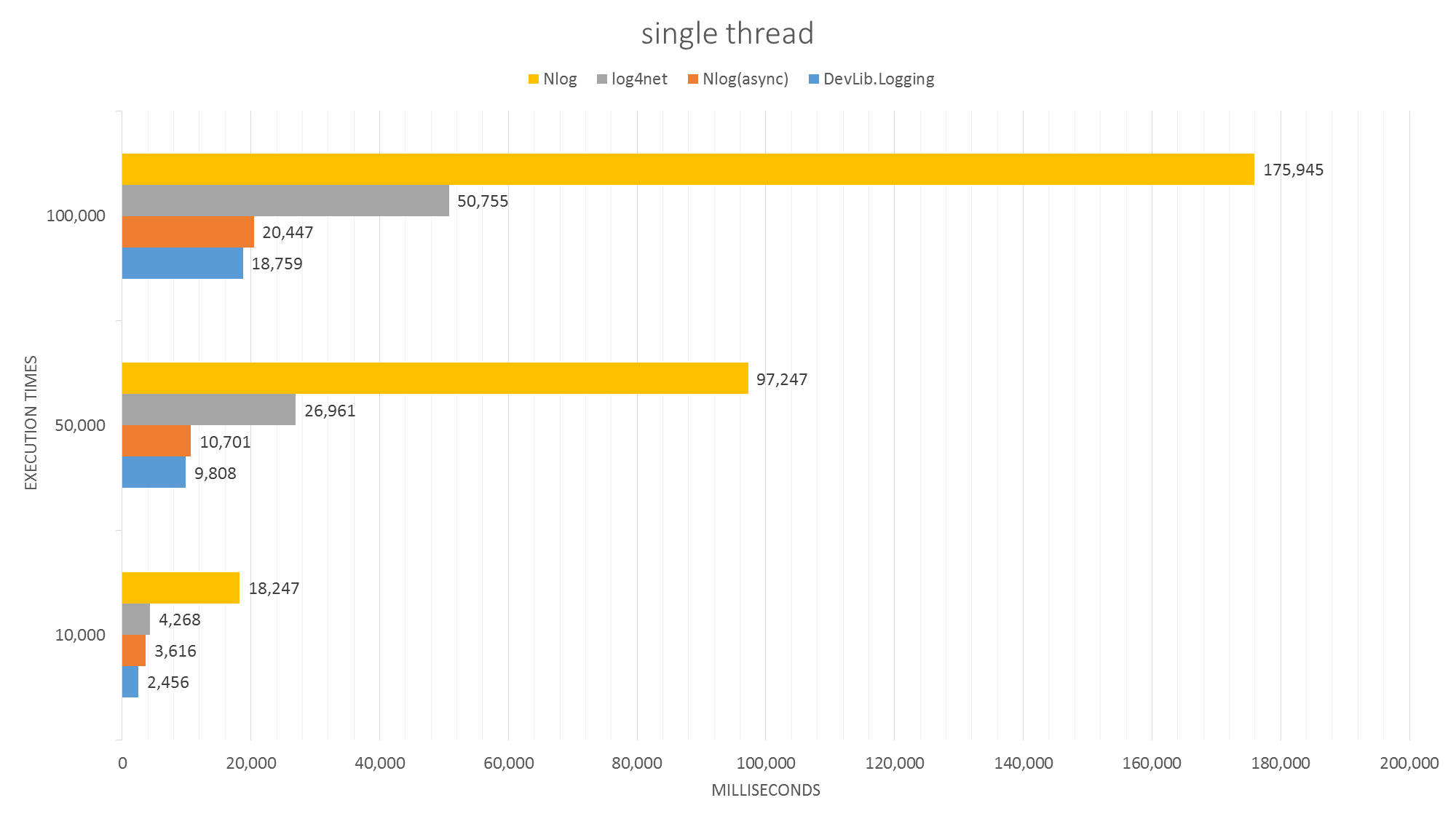
}

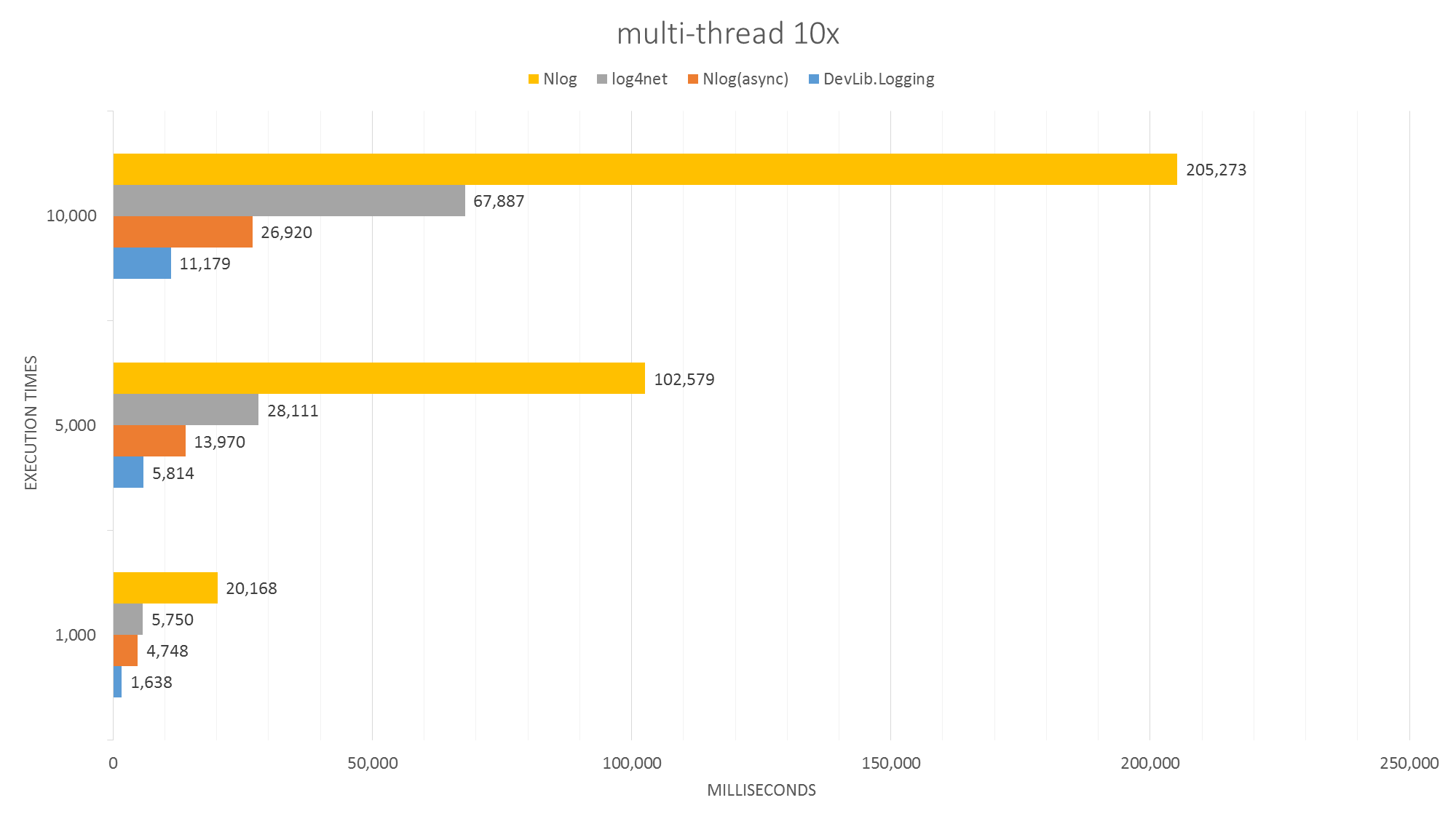
### Benchmark and comparison

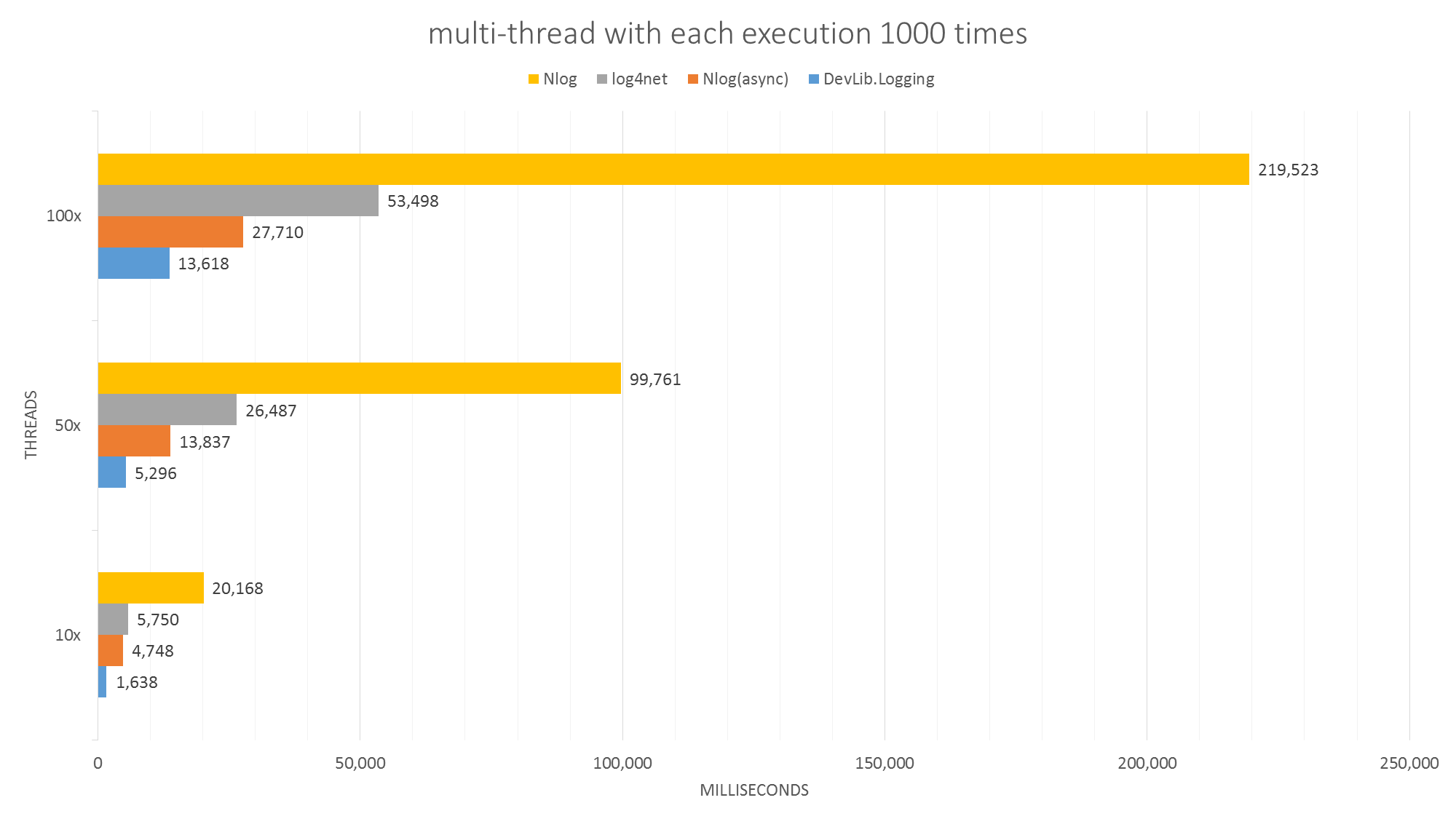
Here evaluated two most popular logging frameworks and compared the important features. All the benchmark results are based on DevLib.Logging v.1.20.\*; log4net v.1.2.13 (http://logging.apache.org/log4net/); NLog v.3.0 (http://www.nlog-project.org/)

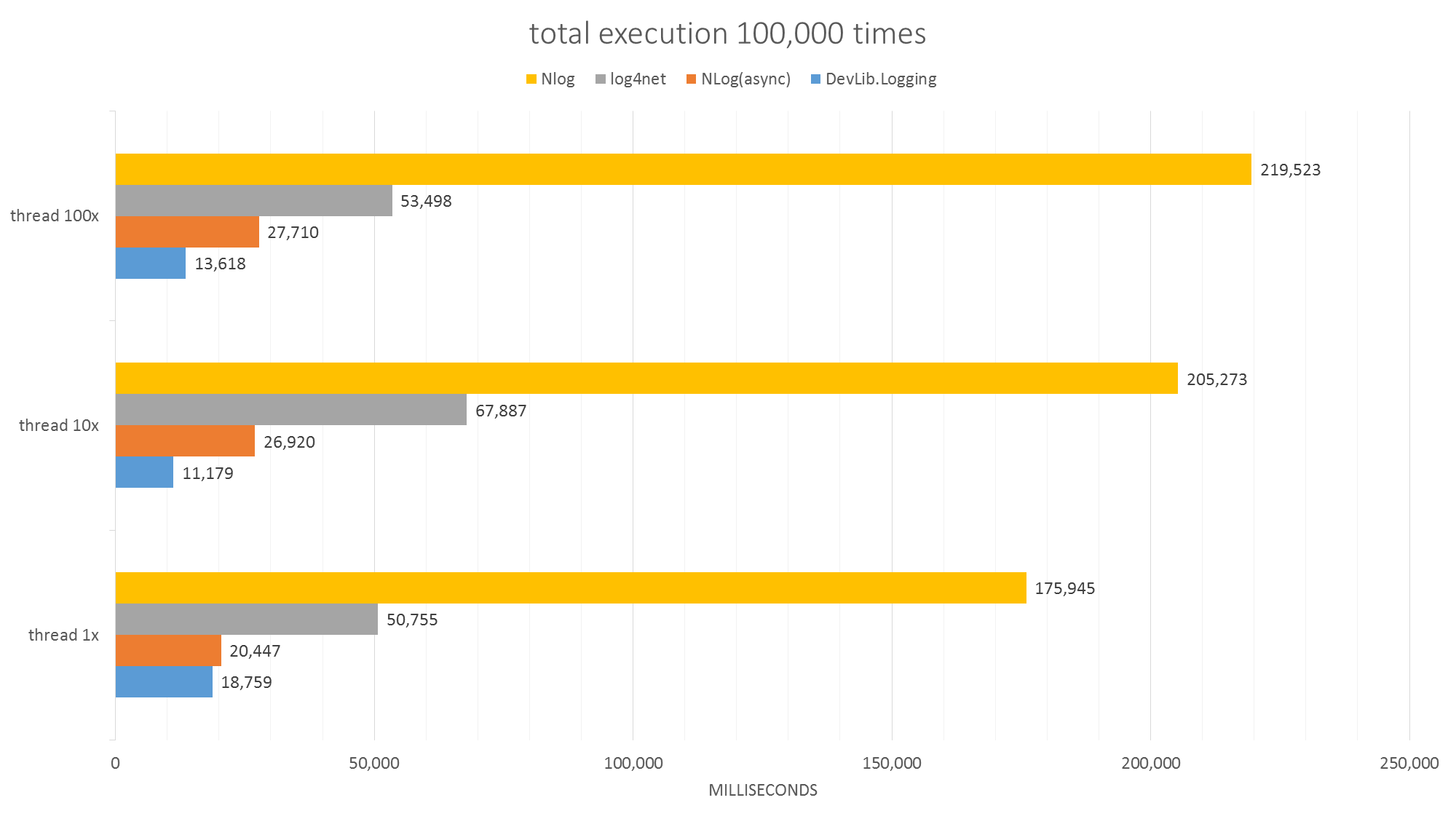
Please see below for the results of the comparison.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **DevLib.Logging** | **log4net** | **NLog** |
| License | [WTFPL](https://devlib.codeplex.com/SourceControl/latest#main/product/codes/DevLib.Main/License .txt) | [Apache](http://logging.apache.org/log4net/license.html) | [BSD](https://github.com/NLog/NLog/blob/master/LICENSE.txt) |
| Source available | √ | √ | √ |
| Vendor | Yu Guan | Apache | multi contributors |
| Website | [Link](https://devlib.codeplex.com/) | [Link](http://logging.apache.org/log4net/) | [Link](http://www.nlog-project.org/) |
| **Basic Features** | **DevLib.Logging** | **log4net** | **NLog** |
| Complexity of configuration | zero config / very easy (1 of 10) | very complex (9 of 10) | very complex (9 of 10) |
| Optional configuration files | √ | √ | √ |
| Log levels to filter messages | DBUG  INFO  WARN  EXCP  ERRO  FAIL | Debug  Info  Warn  Error  Fatal | Debug  Trace  Info  Warn  Error  Fatal |
| Log file rotation by date | √ | √ | √ |
| Log file rotation by size | √ | √ | √ |
| Log file rotation max parts | √ | √ | √ |
| **Advanced Features** | **DevLib.Logging** | **log4net** | **NLog** |
| Change log file at runtime | √ | **×** | **×** |
| Asynchronous logging | √ | **×** | √ |
| Backlog queues | √ | **×** | **×** |
| Work on configuration fault | √ | **×** | **×** |
| Advanced protocols and connections | - | Network  Memory  Database  Custom | Network  Memory  MSMQ  Database  Custom |









### Performance impact

Actually all kinds of logging system have more or less impact on application performance, but a good logging system can minimize the impact.

For example, let’s say in one application, the total logging time is 1 second if using DevLib.Logging, and DevLib.Logging is **x** times faster than other logging system, for example: NLog, so NLog need **x**\*1 seconds, then if base on the application using NLog, if logging takes **y** percent of the whole application, so the total business code time should be



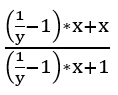
The total time should be



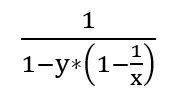
If we change to use DevLib.Logging, the total time should be



Then the performance formula should be



Simplify it to



(**y** is the logging percent of whole application; **x** is how times fast the DevLib.Logging is than other logging)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| x y | 1% | 5% | 10% | 20% | 30% | 50% |
| 2x | 1.005 | 1.0256 | 1.0526 | 1.1111 | 1.1765 | 1.3333 |
| 3x | 1.0067 | 1.0345 | 1.0714 | 1.1538 | 1.25 | 1.5 |
| 4x | 1.0076 | 1.039 | 1.0811 | 1.1765 | 1.2903 | 1.6 |
| 5x | 1.0081 | 1.0417 | 1.087 | 1.1905 | 1.3158 | 1.6667 |
| 10x | 1.009 | 1.047 | 1.0989 | 1.2196 | 1.3699 | 1.8182 |
| 20x | 1.0096 | 1.0499 | 1.105 | 1.235 | 1.3986 | 1.9048 |
| 50x | 1.0099 | 1.0515 | 1.1086 | 1.2438 | 1.4164 | 1.9608 |
| 100x | 1.01 | 1.0521 | 1.1099 | 1.2469 | 1.6556 | 1.9802 |

So if logging take 10% of the whole application and DevLib.Logging is 10x faster than NLog, the whole enhancement is 1.0989, almost 10% faster, and if the application runs 24\*7, 10% is very significant. Considering logging is a very high frequency call in an application, the impact from logging system cannot be ignored.

## DevLib.Net

## DevLib.Options

## DevLib.Reflection

## DevLib.Serialization

## DevLib.ServiceModel

DevLib.ServiceModel namespace includes WCF hosting and WCF client invoking. Windows Communication Foundation framework provides how to host a WCF service and how to call a WCF service, but there is too much configurations, too much generated codes and too complex. DevLib.ServiceModel provides a very simple, reusable, customizable and reliable mechanism for hosting and invoking WCF services.

### Service Hosting

To become active, a service must be hosted within a run-time environment that creates it and controls its context and lifetime. Windows Communication Foundation (WCF) services are designed to run in any Windows process that supports managed code.

These hosting options are

* Self-Hosting in a Managed Application
* Managed Windows Services
* Internet Information Services (IIS)
* Windows Process Activation Service (WAS)

Developers choose the hosting environment that satisfies the service's deployment requirements. These requirements might derive from the platform on which the application is deployed, the transport on which it must send and receive messages, or on the type of process recycling and other process management required to ensure adequate availability, or on some other management or reliability requirements.

DevLib.ServiceModel adapts to all kinds of hosting way. It provides a consistent approach, so developers can focus on design and implement WCF service itself, leave the host task to DevLib.ServiceModel.

To host a WCF service, you need WCF three elements: Address, Binding and Contract (ABC). The traditional way is those elements are defined in a configuration file, it is not easy to customize at run time. Even the configuration file itself, you cannot change to use another file as configuration file. DevLib.ServiceModel supports all kinds of customization at run time, even you can chose which configuration file you want to use.

#### Create hosting and Open

The table below show the options of ABC you can chose:

|  |  |  |
| --- | --- | --- |
| **Address** | **Binding** | **Contract** |
| WCF Config File | WCF Config File | WCF DLLs |
| Address String | Binding Type | Service Type |
| Port(BasicHttpBinding) | Binding Instance | Contract Type |

There are two approaches to host and open a WCF service:

1. Use override constructors:

public WcfServiceHost(string assemblyFile, int port = 80, bool openNow = false);

public WcfServiceHost(Type serviceType, int port = 80, bool openNow = false);

public WcfServiceHost(string assemblyFile, string configFile, string baseAddress = null, bool openNow = false);

public WcfServiceHost(string assemblyFile, Binding binding, string baseAddress, bool openNow = false);

public WcfServiceHost(string assemblyFile, Type bindingType, string baseAddress, bool openNow = false);

public WcfServiceHost(Type serviceType, string configFile, string baseAddress = null, bool openNow = false);

public WcfServiceHost(Type serviceType, Binding binding, string baseAddress, bool openNow = false);

public WcfServiceHost(Type serviceType, Type bindingType, string baseAddress, bool openNow = false);

public WcfServiceHost(string assemblyFile, Type contractType, Binding binding, string baseAddress, bool openNow = false);

public WcfServiceHost(string assemblyFile, Type contractType, Type bindingType, string baseAddress, bool openNow = false);

public WcfServiceHost(Type serviceType, Type contractType, Binding binding, string baseAddress, bool openNow = false);

public WcfServiceHost(Type serviceType, Type contractType, Type bindingType, string baseAddress, bool openNow = false);

**openNow** indicates whether open wcf service immediately or not.

If **openNow** is false, you need to call Open to open the service:

public void Open();

For example, I chose those options from the table:

|  |  |  |
| --- | --- | --- |
| **Address** | **Binding** | **Contract** |
| WCF Config File | WCF Config File | WCF DLLs |
| Address String | Binding Type | Service Type |
| Port(BasicHttpBinding) | Binding Instance | Contract Type |

You can call like:

WcfServiceHost wcfService = new WcfServiceHost("MyWcfService.dll", typeof(BasicHttpBinding), "http://localhost:999/MyWcfService", true);

Then you host a WCF service with BasicHttpBinding and hosting address is http://localhost:999/MyWcfService

|  |  |  |
| --- | --- | --- |
| **Address** | **Binding** | **Contract** |
| WCF Config File | WCF Config File | WCF DLLs |
| Address String | Binding Type | Service Type |
| Port(BasicHttpBinding) | Binding Instance | Contract Type |

You can call like:

WcfServiceHost wcfService = new WcfServiceHost(typeof(MyWcfService), 888);

Then you host a WCF service with BasicHttpBinding and hosting address port is 888, the address is http://localhost:888/{full assembly name of MyWcfService}

2. Use default constructor and call override Initialize():

public void Initialize(string assemblyFile, int port = 80);

public void Initialize(Type serviceType, int port = 80);

public void Initialize(string assemblyFile, string configFile, string baseAddress = null);

public void Initialize(string assemblyFile, Binding binding, string baseAddress);

public void Initialize(string assemblyFile, Type bindingType, string baseAddress);

public void Initialize(Type serviceType, string configFile, string baseAddress = null);

public void Initialize(Type serviceType, Binding binding, string baseAddress);

public void Initialize(Type serviceType, Type bindingType, string baseAddress);

public void Initialize(string assemblyFile, Type contractType, Binding binding, string baseAddress);

public void Initialize(string assemblyFile, Type contractType, Type bindingType, string baseAddress);

public void Initialize(Type serviceType, Type contractType, Binding binding, string baseAddress);

public void Initialize(Type serviceType, Type contractType, Type bindingType, string baseAddress);

For example, I chose those options from the table:

|  |  |  |
| --- | --- | --- |
| **Address** | **Binding** | **Contract** |
| WCF Config File | WCF Config File | WCF DLLs |
| Address String | Binding Type | Service Type |
| Port(BasicHttpBinding) | Binding Instance | Contract Type |

You can call like:

WcfServiceHost wcfService = new WcfServiceHost();

wcfService.Initialize("MyWcfService.dll", "MyConfig.xml", "http://localhost:777/AnotherAddress");

wcfService.Open();

Then you host a WCF service with binding defined in config file MyConfig.xml and hosting address is http://localhost:777/AnotherAddress

#### Stop, Abort, Restart, Dispose

Causes a communication object to transition from its current state into the closed state.

public void Close()

Causes a communication object to transition immediately from its current state into the closing state.

public void Abort()

You can restart current service if Close or Abort.

public void Restart()

You may dispose current instance if finished using.

public void Dispose()

#### Others

Gets a value indicating whether service host is opened or not.

public bool IsOpened { get; }

Get Wcf service state list.

public List<WcfServiceHostInfo> GetHostInfoList()

**WcfServiceHostInfo** includes ServiceType, BaseAddress and CommunicationState.

### Client Invoking

After you create a service, the next step is to create a WCF client proxy. A client application uses the WCF client proxy to communicate with the service. Client applications usually import a service's metadata to generate WCF client code that can be used to invoke the service.

The basic steps for creating a WCF client include the following:

1. Compile the service code.
2. Generate the WCF client proxy.
3. Instantiate the WCF client proxy.

The WCF client proxy can be generated manually by using the Service Model Metadata Utility Tool (SvcUtil.exe). The WCF client proxy can also be generated within Visual Studio using the Add Service Reference feature. To generate the WCF client proxy using either method the service must be running. If the service is self-hosted you must run the host. If the service is hosted in IIS/WAS you do not need to do anything else.

In either case, you need to make sure the WCF service is running at first, and you need to generate lots of codes by tools. If service side updated, you need to redo all steps and update your generated code manually.

So the traditional way is a little complex and it is easy to make mistake. DevLib.ServiceModel client invoking will rescue developers’ life, help developers leverage WCF without needing to add a service reference to services they control. The main features are:

* Proxyless
* Exception management
* Fault tolerance
* Reliable and efficient reuse
* Automatically Cleanup

#### WcfClientProxy<TChannel>

WcfClientProxy<TChannel> is a static class provides the implementation used to create client objects that can call services. There are five groups of methods, each group of methods have the same method signatures.

##### GetClientBaseInstance group

public static TChannel GetClientBaseInstance(bool fromCaching = true);

public static TChannel GetClientBaseInstance(string remoteUri, bool fromCaching = true);

public static TChannel GetClientBaseInstance(Binding binding, string remoteUri, bool fromCaching = true);

public static TChannel GetClientBaseInstance(string remoteHost, int remotePort, bool fromCaching = true);

public static TChannel GetClientBaseInstance(string endpointConfigurationName, string remoteUri, bool fromCaching = true);

public static TChannel GetClientBaseInstance(Type bindingType, string remoteUri, bool fromCaching = true);

GetClientBaseInstance group will create WCF client proxy based on System.ServiceModel.ClientBase<TChannel>, and the implementation is the same as generated within Visual Studio using the Add Service Reference feature.

public static TChannel GetClientBaseInstance(bool fromCaching = true)

This method will use the default configuration file to create client proxy, Address, Binding are defined in default configuration file.

public static TChannel GetClientBaseInstance(string remoteUri, bool fromCaching = true)

This method will use remoteUri (include ip/uri and port) as WCF service address and the Binding defined in the default configuration file to create client proxy.

public static TChannel GetClientBaseInstance(string remoteHost, int remotePort, bool fromCaching = true)

This method will use **remoteHost** and **remotePort** as WCF service address and the Binding defined in the default configuration file to create client proxy.

public static TChannel GetClientBaseInstance(string endpointConfigurationName, string remoteUri, bool fromCaching = true)

This method will use **remoteUri** as WCF service address and the Binding defined in xml node **endpointConfigurationName** of the default configuration file to create client proxy.

public static TChannel GetClientBaseInstance(Binding binding, string remoteUri, bool fromCaching = true)

This method will use **remoteUri** as WCF service address and **binding** instance to create client proxy.

public static TChannel GetClientBaseInstance(Type bindingType, string remoteUri, bool fromCaching = true)

This method will use **remoteUri** as WCF service address and **bindingType** to create client proxy.

The parameter **fromCaching** indicate whether get instance from caching or not.

##### GetPerSessionThrowableInstance group

public static TChannel GetPerSessionThrowableInstance(bool fromCaching = true);

public static TChannel GetPerSessionThrowableInstance(string remoteUri, bool fromCaching = true);

public static TChannel GetPerSessionThrowableInstance(Binding binding, string remoteUri, bool fromCaching = true);

public static TChannel GetPerSessionThrowableInstance(string remoteHost, int remotePort, bool fromCaching = true);

public static TChannel GetPerSessionThrowableInstance(string endpointConfigurationName, string remoteUri, bool fromCaching = true);

public static TChannel GetPerSessionThrowableInstance(Type bindingType, string remoteUri, bool fromCaching = true);

GetPerSessionThrowableInstance group create Wcf client instance. This instance of the proxy is reused for each session and will throw exception if error happened.

##### GetPerSessionUnthrowableInstance group

public static TChannel GetPerSessionUnthrowableInstance(bool fromCaching = true);

public static TChannel GetPerSessionUnthrowableInstance(string remoteUri, bool fromCaching = true);

public static TChannel GetPerSessionUnthrowableInstance(Binding binding, string remoteUri, bool fromCaching = true);

public static TChannel GetPerSessionUnthrowableInstance(string remoteHost, int remotePort, bool fromCaching = true);

public static TChannel GetPerSessionUnthrowableInstance(string endpointConfigurationName, string remoteUri, bool fromCaching = true);

public static TChannel GetPerSessionUnthrowableInstance(Type bindingType, string remoteUri, bool fromCaching = true);

GetPerSessionUnthrowableInstance group create Wcf client instance. This instance of the proxy is reused for each session and will not throw any exception.

##### GetPerCallThrowableInstance group

public static TChannel GetPerCallThrowableInstance(bool fromCaching = true);

public static TChannel GetPerCallThrowableInstance(string remoteUri, bool fromCaching = true);

public static TChannel GetPerCallThrowableInstance(Binding binding, string remoteUri, bool fromCaching = true);

public static TChannel GetPerCallThrowableInstance(string remoteHost, int remotePort, bool fromCaching = true);

public static TChannel GetPerCallThrowableInstance(string endpointConfigurationName, string remoteUri, bool fromCaching = true);

public static TChannel GetPerCallThrowableInstance(Type bindingType, string remoteUri, bool fromCaching = true);

A new instance is created for each call then disposed and will throw exception if error happened.

##### GetPerCallUnthrowableInstance group

public static TChannel GetPerCallUnthrowableInstance(bool fromCaching = true);

public static TChannel GetPerCallUnthrowableInstance(string remoteUri, bool fromCaching = true);

public static TChannel GetPerCallUnthrowableInstance(Binding binding, string remoteUri, bool fromCaching = true);

public static TChannel GetPerCallUnthrowableInstance(string remoteHost, int remotePort, bool fromCaching = true);

public static TChannel GetPerCallUnthrowableInstance(string endpointConfigurationName, string remoteUri, bool fromCaching = true);

public static TChannel GetPerCallUnthrowableInstance(Type bindingType, string remoteUri, bool fromCaching = true);

A new instance is created for each call then disposed and will not throw any exception.

#### WcfClientChannelFactory<TChannel>

WcfClientChannelFactory<TChannel> is a static class creates channels of different types that are used by clients to send messages to variously configured service endpoints. It is based on System.ServiceModel.ChannelFactory<TChannel>

public static TChannel CreateChannel(bool fromCaching = true)

This method will use the default configuration file to create client proxy, Address, Binding are defined in default configuration file.

public static TChannel CreateChannel(string remoteUri, bool fromCaching = true)

This method will use **remoteUri** (include ip/uri and port) as WCF service address and the Binding defined in the default configuration file to create client proxy.

public static TChannel CreateChannel(string remoteHost, int remotePort, bool fromCaching = true)

This method will use **remoteHost** and **remotePort** as WCF service address and the Binding defined in the default configuration file to create client proxy.

public static TChannel CreateChannel(string endpointConfigurationName, string remoteUri, bool fromCaching = true)

This method will use **remoteUri** as WCF service address and the Binding defined in xml node **endpointConfigurationName** of the default configuration file to create client proxy.

public static TChannel CreateChannel(Binding binding, string remoteUri, bool fromCaching = true)

This method will use **remoteUri** as WCF service address and **binding** instance to create client proxy.

public static TChannel CreateChannel(Type bindingType, string remoteUri, bool fromCaching = true)

This method will use **remoteUri** as WCF service address and **bindingType** to create client proxy.

The parameter **fromCaching** indicate whether get instance from caching or not.

### WcfServiceType

WcfServiceType is a utility static class provides methods deal with WCF type, contract, binding issues.

Get the type list of hosted services from assembly file.

public static List<Type> LoadFile(string assemblyFile)

**assemblyFile** is the name or path of the file that contains the manifest of the assembly.

return value is the list of hosted services type.

Get the type list of hosted services from assembly file.

public static List<Type> LoadFile(string assemblyFile, string configFile)

**assemblyFile** is the name or path of the file that contains the manifest of the assembly.

**configFile** is Wcf configuration file.

return value is the list of hosted services type

Gets the System.Type object with the specified name from assembly file.

public static Type LoadFrom(string assemblyFile, string typeFullName)

**assemblyFile** is the name or path of the file that contains the manifest of the assembly.

**typeFullName** is the full name of the type.

return value is A System.Type object that represents the specified class.

Get the ServiceContract list of hosted services from service type.

public static List<Type> GetServiceContract(Type type)

**type** is the service type.

return value is the list of hosted services contract.

Gets the Binding instance according to a Binding type.

public static Binding GetBinding(Type bindingType)

**bindingType** is the type of T:System.ServiceModel.Channels.Binding for the service.

return value is the Instance of Binding.

Gets the Binding instance according to a Binding type name.

public static Binding GetBinding(string bindingTypeName)

**bindingTypeName** is the name of T:System.ServiceModel.Channels.Binding for the service.

return value is the Instance of Binding.

Check whether a Type is a Wcf service.

public static bool IsWcfServiceClass(Type type)

**type** is the Source Type to check.

return true if type is WcfService Class; otherwise, false.

Check whether a Type has ServiceContract Attribute.

public static bool HasServiceContractAttribute(Type type)

**type** is the Source Type to check.

return true if has ServiceContractAttribute; otherwise, false.

### Sample

Create a WCF service, provides a Calculator feature, for example, Add; and use a client to call this WCF service.

**1. Create Calculator contract**

Open Visual Studio, create a new WCF service library project, let’s say “Sample.CalcContract”.

Delete “Service1.cs” and “App.config”, rename “IService1.cs” to “ICalcContract.cs”.

Modify “ICalcContract.cs” as below:

[ServiceContract]

public interface ICalcContract

{

[OperationContract]

int Add(int a, int b);

}

**2. Implement Calculator contract**

Add a new WCF service library project, “Sample.CalcImplement”.

Delete “IService1.cs” and “App.config”, rename “Service1.cs” to “CalcService.cs”.

Add project “Sample.CalcContract” as reference.

Modify “CalcService.cs” as below:

using Sample.CalcContract;

namespace Sample.CalcImplement

{

public class CalcService : ICalcContract

{

public int Add(int a, int b)

{

return a + b;

}

}

}

**3. Create Host**

Add a new Console Application project called “Sample.CalcHost”.

Add reference, choice “Sample.CalcImplement” and “Sample.CalcContract” from solution tab, choice “System.ServiceModel” from Assemblies tab and choice “DevLib.ServiceModel.dll” from Browse tab click “Browse…” and choice “DevLib.ServiceModel.dll” file.

Modify “Program.cs” as below:

using DevLib.ServiceModel;

using Sample.CalcImplement;

namespace Sample.CalcHost

{

class Program

{

static void Main(string[] args)

{

WcfServiceHost host = new WcfServiceHost(typeof(Sample.CalcImplement.CalcService), 888, true);

Console.ReadLine();

}

}

}

This console application will host CalcService at the address: http://localhost:888/Sample.CalcContract.ICalcContract

**4. Create Client**

Add a new Console Application project called “Sample.CalcClient”.

Add reference, choice “Sample.CalcContract” from solution tab, choice “System.ServiceModel” from Assemblies tab and choice “DevLib.ServiceModel.dll” from Browse tab click “Browse…” and choice “DevLib.ServiceModel.dll” file.

Modify “Program.cs” as below:

using DevLib.ServiceModel;

using Sample.CalcContract;

namespace Sample.CalcClient

{

class Program

{

static void Main(string[] args)

{

var client = WcfClientProxy<ICalcContract>.GetPerCallThrowableInstance("localhost", 888);

int result = client.Add(1, 2);

Console.WriteLine(result);

Console.ReadLine();

}

}

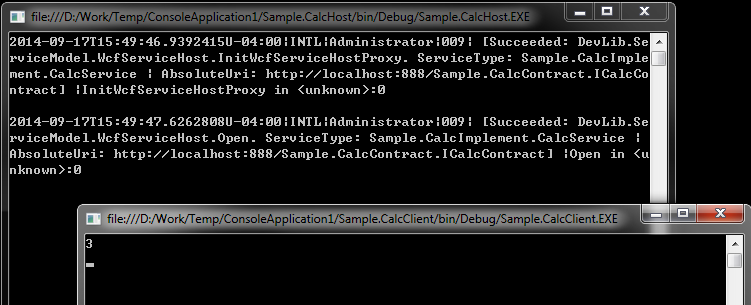
}

**5. Run and verify**

Build all Solution.

Run “Sample.CalcHost.exe” and leave the console open.

Run “Sample.CalcClient.exe”, you will get result “3”.



**6. Requirement changed, need a new method**

Now we have a new requirement, a new function to do subtraction.

Open “ICalcContract.cs”, add a new method Sub:

[OperationContract]

int Sub(int a, int b);

Open “CalcService.cs”, add a new method implementation:

public int Sub(int a, int b)

{

return a - b;

}

Open Sample.CalcClient “Program.cs”, add new code to call Sub:

var client = WcfClientProxy<ICalcContract>.GetPerCallThrowableInstance("localhost", 888);

int result = client.Add(1, 2);

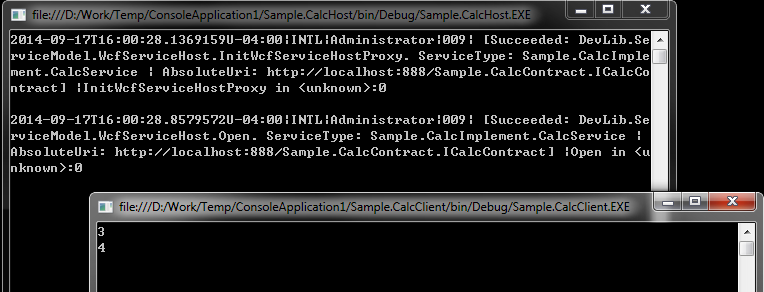
Console.WriteLine(result);

int result1 = client.Sub(9, 5);

Console.WriteLine(result1);

Console.ReadLine();

Now do the step 5, run and verify again.



It is very easy to modify WCF services without changing any code of hosting and client proxy.

## DevLib.ServiceProcess

Microsoft Windows services, formerly known as NT services, enable you to create long-running executable applications that run in their own Windows sessions. These services can be automatically started when the computer boots, can be paused and restarted, and do not show any user interface. These features make services ideal for use on a server or whenever you need long-running functionality that does not interfere with other users who are working on the same computer. You can also run services in the security context of a specific user account that is different from the logged-on user or the default computer account.

In view of those special features, it is not easy to debug Windows services and not easy to develop and deploy them. DevLib.ServiceProcess fill in those gaps, let developers easy to implement, debug and deploy Windows services.

### Walkthrough

#### Create project

Open Visual Studio, create a new Console Application project, let’s say “Sample.MyWinService”.

#### Adding service entry point file

Add a new file “ServiceManager.cs” (can be any name you want) to project “Sample.MyWinService”.

Add reference, choice “System.ServiceProcess” and “System.Configuration.Install” from Assemblies tab and choice “DevLib.ServiceProcess.dll” from Browse tab click “Browse…” and choice “DevLib.ServiceProcess.dll” file.

Modify “ServiceManager.cs” as below:

using System.ComponentModel;

using System.ServiceProcess;

using DevLib.ServiceProcess;

namespace Sample.MyWinService

{

[RunInstaller(true)]

public class ServiceManager : WindowsServiceInstaller, IWindowsService

{

public WindowsServiceSetup ServiceSetupInfo

{

get;

set;

}

public ServiceManager()

{

this.ServiceSetupInfo = new WindowsServiceSetup("Sample.MyWinService");

this.InstallerSetupInfo = this.ServiceSetupInfo;

}

public void OnContinue()

{

// Continue service here

Console.WriteLine("Sample.MyWinService resume.");

}

public void OnCustomCommand(int command)

{

// Custom command

}

public void OnPause()

{

// Pause service here

Console.WriteLine("Sample.MyWinService pause.");

}

public bool OnPowerEvent(PowerBroadcastStatus powerStatus)

{

// When power event

return true;

}

public void OnSessionChange(SessionChangeDescription changeDescription)

{

// When session change

}

public void OnShutdown()

{

// When system shut down

}

public void OnStart(string[] args)

{

// Start service here

Console.WriteLine("Sample.MyWinService start.");

}

public void OnStop()

{

// Stop service here

Console.WriteLine("Sample.MyWinService stop.");

}

}

}

#### Modify “Program.cs”

Modify “Program.cs” as below:

using DevLib.ServiceProcess;

namespace Sample.MyWinService

{

class Program

{

static void Main(string[] args)

{

ServiceManager manager = new ServiceManager();

WindowsServiceBase.Run(manager);

}

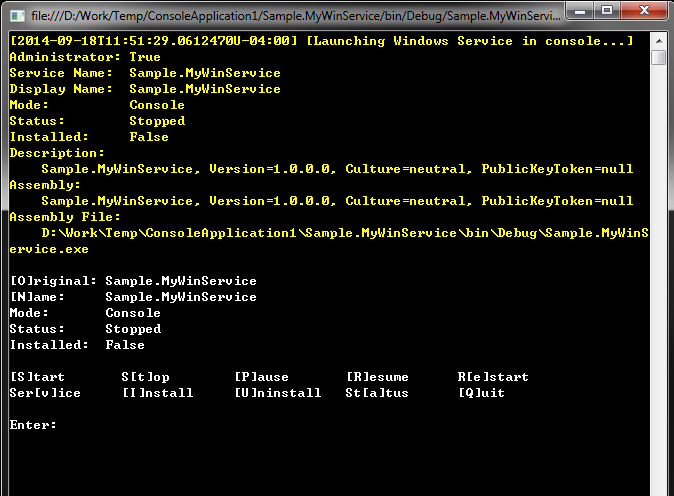
}

}

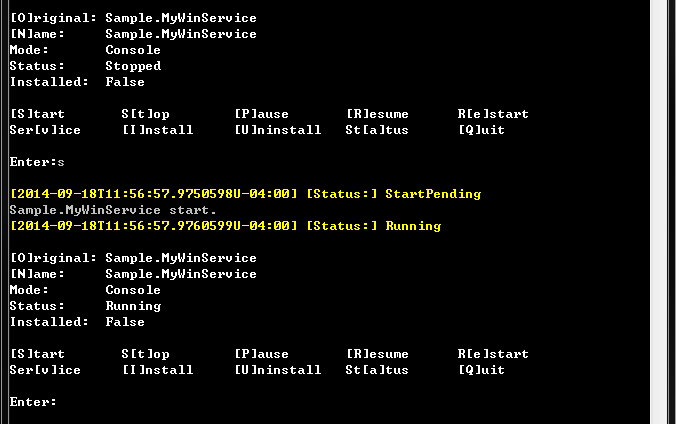
Note: the most critical step is your own public class should inherit from **WindowsServiceInstaller** and **IWindowsService** and mark the class use attribute **[RunInstaller(true)]**, and in default constructor create new instance of **WindowsServiceSetup** and set it to this.ServiceSetupInfo and this.InstallerSetupInfo

#### Build and run

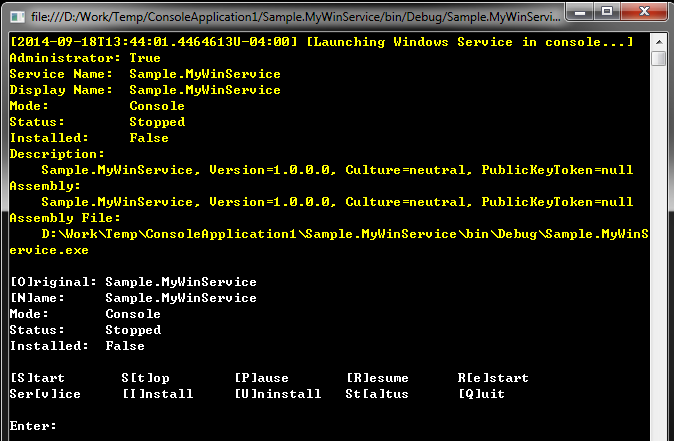
Build your project and since the project is a console application, you can run it directly.



Now the service is launched but not start yet, you need to input “S” or “s” then enter to start this Windows service in console mode.



### Usage



This console allow you to control your Windows service.

The strings in yellow is the properties of current Windows service:

**Administrator**: whether current Windows service has an Administrator right or not. Administrators have complete and unrestricted access to the computer or domain.

**Service Name**: current Windows service name.

**Display Name**: current Windows service display name.

**Mode**: indicate whether current Windows service is running in console mode or in a service mode.

**Status**: current Windows service running status.

**Installed**: indicate whether current Windows service install and register as a Windows service or not.

**Description**: description about current Windows service, by default is the dll assembly information.

**Assembly**: current Windows service assembly information.

**Assembly File**: current Windows service file.

The character inside “[ ]” is the command you can input. It is case ignored.

**S**: start current Windows service.

**T**: stop current Windows service.

**P**: pause current Windows service.

**R**: resume current Windows service.

**E**: restart current Windows service.

**V**: change console mode to service mode, when it is running on service mode, the word will change to “[C]onsole”.

**C**: when it is running on service mode, you can see [C]onsole, and input “C” will change service mode to console mode.

**I**: install and register as Windows service. If use I {new mane}, for example, *i MyService1* it will install the Windows service by using service name “MyService1”

**U**: uninstall and unregister current Windows service. If use U {new mane}, for example, *u MyService1* it will uninstall the Windows service by using service name “MyService1”

**A**: display current status.

**Q**: stop current Windows service and exit this console.

**N**: change the service name of current Windows service, usage: n {new name}, for example: *n MyNewService* the service name will change to “MyNewService”.

**O**: restore the service name to original name.

Note: anytime press Enter, this command prompt will display.

Note: if current Windows service is in service mode, the commands Start, Stop, Pause, Resume and Restart will only control the registered Windows service by using the server name displayed in [N]ame, if current Windows service is in console mode, those commands only control current console process.

### WindowsServiceSetup

This class is used to set Windows service properties.

Gets or sets windows service assembly file path. Default value is current entry assembly.

public string ServiceAssemblyPath { get; set; }

Gets windows service assembly.

public Assembly ServiceAssembly { get; }

Gets or sets the name used by the system to identify this service. This property must be identical to the System.ServiceProcess.ServiceBase.ServiceName of the service you want to install. Default value is current entry assembly name.

public string ServiceName { get; set; }

Gets or sets the friendly name that identifies the service to the user. Default value is current entry assembly name.

public string DisplayName { get; set; }

Gets or sets the description for the service. Default value is "{current entry assembly full name} Installed on {yyyy-MM-ddTHH:mm:ss.fffffffUzzz}"

public string Description { get; set; }

Gets or sets the services that must be running for this service to run. Default value is null.

public string[] ServicesDependedOn { get; set; }

Gets or sets how and when this service is started. Default value is ServiceStartMode.Automatic

public ServiceStartMode StartType { get; set; }

Gets or sets a value indicating whether to start service immediately after installed. Default value is true

public bool StartAfterInstall { get; set; }

Gets or sets a value indicating whether to restart service after failure. Default value is true

public bool RestartOnFailure { get; set; }

Gets or sets a value indicating whether to report Start, Stop, Pause, and Continue commands in the event log. Default value is true

public bool AutoLog { get; set; }

Gets or sets a value indicating whether the service can handle notifications of computer power status changes. Default value is true

public bool CanHandlePowerEvent { get; set; }

Gets or sets a value indicating whether the service can handle session change events received from a Terminal Server session. Default value is true

public bool CanHandleSessionChangeEvent { get; set; }

Gets or sets a value indicating whether the service can be paused and resumed. Default value is true

public bool CanPauseAndContinue { get; set; }

Gets or sets a value indicating whether the service should be notified when the system is shutting down. Default value is true

public bool CanShutdown { get; set; }

Gets or sets a value indicating whether the service can be stopped once it has started. Default value is true

public bool CanStop { get; set; }

Gets or sets the type of account under which to run this service application. Default value is ServiceAccount.LocalSystem

public ServiceAccount Account { get; set; }

Gets or sets the user account under which the service application will run. Default value is null

public string Username { get; set; }

Gets or sets the password associated with the user account under which the service application runs. Default value is null

public string Password { get; set; }

Creates a new object that is a deep copy of the current instance.

public object Clone()

### WindowsServiceBase

WindowsServiceBase is a static class provides features to console Windows service.

Registers the executable for a service with the Service Control Manager (SCM).

public static void Run(IWindowsService windowsService, string[] args = null)

**windowsService** is A IWindowsService which indicates a service to start.

**args** is command line arguments.

Starts a service, passing the specified arguments.

public static bool Start(string serviceName, string[] args = null, bool throwOnError = false)

**serviceName** is the service name.

**args** is an array of arguments to pass to the service when it starts.

**throwOnError** true to throw any exception that occurs.-or- false to ignore any exception that occurs.

return true if succeeded; otherwise, false.

Stops this service and any services that are dependent on this service.

public static bool Stop(string serviceName, bool throwOnError = false)

**serviceName** is the service name.

**throwOnError** true to throw any exception that occurs.-or- false to ignore any exception that occurs.

return true if succeeded; otherwise, false.

Suspends a service's operation.

public static bool Pause(string serviceName, bool throwOnError = false)

**serviceName** is the service name.

**throwOnError** true to throw any exception that occurs.-or- false to ignore any exception that occurs.

return true if succeeded; otherwise, false.

Continues a service after it has been paused.

public static bool Continue(string serviceName, bool throwOnError = false)

**serviceName** is the service name.

**throwOnError** true to throw any exception that occurs.-or- false to ignore any exception that occurs.

return true if succeeded; otherwise, false.

Executes a custom command on the service.

public static bool ExecuteCommand(string serviceName, int command, bool throwOnError = false)

**serviceName** is the service name.

**command** is an application-defined command flag that indicates which custom command to execute.

**throwOnError** true to throw any exception that occurs.-or- false to ignore any exception that occurs.

return true if succeeded; otherwise, false.

Gets the status of the service.

public static ServiceControllerStatus GetServiceStatus(string serviceName, bool throwOnError = false)

**serviceName** is the service name.

**throwOnError** true to throw any exception that occurs.-or- false to ignore any exception that occurs.

return one of the System.ServiceProcess.ServiceControllerStatus values that indicates whether the service is running, stopped, or paused, or whether a start, stop, pause, or continue command is pending.

Determines whether the specified service exists.

public static bool ServiceExists(string serviceName, bool throwOnError = false)

**serviceName** is the service name.

**throwOnError** true to throw any exception that occurs.-or- false to ignore any exception that occurs.

return true if the specified service exists; otherwise, false.

### WindowsServiceInstaller

WindowsServiceInstaller class has two static method to help you install or uninstall Windows service at runtime.

Install a windows service at runtime.

public static void RuntimeInstall(WindowsServiceSetup windowsServiceSetup)

**windowsServiceSetup** is an instance of WindowsServiceSetup.

Uninstall a windows service at runtime.

public static void RuntimeUninstall(WindowsServiceSetup windowsServiceSetup)

**windowsServiceSetup** is an instance of WindowsServiceSetup.

## DevLib.TerminalServices

## DevLib.Timers

## DevLib.Utilities

## DevLib.WinForms

## DevLib.Xml

### XmlConverter

XmlConverter is a static class provides methods for converting between common language runtime types and XML Schema definition language (XSD) types. When converting data types the values returned are locale independent.

#### Object to String

public static string ToString(object value)

**value** is the object to convert.

return a string representation of the object.

#### String to Object

public static object ToObject(string value, Type targetType)

**value** is the string to convert.

**targetType** is the target type to convert.

return an object equivalent of the string.

public static T ToObject<T>(string value)

**T** is the target type to convert.

**value** is the string to convert.

return an object equivalent of the string.

#### CanConvert

Whether this converter can convert the object to a String and vice versa.

public static bool CanConvert(Type sourceType)

**sourceType** represents the type you want to convert.

return true if this converter can perform the conversion; otherwise, false.

# Known Issues

## DevLib.DaemonProcess

### Get command line arguments

The feature get command line arguments implementation idea is firstly try to use WMI to query command line arguments by process id. If failed, try to use Win32 API to get command line arguments.

In some cases, current local machine disabled WMI query or did not support WMI query, so the implementation will call Win32 API instead, the know issue is here, currently DevLib.DaemonProcess only support using Win32 API get command line arguments on Windows 32bit (x86) platform, I don’t know how to use native API to get command line arguments on Windows 64bit (x64) platform. So if the current local machine is x64 Windows and disabled WMI query, DevLib.DaemonProcess may not get the correct command line arguments, and may not work properly.

If you know how to use Win32 API (native API) to get command line arguments on x64 Windows, please tell me the solution.

Appendix

# References