Jiří Činčura

How I put .NET into Firebird database engine

What I mean by .NET into Firebird?

Prerequisites

Not my full-time job

Work in progress

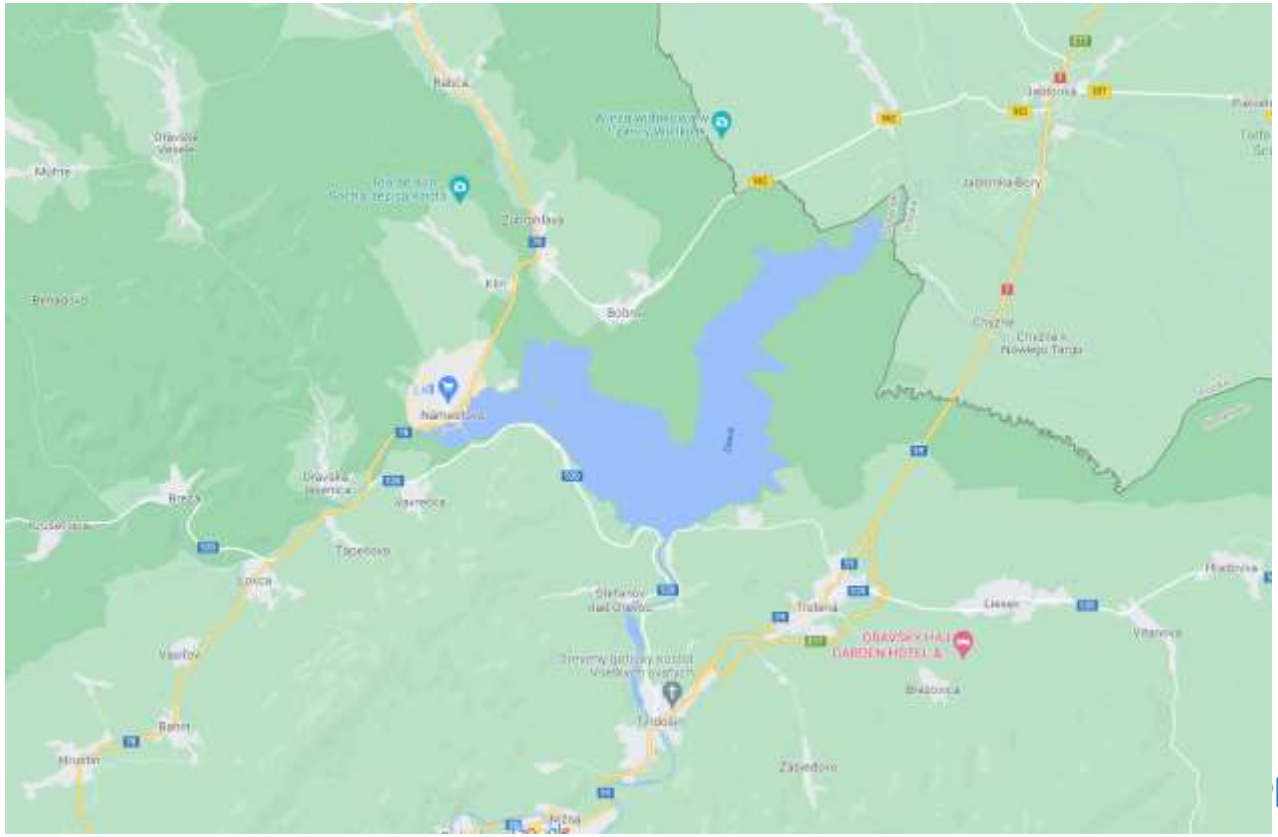
Performance has its (support) cost

Start

Firebird 3 introduced concept of “external engine” plugins

2016 working for a customer ↓ and bored during evenings

“JiriPlugin”



Start

.NET 4.5.2

A lot of reflection “to make it work”

Using Tuple<T, …>

Assembly.LoadFrom

Start


```
public class Argument
{
        readonly IntPtr _valuePtr;
        readonly IntPtr _nullPtr;
        readonly int _type;
        readonly int _length;

        public Argument(IntPtr valuePtr, IntPtr nullPtr, int type, int length)
        {
                _valuePtr = valuePtr;
                _nullPtr = nullPtr;
                _type = type;
                _length = length;
        }
```

ValueTuple<T, …>

C# 7 syntax sugar

Size, indexer access

Reflection

No ITuple yet ☹ (4.7.1)

Hot-reload

Idea to hot-swap assemblies while server is running ☺

AppDomains, MarshalByRefObject, …

```
public class AppDomainExecutor : MarshalByRefObject, IDisposable
var appDomainSetup = new AppDomainSetup()
{
        ApplicationBase = LocationHelper.BaseDirectory,
        LoaderOptimization = LoaderOptimization.MultiDomainHost,
};
```

Marshaling between AppDomains is very slow

Hack: Not loading from filename, but from byte array

- Dependencies

No unloading

Lucky it didn’t work out, because .NET Core dropped AppDomains

DelegateBuilder

Calling methods dynamically is slow

Need to build delegates

- Expressions or Reflection.Emit

Expressions more straightforward

- But also somewhat limited

Caching Type for comparison

Some Type == typeof(…) comparisons in plugin Caching

```
protected static Type TypeInt = typeof(int?);
protected static Type TypeString = typeof(string);
protected static Type TypeShort = typeof(short?);
protected static Type TypeLong = typeof(long?);
protected static Type TypeDateTime = typeof(DateTime?);
protected static Type TypeTimeSpan = typeof(TimeSpan?);
protected static Type TypeBool = typeof(bool?);
protected static Type TypeFloat = typeof(float?);
protected static Type TypeDouble = typeof(double?);
protected static Type TypeDecimal = typeof(decimal?);
```

Enum for Type

Caching was not enough

```
public enum ArgumentType : byte
{
          Int = 1,
          String = 2,
          Short = 3,
          Long = 4,
          DateTime = 5,
          TimeSpan = 6,
          Bool = 7,
          Float = 8,
          Double = 9,
          Decimal = 10,
          Binary = 11,
}
```

Comparing Type instances sucks

Dictionary with delegates

Originally used two ConcurrentDictionary<K, V>

My usage is mostly reads, bit writes

Eventually switched to RWLockSlim and Dictionary<K, V>

- Pre .NET Core 3.1

.NET Core migration

2019/2020

C++/CLI

AssemblyLoadContext

IJW

runtimeconfig

Local .NET Core loading

Distribute .NET Core together with plugin

Shim

DOTNET_ROOT

DOTNET_ROOT(x86)

DOTNET_MULTILEVEL_LOOKUP

Specific code

```
public class MessageMeta
{
        public uint Count { get; }
        public uint[] ValueOffsets { get; }
        public uint[] NullOffsets { get; }
        public uint[] Types { get; }
        public uint[] SubTypes { get; }
        public uint[] Lengths { get; }
        public int[] Scales { get; }
        public uint[] Charsets { get; }
```

Specific code

```
+         Marshal.WriteInt16(GetNullLocation(index), value ? (short)-1 : (short)0);
-         Marshal.WriteInt16(GetNullLocation(index), Convert.ToInt16(value));

+         T? GetValueVal<T>(int index, Func<int, T> getter) where T : struct
+         T GetValueRef<T>(int index, Func<int, T> getter) where T : class
-         object GetValue<T>(int index, Func<int, T> getter)

+         return Marshal.ReadInt16(GetNullLocation(index)) != 0;
-         return Convert.ToBoolean(Marshal.ReadInt16(GetNullLocation(index)));
```

Specific code

```
static (Expression, ParameterExpression, ParameterExpression) CreateMethodCall(MethodInfo method)
{
            var methodParameters = method.GetParameters();
            var argumentsParameter = Expression.Parameter(typeof(Arguments));
            var statementExecutorParameter = Expression.Parameter(typeof(IStatementExecutor));

            var arguments = new Expression[methodParameters.Length];
            var length = arguments.Length;
            if (methodParameters.Length > 0 && methodParameters[^1].ParameterType == typeof(IExecutionContext))
            {
                        length -= 1;
                        var t = typeof(Integration.ExecutionContext);
                        arguments[^1] = Expression.New(t.GetConstructor(new[] { typeof(IStatementExecutor) }),
statementExecutorParameter);
            }
            var getValueMethod = typeof(Arguments).GetMethod(nameof(Arguments.GetValue));
            for (var i = 0; i < length; i++)
            {
                        var a = Expression.Call(argumentsParameter, getValueMethod, Expression.Constant(i),
Expression.Constant(methodParameters[i].ParameterType.ToArgumentType()));
                        arguments[i] = Expression.Convert(a, methodParameters[i].ParameterType);
            }
```

Unsafe and void*

Hope for faster and more tight code compared to Marshal and IntPtr

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
static T Read<T>(void* source, int offset)
{
	return Unsafe.ReadUnaligned<T>(Unsafe.Add<byte>(source, offset));
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
static void Write<T>(void* source, int offset, T value)
{
	Unsafe.WriteUnaligned<T>(Unsafe.Add<byte>(source, offset), value);
}
```

Unsafe and void*

Strings, etc. are composed from smaller pieces, enforce struct

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
static T Read<T>(void* source, int offset) where T : struct
{
	return Unsafe.ReadUnaligned<T>(Unsafe.Add<byte>(source, offset));
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
static void Write<T>(void* source, int offset, T value) where T : struct
{
	Unsafe.WriteUnaligned(Unsafe.Add<byte>(source, offset), value);
}
```

Less indexing, RO structs, ref, in


```
public sealed class MessageMeta
{
        const int SQL_BLOB = 520;

        public readonly struct Value
        {
                public readonly int ValueOffset;
                public readonly int NullOffset;
                public readonly int Type;
                public readonly int SubType;
                public readonly int Length;
                public readonly int Scale;
                public bool IsBlob => Type == SQL_BLOB;

                public Value(int valueOffset, int nullOffset, int type, int subType, int length, int scale)
                {
                        ValueOffset = valueOffset;
                        NullOffset = nullOffset;
                        Type = type;
                        SubType = subType;
                        Length = length;
                        Scale = scale;
                }
        }

        readonly Value[] _items;
        public ref Value this[int index] => ref _items[index];
        public int Count => _items.Length;
```

Less indexing, RO structs, ref, in

```
bool ReadIsNull(in MessageMeta.Value meta)
```

C++/CLI pieces without managed


```
#pragma managed(push, off)
void Function::getCharSet(ThrowStatusWrapper* status, IExternalContext* context, char* name, unsigned
nameSize)
{
        strncpy(name, "UTF8", nameSize);
}
#pragma managed(pop)
```

Split “generation” from “invocation”

Dictionary lookups are expensive ☺

Split “generation” from “invocation”

- Access reference

Reuse generated pieces across invocations

```
public abstract class InvokerBase<TResult>
{
        readonly Func<Arguments, Arguments, IStatementExecutor, TResult> _invoker;
        readonly MessageMeta _inputsMeta;
        readonly MessageMeta _outputsMeta;
```

Less Arguments instances


```
public sealed unsafe class Arguments
{
        public Arguments(MessageMeta messageMeta)
        {
                _messageMeta = messageMeta;
        }

        public void Use(void* msg, IBlobHandler blobHandler)
        {
                _msg = msg;
                _blobHandler = blobHandler;
        }
```

Switch to Reflection.Emit

Control IL generation exactly

Expressions are great, but I’m hunting 1% of performance

Switch to Reflection.Emit


```
public static Func<Arguments, IStatementExecutor, object> BuildFunctionMethodDelegate(MethodInfo method)
{
                var (dynamicMethod, il) = Start<IEnumerator>("JiriFunction");
                var (methodParameters, hasExecutionContext) = PrepareMethod(method);
                EmitArgumentsValues(methodParameters.Length - (hasExecutionContext ? 1 : 0), methodParameters, il);
                if (hasExecutionContext)
                {
                                EmitExecutionContext(il);
                }
                EmitCall(method, il);
                if (method.ReturnType.IsValueType)
                {
                                il.Emit(OpCodes.Box, method.ReturnType); // type (i.e. int?) -> object
                }
                return Finish<object>(dynamicMethod, il);
}

static readonly MethodInfo GetValueMethod = typeof(Arguments).GetMethod(nameof(Arguments.GetValue));
static void EmitArgumentsValues(int count, ParameterInfo[] methodParameters, ILGenerator il)
{
                for (var i = 0; i < count; i++)
                {
                                var parameterType = methodParameters[i].ParameterType;
                                il.Emit(OpCodes.Ldarg_0); // Arguments
                                il.Emit(OpCodes.Ldc_I4, i); // index
                                il.Emit(OpCodes.Ldc_I4_S, (byte)parameterType.ToArgumentType()); // ArgumentType
                                il.Emit(OpCodes.Callvirt, GetValueMethod);
                                if (parameterType.IsValueType)
                                {
                                                il.Emit(OpCodes.Unbox_Any, parameterType); // object -> type (i.e. int?)
                                }
                }
}
```

Hot-reload revisited

Unloading difficult

Dependencies

Difficult to manage from developer POV

- Are restart really that problematic?

Static values

- Caches

https://www.tabsoverspaces.com/233870-csharp-static-constructor-called-multiple-times

https://www.tabsoverspaces.com/233871-going-deeper-into-static-constructors-hole

What’s next?

More performance (.NET 6) ☺

Unsafe casting

Fully generic pipeline from Reflection.Emit pieces

Manual .NET hosting/loading (and marshaling calls/data)

Building functions/procedures from source in DB

Please rate this session using

.NET DeveloperDays mobile app

(available on Google Play and AppStore)

Event Sponsors

Strategic Sponsors

Demant

avanade

Gold Sponsors

Relativity

ProDataConsult

medius

KMD
An NEC Company