1.1 ./Platform.Converters/CachingConverterDecorator.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Converters
{
    public class CachingConverterDecorator<TSource, TTarget> : IConverter<TSource, TTarget>
    {
        private readonly IConverter<TSource, TTarget> _baseConverter;
        private readonly IDictionary<TSource, TTarget> _cache;

        public CachingConverterDecorator(IConverter<TSource, TTarget> baseConverter,
        ↪  IDictionary<TSource, TTarget> cache) => (_baseConverter, _cache) = (baseConverter,
        ↪  cache);

        public CachingConverterDecorator(IConverter<TSource, TTarget> baseConverter) :
        ↪  this(baseConverter, new Dictionary<TSource, TTarget>()) { }

        public TTarget Convert(TSource source)
        {
            if (!_cache.TryGetValue(source, out TTarget value))
            {
                value = _baseConverter.Convert(source);
                _cache.Add(source, value);
            }
            return value;
        }
    }
}
```

1.2 ./Platform.Converters/CheckedConverter.cs

```
using System;
using System.Linq;
using System.Reflection;
using System.Reflection.Emit;
using Platform.Reflection;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Converters
{
    public abstract class CheckedConverter<TSource, TTarget> : IConverter<TSource, TTarget>
    {
        public static CheckedConverter<TSource, TTarget> Default { get; }

        static CheckedConverter()
        {
            AssemblyName assemblyName = new AssemblyName(GetNewName());
            var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
            ↪  AssemblyBuilderAccess.Run);
            var module = assembly.DefineDynamicModule(GetNewName());
            var type = module.DefineType(GetNewName(), TypeAttributes.Public |
            ↪  TypeAttributes.Class | TypeAttributes.Sealed, typeof(CheckedConverter<TSource,
            ↪  TTarget>));
            EmitMethod<Converter<TSource, TTarget>>(type, "Convert", (il) =>
            {
                il.LoadArgument(1);
                if (typeof(TSource) != typeof(TTarget))
                {
                    CheckedConvert(il);
                }
                il.Return();
            });
            var typeInfo = type.CreateTypeInfo();
            Default = (CheckedConverter<TSource, TTarget>)Activator.CreateInstance(typeInfo);
        }

        private static void CheckedConvert(ILGenerator generator)
        {
            var type = typeof(TTarget);
            if (type == typeof(short))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I2);
                }
                else
```

```
            {
                generator.Emit(OpCodes.Conv_Ovf_I2_Un);
            }
        }
        else if (type == typeof(ushort))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_Ovf_U2);
            }
            else
            {
                generator.Emit(OpCodes.Conv_Ovf_U2_Un);
            }
        }
        else if (type == typeof(sbyte))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_Ovf_I1);
            }
            else
            {
                generator.Emit(OpCodes.Conv_Ovf_I1_Un);
            }
        }
        else if (type == typeof(byte))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_Ovf_U1);
            }
            else
            {
                generator.Emit(OpCodes.Conv_Ovf_U1_Un);
            }
        }
        else if (type == typeof(int))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_Ovf_I4);
            }
            else
            {
                generator.Emit(OpCodes.Conv_Ovf_I4_Un);
            }
        }
        else if (type == typeof(uint))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_Ovf_U4);
            }
            else
            {
                generator.Emit(OpCodes.Conv_Ovf_U4_Un);
            }
        }
        else if (type == typeof(long))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_Ovf_I8);
            }
            else
            {
                generator.Emit(OpCodes.Conv_Ovf_I8_Un);
            }
        }
        else if (type == typeof(ulong))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_Ovf_U8);
            }
            else
            {
```

```
                    generator.Emit(OpCodes.Conv_Ovf_U8_Un);
                }
            }
            else if (type == typeof(float))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_R4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_R_Un);
                }
            }
            else if (type == typeof(double))
            {
                generator.Emit(OpCodes.Conv_R8);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        private static void EmitMethod<TDelegate>(TypeBuilder type, string methodName,
        ↪  Action<ILGenerator> emitCode)
        {
            var delegateType = typeof(TDelegate);
            var invoke = delegateType.GetMethod("Invoke");
            var returnType = invoke.ReturnType;
            var parameterTypes = invoke.GetParameters().Select(s => s.ParameterType).ToArray();
            MethodBuilder method = type.DefineMethod(methodName, MethodAttributes.Public |
            ↪  MethodAttributes.Virtual | MethodAttributes.Final | MethodAttributes.HideBySig,
            ↪  returnType, parameterTypes);
            method.SetImplementationFlags(MethodImplAttributes.IL | MethodImplAttributes.Managed
            ↪  | MethodImplAttributes.AggressiveInlining);
            var generator = method.GetILGenerator();
            emitCode(generator);
        }

        private static string GetNewName() => Guid.NewGuid().ToString("N");

        public abstract TTarget Convert(TSource source);
    }
}
```

1.3 ./Platform.Converters/IConverter[TSource, TTarget].cs

```
namespace Platform.Converters
{
    /// <summary>
    /// <para>Defines a converter between two types (TSource and TTarget).</para>
    /// <para>Определяет конвертер между двумя типами (исходным TSource и целевым
    ↪  TTarget).</para>
    /// </summary>
    /// <typeparam name="TSource"><para>Source type of conversion.</para><para>Исходный тип
    ↪  конверсии.</para></typeparam>
    /// <typeparam name="TTarget"><para>Target type of conversion.</para><para>Целевой тип
    ↪  конверсии.</para></typeparam>
    public interface IConverter<in TSource, out TTarget>
    {
        /// <summary>
        /// <para>Converts the value of the source type (TSource) to the value of the target
        ↪  type.</para>
        /// <para>Конвертирует значение исходного типа (TSource) в значение целевого типа.</para>
        /// </summary>
        /// <param name="source"><para>The source type value (TSource).</para><para>Значение
        ↪  исходного типа (TSource).</para></param>
        /// <returns><para>The value is converted to the target type
        ↪  (TTarget).</para><para>Значение ковертированное в целевой тип
        ↪  (TTarget).</para></returns>
        TTarget Convert(TSource source);
    }
}
```

1.4 ./Platform.Converters/IConverter[T].cs

```
namespace Platform.Converters
{
```

```
    /// <summary>
    /// <para>Defines a converter between two values of the same type.</para>
    /// <para>Определяет конвертер между двумя значениями одного типа.</para>
    /// </summary>
    /// <typeparam name="T"><para>Type of value to convert.</para><para>Тип преобразуемого
    ↪ значения.</para></typeparam>
    public interface IConverter<T> : IConverter<T, T>
    {
    }
}
```

1.5 ./Platform.Converters/To.cs

```
using System;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Converters
{
    [Obsolete]
    public static class To
    {
        public static readonly char UnknownCharacter = '';

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(ulong value) => value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long Int64(ulong value) => unchecked(value > long.MaxValue ? long.MaxValue
        ↪ : (long)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static uint UInt32(ulong value) => unchecked(value > uint.MaxValue ?
        ↪ uint.MaxValue : (uint)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int Int32(ulong value) => unchecked(value > int.MaxValue ? int.MaxValue :
        ↪ (int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ushort UInt16(ulong value) => unchecked(value > ushort.MaxValue ?
        ↪ ushort.MaxValue : (ushort)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static short Int16(ulong value) => unchecked(value > (ulong)short.MaxValue ?
        ↪ short.MaxValue : (short)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static byte Byte(ulong value) => unchecked(value > byte.MaxValue ? byte.MaxValue
        ↪ : (byte)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static sbyte SByte(ulong value) => unchecked(value > (ulong)sbyte.MaxValue ?
        ↪ sbyte.MaxValue : (sbyte)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Boolean(ulong value) => value > 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static char Char(ulong value) => unchecked(value > char.MaxValue ?
        ↪ UnknownCharacter : (char)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static DateTime DateTime(ulong value) => unchecked(value > long.MaxValue ?
        ↪ System.DateTime.MaxValue : new DateTime((long)value));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TimeSpan TimeSpan(ulong value) => unchecked(value > long.MaxValue ?
        ↪ System.TimeSpan.MaxValue : new TimeSpan((long)value));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(long value) => unchecked(value < (long)ulong.MinValue ?
        ↪ ulong.MinValue : (ulong)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(int value) => unchecked(value < (int)ulong.MinValue ?
        ↪ ulong.MinValue : (ulong)value);

```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(short value) => unchecked(value < (short)ulong.MinValue ?
        ↪  ulong.MinValue : (ulong)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(sbyte value) => unchecked(value < (sbyte)ulong.MinValue ?
        ↪  ulong.MinValue : (ulong)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(bool value) => value ? 1UL : 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(char value) => value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long Signed(ulong value) => unchecked((long)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int Signed(uint value) => unchecked((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static short Signed(ushort value) => unchecked((short)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static sbyte Signed(byte value) => unchecked((sbyte)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static object Signed<T>(T value) => To<T>.Signed(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong Unsigned(long value) => unchecked((ulong)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static uint Unsigned(int value) => unchecked((uint)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ushort Unsigned(short value) => unchecked((ushort)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static byte Unsigned(sbyte value) => unchecked((byte)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static object Unsigned<T>(T value) => To<T>.Unsigned(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T UnsignedAs<T>(object value) => To<T>.UnsignedAs(value);
    }
}
```

1.6 ./Platform.Converters/To[T].cs

```
using System;
using Platform.Exceptions;
using Platform.Reflection;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Converters
{
    [Obsolete]
    public static class To<T>
    {
        public static readonly Func<T, object> Signed = CompileSignedDelegate();
        public static readonly Func<T, object> Unsigned = CompileUnsignedDelegate();
        public static readonly Func<object, T> UnsignedAs = CompileUnsignedAsDelegate();

        static private Func<T, object> CompileSignedDelegate()
        {
            return DelegateHelpers.Compile<Func<T, object>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<T>();
                emiter.LoadArgument(0);
                var method = typeof(To).GetMethod("Signed", Types<T>.Array);
                emiter.Call(method);
                emiter.Box(method.ReturnType);
                emiter.Return();
            });
        }

        static private Func<T, object> CompileUnsignedDelegate()
```

```
        {
            return DelegateHelpers.Compile<Func<T, object>>(emiter =>
            {
                Ensure.Always.IsSignedInteger<T>();
                emiter.LoadArgument(0);
                var method = typeof(To).GetMethod("Unsigned", Types<T>.Array);
                emiter.Call(method);
                emiter.Box(method.ReturnType);
                emiter.Return();
            });
        }

        static private Func<object, T> CompileUnsignedAsDelegate()
        {
            return DelegateHelpers.Compile<Func<object, T>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<T>();
                emiter.LoadArgument(0);
                var signedVersion = NumericType<T>.SignedVersion;
                emiter.UnboxValue(signedVersion);
                var method = typeof(To).GetMethod("Unsigned", new[] { signedVersion });
                emiter.Call(method);
                emiter.Return();
            });
        }
    }
}
```

1.7 ./Platform.Converters/UncheckedConverter.cs

```
using System;
using System.Linq;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;
using Platform.Reflection;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Converters
{
    public abstract class UncheckedConverter<TSource, TTarget> : IConverter<TSource, TTarget>
    {
        public static UncheckedConverter<TSource, TTarget> Default { get; }

        static UncheckedConverter()
        {
            AssemblyName assemblyName = new AssemblyName(GetNewName());
            var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
            ↪  AssemblyBuilderAccess.Run);
            var module = assembly.DefineDynamicModule(GetNewName());
            var type = module.DefineType(GetNewName(), TypeAttributes.Public |
            ↪  TypeAttributes.Class | TypeAttributes.Sealed, typeof(UncheckedConverter<TSource,
            ↪  TTarget>));
            EmitMethod<Converter<TSource, TTarget>>(type, "Convert", (il) =>
            {
                il.LoadArgument(1);
                if (typeof(TSource) != typeof(TTarget))
                {
                    UncheckedConvert(il);
                }
                il.Return();
            });
            var typeInfo = type.CreateTypeInfo();
            Default = (UncheckedConverter<TSource, TTarget>)Activator.CreateInstance(typeInfo);
        }

        private static void UncheckedConvert(ILGenerator generator)
        {
            var type = typeof(TTarget);
            if (type == typeof(short))
            {
                generator.Emit(OpCodes.Conv_I2);
            }
            else if (type == typeof(ushort))
            {
                generator.Emit(OpCodes.Conv_U2);
            }
            else if (type == typeof(sbyte))
            {
```

```
                generator.Emit(OpCodes.Conv_I1);
            }
            else if (type == typeof(byte))
            {
                generator.Emit(OpCodes.Conv_U1);
            }
            else if (type == typeof(int))
            {
                generator.Emit(OpCodes.Conv_I4);
            }
            else if (type == typeof(uint))
            {
                generator.Emit(OpCodes.Conv_U4);
            }
            else if (type == typeof(long))
            {
                generator.Emit(OpCodes.Conv_I8);
            }
            else if (type == typeof(ulong))
            {
                generator.Emit(OpCodes.Conv_U8);
            }
            else if (type == typeof(float))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_R4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_R_Un);
                }
            }
            else if (type == typeof(double))
            {
                generator.Emit(OpCodes.Conv_R8);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        private static void EmitMethod<TDelegate>(TypeBuilder type, string methodName,
        ↪  Action<ILGenerator> emitCode)
        {
            var delegateType = typeof(TDelegate);
            var invoke = delegateType.GetMethod("Invoke");
            var returnType = invoke.ReturnType;
            var parameterTypes = invoke.GetParameters().Select(s => s.ParameterType).ToArray();
            MethodBuilder method = type.DefineMethod(methodName, MethodAttributes.Public |
            ↪  MethodAttributes.Virtual | MethodAttributes.Final | MethodAttributes.HideBySig,
            ↪  returnType, parameterTypes);
            method.SetImplementationFlags(MethodImplAttributes.IL | MethodImplAttributes.Managed
            ↪  | MethodImplAttributes.AggressiveInlining);
            var generator = method.GetILGenerator();
            emitCode(generator);
        }

        private static string GetNewName() => Guid.NewGuid().ToString("N");

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TTarget Convert(TSource source);
    }
}
```

1.8 ./Platform.Converters.Tests/ConverterTests.cs

```
using Platform.Diagnostics;
using System;
using System.Globalization;
using System.Runtime.CompilerServices;
using Xunit;
using Xunit.Abstractions;

namespace Platform.Converters.Tests
{
    public class ConverterTests
    {
        private readonly ITestOutputHelper _output;
```

```
13          private static readonly UncheckedConverter<ulong, ulong> _uInt64ToUInt64Converter =
            ↪  UncheckedConverter<ulong, ulong>.Default;
14          private static readonly UncheckedConverter<int, ulong> _int32ToUInt64converter =
            ↪  UncheckedConverter<int, ulong>.Default;
15
16          public ConverterTests(ITestOutputHelper output) => _output = output;
17
18          [Fact]
19          public void SameTypeTest()
20          {
21              var result = UncheckedConverter<ulong, ulong>.Default.Convert(2UL);
22              Assert.Equal(2UL, result);
23              result = CheckedConverter<ulong, ulong>.Default.Convert(2UL);
24              Assert.Equal(2UL, result);
25          }
26
27          [Fact]
28          public void SameTypePerformanceComparisonTest()
29          {
30              var N = 10000000;
31              var result = 0UL;
32
33              // Warmup
34              for (int i = 0; i < N; i++)
35              {
36                  result = _uInt64ToUInt64Converter.Convert(2UL);
37              }
38              for (int i = 0; i < N; i++)
39              {
40                  result = UncheckedConverter<ulong, ulong>.Default.Convert(2UL);
41              }
42              for (int i = 0; i < N; i++)
43              {
44                  result = Convert(2UL);
45              }
46              for (int i = 0; i < N; i++)
47              {
48                  result = To.UInt64(2UL);
49              }
50              for (int i = 0; i < N; i++)
51              {
52                  result = System.Convert.ToUInt64(2UL);
53              }
54              for (int i = 0; i < N; i++)
55              {
56                  result = (ulong)System.Convert.ChangeType(2UL, typeof(ulong));
57              }
58
59              var ts1 = Performance.Measure(() =>
60              {
61                  for (int i = 0; i < N; i++)
62                  {
63                      result = _uInt64ToUInt64Converter.Convert(2UL);
64                  }
65              });
66              var ts2 = Performance.Measure(() =>
67              {
68                  for (int i = 0; i < N; i++)
69                  {
70                      result = UncheckedConverter<ulong, ulong>.Default.Convert(2UL);
71                  }
72              });
73              var ts3 = Performance.Measure(() =>
74              {
75                  for (int i = 0; i < N; i++)
76                  {
77                      result = Convert(2UL);
78                  }
79              });
80              var ts4 = Performance.Measure(() =>
81              {
82                  for (int i = 0; i < N; i++)
83                  {
84                      result = To.UInt64(2UL);
85                  }
86              });
87              var ts5 = Performance.Measure(() =>
88              {
```

```
                for (int i = 0; i < N; i++)
                {
                    result = System.Convert.ToUInt64(2UL);
                }
            });
            var ts6 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = (ulong)System.Convert.ChangeType(2UL, typeof(ulong));
                }
            });
            IFormatProvider formatProvider = CultureInfo.InvariantCulture;
            var ts7 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = ((IConvertible)2UL).ToUInt64(formatProvider);
                }
            });
            var ts8 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = (ulong)((IConvertible)2UL).ToType(typeof(ulong), formatProvider);
                }
            });

            _output.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {ts8} {result}");
        }

        [Fact]
        public void Int32ToUInt64Test()
        {
            var result = UncheckedConverter<int, ulong>.Default.Convert(2);
            Assert.Equal(2UL, result);
            result = CheckedConverter<int, ulong>.Default.Convert(2);
            Assert.Equal(2UL, result);
        }

        [Fact]
        public void Int32ToUInt64PerformanceComparisonTest()
        {
            var N = 10000000;
            var result = 0UL;

            // Warmup
            for (int i = 0; i < N; i++)
            {
                result = _int32ToUInt64converter.Convert(2);
            }
            for (int i = 0; i < N; i++)
            {
                result = UncheckedConverter<ulong, ulong>.Default.Convert(2);
            }
            for (int i = 0; i < N; i++)
            {
                result = Convert(2);
            }
            for (int i = 0; i < N; i++)
            {
                result = To.UInt64(2);
            }
            for (int i = 0; i < N; i++)
            {
                result = System.Convert.ToUInt64(2);
            }
            for (int i = 0; i < N; i++)
            {
                result = (ulong)System.Convert.ChangeType(2, typeof(ulong));
            }

            var ts1 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = _int32ToUInt64converter.Convert(2);
                }
```

```
            });
            var ts2 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = UncheckedConverter<ulong, ulong>.Default.Convert(2);
                }
            });
            var ts3 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Convert(2);
                }
            });
            var ts4 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = To.UInt64(2);
                }
            });
            var ts5 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = System.Convert.ToUInt64(2);
                }
            });
            var ts6 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = (ulong)System.Convert.ChangeType(2, typeof(ulong));
                }
            });
            IFormatProvider formatProvider = CultureInfo.InvariantCulture;
            var ts7 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = ((IConvertible)2).ToUInt64(formatProvider);
                }
            });
            var ts8 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = (ulong)((IConvertible)2).ToType(typeof(ulong), formatProvider);
                }
            });

            _output.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {ts8} {result}");
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong Convert(ulong value) => _uInt64ToUInt64Converter.Convert(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong Convert(int value) => _int32ToUInt64converter.Convert(value);
    }
}
```

Index