¿Qué son la DLR y la CLR?

Common Language Runtime (CLR):

El framework .NET ofrece un entorno de ejecución llamado Common Language Runtime para los códigos de los programas que corren sobre esta plataforma y provee servicios que hacen el proceso de desarrollo mucho más fácil.

Los compiladores y las herramientas exponen la funcionalidad de Common Language Runtime y permiten escribir código con las ventajas que proporciona este entorno de ejecución administrado. El código desarrollado con un compilador de lenguaje orientado al tiempo de ejecución se denomina código administrado. Este código se beneficia de características como: la integración entre lenguajes, el control de excepciones entre lenguajes, la seguridad mejorada, la compatibilidad con la implementación y las versiones, un modelo simplificado de interacción y servicios de generación de perfiles y depuración.

Para permitir al motor en tiempo de ejecución proporcionar servicios al código administrado, los compiladores de lenguajes deben emitir metadatos que describen los tipos, los miembros y las referencias del código. Los metadatos se almacenan con el código; cada archivo ejecutable portable (PE) de Common Language Runtime cargable contiene metadatos. El motor en tiempo de ejecución utiliza los metadatos para localizar y cargar clases, colocar instancias en memoria, resolver invocaciones a métodos, generar código nativo, exigir mecanismos de seguridad y establecer los límites del contexto en tiempo de ejecución.

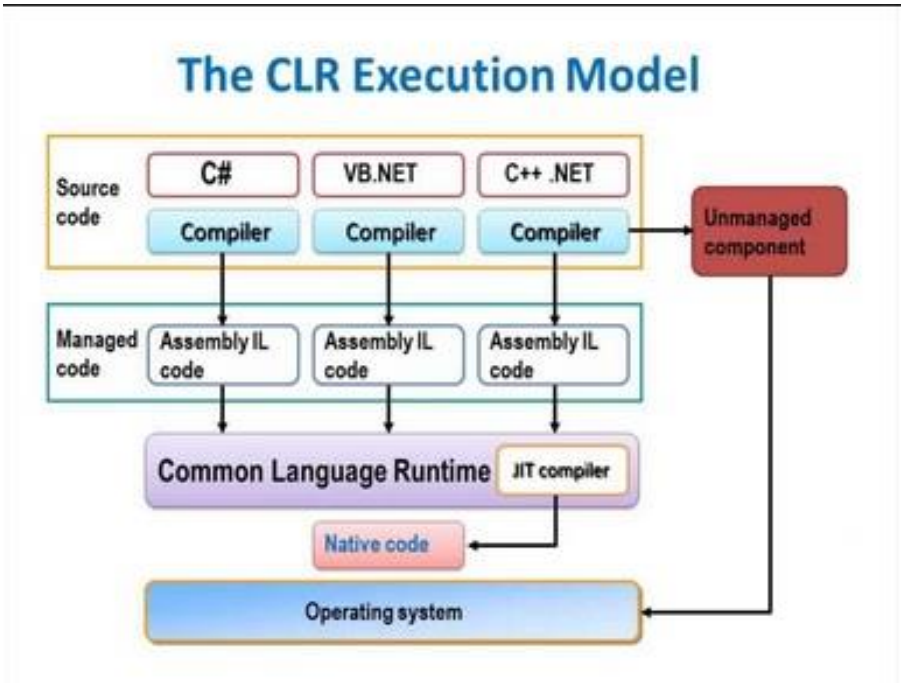
El tiempo de ejecución controla automáticamente la disposición de los objetos y administra las referencias a éstos, liberándolos cuando ya no se utilizan. Los objetos cuya duración se administra de esta forma se denominan datos administrados. La recolección de elementos no utilizados elimina pérdidas de memoria así como otros errores habituales de programación. Con un código administrado se pueden utilizar datos administrados, datos no administrados o estos dos tipos de datos en una aplicación .NET Framework. Como los compiladores de lenguajes proporcionan sus propios tipos, como tipos primitivos, no siempre se sabe (o no es necesario saber) si los datos se están administrando.

Common Language Runtime facilita el diseño de los componentes y de las aplicaciones cuyos objetos interactúan entre lenguajes distintos. Los objetos escritos en lenguajes diferentes pueden comunicarse entre sí, lo que permite integrar sus comportamientos de forma precisa. Por ejemplo, puede definir una clase y, a continuación, utilizar un lenguaje diferente para derivar una clase de la clase original o llamar a un método de la clase original. También se puede pasar al método de una clase una instancia de una clase escrita en un lenguaje diferente. Esta integración entre lenguajes diferentes es posible porque los compiladores y las herramientas de lenguajes orientados al motor en tiempo de ejecución utilizan un sistema de tipos común definido por el motor en tiempo de ejecución, y los lenguajes siguen las reglas en tiempo de ejecución para definir nuevos tipos, así como para crear, utilizar, almacenar y enlazar tipos.

Como parte de los metadatos, todos los componentes administrados contienen información sobre los componentes y los recursos utilizados en su compilación. El motor en tiempo de ejecución utiliza esta información para garantizar que el componente o la aplicación contiene las versiones especificadas de todo lo necesario, por lo que hay menos posibilidades de que la ejecución del código se interrumpa debido a una dependencia inadecuada. La información de registro y los datos de estado ya no se almacenan en el Registro, donde puede ser difícil establecer y mantener datos. En su lugar, la información sobre tipos definidos por el usuario (y sus dependencias) se almacena con el código como metadatos y, de este modo, las tareas de replicación y eliminación de componentes es mucho menos complicada.

Las herramientas y los compiladores de lenguajes exponen la funcionalidad del motor en tiempo de ejecución de forma que resulte útil e intuitiva para los programadores. Esto significa que algunas características en tiempo de ejecución pueden ser más evidentes en un entorno que en otro. El funcionamiento del motor en tiempo de ejecución depende de las herramientas y los compiladores utilizados. Por ejemplo, un programador de Visual Basic observará que con Common Language Runtime, el lenguaje Visual Basic contiene más características orientadas a objetos que antes. El motor en tiempo de ejecución ofrece las siguientes ventajas:

- Mejoras en el rendimiento.
- Capacidad para utilizar fácilmente componentes desarrollados en otros lenguajes.
- Tipos extensibles que proporciona una biblioteca de clases
- Características del lenguaje como herencia, interfaces y sobrecarga para la programación orientada a objetos.
- Compatibilidad con subprocesamiento libre explícito que permite la creación de aplicaciones multiproceso escalables.
- Compatibilidad con el control de excepciones estructurado.
- Compatibilidad con atributos personalizados.
- Recolección de elementos no utilizados.
- Emplee delegados en lugar de punteros a funciones para mayor seguridad y protección de tipos.



Dynamic Language Runtime (DLR):

Dynamic Language Runtime (DLR) es un entorno en tiempo de ejecución que agrega un conjunto de servicios para lenguajes dinámicos en Common Language Runtime (CLR). DLR hace más fácil desarrollar lenguajes dinámicos para ejecutarlos en .NET Framework y agregar características dinámicas a lenguajes de tipos estáticos.

Los lenguajes dinámicos pueden identificar el tipo de un objeto en tiempo de ejecución, mientras que en los lenguajes de tipos estáticos, como C# y Visual Basic (cuando se usa `Option Explicit On`) es necesario especificar los tipos de objeto en tiempo de diseño.

Entre los ejemplos de lenguajes dinámicos se incluyen Lisp, Smalltalk, JavaScript, PHP, Ruby, Python, ColdFusion, Lua, Cobra y Groovy.

La mayoría de los lenguajes dinámicos ofrece las siguientes ventajas a los desarrolladores:

- Capacidad de usar un bucle de comentarios rápido (bucle de lectura-evaluación-impresión o REPL). Esto permite escribir varias instrucciones y ejecutarlas inmediatamente para ver los resultados.
- Compatibilidad con el desarrollo de arriba abajo y el desarrollo más tradicional de abajo arriba. Por ejemplo, si usa un enfoque de arriba abajo, puede llamar a funciones que todavía no se han implementado y, después, agregar implementaciones subyacentes cuando las necesite.
- Refactorización y modificaciones de código más fáciles, ya que no es necesario cambiar las declaraciones de tipo estático en todo el código.

Los lenguajes dinámicos son excelentes lenguajes de scripting. Los clientes pueden extender fácilmente las aplicaciones creadas mediante lenguajes dinámicos con nuevos comandos y funcionalidades. Los lenguajes dinámicos también se suelen usar para crear sitios web y herramientas de ejecución de pruebas, mantener granjas de servidores, desarrollar diversas utilidades y realizar transformaciones de datos.

El propósito de DLR consiste en permitir que un sistema de lenguajes dinámicos se ejecute en .NET Framework y aportarles interoperabilidad .NET. DLR agrega objetos dinámicos en C# y Visual Basic para admitir el comportamiento dinámico en estos lenguajes y permitir su interoperabilidad con lenguajes dinámicos.

DLR también ayuda a crear bibliotecas que admiten operaciones dinámicas. Por ejemplo, si tiene una biblioteca que usa objetos XML o de notación de objetos JavaScript (JSON), los objetos pueden aparecer como objetos dinámicos en los lenguajes que usan DLR. Esto permite que los usuarios de la biblioteca escriban código sintácticamente más sencillo y más natural para trabajar con objetos y obtener acceso a miembros de objeto.

Por ejemplo, podría usar el código siguiente para incrementar un contador en XML en C#.

```
Scriptobj.SetProperty("Count", ((int)GetProperty("Count")) + 1);
```

Mediante el uso de DLR, podría usar el código siguiente en su lugar para la misma operación.

```
scriptobj.Count += 1;
```

Algunos ejemplos de lenguajes desarrollados con DLR son los siguientes:

- IronPython.
- IronRuby.

Principales ventajas de DLR

Simplifica el traslado de lenguajes dinámicos a .NET Framework

DLR evita a los implementadores del lenguaje crear analizadores léxicos, semánticos y de otro tipo, así como generadores de código y otras herramientas que tradicionalmente tenían que crear por sí mismos. Para usar DLR, un lenguaje debe generar *árboles de expresión*, que representan el código del nivel de lenguaje en una estructura en forma de árbol, rutinas del asistente en tiempo de ejecución y objetos dinámicos opcionales que implementan la interfaz IDynamicMetaObjectProvider. DLR y .NET Framework automatizan muchas de las tareas de análisis de código y generación de código. Esto permite que los implementadores del lenguaje se concentren en características únicas del lenguaje.

Habilita características dinámicas en lenguajes de tipos estáticos

Los lenguajes de .NET Framework existentes, como C# y Visual Basic, pueden crear objetos dinámicos y usarlos con objetos de tipos estáticos. Por ejemplo, C# y Visual Basic pueden usar objetos dinámicos para la reflexión .NET, HTML y Document Object Model (DOM).

Proporciona las futuras ventajas de DLR y .NET Framework

Los lenguajes implementados mediante DLR pueden beneficiarse de las mejoras futuras de DLR y .NET Framework. Por ejemplo, si .NET Framework lanza una nueva versión que tiene un recolector de elementos no utilizados mejorado o un tiempo de carga de ensamblados más rápido, los lenguajes implementados mediante DLR pueden beneficiarse de inmediato de dicha ventaja. Si DLR agrega optimizaciones como una mejor compilación, el rendimiento también mejora para todos los lenguajes implementados mediante DLR.

Habilita el uso compartido de bibliotecas y objetos

Los objetos y las bibliotecas implementados en un lenguaje pueden usarlos otros lenguajes. DLR también permite la interoperabilidad entre los lenguajes de tipos estáticos y los lenguajes dinámicos. Por ejemplo, C# puede declarar un objeto dinámico que usa una biblioteca escrita en un lenguaje dinámico. Al mismo tiempo, los lenguajes dinámicos pueden usar bibliotecas de .NET Framework.

Proporciona una distribución y una invocación dinámicas rápidas

DLR permite la ejecución rápida de operaciones dinámicas, ya que admite el almacenamiento en caché polimórfico avanzado. DLR crea reglas para operaciones de enlace que usan objetos en las implementaciones en tiempo de ejecución necesarias y, después, almacena en caché estas reglas para evitar los cálculos de enlace que agotan los recursos durante las ejecuciones sucesivas del mismo código en los mismos tipos de objetos.

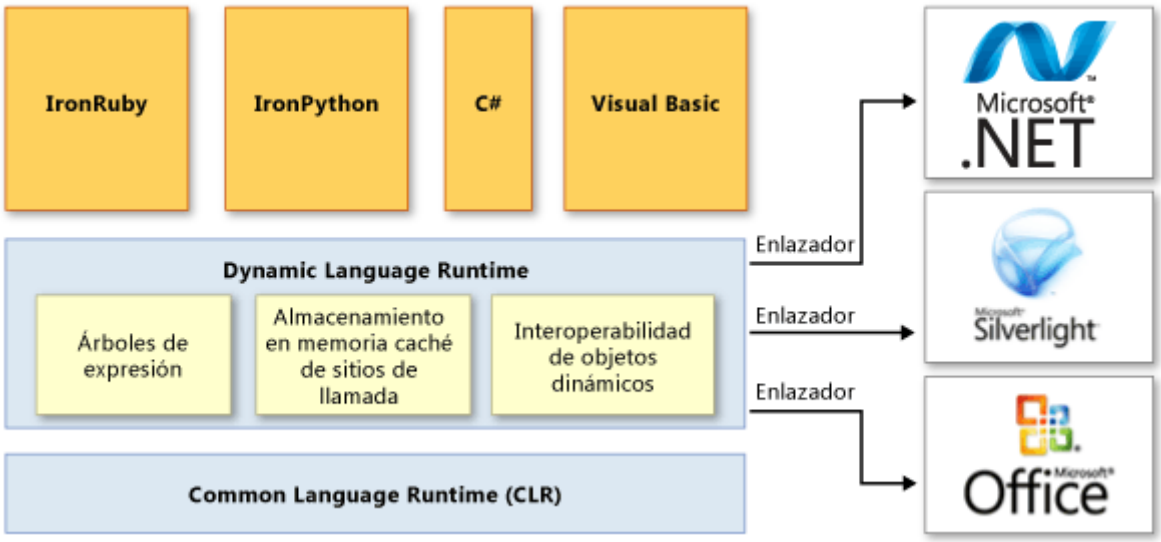
Ejemplo del uso de Dynamic:

```
namespace TestingDynamic
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            dynamic d = "test";
            Console.WriteLine(d.GetType());
            // Prints "System.String".

            d = 100;
            Console.WriteLine(d.GetType());
            // Prints "System.Int32".

            dynamic t= "test";

            // The following line throws an exception at run time.
            t++;
        }
    }
}
```

¿Están al mismo nivel en la arquitectura de .NET?

Arquitectura de DLR

En la ilustración siguiente se muestra la arquitectura de Dynamic Language Runtime.



DLR agrega un conjunto de servicios a CLR para una mejor compatibilidad con los lenguajes dinámicos. Entre estos servicios, se incluyen los siguientes:

- Árboles de expresión. DLR usa árboles de expresión para representar la semántica del lenguaje. Para ello, DLR ha extendido los árboles de expresión LINQ de modo que incluyan el flujo de control, la asignación y otros nodos de modelado del lenguaje.
- Almacenamiento en caché del sitio de llamada. Un *sitio de llamada dinámico* es un lugar del código donde se realiza una operación como a + b o a.b() en objetos dinámicos. DLR almacena en caché las características de a y b (generalmente, los tipos de estos objetos) e información sobre la operación. Si ya se ha realizado previamente una operación de este tipo, DLR recupera toda la información necesaria de la memoria caché para su distribución rápida.
- Interoperabilidad de objetos dinámicos. DLR proporciona un conjunto de clases e interfaces que representan operaciones y objetos dinámicos que pueden usar los implementadores del lenguaje y los autores de bibliotecas dinámicas. Entre estas clases e interfaces se incluyen IDynamicMetaObjectProvider, DynamicMetaObject, DynamicObject y ExpandoObject.

Por tanto en la arquitectura de .NET encontramos que DLR es solo una librería y no se encuentra al mismo nivel que CLR. Pues no tiene que lidiar con compilación JIT, garbage collector, administración de hilos y demás. Aunque DLR no maneje código nativo (que entienda la máquina) directamente, se puede ver como que hace un trabajo similar a CLR; Así como CLR convierte IL (Lenguaje Intermedio) en código nativo, DLR convierte código representado usando binders, call sites, meta-objects, y otros varios conceptos en árboles de expresiones que poder ser luego compilados en IL y eventualmente en código nativo por CLR.

¿Qué son un *call site*, *receiver* y *binder*?

Call Sites

Este es una especie de átomo del DLR, la más pequeña pieza de código que puede considerarse como una sola unidad ejecutable. Una expresión puede contener muchos call sites, pero el comportamiento se desarrolla de forma natural, evaluando un call site a la vez. Para el resto de la discusión, solo consideraremos un solo call site . Será útil tener un pequeño ejemplo de un call site al que hacer referencia, así que aquí hay uno simple, donde d es, por supuesto, una variable de tipo dinámico:
d.Foo (10);
El call site se representa en código como System.Runtime.CompilerServices.Call-Site <T>. Aquí hay un ejemplo del código que se puede llamar para crear el sitio para el fragmento anterior:

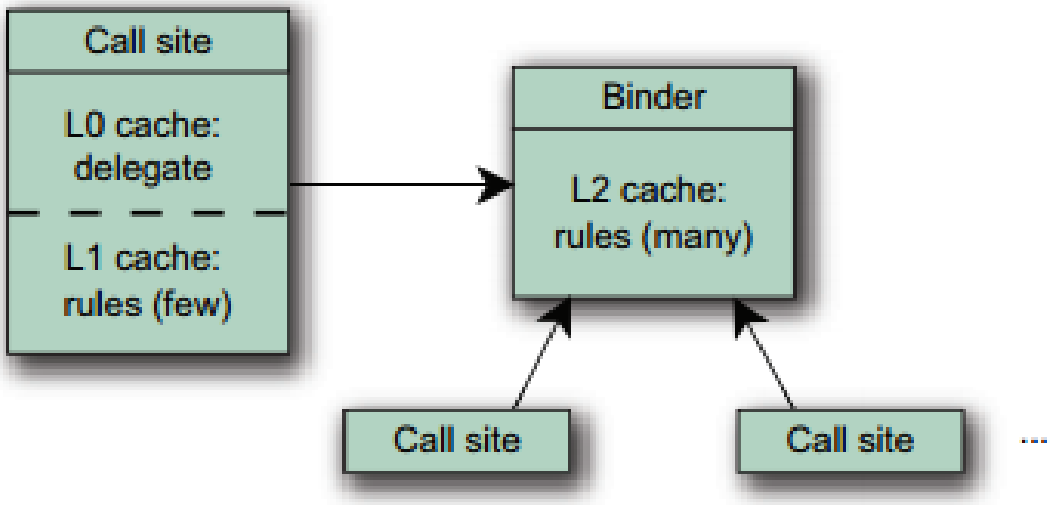
```
CallSite<Action<CallSite, object, int>>.Create(Binder.InvokeMember(
CSharpBinderFlags.ResultDiscarded, "Foo", null, typeof(Test),
new CSharpArgumentInfo[] {
CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null),
CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.Constant |
CSharpArgumentInfoFlags.UseCompileTimeType,
null)}));
```

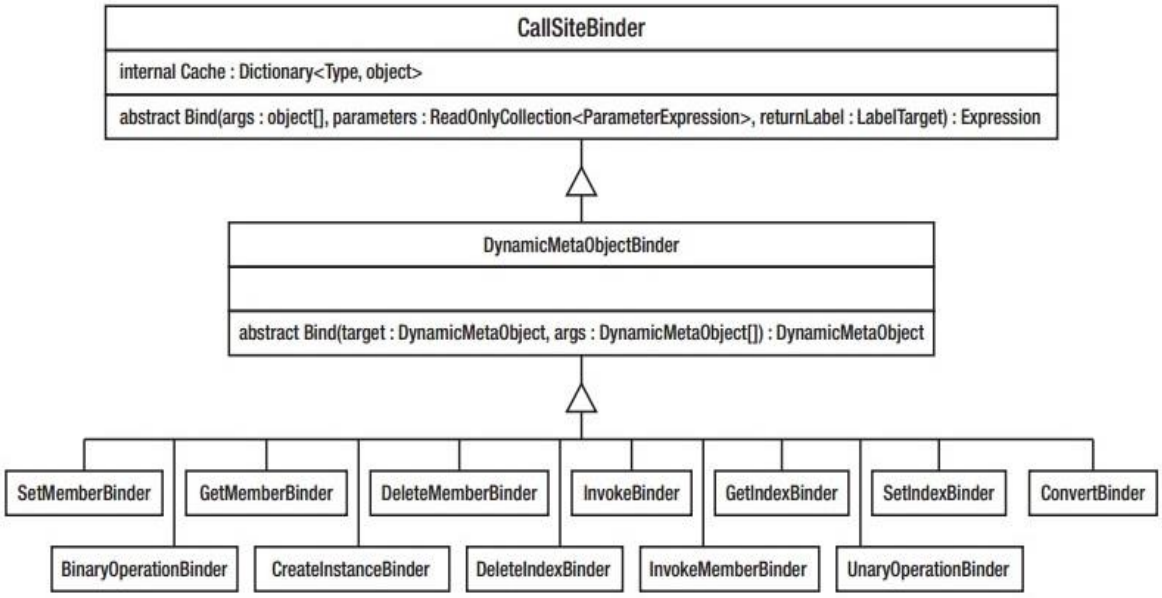
Ejemplo:

```
// Esta es una definición de función
 función sqr (x)
 {
   devolver x * x;
 }


  función foo () {
   // estos son dos call sites de la función sqr en esta función
   a = sqr (b);
   c = sqr (b);
 }
```

RECEVERS Y BINDERS

Además de un call sites, necesitamos algo para decidir qué significa y cómo ejecutarlo.
En el DLR, dos entidades pueden decidir esto: el receiver de una llamada y el binder .
El receiver de una llamada es simplemente el objeto sobre el que se llama a un miembro. En nuestra muestra de call site, el receiver es el objeto al que d se refiere en el momento de la ejecución. El binder depende del idioma de la llamada y forma parte del call site; en este caso, podemos ver que el compilador de C # emite código para crear un cuaderno usando Binder.InvokeMember. La clase Binder en este caso es Microsoft.CSharp.RuntimeBinder.Binder, por lo que realmente es C # -específico. El cuaderno de C # también es compatible con COM y realizará el COM apropiado vinculante si el receiver es un objeto IDispatch. El DLR siempre da prioridad al receiver: si es un objeto dinámico que sabe cómo manejar la llamada, luego usará cualquier ruta de ejecución que proporcione el objeto. El objeto puede anunciarse como dinámico implementando el nuevo IDynamicMeta-Interfaz ObjectProvider. El nombre es un bocado, pero solo contiene un solo miembro: GetMetaObject. Necesitarás ser un árbol de expresión ninja para implementarlo correctamente, además de conocer bastante bien el DLR. Pero en las manos correctas esto puede ser una herramienta poderosa, que le brinda interacción de nivel inferior con el DLR y su caché de ejecución. Si necesita implementar un comportamiento dinámico de alto rendimiento, vale la pena la inversión de aprender los detalles. Hay dos implementaciones públicas de IDynamicMetaObjectProvider incluido en el marco para facilitar la implementación con
comportamiento dinámico en situaciones donde el rendimiento no es tan crítico. La interfaz en si representa la capacidad de un objeto de reaccionar dinámicamente. Si el receiver no es dinámico, el archivador decide cómo se debe ejecutar el código. En nuestro código, aplicaría reglas específicas de C # al código y resolvería qué hacer. Si estuviera creando su propio lenguaje dinámico, podría implementar su propio lenguaje carpeta para decidir cómo debería comportarse en general (cuando el objeto no anula el comportamiento).





Explique que genera el compilador de C# para el siguiente código:

```
string text = "text to cut";
dynamic startIndex = 2;
string substring = text.Substring(startIndex);
```

Creando CALL SITES y BINDERS

No es necesario saber los detalles exactos de que hace el compilador con expresiones dinámicas para poder usarlas, pero es instructivo y sirve en casos que necesitemos ver como se ve el código decompilado por alguna razón y no sorprendernos en cómo se ven las partes dinámicas

Nuestro código es bastante sencillo, pero contiene dos operaciones con expresiones dinámicas, una es el llamado a Substring que recibe un parámetro dinámico y otra es la conversión(implícita) de dicho resultado de forma dinámica a un string

A continuación, veremos el código decompilado (recordar que está implícito la declaración de la clase Main con sus parámetros)
Veamos el código decompilado como luce

```
[CompilerGenerated]
private static class <Main>o__SiteContainer0 {
  public static CallSite<Func<CallSite, object, string>> <>p__Site1;
  public static CallSite<Func<CallSite, string, object, object>>
       <>p__Site2;
}

private static void Main() {
  string text = "text to cut";
  object startIndex = 2;
  if (<Main>o__SiteContainer0.<>p__Site1 == null) {
    <Main>o__SiteContainer0.<>p__Site1 =
      CallSite<Func<CallSite, object, string>>.Create(
        new CSharpConvertBinder(typeof(string),
          CSharpConversionKind.ImplicitConversion, false));
  }

  if (<Main>o__SiteContainer0.<>p__Site2 == null) {
    <Main>o__SiteContainer0.<>p__Site2 =
      CallSite<Func<CallSite, string, object, object>>.Create(
        new CSharpInvokeMemberBinder(CSharpCallFlags.None,
          "Substring", typeof(Snippet), null,
           new CSharpArgumentInfo[] {
             new CSharpArgumentInfo(
               CSharpArgumentInfoFlags.UseCompileTimeType, null),
             new CSharpArgumentInfo(
               CSharpArgumentInfoFlags.None, null) }));
  }

  string substring =
<Main>o__SiteContainer0.<>p__Site1.Target.Invoke(
  <Main>o__SiteContainer0.<>p__Site1,
  <Main>o__SiteContainer0.<>p__Site2.Target.Invoke(
    <Main>o__SiteContainer0.<>p__Site2, text, startIndex));

}
```

1 Call-sites storage

2 Creates conversion call site

3 Creates substring call site

4 Preserves text type

5 Invocation of both calls

Una clase estática anidada es utilizada para guardar todas las call sites (1) para el método, ya que se necesita crear una sola vez (no tiene sentido crearla varias veces, pues la cache sería inútil)

Es posible que las call sites se creen más de una vez debido al multithreading, pero si pasa es solo un poco menos eficiente.
Cada método que utiliza Dynamic binding tiene su propio contenedor, este tiene que ser el caso para métodos genéricos, ya que el call site tiene que cambiar según el tipo de argumentos que recibe de entrada

Después de crear las call sites(2 y 3 crean las call sites para la conversión y para el método Substring, que son las dos operaciones dinámicas que se iban a realizar) son invocados(5) cuando se hace "string substring = …." La llamada a Substring se realiza primero y luego se llama a la conversión implícita. El resultado ya es un valor estáticamente tipado y se asigna a la variable "substring"

> Nota: Analicemos rápido un aspecto adicional del código: como la información de algunos tipos estáticos es preservada en el call site. La información del tipo está presente en un delegado usado como argumento del call site (`Func<CallSite,string, object, object>`), y un flag en el correspondiente `CSharpArgumentInfo indica que la información de tipo va a ser utilizada en el binder (4).`

Muestre para que otros lenguajes además de C# la DLR brinda soporte. Compare la eficiencia con los intérpretes de los lenguajes originales.

Significado de Iron

Implementation

Running

On

.Net

Fantástica integración con .NET

- Fácil de utilizar bibliotecas de .NET
- Fácil de utilizar otros lenguajes de .NET
- Fácil de utilizar en .NET host
- Fácil de utilizar con herramientas de .NET

IronPython

es una implementación del intérprete Python (CPython) escrita totalmente en C#. El proyecto trata de seguir al pie de la letra el lenguaje Python, como implementación de Python que es. Esto hace que cualquier programa escrito en Python pueda ser interpretado con IronPython, con las ventajas añadidas de poder usar las bibliotecas de la plataforma .NET y poder compilar el código a bytecode

Características de IronPython

- Modo interactivo, heredado de Python. Al igual que en Python, consiste en un shell que interpreta las órdenes de forma interactiva.
- Soporte completo de la sintaxis y las bibliotecas (API) de Python.
- Integración con la plataforma .NET y sus bibliotecas.
- Compilación del código a bytecode, de forma que puede usarse en cualquier otro lenguaje soportado por la plataforma .NET.
- Integración del intérprete de IronPython en cualquier aplicación .NET para extender sus funcionalidades de forma sencilla.

La siguiente tabla muestra los tiempos para cada punto de referencia en IronPython and Python 3:

Program	Source Code	CPU secs	Elapsed secs	Memory KB	Code B	≈ CPU Load
thread-ring						
IronPython		1.87	**2.03**	63,780	407	60% 27% 2% 3% †
Python 3		34.13	24.18	13,204	448	23% 37% 20% 48% †
n-body						
IronPython		56.37	56.69	63,324	1337	30% 0% 68% 0%
Python 3		46.53	46.90	8,812	1315	21% 0% 83% 1%
richards						
IronPython		6.50	6.41	63,612	2423	1% 1% 93% 7%
Python 3		1.09	1.10	9,008	2434	5% 0% 0% 100%
fibonacci						
IronPython		5.86	5.86	50,872	181	0% 0% 100% 0%
Python 3		0.68	0.68	9,316	182	6% 0% 100% 0%
regex-dna						
IronPython		0.73	0.73	45,660	501	5% 100% 3% 6%
Python 3		0.05	0.06	?	524	17% 100% 0% 0%
k-nucleotide						
IronPython		1.15	1.57	80,896	593	56% 4% 16% 3% †
Python 3		0.09	0.07	?	801	14% 14% 14% 88% †
spectral-norm						
IronPython		4.00	4.19	57,200	594	0% 92% 1% 1%
Python 3		0.14	0.15	?	394	13% 100% 0% 0%
pidigits						
IronPython		13.14	12.69	78,604	322	0% 57% 3% 42%
Python 3		0.03	0.03	?	379	100% 0% 0% 0%
fannkuch-redux						
			No program			
Python 3		13.17	3.47	47,392	894	97% 99% 91% 99%
reverse-complement						
			No program			
Python 3		0.02	0.03	?	449	0% 100% 0% 0%
fasta						
			No program			
Python 3		7.57	4.17	81,704	2016	40% 27% 59% 61%
pystone						
IronPython		1.18	1.43	64,960	2301	32% 26% 3% 24%
			No program			
fasta-redux						
			No program			
Python 3		0.29	0.29	1,788	1115	4% 0% 3% 100%
chameneos-redux						
			No program			
Python 3		353.29	175.80	8,956	1191	48% 40% 47% 47%
templates						
			No program			
Python 3		1.10	1.16	19,872	322	5% 2% 98% 0%
jsonbench						
			No program			
Python 3		3.36	3.37	12,812	322	5% 100% 0% 0%
iobench						
			No program			
Python 3		5.69	5.69	8,892	367	5% 0% 100% 1%
fib50						
			No program			
Python 3		3.50	3.50	8,600	136	6% 1% 100% 0%
meteor-contest						
			No program			
Python 3		3.24	3.24	8,944	1443	5% 1% 100% 1%
binary-trees						
			No program			
Python 3		3.20	0.96	57,316	741	89% 84% 81% 88%
mandelbrot						
			No programs			
binary-trees-redux						
			No programs			

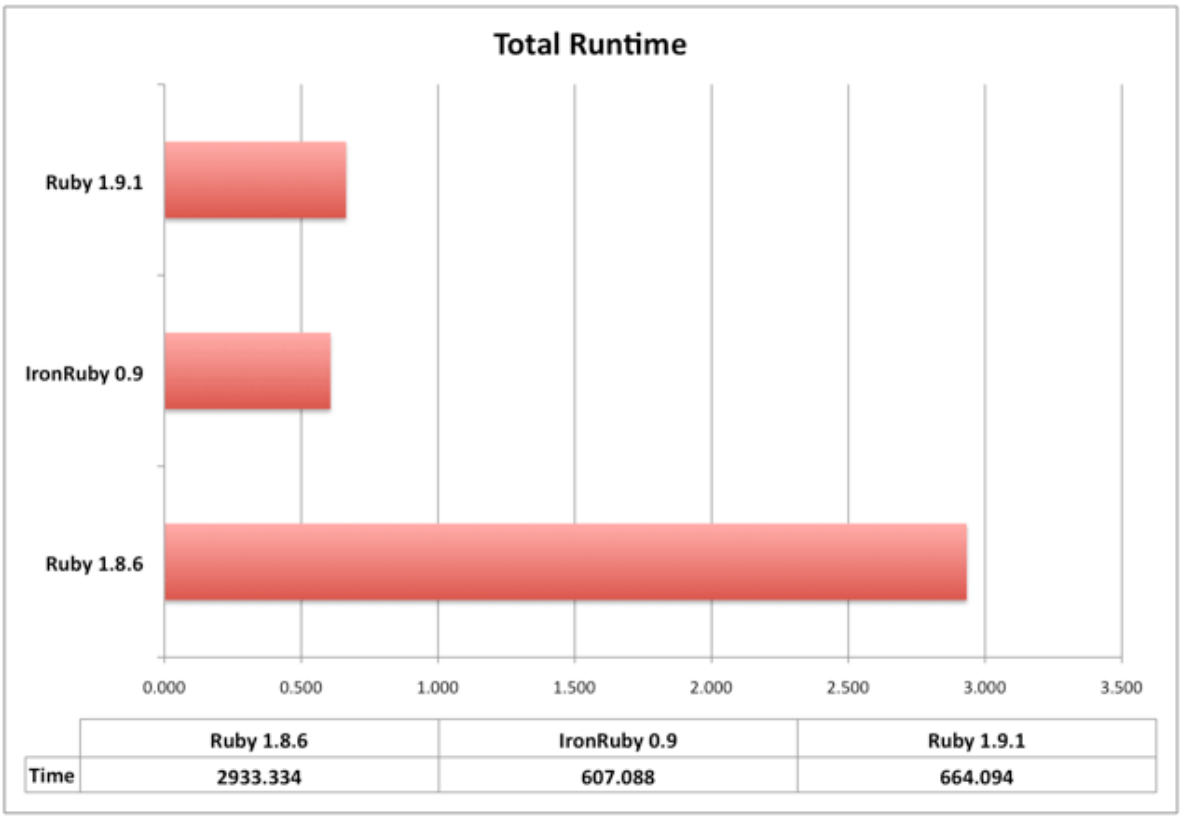
IronRuby

IronRuby es una implementación del lenguaje de programación Ruby dirigido a Microsoft .NET Framework . Se implementa sobre Dynamic Language Runtime (DLR), una biblioteca que se ejecuta sobre Common Language Infrastructure que proporciona tipado dinámico y envío de métodos dinámicos, entre otras cosas, para lenguajes dinámicos.

La siguiente tabla muestra los tiempos para cada punto de referencia en IronRuby 0.9, Ruby 1.8.6 (2008) and Ruby 1.9.1 (2009):

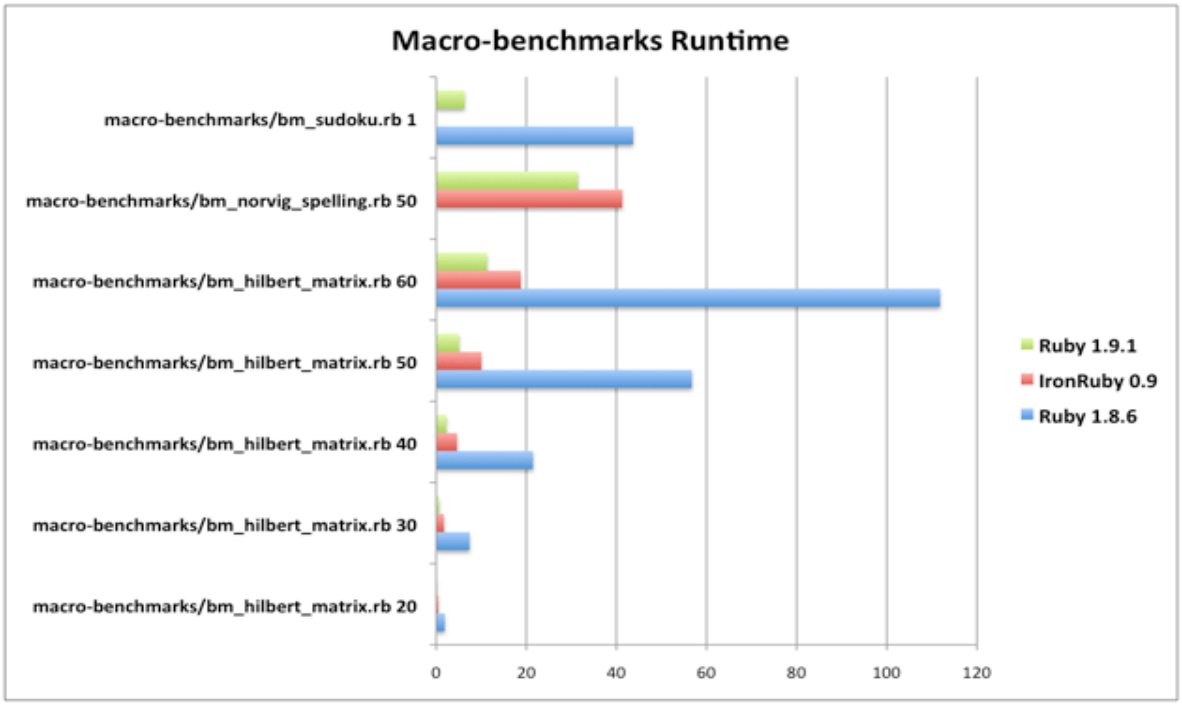
Benchmark File	#	Ruby 1.8.6	IronRuby	Ruby 1.9.1
macro-benchmarks/bm_gzip.rb	100	Timeout	IOError	N/A
macro-benchmarks/bm_hilbert_matrix.rb	20	1.891	0.453	0.125
macro-benchmarks/bm_hilbert_matrix.rb	30	7.422	1.719	0.656
macro-benchmarks/bm_hilbert_matrix.rb	40	21.500	4.625	2.266
macro-benchmarks/bm_hilbert_matrix.rb	50	56.765	10.031	5.109
macro-benchmarks/bm_hilbert_matrix.rb	60	111.859	18.781	11.297
macro-benchmarks/bm_norvig_spelling.rb	50	Timeout	41.313	31.453
macro-benchmarks/bm_sudoku.rb	1	43.734	Timeout	6.313
micro-benchmarks/bm_app_factorial.rb	5000	1.328	0.063	0.266
micro-benchmarks/bm_app_fib.rb	30	6.156	0.594	0.813
micro-benchmarks/bm_app_fib.rb	35	74.125	6.922	9.344
micro-benchmarks/bm_app_mandelbrot.rb	1	11.953	6.922	0.641
micro-benchmarks/bm_app_pentomino.rb	1	Timeout	59.938	75.859
micro-benchmarks/bm_app_strconcat.rb	1.5M	30.469	2.141	4.813
micro-benchmarks/bm_app_tak.rb	7	5.516	0.531	0.578
micro-benchmarks/bm_app_tak.rb	8	15.609	1.484	1.703
micro-benchmarks/bm_app_tak.rb	9	45.843	3.953	4.531
micro-benchmarks/bm_app_tarai.rb	3	19.985	1.844	2.156
micro-benchmarks/bm_app_tarai.rb	4	19.796	2.219	2.656
micro-benchmarks/bm_app_tarai.rb	5	24.235	2.688	3.063
micro-benchmarks/bm_binary_trees.rb	1	Timeout	53.078	37.375
micro-benchmarks/bm_count_multithreaded.rb	16	0.297	0.266	0.328
micro-benchmarks/bm_count_shared_thread.rb	16	0.250	0.188	0.203
micro-benchmarks/bm_mandelbrot.rb	1	Timeout	65.781	81.766
micro-benchmarks/bm_mbari_bogus1.rb	1	0.031	40.406	8.781
micro-benchmarks/bm_mbari_bogus2.rb	1	0.156	Timeout	N/A
micro-benchmarks/bm_mergesort_hongli.rb	3000	25.282	3.531	6.031
micro-benchmarks/bm_mergesort.rb	1	24.735	3.906	3.219
micro-benchmarks/bm_meteor_contest.rb	1	147.704	19.713	19.781
micro-benchmarks/bm_monte_carlo_pi.rb	10M	79.406	5.109	20.672
micro-benchmarks/bm_nbody.rb	100K	37.625	8.281	10.938
micro-benchmarks/bm_nsieve_bits.rb	8	69.656	33.156	6.531
micro-benchmarks/bm_nsieve.rb	9	58.344	5.453	N/A
micro-benchmarks/bm_partial_sums.rb	2.5M	93.391	10.797	26.422
micro-benchmarks/bm_pathname.rb	100	Timeout	Timeout	Timeout
micro-benchmarks/bm_primes.rb	3000	21.359	9.594	0.031
micro-benchmarks/bm_mandelbrot.rb	1	Timeout	65.781	81.766
micro-benchmarks/bm_mbari_bogus1.rb	1	0.031	40.406	8.781
micro-benchmarks/bm_mbari_bogus2.rb	1	0.156	Timeout	N/A
micro-benchmarks/bm_mergesort_hongli.rb	3000	25.282	3.531	6.031
micro-benchmarks/bm_mergesort.rb	1	24.735	3.906	3.219
micro-benchmarks/bm_meteor_contest.rb	1	147.704	19.713	19.781
micro-benchmarks/bm_monte_carlo_pi.rb	10M	79.406	5.109	20.672
micro-benchmarks/bm_nbody.rb	100K	37.625	8.281	10.938
micro-benchmarks/bm_nsieve_bits.rb	8	69.656	33.156	6.531
micro-benchmarks/bm_nsieve.rb	9	58.344	5.453	N/A
micro-benchmarks/bm_partial_sums.rb	2.5M	93.391	10.797	26.422
micro-benchmarks/bm_pathname.rb	100	Timeout	Timeout	Timeout
micro-benchmarks/bm_primes.rb	3000	21.359	9.594	0.031
micro-benchmarks/bm_primes.rb	30K	Timeout	Timeout	0.469
micro-benchmarks/bm_primes.rb	300K	Timeout	Timeout	5.281
micro-benchmarks/bm_primes.rb	3M	Timeout	Timeout	100.406
micro-benchmarks/bm_quicksort.rb	1	51.046	11.594	8.703
micro-benchmarks/bm_regex_dna.rb	20	181.172	21.188	11.938
micro-benchmarks/bm_reverse_compliment.rb	1	61.875	48.469	138.047
micro-benchmarks/bm_so_ackermann.rb	7	2.234	0.563	0.484
micro-benchmarks/bm_so_ackermann.rb	9	50.000	14.938	9.281
micro-benchmarks/bm_so_array.rb	9000	26.328	8.984	10.781
micro-benchmarks/bm_so_count_words.rb	100	Timeout	60.688	42.250
micro-benchmarks/bm_so_exception.rb	500K	78.125	Timeout	32.672
micro-benchmarks/bm_so_lists_small.rb	1000	13.906	4.250	3.172
micro-benchmarks/bm_so_lists.rb	1000	64.531	22.266	16.797
micro-benchmarks/bm_so_matrix.rb	60	8.312	2.781	2.125
micro-benchmarks/bm_so_object.rb	500K	16.375	5.313	1.672
micro-benchmarks/bm_so_object.rb	1M	29.312	10.500	2.844
micro-benchmarks/bm_so_object.rb	1.5M	43.312	16.000	4.281
micro-benchmarks/bm_so_sieve.rb	4000	241.922	37.859	35.688
micro-benchmarks/bm_socket_transfer_1mb.rb	10K	13.266	SocketError	3.359
micro-benchmarks/bm_spectral_norm.rb	100	5.110	0.922	0.719
micro-benchmarks/bm_sum_file.rb	100	Timeout	20.406	23.797
micro-benchmarks/bm_word_anagrams.rb	1	70.828	30.188	8.125
TOTAL TIME	-	2933.334	607.088	664.094

El total de los tiempos se resume en la siguiente gráfica:



	Ruby 1.8.6	IronRuby 0.9	Ruby 1.9.1
Time	2933.334	607.088	664.094

Comparando cada macro de puntos de referencias de forma individual:



Compare la estrategia de implementación de dynamic de C# con la propuesta e implementada para Java.

El enlace dinámico o tardío es un mecanismo en el que el método que se llama sobre un objeto o la función que se llama con argumentos se busca por nombre en tiempo de ejecución.

Con el enlace temprano o estático la fase de compilación corrige todos los tipos de variables y expresiones. Esto generalmente se almacena en el programa compilado como un desplazamiento en una tabla de método virtual ("v-table") y es muy eficiente. Con la vinculación tardía, el compilador no lee suficiente información para verificar que el método existe o vincula su ranura en la v-table. En cambio, el método se busca por nombre en tiempo de ejecución.

La principal ventaja de usar el enlace tardío es que no requiere que el compilador haga referencia a las bibliotecas que contienen el objeto en el momento de la compilación. Esto hace que el proceso de compilación sea más resistente a los conflictos de versión, en los que la v-table de la clase puede modificarse accidentalmente. Esto no es una preocupación en plataformas como .NET o Java, porque la máquina virtual crea la v-table en tiempo de ejecución contra las bibliotecas a medida que se cargan en la aplicación en ejecución.

Con C # 4, el lenguaje agregó el pseudotipo ***dynamic***. Esto se usaría como tipo de objeto para indicar que se desea un enlace tardío, es decir, el compilador no verifica el tipo de la variable de tipo dinámico en tiempo de compilación, en lugar de esto, el compilador obtiene el tipo en tiempo de ejecución.

Existen varios casos de uso para el tipo dynamic de C#, por ejemplo: un uso menos detallado de reflection, al convertir una instancia en el tipo dynamic, las propiedades, métodos, eventos, etc. Se pueden invocar directamente en la instancia sin usar directamente la API de reflection; otro caso es la interoperabilidad con lenguajes dinámicos, el tipo dynamic viene con un soporte para implementar objetos tipados dinámicamente e infraestructura común en tiempo de ejecución para una búsqueda eficiente de miembros; también la creación de abstracciones dinámicas en ejecución es otro uso, que podría proporcionar un acceso más simple a modelos de objetos de documentos, como documentos XML o XHTML.

Java no admite un uso de enlace tardío. Los casos de uso para el tipo dinámico de C# tienen diferentes constructores correspondientes en Java, por ejemplo: para la invocación dinámica de nombre de tipos preexistentes se debe utilizar reflection; para la interoperabilidad con lenguajes dinámicos se debe utilizar alguna forma de API de interoperabilidad específica para ese idioma. La plataforma de máquina virtual de Java tiene múltiples lenguajes dinámicos implementados, pero no hay un estándar común sobre como pasar objetos entre idiomas. Por lo general, esto implica alguna forma de reflection o API similar a esta; para crear e interactuar con objetos completamente en tiempo de ejecución (como por ejemplo, interacción con una abstracción de modelo de objeto de documento) se debe utilizar también una API de abstracción específica.