

# Todo sobre CLR, Mida pronto y periódicamente el rendimiento

Vance Morrison

Mi trabajo como arquitecto de rendimiento en el equipo de lenguaje común en tiempo de ejecución de Microsoft® .NET Framework consiste en ayudar a las personas a utilizar mejor el tiempo de ejecución cuando escriben aplicaciones de alto rendimiento. La verdad es que no tiene ningún misterio en ningún caso; basta con diseñar aplicaciones que rindan desde un principio. Existen demasiadas aplicaciones escritas sin pensar en absoluto en el rendimiento. A menudo, eso no es un problema ya que la mayoría de programas realizan cálculos relativamente pequeños y, además, son mucho más rápidos que los humanos con los que interactúan. Desgraciadamente, cuando se presenta la necesidad del alto rendimiento, carecemos del conocimiento, las habilidades y las herramientas para hacer un buen trabajo.

Aquí veremos lo que se necesita para escribir aplicaciones de alto rendimiento. Aunque los conceptos son universales, me centraré en programas diseñados para .NET. Con .NET resulta mucho más sencillo insertar operaciones costosas en una ruta de acceso de código de escaso rendimiento, ya que .NET abstrae el equipo subyacente mejor que un típico compilador C++ y ofrece, además, elementos como la reflexión, los atributos personalizados, las expresiones regulares, etc., que son eficaces, pero caros. Para ayudar a evitar ese gasto, mostraré cómo cuantificar el costo de varias características de .NET para que sepa cuándo conviene usarlas.

## Tenga un plan

Como ya he mencionado, la mayoría de los programas se escriben sin pensar en el rendimiento, y sin embargo, todos los proyectos deberían incluir un plan para lograrlo. Se deberían considerar varias situaciones de usuario y pensar en lo que significa un rendimiento excelente, un rendimiento bueno y un rendimiento malo. A continuación, basándonos en el volumen de datos, la complejidad algorítmica y nuestra experiencia anterior en la creación de aplicaciones similares, debemos decidir si se pueden alcanzar fácilmente los objetivos de rendimiento que hemos definido. Para muchas aplicaciones GUI, los objetivos son modestos, así que es fácil lograr al menos un buen rendimiento sin ningún diseño especial. Si este es el caso, el plan del rendimiento ya está listo.

Si no estamos seguros de alcanzar el objetivo de rendimiento, necesitaremos escribir un plan enumerando las áreas que probablemente presenten problemas.

Los problemas típicos son el tiempo de inicio, las operaciones de datos masivos y las animaciones gráficas.

### Ejemplo de procesamiento de datos de perfil

Con un ejemplo lo veremos más claro. En la actualidad estoy diseñando la infraestructura .NET para procesar datos de perfil. Necesito presentar de un modo significativo una lista de eventos (errores de página, E/S de disco, cambios de contexto, etc.) generados por el sistema operativo. Los archivos de datos implicados tienden a ser grandes. Los perfiles pequeños rondan los 10 MB y son frecuentes los que pasan ampliamente de 1 GB.

Trabajando en mi plan de rendimiento, llegué a la conclusión de que la visualización de datos no resultaría un problema si medía sólo las partes del conjunto de datos necesarias para componer la pantalla; es decir, una pantalla "perezosa". Desgraciadamente, se necesita un esfuerzo extra para hacer que objetos GUI como los controles de árbol, los controles de la lista y los cuadros de texto se vuelvan perezosos. Y por eso la mayoría de los editores de texto tienen un rendimiento inaceptable cuando el tamaño del archivo es demasiado grande (digamos, por ejemplo, 100 MB). Si hubiera diseñado la GUI sin pensar en el rendimiento, el resultado habría sido, casi con total seguridad, inaceptable.

La pereza, sin embargo, no ayuda en las operaciones que necesitan usar todos los datos del archivo (al computar los resúmenes, por ejemplo). Dado el tamaño del conjunto de datos, el envío de datos y los métodos de procesamiento son rutas de acceso de código complicadas que hay que diseñar con cuidado. La mayor parte del resto del programa no debería tener problemas graves de rendimiento y no necesita especial atención.

Esta es una situación habitual. Incluso en escenarios de alto rendimiento, el 95 por ciento de la aplicación no necesita una planeación de rendimiento, pero sí que es necesario identificar ese 5 por ciento restante. Además, no resulta difícil determinar qué parte del programa constituye ese 5 por ciento tan importante.

### Mida pronto y periódicamente

El paso siguiente en el diseño de alto rendimiento es medir. Antes de escribir código, es necesario saber si los objetivos de rendimiento son factibles, y si es así, de qué forma restringen el diseño. En mi caso, necesito saber los costos de operaciones básicas consideradas en el diseño, como la E/S de archivos sin procesar y el acceso a la base de datos. Para seguir avanzando, necesito algunos números. Este es el momento más importante en el diseño del proyecto.

Tristemente, la mayoría del rendimiento se pierde ya al principio del proceso de desarrollo. Una vez elegidas las estructuras de datos para el núcleo del programa, el perfil de rendimiento de la aplicación ya no se puede cambiar. La elección de algoritmos limita aún más el rendimiento, al igual que la selección de contratos de interfaz entre varios subcomponentes. Es muy importante que entienda los costos de todo esto y decida con sensatez.

El diseño es un proceso iterativo. Es mejor empezar por la elección más clara, más sencilla y más obvia. Entonces se puede hacer un dibujo del diseño (recomiendo un prototipo de código en caliente) y evaluar el rendimiento. También habría que pensar en cómo sería el aspecto del diseño si el rendimiento fuese el único factor a tener en cuenta y hacer una estimación de lo rápida que sería esa aplicación. Y ahora comienza la parte divertida de la ingeniería. Hay que comenzar a jugar con el diseño y pensar en alternativas entre estos dos extremos, buscando los diseños que den el mejor resultado.

De nuevo, mi experiencia con el procesador de datos de perfil es instructiva. Como en la mayoría de proyectos, la elección de la representación de datos era muy importante. ¿Deberían ser datos en memoria? ¿O transmitirse en secuencias mediante un archivo? ¿O estar en una base de datos? La solución estándar es que ningún conjunto de datos grande se debería almacenar en una base de datos. Sin embargo, las bases de datos están optimizadas para cambios relativamente lentos, no para tener grandes volúmenes de datos que cambian con frecuencia. Mi aplicación volcaría automáticamente muchos gigabytes de datos en la base de datos de manera rutinaria. ¿Podría la base de datos hacer frente a esta situación? Con unas mediciones y un análisis de las operaciones de base de datos, fue fácil confirmar que las bases de datos no tenían el perfil de rendimiento que necesitaba.

Tras realizar algunos cálculos más sobre cuánta memoria puede utilizar una aplicación antes de inducir demasiados errores de página, deseché también la solución en memoria. Y la única opción que me quedaba era la transmisión en secuencias de datos desde un archivo para mi representación básica de datos.

Sin embargo, todavía quedaban muchas otras decisiones sobre el diseño por hacer. La forma básica de los datos de perfil es una lista de eventos heterogéneos. ¿Pero a qué deben parecerse los eventos? ¿Son cadenas (bien uniformadas)? ¿Son objetos o estructuras C#?

Si son objetos, la solución obvia sería hacer una asignación por evento, lo cual son muchas asignaciones. ¿Es eso aceptable? ¿Cómo funciona el envío al iterar los eventos? ¿Son un modelo de devolución de llamada o un modelo de iteración? ¿El envío funciona sobre interfaces, delegados, o reflexión? Quedaban muchas decisiones por hacer relacionadas con el diseño y todas tendrían repercusiones

sobre el rendimiento final del programa, así que necesitaba tomar medidas para entender los elementos de compensación.

## La logística de la medición

Algo seguro durante el proceso de diseño es que realizará muchas mediciones. Así que, ¿cómo se hacen exactamente? Existen muchas herramientas útiles de generación de perfiles, pero una técnica de uso general que es también la más simple y la más fácil de conseguir es la microprueba comparativa. La técnica es sencilla: cuando quiera saber el costo de una operación en particular, simplemente tiene que establecer un ejemplo de su uso y medir directamente cuánto tiempo tarda la operación.

El .NET Framework tiene un temporizador de alta resolución denominado System.Diagnostics.Stopwatch que se diseñó específicamente para este fin. Aunque varía en función del hardware, la resolución es normalmente inferior a 1 microsegundo, lo cual es más que suficiente. Y como viene con el .NET Framework, ya tenemos la funcionalidad que necesitamos.

Aunque el Stopwatch está muy bien para empezar, es mejor un agente de prueba. Las operaciones pequeñas deben colocarse en bucles para que el intervalo sea lo suficientemente largo para medir con exactitud. El banco de pruebas debe ejecutarse una vez antes de tomar medidas para asegurarse de que cualquier compilación "justo a tiempo" (JIT, Just in Time) y otra inicialización de un solo uso se hayan completado (a menos, claro está, que el objetivo sea medir esa inicialización). Debido a que las medidas generan ruido, hay que ejecutar varias veces el banco de pruebas y hacer una estadística para determinar la estabilidad de la medición. No debería ser complicado ejecutar masivamente muchos bancos de pruebas (variaciones del diseño) y obtener un informe que muestre todos los resultados para establecer la comparación.

He escrito un agente de prueba denominado MeasureIt.exe. Este agente se crea a partir de la clase Stopwatch y aborda la consecución de estos objetivos. Está disponible con la descarga de código correspondiente a esta columna en el sitio web de MSDN® Magazine. Después de desempaquetar, sólo tiene que escribir lo siguiente para ejecutarlo:

```
MeasureIt
```

En unos segundos ejecutará un conjunto de más de 50 bancos de pruebas estándar y mostrará los resultados como una página web. En la [Figura 1](#) se muestra un extracto de los datos. En estos resultados, cada medida realiza una operación 10.000 veces (la operación se clona 10 veces en un bucle que se ejecuta 1.000 veces). A continuación, cada medida actúa 10 veces y se calculan las

estadísticas habituales (mínimo, máximo, mediana, promedio, desviación estándar).

<b>Operación medida</b>	<b>Mediana</b>	<b>Promedio</b>	<b>Desvest</b>	<b>Mín.</b>	<b>Máx.</b>	<b>Ejemplos</b>
MethodCalls: EmptyStaticFunction () [count=1000 scale=10.0]	1.000	1.005	0.084	0.922	1.136	10
MethodCalls: aClass.Interface() [count=1000 scale=10.0]	1.699	1.769	0.090	1.696	1.943	10
ObjectOps: new Class() [count=1000 scale=10.0]	6.248	8.040	3.556	5.087	16.296	10
Matrices: aIntArray [yo] = 1 [count=1000 scale=10.0]	0.616	0.638	0.071	0.612	0.850	10
Delegados: aInstanceDelegate () [count=1000 scale=10.0]	1.233	1.244	0.088	1.160	1.398	10
PInvoke: FullTrustCall () [count=1000]	7.452	6.946	0.804	5.878	7.913	10
Bloqueos: Monitor lock [count=1000]	11.487	12.129	0.901	11.322	13.843	10

Para que las mediciones de tiempo sean más significativas, se normalizan para que el tiempo de mediana para llamada (y retorno) desde una función estática vacía sea una unidad. No es raro que los bancos de pruebas tengan tiempos muy variados y por eso es importante contar con toda la información estadística. Fíjese en la gran diferencia entre los tiempos mínimo (71.299) y máximo (953.864) correspondientes al banco de pruebas FinalizableClass. Hay que encontrar una explicación para estas variaciones antes de confiar en los datos del banco de pruebas. En este caso particular, es el resultado del tiempo de ejecución que ejecuta periódicamente rutas de acceso a código más lentas para asignar masivamente las estructuras de datos de contabilidad. Disponer de estas estadísticas resulta ya útil en la validación de los datos.

Esta tabla es una mina de oro de datos de rendimiento útiles. Detalla los costos de la mayor parte de las operaciones básicas usadas por el código destinado a .NET. En la siguiente entrega de esta columna aportaré más detalles, pero antes quisiera

explicar una característica importante de MeasureIt: Viene con su propio código fuente. Para desempaquetar el código fuente de MeasureIt e iniciar Visual Studio® para que lo examine (si Visual Studio está disponible), hay que escribir lo siguiente:

```
MeasureIt /edit
```

Disponer del origen significa que se puede entender de forma rápida y exacta lo que el banco de pruebas está midiendo. Significa también que se puede agregar fácilmente otro banco de pruebas al conjunto de aplicaciones.

Al igual que antes, mi experiencia con el procesador de datos de perfil vuelve a ser instructiva. Hubo un momento durante el diseño en el que tuve que decidir realizar una determinada operación común con eventos de C#, con delegados, con métodos virtuales o con interfaces. Para tomar una decisión, necesitaba entender el elemento de compensación del rendimiento entre estas opciones. En pocos minutos había escrito la microprueba comparativa para medir el rendimiento de cada una de las alternativas. La [Figura 2](#) muestra las filas pertinentes y se puede ver que no hay una diferencia substancial entre las alternativas. Este conocimiento me permite elegir la alternativa más natural, sabiendo que no sacrifico el rendimiento.

<b>Operación medida</b>	<b>Mediana</b>	<b>Promedio</b>	<b>Desvest</b>	<b>Mín.</b>	<b>Máx.</b>	<b>Ejemplos</b>
MethodCalls: aClass.Interface() [count=1000 scale=10.0]	1.651	1.660	0.084	1.579	1.814	10
MethodCalls: aClass.VirtualMethod() [count=1000 scale=10.0]	1.228	1.175	0.077	1.083	1.277	10
Delegados: aInstanceDelegate ( ) [count=1000 scale=10.0]	1.151	1.159	0.085	1.075	1.314	10
Eventos: Fire Events [count=1000 scale=10.0]	1.228	1.195	0.070	1.088	1.291	10

### Validación de los resultados de rendimiento

La aplicación MeasureIt facilita mucho la recopilación de datos para una amplia gama de bancos de pruebas. Por desgracia, MeasureIt no aborda un aspecto importante del uso de los datos del banco de pruebas: la validación. Es muy frecuente medir algo distinto de lo que en realidad se quería medir. El resultado son datos erróneos y totalmente inútiles. Cuando hablamos de datos de rendimiento, si algo parece demasiado bueno (o malo) como para ser cierto,

seguramente lo sea. Es imprescindible validar los datos utilizados para tomar decisiones claves para el diseño.

### Validación con depurador de datos obtenidos de una microprueba comparativa

¿Qué significa validar los resultados de rendimiento? Significa recopilar información distinta que también predice el resultado de rendimiento y comprobar si las dos metodologías coinciden. Para micropruebas comparativas muy pequeñas, una excelente comprobación consiste en inspeccionar las instrucciones de máquina y hacer una estimación basada en el número de instrucciones ejecutadas. En un depurador como Visual Studio, debería ser tan fácil como configurar un punto de interrupción en su código de banco de pruebas y conmutar a la ventana de desensamblado (Depuración -> Windows -> Desensamblado). Por desgracia, las opciones predeterminadas para Visual Studio están diseñadas para simplificar la depuración, no para realizar investigaciones sobre el rendimiento, así que es necesario cambiar dos opciones para realizar esta tarea.

Primero, vaya a Herramientas | Opciones... | Depuración | General y desactive la casilla para suprimir la optimización JIT. Esta casilla está activada de forma predeterminada, lo que implica que incluso al depurar un código que debe ser optimizado, el depurador le ordena al tiempo de ejecución que no lo haga. El depurador hace esto para que las optimizaciones no interfieran con la inspección de variables locales, pero también significa que no se atiende al código en ejecución. Siempre desactivo esta opción porque creo que los depuradores deben esforzarse sólo en realizar la inspección y no en cambiar el programa que se está depurando. Tenga en cuenta que desactivar esta opción no tiene efecto alguno sobre el código que se ha compilado para depurar, ya que de todos modos el tiempo de ejecución no habría optimizado ese código.

A continuación, desactive la casilla Habilitar Sólo mi código en Herramientas | Opciones | Depuración | General. La característica Sólo mi código instruye al depurador para que no muestre el código que no se ha escrito. Generalmente, esta característica elimina los marcos de llamada que a menudo no son de interés para el desarrollador de aplicaciones. Sin embargo, esta característica asume que ningún código que se esté optimizado puede ser suyo (asume que su código se compila usando la configuración de depuración o que se ha activado la supresión de las optimizaciones JIT). Si se permiten optimizaciones JIT pero no desactiva Sólo mi código, encontrará que no llega a ningún punto de interrupción porque el depurador no cree que el código sea suyo.

Una vez haya desactivado estas opciones, permanecerán desactivadas para TODOS los proyectos. Generalmente esto funciona, pero a costa de desactivar la característica Sólo mi código. Tendrá que activar y desactivar esa característica mientras alterna entre depuración y evaluación de rendimiento.

Como ejemplo del uso de un depurador para validar los resultados de rendimiento, puede estudiar la anomalía en los datos mostrados en el extracto de la [Figura 3](#). Estos datos muestran que la llamada a un método de la interfaz de una estructura de C# es mucho más rápida que una llamada a un método estático. Parece extraño, dado que lo normal sería que una llamada a método estático fuese el tipo más eficaz de llamada. Para investigarlo, hay que establecer un punto de interrupción en este banco de pruebas y ejecutar la aplicación. Vaya a la ventana de desensamblado (Depuración -> Windows -> Desensamblado) y verá que todo el banco de pruebas consiste sólo en el código siguiente:

Operación medida	Mediana	Promedio	Desves	Mín.	Máx.	Ejemplos
MethodCalls: EmptyStaticFunction () [count=1000 scale=10.0]	1.000	0.964	0.102	0.857	1.196	10
MethodCalls: aStructWithInterface.Interface e() [count=1000 scale=10.0]	0.031	0.029	0.012	0.021	0.039	10

```
aStructWithInterface.InterfaceMethod();
00000000 ret
```

Esto nos demuestra que el banco de pruebas (10 llamadas a un método de interfaz) se ha excluido. La instrucción RET es en realidad el fin del cuerpo delegado que define todo el banco de pruebas. Lo normal es que no hacer nada sea más rápido que realizar llamadas de método, así que esto muestra la razón de la anomalía.

El único misterio es saber por qué los métodos estáticos tampoco están incluidos. Esto se debe a que suprimí la inclusión para los métodos estáticos mediante el atributo `MethodImplOptions.NoInlining`. Me "olvidé" intencionadamente de poner esto en el banco de pruebas de la llamada de interfaz para demostrar que el compilador JIT puede hacer que ciertas llamadas de interfaz sean tan eficientes como las llamadas no virtuales (hay un comentario sobre esto encima del banco de pruebas).

## Conclusión

Insisto en que es muy fácil medir algo que en realidad no se quiere medir, sobre todo cuando lo que se miden son cosas pequeñas que están sujetas a optimizaciones de compilador JIT. No es difícil medir accidentalmente código no optimizado o medir el costo de compilación JIT de un método en vez del método en sí mismo. El comando `MeasureIt/usersGuide` mostrará una guía de usuario que trata muchas de las dificultades con las que se puede encontrar al crear bancos de

pruebas. Es muy recomendable que lea esta información cuando esté dispuesto a escribir sus propios bancos de pruebas.

Lo que quiero destacar es el concepto de validación. Si no se pueden explicar los datos, éstos no deben emplearse para tomar decisiones sobre el diseño. Si los datos son extraños, lo mejor será reunir más datos, depurar los bancos de pruebas o buscar la colaboración de gente más experta hasta que usted pueda explicar sus datos. Debería ser muy suspicaz con los datos que resulten inexplicables y no debería usarlos nunca para tomar una decisión importante.

Este debate se centra en los principios de escritura de aplicaciones de alto rendimiento. Como cualquier otro atributo de software, un buen rendimiento debería incorporarse en el diseño del producto desde el principio. Para hacer esto, se necesitan medidas que cuantifiquen los elementos de compensación de la toma de varias decisiones. Esto implica realizar experimentos de rendimiento. MeasureIt facilita la rápida generación de micropruebas comparativas de buena calidad, que deberían ser un elemento imprescindible en el proceso de diseño. Además, MeasureIt es útil desde el momento en que se adquiere el producto, ya que viene con un conjunto de bancos de pruebas que cubren la mayor parte de las operaciones básicas de .NET Framework.

También se pueden agregar fácilmente bancos de pruebas propios para las partes de .NET Framework que más le interesen. Con estos datos puede formar un modelo de costos de aplicación y hacer así suposiciones razonables (aproximadas) acerca del rendimiento de distintas alternativas de diseño incluso antes de haber escrito el código de aplicación.

Se podrían decir muchas más cosas acerca del rendimiento de las aplicaciones en .NET. Existen problemas potenciales asociados a la creación de micropruebas comparativas, así que por favor, lea la guía de usuario de MeasureIt antes de escribir nada. No he hablado de las situaciones en las que la E/S de disco, la memoria, o la contención de bloqueo representan un obstáculo importante. Ni tampoco de cómo usar varias herramientas de generación de perfiles para validar y supervisar el estado del rendimiento de su aplicación tras el proceso de diseño.

Hay mucho que aprender y, a menudo, el volumen de información desanima a los desarrolladores para realizar pruebas. Sin embargo, debido a que la mayoría del rendimiento se pierde durante el proceso de diseño de una aplicación, se debería dedicar tiempo a intentar mejorarlo en esta etapa inicial. Espero que esta columna le anime a hacer del rendimiento una parte explícita del diseño en su próximo proyecto de software .NET.

Envíe sus preguntas y comentarios a [clrinout@microsoft.com](mailto:clrinout@microsoft.com).

---

**Vance Morrison** es arquitecto compilador de CLR (lenguaje común en tiempo de ejecución) de Microsoft y ha participado en el diseño de .NET desde sus inicios. Dirigió el diseño del Lenguaje intermedio de .NET (IL, Intermediate Language) y fue jefe del equipo compilador de Just-in-Time (JIT).

---

© 2007 Microsoft Corporation and CMP Media, LLC. Reservados todos los derechos; queda prohibida la reproducción parcial o total sin previa autorización.