Arquitetura da Máquina Virtual .NET

Marcelo Coutinho Cordeiro
RA002092
Instituto de Computação
Unicamp
ra002092@ic.unicamp.br

RESUMO

Esse texto apresenta uma visão geral da arquitetura da máquina virtual do *framework* .NET, conhecida como CLR (*Common Language Runtime*). Serão apresentados aspectos arquiteturais e estruturais da CLR, portanto trataremos de tópicos como compilação *just-int-time*, verificações de segurança, linguagens intermediárias utilizadas, seu seu sistema de execução virtual e até mesmo o mecanismo de tratamento de exceções.

Termos Gerais

Arquitetura de computadores, compiladores, linguagens de programação.

Palavras-chave

Máquina virtual, CLR, .NET, Arquitetura, Compiladores, JIT.

1. Introdução

Esse trabalho apresenta uma descrição da arquitetura da máquina virtual, *Common Language Runtime* (CLR), utilizada pelo *framework* .NET da Microsoft. A máquina em questão gerencia a execução do código fonte depois de ser compilado para uma linguagem intermediária como *Microsoft Intermediate Language* (MSIL), OptIL, ou código nativo de máquina. A principal propriedade da CLR é a habilidade de prover *software isolation* aos programas executados dentro um espaço de endereçamento único. Dentre os principais serviços oferecidos pela CLR podemos destacar:

- Gerenciamento de código.
- Isolação da memória utilizada pelo programa.
- Verificação da segurança de tipo da MSIL.
- Conversão da MSIL para código nativo.
- Carregar e executar de *managed code* (MSIL ou nativo)
- Gerenciamento de memória de objetos gerenciados.
- Inserção e execução de checagens de segurança.
- Manipulação de exceções.
- Interoperabilidade entre objetos .NET e objetos COM.

Um das funções mais importantes da máquina virtual CLR é a conversão *on-th-fly* de MSIL (ou OptIL) para código nativo[5]. Compiladores geram código em MSIL, ou OptIL a partir do código fonte. E compiladores JIT convertem esse código em MSIL para código nativo para arquitetura da máquina em questão. Como a conversão ocorre na máquina alvo, o código nativo gerado pode tirar vantagem otimizações específicas de hardware[8].

2. Arquitetura

Essa seção apresenta os principais componentes da arquitetura da CLR.

2.1 MSIL e OptIL

MSIL é um conjunto de instruções baseado em pilha desenvolvido para ser facilmente gerada a partir de um código fonte por compiladores e outras ferramentas. Vários tipos de instruções são providas, incluindo instruções aritméticas e operações lógicas, controle de fluxo, acesso direto à memória, manipulação de exceções, e invocação de métodos. Existe também um conjunto de instruções do MSIL para implementação de construções orientadas a objetos, como chamadas de métodos virtuais, acesso a campos, acesso a *arrays*, e alocação e inicialização de objetos.

O conjunto de instruções do MSIL pode ser diretamente interpretado simplesmente percorrendo os tipos de dados na pilha e emulando as instruções MSIL[8]. Ele também pode ser convertido de forma eficiente em código nativo. A forma como foi projetada a MSIL permite que se produza código nativo otimizado a um custo razoável. Seguindo um simples conjunto de regras é possível gerar programas MSIL que não sejam somente *typesafe*, mas que podem ser facilmente adaptados para que sejam.

OptIL é um subconjunto do MSIL que pode ser gerado por *front ends* de compiladores otimizados. OptIL possui anotações embutidas, que são instruções MSIL que fornecem fluxo de controle e informação sobre alocação de registradores. Uma vez que OptIL é um subconjunto de MSIL, qualquer componente que pode executar ou mesmo analisar MSIL, pode também analisar ou executar OptIL, ignorando as anotações embutidas se necessário. O compilador OptJIT, entretanto, usa essas anotações para gerar rapidamente código nativo otimizado. A corretude desse código depende das anotações, portanto elas estão sujeitas a verificação. OptIL é bastante útil em situações onde temos limitações de tempo e memória para conversão para código nativo.

2.2 JIT

A CLR possui três compiladores *just-in-time* JIT para conversão de MSIL para código nativo: EconoJIT, JIT, e OptJIT[4]. Cada uma delas foi desenvolvida para atender objetivos específicos com respeito a desempenho e utilização de recursos. As características de desempenho são apresentadas na Tabela 1. EconoJIT desempenha a mesma tarefa que JIT, porém com menos recursos, com isso a qualidade do código gerado não é tão alta. Tanto a JIT quanto a EconoJIT aceitam como entrada MSIL, e também OptIL, já que essa é um subconjunto da primeira. Agora, a OptJIT só aceita como entrada a linguagem OptIL.

Tabela 1. Características de Desempenho dos Compiladores JIT do Framework .NET

Compilador JIT	Entrada	JIT Compiler Overhead	Velocidade	Qualidade da Saída
EconoJIT	MSIL	Muito pequena	Muito rápida	Baixa
JIT	MSIL	Média Grande	Moderada	Alta
OptJIT	Só OptIL	Pequena	Rápida	Alta

2.3 Carregamento de Classe

O carregador de classes carrega a implementação de uma classe em MSIL, OptIL , ou código nativo para a memória, checa a consistência esperada por outras classes já carregadas, e prepara para a execução. Para realizar esta tarefa, o carregador de classes verifica se a informação fixada é conhecida[7]. Além disso, o carregador determina se referências feitas pelo tipo carregado estão disponíveis em tempo de execução e se são consistentes.

CLR permite apenas um carregador de classes, o dela própria. O *framework* .NET não dá suporte a carregador de classes escritos pelo usuário[1].

2.4 Verificação

Programas t*ypesafe* referenciam somente a memória que foi alocada para o seu uso, e eles acessam objetos somente através de suas interfaces públicas[3]. Essas duas restrições permitem que objetos compartilhem de forma segura o mesmo espaço de endereçamento, além de garantir que as checagens de segurança oferecidas pelas interfaces de objetos não sejam negligenciadas.

O mecanismo de segurança da CLR pode proteger efetivamente o código de acessos não autorizados somente se existir uma forma de verificar se aquele código é *typesafe*. Para isso, a CLR usa a informação contidas nas assinaturas dos tipos para ajudar a determinar se o código MSIL é *typesafe*. Um programa só é declarado como verificado se este for *typesafe*.

Usado em conjunto com metadados de tipo e MSIL, essa verificação garante *type safety* dos programas escritos em MSIL. O *framework* .NET requer que o código seja verificado antes da sua execução, exceto um tipo específico de verificação de segurança que determina se o código é completamente confiável[6].

2.5 Checagens de Segurança

A CLR está envolvida em diversos aspectos do mecanismo de segurança do *framework* .NET. Em adição ao processo de verificação necessário para segurança do acesso ao código, a CLR suporta checagem declarativa e imperativa.

Checagem de segurança declarativa é acionada automaticamente quando um método é invocado[3]. As permissões necessárias para se acessar o método são armazenadas em um componente de metadados. Em tempo de execução, chamadas a métodos que exigem permissões específicas são interceptadas para determinar se os invocadores possuem as permissões necessárias nesse caso. As vezes é necessário caminhar na pilha para se determinar se o invocador possui tais permissões[8].

Checagem de segurança imperativa ocorre quando funções de segurança, como as que verificam um código de permissão de acesso, são invocadas a partir de um código protegido.

3. Sistema de Execução Virtual

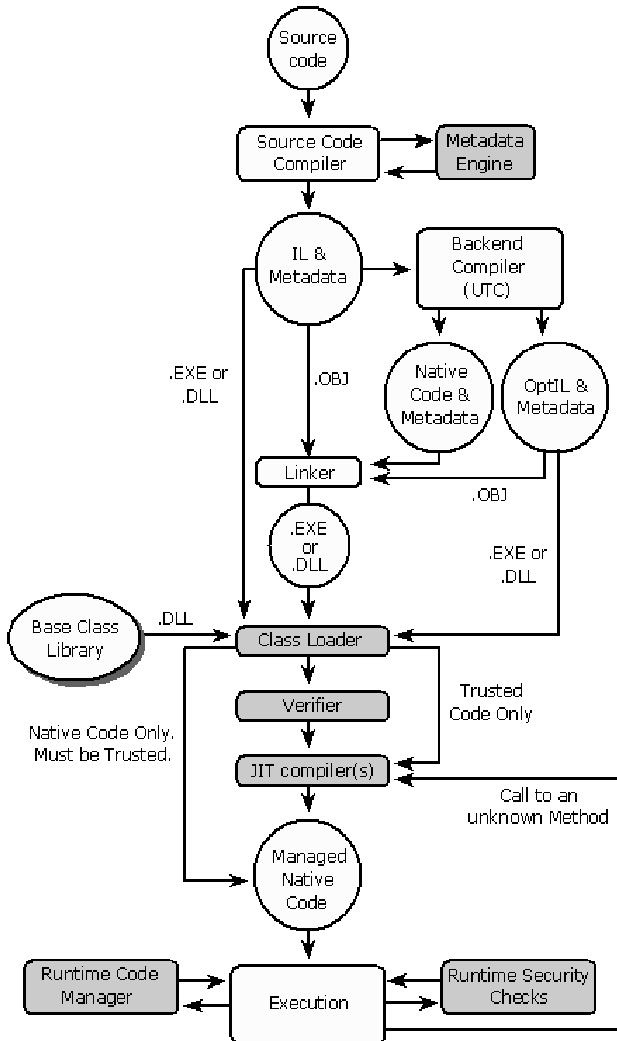
Por ser responsável pelo carregamento de classes, verificação, compilação *just-in-time*, e gerenciamento de código a máquina virtual .NET cria um ambiente para execução de código chamado de Sistema de Execução Virtual. A figura 1 apresenta os principais elementos da CLR em cinza, e indica com setas os possíveis caminhos que podem ser tomados pelo ambiente de execução.

Em geral, o código fonte é compilado e transformado em MSIL, o código em MSIL é carregado e compilado para código nativo em *on-the-fly* usando-se um dos compiladores JIT supracitados, e executado.

O mecanismo de metadados da CLR que o compilador de código fonte insira metadados no código MSIL, ou OptIL gerado. Durante o carregamento e execução, esses metadados fornecem a informação necessária para registrar, depurar, gerenciar memória. Podemos observar no diagrama que as classes da biblioteca do *framework* .NET podem ser carregadas pelo carregador de classes juntamente com o código MSIL.

Outro possível caminho de execução que pode ser tomado envolve pré compilação para código nativo usando o backend de um compilador. Essa opção pode ser escolhida se o código compilado em tempo de execução for ineficiente considerando-se critérios de desempenho. Como podemos ver no diagrama, o código nativo pré compilado desvia a parte de verificação e a compilação JIT.

Figura 2. Arquitetura da CLR



4. Aspectos Estruturais

Essa seção apresentará aspectos estruturais da CLR, que ajudam a compreender melhor o seu funcionamento.

4.1 Estrutura do Executável

A CLR confia nas seguintes informações sobre cada método definido em um executável portável, que é o formato usado para os executáveis (EXE) e as DLLs, que são listadas abaixo:

- As instruções que compõem o corpo do método, incluindo todos manipuladores de exceção.
- A assinatura do método, que especifica o tipo de retorno, o número e a ordem dos parametros, e o tipo de cada um dos argumentos.
- O array de tratamento de exceções.
- O tamanho da pilha de execução que o método necessita, que é além disso é um dos fatores verificados pelo verificador.
- O tamanho dos arrays locais que o método precisa.

A partir das restrições colocadas no código, o carregador de classes da CLR pode evitar que código não portável seja executado em uma arquitetura que não o suporta.

4.2 Fluxo de Controle

O conjunto de instruções MSIL fornece um rico grupo de instruções para se alterar o fluxo de controle normal de uma instrução MSIL para a seguinte, como por exemplo branches condicionais e incondicionais, chamada de método, e *return*.

Enquanto CLR suporta transferências de controle arbitrárias dentro de um método, existem várias restrições que devem ser observadas, e também testadas pelo verificador, dentre as quais podemos destacar as seguintes:

- Não é permitido a transferência para um região de cláusula catch , ou finally, a menos que seja feita pelo mecanismo de tratamento de exceções.
- Transferir o controle para fora de uma região protegida só é permitido através de uma instrução de exceção

4.3 Invocação de Métodos

Um importante objetivo do projeto da CLR é abstrair o leiaute nativo dos métodos, incluindo a convenção de chamadas. Ou seja, instruções emitidas pelo gerador de código MSIL contem informação suficiente para diferente implementações da CLR para usar a convenção de chamada nativa[2].

MSIL possui três instruções que podem ser usadas para invocar métodos. A instrução **call** foi projetada para ser usada quando o endereço de destino é fixado em tempo de ligação, e nesse caso uma referência ao método é colocada diretamente na instrução. É comparável a uma chamada a uma função estática em C, portanto pode ser usada para invocar métodos estáticos. Outra instrução

usada nesse sentido é a **calli**, que é usada quando o endereço de destino é calculado em tempo de execução, aqui um apontador do método é passado a pilha e a instrução contem apenas a descrição do local da chamada. Finalmente, temos a instrução **callvirt**, que determina a classe de um objeto, que é conhecida apenas em tempo de execução, para determinar qual método deve ser chamado. A instrução inclui uma referência ao método, mas o método em particular não é computado até que a chamada ocorra.

5. Tratamento de Exceções

A CLR suporta um modelo de tratamento de exceções baseado na idéia de objetos de exceção e de blocos protegidos de código. Quando uma exceção ocorre, um objeto é criado para representá-la. Todos os objetos de exceção são instancias de alguma classe. Usuários podem criar suas próprias classes de exceção, que são todas subclasses de **System.Exception**.

A CLR possui quatro tipos de manipuladores de exceção para blocos protegidos. Um bloco protegido pode ter exatamente um manipulador de exceções associado a ele:

- Um **finally handler** que deve ser executado quando o bloco sai, sem considerar se isso ocorreu devido ao fluxo de controle normal ou por uma exceção não tratada.
- Um **fault handler** que deve ser executado se ocorreu uma exceção, mas não se completou o fluxo de controle normal.
- Um **type-filtered handler** que manipula qualquer exceção de uma classe especifica e suas subclasses.
- Um **user-filtered handler** que executa um conjunto de instruções MSIL especificadas pelo usuário para determinar se exceção deve ser ignorada, tratada pelo manipulador associado, ou passada para o próximo bloco protegido.

A CLR possui uma classe especial de instruções chamada ExecutionEngineException que pode ser lançada por qualquer instrução e indica um inconsistência inesperada na CLR. O código passado através do verificador de código nunca deve lançar esse tipo de instrução. Entretanto, código não verificado pode causar esse erro se for corrompido ou inconsistente de alguma maneira.

Note que, devido a existência do verificador, não ocorrem exceções para coisas do tipo token não encontrado, por exemplo. O verificador detecta essa inconsistência antes da instrução ser executada, e se o código não for verificado, esse tipo de inconsistência gera uma exceção do tipo ExecutionEngineException.

Cada método em um executável possui um vetor de tratamento de exceções associado. Cada entrada desse vetor descreve um bloco protegido, e seu manipulador. Quando ocorre uma exceção a CLR busca no vetor pelo primeiro bloco protegido que protege a região, e se essa busca não retorna resultado algum no método atual, ela busca no método que invocou o método atual. Se a CLR não acha nada percorrendo o caminho de execução ela aborta o programa.

Alguns aspectos importante sobre o tratamento de exceções são:

- A ordem das clausulas de exceção na tabela de tratamento de exceções é importante, pois se os manipuladores estiverem aninhados, o mais profundo manipulador aninhado deve ocorrer antes.
- Tratadores de exceção podem acessar variáveis locais e a região de memória local da rotina que captura a exceção, porém um resultado intermediário que estava na pilha de avaliação quando a exceção é lançada é perdido.
- Um objeto do tipo exceção descrevendo a exceção é criado automaticamente pela CLR e colocado na pilha de avaliação.

6. Considerações Finais

Ao longo deste documento foi possível conhecer aspectos estruturais da arquitetura da máquina virtual utilizada pelo *framework* .NET, da Microsoft, e algumas de suas peculiaridades. Apresentamos as diferenças entre as linguagens intermediárias utilizadas, assim como as implicações do uso de cada uma delas em etapas como a da compilação *just-in-time*. Podemos também observar como funciona o mecanismo de tratamento de exceções e o de invocação de métodos. Com isso foi possível aumentar a familiaridade com essa ferramenta , além de permitir explorar ainda mais suas potencialidades e se proteger melhor de suas eventuais armadilhas.

7. Bibliografia

[1] Dachuan Yu, Andrew Kennedy, Don Syme. Formalization of generics for the .NET common language runtime. *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2004, Volume 39 Issue 1*.

[2] Jennifer Hamilton. Technical correspondence: Language integration in the common language runtime. ACM SIGPLAN Notices, Volume 38 Issue 2, 2003

[3] Rean Griffith, Gail Kaiser. Manipulating managed execution runtimes to support self-healing systems. ACM SIGSOFT Software Engineering Notes, Volume 30 Issue 4, 2005

[4] Andrew D. Gordon, Don Syme. Typing a multi-language intermediate code. Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages POPL '01, Volume 36 Issue 3

[5] GotDotNet. About the Common Language Runtime (CLR). .NET framework community website.

[6] MSDN Libray. The Common Language Runtime (CLR).

[7] MSDN Library. Common Language Runtime Overview (.NET Framework Developer's Guide)

[8] Microsoft Corporation. .NET Framework Common Language Runtime Architecture.