2
Interface entre o CLR e o Interpretador Lua

Este capítulo descreve a abordagem de integração entre a linguagem Lua e o Common Language Runtime, através da implementação de uma ponte, a biblioteca LuaInterface, entre o interpretador Lua 5.0 e o CLR.

LuaInterface usa a API do interpretador Lua e as APIs PInvoke (para interface com código nativo) e de reflexão do CLR. LuaInterface oferece as capacidades de um consumidor completo da Common Language Specification, ou seja, os scripts Lua podem instanciar e usar objetos do CLR, além de criar *delegates* a partir de funções Lua. A sintaxe para usar os objetos do CLR é a mesma sintaxe para usar objetos Lua. LuaInterface também fornece capacidade limitada de criar novas classes para o CTS.

Para as aplicações do CLR, LuaInterface oferece a possibilidade de executar código Lua, ler e escrever variáveis globais do interpretador, chamar funções Lua e registrar como funções os métodos de objetos CLR.

As seções seguintes descrevem em detalhes os recursos que LuaInterface disponibiliza e a sua implementação.

2.1
Recursos da Interface

LuaInterface encapsula a API da linguagem Lua em uma classe chamada `Lua`, que oferece métodos para executar código Lua, ler e escrever variáveis globais, e registrar métodos do CLR como funções Lua. Classes auxiliares oferecem métodos para acesso aos campos de tabelas Lua e para chamada a funções Lua.

Instanciar a classe `Lua` inicia um interpretador Lua. Várias instâncias podem ser criadas, e são independentes entre si. Os métodos `DoFile` e `DoString` da classe `Lua` executam respectivamente um arquivo fonte e um trecho de código Lua. Aplicações podem ler ou escrever variáveis globais indexando uma instância da classe `Lua` pelo nome da variável. O código C# seguinte demonstra o uso da classe `Lua`:

```
// Iniciar uma instância do interpretador Lua
Lua lua = new Lua();
// Executar trechos de código Lua
lua.DoString("num = 2"); // Cria variável global 'num'
lua.DoString("str = 'uma string'");
// Lê variáveis globais 'num' e 'str'
double num = (double)lua["num"];
string str = (string)lua["str"];
// Escreve na variável global 'str'
lua["str"] = "outra string";
```

2.1.1 Conversões de Tipos

O CLR é estaticamente tipado; logo, sempre que um valor Lua é passado para o CLR, LuaInterface converte o valor Lua para o tipo esperado pelo CLR, se possível; senão LuaInterface gera uma exceção. Se o CLR espera um valor de tipo `object`, LuaInterface usa o seguinte mapeamento: `nil` para `null`, números para `System.Double`, strings para `System.String` e booleanos para `System.Boolean`.

LuaInterface converte tabelas Lua para instâncias de `LuaTable`. Indexar uma instância desta classe acessa o campo da tabela correspondente à chave usada. Funções são convertidas para instâncias de `LuaFunction`. Esta classe define um método `call` que chama a função Lua correspondente e retorna um vetor com os valores de retorno da função.

Se o CLR espera um tipo numérico então LuaInterface converte números para o tipo esperado, arredondando o número, se preciso. LuaInterface também converte strings numéricas para números do CLR. Da mesma forma, LuaInterface converte números em strings, caso o CLR espere uma instância de `System.String`. Se o CLR espera um valor `System.Boolean` então LuaInterface converte qualquer valor Lua, exceto `false` e `nil`, para `true`.

Se o CLR espera um *delegate*, e o valor Lua é uma função, então LuaInterface converte a função para um delegate do tipo esperado. LuaInterface cria um objeto delegate que chama a função Lua, com os argumentos para o delegate se tornando argumentos para a função.

Sempre que um valor é passado do CLR para Lua, LuaInterface converte o valor CLR para o tipo Lua mais próximo: `null` para `nil`, valores numéricos (`System.Int32`, `System.Double`, etc.) para números, instâncias

de `System.String` para strings, instâncias de `System.Boolean` para booleanos. LuaInterface converte instâncias de `LuaTable` e `LuaFunction` para as tabelas e funções correspondentes, respectivamente.

LuaInterface converte todos os outros valores do CLR para *proxies* para o próprio valor. A Seção 2.1.4 cobre esta conversão.

2.1.2 Métodos, Construtores e Sobrecarga

O CLR permite sobrecarga de métodos. Sempre que um script Lua chama um método sobrecarregado, LuaInterface tem que escolher qual método deve chamar. LuaInterface passa por cada método, primeiro verificando se o número de parâmetros é igual ao número de argumentos da chamada. Se é igual então LuaInterface checa se cada argumento pode ser convertido para o tipo do seu parâmetro correspondente, de acordo com as regras descritas na Seção 2.1.1. LuaInterface gera um erro se não acha um método adequado para chamar. Construtores também podem ser sobrecarregados (e geralmente são). LuaInterface usa este mesmo procedimento para escolher qual construtor usar para instanciar um objeto.

Uma consequência deste procedimento de seleção é que podem haver métodos sobrecarregados que não podem ser chamados diretamente. Se o primeiro método recebe um parâmetro `System.Double`, e o segundo um parâmetro `System.Int32`, o segundo método nunca é escolhido; qualquer valor que pode ser convertido para `Int32` também pode ser convertido para `Double`. A Seção 2.1.6 mostra uma maneira de contornar esta situação.

2.1.3 Importando Tipos do CTS e Instanciando Objetos

Para instanciar novos objetos, os scripts Lua precisam de referências para os seus tipos. LuaInterface fornece duas funções para obter estas referências. A função `load_assembly` carrega uma assembly (biblioteca) do CLR e disponibiliza os seus tipos para importação pela função `import_type`. Esta função vasculha as assemblies carregadas em busca do tipo pedido e retorna uma referência para ele. O trecho de código Lua a seguir demonstra o uso das duas funções:

```
load_assembly("System.Windows.Forms")
load_assembly("System.Drawing")
Form = import_type("System.Windows.Forms.Form")
```

```
Button = import_type("System.Windows.Forms.Button")
Point = import_type("System.Drawing.Point")
StartPosition = import_type("System.Windows.Forms." ..
  "FormStartPosition")
```

A função `import_type` não funciona apenas para classes: scripts também podem obter referências para estruturas (ex. `Point`) e enumerações (ex. `FormStartPosition`) através dela.

Para instanciar um objeto do CLR, um script chama o seu respectivo tipo, como uma função. O exemplo seguinte estende o anterior para mostrar como objetos são instanciados:

```
form1 = Form()
botao1 = Button()
botao2 = Button()
posicao = Point(10,10)
posicao_inicio = StartPosition.CenterScreen
```

Se um tipo tem sobrecarga de seus construtores então LuaInterface escolhe qual construtor chamar usando o procedimento apresentado na Seção 2.1.2.

Uma vez importado o script pode usar um tipo para chamar seus métodos estáticos. A sintaxe é a mesma da chamada de métodos em objetos Lua; por exemplo, a expressão `Form:GetAutoScaleSize(arg)` chama o método `GetAutoScaleSize` do tipo `Form`. LuaInterface procura dinamicamente no tipo o método estático desejado.

Os scripts também podem usar um tipo para ler e escrever seus campos e propriedades estáticos. Por exemplo, o comando `var=Form.ActiveForm` atribui à variável `var` o valor da propriedade `ActiveForm` do tipo `Form`.

2.1.4 Acesso a Objetos do CTS

Alguns tipos do CTS têm um mapeamento direto para tipos básicos de Lua. Estes tipos são `null`, tipos numéricos, `System.Boolean`, `System.String` e os tipos `LuaTable` e `LuaFunction` apresentados na Seção 2.1.1.

LuaInterface mapeia todos os outros tipos do CTS para proxies para o respectivo valor. No exemplo de instanciação de objetos da Seção 2.1.3, por exemplo, os valores atribuídos às variáveis `form1`, `form2`, `botao1`, `botao2` e `posicao` são todos proxies para os objetos CLR reais.

Os scripts podem usar um proxy para um objeto CLR do mesmo jeito que usam qualquer outro objeto Lua: podem ler campos, atribuir valores a campos, ler propriedades, atribuir a propriedades, e chamar métodos. O próximo exemplo complementa os dois anteriores, mostrando como propriedades e métodos são usados:

```
botao1.Text = "Ok"
botao2.Text = "Cancelar"
botao1.Location = posicao
botao2.Location = Point(botao1.Left,
  botao1.Height + botao1.Top + 10)
form1.Controls:Add(botao1)
form1.Controls:Add(botao2)
form1.StartPosition = posicao_inicio
form1:ShowDialog()
```

No exemplo anterior, o comando `botao1.Texto="Ok"` atribui a string `"Ok"` à propriedade `Text` do objeto `botao1`. O comando `form1.Controls:Add(botao1)` lê a propriedade `Controls` do objeto `form1`, então chama o método `Add` do valor dessa propriedade, passando o objeto `botao1` como argumento para a chamada. Os três exemplos anteriores combinados, quando executados, exibem uma janela com dois botões no centro da tela.

LuaInterface também provê um atalho para indexar vetores, tanto para sua leitura quanto para sua escrita. O atalho é indexar a referência para o vetor com um número; por exemplo, `arr[3]`. Para matrizes, os scripts devem usar os métodos da classe `System.Array`.

LuaInterface passa para Lua, como um erro, qualquer exceção que ocorra durante a execução de um método do CLR. O objeto da exceção é a mensagem de erro (as "mensagens de erro" de Lua não precisam ser strings). O script podem usar os mecanismos de Lua para capturar e tratar estes erros.

2.1.5 Tratamento de Eventos

Eventos são um recurso do CLR para implementar *callbacks*. Para oferecer um evento, um tipo declara dois métodos: um método para adicionar um tratador para o evento e outro para remover um tratador. Os meta-dados para o tipo declaram o nome do evento, o tipo dos seus tratadores, e quais

métodos são usados para adicionar e remover tratadores. Uma aplicação pode usar a API de reflexão do CLR para descobrir quais eventos um tipo oferece, e para adicionar e remover tratadores de eventos; a API de reflexão obtém esta informação dos meta-dados.

LuaInterface representa eventos como objetos que definem dois métodos: `Add` e `Remove`. Estes métodos respectivamente adicionam e removem um tratador para o evento. O método `Add` recebe uma função Lua, converte a função para um delegate do tipo que o evento espera, e acrescenta o delegate como um tratador para aquele evento. O método `Remove`, em troca, recebe um delegate registrado como um tratador e o remove.

Por exemplo, se um objeto `obj` define um evento `Ev`, a expressão `obj.Ev` retorna um objeto que representa o evento `Ev`. Se `func` é uma função, o comando `obj.Ev:Add(func)` a registra como tratador do evento `Ev`. Toda vez que o evento `Ev` dispara o CLR chama o delegate e este em seguida chama a função `func`.

O trecho a seguir estende o exemplo da janela com dois botões, acrescentando tratadores de evento para os botões:

```
function trata_mouseup(sender,args)
  print(sender:ToString() .. " MouseUp!")
end
botao1.MouseUp:Add(trata_mouseup)
botao2.Click:Add(os.exit)
```

Neste exemplo, o comando `botao1.MouseUp:Add(trata_mouseup)` registra a função `trata_mouseup` como um tratador para o evento `MouseUp` do objeto `botao1`. Esta função escreve uma mensagem no console. O CLR passa os parâmetros `sender` e `args`; eles são, respectivamente, o objeto que disparou o evento e os dados específicos para aquele evento. O comando `botao2.Click:Add(os.exit)` registra a função `os.exit`, da biblioteca padrão de Lua, como tratador para evento `Click` do objeto `botao2`. Esta função encerra o programa. Ela não possui parâmetros, mas isto não é problema: o interpretador Lua descartará os dois argumentos passados à função.

2.1.6 Recursos Adicionais de um Consumidor CLS

Os recursos descritos na Seção 2.1 cobrem a maior parte das capacidades de um consumidor completo da CLS. Os próximos parágrafos apresen-

tam os recursos que cobrem o restante das capacidades que um consumidor completo precisa oferecer.

O CLR oferece tanto passagem de parâmetros por valor quanto por referência. Parâmetros passados por referência são de dois tipos: parâmetros *out* são parâmetros apenas para saída (seus valores não podem ser lidos), e os parâmetros *ref* podem ser lidos e terem valores atribuídos a eles.

A linguagem Lua oferece apenas passagem de parâmetros por valor, mas suas funções não estão limitadas a um único valor de retorno; LuaInterface usa este recurso para dar suporte a parâmetros *out* e *ref* nos métodos do CLR. Os valores de retorno de parâmetros *out* e *ref* são retornados após o valor de retorno do método.

Se o script chama um método sobrecarregado então LuaInterface executa a primeira versão com a assinatura compatível com os argumentos passados, logo alguns métodos de um objeto podem nunca ser escolhidos (como discutido na Seção 2.1.2). Para chamar esses métodos, LuaInterface oferece a função `get_method_bysig`. Ela recebe um objeto, o nome do método e referências para os tipos de sua assinatura. Chamar `get_method_bysig` retorna uma função que, quando chamada, executa o método que corresponde à assinatura passada. O primeiro argumento para a chamada deve ser o seu objeto de destino. Os scripts também podem usar `get_method_bysig` para chamar métodos das classes numéricas e de string do CLR, e para chamar métodos estáticos de tipos do CTS. Construtores também podem ser sobrecarregados, então também existe a função `get_constructor_bysig`.

Um consumidor completo da CLS deve permitir que métodos com nomes inválidos na linguagem sejam chamados. Em Lua as construções `obj:metodo(...)` e `obj["metodo"](obj,...)` são equivalentes, logo para chamar um método com um nome inválido em Lua basta usar a segunda construção. Por exemplo, a expressão `obj:function(...)` não é válida em Lua, pois `function` é uma palavra reservada. O script deve usar a expressão equivalente `obj["function"](obj,...)`, que é válida.

Caso um objeto possua mais de um método com o mesmo nome e assinatura, mas implementando interfaces diferentes, os scripts podem prefixar o nome do método com o nome da interface (a notação `INomeInterface.NomeMetodo` é usada pela API de reflexão do CLR). Por exemplo, se `obj` tem um método chamado `foo`, definido pela interface `IFoo`, deve-se chamar o método com a expressão `obj["IFoo.foo"](obj,...)`.

Finalmente, referências para tipos aninhados também podem ser obtidas facilmente com `import_type`. Basta chamar a função com o nome do tipo aninhado seguindo o nome do tipo que o contém

e um sinal de adição. Novamente, esta notação é usada pela API de reflexão do CLR. Um exemplo do uso desta notação é o comando `import_type("TipoExterno+TipoAninhado")`, que importa o tipo `TipoAninhado` dentro do tipo `TipoExterno`.

2.1.7 Criando Novas Classes com Lua

LuaInterface oferece a função `make_object` para criação de novas classes. A função recebe um objeto Lua e uma interface do Common Type System. LuaInterface automaticamente cria uma nova classe que implementa essa interface. O construtor desta classe recebe um objeto Lua e o guarda. Os métodos da interface delegam sua execução para métodos do objeto Lua armazenado. Depois de criar a classe, LuaInterface a instancia, passando o objeto Lua para o construtor. A função `make_object` retorna o objeto instanciado.

Por exemplo, seja `IExemplo` uma interface definida pelo seguinte código C#:

```
public interface IExemplo {
  float Tarefa(float arg1, float arg2);
}
```

A interface `IExemplo` define um método `Tarefa` que recebe dois parâmetros `float` e retorna outro `float`. Agora, seja `tab` uma tabela Lua definida pelo seguinte código Lua:

```
tab = {
  mult = 2
}

function tab:Tarefa(arg1, arg2)
  return self.mult*arg1*arg2
end
```

A tabela `tab` também define um método `Tarefa`, que recebe dois argumentos, os multiplica, e então multiplica o resultado por um campo de `tab` chamado `mult`, retornando o resultado final.

Definimos então uma classe que usa instâncias de `IExemplo`, com o código C# seguinte:

```
public class TesteExemplo {
  public static void FazTarefa(IExemplo ex, float arg1,
      float arg2) {
    Console.WriteLine(ex.Tarefa(arg1, arg2));
  }
}
```

A classe `TesteExemplo` define um método estático `FazTarefa` que recebe uma instância de `IExemplo` e dois valores `float`, chamando o método `Tarefa` da instância e passando os dois valores `float`. O resultado é exibido no console. Para concluir o exemplo, seja o código Lua seguinte:

```
IExemplo = import_type("IExemplo")
TesteExemplo = import_type("TesteExemplo")

obj = make_object(tab, IExemplo)
TesteExemplo:FazTarefa(obj, 2, 3)
```

Neste código, as primeiras duas linhas importam referências para a interface `IExemplo` e a classe `TesteExemplo` definidas anteriormente. A chamada a `make_object` cria uma instância da interface `IExemplo` que delega seu método `Tarefa` para `tab`. A última linha chama o método `FazTarefa` da classe `TesteExemplo`, passando a instância que criada por `make_object` e os números 2 e 3. Dentro de `FazTarefa`, o método `Tarefa` desta instância é chamado com os números 2 e 3. Na sequência, o método `Tarefa` de `tab` é chamado, novamente com 2 e 3 como argumentos, e o resultado (12) retornado para `FazTarefa`, que o exibe no console.

Sempre que um objeto Lua é passado onde o CLR espera uma interface, LuaInterface automaticamente chama `make_object` com o objeto Lua e o tipo da interface, e passa o objeto retornado por `make_object` no lugar. No exemplo anterior, o último trecho de código Lua poderia ser escrito da seguinte maneira, com o mesmo resultado (exibindo “12” no console):

```
TesteExemplo = import_type("TesteExemplo")
TesteExemplo:FazTarefa(tab, 2, 3)
```

A função `make_object` na verdade pode receber qualquer classe, não apenas interfaces. Ela cria uma nova subclasse da classe. Veja o manual da biblioteca LuaInterface [10] para mais detalhes.

2.2 Implementação da Interface

A biblioteca LuaInterface foi implementada na linguagem C#, com um pequeno trecho (menos de 30 linhas de código) em C. A biblioteca depende do interpretador Lua versão 5.0, assumindo a existência de uma biblioteca de vínculo dinâmico (DLL) chamada `lua-5.0.dll` contendo a implementação da API de Lua, e de outra biblioteca chamada `lualib-5.0.dll` contendo a implementação da API de biblioteca de Lua. A implementação dos recursos de LuaInterface é descrita nas seções seguintes.

2.2.1 A API de Lua

A API de Lua[1] é um conjunto de funções C que um programa pode usar para instanciar e se comunicar com interpretadores Lua. Uma instância do interpretador é criada com a função `lua_open`:

```
lua_State *lua_open(void);
```

O ponteiro retornado aponta para uma estrutura opaca para o programa, servindo apenas para ser passado como argumento para todas as outras funções da API.

Passagem de Valores

A API define uma *pilha virtual* para a passagem de valores de e para o interpretador. Cada elemento da pilha corresponde a um valor Lua. A API define funções para testar o tipo de um valor da pilha, funções para consultar valores da pilha e funções para empilhar valores.

As funções para consultar valores booleanos, numéricos e strings retornam o valor que está na pilha com o tipo C correspondente, respectivamente `int`, `double` e `char*`. Analogamente, as funções que empilham estes tipos de valores recebem o valor com o tipo C correspondente.

Outros valores não podem ser consultados diretamente, mas pode-se obter uma referência para eles e, com essa referência, empilhar de volta o valor.

[1] Esta seção é um resumo da informação contida no Capítulo 3 da Referência da Linguagem Lua [20] que tem relevância direta para a implementação da interface.

Userdata

Userdata é um tipo de Lua usado para armazenar dados arbitrários por aplicações que incorporam o interpretador Lua. O comportamento de um userdata pode ser estendido para permitir operações como indexação, aritmética e comparação.

A API de Lua oferece uma função para criar um userdata, alocando uma área na memória para seus dados, e uma função para obter um ponteiro para a área de memória de um userdata que está na pilha.

Tabelas e Metatables

A API oferece uma função para criar uma nova tabela no topo da pilha, assim como funções para indexar tabelas. As funções de indexação também são usadas para acessar variáveis globais.

Metatables são o mecanismo de Lua para estender o comportamento de objetos (tabelas e userdata). Uma metatable é uma tabela na qual alguns campos com nomes especiais, chamados de meta-métodos, são usados quando certas operações são feitas com o objeto. A API possui funções para atribuir uma metatable a um objeto e obter a metatable atribuída ao objeto, caso exista.

Funções

Para chamar uma função um programa a empilha seguida de seus argumentos, na ordem de chamada, e usa uma das funções da API para chamada de funções. O interpretador põe os valores de retorno na pilha na ordem em que são retornados.

A API também oferece uma função para empilhar ponteiros de função C como funções. A função C empilhada deve ter um único parâmetro `lua_State*`. Os argumentos para ela são passados na pilha, e a função deve empilhar os valores de retorno.

2.2.2 Platform Invoke

Platform Invoke, ou PInvoke, é a funcionalidade do CLR para chamada de código nativo [14, CLI Partition II Section 14.5.2]. PInvoke faz a transição entre o código do CLR e código nativo, fazendo também a conversão entre os tipos do CLR e os nativos. A conversão é dependente de plataforma, mas

toda implementação do CLR deve fazer a conversão bidirecional de pelo menos os tipos abaixo [14, CLI Partition II Section 14.5.4]:

- Tipos inteiros (inclusive `System.Boolean` e `System.Char`) para os seus equivalentes com a mesma largura;
- Enumerações, para o seu tipo de suporte;
- Tipos de ponto flutuante, para seus equivalentes com a mesma precisão;
- O tipo `System.String`, para vetores de caracteres;
- Ponteiros para quaisquer dos tipos acima, para instâncias de `System.IntPtr`, uma estrutura que encapsula ponteiros dentro do CLR.

As implementações do CLR devem fornecer conversões para o código nativo (mas não o contrário) dos seguintes tipos do CLR:

- Vetores de quaisquer dos tipos acima, para vetores do tipo correspondente;
- Delegates, para ponteiros de função.

Em C#, por exemplo, a definição PInvoke da função

```
void lua_pushstring(lua_State *L, const char* s);
```

seria, convertendo os tipos:

```
static extern void lua_pushstring(IntPtr L, string s);
```

O tipo `lua_State*`, nesse caso, é convertido como um ponteiro `void`, e fica totalmente opaco para o código C#.

2.2.3 Encapsulando a API

A tradução dos protótipos da API de Lua, em C, para assinaturas PInvoke, em C#, foi direta, pois os tipos envolvidos possuem um mapeamento automático via PInvoke. A exceção ficou por conta dos parâmetros de tipo ponteiro de função: embora PInvoke converta delegates para ponteiros de função automaticamente, ocorre um conflito entre as convenções de chamada no CLR da plataforma Microsoft .NET.

Os compiladores C usam a convenção CDECL[2] como padrão, logo Lua espera ponteiros de função que seguem esta convenção, mas o compilador Just-In-Time do CLR da Microsoft .NET usa a convenção STDCALL[3] como padrão. A solução foi escrever uma pequena extensão à API em C que encapsula o ponteiro para a função STDCALL dentro de uma função CDECL.

Com a API disponível para programas C#, a implementação da classe `Lua` foi simples. A API já tem funções para conversão de números, strings e valores booleanos de Lua para C e vice-versa. Como PInvoke, por sua vez, converte dos tipos C para os tipos do CTS, a classe `Lua` apenas chama as funções da API quando estes tipos estão envolvidos.

Para tabelas e funções, a API de Lua oferece funções para obter referências (números inteiros) para estes valores. A classe `Lua` guarda estas referências dentro de instâncias de `LuaTable` e `LuaFunction`. Os métodos destas classes obtêm o valor referenciado e chamam a função da API apropriada (ler ou escrever o valor de um campo ou chamar o valor como uma função).

2.2.4 Passando Objetos CLR para o Interpretador Lua

Para passar objetos CLR ao interpretador, LuaInterface usa um esquema similar ao usado para trazer tabelas e funções do interpretador Lua. O objeto primeiro é guardado em um vetor; um novo userdata é alocado, e a posição no vetor onde o objeto foi guardado é armazenada dentro do userdata. LuaInterface então guarda este userdata dentro de uma tabela Lua, usando como chave a posição do objeto CLR no vetor de objetos (o mesmo valor armazenado dentro do userdata). Dessa maneira

[2] A função chamadora limpa a pilha após a chamada.
[3] A função chamada limpa a pilha após a chamada.

o userdata é reaproveitado se o mesmo objeto CLR for passado novamente ao interpretador Lua, ao invés de se criar um novo.

Quando o userdata é coletado pelo interpretador Lua este procura um meta-método chamado `__gc` na metatable do userdata. LuaInterface define, para o userdata correspondente a um objeto CLR, um meta-método `__gc` que retira o objeto do vetor de objetos. A tabela Lua que guarda os userdata correspondentes a objetos CLR os guarda usando referências fracas[4], portanto ela não impede a coleta dos userdata.

2.2.5 Usando Objetos CLR de Lua

Em Lua, uma chamada a um método, como `obj:foo(arg1,arg2)`, equivale a indexar o objeto pelo nome do método e chamar o valor retornado passando o próprio objeto como primeiro argumento, seguido dos outros argumentos, ou seja, `obj["foo"](obj,arg1,arg2)`. Se `obj` for um userdata, a operação `obj["foo"]` faz o interpretador Lua procurar na metatable de `obj` por um meta-método chamado `__index`; o interpretador então chama o meta-método com o objeto e o nome do método como argumentos.

Para os objetos CLR, o meta-método `__index` procura no tipo do objeto, usando a API de reflexão do CLR, por um método com o nome passado ao meta-método. Se for encontrado, LuaInterface retorna um delegate representando este método. Se nenhum método for encontrado, `__index` retorna `nil`.

O interpretador Lua então chama o delegate retornado por `__index`, passando o objeto e os argumentos do método. O delegate consulta, na pilha de Lua, os argumentos para o método, faz a conversão dos argumentos para os tipos que o método espera, e chama o método. Se o método for sobrecarregado então o delegate primeiro verifica qual método é compatível com os argumentos passados, só então trazendo-os da pilha e chamando o primeiro método compatível.

Usar reflexão para procurar por um método no tipo de um objeto e depois criar um delegate para ele é custoso. Logo, LuaInterface define um cache para os delegates, para pagar o custo apenas uma vez, na primeira chamada ao método. LuaInterface mantém este cache nas metatables dos objetos CLR. Todos os objetos de um mesmo tipo compartilham a mesma metatable, e por consequência o mesmo cache. Basta um objeto de determinada classe chamar um método que todos os outros objetos da mesma classe

[4]Referências que não contam como tal para efeitos de coleta de lixo.

usarão o cache caso chamem o mesmo método, o que aumenta a eficiência do cache.

Se um método é sobrecarregado também existe o custo de procurar pelo método compatível com os argumentos passados. Mesmo sem sobrecarga existe o custo de decidir como os argumentos são convertidos para os tipos que o método espera. Logo, LuaInterface mantém um segundo cache dentro do delegate. O cache guarda o último método chamado e as funções usadas para converter os argumentos. O delegate primeiro tenta converter os argumentos e chamar o método usando a informação deste segundo cache; se ocorrer algum problema ele continua com o processo normal de procurar um método compatível com os argumentos.

Retornando ao exemplo do método `foo`, se `foo` for um campo de `obj` o meta-método `__index` encontra aquele campo no tipo de `obj`, usando reflexão, e retorna o valor do campo. O meta-método `__index` também guarda o campo em um cache, para pagar o custo da procura apenas na primeira leitura do campo. Se `foo` for uma propriedade ou um evento então a operação é feita de forma análoga. Se o campo ou a propriedade não existem LuaInterface retorna `nil`.

Quando o script tenta atribuir um valor ao campo, como em `obj.foo=val`, e `obj` é um userdata, Lua procura na metatable do userdata por um meta-método chamado `__newindex`, chamando o meta-método com o objeto, o nome do campo e o valor como argumentos. O meta-método `__newindex` procura no tipo do objeto por aquele campo, converte o valor para o seu tipo e atribui o valor convertido a ele. A atribuição a uma propriedade é feita de maneira análoga. O meta-método `__newindex` aproveita o cache do meta-método `__index`. Se o campo ou a propriedade não existem LuaInterface gera um erro.

Os tipos retornados pela função `import_type` são instâncias da classe `Type`; suas buscas reflexivas (para métodos, campos, etc.) são restritas a métodos estáticos, mas fora isso são como outras instâncias de objetos. Suas metatables também possuem um meta-método `__call`; Lua chama este meta-método quando o tipo o script chama o tipo como uma função. A implementação de `__call` procura entre os construtores do tipo por um compatível com os argumentos passados e instancia um objeto usando este construtor.

2.2.6 Delegates, Eventos e Interfaces

Sempre que uma função Lua é passada onde o CLR espera um delegate, LuaInterface cria dinamicamente uma subclasse de `LuaDelegate`. Esta subclasse define um método com a assinatura do delegate, e um construtor que recebe uma função Lua como argumento. LuaInterface então cria uma instância desta subclasse e cria o delegate a partir desta instância. O delegate criado é passado para o CLR.

A subclasse de `LuaDelegate` é gerada usando a API `Reflection.Emit` do CLR. A API `Reflection.Emit` tem classes para gerar *assemblies*, tipos, e emitir bytecodes da Common Intermediate Language. Os tipos criados pela API podem ser mantidos na memória (e serem temporários) ou gravados em disco (e serem permanentes). As subclasses de `LuaDelegate` que LuaInterface gera são mantidas apenas na memória. LuaInterface guarda a classe em um cache, e ela é reutilizada se LuaInterface precisar de outro delegate do mesmo tipo.

LuaInterface retorna eventos como objetos que implementam um método `Add` e um método `Remove`. O método `Add` recebe uma função Lua e cria um delegate com a mesma assinatura dos tratadores do evento. O método `Add` então registra este delegate como um tratador do evento, e retorna o delegate. O método `Remove` recebe um delegate previamente registrado por `Add` e o remove.

A função `make_object` também usa a API `Reflection.Emit` do CLR para gerar suas classes.