Expr и Eval - вычисление выражений в разных контекстах

admin
December 2021 edited March 18

Функция Expr возвращает выражение, а Eval выполняет его в текущем контексте, т.е. с переменными, видимыми из места вызова. Сначала простой пример:

```
int a, b;
object e=Expr(a+b); //a+b - это не строка, как в eval в JavaScript, а реальное выражение
a=2; b=3;
int c=e.Eval(int); //в Eval можно задать, какого типа вернуть значение (по умолчанию возвращает null)
//e.Eval(int) - это то же самое, что Eval(e, int).
WriteLine(c); //5
a=5; b=10;
string str=e.Eval(string);
WriteLine(str); //15

ReadKey();
```

Выражения в Expr могут быть любой сложности, например:

```
object e=Expr((5*10-Ceiling(2.55))/2);
WriteLine(e.Eval(string)); //23,5
```

А с помощью специальных функций можно делать выражения из нескольких последовательных действий:

```
int a, b, c;
object e=Expr(@(a=1, b=a+2, c=a+b)); //функция @(arg0, arg1, arg2) просто выполняет все действия и возвращает результат последнего (тип objec
WriteLine(e.Eval(string)); //4
WriteLine($"a={a}, b={b}, c={c}"); //a=1, b=3, c=4
```

```
e=Expr(_(a=1, b=a+2, c=a+b)); //функция _(arg0, arg1, arg2) выполняет все аргументы и возвращает значение первого
WriteLine(e.Eval(string)); //1

e=Expr(__(c=a+b, a=1, b=a+2)); //функция __(arg0, arg1, arg2) выполняет все аргументы, начиная со второго, и возвращает значение первого (выч
WriteLine(e.Eval(string)); //4
```

Теперь более сложный пример:

```
int[] arr=new int[]{5, 6, 7};
object someExpr=GetExpr(); //получаем выражение
ForAllElements(arr, someExpr); //передаём в функцию массив, который нужно изменить, и выражение
WriteLine(Join(' ', arr)); //15 16 17 - к каждому элементу прибавилось 10
ReadKey();

int[] ForAllElements(int[] arr, object e){
    //эта функция для каждого индекса элемента массива выполняет выражение, которое меняет значение элемента
    foreach(int index in Range(arr.Length()))
        e.Eval(); //выражение arr[index]+=10 выполняется в контексте функции ForAllElements, т.е. с локальными переменными arr и index

}

object GetExpr(){
    //эта функция создаёт выражение с массивом и индексом
    int[] arr;
    int index;
    return Expr(arr[index]+=10); //суть выражения - прибавление к элементу числа 10
}
ReadKey();
```

Теперь пример с ошибкой:

```
int a, b;
object e=Expr(a+b);
```

```
Test(e);
ReadKey();

Test(object e){
    int a=2, b=3;
    WriteLine(e.Eval(int)); //0
}
```

Вместо 5 получили 0! Это потому, что складывались не локальные переменные функции Test, а переменные a и b, которые в первой строке (равны 0). Переменные вне функций - это переменные экземпляра, в данном случае главного класса App, в который оборачивается весь код. Функция Expr взяла выражение, в котором a и b являются экземплярными. Поэтому Eval выполнил 0+0, а не 2+3.
Если выражение должно работать с ref-переменной, то и при получении выражения переменная в нём должна быть переданной по ссылке:

```
//код выше пропущен
object GetExpr(ref int[] arr){ //при вызове GetExpr надо будет передать какой-нибудь массив (чисто формально)
    int index;
    return Expr(arr[index]+=10); //суть выражения - прибавление к элементу числа 10
}
```

Конечно, это неудобно, поэтому я сделал функцию SetVarScopeKind, которой можно задавать вид (статическая, экземплярная, локальная, ссылка) переменных в выражении. О ней будет ниже.

Вот ещё пример, в котором выражение для изменения экземплярной переменной формируется в другом классе:

```
object e=new Bar().GetExpr();
Foo f=new Foo();
f.SetX(e);
f.PrintX(); //555
ReadKey();

class Foo{
    int X;
```

```
    public PrintX(){WriteLine(X);}
    public SetX(object e){e.Eval();}
}

class Bar{
    int X;
    public object GetExpr(){return Expr(X=555);} // вместо X=555 можно написать this.X=555
}
```

Функция GetExpr находится в классе Bar, но для Expr(X=555) это не важно. Главное, что в выражении X будет значиться как экземплярная. И когда выражение выполняется в классе Foo, то X будет ссылаться именно на X в Foo, а не в Bar.
Перепишем для статических переменных:

```
object e=Bar.Expression;
Foo f=new Foo();
f.SetX(e);
f.PrintX(); //555
ReadKey();

class Foo{
    public static int X; //нужно, чтобы X была публичной, чтобы в Bar можно было создать выражение
    public PrintX(){WriteLine(X);}
    public SetX(object e){e.Eval();}
}

static class Bar{
    static int X;
    public static object Expression=Expr(Foo.X=555); //если вместо Foo.X=555 написать X=555, то X будет ссылаться на X в Bar
}
```

На самом деле в этих двух случаях проще создавать выражения прямо в классе Foo, а не в Bar.

Теперь о функции SetVarScopeKind

Эта функция позволяет менять вид (область видимости) переменных в выражении.

```
Test();

Test(){
    int A, B, C, x;
    object e=Expr(x=A+B+C); //все переменные локальные, а в Foo A и B статические, C экземплярная, а x - ссылка
    e.SetVarScopeKind("Static", A, B); //делаем A и B статическими
    e.SetVarScopeKind("This", C); //делаем C экземплярной
    e.SetVarScopeKind("Ref", x); //делаем x ссылкой
    //Можно одной строкой: e.SetVarScopeKind("Static", A, B).SetVarScopeKind("This", C).SetVarScopeKind("Ref", x);
    int v=5;
    Foo f=new Foo();
    f.Go(e, v);
    WriteLine(v); //9
    e.SetVarScopeKind("Local", C); //делаем C локальной переменной функции
    f.Go(e, v);
    WriteLine(v); //107
}

class Foo{
    static int A=3, B=4;
    int C=2;
    public Go(object e, ref int x){
        int C=100;
        e.Eval();
    }
}
```

Если меняете тип на статический для уже статической переменной, то привязка её к конкретному классу сбрасывается.

```
static int X=123;
object e=Expr(WriteLine(Foo.X));
e.Eval(); //5
e.SetVarScopeKind("Static", X); //X и так статическая, но теперь она не будет привязана к Foo, а будет браться из класса, в котором выполняет
e.Eval(); //123
```

```
Bar.EvalExpr(e); //555

class Foo{
    public static int X=5;
}
class Bar{
    public static int X=555;
    public static EvalExpr(object e){
        e.Eval();
    }
}
```

Вовлечение переменных

В Expr можно указать, какие переменные вовлечь в выражение:

```
Test();

Test(){
    int x=2, y=3;
    object e=Expr(x+y, x, y); //переменные указываются после выражения
    //ещё можно вовлекать переменные так: e.Involve(x, y);
    EvalAndPrint(e);
}

EvalAndPrint(object e){
    WriteLine(e.Eval(int)); //5 - сработало потому, что были вовлечены переменные из Test()
}
```

Я специально не использую термины "захват" и "замыкания" потому, что в OverScript это работает немного иначе, и термин "вовлечение" лучше отражает суть. С экземплярами переменными всё как обычно: экземпляры с вовлечёнными в выражение переменными будут существовать, пока существует само выражение. Локальные переменные тоже можно вовлечь, но они не удерживаются выражением, а значит, если выражение создано в функции с её локальными переменными, а потом функция

завершила работу, то переменные в выражении станут указывать на неактуальную область памяти.
Давайте посмотрим на пример неправильного вовлечения:

```
object e = GetExpr();
//в e выражение x+y, но на что ссылаются его переменные, если GetExpr завершила работу и освободила занятую область памяти, в которой находят
EvalAndPrint(e);

object GetExpr(){
    int x=2, y=3;
    return Expr(x+y, x, y); //вовлекаются локальные переменные
}

EvalAndPrint(object e){
    WriteLine(e.Eval(int)); //5 - сработало, хотя не должно было. Как так-то?!
}
```

Этот пример не должен работать корректно, но дело в том, что по завершении работы функций очищаются только ссылочные переменные, а значимые остаются и потом перезаписываются другими значениями. Это сделано, чтобы не тратить лишнее время на очистку. Вот пример, демонстрирующий перезапись использованных переменных:

```
object e = GetExpr();
//после завершения GetExpr в памяти остаются x=2 и y=3.
SomeFunc(); //SomeFunc будет запущена в той же области памяти, что и GetExpr, а значит перезапишет x и y!
//в e выражение x+y, в котором переменные x и y ссылаются на новые значения
EvalAndPrint(e);

object GetExpr(){
    int x=2, y=3;
    return Expr(x+y, x, y);
}

EvalAndPrint(object e){
    WriteLine(e.Eval(int)); //25!!! Это 10+15 заданные функцией SomeFunc. Теперь вы видели всё!
}
```

```
SomeFunc(){
    int x=10, y=15;
}
```

Очень важно понять, что происходит в этом примере! Это поможет избежать критических ошибок. Вовлекать локальные переменные можно только тогда, когда Eval будет выполнена до завершения функции, в которой создано выражение.

Ещё один уже правильный пример:

```
Foo f=new Foo(2, 3);
object e=f.GetSumExpr(); //получаем выражение x+y, где x и y - переменные из Foo
int v=e.Eval(int);
WriteLine($"{f}, {e}={v}"); //x=2, y=3, x+y=5
f.SetXY(5, 2);
v=e.Eval(int);
WriteLine($"{f}, {e}={v}"); //x=5, y=2, x+y=7

class Foo{
    int x, y;

    New(int x, int y){ //конструктор
        SetXY(x, y);
    }
    public object GetSumExpr(){ //возвращает выражение x+y
        return Expr(x+y, x, y); //указываем не только выражение, а и вовлечённые в него переменные (экземплярные x и y)
    }
    public SetXY(int x, int y){ //устанавливает значения x и y
        this.x=x;
        this.y=y;
    }
    string ToString(){ //возвращает строковое представление объекта (вызывается автоматически при преобразовании в строку)
        return $"x={x}, y={y}";
    }
}
```

Экземпляр Foo будет удерживаться от удаления пока существует ссылка на него в выражении.

Статические переменные не подлежат вовлечению, т.к. они и так ссылаются на конкретные места. А ref-переменные нельзя т.к. они являются ссылками (как и в C#).

Упаковка (прикрепление) переменных

Функцией Box можно упаковать в выражение текущие значения указанных переменных.

```
int x=10;
object[] arr=new object[3];
int count=arr.Length();
for(int i=0; i<count; i++)
    arr[i]=Expr(WriteLine("sum="+(i+x))).Box(i, x); //получение выражения и упаковка в него значений i и x

//в arr теперь следующие выражения:
//WriteLine("i="+(0+10))
//WriteLine("i="+(1+10))
//WriteLine("i="+(2+10))

x=20; //это не повлияет на выражения потому, что Box упаковывает не ссылки, а значения
Test(arr);
//sum=10
//sum=11
//sum=12

Test(object[] arr){
    foreach(object e in arr)
        e.Eval();
}
```

Можно упаковывать сразу все переменные или только локальные функцией BoxAll:

```
int X=5;
Test();
```

```
ReadKey();

Test(){
    string s="Hello!";
    var e=Expr(WriteLine("s="+s+"; X="+X)).BoxAll(); //BoxAll упаковывает все значения (локальные, ref-переменные, экземплярные и статические
    s="test"; X=10;
    e.Eval(); //s=Hello!; X=5
    //если написали BoxAll(true), то результат был бы: s=Hello!; X=10
}
```

Упакованные значения существуют пока существует ссылка на выражение, в котором они упакованы. Потом все задействованные объекты уничтожаются сборщиком мусора.

Переупаковка и перевовлечение

Можно перезаписывать ранее заданные значения. Пример с переупаковкой:

```
int a=2, b=3;
object e=Expr(WriteLine(a+b)).Box(a, b);
e.Eval(); //5
a=4;
e.Box(a, b).Eval(); //7
b=5;
e.Box(a, b).Eval(); //9
```

Пример с перевовлечением:

```
int[] arr;
int index;
object e=Expr(WriteLine(arr[index])); //получение выражения, в котором arr и index экземплярные

new Foo().InvolveVars(e); //в этой функции вовлечение делается в первый раз
//в e будет WriteLine(arr[index]), где arr и index из экземпляра Foo, созданного ранее
Test(e); //30
```

```
new Bar().InvolveVars(e); //а в этой второй раз, т.е. ранее вовлечённые переменные будут перезаданы
//теперь arr и index из экземпляра Bar
Test(e); //200
//экземпляры Foo и Bar будут существовать, пока существует выражение в e


Test(object e){
    e.Eval();
}

class Foo{
    int[] arr=new int[]{10, 20, 30};
    int index=2;
    public InvolveVars(object e){
        e.Involve(arr, index); //вовлечение в выражение arr и index из этого экземпляра Foo
    }
}

class Bar{
    int[] arr=new int[]{100, 200, 300};
    int index=1;
    public InvolveVars(object e){
        e.Involve(arr, index); //вовлечение в выражение arr и index из этого экземпляра Bar
    }
}
```

Функции Involve и Box изменяют данные в выражении, и если выражение уже выполняется в другом потоке, то эти изменения отразятся на результате.

```
int x=10;
object e=GetExpr().Box(x); //сразу упаковываем x=10
object timer=LocalTimer(e.Eval(), 1000, true); //создаётся и запускается таймер, выполняющий выражение
Sleep(5000);
x=15;
e.Box(x); //а теперь упаковываем 15 вместо 10
Sleep(5000);
```

```
timer.StopTimer();
//10 10 10 10 10 15 15 15 15 15

object GetExpr(){
    int x;
    return Expr(Write(x+" "));
}
```

Если вам такое поведение не нужно, то нужно сначала запомнить оригинал выражения функцией FixOrigExpr, а потом на основе копии оригинала, которая возвращается функцией OrigExpr, создавать новые выражения.

```
int i=5;
int x=10;
object e=Expr(WriteLine("sum="+(i+x))).Box(i); //получение выражения и упаковка в него значения i
//обратите внимание, что i и x экземплярные, но менять их тип на локальный не обязательно т.к. после упаковки они будут конкретными значениям
e.FixOrigExpr(); //запоминаем текущую версию выражения как оригинал (WriteLine("sum="+(5+x)))
e.Box(x); //упаковка x, но она не затрагивает оригинал
//выражение теперь: WriteLine("sum="+(5+10))

Test(e); //sum=15

//делее выполним выражение, но с новым значением x
x=20;
e=e.OrigExpr().Box(x); //OrigExpr возвращает копию оригинала, т.е. WriteLine("sum="+(5+x)), а после в неё упаковывается x
//выражение теперь: WriteLine("sum="+(5+20))
Test(e); //sum=25

//повторим с новым x:
x=30;
e=e.OrigExpr().Box(x);
//выражение теперь: WriteLine("sum="+(5+30))
Test(e); //sum=35

Test(object e){
    e.Eval();
}
```

В функцию OrigExpr можно передать переменные для вовлечения:

```
int x=1, y=2;
object e=Expr(WriteLine(x+y)).FixOrigExpr();
e.Box(x, y);
e.Eval(); //3

object e2=new Foo().GetExpr(e);
e2.Eval(); //7 //складываются 4 и 3 из экземпляра Foo

e.Box(x, y);
e.Eval(); //3 // выражение в e осталось прежним

class Foo{
    int x=4, y=3;
    public object GetExpr(object e){
        return e.OrigExpr(x, y); //OrigExpr вернёт копию выражения на момент вызова FixOrigExpr, а также вовлечёт в это новое выражение перем
        //OrigExpr не затрагивает исходное выражение
    }
}
```

Нужно понимать, что Box, Involve и SetVarScopeKind изменяют выражение не моментально, и может получиться, что это выражение, выполняемое другим потоком, может повести себя не так как надо из-за того, что одни его члены изменены, а другие нет. Например, в выражении x+x в операндах могут оказаться разные значения.

Есть ещё функция Renew, которая обновляет (заменяет) выражение другим. При этом в выражении остаётся оригинал, если он был зафиксирован функцией FixOrigExpr. Поясню: экспрешн - это объект ссылочного типа с двумя полями (не путать с System.Linq.Expressions.Expression). В одном лежит главное выражение, а в другом оригинальное выражение. Renew заменяет только главное.

```
object e=Expr(Write("Hello! "));
object timer=LocalTimerByExpr(e, 1000, true);
Sleep(3000);
e.Renew(Expr(Write("Test! ")), true); //true указывает, что нужно сделать горячее обновление
Sleep(3000);
timer.StopTimer();
//Hello! Hello! Hello! Test! Test! Test!
```

Горячее обновление - замена членов выражения (главное выражение) членами из указанного выражения, а не присваивание ссылки. При создании локального таймера захватывается ссылка не на весь экспрешн, а только ссылка на главное выражение в нём. Поэтому, если написать просто `e.Renew(Expr(Write("Test! ")));` на таймер это никак не повлияет, т.к. в нём останется ссылка на прежнее выражение. То же самое с локальными задачами и потоками, создаваемыми функциями LocalTaskByExpr, LocalThreadByExpr.
Если вы используете инструкцию apply$, которая выполняет выражение с кешированием ссылки на него, то вот ещё пример с горячим обновлением выражения.

```
object e=Expr(Write("Hello! "));
object t=RunLocalTask(Test(e));
Sleep(3000);
e.Renew(Expr(Write("Test! ")), true);
t.Wait();
//Hello! Hello! Hello! Test! Test! Test! Test!
ReadKey();

Test(object e){
    foreach(int i in Range(7)){
        apply$ e;
        Sleep(1000);
    }
}
```

Renew не является потокобезопасной. Если обновление происходит в момент выполнения выражения в другом потоке, то может возникнуть ошибка или результат будет неверный.

Резюмируем

Expr получает выражение, но только статические переменные в нём указывают на конкретные области памяти. Остальные - это просто имя + вид (статическая, локальная и т.д.). Вид можно поменять функцией SetVarScopeKind. Если поменять вид на статический, то значение переменной будет браться из класса в котором выполняется выражение.

Переменные можно вовлекать, а можно упаковывать. В выражении вовлеченным переменным можно присваивать новые значения, а упакованным нет, т.к. они уже не переменные, а постоянные значения.
