

QUERIES ZUR LAUFZEIT DYNAMISCH ERSTELLEN

Frag mich nicht immer dasselbe!

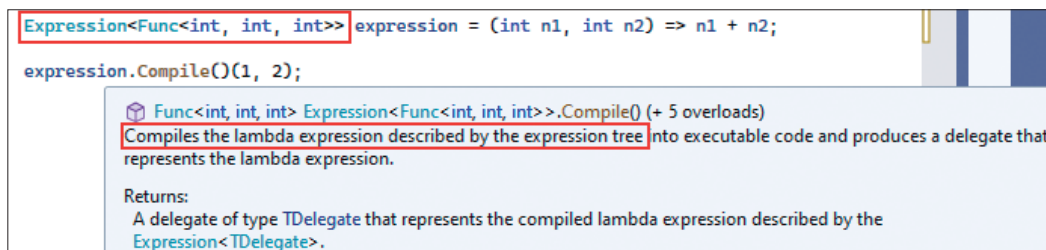
Dynamische Abfragen ermöglichen eine große Flexibilität beim Datenabruf.

Anders als bei statischen Queries, die bereits zur Kompilierungszeit definiert sind und unverändert bleiben, können Entwickler durch den Einsatz von dynamischen Queries mehr Flexibilität und Vielseitigkeit in ihr System bringen, zum Beispiel wenn es um die Erstellung flexibler Filterbedingungen geht (vergleiche [1]).

Doch bevor wir die dynamischen Abfragen in den Fokus rücken, werden wir zunächst die sogenannten Expression Trees betrachten – sie spielen in diesem Kontext die tragende Rolle (vergleiche [2]).

Expression Trees erklärt

Es handelt sich hierbei um die Möglichkeit, Programmcode als Daten in der Form eines Baums darzustellen, wobei jeder Knoten des Baums Folgendes repräsentieren kann:



```
Expression<Func<int, int, int>> expression = (int n1, int n2) => n1 + n2;
expression.Compile()(1, 2);
```

Func<int, int, int> Expression<Func<int, int, int>>.Compile() (+ 5 overloads)
Compiles the lambda expression described by the expression tree into executable code and produces a delegate that represents the lambda expression.

Returns:
A delegate of type TDelegate that represents the compiled lambda expression described by the Expression<TDelegate>.

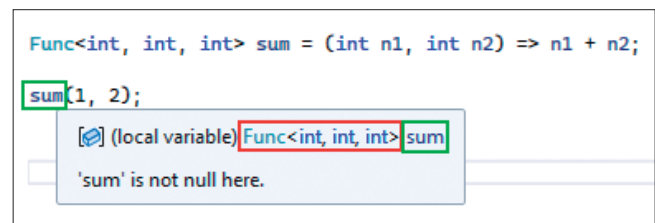
Expression Tree aus einer Lambda Expression (Bild 2)

- eine Operation
- eine Variable
- eine Konstante

Eingesetzt werden Expression Trees, wenn es darum geht, zur Laufzeit Code zu manipulieren. Unter anderem also dann, wenn dieser transformiert oder in eine andere Sprache umgesetzt werden soll, wie zum Beispiel bei der Abfrage einer Datenbank.

Erstellung von Expression Trees

Erstellt werden kann ein Expression Tree zum Beispiel aus einer Lambda Expression oder unter Verwendung der *Expression*-Klasse. Bild 1 zeigt die erste Variante, bei der die Lambda Expression zwei Integers



```
Func<int, int, int> sum = (int n1, int n2) => n1 + n2;
sum(1, 2);
```

(local variable) Func<int, int, int> sum

'sum' is not null here.

Eine Lambda Expression als Ausgangsbasis (Bild 1)

entgegennimmt und deren Summe als Rückgabe liefert. Die Ausführung entspricht syntaktisch einem Funktionsaufruf.

Um aus einer solchen Lambda Expression einen neuen Expression Tree zu konstruieren, wird der Delegate *TDelegate* ersetzt durch *Expression<TDelegate>* (siehe Bild 2). Der Tool-

tip liefert einen schönen Hinweis auf den beschreibenden Charakter des Expression Trees.

Wie schon erwähnt, kann der Expression-Tree auch durch die Factory-Funktionen der Klasse *Expression* erstellt werden, wie in Bild 3 zu sehen. Hier wird zunächst für jeden Sum-

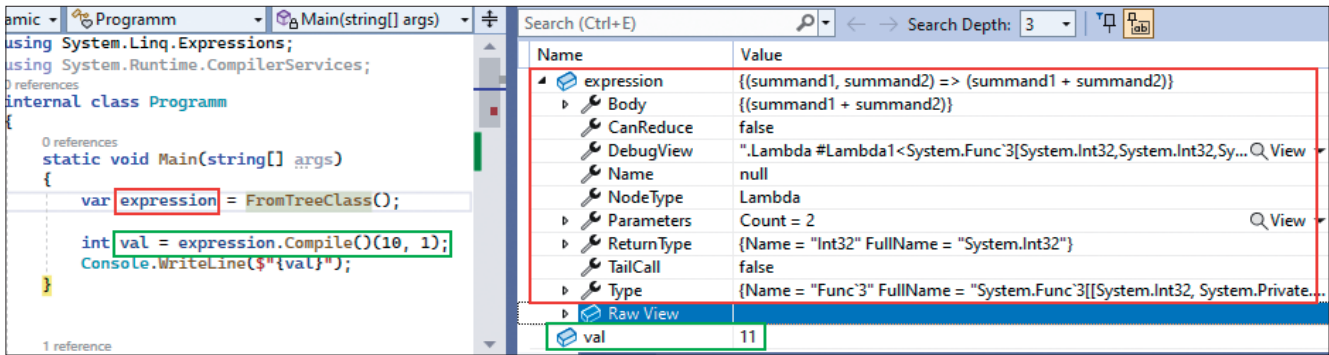
```
public static Expression<Func<int, int, int>> FromTreeClass()
{
    ParameterExpression summand1 = Expression.Parameter(typeof(int), "summand1");
    ParameterExpression summand2 = Expression.Parameter(typeof(int), "summand2");

    BinaryExpression body = Expression.Add(summand1, summand2);

    Expression<Func<int, int, int>> sum =
        Expression.Lambda<Func<int, int, int>>(body, summand1, summand2);

    return sum;
}
```

Expression mit Factory-Funktionen erstellen (Bild 3)



Die erstellte und kompilierte Expression (Bild 4)

manden ein Parameter erstellt. Die Operation zum Addieren der Summanden wird durch den folgenden Aufruf festgelegt:

```
BinaryExpression body = Expression.Add(
    summand1, summand2);
```

Nun soll eine Expression erstellt werden, die den gleichen Lambda-Ausdruck enthält wie zuvor. Dies geschieht durch folgenden Funktionsaufruf:

```
Expression<Func<int, int, int>> sum = Expression.Lambda<
    Func<int, int, int>>(body, summand1, summand2);
```

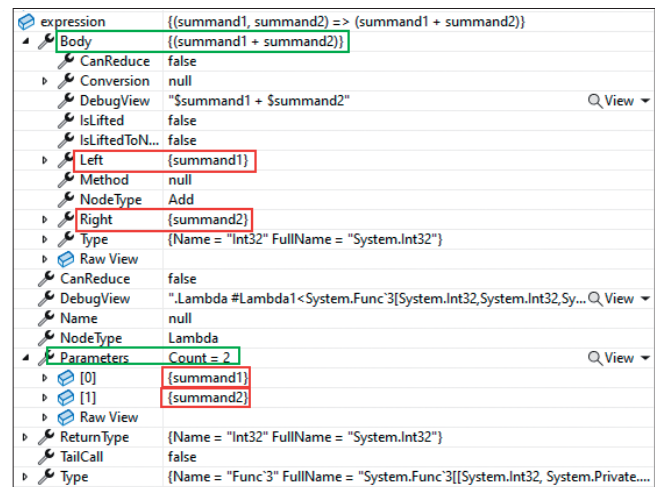
Die Expression kann im Gegensatz zu einem Delegate nicht direkt ausgeführt werden, da die Expression lediglich die Daten beziehungsweise die Beschreibung dessen enthält, was die notwendige Transformation in die Zielsprache – zum Beispiel SQL – enthalten soll. Die Expression muss zunächst kompiliert werden. Bild 4 zeigt die Darstellung der Daten (Beschreibung) der Expression und das erwartete Resultat, nachdem die Expression übersetzt und ausgeführt wurde.

Die Markierungen in Bild 5 sollen darauf hinweisen, dass die beiden Parameter *summand1* und *summand2* sowohl im Body der Expression in den Properties *Left* und *Right* zur Verfügung stehen als auch in der Parameterliste *Parameters*.

Die gesamte Expression kann als Root betrachtet werden, die auf der zweiten Ebene den Body (eine *BinaryExpression*) enthält. Auf der dritten Ebene sind die beiden Parameter *summand1* und *summand2* in den Properties *Left* und *Right* zu finden.

Einsatz von Expressions

Eingesetzt werden Expressions etwa beim Entity Framework, bei dem mittels der Lambda-Ausdrücke die Abfrage formuliert wird. Beispielweise nimmt die *Where*-Funktion eine Ex-

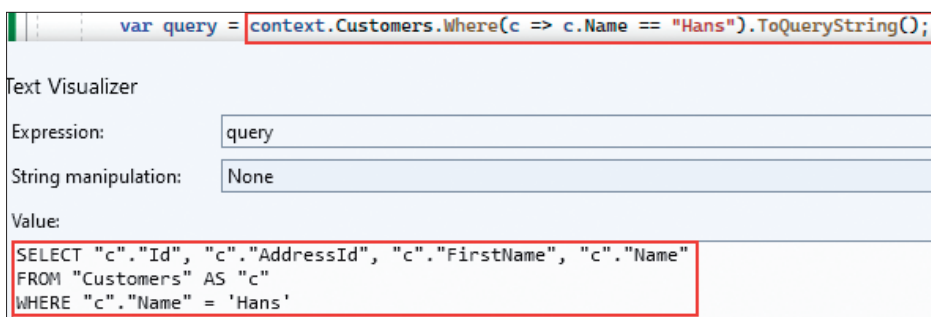


Expression Tree im Detail (Bild 5)

pression für die Filterbedingung entgegen. Der zugehörige Datenbankprovider nimmt diese Expression entgegen und kann sich dann vom Root der Expression durch die darunterliegenden Ebenen arbeiten und aus den Daten, etwa *summand1* und *summand2*, die passende SQL-Abfrage formulieren; dies wird als Query Expression Translation bezeichnet.

In Bild 6 wird mittels der Funktion *ToQueryString()* – sie ist lediglich für Debugging-Zwecke gedacht – eine Ausgabe generiert, die dies nochmals verdeutlicht.

Ein weiteres Beispiel wäre, wenn Code aus Eingabedaten generiert werden soll. Wenn zum Beispiel der Eingabestring =2*5 geparkt wird und aus diesem ein Expression Tree mit zwei Parametern für die Werte 2 und 5 erstellt und kompiliert wird. ▶



Query Expression Translation mit der Funktion *ToQueryString()* (Bild 6)

● Listing 1: Eine Expression für den Vergleich

```
private static Expression<Func<Customer, bool>>
    EqualExpression(string propertyName, object val)
{
    var param = Expression.Parameter(
        typeof(Customer), "c");
    var member = Expression.Property(
        param, propertyName);
    var constant = Expression.Constant(val);
    var body = Expression.Equal(member, constant);
    return Expression.Lambda<Func<Customer, bool>>(
        body, param);
}
```

Einschränkungen bei Expression Trees

Zu beachten sind jedoch die Einschränkungen der Expression. So können zum Beispiel Expression Trees keine *await*-Expressions oder *async*-Lambda-Expressions enthalten. Die gesamte Liste der Limitierungen ist unter [3] verfügbar.

Performance-Betrachtungen mit Expression Trees

Beim Einsatz von Expression Trees muss berücksichtigt werden, dass die Erstellung und Kompilierung einen gewissen Overhead erzeugt; immerhin muss der Speicher für die einzelnen Knoten reserviert werden, und beim Kompilieren wird IL-Code generiert, der an eine dynamische Assembly übergeben werden muss.

Es ist offensichtlich, dass sich die beschriebenen Schritte negativ auf die Performance der Anwendung auswirken können. Daher empfiehlt es sich, Folgendes bei der technischen Umsetzung zu beachten:

- Cachen des Expression Trees.
- Der Kompilierungsaufwand wächst mit der Größe des Expression Trees.
- Sie sollten nur dann eingesetzt werden, wenn es wirklich notwendig ist.

Dynamische Abfragen erstellen

Die bisherigen Abschnitte haben zum Teil bereits gezeigt, wie mit dem Expression-API dynamische Abfragen generiert werden können. Dies soll im Folgenden vertieft werden.

Listing 1 zeigt die Erstellung einer Expression, die dem Vergleich dient. Der Ausdruck *Expression.Parameter(typeof(Customer), "c")* repräsentiert den Eingabeparameter, in diesem Beispiel die Entität *Customer*, die be-

● Listing 2: Mehrere Filterattribute

```
public static Expression<Func<Customer, bool>>
    CreateEqualExpression(IDictionary<string,
        object> compareAttributes)
{
    var param = Expression.Parameter(typeof(Customer),
        "c");
    Expression body = null;
    foreach (var compareAttribute in
        compareAttributes)
    {
        var member = Expression.Property(
            param, compareAttribute.Key);
        var constant = Expression.Constant(
            compareAttribute.Value);
        var expression = Expression.Equal(
            member, constant);
        body = body == null ? expression :
            Expression.AndAlso(body, expression);
    }
    return Expression.Lambda<Func<Customer, bool>>(
        body, param);
}
```

```
var expression = EqualExpression("Name", "Hans");
var equalQuery = context.Customers.Where(expression).ToQueryString();
```

Text Visualizer	
Expression:	equalQuery
String manipulation:	None
Value:	SELECT "c"."Id", "c"."AddressId", "c"."FirstName", "c"."Name" FROM "Customers" AS "c" WHERE "c"."Name" = 'Hans'

Die Equal-Expression (Bild 7)

```
var compareAttributes = new Dictionary<string, object>();
compareAttributes.Add("Name", "Hans");
compareAttributes.Add("FirstName", "Dampf");
var expression = CreateEqualExpression(compareAttributes);
var query = context.Customers.Where(expression).ToQueryString();
```

Text Visualizer	
Expression:	query
String manipulation:	None
Value:	SELECT "c"."Id", "c"."AddressId", "c"."FirstName", "c"."Name" FROM "Customers" AS "c" WHERE "c"."Name" = 'Hans' AND "c"."FirstName" = 'Dampf'

Vergleich mit mehreren Attributen (Bild 8)

```
private static Expression<Func<Customer, bool>> EqualExpression(string propertyName, object val)
{
    var param = Expression.Parameter(typeof(Customer), "c");
    var member = Expression.Property(param, propertyName);
    var constant = Expression.Constant(val);
    var body = Expression.Equal(member, constant);

    var nullCheck = Expression.Equal(member, Expression.Constant(null));
    var condition = Expression.Condition(nullCheck, Expression.Constant(false), body);

    return Expression.Lambda<Func<Customer, bool>>(condition, param);
}
```

Vermeidung von
NullReference-
Exceptions
(Bild 9)

zeichnet wird als *c*. Mit *Expression.Property(param, propertyName)* wird eine Eigenschaft für den zuvor definierten Eingabeparameter durch den Parameternamen definiert. Mit *Expression.Constant(val)* wird ein konstanter Wert für den Ver-

Als nächstes Beispiel sollen mehrere Attribute für den Vergleich berücksichtigt werden. Die Implementierung aus [Listing 2](#) nimmt hierfür eine Liste von Key-Value-Paaren in Form eines Dictionarys entgegen. In der Iteration wird gegebenenfalls die zusätzliche Prüfung berücksichtigt.

Die Verwendung mit der generierten Ausgabe zeigt [Bild 8](#).

```
Value:
SELECT "c"."Id", "c"."AddressId", "c"."FirstName", "c"."Name"
FROM "Customers" AS "c"
WHERE CASE
    WHEN "c"."Name" IS NULL THEN 0
    ELSE "c"."Name" = 'Hans' AND ("c"."Name" IS NOT NULL)
END
```

NullReferences werden vermieden ([Bild 10](#))

gleich definiert, und *Expression.Equal(member, constant)* generiert eine *BinaryExpression*, um den Vergleich zwischen dem Member und dem konstanten Wert durchzuführen.

Abschließend wird eine Lambda-Funktion erstellt. [Bild 7](#) zeigt die zugehörige Ausführung.

Empfehlungen für die Nutzung von dynamischen Abfragen

Wie bereits erwähnt, ist die Kompilierung einer Expression aufwendig. Daher empfiehlt es sich, diese zu cachen. [Listing 3](#) zeigt einen Vorschlag für einen solchen Cache.

Es sollten *NullReferenceExceptions* vermieden werden. In [Bild 9](#) sind die relevanten Anpassungen markiert; es erfolgt ein Vergleich, ob der *propertyName* mit dem konstanten Wert *null* übereinstimmt. Wenn dies der Fall ist, wird die Entität nicht berücksichtigt. [Bild 10](#) zeigt die generierte Ausgabe.

Fazit

In diesem Beitrag wurden die deklarativen Aspekte von Expressions betrachtet und beschrieben, wie durch deren Nutzung dynamische Abfragen erstellt werden können. Neben der Erstellung von eigenen Expressions wurde noch auf einige Best Practices hingewiesen. ■

[1] CodeMaze, *How to Build Dynamic Queries With Expression Trees in C#*, www.dotnetpro.de/SL2404Abfragen1

[2] CodeMaze, *Expression Trees in C#*, www.dotnetpro.de/SL2404Abfragen2

[3] Microsoft Learn, *Limitierungen bei Expression Trees*, www.dotnetpro.de/SL2404Abfragen3

● Listing 3: Cache für kompilierte Expressions

```
internal static class ExpressionCache
{
    private static readonly Dictionary<int, Delegate>
        _hashToDelegate = new Dictionary<int, Delegate>();

    public static Func<T, bool> GetExpression<T>(
        Expression<Func<T, bool>> expression)
    {
        int hash = expression.GetHashCode();

        if (_hashToDelegate.TryGetValue(hash,
            out var @delegate))
            return (Func<T, bool>) @delegate;

        var compiled = expression.Compile();
        hashToDelegate[hash] = compiled;
        return compiled;
    }
}
```



Christian Havel

wohnt in einem Vorort von München und ist seit mehr als zwanzig Jahren in der Softwareentwicklung tätig. Er programmiert hauptsächlich in C#. Sie erreichen ihn unter christian.havel@googlemail.com.

dnpCode

A2404Abfragen