LIST<T>

Die Lieblings-Collection

Performance und Speicherbedarf von List<T> unter der Lupe.

Irgendwann ist es in jedem Projekt so weit: Irgendwann arbeiten Sie in absolut jedem .NET-Projekt mit Mengen. Mengen werden sehr häufig mit einer *List<T>* verwaltet, der generischen Allround-Menge, mit der wir jeden beliebigen Datentyp im Speicher ablegen können.

Aber wissen Sie eigentlich, wie eine *List<T>* arbeitet, und vor allem, was die Probleme einer Liste sein können? Sie haben keine Ahnung, oder doch so ein bisschen? Keine Angst, wir erklären es ausführlich.

Das API

Für den Anfang ziehen wir die Dokumentation des Datentyps zurate [1]. Auf der Suche nach Erkenntnis ist in der Regel auch ein Blick in den Originalquellcode von Microsoft auf GitHub immer sinnvoll. Doch im Fall des Datentyps *List<T>* verläuft diese Unternehmung im Nichts: Zwar sind im Runtime-Repository viele Klassen aus dem Namespace *System.Collection.Generic* zu finden, doch ausgerechnet *List<T>* ist dort nicht dabei [2].

Microsoft verwendet einen kleinen Trick. Gewisse Klassen, die aus Sicht des .NET-Teams von fundamentalem Wert sind, befinden sich in der sogenannten *System.Private.CoreLib*.

Das ist eine Bibliothek beziehungsweise ein Repository, in dem die Implementierung sehr wichtiger Klassen liegt. Diese werden beim Kompilierungsprozess von der jeweiligen Runtime (Aktuell nur noch Mono und .NET) direkt eingebunden und stellen somit eine Basisimplementierung für verschiedene Runtimes dar. Dort findet man auch die Implementierung der *List<T>* [3].

Die Verwendung

Eine *List<T>* mag eine kleine Klasse sein, wenn man die Anzahl der Methoden betrachtet, doch die Klasse selbst hat stolze 1100 Lines of Code und ist damit schon etwas mächtiger, als man meinen könnte. Beim Durchsehen des Quellcodes entdeckt man einige Aufrufe der Klasse *Array*, und es wird

```
List<string> names = new();
names.Add("Christian");
names.Add("Giesswein");

Console.WriteLine(names.Count);
```

Name	Wert		Typ
names	Count = 2		System.Collect...
[0]	"Christian"	Anzeigen	string
[1]	"Giesswein"	Anzeigen	string
Rohdatenansicht			
Capacity	4		int
Count	2		int
Statische Member			
Nicht öffentliche ...			
System.Collecti...	false		bool
System.Collecti...	false		bool
System.Collecti...	Count = 2		object {System...
System.Collecti...	false		bool
System.Collecti...	false		bool
_items	{string[4]}		string[]
[0]	"Christian"	Anzeigen	string
[1]	"Giesswein"	Anzeigen	string
[2]	null		string
[3]	null		string
_size	2		int
_version	2		int

Ein bisschen Zauberpulver, und schon wird das Array im Debugger sichtbar (Bild 1)

names	Count = 5
[0]	"Christian"
[1]	"Giesswein"
[2]	"Christian"
[3]	"Giesswein"
[4]	"Giesswein"
Rohdatenansicht	
Capacity	8
Count	5
Statische Member	
Nicht öffentliche ...	
System.Collecti...	false
System.Collecti...	false
System.Collecti...	Count = 5
System.Collecti...	false
System.Collecti...	false
_items	{string[8]}
[0]	"Christian"
[1]	"Giesswein"
[2]	"Christian"
[3]	"Giesswein"
[4]	"Giesswein"
[5]	null
[6]	null
[7]	null
_size	5
_version	5

Count gibt fünf Elemente an, Capacity steigt allerdings auf acht (Bild 2)

sehr viel mit einer Array-Referenz (intern *_items* genannt) gearbeitet.

Hier steckt auch das wichtige Know-how. .NET ist nicht in der Lage, Arrays einfach zu vergrößern oder zu verkleinern.

Stellen Sie sich dazu vor, dass das Array, das im Speicher liegt, umgeben von anderen Speicherbereichen ist. Ein einfaches Größer- oder Kleinermachen ist somit ein recht kompliziertes Unterfangen.

Gerade unter diesem Gesichtspunkt hilft uns *List<T>* gewaltig. Wir können beliebige Einträge hinzufügen, aber auch wieder entfernen, ohne uns irgendwelche Gedanken darüber machen zu müssen, wie denn nun welcher Eintrag an seinen Platz kommt.

Dieses Array kann jeder sehr einfach aufdecken. Mit dem Beispielcode in Bild 1 und mithilfe des Debuggers von Visual Studio lässt sich das Array sichtbar machen.

Das interne Array

Visual Studio zeigt in der Rohdatenansicht das Innere der Klasse. Das bedeutet, dass auch private Felder oder statische Inhalte sichtbar werden. Unter anderem sieht man hier auch das eingangs erwähnte Array *_items*. Der Beispielcode in Bild 1 hat dabei zwei Einträge hinzugefügt.

Dem aufmerksamen Leser oder der Leserin wird aber bereits aufgefallen sein, dass dieses Array nicht eine Länge von zwei, sondern tatsächlich von vier hat. Dies ist auch in den Eigenschaften der Liste noch einmal deutlich zu sehen: Der

```
195         [MethodImpl(MethodImplOptions.AggressiveInlining)
196         public void Add(T item)
197         {
198             _version++;
199             T[] array = _items;
200             int size = _size;
201             if ((uint)size < (uint)array.Length)
202             {
203                 _size = size + 1;
204                 array[size] = item;
205             }
206             else
207             {
208                 AddWithResize(item);
209             }
210         }
```

Die Add-Methode von List<T> (Bild 3)

Count beträgt zwei, wohingegen die *Capacity* vier aufweist. Fügen wir der Liste nun fünf Elemente hinzu, sieht man, dass *Count* zwar wie erwartet auf fünf steigt, die *Capacity* dann aber auf einmal acht beträgt (Bild 2).

Die *Capacity* steigt somit nicht linear an. Zeit, tiefer in den Code abzutauchen.

Hinzufügen kann teuer sein

Im Quellcode auf GitHub ist die Methode *Add* schnell auffindbar. Siehe dazu [4] und Bild 3. Diese Methode ist sogar recht überschaubar und offenbart uns die interne Logik von unserer beliebten *List<T>*.

Es gibt ein privates Feld *_version,* welches bei jeder Operation inkrementiert wird. In den nächsten Zeilen wird überprüft, ob die *_size* kleiner ist als die aktuelle Länge des internen Arrays *_items*. Sollte sie (noch) kleiner sein, so wird die *_size* erhöht und der Eintrag wird an die neue Stelle im Array eingehängt.

Sollte jedoch der Grenzfall eintreten, dass *_size* größer oder gleich der aktuellen Länge ist, so wird die Methode *AddWithResize* aufgerufen. Diese Methode ruft dabei über einen kleinen Umweg die Methode *Grow* auf (Bild 4).

Und hier versteckt sich auch schon die Erklärung des Verhaltens. Die aktuelle Länge des Arrays wird einfach immer verdoppelt. Das bedeutet, wir haben 4, 8, 16, 32, 64, 128 und so weiter als mögliche Arraygrößen.

Die neu berechnete Größe wird einfach der Property *Capacity* zugewiesen, und in dieser Property passiert anschließend die eigentliche Action für die Aktion „vergrößern" (Bild 5).

Diese Methode prüft, ob die neue *Capacity* ausreichend, also nicht kleiner der aktuellen *_size* ist, und kümmert sich anschließend um die Vergrößerung des internen Arrays. Dabei wird innerhalb des Setters ▶

```
421         private void Grow(int capacity)
422         {
423             Debug.Assert(_items.Length < capacity);
424
425             int newcapacity = _items.Length == 0 ? DefaultCapacity : 2 * _items.Length;
426
427             // Allow the list to grow to maximum possible capacity (~2G elements) before encountering overflow.
428             // Note that this check works even when _items.Length overflowed thanks to the (uint) cast
429             if ((uint)newcapacity > Array.MaxLength) newcapacity = Array.MaxLength;
430
431             // If the computed capacity is still less than specified, set to the original argument.
432             // Capacities exceeding Array.MaxLength will be surfaced as OutOfMemoryException by Array.Resize.
433             if (newcapacity < capacity) newcapacity = capacity;
434
435             Capacity = newcapacity;
436         }
```

Die Funktion Grow verdoppelt die Größe des Arrays (Bild 4)

```
public int Capacity
{
    get => _items.Length;
    set
    {
        if (value < _size)
        {
            ThrowHelper.ThrowArgumentOutOfRangeException(ExceptionArgument.value, ExceptionResource.ArgumentOutOfRange_SmallCapacity);
        }

        if (value != _items.Length)
        {
            if (value > 0)
            {
                T[] newItems = new T[value];
                if (_size > 0)
                {
                    Array.Copy(_items, newItems, _size);
                }
                _items = newItems;
            }
            else
            {
                _items = s_emptyArray;
            }
        }
    }
}
```

Hier passiert es: Die Größe wird angepasst (Bild 5)

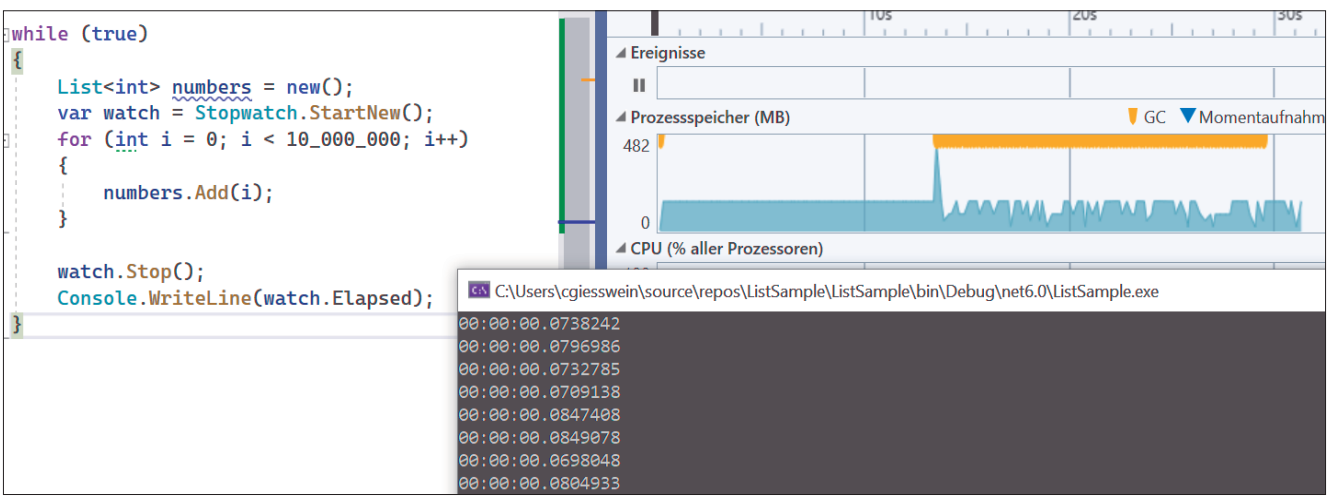
von *Capacity* ein neues Array instanziert, die alten Werte werden in das neue Array übertragen und anschließend die Referenz auf das alte Array ersetzt – Schwupps haben wir mehr Platz. Aber Achtung: Das alte Array wird dem Garbage Collector überlassen.

Wer viel anhängt, erntet viel Müll

Daraus lässt sich nun direkt eine wichtige Erkenntnis ableiten: Fügt man zu einer Liste sehr schnell sehr viel hinzu, entsteht dabei durchaus erwähnenswerter Müll (Bild 6).

Mit einem kleinen Beispielprogramm, welches in einer Schleife 10 Millionen Zahlen in die Liste wirft, erkennt man gerade mit Visual Studio, dass der Garbage Collector ordentlich zu arbeiten hat. Die gelben Dreiecke, die in Bild 6 eher aussehen wie ein Balken, sind tatsächlich die Iterationen des Garbage Collectors. Dies ist aufgrund unserer Vorgehensweise und des internen Codes durchaus sinnvoll. Die Frage, die wir uns stellen müssen, ist, ob wir das optimieren können. Und ja, das ist durchaus möglich.

Die *List<T>* hat einen überladenen Konstruktor, dem wir eine initiale *Capacity* mitgeben können. Dadurch wird das interne Array *_items* automatisch mit dieser Größe instanziert. Das bedeutet, dass es zwischen Hinzufügeaktionen nicht mehr notwendig ist, dass die Implementierung von *List<T>*

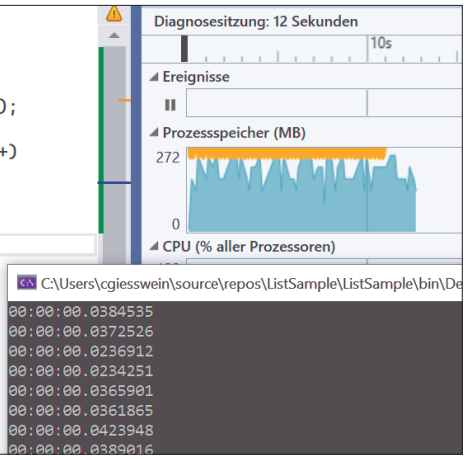


Wer viel anhängt, erzeugt viel Müll (Bild 6)

```
using System.Diagnostics;

while (true)
{
    List<int> numbers = new(10_000_000);
    var watch = Stopwatch.StartNew();
    for (int i = 0; i < 10_000_000; i++)
    {
        numbers.Add(i);
    }

    watch.Stop();
    Console.WriteLine(watch.Elapsed);
}
```



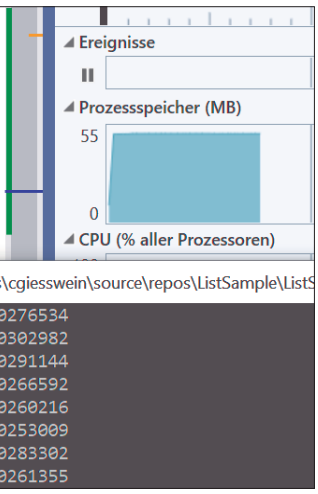
Initialisieren von List<T> mit einer großen Capacity erhöht die Performance (Bild 7)

```
1076        public void TrimExcess()
1077        {
1078            int threshold = (int)(((double)_items.Length) * 0.9);
1079            if (_size < threshold)
1080            {
1081                Capacity = _size;
1082            }
1083        }
```

Die Methode TrimExcess hat die Kraft, zu schrumpfen (Bild 8)

```
List<int> numbers = new(10_000_000);
while (true)
{
    var watch = Stopwatch.StartNew();
    for (int i = 0; i < 10_000_000; i++)
    {
        numbers.Add(i);
    }

    watch.Stop();
    numbers.Clear();
    Console.WriteLine(watch.Elapsed);
}
```



Die geringsten Laufzeiten, und die Garbage Collection muss nicht anspringen (Bild 9)

ein neues Array anlegt. Das Resultat sind zwar immer noch viele Garbage-Collection-Iterationen, doch wesentlich weniger, und auch die Laufzeit hat sich natürlich deutlich verringert (Bild 7).

Räumt Remove oder Clear auf?

Bleibt nun noch die letzte Frage zu klären: Wenn Aufrufe von *Add* für eine Vergrößerung und ein Umkopieren sorgen, wie sieht es dann bei *Remove*, *RemoveRange* oder *Clear* aus. Führen diese Methoden zu einer Verkleinerung? Wenn man sich nun erneut den Code der Implementierung ansieht, wird man feststellen: Fehlanzeige!

Dies hat natürlich Gründe: Woher soll die Runtime einen geeigneten Zeitpunkt entdecken, an dem eine Verkleinerung Sinn ergibt? Es könnte ja sein, dass in der nächsten Methode die Liste wieder benötigt wird. Und außerdem müssen wir nicht alle Tage eine Liste mit mehreren Tausend oder gar Millionen Einträgen anlegen.

Doch eine Methode hat sich als Aufräumaktion herausgestellt. Die Methode *TrimExcess* (Bild 8), welche sogar *public* ist und somit von außen zugreifbar, prüft, ob es möglich ist, die *Capacity* zu verkleinern.

Dabei wird mit einer Schwelle (Threshold) gearbeitet, und wenn diese unterschritten wird, so wird mit der genauen, aktuellen *_size* (entspricht *Count* der Einträge) die *Capacity* minimiert und somit Speicher wieder freigegeben. Das bedeutet, dass unser Beispiel in der letzten Variante am schonendsten ist (Bild 9).

Nachdem in dieser Variante die Liste recycelt wird, wird das Array nur ein einziges Mal allokiert. Die *Clear()*-Methode verändert zwar die Größe des Arrays nicht, macht es aber wieder benutzbar. Dies zeigt Visual Studio deutlich in den besten Zeiten und auch, ohne eine einzige Garbage Collection angestoßen zu haben.

Fazit

Nur selten fügen wir in Listen so viele Elemente ein. Trotzdem ist der Vergleich dieser drei Beispiele beeindruckend: Wie viel Garbage-Collection-Aktivität und wie viel Laufzeitverlängerung durch eine Grundoperation verursacht werden können.

Das heißt im Umkehrschluss, dass man unter den Gesichtspunkten der Performance und des Speicherbedarfs auch bei der Verwendung einer Liste der Klasse *List<T>* ein Auge auf den Code haben sollte. ■

[1] Microsoft Docs, List<T> Class, www.dotnetpro.de/SL2205NETirol1
[2] dotnet/runtime, www.dotnetpro.de/SL2205NETirol2
[3] List.cs, www.dotnetpro.de/SL2205NETirol3
[4] Add-Methode, www.dotnetpro.de/SL2205NETirol4

Christian Giesswein
studierte Wirtschaftsinformatik in Wien und entwickelt von klein auf Software mit .NET und C#. In Tirol hat er das Unternehmen Giesswein Software-Solutions gegründet, das sich auf Individualsoftware und Consulting spezialisiert.
christian@software.tirol

dnpCode A2205NETirol