Хабрахабр Публикации Пользователи Хабы Компании Песочница Войти Регистрация

RoundGiraffe 28 октября 2014 в 10:48

Как устроен ConcurrentBag в .Net

.NET

Из песочницы

Среди concurrent коллекций наибольшей популярностью пользуется ConcurrentDictionary. Также часто используются ConcurrentQueue и ConcurrentStack.

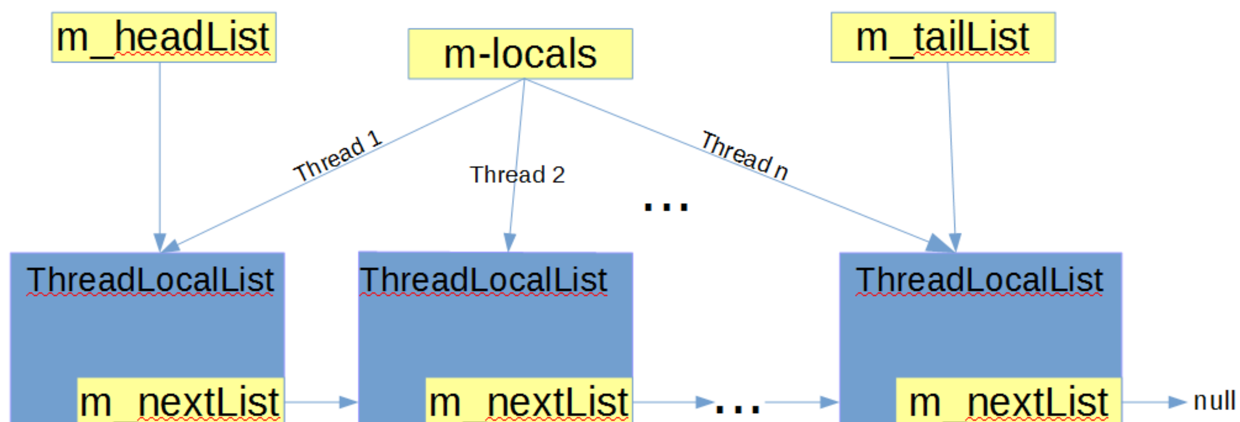
Вообще, решение локкирования частей коллекции для thread-safe хеш-таблицы является очень простым, логичным и оттого ещё более красивым.

Структура ConcurrentDictionary даже была описана в статье на хабре Под капотом у Dictionary и ConcurrentDictionary. ConcurrentBag же является не столь популярной, так как используется в основном там, где реализуется паттерн Producer-Consumer. Причем данная структура наиболее оптимально работает тогда, когда один и тот же поток занимается добавлением и изъятием данных из коллекции. Почему так происходит, будет рассказано далее.

Введение

В основе данной структуры лежит простая идея разделить все данные между потоками, чтобы каждый поток как можно чаще работал с своей «виртуальной» частью хранилища.

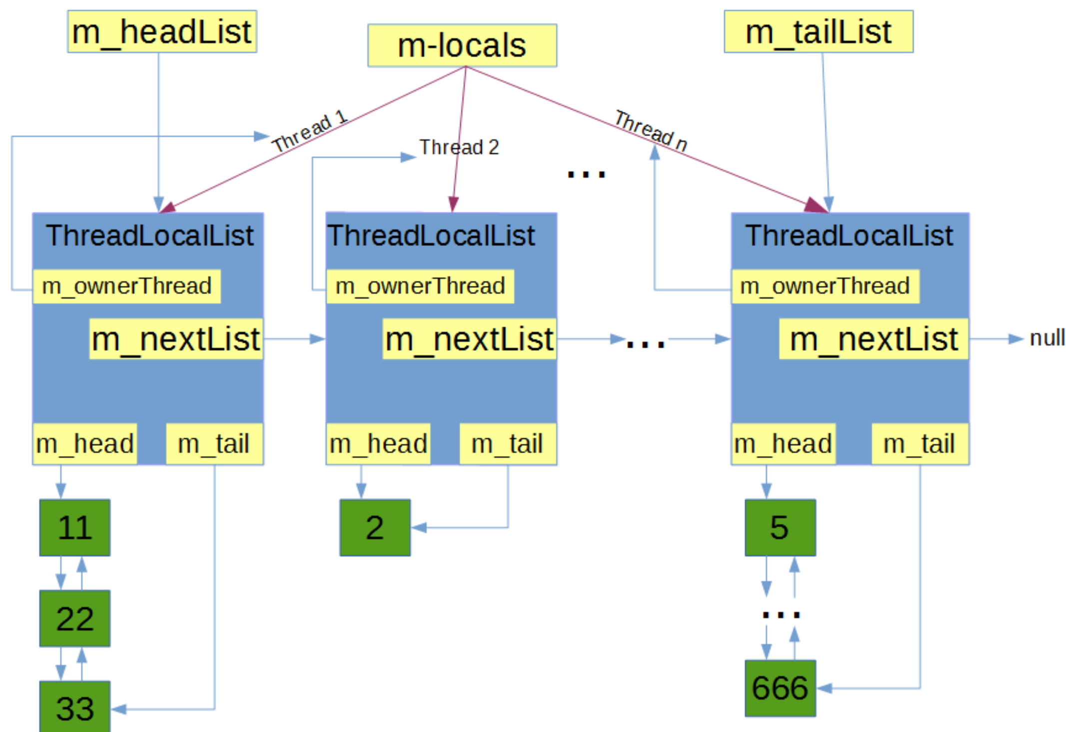
В .Net, как известно, чтобы сделать переменную не в одном экземпляре на все потоки, а свой экземпляр в каждом потоке, используется атрибут [ThreadStatic]. В .Net Framework 4.0 был добавлен класс ThreadLocal, который представляет собой удобную обертку для работы с такими данными.



В ConcurrentBag данные хранятся в ThreadLocal m_locals. То есть у каждого потока который работает с данной структурой есть свой экземпляр ThreadLocalList. Также есть volatile переменные m_headList и m_tailList, которые указывают соответственно на первый и последний элемент в m_locals. Это необходимо для того что бы получать IEnumerator, когда вам необходимо получить всю коллекцию.

Ссылки на «голову» и «хвост» в m_locals существуют, так как хранилище реализовано посредством однонаправленного связного списка. То есть у потока есть экземпляр ThreadLocalList, в этом классе есть поле ThreadLocalList m_nextList, указывающее на следующий экземпляр ThreadLocalList в другом потоке. Это значит, что из одного потока можно получить доступ ко всем экземплярам данной переменной во всех потоках, «шагая» по m_nextList.

Далее разберемся с структурой класса ThreadLocalList. Он тоже представляет собой двунаправленный связный список. Элемент представлен обычным классом Node. Указатель на первый и последний элемент это m_head и m_tail соответственно. Также стоит отметить, что есть поле Thread m_ownerThread, которое хранит ссылку на текущего владельца экземпляра. Почему на текущего, а не на создателя будет рассказано далее. В итоге получается следующая структура:



Добавление элемента

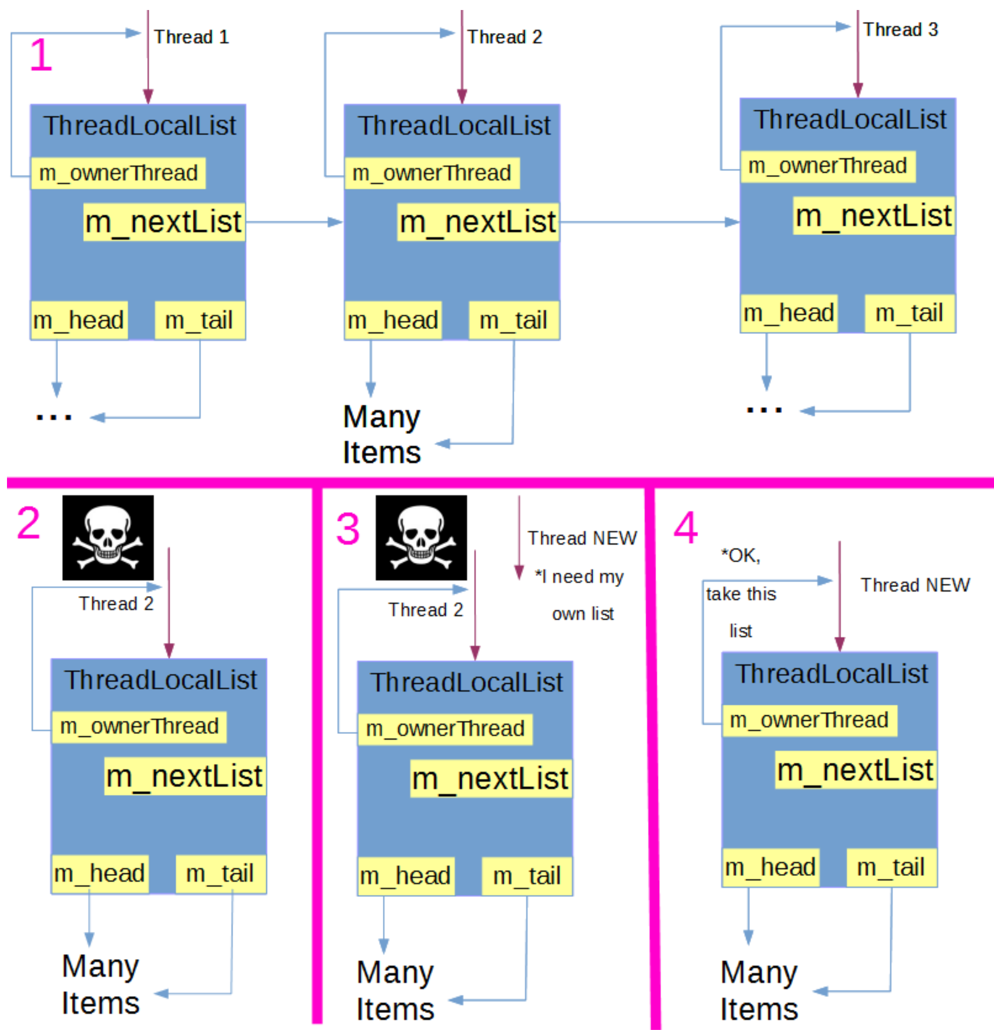
Получение ThreadLocalList

Сначала получается или создается, если не был создан, ThreadLocalList для текущего потока, соответственно, обновляются указатели m_headList и m_tailList. Причем создание происходит в синхронизированном коде, где lock стоит на GlobalListLock (тот же m_locals). Это необходимо для обновления m_tailList. Также этот lock используется, как можно догадаться по названию, везде, где нужна блокировка на всю коллекцию, то есть в CopyTo, ToArray, GetEnumerator, Count, IsEmpty через методы FreezeBag и UnFreezeBag.

Также при создании сначала пробуем найти ThreadLocalList без владельца, то есть поток, который пользовался данной коллекцией и пал смертью храбрых. Мы находим такой список, если есть, и присваиваем полю m_ownerThread ссылку на текущий поток.

Поиск неиспользуемого списка

```
private ThreadLocalList GetUnownedList()
{
    ThreadLocalList currentList = m_headList;
    while (currentList != null)
    {
        if (currentList.m_ownerThread.ThreadState == System.Threading.ThreadState.Stopped)
        {
            currentList.m_ownerThread = Thread.CurrentThread;
            return currentList;
        }
        currentList = currentList.m_nextList;
    }
    return null;
}
```

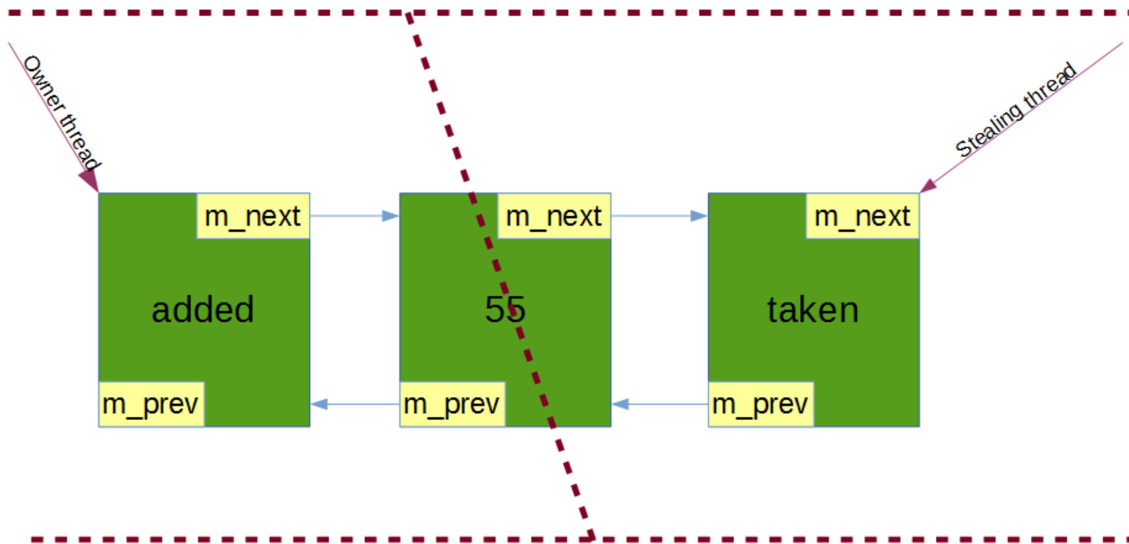


Добавление элемента в ThreadLocalList

Вторым шагом является добавление элемента в ThreadLinkedList. Он добавляется стандартно в «голову» без блокировок. Однако, если количество элементов в ThreadLocalList меньше двух, то накладывается блокировка на текущий экземпляр листа при добавлении элемента, так как в этом случае возможны потери данных. Это связано с тем, что другой поток в это время может забирать данные из ThreadLocalList текущего потока (stealing thread).

Получение элемента из коллекции

Когда поток хочет забрать элемент из коллекции, он сначала идет в свой ThreadLocalList и если он не пустой — берет элемент с «головы» связного списка. Если же локальное хранилище пусто, он идет через m_nextList по всем хранилищам других потоков и ищет не пустой список. Если находит, то «ворует» (steal) элемент оттуда. Причем он должен «своровать» элемент, не перепутав и не помешав потоку владельцу правильно добавлять элемент. Здесь есть важнейший момент. **Если мы забираем элемент из связного списка другого потока, то мы забираем его не с «головы», а с «хвоста».** То есть если в связном списке более двух элементов, то поток может своровать элемент без блокировки всего листа. То есть при в таком случае невозможно «состояние гонки», так как между добавляемым и забираемым элементом есть как минимум один промежуточный.



Если же элементов меньше двух, то добавлять просто так нельзя без блокировки, так же если элементов меньше трех, то нельзя забирать элемент без синхронизации. Далее будет показан пример, добавления и изъятия элементов, когда эти операции выполняются на пустых хранилищах, и когда есть два элемента.

Тестирование работы потоков с своим ThreadLocalList

Код для тестирования (вариант 1)

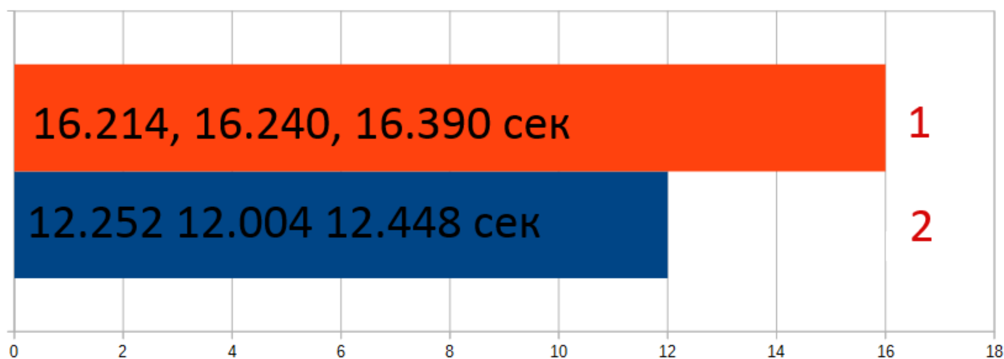
```
Task task1, task2, task3;
ConcurrentBag<int> bagInt = new ConcurrentBag<int>();
int inputSize = 100 * 1024 * 1024;
int[] inputDataInt = new int[inputSize];

for (var i = 0; i < inputSize; i++)
{
    inputDataInt[i] = i;
}

Stopwatch sw = new Stopwatch();
sw.Start();

task1 = Task.Factory.StartNew(() =>
{
    int outInt;
    for (var i = 0; i < inputSize; i++)
    {
        bagInt.Add(inputDataInt[i]);
        bagInt.TryTake(out outInt);
    }
});
task2 = Task.Factory.StartNew(() =>
{
    int outInt;
    for (var i = 0; i < inputSize; i++)
    {
        bagInt.Add(inputDataInt[i]);
        bagInt.TryTake(out outInt);
    }
});
task3 = Task.Factory.StartNew(() =>
{
    int outInt;
    for (var i = 0; i < inputSize; i++)
    {
        bagInt.Add(inputDataInt[i]);
        bagInt.TryTake(out outInt);
    }
});
Task.WaitAll(task1, task2, task3);
sw.Stop();
```

Код для тестирования (вариант 2)

```
Task task1, task2, task3;
ConcurrentBag<int> bagInt = new ConcurrentBag<int>();
int inputSize = 100 * 1024 * 1024;
int[] inputDataInt = new int[inputSize];

for (var i = 0; i < inputSize; i++)
{
    inputDataInt[i] = i;
}

Stopwatch sw = new Stopwatch();
sw.Start();
task1 = Task.Factory.StartNew(() =>
{
    bagInt.Add(-2);
    bagInt.Add(-1);
    int outInt;
    for (var i = 0; i < inputSize; i++)
    {
        bagInt.Add(inputDataInt[i]);
        bagInt.TryTake(out outInt);
    }
});
task2 = Task.Factory.StartNew(() =>
{
    bagInt.Add(-2);
    bagInt.Add(-1);
    int outInt;
    for (var i = 0; i < inputSize; i++)
    {
        bagInt.Add(inputDataInt[i]);
        bagInt.TryTake(out outInt);
    }
});
task3 = Task.Factory.StartNew(() =>
{
    bagInt.Add(-2);
    bagInt.Add(-1);
    int outInt;
    for (var i = 0; i < inputSize; i++)
    {
        bagInt.Add(inputDataInt[i]);
        bagInt.TryTake(out outInt);
    }
});

Task.WaitAll(task1, task2, task3);
sw.Stop();
```

В данном примере три потока, каждый заносит и сразу же забирает элемент n-раз.
В данном примере все потоки будут работать только со своим ThreadLocalList.
Во втором случае, прежде чем выполнять эти операции, мы добавим по два элемента в каждом потоке в локальный список. И получится, что все потоки будут менять размер своего списка с 2 до 3 и обратно.

Массив типа int размера 100 *1024 * 1024.
На пустой коллекции (вариант 1) — 16 секунд,
В локальном хранилище сначала добавлялось два элемента (вариант 2) — 12 секунд.



У ThreadLocalList есть свойство m_currentOp, показывающее текущую операцию, которая выполняется надо коллекцией (None, Add, Take). Однако

во время операции он сбрасывается в None, если количество элементов меньше 2 или 3 на add и take соответственно (тогда производится lock на список).
Когда поток хочет стырить элемент из списка другого потока, он сначала ожидает, пока текущая операция не станет None. Это делается с помощью SpinWait.

```
SpinWait spinner = new SpinWait();
while (list.m_currentOp != (int)ListOperation.None)
{
        spinner.SpinOnce();
}
```

Блокировка на add и take происходит не только, когда количество элементов меньше 2-3, но и тогда, когда поле m_needSync = true. Оно показывает, что произошла блокировка всей коллекции. Когда происходит блокировка всей коллекции, также итеративно накладывается блокировка на все ThreadLocalList всех потоков.

Заключение

Подытоживая, хотелось бы отметить два основных принципа:

1) Каждый поток старается работать только со своей частью хранилища;
2) Даже если поток у себя не находит данные, мы стараемся избежать блокировки локального списка данных другого потока, когда «воруем» данные из него.

На английском Simon Cooper достаточно кратко и хорошо описал все основные принципы в статье
Inside the Concurrent Collections: ConcurrentBag.

Метки: .Net, C#, ConcurrentBag, Concurrent

+35 108 18,8k 4

17,0 Карма 0,0 Рейтинг 0 Подписчики

@RoundGiraffe
Пользователь

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

21 февраля в 13:15
Facebook или Telegram? История украинского .NET Core Community
+17 5,9k 40 6

29 января в 14:42
«…Желают знать, что будет» или пишем гадальный шар в САПР NanoCAD на C# (MultiCAD .NET API)
+10 1,4k 11 0

17 января в 17:17
Еще немного о валидации в ASP.NET
+7 3,9k 43 53

ЗАКАЗЫ ДЛЯ ФРИЛАНСЕРОВ — Фрилансим

Разработка IOS-приложения — Цена договорная
28.02.2018 • 3 отклика

Разработка Android-приложения — Цена договорная
28.02.2018 • 0 откликов

Размещение объявлений на авито — 6000 руб./за проект
28.02.2018 • 2 отклика