Параллельные коллекции

Введение

В версии .NET Framework 4.0 появился набор новых коллекций, определенных в пространстве имен *System.Collections.Concurrent*. Все они полностью безопасны в отношении потоков:

Параллельная коллекция	Непараллельный эквивалент
ConcurrentStack<T>	*Stack<T>*
ConcurrentQueue<T>	*Queue<T>*
ConcurrentBag<T>	(отсутствует)
ConcurrentDictionary<TKey, TValue>	*Dictionary<TKey, TValue>*

Параллельные коллекции оптимизированы для сценариев с высокой степенью параллелизма; тем не менее, они также могут быть полезны в ситуациях, когда требуется коллекция, безопасная к потокам (в качестве альтернативы применению блокировки к обычной коллекции). Однако с параллельными коллекциями связано несколько важных предостережений.

- По производительности традиционные коллекции превосходят параллельные коллекции во всех сценариях кроме тех, которые характеризуются высокой степенью параллелизма.
- Безопасная к потокам коллекция вовсе не гарантирует, что код, в котором она используется, будет безопасным в отношении потоков.
- Если вы производите перечисление параллельной коллекции, в то время как другой поток ее модифицирует, то никаких исключений не возникает – взамен вы получите смесь старого и нового содержимого.
- Параллельной версии *List<T>* не существует.

- Параллельные классы стека, очереди и пакета внутренне реализованы с помощью связных списков. Это делает их менее эффективными в плане потребления памяти, чем непараллельные классы *Stack* и *Queue*, но лучшими для параллельного доступа, т.к. связные списки способствуют построению реализаций с низкой блокировкой или вообще без таковой. (Причина в том, что вставка узла в связный список требует обновления лишь пары ссылок, тогда как вставка элемента в структуру, подобную *List<T>*, может привести к перемещению тысяч существующих элементов.)

Другими словами, параллельные коллекции не являются простыми сокращениями для применения обычных коллекций с блокировками.

Параллельные коллекции также отличаются от традиционных коллекций тем, что они открывают доступ к специальным методам, которые предназначены для выполнения атомарных операций типа “проверить и действовать”, подобных *TryPop*. Большинство таких методов унифицировано посредством интерфейса *IProducerConsumerCollection<T>*.

IProducerConsumerCollection<T>

Коллекция производителей/потребителей является одной из тех, для которых предусмотрены два главных сценария использования:

- добавление элемента (действие “производителя”);
- извлечение элемента с его удалением (действие “потребителя”).

Классическими примерами являются стеки и очереди. Коллекции производителей/потребителей играют важную роль в параллельном программировании, т.к. они способствуют построению эффективных реализаций, свободных от блокировок.

Интерфейс *IProducerConsumerCollection<T>* представляет безопасную к потокам коллекцию производителей/потребителей и реализован следующими классами:

- *ConcurrentStack<T>*
- *ConcurrentQueue<T>*
- *ConcurrentBag<T>*

Интерфейс *IProducerConsumerCollection<T>* расширяет *ICollection*, добавляя перечисленные ниже методы:

```
void CopyTo(T[] array, int index);
T[] ToArray();
bool TryAdd(T item);
bool TryTake(out T item);
```

Методы *TryAdd* и *TryTake* проверяют, может ли быть выполнена операция добавления/удаления, и если может, тогда производят добавление/удаление. Проверка и действие выполняются атомарно, устраняя необходимость в блокировке, к которой пришлось бы прибегнуть в случае традиционной коллекции:

```
int result;
lock (myStack) if (myStack.Count > 0) result = myStack.Pop();
```

Метод *TryTake* возвращает *false*, если коллекция пуста. Метод *TryAdd* всегда выполняется успешно и возвращает *true* в предоставленных трех реализациях. Однако если вы разрабатываете собственную параллельную коллекцию, в которой дубликаты запрещены, то обеспечите возврат методом *TryAdd* значения *false*, когда заданный элемент уже существует (примером может служить реализация параллельного набора).

Конкретный элемент, который *TryTake* удаляет, определяется подклассом:

- в случае стека *TryTake* удаляет элемент, добавленный позже всех других;
- в случае очереди *TryTake* удаляет элемент, добавленный

раньше всех других;
- в случае пакета *TryTake* удаляет любой элемент, который может быть удален наиболее эффективно.

Три конкретных класса главным образом реализуют методы *TryTake* и *TryAdd* явно, делая доступной ту же самую функциональность через открытые методы с более специфичными именами, такими как *TryDequeue* и *TryPop*.

ConcurrentBag<T>

Класс *ConcurrentBag<T>* хранит неупорядоченную коллекцию объектов (с разрешенными дубликатами). Класс *ConcurrentBag<T>* подходит в ситуациях, когда не имеет значения, какой элемент будет получен при вызове *Take* или *TryTake*.

Преимущество *ConcurrentBag<T>* перед параллельной очередью или стеком связано с тем, что метод *Add* пакета не допускает почти никаких состязаний, когда вызывается многими потоками одновременно. В отличие от него вызов *Add* параллельно на очереди или стеке приводит к некоторым состязаниям (хотя и намного меньшим, чем при блокировании непараллельной коллекции). Вызов *Take* на параллельном пакете также очень эффективен – до тех пор, пока каждый поток не извлекает большее количество элементов, чем он добавил с помощью *Add*.

Внутри параллельного пакета каждый поток получает свой закрытый связный список. Элементы добавляются в закрытый список, который принадлежит потоку, вызывающему *Add*, что устраняет состязания. Когда производится перечисление пакета, перечислитель проходит по закрытым спискам всех потоков, выдавая каждый из их элементов по очереди. Когда вызывается метод *Take*, пакет сначала просматривает закрытый список текущего потока. Если в нем имеется хотя бы один элемент, то задача может быть завершена легко и без состязаний. Но если этот список пуст, то пакет должен “позаимствовать” элемент из

закрытого списка другого потока, что потенциально может привести к состязаниям.

Таким образом, чтобы соблюсти точность, вызов *Take* дает элемент, который был добавлен позже других в данном потоке; если в этом потоке элементов нет, тогда *Take* дает последний добавленный элемент в другом потоке, выбранном произвольно.

Параллельные пакеты идеальны, когда параллельная операция на коллекции в основном состоит из добавления элементов посредством *Add* – или когда количество вызовов *Add* и *Take* сбалансировано в рамках потока. Пример первой ситуации приводился ранее во время применения метода *Parallel.ForEach* при реализации параллельной программы проверки орфографии:

```
var misspellings = new ConcurrentBag<Tuple<int, string>>();
Parallel.ForEach(wordsToTest, (word, state, i) =>
{
  if (!wordLookup.Contains(word))
    misspellings.Add(Tuple.Create((int)i, word));
});
```

Параллельный пакет может оказаться неудачным выбором для очереди производителей/потребителей, поскольку элементы добавляются и удаляются разными потоками.

BlockingCollection<T>

В случае вызова метода *TryTake* на любой коллекции производителей/потребителей, рассмотренной ранее:

ConcurrentStack<T>, *ConcurrentQueue<T>*, *ConcurrentBag<T>*,

он возвращает *false*, если коллекция пуста. Иногда в таком сценарии полезнее организовать ожидание, пока элемент не станет доступным. Вместо перегрузки методов *TryTake* для обеспечения такой функциональности (что привело бы к перенасыщению членами после предоставления возможности работы с признаками отмены и тайм-аутами) проектировщики PFX инкапсулировали ее в класс-оболочку по имени

BlockingCollection<T>. Блокирующая коллекция может содержать внутри любую коллекцию, которая реализует интерфейс *IProducerConsumerCollection<T>*, и позволяет получать с помощью метода Take элемент из внутренней коллекции, обеспечивая блокирование, когда доступных элементов нет.

Блокирующая коллекция также позволяет ограничивать общий размер коллекции, блокируя производителя, если этот размер превышен. Коллекция, ограниченная в подобной манере, называется ограниченной блокирующей коллекцией.

Для использования класса *BlockingCollection<T>* необходимо выполнить описанные ниже шаги.

1. Создать экземпляр класса, дополнительно указывая помещаемую внутрь реализацию *IProducerConsumerCollection<T>* и максимальный размер (границу) коллекции.
2. Вызывать метод *Add* или *TryAdd* для добавления элементов во внутреннюю коллекцию.
3. Вызывать метод *Take* или *TryTake* для удаления (потребления) элементов из внутренней коллекции.

Если конструктор вызван без передачи ему коллекции, то автоматически будет создан экземпляр *ConcurrentQueue<T>*. Методы производителя и потребителя позволяют указывать признаки отмены и тайм-ауты. Методы *Add* и *TryAdd* могут блокироваться, если размер коллекции ограничен; методы *Take* и *TryTake* блокируются на время, пока коллекция пуста.

Еще один способ потребления элементов предполагает вызов метода *GetConsumingEnumerable*. Он возвращает (потенциально) бесконечную последовательность, которая выдает элементы по мере того, как они становятся доступными. Чтобы принудительно завершить такую последовательность, необходимо вызвать *CompleteAdding*: этот метод также предотвращает помещение в очередь дальнейших элементов.

Кроме того, класс *BlockingCollection* предоставляет статические методы под названиями *AddToAny* и *TakeFromAny*, которые позволяют добавлять и получать элемент, указывая несколько блокирующих коллекций. Действие затем будет выполнено первой коллекцией, которая способна обслужить данный запрос.

Реализация очереди производителей/потребителей

Очередь производителей/потребителей – структура, полезная как при параллельном программировании, так и в общих сценариях параллелизма. Основные аспекты ее работы:

- Очередь настраивается для описания элементов работы или данных, над которыми выполняется работа.
- Когда задача должна выполниться, она ставится в очередь, а вызывающий код занимается другой работой.
- Один или большее число рабочих потоков функционируют в фоновом режиме, извлекая и запуская элементы из очереди.

Очередь производителей/потребителей обеспечивает точный контроль над тем, сколько рабочих потоков выполняется за раз, что полезно для ограничения эксплуатации не только ЦП, но также и других ресурсов. Скажем, если задачи выполняют интенсивные операции дискового ввода-вывода, то можно ограничить параллелизм, не истощая операционную систему и другие приложения. На протяжении времени жизни очереди можно также динамически добавлять и удалять рабочие потоки. Пул потоков CLR сам представляет собой разновидность очереди производителей/потребителей, которая оптимизирована для кратко выполняющихся заданий с интенсивными вычислениями.

Очередь производителей/потребителей обычно хранит элементы данных, на которых выполняется (одна и та же) задача. Например, элементами данных могут быть имена файлов, а задача может осуществлять шифрование содержимого таких файлов. С другой стороны, применяя делегаты в качестве элементов, можно

построить более универсальную очередь производителей/потребителей, где каждый элемент способен делать все что угодно.

В статье “Parallel Programming” (“Параллельное программирование”) по адресу http://albahari.com/threading показано, как реализовать очередь производителей/потребителей с нуля, используя событие *AutoResetEvent* (а также впоследствии методы *Wait* и *Pulse* класса *Monitor*). Тем не менее, начиная с версии .NET Framework 4.0, написание очереди производителей/потребителей с нуля стало необязательным, т.к. большая часть функциональности предлагается классом *BlockingCollection<T>*. Вот как его задействовать:

```
public class PCQueue : IDisposable
{
  BlockingCollection<Action> _taskQ = new BlockingCollection<Action>();

  public PCQueue(int workerCount)
  {
    // Создать и запустить отдельный объект Task
    // для каждого потребителя:
    for (int i = 0; i < workerCount; i++)
      Task.Factory.StartNew(Consume);
  }

  public void Enqueue(Action action) { _taskQ.Add(action); }

  void Consume()
  {
    // Эта перечисляемая последовательность будет блокироваться
    // когда нет доступных элементов, и заканчиваться, когда вызван
    // метод CompleteAdding.

    foreach (Action action in _taskQ.GetConsumingEnumerable())
      action();     // Выполнить задачу.
  }
  public void Dispose() { _taskQ.CompleteAdding(); }
}
```

Поскольку конструктору *BlockingCollection* ничего не передается, он автоматически создает параллельную очередь. Если бы ему был передан объект *ConcurrentStack*, тогда мы получили бы в итоге стек производителей/потребителей.

Использование задач

Только что написанная очередь производителей/потребителей не является гибкой, т.к. мы не можем отслеживать элементы работы после их помещения в очередь. Очень полезными были бы следующие возможности:

- знать, когда элемент работы завершается (и ожидать его посредством *await*);
- отменять элемент работы;
- элегантно обрабатывать любые исключения, которые сгенерированы тем или иным элементом работы.

Идеальное решение предусматривало бы возможность возвращения методом *Enqueue* какого-то объекта, снабжающего нас описанной выше функциональностью. К счастью, уже существует класс, делающий в точности то, что нам нужно – это *Task*, объект которого можно либо сгенерировать с помощью *TaskCompletionSource*, либо создать напрямую (получив незапущенную или холодную задачу):

```
public class PCQueue : IDisposable
{
  BlockingCollection<Task> _taskQ = new BlockingCollection<Task>();
  public PCQueue(int workerCount)
  {
    // Создать и запустить отдельный объект Task
    // для каждого потребителя:
    for (int i = 0; i < workerCount; i++)
      Task.Factory.StartNew(Consume);
  }
  public Task Enqueue(Action action, CancellationToken cancelToken =
default(CancellationToken))
  {
    var task = new Task(action, cancelToken);
    _taskQ.Add(task);
    return task;
  }
  public Task<TResult> Enqueue<TResult>(Func<TResult> func,
      CancellationToken cancelToken = default(CancellationToken))
  {
    var task = new Task<TResult>(func, cancelToken);
    _taskQ.Add(task);
    return task;
  }
```

```
  void Consume()
  {
    foreach (var task in _taskQ.GetConsumingEnumerable())
      try
      {
        if (!task.IsCanceled) task.RunSynchronously();
      }
      catch (InvalidOperationException) { }  // Условие состязаний
  }
  public void Dispose() { _taskQ.CompleteAdding(); }
}
```

В методе *Enqueue* мы помещаем в очередь и возвращаем вызывающему коду задачу, которая создана, но не запущена.

В методе *Consume* мы запускаем эту задачу синхронно в потоке потребителя. Мы перехватываем исключение *InvalidOperationException*, чтобы обработать маловероятную ситуацию, когда задача будет отменена в промежутке между проверкой, не отменена ли она, и ее запуском.

Ниже показано, как можно применять класс *PCQueue*:

```
// Максимальная степень параллелизма равна 2
var pcQ = new PCQueue(2);
string result = await pcQ.Enqueue(() => "That was easy!");
```

Следовательно, мы имеем все преимущества задач – распространение исключений, возвращаемые значения и возможность отмены – и в то же время обладаем полным контролем над их планированием.