Конкурентные коллекции

Динамические структуры данных пространства `System.Collections` и `System.Collections.Generic` не являются потокобезопасными. Обращение к коллекции нескольких потоков может приводить к проблемам гонки данных: потеря элементов, выход индекса за пределы и аварийное завершение работы приложения.

Рассмотрим многопоточную работу с объектом `Dictionary<TKey, TValue>`. При одновременном инкрементировании элемента с ключом “one” возможно наложение изменений, которое приводит к затиранию изменения одного потока изменением другого потока. При инкрементировании затирание приводит к меньшим значениям счетчика.

```
// Создаем обычный словарь
var dic = new Dictionary<string, int>();
// Параллельно обновляем значение элемента с ключом “one”
Parallel.For(0, 100000, i =>
{
     if (dic.ContainsKey("one"))
          dic["one"]++;
     else
          dic.Add("one", 1);
});
Console.WriteLine("Element \"one\": {0}", dic["one"]);
```

Выполняя этот фрагмент, часто получаем правильный ответ: 100000. Тем не менее, встречаются и 86750, и 56670, и 92030. Возможен вариант возникновения исключения с ошибкой выхода индекса за границы.

Для обеспечения потокобезопасного доступа можно использовать средства синхронизации, рассмотренные выше.

```
var lst = new List<int>();
Parallel.For(0, 100000, i =>
     {
          lock(“sync”)
          {
               lst.Add(i);
          }
     });
```

Более эффективным является применение конкурентных коллекций, обеспечивающие потокобезопасность операций добавления и удаления элементов. Конкурентные коллекции реализованы с применением легковесных средств синхронизации и по возможности избегают блокировок там, где они не нужны.

`ConcurrentQueue`	FIFO-очередь
`ConcurrentStack`	LIFO-стэк
`ConcurrentBag`	Неупорядоченная коллекция
`ConcurrentDictionary`	Словарь
`BlockingCollection`	Ограниченная коллекция

В следующем фрагменте осуществляем многопоточную работу с разделяемым списком типа `ConcurrentBag`:

```
var bag = new ConcurrentBag<int>();
Parallel.For(0, 100000, i =>
{
     bag.Add(i);
});
```

Применение конкурентных коллекций не требует использования дополнительных средств синхронизации.

Обычные коллекции не позволяют изменять объект, который используется в `foreach`-перечислении. Конкурентные коллекции, обеспечивая многопоточный доступ, позволяют добавлять элементы внутри цикла `foreach` при переборе элементов. При этом изменения, вносимые внутри перечисления, не отражаются на текущем перечислителе:

```
var bag = new ConcurrentBag<int>();
for(int i=0; i<10; i++)
     bag.Add(i);

foreach(int k in bag)
{
     bag.Add(k);
     Console.Write(bag.Count + " ");
}
```

В этом фрагменте в `foreach`-цикле осуществляем добавление новых элементов и вывод размера коллекции. Получаем вывод 10 итераций, но размер коллекции при этом увеличивается до 20:

```
11 12 13 14 15 16 17 18 19 20
```

Замена конкурентной коллекции ConcurrentBag на список List привела бы возникновению необработанного исключения.

Эффективность конкурентных коллекций

Конкурентные коллекции спроектированы для применения в многопоточных сценариях и не являются в полной мере эквивалентом обычных коллекций с блокировками. В случае однопроцессорной системы или при использовании конкурентных коллекций в одном потоке их эффективность может быть ниже, чем использование обычных коллекций. Поэтому рекомендуется использовать конкурентные коллекции только в случае многопоточной работы.

Интерфейс IProducerConsumerCollection<T>

Все конкурентные коллекции реализуют интерфейс «производитель-потребитель». Основные операции интерфейса `TryAdd` и `TryTake` проверяют возможность операций записи/извлечения и в случае наличия возможности осуществляют эти действия. Операции осуществляются атомарно, то есть потокобезопасно. Если какой-то поток, убедился в возможности извлечения элемента, то другой поток не сможет вмешаться до завершения первым потоком операции извлечения элемента.

Метод `TryTake` возвращает `false` в случае, если коллекция пуста. Метод `TryAdd` для конкурентных коллекций всегда возвращает `true` и успешно завершает добавление элемента.

Методы `TryTake` для конкурентных стеков и очередей возвращают элементы в определенном порядке – для `ConcurrentStack` получаем последний добавленный элемент, для `ConcurrentQueue` получаем первый добавленный элемент.

ConcurrentBag<T>

Коллекция `ConcurrentBag<T>` предназначена для хранения неупорядоченной коллекции объектов (повторы разрешены). Отсутствие определенного порядка извлечения элементов повышает производительность операции чтения и добавления для `ConcurrentBag`. Добавление элементов в коллекции `ConcurrentStack` или `ConcurrentQueue` в нескольких потоках приводит к дополнительным накладным расходам (неблокирующая синхронизация). Объект `ConcurrentBag` одинаково эффективен при однопоточном добавлении и при многопоточном.

Внутри объекта `ConcurrentBag` содержатся связанные списки для каждого потока. Элементы добавляются в тот список, который ассоциирован с текущим потоком. При извлечении элементов сначала опустошается локальная очередь данного потока. Если в локальной очереди содержатся элементы, то извлечение является максимально эффективным. Если локальная очередь пуста, то поток заимствует элементы из локальных очередей других потоков. Таким образом, извлечение элементов из `ConcurrentBag` осуществляется по принципу `LIFO` с учетом локальности.

BlockingCollection<T>

Объект `BlockingCollection` позволяет сформировать модифицированную конкурентную коллекцию на базе `ConcurrentStack`, `ConcurrentQueue` или `ConcurrentBag`. Модификации конкурентных коллекций применяются для реализации шаблона «производитель-потребитель». Метод `Take` для `BlockingCollection` вызывается потоком–потребителем и приводит к блокировке потока в случае отсутствия элементов. Добавление элементов потоком-производителем приводит к разблокировке (освобождению) потребителя.

При создании коллекции существует возможность установления максимальной емкости. Если коллекция полностью заполнена, то операция добавления элемента приводит к блокировке текущего потока до тех пор, пока поток-потребитель не извлечет, хотя бы один элемент.

Метод `CompleteAdding` позволяет завершить добавление элементов в коллекцию. Обращения к операции `Add` будут приводить к исключениям. Свойство `IsAddingCompleted` позволяет проверить статус завершения. Извлечение элементов из «завершенной» коллекции разрешено, пока коллекция не пуста. Операция `Take` для пустой завершенной коллекции приводит к генерации исключения. Свойство `IsCompleted` позволяет установить, является ли коллекция пустой и завершенной одновременно.

Тип коллекции, которая используется для формирования `BlockingCollection`, определяет порядок извлекаемых элементов. Если объект `BlockingCollection` создается без указания конкурентной коллекции, то в качестве базовой коллекции используется очередь типа `ConcurrentQueue`.

ConcurrentDictionary

Конкурентный словарь кроме потокобезопасности добавления и удаления элементов предоставляет расширенные функциональные возможности: методы условного добавления и методы обновления значений.

```
// Создаем конкурентный словарь
var cd = new ConcurrentDictionary<string, int>();
// Хотим получить элемент с ключом "a", если нет - создаем
int value = cd.GetOrAdd("a", (key) => 1000);

// Параллельно пытаемся обновить элемент с  ключом "a"
Parallel.For(0, 1000, i =>
{
     // Если ключа нет – добавляем
     // Если есть – обновляем значение
     cd.AddOrUpdate("a", 1, (key, oldValue) => oldValue + 1);
});
Console.WriteLine("Element \"a\": {0}", cd["a"]);
```

Метод GetOrAdd реализует возможность получения значения по ключу или в случае остутствия элемента добавления.

```
if(dic.ContainsKey(sKey))
     val = dic[sKey];
else
     dic.Add(sKey, sNewValue);
```

Но метод GetOrAdd выполняется атомарно в отличие от приведенного фрагмента, то есть несколько потоков не могут одновременно проверить наличие элемента и осуществить добавление.