Модель памяти C# в теории и на практике

Это первая из двух статей, в которых пойдет долгое повествование о модели памяти в C#. В первой части поясняются гарантии модели памяти C# и показываются шаблоны кода, которыми обусловлены эти гарантии; во второй части будет подробно рассмотрено, как данные гарантии достигаются на разных аппаратных архитектурах в Microsoft .NET Framework 4.5.

Одна из причин сложности многопоточного программирования в том, что компилятор и аппаратное обеспечение могут слегка трансформировать операции программы с памятью такими способами, которые не влияют на однопоточное поведение, но могут затронуть многопоточное. Рассмотрим следующий метод:

```
void Init() {
  _data = 42;
  _initialized = true;
}
```

Если _data и _initialized — обычные (т. е. неизменяемые) поля, компилятору и процессору разрешается такое переупорядочение операций, чтобы Init выполнялся так, будто он написан следующим образом:

```
void Init() {
  _initialized = true;
  _data = 42;
}
```

Существуют различные оптимизации как в компиляторах, так и в процессорах, которые могут привести к такому переупорядочению, и это будет обсуждаться во второй статье.

В однопоточной программе переупорядочение выражений в Init ничего не меняет в программе. Пока и _initialized и _data обновляются до возврата управления этим методом, порядок присваиваний не имеет значения. В однопоточной программе нет второго потока, который мог бы наблюдать состояние между этими обновлениями.

Даже если компилятору и процессору разрешено переупорядочивать операции с памятью, это не означает, что на практике они всегда так делают.

Но в многопоточной программе порядок присваиваний может иметь значение, поскольку другой поток может считывать эти поля, пока Init находится в середине выполнения. Соответственно в переупорядоченной версии Init другой поток может наблюдать _initialized=true и _data=0.

Модель памяти C# — это набор правил, описывающих, какие виды переупорядочения операций с памятью разрешены, а какие — нет. Все программы должны быть написаны в соответствии с гарантиями, определенными в спецификации.

Однако, даже если компилятору и процессору разрешено переупорядочивать операции с памятью, это не означает, что на практике они всегда так делают. Многие программы, которые содержат «ошибку» согласно абстрактной модели памяти C#, будут по-прежнему корректно выполняться на конкретном аппаратном обеспечении, где работает определенная версия .NET Framework. В частности, процессоры x86 и x64 переупорядочивают операции лишь в некоторых сценариях весьма узкого применения, и аналогично JIT-компилятор в CLR не выполняет многие трансформации, которые ему разрешены.

Модель памяти C# разрешает переупорядочение операций в каком-либо методе, только если поведение при однопоточном выполнении не меняется.

Хотя абстрактная модель памяти C# — это то, что вы учитывать при написании нового кода, понимание истинной реализации модели памяти в различных аппаратных архитектурах может оказаться весьма полезным, в частности если вы пытаетесь разобраться в поведении существующего кода.

Модель памяти C# согласно ECMA-334

Авторитетное определение модели памяти C# дано в Standard ECMA-334 C# Language Specification (bit.ly/MXMCrN). Давайте обсудим эту модель в том виде, как она определена в данной спецификации.

Переупорядочение операций с памятью Согласно ECMA-334, когда поток считывает в C# участок памяти, записанный другим потоком, «читатель» может увидеть устаревшее значение. Эту проблему иллюстрирует **рис. 1**.

Рис. 1. Код, подверженный риску переупорядочения операций с памятью

```
public class DataInit {
  private int _data = 0;
```

```
  private bool _initialized = false;
  void Init() {
    _data = 42;            // запись 1
    _initialized = true;   // запись 2
  }
  void Print() {
    if (_initialized)           // чтение 1
      Console.WriteLine(_data); // чтение 2
    else
      Console.WriteLine("Not initialized");
  }
}
```

Допустим, что Init и Print вызываются параллельно (т. е. в разных потоках) в новом экземпляре DataInit. Если вы посмотрите код Init и Print, вам может показаться, что Print может выводить только «42» или «Not initialized». Но Print также может вывести «0».

Модель памяти C# разрешает переупорядочение операций в каком-либо методе, только если поведение при однопоточном выполнении не меняется. Например, компилятор и процессор могут переупорядочить операции метода Init так:

```
void Init() {
  _initialized = true;   // запись 2
  _data = 42;            // запись 1
}
```

Это переупорядочение не изменило бы поведение метода Init в однопоточной программе. Однако в многопоточной программе другой поток мог бы считать значения полей _initialized и _data после того, как Init модифицировал одно поле, но не успел сделать это со вторым, а затем последующее переупорядочение может изменить поведение программы. В результате метод Print мог бы вывести «0».

Переупорядочение Init — не единственный возможный источник проблем в этом примере кода. Даже если операции записи в Init не переупорядочиваются, операции чтения в методе Print могут быть трансформированы:

```
void Print() {
  int d = _data;     // чтение 2
  if (_initialized)  // чтение 1
    Console.WriteLine(d);
  else
    Console.WriteLine("Not initialized");
}
```

Как и в случае переупорядочения операций записи, эта трансформация никак не влияет на однопоточную программу, но может изменить поведение многопоточной. И равным образом переупорядочение операций чтения тоже может дать значение 0 в выводе.

Во второй части вы увидите, как и почему эти трансформации происходят на практике, когда мы будет подробно рассматривать различные аппаратные архитектуры.

Изменяемые поля Язык программирования C# предоставляет изменяемые поля (volatile fields), которые ограничивают то, как могут быть переупорядочены операции с памятью. В спецификации ECMA утверждается, что изменяемые поля предоставляют семантику получения-освобождения (acquirerelease) (bit.ly/NArSlt).

Чтение изменяемого поля имеет семантику получения, т. е. эта операция не может быть переупорядочена с последующими операциями. Чтение изменяемого поля образует одностороннюю преграду: предшествующие операции могут проникать через нее, а последующие — нет. Возьмем такой пример:

```
class AcquireSemanticsExample {
  int _a;
  volatile int _b;
  int _c;
  void Foo() {
    int a = _a; // чтение 1
    int b = _b; // чтение 2 (изменяемое поле)
    int c = _c; // чтение 3
    ...
  }
}
```

Чтение 1 и чтение 3 — операции с неизменяемыми полями, а чтение 2 — операция с изменяемым полем. Чтение 2 нельзя переупорядочить с чтением 3, но можно — с чтением 1. В **табл. 1** показаны допустимые переупорядочения в теле Foo.

Табл. 1. Допустимое переупорядочение операций чтения в AcquireSemanticsExample

int a = _a; // чтение 1
int b = _b; // чтение 2 (изменяемое поле)
int c = _c; // чтение 3
	int b = _b; // чтение 2 (изменяемое поле)
	int a = _a; // чтение 1
	int c = _c; // чтение 3
		int b = _b; // чтение 2 (изменяемое поле)
		int c = _c; // чтение 3
		int a = _a; // чтение 1

С другой стороны, операция записи в изменяемое поле имеет семантику

освобождения, и поэтому ее нельзя переупорядочить с предыдущими операциями. Запись изменяемого поля образует одностороннюю преграду, как демонстрирует следующий пример:

```
class ReleaseSemanticsExample
{
  int _a;
  volatile int _b;
  int _c;
  void Foo()
  {
    _a = 1; // запись 1
    _b = 1; // запись 2 (изменяемое поле)
    _c = 1; // запись 3
    ...
  }
}
```

Записи 1 и 3 — операции с неизменяемыми полями, а запись 2 — операция с изменяемым полем. Запись 2 нельзя переупорядочить с записью 1, но можно — с записью 3. В **табл. 2** показаны допустимые переупорядочения в теле Foo.

Табл. 2. Допустимое переупорядочение операций записи в ReleaseSemanticsExample

_a = 1; // запись 1	_a = 1; // запись 1	_c = 1; // запись 3
_b = 1; // запись 2 (изменяемое поле)	_c = 1; // запись 3	_a = 1; // запись 1
_c = 1; // запись 3	_b = 1; // запись 2 (изменяемое поле)	_b = 1; // запись 2 (изменяемое поле)

Я вернусь к семантике получения-освобождения в подразделе «Публикация через изменяемое поле» далее в этой статье.

Атомарность Другая проблема C#, о которой нужно знать, заключается в том, что значения не обязательно записываются в память атомарно. Рассмотрим этот пример:

```
class AtomicityExample {
  Guid _value;
  void SetValue(Guid value) { _value = value; }
  Guid GetValue() { return _value; }
}
```

Если один поток повторно вызывает SetValue, а другой вызывает GetValue, то второй поток может наблюдать значение, которое никогда не записывалось первым потоком. Например, если первый поток попеременно вызывает SetValue со значениями Guid (0,0,0,0) и (5,5,5,5), то GetValue может наблюдать (0,0,0,5), (0,0,5,5) или (5,5,0,0), хотя ни одно из этих значений никогда не присваивалось через SetValue.

Причина такого «разрыва» в том, что присваивание _value = value не выполняется атомарно на аппаратном уровне. Аналогично чтение _value тоже не выполняется атомарно.

Спецификация ECMA по C# гарантирует, что следующие типы будут записываться атомарно: ссылочные типы, bool, char, byte, sbyte, short, ushort, uint, int и float. Значения других типов, включая пользовательские значимые типы, могут помещаться в память набором атомарных операций записи. В итоге поток-«читатель» мог бы наблюдать рваные значения, состоящие из частей различных значений.

Один из подвохов в том, что типы, которые обычно считываются и записываются атомарно (например, int), могут считываться и записываться не атомарно, если значение неправильно выровнено в памяти. В нормальных условиях C# гарантирует, что значения выравниваются в памяти правильно, но пользователь может переопределить это выравнивание с помощью класса StructLayoutAttribute (bit.ly/Tqa0MZ).

Оптимизации без переупорядочения Некоторые оптимизации компилятора могут вводить или исключать определенные операции с памятью. Так, компилятор может заменить повторяемые операции чтения какого-либо поля одним чтением. Аналогично, если код считывает поле и сохраняет значение в локальной переменной, а затем повторно считывает эту переменную, то компилятор мог бы вместо этого выбрать повторное чтение поля.

Поскольку ECMA-спецификация C# не исключает оптимизации без упорядочения, они предположительно разрешены. По сути, как будет рассматриваться во второй части, JIT-компилятор действительно выполняет оптимизации этих типов.

Один из подвохов в том, что типы, которые обычно считываются и записываются атомарно (например, int), могут считываться и записываться не атомарно, если значение неправильно выровнено в памяти.

Шаблоны взаимодействия потоков

Цель модели памяти — обеспечить взаимодействие потоков. Когда один из потоков записывает значения в память, а другой — считывает из памяти, модель памяти диктует, какие значения может увидеть читающий поток.

Блокировка Это самый простой способ совместного использования общих данных между потоками. При правильном использовании блокировок вам, в общем, не нужно беспокоиться о дополнительном исследовании модели памяти.

Всякий раз, когда поток захватывает блокировку, CLR гарантирует, что этот поток увидит все обновления, выполненные тем потоком, который владел блокировкой ранее. Добавим блокировку к примеру из начала этой статьи, как показано на **рис. 2**.

Рис. 2. Взаимодействие потоков с блокировкой

```
public class Test {
  private int _a = 0;

  private int _b = 0;
  private object _lock = new object();
  void Set() {
    lock (_lock) {
      _a = 1;
      _b = 1;
    }
  }
  void Print() {
    lock (_lock) {
      int b = _b;
      int a = _a;
      Console.WriteLine("{0} {1}", a, b);
    }
  }
}
```

Добавление блокировки, которую поочередно захватывают Print и Set, дает простое решение. Теперь Set и Print выполняются атомарно по отношению друг к другу. Выражение lock гарантирует, что тела Print и Set будут выполняться в некоем последовательном порядке, даже если они вызываются из нескольких потоков.

Схема на **рис. 3** показывает один из вариантов последовательного порядка, который мог бы наблюдаться, если бы поток 1 вызвал Print три раза, поток 2 вызвал Set один раз, а поток 3 вызвал Print один раз.

Блокировка — универсальный и мощный механизм разделения общего состояния между потоками.

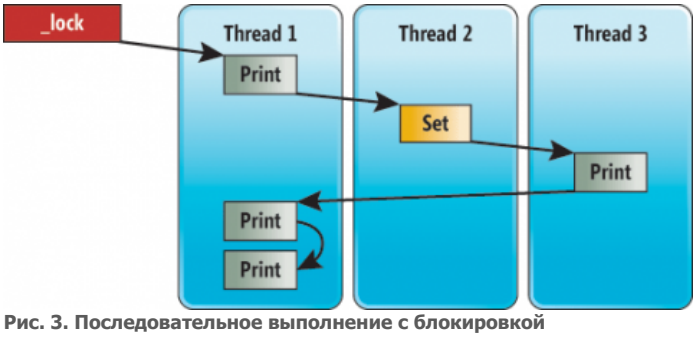

Рис. 3. Последовательное выполнение с блокировкой

_lock	_lock
Thread 1	Поток 1
Thread 2	Поток 2
Thread 3	Поток 3
Print	Print
Set	Set

Когда выполняется порция кода, владеющая блокировкой, она гарантированно видит результаты всех предшествующих в последовательном порядке операций записи. Кроме того, она гарантированно не видит результаты любых операций записи, выполняемых теми порциями кода, которые следуют за блокировкой.

Если в двух словах, то блокировки исключают всю непредсказуемость и сложность модели памяти: вам не надо беспокоиться о переупорядочении операций с памятью при правильном использовании блокировок. Но именно при правильном использовании. Если блокировку использует только Print или только Set (либо Print и Set захватывают две разные блокировки), операции с памятью становятся переупорядочиваемыми и сложность модели памяти вновь возвращается.

Публикация через API потоков Блокировка — универсальный и мощный механизм разделения общего состояния между потоками. Публикация через API потоков является еще одним часто применяемым шаблоном в программировании параллельной обработки.

Публикацию через API потоков легче всего пояснить на примере:

```
class Test2 {
  static int s_value;
  static void Run() {
    s_value = 42;
    Task t = Task.Factory.StartNew(() => {
      Console.WriteLine(s_value);
    });
    t.Wait();
  }
}
```

Изучив предыдущий пример кода, вы, вероятно, ожидали, что на экран будет выведено «42». И интуиция вас не подвела. Этот пример кода гарантированно

выводит «42».

Может быть, это удивительно, что об этом вообще приходится упоминать, но на деле возможны реализации StartNew, которые выводили бы «0» вместо «42», по крайней мере теоретически. В конце концов, два потока взаимодействуют через неизменяемое поле, поэтому операции с памятью могут быть переупорядочены. Этот шаблон показан на схеме на **рис. 4**.

Рис. 4. Два потока, взаимодействующие через неизменяемое поле

Initial State	Начальное состояние
Thread 1	Поток 1
Thread 2	Поток 2

Реализация StartNew должна гарантировать, что запись в s_value в потоке 1 не будет перемещена за <start task t>, а чтение из s_value в потоке 2 не будет перемещено до <task t starting>. И действительно StartNew API это гарантирует.

Все остальные API потоков в .NET Framework, такие как Thread.Start и ThreadPool.QueueUserWorkItem, тоже дают аналогичные гарантии. По сути, почти каждый API потоков должен иметь некую семантику барьера, чтобы правильно работать. Это почти никогда не документируется, но обычно поддается логическому вычислению, если просто поразмыслить о том, какие гарантии должны быть у конкретного API, чтобы от него была польза.

Публикация через инициализацию типа Другой способ надежной публикации некоего значения нескольким потокам — его запись в статическое поле в статическом инициализаторе или в статическом конструкторе. Возьмем пример:

```
class Test3
{
  static int s_value = 42;
  static object s_obj = new object();
  static void PrintValue()
  {
    Console.WriteLine(s_value);
    Console.WriteLine(s_obj == null);
  }
}
```

Если Test3.PrintValue одновременно вызывается из нескольких потоков, гарантируется ли, что каждый вызов PrintValue выведет «42» и «false»? Или же один из вызовов приведет к выводу «0» или «true»? Как и в предыдущем случае, вы получаете именно то поведение, которое ожидаете: каждый поток гарантированно выводит «42» и «false».

Обсуждавшиеся до сих пор шаблоны ведут себя ожидаемым образом. Теперь перейдем к случаям, где поведение может оказаться неожиданным.

Публикация через изменяемое поле Многие параллельные программы можно создать на основе уже рассмотренных трех простых шаблонов в сочетании с параллельными примитивами в .NET-пространствах имен System.Threading и System.Collections.Concurrent.

Шаблон, который я намерен обсудить, столь важен, что для него была разработана семантика ключевого слова volatile. По сути, лучший способ запомнить семантику ключевого слова volatile — запомнить этот шаблон, а не пытаться зазубривать абстрактные правила, пояснявшиеся ранее в этой статье.

Начнем с примера кода на **рис. 5**. В классе DataInit на **рис. 5** два метода: Init и Print; оба могут быть вызваны из нескольких потоков. Если никакие операции с памятью не переупорядочиваются, Print может вывести только «Not initialized» или «42», но есть два возможных случая, когда Print мог бы вывести «0»:

- операции записи 1 и 2 были переупорядочены;
- операции чтения 1 и 2 были переупорядочены.

Рис. 5. Использование ключевого слова volatile

```
public class DataInit {
  private int _data = 0;
  private volatile bool _initialized = false;
  void Init() {
    _data = 42;            // запись 1
    _initialized = true;   // запись 2
  }
  void Print() {
    if (_initialized) {          // чтение 1
      Console.WriteLine( data);  // чтение 2
```

```
    }
    else {
      Console.WriteLine("Not initialized");
    }
  }
}
```

Если бы _initialized не была помечена как volatile, оба переупорядочения были бы разрешены. Однако, когда _initialized помечена как volatile, ни одно из этих переупорядочений не разрешено! В случае записи вы получаете обычную запись, за которой следует запись в изменяемое поле, а последнюю нельзя переупорядочить с предыдущей операцией в памяти. В случае чтения операция чтения из изменяемого поля сменяется обычной операцией чтения, а первую нельзя переупорядочить с последующей операцией в памяти.

Поэтому Print никогда не выведет «0», даже если она будет вызвана одновременно с Init в новом экземпляре DataInit.

Заметьте: если бы поле _data field было изменяемым, а _initialized — нет, оба переупорядочения были бы разрешены. В итоге этот пример является отличным способом запомнить семантику ключевого слова volatile.

Отложенная инициализация Одна из распространенных вариаций публикации через изменяемое поле — отложенная инициализация (lazy initialization) (**рис. 6**).

Рис. 6. Отложенная инициализация

```
class BoxedInt
{
  public int Value { get; set; }
}
class LazyInit
{
  volatile BoxedInt _box;
  public int LazyGet()
  {
    var b = _box;  // чтение 1
    if (b == null)
    {
      lock(this)
      {
        b = new BoxedInt();
        b.Value = 42; // запись 1
        _box = b;     // запись 2
      }
    }
    return b.Value; // чтение 2
  }
}
```

В этом примере LazyGet всегда гарантированно возвращает «42». Однако, если бы поле _box field не было помечено как volatile, LazyGet могла бы вернуть «0» по двум причинам: могли бы быть переупорядочены либо операции чтения, либо операции записи.

Чтобы еще больше акцентировать на этом ваше внимание, рассмотрим такой класс:

```
class BoxedInt2
{
  public readonly int _value = 42;
  void PrintValue()
  {
    Console.WriteLine(_value);
  }
}
```

Теперь возможно (по крайней мере, теоретически), что PrintValue выведет «0» из-за проблемы с моделью памятью. Вот пример использования BoxedInt, где такое разрешается:

```
class Tester
{
  BoxedInt2 _box = null;
  public void Set() {
    _box = new BoxedInt2();
  }
  public void Print() {
    var b = _box;
    if (b != null) b.PrintValue();
  }
}
```

Так как экземпляр BoxedInt был опубликован неправильно (через неизменяемое поле _box), поток, который вызывает Print, может наблюдать частично сконструированный объект! И вновь, сделав поле _box изменяемым, вы устраните проблему.

Interlocked-операции и барьеры памяти Interlocked-операции являются атомарными и иногда используются для уменьшения блокировок в многопоточных программах. Рассмотрим простой класс-счетчик, безопасный в многопоточной среде:

```
class Counter
{
  private int _value = 0;
```

```
  private object _lock = new object();
  public int Increment()
  {
    lock (_lock)
    {
      _value++;
      return _value;
    }
  }
}
```

Используя Interlocked.Increment, вы можете переписать программу так:

```
class Counter
{
  private int _value = 0;
  public int Increment()
  {
    return Interlocked.Increment(ref _value);
  }
}
```

При использовании Interlocked.Increment данный метод должен выполняться быстрее, по крайней мере в некоторых аппаратных архитектурах. В дополнение к операциям приращения класс Interlocked (bit.ly/RksCMF) предоставляет методы для различных атомарных операций: добавления значения, замены значения по условию, замены значения и возврата исходного значения и т. д.

Все Interlocked-методы имеют одно очень интересное свойство: их нельзя переупорядочивать с другими операциями с памятью.

Все Interlocked-методы имеют одно очень интересное свойство: их нельзя переупорядочивать с другими операциями с памятью. Операция, тесно связанная с Interlocked-методами, — Thread.MemoryBarrier, которую можно рассматривать как пустую Interlocked-операцию. Так же, как Interlocked-метод, Thread.MemoryBarrier нельзя переупорядочить с любой предыдущей или последующей операцией с памятью. Однако в отличие от Interlocked-метода Thread.MemoryBarrier не имеет побочного эффекта; он просто ограничивает переупорядочения.

Цикл опроса Это шаблон, который, как правило, не рекомендуется, но, к сожалению, часто используется на практике. Неправильный цикл опроса показан на **рис. 7**.

Рис. 7. Неправильный цикл опроса

```
class PollingLoopExample
{
  private bool _loop = true;
  public static void Main()
  {
    PollingLoopExample test1 = new PollingLoopExample();
    // Задаем _loop как false в другом потоке
    new Thread(() => { test1._loop = false;}).Start();
    // Опрашиваем поле _loop, пока оно не станет false
    while (test1._loop) ;
    // Предыдущий цикл никогда не завершится
  }
}
```

В этом примере основной поток крутится в цикле, опрашивая конкретное неизменяемое поле. Вспомогательный поток тем временем присваивает значение этому полю, но основной поток никогда не увидит измененное значение.

А если бы поле _loop было помечено как volatile? Исправило бы это ситуацию? По общему согласию экспертов, компилятору не разрешается вытаскивать чтение изменяемого поля из цикла, но вопрос о том, дает ли спецификация ECMA на C# такую гарантию, остается открытым.

Рекомендации

- Весь код, который вы пишете, должен полагаться только на гарантии спецификации ECMA C# и не использовать никакие детали реализации, пояснявшиеся в этой статье.
- Избегайте ненужного использования изменяемых полей. По большей части блокировки или параллельные наборы (System.Collections.Concurrent.*) лучше подходят для обмена данными между потоками. В некоторых случаях изменяемые поля можно использовать для оптимизации параллельного кода, но вы должны измерять производительность, чтобы убедиться в том, что выигрыш перевешивает дополнительную сложность.
- Вместо самостоятельной реализации шаблона отложенной инициализации с применением volatile-поля используйте типы System.Lazy<T> и System.Threading.LazyInitializer.
- Избегайте циклов опроса. Зачастую вместо таких циклов можно использовать BlockingCollection<T>, Monitor.Wait/Pulse, события или асинхронное программирование.
- По возможности используйте стандартные параллельные примитивы .NET вместо самостоятельной реализации эквивалентной функциональности.

С одной стороны, спецификация утверждает, что только изменяемые поля подчиняются семантике получения-освобождения (acquire-release semantics),

чего вроде бы недостаточно для предотвращения изъятия операции чтения изменяемого поля из цикла опроса. С другой — пример кода в этой спецификации действительно опрашивает изменяемое поле, а это подразумевает, что операция чтения изменяемого поля не может быть изъята из данного цикла.

На аппаратных платформах x86 и x64 PollingLoopExample.Main будет, как правило, зависать. JIT-компилятор считает поле test1._loop только один раз, сохранит его значение в одном из регистров, а затем будет крутиться в цикле, пока значение в этом регистре не изменится, чего не произойдет никогда.

Однако, если тело цикла содержит некоторые выражения, JIT-компилятору, возможно, потребуется тот же регистр для других целей, поэтому каждая итерация может приводить к повторному чтению test1._loop. В итоге вы можете столкнуться с циклами в существующих программах, которые опрашивают неизменяемое поле и ухитряются нормально работать.

Параллельные примитивы Большая часть параллельного кода может выиграть от применения высокоуровневых параллельных примитивов, которые появились в .NET Framework 4. В **табл. 3** перечислены некоторые из таких .NET-примитивов.

Табл. 3. Параллельные примитивы в .NET Framework 4

Тип	Описание
Lazy<> LazyInitializer	Значения с отложенной инициализацией
BlockingCollection<> ConcurrentBag<> ConcurrentDictionary<,> ConcurrentQueue<> ConcurrentStack<>	Наборы, безопасные в многопоточной среде
AutoResetEvent Barrier CountdownEvent ManualResetEventSlim Monitor SemaphoreSlim	Примитивы для координации выполнения в разных потоках
ThreadLocal<>	Контейнер, хранящий отдельное значение для каждого потока

Используя эти примитивы, зачастую можно избежать низкоуровневого кода, зависимого от модели памяти весьма сложным для понимания образом (через volatile и прочее).

В следующей части

На данный момент я описал модель памяти C# так, как она определена в спецификации ECMA C#, и рассмотрел наиболее важные шаблоны взаимодействия потоков, определяющих модель памяти.

Во второй части этой статьи я поясню, как модель памяти на деле реализуется в различных аппаратных архитектурах, что поможет в понимании поведения настоящих программ.

Автор: *Игорь Островский* • ***Источник:*** *MSDN Magazine* • ***Опубликована:*** *07.03.2013*

Нашли ошибку в тексте? Сообщите о ней автору: выделите мышкой и нажмите CTRL + ENTER

Похожие материалы раздела

- Модель памяти C# в теории и на практике. Часть 2

Теги:

Оценить статью: Ужасно Плохо Средне Хорошо Отлично

Комментарии посетителей

Комментарии отключены. С вопросами по статьям обращайтесь в форум.