Логин
Запомнить ВОЙТИ
Регистрация • Забытый пароль

НОВОСТИ
- Microsoft
- Hardware
- IT

MICROSOFT
- Видео
- Разработка приложений
- Windows 10
- Windows Server 2016
- все подразделы

ЖЕЛЕЗО
- Процессоры
- Материнские платы
- Видео
- Память
- все подразделы

ПРОГРАММЫ
- Обзоры программ
- FAQ
- Microsoft
- Система
- все подразделы

СУБД
- MS SQL
- Oracle
- MySQL
- Общие вопросы
- все подразделы

ОБЛАКО ТЕГОВ

OSzone.net ▸ Microsoft ▸ Разработка приложений ▸ Языки программирования ▸ Модель памяти C# в теории и на практике. Часть 2 RSS

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

Это вторая часть статьи, в которой обсуждается модель памяти C#. Как пояснялось в первой части , 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;
}
```

В предыдущей статье я описал абстрактную модель памяти C#. В этой статье я расскажу, как на самом деле реализуется модель памяти C# на различных аппаратных архитектурах, поддерживаемых Microsoft .NET Framework 4.5.

Оптимизации компилятора

Как упоминалось в первой статье, компилятор может оптимизировать код такими способами, которые приводят к переупорядочению операций с памятью. В .NET Framework 4.5 компилятор csc.exe, транслирующий C# в IL, не выполняет много работы, поэтому он не переупорядочивает такие операции. Однако JIT-компилятор, преобразующий IL в машинный код, действительно выполняет некоторые оптимизации, которые переупорядочивают операции с памятью.

Выделение операции чтения из цикла (loop read hoisting) Рассмотрим шаблон цикла опроса:

```
class Test
{
  private bool _flag = true;
  public void Run()
  {
    // Задаем _flag как false в другом потоке
    new Thread(() => { _flag = false; }).Start();
    // Опрашиваем поле _flag, пока оно не станет равным false
    while (_flag) ;
    // Цикл может никогда не завершиться!
  }
}
```

В этом случае JIT-компилятор в .NET 4.5 может переписать цикл так:

```
if (_flag) { while (true); }
```

В однопоточной программе это преобразование совершенно корректно и, в целом, выделение операции чтения из цикла — отличная оптимизация. Однако, если _flag задается как false в другом потоке, эта оптимизация может вызвать зависание.

Заметьте, что, если бы поле _flag было изменяемым, JIT-компилятор не стал бы выделять чтение из цикла. (Детальные пояснения по этому шаблону см. в разделе «Цикл опроса» в первой части этой статьи.)

Исключение операции чтения (read elimination) Другая оптимизация компилятора, способная вызывать ошибки в многопоточном коде, иллюстрируется следующим примером:

```
class Test
{
  private int _A, _B;
  public void Foo()
  {
    int a = _A;
    int b = _B;
    ...
```

```
  }
}
```

Класс содержит два неизменяемых поля: _A и _B. Метод Foo сначала считывает поле _A, потом поле _B. Однако, поскольку эти поля неизменяемые, компилятор свободен в переупорядочении этих двух операций чтения. Поэтому, если корректность алгоритма зависит от порядка операций чтения, в программе появляется ошибка.

Трудно вообразить, какого выигрыша мог бы добиться компилятор изменением порядка операций чтения. Учитывая то, как написан Foo, компилятор, по-видимому, не стал бы менять порядок операций чтения.

Но переупорядочение все же происходит, если в начало метода Foo я добавляю другое, вполне невинное выражение:

```
public bool Foo()
{
  if (_B == -1) throw new Exception(); // дополнительное чтен
  int a = _A;
  int b = _B;
  return a > b;
}
```

На первой строке метода Foo компилятор загружает в регистр значение _B. Потом при второй загрузке _B просто использует значение, уже находящееся в регистре, вместо генерации команды настоящей загрузки.

В конечном счете компилятор переписывает метод Foo таким образом:

```
public bool Foo()
{
  int b = _B;
  if (b == -1) throw new Exception(); // дополнительное чтени
  int a = _A;
  return a > b;
}
```

Хотя этот пример дает весьма приблизительное представление о том, как компилятор оптимизирует код, поучительно посмотреть и на дизассемблированный код:

```
if (_B == -1) throw new Exception();
  push        eax
  mov         edx,dword ptr [ecx+8]
  // Загрузка поля _B в регистр EDX
  cmp         edx,0FFFFFFFFh
  je          00000016
int a = _A;
  mov         eax,dword ptr [ecx+4]
  // Загрузка поля _A в регистр EAX
return a > b;
  cmp         eax,edx
  // Сравнение регистров EAX и EDX
...
```

Даже если вы не знаете языка ассемблера, то, что здесь происходит, понять несложно. При оценке условия _B == –1 компилятор загружает поле _B в регистр EDX. Позднее, когда поле _B снова считывается, компилятор просто использует повторно значение, которое уже находится в регистре EDX, вместо выдачи команды на реальное чтение памяти. Соответственно операции чтения _A и _B переупорядочиваются.

В этом случае правильным решением будет пометить поле _A как volatile (изменяемое). Тогда компилятор не должен переупорядочивать операции чтения _A и _B, так как загрузка _A получит семантику загрузки-получения (load-acquire). Однако следует отметить, что .NET Framework до версии 4 включительно не обрабатывает этот случай корректно и, по сути, задание поля _A как volatile не предотвратит переупорядочение операций чтения. Эта проблема была исправлена в .NET Framework 4.5.

Введение дополнительного чтения (read introduction) Как я уже пояснял, компилятор иногда объединяет несколько операций чтения в одну. Кроме того, компилятор может разделить одну операцию чтения на несколько операций. В .NET Framework 4.5 введение дополнительного чтения встречается намного реже, чем объединение, и возможно лишь в очень специфических обстоятельствах. Тем не менее, иногда такое бывает.

Чтобы понять введение дополнительного чтения, рассмотрим следующий пример:

```
public class ReadIntro {
  private Object _obj = new Object();
  void PrintObj() {
    Object obj = _obj;
    if (obj != null) {
      Console.WriteLine(obj.ToString());
    // Возможна генерация NullReferenceException
    }
  }
  void Uninitialize() {
    _obj = null;
  }
}
```

При рассмотрении метода PrintObj может показаться, что значение obj никогда не будет null в выражении obj.ToString. Однако эта строка кода на самом деле могла бы привести к генерации исключения NullReferenceException. CLR JIT может скомпилировать метод PrintObj так, будто он написан следующим образом:

```
void PrintObj() {
  if (_obj != null) {
    Console.WriteLine(_obj.ToString());
  }
}
```

Поскольку чтение поля _obj было разбито на две операции чтения этого поля, метод ToString теперь может быть вызван для null.

Заметьте, что вам не удастся воспроизвести NullReferenceException, используя этот код в .NET Framework 4.5 на x86 или x64. Введение дополнительного чтения очень трудно воспроизвести в .NET Framework 4.5, но в определенных условиях оно все же происходит.

Реализация модели памяти C# на архитектурах x86 и x64

Так как x86 и x64 имеют одинаковое поведение в отношении модели памяти, я буду рассматривать здесь эти вариации процессорных архитектур совместно.

В отличие от некоторых архитектур процессоры x86-x64 обеспечивают весьма строгие гарантии порядка операций с памятью. По сути, JIT-компилятору не требуется использовать никаких специальных команд на процессорах x86-x64 для достижения семантики volatile; обычные операции с памятью уже поддерживают эту семантику. Но даже при таких гарантиях все равно есть специфические случаи, когда процессор x86-x64 осуществляет переупорядочение операций с памятью.

Переупорядочение операций с памятью на x86-x64 Хотя процессоры x86-x64 предоставляют весьма строгие гарантии порядка операций с памятью, в специфических условиях аппаратное переупорядочение все же происходит.

Процессор x86-x64 не будет переупорядочивать две операции записи, равно как и две операции чтения. Однако один (и только один) возможный эффект переупорядочения заключается в том, что, когда процессор записывает некое значение, оно не становится сразу же доступным другим процессорам. Пример, демонстрирующий такое поведение, приведен на **рис. 1**.

Рис. 1. StoreBufferExample

```
class StoreBufferExample
{
  // На x86 и .NET Framework 4.5 не имеет значения,
  // изменяемые эти поля или нет
  volatile int A = 0;
  volatile int B = 0;
  volatile bool A_Won = false;
  volatile bool B_Won = false;
  public void ThreadA()
  {
    A = true;
    if (!B) A_Won = true;
  }
  public void ThreadB()
  {
    B = true;
    if (!A) B_Won = true;
  }
}
```

Рассмотрим случай, где методы ThreadA и ThreadB вызываются из разных потоков применительно к новому экземпляру StoreBufferExample, как показано на **рис. 2**. Если подумать о результатах работы программы на **рис. 2**, то вроде бы возможны три случая.

1. Поток 1 завершается до запуска потока 2. Результат: A_Won=true, B_Won=false.
2. Поток 2 завершается до запуска потока 1. Результат: A_Won=false, B_Won=true.
3. Потоки чередуются. Результат: A_Won=false, B_Won=false.

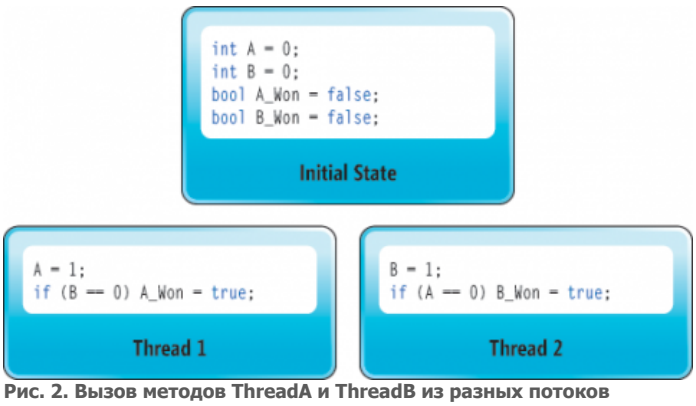

Рис. 2. Вызов методов ThreadA и ThreadB из разных потоков

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

Но, как это ни удивительно, существует и четвертый случай: есть вероятность того, что оба поля (A_Won и B_Won) будут равны true после выполнения этого кода! Из-за буфера сохранения (store buffer) хранилища могут «запаздывать» и поэтому в конечном счете будут переупорядочены с операцией последующей загрузки. Хотя этот результат не согласуется ни с каким чередованием выполнения потоков 1 и 2, тем не менее, он возможен.

Этот пример интересен потому, что мы имеем процессор (x86-x64) с относительно строгими гарантиями порядка операций с памятью и все поля изменяемые, а переупорядочение операций с памятью все равно наблюдаем. Хотя запись в A является volatile и чтение из A_Won тоже volatile, оба препятствия однонаправленные, и это фактически разрешает такое переупорядочение. Следовательно, метод ThreadA может выполняться так, будто он написан следующим образом:

```
public void ThreadA()
{
  bool tmp = B;
  A = true;
  if (!tmp) A_Won = 1;
}
```

Одно из возможных исправлений — вставка барьера памяти (memory barrier) в оба метода: ThreadA и ThreadB. Обновленный метод ThreadA мог бы выглядеть так:

```
public void ThreadA()
{
  A = true;
  Thread.MemoryBarrier();
  if (!B) aWon = 1;
}
```

CLR JIT вставит вместо барьера памяти команду lock or. В x86 команда с блокировкой (locked instruction) дает побочный эффект сброса буфера сохранения (store buffer):

```
mov          byte ptr [ecx+4],1
lock or      dword ptr [esp],0
cmp          byte ptr [ecx+5],0
jne          00000013
mov          byte ptr [ecx+6],1
ret
```

Любопытно, что в языке программирования Java применяется другой подход. В модель памяти Java заложено немного более строгое определение volatile, которое не разрешает переупорядочение операций сохранения-загрузки, поэтому компилятор Java на платформе x86, как правило, генерирует команду с блокировкой после volatile-записи.

Заметки по x86-x64 Процессор x86 имеет весьма строгую модель памяти, и единственный источник переупорядочения на аппаратном уровне — буфер сохранения. Этот буфер может привести к тому, что операция записи будет переупорядочена с последующей операцией чтения (переупорядочение «сохранение-загрузка»).

Кроме того, переупорядочение операций с памятью могут вызвать некоторые оптимизации компилятора. А именно, если несколько операций чтения обращаются к одному и тому же участку памяти, компилятор может выбрать выполнение только одной операции чтения и хранение считанного значения в регистре для последующих операций чтения.

Любопытный факт. Семантика volatile в C# очень близко соответствует гарантиям аппаратного переупорядочения процессоров x86-x64. В итоге операции чтения и записи изменяемых полей не требуют специальных команд на платформе x86 — достаточно обычных инструкций чтения и записи (например, MOV). Конечно, ваш код не должен полагаться на такие детали реализации, поскольку они варьируются на разных аппаратных архитектурах и даже в различных версиях .NET.

Реализация модели памяти C# на архитектуре Itanium

Модель памяти в аппаратной архитектуре Itanium менее строгая, чем на процессорах x86-x64. Itanium поддерживался .NET Framework вплоть до версии 4.

Хотя .NET Framework 4.5 больше не поддерживает Itanium, понимание модели памяти Itanium полезно при чтении старых статей по модели памяти .NET и для поддержки кода, который включал рекомендации из этих статей.

Переупорядочение на Itanium У Itanium совсем другой набор команд по сравнению с таковым у x86-x64, и концепции модели памяти проявляются в этом наборе. Itanium различает обычную загрузку (ordinary load, LD) и загрузку-получение (load-acquire, LD.ACQ), а также обычное сохранение (ordinary store, ST) и сохранение-освобождение (store-release, ST.REL).

Обычные операции загрузки и сохранения могут свободно переупорядочиваться на аппаратном уровне, если это не изменяет поведение однопоточного кода. Взгляните, к примеру, на этот код:

```
class ReorderingExample
{
  int _a = 0, _b = 0;
  void PrintAB()
  {
    int a = _a;
    int b = _b;
    Console.WriteLine("A:{0} B:{1}", a, b);
  }
  ...
}
```

Рассмотрим две операции чтения _a и _b в методе PrintAB. Так как операции чтения обращаются к обычному, неизменяемому полю, компилятор будет использовать LD (не LD.ACQ) для реализации этих операций. Соответственно две операции чтения могут быть переупорядочены аппаратным обеспечением, и PrintAB поведет себя так, словно он написан следующим образом:

```
void PrintAB()
{
  int b = _b;
  int a = _a;
  Console.WriteLine("A:{0} B:{1}", a, b);
}
```

На практике ответ на вопрос о том, произойдет ли переупорядочение, зависит от множества непредсказуемых факторов: что находится в процессорном кеше, насколько занят конвейер процессора и т. д. Однако процессор не будет переупорядочивать две операции чтения, если они связаны через зависимость от данных. Зависимость от данных между двумя операциями чтения наблюдается, когда значение, возвращаемое операцией чтения, определяет адрес данных для последующей операции чтения.

Вот пример, иллюстрирующий зависимость от данных:

```
class Counter { public int _value; }
class Test
{
  private Counter _counter = new Counter();
  void Do()
  {
    Counter c = _counter; // чтение 1
    int value = c._value; // чтение 2
  }
}
```

В методе Do процессор Itanium никогда не будет переупорядочивать операции чтения 1 и 2, хотя чтение 1 — обычная загрузка, а не загрузка-получение. Может показаться очевидным, что эти две операции чтения нельзя переупорядочивать: первая из них определяет участок памяти, к которому должна обратиться вторая операция! Однако некоторые процессоры, отличные от Itanium, на самом деле могут переупорядочивать такие операции чтения. Процесс может предположить, какое значение вернет первая операция чтения, и спекулятивно выполнить чтение 2 — даже до выполнения чтения 1. Но опять же Itanium такого никогда не делает.

Я вернусь к обсуждению зависимости от данных на платформе Itanium чуть позже, и ее значимость для модели памяти C# станет понятнее.

Itanium также не будет переупорядочивать две операции обычного чтения, если они связаны через зависимость управления (control dependency). Такая зависимость наблюдается, когда значение, возвращаемое операцией чтения, определяет, будет ли исполняться последующая команда.

Поэтому в данном примере операции чтения _initialized и _data связаны зависимостью управления:

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

Даже если _initialized и _data являются обычными операциями чтения (неизменяемых полей), процессор Itanium не переупорядочит их. Заметьте, что JIT-компилятор, тем не менее, свободен в переупорядочении этих двух операций и в некоторых случаях так и сделает.

Кроме того, стоит отметить, что подобно x86-x64 процессор Itanium также использует буфер сохранения, поэтому StoreBufferExample, показанный на **рис. 1**, подвергнется на Itanium такому же переупорядочению, что и на x86-x64. Любопытно, что, если вы используете LD.ACQ для всех операций чтения и ST.REL для всех операций записи на Itanium, то, по сути, вы получите модель памяти x86-x64, где единственная причина переупорядочения — буфер сохранения.

Поведение компилятора на Itanium JIT-компилятор в CLR проявляет одно удивительное поведение на Itanium: все операции записи генерируются как ST.REL, а не ST. Соответственно операции записи в изменяемое и неизменяемое поля, как правило, будут приводить к генерации одной и той же команды для Itanium. Однако операция обычного чтения будет транслироваться как LD, и только операции чтения из изменяемых полей будут генерироваться как LD.ACQ.

Это поведение может стать сюрпризом, так как компилятору определенно не требуется выдавать ST.REL для операций записи в неизменяемые поля. По спецификации ECMA C# компилятор мог бы генерировать обычные инструкции ST. Генерация ST.REL — это нечто дополнительное, что выбирается компилятором для того, чтобы гарантировать ожидаемую работу конкретного (но теоретически неправильного) шаблона.

Трудно вообразить столь важный шаблон, где для операций записи нужно было бы использовать ST.REL, но для операций чтения достаточно LD. В примере с PrintAB, представленном ранее в этом разделе, ограничение только операций записи ничем не поможет, поскольку операции чтения по-прежнему могут переупорядочиваться.

Есть один очень важный сценарий, в котором использовать ST.REL в сочетании с обычными LD вполне достаточно: когда операции загрузки упорядочиваются сами, используя зависимость от данных. Этот шаблон проявляется в отложенной инициализации, а она крайне важна. Пример отложенной инициализации (lazy initialization) показан на **рис. 3**.

Рис. 3. Отложенная инициализация

```
// Предупреждение: может не работать на будущих архитектурах
// и версиях .NET. Не использовать!
class LazyExample
{
  private BoxedInt _boxedInt;
  int GetInt()
  {
    BoxedInt b = _boxedInt; // чтение 1
    if (b == null)
    {
      lock(this)
      {
        if (_boxedInt == null)
        {
          b = new BoxedInt();
          b._value = 42;  // запись 1
          _boxedInt = b; // запись 2
        }
      }
    }
    int value = b._value; // чтение 2
    return value;
  }
}
```

Чтобы этот код всегда возвращал 42, даже если GetInt вызывается одновременно из нескольких потоков, чтение 1 нельзя переупорядочивать с чтением 2, а запись 1 — с записью 2. Операции чтения не будут переупорядочиваться на процессоре Itanium, так как они связаны через зависимость от данных. А операции записи не будут переупорядочиваться, поскольку CLR JIT транслирует их в команды ST.REL.

Заметьте: если бы поле _boxedInt было изменяемым, код был бы корректным согласно спецификации ECMA C#. Это лучший вариант и, несомненно, единственно правильный. Однако, даже если _boxed не является изменяемым, текущая версия компилятора гарантирует, что этот код все равно будет работать на Itanium.

Конечно, CLR JIT может выполнять на Itanium выделение операции чтения из цикла (loop read hoisting), исключение операции чтения (read elimination) и введение дополнительного чтения (read introduction) точно так же, как на процессорах x86-x64.

Заметки по Itanium Причина, по которой Itanium является интересной частью нашего обсуждения, заключается в том, что это была первая архитектура с нестрогой моделью памяти, которая поддерживалась .NET Framework.

В итоге в ряде статей по модели памяти C# и ключевому слову volatile в этом языке их авторы фактически имели в виду Itanium. В конце концов, Itanium до появления .NET Framework 4.5 был единственной архитектурой, отличной от x86-x64, которая могла выполнять .NET Framework.

Соответственно автор мог сказать нечто вроде: «В модели памяти .NET 2.0 все операции записи являются volatile, даже если выполняются применительно к неизменяемым полям». При этом автор подразумевал, что CLR на Itanium будет транслировать все операции записи в команды ST.REL. Это поведение не гарантируется спецификацией ECMA C#, а значит, может не сохраниться в будущих версиях .NET Framework и на будущих архитектурах (и действительно такое поведение не сохранено в .NET Framework 4.5 на архитектуре ARM).

Аналогично некоторые могли бы доказывать, что отложенная инициализация корректна в .NET Framework, даже если поле хранения (holding field) неизменяемое, тогда как другие заявляли бы, что это поле должно быть изменяемым.

И конечно, разработчики писали код, опираясь на эти (иногда противоречивые) допущения. Вот почему понимание того, что происходит на Itanium, может оказаться полезным, когда вы будете разбираться в параллельном коде, написанном кем-то другим, читать старые статьи или даже просто общаться с другими разработчиками.

Реализация модели памяти C# на ARM

ARM совсем недавно появилась в списке архитектур, поддерживаемых .NET Framework. Подобно Itanium ARM имеет менее строгую модель памяти, чем x86-x64.

Переупорядочение на ARM Как и Itanium, этот процессор разрешает свободно переупорядочивать обычные операции чтения и записи. Однако решение, заложенное в ARM для управления перемещением операций чтения и записи, несколько отличается от такового на Itanium. ARM предоставляет одну инструкцию — DMB, которая действует как полноценный барьер памяти. Никакие операции с памятью не могут преодолеть DMB в любом направлении.

В дополнение к ограничениям, накладываемым DMB, ARM также поддерживает зависимость от данных, но игнорирует зависимость управления. Пояснения по зависимостям см. в разделе «Переупорядочение на Itanium» ранее в этой статье.

Поведение компилятора на ARM Команда DMB используется для реализации семантики volatile в C#. JIT-компилятор CLR на ARM реализует операцию чтения из изменяемого поля как обычную команду чтения (например, LDR), за которой следует команда DMB. Так как инструкция DMB предотвратит переупорядочение volatile-чтения с любой последующей операцией, это решение корректно реализует семантику получения (acquire semantics).

Запись в изменяемое поле реализуется инструкцией DMB, за которой следует обычная операция записи (например, STR). Так как DMB предотвращает переупорядочение volatile-записи с любыми предыдущими операциями, это решение корректно реализует семантику освобождения (release semantics).

Как и в случае процессора Itanium, было бы неплохо выйти за рамки спецификации ECMA C# и сохранить работоспособность шаблона отложенной инициализации, поскольку на нее полагается огромное количество кода. Однако фактическое превращение любой записи в volatile-операцию — не слишком хорошее решение на ARM, потому что выполнение команды DBM обходится весьма дорого.

В .NET Framework 4.5 JIT-компилятор CLR использует несколько иной фокус, чтобы добиться работы отложенной инициализации. Все перечисленное ниже интерпретируется как барьеры «освобождения».

1. Операции записи в поля ссылочных типов в куче, управляемой сборщиком мусора (garbage collector, GC).
2. Операции записи в статические поля ссылочных типов.

В результате любая операция записи может публиковать объект, обрабатываемый как барьер освобождения.

Ниже дана соответствующая часть LazyExample (помните, что ни одно из полей не является изменяемым):

```
b = new BoxedInt();
b._value = 42;  // запись 1
// DMB будет генерироваться здесь
_boxedInt = b; // запись 2
```

Поскольку CLR JIT генерирует команды DMB до публикации объекта в поле _boxedInt, операции записи 1 и 2 не будут переупорядочиваться. А так как ARM соблюдает зависимость от данных, операции чтения в шаблоне отложенной инициализации тоже не будут переупорядочиваться, и код будет работать корректно на ARM.

Таким образом, CLR JIT предпринимает дополнительные усилия (выходящие за рамки того, что определено в спецификации ECMA C#) для сохранения работоспособности наиболее распространенной вариации некорректной отложенной инициализации на ARM.

В качестве последнего комментария по ARM замечу, что, как и на x86-x64 и Itanium, выделение операции чтения из цикла, исключение операций чтения и введение дополнительной операции чтения являются допустимыми оптимизациями, если речь идет о JIT-компиляторе в CLR.

Пример: отложенная инициализация

Может оказаться весьма поучительным посмотреть на несколько вариаций шаблона отложенной инициализации и подумать о том, как они будут вести себя на разных архитектурах.

Корректная реализация Реализация отложенной инициализации на **рис. 4** корректна согласно модели памяти C#, определенной в спецификации ECMA C#, и поэтому она гарантирует работу на всех архитектурах, поддерживаемых текущей и будущими версиями .NET Framework.

Рис. 4. Корректная реализация отложенной инициализации

```
class BoxedInt
{
  public int _value;
  public BoxedInt() { }
  public BoxedInt(int value) { _value = value; }
}
class LazyExample
{
  private volatile BoxedInt _boxedInt;
  int GetInt()
  {
    BoxedInt b = _boxedInt;
    if (b == null)
    {
      b = new BoxedInt(42);
      _boxedInt = b;
    }

    return b._value;
  }
}
```

Хотя пример кода корректен, на практике все равно предпочтительнее использовать тип Lazy<T> или LazyInitializer.

Некорректная реализация номер 1 На **рис. 5** показана реализация, некорректная согласно модели памяти C#. Несмотря на это, данная реализация, вероятно, будет работать на x86-x64, Itanium и ARM в .NET Framework. Эта версия кода неправильна. Поскольку _boxedInt не является изменяемым, компилятору C# разрешается переупорядочивать чтение 1 с чтением 2 или запись 1 с записью 2. Любое из этих переупорядочений потенциально может привести к тому, что GetInt вернет 0.

Рис. 5. Некорректная реализация отложенной инициализации

```
// Предупреждение: плохой код
class LazyExample
{
  private BoxedInt _boxedInt; // это поле неизменяемое
  int GetInt()
  {
    BoxedInt b = _boxedInt; // чтение 1
    if (b == null)
    {
      b = new BoxedInt(42); // запись 1 (внутри конструктора)
      _boxedInt = b;        // запись 2
    }
    return b._value;        // чтение 2
  }
}
```

Однако этот код будет вести себя корректно (т. е. всегда возвращать 42) на всех архитектурах в .NET Framework версий 4 и 4.5.

- x86-x64:
  - операции записи и чтения не будут переупорядочиваться. В этом коде нет шаблона сохранения-загрузки (store-load), и поэтому нет причин, которые заставили бы компилятор кешировать значения в регистрах.
- Itanium:
  - операции записи не будут переупорядочиваться, так как все они выражаются командами ST.REL;
  - операции чтения не будут переупорядочиваться из-за зависимости от данных.
- ARM:
  - операции записи не будут переупорядочиваться, так как перед «_boxedInt = b» генерируется инструкция DMB;
  - операции чтения не будут переупорядочиваться из-за зависимости от данных.

Конечно, вы должны использовать эту информацию только для того, чтобы понять поведение существующего кода. Не применяйте этот шаблон при написании нового кода.

Некорректная реализация номер 2 Реализация на **рис. 6** может не работать на процессорах ARM и Itanium.

Рис. 6. Вторая некорректная реализация отложенной инициализации

```
// Предупреждение: плохой код
class LazyExample
{
  private int _value;
  private bool _initialized;
  int GetInt()
  {
    if (!_initialized) // чтение 1
    {
      _value = 42;
      _initialized = true;
    }
    return _value;     // чтение 2
  }
}
```

В этой версии отложенной инициализации используются два поля: _value (для отслеживания данных) и _initialized (для определения того, было ли инициализировано поле). В итоге операции чтения 1 и 2 больше не связаны через зависимость от данных. Кроме того, на ARM операции записи тоже могут быть переупорядочены по тем же причинам, что и в следующей некорректной реализации (номер 3).

На практике эта версия может работать неправильно и возвращать 0 на ARM и Itanium. Конечно, GetInt разрешается возвращать 0 на x86-x64 (и это тоже результат JIT-оптимизаций), но такого поведения в .NET Framework 4.5 не наблюдается.

Некорректная реализация номер 3 Наконец, пример можно сделать неработоспособным даже на x86-x64. Достаточно добавить одну невинно выглядящую операцию чтения, как показано на **рис. 7**.

Рис. 7. Третья некорректная реализация отложенной инициализации

```
// Предупреждение: плохой код
class LazyExample
{
  private int _value;
  private bool _initialized;
  int GetInt()
  {
    if (_value < 0) throw new Exception(); // Примечание: доп
                                           // для предваритель
        if (!_initialized)      // чтение 1
    {
      _value = 42;
      _initialized = true;
      return _value;
    }
    return _value;          // чтение 2
  }
}
```

Дополнительная операция чтения, которая выполняет проверку _value < 0, теперь может заставить компилятор кешировать значение в регистре. В результате чтение 2 будет обслужено из регистра и в конечном счете будет переупорядочено с чтением 1. Соответственно эта версия GetInt может на практике возвращать 0 даже на x86-x64.

Заключение

При написании нового многопоточного кода, в целом, следует вообще избегать сложности модели памяти C#, используя высокоуровневые примитивы параллельной обработки вроде блокировок, параллельных наборов, задач и параллельных циклов. Если вы пишете код, интенсивно использующий процессор, иногда имеет смысл применять изменяемые поля при условии, что вы полагаетесь только на гарантии спецификации ECMA C#, а не на детали реализации, специфичные для конкретной архитектуры.

Автор: *Игорь Островский* • ***Источник:*** *MSDN Magazine* • ***Опубликована:*** *18.04.2013*

Похожие материалы раздела

- Модель памяти C# в теории и на практике

Теги:

Комментарии посетителей