tguev
8 апр 2013 в 23:12

Особенности строк в .NET

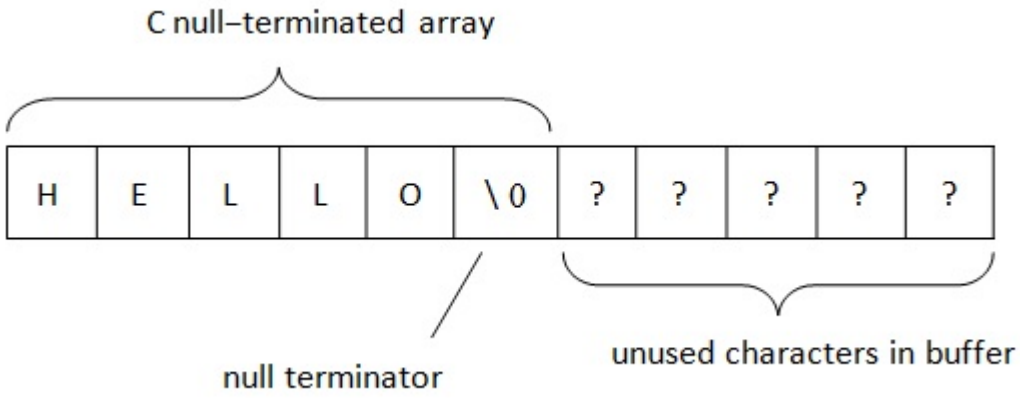
10 мин 105K

.NET*, C#*

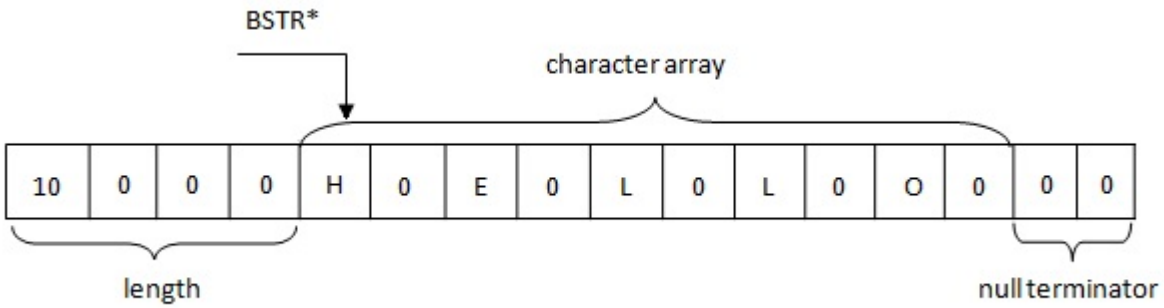
Строковый тип данных является одним из самых важных в любом языке программировании. Вряд ли можно написать полезную программу не задействовав этот тип данных. При этом многие разработчики не знают некоторых нюансов связанных с этим типом. Поэтому давайте рассмотрим кое-какие особенности этого типа в .NET.

Итак, начнем с представления строк в памяти

В.NET строки располагаются согласно правилу BSTR (Basic string or binary string). Данный способ представления строковых данных используется в COM (слово basic от языка программирования VisualBasic, в котором он первоначально использовался). Как известно в C/C++ для представления строк используется PWSZ, что расшифровывается как **Pointer to Wide-character String, Zero-terminated**. При таком расположении в памяти в конце строки находится null-терминированный символ, по которому мы можем определить конец строки. Длина строки в PWSZ ограничена лишь объемом свободной памяти.



С BSTR дело обстоит немного иначе.



Основные особенности BSTR представления строки в памяти:

1. Длина строки ограничена неким числом в отличие от PWSZ, где длина строки ограничена наличием свободной памяти.
2. BSTR строка всегда указывает на первый символ в буфере. PWSZ может указывать на любой символ в буфере.
3. У BSTR всегда в конце находится null символ, так же как и у PWSZ, но в отличие от последнего он является валидным символом и может встречаться в строке где угодно.
4. За счет наличия null-символа в конце BSTR совместим с PWSZ, но не наоборот.

Так вот, строки в .NET представляются в памяти согласно правилу BSTR. В буфере находится четырехбайтовая длина строки, за которой следуют двухбайтовые символы строки в формате UTF-16, за которыми следует два нулевых байта (\u0000).

Использование такой реализации имеет ряд преимуществ: длину строки не нужно пересчитывать она хранится в заголовке, строка может содержать null-символы, где угодно, и самое главное адрес строки(pinned) можно без проблем передавать в неуправляемой код там, где ожидается WCHAR*.

Идем далее…

Сколько памяти занимает объект строкового типа?

Мне встречались статьи где было написано, что размер строкового объекта равен size = 20 + (length/2)*4, однако эта формула не совсем правильная.
Начнем с того, что строка является ссылочным типом, поэтому первые 4 байта содержат SyncBlockIndex, а вторые 4 байта содержат указатель на тип.

Размер строки = 4 + 4 + ...

Как было выше сказано, в буфере хранится длина строки — это поле типа int, значит еще 4 байта.

Размер строки = 4 + 4 + 4 + ...

Для того, чтобы быстро передать строку в неуправляемый код (без копирования) в конце каждой строки стоит null-терминированный символ, который занимает 2 байта, значит

Размер строки = 4 + 4 + 4 + 2 + ...

Осталось вспомнить, что каждый символ в строке находится в UTF -16 кодировке значит, занимает так же 2 байта, следовательно

Размер строки = 4 + 4 + 4 + 2 + 2 * length = 14 + 2 * length

Учтем еще один нюанс, и мы у цели. А именно менеджер памяти в CLR выделяет память кратной 4 байтам (4, 8, 12, 16, 20, 24, ...), то есть если длина строки суммарно будет занимать 34 байта, то выделено будет 36 байта. Нам необходимо округлить наше значение к ближайшему большему кратному четырем числу, для этого необходимо:

Размер строки = 4 * ((14 + 2 * length + 3) / 4) (деление естественно целочисленное)

Вопрос версий: В .NET до 4 версии в классе String хранится дополнительное поле m_arrayLength типа int, которое занимает 4 байта. Данное поле есть реальная длина буфера выделенного под строку включая null — терминированный символ, то есть это length + 1. В .NET 4.0 данное поля удалено из класса, в результате чего объект строкового типа занимает на 4 байта меньше.

Размер пустой строки без поля m_arrayLength(то есть в .NET 4.0 и выше) равен = 4 + 4 + 4 + 2 = 14 байт, а с этим полем (то есть ниже .NET 4.0) равен = 4 + 4 + 4 + 4 + 2 = 18 байт. Если округлять по 4 байта то 16 и 20 байт соответственно.

Особенности строк

Итак, мы рассмотрели, как представляются строки, и сколько на самом деле они занимают места в памяти. Теперь давайте погорим об их особенностях.

Основные особенности строк в .NET:

1. Они являются ссылочными типами.
2. Они неизменяемы. Однажды, создав строку, мы больше не можем ее изменить (честным способом). Каждый вызов метода этого класса возвращает новую строку, а предыдущая строка становится добычей для сборщика мусора.
3. Они переопределяют метод Object.Equals, в результате чего он сравнивает не значения ссылок, а значения символов в строках.

Рассмотрим каждый пункт подробнее.

Строки — ссылочные типы

Строки являются настоящими ссылочными типами, то есть они всегда располагаются в

куче. Многие путают их со значимыми типами, потому что они ведут себя также, например, они неизменяемы и их сравнение происходит по значению, а не по ссылкам, но нужно помнить, что это ссылочный тип.

Строки — неизменяемы

Строки являются неизменяемыми. Это сделано не просто так. В неизменности строк есть немало преимуществ:

- Строковый тип является потокобезопасным, так как ни один поток не может изменить содержимое строки.
- Использование неизменных строк ведет к снижению нагрузки на память, так как нет необходимости хранить 2 экземпляра одной строки. В таком случае и памяти меньше расходуется, и сравнение происходит быстрее, так как требует сравнение лишь ссылок. Механизм, который это реализует в .NET называется интернированием строк (пул строк), о нем поговорим чуть позже.
- При передаче неизменяемого параметра в метод мы можем не беспокоиться, что он будет изменен (если, конечно, он не был передан как ref или out).

Структуры данных можно разделить на два вида — эфемерные и персистентные. Эфемерными называют структуры данных, хранящие только последнюю свою версию. Персистентными называют структуры, которые сохраняют все свои предыдущие версии при изменении. Последние фактически неизменяемы, так как их операции не изменяют структуру на месте, вместо этого они возвращают новую основанную на предыдущей структуру.

Учитывая, что строки неизменны, они могли бы быть и персистентными, однако таковыми не являются. В .NET строки являются эфемерными. Подробнее о том, почему это именно так можно прочитать у Эрика Липперта по ссылке

Для сравнения возьмем строки Java. Они являются неизменяемыми, как и в .NET, но вдобавок и персистентными. Реализация класса String в Java выглядит так:

```
public final class String
    {
        private final char value[];
        private final int offset;
    private final int count;
    private int hash;
```

```
.....
}
```

Помимо тех же 8 байт в заголовке объекта, включающие ссылку на тип и ссылку на объект синхронизации строки содержат следующие поля:

1. Ссылка на массив символов char;
2. Индекс первого символа строки в массиве char (смещение он начала);
3. Количество символов в строке;
4. Посчитанный хэш-код, после первого вызова метода hashCode();

Как видно, строки в Java занимают больше памяти, чем в .NET, так как содержат дополнительные поля, которые и позволяют им быть персистентными. Благодаря персистентости метод **String.substring()** в Java выполняется за O(1), так как не требует копирования строки как в .NET, где этот метод выполняется за O(n).

Реализация метода String.substring() в Java:

```
ublic String substring(int beginIndex, int endIndex)

if (beginIndex < 0)
  throw new StringIndexOutOfBoundsException(beginIndex);
if (endIndex > count)
  throw new StringIndexOutOfBoundsException(endIndex);
if (beginIndex > endIndex)
  throw new StringIndexOutOfBoundsException(endIndex - beginIndex);
return ((beginIndex == 0) && (endIndex == count)) ? this : new String(offset + beginIndex


ublic String(int offset, int count, char value[])

this.value = value;
this.offset = offset;
this.count = count;
```

Однако, согласно принципу ЛДНБ (ланчей даром не бывает), о котором так часто говорит Эрик Липперт не все так хорошо. Если исходная строка будет достаточно большой, а вырезаемая подстрока в пару символов, то весь массив символов первоначальной строки

будет висеть в памяти пока есть ссылка на подстроку или, если вы сериализуете полученную подстроку стандартными средствами и передаете её по сети, то будет сериализован весь оригинальный массив и количество передаваемых байтов по сети будет большим. Поэтому в таком случае вместо кода

s = ss.substring(3)

можно использовать код

s = new String(ss.substring(3)),

который не будет хранить ссылку на массив символов исходной строки, а скопирует только реально используемую часть массива. Кстати, если этот конструктор вызывать на строке длиной равной длине массива символов, то копирования в этом случае происходить не будет, а будет использоваться ссылка на оригинальный массив.

Как оказалось в последней версии Java реализация строкового типа изменилась. @xonix подсказал об этом. Теперь в классе нет полей offset и length, и появился новый hash32 (с другим алгоритмом хеширования). Это означает, что строки перестали быть персистентными. Теперь метод String.substring каждый раз будет создаваться новую строку.

Строки переопределяют Object.Equals

Класс String переопределяет метод Object.Equals, в результате чего сравнение происходит не по ссылке, а по значению. Я думаю, разработчики благодарны создателям класса String за то, что они переопределили оператор ==, так как код, использующий == для сравнения строк, выглядит более изящно, нежели вызов метода.

```
if (s1 == s2)
```

в сравнении

```
if (s1.Equals(s2))
```

Кстати, в Java оператор == сравнивает по ссылке, а для того чтобы сравнить строки посимвольно необходимо использовать метод string.equals().

Интернирование строк

Ну, и на последок поговорим об интернировании строк.
Рассмотрим простой пример, код который переворачивает строку.

```
var s = "Strings are immutuble";
int length = s.Length;
for (int i = 0; i < length / 2; i++)
{
   var c = s[i];
   s[i] = s[length - i - 1];
   s[length - i - 1] = c;
}
```

Очевидно, данный код не с компилируется. Компилятор будет ругаться на эти строки, потому что мы пытаемся изменить содержимое строки. Действительно, любой метод класса String возвращает новый экземпляр строки, вместо того чтобы изменять свое содержимое.

На самом деле строку можно изменить, но для этого придется прибегнуть к unsafe коду. Рассмотрим пример:

```
var s = "Strings are immutable";
int length = s.Length;
  unsafe
   {
    fixed (char* c = s)
     {
      for (int i = 0; i < length / 2; i++)
       {
         var temp = c[i];
         c[i] = c[length - i - 1];
         c[length - i - 1] = temp;
       }
      }
   }
```

После выполнения этого кода, как и ожидалось, в строке будет записано **elbatummi era sgnirtS**.
Тот факт, что строки являются все-таки изменяемыми, приводит к одному очень интересному казусу. Связан он с интернированием строк.

Интернирование строк — это механизм, при котором одинаковые литералы представляют собой один объект в памяти.

Если не вникать глубоко в подробности, то смысл интернирования строк заключается в следующем: в рамках процесса (именно процесса, а не домена приложения) существует одна внутренняя хеш-таблица, ключами которой являются строки, а значениями – ссылки на них. Во время JIT-компиляции литеральные строки последовательно заносятся в таблицу (каждая строка в таблице встречается только один раз). На этапе выполнения ссылки на литеральные строки присваиваются из этой таблицы. Можно поместить строку во внутреннюю таблицу во время выполнения с помощью метода String.Intern. Также можно проверить, содержится ли строка во внутренней таблице с помощью метода String.IsInterned.

```
var s1 = "habrahabr";
var s2 = "habrahabr";
var s3 = "habra" + "habr";

Console.WriteLine(object.ReferenceEquals(s1, s2));//true
Console.WriteLine(object.ReferenceEquals(s1, s3));//true
```

Важно отметить, что интернируются по умолчанию только строковые литералы. Поскольку для реализации интернирования используется внутренняя хеш-таблица, то во время JIT компиляции происходит поиск по ней, что занимает время, поэтому если бы интернировались все строки, то это свело бы на нет всю оптимизацию. Во время компиляции в IL код, компилятор конкатенирует все литеральные строки, так как нет в необходимости содержать их по частям, поэтому 2 — ое равенство возвращает true. Так вот, в чем заключается казус. Рассмотрим следующий код:

```
var s = "Strings are immutable";
int length = s.Length;
unsafe
 {
  fixed (char* c = s)
   {
    for (int i = 0; i < length / 2; i++)
     {
      var temp = c[i];
      c[i] = c[length - i - 1];
      c[length - i - 1] = temp;
     }
   }
```

```
 }
Console.WriteLine("Strings are immutable");
```

Кажется, что здесь все очевидно и, что такой код должен распечатать **Strings are immutable**. Однако, нет! Код напечатает **elbatummi era sgnirtS**. Дело именно в интернировании, изменяя строку s, мы меняем ее содержимое, а так как она является литералом, то интернируется и представляется одним экземпляром строки.

От интернирования строк можно отказаться, если применить специальный атрибут **CompilationRelaxationsAttribute** к сборке. Атрибут **CompilationRelaxationsAttribute** контролирует точность кода, создаваемого JIT-компилятором среды CLR. Конструктор данного атрибута принимает перечисление **CompilationRelaxations** в состав, которого на текущий момент входит только **CompilationRelaxations.NoStringInterning** — что помечает сборку как не требующую интернирования.

Кстати, этот атрибут не обрабатывается в .NET Framework версии 1.0., поэтому отключить интернирование по умолчанию не было возможным. Сборка mscorlib, начиная со второй версии, помечена этим атрибутом.

Получается, что строки в .NET все-таки можно изменить, если очень захотеть, применяя unsafe код.

А что если без unsafe?

Оказывается, изменить содержимое строки было возможно и, не прибегая к unsafe коду, воспользовавшись механизмом рефлексии. Этот трюк мог прокатить в .NET до 2.0 версии включительно, потом разработчики класса String лишили нас такой возможности.
В версии .NET 2.0 у класса String есть два internal метода: **SetChar**, проверяющий выход за границы, и **InternalSetCharNoBoundsCheck**, не проверяющий выход за границы, которые устанавливают указанный символ по определенному индексу. Вот их имплементация:

```
internal unsafe void SetChar(int index, char value)
 {
   if ((uint)index >= (uint)this.Length)
     throw new ArgumentOutOfRangeException("index", Environment.GetResourceString("Argu

   fixed (char* chPtr = &this.m_firstChar)
          chPtr[index] = value;
 }

internal unsafe void InternalSetCharNoBoundsCheck (int index, char value)
 {
```

```
    fixed (char* chPtr = &this.m_firstChar)
           chPtr[index] = value;
  }
```

Таким образом, используя следующий код, можно изменить содержимое строки, даже не прибегая к использованию unsafe коду.

```
var s = "Strings are immutable";
int length = s.Length;
var method = typeof(string).GetMethod("InternalSetCharNoBoundsCheck", BindingFlags.Inst
for (int i = 0; i < length / 2; i++)
  {
      var temp = s[i];
      method.Invoke(s, new object[] { i, s[length - i - 1] });
      method.Invoke(s, new object[] { length - i - 1, temp });
  }

 Console.WriteLine("Strings are immutable");
```

Этот код как уже ожидалось, напечатает **elbatummi era sgnirtS**.

Вопрос версий: В разных версиях .NET Framework string.Empty может интернироваться, а может, и нет.
Рассмотрим код:

```
string str1 = String.Empty;
StringBuilder sb = new StringBuilder().Append(String.Empty);
string str2 = String.Intern(sb.ToString());

if (object.ReferenceEquals(str1, str2))
   Console.WriteLine("Equal");
else
   Console.WriteLine("Not Equal");
```

В .NET Framework 1.0, .NET Framework 1.1 и .NET Framework 3.5 с пакетом обновления 1 (SP1), str1 и str2 равны. В .NET Framework 2.0 с пакетом обновления 1 (SP1) и .NET Framework 3.0, str1 и str2 не равны. В настоящее время string.Empty интернируется.

Особенности производительности

У интернирования есть отрицательный побочный эффект. Дело в том, что ссылка на интернированный объект String, которую хранит CLR, может сохраняться и после завершения работы приложения и даже домена приложения. Поэтому большие литеральные строки использовать не стоит или же, если это необходимо стоит отключить интернирование, применив атрибут CompilationRelaxations к сборке.

Надеюсь, статья оказалось полезной...

Теги: строки, структура данных, особенности

Хабы: .NET, C#

177 Карма 374 Рейтинг

Тимур Гуев @tguev
Основатель BEEGEEK, автор курсов Поколение Python