How do I get a consistent byte representation of strings in C# without manually specifying an encoding?

Asked 10 years, 9 months ago Active 1 month ago Viewed 1.1m times

2107

430

How do I convert a `string` to a `byte[]` in .NET (C#) without manually specifying a specific encoding?

I'm going to encrypt the string. I can encrypt it without converting, but I'd still like to know why encoding comes to play here.

Also, why should encoding be taken into consideration? Can't I simply get what bytes the string has been stored in? Why is there a dependency on character encodings?

c# .net string character-encoding

edited Oct 1 '18 at 12:36
Dragonthoughts
1,819 4 13 17

asked Jan 23 '09 at 13:39
Agnel Kurian
29.1k 36 128 200

22 Every string is stored as an array of bytes right? Why can't I simply have those bytes? – Agnel Kurian Jan 23 '09 at 14:05

129 The encoding *is* what maps the characters to the bytes. For example, in ASCII, the letter 'A' maps to the number 65. In a different encoding, it might not be the same. The high-level approach to strings taken in the .NET framework makes this largely irrelevant, though (except in this case). – Lucas Jones Apr 13 '09 at 14:13

17 To play devil's advocate: If you wanted to get the bytes of an in-memory string (as .NET uses them) and manipulate them somehow (i.e. CRC32), and NEVER EVER wanted to decode it back into the original string...it isn't straight forward why you'd care about encodings or how you choose which one to use. – Greg Dec 1 '09 at 19:47

72 Surprised no-one has given this link yet: joelonsoftware.com/articles/Unicode.html – Bevan Jun 29 '10 at 2:57

24 A char is not a byte and a byte is not a char. A char is both a key into a font table and a lexical tradition. A string is a sequence of chars. (A words, paragraphs, sentences, and titles also have their own lexical traditions that justify their own type definitions -- but I digress). Like integers, floating point numbers, and everything else, chars are encoded into bytes. There was a time when the encoding was simple one to one: ASCII. However, to accommodate all of human symbology, the 256 permutations of a byte were insufficient and encodings were devised to selectively use more bytes. – George Aug 28 '14 at 15:43

39 Answers

1 2 next

1812

Contrary to the answers here, you DON'T need to worry about encoding *if* the bytes don't need to be interpreted!

Like you mentioned, your goal is, simply, to *"get what bytes the string has been stored in"*.
(And, of course, to be able to re-construct the string from the bytes.)

For those goals, I honestly do *not* understand why people keep telling you that you need the encodings. You certainly do NOT need to worry about encodings for this.

Just do this instead:

```
static byte[] GetBytes(string str)
{
    byte[] bytes = new byte[str.Length * sizeof(char)];
    System.Buffer.BlockCopy(str.ToCharArray(), 0, bytes, 0, bytes.Length);
    return bytes;
}

// Do NOT use on arbitrary bytes; only use on GetBytes's output on the SAME system
static string GetString(byte[] bytes)
{
    char[] chars = new char[bytes.Length / sizeof(char)];
    System.Buffer.BlockCopy(bytes, 0, chars, 0, bytes.Length);
    return new string(chars);
}
```

As long as your program (or other programs) don't try to *interpret* the bytes somehow, which you obviously didn't mention you intend to do, then there is **nothing** wrong with this approach! Worrying about encodings just makes your life more complicated for no real reason.

Additional benefit to this approach:

It doesn't matter if the string contains invalid characters, because you can still get the data and reconstruct the original string anyway!

It will be encoded and decoded just the same, because you are *just looking at the bytes*.

If you used a specific encoding, though, it would've given you trouble with encoding/decoding invalid characters.

edited Jun 3 at 20:34

answered Apr 30 '12 at 7:44

Mehrdad

134k 96 432 776

241 What's ugly about this one is, that `GetString` and `GetBytes` need to executed on a system with the same endianness to work. So you can't use this to get bytes you want to turn into a string elsewhere. So I have a hard time to come up with a situations where I'd want to use this. – CodesInChaos May 13 '12 at 11:14

71 @CodeInChaos: Like I said, the whole point of this is if you want to use it on the same kind of system, with the same set of functions. If not, then you shouldn't use it. – Mehrdad May 13 '12 at 18:00

189 -1 I guarantee that someone (who doesn't understand bytes vs characters) is going to want to convert their string into a byte array, they will google it and read this answer, and they will do the wrong thing, because in almost all cases, the encoding *IS* relevant. – artbristol Jun 15 '12 at 11:07

390 @artbristol: If they can't be bothered to read the answer (or the other answers...), then I'm sorry, then there's no better way for me to communicate with them. I generally opt for answering the OP rather than trying to guess what others might do with my answer -- the OP has the right to know, and just because someone might abuse a knife doesn't mean we need to hide all knives in the world for ourselves. Though if you disagree that's fine too. – Mehrdad Jun 15 '12 at 14:04

183 This answer is wrong on so many levels but foremost because of it's decleration "you DON'T need to worry about encoding!". The 2 methods, GetBytes and GetString are superfluous in as much as they are merely re-implementations of what Encoding.Unicode.GetBytes() and Encoding.Unicode.GetString() already do. The statement "As long as your program (or other programs) don't try to interpret the bytes" is also fundamentally flawed as implicitly they mean the bytes should be interpreted as Unicode. – David Jul 11 '12 at 12:36

1092

It depends on the encoding of your string (ASCII, UTF-8, ...).

For example:

```
byte[] b1 = System.Text.Encoding.UTF8.GetBytes (myString);
byte[] b2 = System.Text.Encoding.ASCII.GetBytes (myString);
```

A small sample why encoding matters:

```
string pi = "\u03a0";
byte[] ascii = System.Text.Encoding.ASCII.GetBytes (pi);
byte[] utf8 = System.Text.Encoding.UTF8.GetBytes (pi);

Console.WriteLine (ascii.Length); //Will print 1
```

```
Console.WriteLine (utf8.Length); //Will print 2
Console.WriteLine (System.Text.Encoding.ASCII.GetString (ascii)); //Will print '?'
```

ASCII simply isn't equipped to deal with special characters.

Internally, the .NET framework uses UTF-16 to represent strings, so if you simply want to get the exact bytes that .NET uses, use `System.Text.Encoding.Unicode.GetBytes (...)` .

See *Character Encoding in the .NET Framework* (MSDN) for more information.

edited Apr 24 '15 at 9:52
Peter Mortensen
14.6k 19 89 118

answered Jan 23 '09 at 13:43
bmotmans
14.3k 5 16 14

14 But, why should encoding be taken into consideration? Why can't I simply get the bytes without having to see what encoding is being used? Even if it were required, shouldn't the String object itself know what encoding is being used and simply dump what is in memory? – Agnel Kurian Jan 23 '09 at 13:48

54 A .NET strings are always encoded as Unicode. So use System.Text.Encoding.Unicode.GetBytes(); to get the set of bytes that .NET would using to represent the characters. However why would you want that? I recommend UTF-8 especially when most characters are in the western latin set. – AnthonyWJones Jan 23 '09 at 14:33

7 Also: the exact bytes used internally in the string *don't matter* if the system that retrieves them doesn't handle that encoding or handles it as the wrong encoding. If it's all within .Net, why convert to an array of bytes at all. Otherwise, it's better to be explicit with your encoding – Joel Coehoorn Jan 23 '09 at 15:42

10 @Joel, Be careful with System.Text.Encoding.Default as it could be different on each machine it is run. That's why it's recommended to always specify an encoding, such as UTF-8. – Ash Jan 28 '10 at 9:01

25 You don't need the encodings unless you (or someone else) actually intend(s) to *interpret* the data, instead of treating it as a generic "block of bytes". For things like compression, encryption, etc., worrying about the encoding is meaningless. See my answer for a way to do this without worrying about the encoding. (I might have given a -1 for saying you need to worry about encodings when you don't, but I'm not feeling particularly mean today. :P) – Mehrdad Apr 30 '12 at 7:55

274

The accepted answer is very, very complicated. Use the included .NET classes for this:

```
const string data = "A string with international characters: Norwegian: ÆØÅæøå, Chinese:
喂 谢谢";
var bytes = System.Text.Encoding.UTF8.GetBytes(data);
var decoded = System.Text.Encoding.UTF8.GetString(bytes);
```

Don't reinvent the wheel if you don't have to...

edited Jul 23 '15 at 14:32

Vlad
15.7k 4 33 63

answered Apr 30 '12 at 7:26

Erik A. Brandstadmoen
8,430 2 31 50

14 In case the accepted answer gets changed, for record purposes, it is Mehrdad's answer at this current time and date. Hopefully the OP will revisit this and accept a better solution. – Thomas Eding Sep 27 '13 at 18:20

6 good in principle but, the encoding should be `System.Text.Encoding.Unicode` to be equivalent to Mehrdad's answer. – Jodrell Nov 25 '14 at 9:08

4 The question has been edited an umptillion times since the original answer, so, maybe my answer is a bit outdates. I never intended to give an exace equivalent to Mehrdad's answer, but give a sensible way of doing it. But, you might be right. However, the phrase "get what bytes the string has been stored in" in the original question is very unprecise. Stored, where? In memory? On disk? If in memory, `System.Text.Encoding.Unicode.GetBytes` would probably be more precise. – Erik A. Brandstadmoen Nov 26 '14 at 11:36

7 @AMissico, your suggestion is buggy, unless you are sure your string is compatible with your system default encoding (string containing only ASCII chars in your system default legacy charset). But nowhere the OP states that. – Frédéric Apr 6 '16 at 20:53

5 @AMissico It can cause the program to give different results *on different systems* though. That's *never* a good thing. Even if it's for making a hash or something (I assume that's what OP means with 'encrypt'), the same string should still always give the same hash. – Nyerguds Apr 22 '16 at 10:33

110

```
BinaryFormatter bf = new BinaryFormatter();
byte[] bytes;
MemoryStream ms = new MemoryStream();

string orig = "喂 Hello 谢谢 Thank You";
bf.Serialize(ms, orig);
ms.Seek(0, 0);
bytes = ms.ToArray();

MessageBox.Show("Original bytes Length: " + bytes.Length.ToString());

MessageBox.Show("Original string Length: " + orig.Length.ToString());

for (int i = 0; i < bytes.Length; ++i) bytes[i] ^= 168; // pseudo encrypt
for (int i = 0; i < bytes.Length; ++i) bytes[i] ^= 168; // pseudo decrypt

BinaryFormatter bfx = new BinaryFormatter();
MemoryStream msx = new MemoryStream();
msx.Write(bytes, 0, bytes.Length);
msx.Seek(0, 0);
string sx = (string)bfx.Deserialize(msx);
```

```
MessageBox.Show("Still intact :" + sx);

MessageBox.Show("Deserialize string Length(still intact): "
    + sx.Length.ToString());

BinaryFormatter bfy = new BinaryFormatter();
MemoryStream msy = new MemoryStream();
bfy.Serialize(msy, sx);
msy.Seek(0, 0);
byte[] bytesy = msy.ToArray();

MessageBox.Show("Deserialize bytes Length(still intact): "
   + bytesy.Length.ToString());
```

edited Jan 26 '09 at 6:29

answered Jan 23 '09 at 16:36

Michael Buen
33.3k 6 81 107

2 You could use the same BinaryFormatter instance for all of those operations – Joel Coehoorn Jan 23 '09 at 17:25

3 Very Interesting. Apparently it will drop any high surrogate Unicode character. See the documentation on [BinaryFormatter] – John Robertson Nov 18 '10 at 18:51

1 @ErikA.Brandstadmoen See my tests here: stackoverflow.com/a/10384024 – Michael Buen May 13 '12 at 11:12

92

You need to take the encoding into account, because 1 character could be represented by 1 **or more** bytes (up to about 6), and different encodings will treat these bytes differently.

Joel has a posting on this:

> The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

answered Jan 23 '09 at 14:03

Zhaph - Ben Duguid
24.7k 4 67 108

6 "1 character could be represented by 1 or more bytes" I agree. I just want those bytes regardless of what encoding the string is in. The only way a string

can be stored in memory is in bytes. Even characters are stored as 1 or more bytes. I merely want to get my hands on them bytes. – Agnel Kurian Jan 23 '09 at 14:07

16 You don't need the encodings unless you (or someone else) actually intend(s) to *interpret* the data, instead of treating it as a generic "block of bytes". For things like compression, encryption, etc., worrying about the encoding is meaningless. See my answer for a way to do this without worrying about the encoding. – Mehrdad Apr 30 '12 at 7:54

9 @Mehrdad - Totally, but the original question, as stated when I initially answered, didn't caveat what OP was going to happen with those bytes after they'd converted them, and for future searchers the information around that is pertinent - this is covered by Joel's answer quite nicely - and as you state within your answer: provided you stick within the .NET world, and use your methods to convert to/from, you're happy. As soon as you step outside of that, encoding will matter. – Zhaph - Ben Duguid Apr 30 '12 at 10:48

86

This is a popular question. It is important to understand what the question author is asking, and that it is different from what is likely the most common need. To discourage misuse of the code where it is not needed, I've answered the later first.

Common Need

Every string has a character set and encoding. When you convert a `System.String` object to an array of `System.Byte` you still have a character set and encoding. **For most usages, you'd know which character set and encoding you need and .NET makes it simple to "copy with conversion."** Just choose the appropriate `Encoding` class.

```
// using System.Text;
Encoding.UTF8.GetBytes(".NET String to byte array")
```

The conversion may need to handle cases where the target character set or encoding doesn't support a character that's in the source. You have some choices: exception, substitution or skipping. The default policy is to substitute a '?'.

```
// using System.Text;
var text = Encoding.ASCII.GetString(Encoding.ASCII.GetBytes("You win €100"));
                                                    // -> "You win ?100"
```

Clearly, conversions are not necessarily lossless!

Note: For `System.String` the source character set is Unicode.

The only confusing thing is that .NET uses the name of a character set for the name of one particular encoding of that character set. `Encoding.Unicode` should be called `Encoding.UTF16` .

That's it for most usages. If that's what you need, stop reading here. See the fun Joel Spolsky article if you don't understand what an encoding is.

Specific Need

Now, the question author asks, "Every string is stored as an array of bytes, right? Why can't I simply have those bytes?"

He doesn't want any conversion.

From the C# spec:

> Character and string processing in C# uses Unicode encoding. The char type represents a UTF-16 code unit, and the string type represents a sequence of UTF-16 code units.

So, we know that if we ask for the null conversion (i.e., from UTF-16 to UTF-16), we'll get the desired result:

```
Encoding.Unicode.GetBytes(".NET String to byte array")
```

But to avoid the mention of encodings, we must do it another way. If an intermediate data type is acceptable, there is a conceptual shortcut for this:

```
".NET String to byte array".ToCharArray()
```

That doesn't get us the desired datatype but Mehrdad's answer shows how to convert this Char array to a Byte array using BlockCopy. However, this copies the string twice! And, it too explicitly uses encoding-specific code: the datatype `System.Char`.

The only way to get to the actual bytes the String is stored in is to use a pointer. The `fixed` statement allows taking the address of values. From the C# spec:

> [For] an expression of type string, ... the initializer computes the address of the first character in the string.

To do so, the compiler writes code skip over the other parts of the string object with `RuntimeHelpers.OffsetToStringData`. So, to get the raw bytes, just create a pointer to the string and copy the number of bytes needed.

```
// using System.Runtime.InteropServices
unsafe byte[] GetRawBytes(String s)
{
    if (s == null) return null;
```

```
    var codeunitCount = s.Length;
    /* We know that String is a sequence of UTF-16 codeunits
       and such codeunits are 2 bytes */
    var byteCount = codeunitCount * 2;
    var bytes = new byte[byteCount];
    fixed(void* pRaw = s)
    {
        Marshal.Copy((IntPtr)pRaw, bytes, 0, byteCount);
    }
    return bytes;
}
```

As @CodesInChaos pointed out, the result depends on the endianness of the machine. But the question author is not concerned with that.

edited May 23 '17 at 10:31 — Community ♦ — 1 1

answered Dec 2 '13 at 4:43 — Tom Blodget — 17.3k 2 28 56

2 @Jan That's correct but string length already gives the number of code-units (not codepoints). – Tom Blodget Feb 4 '14 at 2:35

1 @TomBlodget: Interestingly, if one takes instances of `Globalization.SortKey`, extracts the `KeyData`, and packs the resulting bytes from each into a `String` [two bytes per character, *MSB first*], calling `String.CompareOrdinal` upon the resulting strings will be substantially faster than calling `SortKey.Compare` on the instances of `SortKey`, or even calling `memcmp` on those instances. Given that, I wonder why `KeyData` returns a `Byte[]` rather than a `String`? – supercat Nov 13 '14 at 17:56

1 @TomBlodget: You don't need `fixed` or `unsafe` code, you can also do `var gch = GCHandle.Alloc("foo", GCHandleType.Pinned); var arr = new byte[sizeof(char) * ((string)gch.Target).Length]; Marshal.Copy(gch.AddrOfPinnedObject(), arr, 0, arr.Length); gch.Free();` – Mehrdad Jan 28 '18 at 4:27

44

The first part of your question (how to get the bytes) was already answered by others: look in the `System.Text.Encoding` namespace.

I will address your follow-up question: why do you need to pick an encoding? Why can't you get that from the string class itself?

The answer is in two parts.

First of all, the bytes used internally by the string class *don't matter*, and whenever you assume they do you're likely introducing a bug.

If your program is entirely within the .Net world then you don't need to worry about getting byte arrays for strings at all, even if you're sending data across a network. Instead, use .Net Serialization to worry about transmitting the data. You don't worry about the actual bytes any more: the Serialization formatter does it for you.

On the other hand, what if you are sending these bytes somewhere that you can't guarantee will pull in data from a .Net serialized stream? In this case you definitely do need to worry about encoding, because obviously this external system cares. So again, the internal bytes used by the string don't matter: you need to pick an encoding so you can be explicit about this encoding on the receiving end, even if it's the same encoding used internally by .Net.

I understand that in this case you might prefer to use the actual bytes stored by the string variable in memory where possible, with the idea that it might save some work creating your byte stream. However, I put it to you it's just not important compared to making sure that your output is understood at the other end, and to guarantee that you *must* be explicit with your encoding. Additionally, if you really want to match your internal bytes, you can already just choose the `Unicode` encoding, and get that performance savings.

Which brings me to the second part... picking the `Unicode` encoding ***is*** telling .Net to use the underlying bytes. You do need to pick this encoding, because when some new-fangled Unicode-Plus comes out the .Net runtime needs to be free to use this newer, better encoding model without breaking your program. But, for the moment (and forseeable future), just choosing the Unicode encoding gives you what you want.

It's also important to understand your string has to be re-written to wire, and that involves at least some translation of the bit-pattern *even when you use a matching encoding*. The computer needs to account for things like Big vs Little Endian, network byte order, packetization, session information, etc.

edited Sep 25 '17 at 21:13

answered Jan 23 '09 at 15:54

Joel Coehoorn

323k 100 510 744

9 There are areas in .NET where you do have to get byte arrays for strings. Many of the .NET Cryptrography classes contain methods such as ComputeHash() that accept byte array or stream. You have no alternative but to convert a string to a byte array first (choosing an Encoding) and then optionally wrap it in a stream. However as long as you choose an encoding (ie UTF8) an stick with it there are no problems with this. – Ash Jan 28 '10 at 9:33

43

Just to demonstrate that Mehrdrad's sound answer works, his approach can even persist the unpaired surrogate characters(of which many had leveled against my answer, but of which everyone are equally guilty of, e.g. `System.Text.Encoding.UTF8.GetBytes` , `System.Text.Encoding.Unicode.GetBytes` ; those encoding methods can't persist the high surrogate characters `d800` for example, and those just merely replace high surrogate characters with value `fffd`) :

```
using System;

class Program
{
    static void Main(string[] args)
```

```
    {
        string t = "爱虫";
        string s = "Test\ud800Test";

        byte[] dumpToBytes = GetBytes(s);
        string getItBack = GetString(dumpToBytes);

        foreach (char item in getItBack)
        {
            Console.WriteLine("{0} {1}", item, ((ushort)item).ToString("x"));
        }
    }

    static byte[] GetBytes(string str)
    {
        byte[] bytes = new byte[str.Length * sizeof(char)];
        System.Buffer.BlockCopy(str.ToCharArray(), 0, bytes, 0, bytes.Length);
        return bytes;
    }

    static string GetString(byte[] bytes)
    {
        char[] chars = new char[bytes.Length / sizeof(char)];
        System.Buffer.BlockCopy(bytes, 0, chars, 0, bytes.Length);
        return new string(chars);
    }
}
```

Output:

```
T 54
e 65
s 73
t 74
? d800
T 54
e 65
s 73
t 74
```

Try that with **System.Text.Encoding.UTF8.GetBytes** or **System.Text.Encoding.Unicode.GetBytes**, they will merely replace high surrogate characters with value **fffd**

Every time there's a movement in this question, I'm still thinking of a serializer(be it from Microsoft or from 3rd party component) that can persist strings even it contains unpaired surrogate characters; I google this every now and then: **serialization unpaired surrogate**

character .NET. This doesn't make me lose any sleep, but it's kind of annoying when every now and then there's somebody commenting on my answer that it's flawed, yet their answers are equally flawed when it comes to unpaired surrogate characters.

Darn, Microsoft should have just used `System.Buffer.BlockCopy` in its `BinaryFormatter` ツ

谢谢！

edited May 23 '17 at 12:18

community wiki
5 revs
Michael Buen

3 Don't surrogates have to appear in pairs to form valid code points? If that's the case, I can understand why the data would be mangled. – dtanders Jun 14 '12 at 14:27

1 @dtanders Yes,that's my thoughts too, they have to appear in pairs, unpaired surrogate characters just happen if you deliberately put them on string and make them unpaired. What I don't know is why other devs keep on harping that we should use encoding-aware approach instead, as they deemed the serialization approach(my answer,which was an accepted answer for more than 3 years) doesn't keep the unpaired surrogate character intact. But they forgot to check that their encoding-aware solutions doesn't keep the unpaired surrogate character too,the irony ツ – Michael Buen Jun 14 '12 at 23:23

2 @MichaelBuen It seem to me that the main issue is that you are in big bold letters saying something doesn't matter, rather than saying that it does not matter in their case. As a result, you are encouraging people who look at your answer to make basic programming mistakes which will cause others frustration in the future. Unpaired surrogates are invalid in a string. It is not a char array, so it makes sense that converting a string to another format would result in an error `FFFD` on that character. If you want to do manual string manipulation, use a char[] as recommended. – Trisped Nov 11 '14 at 20:06

2 @dtanders: A `System.String` is an immutable sequence of `Char`; .NET has always allowed a `String` object to be constructed from any `Char[]` and export its content to a `Char[]` containing the same values, even if the original `Char[]` contains unpaired surrogates. – supercat Nov 12 '14 at 21:57

Try this, a lot less code:

39

```
System.Text.Encoding.UTF8.GetBytes("TEST String");
```

edited Apr 24 '15 at 9:58

Peter Mortensen
14.6k 19 89 118

answered Jul 25 '11 at 22:52

Nathan
531 1 5 8

7 @mg30rg: Why do you think your example is strange? Surely in a variable-width encoding not all characters have the same byte lengthes. What's

wrong with it? – Vlad Feb 25 '18 at 1:18

25

Well, I've read all answers and they were about using encoding or one about serialization that drops unpaired surrogates.

It's bad when the string, for example, comes from SQL Server where it was built from a byte array storing, for example, a password hash. If we drop anything from it, it'll store an invalid hash, and if we want to store it in XML, we want to leave it intact (because the XML writer drops an exception on any unpaired surrogate it finds).

So I use Base64 encoding of byte arrays in such cases, but hey, on the Internet there is only one solution to this in C#, and it has bug in it and is only one way, so I've fixed the bug and written back procedure. Here you are, future googlers:

```
public static byte[] StringToBytes(string str)
{
    byte[] data = new byte[str.Length * 2];
    for (int i = 0; i < str.Length; ++i)
    {
        char ch = str[i];
        data[i * 2] = (byte)(ch & 0xFF);
        data[i * 2 + 1] = (byte)((ch & 0xFF00) >> 8);
    }

    return data;
}

public static string StringFromBytes(byte[] arr)
{
    char[] ch = new char[arr.Length / 2];
    for (int i = 0; i < ch.Length; ++i)
    {
        ch[i] = (char)((int)arr[i * 2] + (((int)arr[i * 2 + 1]) << 8));
    }
    return new String(ch);
}
```

edited Mar 9 '17 at 8:55

Tshilidzi Mudau

3,532 2 25 35

answered Mar 10 '11 at 8:57

Gman

1,197 1 16 31

Also please explain why encoding should be taken into consideration. Can't I simply get what bytes the string has been stored in?

23

> Why this dependency on encoding?!!!

Because there is no such thing as "the bytes of the string".

A string (or more generically, a text) is composed of characters: letters, digits, and other symbols. That's all. Computers, however, do not know anything about characters; they can only handle bytes. Therefore, if you want to store or transmit text by using a computer, you need to transform the characters to bytes. How do you do that? Here's where encodings come to the scene.

An encoding is nothing but a convention to translate logical characters to physical bytes. The simplest and best known encoding is ASCII, and it is all you need if you write in English. For other languages you will need more complete encodings, being any of the Unicode flavours the safest choice nowadays.

So, in short, trying to "get the bytes of a string without using encodings" is as impossible as "writing a text without using any language".

By the way, I strongly recommend you (and anyone, for that matter) to read this small piece of wisdom: The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

edited Oct 23 '15 at 6:19

answered Jul 16 '09 at 11:45

Konamiman
44.3k 15 102 128

2 Allow me to clarify: An encoding has been used to translate "hello world" to physical bytes. Since the string is stored on my computer, I am sure that it must be stored in bytes. I merely want to access those bytes to save them on disk or for any other reason. I do not want to interpret these bytes. Since I do not want to interpret these bytes, the need for an encoding at this point is as misplaced as requiring a phone line to call printf. – Agnel Kurian Jul 16 '09 at 15:30

3 But again, there is no concept of text-to-physical-bytes-translation unless yo use an encoding. Sure, the compiler stores the strings somehow in memory - but it is just using an internal encoding, which you (or anyone except the compiler developer) do not know. So, whatever you do, you need an encoding to get physical bytes from a string. – Konamiman Jul 22 '09 at 8:35

2 @Gnafoo, A copy of the bytes will do. – Agnel Kurian May 14 '11 at 5:06

22

C# to convert a `string` to a `byte` array:

```
public static byte[] StrToByteArray(string str)
{
   System.Text.UTF8Encoding  encoding=new System.Text.UTF8Encoding();
   return encoding.GetBytes(str);
}
```

edited Aug 12 '16 at 18:39

iliketocode

5,759 4 33 49

answered Jun 5 '13 at 10:52

Shyam sundar shah

1,994 19 32

17

You can use the following code for conversion between string and byte array.

```
string s = "Hello World";

// String to Byte[]

byte[] byte1 = System.Text.Encoding.Default.GetBytes(s);

// OR

byte[] byte2 = System.Text.ASCIIEncoding.Default.GetBytes(s);

// Byte[] to string

string str = System.Text.Encoding.UTF8.GetString(byte1);
```

answered Sep 9 '14 at 11:30

Jarvis Stark

553 4 11

16

```
byte[] strToByteArray(string str)
{
    System.Text.ASCIIEncoding enc = new System.Text.ASCIIEncoding();
    return enc.GetBytes(str);
}
```

answered Jan 23 '09 at 13:43

gkrogers

7,053 2 25 33

5 This doesn't always work. Some special characters can get lost in using such a method I've found the hard way. – JB King Jan 23 '09 at 17:14

1 if the charset was utf it wouldn't work! – ahmadali shafiee Sep 18 '12 at 6:27

14

With the advent of `Span<T>` released with C# 7.2, the canonical technique to capture the underlying memory representation of a string into a managed byte array is:

```
byte[] bytes = "rubbish_\u9999_string".AsSpan().AsBytes().ToArray();
```

Converting it back should be a non-starter because that means you are in fact interpreting the data somehow, but for the sake of completeness:

```
string s;
unsafe
{
    fixed (char* f = &bytes.AsSpan().NonPortableCast<byte, char>
().DangerousGetPinnableReference())
    {
        s = new string(f);
    }
}
```

The names `NonPortableCast` and `DangerousGetPinnableReference` should further the argument that you probably shouldn't be doing this.

Note that working with `Span<T>` requires installing the System.Memory NuGet package.

Regardless, the *actual* original question and follow-up comments imply that the underlying memory is not being "interpreted" (which I assume means is not modified or read beyond the need to write it as-is), indicating that some implementation of the `Stream` class should be used instead of reasoning about the data as strings at all.

answered Jan 10 '18 at 20:21

John Rasch
51.6k 18 97 135

13

I'm not sure, but I think the string stores its info as an array of Chars, which is inefficient with bytes. Specifically, the definition of a Char is "Represents a Unicode character".

take this example sample:

```
String str = "asdf éß";
String str2 = "asdf gh";
EncodingInfo[] info =  Encoding.GetEncodings();
foreach (EncodingInfo enc in info)
{
    System.Console.WriteLine(enc.Name + " - "
      + enc.GetEncoding().GetByteCount(str)
      + enc.GetEncoding().GetByteCount(str2));
}
```

Take note that the Unicode answer is 14 bytes in both instances, whereas the UTF-8 answer is only 9 bytes for the first, and only 7 for the second.

So if you just want the bytes used by the string, simply use `Encoding.Unicode`, but it will be inefficient with storage space.

edited Aug 12 '16 at 18:38

iliketocode
5,759 4 33 49

answered Jan 23 '09 at 14:34

Ed Marty
35.8k 18 93 148

11

The key issue is that a glyph in a string takes 32 bits (16 bits for a character code) but a byte only has 8 bits to spare. A one-to-one mapping doesn't exist unless you restrict yourself to strings that only contain ASCII characters. System.Text.Encoding has lots of ways to map a string to byte[], you need to pick one that avoids loss of information and that is easy to use by your client when she needs to map the byte[] back to a string.

Utf8 is a popular encoding, it is compact and not lossy.

answered Jan 23 '09 at 14:15

Hans Passant
818k 114 1415 2201

3 UTF-8 is compact only if the majority of your characters are in the English (ASCII) character set. If you had a long string of Chinese characters, UTF-16 would be a more compact encoding than UTF-8 for that string. This is because UTF-8 uses one byte to encode ASCII, and 3 (or maybe 4) otherwise. – Joel Mueller Jan 23 '09 at 20:40

7 True. But, how can you not know about encoding if you're familiar with handling Chinese text? – Hans Passant Jan 24 '09 at 3:40

Use:

9

```
string text = "string";
byte[] array = System.Text.Encoding.UTF8.GetBytes(text);
```

The result is:

```
[0] = 115
[1] = 116
[2] = 114
[3] = 105
[4] = 110
[5] = 103
```

edited Jan 9 '17 at 1:22

Peter Mortensen
14.6k 19 89 118

answered Oct 22 '13 at 12:55

mashet
564 1 8 14

8

Fastest way

```
public static byte[] GetBytes(string text)
{
    return System.Text.ASCIIEncoding.UTF8.GetBytes(text);
}
```

EDIT as Makotosan commented this is now the best way:

```
Encoding.UTF8.GetBytes(text)
```

edited Aug 4 '16 at 10:31

answered Mar 22 '10 at 8:40

Alessandro Annini
1,181 1 12 28

8 ASCIIEncoding..... is not needed. Simply using Encoding.UTF8.GetBytes(text) is preferred. – Makotosan Feb 17 '12 at 20:40

8

> How do I convert a string to a byte[] in .NET (C#) without manually specifying a specific encoding?

A string in .NET represents text as a sequence of UTF-16 code units, so the bytes are encoded in memory in UTF-16 already.

Mehrdad's Answer

You can use Mehrdad's answer, but it does actually use an encoding because chars are UTF-16. It calls ToCharArray which looking at the source creates a `char[]` and copies the memory to it directly. Then it copies the data to a byte array that is also allocated. So under the hood it is copying the underlying bytes *twice* and allocating a char array that is not used after the call.

Tom Blodget's Answer

Tom Blodget's answer is 20-30% faster than Mehrdad since it skips the intermediate step of allocating a char array and copying the bytes to it, but it requires you compile with the `/unsafe` option. If you absolutely do not want to use encoding, I think this is the way to go. If you put your encryption login inside the `fixed` block, you don't even need to allocate a separate byte array and copy the bytes to it.

> Also, why should encoding be taken into consideration? Can't I simply get what bytes the string has been stored in? Why is there a dependency on character encodings?

Because that is the proper way to do it. `string` is an abstraction.

Using an encoding could give you trouble if you have 'strings' with invalid characters, but that shouldn't happen. If you are getting data into your string with invalid characters you are doing it wrong. You should probably be using a byte array or a Base64 encoding to start with.

If you use `System.Text.Encoding.Unicode` , your code will be more resilient. You don't have to worry about the endianness of the system your code will be running on. You don't need to worry if the next version of the CLR will use a different internal character encoding.

I think the question isn't why you want to worry about the encoding, but why you want to ignore it and use something else. Encoding is meant to represent the abstraction of a string in a sequence of bytes. `System.Text.Encoding.Unicode` will give you a little endian byte order encoding and will perform the same on every system, now and in the future.

answered Jul 2 '18 at 20:51

Jason Goemaat
22k 9 69 97

7

The closest approach to the OP's question is Tom Blodget's, which actually goes into the object and extracts the bytes. I say closest because it depends on implementation of the String Object.

> "Can't I simply get what bytes the string has been stored in?"

Sure, but that's where the fundamental error in the question arises. The String is an object which could have an interesting data structure. We already know it does, because it allows unpaired surrogates to be stored. It might store the length. It might keep a pointer to each of the 'paired' surrogates allowing quick counting. Etc. All of these extra bytes are not part of the character data.

What you want is each character's bytes in an array. And that is where 'encoding' comes in. By default you will get UTF-16LE. If you don't care about the bytes themselves except for the round trip then you can choose any encoding including the 'default', and convert it back later (assuming the same parameters such as what the default encoding was, code points, bug fixes, things allowed such as unpaired surrogates, etc.

But why leave the 'encoding' up to magic? Why not specify the encoding so that you know what bytes you are gonna get?

> "Why is there a dependency on character encodings?"

Encoding (in this context) simply means the bytes that represent your string. Not the bytes of the string object. You wanted the bytes the string has been stored in -- this is where the question was asked naively. You wanted the bytes of string in a contiguous array that represent the string, and not all of the other binary data that a string object may contain.

Which means how a string is stored is irrelevant. You want a string "Encoded" into bytes in a byte array.

I like Tom Bloget's answer because he took you towards the 'bytes of the string object' direction. It's implementation dependent though, and because he's peeking at internals it might be difficult to reconstitute a copy of the string.

Mehrdad's response is wrong because it is misleading at the conceptual level. You still have a list of bytes, encoded. His particular solution allows for unpaired surrogates to be preserved -- this is implementation dependent. His particular solution would not produce the string's bytes accurately if `GetBytes` returned the string in UTF-8 by default.

I've changed my mind about this (Mehrdad's solution) -- this isn't getting the bytes of the string; rather it is getting the bytes of the character array that was created from the string. Regardless of encoding, the char datatype in c# is a fixed size. This allows a consistent length byte array to be produced, and it allows the character array to be reproduced based on the size of the byte array. So if the encoding were UTF-8, but each char was 6 bytes to accommodate the largest utf8 value, it would still work. So indeed -- encoding of the character does not matter.

But a conversion was used -- each character was placed into a fixed size box (c#'s character type). However what that representation is does not matter, which is technically the answer to the OP. So -- if you are going to convert anyway... Why not 'encode'?

edited Nov 1 '17 at 19:44

answered Aug 18 '15 at 17:04

Gerard ONeill

2,613 25 20

1 Ok, then I think you are not understanding the problem. We know it is a unicode compliant array -- in fact, because it is .net, we know it is UTF-16. So those characters will not exist there. You also didn't fully read my comment about internal representations changing. A String is an object, not an encoded byte array. So I'm going to disagree with your last statement. You want code to convert all unicode strings to any UTF encoding. This does what you want, correctly. – Gerard ONeill Feb 11 '16 at 22:17

6

You can use following code to convert a `string` to a `byte array` in .NET

```
string s_unicode = "abcéabc";
byte[] utf8Bytes = System.Text.Encoding.UTF8.GetBytes(s_unicode);
```

edited May 2 '14 at 7:39

İlker Elçora

413 4 10

answered Sep 2 '13 at 11:21

Shyam sundar shah

1,994 19 32

3

If you really want a copy of the underlying bytes of a string, you can use a function like the one that follows. **However, you shouldn't** please read on to find out why.

```
[DllImport(
        "msvcrt.dll",
        EntryPoint = "memcpy",
        CallingConvention = CallingConvention.Cdecl,
        SetLastError = false)]
private static extern unsafe void* UnsafeMemoryCopy(
    void* destination,
    void* source,
    uint count);

public static byte[] GetUnderlyingBytes(string source)
{
    var length = source.Length * sizeof(char);
```

```
    var result = new byte[length];
    unsafe
    {
        fixed (char* firstSourceChar = source)
        fixed (byte* firstDestination = result)
        {
            var firstSource = (byte*)firstSourceChar;
            UnsafeMemoryCopy(
                firstDestination,
                firstSource,
                (uint)length);
        }
    }

    return result;
}
```

This function will get you a copy of the bytes underlying your string, pretty quickly. You'll get those bytes in whatever way they are encoding on your system. This encoding is almost certainly UTF-16LE but that is an implementation detail you shouldn't have to care about.

It would be **safer, simpler and more reliable** to just call,

```
System.Text.Encoding.Unicode.GetBytes()
```

In all likelihood this will give the same result, is easier to type, and the bytes will always round-trip with a call to

```
System.Text.Encoding.Unicode.GetString()
```

answered Nov 25 '14 at 10:29

Jodrell

27.9k 3 61 107

3

Here is my unsafe implementation of `String` to `Byte[]` conversion:

```
public static unsafe Byte[] GetBytes(String s)
{
    Int32 length = s.Length * sizeof(Char);
    Byte[] bytes = new Byte[length];
```

```
fixed (Char* pInput = s)
fixed (Byte* pBytes = bytes)
{
    Byte* source = (Byte*)pInput;
    Byte* destination = pBytes;

    if (length >= 16)
    {
        do
        {
            *((Int64*)destination) = *((Int64*)source);
            *((Int64*)(destination + 8)) = *((Int64*)(source + 8));

            source += 16;
            destination += 16;
        }
        while ((length -= 16) >= 16);
    }

    if (length > 0)
    {
        if ((length & 8) != 0)
        {
            *((Int64*)destination) = *((Int64*)source);

            source += 8;
            destination += 8;
        }

        if ((length & 4) != 0)
        {
            *((Int32*)destination) = *((Int32*)source);

            source += 4;
            destination += 4;
        }

        if ((length & 2) != 0)
        {
            *((Int16*)destination) = *((Int16*)source);

            source += 2;
            destination += 2;
        }

        if ((length & 1) != 0)
        {
            ++source;
            ++destination;
```

```
                destination[0] = source[0];
            }
        }
    }

    return bytes;
}
```

It's way faster than the accepted anwser's one, even if not as elegant as it is. Here are my Stopwatch benchmarks over 10000000 iterations:

```
[Second String: Length 20]
Buffer.BlockCopy: 746ms
Unsafe: 557ms

[Second String: Length 50]
Buffer.BlockCopy: 861ms
Unsafe: 753ms

[Third String: Length 100]
Buffer.BlockCopy: 1250ms
Unsafe: 1063ms
```

In order to use it, you have to tick "Allow Unsafe Code" in your project build properties. As per .NET Framework 3.5, this method can also be used as String extension:

```
public static unsafe class StringExtensions
{
    public static Byte[] ToByteArray(this String s)
    {
        // Method Code
    }
}
```

edited Aug 12 '16 at 18:38

iliketocode

5,759 4 33 49

answered Jan 15 '13 at 11:43

Tommaso Belluzzo

18.4k 7 54 83

Two ways:

2

```
public static byte[] StrToByteArray(this string s)
{
    List<byte> value = new List<byte>();
    foreach (char c in s.ToCharArray())
        value.Add(c.ToByte());
    return value.ToArray();
}
```

And,

```
public static byte[] StrToByteArray(this string s)
{
    s = s.Replace(" ", string.Empty);
    byte[] buffer = new byte[s.Length / 2];
    for (int i = 0; i < s.Length; i += 2)
        buffer[i / 2] = (byte)Convert.ToByte(s.Substring(i, 2), 16);
    return buffer;
}
```

I tend to use the bottom one more often than the top, haven't benchmarked them for speed.

answered Feb 19 '09 at 21:03
harmonik

4 What about multibyte characters? – Agnel Kurian Feb 23 '09 at 9:57

2

```
bytes[] buffer = UnicodeEncoding.UTF8.GetBytes(string something); //for converting to
UTF then get its bytes

bytes[] buffer = ASCIIEncoding.ASCII.GetBytes(string something); //for converting to
ascii then get its bytes
```

answered Jan 2 '12 at 11:07

user1120193
199 2 11

simple code with LINQ

2

```
string s = "abc"
byte[] b = s.Select(e => (byte)e).ToArray();
```

EDIT : as commented below, it is not a good way.

but you can still use it to understand LINQ with a more appropriate coding :

```
string s = "abc"
byte[] b = s.Cast<byte>().ToArray();
```

edited Dec 18 '13 at 10:13

answered Oct 11 '12 at 9:45

Avlin

395 4 18

2 It's hardly *more faster*, let alone *most fastest*. It's certainly an interesting alternative, but it's essentially the same as `Encoding.Default.GetBytes(s)` which, by the way, is *way faster*. Quick testing suggests that `Encoding.Default.GetBytes(s)` performs at least 79% faster. YMMV. – WynandB Oct 25 '13 at 4:36

5 Try it with a `€` . This code will not crash, but will return a **wrong result** (which is even worse). Try casting to a `short` instead of `byte` to see the difference. – Hans Kesting Dec 18 '13 at 8:57

Simply use this:

2

```
byte[] myByte= System.Text.ASCIIEncoding.Default.GetBytes(myString);
```

edited Jul 1 '15 at 1:14

jonsca

8,845 11 50 58

answered Jun 30 '15 at 14:39

alireza amini

1,239 1 12 31

The string can be converted to byte array in few different ways, due to the following fact: .NET supports Unicode, and Unicode standardizes several difference encodings called UTFs. They have different lengths of byte representation but are equivalent in that

2

sense that when a string is encoded, it can be coded back to the string, but if the string is encoded with one UTF and decoded in the assumption of different UTF if can be screwed up.

Also, .NET supports non-Unicode encodings, but they are not valid in general case (will be valid only if a limited sub-set of Unicode code point is used in an actual string, such as ASCII). Internally, .NET supports UTF-16, but for stream representation, UTF-8 is usually used. It is also a standard-de-facto for Internet.

Not surprisingly, serialization of string into an array of byte and deserialization is supported by the class `System.Text.Encoding`, which is an abstract class; its derived classes support concrete encodings: `ASCIIEncoding` and four UTFs (`System.Text.UnicodeEncoding` supports UTF-16)

Ref this link.

For serialization to an array of bytes using `System.Text.Encoding.GetBytes`. For the inverse operation use `System.Text.Encoding.GetChars`. This function returns an array of characters, so to get a string, use a string constructor `System.String(char[])`.
Ref this page.

Example:

```
string myString = //... some string

System.Text.Encoding encoding = System.Text.Encoding.UTF8; //or some other, but prefer
some UTF is Unicode is used
byte[] bytes = encoding.GetBytes(myString);

//next lines are written in response to a follow-up questions:

myString = new string(encoding.GetChars(bytes));
byte[] bytes = encoding.GetBytes(myString);
myString = new string(encoding.GetChars(bytes));
byte[] bytes = encoding.GetBytes(myString);

//how many times shall I repeat it to show there is a round-trip? :-)
```

edited Aug 17 '17 at 7:33

Bharat Mane
241 9 19

answered Jun 11 '14 at 11:29

Vijay Singh Rana
695 9 29

It depends on what you want the bytes FOR

2

This is because, as Tyler so aptly said, "Strings aren't pure data. They also have information." In this case, the information is an encoding that was assumed when the string was created.

Assuming that you have binary data (rather than text) stored in a string

This is based off of OP's comment on his own question, and is the correct question if I understand OP's hints at the use-case.

Storing binary data in strings is probably the wrong approach because of the assumed encoding mentioned above! Whatever program or library stored that binary data in a `string` (instead of a `byte[]` array which would have been more appropriate) has already lost the battle before it has begun. If they are sending the bytes to you in a REST request/response or anything that *must* transmit strings, Base64 would be the right approach.

If you have a text string with an unknown encoding

Everybody else answered this incorrect question incorrectly.

If the string looks good as-is, just pick an encoding (preferably one starting with UTF), use the corresponding `System.Text.Encoding.???.GetBytes()` function, and tell whoever you give the bytes to which encoding you picked.

edited Nov 8 '17 at 19:15

answered Nov 8 '17 at 18:21

NH.

1,106 1 15 31

protected by Paŭlo Ebermann Jun 27 '13 at 19:25

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?