# Why .NET String is immutable? [duplicate]

Asked 14 years, 9 months ago  Modified 6 years, 6 months ago

Viewed 118k times

202

**This question already has answers here**:

Why can't strings be mutable in Java and .NET? (17 answers)

Closed 11 years ago.

As we all know, String is immutable. What are the reasons for String being immutable and the introduction of StringBuilder class as mutable?

c# .net string immutability

Share

Improve this question

Follow

edited Mar 3, 2012 at 22:10

user

5,401 ● 7 ● 52 ● 65

asked Mar 2, 2010 at 17:33

Nirajan Singh

2,885 ● 4 ● 25 ● 25

9 similar question: stackoverflow.com/questions/93091/… – ram Mar 2, 2010 at 17:36

## 13 Answers

Sorted by: Highest score (default)

260

1. Instances of immutable types are inherently thread-safe, since no thread can modify it, the risk of a thread modifying it in a way that interferes with another is removed (the reference itself is a different matter).
2. Similarly, the fact that aliasing can't produce changes (if x and y both refer to the same object a change to x entails a change to y) allows for considerable compiler optimisations.
3. Memory-saving optimisations are also possible. Interning and atomising being the most obvious examples, though we can do other versions of the same principle. I once produced a memory saving of about half a GB by comparing immutable objects and replacing references to duplicates so that they all pointed to the same instance (time-consuming, but a minute's extra start-up to save a massive amount of memory was a performance win in the case in question). With mutable objects that can't be done.
4. No side-effects can come from passing an immutable type as a method to a parameter unless it is `out` or `ref` (since that changes the reference, not the object). A programmer therefore knows that if `string`

`x = "abc"` at the start of a method, and that doesn't change in the body of the method, then `x == "abc"` at the end of the method.

5. Conceptually, the semantics are more like value types; in particular equality is based on state rather than identity. This means that `"abc" == "ab" + "c"`. While this doesn't require immutability, the fact that a reference to such a string will always equal "abc" throughout its lifetime (which does require immutability) makes uses as keys where maintaining equality to previous values is vital, much easier to ensure correctness of (strings are indeed commonly used as keys).

6. Conceptually, it can make more sense to be immutable. If we add a month onto Christmas, we haven't changed Christmas, we have produced a new date in late January. It makes sense therefore that `Christmas.AddMonths(1)` produces a new `DateTime` rather than changing a mutable one. (Another example, if I as a mutable object change my name, what has changed is which name I am using, "Jon" remains immutable and other Jons will be unaffected.

7. Copying is fast and simple, to create a clone just `return this`. Since the copy can't be changed anyway, pretending something is its own copy is safe.

8. [Edit, I'd forgotten this one]. Internal state can be safely shared between objects. For example, if you

were implementing list which was backed by an array, a start index and a count, then the most expensive part of creating a sub-range would be copying the objects. However, if it was immutable then the sub-range object could reference the same array, with only the start index and count having to change, with a **very** considerable change to construction time.

In all, for objects which don't have undergoing change as part of their purpose, there can be many advantages in being immutable. The main disadvantage is in requiring extra constructions, though even here it's often overstated (remember, you have to do several appends before StringBuilder becomes more efficient than the equivalent series of concatenations, with their inherent construction).

It would be a disadvantage if mutability was part of the purpose of an object (who'd want to be modeled by an Employee object whose salary could never ever change) though sometimes even then it can be useful (in a many web and other stateless applications, code doing read operations is separate from that doing updates, and using different objects may be natural - I wouldn't make an object immutable and then force that pattern, but if I already had that pattern I might make my "read" objects immutable for the performance and correctness-guarantee gain).

Copy-on-write is a middle ground. Here the "real" class holds a reference to a "state" class. State classes are shared on copy operations, but if you change the state, a new copy of the state class is created. This is more often used with C++ than C#, which is why it's std:string enjoys some, but not all, of the advantages of immutable types, while remaining mutable.

Share Improve this answer

Follow

edited Mar 25, 2016 at 23:34

Stephen Byrne

**7,465** ● 1 ● 33 ● 52

answered Aug 7, 2010 at 1:10

Jon Hanna

**113k** ● 10 ● 149 ● 257

2 @IanBoyd Yes, but whether it's a good middle-ground or a worse-of-both-worlds middle-ground is another question. Not really one to get into in detail here, but drdobbs.com/cpp/184403779 has an interesting critique of how COW is used in the STLs string type. Interestingly enough the conclusion is that it could be better to have separate mutable and immutable types, which of course is exactly what we're talking about here. – Jon Hanna Feb 2, 2012 at 10:27

Another advantage of immutable strings (and immutable classes in general, as well as plain-old-data structs) is that it there's no question of whether a routine which accepts an immutable string (or POD struct) is semantically capturing the value at that time it's called. By contrast, if one passes a mutable object to a "SetAttribute" function and subsequently changes it, it may be unclear whether that change will not affect the attribute, will affect it in the "expected" manner, or

will break it in some unexpected manner. – supercat Mar 4, 2012 at 1:02

Note that the C++ standard explicitly forbids using COW semantics for `std::string` since C++11. – Max Truxa Jun 17, 2016 at 6:33

1 @JonHanna This answer seems to explain the benefits of immutable types but doesn't really explain why String is immutable. Why aren't certain other .NET reference types immutable? Can you please go into more detail into why String itself is immutable? – Howiecamp Jul 29, 2017 at 23:08

2 String is immutable because the framework designers decided it should be. **Probably** because of the things listed in this answer. – Lasse V. Karlsen Feb 18, 2018 at 8:43

79

Making strings immutable has many advantages. It provides automatic thread safety, and makes strings behave like an intrinsic type in a simple, effective manner. It also allows for extra efficiencies at runtime (such as allowing effective string interning to reduce resource usage), and has huge security advantages, since it's impossible for an third party API call to change your strings.

StringBuilder was added in order to address the one major disadvantage of immutable strings - runtime construction of immutable types causes a lot of GC pressure and is inherently slow. By making an explicit, mutable class to handle this, this issue is addressed without adding unneeded complication to the string class.

Share Improve this answer

Follow

answered Mar 2, 2010 at 17:37

Reed Copsey
564k ● 80 ● 1.2k ● 1.4k

7 Will have to chime in here that immutability is not inherently slow, even if the particular implementation of the string class is. Strings don't have to be implemented as an array of chars, its wholly possible to implement strings as immutable ropes which have the interesting property of O(1) concats and O(lg n) substrings. – Juliet Aug 7, 2010 at 0:19

8 @Juliet: But you also trade off there -- you get O(1) concat and lg n substrings, but you lose constant time element access and you lose cache locality. There is a reason strings aren't typically implemented like ropes. – Billy ONeal Aug 7, 2010 at 1:45

Give your strings to System.Reflection and friends, and we will see how much they are impossible to change.who says something like that in somewhere like this? – Behrooz Feb 4, 2013 at 22:45

25

Strings are not really immutable. They are just publicly immutable. It means you cannot modify them from their public interface. But in the inside the are actually mutable.

If you don't believe me look at the `String.Concat` definition using reflector. The last lines are...

```
int length = str0.Length;
string dest = FastAllocateString(length + str1.Length)
FillStringChecked(dest, 0, str0);
FillStringChecked(dest, length, str1);
return dest;
```

As you can see the `FastAllocateString` returns an empty but allocated string and then it is modified by `FillStringChecked`

Actually the `FastAllocateString` is an extern method and the `FillStringChecked` is unsafe so it uses pointers to copy the bytes.

Maybe there are better examples but this is the one I have found so far.

Share Improve this answer

Follow

edited Jun 23, 2018 at 22:03

answered Aug 7, 2010 at 1:41

Carlos Muñoz

**17.8k** ● 8 ● 57 ● 81

2 See more on this in this blogpost blog.getpaint.net/2015/07/21/… – Jan Nov 15, 2016 at 22:11

---

16

string management is an expensive process. keeping strings immutable allows repeated strings to be reused, rather than re-created.

Share Improve this answer

Follow

answered Mar 2, 2010 at 17:34

kolosy

**3,099** ● 3 ● 30 ● 49

1 That's half the Java reason, but there's a half-dozen in .Net, security being another big, big one. – Nick Craver Mar 2, 2010 at 17:35

2 Ahh.. so that's why string is reference type either that value type.. actually it is a big question for me **if string is immutable why don't use value type..?** thanks anyway. – ktutnik Aug 8, 2010 at 12:31

6 @up: do you think passing ~100 MB (or even more large) string over stack would be good? – apocalypse Feb 28, 2012 at 12:15

14

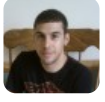
[Why are string types immutable in C#]

> String is a reference type, so it is never copied, but passed by reference. Compare this to the C++ std::string object (which is not immutable), which is passed by value. This means that if you want to use a String as a key in a Hashtable, you're fine in C++, because C++ will copy the string to store the key in the hashtable (actually std::hash_map, but still) for later comparison. So even if you later modify the std::string instance, you're fine. But in .Net, when you use a String in a Hashtable, it will store a reference to that instance. Now assume for a moment that strings aren't immutable, and see what happens: 1. Somebody inserts a value x with key "hello" into a Hashtable. 2. The Hashtable computes the hash value for the String, and places a reference

> to the string and the value x in the appropriate bucket. 3. The user modifies the String instance to be "bye". 4. Now somebody wants the value in the hashtable associated with "hello". It ends up looking in the correct bucket, but when comparing the strings it says "bye"!="hello", so no value is returned. 5. Maybe somebody wants the value "bye"? "bye" probably has a different hash, so the hashtable would look in a different bucket. No "bye" keys in that bucket, so our entry still isn't found.
>
> Making strings immutable means that step 3 is impossible. If somebody modifies the string he's creating a new string object, leaving the old one alone. Which means the key in the hashtable is still "hello", and thus still correct.
>
> So, probably among other things, immutable strings are a way to enable strings that are passed by reference to be used as keys in a hashtable or similar dictionary object.

Share Improve this answer Follow

answered Mar 2, 2010 at 17:38



NebuSoft

**4,000** ● 2 ● 23 ● 24

---

7

Just to throw this in, an often forgotten view is of security, picture this scenario if strings were mutable:

```
string dir = "C:\SomePlainFolder";

//Kick off another thread
GetDirectoryContents(dir);

void GetDirectoryContents(string directory)
{
  if(HasAccess(directory) {
    //Here the other thread changed the string to "C:\
    return Contents(directory);
  }
  return null;
}
```

You see how it could be very, very bad if you were allowed to mutate strings once they were passed.

Share Improve this answer

Follow

answered Mar 2, 2010 at 17:40

Nick Craver

**630k** ● 138 ● 1.3k ● 1.2k

1 Yes i see the problem but I don't see any security issue about this.. and in fact if you already well known that string is mutable it easily solved by using clone either than just pass it like that.. am i missing something? – ktutnik Aug 8, 2010 at 12:23

2 @ktutnik -In a multi-threaded scenario, you can change the contents of that string, *after* it's passed the access check, effectively bypassing it and accessing whatever you want. This one one of *many* examples of security. This answer doesn't address "what would you do *if* they were mutable?"...that's a different question, the question was "why aren't they mutable now?". – Nick Craver Aug 8, 2010 at 12:32

6

You never have to defensively copy immutable data. Despite the fact that you need to copy it to mutate it, often the ability to freely alias and never have to worry about unintended consequences of this aliasing can lead to better performance because of the lack of defensive copying.

Share Improve this answer

Follow

answered Mar 2, 2010 at 17:36

dsimcha

68.6k ● 55 ● 219 ● 340

---

5

Strings are passed as reference types in .NET.

Reference types place a pointer on the stack, to the actual instance that resides on the managed heap. This is different to Value types, who hold their entire instance on the stack.

When a value type is passed as a parameter, the runtime creates a copy of the value on the stack and passes that value into a method. This is why integers must be passed with a 'ref' keyword to return an updated value.

When a reference type is passed, the runtime creates a copy of the pointer on the stack. That copied pointer still points to the original instance of the reference type.

The string type has an overloaded = operator which creates a copy of itself, instead of a copy of the pointer - making it behave more like a value type. However, if only

the pointer was copied, a second string operation could accidently overwrite the value of a private member of another class causing some pretty nasty results.

As other posts have mentioned, the StringBuilder class allows for the creation of strings without the GC overhead.

Share Improve this answer Follow

edited Mar 2, 2010 at 19:36

answered Mar 2, 2010 at 17:55

Kevin McKelvin

**3,547** ● 1 ● 29 ● 27

3 Actually, string does not have an overloaded = operator, if you do string a = b then ReferenceEquals(a, b) and indeed, ReferenceEquals(a, a.Clone()). The point is rather that because of it's immutability, we can act as if = copies, even though it doesn't. We don't have to worry about a change to b affecting a, because no changes to b are possible.
– Jon Hanna Aug 7, 2010 at 13:45

---

**3**

Strings and other concrete objects are typically expressed as immutable objects to improve readability and runtime efficiency. Security is another, a process can't change your string and inject code into the string

Share Improve this answer Follow

answered Mar 2, 2010 at 17:35

SQLMenace

**135k** ● 25 ● 211 ● 225

3

Imagine you pass a mutable string to a function but don't expect it to be changed. Then what if the function changes that string? In C++, for instance, you could simply do call-by-value (difference between `std::string` and `std::string&` parameter), but in C# it's all about references so if you passed mutable strings around every function could change it and trigger unexpected side effects.

This is just one of various reasons. Performance is another one (interned strings, for example).

Share Improve this answer

Follow

answered Mar 2, 2010 at 17:38

AndiDog

**70k** ● 21 ● 163 ● 207

2

There are five common ways by which a class data store data that cannot be modified outside the storing class' control:

1. As value-type primitives
2. By holding a freely-shareable reference to class object whose properties of interest are all immutable
3. By holding a reference to a mutable class object that will never be exposed to anything that might mutate any properties of interest

4. As a struct, whether "mutable" or "immutable", all of whose fields are of types #1-#4 (not #5).
5. By holding the only extant copy of a reference to an object whose properties can only be mutated via that reference.

Because strings are of variable length, they cannot be value-type primitives, nor can their character data be stored in a struct. Among the remaining choices, the only one which wouldn't require that strings' character data be stored in some kind of immutable object would be #5. While it would be possible to design a framework around option #5, that choice would require that any code which wanted a copy of a string that couldn't be changed outside its control would have to make a private copy for itself. While it hardly be impossible to do that, the amount of extra code required to do that, and the amount of extra run-time processing necessary to make defensive copies of everything, would far outweigh the slight benefits that could come from having `string` be mutable, *especially* given that there is a mutable string type (`System.Text.StringBuilder`) which accomplishes 99% of what could be accomplished with a mutable `string`.



answered Aug 13, 2012 at 19:08

supercat

80.8k ● 9 ● 174 ● 220

---

Immutable Strings also prevent concurrency-related issues.

0

Share Improve this answer

Follow

answered Mar 2, 2010 at 17:36

Ken Liu

22.9k ● 19 ● 80 ● 98

Why don't mutable arrays result in those issues?
– AlwaysLearning Jan 16, 2019 at 9:36

0

Imagine being an OS working with a string that some other thread was modifying behind your back. How could you validate anything without making a copy?

Share Improve this answer

Follow

answered Aug 6, 2010 at 23:57

Eton B.

6,281 ● 5 ● 32 ● 43

5 What does the OS have to do with strings in .NET? – siride Aug 7, 2010 at 0:03