

How do I generate a hashcode from a byte array in C#?

Asked 16 years, 4 months ago Modified 10 months ago Viewed 56k times



57



Say I have an object that stores a byte array and I want to be able to efficiently generate a hashcode for it. I've used the cryptographic hash functions for this in the past because they are easy to implement, but they are doing a lot more work than they should to be cryptographically oneway, and I don't care about that (I'm just using the hashcode as a key into a hashtable).



Here's what I have today:



```
struct SomeData : IEquatable<SomeData>
{
    private readonly byte[] data;
    public SomeData(byte[] data)
    {
        if (null == data || data.Length <= 0)
        {
            throw new ArgumentException("data");
        }
        this.data = new byte[data.Length];
        Array.Copy(data, this.data, data.Length);
    }

    public override bool Equals(object obj)
    {
        return obj is SomeData && Equals((SomeData)obj);
    }

    public bool Equals(SomeData other)
    {
        if (other.data.Length != data.Length)
        {
            return false;
        }
        for (int i = 0; i < data.Length; ++i)
        {
            if (data[i] != other.data[i])
            {
                return false;
            }
        }
        return true;
    }

    public override int GetHashCode()
    {
        return BitConverter.ToInt32(new
        MD5CryptoServiceProvider().ComputeHash(data), 0);
    }
}
```

```
}  
}
```

Any thoughts?

dp: You are right that I missed a check in Equals, I have updated it. Using the existing hashCode from the byte array will result in reference equality (or at least that same concept translated to hashcodes). for example:

```
byte[] b1 = new byte[] { 1 };  
byte[] b2 = new byte[] { 1 };  
int h1 = b1.GetHashCode();  
int h2 = b2.GetHashCode();
```

With that code, despite the two byte arrays having the same values within them, they are referring to different parts of memory and will result in (probably) different hash codes. I need the hash codes for two byte arrays with the same contents to be equal.

c# hash

Share

Improve this question

Follow

edited Jun 26, 2015 at 1:24



Peter Mortensen

31.6k ● 22 ● 109 ● 133

asked Aug 19, 2008 at 14:55



Andrew

2,880 ● 2 ● 27 ● 14

12 Answers

Sorted by: Highest score (default)



The hash code of an object does not need to be unique.

72

The checking rule is:

- Are the hash codes equal? Then call the full (slow) `Equals` method.
- Are the hash codes not equal? Then the two items are definitely not equal.



All you want is a `GetHashCode` algorithm that splits up your collection into roughly even groups - it shouldn't form the key as the `HashTable` or `Dictionary<>` will need to use the hash to optimise retrieval.

How long do you expect the data to be? How random? If lengths vary greatly (say for files) then just return the length. If lengths are likely to be similar look at a subset of the bytes that varies.

`GetHashCode` should be a lot quicker than `Equals`, but doesn't need to be unique.

Two identical things *must never* have different hash codes. Two different objects *should not* have the same hash code, but some collisions are to be expected (after all, there are more permutations than possible 32 bit integers).

Share

Improve this answer

Follow

edited Mar 1, 2017 at 10:51



stakx - no longer contributing

84.6k ● 20 ● 173 ● 276

answered Aug 19, 2008 at 15:17



Keith

155k ● 82 ● 306 ● 446

12 +1 That was one of the clearest explanations I have ever heard for why it is beneficial to override Equals and GetHashCode. – Andrew Hare May 4, 2009 at 15:57



54



Don't use cryptographic hashes for a hashtable, that's ridiculous/overkill.

Here ya go... Modified FNV Hash in C#

<http://bretm.home.comcast.net/hash/6.html>

```
public static int ComputeHash(params byte[] data)
{
    unchecked
    {
        const int p = 16777619;
        int hash = (int)2166136261;

        for (int i = 0; i < data.Length; i++)
            hash = (hash ^ data[i]) * p;

        return hash;
    }
}
```

Share

Improve this answer

Follow

edited May 30, 2023 at 3:08



Drew Noakes

310k ● 168 ● 696 ● 761


answered Jan 22, 2009 at 4:55




Munkie

9 This will produce pretty unique hashes, but really won't work well for GetHashCode. The idea is that the hash allows the collection to have a quick method of checking whether two byte[] match before using the slower Equals. In this implementation you're looping the entire array, so for very large arrays the equality check could be a lot faster. This is a good way to compute a general purpose hash, but for how .Net actually uses GetHashCode this could actually slow down the collections. – Keith May 17, 2012 at 13:06

3 @tigrou - I'm not saying that this isn't a useful hash mechanism, but you shouldn't use it for a GetHashCode implementation because .Net hashed collections all assume that GetHashCode will be several orders of magnitude faster than Equals. In fact if the GetHashCode check passed they will go on to call Equals because some amount of

collisions are expected. If both methods loop the entire collection you get a very slow `HashTable` or `Dictionary`. – [Keith](#) Aug 21, 2012 at 21:29 

- 15 @Keith - you are wrong here. The key point is that `GetHashCode()` has to be called only once, while `Equals()` has to be called for every comparison. So it is perfectly fine for the hash calculation to have longer running time than equals. In fact, built-in .NET string hashing does just that. – [kaalus](#) Sep 7, 2012 at 22:02
-
- 5 @Keith: kaalus is correct. A good hash code must include information from the entire object to be hashed, including all property and field values. There is no way to avoid scanning this information per call, unless the object in question is immutable and caches the hash code on creation. – [Frank Hileman](#) Mar 15, 2013 at 19:36 
-
- 1 It's worth noting that the linked page (cached version here - archive.is/MnmRY) actually uses a `uint` so will produce different hashes. – [sclarke81](#) Sep 1, 2015 at 8:02
-



Borrowing from the code generated by JetBrains software, I have settled on this function:

13



```
public override int GetHashCode()
{
    unchecked
    {
        var result = 0;
        foreach (byte b in _key)
            result = (result*31) ^ b;
        return result;
    }
}
```

The problem with just XORing the bytes is that 3/4 (3 bytes) of the returned value has only 2 possible values (all on or all off). This spreads the bits around a little more.

Setting a breakpoint in Equals was a good suggestion. Adding about 200,000 entries of my data to a Dictionary, sees about 10 Equals calls (or 1/20,000).

Share Improve this answer Follow

answered Jan 8, 2009 at 17:37



Brendan

for `ICollection<byte>` definitely use a for loop based on indexing than `foreach`. May be its not much a difference for `byte[]` since `foreach` would be converted to `for` internally.

– nawfal Dec 15, 2013 at 5:08

`foreach` loops sometimes get compiled into `for` loops when looping over `List`, not sure if this also can happen when looping over `ICollection` (which always should be a little slower, doesn't make such a difference for big arrays but for small ones => `foreach` has more initialization than `for`).

– Daniel Bişar May 11, 2020 at 10:56



Have you compared with the [SHA1CryptoServiceProvider.ComputeHash](#) method? It takes a byte array and returns a SHA1 hash, and I believe it's pretty well optimized. I used it in an [Identicon Handler](#) that performed pretty well under load.

4



Share Improve this answer Follow

answered Aug 19, 2008 at 15:53



Jon Galloway

53.1k ● 25 ● 127 ● 194



4 SHA1 is slower than MD5. If you are not worried about security then use MD5.

– Jonathan C Dickinson Jan 22, 2009 at 5:12



I found interesting results:

4

I have the class:



```
public class MyHash : IEquatable<MyHash>
{
    public byte[] Val { get; private set; }

    public MyHash(byte[] val)
    {
        Val = val;
    }

    /// <summary>
    /// Test if this Class is equal to another class
    /// </summary>
    /// <param name="other"></param>
    /// <returns></returns>
    public bool Equals(MyHash other)
    {
        if (other.Val.Length == this.Val.Length)
        {
            for (var i = 0; i < this.Val.Length; i++)
            {
                if (other.Val[i] != this.Val[i])
                {
                    return false;
                }
            }

            return true;
        }
        else
        {
            return false;
        }
    }

    public override int GetHashCode()
    {
        var str = Convert.ToBase64String(Val);
        return str.GetHashCode();
    }
}
```

Then I created a dictionary with keys of type MyHash in order to test how fast I can insert and I can also know how many collisions there are. I did the following

```
// dictionary we use to check for collisions
Dictionary<MyHash, bool> checkForDuplicatesDic = new Dictionary<MyHash,
bool>();
```

```

// used to generate random arrays
Random rand = new Random();

var now = DateTime.Now;

for (var j = 0; j < 100; j++)
{
    for (var i = 0; i < 5000; i++)
    {
        // create new array and populate it with random bytes
        byte[] randBytes = new byte[byte.MaxValue];
        rand.NextBytes(randBytes);

        MyHash h = new MyHash(randBytes);

        if (checkForDuplicatesDic.ContainsKey(h))
        {
            Console.WriteLine("Duplicate");
        }
        else
        {
            checkForDuplicatesDic[h] = true;
        }
    }
    Console.WriteLine(j);
    checkForDuplicatesDic.Clear(); // clear dictionary every 5000
iterations
}

var elapsed = DateTime.Now - now;

Console.Read();

```

Every time I insert a new item to the dictionary the dictionary will calculate the hash of that object. So you can tell what method is most efficient by placing several answers found in here in the method `public override int GetHashCode()` The method that was by far the fastest and had the least number of collisions was:

```

public override int GetHashCode()
{
    var str = Convert.ToBase64String(Val);
    return str.GetHashCode();
}

```

that took 2 seconds to execute. The method

```

public override int GetHashCode()
{
    // 7.1 seconds
    unchecked
    {
        const int p = 16777619;
        int hash = (int)2166136261;

```

```

        for (int i = 0; i < Val.Length; i++)
            hash = (hash ^ Val[i]) * p;

        hash += hash << 13;
        hash ^= hash >> 7;
        hash += hash << 3;
        hash ^= hash >> 17;
        hash += hash << 5;
        return hash;
    }
}

```

had no collisions also but it took 7 seconds to execute!

Share Improve this answer Follow

answered Mar 12, 2014 at 20:40



Tono Nam

36k ● 81 ● 324 ● 489

Could you explain your hash algorithm – [nicolas2008](#) Jan 23, 2018 at 1:17



2

If you are looking for performance, I tested a few hash keys, and I recommend [Bob Jenkin's hash function](#). It is both crazy fast to compute and will give as few collisions as the cryptographic hash you used until now.



I don't know C# at all, and I don't know if it can link with C, but here is [its implementation in C](#).



Share

Improve this answer

Follow

edited Jun 27, 2015 at 10:03



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Aug 19, 2008 at 15:16



fulmicoton

15.9k ● 10 ● 55 ● 74

You can call c functions from c# with via pinvoke. It has some performance impact (like pinning and marshalling of the passed parameters - how depends on the actual used type) but is neglectable when not calling them too frequently (which means like > thousands of times in a loop). Even some frameworks for graphic rendern (namely OpenTK, SkiaSharp) use a lot of pinvoke calls and performance is still decent. – [Daniel Bişar](#) May 11, 2020 at 10:55



1

Is using the existing hashcode from the byte array field not good enough? Also note that in the Equals method you should check that the arrays are the same size before doing the compare.



Share Improve this answer Follow

answered Aug 19, 2008 at 15:19



denis phillips
12.7k ● 5 ● 34 ● 47



1



Generating a good hash is easier said than done. Remember, you're basically representing n bytes of data with m bits of information. The larger your data set and the smaller m is, the more likely you'll get a collision ... two pieces of data resolving to the same hash.

The simplest hash I ever learned was simply XORing all the bytes together. It's easy, faster than most complicated hash algorithms and a halfway decent general-purpose hash algorithm for small data sets. It's the Bubble Sort of hash algorithms really. Since the simple implementation would leave you with 8 bits, that's only 256 hashes ... not so hot. You could XOR chunks instead of individual bytes, but then the algorithm gets much more complicated.

So certainly, the cryptographic algorithms are maybe doing some stuff you don't need ... but they're also a huge step up in general-purpose hash quality. The MD5 hash you're using has 128 bits, with billions and billions of possible hashes. The only way you're likely to get something better is to take some representative samples of the data you expect to be going through your application and try various algorithms on it to see how many collisions you get.

So until I see some reason to not use a canned hash algorithm (performance, perhaps?), I'm going to have to recommend you stick with what you've got.

Share Improve this answer Follow

answered Aug 19, 2008 at 15:31



Lee
18.7k ● 6 ● 60 ● 61



1



Whether you want a perfect hashfunction (different value for each object that evaluates to equal) or just a pretty good one is always a performance tradeoff, it takes normally time to compute a good hashfunction and if your dataset is smallish you're better off with a fast function. The most important (as your second post points out) is correctness, and to achieve that all you need is to return the Length of the array. Depending on your dataset that might even be ok. If it isn't (say all your arrays are equally long) you can go with something cheap like looking at the first and last value and XORing their values and then add more complexity as you see fit for your data.

A quick way to see how your hashfunction performs on your data is to add all the data to a hashtable and count the number of times the Equals function gets called, if it is too often you have more work to do on the function. If you do this just keep in mind that the hashtable's size needs to be set bigger than your dataset when you start,

otherwise you are going to rehash the data which will trigger reinserts and more Equals evaluations (though possibly more realistic?)

For some objects (not this one) a quick GetHashCode can be generated by ToString().GetHashCode(), certainly not optimal, but useful as people tend to return something close to the identity of the object from ToString() and that is exactly what GetHashCode is looking for

Trivia: The worst performance I have ever seen was when someone by mistake returned a constant from GetHashCode, easy to spot with a debugger though, especially if you do lots of lookups in your hashtable

Share

Improve this answer

Follow

edited Aug 1, 2013 at 16:56



Daniel Daranas

22.6k ● 9 ● 65 ● 121

answered Sep 9, 2008 at 22:35



Oskar

2,294 ● 5 ● 28 ● 36



0



```
private int? hashCode;

public override int GetHashCode()
{
    if (!hashCode.HasValue)
    {
        var hash = 0;
        for (var i = 0; i < bytes.Length; i++)
        {
            hash = (hash << 4) + bytes[i];
        }
        hashCode = hash;
    }
    return hashCode.Value;
}
```

Share Improve this answer Follow

answered Aug 28, 2014 at 21:22



Varty

363 ● 1 ● 4



0



I created some unit tests to verify my implementation (using .NET 8). This is for a class that has `string Name` and `byte[] Content` properties. I originally tried this (didn't work):

```
public override int GetHashCode()
{
    return GetHashCode.Combine(Name, Content);
}
```

then came up with this one that works perfectly and passes all unit tests:

```
public override int GetHashCode()
{
    // getting hash code of the byte array needs to be done manually
    var hashCode = new HashCode();
    hashCode.AddBytes(Content.AsSpan());

    // now combine the hash code of the name with the hash code of the content
    return HashCode.Combine(Name, hashCode.ToHashCode());
}
```

Share Improve this answer Follow

answered Jan 25 at 20:34



[Dave Black](#)

7,999 ● 2 ● 56 ● 44



-2



[RuntimeHelpers.GetHashCode](#) might help:

From Msdn:

Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures such as a hash table.

Share Improve this answer Follow

answered Aug 20, 2008 at 2:32



[jfs](#)

16.7k ● 13 ● 63 ● 90