Filter Linq EXCEPT on properties

Ask Question

36

12

This may seem silly, but all the examples I've found for using `Except` in linq use two lists or arrays of only strings or integers and filters them based on the matches, for example:

```
var excludes = users.Except(matches);
```

I want to use exclude to keep my code short and simple, but can't seem to find out how to do the following:

```
class AppMeta
{
    public int Id { get; set; }
}

var excludedAppIds = new List<int> {2, 3, 5, 6};
var unfilteredApps = new List<AppMeta>
                         {
                           new AppMeta {Id = 1},
                           new AppMeta {Id = 2},
                           new AppMeta {Id = 3},
                           new AppMeta {Id = 4},
                           new AppMeta {Id = 5}
                         }
```

How do I get a list of `AppMeta` back that filters on `excludedAppIds` ?

c# linq

edited Mar 28 '17 at 14:16

George Lanetz
275 4 17

asked Mar 21 '13 at 6:31

Wesley
2,341 6 31 54

7 Answers

60

Try a simple where query

```
var filtered = unfilteredApps.Where(i => !excludedAppIds.Contains(i.
```

The except method uses equality, your lists contain objects of different types, so none of the items they contain will be equal!

answered Mar 21 '13 at 6:33

ColinE
54.6k 10 129 186

5 For efficiency, suggest storing `excludedAppIds` as a `HashSet`, otherwise you have an `O(N²)` algorithm that traverses your exclusion list as many times as there are elements in your source. – Nigel Touch Aug 8 '17 at 11:23

@NigelTouch This is helpful! Could you expand on this a little more? What would the Big O efficiency be otherwise? So instead of the algorithm traversing the exclusion list as many times as there are elements in the list being filtered, what happens? And what happens when you're filtering an IQueryable? – nmit026 Feb 19 at 0:01

I like the Except extension methods, but the original question doesn't

Home

PUBLIC

Stack Overflow

Tags

Users

1

have symmetric key access and I prefer Contains (or the Any variation) to join, so with all credit to azuneca's answer:

```
public static IEnumerable<T> Except<T, TKey>(this IEnumerable<TKey> 
    IEnumerable<T> other, Func<T, TKey> getKey) {

    return from item in items
        where !other.Contains(getKey(item))
        select item;
}
```

Which can then be used like:

```
var filteredApps = unfilteredApps.Except(excludedAppIds, ua => ua.Id
```

Also, this version allows for needing a mapping for the exception IEnumerable by using a Select:

```
var filteredApps = unfilteredApps.Except(excludedApps.Select(a => a.
```

edited May 23 '17 at 12:10

Community ♦
1 1

answered Nov 1 '16 at 23:30

NetMage
14.2k 1 20 37

13

ColinE's answer is simple and elegant. If your lists are larger and provided that the excluded apps list is sorted, `BinarySearch<T>` may prove faster than `Contains`.

EXAMPLE:

```
unfilteredApps.Where(i => excludedAppIds.BinarySearch(i.Id) < 0);
```

edited Sep 2 '16 at 17:12

answered Mar 21 '13 at 6:39

dotNET
18.4k 13 87 150

That's very helpful, thank you. In this case, they are neither large nor is the excluded list sorted, but I will file this away and send upvotes your direction. – Wesley Mar 21 '13 at 6:42

Thanks a bunch. I'll post an example for future readers. – dotNET Mar 21 '13 at 6:47

2 `!(a >= b)` is an interesting way of saying `a<b` ;) – jazzcat Sep 2 '16 at 16:03

@jazzcat: Thanks. Updated. – dotNET Sep 2 '16 at 17:12

7

Construct a `List<AppMeta>` from the excluded List and use the Except Linq operator.

```
var ex = excludedAppIds.Select(x => new AppMeta{Id = x}).ToList();
var result = ex.Except(unfilteredApps).ToList();
```

edited Aug 17 '16 at 17:25

Gilad Green
30.7k 5 33 59

answered Mar 21 '13 at 6:36

scartag
16k 3 39 50

This solution allows for large datasets +1 – Kharaone Aug 26 '14 at 15:08

It takes too much memory and time. – George Lanetz Mar 28 '17 at 8:36

-4

```
public static class ExceptByProperty
{
    public static List<T> ExceptBYProperty<T, TProperty>(this List<T
list2, Expression<Func<T, TProperty>> propertyLambda)
    {
        Type type = typeof(T);

        MemberExpression member = propertyLambda.Body as MemberExpre

        if (member == null)
            throw new ArgumentException(string.Format(
                "Expression '{0}' refers to a method, not a property
                propertyLambda.ToString()));

        PropertyInfo propInfo = member.Member as PropertyInfo;
        if (propInfo == null)
            throw new ArgumentException(string.Format(
                "Expression '{0}' refers to a field, not a property.
                propertyLambda.ToString()));

        if (type != propInfo.ReflectedType &&
            !type.IsSubclassOf(propInfo.ReflectedType))
            throw new ArgumentException(string.Format(
                "Expresion '{0}' refers to a property that is not fr
                propertyLambda.ToString(),
                type));
        Func<T, TProperty> func = propertyLambda.Compile();
        var ids = list2.Select<T, TProperty>(x => func(x)).ToArray()
        return list.Where(i => !ids.Contains(((TProperty)propInfo.Ge
null))).ToList();
    }
}
```

```
public class testClass
{
    public int ID { get; set; }
    public string Name { get; set; }
}
```

For Test this:

```
List<testClass> a = new List<testClass>();
List<testClass> b = new List<testClass>();
a.Add(new testClass() { ID = 1 });
a.Add(new testClass() { ID = 2 });
a.Add(new testClass() { ID = 3 });
a.Add(new testClass() { ID = 4 });
a.Add(new testClass() { ID = 5 });

b.Add(new testClass() { ID = 3 });
b.Add(new testClass() { ID = 5 });
a.Select<testClass, int>(x => x.ID);

var items = a.ExceptBYProperty(b, u => u.ID);
```

answered Oct 25 '15 at 16:18

Ali Yousefie

1,165 2 14 34

1 Reflection should always be your last resort, and this is a lot of reflection for a simple task that doesn't need it at all. – NetMage Mar 9 '17 at 22:53

This is what LINQ needs

11

```
public static IEnumerable<T> Except<T, TKey>(this IEnumerable<T> iter
other, Func<T, TKey> getKey)
{
    return from item in items
           join otherItem in other on getKey(item)
           equals getKey(otherItem) into tempItems
```

```
            from temp in tempItems.DefaultIfEmpty()
            where ReferenceEquals(null, temp) || temp.Equals(default
            select item;
}
```

answered Apr 17 '15 at 12:14

azuneca
760 10 13

1 That would be a great extension method – Wesley Apr 20 '15 at 16:31

1 And it works so nice... – azuneca Jun 16 '15 at 11:40

1 That's great, except I'm not sure join is the most efficient way to exclude matches, and the same getKey is used for both sides, which wouldn't work for the original question. – NetMage Nov 1 '16 at 23:15

13

I use an extension method for Except, that allows you to compare Apples with Oranges as long as they both have something common that can be used to compare them, like an Id or Key.

```
public static class ExtensionMethods
{
    public static IEnumerable<TA> Except<TA, TB, TK>(
        this IEnumerable<TA> a,
        IEnumerable<TB> b,
        Func<TA, TK> selectKeyA,
        Func<TB, TK> selectKeyB,
        IEqualityComparer<TK> comparer = null)
    {
        return a.Where(aItem => !b.Select(bItem =>
selectKeyB(bItem)).Contains(selectKeyA(aItem), comparer));
    }
}
```

then use it something like this:

```
var filteredApps = unfilteredApps.Except(excludedAppIds, a => a.Id, I
```

the extension is very similar to ColinE 's answer, it's just packaged up into a neat extension that can be reused without to much mental overhead.

answered Mar 4 '14 at 0:29

Scott

3,072 5 32 49