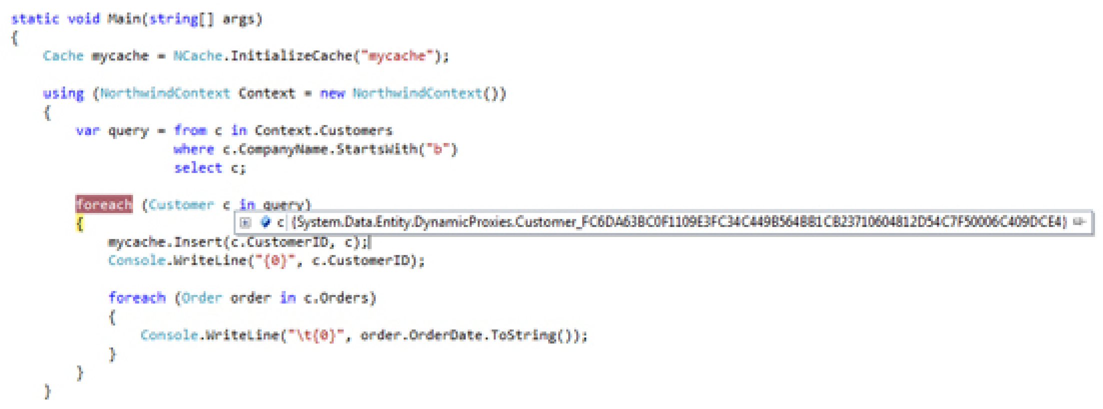
Using Entity Framework POCO Lazy Loading with Distributed Cache

Entity Framework is a very popular object-relational mapping engine provided by Microsoft and is increasingly being used in high traffic applications. And, many of these high traffic applications need scalability that comes by using an in-memory distributed cache. However, in some situations Entity Framework and the in-memory distributed cache become incompatible. Let me explain how.

If you are using Entity Framework with Plain Old CLR Objects (POCO) along with its lazy loading capability, Entity Framework dynamically generates proxy objects that contain the code for doing lazy loading. And, these dynamic object definitions only exist within the application process and therefore cannot be serialized for an out-of-process distributed cache.

Here is an example of proxy being generated for a `Customer` entity where you want to use Entity Framework lazy loading for related `orders`. See how a dynamically generated class name appears. This class name is only available within the application process and would be unknown outside the process.



```
static void Main(string[] args)
{
    Cache mycache = NCache.InitializeCache("mycache");

    using (NorthwindContext Context = new NorthwindContext())
    {
        var query = from c in Context.Customers
                    where c.CompanyName.StartsWith("b")
                    select c;

        foreach (Customer c in query)
        {        c {System.Data.Entity.DynamicProxies.Customer_FC6DA63BC0F1109E3FC34C449B564BB1CB23710604812D54C7F50006C409DCE4}
            mycache.Insert(c.CustomerID, c);
            Console.WriteLine("{0}", c.CustomerID);

            foreach (Order order in c.Orders)
            {
                Console.WriteLine("\t{0}", order.OrderDate.ToString());
            }
        }
    }
```

Example of proxy being generated by Entity Framework lazy loading

Now, the problem is that a distributed cache is always out of process and requires you to serialize all objects before caching them and then these objects may need to be de-serialized on a different machine when they are accessed from another client box. Therefore, the object type must be known on both machines for de-serialization purposes. And this is not possible with dynamically generated proxy in Entity Framework because they are only known inside the application process on one machine. So, the only way for you is to disable proxy generation in Entity Framework in order to use it with a distributed cache but this is not a very practical scenario as this also compromises Entity Framework lazy loading capabilities.

How to Use Entity Framework POCO Lazy Loading without Proxy?

In order to overcome this problem and use Entity Framework with lazy loading, you first of all need to disable proxy so that it does not cause any serialization issues. You then need to inject some additional code in your application to achieve lazy loading yourself because if proxy is turned off then there is no support for lazy loading in Entity Framework.

Also, you should write this additional code such that it helps you achieve Entity Framework lazy loading functionality and at the same time it does not violate the POCO-ness of your entities. Let us discuss this approach step by step in more details with an example.

If you have a `Customer` object that has an `Orders` list in it as a property then lazy loading should work in such a way that if you access the related `Orders` for a Customer and they have not been loaded yet (meaning they do not exist in the Entity Framework context), then the `Customer` object automatically makes a call to load related `Orders`. I am using this example below to demonstrate how to implement Entity Framework lazy loading without proxy and also use them in a distributed cache.

1. First step is to turn off Entity Framework proxy generation and also turn off Entity Framework lazy loading functionality. You implement lazy loading code yourself in the next step. The `context.ContextOptions.ProxyCreationEnabled` property is true by default. You need to explicitly set it to "false" in the default constructor of your context object in order to turn this feature off.

```
{
  public NorthwindContext() : base("name=NorthwindContext")
  {
    this.ContextOptions.ProxyCreationEnabled = false;
    this.ContextOptions.LazyLoadingEnabled = false;
  }
}
```

   Disabling proxy generation and Entity Framework lazy loading settings makes sure that dynamic proxy is not created anymore and there is no serialization issue with a distributed cache. Actual Plain Old CLR Objects (POCO) `Customer` object is used by Entity framework and hence it can be cached and retrieved in a distributed cache without having any issues.

2. Next step is to introduce some code that helps you achieve Entity Framework lazy loading for related orders for a customer. As mentioned above, this code helps you achieve the objective but it should also make sure that actual customer and Order objects still stay Plain Old CLR Objects (POCO).

For this, you create another class that has Entity framework static `ObjectContext` and a function that is used for Entity Framework lazy loading functionality for orders. This function is called by reference in the actual customer POCO object later on. Here is the code definition for your Helper class.

```
namespace Helper
{
  public class CustomerHelper
  {
    public static NorthwindContext CurrentContext;
    static public void DoLazyLoading(Customer customer, List orders)
    {
      if (CurrentContext == null) return; //no lazy loading
      var query = from o in CurrentContext.Orders where o.CustomerID ==
        customer.CustomerID select o;
      foreach (Order o in query)
      {
        orders.Add(o);
      }
    }
  }
}
```

3. You need to now update the `Customer` object so that it has the Entity Framework's ability of lazy loading all related orders for a customer. Here is how you update the Customer object and Orders list getters for calling this functionality by reference. You still need to set the `OrderLazyLoad` method to point to lazy loading function define in Helper class which is done in next step in your main program.

```
namespace MyPOCOs
{
  [Serializable]
  public class Customer
  {
    public string CustomerID { get; set; }
    public string CompanyName { get; set; }
    public static Action < Customer, List < Order >> OrderLazyLoad = null;

    private List < Order > orders;
    public virtual List Orders
    {
      get
      {
        if (orders == null)
        {
          orders = new List < Order >();
          OrderLazyLoad(this, orders);
        }
        return orders;
      }
      set
      {
        orders = value;
      }
    }
  }
  [Serializable]
  public class Order…

}
```

4. You need to now change your actual program to use this extended functionality.
   1. You set object context in the helper class `CustomerHelper.CurrentContext)` to be the same as used in the main program so that all entities are loaded under the same object context i.e. you are using same object context in the consumer program as well as in the function defined in the helper class for lazy loading.

      ```
      NorthwindContext Context = new NorthwindContext();
      CustomerHelper.CurrentContext = Context;
      ```

   2. You also set the `OrderLazyLoad` method to point to Helper lazy loading function. This is a more appropriate place to do this as you do not want your `Customer` object to refer to the Helper class object directly.

      ```
      Customer.OrderLazyLoad = Helper.CustomerHelper.DoLazyLoading;
      ```

5. Run the program where you can query customers and then also cache them in a distributed cache and also see related orders with the additional support of Entity Framework lazy loading. Each time your program accesses the `Orders` from a `Customer`, the induced lazy loading functionality is called as well.

```
static void Main(string[] args)
{
  Cache mycache = NCache.InitializeCache("mycache");
  NorthwindContext Context = new NorthwindContext();
  CustomerHelper.CurrentContext = Context;
  Customer.OrderLazyLoad = Helper.CustomerHelper.DoLazyLoading;

  var query = from c in Context.Customers where c.CompanyName.StartsWith("b") select c;
  foreach (Customer c in query)
  {
    mycache.Insert(c.CustomerID, c);
    Console.WriteLine("{0}", c.CustomerID);
    foreach (Order order in c.Orders)
    {
      Console.WriteLine("\t{0}", order.OrderDate.ToString());
    }
  }
  Console.ReadLine();
}
```

Summary

I have discussed an approach to use a distributed cache for caching Plain Old CLR Objects (POCO) used in Entity Framework without having to worry about serialization issue and still be able to have the full functionality of Entity Framework lazy loading. Although, this approach involves writing some code to achieve lazy loading but in essence you have the benefits of distributed caching such as high performance, scalability and reliability in an Entity framework application.