1. **概述**

这里的Cache主要是提供一个可扩展的缓存框架，为系统提供统一的缓存调用入口，可以通过配置实现多种具体的缓存提供者来实现真正的缓存能力，所以真正的缓存能力都是由第三方的缓存提供者来实现。

现在框架里实现了对MemoryCache（作为本地缓存，是.Net 4.0框架提供的）和Memcached（作为分布式缓存，是一个开源作品）的整合；通过实现框架中的相应接口，还可以扩展更多的缓存实现方案。

1. **使用说明**
2. 配置的说明

参见VSTS中$/ECCentral/02\_Design/01\_设计与开发指南目录下的文档《6.EC-Central项目配置文件说明.docx》中相关部分的说明

1. 程序中对缓存的使用

在程序中对缓存的使用，主要有两种方式，一种是直接在代码中调用相应的API来使用缓存，另一种是通过在相关方法上添加Attribute来控制对方法返回结果的缓存；

1. 在代码中调用API方式

在代码中调用API来使用缓存，能比较灵活的控制缓存的存取、刷新时机，完全控制所需要缓存的数据内容，以及完全控制缓存项的key的设计；当然负面影响就是缓存的使用会侵入代码正常逻辑，需要在写代码的同时考虑缓存的问题，比如需要考虑啥时存取、啥时刷新缓存、缓存什么内容、如何构造缓存的key等问题；

所调用的API主要涉及到1个静态工厂类CacheFactory和1个接口ICache：

|  |
| --- |
| public static class CacheFactory  {  public static ICache GetInstance()  public static ICache GetInstance(string name)  } |

|  |
| --- |
| public interface ICache  {  void InitFromConfig(string cacheName, NameValueCollection parameters);  bool Set(string key, object value, string groupName = null);  bool Set(string key, object value, TimeSpan slidingExpiration, string groupName = null);  bool Set(string key, object value, DateTime absoluteExpiration, string groupName = null);  object Get(string key);  T Get<T>(string key);  List<object> Get(string[] keys);  List<T> Get<T>(string[] keys);  List<object> GetByGroup(string groupName);  List<T> GetByGroup<T>(string groupName);  bool Remove(string key);  bool RemoveByGroup(string groupName);  void FlushAll();  List<string> GetKeysByGroup(string groupName);  } |

在程序中，首先通过CacheFactory的静态方法GetInstance来获取到一个ICache的具体实现类，可以给GetInstance方法传入配置中对应的CacheName来决定使用哪个具体的Cache提供者，如果传入GetInstance方法的CacheName为null或者string.Empty或者是调用的无参的重载方法，那么将默认使用配置中所配置的默认CacheName；如果CacheName在配置文件中没有找到对应的配置，则GetInstance方法将会抛出对应的异常信息；调用的代码片段如下：

|  |
| --- |
| ICache local = CacheFactory.GetInstance("local"); // 将使用配置中name为local的缓存提供者  local.Set("Test", obj); // 存入数据，缓存项的key为Test，缓存的数据为对象obj，需要obj的类型支持二进制序列化  ……  object data = local.Get("Test"); // 获取key为Test的缓存项的数据内容 |

在通过CacheFactory获取到一个ICache的具体实现类的对象实例后，即可调用相关的接口方法来做读写缓存的操作；

ICache接口主要有写入、获取、清除三类方法：

1. 将数据写入缓存中

|  |
| --- |
| bool Set(string key, object value, string groupName = null);  bool Set(string key, object value, TimeSpan slidingExpiration, string groupName = null);  bool Set(string key, object value, DateTime absoluteExpiration, string groupName = null); |

参数string key是缓存项的标识，在获取数据时通过该key来获取到缓存数据；

参数object value是需要缓存的对象数据，有一个要求就是value的具体类型需要支持二进制序列化；

参数string groupName 是一个可选参数，默认为null，如果不为null，则会为该缓存项指定一个Group；根据该GroupName，可以进行批量的操作；

参数TimeSpan slidingExpiration表示相对过期时间，在每次访问过该缓存项后，其过期时间都会延长这么多时间（即在该过期时间段内都没有访问该缓存项，那么该缓存项就过期失效）；

参数DateTime absoluteExpiration表示绝对过期时间，过了该时间后，该缓存项即刻失效过期；

1. 从缓存中获取数据

|  |
| --- |
| object Get(string key);  T Get<T>(string key);  List<object> Get(string[] keys);  List<T> Get<T>(string[] keys);  List<object> GetByGroup(string groupName);  List<T> GetByGroup<T>(string groupName); |

前两个方法是根据缓存的key来获得对应的数据；

中间两个方法是根据一批key，来获取对应的一批数据，在返回的数据的List中，每个数据的index会对应到其key在入参数组的index，如果key在缓存中没有找到数据，那么在返回的List的对应index处的元素值就为null；

最后两个方法是根据GroupName来批量获取所有的在Set方法时被设置为了该GroupName的缓存项所缓存的数据；

1. 从缓存中清除数据

|  |
| --- |
| bool Remove(string key);  bool RemoveByGroup(string groupName);  void FlushAll(); |

第一个方法是根据缓存的key来清除缓存中的对应缓存项；

第二个方法是根据GroupName来批量清除所有的在Set方法时被设置为了该GroupName的缓存项；

第三个方法是清空该缓存提供者所缓存的所有缓存项目；

1. 其它方法

|  |
| --- |
| void InitFromConfig(string cacheName, NameValueCollection parameters);  List<string> GetKeysByGroup(string groupName); |

第一个方法是只会由CacheFactory来调用的，是CacheFactory的GetInstance方法在构建好ICache实现类的对象实例后，会先读取配置文件里的配置信息，然后通过该方法将配置信息传给缓存实现类的对象实例，以供实现类的对象实例根据配置信息做相应的初始化操作，然后才会将对象实例返回给外部调用者；

第二个方法是可以根据GroupName来查找到所有的在Set方法时被设置为了该GroupName的缓存项的key的列表；

1. 在方法上添加Attribute方式

用Attribute的方式来使用缓存，其本质是以一个方法和相关参数值来构造缓存的key，将方法的返回结果缓存起来；这种方式的优点是缓存的实现对正常的代码逻辑没有任何的侵入，写代码的时候基本不用太考虑缓存的问题，可以等到最后统一来Review考虑哪些方法的返回数据需要缓存起来（当然这也不是绝对的，一个方法结果有无缓存可能会导致对方法的设计以及方法调用者的代码设计产生一定影响，所以最后的统一Review可能会导致一些代码的重构）。这种方式的负面影响就是缓存的粒度是固定的，都是缓存一个方法的返回结果，无法在方法内部灵活地对局部数据进行缓存存取（除非对需要缓存的局部数据都抽为一个单独的方法，然后对该方法添加Attribute），另外就是对于缓存的key的设计，是固定的模式，按照方法加上相关指定的入参的值的方式，无法根据实际情况对key的设计做优化和调整；

使用Attribute的方式来调用缓存，主要涉及到使用2个从System.Attribute派生出来的类：（因为采用的PostSharp的AOP织入方式，所以使用了这2个Attribute的工程都需要添加对PostSharp相关dll的引用）

|  |
| --- |
| [Serializable]  [AttributeUsage(AttributeTargets.Method, Inherited = false, AllowMultiple = false)]  public class CachingAttribute : OnMethodBoundaryAspect  {  public string ExpireTime { get; set; }  public ExpirationType ExpiryType { get; set; }  public CachingAttribute()  public CachingAttribute(string[] keys)  public CachingAttribute(string cacheName)  public CachingAttribute(string cacheName, string[] keys)  } |

|  |
| --- |
| [Serializable]  [AttributeUsage(AttributeTargets.Method, Inherited = false)]  public class FlushCacheAttribute : OnMethodBoundaryAspect  {  public FlushCacheAttribute(string typeFullName, string methodName)  public FlushCacheAttribute(Type type, string methodName)  public FlushCacheAttribute(string typeFullName, string methodName, string key)  public FlushCacheAttribute(Type type, string methodName, string key)  public FlushCacheAttribute(string typeFullName, string methodName, string[] keys)  } |

1. 被标记了CachingAttribute的方法（AllowMultiple = false，所以一个方法只能加一次CachingAttribute的特性），那么其返回值将会被缓存，对于使用CachingAttribute的方法有如下要求：

* 不能是类型的构造方法
* 方法必须要有返回值（不能返回void），并且返回值的类型需要能支持二进制序列化
* 方法不能有out参数

如果一个方法不满足上面的要求，但却添加了CachingAttribute特性标签，那么编译时将会出现相应的错误信息；使用的代码片段：

|  |
| --- |
| [Caching("local", new string[] { "customerSysNo" }, ExpiryType = ExpirationType.SlidingTime, ExpireTime = "02:00:00")]  public CustomerBasicInfo GetCustomerBasicInfoBySysNo(int customerSysNo)  {  ……  } |

CachingAttribute对缓存使用的策略，是通过其构造方法和属性来控制的：

* 属性ExpiryType

枚举，有三种值，将配合ExpireTime属性产生效果：

* + SlidingTime：表示使用相对过期时间的过期策略
  + AbsoluteTime：表示使用绝对过期时间的过期策略
  + Never：表示永不过期

如果没有设置该属性，则默认使用AbsoluteTime

* 属性ExpireTime

设置过期时间，格式为”HH:MM:SS”，默认为60分钟，如果属性ExpiryType被设置为Never，则该属性的设置将不起作用；

* 构造方法参数cacheName

对应配置文件中为每个缓存提供者所配置的name，表示具体所使用的缓存实现方式，如果cacheName为null或者string.Empty或者调用的没有传入该参数的构造方法，那么将使用配置中所配置的默认cacheName；如果cacheName在配置中没有找到，则会在运行时抛出相应异常信息；

* 构造方法参数keys

指定用来构造缓存的key的参数；因为缓存项的key是根据类型方法名加指定参数的值来构建的，所以正是通过该参数，来指定了方法的哪些入参的值需要参与到缓存key的构建中；key参数的格式就为方法的参数名或者参数名.属性名.属性名….，即支持指定复杂类型参数的各层次的下级属性；

如果该参数为null或者new string[0]或者调用的没有传入该参数的构造方法，那么该方法的缓存项的key构建上对于使用哪些参数值会采用如下策略：

* 如果方法没有参数的话，那么缓存项的Key的构建就只和类型方法名有关；
* 如果方法有参数，那么首先检查第一个参数的类型，是否为.Net简单类型或简单类型的Nullable泛型或实现了IIdentity接口的类型，如果是，那么将只使用第一个参数的值（简单类型就直接取ToString()的值，实现了IIdentity接口就取SysNo的值）参与到缓存项的key的构建中；
* 否则将会使用全部的参数的值（每个参数做二进制序列化后再转成string，所以需要参数都支持二进制序列化）参与到缓存项的key的构建中；

1. 被标记了FlushCacheAttribute的方法，在该方法执行后就会清除相应的缓存项；那么该清除哪些缓存项呢？这个就是由FlushCacheAttribute的构造方法的参数来决定的；

FlushCacheAttribute的构造方法的参数主要是三部分构成：

* 类型（Type类型或者Type的AssemblyQualifiedName字符串）
* 方法名
* 指定参与构建缓存项的key的参数

其中类型和方法名2个参数就决定了一个具体的类型的方法，该方法上必须是添加了CachingAttribute特性标签的，否则编译时会报错；通过这2个参数就指定了需要刷新的目标缓存方法；

再通过指定参与构建缓存项的key的参数，来构建出需要清除的缓存项的key；

如果keys参数为null或者new string[0]或者调用的没有传入该参数的构造方法，那么会清除所有因目标缓存方法而产生的缓存项；

使用的代码片段（会根据传入的entity的CustomerSysNo属性来清除对应的缓存项）：

|  |
| --- |
| [FlushCache(typeof(CustomerBasicDA), "GetCustomerBasicInfoBySysNo", "entity.CustomerSysNo")]  Public void UpdateBasicInfo(CustomerBasicInfo entity)  {  ……  } |