链滴

缓存键的设计

作者：loogn

原文链接：https://ld246.com/article/1561972390754

来源网站：链滴

> 很多过度设计（overengineering）借着柔性设计的名义而自认为是正当的。但是，过多的抽象层和接设计常常成为项目的绊脚石。看一下真正为用户带来强大功能的软件设计，你会发现他们通常有一非常简单的部分。简单并不容易做到。
>
> ---来自 Eric Evan《领域驱动设计》

上面的引文当然和正文无关，对领域驱动也是了解甚少。偶然读到的，感觉挺有道理，就装B引用一，下面开始正文。

如果在系统中使用过缓存，肯定会意识到有“缓存键”这么一个概念，不管是memcached还是redis是以字符串作为缓存键的。我要说的这个缓存键设计是在我们的系统中以什么样的方式得到这个字符。

可能有些人会说，直接以字符串作为缓存键不就可以了吗？直接用字符串肯定是可以的，但是维护性太好，缓存键可能遍布整个系统，就算在一个地方维护所有的键，使用者也可以随意传参，比如：

```
   class StringCacheKeys
   { public static readonly string SystemName = "SystemName"; public static readonly string
ewsDetails = "NewsDetails_{0}";
   } class AppStringCache
   { public static object GetValue(string key)
      { return null;
      } public static void Invoke()
      {
         GetValue(StringCacheKeys.SystemName);
         GetValue(string.Format(StringCacheKeys.NewsDetails, 23));
         GetValue("sbadsfsdf");
      }
   }
```

如上代码，GetValue方法是使用缓存的方法，参数按我们假设用string类型，在Invoke方法里，可以入任何字符串，虽然保证了灵活性，但失去了规范。

也许有人会用枚举来作为缓存键，单独使用枚举，肯定是很规范的，但是灵活性就不行了，很多时候存键都需要额外的具体参数填充才行，比如上面的NewsDetails_{0}，我们期望根据新闻编号来缓存闻，所以使用枚举的话，必定要借助其他的手段才能实现灵活性，比如特性（Attribute）：

```
 [AttributeUsage(AttributeTargets.Field)] class EnumCacheKeyDescriptorAttribute : Attribute
   { public string Key { get; set; } public EnumCacheKeyDescriptorAttribute(string key)
      {
         Key = key;
      }
   } enum EnumCacheKey
   {
      [EnumCacheKeyDescriptor("SystemName")]
      SystemName,

      [EnumCacheKeyDescriptor("NewsDetails_{0}")]
      NewsDetails,
   } class AppEnumCacheKey
   { public static object GetValue(EnumCacheKey key)
      { return null;
      } public static object GetValue(EnumCacheKey key, params object[] args)
```

```
        { var format = ""; //取出EnumCacheKeyDescriptor.Key;
            var realKey = string.Format(format, args); return null;
        }
    }
```

虽然可以解决问题，但是现在使用缓存的接口已经是两个了，一个没有附加参数，一个有附加参数，觉还是不好。

所以还是求助于类，求助于面向对象：

```
    public class CacheKey
    {
        TimeSpan _expires; string _key; public CacheKey(string key, TimeSpan expires)
        {
            _key = key;
            _expires = expires;
        } public TimeSpan GetExpires()
        { return _expires;
        } public virtual string GetKey()
        { if (_key.IndexOf("{0}") >= 0)
            { throw new Exception(_key + "需要额外参数，请调用BuildWithParams设置");
            } return _key;
        } public CacheKey BuildWithParams(params object[] args)
        { if (args.Length == 0)
            { throw new Exception("如果没有参数，请不要调用BuildWithParams");
            } var m = new ParamsCacheKey(_key, _expires, args); return m;
        } class ParamsCacheKey : CacheKey
        { object[] _args; public ParamsCacheKey(string key, TimeSpan expires, object[] args) : bas
(key, expires)
            {
                _args = args;
            } public override string GetKey()
            { return string.Format(_key, _args);
            }
        }
    }
```

如此这般的设计一番，是否满足了我们需求呢？第一，使用缓存的接口统一为CacheKey，第二，如需要参数，在使用的时候需要调用一下BuildWithParams方法，该方法生产一个CacheKey的不公开类ParamsCacheKey并返回，这个ParamsCacheKey负责参数的处理。代码中还有两处抛出异常的代，异常应该就是在这种情况下使用的吧！我们订制了规则而调用者不按照规则使用，当然要回复以异了。我们可以像上面一样定义一个CacheKeys来统一维护缓存键：

```
    public static class CacheKeys
    { public static CacheKey NameCacheKey = new CacheKey("Name", TimeSpan.FromHours(1))
 public static CacheKey NewsCacheKey = new CacheKey("News_{0}", TimeSpan.FromHours(1));
 }
```

CacheKey到此结束！

那么有参数的缓存键和无参数的缓存键到底有什么区别呢？不知道大家在思考这个问题的时候能想到么，我当时是用这个问题驱动我的思维的。之后还想到的两个相关的概念：

第一个是装饰器模式（允许向一个现有的对象添加新的功能，同时又不改变其结构）。我们用装饰器式可以这样实现：

```
class ThinkDecoration
{ abstract class CacheKey
    { public abstract string GetKey();
    } class StringKey : CacheKey
    { string _key; public StringKey(string key)
        {
            _key = key;
        } public override string GetKey()
        { return _key;
        }
    } class ParamsKey : CacheKey
    {
        CacheKey _cacheKey; object[] _args; public ParamsKey(CacheKey cacheKey, params ob
ect[] args)
        {
            _cacheKey = cacheKey;
            _args = args;
        } public override string GetKey()
        { var format = _cacheKey.GetKey(); return string.Format(format, _args);
        }
    } public static void RunTest()
    { var key1 = new StringKey("SystemName");
        Console.WriteLine(key1.GetKey());

        key1 = new StringKey("NewsDetails_{0}"); var key2 = new ParamsKey(key1, 23);
        Console.WriteLine(key2.GetKey());
    }
}
```

第二个是Python里的偏函数概念（其实很简单，就是设置一个函数的部分参数的默认值，生成新的数），用C#简单表示一下如下：

```
/// <summary>
/// 通过设定参数的默认值，可以降低函数调用的难度 /// </summary>
class ThinkPartialFunction
{ static int Multiply(int x, int y)
    { return x * y;
    } static int MultiplyBy2(int x)
    { return Multiply(x, 2);
    } static Func<int, int> BuildMultiplyBy(int y)
    { return (x) => Multiply(x, y);
    } //python  functools.partial(Multiply,y=2)
    static Func<int, int> BuildPartial(Func<int, int, int> fun, int y)
    { return (x) => fun(x, y);
    }
}
```

回过头来再看CacheKey，应该就是装饰器模式的一种变种应用吧。但是设计的时候我可没想到什么饰器，对设计模式也并不熟识。列出这两点，也是方便大家理解CacheKey的设计。