# Toolkit5.5的缓存

## 简介

Toolkit4时，系统引入了现在的缓存机制。引入的目的主要是为了解决系统缓存的管理，缓存的生命周期管理。当时主要采用内存缓存，同时也考虑了memcache这种分布式缓存的接入。但是在具体实现上以内存管理为主。

Toolkit的缓存主要有两个概念，一个是缓存依赖，另一个是如何创建缓存。

缓存依赖主要解决缓存的生命周期问题，不是缓存创建后，就一直存在，如果缓存依赖的条件发生改变后，缓存就应该及时摧毁，重新创建。比如有些缓存依赖文件，文件一旦改变缓存必须重建。有些缓存需要一段时间存在，过了一段时间后，可以过期重新创建。这些都是缓存依赖需要解决的。不同的情况下，选择不同的缓存依赖，解决不同的问题。

系统统一管理缓存，当API进行查找时，如果命中，则返回给缓存实例。但是如果是第一次，或者说缓存依赖失效，这就涉及到创建缓存的事情了。经过多年的实践，感觉定义一个类让用户继承可以比较好的解决这个问题，因此系统设计了BaseCacheItemCreator基类，让创建缓存可以集中统一管理。

Toolkit5.5由于迁移到.net standard版，注意到微软定义了IDistributeCache接口，并实现了Redis和Sql Server两个分布式缓存。因此将Toolkit的缓存和IDistributeCache进行了接入，支持了分布式缓存。分布式缓存的优点在于可以同时为多个web服务提供服务，因此可以共享缓存。同时，在web应用重启后，缓存依然存在分布式缓存服务器上，不需要再web服务器上重建。

下面详细介绍一下缓存的各个功能点。

## 缓存依赖

缓存依赖的接口定义如下：

public interface ICacheDependency

{

bool HasChanged { get; }

}

当感知缓存发生变化时，HasChanged属性返回true即可，否则返回false。

系统已经实现的缓存依赖如下：

* AlwaysCache：永久有效的缓存（即HasChanged一直返回false）。
* ActiveTime：一段时间活跃的缓存，一旦访问缓存会重置活跃时间。
* DayChange：当时间进入下一个（或n个）自然日时，缓存失效。
* TimeSpan：一段时间存活的缓存，注意访问缓存不会重置时间。
* NoCache：永久无效的缓存（即HasChanged一直返回true）
* FileInfo：监视文件最后修改时间，一旦发生改变，缓存失效。
* UserInfo：监视当前登录用户，一点登录用户改变，缓存失效。

每种缓存依赖，都公开了对应的Attribute，可以标注在类型上。同时还有对应的Xml配置插件，在Xml配置的时候使用。

如果上述缓存依赖都不符合要求，可以自行定义一个类，实现ICacheDependency接口即可。要有对应的Attribute，可以参考如下代码：

[AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = false)]

public sealed class AlwaysCacheAttribute : CacheDependencyAttribute

{

protected override ICacheDependency CreateCacheDependency()

{

return AlwaysDependency.Dependency;

}

public override string ToString()

{

return "永远存在的缓存特性";

}

}

只要重载CreateCacheDependency方法，返回刚刚实现的ICacheDependency的类实例即可。

要实现Xml配置，可以参考如下代码（这里不解释配置插件的机理）：

[CacheDependencyConfig(NamespaceType = NamespaceType.Toolkit, Description = "永久有效的缓存依赖",

Author = "YJC", CreateDate = "2013-09-28")]

internal class AlwaysConfig : IConfigCreator<ICacheDependency>

{

#region IConfigCreator<ICacheDependency> 成员

ICacheDependency IConfigCreator<ICacheDependency>.CreateObject(params object[] args)

{

return AlwaysDependency.Dependency;

}

#endregion IConfigCreator<ICacheDependency> 成员

}

}

定义一个类，实现IConfigCreator<ICacheDependency>接口，并在类上标注CacheDependencyConfigAttribute，即可。

## 缓存对象

缓存对象可以是任意的.net的class。但是缓存对象需要说清楚它对应的缓存依赖。

系统支持两种方式的缓存依赖方式。

第一种比较简单，直接在class的定义上标注需要的Attribute

[UserInfoCache]

public sealed class SessionGlobal

{

…

能够标记Attribute比较简单，但是很多情况下，因为Attribute会带有参数，而参数是程序动态决定的，而不是可以直接标注的。这种情况下，系统提供了一个新的接口

public interface ICacheDependencyCreator

{

ICacheDependency CreateCacheDependency();

}

缓存对象实现这个接口，也可以达到效果。

参考以下例子：

public sealed class FileData : IRegName, ICacheDependencyCreator

{

private readonly FileInfoDependency fDependency;

internal FileData(string fileName)

{

TkDebug.AssertArgumentNullOrEmpty(fileName, "fileName", null);

TkDebug.AssertArgument(File.Exists(fileName), "fileName",

string.Format(ObjectUtil.SysCulture, "{0}不存在", fileName), null);

FileName = fileName;

fDependency = new FileInfoDependency(fileName);

Data = File.ReadAllBytes(fileName);

}

#region ICacheDependencyCreator 成员

public ICacheDependency CreateCacheDependency()

{

return fDependency;

}

#endregion

#region IRegName 成员

string IRegName.RegName

{

get

{

return FileName;

}

}

#endregion

public string FileName { get; private set; }

public byte[] Data { get; private set; }

public override string ToString()

{

return string.Format(ObjectUtil.SysCulture, "{0}的文件内容", FileName);

}

}

这里FileData作为一个缓存对象，缓存一个文件的内容。由于不同的文件，说产生的文件依赖不同，因此，使用接口编程可以有效的解决这个问题。

使用上述两种方法，缓存对象就OK了。如果缓存对象没有标记缓存依赖，在构建缓存时，系统因为找不到缓存依赖，会直接报错。

## BaseCacheItemCreator

BaseCacheItemCreator是创建缓存的基类。如果需要缓存，那么必须从BaseCacheItemCreator继承。

### 创建缓存对象

BaseCacheItemCreator定义了一个函数来创建缓存对象，此函数必须由实现类重载完成

public abstract object Create(string key, params object[] args);

第一个参数key代表此对象在缓存中唯一标记，后面的参数可选，可以传入创建缓存必要的参数。

调用args的方法有两种，一种是记住调用缓存时传入的参数顺序，然后在这里按顺序还原，比如：

object data = args[0];

string modeName = (string)args[1];

这样效率是挺高，但是如果调用者没有遵循这个顺序，代码就有可能出错。上述代码属于内部调用，没有暴露出来，因此可以这么做。但是不推荐上述做法。

系统在ObjectUtil提供了一个两个工具函数QueryObject和ConfirmQueryObject，只要传入指定的类型，该函数会在args中查找类型匹配的并返回。两个函数的区别在于，QueryObject如果找不到就返回null。而ConfirmQueryObject一旦找不到就会抛出错误。

比如：

Type type = ObjectUtil.ConfirmQueryObject<Type>(this, args);

### 转换key

BaseCacheItemCreator还有以下方法可以选择重载

public virtual string TransformCacheKey(string key)

{

return key;

}

这个在使用key前，可以对key做必要的转换。例如在Toolkit4和Toolkit5中，考虑到Windows下文件名称大小写不敏感，因此重载了代码做了如下变化

public override string TransformCacheKey(string key)

{

return key.ToLower(ObjectUtil.SysCulture);

}

但是由于linux下文件名称大小写敏感，所以Toolkit5.5取消了上述代码。要能支持跨平台，必须注意文件名的大小写。

### 可选的构造函数

通常可以使用BaseCacheItemCreator默认的构造函数来创建。还有一个构造函数可以由高级用户使用

protected BaseCacheItemCreator(ICacheCreator cacheCreator)

默认一般不传这个参数，系统将使用默认配置的CacheCreator来创建存储缓存的Cache。ICacheCreator的定义如下：

public interface ICacheCreator

{

ICache CreateCache();

}

ICache是存储缓存对象的缓存管理器。通常情况下，都是基于内存的ICache来进行管理。在Toolkit5.5中实现了基于IDistributeCache来管理缓存的ICache。当前系统提供了基于内存的SimpleCache和基于分布式内存的RedisCache。如果对上述Cache觉得不满意，可以实现自己的Cache来管理缓存。不过这个属于高级话题，这里不会说的太细。

通常情况下，使用默认的就OK了。

### 插件化BaseCacheItemCreator

Toolkit5中，由于对插件进行了深度改造（插件不是本文描述范畴）。开发插件变得非常容易，因此所有从BaseCacheItemCreator继承的子类都必须注册成为插件。这里需要注意的是，这个类通常情况下，只需要创建一次，以后可以反复使用。因此，我们将使用单例模式封住该类的构造函数，同时注册一个Instance的内存实例到插件工厂。

参考代码如下：

[CacheItemCreator(Author = "YJC", CreateDate = "2019-10-04",

Description = "单表Scheme的分析数据的缓存对象创建器")]

[InstancePlugIn, AlwaysCache]

internal class SessionGlobalCacheCreator : BaseCacheItemCreator

{

public static BaseCacheItemCreator Instance = new SessionGlobalCacheCreator();

private SessionGlobalCacheCreator()

{

}

public override object Create(string key, params object[] args)

{

return SessionGlobal.CreateSessionGlobal(key);

}

}

此例中，构造函数被声明为private，那么系统将无法通过构造函数来实例化对象。这时，由于类上标注了InstancePlugIn，那么系统将会去查找该类中名称为Instance的静态对象作为插件实例注册，而类型定义中刚好有。由于插件工厂对该实例进行了缓存，所以要标记AlwaysCache这个缓存依赖，让这个对象永久存活。

最后，记得标记CacheItemCreatorAttribute，注册成为缓存插件。该声明中，没有标记插件的注册名。那么它将根据类型自行生成插件注册名。生成规则如下：如果类型定义以CacheCreator结尾，那么插件注册名会是类型名去掉CacheCreator，上例中插件的注册名为SessionGlobal。如果类型定义不是以CacheCreator结尾，那么插件注册名会是类型的全名。

## 如何调用缓存

定义好了缓存对象和对应的BaseCacheItemCreator，那么剩下的就是调用了。系统管理所有缓存的类叫CacheManager。只要系统被正确初始化（即在程序初始化的时候调用相关的Initialize函数），就可以使用CacheManager获取缓存。

CacheManager提供了GetItem静态方法来获取缓存，原型如下：

public static object GetItem(string cacheName, string key, params object[] args)

具体代码如下：

var global = CacheManager.GetItem("SessionGlobal", userId).Convert<SessionGlobal>();

第一个参数是BaseCacheItemCreator的插件注册名。第二个参数是key。如果创建缓存对象还有其他对象，可以在key后继续传入。

下面是带额外参数的获取缓存的方法：

return CacheManager.GetItem("ObjectInfo", key, data, modelName).Convert<ObjectInfo>();

## 分布式缓存和普通缓存的区别

在Microsoft.Extensions.Caching.Abstractions中，微软定义了一个缓存框架。并定义了IDistributeCache这个接口用以实现分布式缓存。微软实现了基于Redis和基于Sql Server的两个分布式缓存。

分布式缓存可以将缓存存在另外一个服务器上，从而降低了web服务器的开销。如果有多台web服务器，可以共享缓存。甚至于在web服务器重启后，所有的内存缓存都被销毁了，但是分布式缓存中的对象依旧存在，可以重新获取。

但是分布式缓存和普通缓存有很大的区别，普通缓存可以存储任意一个.net的对象实例。而分布式缓存必须将.net对象序列化成byte[]，然后进行存储。序列化和反序列化是分布式缓存和普通缓存的最大区别。

同时缓存依赖在系统中也是以对象实例存在的，如果不存入分布式服务器，那么系统一旦重启，从分布式缓存服务器上获取的缓存对象就不清楚是否有效了。为了保证缓存对象的有效性，缓存依赖一样必须序列化成byte[]，一起存储。

由于Toolkit5提供了强大的Attribute，可以轻松将对象Xml化或者Json化，因此借助这套Attribute系统，我们将对象转换成Json，并进一步转化成byte[]。同样的，从分布式服务器取回byte[]后，先变成Json，在从Json复原成对应的对象。

基于上述原理后，Toolkit5.5在原有缓存的基础上，进行了如下扩展：

### ICacheDataConverter

ICacheDataConverter定义如下：

public interface ICacheDataConverter

{

byte[] Convert(object data);

object CreateEmptyData();

object ReadData(byte[] data);

}

Convert方法将缓存对象转换为byte[]。CreateEmptyData创建需要读取数据的空对象。ReadData是将CreateEmptyData创建的对象读取byte[]后，变成缓存对象。

这个接口的定义，使得缓存对象可以在分布式缓存上存入并读取。

### BaseDistributeCacheItemCreator<T>

BaseDistributeCacheItemCreator从BaseCacheItemCreator继承，并实现了ICacheDataConverter接口。这里T指的是缓存对象的类型。默认的，Convert方法会将缓存对象转成Json在进行UTF8编码。而ReadData方法则将byte[]通过UTF8编码变成json，在由CreateEmptyData创建的对象读取json从而获得缓存对象。CreateEmptyData则是根据T类型，查找其无参的构造函数，并由该构造函数创建对象实例。如果类型T有默认构造函数，可以考虑使用DistributeCacheItemCreator<T>，该类定义如下：

public abstract class DistributeCacheItemCreator<T> : BaseDistributeCacheItemCreator<T> where T : new()

这样，CreateEmptyData就可以直接使用new来创建类，比反射效率高。

### IDistributeCacheDependency

IDistributeCacheDependency解决把ICacheDependency为byte[]的问题。该接口定义如下：

public interface IDistributeCacheDependency : ICacheDependency

{

object CreateStoredObject();

}

这里，需要创建一个存储CacheDependency内部变量的对象，然后和缓存对象一样，先Json化，然后在byte[]，这样可以和缓存对象一起存储到一起。系统现有的CacheDependency全部都实现了对应的IDistributeCacheDependency，可以放心使用。如果是自行开发的CacheDependency需要注意，如果不实现这个接口，一旦使用分布式缓存，系统因为检测不到IDistributeCacheDependency接口，就会抛错。

下面以FileInfo为例，看看它们如何实现：

[CacheDependencyStoreConfig(NamespaceType = NamespaceType.Toolkit, CreateDate = "2019-09-18",

Description = "跟踪单个文件变化(监控文件的修改时间)的缓存依赖", Author = "YJC", RegName = "FileInfo")]

internal class FileInfoStoreConfig : IConfigCreator<IDistributeCacheDependency>

{

[SimpleAttribute]

public long LastWriteTime { get; set; }

[SimpleAttribute]

public string FileName { get; set; }

public IDistributeCacheDependency CreateObject(params object[] args)

{

return new FileInfoDependency(this);

}

}

public sealed class FileInfoDependency : ICacheDependency, IDistributeCacheDependency

{

private readonly FileInfo fFileInfo;

private readonly string fFileName;

private readonly long fLastWriteTime;

public FileInfoDependency(string fileName)

{

TkDebug.AssertArgumentNullOrEmpty(fileName, "fileName", null);

fFileName = fileName;

fFileInfo = new FileInfo(fFileName);

fLastWriteTime = fFileInfo.LastWriteTime.Ticks;

}

internal FileInfoDependency(FileInfoStoreConfig config)

{

fFileName = config.FileName;

fFileInfo = new FileInfo(fFileName);

fLastWriteTime = config.LastWriteTime;

}

#region ICacheDependency 成员

public bool HasChanged

{

get

{

fFileInfo.Refresh();

DateTime time = fFileInfo.LastWriteTime;

return time.Ticks != fLastWriteTime;

}

}

#endregion ICacheDependency 成员

public object CreateStoredObject()

{

return new FileInfoStoreConfig

{

FileName = fFileName,

LastWriteTime = fLastWriteTime

};

}

public override string ToString()

{

return string.Format(ObjectUtil.SysCulture,

"根据文件{0}写入时间判断的缓存依赖", fFileName);

}

}

需要注意的是存储对象，需要标注CacheDependencyStoreConfigAttribute，将其变为配置插件，这样系统才会根据标签名自动找到相关的类进行实例化。

### ICacheDependencyTime

分布式缓存没有Toolkit的缓存依赖那么灵活的控制，但是有基于时间的类似于缓存依赖控制。因此，如果可以大致估算出时间的缓存依赖可以实现这个接口，返回大致时间，这样可以做到相对精确的控制。当然，像文件缓存依赖，没法估算时间的可以不实现该接口，系统大致估算一个时间，原则这个时间不会超过当天。分布式服务器会自行清理过期的缓存，这块与Toolkit无关。

需要注意的是，这里只是借助于分布式缓存固有的功能，真正缓存是否过期，还是以Toolkit定义的ICacheDependency为准，不要弄混。

ICacheDependencyTime定义如下：

public interface ICacheDependencyTime

{

DateTime AbsoluteExpiration { get; }

}

DayChangeDependency的实现如下：

public DateTime AbsoluteExpiration => fCacheDay.AddDays(Days);

### DistributedCache

DistributedCache实现ICache接口，同时接受微软定义的IDistributeCache接口，通过API操作，尝试从分布式服务器读取缓存，如果命中就返回；否则创建缓存，存入分布式服务器并返回。由于每次从分布式服务器读取的都是byte[]，这里反序列化也需要一定的时间，为此，在DistributedCache的基础上，系统又提供了WeakDistributedCache。采用WeakReference的方式管理从分布式服务器读取的缓存。如果WeakReference存在，就不需要去分布式服务器，直接返回对象，如果WeakReference不存在，再去分布式服务器上读取，同时还需要存入WeakReference中。由于WeakReference是弱引用，其中的内容随时可以被GC收回，也不会占用太多资源，而大多数情况下可以省掉大量的反序列化的时间，这样效率会比较高。

原则上，希望大家使用从WeakDistributedCache继承的实现类。

### ICacheCreator插件化

在介绍BaseCacheItemCreator时，曾经说过ICacheCreator。由于需要用户可以自行定义默认使用哪种ICache进行缓存存储，因此将ICacheCreator插件化。这样可以使用相应的注册名进行配置。

## Redis分布式缓存

Redis缓存主要基于微软提供的Microsoft.Extensions.Caching.Redis扩展包，以及前面说的DistributedCache和WeakDistributeCache两个基类来开发。

提供了注册名为Redis和RedisWeak的两个ICacheCreator来创建RedisDistributedCache和RedisWeakDistributedCache。因此主要在配置文件填上适当注册名即可。

Redis的配置如下：

[Config(RegName = REG\_NAME, Author = "YJC", CreateDate = "2019-09-30", Description = "Redis的配置")]

public class RedisConfig

{

public const string REG\_NAME = "RedisCache";

[SimpleElement(Required = true)]

public string Configuration { get; private set; }

[SimpleElement]

public string InstanceName { get; private set; }

}

借助于Toolkit5.5提供的配置机构（参考”Toolkit5.5的配置和缺省值.docx”），在Config.xml中做如下配置：

<RedisCache>

<Configuration>192.168.2.21</Configuration>

<InstanceName>Toolkit</InstanceName>

</RedisCache>

Configuration是Redis服务器的地址，这里填写的是IP地址，如果有域名解析，亦可以填写相应的域名。InstanceName是实例名称，这里任意名称，最好每个项目都不同。

## 如何应用分布式缓存

现有代码缓存代码如果需要改造成为分布式缓存，首先需要检视缓存对象是否标记了Toolkit提供的一套Attribute标记，将需要的数据可以统统Json化，同时从Json读取也不会丢失。如果满足这个条件，就可以改造了。改造过程比较简单，我们这里以DataXml为例。

由于DataXml是使用Toolkit的Attribute标记从Xml读取的，所以它天生就有这个能力，因此无需改造缓存对象。下一步就是DataXml从BaseCacheItemCreator继承的类做基类调整

原有代码为：

internal class Tk5DataXmlCacheCreator : BaseCacheItemCreator

改造后的代码为：

internal class Tk5DataXmlCacheCreator : BaseDistributeCacheItemCreator<InternalTk5DataXml>

也就是说，更换基类就可以了。由于InternalTk5DataXml没有默认构造函数，因此，这里选用的是这个基类。如果有默认构造函数，可以考虑使用DistributeCacheItemCreator<T>泛型。

上述过程后，DataXml已经具备存储到分布式缓存服务器的能力了。但是系统默认了分布式缓存和普通内存缓存的CacheCreator都是Simple（即创造基于内存缓存ICache的CacheCreator）。设定这个默认值的原因是Simple存在于Toolkit.Core这个工程，可以保证不会出错。如果要配置Redis或者RedisWeak，那么需要引入YJC.Toolkit.RedisCache.dll。在Default.xml中，我们在tk:Simple区块中可以重新定义DistributeCache使用的CacheCreator，定义如下：

<tk:Simple>

<DefaultDistributedCache>RedisWeak</DefaultDistributedCache>

</tk:Simple>

这样，那些从DistributeCacheItemCreator和BaseDistributeCacheItemCreator继承的缓存构建类，都会使用RedisWeak创建的ICache存储缓存。

另外，如果想修改默认生成内存缓存CacheCreator，可以在tk:Simple区块定义

<DefaultCache>XXX</DefaultCache>

这样，新定义注册名对应的CacheCreator就会替代默认的。

## 定时清理过期缓存

该功能已经完成。当缓存依赖的HasChanged变为true时，这个缓存就过期了。定期清理过期缓存的程序会通过工作线程，尝试清理这些过期的缓存。因为线程安全的缘故，缓存的读写都有读写锁进行保护，防止出现并发问题。因为清理缓存时，可能缓存有锁正在读取，所以它会尝试进入，一旦发现有锁，就不进入了，防止干扰正常的工作。

此外，由于分布式缓存都存在分布式服务器上，系统没有留底，没法进行遍历工作，因此对于使用DistributedCache管理缓存的，将无法进行清理。而如果使用WeakDistributedCache管理缓存，因为本地存了部分分布式服务器上的缓存，所以可以对本地存储的部分缓存进行遍历，一旦发现过期，将同时清除本地和分布式服务器上的缓存。

## 继续需要做的工作

当前，Toolkit5.5的缓存工作并没有全部完成。还缺少基于Sql Server的分布式缓存。这些工作不做，不影响Toolkit5.5的运行，因此优先级较低。

这部分工作将在后续慢慢完成，完成后，会更新这个文档。