

目 录

- 0. 文档介绍2
 - 0.1 文档目的2
 - 0.2 文档范围2
 - 0.3 读者对象2
 - 0.4 参考文献2
 - 0.5 术语与缩写解释3
- 1. 概述4
 - 1.1 背景4
 - 1.2 主要特性4
 - 1.3 环境5
 - 1.4 下载资源5
- 2. EHCACHE 页面缓存的配置5
 - 2.1 EHCACHE 的类层次模型5
 - 2.2 环境搭建6
 - 2.3 EHCACHE 配置文件中元素说明6
 - 2.4 在工程中单独使用10
- 3. 在 SPRING 中运用 EHCACHE11
- 4. 分布式缓存集群环境配置22
 - 4.1 集群配置方式22
- 5. 测试用例26

0. 文档介绍

0.1 文档目的

记录使用 *EHCache* 实现页面级的缓存以及完成集群设计的过程。

0.2 文档范围

记录使用 *EHCache* 实现页面级的缓存以及完成集群设计的过程。

0.3 读者对象

任何有兴趣的家伙。

0.4 参考文献

提示：列出本文档的所有参考文献（可以是非正式出版物），格式如下：

[标识符] 作者，文献名称，出版单位（或归属单位），日期

大部分都是网络上查询的资料，很多，不列举了。

EHCache 简介：

<http://apps.hi.baidu.com/share/detail/7491847>

<http://wangjicn.cn/data/read/9082403332378.html>

<http://blog.csdn.net/mgoann/archive/2009/04/16/4083179.aspx>

<http://yuanyong.javaeye.com/blog/691499>

Spring 整合 *EHCache*：

<http://wangjicn.cn/data/read/909291257438.html>

<http://www.yybean.com/ehcache-getting-started-series-5-a-distributed-cache-cluster-environment-configuration>

<http://zhyt710.javaeye.com/blog/333213>

<http://tech.ddvip.com/2010-04/1270187299149502.html>

<http://blog.csdn.net/goodboy11111/archive/2010/04/01/5442329.aspx>

0.5 术语与缩写解释

缩写、术语	解 释
EHCache	EHCache 是一个快速的、轻量级的、易于使用的、进程内的缓存。它支持 read-only 和 read/write 缓存，内存和磁盘缓存。是一个非常轻量级的缓存实现，而且从 1.2 之后就支持了集群，目前的最新版本是 2.1。
...	

1. 概述

1.1 背景

系统缓存是位于应用程序与物理数据源之间，用于临时存放复制数据的内存区域，目的是为了减少应用程序对物理数据源访问的次数，从而提高应用程序的运行性能。缓存设想内存是有限的，缓存的时效性也是有限的，所以可以设定内存数量的大小，可以执行失效算法，可以在内存满了的时候，按照最少访问等算法将缓存直接移除或切换到硬盘上。

Ehcache 从 Hibernate 发展而来，逐渐涵盖了 Cache 界的全部功能,是目前发展势头最好的一个项目。具有快速,简单,低消耗，依赖性小，扩展性强,支持对象或序列化缓存，支持缓存或元素的失效，提供 LRU、LFU 和 FIFO 缓存策略，支持内存缓存和磁盘缓存，分布式缓存机制等等特点。

Cache 存储方式：内存或磁盘。

官方网站：<http://ehcache.sourceforge.net/>

1.2 主要特性

1. 快速.
2. 简单.
3. 多种缓存策略
4. 缓存数据有两级：内存和磁盘，因此无需担心容量问题
5. 缓存数据会在虚拟机重启的过程中写入磁盘
6. 可以通过 RMI、可插入 API 等方式进行分布式缓存
7. 具有缓存和缓存管理器的侦听接口

-
8. 支持多缓存管理器实例，以及一个实例的多个缓存区域
 9. 提供 Hibernate 的缓存实现

1.3 环境

Windows XP、JDK1.6.03、Tomcat5.5、EHcache2.1

注意：配置好环境变量。

1.4 下载资源

ehcache-2.1.0-distribution.tar.gz: 以及 ehcache-web-2.0.2-distribution.tar.gz

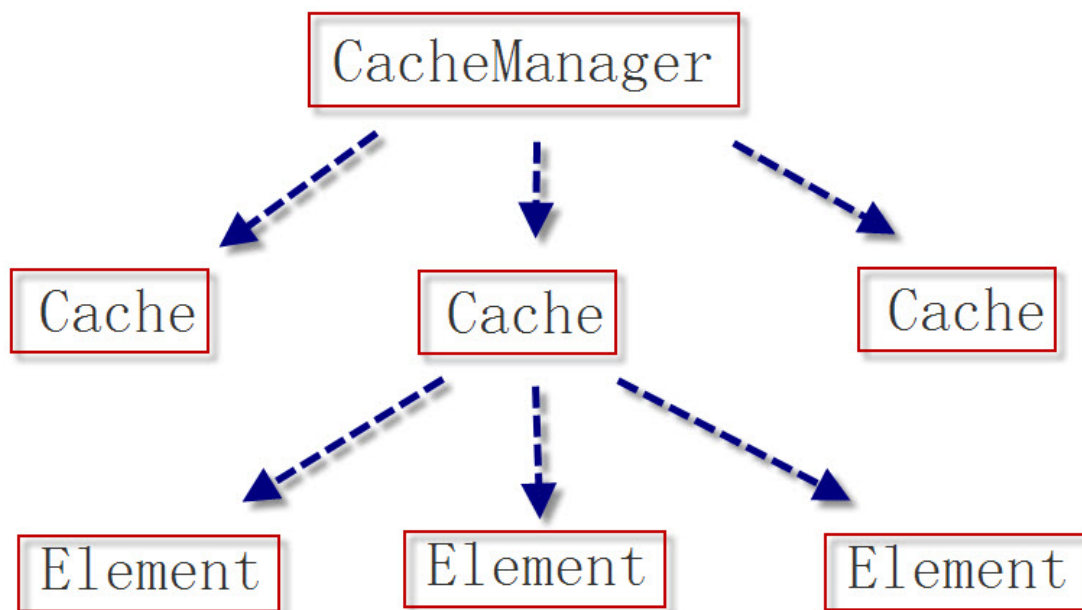
<http://sourceforge.net/projects/ehcache/>

注意：同时要下载源代码，部分功能需要修改源代码，重新做包。

2. EHCACHE 页面缓存的配置

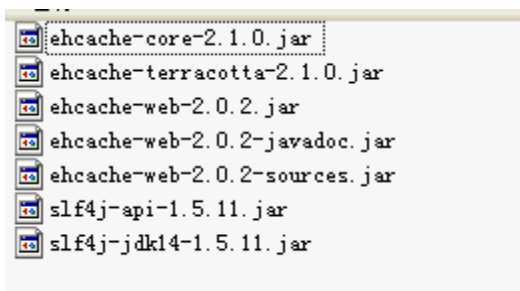
2.1 EHCACHE 的类层次模型

主要为三层，最上层的是 CacheManager，他是操作 Ehcache 的入口。我们可以通过 CacheManager.getInstance() 获得一个单子的 CacheManger，或者通过 CacheManger 的构造函数创建一个新的 CacheManger。每个 CacheManager 都管理着多个 Cache。而每个 Cache 都以一种类 Hash 的方式，关联着多个 Element。Element 则是我们用于存放要缓存内容的地方。



2.2 环境搭建

将 ehcache-2.1.0-distribution.tar.gz: 以及 ehcache-web-2.0.2-distribution.tar.gz 解压得到



需要将它们放置到 WEB-INF/lib 下。

有一个重要的配置文件 ehcache.xml，可以从 ehcache 组件包中拷贝一个，也可以自己建立一个。需要放到 classpath 下。常放的路径为 /WEB-INF/classes/ehcache.xml。

2.3 ehcache 配置文件中元素说明

ehcach.xml 配置文件主要参数的解释,其实文件里有详细的英文注释//DiskStore 配置, cache 文件的存放目录, 主要的值有

- * user.home - 用户主目录
- * user.dir - 用户当前的工作目录
- * java.io.tmpdir - Default temp file path 默认的 temp 文件目录

范例

1、首先设置 EhCache，建立配置文件 ehcache.XML，默认的位置在 class-path，可以放到你的 src 目录下：

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
<diskStore path="Java.io.tmpdir"/>
<defaultCache
maxElementsInMemory="10000" <!-- 缓存最大数目 ->
eternal="false" <!-- 缓存是否持久 ->
overflowToDisk="true" <!-- 是否保存到磁盘，当系统当机时->
timeToIdleSeconds="300" <!-- 当缓存闲置 n 秒后销毁 ->
timeToLiveSeconds="180" <!-- 当缓存存活 n 秒后销毁->
diskPersistent="false"
diskExpiryThreadIntervalSeconds="120"/>
</ehcache>
```

了解 ehcache 的几个概念，

1 timeToIdleSeconds ，多长时间不访问该缓存，那么 ehcache 就会清除该缓存。

2 timeToLiveSeconds ，缓存的存活时间，从开始创建的时间算起。

Ehcache 的三种清空策略

1 FIFO, first in first out, 这个是大家最熟的，先进先出。

2 LFU, Less Frequently Used, 就是上面例子中使用的策略，直白一点就是讲一直以来最少被使用的。如上面所讲，缓存的元素有一个 hit 属性，hit 值最小的将会被清出缓存。

3 LRU, Least Recently Used, 最近最少使用的，缓存的元素有一个时间戳，当缓存容量满了，而又需要腾出地方来缓存新的元素的时候，那么现有缓存元素中时间戳离当前时间最远的元素将被清出缓存。

首页的页面缓存

一个网站的首页估计是被访问的次数最多的，我们可以考虑给首页做一个页面缓存
缓存策略：我认为应该是某个固定时间之内不变的，比如说 2 分钟更新一次，以应用结构 page-filter-action-service-dao-db 为例。

位置：页面缓存做到尽量靠近客户的地方，就是在 page 和 filter 之间，这样的优点就是第一个用户请求之后，页面被缓存，第二个用户再来请求的时候，走到 filter 这

个请求就结束了，无需再走后面的 action- service-dao-db。带来的好处是服务器压力的减低和客户段页面响应速度的加快。

首页的页面缓存的存活时间，我们定的是 2 分钟，那么也就是说我们的 `timeToLiveSeconds` 应该设置为 120，同时我们的 `timeToIdleSeconds` 最好也设置为 2 分钟，或者小于 2 分钟。我们来看一下下面这个配置，这个配置片段应该放到 `ehcache.xml` 中：

```
<cache name="SimplePageCachingFilter"
maxElementsInMemory="10"
maxElementsOnDisk="10"
eternal="false"
overflowToDisk="true"
diskSpoolBufferSizeMB="20"
timeToIdleSeconds="60"
timeToLiveSeconds="120"
memoryStoreEvictionPolicy="LFU"
/>
```

缓存的名字

内存中SimplePageCachingFilter缓存中元素的最大数量为10

硬盘中SimplePageCachingFilter缓存中元素的最大数量为10

该缓存会死亡

overflowToDisk=true 意思是表示当缓存中元素的数量超过限制时，就把这些元素持久化到硬盘，如果overflowToDisk 是false，那么maxElementsOnDisk 的设置就没有什么意义了

多长时间不访问该缓存，那么ehcache就会清除该缓存

缓存的存活时间，从开始创建的时间算起

一直以来最少被使用的。缓存的元素有一个hit属性，hit值最小的将会被清出缓存。

`SimplePageCachingFilter` 是缓存的名字，`maxElementsInMemory` 表示内存中 `SimplePageCachingFilter` 缓存中元素的最大数量为 10，`maxElementsOnDisk` 是指持久化该缓存的元素到硬盘上的最大数量也为 10（），`eternal=false` 意味着该缓存会死亡。
`overflowToDisk=true` 意思是表示当缓存中元素的数量超过限制时，就把这些元素持久化到硬盘，如果 `overflowToDisk` 是 `false`，那么 `maxElementsOnDisk` 的设置就没有什么意义了。`memoryStoreEvictionPolicy=LFU` 是指按照缓存的 `hit` 值来清除，也就是说缓存满了之后，新的对象需要缓存时，将会将缓存中 `hit` 值最小的对象清除出缓存，给新的对象腾出地方来了。

接着我们来看一下 `SimplePageCachingFilter` 的配置，

XML/HTML 代码

`<filter>`

```
<filter-name>indexCacheFilter</filter-name>
<filter-class>
    net.sf.ehcache.constructs.web.filter.SimplePageCachingFilter
</filter-class>
</filter>
<filter-mapping>
    <filter-name>indexCacheFilter</filter-name>
    <url-pattern>*/index.action</url-pattern>
</filter-mapping>
```

就只需要这么多步骤，我们就可以给某个页面做一个缓存的，把上面这段配置放到你的 web.xml 中，那么当你打开首页的时候，你会发现，2 分钟才会有一堆 sql 语句出现在控制台上。当然你也可以调成 5 分钟，总之一切都在控制中。

好了，缓存整个页面看上去是非常的简单，甚至都不需要写一行代码，只需要几行配置就行了，够简单吧，虽然看上去简单，但是事实上内部实现却不简单哦，有兴趣的话，大家可以看看 SimplePageCachingFilter 继承体系的源代码。

上面的配置针对的情况是缓存首页的全部，如果你只想缓存首页的部分内容时，你需要使用 SimplePageFragmentCachingFilter 这个 filter。我们看一下如下片断：

XML/HTML 代码

```
<filter>
    <filter-name>indexCacheFilter</filter-name>
    <filter-class>
        net.sf.ehcache.constructs.web.filter.SimplePageFragmentCachingFilter
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>indexCacheFilter</filter-name>
    <url-pattern>*/index_right.jsp</url-pattern>
</filter-mapping>
```

这个 jsp 需要被 jsp:include 到其他页面，这样就做到的局部页面的缓存。这一点貌似没有 oscache 的 tag 好用。

事实上在 cachefilter 中还有一个特性，就是 gzip，也就是说缓存中的元素是被压缩过的，如果客户浏览器支持压缩的话，filter 会直接返回压缩过的流，这样节省了带宽，把解压的工作交给了客户浏览器，如果客户的浏览器不支持 gzip，那么 filter 会把缓存的元素拿出来解压后再返回给客户浏览器（大多数爬虫是不支持 gzip 的，所以 filter 也会解压后

再返回流), 这样做的优点是节省带宽, 缺点就是增加了客户浏览器的负担 (但是我觉得对当代的计算机而言, 这个负担微乎其微)。

好了, 如果你的页面正好也需要用到页面缓存, 不妨可以考虑一下 ehcache, 因为它实在是非常简单, 而且易用。

总结: ehcache 是一个非常轻量级的缓存实现, 而且从 1.2 之后就支持了集群, 目前的最新版本是 1.3, 而且是 hibernate 默认的缓存 provider。虽然本文是介绍的是 ehcache 对页面缓存的支持, 但是 ehcache 的功能远不止如此, 当然要使用好缓存, 对 JEE 中缓存的原理, 使用范围, 适用场景等等都需要有比较深刻的理解, 这样才能用好缓存, 用对缓存。

最后复习一下 ehcache 中缓存的 3 种清空策略:

1 FIFO, first in first out, 这个是大家最熟的, 先进先出, 不多讲了

2 LFU, Less Frequently Used, 就是上面例子中使用的策略, 直白一点就是讲一直以来最少被使用的。如上面所讲, 缓存的元素有一个 hit 属性, hit 值最小的将会被清出缓存。

2 LRU, Least Recently Used, 最近最少使用的, 缓存的元素有一个时间戳, 当缓存容量满了, 而又需要腾出地方来缓存新的元素的时候, 那么现有缓存元素中时间戳离当前时间最远的元素将被清出缓存。

2.4 在工程中单独使用

1. 创建 CacheManager (net.sf.ehcache.CacheManager)

(1) 使用默认配置文件创建

```
CacheManager manager = CacheManager.create();
```

(2) 使用指定配置文件创建

```
CacheManager manager = CacheManager.create("src/config/ehcache.xml");
```

(3) 从 classpath 找寻配置文件并创建

```
URL url = getClass().getResource("/anothername.xml");
```

```
CacheManager manager = CacheManager.create(url);
```

(4) 通过输入流创建

```
InputStream fis = new FileInputStream(new  
File("src/config/ehcache.xml").getAbsolutePath());  
try { manager = CacheManager.create(fis); } finally { fis.close(); }
```

2. 创建 Caches (net.sf.ehcache.Cache)

(1) 取得配置文件中预先 定义的 sampleCache1 设置, 生成一个 Cache

```
Cache cache = manager.getCache("sampleCache1");
```

(2) 设置一个名为 test 的新 cache, test 属性为默认

```
CacheManager manager = CacheManager.create();
```

```
manager.addCache("test");
```

(3) 设置一个名为 test 的新 cache,并定义其属性

```
CacheManager manager = CacheManager.create();
Cache cache = new Cache("test", 1, true, false, 5, 2);
manager.addCache(cache);
```

(4) 删除 cache

```
CacheManager singletonManager = CacheManager.create();
singletonManager.removeCache("sampleCache1");
```

3.使用 Caches

(1) 往 cache 中加入元素

```
Element element = new Element("key1", "value1");
cache.put(new Element(element);
```

(2) 从 cache 中取得元素

```
Element element = cache.get("key1");
```

(3) 从 cache 中删除元素

```
Cache cache = manager.getCache("sampleCache1");
Element element = new Element("key1", "value1");
cache.remove("key1");
```

3.卸载 CacheManager ,关闭 Cache

```
manager.shutdown();
```

3. 在 Spring 中运用 EHCache

需要使用 Spring 来实现一个 Cache 简单的解决方案,具体需求如下:使用任意一个现有开源 Cache Framework,要求可以 Cache 系统中 Service 或则 DAO 层的 get/find 等方法返回结果,如果数据更新(使用 Create/update/delete 方法),则刷新 cache 中相应的内容。

根据需求,计划使用 Spring AOP + ehCache 来实现这个功能,采用 ehCache 原因之一是 Spring 提供了 ehCache 的支持,至于为何仅仅支持 ehCache 而不支持 osCache 和 JBossCache 无从得知(Hibernate???),但毕竟 Spring 提供了支持,可以减少一部分工作量:。二是后来实现了 OSCache 和 JBoss Cache 的方式后,经过简单测试发现几个 Cache 在效率上没有太大的区别(不考虑集群),决定采用 ehCache。

AOP 嘛,少不了拦截器,先创建一个实现了 MethodInterceptor 接口的拦截器,用来拦截 Service/DAO 的方法调用,拦截到方法后,搜索该方法的结果在 cache 中是否存在,如果存在,返回 cache 中的缓存结果,如果不存在,返回查询数据库的结果,并将结果缓存到 cache 中。

MethodCacheInterceptor.java

Java 代码

```
package com.co.cache.ehcache;
import java.io.Serializable;
import net.sf.ehcache.Cache;
import net.sf.ehcache.Element;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.util.Assert;
public class MethodCacheInterceptor implements MethodInterceptor, InitializingBean
{
    private static final Log logger = LogFactory.getLog(MethodCacheInterceptor.class);

    private Cache cache;

    public void setCache(Cache cache) {
        this.cache = cache;
    }

    public MethodCacheInterceptor() {
        super();
    }

    /**
     * 拦截 Service/DAO 的方法，并查找该结果是否存在，如果存在就返回 cache 中的值，
     * 否则，返回数据库查询结果，并将查询结果放入 cache
     */
    public Object invoke(MethodInvocation invocation) throws Throwable {
        String targetName = invocation.getThis().getClass().getName();
        String methodName = invocation.getMethod().getName();
        Object[] arguments = invocation.getArguments();
        Object result;

        logger.debug("Find object from cache is " + cache.getName());

        String cacheKey = getCacheKey(targetName, methodName, arguments);
        Element element = cache.get(cacheKey);
```

```

        if (element == null) {
            logger.debug("Hold up method , Get method result and create cache.....!");
            result = invocation.proceed();
            element = new Element(cacheKey, (Serializable) result);
            cache.put(element);
        }
        return element.getValue();
    }

    /**
     * 获得 cache key 的方法， cache key 是 Cache 中一个 Element 的唯一标识
     * cache key 包括 包名+类名+方法名，如 com.co.cache.service.UserServiceImpl.getAllUser
     */
    private String getCacheKey(String targetName, String methodName, Object[] arguments) {
        StringBuffer sb = new StringBuffer();
        sb.append(targetName).append(".").append(methodName);
        if ((arguments != null) && (arguments.length != 0)) {
            for (int i = 0; i < arguments.length; i++) {
                sb.append(".").append(arguments[i]);
            }
        }
        return sb.toString();
    }

    /**
     * implement InitializingBean，检查 cache 是否为空
     */
    public void afterPropertiesSet() throws Exception {
        Assert.notNull(cache, "Need a cache. Please use setCache(Cache) create it.");
    }

}

```

上面的代码中可以看到，在方法 `public Object invoke(MethodInvocation invocation)` 中，完成了搜索 Cache/新建 cache 的功能。

```
Element element = cache.get(cacheKey);
```

这句代码的作用是获取 cache 中的 element，如果 cacheKey 所对应的 element 不存在，将会返回一个 null 值

Java 代码

```
result = invocation.proceed();
```

这句代码的作用是获取所拦截方法的返回值，详细请查阅 AOP 相关文档。

随后,再建立一个拦截器 MethodCacheAfterAdvice,作用是在用户进行 create/update/delete 操作时来刷新/remove 相关 cache 内容,这个拦截器实现了 AfterReturningAdvice 接口,将会在所拦截的方法执行后执行在 public void afterReturning(Object arg0, Method arg1, Object[] arg2, Object arg3)方法中所预定的操作

Java 代码

```
package com.co.cache.ehcache;
```

```
import java.lang.reflect.Method;
```

```
import java.util.List;
```

```
import net.sf.ehcache.Cache;
```

```
import org.apache.commons.logging.Log;
```

```
import org.apache.commons.logging.LogFactory;
```

```
import org.springframework.aop.AfterReturningAdvice;
```

```
import org.springframework.beans.factory.InitializingBean;
```

```
import org.springframework.util.Assert;
```

```
public class MethodCacheAfterAdvice implements AfterReturningAdvice, InitializingBean  
{
```

```
private static final Log logger = LogFactory.getLog(MethodCacheAfterAdvice.class);
```

```
private Cache cache;
```

```

public void setCache(Cache cache) {
    this.cache = cache;
}

public MethodCacheAfterAdvice() {
    super();
}

public void afterReturning(Object arg0, Method arg1, Object[] arg2, Object arg3) throws
Throwable {
    String className = arg3.getClass().getName();
    List list = cache.getKeys();
    for(int i = 0;i<list.size();i++){
        String cacheKey = String.valueOf(list.get(i));
        if(cacheKey.startsWith(className)){
            cache.remove(cacheKey);
            logger.debug("remove cache " + cacheKey);
        }
    }
}

public void afterPropertiesSet() throws Exception {
    Assert.notNull(cache, "Need a cache. Please use setCache(Cache) create it.");
}

}

```

上面的代码很简单，实现了 `afterReturning` 方法实现自 `AfterReturningAdvice` 接口，方法中所定义的内容将会在目标方法执行后执行，在该方法中的作用是获取目标 class 的全名，如：`com.co.cache.test.TestServiceImpl`，然后循环 cache 的 key list，remove cache 中所有和该 class 相关的 element。

Java 代码

```
String className = arg3.getClass().getName();
```

随后，开始配置 ehCache 的属性，ehCache 需要一个 xml 文件来设置 ehCache 相关的一些属性，如最大缓存数量、cache 刷新的时间等等。

ehcache.xml

Java 代码

```
<ehcache>
<diskStore path="c:\\myapp\\cache"/>
<defaultCache
maxElementsInMemory="1000"
eternal="false"
timeToIdleSeconds="120"
timeToLiveSeconds="120"
overflowToDisk="true"
/>
<cache name="DEFAULT_CACHE"
maxElementsInMemory="10000"
eternal="false"
timeToIdleSeconds="300000"
timeToLiveSeconds="600000"
overflowToDisk="true"
/>
</ehcache>
```

配置每一项的详细作用不再详细解释，有兴趣的请 google 下，这里需要注意一点 defaultCache 标签定义了一个默认的 Cache，这个 Cache 是不能删除的，否则会抛出 No default cache is configured 异常。另外，由于使用拦截器来刷新 Cache 内容，因此在定义 cache 生命周期时可以定义较大的数值，timeToIdleSeconds="300000" timeToLiveSeconds="600000"，好像还不够大？

然后，在将 Cache 和两个拦截器配置到 Spring，这里没有使用 2.0 里面 AOP 的标签。
cacheContext.xml

Java 代码

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<!-- 引用 ehCache 的配置 -->
<bean id="defaultCacheManager"
```

```
class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
    <property name="configLocation">
        <value>ehcache.xml</value>
    </property>
</bean>

<!-- 定义 ehCache 的工厂，并设置所使用的 Cache name -->
<bean id="ehCache" class="org.springframework.cache.ehcache.EhCacheFactoryBean">
    <property name="cacheManager">
        <ref local="defaultCacheManager"/>
    </property>
    <property name="cacheName">
        <value>DEFAULT_CACHE</value>
    </property>
</bean>

<!-- find/create cache 拦截器 -->
<bean id="methodCacheInterceptor"
class="com.co.cache.ehcache.MethodCacheInterceptor">
    <property name="cache">
        <ref local="ehCache" />
    </property>
</bean>

<!-- flush cache 拦截器 -->
<bean id="methodCacheAfterAdvice"
class="com.co.cache.ehcache.MethodCacheAfterAdvice">
    <property name="cache">
        <ref local="ehCache" />
    </property>
</bean>

<bean id="methodCachePointCut"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref local="methodCacheInterceptor"/>
    </property>
    <property name="patterns">
        <list>
            <value>.*find.*</value>
            <value>.*get.*</value>
        </list>
    </property>
</bean>
```

```
</list>
</property>
</bean>
<bean id="methodCachePointCutAdvice"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
  <property name="advice">
    <ref local="methodCacheAfterAdvice"/>
  </property>
  <property name="patterns">
    <list>
      <value>.*create.*</value>
      <value>.*update.*</value>
      <value>.*delete.*</value>
    </list>
  </property>
</bean>
</beans>
```

上面的代码最终创建了两个"切入点", `methodCachePointCut` 和 `methodCachePointCutAdvice`, 分别用于拦截不同方法名的方法, 可以根据需要任意增加所需要拦截方法的名称。

需要注意的是

Java 代码

```
<bean id="ehCache" class="org.springframework.cache.ehcache.EhCacheFactoryBean">
  <property name="cacheManager">
    <ref local="defaultCacheManager"/>
  </property>
  <property name="cacheName">
    <value>DEFAULT_CACHE</value>
  </property>
</bean>
```

如果 `cacheName` 属性内设置的 `name` 在 `ehCache.xml` 中无法找到, 那么将使用默认的 `cache`(`defaultCache` 标签定义).

事实上到了这里, 一个简单的 Spring + ehCache Framework 基本完成了, 为了测试效果,

举一个实际应用的例子，定义一个 `TestService` 和它的实现类 `TestServiceImpl`，里面包含

两个方法 `getAllObject()`和 `updateObject(Object Object)`，具体代码如下

`TestService.java`

Java 代码

```
package com.co.cache.test;
```

```
import java.util.List;
```

```
public interface TestService {  
    public List getAllObject();
```

```
    public void updateObject(Object Object);  
}
```

`TestServiceImpl.java`

Java 代码

```
package com.co.cache.test;
```

```
import java.util.List;
```

```
public class TestServiceImpl implements TestService  
{
```

```
    public List getAllObject() {  
        System.out.println("---TestService: Cache 内不存在该 element，查找并放入 Cache! ");  
        return null;  
    }
```

```
    public void updateObject(Object Object) {  
        System.out.println("---TestService: 更新了对象, 这个 Class 产生的 cache 都将被 remove !  
");  
    }  
}
```

使用 Spring 提供的 AOP 进行配置
applicationContext.xml

XML/HTML 代码

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
<import resource="cacheContext.xml"/>

<bean id="testServiceTarget" class="com.co.cache.test.TestServiceImpl"/>

<bean id="testService" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref local="testServiceTarget"/>
    </property>
    <property name="interceptorNames">
        <list>
            <value>methodCachePointCut</value>
            <value>methodCachePointCutAdvice</value>
        </list>
    </property>
</bean>
</beans>
```

这里一定不能忘记 import cacheContext.xml 文件，不然定义的两个拦截器就没办法使用了。

最后，写一个测试的代码
MainTest.java

Java 代码

```
(DEFAULT_CONTEXT_FILE);
    TestService testService = (TestService)context.getBean("testService");
```

```
System.out.println("1--第一次查找并创建 cache");
testService.getAllObject();

System.out.println("2--在 cache 中查找");
testService.getAllObject();

System.out.println("3--remove cache");
testService.updateObject(null);

System.out.println("4--需要重新查找并创建 cache");
testService.getAllObject();
}
}
```

运行，结果如下

Java 代码

```
1--第一次查找并创建 cache
---TestService: Cache 内不存在该 element，查找并放入 Cache!
2--在 cache 中查找
3--remove cache
---TestService: 更新了对象，这个 Class 产生的 cache 都将被 remove!
4--需要重新查找并创建 cache
---TestService: Cache 内不存在该 element，查找并放入 Cache!
```

大功告成。可以看到，第一步执行 `getAllObject()`，执行 `TestServiceImpl` 内的方法，并创建了 cache，在第二次执行 `getAllObject()` 方法时，由于 cache 有该方法的缓存，直接从 cache 中 get 出方法的结果，所以没有打印出 `TestServiceImpl` 中的内容，而第三步，调用了 `updateObject` 方法，和 `TestServiceImpl` 相关的 cache 被 remove，所以在第四步执行时，又执行 `TestServiceImpl` 中的方法，创建 Cache。

网上也有不少类似的例子，但是很多都不是很完备，自己参考了一些例子的代码，其实在 `spring-modules` 中也提供了几种 cache 的支持，`ehCache`，`OSCache`，`JBossCache` 这些，看了一下，基本上都是采用类似的方式，只不过封装的更完善一些，主要思路也还是 Spring 的 AOP，有兴趣的可以研究一下。

4. 分布式缓存集群环境配置

4.1 集群配置方式

ehcache 提供三种网络连接策略来实现集群，rmi,jgroup 还有 jms。这里只说 rmi 方式。同时 ehcache 可以实现多播的方式实现集群。也可以手动指定集群主机序列实现集群，本例应用手动指定。

这里说点题外话，本来看着分发包中的原来的例子配置是一件不怎么难的事情，应该很容易就能实现。但是一开始，我是在我的 linux 主机上和我的主操作系统 windows 上实现集群配置。结果反过来弄过去，都没有成功。然后在网上找一些别人的配置经验，竟然都是配置片段，没有完整的实例文件。结果配置半天没成功。但我怀疑是我的 linux 系统有些地方可能没有配置好，于是先不管他。有开启了我的另一个 windows 主机。然后把程序部署上去，竟然一次试验成功。高兴的同时，我得发句话“不要把代码片段称作实例，这很不负责任”。同时还存在一个问题，在 linux 下没有部署成功的原因有待查明。


具体说明：配置 cacheManagerPeerListenerFactory 是配宿主主机配置监听程序，来发现其他主机发来的同步请求配置 cacheManagerPeerProviderFactory 是指定除自身之外的网络群体中其他提供同步的主机列表，用“|”分开不同的主机。

下面的例子的测试过程是：主机 B 缓存开启，并从名为 UserCache 的缓存中循环抓取键值为“key1”的元素，直到取到，才退出循环。主机 A 缓存启动，并在名为 UserCache 的缓存中放入键值为“key1”的元素。显然，如果主机 B 取到的元素，那么就证明同步成功，也就是集群成功。所以在测试过程中先启动主机 B 的测试程序，在启动主机 A 的测试程序。

下面具体说配置文件以及测试程序：

1. 主机 A 的配置文件以及测试源代码

config/ehcache_cluster.xml

Xml 代码 


1. `<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance`
2. `xsi:noNamespaceSchemaLocation="ehcache.xsd">`
3. `<cacheManagerPeerProviderFactory`
4. `class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"`

```

5.         properties="peerDiscovery=manual,
6.         rmiUrls=//192.168.1.254:40000/UserCache" />
7.
8.     <cacheManagerPeerListenerFactory
9.         class="net.sf.ehcache.distribution.RMICacheManagerPee
rListenerFactory"
10.        properties="hostName=192.168.1.126, port=40000, socket
TimeoutMillis=120000" />
11.
12.    <defaultCache maxElementsInMemory="10000" eternal="false
"
13.        timeToIdleSeconds="120" timeToLiveSeconds="120" over
flowToDisk="true"
14.        diskSpoolBufferSizeMB="30" maxElementsOnDisk="100000
00"
15.        diskPersistent="false" diskExpiryThreadIntervalSecon
ds="120"
16.        memoryStoreEvictionPolicy="LRU">
17.        <cacheEventListenerFactory
18.            class="net.sf.ehcache.distribution.RMICacheRepli
catorFactory" />
19.    </defaultCache>
20.
21.    <cache name="UserCache" maxElementsInMemory="1000" etern
al="false"
22.        timeToIdleSeconds="100000" timeToLiveSeconds="100000
"
23.        overflowToDisk="false">
24.        <cacheEventListenerFactory
25.            class="net.sf.ehcache.distribution.RMICacheRepli
catorFactory" />
26.    </cache>
27. </ehcache>

```

tutorial/UsingCacheCluster

Java 代码 

```

1. package tutorial;
2.

```


```

3. import java.net.URL;
4.
5. import net.sf.ehcache.Cache;
6. import net.sf.ehcache.CacheManager;
7. import net.sf.ehcache.Element;
8.
9. public class UsingCacheCluster {
10.
11.     public static void main(String[] args) throws Exception
12.     {
13.         URL url = UsingCacheCluster.class.getClassLoader().getResource(
14.             "config/ehcache_cluster.xml");
15.         CacheManager manager = new CacheManager(url);
16.         //取得 Cache
17.         Cache cache = manager.getCache("UserCache");
18.         Element element = new Element("key1", "value1");
19.         cache.put(element);
20.
21.         Element element1 = cache.get("key1");
22.         System.out.println(element1.getValue());
23.     }
24. }

```

2. 主机 B 上的配置文件以及测试代码

config/ehcache_cluster.xml

Xml 代码 

```

1. <ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
2.     xsi:noNamespaceSchemaLocation="ehcache.xsd">
3.     <cacheManagerPeerProviderFactory
4.         class="net.sf.ehcache.distribution.RMICacheManagerPeer
5.             rProviderFactory"
6.         properties="peerDiscovery=manual,
7.             rmiUrls=//192.168.1.126:40000/UserCache" />


```

```

8.      <cacheManagerPeerListenerFactory
9.          class="net.sf.ehcache.distribution.RMCacheManagerPeer
    rListenerFactory"
10.          properties="hostName=192.168.1.254,port=40000, socket
    tTimeoutMillis=120000" />
11.
12.      <defaultCache maxElementsInMemory="10000" eternal="false
    "
13.          timeToIdleSeconds="120" timeToLiveSeconds="120" over
    flowToDisk="true"
14.          diskSpoolBufferSizeMB="30" maxElementsOnDisk="100000
    00"
15.          diskPersistent="false" diskExpiryThreadIntervalSecon
    ds="120"
16.          memoryStoreEvictionPolicy="LRU">
17.          <cacheEventListenerFactory
18.              class="net.sf.ehcache.distribution.RMCacheRepli
    catorFactory" />
19.      </defaultCache>
20.
21.      <cache name="UserCache" maxElementsInMemory="1000" etern
    al="false"
22.          timeToIdleSeconds="100000" timeToLiveSeconds="100000
    "
23.          overflowToDisk="false">
24.          <cacheEventListenerFactory
25.              class="net.sf.ehcache.distribution.RMCacheRepli
    catorFactory" />
26.      </cache>
27. </ehcache>

```

tutorial/UsingCacheCluster

Java 代码 

```

1. package tutorial;
2.

```

```
3. import java.net.URL;
4.
5. import net.sf.ehcache.Cache;
6. import net.sf.ehcache.CacheManager;
7. import net.sf.ehcache.Element;
8.
9. public class UsingCacheCluster {
10.
11.     public static void main(String[] args) throws Exception
12.     {
13.         URL url = UsingCacheCluster.class.getClassLoader().getResource(
14.             "config/ehcache_cluster.xml");
15.         CacheManager manager = new CacheManager(url);
16.         //取得 Cache
17.         Cache cache = manager.getCache("UserCache");
18.         while(true) {
19.             Element e = cache.get("key1");
20.             if(e != null) {
21.                 System.out.println(e.getValue());
22.                 break;
23.             }
24.             Thread.sleep(1000);
25.         }
26.     }
27.
28. }
```

5. 测试用例

直接进入测试文档 [EHCACHE 测试文档.docx](#)