

- [CSDN 首页](#)
- [资讯](#)
- [论坛](#)
- [博客](#)
- [下载](#)
- [搜索](#)

[更多](#)

# 先相信你自己，然后别人才会相信你。

-  [目录视图](#)
-  [摘要视图](#)
-  [RSS 订阅](#)

用开源 **IaaS** 构建自己的云——**OpenStack** 征稿启事

不用买彩票，就有**408万**！

CSDN 博客频道“移动开发之我见”主题征文活动

**2012CSDN 网站八大职位急聘**

## Spring + Ehcache 配置

分类: [Cache](#) 2010-07-22 15:20 529人阅读 [评论\(0\)](#) [收藏](#) [举报](#)

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

```
1  package com.co.cache.ehcache;
2
3  import java.io.Serializable;
4
5  import net.sf.ehcache.Cache;
6  import net.sf.ehcache.Element;
7
8  import org.aopalliance.intercept.MethodInterceptor;
9  import org.aopalliance.intercept.MethodInvocation;
10 import org.apache.commons.logging.Log;
11 import org.apache.commons.logging.LogFactory;
12 import org.springframework.beans.factory.InitializingBean;
13 import org.springframework.util.Assert;
14
15 public class MethodCacheInterceptor implements MethodInterceptor,
InitializingBean
16 {
17     private static final Log logger =
LogFactory.getLog(MethodCacheInterceptor.class);
18
19     private Cache cache;
20
21     public void setCache(Cache cache) {
22         this.cache = cache;
23     }
24
25     public MethodCacheInterceptor() {
26         super();
27     }
28
29     /**
30      * 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，
31      * 否则，返回数据库查询结果，并将查询结果放入 cache
32      */
33     public Object invoke(MethodInvocation invocation) throws Throwable {
34         String targetName = invocation.getThis().getClass().getName();
35         String methodName = invocation.getMethod().getName();
36         Object[] arguments = invocation.getArguments();
37         Object result;
38
39         logger.debug("Find object from cache is " + cache.getName());
```

```

40
41     String cacheKey = getCacheKey(targetName, methodName, arguments);
42     Element element = cache.get(cacheKey);
43
44     if (element == null) {
45         logger.debug("Hold up method , Get method result and create
46 cache.....!");
47         result = invocation.proceed();
48         element = new Element(cacheKey, (Serializable) result);
49         cache.put(element);
50     }
51     return element.getValue();
52 }
53 /**
54  * 获得 cache key 的方法, cache key 是 Cache 中一个 Element 的唯一标识
55  * cache key 包括 包名+类名+方法名, 如
56 com.co.cache.service.UserServiceImpl.getAllUser
57  */
58 private String getCacheKey(String targetName, String methodName, Object[]
59 arguments) {
60     StringBuffer sb = new StringBuffer();
61     sb.append(targetName).append(".").append(methodName);
62     if ((arguments != null) && (arguments.length != 0)) {
63         for (int i = 0; i < arguments.length; i++) {
64             sb.append(".").append(arguments[i]);
65         }
66     }
67     return sb.toString();
68 }
69 /**
70  * implement InitializingBean, 检查 cache 是否为空
71  */
72 public void afterPropertiesSet() throws Exception {
73     Assert.notNull(cache, "Need a cache. Please use setCache(Cache) create
74 it.");
75 }

```

```

76 package com.co.cache.ehcache;

```

```
77
78 import java.io.Serializable;
79
80 import net.sf.ehcache.Cache;
81 import net.sf.ehcache.Element;
82
83 import org.aopalliance.intercept.MethodInterceptor;
84 import org.aopalliance.intercept.MethodInvocation;
85 import org.apache.commons.logging.Log;
86 import org.apache.commons.logging.LogFactory;
87 import org.springframework.beans.factory.InitializingBean;
88 import org.springframework.util.Assert;
89
90 public class MethodCacheInterceptor implements MethodInterceptor,
InitializingBean
91 {
92     private static final Log logger =
LogFactory.getLog(MethodCacheInterceptor.class);
93
94     private Cache cache;
95
96     public void setCache(Cache cache) {
97         this.cache = cache;
98     }
99
100    public MethodCacheInterceptor() {
101        super();
102    }
103
104    /**
105     * 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，
106     * 否则，返回数据库查询结果，并将查询结果放入 cache
107     */
108    public Object invoke(MethodInvocation invocation) throws Throwable {
109        String targetName = invocation.getThis().getClass().getName();
110        String methodName = invocation.getMethod().getName();
111        Object[] arguments = invocation.getArguments();
112        Object result;
113
114        logger.debug("Find object from cache is " + cache.getName());
115
116        String cacheKey = getCacheKey(targetName, methodName, arguments);
117        Element element = cache.get(cacheKey);
118
```

```

119         if (element == null) {
120             logger.debug("Hold up method , Get method result and create
cache.....!");
121             result = invocation.proceed();
122             element = new Element(cacheKey, (Serializable) result);
123             cache.put(element);
124         }
125         return element.getValue();
126     }
127
128     /**
129      * 获得 cache key 的方法, cache key 是 Cache 中一个 Element 的唯一标识
130      * cache key 包括 包名+类名+方法名, 如
com.co.cache.service.UserServiceImpl.getAllUser
131      */
132     private String getCacheKey(String targetName, String methodName, Object[]
arguments) {
133         StringBuffer sb = new StringBuffer();
134         sb.append(targetName).append(".").append(methodName);
135         if ((arguments != null) && (arguments.length != 0)) {
136             for (int i = 0; i < arguments.length; i++) {
137                 sb.append(".").append(arguments[i]);
138             }
139         }
140         return sb.toString();
141     }
142
143     /**
144      * implement InitializingBean, 检查 cache 是否为空
145      */
146     public void afterPropertiesSet() throws Exception {
147         Assert.notNull(cache, "Need a cache. Please use setCache(Cache) create
it.");
148     }
149
150 }

```

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。

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

```
152 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

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

```
154 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

```
155 package com.co.cache.ehcache;
156
157 import java.lang.reflect.Method;
158 import java.util.List;
159
160 import net.sf.ehcache.Cache;
161
162 import org.apache.commons.logging.Log;
163 import org.apache.commons.logging.LogFactory;
164 import org.springframework.aop.AfterReturningAdvice;
165 import org.springframework.beans.factory.InitializingBean;
166 import org.springframework.util.Assert;
167
```

```

168 public class MethodCacheAfterAdvice implements AfterReturningAdvice,
InitializingBean
169 {
170     private static final Log logger =
LogFactory.getLog(MethodCacheAfterAdvice.class);
171
172     private Cache cache;
173
174     public void setCache(Cache cache) {
175         this.cache = cache;
176     }
177
178     public MethodCacheAfterAdvice() {
179         super();
180     }
181
182     public void afterReturning(Object arg0, Method arg1, Object[] arg2, Object
arg3) throws Throwable {
183         String className = arg3.getClass().getName();
184         List list = cache.getKeys();
185         for(int i = 0;i<list.size();i++){
186             String cacheKey = String.valueOf(list.get(i));
187             if(cacheKey.startsWith(className)){
188                 cache.remove(cacheKey);
189                 logger.debug("remove cache " + cacheKey);
190             }
191         }
192     }
193
194     public void afterPropertiesSet() throws Exception {
195         Assert.notNull(cache, "Need a cache. Please use setCache(Cache) create
it.");
196     }
197
198 }

```

```

199 package com.co.cache.ehcache;
200
201 import java.lang.reflect.Method;
202 import java.util.List;
203
204 import net.sf.ehcache.Cache;

```

```
205
206 import org.apache.commons.logging.Log;
207 import org.apache.commons.logging.LogFactory;
208 import org.springframework.aop.AfterReturningAdvice;
209 import org.springframework.beans.factory.InitializingBean;
210 import org.springframework.util.Assert;
211
212 public class MethodCacheAfterAdvice implements AfterReturningAdvice,
InitializingBean
213 {
214     private static final Log logger =
LogFactory.getLog(MethodCacheAfterAdvice.class);
215
216     private Cache cache;
217
218     public void setCache(Cache cache) {
219         this.cache = cache;
220     }
221
222     public MethodCacheAfterAdvice() {
223         super();
224     }
225
226     public void afterReturning(Object arg0, Method arg1, Object[] arg2, Object
arg3) throws Throwable {
227         String className = arg3.getClass().getName();
228         List list = cache.getKeys();
229         for(int i = 0;i<list.size();i++){
230             String cacheKey = String.valueOf(list.get(i));
231             if(cacheKey.startsWith(className)){
232                 cache.remove(cacheKey);
233                 logger.debug("remove cache " + cacheKey);
234             }
235         }
236     }
237
238     public void afterPropertiesSet() throws Exception {
239         Assert.notNull(cache, "Need a cache. Please use setCache(Cache) create
it.");
240     }
241
242 }
```



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

Java 代码

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

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

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`。

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

```
245 <ehcache>
246   <diskStore path="c://myapp//cache"/>
247   <defaultCache
248     maxElementsInMemory="1000"
249     eternal="false"
250     timeToIdleSeconds="120"
251     timeToLiveSeconds="120"
252     overflowToDisk="true"
253   />
254   <cache name="DEFAULT_CACHE"
255     maxElementsInMemory="10000"
256     eternal="false"
257     timeToIdleSeconds="300000"
258     timeToLiveSeconds="600000"
259     overflowToDisk="true"
260   />
261 </ehcache>
```

```
262 <ehcache>
```

```

263     <diskStore path="c://myapp//cache"/>
264     <defaultCache
265         maxElementsInMemory="1000"
266         eternal="false"
267         timeToIdleSeconds="120"
268         timeToLiveSeconds="120"
269         overflowToDisk="true"
270     />
271     <cache name="DEFAULT_CACHE"
272         maxElementsInMemory="10000"
273         eternal="false"
274         timeToIdleSeconds="300000"
275         timeToLiveSeconds="600000"
276         overflowToDisk="true"
277     />
278 </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

```

279 <?xml version="1.0" encoding="UTF-8"?>
280 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
281 "http://www.springframework.org/dtd/spring-beans.dtd">
282 <beans>
283     <!-- 引用 ehCache 的配置 -->
284     <bean id="defaultCacheManager"
285         class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
286         <property name="configLocation">
287             <value>ehcache.xml</value>
288         </property>
289     </bean>
290     <!-- 定义 ehCache 的工厂，并设置所使用的 Cache name -->
291     <bean id="ehCache"
292         class="org.springframework.cache.ehcache.EhCacheFactoryBean">

```

```
291     <property name="cacheManager">
292         <ref local="defaultCacheManager"/>
293     </property>
294     <property name="cacheName">
295         <value>DEFAULT_CACHE</value>
296     </property>
297 </bean>
298
299 <!-- find/create cache 拦截器 -->
300 <bean id="methodCacheInterceptor"
301     class="com.co.cache.ehcache.MethodCacheInterceptor">
302     <property name="cache">
303         <ref local="ehCache" />
304     </property>
305 </bean>
306 <!-- flush cache 拦截器 -->
307 <bean id="methodCacheAfterAdvice"
308     class="com.co.cache.ehcache.MethodCacheAfterAdvice">
309     <property name="cache">
310         <ref local="ehCache" />
311     </property>
312 </bean>
313
314 <bean id="methodCachePointCut"
315     class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
316     <property name="advice">
317         <ref local="methodCacheInterceptor"/>
318     </property>
319     <property name="patterns">
320         <list>
321             <value>.*find.*</value>
322             <value>.*get.*</value>
323         </list>
324     </property>
325 </bean>
326
327 <bean id="methodCachePointCutAdvice"
328     class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
329     <property name="advice">
330         <ref local="methodCacheAfterAdvice"/>
331     </property>
332     <property name="patterns">
333         <list>
334             <value>.*create.*</value>
335             <value>.*update.*</value>
```

```

331         <value>.*delete.*</value>
332     </list>
333 </property>
334 </bean>
335 </beans>

```

```

336 <?xml version="1.0" encoding="UTF-8"?>
337 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
338 "http://www.springframework.org/dtd/spring-beans.dtd">
339 <beans>
340     <!-- 引用 ehCache 的配置 -->
341     <bean id="defaultCacheManager"
342         class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
343         <property name="configLocation">
344             <value>ehcache.xml</value>
345         </property>
346     </bean>
347     <!-- 定义 ehCache 的工厂，并设置所使用的 Cache name -->
348     <bean id="ehCache"
349         class="org.springframework.cache.ehcache.EhCacheFactoryBean">
350         <property name="cacheManager">
351             <ref local="defaultCacheManager"/>
352         </property>
353         <property name="cacheName">
354             <value>DEFAULT_CACHE</value>
355         </property>
356     </bean>
357     <!-- find/create cache 拦截器 -->
358     <bean id="methodCacheInterceptor"
359         class="com.co.cache.ehcache.MethodCacheInterceptor">
360         <property name="cache">
361             <ref local="ehCache" />
362         </property>
363     </bean>
364     <!-- flush cache 拦截器 -->
365     <bean id="methodCacheAfterAdvice"
366         class="com.co.cache.ehcache.MethodCacheAfterAdvice">
367         <property name="cache">
368             <ref local="ehCache" />
369         </property>

```

```

367     </bean>
368
369     <bean id="methodCachePointCut"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
370         <property name="advice">
371             <ref local="methodCacheInterceptor"/>
372         </property>
373         <property name="patterns">
374             <list>
375                 <value>.*find.*</value>
376                 <value>.*get.*</value>
377             </list>
378         </property>
379     </bean>
380     <bean id="methodCachePointCutAdvice"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
381         <property name="advice">
382             <ref local="methodCacheAfterAdvice"/>
383         </property>
384         <property name="patterns">
385             <list>
386                 <value>.*create.*</value>
387                 <value>.*update.*</value>
388                 <value>.*delete.*</value>
389             </list>
390         </property>
391     </bean>
392 </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

```

393 <bean id="ehCache"
class="org.springframework.cache.ehcache.EhCacheFactoryBean">
394     <property name="cacheManager">
395         <ref local="defaultCacheManager"/>
396     </property>
397     <property name="cacheName">
398         <value>DEFAULT_CACHE</value>

```

```
399     </property>
400 </bean>
```

```
401 <bean id="ehCache"
class="org.springframework.cache.ehcache.EhCacheFactoryBean">
402     <property name="cacheManager">
403         <ref local="defaultCacheManager"/>
404     </property>
405     <property name="cacheName">
406         <value>DEFAULT_CACHE</value>
407     </property>
408 </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

```
409 package com.co.cache.test;
410
411 import java.util.List;
412
413 public interface TestService {
414     public List getAllObject();
415
416     public void updateObject(Object Object);
417 }
```

```
418 package com.co.cache.test;
419
420 import java.util.List;
```

```
421
422 public interface TestService {
423     public List getAllObject();
424
425     public void updateObject(Object Object);
426 }
```

## TestServiceImpl.java

Java 代码

```
427 package com.co.cache.test;
428
429 import java.util.List;
430
431 public class TestServiceImpl implements TestService
432 {
433     public List getAllObject() {
434         System.out.println("---TestService: Cache 内不存在该 element，查找并放入
Cache! ");
435         return null;
436     }
437
438     public void updateObject(Object Object) {
439         System.out.println("---TestService: 更新了对象，这个 Class 产生的 cache 都
将被 remove! ");
440     }
441 }
```

```
442 package com.co.cache.test;
443
444 import java.util.List;
445
446 public class TestServiceImpl implements TestService
447 {
448     public List getAllObject() {
449         System.out.println("---TestService: Cache 内不存在该 element，查找并放入
Cache! ");
450         return null;
451     }
452 }
```

```

451     }
452
453     public void updateObject(Object Object) {
454         System.out.println("---TestService: 更新了对象, 这个 Class 产生的 cache 都
将被 remove! ");
455     }
456 }

```

使用 Spring 提供的 AOP 进行配置  
applicationContext.xml

Java 代码

```

457 <?xml version="1.0" encoding="UTF-8"?>
458 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
459
460 <beans>
461     <import resource="cacheContext.xml"/>
462
463     <bean id="testServiceTarget" class="com.co.cache.test.TestServiceImpl"/>
464
465     <bean id="testService"
class="org.springframework.aop.framework.ProxyFactoryBean">
466         <property name="target">
467             <ref local="testServiceTarget"/>
468         </property>
469         <property name="interceptorNames">
470             <list>
471                 <value>methodCachePointCut</value>
472                 <value>methodCachePointCutAdvice</value>
473             </list>
474         </property>
475     </bean>
476 </beans>

```

```

477 <?xml version="1.0" encoding="UTF-8"?>
478 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
479

```



```

480 <beans>
481     <import resource="cacheContext.xml"/>
482
483     <bean id="testServiceTarget" class="com.co.cache.test.TestServiceImpl"/>
484
485     <bean id="testService"
class="org.springframework.aop.framework.ProxyFactoryBean">
486         <property name="target">
487             <ref local="testServiceTarget"/>
488         </property>
489         <property name="interceptorNames">
490             <list>
491                 <value>methodCachePointCut</value>
492                 <value>methodCachePointCutAdvice</value>
493             </list>
494         </property>
495     </bean>
496 </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

```

497 package com.co.cache.test;
498
499 import org.springframework.context.ApplicationContext;
500 import org.springframework.context.support.ClassPathXmlApplicationContext;
501
502 public class MainTest{
503     public static void main(String args[]){
504         String DEFAULT_CONTEXT_FILE = "/applicationContext.xml";
505         ApplicationContext context = new
ClassPathXmlApplicationContext(DEFAULT_CONTEXT_FILE);
506         TestService testService = (TestService)context.getBean("testService");
507
508         System.out.println("1--第一次查找并创建 cache");
509         testService.getAllObject();
510
511         System.out.println("2--在 cache 中查找");

```

```

512         testService.getAllObject();
513
514         System.out.println("3--remove cache");
515         testService.updateObject(null);
516
517         System.out.println("4--需要重新查找并创建 cache");
518         testService.getAllObject();
519     }
520 }

```

```

521 package com.co.cache.test;
522
523 import org.springframework.context.ApplicationContext;
524 import org.springframework.context.support.ClassPathXmlApplicationContext;
525
526 public class MainTest{
527     public static void main(String args[]){
528         String DEFAULT_CONTEXT_FILE = "/applicationContext.xml";
529         ApplicationContext context = new
ClassPathXmlApplicationContext(DEFAULT_CONTEXT_FILE);
530         TestService testService = (TestService)context.getBean("testService");
531
532         System.out.println("1--第一次查找并创建 cache");
533         testService.getAllObject();
534
535         System.out.println("2--在 cache 中查找");
536         testService.getAllObject();
537
538         System.out.println("3--remove cache");
539         testService.updateObject(null);
540
541         System.out.println("4--需要重新查找并创建 cache");
542         testService.getAllObject();
543     }
544 }

```

运行，结果如下

```
545 1--第一次查找并创建 cache
546 ---TestService: Cache 内不存在该 element，查找并放入 Cache！
547 2--在 cache 中查找
548 3--remove cache
549 ---TestService: 更新了对象，这个 Class 产生的 cache 都将被 remove！
550 4--需要重新查找并创建 cache
551 ---TestService: Cache 内不存在该 element，查找并放入 Cache！
```

```
552 1--第一次查找并创建 cache
553 ---TestService: Cache 内不存在该 element，查找并放入 Cache！
554 2--在 cache 中查找
555 3--remove cache
556 ---TestService: 更新了对象，这个 Class 产生的 cache 都将被 remove！
557 4--需要重新查找并创建 cache
558 ---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` 中也提供了对几种 `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。