一、简介

Cache 是一种用于提高系统响应速度、改善系统运行性能的技术。尤其是在 Web 应用中，通过缓存页面的输出结果，可以很显著的改善系统运行性能。

OSCache 标记库由 OpenSymphony 设计，它是一种开创性的缓存方案，它提供了在现有 JSP 页面之内实现内存缓存的功能。OSCache 是个一个被广泛采用的高性能的 J2EE 缓存框架，OSCache 还能应用于任何 Java 应用程序的普通的缓存解决方案。

OSCache 是当前运用最广的缓存方案，JBoss,Hibernate,Spring 等都对其有支持。

Oscache 的使用非常方便，特别是 jsp cache 用的非常广泛。Oscache 的文档中也对 jsp cache tag 的配置有详细说明。相关内容请参考 OSCache 的 在线文档

对使用 Cache 的测试结论：

使用 cache，随着循环的增多，用时增长较缓慢，而不使用 cache 基本是等比例增长。在循环次数较多时，使用 cache cpu 利用率显著提高，能达到 90％以上。不使用 cache 则只能上到 50%左右，更多是在等待数据库返回结果。所以使用 cache 能大大减轻数据库的压 力，提高应用服务器的利用率，符合我们对应用服务器进行水平扩展的要求。

二、OSCache 的特点和主要特征

（一）、OSCache 有以下特点

1、缓存任何对象：你可以不受限制的缓存部分 jsp 页面或 HTTP 请求，任何 java 对象都可以缓存。

2、拥有全面的 API：OSCache API 允许你通过编程的方式来控制所有的 OSCache 特性。

3、永久缓存：缓存能被配置写入硬盘，因此允许在应用服务器的多次生命周期间缓存创建开销昂贵的数据。

4、支持集群：集群缓存数据能被单个的进行参数配置，不需要修改代码。

5、缓存过期：你可以有最大限度的控制缓存对象的过期，包括可插入式的刷新策略（如果默认性能不能满足需要时）。

（二）、主要特征

1. 兼容多种支持 JSP 的 web 服务器

已经通过兼容测试的 web 服务器包括 OrionServer (1.4.0 或者以上版本) 、Macromedia JRun (3.0 或者以上版本) 、BEA Weblogic (7.x 或者以上版本) 、IBM Websphere (5.0 版本)、Silverstream (3.7.4 版本)、Caucho Resin (1.2.3 或者以上版本)、Tomcat (4.0 或者以上版本) ，其他支持 servlet2.3、jsp1.2 的 web 服务器应该都是完全兼容 OSCache 的。

2. 可选的缓存区

你可以使用内存、硬盘空间、同时使用内存和硬盘或者提供自己的其他资源（需要自己提供适配器）作为缓存区。

使用内存作为缓存区将可以提供更好的性能

使用硬盘作为缓存区可以在服务器重起后迅速恢复缓存内容

同时使用内存和硬盘作为缓存区则可以减少对内存的占用

3. 灵活的缓存系统

OSCache 支持对部分页面内容或者对页面级的响应内容进行缓存，编程者可以根据不同的需求、不同的环境选择不同的缓存级别。

4. 容错

在一般的 web 应用中，如果某个页面需要和数据库打交道，而当客户请求到达时， web 应用和数据库之间无法进行交互，那么将返回给用户"系统出错"或者类似的提示信息，如果使用了 OSCache 的话，你可以使用缓存提供给用户，给自己赢得维护系统或者采取其他补救

的时间。
其它特性还包括对集群的支持、缓存主动刷新等特性，大家可以参考 OpenSymphony 网站上的其他资源获取更多的信息。

二、OSCache 使用指南

一、下载安装
OSCache 是一个基于 web 应用的组件，他的安装工作主要是对 web 应用进行配置，大概的步骤如下：
1. 下载、解压缩 OSCache
从 http://www.opensymphony.com/oscache/download.html 下载合适的 OSCache 版本， 解压缩下载的文件到指定目录 。
2、新建立一个 web 应用
3、将 OSCache 集成到 web 项目当中。
（1）从解压缩目录取得 oscache.jar 文件放到 /WEB-INF/lib 或相应类库目录中，jar 文件名可能含有版本号和该版本的发布日期信息等。
（2）将 oscache.properties、oscache.tld 放入 WEB-INF\class 目录（确切说是放在项目的 src 目录下，编译的时候会自动生成在 WEB-INF\class 目录）。
（3）配置项目对应的 oscache.properties 参数信息。
（4）具体使用
A、缓存对象：直接调用 API 的接口即可（详见[Java]用 OSCache 进行缓存对象）
B、部分页面缓存：使用 OSCache 提供的 taglib（修改 web.xml 文件，在 web.xml 文件中增加下面的内容，增加对 OSCache 提供的 taglib 的支持： <taglib> <taglib-uri>oscache</taglib-uri> <taglib-location>/WEB-INF/classes/oscache.tld</taglib-location></taglib>或者在 jsp 页面使用以下标签
<%@ taglib uri="/WEB-INF/classes/oscache.tld" prefix="cache"%>）
C、整个页面的缓存：用 CashFilter 实现页面级缓存，可缓存单个文件、缓存 URL pattern 和自己设定缓存属性的缓存。

```
<filter>
<filter-name>CacheFilter</filter-name>
<filter-class>com.opensymphony.oscache.web.filter.CacheFilter</filter-class>
<init-param>
<param-name>time</param-name>
<param-value>600</param-value>
</init-param>
<init-param>


<param-name>scope</param-name>
<param-value>session</param-value>
</init-param>
</filter>
<filter-mapping>
<filter-name>CacheFilter</filter-name>
```

<!-对所有 jsp 页面内容进行缓存-->
<url-pattern>*.jsp</url-pattern>
</filter-mapping>
[注] 只有客户访问时返回 http 头信息中代码为 200（也就是访问已经成功）的页面信息才能够被缓存
4、配置日志输出信息。
二、oscache.properties 文件配置向导
oscache.properties 中的配置项详解:
1、cache.memory：
是否使用内存缓存; true 或 false。默认为 true; 如设置为 false，那 cache 只能缓存到数据库或硬盘中。
2、cache.capacity
缓存的最大数量。默认是不限制，cache 不会移走任何缓存内容。负数被视不限制。
3、cache.algorithm
运算规则。为了使用规则，cache 的 size 必须是指定的。
如果 cache 的 size 不指定的话, 将不会限制缓存对象的大小。如果指定了 cache 的 size，但不指定 algorithm，那它会默认使用:com.opensymphony.oscache.base.algorithm.LRUCache
有下面三种规则：
com.opensymphony.oscache.base.algorithm.LRUCache:
last in first out(最后插入的最先调用)。默认选项。
com.opensymphony.oscache.base.algorithm.FIFOCache:
first int first out(最先插入的最先调用)。
com.opensymphony.oscache.base.algorithm.UnlimitedCache :
cache 中的内容将永远不会被丢弃。
如果 cache.capacity 不指定值的话，它将被设为默认选项。
4、cache.blocking
是否同步。true 或者 false。一般设为 true，避免读取脏数据。
5、cache.unlimited.disk
指定硬盘缓存是否要作限制。默认值为 false。false 的状况下，disk cache capacity 和 cache.capacity 的值相同。
6、cache.persistence.class
指定类是被持久化缓存的类。class 必须实现 PersistenceListener 接口。
作为硬盘持久，可以实现
com.opensymphony.oscache.plugins.diskpersistence.HashDiskPersistenceListener 接口。
它把 class 的 toString()输出的 hash 值作为文件的名称。如果你要想文件名易读些（自己设定），DiskPersistenceListener 的父类也能使用，但其可能有非法字符或者过长的名字。
注意：HashDiskPersistenceListener 和 DiskPersistenceListener 需要设定硬盘路径：cache.path

7、cache.path
指定硬盘缓存的路径。目录如果不存在将被建立。同时注意 oscache 应该要有权限写文件系统。
例:
cache.path=c:\\myapp\\cache
cache.path=/opt/myapp/cache

8、cache.persistence.overflow.only (NEW! Since 2.1)
指定是否只有在内存不足的情况下才使用硬盘缓存。
默认值 false。但推荐是 true 如果内存 cache 被允许的话。这个属性彻底的改变了 cache 的行为，使得 persisted cache 和 memory 是完全不同。
9、cache.event.listeners
class 名列表(用逗号隔开)。每个 class 必须实现以下接口中的一个 或者几个
CacheEntryEventListener：接收 cache add/update/flush and remove 事件
CacheMapAccessEventListener ：接收 cache 访问事件。这个可以让你跟踪 cache 怎么工作。
默认是不配置任何 class 的。当然你可以使用一下的 class：
*com.opensymphony.oscache.plugins.clustersupport.BroadcastingCacheEventListener ： 分布式的监听器。可以广播到局域网内的其他 cache 实例。
* com.opensymphony.oscache.extra.CacheEntryEventListenerImpl :一个简单的监听器。在 cache 的生命周期中记录所有 entry 的事件。
* com.opensymphony.oscache.extra.CacheMapAccessEventListenerImpl ： 记录 count of cache map events（cache hits，misses and state hits）.
10、cache.key
在 application 和 session 的作用域时 用于标识 cache 对象的,
用于 ServletCacheAdministrator;此属性不是指定为"__oscache_cache"格式时为默认值, 如果代码中需要用到默认值时可以通使用
com.opensymphony.oscache.base.Const.DEFAULT_CACHE_KEY 来取得;
11、cache.use.host.domain.in.key
当配置多个服务器时,想通过服备器名称自动生成 cache key 时,可将此属性设为 true. 默认值为 false;
12、Additional Properties
在以上基础选项之上可以加入一些额外的属性到此文件中.
例: JavaGroupsBroadcastingListener 便是额外的.
13、cache.cluster.multicast.ip
用于缓存集群. 默认为 231.12.21.132
14、cache.cluster.properties
指集群中的额外配置项. 以下是默认设置:(此属性的相关说将在集群文档中说明)
UDP(mcast_addr=231.12.21.132;mcast_port=45566;ip_ttl=32;\
mcast_send_buf_size=150000;mcast_recv_buf_size=80000):\
PING(timeout=2000;num_initial_members=3):\
MERGE2(min_interval=5000;max_interval=10000):\
FD_SOCK:VERIFY_SUSPECT(timeout=1500):\
pbcast.NAKACK(gc_lag=50;retransmit_timeout=300,600,1200,2400,4800;max_xmit_size=8192):\
UNICAST(timeout=300,600,1200,2400):\
pbcast.STABLE(desired_avg_gossip=20000):\
FRAG(frag_size=8096;down_thread=false;up_thread=false):\
pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;shun=false;print_local_addr=true)

三、OSCache 的基本用法（缓存 JSP 页面中部分）

（一）：Cache-OSCache 提供的缓存标签
这是 OSCache 提供的标签库中最重要的一个标签，包括在标签中的内容将应用缓存机制进行处理，处理的方式将取决于编程者对 cache 标签属性的设置。
第一次请求到达时，标签中的内容被处理并且缓存起来，当下一个请求到达时，缓存系统会检查这部分内容的缓存是否已经失效，主要是以下几项：
1. 缓存时间超过了 cache 标签设置的 time 或者 duration 属性规定的超时时间
2. cron 属性规定的时间比缓存信息的开始时间更晚
3. 标签中缓存的内容在缓存后又被重新刷新过
4. 其他缓存超期设定
如果符合上面四项中的任何一项，被缓存的内容视为已经失效，这时被缓存的内容将被重新处理并且返回处理过后的信息，如果被缓存的内容没有失效，那么返回给用户的将是缓存中的信息。
cache 标签的属性说明:
key － 标识缓存内容的关键词。在指定的作用范围内必须是唯一的。默认的 key 是被访问页面的 URI 和后面的请求字符串。
你可以在同一个页面中使用很多 cache 标签而不指定他的 key 属性，这种情况下系统使用该页面的 URI 和后面的请求字符串，另外再自动给这些 key 增加一个索引值来区分这些缓存内容。但是不推荐采用这样的方式。
scope － 缓存发生作用的范围，可以是 application 或者 session
time － 缓存内容的时间段，单位是秒，默认是 3600 秒，也就是一个小时，如果设定一个负值，那么这部分被缓存的内容将永远不过期。
duration － 指定缓存内容失效的时间，是相对 time 的另一个选择，可以使用简单日期格式或者符合 USO-8601 的日期格式。如：duration=”PT5M” duration=”5s”等
refresh － false 或者 true。
如果 refresh 属性设置为 true，不管其他的属性是否符合条件，这部分被缓存的内容都将被更新，这给编程者一种选择，决定什么时候必须刷新。
mode － 如果编程者不希望被缓存的内容增加到给用户的响应中，可以设置 mode 属性为 "silent"
其它可用的属性还包括：cron 、groups、language、refreshpolicyclass、refreshpolicyparam。
上面的这些属性可以单独使用，也可以根据需要组合使用，下面的例子将讲解这些常用属性的使用方式。
（二） Cache 标签实例分析:
1. 最简单的 cache 标签用法
使用默认的关键字来标识 cache 内容，超时时间是默认的 3600 秒

```
<cache:cache>
<% //自己的 JSP 代码内容 %>
</cache:cache>
```

2. 用自己指定的字符串标识缓存内容，并且设定作用范围为 session。

```
<cache:cache key="foobar" scope="session">
<% //自己的 JSP 代码内容 %>
</cache:cache>
```

3.动态设定 key 值，使用自己指定的 time 属性设定缓存内容的超时时间，使用动态 refresh 值决定是否强制内容刷新。
因为 OSCache 使用 key 值来标识缓存内容，使用相同的 key 值将会被认为使用相同的的缓存内容，所以使用动态的 key 值可以自由的根据不同的角色、不同的要求决定使用不同的缓存内容。
<cache:cache key="<%= product.getId() %>" time="1800" refresh="<%= needRefresh %>">
<% //自己的 JSP 代码内容 %>
</cache:cache>
4. 设置 time 属性为负数使缓存内容永不过期
<cache:cache time="-1">
<% //自己的 JSP 代码内容 %>
</cache:cache>
5. 使用 duration 属性设置超期时间
<cache:cache duration=”PT5M”>
<% //自己的 JSP 代码内容 %>
</cache:cache>
6. 使用 mode 属性使被缓存的内容不加入给客户的响应中
<cache:cache mode=’’silent”>
<% //自己的 JSP 代码内容 %>
</cache:cache>

（三）缓存过滤器 CacheFilter
用 CashFilter 实现页面级缓存
在 OSCache 组件中提供了一个 CacheFilter 用于实现页面级的缓存，主要用于对 web 应用中的某些动态页面进行缓存，尤其是那些需要生成 pdf 格式文件/报表、图片文件等的页面，不仅减少了数据库的交互、减少数据库服务器的压力，而且对于减少 web 服务器的性能消耗有很显著的效果。
这种功能的实现是通过在 web.xml 中进行配置来决定缓存哪一个或者一组页面，而且还可以设置缓存的相关属性，这种基于配置文件的实现方式对于 J2EE 来说应该是一种标准的实现方式了。
[注] 只有客户访问时返回 http 头信息中代码为 200（也就是访问已经成功）的页面信息才能够被缓存
1. 缓存单个文件
修改 web.xml，增加如下内容，确定对/testContent.jsp 页面进行缓存。
<filter> <filter-name>CacheFilter</filter-name>
<filter-class>com.opensymphony.oscache.web.filter.CacheFilter</filter-class> </filter>
<filter-mapping>
<filter-name>CacheFilter</filter-name>
<!-对/testContent.jsp 页面内容进行缓存－>
<url-pattern>/testContent.jsp</url-pattern>
</filter-mapping>
2. 缓存 URL pattern
修改 web.xml，增加如下内容，确定对*.jsp 页面进行缓存。
<filter> <filter-name>CacheFilter</filter-name>

<filter-class>com.opensymphony.oscache.web.filter.CacheFilter</filter-class> </filter>
<filter-mapping>
<filter-name>CacheFilter</filter-name>
<!-对所有 jsp 页面内容进行缓存 - >
<url-pattern>*.jsp</url-pattern>
</filter-mapping>
3. 自己设定缓存属性
在页面级缓存的情况下，可以通过设置 CacheFilter 的初始属性来决定缓存的一些特性：time 属性设置缓存的时间段，默认为 3600 秒，可以根据自己的需要只有的设置，而 scope 属性设置，默认为 application，可选项包括 application、session
<filter> <filter-name>CacheFilter</filter-name>
<filter-class>com.opensymphony.oscache.web.filter.CacheFilter</filter-class> <init-param>
<param-name>time</param-name>
<param-value>600</param-value>
</init-param>
<init-param>
<param-name>scope</param-name>
<param-value>session</param-value>
</init-param>
</filter>
<filter-mapping>
<filter-name>CacheFilter</filter-name>
<!-对所有 jsp 页面内容进行缓存-->
<url-pattern>*.jsp</url-pattern>
</filter-mapping>

四、demo

一、对象缓存
1、Cache 操作类

```
import java.util.Date;
import com.opensymphony.oscache.base.NeedsRefreshException;
import com.opensymphony.oscache.general.GeneralCacheAdministrator;
public class BaseCache extends GeneralCacheAdministrator {
private int refreshPeriod; //过期时间(单位为秒);
private String keyPrefix; //关键字前缀字符;
private static final long serialVersionUID = -4397192926052141162L;
public BaseCache(String keyPrefix,int refreshPeriod){
super();
this.keyPrefix = keyPrefix;
this.refreshPeriod = refreshPeriod;
}
//添加被缓存的对象;
public void put(String key,Object value){
```

```
this.putInCache(this.keyPrefix+&quot;_&quot;+key,value);
}
//删除被缓存的对象;
public void remove(String key){
this.flushEntry(this.keyPrefix+&quot;_&quot;+key);
}
//删除所有被缓存的对象;
public void removeAll(Date date){
this.flushAll(date);
}
public void removeAll(){
this.flushAll();
}
//获取被缓存的对象;
public Object get(String key) throws Exception{
try{
return this.getFromCache(this.keyPrefix+&quot;_&quot;+key,this.refreshPeriod);
} catch (NeedsRefreshException e) {
this.cancelUpdate(this.keyPrefix+&quot;_&quot;+key);
throw e;
}
}
}
```

2、Cache 管理类

```
public class CacheManager {
private BaseCache newsCache;
private static CacheManager instance;
private static Object lock = new Object();
private CacheManager() {
//这个根据配置文件来，初始 BaseCache 而已;
newsCache = new BaseCache("news",120);
}
public static CacheManager getInstance(){
if (instance == null){
synchronized( lock ){
if (instance == null){
instance = new CacheManager();
}
}
}
return instance;
}
public void putUser(User news) { newsCache.put(news.getId()+"",news); }
public void removeUser(String newsID) { newsCache.remove(newsID); }
```

```
public User getUser(int newsID) {
try {
return (User) newsCache.get(newsID+"");
} catch (Exception e) {
System.out.println("getNews>>newsID["+newsID+"]>>"+e.getMessage());
User news = new User(newsID);
this.putUser(news);
return news;
}
}
public void removeAllNews() {
newsCache.removeAll();
}
}
```

3、对象 Bean

```
public class User {
private int id;
private String name;
private String sex;
private int age;
private Date accessTime; public User(int id) {
super();
this.id = id;
this.accessTime = new Date(System.currentTimeMillis());
}
public String toString() {
return "User info is : id=" + id + " accessTime="
+ accessTime.toString();
}
public User(String name, String sex, int age) {
super();
this.name = name;
this.sex = sex;
this.age = age;
}
public User() {
}
public int getAge() {
return age;
}
public void setAge(int age) {
this.age = age;
}
public String getName() {
```

```
return name;
}
public void setName(String name) {
this.name = name;
}
public String getSex() {
return sex;
}
public void setSex(String sex) {
this.sex = sex;
}
public int getId() {
return id;
}
public void setId(int id) {
this.id = id;
}
public Date getAccessTime() {
return accessTime;
}
public void setAccessTime(Date accessTime) {
this.accessTime = accessTime;
}
}
```

4、测试类

```
public class TestObjectCache {
public static void main(String[] args) {
CacheManager cm=CacheManager.getInstance();
TestObjectCache test=new TestObjectCache();
test.print(cm);
}
public void print(CacheManager cm){
User user=null;
for (int i = 0; i < 1000; i++) {
user=cm.getUser(100);
System.out.println("<<"+i+">>: "+user);
if(i==10){
//删除缓存 id 的对象
cm.removeUser(100+"");
}
if(i==20){
//删除所有缓存的对象
cm.removeAllNews();
}
```

```
// 睡眠部分
try {
Thread.sleep(30000);
} catch (Exception e) {
}
}
}
}
```

五、小结及引申

缓存是在提升系统响应时常用的一种技术，在系统缓存上通常采用的是有页面缓存、处理缓存和数据缓存这三种具体的类别，应该说这三种缓存在实现上还是稍有不同，尽管底层的缓存实现是一样的。

页面缓存

页面缓存是指对页面中的内容片断进行缓存的方案。比如页面中有一个部分是显示栏目中的内容的，那么就可以缓存这个部分，在进行第二次请求的时候就直接从缓存中取出这部分的内容(其实就是这部分的 html 了)，这种情况下，缓存的作用其实非常明显，在典型的 action+service+dao 这样的结构中，在采用页面缓存后就意味着不需要经过 action、service、dao 这些层次的处理了，而是直接就返回了，对于系统响应速度的提升来说是非常明显的。

页面缓存通常采用 oscache 来进行实现，oscache 提供了一个 jsp tag，可通过这个 tag 来包含需要缓存的内容部分，当然，缓存的这个内容部分需要有对服务器的请求或逻辑计算等的，可想而知，去缓存一段静态 html 是没有意义的。

其次需要定义缓存的这段内容的 key，例如我们要去缓存页面中某个栏目的某页的内容，对于这段内容而言唯一的 key 就是栏目 ID 以及当前页数，这样就组成了这段缓存的 key 了，其实这个部分看起来好像是很简单，但有些时候会很麻烦，要仔细的想清楚这段内容的唯一的标识的 key 到底是什么，^_^，通常的做法其实可以从 action 中需要获取的参数或 service 接口的参数来决定….

页面缓存中还需要做的一个步骤就是通知缓存需要更新，页面缓存和其他缓存稍有不同，需要告诉它，这个时候不能再使用缓存中的内容了，需要从后台再重新获取来生成新的缓存内容，这个其实很简单，因为很难在后台发生变化的时候自己来更新缓存的内容，只能是去通知它，然后让它再次发起请求来生成新的内容放入缓存中。

页面的缓存的使用对于系统的响应速度确实会有很大的提升，在实现页面缓存时最麻烦的主要是缓存的 key 的定义以及缓存更新的通知，缓存 key 的定义这个自然框架是没法解决的，不过缓存更新的通知其实在框架中可以考虑一种通知模型的，^_^，就像事件通知那样……..在实际的项目中，可以自己去实现一个这样的通知模型或者就是简单的采用单例方式来标识某个 key 是否需要更新。

页面缓存在实际的项目中使用非常的多。

处理缓存

处理缓存是指对于 action、service、dao 或者系统层次中的某方法进行缓存，说直接点，就是对某个类的某个方法的结果做缓存，这样在下次进行完全相同的请求的时候就可以直接取缓存了，这种响应速度的提升也是非常明显的。

处理缓存在现在的情况下其实采用任务的缓存工具包都可以实现，如 oscache、ehcache、jbosscache 等，但目前还没有处理缓存框架的出现，这个和处理缓存是否应该存在的意义也

是有关系的，处理缓存框架要做到的其实就像拦截一样的方式，和 oscache　tag 类似。
同样，处理缓存的麻烦也在于怎么样去定义这个 key，很多情况下可以根据方法的输入作为 key，方法的输出作为 key 的值，但也会有其他一些复杂的情况，这个时候定义 key 就会变得复杂些了。
处理缓存同样有通知更新缓存的情况，和页面缓存基本是一样的。
应该说，处理缓存和页面缓存非常的相似，从实现上来说基本是完全一致的，在使用上来讲处理缓存使用的好像不多。
数据缓存
数据缓存估计大家都很熟悉，就是对系统的数据进行缓存的方式，典型的就是 Hibernate 的一级、二级数据缓存。
数据缓存在实现上如果是用 hibernate 的话更多的是直接使用 hibernate 的一级、二级以及查询缓存，如果自己要实现的话可以去参考 hibernate 的实现机制。
数据缓存的 key 在一级、二级缓存中采用的都是数据的标识键的值的方式，查询缓存采用的是查询参数、查询语句的方式。
数据缓存的更新则是 hibernate 在进行存储时直接更新缓存的内容，而对于查询缓存则是采用全部直接清除的方式，这样在下次进行查询时自然会重新去查询，^_^，大家可能会想，为什么页面缓存和处理缓存不采用这样的方式来实现缓存的更新，稍微想想就知道了，在后台发生改变的时候其实是不知道需要移除哪些 key 的，所以 hibernate 为了避免这个麻烦，采用的就是当数据一旦发生改变的时候就清除全部的查询缓存，而不是只去清除相关的缓存，其实这里可以采用一种订阅式的模型，呵呵，当然，也增加了框架的复杂度。
数据缓存使用的应该是最多的，效果也是很明显的。
以上三种缓存是目前缓存实现时通常碰到的三种状况，里面按使用的多少来排序应该是：数据缓存、页面缓存和处理缓存；实现的难度上从难到易的顺序应该是：处理缓存、页面缓存、数据缓存；对于系统响应速度提升的效果来说从最好到好的顺序应该是：页面缓存、处理缓存、数据缓存。

六、一个典型应用场景

在 SSH 项目应用中，可以以对象的形式来缓存展现给用户的数据信息。对象的缓存要充分利用分组带来的好处（可以分组删除被缓存的对象），这样在执行数据库的 CUD 操作时，可以调用删除相应组别的缓存对象。
示例代码：

```
private CacheManager cm;
    private final static String CACHE_KEY_SUB="RetJobs";
    public JobAction() {
      //获取缓存管理对象
       cm = CacheManager.getInstance();
    }
查询部分
        page=(Page<RetJob>)(cm.get(CACHE_KEY_SUB+"_"+currentPage));
        if(page==null){
            //--------------------需要缓存对象部分----------------------
            page = retJobBaseModel.getJobs(currentPage, pageSize, statusCondition);
            //-----------------------------------------
```

```
            //缓存对象（含所属分组信息）
            cm.put(page,                                CACHE_KEY_SUB+"_"+currentPage,new
String[]{CACHE_KEY_SUB});
        }
```

CUD 操作部分

```
setCacheDisabled(CACHE_KEY_SUB);
    private void setCacheDisabled(String group) {
        //通过组别信息来删除缓存的对象。
        cm.removeObjectByGroup(group);
    }
```

CacheManager 类

```
public class CacheManager {
    private BaseCache newsCache;
    private static CacheManager instance;
    private static Object lock = new Object();
    private CacheManager() {
        // 这个根据配置文件来，初始 BaseCache 而已;
        newsCache = new BaseCache("hrms", 300);
    }
    public static CacheManager getInstance() {
        if (instance == null) {
            synchronized (lock) {
                if (instance == null) {
                    instance = new CacheManager();
                }
            }
        }
        return instance;
    }
    public void put(Object news,String key,String[] groups) {
        newsCache.put(key, news,groups);
    }
    public void remove(String key) {
        newsCache.remove(key);
    }
    public Object get(String key) {
        try {
            return newsCache.get(key);
        } catch (Exception e) {
            return null;
        }
    }
    public void removeAll() {
```

```
        newsCache.removeAll();
    }
    public void removeObjectByGroup(String group){
        newsCache.removeObjectByGroup(group);
    }
}
```

BaseCache 类增加的 2 个方法如下：

```
    // 添加被缓存的对象;
    public void put(String key, Object value,String[] groups) {
        this.putInCache(this.keyPrefix + "_" + key, value,groups);
    }
    //删除该组的缓存对象
    public void removeObjectByGroup(String group){
        this.flushGroup(group);
    }
```