Grails主要使用缓存插件来实现缓存功能。cache插件使用基于内存的缓存机制而cache-ehcache插件则使用ehcache作为缓存机制。

# 基本缓存插件使用

## 安装插件

用create-app命令新建一个项目后会自动安装缓存插件，也可以在BuildConfig.groovy中使用如下代码安装缓存插件。

|  |
| --- |
| compile ':cache:1.0.1' |

## 使用插件

cache插件能够使spring bean的方法调用，controller的action调用和GSP页面上的内容和template能够被缓存起来。

可以在Config.groovy中配置多个缓存，也可以使用以CacheConfig.groovy结尾的文件来配置单个缓存。以CacheConfig.groovy结尾的文件必须放在grails-app/conf目录下，如果CacheConfig.groovy在某个package下面，则将该package放在grails-app/conf目录下。

在controller和service中使用annotation给方法调用加上缓存功能，而在GSP页面中则使用缓存标签来缓存部分页面内容或者template。

针对缓存的annotation主要有三个：Cacheable、CachePut和CacheEvict。使用@Cacheable来标注该方法调用需要先从缓存中查找是否已经有调用结果，如果有则直接从缓存中获取数据，如果没有则调用方法获取返回结果并加入到缓存中。使用CachePut表示该方法不管缓存中是否已经有返回结果，而每次都调用方法，再将结果放入缓存中。使用@CacheEvict来刷新缓存让方法重新获取结果。

cache插件使用内存缓存，使用到的cache和cache manager都是基于线程安全的java.util.concurrent.ConcurrentMap实现的。内存缓存在开发、测试阶段非常方便，但是如果在生产环境需要用到集群、磁盘存储、应用重启后缓存内容任然存在、可以详细的定义缓存对象的保留时间、最大缓存大小等功能，则需要使用扩展的缓存插件，如：cache-ehcache。

### 配置

#### Config.groovy和artefact文件

缓存的配置可以放在Config.groovy或者多个\*CacheConfig.groovy文件中。配置支持environments配置块以针对不同环境配置不同的缓存。

在Config.groovy中可以使用如下几个配置项：

|  |  |  |
| --- | --- | --- |
| **Property** | **Default** | **Description** |
| grails.cache.enabled | true | 是否启用缓存 |
| grails.cache.proxyTargetClass | false | From the Spring Javadoc: "By default, all proxies are created as JDK proxies. This may cause some problems if you are injecting objects as concrete classes rather than interfaces. To overcome this restriction you can set the proxy-target-class attribute to true which will result in class-based proxies being created." |
| grails.cache.aopOrder | Ordered.LOWEST\_PRECEDENCE | From the Spring docs: "Defines the order of the cache advice that is applied to beans annotated with @Cacheable or @CacheEvict. No specified ordering means that the AOP subsystem determines the order of the advice." |
| grails.cache.clearAtStartup | false | 在应用启动的时候是否清楚所有缓存 |
| grails.cache.keyGenerator | "customCacheKeyGenerator" | 是否使用自定义的键生成器。如果使用自定义的键生成器可以配置 grails.cache.keyGenerator="myCacheKeygen", 然后在resource.groovy中注册自己的键生成器的beanmyCacheKeygen(my.company.MyKeyGen). 自定义键生成器必须实现 KeyGenerator接口。 |

### Cache DSL

内存缓存的定义非常简单，所有没有多少内容可以配置。缓存中的内容也没有过期时间，只有应用重启或者调用了某个使用@CacheEvict的方法的时候才会更新。

在Config.groovy中可以如下定义缓存：

|  |
| --- |
| grails.cache.config = {  cache {  name 'message'  }  cache {  name 'maps'  }  } |

或者在\*CacheConfig.groovy中：

|  |
| --- |
| config = {  cache {  name 'message'  }  cache {  name 'maps'  }  } |

上面的两个例子都会创建两个缓存，一个名字叫message另一个名字叫maps。在配置中也可以加入其它配置项，如果这些配置项不被当前的内存缓存支持，则会被忽略掉：

|  |
| --- |
| grails.cache.config = {  cache {  name 'message'  eternal false  overflowToDisk true  maxElementsInMemory 10000  maxElementsOnDisk 10000000  }  cache {  name 'maps'  }  } |

#### Order

可以使用order配置项来定义缓存的配置的加载顺序。order的值越大则越晚加载。后加载的缓存配置会覆盖掉前面加载的相同配置项。

|  |
| --- |
| order = 2000  config = {  cache {  name 'message'  }  cache {  name 'maps'  }  } |

### Annotations

@Cacheable和@CacheEvic的使用基本是Spring所用的这两个注解的翻版，下面的@Cacheable、@CachePut和@CacheEvict都是基于spring的文档内容，用于说明这三个注解的基本用法。Grails对这三个注解加入了自己的参数项，将在下面详细说明。

#### @Cacheable

@Cacheable用于说明被注解的方法将被缓存，这就是说方法调用的返回结果将被放到缓存中，下一次方法调用的时候直接从缓存中获取结果，而不用在执行方法。@Cacheable需要一个name参数用于说明结果放在哪个缓存中:

|  |
| --- |
| @Cacheable("books")  public Book findBook(ISBN isbn) {...} |

上面的例子说明，方法findBook将和缓存books相关。

##### 缓存对象的key

缓存中的内容都是key-value对，通过key来获取缓存的内容。所以，对于缓存的方法都需要有一个key值来从缓存中获取内容。Grails已经提供了一个key生成器，key值得算法如下：

* 如果方法没有参数，则key为 0.
* 如果有一个参数，则key为该参数
* 如果有多个参数，则key就从所有参数的hash值计算而来.

##### 有条件的缓存

有时，方法并不是一直都需要缓存，而是需要根据传入的参数，在某个条件下才需要缓存。@Cacheable通过conditional能够支持这种有条件的缓存。conditional使用一个返回boolean值的SpEL语法的表达式，如果返回true则缓存否则就不缓存：

|  |
| --- |
| @Cacheable(value="book", condition="#name.length < 32")  public Book findBook(String name) |

上面的例子就说明只有name参数的长度小于32的时候，该方法才需要缓存。

 Cache SpEL available metadata

| **Name** | **Location** | **Description** | **Example** |
| --- | --- | --- | --- |
| methodName | root object | The name of the method being invoked | #root.methodName |
| method | root object | The method being invoked | #root.method.name |
| target | root object | The target object being invoked | #root.target |
| targetClass | root object | The class of the target being invoked | #root.targetClass |
| args | root object | The arguments (as array) used for invoking the target | #root.args[0] |
| caches | root object | Collection of caches against which the current method is executed | #root.caches[0].name |
| *argument name* | evaluation context | Name of any of the method argument. If for some reason the names are not available (ex: no debug information), the argument names are also available under the a<#arg>where *#arg* stands for the argument index (starting from 0). | iban  or  a0  (one can also use  p0  or p<#arg> notation as an alias). |

#### @CachePut

对于只用于更新缓存的方法，可以使用@CachePut注解。这表示该方法将一直被执行，并且方法的返回结构会被放到缓存中。@Cacheable和@CachePut不能同时使用。

#### @CacheEvict

@CacheEvict用于从缓存中除去缓存内容。例如：

|  |
| --- |
| @CacheEvict(value = "books", allEntries=true)  public void loadBooks(InputStream batch) |

上面的例子从缓存books中去掉所有缓存的对象。

#### service方法缓存

例如下面的service代码：

|  |
| --- |
| package com.yourcompany  import grails.plugin.cache.CacheEvict  import grails.plugin.cache.Cacheable  class MessageService {  @Cacheable('message')  Message getMessage(String title) {  println 'Fetching message'  Message.findByTitle(title)  }  @CachePut(value='message', key='#message.title')  void save(Message message) {  println "Saving message $message"  message.save()  }  @CacheEvict(value='message', key='#message.title')  void delete(Message message) {  println "Deleting message $message"  message.delete()  }  } |

可以看到getMessage的调用结果将被缓存到message缓存中，缓存的key就是方法的参数名title。如果有多个参数，则这些参数会组合起来作为key值。

save方法用来将返回结果放入缓存，参数key说明它只清除key值是message.titile的内容，并用这个key放入自己的返回结果。

delete方法用于清楚缓存中key值是#message.title的内容。

下面是Message domain对象的代码：

|  |
| --- |
| package com.yourcompany  class Message implements Serializable {  private static final long serialVersionUID = 1  String title  String body  String toString() {  "$title: $body"  }  } |

对于内存缓存，不需要对象实现Serializable接口，但是对于集群环境的分布式缓存则放入缓存的对象必须都实现Serializable接口。

上面的代码需要在Config.groovy或者@CacheConfig.groovy中配置一个名为message的缓存。

#### Controller Action缓存

除了对方法调用结果做缓存，也可以对http请求的response做缓存，只要对controller的action方法加上缓存注解就行了：

|  |
| --- |
| package com.yourcompany  import grails.plugin.cache.CacheEvict  import grails.plugin.cache.Cacheable  class TestController {  @Cacheable('message')  def lookup() {  // perform some expensive operations  println "called 'lookup'"  }  @CacheEvict(value='message', allEntries=true)  def evict() {  println "called 'evict'"  }  } |

上面的例子对lookup action做了缓存，所以当第一次请求这个action的时候会打印called lookup，但是后续的调用则不会再打印called lookup了。而evict action则会清楚message缓存中的所有内容。

### CacheManager

插件注册了一个grailsCacheManager bean，所以很容易通过依赖注入获取该对象。最常用的CacheManager的方法是getCache(String name)，该方法用于在代码中获取cache对象，通过cache.getNativeCache()方法可以获取底层的缓存实现。

## GSP Cache Tag

插件提供了用于GSP页面的tag，用于给页面内容加上缓存。

### block Tag

展现部分页面中的标记内容，并对展现结果做缓存。这样当下一次展现这块内容时，就不需要再执行一边标记内容了。例如：

|  |
| --- |
| <cache:block>  <!-- Any valid markup may be included here, including dynamic expressions, invoking other tags, etc.... -->  </cache:block> |

tag还可以有一个参数key，用于区别不同对象展现不同内容：

|  |
| --- |
| <cache:block key="${currentUser.id}">  <!-- Any valid markup may be included here, including dynamic expressions, invoking other tags, etc.... -->  </cache:block> |

### render Tag

展现一个GSP Template，这样当下一次该template内容展现的时候就不需要去执行template的标记内容了。例如：

|  |
| --- |
| <cache:render template="myTemplate" model="[user: currentUser]" key="${currentUser.id}"/> |

可用的属性有：

* template (required) - The name of the template to apply
* key (optional) - An optional key that may be provided to allow for different instances of the rendered template to be held in the cache, presumably with different content.
* contextPath (optional) - the context path to use (relative to the application context path). Defaults to "" or path to the plugin for a plugin view or template.
* bean (optional) - The bean to apply the template against
* model (optional) - The model to apply the template against as a java.util.Map
* collection (optional) - A collection of model objects to apply the template to
* var (optional) - The variable name of the bean to be referenced in the template
* plugin (optional) - The plugin to look for the template in

## Grails Cache Admin Service

插件提供了一个grailsCacheAdminService bean，可以用来用清楚缓存，例如：

|  |
| --- |
| class ReportingController {  def grailsCacheAdminService  def report() {  // clear the cache used by the blocks tag…  grailsCacheAdminService.clearBlocksCache()  // clear the cache used by the render tag…  grailsCacheAdminService.clearTemplatesCache()  …  }  } |

# EhCache插件使用

## 安装插件

在BuildConfig.groovy中的插件配置块中增加：

|  |
| --- |
| compile ":cache-ehcache:1.0.0" |

## 使用插件

除了缓存的实现方式是EhCache以外，这个插件的使用方式和基本缓存插件是完全相同的。同样都是对service方法和controller的action加上注解，同样在Config.groovy或者\*CacheConfig.groovy中配置缓存。

EhCache插件的cache DSL支持更多的缓存配置项，同样也支持分布式缓存的配置。

### Cache DSL

大多数的cache DSL都基本和常规的ehcache的配置文件ehcache.xml中的配置项相同。在DSL中可以使用数值对象和布尔对象给配置项赋值，而ehcache.xml中只能使用字符串。

#### 定义缓存

同基本缓存插件一样，ehcache的缓存也在cache块中定义：

|  |
| --- |
| cache {  name 'mycache'  eternal false  overflowToDisk true  maxElementsInMemory 10000  maxElementsOnDisk 10000000  } |

其中name属性是必须的，其他所有属性都同ehcache.xml中<cache>的属性。

#### 默认缓存和缓存的默认配置

EhCache能够支持自动创建缓存，同样grails的这个插件也同样支持自动创建缓存。自动创建缓存的意思是指，当用CacheManager.getCache(String name)方法获取一个缓存实例时（通过grailsCacheManager），如果这个缓存还没有在Config.groovy或者\*CacheConfig.groovy中定义，则EhCache会自动创建这个缓存。因为这个缓存没有经过定义，所以需要在Config.groovy或者\*CacheConfig.groovy中定义一个默认缓存用于自动创建：

|  |
| --- |
| defaultCache {  maxElementsInMemory 10000  eternal false  timeToIdleSeconds 120  timeToLiveSeconds 120  overflowToDisk true  maxElementsOnDisk 10000000  diskPersistent false  diskExpiryThreadIntervalSeconds 120  memoryStoreEvictionPolicy 'LRU'  } |

如果没有定义这个defaultCache，那么当使用cacheManager.getCache(name)自动创建一个cache时就会抛出异常。

grails还可以定义cache的默认配置，defaulte配置块是给所有定义的cache提供默认值的，不要和defaultCache搞混了：

|  |
| --- |
| defaults {  maxElementsInMemory 1000  eternal false  overflowToDisk false  maxElementsOnDisk 0  } |

需要注意的是对于hibernateQuery和hibernateTimestamps这两个特殊的缓存和所有自动创建的缓存，它们的默认配置都来自defaultCache而不是default。

#### 'provider'块

所有的缓存DSL都支持一个叫做provider的配置块：

|  |
| --- |
| provider {  updateCheck false  monitoring 'on'  dynamicConfig false  defaultTransactionTimeoutInSeconds 30  maxBytesLocalHeap 10  maxBytesLocalOffHeap 20  maxBytesLocalDisk 30  } |

对于EhCache来说，上面的配置相当于传统的<ehcache>标签的配置：

|  |
| --- |
| <ehcache defaultTransactionTimeoutInSeconds="30" dynamicConfig="false"  maxBytesLocalDisk="30" maxBytesLocalHeap="10"  maxBytesLocalOffHeap="20" monitoring="on" updateCheck="false"> |

#### Hibernate二级缓存

插件支持使用EhCache作为hibernate的二级缓存，并且提供缓存管理的功能。

hibernate使用两个缓存，一个用于query cache（org.hibernate.cache.StandardQueryCache），另一个用来跟踪最近表更新的时间戳（org.hibernate.cache.UpdateTimestampsCache）。插件的cache DSL支持使用两种方式来定义这两个缓存。调用hibernateQuery()方法产生如下的XML配置：

|  |
| --- |
| <cache name="org.hibernate.cache.StandardQueryCache"  maxElementsInMemory="50"  timeToLiveSeconds="120"  eternal="false"  overflowToDisk="true"  maxElementsOnDisk="0"  /> |

调用hibernateTimestamp()方法产生如下的XML配置：

|  |
| --- |
| <cache name="org.hibernate.cache.UpdateTimestampsCache"  maxElementsInMemory="5000"  eternal="true"  overflowToDisk="false"  maxElementsOnDisk="0"  /> |

#### Hibernate domain class二级缓存

使用domain方法来配置hibernate的domain二级缓存，注意使用domain class的全名作为缓存的名字：

|  |
| --- |
| domain {  name 'com.yourcompany.yourapp.Person'  eternal false  overflowToDisk true  maxElementsInMemory 10000  maxElementsOnDisk 10000000  } |

也可以使用如下的方式：

|  |
| --- |
| import com.yourcompany.yourapp.Person  ...  domain {  name Person  eternal false  overflowToDisk true  maxElementsInMemory 10000  maxElementsOnDisk 10000000  } |

对于domain的集合属性，需要使用domainCollection方法配置集合：

|  |
| --- |
| domain {  name 'com.foo.Author'  eternal false  overflowToDisk true  maxElementsInMemory 10000  maxElementsOnDisk 10000000  }  domainCollection {  name 'books'  domain 'com.foo.Author'  eternal true  overflowToDisk true  maxElementsInMemory 100  maxElementsOnDisk 10000  } |

或者使用inner closure：

|  |
| --- |
| domain {  name 'com.foo.Author'  eternal false  overflowToDisk true  maxElementsInMemory 10000  maxElementsOnDisk 10000000  collection {  name 'books'  eternal true  overflowToDisk true  maxElementsInMemory 100  maxElementsOnDisk 10000  }  } |

#### 磁盘存储

对于任何overflowing到磁盘的缓存，都需要知道具体的磁盘存储路径，这就需要配置diskStore:

|  |
| --- |
| diskStore {  path '/tmp/ehcache'  } |

也可以使用temp true，用于保存到java.io.temdir系统属性中：

|  |
| --- |
| diskStore {  temp true  } |

使用home true，保存到用户目录下：

|  |
| --- |
| diskStore {  home true  } |

使用current true，保存到user.dir系统属性下：

|  |
| --- |
| diskStore {  current true  } |

#### 配置重写（overriding value）

有多个地方可以定义缓存，Config.groovy，\*CacheConfig.groovy，插件的\*CacheConfig.groovy。缓存的定义有加载的顺序，后加载的会覆盖掉前面加载的配置项的值。

例如，某个插件有下面的缓存定义：

|  |
| --- |
| order = 500  config = {  cache {  name 'the\_cache'  eternal false  overflowToDisk true  maxElementsInMemory 2  maxElementsOnDisk 3  }  } |

而在应用的Config.groovy中有如下的定义：

|  |
| --- |
| grails.cache.config = {  cache {  name 'the\_cache'  maxElementsInMemory 10000  maxElementsOnDisk 10000000  }  cache {  name 'another\_cache'  …  }  defaults {  timeToLiveSeconds 1234  }  } |

因为插件中的配置的order是500，而order的默认值是1000，所以Config.groovy中的缓存定义会覆盖掉插件中的定义。所以最终的缓存定义等同如下：

|  |
| --- |
| grails.cache.config = {  cache {  name 'the\_cache'  eternal false  overflowToDisk true  maxElementsInMemory 10000  maxElementsOnDisk 10000000  }  cache {  name 'another\_cache'  …  }  defaults {  timeToLiveSeconds 1234  }  } |

#### 环境

可以使用env来说明该缓存针对什么环境：

|  |
| --- |
| domain {  name 'com.foo.Other'  env 'staging'  } |

或者：

|  |
| --- |
| domain {  name 'com.foo.Book'  env(['staging', 'production'])  } |

#### 分布式缓存

分布式缓存的配置基本和EhCache的XML文件相同，通过配置cacheManagerPeerProviderFactory, cacheManagerPeerListenerFactory, cacheManagerEventListenerFactory, bootstrapCacheLoaderFactory, cacheExceptionHandlerFactory, cacheEventListenerFactory, cacheLoaderFactory, 和cacheExtensionFactory，唯一区别就是XML配置的class属性在DSL中变成了className。

插件还提供一些省略词，例如：cacheManagerPeerProviderFactory的timeToLive属性可以接受如下值：

|  |  |
| --- | --- |
| **DSL name** | **value** |
| "host" | 0 |
| "subnet" | 1 |
| "site" | 32 |
| "region" | 64 |
| "continent" | 128 |
| "unrestricted" | 255 |

cacheManagerPeerProviderFactory的className属性可接受如下省略词：

|  |  |
| --- | --- |
| **DSL name** | **class name** |
| "rmi" | "net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory" |
| "jgroups" | "net.sf.ehcache.distribution.jgroups.JGroupsCacheManagerPeerProviderFactory" |
| "jms" | "net.sf.ehcache.distribution.jms.JMSCacheManagerPeerProviderFactory" |

cacheEventListenerFactory的className有：

|  |  |
| --- | --- |
| **DSL name** | **class name** |
| "rmi" | "net.sf.ehcache.distribution.RMICacheReplicatorFactory" |
| "jgroups" | "net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory" |
| "jms" | "net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory" |

bootstrapCacheLoaderFactory的className有：

|  |  |
| --- | --- |
| **DSL name** | **class name** |
| "rmi" | "net.sf.ehcache.distribution.RMIBootstrapCacheLoaderFactory" |
| "jgroups" | "net.sf.ehcache.distribution.jgroups.JGroupsBootstrapCacheLoaderFactory" |

注意分布式缓存一般只用于生产环境，所以一般都需要结合env一起使用：

|  |
| --- |
| defaults {  maxElementsInMemory 1000  eternal false  overflowToDisk false  maxElementsOnDisk 0  cacheEventListenerFactoryName 'cacheEventListenerFactory'  }  domain {  name 'com.foo.Book'  }  cacheManagerPeerProviderFactory {  env 'production'  factoryType 'rmi'  multicastGroupAddress '${ehcacheMulticastGroupAddress}'  multicastGroupPort '${ehcacheMulticastGroupPort}'  timeToLive 'subnet'  }  cacheManagerPeerListenerFactory {  env 'production'  }  cacheEventListenerFactory {  env 'production'  name 'cacheEventListenerFactory'  factoryType 'rmi'  replicateAsynchronously false  } |

上面的配置在生产环境中会产生如下的XML配置：

|  |
| --- |
| <ehcache ...>  <diskStore path="java.io.tmpdir" />  <defaultCache  maxElementsInMemory="10000"  eternal="false"  timeToIdleSeconds="120"  timeToLiveSeconds="120"  overflowToDisk="true"  maxElementsOnDisk="10000000"  diskPersistent="false"  diskExpiryThreadIntervalSeconds="120"  memoryStoreEvictionPolicy="LRU"  />  <cacheManagerPeerProviderFactory  class='net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory'  properties="multicastGroupAddress=${ehcacheMulticastGroupAddress},  multicastGroupPort=${ehcacheMulticastGroupPort},  timeToLive=1,peerDiscovery=automatic"  propertySeparator=','  />  <cacheManagerPeerListenerFactory  class='net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory' />  <cache name="com.foo.Book"  maxElementsInMemory="1000"  eternal="false"  overflowToDisk="false"  maxElementsOnDisk="0">  <cacheEventListenerFactory  class='net.sf.ehcache.distribution.RMICacheReplicatorFactory'  properties="replicateAsynchronously=false"  propertySeparator=','  />  </cache>  </ehcache> |

而在开发环境产生的XML文件如下：

|  |
| --- |
| <ehcache ...>  <diskStore path="java.io.tmpdir" />  <defaultCache  maxElementsInMemory="10000"  eternal="false"  timeToIdleSeconds="120"  timeToLiveSeconds="120"  overflowToDisk="true"  maxElementsOnDisk="10000000"  diskPersistent="false"  diskExpiryThreadIntervalSeconds="120"  memoryStoreEvictionPolicy="LRU"  />  <cache name="com.foo.Book"  maxElementsInMemory="1000"  eternal="false"  overflowToDisk="false"  maxElementsOnDisk="0"  />  </ehcache> |