Ehcache v1.2.3 User Guide

Greg Luck

2 September 2006

Contents

Chapter 1

Preface

This is a book about ehcache, a widely used open source Java cache. Ehcache has grown in size and scope since it was introduced in October 2003. As people used it they often noticed it was missing a feature they wanted. Over time, the features that were repeatedly asked for, and make sense for a Cache, have been added.

Ehcache is now used for Hibernate caching, data access object caching, security credential caching, web caching, application persistence and distributed caching. The biggest issue faced by Ehcache users at the time of writing is understanding when and how to use these features.

1.1 Audience

The intended audience for this book is developers who use ehcache. It should be able to be used to start from scratch, get up and running quickly, and also be useful for the more complex options.

Ehcache is about performance and load reduction of underlying resources. Another natural audience is performance specialists.

It is also intended for application and enterprise architects. Some of the features of ehcache, such as distributed caching and J2EE caching, are alternatives to be considered along with other ways of solving those problems. This book discusses the trade-offs in ehcache's approach to help make a decision about appropriateness of use.

1.2 Book Format

This is the first time that the ehcache documentation has been put in book form suitable for use as an online PDF or printed. It is designed to be printed from PDF, so blank pages have been deliberately left to give a good flow.

1.3 Acknowledgements

Ehcache has had many contributions in the form of forum discussions, feature requests, bug reports, patches and code commits.

Rather than try and list the many hundreds of people who have contributed to ehcache in some way it is better to link to the web site where contributions are acknowledged in the following ways:

- Bug reports and features requests appear in the changes report here:

- Patch contributors generally end up with an author tag in the source they contributed to
- Team members appear on the team list page here:

  Thanks to Denis Orlov for suggesting the need for a book in the first place.

1.4 About the ehcache name and logo

Adam Murdoch (an all round top Java coder) came up with the name in a moment of inspiration while we were stuck on the SourceForge project create page. Ehcache is a palindrome. He thought the name was wicked cool and we agreed.

The logo is similarly symmetrical, and is evocative of the diagram symbol for a doubly-linked list. That structure lies at the heart of ehcache.

Greg Luck Brisbane, Australia June, 2006

Chapter 2

Introduction

Ehcache is a cache library. Before getting into ehcache, it is worth stepping back and thinking about caching generally.

2.1 About Caches

Wiktionary defines a cache as *A store of things that will be required in future, and can be retrieved rapidly*. That is the nub of it.

In computer science terms, a cache is a collection of temporary data which either duplicates data located elsewhere or is the result of a computation. Once in the cache, the data can be repeatedly accessed inexpensively.

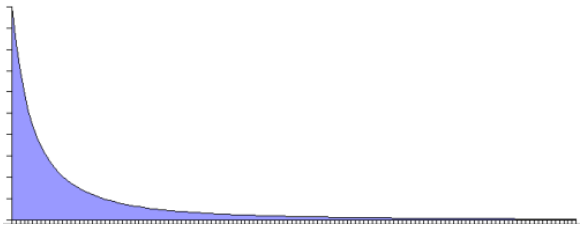
2.2 Why caching works

2.2.1 Locality of Reference

While ehcache concerns itself with Java objects, caching is used throughout computing, from CPU caches to the DNS system. Why? Because many computer systems exhibit *locality of reference*. Data that is near other data or has just been used is more likely to be used again.

2.2.2 The Long Tail

Chris Anderson, of Wired Magazine, coined the term *The Long Tail* to refer to Ecommerce systems. The idea that a small number of items may make up the bulk of sales, a small number of blogs might get the most hits and so on. While there is a small list of popular items, there is a long tail of less popular ones.



The Long Tail

The Long Tail is itself a vernacular term for a Power Law probability distribution. They don't just appear in ecommerce, but throughout nature. One form of a Power Law distribution is the Pareto distribution, commonly know as the 80:20 rule.

This phenomenon is useful for caching. If 20% of objects are used 80% of the time, and the a way can be found to reduce the cost of obtaining that 20% the system performance will improve.

2.3 Will an Application Benefit from Caching?

The short answer is that it often does, due to the effects noted above.

The medium answer is that it often depends on whether it is CPU bound or I/O bound. If an application is I/O bound then then the time taken to complete a computation depends principally on the rate at which data can be obtained. If it is CPU bound, then the time taken principally depends on the speed of the CPU and main memory.

While the focus for caching is on improving performance, it it also worth realizing that it reduces load. The time it takes something to complete is usually related to the expense of it. So, caching often reduces load on scarce resources.

2.3.1 Speeding up CPU bound Applications

CPU bound applications are often sped up by:

- improving algorithm performance
- parallelizing the computations across multiple CPUs (SMP) or multiple machines (Clusters).
- upgrading the CPU speed.

  The role of caching, if there is one, is to temporarily store computations that may be reused again.

  An example from ehcache would be large web pages that have a high rendering cost. Another caching of authentication status, where authentication requires cryptographic transforms.

2.3.2 Speeding up I/O bound Applications

Many applications are I/O bound, either by disk or network operations. In the case of databases they can be limited by both.

There is no Moore's law for hard disks. A 10,000 RPM disk was fast 10 years ago and is still fast. Hard disks are speeding up by using their own caching of blocks into memory.

Network operations can be bound by a number of factors:

- time to set up and tear down connections
- latency, or the minimum round trip time
- throughput limits
- marshalling and unmarhshalling overhead

  The caching of data can often help a lot with I/O bound applications. Some examples of ehcache uses are:

- Data Access Object caching for Hibernate
- Web page caching, for pages generated from databases.

2.3.3 Increased Application Scalability

The flip side of increased performance is increased scalability. Say you have a database which can do 100 expensive queries per second. After that it backs up and if connections are added to it it slowly dies.

In this case, caching may be able to reduce the workload required. If caching can cause 90 of that 100 to be cache hits and not even get to the database, then the database can scale 10 times higher than otherwise.

2.4 How much will an application speed up with Caching?

2.4.1 The short answer

The short answer is that it depends on a multitude of factors being:

- how many times a cached piece of data can and is reused by the application
- the proportion of the response time that is alleviated by caching

  In applications that are I/O bound, which is most business applications, most of the response time is getting data from a database. Therefore the speed up mostly depends on how much reuse a piece of data gets.

  In a system where each piece of data is used just once, it is zero. In a system where data is reused a lot, the speed up is large.

The long answer, unfortunately, is complicated and mathematical. It is considered next.

2.4.2 Applying Amdahl's Law

Amdahl's law, after Gene Amdahl, is used to find the system speed up from a speed up in part of the system.

```
1 / ((1 - Proportion Sped Up) + Proportion Sped Up / Speed up)
```

The following examples show how to apply Amdahl's law to common situations. In the interests of simplicity, we assume:

- a single server
- a system with a single thing in it, which when cached, gets 100% cache hits and lives forever.

Persistent Object Relational Caching

A Hibernate Session.load() for a single object is about 1000 times faster from cache than from a database.

A typical Hibernate query will return a list of IDs from the database, and then attempt to load each. If Session.iterate() is used Hibernate goes back to the database to load each object.

Imagine a scenario where we execute a query against the database which returns a hundred IDs and then load each one.

The query takes 20% of the time and the roundtrip loading takes the rest (80%). The database query itself is 75% of the time that the operation takes. The proportion being sped up is thus 60% (75% * 80%).

The expected system speedup is thus:

```
1 / ((1 - .6) + .6 / 1000)

= 1 / (.4 + .006)

= 2.5 times system speedup
```

Web Page Caching

An observed speed up from caching a web page is 1000 times. Ehcache can retrieve a page from its SimplePageCachingFilter in a few ms.

Because the web page is the end result of a computation, it has a proportion of 100%.

The expected system speedup is thus:

```
1 / ((1 - 1) + 1 / 1000)

= 1 / (0 + .001)

= 1000 times system speedup
```

Web Page Fragment Caching

Caching the entire page is a big win. Sometimes the liveness requirements vary in different parts of the page. Here the SimplePageFragmentCachingFilter can be used.

Let's say we have a 1000 fold improvement on a page fragment that taking 40% of the page render time.

The expected system speedup is thus:

```
1 / ((1 - .4) + .4 / 1000)

= 1 / (6 + .004)

= 1.6 times system speedup
```

2.4.3 Cache Efficiency

In real life cache entrie do not live forever. Some examples that come close are "static" web pages or fragments of same, like page footers, and in the database realm, reference data, such as the currencies in the world.

Factors which affect the efficiency of a cache are:

liveness how live the data needs to be. The less live the more it can be cached

proportion of data cached what proportion of the data can fit into the resource limits of the machine. For 32 bit Java systems, there was a hard limit of 2GB of address space. While now relaxed, garbage collection issues make it harder to go a lot large. Various eviction algorithms are used to evict excess entries.

Shape of the usage distribution If only 300 out of 3000 entries can be cached, but the Pareto distribution applies, it may be that 80% of the time, those 300 will be the ones requested. This drives up the average request lifespan.

Read/Write ratio The proportion of times data is read compared with how often it is written. Things such as the number of rooms left in a hotel will be written to quite a lot. However the details of a room

sold are immutable once created so have a maximum write of 1 with a potentially large number of reads.

Ehcache keeps these statistics for each Cache and each element, so they can be measured directly rather than estimated.

2.4.4 Cluster Efficiency

Also in real life, we generally do not find a single server?

Assume a round robin load balancer where each hit goes to the next server.

The cache has one entry which has a variable lifespan of requests, say caused by a time to live. The following table shows how that lifespan can affect hits and misses.

```
 Server 1    Server 2   Server 3    Server 4

  M             M          M           M
  H             H          H           H
  H             H          H           H
  H             H          H           H
  H             H          H           H
  ...           ...        ...         ...
```

The cache hit ratios for the system as a whole are as follows:

```
Entry
Lifespan  Hit Ratio   Hit Ratio  Hit Ratio   Hit Ratio
in Hits   1 Server    2 Servers  3 Servers   4 Servers

2          1/2           0/2         0/2         0/2
4          3/4           2/4         1/4         0/4
10         9/10          8/10        7/10        6/10
20         19/20         18/20       17/20       16/10
50         49/50         48/50       47/20       46/50
```

The efficiency of a cluster of standalone caches is generally:

```
(Lifespan in requests - Number of Standalone Caches) / Lifespan in requests
```

Where the lifespan is large relative to the number of standalone caches, cache efficiency is not much affected.

However when the lifespan is short, cache efficiency is dramatically affected.

(To solve this problem, ehcache supports distributed caching, where an entry put in a local cache is also propagated to other servers in the cluster.)

2.4.5 A cache version of Amdahl's law

From the above we now have:

```
 1 / ((1 - Proportion Sped Up * effective cache efficiency) + (Proportion Sped Up  * effective ca
```

effective cache efficiency = cache efficiency * cluster efficiency

2.4.6 Web Page example

Applying this to the earlier web page cache example where we have cache efficiency of 35% and average request lifespan of 10 requests and two servers:

```
cache efficiency = .35

cluster efficiency = .(10 - 1) / 10
                   = .9

effective cache efficiency = .35 * .9
                           = .315

      1 / ((1 - 1 * .315) + 1 * .315 / 1000)

      = 1 / (.685 + .000315)

      = 1.45 times system speedup
```

What if, instead the cache efficiency is 70%; a doubling of efficiency. We keep to two servers.

```
cache efficiency = .70

cluster efficiency = .(10 - 1) / 10
                   = .9

effective cache efficiency = .70 * .9
                           = .63

      1 / ((1 - 1 * .63) + 1 * .63 / 1000)

      = 1 / (.37 + .00063)

      = 2.69 times system speedup
```

What if, instead the cache efficiency is 90%; a doubling of efficiency. We keep to two servers.

```
cache efficiency = .90

cluster efficiency = .(10 - 1) / 10
                   = .9

effective cache efficiency = .9 * .9
                           = .81

      1 / ((1 - 1 * .81) + 1 * .81 / 1000)

      = 1 / (.19 + .00081)

      = 5.24 times system speedup
```

Why is the reduction so dramatic? Because Amdahl's law is most sensitive to the proportion of the system that is sped up.

Chapter 3

Getting Started

Ehcache can be used directly. It can also be used with the popular Hibernate Object/Relational tool. Finally, it can be used for J2EE Servlet Caching.

This quick guide gets you started on each of these. The rest of the documentation can be explored for a deeper understanding.

3.1 General Purpose Caching

- Make sure you are using a supported Java version.
- Place the ehcache jar into your classpath.
- Ensure that any libraries required to satisfy dependencies are also in the classpath.
- Configure ehcache.xml and place it in your classpath.
- Optionally, configure an appropriate logging level.

  See Code Samples for more information on direct interaction with ehcache.

3.2 Hibernate

- Perform the same steps as General Purpose Caching.
- Create caches in ehcache.xml.

  See Hibernate Caching for more information.

3.3 J2EE Servlet Caching

- Perform the same steps as General Purpose Caching.
- Configure a cache for your web page in ehcache.xml.
- To cache an entire web page, either use SimplePageCachingFilter or create your own subclass of CachingFilter
- To cache a jsp:Include or anything callable from a RequestDispatcher, either use SimplePageFragmentCachingFilter or create a subclass of PageFragmentCachingFilter.

- Configure the web.xml. Declare the filters created above and create filter mapping associating the filter with a URL.

  See J2EE Servlet Caching for more information.

3.4 Spring, Cocoon, Acegi and other frameworks

Usually, with these, you are using ehcache without even realising it. The first steps in getting more control over what is happening are:

- discover the cache names used by the framework
- create your own ehcache.xml with settings for the caches and place it in the application classpath.

Chapter 4

Features

- Fast and Light Weight
  - Fast
  - Simple
  - Small foot print
  - Minimal dependencies
- Scalable
  - Provides Memory and Disk stores for scalabilty into gigabytes
  - Scalable to hundreds of caches
  - Tuned for high concurrent load on large multi-cpu servers
  - Multiple CacheManagers per virtual machine
- Complete
  - Supports Object or Serializable caching
  - Support cache-wide or Element-based expiry policies
  - Provides LRU, LFU and FIFO cache eviction policies
  - Provides Memory and Disk stores
  - Distributed Caching
- Extensible
  - Listeners may be plugged in
  - Peer Discovery, Replicators and Listeners may be plugged in
- Application Persistence
  - Persistent disk store which stores data between VM restarts
  - Flush to disk on demand
- Supports Listeners
  - CacheManager listeners
  - Cache event listeners

- Distributed
  - Peer Discovery
  - Reliable Delivery
  - Synchronous Or Asynchronous Replication
  - Copy Or Invalidate Replication
  - Transparent Replication
  - Extensible
  - Bootstrapping from Peers
- J2EE and Applied Caching
  - Blocking Cache to avoid duplicate processing for concurrent operations
  - SelfPopulating Cache for pull through caching of expensive operations
  - J2EE Gzipping Servlet Filter
  - Cacheable Commands
  - Works with Hibernate
- High Quality
  - High Test Coverage
  - Automated Load, Limit and Performance System Tests
  - Production tested
  - Fully documented
  - Trusted by Popular Frameworks
  - Conservative Commit policy
  - Full public information on the history of every bug
  - Responsiveness to serious bugs
- Open Source Licensing
  - Apache 2.0 license

4.1 Fast and Light Weight

4.1.1 Fast

Over the years, various performance tests have shown ehcache to be one of the fastest Java caches. Ehcache's threading is designed for large, high concurrency systems.

Extensive performance tests in the test suite keep ehcache's performance consistent between releases.

As an example, some guys have created a java cache test tool called cache4j_perfomance_tester.

The results for ehcache-1.1 and ehcache-1.2 follow.

```
ehcache-1.1

[java] ------------------------------------------------------------------
[java] java.version=1.4.2_09
[java] java.vm.name=Java HotSpot(TM) Client VM
[java] java.vm.version=1.4.2-54
[java] java.vm.info=mixed mode
[java] java.vm.vendor="Apple Computer, Inc."
[java] os.name=Mac OS X
[java] os.version=10.4.5
[java] os.arch=ppc
[java] ------------------------------------------------------------------
[java] This test can take about 5-10 minutes. Please wait ...
[java] ------------------------------------------------------------------
[java] |GetPutRemoveT |GetPutRemove |Get |
[java] ------------------------------------------------------------------
[java] cache4j 0.4 |9240 |9116 |5556 |
[java] oscache 2.2 |33577 |30803 |8350 |
[java] ehcache 1.1 |7697 |6145 |3395 |
[java] jcs 1.2.7.0 |8966 |9455 |4072 |
[java] ------------------------------------------------------------------

ehcache-1.2
[java] ------------------------------------------------------------------
[java] java.version=1.4.2_09
[java] java.vm.name=Java HotSpot(TM) Client VM
[java] java.vm.version=1.4.2-54
[java] java.vm.info=mixed mode
[java] java.vm.vendor="Apple Computer, Inc."
[java] os.name=Mac OS X
[java] os.version=10.4.5
[java] os.arch=ppc
[java] ------------------------------------------------------------------
[java] This test can take about 5-10 minutes. Please wait ...
[java] ------------------------------------------------------------------
[java] |GetPutRemoveT |GetPutRemove |Get |
[java] ------------------------------------------------------------------
[java] cache4j 0.4 |9410 |9053 |5865 |
[java] oscache 2.2 |28076 |30833 |8031 |
[java] ehcache 1.2 |8753 |7072 |3479 |
[java] jcs 1.2.7.0 |8806 |9522 |4097 |
[java] ------------------------------------------------------------------
```

4.1.2 Simple

Many users of ehcache hardly know they are using it. Sensible defaults require no initial configuration.

The API is very simple and easy to use, making it possible to get up and running in minutes. See the Code Samples for details.

4.1.3 Small foot print

Ehcache 1.2 is 110KB making it convenient to package.

4.1.4 Minimal dependencies

Commons logging and collections are the only dependencies for most JDKs.

4.2 Scalable

4.2.1 Provides Memory and Disk stores for scalabilty into gigabytes

The largest ehcache installations use memory and disk stores in the gigabyte range. Ehcache is tuned for these large sizes.

4.2.2 Scalable to hundreds of caches

The largest ehcache installations use hundreds of caches.

4.2.3 Tuned for high concurrent load on large multi-cpu servers

There is a tension between thread safety and performance. Ehcache's threading started off coarse-grained, but has increasingly made use of ideas from Doug Lea to achieve greater performance. Over the years there have been a number of scalability bottlenecks identified and fixed.

4.2.4 Multiple CacheManagers per virtual machine

Ehcache 1.2 introduced multiple CacheManagers per virtual machine. This enables completely difference ehcache.xml configurations to be applied.

4.3 Complete

4.3.1 Supports Object or Serializable caching

As of ehcache-1.2 there is an API for Objects in addition to the one for Serializable. Non-serializable Objects can use all parts of ehcache except for DiskStore and replication. If an attempt is made to persist or replicate them they are discarded without error and with a DEBUG level log message.

The APIs are identical except for the return methods from Element. Two new methods on Element: getObjectValue and getKeyValue are the only API differences between the Serializable and Object APIs. This makes it very easy to start with caching Objects and then change your Objects to Seralizable to participate in the extra features when needed. Also a large number of Java classes are simply not Serializable.

4.3.2 Support cache-wide or Element-based expiry policies

Time to lives and time to idles are settable per cache. In addition, from ehcache-1.2.1, overrides to these can be set per Element.

4.3.3 Provides LRU, LFU and FIFO cache eviction policies

Ehcache 1.2 introduced Less Frequently Used and First In First Out caching eviction policies. These round out the eviction policies.

4.3.4 Provides Memory and Disk stores

Ehcache, like most of the cache solutions, provides high performance memory and disk stores.

4.3.5 Distributed

Flexible, extensible, high performance distributed caching. The default implementation supports cache discovery via multicast or manual configuration. Updates are delivered either asynchronously or synchronously via custom RMI connections. Additional discovery or delivery schemes can be plugged in by third parties.

See the Distributed Caching documentation for more feature details.

4.4 Extensible

4.4.1 Listeners may be plugged in

Ehcache 1.2 provides `CacheManagerEventListener` and `CacheEventListener` interfaces. Implementations can be plugged in and configured in ehcache.xml.

4.4.2 Peer Discovery, Replicators and Listeners may be plugged in

Distributed caching, introduced in ehcache 1.2 involves many choices and tradeoffs. The ehcache team believe that one size will not fit all. Implementers can use built-in mechanisms or write their own. A plugin development guide is included for this purpose.

4.5 Application Persistence

4.5.1 Persistent disk store which stores data between VM restarts

With ehcache 1.1 in 2004, ehcache was the first open source Java cache to introduce persistent storage of cache data on disk on shutdown. The cached data is then accessible the next time the application runs.

4.5.2 Flush to disk on demand

With ehcache 1.2, the flushing of entries to disk can be executed with a `cache.flush()` method whenever required, making it easier to use ehcache

4.6 Listeners

4.6.1 CacheManager listeners

Ehcache 1.2 introduced the `CacheManagerEventListener` interface with the following event methods:

- `notifyCacheAdded()`
- `notifyCacheRemoved()`

4.6.2 Cache event listeners

Ehcache 1.2 introduced the `CacheEventListener` interfaces, providing a lot of flexibility for post-processing of cache events. The methods are:

- `notifyElementRemoved`
- `notifyElementPut`
- `notifyElementUpdated`
- `notifyElementExpired`

4.7 Distributed Caching

Ehcache 1.2 introduced a full-featured, fine-grained distributed caching mechanism for clusters.

4.7.1 Peer Discovery

Peer discovery may be either manually configured or automatic, using multicast. Multicast is simple, and adds and removes peers automatically. Manual configuration gives fine control and is useful for situations where multicast is blocked.

4.7.2 Reliable Delivery

The built-in delivery mechanism uses RMI with custom sockets over TCP, not UDP.

4.7.3 Synchronous Or Asynchronous Replication

Replication can be set to synchronous Or asynchronous, per cache.

4.7.4 Copy Or Invalidate Replication

Replication can be set to copy or invalidate, per cache, as is appropriate.

4.7.5 Transparent Replication

No programming changes are required to make use of replication. Only configuration in ehcache.xml.

4.7.6 Extensible

Distributed caching, introduced in ehcache 1.2 involves many choices and tradeoffs. The ehcache team believe that one size will not fit all. Implementers can use built-in mechanisms or write their own. A plugin development guide is included for this purpose.

4.7.7 Bootstrapping from Peers

Distributed caches enter and leave the cluster at different times. Caches can be configured to bootstrap themselves from the cluster when they are first initialized.

An abstract factory, BootstrapCacheLoaderFactory has been defined along with an interface BootstrapCacheLoader along with an RMI based default implementation.

4.8 J2EE and Applied Caching

High quality implementations for common caching scenarios and patterns.

4.8.1 Blocking Cache to avoid duplicate processing for concurrent operations

A cache which blocks subsequent threads until the first read thread populates a cache entry.

4.8.2 SelfPopulating Cache for pull through caching of expensive operations

SelfPopulatingCache - a read-through cache. A cache that populates elements as they are requested without requiring the caller to know how the entries are populated. It also enables refreshes of cache entries without blocking reads on the same entries.

4.8.3 J2EE Gzipping Servlet Filter

- CachingFilter - an abstract, extensible caching filter.
- SimplePageCachingFilter

  A high performance J2EE servlet filter that caches pages based on the request URI and Query String. It also gzips the pages and delivers them to browsers either gzipped or ungzipped depending on the HTTP request headers. Use to cache entire Servlet pages, whether from JSP, velocity, or any other rendering technology.

  Tested with Orion and Tomcat.
- SimplePageFragmentCachingFilter

  A high performance J2EE filter that caches page fragments based on the request URI and Query String. Use with Servlet request dispatchers to cache parts of pages, whether from JSP, velocity, or any other rendering technology. Can be used from JSPs using jsp:include.

  Tested with Orion and Tomcat.
- Works with Servlet 2.3 and Servlet 2.4 specifications.

4.8.4 Cacheable Commands

This is the trusty old command pattern with a twist: asynchronous behaviour, fault tolerance and caching. Creates a command, caches it and then attempts to execute it.

4.8.5 Works with Hibernate

Tested with Hibernate2.1.8 and Hibernate3.1.3, which can utilise all of the new features except for Object API and multiple session factories each using a different ehcache CacheManager. A new `net.sf.ehcache.hibernate.EhCache` makes those additional features available to Hibernate-3.1.3. A version of the new provider should make it into the Hibernate3.2 release.

4.9 High Quality

4.9.1 High Test Coverage

The ehcache team believe that the first and most important quality measure is a well designed and comprehensive test suite.

Ehcache has a relatively high 86% test coverage of source code. This has edged higher over time. Clover enforces the test coverage. Most of the missing 14% is logging and exception paths.

4.9.2 Automated Load, Limit and Performance System Tests

The ehcache JUnit test suite contains some long-running system tests which place high load on different ehcache subsystems to the point of failure and then are back off to just below that point. The same is done with limits such as the amount of Elements that can fit in a given heap size. The same is also done with performance testing of each subsystem and the whole together. The same is also done with network tests for cache replication.

The tests serve a number of purposes:

- establishing well understood metrics and limits
- preventing regressions
- reproducing any reported issues in production
- Allowing the design principle of graceful degradation to be achieved. For example, the asynchronous cache replicator uses SoftReferences for queued messages, so that the messages will be reclaimed before before an OutOfMemoryError occurs, thus favouring stability over replication.

4.9.3 Specific Concurrency Testing

Ehcache also has concurrency testing, which typically uses 50 concurrent threads hammering a piece of code. The test suites are also run on multi-core or multi-cpu machines so that concurrency is real rather than simulated. Additionally, every concurrency related issue that has ever been anticipated or resulted in a bug report has a unit test which prevents the condition from recurring. There are no reported issues that have not been reproduced in a unit test.

Concurrency unit tests are somewhat difficult to write, and are often overlooked. The team considers these tests a major factor in ehcache's quality.

4.9.4 Production tested

Ehcache came about in the first place because of production issues with another open source cache.

Final release versions of ehcache have been production tested on a very busy e-commerce site, supporting thousands of concurrent users, gigabyte size caches on large multi-cpu machines. It has been the experience

of the team that most threading issues do not surface until this type of load has been applied. Once an issue has been identified and investigated a concurrency unit test can then be crafted.

4.9.5 Fully documented

A core belief held by the project team is that a project needs good documentation to be useful.

In ehcache, this is manifested by:

- comprehensive written documentation
- Complete, meaningful JavaDoc for every package, class and public and protected method. Checkstyle rules enforce this level of documentation.
- an up-to-date FAQ

4.9.6 Trusted by Popular Frameworks

Ehcache is used extensively. See the Who is Using? page, or browse Google.

4.9.7 Conservative Commit policy

Projects like Linux maintain their quality through a restricted change process, whereby changes are submitted as patches, then reviewed by the maintainer and included, or modified. Ehcache follows the same process.

4.9.8 Full public information on the history of every bug

Through the SourceForge project bug tracker, the full history of all bugs are shown, including current status. We take this for granted in an open source project, as this is typically a feature that all open source projects have, but this transparency makes it possible to gauge the quality and riskiness of a library, something not usually possible in commercial products.

4.9.9 Responsiveness to serious bugs

The ehcache team is serious about quality. If one user is having a problem, it probably means others are too, or will have. The ehcache team use ehcache themselves in production. Every effort will be made to provide fixes for serious production problems as soon as possible. These will be committed to trunk. From there an affected user can apply the fix to their own branch.

4.10 Open Source Licensing

4.10.1 Apache 2.0 license

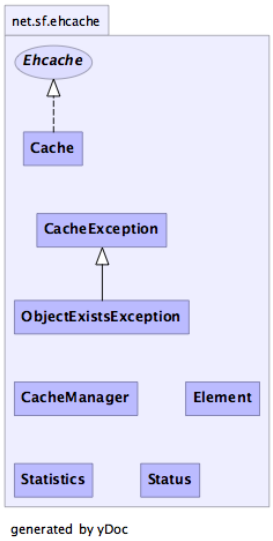
Ehcache's original Apache1.1 copyright and licensing was reviewed and approved by the Apache Software Foundation, making ehcache suitable for use in Apache projects. ehcache-1.2 is released under the updated Apache 2.0 license.

The Apache license is also friendly one, making it safe and easy to include ehcache in other open source projects or commercial products.

Chapter 5

Key Ehcache Concepts

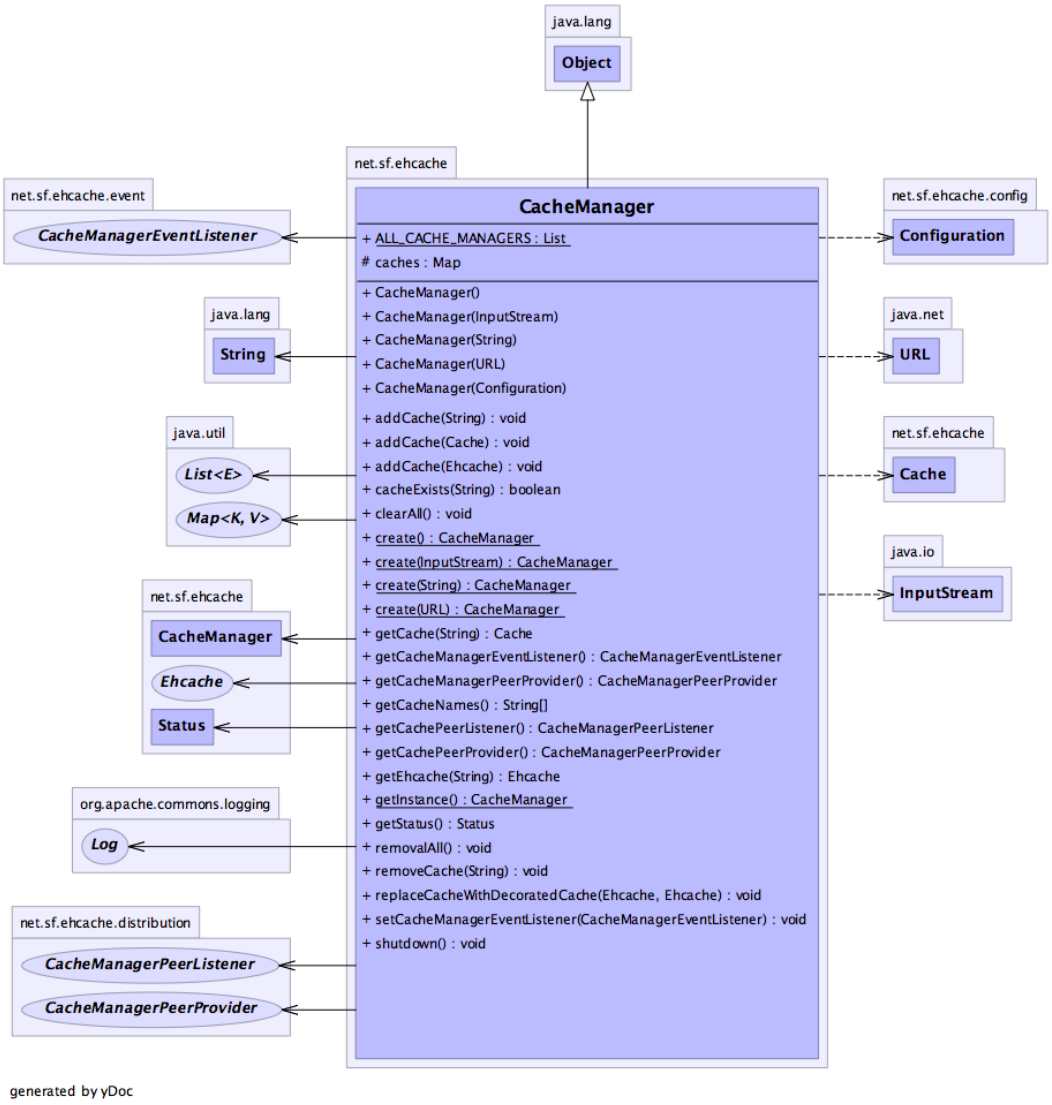
5.1 Key Ehcache Classes



Top Level Package Diagram

Ehcache consists of a `CacheManager`, which manages caches. Caches contain elements, which are essentially name value pairs. Caches are physically implemented either in-memory, or on disk.

5.1.1 CacheManager



CacheManager Class Diagram

The `CacheManager` comprises Caches which in turn comprise Elements.

Creation of, access to and removal of caches is controlled by the `CacheManager`.

CacheManager Creation Modes

`CacheManager` supports two creation modes: singleton and instance.

Singleton Mode Ehcache-1.1 supported only one `CacheManager` instance which was a singleton. CacheManager can still be used in this way using the static factory methods.

Instance Mode From ehcache-1.2, CacheManager has constructors which mirror the various static create methods. This enables multiple CacheManagers to be created and used concurrently. Each CacheManager

requires its own configuration.

If the Caches under management use only the MemoryStore, there are no special considerations. If Caches use the DiskStore, the diskStore path specified in each CacheManager configuration should be unique. When a new CacheManager is created, a check is made that there are no other CacheManagers using the same diskStore path. If there are, a CacheException is thrown. If a CacheManager is part of a cluster, there will also be listener ports which must be unique.

Mixed Singleton and Instance Mode If an application creates instances of CacheManager using a constructor, and also calls a static create method, there will exist a singleton instance of CacheManager which will be returned each time the create method is called together with any other instances created via constructor. The two types will coexist peacefully.

5.1.2 Ehcache

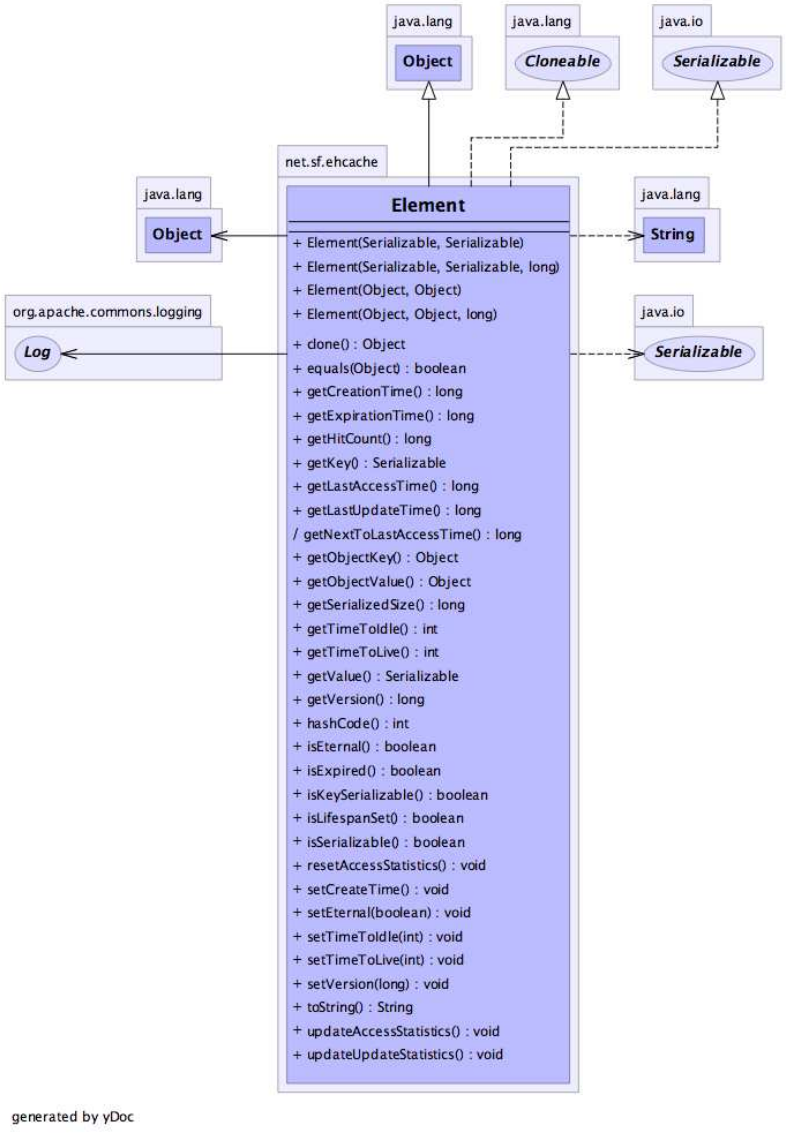
Ehcache Interface Diagram

All caches implement the `Ehcache` interface. A cache has a name and attributes. Each cache contains Elements.

A Cache in ehcache is analogous to a cache region in other caching systems.

Cache elements are stored in the `MemoryStore`. Optionally they also overflow to a `DiskStore`.

5.1.3 Element



Element Class Diagram

An element is an atomic entry in a cache. It has a key, a value and a record of accesses. Elements are put into and removed from caches. They can also expire and be removed by the Cache, depending on the Cache settings.

As of ehcache-1.2 there is an API for Objects in addition to the one for Serializable. Non-serializable Objects can use all parts of ehcache except for DiskStore and replication. If an attempt is made to persist or replicate them they are discarded without error and with a DEBUG level log message.

The APIs are identical except for the return methods from Element. Two new methods on Element: getObjectValue and getKeyValue are the only API differences between the Serializable and Object APIs. This makes it very easy to start with caching Objects and then change your Objects to Seralizable to participate in the extra features when needed. Also a large number of Java classes are simply not Serializable.

5.2 Cache Eviction Algorithms

A cache eviction algorithm is a way of deciding which `Element` to evict when the cache is full. In ehcache the `MemoryStore` has a fixed limited size and the `DiskStore` is unlimited. So, the only store that can be full is the `MemoryStore`. If a cache is set to only use a `MemoryStore` then the cache will also be full when the `MemoryStore` is full, otherwise it will overflow to the `DiskStore`.

The eviction algorithms in ehcache thus determine when the `MemoryStore` evicts an element. If there is no `DiskStore` this will also be a cache eviction, otherwise it will cause an overflow to disk.

5.2.1 About Eviction Algorithms

The idea here is, given a limit on the number of items to cache, how to choose the thing to evict that gives the *best* result.

In 1966 Laszlo Belady showed that the most efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This it a theoretical result that is unimplementable without domain knowledge. The Least Recently Used ("LRU") algorithm is often used as a proxy. It works pretty well because of the locality of reference phenonemon. As a result, LRU is the default eviction algorithm in ehcache, as it is in most caches.

Ehcache users may sometimes have a good domain knowledge. Accordingly, ehcache provides three eviction algorithms to choose from.

5.2.2 Ehcache's Eviction Algorithms

Ehcache supports three eviction algorithms: LRU, LFU and FIFO

Least Recently Used (LRU)

The eldest element, is the Least Recently Used (LRU). The last used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.

LRU is the default in ehcache.

Less Frequently Used (LFU)

For each get call on the element the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element with least number of hits, the Less Frequently Used element, is evicted.

If cache element use follows a pareto distribution, this algorithm may give better results than LRU.

First In First Out (FIFO)

Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

This algorithm is used if the use of an element makes it less likely to be used in the future. An example here would be an authentication cache.

5.3 Cache Usage Patterns

Caches can be used in different ways. Each of these ways follows a cache usage pattern. Ehcache supports the following:

- direct manipulation
- pull-through
- self-populating

5.3.1 Direct Manipulation

Here, to put something in the cache you do `cache.put(Element element)` and to get something from the cache you do `cache.get(Object key)`.

You are aware you are using a cache and you are doing so consciously.

5.3.2 Self Populating

Here, you just do gets to the cache using `cache.get(Object key)`. The cache itself knows how to populate an entry.

See the SelfPopulatingCache for more on this pattern.

Chapter 6

Code Samples

- Using the CacheManager
  - Singleton versus Instance
  - Ways of loading Cache Configuration
  - Adding and Removing Caches Programmatically
  - Shutdown the CacheManager
- Using Caches
  - Obtaining a reference to a Cache
  - CRUD operations
  - Disk Persistence on demand
  - Cache Sizes
  - Statistics of Cache Hits and Misses
- Programmatically Creating Caches
  - Creating a new cache from defaults
  - Creating a new cache with custom parameters
- Browse the JUnit Tests

6.1 Using the CacheManager

All usages of ehcache start with the creation of a CacheManager.

6.1.1 Singleton versus Instance

As of ehcache-1.2, ehcache CacheManagers can be created as either singletons (use the create factory method) or instances (use new).

Create a singleton CacheManager using defaults, then list caches.

```
CacheManager.create();
String[] cacheNames = CacheManager.getInstance().getCacheNames();
```

Create a CacheManager instance using defaults, then list caches.

```
CacheManager manager = new CacheManager();
String[] cacheNames = manager.getCacheNames();
```

Create two CacheManagers, each with a different configuration, and list the caches in each.

```
CacheManager manager1 = new CacheManager("src/config/ehcache1.xml");
CacheManager manager2 = new CacheManager("src/config/ehcache2.xml");
String[] cacheNamesForManager1 = manager1.getCacheNames();
String[] cacheNamesForManager2 = manager2.getCacheNames();
```

6.1.2 Ways of loading Cache Configuration

When the CacheManager is created it creates caches found in the configuration.

Create a CacheManager using defaults. Ehcache will look for ehcache.xml in the classpath.

```
CacheManager manager = new CacheManager();
```

Create a CacheManager specifying the path of a configuration file.

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

Create a CacheManager from a configuration resource in the classpath.

```
URL url = getClass().getResource("/anotherconfigurationname.xml");
CacheManager manager = new CacheManager(url);
```

Create a CacheManager from a configuration in an InputStream.

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

6.1.3 Adding and Removing Caches Programmatically

You are not just stuck with the caches that were placed in the configuration. You can create and remove them programmatically.

Add a cache using defaults, then use it. The following example creates a cache called *testCache*, which will be configured using defaultCache from the configuration.

```
CacheManager singletonManager = CacheManager.create();
singletonManager.addCache("testCache");
Cache test = singletonManager.getCache("testCache");
```

Create a Cache and add it to the CacheManager, then use it. Note that Caches are not usable until they have been added to a CacheManager.

```
CacheManager singletonManager = CacheManager.create();
Cache memoryOnlyCache = new Cache("testCache", 5000, false, false, 5, 2);
manager.addCache(memoryOnlyCache);
Cache test = singletonManager.getCache("testCache");
```

See Cache#Cache(...) for the full parameters for a new Cache:

Remove cache called sampleCache1

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

6.1.4 Shutdown the CacheManager

Ehcache should be shutdown after use. It does have a shutdown hook, but it is best practice to shut it down in your code.

Shutdown the singleton CacheManager

```
CacheManager.getInstance().shutdown();
```

Shutdown a CacheManager instance, assuming you have a reference to the CacheManager called *manager*

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

See the CacheManagerTest for more examples.

6.2 Using Caches

All of these examples refer to *manager*, which is a reference to a CacheManager, which has a cache in it called *sampleCache1*.

6.2.1 Obtaining a reference to a Cache

Obtain a Cache called "sampleCache1", which has been preconfigured in the configuration file

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

6.2.2 Performing CRUD operations

Put an element into a cache

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

Update an element in a cache. Even though `cache.put()` is used, ehcache knows there is an existing element, and considers the put an update for the purpose of notifying cache listeners.

```
Cache cache = manager.getCache("sampleCache1");
cache.put(new Element("key1", "value1");
//This updates the entry for "key1"
cache.put(new Element("key1", "value2");
```

Get a Serializable value from an element in a cache with a key of "key1".

```
Cache cache = manager.getCache("sampleCache1");
Element element = cache.get("key1");
Serializable value = element.getValue();
```

Get a NonSerializable value from an element in a cache with a key of "key1".

```
Cache cache = manager.getCache("sampleCache1");
Element element = cache.get("key1");
Object value = element.getObjectValue();
```

Remove an element from a cache with a key of "key1".

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

6.2.3 Disk Persistence on demand

sampleCache1 has a persistent diskStore. We wish to ensure that the data and index are written immediately.

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

6.2.4 Obtaining Cache Sizes

Get the number of elements currently in the `Cache`.

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

Get the number of elements currently in the `MemoryStore`.

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

Get the number of elements currently in the `DiskStore`.

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

6.2.5 Obtaining Statistics of Cache Hits and Misses

These methods are useful for tuning cache configurations.

Get the number of times requested items were found in the cache. i.e. cache hits

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

Get the number of times requested items were found in the `MemoryStore` of the cache.

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

Get the number of times requested items were found in the `DiskStore` of the cache.

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

Get the number of times requested items were not found in the cache. i.e. cache misses.

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

Get the number of times requested items were not found in the cache due to expiry of the elements.

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

These are just the most commonly used methods. See CacheTest for more examples. See Cache for the full API.

6.3 Creating a new cache from defaults

A new cache with a given name can be created from defaults very simply:

```
manager.addCache("cache name");
```

6.4 Creating a new cache with custom parameters

The configuration for a Cache can be specified programmatically in the Cache constructor:

```
public Cache(
    String name,
    int maxElementsInMemory,
    MemoryStoreEvictionPolicy memoryStoreEvictionPolicy,
    boolean overflowToDisk,
    boolean eternal,
    long timeToLiveSeconds,
    long timeToIdleSeconds,
    boolean diskPersistent,
    long diskExpiryThreadIntervalSeconds) {
    ...
   }
```

Here is an example which creates a cache called test.

```
   //Create a CacheManager using defaults
   CacheManager manager = CacheManager.create();

   //Create a Cache specifying its configuration.

   Cache testCache = new Cache("test", maxElements,
   MemoryStoreEvictionPolicy.LFU, true, false, 60, 30, false, 0);
   manager.addCache(cache);
```

Once the cache is created, add it to the list of caches managed by the CacheManager:

```
manager.addCache(testCache);
```

The cache is not usable until it has been added.

6.5 Browse the JUnit Tests

Ehcache comes with a comprehensive JUnit test suite, which not only tests the code, but shows you how to use ehcache.

A link to browsable unit test source code for the major ehcache classes is given per section. The unit tests are also in the src.zip in the ehcache tarball.

Chapter 7

Dependencies

7.1 Java Requirements

Ehcache supports 1.3, 1.4, 1.5 and 1.6 at runtime. Ehcache final releases are compiled with -target 1.3. This produces Java class data, version 47.0.

When compiling from source, the build process requires at least JDK 1.4, because 1.4 features are compile in but switched out at runtime if the JDK is 1.3. JDK1.3 is supported by catching NoSuchMethodError and providing an alternate implementation. No JDK1.4 or 1.5 language features are used.

Ehcache is known not to work with JDK1.1 and is not tested on JDK1.2.

Because of an RMI bug, in JDKs before JDK1.5 ehcache is limited to one CacheManager operating in distributed mode per virtual machine. (The bug limits the number of RMI registries to one per virtual machine). Because this is the expected deployment configuration, however, ther should be no practical effect.

On JDK1.5 and higher it is possible to have multiple CacheManagers per VM each participating in the same or different clusters. Indeed the replication tests do this with 5 CacheManagers on the same VM all run from JUnit.

7.2 Dependencies

For JDK1.4, JDK1.5 and JDK 1.6, ehcache requires commons-logging and commons-collections 2.1.1 from Apache's Jakarta project.

For JDK 1.3, ehcache also requires Apache xerces (xml-apis.jar and xercesImpl.jar), version 2.5.

These dependencies are very common, so they are probably already met in your project.

Chapter 8

Logging And Debugging

8.1 Commons Logging

Ehcache uses the Apache Commons Logging library for logging.

It acts as a thin bridge between logging statements in the code and logging infrastructure detected in the classpath. It will use in order of preference:

- log4j
- JDK1.4 logging
- and then its own `SimpleLog`

  This enables ehcache to use logging infrastructures compatible with Java versions from JDK1.2 to JDK5. It does create a dependency on Apache Commons Logging, however many projects, including Hibernate, share the same dependency.

For normal production use, use the `WARN` level in log4J and the `WARNING` level for JDK1.4 logging.

8.2 Logging Philosophy

Ehcache seeks to trade off informing production support developers or important messages and cluttering the log.

ERROR (JDK logging SEVERE_ messages should not occur in normal production and indicate that action should be taken.

WARNING (JDK logging WARN) messages generally indicate a configuration change should be made or an unusual event has occurred.

DEBUG (JDK logging FINE) messages are for development use. All DEBUG level statements are surrounded with a guard so that they are not executed unless the level is DEBUG.

Setting the logging level to DEBUG (JDK level FINE) should provide more information on the source of any problems. Many logging systems enable a logging level change to be made without restarting the application.

8.3 Remote Network debugging and monitoring for Distributed Caches

A simple new tool in ehcache-1.2, ehcache-1.x-remote-debugger.jar can be used to debug replicated cache operations. It is included in the distribution tarball for ehcache-1.2.3 and higher.

It is invoked using:

```
java -jar ehcache-1.x-remote-debugger.jar path_to_ehcache.xml cacheToMonitor
```

It will print a configuration of the cache, including replication settings and monitor the number of elements in the cache. If you are not seeing replication in your application, run up this tool to see what is going on.

It is a command line application, so it can easily be run from a terminal session.

Chapter 9

Class loading and Class Loaders

Class loading within the plethora of environments ehcache can be running is a somewhat vexed issue.
Since ehcache-1.2 all classloading is done in a standard way in one utility class: `ClassLoaderUtil`.

9.1 Plugin class loading

Ehcache allows plugins for events and distribution. These are loaded and created as follows:

```
/**
 * Creates a new class instance. Logs errors along the way. Classes are loaded using the
 * ehcache standard classloader.
 *
 * @param className a fully qualified class name
 * @return null if the instance cannot be loaded
 */
public static Object createNewInstance(String className) throws CacheException {
    Class clazz;
    Object newInstance;
    try {
        clazz = Class.forName(className, true, getStandardClassLoader());
    } catch (ClassNotFoundException e) {
        //try fallback
        try {
            clazz = Class.forName(className, true, getFallbackClassLoader());
        } catch (ClassNotFoundException ex) {
            throw new CacheException("Unable to load class " + className +
                    ". Initial cause was " + e.getMessage(), e);
        }
    }

    try {
        newInstance = clazz.newInstance();
    } catch (IllegalAccessException e) {
        throw new CacheException("Unable to load class " + className +
                ". Initial cause was " + e.getMessage(), e);
    } catch (InstantiationException e) {
        throw new CacheException("Unable to load class " + className +
                ". Initial cause was " + e.getMessage(), e);
    }
    return newInstance;
```

```
    }

    /**
     * Gets the <code>ClassLoader</code> that all classes in ehcache, and extensions, should
     * use for classloading. All ClassLoading in ehcache should use this one. This is the only
     * thing that seems to work for all of the class loading situations found in the wild.
     * @return the thread context class loader.
     */
    public static ClassLoader getStandardClassLoader() {
        return Thread.currentThread().getContextClassLoader();
    }

    /**
     * Gets a fallback <code>ClassLoader</code> that all classes in ehcache, and extensions,
     * should use for classloading. This is used if the context class loader does not work.
     * @return the <code>ClassLoaderUtil.class.getClassLoader();</code>
     */
    public static ClassLoader getFallbackClassLoader() {
        return ClassLoaderUtil.class.getClassLoader();
    }
```

If this does not work for some reason a CacheException is thrown with a detailed error message.

9.2 Loading of ehcache.xml resources

If the configuration is otherwise unspecified, ehcache looks for a configuration in the following order:

- Thread.currentThread().getContextClassLoader().getResource("/ehcache.xml")
- ConfigurationFactory.class.getResource("/ehcache.xml")
- ConfigurationFactory.class.getResource("/ehcache-failsafe.xml")

Ehcache uses the first configuration found.

Note the use of "/ehcache.xml" which requires that ehcache.xml be placed at the root of the classpath, i.e. not in any package.

Chapter 10

Performance Considerations

10.1 DiskStore

Ehcache comes with a `MemoryStore` and a `DiskStore`. The `MemoryStore` is approximately an order of magnitude faster than the `DiskStore`. The reason is that the `DiskStore` incurs the following extra overhead:

- Serialization of the key and value
- Eviction from the `MemoryStore` using an eviction algorithm
- Reading from disk

Note that writing to disk is not a synchronous performance overhead because it is handled by a separate thread.

A Cache should alway have its `maximumSize` attribute set to 1 or higher. A Cache with a maximum size of 1 has twice the performance of a disk only cache, i.e. one where the `maximumSize` is set to 0. For this reason a warning will be issued if a Cache is created with a 0 `maximumSize`.

10.2 Replication

The asynchronous replicator is the highest performance. There are two different effects:

- Because it is asynchronous the caller returns immediately
- The messages are placed in a queue. As the queue is processed, multiple messages are sent in one RMI call, dramatically accelerating replication performance.

Chapter 11

Cache Decorators

Ehcache 1.2 introduced the Ehcache interface, of which Cache is an implementation. It is possible and encouraged to create Ehcache decorators that are backed by a Cache instance, implement Ehcache and provide extra functionality.

The Decorator pattern is one of the the well known Gang of Four patterns.

11.1 Creating a Decorator

Cache decorators are created as follows:

```
BlockingCache newBlockingCache = new BlockingCache(cache);
```

The class must implement Ehcache.

11.2 Accessing the decorated cache

Having created a decorator it is generally useful to put it in a place where multiple threads may access it. This can be achieved in multiple ways.

11.2.1 Using CacheManager to access decorated caches

A built-in way is to replace the Cache in CacheManager with the decorated one. This is achieved as in the following example:

```
cacheManager.replaceCacheWithDecoratedCache(cache, newBlockingCache);
```

The `CacheManager` `replaceCacheWithDecoratedCache` method requires that the decorated cache be built from the underlying cache from the same name.

Note that any overwritten Ehcache methods will take on new behaviours without casting, as per the normal rules of Java. Casting is only required for new methods that the decorator introduces.

Any calls to get the cache out of the CacheManager now return the decorated one.

A word of caution. This method should be called in an appropriately synchronized init style method before multiple threads attempt to use it. All threads must be referencing the same decorated cache. An example of a suitable init method is found in `CachingFilter`:

```
    /**
     * The cache holding the web pages. Ensure that all threads for a given cache name are using
     */
    private BlockingCache blockingCache;

    /**
     * Initialises blockingCache to use
     *
     * @throws CacheException The most likely cause is that a cache has not been
     *                        configured in ehcache's configuration file ehcache.xml for the filt
     */
    public void doInit() throws CacheException {
        synchronized (this.getClass()) {
            if (blockingCache == null) {
                final String cacheName = getCacheName();
                Ehcache cache = getCacheManager().getEhcache(cacheName);
                if (!(cache instanceof BlockingCache)) {
                    //decorate and substitute
                    BlockingCache newBlockingCache = new BlockingCache(cache);
                    getCacheManager().replaceCacheWithDecoratedCache(cache, newBlockingCache);
                }
                blockingCache = (BlockingCache) getCacheManager().getEhcache(getCacheName());
            }
        }
    }
```

```
Ehcache blockingCache = singletonManager.getEhcache("sampleCache1");
```

The returned cache will exhibit the decorations.

11.3 Built-in Decorators

11.3.1 BlockingCache

A blocking decorator for an Ehcache, backed by a @link Ehcache.

It allows concurrent read access to elements already in the cache. If the element is null, other reads will block until an element with the same key is put into the cache.

This is useful for constructing read-through or self-populating caches.

BlockingCache is used by `CachingFilter`.


java.lang
Object
net.sf.ehcache
Ehcache
net.sf.ehcache.constructs.blocking
BlockingCache
cache : Ehcache
+ LOCK_NUMBER : int
# locks : Mutex[]
# timeoutMillis : int
+ BlockingCache(Ehcache)
+ bootstrap() : void
+ calculateInMemorySize() : long
+ clearStatistics() : void
+ clone() : Object
+ dispose() : void
+ evictExpiredElements() : void
+ flush() : void
+ get(Serializable) : Element
+ get(Object) : Element
+ getBootstrapCacheLoader() : BootstrapCacheLoader
# getCache() : Ehcache
+ getCacheEventNotificationService() : RegisteredEventListeners
+ getCacheManager() : CacheManager
+ getDiskExpiryThreadIntervalSeconds() : long
+ getDiskStoreHitCount() : int
+ getDiskStoreSize() : int
+ getGuid() : String
+ getHitCount() : int
+ getKeys() : List
+ getKeysNoDuplicateCheck() : List
+ getKeysWithExpiryCheck() : List
# getLockForKey(Object) : Mutex
+ getMaxElementsInMemory() : int
+ getMemoryStoreEvictionPolicy() : MemoryStoreEvictionPolicy
+ getMemoryStoreHitCount() : int
+ getMemoryStoreSize() : long
+ getMissCountExpired() : int
+ getMissCountNotFound() : int
+ getName() : String
+ getQuiet(Serializable) : Element
+ getQuiet(Object) : Element
+ getSize() : int
+ getStatistics() : Statistics
+ getStatisticsAccuracy() : int
+ getStatus() : Status
+ getTimeoutMillis() : int
+ getTimeToIdleSeconds() : long
+ getTimeToLiveSeconds() : long
+ initialise() : void
+ isDiskPersistent() : boolean
+ isElementInMemory(Serializable) : boolean
+ isElementInMemory(Object) : boolean
+ isElementOnDisk(Serializable) : boolean
+ isElementOnDisk(Object) : boolean
+ isEternal() : boolean
+ isExpired(Element) : boolean
+ isKeyInCache(Object) : boolean
+ isOverflowToDisk() : boolean
+ isValueInCache(Object) : boolean
+ liveness() : String
+ put(Element) : void
+ put(Element, boolean) : void
+ putQuiet(Element) : void
+ remove(Serializable) : boolean
+ remove(Serializable, boolean) : boolean
+ remove(Object) : boolean
+ remove(Object, boolean) : boolean
+ removeAll() : void
+ removeAll(boolean) : void
+ removeQuiet(Serializable) : boolean
+ removeQuiet(Object) : boolean
+ setBootstrapCacheLoader(BootstrapCacheLoader) : void
+ setCacheManager(CacheManager) : void
+ setDiskStorePath(String) : void
+ setName(String) : void
+ setStatisticsAccuracy(int) : void
+ setTimeoutMillis(int) : void
net.sf.ehcache
Ehcache
net.sf.ehcache.event
RegisteredEventListeners
org.apache.commons.logging
Log
java.lang
Object
String
net.sf.ehcache.constructs.concurrent
Mutex
java.util
List<E>
net.sf.ehcache
CacheManager
Element
Statistics
Status
net.sf.ehcache.store
MemoryStoreEvictionPolicy
java.io
Serializable
net.sf.ehcache.bootstrap
BootstrapCacheLoader

11.3.2 SelfPopulatingCache

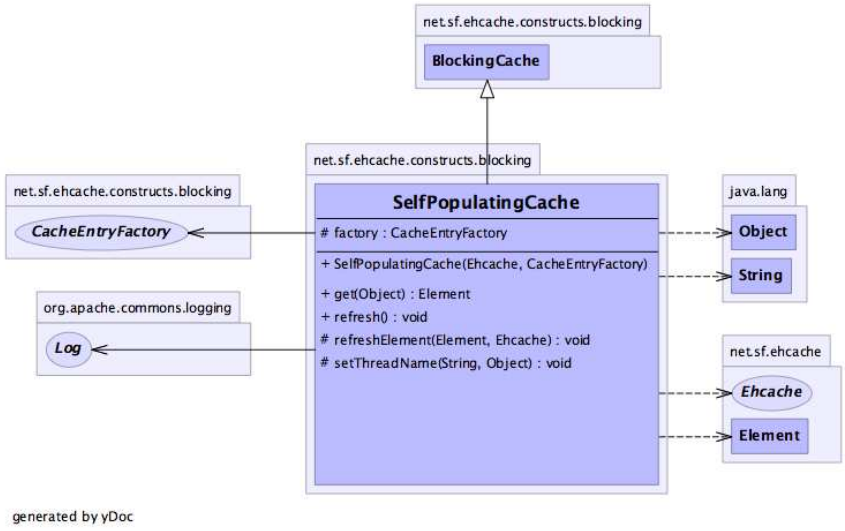
A selfpopulating decorator for @link Ehcache that creates entries on demand.

Clients of the cache simply call it without needing knowledge of whether the entry exists in the cache. If null the entry is created.

The cache is designed to be refreshed. Refreshes operate on the backing cache, and do not degrade performance of get calls.

SelfPopulatingCache extends BlockingCache. Multiple threads attempting to access a null element will block until the first thread completes. If refresh is being called the threads do not block - they return the stale data.

This is very useful for engineering highly scalable systems.



SelfPopulatingCache

Chapter 12

Cache Configuration

Caches can be configured in ehcache either declaratively, in xml, or by creating them programmatically and specifying their parameters in the constructor.

While both approaches are fully supported it is generally a good idea to separate the cache configuration from runtime use. There are also these benefits:

- It is easy if you have all of your configuration in one place. Caches consume memory, and disk space. They need to be carefully tuned. You can see the total effect in a configuration file. You could do this code, but it would not as visible.
- Cache configuration can be changed at deployment time.
- Configuration errors can be checked for at start-up, rather than causing a runtime error.

This chapter covers XML declarative configuration. See the Code samples for programmatic configuration.

Ehcache is redistributed by lots of projects. They may or may not provide a sample ehcache XML configuration file. If one is not provided, download ehcache from http://ehcache.sf.net. It, and the ehcache.xsd is provided in the distribution.

12.1 ehcache.xsd

Ehcache configuration files must be comply with the ehcache XML schema, ehcache.xsd, reproduced below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">
    <xs:element name="ehcache">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="diskStore"/>
                <xs:element minOccurs="0" maxOccurs="1"
                            ref="cacheManagerEventListenerFactory"/>
                <xs:element minOccurs="0" maxOccurs="1"
                            ref="cacheManagerPeerProviderFactory"/>
                <xs:element minOccurs="0" maxOccurs="1"
                            ref="cacheManagerPeerListenerFactory"/>
                <xs:element ref="defaultCache"/>
```

```
            <xs:element maxOccurs="unbounded" ref="cache"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="diskStore">
    <xs:complexType>
        <xs:attribute name="path" use="required" type="xs:NCName"/>
    </xs:complexType>
</xs:element>
<xs:element name="cacheManagerEventListenerFactory">
    <xs:complexType>
        <xs:attribute name="class" use="required"/>
        <xs:attribute name="properties" use="optional"/>
    </xs:complexType>
</xs:element>
<xs:element name="cacheManagerPeerProviderFactory">
    <xs:complexType>
        <xs:attribute name="class" use="required"/>
        <xs:attribute name="properties" use="optional"/>
    </xs:complexType>
</xs:element>
<xs:element name="cacheManagerPeerListenerFactory">
    <xs:complexType>
        <xs:attribute name="class" use="required"/>
        <xs:attribute name="properties" use="optional"/>
    </xs:complexType>
</xs:element>
<xs:element name="defaultCache">
    <xs:complexType>
        <xs:sequence>
            <xs:element minOccurs="0" ref="cacheEventListenerFactory"/>
            <xs:element minOccurs="0" ref="bootstrapCacheLoaderFactory"/>
        </xs:sequence>
        <xs:attribute name="diskExpiryThreadIntervalSeconds"
                      use="optional" type="xs:integer"/>
        <xs:attribute name="diskPersistent" use="optional" type="xs:boolean"/>
        <xs:attribute name="eternal" use="required" type="xs:boolean"/>
        <xs:attribute name="maxElementsInMemory" use="required"
                      type="xs:integer"/>
        <xs:attribute name="memoryStoreEvictionPolicy" use="optional"
                      type="xs:NCName"/>
        <xs:attribute name="overflowToDisk" use="required" type="xs:boolean"/>
        <xs:attribute name="timeToIdleSeconds" use="optional" type="xs:integer"/>
        <xs:attribute name="timeToLiveSeconds" use="optional" type="xs:integer"/>
    </xs:complexType>
</xs:element>
<xs:element name="cache">
    <xs:complexType>
        <xs:sequence>
            <xs:element minOccurs="0" ref="cacheEventListenerFactory"/>
            <xs:element minOccurs="0" ref="bootstrapCacheLoaderFactory"/>
        </xs:sequence>
        <xs:attribute name="diskExpiryThreadIntervalSeconds" type="xs:integer"/>
        <xs:attribute name="diskPersistent" type="xs:boolean"/>
        <xs:attribute name="eternal" use="required" type="xs:boolean"/>
        <xs:attribute name="maxElementsInMemory" use="required"
                      type="xs:integer"/>
        <xs:attribute name="memoryStoreEvictionPolicy" type="xs:NCName"/>
```

```
            <xs:attribute name="name" use="required" type="xs:NCName"/>
            <xs:attribute name="overflowToDisk" use="required" type="xs:boolean"/>
            <xs:attribute name="timeToIdleSeconds" type="xs:integer"/>
            <xs:attribute name="timeToLiveSeconds" type="xs:integer"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="cacheEventListenerFactory">
        <xs:complexType>
            <xs:attribute name="class" use="required"/>
            <xs:attribute name="properties" use="optional"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="bootstrapCacheLoaderFactory">
        <xs:complexType>
            <xs:attribute name="class" use="required"/>
            <xs:attribute name="properties" use="optional"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

12.2 ehcache-failsafe.xml

If the CacheManager default constructor or factory method is called, ehcache looks for a file called ehcache.xml in the top level of the classpath. Failing that it looks for ehcache-failsafe.xml in the classpath. ehcache-failsafe.xml is packaged in the ehcache jar and should always be found.

ehcache-failsafe.xml provides an extremely simple default configuration to enable users to get started before they create their own ehcache.xml.

If it used ehcache will emit a warning, reminding the user to set up a proper configuration.

The meaning of the elments and attributes are explained in the section on ehcache.xml. --- *ehcache diskStore path="java.io.tmpdir"/ defaultCache maxElementsInMemory="10000" eternal="false" timeToIdleSeconds="120" timeToLiveSeconds="120" overflowToDisk="true" diskPersistent="true" diskExpiryThreadIntervalSeconds="120" memoryStoreEvictionPolicy="LRU" / /ehcache* ---

12.3 ehcache.xml and other configuration files

If the CacheManager default constructor or factory method is called, ehcache looks for a file called ehcache.xml in the top level of the classpath.

The non-default creation methods allow a configuration file to be specified which can be called anything.

One XML configuration is required for each CacheManager that is created. It is an error to use the same configuration, because things like directory paths and listener ports will conflict. Ehcache will attempt to resolve conflicts and will emit a warning reminding the user to configure a separate configuration for multiple CacheManagers with conflicting settings.

The sample ehcache.xml, which is included in the ehcache distribution is shown below:

```
<ehcache>

    <!--
    Sets the path to the directory where cache files are created.

    If the path is a Java System Property it is replaced by its value in the
    running VM.
```

```
The following properties are translated:
* user.home - User's home directory
* user.dir - User's current working directory
* java.io.tmpdir - Default temp file path

Subdirectories can be specified below the property e.g. java.io.tmpdir/one
-->
<diskStore path="java.io.tmpdir"/>

<!--
Specifies a CacheManagerEventListenerFactory, be used to create a CacheManagerPeerProvider,
which is notified when Caches are added or removed from the CacheManager.

The attributes of CacheManagerEventListenerFactory are:
* class - a fully qualified factory class name
* properties - comma separated properties having meaning only to the factory.

Sets the fully qualified class name to be registered as the CacheManager event listener.

The events include:
* adding a Cache
* removing a Cache

Callbacks to listener methods are synchronous and unsynchronized. It is the responsibility
of the implementer to safely handle the potential performance and thread safety issues
depending on what their listener is doing.

If no class is specified, no listener is created. There is no default.
-->
<cacheManagerEventListenerFactory class="" properties=""/>


<!--
(Enable for distributed operation)

Specifies a CacheManagerPeerProviderFactory which will be used to create a
CacheManagerPeerProvider, which discovers other CacheManagers in the cluster.

The attributes of cacheManagerPeerProviderFactory are:
* class - a fully qualified factory class name
* properties - comma separated properties having meaning only to the factory.

Ehcache comes with a built-in RMI-based distribution system with two means of discovery of
CacheManager peers participating in the cluster:
* automatic, using a multicast group. This one automatically discovers peers and detects
  changes such as peers entering and leaving the group
* manual, using manual rmiURL configuration. A hardcoded list of peers is provided at
  configuration time.

Configuring Automatic Discovery:
Automatic discovery is configured as per the following example:
<cacheManagerPeerProviderFactory
                    class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
                    properties="peerDiscovery=automatic, multicastGroupAddress=230.0.0.1,
                                multicastGroupPort=4446"/>

Valid properties are:
```

```
* peerDiscovery (mandatory) - specify "automatic"
* multicastGroupAddress (mandatory) - specify a valid multicast group address
* multicastGroupPort (mandatory) - specify a dedicated port for the multicast heartbeat
  traffic

Configuring Manual Discovery:
Manual discovery is configured as per the following example:
<cacheManagerPeerProviderFactory class=
                      "net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
                      properties="peerDiscovery=manual,
                      rmiUrls=//server1:40000/sampleCache1|//server2:40000/sampleCache1
                      | //server1:40000/sampleCache2|//server2:40000/sampleCache2"/>

Valid properties are:
* peerDiscovery (mandatory) - specify "manual"
* rmiUrls (mandatory) - specify a pipe separated list of rmiUrls, in the form
                        //hostname:port

The hostname is the hostname of the remote CacheManager peer. The port is the listening
port of the RMICacheManagerPeerListener of the remote CacheManager peer.

An alternate CacheManagerPeerProviderFactory can be used for JNDI discovery of other
CacheManagers in the cluster. Only manual discovery is supported.

For cacheManagerPeerProviderFactory specify class
net.sf.ehcache.distribution.JNDIManualRMICacheManagerPeerProviderFactoryerFactory.

Correspondingly for cacheManagerPeerListenerFactory specify class
net.sf.ehcache.distribution.JNDIRMICacheManagerPeerListenerFactoryory.

Configuring JNDI Manual Discovery:
Manual JNDI discovery is configured as per the following example:
<cacheManagerPeerProviderFactory class=
    "net.sf.ehcache.distribution.JNDIManualRMICacheManagerPeerProviderFactoryerFactory"
    properties="peerDiscovery=manual, stashContexts=true, stashRemoteCachePeers=true,
          jndiUrls=t3//server1:40000/sampleCache1|t3//server2:40000/sampleCache1
          |t3//server1:40000/sampleCache2|t3//server2:40000/sampleCache2"/>

Valid properties are:
* peerDiscovery (mandatory) - specify "manual"
* stashContexts (optional) - specify "true" or "false".  Defaults to true.
  java.naming.Context objects are stashed for performance.
* stashRemoteCachePeers (optional) - specify "true" or "false".  Defaults to true.
  CachePeer objects are stashed for performance.
* jndiUrls (mandatory) - specify a pipe separated list of jndiUrls,
  in the form protocol//hostname:port
-->
<cacheManagerPeerProviderFactory
        class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
        properties="peerDiscovery=automatic,
                    multicastGroupAddress=230.0.0.1,
                    multicastGroupPort=4446"/>


<!--
(Enable for distributed operation)

Specifies a CacheManagerPeerListenerFactory which will be used to create a
```

```
CacheManagerPeerListener, which
listens for messages from cache replicators participating in the cluster.

The attributes of cacheManagerPeerListenerFactory are:
class - a fully qualified factory class name
properties - comma separated properties having meaning only to the factory.

Ehcache comes with a built-in RMI-based distribution system. The listener component is
RMICacheManagerPeerListener which is configured using
RMICacheManagerPeerListenerFactory. It is configured as per the following example:

<cacheManagerPeerListenerFactory
    class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"
    properties="hostName=fully_qualified_hostname_or_ip,
                port=40001,
                socketTimeoutMillis=120000"/>

All properties are optional. They are:
* hostName - the hostName of the host the listener is running on. Specify
  where the host is multihomed and you want to control the interface over which cluster
  messages are received. Defaults to the host name of the default interface if not
  specified.
* port - the port the listener listens on. This defaults to a free port if not specified.
* socketTimeoutMillis - the number of ms client sockets will stay open when sending
  messages to the listener. This should be long enough for the slowest message.
  If not specified it defaults 120000ms.


An alternate CacheManagerPeerListenerFactory can be also be used for JNDI binding of
listeners for messages from cache replicators participating in the cluster. For
cacheManagerPeerListenerFactory specify
class net.sf.ehcache.distribution.JNDIRMICacheManagerPeerListenerFactory.
Correspondingly for cacheManagerPeerProviderFactory specify class
net.sf.ehcache.distribution.JNDIManualRMICacheManagerPeerProviderFactoryerFactory.
Properties for JNDIRMICacheManagerPeerListenerFactory are the same as
RMICacheManagerPeerListenerFactory.

-->
<cacheManagerPeerListenerFactory
        class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"/>

<!-- Cache configuration.

The following attributes are required.

name:
Sets the name of the cache. This is used to identify the cache. It must be unique.

maxElementsInMemory:
Sets the maximum number of objects that will be created in memory

eternal:
Sets whether elements are eternal. If eternal,  timeouts are ignored and the
element is never expired.

overflowToDisk:
Sets whether elements can overflow to disk when the in-memory cache
has reached the maxInMemory limit.
```

```
The following attributes are optional.

timeToIdleSeconds:
Sets the time to idle for an element before it expires.
i.e. The maximum amount of time between accesses before an element expires
Is only used if the element is not eternal.
Optional attribute. A value of 0 means that an Element can idle for infinity.
The default value is 0.

timeToLiveSeconds:
Sets the time to live for an element before it expires.
i.e. The maximum time between creation time and when an element expires.
Is only used if the element is not eternal.
Optional attribute. A value of 0 means that and Element can live for infinity.
The default value is 0.

diskPersistent:
Whether the disk store persists between restarts of the Virtual Machine.
The default value is false.

diskExpiryThreadIntervalSeconds:
The number of seconds between runs of the disk expiry thread. The default value
is 120 seconds.

memoryStoreEvictionPolicy:
Policy would be enforced upon reaching the maxElementsInMemory limit. Default
policy is Least Recently Used (specified as LRU). Other policies available -
First In First Out (specified as FIFO) and Less Frequently Used
(specified as LFU)

Cache elements can also contain sub elements which take the same format of a factory class
and properties. Defined sub-elements are:

* cacheEventListenerFactory - Enables registration of listeners for cache events, such as
  put, remove, update, and expire.

* bootstrapCacheLoaderFactory - Specifies a BootstrapCacheLoader, which is called by a
  cache on initialisation to prepopulate itself.

Each cache that will be distributed needs to set a cache event listener which replicates
messages to the other CacheManager peers. For the built-in RMI implementation this is done
by adding a cacheEventListenerFactory element of type RMICacheReplicatorFactory to each
distributed cache's configuration as per the following example:

<cacheEventListenerFactory class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"
     properties="replicateAsynchronously=true,
     replicatePuts=true,
     replicateUpdates=true,
     replicateUpdatesViaCopy=true,
     replicateRemovals=true "/>

The RMICacheReplicatorFactory recognises the following properties:

* replicatePuts=true|false - whether new elements placed in a cache are
  replicated to others. Defaults to true.

* replicateUpdates=true|false - whether new elements which override an
```

```
  element already existing with the same key are replicated. Defaults to true.

* replicateRemovals=true - whether element removals are replicated. Defaults to true.

* replicateAsynchronously=true | false - whether replications are
  asynchronous (true) or synchronous (false). Defaults to true.

* replicateUpdatesViaCopy=true | false - whether the new elements are
  copied to other caches (true), or whether a remove message is sent. Defaults to true.


The RMIBootstrapCacheLoader bootstraps caches in clusters where RMICacheReplicators are
used. It is configured as per the following example:

<bootstrapCacheLoaderFactory
    class="net.sf.ehcache.distribution.RMIBootstrapCacheLoaderFactory"
    properties="bootstrapAsynchronously=true, maximumChunkSizeBytes=5000000"/>

The RMIBootstrapCacheLoaderFactory recognises the following optional properties:

* bootstrapAsynchronously=true|false - whether the bootstrap happens in the background
  after the cache has started. If false, bootstrapping must complete before the cache is
  made available. The default value is true.

* maximumChunkSizeBytes=<integer> - Caches can potentially be very large, larger than the
  memory limits of the VM. This property allows the bootstraper to fetched elements in
  chunks. The default chunk size is 5000000 (5MB).

-->


<!--
Mandatory Default Cache configuration. These settings will be applied to caches
created programmtically using CacheManager.add(String cacheName)
-->
<defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        overflowToDisk="true"
        diskPersistent="false"
        diskExpiryThreadIntervalSeconds="120"
        memoryStoreEvictionPolicy="LRU"
        />

<!--
Sample caches. Following are some example caches. Remove these before use.
-->

<!--
Sample cache named sampleCache1
This cache contains a maximum in memory of 10000 elements, and will expire
an element if it is idle for more than 5 minutes and lives for more than
10 minutes.

If there are more than 10000 elements it will overflow to the
disk cache, which in this configuration will go to wherever java.io.tmp is
```

```
defined on your system. On a standard Linux system this will be /tmp"
-->
<cache name="sampleCache1"
       maxElementsInMemory="10000"
       eternal="false"
       overflowToDisk="true"
       timeToIdleSeconds="300"
       timeToLiveSeconds="600"
       memoryStoreEvictionPolicy="LFU"
        />


<!--
Sample cache named sampleCache2
This cache has a maximum of 1000 elements in memory. There is no overflow to disk, so 1000
is also the maximum cache size. Note that when a cache is eternal, timeToLive and
timeToIdle are not used and do not need to be specified.
-->
<cache name="sampleCache2"
       maxElementsInMemory="1000"
       eternal="true"
       overflowToDisk="false"
       memoryStoreEvictionPolicy="FIFO"
        />


<!--
Sample cache named sampleCache3. This cache overflows to disk. The disk store is
persistent between cache and VM restarts. The disk expiry thread interval is set to 10
minutes, overriding the default of 2 minutes.
-->
<cache name="sampleCache3"
       maxElementsInMemory="500"
       eternal="false"
       overflowToDisk="true"
       timeToIdleSeconds="300"
       timeToLiveSeconds="600"
       diskPersistent="true"
       diskExpiryThreadIntervalSeconds="1"
       memoryStoreEvictionPolicy="LFU"
        />


<!--
Sample distributed cache named sampleDistributedCache1.
This cache replicates using defaults.
It also bootstraps from the cluster, using default properties.
-->
<cache name="sampleDistributedCache1"
       maxElementsInMemory="10"
       eternal="false"
       timeToIdleSeconds="100"
       timeToLiveSeconds="100"
       overflowToDisk="false">
    <bootstrapCacheLoaderFactory
            class="net.sf.ehcache.distribution.RMIBootstrapCacheLoaderFactory"/>
    <cacheEventListenerFactory
            class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>
```

```
    </cache>


    <!--
    Sample distributed cache named sampleDistributedCache2.
    This cache replicates using specific properties.
    It only replicates updates and does so synchronously via copy
    -->
    <cache name="sampleDistributedCache2"
           maxElementsInMemory="10"
           eternal="false"
           timeToIdleSeconds="100"
           timeToLiveSeconds="100"
           overflowToDisk="false">
        <cacheEventListenerFactory
                class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"
                properties="replicateAsynchronously=false, replicatePuts=false,
                            replicateUpdates=true, replicateUpdatesViaCopy=true,
                            replicateRemovals=false"/>
    </cache>

</ehcache>
```

Chapter 13

Storage Options

Ehcache has two stores:

- a MemoryStore and
- a DiskStore

13.1 Memory Store

The `MemoryStore` is always enabled. It is not directly manipulated, but is a component of every cache.

- Suitable Element Types

  All Elements are suitable for placement in the MemoryStore.

  It has the following characteristics:

  – Safety

    Thread safe for use by multiple concurrent threads.

    Tested for memory leaks. See MemoryCacheTest#testMemoryLeak. This test passes for ehcache but exploits a number of memory leaks in JCS. JCS will give an OutOfMemory error with a default 64M in 10 seconds.

  – Backed By JDK

    LinkedHashMap The `MemoryStore` for JDK1.4 and JDK 5 it is backed by an extended LinkedHashMap. This provides a combined linked list and a hash map, and is ideally suited for caching. Using this standard Java class simplifies the implementation of the memory cache. It directly supports obtaining the least recently used element.

    For JDK1.2 and JDK1.3, the LRUMap from Apache Commons is used. It provides similar features to LinkedHashMap.

    The implementation is determined dynamically at runtime. LinkedHashMap is preferred if found in the classpath.

  – Fast

    The memory store, being all in memory, is the fastest caching option.

13.1.1 Memory Use, Spooling and Expiry Strategy

All caches specify their maximum in-memory size, in terms of the number of elements, at configuration time.

When an element is added to a cache and it goes beyond its maximum memory size, an existing element is either deleted, if overflowToDisk is false, or evaluated for spooling to disk, if overflowToDisk is true. In the latter case, a check for expiry is carried out. If it is expired it is deleted; if not it is spooled. The eviction of an item from the memory store is based on the MemoryStoreEvictionPolicy setting specified in the configuration file.

memoryStoreEvictionPolicy is an optional attribute in ehcache.xml introduced since 1.2. Legal values are LRU (default), LFU and FIFO.

LRU, LFU and FIFO eviction policies are supported. LRU is the default, consistent with all earlier releases of ehcache.

- Least Recently Used (LRU) - Default

  The eldest element, is the Least Recently Used (LRU). The last used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.

- Less Frequently Used (LFU)

  For each get call on the element the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element with least number of hits, the Less Frequently Used element, is evicted.

- First In First Out (FIFO)

  Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

  For all the eviction policies there are also `putQuiet` and `getQuiet` methods which do not update the last used timestamp.

  When there is a `get` or a `getQuiet` on an element, it is checked for expiry. If expired, it is removed and null is returned.

  Note that at any point in time there will usually be some expired elements in the cache. Memory sizing of an application must always take into account the maximum size of each cache. There is a convenience method which can provide an estimate of the size in bytes of the `MemoryStore`. See calculateInMemorySize(). It returns the serialized size of the cache. Do not use this method in production. It is very slow. It is only meant to provide a rough estimate.

  The alternative would have been to have an expiry thread. This is a trade-off between lower memory use and short locking periods and cpu utilisation. The design is in favour of the latter. For those concerned with memory use, simply reduce the `maxElementsInMemory`.

13.2 DiskStore

The `DiskStore` provides a disk spooling facility.

- Suitable Element Types

  Only `Elements` which are `Serializable` can be placed in the DiskStore. Any non serializable `Elements` which attempt to overflow to the `DiskStore` will be removed instead, and a WARNING level log message emitted.

  It has the following characteristics:

- Storage Files

  The disk store creates one file per cache called "cache name.data".

  If the `DiskStore` is configured to be persistent, a "**cache name**.index" file is also created.

Files are created in the directory specified by the diskStore configuration element. The default configuration is "java.io.tmpdir", which causes files to be created in the system's temporary directory.

Following is a list of Java system properties which are supported as values for diskStore:

- user.home - User's home directory
- user.dir - User's current working directory
- java.io.tmpdir - Default temp file path

Apart from these, any directory can be specified using syntax appropriate to the operating system. e.g. for Unix "/home/application/cache".

- Expiry Strategy

  One thread per cache is used to remove expired elements. The optional attribute `diskExpiryThreadIntervalSeconds` sets the interval between runs of the expiry thread. Warning: setting this to a low value is not recommended. It can cause excessive `DiskStore` locking and high cpu utilisation. The default value is 120 seconds.

- Serializable Objects

  Only Serializable objects can be stored in a `DiskStore`. A NotSerializableException will be thrown if the object is not serializable.

- Safety

  `DiskStore`s are thread safe.

- Persistence

  `DiskStore` persistence is controlled by the diskPersistent configuration element. If false or omitted, `DiskStore`s will not persist between `CacheManager` restarts. The data file for each cache will be deleted, if it exists, both on shutdown and startup. No data from a previous instance `CacheManager` is available.

  If diskPersistent is true, the data file, and an index file, are saved. Cache Elements are available to a new `CacheManager`. This `CacheManager` may be in the same VM instance, or a new one.

  The data file is updated continuously during operation of the Disk Store. New elements are spooled to disk, and deleted when expired. The index file is only written when dispose is called on the `DiskStore`. This happens when the CacheManager is shut down, a Cache is disposed, or the VM is being shut down. It is recommended that the CacheManager shutdown() method be used. See Virtual Machine Shutdown Considerations for guidance on how to safely shut the Virtual Machine down.

  When a `DiskStore` is persisted, the following steps take place:

  - Any non-expired Elements of the `MemoryStore` are flushed to the DiskStore
  - Elements awaiting spooling are spooled to the data file
  - The free list and element list are serialized to the index file

  On startup the following steps take place:

  - An attempt is made to read the index file. If it does not exist or cannot be read successfully, due to disk corruption, upgrade of ehcache, change in JDK version etc, then the data file is deleted and the `DiskStore` starts with no Elements in it.
  - If the index file is read successfully, the free list and element list are loaded into memory. Once this is done, the index file contents are removed. This way, if there is a dirty shutdown, when restarted, ehcache will delete the dirt index and data files.
  - The `DiskStore` starts. All data is available.

- The expiry thread starts. It will delete Elements which have expired.

These actions favour safety over persistence. Ehcache is a cache, not a database. If a file gets dirty, all data is deleted. Once started there is further checking for corruption. When a get is done, if the Element cannot be successfully derserialized, it is deleted, and null is returned. These measures prevent corrupt and inconsistent data being returned.

- Fragmentation

  Expiring an element frees its space on the file. This space is available for reuse by new elements. The element is also removed from the in-memory index of elements.

- Speed

  Spool requests are placed in-memory and then asynchronously written to disk. There is one thread per cache. An in-memory index of elements on disk is maintained to quickly resolve whether a key exists on disk, and if so to seek it and read it.

- Serialization

  Writes to and from the disk use ObjectInputStream and the Java serialization mechanism. This is not required for the MemoryStore. As a result the DiskStore can never be as fast as the MemoryStore.

  Serialization speed is affected by the size of the objects being serialized and their type. It has been found in the ElementTest test that:

  * The serialization time for a Java object being a large Map of String arrays was 126ms, where the a serialized size was 349,225 bytes.
  * The serialization time for a byte[] was 7ms, where the serialized size was 310,232 bytes

  Byte arrays are 20 times faster to serialize. Make use of byte arrays to increase DiskStore performance.

- RAMFS

  One option to speed up disk stores is to use a RAM file system. On some operating systems there are a plethora of file systems to choose from. For example, the Disk Cache has been successfully used with Linux' RAMFS file system. This file system simply consists of memory. Linux presents it as a file system. The Disk Cache treats it like a normal disk - it is just way faster. With this type of file system, object serialization becomes the limiting factor to performance.

Chapter 14

Virtual Machine Shutdown Considerations

14.1

The DiskStore can optionally be configured to persist between CacheManager and Virtual Machine instances. See documentation on the diskPersistent cache attribute for information on how to do this.

When diskPersistent is turned on for a cache, a Virtual Machine shutdown hook is added to enable the DiskStore to persist itself. When the Virtual Machine shuts down, the the hook runs and, if the cache is not already disposed, it calls dispose. Any elements in the MemoryStore are spooled to the DiskStore. The DiskStore then flushes the spool, and writes the index to disk.

The cache shutdown hooks will run when:

- a program exists normally. e.g. System.exit() is called, or the last non-daemon thread exits
- the Virtual Machine is terminated. e.g. CTRL-C. This corresponds to `kill -SIGTERM pid` or `kill -15 pid` on Unix systems.

The cache shutdown hooks will not run when:

- the Virtual Machine aborts
- A SIGKILL signal is sent to the Virtual Machine process on Unix systems. e.g. `kill -SIGKILL pid` or `kill -9 pid`
- A `TerminateProcess` call is sent to the process on Windows systems.

If dispose was not called on the cache either by CacheManager.shutdown() or the shutdown hook, then the DiskStore will be corrupt when the application is next started. If this happens, it will be detected and the DiskStore file will be automatically truncated and a log warning level message is emitted. The cache will work normally, except that it will have lost all data.

Chapter 15

Hibernate Caching

Note these instructions are for Hibernate 3.1. Go to Guide for Version 1.1 for older instructions on how to use Hibernate 2.1.

Ehcache easily integrates with the Hibernate Object/Relational persistence and query service. Gavin King, the maintainer of Hibernate, is also a committer to the ehcache project. This ensures ehcache will remain a first class cache for Hibernate.

Since Hibernate 2.1, ehcache has been the default cache, for Hibernate.

The net.sf.ehcache.hibernate package provides classes integrating ehcache with Hibernate. Hibernate is an application of ehcache. Ehcache is also widely used a general-purpose Java cache.

To use ehcache with Hibernate do the following:

- Ensure ehcache is enabled in the Hibernate configuration.
- Add the cache element to the Hibernate mapping file, either manually, or via hibernatedoclet for each Domain Object you wish to cache.
- Add the cache element to the Hibernate mapping file, either manually, or via hibernatedoclet for each Domain Object collection you wish to cache.
- Add the cache element to the Hibernate mapping file, either manually, or via hibernatedoclet for each Hibernate query you wish to cache.
- Create a cache element in ehcache.xml

Each of these steps is illustrated using a fictional Country Domain Object.

For more about cache configuration in Hibernate see the Hibernate documentation. Parts of this chapter are drawn from Hibernate documentation and source code comments.

They are reproduced here for convenience in using ehcache.

15.1 Setting ehcache as the cache provider

15.1.1 Using the ehcache provider from the ehcache project

To ensure ehcache is enabled, verify that the hibernate.cache.provider_class property is set to net.sf.hibernate.cache.EhCacheProvide in the Hibernate configuration file; either hibernate.cfg.xml or hibernate.properties. The format given is for hibernate.cfg.xml.

If you are using hibernate-3 or hibernate-3.1 you will need to use the ehcache provider to use multiple SessionFactories/CacheManagers in a single VM. That provider should be integrated into the Hibernate-3.2 version.1

```
hibernate.cache.provider_class=net.sf.hibernate.cache.EhCacheProvider
net.sf.ehcache.configurationResourceName=/name_of_configuration_resource
```

The meaning of the properties is as follows:

hibernate.cache.provider_class - The fully qualified class name of the cache provider

net.sf.ehcache.configurationResourceName - The name of a configuration resource to use.

The resource is searched for in the root of the classpath. It is needed to support multiple CacheManagers in the same VM. It tells Hibernate which configuration to use. An example might be "ehcache-2.xml".

15.1.2 Using the ehcache provider from the Hibernate project

To use the one from the Hibernate project:

```
hibernate.cache.provider_class=org.hibernate.cache.EhCacheProvider
hibernate.cache.provider_configuration_file_resource_path=/name_of_configuration_resource
```

15.1.3 Programmatic setting of the Hibernate Cache Provider

The provider can also be set programmatically in Hibernate using Configuration.setProperty("hibernate.cache.provider_class", "net.sf.hibernate.cache.EhCacheProvider").

15.2 Hibernate Mapping Files

In Hibernate, each domain object requires a mapping file.

For example to enable cache entries for the domain object com.somecompany.someproject.domain.Country there would be a mapping file something like the following:

```
<hibernate-mapping>

<class
    name="com.somecompany.someproject.domain.Country"
    table="ut_Countries"
    dynamic-update="false"
    dynamic-insert="false"
>
...
</hibernate-mapping>
```

To enable caching, add the following element.

```
<cache usage="read-write|nonstrict-read-write|read-only" />
```

e.g.

```
<cache usage="read-write" />
```

15.2.1 read-write

Caches data that is sometimes updated while maintaining the semantics of "read committed" isolation level. If the database is set to "repeatable read", this concurrency strategy almost maintains the semantics. Repeatable read isolation is compromised in the case of concurrent writes.

This is an "asynchronous" concurrency strategy.

15.2.2 nonstrict-read-write

Caches data that is sometimes updated without ever locking the cache. If concurrent access to an item is possible, this concurrency strategy makes no guarantee that the item returned from the cache is the latest version available in the database. Configure your cache timeout accordingly! This is an "asynchronous" concurrency strategy.

This policy is the fastest. It does not use synchronized methods whereas read-write and read-only both do.

15.2.3 read-only

Caches data that is never updated.

15.3 Hibernate Doclet

Hibernate Doclet, part of the XDoclet project, can be used to generate Hibernate mapping files from markup in JavaDoc comments.

Following is an example of a Class level JavaDoc which configures a read-write cache for the Country Domain Object:

```
/**
* A Country Domain Object
*
* @hibernate.class table="COUNTRY"
* @hibernate.cache usage="read-write"
*/
public class Country implements Serializable
{
  ...
}
```

The @hibernate.cache usage tag should be set to one of read-write, nonstrict-read-write and read-only.

15.4 Configuration with ehcache.xml

Because ehcache.xml has a defaultCache, caches will always be created when required by Hibernate. However more control can be exerted by specifying a configuration per cache, based on its name.

In particular, because Hibernate caches are populated from databases, there is potential for them to get very large. This can be controlled by capping their maxElementsInMemory and specifying whether to overflowToDisk beyond that.

Hibernate uses a specific convention for the naming of caches of Domain Objects, Collections, and Queries.

15.4.1 Domain Objects

Hibernate creates caches named after the fully qualified name of Domain Objects.

So, for example to create a cache for com.somecompany.someproject.domain.Country create a cache configuration entry similar to the following in ehcache.xml.

```
<cache
    name="com.somecompany.someproject.domain.Country"
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="true"
/>
```

15.4.2 Hibernate

CacheConcurrencyStrategy read-write, nonstrict-read-write and read-only policies apply to Domain Objects.

15.4.3 Collections

Hibernate creates collection caches named after the fully qualified name of the Domain Object followed by "." followed by the collection field name.

For example, a Country domain object has a set of advancedSearchFacilities. The Hibernate doclet for the accessor looks like:

```
 /**
  * Returns the advanced search facilities that should appear for this country.
  * @hibernate.set cascade="all" inverse="true"
  * @hibernate.collection-key column="COUNTRY_ID"
  * @hibernate.collection-one-to-many class="com.wotif.jaguar.domain.AdvancedSearchFacility"
  * @hibernate.cache usage="read-write"
  */
 public Set getAdvancedSearchFacilities() {
    return advancedSearchFacilities;
 }
```

You need an additional cache configured for the set. The ehcache.xml configuration looks like:

```
<cache name="com.somecompany.someproject.domain.Country"
     maxElementsInMemory="50"
```

```
     eternal="false"
     timeToLiveSeconds="600"
     overflowToDisk="true"
 />
<cache
     name="com.somecompany.someproject.Country.advancedSearchFacilities"
     maxElementsInMemory="450"
     eternal="false"
     timeToLiveSeconds="600"
     overflowToDisk="true"
 />
```

15.4.4 Hibernate CacheConcurrencyStrategy

read-write, nonstrict-read-write and read-only policies apply to Domain Object collections.

15.4.5 Queries

Hibernate allows the caching of query results. Two caches, one called "net.sf.hibernate.cache.StandardQueryCache" in version 2.1.4 and higher and "net.sf.hibernate.cache.QueryCache" in versions 2.1.0 - 2.1.3, and one called "net.sf.hibernate.cache.UpdateTimestampsCache" are always used.

15.4.6 StandardQueryCache

This cache is used if you use a query cache without setting a name. A typical ehcache.xml configuration is:

```
<cache
    name="net.sf.hibernate.cache.StandardQueryCache"
    maxElementsInMemory="5"
    eternal="false"
    timeToLiveSeconds="120"
    overflowToDisk="true"/>
```

15.4.7 UpdateTimestampsCache

Tracks the timestamps of the most recent updates to particular tables. It is important that the cache timeout of the underlying cache implementation be set to a higher value than the timeouts of any of the query caches. In fact, it is recommend that the the underlying cache not be configured for expiry at all.

A typical ehcache.xml configuration is:

```
<cache
    name="net.sf.hibernate.cache.UpdateTimestampsCache"
    maxElementsInMemory="5000"
    eternal="true"
    overflowToDisk="true"/>
```

15.4.8 Named Query Caches

In addition, a QueryCache can be given a specific name in Hibernate using Query.setCacheRegion(String name). The name of the cache in ehcache.xml is then the name given in that method. The name can be whatever you want, but by convention you should use "query." followed by a descriptive name.

E.g.

```
<cache name="query.AdministrativeAreasPerCountry"
    maxElementsInMemory="5"
    eternal="false"
    timeToLiveSeconds="86400"
    overflowToDisk="true"/>
```

15.4.9 Using Query Caches

For example, let's say we have a common query running against the Country Domain.

Code to use a query cache follows:

```
public List getStreetTypes(final Country country) throws HibernateException {
 final Session session = createSession();
 try {
        final Query query = session.createQuery(

        "select st.id, st.name"
        + " from StreetType st "
        + " where st.country.id = :countryId "
        + " order by st.sortOrder desc, st.name");
        query.setLong("countryId", country.getId().longValue());
        query.setCacheable(true);
        query.setCacheRegion("query.StreetTypes");
        return query.list();
     } finally {
        session.close();
     }
 }
```

The `query.setCacheable(true)` line caches the query.

The `query.setCacheRegion("query.StreetTypes")` line sets the name of the Query Cache.

15.4.10 Hibernate CacheConcurrencyStrategy

None of read-write, nonstrict-read-write and read-only policies apply to Domain Objects. Cache policies are not configurable for query cache. They act like a non-locking read only cache.

15.5 Hibernate Caching Performance Tips

To get the most out of ehcache with Hibernate, Hibernate's use of it's in-process cache is important to understand.

15.5.1 In-Process Cache

From Hibernate's point of view, Ehcache is an in-process cache. Cached objects are accessible across different sessions. They are common to the Java process.

15.5.2 Object Id

Hibernate identifies cached objects via an object id. This is normally the primary key of a database row.

15.5.3 Session.load

Session.load will always try to use the cache.

15.5.4 Session.find and Query.find

Session.find does not use the cache for the primary object. Hibernate will try to use the cache for any associated objects. Session.find does however cause the cache to be populated.

Query.find works in exactly the same way.

Use these where the chance of getting a cache hit is low.

15.5.5 Session.iterate and Query.iterate

Session.iterate always uses the cache for the primary object and any associated objects.

Query.iterate works in exactly the same way.

Use these where the chance of getting a cache hit is high.

Chapter 16

The Design of distributed ehcache

This is a discussion and explanation of the distributed design choices made in ehcache. One or more default implementations are provided in each area. A plug in mechanism has been provided which will allow interested parties to implement alternative approaches discussed here and hopefully contribute them back to ehcache.

16.1 Acknowledgements

Much of the material here was drawn from Data Access Patterns, by Clifton Nock.

Thanks to Will Pugh and ehcache contributor Surya Suravarapu for suggesting we take ehcache distributed.

Finally, thanks to James Strachan for making helpful suggestions.

16.2 Problems with Instance Caches in a Clustered Environment

Many production applications are deployed in clusters. If each application maintains its own cache, then updates made to one cache will not appear in the others. A workaround for web based applications is to use sticky sessions, so that a user, having established a session on one server, stays on that server for the rest of the session. A workaround for transaction processing systems using Hibernate is to do a session.refresh on each persistent object as part of the save. session.refresh explicitly reloads the object from the database, ignoring any cache values.

16.3 Replicated Cache

Another solution is to replicate data between the caches to keep them consistent. This is sometimes called cache coherency. Applicable operations include:

- put
- update (put which overwrites an existing entry)
- remove

16.4 Distributed Cache Terms

Distributed Cache - a cache instance that notifies others when its contents change

Notification - a mechanism to replicate changes

Topology - a layout for how replicated caches connect with and notify each other

16.5 Notification Strategies

The best way of notifying of put and update depends on the nature of the cache.

If the Element is not available anywhere else then the Element itself should form the payload of the notification. An example is a cached web page. This notification strategy is called copy. Where the cached data is available in a database, there are two choices. Copy as before, or invalidate the data. By invalidating the data, the application tied to the other cache instance will be forced to refresh its cache from the database, preserving cache coherency. Only the Element key needs to be passed over the network.

Ehcache supports notification through copy and invalidate, selectable per cache.

16.6 Topology Choices

16.6.1 Peer Cache Replicator

Each replicated cache instance notifies every other cache instance when its contents change. This requires n-1 notifications per change, where n is the number of cache instances in the cluster. If multicast is used, these notifications can be emitted as one notification from the originating cache.

16.6.2 Centralised Cache Replicator

Each replicated cache instance notifies a master cache instance when its contents change. The master cache then notifies the other instances. This requires one notification from the originating cache and n-2 notifications from the master cache to other slaves.

Ehcache uses a peer topology. The main advantages are simplicity and greater redundancy as there is no single point of failure.

16.7 Discovery Choices

In a peer based system, there needs to be a way for peers to discover each other so as to perform delivery of changes.

16.7.1 Multicast Discovery

In multicast discovery, peers join a multicast group on a specific IP address in the multicast range of 224.0.0.1 to 239.255.255.255 (specified in RFC1112) and a specific port. Each peer notifies the other group members of its membership.

This approach is simple and allows for dynamic entry and exit from the cluster.

16.7.2 Static List

Here a list of listeners in the cluster is configured. There is no dynamic entry or exit. Peer listener addresses must be known in advance.

Ehcache provides both techniques.

16.8 Delivery Mechanism Choices

16.8.1 Custom Socket Protocol

This approach uses a protocol built directly on TCP or UDP. Its primary advantage is high performance.

16.8.2 Multicast Delivery

The advantage with multicast is that the sender only transmits once. It is however based on UDP datagrams and is nonreliable. Practical experience on modern networks, network cards and operating systems has shown this approach to be quite lossy. Whether it would be for a specific combination is hard to predict. This approach is thought unlikely to produce sufficient reliability.

16.8.3 JMS Topics

JMS Topics are standard, well understood way to propagate messages to multiple subscribers. JMS is not used in the default ehcache implementation because many ehcache users are outside the scope of J2EE. However JMS based delivery, with its richer services, could be a could choice for J2EE bases systems.

16.8.4 RMI RMI is the default RPC mechanism in Java.

16.8.5 JXTA

JXTA is a peer to peer technology that provides discovery and delivery, together with much else.

16.8.6 JGroups

JGroups provides many of the desired features for a peer to peer distributed system. The default mode for JGroups on a LAN is UDP, which is not desired. However JGroups does provide reliably transmission using TCP, similar to the approach taken in ehcache.

16.8.7 The Default Implementation

Ehcache uses RMI, based on custom socket options for delivery in its default implementation.

Ehcache does not use JXTA or JGroups for the following reasons:

- enables fine control over distribution behaviour
- allows tuning specific to a distributed cache, rather than distribution generally
- reduces the number of dependent libraries to run ehcache

RMI is used by default because:

- it itself is the default remoting mechanism in Java
- it is mature
- it allows tuning of TCP socket options
- Element keys and values for disk storage must already be Serializable, therefore directly transmittable over RMI without the need for conversion to a third format such as XML.
- it can be configured to pass through firewalls
- RMI had improvements added to it with each release of Java, which can then be taken advantage of.

However the pluggable nature of ehcache's distribution mechanism allows for both of these approaches to be plugged in. These approaches may become a standard part of ehcache in a future release.

A JGroups implementation is planned for ehcache-1.2.1.

16.9 Replication Drawbacks and Solutions in ehcache's implementation

Some potentially significant obstacles have to be overcome if replication is to provide a net benefit.

16.9.1 Chatty Protocol

n-1 notifications need to happen each time a a cache instance change occurs. A very large amount of network traffic can be generated. This issue affect the synchronous replication mode of ehcache.

Ehcache provides an asynchronous replication mode which mitigates this effect. All changes are buffered for delivery. The queue is then checked each second and all messages delivered in one RMI call, as a list of messages, to each peer.

The characteristics of each RMI call will be those of RMI. Ehcache does however use a custom socket factory so that socked read timeout can be set.

16.9.2 Redundant Notifications

The cache instance that initiated the change should not receive its own notifications. To do so would add additional overhead. Also, notifications should not endlessly go back and forth as each cache listener gets changes caused by a remote replication.

Ehcache's CachePeerProvider indentifies the local cache instance and excludes it from the notification list. Each Cache has a GUID. That GUID can be compared with list of cache peers and the local peer excluded.

Infinite notifications are prevented by passing a flag when the cache operation occurs. Events with that flag are ignored by instanced of CacheReplicator.

16.9.3 Potential for Inconsisent Data

Timing scenarios, race conditions, delivery, reliability constraints and concurrent updates to the same cached data can cause inconsistency (and thus a lack of coherency) across the cache instances.

This potential exists within the ehcache implementation. These issues are the same as what is seen when two completely separate systems are sharing a database; a common scenario.

Whether data inconsistency is a problem depends on the data and how it is used. For those times when it is important, ehcache provides for synchronous delivery of updates via invalidation. These are discussed below:

16.9.4 Synchronous Delivery

Delivery can be specified to be synchronous or asynchronous. Asynchronous delivery gives faster returns to operations on the local cache and is usually preferred. Synchronous delivery adds time to the local operation, however requires successful delivery of an update to all peers in the cluster before the cache operation returns.

16.9.5 Update via Invalidation

The default is to update other caches by copying the new value to them. If the replicateUpdatesViaCopy property is set to false in the replication configuration, updates are made by removing the element in any other cache peers. This forces the applications using the cache peers to return to a canonical source for the data.

A similar effect can be obtained by setting the element TTL to a low value such as a second.

Note that these features impact cache performance and should not be used where the main purpose of a cache is performance boosting over coherency.

Chapter 17

Distributed Caching

As of version 1.2, Ehcache can be used as a distributed cache.

The distribution feature is built using plugins. Ehcache comes with some default distribution plugins which should be suitable for most applications. Other plugins can be developed. Developers should see the source code in the distribution package for the fullly documented API to see how to do that.

Though not necessary to use distributed caching an insight into the design decisions used in ehcache may be helpful. See the Design of distributed ehcache chapter.

The rest of this section documents the distribution plugins which are bundled with ehcache.

The following concepts are central to cache distribution:

- How do you know about the other caches that are in your cluster?
- What form of communication will be used to distribute messages?
- What is replicated? Puts, Updates, Expiries?
- When is it replicated? Synchronous or asynchronous?

To set up distributed caching you need to configure a PeerProvider, a CacheManagerPeerListener, which is done globally for a CacheManager. For each cache that will operate distributed, you then need to add a cacheEventListener to propagate messages.

17.1 Suitable Element Types

Only Serializable Elements are suitable for replication.

Some operations, such as remove, work off Element keys rather than the full Element itself. In this case the operation will be replicated provided the key is Serializable, even if the Element is not.

17.2 Peer Discovery

Ehcache has the notion of a group of caches acting as a distributed cache. Each of the caches is a peer to the others. There is no master cache. How do you know about the other caches that are in your cluster? This problem can be given the name Peer Discovery.

Ehcache provides two mechanisms for peer discovery, just like a car: manual and automatic.

To use one of the built-in peer discovery mechanisms specify the class attribute of `cacheManagerPeerProviderFactory` as `net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory` in the ehcache.xml configuration file.

17.2.1 Automatic Peer Discovery

Automatic discovery uses TCP multicast to establish and maintain a multicast group. It features minimal configuration and automatic addition to and deletion of members from the group. No a priori knowledge of the servers in the cluster is required. This is recommended as the default option.

Peers send heartbeats to the group once per second. If a peer has not been heard of for 5 seconds it is dropped from the group. If a new peer starts sending heartbeats it is admitted to the group.

Any cache within the configuration set up as replicated will be made available for discovery by other peers.

To set automatic peer discovery, specify the properties attribute of `cacheManagerPeerProviderFactory` as follows:

peerDiscovery=automatic multicastGroupAddress=multicast address |multicast host name multicastGroup-Port=port

Example

Suppose you have two servers in a cluster. You wish to distribute sampleCache11 and sampleCache12. The configuration required for each server is identical:

Configuration for server1 and server2

```
<cacheManagerPeerProviderFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=automatic, multicastGroupAddress=230.0.0.1,
multicastGroupPort=4446"/>
```

17.2.2 Manual Peer Discovery

Manual peer configuration requires the IP address and port of each listener to be known. Peers cannot be added or removed at runtime. Manual peer discovery is recommended where there are technical difficulties using multicast, such as a router between servers in a cluster that does not propagate multicast datagrams. You can also use it to set up one way replications of data, by having server2 know about server1 but not vice versa.

To set manual peer discovery, specify the properties attribute of `cacheManagerPeerProviderFactory` as follows: peerDiscovery=manual rmiUrls=//server:port/cacheName, ...

The rmiUrls is a list of the cache peers of the server being configured. Do not include the server being configured in the list.

Example

Suppose you have two servers in a cluster. You wish to distribute sampleCache11 and sampleCache12. Following is the configuration required for each server:

Configuration for server1

```
<cacheManagerPeerProviderFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
```

```
properties="peerDiscovery=manual,
rmiUrls=//server2:40001/sampleCache11|//server2:40001/sampleCache12"/>
```

Configuration for server2

```
<cacheManagerPeerProviderFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=manual,
rmiUrls=//server1:40001/sampleCache11|//server1:40001/sampleCache12"/>
```

17.3 Configuring a CacheManagerPeerListener

A CacheManagerPeerListener listens for messages from peers to the current CacheManager.

You configure the CacheManagerPeerListener by specifiying a CacheManagerPeerListenerFactory which is used to create the CacheManagerPeerListener using the plugin mechanism.

The attributes of cacheManagerPeerListenerFactory are:

- class - a fully qualified factory class name * properties - comma separated properties having meaning only to the factory.

  Ehcache comes with a built-in RMI-based distribution system. The listener component is RMI-CacheManagerPeerListener which is configured using RMICacheManagerPeerListenerFactory. It is configured as per the following example:

  ```
  <cacheManagerPeerListenerFactory
  class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"

  properties="hostName=localhost, port=40001,
  socketTimeoutMillis=2000"/>
  ```

  Valid properties are:

- hostName (optional) - the hostName of the host the listener is running on. Specify where the host is multihomed and you want to control the interface over which cluster messages are received.

  The hostname is checked for reachability during CacheManager initialisation.

  If the hostName is unreachable, the CacheManager will refuse to start and an CacheException will be thrown indicating connection was refused.

  If unspecified, the hostname will use `InetAddress.getLocalHost().getHostAddress()`, which corresponds to the default host network interface.

  Warning: Explicitly setting this to localhost refers to the local loopback of 127.0.0.1, which is not network visible and will cause no replications to be received from remote hosts. You should only use this setting when multiple CacheManagers are on the same machine.

- port (mandatory) - the port the listener listens on.

- socketTimeoutMillis (optional) - the number of seconds client sockets will wait when sending messages to this listener until they give up. By default this is 2000ms.

17.4 Configuring CacheReplicators

Each cache that will be distributed needs to set a cache event listener which then replicates messages to the other CacheManager peers. This is done by adding a cacheEventListenerFactory element to each cache's configuration.

```
<!-- Sample cache named sampleCache2. -->
<cache name="sampleCache2"
       maxElementsInMemory="10"
       eternal="false"
       timeToIdleSeconds="100"
       timeToLiveSeconds="100"
       overflowToDisk="false">
    <cacheEventListenerFactory class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"
                               properties="replicateAsynchronously=true, replicatePuts=true,
</cache>
```

class - use net.sf.ehcache.distribution.RMICacheReplicatorFactory

The factory recognises the following properties:

- replicatePuts=true |false - whether new elements placed in a cache are replicated to others. Defaults to true.
- replicateUpdates=true |false - whether new elements which override an element already existing with the same key are replicated. Defaults to true.
- replicateRemovals=true - whether element removals are replicated. Defaults to true.
- replicateAsynchronously=true |false - whether replications are asyncrhonous (true) or synchronous (false). Defaults to true.
- replicateUpdatesViaCopy=true |false - whether the new elements are copied to other caches (true), or whether a remove message is sent. Defaults to true.

To reduce typing if you want default behaviour, which is replicate everything in asynchronous mode, you can leave off the `RMICacheReplicatorFactory` properties as per the following example:

```
<!-- Sample cache named sampleCache4. All missing RMICacheReplicatorFactory properties defaul
<cache name="sampleCache4"
       maxElementsInMemory="10"
       eternal="true"
       overflowToDisk="false"
       memoryStoreEvictionPolicy="LFU">
    <cacheEventListenerFactory class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/
</cache>
```

17.5 Common Problems

17.5.1 Tomcat on Windows

There is a bug in Tomcat and/or the JDK where any RMI listener will fail to start on Tomcat if the installation path has spaces in it. See http://archives.java.sun.com/cgi-bin/wa?A2=ind0205&L=rmi-users&P=797 and http://www.ontotext.com/kim/doc/sys-doc/faq-howto-bugs/known-bugs.html.

As the default on Windows is to install Tomcat in "Program Files", this issue will occur by default.

17.5.2 Multicast Blocking

The automatic peer discovery process relies on multicast. Multicast can be blocked by routers. Virtualisation technologies like Xen and VMWare may be blocking multicast. If so enable it. You may also need to turn it on in the configuration for your network interface card.

An easy way to tell if your mutlicast is getting through is to use the ehcache remote debugger and watch for the heartbeat packets to arrive.

Chapter 18

The Design of the ehcache constructs package

This is a discussion and explanation of the reasons for and the design forces behind the constructs package in ehcache.

18.1 Acknowledgements

Much of the material here was drawn from Concurrent Programming in Java by Doug Lea. Thanks also to Doug for answering several questions along the way.

18.2 The purpose of the Constructs package

Doug Lea in his book Concurrent Programming in Java talks about concurrency support constructs. One meaning of a construct is "an abstract or general idea inferred or derived from specific instances". Just like patterns emerge from noting the similarities of problems and gradually finding a solution to classes of them, so to constructs are general solutions to commond problems.

The ehcache constructs package, literally the net.sf.ehcache.constructs package, provides ready to use, extensible implementations are offered to solve common problems in J2EE and light-weight container applications.

Why not leave ehcache at the core and let everyone create their own applications? Well, everyone is doing that. But getting it right can be devilishly hard.

18.3 Caching meets Concurrent Programming

So, why not just use Doug's library or the one he contributed to in JDK1.5? The ehcache constructs are around the intersection of concurrency programming and caching. It uses a number of Doug's classes copied verbatim into the net.sf.ehcache.concurrent package, as permiited under the license.

18.4 What can possibly go wrong?

That is a favourite tongue in cheek saying of Adam Murdoch, an original contributor to the ehcache project. The answer in concurrent programming is a lot.

(The following section is based heavily on Chapter 1.3 of Doug Lea's Concurrent Programming in Java).

There are two often conflicting design goals at play in concurrent programming. They are:

- liveness, where something eventually happens within an activity.
- safety, where nothing bad ever happens to an object.

18.4.1 Safety Failures

Failures of safety include:

- Read/Write Conflicts, where one thread is reading from a field and another is writing to it. The value read depends on who won the race.
- Write/Write Conflicts, where two threads write to the same field. The value on the next read is impossible to predict.

  A cache is similar to a global variable. By its nature it is accessible to multiple threads. Cache entries, and the locking around them, are often highly contended for.

18.4.2 Liveness Failures

Failures of liveness include:

- Deadlock. This is caused by a circular dependency among locks. The threads involved cannot make progress.
- Missed Signals. A thread entered the wait state after a notification to wake it up was produced.
- Nested monitor lockouts. A waiting thread holds a lock needed by a thread wishing to wake it up
- Livelock. A continously retried action continously fails.
- Starvation. Some threads never get allocated CPU time.
- Resource Exhaustion. All resourcesof some kind are in use by threads, none of which will give one up.
- Distributed Failure. A remote machine connected by socket becomes inaccessible.
- Stampede. With notifyAll(), all threads wake up and in a stampede, attempt to make progress.

18.5 The constructs

18.5.1 Blocking Cache

Imagine you have a very busy web site with thousands of concurrent users. Rather than being evenly distributed in what they do, they tend to gravitate to popular pages. These pages are not static, they have dynamic data which goes stale in a few minutes. Or imagine you have collections of data which go stale in a few minutes. In each case the data is extremely expensive to calculate.

Let's say each request thread asks for the same thing. That is a lot of work. Now, add a cache. Get each thread to check the cache; if the data is not there, go and get it and put it in the cache. Now, imagine that there are so many users contending for the same data that in the time it takes the first user to request the data and put it in the cache, 10 other users have done the same thing. The upstream system, whether a JSP or velocity page, or interactions with a service layer or database are doing 10 times more work than they need to.

Enter the BlockingCache.


java.lang
Object
net.sf.ehcache
Ehcache
net.sf.ehcache.constructs.blocking
BlockingCache
cache : Ehcache
+ LOCK_NUMBER : int
# locks : Mutex[]
# timeoutMillis : int
+ BlockingCache(Ehcache)
+ bootstrap() : void
+ calculateInMemorySize() : long
+ clearStatistics() : void
+ clone() : Object
+ dispose() : void
+ evictExpiredElements() : void
+ flush() : void
+ get(Serializable) : Element
+ get(Object) : Element
+ getBootstrapCacheLoader() : BootstrapCacheLoader
# getCache() : Ehcache
+ getCacheEventNotificationService() : RegisteredEventListeners
+ getCacheManager() : CacheManager
+ getDiskExpiryThreadIntervalSeconds() : long
+ getDiskStoreHitCount() : int
+ getDiskStoreSize() : int
+ getGuid() : String
+ getHitCount() : int
+ getKeys() : List
+ getKeysNoDuplicateCheck() : List
+ getKeysWithExpiryCheck() : List
# getLockForKey(Object) : Mutex
+ getMaxElementsInMemory() : int
+ getMemoryStoreEvictionPolicy() : MemoryStoreEvictionPolicy
+ getMemoryStoreHitCount() : int
+ getMemoryStoreSize() : long
+ getMissCountExpired() : int
+ getMissCountNotFound() : int
+ getName() : String
+ getQuiet(Serializable) : Element
+ getQuiet(Object) : Element
+ getSize() : int
+ getStatistics() : Statistics
+ getStatisticsAccuracy() : int
+ getStatus() : Status
+ getTimeoutMillis() : int
+ getTimeToIdleSeconds() : long
+ getTimeToLiveSeconds() : long
+ initialise() : void
+ isDiskPersistent() : boolean
+ isElementInMemory(Serializable) : boolean
+ isElementInMemory(Object) : boolean
+ isElementOnDisk(Serializable) : boolean
+ isElementOnDisk(Object) : boolean
+ isEternal() : boolean
+ isExpired(Element) : boolean
+ isKeyInCache(Object) : boolean
+ isOverflowToDisk() : boolean
+ isValueInCache(Object) : boolean
+ liveness() : String
+ put(Element) : void
+ put(Element, boolean) : void
+ putQuiet(Element) : void
+ remove(Serializable) : boolean
+ remove(Serializable, boolean) : boolean
+ remove(Object) : boolean
+ remove(Object, boolean) : boolean
+ removeAll() : void
+ removeAll(boolean) : void
+ removeQuiet(Serializable) : boolean
+ removeQuiet(Object) : boolean
+ setBootstrapCacheLoader(BootstrapCacheLoader) : void
+ setCacheManager(CacheManager) : void
+ setDiskStorePath(String) : void
+ setName(String) : void
+ setStatisticsAccuracy(int) : void
+ setTimeoutMillis(int) : void
net.sf.ehcache
Ehcache
net.sf.ehcache.event
RegisteredEventListeners
org.apache.commons.logging
Log
java.lang
Object
String
net.sf.ehcache.constructs.concurrent
Mutex
java.util
List<E>
net.sf.ehcache
CacheManager
Element
Statistics
Status
net.sf.ehcache.store
MemoryStoreEvictionPolicy
java.io
Serializable
net.sf.ehcache.bootstrap
BootstrapCacheLoader

It is blocking because all threads requesting the same key wait for the first thread to complete. Once the first thread has completed the other threads simply obtain the cache entry and return.

The BlockingCache can scale up to very busy systems.

18.5.2 SelfPopulatingCache

You want to use the BlockingCache, but the requirement to always release the lock creates gnarly code. You also want to think about what you are doing without thinking about the caching.

Enter the SelfPopulatingCache. The name SelfPopulatingCache is synonymous with Pull-through cache, which is a common caching term. SelfPopulatingCache though always is in addition to a BlockingCache.

SelfPopulatingCache uses a `CacheEntryFactory`, that given a key, knows how to populate the entry.

18.5.3 CachingFilter

You want to use the BlockingCache with web pages, but the requirement to always release the lock creates gnarly code. You also want to think about what you are doing without thinking about the caching.

Enter the CachingFilter, a Servlet 2.3 compliant filter. Why not just do a JSP tag library, like OSCache? The answer is that you want the caching of your responses to be independent of the rendering technology. The filter chain is reexcuted every time a RequestDispatcher is involved. This is on every jsp:include and every Servlet. And you can programmatically add your own. If you have content generated by JSP, Velocity, XSLT, Servlet output or anything else, it can all be cached by CachingFilter. A separation of concerns.

How do you determine what the key of a page is? The filter has an abstract calculateKey method, so it is up to you.

You notice a problem and an opportunity. The problem is that the web pages you are caching are huge. That chews up either a lot of memory (MemoryStore) or a lot of disk space (DiskStore). Also you notive that these pages take their time going over the Internet. The opportunity is that you notice that all modern browsers support gzip encoding. A survey of logs reveals that 85% of the time the browser accepts gzipping. (The majority of the 15% that does not is IE behind a proxy). Ok, so gzip the response before caching it. Ungzipping is fast - so just ungzip for the 15% of the time the browser does not accept gzipping.

18.5.4 SimplePageCachingFilter

What if you just want to get started with the CachingFilter and don't want to think too hard? Just use SimplePageCachingFilter which has a calculateKey method already implemented. It uses `httpRequest.getRequestURI()).appen` for the key. This works most of the time. It tends to get less effective when referrals and affiliates are added to the query, which is the case for a lot of e-commerce sites.

SimplePageCachingFilter is 10 lines of code.

18.5.5 PageFragmentCachingFilter

You notice that an entire page cannot be cached because the data on it vary in staleness. Say, an address which changes very infrequently, and the price and availability of inventory, which changes quite a lot. Or you have a portal, with lots of components and with different stalenesses. Or you use the replicated cache functionality in ehcache and you only want to rebuild the part of the page that got invalidated.

Enter the PageFragmentCachingFilter. It does everyting that SimplePageCachingFilter does, except it never gzips, so the fragments can be combined.

18.5.6 SimplePageFragmentCachingFilter

What if you just want to get started with the PageFragmentCachingFilter and don't want to think too hard? Just use SimplePageFragmentCachingFilter which has a calculateKey method already implemented. It uses `httpRequest.getRequestURI()).append(httpRequest.getQueryString()` for the key. This works most of the time. It tends to get less effective when referrals and affiliates are added to the query, which is the case for a lot of e-commerce sites.

SimplePageFragmentCachingFilter is 10 lines of code.

18.5.7 AsynchronousCommandExecutor

What happens if your JMS server is down? The usual answer it to have two of them. Unfortunately, not all JMS servers do a good job of clustering. Plus it takes twice the hardware.

Once a message makes it to a JMS server, they can usually be configured to store the message in a database. You are pretty safe after that if there is a crash.

Enter AsynchronousCommandExecutor. It lets you create a command for future execution. The command is cached and is then immediately executed in another thread. Thus the asynchronous bit. If it fails, it retries on a set interval up to a set number of times. Thus it is fault-tolerant.

Use this where you really don't want to lose messages or commands that execute against another system.

18.6 Real-life problems in the constructs package and their solutions

At the time of revising this document, ehcache is almost three years old. That leaves plenty of time to observe some concurrency failures. The problems that arose and how they were fixed are illustrative of the subtleties of concurrent programming.

18.6.1 The Blocking Cache Stampede

The first BlockingCache implementation ran for almost a year on a very busy application before the first problems came to light. It was using notifyAll() together with coarse grained synchronization on the BlockingCache instance.

Once the load on the cache got very high indeed, the thread with the lock would notifyAll. Then hundreds of threads would "stampede" - they would each attempt to get the lock. Gradually more and more CPU time was spent resolving contention for the object lock after each notifyAll. Eventually the server threads went to 1500 and server output dropped to almost nothing.

The solution was to create a Mutex representing each key as it was requested and to lock on that rather than the BlockingCache itself. That gave a 10 times improvement in scalability. See Scalability Test vs the old ScalabilityTest.

18.6.2 The Blank Page problem

About a year into the use of the CachingFilter, the idea to gzip was born. Having implemented it, it worked fine. A few weeks into production use strange reports came in that people were occasionally getting blank pages. Timing suggested the gzip change, but how? A tester came across similar issues that had been reported with Apache mod_gzip. It looked like there was a rare code path that was somehow screwing up.

In the end, that was how the filters made their way into the ehcache project. The level of testing required to focus on the issue was way beyond what you would normally do in a business app. In the end I sat down with the Servlet specification and looked at everything that could go wrong. I ended up creating

FilterNonReentrantException, AlreadyGzippedException and ResponseHeadersNotModifiableException. These conditions are detected and an exception thrown rather than a blank page. Then the developer fixes the coding error that produced it.

The exception contain comments on how each issue happens, which are reproduced below:

FilterNonReentrantException - Thrown when it is detected that a caching filter's doFilter method is reentered by the same thread. Reentrant calls will block indefinitely because the first request has not yet unblocked the cache. Nasty.

AlreadyGzippedException - The web package performs gzipping operations. One cause of problems on web browsers is getting content that is double or triple gzipped. They will either get gobblydeegook or a blank page. This exception is thrown when a gzip is attempted on already gzipped content.

ResponseHeadersNotModifiableException - A gzip encoding header needs to be added for gzipped content. The HttpServletResponse#setHeader() method is used for that purpose. If the header had already been set, the new value normally overwrites the previous one. In some cases according to the servlet specification, setHeader silently fails. Two scenarios where this happens are:

- The response is committed.
- RequestDispatcher#include method caused the request.

This issue is extremely subtle and nasty.

There are tests that reproduce each of these issues. The CachingFilter and its subclasses have been in production for nearly two years with no more reports of trouble.

18.6.3 Blocking Cascade

Let's say you do use the BlockingCache but something goes wrong upstream. Maybe it is something like a database backup that slows the database down for 10 minutes. Or greedy SQL. With the BlockingCache the JDBC connection will eventually timeout. The first thread fails. The next queued thread then attempts the same thing. It fails. And so on. While this is going on, more and more threads queue up. The result is a Blocking cascade. Eventually, if the slow upstream server or process does not pick up you exhaust the thread limit on your server and it goes down with an OutOfMemoryError.

Is this what you want? Or would you prefer to have the affected part of the system degrade with errors while the rest of the system keeps ticking? That is a judgement call.

BlockingCache has a parameter in its constructor called timeoutMillis. If you set that then any queued thread will immediately timeout when its turn comes in the above scenario. Some requests get exceptions, but you do not lose your VM.

Chapter 19

CacheManager Event Listeners

- Configuration
- Implementing a CacheManagerEventListenerFactory and CacheManagerEventListener

CacheManager event listeners allow implementers to register callback methods that will be executed when a `CacheManager` event occurs. Cache listeners implement the CacheManagerEventListener interface.

The events include:

- adding a `Cache`
- removing a `Cache`

Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

19.1 Configuration

One CacheManagerEventListenerFactory and hence one CacheManagerEventListener can be specified per CacheManager instance.

The factory is configured as below:

```
<cacheManagerEventListenerFactory class=""
properties=""/>
```

The entry specifies a CacheManagerEventListenerFactory which will be used to create a CacheManager-PeerProvider, which is notified when Caches are added or removed from the CacheManager.

The attributes of CacheManagerEventListenerFactory are:

- `class` - a fully qualified factory class name
- `properties` - comma separated properties having meaning only to the factory.

  Callbacks to listener methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

  If no class is specified, or there is no cacheManagerEventListenerFactory element, no listener is created. There is no default.

19.2 Implementing a CacheManagerEventListenerFactory and CacheManagerEventListener

CacheManagerEventListenerFactory is an abstract factory for creating cache manager listeners. Implementers should provide their own concrete factory extending this abstract factory. It can then be configured in ehcache.xml.

The factory class needs to be a concrete subclass of the abstract factory CacheManagerEventListenerFactory, which is reproduced below:

```
/**
 * An abstract factory for creating {@link CacheManagerEventListener}s. Implementers should
 * provide their own concrete factory extending this factory. It can then be configured in
 * ehcache.xml
 *
 * @author Greg Luck
 * @version $Id: cachemanager_event_listeners.apt 135 2006-06-26 06:55:03Z gregluck $
 * @see "http://ehcache.sourceforge.net/documentation/cachemanager_event_listeners.html"
 */
public abstract class CacheManagerEventListenerFactory {

    /**
     * Create a <code>CacheEventListener</code>
     *
     * @param properties implementation specific properties. These are configured as comma
     *                   separated name value pairs in ehcache.xml. Properties may be null
     * @return a constructed CacheManagerEventListener
     */
    public abstract CacheManagerEventListener
            createCacheManagerEventListener(Properties properties);
}
```

The factory creates a concrete implementation of CacheManagerEventListener, which is reproduced below:

```
/**
 * Allows implementers to register callback methods that will be executed when a
 * <code>CacheManager</code> event occurs.
 * The events include:
 * <ol>
 * <li>adding a <code>Cache</code>
 * <li>removing a <code>Cache</code>
 * </ol>
 * <p/>
 * Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of
 * the implementer to safely handle the potential performance and thread safety issues
 * depending on what their listener is doing.
 * @author Greg Luck
 * @version $Id: cachemanager_event_listeners.apt 135 2006-06-26 06:55:03Z gregluck $
 * @since 1.2
 * @see CacheEventListener
 */
public interface CacheManagerEventListener {


    /**
     * Called immediately after a cache has been added and activated.
     * <p/>
```

```
     * Note that the CacheManager calls this method from a synchronized method. Any attempt to
     * call a synchronized method on CacheManager from this method will cause a deadlock.
     * <p/>
     * Note that activation will also cause a CacheEventListener status change notification
     * from {@link net.sf.ehcache.Status#STATUS_UNINITIALISED} to
     * {@link net.sf.ehcache.Status#STATUS_ALIVE}. Care should be taken on processing that
     * notification because:
     * <ul>
     * <li>the cache will not yet be accessible from the CacheManager.
     * <li>the addCaches methods whih cause this notification are synchronized on the
     * CacheManager. An attempt to call {@link net.sf.ehcache.CacheManager#getCache(String)}
     * will cause a deadlock.
     * </ul>
     * The calling method will block until this method returns.
     * <p/>
     * @param cacheName the name of the <code>Cache</code> the operation relates to
     * @see CacheEventListener
     */
    void notifyCacheAdded(String cacheName);

    /**
     * Called immediately after a cache has been disposed and removed. The calling method will
     * block until this method returns.
     * <p/>
     * Note that the CacheManager calls this method from a synchronized method. Any attempt to
     * call a synchronized method on CacheManager from this method will cause a deadlock.
     * <p/>
     * Note that a {@link CacheEventListener} status changed will also be triggered. Any
     * attempt from that notification to access CacheManager will also result in a deadlock.
     * @param cacheName the name of the <code>Cache</code> the operation relates to
     */
    void notifyCacheRemoved(String cacheName);

}
```

The implementations need to be placed in the classpath accessible to ehcache. Ehcache uses the Class-Loader returned by `Thread.currentThread().getContextClassLoader()` to load classes.

Chapter 20

Cache Event Listeners

Cache listeners allow implementers to register callback methods that will be executed when a cache event occurs. Cache listeners implement the CacheEventListener interface.

The events include:

- an Element has been put
- an Element has been updated. Updated means that an Element exists in the Cache with the same key as the Element being put.
- an Element has been removed
- an Element expires, either because timeToLive or timeToIdle have been reached.

Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

Listeners are guaranteed to be notified of events in the order in which they occurred.

Elements can be put or removed from a Cache without notifying listeners by using the putQuiet and removeQuiet methods.

20.1 Configuration

Cache event listeners are configured per cache. Each cache can have multiple listeners.

Each listener is configured by adding a cacheManagerEventListenerFactory element as follows:

```
<cache ...>
<cacheEventListenerFactory class="" properties=""/>
...
</cache>
```

The entry specifies a CacheManagerEventListenerFactory which is used to create a CachePeerProvider, which then receives notifications.

The attributes of CacheManagerEventListenerFactory are:

- class - a fully qualified factory class name * properties - an optional comma separated properties having meaning only to the factory.

  Callbacks to listener methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

20.2 Implementing a CacheEventListenerFactory and CacheEventListener

CacheEventListenerFactory is an abstract factory for creating cache event listeners. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml

The factory class needs to be a concrete subclass of the abstract factory class CacheEventListenerFactory, which is reproduced below:

```
/**
 * An abstract factory for creating listeners. Implementers should provide their own
 * concrete factory extending this factory. It can then be configured in ehcache.xml
 *
 * @author Greg Luck
 * @version $Id: cache_event_listeners.apt 135 2006-06-26 06:55:03Z gregluck $
 */
public abstract class CacheEventListenerFactory {

    /**
     * Create a <code>CacheEventListener</code>
     *
     * @param properties implementation specific properties. These are configured as comma
     *                   separated name value pairs in ehcache.xml
     * @return a constructed CacheEventListener
     */
    public abstract CacheEventListener createCacheEventListener(Properties properties);

}
```

The factory creates a concrete implementation of the CacheEventListener interface, which is reproduced below:

```
/**
 * Allows implementers to register callback methods that will be executed when a cache event
 *  occurs.
 * The events include:
 * <ol>
 * <li>put Element
 * <li>update Element
 * <li>remove Element
 * <li>an Element expires, either because timeToLive or timeToIdle has been reached.
 * </ol>
 * <p/>
 * Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of
 * the implementer to safely handle the potential performance and thread safety issues
 * depending on what their listener is doing.
 * <p/>
 * Events are guaranteed to be notified in the order in which they occurred.
 * <p/>
```

```
 * Cache also has putQuiet and removeQuiet methods which do not notify listeners.
 *
 * @author Greg Luck
 * @version $Id: cache_event_listeners.apt 135 2006-06-26 06:55:03Z gregluck $
 * @see CacheManagerEventListener
 * @since 1.2
 */
public interface CacheEventListener extends Cloneable {

    /**
     * Called immediately after an element has been removed. The remove method will block until
     * this method returns.
     * <p/>
     * Ehcache does not chech for
     * <p/>
     * As the {@link net.sf.ehcache.Element} has been removed, only what was the key of the
     * element is known.
     * <p/>
     *
     * @param cache   the cache emitting the notification
     * @param element just deleted
     */
    void notifyElementRemoved(final Ehcache cache, final Element element) throws CacheException;

    /**
     * Called immediately after an element has been put into the cache. The
     * {@link net.sf.ehcache.Cache#put(net.sf.ehcache.Element)} method
     * will block until this method returns.
     * <p/>
     * Implementers may wish to have access to the Element's fields, including value, so the
     * element is provided. Implementers should be careful not to modify the element. The
     * effect of any modifications is undefined.
     *
     * @param cache   the cache emitting the notification
     * @param element the element which was just put into the cache.
     */
    void notifyElementPut(final Ehcache cache, final Element element) throws CacheException;

    /**
     * Called immediately after an element has been put into the cache and the element already
     * existed in the cache. This is thus an update.
     * <p/>
     * The {@link net.sf.ehcache.Cache#put(net.sf.ehcache.Element)} method
     * will block until this method returns.
     * <p/>
     * Implementers may wish to have access to the Element's fields, including value, so the
     * element is provided. Implementers should be careful not to modify the element. The
     * effect of any modifications is undefined.
     *
     * @param cache   the cache emitting the notification
     * @param element the element which was just put into the cache.
     */
    void notifyElementUpdated(final Ehcache cache, final Element element) throws CacheException;


    /**
     * Called immediately after an element is <i>found</i> to be expired. The
     * {@link net.sf.ehcache.Cache#remove(Object)} method will block until this method returns.
```

```
     * <p/>
     * As the {@link Element} has been expired, only what was the key of the element is known.
     * <p/>
     * Elements are checked for expiry in ehcache at the following times:
     * <ul>
     * <li>When a get request is made
     * <li>When an element is spooled to the diskStore in accordance with a MemoryStore
     * eviction policy
     * <li>In the DiskStore when the expiry thread runs, which by default is
     * {@link net.sf.ehcache.Cache#DEFAULT_EXPIRY_THREAD_INTERVAL_SECONDS}
     * </ul>
     * If an element is found to be expired, it is deleted and this method is notified.
     *
     * @param cache   the cache emitting the notification
     * @param element the element that has just expired
     *                <p/>
     *                Deadlock Warning: expiry will often come from the <code>DiskStore</code>
     *                expiry thread. It holds a lock to the DiskStorea the time the
     *                notification is sent. If the implementation of this method calls into a
     *                synchronized <code>Cache</code> method and that subsequently calls into
     *                DiskStore a deadlock will result. Accordingly implementers of this method
     *                should not call back into Cache.
     */
    void notifyElementExpired(final Ehcache cache, final Element element);


    /**
     * Give the replicator a chance to cleanup and free resources when no longer needed
     */
    void dispose();


    /**
     * Creates a clone of this listener. This method will only be called by ehcache before a
     * cache is initialized.
     * <p/>
     * This may not be possible for listeners after they have been initialized. Implementations
     * should throw CloneNotSupportedException if they do not support clone.
     * @return a clone
     * @throws CloneNotSupportedException if the listener could not be cloned.
     */
    public Object clone() throws CloneNotSupportedException;

}
```

The implementations need to be placed in the classpath accessible to ehcache.

See the chapter on Classloading for details on how classloading of these classes will be done.

Chapter 21

Frequently Asked Questions

21.1 Does ehcache run on JDK1.3?

Yes. It runs on JDK1.3, 1.4 and 5. The restriction for JDK1.3 is that you must either use the precompiled ehcache.jar or build it using JDK1.4 with a target of 1.3. This is because ehcache makes use of some JDK1.4 features but substitutes alternatives at runtime if it does not find those features.

21.2 Can you use more than one instance of ehcache in a single VM?

As of ehcache-1.2, yes. Create your CacheManager using new CacheManager(...) and keep hold of the reference. The singleton approach accessible with the getInstance(...) method is still available too. Remember that ehcache can supports hundreds of caches within one CacheManager. You would use separate CacheManagers where you want quite different configurations.

The Hibernate EhCacheProvider has also been updated to support this behaviour.

21.3 Can you use ehcache with Hibernate and outside of Hibernate at the same time?

Yes. You use 1 instance of ehcache and 1 ehcache.xml. You configure your caches with Hibernate names for use by Hibernate. You can have other caches which you interact with directly outside of Hibernate.

That is how I use ehcache in the original project it was developed in. For Hibernate we have about 80 Domain Object caches, 10 StandardQueryCaches, 15 Domain Object Collection caches.

We have around 5 general caches we interact with directly using BlockingCacheManager. We have 15 general caches we interact with directly using SelfPopulatingCacheManager. You can use one of those or you can just use CacheManager directly.

I have updated the documentation extensively over the last few days. Check it out and let me know if you have any questions. See the tests for example code on using the caches directly. Look at CacheManagerTest, CacheTest and SelfPopulatingCacheTest.

21.4 What happens when maxElementsInMemory is reached? Are the oldest items are expired when new ones come in?

When the maximum number of elements in memory is reached, the least recently used ("LRU") element is removed. Used in this case means inserted with a put or accessed with a get.

If the overflowToDisk cache attribute is false, the LRU Element is discarded. If true, it is transferred asynchronously to the DiskStore.

21.5 Is it thread safe to modify Element values after retrieval from a Cache?

Remember that a value in a cache element is globally accessible from multiple threads. It is inherently not thread safe to modify the value. It is safer to retrieve a value, delete the cache element and then reinsert the value.

The UpdatingCacheEntryFactory does work by modifying the contents of values in place in the cache. This is outside of the core of ehcache and is targeted at high performance CacheEntryFactories for SelfPopulatingCaches.

21.6 Can non-Serializable objects be stored in a cache?

As of ehcache-1.2, they can be stored in caches with MemoryStores.

Elements attempted to be replicated or overflowed to disk will be removed and a warning logged if not Serializable.

21.7 Why is there an expiry thread for the DiskStore but not for the MemoryStore?

Because the memory store has a fixed maximum number of elements, it will have a maximum memory use equal to the number of elements * the average size. When an element is added beyond the maximum size, the LRU element gets pushed into the DiskStore.

While we could have an expiry thread to expire elements periodically, it is far more efficient to only check when we need to. The tradeoff is higher average memory use.

The DiskStore's size is unbounded. The expiry thread keeps the disk store clean. There is hopefully less contention for the DiskStore's locks because commonly used values are in the MemoryStore. We mount our DiskStore on Linux using RAMFS so it is using OS memory. While we have more of this than the 2GB 32 bit process size limit it is still an expensive resource. The DiskStore thread keeps it under control.

If you are concerned about cpu utilisation and locking in the DiskStore, you can set the diskExpiryThreadIntervalSeconds to a high number - say 1 day. Or you can effectively turn it off by setting the diskExpiryThreadIntervalSeconds to a very large value.

21.8 What elements are mandatory in ehcache.xml?

The documentation has been updated with comprehensive coverage of the schema for ehcache and all elements and attributes, including whether they are mandatory. See the Declarative Configuration chapter.

21.9 Can I use ehcache as a memory cache only?

Yes. Just set the overflowToDisk attribute of cache to false.

21.10 Can I use ehcache as a disk cache only?

Yes. Set the maxElementsInMemory attribute of cache to 0.

This is strongly not recommended however. The minimum recommended value is 1. Performance is as much as 10 times higher when to one rather than 0. If not set to at least 1 a warning will be issued at Cache creation time.

21.11 Where is the source code? The source code is distributed in the root directory of the download.

It is called ehcache-x.x.zip. It is also available from SourceForge online or through cvs.

21.12 How do you get statistics on an Element without affecting them?

Use the Cache.getQuiet() method. It returns an Element without updating statistics.

21.13 How do you get WebSphere to work with ehcache?

It has been reported that IBM Websphere 5.1 running on IBM JDK1.4 requires commons-collection.jar in its classpath even though ehcache will not use it for JDK1.4 and JDK5.

21.14 Do you need to call CacheManager.getInstance().shutdown() when you finish with ehcache?

Yes, it is recommended. If the JVM keeps running after you stop using ehcache, you should call CacheManager.getInstance().shutdown() so that the threads are stopped and cache memory released back to the JVM. Calling shutdown also insures that your persistent disk stores get written to disk in a consistent state and will be usable the next time they are used.

If the CacheManager does not get shutdown it should not be a problem. There is a shutdown hook which calls the shutdown on JVM exit. This is explained in the documentation here.

21.15 Can you use ehcache after a CacheManager.shutdown()?

Yes. When you call CacheManager.shutdown() is sets the singleton in CacheManager to null. If you try an use a cache after this you will get a CacheException.

You need to call CacheManager.create(). It will create a brand new one good to go. Internally the CacheManager singleton gets set to the new one. So you can create and shutdown as many times as you like.

There is a test which expliciyly confirms this behaviour. See CacheManagerTest#testCreateShutdownCreate()

21.16 I have created a new cache and its status is STATUS_UNINITIALISED. How do I initialise it?

You need to add a newly created cache to a CacheManager before it gets intialised. Use code like the following:

```
CacheManager manager = CacheManager.create();
Cache myCache = new Cache("testDiskOnly", 0, true, false, 5, 2);
manager.addCache(myCache);
```

21.17 Is there a simple way to disable ehcache when testing?

Yes. There is a System Property based method of disabling ehcache. If disabled no elements will be added to a cache. Set the property "net.sf.ehcache.disabled=true" to disable ehcache.

This can easily be done using `-Dnet.sf.ehcache.disabled=true`> in the command line.

21.18 Is there a Maven bundle for ehcache?

Yes. http://www.ibiblio.org/maven/net.sf.ehcache/ for ehcache-1.2 and higher.

http://www.ibiblio.org/maven/ehcache/ for earlier versions.

21.19 How do I dynamically change Cache attributes at runtime?

You can't but you can achieve the same result as follows:

Cache cache = new Cache("test2", 1, true, true, 0, 0, true, 120, ...); cacheManager.addCache(cache);

See the JavaDoc for the full parameters, also reproduced here:

Having created the new cache, get a list of keys using cache.getKeys, then get each one and put it in the new cache. None of this will use much memory because the new cache element have values that reference the same data as the original cache. Then use cacheManager.removeCache("oldcachename") to remove the original cache.

21.20 I get net.sf.ehcache.distribution.RemoteCacheException: Error doing put to remote peerremote peer. Message was: Error unmarshaling return header; nested exception is: java.net.SocketTimeoutExce Read timed out. What does this mean.

It typically means you need to increase your socketTimeoutMillis. This is the amount of time a sender should wait for the call to the remote peer to complete. How long it takes depends on the network and the size of the Elements being replicated.

The configuration that controls this is the socketTimeoutMillis setting in cacheManagerPeerListenerFactory. 120000 seems to work well for most scenarios.

```
<cacheManagerPeerListenerFactory
        class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"
```

```
            properties="hostName=fully_qualified_hostname_or_ip,
                        port=40001,
                        socketTimeoutMillis=120000"/>
```

21.21 Should I use this directive when doing distributed caching? *cacheManagerEventListenerFactory class="" properties=""/*

No. It is unrelated. It is for listening to changes in your local CacheManager.

21.22 What is the minimum config to get distributed caching going?

The minimum configuration you need to get distributed going is:

```
    <cacheManagerPeerProviderFactory
            class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
            properties="peerDiscovery=automatic,
                        multicastGroupAddress=230.0.0.1,
                        multicastGroupPort=4446"/>


<cacheManagerPeerListenerFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"/>
```

and then at least one cache declaration with

```
<cacheEventListenerFactory class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>>>>>
```

in it. An example cache is:

```
    <cache name="sampleDistributedCache1"
           maxElementsInMemory="10"
           eternal="false"
           timeToIdleSeconds="100"
           timeToLiveSeconds="100"
           overflowToDisk="false">
        <cacheEventListenerFactory class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/
    </cache>
```

Each server in the cluster can have the same config.

21.23 How can I see if distributed caching is working?

You should see the listener port open on each server.

You can use the distributed debug tool to see what is going on. (See).

21.24 I get net.sf.ehcache.CacheException: Problem starting listener for RMICachePeer ... java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is: java.net.MalformedURLExcep no protocol: Files/Apache. What is going on?

This issue occurs to any RMI listener started on Tomcat, when Tomcat has spaces in its installation path.

It is is a JDK bug which can be worked around in Tomcat but is not. See http://archives.java.sun.com/cgi-bin/wa?A2=ind0205&L=rmi-users&P=797 and http://www.ontotext.com/kim/doc/sys-doc/faq-howto-bugs/known-bugs.html.

The workaround is to remove the spaces in your tomcat installation path.

21.25 Why can't I run multiple applications using ehcache on one machine?

Because of an RMI bug, in JDKs before JDK1.5 such as JDK1.4.2, ehcache is limited to one CacheManager operating in distributed mode per virtual machine. (The bug limits the number of RMI registries to one per virtual machine). Because this is the expected deployment configuration, however, there should be no practical effect. The tell tail error is `java.rmi.server.ExportException: internal error: ObjID already in use`

On JDK1.5 and higher it is possible to have multiple CacheManagers per VM each participating in the same or different clusters. Indeed the replication tests do this with 5 CacheManagers on the same VM all run from JUnit.

21.26 How many threads does ehcache use, and how much memory does that consume?

The amount of memory consumed per thread is determined by the Stack Size. This is set using -Xss. The amount varies by OS. It is 512KB for Linux. I tend to override the default and set it to 100kb.

The threads are created per cache as follows:

- DiskStore expiry thread - if DiskStore is used
- DiskStore spool thread - if DiskStore is used
- Replication thread - if asyncrhonous replication is configured.

  If you are not doing any of the above, no extra threads are created

Chapter 22

About the ehcache name and logo

Adam Murdoch (an all round top Java coder) came up with the name in a moment of inspiration while we were stuck on the SourceForge project create page. Ehcache is a palindrome. We thought the name was wicked cool.

The logo is similarly symmetrical, and is evocative of the diagram symbol for a doubly-linked list. The JDK1.4 LinkedHashMap, and Apache's LRUMap are a HashMap with a doubly-linked list running through all of its entries. These structures lie at the heart of ehcache.

Index