SetupAndConfiguration

整理：jiangzz 公众账号：jiangzz_wy

Introduction

For the most simple integration you just need to have a tomcat (6, 7 or 8) and a memcached or Redis installed (or s.th. supporting the memcached protocol). In your production environment you probably will have several tomcats and you should also have several memcached nodes available, on different pieces of hardware. Alternatively, you can store session data in Redis instead of memcached. You can use sticky sessions or non-sticky sessions, memcached-session-manager (msm) supports both operation modes.

The following description shows an example for a setup with sticky sessions, with two instances of tomcat and two instances of memcached installed.

Tomcat-1 (t1) will primarily store it's sessions in memcached-2 (m2) which is running on another machine (m2 is a regular node for t1). Only if m2 is not available, t1 will store it's sessions in memcached-1 (m1, m1 is the failoverNode for t1). With this configuration, sessions won't be lost when machine 1 (serving t1 and m1) crashes. The following really nice ASCII art shows this setup.

```
<t1>   <t2>
  . \ / .
  .  X  .
  . / \ .
<m1>   <m2>
```

So what needs to be done for this?

Decide which serialization strategy to use

There are several session serialization strategies available, as they are described on SerializationStrategies. The default strategy uses java serialization and is already provided by the memcached-session-manager jar. Other strategies are provided by separate jars, in the section below you'll see which jars are required for which strategy.

Configure tomcat

The configuration of tomcat requires two things: you need to drop some jars in your `$CATALINA_HOME/lib/` and `WEB-INF/lib/` directories and you have to configure the memcached session manager in the related `<Context>` element (e.g. in `META-INF/context.xml` inside the application files).

Add memcached-session-manager jars to tomcat

Independent of the chosen serialization strategy you always need the memcached-session-manager-${version}.jar and either memcached-session-manager-tc6-${version}.jar for tomcat6, memcached-session-manager-tc7-${version}.jar for tomcat7 (attention: tomcat 7.0.23+) or memcached-session-manager-tc8-${version}.jar for tomcat8.

If you're using memcached, you also need the spymemcached-2.11.1.jar.

If you're using couchbase, you need additionally these jars: couchbase-client-1.4.0.jar jettison-1.3.jar, commons-codec-1.5.jar, httpcore-4.3.jar, httpcore-nio-4.3.jar, netty-3.5.5.Final.jar.

If you're using Redis, you need the jedis-2.9.0.jar.

Please download the appropriate jars and put them in `$CATALINA_HOME/lib/`.

Add custom serializers to your webapp (optional)

If you want to use java's built in serialization nothing more has to be done. If you want to use a custom serialization strategy (e.g. because of better performance) this has to be deployed with your webapp so that they're available in `WEB-INF/lib/`.

As msm is available in maven central (under groupId `de.javakaffee.msm`) you can just pull it in using the dependency management of your build system. With maven you can use this dependency definition for the kryo-serializer:

```
<dependency>
    <groupId>de.javakaffee.msm</groupId>
    <artifactId>msm-kryo-serializer</artifactId>
    <version>1.9.7</version>
    <scope>runtime</scope>
</dependency>
```

For javolution the artifactId is `msm-javolution-serializer`, for xstream `msm-xstream-serializer` and for flexjson it's `msm-flexjson-serializer`.

If you're not using a dependency management based on maven repositories these are the jars you need for the different serializers:

- kryo-serializer: msm-kryo-serializer, kryo-serializers-0.34+, kryo-3.x, minlog, reflectasm, asm-5.x
- javolution-serializer: msm-javolution-serializer, javolution-5.4.3.1
- xstream-serializer: msm-xstream-serializer, xstream, xmlpull, xpp3_min
- flexjson-serializer: msm-flexjson-serializer, flexjson

Configure memcached-session-manager as `<Context>` Manager

Update the `<Context>` element (in `META-INF/context.xml` or what else you choose for context definition, please check the related tomcat documentation for this) so that it contains the `Manager` configuration for the memcached-session-manager, like in the examples below.

The following examples show configurations for sticky sessions and non-sticky sessions with memcached servers and for non-sticky sessions with membase. The examples with memcached servers assume that there are two memcacheds running, one on host1 and another one on host2. All sample configurations assume that you want to use kryo based serialization.

Example for sticky sessions + kryo

The following example shows the configuration of the first tomcat, assuming that it runs on host1, together with memcached "n1". The attribute `failoverNodes="n1"` tells msm to store sessions preferably in memcached "n2" and only store sessions in "n1" (running on the same host/machine) if no other memcached node (here only n2) is available (even if host1 goes down completely, the session is still available in memcached "n2" and could be served by the tomcat on host2). For the second tomcat (on host2) you just need to change the failover node to "n2", so that it prefers the memcached "n1". Everything else should be left unchanged.

```
<Context>
  ...
  <Manager className="de.javakaffee.web.msm.MemcachedBackupSessionManager"
    memcachedNodes="n1:host1.yourdomain.com:11211,n2:host2.yourdomain.com:11211"
    failoverNodes="n1"
    requestUriIgnorePattern=".*\.(ico|png|gif|jpg|css|js)$"
    transcoderFactoryClass="de.javakaffee.web.msm.serializer.kryo.KryoTranscoderFactory"
    />
</Context>
```

Example for non-sticky sessions + kryo

The following example shows a configuration for non-sticky sessions. In this case there's no need for `failoverNodes`, as sessions are served by all tomcats round-robin and they're not bound to a single tomcat. For non-sticky sessions the configuration (for both/all tomcats) would look like this:

```
<Context>
  ...
  <Manager className="de.javakaffee.web.msm.MemcachedBackupSessionManager"
    memcachedNodes="n1:host1.yourdomain.com:11211,n2:host2.yourdomain.com:11211"
    sticky="false"
    sessionBackupAsync="false"
    lockingMode="uriPattern:/path1|/path2"
    requestUriIgnorePattern=".*\.(ico|png|gif|jpg|css|js)$"
    transcoderFactoryClass="de.javakaffee.web.msm.serializer.kryo.KryoTranscoderFactory"
    />
</Context>
```

Example for non-sticky sessions + kryo + Redis

The following example shows a configuration which uses a Redis server at the URL "redis.example.com" for session storage for non-sticky sessions. Here the configuration (for both/all tomcats) would look like this:

```
<Context>
  ...
  <Manager className="de.javakaffee.web.msm.MemcachedBackupSessionManager"
    memcachedNodes="redis://redis.example.com"
    sticky="false"
    sessionBackupAsync="false"
    lockingMode="uriPattern:/path1|/path2"
    requestUriIgnorePattern=".*\.(ico|png|gif|jpg|css|js)$"
    transcoderFactoryClass="de.javakaffee.web.msm.serializer.kryo.KryoTranscoderFactory"
    />
</Context>
```

The built-in support for Redis does currently not allow connections to multiple Redis nodes, nor does it support the failoverNodes property. However, with Redis, automatic failover could be implemented directly in Redis by building a Redis cluster. For example, Amazon ElastiCache for Redis implements a failover mode which constantly replicates all data from the primary node to one or more slaves, then, if the master goes down, promotes another node to be the master and in the process changes the DNS entry of the primary node to point to the new master. MSM supports this mode by automatically reconnecting to the Redis server when the connection goes down, potentially picking up the new DNS settings and therefore connecting to the new master. In the future, support for Redis Sentinel or Redis Cluster may be added.

Example for non-sticky sessions with couchbase + kryo

To connect to a membase bucket "bucket1" the configuration would look like this:

```
<Context>
  ...
  <Manager className="de.javakaffee.web.msm.MemcachedBackupSessionManager"
    memcachedNodes="http://host1.yourdomain.com:8091/pools"
    username="bucket1"
    password="topsecret"
    memcachedProtocol="binary"
    sticky="false"
    sessionBackupAsync="false"
    requestUriIgnorePattern=".*\.(ico|png|gif|jpg|css|js)$"
    transcoderFactoryClass="de.javakaffee.web.msm.serializer.kryo.KryoTranscoderFactory"
    />
</Context>
```

Example for multiple contexts sharing the same session id

If you are running multiple webapps/contexts sharing the same session id (e.g. by having set `sessionCookiePath="/"` - or `emptySessionPath="true"` in tomcat 6) you must tell memcached session manager to add a prefix to the session id when storing a session in memcached. For this you can use the `storageKeyPrefix` attribute as shown by this example (see also the more detailed attribute description below):

```
<Context sessionCookiePath="/">
  ...
  <Manager className="de.javakaffee.web.msm.MemcachedBackupSessionManager"
    memcachedNodes="n1:host1.yourdomain.com:11211,n2:host2.yourdomain.com:11211"
    failoverNodes="n1"
    storageKeyPrefix="context"
    requestUriIgnorePattern=".*\.(ico|png|gif|jpg|css|js)$"
    transcoderFactoryClass="de.javakaffee.web.msm.serializer.kryo.KryoTranscoderFactory"
    />
</Context>
```

More details for all configuration attributes are provided in the section below.

After you have configured msm in the `Context/Manager` element, you can just start your application and sessions will be stored in the memcached nodes or in membase as configured. Now you should do some tests with a simulated tomcat failure, a restart of a memcached node etc. - have fun! :-)

Overview over memcached-session-manager configuration attributes

className (required)

> This should be set to `de.javakaffee.web.msm.MemcachedBackupSessionManager` to get sessions stored in memcached. However, since version 1.3.6 there's also a `de.javakaffee.web.msm.DummyMemcachedBackupSessionManager` that can be used for development purposes:

> it simply serializes sessions to a special in memory map and deserializes the serialized data at the next request when a session is requested (without really using this session) - just to see if deserialization is working for each request. Your application is still using sessions as if the memcached-session-manager (or DummyMemcachedBackupSessionManager) would not be existing. Session serialization/deserialization is done asynchronously. The configuration attributes `memcachedNodes` and `failoverNode` are not used to create a memcached client, so serialized session data will **not** be sent to memcached - and therefore no running memcacheds are required.

memcachedNodes (required)

> This attribute must contain all memcached nodes you have running, or the membase bucket uri(s). It should be the same for all tomcats.
> memcached nodes: Each memcached node is defined as `<id>:<host>:<port>`. Several definitions are separated by space or comma (e.g. `memcachedNodes="n1:app01:11211,n2:app02:11211"`). For a single node the `<id>` is optional so that it's allowed to set only `<host>:<port>` (e.g. `memcachedNodes="localhost:11211"`). Then the sessionId will be left unchanged (no node id appended). This option is useful for the usage with e.g. membase+moxi, where each tomcat knows only a single "memcached" node (moxi actually).
> membase bucket uris (since 1.6.0): For usage with membase it's possible to specify one or more membase bucket uris, e.g. `http://host1:8091/pools,http://host2:8091/pools`. Bucket name and password are specified via the attributes `username` and `password` (see below). Connecting to membase buckets requires the binary memcached protocol. Also remember to have the jettison.jar and netty.jar available in `CATALINA_HOME/lib/`.
> Redis uris (since 2.0.0): For usage with Redis you should use a "Redis URI" in the form `redis://example.com` or `redis://example.com:port`, or `rediss://example.com` or `rediss://example.com:port` if you want to use SSL to connect to Redis.

failoverNodes (optional, must not be used for non-sticky sessions)

> This attribute must contain the ids of the memcached nodes that must only be used for session backup when none of the other memcached nodes are available. Therefore, you should list those memcached nodes, that are running on the same machine as this tomcat. E.g. if you have tomcat1 and memcached1 (n1) running on host1, and tomcat2 and memcached2 (n2) on host2, then you set n1 as failover node for tomcat1, so that tomcat1 does only store sessions in memcached1 if memcached2 is not available (for tomcat2 failoverNodes would be set to n2). With this setting host1 could completely crash and sessions from tomcat1 would still be available.
>
> For non-sticky sessions failoverNodes must not be specified as a session is not tied to a single tomcat. For membase buckets this attribute should also be left out.
>
> Several memcached node ids are separated by space or comma.

username (since 1.6.0, optional)

> Specifies the username used for a membase bucket or SASL. If the *memcachedNodes* contains a membase bucket uri (or multiple) this is the bucket name. If the *memcachedNodes* contains memcached node definitions this is the username used for SASL authentication. Both require the binary memcached protocol.

password (since 1.6.0, optional)

> Specifies the password used for membase bucket or SASL authentication (can be left empty / omitted if the "default" membase bucket without a password shall be used).

memcachedProtocol (since 1.3, optional, default *text*)

> This attribute specifies the memcached protocol to use, one of *text* or *binary*.

sticky (since 1.4.0, optional, default *true*)

> Specifies if sticky or non-sticky sessions are used.

lockingMode (since 1.4.0, optional, for non-sticky sessions only, default *none*)

> Specifies the locking strategy for non-sticky sessions. Session locking is useful to prevent concurrent modifications and lost updates of the session in the case of parallel requests (tabbed browsing with long requests, ajax etc.). Session locking is done using memcached. Possible values for `lockingMode` are:
>
> - `none`: sessions are never locked
> - `all`: the session is locked when requested by the app until the end of the request
> - `auto`: readonly requests are detected, for them the session is not locked. For requests that are not classified as "readonly" the session is locked
> - `uriPattern:<regexp>`: the provided regular expression pattern is compared with the `requestURI` + "?" + `queryString` and the session is locked if they match.

requestUriIgnorePattern (optional)

> This attribute contains a regular expression for request URIs, that shall not trigger a session backup. If static resources like css, javascript, images etc. are delivered by the same tomcat and the same web application context these requests will also pass the memcached-session-manager. However, as these requests should not change anything in a http session, they should also not trigger a session backup. So you should check if any static resources are delivered by tomcat and in this case you should exclude them by using this attribute. The `requestUriIgnorePattern` must follow the java regex Pattern.

sessionBackupAsync (optional, default *true*)

> Specifies if the session shall be stored asynchronously in memcached. If this is *true*, the *backupThreadCount* setting is evaluated. If this is *false*, the timeout set via *sessionBackupTimeout* is evaluated.

backupThreadCount (since 1.3, optional, default *number-of-cpu-cores*)

> The number of threads that are used for asynchronous session backup (if *sessionBackupAsync="true"*). For the default value the number of available processors (cores) is used.

sessionBackupTimeout (optional, default *100*)

> The timeout in milliseconds after that a session backup is considered as beeing failed. This property is only evaluated if sessions are stored synchronously (set via *sessionBackupAsync*). The default value is *100* milliseconds.

operationTimeout (since 1.6.0, optional, default *1000*)

> The memcached operation timeout used at various places, e.g. used for the spymemcached ConnectionFactory.

storageKeyPrefix (since 1.8.0, optional, default *webappVersion*)

> Allows to configure a prefix that's added to the session id when a session is stored in memcached. This is useful if there are multiple webapps sharing the same session id, e.g. because contexts are configured with `sessionCookiePath="/"` (tomcat 7, or `emptySessionPath="true"` in tomcat 6)
>
> When tomcats parallel deployment is used there are also multiple webapps sharing the same session id. This is supported by default, because the default value (if `storageKeyPrefix` is not set explicitely) is `webappVersion`, so that the webapp version is used as prefix for the session id / key in memcached.
>
> The `storageKeyPrefix` attribute supports both a static prefix (`static:somePrefix`) and dynamic values (`host`, `context`, `webappVersion`), multiple values can be specified seperated by comma.
>
> Here are some examples that demonstrate which config would create which storage key (for a session id "foo" with context path "ctxt" and host "hst"):
>
> - `static:x` ⇒ `x_foo`
> - `host` ⇒ `hst_foo`
> - `host.hash` ⇒ `e93c085e_foo`
> - `context` ⇒ `ctxt_foo`
> - `context.hash` ⇒ `45e6345f_foo`
> - `host,context` ⇒ `hst:ctxt_foo`
> - `webappVersion` ⇒ `001_foo`
> - `host.hash,context.hash,webappVersion` ⇒ `e93c085e:45e6345f:001_foo`

sessionAttributeFilter (since 1.5.0, optional)

> A regular expression to control which session attributes are serialized to memcached. The regular expression is evaluated with session attribute names. E.g. `sessionAttributeFilter="^(userName|sessionHistory)$"` specifies that only "userName" and "sessionHistory" attributes are stored in memcached. Works independently from the chosen serialization strategy.

transcoderFactoryClass (since 1.1, optional, default *de.javakaffee.web.msm.JavaSerializationTranscoderFactory*)

> The class name of the factory that creates the transcoder to use for serializing/deserializing sessions to/from memcached. The specified class must implement `de.javakaffee.web.msm.TranscoderFactory` and provide a no-args constructor. Other `TranscoderFactory` implementations are available through other packages/jars like *msm-kryo-serializer*, *msm-xstream-serializer* and *msm-javolution-serializer* (as describe above), those are listed and compared on SerializationStrategies.
>
> Available `TranscoderFactory` implementations:
>
> - Java serialization: `de.javakaffee.web.msm.JavaSerializationTranscoderFactory`
> - Kryo based serialization: `de.javakaffee.web.msm.serializer.kryo.KryoTranscoderFactory`

- Javolution based serialization: `de.javakaffee.web.msm.serializer.javolution.JavolutionTranscoderFactory`
- XStream based serialization: `de.javakaffee.web.msm.serializer.xstream.XStreamTranscoderFactory`

copyCollectionsForSerialization (since 1.1, optional, default *false*)

> A boolean value that specifies, if iterating over collection elements shall be done on a copy of the collection or on the collection itself. This configuration property must be supported by the serialization strategy specified with *transcoderFactoryClass*. Which strategy supports this feature can be seen in the column *Copy Collections before serialization* in the list of available serialization strategies. This feature and its motivation is described more deeply at SerializationStrategies#Concurrent_modifications_of_collections.

customConverter (since 1.2, optional)

> Custom converter allow you to provide custom serialization of application specific types. Multiple custom converter class names are specified separated by comma (with optional space following the comma). Converter classes must be available in the classpath of the web application (place jars in `WEB-INF/lib`).
> This option is useful if you need some special serialization for a certain type or if reflection based serialization is just very verbose and you want to provide a more efficient serialization for a specific type.
> Custom converter must be supported by the serialization strategy specified with *transcoderFactoryClass*. Requirements regarding the specific custom converter classes depend on the actual serialization strategy, but a common requirement would be that they must provide a default/no-args constructor. For more details have a look at available serialization strategies.
>
> Available converter implementations:
>
> - Kryo (converters provided by msm-kryo-serializer jar)
>   - `de.javakaffee.web.msm.serializer.kryo.JodaDateTimeRegistration` : A more efficient serialization of Joda's DateTime with kryo.
>   - `de.javakaffee.web.msm.serializer.kryo.CGLibProxySerializerFactory` : serializes/deserializes CGLIB proxies.
>   - `de.javakaffee.web.msm.serializer.kryo.HibernateCollectionsSerializerFactory` : serializes/deserializes hibernate persistent collections (required if you store collections loaded by hibernate in your session).
>   - `de.javakaffee.web.msm.serializer.kryo.WicketSerializerFactory` : required if you're running a wicket web application.
>   - `de.javakaffee.web.msm.serializer.kryo.FacesLRUMapRegistration` : Needed if kryo serialization shall be used in combination with JSF2/Mojarra (see also issue #97).
>   - `de.javakaffee.web.msm.serializer.kryo.GrailsFlashScopeRegistration` : Support for grails flash scope (see also issue #107).
>   - `de.javakaffee.web.msm.serializer.kryo.SpringSecurityUserRegistration` : Support for Spring Security User class (see also issue #145).
> - Javolution (converters provided by msm-javolution-serializer jar)
>   - `de.javakaffee.web.msm.serializer.javolution.JodaDateTimeFormat` : A more efficient serialization of Joda's DateTime, see also issue #32.
>   - `de.javakaffee.web.msm.serializer.javolution.CGLibProxyFormat` : serializes/deserializes CGLIB proxies, see also issue #59.
>   - `de.javakaffee.web.msm.serializer.javolution.HibernateCollectionsXMLFormat` : serializes/deserializes hibernate persistent collections (required if you store collections loaded by hibernate in your session).

enableStatistics (since 1.2, optional, default *true*)

> A boolean value that specifies, if statistics shall be gathered. For more info see the JMXStatistics page.

enabled (since 1.4.0, optional, default *true*)

> Specifies if session storage in memcached is enabled or not, can also be changed at runtime via JMX. Only allowed in sticky mode.

Overview over available system properties (all optional)

msm.maxReconnectDelay (since 1.7, default *30*)

> The max reconnect delay for the spymemcached MemcachedClient, in seconds.

msm.nodeAvailabilityCacheTTL (since 1.7, default *1000*)

> The TTL for msm NodeAvailabilityCache entries, in millis.

msm.kryo.bufferSize.initial (since 1.1, default *102400*)

> The initial size in bytes of the kryo Output/Input when serializing a session. Is passed as `bufferSize` when creating a kryo Output.

msm.kryo.bufferSize.max (since 1.1, default *2048000*)

> The maximum size in bytes of the kryo Output when serializing a session - the maximum size in bytes that a serialized session can take. Is passed as `maxBufferSize` when creating a kryo Output.

msm.kryo.defaultSerializerFactory (since 1.8, default *de.javakaffee.web.msm.serializer.kryo.DefaultFieldSerializerFactory*)

> The factory to create the Serializer used by Kryo when Kryo#newDefaultSerializer is called. Must implement `de.javakaffee.web.msm.serializer.kryo.KryoDefaultSerializerFactory`.
>
> Currently available implementations:
>
> - `de.javakaffee.web.msm.serializer.kryo.DefaultFieldSerializerFactory`: Factory to create a FieldSerializer.
> - `de.javakaffee.web.msm.serializer.kryo.CompatibleFieldSerializerFactory`: Factory to create a CompatibleFieldSerializer, with limited support for forward and backward compatibility.

Configure logging

If you want to enable fine grained / debug logging you can add

```
de.javakaffee.web.msm.level=FINE
```

to `$CATALINA_HOME/conf/logging.properties`.

As the memcached-session-manager uses spymemcached also the logging hints of spymemcached might be interesting to you. A short summary how you can make spymemcached more silent:

1. Add the following to `$CATALINA_HOME/bin/catalina.sh`:

```
CATALINA_OPTS="-Dnet.spy.log.LoggerImpl=net.spy.memcached.compat.log.SunLogger"
```

1. Add this to `$CATALINA_HOME/conf/logging.properties`:

```
# A handler's log level threshold can be set using SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST or ALL
net.spy.memcached.level = WARNING
# To make only the MemcachedConnection less verbose:
#net.spy.memcached.MemcachedConnection.level = WARNING
```

More info about logging in tomcat can be found in the tomcat logging documentation.