[Hibernate的缓存](http://blog.csdn.net/accpsz/article/details/2458791)

**14.2.1    什么是缓存**

       缓存是介于应用程序和物理数据之间的，其作用是为了降低应用程序对物理数据访问的频次从而提高应用系统的性能。缓存思想的提出主要是因为对物理数据的访问效率要远远低于对内存的访问速度，因而采用了将部分物理数据存放于内存当中，这样可以有效地减少对物理数据的访问次数，从而提高系统的性能。

       缓存广泛地存在于我们所接触的各种应用系统中，例如数据库系统、Windows操作系统等，在进行物理数据的访问时无一例外地都使用了缓存机制来提高操作的性能。

       缓存内的数据是对物理数据的复制，因此一个缓存系统所应该包括的最基本的功能是数据的缓存和读取，同时在使用缓存的时候还要考虑缓存中的数据与物理数据的同步，也就是要保持两者是一致的。

       缓存要求对数据的读写速度很高，因此，一般情况下会选用内存作为存储的介质。但如果内存有限，并且缓存中存放的数据量非常大时，也会用硬盘作为缓存介质。缓存的实现不仅仅要考虑存储的介质，还要考虑到管理缓存的并发访问和缓存数据的生命周期。

       为了提高系统的性能，Hibernate也使用了缓存的机制。在Hibernate框架中，主要包括以下两个方面的缓存：一级缓存和二级缓存（包含查询缓存）。Hibernate中缓存的作用主要表现在以下两个方面：

●   通过主键（ID）加载数据的时候

●   延迟加载中

**14.2.2    一级缓存**

       Hibernate的一级缓存是由Session提供的，因此它只存在于Session的生命周期中，也就是当Session关闭的时候该Session所管理的一级缓存也会立即被清除。

       Hibernate的一级缓存是Session所内置的，不能被卸载，也不能进行任何配置。

       一级缓存采用的是key-value的Map方式来实现的，在缓存实体对象时，对象的主关键字ID是Map的key，实体对象就是对应的值。所以说，一级缓存是以实体对象为单位进行存储的，在访问的时候使用的是主关键字ID。

       虽然，Hibernate对一级缓存使用的是自动维护的功能，没有提供任何配置功能，但是可以通过Session中所提供的方法来对一级缓存的管理进行手工干预。Session中所提供的干预方法包括以下两种。

●   evict() ：用于将某个对象从Session的一级缓存中清除。

●   clear() ：用于将一级缓存中的对象全部清除。

       在进行大批量数据一次性更新的时候，会占用非常多的内存来缓存被更新的对象。这时就应该阶段性地调用clear()方法来清空一级缓存中的对象，控制一级缓存的大小，以避免产生内存溢出的情况。具体的实现方法如清单14.8所示。

       清单14.8    大批量更新时缓存的处理方法

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {

    Customer customer = new Customer(……);

    session.save(customer);

    if ( i % 20 == 0 ) {

        //将本批插入的对象立即写入数据库并释放内存

        session.flush();

        session.clear();

    }

}

tx.commit();

session.close();

**14.2.3    二级缓存**

       与Session相对的是，SessionFactory也提供了相应的缓存机制。SessionFactory缓存可以依据功能和目的的不同而划分为内置缓存和外置缓存。

       SessionFactory的内置缓存中存放了映射元数据和预定义SQL语句，映射元数据是映射文件中数据的副本，而预定义SQL语句是在Hibernate初始化阶段根据映射元数据推导出来的。SessionFactory的内置缓存是只读的，应用程序不能修改缓存中的映射元数据和预定义SQL语句，因此SessionFactory不需要进行内置缓存与映射文件的同步。

       SessionFactory的外置缓存是一个可配置的插件。在默认情况下，SessionFactory不会启用这个插件。外置缓存的数据是数据库数据的副本，外置缓存的介质可以是内存或者硬盘。SessionFactory的外置缓存也被称为Hibernate的二级缓存。

       Hibernate的二级缓存的实现原理与一级缓存是一样的，也是通过以ID为key的Map来实现对对象的缓存。

       由于Hibernate的二级缓存是作用在SessionFactory范围内的，因而它比一级缓存的范围更广，可以被所有的Session对象所共享。

**14.2.3.1    二级缓存的工作内容**

       Hibernate的二级缓存同一级缓存一样，也是针对对象ID来进行缓存。所以说，二级缓存的作用范围是针对根据ID获得对象的查询。

       二级缓存的工作可以概括为以下几个部分：

●   在执行各种条件查询时，如果所获得的结果集为实体对象的集合，那么就会把所有的数据对象根据ID放入到二级缓存中。

●   当Hibernate根据ID访问数据对象的时候，首先会从Session一级缓存中查找，如果查不到并且配置了二级缓存，那么会从二级缓存中查找，如果还查不到，就会查询数据库，把结果按照ID放入到缓存中。

●   **删除、更新、增加数据的时候，同时更新缓存。**

**14.2.3.2    二级缓存的适用范围**

       Hibernate的二级缓存作为一个可插入的组件在使用的时候也是可以进行配置的，但并不是所有的对象都适合放在二级缓存中。

       在通常情况下会将具有以下特征的数据放入到二级缓存中：

●   **很少被修改的数据。**

●   不是很重要的数据，允许出现偶尔并发的数据。

●   不会被并发访问的数据。

●   参考数据。

       而对于具有以下特征的数据则不适合放在二级缓存中：

●   经常被修改的数据。

●   财务数据，绝对不允许出现并发。

●   与其他应用共享的数据。

       在这里特别要注意的是对放入缓存中的数据不能有第三方的应用对数据进行更改（其中也包括在自己程序中使用其他方式进行数据的修改，例如，JDBC），因为那样Hibernate将不会知道数据已经被修改，也就无法保证缓存中的数据与数据库中数据的一致性。

**14.2.3.3    二级缓存组件**

       在默认情况下，Hibernate会使用EHCache作为二级缓存组件。但是，可以通过设置hibernate.cache.provider\_class属性，指定其他的缓存策略，该缓存策略必须实现org.hibernate.cache.CacheProvider接口。

       通过实现org.hibernate.cache.CacheProvider接口可以提供对不同二级缓存组件的支持。

       Hibernate内置支持的二级缓存组件如表14.1所示。

表14.1    Hibernate所支持的二级缓存组件

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 组件 | Provider类 | 类型 | 集群 | 查询缓存 |
| Hashtable | org.hibernate.cache.HashtableCacheProvider | 内存 | 不支持 | 支持 |
| EHCache | org.hibernate.cache.EhCacheProvider | 内存，硬盘 | 不支持 | 支持 |
| OSCache | org.hibernate.cache.OSCacheProvider | 内存，硬盘 | 不支持 | 支持 |
| SwarmCache | org.hibernate.cache.SwarmCacheProvider | 集群 | 支持 | 不支持 |
| JBoss TreeCache | org.hibernate.cache.TreeCacheProvider | 集群 | 支持 | 支持 |

       Hibernate已经不再提供对JCS（Java Caching System）组件的支持了。

**14.2.3.4    二级缓存的配置**

       在使用Hibernate的二级缓存时，对于每个需要使用二级缓存的对象都需要进行相应的配置工作。也就是说，只有配置了使用二级缓存的对象才会被放置在二级缓存中。二级缓存是通过<cache>元素来进行配置的。

       <cache>元素的属性定义说明如下所示：

       <cache

           usage="transactional|read-write|nonstrict-read-write|read-only"               (1)

           region="RegionName"                                                                                                   (2)

           include="all|non-lazy"                                                                                                   (3)

/>

       <cache>元素的属性说明如表14.2所示。

表14.2    <cache>元素的属性说明

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 属性 | 含义和作用 | 必须 | 默认值 |
| (1) | usage | 指定缓存策略，可选的策略包括：transactional，read-write，nonstrict-read-write或read-only | Y |  |
| (2) | region | 指定二级缓存区域名 | N |  |
| (3) | include | 指定是否缓存延迟加载的对象。all，表示缓存所有对象；non-lazy，表示不缓存延迟加载的对象 | N | all |

**14.2.3.5    二级缓存的策略**

       当多个并发的事务同时访问持久化层的缓存中的相同数据时，会引起并发问题，必须采用必要的事务隔离措施。

       在进程范围或集群范围的缓存，即第二级缓存，会出现并发问题。因此可以设定以下4种类型的并发访问策略，每一种策略对应一种事务隔离级别。

●   只读缓存（read-only）

       如果应用程序需要读取一个持久化类的实例，但是并不打算修改它们，可以使用read-only缓存。这是最简单，也是实用性最好的策略。

对于从来不会修改的数据，如参考数据，可以使用这种并发访问策略。

●   读/写缓存（read-write）

       如果应用程序需要更新数据，可能read-write缓存比较合适。如果需要序列化事务隔离级别，那么就不能使用这种缓存策略。

对于经常被读但很少修改的数据，可以采用这种隔离类型，因为它可以防止脏读这类的并发问题。

●   不严格的读/写缓存（nonstrict-read-write）

       如果程序偶尔需要更新数据（也就是说，出现两个事务同时更新同一个条目的现象很不常见），也不需要十分严格的事务隔离，可能适用nonstrict-read-write缓存。

对于极少被修改，并且允许偶尔脏读的数据，可以采用这种并发访问策略。

●   事务缓存（transactional）

       transactional缓存策略提供了对全事务的缓存，仅仅在受管理环境中使用。它提供了Repeatable Read事务隔离级别。对于经常被读但很少修改的数据，可以采用这种隔离类型，因为它可以防止脏读和不可重复读这类的并发问题。

       在上面所介绍的隔离级别中，事务型并发访问策略的隔离级别最高，然后依次是读/写型和不严格读写型，只读型的隔离级别最低。事务的隔离级别越高，并发性能就越低。

**14.2.3.6    在开发中使用二级缓存**

       在这一部分中，将细致地介绍如何在Hibernate中使用二级缓存。在这里所使用的二级缓存组件为EHCache。

       关于EHCache的详细信息请参考http://ehcache.sourceforge.net上的内容。

       在Hibernate中使用二级缓存需要经历以下步骤：

●   在Hibernate配置文件（通常为hibernate.cfg.xml）中，设置二级缓存的提供者类。

●   配置EHCache的基本参数。

●   在需要进行缓存的实体对象的映射文件中配置缓存的策略。

       下面就来逐步演示一下如何在开发中使用Hibernate的二级缓存。

**修改Hibernate的配置文件**

       在使用Hibernate的二级缓存时，需要在Hibernate的配置文件中指定缓存提供者对象，以便于Hibernate可以通过其实现对数据的缓存处理。

       在这里需要设置的参数是hibernate.cache.provider\_class，在使用EHCache时，需要将其值设置为org.hibernate.cache.EhCacheProvider。具体要增加的配置如下所示：

       <property name="hibernate.cache.provider\_class">

              org.hibernate.cache.EhCacheProvider</property>

       Hibernate配置文件的详细内容请参考配套光盘中的hibernate/src/cn/hxex/ hibernate/cache/hibernate.cfg.xml文件。

**增加EHCache配置参数**

       在默认情况下，EHCache会到classpath所指定的路径中寻找ehcache.xml文件来作为EHCache的配置文件。

       在配置文件中，包含了EHCache进行缓存管理时的一些基本的参数。具体的配置方法如清单14.9所示。

       清单14.9    EHCache的配置

<?xml version="1.0" encoding="UTF-8"?>

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

       xsi:noNamespaceSchemaLocation="ehcache.xsd">

       <diskStore path="java.io.tmpdir" />

       <defaultCache

              maxElementsInMemory="10000"

              eternal="false"

              timeToIdleSeconds="120"

              timeToLiveSeconds="120"

              overflowToDisk="true"

              diskPersistent="false"

              diskExpiryThreadIntervalSeconds="120"

              memoryStoreEvictionPolicy="LRU" />

</ehcache>

       在这里只是使用EHCache所提供的默认配置文件进行了EHCache的基本配置，对于这些参数的详细含义请参考其官方网站（http://ehcache.sourceforge.net/）中的资料。在实际的开发中，应该依据自己的具体情况来设置这些参数的值。

**开发实体对象**

       这里所使用的是一个非常简单的User对象，它只包含了ID，name和age三个属性，具体的实现方法请参见配套光盘中的hibernate/src/cn/hxex/cache/User.java文件。

**配置映射文件**

       映射文件的配置与不使用二级缓存的Java对象的区别就在于需要增加前面所介绍的<cache>元素来配置此对象的缓存策略。在这里所使用的缓存策略为“read-write”。所以，应该在映射文件中增加如下的配置：

       <cache usage="read-write"/>

       映射文件的详细配置请参考配套光盘中的hibernate/src/cn/hxex/cache/User.hbm.xml文件。

**测试主程序**

       在这里的测试主程序采用了多线程的运行方式，以模拟在不同Session的情况下是否真的可以避免查询的重复进行。

       在这个测试程序中，所做的工作就是依据ID来得到对应的实体对象，并将其输出。然后通过多次运行此程序，来检查后面的运行是否进行了数据库的操作。

       测试主程序的主要测试方法的实现如清单14.10所示。

       清单14.10    测试主程序的实现

……

    public void run() {

           SessionFactory sf = CacheMain.getSessionFactory();

           Session session = sf.getCurrentSession();

                    session.beginTransaction();

           User user = (User)session.get( User.class, "1" );

           System.out.println( user );

           session.getTransaction().commit();

}

       public static void main(String[] args) {

              CacheMain main1 = new CacheMain();

              main1.start();

              CacheMain main2 = new CacheMain();

              main2.start();

       }

}

**运行测试程序**

       在运行测试程序之前，需要先手动地向数据库中增加一条记录。该记录的ID值为1，可以采用下面的SQL语句。

       INSERT INTO userinfo(userId, name, age) VALUES( '1', 'galaxy', 32 );

       接下来在运行测试主程序时，应该看到类似下面的输出：

Hibernate: select user0\_.userId as userId0\_0\_, user0\_.name as name0\_0\_, user0\_.age as age0\_0\_ from USERINFO user0\_ where user0\_.userId=?

ID:  1

Namge:   galaxy

Age:       32

ID:  1

Namge:   galaxy

Age:       32

       通过上面的结果可以看到，每个运行的进程都输出了User对象的信息，但在运行中只进行了一次数据库读取操作，这说明第二次User对象的获得是从缓存中抓取的，而没有进行数据库的查询操作。

**14.2.3.7    查询缓存**

       查询缓存是专门针对各种查询操作进行缓存。查询缓存会在整个SessionFactory的生命周期中起作用，存储的方式也是采用key-value的形式来进行存储的。

       查询缓存中的key是根据查询的语句、查询的条件、查询的参数和查询的页数等信息组成的。而数据的存储则会使用两种方式，使用SELECT语句只查询实体对象的某些列或者某些实体对象列的组合时，会直接缓存整个结果集。而对于查询结果为某个实体对象集合的情况则只会缓存实体对象的ID值，以达到缓存空间可以共用，节省空间的目的。

       在使用查询缓存时，除了需要设置hibernate.cache.provider\_class参数来启动二级缓存外，还需要通过hibernate.cache.use\_query\_cache参数来启动对查询缓存的支持。

       另外需要注意的是，查询缓存是在执行查询语句的时候指定缓存的方式以及是否需要对查询的结果进行缓存。

       下面就来了解一下查询缓存的使用方法及作用。

**修改Hibernate配置文件**

       首先需要修改Hibernate的配置文件，增加hibernate.cache.use\_query\_cache参数的配置。配置方法如下：

       <property name="hibernate.cache.use\_query\_cache">true</property>

       Hibernate配置文件的详细内容请参考配套光盘中的hibernate/src/cn/hxex/ hibernate/cache/hibernate.cfg.xml文件。

**编写主测试程序**

       由于这是在前面二级缓存例子的基础上来开发的，所以，对于EHCache的配置以及视图对象的开发和映射文件的配置工作就都不需要再重新进行了。下面就来看一下主测试程序的实现方法，如清单14.11所示。

       清单14.11    主程序的实现

……

    public void run() {

           SessionFactory sf = QueryCacheMain.getSessionFactory();

           Session session = sf.getCurrentSession();

           session.beginTransaction();

           Query query = session.createQuery( "from User" );

           Iterator it = query.setCacheable( true ).list().iterator();

           while( it.hasNext() ) {

                  System.out.println( it.next() );

           }

           User user = (User)session.get( User.class, "1" );

           System.out.println( user );

           session.getTransaction().commit();

    }

       public static void main(String[] args) {

              QueryCacheMain main1 = new QueryCacheMain();

              main1.start();

              try {

                     Thread.sleep( 2000 );

              } catch (InterruptedException e) {

                     e.printStackTrace();

              }

              QueryCacheMain main2 = new QueryCacheMain();

              main2.start();

       }

}

       主程序在实现的时候采用了多线程的方式来运行。首先将“from User”查询结果进行缓存，然后再通过ID取得对象来检查是否对对象进行了缓存。另外，多个线程的执行可以看出对于进行了缓存的查询是不会执行第二次的。

**运行测试主程序**

       接着就来运行测试主程序，其输出结果应该如下所示：

Hibernate: select user0\_.userId as userId0\_, user0\_.name as name0\_, user0\_.age as age0\_ from USERINFO user0\_

ID:  1

Namge:   galaxy

Age:       32

ID:  1

Namge:   galaxy

Age:       32

ID:  1

Namge:   galaxy

Age:       32

ID:  1

Namge:   galaxy

Age:       32

       通过上面的执行结果可以看到，在两个线程执行中，只执行了一个SQL查询语句。这是因为根据ID所要获取的对象在前面的查询中已经得到了，并进行了缓存，所以没有再次执行查询语句。

**14.2.4    Hibernate查询方法与缓存的关系**

       在前面介绍了Hibernate的缓存技术以及基本的用法，在这里就具体的Hibernate所提供的查询方法与Hibernate缓存之间的关系做一个简单的总结。

       在开发中，通常是通过两种方式来执行对数据库的查询操作的。一种方式是通过ID来获得单独的Java对象，另一种方式是通过HQL语句来执行对数据库的查询操作。下面就分别结合这两种查询方式来说明一下缓存的作用。

       通过ID来获得Java对象可以直接使用Session对象的load()或者get()方法，这两种方式的区别就在于对缓存的使用上。

●   load()方法

       在使用了二级缓存的情况下，使用load()方法会在二级缓存中查找指定的对象是否存在。

在执行load()方法时，Hibernate首先从当前Session的一级缓存中获取ID对应的值，在获取不到的情况下，将根据该对象是否配置了二级缓存来做相应的处理。

       如配置了二级缓存，则从二级缓存中获取ID对应的值，如仍然获取不到则还需要根据是否配置了延迟加载来决定如何执行，如未配置延迟加载则从数据库中直接获取。在从数据库获取到数据的情况下，Hibernate会相应地填充一级缓存和二级缓存，如配置了延迟加载则直接返回一个代理类，只有在触发代理类的调用时才进行数据库的查询操作。

       在Session一直打开的情况下，并在该对象具有单向关联维护的时候，需要使用类似Session.clear()，Session.evict()的方法来强制刷新一级缓存。

●   get()方法

       get()方法与load()方法的区别就在于不会查找二级缓存。在当前Session的一级缓存中获取不到指定的对象时，会直接执行查询语句从数据库中获得所需要的数据。

       在Hibernate中，可以通过HQL来执行对数据库的查询操作。具体的查询是由Query对象的list()和iterator()方法来执行的。这两个方法在执行查询时的处理方法存在着一定的差别，在开发中应该依据具体的情况来选择合适的方法。

●   list()方法

       在执行Query的list()方法时，Hibernate的做法是首先检查是否配置了查询缓存，如配置了则从查询缓存中寻找是否已经对该查询进行了缓存，如获取不到则从数据库中进行获取。从数据库中获取到后，Hibernate将会相应地填充一级、二级和查询缓存。如获取到的为直接的结果集，则直接返回，如获取到的为一些ID的值，则再根据ID获取相应的值（Session.load()），最后形成结果集返回。可以看到，在这样的情况下，list()方法也是有可能造成N次查询的。

       查询缓存在数据发生任何变化的情况下都会被自动清空。

●   iterator()方法

       Query的iterator()方法处理查询的方式与list()方法是不同的，它首先会使用查询语句得到ID值的列表，然后再使用Session的load()方法得到所需要的对象的值。

       在获取数据的时候，应该依据这4种获取数据方式的特点来选择合适的方法。在开发中可以通过设置show\_sql选项来输出Hibernate所执行的SQL语句，以此来了解Hibernate是如何操作数据库的

**14.3    Hibernate的性能优化**

       Hibernate是对JDBC的轻量级封装，因此在很多情况下Hibernate的性能比直接使用JDBC存取数据库要低。然而，通过正确的方法和策略，在使用Hibernate的时候还是可以非常接近直接使用JDBC时的效率的，并且，在有些情况下还有可能高于使用JDBC时的执行效率。

       在进行Hibernate性能优化时，需要从以下几个方面进行考虑：

●   数据库设计调整。

●   HQL优化。

●   API的正确使用（如根据不同的业务类型选用不同的集合及查询API）。

●   主配置参数（日志、查询缓存、fetch\_size、batch\_size等）。

●   映射文件优化（ID生成策略、二级缓存、延迟加载、关联优化）。

●   一级缓存的管理。

●   针对二级缓存，还有许多特有的策略。

●   事务控制策略。

       数据的查询性能往往是影响一个应用系统性能的主要因素。对查询性能的影响会涉及到系统软件开发的各个阶段，例如，良好的设计、正确的查询方法、适当的缓存都有利于系统性能的提升。

       系统性能的提升设计到系统中的各个方面，是一个相互平衡的过程，需要在应用的各个阶段都要考虑。并且在开发、运行的过程中要不断地调整和优化才能逐步提升系统的性能。