[**《深入理解mybatis原理》 MyBatis的一级缓存实现详解 及使用注意事项**](http://blog.csdn.net/luanlouis/article/details/41280959)

标签： [MyBatis原理](http://www.csdn.net/tag/MyBatis%e5%8e%9f%e7%90%86)[MyBatis](http://www.csdn.net/tag/MyBatis)[ORM](http://www.csdn.net/tag/ORM)[缓存机制](http://www.csdn.net/tag/%e7%bc%93%e5%ad%98%e6%9c%ba%e5%88%b6)[源码](http://www.csdn.net/tag/%e6%ba%90%e7%a0%81)

2014-11-21 23:01 23308人阅读 [评论](http://blog.csdn.net/luanlouis/article/details/41280959#comments)(26) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/luanlouis/article/details/41280959#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

MyBatis（8） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg MyBatis教程（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/luanlouis/article/details/41280959)

**0.写在前面**

        MyBatis是一个简单，小巧但功能非常强大的ORM开源框架，它的功能强大也体现在它的缓存机制上。MyBatis提供了一级缓存、二级缓存 这两个缓存机制，能够很好地处理和维护缓存，以提高系统的性能。本文的目的则是向读者详细介绍MyBatis的一级缓存，深入源码，解析MyBatis一级缓存的实现原理，并且针对一级缓存的特点提出了在实际使用过程中应该注意的事项。

读完本文，你将会学到：

**1、什么是一级缓存？为什么使用一级缓存？**

**2、MyBatis的一级缓存是怎样组织的？（即SqlSession对象中的缓存是怎样组织的？）**

**3、一级缓存的生命周期有多长？**

**4、Cache接口的设计以及CacheKey的定义**

**5、一级缓存的性能分析以及应该注意的事项**

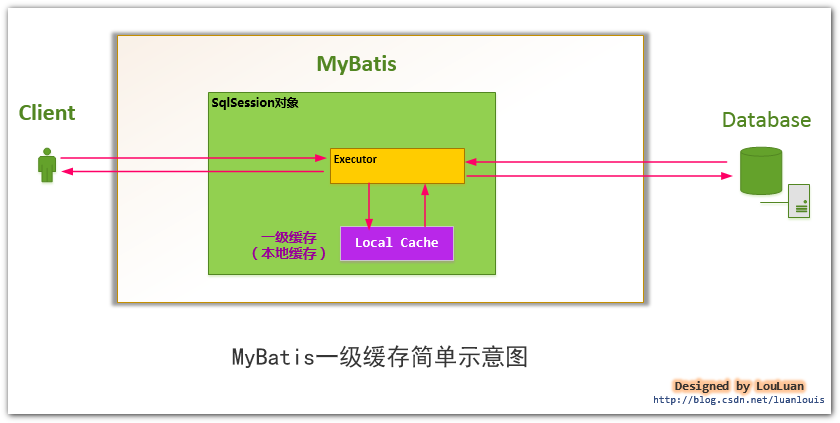
**1. 什么是一级缓存？ 为什么使用一级缓存？**

     每当我们使用**MyBatis**开启一次和数据库的会话，**MyBatis**会创建出**一个SqlSession对象表示一次数据库会话**。

      在对数据库的一次会话中，我们有可能会反复地执行完全相同的查询语句，如果不采取一些措施的话，每一次查询都会查询一次数据库,而我们在极短的时间内做了完全相同的查询，那么它们的结果极有可能完全相同，由于查询一次数据库的代价很大，这有可能造成很大的资源浪费。

      为了解决这一问题，减少资源的浪费，**MyBatis会在表示会话的SqlSession对象中建立一个简单的缓存，将每次查询到的结果结果缓存起来，当下次查询的时候，如果判断先前有个完全一样的查询，会直接从缓存中直接将结果取出，返回给用户，不需要再进行一次数据库查询了。**

**如下图所示，MyBatis会在一次会话的表示----一个SqlSession对象中创建一个本地缓存(local cache)，对于每一次查询，都会尝试根据查询的条件去本地缓存中查找是否在缓存中，如果在缓存中，就直接从缓存中取出，然后返回给用户；否则，从数据库读取数据，将查询结果存入缓存并返回给用户。**

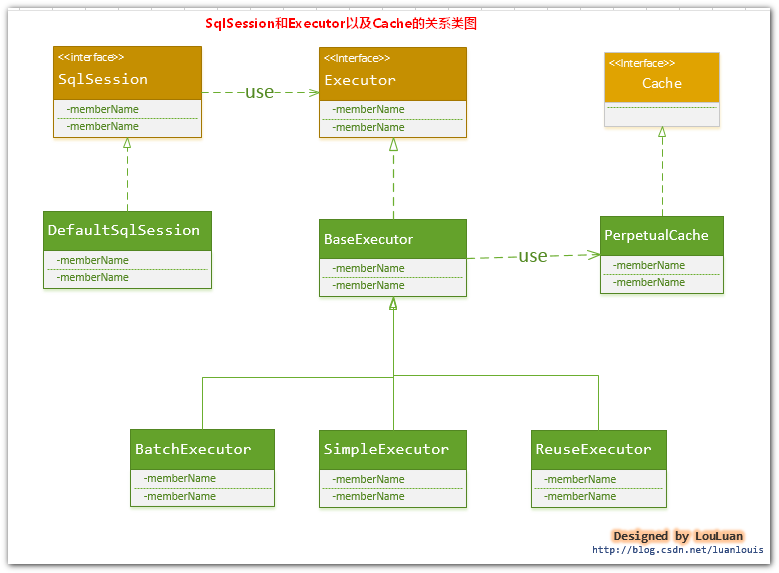
****

**对于会话（Session）级别的数据缓存，我们称之为一级数据缓存，简称一级缓存。**

**2. MyBatis中的一级缓存是怎样组织的？（即SqlSession中的缓存是怎样组织的？）**

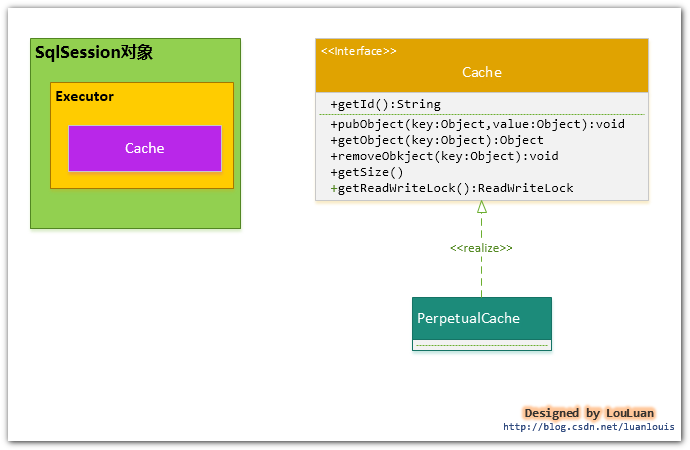
      由于**MyBatis**使用**SqlSession**对象表示一次数据库的会话，那么，对于会话级别的一级缓存也应该是在SqlSession中控制的。

      实际上, **MyBatis**只是一个**MyBatis**对外的接口，**SqlSession**将它的工作交给了**Executor**执行器这个角色来完成，负责完成对数据库的各种操作。当创建了一个**SqlSession**对象时，**MyBatis**会为这个**SqlSession**对象创建一个新的**Executor**执行器，而缓存信息就被维护在这个**Executor**执行器中，**MyBatis**将缓存和对缓存相关的操作封装成了Cache接口中。**SqlSession**、**Executor**、**Cache**之间的关系如下列类图所示：



      如上述的类图所示，**Executor**接口的实现类**BaseExecutor**中拥有一个**Cache**接口的实现类**PerpetualCache**，则对于**BaseExecutor**对象而言，它将使用**PerpetualCache**对象维护缓存。

综上，**SqlSession**对象、**Executor**对象、**Cache**对象之间的关系如下图所示：



由于**Session**级别的一级缓存实际上就是使用**PerpetualCache**维护的，那么**PerpetualCache**是怎样实现的呢？

**PerpetualCache**实现原理其实很简单，其内部就是通过一个简单的**HashMap<k,v>** 来实现的，没有其他的任何限制。如下是**PerpetualCache**的实现代码：

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/41280959) [copy](http://blog.csdn.net/luanlouis/article/details/41280959)

[print?](http://blog.csdn.net/luanlouis/article/details/41280959)

1. **package** org.apache.ibatis.cache.impl;
3. **import** java.util.HashMap;
4. **import** java.util.Map;
5. **import** java.util.concurrent.locks.ReadWriteLock;
7. **import** org.apache.ibatis.cache.Cache;
8. **import** org.apache.ibatis.cache.CacheException;
10. /\*\*
11. \* 使用简单的HashMap来维护缓存
12. \* @author Clinton Begin
13. \*/
14. **public** **class** PerpetualCache **implements** Cache {
16. **private** String id;
18. **private** Map<Object, Object> cache = **new** HashMap<Object, Object>();
20. **public** PerpetualCache(String id) {
21. **this**.id = id;
22. }
24. **public** String getId() {
25. **return** id;
26. }
28. **public** **int** getSize() {
29. **return** cache.size();
30. }
32. **public** **void** putObject(Object key, Object value) {
33. cache.put(key, value);
34. }
36. **public** Object getObject(Object key) {
37. **return** cache.get(key);
38. }
40. **public** Object removeObject(Object key) {
41. **return** cache.remove(key);
42. }
44. **public** **void** clear() {
45. cache.clear();
46. }
48. **public** ReadWriteLock getReadWriteLock() {
49. **return** **null**;
50. }
52. **public** **boolean** equals(Object o) {
53. **if** (getId() == **null**) **throw** **new** CacheException("Cache instances require an ID.");
54. **if** (**this** == o) **return** **true**;
55. **if** (!(o **instanceof** Cache)) **return** **false**;
57. Cache otherCache = (Cache) o;
58. **return** getId().equals(otherCache.getId());
59. }
61. **public** **int** hashCode() {
62. **if** (getId() == **null**) **throw** **new** CacheException("Cache instances require an ID.");
63. **return** getId().hashCode();
64. }
66. }

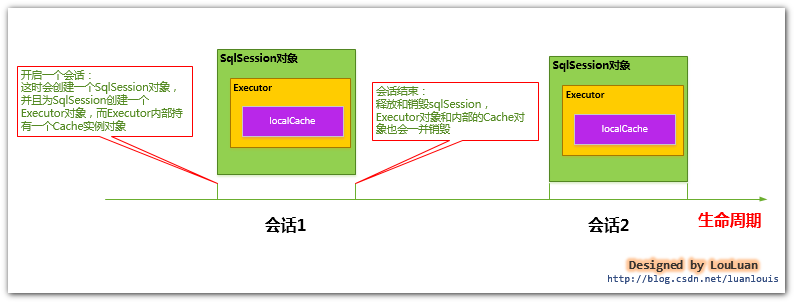
**3.一级缓存的生命周期有多长？**

**a. MyBatis**在开启一个数据库会话时，会 创建一个新的**SqlSession**对象，**SqlSession**对象中会有一个新的**Executor**对象，**Executor**对象中持有一个新的**PerpetualCache**对象；当会话结束时，**SqlSession**对象及其内部的**Executor**对象还有**PerpetualCache**对象也一并释放掉。

**b.**如果**SqlSession**调用了**close()**方法，会释放掉一级缓存**PerpetualCache**对象，一级缓存将不可用；

**c.**如果**SqlSession**调用了**clearCache()**，会清空**PerpetualCache**对象中的数据，但是该对象仍可使用；

**d.SqlSession**中执行了任何一个**update**操作(**update()、delete()、insert()**) ，都会清空**PerpetualCache**对象的数据，但是该对象可以继续使用；



**4. SqlSession 一级缓存的工作流程：**

1.对于某个查询，根据**statementId,params,rowBounds**来构建一个**key**值，根据这个**key**值去缓存**Cache**中取出对应的**key**值存储的缓存结果；

2. 判断从**Cache**中根据特定的**key**值取的数据数据是否为空，即是否命中；

3. 如果命中，则直接将缓存结果返回；

4. 如果没命中：

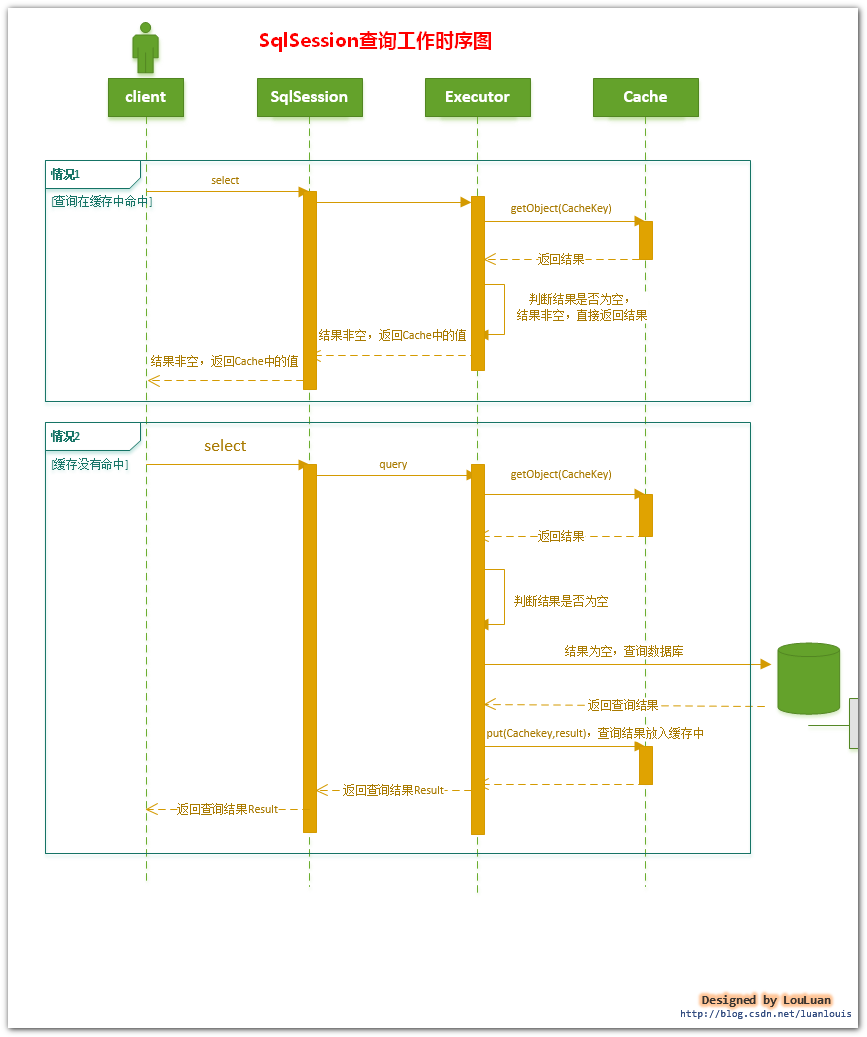
        4.1  去数据库中查询数据，得到查询结果；

        4.2  将key和查询到的结果分别作为**key**,**value**对存储到**Cache**中；

        4.3. 将查询结果返回；

5. 结束。

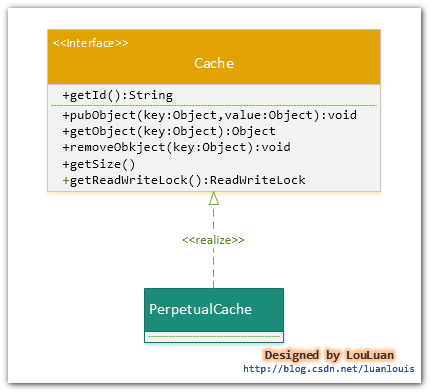
***[关于上述工作过程中 key值的构建，我们将在第下一节中重点探讨，这也是MyBatis缓存机制中非常重要的一个概念。]***



**5. Cache接口的设计以及CacheKey的定义（非常重要）**

      如下图所示，**MyBatis**定义了一个**org.apache.ibatis.cache.Cache**接口作为其**Cache**提供者的**SPI(Service Provider Interface)** ，所有的**MyBatis**内部的**Cache**缓存，都应该实现这一接口。**MyBatis**定义了一个**PerpetualCache**实现类实现了**Cache**接口，**实际上，在SqlSession对象里的Executor 对象内维护的Cache类型实例对象，就是PerpetualCache子类创建的**。

    （**MyBatis**内部还有很多**Cache**接口的实现，一级缓存只会涉及到这一个**PerpetualCache**子类，**Cache**的其他实现将会放到二级缓存中介绍）。



我们知道，**Cache最核心的实现其实就是一个Map，将本次查询使用的特征值作为key，将查询结果作为value存储到Map中。**

现在最核心的问题出现了：**怎样来确定一次查询的特征值？**

换句话说就是：**怎样判断某两次查询是完全相同的查询？**

也可以这样说：**如何确定Cache中的key值？**

**MyBatis**认为，对于两次查询，如果以下条件都完全一样，那么就认为它们是完全相同的两次查询：

***1. 传入的 statementId***

***2. 查询时要求的结果集中的结果范围 （结果的范围通过rowBounds.offset和rowBounds.limit表示）；***

***3. 这次查询所产生的最终要传递给JDBC java.sql.Preparedstatement的Sql语句字符串（boundSql.getSql() ）***

***4. 传递给java.sql.Statement要设置的参数值***

现在分别解释上述四个条件：

1. 传入的***statementId***，对于**MyBatis**而言，你要使用它，必须需要一个***statementId***，它代表着你将执行什么样的**Sql**；

2. **MyBatis**自身提供的分页功能是通过**RowBounds**来实现的，它通过***rowBounds.offset***和***rowBounds.limit***来过滤查询出来的结果集，这种分页功能是基于查询结果的再过滤，而不是进行数据库的物理分页；

由于**MyBatis**底层还是依赖于**JDBC**实现的，那么，对于两次完全一模一样的查询，**MyBatis**要保证对于底层**JDBC**而言，也是完全一致的查询才行。而对于**JDBC**而言，两次查询，只要传入给**JDBC**的**SQL**语句完全一致，传入的参数也完全一致，就认为是两次查询是完全一致的。

上述的第3个条件正是要求保证传递给**JDBC**的**SQL**语句完全一致；第4条则是保证传递给**JDBC**的参数也完全一致；

3、4讲的有可能比较含糊，举一个例子：

**[html]** [view plain](http://blog.csdn.net/luanlouis/article/details/41280959) [copy](http://blog.csdn.net/luanlouis/article/details/41280959)

[print?](http://blog.csdn.net/luanlouis/article/details/41280959)

1. **<select** id="selectByCritiera" parameterType="java.util.Map" resultMap="BaseResultMap"**>**
2. select employee\_id,first\_name,last\_name,email,salary
3. from louis.employees
4. where  employee\_id = #{employeeId}
5. and first\_name= #{firstName}
6. and last\_name = #{lastName}
7. and email = #{email}
8. **</select>**

如果使用上述的"**selectByCritiera**"进行查询，那么，**MyBatis**会将上述的**SQL**中的**#{}** 都替换成 **?**如下：

**[sql]** [view plain](http://blog.csdn.net/luanlouis/article/details/41280959) [copy](http://blog.csdn.net/luanlouis/article/details/41280959)

[print?](http://blog.csdn.net/luanlouis/article/details/41280959)

1. **select** employee\_id,first\_name,last\_name,email,salary
2. **from** louis.employees
3. **where**  employee\_id = ?
4. and first\_name= ?
5. and last\_name = ?
6. and email = ?

**MyBatis**最终会使用上述的**SQL**字符串创建**JDBC**的**java.sql.PreparedStatement**对象，对于这个**PreparedStatement**对象，还需要对它设置参数，调用**setXXX()**来完成设值，第4条的条件，就是要求对设置**JDBC**的**PreparedStatement**的参数值也要完全一致。

          即3、4两条MyBatis最本质的要求就是：

**调用JDBC的时候，传入的SQL语句要完全相同，传递给JDBC的参数值也要完全相同。**

综上所述,CacheKey由以下条件决定：

**statementId  + rowBounds  + 传递给JDBC的SQL  + 传递给JDBC的参数值**

**CacheKey的创建**

对于每次的查询请求，**Executor**都会根据传递的参数信息以及动态生成的**SQL**语句，将上面的条件根据一定的计算规则，创建一个对应的**CacheKey**对象。

我们知道创建**CacheKey**的目的，就两个：

    1. 根据**CacheKey**作为**key**,去**Cache缓存**中查找缓存结果；

    2. 如果查找缓存命中失败，则通过此**CacheKey**作为**key**，将**从数据库查询到的结果**作为**value**，组成**key**,**value**对存储到**Cache**缓存中。

**CacheKey**的构建被放置到了**Executor**接口的实现类**BaseExecutor**中，定义如下：

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/41280959) [copy](http://blog.csdn.net/luanlouis/article/details/41280959)

[print?](http://blog.csdn.net/luanlouis/article/details/41280959)

1. /\*\*
2. \* 所属类:  org.apache.ibatis.executor.BaseExecutor
3. \* 功能   :   根据传入信息构建CacheKey
4. \*/
5. **public** CacheKey createCacheKey(MappedStatement ms, Object parameterObject, RowBounds rowBounds, BoundSql boundSql) {
6. **if** (closed) **throw** **new** ExecutorException("Executor was closed.");
7. CacheKey cacheKey = **new** CacheKey();
8. //1.statementId
9. cacheKey.update(ms.getId());
10. //2. rowBounds.offset
11. cacheKey.update(rowBounds.getOffset());
12. //3. rowBounds.limit
13. cacheKey.update(rowBounds.getLimit());
14. //4. SQL语句
15. cacheKey.update(boundSql.getSql());
16. //5. 将每一个要传递给JDBC的参数值也更新到CacheKey中
17. List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
18. TypeHandlerRegistry typeHandlerRegistry = ms.getConfiguration().getTypeHandlerRegistry();
19. **for** (**int** i = 0; i < parameterMappings.size(); i++) { // mimic DefaultParameterHandler logic
20. ParameterMapping parameterMapping = parameterMappings.get(i);
21. **if** (parameterMapping.getMode() != ParameterMode.OUT) {
22. Object value;
23. String propertyName = parameterMapping.getProperty();
24. **if** (boundSql.hasAdditionalParameter(propertyName)) {
25. value = boundSql.getAdditionalParameter(propertyName);
26. } **else** **if** (parameterObject == **null**) {
27. value = **null**;
28. } **else** **if** (typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) {
29. value = parameterObject;
30. } **else** {
31. MetaObject metaObject = configuration.newMetaObject(parameterObject);
32. value = metaObject.getValue(propertyName);
33. }
34. //将每一个要传递给JDBC的参数值也更新到CacheKey中
35. cacheKey.update(value);
36. }
37. }
38. **return** cacheKey;
39. }

CacheKey的hashcode生成算法

刚才已经提到，Cache接口的实现，本质上是使用的HashMap<k,v>,而构建CacheKey的目的就是为了作为HashMap<k,v>中的key值。而HashMap是通过key值的hashcode 来组织和存储的，那么，构建CacheKey的过程实际上就是构造其hashCode的过程。下面的代码就是CacheKey的核心hashcode生成算法，感兴趣的话可以看一下：

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/41280959) [copy](http://blog.csdn.net/luanlouis/article/details/41280959)

[print?](http://blog.csdn.net/luanlouis/article/details/41280959)

1. **public** **void** update(Object object) {
2. **if** (object != **null** && object.getClass().isArray()) {
3. **int** length = Array.getLength(object);
4. **for** (**int** i = 0; i < length; i++) {
5. Object element = Array.get(object, i);
6. doUpdate(element);
7. }
8. } **else** {
9. doUpdate(object);
10. }
11. }
13. **private** **void** doUpdate(Object object) {
15. //1. 得到对象的hashcode;
16. **int** baseHashCode = object == **null** ? 1 : object.hashCode();
17. //对象计数递增
18. count++;
19. checksum += baseHashCode;
20. //2. 对象的hashcode 扩大count倍
21. baseHashCode \*= count;
22. //3. hashCode \* 拓展因子（默认37）+拓展扩大后的对象hashCode值
23. hashcode = multiplier \* hashcode + baseHashCode;
24. updateList.add(object);
25. }

**一级缓存的性能分析**

我将从两个 一级缓存的特性来讨论**SqlSession**的一级缓存性能问题：

**1.MyBatis对会话（Session）级别的一级缓存设计的比较简单，就简单地使用了HashMap来维护，并没有对HashMap的容量和大小进行限制。**

读者有可能就觉得不妥了：如果我一直使用某一个**SqlSession**对象查询数据，这样会不会导致**HashMap**太大，而导致 java.lang.OutOfMemoryError错误啊？ 读者这么考虑也不无道理，不过**MyBatis**的确是这样设计的。

**MyBatis**这样设计也有它自己的理由：

a.  一般而言**SqlSession**的生存时间很短。一般情况下使用一个**SqlSession**对象执行的操作不会太多，执行完就会消亡；

b.  对于某一个**SqlSession**对象而言，只要执行**update**操作（**update、insert、delete**），都会将这个**SqlSession**对象中对应的一级缓存清空掉，所以一般情况下不会出现缓存过大，影响JVM内存空间的问题；

c.  可以手动地释放掉**SqlSession**对象中的缓存。

**2.  一级缓存是一个粗粒度的缓存，没有更新缓存和缓存过期的概念**

**MyBatis**的一级缓存就是使用了简单的**HashMap**，**MyBatis**只负责将查询数据库的结果存储到缓存中去， 不会去判断缓存存放的时间是否过长、是否过期，因此也就没有对缓存的结果进行更新这一说了。

根据一级缓存的特性，在使用的过程中，我认为应该注意：

1、对于数据变化频率很大，并且需要高时效准确性的数据要求，我们使用**SqlSession**查询的时候，要控制好**SqlSession**的生存时间，**SqlSession**的生存时间越长，它其中缓存的数据有可能就越旧，从而造成和真实数据库的误差；同时对于这种情况，用户也可以手动地适时清空**SqlSession**中的缓存；

2、对于只执行、并且频繁执行大范围的**select**操作的**SqlSession**对象，**SqlSession**对象的生存时间不应过长。

**举例：**

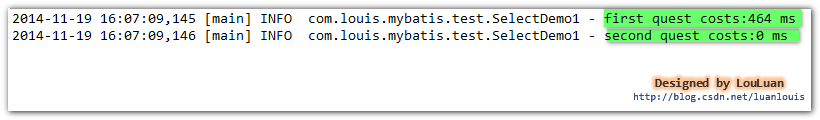
**例1、看下面这个例子，下面的例子使用了同一个SqlSession指令了两次完全一样的查询，将两次查询所耗的时间打印出来，结果如下：**

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/41280959) [copy](http://blog.csdn.net/luanlouis/article/details/41280959)

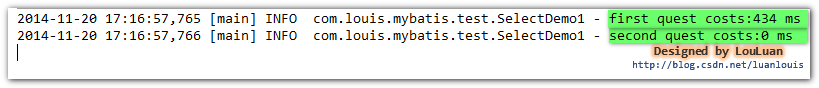
[print?](http://blog.csdn.net/luanlouis/article/details/41280959)

1. **package** com.louis.mybatis.test;
3. **import** java.io.InputStream;
4. **import** java.util.Date;
5. **import** java.util.HashMap;
6. **import** java.util.List;
7. **import** java.util.Map;
9. **import** org.apache.commons.logging.Log;
10. **import** org.apache.commons.logging.LogFactory;
11. **import** org.apache.ibatis.executor.BaseExecutor;
12. **import** org.apache.ibatis.io.Resources;
13. **import** org.apache.ibatis.session.SqlSession;
14. **import** org.apache.ibatis.session.SqlSessionFactory;
15. **import** org.apache.ibatis.session.SqlSessionFactoryBuilder;
16. **import** org.apache.log4j.Logger;
18. **import** com.louis.mybatis.model.Employee;
20. /\*\*
21. \* SqlSession 简单查询演示类
22. \* @author louluan
23. \*/
24. **public** **class** SelectDemo1 {
26. **private** **static** **final** Logger loger = Logger.getLogger(SelectDemo1.**class**);
28. **public** **static** **void** main(String[] args) **throws** Exception {
29. InputStream inputStream = Resources.getResourceAsStream("mybatisConfig.xml");
30. SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();
31. SqlSessionFactory factory = builder.build(inputStream);
33. SqlSession sqlSession = factory.openSession();
34. //3.使用SqlSession查询
35. Map<String,Object> params = **new** HashMap<String,Object>();
36. params.put("min\_salary",10000);
37. //a.查询工资低于10000的员工
38. Date first = **new** Date();
39. //第一次查询
40. List<Employee> result = sqlSession.selectList("com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary",params);
41. loger.info("first quest costs:"+ (**new** Date().getTime()-first.getTime()) +" ms");
42. Date second = **new** Date();
43. result = sqlSession.selectList("com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary",params);
44. loger.info("second quest costs:"+ (**new** Date().getTime()-second.getTime()) +" ms");
45. }
47. }

**运行结果：**

  
由上面的结果你可以看到，第一次查询耗时464ms，而第二次查询耗时不足1ms,这是因为第一次查询后，MyBatis会将查询结果存储到SqlSession对象的缓存中，当后来有完全相同的查询时，直接从缓存中将结果取出。

**例2、对上面的例子做一下修改：在第二次调用查询前，对参数 HashMap类型的params多增加一些无关的值进去，然后再执行，看查询结果：**



    从结果上看，虽然第二次查询时传递的params参数不一致，但还是从一级缓存中取出了第一次查询的缓存。

读到这里，请读者晓得这一个问题：

**MyBatis认为的完全相同的查询，不是指使用sqlSession查询时传递给算起来Session的所有参数值完完全全相同，你只要保证statementId，rowBounds,最后生成的SQL语句，以及这个SQL语句所需要的参数完全一致就可以了。**

以上就是MyBatis一级缓存的所有内容，如果有什么叙述不妥或者错误的地方，希望大虾们批评指正！！！！！

接下来的一篇文章，我将给大家揭开**MyBatis二级缓存**的神秘面纱，敬请关注~~~~~~

作者的话

    本文是《深入理解mybatis原理》系列的其中一篇，如果您有兴趣，请关注该系列的其他文章～

   觉得本文不错，顺手点个赞哦～～您的鼓励，是我继续分享知识的强大动力！

-----------------------------------------------------------------------------------------------------------------------------------------

                                                                本文源自  http://blog.csdn.net/luanlouis/，如需转载，请注明出处，谢谢！

顶

67

踩

0