MyBatis中一级缓存的配置与实现原理

> 本文为稀土掘金技术社区首发签约文章，30天内禁止转载，30天后未获授权禁止转载，侵权必究！
>
> 大家好，我是[王有志](http://cxyroad.com/ ”https://www.yuque.com/wangyouzhi-u3woi/cr7d5y/uw8c5iyvpgnqpzmg”)，一个分享硬核 Java 技术的金融摸鱼侠，欢迎大家加入 Java 人自己的交流群“[共同富裕的 Java 人](http://cxyroad.com/ ”https://www.yuque.com/wangyouzhi-u3woi/cr7d5y/nwry2mdlktok50bt”)” 。

今天我们的主题是 MyBatis 的一级缓存，本文中我会和大家一起认识并学习如何配置 MyBatis 的一级缓存，并通过源码来分析 MyBatis 一级缓存的实现原理。

MyBatis 支持两级缓存，在不进行任何配置的情况话，MyBatis 会默认开启一级缓存。MyBatis 一级缓存的作用域是 SqlSession 实例，在没有执行新增，更新和删除操作，以及没有主动刷新缓存的情况下，同一个 SqlSession 实例中多次执行完全相同（调用的查询方法以及入参都完全相同）的查询语句时，只有第一次执行的查询语句会查询数据库，并将查询结果存储到 MyBatis 的一级缓存中，其余的查询都会直接从 MyBatis 的一级缓存中获取数据。

下面我们就通过两个例子，一起来看下 MyBatis 一级缓存的效果。

Tips：

* 如无说明，本文中默认使用《MyBatis映射器：一对一关联查询》和《MyBatis映射器：一对多关联查询》中的 Mapper 接口，SQL 语句和数据库；
* 本文是基于 MyBatis 3.5.15 版本完成的，所有的结论也是基于该版本得出的，可能存在与 MyBatis 早期版本不符的情况。

相同 SqlSession 实例执行完全相同的查询语句

这个例子中，我们使用同一个 SqlSession 实例来获取两次 UserOrderMapper

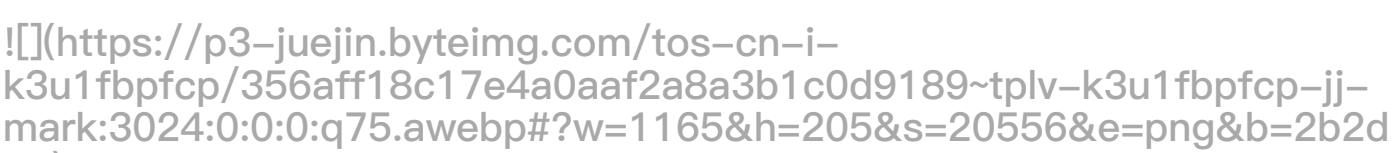
接口实例，并使用相同的入参分别调用
`UserOrderMapper#selectByOrderNo`方法，测试代码如下：

```
public class CacheTest {
  public void testFirstLevelCacheUseSameSqlSession() throws
IOException {
    Reader mysqlReader = Resources.getResourceAsReader("mybatis-
config.xml");
    SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(mysqlReader);
    SqlSession sqlSession = sqlSessionFactory.openSession();

    // 第一次执行查询
    UserOrderMapper userOrderMapper =
sqlSession.getMapper(UserOrderMapper.class);
    System.out.println("第一次查询，使用【userOrderMapper】执行查询");
    UserOrderDO userOrder =
userOrderMapper.selectByOrderNo("D20240508220804578 8");
    System.out.println("第一次查询结果：" +
JSON.toJSONString(userOrder));

    // 第二次执行查询
    UserOrderMapper newUserOrderMapper =
sqlSession.getMapper(UserOrderMapper.class);
    System.out.println("第二次查询，使用【newUserOrderMapper】执行查
询");
    UserOrderDO newUserOrder =
newUserOrderMapper.selectByOrderNo("D20240508220804578 8");
    System.out.println("第二次查询结果：" +
JSON.toJSONString(userOrder));
  }
}
```

我们来看控制台输出的结果：

可以看到，控制台输出的日志中，两次查询只执行了一次 SQL 语句，这说明第

二次查询并没有真正的查询数据库。如果你在测试代码的第 16 行打上断点，你还能够看到第一次查询出来的 userOrder 对象与第二次查询出来的 newUserOrder 对象是同一个对象，如下：



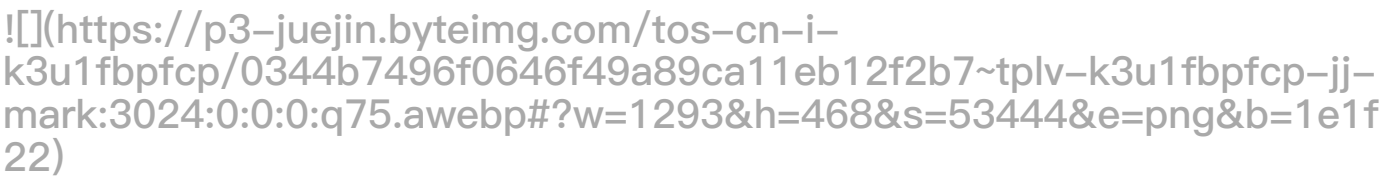
不同 SqlSession 实例执行完全相同的查询语句

现在我们创建一个新的 SqlSession 实例，并获取 UserOrderMapper 接口实例，执行完全相同的查询语句，测试代码如下：

```
public class CacheTest {

  public void testFirstLevelCacheUseDifferentSqlSession() throws IOException {
    Reader mysqlReader = Resources.getResourceAsReader("mybatis-config.xml");
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(mysqlReader);

    // 第一次执行查询
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserOrderMapper userOrderMapper = sqlSession.getMapper(UserOrderMapper.class);
    System.out.println("第一次查询，使用【userOrderMapper】执行查询");
    UserOrderDO userOrder = userOrderMapper.selectByOrderNo("D20240508220804578 8");
    System.out.println("第一次查询结果：" + JSON.toJSONString(userOrder));

    // 第二次执行查询
    SqlSession newSqlsession = sqlSessionFactory.openSession();
    UserOrderMapper newUserOrderMapper = newSqlsession.getMapper(UserOrderMapper.class);
    System.out.println("第二次查询，使用【newUserOrderMapper】执行查询");
    UserOrderDO newUserOrder = newUserOrderMapper.selectByOrderNo("D20240508220804578 8");
```

```
    System.out.println(”第二次查询结果：” +
JSON.toJSONString(userOrder));
  }
}
```

再来看控制台输出的查询结果：



可以看到，虽然依旧是两次完全相同的查询，但是 MyBatis 却执行了两次查询数据库的动作，这也表明了在**默认开始 MyBatis 一级缓存的场景下，不同的 SqlSession 之间，一级缓存是相互隔离的**。

关闭 MyBatis 的一级缓存

关于关闭 MyBatis 的一级缓存的方法，网上有一部分文章给出的答案是，在 mybatis-config.xml 文件中配置 cacheEnabled 参数为 false，如下：

```
<configuration>
  <settings>
    <setting name=”cacheEnabled” value=”false”/>
  </settings>
</configuration>
```

首先要强调一下：

这个答案是错误的！

这个答案是错误的！

这个答案是错误的！

重要的事情我们说三遍。

MyBatis 核心配置中的 cacheEnabled 配置并不是用来控制以一级缓存的，而是用来控制二级缓存的，它并不能起到关闭一级缓存的作用。

下面我们就一起来看看在 MyBatis 中，“关闭”一级缓存的正确姿势。

使用 localCacheScope 配置

实际上，MyBatis 没有提供关闭一级缓存的功能，因此我所说的“关闭”MyBatis 的一级缓存，只是通过使用某些配置让一级缓存看起来像是被“关闭”了一样。

第一种方式是在 **mybatis-config.xml 文件中配置一级缓存的作用域，默认的作用域是 SqlSession（MyBatis 的配置为 SESSION），同时也提供了基于 Statement 的作用域，也就是基于每次查询操作**。

MyBatis 的配置枚举类源码如下：

```
public enum LocalCacheScope {
  SESSION, STATEMENT
}
```

mybaits-config.xml 中的配置方式如下：

```
<configuration>
  <settings>
    <setting name=”localCacheScope” value=”STATEMENT”/>
  </settings>
</configuration>
```

```

可以通过将一级缓存的作用域修改为基于 STATEMENT 来间接的实现“关闭”MyBatis 的一级缓存。

配置完成后，你可以再次执行测试案例`CacheTest#testFirstLevelCacheUseSameSqlSession`，通过控制台输出的执行日志，来观察 MyBatis 的一级缓存是否起到了作用。

### 使用 flushCache 配置

如果你希望**只有在执行某个指定的查询语句时关闭一级缓存，我们可以在 MyBatis 的映射器中，为指定 SQL 语句添加 flushCache 配置实现**。

例如，我们修改`UserOrderMapper#selectByOrderNo`方法对应映射器中的 SQL 语句，为其添加 flushCache 配置，如下：

```
<select id=”selectByOrderNo” resultMap=”BaseResultMap” flushCache=”true”>
  select * from user_order where order_no = #{orderNo,jdbcType=VARCHAR}
</select>
```

注意，别忘了先将 mybatis-config.xml 文件中的 localCacheScope 配置注释掉，避免对测试结果产生影响。

我们再次行测试案例`CacheTest#testFirstLevelCacheUseSameSqlSession`，可以看到同样执行了两次查询语句。

**Tips**：使用 flushCache 配置会清空当前 SqlSession 中的一级缓存，也就是说此时执行未使用 flushCache 配置的查询语句也会重新查询数据库，这点我们在下文的实现原理分析中还会看到。

MyBatis 一级缓存的实现原理
---
```

前面我们说过，MyBatis 的一级缓存的作用域是 SqlSession 实例，这么说是没问题的，但是不够精确，确切的说，**MyBatis 的一级缓存是基于 Executor 实例的，而每个 SqlSession 实例都会持有一个 Executor 实例**，由于我们通常不会直接使用 Executor，因此从直观上来看，MyBatis 的一级缓存的作用域是 SqlSession 实例。

MyBatis 一级缓存的创建

我们先来看 MyBatis 是如何通过 SqlSessionFactory 获取 SqlSession 的，`DefaultSqlSessionFactory#openSession`方法的源码如下：

```
public SqlSession openSession() {
  return
openSessionFromDataSource(configuration.getDefaultExecutorType(),
null, false);
}
```

接着我们来看`DefaultSqlSessionFactory#openSessionFromDataSource`的部分源码：

```
private SqlSession openSessionFromDataSource(ExecutorType
execType, TransactionIsolationLevel level, boolean autoCommit) {
  final Environment environment = configuration.getEnvironment();
  final TransactionFactory transactionFactory =
getTransactionFactoryFromEnvironment(environment);
  Transaction tx =
transactionFactory.newTransaction(environment.getDataSource(), level,
autoCommit);
  final Executor executor = configuration.newExecutor(tx, execType);
  return new DefaultSqlSession(configuration, executor, autoCommit);
}
```

Tips：我对展示的源码做了一些删减，但是不影响其核心逻辑。

第 5 行的源码中，MyBatis 创建了 Executor 实例，这里默认创建的是 SimpleExecutor 实例，并使用 CachingExecutor 实例进行装饰（装饰器模式）。这个方法我在《[大厂面试题：两道来自京东的关于MyBatis执行器的面试题](http://cxyroad.com/ "https://mp.weixin.qq.com/s/g8iSJ-S_M5DbgGpbj_k14w")》中和大家聊过，大家可以翻一翻之前的文章。

在我们的测试案例中 MyBatis 创建的 Executor 的类型是 SimpleExecutor，这里我们来看一下 SimpleExecutor 的构造方法：

```
public class SimpleExecutor extends BaseExecutor {
  public SimpleExecutor(Configuration configuration, Transaction transaction) {
    super(configuration, transaction);
  }
}
```

SimpleExecutor 的构造方法中并没有做其它的额外处理，而是直接调用了父类 BaseExecutor 的构造方法，实际上其它类型的 Executor 的构造方法也没有做任何处理，这说明不同 Executor 的只有在行为（方法）上是有区别的。

我们来看 BaseExecutor 构造方法的源码：

```
public abstract class BaseExecutor implements Executor {
  protected BaseExecutor(Configuration configuration, Transaction transaction) {
    this.transaction = transaction;
    this.deferredLoads = new ConcurrentLinkedQueue<>();
    this.localCache = new PerpetualCache("LocalCache");
    this.localOutputParameterCache = new PerpetualCache("LocalOutputParameterCache");
    this.closed = false;
    this.configuration = configuration;
    this.wrapper = this;
  }
}
```

```

第 5 行源码为 Executor 的实例创建了 localCache，类型为 PerpetualCache，这里我们很容易从变量命名上可以得知，这是一个本地缓存，实际上这就是我们说的 MyBatis 的一级缓存。

再来看 PerpetualCache 的构造方法：

```
public class PerpetualCache implements Cache {

  private final String id;

  private final Map<Object, Object> cache = new HashMap<>();

  public PerpetualCache(String id) {
    this.id = id;
  }
}
```

通过 PerpetualCache 的构造方法我们可以知道，**Executor 的本地缓存（localCache）的底层容器是 HashMap**。这与很多本地缓存的底层存储容器是相似的，例如 Caffeine 的底层存储容器是 ConcurrentHashMap。

我们回到`DefaultSqlSessionFactory#openSessionFromDataSource`方法的第 6 行调用了 DefaultSqlSession 的构造方法，我们一起来看这个方法的源码：

```
public DefaultSqlSession(Configuration configuration, Executor executor, boolean autoCommit) {
  this.configuration = configuration;
  this.executor = executor;
  this.dirty = false;
  this.autoCommit = autoCommit;
}
```

这里我们就可以看到，我们通过 SqlSessionFactory 获取到的 SqlSession 实
```

例，其本身是持有 Executor 实例的，而 Executor 实例才是缓存的真正持有者，这也就是为什么我说“**MyBatis 的一级缓存是基于 Executor 实例的**”。

MyBatis 一级缓存的使用

接着我们跳转到 MyBatis 执行查询语句的方法，这里会通过一系列代理逐步跳转到`BaseExecutor#query`方法，因为这个过程中与我们要说的一级缓存没有关系，所有我们直接来看`BaseExecutor#query`方法，源码如下：

```
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler) throws SQLException {
  BoundSql boundSql = ms.getBoundSql(parameter);
  CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);
  return query(ms, parameter, rowBounds, resultHandler, key, boundSql);
}
```

注意，`BaseExecutor#query`有两个重载方法，在只使用一级缓存的场景下，两个方法都会执行到。

第 3 行中调用了`BaseExecutor#createCacheKey`方法，用于创建一级缓存中的 Key，我们来看这个方法的源码：

```
public CacheKey createCacheKey(MappedStatement ms, Object parameterObject, RowBounds rowBounds, BoundSql boundSql) {
  CacheKey cacheKey = new CacheKey();
  // 获取查询方法的Id，即接口全限名+方法名，如：com.wyz.mapper.UserOrderMapper.selectByOrderNo
  cacheKey.update(ms.getId());
  // RowBounds为MyBatis中逻辑分页参数
  cacheKey.update(rowBounds.getOffset());
  cacheKey.update(rowBounds.getLimit());
  // 获取查询方法的 SQL 语句
  cacheKey.update(boundSql.getSql());
  List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
```

```
  TypeHandlerRegistry typeHandlerRegistry =
ms.getConfiguration().getTypeHandlerRegistry();
  MetaObject metaObject = null;
  for (ParameterMapping parameterMapping : parameterMappings) {
    if (parameterMapping.getMode() != ParameterMode.OUT) {
      Object value;
      String propertyName = parameterMapping.getProperty();
      if (boundSql.hasAdditionalParameter(propertyName)) {
        value = boundSql.getAdditionalParameter(propertyName);
      } else if (parameterObject == null) {
        value = null;
      } else if
(typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) {
        value = parameterObject;
      } else {
        // 获取查询方法的参数
        if (metaObject == null) {
          metaObject = configuration.newMetaObject(parameterObject);
        }
        value = metaObject.getValue(propertyName);
      }
      cacheKey.update(value);
    }
  }
  // 获取 MyBatis 的环境配置Id
  if (configuration.getEnvironment() != null) {
    cacheKey.update(configuration.getEnvironment().getId());
  }
  return cacheKey;
}

```

`BaseExecutor#createCacheKey`方法的内容还是比较多的，拼装了非常多查询方法相关的内容（具体哪些内容大家可以看注释），这也是为了能够保证只有完全相同的查询才能命中缓存所作出的努力。

我们回到`BaseExecutor#query`方法中第 4 行调用的重载方法，部分源码如下：

```
public <E> List<E> query(MappedStatement ms, Object parameter,
RowBounds rowBounds, ResultHandler resultHandler, CacheKey key,
BoundSql boundSql) throws SQLException {
  if (ms.isFlushCacheRequired()) {
```

```
    clearLocalCache();
  }

  List<E> list = resultHandler == null ? (List<E>)
localCache.getObject(key) : null;
  if (list == null) {
    list = queryFromDatabase(ms, parameter, rowBounds, resultHandler,
key, boundSql);
  }

  if (configuration.getLocalCacheScope() ==
LocalCacheScope.STATEMENT) {
    clearLocalCache();
  }
  return list;
}
```

Tips：这里我对源码做了大量的删减和修改，例如第 4 行到第 6 行的条件语句，这里原本是“if...else”的形式，涉及到针对 statementType 为 CALLABLE 类型的 SQL 语句的处理，因为通常我们不会为映射器中的 SQL 语句配置 statementType，只会使用默认的 PREPARED 类型，所以我对源码做了一些修改，目的是为了突出 MyBatis 中一级缓存的逻辑。

第 6 行源码是通过 localCache 获取数据。紧接着来到了第 7 行到第 9 行源码的条件语句，如果缓存中可以获取到数据，那么就使用缓存中的数据，如果缓存中无法获取数据，就执行数据库查询来获取数据。

接着我们来看第 8 行中调用的`BaseExecutor#queryFromDatabase`方法，部分源码如下：

```
private <E> List<E> queryFromDatabase(MappedStatement ms, Object
parameter, RowBounds rowBounds, ResultHandler resultHandler,
CacheKey key, BoundSql boundSql) throws SQLException {
  List<E> list list = doQuery(ms, parameter, rowBounds, resultHandler,
boundSql);
  localCache.putObject(key, list);
  return list;
}
```

这段源码我删了很多，但核心逻辑非常简单，调用 Executor 实例的 doQuery 方法查询数据，然后将结果存储到 MyBatis 的一级缓存 localCache 中。

到这里，第一次查询的链路就结束了，当第二次执行完全相同的查询时，可以通过 MyBatis 的一级缓存 localCache 获取到数据，会将缓存中查询到的数据直接返回。

localCache 配置与 flushCache 配置分析

最后我们再来看前面提到的两种“关闭” MyBatis 一级缓存的处理逻辑。

我们接着来分析`BaseExecutor#query`方法：

```
public <E> List<E> query(MappedStatement ms, Object parameter,
RowBounds rowBounds, ResultHandler resultHandler, CacheKey key,
BoundSql boundSql) throws SQLException {
  if (ms.isFlushCacheRequired()) {
    clearLocalCache();
  }

  List<E> list = resultHandler == null ? (List<E>)
localCache.getObject(key) : null;
  if (list == null) {
    list = queryFromDatabase(ms, parameter, rowBounds, resultHandler,
key, boundSql);
  }

  if (configuration.getLocalCacheScope() ==
LocalCacheScope.STATEMENT) {
    clearLocalCache();
  }
  return list;
}
```

首先是第 2 行到第 4 行中的条件语句，根据映射器中 SQL 语句是否配置了 flushCache 属性，来决定是否调用`BaseExecutor#clearLocalCache`方法，如果是的话，会执行`BaseExecutor#clearLocalCache`方法，从名称来看就是用于清空缓存的方法，不过我们还是来看下`BaseExecutor#clearLocalCache`方法的源码：

```
public void clearLocalCache() {
  // 判断 Executor 实例是否关闭
  if (!closed) {
    localCache.clear();
  }
}
```

`BaseExecutor#clearLocalCache`方法的逻辑非常简单，在没有关闭 Executor 实例的情况下清空缓存，这里调用的是`PerpetualCache#clear`方法，源码如下：

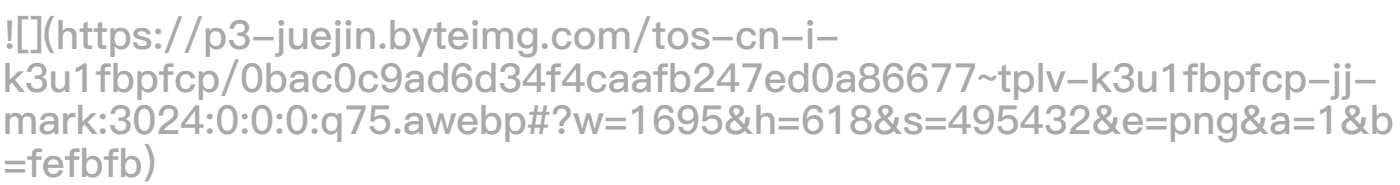
```
public void clear() {
  cache.clear();
}
```

前面我们已经得知了 PerpetualCache 的底层存储容器是 HashMap，因此这里调用`HashMap#clear`方法会将 HashMap 中所有的数据删除。

那么`BaseExecutor#query`方法中第 2 行到第 4 行的逻辑就很清晰了：**如果映射器中的 SQL 语句配置了 flushCache 属性，执行**`**BaseExecutor#clearLocalCache**`**方法，清空 MyBatis 的一级缓存，然后再执行查询逻辑**。

最后我们`BaseExecutor#query`方法的第 11 行到第 13 行的条件语句，得知了`BaseExecutor#clearLocalCache`方法的作用，那这个条件语句的逻辑是：**如果 mybatis-config.xml 中的 localCacheScope 配置为 STATEMENT，则在执行查询后（第一次查询会将数据库中的数据存储到缓存中）清空 MyBatis 的一级缓存**。

到这里，也解释了为什么前面我会说“MyBatis 的一级缓存不能被关闭”了，因为在实际的执行过程中，**MyBatis 通过数据库查询出来的数据依旧会被添加到一级缓存中，只不过根据配置的不同，会在不同的时机清除所有缓存**。



原文链接: https://juejin.cn/post/7372813290650484751