聊聊MyBatis缓存机制 (https://tech.meituan.com/2018/01/19/mybatis-cache.html)

2018年01月19日 作者: 凯伦 **文章链接 (https://tech.meituan.com/2018/01/19/mybatis-cache.html)** 18778字 38分钟阅读

前言

MyBatis是常见的Java数据库访问层框架。在日常工作中，开发人员多数情况下是使用MyBatis的默认缓存配置，但是MyBatis缓存机制有一些不足之处，在使用中容易引起脏数据，形成一些潜在的隐患。个人在业务开发中也处理过一些由于MyBatis缓存引发的开发问题，带着个人的兴趣，希望从应用及源码的角度为读者梳理MyBatis缓存机制。

本次分析中涉及到的代码和数据库表均放在GitHub上，地址： **mybatis-cache-demo (https://github.com/kailuncen/mybatis-cache-demo)** 。

目录

本文按照以下顺序展开。

一级缓存

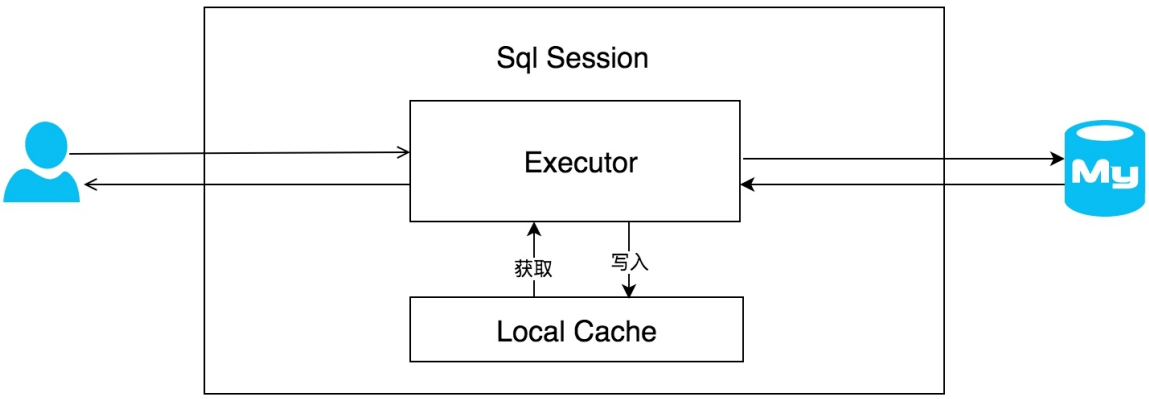
一级缓存介绍

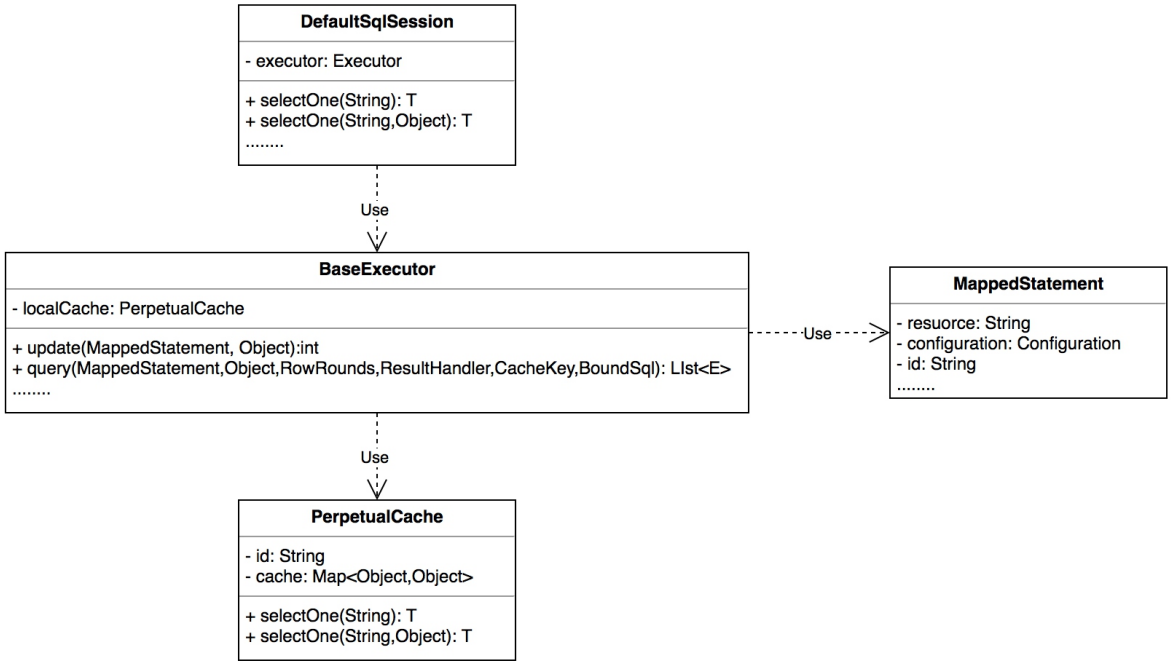
在应用运行过程中，我们有可能在一次数据库会话中，执行多次查询条件完全相同的SQL，MyBatis提供了一级缓存的方案优化这部分场景，如果是相同的SQL语句，会优先命中一级缓存，避免直接对数据库进行查询，提高性能。具体执行过程如下图所示。

每个SqlSession中持有了Executor，每个Executor中有一个LocalCache。当用户发起查询时，MyBatis根据当前执行的语句生成 `MappedStatement` ，在Local Cache进行查询，如果缓存命中的话，直接返回结果给用户，如果缓存没有命中的话，查询数据库，结果写入 `Local Cache` ，最后返回结果给用户。具体实现类的类关系图如下图所示。

一级缓存配置

我们来看看如何使用MyBatis一级缓存。开发者只需在MyBatis的配置文件中，添加如下语句，就可以使用一级缓存。共有两个选项， `SESSION` 或者 `STATEMENT` ，默认是 `SESSION` 级别，即在一个MyBatis会话中执行的所有语句，都会共享这一个缓存。一种是 `STATEMENT` 级别，可





以理解为缓存只对当前执行的这一个`Statement`有效。

```
<setting name="localCacheScope" value="SESSION"/>
```

一级缓存实验

接下来通过实验，了解MyBatis一级缓存的效果，每个单元测试后都请恢复被修改的数据。

首先是创建示例表student，创建对应的POJO类和增改的方法，具体可以在entity包和mapper包中查看。

```
CREATE TABLE `student` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(200) COLLATE utf8_bin DEFAULT NULL,
  `age` tinyint(3) unsigned DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8 COLLATE=u
tf8_bin;
```

在以下实验中，id为1的学生名称是凯伦。

实验1

开启一级缓存，范围为会话级别，调用三次 `getStudentById` ，代码如下所示：

```
public void getStudentById() throws Exception {
        SqlSession sqlSession = factory.openSession(true); // 自动提交事务
        StudentMapper studentMapper = sqlSession.getMapper(StudentMapper.class);
        System.out.println(studentMapper.getStudentById(1));
        System.out.println(studentMapper.getStudentById(1));
        System.out.println(studentMapper.getStudentById(1));
    }
```

执行结果：

```
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 凯伦, 25
DEBUG [main] - <==      Total: 1
StudentEntity{id=1, name='凯伦', age=25}
StudentEntity{id=1, name='凯伦', age=25}
StudentEntity{id=1, name='凯伦', age=25}
```

只有第一次真正查询了数据库

我们可以看到，只有第一次真正查询了数据库，后续的查询使用了一级缓存。

实验2

增加了对数据库的修改操作，验证在一次数据库会话中，如果对数据库发生了修改操作，一级缓存是否会失效。

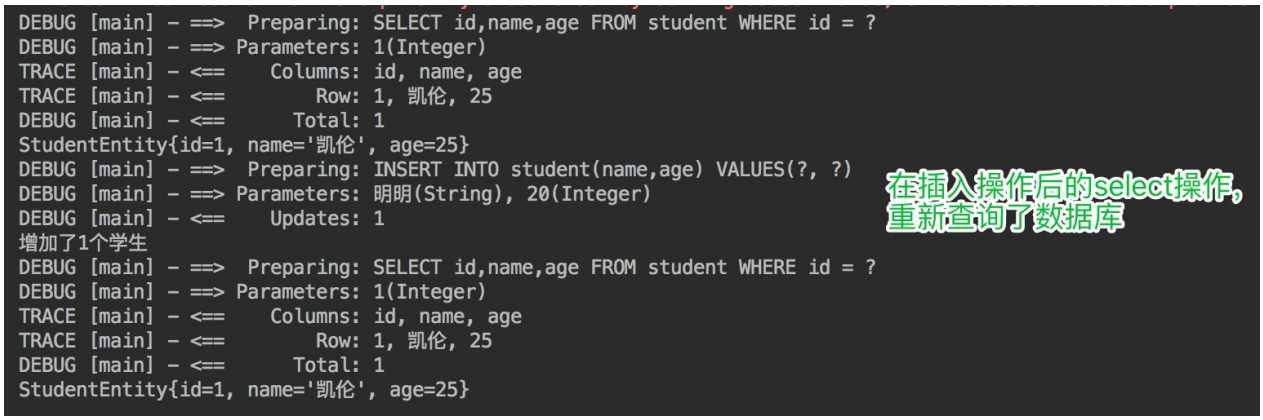
```
@Test
public void addStudent() throws Exception {
        SqlSession sqlSession = factory.openSession(true); // 自动提交事务
        StudentMapper studentMapper = sqlSession.getMapper(StudentMapper.class);
        System.out.println(studentMapper.getStudentById(1));
        System.out.println("增加了" + studentMapper.addStudent(buildStudent()) + "个学生");
        System.out.println(studentMapper.getStudentById(1));
        sqlSession.close();
}
```

执行结果：

我们可以看到，在修改操作后执行的相同查询，查询了数据库，**一级缓存失效。**

实验3

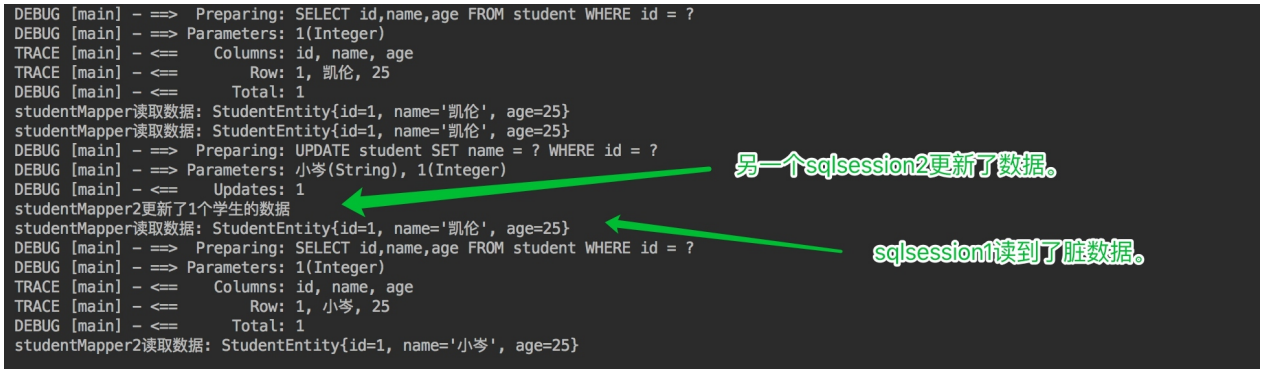
开启两个 `SqlSession` ，在 `sqlSession1` 中查询数据，使一级缓存生效，在 `sqlSession2` 中更新数据库，验证一级缓存只在数据库会话内部共享。




```
@Test
public void testLocalCacheScope() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);

        StudentMapper studentMapper = sqlSession1.getMapper(Stu
dentMapper.class);
        StudentMapper studentMapper2 = sqlSession2.getMapper(St
udentMapper.class);

        System.out.println("studentMapper读取数据: " + studentMa
pper.getStudentById(1));
        System.out.println("studentMapper读取数据: " + studentMa
pper.getStudentById(1));
        System.out.println("studentMapper2更新了" + studentMappe
r2.updateStudentName("小岑",1) + "个学生的数据");
        System.out.println("studentMapper读取数据: " + studentMa
pper.getStudentById(1));
        System.out.println("studentMapper2读取数据: " + studentM
apper2.getStudentById(1));
}
```



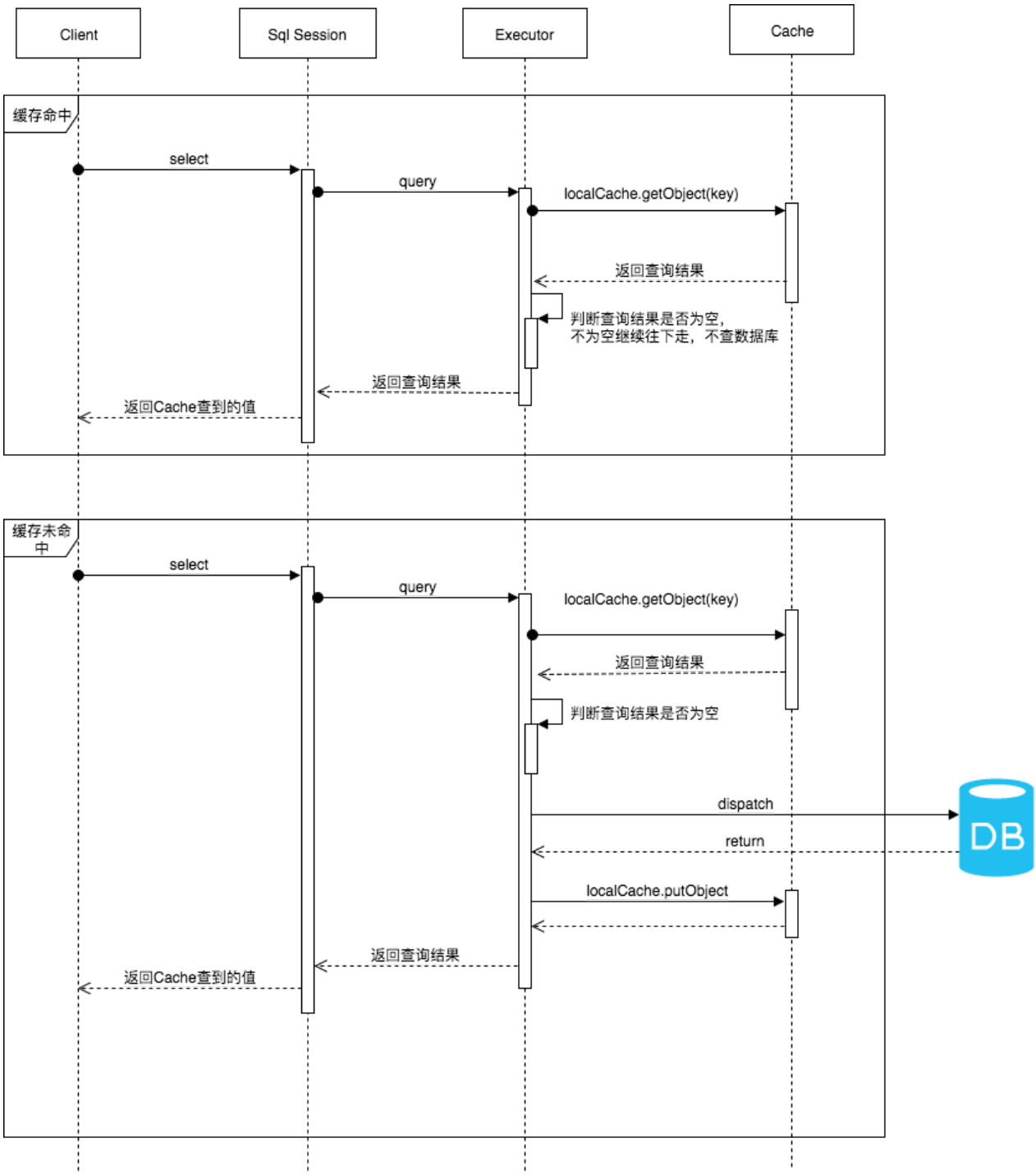
sqlSession2 更新了id为1的学生的姓名，从凯伦改为了小岑，但session1之后的查询中，id为1的学生的名字还是凯伦，出现了脏数据，也证明了之前的设想，一级缓存只在数据库会话内部共享。

一级缓存工作流程&源码分析

那么，一级缓存的工作流程是怎样的呢？我们从源码层面来学习一下。

工作流程

一级缓存执行的时序图，如下图所示。



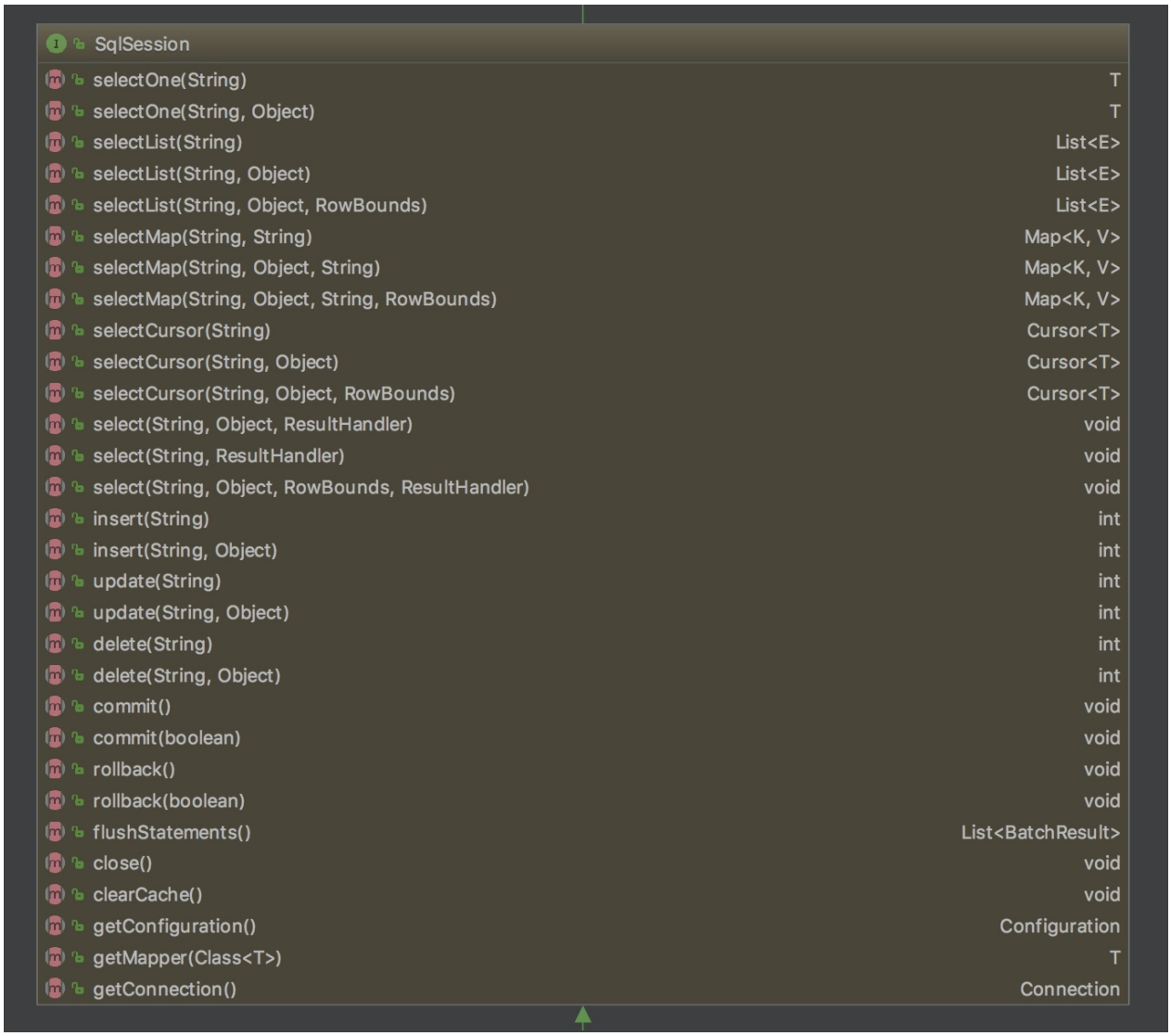
源码分析

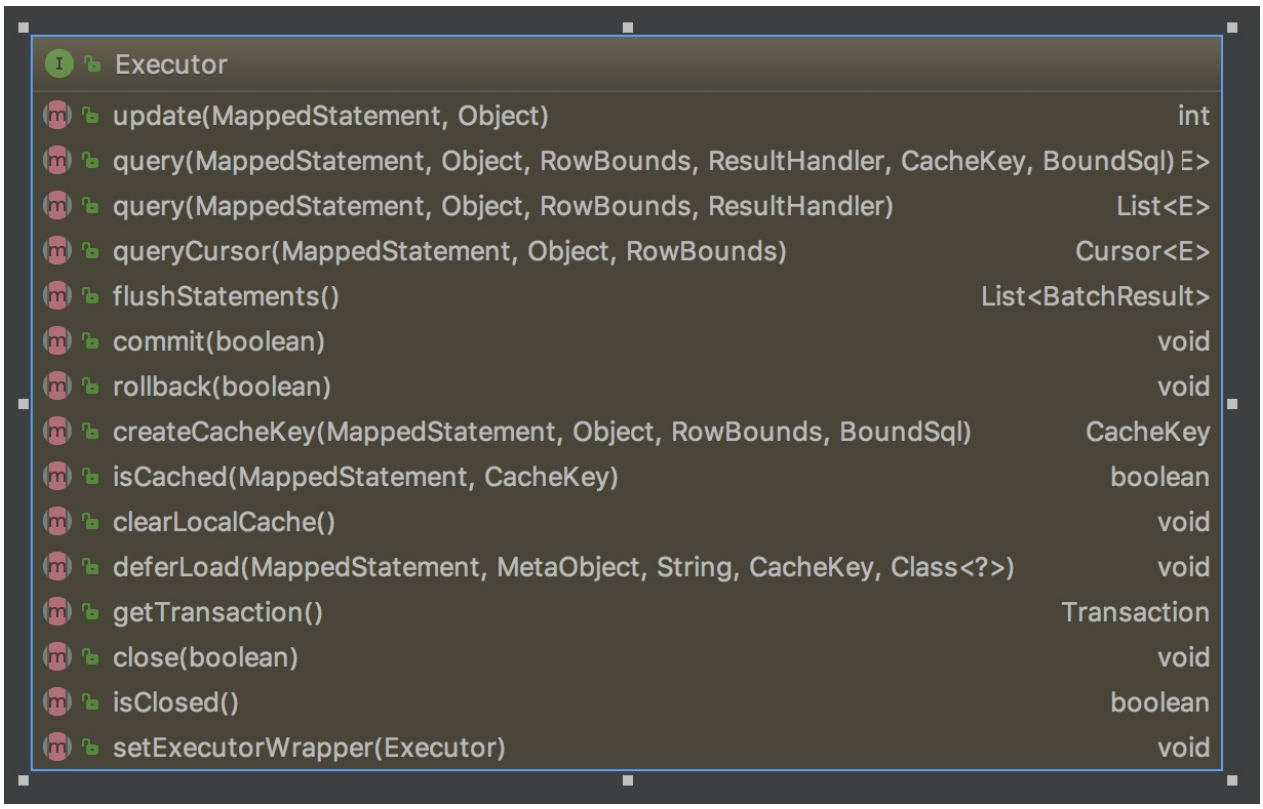
接下来将对MyBatis查询相关的核心类和一级缓存的源码进行走读。这对后面学习二级缓存也有帮助。

SqlSession： 对外提供了用户和数据库之间交互需要的所有方法，隐藏了底层的细节。默认实现类是 `DefaultSqlSession` 。

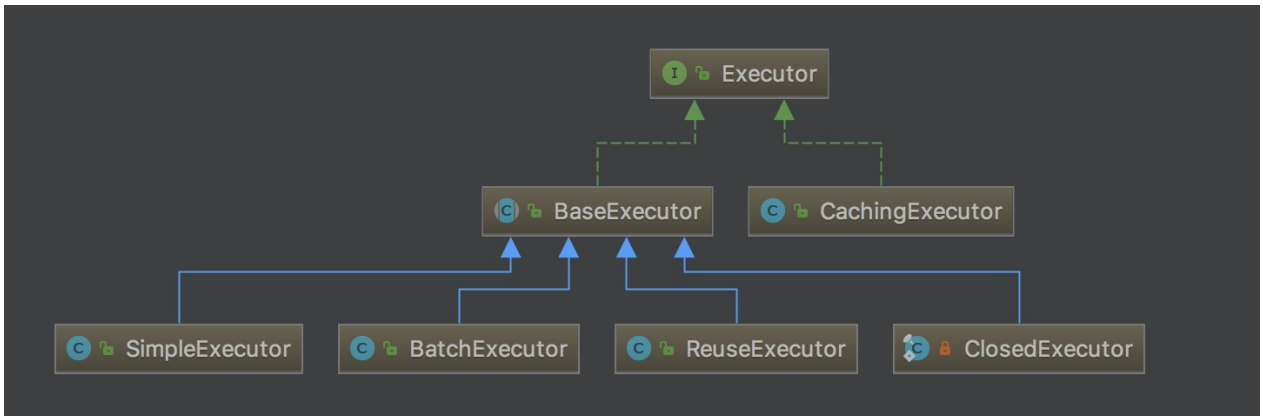
Executor： `SqlSession` 向用户提供操作数据库的方法，但和数据库操作有关的职责都会委托给Executor。

如下图所示，Executor有若干个实现类，为Executor赋予了不同的能力，大家可以根据类名，自行学习每个类的基本作用。





在一级缓存的源码分析中，主要学习 `BaseExecutor` 的内部实现。



BaseExecutor： `BaseExecutor` 是一个实现了Executor接口的抽象类，定义若干抽象方法，在执行的时候，把具体的操作委托给子类进行执行。

```
protected abstract int doUpdate(MappedStatement ms, Object para
meter) throws SQLException;
protected abstract List<BatchResult> doFlushStatements(boolean
 isRollback) throws SQLException;
protected abstract <E> List<E> doQuery(MappedStatement ms, Obje
ct parameter, RowBounds rowBounds, ResultHandler resultHandler,
BoundSql boundSql) throws SQLException;
protected abstract <E> Cursor<E> doQueryCursor(MappedStatement
 ms, Object parameter, RowBounds rowBounds, BoundSql boundSql)
throws SQLException;
```

在一级缓存的介绍中提到对 `Local Cache` 的查询和写入是在 `Executor` 内部完成的。在阅读 `BaseExecutor` 的代码后发现 `Local Cache` 是 `BaseExecutor` 内部的一个成员变量，如下代码所示。

```
public abstract class BaseExecutor implements Executor {
protected ConcurrentLinkedQueue<DeferredLoad> deferredLoads;
protected PerpetualCache localCache;
```

Cache： MyBatis中的Cache接口，提供了和缓存相关的最基本的操作，如下图所示：

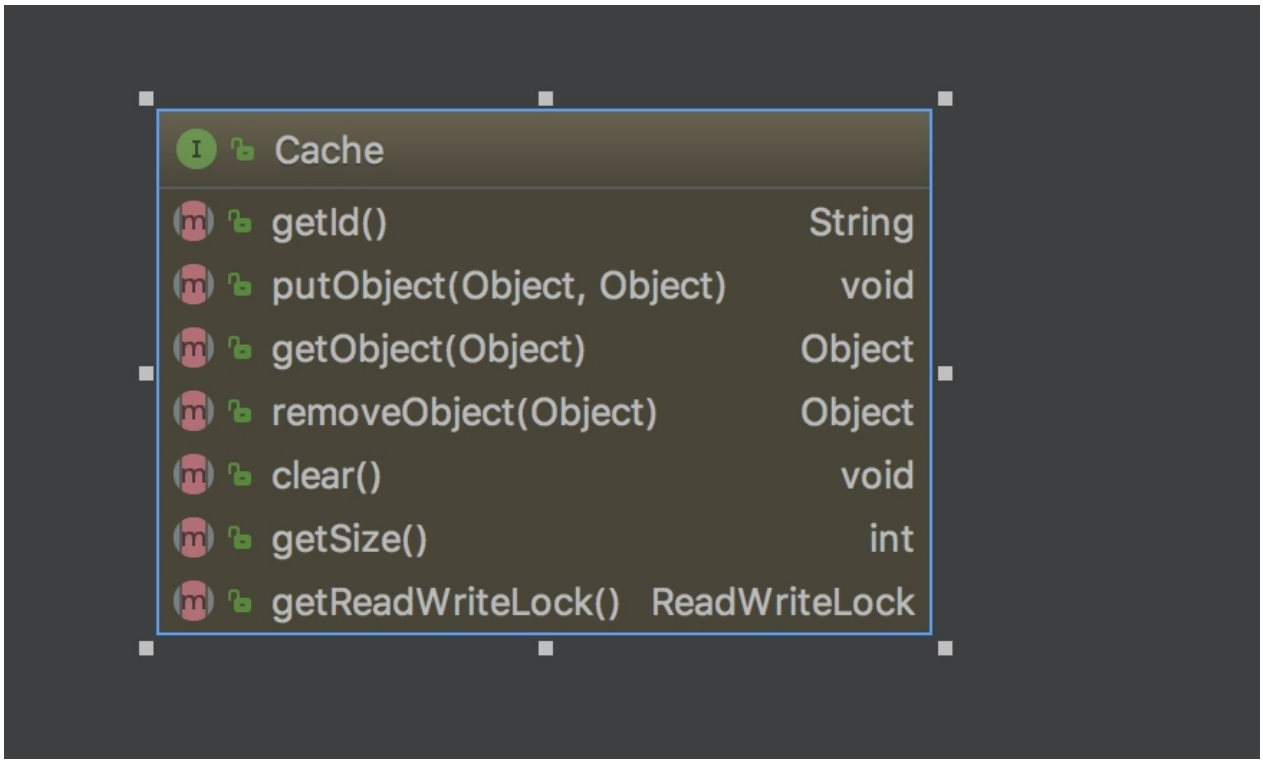
有若干个实现类，使用装饰器模式互相组装，提供丰富的操控缓存的能力，部分实现类如下图所示：

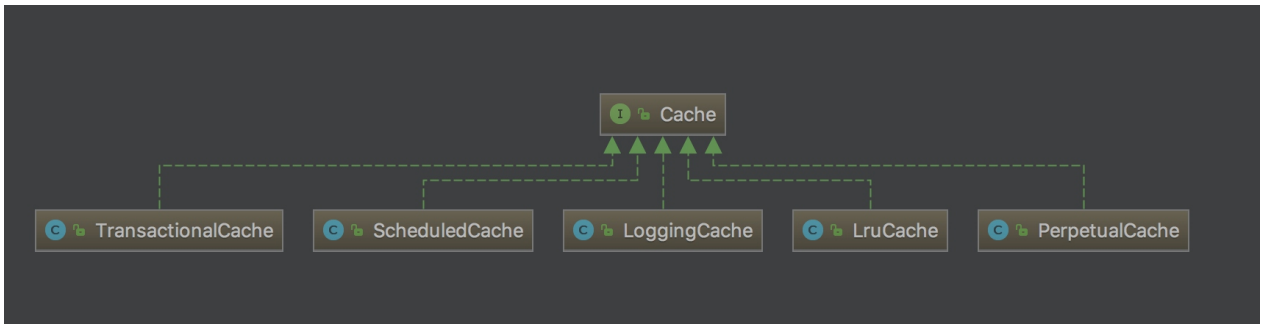
`BaseExecutor` 成员变量之一的 `PerpetualCache` ，是对Cache接口最基本的实现，其实现非常简单，内部持有HashMap，对一级缓存的操作实则是对HashMap的操作。如下代码所示：

```
public class PerpetualCache implements Cache {
  private String id;
  private Map<Object, Object> cache = new HashMap<Object, Objec
t>();
```

在阅读相关核心类代码后，从源代码层面对一级缓存工作中涉及到的相关代码，出于篇幅的考虑，对源码做适当删减，读者朋友可以结合本文，后续进行更详细的学习。

为执行和数据库的交互，首先需要初始化 `SqlSession` ，通过 `DefaultSqlSessionFactory`





开启`SqlSession`：

```
private SqlSession openSessionFromDataSource(ExecutorType execT
ype, TransactionIsolationLevel level, boolean autoCommit) {
    ............
    final Executor executor = configuration.newExecutor(tx, exe
cType);
    return new DefaultSqlSession(configuration, executor, autoC
ommit);
}
```

在初始化`SqlSesion`时，会使用`Configuration`类创建一个全新的`Executor`，作为`DefaultSqlSession`构造函数的参数，创建Executor代码如下所示：

```
public Executor newExecutor(Transaction transaction, ExecutorType executorType) {
    executorType = executorType == null ? defaultExecutorType : executorType;
    executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
    Executor executor;
    if (ExecutorType.BATCH == executorType) {
      executor = new BatchExecutor(this, transaction);
    } else if (ExecutorType.REUSE == executorType) {
      executor = new ReuseExecutor(this, transaction);
    } else {
      executor = new SimpleExecutor(this, transaction);
    }
    // 尤其可以注意这里，如果二级缓存开关开启的话，是使用CahingExecutor装饰BaseExecutor的子类
    if (cacheEnabled) {
      executor = new CachingExecutor(executor);
    }
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
}
```

SqlSession 创建完毕后，根据Statment的不同类型，会进入 SqlSession 的不同方法中，如果是 Select 语句的话，最后会执行到 SqlSession 的 selectList，代码如下所示：

```
@Override
public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {
      MappedStatement ms = configuration.getMappedStatement(statement);
      return executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO_RESULT_HANDLER);
}
```

SqlSession 把具体的查询职责委托给了Executor。如果只开启了一级缓存的话，首先会进入 BaseExecutor 的 query 方法。代码如下所示：

```
@Override
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);
}
```

在上述代码中，会先根据传入的参数生成CacheKey，进入该方法查看CacheKey是如何生成的，代码如下所示：

```
CacheKey cacheKey = new CacheKey();
cacheKey.update(ms.getId());
cacheKey.update(rowBounds.getOffset());
cacheKey.update(rowBounds.getLimit());
cacheKey.update(boundSql.getSql());
//后面是update了sql中带的参数
cacheKey.update(value);
```

在上述的代码中，将 `MappedStatement` 的Id、SQL的offset、SQL的limit、SQL本身以及SQL中的参数传入了CacheKey这个类，最终构成CacheKey。以下是这个类的内部结构：

```
private static final int DEFAULT_MULTIPLYER = 37;
private static final int DEFAULT_HASHCODE = 17;

private int multiplier;
private int hashcode;
private long checksum;
private int count;
private List<Object> updateList;

public CacheKey() {
    this.hashcode = DEFAULT_HASHCODE;
    this.multiplier = DEFAULT_MULTIPLYER;
    this.count = 0;
    this.updateList = new ArrayList<Object>();
}
```

首先是成员变量和构造函数，有一个初始的 `hachcode` 和乘数，同时维护了一个内部的 `updatelist` 。在 `CacheKey` 的 `update` 方法中，会进行一个 `hashcode` 和 `checksum` 的计算，同时把传入的参数添加进 `updatelist` 中。如下代码所示：

```
public void update(Object object) {
    int baseHashCode = object == null ? 1 : ArrayUtil.hashCode
(object);
    count++;
    checksum += baseHashCode;
    baseHashCode *= count;
    hashcode = multiplier * hashcode + baseHashCode;

    updateList.add(object);
}
```

同时重写了 `CacheKey` 的 `equals` 方法，代码如下所示：

```
@Override
public boolean equals(Object object) {
    ..............
    for (int i = 0; i < updateList.size(); i++) {
      Object thisObject = updateList.get(i);
      Object thatObject = cacheKey.updateList.get(i);
      if (!ArrayUtil.equals(thisObject, thatObject)) {
        return false;
      }
    }
    return true;
}
```

除去hashcode、checksum和count的比较外，只要updatelist中的元素一一对应相等，那么就可以认为是CacheKey相等。只要两条SQL的下列五个值相同，即可以认为是相同的SQL。

> “ Statement Id + Offset + Limmit + Sql + Params

BaseExecutor的query方法继续往下走，代码如下所示：

```
list = resultHandler == null ? (List<E>) localCache.getObject(k
ey) : null;
if (list != null) {
    // 这个主要是处理存储过程用的。
    handleLocallyCachedOutputParameters(ms, key, parameter, bou
ndSql);
    } else {
    list = queryFromDatabase(ms, parameter, rowBounds, resultHa
ndler, key, boundSql);
}
```

如果查不到的话，就从数据库查，在 `queryFromDatabase` 中，会对 `localcache` 进行写入。

在 `query` 方法执行的最后，会判断一级缓存级别是否是 `STATEMENT` 级别，如果是的话，就清空缓存，这也就是 `STATEMENT` 级别的一级缓存无法共享 `localCache` 的原因。代码如下所示：

```
if (configuration.getLocalCacheScope() == LocalCacheScope.STATE
MENT) {
        clearLocalCache();
}
```

在源码分析的最后，我们确认一下，如果是 `insert/delete/update` 方法，缓存就会刷新的原因。

`SqlSession` 的 `insert` 方法和 `delete` 方法，都会统一走 `update` 的流程，代码如下所示：

```
@Override
public int insert(String statement, Object parameter) {
    return update(statement, parameter);
  }
   @Override
  public int delete(String statement) {
    return update(statement, null);
}
```

`update` 方法也是委托给了 `Executor` 执行。 `BaseExecutor` 的执行方法如下所示：

```
@Override
public int update(MappedStatement ms, Object parameter) throws
 SQLException {
    ErrorContext.instance().resource(ms.getResource()).activity
("executing an update").object(ms.getId());
    if (closed) {
      throw new ExecutorException("Executor was closed.");
    }
    clearLocalCache();
    return doUpdate(ms, parameter);
}
```

每次执行 `update` 前都会清空 `localCache` 。

至此，一级缓存的工作流程讲解以及源码分析完毕。

总结

- MyBatis一级缓存的生命周期和SqlSession一致。
- MyBatis一级缓存内部设计简单，只是一个没有容量限定的HashMap，在缓存的功能性上有所欠缺。
- MyBatis的一级缓存最大范围是SqlSession内部，有多个SqlSession或者分布式的环境下，数据库写操作会引起脏数据，建议设定缓存级别为Statement。

二级缓存

二级缓存介绍

在上文中提到的一级缓存中，其最大的共享范围就是一个SqlSession内部，如果多个SqlSession之间需要共享缓存，则需要使用到二级缓存。开启二级缓存后，会使用CachingExecutor装饰Executor，进入一级缓存的查询流程前，先在CachingExecutor进行二级缓存的查询，具体的工作流程如下所示。

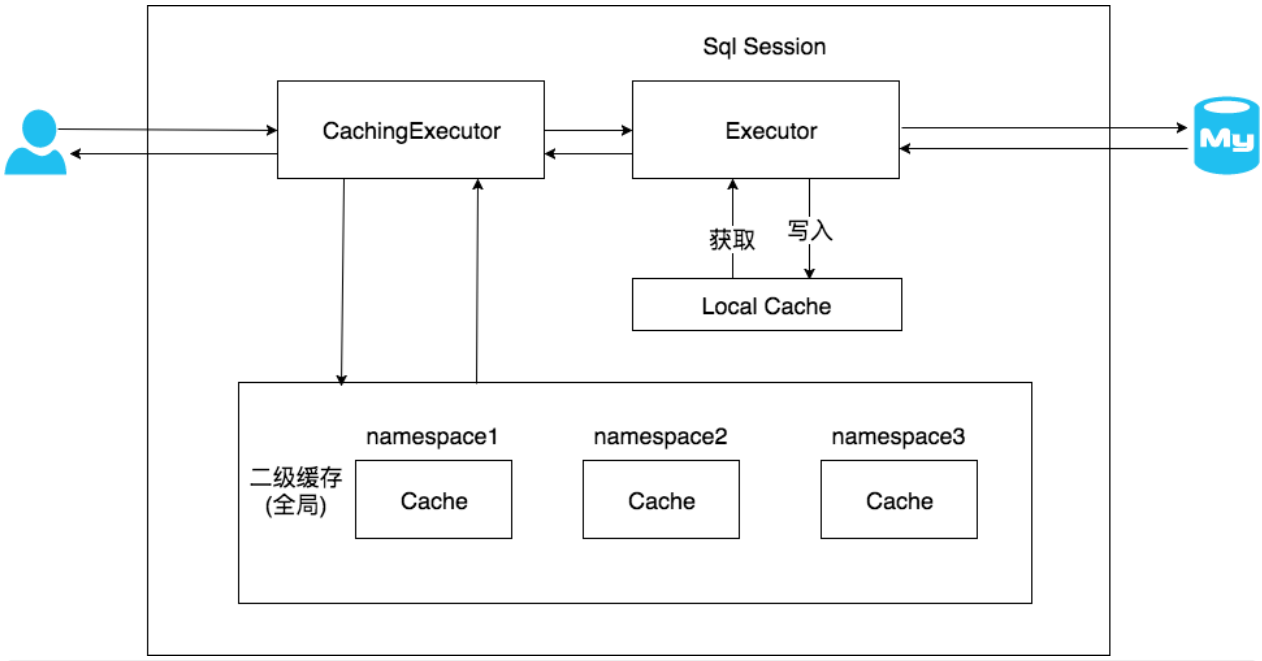
二级缓存开启后，同一个namespace下的所有操作语句，都影响着同一个Cache，即二级缓存被多个SqlSession共享，是一个全局的变量。

当开启缓存后，数据的查询执行的流程就是 二级缓存 -> 一级缓存 -> 数据库。

二级缓存配置

要正确的使用二级缓存，需完成如下配置的。

- 在MyBatis的配置文件中开启二级缓存。




```
<setting name="cacheEnabled" value="true"/>
```

- 在MyBatis的映射XML中配置cache或者 cache-ref 。

cache标签用于声明这个namespace使用二级缓存，并且可以自定义配置。

```
<cache/>
```

- `type`：cache使用的类型，默认是`PerpetualCache`，这在一级缓存中提到过。
- `eviction`： 定义回收的策略，常见的有FIFO，LRU。
- `flushInterval`： 配置一定时间自动刷新缓存，单位是毫秒。
- `size`： 最多缓存对象的个数。
- `readOnly`： 是否只读，若配置可读写，则需要对应的实体类能够序列化。
- `blocking`： 若缓存中找不到对应的key，是否会一直blocking，直到有对应的数据进入缓存。

`cache-ref`代表引用别的命名空间的Cache配置，两个命名空间的操作使用的是同一个Cache。

```
<cache-ref namespace="mapper.StudentMapper"/>
```

二级缓存实验

接下来我们通过实验，了解MyBatis二级缓存在使用上的一些特点。

在本实验中，id为1的学生名称初始化为点点。

实验1

测试二级缓存效果，不提交事务，`sqlSession1`查询完数据后，`sqlSession2`相同的查询是否会从缓存中获取数据。

```
@Test
public void testCacheWithoutCommitOrClose() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);

        StudentMapper studentMapper = sqlSession1.getMapper(Stu
dentMapper.class);
        StudentMapper studentMapper2 = sqlSession2.getMapper(St
udentMapper.class);

        System.out.println("studentMapper读取数据: " + studentMa
pper.getStudentById(1));
        System.out.println("studentMapper2读取数据: " + studentM
apper2.getStudentById(1));
}
```

执行结果:

```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 点点, 16
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 点点, 16
DEBUG [main] - <==      Total: 1
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
```

我们可以看到，当 `sqlsession` 没有调用 `commit()` 方法时，二级缓存并没有起到作用。

实验2

测试二级缓存效果，当提交事务时， `sqlSession1` 查询完数据后， `sqlSession2` 相同的查询是否会从缓存中获取数据。

```
@Test
public void testCacheWithCommitOrClose() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);

        StudentMapper studentMapper = sqlSession1.getMapper(Stu
dentMapper.class);
        StudentMapper studentMapper2 = sqlSession2.getMapper(St
udentMapper.class);

        System.out.println("studentMapper读取数据: " + studentMa
pper.getStudentById(1));
        sqlSession1.commit();
        System.out.println("studentMapper2读取数据: " + studentM
apper2.getStudentById(1));
}
```

从图上可知， `sqlsession2` 的查询，使用了缓存，缓存的命中率是0.5。

```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 点点, 16
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.5
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
```

实验3

测试 `update` 操作是否会刷新该 `namespace` 下的二级缓存。

```
@Test
public void testCacheWithUpdate() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);
        SqlSession sqlSession3 = factory.openSession(true);

        StudentMapper studentMapper = sqlSession1.getMapper(Stu
dentMapper.class);
        StudentMapper studentMapper2 = sqlSession2.getMapper(St
udentMapper.class);
        StudentMapper studentMapper3 = sqlSession3.getMapper(St
udentMapper.class);

        System.out.println("studentMapper读取数据: " + studentMa
pper.getStudentById(1));
        sqlSession1.commit();
        System.out.println("studentMapper2读取数据: " + studentM
apper2.getStudentById(1));

        studentMapper3.updateStudentName("方方",1);
        sqlSession3.commit();
        System.out.println("studentMapper2读取数据: " + studentM
apper2.getStudentById(1));
}
```

```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 点点, 16
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.5
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
DEBUG [main] - ==>  Preparing: UPDATE student SET name = ? WHERE id = ?
DEBUG [main] - ==> Parameters: 方方(String), 1(Integer)
DEBUG [main] - <==    Updates: 1
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.3333333333333333
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 方方, 16
DEBUG [main] - <==      Total: 1
studentMapper2读取数据: StudentEntity{id=1, name='方方', age=16, className='null'}
```

更新后，缓存被刷新，之后相同的查询走了数据库

我们可以看到，在 `sqlSession3` 更新数据库，并提交事务后，`sqlsession2` 的 `StudentMapper namespace` 下的查询走了数据库，没有走Cache。

实验4

验证MyBatis的二级缓存不适应用于映射文件中存在多表查询的情况。

通常我们会为每个单表创建单独的映射文件，由于MyBatis的二级缓存是基于 namespace 的，多表查询语句所在的 namspace 无法感应到其他 namespace 中的语句对多表查询中涉及的表进行的修改，引发脏数据问题。

```
@Test
public void testCacheWithDiffererntNamespace() throws Exception
{
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);
        SqlSession sqlSession3 = factory.openSession(true);

        StudentMapper studentMapper = sqlSession1.getMapper(Stu
dentMapper.class);
        StudentMapper studentMapper2 = sqlSession2.getMapper(St
udentMapper.class);
        ClassMapper classMapper = sqlSession3.getMapper(ClassMa
pper.class);

        System.out.println("studentMapper读取数据: " + studentMa
pper.getStudentByIdWithClassInfo(1));
        sqlSession1.close();
        System.out.println("studentMapper2读取数据: " + studentM
apper2.getStudentByIdWithClassInfo(1));

        classMapper.updateClassName("特色一班",1);
        sqlSession3.commit();
        System.out.println("studentMapper2读取数据: " + studentM
apper2.getStudentByIdWithClassInfo(1));
}
```

执行结果：

```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT s.id,s.name,s.age,class.name as className FROM classroom c JOIN student s ON c.student_id = s.id JOI
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age, className
TRACE [main] - <==        Row: 1, 点点, 16, 一班
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='一班'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.5
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='一班'}
DEBUG [main] - ==>  Preparing: UPDATE class SET name = ? WHERE id = ?
DEBUG [main] - ==> Parameters: 特色一班(String), 1(Integer)
DEBUG [main] - <==    Updates: 1
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.6666666666666666
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='一班'}   缓存中存的仍然是旧的className
```

在这个实验中，我们引入了两张新的表，一张class，一张classroom。class中保存了班级的id和班级名，classroom中保存了班级id和学生id。我们在 StudentMapper 中增加了一个查询方法 getStudentByIdWithClassInfo ，用于查询学生所在的班级，涉及到多表查询。在 ClassMapper 中添加了 updateClassName ，根据班级id更新班级名的操作。

当 sqlsession1 的 studentmapper 查询数据后，二级缓存生效。保存在StudentMapper的namespace下的cache中。当 sqlSession3 的 classMapper 的 updateClassName 方法对class表进行更新时， updateClassName 不属于 StudentMapper 的 namespace ，所以 StudentMapper 下的cache没有感应到变化，没有刷新缓存。当 StudentMapper 中同样的查询再次发起时，从缓存中读取了脏数据。

实验5

为了解决实验4的问题呢，可以使用Cache ref，让`ClassMapper`引用`StudenMapper`命名空间，这样两个映射文件对应的SQL操作都使用的是同一块缓存了。

执行结果：

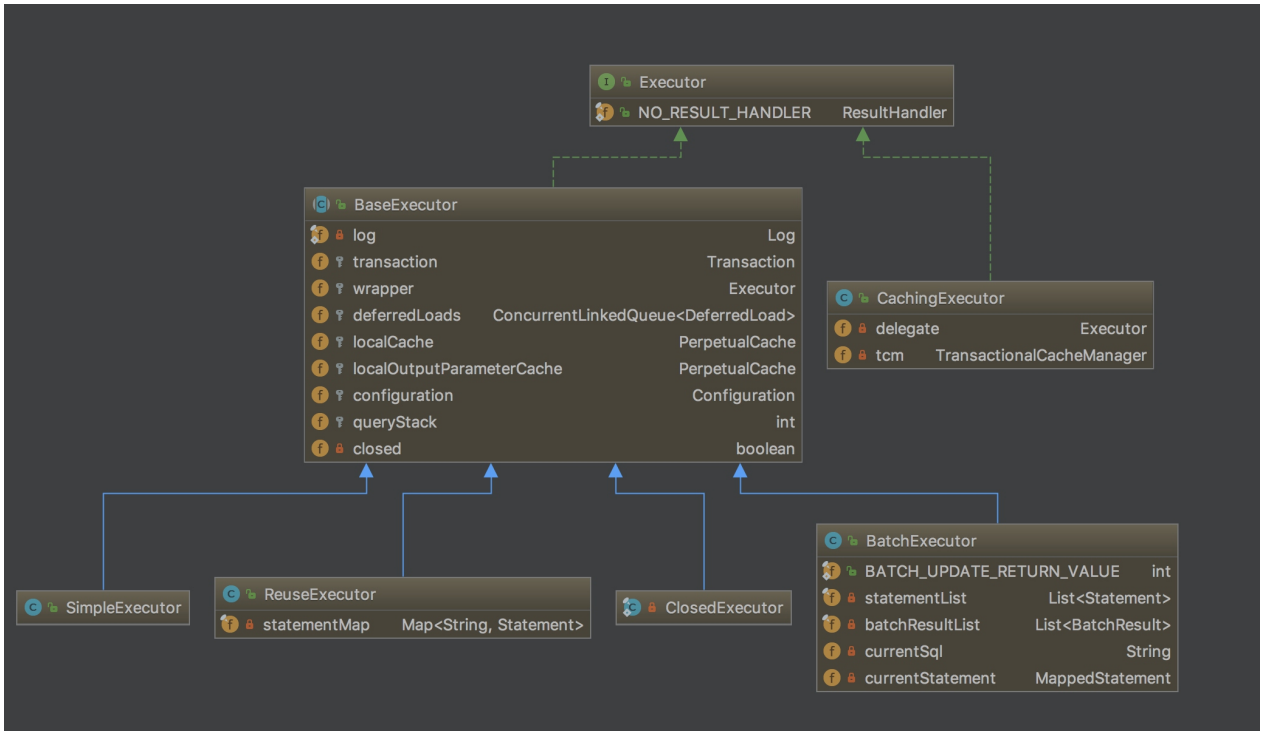
```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT s.id,s.name,s.age,class.name as className FROM classroom c JOIN student s ON c.student_id = s.id JOIN
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age, className
TRACE [main] - <==        Row: 1, 点点, 16, 一班
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='一班'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.5
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='一班'}
DEBUG [main] - ==>  Preparing: UPDATE class SET name = ? WHERE id = ?
DEBUG [main] - ==> Parameters: 特色一班(String), 1(Integer)
DEBUG [main] - <==    Updates: 1
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.3333333333333333
DEBUG [main] - ==>  Preparing: SELECT s.id,s.name,s.age,class.name as className FROM classroom c JOIN student s ON c.student_id = s.id JOIN
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age, className
TRACE [main] - <==        Row: 1, 点点, 16, 特色一班
DEBUG [main] - <==      Total: 1
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='特色一班'}
```

刷新了共同的缓存，后续的select走了数据库

不过这样做的后果是，缓存的粒度变粗了，多个`Mapper namespace`下的所有操作都会对缓存使用造成影响。

二级缓存源码分析

MyBatis二级缓存的工作流程和前文提到的一级缓存类似，只是在一级缓存处理前，用`CachingExecutor`装饰了`BaseExecutor`的子类，在委托具体职责给`delegate`之前，实现了二级缓存的查询和写入功能，具体类关系图如下图所示。



源码分析

源码分析从`CachingExecutor`的`query`方法展开，源代码走读过程中涉及到的知识点较多，不能一一详细讲解，读者朋友可以自行查询相关资料来学习。

`CachingExecutor` 的 `query` 方法，首先会从 `MappedStatement` 中获得在配置初始化时赋予的Cache。

```
Cache cache = ms.getCache();
```

本质上是装饰器模式的使用，具体的装饰链是：

> SynchronizedCache -> LoggingCache -> SerializedCache -> LruCache -> PerpetualCache。

以下是具体这些Cache实现类的介绍，他们的组合为Cache赋予了不同的能力。

- `SynchronizedCache`：同步Cache，实现比较简单，直接使用synchronized修饰方法。
- `LoggingCache`：日志功能，装饰类，用于记录缓存的命中率，如果开启了DEBUG模式，则会输出命中率日志。
- `SerializedCache`：序列化功能，将值序列化后存到缓存中。该功能用于缓存返回一份实例的Copy，用于保存线程安全。
- `LruCache`：采用了Lru算法的Cache实现，移除最近最少使用的Key/Value。
- `PerpetualCache`： 作为为最基础的缓存类，底层实现比较简单，直接使用了HashMap。

然后是判断是否需要刷新缓存，代码如下所示：

```
flushCacheIfRequired(ms);
```

在默认的设置中 `SELECT` 语句不会刷新缓存， `insert/update/delte` 会刷新缓存。进入该方法。代码如下所示：

```
private void flushCacheIfRequired(MappedStatement ms) {
    Cache cache = ms.getCache();
    if (cache != null && ms.isFlushCacheRequired()) {
      tcm.clear(cache);
    }
}
```

MyBatis的 `CachingExecutor` 持有了 `TransactionalCacheManager`，即上述代码中的tcm。

`TransactionalCacheManager` 中持有了一个Map，代码如下所示：

```
private Map<Cache, TransactionalCache> transactionalCaches = ne
w HashMap<Cache, TransactionalCache>();
```

这个Map保存了Cache和用 `TransactionalCache` 包装后的Cache的映射关系。

`TransactionalCache` 实现了Cache接口， `CachingExecutor` 会默认使用他包装初始生成的Cache，作用是如果事务提交，对缓存的操作才会生效，如果事务回滚或者不提交事务，则不对缓存产生影响。

在 `TransactionalCache` 的clear，有以下两句。清空了需要在提交时加入缓存的列表，同时设定提交时清空缓存，代码如下所示：

```
@Override
public void clear() {
        clearOnCommit = true;
        entriesToAddOnCommit.clear();
}
```

`CachingExecutor` 继续往下走， `ensureNoOutParams` 主要是用来处理存储过程的，暂时不用考虑。

```
if (ms.isUseCache() && resultHandler == null) {
        ensureNoOutParams(ms, parameterObject, boundSql);
```

之后会尝试从tcm中获取缓存的列表。

```
List<E> list = (List<E>) tcm.getObject(cache, key);
```

在 `getObject` 方法中，会把获取值的职责一路传递，最终到 `PerpetualCache` 。如果没有查到，会把key加入Miss集合，这个主要是为了统计命中率。

```
Object object = delegate.getObject(key);
if (object == null) {
        entriesMissedInCache.add(key);
}
```

`CachingExecutor` 继续往下走，如果查询到数据，则调用 `tcm.putObject` 方法，往缓存中放入值。

```
if (list == null) {
        list = delegate.<E> query(ms, parameterObject, rowBound
s, resultHandler, key, boundSql);
        tcm.putObject(cache, key, list); // issue #578 and #116
}
```

tcm的 `put` 方法也不是直接操作缓存，只是在把这次的数据和key放入待提交的Map中。

```
@Override
public void putObject(Object key, Object object) {
    entriesToAddOnCommit.put(key, object);
}
```

从以上的代码分析中，我们可以明白，如果不调用 `commit` 方法的话，由于 `TranscationalCache` 的作用，并不会对二级缓存造成直接的影响。因此我们看看 `Sqlsession` 的 `commit` 方法中做了什么。代码如下所示：

```
@Override
public void commit(boolean force) {
    try {
      executor.commit(isCommitOrRollbackRequired(force));
```

因为我们使用了CachingExecutor，首先会进入CachingExecutor实现的commit方法。

```
@Override
public void commit(boolean required) throws SQLException {
    delegate.commit(required);
    tcm.commit();
}
```

会把具体commit的职责委托给包装的 `Executor` 。主要是看下 `tcm.commit()` ，tcm最终又会调用到 `TrancationalCache` 。

```
public void commit() {
    if (clearOnCommit) {
      delegate.clear();
    }
    flushPendingEntries();
    reset();
}
```

看到这里的 `clearOnCommit` 就想起刚才 `TrancationalCache` 的 `clear` 方法设置的标志位，真正的清理Cache是放到这里来进行的。具体清理的职责委托给了包装的Cache类。之后进入 `flushPendingEntries` 方法。代码如下所示：

```
private void flushPendingEntries() {
    for (Map.Entry<Object, Object> entry : entriesToAddOnCommi
t.entrySet()) {
      delegate.putObject(entry.getKey(), entry.getValue());
    }
    ................
}
```

在 `flushPending` Entries中，将待提交的Map进行循环处理，委托给包装的Cache类，进行 `putObject` 的操作。

后续的查询操作会重复执行这套流程。如果是 `insert|update|delete` 的话，会统一进入 `CachingExecutor` 的 `update` 方法，其中调用了这个函数，代码如下所示：

```
private void flushCacheIfRequired(MappedStatement ms)
```

在二级缓存执行流程后就会进入一级缓存的执行流程，因此不再赘述。

总结

- MyBatis的二级缓存相对于一级缓存来说，实现了 `SqlSession` 之间缓存数据的共享，同时粒度更加的细，能够到 `namespace` 级别，通过Cache接口实现类不同的组合，对Cache的可控性也更强。
- MyBatis在多表查询时，极大可能会出现脏数据，有设计上的缺陷，安全使用二级缓存的条件比较苛刻。

- 在分布式环境下，由于默认的MyBatis Cache实现都是基于本地的，分布式环境下必然会出现读取到脏数据，需要使用集中式缓存将MyBatis的Cache接口实现，有一定的开发成本，直接使用Redis、Memcached等分布式缓存可能成本更低，安全性也更高。

全文总结

本文对介绍了MyBatis一二级缓存的基本概念，并从应用及源码的角度对MyBatis的缓存机制进行了分析。最后对MyBatis缓存机制做了一定的总结，个人建议MyBatis缓存特性在生产环境中进行关闭，单纯作为一个ORM框架使用可能更为合适。

作者简介

- 凯伦，美团点评后端研发工程师，2016年毕业于上海海事大学，现从事美团点评餐饮平台相关的开发工作。

招聘信息

美团点评点餐事业部期待你的加入，上海在招岗位：Java后台，数据开发，前端，QA，产品，产品运营，商业分析等。内推简历邮箱：weiyanping#meituan.com