# 第5章 缓存和存储过程

在日常生活中，需要经常查询信息。如果每次查询都要做一些无用的重复工作，非常浪费时间。故产生如下需求：同样的查询，如果不用每次都从数据库获取结果该有多好！缓存由此出现，既可以提升数据访问效率，又可以降低数据库压力。正如其他 ORM 框架，MyBatis 亦提供缓存机制，分别是一级缓存和二级缓存。

存储过程是为了完成特定需求的大段 SQL 语句，通常会在大型的数据库系统中使用。存储过程具有运行速度快、稳定的特点，能够在一定程度上减轻客户端压力。

## 5.1 一级缓存

一级缓存又名本地缓存(local cache)，默认是开启状态，即用户无须额外配置。根据其作用域，细分为基于 Session 的缓存和基于 Statement 的缓存。这里将以基于 Session 的缓存为例讲解。

1.1.3 节曾讲过，SqlSession 会通过 Configuration 创建 Executor，并将具体的任务交给 Executor 执行。既然是本地缓存，肯定不适合用全局的属性存储，结合 1.4 节的图 1-3 可知，剩下的选择是将缓存放入 Executor 维护。接着详细看看 Executor 接口(与先前提及的宽泛的 Executor 不同)与其实现类之间的关系图，如图 5-1 所示。注：图中仅列出部分重点属性、方法和参数。

Executor 是一个接口，共有两个直接实现类：CachingExecutor 和 BaseExecutor(抽象类)。CachingExecutor 主要用于二级缓存，故重点关注 BaseExecutor。BaseExecutor 有一个名为 localCache 的属性，由名字可知，是用于本地缓存的，localCache 类型为 PerpetualCache(cache 接口的实现类)。PerpetualCache 类包含一个 Map<Object，Object>类型的 cache 属性，由此可知，一级缓存使用 Map 结构存储。BaseExecutor 中定义了与数据库交互的方法，包括查询、更新、提交数据至数据库。其具有 4 个子类：SimpleExecutor、BatchExecutor、ReuseExecutor 和

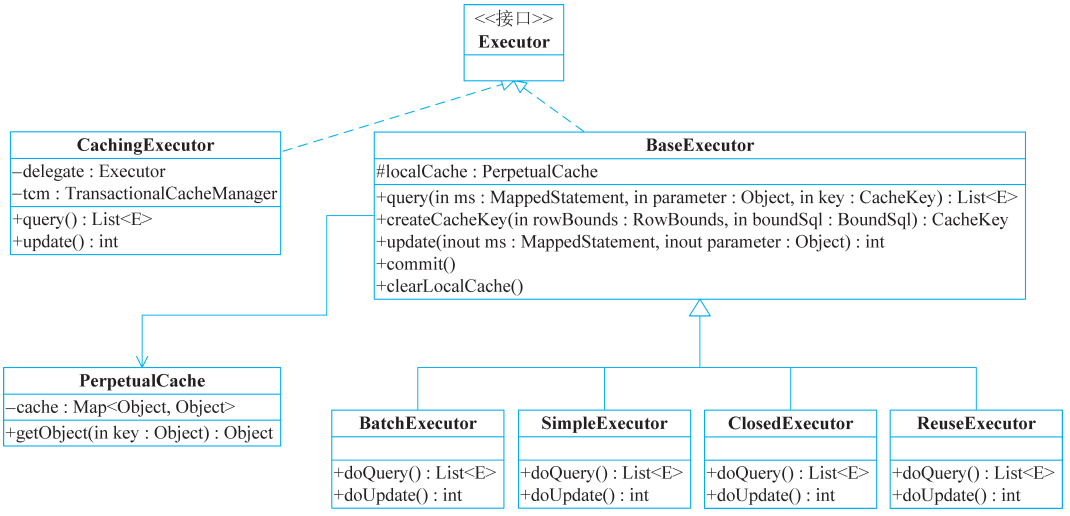


图 5-1 Executor 的类关系图

ClosedExecutor，分别实现与数据库不同需求的交互。

了解完整体结构，接着学习一级缓存的工作流程。查询有两种结果：①击中缓存；②未击中缓存。击中缓存的工作流程如图 5-2 所示。

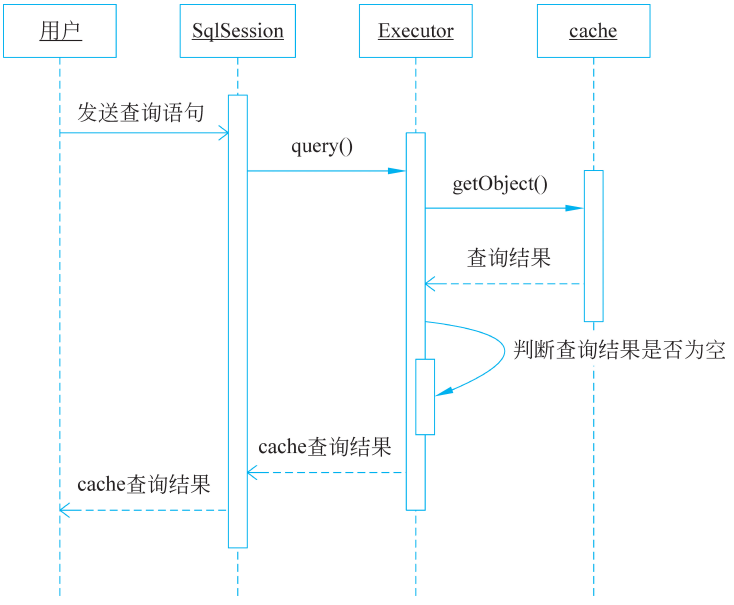


图 5-2 查询击中一级缓存时序图

用户根据自身需求发送查询请求，程序接收到用户的查询请求，判断需要执行的方法，SqlSession 职责下放至 Executor，即通过调用 Executor 的 query 方法令 Executor 执行具体任务。Executor 则通过调用 cache 的 getObject 方法获取缓存中的数据，该方法需要一个 CacheKey 类型的参数 key。key 由 MappedStatement 的 ID、SQL 的 offset、SQL 的 limit、SQL 语句以及相关参数组成。cache 具体指的是其中一个实现类 PerpetualCache，其中的 getObject 方法根据键获取 HashMap 中存放的缓存数据，并返回结果至 Executor。

Executor 判断返回数据是否为空，不为空则处理输出参数，接着判断缓存级别，若是 statement 级别，则清空缓存，否则将获取到的缓存数据返回至 SqlSession，再由 SqlSession 返回给用户。

如果 Executor 判断 cache 返回数据为空，则会调用 queryFromDatabase 方法，从数据库中查询数据。数据库将查询结果返回，Executor 调用 cache 的 putObject 方法，将数据放入缓存，即存入 localCache，接着将查询结果反馈给 SqlSession，再由 SqlSession 返回给用户。未击中缓存的工作流程如图 5-3 所示。

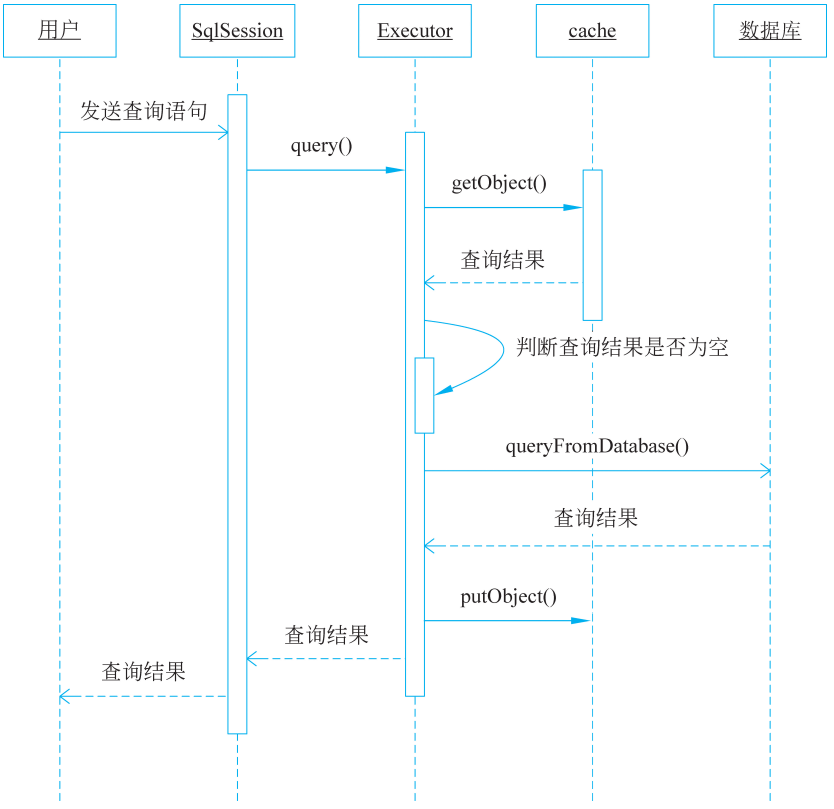


图 5-3　查询未击中一级缓存时序图

针对数据的操作包含查询、增加、修改和删除。Executor 执行除查询以外的 3 种操作时，都会转换为更新操作。且执行上述 3 种操作时，需要提交事务。故当有增加、删除、修改、提交数据，关闭 Session 操作时，缓存会失效。Session 关闭后，PerpetualCache 对象不可用，其余情况下则只清空 PerpetualCache 对象。

在默认情况下，开启的一级缓存为 SqlSession 级别，即同一 SqlSession 多次执行同一 SQL，直接从内存中取到缓存结果，而无须查询数据库。笔者将通过 4 个例子讲解。为了便于观察缓存效果，需添加日志配置，通过输出日志确定其是否击中缓存。在 XML 配置文件 mybatis-config.xml 中添加日志配置项，代码如下。

```
<settings>
    <setting name="logImpl" value="STDOUT_LOGGING" />
</settings>
```

具体配置信息含义可回顾 2.1.2 节的表 2-1。

## 5.1.1 相同 SqlSession

输出日志配置完毕，接着编写测试案例。首先是在相同 SqlSession 的前提下，分别从查询和更新两方面测试的案例。

### 1. 查询

例 5-1：演示根据选课 ID 查询选课记录的方式，其中 SqlSession 和 Mapper 使用的是同一个，连续执行的两条查询语句一致。需要改动的文件为 CcoursesMapper.java、CcoursesMapper.xml 和 TestMB.java。其余文件均沿用例 2-25 的即可。

CcoursesMapper.java 的代码如下。

```
package com.imut.mapper;
import com.imut.pojo.Ccourses;
public interface CcoursesMapper {
    Ccourses selectCcoursesByid(Integer id);
}
```

CcoursesMapper.xml 的代码如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.
org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.imut.mapper.CcoursesMapper">
    <select id="selectCcoursesByid" resultType="ccourses">
        select * from ccourses where
        <if test="id!=null">
            id = #{id,jdbcType=INTEGER}
        </if>
    </select>
</mapper>
```

TestMB.java 的代码如下。

```
package com.imut.test;
import com.imut.mapper.CcoursesMapper;
import com.imut.pojo.*;
import com.imut.utils.MyBatisUtil;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.junit.Test;
public class TestMB {
    @Test
    public void testCcoursesMapper(){
        SqlSessionFactory ssf = null;
        SqlSession session = null;
        int id = 6;
        try {
```

```
            ssf = new MyBatisUtil().getSqlSessionFactory();
            session = ssf.openSession();
            CcoursesMapper ccoursesMapper = session.getMapper(CcoursesMapper.class);
            Ccourses ccourses = ccoursesMapper.selectCcoursesByid(id);
            System.out.println(ccourses);
            Ccourses ccourses1 = ccoursesMapper.selectCcoursesByid(id);
            System.out.println(ccourses1);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            session.close();
        }
    }
}
```

上述代码实现根据选课 ID 查询选课记录的功能，且查询方法输入 ID 值是通过一个变量传入的，确保一致性。Mapper 的同一个方法连续调用两次，然后输出查询数据。运行结果如图 5-4 所示。

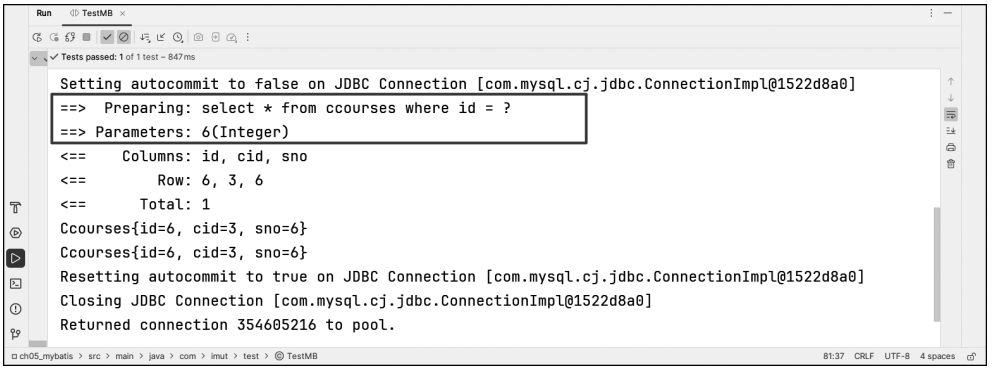


图 5-4　一级缓存例 5-1 运行结果

由图 5-4 可知，只向数据库发送了一次 SQL 查询请求，即框住部分。第二次获取的数据并未与数据库交互，而是从缓存中得到。可以看出，两次查询获取的数据一致。

因插入、删除操作最终都会转化为调用 update 方法，即逻辑是一致的，故笔者只以更新操作为例。

### 2. 更新

例 5-2：演示更新操作令缓存失效的方式，其中 SqlSession 和 Mapper 都使用的是同一个。执行两条相同的查询语句，两次查询之间执行一次更新语句。需要改动的文件为 Ccourses.java、CcoursesMapper.java、CcoursesMapper.xml 和 TestMB.java。其余文件均沿用例 2-25 的即可。

Ccourses.java 的新增代码如下。

```
public Ccourses(Integer id, Integer cid, Integer sno) {
    this.id = id;
    this.cid = cid;
```

```
    this.sno = sno;
}
```

为了满足不同需求的赋值，为 Ccourses 类添加 3 参构造方法。

CcoursesMapper.java 的新增代码如下。

```
int updateCcoursesByid(Ccourses ccourses);
```

CcoursesMapper.xml 的新增代码如下。

```
<update id="updateCcoursesByid" parameterType="com.imut.pojo.Ccourses">
    update ccourses
    set cid = #{cid,jdbcType=INTEGER},
    sno = #{sno,jdbcType=INTEGER}
    where id = #{id,jdbcType=INTEGER}
</update>
```

TestMB.java 的代码如下。

```
package com.imut.test;
import com.imut.mapper.CcoursesMapper;
import com.imut.pojo.*;
import com.imut.utils.MyBatisUtil;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.junit.Test;
public class TestMB {
    @Test
    public void testCcoursesMapper(){
        SqlSessionFactory ssf = null;
        SqlSession session = null;
        int id = 2;
        try {
            ssf = new MyBatisUtil().getSqlSessionFactory();
            session = ssf.openSession();
            CcoursesMapper ccoursesMapper = session.getMapper(CcoursesMapper.class);
            Ccourses ccourses = ccoursesMapper.selectCcoursesByid(id);
            System.out.println(ccourses);
            int result = ccoursesMapper.updateCcoursesByid(new Ccourses(1, 2, 3));
            System.out.println(result);
            Ccourses ccourses1 = ccoursesMapper.selectCcoursesByid(id);
            System.out.println(ccourses1);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            session.close();
        }
    }
}
```

上述代码实现根据选课 ID 查询选课记录，更新选课记录，再次查询选课记录的功能，且查询方法输入 ID 值是通过一个变量传入的，确保一致性，更新选课记录的 ID 与查询选课记录的 ID 不一致。Mapper 的同一个查询方法调用两次，更新方法在两次查询方法中间调用一次，输出查询数据，数据更新影响条数。运行结果如图 5-5 所示。

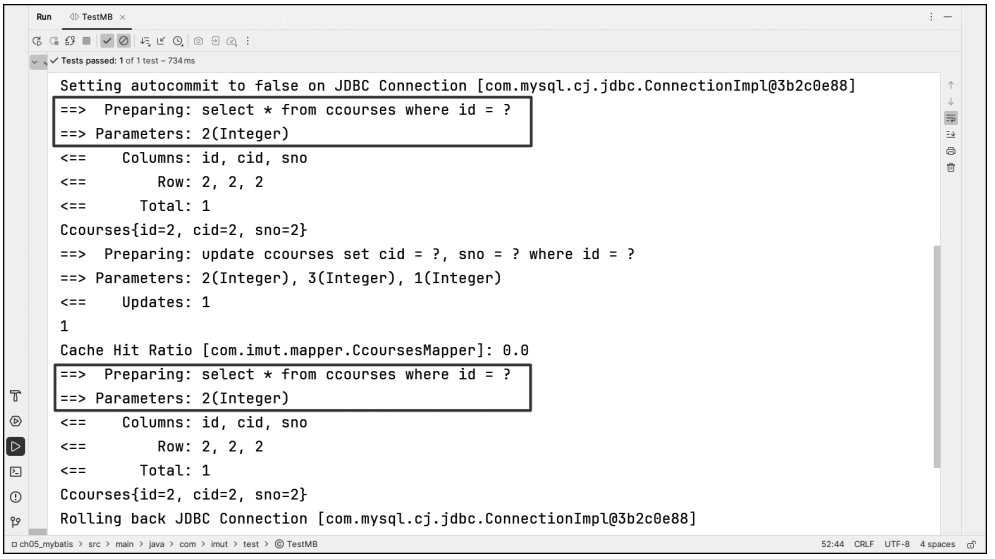


图 5-5　一级缓存例 5-2 运行结果

由图 5-5 可知，共向数据库发送了 3 次 SQL 查询请求，其中向数据库发送了 2 次 SQL 查询请求，即框住部分。也就意味着第 2 次并未击中缓存，而是从数据库中获得的。可以看出，尽管两次查询获取的数据一致，但因为中间执行了更新操作，故缓存被清空，所以第 2 次才需要和数据库交互。这里故意选用更新的 ID 和查询 ID 不一致，避免读者认为是因为更新了查询数据缓存才被清空。且这里并未执行 commit 操作，缓存亦被清空，再次印证无论更新操作是否提交，都将清空缓存。

既然默认开启的一级缓存是基于 SqlSession，接着列举不在同一个 SqlSession 的例子，依然涉及查询和更新两方面。

## 5.1.2　不同 SqlSession

### 1. 查询

例 5-3：演示根据选课 ID 查询选课记录的方式，其中 SqlSession 既包含相同的，又包含不同的，连续执行的两条查询语句一致。需要改动的文件为 TestMB.java。CcoursesMapper.java、CcoursesMapper.xml 文件内容详见例 5-2，其余文件均沿用例 2-25 的即可。

TestMB.java 的代码如下。

```
package com.imut.test;
import com.imut.mapper.CcoursesMapper;
import com.imut.pojo.*;
import com.imut.utils.MyBatisUtil;
```

```
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.junit.Test;
public class TestMB {
    @Test
    public void testCcoursesMapper(){
        SqlSessionFactory ssf = null;
        SqlSession session = null;
        SqlSession session1 = null;
        int id = 6;
        try {
            ssf = new MyBatisUtil().getSqlSessionFactory();
            session = ssf.openSession();
            CcoursesMapper ccoursesMapper = session.getMapper(CcoursesMapper.class);
            Ccourses ccourses = ccoursesMapper.selectCcoursesByid(id);
            System.out.println(ccourses);
            Ccourses ccourses1 = ccoursesMapper.selectCcoursesByid(id);
            System.out.println(ccourses1);
            session1 = ssf.openSession();
            CcoursesMapper ccoursesMapper2 = session1.getMapper(CcoursesMapper.class);
            Ccourses ccourses2 = ccoursesMapper2.selectCcoursesByid(id);
            System.out.println(ccourses2);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            session.close();
        }
    }
}
```

上述代码分别实现在相同 SqlSession 和不同 SqlSession 前提下，根据同样的选课 ID 查询选课记录的功能，且查询方法输入 ID 值是通过一个变量传入的，确保一致性。即在相同 SqlSession 的情况下连续执行两条一样的查询语句；在不同 SqlSession 的情况下执行与前两次查询相同的 SQL 语句。同样需要输出每次的查询数据。运行结果如图 5-6 所示。

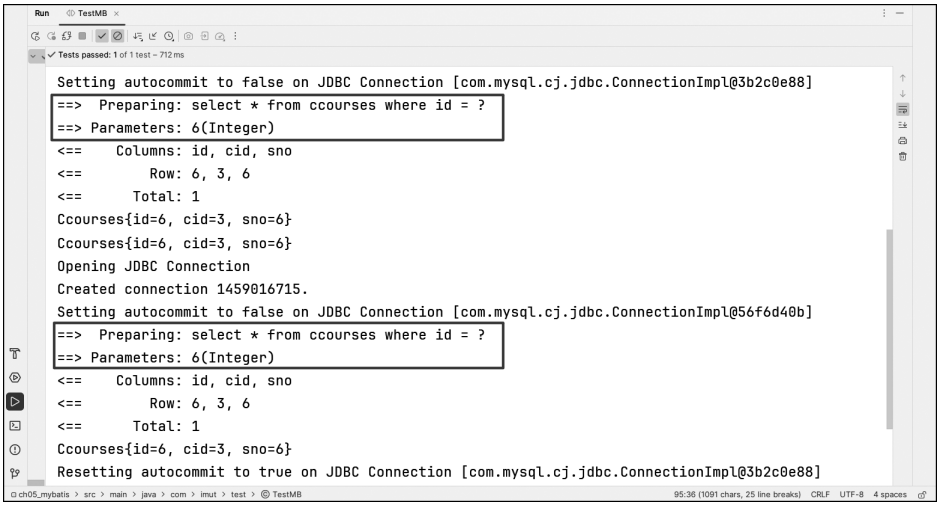


图 5-6　一级缓存例 5-3 运行结果

由图 5-6 可知，共向数据库发送了 2 次查询请求，即框住部分。这里共执行了 3 次查询请求，也就意味着某次请求击中缓存，是相同 SqlSession 的两次查询中的第 2 次。可以看出：在相同 SqlSession 的情况下，击中缓存；在不同 SqlSession 的情况下，缓存被清空。本例无论 SqlSession 是否相同，都不会影响查询的结果，3 次查询获取的数据一致。

尽管不同 SqlSession 的查询已经可以证明，一级缓存的作用范围是 SqlSession 内部，但这里依然有必要讲解更新的例子，不仅仅是为了说明一级缓存不同 SqlSession 会失效的问题。

### 2. 更新

**例 5-4**：演示不同 SqlSession 缓存失效的方式，其中 SqlSession 既涉及同一个，又涉及不同的，既包含查询操作，又包含更新操作。需要改动的文件为 TestMB.java。CcoursesMapper.java、CcoursesMapper.xml 文件内容详见例 5-2，其余文件均沿用例 2-25 的即可。

TestMB.java 的代码如下。

```
package com.imut.test;
import com.imut.mapper.CcoursesMapper;
import com.imut.pojo.*;
import com.imut.utils.MyBatisUtil;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.junit.Test;
public class TestMB {
    @Test
    public void testCcoursesMapper(){
        SqlSessionFactory ssf = null;
        SqlSession session = null;
        SqlSession session1 = null;
        int id = 4;
        try {
            ssf = new MyBatisUtil().getSqlSessionFactory();
            session = ssf.openSession();
            CcoursesMapper ccoursesMapper = session.getMapper(CcoursesMapper.class);
            Ccourses ccourses = ccoursesMapper.selectCcoursesByid(id);
            System.out.println(ccourses);
            Ccourses ccourses1 = ccoursesMapper.selectCcoursesByid(id);
            System.out.println(ccourses1);
            session1 = ssf.openSession(true);
            CcoursesMapper ccoursesMapper2 = session1.getMapper(CcoursesMapper.class);
            int result = ccoursesMapper2.updateCcoursesByid(new Ccourses(id, 3, 3));
            System.out.println(result);
            Ccourses ccourses3 = ccoursesMapper.selectCcoursesByid(id);
            System.out.println(ccourses3);
            Ccourses ccourses2 = ccoursesMapper2.selectCcoursesByid(id);
            System.out.println(ccourses2);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
```

```
            session.close();
        }
    }
}
```

上述代码实现根据选课 ID 查询选课记录，更新选课记录，再次查询选课记录的功能，且查询方法输入 ID 值和更新选课记录的 ID 是通过一个变量传入的，确保一致性。但更新操作是在不同的 Session 中执行的，即先在第 1 个 SqlSession 中连续执行两条相同的查询语句，接着在第 2 个 SqlSession 中执行一条更新语句，然后再使用第 1 个 SqlSession 执行查询语句，最后在第 2 个 SqlSession 中再执行同一条查询语句。即 Mapper 的同一个查询方法调用 4 次，更新方法在两次查询方法后调用 1 次，输出查询数据，数据更新影响条数。运行结果如图 5-7 所示。

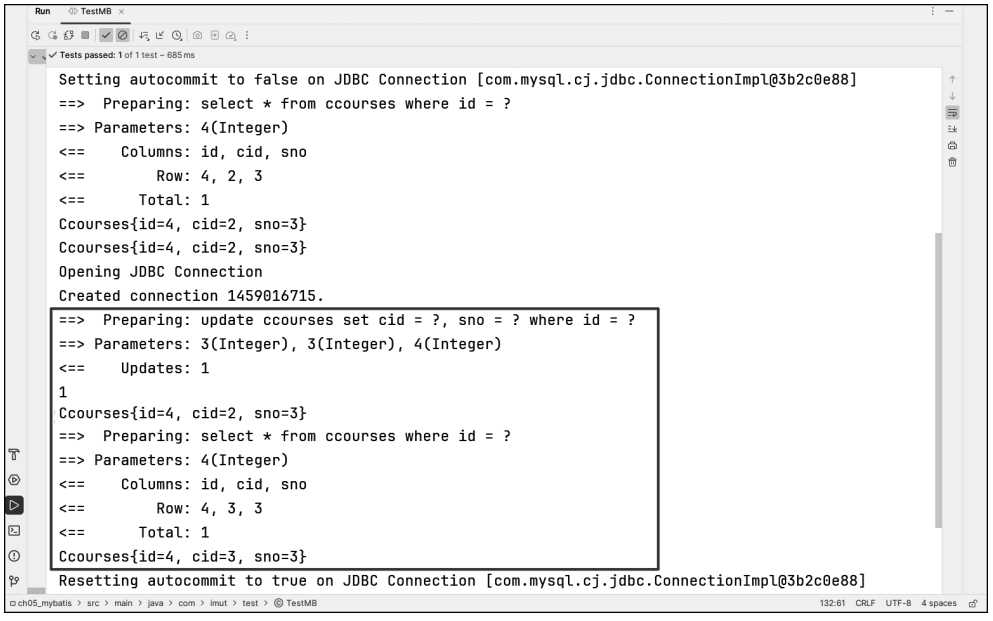


图 5-7　一级缓存例 5-4 运行结果

由图 5-7 可知，共向数据库发送了 3 次 SQL 查询请求，其中向数据库发送了 2 次 SQL 查询请求，而这里共执行了 5 次操作，也就意味着有二次查询击中缓存。由图中框住部分可以看出：两次查询获取的数据不一致，是因为更新操作是在第 2 个 SqlSession 中执行的，而使用第 1 个 SqlSession 的查询语句(在更新语句后面)击中了缓存，故读取的是缓存中的数据，也就是旧数据。第 2 个 SqlSession 的查询操作因为与数据库发生交互，故读取的是新数据。由例 5-4 可知，当使用多个 SqlSession 操作数据库时，可能出现脏数据，故需谨慎使用 Session 级别的一级缓存。

一级缓存支持 statement 级别，每次查询都会清空缓存，故可以在一定程度上解决脏数据问题。这里不再举例说明，感兴趣的读者可以自行尝试。使用前需在 mybatis-config.xml 中添加配置如下。

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