

1 我会更努力的去生活， 过好自己这一生。

--老洋

1、连接池POOLED和UNPOOLED技术

1.1、DataSource

```
public interface DataSource extends CommonDataSource, Wrapper {  
    /**  
     * <p>Attempts to establish a connection with the data source that  
     * this {@code DataSource} object represents.  
     *  
     * @return a connection to the data source  
     * @exception SQLException if a database access error occurs  
     * @throws java.sql.SQLTimeoutException when the driver has determined that  
     * timeout value specified by the {@code setLoginTimeout} method  
     * has been exceeded and has at least tried to cancel the  
     * current database connection attempt  
     */  
    Connection getConnection() throws SQLException;
```

1.2、PooledDataSource

```
    * @author Clinton Begin  
    */  
    public class PooledDataSource implements DataSource {  
        private static final Log log = LogFactory.getLog(PooledDataSource.class);  
        private final PoolState state = new PoolState(dataSource this);  
        private final UnpooledDataSource dataSource;
```

```
        @Override  
        public Connection getConnection() throws SQLException {  
            return popConnection(dataSource.getUsername(), dataSource.getPassword()).getProxyConnection();  
        }  
        // 也是传入：账号密码  
        @Override  
        public Connection getConnection(String username, String password) throws SQLException {  
            return popConnection(username, password).getProxyConnection();  
        }
```

```
        private PooledConnection popConnection(String username, String password) throws SQLException {  
            boolean countedWait = false;  
            PooledConnection conn = null;  
            long t = System.currentTimeMillis();  
            int localBadConnectionCount = 0;
```

```

while (conn == null) {
    synchronized (state) {
        if (!state.idleConnections.isEmpty()) {
            // Pool has available connection
            conn = state.idleConnections.remove(index 0);
            if (log.isDebugEnabled()) {
                log.debug("Checked out connection " + conn.getRealHashCode() + " from pool.");
            }
        } else {
            // Pool does not have available connection
            if (state.activeConnections.size() < poolMaximumActiveConnections) {
                // Can create new connection
                conn = new PooledConnection(dataSource.getConnection(), dataSource, this);
                if (log.isDebugEnabled()) {
                    log.debug("Created connection " + conn.getRealHashCode() + ".");
                }
            } else {
                // Cannot create new connection
                PooledConnection oldestActiveConnection = state.activeConnections.get(0);
                long longestCheckoutTime = oldestActiveConnection.getCheckoutTime();
                if (longestCheckoutTime > poolMaximumCheckoutTime) {
                    // Can claim overdue connection
                    state.claimedOverdueConnectionCount++;
                    state.accumulatedCheckoutTimeOfOverdueConnections += longestCheckoutTime;
                    state.accumulatedCheckoutTime += longestCheckoutTime;
                    state.activeConnections.remove(oldestActiveConnection);
                    if (!oldestActiveConnection.getRealConnection().getAutoCommit()) {
                        try {
                            oldestActiveConnection.getRealConnection().rollback();
                        } catch (SQLException e) {
                            /*
                             Just log a message for debug and continue to execute the following
                             statement like nothing happen.
                             Wrap the bad connection with a new PooledConnection, this will help
                             to not interrupt current executing thread and give current thread a
                             chance to join the next competition for another valid/good database
                             connection. At the end of this loop, bad {@link @conn} will be set as null.
                             */
                        }
                    }
                }
            }
        }
    }
}

```

如果此时：空闲的连接，还有

直接拿一个出来，注意：remove (0) 透露出是：集合存放

活动的连接数目 < 池中定义最大可（活动的数目）

就从池中，拿一个出来连接对象出来

池中：都是在活动的，没有空闲时
把最早成为（活动状态）的拿出去处理掉：来用

```

public class PoolState {

    protected PooledDataSource dataSource;

    protected final List<PooledConnection> idleConnections = new ArrayList<PooledConnection>();
    protected final List<PooledConnection> activeConnections = new ArrayList<>();
}

```

```

@author Neal Gafter
* @see Collection
* @see List
* @see LinkedList
* @see Vector
* @since 1.2
*/

public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;

    /**
     * Default initial capacity.
     */
    private static final int DEFAULT_CAPACITY = 10;
}

```

Object (java.lang)

AbstractCollection (java.util)

AbstractList (java.util)

ArrayList (java.util)

1 in CacheUtil (com.sun.javav)

1 in MethodNode (jdk.intern)

ArrayListWrapper in ProxyBu

FinalArrayList (com.sun.xml.li

RoleList (javax.management)

JsonArray (com.mysql.cj.xder

FinalArrayList (com.sun.xml.li

BakedArrayList (sun.swing)

FinalArrayList (com.sun.isack

Pack in Lister (com.sun.xml.in

AttributeList (javax.managen

HeaderList (com.sun.xml.Linte

RoleUnresolvedList (javax.ma

1.3、UnPooledDataSource

```

public class UnPooledDataSource implements DataSource {

    private ClassLoader driverClassLoader;
    private Properties driverProperties;
    private static Map<String, Driver> registeredDrivers = new ConcurrentHashMap<>()
}

```

```

@Override
public Connection getConnection() throws SQLException {
    return doGetConnection(username, password);
}

@Override
public Connection getConnection(String username, String password) throws SQLException {
    return doGetConnection(username, password);
}

```

```

private Connection doGetConnection(String username, String password) throws SQLException {
    Properties props = new Properties();
    if (driverProperties != null) {
        props.putAll(driverProperties);
    }
    if (username != null) {
        props.setProperty("user", username);
    }
    if (password != null) {
        props.setProperty("password", password);
    }
    return doGetConnection(props);
}

private Connection doGetConnection(Properties properties) throws SQLException {
    initializeDriver();
    Connection connection = DriverManager.getConnection(url, properties);
    configureConnection(connection);
    return connection;
}

```

弄了个Properties对象 props, 将username, password 放进去

```

private Connection doGetConnection(Properties properties) throws SQLException {
    initializeDriver();
    Connection connection = DriverManager.getConnection(url, properties);
    configureConnection(connection);
    return connection;
}

private synchronized void initializeDriver() throws SQLException {
    if (!registeredDrivers.containsKey(driver)) {
        Class<?> driverType;
        try {
            if (driverClassLoader != null) {
                driverType = Class.forName(driver, initialize, driverClassLoader);
            } else {
                driverType = Resources.classForName(driver);
            }
            // DriverManager requires the driver to be loaded via the system ClassLoader.
            // http://www.kfu.com/~nsayer/Java/dyn-jdbc.html
            Driver driverInstance = (Driver)driverType.newInstance();
            DriverManager.registerDriver(new DriverProxy(driverInstance));
            registeredDrivers.put(driver, driverInstance);
        } catch (Exception e) {
            // ...
        }
    }
}

```

如果Map中没有加载驱动就: 加载驱动

2、Mybatis的动态Sql

2.1、if标签

```

<!--根据传入参数条件查询-->
<select id="findUserByCondition" parameterType="user" resultMap="reMap">
    select * from user where 1 = 1
    <if test="userName != null">        <!--注意：非Sql语句：需要注意大小写-->
        and username = #{userName}
    </if>

    <if test="userSex != null">
        and sex = #{userSex}
    </if>
</select>

```

要自己添加: where 1 = 1, 不好

2.2、where标签

```

<!--根据传入参数条件查询-->
<select id="findUserByCondition" parameterType="user" resultMap="reMap">
    select * from user
    <where>
        <if test="userName != null">        <!--注意：非Sql语句：需要注意大小写-->
            and username = #{userName}
        </if>

        <if test="userSex != null">
            and sex = #{userSex}
        </if>
    </where>
</select>

```

采用where标签: 如果内部为空, 自动去掉where关键字

2.3、where标签

```

<!--根据: QueryVo中的: List<Integer> ids 进行: 子查询-->
<select id="findUserByInIds" parameterType="QueryVo" resultMap="reMap">
    select * from user

    <where>
        <if test="ids != null">
            <foreach collection="ids" open="id in(" close=")" item="uid" separator=",">
                #{uid}
            </foreach>
        </if>
    </where>
</select>

```

foreach实现遍历: open = "遍历前内容" close = "遍历后内容" item = "迭代对象" separator = "分隔符"
注意#{uid}为迭代对象

2.4、sql和include标签

```

<!--抽取重复的: Sql语句-->
<sql id="duplicateSqlEncapsulation">
    select * from user
</sql>

<!--查询操作-->
<select id="selAll" resultMap="reMap">
    <include refid="duplicateSqlEncapsulation">
    </include>
</select>

```

切记: 抽取内容不要添加; 防止出错

3、多表查询

- 1 示例：用户和账户
- 2 一个用户可以有：多个账户
- 3 一个账户只能属于一个用户（多个账户可以：属于一个用
- 4 户）
- 4 步骤
- 5 1、建立两张表：用户表， 账户类
- 6 让用户表（和）账户表之间具备一对多的关系， 需要
- 7 使用（外键）在（账户表）中添加
- 8 2、建立两个实体类：用户实体类能体现出一对多的关系。
- 9 3、建立两个配置文件
- 10 用户配置文件
- 11 账户的配置文件
- 12 4、实现配置
- 13 当我们（查询用户时），可以同时得到（用户）所包含
- 14 的（账户信息）
- 当我们（查询账户时），可以同时得到（账户）的所有
- 的（用户信息）

3.1、一对一（一个账户 --->>> 一个用户）

```
public class Account implements Serializable{
    private Integer id;
    private Integer uid;
    private Double money;

    //实现：一对一的查询（一个账户， 对应一个用户， 所以应该当使用：组合）
    private User user;    账户表中：组合了（用户对象）
```

<!--1、查询所有账户信息 + 各自用户信息（实现：一对一的查询）-->

```
<resultMap id="accountMap" type="account">
  <id property="id" column="aid"/>
  <result property="uid" column="uid"/>
  <result property="money" column="money"/>

  <association property="user" javaType="user">
    <id property="id" column="id" />
    <result property="username" column="username"/>
    <result property="birthday" column="birthday"/>
    <result property="sex" column="sex"/>
    <result property="address" column="address"/>
  </association>
</resultMap>
```

此处虽然：是所有的账户信息，但是会逐条匹配

```
<select id="findAll" resultMap="accountMap">
  { select u.*, a.id aid, a.uid, a.money from account a, user u
    where a.uid = u.id
  }
</select>
```

查询出：所有账户信息 + 用户信息

3.2、一对多（一个用户 --- >>> 账户）

```
public class User implements Serializable{
  private int id;
  private String username;
  private Date birthday;
  private String sex;
  private String address;

  //实现：一个用户，有多个账户（一对多），需要利用组合
  private List<Account> accounts;  用户表中：组合了（账户集合）
}
```

```

<!--1、查询所有用户信息 + 各个用户（拥有的账户）信息-->
<resultMap id="myUser" type="user">
  <id property="id" column="id"/>
  <result property="username" column="username"/>
  <result property="birthday" column="birthday"/>
  <result property="sex" column="sex"/>
  <result property="address" column="address"/>

  <collection property="accounts" ofType="account">
    <id property="id" column="aid" />
    <result property="uid" column="uid" />
    <result property="money" column="money" />
  </collection>
</resultMap>

<select id="selAll" resultMap="myUser">
  { select u.*, a.id aid, a.uid, a.money from user u
    left outer join account a
    on u.id = a.uid }
</select>

<!--通过id查询：某个人-->
<select id="selOneById" parameterType="int" resultType="user">
  select * from user where id = #{0}

```

mybatis会对所有内容：分割为（类似指针）

按顺序：为当前用户：设置好后，再设置下一个用户

此处使用（左外连接）查出（用户表 + 各自账户信息）

- 1 强大的敌人并不可怕，可怕的是自己内容的堕落。
--老洋

3.3、多对多：（用户 --->>> 角色）

- 1 示例：用户和角色
- 2 1、（一个用户）可以有（多个角色）
- 3 2、（一个角色）可以赋予（多个用户）
- 4 步骤
- 5 1、建立两张表， 用户表， 角色表
- 6 ①、让用户表（和）角色表具有（多对多关系）
- 7 ②、需要使用（中间表）
- 8 ③、中间表包含（两表）的（主键），在中间表做（外键）
- 9 2、建立两个实体类：用户类（和）角色类
- 10 ①、让用户（和）角色的实体类能体现出（多对多关系）
- 11 ②、各自包含对方的（一个集合引用）
- 12 3、建立两个配置文件
- 13 ①、用户配置文件
- 14 ②、角色配置文件
- 15 4、实现配置

- 16 | ①、当我们（查询用户）时，可以同时得到（用户）所包含的（角色信息）
- 17 | ②、当我们（查询角色）时，可以同时得到（角色）所赋予的（用户信息）

1 | 第一步：一个角色 --->>> 授予 --->>> 多个用户

```
public class Role implements Serializable{
    private Integer id;           //角色ID
    private String roleName;      //角色名称
    private String roleDesc;      //角色描述
    private List<User> users;     //角色赋予的：用户集合
```

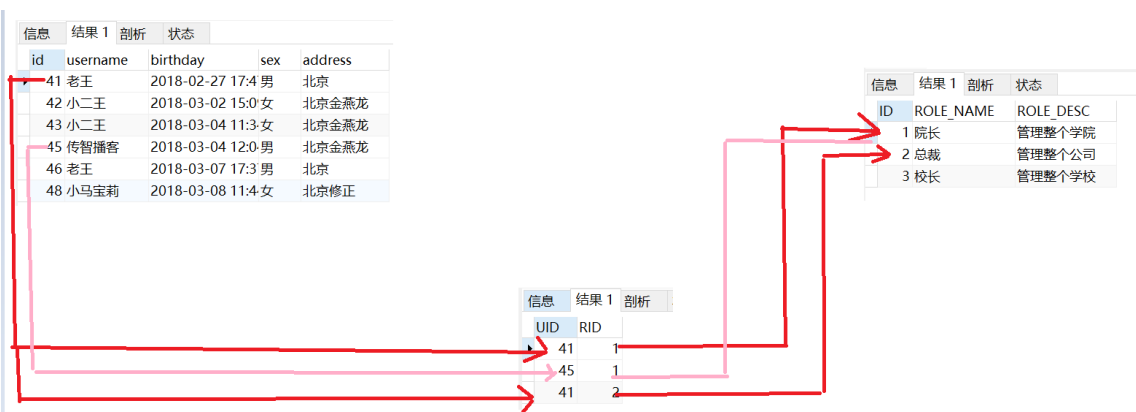
一个角色：多个用户

```
<resultMap id="myMap" type="role">
    <id property="id" column="rid"/>
    <result property="roleName" column="ROLE_NAME" />
    <result property="roleDesc" column="ROLE_DESC" />

    <collection property="users" ofType="user">
        <id property="id" column="id"/>
        <result property="username" column="username"/>
        <result property="birthday" column="birthday"/>
        <result property="sex" column="sex"/>
        <result property="address" column="address"/>
    </collection>
</resultMap>

<select id="findRoles" resultMap="myMap">
    select r.id rid, r.role_Name, r.ROLE_DESC, u.*
    from role r
    left outer join user_role
    on r.ID = user_role.RID
    left outer join user u
    on user_role.UID = u.id
</select>
```

1 | 第二步：一个用户 --->>> 拥有 --->>> 多个角色




```
public class User implements Serializable{
    private int id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;

    private List<Role> roles;    一个用户: 拥有多个角色
}
```

<!--1、查询所有用户信息 + 各个用户（拥有的账户）信息-->

```
<resultMap id="myUser" type="user">
    <id property="id" column="id"/>
    <result property="username" column="username"/>
    <result property="birthday" column="birthday"/>
    <result property="sex" column="sex"/>
    <result property="address" column="address"/>

    <collection property="roles" ofType="role">
        <id property="id" column="rid" />
        <result property="roleName" column="ROLE_NAME"/>
        <result property="roleDesc" column="ROLE_DESC"/>
    </collection>
</resultMap>
```

<!--查询所有的用户信息 + 各个用户（拥有）的角色-->

```
<select id="selAll" resultMap="myUser">
    select u.*, r.id rid, r.ROLE_NAME, r.ROLE_DESC from user u
    left outer join user_role u_r
    on u.ID = u_r.UID
    left outer join role r
    on r.ID = u_r.RID
</select>
```

用户表 -->>> 左外连接 --->>> 中间表
然后再 -->>> 左外连接 --->>> 角色表
切记: 注意sql语句后面加个空格, 再换行

4、JNDI

4.1、JNDI是什么?

- 1 JNDI: Java Naming and Directory Interface。是SUN公司推出的一套规范, 属于JavaEE技术之一

4.2、JNDI有什么用?

- 1 模仿windows系统中的注册表, 在服务器中注册数据源

4.3、JNDI与windows的注册表: 有什么不同?

Map结构

| key : 存的是路径+名称 | value : 存的就是数据 在jndi中存的就是对象 |
|--|--------------------------------|
| HKEY_USERS\DEFAULT \Control Panel\Accessibility \SoundSentry  | 2 |
| HKEY_USERS\DEFAULT \Control Panel\Accessibility \StickyKeys  | 510 |

tomcat服务器一启动

| key : 是一个字符串 | value : 是一个Object |
|-------------------------------|--------------------------------------|
| directory是固定的 name是可以自己指定的 | 要存放什么对象是可以指定的 指定的方式是通过配置 文件的方式 |

5、Mybatis的延迟加载

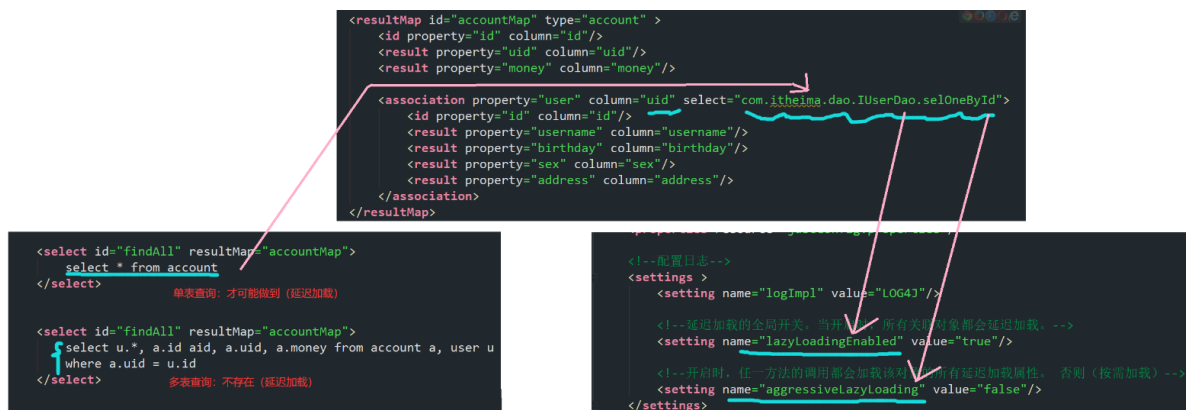
5.1、什么是：延迟加载？

5.1.1、概念举例

- 1 延迟加载：就是用的时候再加载（如：懒汉式）主要体现在jdbc显示条目（是一口气全部，还是一条一条来）

5.1.2、延迟加载：出现的（必要条件）

- 1 必须要是：单表查询，一张一张来， 否则全部查询出来，直接入内存了，不存在（延迟加载的操作）例如：



```

<resultMap id="accountMap" type="account">
  <id property="id" column="id"/>
  <result property="uid" column="uid"/>
  <result property="money" column="money"/>
  <association property="user" column="uid" select="com.itheima.dao.IUserDao.selOneById">
    <id property="id" column="id"/>
    <result property="username" column="username"/>
    <result property="birthday" column="birthday"/>
    <result property="sex" column="sex"/>
    <result property="address" column="address"/>
  </association>
</resultMap>

<select id="findAll" resultMap="accountMap">
  select * from account
</select>

<select id="findAll" resultMap="accountMap">
  select u.*, a.id aid, a.uid, a.money from account a, user u
  where a.uid = u.id
</select>

<!--配置日志-->
<settings>
  <setting name="logImpl" value="LOG4J"/>
  <!--延迟加载的全局开关，当开启时，所有关联对象都会延迟加载。-->
  <setting name="lazyLoadingEnabled" value="true"/>
  <!--开启时，任一方法的调用都会加载该对象的所有延迟加载属性。否则（按需加载）-->
  <setting name="aggressiveLazyLoading" value="false"/>
</settings>

```

单表查询：才可能做到（延迟加载）

多表查询：不存在（延迟加载）

5.1.3、实现一对一（延迟加载）

```

<resultMap id="accountMap" type="account" >
    <id property="id" column="id"/>
    <result property="uid" column="uid"/>
    <result property="money" column="money"/>

    <association property="user" column="uid" select="com.itheima.dao.IUserDao.selOneById">
        <id property="id" column="id"/>
        <result property="username" column="username"/>
        <result property="birthday" column="birthday"/>
        <result property="sex" column="sex"/>
        <result property="address" column="address"/>
    </association>
</resultMap>

```

AccountMapper.xml

直接加载、延迟加载：这样写都可以，不是区分的关键

```

<!--配置日志-->
<settings>
    <setting name="logImpl" value="LOG4J"/>

    <!--延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。-->
    <setting name="lazyLoadingEnabled" value="true"/>

    <!--开启时，任一方法的调用都会加载该对象的所有延迟加载属性。 否则（按需加载）-->
    <setting name="aggressiveLazyLoading" value="false"/>
</settings>

```

MybatisConfig.xml

这才是关键

5.1.4、对比一对一（延迟加载）

```

=> Preparing: select * from user where id = ?
=> Parameters: 46(Integer)
==      Total: 1
-> Preparing: select * from user where id = ? |
=> Parameters: 45(Integer)
==      Total: 1
==      Total: 3

```

直接加载

```

ccountDao.findAll - ==> Preparing: select * from account
ccountDao.findAll - ==> Parameters:
ccountDao.findAll - <==      Total: 3

serDao.selOneById - ==> Preparing: select * from user where id = ?
serDao.selOneById - ==> Parameters: 46(Integer)
serDao.selOneById - <==      Total: 1

```

延迟加载

此处体现不出（二者区别）
因为：都进行了打印输出

```

ex='男', address='北京'}

serDao.selOneById - ==> Preparing: select * from user where id = ?
serDao.selOneById - ==> Parameters: 45(Integer)
serDao.selOneById - <==      Total: 1

```

5.1.5、为什么（立即加载、延迟加载）都最终显示了出来呢？

1 | 原因是：代码控制去打印了

```
public void testFindAll() throws IOException {
    //1、查询所有账户信息 + 各自用户信息
    List<Account> accounts = mapper.findAll();
}
```

直接加载: 会把 (账户) 所属的 (用户) 全部查出来

1 test passed - 2s 811ms

```
IAccountDao.findAll - ==> Parameters:
IUserDao.selOneById - ==> Preparing: select * from user where id = ?
IUserDao.selOneById - ==> Parameters: 46(Integer)
IUserDao.selOneById - <==== Total: 1
IUserDao.selOneById - ==> Preparing: select * from user where id = ?
IUserDao.selOneById - ==> Parameters: 45(Integer)
IUserDao.selOneById - <==== Total: 1
IAccountDao.findAll - <==== Total: 3
```

```
@Test
public void testFindAll() throws IOException {
    //1、查询所有账户信息 + 各自用户信息
    List<Account> accounts = mapper.findAll();
}
```

延迟加载:

不打印时: 只查出 (账户) 信息
打印时: 才把 (账户) 所属 (用户) 一并查出

1 test passed - 2s 804ms

```
on.jdbc.JdbcTransaction - Opening JDBC Connection
pooled.PooledDataSource - Created connection 1748876332.
on.jdbc.JdbcTransaction - Setting autocommit to false on JDBC Connection [com.mysql.
dao.IAccountDao.findAll - ==> Preparing: select * from account
dao.IAccountDao.findAll - ==> Parameters:
dao.IAccountDao.findAll - <==== Total: 3
```

5.2、什么是立即加载?

5.2.1、概念举例

- 1 立即加载: 就是全部一口气加载, 如 (饿汉式)

6、Mybatis中的缓存

6.1、什么是: 缓存?

- 1 存在于内存中的: 临时数据

6.2、为什么: 使用缓存?

- 1 减少和数据库的 (交互次数), 提高执行效率

6.3、什么样数据: 适合使用缓存?

- 1 1、(经常查询) 并 (不经常改变)
- 2 2、数据的 (正确与否) 对 (最终结果影响不大)

6.4、什么样数据：（不适合）使用缓存？

- 1、（经常改变）的数据
- 2、数据的（正确与否）对（最终结果）影响很大
- 3 例如：商品的库存、银行的汇率、股市的牌价

6.5、Mybatis中的一级缓存

6.5.1、（一级缓存）概念

- 1、它指的是：Mybatis中SqlSession对象的缓存
- 2、当我们执行查询之后， 查询的结果会同时存入到：SqlSession为我们提供的一块内存区域中
- 3 该区域的结构是：一个Map，当我们再次查询同样的数据，mybatis会先去sqlSession中查询
- 4 是否有，有的话直接拿出来用。
- 5 3、当SqlSession对象消失时， mybatis的一级缓存也就消失了

6.5.2、有Mybatis一级缓存：效果图

```
test
public void testLevelOneCache(){
    User user1 = mapper.selOneById(45);
    System.out.println("user1 = " + user1);

    User user2 = mapper.selOneById(45);
    System.out.println("user2 = " + user2);

    System.out.println("user1 == user2 ? " + (user1 == user2));
}

1 test passed - 2s 517ms

- ==> Preparing: select * from user where id = ?
- ==> Parameters: 45(Integer)
- <== Total: 1 第一次查询到数据库，第二次到SqlSession占据的空间内存处

= com.itheima.domain.User@306cf3ea
= com.itheima.domain.User@306cf3ea 此时：它们是同一个SqlSession
== user2 ? true 默认启动了：一级缓存
```

6.5.3、无mybatis一级缓存的：效果图

```
65 //Test
66 public void testLevelOneCache(){
67     User user1 = mapper.selOneById(45);
68     System.out.println("user1 = " + user1);
69     session.clearCache(); //清除缓存时， 将会查询两次， ID将不同
70
71     User user2 = mapper.selOneById(45);
72     System.out.println("user2 = " + user2);
73
74     System.out.println("user1 == user2 ? " + (user1 == user2));
75 }
76
```

```
66 public void testLevelOneCache(){
67     User user1 = mapper.selOneById(45); //第一个SqlSession
68     System.out.println("user1 = " + user1);
69     session.close();
70
71     SqlSession session2 = factory.openSession(); //第二个SqlSession
72     IUserDao mapper2 = session2.getMapper(IUserDao.class);
73     User user2 = mapper2.selOneById(45);
74
75     System.out.println("user2 = " + user2);
76
77     System.out.println("user1 == user2 ? " + (user1 == user2)); //返回false
78 } //因为：mybatis一级缓存是存在SqlSession中分配的（内存空间中）
```

1 test passed - 2s.735ms

- ==> Preparing: select * from user where id = ?

- ==> Parameters: 45(Integer)

- <== Total: 1

- ==> Preparing: select * from user where id = ?

- ==> Parameters: 45(Integer)

- <== Total: 1

user1 = com.itheima.domain.User@306cf3ea

2020-11-11 15:43:30,655 2527 [main] DEBUG theima.dao.IUserDao.selOne

2020-11-11 15:43:30,657 2529 [main] DEBUG theima.dao.IUserDao.selOne

2020-11-11 15:43:30,666 2538 [main] DEBUG theima.dao.IUserDao.selOne

user2 = com.itheima.domain.User@5136d012

user1 == user2 ? false

进行了：SqlSession缓存清除

1 test passed - 2s.651ms

- ==> Preparing: select * from user where id = ?

- ==> Parameters: 45(Integer)

- <== Total: 1

- Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@6db9f5a4]

- Returned connection 1840903588 to pool.

- Opening JDBC Connection

- Checked out connection 1840903588 from pool.

- Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@6db9f5a4]

- ==> Preparing: select * from user where id = ?

- ==> Parameters: 45(Integer)

- <== Total: 1

因为SqlSession不同，所以第二次查询，是到（数据库）中查询获取的信息

user1 = com.itheima.domain.User@306cf3ea

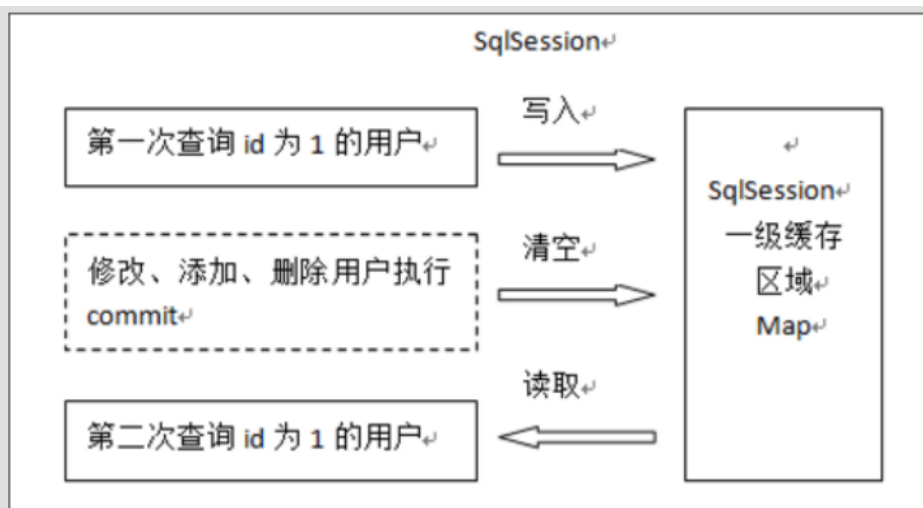
user2 = com.itheima.domain.User@5136d012

user1 == user2 ? false

6.5.4、同样的SqlSession为什么也会出现：到数据库查询的情况？

- 1 一级缓存是 sqlSession 范围的缓存，当调用 sqlSession 的修改，添加，删除，commit(), close()等
- 2 方法时，就会清空一级缓存。

6.5.5、原理图



第一次发起查询用户 id 为 1 的用户信息，先去找缓存中是否有 id 为 1 的用户信息，如果没有，从数据库查询用户信息。

得到用户信息，将用户信息存储到一级缓存中。

如果 sqlSession 去执行 commit 操作（执行插入、更新、删除），清空 sqlSession 中的一级缓存，这样做的目的是为了缓存中存储的是最新的信息，避免脏读。

第二次发起查询用户 id 为 1 的用户信息，先去找缓存中是否有 id 为 1 的用户信息，缓存中有，直接从缓存中获取用户信息。

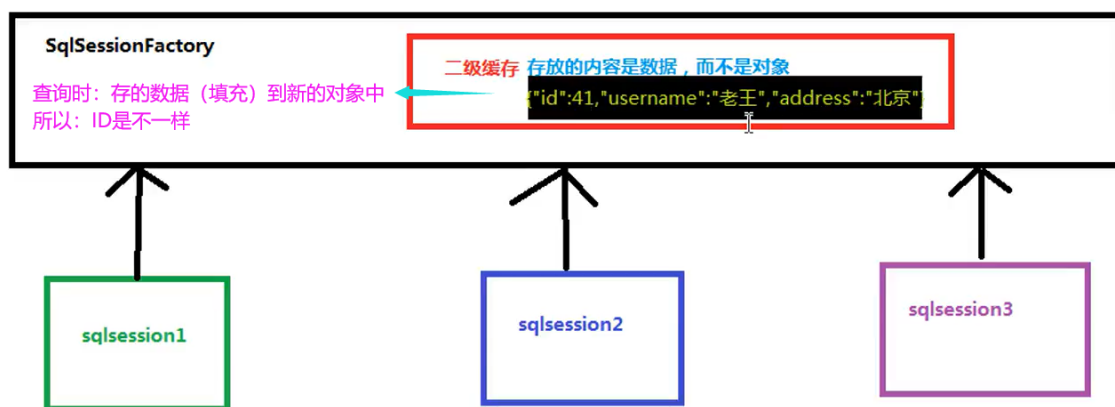
6.6、Mybatis中的二级缓存

6.6.1、（二级缓存）概念

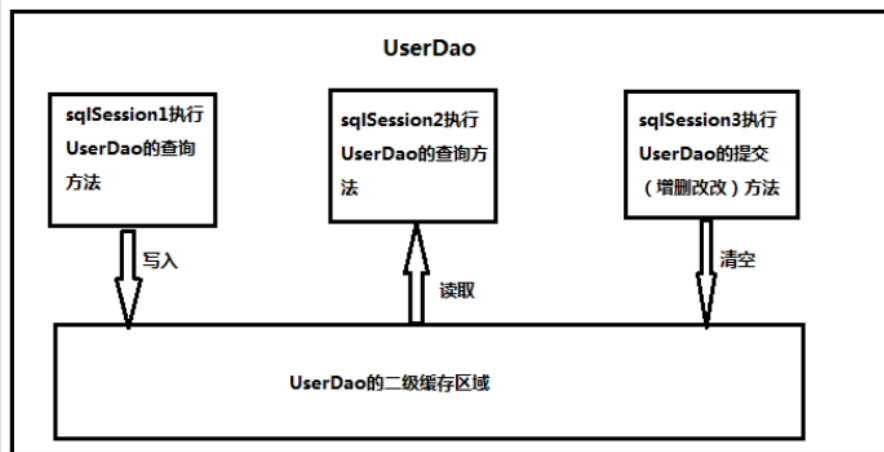
- 1 二级缓存是 mapper 映射级别的缓存，多个 SqlSession 去操作同一个 Mapper 映射的 sql 语句
- 2 多个SqlSession 可以共用二级缓存，二级缓存是跨 SqlSession 的。

6.6.2、（二级缓存）原理图

图一



图二



首先开启 mybatis 的二级缓存。

sqlSession1 去查询用户信息，查询到用户信息会将查询数据存储到二级缓存中。

如果 SqlSession3 去执行相同 mapper 映射下 sql，执行 commit 提交，将会清空该 mapper 映射下的二级缓存区域的数据。

sqlSession2 去查询与 sqlSession1 相同的用户信息，首先会去缓存中找是否存在数据，如果存在直接从缓存中取出数据。

6.6.2、二级缓存的：开启

1 | 第一步：在 SqlMapConfig.xml

```
<settings>
  <!-- 开启二级缓存的支持 -->
  <setting name="cacheEnabled" value="true"/>
</settings>
```

因为 cacheEnabled 的取值默认就为 true，所以这一步可以省略不配置。为 true 代表开启二级缓存；为 false 代表不开启二级缓存。

1 | 第二步：配置相关的 Mapper 映射文件

```
<mapper namespace="com.itheima.dao.IUserDao">
  <!--开启user支持二级缓存-->
  <cache/>
  <!-- 查询所有 -->
  <select id="findAll" resultType="user">
    select * from user
  </select>

  <!-- 根据id查询用户 -->
  <select id="findById" parameterType="INT" resultType="user" useCache="true">
    select * from user where id = #{uid}
  </select>
```

6.6.3、二级缓存发挥了作用：效果图

```
- ==> Preparing: select * from user where id = ?
- ==> Parameters: 45(Integer)
- <==      Total: 1

- Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@3098cf3b]
- Returned connection 815320891 to pool.
- Cache Hit Ratio [com.itheima.dao.IUserDao]: 0.5
```

6.6.4、当第一个SqlSession进行的：增删改，并commit时，二级缓存：失效图


```
- ==> Preparing: select * from user where id = ?
- ==> Parameters: 45(Integer)
- <==      Total: 1
- ==> Preparing: update user set username = "张三" where id = ?
- ==> Parameters: 45(Integer)
- <==      Updates: 1
- Committing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@3098cf3b]
- Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@3098cf3b]
- Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@3098cf3b]
- Returned connection 815320891 to pool.

- Cache Hit Ratio [com.itheima.dao.IUserDao]: 0.0
- Opening JDBC Connection
- Checked out connection 815320891 from pool.
- Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@3098cf3b]
- ==> Preparing: select * from user where id = ?
- ==> Parameters: 45(Integer)
- <==      Total: 1
```

7、Mybatis中的注解开发

7.1、为什么要使用Mybatis的注解开发？

1 | 为了减少编写 `Mapper.xml` 的映射文件

7.2、Mybatis中常用的：注解图

`@Insert`: 实现新增
`@Update`: 实现更新
`@Delete`: 实现删除
`@Select`: 实现查询
`@Result`: 实现结果集封装
`@Results`: 可以与 `@Result` 一起使用，封装多个结果集
`@ResultMap`: 实现引用 `@Results` 定义的封装
`@One`: 实现一对一结果集封装
`@Many`: 实现一对多结果集封装
`@SelectProvider`: 实现动态 SQL 映射
`@CacheNamespace`: 实现注解二级缓存的使用

7.3、环境搭建

1 | 1、以前的： `mapper.xml` 不用写

1 | 2、接口上方法：改用注解， 其余不再操作依旧

7.4、解决：（实体类）与（数据库）表名，参数不对应方法

1. @Results 和 @Result : 实现property 和 column 对应
2. @ResultMap 实现 (对应关系: 复用)

```
public interface IUserDao {  注解: 写接口中

    //1、查询: User表所有内容
    @Select("select * from user")
    @Results(id = "myMap", value = {  @Results中的id是: 为了复用, 参数对应
        @Result(id = true, property = "userId", column = "id"), @Result中的id是: 主键标识
        @Result(property = "userName", column = "username"),
        @Result(property = "userBirthday", column = "birthday"),
        @Result(property = "userSex", column = "sex"),
        @Result(property = "userAddress", column = "address")
    })
    List<User> selAll();

    //2、通过ID查询数据  此处: @ResultMap作用是 (复用) @Results配置的参数
    @ResultMap(value={"myMap"})  //此处还可以写{"myMap"}, 原因是只有一个值是可以: 省略value
    @Select("select * from user where id = #{id}")
    User selById(Integer id);

    //3、通过姓名: 模糊查询
    @ResultMap("myMap")  //只有一个值时: 可以省略{}
    //@Select("select * from user where username like #{username}")  //占位符
    @Select("select * from user where username like '%${value}%'")  //字符串拼接
    List<User> selByName(String username);
}
```

7.5、一对一查询 (1个账户 --->> 1个用户)

```
public interface IAccountDao {
    //1、查询: 所有账户信息, 并查询 (对应的: 用户信息) --->>>要求: 延迟加载  实现: (一对一) 一个账户 ->> 对应一个用户
    @Select("select * from account")
    @Results(id = "AccountMap", value = {
        @Result(id = true, property = "accountId", column = "id"),
        @Result(property = "userId", column = "uid"),
        @Result(property = "money", column = "money"),

        @Result(
            property = "user", column = "uid", column = "uid" 查询: 用户表的参数
            one=@One(select = "com.yyy.Dao.IUserDao.seluserById", fetchType = FetchType.LAZY)
        )
    })
    List<Account> selAllAccounts();
}
```

@Result 中有个参数为 one = @One(select = "", fetchType =)
1、其中: @one中的select = "另一张表方法的: 全限定名"
2、fetchType= (加载方式, 是个枚举类型)

7.6、一对多查询 (1个用户 -->> 多个账户)

```
//4、延迟加载实现：一对多(1个用户-->多个账户)
@Results(id = "userMap", value = {
    @Result(id = true, property = "userId", column = "id"),
    @Result(property = "userName", column = "username"),
    @Result(property = "userBirthday", column = "birthday"),
    @Result(property = "userSex", column = "sex"),
    @Result(property = "userAddress", column = "address"),
    @Result(
        property = "accounts", column = "id",
        many = @Many(select = "com.yyy.Dao.IAccountDao.selByUserId", fetchType = FetchType.LAZY)
    )
})
@Select("select * from user")
List<User> selAllUsers();
}
```

7.7、缓存配置

- 1、mybatis.xml中的<setting>中设置：CacheEnabled （非必须：此处只是提醒）

设置 (settings)

这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。下表描述了设置中各项设置的含义、默认值等。

| 设置名 | 描述 | 有效值 | 默认值 |
|--------------|------------------------------|--------------|------|
| cacheEnabled | 全局性地开启或关闭所有映射器配置文件中已配置的任何缓存。 | true false | true |

- 2、接口中注解：@CacheNamespace(blocking = true)

```
@CacheNamespace(blocking = true) 注意：是在接口上面加，不是在（抽象方法上面）
public interface IUserDao {
    //1、查询：User表所有内容
    @Results(id = "myMap", value = {
        @Result(id = true, property = "userId", column = "id"),
        @Result(property = "userName", column = "username"),
        @Result(property = "userBirthday", column = "birthday"),
        @Result(property = "userSex", column = "sex"),
        @Result(property = "userAddress", column = "address")
    })
    @Select("select * from user")
    List<User> selAll();
}
```

n、Mybatis中的四个问题

1、什么是事务？

2、事务的四大特性ACID？

3、不考虑隔离性会产生的3个问题？

4、解决办法：四种隔离级别

