MyBatis-概览及第一个程序

笔记本： 框架
创建时间： 2022/2/18 10:51 **更新时间：** 2022/3/3 17:24
作者： r37qboxc

MyBatis-概览及第一个程序

作者： r37qboxc

简介

- 持久层框架
- 支持定制化SQL:可以自己写SQL
- 避免了JDBC代码以及获取结果集
- 支持简单的XML

如何获得MyBatis

- Maven仓库 https://mvnrepository.com/
- Github开源

分页

- 内存分页(把所有数据找出来再进行分页, 效能差):Mybatis默认内存分页, 需要使用分页插件
- 物理分页(拼SQL语句后面有limit): Hibernate setMaxResult是物理分页

持久化

- 将程序的数据在持久状态(数据库)和瞬时状态转化的过程
- 内存:断电即失
- 数据库(jdbc), io文件持久化 eg:冰箱冷藏,对食物持久化

持久层

其它层:dao层, service层, controller层

- 完成持久化工作的模块
- 层界限十分明显

为什么需要Mybatis

- 传统的JDBC代码复杂 -> 简化 ->框架 -> 自动化
- 帮助programmer将数据存入到数据库中

- sql写在xml文件里面, sql与代码分离, 提高了可维护性
- 主流框架

XML中构建SqlSessionFactory

- 每个MyBatis应用都是以SqlSessionFactory的实例为核心的(有session builder创建出来)
- xml可以配置SqlSessionFactory的属性

```
String resource = "org/mybatis/example/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
```

基本配置

- 导入依赖 -> 配置文件 -> 写utils(SqlSessFactory)

编写代码

- 实体类
- 接口类
- mapper.xml(实现)

测试

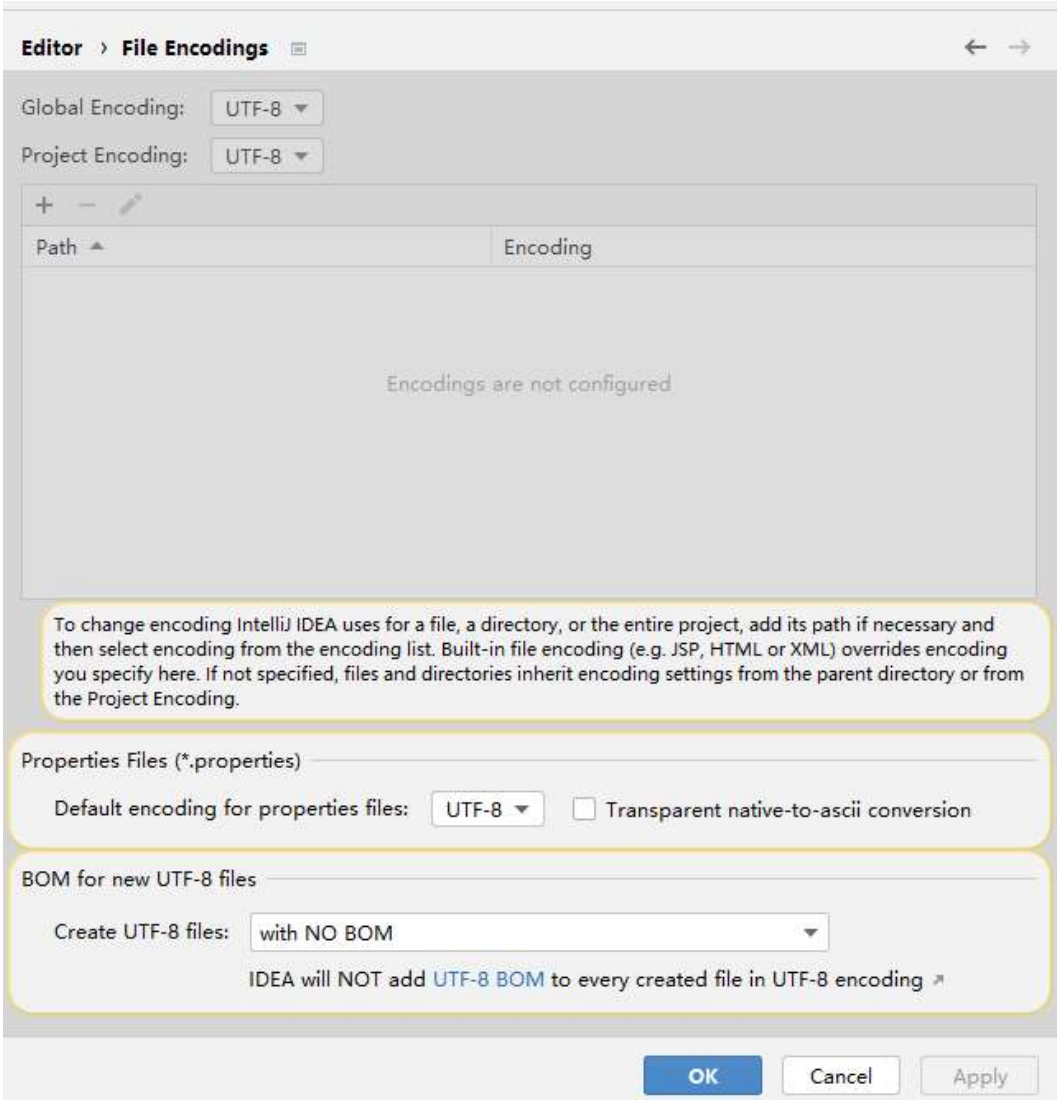
- 注意绑定报错org.apache.ibatis.binding.BindingException: Type interface com.mybatis.dao.UserDao is not known to the MapperRegistry. 要在 config.xml 注册该mapper.xml
- MapperReistry是什么? 核心配置文件中注册mappers
- Maven:约定大于配置, 我们的配置有可能不生效, 需要配置资源过滤文件 在pom配置build
- junit测试

配置时注意点:

```
    </environments>
    <mappers>
        <mapper resource="com/mybatis/dao/UserMapper.xml"></mapper>
    </mappers>
```

路径是斜杠 不是 . 错误导致扫描不到 com.mybatis.dao.UserMapper.xml
mysql连接url需要齐全
pom的定义文件需要按顺序, 父pom定义版本 子导入依赖 否则会下载不了依赖包
xml文件的中文注释有机会引发问题, 如下, 需要在setting找
file encodeing设置 utf-8

```
<mapper namespace="com.mybatis.dao.UserDao">
<!--    中文注释-->
    <select id="getUserList" resultType="com.mybatis.pojo.User">
        select * from my_frame_demo.user
    </select>
</mapper>
```



MyBatis-CRUD

作者： r37qboxc

CRUD

namespace 中的包名要和接口(dao/mapper)的包名一致

https://github.com/ChrisLuiBird/MyDemoForFrame

mybatis

Select语句

- id:对应namespace中的方法
- resultType:sql语句的返回值
- parameterType: 传参类型

sql的返回值: 基本类型, 类

```
<select id="getUserById" parameterType="int" resultType="com.kuang.pojo.User">
    select * from mybatis.user where id = #{id}
</select>
```

id要和接口方法一致, #{}有预编译机制可以防止sql注入,效率高

增删改需要提交事务

insert语句

```
<!--对象中的属性，可以直接取出来-->
<insert id="addUser" parameterType="com.kuang.pojo.User">
    insert into mybatis.user (id, name, pwd) values (#{id},#{name},#{pwd});
</insert>
```

传参为类, 传参属性要跟类定义的一样

Map和模糊查询

```
Map<String, Object> map = new HashMap<String, Object>();

map.put("userid",5);
map.put("userName","Hello");
map.put("passWord","2222333");

mapper.addUser2(map);
```

将属性装入到Map再插入数据库, 应对大量的表中attribute数量多个参数用Map或者注解

模糊查询

```
<select id="getUserLike" resultType="com.kuang.pojo.User">
    select * from mybatis.user where name like "%"#{value}"%"
</select>
```

sql注入~~~

MyBatis-配置解析

作者： r37qboxc

配置解析

https://github.com/ChrisLuiBird/MyDemoForFrame
mybatis-config
核心配置文件
mybatis-config.xml

```
configuration（配置）
properties（属性）
settings（设置）
typeAliases（类型别名）
typeHandlers（类型处理器）
objectFactory（对象工厂）
plugins（插件）
environments（环境配置）
environment（环境变量）
transactionManager（事务管理器）
dataSource（数据源）
databaseIdProvider（数据库厂商标识）
mappers（映射器）
```

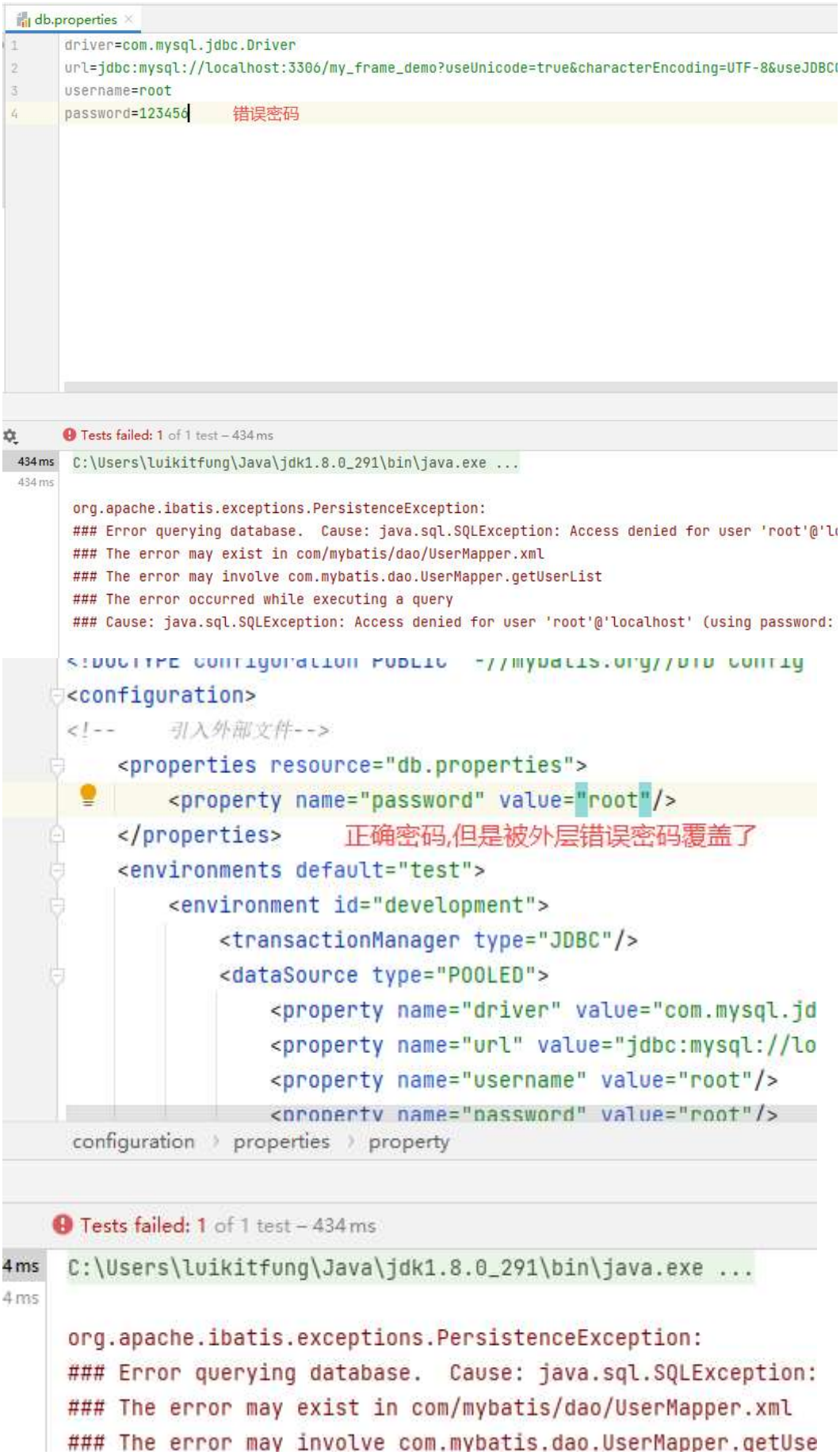
环境配置
事务管理器: JDBC->提交和回滚设置
MANAGED -> 从来不做回滚和提交

数据源:
配置多套环境
连接数据库: dbcp, c3p0, druid
UNPOOLED(没有连接池,每次请求时打开和关闭连接) 池:用完可以回收
POOLED
JNDI:极少用

属性(properties)
通过properties来引用配置文件: db.properties

properties标签必须按顺序定义, 不能放在最下面

配置文件优先级



别名(typeAlias), 为Java类型设置一个短的名字

```
<typeAliases>
    <!--    实体类别名-->
<!--        <typeAlias type="com.mybatis.pojo.User" alias="User"></typeAlias>-->

<!--        包别名，扫描这个包下的类，并且以这个包下的类名首字母小写为该类的别名-->
    <package name="com.mybatis.pojo"/>
</typeAliases>
```

如果实体类十分多，建议使用第二种。

第一种可以DIY别名，第二种则·不行·，如果非要改，需要在实体上增加注解

```
@Alias("user")
public class User {}
```

基本类型, 数据类型别名

基本类型 _int -> int

int -> Integer

map -> Map

设置(必懂3个)

缓存 cache

懒加载lazyloadingEnabled

日志指定MyBatis所用日志的具体实现

6、其他配置

- typeHandlers（类型处理器）
- objectFactory（对象工厂）
- plugins插件
  - mybatis-generator-core
  - mybatis-plus
  - 通用mapper

映射器

resource 单个xml注册

class 类注册,但是类注册需要和xml在同个包下且命名前缀要相同

package 包注册 但是java类注册需要和xml在同个包下且命名前缀要相同

属性名与字段名不一致的问题

pojo层的属性与数据库的名不一致(password 与 pwd)

解决方法:

- 起别名

```
<select id="getUserById" resultType="com.kuang.pojo.User">
    select id,name,pwd as password from mybatis.user where id = #{id}
</select>
```

- resultMap

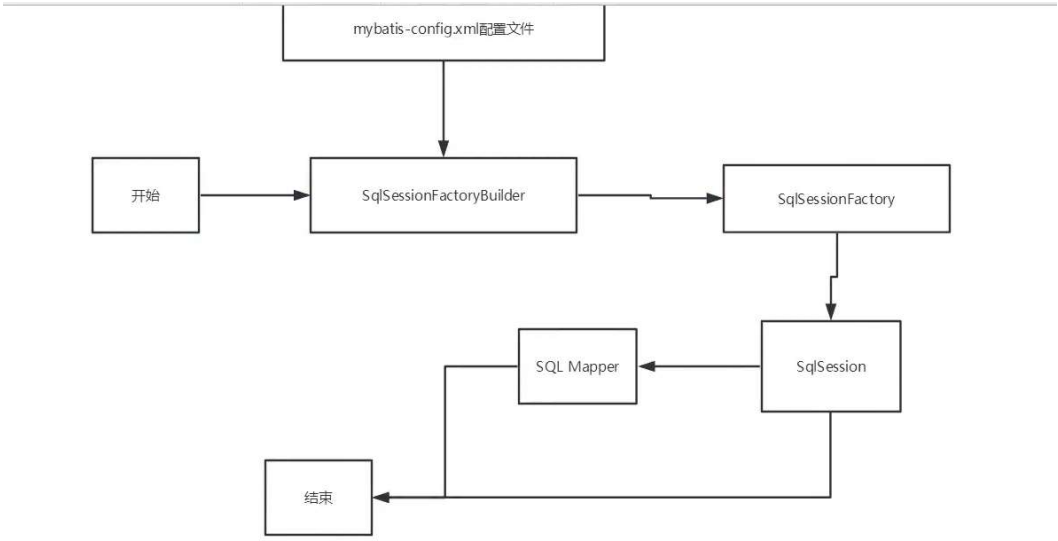
  结果集映射

- `resultMap` 元素是 MyBatis 中最重要最强大的元素
- ResultMap 的设计思想是，对于简单的语句根本不需要配置显式的结果映射，而对于复杂一点的语句只需要描述它们的关系就行了。
- `ResultMap` 最优秀的地方在于，虽然你已经对它相当了解了，但是根本就不需要显式地用到他们。

MyBatis-生命周期和作用域

作者： r37qboxc

生命周期和作用域是很重要的, 错误的使用会导致严重的并发问题

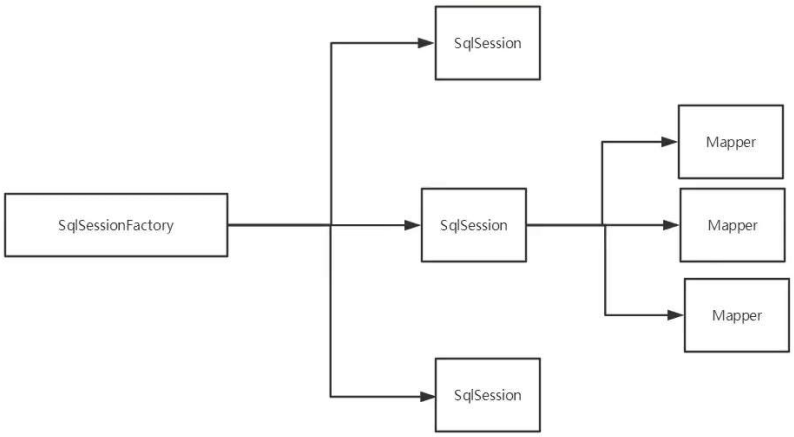


Builder 创建器
一旦创建了 Factory, 就不在需要他了

Factory工厂
应用作用域, 全局程序, 运行期间就一直存在, 全局只有1个,不需重建

Session请求

- 连接到连接池的一个请求
- session实例不是线程安全, 不能共享, 请求作用域
- 打开用完后赶紧关掉,让其他请求进来



- 每一个Mapper代表一个具体的业务(CRUD sql)

MyBatis - 分页及注解开发

作者： r37qboxc

分页

https://github.com/ChrisLuiBird/MyDemoForFrame

mybatis-config

sql: limit

rowbounds分页: 不可取, 缓存分页, 影响效率

注解开发


```
<!--绑定接口-->
<mappers>
    <mapper class="com.kuang.dao.UserMapper"/>
</mappers>
```

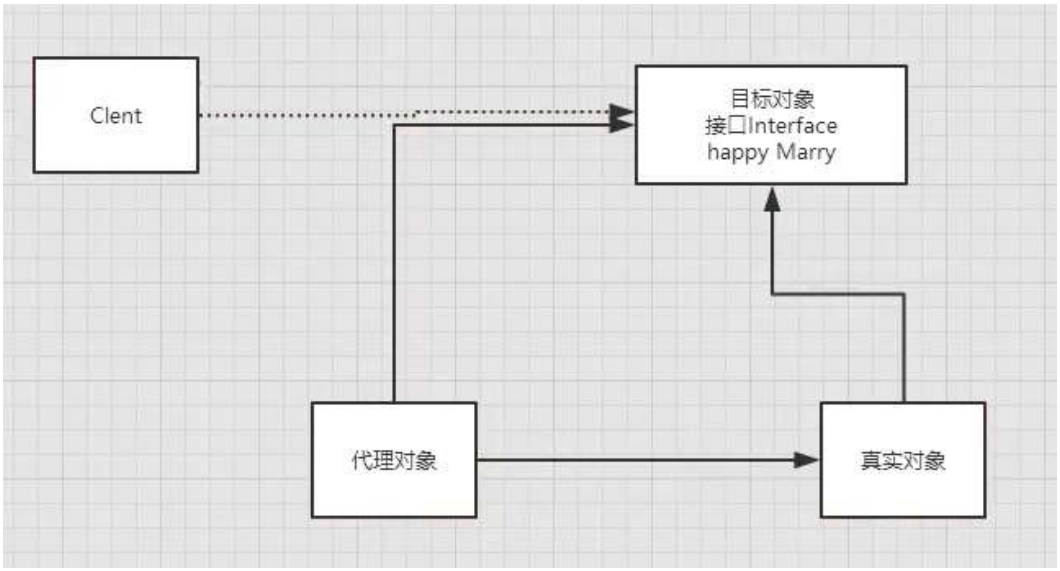
原理:

本质: 利用反射加载获取接口对象的方法.

底层: 动态代理

在接口interface上使用注解实现
在核心配置文件配置上绑定类(可以和xml文件一起用)

动态代理模式



CRUD
事务自动提交:在配置util类 MyBatilUtil.java
openSession(true)

```java
/**
 * @author LUIKITFUNG
 * @create 2022 - 02 - 21 - 14:52
 **/
public interface UserDao {
    @Select("select * from user")
    @Results(id = "UserMap", value = {
            @Result(column = "pwd", property = "password")
    })
    public List<User> getUserList();

    @Select("select * from user where id = #{id}")
    @ResultMap("UserMap")
    public User getUserById(@Param("id") int id);

    @Insert("insert into user(id, name, pwd) values (#{id}, #{name}, #{pwd})")
    public int addUser(User user);

    @Update("update user set name = #{name}, pwd = #{pwd} where id = #{id}")
    public int updateUser(User user);

    @Delete("delete from user where id = #{id}")
    public int deleteUser(@Param("id") int id);
}
```

类名属性与库名不一致情况下
注解resultmap写法

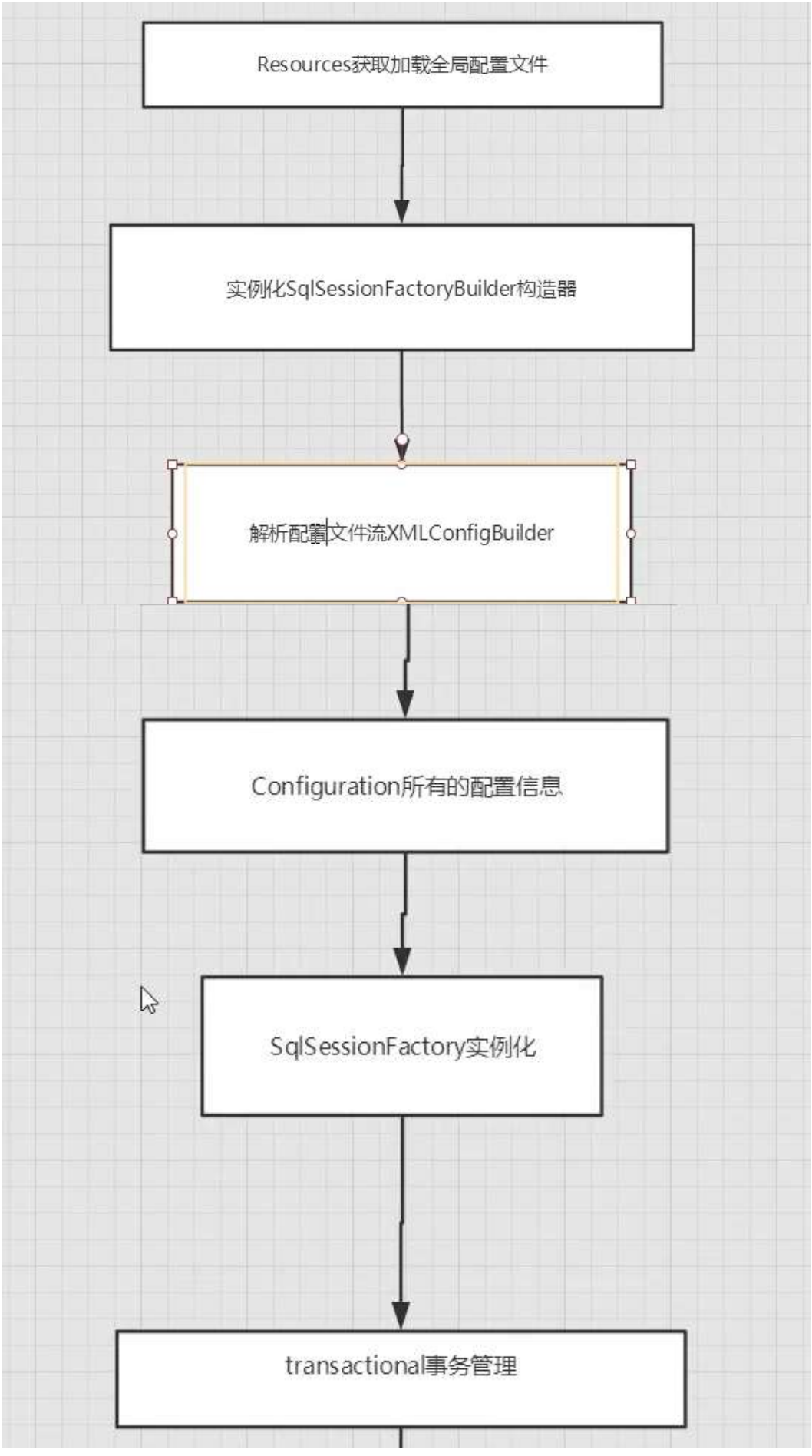
#{}与${} 区别
#{}会加上 空个 " "来解析,很大程度防止sql注入

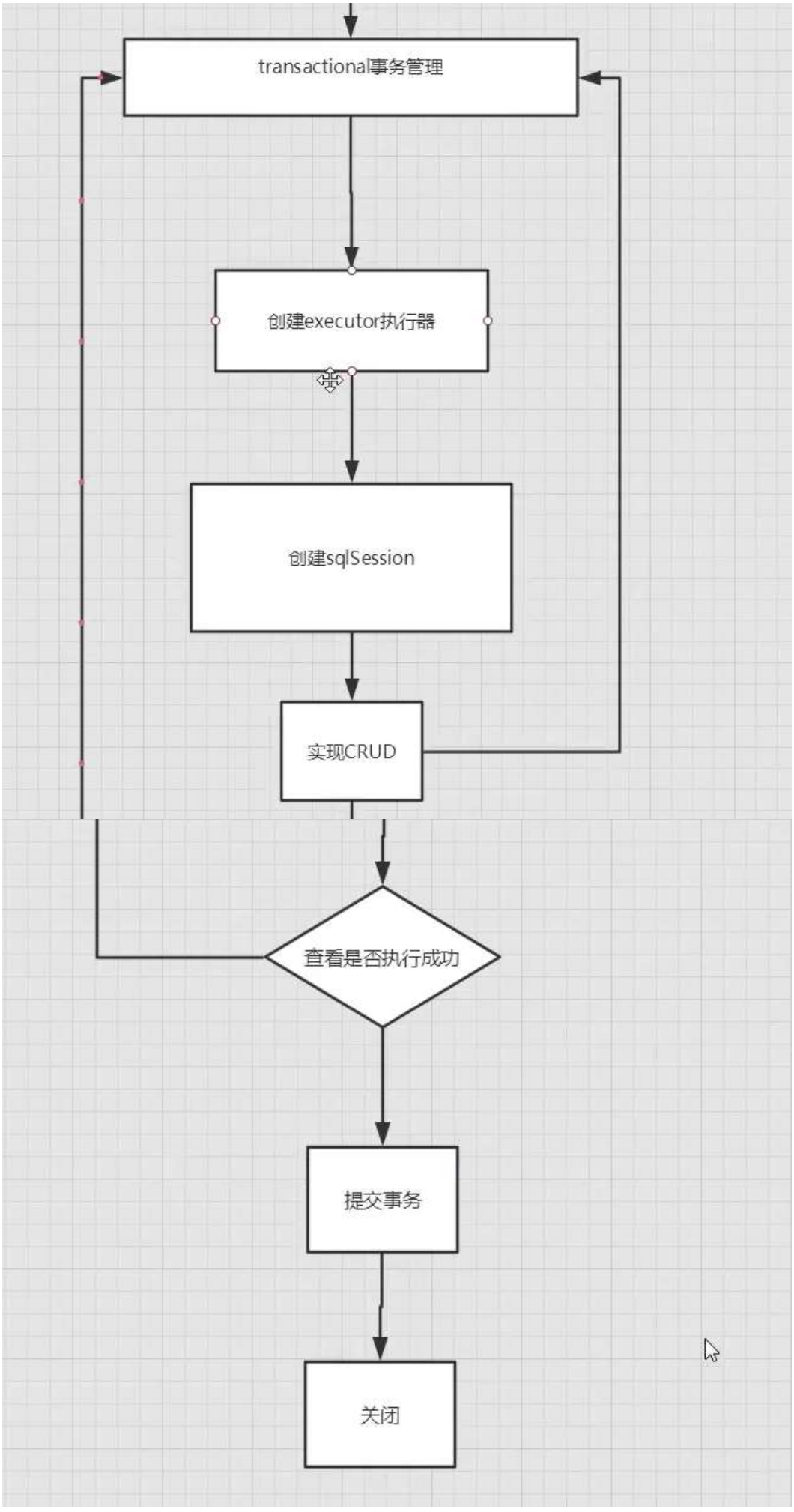
${}是拼接的, 无法防止sql注入, 除非在mybatis使用order by动态参数时使用

MyBatis-实现流程

作者： r37qboxc

Resource 获取全局核心配置文件 -> 实例化sqlSessionFactoryBuilder构造器 -> 解析配置文件流XMLConfigBuilder-> Configuration所有配置信息 -> SessionFactory实例化 -> transaction事务管理器 -> executor执行器 -> 创建sqlSession -> 实现CRUD -> 查看是否执行成功 -> 提交事务 -> 关闭


Resources获取加载全局配置文件
实例化SqlSessionFactoryBuilder构造器
解析配置文件流XMLConfigBuilder
Configuration所有的配置信息
SqlSessionFactory实例化
transactional事务管理


transactional事务管理
创建executor执行器
创建sqlSession
实现CRUD
查看是否执行成功
提交事务
关闭

MyBatis-复杂查询环境搭建

作者： r37qboxc

多对一

查询所有的学生信息, 以及对应的老师的信息

思路:

1. 查询所有学生,
2. 根据学生查询出来的老师id 查询对应的老师
3. 用resultMap处理

resultMap处理:
association: 对应一个老师
collection:对应一堆学生

按照查询嵌套处理

```xml
<?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.mybatis.dao.StudentMapper">

    <select id="getStudents" resultMap="StudentWithTeacher">
        select * from student
    </select>
    <select id="getTeacher" resultType="Teacher">
        select * from te  id = #{id}
    </select>
    <resultMap id="StudentWithTeacher" type="Student">
        <association property="teacher" column="tid" javaType="Teacher" select="getTeacher"></association>
    </resultMap>
</mapper>
```

类里面的属性

库里面的外键

按照结果嵌套处理

```xml
    <select id="getStudentsForResult" resultMap="StudentWithTeacher2">
        select s.id sid,s.name sname, t.id tid, t.name tname from student s, teacher t where s.tid = t.id
    </select>

    <resultMap id="StudentWithTeacher2" type="Student">
        <result property="id" column="sid"></result>
        <result property="name" column="sname"></result>
        <association property="teacher" javaType="Teacher">
            <result property="name" column="tname" />
        </association>
    </resultMap>
</mapper>
```

一对多(一个老师对应多个学生)

结果嵌套

```xml
<mapper namespace="com.mybatis.dao.TeacherMapper">
<!--    结果嵌套-->
    <select id="getTeacherById" resultMap="TeacherStudent">
        select s.id sid, s.name sname, t.id tid, t.name tname
            from student s, teacher t
        where s.id = t.id and t.id = #{id}
    </select>
    <resultMap id="TeacherStudent" type="Teacher">
        <result property="id" column="tid" />
        <result property="name" column="tname" />

<!--        结果集 collection 要用集合内的泛型 要用 ofType而不是javaType-->
        <collection property="students" ofType="Student">
            <result column="sname" property="name" />
            <result column="sid" property="id" />
        </collection>
    </resultMap>
```

注意 要用ofType 而不是 javaType收集

查询嵌套

```xml
    <!--    查询嵌套-->
    <select id="getTeacherByIdForQuery" resultMap="TeacherStudent2">
        select * from teacher where id = #{id}
    </select>
    <resultMap id="TeacherStudent2" type="Teacher">
        <collection property="students" ofType="Student" javaType="ArrayList" select="getStudents" column="id"></collection>
    </resultMap>
    <select id="getStudents" resultType="Student">
        select * from student where tid = #{id}
    </select>
</mapper>
```

回传类型里面的泛类

回传类型

传值给 下面的getStudents

注意

- 要加回传类型 javaType 因为他是一次性获取全部, 按结果查询不加是因为本来就在遍历
- 要加column传tid给调用方法

- 保证sql的可读性
- 注意一对多和多对一属性名和字段的问题

避免慢sql

面试高频:

1. mysql引擎
2. InnoDB的底层原理

3. 索引
4. 索引优化

MyBatis-动态sql

作者： r37qboxc

动态sql

what: 根据不同的条件生成不同的SQL语句,避免拼接sql的痛苦

```
 **/
@Data
@SuppressWarnings("all")
public class Blog {
    private String id;
    private String title;
    private String author;
    private Date createTime; //字段名不抑制, 用经典_ 数据库名 变成 java 驼峰
    private int views;
}
```

配置 数据库 下划线 转 java 驼峰

```
<properties resource="db.properties" />
<settings>
    <setting name="logImpl" value="STDOUT_LOGGING"/>
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
<typeAliases>
    <!--    实体类别名-->
```

IF判断

```xml
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-
<mapper namespace="com.mybatis.dao.BlogMapper">
    <insert id="addBlog" parameterType="Blog">
        insert into blog (id, title, author, create_time, views)
        values (#{id}, #{title}, #{author}, #{createTime}, #{views});

    </insert>

    <select id="getBlogs" parameterType="map" resultType="Blog">
        select * from blog where 1=1

        <if test="title">
            <foreach collection="title" item="item">
                and title like "%" #{item} "%"
            </foreach>
        </if>
        <if test="author">
            and author like "%" #{author} "%"
        </if>


    </select>
</mapper>
```

可以不写 != null 或者.size()>0

Choose语句(像 Java的 switch catch)

```xml
    <select id="getBlogsByChoose" parameterType="map" resultType="Blog">
        select * from blog
        <where>
            <choose>
                <when test="title">
                    title like "%" #{title} "%"
                </when>
                <when test="title">
                    and author like "%" #{author} "%"
                </when>
                <otherwise>
                    and views like "%" #{views} "%"
                </otherwise>

            </choose>
        </where>

    </select>
```

使用where语句时, 如果里面没有则跳过, 有则抹去里面第一个的 and或者or
其后的判断语句都要加 and

```java
@Test
public void geBlogsByChoose(){
    SqlSession session = MyBatisUtils.getSession();
    BlogMapper mapper = session.getMapper(BlogMapper.class);

    Map<String, Object> queryMap = new HashMap<>();
//      queryMap.put("title","T1");
//      queryMap.put("author","c");
    queryMap.put("views", "3");
    List<Blog> result = mapper.getBlogsByChoose(queryMap);
    result.forEach(blog -> {
        System.out.println(blog);
    });
    session.close();
}
```

there is Number type in DB,
but it will throw exception if I use INT to query

Update 语句

```xml
<update id="updateBlog" parameterType="Blog">
    update blog
        <set>
            <if test="title">
                title = #{title} ,
            </if>
            <if test="author">
                author = #{author} ,
            </if>
        </set>
        where id = #{id}
</update>
```

mybatis会帮我们抹去不必要的逗号

```java
@Test
public void updateBlog(){
    SqlSession session = MyBatisUtils.getSession();
    BlogMapper mapper = session.getMapper(BlogMapper.class);
    Blog blog = new Blog();
    blog.setId("920b2e580825411b8ac96c2c249aa926");
//      blog.setTitle("H3");
    blog.setAuthor("cryin");

    int resultRow = mapper.updateBlog(blog);
    System.out.println(resultRow);
    session.close();
}
```

Trim(定制化)

prefix前缀

suffix后缀

所有的动态sql本质是sql语句, 只是可以在sql层面执行逻辑代码

id, where, set, choose, when

SQL片段
what: 我们可能将一些公共的sql片段提取出来(类似方法), 减少冗余代码
<sql>标签, 配合<include refid>

注意:

1. 最好基于单表来定义sql片段
2. 不存在 where 标签

Foreach

MyBatis-缓存

作者: r37qboxc

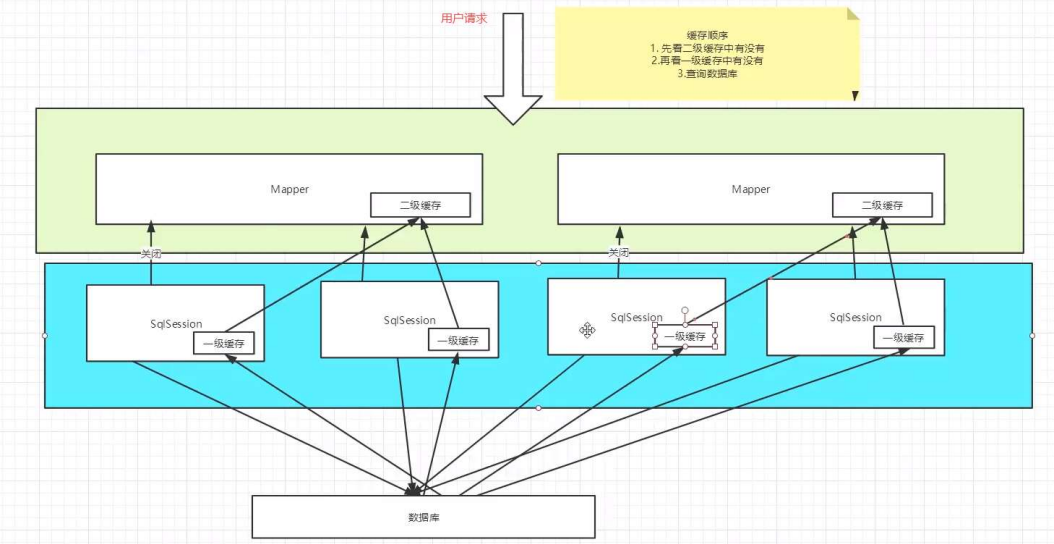
查询: 连接数据库, 耗资源

一次查询的结果,给他暂存在一个可以直接取到的地方 -->
内存: 缓存, 减少与数据库的交互次数, 减少系统开销, 提高性能
再次查询相同数据的时候, 直接走缓存, 不用走数据库了

读写分离:缓存控制
主从复制:保持数据库的一致性

什么数据能使用缓存: 经常查询且不常修改

MyBati缓存



定义了:一级缓存和二级缓存

默认只开一级缓存: SqlSeesion级别的缓存, 本地缓存, session,close()缓存清掉
二级缓存: 需要手动开启, 基于namespace级别的, 对应一个mapper
扩展: Cache缓存接口, 通过实现Cache接口来自定义二级缓存
Cache:

可用的清除策略有：

- `LRU` – 最近最少使用：移除最长时间不被使用的对象。
- `FIFO` – 先进先出：按对象进入缓存的顺序来移除它们。
- `SOFT` – 软引用：基于垃圾回收器状态和软引用规则移除对象。
- `WEAK` – 弱引用：更积极地基于垃圾收集器状态和弱引用规则移除对象。

默认的清除策略是 LRU。

一级缓存

```java
@Test
public void getUserById(){
    SqlSession session = MyBatisUtils.getSession();
    UserMapper mapper = session.getMapper(UserMapper.class);

    User user = mapper.getUserById(1);
    System.out.println(user);
    System.out.println("==============启用一级缓存不会再执行SQL==============");
    System.out.println(user);
    session.close();

}
}
```

session.close()之前在执行同一个sql不会再次连接数据库(如果其中用了增删改查,则会清除缓存), 也可以用 sqlSession.clearCache()手动清理

- 映射语句文件中的所有 insert、update 和 delete 语句会刷新缓存。
- 缓存会使用最近最少使用算法（LRU, Least Recently Used）算法来清除不需要的缓存。
- 缓存不会定时进行刷新（也就是说，没有刷新间隔）。
- 缓存会保存列表或对象（无论查询方法返回哪种）的 1024 个引用。
- 缓存会被视为读/写缓存，这意味着获取到的对象并不是共享的，可以安全地被调用者修改，而不干扰其他调用者或线程所做的潜在修改。

一级缓存相当于一个map

二级缓存
<cache /> (需要开启全局缓存 <setting>)

```xml
<!--    需要去setting开启全局缓存-->
    <cache />

    <select id="getUserById" parameterType="_int" resultMap="iniUser" useCache="false">
        select *
        from user where id = #{id};
    </select>
```

可以手动不使用缓存

cache配置

```xml
<!--    需要去setting开启全局缓存-->
<cache eviction="FIFO"
       flushInterval="60000"
       size="512"
       readOnly="true"/>

<select id="getUserById" parameterType="_int" resultMap="iniUser" useCache="false">
    select *
    from user where id = #{id};
```

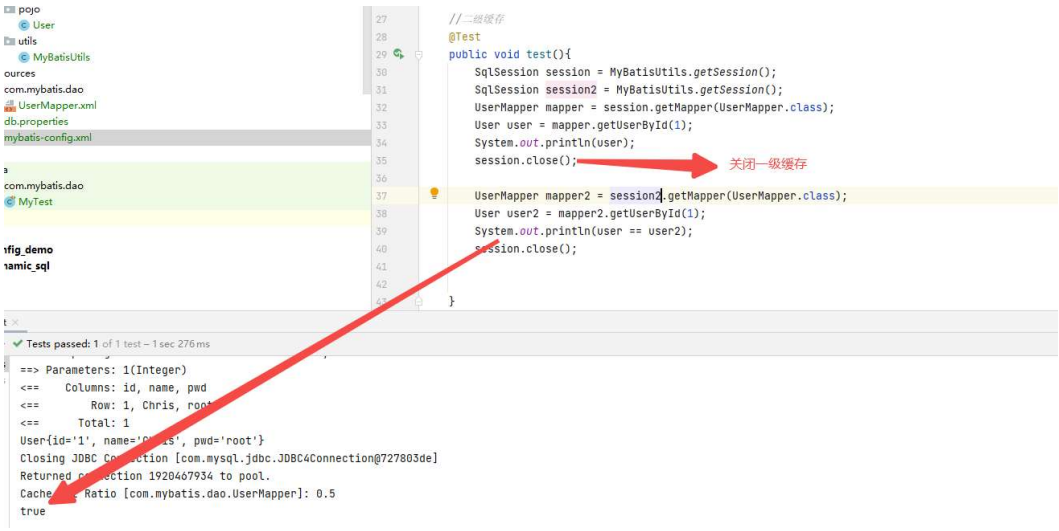
输入输出

刷新缓存时间 60s(以毫秒为单位)

最大容量

二级工作机制

- 一个会话查询一条数据,数据放在一级缓存
- 当一级缓存关闭了, 数据就会被保存到二级缓存中
- 新的查询信息, 可以从二级缓存获取内容
- 不同的mapper查出来的数据会放在对应的缓存map中



注意点

- 需要将实体类序列化, 否则会报错

```
**/
public class User implements Serializable {
    private String id;
    private String name;
```

Ehcache自定义缓存(分布式缓存)

```
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.1.0</version>
</dependency>
```

```
<cache
        name="cloud_user"
        eternal="false"
        maxElementsInMemory="5000"
        overflowToDisk="false"
        diskPersistent="false"
        timeToIdleSeconds="1800"
        timeToLiveSeconds="1800"
        memoryStoreEvictionPolicy="LRU"/>
<!--
   defaultCache：默认缓存策略，当ehcache找不到定义的缓存时，则使用这个缓存策略。只能定义一个。
 -->
<!--
  name:缓存名称。
  maxElementsInMemory:缓存最大数目
  maxElementsOnDisk：硬盘最大缓存个数。
  eternal:对象是否永久有效，一但设置了，timeout将不起作用。
  overflowToDisk:是否保存到磁盘，当系统当机时
  timeToIdleSeconds:设置对象在失效前的允许闲置时间（单位：秒）。仅当eternal=false对象不是永久有效时使用，可选属性，默认值是0，也就是
  timeToLiveSeconds:设置对象在失效前允许存活时间（单位：秒）。最大时间介于创建时间和失效时间之间。仅当eternal=false对象不是永久有效时
  diskPersistent：是否缓存虚拟机重启期数据 Whether the disk store persists between restarts of the Virtual Machine. The defaul
  diskSpoolBufferSizeMB：这个参数设置DiskStore（磁盘缓存）的缓存区大小。默认是30MB。每个Cache都应该有自己的一个缓冲区。
  diskExpiryThreadIntervalSeconds：磁盘失效线程运行时间间隔，默认是120秒。
  memoryStoreEvictionPolicy：当达到maxElementsInMemory限制时，Ehcache将会根据指定的策略去清理内存。默认策略是LRU（最近最少使用）。
  clearOnFlush：内存数量最大时是否清除。
  memoryStoreEvictionPolicy:可选策略有：LRU（最近最少使用，默认策略）、FIFO（先进先出）、LFU（最少访问次数）。
  FIFO，first in first out，这个是大家最熟的，先进先出。
  LFU,  Less Frequently Used，就是上面例子中使用的策略，直白一点就是讲一直以来最少被使用的。如上面所讲，缓存的元素有一个hit属性，hit
  LRU，Least Recently Used，最近最少使用的，缓存的元素有一个时间戳，当缓存容量满了，而又需要腾出地方来缓存新的元素的时候，那么现有缓存
-->

</ehcache>
```