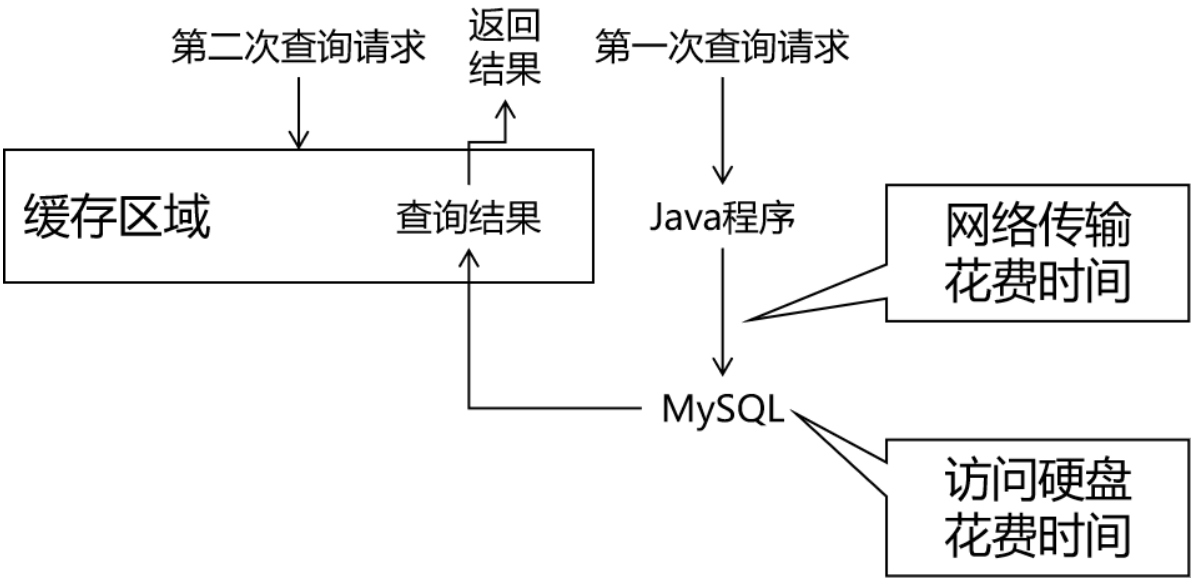
Mybatis-day04

第一章 Mybatis的缓存机制

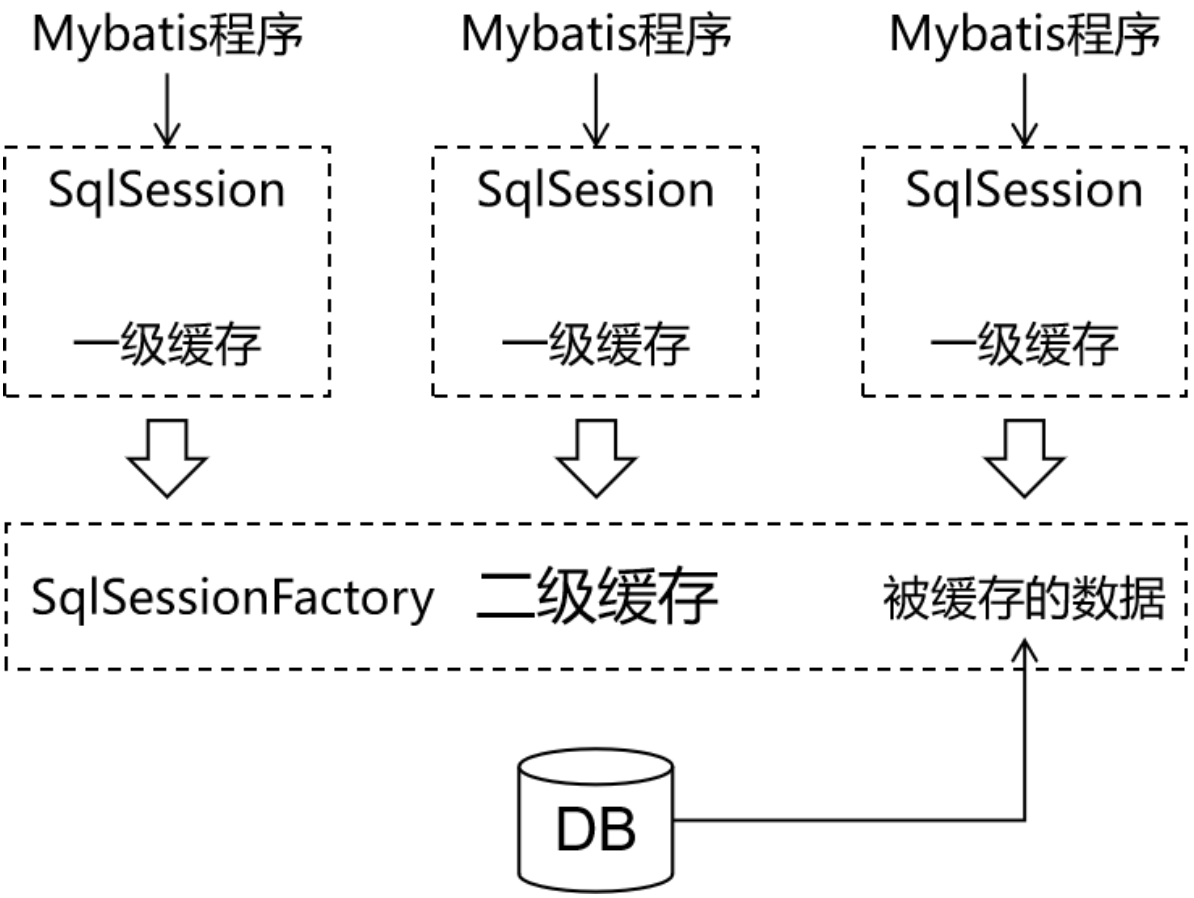
第一节 缓存机制的概述

1. 什么是缓存



2. 一级缓存和二级缓存的对比

2.1 使用顺序

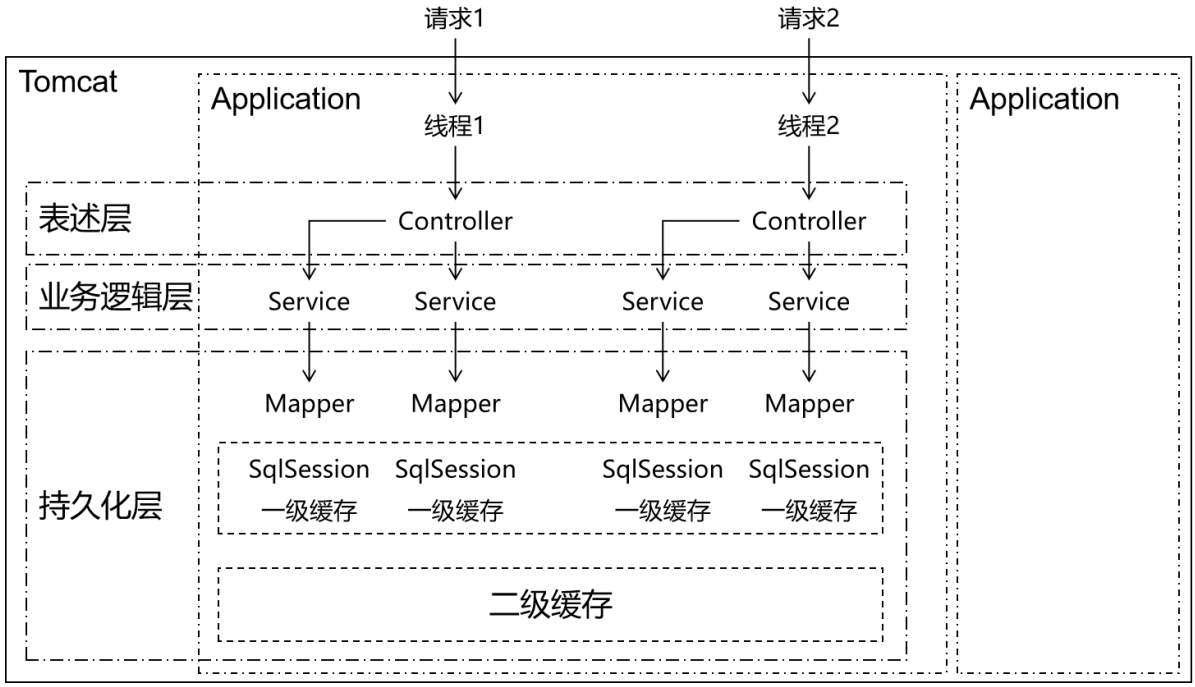


查询的顺序是：

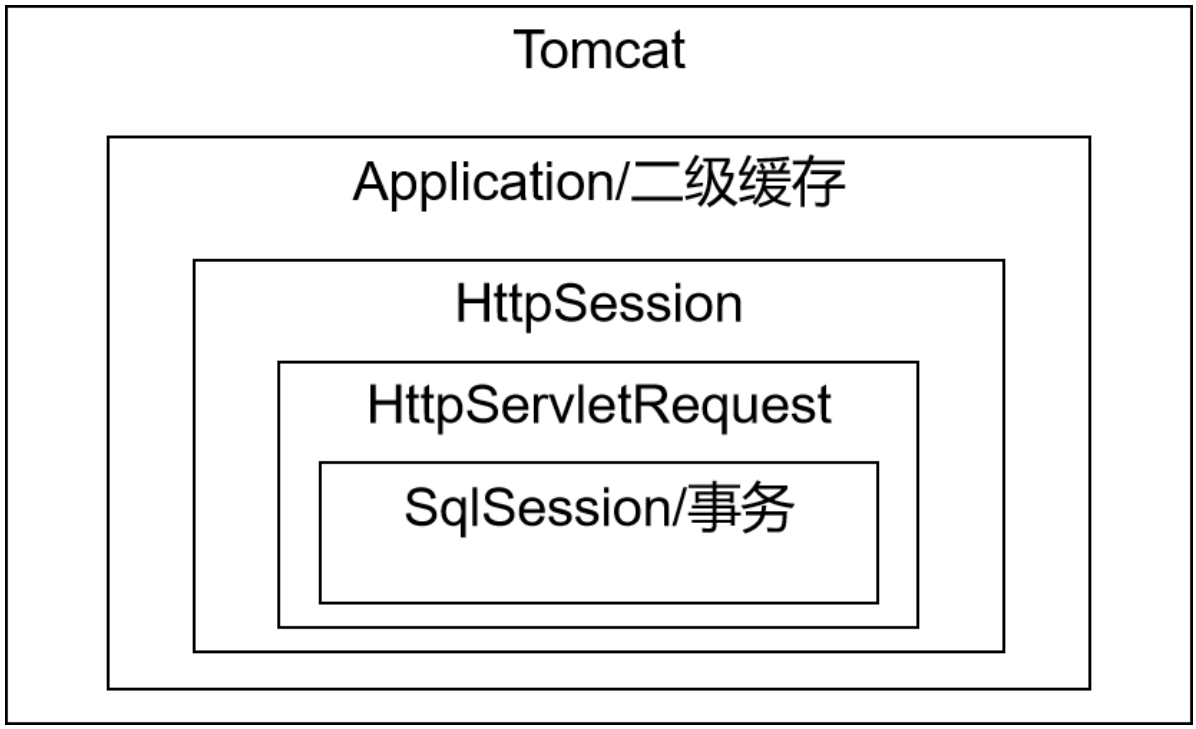
- 先查询二级缓存，因为二级缓存中可能会有其他SqlSession已经查出来的数据，可以拿来直接使用。
- 如果二级缓存没有命中，再查询一级缓存
- 如果一级缓存也没有命中，则查询数据库，查询到数据之后会把数据写入到一级缓存中
- SqlSession关闭之前，一级缓存中的数据会写入二级缓存

2.2 作用范围

- 一级缓存：SqlSession级别(同一次操作中，才能共用同一个sqlSession)
- 二级缓存：SqlSessionFactory级别(整个项目中都是共用一个SqlSessionFactory)



它们之间范围的大小参考下面图：



第二节 一级缓存

1 代码验证一级缓存

```
@Test
public void testFirstLevelCache(){
    //验证一级缓存的存在:要求两次查询使用的是同一个sqlSession对象
    //一级缓存是Mybatis自动开启的，不需要配置，也无法关闭(表示你必须使用一级缓存)
    SqlSession sqlSession = sessionFactory.openSession();
    EmployeeMapper employeeMapper = sqlSession.getMapper(EmployeeMapper.class);

    Employee employee1 = employeeMapper.selectEmployee(7);
    System.out.println(employee1);
    //一级缓存什么时候会被清除呢?
    //1. sqlSession提交事务   2. sqlSession调用clearCache()方法
    //3. sqlSession被销毁了close()一级缓存的内容会被写入到二级缓存
    //4. 数据发生改变:其实并没有清除一级缓存，而是修改缓存中数据
    SqlSession sqlSession2 = sessionFactory.openSession();
    EmployeeMapper employeeMapper2 =
sqlSession2.getMapper(EmployeeMapper.class);
    employee1.setEmpName("zs");
    employeeMapper2.updateEmployee(employee1);
    sqlSession2.commit();


    Employee employee2 = employeeMapper.selectEmployee(7);
    System.out.println(employee2);
}
```

一共只打印了一条SQL语句。

2 一级缓存失效的情况

- 不是同一个SqlSession(因为一级缓存只能用在同一个SqlSession中)
- 同一个SqlSession但是查询条件发生了变化
- 同一个SqlSession两次查询期间执行了任何一次增删改操作,那么会改变缓存的数据
- 同一个SqlSession两次查询期间手动清空了缓存: 调用sqlSession的clearCache()方法
- 同一个SqlSession两次查询期间提交了事务: 调用sqlSession的commit()方法

第三节 二级缓存

1 代码测试二级缓存

1.1 开启二级缓存功能

在想要使用二级缓存的Mapper配置文件中加入cache标签

```
<!-- 加入cache标签启用二级缓存功能 -->
<cache/>
```

1.2 让实体类支持序列化

```
public class Employee implements Serializable {
}
```

1.3 junit测试

这个功能的测试操作需要将SqlSessionFactory对象设置为成员变量

```
@Test
public void testSecondCacheLevel(){
    //测试二级缓存的存在:二级缓存是使用在不同的SqlSession中，但是是同一个SqlSessionFactory
    //二级缓存不是Mybatis自动开启的，需要我们手动进行配置
    //1. 在要进行二级缓存的映射配置文件中开启二级缓存(使用<cache/>标签)
    //2. 要进行二级缓存的POJO类需要实现Serializable接口(进行序列化:将对象的数据存储到硬盘
中)
    //3. 二级缓存的创建时机(什么时候数据会写入到二级缓存中):sqlSession对象close()的时候
    SqlSession sqlSession1 = sessionFactory.openSession();
    EmployeeMapper employeeMapper1 =
sqlSession1.getMapper(EmployeeMapper.class);

    SqlSession sqlSession2 = sessionFactory.openSession();
    EmployeeMapper employeeMapper2 =
sqlSession2.getMapper(EmployeeMapper.class);

    Employee employee1 = employeeMapper1.selectEmployee(7);
    System.out.println(employee1);
    //sqlSession1关闭，那么sqlSession1查询到的数据就会写入到二级缓存中
    sqlSession1.close();

    Employee employee2 = employeeMapper2.selectEmployee(7);
    System.out.println(employee2);
    sqlSession2.close();
}
```

1.4 缓存命中率

日志中打印的Cache Hit Ratio叫做缓存命中率

```
Cache Hit Ratio [com.atguigu.mybatis.EmployeeMapper]: 0.0（0/1）
Cache Hit Ratio [com.atguigu.mybatis.EmployeeMapper]: 0.5（1/2）
Cache Hit Ratio [com.atguigu.mybatis.EmployeeMapper]: 0.6666666666666666（2/3）
Cache Hit Ratio [com.atguigu.mybatis.EmployeeMapper]: 0.75（3/4）
Cache Hit Ratio [com.atguigu.mybatis.EmployeeMapper]: 0.8（4/5）
```

缓存命中率=命中缓存的次数/查询的总次数

2 查询结果存入二级缓存的时机

结论：SqlSession关闭或者提交的时候，一级缓存中的内容会被存入二级缓存

```
// 1.开启两个SqlSession
SqlSession session01 = factory.openSession();
SqlSession session02 = factory.openSession();

// 2.获取两个EmployeeMapper
```

```
EmployeeMapper employeeMapper01 = session01.getMapper(EmployeeMapper.class);
EmployeeMapper employeeMapper02 = session02.getMapper(EmployeeMapper.class);

// 3.使用两个EmployeeMapper做两次查询，返回两个Employee对象
Employee employee01 = employeeMapper01.selectEmployeeById(2);
Employee employee02 = employeeMapper02.selectEmployeeById(2);

// 4.比较两个Employee对象
System.out.println("employee02.equals(employee01) = " +
employee02.equals(employee01));
```

上面代码打印的结果是：

```
DEBUG 12-01 10:10:32,209 Cache Hit Ratio [com.atguigu.mybatis.EmployeeMapper]:
0.0  (LoggingCache.java:62)
DEBUG 12-01 10:10:32,570 ==>  Preparing: select
emp_id,emp_name,emp_salary,emp_gender,emp_age from t_emp where emp_id=?
(BaseJdbcLogger.java:145)
DEBUG 12-01 10:10:32,624 ==> Parameters: 2(Integer)  (BaseJdbcLogger.java:145)
DEBUG 12-01 10:10:32,643 <==      Total: 1  (BaseJdbcLogger.java:145)
DEBUG 12-01 10:10:32,644 Cache Hit Ratio [com.atguigu.mybatis.EmployeeMapper]:
0.0  (LoggingCache.java:62)
DEBUG 12-01 10:10:32,661 ==>  Preparing: select
emp_id,emp_name,emp_salary,emp_gender,emp_age from t_emp where emp_id=?
(BaseJdbcLogger.java:145)
DEBUG 12-01 10:10:32,662 ==> Parameters: 2(Integer)  (BaseJdbcLogger.java:145)
DEBUG 12-01 10:10:32,665 <==      Total: 1  (BaseJdbcLogger.java:145)
employee02.equals(employee01) = false
```

修改代码：

```
// 1.开启两个SqlSession
SqlSession session01 = factory.openSession();
SqlSession session02 = factory.openSession();

// 2.获取两个EmployeeMapper
EmployeeMapper employeeMapper01 = session01.getMapper(EmployeeMapper.class);
EmployeeMapper employeeMapper02 = session02.getMapper(EmployeeMapper.class);

// 3.使用两个EmployeeMapper做两次查询，返回两个Employee对象
Employee employee01 = employeeMapper01.selectEmployeeById(2);

// ※第一次查询完成后，把所在的SqlSession关闭，使一级缓存中的数据存入二级缓存
session01.close();
Employee employee02 = employeeMapper02.selectEmployeeById(2);

// 4.比较两个Employee对象
System.out.println("employee02.equals(employee01) = " +
employee02.equals(employee01));

// 5.另外一个SqlSession用完正常关闭
session02.close();
```

打印结果：

```
DEBUG 12-01 10:14:06,804 Cache Hit Ratio [com.atguigu.mybatis.EmployeeMapper]:
0.0  (LoggingCache.java:62)
DEBUG 12-01 10:14:07,135 ==>  Preparing: select
emp_id,emp_name,emp_salary,emp_gender,emp_age from t_emp where emp_id=?
(BaseJdbcLogger.java:145)
DEBUG 12-01 10:14:07,202 ==> Parameters: 2(Integer)  (BaseJdbcLogger.java:145)
DEBUG 12-01 10:14:07,224 <==      Total: 1  (BaseJdbcLogger.java:145)
DEBUG 12-01 10:14:07,308 Cache Hit Ratio [com.atguigu.mybatis.EmployeeMapper]:
0.5  (LoggingCache.java:62)
employee02.equals(employee01) = false
```

3 二级缓存相关配置(了解)

在Mapper配置文件中添加的cache标签可以设置一些属性：

- eviction属性：缓存回收策略

  LRU（Least Recently Used） – 最近最少使用的：移除最长时间不被使用的对象。

  FIFO（First in First out） – 先进先出：按对象进入缓存的顺序来移除它们。

  SOFT – 软引用：移除基于垃圾回收器状态和软引用规则的对象。

  WEAK – 弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象。

  默认的是 LRU。

- flushInterval属性：刷新间隔，单位毫秒

  默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新

- size属性：引用数目，正整数

  代表缓存最多可以存储多少个对象，太大容易导致内存溢出

- readOnly属性：只读，true/false

  true：只读缓存；会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。

  false：读写缓存；会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是false。

第四节 整合EHCache

1. EHCache简介

Ehcache 是一种开源的、基于标准的缓存，可提高性能、卸载数据库并简化可扩展性。它是最广泛使用的基于 Java 的缓存，因为它健壮、经过验证、功能齐全，并且与其他流行的库和框架集成。 Ehcache 从进程内缓存一直扩展到具有 TB 级缓存的混合进程内/进程外部署。 官网地址为：https://www.ehcache.org/

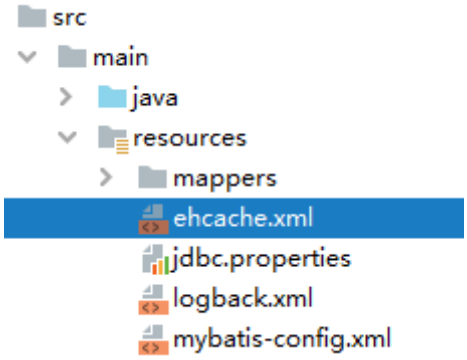
2. Mybatis整合操作

2.1 添加依赖

```
<!-- Mybatis EHCache整合包 -->
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.2.1</version>
</dependency>
<!-- slf4j日志门面的一个具体实现 -->
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.3</version>
</dependency>
```

2.2 创建EHCache配置文件

文件路径必须在resources根路径下，文件名必须: ehcache.xml



文件内容

```
<?xml version="1.0" encoding="utf-8" ?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd">
    <!-- 磁盘保存路径 -->
    <diskStore path="D:\atguigu\ehcache"/>
    <!--
    maxElementsInMemory:设置 在内存中缓存 对象的个数
    maxElementsOnDisk：设置 在硬盘中缓存 对象的个数
    eternal：设置缓存是否 永远不过期
    overflowToDisk：当系统宕机的时候是否保存到磁盘上
    maxElementsInMemory的时候，是否转移到硬盘中
    timeToIdleSeconds：当2次访问 超过该值的时候，将缓存对象失效
    timeToLiveSeconds：一个缓存对象 最多存放的时间（生命周期）
    diskExpiryThreadIntervalSeconds：设置每隔多长时间，通过一个线程来清理硬盘中的缓存
    clearOnFlush： 内存数量最大时是否清除
    memoryStoreEvictionPolicy：当超过缓存对象的最大值时，处理的策略；LRU （最少使用），FIFO
(先进先出)， LFU （最少访问次数）
    -->
    <defaultCache
                  maxElementsInMemory="1000"
                  maxElementsOnDisk="10000000"
                  eternal="false"
                  overflowToDisk="true"
                  timeToIdleSeconds="120"
                  timeToLiveSeconds="120"
                  diskExpiryThreadIntervalSeconds="120"
                  memoryStoreEvictionPolicy="LRU">
```

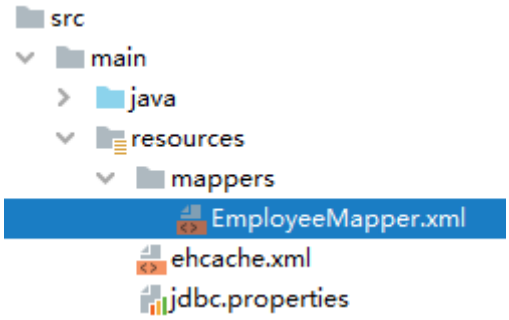
```
    </defaultCache>
</ehcache>
```

引入第三方框架或工具时，配置文件的文件名可以自定义吗?

- 可以自定义：文件名是由我告诉其他环境
- 不能自定义：文件名是框架内置的、约定好的，就不能自定义，以避免框架无法加载这个文件

2.3 指定缓存管理器的具体类型

还是到查询操作所的Mapper配置文件中，找到之前设置的cache标签：




```
<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
```

2.4 加入logback日志

存在SLF4J时，作为简易日志的log4j将失效，此时我们需要借助SLF4J的具体实现logback来打印日志。

2.4.1 各种Java日志框架简介

门面：

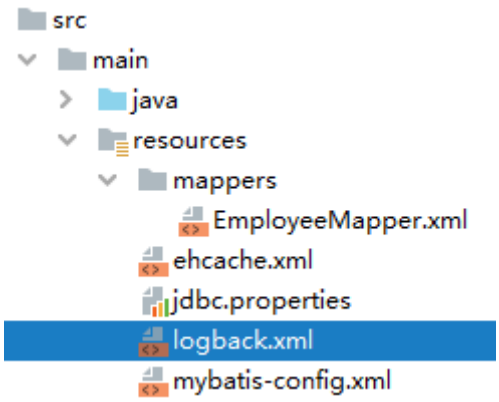
名称	说明
JCL（Jakarta Commons Logging）	陈旧
SLF4J（Simple Logging Facade for Java）★	适合
jboss-logging	特殊专业领域使用

实现：

名称	说明
log4j★	最初版
JUL（java.util.logging）	JDK自带
log4j2	Apache收购log4j后全面重构，内部实现和log4j完全不同
logback★	优雅、强大

注：标记★的技术是同一作者。

2.4.2 logback配置文件



配置文件存储位置必须在resources的根路径下，配置文件的名字必须叫做logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="true">
    <!-- 指定日志输出的位置 -->
    <appender name="STDOUT"
        class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <!-- 日志输出的格式 -->
            <!-- 按照顺序分别是：时间、日志级别、线程名称、打印日志的类、日志主体内容、换行 -
->
            <pattern>[%d{HH:mm:ss.SSS}] [%-5level] [%thread] [%logger]
[%msg]%n</pattern>
        </encoder>
    </appender>

    <!-- 设置全局日志级别。日志级别按顺序分别是：DEBUG、INFO、WARN、ERROR -->
    <!-- 指定任何一个日志级别都只打印当前级别和后面级别的日志。 -->
    <root level="DEBUG">
        <!-- 指定打印日志的appender，这里通过“STDOUT”引用了前面配置的appender -->
        <appender-ref ref="STDOUT" />
    </root>

    <!-- 根据特殊需求指定局部日志级别 -->
    <logger name="com.atguigu.crowd.mapper" level="DEBUG"/>

</configuration>
```

2.4.3 junit测试

正常按照二级缓存的方式测试即可。因为整合EHCache后，其实就是使用EHCache代替了Mybatis自带的二级缓存。

2.4.4 EHCache配置文件说明

当借助CacheManager.add("缓存名称")创建Cache时，EhCache便会采用指定的的管理策略。

defaultCache标签各属性说明：

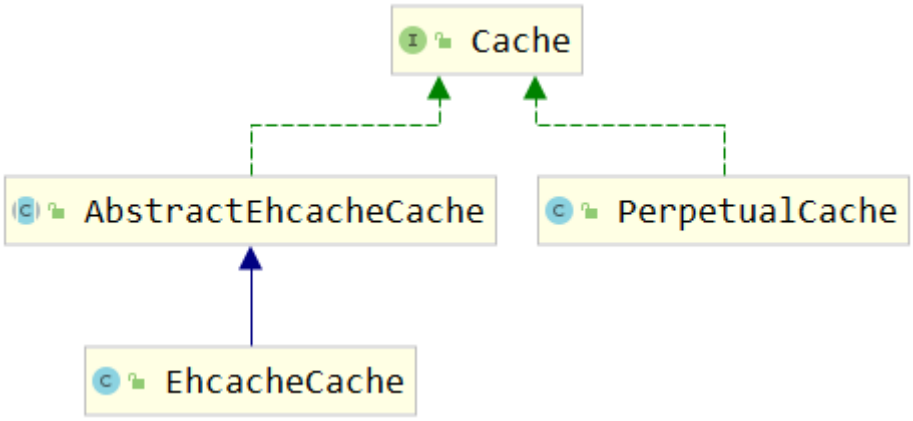
属性名	是否必须	作用
maxElementsInMemory	是	在内存中缓存的element的最大数目
maxElementsOnDisk	是	在磁盘上缓存的element的最大数目，若是0表示无穷大
eternal	是	设定缓存的elements是否永远不过期。 如果为true，则缓存的数据始终有效， 如果为false那么还要根据timeToIdleSeconds、timeToLiveSeconds判断
overflowToDisk	是	设定当内存缓存溢出的时候是否将过期的element缓存到磁盘上
timeToIdleSeconds	否	当缓存在EhCache中的数据前后两次访问的时间超过timeToIdleSeconds的属性取值时， 这些数据便会删除，默认值是0,也就是可闲置时间无穷大
timeToLiveSeconds	否	缓存element的有效生命期，默认是0.,也就是element存活时间无穷大
diskSpoolBufferSizeMB	否	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
diskPersistent	否	在VM重启的时候是否启用磁盘保存EhCache中的数据，默认是false。
diskExpiryThreadIntervalSeconds	否	磁盘缓存的清理线程运行间隔，默认是120秒。每个120s， 相应的线程会进行一次EhCache中数据的清理工作
memoryStoreEvictionPolicy	否	当内存缓存达到最大，有新的element加入的时候，移除缓存中element的策略。 默认是LRU（最近最少使用），可选的有LFU（最不常使用）和FIFO（先进先出）

第五节 缓存的原理

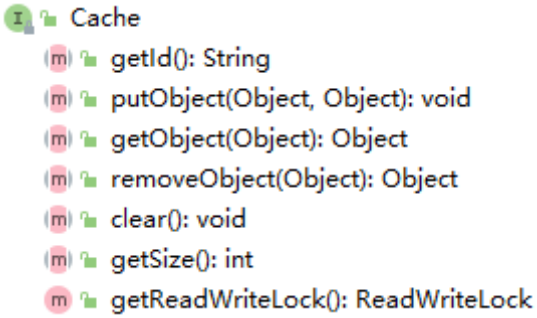
1. Cache接口

1.1 Cache接口的重要地位

org.apache.ibatis.cache.Cache接口：所有缓存都必须实现的顶级接口



1.2 Cache接口中的方法

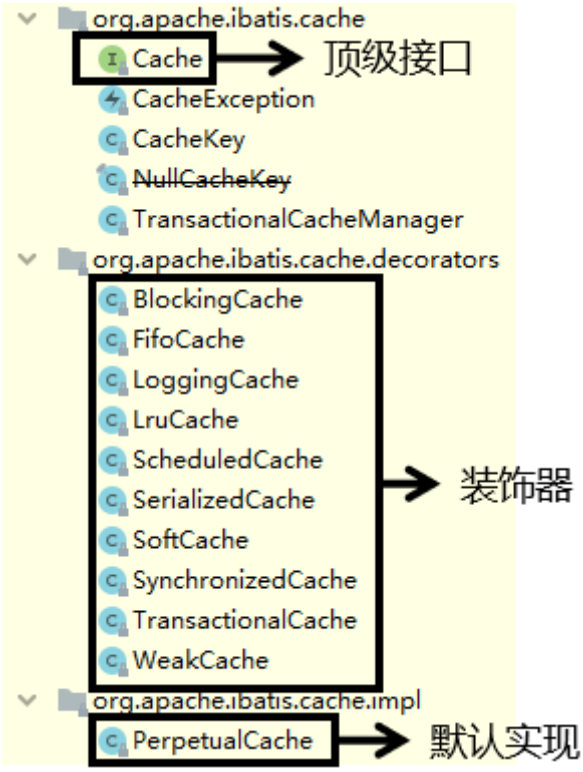


方法名	作用
putObject()	将对象存入缓存
getObject()	从缓存中取出对象
removeObject()	从缓存中删除对象

1.3 缓存的本质

根据Cache接口中方法的声明我们能够看到，缓存的本质是一个**Map**。

2. PerpetualCache类

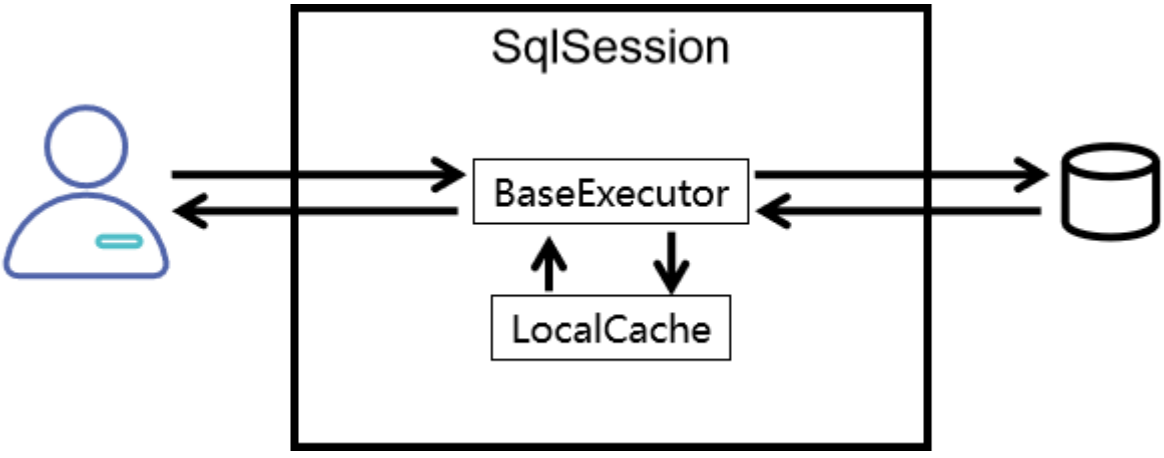


org.apache.ibatis.cache.impl.PerpetualCache是Mybatis的默认缓存，也是Cache接口的默认实现。Mybatis一级缓存和自带的二级缓存都是通过PerpetualCache来操作缓存数据的。但是这就奇怪了，同样是PerpetualCache这个类，怎么能区分出来两种不同级别的缓存呢?

其实很简单，调用者不同。

- 一级缓存：由BaseExecutor调用PerpetualCache
- 二级缓存：由CachingExecutor调用PerpetualCache，而CachingExecutor可以看做是对BaseExecutor的装饰

3. 一级缓存机制



org.apache.ibatis.executor.BaseExecutor类中的关键方法：

3.1 query()方法

```
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws
SQLException {
    ErrorContext.instance().resource(ms.getResource()).activity("executing a
query").object(ms.getId());
    if (closed) {
        throw new ExecutorException("Executor was closed.");
```

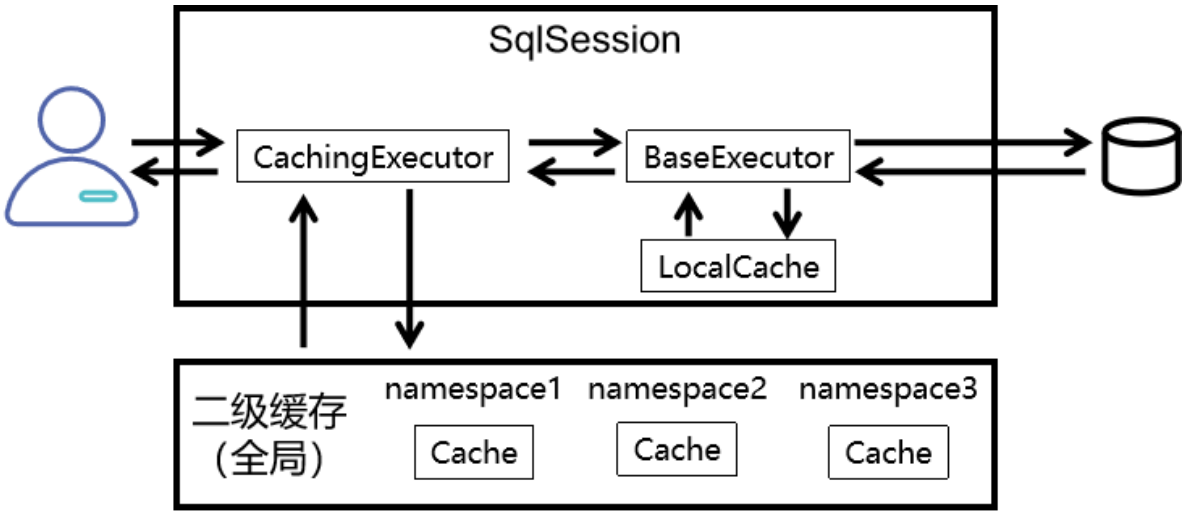
```
    }
    if (queryStack == 0 && ms.isFlushCacheRequired()) {
        clearLocalCache();
    }
    List<E> list;
    try {
        queryStack++;

        // 尝试从本地缓存中获取数据
        list = resultHandler == null ? (List<E>) localCache.getObject(key) :
null;

        if (list != null) {
            handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
        } else {

            // 如果本地缓存中没有查询到数据，则查询数据库
            list = queryFromDatabase(ms, parameter, rowBounds, resultHandler,
key, boundSql);
        }
    } finally {
        queryStack--;
    }
    if (queryStack == 0) {
        for (org.apache.ibatis.executor.BaseExecutor.DeferredLoad deferredLoad :
deferredLoads) {
            deferredLoad.load();
        }
        // issue #601
        deferredLoads.clear();
        if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
            // issue #482
            clearLocalCache();
        }
    }
    return list;
}
```

3.2 queryFromDatabase()方法

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

        // 从数据库中查询数据
        list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
    } finally {
        localCache.removeObject(key);
    }

    // 将数据存入本地缓存
    localCache.putObject(key, list);
    if (ms.getStatementType() == StatementType.CALLABLE) {
        localOutputParameterCache.putObject(key, parameter);
```

```
    }
    return list;
}
```

4. 二级缓存机制



下面我们来看看CachingExecutor类中的query()方法在不同情况下使用的具体缓存对象：

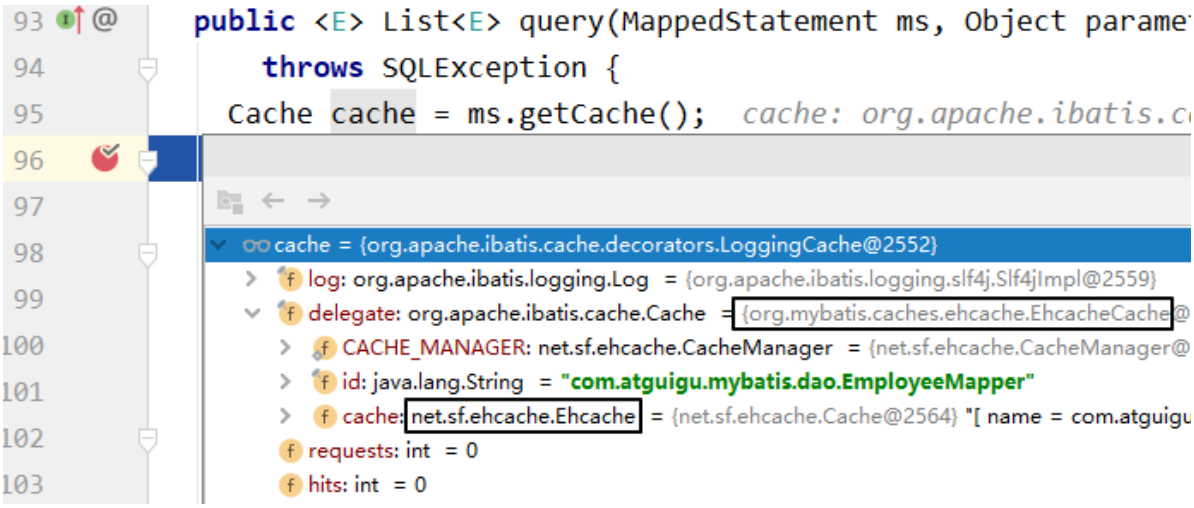
4.1 未开启二级缓存

```
public <E> List<E> query(MappedStatement ms, Object par
    throws SQLException {
  Cache cache = ms.getCache();  cache: null  ms: org.ap
  if (cache != null) {  cache: null
    flushCacheIfRequired(ms);
    if (ms.isUseCache() && resultHandler == null) {
```

4.1 使用Mybatis自带的二级缓存

```
93  @   public <E> List<E> query(MappedStatement ms, Object parameterObject, Rc
94          throws SQLException {
95      Cache cache = ms.getCache();  cache: org.apache.ibatis.cache.decorato
96                                                                    cache
97
98          cache = {org.apache.ibatis.cache.decorators.SynchronizedCache@2043}
99              delegate: org.apache.ibatis.cache.Cache = {org.apache.ibatis.cache.decorators.LoggingCache@2050}
100                 log: org.apache.ibatis.logging.Log = {org.apache.ibatis.logging.slf4j.Slf4jImpl@2051}
101             delegate: org.apache.ibatis.cache.Cache = {org.apache.ibatis.cache.decorators.SerializedCache@2052}
102                 delegate: org.apache.ibatis.cache.Cache = {org.apache.ibatis.cache.decorators.LruCache@2053}
103                     delegate: org.apache.ibatis.cache.Cache = {org.apache.ibatis.cache.impl.PerpetualCache@2054}
104                         id: java.lang.String = "com.atguigu.mybatis.dao.EmployeeMapper"
105                         cache: java.util.Map = {java.util.HashMap@2057} size = 0
106                         keyMap: java.util.Map = {org.apache.ibatis.cache.decorators.LruCache$1@2055} size = 0
                            eldestKey: java.lang.Object = null
                    requests: int = 0
                    hits: int = 0
```

4.1 使用EHCache

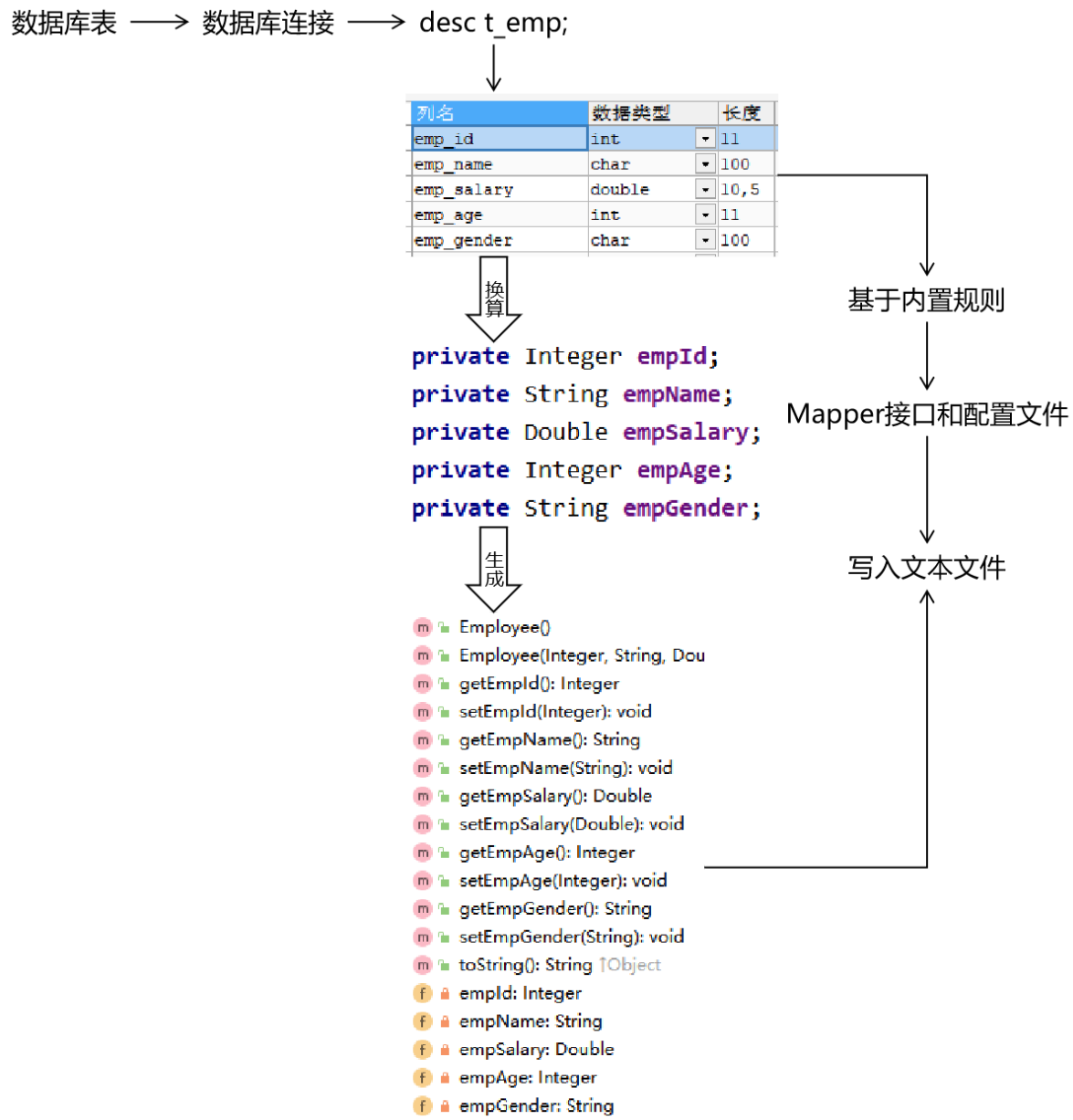


第二章 逆向工程

第一节 概念

- 正向工程：先创建Java实体类，由框架负责根据实体类生成数据库表。Hibernate是支持正向工程的。
- 逆向工程：先创建数据库表，由框架负责根据数据库表，反向生成如下资源：
  - Java实体类
  - Mapper接口
  - Mapper配置文件

第二节 基本原理



第三节 逆向工程的具体操作

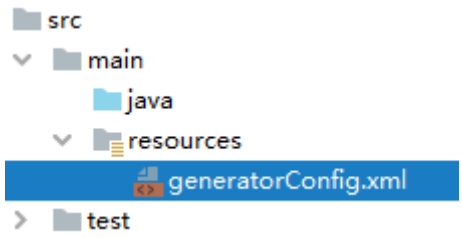
1. 配置POM

```
<!-- 依赖MyBatis核心包 -->
<dependencies>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.7</version>
    </dependency>
    <!--mysql驱动-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.3</version>
        <scope>runtime</scope>
    </dependency>
    <!--log4j-->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
```

```
        <version>1.2.17</version>
    </dependency>
    <!--junit-->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <!--lombok-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.8</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

<!-- 控制Maven在构建过程中相关配置 -->
<build>

    <!-- 构建过程中用到的插件 -->
    <plugins>

        <!-- 具体插件，逆向工程的操作是以构建过程中插件形式出现的 -->
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifactId>
            <version>1.3.0</version>

            <!-- 插件的依赖 -->
            <dependencies>

                <!-- 逆向工程的核心依赖 -->
                <dependency>
                    <groupId>org.mybatis.generator</groupId>
                    <artifactId>mybatis-generator-core</artifactId>
                    <version>1.3.2</version>
                </dependency>

                <!-- 数据库连接池 -->
                <dependency>
                    <groupId>com.mchange</groupId>
                    <artifactId>c3p0</artifactId>
                    <version>0.9.2</version>
                </dependency>

                <!-- MySQL驱动 -->
                <dependency>
                    <groupId>mysql</groupId>
                    <artifactId>mysql-connector-java</artifactId>
                    <version>5.1.8</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
```

2. MBG配置文件

文件名必须是:generatorConfig.xml，而且必须放在类路径下

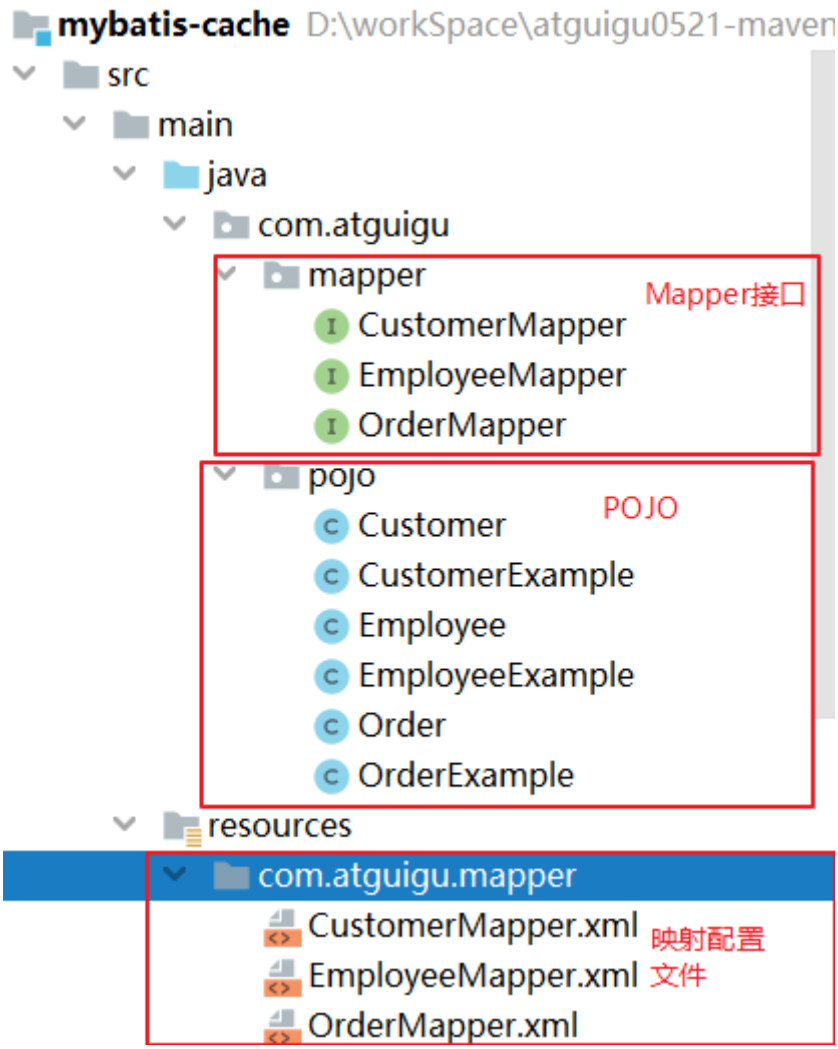



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
        PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
        "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>
    <!--
            targetRuntime: 执行生成的逆向工程的版本
                    MyBatis3Simple: 生成基本的CRUD（清新简洁版）
                    MyBatis3: 生成带条件的CRUD（奢华尊享版）
     -->
    <context id="DB2Tables" targetRuntime="MyBatis3">
        <!-- 数据库的连接信息 -->
        <jdbcConnection driverClass="com.mysql.jdbc.Driver"
                        connectionURL="jdbc:mysql://localhost:3306/mybatis-
example"
                        userId="root"
                        password="123456">
        </jdbcConnection>
        <!--
            javaBean的生成策略
            targetPackage 表示生成的JavaBean存放到哪个包中
            targetProject 表示生成的JavaBean存放到哪个主目录中
        -->
        <javaModelGenerator targetPackage="com.atguigu.pojo"
targetProject=".\src\main\java">
            <property name="enableSubPackages" value="true" />
            <property name="trimStrings" value="true" />
        </javaModelGenerator>
        <!--
            SQL映射文件的生成策略
         -->
        <sqlMapGenerator targetPackage="com.atguigu.mapper"
 targetProject=".\src\main\resources">
            <property name="enableSubPackages" value="true" />
        </sqlMapGenerator>
        <!-- Mapper接口的生成策略 -->
        <javaClientGenerator type="XMLMAPPER" targetPackage="com.atguigu.mapper"
 targetProject=".\src\main\java">
            <property name="enableSubPackages" value="true" />
        </javaClientGenerator>
        <!-- 逆向分析的表 -->
        <!-- tableName设置为*号，可以对应所有表，此时不写domainObjectName -->
        <!-- domainObjectName属性指定生成出来的实体类的类名 -->
        <table tableName="t_emp" domainObjectName="Employee"/>
        <table tableName="t_customer" domainObjectName="Customer"/>
        <table tableName="t_order" domainObjectName="Order"/>
```

```
    </context>
</generatorConfiguration>
```

3. 执行MBG插件的generate目标

4. 效果



5. 测试代码

```
package com.atguigu;

import com.atguigu.mapper.EmployeeMapper;
import com.atguigu.pojo.Employee;
import com.atguigu.pojo.EmployeeExample;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import java.io.IOException;
import java.io.InputStream;
import java.util.List;

/**
 * 包名:com.atguigu
 *
 * @author Leevi
 * 日期2021-08-28  14:06
 */
public class TestMybatis {
    private SqlSession sqlSession;
    private EmployeeMapper employeeMapper;
    private InputStream is;

    @Before
    public void init() throws IOException {
        //目标:创建出EmployeeManager接口的代理对象
        //1. 加载核心配置文件，转成字节输入流
        is = Resources.getResourceAsStream("mybatis-config.xml");
        //2. 创建SqlSessionFactoryBuilder对象
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
        //3. 构建出SqlSessionFactory
        SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(is);
        //4. 使用SqlSessionFactory对象创建出sqlSession对象
        sqlSession = sqlSessionFactory.openSession();
        //5. 使用sqlSession对象创建UserMapper接口的代理对象
        employeeMapper = sqlSession.getMapper(EmployeeMapper.class);
    }
    @After
    public void destroy() throws IOException {
        //提交事务
        sqlSession.commit();
        //关闭资源
        is.close();
        sqlSession.close();
    }

    @Test
    public void testInsertEmployee(){
```

```
        employeeMapper.insert(new Employee(null,"王五",3000d));
    }

    @Test
    public void testDeleteEmployeeByPrimaryKey(){
        employeeMapper.deleteByPrimaryKey(15014);
    }

    @Test
    public void testDeleteByExample(){
        //根据较为复杂的条件进行删除，比如要删除empId在2000到8000之间的所有员工
        //1. 创建一个EmployeeExample对象
        EmployeeExample employeeExample = new EmployeeExample();
        //2. 使用EmployeeExample对象获取criteria对象
        EmployeeExample.Criteria criteria = employeeExample.createCriteria();
        //3. 通过criteria来拼接条件
        criteria.andEmpIdBetween(3000,8000);
        employeeMapper.deleteByExample(employeeExample);
    }

    @Test
    public void testUpdateEmployee(){
        employeeMapper.updateByPrimaryKeySelective(new Employee(4,null,2000d));
    }

    @Test
    public void testQueryByPrimaryKey(){
        //根据主键查询
        Employee employee = employeeMapper.selectByPrimaryKey(4);
        System.out.println(employee);
    }

    @Test
    public void testQueryAll(){
        EmployeeExample employeeExample = new EmployeeExample();
        //查询所有数据
        List<Employee> employeeList =
employeeMapper.selectByExample(employeeExample);
    }

    @Test
    public void testQueryByExample(){
        //复杂条件:查询(名字中包含s，并且大于3000)  或者 (emp_id在8001-8234之间,并且名字
中包含3）的所有元素
        EmployeeExample employeeExample = new EmployeeExample();
        EmployeeExample.Criteria criteria1 = employeeExample.createCriteria();
        criteria1.andEmpNameLike("%s%")
                 .andEmpSalaryGreaterThan(3000d);

        EmployeeExample.Criteria criteria2 = employeeExample.or();
        criteria2.andEmpIdBetween(8001,8234)
                 .andEmpNameLike("%3%");
        employeeMapper.selectByExample(employeeExample);
    }
}
```

第四节 QBC查询

1. 概念

QBC：Query By Criteria， 最大的特点就是将SQL语句中的WHERE子句进行了组件化的封装，让我们可以通过调用Criteria对象的方法自由的拼装查询条件。

2. 例子

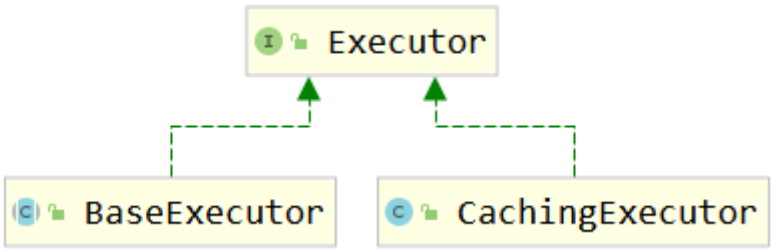
```
@Test
public void testQueryByExample(){
    //复杂条件:查询(名字中包含s，并且大于3000)  或者 (emp_id在8001-8234之间,并且名字中包含3）的所有元素
    EmployeeExample employeeExample = new EmployeeExample();
    EmployeeExample.Criteria criteria1 = employeeExample.createCriteria();
    criteria1.andEmpNameLike("%s%")
        .andEmpSalaryGreaterThan(3000d);

    EmployeeExample.Criteria criteria2 = employeeExample.or();
    criteria2.andEmpIdBetween(8001,8234)
        .andEmpNameLike("%3%");
    employeeMapper.selectByExample(employeeExample);
}
//实际执行的SQL语句:select emp_id, emp_name, emp_salary from t_emp WHERE ( emp_name like ? and emp_salary > ? ) or( emp_id between ? and ? and emp_name like ? )
```

第三章 Mybatis的其它补充内容(了解)

第一节 插件机制

1. Mybatis四大对象

1.1 Executor



1.2 ParameterHandler

```
public interface ParameterHandler {

  Object getParameterObject();

  void setParameters(PreparedStatement ps) throws SQLException;

}
```

2.3 ResultSetHandler

```
public interface ResultSetHandler {

  <E> List<E> handleResultSets(Statement stmt) throws SQLException;

  <E> Cursor<E> handleCursorResultSets(Statement stmt) throws SQLException;

  void handleOutputParameters(CallableStatement cs) throws SQLException;

}
```

2.4 StatementHandler

```
public interface StatementHandler {

  Statement prepare(Connection connection, Integer transactionTimeout)
      throws SQLException;

  void parameterize(Statement statement)
      throws SQLException;

  void batch(Statement statement)
      throws SQLException;

  int update(Statement statement)
      throws SQLException;

  <E> List<E> query(Statement statement, ResultHandler resultHandler)
      throws SQLException;

  <E> Cursor<E> queryCursor(Statement statement)
      throws SQLException;

  BoundSql getBoundSql();

  ParameterHandler getParameterHandler();

}
```

2. Mybatis插件机制的作用

插件是MyBatis提供的一个非常强大的机制，我们可以通过插件来修改MyBatis的一些核心行为。插件通过**动态代理**机制，可以介入四大对象的任何一个方法的执行。著名的Mybatis插件包括 PageHelper（分页插件）、通用 Mapper（SQL生成插件）等。

如果想编写自己的Mybatis插件可以通过实现org.apache.ibatis.plugin.Interceptor接口来完成，表示对Mybatis常规操作进行拦截，加入自定义逻辑。

```java
package org.apache.ibatis.plugin;

import java.util.Properties;

/**
 * @author Clinton Begin
 */
public interface Interceptor {

  Object intercept(Invocation invocation) throws Throwable;

  default Object plugin(Object target) { return Plugin.wrap(target, interceptor: this); }

  default void setProperties(Properties properties) {
    // NOP
  }

}
```

但是由于插件涉及到Mybatis底层工作机制，在没有足够把握时不要轻易尝试。

第二节 Mybatis底层的JDBC封装

org.apache.ibatis.executor.statement.PreparedStatementHandler类：

```java
public int update(Statement statement) throws SQLException {  statement: "org.apache.ibat
    PreparedStatement ps = (PreparedStatement)statement;  ps: "org.apache.ibatis.logging.
    ps.execute();  ps: "org.apache.ibatis.logging.jdbc.PreparedStatementLogger@2b91004a"
    int rows = ps.getUpdateCount();
    Object parameterObject = this.boundSql.getParameterObject();
    KeyGenerator keyGenerator = this.mappedStatement.getKeyGenerator();
    keyGenerator.processAfter(this.executor, this.mappedStatement, ps, parameterObject);
    return rows;
}
```

查找上面目标时，Debug查看源码的切入点是：

org.apache.ibatis.session.defaults.DefaultSqlSession类的update()方法

```java
public int update(String statement, Object parameter) {
    int var4;
    try {
        this.dirty = true;
        MappedStatement ms = this.configuration.getMappedStatement(statement);
        var4 = this.executor.update(ms, this.wrapCollection(parameter));
    } catch (Exception var8) {
        throw ExceptionFactory.wrapException("Error updating database.  Cause: " + var8, var8);
    } finally {
        ErrorContext.instance().reset();
    }

    return var4;
}
```

然后在分析`this.executor.update()方法`