

课程目标

内容定位

1. MyBatis的工作流程分析

- 1.1. 解析配置文件
- 1.2. 提供操作接口
- 1.3. 执行SQL操作

2. MyBatis架构分层与模块划分(总)

- 2.1. 接口层
- 2.2. 核心处理层
- 2.3. 基础支持层

3. MyBatis 缓存详解

- 3.1. cache缓存
- 3.2. 缓存体系结构
- 3.3. 一级缓存
 - 3.3.1. 一级缓存(本地缓存) 介绍
 - 3.3.2. 一级缓存验证
 - 3.3.3. 一级缓存的不足
- 3.4. 二级缓存
 - 3.4.1. 二级缓存介绍
 - 3.4.2. 开启二级缓存的方法
 - 3.4.3. 二级缓存验证
 - 3.4.3.1. 事务不提交，二级缓存不存在
 - 3.4.3.2. 使用不同的 session和mapper，并且提交事务，验证二级缓存可以跨 session存在
 - 3.4.3.3. 在其他的 session中执行增删改操作，验证缓存会被刷新
 - 3.4.4. 为什么增删改操作会清空缓存？
 - 3.4.5. 什么时候开启二级缓存？
 - 3.4.6. 第三方缓存做二级缓存

4. MyBatis源码解读

- 4.1. 带着问题看源码
- 4.2. 看源码的注意事项
- 4.3. 一. 配置解析过程
 - 4.3.1. XMLConfigBuilder
 - 4.3.2. propertiesElement()
 - 4.3.3. settingsAsProperties()
 - 4.3.4. loadCustomVfs (settings)

- 4.3.5. loadCustomLogImpl (settings)
- 4.3.6. typeAliasesElement()
- 4.3.7. pluginElement()
- 4.3.8. objectFactoryElement()、objectWrapperFactoryElement()、reflectorFactoryElement()
- 4.3.9. settingsElement (settings)
- 4.3.10. environmentsElement()
- 4.3.11. databaseIdProviderElement()
- 4.3.12. typeHandlerElement()
- 4.3.13. mapperElement()
 - 4.3.13.1. 1) configurationElement()
 - 4.3.13.2. 2) bindMapperForNamespace()
 - 4.3.13.3. 3) build()
- 4.3.14. 总结
- 4.4. 二. 会话创建过程
 - 4.4.1. 1、创建Transaction
 - 4.4.2. 2、创建 Executor
 - 4.4.2.1. 1) 创建执行器
 - 4.4.2.2. 2) 缓存装饰
 - 4.4.2.3. 3) 插件代理
 - 4.4.2.4. 4) 返回SqlSession 实现类
 - 4.4.3. 总结
- 4.5. 三. 获得Mapper对象
 - 4.5.1. 1、getMapper()
 - 4.5.2. 2、MapperProxy如何实现对接口的代理
 - 4.5.3. 3、总结
- 4.6. 四. 执行SQL
 - 4.6.1. 1、MapperProxy.invoke()
 - 4.6.2. 2、MapperMethod.execute()
 - 4.6.3. 3、DefaultSqlSession.selectOne()
 - 4.6.4. 4、CachingExecutor.query()
 - 4.6.4.1. 1) 创建 Cachekey
 - 4.6.4.2. 2) 处理二级缓存
 - 4.6.4.2.1. 1) 写入二级缓存
 - 4.6.4.2.2. 2) 获取二级缓存
 - 4.6.5. 5、BaseExecutor.query()
 - 4.6.5.1. 1) 清空本地缓存
 - 4.6.5.2. 2) 从数据库查询

4.6.6. 6、SimpleExecutor.doQuery()

4.6.6.1. 1) 创建 StatementHandler

4.6.6.2. 2) 创建Statement

4.6.6.3. 3) 执行的StatementHandler的query()方法

4.6.6.4. 4) 执行PreparedStatement的execute()

4.6.6.5. 5) ResultSetHandler 处理结果集

4.7. MyBatis核心对象

课程目标

- 1、掌握MyBatis的工作流程
- 2、掌握MyBatis的架构分层与模块划分
- 3、掌握MyBatis 缓存机制
- 4、通过阅读MyBatis 源码掌握MyBatis 底层工作原理与设计思想

内容定位

从宏观角度学习MyBatis的架构、工作流程、主要模块，从微观角度学习MyBatis的工作原理与设计思想，为手写MyBatis做准备。

适合已经掌握MyBatis基本使用方法的同学。

1. MyBatis的工作流程分析

MyBatis的主要工作流程：

1.1. 解析配置文件

首先在MyBatis启动的时候我们要去解析配置文件，包括全局配置文件和映射器配置文件，这里面包含了我们怎么控制MyBatis的行为，和我们要对数据库下达的指令，也就是我们的SQL信息。我们会把它们解析成一个Configuration对象。

1.2. 提供操作接口

接下来就是我们操作数据库的接口，它在应用程序和数据库中间，代表我们跟数据库之间的一次连接：这个就是SqlSession对象。

我们要获得一个会话，必须有一个会话工厂SqlSessionFactory。

SqlSessionFactory里面又必须包含我们的所有的配置信息，所以我们会通过一个Builder来创建工厂类。

MyBatis是对JDBC的封装，也就是意味着底层一定会出现JDBC的一些核心对象，比如执行SQL的Statement，结果集ResultSet。在Mybatis 里面，SqlSession只是提供给应用的一个接口，还不是

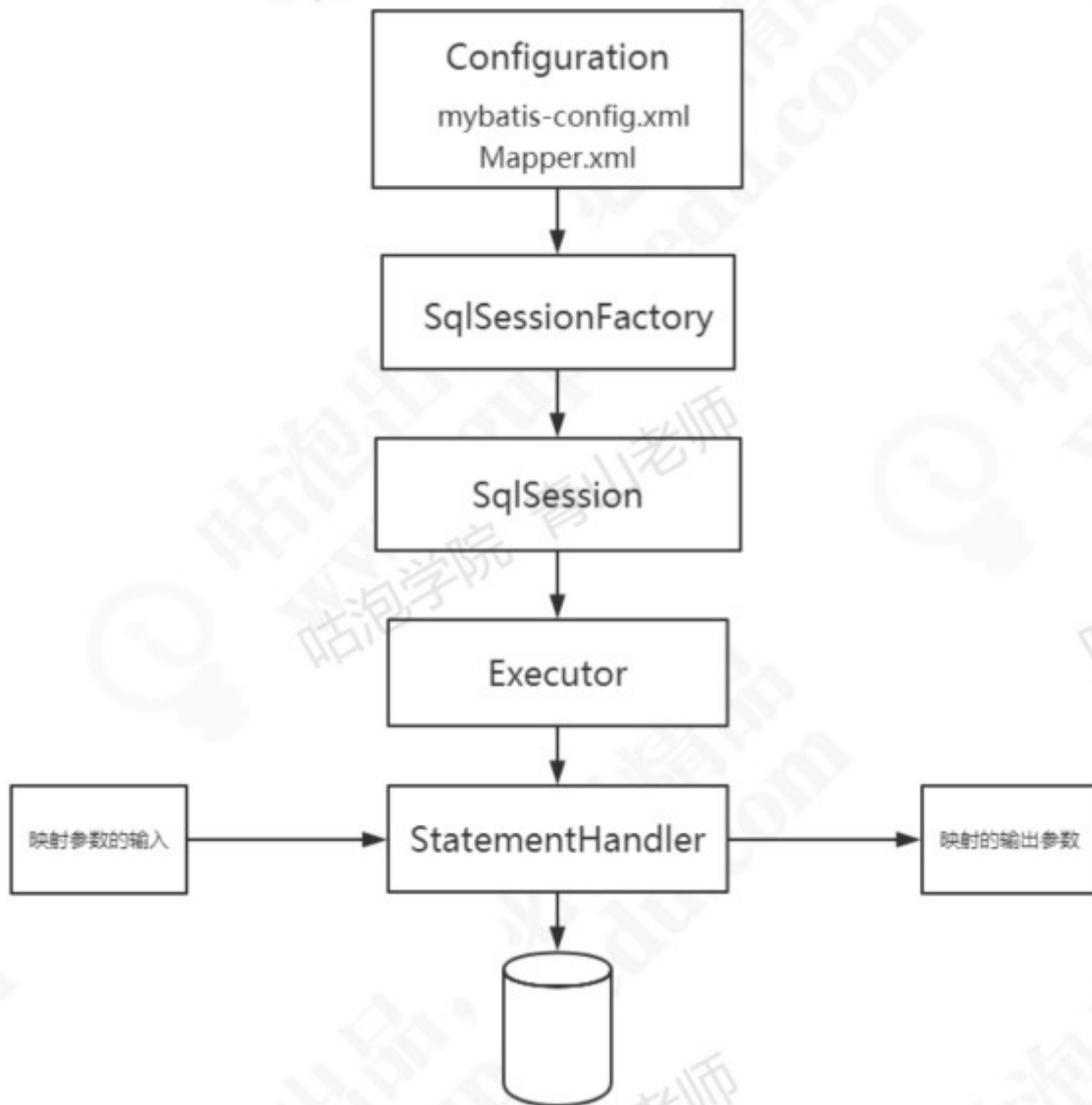
SQL的真正的执行对象。

1.3. 执行SQL操作

SqlSession 持有了一个Executor对象，用来封装对数据库的操作。

在执行器Executor 执行query或者update操作的时候我们创建一系列的对象，来处理参数、执行SQL、处理结果集，这里我们把它简化成一个对象：StatementHandler，可以把它理解为对Statement的封装，在阅读源码的时候我们再去了解还有什么其他的对象。

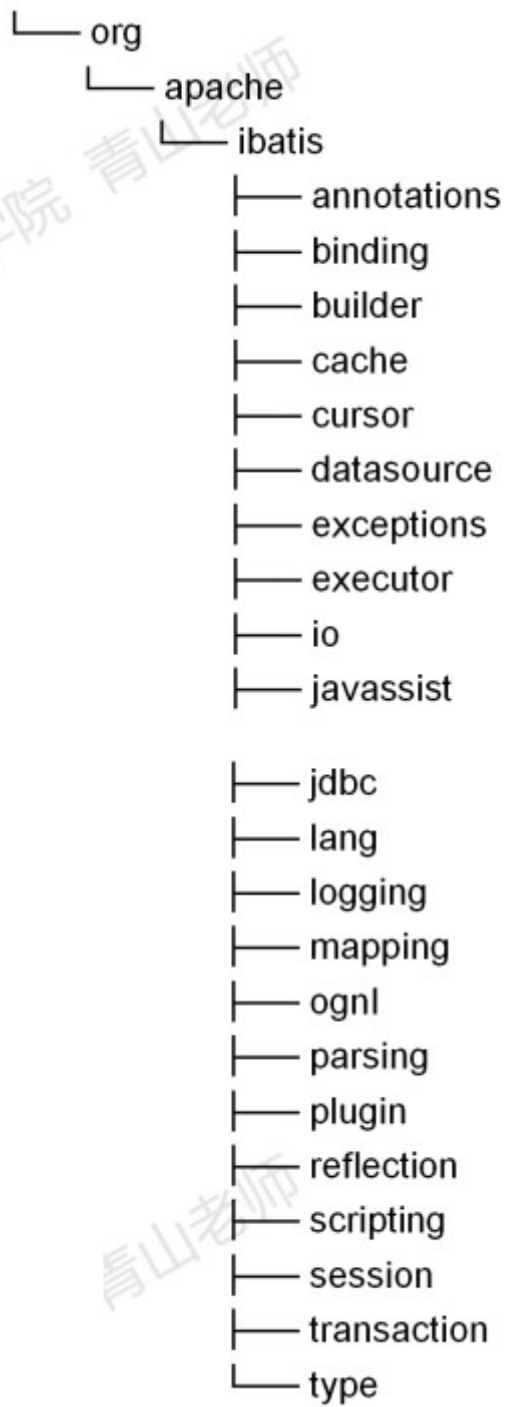
这个就是MyBatis主要的工作流程，如图：



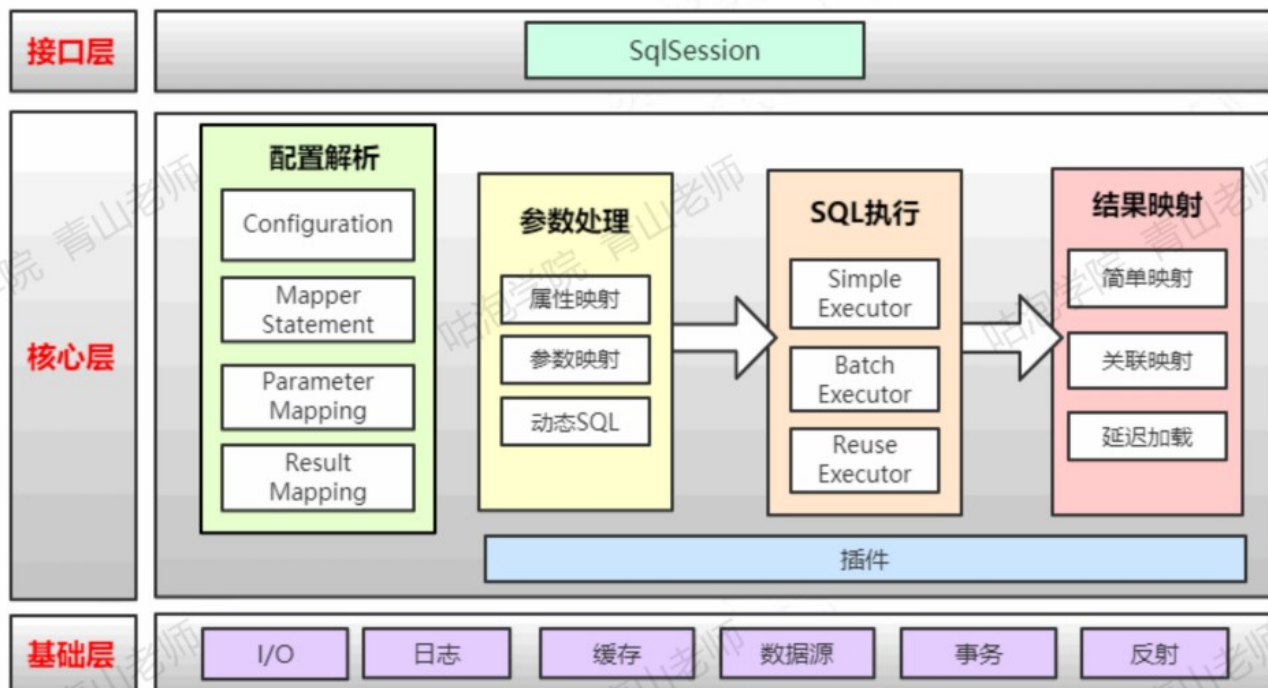
2. MyBatis架构分层与模块划分(总)

在MyBatis的主要工作流程里面，不同的功能是由很多不同的类协作完成的，它们分布在MyBatis jar包的不同的package里面。

MyBatis的jar包结构是这样的（21个包，基于3.5.4）：



按照功能职责的不同，所有的 package 可以分成不同的工作层次。



注：省略了基础层的其他模块

2.1. 接口层

首先接口层是我们打交道最多的。核心对象是 `SqlSession`，它是上层应用和 MyBatis 打交道的桥梁，`SqlSession` 上定义了对数据库的操作方法。接口层在接收到调用请求的时候，会调用核心处理层的相应模块来完成具体的数据库操作。

2.2. 核心处理层

接下来是核心处理层。既然叫核心处理层，也就是跟数据库操作相关的动作都是在这一层完成的。

核心处理层主要做了这几件事：

1. 把接口中传入的参数解析并且映射成 JDBC 类型；
2. 解析 xml 文件中的 SQL 语句，包括插入参数，和动态 SQL 的生成；
3. 执行 SQL 语句；
4. 处理结果集，并映射成 Java 对象。

插件也属于核心层，这是由它的工作方式和拦截的对象决定的。

2.3. 基础支持层

最后一个就是基础支持层。基础支持层主要是一些抽取出来的通用的功能（实现复用），用来支持核心处理层的功能。比如数据源、缓存、日志、xml 解析、反射、IO、事务等等这些功能。

3. MyBatis 缓存详解

3.1. cache 缓存

代码：mybatis-standalone工程，单元测试，cache

缓存是一般的ORM框架都会提供的功能，目的就是提升查询的效率和减少数据库的压力。跟Hibernate一样，MyBatis 也有一级缓存和二级缓存，并且预留了集成第三方缓存的接口。

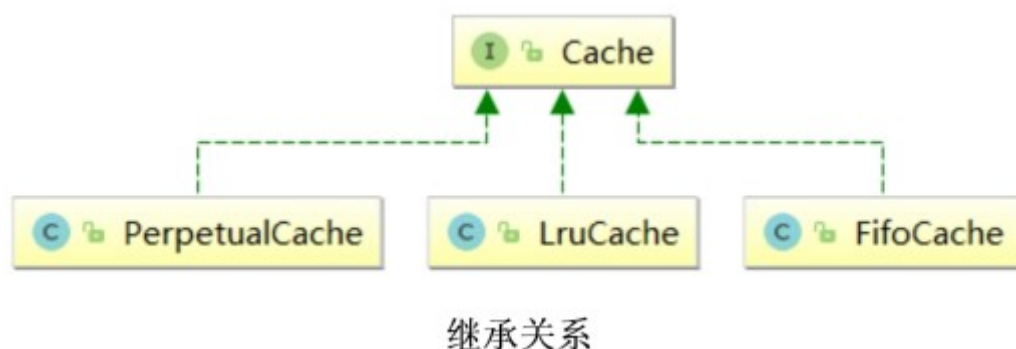
3.2. 缓存体系结构

MyBatis 跟缓存相关的类都在cache 包里面，其中有一个Cache接口，只有一个默认的实现类PerpetualCache，它是用HashMap实现的。

PerpetualCache 这个对象一定会创建，所以这个叫做基础缓存。但是缓存又可以有很多额外的功能，比如回收策略、日志记录、定时刷新等等，如果需要的话，就可以给基础缓存加上这些功能，如果不需要，就不加。

除了基础缓存之外，MyBatis 也定义了很多的装饰器，同样实现了Cache接口，通过这些装饰器可以额外实现很多的功能

装饰者模式（Decorator Pattern）是指在不改变原有对象的基础之上，将功能附加到对象上，提供了比继承更有弹性的替代方案（扩展原有对象的功能）。



debug源码的时候，有可能会看到基础缓存被装饰四五层，当然不管怎么装饰，经过多少层装饰，最后使用的还是基本的实现类（默认PerpetualCache）。

```
▼ delegate = {SynchronizedCache@1950}
  ▼ delegate = {LoggingCache@1973}
    > log = {StdOutImpl@1974}
  ▼ delegate = {SerializedCache@1975}
    ▼ delegate = {LruCache@1976}
      ▼ delegate = {PerpetualCache@1977}
        > id = "com.gupaoedu.mapper.BlogMapper"
        > cache = {HashMap@1980} size = 0
```

CachingExecutor 中 Debug 查看：缓存的层层装饰

所有的缓存实现类总体上可分为三类：基本缓存、淘汰算法缓存、装饰器缓存。

缓存实现类	描述	作用	装饰条件
基本缓存	缓存基本实现类	默认是 PerpetualCache，也可以自定义比如 RedisCache、EhCache 等，具备基本功能的缓存类	无
LruCache	LRU 策略的缓存	当缓存到达上限时候，删除最近最少使用的缓存（Least Recently Use）	eviction="LRU"（默认）
FifoCache	FIFO 策略的缓存	当缓存到达上限时候，删除最先入队的缓存	eviction="FIFO"
SoftCache WeakCache	带清理策略的缓存	通过 JVM 的软引用和弱引用来实现缓存，当 JVM 内存不足时，会自动清理掉这些缓存，基于 SoftReference 和 WeakReference	eviction="SOFT" eviction="WEAK"
LoggingCache	带日志功能的缓存	比如：输出缓存命中率	基本
SynchronizedCache	同步缓存	基于 synchronized 关键字实现，解决并发问题	基本
BlockingCache	阻塞缓存	通过在 get/put 方式中加锁，保证只有一个线程操作缓存，基于 Java 重入锁实现	blocking=true
SerializedCache	支持序列化的缓存	将对象序列化以后存到缓存中，取出时反序列化	readOnly=false（默认）
ScheduledCache	定时调度的缓存	在进行 get/put/remove/getSize 等操作前，判断缓存时间是否超过了设置的最长缓存时间（默认是一小时），如果是则清空缓存—即每隔一段时间清空一次缓存	flushInterval 不为空
TransactionalCache	事务缓存	在二级缓存中使用，可一次存入多个缓存，移除多个缓存	在 TransactionalCacheManager 中用 Map 维护对应关系

3.3. 一级缓存

3.3.1. 一级缓存(本地缓存) 介绍

一级缓存也叫本地缓存（Local Cache），MyBatis 的一级缓存是在会话（SqlSession）层面进行缓存的。MyBatis 的一级缓存是默认开启的，不需要任何的配置（localCacheScope 设置为 STATEMENT 关闭一级缓存）。

```

1. public abstract class BaseExecutor implements Executor {
2.     @Override
3.     public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
        rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws
        SQLException {
4.         ...
5.         if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
6.             // issue #482
7.             clearLocalCache();
8.         }

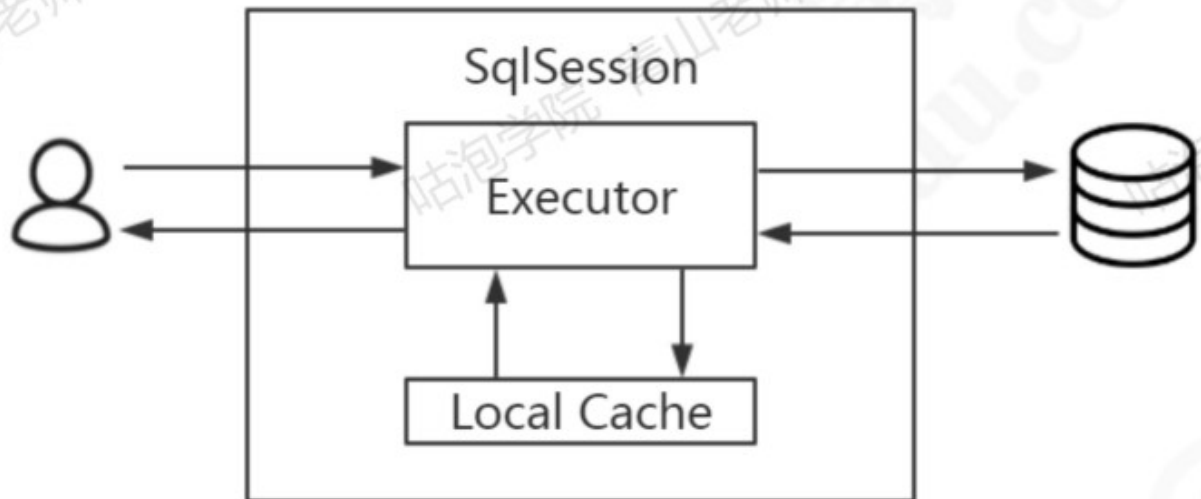
```

首先我们必须去弄清楚一个问题，在MyBatis执行的流程里面，涉及到这么多的对象，那么缓存 PerpetualCache 应该放在哪个对象里面去维护？

如果要在同一个会话里面共享一级缓存，最好的办法是在SqlSession里面创建的，作为SqlSession的一个属性，跟SqlSession 共存亡，这样就不需要为SqlSession编号、再根据SqlSession的编号去查找对应的缓存了。

DefaultSqlSession 里面只有两个对象属性：Configuration和Executor。

Configuration是全局的，不属于SqlSession，所以缓存只可能放在Executor里面维护——实际上它是在基本执行器SimpleExecutor/ReuseExecutor/BatchExecutor的父类BaseExecutor的构造函数中持有了PerpetualCache。在同一个会话里面，多次执行相同的SQL语句，会直接从内存取到缓存的结果，不会再发送SQL 到数据库。但是不同的会话里面，即使执行的SQL一模一样（通过一个Mapper的同一个方法的相同参数调用），也不能使用到一级缓存。



```
1. public abstract class BaseExecutor implements Executor {
2.     ...
3.     protected PerpetualCache localCache;
4.     ...
}
```

接下来我们来验证一下，MyBatis的一级缓存到底是不是只能在一个会话里面共享，以及跨会话（不同 session）操作相同的数据会产生什么问题。

3.3.2. 一级缓存验证

注意演示一级缓存需要先关闭二级缓存，localCacheScope设置为SESSION。

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

怎么判断是否命中缓存？如果再次发送SQL到数据库执行（控制台打印了SQL语句），说明没有命中缓存；如果直接打印对象，说明是从内存缓存中取到了结果。

单元测试类参见mybatis-standalone2-lesson工程cn.sitedev.cache.FirstLevelCacheTest

1、在同一个session中共享

```

1.     @Test
2.     public void testCache() throws IOException {
3.         String resource = "mybatis-config.xml";
4.         InputStream inputStream = Resources.getResourceAsStream(resource);
5.         SqlSessionFactory sqlSessionFactory = new
        SqlSessionFactoryBuilder().build(inputStream);
6.
7.         SqlSession session1 = sqlSessionFactory.openSession();
8.
9.         try {
10.             BlogMapper mapper0 = session1.getMapper(BlogMapper.class);
11.             BlogMapper mapper1 = session1.getMapper(BlogMapper.class);
12.             Blog blog = mapper0.selectBlogById(1);
13.             System.out.println(blog);
14.
15.             System.out.println("第二次查询，相同会话，获取到缓存了吗? ");
16.             System.out.println(mapper1.selectBlogById(1));
17.
18.         } finally {
19.             session1.close();
20.         }
21.     }

```

» ✓ Tests passed: 1 of 1 test - 1s 324ms

```

ms Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
ms Opening JDBC Connection
Created connection 439928219.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
==> Preparing: SELECT * FROM blog WHERE bid = ?
==> Parameters: 1(Integer)
<== Columns: bid, name, author_id
<== Row: 1, RabbitMQ延时消息, 1001
<== Total: 1
Blog(bid=1, name=RabbitMQ延时消息, authorId=1001)
第二次查询，相同会话，获取到缓存了吗？
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
Blog(bid=1, name=RabbitMQ延时消息, authorId=1001)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
Returned connection 439928219 to pool.

Process finished with exit code 0

```

2、不同session不能共享

```
1.     @Test
2.     public void testCache() throws IOException {
3.         String resource = "mybatis-config.xml";
4.         InputStream inputStream = Resources.getResourceAsStream(resource);
5.         SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
6.
7.         SqlSession session1 = sqlSessionFactory.openSession();
8.         SqlSession session2 = sqlSessionFactory.openSession();
9.         try {
10.             BlogMapper mapper0 = session1.getMapper(BlogMapper.class);
11.             BlogMapper mapper1 = session1.getMapper(BlogMapper.class);
12.             Blog blog = mapper0.selectBlogById(1);
13.             System.out.println(blog);
14.
15.             System.out.println("第二次查询，相同会话，获取到缓存了吗? ");
16.             System.out.println(mapper1.selectBlogById(1));
17.
18.             System.out.println("第三次查询，不同会话，获取到缓存了吗? ");
19.             BlogMapper mapper2 = session2.getMapper(BlogMapper.class);
20.             System.out.println(mapper2.selectBlogById(1));
21.
22.         } finally {
23.             session1.close();
24.         }
25.     }
```

```

PooledDataSource forcefully closed/removed all connections.
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
Opening JDBC Connection
Created connection 439928219.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
==> Preparing: SELECT * FROM blog WHERE bid = ?
==> Parameters: 1(Integer)
<==      Columns: bid, name, author_id
<==      Row: 1, RabbitMQ延时消息, 1001
<==      Total: 1
Blog(bid=1, name=RabbitMQ延时消息, authorId=1001)
第二次查询, 相同会话, 获取到缓存了吗?
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
Blog(bid=1, name=RabbitMQ延时消息, authorId=1001)
第三次查询, 不同会话, 获取到缓存了吗?
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
Opening JDBC Connection
Created connection 558569884.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@214b199c]
==> Preparing: SELECT * FROM blog WHERE bid = ?
==> Parameters: 1(Integer)
<==      Columns: bid, name, author_id
<==      Row: 1, RabbitMQ延时消息, 1001
<==      Total: 1
Blog(bid=1, name=RabbitMQ延时消息, authorId=1001)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
Returned connection 439928219 to pool.

Process finished with exit code 0

```

PS: 一级缓存在BaseExecutor的query () —queryFromDatabase()中存入。在queryFromDatabase之前会get ()。

```

1.    public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
    rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws
    SQLException {
2.        ...
3.        list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;
4.        if (list != null) {
5.            handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
6.        } else {
7.            list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key,
            boundSql);
8.        }
9.        ...

```

一级缓存在什么时候put, 什么时候 get, 什么时候 clear?

BaseExecutor 的 queryFromDatabase ()

```

1.    localCache.putObject(key, list);

```

一级缓存怎么命中? CacheKey怎么构成?

BaseExecutor的query ()

```
1. list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;
```

一级缓存什么时候会被清空呢?

3、同一个会话中, update (包括delete) 会导致一级缓存被清空

```
1. @Test
2. public void testCacheInvalid() throws IOException {
3.     String resource = "mybatis-config.xml";
4.     InputStream inputStream = Resources.getResourceAsStream(resource);
5.     SqlSessionFactory sqlSessionFactory = new
6.     SqlSessionFactoryBuilder().build(inputStream);
7.
8.     SqlSession session = sqlSessionFactory.openSession();
9.     try {
10.         BlogMapper mapper = session.getMapper(BlogMapper.class);
11.         System.out.println(mapper.selectBlogById(1));
12.
13.         Blog blog = new Blog();
14.         blog.setBid(1);
15.         blog.setName("after modified 666");
16.         mapper.updateByPrimaryKey(blog);
17.         session.commit();
18.
19.         // 相同会话执行了更新操作, 缓存是否被清空?
20.         System.out.println("在[同一个会话]执行更新操作之后, 是否命中缓存? ");
21.         System.out.println(mapper.selectBlogById(1));
22.     } finally {
23.         session.close();
24.     }
25. }
```

```

Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
==> Preparing: SELECT * FROM blog WHERE bid = ?
==> Parameters: 1(Integer)
<==      Columns: bid, name, author_id
<==      Row: 1, RabbitMQ延时消息, 1001
<==      Total: 1
Blog(bid=1, name=RabbitMQ延时消息, authorId=1001)
==> Preparing: update blog SET name = ? where bid = ?
==> Parameters: after modified 666(String), 1(Integer)
<==      Updates: 1
Committing JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
在[同一个会话]执行更新操作之后，是否命中缓存？
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
==> Preparing: SELECT * FROM blog WHERE bid = ?
==> Parameters: 1(Integer)
<==      Columns: bid, name, author_id
<==      Row: 1, after modified 666, 1001
<==      Total: 1
Blog(bid=1, name=after modified 666, authorId=1001)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
Returned connection 439928219 to pool.

```

问题：只有更新会清空缓存吗？查询会清空缓存吗？如果要清空呢？

一级缓存是在BaseExecutor中的update () 方法中调用clearLocalCache()清空的（无条件），如果是query会判断（只有 select 标签的flushCache=true才清空）。

一级缓存的工作范围是一个会话。如果跨会话，会出现什么问题？

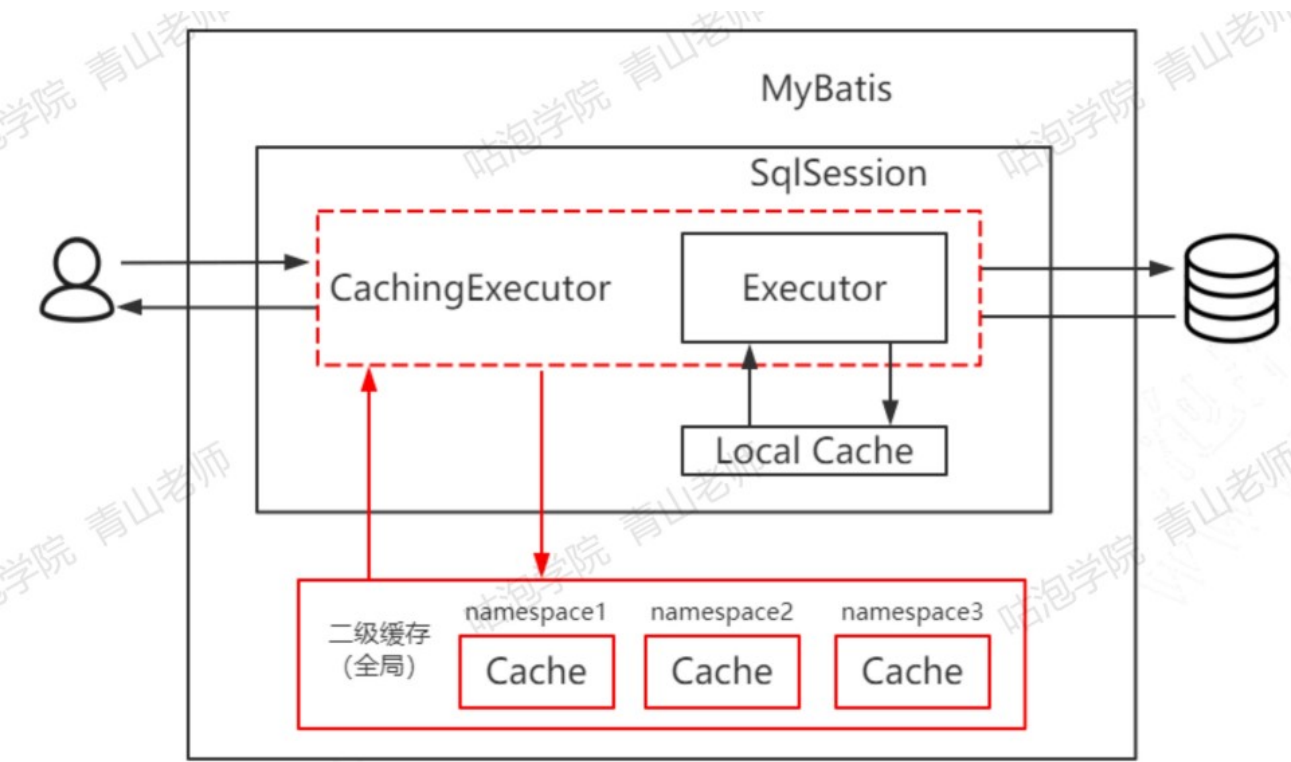
4、其他会话更新了数据，导致读取到过时的数据（一级缓存不能跨会话共享）

```
1.     @Test
2.     public void testDirtyRead() throws IOException {
3.         String resource = "mybatis-config.xml";
4.         InputStream inputStream = Resources.getResourceAsStream(resource);
5.         SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
6.
7.         SqlSession session1 = sqlSessionFactory.openSession();
8.         SqlSession session2 = sqlSessionFactory.openSession();
9.         try {
10.             BlogMapper mapper1 = session1.getMapper(BlogMapper.class);
11.             System.out.println(mapper1.selectBlogById(1));
12.
13.             // 会话2更新了数据，会话2的一级缓存更新
14.             Blog blog = new Blog();
15.             blog.setBid(1);
16.             blog.setName("after modified 3333333333333333");
17.             BlogMapper mapper2 = session2.getMapper(BlogMapper.class);
18.             mapper2.updateByPrimaryKey(blog);
19.             session2.commit();
20.
21.             // 其他会话更新了数据，本会话的一级缓存还在么？
22.             System.out.println("会话1查到最新的数据了吗？");
23.             System.out.println(mapper1.selectBlogById(1));
24.         } finally {
25.             session1.close();
26.             session2.close();
27.         }
28.     }
```


但是，二级缓存是不一定开启的。也就是说，开启了二级缓存，就启用这个对象，如果没有，就不用这个对象，我们应该怎么做呢？

实际上MyBatis用了一个装饰器的类来维护，就是CachingExecutor。如果启用了二级缓存，MyBatis 在创建Executor对象的时候会对Executor 进行装饰。

CachingExecutor 对于查询请求，会判断二级缓存是否有缓存结果，如果有就直接返回，如果没有委派交给真正的查询器Executor 实现类，比如 SimpleExecutor来执行查询，再走到一级缓存的流程。最后会把结果缓存起来，并且返回给用户。



我们知道，一级缓存是默认开启的，那二级缓存怎么开启呢？我们来看下二级缓存的开启方式。

3.4.2. 开启二级缓存的方法

第一步：在mybatis-config.xml中配置了（可以不配置，默认是true）：

```
1. <!-- 控制全局缓存（二级缓存），默认 true-->
2. <setting name="cacheEnabled" value="true"/>
```

只要没有显式地设置 `cacheEnabled=false`，都会用CachingExecutor装饰基本的执行器（SIMPLE、REUSE、BATCH）。

二级缓存的总开关是默认开启的。但是每个Mapper的二级缓存开关是默认关闭的。

一个Mapper要用到二级缓存，还要单独打开它自己的开关。

第二步：在Mapper.xml中配置 `<cache/>` 标签：

```

1.      <!--声明这个namespace使用二级缓存-->
2.      <!--size: 最多缓存对象个数，默认1024-->
3.      <!--eviction: 回收策略-->
4.      <!--flushInterval: 自动刷新时间，单位ms，未配置时只有调用时刷新-->
5.      <!--readOnly: 默认是false(安全)，改为true可读写时，对象必须支持序列化-->
6.      <cache type="org.apache.ibatis.cache.impl.PerpetualCache" size="1024"
    eviction="LRU" flushInterval="120000"
7.          readOnly="false"/>

```

cache 属性详解:

属性	含义	取值
type	缓存实现类	需要实现 Cache 接口，默认是 PerpetualCache，可以使用第三方缓存
size	最多缓存对象个数	默认 1024
eviction	回收策略 (缓存淘汰算法)	LRU - 最近最少使用的：移除最长时间不被使用的对象（默认）。 FIFO - 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。
flushInterval	定时自动清空缓存间隔	自动刷新时间，单位 ms，未配置时只有调用时刷新
readOnly	是否只读	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），不会共享。这会慢一些，但是安全，因此默认是 false。 改为 false 可读写时，对象必须支持序列化。
blocking	启用阻塞缓存	通过在 get/put 方式中加锁，保证只有一个线程操作缓存，基于 Java 重入锁实现

Mapper.xml配置了 `<cache>` 之后，select()会被缓存。update()、delete ()、insert () 会刷新缓存。

如果二级缓存拿到结果了，就直接返回（最外层的判断），否则再到一级缓存，最后到数据库。

如果cacheEnabled=true，Mapper.xml没有配置 `<cache>` 标签，还有二级缓存吗？还会出现CachingExecutor包装对象吗？

只要cacheEnabled=true 基本执行器就会被装饰。有没有配置 `<cache>`，决定了在启动的时候会不会创建这个mapper的Cache对象，最终会影响到CachingExecutor query 方法里面的判断：

```

1.      @Override
2.      public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds
    rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql)
3.          throws SQLException {
4.          Cache cache = ms.getCache();
5.          if (cache != null) {
6.              flushCacheIfRequired(ms);
7.              ...

```

也就是说，此时会装饰，但是没有 cache对象，依然不会走二级缓存流程。

如果一个Mapper 需要开启二级缓存，但是这个里面的某些查询方法对数据的实时性要求很高，不需

要二级缓存，怎么办？

我们可以在单个Statement ID上显式关闭二级缓存（默认是true）：

```
1. <select id="selectBlog" resultMap="BaseResultMap" useCache="false">
```

CachingExecutor query 方法有对这个属性的判断：

```
1. @Override
2. public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds
   rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql)
3.     throws SQLException {
4.     ...
5.     if (ms.isUseCache() && resultHandler == null) {
6.         ...
```

了解了二级缓存的工作位置和开启关闭的方法之后，我们也来验证一下二级缓存。

3.4.3. 二级缓存验证

注：验证二级缓存需要先开启二级缓存

3.4.3.1. 事务不提交，二级缓存不存在

```
1.    /**
2.     * 测试二级缓存一定要打开二级缓存开关
3.     *
4.     * @throws IOException
5.     */
6.    @Test
7.    public void testCache() throws IOException {
8.        String resource = "mybatis-config.xml";
9.        InputStream inputStream = Resources.getResourceAsStream(resource);
10.        SqlSessionFactory sqlSessionFactory = new
11.        SqlSessionFactoryBuilder().build(inputStream);
12.
13.        SqlSession session1 = sqlSessionFactory.openSession();
14.        SqlSession session2 = sqlSessionFactory.openSession();
15.        try {
16.            BlogMapper mapper1 = session1.getMapper(BlogMapper.class);
17.            System.out.println(mapper1.selectBlogById(1));
18.            // 事务不提交的情况下，二级缓存会写入吗？
19.            session1.commit();
20.
21.            System.out.println("第二次查询");
22.            BlogMapper mapper2 = session2.getMapper(BlogMapper.class);
23.            System.out.println(mapper2.selectBlogById(1));
24.        } finally {
25.            session1.close();
26.        }
```

✓ Tests passed: 1 of 1 test - 1s 326ms

```
Created connection 439928219.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
==> Preparing: SELECT * FROM blog WHERE bid = ?
==> Parameters: 1(Integer)
<==      Columns: bid, name, author_id
<==      Row: 1, after modified 3333333333333333, 1001
<==      Total: 1
Blog(bid=1, name=after modified 3333333333333333, authorId=1001)
第二次查询
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
Opening JDBC Connection
Created connection 558569884.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@214b199c]
==> Preparing: SELECT * FROM blog WHERE bid = ?
==> Parameters: 1(Integer)
<==      Columns: bid, name, author_id
<==      Row: 1, after modified 3333333333333333, 1001
<==      Total: 1
Blog(bid=1, name=after modified 3333333333333333, authorId=1001)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
Returned connection 439928219 to pool.

Process finished with exit code 0
```

思考：为什么事务不提交，二级缓存不生效？

因为二级缓存使用TransactionalCacheManager (TCM) 来管理，最后又调用了TransactionalCache的 getObject () 、putObject()和commit () 方法，TransactionalCache里面又持有了真正的Cache对象，比如是经过层层装饰的PerpetualCache。

```

1. public class TransactionalCacheManager {
2.
3.     private final Map<Cache, TransactionalCache> transactionalCaches = new HashMap<>
        ();
4.
5.     public void clear(Cache cache) {
6.         getTransactionalCache(cache).clear();
7.     }
8.
9.     public Object getObject(Cache cache, CacheKey key) {
10.        return getTransactionalCache(cache).getObject(key);
11.    }
12.
13.    public void putObject(Cache cache, CacheKey key, Object value) {
14.        getTransactionalCache(cache).putObject(key, value);
15.    }
16.
17.    public void commit() {
18.        for (TransactionalCache txCache : transactionalCaches.values()) {
19.            txCache.commit();
20.        }
21.    }
22.
23.    public void rollback() {
24.        for (TransactionalCache txCache : transactionalCaches.values()) {
25.            txCache.rollback();
26.        }
27.    }
28.
29.    private TransactionalCache getTransactionalCache(Cache cache) {
30.        return transactionalCaches.computeIfAbsent(cache, TransactionalCache::new);
31.    }
32.
33. }

```

在putObject的时候，只是添加到了entriesToAddOnCommit里面，只有它的commit（）方法被调用的时候才会调用flushPendingEntries（）真正写入缓存。它就是在DefaultSqlSession调用commit（）的时候被调用的。


```
1. public class TransactionalCache implements Cache {
2.     ...
3.     @Override
4.     public void putObject(Object key, Object object) {
5.         entriesToAddOnCommit.put(key, object);
6.     }
7.
8.     public void commit() {
9.         if (clearOnCommit) {
10.            delegate.clear();
11.        }
12.        flushPendingEntries();
13.        reset();
14.    }
15.
16.    private void flushPendingEntries() {
17.        for (Map.Entry<Object, Object> entry : entriesToAddOnCommit.entrySet()) {
18.            delegate.putObject(entry.getKey(), entry.getValue());
19.        }
20.        for (Object entry : entriesMissedInCache) {
21.            if (!entriesToAddOnCommit.containsKey(entry)) {
22.                delegate.putObject(entry, null);
23.            }
24.        }
25.    }
26.    ...
```

3.4.3.2. 使用不同的 session和mapper, 并且提交事务, 验证二级缓存可以跨 session存在

取消以上代码commit () 的注释, 测试。

```

1.      /**
2.       * 测试二级缓存一定要打开二级缓存开关
3.       *
4.       * @throws IOException
5.       */
6.      @Test
7.      public void testCache() throws IOException {
8.          String resource = "mybatis-config.xml";
9.          InputStream inputStream = Resources.getResourceAsStream(resource);
10.         SqlSessionFactory sqlSessionFactory = new
11.         SqlSessionFactoryBuilder().build(inputStream);
12.
13.         SqlSession session1 = sqlSessionFactory.openSession();
14.         SqlSession session2 = sqlSessionFactory.openSession();
15.         try {
16.             BlogMapper mapper1 = session1.getMapper(BlogMapper.class);
17.             System.out.println(mapper1.selectBlogById(1));
18.             // 事务不提交的情况下，二级缓存会写入吗？
19.             session1.commit();
20.
21.             System.out.println("第二次查询");
22.             BlogMapper mapper2 = session2.getMapper(BlogMapper.class);
23.             System.out.println(mapper2.selectBlogById(1));
24.         } finally {
25.             session1.close();
26.         }
27.     }

```

```

Opening JDBC Connection
Created connection 439928219.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
==> Preparing: SELECT * FROM blog WHERE bid = ?
==> Parameters: 1(Integer)
<==      Columns: bid, name, author_id
<==      Row: 1, after modified 3333333333333333, 1001
<==      Total: 1
Blog(bid=1, name=after modified 3333333333333333, authorId=1001)
第二次查询
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.5
Blog(bid=1, name=after modified 3333333333333333, authorId=1001)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
Returned connection 439928219 to pool.

Process finished with exit code 0

```

3.4.3.3. 在其他的 session 中执行增删改操作，验证缓存会被刷新

```
1.    /**
2.     * 测试二级缓存一定要打开二级缓存开关
3.     *
4.     * @throws IOException
5.     */
6.    @Test
7.    public void testCacheInvalid() throws IOException {
8.        String resource = "mybatis-config.xml";
9.        InputStream inputStream = Resources.getResourceAsStream(resource);
10.        SqlSessionFactory sqlSessionFactory = new
11.        SqlSessionFactoryBuilder().build(inputStream);
12.
13.        SqlSession session1 = sqlSessionFactory.openSession();
14.        SqlSession session2 = sqlSessionFactory.openSession();
15.        SqlSession session3 = sqlSessionFactory.openSession();
16.        try {
17.            BlogMapper mapper1 = session1.getMapper(BlogMapper.class);
18.            BlogMapper mapper2 = session2.getMapper(BlogMapper.class);
19.            BlogMapper mapper3 = session3.getMapper(BlogMapper.class);
20.            System.out.println(mapper1.selectBlogById(1));
21.            session1.commit();
22.
23.            // 是否命中二级缓存
24.            System.out.println("是否命中二级缓存? ");
25.            System.out.println(mapper2.selectBlogById(1));
26.
27.            Blog blog = new Blog();
28.            blog.setBid(1);
29.            blog.setName("2019年1月6日15:03:38");
30.            mapper3.updateByPrimaryKey(blog);
31.            session3.commit();
32.
33.            System.out.println("更新后再次查询，是否命中二级缓存? ");
34.            // 在其他会话中执行了更新操作，二级缓存是否被清空？
35.            System.out.println(mapper2.selectBlogById(1));
36.        } finally {
37.            session1.close();
38.            session2.close();
39.            session3.close();
40.        }
41.    }
```

```

Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
Opening JDBC Connection
Created connection 439928219.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
==> Preparing: SELECT * FROM blog WHERE bid = ?
==> Parameters: 1(Integer)
<==      Columns: bid, name, author_id
<==      Row: 1, after modified 333333333333333333, 1001
<==      Total: 1
Blog(bid=1, name=after modified 333333333333333333, authorId=1001)
是否命中二级缓存?
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.5
Blog(bid=1, name=after modified 333333333333333333, authorId=1001)
Opening JDBC Connection
Created connection 750468423.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@2cbb3d47]
==> Preparing: update blog SET name = ? where bid = ?
==> Parameters: 2019年1月6日15:03:38(String), 1(Integer)
<==      Updates: 1
Committing JDBC Connection [com.mysql.jdbc.JDBC4Connection@2cbb3d47]
更新后再次查询，是否命中二级缓存?
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.3333333333333333
Opening JDBC Connection
Created connection 1344199921.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@501edcf1]
==> Preparing: SELECT * FROM blog WHERE bid = ?
==> Parameters: 1(Integer)
<==      Columns: bid, name, author_id
<==      Row: 1, 2019年1月6日15:03:38, 1001
<==      Total: 1
Blog(bid=1, name=2019年1月6日15:03:38, authorId=1001)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a38c59b]
Returned connection 439928219 to pool

```

3.4.4. 为什么增删改操作会清空缓存?

在CachingExecutor的update () 方法里面会调用flushCacheIfRequired (ms) , isFlushCacheRequired 就是从标签里面取到的flushCache的值。而增删改操作的flushCache 属性默认为true。

```

1.      @Override
2.      public int update(MappedStatement ms, Object parameterObject) throws
        SQLException {
3.          flushCacheIfRequired(ms);
4.          return delegate.update(ms, parameterObject);
5.      }
6.
7.      private void flushCacheIfRequired(MappedStatement ms) {
8.          Cache cache = ms.getCache();
9.          if (cache != null && ms.isFlushCacheRequired()) {
10.             tcm.clear(cache);
11.          }
12.      }

```

也就是说，如果不需要清空二级缓存，可以把flushCache 属性修改为false（这样会造成过时数据的问题）。

3.4.5. 什么时候开启二级缓存？

一级缓存默认是打开的，二级缓存需要配置才可以开启。那么我们必须思考一个问题，在什么情况下才有必要去开启二级缓存？

- 1、因为所有的增删改都会刷新二级缓存，导致二级缓存失效，所以适合在查询为主的应用中使用，比如历史交易、历史订单的查询。否则缓存就失去了意义。
- 2、如果多个namespace中有针对于同一个表的操作，比如Blog表，如果在一个namespace 中刷新了缓存，另一个namespace中没有刷新，就会出现读到脏数据的情况。

所以，推荐在一个Mapper里面只操作单表的情况使用。

如果要让多个namespace共享一个二级缓存，应该怎么做？

跨namespace的缓存共享的问题，可以使用 `<cache-ref>` 来解决：

```
1. <cache-ref namespace="cn.sitedev.crud.dao.DepartmentMapper"/>
```

cache-ref代表引用别的命名空间的Cache配置，两个命名空间的操作使用的是同一个Cache。在关联的表比较少，或者按照业务可以对表进行分组的时候可以使用。

注意：在这种情况下，多个Mapper的操作都会引起缓存刷新，缓存的意义已经不大了。

3.4.6. 第三方缓存做二级缓存

除了MyBatis 自带的二级缓存之外，我们也可以通过实现Cache 接口来自定义二级缓存。

MyBatis 官方提供了一些第三方缓存集成方式，比如ehcache 和redis：

<https://github.com/mybatis/redis-cache>

pom文件引入依赖：

```
1. <dependency>
2.     <groupId>org.mybatis.caches</groupId>
3.     <artifactId>mybatis-redis</artifactId>
4.     <version>1.0.0-beta2</version>
5. </dependency>
```

Mapper.xml配置，type 使用RedisCache：

```
1. <!--使用Redis作为二级缓存-->
2. <cache type="org.mybatis.caches.redis.RedisCache" eviction="FIFO"
    flushInterval="60000" size="512" readOnly="true"/>
```

redis.properties配置：

1. host=localhost
2. port=6379
3. connectionTimeout=5000
4. soTimeout=5000
5. database=0

Redis作为二级缓存的验证（需要安装Redis 桌面客户端, 这里使用的是Redis Desktop Manager）：



当然，在分布式环境中，我们也可以使用独立的缓存服务，不使用MyBatis自带的二级缓存。

4. MyBatis源码解读

4.1. 带着问题看源码

分析源码，我们还是从编程式的demo入手。

```

1.  public class MyBatisTest {
2.
3.      private SqlSessionFactory sqlSessionFactory;
4.
5.      @Before
6.      public void prepare() throws IOException {
7.          String resource = "mybatis-config.xml";
8.          InputStream inputStream = Resources.getResourceAsStream(resource);
9.          sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
10.     }
11.
12.     /**
13.      * 通过 SqlSession.getMapper(XXXMapper.class) 接口方式
14.      *
15.      * @throws IOException
16.      */
17.     @Test
18.     public void testSelect() throws IOException {
19.         SqlSession session = sqlSessionFactory.openSession(); //
ExecutorType.BATCH
20.         try {
21.             BlogMapper mapper = session.getMapper(BlogMapper.class);
22.             Blog blog = mapper.selectBlogById(1);
23.             System.out.println(blog);
24.         } finally {
25.             session.close();
26.         }
27.     }
28.     ...

```

把文件读取成流的这一步我们就省略了。所以下面我们分成五步来分析。

第一步，我们创建一个工厂类，配置文件的解析就是在这一步完成的，包括mybatis-config.xml和Mapper映射器文件。

这一步我们关心的内容：解析的时候做了什么，产生了什么对象，解析的结果存放到了哪里。解析的结果决定着后面有什么对象可以使用，和到哪里去取。

第二步，通过SqlSessionFactory 创建一个SqlSession。

问题：SqlSession 上面定义了各种增删改查的API，是给客户端调用的。返回了什么实现类？除了SqlSession，还创建了什么对象，创建了什么环境？

第三步，获得一个Mapper对象。

问题：Mapper是一个接口，没有实现类，是不能被实例化的，那获取到的这个Mapper 对象是什么对象？为什么要从SqlSession里面去获取？为什么传进去一个接口，然后还要用接口类型来接收？

第四步，调用接口方法。

问题：我们的接口没有创建实现类，为什么可以调用它的方法？那它调用的是什么方法？

这一步实际做的事情是执行SQL，那它又是根据什么找到XML映射器里面的SQL的？此外，我们的方法参数（对象或者Map）是怎么转换成SQL参数的？获取到的结果集是怎么转换成对象的？

最后一步，关闭 session，这一步是必须要做的。

下面我们会按照这五个步骤，去理解MyBatis的运行原理，这里面会涉及到很多核心的对象和关键的方法。

4.2. 看源码的注意事项

- 1、一定要带着问题去看，猜想验证。
- 2、不要只记忆流程，学编程风格，设计思想（他的代码为什么这么写？如果不这么写呢？包括接口的定义，类的职责，涉及模式的应用，高级语法等等）。
- 3、先抓重点，就像开车熟路，哪个地方限速，哪个地方变道，要走很多次才会熟练。先走主干道，再去覆盖分支小路。
- 4、记录核心流程和对象，总结层次、结构、关系，输出（图片或者待注释的源码）。
- 5、培养看源码的信心和感觉，从带着看到自己去看，看更多的源码。
- 6、debug 还是直接Ctrl+Alt+B跟方法？

debug 可以看到实际的值，比如到底是哪个实现类，value到底是什么。

4.3. 一. 配置解析过程

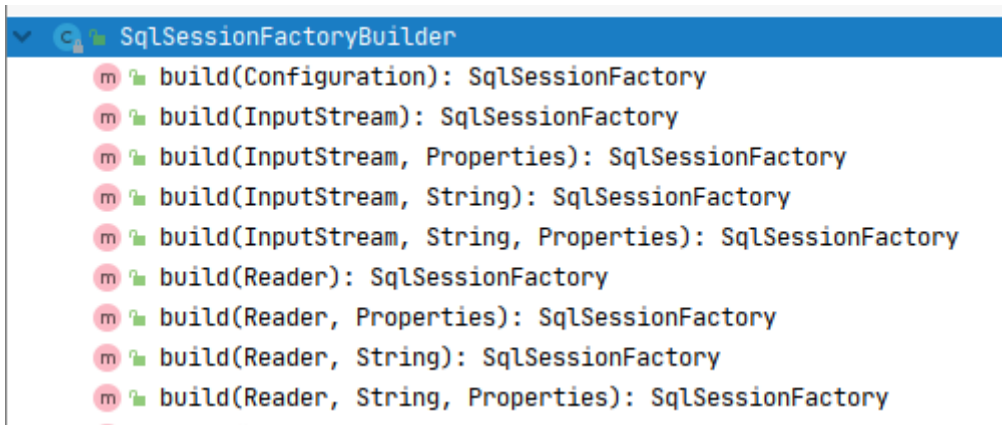
首先我们要清楚的是配置解析的过程全部只解析了两种文件。一个是mybatis-config.xml 全局配置文件。另外就是所有的Mapper.xml文件，也包括在Mapper 接口类上面定义的注解。

我们从mybatis-config.xml开始。在第一节课的时候我们已经分析了核心配置了，大概明白了MyBatis 有哪些配置项，和这些配置项的大致含义。这里我们再具体看一下这里面的标签都是怎么解析的，解析的时候做了什么。

```
1.  SqlSessionFactory sqlSessionFactory = new
    SqlSessionFactoryBuilder().build(inputStream);
```

首先我们new了一个SqlSessionFactoryBuilder，这是建造者模式的运用（建造者模式用来创建复杂对象，而不需要关注内部细节，是一种封装的体现）。MyBatis中很多地方用到了建造者模式（名字以Builder结尾的类还有9个）。

SqlSessionFactoryBuilder 中用来创建SqlSessionFactory对象的方法是build()，build()方法有9个重载，可以用不同的方式来创建 SqlSessionFactory对象。



SqlSessionFactory 对象默认是单例的。

4.3.1. XMLConfigBuilder

这里面创建了一个XMLConfigBuilder对象（用来存放所有配置信息的Configuration 对象也是这个时候创建的）。

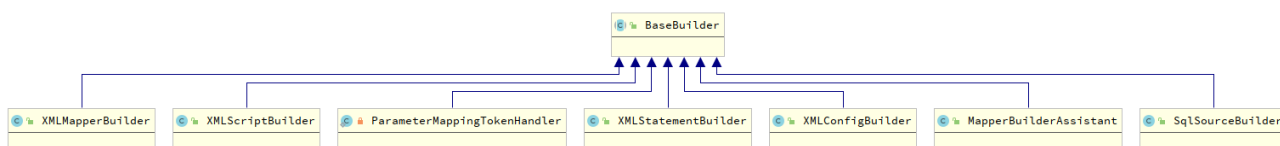
```
1. XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);
```

XMLConfigBuilder 是抽象类BaseBuilder的一个子类，专门用来解析全局配置文件，针对不同的构建目标还有其他的一些子类（关联到源码路径），比如：

XMLMapperBuilder：解析Mapper映射器

XMLStatementBuilder：解析增删改查标签

XMLScriptBuilder：解析动态SQL



根据我们解析的文件流，这里后面两个参数都是空的，创建了一个parser。

```
1. return build(parser.parse());
```

这里有两步，第一步是调用parser的parse()方法，它会返回一个Configuration类。

之前我们说过，也就是配置文件里面所有的信息都会放在Configuration里面。 `<configuration>` 的子标签跟Configuration类的属性是直接对应的。

我们先看一下parse()方法：

```

1. public Configuration parse() {
2.     if (parsed) {
3.         throw new BuilderException("Each XMLConfigBuilder can only be used once.");
4.     }
5.     parsed = true;
6.     parseConfiguration(parser.evalNode("/configuration"));
7.     return configuration;
8. }

```

首先会检查全局配置文件是不是已经解析过，也就是说在应用的生命周期里面，config 配置文件只需要解析一次，生成的Configuration对象也会存在应用的整个生命周期中。

接下来就是parseConfiguration方法：

```

1. parseConfiguration(parser.evalNode("/configuration"));

```

解析XML有很多方法，MyBatis对dom和SAX做了封装，方便使用。

这下面有十几个方法，对应着config文件里面的所有一级标签。

```

1. private void parseConfiguration(XNode root) {
2.     try {
3.         //issue #117 read properties first
4.         propertiesElement(root.evalNode("properties"));
5.         Properties settings = settingsAsProperties(root.evalNode("settings"));
6.         loadCustomVfs(settings);
7.         loadCustomLogImpl(settings);
8.         typeAliasesElement(root.evalNode("typeAliases"));
9.         pluginElement(root.evalNode("plugins"));
10.        objectFactoryElement(root.evalNode("objectFactory"));
11.        objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
12.        reflectorFactoryElement(root.evalNode("reflectorFactory"));
13.        settingsElement(settings);
14.        // read it after objectFactory and objectWrapperFactory issue #631
15.        environmentsElement(root.evalNode("environments"));
16.        databaseIdProviderElement(root.evalNode("databaseIdProvider"));
17.        typeHandlerElement(root.evalNode("typeHandlers"));
18.        mapperElement(root.evalNode("mappers"));
19.    } catch (Exception e) {
20.        throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: "
21.            + e, e);
22.    }
}

```

问题：MyBatis 全局配置文件中标签的顺序可以颠倒吗？比如把settings放在plugin 之后？会报错。所以顺序必须严格一致。

```
### Cause: org.apache.ibatis.builder.BuilderException: Error creating document instance. Cause:
org.xml.sax.SAXParseException; lineNumber: 65; columnNumber: 17; 元素类型为 "configuration" 的内容必须匹配
"(properties?,settings?,typeAliases?,typeHandlers?,objectFactory?,objectWrapperFactory?,reflectorFactory?,plugins?,envi
ronments?,databaseIdProvider?,mappers?)"。
```

4.3.2. propertiesElement()

第一个是解析 `<properties>` 标签，读取我们引入的外部配置文件，例如db.properties。

```
1.     private void propertiesElement(XNode context) throws Exception {
2.         if (context != null) {
3.             Properties defaults = context.getChildrenAsProperties();
4.             String resource = context.getStringAttribute("resource");
5.             String url = context.getStringAttribute("url");
6.             if (resource != null && url != null) {
7.                 throw new BuilderException("The properties element cannot specify both a
URL and a resource based property file reference. Please specify one or the
other.");
8.             }
9.             if (resource != null) {
10.                defaults.putAll(Resources.getResourceAsProperties(resource));
11.            } else if (url != null) {
12.                defaults.putAll(Resources.getUrlAsProperties(url));
13.            }
14.            Properties vars = configuration.getVariables();
15.            if (vars != null) {
16.                defaults.putAll(vars);
17.            }
18.            parser.setVariables(defaults);
19.            configuration.setVariables(defaults);
20.        }
21.    }
```

这里面又有两种类型，一种是放在resource目录下的，是相对路径，一种是写的绝对路径的(url)。

解析的最终结果就是我们会把所有的配置信息放到名为defaults的Properties对象里面（Hashtable对象，KV存储），最后把XPathParser和Configuration的Properties属性都设置成我们填充后的Properties对象。

```
1.         parser.setVariables(defaults);
2.         configuration.setVariables(defaults);
```

4.3.3. settingsAsProperties()

第二个，我们把 `<settings>` 标签也解析成了一个Properties对象，对于 `<settings>` 标签的子标签的处理在后面（先解析，后设置）。

```
1. Properties settings = settingsAsProperties(root.evalNode("settings"));
```

```
1. private Properties settingsAsProperties(XNode context) {
2.     if (context == null) {
3.         return new Properties();
4.     }
5.     Properties props = context.getChildrenAsProperties();
6.     // Check that all settings are known to the configuration class
7.     MetaClass metaConfig = MetaClass.forClass(Configuration.class,
localReflectorFactory);
8.     for (Object key : props.keySet()) {
9.         if (!metaConfig.hasSetter(String.valueOf(key))) {
10.            throw new BuilderException("The setting " + key + " is not known. Make
sure you spelled it correctly (case sensitive).");
11.        }
12.    }
13.    return props;
14. }
```

在早期的版本里面解析和设置都是在后面一起的，这里先解析成Properties对象是因为下面的两个方法要用到。

4.3.4. loadCustomVfs (settings)

loadCustomVfs 是获取Virtual File System的自定义实现类，比如要读取本地文件，或者FTP远程文件的时候，就可以用到自定义的VFS类。

根据 `<settings>` 标签里面的 `<vfsImpl>` 标签，生成了一个抽象类VFS的子类，在MyBatis中有JBoss6VFS和DefaultVFS两个实现，在io包中。

```
1. private void loadCustomVfs(Properties props) throws ClassNotFoundException {
2.     String value = props.getProperty("vfsImpl");
3.     if (value != null) {
4.         String[] clazzes = value.split(",");
5.         for (String clazz : clazzes) {
6.             if (!clazz.isEmpty()) {
7.                 @SuppressWarnings("unchecked")
8.                 Class<? extends VFS> vfsImpl = (Class<? extends
VFS>)Resources.class.forName(clazz);
9.                 configuration.setVfsImpl(vfsImpl);
10.            }
11.        }
12.    }
13. }
```

最后赋值到Configuration中。

4.3.5. loadCustomLogImpl (settings)

loadCustomLogImpl是根据 `<logImpl>` 标签获取日志的实现类，我们可以用到很多的日志的方案，包括LOG4J, LOG4J2, SLF4J等等，在logging包中。

```
1. private void loadCustomLogImpl(Properties props) {
2.     Class<? extends Log> logImpl = resolveClass(props.getProperty("logImpl"));
3.     configuration.setLogImpl(logImpl);
4. }
```

这里生成了一个Log 接口的实现类，并且赋值到Configuration中。

4.3.6. typeAliasesElement()

这一步解析的是类型别名。

```
1. private void typeAliasesElement(XNode parent) {
2.     if (parent != null) {
3.         for (XNode child : parent.getChildren()) {
4.             if ("package".equals(child.getName())) {
5.                 String typeAliasPackage = child.getStringAttribute("name");
6.                 configuration.getTypeAliasRegistry().registerAliases(typeAliasPackage);
7.             } else {
8.                 String alias = child.getStringAttribute("alias");
9.                 String type = child.getStringAttribute("type");
10.                try {
11.                    Class<?> clazz = Resources.classForName(type);
12.                    if (alias == null) {
13.                        typeAliasRegistry.registerAlias(clazz);
14.                    } else {
15.                        typeAliasRegistry.registerAlias(alias, clazz);
16.                    }
17.                } catch (ClassNotFoundException e) {
18.                    throw new BuilderException("Error registering typeAlias for '" + alias
19. + "'. Cause: " + e, e);
20.                }
21.            }
22.        }
23.    }
```

我们在讲配置的时候也讲过，它有两种定义方式，一种是直接定义一个类的别名（例如cn.sitedev.domain.Blog定义成blog），一种就是指定一个package，那么这个包下面所有的类的名字就会成为这个类全路径的别名。

类的别名和类的关系，我们放在一个TypeAliasRegistry对象里面。

```
1. typeAliasRegistry.registerAlias(alias, clazz);
```

大家也可以推测一下，如果要保存这种类名（String）和类（Class）的对应关系，TypeAliasRegistry 应该是一个什么样的数据结构。

```
1. public class TypeAliasRegistry {
2.
3.     private final Map<String, Class<?>> typeAliases = new HashMap<>();
4.     ...
}
```

4.3.7. pluginElement()

接下来就是解析 `<plugins>` 标签，比如Pagehelper的翻页插件，或者我们自定义的插件。

`<plugins>` 标签里面只有 `<plugin>` 标签，`<plugin>` 标签里面只有 `<property>` 标签。

```
1. private void pluginElement(XNode parent) throws Exception {
2.     if (parent != null) {
3.         for (XNode child : parent.getChildren()) {
4.             String interceptor = child.getStringAttribute("interceptor");
5.             Properties properties = child.getChildrenAsProperties();
6.             Interceptor interceptorInstance = (Interceptor)
7. resolveClass(interceptor).getDeclaredConstructor().newInstance();
8.             interceptorInstance.setProperties(properties);
9.             configuration.addInterceptor(interceptorInstance);
10.        }
11.    }
}
```

因为所有的插件都要实现Interceptor接口，所以这一步做的事情就是把插件解析成Interceptor类，设置属性，然后添加到Configuration的InterceptorChain 属性里面，它是一个List。

```
1. String interceptor = child.getStringAttribute("interceptor");
2. Properties properties = child.getChildrenAsProperties();
3. Interceptor interceptorInstance = (Interceptor)
4. resolveClass(interceptor).getDeclaredConstructor().newInstance();
5. interceptorInstance.setProperties(properties);
6. configuration.addInterceptor(interceptorInstance);
```

注意，插件的工作流程分成三步，第一步解析，第二步包装（代理），第三步运行时拦截。这里完成了第一步的工作。

4.3.8. objectFactoryElement()、objectWrapperFactoryElement()、reflectorFactoryElement()

ObjectFactory用来创建返回的对象。


```

1.     private void objectFactoryElement(XNode context) throws Exception {
2.         if (context != null) {
3.             String type = context.getStringAttribute("type");
4.             Properties properties = context.getChildrenAsProperties();
5.             ObjectFactory factory = (ObjectFactory)
        resolveClass(type).getDeclaredConstructor().newInstance();
6.             factory.setProperties(properties);
7.             configuration.setObjectFactory(factory);
8.         }
9.     }

```

ObjectWrapperFactory 用来对对象做特殊的处理。比如：select没有写别名，查询返回的是一个Map，可以在自定义的objectWrapperFactory中把下划线命名变成驼峰命名。

```

1.     private void objectWrapperFactoryElement(XNode context) throws Exception {
2.         if (context != null) {
3.             String type = context.getStringAttribute("type");
4.             ObjectWrapperFactory factory = (ObjectWrapperFactory)
        resolveClass(type).getDeclaredConstructor().newInstance();
5.             configuration.setObjectWrapperFactory(factory);
6.         }
7.     }

```

ReflectorFactory是反射的工具箱，对反射的操作进行了封装（官网和文档没有这个对象的描述）。

```

1.     private void reflectorFactoryElement(XNode context) throws Exception {
2.         if (context != null) {
3.             String type = context.getStringAttribute("type");
4.             ReflectorFactory factory = (ReflectorFactory)
        resolveClass(type).getDeclaredConstructor().newInstance();
5.             configuration.setReflectorFactory(factory);
6.         }
7.     }

```

以上四个对象，都是用resolveClass创建的。

```

1.     Interceptor interceptorInstance = (Interceptor)
        resolveClass(interceptor).getDeclaredConstructor().newInstance();
2.     ///////////////////////////////////////////////////
3.     ObjectFactory factory = (ObjectFactory)
        resolveClass(type).getDeclaredConstructor().newInstance();
4.     ///////////////////////////////////////////////////
5.     ObjectWrapperFactory factory = (ObjectWrapperFactory)
        resolveClass(type).getDeclaredConstructor().newInstance();
6.     ///////////////////////////////////////////////////
7.     ReflectorFactory factory = (ReflectorFactory)
        resolveClass(type).getDeclaredConstructor().newInstance();

```

4.3.9. settingsElement (settings)

这里就是对 `<settings>` 标签里面所有子标签的处理了，前面我们已经把子标签全部转换成了 Properties 对象，所以在这里处理 Properties 对象就可以了。

settings 二级标签中一共26个配置，比如二级缓存、延迟加载、默认执行器类型等等。

需要注意的是，我们之前提到的所有的默认值，都是在这里赋值的。如果说后面我们不知道这个属性的值是什么，也可以到这一步来确认一下。

所有的值，都会赋值到Configuration的属性里面去。

```
1.     private void settingsElement(Properties props) {
2.
3.         configuration.setAutoMappingBehavior(AutoMappingBehavior.valueOf(props.getProperty(
4.             "autoMappingBehavior", "PARTIAL")));
5.
6.         configuration.setAutoMappingUnknownColumnBehavior(AutoMappingUnknownColumnBehavior
7.             .valueOf(props.getProperty("autoMappingUnknownColumnBehavior", "NONE")));
8.
9.         configuration.setCacheEnabled(booleanValueOf(props.getProperty("cacheEnabled"),
10.             true));
11.         configuration.setProxyFactory((ProxyFactory)
12.             createInstance(props.getProperty("proxyFactory")));
13.
14.         configuration.setLazyLoadingEnabled(booleanValueOf(props.getProperty("lazyLoadingE
15.             nabled"), false));
16.
17.         configuration.setAggressiveLazyLoading(booleanValueOf(props.getProperty("aggressiv
18.             eLazyLoading"), false));
19.
20.         configuration.setMultipleResultSetsEnabled(booleanValueOf(props.getProperty("multi
21.             pleResultSetsEnabled"), true));
22.
23.         configuration.setUseColumnLabel(booleanValueOf(props.getProperty("useColumnLabel")
24.             , true));
25.
26.         configuration.setUseGeneratedKeys(booleanValueOf(props.getProperty("useGeneratedKe
27.             ys"), false));
28.
29.         configuration.setDefaultExecutorType(ExecutorType.valueOf(props.getProperty("defau
30.             ltExecutorType", "SIMPLE")));
31.
32.         configuration.setDefaultStatementTimeout(integerValueOf(props.getProperty("default
33.             StatementTimeout"), null));
34.
35.         configuration.setDefaultFetchSize(integerValueOf(props.getProperty("defaultFetchSi
36.             ze"), null));
37.
38.         configuration.setDefaultResultSetType(resolveResultSetType(props.getProperty("defa
39.             ultResultSetType")));
40.
41.         configuration.setMapUnderscoreToCamelCase(booleanValueOf(props.getProperty("mapUnd
42.             erscoreToCamelCase"), false));
43.
44.         configuration.setSafeRowBoundsEnabled(booleanValueOf(props.getProperty("safeRowBou
45.             ndsEnabled"), false));
46.
47.         configuration.setLocalCacheScope(LocalCacheScope.valueOf(props.getProperty("localC
48.             acheScope", "SESSION")));
```

```

18. configuration.setJdbcTypeForNull(JdbcType.valueOf(props.getProperty("jdbcTypeForNull", "OTHER")));
19. configuration.setLazyLoadTriggerMethods(stringSetValueOf(props.getProperty("lazyLoadTriggerMethods"), "equals,clone,hashCode,toString"));
20. configuration.setSafeResultHandlerEnabled(booleanValueOf(props.getProperty("safeResultHandlerEnabled"), true));
21. configuration.setDefaultScriptingLanguage(resolveClass(props.getProperty("defaultScriptingLanguage")));
22. configuration.setDefaultEnumTypeHandler(resolveClass(props.getProperty("defaultEnumTypeHandler")));
23. configuration.setCallSettersOnNulls(booleanValueOf(props.getProperty("callSettersOnNulls"), false));
24. configuration.setUseActualParamName(booleanValueOf(props.getProperty("useActualParamName"), true));
25. configuration.setReturnInstanceForEmptyRow(booleanValueOf(props.getProperty("returnInstanceForEmptyRow"), false));
26.     configuration.setLogPrefix(props.getProperty("logPrefix"));
27. configuration.setConfigurationFactory(resolveClass(props.getProperty("configurationFactory")));
28.     }

```

4.3.10. environmentsElement()

这一步是解析 `<environments>` 标签。

我们前面讲过，一个environment 就是对应一个数据源，所以在这里我们会根据配置的 `<transactionManager>` 创建一个事务工厂，根据 `<dataSource>` 标签创建一个数据源，最后把这两个对象设置成Environment对象的属性，放到Configuration里面。

```

1.     private void environmentsElement(XNode context) throws Exception {
2.         if (context != null) {
3.             if (environment == null) {
4.                 environment = context.getStringAttribute("default");
5.             }
6.             for (XNode child : context.getChildren()) {
7.                 String id = child.getStringAttribute("id");
8.                 if (isSpecifiedEnvironment(id)) {
9.                     TransactionFactory txFactory =
transactionManagerElement(child.evalNode("transactionManager"));
10.                    DataSourceFactory dsFactory =
dataSourceElement(child.evalNode("dataSource"));
11.                    DataSource dataSource = dsFactory.getDataSource();
12.                    Environment.Builder environmentBuilder = new Environment.Builder(id)
13.                        .transactionFactory(txFactory)
14.                        .dataSource(dataSource);
15.                    configuration.setEnvironment(environmentBuilder.build());
16.                }
17.            }
18.        }
19.    }

```

4.3.11. databaseIdProviderElement()

解析 databaseIdProvider 标签，生成 DatabaseIdProvider 对象（用来支持不同厂商的数据库）。

```

1.     private void databaseIdProviderElement(XNode context) throws Exception {
2.         DatabaseIdProvider databaseIdProvider = null;
3.         if (context != null) {
4.             String type = context.getStringAttribute("type");
5.             // awful patch to keep backward compatibility
6.             if ("VENDOR".equals(type)) {
7.                 type = "DB_VENDOR";
8.             }
9.             Properties properties = context.getChildrenAsProperties();
10.            databaseIdProvider = (DatabaseIdProvider)
resolveClass(type).getDeclaredConstructor().newInstance();
11.            databaseIdProvider.setProperties(properties);
12.        }
13.        Environment environment = configuration.getEnvironment();
14.        if (environment != null && databaseIdProvider != null) {
15.            String databaseId =
databaseIdProvider.getDatabaseId(environment.getDataSource());
16.            configuration.setDatabaseId(databaseId);
17.        }
18.    }

```

4.3.12. typeHandlerElement()

跟TypeAlias一样，TypeHandler 有两种配置方式，一种是单独配置一个类，一种是指定一个package。最后我们得到的是JavaType和JdbcType，以及用来做相互映射的TypeHandler 之间的映射关系，存放在TypeHandlerRegistry 对象里面。

```
1. private void typeHandlerElement(XNode parent) {
2.     if (parent != null) {
3.         for (XNode child : parent.getChildren()) {
4.             if ("package".equals(child.getName())) {
5.                 String typeHandlerPackage = child.getStringAttribute("name");
6.                 typeHandlerRegistry.register(typeHandlerPackage);
7.             } else {
8.                 String javaTypeName = child.getStringAttribute("javaType");
9.                 String jdbcTypeName = child.getStringAttribute("jdbcType");
10.                String handlerTypeName = child.getStringAttribute("handler");
11.                Class<?> javaTypeClass = resolveClass(javaTypeName);
12.                JdbcType jdbcType = resolveJdbcType(jdbcTypeName);
13.                Class<?> typeHandlerClass = resolveClass(handlerTypeName);
14.                if (javaTypeClass != null) {
15.                    if (jdbcType == null) {
16.                        typeHandlerRegistry.register(javaTypeClass, typeHandlerClass);
17.                    } else {
18.                        typeHandlerRegistry.register(javaTypeClass, jdbcType,
19.                            typeHandlerClass);
20.                    }
21.                } else {
22.                    typeHandlerRegistry.register(typeHandlerClass);
23.                }
24.            }
25.        }
26.    }
```

问题：这种三个对象（Java类型，JDBC类型，Handler）的关系怎么映射？（Map里面再放一个Map）

```
1. public final class TypeHandlerRegistry {
2.
3.     private final Map<Type, Map<JdbcType, TypeHandler<?>>> typeHandlerMap = new
4.         ConcurrentHashMap<>();
5.
6.     ...
7. }
```

4.3.13. mapperElement()

<http://www.mybatis.org/mybatis-3/zh/configuration.html#mappers>

最后就是 `<mappers>` 标签的解析。

根据全局配置文件中不同的注册方式，用不同的方式扫描，但最终都是做了两件事情，对于语句的注册和接口的注册。

扫描类型	含义
resource	相对路径
url	绝对路径
package	包
class	单个接口

```

1. private void mapperElement(XNode parent) throws Exception {
2.     if (parent != null) {
3.         for (XNode child : parent.getChildren()) {
4.             // 不同定义方式的扫描，最终都是调用addMapper()方法(添加到MapperRegistry)
5.             // 这个方法和getMapper()对应
6.             // package包
7.             if ("package".equals(child.getName())) {
8.                 String mapperPackage = child.getStringAttribute("name");
9.                 configuration.addMappers(mapperPackage);
10.            } else {
11.                String resource = child.getStringAttribute("resource");
12.                String url = child.getStringAttribute("url");
13.                String mapperClass = child.getStringAttribute("class");
14.                if (resource != null && url == null && mapperClass == null) {
15.                    // resource相对路径
16.                    ErrorContext.instance().resource(resource);
17.                    InputStream inputStream = Resources.getResourceAsStream(resource);
18.                    XMLMapperBuilder mapperParser = new XMLMapperBuilder(inputStream,
configuration, resource, configuration.getSqlFragments());
19.                    // 解析Mapper.xml，总体上做了两件事
20.                    mapperParser.parse();
21.                } else if (resource == null && url != null && mapperClass == null) {
22.                    // url绝对路径
23.                    ErrorContext.instance().resource(url);
24.                    InputStream inputStream = Resources.getUrlAsStream(url);
25.                    XMLMapperBuilder mapperParser = new XMLMapperBuilder(inputStream,
configuration, url, configuration.getSqlFragments());
26.                    mapperParser.parse();
27.                } else if (resource == null && url == null && mapperClass != null) {
28.                    // class单个接口
29.                    Class<?> mapperInterface = Resources.classForName(mapperClass);
30.                    configuration.addMapper(mapperInterface);
31.                } else {
32.                    throw new BuilderException("A mapper element may only specify a url,
resource or class, but not more than one.");
33.                }
34.            }
35.        }
36.    }
37. }

```

先从解析Mapper.xml的mapperParser.parse()方法入手。


```

1.     public void parse() {
2.         if (!configuration.isResourceLoaded(resource)) {
3.             configurationElement(parser.evalNode("/mapper"));
4.             configuration.addLoadedResource(resource);
5.             bindMapperForNamespace();
6.         }
7.
8.         parsePendingResultMaps();
9.         parsePendingCacheRefs();
10.        parsePendingStatements();
11.    }

```

configurationElement()——解析所有的子标签，最终获得MappedStatement对象。

bindMapperForNamespace()——把namespace（接口类型）和工厂类MapperProxyFactory绑定起来。

4.3.13.1. 1) configurationElement()

configurationElement是对Mapper.xml中所有具体标签的解析，包括namespace、cache、parameterMap、resultMap、sql和select|insert|update|delete。

```

1.     private void configurationElement(XNode context) {
2.         try {
3.             String namespace = context.getStringAttribute("namespace");
4.             if (namespace == null || namespace.equals("")) {
5.                 throw new BuilderException("Mapper's namespace cannot be empty");
6.             }
7.             builderAssistant.setCurrentNamespace(namespace);
8.             // 添加缓存对象
9.             cacheRefElement(context.evalNode("cache-ref"));
10.            // 解析cache属性，添加缓存对象
11.            cacheElement(context.evalNode("cache"));
12.            // 创建ParameterMapping对象
13.            parameterMapElement(context.evalNodes("/mapper/parameterMap"));
14.            // 创建List<ResultMapping>
15.            resultMapElements(context.evalNodes("/mapper/resultMap"));
16.            // 解析可以复用的SQL
17.            sqlElement(context.evalNodes("/mapper/sql"));
18.            // 解析增删改查标签，得到MappedStatement
19.            buildStatementFromContext(context.evalNodes("select|insert|update|delete"));
20.        } catch (Exception e) {
21.            throw new BuilderException("Error parsing Mapper XML. The XML location is '"
+ resource + "'. Cause: " + e, e);
22.        }
23.    }

```

在buildStatementFromContext () 方法中，创建了用来解析增删改查标签的XMLStatementBuilder，并且把创建的MappedStatement 添加到mappedStatements中。

```
1. private void buildStatementFromContext(List<XNode> list) {
2.     if (configuration.getDatabaseId() != null) {
3.         buildStatementFromContext(list, configuration.getDatabaseId());
4.     }
5.     buildStatementFromContext(list, null);
6. }
7.
8. private void buildStatementFromContext(List<XNode> list, String
requiredDatabaseId) {
9.     for (XNode context : list) {
10.         final XMLStatementBuilder statementParser = new
XMLStatementBuilder(configuration, builderAssistant, context, requiredDatabaseId);
11.         try {
12.             statementParser.parseStatementNode();
13.         } catch (IncompleteElementException e) {
14.             configuration.addIncompleteStatement(statementParser);
15.         }
16.     }
17. }
```

```
1. public class XMLStatementBuilder extends BaseBuilder {
2.     public void parseStatementNode() {
3.         ...
4.         builderAssistant.addMappedStatement(id, sqlSource, statementType,
sqlCommandType,
5.             fetchSize, timeout, parameterMap, parameterTypeClass, resultMap,
resultTypeClass,
6.             resultSetTypeEnum, flushCache, useCache, resultOrdered,
keyGenerator, keyProperty, keyColumn, databaseId, langDriver, resultSets);
7.     }
8. }
```

```
1. public class MapperBuilderAssistant extends BaseBuilder {
2.     public MappedStatement addMappedStatement(
3.         ...
4.         MappedStatement statement = statementBuilder.build();
5.         configuration.addMappedStatement(statement);
6.         return statement;
7.     }
```

4.3.13.2. 2) bindMapperForNamespace()

主要是调用了addMapper () 。

```

1. private void bindMapperForNamespace() {
2.     String namespace = builderAssistant.getCurrentNamespace();
3.     if (namespace != null) {
4.         Class<?> boundType = null;
5.         try {
6.             boundType = Resources.classForName(namespace);
7.         } catch (ClassNotFoundException e) {
8.             //ignore, bound type is not required
9.         }
10.        if (boundType != null) {
11.            if (!configuration.hasMapper(boundType)) {
12.                // Spring may not know the real resource name so we set a flag
13.                // to prevent loading again this resource from the mapper interface
14.                // look at MapperAnnotationBuilder#loadXmlResource
15.                configuration.addLoadedResource("namespace:" + namespace);
16.                configuration.addMapper(boundType);
17.            }
18.        }
19.    }
20. }

```

addMapper () 方法中，把接口类型注册到MapperRegistry中：实际上是为接口创建一个对应的MapperProxyFactory（用于为这个type提供工厂类，创建MapperProxy）。

```

1. public <T> void addMapper(Class<T> type) {
2.     mapperRegistry.addMapper(type);
3. }

```

```

1. public <T> void addMapper(Class<T> type) {
2.     if (type.isInterface()) {
3.         if (hasMapper(type)) {
4.             throw new BindingException("Type " + type + " is already known to the
MapperRegistry.");
5.         }
6.         boolean loadCompleted = false;
7.         try {
8.             knownMappers.put(type, new MapperProxyFactory<>(type));
9.             // It's important that the type is added before the parser is run
10.            // otherwise the binding may automatically be attempted by the
11.            // mapper parser. If the type is already known, it won't try.
12.            MapperAnnotationBuilder parser = new MapperAnnotationBuilder(config,
type);
13.            parser.parse();
14.            loadCompleted = true;
15.        } finally {
16.            if (!loadCompleted) {
17.                knownMappers.remove(type);
18.            }
19.        }
20.    }
21. }

```

注册了接口之后，开始解析接口类和所有方法上的注解，例如 @CacheNamespace、@Select。

此处创建了一个MapperAnnotationBuilder 专门用来解析注解。

```

1. MapperAnnotationBuilder parser = new MapperAnnotationBuilder(config, type);
2. parser.parse();

```

parse()方法中的parseCache()和parseCacheRef()方法其实是对@CacheNamespace 和 @CacheNamespaceRef这两个注解的处理。

```
1. public void parse() {
2.     String resource = type.toString();
3.     if (!configuration.isResourceLoaded(resource)) {
4.         loadXmlResource();
5.         configuration.addLoadedResource(resource);
6.         assistant.setCurrentNamespace(type.getName());
7.         parseCache();
8.         parseCacheRef();
9.         Method[] methods = type.getMethods();
10.        for (Method method : methods) {
11.            try {
12.                // issue #237
13.                if (!method.isBridge()) {
14.                    parseStatement(method);
15.                }
16.            } catch (IncompleteElementException e) {
17.                configuration.addIncompleteMethod(new MethodResolver(this, method));
18.            }
19.        }
20.    }
21.    parsePendingMethods();
22. }
```

parseStatement()方法里面的各种 getAnnotation () , 都是对相应的注解的解析, 比如 @Options, @SelectKey, @ResultMap等等。

```

1.  void parseStatement(Method method) {
2.      Class<?> parameterTypeClass = getParameterType(method);
3.      LanguageDriver languageDriver = getLanguageDriver(method);
4.      SqlSource sqlSource = getSqlSourceFromAnnotations(method, parameterTypeClass,
languageDriver);
5.      if (sqlSource != null) {
6.          Options options = method.getAnnotation(Options.class);
7.          final String mappedStatementId = type.getName() + "." + method.getName();
8.          Integer fetchSize = null;
9.          Integer timeout = null;
10.         StatementType statementType = StatementType.PREPARED;
11.         ResultSetType resultSetType = configuration.getDefaultResultSetType();
12.         SqlCommandType sqlCommandType = getSqlCommandType(method);
13.         boolean isSelect = sqlCommandType == SqlCommandType.SELECT;
14.         boolean flushCache = !isSelect;
15.         boolean useCache = isSelect;
16.
17.         KeyGenerator keyGenerator;
18.         String keyProperty = null;
19.         String keyColumn = null;
20.         if (SqlCommandType.INSERT.equals(sqlCommandType) ||
SqlCommandType.UPDATE.equals(sqlCommandType)) {
21.             // first check for SelectKey annotation - that overrides everything else
22.             SelectKey selectKey = method.getAnnotation(SelectKey.class);
23.             if (selectKey != null) {
24.                 keyGenerator = handleSelectKeyAnnotation(selectKey, mappedStatementId,
getParameterType(method), languageDriver);
25.                 keyProperty = selectKey.keyProperty();
26.             } else if (options == null) {
27.                 keyGenerator = configuration.isUseGeneratedKeys() ?
Jdbc3KeyGenerator.INSTANCE : NoKeyGenerator.INSTANCE;
28.             } else {
29.                 keyGenerator = options.useGeneratedKeys() ? Jdbc3KeyGenerator.INSTANCE :
NoKeyGenerator.INSTANCE;
30.                 keyProperty = options.keyProperty();
31.                 keyColumn = options.keyColumn();
32.             }
33.         } else {
34.             keyGenerator = NoKeyGenerator.INSTANCE;
35.         }
36.
37.         if (options != null) {
38.             if (FlushCachePolicy.TRUE.equals(options.flushCache())) {
39.                 flushCache = true;
40.             } else if (FlushCachePolicy.FALSE.equals(options.flushCache())) {
41.                 flushCache = false;
42.             }
43.             useCache = options.useCache();

```

```

44.         fetchSize = options.fetchSize() > -1 || options.fetchSize() ==
Integer.MIN_VALUE ? options.fetchSize() : null; //issue #348
45.         timeout = options.timeout() > -1 ? options.timeout() : null;
46.         statementType = options.statementType();
47.         if (options.resultSetType() != ResultSetType.DEFAULT) {
48.             resultSetType = options.resultSetType();
49.         }
50.     }
51.
52.     String resultMapId = null;
53.     ResultMap resultMapAnnotation = method.getAnnotation(ResultMap.class);
54.     if (resultMapAnnotation != null) {
55.         resultMapId = String.join(",", resultMapAnnotation.value());
56.     } else if (isSelect) {
57.         resultMapId = parseResultMap(method);
58.     }
59.
60.     assistant.addMappedStatement(
61.         mappedStatementId,
62.         sqlSource,
63.         statementType,
64.         sqlCommandType,
65.         fetchSize,
66.         timeout,
67.         // ParameterMapID
68.         null,
69.         parameterTypeClass,
70.         resultMapId,
71.         getReturnType(method),
72.         resultSetType,
73.         flushCache,
74.         useCache,
75.         // TODO gcode issue #577
76.         false,
77.         keyGenerator,
78.         keyProperty,
79.         keyColumn,
80.         // DatabaseID
81.         null,
82.         languageDriver,
83.         // ResultSets
84.         options != null ? nullOrEmpty(options.resultSets()) : null);
85.     }
86. }

```

最后同样会创建 MappedStatement对象，添加到MapperRegistry中。也就是说在XML中配置，和使用注解配置，最后起到一样的效果。

```

1.      assistant.addMappedStatement(
2.          mappedStatementId,
3.          sqlSource,
4.          statementType,
5.          sqlCommandType,
6.          fetchSize,
7.          timeout,
8.          // ParameterMapID
9.          null,
10.         parameterTypeClass,
11.         resultMapId,
12.         getReturnType(method),
13.         resultSetType,
14.         flushCache,
15.         useCache,
16.         // TODO gcode issue #577
17.         false,
18.         keyGenerator,
19.         keyProperty,
20.         keyColumn,
21.         // DatabaseID
22.         null,
23.         languageDriver,
24.         // ResultSets
25.         options != null ? nullOrEmpty(options.resultSets()) : null);
26.     }

```

4.3.13.3. 3) build()

Mapper.xml 解析完之后，调用另一个build () 方法，返回SqlSessionFactory的默认实现类DefaultSqlSessionFactory。

```

1.     public SqlSessionFactory build(Configuration config) {
2.         return new DefaultSqlSessionFactory(config);
3.     }

```

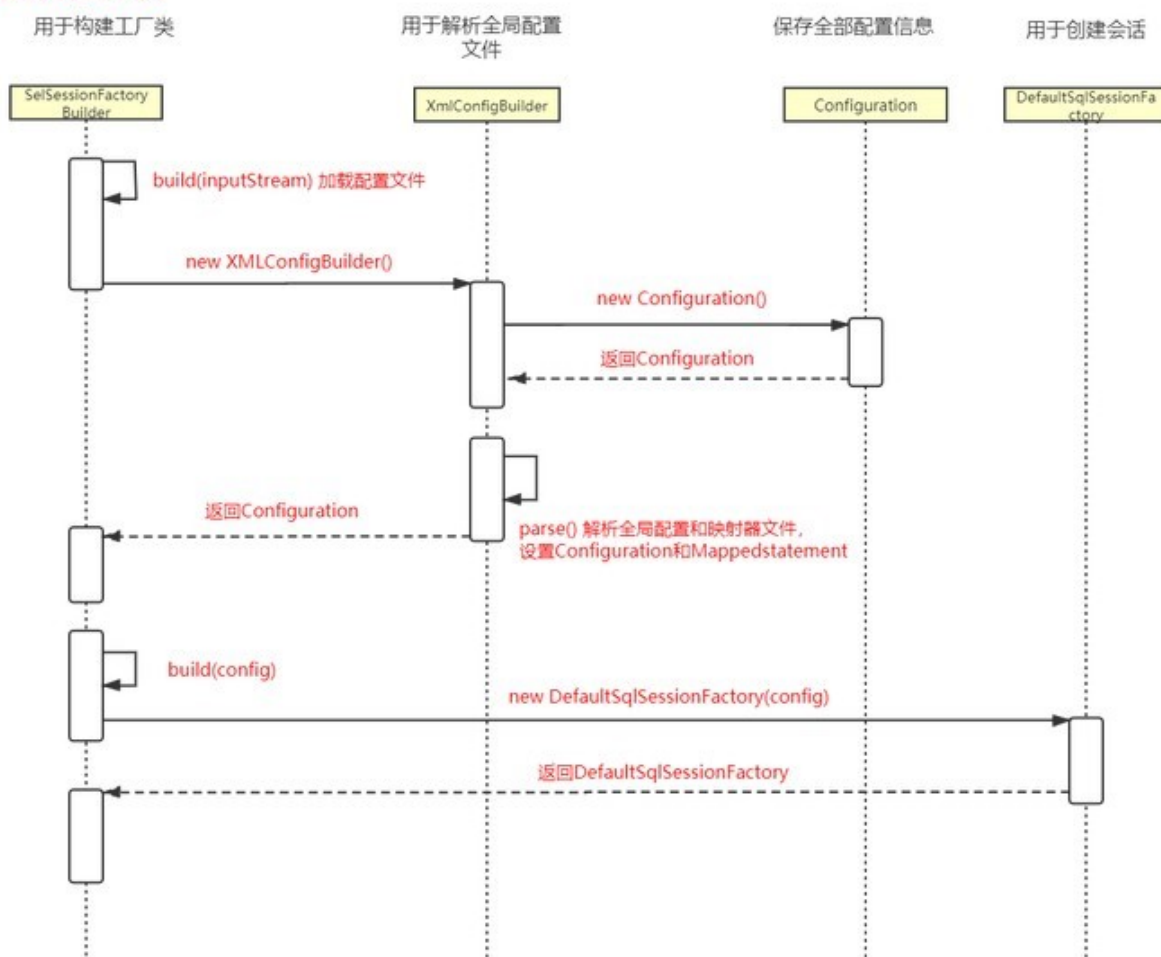
4.3.14. 总结

在这一步，我们主要完成了config 配置文件、Mapper文件、Mapper 接口中注解的解析。

我们得到了一个最重要的对象 Configuration，这里面存放了全部的配置信息，它在属性里面还有各种各样的容器。

最后，返回了一个DefaultSqlSessionFactory，里面持有了Configuration的实例。

创建会话工厂类



4.4. 二. 会话创建过程

程序每一次操作数据库，都需要创建一个会话，我们用`openSession()`方法来创建。

```
1. SqlSession session = sqlSessionFactory.openSession();
```

这里用到了上一步创建的`DefaultSqlSessionFactory`，在`openSessionFromDataSource()`方法中创建。

```
1. @Override  
2. public SqlSession openSession() {  
3.     return openSessionFromDataSource(configuration.getDefaultExecutorType(), null,  
   false);  
4. }
```

这个会话里面，需要包含一个`Executor`用来执行SQL。`Executor`又要指定事务类型和执行器的类型。

所以我们会先从`Configuration`里面拿到`Environment`，`Environment`里面就有事务工厂。

```

1.     private SqlSession openSessionFromDataSource(ExecutorType execType,
TransactionIsolationLevel level, boolean autoCommit) {
2.         Transaction tx = null;
3.         try {
4.             final Environment environment = configuration.getEnvironment();
5.             final TransactionFactory transactionFactory =
getTransactionFactoryFromEnvironment(environment);
6.             tx = transactionFactory.newTransaction(environment.getDataSource(), level,
autoCommit);
7.             final Executor executor = configuration.newExecutor(tx, execType);
8.             return new DefaultSqlSession(configuration, executor, autoCommit);
9.         } catch (Exception e) {
10.            closeTransaction(tx); // may have fetched a connection so lets call close()
11.            throw ExceptionFactory.wrapException("Error opening session. Cause: " + e,
e);
12.        } finally {
13.            ErrorContext.instance().reset();
14.        }
15.    }

```

4.4.1. 1、创建Transaction

这里会从Environment 对象中取出一个TransactionFactory，它是解析 `<environments>` 标签的时候创建的。

```

1.     final Environment environment = configuration.getEnvironment();
2.     final TransactionFactory transactionFactory =
getTransactionFactoryFromEnvironment(environment);
3.     tx = transactionFactory.newTransaction(environment.getDataSource(), level,
autoCommit);

```

事务工厂类型可以配置成JDBC或者MANAGED。

属性	产生工厂类	产生事务
JDBC	JdbcTransactionFactory	JdbcTransaction
MANAGED	ManagedTransactionFactory	ManagedTransaction

如果配置的是JDBC，则会使用Connection 对象的 commit () 、 rollback () 、 close() 管理事务。

如果配置成MANAGED，会把事务交给容器来管理，比如JBOSS，Weblogic。因为我们跑的是本地程序，如果配置成MANAGE不会有任何事务。

如果是Spring+MyBatis，则没有必要配置，因为我们会直接在applicationContext.xml 里面配置数据源和事务管理器，覆盖MyBatis的配置。

4.4.2. 2、创建 Executor

使用newExecutor 方法创建：

```
1. final Executor executor = configuration.newExecutor(tx, execType);
```

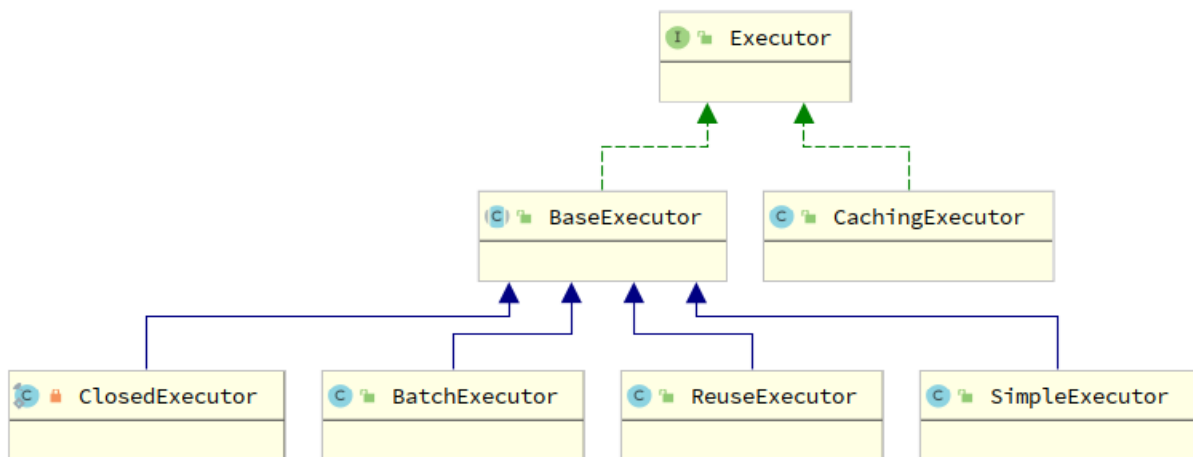
可以细分成三步。

4.4.2.1. 1) 创建执行器

Executor的基本类型有三种：SIMPLE、BATCH、REUSE，默认是SIMPLE（settingsElement（）读取默认值）。

```
1. Executor executor;  
2. if (ExecutorType.BATCH == executorType) {  
3.     executor = new BatchExecutor(this, transaction);  
4. } else if (ExecutorType.REUSE == executorType) {  
5.     executor = new ReuseExecutor(this, transaction);  
6. } else {  
7.     executor = new SimpleExecutor(this, transaction);  
8. }
```

他们都继承了抽象类BaseExecutor。抽象类实现了Executor接口。



为什么要让抽象类BaseExecutor实现Executor接口，然后让具体实现类继承抽象类？

这是模板方法的体现。

模板方法定义一个算法的骨架，并允许子类为一个或者多个步骤提供实现。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法的某些步骤。

抽象方法是在子类中实现的，BaseExecutor 最终会调用到具体的子类。

```

1.     protected abstract int doUpdate(MappedStatement ms, Object parameter)
2.         throws SQLException;
3.
4.     protected abstract List<BatchResult> doFlushStatements(boolean isRollback)
5.         throws SQLException;
6.
7.     protected abstract <E> List<E> doQuery(MappedStatement ms, Object parameter,
8. RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql)
9.         throws SQLException;
10.
11.    protected abstract <E> Cursor<E> doQueryCursor(MappedStatement ms, Object
    parameter, RowBounds rowBounds, BoundSql boundSql)
        throws SQLException;

```

4.4.2.2. 2) 缓存装饰

如果cacheEnabled=true, 会用装饰器模式对executor 进行装饰。

```

1.     if (cacheEnabled) {
2.         executor = new CachingExecutor(executor);
3.     }

```

4.4.2.3. 3) 插件代理

装饰完毕后, 会执行:

```

1.     executor = (Executor) interceptorChain.pluginAll(executor);

```

此处会对executor植入插件逻辑。

4.4.2.4. 4) 返回SqlSession 实现类

最终返回DefaultSqlSession, 它的属性包括Configuration、Executor对象。

```

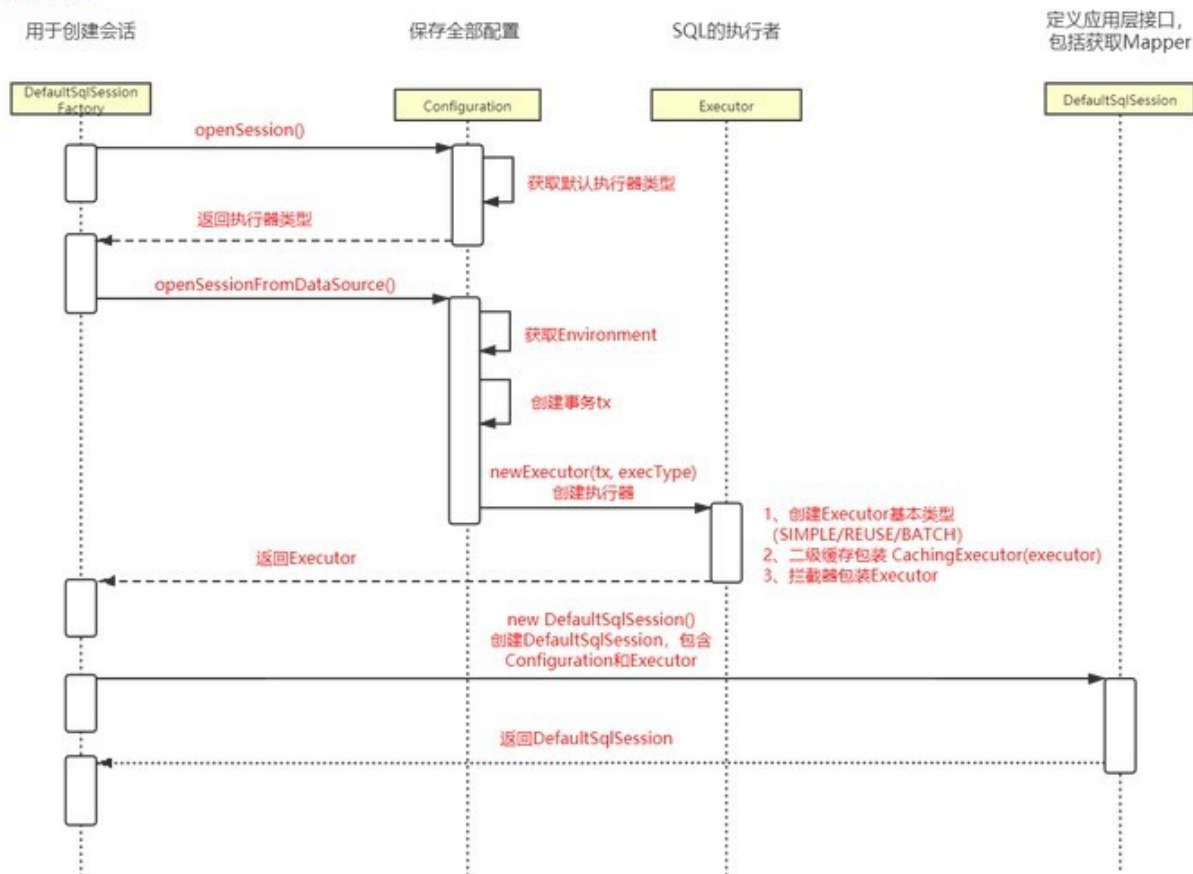
1.     return new DefaultSqlSession(configuration, executor, autoCommit);

```

4.4.3. 总结

创建会话的过程, 我们获得了一个DefaultSqlSession, 里面包含了一个Executor, Executor是SQL的实际执行对象。

创建会话



4.5. 三. 获得Mapper对象

在旧版的MyBatis中，DefaultSqlSession的selectOne()方法可以直接根据Mapper.xml中的Statement ID，找到SQL执行。但是这种方式属于硬编码，我们没办法知道有多少处调用，修改起来也很麻烦。

另一个问题是如果参数传入错误，在编译阶段也是不会报错的，不利于预先发现问题。

```

1.  /**
2.   * 使用 MyBatis API方式
3.   *
4.   * @throws IOException
5.   */
6.  @Test
7.  public void testStatement() throws IOException {
8.      SqlSession session = sqlSessionFactory.openSession();
9.      try {
10.         Blog blog = (Blog)
session.selectOne("cn.sitedev.mapper.BlogMapper.selectBlogById", 1);
11.         System.out.println(blog);
12.     } finally {
13.         session.close();
14.     }
15. }
  
```

在MyBatis 后期的版本提供了第二种调用方式，就是定义一个接口，然后再调用Mapper接口的方法。

由于我们的接口名称跟Mapper.xml的namespace是对应的，接口的方法跟statement ID也都是对应的，所以根据方法就能找到对应的要执行的SQL。

```
1.      /**
2.       * 通过 SqlSession.getMapper(XXXMapper.class) 接口方式
3.       *
4.       * @throws IOException
5.       */
6.      @Test
7.      public void testSelect() throws IOException {
8.          SqlSession session = sqlSessionFactory.openSession(); //
9.          ExecutorType.BATCH
10.         try {
11.             BlogMapper mapper = session.getMapper(BlogMapper.class);
12.             Blog blog = mapper.selectBlogById(1);
13.             System.out.println(blog);
14.         } finally {
15.             session.close();
16.         }
17.     }
```

这里有两个问题需要解决：

- 1、getMapper获得的是一个什么对象？为什么可以执行它的方法？
- 2、到底是怎么根据Mapper找到XML中的SQL执行的？

4.5.1. 1、getMapper()

DefaultSqlSession的 getMapper () 方法，调用了Configuration的 getMapper () 方法。

```
1.      @Override
2.      public <T> T getMapper(Class<T> type) {
3.          return configuration.getMapper(type, this);
4.      }
```

Configuration的getMapper()方法，又调用了MapperRegistry的 getMapper()方法。

```
1.      public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
2.          return mapperRegistry.getMapper(type, sqlSession);
3.      }
```

我们知道，在解析mapper标签和Mapper.xml的时候已经把接口类型和类型对应的MapperProxyFactory放到了一个Map中。获取Mapper代理对象，实际上是从Map中获取对应的工厂类后，调用以下方法创建对象：

```

1.     public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
2.         final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>)
knownMappers.get(type);
3.         if (mapperProxyFactory == null) {
4.             throw new BindingException("Type " + type + " is not known to the
MapperRegistry.");
5.         }
6.         try {
7.             return mapperProxyFactory.newInstance(sqlSession);
8.         } catch (Exception e) {
9.             throw new BindingException("Error getting mapper instance. Cause: " + e, e);
10.        }
11.    }

```

在newInstance () 方法中，先创建MapperProxy。

MapperProxy 实现了InvocationHandler 接口，主要属性有三个：sqlSession、mapperInterface、methodCache。

```

1.     public T newInstance(SqlSession sqlSession) {
2.         final MapperProxy<T> mapperProxy = new MapperProxy<>(sqlSession,
mapperInterface, methodCache);
3.         return newInstance(mapperProxy);
4.     }

```



```

▼ mapper = {$Proxy2@1618} "org.apache.ibatis.binding.MapperProxy@42a48628"
  ▼ f h = {MapperProxy@1620}
    > f sqlSession = {DefaultSqlSession@1617}
    > f mapperInterface = {Class@1594} "interface com.gupaoedu.mapper.BlogMapper" ... Navigate
    f methodCache = {ConcurrentHashMap@1621} size = 0

```

最终通过JDK动态代理模式创建、返回代理对象：

```

1.     protected T newInstance(MapperProxy<T> mapperProxy) {
2.         return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new
Class[] { mapperInterface }, mapperProxy);
3.     }

```

也就是说，getMapper () 返回的是一个JDK动态代理对象（类型是\$Proxy数字）。这个代理对象会继承Proxy类，实现被代理的接口，里面持有了一个MapperProxy类型的触发管理类。

回答了前面的问题：为什么要在MapperRegistry中保存一个工厂类，原来它是用来创建返回代理类的。

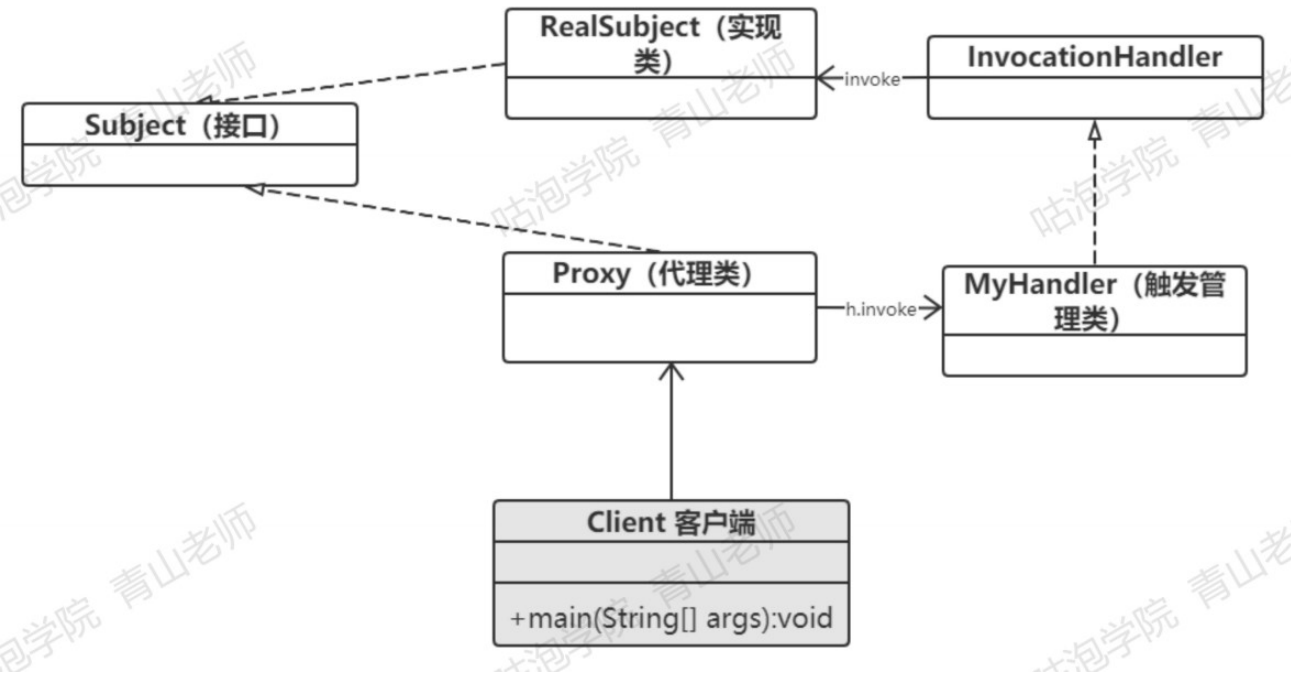
这里是代理模式的一个非常经典的应用。

但是为什么要直接代理一个接口呢？

4.5.2. 2、MapperProxy如何实现对接口的代理

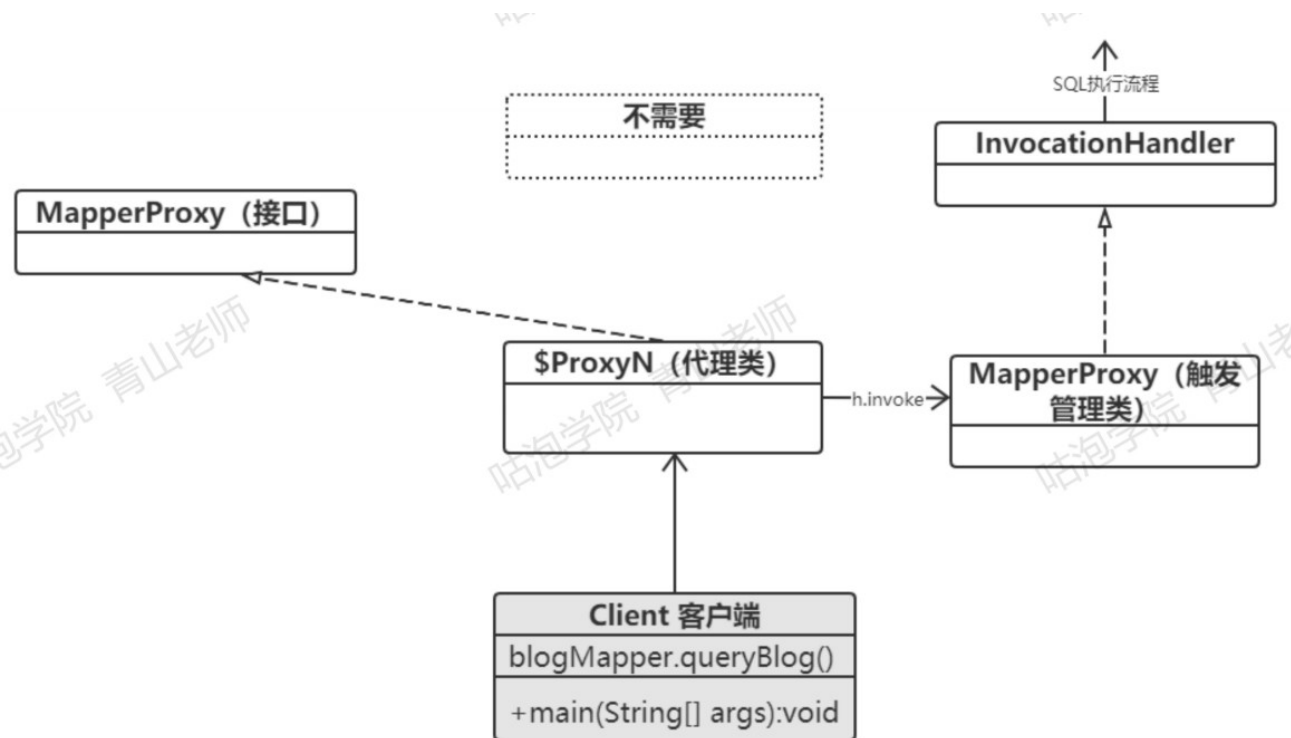
我们知道，JDK的动态代理，有三个核心角色：被代理类（实现类）、接口、实现了InvocationHandler的触发管理类，用来生成代理对象。

被代理类必须实现接口，因为要通过接口获取方法，而且代理类也要实现这个接口。



而MyBatis 里面的Mapper 没有实现类，怎么被代理呢？它忽略了实现类，直接对接口进行代理。

MyBatis的动态代理：



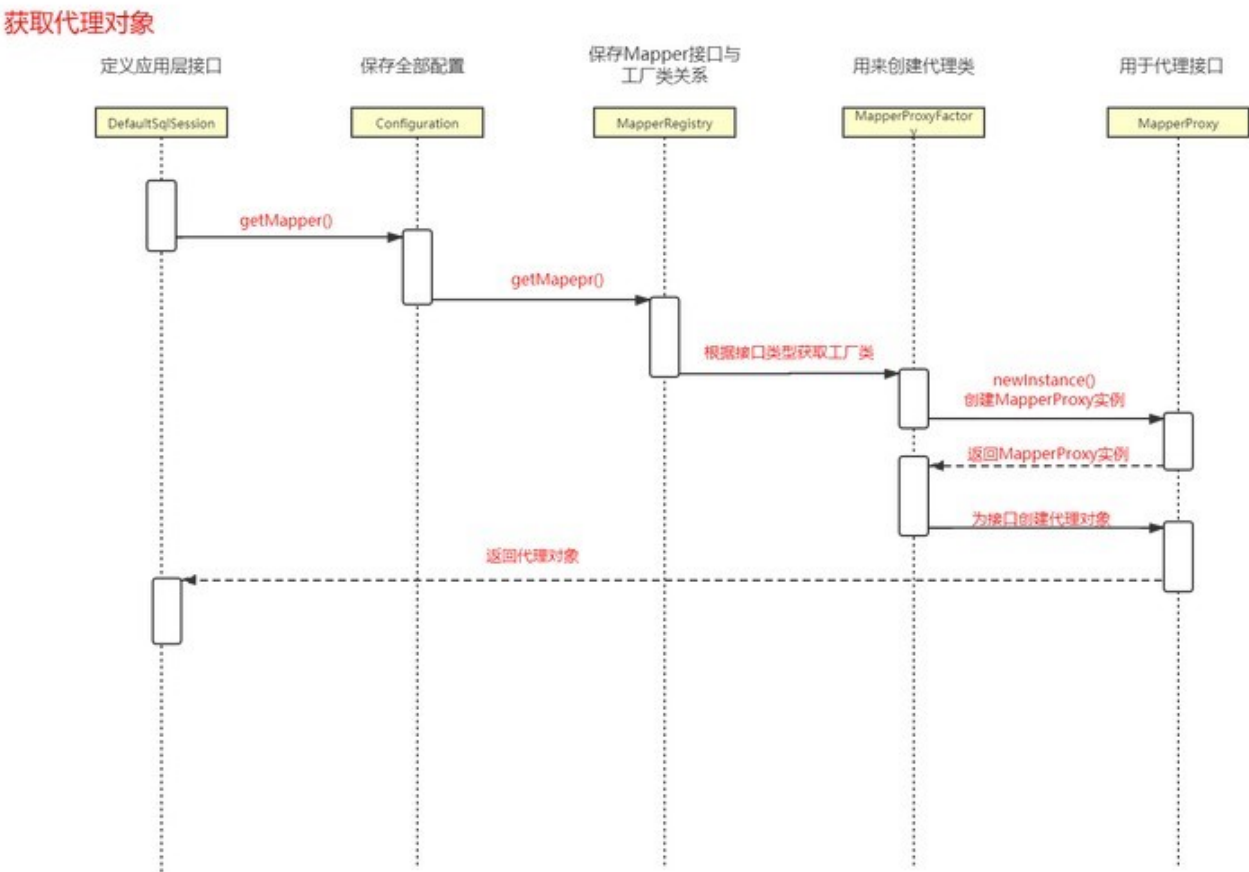
在MyBatis里面，动态代理为什么不需要实现类呢？

这里我们要想想我们的目的。我们的目的是根据一个可以执行的方法，直接找到Mapper.xml中的Statement ID，方便调用。

如果根据接口类型+方法的名称找到Statement ID这个逻辑在Handler类（MapperProxy）中就可以完成，其实也就没有实现类的什么事了。

4.5.3. 3、总结

获得Mapper对象的过程，实质上是获取了一个JDK动态代理对象（类型是\$Proxy数字）。这个代理类会继承Proxy类，实现被代理的接口，里面持有了一个MapperProxy类型的触发管理类。



4.6. 四. 执行SQL

```
1. Blog blog1 = mapper.selectBlogById(1);
```

由于所有的Mapper都是JDK动态代理对象，所以任意的方法都是执行触发管理类MapperProxy的invoke()方法。

问题1：我们引入MapperProxy为了解决什么问题？硬编码和编译时检查问题。它需要做的事情是：根据方法查找Statement ID的问题。

问题2：进入到invoke方法的时候做了什么事情？它是怎么找到我们要执行的SQL的？

我们看一下invoke () 方法：

4.6.1. 1、MapperProxy.invoke()

1) 首先判断是否需要去执行SQL，还是直接执行方法。Object本身的方法不需要去执行SQL，比如toString()、hashCode()、equals()、getClass ()。

```
1.         if (Object.class.equals(method.getDeclaringClass())) {
2.             return method.invoke(this, args);
3.         } else {
```

2) 获取缓存

这里加入缓存是为了提升MapperMethod的获取速度。很巧妙的设计。缓存的使用在MyBatis中随处可见。

```
1.     // 获取缓存，保存了方法签名和接口方法的关系
2.     final MapperMethod mapperMethod = cachedMapperMethod(method);
```

Map的computeIfAbsent () 方法：根据key获取值，如果值是null，则把后面Object 的值赋给key。

Java8和Java9中的接口默认方法有特殊处理，返回 DefaultMethodInvoker。

```
1.     private MapperMethodInvoker cachedInvoker(Method method) throws Throwable {
2.         try {
3.             return methodCache.computeIfAbsent(method, m -> {
4.                 if (m.isDefault()) {
5.                     try {
6.                         if (privateLookupInMethod == null) {
7.                             return new DefaultMethodInvoker(getMethodHandleJava8(method));
8.                         } else {
9.                             return new DefaultMethodInvoker(getMethodHandleJava9(method));
10.                        }
11.                    } catch (IllegalAccessException | InstantiationException |
12.                        InvocationTargetException
13.                        | NoSuchMethodException e) {
14.                        throw new RuntimeException(e);
15.                    }
16.                } else {
17.                    return new PlainMethodInvoker(new MapperMethod(mapperInterface, method,
18.                        sqlSession.getConfiguration()));
19.                }
20.            });
21.        } catch (RuntimeException re) {
22.            Throwable cause = re.getCause();
23.            throw cause == null ? re : cause;
24.        }
```

普通的方法返回的是PlainMethodInvoker，返回MapperMethod。

MapperMethod中有两个主要的属性：

```
1. private final SqlCommand command;
2. private final MethodSignature method;
```

一个是SqlCommand，封装了statement id（例如：cn.sitedev.mapper.BlogMapper.selectBlogById）和SQL类型。

```
1. public static class SqlCommand {
2.
3.     private final String name;
4.     private final SqlCommandType type;
```

一个是MethodSignature，主要是主要封装是返回值的类型。

```
1. public static class MethodSignature {
2.
3.     private final boolean returnsMany;
4.     private final boolean returnsMap;
5.     private final boolean returnsVoid;
6.     private final boolean returnsCursor;
7.     private final boolean returnsOptional;
8.     private final Class<?> returnType;
9.     private final String mapKey;
10.    private final Integer resultHandlerIndex;
11.    private final Integer rowBoundsIndex;
12.    private final ParamNameResolver paramNameResolver;
```

这两个属性都是MapperMethod的内部类。

另外MapperMethod种定义了多种execute()方法。

4.6.2. 2、MapperMethod.execute()

接下来又调用了mapperMethod的 execute方法：

```
1. private static class PlainMethodInvoker implements MapperMethodInvoker {
2.     ...
3.     @Override
4.     public Object invoke(Object proxy, Method method, Object[] args, SqlSession
sqlSession) throws Throwable {
5.         return mapperMethod.execute(sqlSession, args);
6.     }
```

```
mapperMethod = {MapperMethod@1628}
  ✓ f command = {MapperMethod$SqlCommand@1634}
    > f name = "com.gupaoedu.mapper.BlogMapper.selectBlog"
    > f type = {SqlCommandType@1639} "SELECT"
  ✓ f method = {MapperMethod$MethodSignature@1635}
    f returnsMany = false
    f returnsMap = false
    f returnsVoid = false
    f returnsCursor = false
    f returnsOptional = false
    > f returnType = {Class@1505} "class com.gupaoedu.domain.Blog" ...
    f mapKey = null
    f resultHandlerIndex = null
    f rowBoundsIndex = null
    > f paramNameResolver = {ParamNameResolver@1636}
```

在这一步，根据不同的type（INSERT、UPDATE、DELETE、SELECT）和返回类型：

1) 调用convertArgsToSqlCommandParam（）将方法参数转换为SQL的参数。

```

1. public Object execute(SqlSession sqlSession, Object[] args) {
2.     Object result;
3.     switch (command.getType()) {
4.         case INSERT: {
5.             Object param = method.convertArgsToSqlCommandParam(args);
6.             result = rowCountResult(sqlSession.insert(command.getName(), param));
7.             break;
8.         }
9.         case UPDATE: {
10.            Object param = method.convertArgsToSqlCommandParam(args);
11.            result = rowCountResult(sqlSession.update(command.getName(), param));
12.            break;
13.        }
14.        case DELETE: {
15.            Object param = method.convertArgsToSqlCommandParam(args);
16.            result = rowCountResult(sqlSession.delete(command.getName(), param));
17.            break;
18.        }
19.        case SELECT:
20.            if (method.returnsVoid() && method.hasResultHandler()) {
21.                executeWithResultHandler(sqlSession, args);
22.                result = null;
23.            } else if (method.returnsMany()) {
24.                result = executeForMany(sqlSession, args);
25.            } else if (method.returnsMap()) {
26.                result = executeForMap(sqlSession, args);
27.            } else if (method.returnsCursor()) {
28.                result = executeForCursor(sqlSession, args);
29.            } else {
30.                Object param = method.convertArgsToSqlCommandParam(args);
31.                result = sqlSession.selectOne(command.getName(), param);
32.                if (method.returnsOptional()
33.                    && (result == null ||
!method.getReturnType().equals(result.getClass()))) {
34.                    result = Optional.ofNullable(result);
35.                }
36.            }
37.            break;
38.        case FLUSH:
39.            result = sqlSession.flushStatements();
40.            break;
41.        default:
42.            throw new BindingException("Unknown execution method for: " +
command.getName());
43.    }
44.    if (result == null && method.getReturnType().isPrimitive() &&
!method.returnsVoid()) {
45.        throw new BindingException("Mapper method '" + command.getName()

```

```

46.         + " attempted to return null from a method with a primitive return type
        (" + method.getReturnType() + ").");
47.     }
48.     return result;
49. }

```

2) 调用sqlSession的insert () 、 update () 、 delete()、 selectOne()方法。我们以查询为例，会走到selectOne()方法。

```

1.         Object param = method.convertArgsToSqlCommandParam(args);
2.         result = sqlSession.selectOne(command.getName(), param);

```

4.6.3. 3、DefaultSqlSession.selectOne()

这里来到了对外的接口的默认实现类DefaultSqlSession。

selectOne()最终也是调用了selectList () 。

```

1.     @Override
2.     public <T> T selectOne(String statement, Object parameter) {
3.         // Popular vote was to return null on 0 results and throw exception on too
        many.
4.         List<T> list = this.selectList(statement, parameter);
5.         if (list.size() == 1) {
6.             return list.get(0);
7.         } else if (list.size() > 1) {
8.             throw new TooManyResultsException("Expected one result (or null) to be
                returned by selectOne(), but found: " + list.size());
9.         } else {
10.            return null;
11.        }
12.    }

```

在SelectList () 中，我们先根据 command name (Statement ID) 从Configuration中拿到MappedStatement。ms 里面有xml中增删改查标签配置的所有属性，包括id、statementType、sqlSource、useCache、入参、出参等等。

```

1.  @Override
2.  public <E> List<E> selectList(String statement, Object parameter, RowBounds
rowBounds) {
3.      try {
4.          MappedStatement ms = configuration.getMappedStatement(statement);
5.          return executor.query(ms, wrapCollection(parameter), rowBounds,
Executor.NO_RESULT_HANDLER);
6.      } catch (Exception e) {
7.          throw ExceptionFactory.wrapException("Error querying database. Cause: " +
e, e);
8.      } finally {
9.          ErrorContext.instance().reset();
10.     }

```

```

ms = {MappedStatement@1638}
> f resource = "BlogMapper.xml"
> f configuration = {Configuration@1639}
> f id = "com.gupaoedu.mapper.BlogMapper.selectBlog"
  f fetchSize = null
  f timeout = null
> f statementType = {StatementType@1642} "PREPARED"
> f resultSetType = {ResultSetType@1643} "DEFAULT"
> f sqlSource = {RawSqlSource@1644}
> f cache = {LoggingCache@1645}
> f parameterMap = {ParameterMap@1646}
> f resultMaps = {Collections$UnmodifiableRandomAccessList@1647} size = 1
  f flushCacheRequired = false
  f useCache = true
  f resultOrdered = false
> f sqlCommandType = {SqlCommandType@1648} "SELECT"
> f keyGenerator = {NoKeyGenerator@1649}
  f keyProperties = null
  f keyColumns = null
  f hasNestedResultMaps = false
  f databaseId = null
> f statementLog = {StdOutImpl@1650}
> f lang = {XMLLanguageDriver@1651}
  f resultSets = null

```

然后执行了Executor的query()方法。

Executor是第二步openSession的时候创建的，创建了执行器基本类型之后，依次执行了二级缓存装饰，和插件拦截。

所以，如果有被插件拦截，这里会先走到插件的逻辑。如果没有显式地在settings中配置cacheEnabled=false，再走到CachingExecutor的逻辑，然后会走到BaseExecutor的query()方法。

4.6.4. 4、CachingExecutor.query()

4.6.4.1. 1) 创建 Cachekey

二级缓存的CacheKey是怎么构成的呢？或者说，什么样的查询才能确定是同一个查询呢？

在BaseExecutor的createCacheKey方法中，用到了六个要素：

```
1. public CacheKey createCacheKey(MappedStatement ms, Object parameterObject,
   RowBounds rowBounds, BoundSql boundSql) {
2.     ...
3.     CacheKey cacheKey = new CacheKey();
4.     cacheKey.update(ms.getId());
5.     cacheKey.update(rowBounds.getOffset());
6.     cacheKey.update(rowBounds.getLimit());
7.     cacheKey.update(boundSql.getSql());
8.     ...
9.     cacheKey.update(value);
10.    ... cacheKey.update(configuration.getEnvironment().getId());
```

```
cacheKey.update(ms.getId()); // com.gupaoedu.mapper.BlogMapper.selectBlogById
cacheKey.update(rowBounds.getOffset()); // 0
cacheKey.update(rowBounds.getLimit()); // 2147483647 = 2^31-1
cacheKey.update(boundSql.getSql());
cacheKey.update(value);
cacheKey.update(configuration.getEnvironment().getId());
```

也就是说，方法相同、翻页偏移相同、SQL相同、参数值相同、数据源环境相同，才会被认为是同一个查询。

Cachekey的实际值举例（toString()生成的），debug可以看到：

```
-1381545870:4796102018:com.gupaoedu.mapper.BlogMapper.selectBlogById:0:2147483647:select * from
blog where bid = ?:1:development
```

注意看一下CacheKey的属性，里面有一个List 按顺序存放了这些要素。


```
1. public class CacheKey implements Cloneable, Serializable {
2.     ...
3.     private static final int DEFAULT_MULTIPLIER = 37;
4.     private static final int DEFAULT_HASHCODE = 17;
5.
6.     private final int multiplier;
7.     private int hashCode;
8.     private long checksum;
9.     private int count;
10.    // 8/21/2017 - Sonarlint flags this as needing to be marked transient. While
    true if content is not serializable, this is not always true and thus should not
    be marked transient.
11.    private List<Object> updateList;
12.    ...
```

怎么比较两个CacheKey是否相等呢？如果一上来就是依次比较六个要素是否相等，要比较6次，这样效率不高。有没有更高效的方法呢？继承Object的每个类，都有一个hashCode () 方法，用来生成哈希码。它是用来在集合中快速判重的。

在生成CacheKey的时候（update方法），也更新了CacheKey的hashCode，它是用乘法哈希生成的（基数baseHashCode=17，乘法因子multiplier=37）。

```
1.     hashCode = multiplier * hashCode + baseHashCode;
```

Object中的hashCode()是一个本地方法，通过随机数算法生成（OpenJDK8，默认，可以通过-XX:hashCode修改）。CacheKey中的hashCode () 方法进行了重写，返回自己生成的hashCode。

为什么要用37作为乘法因子呢？跟String中的31类似。

CacheKey中的 equals 也进行了重写，比较CacheKey是否相等。

```

1.  @Override
2.  public boolean equals(Object object) {
3.      if (this == object) {
4.          return true;
5.      }
6.      if (!(object instanceof CacheKey)) {
7.          return false;
8.      }
9.
10.     final CacheKey cacheKey = (CacheKey) object;
11.
12.     if (hashCode != cacheKey.hashCode) {
13.         return false;
14.     }
15.     if (checksum != cacheKey.checksum) {
16.         return false;
17.     }
18.     if (count != cacheKey.count) {
19.         return false;
20.     }
21.
22.     for (int i = 0; i < updateList.size(); i++) {
23.         Object thisObject = updateList.get(i);
24.         Object thatObject = cacheKey.updateList.get(i);
25.         if (!ArrayUtil.equals(thisObject, thatObject)) {
26.             return false;
27.         }
28.     }
29.     return true;
30. }

```

如果哈希值（乘法哈希）、校验值（加法哈希）、要素个数任何一个不相等，都不是同一个查询，最后才循环比较要素，防止哈希碰撞。

Cachekey生成之后，调用另一个query()方法。

4.6.4.2. 2) 处理二级缓存

首先从ms中取出cache对象，判断cache 对象是否为空，如果为空，则没有查询二级缓存、写入二级缓存的流程。

```

1.  public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds
    rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql)
2.      throws SQLException {
3.      Cache cache = ms.getCache();
4.      if (cache != null) {

```

Cache 对象是什么时候创建的呢？

用来解析Mapper.xml的XMLMapperBuilder类，cacheElement()方法：

```
1. cacheElement(context.evalNode("cache"));
```

只有Mapper.xml中的 `<cache>` 标签不为空才解析。

```
1. builderAssistant.useNewCache(typeClass, evictionClass, flushInterval, size,
    readWrite, blocking, props);
```

此处创建了一个Cache对象。

```
1. public Cache useNewCache(Class<? extends Cache> typeClass,
2.     Class<? extends Cache> evictionClass,
3.     Long flushInterval,
4.     Integer size,
5.     boolean readWrite,
6.     boolean blocking,
7.     Properties props) {
8.     Cache cache = new CacheBuilder(currentNamespace)
9.         .implementation(valueOrDefault(typeClass, PerpetualCache.class))
10.        .addDecorator(valueOrDefault(evictionClass, LruCache.class))
11.        .clearInterval(flushInterval)
12.        .size(size)
13.        .readWrite(readWrite)
14.        .blocking(blocking)
15.        .properties(props)
16.        .build();
17.    configuration.addCache(cache);
18.    currentCache = cache;
19.    return cache;
20. }
```

大家可以自行验证一下，注释 `<cache>`，是否不解析，cache是否为null。开启 `<cache>`，是否解析，cache 是否不为null。

二级缓存为什么要用TCM来管理？我们来思考一个问题，在一个事务中：

- 1、首先插入一条数据（没有提交），此时二级缓存会被清空。
- 2、在这个事务中查询数据，写入二级缓存。
- 3、提交事务，出现异常，数据回滚。

此时出现了数据库没有这条数据，但是二级缓存有这条数据的情况。所以MyBatis的二级缓存需要跟事务关联起来。疑问：为什么一级缓存不这么做？

因为一个session就是一个事务，事务回滚，会话就结束了，缓存也清空了，不存在读到一级缓存中脏数据的情况。二级缓存是跨session的，也就是跨事务的，才有可能出现对同一个方法的不同事务访问。

4.6.4.2.1. 1) 写入二级缓存

```
1. tcm.putObject(cache, key, list);
```

从map 中拿出TransactionalCache对象，把 value添加到待提交的Map。此时缓存还没有真正地写入。

```
1. @Override
2. public void putObject(Object key, Object object) {
3.     entriesToAddOnCommit.put(key, object);
4. }
```

只有事务提交的时候缓存才真正写入（close或者commit最后分析）。

4.6.4.2.2. 2) 获取二级缓存

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

从map中拿出TransactionalCache对象，这个对象也是对PerpetualCache经过层层装饰的缓存对象：

☞

```
✓ delegate = {SynchronizedCache@2009}
  ✓ delegate = {LoggingCache@2326}
    > log = {StdOutImpl@2331}
    ✓ delegate = {SerializedCache@2332}
      ✓ delegate = {ScheduledCache@2333}
        ✓ delegate = {LruCache@2334}
          ✓ delegate = {PerpetualCache@2335}
            > id = "com.gupaoedu.mapper.BlogMapper"
            > cache = {HashMap@2338} size = 1
```

得到再 getObject ()，这个是一个会递归调用的方法，直到到达PerpetualCache，拿到value。

```
1. @Override
2. public Object getObject(Object key) {
3.     return cache.get(key);
4. }
```

4.6.5. 5、BaseExecutor.query()

4.6.5.1. 1) 清空本地缓存

queryStack 用于记录查询栈，防止递归查询重复处理缓存。

flushCache=true的时候，会先清理本地缓存（一级缓存）：

```
1.     if (queryStack == 0 && ms.isFlushCacheRequired()) {  
2.         clearLocalCache();  
3.     }
```

如果没有缓存，会从数据库查询：queryFromDatabase()

```
1. list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key, boundSql);
```

如果LocalCacheScope==STATEMENT，会清理本地缓存。

```
1.     if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {  
2.         // issue #482  
3.         clearLocalCache();  
4.     }
```

4.6.5.2. 2) 从数据库查询

a) 缓存

先在缓存用占位符占位。执行查询后，移除占位符，放入数据。

```
1. localCache.putObject(key, EXECUTION_PLACEHOLDER);
```

b) 查询

执行Executor的doQuery ()；默认是SimpleExecutor。

```
1. list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
```

4.6.6. 6、SimpleExecutor.doQuery()

4.6.6.1. 1) 创建 StatementHandler

在configuration.newStatementHandler () 中，new一个StatementHandler，先得到RoutingStatementHandler。

```
1. StatementHandler handler = configuration.newStatementHandler(wrapper, ms,  
    parameter, rowBounds, resultHandler, boundSql);
```

RoutingStatementHandler里面没有任何的实现，是用来创建基本的StatementHandler的。这里会根据MappedStatement里面的 statementType决定StatementHandler的类型。默认是PREPARED (STATEMENT、PREPARED、CALLABLE) 。

```

1.     public RoutingStatementHandler(Executor executor, MappedStatement ms, Object
parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) {
2.
3.         switch (ms.getStatementType()) {
4.             case STATEMENT:
5.                 delegate = new SimpleStatementHandler(executor, ms, parameter, rowBounds,
resultHandler, boundSql);
6.                 break;
7.             case PREPARED:
8.                 delegate = new PreparedStatementHandler(executor, ms, parameter,
rowBounds, resultHandler, boundSql);
9.                 break;
10.            case CALLABLE:
11.                delegate = new CallableStatementHandler(executor, ms, parameter,
rowBounds, resultHandler, boundSql);
12.                break;
13.            default:
14.                throw new ExecutorException("Unknown statement type: " +
ms.getStatementType());
15.        }
16.
17.    }

```

StatementHandler 里面包含了处理参数的ParameterHandler和处理结果集的ResultSetHandler。这两个对象都是在上面new的时候创建的。

```

1.     this.parameterHandler = configuration.newParameterHandler(mappedStatement,
parameterObject, boundSql);
2.     this.resultSetHandler = configuration.newResultSetHandler(executor,
mappedStatement, rowBounds, parameterHandler, resultHandler, boundSql);

```

这三个对象都是可以被插件拦截的四大对象之一，所以在创建之后都要用拦截器进行包装的方法。

```

1.     statementHandler=(StatementHandler) interceptorChain. pluginAll(statementHandler);
2.
3.     parameterHandler=(ParameterHandler) interceptorChain. pluginAll(parameterHandler);
4.
5.     resultSetHandler=(ResultSetHandler) interceptorChain. plugin
All(resultSetHandler);

```

至此，四大对象的包装已经全部完成。

PS：四大对象还有一个是谁？在什么时候创建的？（Executor）

4.6.6.2.2) 创建Statement

用new 出来的StatementHandler 创建 Statement对象。

```

1.  @Override
2.  public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds
    rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws SQLException {
3.      ...
4.      stmt = prepareStatement(handler, ms.getStatementLog());

```

如果有插件包装，会先走到被拦截的业务逻辑。

```

1.  private Statement prepareStatement(StatementHandler handler, Log statementLog)
    throws SQLException {
2.      ...
3.      stmt = handler.prepare(connection, transaction.getTimeout());
4.      ...
5.  }

```

prepareStatement () 方法对语句进行预编译，处理参数：

```

1.  private Statement prepareStatement(StatementHandler handler, Log statementLog)
    throws SQLException {
2.      ...
3.      handler.parameterize(stmt);
4.      ...
5.  }

```

这里面会调用parameterHandler设置参数，如果有插件包装，会先走到被拦截的业务逻辑。

```

1.  @Override
2.  public void parameterize(Statement statement) throws SQLException {
3.      parameterHandler.setParameters((PreparedStatement) statement);
4.  }

```

4.6.6.3. 3) 执行的StatementHandler的query()方法

```

1.  @Override
2.  public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds
    rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws SQLException {
3.      ...
4.      return handler.query(stmt, resultHandler);
5.      ...
6.  }

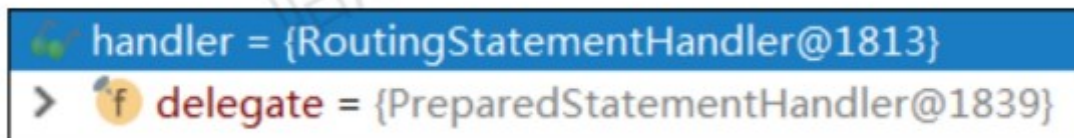
```

RoutingStatementHandler的 query () 方法。

```
1.  @Override
2.  public <E> List<E> query(Statement statement, ResultHandler resultHandler)
    throws SQLException {
3.      return delegate.query(statement, resultHandler);
4.  }
```

delegate 委派, 最终执行PreparedStatementHandler的 query()方法。

```
1.  @Override
2.  public <E> List<E> query(Statement statement, ResultHandler resultHandler)
    throws SQLException {
3.      PreparedStatement ps = (PreparedStatement) statement;
4.      ps.execute();
5.      return resultSetHandler.handleResultSets(ps);
6.  }
```



handler = {RoutingStatementHandler@1813}
> delegate = {PreparedStatementHandler@1839}

4.6.6.4. 4) 执行PreparedStatement的execute()

方法后面就是JDBC包中的PreparedStatement的执行了。


```
1.     public boolean execute() throws SQLException {
2.         synchronized (checkClosed()) {
3.
4.             MySqlConnection locallyScopedConn = this.connection;
5.
6.             if(!checkReadOnlySafeStatement()) {
7.                 throw
8.                 SQLException.createSQLException(Messages.getString("PreparedStatement.20") //$NON-NLS-
9.                 1$
10.                    + Messages.getString("PreparedStatement.21"),
11.                //$NON-NLS-1$
12.                SQLException.SQL_STATE_ILLEGAL_ARGUMENT,
13.                getExceptionInterceptor());
14.             }
15.
16.             ResultSetInternalMethods rs = null;
17.
18.             CachedResultSetMetaData cachedMetadata = null;
19.
20.             lastQueryIsOnDupKeyUpdate = false;
21.
22.             if (retrieveGeneratedKeys) {
23.                 lastQueryIsOnDupKeyUpdate = containsOnDuplicateKeyUpdateInSQL();
24.             }
25.
26.             boolean doStreaming = createStreamingResultSet();
27.
28.             clearWarnings();
29.
30.             // Adjust net_write_timeout to a higher value if we're
31.             // streaming result sets. More often than not, someone runs into
32.             // an issue where they blow net_write_timeout when using this
33.             // feature, and if they're willing to hold a result set open
34.             // for 30 seconds or more, one more round-trip isn't going to hurt
35.             //
36.             // This is reset by RowDataDynamic.close().
37.
38.             if (doStreaming
39.                 && this.connection.getNetTimeoutForStreamingResults() > 0) {
40.                 executeSimpleNonQuery(locallyScopedConn,
41.                     "SET net_write_timeout="
42.                     + this.connection
43.                     .getNetTimeoutForStreamingResults());
44.             }
45.
46.             this.batchedGeneratedKeys = null;
47.
48.             Buffer sendPacket = fillSendPacket();
```

```

46.         String oldCatalog = null;
47.
48.         if (!locallyScopedConn.getCatalog().equals(this.currentCatalog)) {
49.             oldCatalog = locallyScopedConn.getCatalog();
50.             locallyScopedConn.setCatalog(this.currentCatalog);
51.         }
52.
53.         //
54.         // Check if we have cached metadata for this query...
55.         //
56.         if (locallyScopedConn.getCacheResultSetMetadata()) {
57.             cachedMetadata =
locallyScopedConn.getCachedMetaData(this.originalSql);
58.         }
59.
60.         Field[] metadataFromCache = null;
61.
62.         if (cachedMetadata != null) {
63.             metadataFromCache = cachedMetadata.fields;
64.         }
65.
66.         boolean oldInfoMsgState = false;
67.
68.         if (this.retrieveGeneratedKeys) {
69.             oldInfoMsgState = locallyScopedConn.isReadInfoMsgEnabled();
70.             locallyScopedConn.setReadInfoMsgEnabled(true);
71.         }
72.
73.         // If there isn't a limit clause in the SQL
74.         // then limit the number of rows to return in
75.         // an efficient manner. Only do this if
76.         // setMaxRows() hasn't been used on any Statements
77.         // generated from the current Connection (saves
78.         // a query, and network traffic).
79.         //
80.         // Only apply max_rows to selects
81.         //
82.         if (locallyScopedConn.useMaxRows()) {
83.             int rowLimit = -1;
84.
85.             if (this.firstCharOfStmt == 'S') {
86.                 if (this.hasLimitClause) {
87.                     rowLimit = this.maxRows;
88.                 } else {
89.                     if (this.maxRows <= 0) {
90.                         executeSimpleNonQuery(locallyScopedConn,
91.                                                 "SET SQL_SELECT_LIMIT=DEFAULT");
92.                     } else {
93.                         executeSimpleNonQuery(locallyScopedConn,

```

```

94.         "SET SQL_SELECT_LIMIT=" + this.maxRows);
95.     }
96. }
97. } else {
98.     executeSimpleNonQuery(locallyScopedConn,
99.         "SET SQL_SELECT_LIMIT=DEFAULT");
100. }
101.
102. // Finally, execute the query
103. rs = executeInternal(rowLimit, sendPacket,
104.     doStreaming,
105.     (this.firstCharOfStmt == 'S'), metadataFromCache, false);
106. } else {
107.     rs = executeInternal(-1, sendPacket,
108.     doStreaming,
109.     (this.firstCharOfStmt == 'S'), metadataFromCache, false);
110. }
111.
112. if (cachedMetadata != null) {
113.
114.     locallyScopedConn.initializeResultsMetadataFromCache(this.originalSql,
115.         cachedMetadata, this.results);
116.     } else {
117.         if (rs.reallyResult() &&
118. locallyScopedConn.getCacheResultSetMetadata()) {
119.
120. locallyScopedConn.initializeResultsMetadataFromCache(this.originalSql,
121.         null /* will be created */, rs);
122.     }
123. }
124.
125. if (this.retrieveGeneratedKeys) {
126.     locallyScopedConn.setReadInfoMsgEnabled(oldInfoMsgState);
127.     rs.setFirstCharOfQuery(this.firstCharOfStmt);
128. }
129.
130. if (oldCatalog != null) {
131.     locallyScopedConn.setCatalog(oldCatalog);
132. }
133.
134. if (rs != null) {
135.     this.lastInsertId = rs.getUpdateID();
136.
137.     this.results = rs;
138. }
139.
140. return ((rs != null) && rs.reallyResult());
141. }
142. }

```

4.6.6.5. 5) ResultSetHandler 处理结果集

如果有插件包装，会先走到被拦截的业务逻辑。

```
1.  @Override
2.  public <E> List<E> query(Statement statement, ResultHandler resultHandler)
   throws SQLException {
3.      ...
4.      return resultSetHandler.handleResultSets(ps);
5.  }
```

问题：怎么把ResultSet 转换成 `List<Object>` ？

ResultSetHandler 只有一个实现类：DefaultResultSetHandler。也就是执行 DefaultResultSetHandler的 handleResultSets () 方法。

首先我们会先拿到第一个结果集，如果没有配置一个查询返回多个结果集的情况，一般只有一个结果集。如果下面的这个while循环我们也不用，就是执行一次。

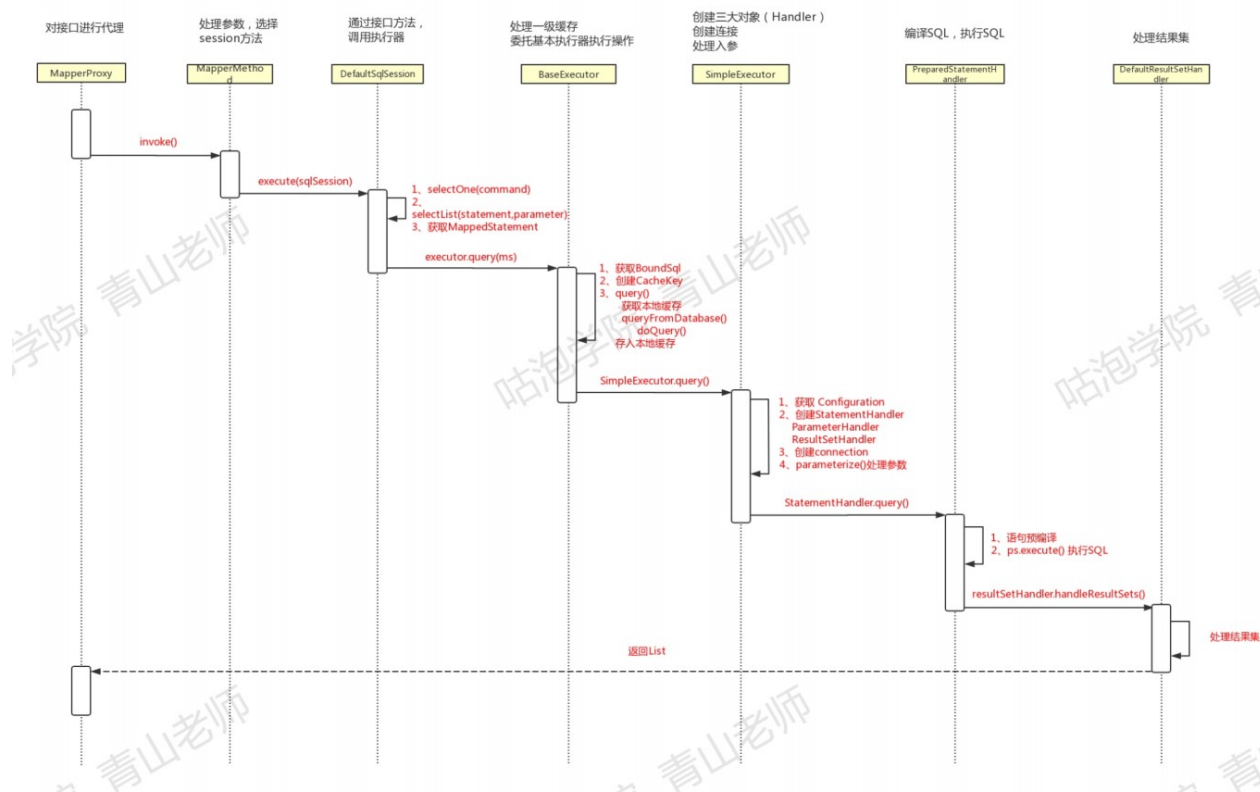
然后会调用handleResultSet()方法。

```

1.  @Override
2.  public List<Object> handleResultSets(Statement stmt) throws SQLException {
3.      ErrorContext.instance().activity("handling
results").object(mappedStatement.getId());
4.
5.      final List<Object> multipleResults = new ArrayList<>();
6.
7.      int resultSetCount = 0;
8.      ResultSetWrapper rsw = getFirstResultSet(stmt);
9.
10.     List<ResultMap> resultMaps = mappedStatement.getResultMaps();
11.     int resultMapCount = resultMaps.size();
12.     validateResultMapsCount(rsw, resultMapCount);
13.     while (rsw != null && resultMapCount > resultSetCount) {
14.         ResultMap resultMap = resultMaps.get(resultSetCount);
15.         handleResultSet(rsw, resultMap, multipleResults, null);
16.         rsw = getNextResultSet(stmt);
17.         cleanUpAfterHandlingResultSet();
18.         resultSetCount++;
19.     }
20.
21.     String[] resultSets = mappedStatement.getResultSets();
22.     if (resultSets != null) {
23.         while (rsw != null && resultSetCount < resultSets.length) {
24.             ResultMapping parentMapping =
nextResultMaps.get(resultSets[resultSetCount]);
25.             if (parentMapping != null) {
26.                 String nestedResultMapId = parentMapping.getNestedResultMapId();
27.                 ResultMap resultMap = configuration.getResultMap(nestedResultMapId);
28.                 handleResultSet(rsw, resultMap, null, parentMapping);
29.             }
30.             rsw = getNextResultSet(stmt);
31.             cleanUpAfterHandlingResultSet();
32.             resultSetCount++;
33.         }
34.     }
35.
36.     return collapseSingleResultList(multipleResults);
37. }

```

调用代理对象方法，执行SQL



4.7. MyBatis核心对象

对象	相关对象	作用
Configuration	MapperRegistry TypeAliasRegistry TypeHandlerRegistry	包含了 MyBatis 的所有的配置信息
SqlSession	SqlSessionFactory DefaultSqlSession	对操作数据库的增删改查的 API 进行了封装，提供给应用层使用
Executor	BaseExecutor SimpleExecutor BatchExecutor ReuseExecutor	MyBatis 执行器，是 MyBatis 调度的核心，负责 SQL 语句的生成和查询缓存的维护
StatementHandler	BaseStatementHandler SimpleStatementHandler PreparedStatementHandler CallableStatementHandler	封装了 JDBC Statement 操作，负责对 JDBC statement 的操作，如设置参数、将 Statement 结果集转换成 List 集合
ParameterHandler	DefaultParameterHandler	把用户传递的参数转换成 JDBC Statement 所需要的参数
ResultSetHandler	DefaultResultSetHandler	把 JDBC 返回的 ResultSet 结果集对象转换成 List 类型的集合

MapperProxy	MapperProxyFactory	触发管理类，用于代理 Mapper 接口方法
MappedStatement	SqlSource BoundSql	MappedStatement 维护了一条<select update delete insert>节点的封装，表示一条 SQL 包括了 SQL 信息、入参信息、出参信息