

[技术] MyBatis 3x

1 简介

作者：Clinton Begin（克林顿·比格）男。

MyBatis 是一个持久层框架，支持定制化 SQL、存储过程和高级映射。

它消除了绝大部分的 JDBC 代码书写，不需要手动拼接参数和检索结果。

它可以使用简单的 XML 或注解来配置和映射原生类型、接口和 Java 的 POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

MyBatis 的前世为 iBatis（该名称为 internet 和 abatis 的组合，寓意为互联网而生），iBatis 是由 Clinton Begin 在 2001 年发起的开源项目。

在 2004 年，iBatis 发布了 2.0 版本，随后 Clinton Begin 将 iBatis 献给 Apache 软件基金会（Apache Software Foundation, ASF）。

在之后的 6 年中，iBatis 在方法论、源码管理、社交、开源基础建设等方面都取得了很大的进步。

2010 年 5 月 21 日，iBatis 项目迁移到 Google Code，并更名为 MyBatis，正式投胎转世。版本从 3.0.1 一直更新到 3.2.3，稳定性得到很大提升。

2010 年 6 月 16 日，iBatis 项目被正式归入 Apache Attic，属性变为“只读”，这意味着该项目在 iBatis 时代正式结束。

2013 年 11 月 10 日，让更多的人参与到项目中，MyBatis 项目被迁移至 GitHub，而后一直发展至今。

目前最新的稳定版本为 **3.5.4**。

Mybatis 的特点：

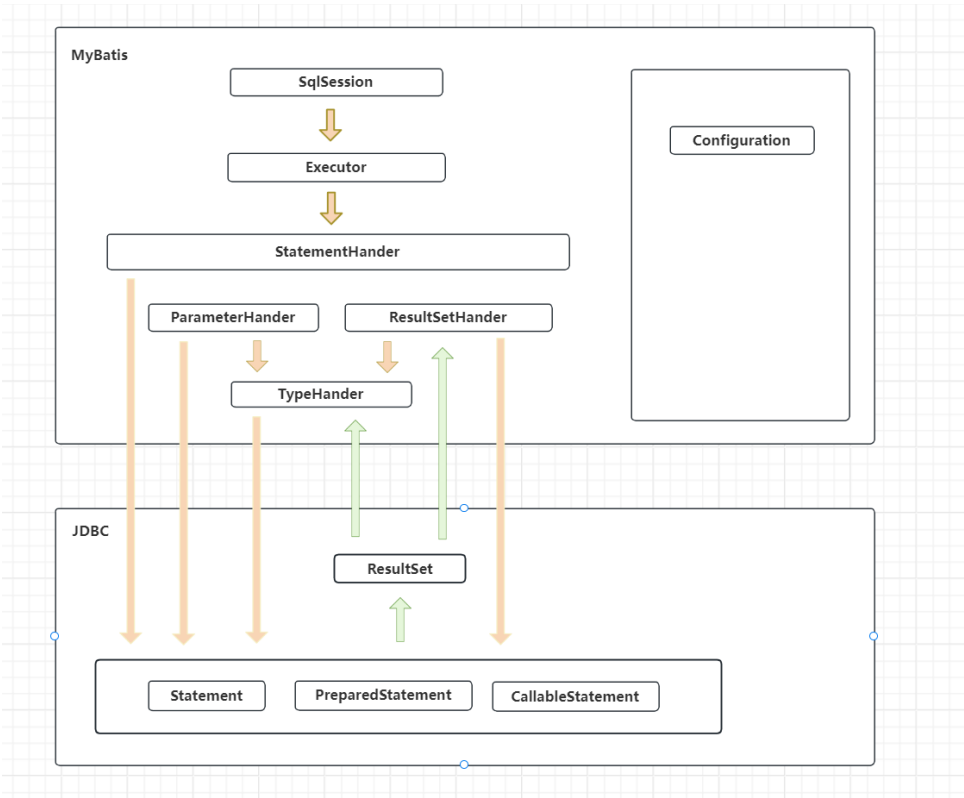
- 消除了大量的 JDBC 冗余代码

- 学习门槛低
- 能很好地与传统数据库协同工作
- 可以接受 SQL 语句
- 提供了与 Spring 框架的集成支持
- 提供了与第三方缓存类库的集成支持
- 引入了更好的性能

2 框架



核心组件



3 核心流程

JDBC 执行流程:

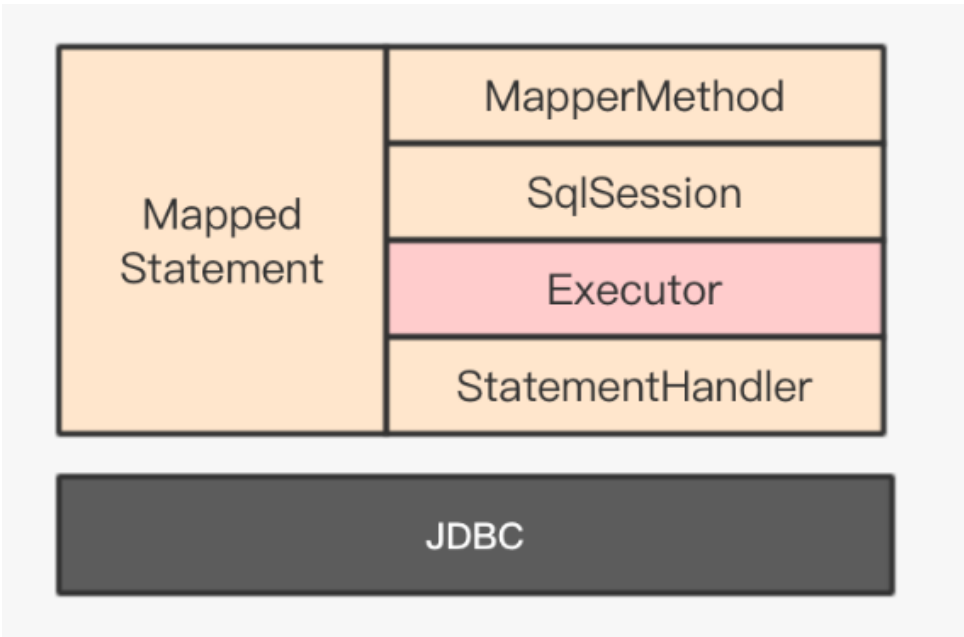
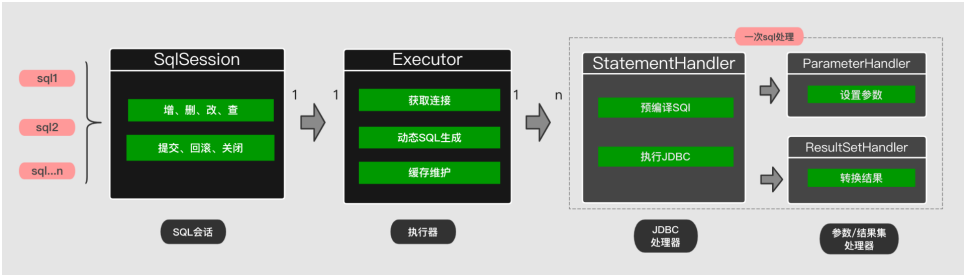


Code 示例:

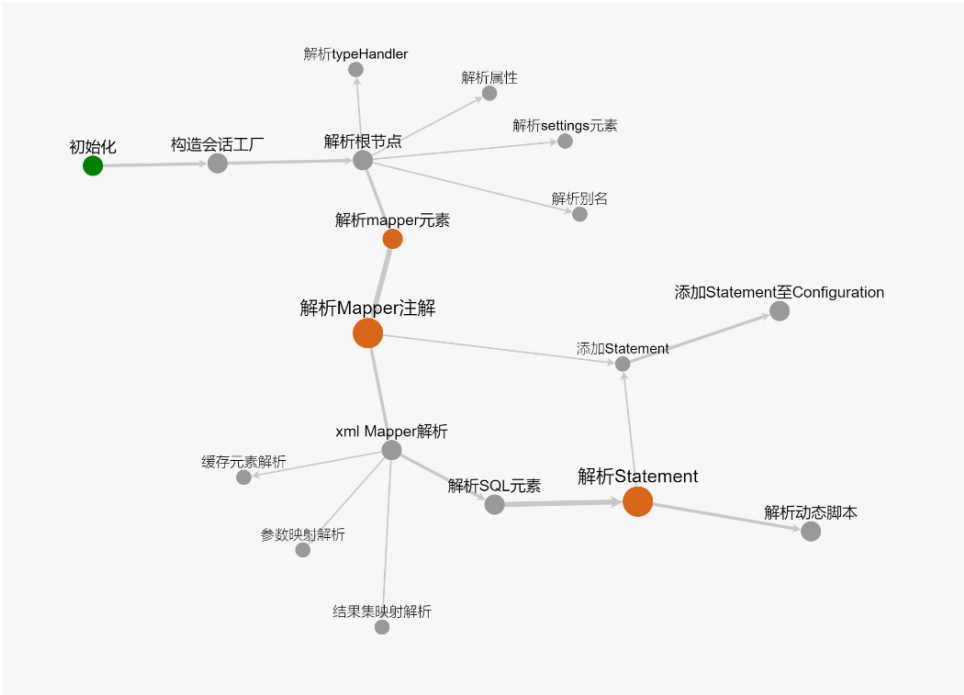
```
1
2  /** 第一步： 获取连接 */
3  Class.forName(...) //加载驱动
4  Connection connection = DriverManager
5      .getConnection(JDBC.URL,
6                     JDBC.USERNAME, JDBC.PASSWORD);
7
8  /** 第二步： 预编译SQL */
9  PreparedStatement statement = connection
10     .prepareStatement("select * from
11                        users ");
12
13  /** 第三步： 执行查询 */
14  ResultSet resultSet = statement.executeQuery();
```

```
14  /** 第四步： 读取结果 */
15  readResultSet(resultSet);
16
```

Mybatis整体流程



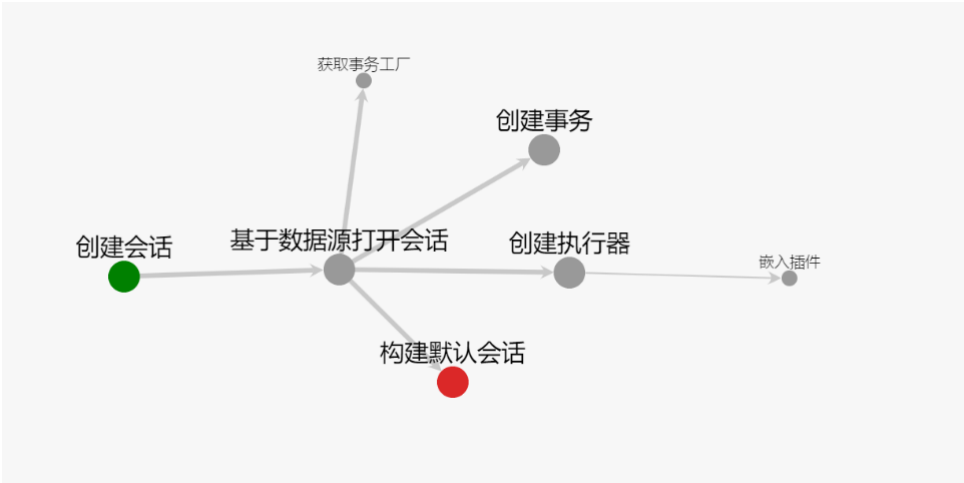
3.1 初始化



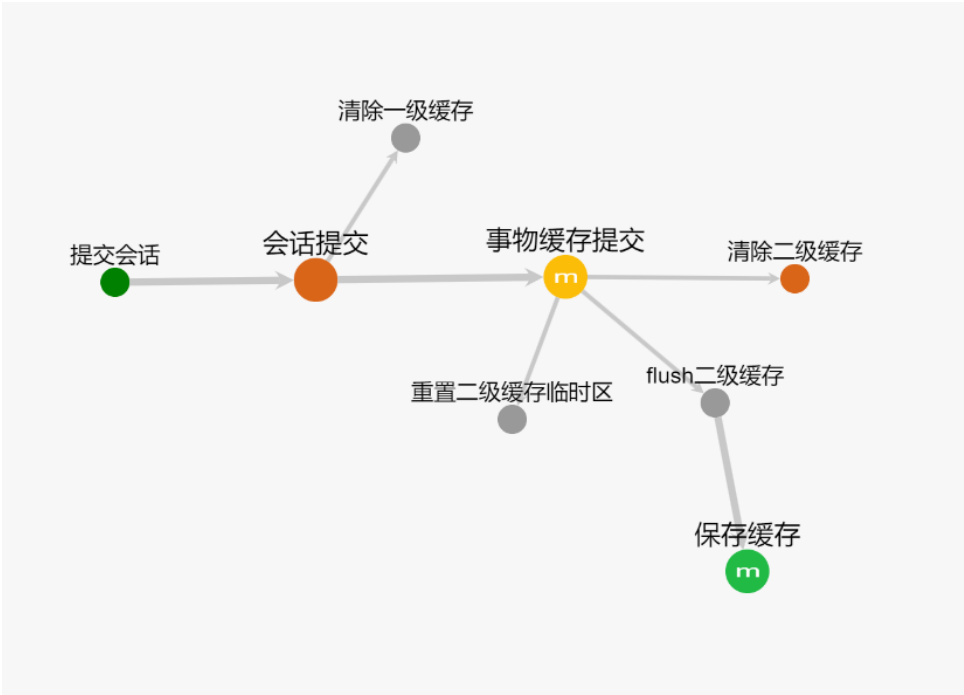
3.2 SqlSession（会话）

SqlSession 是myBatis的门面(门面模式设计)，核心作用是为用户提供API。API包括增、删、改、查以及提交、关闭等。其自身是没有能力处理这些请求的，所以内部会包含一个唯一的执行器 Executor，所有请求都会交给执行器来处理。

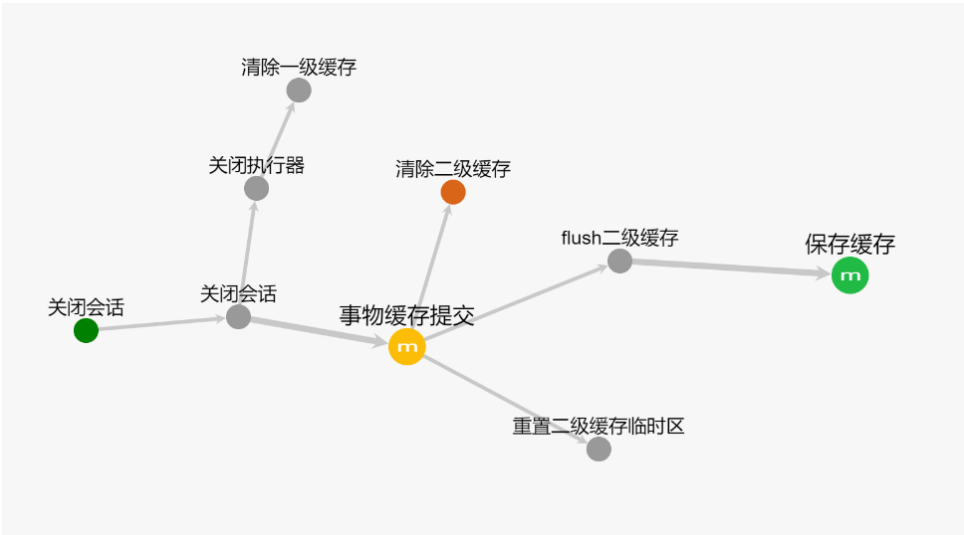
创建



提交

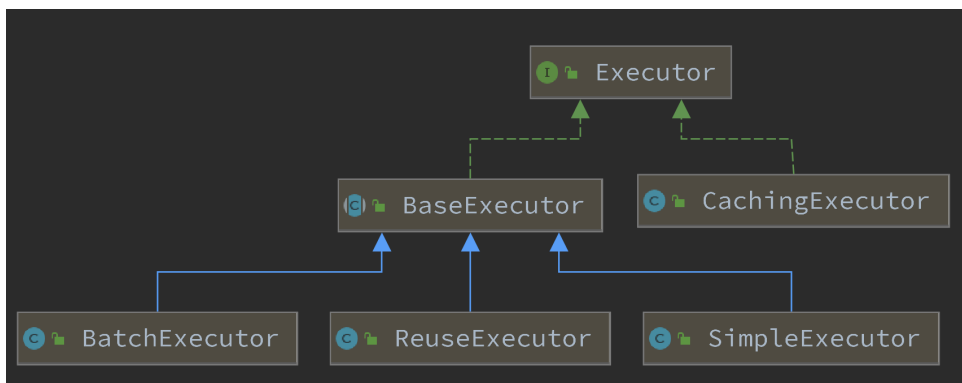


关闭



3.3 Executor（执行器）

Executor是一个大管家，核心功能包括：缓存维护、获取动态SQL、获取连接、以及最终的JDBC调用等。在图中所有蓝色节点全部都是在Executor中完成。

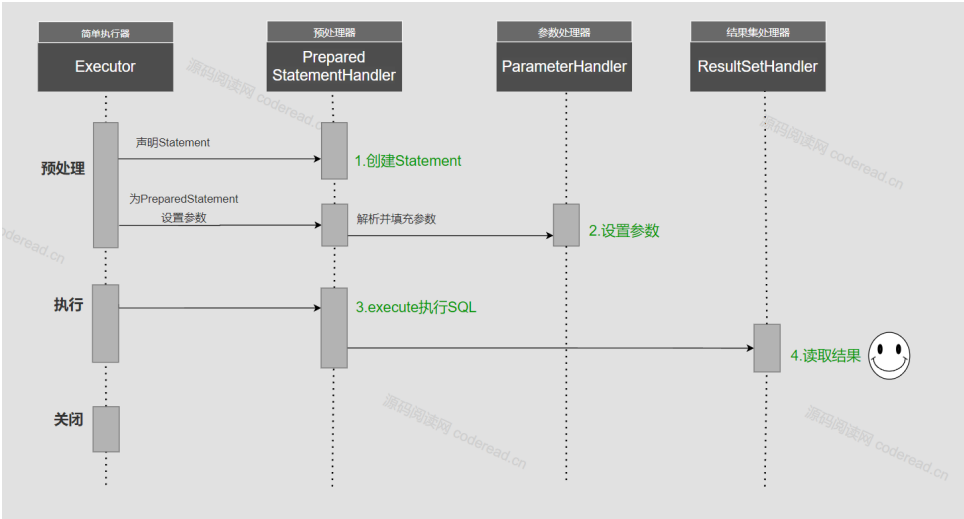


1. **BaseExecutor**: 执行器基类，基础方法都放置在此。
2. **SimpleExecutor**: 默认执行器
3. **ReuseExecutor**: 重用执行器，相同sql的statement 将会被缓存已重复利用
4. **BatchExecutor**: 批处理执行器，基于 JDBC 的 `addBatch`、`executeBatch` 功能，并且在当前 sql 和上一条 sql 完全一样的时候，重用 Statement，在调用 `doFlushStatements` 的时候，将数据刷新到数据库
5. **CachingExecutor**: 缓存执行器，装饰器模式，在开启二级缓存的时候。会在上面三种执行器的外面包上 **CachingExecutor**

Executor内部还会包含若干个组件：

- 缓存维护：cache
- 获取连接：Transaction
- 获取动态SQL：SqlSource
- 调用JDBC：StatementHandler

上述组件中前三个和Executor是1对1关系，只有StatementHandler是1对多。每执行一次SQL 就会构造一个新的StatementHandler。想必你也能猜出StatementHandler的作用就是专门和JDBC打交道，执行SQL。



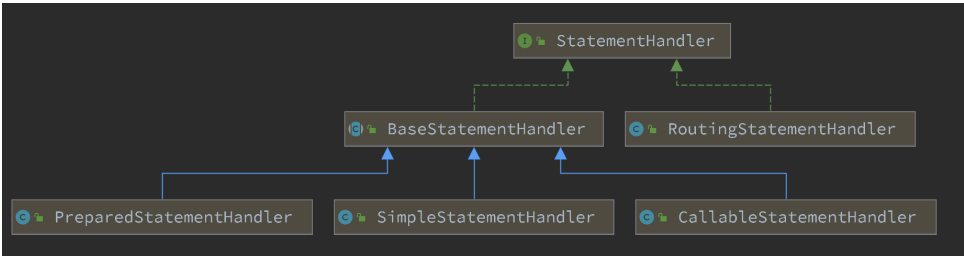
执行过程:

Executor执行的时候并不直接拿着JDBC的API一顿操作，而是由StatementHandler，ResultSetHandler 两位做具体的操作。

StatementHandler

用于获取预处理器，共有三种类型。通过statementType= "STATEMENT|PREPARED|CALLABLE" 可分别进行指定。

- PreparedStatementHandler：带预处理的执行器
- CallableStatementHandler：存储过程执行器
- SimpleStatementHandler：基于Sql执行器



ResultSetHandler

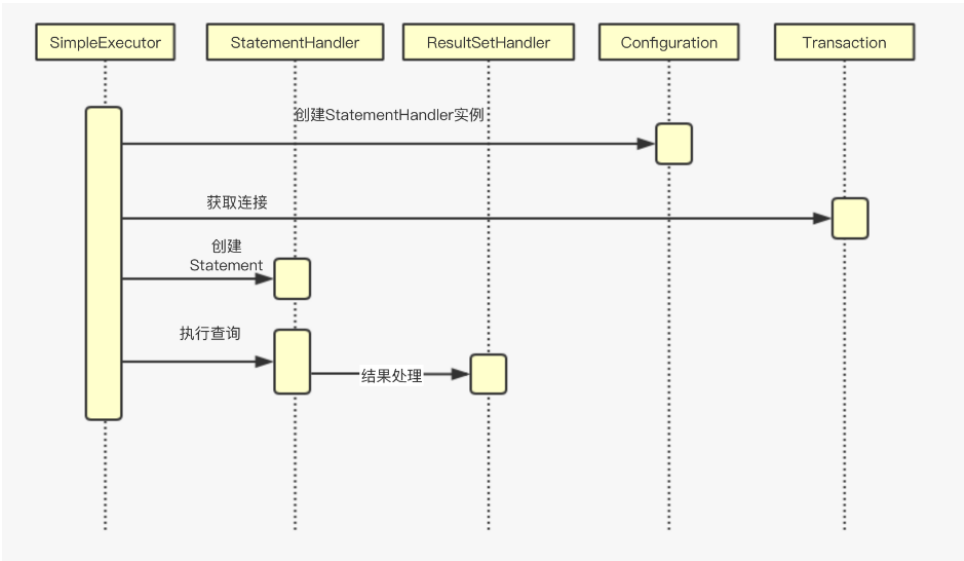
用于处理和封装返回结果。可在SqlSession中查询时自行定义ResultSetHandler。

说明:

1. 通过Configuration获取StatementHandler实例（由statementType 决定）。
2. 通过事务获取连接
3. 创建JDBC Statement对像
4. 执行 JDBC Statement execute

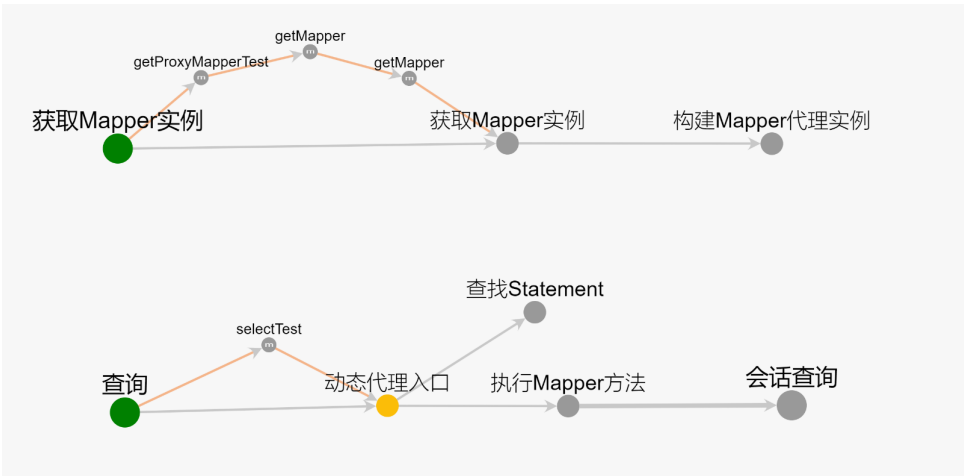
5. 处理返回结果

时序图：

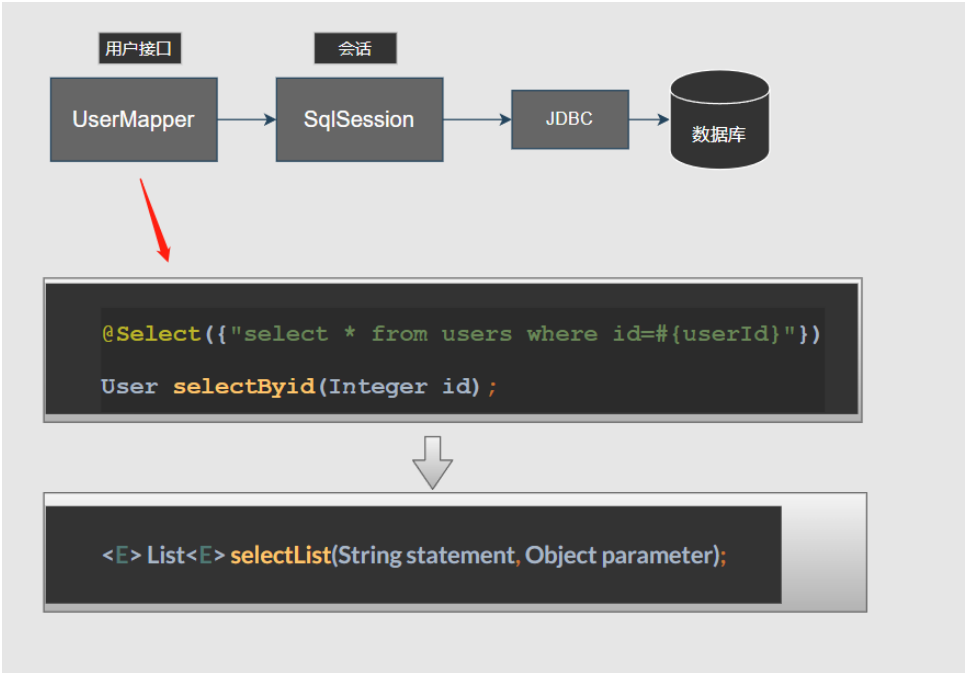


3.4 Mapper的动态代理

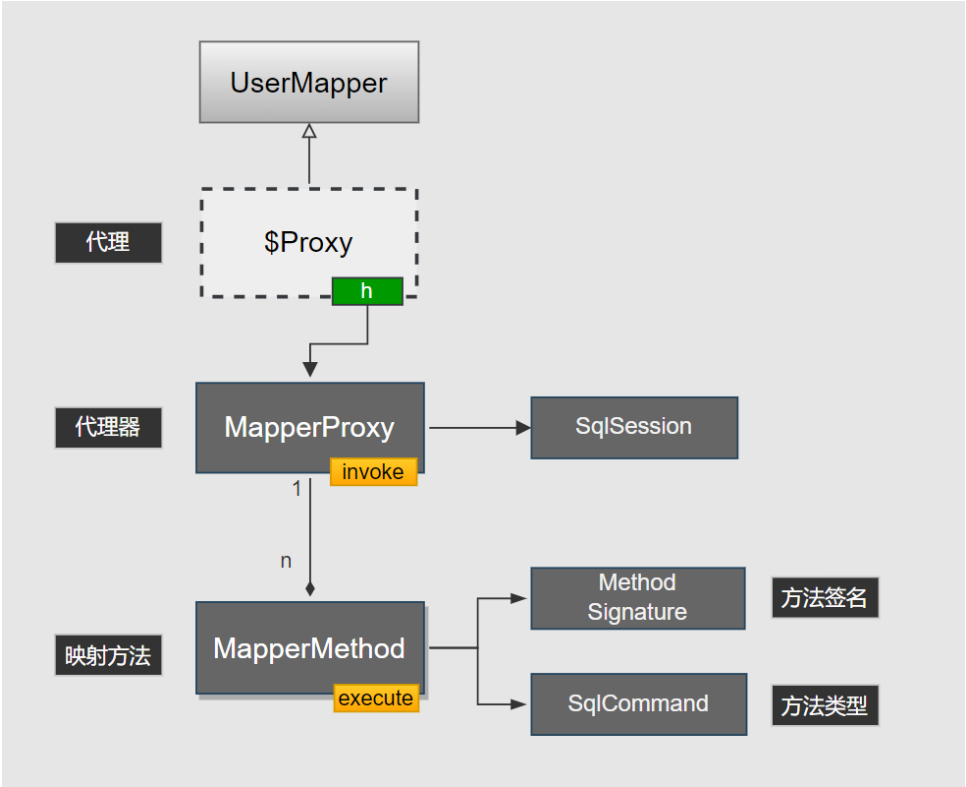
代理过程



查询用户的案例



Mapper结构



MapperProxy#invoke

```

/**
 * @author Clinton Begin
 * @author Eduardo Macarron
 */
public class MapperProxy<T> implements InvocationHandler, Serializable {

    private static final long serialVersionUID = -6424540398559729838L;
    private final SqlSession sqlSession;
    private final Class<T> mapperInterface;
    private final Map<Method, MapperMethod> methodCache;

    public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method, MapperMethod> methodCache) {
        this.sqlSession = sqlSession;
        this.mapperInterface = mapperInterface;
        this.methodCache = methodCache;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        try {
            if (Object.class.equals(method.getDeclaringClass())) {
                return method.invoke(this, args);
            } else if (isDefaultMethod(method)) {
                return invokeDefaultMethod(proxy, method, args);
            }
        } catch (Throwable t) {
            throw ExceptionUtil.unwrapThrowable(t);
        }

        final MapperMethod mapperMethod = cachedMapperMethod(method);
        return mapperMethod.execute(sqlSession, args);
    }

    private MapperMethod cachedMapperMethod(Method method) {
        return methodCache.computeIfAbsent(method, b -> new MapperMethod(mapperInterface, method, sqlSession.getConfiguration()));
    }
}

```

MapperMethod#invoke

```

public class MapperMethod {

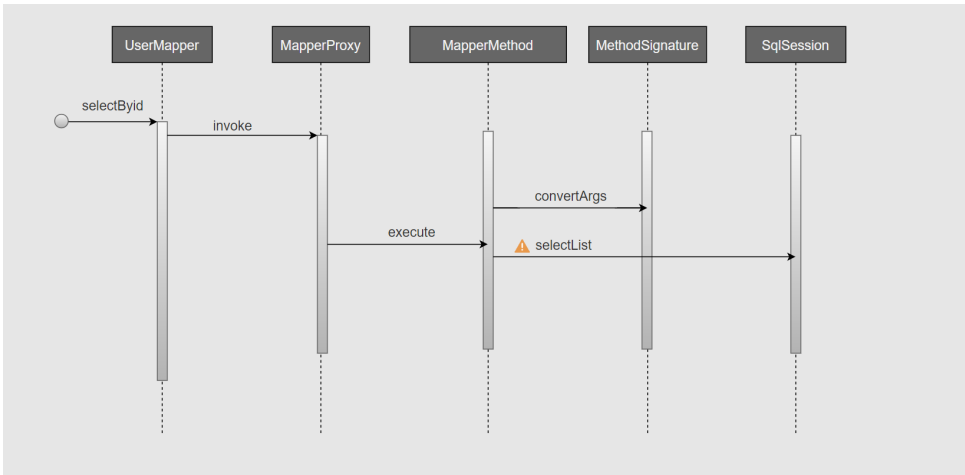
    private final SqlCommand command;
    private final MethodSignature method;

    public MapperMethod(Class<?> mapperInterface, Method method, Configuration config) {
        this.command = new SqlCommand(config, mapperInterface, method);
        this.method = new MethodSignature(config, mapperInterface, method);
    }

    public Object execute(SqlSession sqlSession, Object[] args) {
        Object result;
        switch (command.getType()) {
            case INSERT: {
                Object param = method.convertArgsToSqlCommandParam(args);
                result = rowCountResult(sqlSession.insert(command.getName(), param));
                break;
            }
            case UPDATE: {
                Object param = method.convertArgsToSqlCommandParam(args);
                result = rowCountResult(sqlSession.update(command.getName(), param));
                break;
            }
            case DELETE: {
                Object param = method.convertArgsToSqlCommandParam(args);
                result = rowCountResult(sqlSession.delete(command.getName(), param));
                break;
            }
            case SELECT:
                if (method.returnsVoid() && method.hasResultHandler()) {
                    executeWithResultHandler(sqlSession, args);
                    result = null;
                } else if (method.returnsMany()) {
                    result = executeForMany(sqlSession, args);
                } else if (method.returnsMap()) {
                    result = executeForMap(sqlSession, args);
                } else if (method.returnsCursor()) {
                    result = executeForCursor(sqlSession, args);
                } else {
                    Object param = method.convertArgsToSqlCommandParam(args);
                    result = sqlSession.selectOne(command.getName(), param);
                    if (method.returnsOptional()) {
                        && (result == null || !method.getReturnType().equals(result.getClass())) {
                            result = Optional.ofNullable(result);
                        }
                    }
                    break;
                }
            case FLUSH:
                result = sqlSession.flushStatements();
                break;
            default:
                throw new BindingException("Unknown execution method for: " + command.getName());
        }
        if (result == null && method.getReturnType().isPrimitive() && !method.returnsVoid()) {
            throw new BindingException("Mapper method '" + command.getName()
                + " attempted to return null from a method with a primitive return type (" + method.getReturnType() + ").");
        }
        return result;
    }
}

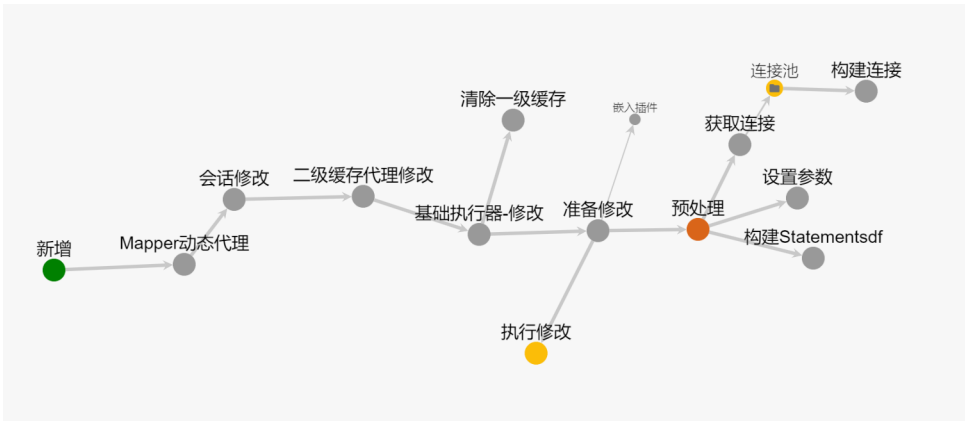
```

完整执行流程

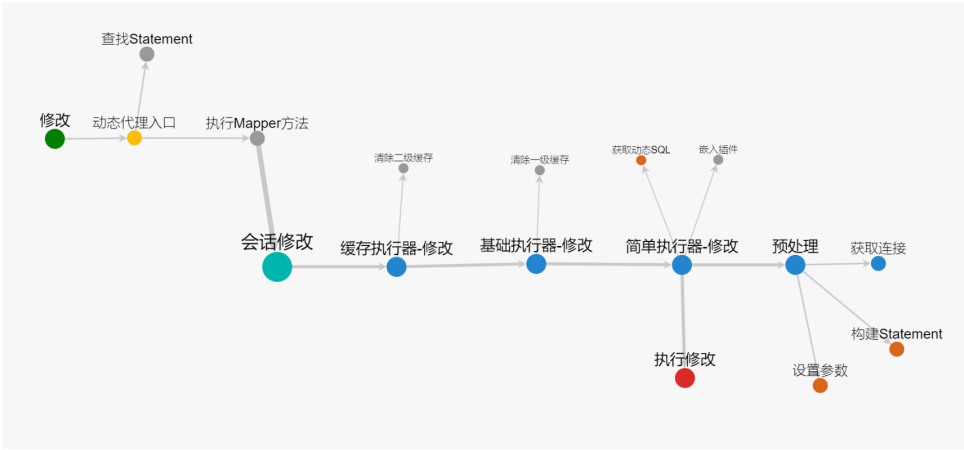


- 1 调用SqlSession方法，基于以逻辑决定调用具体哪个方法：
- 2 1.MapperStatement类型，即增删改查
- 3 2.如果是查询会受到以下值的影响：
- 4 a.【返回结果类型】void、List、Object、
- 5 Cursor、Map
- 6 b.【参数类型】参数中包含分页对象(RowBounds)、自
- 7 定义结果处理器(ResultSetHandler)

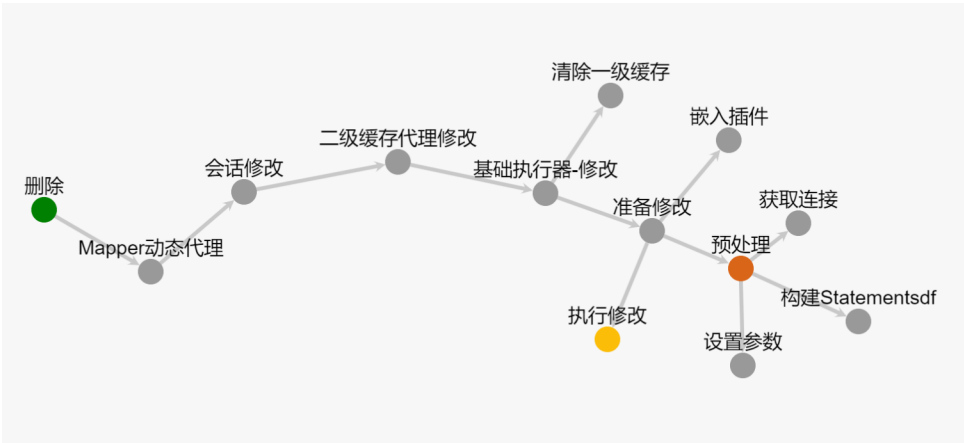
3.5 Insert



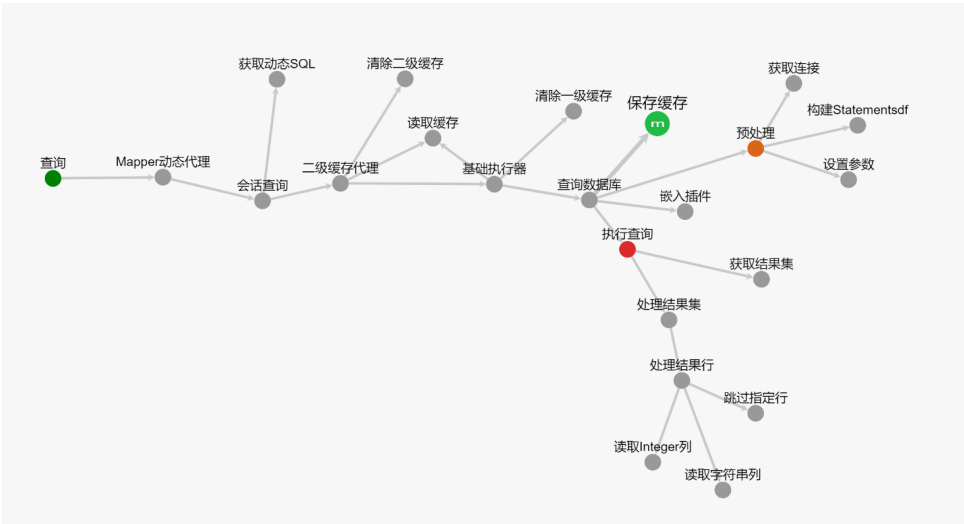
3.6 Update



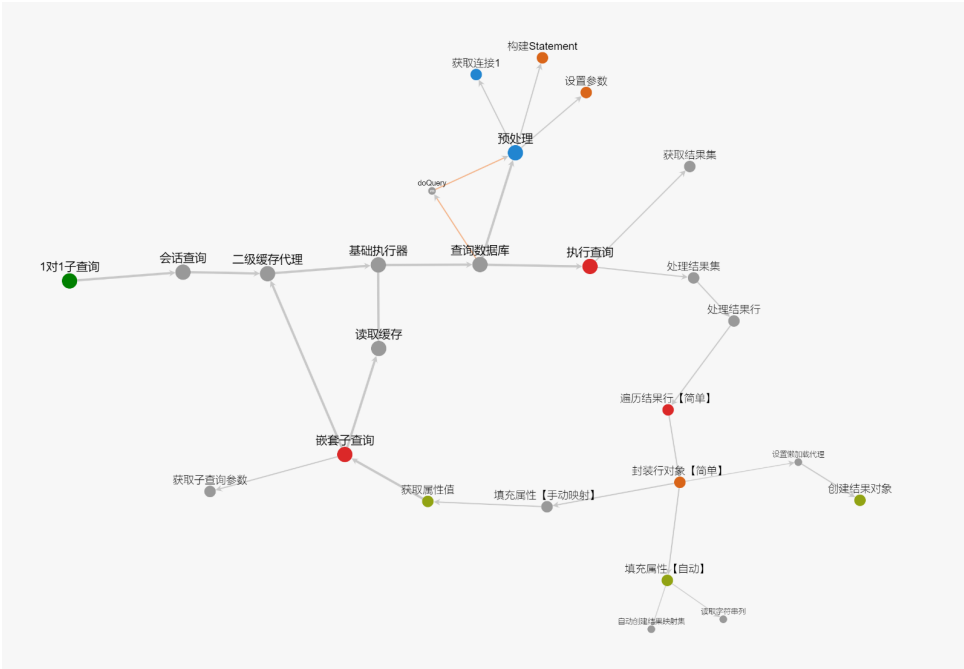
3.7 Delete



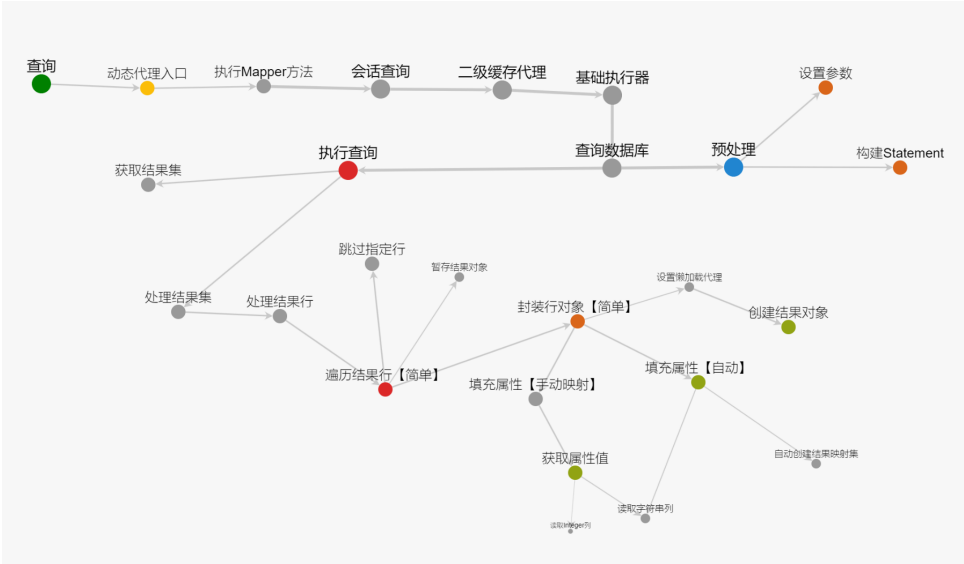
3.8 Select



嵌套子查询



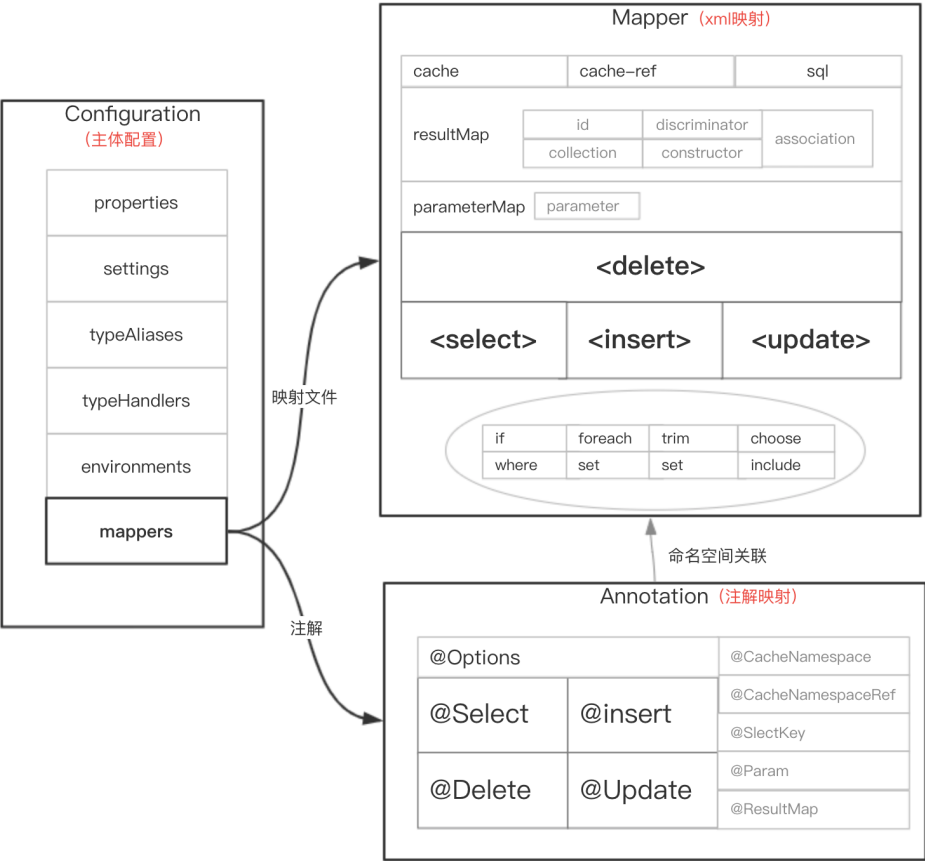
结果集处理



4 Configuration 配置

4..1 结构

主体配置(Configuration)、Xml映射 (Mapper)、注解映射 (Annoation)。

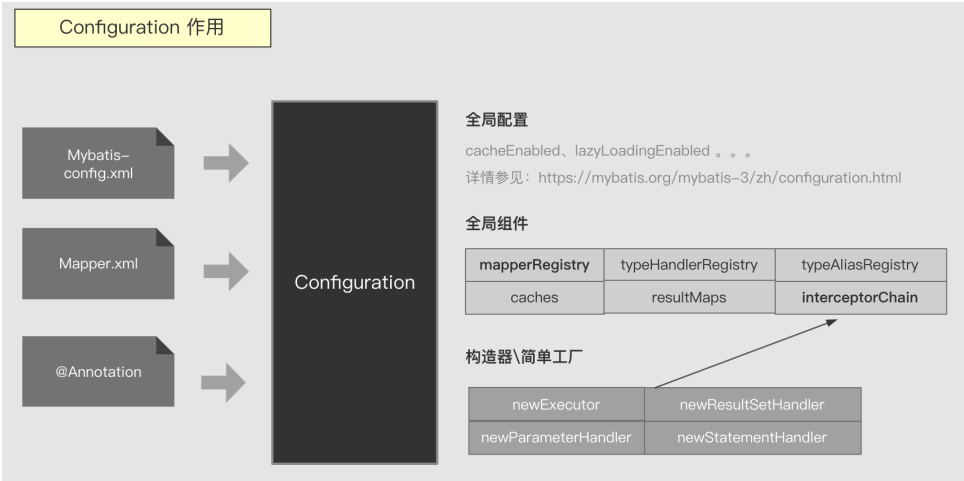


4.2 主要作用

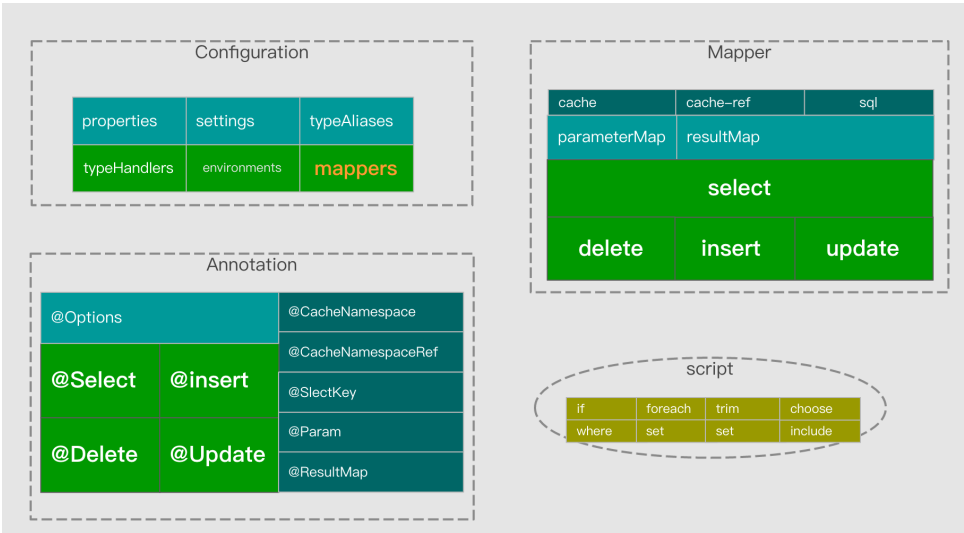
- 存储全局配置信息，其来源于settings（设置）
- 初始化并维护全局基础组件
 - typeAliases（类型别名）
 - typeHandlers（类型处理器）
 - plugins（插件）
 - environments（环境配置）
 - cache(二级缓存空间)
- 初始化并维护MappedStatement
- 组件构造器,并基于插件进行增强
 - newExecutor（执行器）
 - newStatementHandler（JDBC处理器）
 - newResultSetHandler（结果集处理器）
 - newParameterHandler（参数处理器）

4.3 配置来源

- 1. Mybatis-config.xml： 启动文件，全局配置、全局组件都是来源于此。
- 2. Mapper.xml ： SQL映射(MappedStatement) \结果集映射(ResultMapper)都来源于此。
- 3. @Annotation ： SQL映射与结果集映射的另一种表达形式。

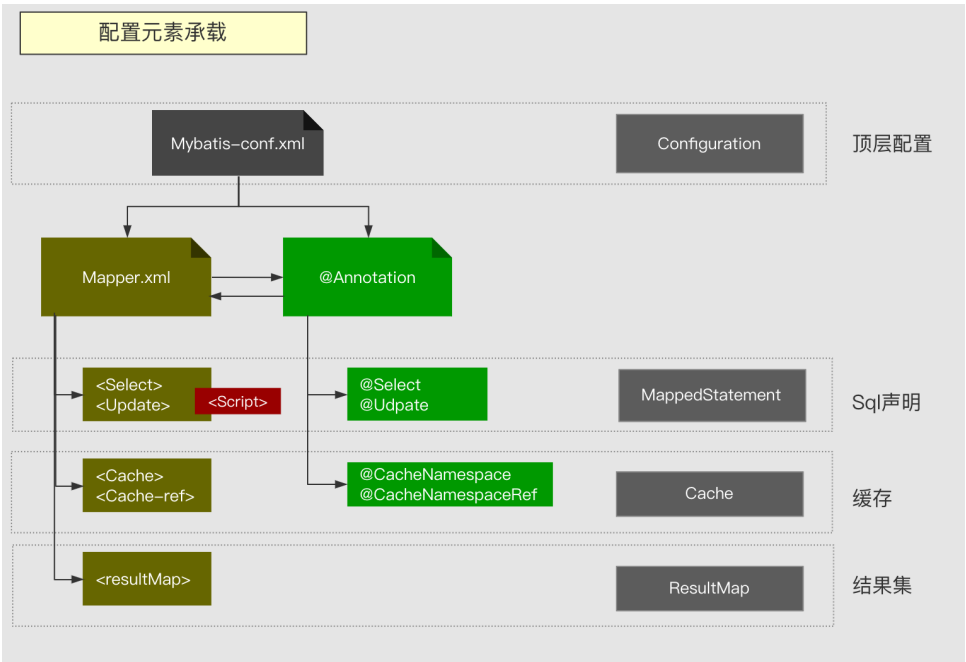


配置元素：



元素承载：

- 1. 全配置(config.xml) 由Configuration对象属性承载
- 2. sql映射<select|insert...> 或@Select 等由MappedStatement对象承载
- 3. 缓存<cache..> 或@CacheNamespace 由Cache对象承载
- 4. 结果集映射 由ResultMap 对象承载



4.4 解析

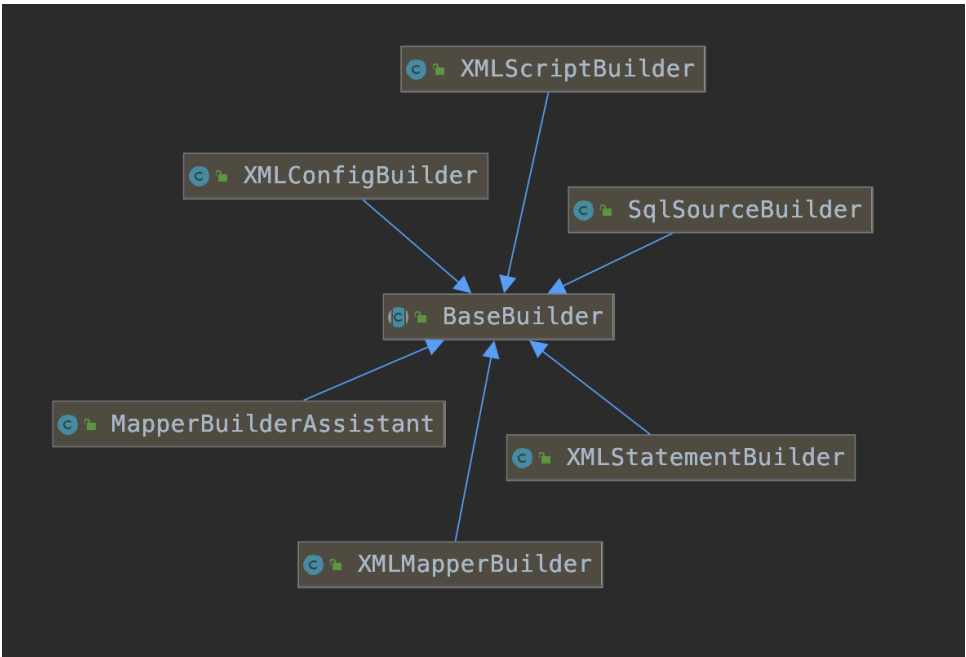
配置文件 >> 解析器 >> 对象

解析过程就是将配置转换成模型的过程

具体解析过程是通过`XmlConfigBuilder` 作为入口，然后就是`MapperAnnotationBuilder`、`XMLMapperBuilder`等解析器层层递进完成。

解析器的基类是`BaseBuilder` 其内部包含全局的`configuration` 对象，这么做的用意是所有要解析的组件最后都要集中归属至`configuration`

类图结构：

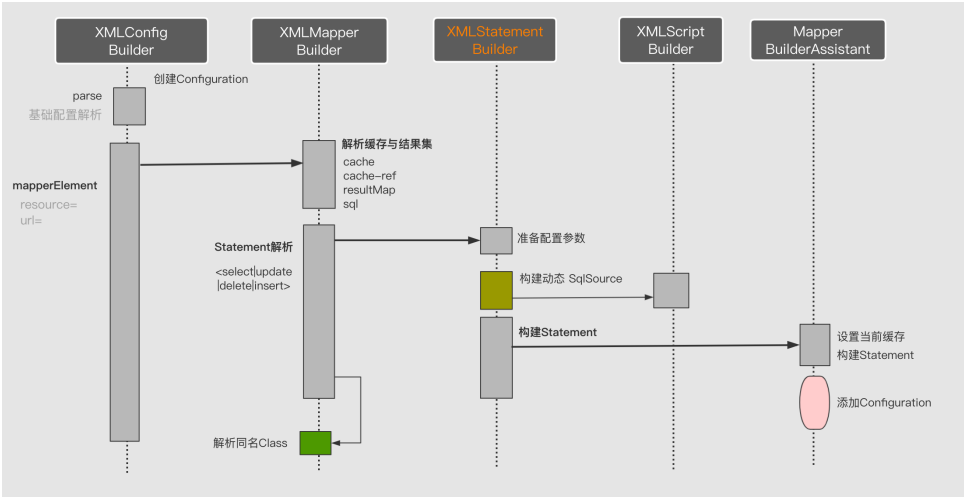


说明	文件	解析器	对像
主配置文件	Config.xml	XMLConfigBuilder	Configuration
映射文件	Mapper.xml	XMLMapperBuilder	ParameterMap\ResultMap\Cache
Statement元素	<select update..>	XMLStatementBuilder	MappedStatement
动态Sql运算元素	<if foreach trim...>	XMLScriptBuilder	BoundSql
注解元素	<@Select @Update..>	MapperAnnotationBuilder	MappedStatement

XML文件解析流程

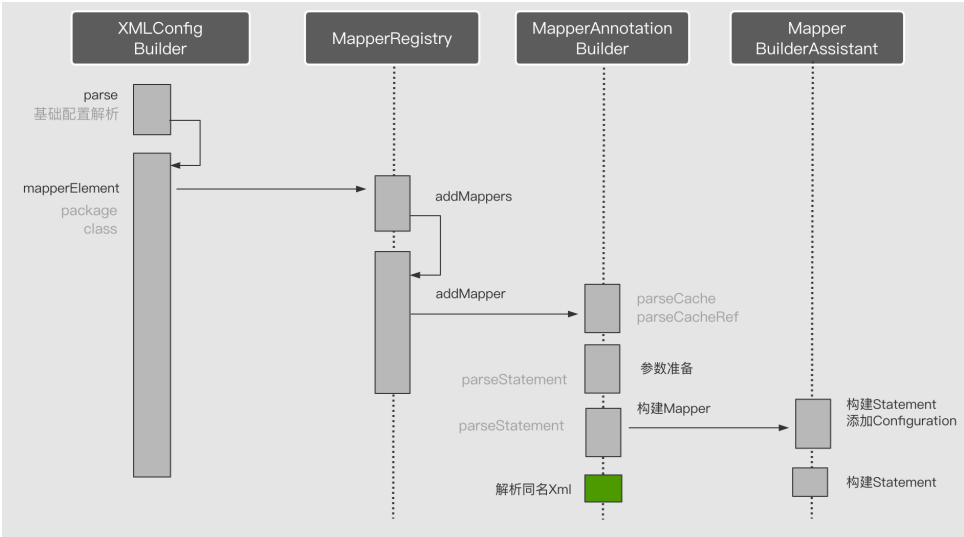
整体解析流程是从XmlConfigBuilder 开始，然后逐步向内解析，直到解析完所有节点。

通过一个MappedStatement 解析过程为例：

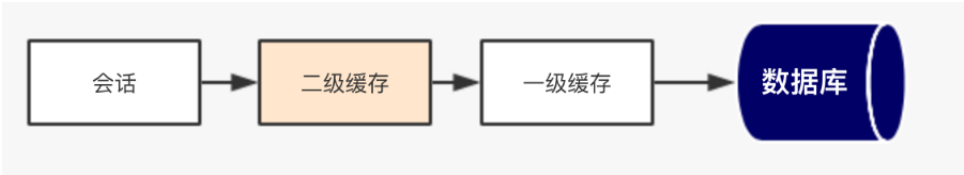


注解配置解析流程

注解解析底层实现是通过反射获取Mapper接口当中注解元素实现。有两种方式一种是直接指定接口名，一种是指定包名然后自动扫描包下所有的接口类。这些逻辑均由Mapper注册器(MapperRegistry)实现。其接收一个接口类参数，并基于该参数创建针对该接口的动态代理工厂，然后解析内部方法注解生成每个MapperStatement 最后添加至Configuration 完成解析。



5 缓存



缓存是MyBatis中非常重要的特性。在应用程序和数据库都是单节点的情况下，合理使用缓存能够减少数据库IO，显著提升系统性能。但是在分布式环境下，如果使用不当，则可能会带来数据一致性问题。

缓存总共有两级：

一级缓存：会话级缓存，**基于SqlSession实现**。在同一会话内如果有两次相同的查询（Sql和参数均相同），那么第二次就会命中缓存。一级缓存通过会话进行存储，当会话关闭，缓存也就没有了。此外如果会话进行了修改（增删改）操作，缓存也会被清空。

二级缓存：应用级的缓存，**基于Mapper实现**，即作用于整个应用的生命周期。相对一级缓存会有更高的命中率。所以在顺序上是先访问二级然后在是一级和数据库。

两个配置：useCache 和 flushCache：

1、select语句

flushCache（默认false），任何时候语句被调用，都不会去清空本地缓存和二级缓存。

useCache（默认true），将本条语句的结果进行二级缓存。

2、insert/update/delete语句

flushCache（默认true）。任何时候语句被调用，都会导致本地缓存和二级缓存被清空。

useCache 不支持。

案例：

```
1 <select id="save" parameterType="XX" flushCache="true" useCache="false">
2 </select>
```

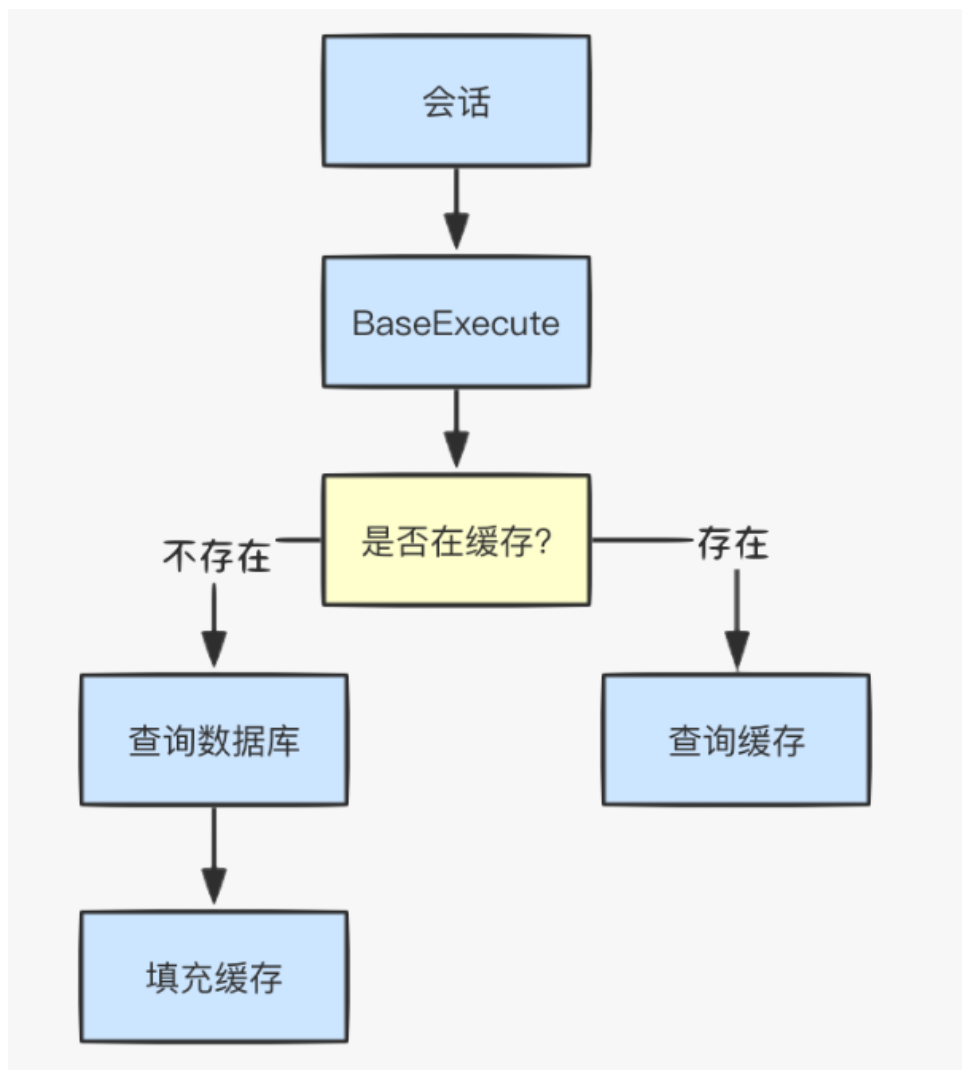
默认是开启一级缓存的，而且不可以关闭。

useCache=false 只能关闭二级缓存，不能关闭一级缓存。

如果一定要关闭一级缓存只能在查询中配置 flushCache=true

5.1 一级缓存

SqlSession会话级缓存，默认开启。基于SqlSession实现。



命中条件（所有条件And）

1. 相同的statement id
2. 相同的Session
3. 相同的Sql与参数
4. 返回行范围相同

清空场景

会话在以下情况会清空一级缓存：

1. 执行update,只要会话中执行了增删改就会被清空，并且跟sql、参数、statement id无关。
2. 手动清空，即执行SqlSession.clearCache() 方法。
3. 查询清空，即配置了 flushCache= true,查询前会清空全部缓存。
4. 提交，回滚清空。

特点：

生命周期短暂，缓在命中率较低

为什么不能关闭?

MyBatis核心开发人员做出了解释：MyBatis的一些关键特性（例如通过<association>和<collection>建立级联映射、避免循环引用（circular references）、加速重复嵌套查询等）都是基于MyBatis一级缓存实现的，而且MyBatis结果集映射相关代码重度依赖CacheKey，所以目前MyBatis不支持关闭一级缓存。

控制：用户只能控制缓存的级别，并不能关闭。

配置参数 localCacheScope：控制缓存的级别。

取值：

1、SESSION

缓存对整个SqlSession有效，只有执行DML语句（更新语句）时，缓存才会被清除

2、STATEMENT

缓存仅对当前执行的语句有效，当语句执行完毕后，缓存就会被清空。

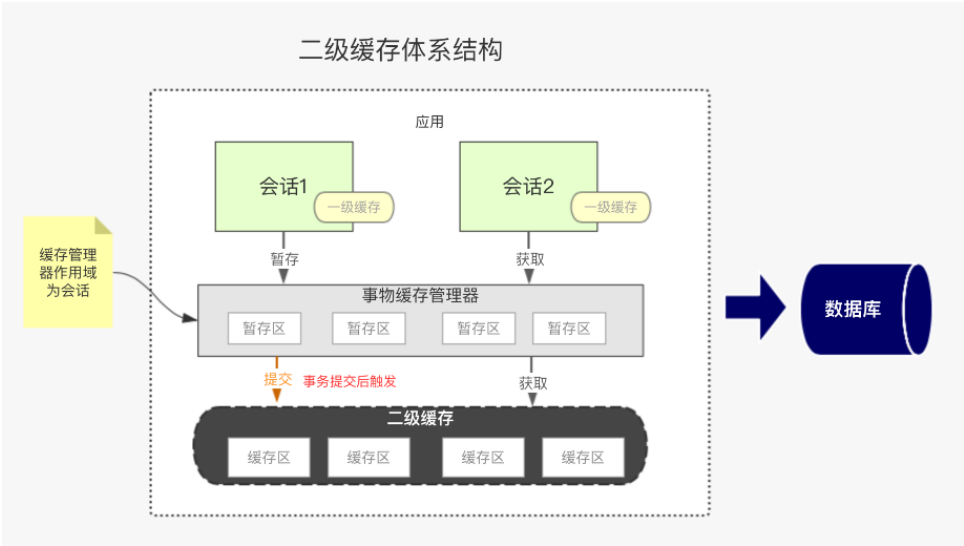
Spring下一级缓存失效?

原因在于Spring每次执行Sql请求都会通过MyBatis获取一个新的SqlSession自然就不会命中一级缓存了。解决办法是给服务方法添加事物。通常只有增删改操作会添加事物，而如果是纯查询的我们会忽略事物。事物对于查询也是有必要的。

回想一下，你的程序中，是怎么获取SqlSession的?

5.2 二级缓存

应用级的缓存，即作用于整个应用的生命周期。默认情况下关闭，需要通过设置cacheEnabled参数值为true来开启二级缓存。



暂存区 TransactionalCache

暂时存放待缓存的数据区域，和缓存区是一一对应的。如果会话会涉及多个二级缓存的访问，那么对应暂存区也会有多个。暂存区生命周期与会话保持一致。

缓存区

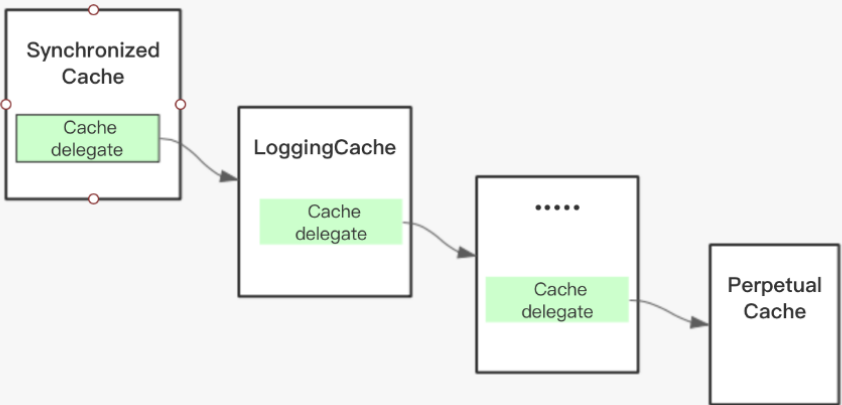
缓存区是通过Mapper声明而获得，默认每个Mapper都有独立的缓存区。其作用是真正存放数据和实现缓存的业务逻辑。如序列化、防止缓存穿透、缓存效期等。

缓存区的设计模式

在设计上采用的是装饰器模式。即不同的功能由不同缓存装饰器实现。下表为装饰器类和对应的功能。

装饰器	描述
SynchronizedCache	同步锁，用于保证对指定缓存区的操作都是同步的
LoggingCache	统计器，记录缓存命中率
BlockingCache	阻塞器，基于key加锁，防止缓存穿透
ScheduledCache	时效检查，用于验证缓存有效器，并清除无效数据
LruCache	溢出算法，淘汰闲置最久的缓存。
FifoCache	溢出算法，淘汰加入时间最久的缓存
WeakCache	溢出算法，基于java弱引用规则淘汰缓存
SoftCache	溢出算法，基于java软引用规则淘汰缓存
PerpetualCache	实际存储，内部采用HashMap进行存储。

这些装饰器是如何组织起来的呢？查看源码可得知，每个装饰器都会通过属性引用下一个装饰器，从而组成一个链条。缓存逻辑基于链条进行传递。



命中条件（所有条件And）：

- 1. 相同的statement id
- 2. 相同的Sql与参数
- 3. 返回行范围相同
- 4. 没有使用ResultHandler来自定义返回数据
- 5. 没有配置UseCache=false 来关闭缓存
- 6. 没有配置FlushCache=true 来清空缓存
- 7. 在调用存储过程中不能使用出参，即Parameter中 mode=out|inout

缓存写入时机

二级缓存是在事物提交或会话关闭之后才会触发缓存写入。

这么做其实也好理解，因为二级缓存是跨会话的，如果没有提交就写入，如果事物最后回滚，肯定导致别的会话脏读。（脏读：读取到没有提交的数据）

缓存更新与关闭

二级缓存的数据默认作用于整个应用的生命周期，

那怎么保证数据的一至性？有以下几种方式：

1. 默认的update操作会清空该namespace下的缓存（可设定flushCache=false 来禁止）。
2. 设定缓存的失效时间（默认一小时）。
3. 将指定查询的缓存关闭即设置useCache=false。
4. 为指定Statement设定 flushCache=true清空缓存

缓存引用

不同的namespace有着独立的缓存容器，只有该namespace下的statement才能访问该缓存。但表与表之间是存在关联的。而对应的Mapper又是独立的。这时我们就可以通过缓存引用，让多个Mapper共享一个缓存。具体做法是设定@CacheNamespaceRef 与 指定namespace 值就可以。

同时使用注解与xml映射文件时，虽然它们namespace相同但一样不能共享缓存，这就必须一方设定缓存，另一方引用才可以。

5.3 二级缓存（第三方引入）

MyBatis除了提供内置的一级缓存和二级缓存外，还支持使用第三方缓存（例如Redis、Ehcache）作为二级缓存。


MyBatis官方提供了一个mybatis-redis模块，该模块用于整合Redis作为二级缓存。

```
<dependencies>
  ...
  <dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-redis</artifactId>
    <version>1.0.0-beta2</version>
  </dependency>
  ...
</dependencies>
```

此特性不做详细说明。感兴趣可以自行了解。

参考资料:

1 源码阅读网Mybatis部分

 <https://coderead.cn/p/mybatis/doc/sql%E6%89%A7%E8%A1%8C%E8%BF%87%...>

2 《Mybatis3源码深度解析》宋荣波

3 <https://developer.aliyun.com/article/1144876>