

# MyBatis执行Sql的流程分析

## MyBatis执行Sql的流程分析

openSession的过程:

获取Mapper的流程

Mapper方法的执行流程

简单总结

Mybatis-插件原理

重要类

调试主要关注点

文档: Mybatis-设计模式总结.note

链接: [http://note.youdao.com/noteshare?](http://note.youdao.com/noteshare?id=4975cd9e83f1e73e14a369598a232abe&sub=5D52C27921074712B1AB91C9A72455C0)

[id=4975cd9e83f1e73e14a369598a232abe&sub=5D52C27921074712B1AB91C9A72455C0](http://note.youdao.com/noteshare?id=4975cd9e83f1e73e14a369598a232abe&sub=5D52C27921074712B1AB91C9A72455C0)

本章着重介绍MyBatis执行Sql的流程, 关于在执行过程中缓存、动态SQL生成等细节不在本章中体现

还是以之前的查询作为例子:

```
1
2 /**
3  * @Author 徐庶 QQ:1092002729
4  * @Slogan 致敬大师, 致敬未来的你
5  */
6 public class App {
7     public static void main(String[] args) {
8         String resource = "mybatis-config.xml";
9         Reader reader;
10        try {
11            //将XML配置文件构建为Configuration配置类
12            reader = Resources.getResourceAsReader(resource);
13            // 通过加载配置文件流构建一个SqlSessionFactory DefaultSqlSessionFactory
14            SqlSessionFactory sqlMapper = new SqlSessionFactoryBuilder().build(reader);
15            // 数据源 执行器 DefaultSqlSession
16            SqlSession session = sqlMapper.openSession();
17            try {
18                // 执行查询 底层执行jdbc
```

```

19 //User user = (User)session.selectOne("com.tuling.mapper.selectById", 1);
20
21 UserMapper mapper = session.getMapper(UserMapper.class);
22 System.out.println(mapper.getClass());
23 User user = mapper.selectById(1L);
24 System.out.println(user.getUserName());
25 } catch (Exception e) {
26     e.printStackTrace();
27 }finally {
28     session.close();
29 }
30 } catch (IOException e) {
31     e.printStackTrace();
32 }
33 }
34 }

```

之前提到拿到sqlSession之后就能进行各种CRUD操作了，所以我们就从sqlSession.getMapper这个方法开始分析，看下整个Sql的执行流程是怎么样的。

## openSession的过程:

Copy

```

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         //获取执行器，这边获得的执行器已经代理拦截器的功能（见下面代码）
8         final Executor executor = configuration.newExecutor(tx, execType);
9         //根据获取的执行器创建SqlSession
10        return new DefaultSqlSession(configuration, executor, autoCommit);
11    } catch (Exception e) {
12        closeTransaction(tx); // may have fetched a connection so lets call close()
13        throw ExceptionFactory.wrapException("Error opening session. Cause: " + e, e);
14    } finally {
15        ErrorContext.instance().reset();
16    }
17 }
18 Copy
19 //interceptorChain生成代理类，具体参见Plugin这个方法
20 public Executor newExecutor(Transaction transaction, ExecutorType executorType) {

```

```

21 executorType = executorType == null ? defaultExecutorType : executorType;
22 executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
23 Executor executor;
24 if (ExecutorType.BATCH == executorType) {
25     executor = new BatchExecutor(this, transaction);
26 } else if (ExecutorType.REUSE == executorType) {
27     executor = new ReuseExecutor(this, transaction);
28 } else {
29     executor = new SimpleExecutor(this, transaction);
30 }
31 if (cacheEnabled) {
32     executor = new CachingExecutor(executor);
33 }
34 executor = (Executor) interceptorChain.pluginAll(executor);
35 return executor;
36 }

```

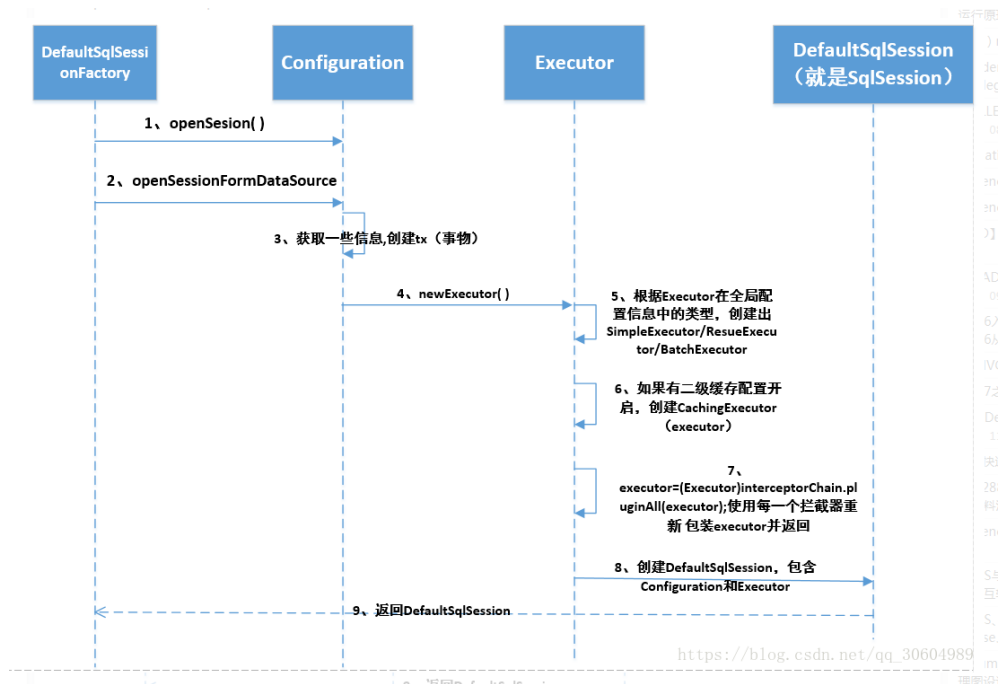
Executor分成两大类，一类是CacheExecutor，另一类是普通Executor。

普通Executor又分为三种基本的Executor执行器，SimpleExecutor、ReuseExecutor、BatchExecutor。

- SimpleExecutor: 每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。
- ReuseExecutor: 执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。简言之，就是重复使用Statement对象。
- BatchExecutor: 执行update（没有select，JDBC批处理不支持select），将所有sql都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行executeBatch()批处理。与JDBC批处理相同。

作用范围：Executor的这些特点，都严格限制在SqlSession生命周期范围内。

CacheExecutor其实是封装了普通的Executor，和普通的区别是在查询前会先查询缓存中是否存在结果，如果存在就使用缓存中的结果，如果不存在还是使用普通的Executor进行查询，再将查询出来的结果存入缓存。



到此为止，我们已经获得了SqlSession，拿到SqlSession就可以执行各种CRUD方法了。

简单总结

- 拿到SqlSessionFactory对象后，会调用SqlSessionFactory的openSession方法，这个方法会创建一个Sql执行器（Executor），这个Sql执行器会代理你配置的拦截器方法。
- 获得上面的Sql执行器后，会创建一个SqlSession（默认使用DefaultSqlSession），这个SqlSession中也包含了Configuration对象，所以通过SqlSession也能拿到全局配置；
- 获得SqlSession对象后就能执行各种CRUD方法了。

SQL的具体执行流程见后续博客。

一些重要类总结：

- SqlSessionFactory
- SqlSessionFactoryBuilder
- SqlSession（默认使用DefaultSqlSession）
- Executor接口
- Plugin、InterceptorChain的pluginAll方法

## 获取Mapper的流程

进入sqlSession.getMapper方法，会发现调的是Configuration对象的getMapper方法：

```

1 public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
2     //mapperRegistry实质上是一个Map，里面注册了启动过程中解析的各种Mapper.xml
3     //mapperRegistry的key是接口的Class类型
4     //mapperRegistry的Value是MapperProxyFactory,用于生成对应的MapperProxy（动态代理类）
5     return mapperRegistry.getMapper(type, sqlSession);
6 }
  
```

进入getMapper方法：

```

1 public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
  
```

```

2  final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>) knownMappers.get(type);
3  //如果配置文件中没有配置相关Mapper,直接抛异常
4  if (mapperProxyFactory == null) {
5      throw new BindingException("Type " + type + " is not known to the MapperRegistry.");
6  }
7  try {
8      //关键方法
9      return mapperProxyFactory.newInstance(sqlSession);
10 } catch (Exception e) {
11     throw new BindingException("Error getting mapper instance. Cause: " + e, e);
12 }
13 }

```

进入MapperProxyFactory的newInstance方法:

```

1
2  public class MapperProxyFactory<T> {
3
4      private final Class<T> mapperInterface;
5      private final Map<Method, MapperMethod> methodCache = new
        ConcurrentHashMap<Method, MapperMethod>();
6
7      public MapperProxyFactory(Class<T> mapperInterface) {
8          this.mapperInterface = mapperInterface;
9      }
10
11     public Class<T> getMapperInterface() {
12         return mapperInterface;
13     }
14
15     public Map<Method, MapperMethod> getMethodCache() {
16         return methodCache;
17     }
18
19     //生成Mapper接口的动态代理类MapperProxy, MapperProxy实现了InvocationHandler 接口
20     @SuppressWarnings("unchecked")
21     protected T newInstance(MapperProxy<T> mapperProxy) {
22         return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[] {
            mapperInterface }, mapperProxy);
23     }
24
25     public T newInstance(SqlSession sqlSession) {
26         final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface,
            methodCache);
27         return newInstance(mapperProxy);

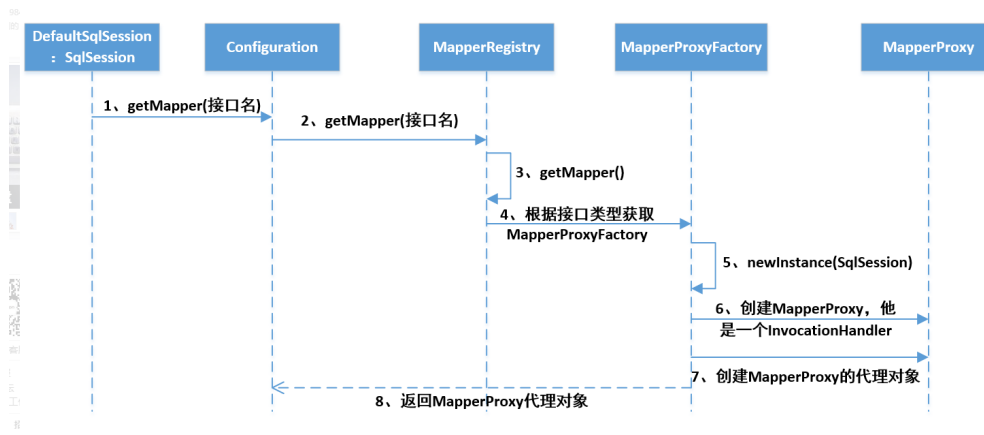
```

```

28  }
29
30  }

```

获取Mapper的流程总结如下：



## Mapper方法的执行流程

下面是动态代理类MapperProxy，调用Mapper接口的所有方法都会先调用到这个代理类的invoke方法（注意由于Mybatis中的Mapper接口没有实现类，所以MapperProxy这个代理对象中没有委托类，也就是说MapperProxy干了代理类和委托类的事情）。好了下面重点看下invoke方法。

```

1
2 //MapperProxy代理类
3 public class MapperProxy<T> implements InvocationHandler, Serializable {
4
5     private static final long serialVersionUID = -6424540398559729838L;
6     private final SqlSession sqlSession;
7     private final Class<T> mapperInterface;
8     private final Map<Method, MapperMethod> methodCache;
9
10    public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method, MapperMethod> methodCache) {
11        this.sqlSession = sqlSession;
12        this.mapperInterface = mapperInterface;
13        this.methodCache = methodCache;
14    }
15
16    @Override
17    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
18        try {
19            if (Object.class.equals(method.getDeclaringClass())) {
20                return method.invoke(this, args);
21            } else if (isDefaultMethod(method)) {
22                return invokeDefaultMethod(proxy, method, args);
23            }

```

```

24 } catch (Throwable t) {
25     throw ExceptionUtil.unwrapThrowable(t);
26 }
27 //获取MapperMethod，并调用MapperMethod
28 final MapperMethod mapperMethod = cachedMapperMethod(method);
29 return mapperMethod.execute(sqlSession, args);
30 }

```

MapperProxy的invoke方法非常简单，主要干的工作就是创建MapperMethod对象或者是从缓存中获取MapperMethod对象。获取到这个对象后执行execute方法。

所以这边需要进入MapperMethod的execute方法：这个方法判断你当前执行的方式是增删改查哪一种，并通过SqlSession执行相应的操作。（这边以sqlSession.selectOne这种方式进行分析~）

```

1 public Object execute(SqlSession sqlSession, Object[] args) {
2     Object result;
3     //判断是CRUD那种方法
4     switch (command.getType()) {
5     case INSERT: {
6         Object param = method.convertArgsToSqlCommandParam(args);
7         result = rowCountResult(sqlSession.insert(command.getName(), param));
8         break;
9     }
10    case UPDATE: {
11        Object param = method.convertArgsToSqlCommandParam(args);
12        result = rowCountResult(sqlSession.update(command.getName(), param));
13        break;
14    }
15    case DELETE: {
16        Object param = method.convertArgsToSqlCommandParam(args);
17        result = rowCountResult(sqlSession.delete(command.getName(), param));
18        break;
19    }
20    case SELECT:
21        if (method.returnsVoid() && method.hasResultHandler()) {
22            executeWithResultHandler(sqlSession, args);
23            result = null;
24        } else if (method.returnsMany()) {
25            result = executeForMany(sqlSession, args);
26        } else if (method.returnsMap()) {
27            result = executeForMap(sqlSession, args);
28        } else if (method.returnsCursor()) {
29            result = executeForCursor(sqlSession, args);
30        } else {

```

```

31 Object param = method.convertArgsToSqlCommandParam(args);
32 result = sqlSession.selectOne(command.getName(), param);
33 }
34 break;
35 case FLUSH:
36 result = sqlSession.flushStatements();
37 break;
38 default:
39 throw new BindingException("Unknown execution method for: " + command.getName());
40 }
41 if (result == null && method.getReturnType().isPrimitive() &&
!method.returnsVoid()) {
42 throw new BindingException("Mapper method '" + command.getName()
43 + " attempted to return null from a method with a primitive return type (" + method.getReturnType() + ").");
44 }
45 return result;
46 }

```

详细流程图

<https://www.processon.com/view/link/5efc23966376891e81f2a37e>

`sqlSession.selectOne`方法会会调到DefaultSqlSession的`selectList`方法。这个方法获取了获取了MappedStatement对象，并最终调用了Executor的query方法。

```

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

```

然后，通过一层一层的调用（这边省略了缓存操作的环节，会在后面的文章中介绍），最终会来到doQuery方法，这儿咱们就随便找个Excutor看看doQuery方法的实现吧，我这儿选择了SimpleExecutor:

Copy

```

1 public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws SQLException {
2     Statement stmt = null;
3     try {

```



```

4 Configuration configuration = ms.getConfiguration();
5 //内部封装了ParameterHandler和ResultSetHandler
6 StatementHandler handler = configuration.newStatementHandler(wrapper, ms, parameter, rowBounds, resultHandler, boundSql);
7 stmt = prepareStatement(handler, ms.getStatementLog());
8 //StatementHandler封装了Statement, 让 StatementHandler 去处理
9 return handler.<E>query(stmt, resultHandler);
10 } finally {
11     closeStatement(stmt);
12 }
13 }

```

接下来，咱们看看StatementHandler的一个实现类 PreparedStatementHandler（这也是我们最常用的，封装的是PreparedStatement），看看它是怎么去处理的：

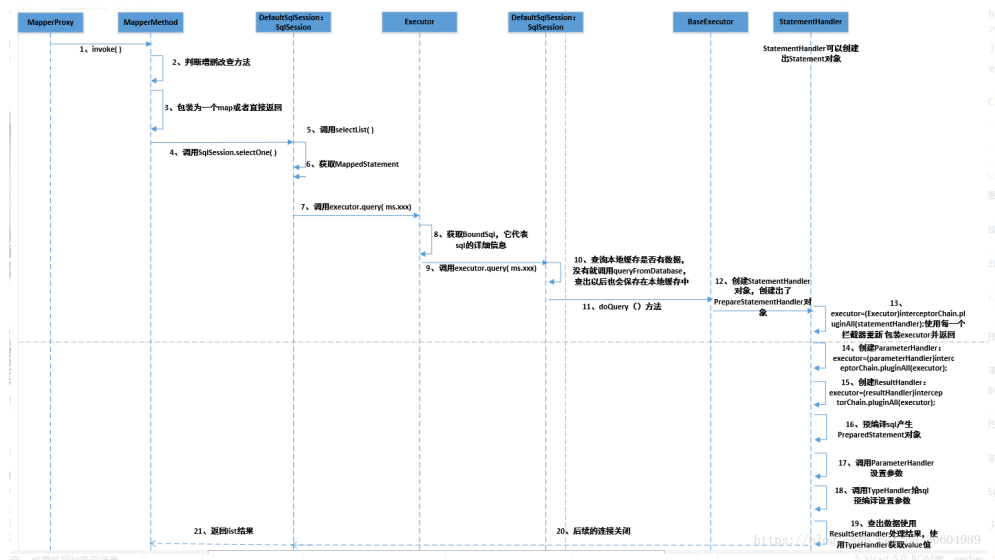
Copy

```

1 public <E> List<E> query(Statement statement, ResultHandler resultHandler) throws SQLException {
2     //到此，原形毕露， PreparedStatement，这个大家都已经滚瓜烂熟了吧
3     PreparedStatement ps = (PreparedStatement) statement;
4     ps.execute();
5     //结果交给了ResultSetHandler 去处理,处理完之后返回给客户端
6     return resultHandler.<E> handleResultSets(ps);
7 }

```

到此，整个调用流程结束。



## 简单总结

这边结合获取SqlSession的流程，做下简单的总结：

- SqlSessionFactoryBuilder解析配置文件，包括属性配置、别名配置、拦截器配置、环境（数据源和事务管理器）、Mapper配置等；解析完这些配置后会生成一个Configuration对象，这个对象中包含了MyBatis需要的所有配置，然后会用这个Configuration对象创建一个SqlSessionFactory对象，这个对象中包含了Configuration对象；

- 拿到SqlSessionFactory对象后，会调用SqlSessionFactory的openSesion方法，这个方法会创建一个Sql执行器（Executor组件中包含了Transaction对象），这个Sql执行器会代理你配置的拦截器方法。
- 获得上面的Sql执行器后，会创建一个SqlSession（默认使用DefaultSqlSession），这个SqlSession中也包含了Configuration对象和上面创建的Executor对象，所以通过SqlSession也能拿到全局配置；
- 获得SqlSession对象后就能执行各种CRUD方法了。

以上是获得SqlSession的流程，下面总结下本博客中介绍的Sql的执行流程：

- 调用SqlSession的getMapper方法，获得Mapper接口的动态代理对象MapperProxy，调用Mapper接口的所有方法都会调用到MapperProxy的invoke方法（动态代理机制）；
- MapperProxy的invoke方法中唯一做的就是创建一个MapperMethod对象，然后调用这个对象的execute方法，sqlSession会作为execute方法的入参；
- 往下，层层调下来会进入Executor组件（如果配置插件会对Executor进行动态代理）的query方法，这个方法中会创建一个StatementHandler对象，这个对象中同时会封装ParameterHandler和ResultSetHandler对象。调用StatementHandler预编译参数以及设置参数值，使用ParameterHandler来给sql设置参数。

Executor组件有两个直接实现类，分别是BaseExecutor和CachingExecutor。

CachingExecutor静态代理了BaseExecutor。Executor组件封装了Transction组件，Transction组件中又分装了Datasource组件。

- 调用StatementHandler的增删改查方法获得结果，ResultSetHandler对结果进行封装转换，请求结束。

Executor、StatementHandler、ParameterHandler、ResultSetHandler，Mybatis的插件会对上面的四个组件进行动态代理。

## Mybatis-插件原理

链接：[http://note.youdao.com/noteshare?](http://note.youdao.com/noteshare?id=80acf548788cef82ffb924f043241365&sub=FAE1C62BE5C4422EBA80EF27A171C067)

[id=80acf548788cef82ffb924f043241365&sub=FAE1C62BE5C4422EBA80EF27A171C067](http://note.youdao.com/noteshare?id=80acf548788cef82ffb924f043241365&sub=FAE1C62BE5C4422EBA80EF27A171C067)

## 重要类

- MapperRegistry：本质上是一个Map，其中的key是Mapper接口的全限定名，value的MapperProxyFactory；

- **MapperProxyFactory**: 这个类是MapperRegistry中存的value值，在通过sqlSession获取Mapper时，其实先获取到的是这个工厂，然后通过这个工厂创建Mapper的动态代理类；
- **MapperProxy**: 实现了InvocationHandler接口，Mapper的动态代理接口方法的调用都会到达这个类的invoke方法；
- **MapperMethod**: 判断你当前执行的方式是增删改查哪一种，并通过SqlSession执行相应的操作；
- **SqlSession**: 作为MyBatis工作的主要顶层API，表示和数据库交互的会话，完成必要数据库增删改查功能；
- **Executor**: 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**: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**:负责对用户传递的参数转换成JDBC Statement 所需要的参数，

**ResultSetHandler**:负责将JDBC返回的ResultSet结果集对象转换成List类型的集合；

**TypeHandler**:负责java数据类型和jdbc数据类型之间的映射和转换

**MappedStatement**:MappedStatement维护了一条<select|update|delete|insert>节点的封装，

**SqlSource**:负责根据用户传递的parameterObject，动态地生成SQL语句，将信息封装到BoundSql对象中，并返回

**BoundSql**:表示动态生成的SQL语句以及相应的参数信息

**Configuration**:MyBatis所有的配置信息都维持在Configuration对象之中。

### 调试主要关注点

- **MapperProxy.invoke方法**: MyBatis的所有Mapper对象都是通过动态代理生成的，任何方法的调用都会调到invoke方法，这个方法的主要功能就是创建MapperMethod对象，并放进缓存。所以调试时我们可以在这个位置打个断点，看下是否成功拿到了MapperMethod对象，并执行了execute方法。
- **MapperMethod.execute方法**: 这个方法会判断你当前执行的方式是增删改查哪一种，并通过SqlSession执行相应的操作。Debug时也建议在此打个断点看下。
- **DefaultSqlSession.selectList方法**: 这个方法获取了MappedStatement对象，并最终调用了Executor的query方法；

### 问题:

#### 1.请介绍下MyBatisSQL语句的解析过程原理

#### 2.请介绍下MyBatis缓存的原理

### 3.请介绍下MyBatis插件的原理

文档: 02-MyBatis-执行Sql的流程分析.note

链接: [http://note.youdao.com/noteshare?](http://note.youdao.com/noteshare?id=c349d36bd4eca8aabcb33dc385a6ebaf&sub=E653129B77054064B8E17E5062681500)

[id=c349d36bd4eca8aabcb33dc385a6ebaf&sub=E653129B77054064B8E17E5062681500](http://note.youdao.com/noteshare?id=c349d36bd4eca8aabcb33dc385a6ebaf&sub=E653129B77054064B8E17E5062681500)