MyBatis的解析和运行原理

构建SqlSessionFactory过程

SqlSessionFactory提供创建MyBatis的核心接口SqlSession。MyBatis采用构造模式去创建SqlSessionFactory，我们可以通过SqlSessionFactoryBuilder去构建。

- 第一步，通过XMLConfigBuilder解析配置的XML文件，读出配置参数，并将读取的数据存入这个Configuration类中。
- 第二步，使用Configuration对象去创建SqlSessionFactory。

SqlSessionFactoryBuilder的源码：

```
public class SqlSessionFactoryBuilder {
  .....
  public SqlSessionFactory build(InputStream inputStream, String environment,
Properties properties) {
      try {
        XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment,
properties);
        // XMLConfigBuilder解析配置的XML文件，构建Configuration
        return build(parser.parse());
      } catch (Exception e) {
        throw ExceptionFactory.wrapException("Error building SqlSession.", e);
      } finally {
        ErrorContext.instance().reset();
        try {
          inputStream.close();
        } catch (IOException e) {
          // Intentionally ignore. Prefer previous error.
        }
      }
  }

  // 使用Configuration对象去创建SqlSessionFactory
  public SqlSessionFactory build(Configuration config) {
      // SqlSessionFactory是一个接口，为此MyBatis提供了一个默认实现类
      return new DefaultSqlSessionFactory(config);
  }
}
```

构建Configuration

在XMLConfigBuilder中，MyBatis会读出所有XML配置的信息，然后将这些信息保存到Configuration类的单例中。

它会做如下初始化：

- properties全局参数
- setting设置
- typeAliases别名
- typeHandler类型处理器
- ObjectFactory对象
- plugin插件
- environment环境
- DatabaseIdProvider数据库标识
- Mapper映射器

XMLConfigBuilder的源码：

```
public class XMLConfigBuilder extends BaseBuilder {
  ...
  public Configuration parse() {
      if (parsed) {
        throw new BuilderException("Each XMLConfigBuilder can only be used once.");
      }
      parsed = true;
      // 解析配置文件，设置Configuration
      parseConfiguration(parser.evalNode("/configuration"));
      return configuration;
    }

  private void parseConfiguration(XNode root) {
    // 读出MyBatis配置文件中的configuration下的各个子标签元素
    // 把全部信息保存到Configuration类的单例中
    try {
      //issue #117 read properties first
      propertiesElement(root.evalNode("properties"));
      Properties settings = settingsAsProperties(root.evalNode("settings"));
      loadCustomVfs(settings);
      typeAliasesElement(root.evalNode("typeAliases"));
      pluginElement(root.evalNode("plugins"));
      objectFactoryElement(root.evalNode("objectFactory"));
      objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
      reflectorFactoryElement(root.evalNode("reflectorFactory"));
      settingsElement(settings);
      // read it after objectFactory and objectWrapperFactory issue #631
      environmentsElement(root.evalNode("environments"));
      databaseIdProviderElement(root.evalNode("databaseIdProvider"));
      typeHandlerElement(root.evalNode("typeHandlers"));
      // 设置mapper映射器
      mapperElement(root.evalNode("mappers"));
```

```
    } catch (Exception e) {
      throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: "
+ e, e);
    }
  }
}
```

映射器的内部组成

XMLMapperBuilder负责对配置文件中的Mapper映射器进行解析,其中在configurationElement方法中可以看出来，会分别对配置文件中的parameterMap、resultMap、sql、select|insert|update|delete元素进行解析。

```
public class XMLMapperBuilder extends BaseBuilder {
  public void parse() {
    if (!configuration.isResourceLoaded(resource)) {
      // 解析配置文件中的mapper映射器
      configurationElement(parser.evalNode("/mapper"));
      configuration.addLoadedResource(resource);
      bindMapperForNamespace();
    }

    parsePendingResultMaps();
    parsePendingCacheRefs();
    parsePendingStatements();
  }

  private void configurationElement(XNode context) {
    try {
      String namespace = context.getStringAttribute("namespace");
      if (namespace == null || namespace.equals("")) {
        throw new BuilderException("Mapper's namespace cannot be empty");
      }
      builderAssistant.setCurrentNamespace(namespace);
      cacheRefElement(context.evalNode("cache-ref"));
      cacheElement(context.evalNode("cache"));
      // 解析我们配置的parameterMap元素
      parameterMapElement(context.evalNodes("/mapper/parameterMap"));
      // 解析我们配置的resultMap元素
      resultMapElements(context.evalNodes("/mapper/resultMap"));
      // 解析我们配置的sql元素
      sqlElement(context.evalNodes("/mapper/sql"));
       // 解析我们配置的select、insert、update、delete元素
      buildStatementFromContext(context.evalNodes("select|insert|update|delete"));
    } catch (Exception e) {
      throw new BuilderException("Error parsing Mapper XML. Cause: " + e, e);
    }
```

```
  }
}
```

在方法buildStatementFromContext()中，会根据配置信息创建一个MappedStatement对象。

MappedStatement,它保存映射器的一个节点(select|insert|update|delete)。包括许多我们配置的SQL、SQL的id、缓存信息、resultMap、parameterType、resultType、languageDriver等重要配置内容。

```
public final class MappedStatement {
  private Configuration configuration;
  private String id;
  private StatementType statementType;
  private ResultSetType resultSetType;
  private SqlSource sqlSource;
  private Cache cache;
  private ParameterMap parameterMap;
  private List<ResultMap> resultMaps;
  private boolean flushCacheRequired;
  private boolean useCache;
  private SqlCommandType sqlCommandType;
  private KeyGenerator keyGenerator;
  private String databaseId;
  private LanguageDriver lang;
  ......
}
```

SqlSource,它是提供BoundSql对象的地方，它是MappedStatement的一个属性。

```
public interface SqlSource {

  BoundSql getBoundSql(Object parameterObject);

}
```

BoundSql,它是建立SQL和参数的地方。

```
public class BoundSql {

  private final String sql;
  private final List<ParameterMapping> parameterMappings;
  private final Object parameterObject;
  private final Map<String, Object> additionalParameters;
  private final MetaObject metaParameters;
}
```

SqlSession运行过程

SqlSession是一个接口，在MyBatis中有一个默认实现DefaultSqlSession。我们构建SqlSessionFactory就可以轻易地拿到SqlSession了。通过SqlSession，我们拿到Mapper，之后可以做查询、插入、更新、删除的方法。

```
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
```

其实getMapper()方法拿到的mapper是通过Java动态代理实现的。从getMapper()方法逐级往下看，可以发现在MapperRegistry类的getMapper()方法中会拿到一个MapperProxyFactory的对象，最后是通过MapperProxyFactory对象去生成一个Mapper的。

```
public class DefaultSqlSession implements SqlSession {
  .....
  @Override
  public <T> T getMapper(Class<T> type) {
    return configuration.<T>getMapper(type, this);
  }
}

public class Configuration {
  .....
  public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    return mapperRegistry.getMapper(type, sqlSession);
  }
}

public class MapperRegistry {
  ......
  public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>)
knownMappers.get(type);
    if (mapperProxyFactory == null) {
      throw new BindingException("Type " + type + " is not known to the
MapperRegistry.");
    }
    try {
      return mapperProxyFactory.newInstance(sqlSession);
    } catch (Exception e) {
      throw new BindingException("Error getting mapper instance. Cause: " + e, e);
    }
  }
}
```

映射器的动态代理

Mapper映射是通过动态代理实现的，MapperProxyFactory用来生成动态代理对象。

```
public class MapperProxyFactory<T> {
  ......
  protected T newInstance(MapperProxy<T> mapperProxy) {
     // 动态代理
     return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new
Class[] { mapperInterface }, mapperProxy);
  }

  public T newInstance(SqlSession sqlSession) {
     final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession,
mapperInterface, methodCache);
     return newInstance(mapperProxy);
  }
}
```

在MapperProxyFactory的newInstance方法中可以看到有一个MapperProxy对象，MapperProxy实现InvocationHandler接口(动态代理需要实现这一接口)的代理方法invoke()，这invoke()方法实现对被代理类的方法进行拦截。

而在invoke()方法中，MapperMethod对象会执行Mapper接口的查询或其他方法。

```
public class MapperProxy<T> implements InvocationHandler, Serializable {
  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    try {
      // 先判断是否一个类，在这里Mapper显然是一个接口
     if (Object.class.equals(method.getDeclaringClass())) {
      return method.invoke(this, args);
      // 判断是不是接口默认实现方法
     } else if (isDefaultMethod(method)) {
      return invokeDefaultMethod(proxy, method, args);
     }
    } catch (Throwable t) {
     throw ExceptionUtil.unwrapThrowable(t);
    }
    // 缓存中取出MapperMethod，不存在的话，则根据Configuration初始化一个
    final MapperMethod mapperMethod = cachedMapperMethod(method);
    // 执行Mapper接口的查询或其他方法
    return mapperMethod.execute(sqlSession, args);
}

private MapperMethod cachedMapperMethod(Method method) {
    MapperMethod mapperMethod = methodCache.get(method);
    if (mapperMethod == null) {
     mapperMethod = new MapperMethod(mapperInterface, method,
sqlSession.getConfiguration());
     methodCache.put(method, mapperMethod);
```

```
    }
    return mapperMethod;
  }
}
```

MapperMethod采用命令模式运行，并根据上下文跳转。MapperMethod在构造器初始化时会根据Configuration和Mapper的Method方法解析为SqlCommand命令。之后在execute方法，根据SqlCommand的Type进行跳转。然后采用命令模式，SqlSession通过SqlCommand执行插入、更新、查询、选择等方法。

```
public MapperMethod(Class<?> mapperInterface, Method method, Configuration
config) {
    // 根据Configuration和Mapper的Method方法解析为SqlCommand
    this.command = new SqlCommand(config, mapperInterface, method);
    this.method = new MethodSignature(config, mapperInterface, method);
}

public Object execute(SqlSession sqlSession, Object[] args) {
    Object result;
    // 根据Type进行跳转，通过sqlSession执行相关的操作
    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);
      }
      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;
}
```

看到这里，应该大概知道了MyBatis为什么只用Mapper接口便能够运行SQL，因为映射器的XML文件的命名空间namespace对应的便是这个接口的全路径，那么它根据全路径和方法名便能够绑定起来，通过动态代理技术，让这个接口跑起来。而后采用命令模式，最后还是使用SqlSession接口的方法使得它能够执行查询，有了这层封装我们便可以使用这个接口编程。不过还是可以看到，最后插入、更新、删除、查询操作还是会回到SqlSession中进行处理。

Sqlsession下的四大对象

我们已经知道了映射器其实就是一个动态代理对象，进入到了MapperMethod的execute方法。它经过简单判断就是进入了SqlSession的删除、更新、插入、选择等方法。sqlSession执行一个查询操作。可以看到是通过一个executor来执行的。

其实SqlSession中的Executor执行器负责调度StatementHandler、ParameterHandler、ResultHandler等来执行相关的SQL。

- StatementHandler：使用数据库的Statement（PrepareStatement)执行操作
- ParameterHandler：用于SQL对参数的处理
- ResultHandler：进行最后数据集（ResultSet)的封装返回处理

Sqlsession其实是一个接口，它有一个DefaultSqlSession的默认实现类。

```
public class DefaultSqlSession implements SqlSession {
  private final Configuration configuration;
  // Executor执行器,负责调度SQL的执行
  private final Executor executor;

  ......
  @Override
  public <E> List<E> selectList(String statement, Object parameter, RowBounds
rowBounds) {
    try {
      MappedStatement ms = configuration.getMappedStatement(statement);
      // 通过executor执行查询操作
      return executor.query(ms, wrapCollection(parameter), rowBounds,
Executor.NO_RESULT_HANDLER);
    } catch (Exception e) {
      throw ExceptionFactory.wrapException("Error querying database.  Cause: " + e,
e);
    } finally {
      ErrorContext.instance().reset();
    }
  }
}
```

Executor执行器

执行器起到了至关重要的作用，它是一个真正执行Java和数据库交互的东西。在MyBatis中存在三种执行器，我们可以在MyBatis的配置文件中进行选择。

SIMPLE,简易执行器

REUSE，是一种执行器重用预处理语句

BATCH，执行器重用语句和批量更新，她是针对批量专用的执行器

它们都提供了查询和更新方法，以及相关的事务方法。

Executor是通过Configuration类创建的,MyBatis将根据配置类型去确定你需要创建三种执行器中的哪一种。

```
public class Configuration {
  ......
    public Executor newExecutor(Transaction transaction, ExecutorType
executorType) {
    executorType = executorType == null ? defaultExecutorType : executorType;
    executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
    Executor executor;
    if (ExecutorType.BATCH == executorType) {
      executor = new BatchExecutor(this, transaction);
```

```
    } else if (ExecutorType.REUSE == executorType) {
      executor = new ReuseExecutor(this, transaction);
    } else {
      executor = new SimpleExecutor(this, transaction);
    }
    if (cacheEnabled) {
      executor = new CachingExecutor(executor);
    }
    // MyBatis插件，构建一层层的动态代理对象
    // 在调度真实的方法之前执行配置插件的代码
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
  }
}
```

显然MyBatis根据Configuration来构建StatementHandler，然后使用prepareStatement方法，对SQL编译并对参数进行初始化，resultHandler再组装查询结果返回给调用者来完成一次查询。

```
public class SimpleExecutor extends BaseExecutor {
    .....
    @Override
    public <E> List<E> doQuery(MappedStatement ms, Object parameter,
RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws
SQLException {
        Statement stmt = null;
        try {
            Configuration configuration = ms.getConfiguration();
            // 根据Configuration来构建StatementHandler
            StatementHandler handler = configuration.newStatementHandler(wrapper,
ms, parameter, rowBounds, resultHandler, boundSql);
            // 对SQL编译并对参数进行初始化
            stmt = prepareStatement(handler, ms.getStatementLog());
            // 组装查询结果返回给调用者
            return handler.<E>query(stmt, resultHandler);
        } finally {
            closeStatement(stmt);
        }
    }

  private Statement prepareStatement(StatementHandler handler, Log
statementLog) throws SQLException {
    Statement stmt;
    Connection connection = getConnection(statementLog);
    // 进行预编译和基础设置
    stmt = handler.prepare(connection, transaction.getTimeout());
```

```
    // 设置参数
    handler.parameterize(stmt);
    return stmt;
  }
}
```

StatementHandler数据库会话器

StatementHandler就是专门处理数据库会话的。

创建StatementHandler：

```
public class Configuration {
    ……
    public StatementHandler newStatementHandler(Executor executor,
MappedStatement mappedStatement, Object parameterObject, RowBounds
rowBounds, ResultHandler resultHandler, BoundSql boundSql) {
        StatementHandler statementHandler = new
RoutingStatementHandler(executor, mappedStatement, parameterObject,
rowBounds, resultHandler, boundSql);
        // MyBatis插件，生成一层层的动态代理对象
        statementHandler = (StatementHandler)
interceptorChain.pluginAll(statementHandler);
        return statementHandler;
  }
}
```

RoutingStatementHandler其实不是我们真实的服务对象，它是通过适配模式找到对应的StatementHandler来执行。

StatementHandler分为三种：

- SimleStatementHandler
- PrepareStatementHandler
- CallableStatementHandler

在初始化RoutingStatementHandler对象的时候它会根据上下文环境来决定创建哪个StatementHandler对象。

```
public class RoutingStatementHandler implements StatementHandler {
    ……
    private final StatementHandler delegate;

    public RoutingStatementHandler(Executor executor, MappedStatement ms,
Object parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql
boundSql) {
    switch (ms.getStatementType()) {
```

```
      case STATEMENT:
        delegate = new SimpleStatementHandler(executor, ms, parameter, rowBounds,
resultHandler, boundSql);
        break;
      case PREPARED:
        delegate = new PreparedStatementHandler(executor, ms, parameter,
rowBounds, resultHandler, boundSql);
        break;
      case CALLABLE:
        delegate = new CallableStatementHandler(executor, ms, parameter,
rowBounds, resultHandler, boundSql);
        break;
      default:
        throw new ExecutorException("Unknown statement type: " +
ms.getStatementType());
    }
  }
}
```

数据库会话器定义了一个对象的适配器delegate，它是一个StatementHandler接口对象，构造器根据配置来适配对应的StatementHandler对象。它的作用是给实现类对象的使用提供一个统一、简易的使用适配器。此为对象的适配模式，可以让我们使用现有的类和方法对外提供服务，也可以根据实际的需求对外屏蔽一些方法，甚至加入新的服务。

在执行器Executor执行查询操作的时候，我们看到PreparedStatementHandler的三个方法：prepare、parameterize和query。

```
public abstract class BaseStatementHandler implements StatementHandler {
  .....
    @Override
    public Statement prepare(Connection connection, Integer transactionTimeout)
throws SQLException {
        ErrorContext.instance().sql(boundSql.getSql());
        Statement statement = null;
        try {
            // 对SQL进行了预编译
            statement = instantiateStatement(connection);
            setStatementTimeout(statement, transactionTimeout);
            setFetchSize(statement);
            return statement;
        } catch (SQLException e) {
            closeStatement(statement);
        throw e;
        } catch (Exception e) {
            closeStatement(statement);
```

```
            throw new ExecutorException("Error preparing statement.  Cause: " + e, e);
        }
      }
    }

public class PreparedStatementHandler extends BaseStatementHandler {
    @Override
    public void parameterize(Statement statement) throws SQLException {
        // 设置参数
        parameterHandler.setParameters((PreparedStatement) statement);
    }

    @Override
    protected Statement instantiateStatement(Connection connection) throws
SQLException {
        String sql = boundSql.getSql();
        if (mappedStatement.getKeyGenerator() instanceof Jdbc3KeyGenerator) {
        String[] keyColumnNames = mappedStatement.getKeyColumns();
        if (keyColumnNames == null) {
            return connection.prepareStatement(sql,
PreparedStatement.RETURN_GENERATED_KEYS);
        } else {
            return connection.prepareStatement(sql, keyColumnNames);
        }
        } else if (mappedStatement.getResultSetType() != null) {
            return connection.prepareStatement(sql,
mappedStatement.getResultSetType().getValue(), ResultSet.CONCUR_READ_ONLY);
        } else {
            return connection.prepareStatement(sql);
        }
    }

    @Override
    public <E> List<E> query(Statement statement, ResultHandler resultHandler)
throws SQLException {
        PreparedStatement ps = (PreparedStatement) statement;
        // 执行SQL
        ps.execute();
        // resultSetHandler封装结果返回
        return resultSetHandler.<E> handleResultSets(ps);
    }
}
```

一条查询SQL的执行过程，Executor会先调用StatementHandler的prepare()方法预编译SQL语句，同时设置一些基本运行的参数。然后用parameterize()方法启动ParameterHandler设

置参数，完成预编译，跟着就是执行查询。