SqlSessionFactoryBuilder.build # 创建 sessionFactory 默认 DefaultSqlSessionFactory

XMLConfigBuilder.parse # 解析 xml 配置 Configuration(重要) 整个过程都参与

将 Xml配置中的 mapper 中的所有方法解析映射成对应的 MappedStatement

存放在Configuration.mappedStatements 中实际就是一个Map，Key为 mapper的 包名.类名.方法名 ，Value为生成的MappedStatement

sessionFactory.openSession # 打开一个session 内部调用 openSessionFromDataSource

openSessionFromDataSource 创建JdbcTransaction，创建 CachingExecutor 生成DefaultSqlSession

session.getMapper #生成 mapper 代理 MapperProxy

执行查询 eg：mapper.getById # 实际执行的是 MapperProxy.invoke

缓存Method：

MapperProxy.cacheMapperMethod()

创建 SqlCommand 对象

从 Configuration.mappedStatements 中 获取 MappedStatement 赋值

name = MappedStatement.getId type=MappedStatement.getSqlCommondType

创建 MethodSignature 对象

方法执行 ： MapperMethod.execute()

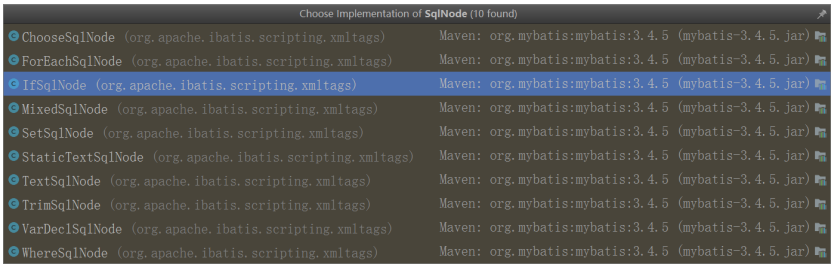
```
public Object execute(SqlSession sqlSession, Object[] args) {
  Object result;
  switch (command.getType()) {
    case INSERT: {
      Object param = method.convertArgsToSqlCommandParam(args);
      result = rowCountResult(sqlSession.insert(command.getName(), param));
      break;
    }
    case 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);
  }
  break;
case FLUSH:
  result = sqlSession.flushStatements();
  break;
default:
  throw new BindingException("Unknown execution method for: " + command.getName());
```

执行对应的case by command.getType()

动态SQL解析

所有的SqlNode都实现了SqlNode接口 此接口只有一个 apply(DynamicContext context) 方法



占位符对应的是TextSqlNode

`DynamicSqlSource.getBoundSql()`

41 行 rootSqlNode.apply(context)　　对SQL中的${}占位方式进行参数值替换

```
39  public BoundSql getBoundSql(Object parameterObject) {  parameterObject:  size = 2
40    DynamicContext context = new DynamicContext(configuration, parameterObject);  context: Dynamic
41    rootSqlNode.apply(context);
42    SqlSourceBuilder sqlSourceParser = new SqlSourceBuilder(configuration);  sqlSourceParser: SqlS
43    Class<?> parameterType = parameterObject == null ? Object.class : parameterObject.getClass();
44    SqlSource sqlSource = sqlSourceParser.parse(context.getSql(), parameterType, context.getBindin
45    BoundSql boundSql = sqlSource.getBoundSql(parameterObject);
46    for (Map.Entry<String, Object> entry : context.getBindings().entrySet()) {
47      boundSql.setAdditionalParameter(entry.getKey(), entry.getValue());
48    }
49    return boundSql;
```

GenericTokenParser.parse　占位符解析与替换

`TextSqlNode.apply()`

SQL解析,替换${} 占位符 为具体的参数值

```
public boolean apply(DynamicContext context) {
  GenericTokenParser parser = createParser(new BindingTokenParser(context, injectionFilter));
  context.appendSql(parser.parse(text));
  return true;
}

private GenericTokenParser createParser(TokenHandler handler) {
  return new GenericTokenParser( openToken: "${", closeToken: "}", handler);
}
```

SqlSourceBuilder.parse() 生成 SqlSource 内部也使用 GenericTokenParser 解析占位符为#{}

```
public SqlSource parse(String originalSql, Class<?> parameterType, Map<String, Object> additionalParameters) {
  ParameterMappingTokenHandler handler = new ParameterMappingTokenHandler(configuration, parameterType, additionalParamete
  GenericTokenParser parser = new GenericTokenParser( openToken: "#{", closeToken: "}", handler);
  String sql = parser.parse(originalSql);
  return new StaticSqlSource(configuration, sql, handler.getParameterMappings());
}
```

select （查询） 最终落到 CachingExecutor.query() 方法上

```
@Override
public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler
  BoundSql boundSql = ms.getBoundSql(parameterObject);  boundSql: BoundSql@1670
  CacheKey key = createCacheKey(ms, parameterObject, rowBounds, boundSql);  ms: MappedStatement@1407  parameterObject
  return query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
}
```

```
@Override
public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds rowBounds, ResultHandler resu
    throws SQLException {
  Cache cache = ms.getCache();
  if (cache != null) {
    flushCacheIfRequired(ms);
    if (ms.isUseCache() && resultHandler == null) {
      ensureNoOutParams(ms, parameterObject, boundSql);
      /unchecked/
      List<E> list = (List<E>) tcm.getObject(cache, key);
      if (list == null) {
        list = delegate.<E> query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
        tcm.putObject(cache, key, list); // issue #578 and #116
      }
      return list;
    }
  }
  return delegate.<E> query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
}
```

此处以key去查询缓存，存在的话做一些缓存相关的处理，返回缓存数据

若缓存不存时，会从数据库中进行查询 最终会执行

BaseExecutor.query() 方法

```java
@SuppressWarnings("unchecked")
@Override
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler result
  ErrorContext.instance().resource(ms.getResource()).activity("executing a query").object(ms.getId());
  if (closed) {
    throw new ExecutorException("Executor was closed.");
  }
  if (queryStack == 0 && ms.isFlushCacheRequired()) {
    clearLocalCache();
  }
  List<E> list;
  try {
    queryStack++;
    list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;
    if (list != null) {
      handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
    } else {
      list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key, boundSql);
    }
  } finally {
    queryStack--;
  }
```

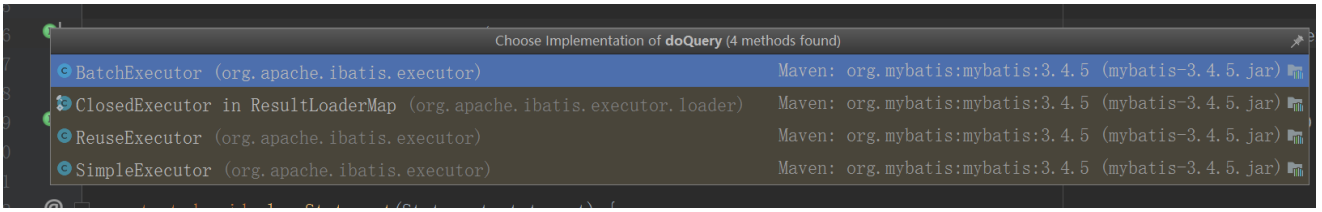
```java
  if (queryStack == 0) {
    for (DeferredLoad deferredLoad : deferredLoads) {
      deferredLoad.load();
    }
    // issue #601
    deferredLoads.clear();
    if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
      // issue #482
      clearLocalCache();
    }
  }
  return list;
```

实际执行 BaseExecutor.queryFromDatabase()　方法

```java
private <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheK
  List<E> list;
  localCache.putObject(key, EXECUTION_PLACEHOLDER);
  try {
    list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
  } finally {
    localCache.removeObject(key);
  }
  localCache.putObject(key, list);
  if (ms.getStatementType() == StatementType.CALLABLE) {
    localOutputParameterCache.putObject(key, parameter);
  }
  return list;
}
```

由上图可见 执行的是BaseExecutor.doQuery方法

doQuery是一个抽象方法有四个实现类



SimpleExecutor.doQuery方法实现

```
@Override
public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql
  Statement stmt = null;
  try {
    Configuration configuration = ms.getConfiguration();
    StatementHandler handler = configuration.newStatementHandler(wrapper, ms, parameter, rowBounds, resultHandler, boundSql);
    stmt = prepareStatement(handler, ms.getStatementLog());
    return handler.<E>query(stmt, resultHandler);
  } finally {
    closeStatement(stmt);
  }
}
```

prepareStatement **方法中会准备执行SQL相关的预处理**

```
private Statement prepareStatement(StatementHandler handler, Log statementLog) throws SQLException {
  Statement stmt;
  Connection connection = getConnection(statementLog);
  stmt = handler.prepare(connection, transaction.getTimeout());
  handler.parameterize(stmt);
  return stmt;
}
```

handler.query()实际调用 SimpleStatementHandler.query() **真正执行Sql的地方**

```
@Override
public <E> List<E> query(Statement statement, ResultHandler resultHandler) throws SQLException {
  String sql = boundSql.getSql();
  statement.execute(sql);
  return resultSetHandler.<E>handleResultSets(statement);
}
```