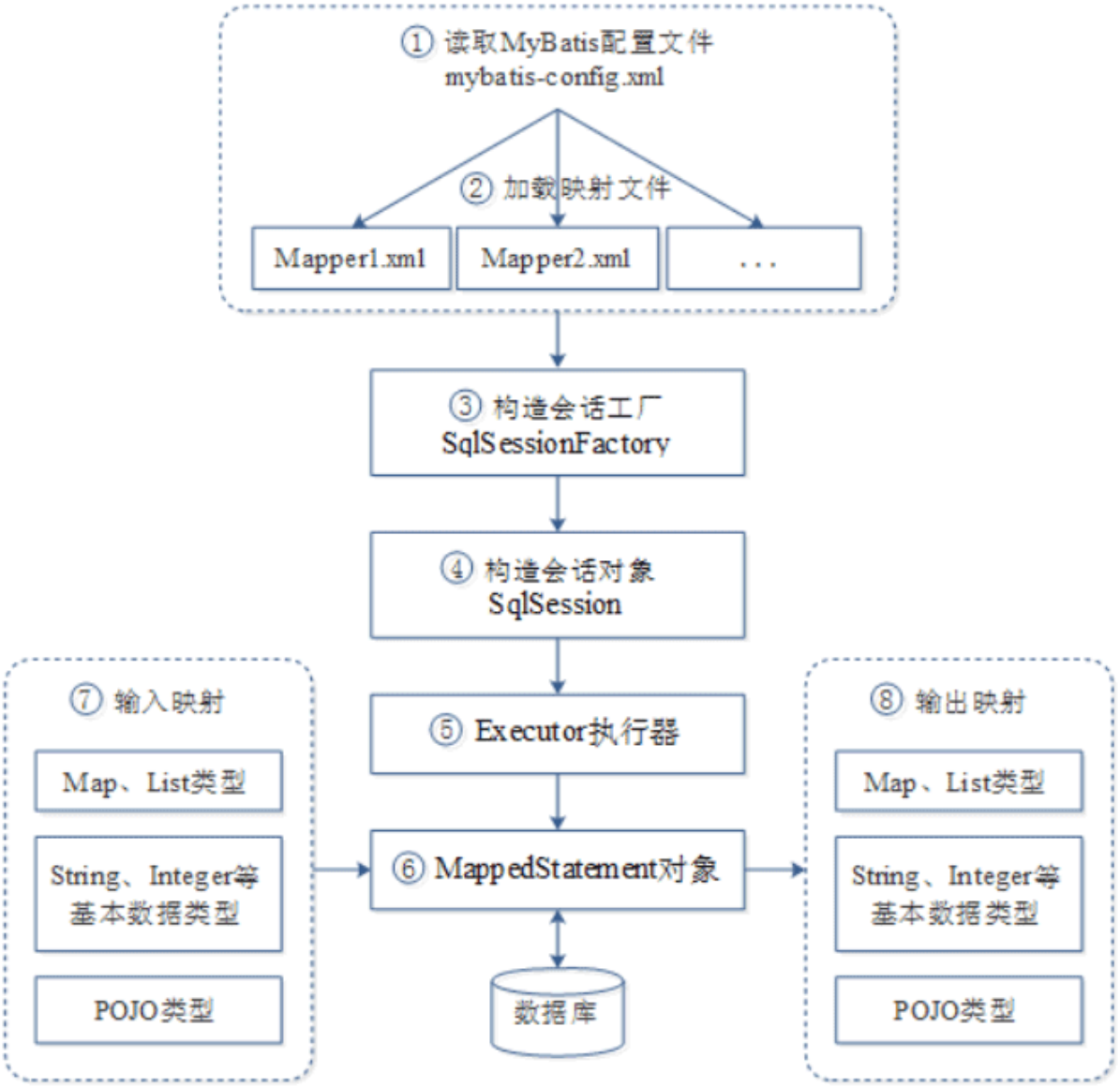
MyBatis程序的工作原理，如图所示



MyBatis框架执行流程图

MyBatis框架在操作数据库时，大体经过了8个步骤。下面就对图1中的每一步流程进行详细讲解，具体如下：

(1)读取MyBatis配置文件mybatis-config.xml。mybatis-config.xml作为MyBatis的全局配置文件，配置了MyBatis的运行环境等信息，其中主要内容是获取数据库连接。初始化配置文件信息的本质就是创建Configuration对象，将解析的xml数据封装到Configuration内部的属性中。根据mybatis-config.xml全局配置文件创建SqlSessionFactory对象、就是把配置文件的详细信息解析保存在了configuration对象中，返回包含了configuration的defaultSqsessionFactory对象

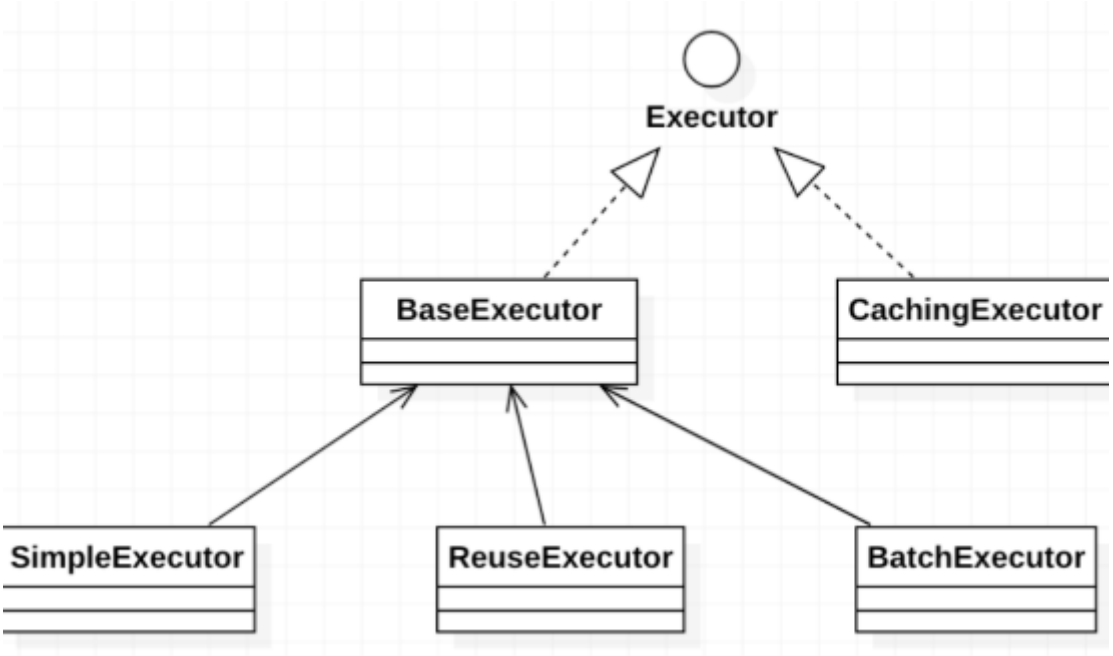
(2)加载映射文件Mapper.xml。Mapper.xml文件即SQL映射文件，该文件中配置了操作数据库的SQL语句，需要在mybatis-config.xml中加载才能执行。mybatis-config.xml可以加载多个配置文件，每个配置文件(Mapper)对应数据库中的一张表。

(3)构建会话工厂。通过MyBatis的环境等配置信息构建会话工厂SqlSessionFactory。

(4)创建SqlSession对象。由会话工厂创建SqlSession对象，该对象中包含了执行SQL的所有方法。**SqlSession是一个接口**，它有两个实现类：DefaultSqlSession（默认）和SqlSessionManager（弃用，不做介绍）,SqlSession是MyBatis中用于和数据库交互的顶层类，通常将它与ThreadLocal绑定，一个会话使用一个SqlSession，并且在使用完毕后需要close。SqlSession中的两个最重要的参数，configuration与初始化时的相同，Executor为执行器，

(5)MyBatis底层定义了一个Executor接口来操作数据库，它会根据SqlSession传递的参数动态的生成需要执行的SQL语句，同时负责查询缓存的维护。

(6)在Executor接口的执行方法中，包含一个MappedStatement类型的参数，该参数是对映射信息的封装，用来存储要映射的SQL语句的id、参数等。Mapper.xml文件中一个SQL对应一个MappedStatement对象，SQL的id即是MappedStatement的id。Executor也是一个接口，他有三个常用的实现类BatchExecutor（重用语句并执行批量更新），ReuseExecutor（重用预处理语句prepared statements），SimpleExecutor（普通的执行器，默认）。



其中Configuration类中对Executor的初始化方法：

```
public Executor newExecutor(Transaction transaction) {
  return newExecutor(transaction, defaultExecutorType);
```

```
}

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);
  }
  executor = (Executor) interceptorChain.pluginAll(executor);
  return executor;
}
```

SqlSession调用executor对象的相应方法，如executor.query，而executor对象，如SimpleExecutor调用prepareStatement(handler, ms.getStatementLog());方法操作数据库.

```
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;
}
```

(7)输入参数映射。在执行方法时，MappedStatement对象会对用户执行SQL语句的输入参数进行定义(可以定义为Map、List类型、基本类型和POJO类型)，Executor执行器会通过MappedStatement对象在执行SQL前，将输入的Java对象映射到SQL语句中。这里对输入参数的映射过程就类似于JDBC编程中对preparedStatement对象设置参数的过程。

介绍一下MappedStatement ：

作用： MappedStatement与Mapper配置文件中的一个select/update/insert/delete节点相对应。mapper中配置的标签都被封装到了此对象中，主要用途是描述一条SQL语句。

初始化过程：回顾刚开始介绍的加载配置文件的过程中，会对mybatis-config.xml中的各个标签都进行解析，其中有 mappers标签用来引入mapper.xml文件或者配置mapper接口的目录。

```
<select id="getUser" resultType="user" >
   select * from user where id=#{id}
```

</select>

这样的一个select标签会在初始化配置文件时被解析封装成一个MappedStatement对象，然后存储在Configuration对象的mappedStatements属性中，mappedStatements 是一个HashMap，存储时

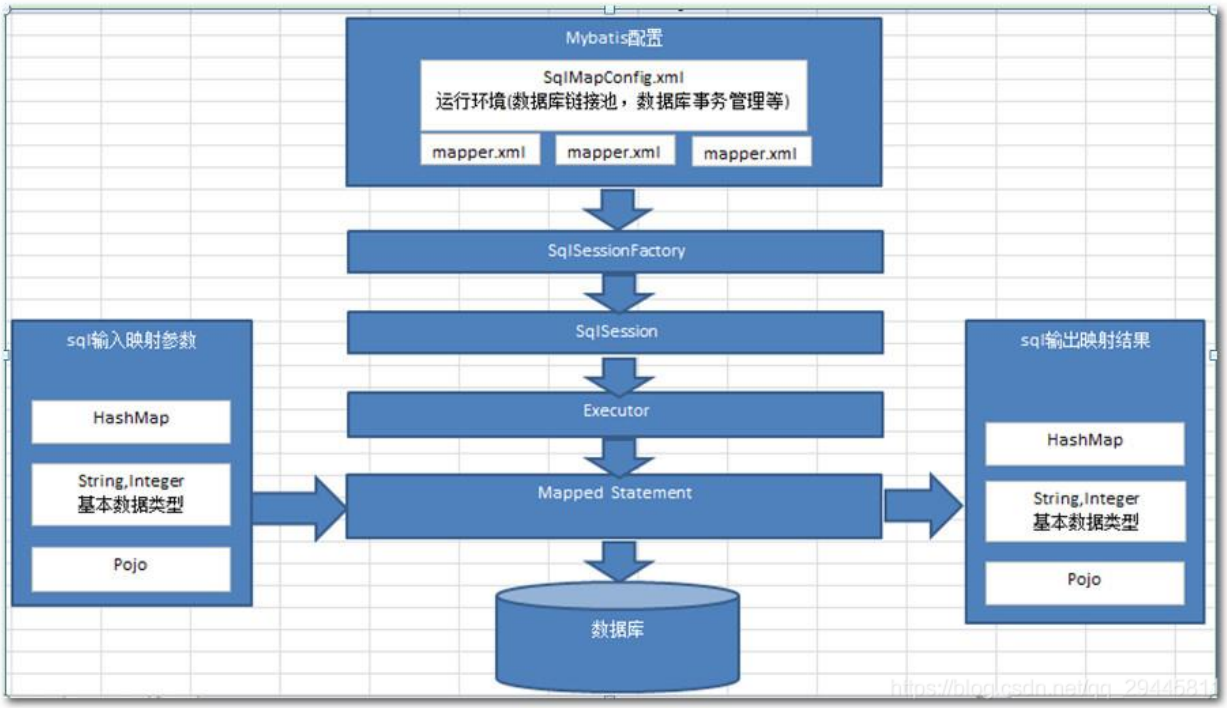
key = 全限定类名 + 方法名，value = 对应的MappedStatement对象。

一个MappedStatement对象代表一个增删改查标签的详细信息（id sqlResource等）

```
statement= MappedStatement (id=201)
    cache= null
    configuration= Configuration (id=54)
    databaseId= null
    fetchSize= null
    flushCacheRequired= false
    hasNestedResultMaps= false
    id= "com.atguigu.mybatis.dao.EmployeeMapper.getEmpById" (id=204)
    keyColumns= null
    keyGenerator= NoKeyGenerator (id=207)
    keyProperties= null
    lang= XMLLanguageDriver (id=210)
    parameterMap= ParameterMap (id=213)
    resource= "EmployeeMapper.xml" (id=133)
    resultMaps= Collections$UnmodifiableRandomAccessList<E> (id=215)
    resultOrdered= false
    resultSets= null
    resultSetType= null
    sqlCommandType= SqlCommandType (id=220)
    sqlSource= RawSqlSource (id=221)
        sqlSource= StaticSqlSource (id=233)
            configuration= Configuration (id=54)
            parameterMappings= ArrayList<E> (id=236)
            sql= "select id,last_name lastName,email,gender from tbl_employee where id = ?" (id=238)
    statementLog= Log4jImpl (id=224)
    statementType= StatementType (id=227)
select id,last_name lastName,email,gender from tbl_employee where
```

(8)输出结果映射。在数据库中执行完SQL语句后，MappedStatement对象会对SQL执行输出的结果进行定义(可以定义为Map和List类型、基本类型、POJO类型)，Executor执行器会通过MappedStatement对象在执行SQL语句后，将输出结果映射至Java对象中。这种将输出结果映射到Java对象的过程就类似于JDBC编程中对结果的解析处理过程。

Mybatis工作原理图



工作原理解析

mybatis应用程序通过SqlSessionFactoryBuilder从mybatis-config.xml配置文件（也可以用Java文件配置的方式，需要添加@Configuration）来构建SqlSessionFactory（SqlSessionFactory是线程安全的）；

然后，SqlSessionFactory的实例直接开启一个SqlSession，再通过SqlSession实例获得Mapper对象并运行Mapper映射的SQL语句，完成对数据库的CRUD和事务提交，之后关闭SqlSession。

说明：SqlSession是单线程对象，因为它是非线程安全的，是持久化操作的独享对象，类似jdbc中的Connection，底层就封装了jdbc连接。

详细流程如下：

1、加载mybatis全局配置文件（数据源、mapper映射文件等），解析配置文件，MyBatis基于XML配置文件生成Configuration，和一个个MappedStatement（包括了参数映射配置、动态SQL语句、结果映射配置），其对应着<select | update | delete | insert>标签项。

```
public Configuration(Environment environment) {
    this();
    this.environment = environment;
}
```

2、SqlSessionFactoryBuilder通过Configuration对象生成SqlSessionFactory，用来开启SqlSession。

```
public SqlSessionFactory build(Configuration config) {
    return new DefaultSqlSessionFactory(config);
}

public class DefaultSqlSessionFactory implements SqlSessionFactory {
    private final Configuration configuration;

    public DefaultSqlSessionFactory(Configuration configuration) {
        this.configuration = configuration;
    }

    public SqlSession openSession() {
        return
this.openSessionFromDataSource(this.configuration.getDefaultExecutorType(),
        (TransactionIsolationLevel)null, false);
    }
    ...
    ...

    private SqlSession openSessionFromDataSource(ExecutorType execType,
TransactionIsolationLevel level, boolean autoCommit) {
    Transaction tx = null;

    DefaultSqlSession var8;
    try {
        Environment environment = this.configuration.getEnvironment();
        TransactionFactory transactionFactory =
this.getTransactionFactoryFromEnvironment(environment);
        tx = transactionFactory.newTransaction(environment.getDataSource(), level,
autoCommit);
        Executor executor = this.configuration.newExecutor(tx, execType);
        var8 = new DefaultSqlSession(this.configuration, executor, autoCommit);
    } catch (Exception var12) {
        this.closeTransaction(tx);
        throw ExceptionFactory.wrapException("Error opening session.  Cause: " +
var12, var12);
    } finally {
        ErrorContext.instance().reset();
    }

    return var8;
}
```

3、SqlSession对象完成和数据库的交互：

a、用户程序调用mybatis接口层api（即Mapper接口中的方法）

b、SqlSession通过调用api的Statement ID找到对应的MappedStatement对象

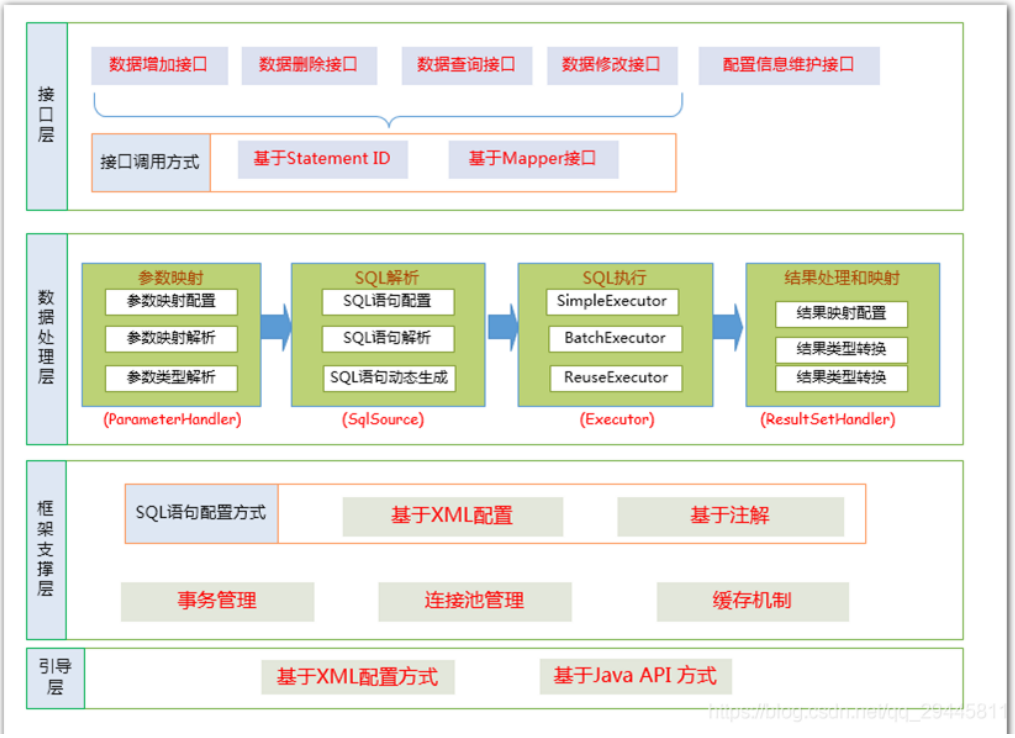
c、通过Executor（负责动态SQL的生成和查询缓存的维护）将MappedStatement对象进行解析，sql参数转化、动态sql拼接，生成jdbc Statement对象

d、JDBC执行sql。

e、借助MappedStatement中的结果映射关系，将返回结果转化成HashMap、JavaBean等存储结构并返回。

mybatis层次图：

思考一个问题，通常的Mapper接口我们都没有实现的方法却可以使用，是为什么呢？答案很简单 动态代理



开始之前介绍一下MyBatis初始化时对接口的处理：MapperRegistry是Configuration中的一个属性，它内部维护一个HashMap用于存放mapper接口的工厂类，每个接口对应一个工厂类。mappers中可以配置接口的包路径，或者某个具体的接口类。

```
<!-- 将包内的映射器接口实现全部注册为映射器 -->
<mappers>
  <mapper class="com.demo.mapper.UserMapper"/>
  <package name="com.demo.mapper"/>
</mappers>
```

当解析mappers标签时，它会判断解析到的是mapper配置文件时，会再将对应配置文件中的增删改查标签一 一封装成MappedStatement对象，存入mappedStatements中。（上文介绍了）

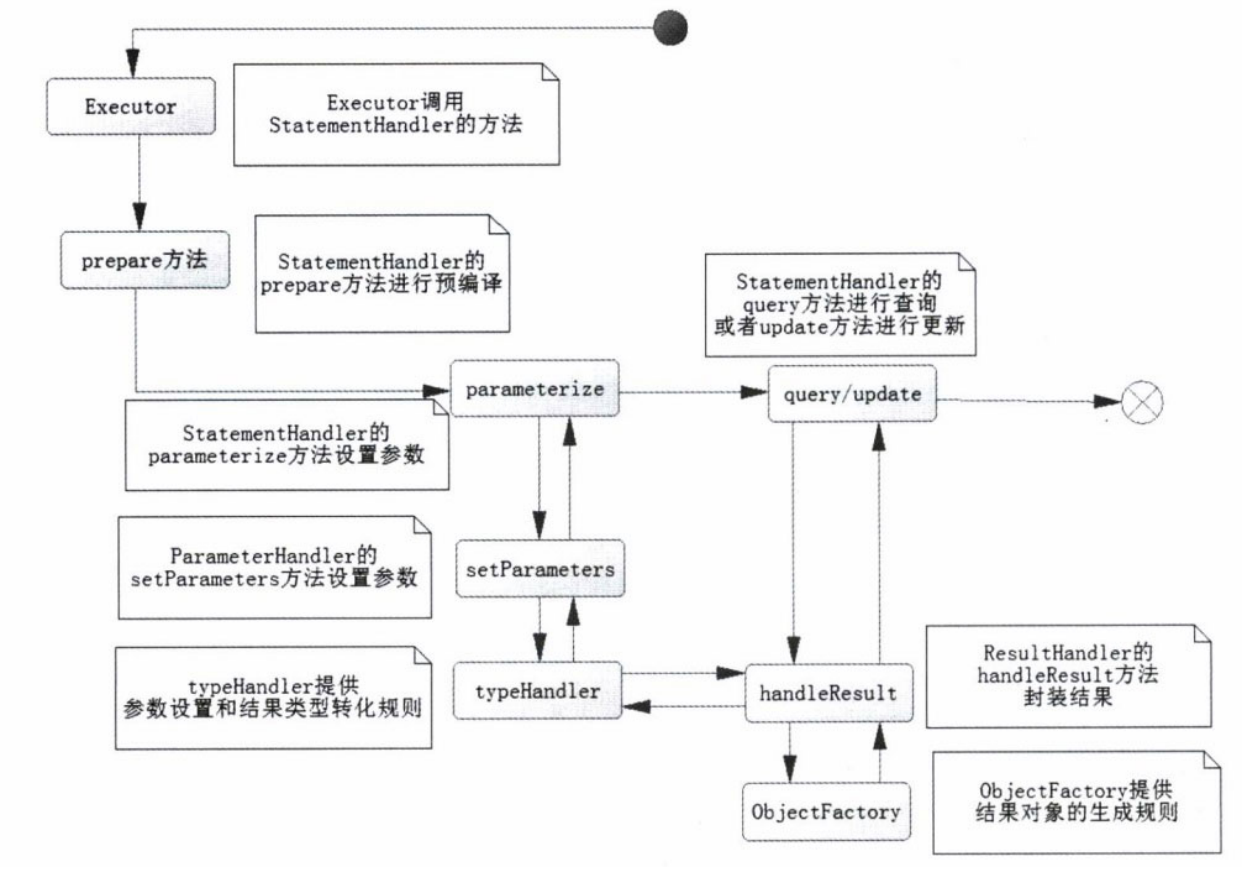
当判断解析到接口时，会创建此接口对应的MapperProxyFactory对象，存入HashMap中，key = 接口的字节码对象，value = 此接口对应的MapperProxyFactory对象。

```
public <T> void addMapper(Class<T> type) {
  if (type.isInterface()) {
    if (hasMapper(type)) {
      throw new BindingException("Type " + type + " is already known to the
MapperRegistry.");
    }
    boolean loadCompleted = false;
    try {
        //重点在这行，以接口类的class对象为key，value为其对应的工厂对象，
        //构造方法中指定了接口对象
      knownMappers.put(type, new MapperProxyFactory<>(type));
      // It's important that the type is added before the parser is run
      // otherwise the binding may automatically be attempted by the
      // mapper parser. If the type is already known, it won't try.
      MapperAnnotationBuilder parser = new MapperAnnotationBuilder(config, type);
      parser.parse();
      loadCompleted = true;
    } finally {
      if (!loadCompleted) {
        knownMappers.remove(type);
      }
    }
  }
}
```

总结：

SqlSession在一个查询开启的时候会先通过CacheExecutor查询缓存。击穿缓存后会通过BaseExector子类的SimpleExecutor创建StatementHandler。PrepareStatementHandler会基于PrepareStament执行数据库操作。并针对返回结果通过ResultSetHandler返回结果数据。

获取sqlSession对象：返回sqlsession的实现类defaultSqlsession对象，defaultSqlsession对象包含了executor和configuration，Executor(四大对象)对象会在这一步被创建



Mybatis运行原理总结

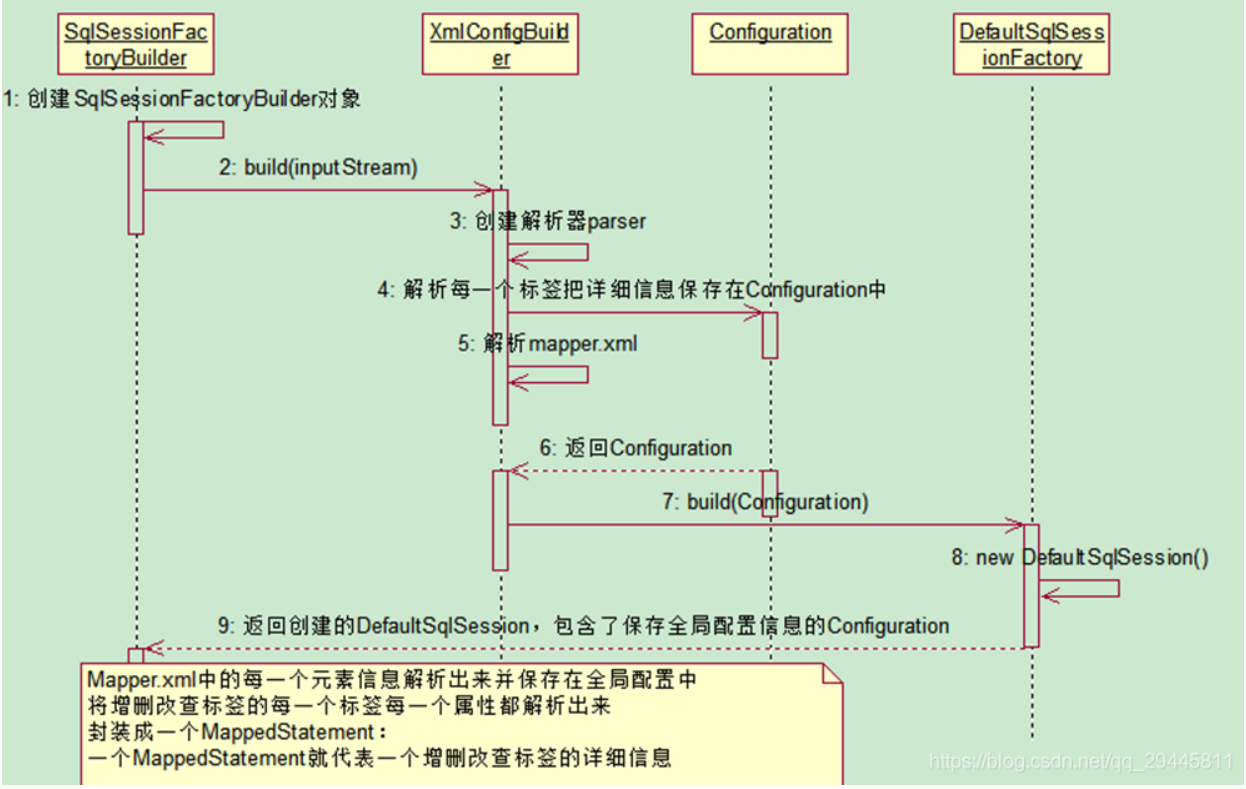
1、根据配置文件（全局、SQL映射文件）初始化出configuration对象

configuration对象中的几个重要属性：

```
protected final MapperRegistry mapperRegistry = new MapperRegistry(this);

public ParameterHandler newParameterHandler(MappedStatement
mappedStatement, Object parameterObject, BoundSql boundSql) {
  ParameterHandler parameterHandler =
mappedStatement.getLang().createParameterHandler(mappedStatement,
parameterObject, boundSql);
  parameterHandler = (ParameterHandler)
interceptorChain.pluginAll(parameterHandler);
  return parameterHandler;
}

public ResultSetHandler newResultSetHandler(Executor executor,
MappedStatement mappedStatement, RowBounds rowBounds, ParameterHandler
```

```
parameterHandler,
    ResultHandler resultHandler, BoundSql boundSql) {
  ResultSetHandler resultSetHandler = new DefaultResultSetHandler(executor,
mappedStatement, parameterHandler, resultHandler, boundSql, rowBounds);
  resultSetHandler = (ResultSetHandler)
interceptorChain.pluginAll(resultSetHandler);
  return resultSetHandler;
}

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);
  statementHandler = (StatementHandler)
interceptorChain.pluginAll(statementHandler);
  return statementHandler;
}
```

2、创建一个defaultSqlSession对象，它里面包含configuration和executor(根据配置文件中的defaultEXecutorType创建出对应的Executor)



由SqlSessionFactoryBuilder创建SqlSessionFactory

```
public SqlSessionFactory build(InputStream inputStream, String environment,
Properties properties)
```

SqlSessionFactory的其中一个实现类

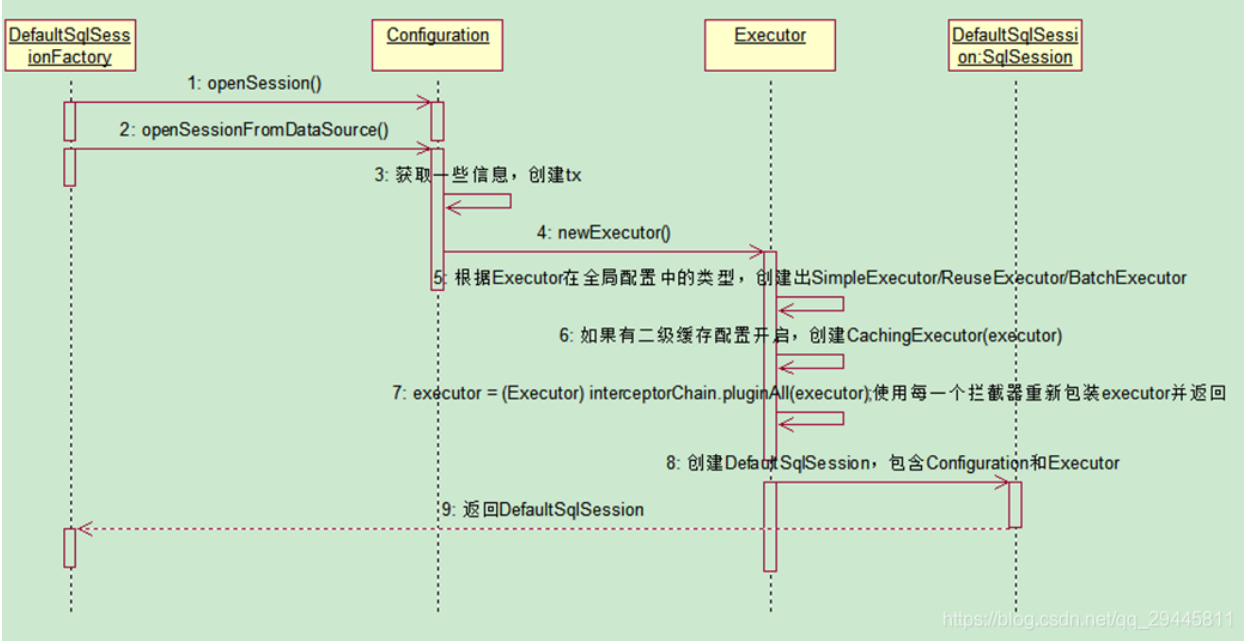
```
 public class DefaultSqlSessionFactory implements SqlSessionFactory{
        private final Configuration configuration;

    public DefaultSqlSessionFactory(Configuration configuration) {
      this.configuration = configuration;
    }

    @Override
    public SqlSession openSession() {
      return openSessionFromDataSource(configuration.getDefaultExecutorType(),
null, false);
    }

    ...

 }
```

SqlSession的其中一个实现类

```
public class DefaultSqlSession implements SqlSession {

  private final Configuration configuration;
  private final Executor executor;

  private final boolean autoCommit;
  private boolean dirty;
  private List<Cursor<?>> cursorList;

  public DefaultSqlSession(Configuration configuration, Executor executor, boolean
autoCommit) {
    this.configuration = configuration;
    this.executor = executor;
    this.dirty = false;
    this.autoCommit = autoCommit;
  }

  许多这种方法
  @Override
    public <E> List<E> selectList(String statement, Object parameter, RowBounds
rowBounds) {
      try {
        MappedStatement ms = configuration.getMappedStatement(statement);
        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();
      }
    }
    ...
```

调用的都是executor.xxx方法，Executor接口有六个实现类：BaseExecutor、BatchExecutor 、CachingExecutor、 CloseExecutor、 ReuseExecutor、 SimpleExecutor



3、defaultSqlSession.getMapper（）获取Mapper接口对应的MapperProxy，即调用Configuration类的getMapper方法获得sqlSession的代理对象

DefaultSqlSession类中

```
public <T> T getMapper(Class<T> type) {
  return configuration.getMapper(type, this);
}
```

Configuration类中有getMapper，调用mapperRegistry的getMapper方法：

```
protected final MapperRegistry mapperRegistry = new MapperRegistry(this);

public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
  return (type, sqlSession);
}
```

MapperRegistry类中有getMapper方法，返回sqlSession的代理对象：

```
public class MapperRegistry {

  private final Configuration config;
  private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap<>();
```

```
  public MapperRegistry(Configuration config) {
    this.config = config;
  }

  @SuppressWarnings("unchecked")
  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);
    }
  }
```

MapperProxyFactory类创建sqlSession的代理对象，如下：

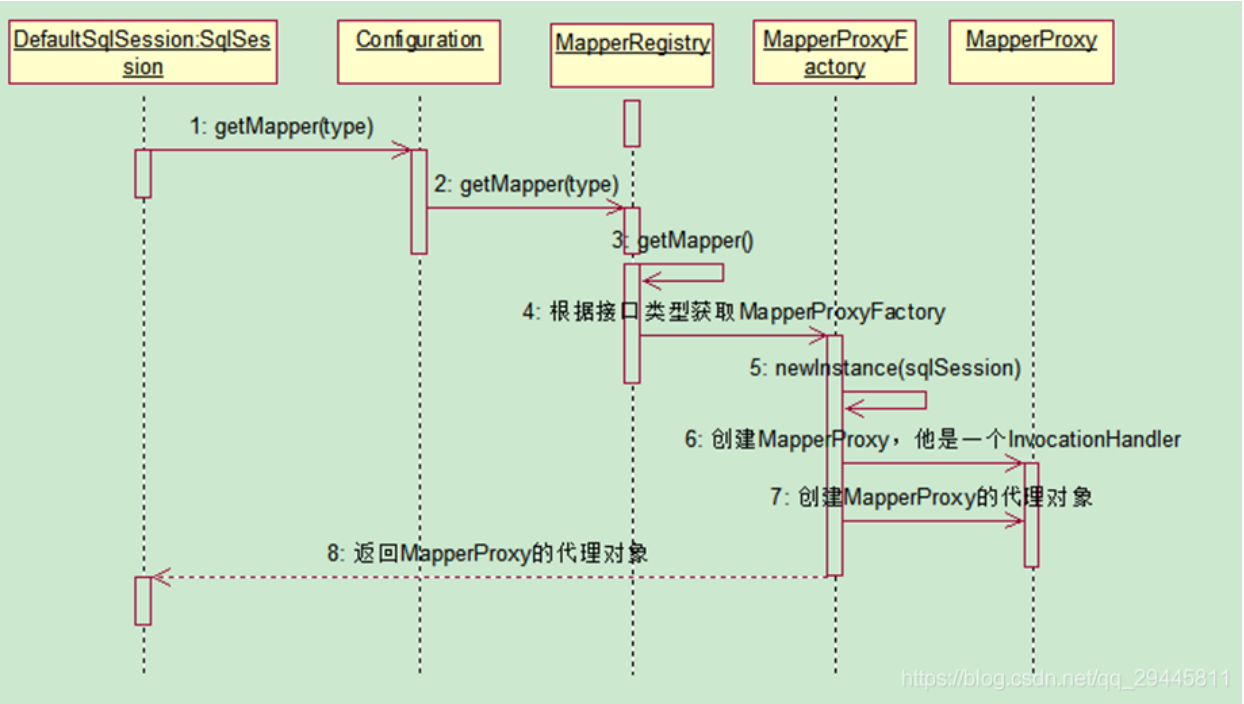
```
public class MapperProxyFactory<T> {

  private final Class<T> mapperInterface;
  private final Map<Method, MapperMethod> methodCache = new
ConcurrentHashMap<>();

  public MapperProxyFactory(Class<T> mapperInterface) {
    this.mapperInterface = mapperInterface;
  }

  public Class<T> getMapperInterface() {
    return mapperInterface;
  }

  public Map<Method, MapperMethod> getMethodCache() {
    return methodCache;
  }

  @SuppressWarnings("unchecked")
  protected T newInstance(MapperProxy<T> mapperProxy) {
    return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[]
{ mapperInterface }, mapperProxy);
  }

  public T newInstance(SqlSession sqlSession) {
    final MapperProxy<T> mapperProxy = new MapperProxy<>(sqlSession,
```

```
mapperInterface, methodCache);
    return newInstance(mapperProxy);
  }
}
```

其中mapperProxy就是实现了`InvocationHandler`接口的、别代理的目标类，这是典型的JDK动态代理

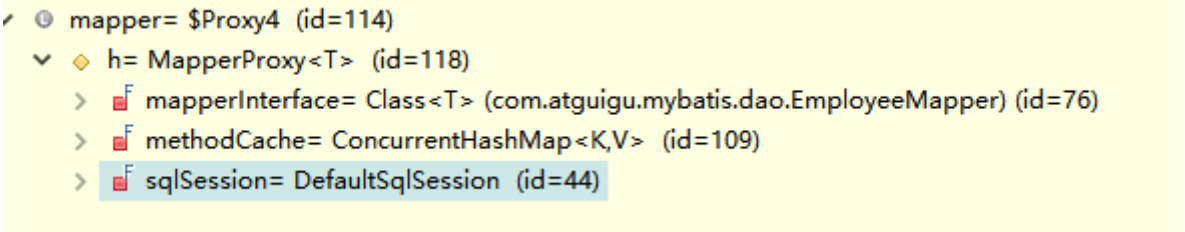
进入`public class MapperProxy<T> implements InvocationHandler, Serializable`可以看到：

```
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 (method.isDefault()) {
        return invokeDefaultMethod(proxy, method, args);
      }
    } catch (Throwable t) {
      throw ExceptionUtil.unwrapThrowable(t);
    }
    final MapperMethod mapperMethod = cachedMapperMethod(method);
    return mapperMethod.execute(sqlSession, args);
  }
```

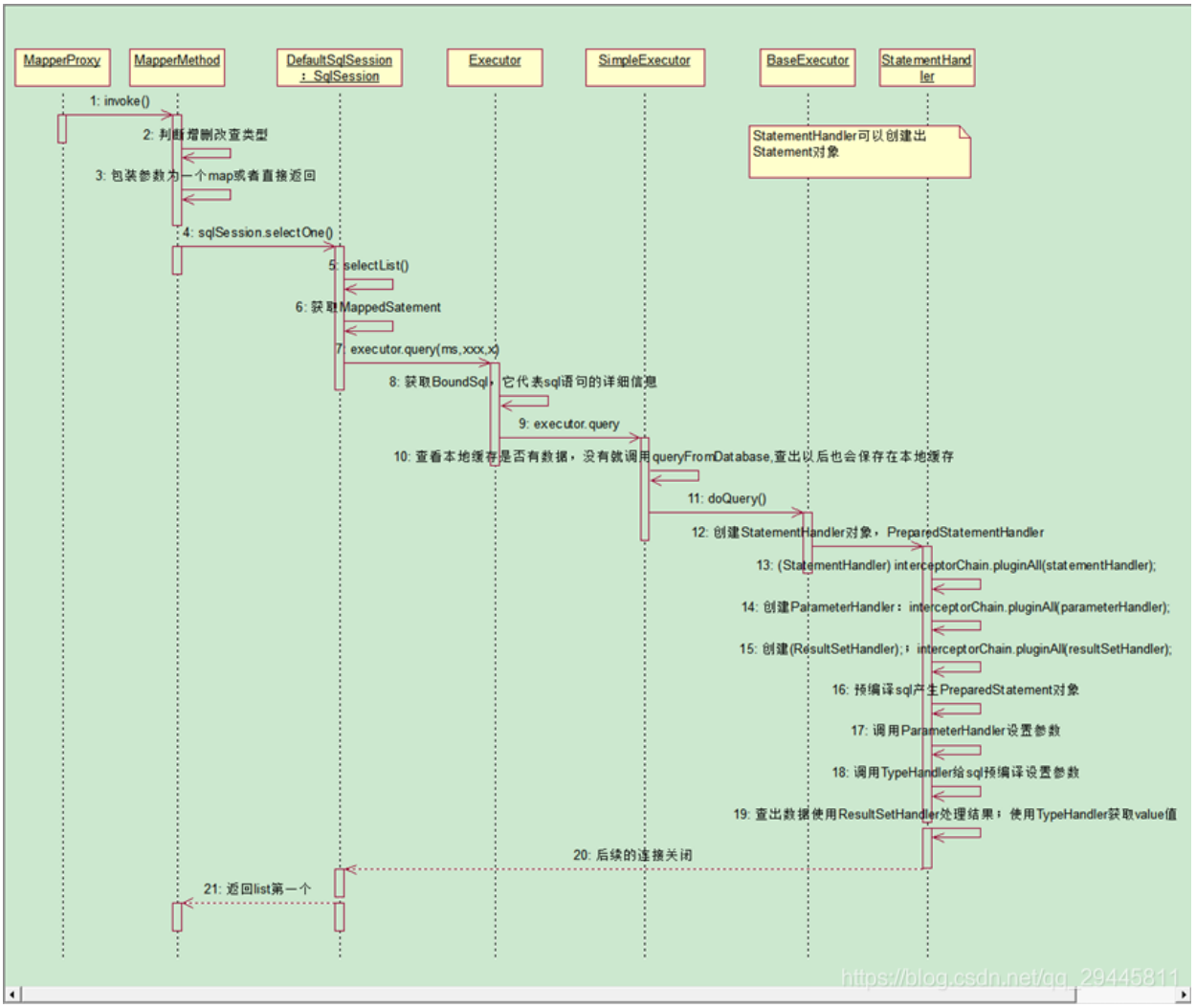


4、MapperProxy里面有defaultSqlSession

获取Mapper接口代理对象（MapperProxy）



返回getMapper接口的代理对象、包含了SqlSession对象



5、执行增删改查方法：

1、调用的是defaultSqlsesion的增删改查（会调用Executor的crud）

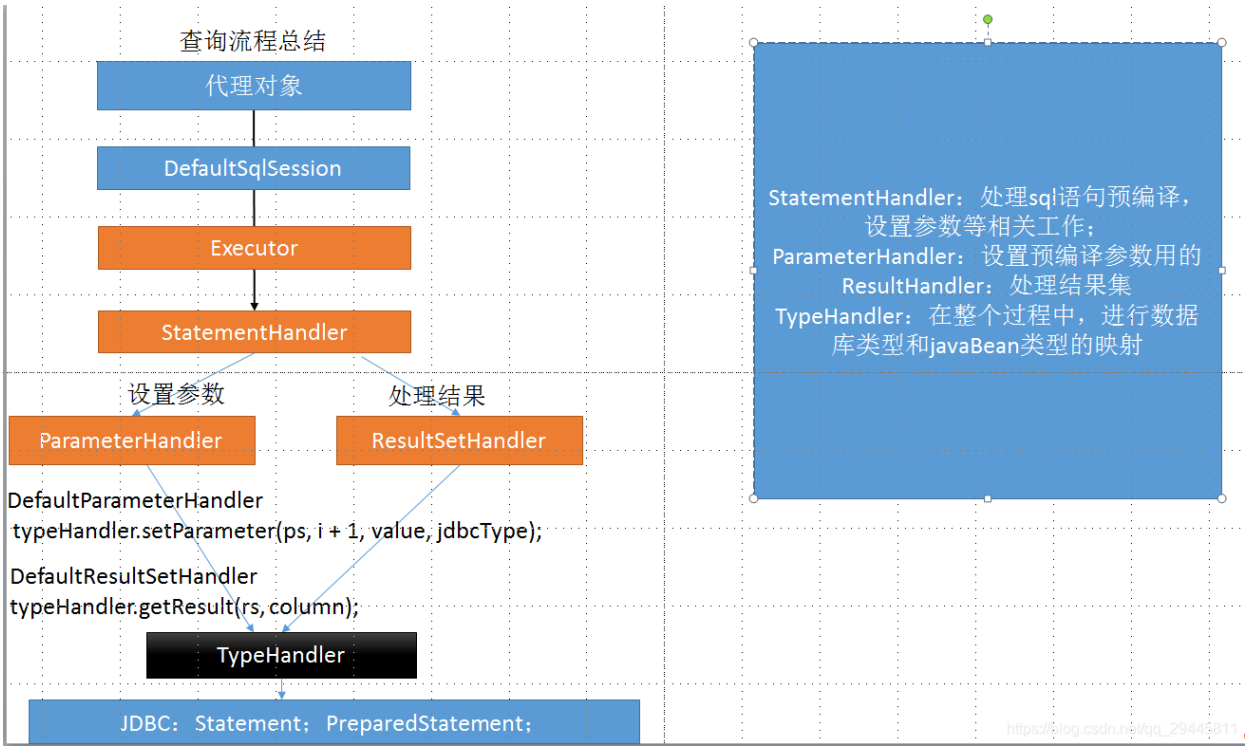
2、会创建一个statementhandler对象(同时也会创建出parameterHandler和resultSetHandler)

3、调用StatementHandler的prepareStatement()方法进行预编译handler.prepare()和参数设置handler.parameterize(stmt)

4、设置完成后调用StatementHandler的增删改查方法query()

5、参数预编译完成后使用resultSetHandler封装结果集

执行增删改查方法



注意：四大对象每个创建的时候都有一个interceptorChain.pluginAll()方法