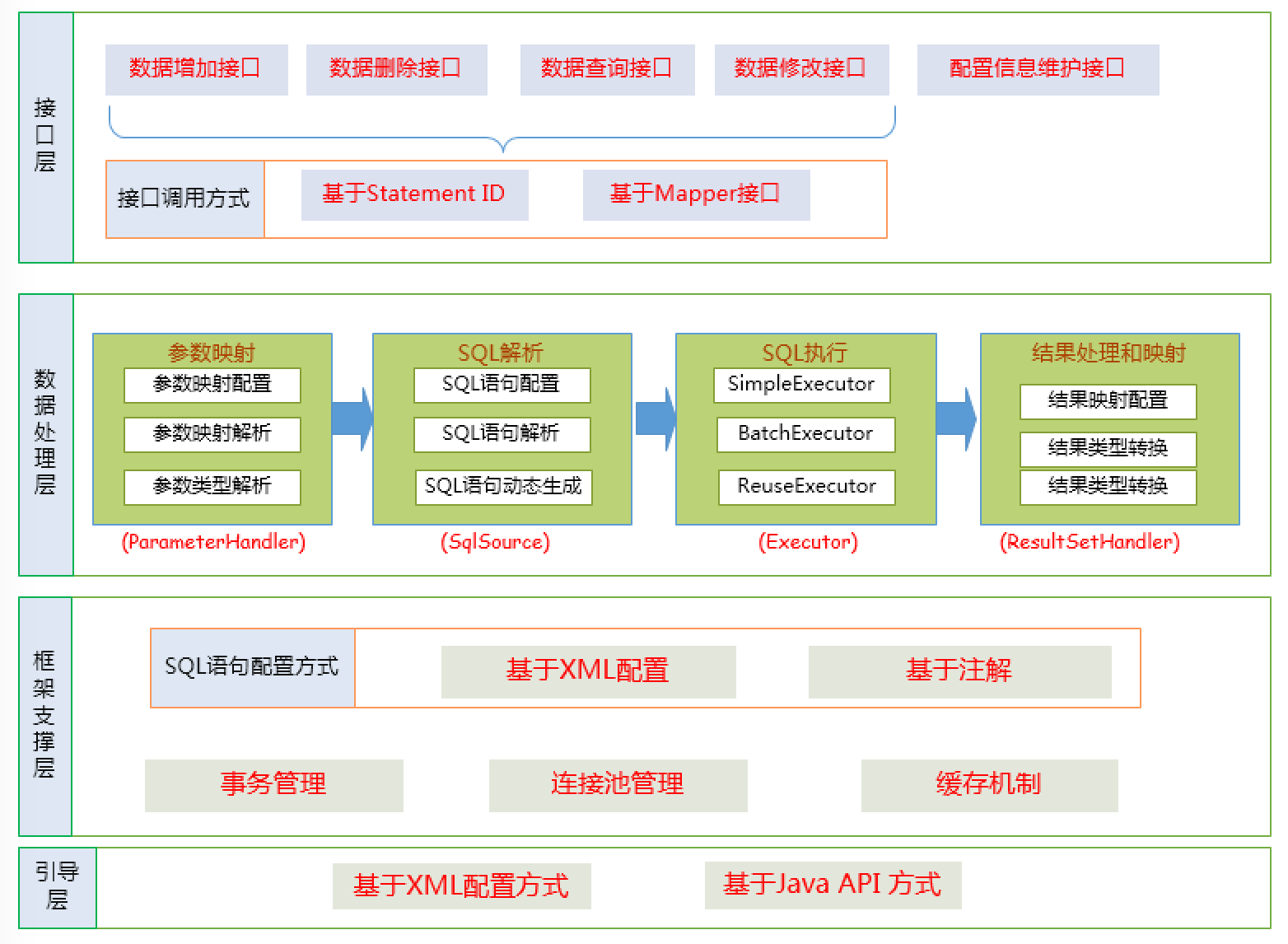
# Mybatis架构与原理

MyBatis功能架构设计



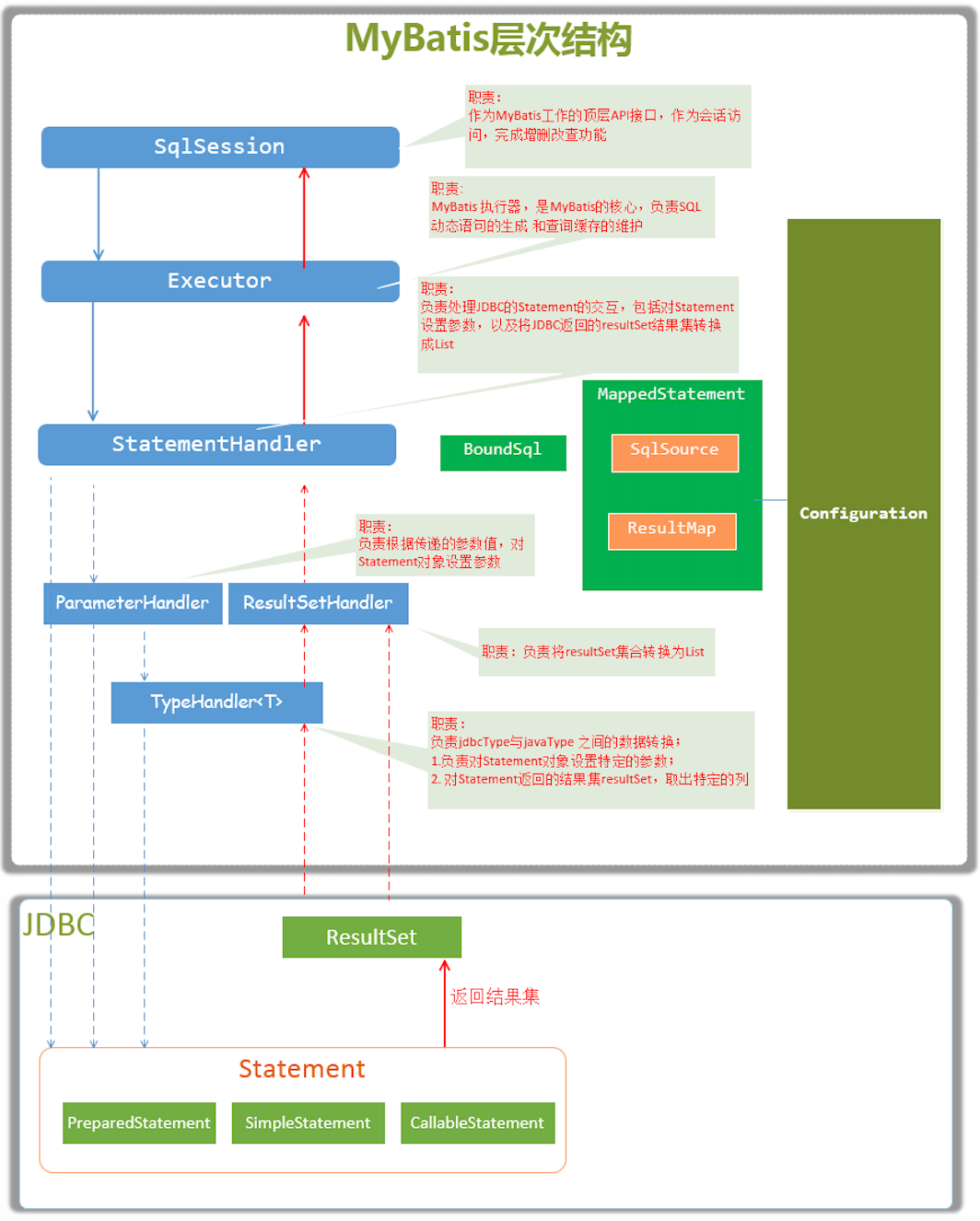
我们把Mybatis的功能架构分为三层：

**(1)API接口层**：提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理。

**(2)数据处理层**：负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作。

**(3)基础支撑层**：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑。

MyBatis成员层次&职责



1.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

2.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

3.StatementHandler 封装了JDBC Statement操作，负责对JDBCstatement的操作，如设置参数、将Statement结果集转换成List集合。

4.ParameterHandler 负责对用户传递的参数转换成JDBC Statement 所需要的参数

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

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

7.MappedStatement MappedStatement维护了一条<select|update|delete|insert>节点的封

8.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

9.BoundSql 表示动态生成的SQL语句以及相应的参数信息

10.Configuration MyBatis所有的配置信息都维持在Configuration对象之中

MyBatis核心类

1、SqlSessionFactoryBuilder

每一个MyBatis的应用程序的入口是SqlSessionFactoryBuilder。

它的作用是通过XML配置文件创建Configuration对象（当然也可以在程序中自行创建），然后通过build方法创建SqlSessionFactory对象。没有必要每次访问Mybatis就创建一次SqlSessionFactoryBuilder，通常的做法是创建一个全局的对象就可以了。示例程序如下：

private static SqlSessionFactoryBuilder sqlSessionFactoryBuilder;private static SqlSessionFactory sqlSessionFactory;

private static void init() throws IOException {

String resource = "mybatis-config.xml";

Reader reader = Resources.getResourceAsReader(resource);

sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();

sqlSessionFactory = sqlSessionFactoryBuilder.build(reader);}

org.apache.ibatis.session.Configuration 是mybatis初始化的核心。

mybatis-config.xml中的配置，最后会解析xml成Configuration这个类。

SqlSessionFactoryBuilder根据传入的数据流(XML)生成Configuration对象，然后根据Configuration对象创建默认的SqlSessionFactory实例。

2、SqlSessionFactory对象

由SqlSessionFactoryBuilder创建：

它的主要功能是创建SqlSession对象，和SqlSessionFactoryBuilder对象一样，没有必要每次访问Mybatis就创建一次SqlSessionFactory，通常的做法是创建一个全局的对象就可以了。SqlSessionFactory对象一个必要的属性是Configuration对象，它是保存Mybatis全局配置的一个配置对象，通常由SqlSessionFactoryBuilder从XML配置文件创建。这里给出一个简单的示例：

3、SqlSession

SqlSession对象的主要功能是完成一次数据库的访问和结果的映射，它类似于数据库的session概念，由于不是线程安全的，所以SqlSession对象的作用域需限制方法内。SqlSession的默认实现类是DefaultSqlSession，它有两个必须配置的属性：Configuration和Executor。Configuration前文已经描述这里不再多说。SqlSession对数据库的操作都是通过Executor来完成的。

SqlSession ：默认创建DefaultSqlSession 并且开启一级缓存，创建执行器 、赋值。

SqlSession有一个重要的方法getMapper，顾名思义，这个方式是用来获取Mapper对象的。什么是Mapper对象？根据Mybatis的官方手册，应用程序除了要初始并启动Mybatis之外，还需要定义一些接口，接口里定义访问数据库的方法，存放接口的包路径下需要放置同名的XML配置文件。

SqlSession的getMapper方法是联系应用程序和Mybatis纽带，应用程序访问getMapper时，Mybatis会根据传入的接口类型和对应的XML配置文件生成一个代理对象，这个代理对象就叫Mapper对象。应用程序获得Mapper对象后，就应该通过这个Mapper对象来访问Mybatis的SqlSession对象，这样就达到里插入到Mybatis流程的目的。

4、Executor

Executor对象在创建Configuration对象的时候创建，并且缓存在Configuration对象里。Executor对象的主要功能是调用StatementHandler访问数据库，并将查询结果存入缓存中（如果配置了缓存的话）。

5、StatementHandler

StatementHandler是真正访问数据库的地方，并调用ResultSetHandler处理查询结果。

6、ResultSetHandler

处理查询结果。

Mybatis初始化过程

任何框架的初始化，无非是加载自己运行时所需要的配置信息

MyBatis的上述配置信息会配置在XML配置文件中(mybatis-config.xml)

这些配置信息被加载进Configuration 对象，mybatis使用Configuration 对象作为一个所有配置信息的容器

Configuration对象的组织结构和XML配置文件的组织结构几乎完全一样（当然，Configuration对象的功能并不限于此，它还负责创建一些MyBatis内部使用的对象，如Executor等，这将在后续的文章中讨论）

MyBatis根据初始化好Configuration信息，这时候用户就可以使用MyBatis进行数据库操作了

**可以这么说，MyBatis初始化的过程，就是创建 Configuration对象的过程**。

MyBatis的初始化可以有两种方式：

基于XML配置文件：基于XML配置文件的方式是将MyBatis的所有配置信息放在XML文件中，MyBatis通过加载并XML配置文件，将配置文信息组装成内部的Configuration对象

基于Java API：这种方式不使用XML配置文件，需要MyBatis使用者在Java代码中，手动创建Configuration对象，然后将配置参数set 进入Configuration对象中

MyBatis基于XML配置文件创建Configuration对象的过程

mybatis的初始化就发生在SqlSessionFactory的创建过程中：

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

mybatis初始化要经过简单的以下几步：

1.调用SqlSessionFactoryBuilder对象的build(inputStream)方法；

2.SqlSessionFactoryBuilder会根据输入流inputStream等信息创建XMLConfigBuilder对象;

3.SqlSessionFactoryBuilder调用XMLConfigBuilder对象的parse()方法；

4.XMLConfigBuilder对象返回Configuration对象；

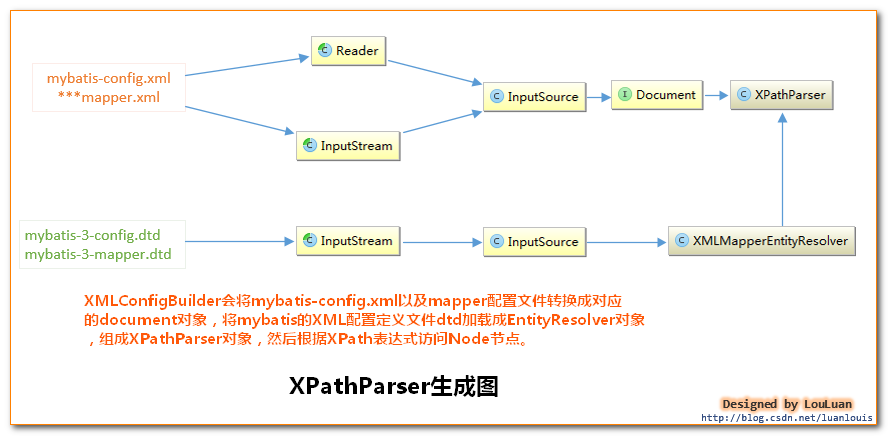
5.SqlSessionFactoryBuilder根据Configuration对象创建一个DefaultSessionFactory对象；

6.SqlSessionFactoryBuilder返回 DefaultSessionFactory对象给Client，供Client使用。

创建Configuration对象的过程

当SqlSessionFactoryBuilder执行build()方法，调用了XMLConfigBuilder的parse()方法，然后返回了Configuration对象。那么parse()方法是如何处理XML文件，生成Configuration对象的呢？

1. XMLConfigBuilder会将XML配置文件的信息转换为Document对象，而XML配置定义文件DTD转换成XMLMapperEntityResolver对象，然后将二者封装到XpathParser对象中，XpathParser的作用是提供根据Xpath表达式获取基本的DOM节点Node信息的操作。如图所示：



2.之后XMLConfigBuilder调用parse()方法：会从XPathParser中取出 <configuration>节点对应的Node对象，然后解析此Node节点的子Node：properties, settings, typeAliases,typeHandlers, objectFactory, objectWrapperFactory, plugins, environments,databaseIdProvider, mappers

3.然后将这些值解析出来设置到Configuration对象中。

解析子节点的过程这里就不一一介绍了，用户可以参照MyBatis源码仔细揣摩，我们就看上述的environmentsElement(root.evalNode("environments")); 方法是如何将environments的信息解析出来，设置到Configuration对象中的：

/\*

解析environments节点，并将结果设置到Configuration对象中

注意：创建envronment时，如果SqlSessionFactoryBuilder指定了特定的环境（即数据源）；

则返回指定环境（数据源）的Environment对象，否则返回默认的Environment对象；

这种方式实现了MyBatis可以连接多数据源

\*/private void environmentsElement(XNode context) throws Exception{

if (context != null)

{

if (environment == null)

{

environment = context.getStringAttribute("default");

}

for (XNode child : context.getChildren())

{

String id = child.getStringAttribute("id");

if (isSpecifiedEnvironment(id))

{

//1.创建事务工厂 TransactionFactory

TransactionFactory txFactory = transactionManagerElement(child.evalNode("transactionManager"));

DataSourceFactory dsFactory = dataSourceElement(child.evalNode("dataSource"));

//2.创建数据源DataSource

DataSource dataSource = dsFactory.getDataSource();

//3. 构造Environment对象

Environment.Builder environmentBuilder = new Environment.Builder(id)

.transactionFactory(txFactory)

.dataSource(dataSource);

//4. 将创建的Envronment对象设置到configuration 对象中

configuration.setEnvironment(environmentBuilder.build());

}

}

}

}

4.返回Configuration对象

初始化过程中涉及到的设计模式-----Builder模式

Builder模式应用1： SqlSessionFactory的创建

Builder模式应用2： 数据库连接环境Environment对象的创建

MyBatis的配置信息，大概包含以下信息，其高层级结构如下：

<configuration> 配置

<properties> 属性

<settings> 设置

<typeAliases> 类型命名

<typeHandlers> 类型处理器

<objectFactory> 对象工厂

<plugins> 插件

<environments> 环境

<environment> 环境变量

<transactionManager> 事务管理器

<dataSource> 数据源

</configuration> 配置

Mybatis执行sql的过程

1.最佳实践中，通常一个 XML 映射文件，都会写一个 Mapper 接口与之对应。请问，这个 Mapper 接口的工作原理是什么？Mapper 接口里的方法，参数不同时，方法能重载吗？

Mapper 接口，对应的关系如下：

接口的全限名，就是映射文件中的 "namespace" 的值。

接口的方法名，就是映射文件中 MappedStatement 的 "id" 值。

接口方法内的参数，就是传递给 SQL 的参数。

Mapper 接口是没有实现类的，当调用接口方法时，接口全限名 + 方法名拼接字符串作为 key 值，可唯一定位一个对应的 MappedStatement 。举例：com.mybatis3.mappers.StudentDao.findStudentById ，可以唯一找到 "namespace" 为 com.mybatis3.mappers.StudentDao 下面 "id" 为 findStudentById 的 MappedStatement 。

总结来说，在 Mybatis 中，每一个 <select />、<insert />、<update />、<delete /> 标签，都会被解析为一个 MappedStatement 对象。

另外，Mapper 接口的实现类，通过 MyBatis 使用 JDK Proxy 自动生成其代理对象 Proxy ，而代理对象 Proxy 会拦截接口方法，从而“调用”对应的 MappedStatement 方法，最终执行 SQL ，返回执行结果。整体流程如下图：

1. Mapper接口通过jdk动态代理来实现的，此处是一个demo展示  
   描述：

我们知道使用Mybatis的四个步骤

获取SqlSessionFactory对象

获取sqlSession对象

获取接口的代理对象（MapperProxy）

执行增删改查方法

具体到获取接口的代理对象主要有以下几步：

1.调用SqlSession的getMapper()方法：内部委托给了Configuration对象去进行处理

//调用的是Configuration的getMapper()方法，参数type为传入的接口全类名

//什么都不做，直接去configuration中找

@Override

public <T> T getMapper(Class<T> type) {

return configuration.<T>getMapper(type, this);

}

2.SqlSession把包袱甩给了Configuration，接下来就看看Configuration，其内部委托给MapperRegistry处理

//在Configuration中，又调用了MapperRegistry的getMapper()方法

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

//烫手的山芋，俺不要，你找mapperRegistry去要

return mapperRegistry.getMapper(type, sqlSession);

}

3.MapperRegistry.getMapper方法进行处理

首先根据Mapper类型为其创建一个MapperProxyFactory

接着mapperProxyFactory.newInstance(sqlSession) 去创建一个MapperProxy

最后调用Proxy.newProxyInstance方法生成代理类返回

注意在最终指向Mapper接口的方法时实际上是执行的依据该Mapper创建的MapperProxy中invoke方法，该方法中的逻辑：

//这里会拦截Mapper接口的所有方法

public Object **invoke**(Object proxy, Method method, Object[] args) throws Throwable {

//如果是Object中定义的方法，直接执行。如toString(),hashCode()等

if (Object.class.equals(method.getDeclaringClass())) {

try {

return method.invoke(this, args);//

} catch (Throwable t) {

throw ExceptionUtil.unwrapThrowable(t);

}

}

//其他Mapper接口定义的方法交由mapperMethod来执行

final MapperMethod mapperMethod = cachedMapperMethod(method);

return mapperMethod.execute(sqlSession, args);

}

-------------------------------------------->下面通过一个demo实现类似功能进行展示<----------------------------------------------------

1.新建两个Mapper dao接口

public interface OrderDao {

int add(String str);

int update(String str);

}

public interface UserDao {

int add(String str);

int update(String str);

}

2.创建Mapper接口的代理类 实现InvocationHandler接口

import java.io.Serializable;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.util.Map;

/\*\*

\* @Author 了个

\* @date 2020/4/7 15:36

\*/

public class MapperProxy<T> implements InvocationHandler, Serializable {

private final Class<T> mapperInterface;

private final Map<Method, MapperMethod> mapperMethodCache;

public MapperProxy(Class<T> mapperInterface, Map<Method, MapperMethod> mapperMethodCache) {

this.mapperInterface = mapperInterface;

this.mapperMethodCache = mapperMethodCache;

}

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

//如果是Object中定义的方法，直接执行。如toString(),hashCode()等

if (Object.class.equals(method.getDeclaringClass())) {

try {

return method.invoke(this, args);

} catch (Throwable t) {

}

}

//其他Mapper接口定义的方法交由mapperMethod来执行

final MapperMethod mapperMethod = cachedMapperMethod(method);

return mapperMethod.execute(args);

}

private MapperMethod cachedMapperMethod(Method method) {

MapperMethod mapperMethod = mapperMethodCache.get(method);

if (mapperMethod == null) {

mapperMethod = new MapperMethod(method, this);

mapperMethodCache.put(method, mapperMethod);

}

return mapperMethod;

}

}

3.创建代理处理的方法类

import java.lang.annotation.Annotation;

import java.lang.reflect.Method;

/\*\*

\* @Author 了个

\* @date 2020/4/7 15:35

\*/

public class MapperMethod {

private final Method method;

private final MapperProxy mapperProxy;

public MapperMethod(Method method, MapperProxy mapperProxy) {

this.method = method;

this.mapperProxy = mapperProxy;

}

public Object execute(Object[] args){

System.out.println("MapperProxy: " + mapperProxy);

System.out.println("execute method {" + method.getName() + "}" );

// 读取mapper.xml中的语句块，并将其作为SqlStatement执行，

// 方法名获取sql语句，参数通过代理的Invocation等获得

// 有注解则处理注解中的参数

// 此处只是简单的模拟一下

final Annotation[] annotations = method.getAnnotations();

for (Annotation annotation : annotations) {

Class<? extends Annotation> aClass = annotation.annotationType();

if (aClass.equals(Insert.class)) {

System.out.println("execute insert {" + args[0] + "} completed");

}

if (aClass.equals(Update.class)) {

System.out.println("execute update {" + args[0] + "} completed");

}

}

return 1;

}

}

4.创建代理工厂类

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

/\*\*

\* @Author 了个

\* @date 2020/4/7 15:35

\*/

public class MapperProxyFactory<T> {

private final Class<T> mapperInterface;

private final Map<Method, MapperMethod> mapperMethodCache = new ConcurrentHashMap<>();

public MapperProxyFactory(Class<T> mapperInterface) {

this.mapperInterface = mapperInterface;

}

public Class<T> getMapperInterface() {

return mapperInterface;

}

public Map<Method, MapperMethod> getMethodCache() {

return mapperMethodCache;

}

protected T newInstance(MapperProxy<T> mapperProxy){

return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[]{mapperInterface}, mapperProxy);

}

public T newInstance(){

final MapperProxy<T> mapperProxy = new MapperProxy<>(mapperInterface, mapperMethodCache);

return newInstance(mapperProxy);

}

}

@Target({ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

public @interface Insert {

}

@Target({ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

public @interface Update {

boolean required() default true;

}

SQL执行流程

上面，咱们拿到了MapperProxy, 每个MapperProxy对应一个dao接口， 那么咱们在使用的时候，MapperProxy是怎么做的呢？ 源码奉上：

MapperProxy:

/\*\*

\* MapperProxy在执行时会触发此方法

\*/

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

if (Object.class.equals(method.getDeclaringClass())) {

try {

return method.invoke(this, args);

} catch (Throwable t) {

throw ExceptionUtil.unwrapThrowable(t);

}

}

final MapperMethod mapperMethod = cachedMapperMethod(method);

//二话不说，主要交给MapperMethod自己去管

return mapperMethod.execute(sqlSession, args);

}

MapperMethod:

/\*\*

\* 看着代码不少，不过其实就是先判断CRUD类型，然后根据类型去选择到底执行sqlSession中的哪个方法，绕了一圈，又转回sqlSession了

\* @param sqlSession

\* @param args

\* @return

\*/

public Object execute(SqlSession sqlSession, Object[] args) {

Object result;

if (SqlCommandType.INSERT == command.getType()) {

Object param = method.convertArgsToSqlCommandParam(args);

result = rowCountResult(sqlSession.insert(command.getName(), param));

} else if (SqlCommandType.UPDATE == command.getType()) {

Object param = method.convertArgsToSqlCommandParam(args);

result = rowCountResult(sqlSession.update(command.getName(), param));

} else if (SqlCommandType.DELETE == command.getType()) {

Object param = method.convertArgsToSqlCommandParam(args);

result = rowCountResult(sqlSession.delete(command.getName(), param));

} else if (SqlCommandType.SELECT == command.getType()) {

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 {

Object param = method.convertArgsToSqlCommandParam(args);

result = sqlSession.selectOne(command.getName(), param);

}

} else {

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;

既然又回到SqlSession了， 那么咱们就看看SqlSession的CRUD方法了，为了省事，还是就选择其中的一个方法来做分析吧。这儿，咱们选择了selectList方法：

public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {

try {

MappedStatement ms = configuration.getMappedStatement(statement);

//CRUD实际上是交给Excetor去处理， excutor其实也只是穿了个马甲而已，小样，别以为穿个马甲我就不认识你嘞！

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();

}

}

然后，通过一层一层的调用，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:

复制代码

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();

StatementHandler handler = configuration.newStatementHandler(wrapper, ms, parameter, rowBounds, resultHandler, boundSql);

stmt = prepareStatement(handler, ms.getStatementLog());

//StatementHandler封装了Statement, 让 StatementHandler 去处理

return handler.<E>query(stmt, resultHandler);

} finally {

closeStatement(stmt);

}

}

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）, 看看它使怎么去处理的：

public <E> List<E> query(Statement statement, ResultHandler resultHandler) throws SQLException {

//到此，原形毕露， PreparedStatement, 这个大家都已经滚瓜烂熟了吧

PreparedStatement ps = (PreparedStatement) statement;

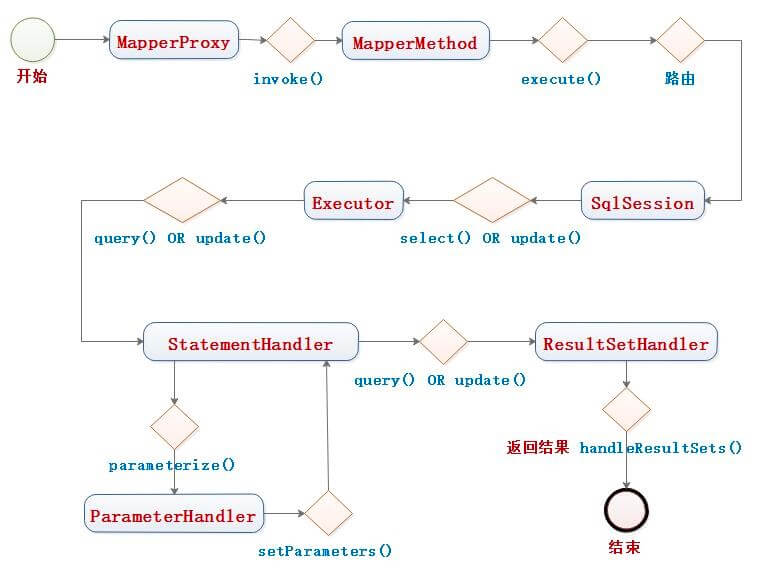
ps.execute();

//结果交给了ResultSetHandler 去处理

return resultSetHandler.<E> handleResultSets(ps);

}

到此， 一次sql的执行流程就完了。



3.说说MyBatis拦截器原理，即MyBatis插件原理

MyBatis提供了一种插件(plugin)的功能，虽然叫做插件，但其实这是拦截器功能。那么拦截器拦截MyBatis中的哪些内容呢？

我们进入官网看一看：

MyBatis 允许你在已映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis 允许使用插件来拦截的方法调用包括：

Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)

ParameterHandler (getParameterObject, setParameters)

ResultSetHandler (handleResultSets, handleOutputParameters)

StatementHandler (prepare, parameterize, batch, update, query)

我们看到了可以拦截Executor接口的部分方法，比如update，query，commit，rollback等方法，还有其他接口的一些方法等。

总体概括为：

拦截执行器的方法

拦截参数的处理

拦截结果集的处理

拦截Sql语法构建的处理

# Mybatis数据源和连接池

有个问题：数据源中创建的connection要不要设置成autocommit：

JDBC中，一个connection被创建时，默认是auto-commit模式，也即一个sql statement作为一个事务，执行完成后自动commit。如果支持多个statement组成一个事务，则要禁止auto-commit模式。

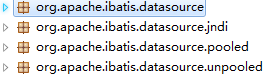
但是，MyBatis默认创建的Connection好像autoCommit是false

对于ORM框架而言，数据源的组织是一个非常重要的一部分，这直接影响到框架的性能问题。本文将通过对MyBatis框架的数据源结构进行详尽的分析，并且深入解析MyBatis的连接池。

本文首先会讲述MyBatis的数据源的分类，然后会介绍数据源是如何加载和使用的。紧接着将分类介绍UNPOOLED、POOLED和JNDI类型的数据源组织；期间我们会重点讲解POOLED类型的数据源和其实现的连接池原理。

一、MyBatis数据源DataSource分类

MyBatis数据源实现是在以下四个包中：



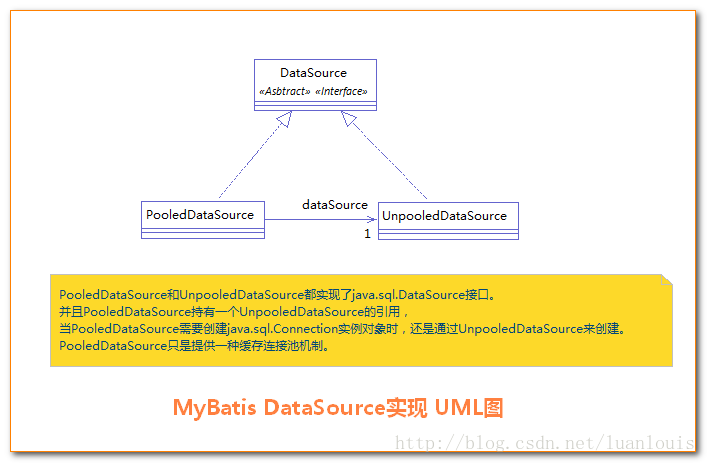
MyBatis把数据源DataSource分为三种：

1、UNPOOLED 不使用连接池的数据源

2、POOLED 使用连接池的数据源

3、JNDI 使用JNDI实现的数据源

相应地，MyBatis内部分别定义了实现了java.sql.DataSource接口的UnpooledDataSource，PooledDataSource类来表示UNPOOLED、POOLED类型的数据源。 如下图所示：



二、数据源DataSource的创建过程

1.MyBatis数据源DataSource对象的创建发生在**MyBatis初始化的过程**中。

初始过程中可以通过配置去设置 数据源中出来的Connection是否是自动提交，如果没有设置的话，那就默认为JDBC创建的原生Connection的自动提交（JDBC创建的Connection是默认自动提交的）

2.MyBatis是通过工厂模式来创建数据源DataSource对象的，MyBatis定义了抽象的工厂接口:org.apache.ibatis.datasource.DataSourceFactory,通过其getDataSource()方法返回数据源DataSource：

根据从配置文件中的<dataSource type="DRUID"> 中定义的type ，不同类型的type，则有对应的以下dataSource工厂，如下为框架的三个DataSourceFactory：

POOLED PooledDataSourceFactory

UNPOOLED UnpooledDataSourceFactory

JNDI JndiDataSourceFactory

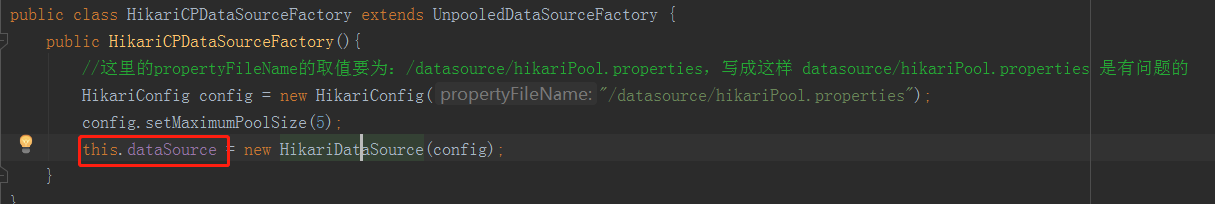
通过反射newinstance获取到具体的数据源工厂实例

3.MyBatis创建了DataSource实例后，会将其放到**Configuration对象内的Environment对象中**， 供以后使用。

4.其他第三方数据源的接入：

1.通过实现DataSourceFactory接口，实现setProperties方法（属性在mybatis配置文件中自动读取进来），实现getDataSource方法，初始返回DataSource

2.也可以通过继承UnpooledDataSourceFactory来实现，具体代码demo如下：



关键所在在于this.dataSource的赋值

在构造enviroment对象的时候有：

DataSourceFactory dsFactory = dataSourceElement(child.evalNode("dataSource"));

DataSource dataSource = dsFactory.getDataSource();

5.DataSource什么时候创建Connection对象

当我们需要创建SqlSession对象并需要执行SQL语句时，这时候MyBatis才会去调用dataSource对象来创建java.sql.Connection对象。也就是说，java.sql.Connection对象的创建一直延迟到执行SQL语句的时候。

不使用连接池的UnpooledDataSource

当 的type属性被配置成了”UNPOOLED”，MyBatis首先会实例化一个UnpooledDataSourceFactory工厂实例，然后通过.getDataSource()方法返回一个UnpooledDataSource实例对象引用，我们假定为dataSource。

6.使用UnpooledDataSource的getConnection(),每调用一次就会产生一个新的Connection实例对象。

UnpooledDataSource会做以下事情：

1.初始化驱动： 判断driver驱动是否已经加载到内存中，如果还没有加载，则会动态地加载driver类，并实例化一个Driver对象，使用DriverManager.registerDriver()方法将其注册到内存中，以供后续使用。

2.创建Connection对象： 使用DriverManager.getConnection()方法创建连接。

3.配置Connection对象： 设置是否自动提交autoCommit和隔离级别isolationLevel。

4.返回Connection对象。

7.使用了连接池的PooledDataSource

PooledDataSource将java.sql.Connection对象包裹成PooledConnection对象放到了PoolState类型的容器中维护。 MyBatis将连接池中的PooledConnection分为两种状态： 空闲状态（idle）和活动状态(active)，这两种状态的PooledConnection对象分别被存储到PoolState容器内的idleConnections和activeConnections两个List集合中：

idleConnections:空闲(idle)状态PooledConnection对象被放置到此集合中，表示当前闲置的没有被使用的PooledConnection集合，调用PooledDataSource的getConnection()方法时，会优先从此集合中取PooledConnection对象。当用完一个java.sql.Connection对象时，MyBatis会将其包裹成PooledConnection对象放到此集合中。

activeConnections:活动(active)状态的PooledConnection对象被放置到名为activeConnections的ArrayList中，表示当前正在被使用的PooledConnection集合，调用PooledDataSource的getConnection()方法时，会优先从idleConnections集合中取PooledConnection对象,如果没有，则看此集合是否已满，如果未满，PooledDataSource会创建出一个PooledConnection，添加到此集合中，并返回。

PooledDataSource 的getConnection()方法获取Connection对象的实现，具体是popConnection()方法，会从连接池中返回一个可用的PooledConnection对象，然后再调用getProxyConnection()方法最终返回Conection对象。

popConnection()方法到底做了什么：

1.先看是否有空闲(idle)状态下的PooledConnection对象，如果有，就直接返回一个可用的PooledConnection对象；否则进行第2步。

2.查看活动状态的PooledConnection池activeConnections是否已满；如果没有满，则创建一个新的PooledConnection对象，然后放到activeConnections池中，然后返回此PooledConnection对象；否则进行第三步；

3.看最先进入activeConnections池中的PooledConnection对象是否已经过期：如果已经过期，从activeConnections池中移除此对象，然后创建一个新的PooledConnection对象，添加到activeConnections中，然后将此对象返回；否则进行第4步。

4.线程等待，循环2步

8.java.sql.Connection对象的回收

怎样实现Connection对象调用了close()方法，而实际是将其添加到连接池中

MyBatis的PooledDataSource的PoolState内部维护的对象是PooledConnection类型的对象，而PooledConnection则是对真正的数据库连接java.sql.Connection实例对象的包裹器。

PooledConnection对象内持有一个真正的数据库连接java.sql.Connection实例对象和一个java.sql.Connection的代理：

PooledConenction实现了InvocationHandler接口，并且，proxyConnection对象也是根据这个它来生成的代理对象

我们使用了pooledDataSource.getConnection()返回的Connection对象的close()方法时，不会调用真正Connection的close()方法，而是将此Connection对象放到连接池中。

这里分析下MyBatis下配置事务的类型为JDBC的情况下，针对dataSource不同配置下的数据源Connection加载的流程是怎样的？

<transactionManager type="JDBC" />

三大步：

第一步：new SqlSessionFactoryBuilder().build(reader) 也就是mybatis初始化配置的时候

第二步：SqlSession session = sessionFactory.openSession(isAutoCommit) 新建会话连接的时候

第三步：利用MapperProxy去具体执行接口的时候#MapperProxy类中的invoke方法中的mapperMethod.execute

**1.<dataSource type="POOLED">**

**第一步：new SqlSessionFactoryBuilder().build(reader) 也就是mybatis初始化配置的时候**

1.1.SqlSessionFactoryBuilder.build

1.2.XMLConfigBuilder#parse() ----> Configuration

parseConfiguration(XNode root)

environmentsElement(XNode context)

结果：返回configuration对象实例

//解析 创建事务工厂实例，这里是JdbcTransactionFactory  
TransactionFactory txFactory = transactionManagerElement(child.evalNode("transactionManager"));  
//解析 创建数据源工厂实例，这里是PooledDataSourceFactory  
DataSourceFactory dsFactory = dataSourceElement(child.evalNode("dataSource"));  
//从上面创建的数据源工厂中获取创建 DataSource 对象：PooledDataSource  
DataSource dataSource = dsFactory.getDataSource();  
//构建 Environment 对象，并设置到 configuration 中（会在enviroment对象中包含【事务工厂实例 和 数据源实例】）  
Environment.Builder environmentBuilder = new Environment.Builder(id)  
 .transactionFactory(txFactory)  
 .dataSource(dataSource);  
configuration.setEnvironment(environmentBuilder.build());

1.3.最终生成SqlSessionFactory对象，内涵configuration实例

**第二步：SqlSession session = sessionFactory.openSession(isAutoCommit) 新建会话连接的时候**

2.1在第一步创建的SqlSessionFactory基础上调用newSession

2.2 DefaultSqlSessionFactory#openSession

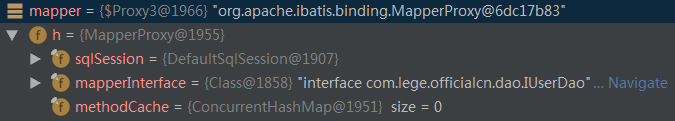
2.3 DefaultSqlSessionFactory#openSessionFromDataSource

Transaction tx = null;  
try {  
 final Environment environment = configuration.getEnvironment();  
 */\*\*  
 \* 1.创建事务对象工厂：这里是JdbcTransactionFactory实例  
 \* 2.利用JdbcTransactionFactory创建事务Transaction实例  
 \*/* final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment);  
 */\*\*  
 \* 调用JdbcTransactionFactory#newTransaction 方法  
 \* 内部调用new JdbcTransaction(ds, level, autoCommit)  
 \* 其中autoCommit是openSession时传过来的值，level也是传过来的值，  
 \* 理论上，这两个值可以设置也可以不设置  
 \* 这样JdbcTransaction实例中的实例变量将会被赋值：为后续处理埋下伏笔，  
 \* 其中1是从数据源处获得的Connection实例的autoCommit属性将会被这里传入的值给覆盖  
 \* dataSource = ds;  
 \* level = desiredLevel;  
 \* autoCommit = desiredAutoCommit;  
 \*  
 \*/* tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);  
 */\*\*很大一个知识点，mybatis缓存，这里是通过装饰器模式来实现的TODO  
 \* executor = new SimpleExecutor(configuration, transaction);  
 \* executor = new CachingExecutor(executor);  
 \* 这里返回的是CachingExecutor实例  
 \*/* final Executor executor = configuration.newExecutor(tx, execType);  
 */\*\*  
 \* 最后返回创建一个SqlSession的默认实现DefaultSqlSession实例，  
 \* 其中包含一下属性：  
 \* this.configuration = configuration;  
 \* this.executor = executor;  
 \* this.dirty = false;  
 \* this.autoCommit = autoCommit;  
 \*/* return new DefaultSqlSession(configuration, executor, autoCommit);  
} catch (Exception e) {

2.4 最终返回DefaultSqlSession实例

**第三步：利用MapperProxy去具体执行接口的时候#MapperProxy类中的invoke方法中的mapperMethod.execute**

2.1 通过上一步返回的SqlSession，来获取对应Mapper接口的代理实现类MapperProxy



2.2 MapperProxy<T>#invoke

2.3 MapperMethod#execute(sqlSession, args)

根据SQL的不同操作类型调用SqlSession的不同的查询接口

2.4 MapperMethod#executeForMany(sqlSession, args)

2.5 DefaultSqlSession#selectList

try {  
 */\*\*  
 \* MappedStatement维护了一条<select|update|delete|insert>节点的封装  
 \* 其id值就是：接口方法的全限定名，所以这里也就说明了Mapper接口不能重载  
 \*/* MappedStatement ms = configuration.getMappedStatement(statement);  
 return executor.query(ms, wrapCollection(parameter), rowBounds, Executor.*NO\_RESULT\_HANDLER*);

2.6 CachingExecutor#query方法

构建BoundSql 和 CacheKey

2.7 BaseExecutor#query方法

--> #queryFromDatabase

**2.8 SimpleExecutor#doQuery方法 会走完下面所有逻辑，返回查询结果List**

该方法中有个很关键的处理

**stmt = prepareStatement(handler, ms.getStatementLog());**

SimpleExecutor#prepareStatement方法

BaseExecutor#getConnection方法

JdbcTransaction#getConnection方法

openConnection()

protected void openConnection() throws SQLException {  
 if (*log*.isDebugEnabled()) {  
 *log*.debug("Opening JDBC Connection");  
 }  
 */\*\*这里从不同数据源获取Connection连接  
 \* dataSource 是MyBatis初始化时，更具具体的不同的数据源实现所初始化的DataSource  
 \* 这里获取到的Connection的autoCommit是默认的 自动提交 即true  
 \*/* connection = dataSource.getConnection();  
 if (level != null) {  
 connection.setTransactionIsolation(level.getLevel());  
 }  
 */\*\*  
 \* 这个地方很重，connection的autoCommit会被修改为JdbcTransaction的 autoComit值  
 \* 首先JbdcTransaction中的autoCommit属性值 是mybatis初始化时设置进来的 我们设置的为true  
 \* 这里会重新设置我们传进来的值用来覆盖JDBC中原生的Conneciton autoCommit的true的属性值  
 \*/* setDesiredAutoCommit(autoCommit);  
}

如下为 connection = dataSource.getConnection(); 方法中逻辑

PooledDataSource#popConnection

TODO 很大一块关于MyBatis自身提供的连接池的问题

unpooledDataSource#doGetConnection

private Connection doGetConnection(Properties properties) throws SQLException {  
 // <1> 初始化 Driver  
 initializeDriver();  
 // <2> 获得 Connection 对象 JDBC中，一个connection被创建时，默认是auto-commit模式  
 Connection connection = DriverManager.*getConnection*(url, properties);  
 // <3> 配置 Connection 对象  
 configureConnection(connection);  
 return connection;  
}

unpooledDataSource#configurationConnection

*/\*\*  
 \* 配置 Connection 对象  
 \** ***@param*** *conn  
 \** ***@throws*** *SQLException  
 \*/*private void configureConnection(Connection conn) throws SQLException {  
 if (defaultNetworkTimeout != null) {  
 conn.setNetworkTimeout(Executors.*newSingleThreadExecutor*(), defaultNetworkTimeout);  
 }  
 // 设置自动提交 从PooledDataSource过来的autoCommit和事务隔离级别都是null  
 if (autoCommit != null && autoCommit != conn.getAutoCommit()) {  
 conn.setAutoCommit(autoCommit);  
 }  
 // 设置事务隔离级别  
 if (defaultTransactionIsolationLevel != null) {  
 conn.setTransactionIsolation(defaultTransactionIsolationLevel);  
 }  
}

Statement stmt = handler.prepare(connection, transaction.getTimeout());

return handler.query(stmt, resultHandler);

RoutingStatementHandler#query

PreparedStatementHandler#query

@Override  
public <E> List<E> query(Statement statement, ResultHandler resultHandler) throws SQLException {  
 PreparedStatement ps = (PreparedStatement) statement;  
 System.*out*.println(ps.getConnection().getAutoCommit());  
 System.*out*.println(JSON.*toJSONString*(ps.getQueryTimeout()));  
 ps.execute();  
 return resultSetHandler.handleResultSets(ps);  
}

2.<dataSource type="DRUID">

区别在于JdbcTransaction的方法openConnection从dataSource具体实现中获取Connection方法的不同：

其中Connection的autoCommit属性，在DRUID中默认是true

可以通过配置文件进行配置

3.<dataSource type="com.lege.datasource.HikariCPDataSource.HikariCPDataSourceFactory"/>

**综上可以发现Connection的优先级是**：

1，获取SqlSession时手动设置的为最高

2，数据源支持的配置的其次

3，默认JDBC生成的Connection为true

MyBatis的架构设计以及实例分析

<https://blog.csdn.net/luanlouis/article/details/40422941>

MyBatis缓存机制的设计与实现

<https://blog.csdn.net/luanlouis/article/details/41390801>

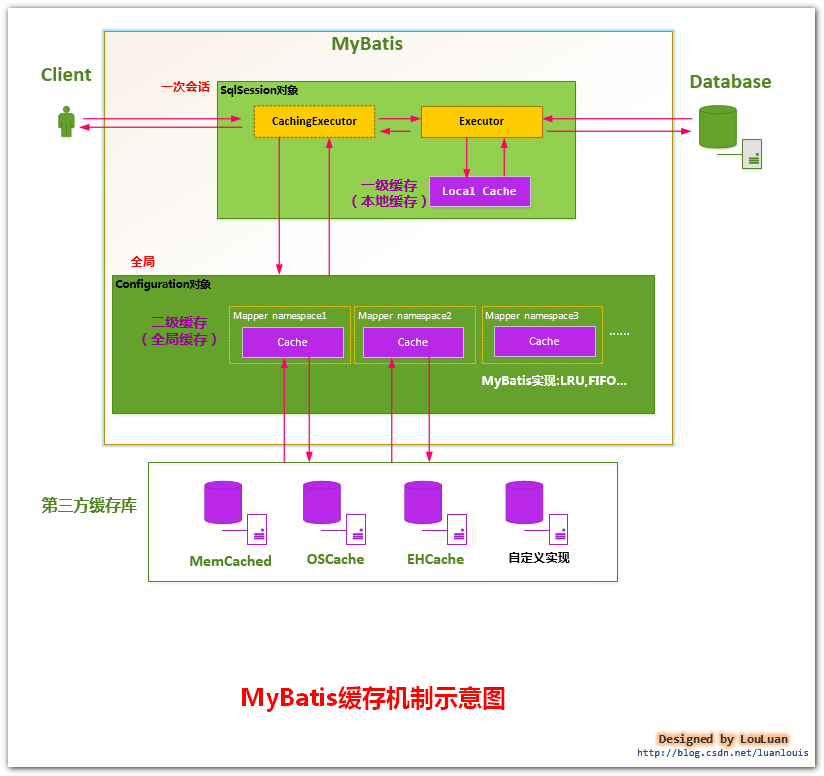
1.Mybatis缓存机制概述

MyBatis将数据缓存设计成两级结构，分为一级缓存、二级缓存：

**一级缓存是Session会话级别的缓存**，位于表示一次数据库会话的SqlSession对象之中，又被称之为本地缓存。一级缓存是MyBatis内部实现的一个特性，用户不能配置，默认情况下自动支持的缓存，用户没有定制它的权利（不过这也不是绝对的，可以通过开发插件对它进行修改）；

**二级缓存是Application应用级别的缓存**，它的是生命周期很长，跟Application的声明周期一样，也就是说它的作用范围是整个Application应用。

MyBatis中一级缓存和二级缓存的组织如下图所示：



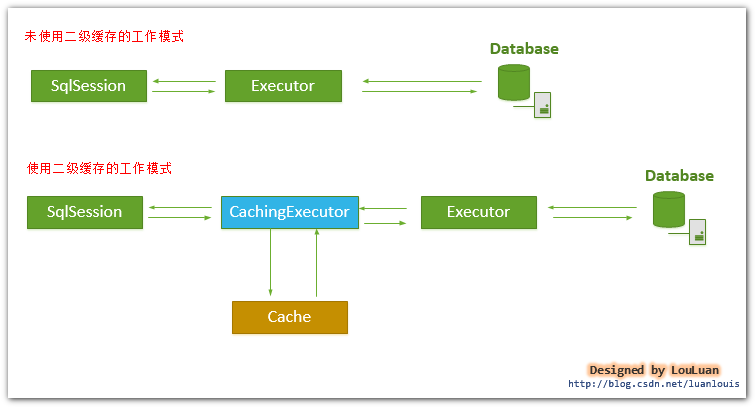
一级缓存的工作机制：

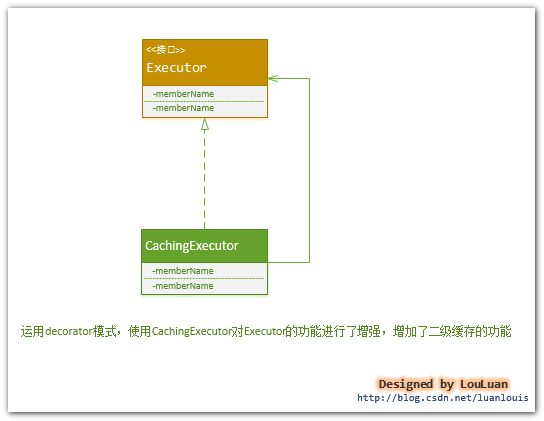
      一级缓存是Session会话级别的，一般而言，一个SqlSession对象会使用一个Executor对象来完成会话操作，Executor对象会维护一个Cache缓存，以提高查询性能。关于一级缓存的详细实现，我已经在《深入理解mybatis原理》 MyBatis的一级缓存实现详解 及使用注意事项 一文中有非常详尽的讨论，读者可以前去了解。

二级缓存的工作机制：

       如上所言，一个SqlSession对象会使用一个Executor对象来完成会话操作，MyBatis的二级缓存机制的关键就是对这个Executor对象做文章。如果用户配置了"cacheEnabled=true"，那么MyBatis在为SqlSession对象创建Executor对象时，会对Executor对象加上一个装饰者：CachingExecutor，这时SqlSession使用CachingExecutor对象来完成操作请求。CachingExecutor对于查询请求，会先判断该查询请求在Application级别的二级缓存中是否有缓存结果，如果有查询结果，则直接返回缓存结果；如果缓存中没有，再交给真正的Executor对象来完成查询操作，之后CachingExecutor会将真正Executor返回的查询结果放置到缓存中，然后在返回给用户。

     MyBatis的二级缓存设计得比较灵活，你可以使用MyBatis自己定义的二级缓存实现；你也可以通过实现org.apache.ibatis.cache.Cache接口自定义缓存；也可以使用第三方内存缓存库，如Memcached等，这个我们会在后续的文章中详细讨论。





2.MyBatis的一级缓存实现详解 及使用注意事项

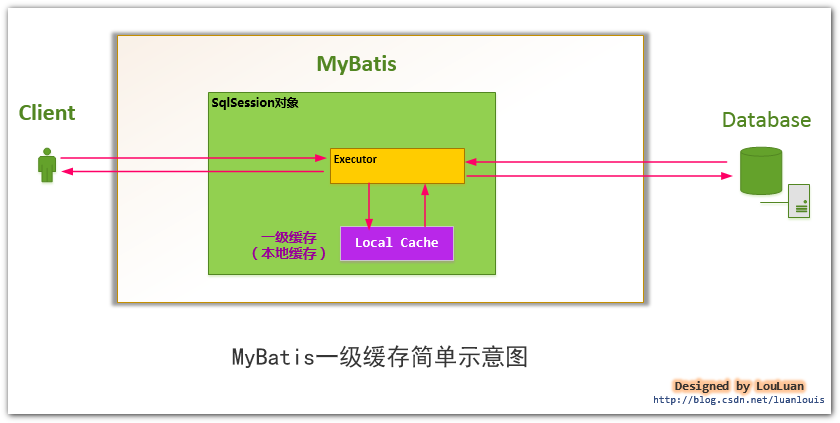
<https://blog.csdn.net/luanlouis/article/details/41280959>

每当我们使用MyBatis开启一次和数据库的会话，MyBatis会创建出一个SqlSession对象表示一次数据库会话。

      在对数据库的一次会话中，我们有可能会反复地执行完全相同的查询语句，如果不采取一些措施的话，每一次查询都会查询一次数据库,而我们在极短的时间内做了完全相同的查询，那么它们的结果极有可能完全相同，由于查询一次数据库的代价很大，这有可能造成很大的资源浪费。

      为了解决这一问题，减少资源的浪费，MyBatis会在表示会话的SqlSession对象中建立一个简单的缓存，将每次查询到的结果结果缓存起来，当下次查询的时候，如果判断先前有个完全一样的查询，会直接从缓存中直接将结果取出，返回给用户，不需要再进行一次数据库查询了。

     如下图所示，MyBatis会在一次会话的表示----一个SqlSession对象中创建一个本地缓存(local cache)，对于每一次查询，都会尝试根据查询的条件去本地缓存中查找是否在缓存中，如果在缓存中，就直接从缓存中取出，然后返回给用户；否则，从数据库读取数据，将查询结果存入缓存并返回给用户。



MyBatis中的一级缓存是怎样组织的？（即SqlSession中的缓存是怎样组织的？）

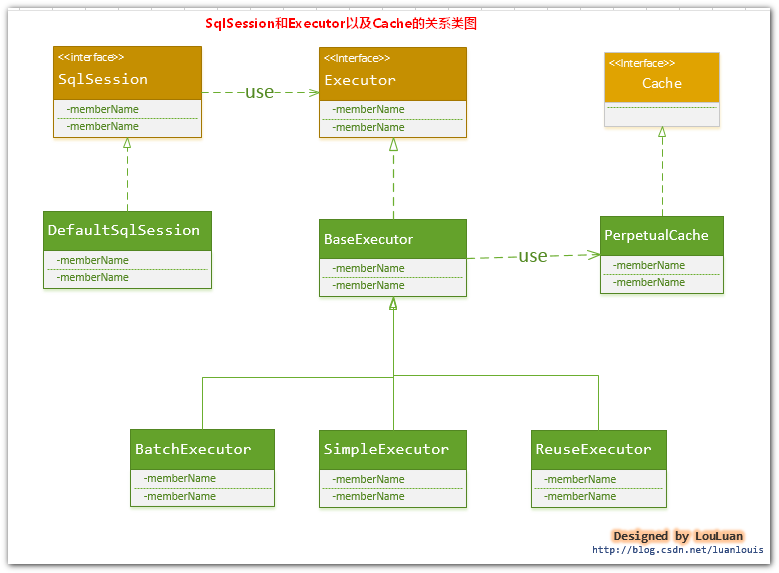
由于MyBatis使用SqlSession对象表示一次数据库的会话，那么，对于会话级别的一级缓存也应该是在SqlSession中控制的。

实际上, SqlSession只是一个MyBatis对外的接口，SqlSession将它的工作交给了Executor执行器这个角色来完成，负责完成对数据库的各种操作。当创建了一个SqlSession对象时，MyBatis会为这个SqlSession对象创建一个新的Executor执行器，而缓存信息就被维护在这个Executor执行器中，MyBatis将缓存和对缓存相关的操作封装成了Cache接口中。SqlSession、Executor、Cache之间的关系如下列类图所示：

每个SqlSession的创建都会创建一个其对应的Executor

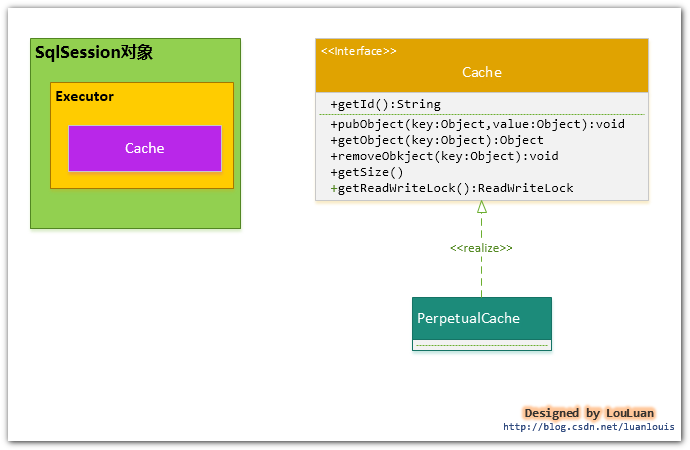
Executor executor = configuration.newExecutor(tx, execType);

new DefaultSqlSession(configuration, executor, autoCommit)



如上述的类图所示，Executor接口的实现类BaseExecutor中拥有一个Cache接口的实现类PerpetualCache，则对于BaseExecutor对象而言，它将使用PerpetualCache对象维护缓存。

综上，SqlSession对象、Executor对象、Cache对象之间的关系如下图所示：



由于Session级别的一级缓存实际上就是使用PerpetualCache维护的，那么PerpetualCache是怎样实现的呢？

PerpetualCache实现原理其实很简单，其内部就是通过一个简单的HashMap<k,v> 来实现的，没有其他的任何限制。

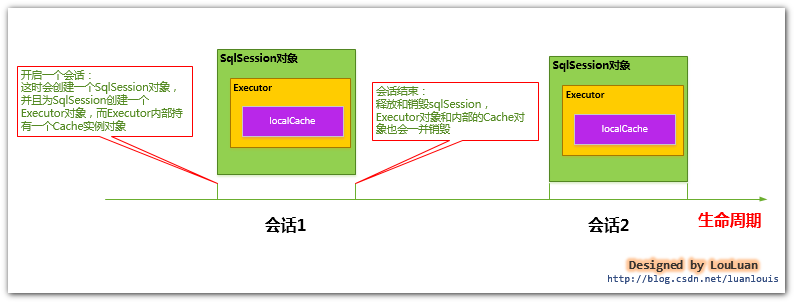
一级缓存的生命周期有多长？

a. MyBatis在开启一个数据库会话时，会 创建一个新的SqlSession对象，SqlSession对象中会有一个新的Executor对象，Executor对象中持有一个新的PerpetualCache对象；当会话结束时，SqlSession对象及其内部的Executor对象还有PerpetualCache对象也一并释放掉。

b. 如果SqlSession调用了close()方法，会释放掉一级缓存PerpetualCache对象，一级缓存将不可用；

c. 如果SqlSession调用了clearCache()，会清空PerpetualCache对象中的数据，但是该对象仍可使用；

d.SqlSession中执行了任何一个update操作(update()、delete()、insert()) ，都会清空PerpetualCache对象的数据，但是该对象可以继续使用；



4. SqlSession 一级缓存的工作流程：

1.对于某个查询，根据statementId,params,rowBounds来构建一个key值，根据这个key值去缓存Cache中取出对应的key值存储的缓存结果；

2. 判断从Cache中根据特定的key值取的数据数据是否为空，即是否命中；

3. 如果命中，则直接将缓存结果返回；

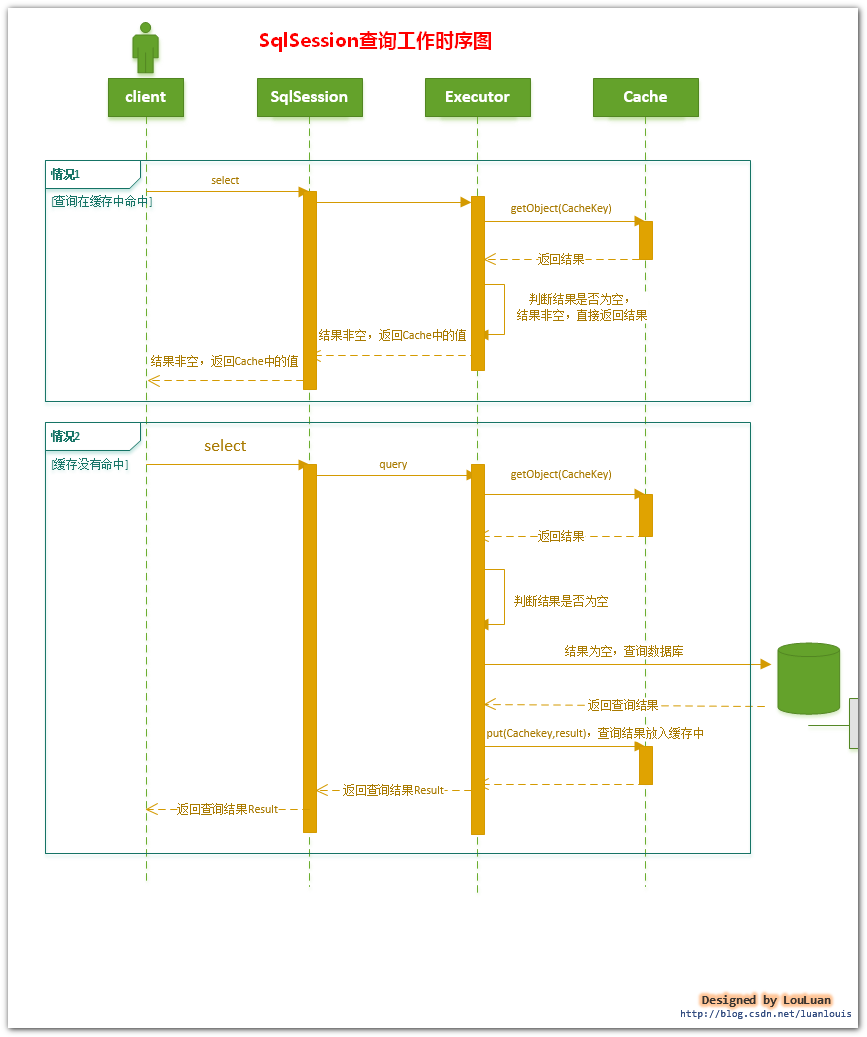
4. 如果没命中：

        4.1  去数据库中查询数据，得到查询结果；

        4.2  将key和查询到的结果分别作为key,value对存储到Cache中；

        4.3. 将查询结果返回；

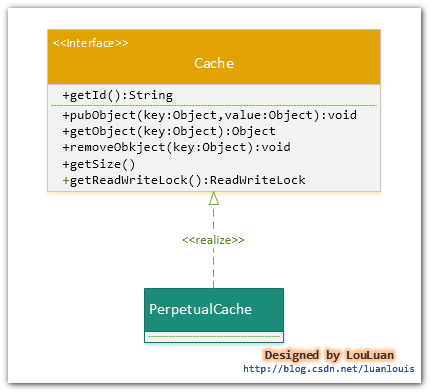
5. 结束。



Cache接口的设计以及CacheKey的定义（非常重要）

  如下图所示，MyBatis定义了一个org.apache.ibatis.cache.Cache接口作为其Cache提供者的SPI(Service Provider Interface) ，所有的MyBatis内部的Cache缓存，都应该实现这一接口。MyBatis定义了一个PerpetualCache实现类实现了Cache接口，实际上，在SqlSession对象里的Executor 对象内维护的Cache类型实例对象，就是PerpetualCache子类创建的。

    （MyBatis内部还有很多Cache接口的实现，一级缓存只会涉及到这一个PerpetualCache子类，Cache的其他实现将会放到二级缓存中介绍）。



我们知道，Cache最核心的实现其实就是一个Map，将本次查询使用的特征值作为key，将查询结果作为value存储到Map中。

现在最核心的问题出现了：怎样来确定一次查询的特征值？

换句话说就是：怎样判断某两次查询是完全相同的查询？

也可以这样说：如何确定Cache中的key值？

MyBatis认为，对于两次查询，如果以下条件都完全一样，那么就认为它们是完全相同的两次查询：

1. 传入的 statementId

2. 查询时要求的结果集中的结果范围 （结果的范围通过rowBounds.offset和rowBounds.limit表示）；

3. 这次查询所产生的最终要传递给JDBC java.sql.Preparedstatement的Sql语句字符串（boundSql.getSql() ）

4. 传递给java.sql.Statement要设置的参数值

现在分别解释上述四个条件：

1. 传入的statementId，对于MyBatis而言，你要使用它，必须需要一个statementId，它代表着你将执行什么样的Sql；

2. MyBatis自身提供的分页功能是通过RowBounds来实现的，它通过rowBounds.offset和rowBounds.limit来过滤查询出来的结果集，这种分页功能是基于查询结果的再过滤，而不是进行数据库的物理分页；

由于MyBatis底层还是依赖于JDBC实现的，那么，对于两次完全一模一样的查询，MyBatis要保证对于底层JDBC而言，也是完全一致的查询才行。而对于JDBC而言，两次查询，只要传入给JDBC的SQL语句完全一致，传入的参数也完全一致，就认为是两次查询是完全一致的。

即3、4两条MyBatis最本质的要求就是：

                  调用JDBC的时候，传入的SQL语句要完全相同，传递给JDBC的参数值也要完全相同。

综上所述,CacheKey由以下条件决定：

                                     statementId  + rowBounds  + 传递给JDBC的SQL  + 传递给JDBC的参数值

对于每次的查询请求，Executor都会根据传递的参数信息以及动态生成的SQL语句，将上面的条件根据一定的计算规则，创建一个对应的CacheKey对象。

我们知道创建CacheKey的目的，就两个：

    1. 根据CacheKey作为key,去Cache缓存中查找缓存结果；

    2. 如果查找缓存命中失败，则通过此CacheKey作为key，将从数据库查询到的结果作为value，组成key,value对存储到Cache缓存中。

CacheKey的构建被放置到了Executor接口的实现类BaseExecutor中，定义如下：

CacheKey的hashcode生成算法

一级缓存的性能分析 TODO

我将从两个 一级缓存的特性来讨论SqlSession的一级缓存性能问题：

**1.MyBatis对会话（Session）级别的一级缓存设计的比较简单，就简单地使用了HashMap来维护，并没有对HashMap的容量和大小进行限制**。

读者有可能就觉得不妥了：如果我一直使用某一个SqlSession对象查询数据，这样会不会导致HashMap太大，而导致 java.lang.OutOfMemoryError错误啊？ 读者这么考虑也不无道理，不过MyBatis的确是这样设计的。

MyBatis这样设计也有它自己的理由：

a.  一般而言SqlSession的生存时间很短。一般情况下使用一个SqlSession对象执行的操作不会太多，执行完就会消亡；

b.  对于某一个SqlSession对象而言，只要执行update操作（update、insert、delete），都会将这个SqlSession对象中对应的一级缓存清空掉，所以一般情况下不会出现缓存过大，影响JVM内存空间的问题；

c.  可以手动地释放掉SqlSession对象中的缓存。

**2.  一级缓存是一个粗粒度的缓存，没有更新缓存和缓存过期的概念**

      MyBatis的一级缓存就是使用了简单的HashMap，MyBatis只负责将查询数据库的结果存储到缓存中去， 不会去判断缓存存放的时间是否过长、是否过期，因此也就没有对缓存的结果进行更新这一说了。

根据一级缓存的特性，在使用的过程中，我认为应该注意：

1、对于数据变化频率很大，并且需要高时效准确性的数据要求，我们使用SqlSession查询的时候，要控制好SqlSession的生存时间，SqlSession的生存时间越长，它其中缓存的数据有可能就越旧，从而造成和真实数据库的误差；同时对于这种情况，用户也可以手动地适时清空SqlSession中的缓存；

2、对于只执行、并且频繁执行大范围的select操作的SqlSession对象，SqlSession对象的生存时间不应过长。

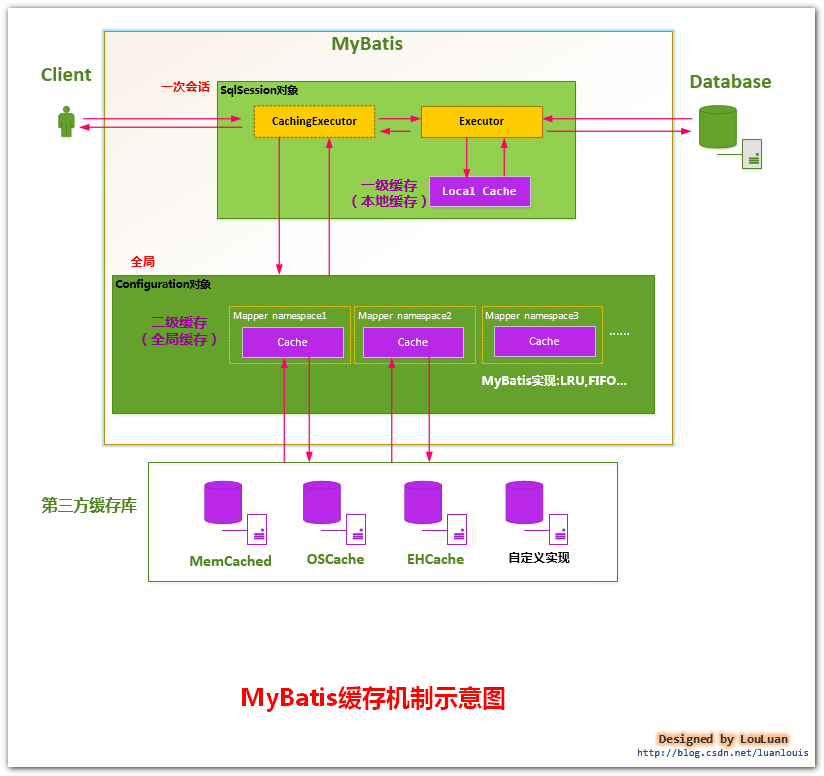
MyBatis认为的完全相同的查询，不是指使用sqlSession查询时传递给算起来Session的所有参数值完完全全相同，你只要保证statementId，rowBounds,最后生成的SQL语句，以及这个SQL语句所需要的参数完全一致就可以了。

3.MyBatis的二级缓存的设计原理

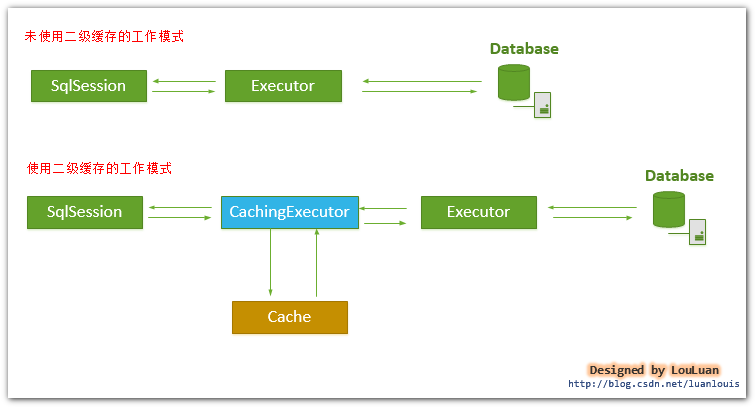
<https://blog.csdn.net/luanlouis/article/details/41408341>

MyBatis的二级缓存是Application级别的缓存，它可以提高对数据库查询的效率，以提高应用的性能。本文将全面分析MyBatis的二级缓存的设计原理。

1.MyBatis的缓存机制整体设计以及二级缓存的工作模式



如上图所示，当开一个会话时，一个SqlSession对象会使用一个Executor对象来完成会话操作，MyBatis的二级缓存机制的关键就是对这个Executor对象做文章。如果用户配置了"cacheEnabled=true"，那么MyBatis在为SqlSession对象创建Executor对象时，会对Executor对象加上一个装饰者：CachingExecutor，这时SqlSession使用CachingExecutor对象来完成操作请求。CachingExecutor对于查询请求，会先判断该查询请求在Application级别的二级缓存中是否有缓存结果，如果有查询结果，则直接返回缓存结果；如果缓存中没有，再交给真正的Executor对象来完成查询操作，之后CachingExecutor会将真正Executor返回的查询结果放置到缓存中，然后在返回给用户。



CachingExecutor是Executor的装饰者，以增强Executor的功能，使其具有缓存查询的功能，这里用到了设计模式中的装饰者模式，

CachingExecutor和Executor的接口的关系如下类图所示：

2 . MyBatis二级缓存的划分

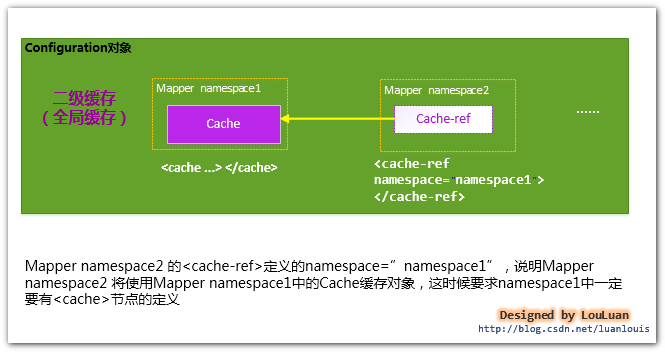
MyBatis并不是简单地对整个Application就只有一个Cache缓存对象，它将缓存划分的更细，即是Mapper级别的，即每一个Mapper都可以拥有一个Cache对象，具体如下：

a.为每一个Mapper分配一个Cache缓存对象（使用<cache>节点配置）；

b.多个Mapper共用一个Cache缓存对象（使用<cache-ref>节点配置）；

b.多个Mapper共用一个Cache缓存对象（使用<cache-ref>节点配置）

如果你想让多个Mapper公用一个Cache的话，你可以使用<cache-ref namespace="">节点，来指定你的这个Mapper使用到了哪一个Mapper的Cache缓存。



3. 使用二级缓存，必须要具备的条件

     MyBatis对二级缓存的支持粒度很细，它会指定某一条查询语句是否使用二级缓存。

     虽然在Mapper中配置了<cache>,并且为此Mapper分配了Cache对象，这并不表示我们使用Mapper中定义的查询语句查到的结果都会放置到Cache对象之中，我们必须指定Mapper中的某条选择语句是否支持缓存，即如下所示，在<select> 节点中配置useCache="true"，Mapper才会对此Select的查询支持缓存特性，否则，不会对此Select查询，不会经过Cache缓存。如下所示，Select语句配置了useCache="true"，则表明这条Select语句的查询会使用二级缓存。

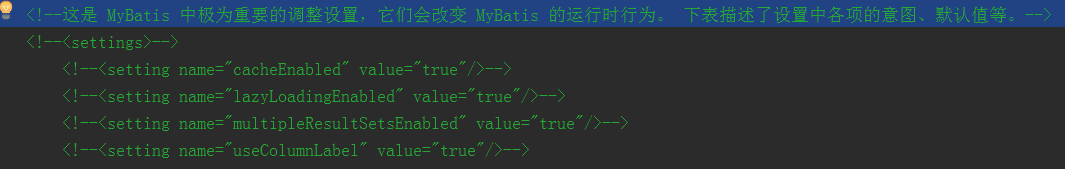
<select id="selectByMinSalary" resultMap="BaseResultMap" parameterType="java.util.Map" useCache="true">

总之，要想使某条Select查询支持二级缓存，你需要保证：

   1.  MyBatis支持二级缓存的总开关：全局配置变量参数   cacheEnabled=true

   2. 该select语句所在的Mapper，配置了<cache> 或<cached-ref>节点，并且有效

   3. 该select语句的参数 useCache=true



4. 一级缓存和二级缓存的使用顺序

      请注意，如果你的MyBatis使用了二级缓存，并且你的Mapper和select语句也配置使用了二级缓存，那么在执行select查询的时候，MyBatis会先从二级缓存中取输入，其次才是一级缓存，即MyBatis查询数据的顺序是：

               二级缓存    ———> 一级缓存——> 数据库

5. 二级缓存实现的选择

    MyBatis对二级缓存的设计非常灵活，它自己内部实现了一系列的Cache缓存实现类，并提供了各种缓存刷新策略如LRU，FIFO等等；另外，MyBatis还允许用户自定义Cache接口实现，用户是需要实现org.apache.ibatis.cache.Cache接口，然后将Cache实现类配置在<cache  type="">节点的type属性上即可；除此之外，MyBatis还支持跟第三方内存缓存库如Memecached的集成，总之，使用MyBatis的二级缓存有三个选择:

        1.MyBatis自身提供的缓存实现；

        2. 用户自定义的Cache接口实现；

        3.跟第三方内存缓存库的集成；

6.  MyBatis自身提供的二级缓存的实现

     MyBatis自身提供了丰富的，并且功能强大的二级缓存的实现，它拥有一系列的Cache接口装饰者，可以满足各种对缓存操作和更新的策略。

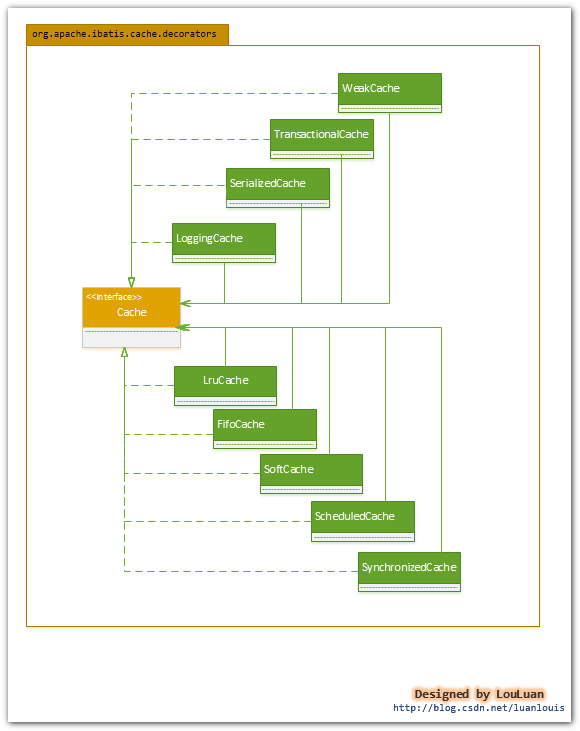
     MyBatis定义了大量的Cache的装饰器来增强Cache缓存的功能，如下类图所示。

     对于每个Cache而言，都有一个容量限制，MyBatis各供了各种策略来对Cache缓存的容量进行控制，以及对Cache中的数据进行刷新和置换。MyBatis主要提供了以下几个刷新和置换策略：

     LRU：（Least Recently Used）,最近最少使用算法，即如果缓存中容量已经满了，会将缓存中最近做少被使用的缓存记录清除掉，然后添加新的记录；

     FIFO：（First in first out）,先进先出算法，如果缓存中的容量已经满了，那么会将最先进入缓存中的数据清除掉；

     Scheduled：指定时间间隔清空算法，该算法会以指定的某一个时间间隔将Cache缓存中的数据清空；



6. 写在后面（关于涉及到的设计模式）

在二级缓存的设计上，MyBatis大量地运用了装饰者模式，如CachingExecutor, 以及各种Cache接口的装饰器。关于装饰者模式，读者可以阅读相关资料，我的另外一篇博文 Java 设计模式 装饰者模式 供读者参考。

7.一个关于MyBatis的二级缓存的实际问题 很重要

现有AMapper.xml中定义了对数据库表 ATable 的CRUD操作，BMapper定义了对数据库表BTable的CRUD操作；

假设 MyBatis 的二级缓存开启，并且 AMapper 中使用了二级缓存，AMapper对应的二级缓存为ACache；

除此之外，AMapper 中还定义了一个跟BTable有关的查询语句，类似如下所述：

<select id="selectATableWithJoin" resultMap="BaseResultMap" useCache="true">

select \* from ATable left join BTable on ....

</select>

执行以下操作：

1. 执行AMapper中的"selectATableWithJoin" 操作，此时会将查询到的结果放置到AMapper对应的二级缓存ACache中；

2. 执行BMapper中对BTable的更新操作(update、delete、insert)后，BTable的数据更新；

3. 再执行1完全相同的查询，这时候会直接从AMapper二级缓存ACache中取值，将ACache中的值直接返回；

好，问题就出现在第3步上：

     由于AMapper的“selectATableWithJoin” 对应的SQL语句需要和BTable进行join查找，而在第 2 步BTable的数据已经更新了，但是第 3 步查询的值是第 1 步的缓存值，已经极有可能跟真实数据库结果不一样，即ACache中缓存数据过期了！

总结来看，就是：

         对于某些使用了 join连接的查询，如果其关联的表数据发生了更新，join连接的查询由于先前缓存的原因，导致查询结果和真实数据不同步；

从MyBatis的角度来看，这个问题可以这样表述：

        对于某些表执行了更新(update、delete、insert)操作后，如何去清空跟这些表有关联的查询语句所造成的缓存；

当前的MyBatis的缓存机制不能很好地处理这一问题，下面我们将从当前的MyBatis的缓存机制入手，分析这一问题：

4.如何细粒度地控制你的MyBatis二级缓存(mybatis-enhanced-cache插件实现)

<https://blog.csdn.net/luanlouis/article/details/41800511>

数据库连接和Java线程的关系

<https://blog.csdn.net/luanlouis/article/details/90760372>

MyBatis事务管理机制

<https://blog.csdn.net/luanlouis/article/details/37992171>

# 数据库隔离级别

一个线上问题：

问题描述：海外环境，读取DB，1-查询，2-修改，3-查询：结果是1和3的查询结果是一致的

场景情况：orm框架的connection 没有 设置autoCommit，数据库默认隔离级别是repeatable\_read

解决：框架ORM设置成autoCommit 这个是主要原因，同时数据库层面的默认隔离级别设置成read\_committed

情况是：

在ORM设置autocommit为false，数据库隔离级别设置成repeatable\_read会有问题，设置为read\_committed没问题。

在ORM指定autocommit为true，数据库隔离级别设置成任意都可以。

mysql 有四级事务隔离级别 每个级别都有字符或数字编号



**读未提交是指**：一个事务还未提交的时候，它做的变更可以被其他的事务看到。

**读提交是指**：一个事务提交之后，它做的事务才可以被其他事务看到。

**可重复读是指：**一**个事务在执行过程中，它看到的总是和这个事务启动时的数据一致**。可重复读隔离级别使用MMVC机制解决不可重复读问题（一个事务中 一条行数据两次读取数据可能不一致 ），实现可重复读。

**串行化是指**：写会加写锁，读会加读锁，当出现读写锁冲突的时候，后面访问的事务必须等待前一个事务执行完成之后，才可以继续执行。

MySQL

**mysql默认的事务处理级别是'REPEATABLE-READ',也就是可重复读**

1.查看当前会话隔离级别

select @@tx\_isolation;

2.查看系统当前隔离级别

select @@global.tx\_isolation;

3.设置当前会话隔离级别

set session transaction isolatin level repeatable read;

4.设置系统当前隔离级别

set global transaction isolation level repeatable read;

结合这里重新 <https://blog.csdn.net/luanlouis/article/details/95319795>

先我们要搞明白何谓幻读，目前网上的众多解释幻读的博文个人感觉仔细设想一下就能找出推翻的例子，就像博文把 非阻塞IO 等同为 异步IO，然后好多文章都纷纷借用，其实这俩货是完全不同，非阻塞IO 是 同步IO 中的一种模式，并非 异步IO。错误的观点都被大众认同的 "正确化" 了，扯远了，回归主题。

幻读会在 RU / RC / RR 级别下出现，SERIALIZABLE 则杜绝了幻读，但 RU / RC 下还会存在脏读，不可重复读，故我们就以 RR 级别来研究幻读，排除其他干扰。

注意：RR 级别下存在幻读的可能，但也是可以使用对记录手动加 X锁 的方法消除幻读。SERIALIZABLE 正是对所有事务都加 X锁 才杜绝了幻读，但很多场景下我们的业务sql并不会存在幻读的风险。SERIALIZABLE 的一刀切虽然事务绝对安全，但性能会有很多不必要的损失。故可以在 RR 下根据业务需求决定是否加锁，存在幻读风险我们加锁，不存在就不加锁，事务安全与性能兼备，这也是 RR 作为mysql默认隔是个事务离级别的原因，所以需要正确的理解幻读。

幻读错误的理解：说幻读是 事务A 执行两次 select 操作得到不同的数据集，即 select 1 得到 10 条记录，select 2 得到 11 条记录。这其实并不是幻读，这是不可重复读的一种，只会在 R-U R-C 级别下出现，而在 mysql 默认的 RR 隔离级别是不会出现的。

这里给出我对幻读的比较白话的理解：

幻读，并不是说两次读取获取的结果集不同，幻读侧重的方面是某一次的 select 操作得到的结果所表征的数据状态无法支撑后续的业务操作。更为具体一些：select 某记录是否存在，不存在，准备插入此记录，但执行 insert 时发现此记录已存在，无法插入，此时就发生了幻读。

这里给出 mysql 幻读的比较形象的场景（借用我在知乎上的回答）：

一个幻读的例子：

1.在事务1中，查询User表id为1的是用户否存在，如果不存在则插入一条id为1的数据。

2.在事务1查询结束后，事务2往User表中插入了一条id为1的数据。

3.此时，由于事务1查询到id为1的用户不存在，因此插入1条id为1的数据。

4.但是由于事务2已经插入了1条id为1的数据，因此此时会报主键冲突，对于事务1 的业务来说是执行失败的，这里事务1 就是发生了幻读，因为事务1读取的数据状态并不能支持他的下一步的业务，见鬼了一样。这里要灵活的理解读取的意思，第一次select是读取，第二次的insert其实也属于隐式的读取，只不过是在mysql的机制中读取的，插入数据也是要先读取一下有没有主键冲突才能决定是否执行插入。

# MyBatis-Spring源码

**源码最最重要的就是下面四个类，讲讲这四个类就完全ok了**！

**SqlSessionFactoryBean**

**MapperScannerRegistrar**

**MapperScannerConfigurer**

**MapperFactoyBean**

SqlSessionFactoryBean的设计与实现

<https://www.cnblogs.com/java-chen-hao/p/11833780.html>

mybatis-spring为了实现spring对mybatis的整合，即将mybatis的相关组件作为spring的IOC容器的bean来管理，使用了spring的FactoryBean接口来对mybatis的相关组件进行包装。spring的IOC容器在启动加载时，如果发现某个bean实现了FactoryBean接口，则会调用该bean的getObject方法，获取实际的bean对象注册到IOC容器，其中FactoryBean接口提供了getObject方法的声明，从而统一spring的IOC容器的行为。

SqlSessionFactory作为mybatis的启动组件，在mybatis-spring中提供了SqlSessionFactoryBean来进行包装，所以在spring项目中整合mybatis，首先需要在spring的配置，如XML配置文件applicationContext.xml中，配置SqlSessionFactoryBean来引入SqlSessionFactory，即在spring项目启动时能加载并创建SqlSessionFactory对象，然后注册到spring的IOC容器中，从而可以直接在应用代码中注入使用或者作为属性，注入到mybatis的其他组件对应的bean对象。在applicationContext.xml的配置如下：

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

// 数据源

<property name="dataSource" ref="dataSource" />

// mapper.xml的资源文件,也就是SQL文件

<property name="mapperLocations" value="classpath:mybatis/mapper/\*\*/\*.xml" />

//mybatis配置mybatisConfig.xml的资源文件

<property name="configLocation" value="classpath:mybatis/mybitas-config.xml" />

</bean>

## @MapperScan将Mapper接口生成代理注入到Spring-静态代理和动态代理结合使用

<https://www.cnblogs.com/java-chen-hao/p/11839958.html>

#### 扫描Mapper接口

我们上一篇文章介绍了扫描Mapper接口有两种方式，一种是**通过bean.xml注册MapperScannerConfigurer对**象，一种是通过**@MapperScan("com.chenhao.mapper")**注解的方式，如下

方式一：

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<property name="basePackage" value="com.chenhao.mapper" />

</bean>

方式二：

@Configuration

@MapperScan("com.chenhao.mapper")

public class AppConfig {

@MapperScan

我们来看看@MapperScan这个注解

@Import(MapperScannerRegistrar.class)

public @interface MapperScan {

MapperScan使用@Import将MapperScannerRegistrar导入。

#### @MapperScan

我们来看看@MapperScan这个注解

@Import(**MapperScannerRegistrar**.class)

public @interface MapperScan {

MapperScan使用@Import将MapperScannerRegistrar导入。

#### MapperScannerRegistrar

首先MapperScannerRegistrar是实现了ImportBeanDefinitionRegistrar接口

而ImportBeanDefinitionRegistrar类具有如下属性：

[

ImportBeanDefinitionRegistrar类只能通过其他类@Import的方式来加载，通常是启动类或配置类。

使用@Import，如果括号中的类是ImportBeanDefinitionRegistrar的实现类，则会调用接口方法，将其中要注册的类注册成bean。

实现该接口的类拥有注册bean的能力。

]

该类的执行时机：

执行时机：ImportBeanDefinitionRegistrar接口不是直接注册Bean到IOC容器，它的执行时机比较早，准确的说更是注册Bean的定义信息以便后面的Bean的创建。

通过重写ImportBeanDefinitionRestrar中的默认方法registerBeanDefinitions

在MapperScannerRegistrar类中该方法的主要作用是：

这里面是注入的一个bean名称是beanName（如下），beanClass是MapperScannerConfigurer的 BeanDifinition

beanName: **com.lege.mybatispring.annotationconfig.MyBatisSpringAnnotationConfig#MapperScannerRegistrar#0**

beanClass: **MapperScannerConfigurer**.class 这是一个BeanDefinitionRegistryPostProcessor

#### MapperScannerConfigurer

该类继承BeanDefinitionRegistryPostProcessor

会在在spring refresh流程中的invokeBeanFactoryPostProcessors(beanFactory)方法中，该MapperScannerConfigurer中**postProcessBeanDefinitionRegistry**方法会被执行

该方法中会使用ClassPathMapperScanner去扫描MapperDao目录下的Mapper类型的BeanDifinition

BeanDifinition中beanCalss会被赋值为MapperFactoryBean

ClassPathMapperScanner下的doScan方法中的processBeanDefinitions(beanDefinitions)来进行definition.setBeanClass(this.mapperFactoryBeanClass)赋值操作

#### MapperFactoyBean

而后面getBean的获取Mapper接口实例的时候，会调用MapperFactoyBean中的getObject方法，而该方法内的实现交给了MyBatis去具体获取Mapper接口实例的代理实现MapperProxy

MapperProxy

## mybatis-spring-boot-starter原理

Springboot中只需要引入