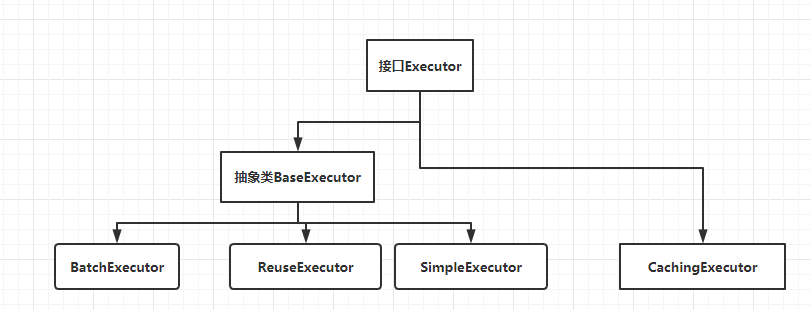
# Mybatis的4大接口

## Executor

org.apache.ibatis.executor.Executor：Executor执行器，SqlSession执行的数据库脚本都是通过Executor这个接口完成的，SqlSession会持有一个Executor的实例。Executor会调度StatementHandler来完成statement对象的创建，然后通过ParameterHandler来为 PreparedStatement 的 sql 语句参数动态赋值，通过resultsetHandler来封装结果

### 体系结构



### 定义的方法

更新

int update(MappedStatement ms, Object parameter) throws SQLException;

// 查询，先查缓存，再查数据库

<E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey cacheKey, BoundSql boundSql) throws SQLException;

// 查询

<E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler) throws SQLException;

<E> Cursor<E> queryCursor(MappedStatement ms, Object parameter, RowBounds rowBounds) throws SQLException;

List<BatchResult> flushStatements() throws SQLException;

// 事务提交

void commit(boolean required) throws SQLException;

// 事务回滚

void rollback(boolean required) throws SQLException;

// 创建缓存的键对象

CacheKey createCacheKey(MappedStatement ms, Object parameterObject, RowBounds rowBounds, BoundSql boundSql);

// 缓存中是否有这个查询的结果

boolean isCached(MappedStatement ms, CacheKey key);

// 清空缓存

void clearLocalCache();

void deferLoad(MappedStatement ms, MetaObject resultObject, String property, CacheKey key, Class<?> targetType);

Transaction getTransaction();

void close(boolean forceRollback);

boolean isClosed();

void setExecutorWrapper(Executor executor);

org.apache.ibatis.executor.BaseExecutor

### BaseExecutor

BaseExecutor是一个抽象类，采用模板方法的设计模式。实现executor接口的大部分方法，主要提供了缓冲管理和事务管理的能力，其他子类需要

实现的抽象方法：doUpdate,doQuery等方法

具体使用哪一个Executor则是可以在 mybatis 的 config.xml 中进行配置的。默认为SimpleExecutor；

### SimpleExecutor

简单执行器，是MyBatis中默认使用的执行器，每执行一次update或select，就开启一个Statement对象，用完就直接关闭Statement对象(可以是Statement或者是PreparedStatment对象)

### ReuseExecutor

可重用执行器，这里的重用指的是重复使用Statement，它会在内部使用一个Map把创建的Statement都缓存起来，每次执行SQL命令的时候，都会去判断是否存在基于该SQL的Statement对象，如果存在Statement对象并且对应的connection还没有关闭的情况下就继续使用之前的Statement对象，并将其缓存起来。因为每一个SqlSession都有一个新的Executor对象，所以我们缓存在ReuseExecutor上的Statement作用域是同一个SqlSession。

### BatchExecutor

批处理执行器，用于将多个SQL一次性输出到数据库

### CachingExecutor

启用于二级缓存时的执行器；

缓存执行器，先从缓存中查询结果，如果存在，就返回；如果不存在，再委托给Executor delegate 去数据库中取，delegate可以是上面任何一个执行器

采用静态代理；代理一个 Executor 对象。（也即是将目标对象是SimpleExecutor、ReuseExecutor、BatchExecutor，执行数据库的操作委派给这些目标对象）

### Executor的创建

// 如果允许缓存，则使用缓存执行器

// 默认是true，如果不允许缓存的话，需要手动设置

创建过程

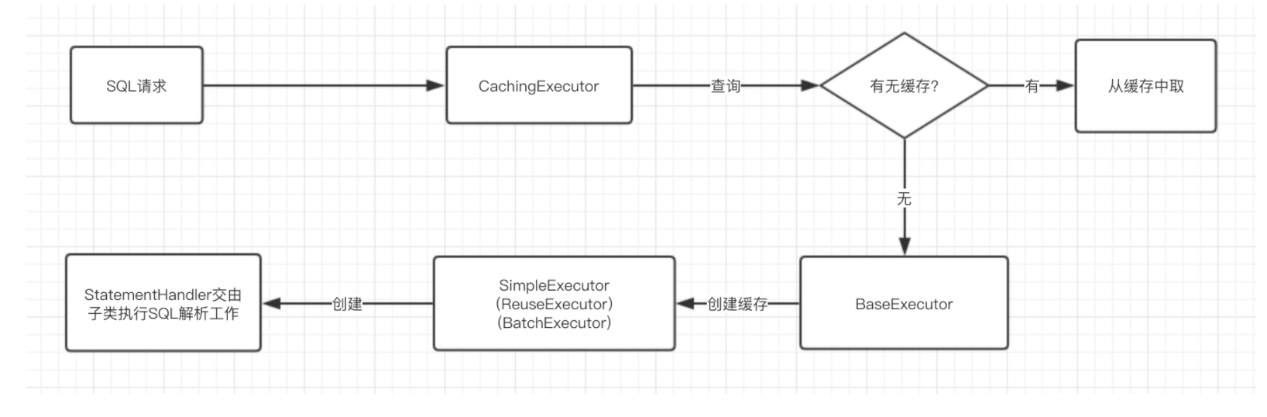
创建好SQLSessionFactory实例后调用openSession方法来创建（sqlSessionFactory.openSession()）;在openSession中执行org.apache.ibatis.session.defaults.DefaultSqlSessionFactory.openSessionFromDataSource(ExecutorType, TransactionIsolationLevel, boolean)方法，创建过程在这个方法中实现；

调用Configuration的newExecutor方法来创建Executor对象（ 根据配置的执行器类型创建对应的执行器实现对象（默认是SimpleExecutor）），

如果使用缓存（一级缓存），那么就创建CachingExecutor，采用装饰器模式将执行器放到CachingExecutor中，CachingExecutor也是Executor的实现类；当没有使用缓存的时候创建的执行器就是simpleExecutor，reuseExecutor,BatchExecutor中的一个，使用了那么就创建CachingExecutor（缓存执行器），用来处理缓存相关的操作，其他的执行还是交给开始创建的执行器，所以这里使用了装饰器模式

然后将执行器实例对象交给插件，执行interceptor的plugin方法，执行Plugin.wrap方法来创建CachingExecutor的动态代理对象（可以在plugin中判断是否调Plugin.wrap方法，来控制是否创建动态代理对象）,这样在执行Executor的代理对象时就实现了功能的增强

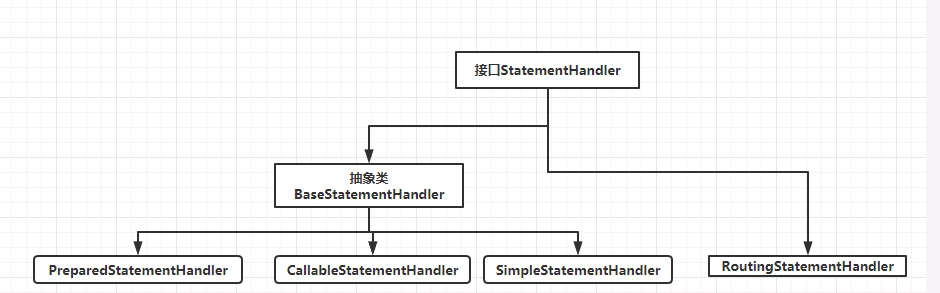
返回的Executor实例可能是代理对象，也可能不是代理对象



## StatementHandler

完成mybatis最核心的工作，也就是Executor实现的基础，功能保存创建statement对象，为sql语句绑定参数，执行增删改查等sql语句，在工作时还会使用 ParameterHandler 和 ResultSetHandler 对参数进行映射，对结果进行实体类的绑定

### 体现结构



### BaseStatementHandler

所有子类的抽象父类，定义初始化statement的操作顺序，由子类实现具体的实例化不同的statement（模板模式）

### PreparedStatementHandler

使用预编译PrepareStatement访问数据库

### SimpleStatementHandler

使用statement对象访问数据

### CallableStatementHandler

调用存储过程

### RoutingStatementHandler

RoutingStatementHandler 并没有对 Statement 对象进行使用，只是根据StatementType 来创建一个代理，代理的就是对应Handler的三种实现类。也就是使用静态代理模式，根据配置创建指定的handler实现类

### Statementhandler的创建

Statementhandler实例的创建都是在Excutor实现类的doQuery,doUpdate中完成的；

以SimpleExecutor为例

org.apache.ibatis.executor.SimpleExecutor.doQuery(MappedStatement, Object, RowBounds, ResultHandler, BoundSql)

在doQuery方法中调用Configuration的newStatementHandler创建

RoutingStatementHandler，在RoutingStatementHandler的构造函数中根据配置的StatementType来决定创建那种工作的Statementhandler，默认是创建PrepareStatementhandler

当创建完成RoutingStatementHandler实例后，调用拦截器的plugin方法，在方法中决定是否创建代理对象（和Executor类似）

## ParameterHandler

ParameterHandler 相比于其他的组件就简单很多了，ParameterHandler 译为参数处理器，

负责为 PreparedStatement 的 sql 语句参数动态赋值，这个接口很简单只有两个方法

### DefaultParameterHandler

ParameterHandler 只有一个实现类 DefaultParameterHandler ， 它实现了这两个方法。

* getParameterObject： 用于读取参数
* setParameters: 用于对 PreparedStatement 的参数赋值

Executor 管理的是 StatementHandler 对象的创建以及参数赋值，ParameterHandler的创建是由statementHandler的实现类BaseStatementHandler来完成的，

## ResultSetHandler

首先会经过 Executor 执行器，它主要负责管理创建 StatementHandler 对象，然后由 StatementHandler 对象做数据库的连接以及生成 Statement 对象，并解析 SQL 参数，由 ParameterHandler 对象负责把 Mapper 方法中的参数映射到 XML 中的 SQL 语句中，那么是不是还少了一个步骤，就能完成一个完整的 SQL 请求了？没错，这最后一步就是 SQL 结果集的处理工作，也就是 ResultSetHandler 的主要工作

解析查询结果，将结果封装成指定的结果然后返回

### DefaultResultSetHandler

MyBatis 只有一个默认的实现类就是 DefaultResultSetHandler，ResultSetHandler 主要负责处理两件事

1. 处理 Statement 执行后产生的结果集，生成结果列表
2. 处理存储过程执行后的输出参数

按照 Mapper 文件中配置的 ResultType 或 ResultMap 来封装成对应的对象，最后将封装的对象返回即可。

### 创建ParameterHandler和ResultSetHandler

在创建statementHandler实例的时候，会执行父类BaseStatementHandler的构造函数，创建工作在父类中完成

实例化ParameterHandler：configuration.newParameterHandler

实例化ResultSetHandler：configuration.newResultSetHandler

实现过程都一样，创建好对象之后，然后判断是否创建代理对象

# Mybatis的插件

## 插件的实现：

1. 定义插件类，实现org.apache.ibatis.plugin.Interceptor接口
2. 然后在类上使用@Intercepts，value就是@Signature的数组，在@Signature中配置type:拦截的接口类型，配置拦截的方法method，配置拦截方法的参数类型的class对象args
3. 接口的方法：setProperties：获取配置的属性

org.apache.ibatis.plugin.Interceptor.plugin(Object)这个方法中默认实现，用来创建代理对象

org.apache.ibatis.plugin.Interceptor.intercept(Invocation)，具体的增强方法，在方法实现时

Invocation的proceed方法就是执行目标方法

插件也就是拦截器，拦截的对象（拦截的方法）

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

ParameterHandler (getParameterObject, setParameters)

ResultSetHandler (handleResultSets, handleOutputParameters)

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

## Interceptor

接口org.apache.ibatis.plugin.Interceptor定义了3个方法

1. setProperties，设置配置的属性
2. plugin：用来创建代理对象，默认就是执行Plugin.wrap方法来创建代理对象
3. intercept：功能增强的具体操作，代理对象执行对应方法时，最后就是执行这个方法，方法参数是Invocation，Invocation执行proceed方法就是在之前目标方法

插件其实就是通过拦截mybatis的4大接口从而在执行的过程中实现一些定制化的操作，这也就是使用jdk的动态代理实现的，创建代理对象的方法就是Plugin.wrap

Plugin.wrap方法执行过程

getSignatureMap获取配置信息

获取拦截器的Intercepts注解（没有这个注解直接返回异常）

获取Intercepts注解的value，也就是获取Signature，获取配置的拦截方法和拦截的参数类型

　将信息封装在Map中，key是拦截接口的class对象，value是拦截的方法

　创建动态代理对象，org.apache.ibatis.plugin.Plugin实现了InvocationHandler接口，方法增强就是在这个plugin中实现的（增强也就是执行接口的intercept方法，改方法的参数就是封装在Invocation中－－包含了目标对象，执行的方法，个执行方法的参数）

这样在执行代理对象时就实现了功能的增强

# Mybatis的参数类型

MyBatis 在设置预处理语句（PreparedStatement）中的参数或从结果集中取出一个值时， 都会用类型处理器将获取到的值以合适的方式转换成 Java 类型（mybatis自带的基本够用）

# Mybatis整体架构

接口层 sqlsession

核心处理层 配置解析 参数映射 sql解析

Sql执行 结果集映射 插件

基础支持层 数据源 事务管理 缓存 binding模块

反射 类型转换 日志 资源加载 解析器

# 日志模块

mybatis没有提供日子的实现类，需要接入第三方的日志组件，但第三方日志组件都有各自的log级别，且各不相同，mybatis统一提供了trace，debug，warn，error四个级别

Mybatis会自动扫描日志实现，并且第三方日志插件加载优先级如下：slf4j->commonsLoging->Log4j2->Log4j->jdkLog

## 加载日志组件

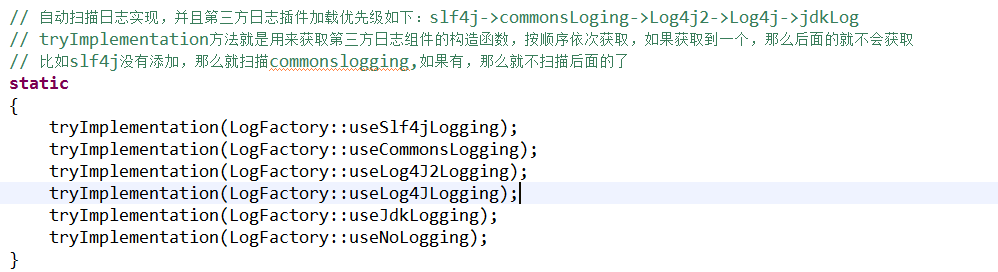
在org.apache.ibatis.logging.LogFactory中完成日志组件的加载

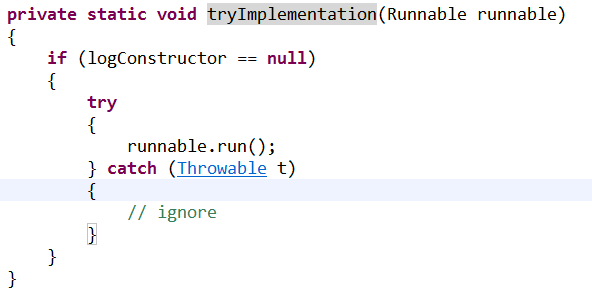
tryImplementation方法就是用来获取第三方日志组件的构造函数，按顺序依次获取，如果获取到一个，那么后面的就不会获取

比如slf4j没有添加，那么就扫描commonslogging,如果有，那么就不扫描后面的了

自动扫描日志实现，并且第三方日志插件加载优先级如下：

slf4j->commonsLoging->Log4j2->Log4j->jdkLog





## 适配各个日志组件

定义org.apache.ibatis.logging.Log接口，实现类用来适配各种日志组件

public interface Log {

boolean isDebugEnabled();

boolean isTraceEnabled();

void error(String s, Throwable e);

void error(String s);

void debug(String s);

void trace(String s);

void warn(String s);

}

org.apache.ibatis.logging.commons.JakartaCommonsLoggingImpl

org.apache.ibatis.logging.jdk14.Jdk14LoggingImpl

org.apache.ibatis.logging.log4j.Log4jImpl

org.apache.ibatis.logging.slf4j.Slf4jImpl

这些类实现log接口，再类中获取到对应组件的对象，再在对应的方法中适配日志级别

## 日志嵌入到主体功能（jdk动态代理）

BaseJdbcLogger：所有日志增强的抽象基类，所有日志增强的父类

子类：

ConnectionLogger 负责打印连接信息，和sql语句，并创建PreparedStatementLogger

PreparedStatementLogger 负责打印参数信息，并创建ResultSetLogger

ResultSetLogger 负责打印结果信息

StatementLogger

上面的几个类，父类都是 BaseJdbcLogger ，并且都实现了InvocationHandler

以连接的日志对象为例

org.apache.ibatis.logging.jdbc.ConnectionLogger

实现了java.lang.reflect.InvocationHandler接口

创建连接对象后，判断是否支持日志，如果不支持就直接返回连接对象，如果支持就用ConnectionLogger创建连接对象的代理对象

创建Connection的代理对象，在invoke中判断执行的方法是否需要输出日志

判断执行的方法，根据结果来创建对应的代理日志对象

org.apache.ibatis.executor.BaseExecutor.getConnection(Log)在这个方法中创建连接对象

protected Connection getConnection(Log statementLog) throws SQLException {

Connection connection = transaction.getConnection();

if (statementLog.isDebugEnabled()) {

return ConnectionLogger.newInstance(connection, statementLog, queryStack);

} else {

return connection;

}

}

返回的是代理对象，那么就会在执行对应方法时输出日志

# 数据源模块

1. 常见的数据源组件都实现了javax.sql.DataSource接口
2. mybatis不但要集成第三方的数据源组件，自身也提供了数据源的实现
3. 数据源初始化过程参数较多，比较复杂

这3种情况，所以mybatis使用了工厂模式，来实现数据源的创建

用工厂模式创建数据源的原因

数据源模块创建比较复杂

对象创建与使用耦合在一起 —— 违反单一原则

当业务扩展时，需要修改代码 —— 违反开闭原则

DataSourceFactory

UnpooledDataSourceFactory --不用连接池的数据连接工厂，每次都会创建一个新的连接

PooledDataSourceFactory --使用连接池的数据连接工厂

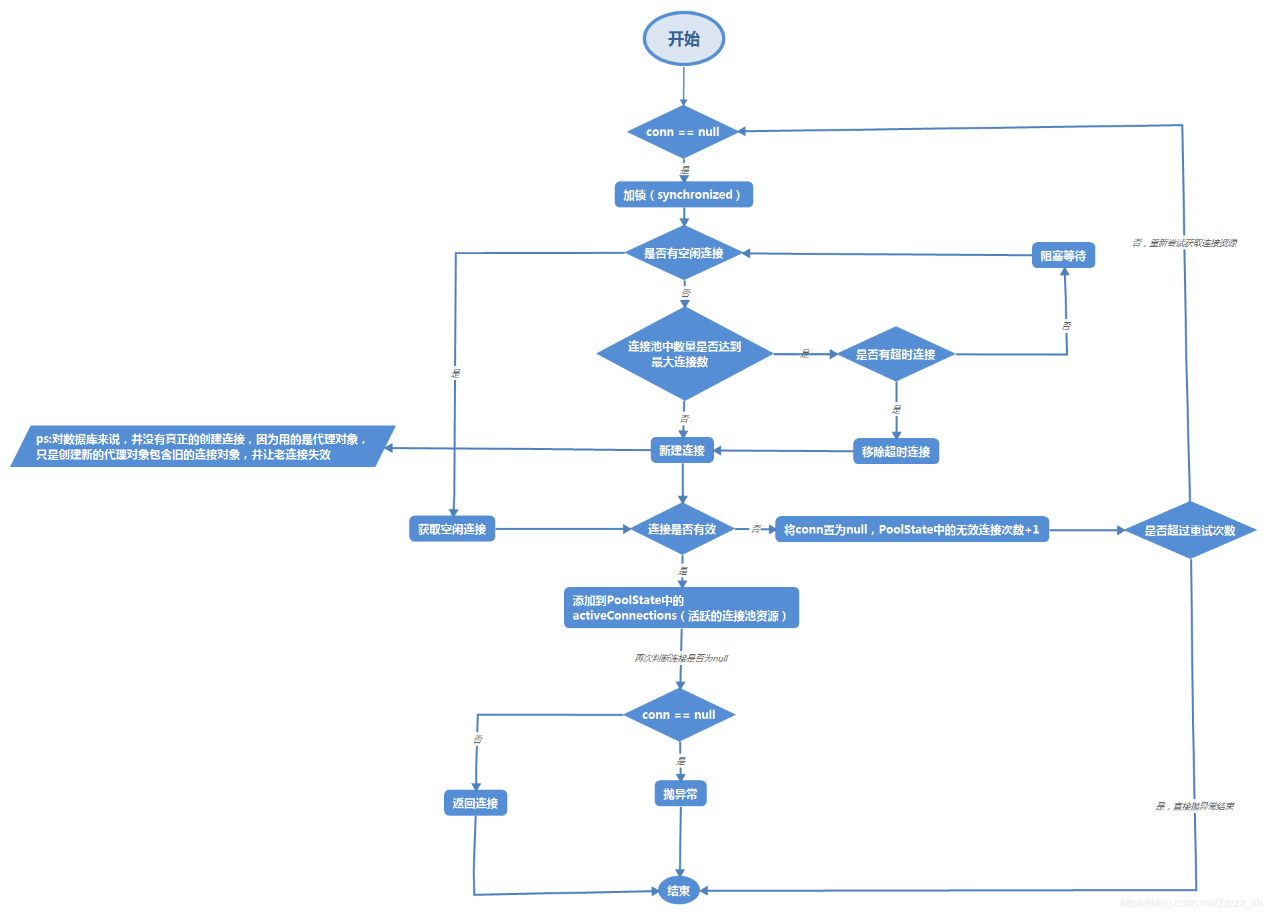
PooledConnection:使用动态代理封装真正的数据库连接对象

PoolState:用来管理PooledConnection对象状态的组件，通过两个list分别管理空闲状态的连接资源封装连接池的状态信息

和活跃状态的连接资源

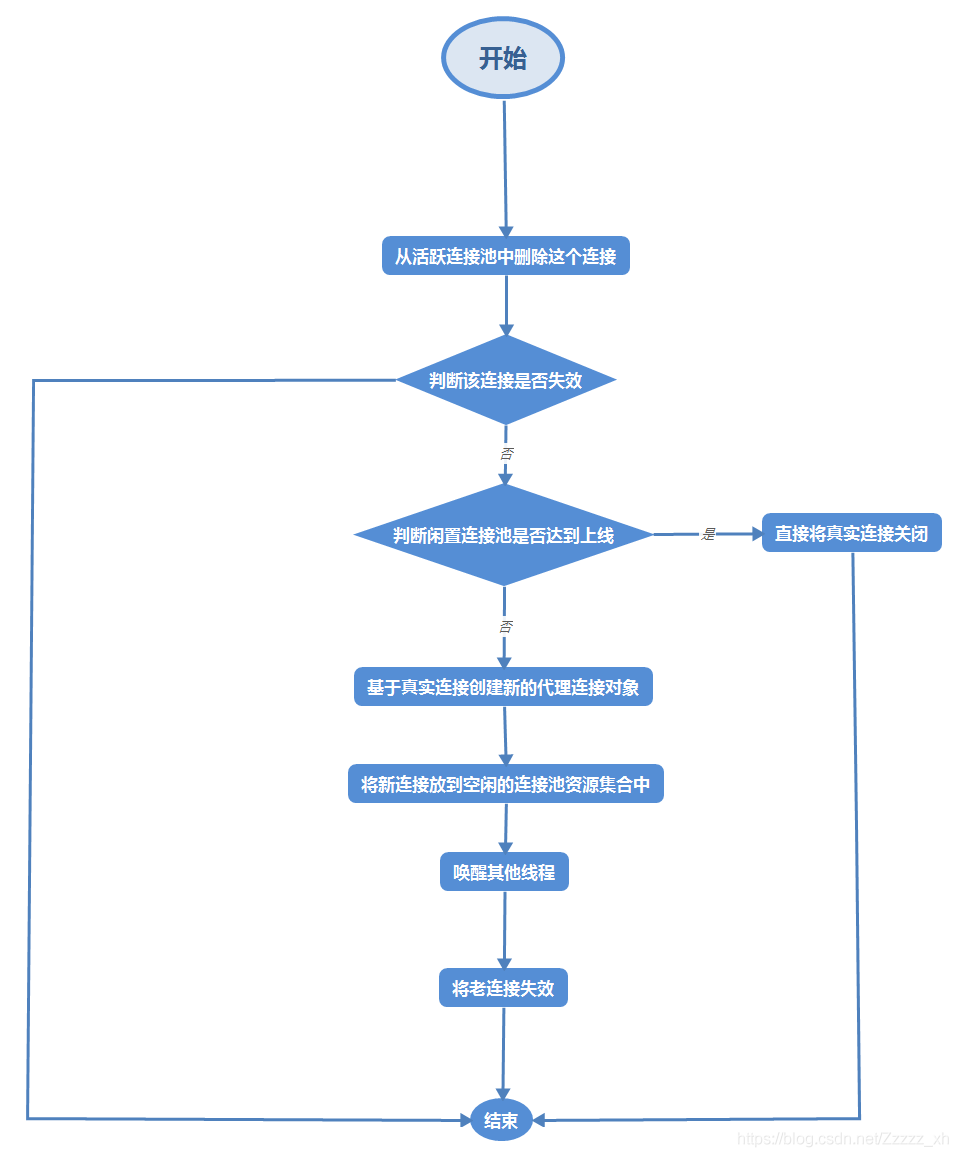
PooledDataSource：一个简单，同步的，线程安全的数据库连接池

获取数据库连接  
PooledDataSource.popConnection(String username, String password)方法



释放数据库连接

PooledDataSource.pushConnection(PooledConnection conn)



（数据连接池）

# 缓存模块

装饰器模式

为什么要用装饰器模式？  
Mybatis缓存的核心模块就是在缓存中读写数据，但是除了在缓存中读写数据wait，还有其他的附加功能，这些附加功能可以任意的加在缓存这个核心功能上。

加载附加功能可以有很多种方法，动态代理或者继承都可以实现，但是附加功能存在多种组合，用这两种方法，会导致生成大量的子类，所以mybatis选择使用装饰器模式.(灵活性、扩展性)

装饰器模式是一种用于代替继承的技术，无需通过继承增加子类就能扩展对象的新功能。使用对象的关联关系代替继承关系，更加灵活，同时避免类型体系的快速膨胀。

Mybatis缓存的实现是基于Map的，从缓存里面读写数据是缓存模块的核心基础功能；

除核心功能之外，有很多额外的附加功能，如：防止缓存击穿，添加缓存情况策略，序列化功能，日志能力，定时清空能力

附加功能可以以任意的组合附加到核心基础功能之上

org.apache.ibatis.cache.Cache:缓存模块的核心接口，定义缓存的基本操作

public interface Cache {

String getId();//缓存实现类的id

void putObject(Object key, Object value);//往缓存中添加数据，key一般是CacheKey对象

Object getObject(Object key);//根据指定的key从缓存获取数据

Object removeObject(Object key);//根据指定的key从缓存删除数据

void clear();//清空缓存

int getSize();//获取缓存的个数

ReadWriteLock getReadWriteLock();//获取读写锁

}

PerpetualCache：在缓存模块中扮演ConcreteComponent 角色，使用 HashMap来实现 BlockingCache：阻塞版本的缓存装饰器，保证只有一个线程到数据库去查找指定的 key 对应的数据

CacheKey

为什么不能通过String类型的key来当做缓存的key值？

因为Mybatis存在动态sql的缘故，所以不能单单用String来作为缓存的key （如果用String作为key，因为条件不同，sql不同，此时查询会有问题）

CacheKey构成：

1.mappedStatement的id

2.指定查询结果集的范围（分页信息）

3.查询所使用的的sql语句

4.用户传递给sql语句的实际参数值

Mybatis 缓存的使用

分为一级缓存跟二级缓存

– 一级缓存默认开启

– 二级缓存开启要在mapper.xml 文件中 配置

ps:二级缓存建议不要开启 因为生命周期过长，而且是根据namespace走的，所以如果有两个namespace，查询相同的sql，缓存相同的数据。其中一个namespace对该对象进行增、删、改， 就会让改缓存失效，但是另一个缓存依旧存在。

结果会导致两个查询结果出现不一致的情况

一级缓存

默认开启，生命周期是sqlsession，方法级别

缓存失效的原因

1. 执行增删改操作会清空缓存
2. 并不sqlsession—一级缓存失效

二级缓存

默认不开启，生命周期是SqlSessionFactory级别，类级别，但是每个二级缓存都是跟着namespace的

同一个namespaces不同的sqlsession, 虽然不在同一个sqlSession中，但是因为开启了二级缓存。所以第二次查询，走的是缓存，没有在数据库中查询

# mybatis中的反射

## ObjectFactory

ObjectFactory：mybatis每次创建结果对象的新实例时，它都会使用对象工厂去构件pojo

实现类DefaultObjectFactory；

org.apache.ibatis.reflection.factory.DefaultObjectFactory.instantiateClass(Class<T>, List<Class<?>>, List<Object>)

判断是否配置了构造函数的标签，没有配置就使用默认的无参构造函数，配置了就按配置的来初始化

## ReflectorFactory

ReflectorFactory：创建reflector（reflector封装的一个类的元信息，包括方法，属性等）的工厂类，reflector是mybatis反射模块的基础，每个reflector对象都对应一个类，在其中缓存了反射操作所需要的类元信息，实现类DefaultReflectorFactory；将Reflector保存在ConcurrentHashMap中，key为class类型，value就是Reflector

Reflector

private final Class<?> type;//对应的class

private final String[] readablePropertyNames;//可读属性的名称集合，存在get方法即可读

private final String[] writeablePropertyNames;//可写属性的名称集合，存在set方法即可写

private final Map<String, Invoker> setMethods = new HashMap<>();//保存属性相关的set方法

private final Map<String, Invoker> getMethods = new HashMap<>();//保存属性相关的get方法

private final Map<String, Class<?>> setTypes = new HashMap<>();//保存属性相关的set方法入参类型

private final Map<String, Class<?>> getTypes = new HashMap<>();//保存属性相关的get方法返回类型

private Constructor<?> defaultConstructor;//class默认的构造函数

## ObjectWrapper

ObjectWrapper：对对象的包装类，抽象了对象的属性信息，它定义了一些列查询对象属性信息的方法，以及更新属性的方法（也就是给对象赋值）

ObjectWrapperFactory：ObjectWrapper的工厂类，用于创建ObjectWrapper

MetaObject:封装了对象元信息，包装了mybatis中五个核心的反射类，也提供给外部使用的反射工具类，可以利用它读取或者修改对象的属性信息

public class MetaObject {

//原始的java对象

private final Object originalObject;

//对对象的包装，抽象了对象的属性信息，他定义了一系列查询对象属性信息的方法，以及更新属性的方法；

private final ObjectWrapper objectWrapper;

private final ObjectFactory objectFactory;

private final ObjectWrapperFactory objectWrapperFactory;

private final ReflectorFactory reflectorFactory;

// 此处有省略......

}

ObjectFactory—》创建对象

ReflectorFactory—》获取类中的信息（方法，属性等）

ObjectWrapper—》给对象赋值，

# Mybatis核心流程

## mybatis初始化阶段

读取xml配置文件和注解中的配置信息，创建配置对象，并完成各个模块的初始化工作

### Configuration

**Configuration**：mybatis启动初始化的核心就是将所有xml配置文件信息加载到Configuration对象中，Configuration是单例的，生命周期是应用级的

MapperRegistry：mapper接口动态代理工厂类的注册中心，在mybatisproxy实现invocationHandler接口，MapperProxyFactory用于生成动态代理的实例对象

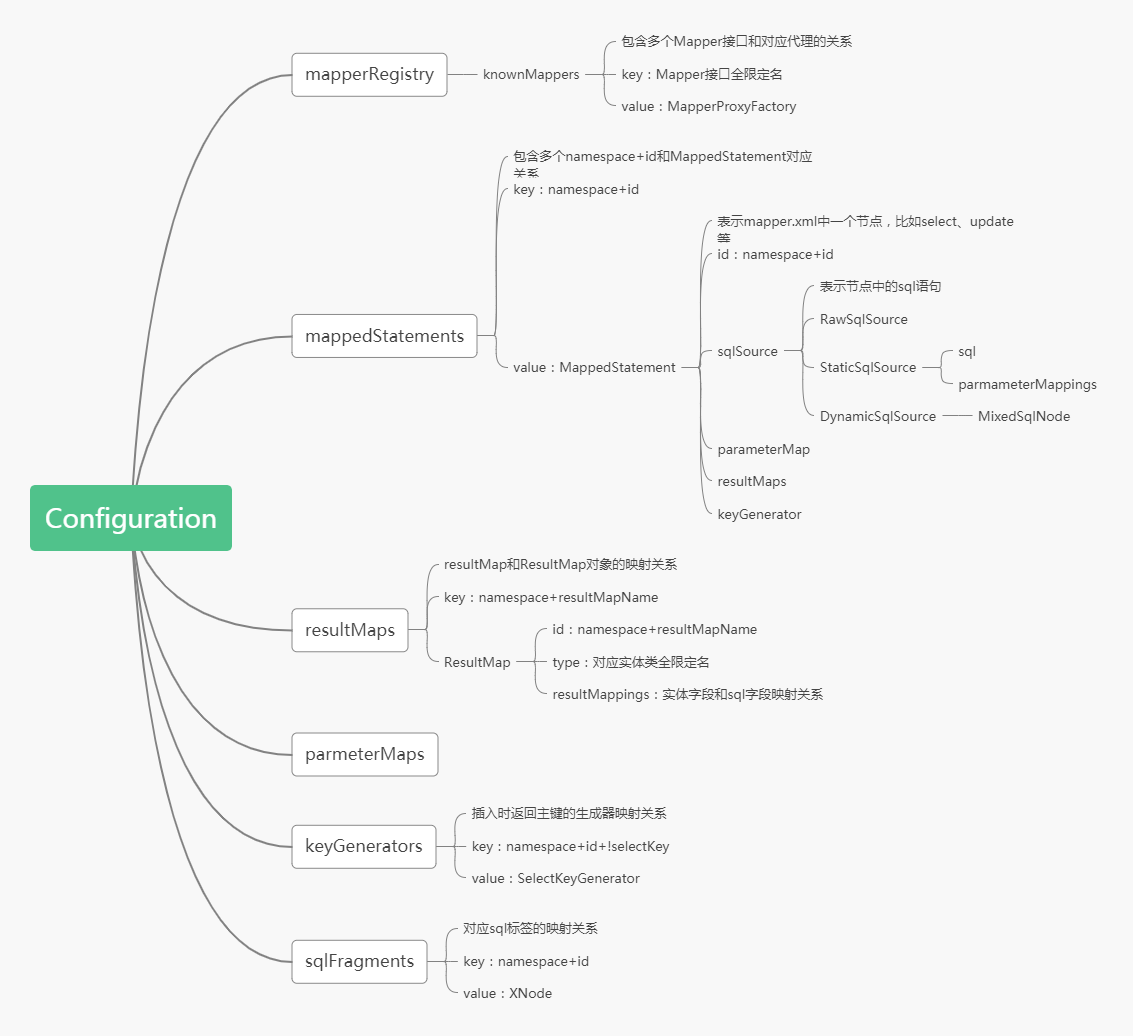
ResultMap：用于解析mapper.xml文件中的resultMap节点，使用resultMapping来封装id，result等子元素

MapperStatement，用于存储mapper.xml文件中的select，insert，update和delete节点，同时还包含了很多重要的属性

SQLSource：mapper.xml文件中的sql语句会被解析成sqlsource对象，经过解析sqlsource包含的语句最终仅仅包含？占位符，可以直接交给数据库执行

Mybatis初始化就是在创建SQLSessionFactory时执行的，其目的就是把配置文件的信息读取出来保存到Configuration中，然后将Configuration保存到SQLSessionFactory中

Configuration的类图解



### XMLConfigBuilder

XMLConfigBuilder主要负责解析mybatis-config.xml文件，解析的过程在parse方法中完成

### XMLMapperBuilder

解析mappers标签，并且解析映射文件，解析映射文件主要是交给XMLMapperBuilder来完成

XMLStatementBuilder:

XMLStatementBuilder: 主要负责解析映射配置文件中的SQL节点 ( …)负责解析的方法是parseStatementNode

### 解析过程

Reader reader = Resources.getResourceAsReader("mybatis-config.xml");

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

创建SqlSessionFactoryBuilder对象，加载工作在build方法中完成的；这里采用了构造者模式，创建SQLSessionfactory对象（DefaultSqlSessionFactory）

具体过程

#### Mybatis-config.xml

1.new SqlSessionFactoryBuilder() --build

1.1在build方法中创建XMLConfigBuilder对象，具体的解析工作就是由这个对象来完成的，解析后将配置文件的信息封装到Configuration类中

1.1.1在执行XMLConfigBuilder的构造函数中会创建XPathParser对象，这类中创建使用DOM方法解析xml配置文件，提供获取节点信息的方法

解析工作在XMLConfigBuilder对象的parse方法中完成

在parse方法中调用

org.apache.ibatis.builder.xml.XMLConfigBuilder.parseConfiguration(XNode)

在parseConfiguration方法中

propertiesElement方法解析properties标签，解析后将信息保存到Configuration的Properties变量variables

settingsAsProperties方法解析settings标签，

typeAliasesElement方法解析typeAliases标签，解析收将信息放到TypeAliasRegistry typeAliasRegistry = new TypeAliasRegistry();中

pluginElement方法解析plugins标签，

objectFactoryElement解析objectFactory标签

objectWrapperFactoryElement解析objectWrapperFactory标签

reflectorFactoryElement解析reflectorFactory标签

settingsElement将settings的子标签信息读取出来

environmentsElement解析environments标签

databaseIdProviderElement解析databaseIdProvider标签

typeHandlerElement解析typeHandlers标签

mapperElement解析mappers标签

上面这些方法都是将对应信息解析出来封装到Configuration中

在mapperElement方法中，获取到属性信息后，就会去解析映射文件（这里只分析xml配置方法，注解方法不考虑）

#### Mapper.xml

获取到映射文件路径后创建XMLMapperBuilder，XMLMapperBuilder就是用来完成映射文件的解析。

XMLMapperBuilder mapperParser = new XMLMapperBuilder(inputStream, configuration, url,configuration.getSqlFragments());

mapperParser.parse();

XMLMapperBuilder的构造函数，创建XPathParser对象，这类中创建使用DOM方法解析映射配置文件，提供获取节点信息的方法

org.apache.ibatis.builder.xml.XMLMapperBuilder.configurationElement(XNode)解析过程就是在这个方法中完成的

String namespace = context.getStringAttribute("namespace");解析命名空间

builderAssistant.setCurrentNamespace(namespace);

cacheRefElement(context.evalNode("cache-ref"));

cacheElement(context.evalNode("cache"));解析cache标签,将cache的配置属性封装到CacheBuilder中存放到Configuration中

parameterMapElement(context.evalNodes("/mapper/parameterMap"));解析parameterMap标签

resultMapElements(context.evalNodes("/mapper/resultMap"));解析resultMap标签，resultMap解析后通过ResultMapResolver将信息封装到

ResultMap中

sqlElement(context.evalNodes("/mapper/sql"));解析sql标签

解析后将信息保存到map中，key是id，value就是标签的内容

#### select|insert|update|delete

buildStatementFromContext(context.evalNodes("select|insert|update|delete"));解析select|insert|update|delete标签

解析是由XMLStatementBuilder来完成中，首先创建XMLStatementBuilder对象，然后执行XMLStatementBuilder对象的parseStatementNode

方法，解析各种属性和内容，然后将信息封装到MappedStatement

id:命名空间+标签的id

sqlSource:这个对象封装sql

parameterMap：

resultMaps：resultMap解析的存放容器

keyGenerator：主键生成器

## 操作数据库

### 执行数据库操作的方式

Mybatis初始化完成后获取SQLSessionfactory对象，创建SQLSession对象

通过SQLSession来执行sql,有2中方式

以查询为例

1. sqlSession.selectList(namespace+id);执行查询，使用命名空间+id来执行sql语句
2. sqlSession.getMapper(class)，执行getMapper方法获取映射接口的动态代理对象，通过代理对象来执行对应的sql（第二种方式在执行的过程中也是获取对应sql的命名空间+id，最终也是执行selectList方法，不过这样就和xml文件的耦合没有那么强了，如果有修改xml是不用修改java代码的）

### 创建SQLSession（DefaultSQLSession）

获取到SQLSessionFactory后，调用opensession来创建SQLSession

opensession的实现，内部调用

org.apache.ibatis.session.defaults.DefaultSqlSessionFactory.openSessionFromDataSource(ExecutorType, TransactionIsolationLevel, boolean)方法

这个方法的实现流程

1. 获取环境信息，创建事务工厂，然后获取事务
2. 调用Configuration的newExecutor方法来创建Executor对象

创建过程

根据配置的执行器类型创建对应的执行器实现对象（默认是SimpleExecutor），如果配置了开启一级缓存（默认开启的），那么就是创建CachingExecutor实例作为执行器用来处理一级缓存，并且采用装饰器模式将之前创建的执行器封装到缓存执行器中用来处理其他的sql执行过程

1. 创建DefaultSqlSession，将执行封装到DefaultSqlSession中，返回对象

### getMapper方法获取代理对象的过程

SQLSession调用getMapper方法获取到接口的代理实现

getMapper方法

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

return configuration.getMapper(type, this);

}

方法执行的是configuration的getMapper方法，在这个方法是执行的MapperRegistry的getMapper方法

MapperRegistry：mapper接口动态代理工厂类的注册中心，在mybatisproxy实现invocationHandler接口，

MapperProxyFactory用于生成动态代理的实例对象

在MapperRegistry的getMapper方法

创建代理对象工厂MapperProxyFactory，调用newInstance方法创建代理对象

@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);

}

MapperProxy映射代理，实现了java.lang.reflect.InvocationHandler，

也就是在这个类的invoke方法中实现了功能增强：执行sql，将sql的执行交给执行器来完成

### 执行sql的过程

代理对象执行调用对应的方法就会执行对应的sql;

在MapperProxy的invoke方法中，调用cachedInvoker方法中获取到MapperMethodInvoker的实现，MapperMethodInvoker的实现有 2个：

PlainMethodInvoker和DefaultMethodInvoker；执行是获取的是PlainMethodInvoker对象（具体为什么还不清楚）在PlainMethodInvoker中封装了MapperMethod

获取到PlainMethodInvoker对象后调用invoke方法，在方法中执行mapperMethod的execute方法（创建PlainMethodInvoker对象时会创建mapperMethod,这样就在mapperMethod创建sqlCommd(sql增删改查的类型，并且根据方法获取到了对应的MappedStatement对象)， sql就是在这个execute方法中执行的

在execute方法中操作的sql类型来决定执行select insert delete update

执行select时（只分析select的多结果集，其他的只用执行sql不需要返回，流程基本上都会出现在select的执行中，select处理要执行sql之外）还需要封装返回结果）

当返回的结果是结果集时，执行executeForMany方法，判断是否分页，要分页的话就创建rowBounds对象来获取分页的数据，不分页，直接执行sqlSession.selectList方法

也就是将执行sql的过程交给了sqlsession的实现也就是DefaultSqlSession

在DefaultSqlSession中最终还是交给了Executor来实现，执行Executor的query方法

执行过程

1．执行Executor.query方法

2.获取sql信息，boundSql

3.拼装cacheKey

上面这三步在org.apache.ibatis.executor.BaseExecutor.query(MappedStatement, Object, RowBounds, ResultHandler)方法中实现

4.判断是否需要清空一级缓存

5.根据cacheKey查找一级缓存

6.是否命中一级缓存

上面这几步在org.apache.ibatis.executor.CachingExecutor.query(MappedStatement, Object, RowBounds, ResultHandler, CacheKey, BoundSql)

查询数据库，

没有缓存的时候就执行queryFromDatabase查询，这个方法中执行doQuery方法，默认就是执行SimpleExecutor中的实现

**首先获取Configuration**

**首先获取Configuration**，创建StatementHandler的实现类RoutingStatementHandler对象，

调用org.apache.ibatis.executor.SimpleExecutor.prepareStatement(StatementHandler,

Log)方法、获取connection对象，如果配置了日志，那么返回的就是Connection的动态代理对象这就可以打印日志

**然后创建StatementHandler**

**然后创建StatementHandler**方法就是Configuration的newStatementHandler方法

内部实现的细节

首先创建RoutingStatementHandler对象，然后执行拦截器链，来创建StatementHandler的代理对象（如果定义了StatementHandler的拦截），

这样就是实现了对StatementHandler的功能增强（增强也就是执行接口的intercept方法，该方法的参数就是封装在Invocation中－－包含了目标对象，

执行的方法，个执行方法的参数）

执行RoutingStatementHandler对象的prepare，在prepare方法中委派给具体的实现类来创建preparestatement对象（具体的创建在

org.apache.ibatis.executor.statement.PreparedStatementHandler.instantiateStatement(Connection)方法中）

**sql参数赋值**

执行RoutingStatementHandler对象的parameterize方法来设置sql的参数，设置的过程交给parameterHandler来完成（DefaultParameterHandler）

org.apache.ibatis.scripting.defaults.DefaultParameterHandler.setParameters(PreparedStatement)

根据参数的类型执行不同获取不同的TypeHandler来实现sql值的设置（TypeHandler）

// 1. 获取boundSql中的参数映射信息列表

// 1.1. 遍历参数映射列表，这个列表信息就是我们xml文件中定义的某个查询语句的所有参数映射信息，注意这个List中的参数映射元素的顺序是和真实xml中sql的参数顺序对应的

获取每个属性对应的值

然后获取对应的参数类型处理器来设置占位符的值

**准备好preparestatement后执行sql**

org.apache.ibatis.executor.statement.PreparedStatementHandler.query(Statement, ResultHandler)）

调用ResultSetHandler的handleResultSets方法来来封装resultSet返回的结果

**最后结果根据情况保存到一级缓存**