**1 引言**

本文主要讲解JDBC怎么演变到Mybatis的渐变过程，**重点讲解了为什么要将JDBC封装成Mybaits这样一个持久层框架**。再而论述Mybatis作为一个数据持久层框架本身有待改进之处。

**2 JDBC实现查询分析**

我们先看看我们最熟悉也是最基础的通过JDBC查询数据库数据，一般需要以下七个步骤：

1. 加载JDBC驱动；
2. 建立并获取数据库连接；
3. 创建 JDBC Statements 对象；
4. 设置SQL语句的传入参数；
5. 执行SQL语句并获得查询结果；
6. 对查询结果进行转换处理并将处理结果返回；
7. 释放相关资源（关闭Connection，关闭Statement，关闭ResultSet）；

以下是具体的实现代码：

public static List<Map<String,Object>> queryForList(){

Connection connection = null;

ResultSet rs = null;

PreparedStatement stmt = null;

List<Map<String,Object>> resultList = new ArrayList<Map<String,Object>>();

try {

// 加载JDBC驱动

Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();

String url = "jdbc:oracle:thin:@localhost:1521:ORACLEDB";

String user = "trainer";

String password = "trainer";

// 获取数据库连接

connection = DriverManager.getConnection(url,user,password);

String sql = "select \* from userinfo where user\_id = ? ";

// 创建Statement对象（每一个Statement为一次数据库执行请求）

stmt = connection.prepareStatement(sql);

// 设置传入参数

stmt.setString(1, "zhangsan");

// 执行SQL语句

rs = stmt.executeQuery();

// 处理查询结果（将查询结果转换成List<Map>格式）

ResultSetMetaData rsmd = rs.getMetaData();

int num = rsmd.getColumnCount();

while(rs.next()){

Map map = new HashMap();

for(int i = 0;i < num;i++){

String columnName = rsmd.getColumnName(i+1);

map.put(columnName,rs.getString(columnName));

}

resultList.add(map);

}

} catch (Exception e) {

e.printStackTrace();

} finally {

try {

// 关闭结果集

if (rs != null) {

rs.close();

rs = null;

}

// 关闭执行

if (stmt != null) {

stmt.close();

stmt = null;

}

if (connection != null) {

connection.close();

connection = null;

}

} catch (SQLException e) {

e.printStackTrace();

}

}

return resultList;

}

**3 JDBC演变到Mybatis过程**

上面我们看到了实现JDBC有七个步骤，哪些步骤是可以进一步封装的，减少我们开发的代码量。

**3.1 第一步优化：连接获取和释放**

1. **问题描述：**

数据库连接频繁的开启和关闭本身就造成了**资源的浪费，影响系统的性能**。

**解决问题：**

数据库连接的获取和关闭我们**可以使用数据库连接池来解决资源浪费的问题**。通过连接池就可以反复利用已经建立的连接去访问数据库了。减少连接的开启和关闭的时间。

1. **问题描述：**

但是现在**连接池多种多样，可能存在变化**，有可能采用DBCP的连接池，也有可能采用容器本身的JNDI数据库连接池。

**解决问题：**

我们可以**通过DataSource进行隔离解耦**，我们统一从DataSource里面获取数据库连接，**DataSource具体由DBCP实现还是由容器的JNDI实现都可以**，所以我们将DataSource的具体实现通过让用户配置来应对变化。

**3.2 第二步优化：SQL统一存取**

1. **问题描述：**

我们使用JDBC进行操作数据库时，**SQL语句基本都散落在各个JAVA类中**，这样有三个不足之处：

第一，可读性很差，不利于维护以及做性能调优。

第二，改动Java代码需要重新编译、打包部署。

第三，不利于取出SQL在数据库客户端执行（取出后还得删掉中间的Java代码，编写好的SQL语句写好后还得通过＋号在Java进行拼凑）。

**解决问题：**

我们可以考虑不把SQL语句写到Java代码中，那么把SQL语句放到哪里呢？首先需要有一个统一存放的地方，我们可以将这些**SQL语句统一集中放到配置文件或者数据库里面（以key-value的格式存放）**。然后通过SQL语句的key值去获取对应的SQL语句。

既然我们将SQL语句都统一放在配置文件或者数据库中，**那么这里就涉及一个SQL语句的加载问题**。

**3.3 第三步优化：传入参数映射和动态SQL**

1. **问题描述：**

很多情况下，我们都可以通过在SQL语句中设置占位符来达到使用传入参数的目的，这种方式本身就有一定局限性，它是按照一定顺序传入参数的，要与占位符一一匹配。但是，如果我们**传入的参数是不确定的**（比如列表查询，根据用户填写的查询条件不同，传入查询的参数也是不同的，有时是一个参数、有时可能是三个参数），那么我们就得**在后台代码中自己根据请求的传入参数去拼凑相应的SQL语句**，这样的话还是**避免不了在Java代码里面写SQL语句的命运**。既然我们已经把SQL语句统一存放在配置文件或者数据库中了，**怎么做到能够根据前台传入参数的不同，动态生成对应的SQL语句呢？**

**解决问题：**

第一，我们先解决这个动态问题，**按照我们正常的程序员思维是，通过if和else这类的判断来进行是最直观的**，这个时候我们想到了JSTL中的<if test=””></if>这样的标签，那么，能不能将这类的标签引入到SQL语句中呢？假设可以，那么我们这里就需要一个专门的SQL解析器来解析这样的SQL语句，但是，if判断的变量来自于哪里呢？传入的值本身是可变的，那么我们得为这个值定义一个不变的变量名称，而且这个变量名称必须和对应的值要有对应关系，可以通过这个变量名称找到对应的值，这个时候我们想到了key-value的Map。解析的时候根据变量名的具体值来判断。

假如前面可以判断没有问题，那么假如判断的结果是true，那么就需要输出的标签里面的SQL片段，但是怎么解决在标签里面使用变量名称的问题呢？这里我们需要**使用一种有别于SQL的语法来嵌入变量（比如使用＃变量名＃）**。这样，SQL语句经过解析后就可以动态的生成符合上下文的SQL语句。

还有，**怎么区分开占位符变量和非占位变量？**有时候我们单单使用占位符是满足不了的，占位符只能为查询条件占位，SQL语句其他地方使用不了。**这里我们可以使用#变量名#表示占位符变量，使用$变量名$表示非占位符变量**。

**3.4 第四步优化：结果映射和结果缓存**

1. **问题描述：**

执行SQL语句、获取执行结果、对执行结果进行转换处理、释放相关资源是一整套下来的。假如是执行查询语句，那么执行SQL语句后，返回的是一个ResultSet结果集，**这个时候我们就需要将ResultSet对象的数据取出来，不然等到释放资源时就取不到这些结果信息了**。我们从前面的优化来看，以及将获取连接、设置传入参数、执行SQL语句、释放资源这些都封装起来了，只剩下结果处理这块还没有进行封装，如果能封装起来，每个数据库操作都不用自己写那么一大堆Java代码，直接调用一个封装的方法就可以搞定了。

**解决问题：**

我们分析一下，一般对执行结果的有哪些处理，**有可能将结果不做任何处理就直接返回，也有可能将结果转换成一个JavaBean对象返回、一个Map返回、一个List返回等**`，结果处理可能是多种多样的。从这里看，我们必须告诉SQL处理器两点：**第一，需要返回什么类型的对象；第二，需要返回的对象的数据结构怎么跟执行的结果映射**，这样才能将具体的值copy到对应的数据结构上。

接下来，**我们可以进而考虑对SQL执行结果的缓存来提升性能**。缓存数据都是key-value的格式，那么这个key怎么来呢？怎么保证唯一呢？即使同一条SQL语句几次访问的过程中由于传入参数的不同，得到的执行SQL语句也是不同的。那么缓存起来的时候是多对。**但是SQL语句和传入参数两部分合起来可以作为数据缓存的key值**。

**3.5 第五步优化：解决重复SQL语句问题**

1. **问题描述：**

由于我们将所有SQL语句都放到配置文件中，**这个时候会遇到一个SQL重复的问题**，几个功能的SQL语句其实都差不多，有些可能是SELECT后面那段不同、有些可能是WHERE语句不同。有时候表结构改了，那么我们就需要改多个地方，不利于维护。

**解决问题：**

当我们的代码程序出现重复代码时怎么办？**将重复的代码抽离出来成为独立的一个类，然后在各个需要使用的地方进行引用**。对于SQL重复的问题，我们也可以采用这种方式，通过将SQL片段模块化，**将重复的SQL片段独立成一个SQL块，然后在各个SQL语句引用重复的SQL块**，这样需要修改时只需要修改一处即可。

**4 Mybaits有待改进之处**

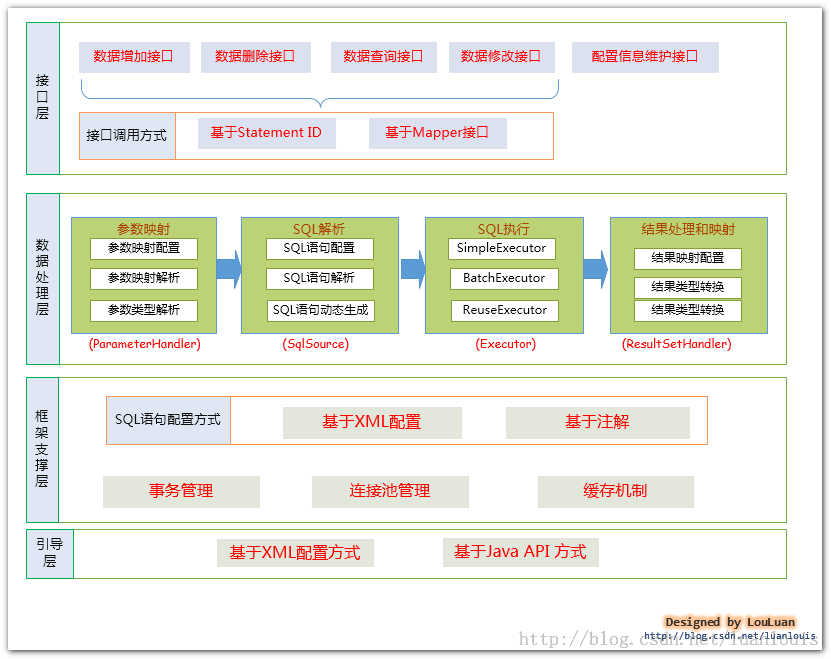
1. **问题描述：**

Mybaits所有的数据库操作都是基于SQL语句，**导致什么样的数据库操作都要写SQL语句**。一个应用系统要写的SQL语句实在太多了。

**改进方法：**

我们对数据库进行的操作大部分都是对表数据的增删改查，很多都是对单表的数据进行操作，由这点我们可以想到一个问题：**单表操作可不可以不写SQL语句，通过JavaBean的默认映射器生成对应的SQL语句**，比如：一个类UserInfo对应于USER\_INFO表， userId属性对应于USER\_ID字段。**这样我们就可以通过反射可以获取到对应的表结构了，拼凑成对应的SQL语句显然不是问题**。

**5 MyBatis框架整体设计**



MyBatis框架整体设计

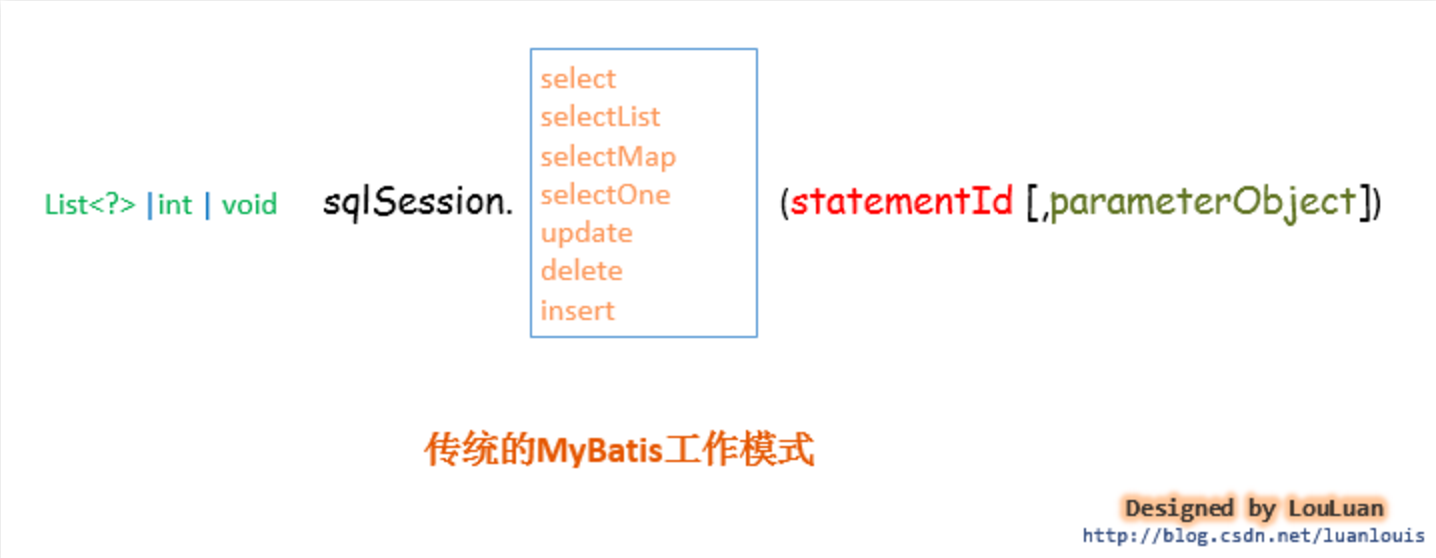
**5.1 接口层-和数据库交互的方式**

MyBatis和数据库的交互有两种方式：

1. 使用传统的MyBatis提供的API；
2. 使用Mapper接口；

**5.1.1 使用传统的MyBatis提供的API**

**这是传统的传递Statement Id 和查询参数给 SqlSession 对象，使用 SqlSession对象完成和数据库的交互**；MyBatis提供了非常方便和简单的API，供用户实现对数据库的增删改查数据操作，以及对数据库连接信息和MyBatis 自身配置信息的维护操作。



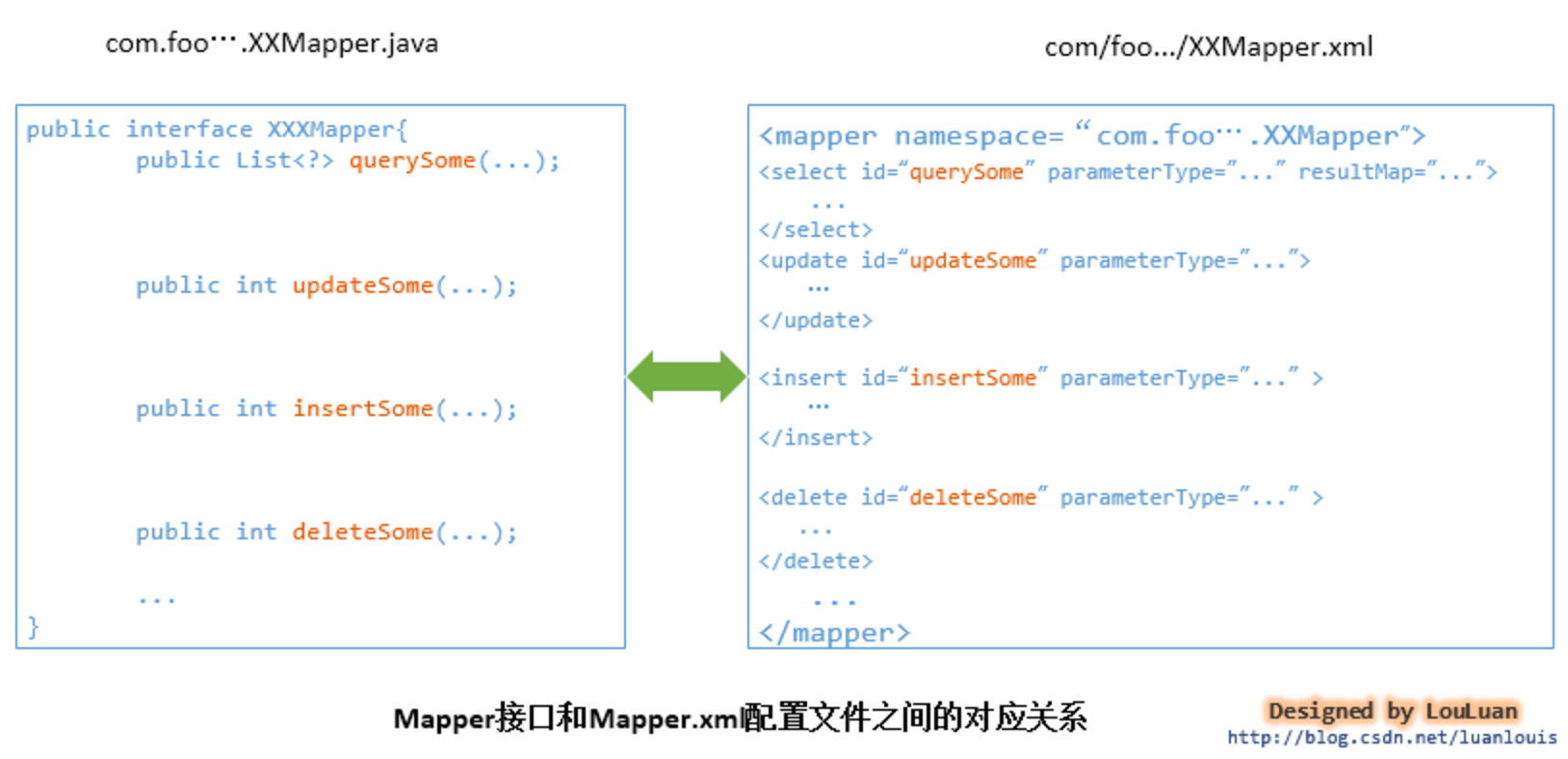
传统的MyBatis工作模式

上述使用MyBatis 的方法，是**创建一个和数据库打交道的SqlSession对象，然后根据Statement Id 和参数来操作数据库**，这种方式固然很简单和实用，但是**它不符合面向对象语言的概念和面向接口编程的编程习惯**。由于面向接口的编程是面向对象的大趋势，MyBatis 为了适应这一趋势，增加了第二种使用MyBatis 支持接口（Interface）调用方式。

**5.1.2 使用Mapper接口**

**MyBatis 将配置文件中的每一个<mapper> 节点抽象为一个 Mapper 接口：**

**这个接口中声明的方法和<mapper> 节点中的<select|update|delete|insert> 节点项对应**，即<select|update|delete|insert> 节点的id值为Mapper 接口中的方法名称，**parameterType 值表示Mapper 对应方法的入参类型**，而**resultMap 值则对应了Mapper 接口表示的返回值类型或者返回结果集的元素类型**。



Mapper接口和Mapper.xml配置文件之间的对应关系

**根据MyBatis 的配置规范配置好后，通过SqlSession.getMapper(XXXMapper.class)方法，MyBatis 会根据相应的接口声明的方法信息，通过动态代理机制生成一个Mapper 实例**，我们使用Mapper接口的某一个方法时，MyBatis会根据这个方法的方法名和参数类型，确定Statement Id，底层还是通过SqlSession.select("statementId",parameterObject);或者SqlSession.update("statementId",parameterObject); 等等来实现对数据库的操作，**MyBatis引用Mapper 接口这种调用方式，纯粹是为了满足面向接口编程的需要**。（其实还有一个原因是在于，面向接口的编程，使得用户在接口上可以使用注解来配置SQL语句，这样就可以脱离XML配置文件，实现“0配置”）。

**5.2 数据处理层**

**数据处理层可以说是MyBatis的核心**，从大的方面上讲，它要完成两个功能：

1. 通过传入参数构建动态SQL语句；
2. SQL语句的执行以及封装查询结果集成List<E>；

**5.2.1 参数映射和动态SQL语句生成**

动态语句生成可以说是MyBatis框架非常优雅的一个设计，**MyBatis 通过传入的参数值，使用 Ognl 来动态地构造SQL语句**，使得MyBatis 有很强的灵活性和扩展性。

**参数映射指的是对于java 数据类型和jdbc数据类型之间的转换：**这里有包括两个过程：**查询阶段**，我们要将java类型的数据，转换成jdbc类型的数据，通过 preparedStatement.setXXX() 来设值；另一个就是**对resultset查询结果集的jdbcType 数据转换成java 数据类型**。

**5.2.2 SQL语句的执行以及封装查询结果集成List<E>**

动态SQL语句生成之后，MyBatis 将执行SQL语句，并将可能返回的结果集转换成List<E> 列表。**MyBatis 在对结果集的处理中，支持结果集关系一对多和多对一的转换**，并且有两种支持方式，**一种为嵌套查询语句的查询，还有一种是嵌套结果集的查询**。

**5.3 框架支撑层**

1. 事务管理机制

**事务管理机制对于ORM框架而言是不可缺少的一部分**，事务管理机制的质量也是考量一个ORM框架是否优秀的一个标准。

1. 连接池管理机制

由于创建一个数据库连接所占用的资源比较大，**对于数据吞吐量大和访问量非常大的应用而言，连接池的设计就显得非常重要**。

1. 缓存机制

为了提高数据利用率和减小服务器和数据库的压力，**MyBatis 会对于一些查询提供会话级别的数据缓存**，会将对某一次查询，放置到SqlSession 中，在允许的时间间隔内，对于完全相同的查询，MyBatis会直接将缓存结果返回给用户，而不用再到数据库中查找。

1. SQL语句的配置方式

传统的MyBatis 配置SQL语句方式就是使用XML文件进行配置的，但是这种方式不能很好地支持面向接口编程的理念，**为了支持面向接口的编程，MyBatis 引入了Mapper接口的概念，面向接口的引入，对使用注解来配置SQL语句成为可能，用户只需要在接口上添加必要的注解即可，不用再去配置XML文件了**，但是，目前的MyBatis 只是对注解配置SQL语句提供了有限的支持，某些高级功能还是要依赖XML配置文件配置SQL 语句。

**5.4 引导层**

**引导层是配置和启动MyBatis配置信息的方式**。MyBatis 提供两种方式来引导MyBatis ：**基于XML配置文件的方式和基于Java API 的方式**。

**5.5 主要构件及其相互关系**

从MyBatis代码实现的角度来看，MyBatis的主要的核心部件有以下几个：

**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；

**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；

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

**ParameterHandler：**负责对用户传递的参数转换成JDBC Statement 所需要的参数；

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

**TypeHandler：**负责java数据类型和jdbc数据类型之间的映射和转换；

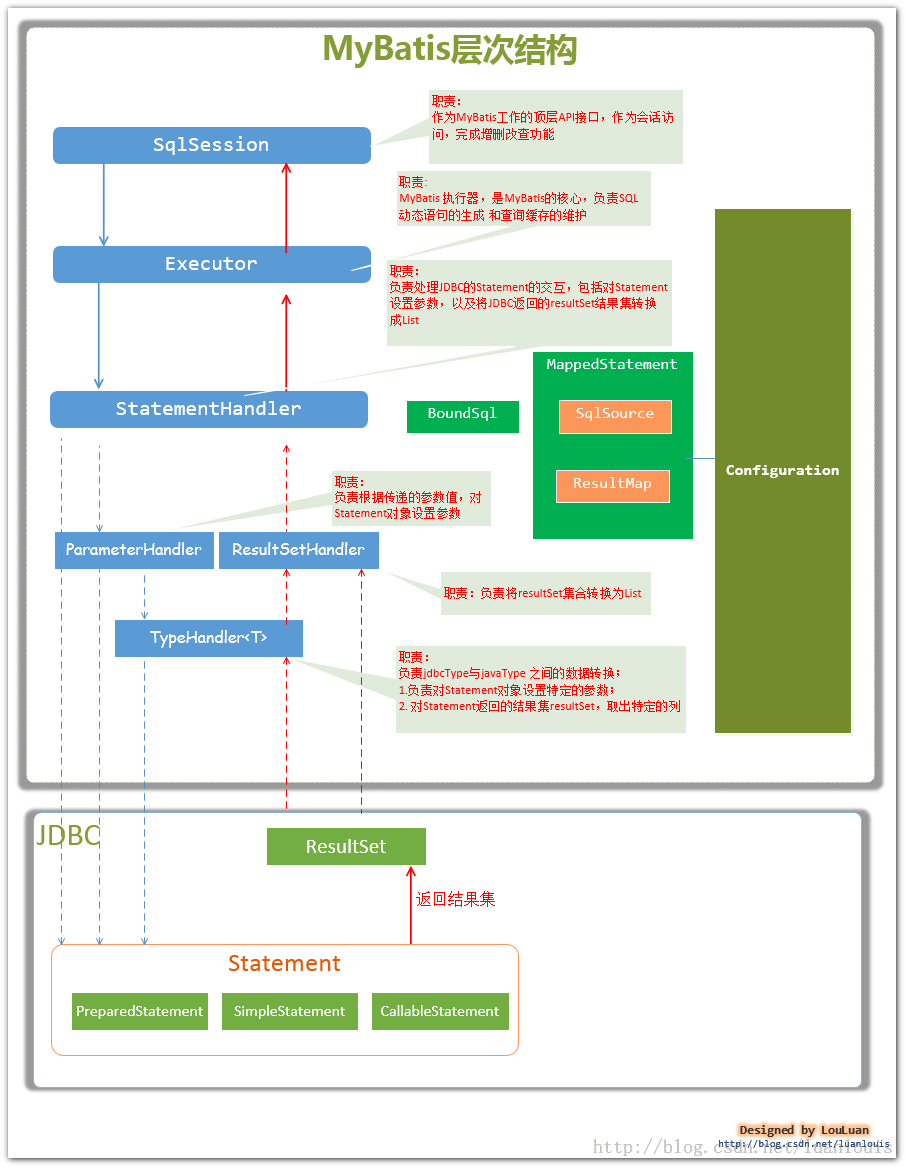
**MappedStatement：**MappedStatement维护了一条<select|update|delete|insert>节点的封装；

**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；

**BoundSql：**表示动态生成的SQL语句以及相应的参数信息；

**Configuration：**MyBatis所有的配置信息都维持在Configuration对象之中；

**它们的关系如下图所示：**

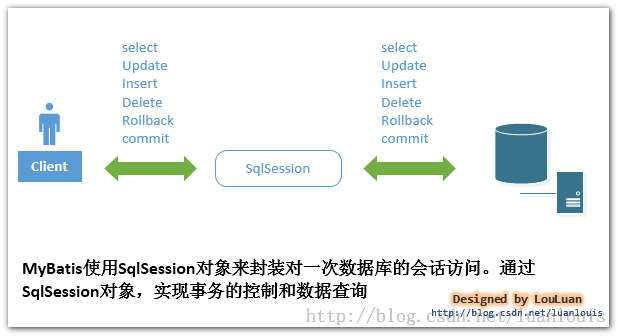


MyBatis主要构件关系如图

**6 SqlSession工作过程分析**

1. **开启一个数据库访问会话---创建SqlSession对象**

SqlSession sqlSession = factory.openSession();



MyBatis封装了对数据库的访问，把对数据库的会话和事务控制放到了SqlSession对象中

1. **为SqlSession传递一个配置的Sql语句的Statement Id和参数，然后返回结果：**

List<Employee> result = sqlSession.selectList("com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary",params);

上述的"com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary"，是配置在EmployeesMapper.xml 的Statement ID，params是传递的查询参数。

让我们来看一下sqlSession.selectList()方法的定义：

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

return this.selectList(statement, parameter, RowBounds.DEFAULT);

}

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

try {

//1.根据Statement Id，在mybatis 配置对象Configuration中查找和配置文件相对应的MappedStatement

MappedStatement ms = configuration.getMappedStatement(statement);

//2. 将查询任务委托给MyBatis 的执行器 Executor

List<E> result = executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO\_RESULT\_HANDLER);

return result;

} catch (Exception e) {

throw ExceptionFactory.wrapException("Error querying database. Cause: " + e, e);

} finally {

ErrorContext.instance().reset();

}

}

MyBatis在初始化的时候，会将MyBatis的配置信息全部加载到内存中，**使用org.apache.ibatis.session.Configuration实例来维护**。使用者可以使用sqlSession.getConfiguration()方法来获取。**MyBatis的配置文件中配置信息的组织格式和内存中对象的组织格式几乎完全对应的**。

上述例子中的：

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

select

EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, SALARY

from LOUIS.EMPLOYEES

<if test="min\_salary != null">

where SALARY < #{min\_salary,jdbcType=DECIMAL}

</if>

</select>

**加载到内存中会生成一个对应的MappedStatement对象，然后会以key="com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary" ，value为MappedStatement对象的形式维护到Configuration的一个Map中**。当以后需要使用的时候，只需要通过Id值来获取就可以了。

从上述的代码中我们可以看到SqlSession的职能是：**SqlSession根据Statement ID, 在mybatis配置对象Configuration中获取到对应的MappedStatement对象，然后调用mybatis执行器来执行具体的操作**。

1. **MyBatis执行器Executor根据SqlSession传递的参数执行query()方法（由于代码过长，读者只需阅读我注释的地方即可）：**

/\*\*

\* BaseExecutor 类部分代码

\*

\*/

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

// 1. 根据具体传入的参数，动态地生成需要执行的SQL语句，用BoundSql对象表示

BoundSql boundSql = ms.getBoundSql(parameter);

// 2. 为当前的查询创建一个缓存Key

CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);

return query(ms, parameter, rowBounds, resultHandler, key, boundSql);

}

@SuppressWarnings("unchecked")

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

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 {

// 3.缓存中没有值，直接从数据库中读取数据

list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key, boundSql);

}

} finally {

queryStack--;

}

if (queryStack == 0) {

for (DeferredLoad deferredLoad : deferredLoads) {

deferredLoad.load();

}

deferredLoads.clear(); // issue #601

if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {

clearLocalCache(); // issue #482

}

}

return list;

}

private <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {

List<E> list;

localCache.putObject(key, EXECUTION\_PLACEHOLDER);

try {

//4. 执行查询，返回List 结果，然后 将查询的结果放入缓存之中

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;

}

/\*\*

\*

\* SimpleExecutor类的doQuery()方法实现

\*

\*/

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();

//5. 根据既有的参数，创建StatementHandler对象来执行查询操作

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

//6. 创建java.Sql.Statement对象，传递给StatementHandler对象

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

//7. 调用StatementHandler.query()方法，返回List结果集

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

} finally {

closeStatement(stmt);

}

}

上述的Executor.query()方法几经转折，**最后会创建一个StatementHandler对象，然后将必要的参数传递给StatementHandler**，使用StatementHandler来完成对数据库的查询，最终返回List结果集。

**从上面的代码中我们可以看出，Executor的功能和作用是：**

1. 根据传递的参数，完成SQL语句的动态解析，生成BoundSql对象，供StatementHandler使用；
2. 为查询创建缓存，以提高性能；
3. 创建JDBC的Statement连接对象，传递给StatementHandler对象，返回List查询结果；
4. **StatementHandler对象负责设置Statement对象中的查询参数、处理JDBC返回的resultSet，将resultSet加工为List 集合返回：**

接着上面的Executor第六步，看一下：prepareStatement() 方法的实现：

/\*\*

\*

\* SimpleExecutor类的doQuery()方法实现

\*

\*/

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);

// 1.准备Statement对象，并设置Statement对象的参数

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

// 2. StatementHandler执行query()方法，返回List结果

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

} finally {

closeStatement(stmt);

}

}

private Statement prepareStatement(StatementHandler handler, Log statementLog) throws SQLException {

Statement stmt;

Connection connection = getConnection(statementLog);

stmt = handler.prepare(connection);

//对创建的Statement对象设置参数，即设置SQL 语句中 ? 设置为指定的参数

handler.parameterize(stmt);

return stmt;

}

以上我们可以总结StatementHandler对象主要完成两个工作：

1. 对于JDBC的PreparedStatement类型的对象，创建的过程中，我们使用的是SQL语句字符串会包含 若干个? 占位符，我们其后再对占位符进行设值。  
   **StatementHandler通过parameterize(statement)方法对Statement进行设值；**
2. StatementHandler通过List<E> query(Statement statement, ResultHandler resultHandler)方法来完成执行Statement，和将Statement对象返回的resultSet封装成List；
3. **StatementHandler 的parameterize(statement) 方法的实现：**

/\*\*

\* StatementHandler 类的parameterize(statement) 方法实现

\*/

public void parameterize(Statement statement) throws SQLException {

//使用ParameterHandler对象来完成对Statement的设值

parameterHandler.setParameters((PreparedStatement) statement);

}

/\*\*

\*

\* ParameterHandler类的setParameters(PreparedStatement ps) 实现

\* 对某一个Statement进行设置参数

\*/

public void setParameters(PreparedStatement ps) throws SQLException {

ErrorContext.instance().activity("setting parameters").object(mappedStatement.getParameterMap().getId());

List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();

if (parameterMappings != null) {

for (int i = 0; i < parameterMappings.size(); i++) {

ParameterMapping parameterMapping = parameterMappings.get(i);

if (parameterMapping.getMode() != ParameterMode.OUT) {

Object value;

String propertyName = parameterMapping.getProperty();

if (boundSql.hasAdditionalParameter(propertyName)) { // issue #448 ask first for additional params

value = boundSql.getAdditionalParameter(propertyName);

} else if (parameterObject == null) {

value = null;

} else if (typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) {

value = parameterObject;

} else {

MetaObject metaObject = configuration.newMetaObject(parameterObject);

value = metaObject.getValue(propertyName);

}

// 每一个Mapping都有一个TypeHandler，根据TypeHandler来对preparedStatement进行设置参数

TypeHandler typeHandler = parameterMapping.getTypeHandler();

JdbcType jdbcType = parameterMapping.getJdbcType();

if (value == null && jdbcType == null) jdbcType = configuration.getJdbcTypeForNull();

// 设置参数

typeHandler.setParameter(ps, i + 1, value, jdbcType);

}

}

}

}

从上述的代码可以看到,StatementHandler的parameterize(Statement) 方法调用了 ParameterHandler的setParameters(statement) 方法，  
**ParameterHandler的setParameters(Statement)方法负责 根据我们输入的参数，对statement对象的 ? 占位符处进行赋值。**

1. **StatementHandler 的List<E> query(Statement statement, ResultHandler resultHandler)方法的实现：**

/\*\*

\* PreParedStatement类的query方法实现

\*/

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

//1.调用preparedStatemnt。execute()方法，然后将resultSet交给ResultSetHandler处理

PreparedStatement ps = (PreparedStatement) statement;

ps.execute();

//2. 使用ResultHandler来处理ResultSet

return resultSetHandler.<E> handleResultSets(ps);

}

从上述代码我们可以看出，StatementHandler 的List<E> query(Statement statement, ResultHandler resultHandler)方法的实现，是调用了ResultSetHandler的handleResultSets(Statement) 方法。**ResultSetHandler的handleResultSets(Statement) 方法会将Statement语句执行后生成的resultSet 结果集转换成List<E> 结果集**：

/\*\*

\* ResultSetHandler类的handleResultSets()方法实现

\*

\*/

public List<Object> handleResultSets(Statement stmt) throws SQLException {

final List<Object> multipleResults = new ArrayList<Object>();

int resultSetCount = 0;

ResultSetWrapper rsw = getFirstResultSet(stmt);

List<ResultMap> resultMaps = mappedStatement.getResultMaps();

int resultMapCount = resultMaps.size();

validateResultMapsCount(rsw, resultMapCount);

while (rsw != null && resultMapCount > resultSetCount) {

ResultMap resultMap = resultMaps.get(resultSetCount);

//将resultSet

handleResultSet(rsw, resultMap, multipleResults, null);

rsw = getNextResultSet(stmt);

cleanUpAfterHandlingResultSet();

resultSetCount++;

}

String[] resultSets = mappedStatement.getResulSets();

if (resultSets != null) {

while (rsw != null && resultSetCount < resultSets.length) {

ResultMapping parentMapping = nextResultMaps.get(resultSets[resultSetCount]);

if (parentMapping != null) {

String nestedResultMapId = parentMapping.getNestedResultMapId();

ResultMap resultMap = configuration.getResultMap(nestedResultMapId);

handleResultSet(rsw, resultMap, null, parentMapping);

}

rsw = getNextResultSet(stmt);

cleanUpAfterHandlingResultSet();

resultSetCount++;

}

}

return collapseSingleResultList(multipleResults);

}

**7 MyBatis初始化机制**

**7.1 MyBatis的初始化做了什么**

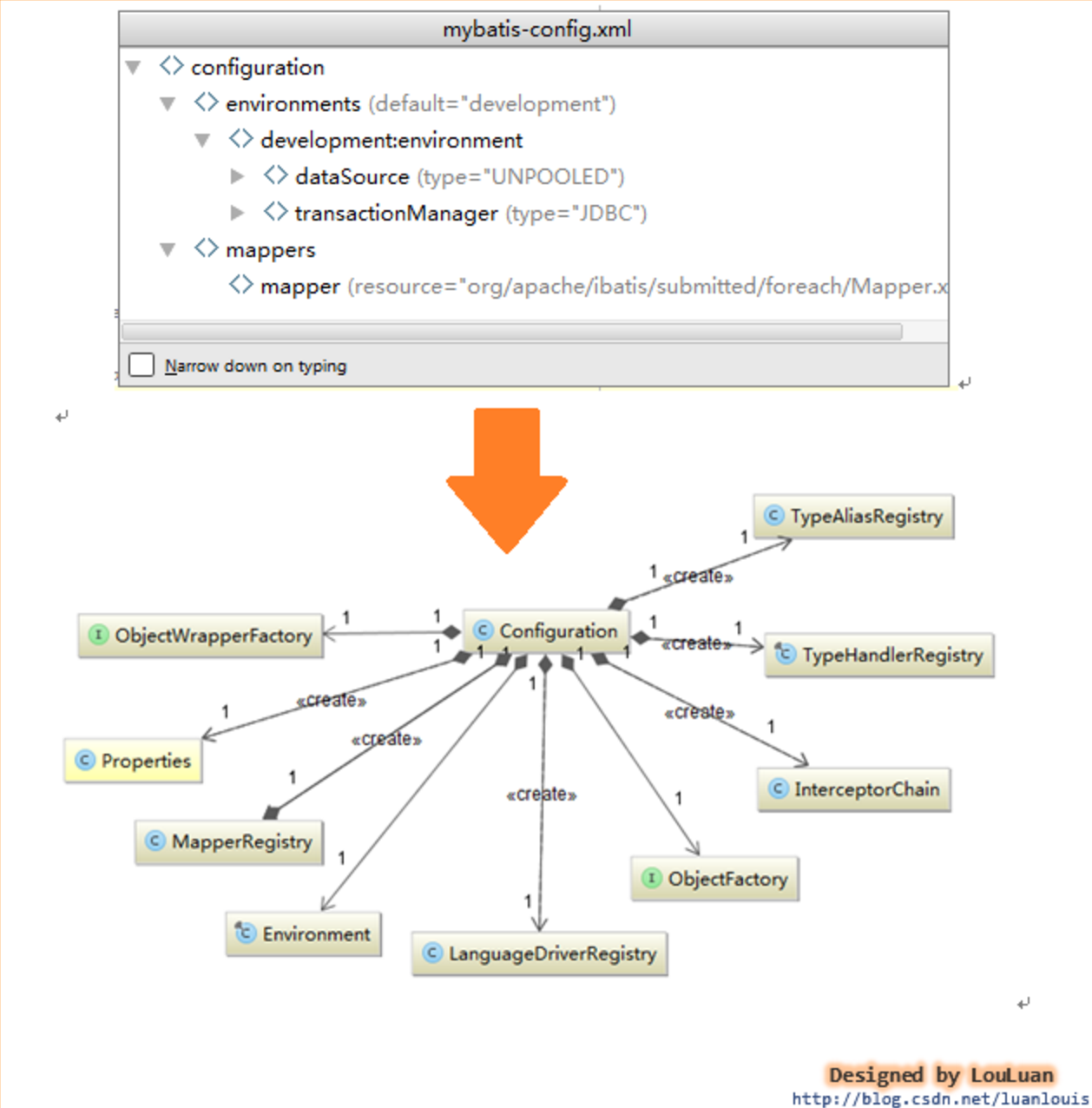
**任何框架的初始化，无非是加载自己运行时所需要的配置信息。**MyBatis的配置信息，大概包含以下信息，其高层级结构如下：



MyBatis配置信息结构图

**MyBatis的上述配置信息会配置在XML配置文件中，那么，这些信息被加载进入MyBatis内部，MyBatis是怎样维护的呢？**

MyBatis采用了一个非常直白和简单的方式---**使用 org.apache.ibatis.session.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）。如下图所示：



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

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。

接下来我们将通过 基于XML配置文件方式的MyBatis初始化，深入探讨MyBatis是如何通过配置文件构建Configuration对象，并使用它。

**7.2 基于XML配置文件创建Configuration对象**

现在就从使用MyBatis的简单例子入手，深入分析一下MyBatis是怎样完成初始化的，都初始化了什么。看以下代码：

String resource = "mybatis-config.xml";

InputStream inputStream = Resources.getResourceAsStream(resource);

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

SqlSession sqlSession = sqlSessionFactory.openSession();

List list = sqlSession.selectList("com.foo.bean.BlogMapper.queryAllBlogInfo");

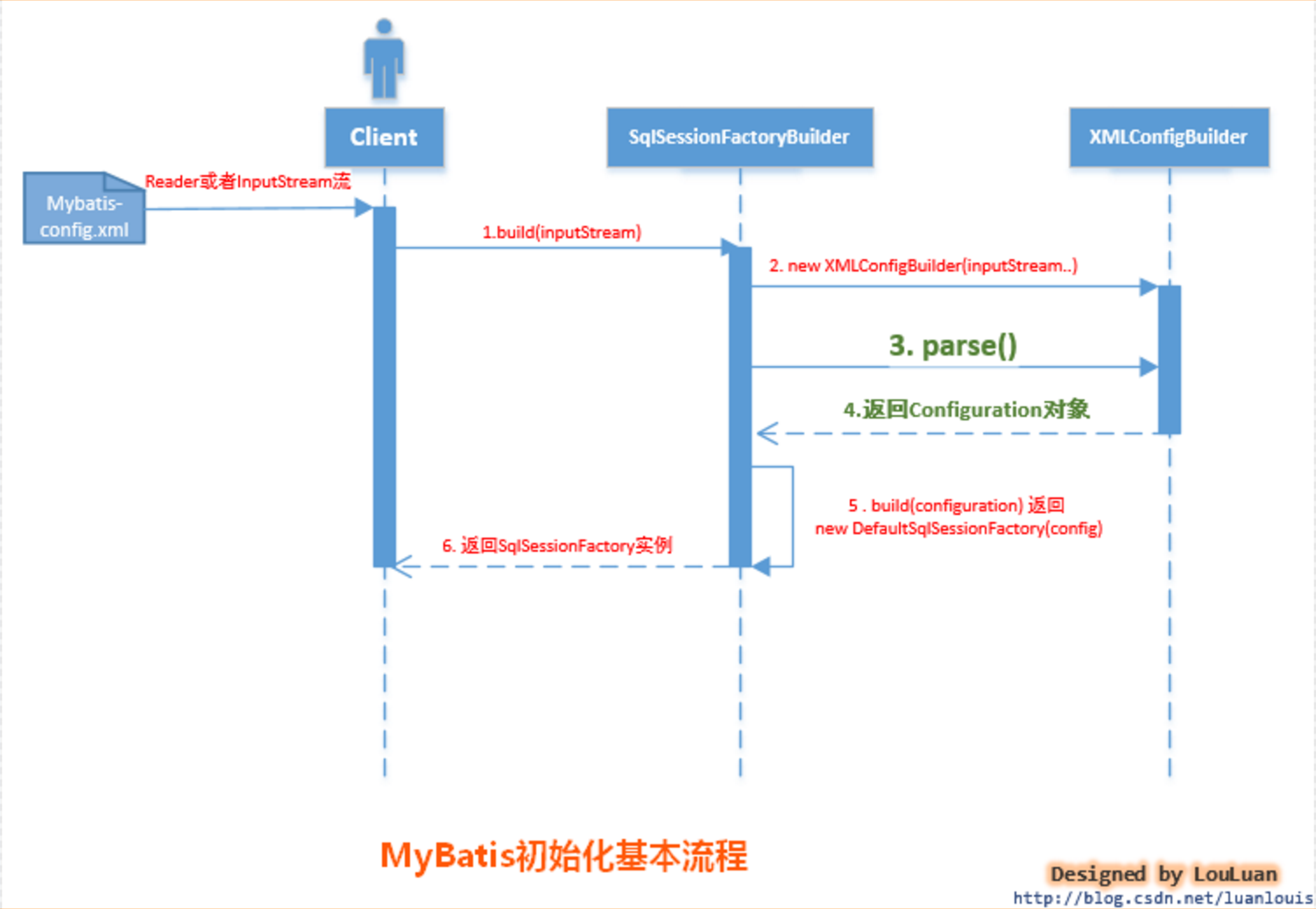
有过MyBatis使用经验的读者会知道，上述语句的作用是执行com.foo.bean.BlogMapper.queryAllBlogInfo 定义的SQL语句，返回一个List结果集。总的来说，上述代码经历了**mybatis初始化 -->创建SqlSession -->执行SQL语句**返回结果三个过程。

上述代码的功能是根据配置文件mybatis-config.xml 配置文件，创建SqlSessionFactory对象，然后产生SqlSession，执行SQL语句。**而mybatis的初始化就发生在第三句：SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);** 现在就让我们看看第三句到底发生了什么。

1. **MyBatis初始化基本过程：**

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

**初始化的基本过程如下序列图所示：**



MyBatis初始化序列图

**由上图所示，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。

**SqlSessionFactoryBuilder相关的代码如下所示：**

public SqlSessionFactory build(InputStream inputStream) {

return build(inputStream, null, null);

}

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

try {

//2. 创建XMLConfigBuilder对象用来解析XML配置文件，生成Configuration对象

XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);

//3. 将XML配置文件内的信息解析成Java对象Configuration对象

Configuration config = parser.parse();

//4. 根据Configuration对象创建出SqlSessionFactory对象

return build(config);

} catch (Exception e) {

throw ExceptionFactory.wrapException("Error building SqlSession.", e);

} finally {

ErrorContext.instance().reset();

try {

inputStream.close();

} catch (IOException e) {

// Intentionally ignore. Prefer previous error.

}

}

}

// 从此处可以看出，MyBatis内部通过Configuration对象来创建SqlSessionFactory,用户也可以自己通过API构造好Configuration对象，调用此方法创SqlSessionFactory

public SqlSessionFactory build(Configuration config) {

return new DefaultSqlSessionFactory(config);

}

上述的初始化过程中，涉及到了以下几个对象：

**SqlSessionFactoryBuilder ：**SqlSessionFactory的构造器，用于创建SqlSessionFactory，采用了Builder设计模式

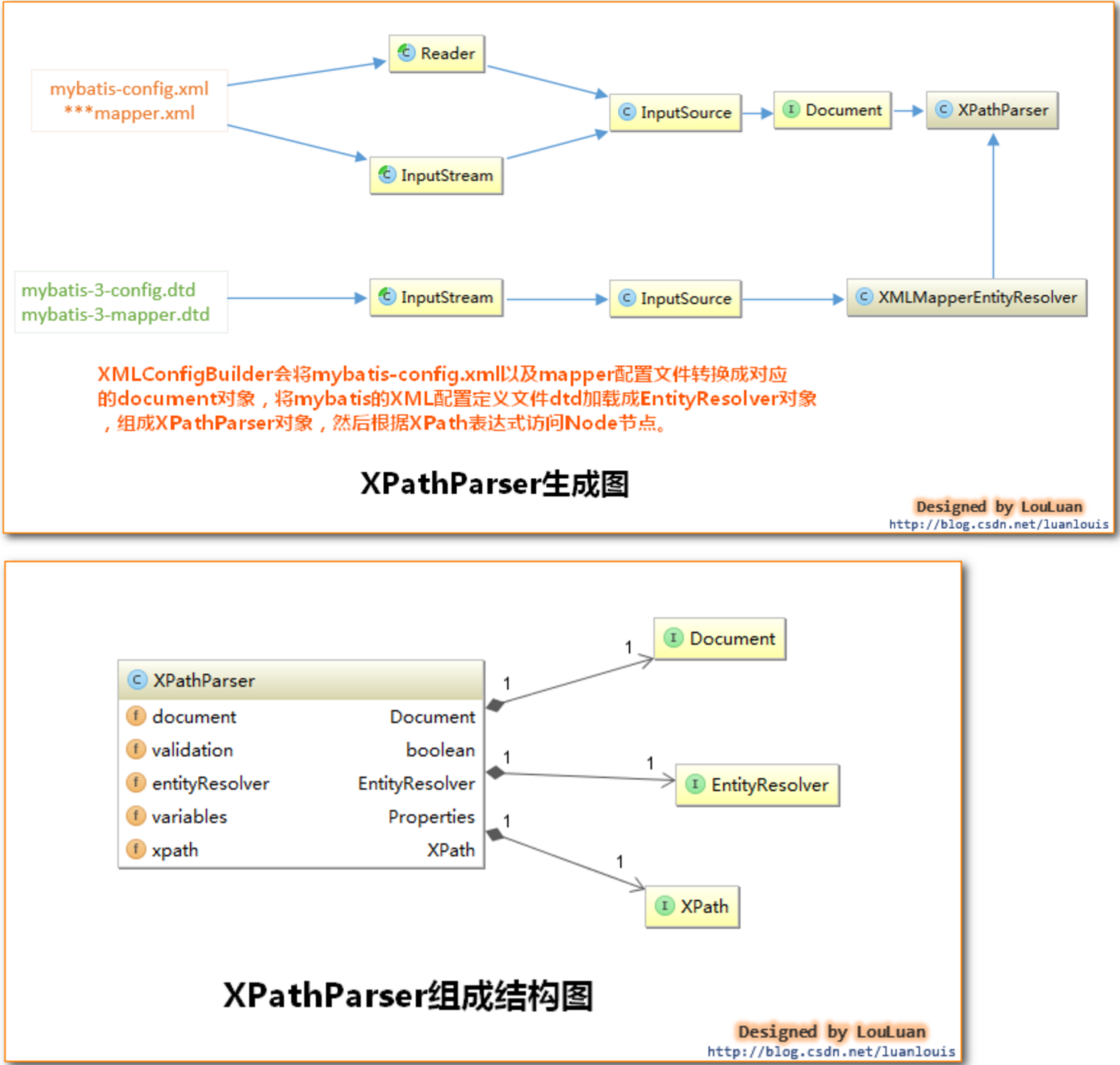
**Configuration ：**该对象是mybatis-config.xml文件中所有mybatis配置信息

**SqlSessionFactory：**SqlSession工厂类，以工厂形式创建SqlSession对象，采用了Factory工厂设计模式

**XmlConfigParser ：**负责将mybatis-config.xml配置文件解析成Configuration对象，共SqlSessonFactoryBuilder使用，创建SqlSessionFactory

1. **创建Configuration对象的过程：**  
   接着上述的 MyBatis初始化基本过程讨论，**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**。如下图所示：



XpathParser组成结构图和生成图

* （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：
* public Configuration parse() {
* if (parsed) {
* throw new BuilderException("Each XMLConfigBuilder can only be used once.");
* }
* parsed = true;
* //源码中没有这一句，只有parseConfiguration(parser.evalNode("/configuration"));
* //为了让读者看得更明晰，源码拆分为以下两句
* XNode configurationNode = parser.evalNode("/configuration");
* parseConfiguration(configurationNode);
* return configuration;
* }
* /\*\*
* \* 解析 "/configuration"节点下的子节点信息，然后将解析的结果设置到Configuration对象中
* \*/
* private void parseConfiguration(XNode root) {
* try {
* //1.首先处理properties 节点
* propertiesElement(root.evalNode("properties")); //issue #117 read properties first
* //2.处理typeAliases
* typeAliasesElement(root.evalNode("typeAliases"));
* //3.处理插件
* pluginElement(root.evalNode("plugins"));
* //4.处理objectFactory
* objectFactoryElement(root.evalNode("objectFactory"));
* //5.objectWrapperFactory
* objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
* //6.settings
* settingsElement(root.evalNode("settings"));
* //7.处理environments
* environmentsElement(root.evalNode("environments")); // read it after objectFactory and objectWrapperFactory issue #631
* //8.database
* databaseIdProviderElement(root.evalNode("databaseIdProvider"));
* //9.typeHandlers
* typeHandlerElement(root.evalNode("typeHandlers"));
* //10.mappers
* mapperElement(root.evalNode("mappers"));
* } catch (Exception e) {
* throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + e, e);
* }
* }

注意：在上述代码中，还有一个非常重要的地方，**就是解析XML配置文件子节点<mappers>的方法mapperElements(root.evalNode("mappers")), 它将解析我们配置的Mapper.xml配置文件，Mapper配置文件可以说是MyBatis的核心**，MyBatis的特性和理念都体现在此Mapper的配置和设计上。

* （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());

}

}

}

}

private boolean isSpecifiedEnvironment(String id)

{

if (environment == null)

{

throw new BuilderException("No environment specified.");

}

else if (id == null)

{

throw new BuilderException("Environment requires an id attribute.");

}

else if (environment.equals(id))

{

return true;

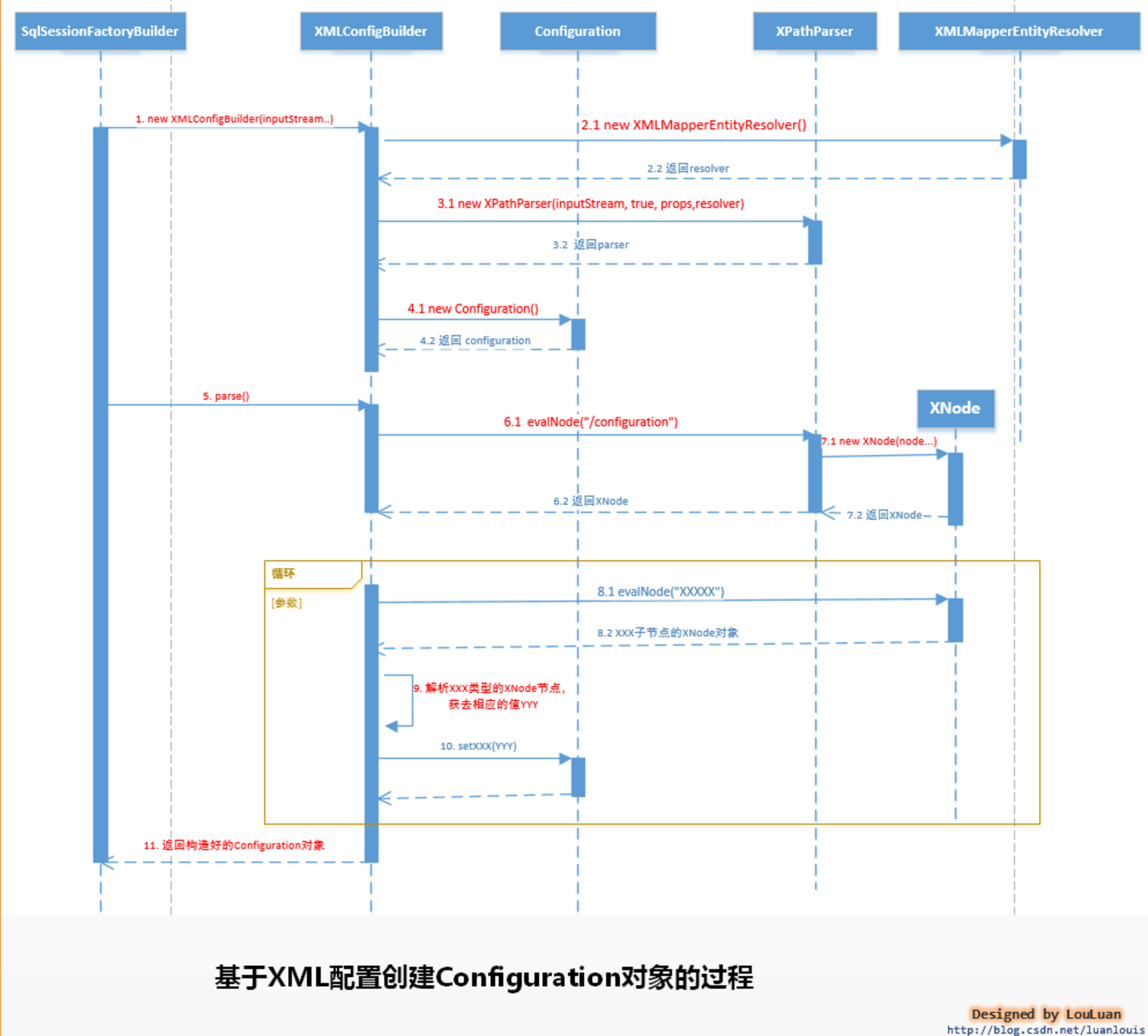
}

return false;

}

* （4）**返回Configuration对象：**

将上述的MyBatis初始化基本过程的序列图细化：



基于XML配置创建Configuration对象的过程

**7.3 基于Java API手动加载XML配置文件创建Configuration对象，并使用SqlSessionFactory对象**

我们可以使用XMLConfigBuilder手动解析XML配置文件来创建Configuration对象，代码如下：

String resource = "mybatis-config.xml";

InputStream inputStream = Resources.getResourceAsStream(resource);

// 手动创建XMLConfigBuilder，并解析创建Configuration对象

XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, null,null);

Configuration configuration=parse();

// 使用Configuration对象创建SqlSessionFactory

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

// 使用MyBatis

SqlSession sqlSession = sqlSessionFactory.openSession();

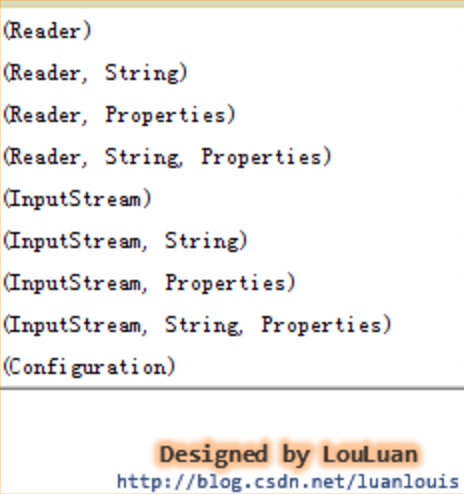
List list = sqlSession.selectList("com.foo.bean.BlogMapper.queryAllBlogInfo");

**7.4 涉及到的设计模式**

初始化的过程涉及到创建各种对象，所以会使用一些创建型的设计模式。**在初始化的过程中，Builder模式运用的比较多**。

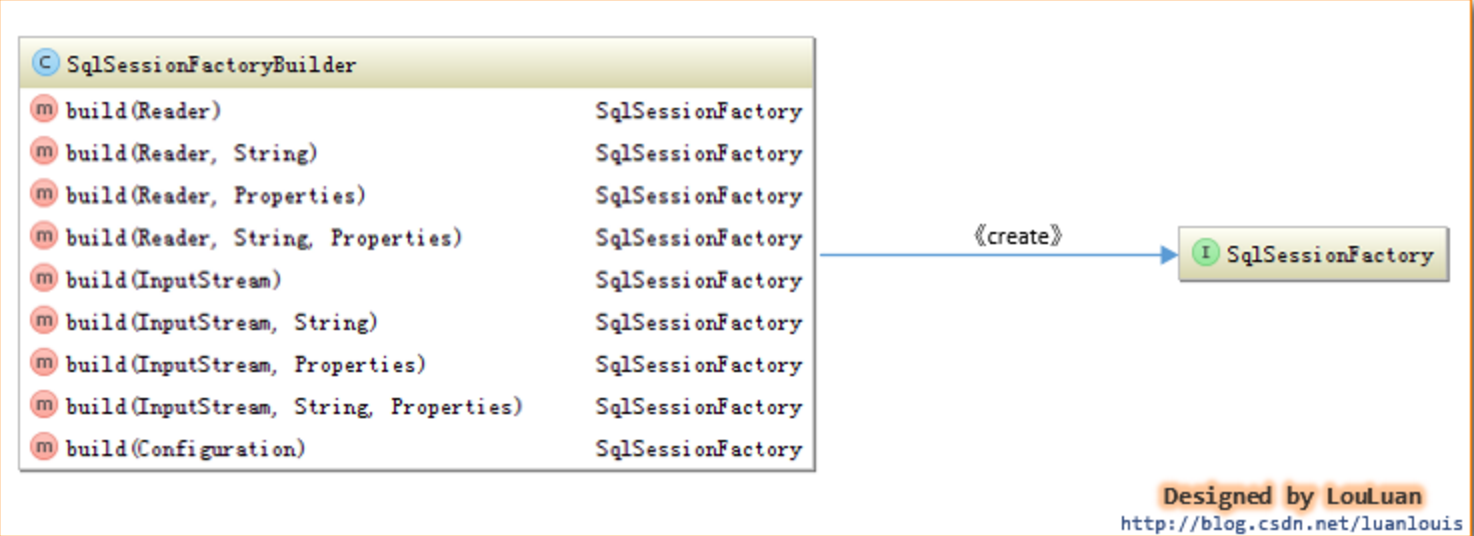
**7.4.1 Builder模式应用1： SqlSessionFactory的创建**

对于创建SqlSessionFactory时，会**根据情况提供不同的参数，其参数组合可以有以下几种**：



根据情况提供不同的参数，创建SqlSessionFactory

由于构造时参数不定，可以为其创建一个构造器Builder，**将SqlSessionFactory的构建过程和表示分开**：



MyBatis将SqlSessionFactoryBuilder和SqlSessionFactory相互独立

**7.4.2 Builder模式应用2： 数据库连接环境Environment对象的创建**

在构建Configuration对象的过程中，XMLConfigParser解析 mybatis XML配置文件节点<environment>节点时，会有以下相应的代码：

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");

//是和默认的环境相同时，解析之

if (isSpecifiedEnvironment(id)) {

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

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

DataSource dataSource = dsFactory.getDataSource();

//使用了Environment内置的构造器Builder，传递id 事务工厂和数据源

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

.transactionFactory(txFactory)

.dataSource(dataSource);

configuration.setEnvironment(environmentBuilder.build());

}

}

}

}

**在Environment内部，定义了静态内部Builder类：**

public final class Environment {

private final String id;

private final TransactionFactory transactionFactory;

private final DataSource dataSource;

public Environment(String id, TransactionFactory transactionFactory, DataSource dataSource) {

if (id == null) {

throw new IllegalArgumentException("Parameter 'id' must not be null");

}

if (transactionFactory == null) {

throw new IllegalArgumentException("Parameter 'transactionFactory' must not be null");

}

this.id = id;

if (dataSource == null) {

throw new IllegalArgumentException("Parameter 'dataSource' must not be null");

}

this.transactionFactory = transactionFactory;

this.dataSource = dataSource;

}

public static class Builder {

private String id;

private TransactionFactory transactionFactory;

private DataSource dataSource;

public Builder(String id) {

this.id = id;

}

public Builder transactionFactory(TransactionFactory transactionFactory) {

this.transactionFactory = transactionFactory;

return this;

}

public Builder dataSource(DataSource dataSource) {

this.dataSource = dataSource;

return this;

}

public String id() {

return this.id;

}

public Environment build() {

return new Environment(this.id, this.transactionFactory, this.dataSource);

}

}

public String getId() {

return this.id;

}

public TransactionFactory getTransactionFactory() {

return this.transactionFactory;

}

public DataSource getDataSource() {

return this.dataSource;

}

}

**8 SQL优化**

**8.1 where子句优化**

下面就某些SQL语句的where子句编写中需要注意的问题作详细介绍。在这些where子句中，即使某些列存在索引，但是由于编写了劣质的SQL，系统在运行该SQL语句时也不能使用该索引，而同样使用全表扫描，这就造成了响应速度的极大降低。

**1. IS NULL 与 IS NOT NULL**

　　不能用null作索引，任何包含null值的列都将不会被包含在索引中。即使索引有多列这样的情况下，只要这些列中有一列含有null，该列就会从索引中排除。也就是说如果某列存在空值，即使对该列建索引也不会提高性能。

　　任何在where子句中使用is null或is not null的语句优化器是不允许使用索引的。

**2. 联接列**

　　对于有联接的列，即使最后的联接值为一个静态值，优化器是不会使用索引的。我们一起来看一个例子，假定有一个职工表(employee)，对于 一个职工的姓和名分成两列存放(FIRST\_NAME和LAST\_NAME)，现在要查询一个叫比尔.克林顿(Bill Cliton)的职工。

　　下面是一个采用联接查询的SQL语句，

    select \* from employss where first\_name||”||last\_name =’Beill Cliton';

    上面这条语句完全可以查询出是否有Bill Cliton这个员工，但是这里需要注意，系统优化器对基于last\_name创建的索引没有使用。

　　当采用下面这种SQL语句的编写，Oracle系统就可以采用基于last\_name创建的索引。

    \*\*\* where first\_name =’Beill’ and last\_name =’Cliton';

**3. 带通配符(%)的like语句**

　　同样以上面的例子来看这种情况。目前的需求是这样的，要求在职工表中查询名字中包含cliton的人。可以采用如下的查询SQL语句:

    select \* from employee where last\_name like ‘%cliton%';

    这里由于通配符(%)在搜寻词首出现，所以Oracle系统不使用last\_name的索引。在很多情况下可能无法避免这种情况，但是一定要心中有底，通 配符如此使用会降低查询速度。然而当通配符出现在字符串其他位置时，优化器就能利用索引。在下面的查询中索引得到了使用:

    select \* from employee where last\_name like ‘c%';

**4. Order by语句**

　　ORDER BY语句决定了Oracle如何将返回的查询结果排序。Order by语句对要排序的列没有什么特别的限制，也可以将函数加入列中(象联接或者附加等)。任何在Order by语句的非索引项或者有计算表达式都将降低查询速度。

　　仔细检查order by语句以找出非索引项或者表达式，它们会降低性能。解决这个问题的办法就是重写order by语句以使用索引，也可以为所使用的列建立另外一个索引，同时应绝对避免在order by子句中使用表达式。

**5. NOT**

　　我们在查询时经常在where子句使用一些逻辑表达式，如大于、小于、等于以及不等于等等，也可以使用and(与)、or(或)以及not(非)。NOT可用来对任何逻辑运算符号取反。下面是一个NOT子句的例子:

… where not (status =’VALID’)

如果要使用NOT，则应在取反的短语前面加上括号，并在短语前面加上NOT运算符。NOT运算符包含在另外一个逻辑运算符中，这就是不等于(<>)运算符。换句话说，即使不在查询where子句中显式地加入NOT词，NOT仍在运算符中，见下例:

… where status <>’INVALID';

对这个查询，可以改写为不使用NOT:

select \* from employee where salary<3000 or salary>3000;

虽然这两种查询的结果一样，但是第二种查询方案会比第一种查询方案更快些。第二种查询允许Oracle对salary列使用索引，而第一种查询则不能使用索引。

**8.2 SQL性能优化总结**

（1）选择最有效率的表名顺序(只在基于规则的优化器中有效)：   
ORACLE 的解析器按照从右到左的顺序处理FROM子句中的表名，FROM子句中写在最后的表(基础表 driving table)将被最先处理，在FROM子句中包含多个表的情况下,你必须选择记录条数最少的表作为基础表。如果有3个以上的表连接查询, 那就需要选择交叉表(intersection table)作为基础表, 交叉表是指那个被其他表所引用的表.   
（2）WHERE子句中的连接顺序．：   
ORACLE采用自下而上的顺序解析WHERE子句,根据这个原理,表之间的连接必须写在其他WHERE条件之前, 那些可以过滤掉最大数量记录的条件必须写在WHERE子句的末尾.   
（3）SELECT子句中避免使用 ‘ \* ‘：   
ORACLE在解析的过程中, 会将’\*’ 依次转换成所有的列名, 这个工作是通过查询数据字典完成的, 这意味着将耗费更多的时间   
（4）减少访问数据库的次数：   
ORACLE在内部执行了许多工作: 解析SQL语句, 估算索引的利用率, 绑定变量 , 读数据块等；   
（5）在SQL\*Plus , SQL\*Forms和Pro\*C中重新设置ARRAYSIZE参数, 可以增加每次数据库访问的检索数据量 ,建议值为200   
（6）使用DECODE函数来减少处理时间：   
使用DECODE函数可以避免重复扫描相同记录或重复连接相同的表.   
（7）整合简单,无关联的数据库访问：   
如果你有几个简单的数据库查询语句,你可以把它们整合到一个查询中(即使它们之间没有关系)   
（8）删除重复记录：   
最高效的删除重复记录方法 ( 因为使用了ROWID)例子：   
DELETE  FROM  EMP E  WHERE  E.ROWID > (SELECT MIN(X.ROWID)   
FROM  EMP X  WHERE  X.EMP\_NO = E.EMP\_NO);   
（9）用TRUNCATE替代DELETE：   
当删除表中的记录时,在通常情况下, 回滚段(rollback segments ) 用来存放可以被恢复的信息. 如果你没有COMMIT事务,ORACLE会将数据恢复到删除之前的状态(准确地说是恢复到执行删除命令之前的状况) 而当运用TRUNCATE时, 回滚段不再存放任何可被恢复的信息.当命令运行后,数据不能被恢复.因此很少的资源被调用,执行时间也会很短. (译者按: TRUNCATE只在删除全表适用,TRUNCATE是DDL不是DML)   
（10）尽量多使用COMMIT：   
只要有可能,在程序中尽量多使用COMMIT, 这样程序的性能得到提高,需求也会因为COMMIT所释放的资源而减少:   
COMMIT所释放的资源:   
a. 回滚段上用于恢复数据的信息.   
b. 被程序语句获得的锁   
c. redo log buffer 中的空间   
d. ORACLE为管理上述3种资源中的内部花费   
（11）用Where子句替换HAVING子句：   
避免使用HAVING子句, HAVING 只会在检索出所有记录之后才对结果集进行过滤. 这个处理需要排序,总计等操作. 如果能通过WHERE子句限制记录的数目,那就能减少这方面的开销. (非oracle中)on、where、having这三个都可以加条件的子句中，on是最先执行，where次之，having最后，因为on是先把不 符合条件的记录过滤后才进行统计，它就可以减少中间运算要处理的数据，按理说应该速度是最快的，where也应该比having快点的，因为它过滤数据后 才进行sum，在两个表联接时才用on的，所以在一个表的时候，就剩下where跟having比较了。在这单表查询统计的情况下，如果要过滤的条件没有涉及到要计算字段，那它们的结果是一样的，只是where可以使用rushmore技术，而having就不能，在速度上后者要慢如果要涉及到计算的字 段，就表示在没计算之前，这个字段的值是不确定的，根据上篇写的工作流程，where的作用时间是在计算之前就完成的，而having就是在计算后才起作 用的，所以在这种情况下，两者的结果会不同。在多表联接查询时，on比where更早起作用。系统首先根据各个表之间的联接条件，把多个表合成一个临时表 后，再由where进行过滤，然后再计算，计算完后再由having进行过滤。由此可见，要想过滤条件起到正确的作用，首先要明白这个条件应该在什么时候起作用，然后再决定放在那里   
（12）减少对表的查询：   
在含有子查询的SQL语句中,要特别注意减少对表的查询.例子：   
    SELECT  TAB\_NAME FROM TABLES WHERE (TAB\_NAME,DB\_VER) = ( SELECT   
TAB\_NAME,DB\_VER FROM  TAB\_COLUMNS  WHERE  VERSION = 604)   
（13）通过内部函数提高SQL效率.：   
复杂的SQL往往牺牲了执行效率. 能够掌握上面的运用函数解决问题的方法在实际工作中是非常有意义的   
（14）使用表的别名(Alias)：   
当在SQL语句中连接多个表时, 请使用表的别名并把别名前缀于每个Column上.这样一来,就可以减少解析的时间并减少那些由Column歧义引起的语法错误.   
（15）用EXISTS替代IN、用NOT EXISTS替代NOT IN：   
在许多基于基础表的查询中,为了满足一个条件,往往需要对另一个表进行联接.在这种情况下, 使用EXISTS(或NOT EXISTS)通常将提高查询的效率. 在子查询中,NOT IN子句将执行一个内部的排序和合并. 无论在哪种情况下,NOT IN都是最低效的 (因为它对子查询中的表执行了一个全表遍历). 为了避免使用NOT IN ,我们可以把它改写成外连接(Outer Joins)或NOT EXISTS.   
例子：   
（高效）SELECT \* FROM  EMP (基础表)  WHERE  EMPNO > 0  AND  EXISTS (SELECT ‘X’  FROM DEPT  WHERE  DEPT.DEPTNO = EMP.DEPTNO  AND  LOC = ‘MELB’)   
(低效)SELECT  \* FROM  EMP (基础表)  WHERE  EMPNO > 0  AND  DEPTNO IN(SELECT DEPTNO  FROM  DEPT  WHERE  LOC = ‘MELB’)   
（16）识别’低效执行’的SQL语句：   
虽然目前各种关于SQL优化的图形化工具层出不穷,但是写出自己的SQL工具来解决问题始终是一个最好的方法：   
SELECT  EXECUTIONS , DISK\_READS, BUFFER\_GETS,   
ROUND((BUFFER\_GETS-DISK\_READS)/BUFFER\_GETS,2) Hit\_radio,   
ROUND(DISK\_READS/EXECUTIONS,2) Reads\_per\_run,   
SQL\_TEXT   
FROM  V$SQLAREA   
WHERE  EXECUTIONS>0   
AND  BUFFER\_GETS > 0   
AND  (BUFFER\_GETS-DISK\_READS)/BUFFER\_GETS < 0.8   
ORDER BY  4 DESC;  
（17）用索引提高效率：   
索引是表的一个概念部分,用来提高检索数据的效率，ORACLE使用了一个复杂的自平衡B-tree结构. 通常,通过索引查询数据比全表扫描要快. 当ORACLE找出执行查询和Update语句的最佳路径时, ORACLE优化器将使用索引. 同样在联结多个表时使用索引也可以提高效率. 另一个使用索引的好处是,它提供了主键(primary key)的唯一性验证.。那些LONG或LONG RAW数据类型, 你可以索引几乎所有的列. 通常, 在大型表中使用索引特别有效. 当然,你也会发现, 在扫描小表时,使用索引同样能提高效率. 虽然使用索引能得到查询效率的提高,但是我们也必须注意到它的代价. 索引需要空间来存储,也需要定期维护, 每当有记录在表中增减或索引列被修改时, 索引本身也会被修改. 这意味着每条记录的INSERT , DELETE , UPDATE将为此多付出4 , 5 次的磁盘I/O . 因为索引需要额外的存储空间和处理,那些不必要的索引反而会使查询反应时间变慢.。定期的重构索引是有必要的.：   
ALTER  INDEX <INDEXNAME> REBUILD <TABLESPACENAME>   
(18）用EXISTS替换DISTINCT：   
当提交一个包含一对多表信息(比如部门表和雇员表)的查询时,避免在SELECT子句中使用DISTINCT. 一般可以考虑用EXIST替换, EXISTS 使查询更为迅速,因为RDBMS核心模块将在子查询的条件一旦满足后,立刻返回结果. 例子：   
      (低效):   
SELECT  DISTINCT  DEPT\_NO,DEPT\_NAME  FROM  DEPT D , EMP E   
WHERE  D.DEPT\_NO = E.DEPT\_NO   
(高效):   
SELECT  DEPT\_NO,DEPT\_NAME  FROM  DEPT D  WHERE  EXISTS ( SELECT ‘X’   
FROM  EMP E  WHERE E.DEPT\_NO = D.DEPT\_NO);   
（19）sql语句用大写的；因为oracle总是先解析sql语句，把小写的字母转换成大写的再执行   
（20）在java代码中尽量少用连接符“＋”连接字符串！   
（21）避免在索引列上使用NOT 通常，　   
我们要避免在索引列上使用NOT, NOT会产生在和在索引列上使用函数相同的影响. 当ORACLE”遇到”NOT,他就会停止使用索引转而执行全表扫描.   
（22）避免在索引列上使用计算．   
WHERE子句中，如果索引列是函数的一部分．优化器将不使用索引而使用全表扫描．   
举例:   
低效：   
SELECT … FROM  DEPT  WHERE SAL \* 12 > 25000;   
高效:   
SELECT … FROM DEPT WHERE SAL > 25000/12;   
（23）用>=替代>   
高效:   
SELECT \* FROM  EMP  WHERE  DEPTNO >=4   
低效:   
SELECT \* FROM EMP WHERE DEPTNO >3   
两者的区别在于, 前者DBMS将直接跳到第一个DEPT等于4的记录而后者将首先定位到DEPTNO=3的记录并且向前扫描到第一个DEPT大于3的记录.   
（24）用UNION替换OR (适用于索引列)   
通常情况下, 用UNION替换WHERE子句中的OR将会起到较好的效果. 对索引列使用OR将造成全表扫描. 注意, 以上规则只针对多个索引列有效. 如果有column没有被索引, 查询效率可能会因为你没有选择OR而降低. 在下面的例子中, LOC\_ID 和REGION上都建有索引.   
高效:   
SELECT LOC\_ID , LOC\_DESC , REGION   
FROM LOCATION   
WHERE LOC\_ID = 10   
UNION   
SELECT LOC\_ID , LOC\_DESC , REGION   
FROM LOCATION   
WHERE REGION = “MELBOURNE”   
低效:   
SELECT LOC\_ID , LOC\_DESC , REGION   
FROM LOCATION   
WHERE LOC\_ID = 10 OR REGION = “MELBOURNE”   
如果你坚持要用OR, 那就需要返回记录最少的索引列写在最前面.   
（25）用IN来替换OR    
这是一条简单易记的规则，但是实际的执行效果还须检验，在ORACLE8i下，两者的执行路径似乎是相同的．　   
低效:   
SELECT…. FROM LOCATION WHERE LOC\_ID = 10 OR LOC\_ID = 20 OR LOC\_ID = 30   
高效   
SELECT… FROM LOCATION WHERE LOC\_IN  IN (10,20,30);   
（26）避免在索引列上使用IS NULL和IS NOT NULL   
避免在索引中使用任何可以为空的列，ORACLE将无法使用该索引．对于单列索引，如果列包含空值，索引中将不存在此记录. 对于复合索引，如果每个列都为空，索引中同样不存在此记录.　如果至少有一个列不为空，则记录存在于索引中．举例: 如果唯一性索引建立在表的A列和B列上, 并且表中存在一条记录的A,B值为(123,null) , ORACLE将不接受下一条具有相同A,B值（123,null）的记录(插入). 然而如果所有的索引列都为空，ORACLE将认为整个键值为空而空不等于空. 因此你可以插入1000 条具有相同键值的记录,当然它们都是空! 因为空值不存在于索引列中,所以WHERE子句中对索引列进行空值比较将使ORACLE停用该索引.   
低效: (索引失效)   
SELECT … FROM  DEPARTMENT  WHERE  DEPT\_CODE IS NOT NULL;   
高效: (索引有效)   
SELECT … FROM  DEPARTMENT  WHERE  DEPT\_CODE >=0;   
（27）总是使用索引的第一个列：   
如果索引是建立在多个列上, 只有在它的第一个列(leading column)被where子句引用时,优化器才会选择使用该索引. 这也是一条简单而重要的规则，当仅引用索引的第二个列时,优化器使用了全表扫描而忽略了索引   
(28）用UNION-ALL 替换UNION ( 如果有可能的话)：   
当SQL 语句需要UNION两个查询结果集合时,这两个结果集合会以UNION-ALL的方式被合并, 然后在输出最终结果前进行排序. 如果用UNION ALL替代UNION, 这样排序就不是必要了. 效率就会因此得到提高. 需要注意的是，UNION ALL 将重复输出两个结果集合中相同记录. 因此各位还是要从业务需求分析使用UNION ALL的可行性. UNION 将对结果集合排序,这个操作会使用到SORT\_AREA\_SIZE这块内存. 对于这块内存的优化也是相当重要的. 下面的SQL可以用来查询排序的消耗量   
低效：   
SELECT  ACCT\_NUM, BALANCE\_AMT   
FROM  DEBIT\_TRANSACTIONS   
WHERE TRAN\_DATE = ’31-DEC-95′   
UNION   
SELECT ACCT\_NUM, BALANCE\_AMT   
FROM DEBIT\_TRANSACTIONS   
WHERE TRAN\_DATE = ’31-DEC-95′   
高效:   
SELECT ACCT\_NUM, BALANCE\_AMT   
FROM DEBIT\_TRANSACTIONS   
WHERE TRAN\_DATE = ’31-DEC-95′   
UNION ALL   
SELECT ACCT\_NUM, BALANCE\_AMT   
FROM DEBIT\_TRANSACTIONS   
WHERE TRAN\_DATE = ’31-DEC-95′   
（29）用WHERE替代ORDER BY：   
ORDER BY 子句只在两种严格的条件下使用索引.   
ORDER BY中所有的列必须包含在相同的索引中并保持在索引中的排列顺序.   
ORDER BY中所有的列必须定义为非空.   
WHERE子句使用的索引和ORDER BY子句中所使用的索引不能并列.   
例如:   
表DEPT包含以下列:   
DEPT\_CODE PK NOT NULL   
DEPT\_DESC NOT NULL   
DEPT\_TYPE NULL   
低效: (索引不被使用)   
SELECT DEPT\_CODE FROM  DEPT  ORDER BY  DEPT\_TYPE   
高效: (使用索引)   
SELECT DEPT\_CODE  FROM  DEPT  WHERE  DEPT\_TYPE > 0   
（30）避免改变索引列的类型.:   
当比较不同数据类型的数据时, ORACLE自动对列进行简单的类型转换.   
假设 EMPNO是一个数值类型的索引列.   
SELECT …  FROM EMP  WHERE  EMPNO = ‘123′   
实际上,经过ORACLE类型转换, 语句转化为:   
SELECT …  FROM EMP  WHERE  EMPNO = TO\_NUMBER(‘123′)   
幸运的是,类型转换没有发生在索引列上,索引的用途没有被改变.   
现在,假设EMP\_TYPE是一个字符类型的索引列.   
SELECT …  FROM EMP  WHERE EMP\_TYPE = 123   
这个语句被ORACLE转换为:   
SELECT …  FROM EMP  WHERETO\_NUMBER(EMP\_TYPE)=123   
因为内部发生的类型转换, 这个索引将不会被用到! 为了避免ORACLE对你的SQL进行隐式的类型转换, 最好把类型转换用显式表现出来. 注意当字符和数值比较时, ORACLE会优先转换数值类型到字符类型   
（31）需要当心的WHERE子句:   
某些SELECT 语句中的WHERE子句不使用索引. 这里有一些例子.   
在下面的例子里, (1)‘!=’ 将不使用索引. 记住, 索引只能告诉你什么存在于表中, 而不能告诉你什么不存在于表中. (2) ‘ ¦ ¦’是字符连接函数. 就象其他函数那样, 停用了索引. (3) ‘+’是数学函数. 就象其他数学函数那样, 停用了索引. (4)相同的索引列不能互相比较,这将会启用全表扫描.   
（32）a. 如果检索数据量超过30%的表中记录数.使用索引将没有显著的效率提高.   
b. 在特定情况下, 使用索引也许会比全表扫描慢, 但这是同一个数量级上的区别. 而通常情况下,使用索引比全表扫描要块几倍乃至几千倍!   
（33）避免使用耗费资源的操作:   
带有DISTINCT,UNION,MINUS,INTERSECT,ORDER BY的SQL语句会启动SQL引擎   
执行耗费资源的排序(SORT)功能. DISTINCT需要一次排序操作, 而其他的至少需要执行两次排序. 通常, 带有UNION, MINUS , INTERSECT的SQL语句都可以用其他方式重写. 如果你的数据库的SORT\_AREA\_SIZE调配得好, 使用UNION , MINUS, INTERSECT也是可以考虑的, 毕竟它们的可读性很强   
（34）优化GROUP BY:   
提高GROUP BY 语句的效率, 可以通过将不需要的记录在GROUP BY 之前过滤掉.下面两个查询返回相同结果但第二个明显就快了许多.   
低效:   
SELECT JOB , AVG(SAL)   
FROM EMP   
GROUP by JOB   
HAVING JOB = ‘PRESIDENT’   
OR JOB = ‘MANAGER’   
高效:   
SELECT JOB , AVG(SAL)   
FROM EMP   
WHERE JOB = ‘PRESIDENT’   
OR JOB = ‘MANAGER’   
GROUP by JOB