

《深入理解mybatis原理》 Mybatis初始化机制详解

原创

亦山

于 2014-07-18 21:54:14 发布

阅读量4.2w

收藏 197

点赞数 78

分类专栏：

MyBatis

数据库

深入理解MyBatis原理

文章标签：


MyBatis原理

MyBatis

设计模式

java

版权

 MyBatis

同时被 3 个专栏收录 ▼

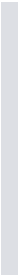
69 订阅 8 篇文章

已订阅

篇文章

已订阅

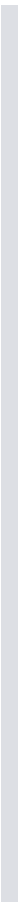
对于任何框架而言，在使用前都要进行一系列的初始化，MyBatis也不例外。本章将通过以下几点详细介绍MyBatis的初始化过程。



- 1.MyBatis的初始化做了什么
- 2. MyBatis基于XML配置文件创建Configuration对象的过程
- 3. 手动加载XML配置文件创建Configuration对象完成初始化，创建并使用SqlSessionFactory对象
- 4. 涉及到的设计模式

一、 MyBatis的初始化做了什么

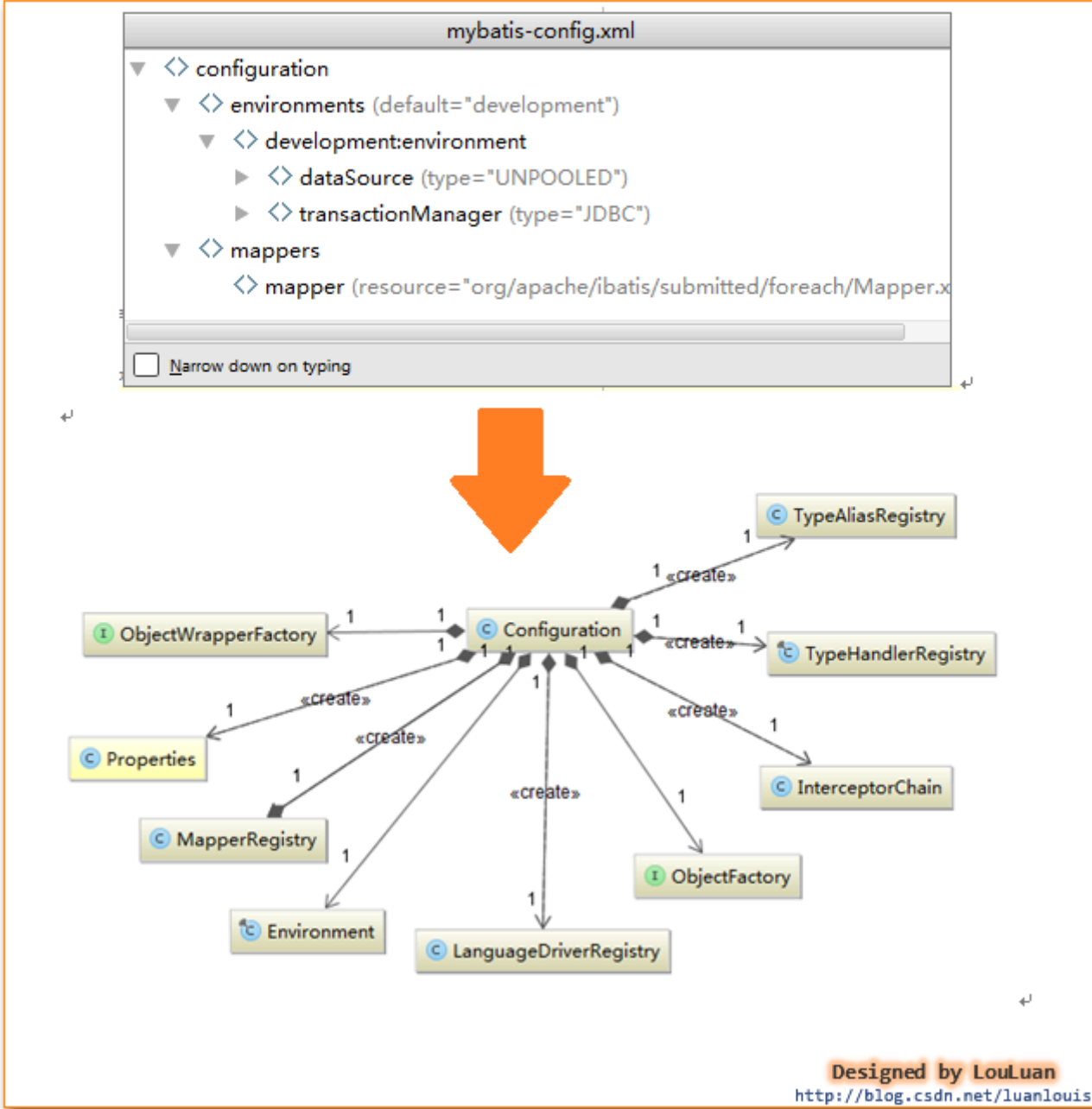
任何框架的初始化，无非是加载自己运行时所需要的配置信息。MyBatis的配置信息，大概包含以下信息，其高层级结构如下：



- × **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
- ×**映射器**

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：



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

(PS: MyBatis具体配置信息有哪些，又分别表示什么意思，不在本文的叙述范围，读者可以参考我的《Java Persistence with MyBatis 3 (中文版)》的第二章 引导MyBatis中有详细的描述)

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，并使用它的。

二、MyBatis基于XML配置文件创建Configuration对象的过程

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");
```

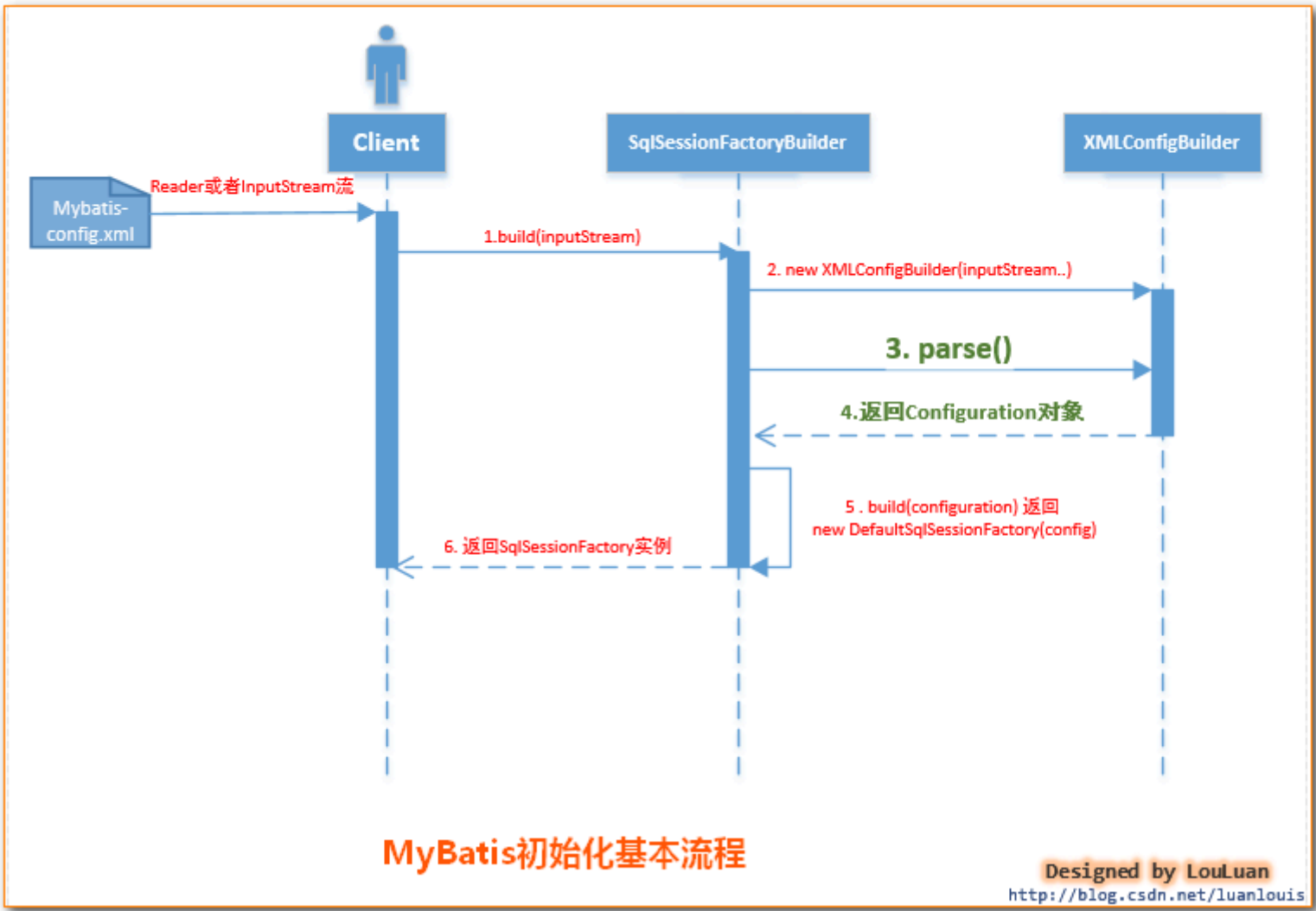
有过MyBatis使用经验的读者会知道，上述语句的作用是执行`com.foo.bean.BlogMapper.queryAllBlogInfo` 定义的SQL语句，返回一个List结果集。总的来说，上述代码经历了 mybatis初始化 --> 创建SqlSession --> 执行SQL语句 返回结果三个过程。

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。

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：



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
// 此方法创建SqlSessionFactory
public SqlSessionFactory build(Configuration config)
{
    return new DefaultSqlSessionFactory(config);
}
}
```

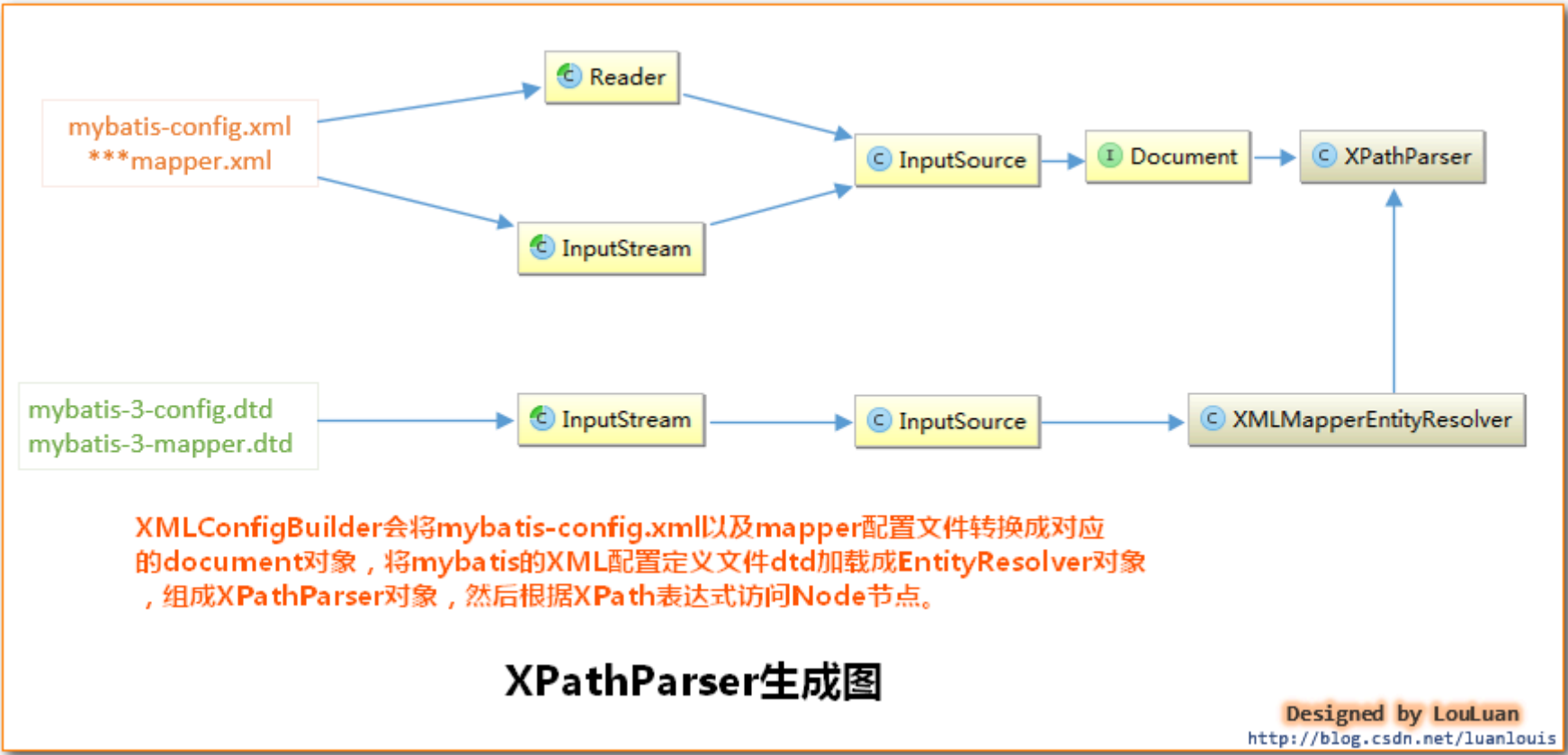
上述的初始化过程中，涉及到了以下几个对象：

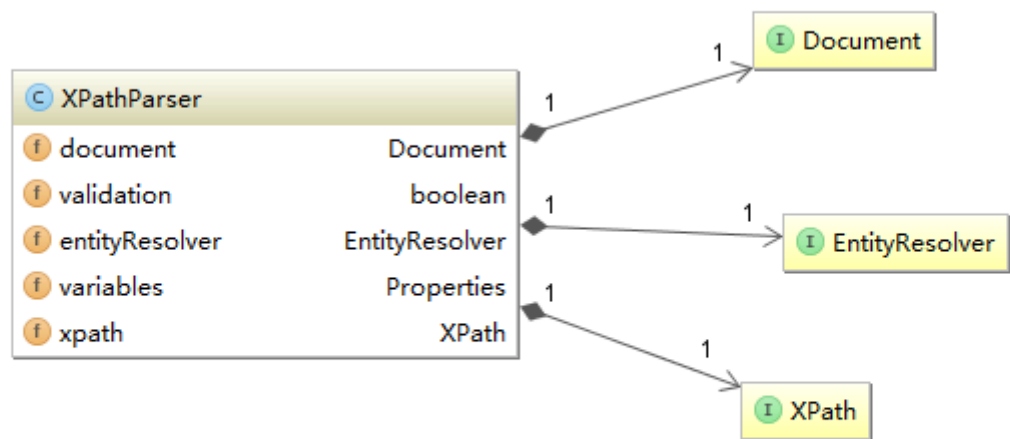
- SqlSessionFactoryBuilder：SqlSessionFactory的构造器，用于创建SqlSessionFactory，采用了Builder设计模式
- Configuration：该对象是mybatis-config.xml文件中所有mybatis配置信息
- SqlSessionFactory：SqlSession工厂类，以工厂形式创建SqlSession对象，采用了Factory工厂设计模式
- XmlConfigParser：负责将mybatis-config.xml配置文件解析成Configuration对象，共SqlSessonFactoryBuilder使用，创建SqlSessionFactory

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

Designed by LouLuan
<http://blog.csdn.net/luanlouis>

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 is:
        //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，我们将在后续的文章中讨论它，敬请期待

~

3. 然后将这些值解析出来设置到Configuration对象中。

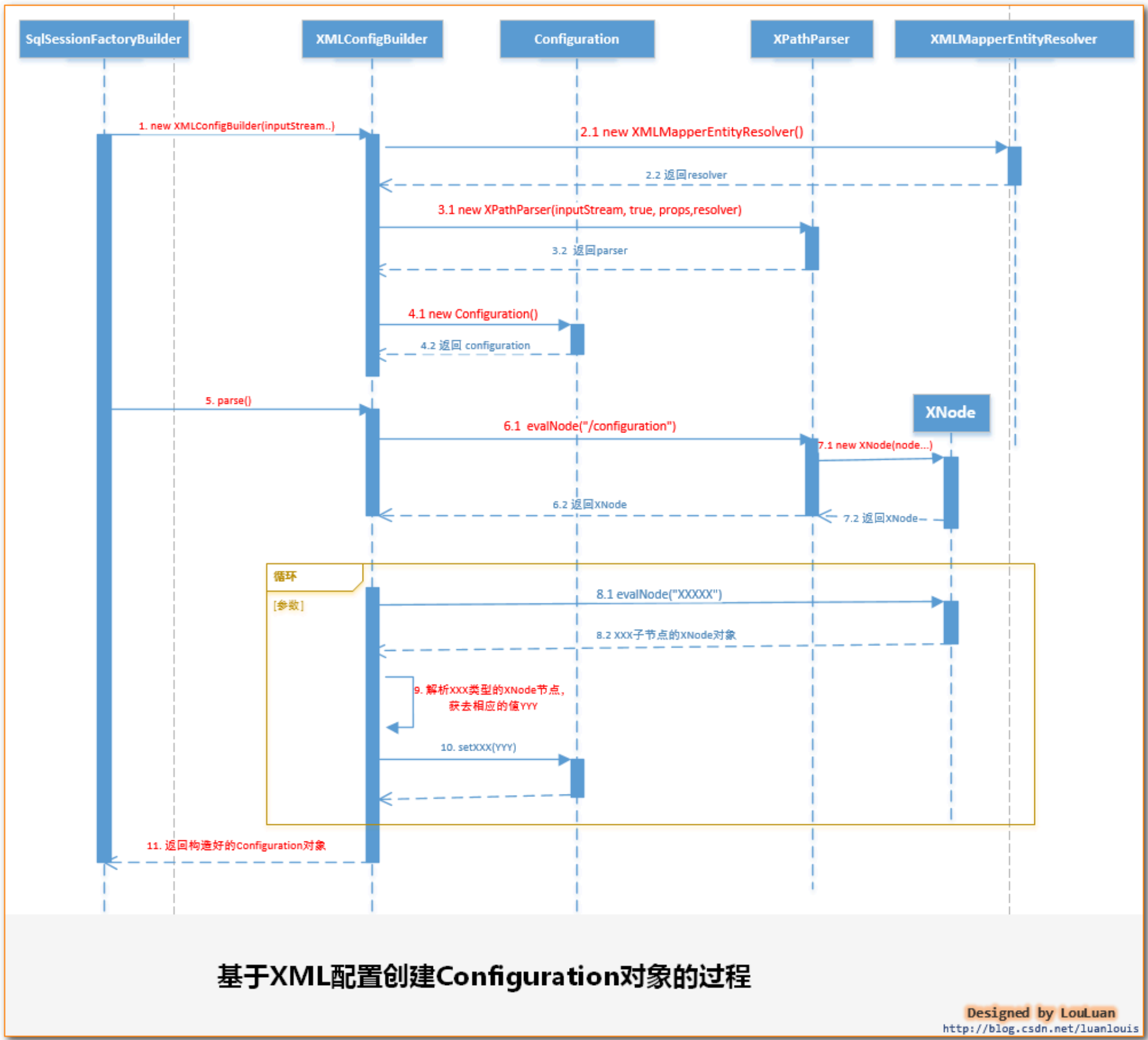
解析子节点的过程这里就不一一介绍了，用户可以参照MyBatis源码仔细揣摩，我们就看上述的environmentsElement(root.evalNode("environments")); 方法是如何将environments的信息解析出来，设置到Configuration对象中的：

```
/*
    解析environments节点，并将结果设置到Configuration对象中
    注意：创建environment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Environment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

我们将上述的 MyBatis 初始化基本过程的序列图细化，



三、手动加载XML配置文件创建Configuration对象完成初始化，创建并使用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");
```

四、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。

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：

(Reader)

(Reader, String)

(Reader, Properties)

(Reader, String, Properties)

(InputStream)

(InputStream, String)

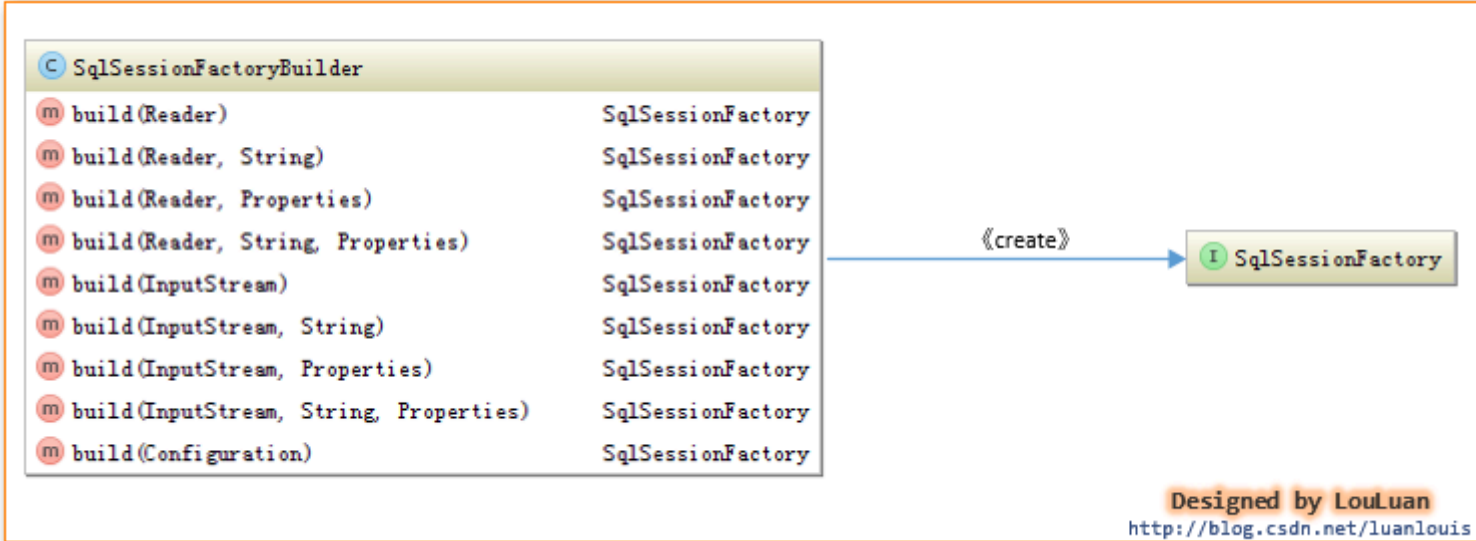
(InputStream, Properties)

(InputStream, String, Properties)

(Configuration)

Designed by LouLuan
<http://blog.csdn.net/luanlouis>

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。

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;
    }
}
```

以上就是本文 《**深入理解mybatis原理**》 **Mybatis初始化机制详解** 的全部内容，希望对大家有所帮助！上述内容如有不妥之处，还请读者指出，共同探讨，共同进步！

《深入理解mybatis原理》 MyBatis事务管理机制

原创

亦山

于 2014-07-20 22:09:57 发布

阅读量6.4w

收藏 212

点赞数 50

版权

分类专栏：

MyBatis

深入理解MyBatis原理

文章标签：


MyBatis原理

database

事务

数据库

MyBatis

 MyBatis

同时被 2 个专栏收录 ▼

69 订阅 8 篇文章

已订阅

篇文章

已订阅

MyBatis作为Java语言的数据库框架，对数据库的事务管理是其非常重要的一个方面。本文将讲述MyBatis的事务管理的实现机制。首先介绍MyBatis的事务Transaction的接口设计以及其不同实现JdbcTransaction 和 ManagedTransaction；接着，从MyBatis的XML配置文件入手，讲解MyBatis事务工厂的创建和维护，进而阐述了MyBatis事务的创建和使用；最后分析JdbcTransaction和ManagedTransaction的实现和二者的不同特点。

以下是本文的组织结构：

目录(?)

[-]

1. 一概述

2. 二事务的配置创建和使用

1. 事务的配置

2. 事务工厂的创建

3. 事务工厂TransactionFactory

4. 事务Transaction的创建

5. JdbcTransaction





6. ManagedTransaction

Designed by LouLuan

<http://blog.csdn.net/luanlouis>

一、概述

对数据库的事务而言，应该具有以下几点：创建（create）、提交（commit）、回滚（rollback）、关闭（close）。对应地，MyBatis将事务抽象成了Transaction接口：其接口定义如下：

I Transaction	
 getConnection()	Connection
 commit()	void
 rollback()	void
 close()	void

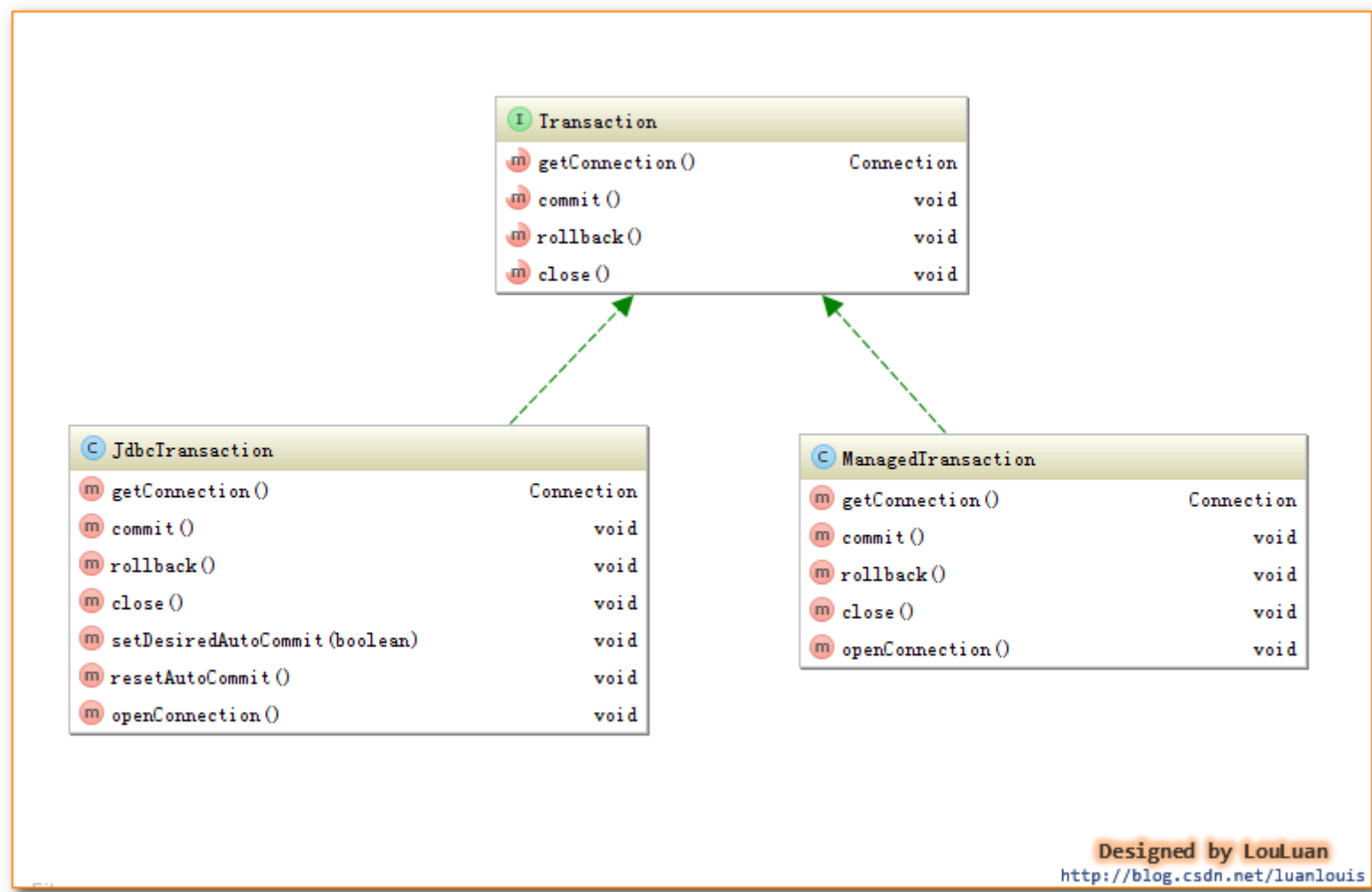
org.apache.ibatis.transaction.Transaction 接口定义了获取Connection 连接、提交、回滚和关闭的功能。

Designed by LouLuan
<http://blog.csdn.net/luanlouis>

MyBatis的事务管理分为两种形式：

- 一、使用JDBC的事务管理机制：即利用java.sql.Connection对象完成对事务的提交（commit()）、回滚（rollback()）、关闭（close()）等
- 二、使用MANAGED的事务管理机制：这种机制MyBatis自身不会去实现事务管理，而是让程序的容器如（JBoss，Weblogic）来实现对事务的管理

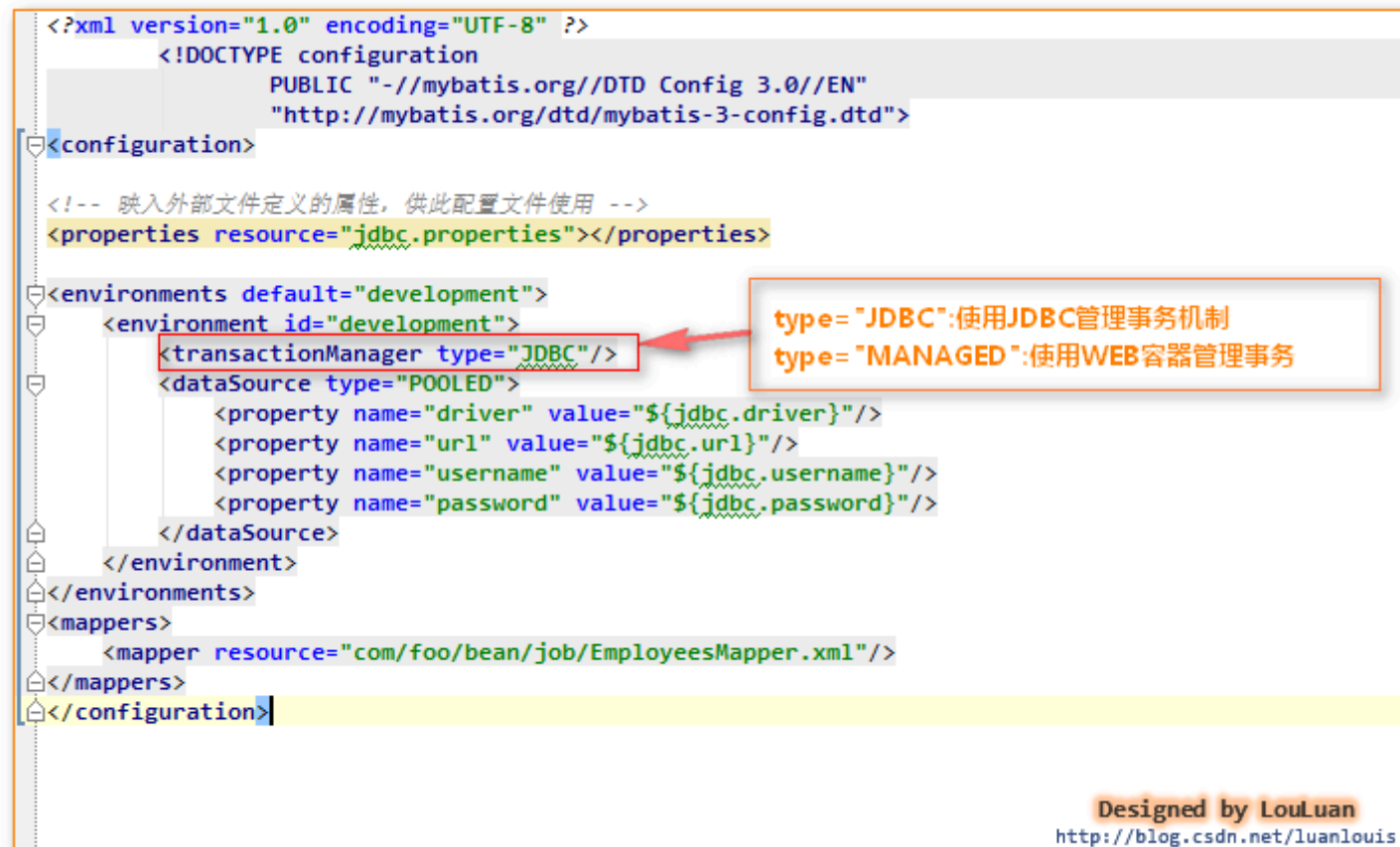
这两者的类图如下所示：



二、事务的配置、创建和使用

1. 事务的配置

我们在使用MyBatis时，一般会在MyBatisXML配置文件中定义类似如下的信息：



<environment>节点定义了连接某个数据库的信息，其子节点<transactionManager>的type会决定我们用什么类型的事务管理机制。

2. 事务工厂的创建

MyBatis事务的创建是交给TransactionFactory 事务工厂来创建的，如果我们将<transactionManager>的type 配置为"JDBC",那么，在MyBatis初始化解析<environment>节点时，会根据type="JDBC"创建一个JdbcTransactionFactory工厂，其源码如下：

```
/**
 * 解析<transactionManager>节点，创建对应的TransactionFactory
 * @param context
 * @return
 * @throws Exception
 */
private TransactionFactory transactionManagerElement(XNode context) throws Exception {
    if (context != null) {
        String type = context.getStringAttribute("type");
```

```

Properties props = context.getChildrenAsProperties();
/*
    在Configuration初始化的时候，会通过以下语句，给JDBC和MANAGED对应的工厂类
    typeAliasRegistry.registerAlias("JDBC", JdbcTransactionFactory.class);
    typeAliasRegistry.registerAlias("MANAGED", ManagedTransactionFactory.class);
    下述的resolveClass(type).newInstance()会创建对应的工厂实例
*/
TransactionFactory factory = (TransactionFactory) resolveClass(type).newInstance();
factory.setProperties(props);
return factory;
}
throw new BuilderException("Environment declaration requires a TransactionFactory.");
}

```

如上述代码所示，如果type = "JDBC",则MyBatis会创建一个JdbcTransactionFactory.class 实例；如果type="MANAGED"，则MyBatis会创建一个MangedTransactionFactory.class 实例。

MyBatis对<transactionManager>节点的解析会生成 TransactionFactory实例；而对<dataSource>解析会生成datasouce实例(关于dataSource的解析和原理，读者可以参照我的另一篇博文：《深入理解mybatis原理》 Mybatis数据源与连接池

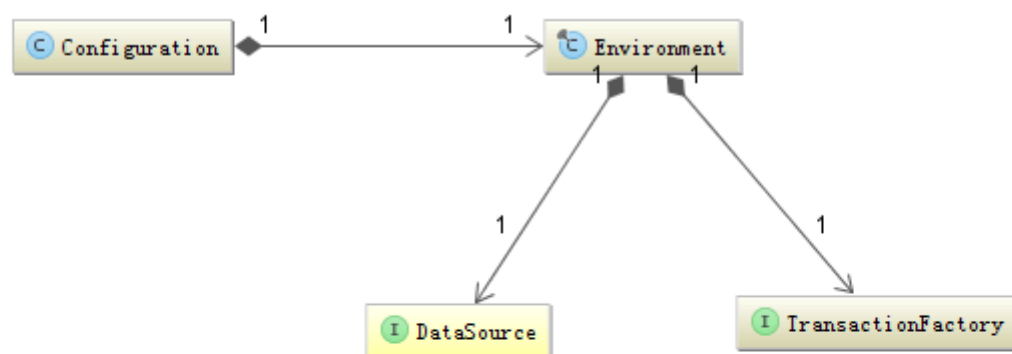
), 作为<environment>节点，会根据TransactionFactory和DataSource实例创建一个Environment对象，代码如下所示：

```

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)) {
                //1.解析<transactionManager>节点，决定创建什么类型的TransactionFactory
                TransactionFactory txFactory = transactionManagerElement(child.evalNode("transactionManager"));
                //2. 创建dataSource
                DataSourceFactory dsFactory = dataSourceElement(child.evalNode("dataSource"));
                DataSource dataSource = dsFactory.getDataSource();
                //3. 使用了Environment内置的构造器Builder，传递id 事务工厂TransactionFactory和数据源DataSource
                Environment.Builder environmentBuilder = new Environment.Builder(id)
                    .transactionFactory(txFactory)
                    .dataSource(dataSource);
                configuration.setEnvironment(environmentBuilder.build());
            }
        }
    }
}

```

Environment表示着一个数据库的连接，生成后的Environment对象会被设置到Configuration实例中，以供后续的使用。



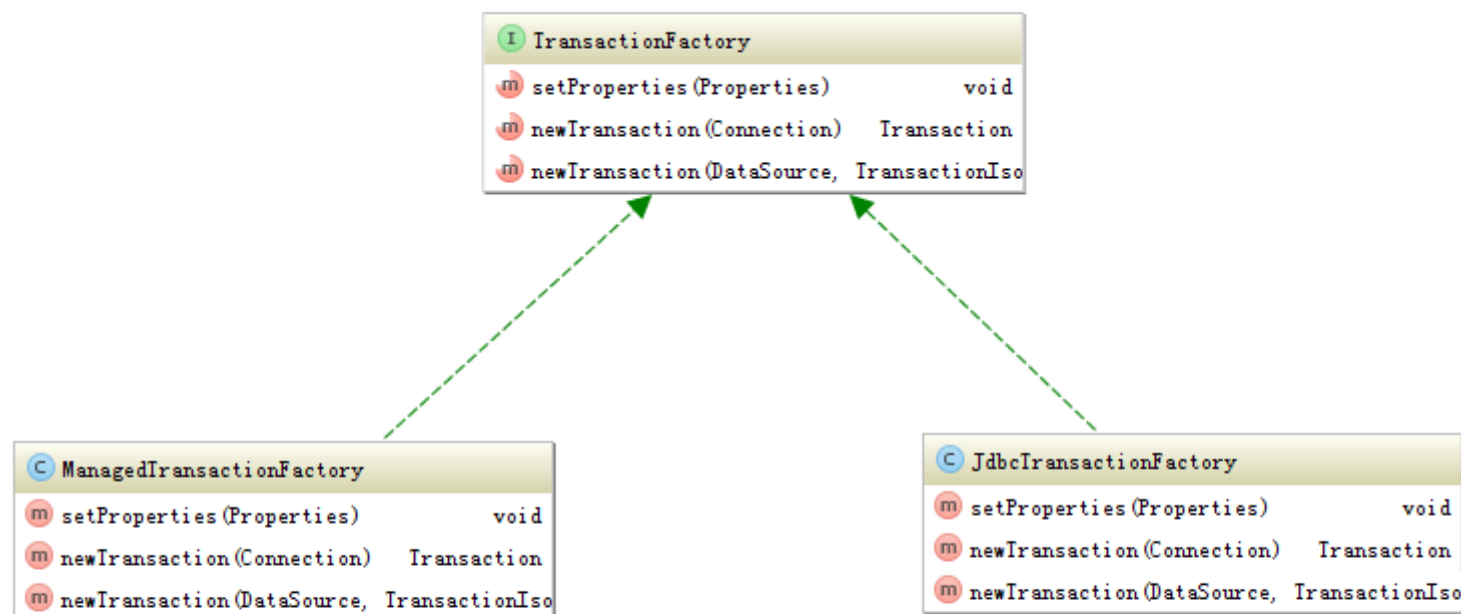
Environment是某个数据库环境连接，它含有**DataSource**和**TransactionFactory**
DataSource 表示数据源，
TransactionFactory表示事务工厂，用来创建事务

Designed by LouLuan
<http://blog.csdn.net/luanlouis>

上述一直在讲事务工厂TransactionFactory来创建的Transaction，现在让我们看一下MyBatis中的TransactionFactory的定义吧。

3. 事务工厂TransactionFactory

事务工厂TransactionFactory定义了创建Transaction的两个方法：一个是通过指定的Connection对象创建Transaction，另外是通过数据源DataSource来创建Transaction。与JDBC 和 MANAGED两种Transaction相对应，TransactionFactory有两个对应的实现的子类：如下所示：



Designed by LouLuan
<http://blog.csdn.net/luanlouis>

4. 事务Transaction的创建

通过事务工厂TransactionFactory很容易获取到Transaction对象实例。我们以JdbcTransaction为例，看一下JdbcTransactionFactory是怎样生成JdbcTransaction的，代码如下：

```
public class JdbcTransactionFactory implements TransactionFactory {

    public void setProperties(Properties props) {
    }

    /**
     * 根据给定的数据库连接Connection创建Transaction
     * @param conn Existing database connection
     * @return
     */
    public Transaction newTransaction(Connection conn) {
        return new JdbcTransaction(conn);
    }
}
```



```

    /**
     * 根据DataSource、隔离级别和是否自动提交创建Transacion
     *
     * @param ds
     * @param level Desired isolation level
     * @param autoCommit Desired autocommit
     * @return
     */
    public Transaction newTransaction(DataSource ds, TransactionIsolationLevel level, boolean autoCommit) {
        return new JdbcTransaction(ds, level, autoCommit);
    }
}

```

如上说是，JdbcTransactionFactory会创建JDBC类型的Transaction，即JdbcTransaction。类似地，ManagedTransactionFactory也会创建ManagedTransaction。下面我们会分别深入JdbcTranaction 和ManagedTransaction，看它们到底是怎样实现事务管理的。

5. JdbcTransaction

JdbcTransaction直接使用JDBC的提交和回滚事务管理机制。它依赖与从dataSource中取得的连接connection 来管理transaction 的作用域，connection对象的获取被延迟到调用getConnection()方法。如果autocommit设置为on，开启状态的话，它会忽略commit和rollback。

直观地讲，就是JdbcTransaction是使用的java.sql.Connection 上的commit和rollback功能，JdbcTransaction只是相当于对java.sql.Connection事务处理进行了一次包装(wrapper)，Transaction的事务管理都是通过java.sql.Connection实现的。JdbcTransaction的代码实现如下：

```

/**
 * @see JdbcTransactionFactory
 */
/**
 * @author Clinton Begin
 */
public class JdbcTransaction implements Transaction {

    private static final Log log = LogFactory.getLog(JdbcTransaction.class);

    //数据库连接
    protected Connection connection;
    //数据源
    protected DataSource dataSource;
    //隔离级别
    protected TransactionIsolationLevel level;
    //是否为自动提交
    protected boolean autoCommit;

    public JdbcTransaction(DataSource ds, TransactionIsolationLevel desiredLevel, boolean desiredAutoCommit) {
        dataSource = ds;
        level = desiredLevel;
        autoCommit = desiredAutoCommit;
    }

    public JdbcTransaction(Connection connection) {
        this.connection = connection;
    }

    public Connection getConnection() throws SQLException {
        if (connection == null) {
            openConnection();
        }
        return connection;
    }

    /**
     * commit()功能 使用connection的commit()
     * @throws SQLException
     */
    public void commit() throws SQLException {
        if (connection != null && !connection.getAutoCommit()) {

```

```

        if (log.isDebugEnabled()) {
            log.debug("Committing JDBC Connection [" + connection + "]");
        }
        connection.commit();
    }
}

/**
 * rollback()功能 使用connection的rollback()
 * @throws SQLException
 */
public void rollback() throws SQLException {
    if (connection != null && !connection.getAutoCommit()) {
        if (log.isDebugEnabled()) {
            log.debug("Rolling back JDBC Connection [" + connection + "]");
        }
        connection.rollback();
    }
}

/**
 * close()功能 使用connection的close()
 * @throws SQLException
 */
public void close() throws SQLException {
    if (connection != null) {
        resetAutoCommit();
        if (log.isDebugEnabled()) {
            log.debug("Closing JDBC Connection [" + connection + "]");
        }
        connection.close();
    }
}

protected void setDesiredAutoCommit(boolean desiredAutoCommit) {
    try {
        if (connection.getAutoCommit() != desiredAutoCommit) {
            if (log.isDebugEnabled()) {
                log.debug("Setting autocommit to " + desiredAutoCommit + " on JDBC Connection [" + connection + "]");
            }
            connection.setAutoCommit(desiredAutoCommit);
        }
    } catch (SQLException e) {
        // Only a very poorly implemented driver would fail here,
        // and there's not much we can do about that.
        throw new TransactionException("Error configuring AutoCommit. "
            + "Your driver may not support getAutoCommit() or setAutoCommit(). "
            + "Requested setting: " + desiredAutoCommit + ". Cause: " + e, e);
    }
}

protected void resetAutoCommit() {
    try {
        if (!connection.getAutoCommit()) {
            // MyBatis does not call commit/rollback on a connection if just selects were performed.
            // Some databases start transactions with select statements
            // and they mandate a commit/rollback before closing the connection.
            // A workaround is setting the autocommit to true before closing the connection.
            // Sybase throws an exception here.
            if (log.isDebugEnabled()) {
                log.debug("Resetting autocommit to true on JDBC Connection [" + connection + "]");
            }
            connection.setAutoCommit(true);
        }
    } catch (SQLException e) {
        log.debug("Error resetting autocommit to true "
            + "before closing the connection. Cause: " + e);
    }
}

```

```

        protected void openConnection() throws SQLException {
    if (log.isDebugEnabled()) {
        log.debug("Opening JDBC Connection");
    }
    connection = dataSource.getConnection();
    if (level != null) {
        connection.setTransactionIsolation(level.getLevel());
    }
    setDesiredAutoCommit(autoCommit);
}
}

```

6. ManagedTransaction

ManagedTransaction让容器来管理事务Transaction的整个生命周期，意思就是说，使用ManagedTransaction的commit和rollback功能不会对事务有任何的影响，它什么都不会做，它将事务管理的权利移交给了容器来实现。看如下Managed的实现代码大家就会一目了然：

```

/**
 *
 * 让容器管理事务transaction的整个生命周期
 * connection的获取延迟到getConnection()方法的调用
 * 忽略所有的commit和rollback操作
 * 默认情况下，可以关闭一个连接connection，也可以配置它不可以关闭一个连接
 * 让容器来管理transaction的整个生命周期
 * @see ManagedTransactionFactory
 */
/**
 * @author Clinton Begin
 */
public class ManagedTransaction implements Transaction {

    private static final Log log = LogFactory.getLog(ManagedTransaction.class);

    private DataSource dataSource;
    private TransactionIsolationLevel level;
    private Connection connection;
    private boolean closeConnection;

    public ManagedTransaction(Connection connection, boolean closeConnection) {
        this.connection = connection;
        this.closeConnection = closeConnection;
    }

    public ManagedTransaction(DataSource ds, TransactionIsolationLevel level, boolean closeConnection) {
        this.dataSource = ds;
        this.level = level;
        this.closeConnection = closeConnection;
    }

    public Connection getConnection() throws SQLException {
        if (this.connection == null) {
            openConnection();
        }
        return this.connection;
    }

    public void commit() throws SQLException {
        // Does nothing
    }

    public void rollback() throws SQLException {
        // Does nothing
    }
}

```

```
|
|
| public void close() throws SQLException {
if (this.closeConnection && this.connection != null) {
    if (log.isDebugEnabled()) {
        log.debug("Closing JDBC Connection [" + this.connection + "]");
    }
    this.connection.close();
}
}

protected void openConnection() throws SQLException {
    if (log.isDebugEnabled()) {
        log.debug("Opening JDBC Connection");
    }
    this.connection = this.dataSource.getConnection();
    if (this.level != null) {
        this.connection.setTransactionIsolation(this.level.getLevel());
    }
}

}
```

注意：如果我们使用MyBatis构建本地程序，即不是WEB程序，若将type设置成"MANAGED"，那么，我们执行的任何update操作，即使我们最后执行了commit操作，数据也不会保留，不会对数据库造成任何影响。因为我们将MyBatis配置成了“MANAGED”，即MyBatis自己不管理事务，而我们又是运行的本地程序，没有事务管理功能，所以对数据库的update操作都是无效的。

以上就是《**深入理解mybatis原理**》 **MyBatis事务管理机制** 的全部内容，如有错误或者不准确的地方，请读者指正，共同进步！

《深入理解mybatis原理》 Mybatis数据源与连接池

原创

亦山

于 2014-07-10 23:32:51 发布

阅读量7.4w

收藏 321

点赞数 126

版权

分类专栏：

MyBatis

MyBatis教程


深入理解MyBatis原理

文章标签：

数据库连接池

MyBatis

MyBatis原理

 MyBatis

同时被 3 个专栏收录 ▼

69 订阅 8 篇文章

已订阅

篇文章

已订阅

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。

以下是本章的组织结构：

- 一、MyBatis数据源DataSource分类
- 二、数据源DataSource的创建过程
- 三、DataSource什么时候创建Connection对象
- 四、不使用连接池的UnpooledDataSource
- 五、为什么要使用连接池？
- 六、使用了连接池的PooledDataSource

一、MyBatis数据源DataSource分类

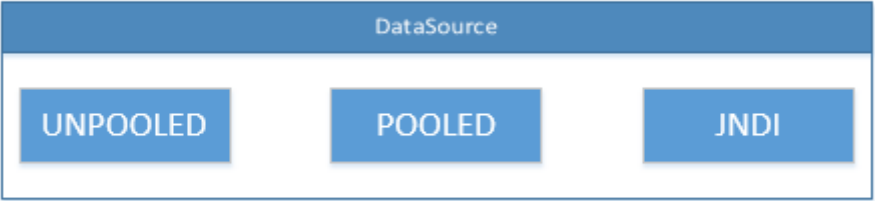
MyBatis数据源实现是在以下四个包中：

- org.apache.ibatis.datasource
- org.apache.ibatis.datasource.jndi
- org.apache.ibatis.datasource.pooled
- org.apache.ibatis.datasource.unpooled

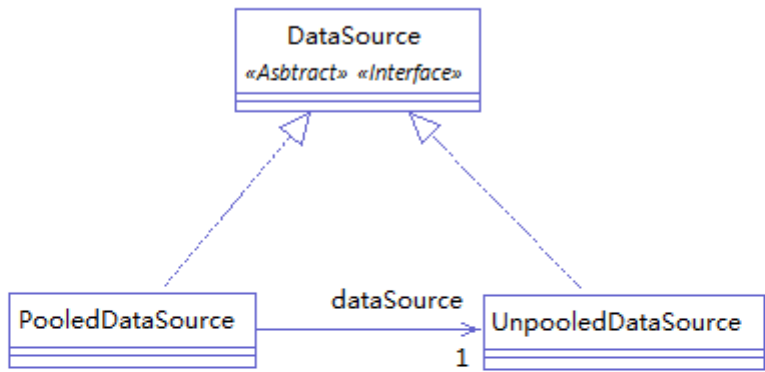
MyBatis把数据源DataSource分为三种：

- UNPOOLED 不使用连接池的数据源
- POOLED 使用连接池的数据源
- JNDI 使用JNDI实现的数据源

即：



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：



PooledDataSource和UnpooledDataSource都实现了java.sql.DataSource接口。并且PooledDataSource持有一个UnpooledDataSource的引用，当PooledDataSource需要创建java.sql.Connection实例对象时，还是通过UnpooledDataSource来创建。PooledDataSource只是提供一种缓存连接池机制。

MyBatis DataSource实现 UML图

<http://blog.csdn.net/luanlouiscn>

对于JNDI类型的数据源DataSource，则是通过JNDI上下文中取值。

二、数据源DataSource的创建过程

MyBatis数据源DataSource对象的创建发生在MyBatis初始化的过程中。下面让我们一步步地了解MyBatis是如何创建数据源DataSource的。

在mybatis的XML配置文件中，使用<dataSource>元素来配置数据源：

```
1 <dataSource type="POOLED">
2   <property name="driver" value="${jdbc.driverClassName}" />
3   <property name="url" value="${jdbc.url}" />
4   <property name="username" value="${jdbc.username}" />
5   <property name="password" value="${jdbc.password}" />
6 </dataSource>
```

1. MyBatis在初始化时，解析此文件，根据<dataSource>的type属性来创建相应类型的的数据源DataSource，即：

- type=" POOLED" ： MyBatis会创建PooledDataSource实例
- type=" UNPOOLED" ： MyBatis会创建UnpooledDataSource实例
- type=" JNDI" ： MyBatis会从JNDI服务上查找DataSource实例，然后返回使用

2. 顺便说一下，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：

定义如下：

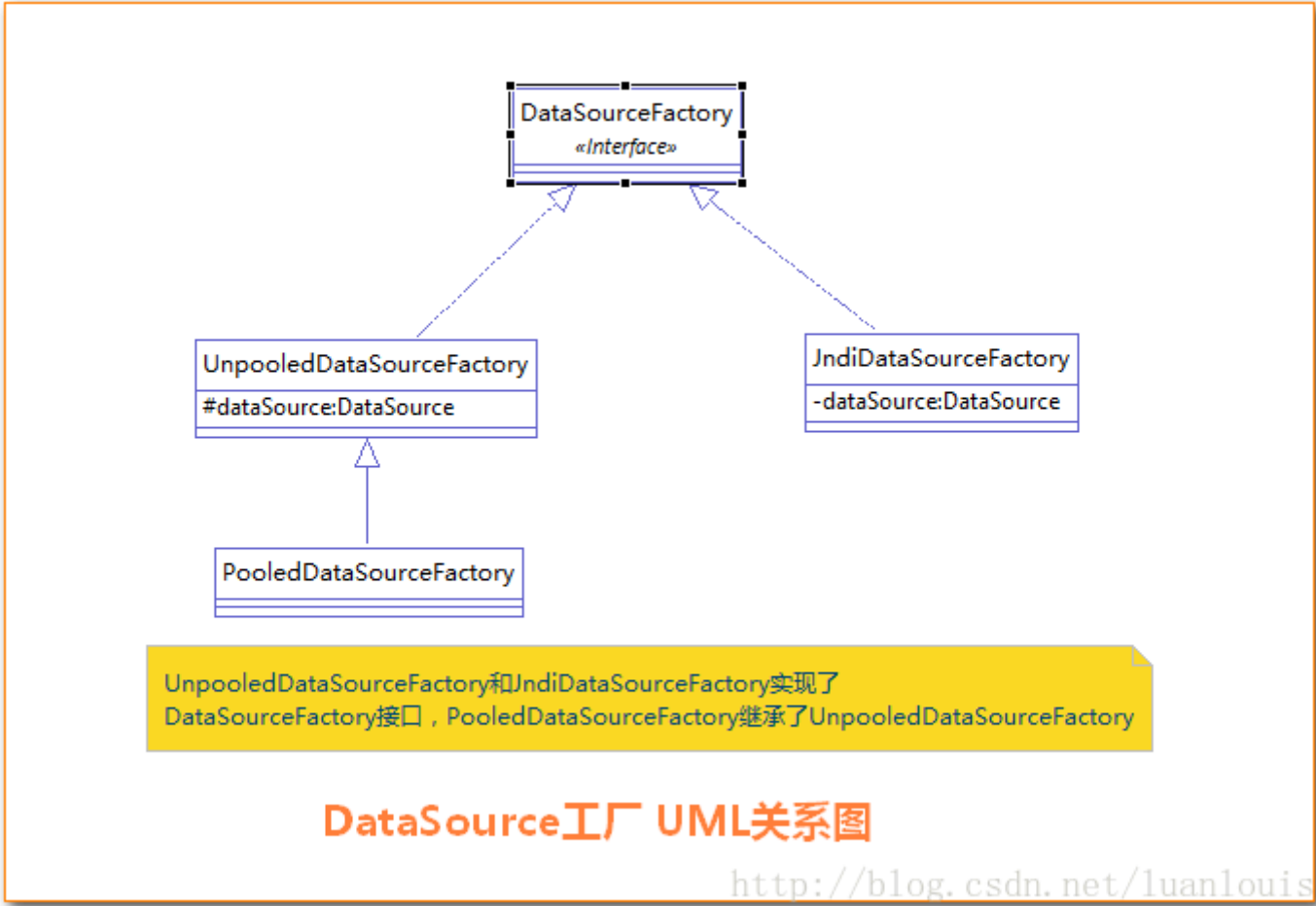
```
public interface DataSourceFactory {

    void setProperties(Properties props);
    //生产DataSource
    DataSource getDataSource();
}
```

上述三种不同类型的type，则有对应的以下dataSource工厂：

- POOLED PooledDataSourceFactory
- UNPOOLED UnpooledDataSourceFactory
- JNDI JndiDataSourceFactory

其类图如下所示：



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。

三、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。

比如，我们有如下方法执行一个简单的SQL语句：

```
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
SqlSession sqlSession = sqlSessionFactory.openSession();
sqlSession.selectList("SELECT * FROM STUDENTS");
```

前4句都不会导致java.sql.Connection对象的创建，只有当第5句sqlSession.selectList("SELECT * FROM STUDENTS")，才会触发MyBatis在底层执行下面这个方法来自创建java.sql.Connection对象：

```
protected void openConnection() throws SQLException {
    if (log.isDebugEnabled()) {
        log.debug("Opening JDBC Connection");
    }
    connection = dataSource.getConnection();
    if (level != null) {
        connection.setTransactionIsolation(level.getLevel());
    }
    setDesiredAutoCommit(autoCommit);
}
```

而对于DataSource的UNPOOLED的类型的实现-UnpooledDataSource是怎样实现getConnection()方法的呢？请看下一节。

四、不使用连接池的UnpooledDataSource

当 <dataSource>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。

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

UnPooledDataSource的getConnection()方法实现如下：

```
/*
UnpooledDataSource的getConnection()实现
*/
public Connection getConnection() throws SQLException
{
    return doGetConnection(username, password);
}

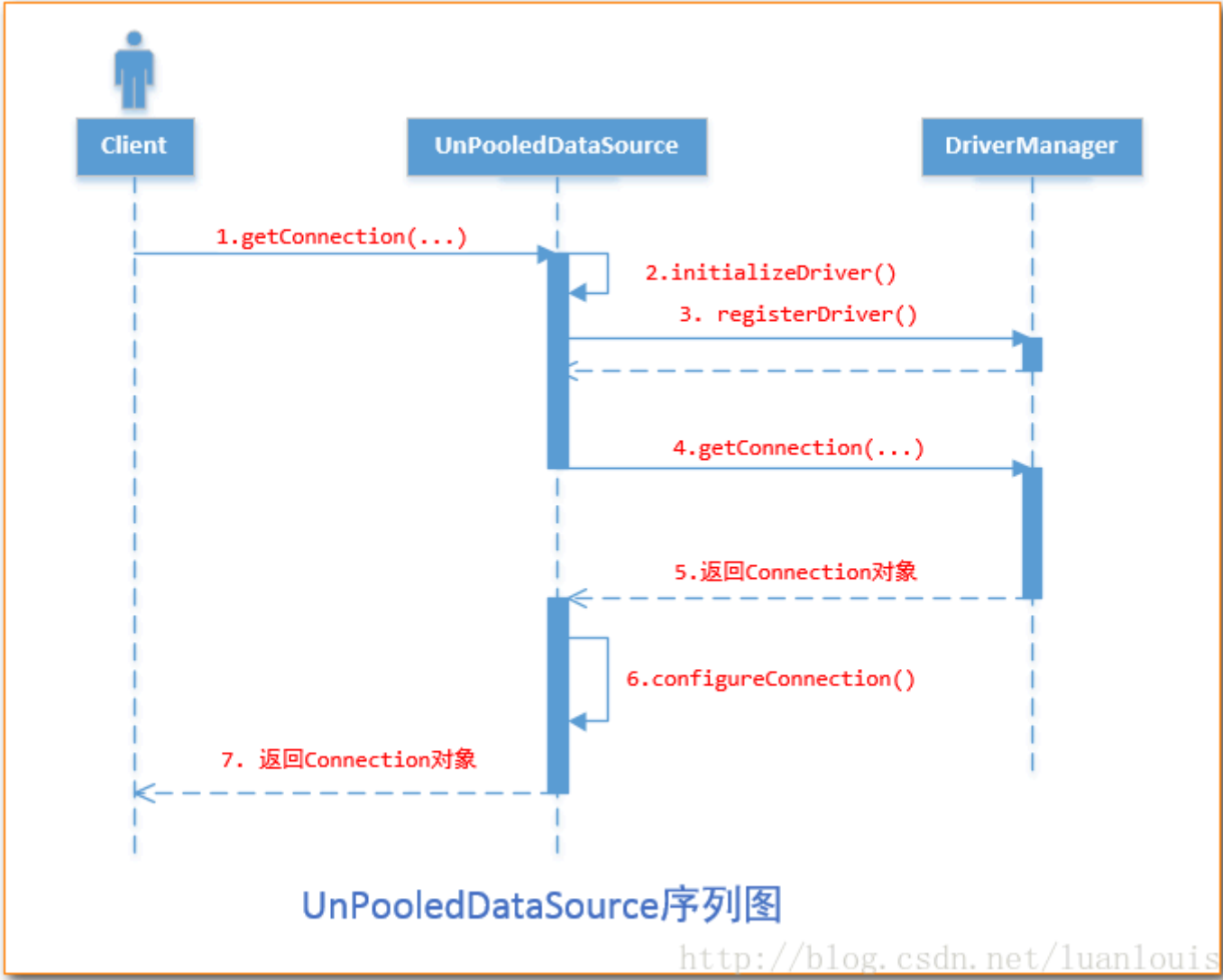
private Connection doGetConnection(String username, String password) throws SQLException
{
    //封装username和password成properties
    Properties props = new Properties();
    if (driverProperties != null)
    {
        props.putAll(driverProperties);
    }
    if (username != null)
    {
        props.setProperty("user", username);
    }
    if (password != null)
    {
        props.setProperty("password", password);
    }
    return doGetConnection(props);
}

/*
 * 获取数据连接
 */
private Connection doGetConnection(Properties properties) throws SQLException
{
    //1.初始化驱动
    initializeDriver();
    //2.从DriverManager中获取连接，获取新的Connection对象
    Connection connection = DriverManager.getConnection(url, properties);
    //3.配置connection属性
    configureConnection(connection);
    return connection;
}
```

如上代码所示，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。**

上述的序列图如下所示：



总结：从上述的代码中可以看到，我们每调用一次getConnection()方法，都会通过DriverManager.getConnection()返回新的java.sql.Connection实例。

五、为什么要使用连接池？

1. 创建一个java.sql.Connection实例对象的代价

首先让我们来看一下创建一个java.sql.Connection对象的资源消耗。我们通过连接Oracle数据库，创建创建Connection对象，来看创建一个Connection对象、执行SQL语句各消耗多长时间。代码如下：

```
public static void main(String[] args) throws Exception
{

    String sql = "select * from hr.employees where employee_id < ? and employee_id >= ?";
    PreparedStatement st = null;
    ResultSet rs = null;

    long beforeTimeOffset = -1L; //创建Connection对象前时间
    long afterTimeOffset = -1L; //创建Connection对象后时间
    long executeTimeOffset = -1L; //创建Connection对象后时间

    Connection con = null;
    Class.forName("oracle.jdbc.driver.OracleDriver");

    beforeTimeOffset = new Date().getTime();
    System.out.println("before:\t" + beforeTimeOffset);

    con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "louluan", "123456");

    afterTimeOffset = new Date().getTime();
    System.out.println("after:\t\t" + afterTimeOffset);
    System.out.println("Create Costs:\t\t" + (afterTimeOffset - beforeTimeOffset) + " ms");

    st = con.prepareStatement(sql);
    //设置参数
    st.setInt(1, 101);
    st.setInt(2, 0);
    //查询，得出结果集
    rs = st.executeQuery();
    executeTimeOffset = new Date().getTime();
    System.out.println("Exec Costs:\t\t" + (executeTimeOffset - afterTimeOffset) + " ms");
```

```
}

```

上述程序在我笔记本上的执行结果为：

```
before: 1404363138126
after: 1404363138376
Create Costs: 250 ms
Exec Costs: 170 ms
```

从此结果可以清楚地看出，创建一个Connection对象，用了**250 毫秒**；而执行SQL的时间用了**170毫秒**。

创建一个Connection对象用了250毫秒！这个时间对计算机来说可以说是一个**非常奢侈的**！

这仅仅是一个Connection对象就有这么大的代价，设想一下另外一种情况：如果我们在Web应用程序中，为用户的每一个请求就操作一次数据库，当有10000个在线用户并发操作的话，对计算机而言，仅仅创建Connection对象不包括做业务的时间就要损耗10000×250ms= 250 0000 ms = 2500 s = 41.6667 min,竟然要**41**分钟！！！如果对高用户群体使用这样的系统，简直就是开玩笑！

2. 问题分析：

创建一个java.sql.Connection对象的代价是如此巨大，是因为创建一个Connection对象的过程，在底层就相当于和数据库建立的通信连接，在建立通信连接的过程，消耗了这么多的时间，而往往我们建立连接后（即创建Connection对象后），就执行一个简单的SQL语句，然后就要抛弃掉，这是一个非常大的资源浪费！

3.解决方案:

对于需要频繁地跟数据库交互的应用程序，可以在创建了Connection对象，并操作完数据库后，可以不释放掉资源，而是将它放到内存中，当下次需要操作数据库时，可以直接从内存中取出Connection对象，不需要再创建了，这样就极大地节省了创建Connection对象的资源消耗。由于内存也是有限和宝贵的，这又对我们对内存中的Connection对象怎么有效地维护提出了很高的要求。我们将在内存中存放Connection对象的容器称之为 连接池（Connection Pool）。下面让我们来看一下MyBatis的线程池是怎样实现的。

六、使用了连接池的PooledDataSource

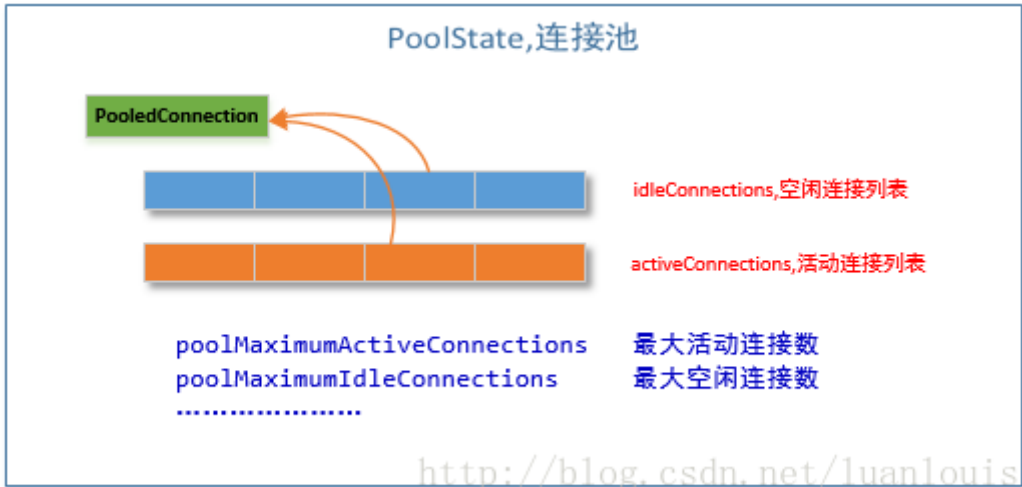
同样地，我们也是使用PooledDataSource的getConnection()方法来返回Connection对象。现在让我们看一下它的基本原理：

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，如果未滿，PooledDataSource会创建出一个PooledConnection，添加到此集合中，并返回。

PoolState连接池的大致结构如下所示：



6.1 获取java.sql.Connection对象的过程

下面我们看一下PooledDataSource 的getConnection()方法获取Connection对象的实现：

```
public Connection getConnection() throws SQLException {
    return popConnection(dataSource.getUsername(), dataSource.getPassword()).getProxyConnection();
}

public Connection getConnection(String username, String password) throws SQLException {
    return popConnection(username, password).getProxyConnection();
}
```

上述的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。（至于为什么会有getProxyConnection(),请关注下一节）

现在让我们看一下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

```
/*
 * 传递一个用户名和密码，从连接池中返回可用的PooledConnection
 */
private PooledConnection popConnection(String username, String password) throws SQLException
{
    boolean countedWait = false;
    PooledConnection conn = null;
    long t = System.currentTimeMillis();
    int localBadConnectionCount = 0;

    while (conn == null)
    {
        synchronized (state)
        {
            if (state.idleConnections.size() > 0)
            {
                // 连接池中有空闲连接，取出第一个
                conn = state.idleConnections.remove(0);
                if (log.isDebugEnabled())
                {
                    log.debug("Checked out connection " + conn.getRealHashCode() + " from pool.");
                }
            }
            else
            {
                // 连接池中沒有空闲连接，则取当前正在使用的连接数小于最大限定值，
                if (state.activeConnections.size() < poolMaximumActiveConnections)
                {
                    // 创建一个新的connection对象
                    conn = new PooledConnection(dataSource.getConnection(), this);
                    @SuppressWarnings("unused")
                    //used in logging, if enabled
                    Connection realConn = conn.getRealConnection();
                    if (log.isDebugEnabled())
                    {
                        log.debug("Created connection " + conn.getRealHashCode() + ".");
                    }
                }
                else
                {
                    // Cannot create new connection 当活动连接池已满，不能创建时，取出活动连接池的第一个，即最先进入连接池的Poo:
                    // 计算它的校验时间，如果校验时间大于连接池规定的最大校验时间，则认为它已经过期了，利用这个PoolConnection内部
                    //
                    PooledConnection oldestActiveConnection = state.activeConnections.get(0);
                    long longestCheckoutTime = oldestActiveConnection.getCheckoutTime();
                    if (longestCheckoutTime > poolMaximumCheckoutTime)
                    {
                        // Can claim overdue connection
                        state.claimedOverdueConnectionCount++;
                        state.accumulatedCheckoutTimeOfOverdueConnections += longestCheckoutTime;
                        state.accumulatedCheckoutTime += longestCheckoutTime;
                        state.activeConnections.remove(oldestActiveConnection);
                        if (!oldestActiveConnection.getRealConnection().getAutoCommit())
                        {
                            oldestActiveConnection.getRealConnection().rollback();
                        }
                    }
                }
            }
        }
    }
}
```

```

        conn = new PooledConnection(oldestActiveConnection.getRealConnection(), this);

        if (log.isDebugEnabled())
        {
            log.debug("Claimed overdue connection " + conn.getRealHashCode() + ".");
        }
    }
    else
    {
        //如果不能释放，则必须等待有
        // Must wait
        try
        {
            if (!countedWait)
            {
                state.hadToWaitCount++;
                countedWait = true;
            }
            if (log.isDebugEnabled())
            {
                log.debug("Waiting as long as " + poolTimeToWait + " milliseconds for connection.");
            }
            long wt = System.currentTimeMillis();
            state.wait(poolTimeToWait);
            state.accumulatedWaitTime += System.currentTimeMillis() - wt;
        }
        catch (InterruptedException e)
        {
            break;
        }
    }
}

//如果获取PooledConnection成功，则更新其信息

if (conn != null)
{
    if (conn.isValid())
    {
        if (!conn.getRealConnection().getAutoCommit())
        {
            conn.getRealConnection().rollback();
        }
        conn.setConnectionTypeCode(assembleConnectionTypeCode(dataSource.getUrl(), username, password));
        conn.setCheckoutTimestamp(System.currentTimeMillis());
        conn.setLastUsedTimestamp(System.currentTimeMillis());
        state.activeConnections.add(conn);
        state.requestCount++;
        state.accumulatedRequestTime += System.currentTimeMillis() - t;
    }
    else
    {
        if (log.isDebugEnabled())
        {
            log.debug("A bad connection (" + conn.getRealHashCode() + ") was returned from the pool, getting
        }
        state.badConnectionCount++;
        localBadConnectionCount++;
        conn = null;
        if (localBadConnectionCount > (poolMaximumIdleConnections + 3))
        {
            if (log.isDebugEnabled())
            {
                log.debug("PooledDataSource: Could not get a good connection to the database.");
            }
            throw new SQLException("PooledDataSource: Could not get a good connection to the database.");
        }
    }
}

```



```

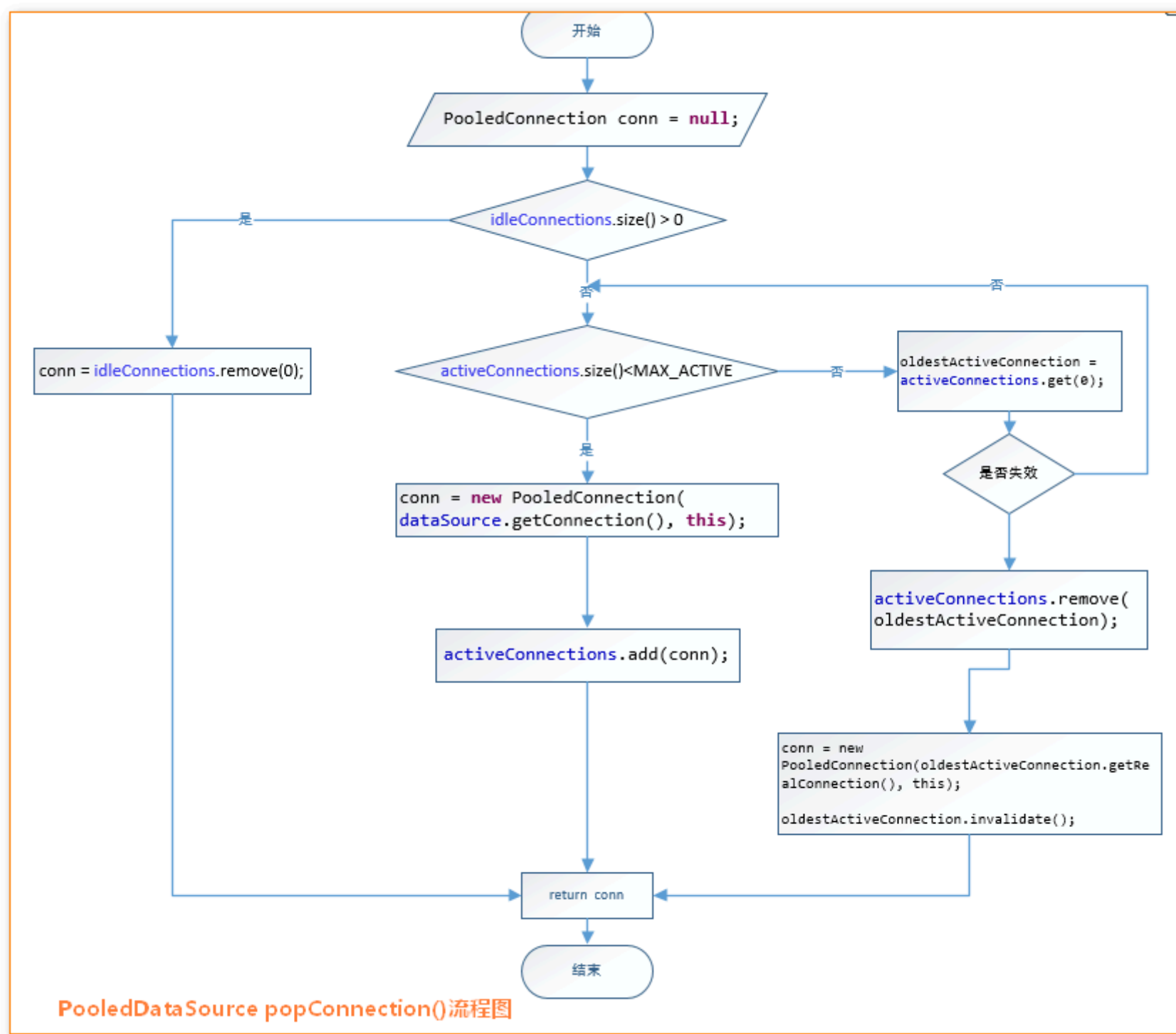
    }
    }
}

if (conn == null)
{
    if (log.isDebugEnabled())
    {
        log.debug("PooledDataSource: Unknown severe error condition. The connection pool returned a null connection.
    }
    throw new SQLException("PooledDataSource: Unknown severe error condition. The connection pool returned a null co
}

return conn;
}

```

对应的处理流程图如下所示:



<http://blog.csdn.net/luanlouis>

如上所示,对于PooledDataSource的getConnection()方法内,先是调用类PooledDataSource的popConnection()方法返回了一个PooledConnection对象,然后调用了PooledConnection的getProxyConnection()来返回Connection对象。

6.2.java.sql.Connection对象的回收

当我们的程序中使用完Connection对象时,如果不使用数据库连接池,我们一般会调用 connection.close()方法,关闭connection连接,释放资源。如下所示:

```

private void test() throws ClassNotFoundException, SQLException
{
    String sql = "select * from hr.employees where employee_id < ? and employee_id >= ?";
    PreparedStatement st = null;
    ResultSet rs = null;

    Connection con = null;

```

```

        Class.forName("oracle.jdbc.driver.OracleDriver");
    }
    try
    {
        con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "louluan", "123456");
        st = con.prepareStatement(sql);
        //设置参数
        st.setInt(1, 101);
        st.setInt(2, 0);
        //查询，得出结果集
        rs = st.executeQuery();
        //取数据，省略
        //关闭，释放资源
        con.close();
    }
    catch (SQLException e)
    {
        con.close();
        e.printStackTrace();
    }
}

```

调用过close()方法的Connection对象所持有的资源会被全部释放掉，Connection对象也就不能再使用。

那么，如果我们使用了连接池，我们在用完了Connection对象时，需要将它放在连接池中，该怎样做呢？

可能大家第一个在脑海里闪现出来的想法就是：我在应该调用con.close()方法的时候，不调用close()方法，将其换成将Connection对象放到连接池容器中的代码！好，我们将上述的想法实现，首先定义一个简易连接池Pool，然后将上面的代码改写：

```

package com.foo.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Vector;

/**
 *
 * 一个线程安全的简易连接池实现，此连接池是单例的
 * putConnection()将Connection添加到连接池中
 * getConnection()返回一个Connection对象
 */
public class Pool {

    private static Vector<Connection> pool = new Vector<Connection>();

    private static int MAX_CONNECTION =100;

    private static String DRIVER="oracle.jdbc.driver.OracleDriver";
    private static String URL = "jdbc:oracle:thin:@127.0.0.1:1521:xe";
    private static String USERNAME = "louluan";
    private static String PASSWROD = "123456";

    static {
        try {
            Class.forName(DRIVER);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    /**
     * 将一个Connection对象放置到连接池中
     */
    public static void putConnection(Connection connection){

        synchronized(pool)
        {
            if(pool.size()<MAX_CONNECTION)

```

```

        {
            |
            pool.add(connection);
        }
    }
}

/**
 * 返回一个Connection对象，如果连接池内有元素，则pop出第一个元素；
 * 如果连接池Pool中没有元素，则创建一个connection对象，然后添加到pool中
 * @return Connection
 */
public static Connection getConnection(){
    Connection connection = null;
    synchronized(pool)
    {
        if(pool.size()>0)
        {
            connection = pool.get(0);
            pool.remove(0);
        }
        else
        {
            connection = createConnection();
            pool.add(connection);
        }
    }
    return connection;
}

/**
 * 创建一个新的Connection对象
 */
private static Connection createConnection()
{
    Connection connection = null;
    try {
        connection = DriverManager.getConnection(URL, USERNAME,PASSWROD);
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return connection;
}
}

```

```

package com.foo.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Vector;

public class PoolTest
{

    private void test() throws ClassNotFoundException, SQLException
    {
        String sql = "select * from hr.employees where employee_id < ? and employee_id >= ?";
        PreparedStatement st = null;
        ResultSet rs = null;

        Connection con = null;
    }
}

```

```

        Class.forName("oracle.jdbc.driver.OracleDriver");
        try
        {
            con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "louluan", "123456");
            st = con.prepareStatement(sql);
            //设置参数
            st.setInt(1, 101);
            st.setInt(2, 0);
            //查询，得出结果集
            rs = st.executeQuery();
            //取数据，省略
            //将不再使用的Connection对象放到连接池中，供以后使用
            Pool.putConnection(con);
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
    }
}

```

上述的代码就是我们将使用过的Connection对象放到Pool连接池中，我们需要Connection对象的话，只需要使用Pool.getConnection()方法从里面取即可。

是的,上述的代码完全可以实现此能力，不过有一个很不优雅的实现：**就是我们需要手动地将Connection对象放到Pool连接池中，这是一个很傻的实现方式。这也和一般使用Connection对象的方式不一样：一般使用Connection的方式是使用完后，然后调用.close()方法释放资源。**

为了和一般的使用Conneciton对象的方式保持一致，我们希望当Connection使用完后，调用.close()方法，而实际上Connection资源并没有被释放，而实际上被添加到了连接池中。这样可以做到吗？答案是可以。上述的要求从另外一个角度来描述就是：能否提供一种机制，让我们知道Connection对象调用了什么方法，从而根据不同的方法自定义相应的处理机制。恰好代理机制就可以完成上述要求。

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

这是要使用代理模式，为真正的Connection对象创建一个代理对象，代理对象所有的方法都是调用相应的真正Connection对象的方法实现。当代理对象执行close()方法时，要特殊处理，不调用真正Connection对象的close()方法，而是将Connection对象添加到连接池中。

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：

其部分定义如下：

```

class PooledConnection implements InvocationHandler {

    //.....
    //所创建它的datasource引用
    private PooledDataSource dataSource;
    //真正的Connection对象
    private Connection realConnection;
    //代理自己的代理Connection
    private Connection proxyConnection;

    //.....
}

```

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：

```

public PooledConnection(Connection connection, PooledDataSource dataSource) {
    this.hashCode = connection.hashCode();
    this.realConnection = connection;
    this.dataSource = dataSource;
    this.createdTimestamp = System.currentTimeMillis();
    this.lastUsedTimestamp = System.currentTimeMillis();
    this.valid = true;
    this.proxyConnection = (Connection) Proxy.newProxyInstance(Connection.class.getClassLoader(), IFACES, this);
}

```

实际上，我们调用PooledDataSource的getConnection()方法返回的就是这个proxyConnection对象。

当我们调用此proxyConnection对象上的任何方法时，都会调用PooledConnection对象内invoke()方法。

让我们看一下PooledConnection类中的invoke()方法定义：

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    String methodName = method.getName();
    //当调用关闭的时候，回收此Connection到PooledDataSource中
    if (CLOSE.hashCode() == methodName.hashCode() && CLOSE.equals(methodName)) {
        dataSource.pushConnection(this);
        return null;
    } else {
        try {
            if (!Object.class.equals(method.getDeclaringClass())) {
                checkConnection();
            }
            return method.invoke(realConnection, args);
        } catch (Throwable t) {
            throw ExceptionUtil.unwrapThrowable(t);
        }
    }
}
```

从上述代码可以看到，当我们使用了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。

七、JNDI类型的数据源DataSource

对于JNDI类型的数据源DataSource的获取就比较简单，MyBatis定义了一个JndiDataSourceFactory工厂来创建通过JNDI形式生成的DataSource。

下面让我们看一下JndiDataSourceFactory的关键代码：

```
if (properties.containsKey(INITIAL_CONTEXT)
    && properties.containsKey(DATA_SOURCE))
{
    //从JNDI上下文中找到DataSource并返回
    Context ctx = (Context) initCtx.lookup(properties.getProperty(INITIAL_CONTEXT));
    dataSource = (DataSource) ctx.lookup(properties.getProperty(DATA_SOURCE));
}
else if (properties.containsKey(DATA_SOURCE))
{
    // //从JNDI上下文中找到DataSource并返回
    dataSource = (DataSource) initCtx.lookup(properties.getProperty(DATA_SOURCE));
}
```

《深入理解mybatis原理》 MyBatis的架构设计以及实例分析

原创

亦山

于 2014-11-04 16:44:53 发布

阅读量10w+

收藏 1.3k

点赞数 511

版权

分类专栏：

MyBatis

深入理解MyBatis原理

文章标签：

MyBatis

深入理解MyBatis原理

MyBatis原理

数据管理

ORM



MyBatis

同时被 2 个专栏收录

69 订阅

8 篇文章

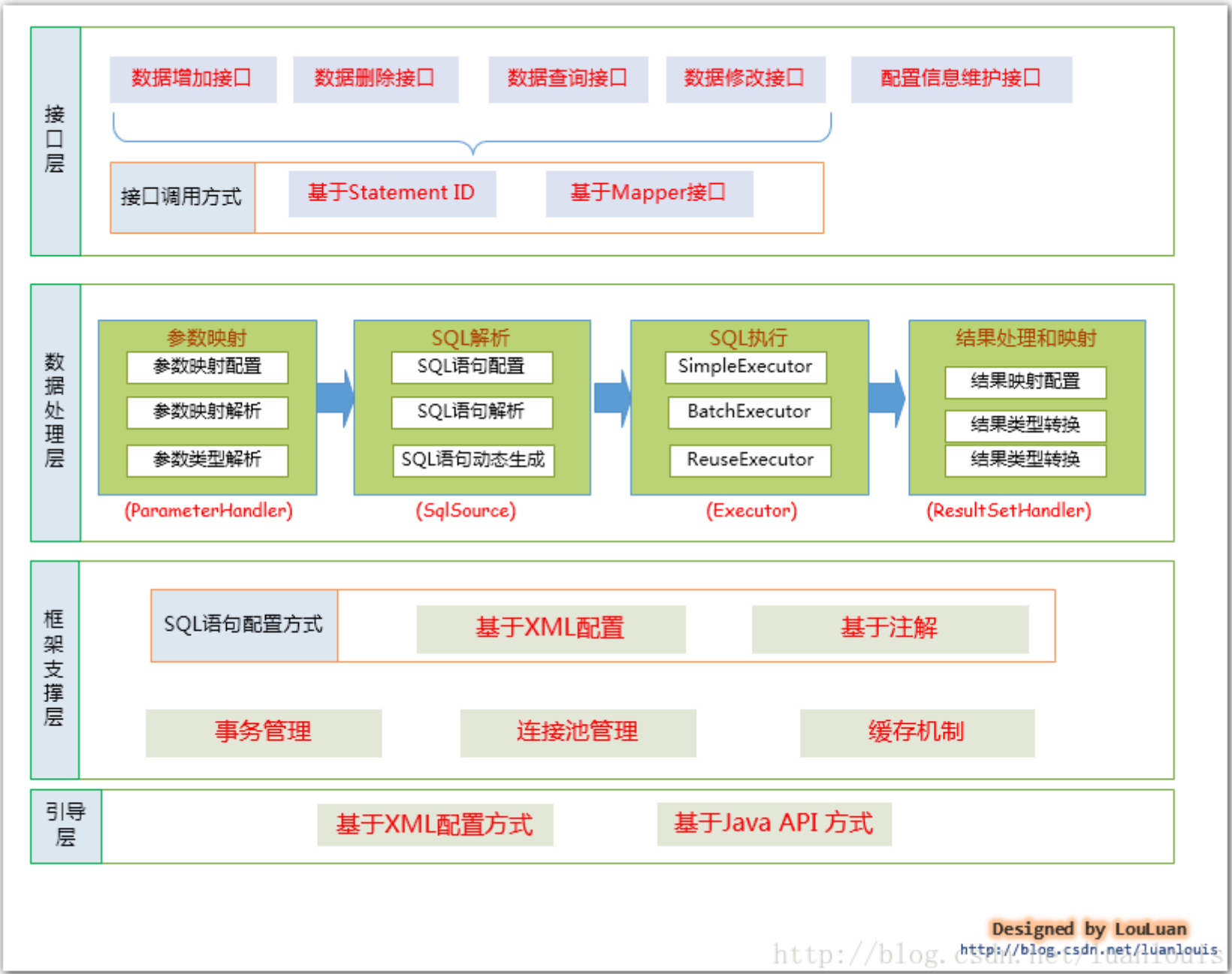
已订阅

篇文章

已订阅

MyBatis是目前非常流行的ORM框架，它的功能很强大，然而其实现却比较简单、优雅。本文主要讲述MyBatis的架构设计思路，并且讨论MyBatis的几个核心部件，然后结合一个select查询实例，深入代码，来探究MyBatis的实现。

一、MyBatis的框架设计



注：上图很大程

度上参考了iteye 上的chenjc_it 所写的博文原理分析之二：框架整体设计 中的MyBatis架构体图，chenjc_it总结的非常好，赞一个！

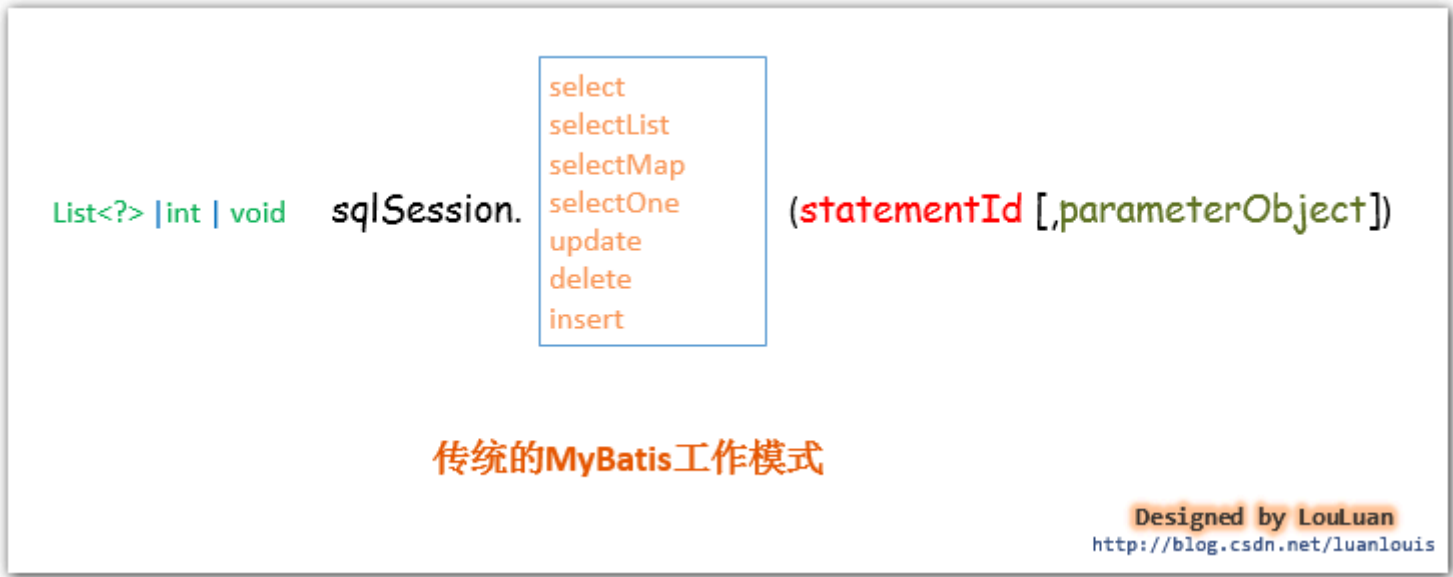
1.接口层---和数据库交互的方式

MyBatis和数据库的交互有两种方式：

- a.使用传统的MyBatis提供的API;
- b. 使用Mapper接口

1.1.使用传统的MyBatis提供的API

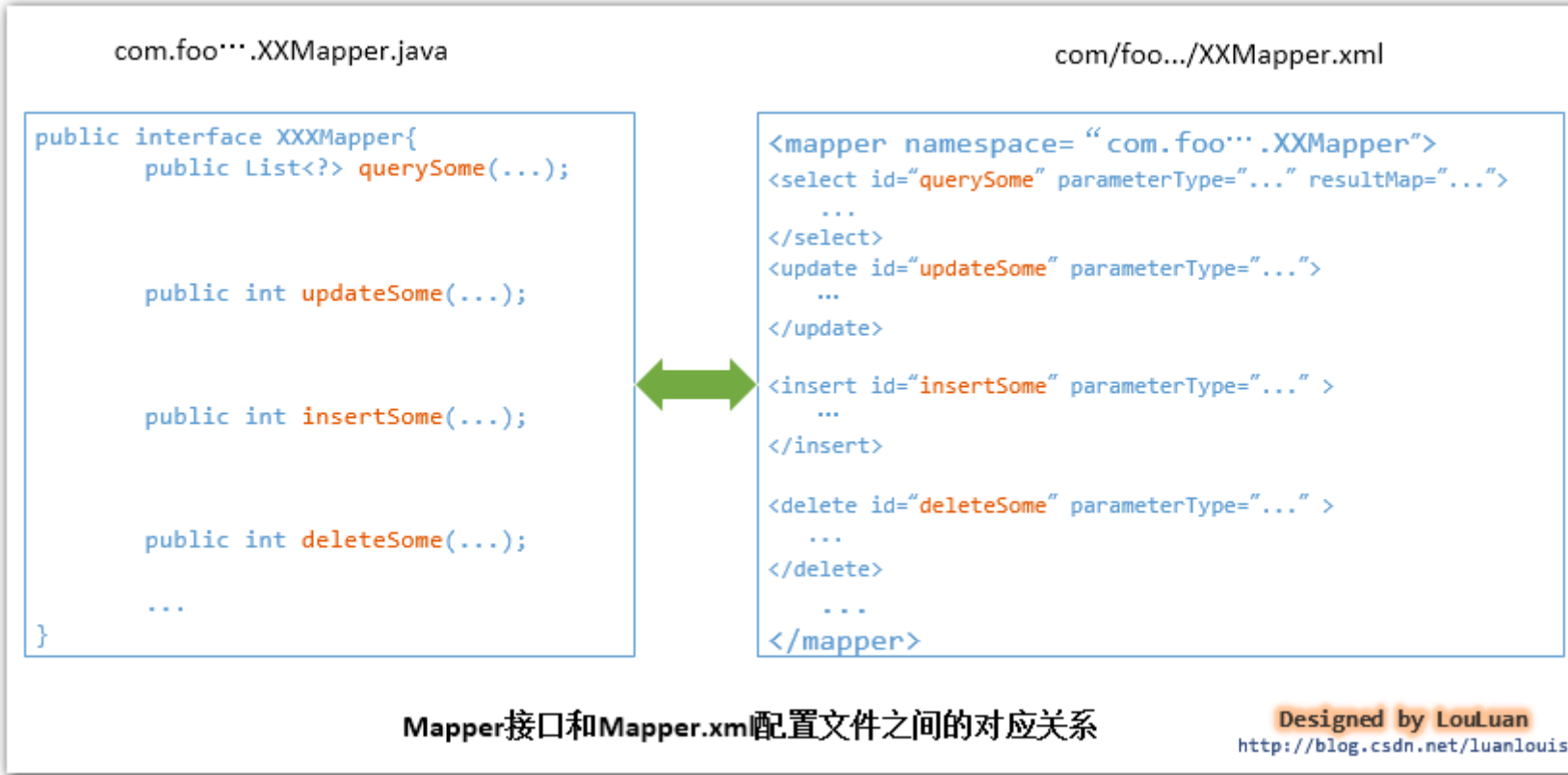
这是传统的传递Statement Id 和查询参数给 SqlSession 对象，使用 SqlSession对象完成和数据库的交互；MyBatis 提供了非常方便和简单的API，供用户实现对数据库的增删改查数据操作，以及对数据库连接信息和MyBatis 自身配置信息的维护操作。



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。

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，而这个接口中声明的方法和跟<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。



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，（至于这里的动态机制是怎样实现的，我将准备专门一片文章来讨论，敬请关注~）

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”）。

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，它要完成三个功能：

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

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。
(至于具体的MyBatis是如何动态构建SQL语句的，我将准备专门一篇文章来讨论，敬请关注~)

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。

3. 框架支撑层

3.1. 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，对于数据管理机制我已经在我的博文[《深入理解mybatis原理》](#) [MyBatis事务管理机制](#) 中有非常详细的讨论，感兴趣的读者可以点击查看。

3.2. 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，对于连接池管理机制我已经在我的博文[《深入理解mybatis原理》](#) [Mybatis数据源与连接池](#) 中有非常详细的讨论，感兴趣的读者可以点击查看。

3.3. 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。（至于具体的MyBatis缓存机制，我将准备专门一篇文章来讨论，敬请关注~）

3. 4. 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。

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，读者可以参考我的另一片博文：[Java Persistence with MyBatis 3\(中文版\) 第二章 引导MyBatis](#)

二、MyBatis的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：

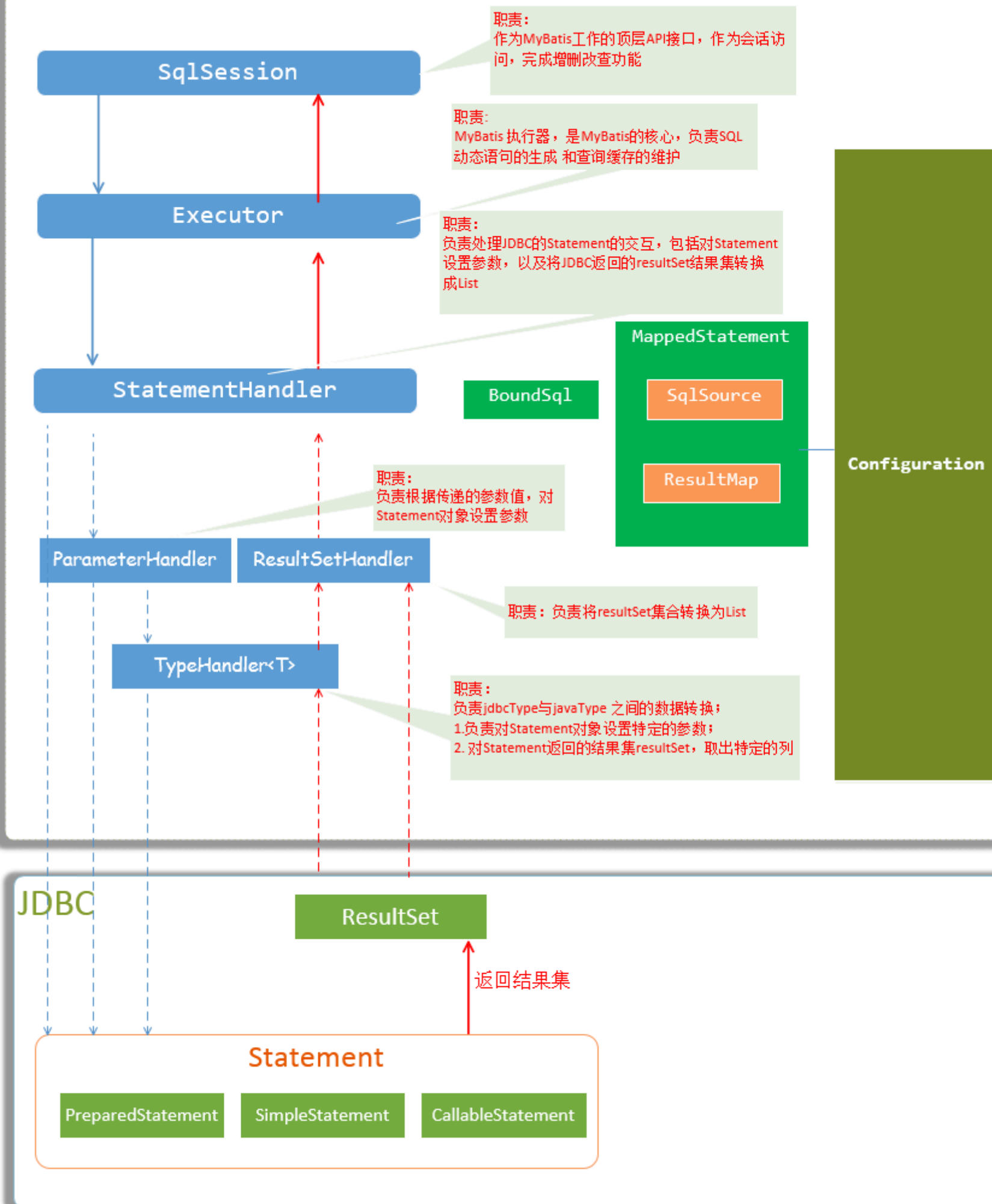
- SqlSession** 作为MyBatis工作的主要顶层API，表示和数据库交互的会话，完成必要数据库增删改查功能

- **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。

（注：这里只是列出了我个人认为属于核心的部件，请读者不要先入为主，认为MyBatis就只有这些部件哦！每个人对MyBatis的理解不同，分析出的结果自然会有所不同，欢迎读者提出质疑和不同的意见，我们共同探讨~）

它们的关系如下图所示：

MyBatis层次结构



Designed by LouLuan

<http://blog.csdn.net/luanlou1985>

三、从MyBatis一次select 查询语句来分析MyBatis的架构设计

一、数据准备 (非常熟悉和应用过MyBatis 的读者可以迅速浏览此节即可)

1. 准备数据库数据, 创建EMPLOYEES表, 插入数据:

```
--创建一个员工基本信息表
create table "EMPLOYEES"(
    "EMPLOYEE_ID" NUMBER(6) not null,
    "FIRST_NAME" VARCHAR2(20),
    "LAST_NAME" VARCHAR2(25) not null,
    "EMAIL" VARCHAR2(25) not null unique,
    "SALARY" NUMBER(8,2),
    constraint "EMP_EMP_ID_PK" primary key ("EMPLOYEE_ID")
);
comment on table EMPLOYEES is '员工信息表';
comment on column EMPLOYEES.EMPLOYEE_ID is '员工id';
comment on column EMPLOYEES.FIRST_NAME is 'first name';
comment on column EMPLOYEES.LAST_NAME is 'last name';
comment on column EMPLOYEES.EMAIL is 'email address';
comment on column EMPLOYEES.SALARY is 'salary';

--添加数据
insert into EMPLOYEES (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, SALARY)
values (100, 'Steven', 'King', 'SKING', 24000.00);

insert into EMPLOYEES (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, SALARY)
values (101, 'Neena', 'Kochhar', 'NKOCHHAR', 17000.00);

insert into EMPLOYEES (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, SALARY)
values (102, 'Lex', 'De Haan', 'LDEHAAN', 17000.00);

insert into EMPLOYEES (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, SALARY)
values (103, 'Alexander', 'Hunold', 'AHUNOLD', 9000.00);

insert into EMPLOYEES (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, SALARY)
values (104, 'Bruce', 'Ernst', 'BERNST', 6000.00);

insert into EMPLOYEES (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, SALARY)
values (105, 'David', 'Austin', 'DAUSTIN', 4800.00);

insert into EMPLOYEES (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, SALARY)
values (106, 'Valli', 'Pataballa', 'VPATABAL', 4800.00);

insert into EMPLOYEES (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, SALARY)
values (107, 'Diana', 'Lorentz', 'DLORENTZ', 4200.00);
```

2. 配置Mybatis的配置文件，命名为mybatisConfig.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC" />
            <dataSource type="POOLED">
                <property name="driver" value="oracle.jdbc.driver.OracleDriver" />
                <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
                <property name="username" value="louis" />
                <property name="password" value="123456" />
            </dataSource>
        </environment>
    </environments>
    <mappers>
```

```
        <mapper resource="com/louis/mybatis/domain/EmployeesMapper.xml"/>
    </mappers>    </configuration>
```

3. 创建Employee实体Bean 以及配置Mapper配置文件

```
package com.louis.mybatis.model;

import java.math.BigDecimal;

public class Employee {
    private Integer employeeId;

    private String firstName;

    private String lastName;

    private String email;

    private BigDecimal salary;

    public Integer getEmployeeId() {
        return employeeId;
    }

    public void setEmployeeId(Integer employeeId) {
        this.employeeId = employeeId;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public BigDecimal getSalary() {
        return salary;
    }

    public void setSalary(BigDecimal salary) {
        this.salary = salary;
    }
}
```

```
}

```

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.louis.mybatis.dao.EmployeesMapper" >

    <resultMap id="BaseResultMap" type="com.louis.mybatis.model.Employee" >
        <id column="EMPLOYEE_ID" property="employeeId" jdbcType="DECIMAL" />
        <result column="FIRST_NAME" property="firstName" jdbcType="VARCHAR" />
        <result column="LAST_NAME" property="lastName" jdbcType="VARCHAR" />
        <result column="EMAIL" property="email" jdbcType="VARCHAR" />
        <result column="SALARY" property="salary" jdbcType="DECIMAL" />
    </resultMap>

    <select id="selectByPrimaryKey" resultMap="BaseResultMap" parameterType="java.lang.Integer" >
        select
            EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, SALARY
        from LOUIS.EMPLOYEES
        where EMPLOYEE_ID = #{employeeId,jdbcType=DECIMAL}
    </select>
</mapper>
```

4. 创建eclipse 或者myeclipse 的maven项目，maven配置如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>batis</groupId>
    <artifactId>batis</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>batis</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.2.7</version>
        </dependency>

        <dependency>
            <groupId>com.oracle</groupId>
            <artifactId>ojdbc14</artifactId>
```



```
        <version>10.2.0.4.0</version>    </dependency>

    </dependencies>
</project>
```

5. 客户端代码:

```
package com.louis.mybatis.test;

import java.io.InputStream;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import com.louis.mybatis.model.Employee;

/**
 * SqlSession 简单查询演示类
 * @author louluan
 */
public class SelectDemo {

    public static void main(String[] args) throws Exception {
        /*
         * 1.加载mybatis的配置文件，初始化mybatis，创建出SqlSessionFactory，是创建SqlSession的工厂
         * 这里只是为了演示的需要，SqlSessionFactory临时创建出来，在实际的使用中，SqlSessionFactory只需要创建一次，当作单例来使用
         */
        InputStream inputStream = Resources.getResourceAsStream("mybatisConfig.xml");
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);

        //2. 从SqlSession工厂 SqlSessionFactory中创建一个SqlSession，进行数据库操作
        SqlSession sqlSession = factory.openSession();

        //3.使用SqlSession查询
        Map<String,Object> params = new HashMap<String,Object>();

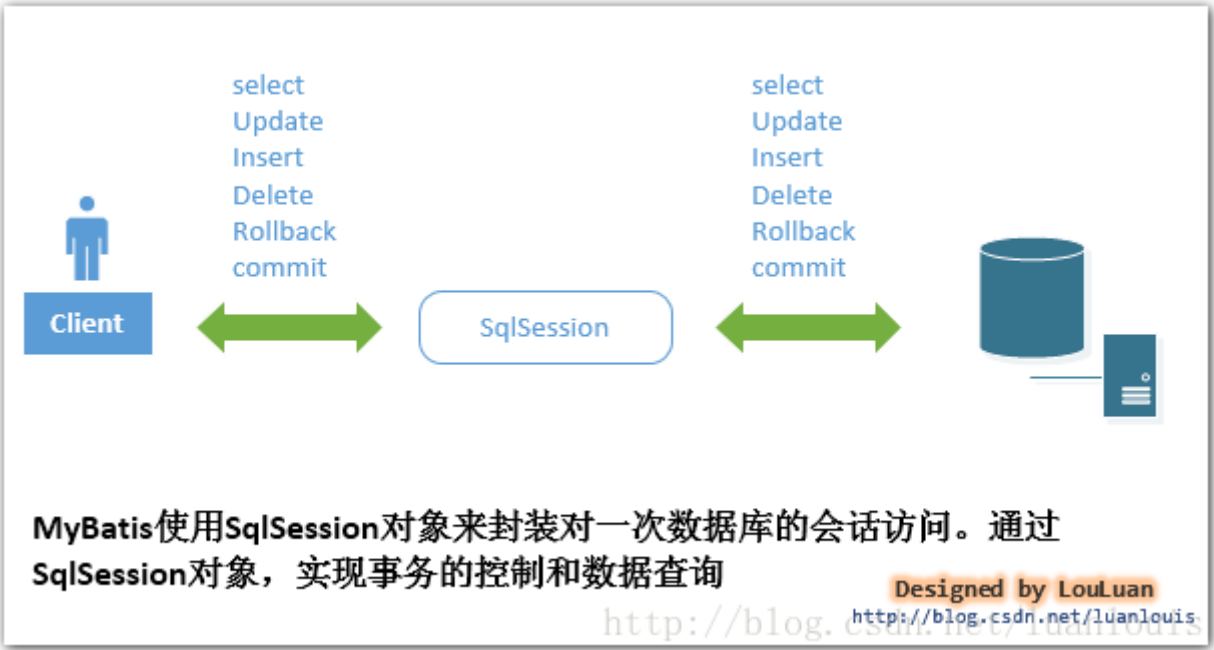
        params.put("min_salary",10000);
        //a.查询工资低于10000的员工
        List<Employee> result = sqlSession.selectList("com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary",params);
        //b.未传最低工资，查所有员工
        List<Employee> result1 = sqlSession.selectList("com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary");
        System.out.println("薪资低于10000的员工数: "+result.size());
        //~output :  查询到的数据总数: 5
        System.out.println("所有员工数: "+result1.size());
        //~output :  所有员工数: 8
    }
}
```

二、SqlSession 的工作过程分析:

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。



2. 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。

3.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, boolean isLocalCacheOnly, 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, boolean isLocalCacheOnly) {
    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, Bound
    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, bound
        //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（具体它的缓存机制不是本文的重点，我会单独拿出来跟大家探讨，感兴趣的读者可以关注我的其他博文）；
- (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, Bound

    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置。StatementHandler通过parameterize(statement)方法对Statement进行设置；
- (2.StatementHandler通过List<E> query(Statement statement, ResultHandler resultHandler)方法来完成执行Statement，并将Statement对象返回的resultSet封装成List；

5. 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的设置
    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。

6. 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);
}
```

```
/**
 *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.getResultSets();
    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);
}
```

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:

```
//
// DefaultResultSetHandler 类的handleResultSets(Statement stmt)实现
//HANDLE RESULT SETS
//

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);
}
```

由于上述的过程时序图太过复杂，就不贴出来了，读者可以下载MyBatis源码，使用Eclipse、IntelliJ IDEA、NetBeans 等IDE集成环境创建项目，Debug MyBatis源码，一步步跟踪MyBatis的实现，这样对学习MyBatis框架很有帮助~

作者的话

本文是《深入理解mybatis原理》系列的其中一篇，如果您有兴趣，请关注该系列的其他文章~
觉得本文不错，顺手点个赞哦~~您的鼓励，是我继续分享知识的强大动力！

《深入理解mybatis原理》 MyBatis的一级缓存实现详解 及使用注意事项

原创

亦山

于 2014-11-21 23:01:28 发布

阅读量5.1w

收藏 356

点赞数 179

版权

分类专栏：

MyBatis

MyBatis教程

深入理解MyBatis原理

文章标签：


MyBatis原理

MyBatis

ORM

缓存机制

源码

 MyBatis

同时被 3 个专栏收录 ▼

69 订阅

8 篇文章

已订阅

篇文章

已订阅

0.写在前面

MyBatis是一个简单，小巧但功能非常强大的ORM开源框架，它的功能强大也体现在它的缓存机制上。MyBatis提供了一级缓存、二级缓存 这两个缓存机制，能够很好地处理和维护缓存，以提高系统的性能。本文的目的则是向读者详细介绍MyBatis的一级缓存，深入源码，解析MyBatis一级缓存的实现原理，并且针对一级缓存的特点提出了在实际使用过程中应该注意的事项。

读完本文，你将会学到：

1、什么是一级缓存？为什么使用一级缓存？

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

3、一级缓存的生命周期有多长？

4、Cache接口的设计以及CacheKey的定义

5、一级缓存的性能分析以及应该注意的事项

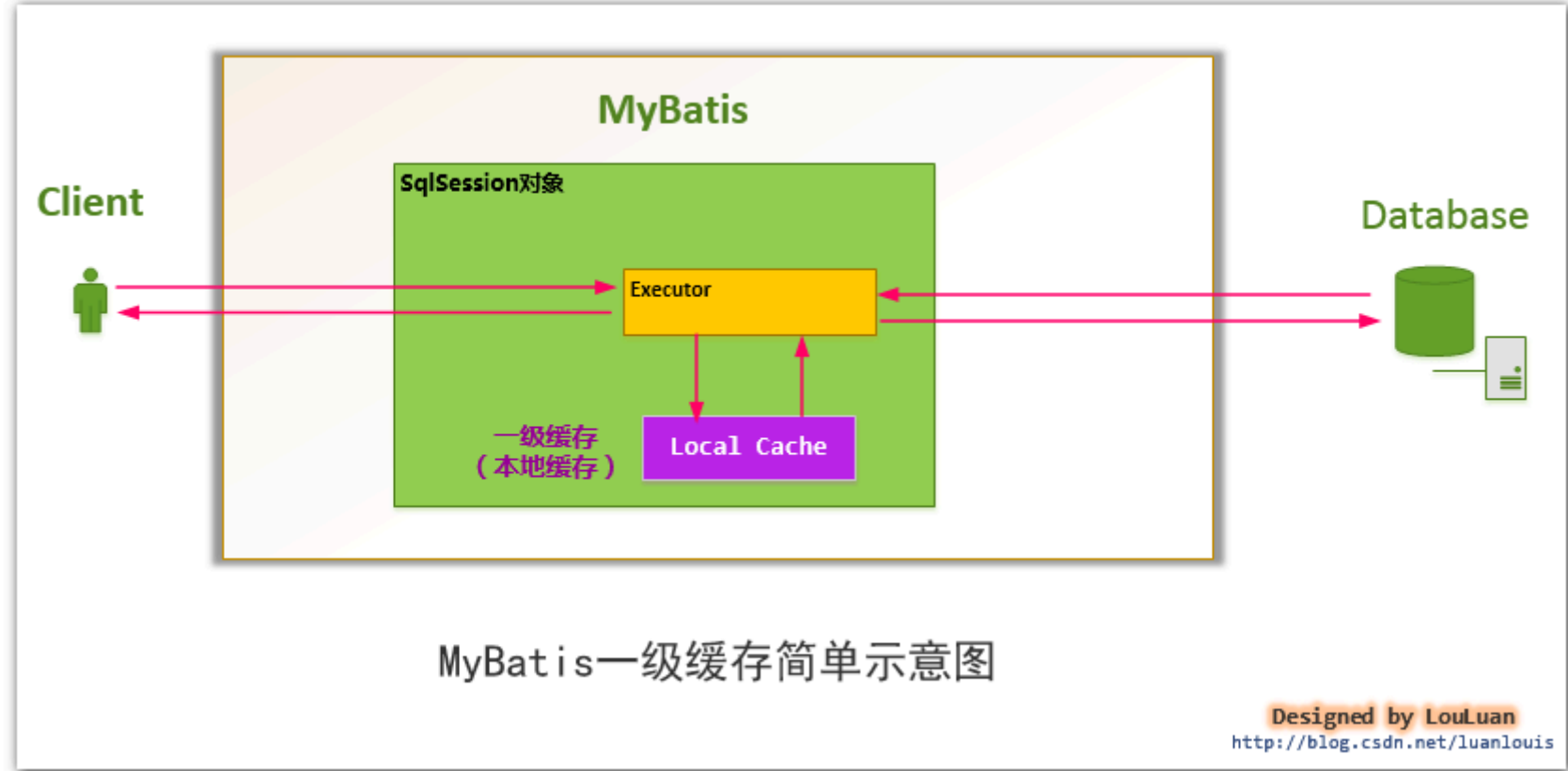
1. 什么是一级缓存？ 为什么使用一级缓存？

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。



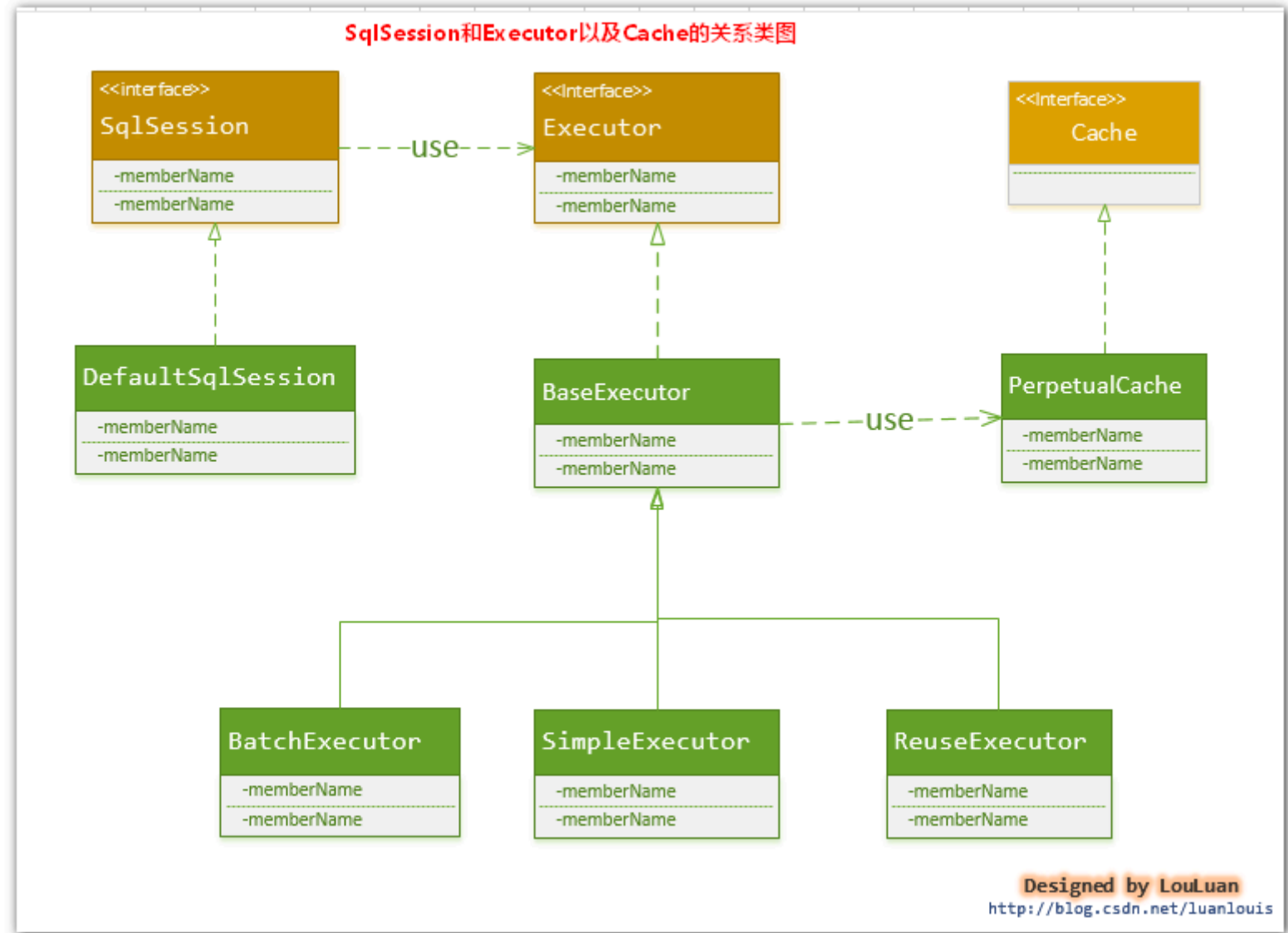
对于会话（Session）级别的数据缓存，我们称之为一级数据缓存，简称一级缓存。

2. 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。

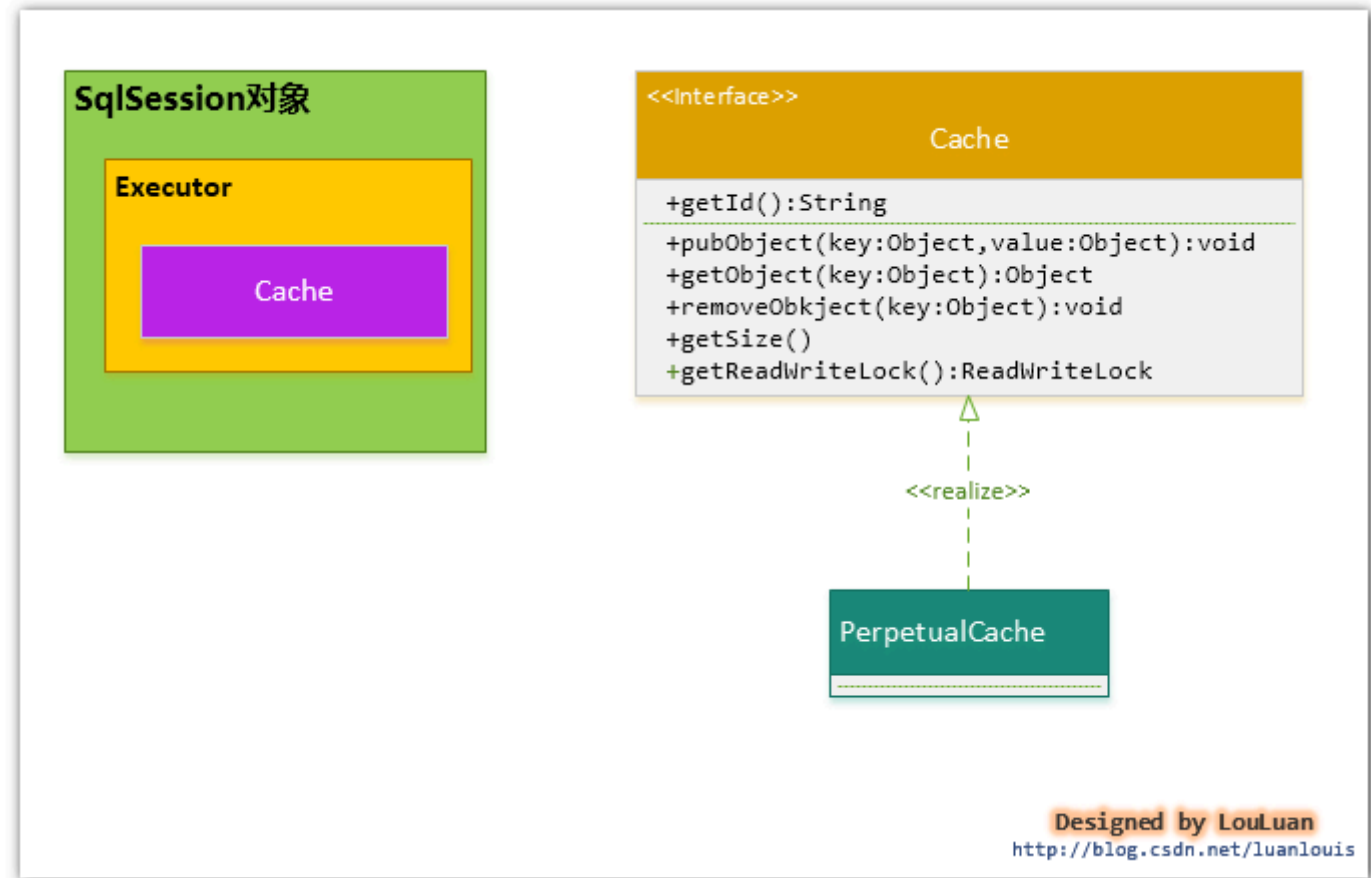
实际上, **MyBatis**只是一个**MyBatis**对外的接口, **SqlSession**将它的工作交给了**Executor**执行器这个角色来完成, 负责完成对数据库的各种操作。当创建了一个**SqlSession**对象时, **MyBatis**会为这个**SqlSession**对象创建一个新的**Executor**执行器, 而缓存信息就被维护在这个**Executor**执行器中, **MyBatis**将缓存和对缓存相关的操作封装成了Cache接口中。

SqlSession、**Executor**、**Cache**之间的关系如下列类图所示:



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。如下是**PerpetualCache**的实现代码:

```
package org.apache.ibatis.cache.impl;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.locks.ReadWriteLock;

import org.apache.ibatis.cache.Cache;
import org.apache.ibatis.cache.CacheException;

/**
 * 使用简单的HashMap来维护缓存
 * @author Clinton Begin
 */
```

```

public class PerpetualCache implements Cache {

    private String id;

    private Map<Object, Object> cache = new HashMap<Object, Object>();

    public PerpetualCache(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }

    public int getSize() {
        return cache.size();
    }

    public void putObject(Object key, Object value) {
        cache.put(key, value);
    }

    public Object getObject(Object key) {
        return cache.get(key);
    }

    public Object removeObject(Object key) {
        return cache.remove(key);
    }

    public void clear() {
        cache.clear();
    }

    public ReadWriteLock getReadWriteLock() {
        return null;
    }

    public boolean equals(Object o) {
        if (getId() == null) throw new CacheException("Cache instances require an ID.");
        if (this == o) return true;
        if (!(o instanceof Cache)) return false;

        Cache otherCache = (Cache) o;
        return getId().equals(otherCache.getId());
    }

    public int hashCode() {
        if (getId() == null) throw new CacheException("Cache instances require an ID.");
        return getId().hashCode();
    }
}

```

3.一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；

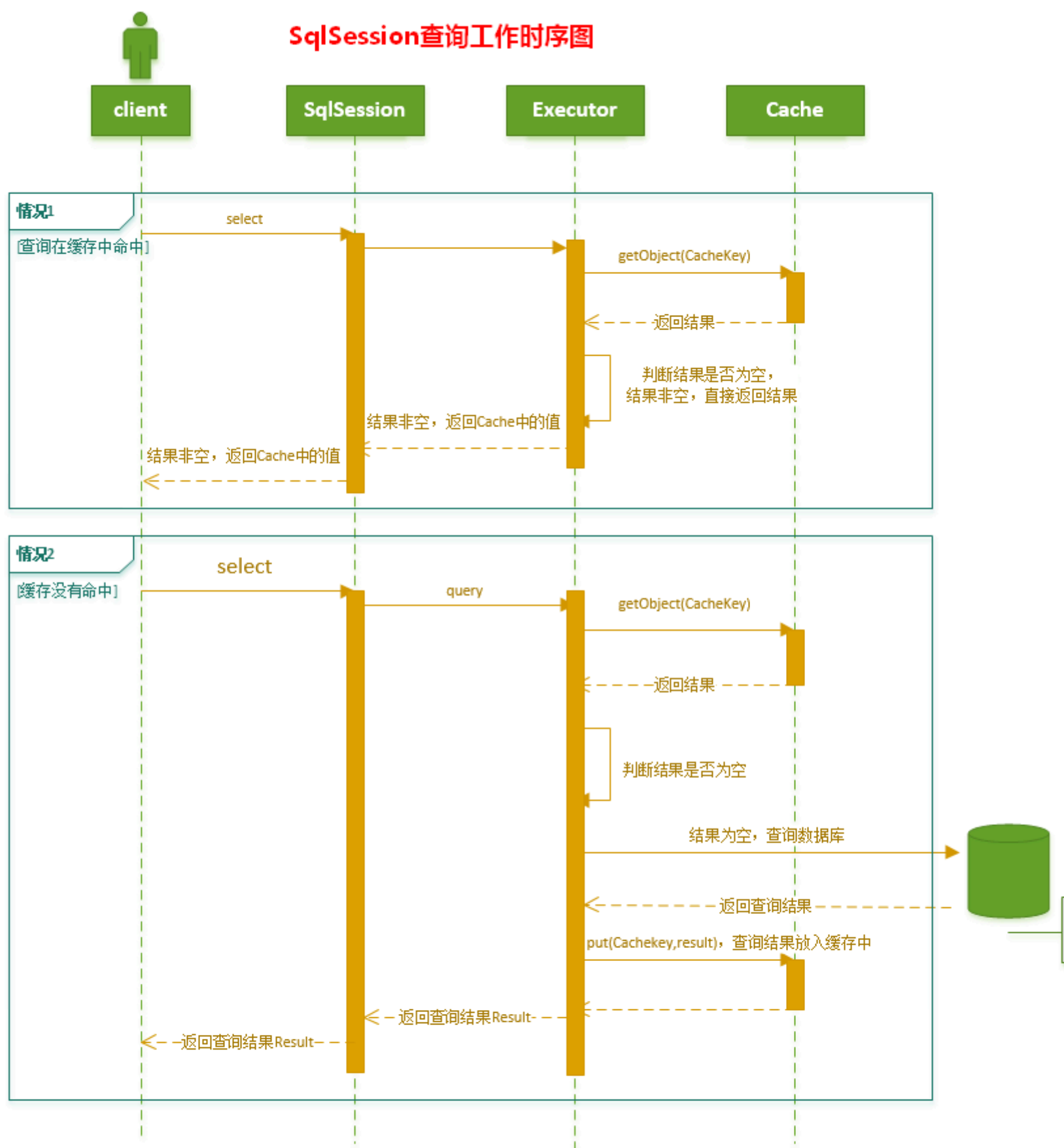


4. SqlSession 一级缓存的工作流程：

- 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。

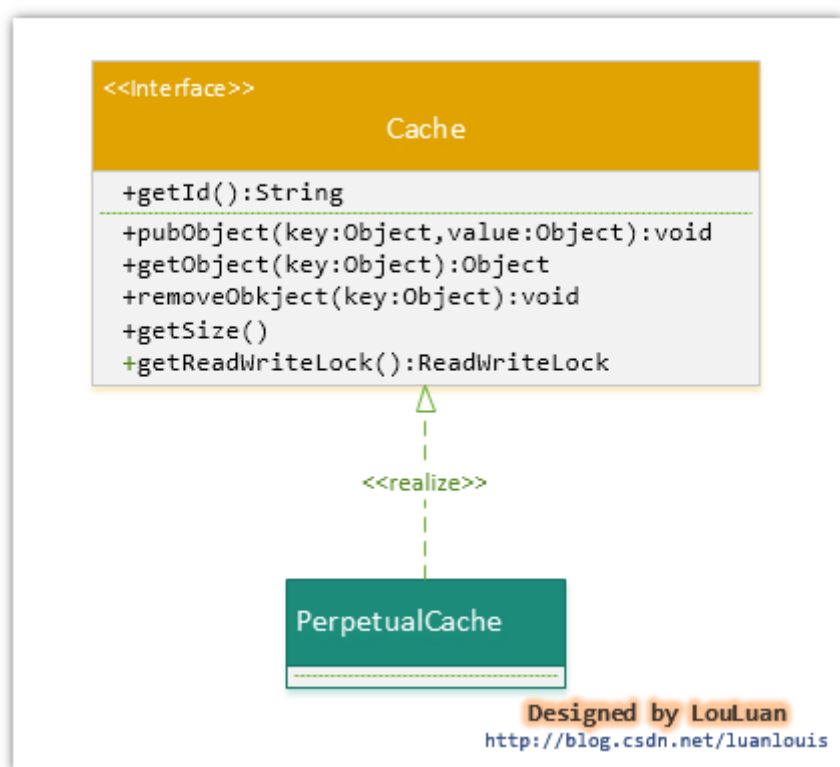
[关于上述工作过程中 key值的构建，我们将在第下一节中重点探讨，这也是MyBatis缓存机制中非常重要的一个概念。]

SqlSession查询工作时序图



5. 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。

上述的第3个条件正是要求保证传递给**JDBC**的**SQL**语句完全一致；第4条则是保证传递给**JDBC**的参数也完全一致；

3、4讲的有可能比较含糊，举一个例子：

```
<select id="selectByCriteria" parameterType="java.util.Map" resultMap="BaseResultMap">
    select employee_id,first_name,last_name,email,salary
    from louis.employees
    where employee_id = #{employeeId}
    and first_name= #{firstName}
    and last_name = #{lastName}
    and email = #{email}
</select>
```

如果使用上述的**"selectByCriteria"**进行查询，那么，**MyBatis**会将上述的**SQL**中的**#{}**都替换成**?**如下：

```
select employee_id,first_name,last_name,email,salary
from louis.employees
where employee_id = ?
and first_name= ?
and last_name = ?
and email = ?
```

MyBatis最终会使用上述的**SQL**字符串创建**JDBC**的**java.sql.PreparedStatement**对象，对于这个**PreparedStatement**对象，还需要对它设置参数，调用**setXXX()**来完成设值，第4条的条件，就是要求对设置**JDBC**的**PreparedStatement**的参数值也要完全一致。

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

CacheKey的创建

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：

```
/**
 * 所属类： org.apache.ibatis.executor.BaseExecutor
 * 功能    ： 根据传入信息构建CacheKey
 */
public CacheKey createCacheKey(MappedStatement ms, Object parameterObject, RowBounds rowBounds, BoundSql boundSql) {
    if (closed) throw new ExecutorException("Executor was closed.");
    CacheKey cacheKey = new CacheKey();
    //1.statementId
    cacheKey.update(ms.getId());
    //2. rowBounds.offset
    cacheKey.update(rowBounds.getOffset());
    //3. rowBounds.limit
    cacheKey.update(rowBounds.getLimit());
    //4. SQL语句
    cacheKey.update(boundSql.getSql());
    //5. 将每一个要传递给JDBC的参数值也更新到CacheKey中
    List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
    TypeHandlerRegistry typeHandlerRegistry = ms.getConfiguration().getTypeHandlerRegistry();
    for (int i = 0; i < parameterMappings.size(); i++) { // mimic DefaultParameterHandler logic
        ParameterMapping parameterMapping = parameterMappings.get(i);
        if (parameterMapping.getMode() != ParameterMode.OUT) {
            Object value;
            String propertyName = parameterMapping.getProperty();
            if (boundSql.hasAdditionalParameter(propertyName)) {
                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);
            }
            //将每一个要传递给JDBC的参数值也更新到CacheKey中
            cacheKey.update(value);
        }
    }
    return cacheKey;
}
```

CacheKey的hashCode生成算法

刚才已经提到，Cache接口的实现，本质上是使用的HashMap<k,v>,而构建CacheKey的目的就是为了作为HashMap<k,v>中的key值。而HashMap是通过key值的hashCode 来组织和存储的，那么，构建CacheKey的过程实际上就是构造其hashCode的过程。下面的代码就是CacheKey的核心hashCode生成算法，感兴趣的话可以看一下：

```
public void update(Object object) {
    if (object != null && object.getClass().isArray()) {
        int length = Array.getLength(object);
        for (int i = 0; i < length; i++) {
```



```
        Object element = Array.get(object, i);
        doUpdate(element);
    } else {
        doUpdate(object);
    }
}

private void doUpdate(Object object) {

    //1. 得到对象的hashCode;
    int baseHashCode = object == null ? 1 : object.hashCode();
    //对象计数递增
    count++;
    checksum += baseHashCode;
    //2. 对象的hashCode 扩大count倍
    baseHashCode *= count;
    //3. hashCode * 拓展因子（默认37）+拓展扩大后的对象hashCode值
    hashCode = multiplier * hashCode + baseHashCode;
    updateList.add(object);
}
```

一级缓存的性能分析

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。

举例：

例1、看下面这个例子，下面的例子使用了同一个**SqlSession**指令了两次完全一样的查询，将两次查询所耗的时间打印出来，结果如下：

```
package com.louis.mybatis.test;

import java.io.InputStream;
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```

import org.apache.commons.logging.Log;      | import org.apache.commons.logging.LogFactory;
import org.apache.ibatis.executor.BaseExecutor;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.apache.log4j.Logger;

import com.louis.mybatis.model.Employee;

/**
 * SqlSession 简单查询演示类
 * @author louluan
 */
public class SelectDemo1 {

    private static final Logger logger = Logger.getLogger(SelectDemo1.class);

    public static void main(String[] args) throws Exception {
        InputStream inputStream = Resources.getResourceAsStream("mybatisConfig.xml");
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory factory = builder.build(inputStream);

        SqlSession sqlSession = factory.openSession();
        //3.使用SqlSession查询
        Map<String, Object> params = new HashMap<String, Object>();
        params.put("min_salary", 10000);
        //a.查询工资低于10000的员工
        Date first = new Date();
        //第一次查询
        List<Employee> result = sqlSession.selectList("com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary", params);
        logger.info("first quest costs:" + (new Date().getTime() - first.getTime()) + " ms");
        Date second = new Date();
        result = sqlSession.selectList("com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary", params);
        logger.info("second quest costs:" + (new Date().getTime() - second.getTime()) + " ms");
    }
}

```

运行结果：

```

2014-11-19 16:07:09,145 [main] INFO    com.louis.mybatis.test.SelectDemo1 - first quest costs:464 ms
2014-11-19 16:07:09,146 [main] INFO    com.louis.mybatis.test.SelectDemo1 - second quest costs:0 ms

```

Designed by LouLuan
<http://blog.csdn.net/luanlouis>

由上面的结果你可以看到，第一次查询耗时464ms，而第二次查询耗时不足1ms,这是因为第一次查询后，MyBatis会将查询结果存储到SqlSession对象的缓存中，当后来有完全相同的查询时，直接从缓存中将结果取出。

例2、对上面的例子做一下修改：在第二次调用查询前，对参数 HashMap类型的params多增加一些无关的值进去，然后再执行，看查询结果：

```

2014-11-20 17:16:57,765 [main] INFO    com.louis.mybatis.test.SelectDemo1 - first quest costs:434 ms
2014-11-20 17:16:57,766 [main] INFO    com.louis.mybatis.test.SelectDemo1 - second quest costs:0 ms

```

Designed by LouLuan
<http://blog.csdn.net/luanlouis>

从结果上看，虽然第二次查询时传递的params参数不一致，但还是从一级缓存中取出了第一次查询的缓存。

读到这里，请读者晓得这一个问题：

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。

《深入理解mybatis原理》 MyBatis的二级缓存的设计原理

原创

亦山

于 2014-11-23 13:53:26 发布

阅读量4.4w

收藏 284

点赞数 129

分类专栏：

MyBatis

MyBatis教程

深入理解MyBatis原理

文章标签：

MyBatis原理

MyBatis

设计模式

缓存机制

cache



MyBatis

同时被 3 个专栏收录

69 订阅

8 篇文章

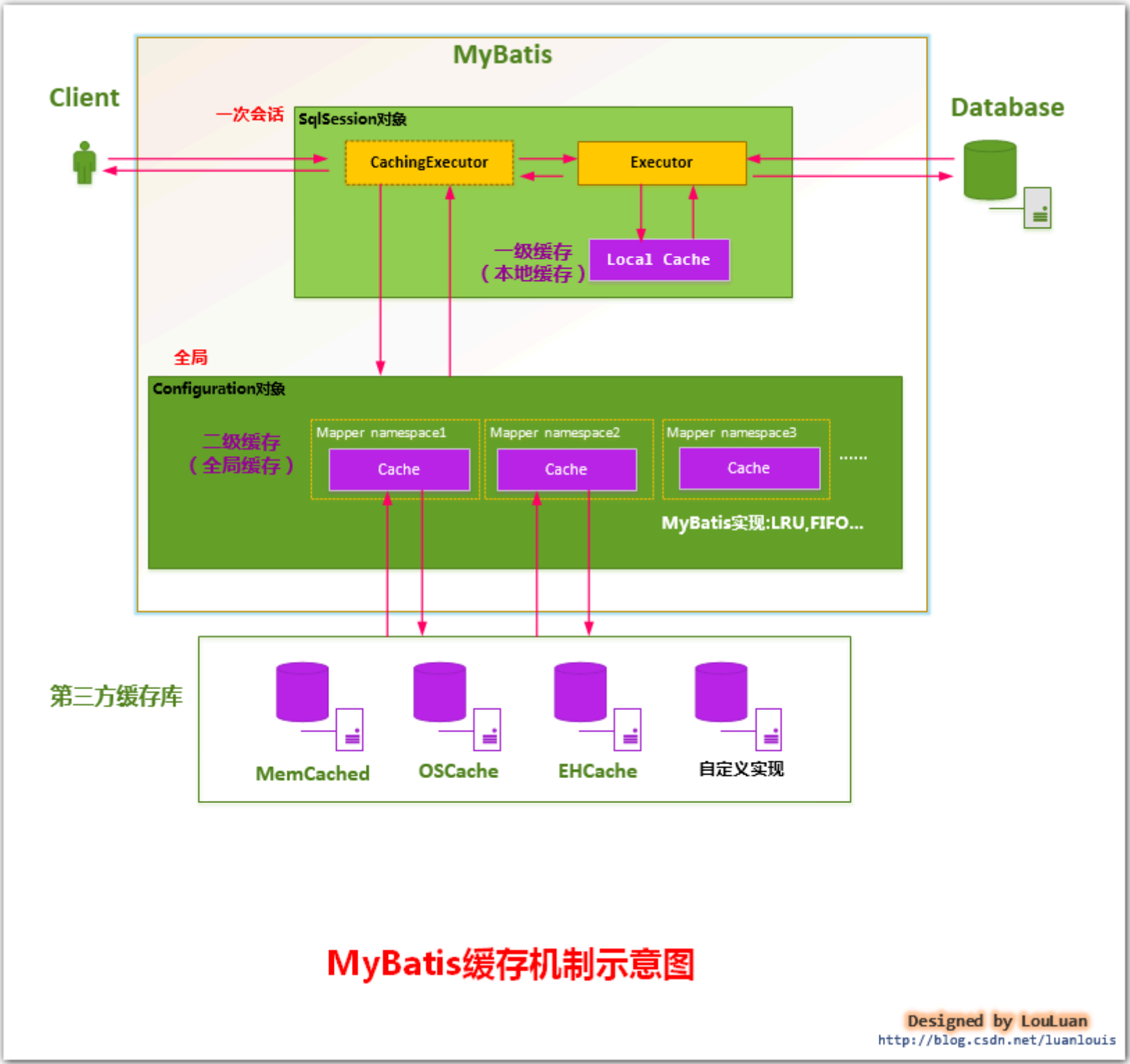
已订阅

篇文章

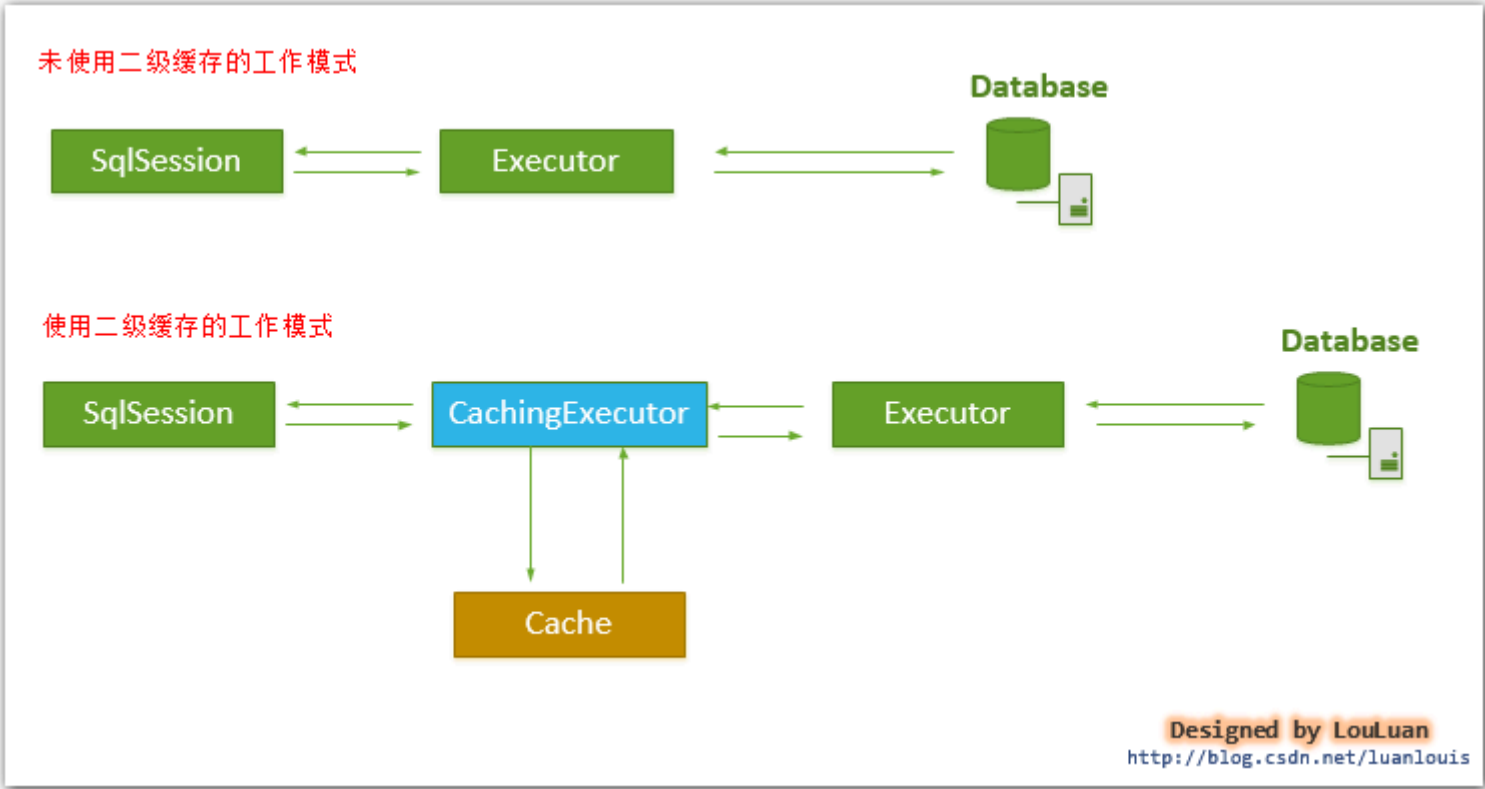
已订阅

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

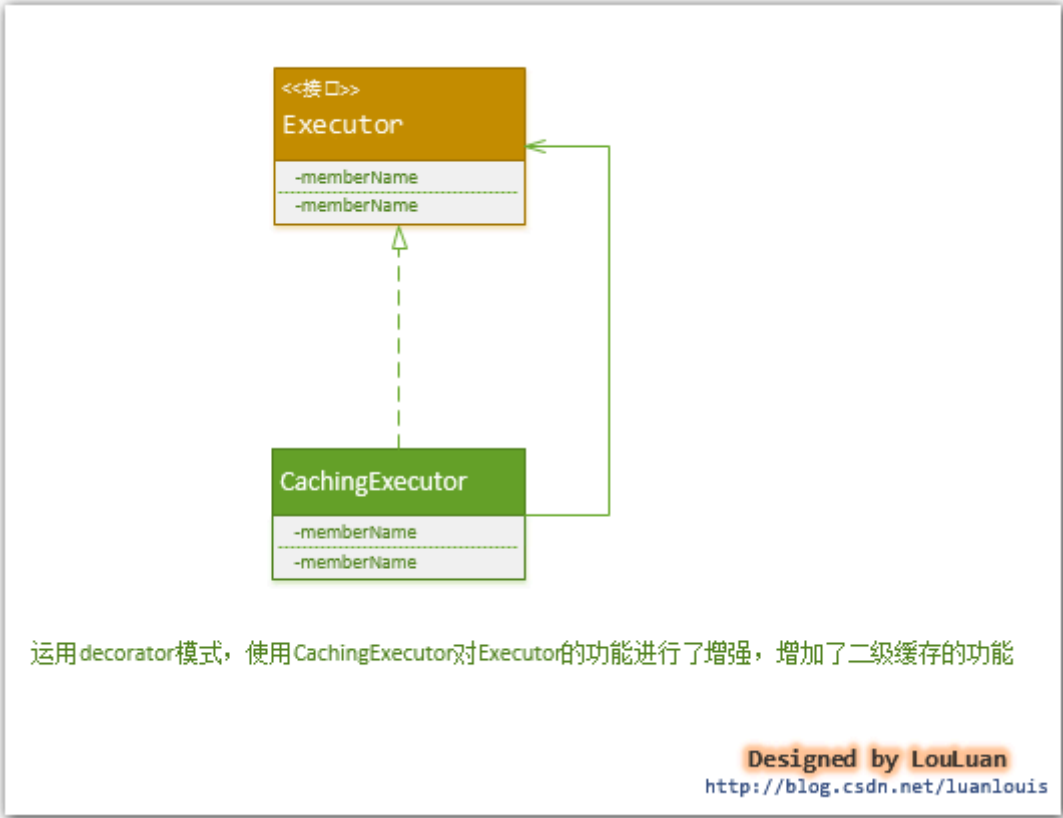


如上图所示，当开一个会话时，一个**SqlSession**对象会使用一个**Executor**对象来完成会话操作，**MyBatis**的二级缓存机制的关键就是对这个**Executor**对象做文章。如果用户配置了"**cacheEnabled=true**"，那么**MyBatis**在为**SqlSession**对象创建**Executor**对象时，会对**Executor**对象加上一个装饰者：**CachingExecutor**，这时**SqlSession**使用**CachingExecutor**对象来完成操作请求。**CachingExecutor**对于查询请求，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：



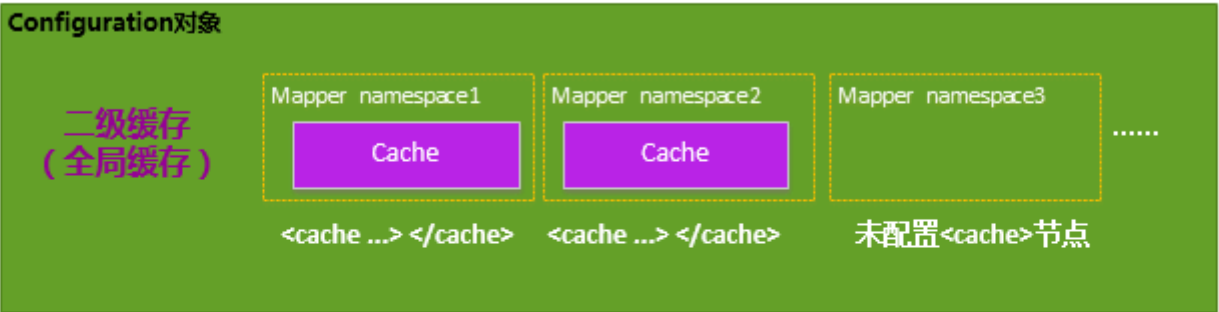
2 . MyBatis二级缓存的划分

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）；

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

MyBatis将Application级别的二级缓存细分到Mapper级别，即对于每一个Mapper.xml,如果在其中使用了<cache> 节点，则MyBatis会为此Mapper创建一个Cache缓存对象，如下图所示：



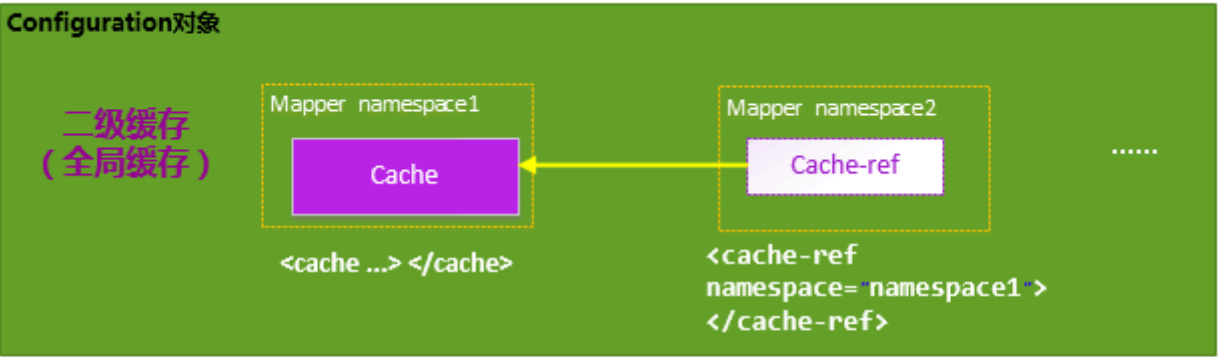
对于namespace1和namesapce2两个Mapper,都配置了<cache>会分别为这两个Mapper根据配置信息创建一个Cache对象；而namespace3的Mapper，没有配置<cache>节点，则对于Mapper namespace3就没有对应的二级缓存

Designed by LouLuan
<http://blog.csdn.net/luanlouis>

注： 上述的每一个Cache对象，都会有一个自己所属的namespace命名空间，并且会将Mapper的 namespace作为它们的ID；

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。



Mapper namespace2 的<cache-ref>定义的namespace=" namespace1"，说明Mapper namespace2 将使用Mapper namespace1中的Cache缓存对象，这时候要求namespace1中一定要有<cache>节点的定义

Designed by LouLuan
<http://blog.csdn.net/luanlouis>

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。

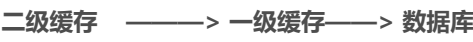
```
<select id="selectByMinSalary" resultMap="BaseResultMap" parameterType="java.util.Map" useCache="true">
```

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：



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

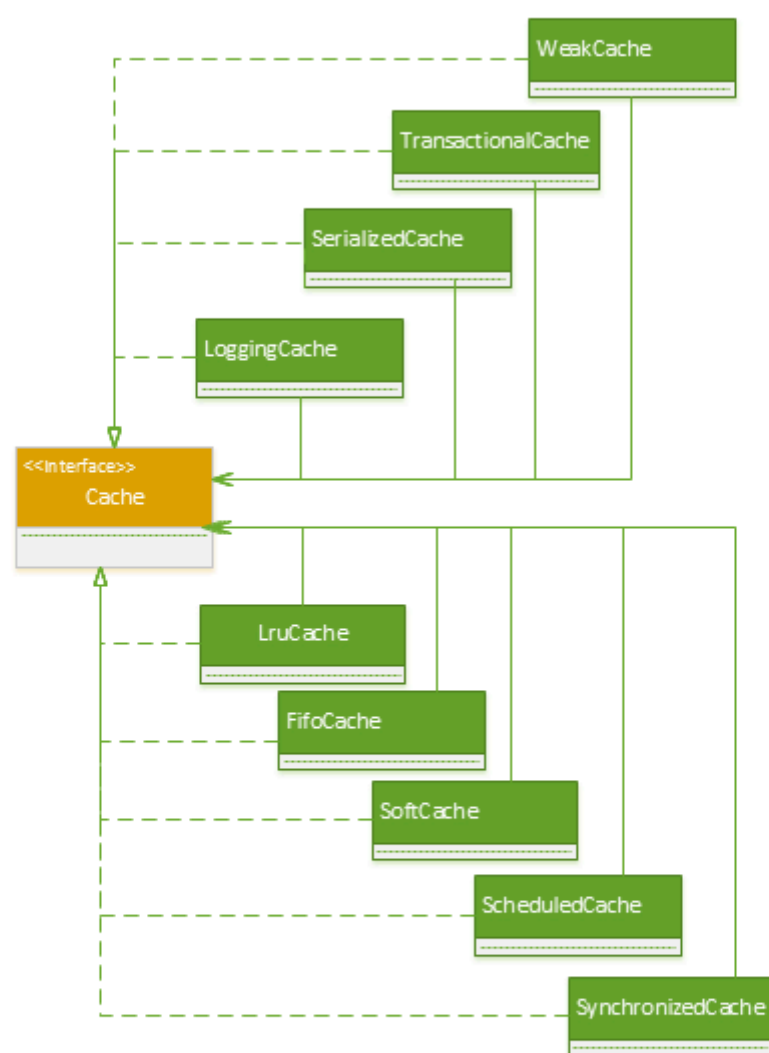
MyBatis自身提供了丰富的，并且功能强大的二级缓存的实现，它拥有一系列的Cache接口装饰者，可以满足各种对缓存操作和更新的策略。

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；

org.apache.ibatis.cache.decorators



Designed by LouLuan
<http://blog.csdn.net/luanlouis>

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。