本文内容出自：https://github.com/gzc426/Java-Interview

以后有更新内容，会在 github 更新

加入**互联网 IT** 求职、技术交流、资料共享 QQ 群 **691206173**，3T 编程资料等你来拿，

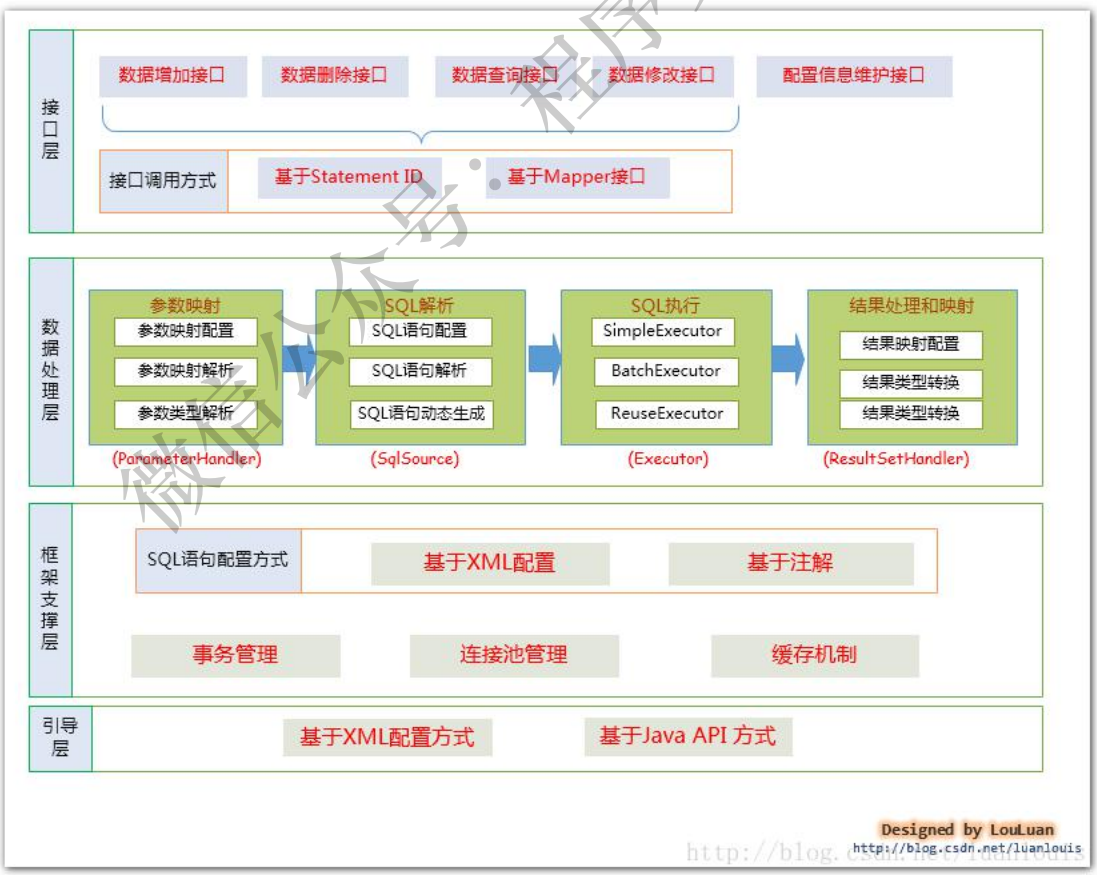
群主（群号：**691206173**）：本硕就读于**哈尔滨工业大学**，计算机专业，2019 硕士毕业，已拿**百度** java 后台开发 offer，另外还有去哪儿，华为，茄子快传，vipkid,秒针，创新工厂一点资讯这些公司的 offer。公众号中的文章有分享群主找工作的经验，java 学习/C++学习/机器学习/前端的指导路线，以及未来在百度的学习成长之路，满满都是干货，除了干货分享还有 3T 编程资料**（java/C++/算法/php/机器学习/大数据/人工智能/面试等）**等你来拿，另外还有微信交流群以及群主的**个人微信（抽空提供一对一指导意见）,当然也希望你能帮忙在朋友圈转发推广一下**

公众号

MyBatis

让你自己实现一个 orm 框架会如何实现
MyBatis/Hibernate 原理，源码，区别；批量操作；MyBatis 缓存（一级、二级）； mybatis 的#和$号区别 ；mybatis 一级缓存及可能存在的问题，两个机器能否共用同一个 SqlSession 实现一级缓存；mybatis 如何映射表结构
在 Hibernate 中 java 的对象状态有哪些；hibernate 主键生成策略
mybatis 和 hibernate 各自的缓存原理和比较，hibernate 的一级二级和查询缓存，还有针对缓存的 miss 率，置换策略，容量设置和性能的平衡
mybatis 和 ibatis 的区别（配置文件格式）

架构设计



接口层：
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

数据处理层：
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>

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。

OGNL 是 Object-Graph Navigation Language 的缩写，它是一种功能强大的表达式语言，通过它简单一致的表达式语法，可以存取对象的任意属性，调用对象的方法，遍历整个对象的结构图，实现字段类型转化等功能。它使用相同的表达式去存取对象的属性。

OGNL（ Object Graph Navigation Language ）对象图导航语言，这是一种强大的表达式语言，通过它可以非常方便的来操作对象属性。类似于我们的EL，SpEL等

访问对象属性：	person.name
调用方法：	person.getName()
调用静态属性/方法：	@java.lang.Math@PI @java.util.UUID@randomUUID()
调用构造方法：	new com.atguigu.bean.Person('admin').name
运算符：	+,-*,/,%
逻辑运算符：	in,not in,>,>=,<,<=,==,!=

注意：xml中特殊符号如",>,<等这些都需要使用转义字符

访问集合伪属性：

类型	伪属性	伪属性对应的 Java 方法
List、Set、Map	size、isEmpty	List/Set/Map.size(),List/Set/Map.isEmpty()
List、Set	iterator	List.iterator()、Set.iterator()
Map	keys、values	Map.keySet()、Map.values()
Iterator	next、hasNext	Iterator.next()、Iterator.hasNext()

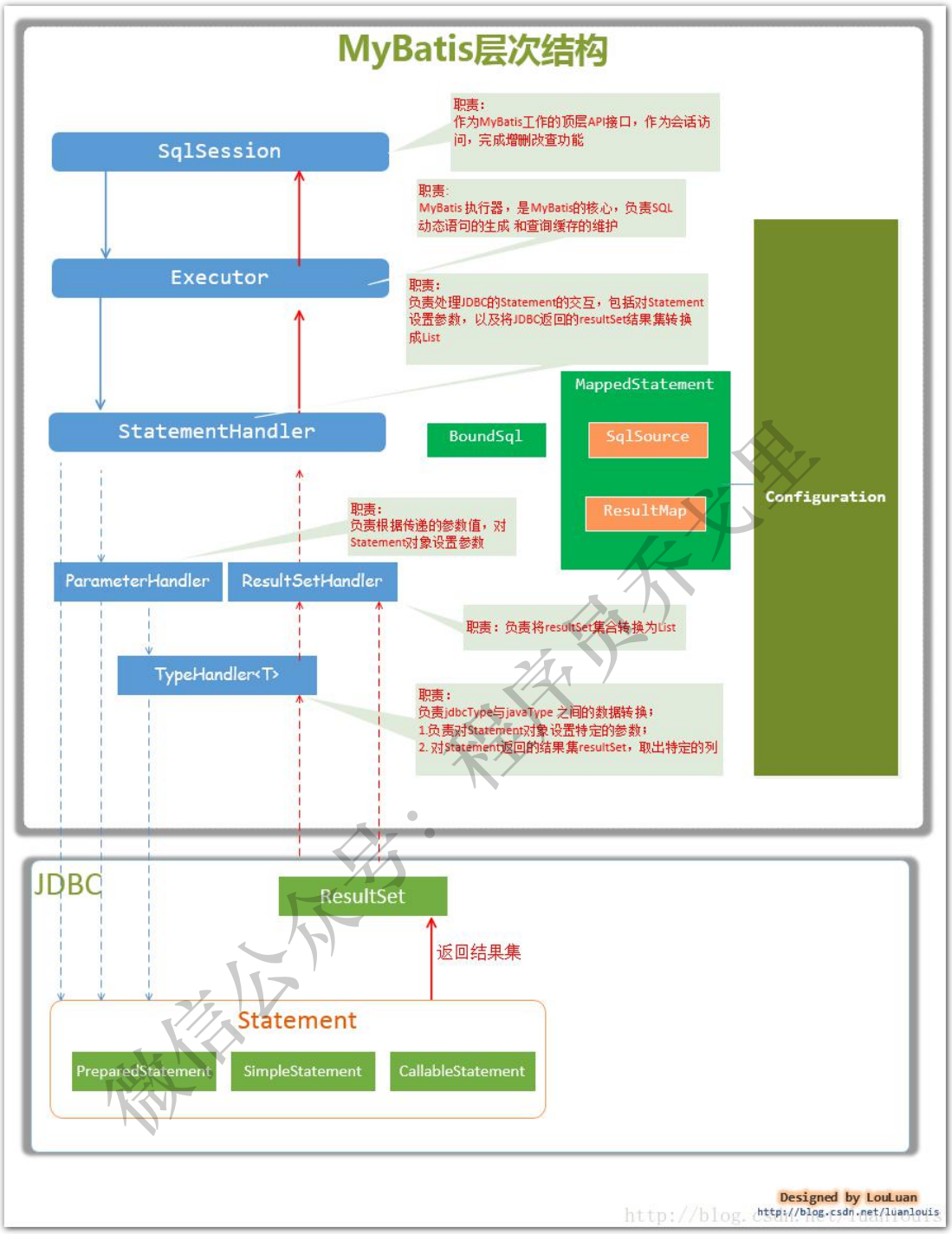
框架支撑层：
事务管理；连接池管理；缓存等

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。



初始化过程

MyBatis 初始化的过程，就是创建 Configuration 对象的过程。

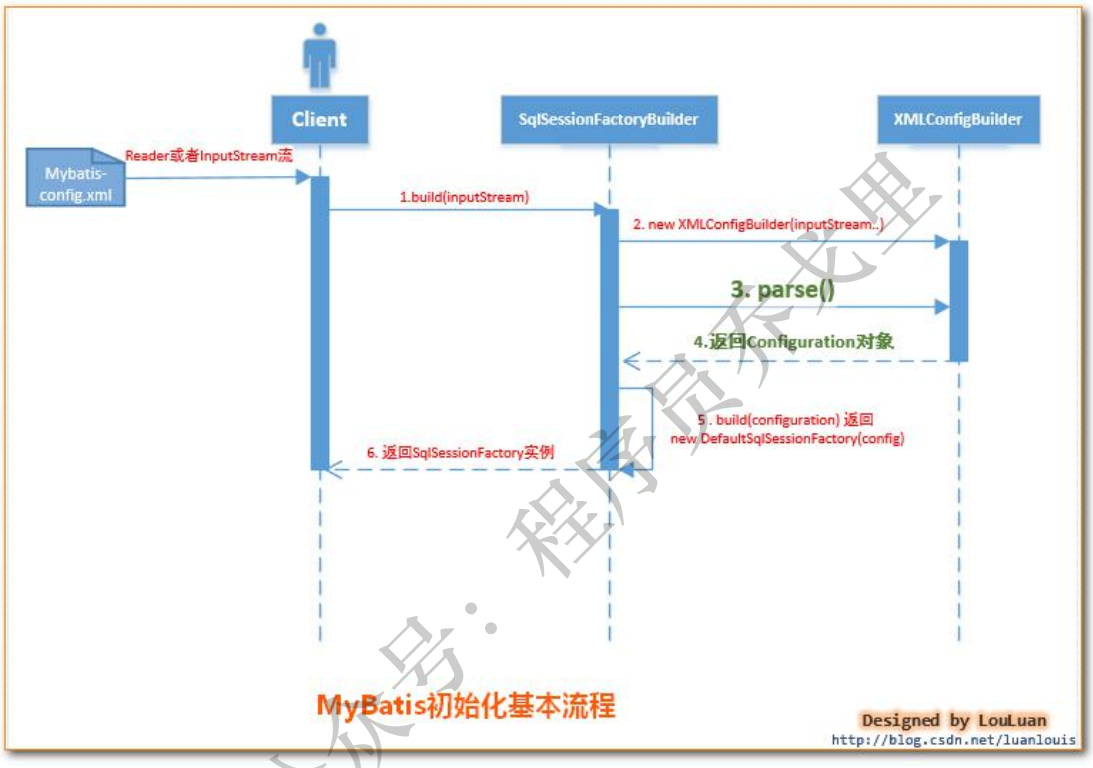
MyBatis 的初始化可以有两种方式：

基于 XML 配置文件：基于 XML 配置文件的方式是将 MyBatis 的所有配置信息放在 XML 文

件中，MyBatis 通过加载并 XML 配置文件，将配置文信息组装成内部的 Configuration 对象
基于 Java API：这种方式不使用 XML 配置文件，需要 MyBatis 使用者在 Java 代码中，手动创建 Configuration 对象，然后将配置参数 set 进入 Configuration 对象中。

mybatis 初始化 -->创建 SqlSession -->执行 SQL 语句

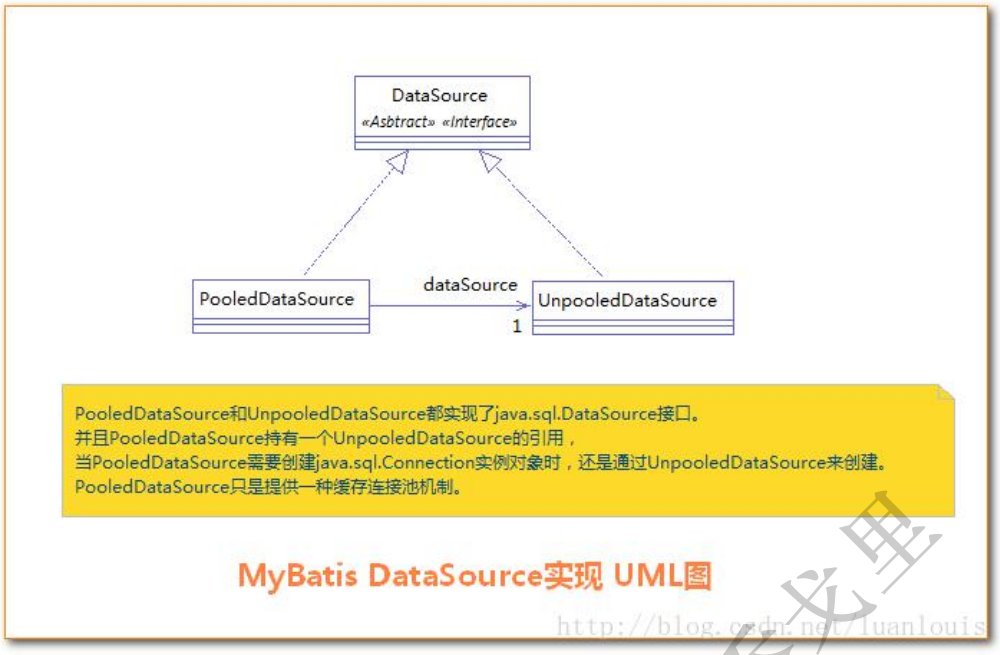
SqlSessionFactoryBuilder 根据传入的数据流生成 Configuration 对象，然后根据 Configuration 对象创建默认的 SqlSessionFactory 实例。



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。

数据源与连接池

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



DataSource 创建

```
<dataSource type="POOLED">
  <property name="driver" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</dataSource>
```

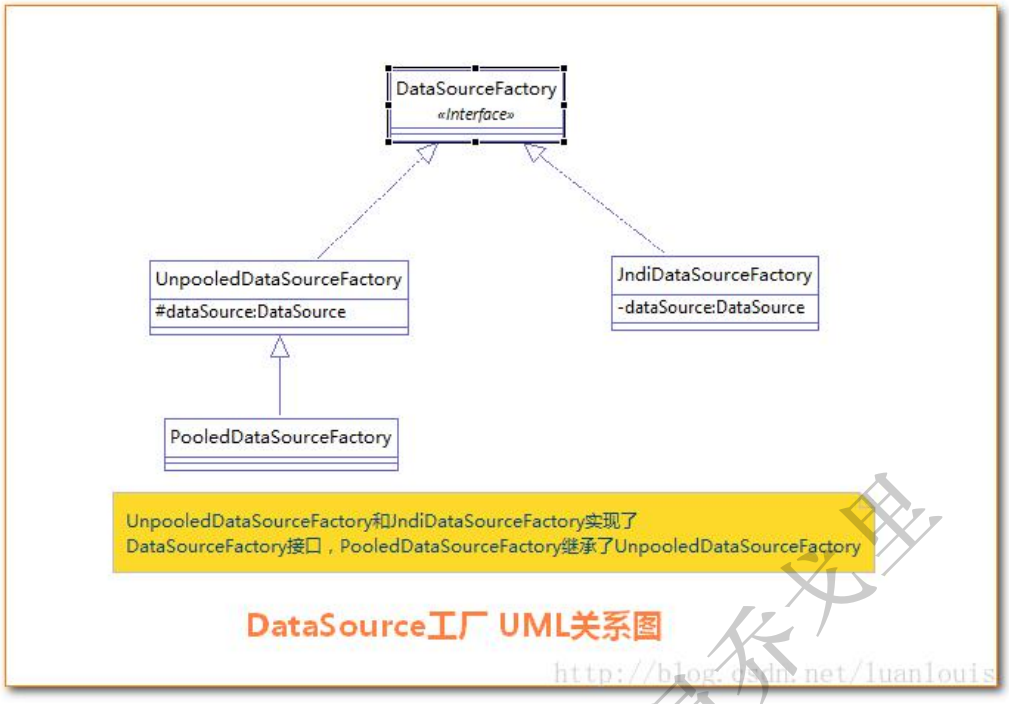
MyBatis 是通过工厂模式来创建数据源 DataSource 对象的，MyBatis 定义了抽象的工厂接口:org.apache.ibatis.datasource.DataSourceFactory,通过其 getDataSource()方法返回数据源 DataSource。

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



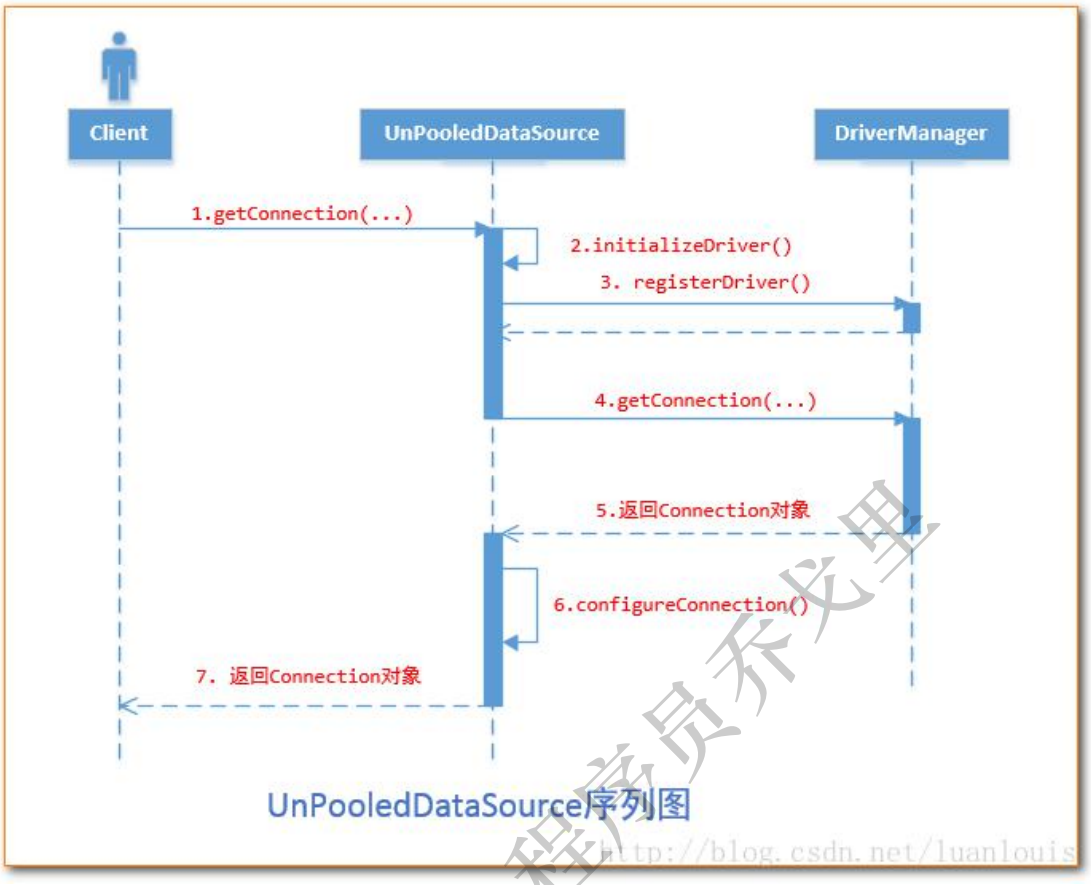
Connection 创建

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。

Unpooled

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 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。



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。

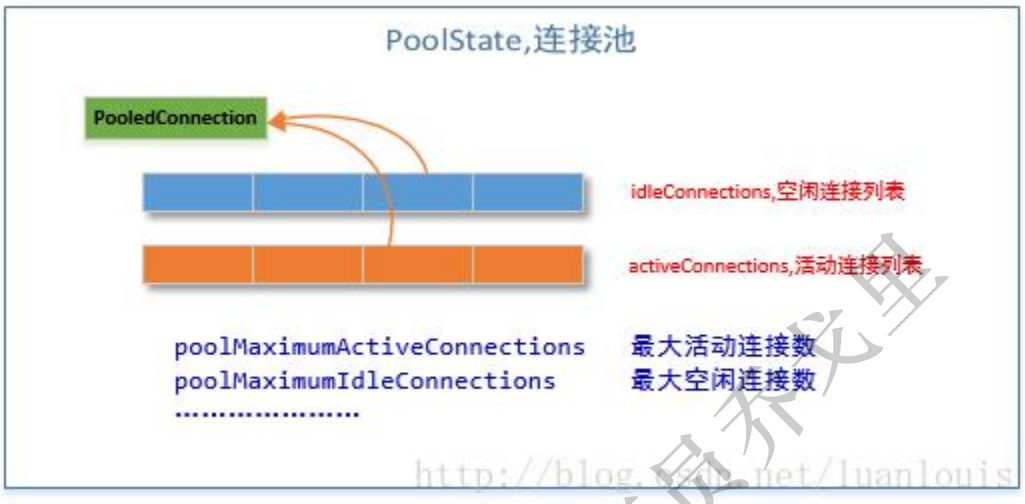
对于需要频繁地跟数据库交互的应用程序，可以在创建了 Connection 对象，并操作完数据库后，可以不释放掉资源，而是将它放到内存中，当下次需要操作数据库时，可以直接从内存中取出 Connection 对象，不需要再创建了，这样就极大地节省了创建 Connection 对象的资源消耗。由于内存也是有限和宝贵的，这又对我们对内存中的 Connection 对象怎么有效地维护提出了很高的要求。我们将在内存中存放 Connection 对象的容器称之为 连接池（Connection Pool）。

Pooled

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

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

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



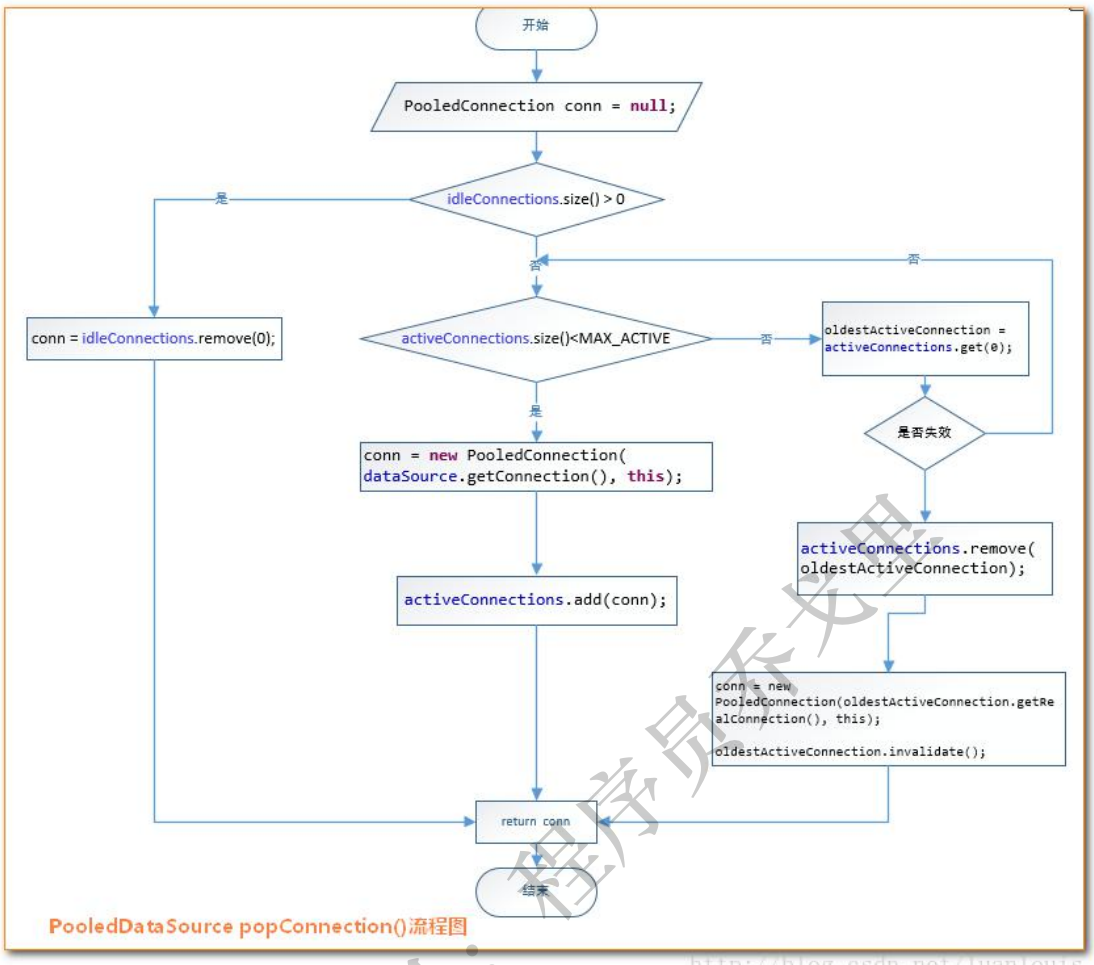
popConnection()方法到底做了什么：

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



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。

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。

这里要使用代理模式，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。

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。

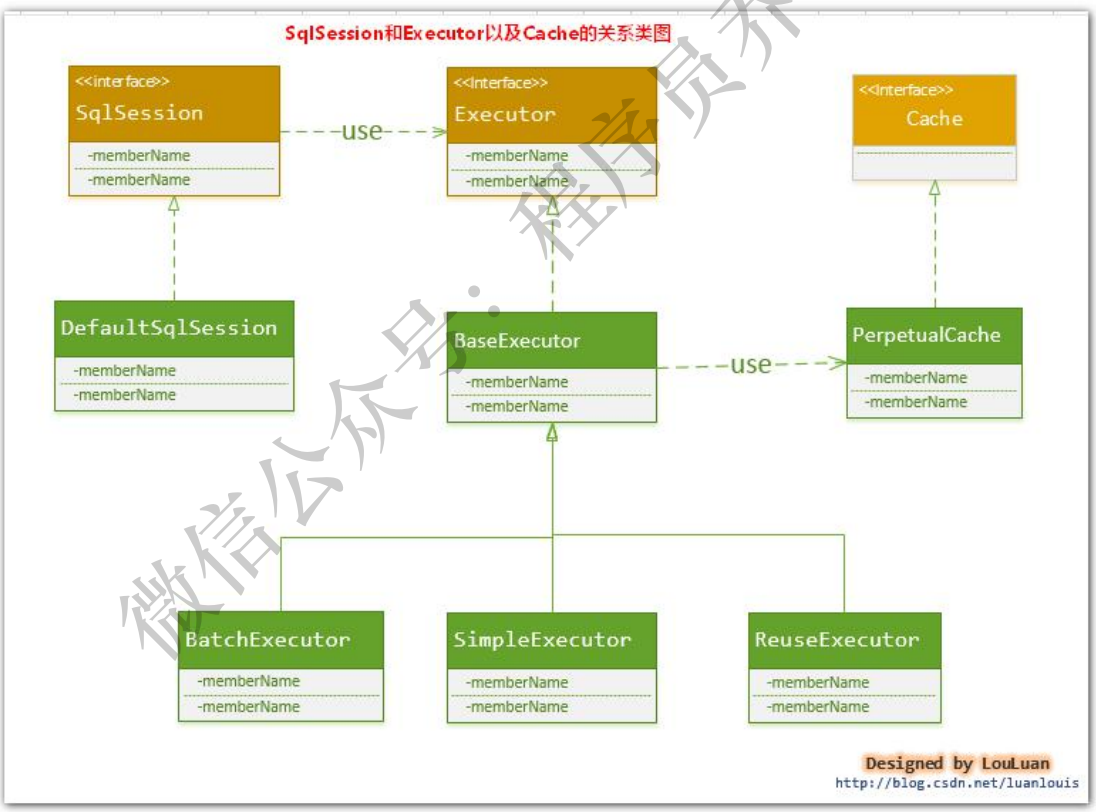
一级缓存（Session 级别的缓存）

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。

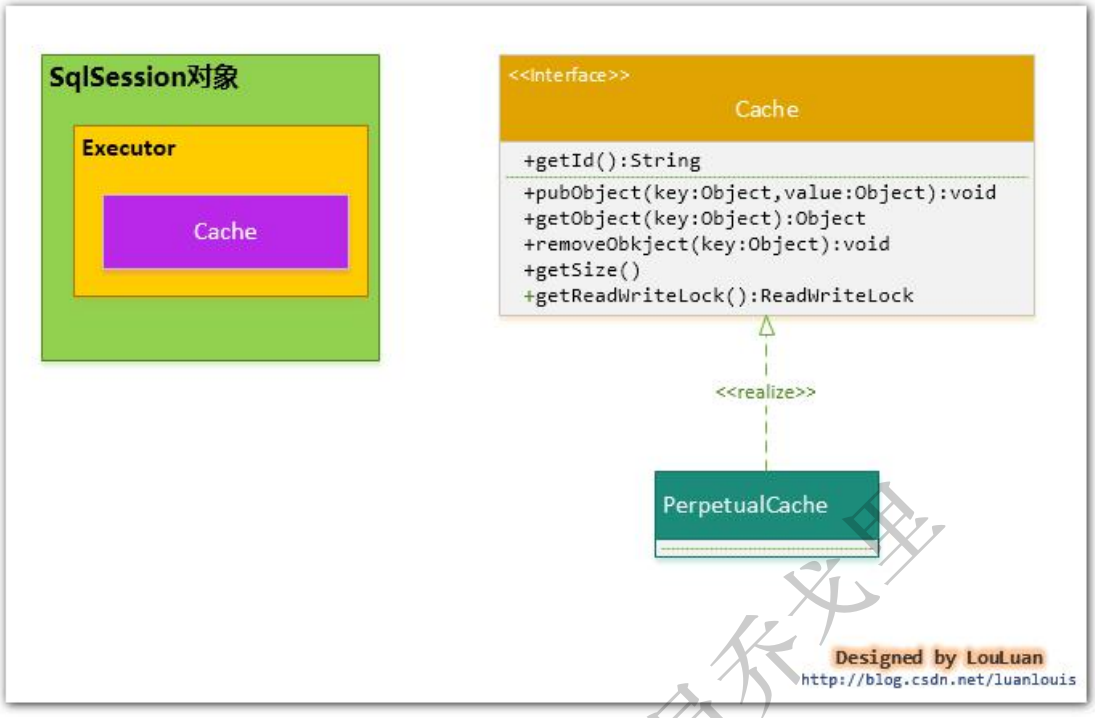
在对数据库的一次会话中，我们有可能会反复地执行完全相同的查询语句，如果不采取一些措施的话，每一次查询都会查询一次数据库,而我们在极短的时间内做了完全相同的查询，那么它们的结果极有可能完全相同，由于查询一次数据库的代价很大，这有可能造成很大的资源浪费。

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。

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：



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。



Perpetual Cache（永久的）

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。

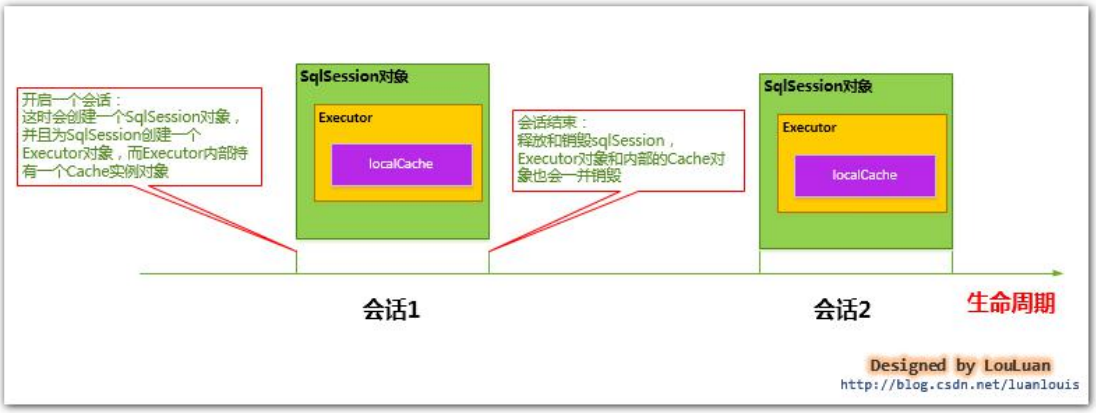
生命周期

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；



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；

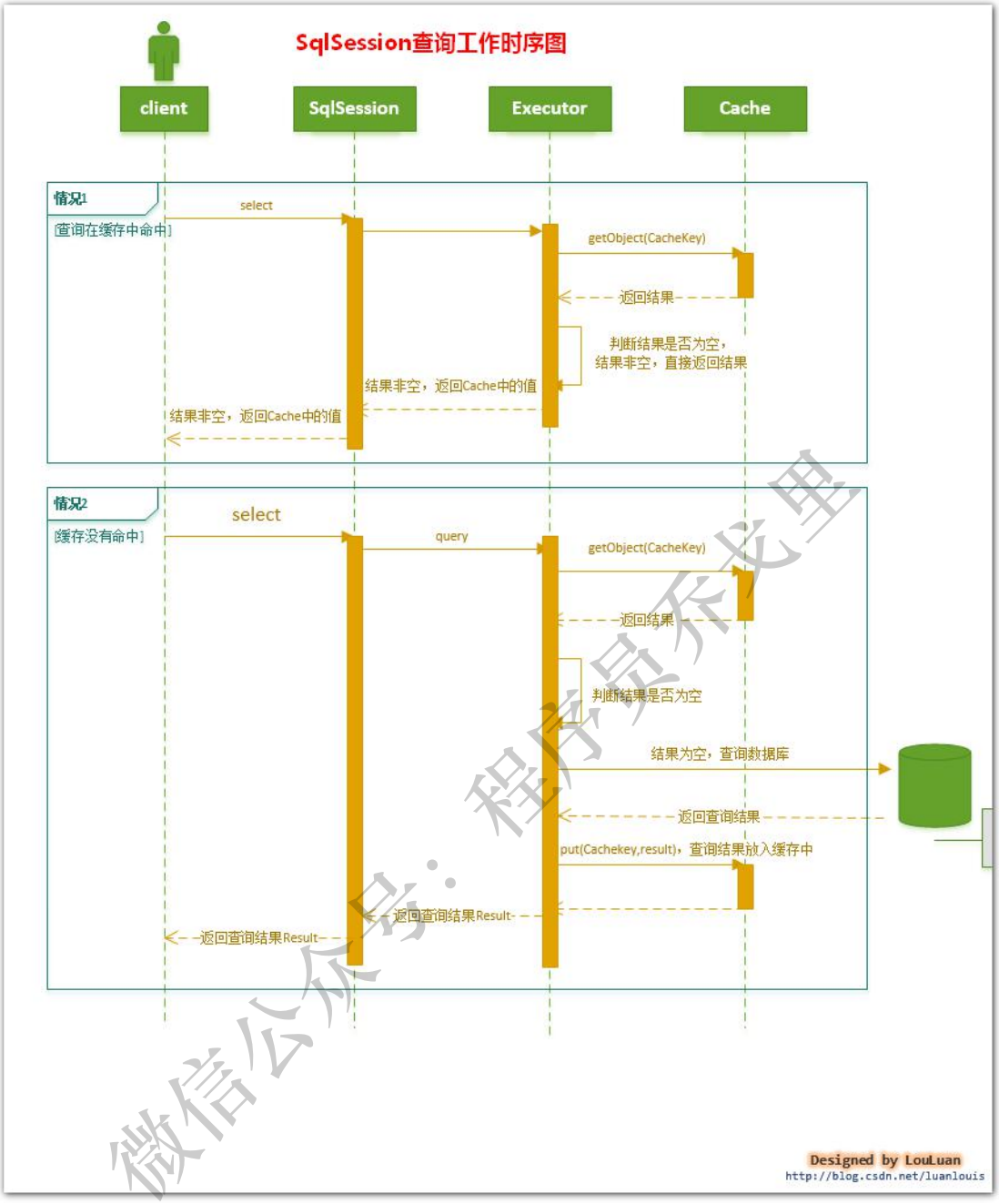
3. 如果命中，则直接将缓存结果返回；

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；



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：

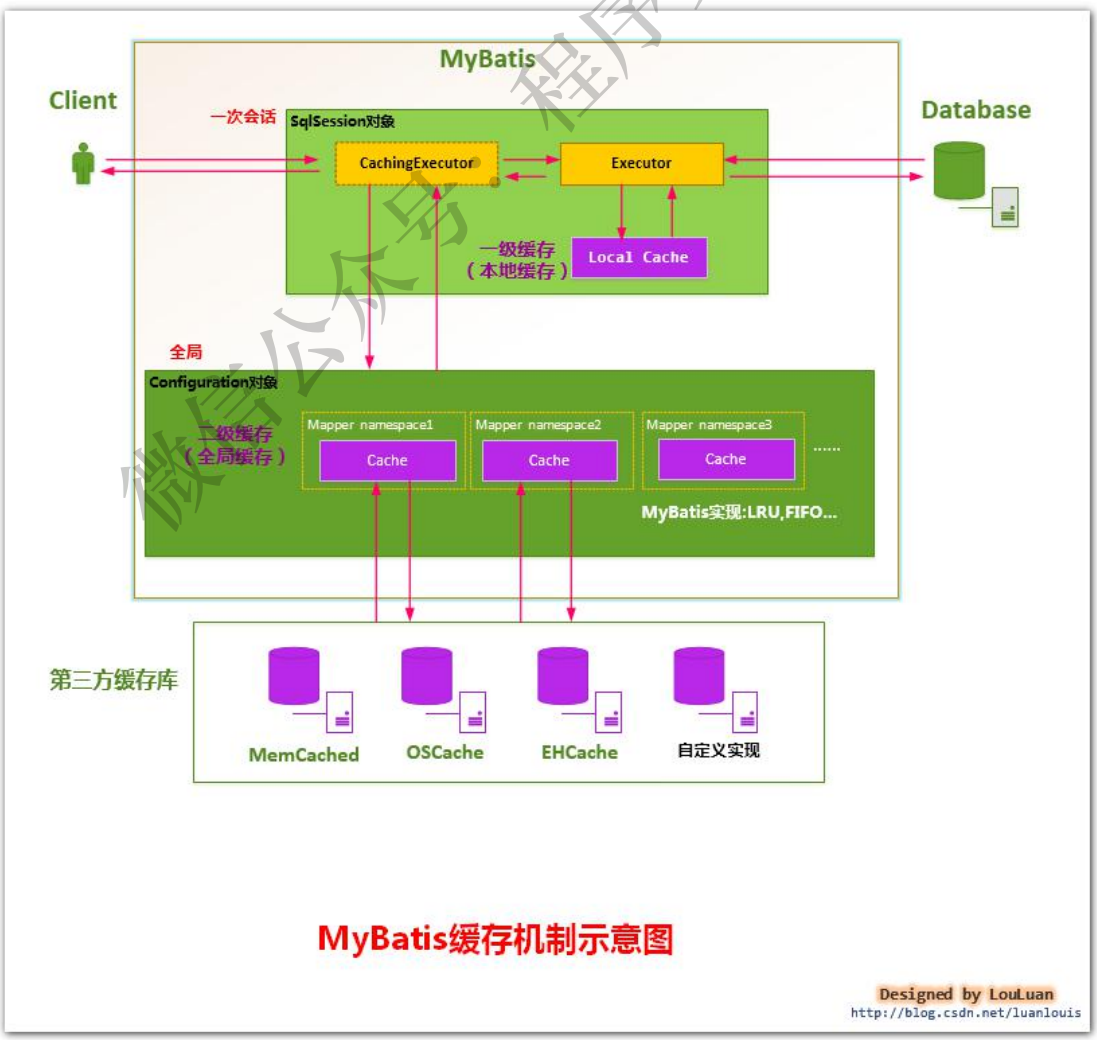
1. 传入的 statementId
2. 查询时要求的结果集中的结果范围（结果的范围通过 rowBounds.offset 和 rowBounds.limit 表示）；
3. 这次查询所产生的最终要传递给 JDBC java.sql.Preparedstatement 的 Sql 语句字符串（boundSql.getSql()）
4. 传递给 java.sql.Statement 要设置的参数值

CacheKey 由以下条件决定：statementId + rowBounds + 传递给 JDBC 的 SQL + 传递给 JDBC 的参数值

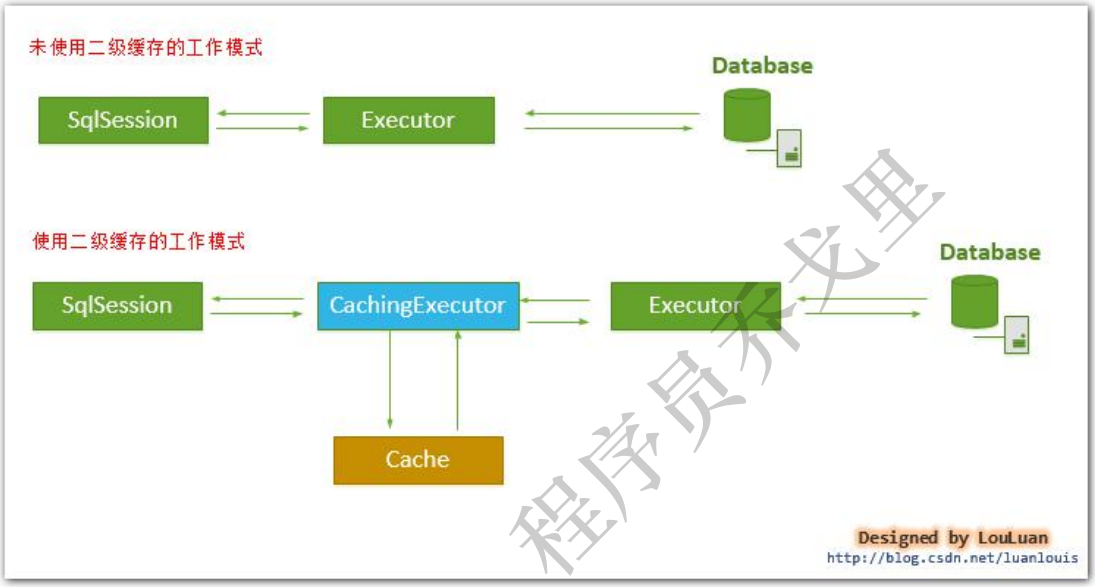
问题：
如果我一直使用某一个 SqlSession 对象查询数据，这样会不会导致 HashMap 太大，而导致 java.lang.OutOfMemoryError 错误啊？ 读者这么考虑也不无道理，不过 MyBatis 的确是这样设计的。
MyBatis 这样设计也有它自己的理由：
a. 一般而言 SqlSession 的生存时间很短。一般情况下使用一个 SqlSession 对象执行的操作不会太多，执行完就会消亡；
b. 对于某一个 SqlSession 对象而言，只要执行 update 操作（update、insert、delete），都会将这个 SqlSession 对象中对应的一级缓存清空掉，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。

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

二级缓存（Mapper 级别的缓存）



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。

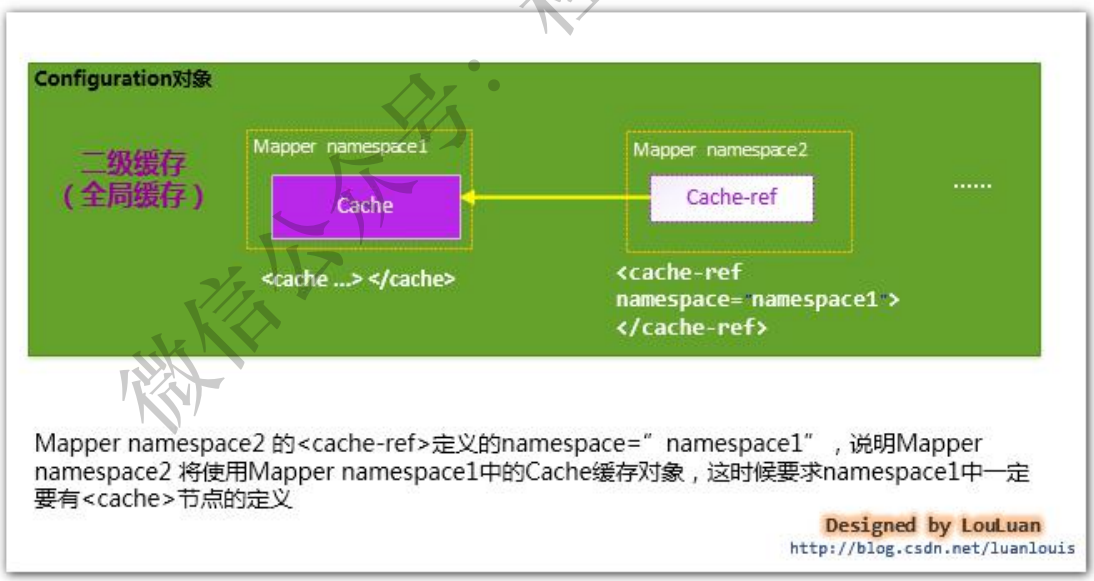
分类

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）；

对于每一个 Mapper.xml,如果在其中使用了<cache> 节点，则 MyBatis 会为这个 Mapper 创建一个 Cache 缓存对象，如下图所示：

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；

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。



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。
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

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

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

二级缓存　———> 一级缓存——> 数据库

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；

#和$号区别

#是一个占位符，接收输入参数，类型可以是基本数据类型、POJO 类型、Map 类型
会转为 JDBC 的？，是预编译的形式 PreparedStatement

$是一个拼接符，会引起 SQL 注入，所以不建议使用，而#不会引起。
可以接收基本数据类型、POJO 类型、Map 类型。
直接拼接在 SQL 语句中

大部分情况下都应该使用#{}

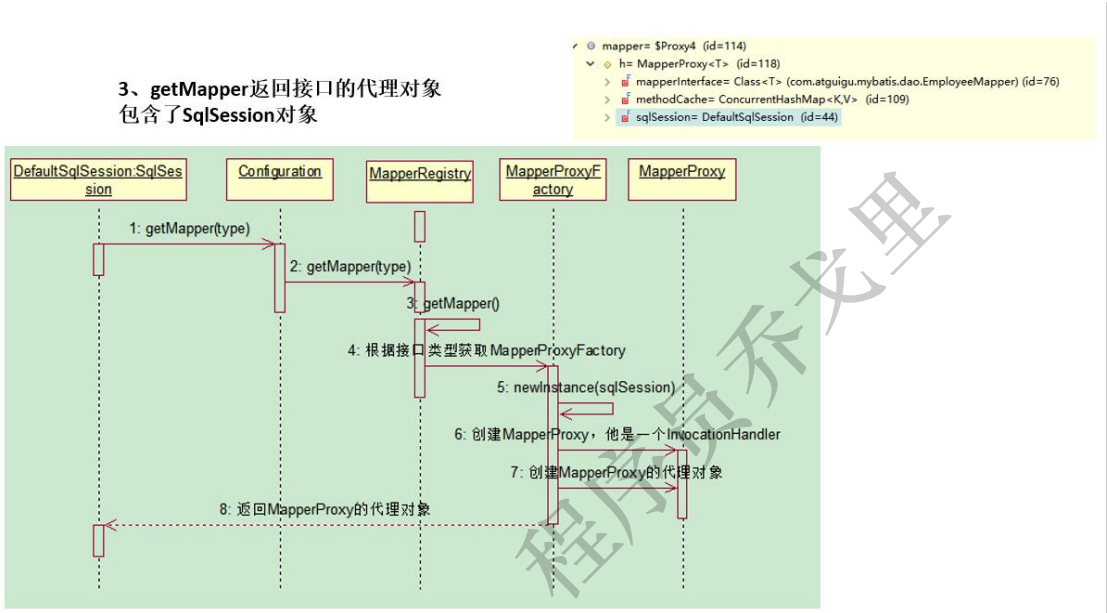
以下情况使用${}

比如分表：财务表每年一张表
此时表名可以拼接出来，但无法使用占位符填充
select * from ${year}_salary where …
2016_salary

比如排序：按某个字段排序，升降序也需要使用${}来指定，因为 SQL 是不支持将排序填充进去的，必须一开始就指定（同表名）
select * .. from .. order by xxx xxx

接口代理对象的创建

openSession.getMapper(Mapper.class)
会调用 Configuration 的 getMapper 的方法，它又调用 MapperRegistery 的 getMapper 的方法。
它会获取接口的代理对象 MapperProxy（实现了 JDK 的 InvocationHandler 接口）。
持有 DefaultSqlSession 对象（它又持有 Executor），可以进行增改删查操作。



执行方法（以查询为例）

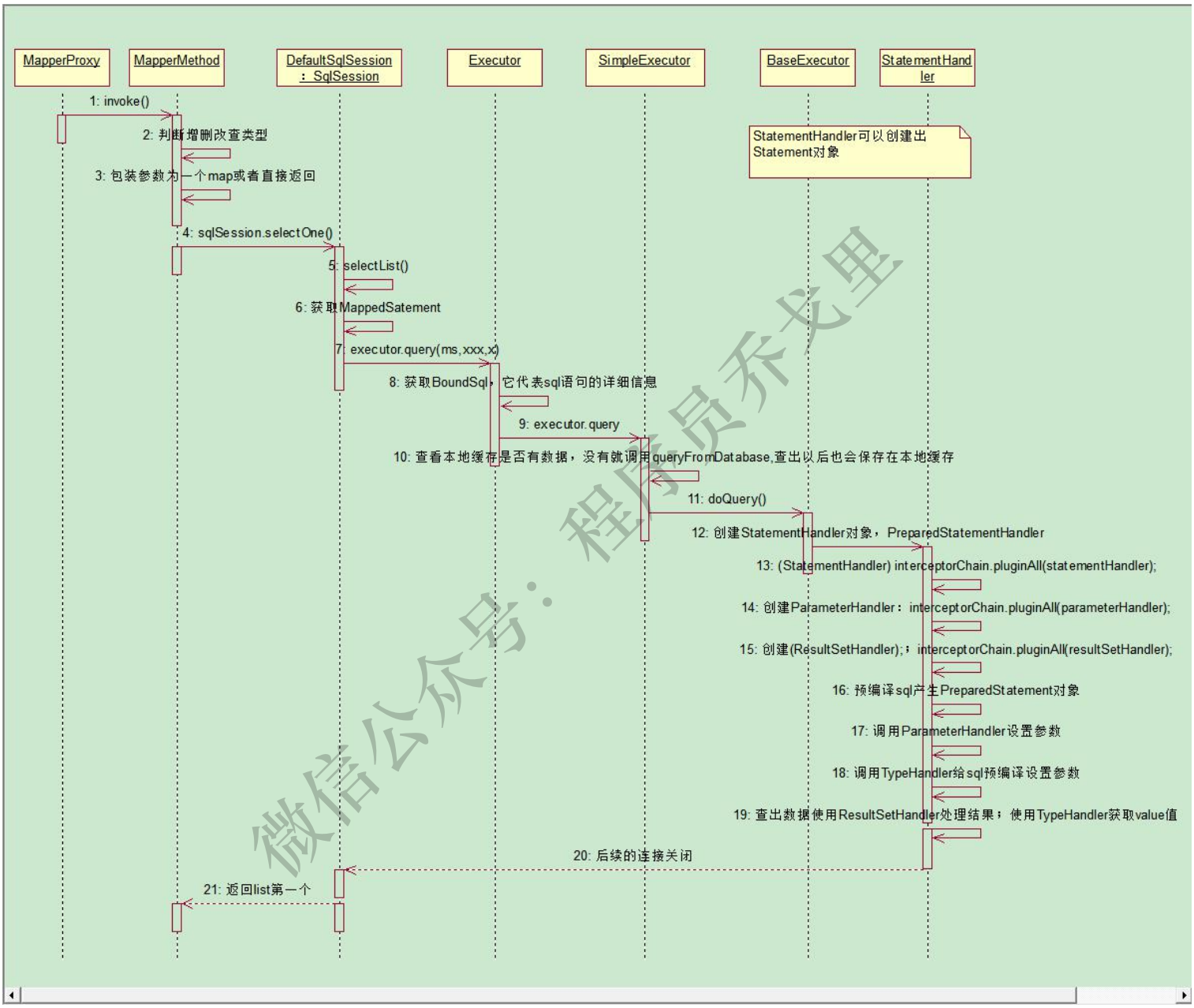
调用 find 方法后，会被 invoke 方法拦截。
如果是 Object 类的方法，那么直接执行；如果不是，那么会把 method 对象包装为 MapperMethod，然后执行该方法（传入 SqlSession）。
执行时，会判断是增改删查的哪一种方法，然后**解析入口参数**（依赖于 paramNameResolver），封装为一个 Map。
然后调用 sqlSession 的 selectOne 方法
在 sqlSession 中仍会转到 selectList 方法，只是返回结果集的第一个对象。
selectList 中会从 Configuration 中取出对应的 MappedStatement（封装了 SQL 语句），然后把这个 MappedStatement 交给 Executor，由 Executor 来执行真正的查询（query 方法）。
在 query 方法中调用 getBoundSql（封装了 SQL 的信息），创建缓存的 key，有二级缓存则使用缓存，没有就去查找一级缓存，如果还没有，那么去查询数据库，查询后将查询结果放入一级缓存。
查询数据库调用的是 doQuery 方法，在这个方法中会使用 JDBC 的 stratement，在创建 statement 时会创建并传入 StatementHandler（可以创建 Statement 对象）。
StatementHandler 在创建时会读取 MapperStatement 中关于 statementType 的配置属性，根据该属性创建对应的 StatementHandler（Statement、Prepared、Callable），注意在创建时又会将 InterceptorChain 挂接到 StatementHandler 上，同时会创建 ParameterHandler/ResultSetHandler，还会将 InterceptorChain 挂接到 ParameterHandler/

ResultSetHandler 上。

创建真正的 PreparedStatement 时会预编译 SQL，此时会调用 ParameterHandler 的设置参数，设置参数时会调用 TypeHandler 的 setParameter 方法。

查询完后，会使用 ResultSetHandler 来封装查询结果（其中又会调用 TypeHandler）。

最后会执行清理工作。



MyBatis 与 Hibernate 比较

第一方面：开发速度的对比

就开发速度而言，Hibernate 的真正掌握要比 Mybatis 来得难些。Mybatis 框架相对简单很容易上手，但也相对简陋些。个人觉得要用好 Mybatis 还是首先要先理解好 Hibernate。

比起两者的开发速度，不仅仅要考虑到两者的特性及性能，更要根据项目需求去考虑究竟哪一个更适合项目开发，比如：一个项目中用到的复杂查询基本没有，就是简单的增删改查，这样选择 hibernate 效率就很快了，因为基本的 sql 语句已经被封装好了，根本不需要你去写 sql 语句，这就节省了大量的时间，但是对于一个大型项目，复杂语句较多，这样再去选择 hibernate 就不是一个太好的选择，选择 mybatis 就会加快许多，而且语句的管理也比较方便。

第二方面：开发工作量的对比

Hibernate 和 MyBatis 都有相应的代码生成工具。可以生成简单基本的 DAO 层方法。针对高级查询，Mybatis 需要手动编写 SQL 语句，以及 ResultMap。而 Hibernate 有良好的映射机制，开发者无需关心 SQL 的生成与结果映射，可以更专注于业务流程。

第三方面：sql 优化方面

Hibernate 的查询会将表中的所有字段查询出来，这一点会有性能消耗。Hibernate 也可以自己写 SQL 来指定需要查询的字段，但这样就破坏了 Hibernate 开发的简洁性。而 Mybatis 的 SQL 是手动编写的，所以可以按需求指定查询的字段。

Hibernate HQL 语句的调优需要将 SQL 打印出来，而 Hibernate 的 SQL 被很多人嫌弃因为太丑了。MyBatis 的 SQL 是自己手动写的所以调整方便。但 Hibernate 具有自己的日志统计。Mybatis 本身不带日志统计，使用 Log4j 进行日志记录。

第四方面：对象管理的对比

Hibernate 是完整的对象/关系映射解决方案，它提供了对象状态管理（state management）的功能，使开发者不再需要理会底层数据库系统的细节。也就是说，相对于常见的 JDBC/SQL 持久层方案中需要管理 SQL 语句，Hibernate 采用了更自然的面向对象的视角来持久化 Java 应用中的数据。

换句话说，使用 Hibernate 的开发者应该总是关注对象的状态（state），不必考虑 SQL 语句的执行。这部分细节已经由 Hibernate 掌管妥当，只有开发者在进行系统性能调优的时候才需要进行了解。而 MyBatis 在这一块没有文档说明，用户需要对对象自己进行详细的管理。

第五方面：缓存机制

Hibernate 一级缓存是 Session 缓存，利用好一级缓存就需要对 Session 的生命周期进行管理

好。建议在一个 Action 操作中使用一个 Session。一级缓存需要对 Session 进行严格管理。

Hibernate 二级缓存是 SessionFactory 级的缓存。 SessionFactory 的缓存分为内置缓存和外置缓存。内置缓存中存放的是 SessionFactory 对象的一些集合属性包含的数据(映射元素据及预定 SQL 语句等),对于应用程序来说,它是只读的。外置缓存中存放的是数据库数据的副本,其作用和一级缓存类似.二级缓存除了以内存作为存储介质外,还可以选用硬盘等外部存储设备。二级缓存称为进程级缓存或 SessionFactory 级缓存，它可以被所有 session 共享，它的生命周期伴随着 SessionFactory 的生命周期存在和消亡。

Spring 对 SqlSession 的管理

开发的时候肯定会用到 Spring，也会用到 mybatis-spring 框架，在使用 MyBatis 与 Spring 集成的时候我们会用到了 SqlSessionTemplate 这个类，例如下边的配置，注入一个单例的 SqlSessionTemplate 对象。

SqlSessionTemplate 实现了 SqlSession 接口，也就是说我们可以使用 SqlSessionTemplate 来代理以往的 DefaultSqlSession 完成对数据库的操作，但是 DefaultSqlSession 这个类不是线程安全的，所以 DefaultSqlSession 这个类不可以被设置成单例模式的。

如果是常规开发模式的话，我们每次在使用 DefaultSqlSession 的时候都从 SqlSessionFactory 当中获取一个就可以了。但是与 Spring 集成以后，Spring 提供了一个全局唯一的 SqlSessionTemplate 对象来完成 DefaultSqlSession 的功能，问题就是：无论是多个 Dao 使用一个 SqlSessionTemplate，还是一个 Dao 使用一个 SqlSessionTemplate，SqlSessionTemplate 都是对应一个 sqlSession 对象，当多个 web 线程调用同一个 Dao 时，它们使用的是同一个 SqlSessionTemplate，也就是同一个 SqlSession，这可能存在着线程安全问题。

动态代理 SqlSessionFactory

```
public SqlSessionTemplate(SqlSessionFactory sqlSessionFactory, ExecutorType executorType,
                          PersistenceExceptionTranslator exceptionTranslator) {

    notNull(sqlSessionFactory, "Property 'sqlSessionFactory' is required");
    notNull(executorType, "Property 'executorType' is required");

    this.sqlSessionFactory = sqlSessionFactory;
    this.executorType = executorType;
    this.exceptionTranslator = exceptionTranslator;
    this.sqlSessionProxy = (SqlSession) newProxyInstance(
            SqlSessionFactory.class.getClassLoader(),
            new Class[]{SqlSession.class},
            new SqlSessionInterceptor());
}
```


SqlSessionTemplate 中使用的 SqlSessionFactory 是经过动态代理的，实现动态代理接口的是 SqlSessionInterceptor 类。

```
private class SqlSessionInterceptor implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        //获取SqlSession(这个SqlSession才是真正使用的，它不是线程安全的)
        //这个方法可以根据Spring的事物上下文来获取事物范围内的sqlSession
        SqlSession sqlSession = getSqlSession(
                SqlSessionTemplate.this.sqlSessionFactory,
                SqlSessionTemplate.this.executorType,
                SqlSessionTemplate.this.exceptionTranslator);
        try {
            //调用从Spring的事物上下文获取事物范围内的sqlSession对象
            Object result = method.invoke(sqlSession, args);
            //然后判断一下当前的sqlSession是否被Spring托管 如果未被Spring托管则自动commit
            if (!isSqlSessionTransactional(sqlSession,
                    SqlSessionTemplate.this.sqlSessionFactory)) {
                // force commit even on non-dirty sessions because some databases require
                // a commit/rollback before calling close()
                sqlSession.commit(true);
            }
            return result;
        } catch (Throwable t) {
            //如果出现异常则根据情况转换后抛出
            Throwable unwrapped = unwrapThrowable(t);
            if (SqlSessionTemplate.this.exceptionTranslator != null &&
                    unwrapped instanceof PersistenceException) {
                // release the connection to avoid a deadlock if the
                // translator is no loaded. See issue #22
                closeSqlSession(sqlSession, SqlSessionTemplate.this.sqlSessionFactory);
                sqlSession = null;
                Throwable translated = SqlSessionTemplate.this.exceptionTranslator.
                        translateExceptionIfPossible((PersistenceException) unwrapped);
                if (translated != null) {
                    unwrapped = translated;
                }
            }
            throw unwrapped;
        } finally {
```

注意 getSqlSession 和 closeSqlSession 方法。

getSqlSession 为：

其实本质上就是 ThreadLocal，每个线程有着自己对应的 SqlSession，不同线程间不会共用同一个 SqlSession。SqlSession 被包在 SqlSessionHolder，它使用了引用计数。关闭 session 时不是关闭 session，而是减少引用计数值。

```
public static SqlSession getSqlSession(SqlSessionFactory sessionFactory,
        ExecutorType executorType, PersistenceExceptionTranslator exceptionTranslator) {

    notNull(sessionFactory, NO_SQL_SESSION_FACTORY_SPECIFIED);
    notNull(executorType, NO_EXECUTOR_TYPE_SPECIFIED);

    //根据sqlSessionFactory从当前线程对应的资源map中获取SqlSessionHolder,
    // 当sqlSessionFactory创建了sqlSession,
    //就会在事务管理器中添加一对映射：key为sqlSessionFactory，value为SqlSessionHolder,
    // 该类保存sqlSession及执行方式
    SqlSessionHolder holder = (SqlSessionHolder) TransactionSynchronizationManager.
            getResource(sessionFactory);

    //从SqlSessionHolder中提取SqlSession对象
    SqlSession session = sessionHolder(executorType, holder);
    if (session != null) {
        return session;
    }

    if (LOGGER.isDebugEnabled()) {
        LOGGER.debug("Creating a new SqlSession");
    }
    //如果当前事物管理器中获取不到SqlSessionHolder对象就重新创建一个
    session = sessionFactory.openSession(executorType);

    //将新创建的SqlSessionHolder对象注册到TransactionSynchronizationManager中
    registerSessionHolder(sessionFactory, executorType, exceptionTranslator, session);

    return session;
}
```

```
public abstract class TransactionSynchronizationManager {

   private static final Log logger =
LogFactory.getLog(TransactionSynchronizationManager.class);

   private static final ThreadLocal<Map<Object, Object>> resources =
         new NamedThreadLocal<Map<Object, Object>>("Transactional
resources");
}
```

closeSqlSession 为

```
public static void closeSqlSession(SqlSession session, SqlSessionFactory sessionFactory) {
        //其实下面就是判断session是否被Spring事务管理，如果管理就会得到holder
        SqlSessionHolder holder = (SqlSessionHolder)
        TransactionSynchronizationManager.getResource(sessionFactory);
        if ((holder != null) && (holder.getSqlSession() == session)) {
            //这里释放的作用，不是关闭，只是减少一下引用数，因为后面可能会被复用
            holder.released();
        } else {
            //如果不是被spring管理，那么就不会被Spring去关闭回收，就需要自己close
            session.close();
        }
    }
```