# 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
* 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。
* 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

* 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 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；
  1. 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；
  2. 如果命中，则直接将缓存结果返回；
  3. 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（使用节点配置)；
* b.多个Mapper共用一个Cache缓存对象（使用节点配置)；
* 对于每一个Mapper.xml,如果在其中使用了 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,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 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 或节点，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 或节点，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；
  1. 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

* 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。
* 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 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