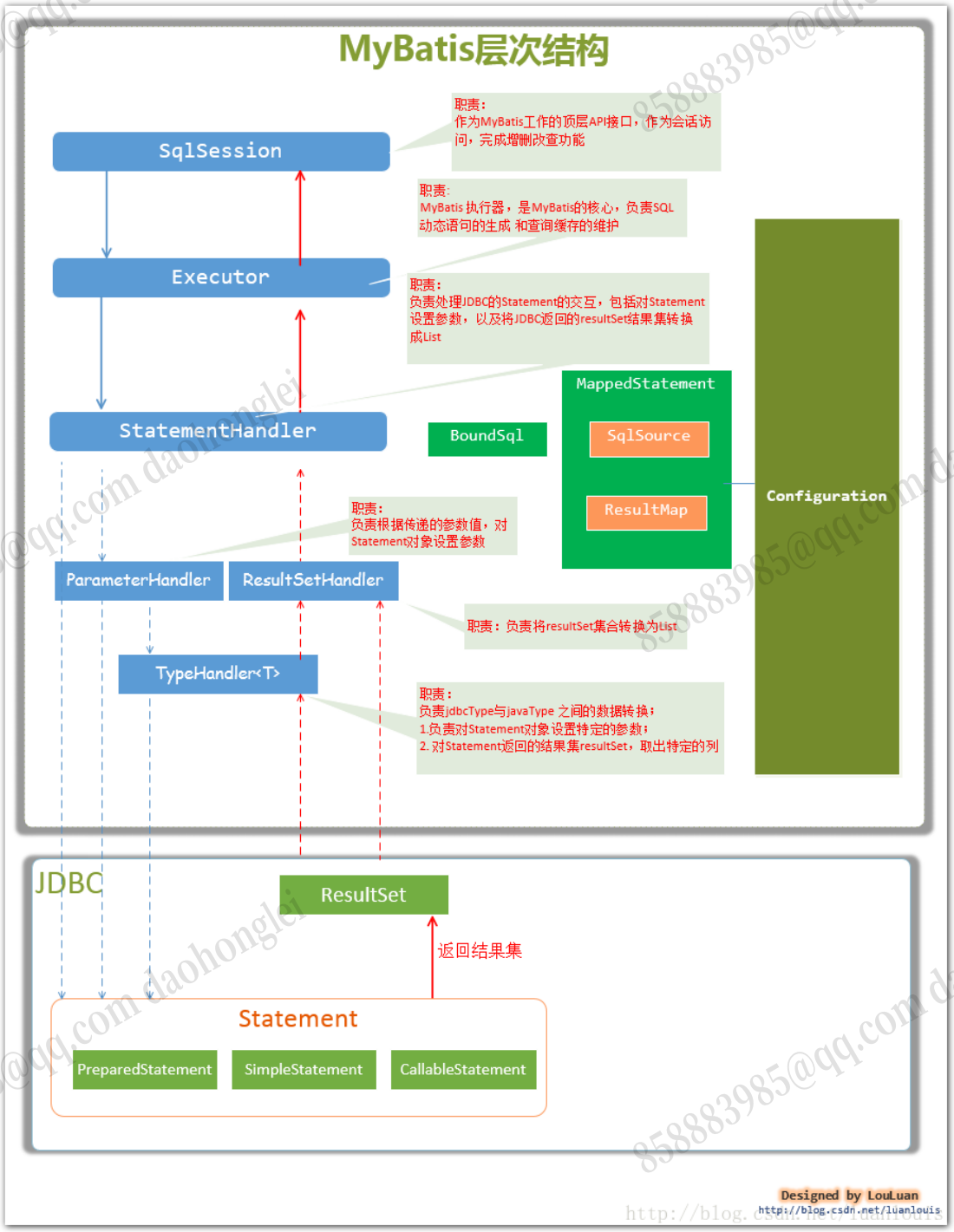
MyBatis 的主要构件及其相互关系

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。

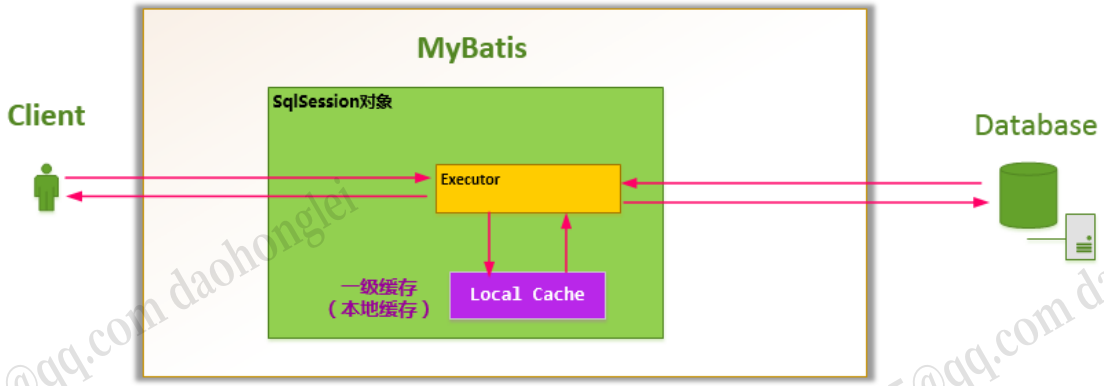
（注：这里只是列出了我个人认为属于核心的部件，请读者不要先入为主，认为 MyBatis 就只有这些部件哦！每个人对 MyBatis 的理解不同，分析出的结果自然会有所不同，欢迎读者提出质疑和不同的意见，我们共同探讨~）



Mybatis 一级缓存

Mybatis 对缓存提供支持，但是在没有配置的默认情况下，它只开启一级缓存，一级缓存只是相对于同一个 SqlSession 而言。所以在参数和 SQL 完全一样的情况下，我们使用同一个 SqlSession 对象调用一个 Mapper 方法，往往只执行一次 SQL，因为使用

SelSession 第一次查询后，MyBatis 会将其放在缓存中，以后再查询的时候，如果没有声明需要刷新，并且缓存没有超时的情况下，SqlSession 都会取出当前缓存的数据，而不会再次发送 SQL 到数据库。



MyBatis一级缓存简单示意图

为什么要使用一级缓存，不用多说也知道个大概。但是还有几个问题我们要注意一下。

1、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

2、怎么判断某两次查询是完全相同的查询?

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。

2.1 传入的 statementId

2.2 查询时要求的结果集中的结果范围

2.3. 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() ）

2.4 传递给 java.sql.Statement 要设置的参数值

3.一级缓存的作用

a.通过 association 和 collection 建立级联映射

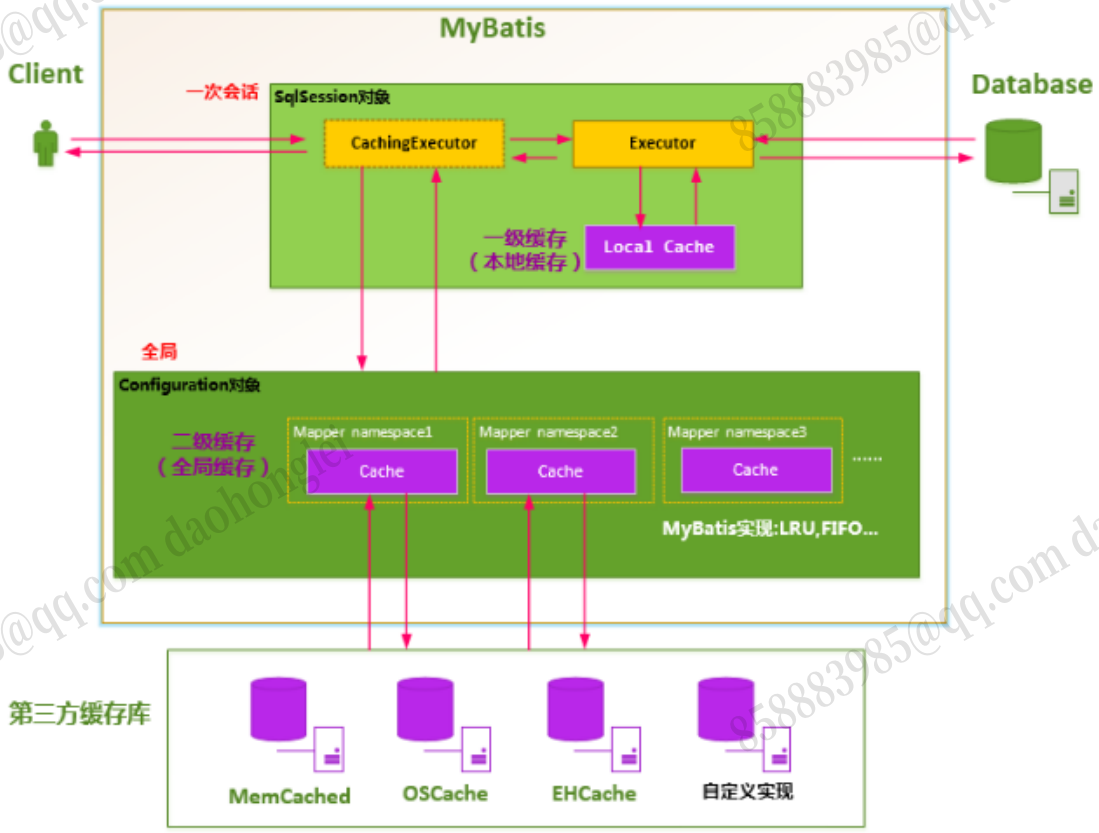
b.避免循环引用

c.加速重复嵌套查询

Mybatis 二级缓存

MyBatis 的二级缓存是 Mapper 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。

MyBatis 的缓存机制整体设计以及二级缓存的工作模式



SqlSessionFactory 层面上的二级缓存默认是不开启的，二级缓存的开席需要进行配置，实现二级缓存的时候，MyBatis 要求返回的 POJO 必须是可序列化的， 也就是要求实现 Serializable 接口。配置方法很简单，只需要在映射 XML 文件配置就可以开启缓存了 <cache/>，如果我们配置了二级缓存就意味着：

- 映射语句文件中的所有 select 语句将会被缓存。
- 映射语句文件中的所有 insert、update 和 delete 语句会刷新缓存。
- 缓存会使用默认的 Least Recently Used（LRU，最近最少使用的）算法来收回。
- 根据时间表，比如 No Flush Interval，（CNFI 没有刷新间隔），缓存不会以任何时间顺序来刷新。
- 缓存会存储列表集合或对象(无论查询方法返回什么)的 1024 个引用
- 缓存会被视为是 read/write(可读/可写)的缓存，意味着对象检索不是共享的，而且可以安全的被调用者修改，不干扰其他调用者或线程所做的潜在修改。

在 MyBatis 中有 flushCache、useCache 这两个配置属性，分为下面几种情况：

（1）当为 select 语句时：

flushCache 默认为 false，表示任何时候语句被调用，都不会去清空本地缓存和二级缓存。

useCache 默认为 true，表示会将本条语句的结果进行二级缓存。

（2）当为 insert、update、delete 语句时：

flushCache 默认为 true，表示任何时候语句被调用，都会导致本地缓存和二级缓存被清空。

useCache 属性在该情况下没有。

上面的信息我是从 MyBatis 官方文档中找到的，会发现当为 select 语句的时候，如果没有去配置 flushCache、useCache，那么默认是启用缓存的，所以，如果有必要，那么就需要人工修改配置，修改结果类似下面：

```
<select id="save" parameterType="XXXXXEO" statementType="CALLABLE"
flushCache="true" useCache="false">
      ……
</select>
```

上面的 statementType="CALLABLE"这个属性是我的项目中需要用到的，如果用不到就不用管它，注意后面的 flushCache="true"和 useCache="false"，做了如上设置以后，发现问题就解决了

使 MyBatis 一级缓存失效

- 全局设置 `local-cache-scope=statement` ，则查询之后即便放入了一级缓存，但存放完立马就给清了，下一次还是要查数据库；
- statement 设置 `flushCache="true"` ，则查询之前先清空一级缓存，还是得查数据库；
- 设置随机数，如果随机数的上限足够大，那随机到相同数的概率就足够低，也能类似的看成不同的数据库请求，那缓存的 key 都不一样，自然就不会匹配到缓存。

Mybatis 如何通过只需要配置接口就能实现数据库的操作

我们进入 DefaultSqlSession 的 getMapper(Class type)方法，

```
@Override
public <T> T getMapper(Class<T> type) {
    return configuration.<T>getMapper(type, this);
}
```

其中 configuration 是 Configuration 的实例，在解析 mybatis 的配置文件的时候进行的初始化。继续追进去

```
public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    return mapperRegistry.getMapper(type, sqlSession);
}
```

继续进到 MapperRegistry 的 getMapper(Class type, SqlSession sqlSession)方法

```
public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>) knownMappers.get(type);
    if (mapperProxyFactory == null) {
        throw new BindingException("Type " + type + " is not known to the MapperRegistry.");
    }
    try {
        return mapperProxyFactory.newInstance(sqlSession);
    } catch (Exception e) {
        throw new BindingException("Error getting mapper instance. Cause: " + e, e);
    }
}
```

此方法根据传进来的 type 生成对应的代理，我们进入看看 MapperProxyFactory

```
public T newInstance(SqlSession sqlSession) {
    final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface, methodCache);
    return newInstance(mapperProxy);
}
```

```
protected T newInstance(MapperProxy<T> mapperProxy) {
    return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[] { mapperInterface }, mapperProxy);
}
```

到这里已经看到已经完成代理的生成，MapperProxyFactory 是个工厂。再继续看 MapperProxy 这个类

```
public class MapperProxy<T> implements InvocationHandler, Serializable {
    private static final long serialVersionUID = -6424540398559729838L;
    private final SqlSession sqlSession;
    private final Class<T> mapperInterface;
    private final Map<Method, MapperMethod> methodCache;

    public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method, MapperMethod> methodCache) {
        this.sqlSession = sqlSession;
        this.mapperInterface = mapperInterface;
        this.methodCache = methodCache;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        if (Object.class.equals(method.getDeclaringClass())) {
            try {
                return method.invoke(this, args);
            } catch (Throwable t) {
                throw ExceptionUtil.unwrapThrowable(t);
            }
        }
        final MapperMethod mapperMethod = cachedMapperMethod(method);
        return mapperMethod.execute(sqlSession, args);
    }
    private MapperMethod cachedMapperMethod(Method method) {
        MapperMethod mapperMethod = methodCache.get(method);
        if (mapperMethod == null) {
            mapperMethod = new MapperMethod(mapperInterface, method, sqlSession.getConfiguration());
```

```
            methodCache.put(method, mapperMethod);
        }
        return mapperMethod;
    }
}
```

当我们通过生成的对象调用方法的时候，都会进入这个类的 invoke 方法，我看到如果调用的是我们自定的方法，直接就是调用的 mybatis 的实现，通过接口找到配置信息，然后根据我们的配置去操作数据库。

最后总结一下，mybatis 之所以配置接口以后就能执行是因为在生产 mapper 的时候实质上是生成的一个代理，然后通过 mapper 调用接口方法的时候直接被 MapperProxy 的 invoke 截断了，直接去调用了 mybatis 为我们制定的实现，而没有去回调。

Mybatis 懒加载使用及源码分析

1.ResultSetHandler 通过反射创建出 resultObject，然后遍历去检查这些属性是否是懒加载的，如果是那么就通过代理工厂去创建一个代理对象，由于这里创建的是一个返回对象，不是一个接口因此动态代理实现是通过 cglib 实现的，Mybatis 这里使用 javassist 包下的代理进行创建代理对象，代理工厂默认就是 JavassistProxyFactory。

2.当我们调用 getOrderList 方法的时候就会执行到 invoke 方法中，并且判断是否是延迟加载的，如果是那么就会执行 lazyLoader.load 方法执行延迟加载，也就是执行 sql 查询数据。

递归查询

```
<resultMap id="BaseResultMap"
```

```
type="com.itiaoling.system.dto.district.DistrictResponseDto">
    <result column="district_type" jdbcType="TINYINT" property="districtType" />
    <result column="district_code" jdbcType="VARCHAR" property="districtCode"
/>
    <result column="custom_district_code" jdbcType="VARCHAR"
property="customDistrictCode" />
    <result column="version" jdbcType="BIGINT" property="version"/>
    <collection property="languageNames" javaType="java.util.List"
ofType="com.itiaoling.system.dto.district.BaseDistrictLanguageDto">
        <result column="language" jdbcType="VARCHAR" property="language"/>
        <result column="district_name" jdbcType="VARCHAR" property="name"/>
        <result column="name_abbreviation" jdbcType="VARCHAR"
property="nameAbbreviation"/>
    </collection>
</resultMap>


<resultMap id="BaseResultMap2"
type="com.itiaoling.system.dto.district.DistrictResponseDto"
extends="BaseResultMap">
    <collection property="subsetDdistricts"
ofType="com.itiaoling.system.dto.district.DistrictResponseDto"
            javaType="java.util.List"
```

```
            column="{parentDistrictCode=district_code,source=project_code}"
select="pullDistrictDataWithSub">
    </collection>
</resultMap>


<select id="pullDistrictDataWithSub"
parameterType="com.itiaoling.system.dto.district.PullDistrictDto"
resultMap="BaseResultMap2">
    <include refid="BaseSql"/>
    <where>
        base.is_active=1
        <if test="districtCode !=null and districtCode!=''">
            and base.district_code=#{districtCode}
        </if>
        <if test="parentDistrictCode !=null and parentDistrictCode!=''">
            and base.parent_district_code=#{parentDistrictCode}
        </if>
    </where>
</select>
```

https://github.com/daohonglei/javaStereotypedWriting

https://gitee.com/daohonglei/javaStereotypedWriting