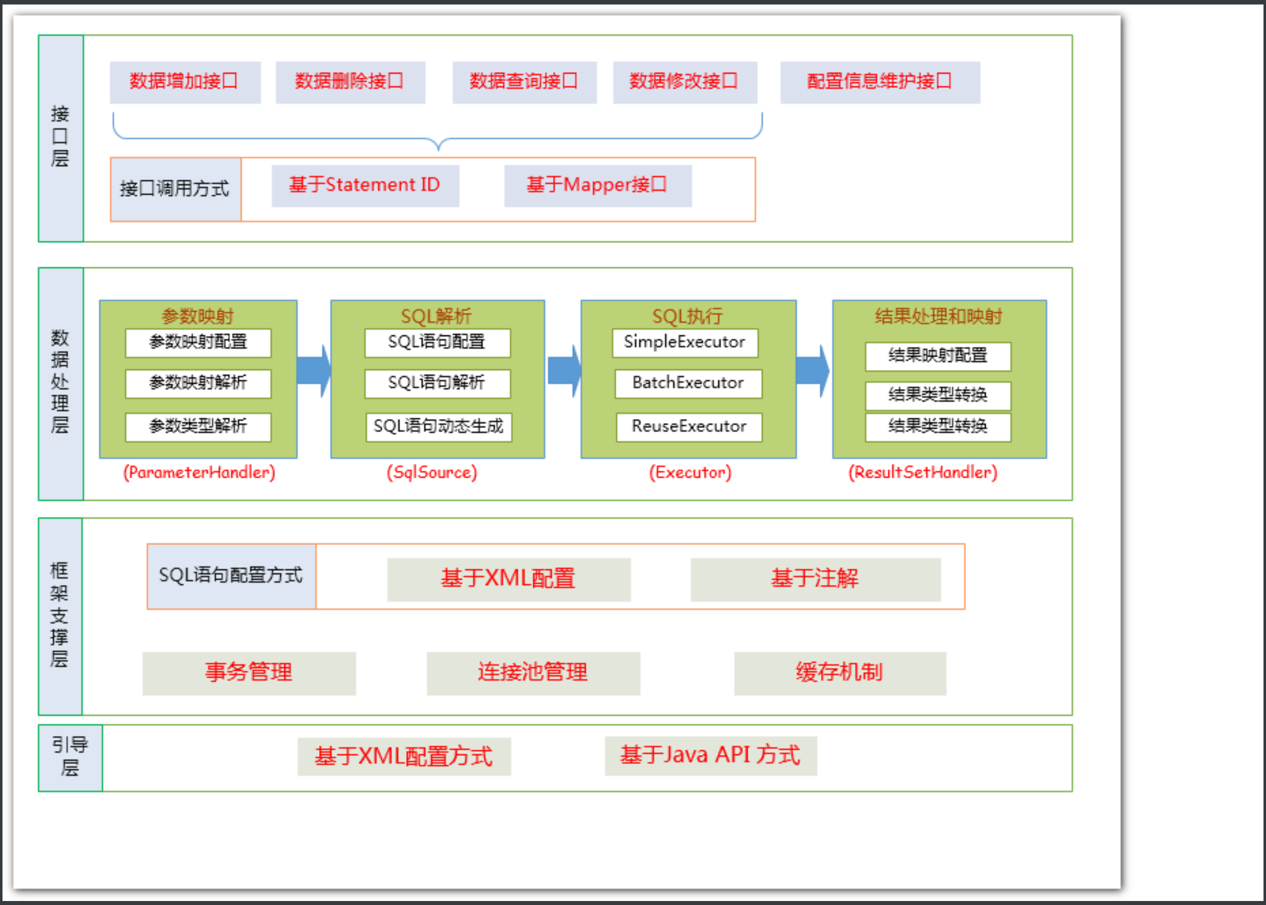
Mybatis源码解析

> Mybatis 是一款优秀的持久化框架。
>
> 支持定制化SQL、存储过程以及高级映射。
>
> 避免了几乎所有JDBC代码、手动参数设置、获取结果集。
>
> 通过xml或者annotation来配置和映射原生信息。

1、架构



接口层

> 定义与数据库进行交互的方式。
>
> 分为mybatis的api和mapper接口的调用方式。

- api方式【不常用】

  > mybatis提供的调用方式
  >
  > 获取sqlSession对象，根据statementId和参数调用数据库。

```
SqlSessionFactory sessionFactory = getSessionFactory();
SqlSession sqlSession = sessionFactory.openSession();
try {
  //statementId:UserMapper.selectUser
  //args:1
  UserDo userDo = sqlSession.selectOne("UserMapper.selectUser", 1);
  log.info("userDo:{}", userDo.toString());
} finally {
  sqlSession.close();
}
```

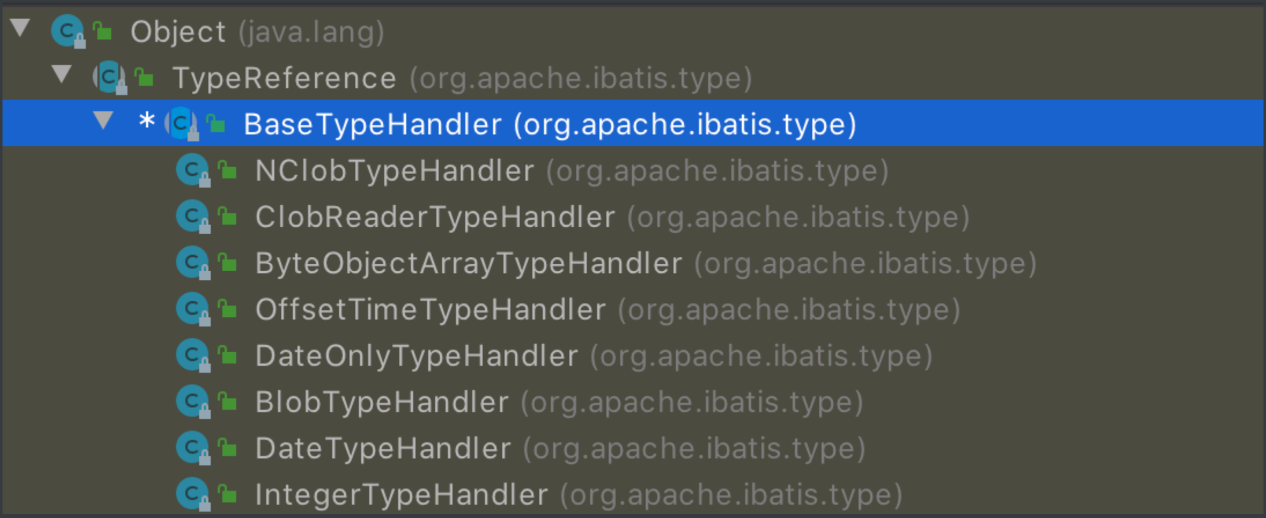
- mapper接口【也就是dao的interface】

  > 每个mapper接口中定义的方法一一对应着mapper.xml文件中的增删改查节点。
  >
  > 每个节点对应的id是mapper接口中的方法名。
  >
  > 最终还是调用sqlSession的增删改查方法，但是有一层mapperProxy的JDK代理封装

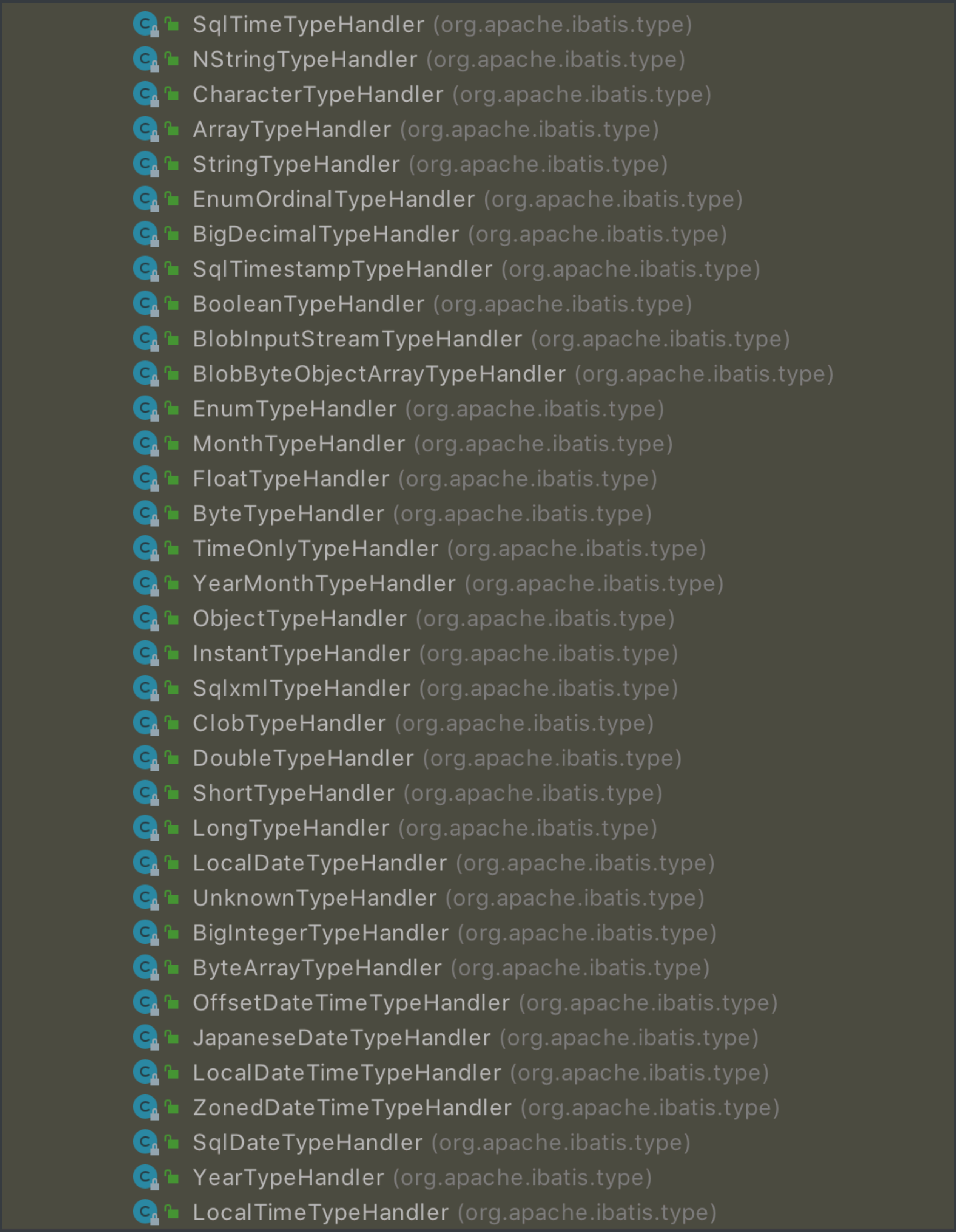
数据处理层

> 负责参数映射和动态sql生成，生成sql执行语句，进行结果集转换
>
> 使用typeHandler进行java类型和jdbc类型的相互转换

数据处理器TypeHandler的hierarchy。



```
▼ Object (java.lang)
  ▼ TypeReference (org.apache.ibatis.type)
    ▼ * BaseTypeHandler (org.apache.ibatis.type)
          NClobTypeHandler (org.apache.ibatis.type)
          ClobReaderTypeHandler (org.apache.ibatis.type)
          ByteObjectArrayTypeHandler (org.apache.ibatis.type)
          OffsetTimeTypeHandler (org.apache.ibatis.type)
          DateOnlyTypeHandler (org.apache.ibatis.type)
          BlobTypeHandler (org.apache.ibatis.type)
          DateTypeHandler (org.apache.ibatis.type)
          IntegerTypeHandler (org.apache.ibatis.type)
```



```
SqlTimeTypeHandler (org.apache.ibatis.type)
NStringTypeHandler (org.apache.ibatis.type)
CharacterTypeHandler (org.apache.ibatis.type)
ArrayTypeHandler (org.apache.ibatis.type)
StringTypeHandler (org.apache.ibatis.type)
EnumOrdinalTypeHandler (org.apache.ibatis.type)
BigDecimalTypeHandler (org.apache.ibatis.type)
SqlTimestampTypeHandler (org.apache.ibatis.type)
BooleanTypeHandler (org.apache.ibatis.type)
BlobInputStreamTypeHandler (org.apache.ibatis.type)
BlobByteObjectArrayTypeHandler (org.apache.ibatis.type)
EnumTypeHandler (org.apache.ibatis.type)
MonthTypeHandler (org.apache.ibatis.type)
FloatTypeHandler (org.apache.ibatis.type)
ByteTypeHandler (org.apache.ibatis.type)
TimeOnlyTypeHandler (org.apache.ibatis.type)
YearMonthTypeHandler (org.apache.ibatis.type)
ObjectTypeHandler (org.apache.ibatis.type)
InstantTypeHandler (org.apache.ibatis.type)
SqlxmlTypeHandler (org.apache.ibatis.type)
ClobTypeHandler (org.apache.ibatis.type)
DoubleTypeHandler (org.apache.ibatis.type)
ShortTypeHandler (org.apache.ibatis.type)
LongTypeHandler (org.apache.ibatis.type)
LocalDateTypeHandler (org.apache.ibatis.type)
UnknownTypeHandler (org.apache.ibatis.type)
BigIntegerTypeHandler (org.apache.ibatis.type)
ByteArrayTypeHandler (org.apache.ibatis.type)
OffsetDateTimeTypeHandler (org.apache.ibatis.type)
JapaneseDateTypeHandler (org.apache.ibatis.type)
LocalDateTimeTypeHandler (org.apache.ibatis.type)
ZonedDateTimeTypeHandler (org.apache.ibatis.type)
SqlDateTypeHandler (org.apache.ibatis.type)
YearTypeHandler (org.apache.ibatis.type)
LocalTimeTypeHandler (org.apache.ibatis.type)
```

框架支撑层

- 事务管理

  提供平台事务管理类接口PlatformTransactionManager供平台管理。比如spring的DataSourceTransactionManager。

  提供TransactionStatus事务状态、TransactionDefinition事务定义信息...

- 连接池

不可能每次执行sql的时候，都去数据库实例获取一次连接。

因为创建连接比较耗时。通常做法是提前创建好N个连接到连接池，用的时候去连接池拿，用完再还回去。开源实现：c3p0,DBCP,tomcat jdbc pool,druid,HikariCP...

- 缓存

  提高数据利用率，减少对数据库的查询压力，mybatis支持两层缓存。

  一层是SqlSession的缓存，第一次查询会把从数据库实例查询到的数据写进缓存，也就是jvm中，第二次执行相同查询就会直接从缓存中取。

  二层是mapper的缓存，多个sqlSession去操作同一个mapper的sql语句，会使用共享的二级缓存，需要在mybatis的sql映射文件中单独配置节点。将所有的select缓存，insert/update/delete时刷新缓存。缓存算法有很多：LRU最久未使用、FIFO先进先出、OPT最佳置换、NRU Clock置换、LFU最少使用置换、PBA页面缓存...

sql配置

annotation和xml两种方式。

annotation直接标注在mapper接口的方法上。

xml写在mapper.xml里。

引导层

引导层是配置mybatis的。也就是Configuration的配置。可以使用xml也可以使用java API注入bean。但理论上都是spring的bean。

2、组件

- SqlSession

  跟数据库交互的一次会话，完成跟数据库的增删改查。

- Executor

  执行器，也负责sql生成和缓存维护。传入MapperStatement。

- StatementHandler

  封装了jdbc statement操作。设置参数，结果转换。

- ParameterHandler

  java参数转换为jdbc参数。

- ResultSetHandler

  jdbc返回结果集转换java对象。

- TypeHandler

java类型和jdbc类型互转。

- MapperStatement

  维护了一条<select/update/insert/delete>的封装。

- SqlSource

  根据用户传递的java对象，动态的生成SQL语句，封装到BoundSql对象中。

- BoundSql

  动态生成sql语句以及相应参数信息。

- Configuration

  整个的mybatis的配置信息。

3、spring集成

```xml
<!--数据源的事物管理器DataSourceTransactionManager-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>

<!--池化数据源PooledDataSource-->
<bean id="dataSource"
class="org.apache.ibatis.datasource.pooled.PooledDataSource">
  <property name="driver" value="${datasource.driverClassName}"/>
  <property name="url" value="${datasource.url}"/>
  <property name="username" value="${datasource.username}"/>
  <property name="password" value="${datasource.password}"/>
</bean>
<!--sqlSessionFactoryBean工厂bean解析configuration生成SqlSessionFactory-->
<bean name="sqlSessionFactoryBean"
class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="mapperLocations" value="classpath:mappers/*.xml"/>
  <property name="plugins">
    <array>
      <!--自定义的DO对象拦截器设置id-->
      <bean id="myBatisIDInterceptor"
class="com.uteam.zen.core.interceptors.MyBatisIDInterceptor"/>
    </array>
  </property>
</bean>
<!--MapperScannerConfigurer是bean定义信息的后置处理器，在系统启动之初调用-->
<!--扫描包，通过定义的Repository注解-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
```

```
  <property name="basePackage"
value="com.uteam.zen.system.persistent.dao"/>
  <property name="annotationClass"
value="org.springframework.stereotype.Repository"/>
  <property name="sqlSessionFactoryBeanName"
value="sqlSessionFactoryBean"/>
</bean>
```

3.1、如果不使用spring，也可以使用mybatis自带的sqlFactorybean创建方式

```
//这里得是mybatis的configuration的配置xml
String resource = "org/mybatis/example/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory =
        new SqlSessionFactoryBuilder().build(inputStream);
```

> 通过xml和SqlSessionFactoryBuilder建造者建造SqlSessionFactory

3.2、SqlSessionFactoryBean执行

> SqlSessionFactoryBean实现了InitializingBeanbean的初始化接口，会在实例化之后调用
>
> afterPropertiesSet方法

```
@Override
public void afterPropertiesSet() throws Exception {
  //bean属性判断
  notNull(dataSource, "Property 'dataSource' is required");
  notNull(sqlSessionFactoryBuilder, "Property 'sqlSessionFactoryBuilder' is
required");
  state((configuration == null && configLocation == null) || !
(configuration != null && configLocation != null),
              "Property 'configuration' and 'configLocation' can not
specified with together");
  //装载sqlSessionFactory
  this.sqlSessionFactory = buildSqlSessionFactory();
}
```

4、构建SqlSessionFactory

4.1、SqlSessionFactory接口

```
public interface SqlSessionFactory {

  //获取sqlSession对象，完成数据库增删改查功能
  SqlSession openSession();
  SqlSession openSession(boolean autoCommit);
  SqlSession openSession(Connection connection);
  SqlSession openSession(TransactionIsolationLevel level);
  SqlSession openSession(ExecutorType execType);
  SqlSession openSession(ExecutorType execType, boolean autoCommit);
  SqlSession openSession(ExecutorType execType, TransactionIsolationLevel
level);
  SqlSession openSession(ExecutorType execType, Connection connection);

  //获取Configuration,存放了mybatis配置的所有信息
  Configuration getConfiguration();

}
```

4.2 Configuration

> SqlSessionFactoryBean创建SqlSessionFactory时buildSqlSessionFactory，会解析生成Configuration放在sqlSessionFactory中。

4.2.1 Configuration的属性

```
//环境
protected Environment environment;

protected boolean safeRowBoundsEnabled;
protected boolean safeResultHandlerEnabled = true;
protected boolean mapUnderscoreToCamelCase;
protected boolean aggressiveLazyLoading;
protected boolean multipleResultSetsEnabled = true;
protected boolean useGeneratedKeys;
protected boolean useColumnLabel = true;
protected boolean cacheEnabled = true;
protected boolean callSettersOnNulls;
protected boolean useActualParamName = true;
protected boolean returnInstanceForEmptyRow;

//日志前缀
protected String logPrefix;
//日志接口
```

```
protected Class<? extends Log> logImpl;
//系统文件接口
protected Class<? extends VFS> vfsImpl;
//本地缓存域，默认session
protected LocalCacheScope localCacheScope = LocalCacheScope.SESSION;
//数据库类型
protected JdbcType jdbcTypeForNull = JdbcType.OTHER;
//延迟加载的方法名
protected Set<String> lazyLoadTriggerMethods = new HashSet<>
(Arrays.asList("equals", "clone", "hashCode", "toString"));
//默认执行语句超时时间
protected Integer defaultStatementTimeout;
protected Integer defaultFetchSize;
//默认结果集类型
protected ResultSetType defaultResultSetType;
//默认执行器类型
protected ExecutorType defaultExecutorType = ExecutorType.SIMPLE;
protected AutoMappingBehavior autoMappingBehavior =
AutoMappingBehavior.PARTIAL;
protected AutoMappingUnknownColumnBehavior autoMappingUnknownColumnBehavior
= AutoMappingUnknownColumnBehavior.NONE;
//参数
protected Properties variables = new Properties();
protected ReflectorFactory reflectorFactory = new
DefaultReflectorFactory();
//默认对象工厂
protected ObjectFactory objectFactory = new DefaultObjectFactory();
//默认对象包装工厂
protected ObjectWrapperFactory objectWrapperFactory = new
DefaultObjectWrapperFactory();
//延迟加载
protected boolean lazyLoadingEnabled = false;
//代理工厂Javassist字节码
protected ProxyFactory proxyFactory = new JavassistProxyFactory(); // #224
Using internal Javassist instead of OGNL
//数据库id
protected String databaseId;
/**
 * Configuration factory class.
 * Used to create Configuration for loading deserialized unread properties.
 *
 * @see <a href='https://code.google.com/p/mybatis/issues/detail?
id=300'>Issue 300 (google code)</a>
 */
protected Class<?> configurationFactory;
```

```
//mapper注册表
protected final MapperRegistry mapperRegistry = new MapperRegistry(this);
//拦截器链
protected final InterceptorChain interceptorChain = new InterceptorChain();
//类型处理器注册表
protected final TypeHandlerRegistry typeHandlerRegistry = new
TypeHandlerRegistry();
//别名注册表
protected final TypeAliasRegistry typeAliasRegistry = new
TypeAliasRegistry();
//语言驱动,构建SQLSource
protected final LanguageDriverRegistry languageRegistry = new
LanguageDriverRegistry();
//mapper_id和mapper文件对应关系
protected final Map<String, MappedStatement> mappedStatements = new
StrictMap<MappedStatement>("Mapped Statements collection")
      .conflictMessageProducer((savedValue, targetValue) ->
          ". please check " + savedValue.getResource() + " and " +
targetValue.getResource());
//mapper_id和缓存映射
protected final Map<String, Cache> caches = new StrictMap<>("Caches
collection");
//mapper_id和ResultMap的映射
protected final Map<String, ResultMap> resultMaps = new StrictMap<>("Result
Maps collection");
//mapper_id和ParameterMap的映射
protected final Map<String, ParameterMap> parameterMaps = new StrictMap<>
("Parameter Maps collection");
//mapper_id和主键生成器的映射
protected final Map<String, KeyGenerator> keyGenerators = new StrictMap<>
("Key Generators collection");
//已加载资源集合
protected final Set<String> loadedResources = new HashSet<>();
//mapper_id和sql片段的映射
protected final Map<String, XNode> sqlFragments = new StrictMap<>("XML
fragments parsed from previous mappers");

//未处理完成的集合，可能存在继承关系，导致有些不能一次初始化完成，先存起来，后续处理。
protected final Collection<XMLStatementBuilder> incompleteStatements = new
LinkedList<>();
protected final Collection<CacheRefResolver> incompleteCacheRefs = new
LinkedList<>();
```

```java
    protected final Collection<ResultMapResolver> incompleteResultMaps = new
    LinkedList<>();
    protected final Collection<MethodResolver> incompleteMethods = new
    LinkedList<>();

    /*
     * A map holds cache-ref relationship. The key is the namespace that
     * references a cache bound to another namespace and the value is the
     * namespace which the actual cache is bound to.
     */
    //名称空间对应的cache映射
    protected final Map<String, String> cacheRefMap = new HashMap<>();
```

configuration的无参构造器里还会初始化一下数据

```java
    public Configuration() {
        //事务工厂相关alias
        typeAliasRegistry.registerAlias("JDBC", JdbcTransactionFactory.class);
        typeAliasRegistry.registerAlias("MANAGED",
    ManagedTransactionFactory.class);
        //数据源工厂相关alias
        typeAliasRegistry.registerAlias("JNDI", JndiDataSourceFactory.class);
        typeAliasRegistry.registerAlias("POOLED",
    PooledDataSourceFactory.class);
        typeAliasRegistry.registerAlias("UNPOOLED",
    UnpooledDataSourceFactory.class);
        //缓存相关alias
        typeAliasRegistry.registerAlias("PERPETUAL", PerpetualCache.class);
        typeAliasRegistry.registerAlias("FIFO", FifoCache.class);
        typeAliasRegistry.registerAlias("LRU", LruCache.class);
        typeAliasRegistry.registerAlias("SOFT", SoftCache.class);
        typeAliasRegistry.registerAlias("WEAK", WeakCache.class);
        //数据库id生成器相关alias
        typeAliasRegistry.registerAlias("DB_VENDOR",
    VendorDatabaseIdProvider.class);
        //LanguageDriver相关alias
        typeAliasRegistry.registerAlias("XML", XMLLanguageDriver.class);
        typeAliasRegistry.registerAlias("RAW", RawLanguageDriver.class);
        //日志相关alias
        typeAliasRegistry.registerAlias("SLF4J", Slf4jImpl.class);
        typeAliasRegistry.registerAlias("COMMONS_LOGGING",
    JakartaCommonsLoggingImpl.class);
        typeAliasRegistry.registerAlias("LOG4J", Log4jImpl.class);
        typeAliasRegistry.registerAlias("LOG4J2", Log4j2Impl.class);
        typeAliasRegistry.registerAlias("JDK_LOGGING", Jdk14LoggingImpl.class);
        typeAliasRegistry.registerAlias("STDOUT_LOGGING", StdOutImpl.class);
        typeAliasRegistry.registerAlias("NO_LOGGING", NoLoggingImpl.class);
        //动态代理相关alias
        typeAliasRegistry.registerAlias("CGLIB", CglibProxyFactory.class);
        typeAliasRegistry.registerAlias("JAVASSIST",
    JavassistProxyFactory.class);
        //设置默认语言驱动是XMLLanguageDriver
        languageRegistry.setDefaultDriverClass(XMLLanguageDriver.class);
        languageRegistry.register(RawLanguageDriver.class);
      }
```

4.3、buildSqlSessionFactory构建过程

> 分为两部分。
>
> 一个是组件加载到configuration。
>
> 一个是解析mapper文件，封装MapperStatement对象，配置到configuration里。

```
protected SqlSessionFactory buildSqlSessionFactory() throws IOException {

    Configuration configuration;

    //如果configuration不是空，就往里面塞值
    //把Variables和configurationProperties的值放进去。
    XMLConfigBuilder xmlConfigBuilder = null;
    if (this.configuration != null) {
      configuration = this.configuration;
      if (configuration.getVariables() == null) {
        configuration.setVariables(this.configurationProperties);
      } else if (this.configurationProperties != null) {
        configuration.getVariables().putAll(this.configurationProperties);
      }
    } else if (this.configLocation != null) {
      //或者通过configuration.xml配置文件地址解析configuration
      //这里只组装xmlConfigBuilder，后面解析。
      xmlConfigBuilder = new
XMLConfigBuilder(this.configLocation.getInputStream(), null,
this.configurationProperties);
      configuration = xmlConfigBuilder.getConfiguration();
    } else {
      if (LOGGER.isDebugEnabled()) {
        LOGGER.debug("Property 'configuration' or 'configLocation' not
specified, using default MyBatis Configuration");
      }
      //上面两种方式都不能满足Configuration加载
      //在这里new一个Configuration
      configuration = new Configuration();
      if (this.configurationProperties != null) {
        configuration.setVariables(this.configurationProperties);
      }
    }

    //可以使用指定的objectFactory工厂替换DefaultObjectFactory
    if (this.objectFactory != null) {
      configuration.setObjectFactory(this.objectFactory);
    }
    //可以使用指定的objectWrapperFactory工厂替换DefaultObjectWrapperFactory
    if (this.objectWrapperFactory != null) {
      configuration.setObjectWrapperFactory(this.objectWrapperFactory);
    }

    if (this.vfs != null) {
      configuration.setVfsImpl(this.vfs);
    }
    //如果制定了typeAliasesPackage
    if (hasLength(this.typeAliasesPackage)) {
      //就把每一个包路径取出来
      String[] typeAliasPackageArray =
tokenizeToStringArray(this.typeAliasesPackage,
          ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS);
      //遍历
      for (String packageToScan : typeAliasPackageArray) {
        //把每一个报下的组件，批量注册到TypeAlias中
        configuration.getTypeAliasRegistry().registerAliases(packageToScan,
                typeAliasesSuperType == null ? Object.class :
typeAliasesSuperType);
        if (LOGGER.isDebugEnabled()) {
          LOGGER.debug("Scanned package: '" + packageToScan + "' for
aliases");
        }
      }
    }
    //如果制定了多个typeAliases
    if (!isEmpty(this.typeAliases)) {
```

```java
      //遍历
      for (Class<?> typeAlias : this.typeAliases) {
        //一个个注册到TypeAlias中
        configuration.getTypeAliasRegistry().registerAlias(typeAlias);
        if (LOGGER.isDebugEnabled()) {
          LOGGER.debug("Registered type alias: '" + typeAlias + "'");
        }
      }
    }

    //指定插件不为空，添加到拦截器链中
    if (!isEmpty(this.plugins)) {
      for (Interceptor plugin : this.plugins) {
        configuration.addInterceptor(plugin);
        if (LOGGER.isDebugEnabled()) {
          LOGGER.debug("Registered plugin: '" + plugin + "'");
        }
      }
    }
    //如果指定typeHandler的扫包路径集合
    if (hasLength(this.typeHandlersPackage)) {
      String[] typeHandlersPackageArray =
tokenizeToStringArray(this.typeHandlersPackage,
          ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS);
      //按照包路径
      for (String packageToScan : typeHandlersPackageArray) {
        //添加到TypeHandlerMap中
        configuration.getTypeHandlerRegistry().register(packageToScan);
        if (LOGGER.isDebugEnabled()) {
          LOGGER.debug("Scanned package: '" + packageToScan + "' for type
handlers");
        }
      }
    }
    //如果指定TypeHandler
    if (!isEmpty(this.typeHandlers)) {
      //就一个个的注册到TypeHandlerMap中
      for (TypeHandler<?> typeHandler : this.typeHandlers) {
        configuration.getTypeHandlerRegistry().register(typeHandler);
        if (LOGGER.isDebugEnabled()) {
          LOGGER.debug("Registered type handler: '" + typeHandler + "'");
        }
      }
    }
    //可以指定databaseId的提供者
    //通过dataSource和databaseIdProvider获得DatabaseId
    if (this.databaseIdProvider != null) {//fix #64 set databaseId before
parse mapper xmls
      try {

 configuration.setDatabaseId(this.databaseIdProvider.getDatabaseId(this.dat
aSource));
      } catch (SQLException e) {
        throw new NestedIOException("Failed getting a databaseId", e);
      }
    }
    //添加缓存
    if (this.cache != null) {
      configuration.addCache(this.cache);
    }

    //如果上面构建了xmlConfigBuilder，就在这里进行解析。
    if (xmlConfigBuilder != null) {
      try {
        //这里的解析。后面再说。
        xmlConfigBuilder.parse();

        if (LOGGER.isDebugEnabled()) {
          LOGGER.debug("Parsed configuration file: '" + this.configLocation
+ "'");
        }
      } catch (Exception ex) {
        throw new NestedIOException("Failed to parse config resource: " +
this.configLocation, ex);
      } finally {
        ErrorContext.instance().reset();
      }
    }
    //如果没有指定事务工厂transactionFactory，默认设置
SpringManagedTransactionFactory
    if (this.transactionFactory == null) {
      this.transactionFactory = new SpringManagedTransactionFactory();
    }
    //设置环境拼装一个Environment对象
    configuration.setEnvironment(new Environment(this.environment,
this.transactionFactory, this.dataSource));

    //设置所有的mapper。通过mapperLocations。
    //<property name="mapperLocations" value="classpath:mappers/*.xml"/>
    if (!isEmpty(this.mapperLocations)) {
      for (Resource mapperLocation : this.mapperLocations) {
        if (mapperLocation == null) {
          continue;
        }

        try {
          //每一个都使用XMLMapperBuilder进行解析。放在configuration中。
          XMLMapperBuilder xmlMapperBuilder = new
XMLMapperBuilder(mapperLocation.getInputStream(),
              configuration, mapperLocation.toString(),
configuration.getSqlFragments());
          xmlMapperBuilder.parse();
        } catch (Exception e) {
          throw new NestedIOException("Failed to parse mapping resource: '"
+ mapperLocation + "'", e);
        } finally {
          ErrorContext.instance().reset();
        }

        if (LOGGER.isDebugEnabled()) {
          LOGGER.debug("Parsed mapper file: '" + mapperLocation + "'");
        }
      }
    } else {
      if (LOGGER.isDebugEnabled()) {
        LOGGER.debug("Property 'mapperLocations' was not specified or no
matching resources found");
      }
    }
    //通过Configuration和sqlSessionFactoryBuilder构建器，构建sqlSessionFactory
    return this.sqlSessionFactoryBuilder.build(configuration);
  }
```

4.3.1、typeAliases

> 类型别名。比如DO的全类名com.uteam.zen.system.persistent.dataObject.UserDo，可以指定为user。
>
> 在resultType处设置相同。避免写全类名。

看上面源码也提到了。可以有两种配置方式。

```
<property name="typeAliasesPackage">
    <array>
        <value>com.uteam.zen.system.persistent.dataObject</value>
    </array>
</property>
或者
<property name="typeAliases">
    <array>
        <value>com.uteam.zen.system.persistent.dataObject.UserDo</value>
    </array>
</property>
```

解析过程：

> 通过全类名拿到反射对象，获取类名，key值是类名小写，value是class对象。放入map中。
>
> 下面是源码

```
  public void registerAlias(Class<?> type) {
    //获得他的类名
    String alias = type.getSimpleName();
    //或者是通过@Alias注解的value值
    Alias aliasAnnotation = type.getAnnotation(Alias.class);
    if (aliasAnnotation != null) {
      alias = aliasAnnotation.value();
    }
    //注册进typeAliases的map中
    registerAlias(alias, type);
  }

  public void registerAlias(String alias, Class<?> value) {
    if (alias == null) {
      throw new TypeException("The parameter alias cannot be null");
    }
    // issue #748
    //名称变小写
    String key = alias.toLowerCase(Locale.ENGLISH);
    //如果有重复就报错。
    if (typeAliases.containsKey(key) && typeAliases.get(key) != null &&
!typeAliases.get(key).equals(value)) {
      throw new TypeException("The alias '" + alias + "' is already mapped
to the value '" + typeAliases.get(key).getName() + "'.");
```

```
        }
        //放入map中
        typeAliases.put(key, value);
      }
```

typeAliases就是个HashMap

```
public class TypeAliasRegistry {
  private final Map<String, Class<?>> typeAliases = new HashMap<>();
```

初始会放一些常用的类型别名

```
registerAlias("string", String.class);

    registerAlias("byte", Byte.class);
    registerAlias("long", Long.class);
    registerAlias("short", Short.class);
    registerAlias("int", Integer.class);
    registerAlias("integer", Integer.class);
    registerAlias("double", Double.class);
    registerAlias("float", Float.class);
    registerAlias("boolean", Boolean.class);

    registerAlias("byte[]", Byte[].class);
    registerAlias("long[]", Long[].class);
    registerAlias("short[]", Short[].class);
    registerAlias("int[]", Integer[].class);
    registerAlias("integer[]", Integer[].class);
    registerAlias("double[]", Double[].class);
    registerAlias("float[]", Float[].class);
    registerAlias("boolean[]", Boolean[].class);

    registerAlias("_byte", byte.class);
    registerAlias("_long", long.class);
    registerAlias("_short", short.class);
    registerAlias("_int", int.class);
    registerAlias("_integer", int.class);
    registerAlias("_double", double.class);
    registerAlias("_float", float.class);
    registerAlias("_boolean", boolean.class);

    registerAlias("_byte[]", byte[].class);
    registerAlias("_long[]", long[].class);
```

```
        registerAlias("_short[]", short[].class);
        registerAlias("_int[]", int[].class);
        registerAlias("_integer[]", int[].class);
        registerAlias("_double[]", double[].class);
        registerAlias("_float[]", float[].class);
        registerAlias("_boolean[]", boolean[].class);

        registerAlias("date", Date.class);
        registerAlias("decimal", BigDecimal.class);
        registerAlias("bigdecimal", BigDecimal.class);
        registerAlias("biginteger", BigInteger.class);
        registerAlias("object", Object.class);

        registerAlias("date[]", Date[].class);
        registerAlias("decimal[]", BigDecimal[].class);
        registerAlias("bigdecimal[]", BigDecimal[].class);
        registerAlias("biginteger[]", BigInteger[].class);
        registerAlias("object[]", Object[].class);

        registerAlias("map", Map.class);
        registerAlias("hashmap", HashMap.class);
        registerAlias("list", List.class);
        registerAlias("arraylist", ArrayList.class);
        registerAlias("collection", Collection.class);
        registerAlias("iterator", Iterator.class);

        registerAlias("ResultSet", ResultSet.class);
```

4.3.2、typeHandlers

> 类型转换器。用于java对象和jdbc对象的转换
>
> 跟typeAliases一样。也是两种方式配置package或class
>
> 需要注意的是java类型作为key和jdbc类型作为key，会分别存两张表
>
> jdbcTypeHandlerMap、typeHandlerMap

```
<property name="typeHandlers">
    <array>
        <bean
class="com.uteam.zen.system.persistent.typeHandlers.CustomTimestampTypeHand
ler">
      </bean>
    </array>
</property>
或者
<property name="typeHandlersPackage">
    <array>
        <bean class="com.uteam.zen.system.persistent.typeHandlers"></bean>
    </array>
</property>
```

例子：自定义时间戳类型转换器

```
//JDBC类型用@MappedJdbcTypes指定
@MappedJdbcTypes({ JdbcType.TIMESTAMP })
//java类型用@MappedTypes指定
@MappedTypes({ String.class })
//这里的泛型类型是java返回类型
public class CustomTimestampTypeHandler extends BaseTypeHandler<String> {
    @Override
    public void setNonNullParameter(PreparedStatement ps, int i, String
parameter, JdbcType jdbcType)
            throws SQLException {
        ps.setString(i, parameter);
    }
    @Override
    public String getNullableResult(ResultSet rs, String columnName) throws
SQLException {
        return substring(rs.getString(columnName));
    }
    @Override
    public String getNullableResult(ResultSet rs, int columnIndex) throws
SQLException {
        return rs.getString(columnIndex);
    }
    @Override
    public String getNullableResult(CallableStatement cs, int columnIndex)
throws SQLException {
        return cs.getString(columnIndex);
    }
    private String substring(String value) {
        //只做了一个去处最后两位
        //也就是毫秒不显示
        if (!"".endsWith(value) && value != null) {
            return value.substring(0, value.length() - 2);
        }
        return value;
    }
}
```

注册源码

```
      public <T> void register(TypeHandler<T> typeHandler) {
        boolean mappedTypeFound = false;
        //获取TypeHandler类的注解@MappedTypes
        MappedTypes mappedTypes =
    typeHandler.getClass().getAnnotation(MappedTypes.class);
        if (mappedTypes != null) {
          for (Class<?> handledType : mappedTypes.value()) {
            //注册
            register(handledType, typeHandler);
            mappedTypeFound = true;
          }
        }
        //如果要是没有配置注解。就使用类上标注的泛型类注册
        // @since 3.1.0 - try to auto-discover the mapped type
        if (!mappedTypeFound && typeHandler instanceof TypeReference) {
          try {
            TypeReference<T> typeReference = (TypeReference<T>) typeHandler;
            register(typeReference.getRawType(), typeHandler);
            mappedTypeFound = true;
          } catch (Throwable t) {
            // maybe users define the TypeReference with a different type and
    are not assignable, so just ignore it
          }
        }
        if (!mappedTypeFound) {
          register((Class<T>) null, typeHandler);
        }
      }
    //-------------------------中间省略一些重载方法------------------------------
    -------
```

```
  private <T> void register(Type javaType, TypeHandler<? extends T>
typeHandler) {
    //获取TypeHandler类的注解@MappedJdbcTypes
    //也就是jdbc类型
    MappedJdbcTypes mappedJdbcTypes =
      typeHandler.getClass().getAnnotation(MappedJdbcTypes.class);
    if (mappedJdbcTypes != null) {
      for (JdbcType handledJdbcType : mappedJdbcTypes.value()) {
        register(javaType, handledJdbcType, typeHandler);
      }
      if (mappedJdbcTypes.includeNullJdbcType()) {
        register(javaType, null, typeHandler);
      }
    } else {
      register(javaType, null, typeHandler);
    }
  }

  private void register(Type javaType, JdbcType jdbcType, TypeHandler<?>
handler) {
    if (javaType != null) {
      //typeHandlerMap是java类型的默认处理器
      //以String为例。默认可以处理VARCHAR、CHAR、NVARCHAR、CLOB、NCLOB、NULL
      Map<JdbcType, TypeHandler<?>> map = typeHandlerMap.get(javaType);
      if (map == null || map == NULL_TYPE_HANDLER_MAP) {
        map = new HashMap<>();
        typeHandlerMap.put(javaType, map);
      }
      //给String类型添加一种jdbc处理器
      map.put(jdbcType, handler);
    }
    //注册处理器实例
    allTypeHandlersMap.put(handler.getClass(), handler);
  }
```

默认会放置一些常用的类型处理器

```
register(Boolean.class, new BooleanTypeHandler());
    register(boolean.class, new BooleanTypeHandler());
    register(JdbcType.BOOLEAN, new BooleanTypeHandler());
    register(JdbcType.BIT, new BooleanTypeHandler());

    register(Byte.class, new ByteTypeHandler());
    register(byte.class, new ByteTypeHandler());
```

```java
    register(JdbcType.TINYINT, new ByteTypeHandler());

    register(Short.class, new ShortTypeHandler());
    register(short.class, new ShortTypeHandler());
    register(JdbcType.SMALLINT, new ShortTypeHandler());

    register(Integer.class, new IntegerTypeHandler());
    register(int.class, new IntegerTypeHandler());
    register(JdbcType.INTEGER, new IntegerTypeHandler());

    register(Long.class, new LongTypeHandler());
    register(long.class, new LongTypeHandler());

    register(Float.class, new FloatTypeHandler());
    register(float.class, new FloatTypeHandler());
    register(JdbcType.FLOAT, new FloatTypeHandler());

    register(Double.class, new DoubleTypeHandler());
    register(double.class, new DoubleTypeHandler());
    register(JdbcType.DOUBLE, new DoubleTypeHandler());

    register(Reader.class, new ClobReaderTypeHandler());
    register(String.class, new StringTypeHandler());
    register(String.class, JdbcType.CHAR, new StringTypeHandler());
    register(String.class, JdbcType.CLOB, new ClobTypeHandler());
    register(String.class, JdbcType.VARCHAR, new StringTypeHandler());
    register(String.class, JdbcType.LONGVARCHAR, new StringTypeHandler());
    register(String.class, JdbcType.NVARCHAR, new NStringTypeHandler());
    register(String.class, JdbcType.NCHAR, new NStringTypeHandler());
    register(String.class, JdbcType.NCLOB, new NClobTypeHandler());
    register(JdbcType.CHAR, new StringTypeHandler());
    register(JdbcType.VARCHAR, new StringTypeHandler());
    register(JdbcType.CLOB, new ClobTypeHandler());
    register(JdbcType.LONGVARCHAR, new StringTypeHandler());
    register(JdbcType.NVARCHAR, new NStringTypeHandler());
    register(JdbcType.NCHAR, new NStringTypeHandler());
    register(JdbcType.NCLOB, new NClobTypeHandler());

    register(Object.class, JdbcType.ARRAY, new ArrayTypeHandler());
    register(JdbcType.ARRAY, new ArrayTypeHandler());

    register(BigInteger.class, new BigIntegerTypeHandler());
    register(JdbcType.BIGINT, new LongTypeHandler());

    register(BigDecimal.class, new BigDecimalTypeHandler());
    register(JdbcType.REAL, new BigDecimalTypeHandler());
    register(JdbcType.DECIMAL, new BigDecimalTypeHandler());
    register(JdbcType.NUMERIC, new BigDecimalTypeHandler());

    register(InputStream.class, new BlobInputStreamTypeHandler());
    register(Byte[].class, new ByteObjectArrayTypeHandler());
    register(Byte[].class, JdbcType.BLOB, new
BlobByteObjectArrayTypeHandler());
    register(Byte[].class, JdbcType.LONGVARBINARY, new
BlobByteObjectArrayTypeHandler());
    register(byte[].class, new ByteArrayTypeHandler());
    register(byte[].class, JdbcType.BLOB, new BlobTypeHandler());
    register(byte[].class, JdbcType.LONGVARBINARY, new BlobTypeHandler());
    register(JdbcType.LONGVARBINARY, new BlobTypeHandler());
    register(JdbcType.BLOB, new BlobTypeHandler());

    register(Object.class, unknownTypeHandler);
    register(Object.class, JdbcType.OTHER, unknownTypeHandler);
    register(JdbcType.OTHER, unknownTypeHandler);

    register(Date.class, new DateTypeHandler());
    register(Date.class, JdbcType.DATE, new DateOnlyTypeHandler());
    register(Date.class, JdbcType.TIME, new TimeOnlyTypeHandler());
    register(JdbcType.TIMESTAMP, new DateTypeHandler());
    register(JdbcType.DATE, new DateOnlyTypeHandler());
    register(JdbcType.TIME, new TimeOnlyTypeHandler());

    register(java.sql.Date.class, new SqlDateTypeHandler());
    register(java.sql.Time.class, new SqlTimeTypeHandler());
    register(java.sql.Timestamp.class, new SqlTimestampTypeHandler());

    register(String.class, JdbcType.SQLXML, new SqlxmlTypeHandler());

    register(Instant.class, new InstantTypeHandler());
    register(LocalDateTime.class, new LocalDateTimeTypeHandler());
    register(LocalDate.class, new LocalDateTypeHandler());
    register(LocalTime.class, new LocalTimeTypeHandler());
    register(OffsetDateTime.class, new OffsetDateTimeTypeHandler());
    register(OffsetTime.class, new OffsetTimeTypeHandler());
    register(ZonedDateTime.class, new ZonedDateTimeTypeHandler());
    register(Month.class, new MonthTypeHandler());
    register(Year.class, new YearTypeHandler());
    register(YearMonth.class, new YearMonthTypeHandler());
    register(JapaneseDate.class, new JapaneseDateTypeHandler());
```

```
        // issue #273
        register(Character.class, new CharacterTypeHandler());
        register(char.class, new CharacterTypeHandler());
```

所有的处理器对应实例都会以class作为类名，实例作为value，放在实例映射表里allTypeHandlersMap。

> 处理器类。会在getInstance方法里进行初始化。就是反射。

4.3.4、plugins

> 是个常用的功能。可以让开发插手数据处理。最常用的就是分页，id过滤。

xml样例

```
    <property name="plugins">
        <array>
            <bean id="myBatisIDInterceptor"
    class="com.uteam.zen.core.interceptors.MyBatisIDInterceptor"/>
        </array>
    </property>
```

自定义id填充的拦截器

```
    //拦截Executor.class接口。的update类型。参数为args的。
    @Intercepts({@Signature(type = Executor.class, method = "update", args =
    {MappedStatement.class, Object.class})})
    public class MyBatisIDInterceptor implements Interceptor {

        /**
         * 单个插入名称
         */
        private static final String INSERT = "insert";
        /**
         * 批量插入名称
         */
        private static final String BATCH_INSERT = "batchInsert";


        @Override
        public Object intercept(final Invocation invocation) throws Throwable {
```

```
        MappedStatement mappedStatement = (MappedStatement)
invocation.getArgs()[0];

        // 获取 SQL
        SqlCommandType sqlCommandType =
mappedStatement.getSqlCommandType();

        // 不是 insert 类型的跳过
        if (!SqlCommandType.INSERT.equals(sqlCommandType)) {
            return invocation.proceed();
        }


        // 获取参数
        Map parameter = (Map) invocation.getArgs()[1];

        //拦截sql是insert或batchInsert的
        if (mappedStatement.getId().contains(INSERT)) {
            Object vo = parameter.get("vo");
            generatedKey(vo);
        } else if (mappedStatement.getId().contains(BATCH_INSERT)) {
            // 获取批量查询的参数并生成主键
            Object list = parameter.get("list");
            if (list instanceof ArrayList) {
                for (Object o : (ArrayList) list) {
                    generatedKey(o);
                }
            }
        }
        return invocation.proceed();
    }

    private void generatedKey(Object parameter) throws Throwable {

        List<Field> fieldList = new ArrayList<>();
        Class tempClass = parameter.getClass();

        //获取该实体类的字段,包括该实体类的父类,当父类为null的时候说明到达了最上层的父
类(Object类).
        while (tempClass != null &&
!"java.lang.object".equals(tempClass.getName().toLowerCase())) {
            fieldList.addAll(Arrays.asList(tempClass.getDeclaredFields()));
            //得到父类,然后赋给自己
            tempClass = tempClass.getSuperclass();
        }
        //遍历所有的DO属性，使用自定义注解Automatic指定主键
        for (Field field : fieldList) {
            if (!field.isAnnotationPresent(Automatic.class))
                continue;
            field.setAccessible(true);
            if (field.get(parameter) == null ||
StringUtils.isBlank((String) (field.get(parameter)))) {
                // 这里设置uuid
                field.set(parameter, IDUtils.createUUID());
            }
        }
    }

    @Override
    public Object plugin(final Object o) {
        return Plugin.wrap(o, this);
    }

    @Override
    public void setProperties(final Properties properties) {

    }
}
```

添加进拦截器链中。源码就很简单。

```
  public void addInterceptor(Interceptor interceptor) {
    interceptorChain.addInterceptor(interceptor);
  }

public class InterceptorChain {

  //就是个list集合
  private final List<Interceptor> interceptors = new ArrayList<>();

  public Object pluginAll(Object target) {
    for (Interceptor interceptor : interceptors) {
      target = interceptor.plugin(target);
    }
    return target;
  }
  public void addInterceptor(Interceptor interceptor) {
    interceptors.add(interceptor);
```

```
      }
      public List<Interceptor> getInterceptors() {
        //读取不可修改
        return Collections.unmodifiableList(interceptors);
      }

    }
```

4.3.5、mapper【重点】

> 读取mapper.xml。解析每一个select、insert、update、delete节点。一个节点生成一个MappedStatement对象。注册到Configuration中。可以是mapper的namespace+节点id

> 这里是解析方法的总入口

```
      public void parse() {
        if (!configuration.isResourceLoaded(resource)) {
          //这里是真正的解析
          configurationElement(parser.evalNode("/mapper"));
          //把已加载的资源放入Configuration中
          configuration.addLoadedResource(resource);
          //绑定名称空间和dao/mapper接口
          bindMapperForNamespace();
        }

        //处理待处理ResultMaps、CacheRefs、Statements
        //就是上面解析步骤里，有继承关系。不能生成的在这里生成。
        parsePendingResultMaps();
        parsePendingCacheRefs();
        parsePendingStatements();
      }
```

> 真正的解析会解析不同的模块

```
      private void configurationElement(XNode context) {
        try {
          //解析名称空间
          //也就是dao/mapper接口的全类名
          String namespace = context.getStringAttribute("namespace");
          if (namespace == null || namespace.equals("")) {
            throw new BuilderException("Mapper's namespace cannot be empty");
          }
          //设置到当前的名称空间中
          builderAssistant.setCurrentNamespace(namespace);
          //解析缓存引用
          cacheRefElement(context.evalNode("cache-ref"));
          //解析缓存
          cacheElement(context.evalNode("cache"));
          //解析parameterMap
          parameterMapElement(context.evalNodes("/mapper/parameterMap"));
          //解析resultMap
          resultMapElements(context.evalNodes("/mapper/resultMap"));
          //解析sql节点
          sqlElement(context.evalNodes("/mapper/sql"));
          //解析select|insert|update|delete节点

     buildStatementFromContext(context.evalNodes("select|insert|update|delete")
    );
        } catch (Exception e) {
          throw new BuilderException("Error parsing Mapper XML. The XML
    location is '" + resource + "'. Cause: " + e, e);
        }
      }
```

缓存cache

> mapper级别的缓存
>
> 通过在mapper文件中声明开启二级缓存。
>
> 这里单独讲解析。使用会后面讲。

cache解析

```
  private void cacheElement(XNode context) {
    if (context != null) {
      //获取缓存的实例类型，在Configuration初始化的时候注册
      // typeAliasRegistry.registerAlias("PERPETUAL",
PerpetualCache.class);
      // typeAliasRegistry.registerAlias("FIFO", FifoCache.class);
      // typeAliasRegistry.registerAlias("LRU", LruCache.class);
      String type = context.getStringAttribute("type", "PERPETUAL");
```

```java
      Class<? extends Cache> typeClass =
typeAliasRegistry.resolveAlias(type);
      //LRU回收算法
      String eviction = context.getStringAttribute("eviction", "LRU");
      Class<? extends Cache> evictionClass =
typeAliasRegistry.resolveAlias(eviction);
      //刷新间隔
      Long flushInterval = context.getLongAttribute("flushInterval");
      //大小
      Integer size = context.getIntAttribute("size");
      //是否只读
      boolean readWrite = !context.getBooleanAttribute("readOnly", false);
      //是否阻塞
      boolean blocking = context.getBooleanAttribute("blocking", false);
      //获取子节点name-value存放属性
      Properties props = context.getChildrenAsProperties();
      //生成一个Cache对象放在Configuration中
      builderAssistant.useNewCache(typeClass, evictionClass, flushInterval,
size, readWrite, blocking, props);
    }
  }
```

```java
  public Cache useNewCache(Class<? extends Cache> typeClass,
      Class<? extends Cache> evictionClass,
      Long flushInterval,
      Integer size,
      boolean readWrite,
      boolean blocking,
      Properties props) {
    //就是构造者模式-塞值-build
    //使用当前的名称空间-也就是dao/mapper全类名
    Cache cache = new CacheBuilder(currentNamespace)
        .implementation(valueOrDefault(typeClass, PerpetualCache.class))
        .addDecorator(valueOrDefault(evictionClass, LruCache.class))
        .clearInterval(flushInterval)
        .size(size)
        .readWrite(readWrite)
        .blocking(blocking)
        .properties(props)
        .build();
    //添加进Configuration
    configuration.addCache(cache);
    currentCache = cache;
    return cache;
  }
```

cache-ref 解析

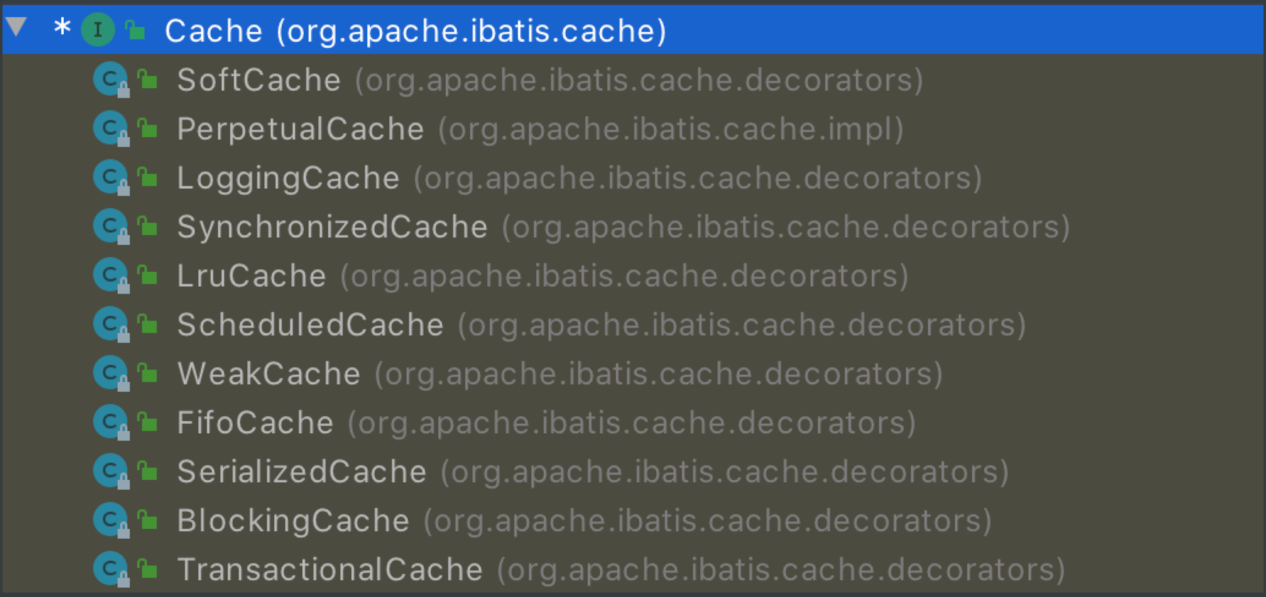
> cache-ref 指的是cache的引用。
>
> 简单理解就是两个名称空间的缓存合并。在一个名称空间上建cache【AuthorityDAO】。在另一个名称空间比如【AuthorityExtDAO】上使用

```
  private void cacheRefElement(XNode context) {
    if (context != null) {
      //添加一个从当前名称空间到目标名称空间的缓存引用
      configuration.addCacheRef(builderAssistant.getCurrentNamespace(),
context.getStringAttribute("namespace"));
      //创建一个缓存引用的处理器进行操作
      CacheRefResolver cacheRefResolver = new
CacheRefResolver(builderAssistant,
context.getStringAttribute("namespace"));
      try {
        //分析cache-ref
        cacheRefResolver.resolveCacheRef();
      } catch (IncompleteElementException e) {
        //如果上面操作不了说明目标cache没有创建完成。
        //放在未完成中，后续操作。
        configuration.addIncompleteCacheRef(cacheRefResolver);
      }
    }
  }
```

```
  public Cache resolveCacheRef() {
    //从mapper构建助手那里获取已经构建的目标名称空间的cache
    return assistant.useCacheRef(cacheRefNamespace);
  }
  public Cache useCacheRef(String namespace) {
    //校验目标名称空间
    if (namespace == null) {
      throw new BuilderException("cache-ref element requires a namespace
attribute.");
    }
    try {
      //获取名称空间
      unresolvedCacheRef = true;
```

```
        Cache cache = configuration.getCache(namespace);
        if (cache == null) {
          //说明cache还没有生成。后续处理
          throw new IncompleteElementException("No cache for namespace '" +
  namespace + "' could be found.");
        }
        currentCache = cache;
        unresolvedCacheRef = false;
        return cache;
      } catch (IllegalArgumentException e) {
        throw new IncompleteElementException("No cache for namespace '" +
  namespace + "' could be found.", e);
      }
    }
```

> 缓存实现算法有很多。默认是LRU最近最少使用。
>
> 下面是实现hierarchy.
>
> 缓存的源码。要后面再聊。



```
Cache (org.apache.ibatis.cache)
    SoftCache (org.apache.ibatis.cache.decorators)
    PerpetualCache (org.apache.ibatis.cache.impl)
    LoggingCache (org.apache.ibatis.cache.decorators)
    SynchronizedCache (org.apache.ibatis.cache.decorators)
    LruCache (org.apache.ibatis.cache.decorators)
    ScheduledCache (org.apache.ibatis.cache.decorators)
    WeakCache (org.apache.ibatis.cache.decorators)
    FifoCache (org.apache.ibatis.cache.decorators)
    SerializedCache (org.apache.ibatis.cache.decorators)
    BlockingCache (org.apache.ibatis.cache.decorators)
    TransactionalCache (org.apache.ibatis.cache.decorators)
```

解析parameterMap

> 暂时不看。官方已经不建议使用。

解析resultMap

对/mapper/resultMap节点逐个解析。

解析这样的元素。

```
<!--这里是resultMap-->
<resultMap type="com.uteam.zen.system.persistent.dataObject.AuthorityDo"
id="AuthorityResult">
  <!--这里是resultMapping-->
  <id property="id" column="id"/>
  <result property="name" column="name"/>
  <result property="code" column="code"/>
</resultMap>
```

```
  private void resultMapElements(List<XNode> list) throws Exception {
    //就是循环。啥也没有。
    for (XNode resultMapNode : list) {
      try {
        resultMapElement(resultMapNode);
      } catch (IncompleteElementException e) {
        //忽略未完成创建元素异常。
        // ignore, it will be retried
      }
    }
  }
```

```
  private ResultMap resultMapElement(XNode resultMapNode,
List<ResultMapping> additionalResultMappings, Class<?> enclosingType)
throws Exception {
    //把threadlocal中的正在创建对象改为当前节点
    //当前节点的唯一标识processing mapper_resultMap[AuthorityResult]
    //默认使用id。
    ErrorContext.instance().activity("processing " +
resultMapNode.getValueBasedIdentifier());
    //获取类型。优先级是 type -> ofType -> resultType -> javaType
    String type = resultMapNode.getStringAttribute("type",
        resultMapNode.getStringAttribute("ofType",
            resultMapNode.getStringAttribute("resultType",
                resultMapNode.getStringAttribute("javaType"))));
    //通过名称获取类class。
    //优先去alias里找。没有再使用Class.forName获取类。
    Class<?> typeClass = resolveClass(type);
    if (typeClass == null) {
      typeClass = inheritEnclosingType(resultMapNode, enclosingType);
    }
    //鉴别器-是一个children节点
    //根据某一列的字段值。对输出对象进行控制。比如组装结果不同。
    Discriminator discriminator = null;
    //resultMapping节点。就是行映射节点。
    List<ResultMapping> resultMappings = new ArrayList<>();
    resultMappings.addAll(additionalResultMappings);
    List<XNode> resultChildren = resultMapNode.getChildren();
    //取出所有的子节点
    for (XNode resultChild : resultChildren) {
      //如果当前节点名称是构造器
      if ("constructor".equals(resultChild.getName())) {
        //解析构造器节点中的每一个子节点。添加进resultMapping表中...
        processConstructorElement(resultChild, typeClass, resultMappings);
      } else if ("discriminator".equals(resultChild.getName())) {
        //解析鉴别器传唤类型和子节点
        discriminator = processDiscriminatorElement(resultChild, typeClass,
resultMappings);
      } else {
        //解析其他所有行。
        List<ResultFlag> flags = new ArrayList<>();
        if ("id".equals(resultChild.getName())) {
          flags.add(ResultFlag.ID);
        }
        resultMappings.add(buildResultMappingFromContext(resultChild,
typeClass, flags));
      }
    }
    String id = resultMapNode.getStringAttribute("id",
            resultMapNode.getValueBasedIdentifier());
    //获取当前节点的继承节点
    String extend = resultMapNode.getStringAttribute("extends");
    //resultMap节点是否开启自动mapping
    Boolean autoMapping = resultMapNode.getBooleanAttribute("autoMapping");
    //创建一个resultMap的构建器
    ResultMapResolver resultMapResolver = new
ResultMapResolver(builderAssistant, id, typeClass, extend, discriminator,
resultMappings, autoMapping);
    try {
      //执行构建
      return resultMapResolver.resolve();
    } catch (IncompleteElementException  e) {
      configuration.addIncompleteResultMap(resultMapResolver);
      throw e;
```

```
    }
  }
```

resultMap构建方法

```
  public ResultMap addResultMap(
      String id,
      Class<?> type,
      String extend,
      Discriminator discriminator,
      List<ResultMapping> resultMappings,
      Boolean autoMapping) {
    //namespace+id
    id = applyCurrentNamespace(id, false);
    //继承
    extend = applyCurrentNamespace(extend, true);

    if (extend != null) {
      //存在继承。但是继承目标没有生成。
      //暂停。等待后续处理。
      if (!configuration.hasResultMap(extend)) {
        throw new IncompleteElementException("Could not find a parent
resultmap with id '" + extend + "'");
      }
      //获取继承目标
      ResultMap resultMap = configuration.getResultMap(extend);
      List<ResultMapping> extendedResultMappings = new ArrayList<>
(resultMap.getResultMappings());
      //获取父类所有的resultMapping。需要覆盖的，就删除掉。只保留继承部分。
      extendedResultMappings.removeAll(resultMappings);
      // Remove parent constructor if this resultMap declares a
constructor.
      // 如果继承父类构造器。删除父类构造器
      boolean declaresConstructor = false;
      for (ResultMapping resultMapping : resultMappings) {
        if (resultMapping.getFlags().contains(ResultFlag.CONSTRUCTOR)) {
          declaresConstructor = true;
          break;
        }
      }
      if (declaresConstructor) {
        extendedResultMappings.removeIf(resultMapping ->
resultMapping.getFlags().contains(ResultFlag.CONSTRUCTOR));
      }
      resultMappings.addAll(extendedResultMappings);
    }
    //构造一个resultMap。放入Configuration的resultMap中。
    ResultMap resultMap = new ResultMap.Builder(configuration, id, type,
resultMappings, autoMapping)
        .discriminator(discriminator)
        .build();
    configuration.addResultMap(resultMap);
    return resultMap;
  }
```

resultMapping具体节点解析

> Todo: 这里还有一些字段不知道什么意思。等具体用到的时候。在记录。

```
  private ResultMapping buildResultMappingFromContext(XNode context,
Class<?> resultType, List<ResultFlag> flags) throws Exception {
    //构造器的行和普通mapping行不同。
    //构造器的行就获取name。普通获取行获取property。
    String property;
    if (flags.contains(ResultFlag.CONSTRUCTOR)) {
      property = context.getStringAttribute("name");
    } else {
      property = context.getStringAttribute("property");
    }
    //列名
    String column = context.getStringAttribute("column");
    //java类型
    String javaType = context.getStringAttribute("javaType");
    //jdbc类型
    String jdbcType = context.getStringAttribute("jdbcType");
    String nestedSelect = context.getStringAttribute("select");
    //一个resultMapping可以对应一个resultMap。内部映射。
    //只有association, collection, case节点才会生成内部映射，其他不生成
    String nestedResultMap = context.getStringAttribute("resultMap",
        //解析里面的每一行
        processNestedResultMappings(context, Collections.emptyList(),
resultType));
```

```
    String notNullColumn = context.getStringAttribute("notNullColumn");
    //列前缀
    String columnPrefix = context.getStringAttribute("columnPrefix");
    //指定类型转换器
    String typeHandler = context.getStringAttribute("typeHandler");
    String resultSet = context.getStringAttribute("resultSet");
    String foreignColumn = context.getStringAttribute("foreignColumn");
    boolean lazy = "lazy".equals(context.getStringAttribute("fetchType",
configuration.isLazyLoadingEnabled() ? "lazy" : "eager"));
    //解析java类型class
    Class<?> javaTypeClass = resolveClass(javaType);
    //解析TypeHandler类型class
    Class<? extends TypeHandler<?>> typeHandlerClass =
resolveClass(typeHandler);
    //解析jdbc类型
    JdbcType jdbcTypeEnum = resolveJdbcType(jdbcType);
    //通过resultMapping构造者模式build一个resultMapping。放入Configuration的
resultMapping中。
    //这里的构建暂时不看。
    return builderAssistant.buildResultMapping(resultType, property,
column, javaTypeClass, jdbcTypeEnum, nestedSelect, nestedResultMap,
notNullColumn, columnPrefix, typeHandlerClass, flags, resultSet,
foreignColumn, lazy);
  }
```

解析sql标签

> sql标签可以将重复的sql提取出来，使用的时候使用include引入。
>
> 生成名称空间+节点id，放入sqlFragments容器。

```
  private void sqlElement(List<XNode> list, String requiredDatabaseId) {
    for (XNode context : list) {
      String databaseId = context.getStringAttribute("databaseId");
      String id = context.getStringAttribute("id");
      id = builderAssistant.applyCurrentNamespace(id, false);
      if (databaseIdMatchesCurrent(id, databaseId, requiredDatabaseId)) {
        //map的存放。
        //value的类型是XNode
        sqlFragments.put(id, context);
      }
    }
  }
```

解析select|insert|update|delete

> 动态sql解析是mybatis的核心。
>
> 拥有不同的动态标签，比如choose、foreach、if、set等。mybatis把他们封装成不同的对象。共同接口都是SqlNode

```
        <select id="query" resultMap="AuthorityResult">
            select
            <include refid="columnsSql"/>
            from authority
            <include refid="whereSql"/>
        </select>
```

每个节点这样解析

```
     public void parseStatementNode() {
        //获取节点id
        String id = context.getStringAttribute("id");
        String databaseId = context.getStringAttribute("databaseId");

        if (!databaseIdMatchesCurrent(id, databaseId, this.requiredDatabaseId))
    {
          return;
        }
        //获取节点名称。就是标签名。select|insert|update|delete.
        String nodeName = context.getNode().getNodeName();
        //转换成小写英文。再转换成sql命令类型枚举SqlCommandType
        SqlCommandType sqlCommandType =
    SqlCommandType.valueOf(nodeName.toUpperCase(Locale.ENGLISH));
        //判断是否是查询
        boolean isSelect = sqlCommandType == SqlCommandType.SELECT;
        //是否刷新
        boolean flushCache = context.getBooleanAttribute("flushCache",
    !isSelect);
        //是否使用缓存
        boolean useCache = context.getBooleanAttribute("useCache", isSelect);
        //结果是否排序
        boolean resultOrdered = context.getBooleanAttribute("resultOrdered",
    false);
    //解析之前先把include的sql片段放进去
    // Include Fragments before parsing
    XMLIncludeTransformer includeParser = new
XMLIncludeTransformer(configuration, builderAssistant);
    includeParser.applyIncludes(context.getNode());

    //参数类型，获取java class
    String parameterType = context.getStringAttribute("parameterType");
    Class<?> parameterTypeClass = resolveClass(parameterType);
    //获取指定的语言驱动。没有就使用default。
    String lang = context.getStringAttribute("lang");
    LanguageDriver langDriver = getLanguageDriver(lang);

    // selectKey指定了id生成模式
    // Parse selectKey after includes and remove them.
    processSelectKeyNodes(id, parameterTypeClass, langDriver);

    // 指定主键生成器
    // Parse the SQL (pre: <selectKey> and <include> were parsed and
removed)
    KeyGenerator keyGenerator;
    String keyStatementId = id + SelectKeyGenerator.SELECT_KEY_SUFFIX;
    keyStatementId = builderAssistant.applyCurrentNamespace(keyStatementId,
true);
    if (configuration.hasKeyGenerator(keyStatementId)) {
      keyGenerator = configuration.getKeyGenerator(keyStatementId);
    } else {
      keyGenerator = context.getBooleanAttribute("useGeneratedKeys",
          configuration.isUseGeneratedKeys() &&
SqlCommandType.INSERT.equals(sqlCommandType))
          ? Jdbc3KeyGenerator.INSTANCE : NoKeyGenerator.INSTANCE;
    }

    //使用语言驱动。生成sqlSource。
    SqlSource sqlSource = langDriver.createSqlSource(configuration,
context, parameterTypeClass);
    //指定statement类型。默认PREPARED
    StatementType statementType =
StatementType.valueOf(context.getStringAttribute("statementType",
StatementType.PREPARED.toString()));
    //一次性取出所有sql，可能会导致oom。
    //这里指定每次返回最大条数。最终结果条数还是以查询为准
    Integer fetchSize = context.getIntAttribute("fetchSize");
    //超时时间
    Integer timeout = context.getIntAttribute("timeout");
    //参数表
    String parameterMap = context.getStringAttribute("parameterMap");
    //返回类型class
    String resultType = context.getStringAttribute("resultType");
    Class<?> resultTypeClass = resolveClass(resultType);
    //返回的resultMap
    String resultMap = context.getStringAttribute("resultMap");
    String resultSetType = context.getStringAttribute("resultSetType");
    // ResultSet.TYPE_FORWORD_ONLY 结果集的游标只能向下滚动。
    // ResultSet.TYPE_SCROLL_INSENSITIVE 结果集的游标可以上下移动，当数据库变化
时，当前结果集不变。
    // ResultSet.TYPE_SCROLL_SENSITIVE 返回可滚动的结果集，当数据库变化时，当前结
果集同步改变。
    ResultSetType resultSetTypeEnum = resolveResultSetType(resultSetType);
    if (resultSetTypeEnum == null) {
      resultSetTypeEnum = configuration.getDefaultResultSetType();
    }
    String keyProperty = context.getStringAttribute("keyProperty");
    String keyColumn = context.getStringAttribute("keyColumn");
    String resultSets = context.getStringAttribute("resultSets");
    //生成mappedStatement放入Configuration中。
    builderAssistant.addMappedStatement(id, sqlSource, statementType,
sqlCommandType,
        fetchSize, timeout, parameterMap, parameterTypeClass, resultMap,
resultTypeClass,
        resultSetTypeEnum, flushCache, useCache, resultOrdered,
        keyGenerator, keyProperty, keyColumn, databaseId, langDriver,
resultSets);
  }
```

SqlSource生成

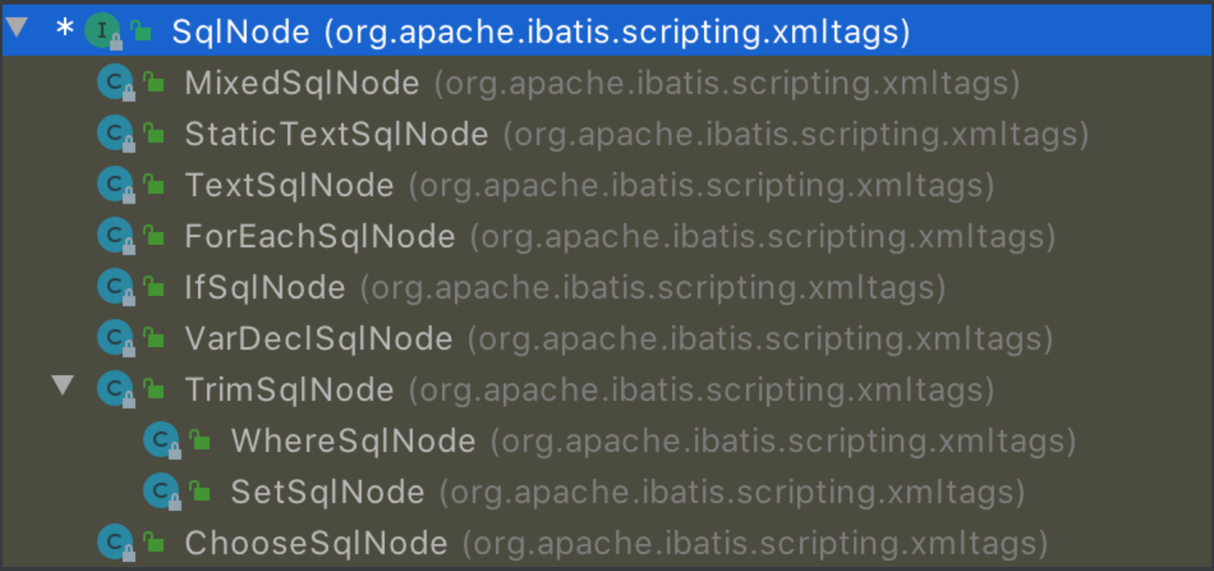
> SqlSource sqlSource = langDriver.createSqlSource(configuration, context, parameterTypeClass);

- 创建建造者

```
  public XMLScriptBuilder(Configuration configuration, XNode context,
  Class<?> parameterType) {
    //创建父建造者BaseBuilder。包有Configuration，typeAlias，typeHandler
    super(configuration);
    this.context = context;//当前节点
    this.parameterType = parameterType;//参数类型java class
```

```
      //初始化节点处理器
      initNodeHandlerMap();
    }
    //针对每一种标签。都有对应的处理器。
    private void initNodeHandlerMap() {
      nodeHandlerMap.put("trim", new TrimHandler());
      nodeHandlerMap.put("where", new WhereHandler());
      nodeHandlerMap.put("set", new SetHandler());
      nodeHandlerMap.put("foreach", new ForEachHandler());
      nodeHandlerMap.put("if", new IfHandler());
      nodeHandlerMap.put("choose", new ChooseHandler());
      nodeHandlerMap.put("when", new IfHandler());
      nodeHandlerMap.put("otherwise", new OtherwiseHandler());
      nodeHandlerMap.put("bind", new BindHandler());
    }
```

sqlNode节点hierarchy。



```
▼ * SqlNode (org.apache.ibatis.scripting.xmltags)
      MixedSqlNode (org.apache.ibatis.scripting.xmltags)
      StaticTextSqlNode (org.apache.ibatis.scripting.xmltags)
      TextSqlNode (org.apache.ibatis.scripting.xmltags)
      ForEachSqlNode (org.apache.ibatis.scripting.xmltags)
      IfSqlNode (org.apache.ibatis.scripting.xmltags)
      VarDeclSqlNode (org.apache.ibatis.scripting.xmltags)
  ▼   TrimSqlNode (org.apache.ibatis.scripting.xmltags)
          WhereSqlNode (org.apache.ibatis.scripting.xmltags)
          SetSqlNode (org.apache.ibatis.scripting.xmltags)
      ChooseSqlNode (org.apache.ibatis.scripting.xmltags)
```

解析入口

```
    public SqlSource parseScriptNode() {
      //解析为SqlNode的list集合
      MixedSqlNode rootSqlNode = parseDynamicTags(context);
      SqlSource sqlSource;
      if (isDynamic) {
        sqlSource = new DynamicSqlSource(configuration, rootSqlNode);
      } else {
        //生成静态sqlSource的时候，会把'#{}'转换为'?'
  //通过【ParameterMappingTokenHandler】把'#{}'添加到StaticSqlSource的
  parameterMappings参数列表中。
        sqlSource = new RawSqlSource(configuration, rootSqlNode,
  parameterType);
      }
      return sqlSource;
    }
```

- 组装MixedSqlNode

> MixedSqlNode parseDynamicTags(context)
>
> 【xml语法】用于范围查询。如果想通过>=或者<=或者<或者>查询符合范围的数据。
>
> 但是如果写入的sql语句有类似<或者>或者&特殊字符，在解析xml文件的时候会被转义，需要用来解决。
>
> ```
> <![CDATA[and DATE_FORMAT(CREATE_DATA,"%Y-%m-%d") >= #
> {startDate}]]>
> 或者
> and natural_length<![CDATA[>=]]>#{naturalLengthStart}
> ......
> ```

```
    protected MixedSqlNode parseDynamicTags(XNode node) {
      //这就是MixedSqlNode的sqlNode节点集合
      List<SqlNode> contents = new ArrayList<>();
      //当前节点的所有子节点
      NodeList children = node.getNode().getChildNodes();
      for (int i = 0; i < children.getLength(); i++) {
        //每个子节点都生成一个新的节点引用
        //把当前节点的路径解析器和参数传进去
        XNode child = node.newXNode(children.item(i));
```

```
      if (child.getNode().getNodeType() == Node.CDATA_SECTION_NODE ||
child.getNode().getNodeType() == Node.TEXT_NODE) {
        //如果当前节点只包含文本，生成一个TextSqlNode
        String data = child.getStringBody("");
        TextSqlNode textSqlNode = new TextSqlNode(data);
        //GenericTokenParser判断是通过当前节点有没有openToken'#{'和
closeToken'}'判断的
        if (textSqlNode.isDynamic()) {
          contents.add(textSqlNode);
          isDynamic = true;
        } else {
          //生成静态sqlNode
          contents.add(new StaticTextSqlNode(data));
        }
      } else if (child.getNode().getNodeType() == Node.ELEMENT_NODE) {
// issue #628
        //如果是有xml标签的node。交给初始化的nodeHandler处理。
        //同时被认为是动态的。
        String nodeName = child.getNode().getNodeName();
        //获取标签节点处理器
        NodeHandler handler = nodeHandlerMap.get(nodeName);
        if (handler == null) {
          throw new BuilderException("Unknown element <" + nodeName +
"> in SQL statement.");
        }
        //处理。并放入sql片段集合中
        handler.handleNode(child, contents);
        isDynamic = true;
      }
    }
    return new MixedSqlNode(StaticSqlSource);
  }
```

- 动态sql解析

```
<!--以这个update为例-->
<update id="update">
  update authority
    <set>
        <if test="vo.name != null">name=#{vo.name},</if>
        <if test="vo.code != null">code=#{vo.code},</if>
    </set>
  where id=#{vo.id}
</update>
```

> 猜测应该会分为三个节点。
>
> 两个静态节点和一个set动态节点；set节点又分为两个if动态节点。

- 第一个child：[#text: \n   update authority   \n]
- 第二个child: [set: null]

> 调用SetHandler.handleNode(child, contents);

> 静态sql还是照常解析
> [#text: \n   ]
>
> > [if: null]   name='if',value='name=#{vo.name},'
>
> [#text: \n   ]
>
> [if: null]   name='if',value='name=#{vo.code},'
>
> [#text: \n   ]

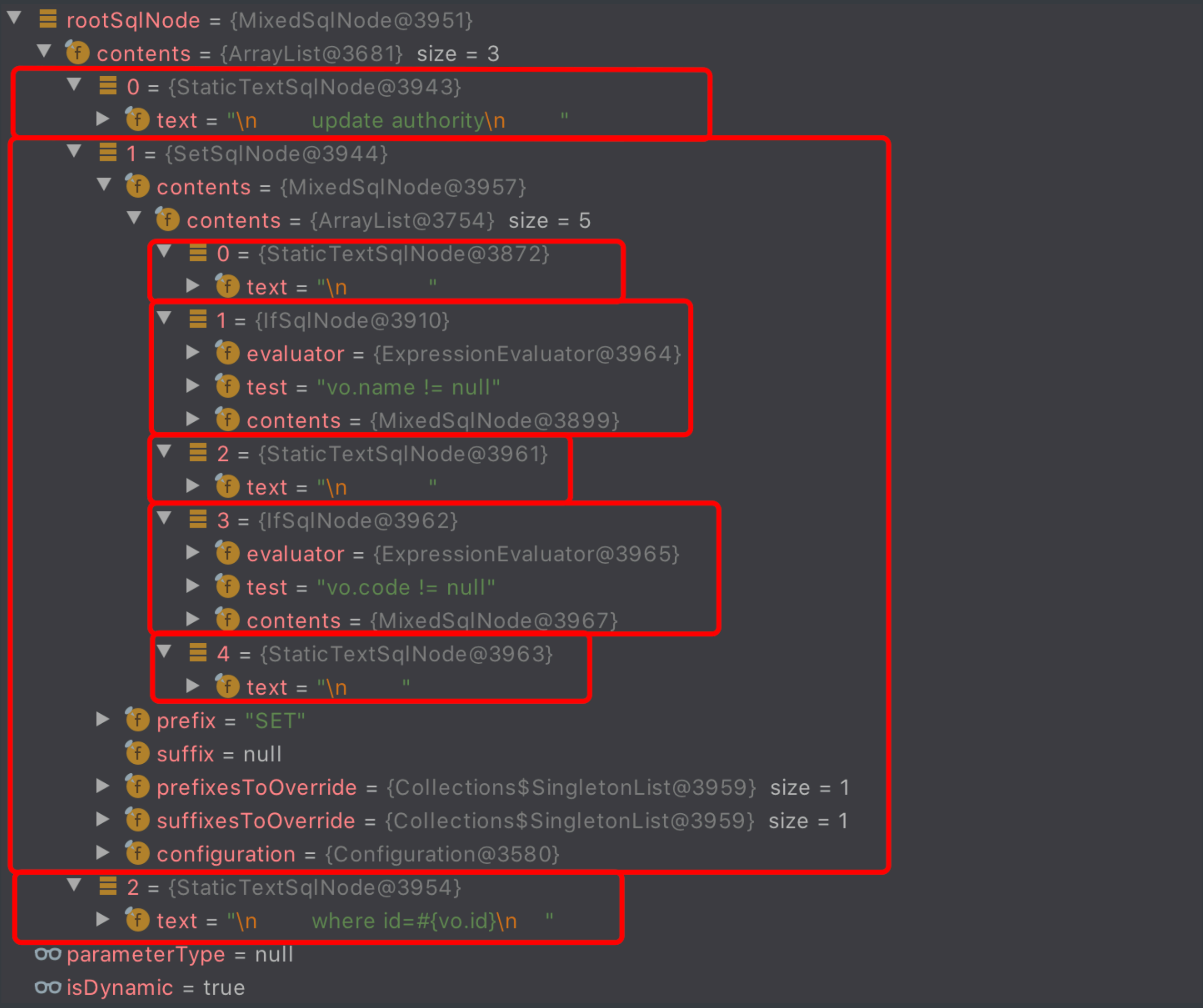
```
    @Override
    //SetHandler
    public void handleNode(XNode nodeToHandle, List<SqlNode>
targetContents) {
      //回调parseDynamicTags。处理整个set节点
      //回调的时候。就会使用IfHandler处理两个if节点
      MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);
      //封装成一个SetSqlNode,放入集合
      SetSqlNode set = new SetSqlNode(configuration, mixedSqlNode);
      targetContents.add(set);
    }
```

```
        @Override
        //IfHandler
        public void handleNode(XNode nodeToHandle, List<SqlNode>
    targetContents) {
          //回调parseDynamicTags。
          MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);
          //获取test属性。即条件表达式
          String test = nodeToHandle.getStringAttribute("test");
          //组装成IfSqlNode，放入集合
          IfSqlNode ifSqlNode = new IfSqlNode(mixedSqlNode, test);
          targetContents.add(ifSqlNode);
        }
```

```
    - 第三个child:  [#text:  \n     where id=#{vo.id}      \n ]
```

这样就解析完成

> 完了生成DynamicSqlSource。



```
▼ rootSqlNode = {MixedSqlNode@3951}
  ▼ contents = {ArrayList@3681}  size = 3
    ▼ 0 = {StaticTextSqlNode@3943}
      ▶ text = "\n        update authority\n        "
    ▼ 1 = {SetSqlNode@3944}
      ▼ contents = {MixedSqlNode@3957}
        ▼ contents = {ArrayList@3754}  size = 5
          ▼ 0 = {StaticTextSqlNode@3872}
            ▶ text = "\n            "
          ▼ 1 = {IfSqlNode@3910}
            ▶ evaluator = {ExpressionEvaluator@3964}
            ▶ test = "vo.name != null"
            ▶ contents = {MixedSqlNode@3899}
          ▼ 2 = {StaticTextSqlNode@3961}
            ▶ text = "\n            "
          ▼ 3 = {IfSqlNode@3962}
            ▶ evaluator = {ExpressionEvaluator@3965}
            ▶ test = "vo.code != null"
            ▶ contents = {MixedSqlNode@3967}
          ▼ 4 = {StaticTextSqlNode@3963}
            ▶ text = "\n        "
      ▶ prefix = "SET"
        suffix = null
      ▶ prefixesToOverride = {Collections$SingletonList@3959}  size = 1
      ▶ suffixesToOverride = {Collections$SingletonList@3959}  size = 1
      ▶ configuration = {Configuration@3580}
    ▼ 2 = {StaticTextSqlNode@3954}
      ▶ text = "\n        where id=#{vo.id}\n    "
  parameterType = null
  isDynamic = true
```

- 生成MappedStatement对象。

```java
  public MappedStatement addMappedStatement(/****参数太多****/) {

    if (unresolvedCacheRef) {
      throw new IncompleteElementException("Cache-ref not yet
resolved");
    }
    //namespace+方法名
    id = applyCurrentNamespace(id, false);
    //是否为查询语句
    boolean isSelect = sqlCommandType == SqlCommandType.SELECT;

    //调用建造者
    MappedStatement.Builder statementBuilder = new
MappedStatement.Builder(configuration, id, sqlSource, sqlCommandType)
        .resource(resource)
        .fetchSize(fetchSize)
        .timeout(timeout)
```

```
          .statementType(statementType)
          .keyGenerator(keyGenerator)
          .keyProperty(keyProperty)
          .keyColumn(keyColumn)
          .databaseId(databaseId)
          .lang(lang)
          .resultOrdered(resultOrdered)
          .resultSets(resultSets)
          .resultMaps(getStatementResultMaps(resultMap, resultType, id))
          .resultSetType(resultSetType)
          .flushCacheRequired(valueOrDefault(flushCache, !isSelect))
          .useCache(valueOrDefault(useCache, isSelect))
          .cache(currentCache);
      //配置参数列表#{vo.name},#{vo.code}...
      ParameterMap statementParameterMap =
  getStatementParameterMap(parameterMap, parameterType, id);
      if (statementParameterMap != null) {
        statementBuilder.parameterMap(statementParameterMap);
      }
      //构建。
      MappedStatement statement = statementBuilder.build();
      configuration.addMappedStatement(statement);
      return statement;
    }
```

4.4、注解驱动

> 从MapperAnnotationBuilder开始。暂不整理。

5、mapper加载与调用

> mapper的接口层，前面提到有两种方式的调用。
>
> 一个是mybatis提供的api
>
> ```
>     String statement = "XXXXX.dao.XXMapper/Dao.getXXXList";
>     List<T> result = sqlsession.selectList(statement);
> ```
>
> 一个是spring-mybatis提供的service层注入mapper

> ```
> @Repository
> public interface AuthorityDao extends
> IPersistentDao<AuthorityDo,String> {
> //super.delete
> }
> @Service
> @Component
> public class AuthorityServiceImpl implements IAuthorityService {
>
>  @Autowired
>  private AuthorityDao authorityDao;
>
>  @Override
>  public void delAuthority(String id) {
>      authorityDao.delete(id);
>  }
> }
> ```

5.1、spring-mybatis配置

> MapperScannerConfigurer是一个BeanDefinitionRegistryPostProcessor【bean定义信息注册的后置处理器】，
>
> 可以为容器注入bean的定义信息。

```

    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <!--扫描的基包路径-->
        <property name="basePackage"
value="com.uteam.zen.system.persistent.dao"/>
        <!--通过指定注解类，进行过滤。自定义注解也可以。-->
        <!--当然不使用注解也可以。但要保证包下的所有类都要注册。-->
        <property name="annotationClass"
value="org.springframework.stereotype.Repository"/>
        <property name="sqlSessionFactoryBeanName"
value="sqlSessionFactoryBean"/>
    </bean>
```

5.2、扫包

直接看MapperScannerConfigurer的postProcessBeanDefinitionRegistry()方法。

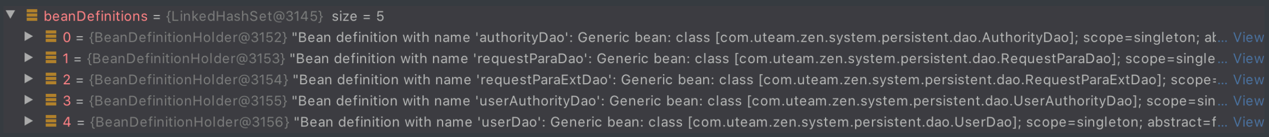
```
  @Override
  public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry
registry) {
    if (this.processPropertyPlaceHolders) {
      //容器刷新
      //->invokeBeanFactoryPostProcessors()
      //->PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors
      //实现BeanDefinitionRegistry的MapperScannerConfigurer优先调用。
      //BeanFactoryPostProcessor类型的PropertyResourceConfigurer最后调用。
      //所以MapperScannerConfigurer的任何属性替换都将失败。
      //如果有属性替换，先加载PropertyResourceConfigurer。执行他的
postProcessBeanFactory()
      //再把basePackage、sqlSessionFactoryBeanName、
sqlSessionTemplateBeanName等属性放进去
      processPropertyPlaceHolders();
    }

    //创建扫描器
    ClassPathMapperScanner scanner = new ClassPathMapperScanner(registry);
    scanner.setAddToConfig(this.addToConfig);
    //这是配置的注解class
    scanner.setAnnotationClass(this.annotationClass);
    scanner.setMarkerInterface(this.markerInterface);
    scanner.setSqlSessionFactory(this.sqlSessionFactory);
    scanner.setSqlSessionTemplate(this.sqlSessionTemplate);
    //sqlSession工厂bean的名称
    scanner.setSqlSessionFactoryBeanName(this.sqlSessionFactoryBeanName);
    scanner.setSqlSessionTemplateBeanName(this.sqlSessionTemplateBeanName);
    scanner.setResourceLoader(this.applicationContext);
    scanner.setBeanNameGenerator(this.nameGenerator);
    //给扫描器注册过滤器。
    //包括基于annotationClass的过滤器AnnotationTypeFilter
    scanner.registerFilters();
    //使用分隔符，string2Array。
    //扫描
    scanner.scan(StringUtils.tokenizeToStringArray(this.basePackage,
ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS));
  }
```

扫描就是调用super.doScan(basePackages);

也就是ClassPathBeanDefinitionScanner的doScan方法。在spring注解驱动的AnnotationConfigApplicationContext里有用到。

扫描到的bean定义信息的包装类集合。


```
▼ beanDefinitions = {LinkedHashSet@3145}  size = 5
  ▶ 0 = {BeanDefinitionHolder@3152} "Bean definition with name 'authorityDao': Generic bean: class [com.uteam.zen.system.persistent.dao.AuthorityDao]; scope=singleton; at... View
  ▶ 1 = {BeanDefinitionHolder@3153} "Bean definition with name 'requestParaDao': Generic bean: class [com.uteam.zen.system.persistent.dao.RequestParaDao]; scope=single... View
  ▶ 2 = {BeanDefinitionHolder@3154} "Bean definition with name 'requestParaExtDao': Generic bean: class [com.uteam.zen.system.persistent.dao.RequestParaExtDao]; scope=... View
  ▶ 3 = {BeanDefinitionHolder@3155} "Bean definition with name 'userAuthorityDao': Generic bean: class [com.uteam.zen.system.persistent.dao.UserAuthorityDao]; scope=sin... View
  ▶ 4 = {BeanDefinitionHolder@3156} "Bean definition with name 'userDao': Generic bean: class [com.uteam.zen.system.persistent.dao.UserDao]; scope=singleton; abstract=f... View
```

5.3、注册BeanDefinition

```
  private void processBeanDefinitions(Set<BeanDefinitionHolder>
beanDefinitions) {
    GenericBeanDefinition definition;
    for (BeanDefinitionHolder holder : beanDefinitions) {
      //从包装类中获取每一个bean定义信息
      definition = (GenericBeanDefinition) holder.getBeanDefinition();

      if (logger.isDebugEnabled()) {
        logger.debug("Creating MapperFactoryBean with name '" +
holder.getBeanName()
          + "' and '" + definition.getBeanClassName() + "'
mapperInterface");
      }

      // the mapper interface is the original class of the bean
      // but, the actual class of the bean is MapperFactoryBean

      //bean名称

 definition.getConstructorArgumentValues().addGenericArgumentValue(definiti
on.getBeanClassName()); // issue #59
      //bean的class为mapperFactoryBean!
      //实际初始化的是mapperFactoryBean实例
      //bean的name则是XXXXMapper/Dao
      definition.setBeanClass(this.mapperFactoryBean.getClass());

      definition.getPropertyValues().add("addToConfig", this.addToConfig);

      boolean explicitFactoryUsed = false;
      if (StringUtils.hasText(this.sqlSessionFactoryBeanName)) {
        //把sqlSessionFactory放到定义信息的属性列表中。
        //在当前bean初始化的时候，会调用sqlSessionFactory的set方法。
        //value就是new
RuntimeBeanReference(this.sqlSessionFactoryBeanName)。引用。
        //注意！这里调用的是mapperFactoryBean的setSqlSessionFactory()方法。
        definition.getPropertyValues().add("sqlSessionFactory", new
RuntimeBeanReference(this.sqlSessionFactoryBeanName));
```

```
        explicitFactoryUsed = true;
      } else if (this.sqlSessionFactory != null) {
        definition.getPropertyValues().add("sqlSessionFactory",
this.sqlSessionFactory);
        explicitFactoryUsed = true;
      }

      if (StringUtils.hasText(this.sqlSessionTemplateBeanName)) {
        if (explicitFactoryUsed) {
          logger.warn("Cannot use both: sqlSessionTemplate and
sqlSessionFactory together. sqlSessionFactory is ignored.");
        }
        definition.getPropertyValues().add("sqlSessionTemplate", new
RuntimeBeanReference(this.sqlSessionTemplateBeanName));
        explicitFactoryUsed = true;
      } else if (this.sqlSessionTemplate != null) {
        if (explicitFactoryUsed) {
          logger.warn("Cannot use both: sqlSessionTemplate and
sqlSessionFactory together. sqlSessionFactory is ignored.");
        }
        definition.getPropertyValues().add("sqlSessionTemplate",
this.sqlSessionTemplate);
        explicitFactoryUsed = true;
      }

      if (!explicitFactoryUsed) {
        if (logger.isDebugEnabled()) {
          logger.debug("Enabling autowire by type for MapperFactoryBean
with name '" + holder.getBeanName() + "'.");
        }

 definition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_TYPE);
      }
    }
  }
```

5.4 setSqlSessionFactory

> MapperFactoryBean里没有。在父类SqlSessionDaoSupport里找到了【笑哭】

```
  public void setSqlSessionFactory(SqlSessionFactory sqlSessionFactory) {
    if (!this.externalSqlSession) {
      //生成了一个SqlSessionTemplate模板
      this.sqlSession = new SqlSessionTemplate(sqlSessionFactory);
    }
  }
```

SqlSessionTemplate构造方法

```
  public SqlSessionTemplate(SqlSessionFactory sqlSessionFactory,
ExecutorType executorType) {
    this(sqlSessionFactory, executorType,
        //这是一个异常翻译器。
        //把mysql的异常翻译成mybatis封装异常。吧?
        new MyBatisExceptionTranslator(

 sqlSessionFactory.getConfiguration().getEnvironment().getDataSource(),
true));
  }

  public SqlSessionTemplate(SqlSessionFactory sqlSessionFactory,
ExecutorType executorType,
      PersistenceExceptionTranslator exceptionTranslator) {
    //类型校验
    notNull(sqlSessionFactory, "Property 'sqlSessionFactory' is required");
    notNull(executorType, "Property 'executorType' is required");

    //设置SqlSessionFactory
    this.sqlSessionFactory = sqlSessionFactory;
    //设置执行器类型，默认是SIMPLE
    this.executorType = executorType;
    //设置异常翻译器
    this.exceptionTranslator = exceptionTranslator;
    //设置sqlSession代理，使用了JDK反射包的Proxy。
    this.sqlSessionProxy = (SqlSession) newProxyInstance(
        SqlSessionFactory.class.getClassLoader(),
        new Class[] { SqlSession.class },
        new SqlSessionInterceptor());
  }
```

> sqlSession代理意思就是。调用SqlSession的时候，实际执行他的代理类的SqlSessionInterceptor【继承InvocationHandler】的invoke方法。

> 在sqlSession的具体方法调用前后，进行openSession，closeSession等操作。

5.5、MapperFactoryBean的实例化

> FactoryBean工厂bean的实例化，返回的不是this，而是getObject的返回。

```
@Override
public T getObject() throws Exception {
  //getSqlSession()返回的就是刚刚的sqlSessionTemplate
  //getMapper()返回的是mapper接口的代理对象
  return getSqlSession().getMapper(this.mapperInterface);
}
```

```
//sqlSessionTemplate.getMapper()
@Override
public <T> T getMapper(Class<T> type) {
  return getConfiguration().getMapper(type, this);
}
//conguration.getMapper()
public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
  return mapperRegistry.getMapper(type, sqlSession);
}
```

MapperRegistry是configuration对象。

里面保存了mapper.class->MapperProxyFactory<mapper.class>实例。

通过实例创建mapperProxy代理对象。

```
public class MapperRegistry {

  private final Configuration config;
  //这里存放了mapper/dao的class和MapperProxyFactory集合。
  //每一个value都是一个MapperProxyFactory实例。泛型是mapper.class
  private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new
HashMap<>();

  public MapperRegistry(Configuration config) {
    this.config = config;
  }

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

```
  public T newInstance(SqlSession sqlSession) {
    //创建代理。
    //MapperProxy实现了InvocationHandler接口
    //methodCache是一个成员ConcurrentHashMap
    final MapperProxy<T> mapperProxy = new MapperProxy<>(sqlSession,
mapperInterface, methodCache);
    //使用jdk生成代理
    //(T) Proxy.newProxyInstance(
    //        mapperInterface.getClassLoader(),
    //        new Class[] { mapperInterface },
    //        mapperProxy);
    return newInstance(mapperProxy);
  }
```

这样就生成了mapper的代理。调用mapper，进入mapperProxy的invoke方法。

> methodCache为每一个mepper目标方法。都提供了一个MapperMethod的缓存。
>
> 下面说具体调用。

5.6、mapper调用

> MapperProxy代理类的invoke方法。

```
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
  Throwable {
      try {
        //如果不是代理。而是。实现类，就直接调用
        if (Object.class.equals(method.getDeclaringClass())) {
          return method.invoke(this, args);
        } else if (method.isDefault()) {
          if (privateLookupInMethod == null) {
            return invokeDefaultMethodJava8(proxy, method, args);
          } else {
            return invokeDefaultMethodJava9(proxy, method, args);
          }
        }
      } catch (Throwable t) {
        throw ExceptionUtil.unwrapThrowable(t);
      }
      //从mapperMethod缓存中，获取目标执行方法。
      final MapperMethod mapperMethod = cachedMapperMethod(method);
      //执行
      return mapperMethod.execute(sqlSession, args);
    }
```

5.6.1、MapperMethod

> 两个成员。
>
> SqlCommand：包含了执行方法名称name和方法类型SqlCommandType
>
> MethodSignature：方法签名信息。
>
> ```
>     private final boolean returnsMany;//是否返回多条也就是list
>     private final boolean returnsMap;//是否返回map
>     private final boolean returnsVoid;//是否void
>     private final boolean returnsCursor;//是否返回游标
>     private final boolean returnsOptional;//是否返回Optional
>     private final Class<?> returnType;//返回类型
>     private final String mapKey;
>     private final Integer resultHandlerIndex;
>     private final Integer rowBoundsIndex;
>     private final ParamNameResolver paramNameResolver;//参数名解析器
> ```

```
public class MapperMethod {
  //成员
  private final SqlCommand command;
  private final MethodSignature method;
  //构造器
  public MapperMethod(Class<?> mapperInterface, Method method,
Configuration config) {
    this.command = new SqlCommand(config, mapperInterface, method);
    this.method = new MethodSignature(config, mapperInterface, method);
  }
  //执行
  public Object execute(SqlSession sqlSession, Object[] args) {
    Object result;
    switch (command.getType()) {
      //增
      case INSERT: {
        Object param = method.convertArgsToSqlCommandParam(args);
        result = rowCountResult(sqlSession.insert(command.getName(),
param));
        break;
      }
      //改
      case UPDATE: {
        Object param = method.convertArgsToSqlCommandParam(args);
        result = rowCountResult(sqlSession.update(command.getName(),
param));
        break;
      }
      //删
      case DELETE: {
        Object param = method.convertArgsToSqlCommandParam(args);
        result = rowCountResult(sqlSession.delete(command.getName(),
param));
        break;
      }
      //查
      case SELECT:
        if (method.returnsVoid() && method.hasResultHandler()) {
          //无返回值
          executeWithResultHandler(sqlSession, args);
          result = null;
        } else if (method.returnsMany()) {
          //返回list
          result = executeForMany(sqlSession, args);
        } else if (method.returnsMap()) {
          //返回map类型
          result = executeForMap(sqlSession, args);
        } else if (method.returnsCursor()) {
          //返回游标
          result = executeForCursor(sqlSession, args);
        } else {
          //返回对象
          Object param = method.convertArgsToSqlCommandParam(args);
          result = sqlSession.selectOne(command.getName(), param);
          if (method.returnsOptional()
              && (result == null ||
!method.getReturnType().equals(result.getClass()))) {
            result = Optional.ofNullable(result);
          }
        }
        break;
      //这个还真没用过。先记一下
      case FLUSH:
        result = sqlSession.flushStatements();
        break;
      default:
        throw new BindingException("Unknown execution method for: " +
command.getName());
    }
    if (result == null && method.getReturnType().isPrimitive() &&
!method.returnsVoid()) {
      throw new BindingException("Mapper method '" + command.getName()
          + " attempted to return null from a method with a primitive
return type (" + method.getReturnType() + ").");
    }
    return result;
  }
  //------------------------方法省略---------------------------
}
```

> 参数解析：
>
> 优先使用@Param注解声明的名称，再使用param作为前缀+参数序号
>
> 比方说：@Param("id") String id ,传参1，就会被解析为[id:'1',params:'1']

5.6.2、SqlSession执行

> sqlSession.selectOne是具体的执行sql方法，并完成返回转换。
>
> sqlSession就是前面SqlSessionFactory生成的SqlSessionTemplate，里面有sqlSessionProxy代理。
>
> 调用到了SqlSessionInterceptor.invoke()方法。

```
  private class SqlSessionInterceptor implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        //通过DefaultSqlSessionFactory的实例获取sqlSession。这里稍后再说。
        //返回了一个DefaultSqlSession对象
        SqlSession sqlSession = getSqlSession(
            //sqlSession工厂
            SqlSessionTemplate.this.sqlSessionFactory,
            //执行器类型 SIMPLE
            SqlSessionTemplate.this.executorType,
            //异常翻译器
            SqlSessionTemplate.this.exceptionTranslator);
        try {
          //执行sql
          Object result = method.invoke(sqlSession, args);
          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) {
          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((P
ersistenceException) unwrapped);
          if (translated != null) {
            unwrapped = translated;
          }
        }
        throw unwrapped;
      } finally {
        if (sqlSession != null) {
          closeSqlSession(sqlSession,
SqlSessionTemplate.this.sqlSessionFactory);
        }
      }
    }
  }
```

> 通过反射调用DefaultSqlSession的selectOne()方法
>
> selectOne也是调用selectList
>
> ```
> public <T> T selectOne(String statement, Object parameter) {
>  // Popular vote was to return null on 0 results and throw
> exception on too many.
>  // 实际调用selectList
>  // 使用默认的行界限 offset=0,limit=Integer.MAX_VAUE
>  List<T> list = this.selectList(statement, parameter);
>  if (list.size() == 1) {
>    return list.get(0);
>  } else if (list.size() > 1) {
>    //返回多条就抛异常
>    throw new TooManyResultsException("Expected one result (or null)
> to be returned by selectOne(), but found: " + list.size());
>  } else {
>    return null;
>  }
> }
> ```

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

```
      //statement='com.uteam.zen.system.persistent.dao.AuthorityDao.load'
      //从configuration中获取之前封装好的MappedStatement
      MappedStatement ms = configuration.getMappedStatement(statement);
      //把对象或数组对象包裹一下。
      //参数实现了collection接口，存放'collection'->obj
      //参数实现了List接口，存放'list'->obj
      //参数是数组，存放'array'->obj
      //如果进行了包裹，就返回StrictMap对象【严格之处在于，重写父类get,获取不到就抛异
常BindingException】
      //执行SimpleExecutor的query
      return executor.query(ms, wrapCollection(parameter), rowBounds,
Executor.NO_RESULT_HANDLER);
    } catch (Exception e) {
      throw ExceptionFactory.wrapException("Error querying database.
Cause: " + e, e);
    } finally {
      ErrorContext.instance().reset();
    }
  }
```

> 这个Executor。因为设置了，所以使用SimpleExecutor。如果改为使用二级缓存，这里就是使用CachingExecutor。

executor.query

```
  //BaseExecutor
  @Override
  public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler) throws SQLException {
    //获取BoundSql
    BoundSql boundSql = ms.getBoundSql(parameter);
    //根据statement,参数，boundSql生成缓存key
  //458111696:1383925621:com.uteam.zen.system.persistent.dao.AuthorityDao.l
oad:0:2147483647:select id,name,code from authority where id =
?:1:SqlSessionFactoryBean
    CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);
    //查询
    return query(ms, parameter, rowBounds, resultHandler, key, boundSql);
  }
```

获取boundSql

> 之前已经在MappedStatement中生成了sqlSource.
>
> sqlSource 分为动态DynamicSqlSource和静态RowSqlSource。两个都保存了sqlNode节点。
>
> 也就是前面提到的MixedSqlNode【sql片段集合】

BoundSql结构

```
public class BoundSql {
  //动态生成的sql，不是#{},而是带有? 占位符的sql，能够像jdbc一样被PreparedStatement
执行的sql
  private final String sql;
  //每个参数的信息。参数名称，参数TypeHandler，参数JDBC类型...
  private final List<ParameterMapping> parameterMappings;
  //参数。前面生成的mapperMethod里的ParamMap
  private final Object parameterObject;
  //暂时不知道
  private final Map<String, Object> additionalParameters;
  private final MetaObject metaParameters;
}
```

动态sql

> StringJoiner是java8提供的一个util类，对字符串进行拼接，指定分隔，指定拼接前缀后缀。
>
> 省的每人都写一遍StringBuilder。内部使用StringBuilder封装，线程不安全。

```
  @Override
  public BoundSql getBoundSql(Object parameterObject) {
    //组装动态上下文，包括参数paramMap，参数使用到的
TypeHandler,databaseId,StringJoiner...
    DynamicContext context = new DynamicContext(configuration,
parameterObject);
    //使用组合模式。多个sqlNode节点组成MixedSqlNode
    //调用MixedSqlNode.apply()是顺序执行每一个sqlNode.apply()
    //结果在context.getSql得到完整sql
    rootSqlNode.apply(context);
    //sqlSourceBuilder是sql的解析器。
    //有参数解析ParameterMappingTokenHandler
    //有GenericTokenParser对#{}进行占位符? 转换
    SqlSourceBuilder sqlSourceParser = new SqlSourceBuilder(configuration);
    Class<?> parameterType = parameterObject == null ? Object.class :
parameterObject.getClass();
```

```
    //sql解析。这里不进行占位符转换。仍然保留#{vo.name}
    //内部节点已经拼装完成
    //这里的sqlSource是静态sql【StaticSqlSource】
    SqlSource sqlSource = sqlSourceParser.parse(context.getSql(),
parameterType, context.getBindings());
    //创建BoundSql
    //new BoundSql(configuration, sql, parameterMappings, parameterObject);
    //这一步跟静态sql解析一样
    BoundSql boundSql = sqlSource.getBoundSql(parameterObject);
    context.getBindings().forEach(boundSql::setAdditionalParameter);
    return boundSql;
  }
```

静态sql

```
  public BoundSql getBoundSql(Object parameterObject) {
    //new BoundSql(configuration, sql, parameterMappings, parameterObject);
    return sqlSource.getBoundSql(parameterObject);
  }
```

> ParameterMappings是这样的集合，保存参数类型
>
> ParameterMapping{property='vo.name', mode=IN, javaType=class java.lang.Object, jdbcType=null, numericScale=null, resultMapId='null', jdbcTypeName='null', expression='null'}
>
> ParameterMapping{property='vo.code', mode=IN, javaType=class java.lang.Object, jdbcType=null, numericScale=null, resultMapId='null', jdbcTypeName='null', expression='null'}
>
> ParameterMapping{property='vo.id', mode=IN, javaType=class java.lang.Object, jdbcType=null, numericScale=null, resultMapId='null', jdbcTypeName='null', expression='null'}

执行sql

CachingExecutor.query

> CachingExecutor重写了父类的query方法
>
> 里面的delegate是SimpleExecutor。

> 意思也就行CachingExecutor只处理缓存。具体操作还是调用SimpleExecutor操作数据库

```
  @Override
  public <E> List<E> query(MappedStatement ms, Object parameterObject,
RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql
boundSql)
      throws SQLException {
    //当前mapper是否需要缓存
    Cache cache = ms.getCache();
    if (cache != null) {
      //根据需要刷新缓存【是刷新缓存请求】
      flushCacheIfRequired(ms);
      if (ms.isUseCache() && resultHandler == null) {
        ensureNoOutParams(ms, boundSql);
        @SuppressWarnings("unchecked")
        //这里就是缓存tcm【TransactionalCacheManager】是个hashMap
        List<E> list = (List<E>) tcm.getObject(cache, key);
        if (list == null) {
          //调用SimpleExecutor执行查询
          list = delegate.query(ms, parameterObject, rowBounds,
resultHandler, key, boundSql);
          //放缓存
          tcm.putObject(cache, key, list); // issue #578 and #116
        }
        return list;
      }
    }
    //不过缓存。直接执行查询。
    return delegate.query(ms, parameterObject, rowBounds, resultHandler,
key, boundSql);
  }
```

SimpleExecutor.query

父类BaseExecutor

> 这里的缓存是PerpetualCache永久缓存。保存在执行器中。而执行器又跟sqlSession生命周期相同。
>
> 所以是session级别的一级永久缓存

```
  @Override
```

```java
  public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {
    ErrorContext.instance().resource(ms.getResource()).activity("executing a query").object(ms.getId());
    if (closed) {
      throw new ExecutorException("Executor was closed.");
    }
    //如果有刷新缓存请求。就清空本地缓存
    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 {
        //去数据库查询
        list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key, boundSql);
      }
    } finally {
      queryStack--;
    }
    if (queryStack == 0) {
      for (DeferredLoad deferredLoad : deferredLoads) {
        deferredLoad.load();
      }
      // issue #601
      deferredLoads.clear();
      if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
        // issue #482
        clearLocalCache();
      }
    }
    return list;
  }
```

数据库查询

```
  @Override
  public <E> List<E> doQuery(MappedStatement ms, Object parameter,
RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws
SQLException {
    Statement stmt = null;
    try {
      Configuration configuration = ms.getConfiguration();
      //获取statement的类型。默认是PreparedStatementHandler
      //注意。这里如果使用了插件。返回的是一个代理
      //wrapper是调用方。意思也就是如果是从CachingExecutor来的。wrapper就是
CachingExecutor对象
      StatementHandler handler = configuration.newStatementHandler(wrapper,
ms, parameter, rowBounds, resultHandler, boundSql);
      //创建prepareStatement对象。并设置参数值
      stmt = prepareStatement(handler, ms.getStatementLog());
      //执行sql，返回结果集
      return handler.query(stmt, resultHandler);
    } finally {
      closeStatement(stmt);
    }
  }
```

创建PrepareStatement

```
  private Statement prepareStatement(StatementHandler handler, Log
statementLog) throws SQLException {
    Statement stmt;
    //获取连接
    Connection connection = getConnection(statementLog);
    //预编译
    stmt = handler.prepare(connection, transaction.getTimeout());
    handler.parameterize(stmt);
    return stmt;
  }
```

> 获取连接。入参是个Log。是为了方便的打印日志。
>
> 返回一个代理。ConnectionLogger。

获取连接

```
    protected Connection getConnection(Log statementLog) throws SQLException
  {
      //从SpringManagedTransaction中获取连接
      //没有设置过TransactionFactory。默认就是SpringManagedTransactionFactory。
      Connection connection = transaction.getConnection();
      //如果日志输出级别是debug。返回ConnectionLogger代理。
      if (statementLog.isDebugEnabled()) {
        return ConnectionLogger.newInstance(connection, statementLog,
  queryStack);
      } else {
        return connection;
      }
    }
```

```
    //SpringManagedTransaction.getConnection();
    //dataSource就是xml里配置的mybatis提供的池化PooledDataSource。
    //当然也可以使用其他的数据库连接池。
    private void openConnection() throws SQLException {
      this.connection = DataSourceUtils.getConnection(this.dataSource);
      this.autoCommit = this.connection.getAutoCommit();
      this.isConnectionTransactional =
  DataSourceUtils.isConnectionTransactional(this.connection,
  this.dataSource);

      if (LOGGER.isDebugEnabled()) {
        LOGGER.debug(
            "JDBC Connection ["
                + this.connection
                + "] will"
                + (this.isConnectionTransactional ? " " : " not ")
                + "be managed by Spring");
      }
    }
```

然后jdk代理成ConnectionLogger对象。

```
  public final class ConnectionLogger extends BaseJdbcLogger implements
  InvocationHandler {
    //代理对象
    private final Connection connection;
    private ConnectionLogger(Connection conn, Log statementLog, int
  queryStack) {
      super(statementLog, queryStack);
      this.connection = conn;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] params)
        throws Throwable {
      try {
        //直接调用
        if (Object.class.equals(method.getDeclaringClass())) {
          return method.invoke(this, params);
        }
        //调用prepareStatement方法
        if ("prepareStatement".equals(method.getName())) {
          if (isDebugEnabled()) {
            debug(" Preparing: " + removeBreakingWhitespace((String)
  params[0]), true);
          }
          PreparedStatement stmt = (PreparedStatement)
  method.invoke(connection, params);
          //创建代理PreparedStatementLogger
          stmt = PreparedStatementLogger.newInstance(stmt, statementLog,
  queryStack);
          return stmt;
          //调用prepareCall
        } else if ("prepareCall".equals(method.getName())) {
          if (isDebugEnabled()) {
            debug(" Preparing: " + removeBreakingWhitespace((String)
  params[0]), true);
          }
          PreparedStatement stmt = (PreparedStatement)
  method.invoke(connection, params);
          //创建代理PreparedStatementLogger
          stmt = PreparedStatementLogger.newInstance(stmt, statementLog,
  queryStack);
          return stmt;
          //调用createStatement
        } else if ("createStatement".equals(method.getName())) {
          Statement stmt = (Statement) method.invoke(connection, params);
          //创建代理PreparedStatementLogger
          stmt = StatementLogger.newInstance(stmt, statementLog, queryStack);
          return stmt;
        } else {
          //其他不输出日志
          return method.invoke(connection, params);
        }
      } catch (Throwable t) {
        throw ExceptionUtil.unwrapThrowable(t);
      }
    }

    /**
     * Creates a logging version of a connection.
     *
     * @param conn - the original connection
     * @return - the connection with logging
     */
    public static Connection newInstance(Connection conn, Log statementLog,
  int queryStack) {
      InvocationHandler handler = new ConnectionLogger(conn, statementLog,
  queryStack);
      ClassLoader cl = Connection.class.getClassLoader();
      return (Connection) Proxy.newProxyInstance(cl, new Class[]
  {Connection.class}, handler);
    }

    /**
     * return the wrapped connection.
     *
     * @return the connection
     */
    public Connection getConnection() {
      return connection;
    }

  }
```

预编译

> 跟JDBC一样。拿到sql。调用collection的prepareStatement(sql)。

从

```
//SimpleExecutor.RoutingStatementHandler.prepare
stmt = handler.prepare(connection, transaction.getTimeout());
```

开始。经过statement路由到了PreparedStatementHandler。

> new PreparedStatementHandler(executor, ms, parameter, rowBounds, resultHandler, boundSql);

```
  @Override
  public Statement prepare(Connection connection, Integer
transactionTimeout) throws SQLException {
    //PreparedStatementHandler.prepare
    return delegate.prepare(connection, transactionTimeout);
  }
  //---------------------------------------------------------------------------
----------
  @Override
  public Statement prepare(Connection connection, Integer
transactionTimeout) throws SQLException {
    //当前上下文。标记正在处理的sql是boundSql.getSql()
    ErrorContext.instance().sql(boundSql.getSql());
    Statement statement = null;
    try {
      //预编译。这里调用collection.prepareStatement(sql)
      statement = instantiateStatement(connection);
      setStatementTimeout(statement, transactionTimeout);
      setFetchSize(statement);
      return statement;
    } catch (SQLException e) {
      closeStatement(statement);
      throw e;
    } catch (Exception e) {
      closeStatement(statement);
      throw new ExecutorException("Error preparing statement.  Cause: " +
e, e);
    }
  }
```

```
  @Override
  protected Statement instantiateStatement(Connection connection) throws
SQLException {
    //获取sql
    String sql = boundSql.getSql();
    //处理KeyGenerator
    if (mappedStatement.getKeyGenerator() instanceof Jdbc3KeyGenerator) {
      String[] keyColumnNames = mappedStatement.getKeyColumns();
      if (keyColumnNames == null) {
        return connection.prepareStatement(sql,
PreparedStatement.RETURN_GENERATED_KEYS);
      } else {
        return connection.prepareStatement(sql, keyColumnNames);
      }
    } else if (mappedStatement.getResultSetType() == ResultSetType.DEFAULT)
{
      //预编译sql
      //这里就是调用CollectionLogger代理的invoke方法。
      //返回PrepareStatement对象
      //又为PrepareStatement创建了PrepareStatementLogger代理
      return connection.prepareStatement(sql);
    } else {
      return connection.prepareStatement(sql,
mappedStatement.getResultSetType().getValue(), ResultSet.CONCUR_READ_ONLY);
    }
  }
```

设置参数

> handler.parameterize(stmt);
>
> 设置参数的时候有很多选项
>
> prepareStatement.setString()、prepareStatement.setInt()、prepareStatement.setFloat()、prepareStatement.setObject()
>
> 如果使用默认的DefaultParameterHandler参数处理器。

```
  @Override
  public void setParameters(PreparedStatement ps) {
    ErrorContext.instance().activity("setting
parameters").object(mappedStatement.getParameterMap().getId());
    //拿出所有的参数列表
    //ParameterMapping{property='id', mode=IN, javaType=class
java.lang.Object, jdbcType=null, numericScale=null, resultMapId='null',
jdbcTypeName='null', expression='null'}
    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;
          //获取属性名称  比如上面例子id
          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 {
            //从参数对象中获取值
            //parameterObject[id:1,param1:1]
            MetaObject metaObject =
configuration.newMetaObject(parameterObject);
            value = metaObject.getValue(propertyName);
          }
          //value=1
          //获取TypeHandler
          TypeHandler typeHandler = parameterMapping.getTypeHandler();
          //获取jdbc类型
          JdbcType jdbcType = parameterMapping.getJdbcType();
          //处理null
          if (value == null && jdbcType == null) {
            jdbcType = configuration.getJdbcTypeForNull();
          }
          try {
            //使用typeHandler塞值。
            typeHandler.setParameter(ps, i + 1, value, jdbcType);
          } catch (TypeException | SQLException e) {
            throw new TypeException("Could not set parameters for mapping:
" + parameterMapping + ". Cause: " + e, e);
          }
        }
      }
    }
  }
```

到这里就完成了全部sql的组装。

执行

```
  @Override
  public <E> List<E> query(Statement statement, ResultHandler
resultHandler) throws SQLException {
    //这里的PreparedStatement。也可能已经被PreparedStatementLogger包成了代理。用
于日志输出。
    PreparedStatement ps = (PreparedStatement) statement;
    //查询
    ps.execute();
    //处理结果集
    return resultSetHandler.handleResultSets(ps);
  }
```

处理返回集

> 默认采用DefaultResultSetHandler处理preparedStatement执行后的结果。
>
> resultSet相当于返回结果集的一个迭代器。
>
> jdbc中使用resultSet指向next节点。使用getString(列名)，getInt(列名)等方式获取每一行属性。

```
@Override
public List<Object> handleResultSets(Statement stmt) throws SQLException {
  //标记当前正在进行哪个mapper方法的结果处理。
  ErrorContext.instance().activity("handling
results").object(mappedStatement.getId());

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

  int resultSetCount = 0;
  //将resultSet包装成一个ResultSetWrapper对象
  //resultSet也是被ResultSetLogger的代理类
  ResultSetWrapper rsw = getFirstResultSet(stmt);

  //读取xml中的resultMaps.
  List<ResultMap> resultMaps = mappedStatement.getResultMaps();
  int resultMapCount = resultMaps.size();
  validateResultMapsCount(rsw, resultMapCount);
  while (rsw != null && resultMapCount > resultSetCount) {
    ResultMap resultMap = resultMaps.get(resultSetCount);
    //处理所有结果放入multipleResults集合
    //就是每一行的元素。按照ResultMap组装
    //分页在这里 跳过行。使用分页对象RowBounds。
```

```
      handleResultSet(rsw, resultMap, multipleResults, null);
      //rs.next
      rsw = getNextResultSet(stmt);
      cleanUpAfterHandlingResultSet();
      resultSetCount++;
    }
    //返回multipleResults
    return collapseSingleResultList(multipleResults);
  }
```

ResultSetWrapper对象

```
    public ResultSetWrapper(ResultSet rs, Configuration configuration) throws
  SQLException {
      super();
      //获取配置的所有类型处理器
      this.typeHandlerRegistry = configuration.getTypeHandlerRegistry();
      //当前的resultSet
      this.resultSet = rs;
      //当前的resultSet元数据 列名、列类型等
      final ResultSetMetaData metaData = rs.getMetaData();
      //列数
      final int columnCount = metaData.getColumnCount();
      //遍历所有列
      for (int i = 1; i <= columnCount; i++) {
        //添加进列名集合
        columnNames.add(configuration.isUseColumnLabel() ?
  metaData.getColumnLabel(i) : metaData.getColumnName(i));
        //添加进jdbc类型集合
        jdbcTypes.add(JdbcType.forCode(metaData.getColumnType(i)));
        //添加进javaClass集合
        classNames.add(metaData.getColumnClassName(i));
      }
    }
```

RowBounds

> RowBounds对象中有两个属性控制着分页：`offset、limit`。offset是说分页从第几条数据开始，limit是说一共取多少条数据。因为我们没有配置它，所以它默认是offset从0开始，limit取Int的最大值。
> 链接：https://www.jianshu.com/p/43f304e4b784
>
> 但是我们不会实际使用它。因为他是逻辑分页。不是物理分页。

```
public class RowBounds {
    public static final int NO_ROW_OFFSET = 0;
    public static final int NO_ROW_LIMIT = Integer.MAX_VALUE;
    public RowBounds() {
        this.offset = NO_ROW_OFFSET;
        this.limit = NO_ROW_LIMIT;
    }
}
```

6、插件代理过程

```
<property name="plugins">
  <array>
<bean id="myBatisIDInterceptor"
class="com.uteam.zen.core.interceptors.MyBatisIDInterceptor"/>
  </array>
</property>
```

> Plugins 是通过mybatis-config.xml或者spring-config的SqlSessionFactoryBean节点注入configuration的。
>
> 需要实现org.apache.ibatis.plugin.Interceptor接口。
>
> 同时。通过@Intercepts声明接口的名称，方法名称，参数列表
>
> ```
> @Intercepts({@Signature(type = Executor.class, method = "update",
> args = {MappedStatement.class, Object.class})})
> //这个就是声明了拦截接口为Executor，方法名为query，参数为args的方法。
> ```

6.1、Executor

- 在上面mapper加载的时候，会调用setSqlSessionFactory生成一个SqlSessionTemplate。用于sqlSession生成。

  成员sqlSessionProxy实际上是一个SqlSessionInterceptor。拦截sqlSession执行。

- mapper代理执行的时候，调用sqlSession执行selectList操作。
- 在SqlSessionInterceptor执行目标方法之前。进行了openSession，创建Executor。

```
SqlSession sqlSession = getSqlSession(
          SqlSessionTemplate.this.sqlSessionFactory,
          SqlSessionTemplate.this.executorType,
          SqlSessionTemplate.this.exceptionTranslator);
```

getSqlSession具体代码调用如下

```
//SqlSessionUtils
public static SqlSession getSqlSession(SqlSessionFactory
sessionFactory, ExecutorType executorType,
PersistenceExceptionTranslator exceptionTranslator) {
//通过session工厂和【执行器类型】获取session
session = sessionFactory.openSession(executorType);
}
//---------------------------------------------------------------------
---------------
//DefaultSqlSessionFactory
@Override
public SqlSession openSession(ExecutorType execType) {
  return openSessionFromDataSource(execType, null, false);
}
  private SqlSession openSessionFromDataSource(ExecutorType execType,
TransactionIsolationLevel level, boolean autoCommit) {
    Transaction tx = null;
    try {
      final Environment environment = configuration.getEnvironment();
      final TransactionFactory transactionFactory =
getTransactionFactoryFromEnvironment(environment);
      tx =
transactionFactory.newTransaction(environment.getDataSource(), level,
autoCommit);
      //在这里创建了sqlSession里的执行器
      final Executor executor = configuration.newExecutor(tx,
execType);
      return new DefaultSqlSession(configuration, executor,
autoCommit);
    } catch (Exception e) {
      closeTransaction(tx); // may have fetched a connection so lets
call close()
      throw ExceptionFactory.wrapException("Error opening session.
Cause: " + e, e);
    } finally {
      ErrorContext.instance().reset();
```

```
        }
      }
    //-----------------------------------------------------------------------
    ---------------
    public Executor newExecutor(Transaction transaction, ExecutorType
    executorType) {
        executorType = executorType == null ? defaultExecutorType :
    executorType;
        executorType = executorType == null ? ExecutorType.SIMPLE :
    executorType;
        Executor executor;
        if (ExecutorType.BATCH == executorType) {
          executor = new BatchExecutor(this, transaction);
        } else if (ExecutorType.REUSE == executorType) {
          executor = new ReuseExecutor(this, transaction);
        } else {
          //默认的执行器
          executor = new SimpleExecutor(this, transaction);
        }
        if (cacheEnabled) {
          //开启缓存默认为true，把SimpleExecutor包装成CachingExecutor
          executor = new CachingExecutor(executor);
        }
        //如果配置了插件。包装执行器，返回executor的代理对象。
        executor = (Executor) interceptorChain.pluginAll(executor);
        return executor;
      }
```

executor代理创建过程

```
      public Object pluginAll(Object target) {
        //所有的插件。遍历。包裹。
        for (Interceptor interceptor : interceptors) {
          target = interceptor.plugin(target);
        }
        return target;
      }
```

```
      default Object plugin(Object target) {
        //执行当前插件和executor对象的wrap。
        return Plugin.wrap(target, this);
      }
```

包装

```java
    public static Object wrap(Object target, Interceptor interceptor) {
      //获取@Signature注解的接口，方法和参数。
      Map<Class<?>, Set<Method>> signatureMap =
getSignatureMap(interceptor);
      //当前执行器类型
      Class<?> type = target.getClass();
      //获取目标类实现的接口
      Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
      if (interfaces.length > 0) {
        //使用jdk代理。
        //代理类就是本类Plugin。实现了InvocationHandler接口。
        return Proxy.newProxyInstance(
            type.getClassLoader(),
            interfaces,
            new Plugin(target, interceptor, signatureMap));
      }
      return target;
    }
```

执行

```java
  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    try {
      //判断当前执行方法是否在拦截器注解范围内
      Set<Method> methods = signatureMap.get(method.getDeclaringClass());
      if (methods != null && methods.contains(method)) {
        //执行拦截器
        return interceptor.intercept(new Invocation(target, method, args));
      }
      //否则直接执行目标方法
      return method.invoke(target, args);
    } catch (Exception e) {
      throw ExceptionUtil.unwrapThrowable(e);
    }
  }
```

可以做自定义缓存的取数据拦截。不使用mapper级别的本地缓存。而是进行redis等的统一管理。

6.2、StatementHandler

> 执行SimpleExecutor的doQuery方法的时候，要创建StatementHandler对象。
>
> 也可以创建拦截器。
>
> ```
> //doQuery
> StatementHandler handler =
> configuration.newStatementHandler(wrapper, ms, parameter,
> rowBounds, resultHandler, boundSql);
>       stmt = prepareStatement(handler, ms.getStatementLog());
> //configuration.newStatementHandler
>   public StatementHandler newStatementHandler(Executor executor,
> MappedStatement mappedStatement, Object parameterObject, RowBounds
> rowBounds, ResultHandler resultHandler, BoundSql boundSql) {
>     StatementHandler statementHandler = new
> RoutingStatementHandler(executor, mappedStatement, parameterObject,
> rowBounds, resultHandler, boundSql);
>     //相同的代理创建逻辑
>     statementHandler = (StatementHandler)
> interceptorChain.pluginAll(statementHandler);
>     return statementHandler;
>   }
> //对RoutingStatementHandler进行代理
> ```

可以做自定义分页用。拦截自定义分页对象。添加limit和offset参数到sql语句中。

6.3、ResultSetHandler

> 6.3创建StatementHandler的时候，BaseStatementHandler会创建ResultSetHandler和ParameterHandler对象
>
> 分别对设置参数和结果解析进行代理wrap。
>
> 默认就是DefaultResultSetHandler和DefaultParameterHandler对象。

```
  public ParameterHandler newParameterHandler(MappedStatement
mappedStatement, Object parameterObject, BoundSql boundSql) {
    ParameterHandler parameterHandler =
mappedStatement.getLang().createParameterHandler(mappedStatement,
parameterObject, boundSql);
    //wrap
```

```
    parameterHandler = (ParameterHandler)
interceptorChain.pluginAll(parameterHandler);
    return parameterHandler;
  }

  public ResultSetHandler newResultSetHandler(Executor executor,
MappedStatement mappedStatement, RowBounds rowBounds, ParameterHandler
parameterHandler,
      ResultHandler resultHandler, BoundSql boundSql) {
    ResultSetHandler resultSetHandler = new
DefaultResultSetHandler(executor, mappedStatement, parameterHandler,
resultHandler, boundSql, rowBounds);
    //wrap
    resultSetHandler = (ResultSetHandler)
interceptorChain.pluginAll(resultSetHandler);
    return resultSetHandler;
  }
```

可以做自定义缓存的存数据拦截。不然。需要手动入缓存嘛。

如果做了整库的全量缓存的话。可能更需要。而我还不知道。

自定义缓存

> 参考文档https://www.jianshu.com/p/2aa8c4c61ef8

自定义分页

> 参考文档https://www.jianshu.com/p/2aa8c4c61ef8

mybatis 缓存实验

> 参考博客

> 你真的会用Mybatis的缓存么，不知道原理的话，容易踩坑哦
> https://www.jianshu.com/p/c553169c5921
> 从源码角度聊一聊JDBC和Mysql的预编译特性
> https://my.oschina.net/kailuncen/blog/905395

mybatis实现分库分表

mybatis实现读写分离