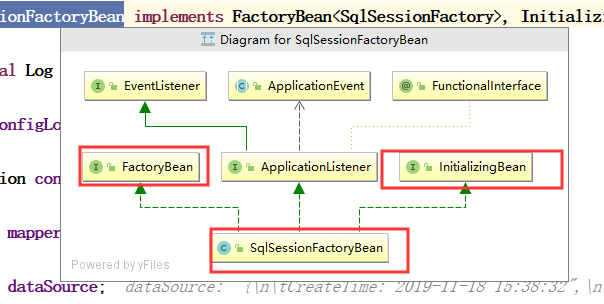
**Mybatis**

# SqlSessionFactoryBean

## 类图



## buildSqlSessionFactory

|  |
| --- |
| ***/\*\*  \* Build a {@code SqlSessionFactory} instance.  \*  \* The default implementation uses the standard MyBatis {@code XMLConfigBuilder} API to build a  \* {@code SqlSessionFactory} instance based on an Reader.  \* Since 1.3.0, it can be specified a {@link Configuration} instance directly(without config file).  \*  \* @return SqlSessionFactory  \* @throws IOException if loading the config file failed  \*/* protected SqlSessionFactory buildSqlSessionFactory() throws IOException {   Configuration configuration;   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) {  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");  }  configuration = new Configuration();  if (this.configurationProperties != null) {  configuration.setVariables(this.configurationProperties);  }  }   if (this.objectFactory != null) {  configuration.setObjectFactory(this.objectFactory);  }   if (this.objectWrapperFactory != null) {  configuration.setObjectWrapperFactory(this.objectWrapperFactory);  }   if (this.vfs != null) {  configuration.setVfsImpl(this.vfs);  }   if (*hasLength*(this.typeAliasesPackage)) {  String[] typeAliasPackageArray = *tokenizeToStringArray*(this.typeAliasesPackage,  ConfigurableApplicationContext.*CONFIG\_LOCATION\_DELIMITERS*);  for (String packageToScan : typeAliasPackageArray) {  configuration.getTypeAliasRegistry().registerAliases(packageToScan,  typeAliasesSuperType == null ? Object.class : typeAliasesSuperType);  if (*LOGGER*.isDebugEnabled()) {  *LOGGER*.debug("Scanned package: '" + packageToScan + "' for aliases");  }  }  }   if (!*isEmpty*(this.typeAliases)) {  for (Class<?> typeAlias : this.typeAliases) {  configuration.getTypeAliasRegistry().registerAlias(typeAlias);  if (*LOGGER*.isDebugEnabled()) {  *LOGGER*.debug("Registered type alias: '" + typeAlias + "'");  }  }  }   if (!*isEmpty*(this.plugins)) {  for (Interceptor plugin : this.plugins) {  configuration.addInterceptor(plugin);  if (*LOGGER*.isDebugEnabled()) {  *LOGGER*.debug("Registered plugin: '" + plugin + "'");  }  }  }   if (*hasLength*(this.typeHandlersPackage)) {  String[] typeHandlersPackageArray = *tokenizeToStringArray*(this.typeHandlersPackage,  ConfigurableApplicationContext.*CONFIG\_LOCATION\_DELIMITERS*);  for (String packageToScan : typeHandlersPackageArray) {  configuration.getTypeHandlerRegistry().register(packageToScan);  if (*LOGGER*.isDebugEnabled()) {  *LOGGER*.debug("Scanned package: '" + packageToScan + "' for type handlers");  }  }  }   if (!*isEmpty*(this.typeHandlers)) {  for (TypeHandler<?> typeHandler : this.typeHandlers) {  configuration.getTypeHandlerRegistry().register(typeHandler);  if (*LOGGER*.isDebugEnabled()) {  *LOGGER*.debug("Registered type handler: '" + typeHandler + "'");  }  }  }   if (this.databaseIdProvider != null) {*//fix #64 set databaseId before parse mapper xmls* try {  configuration.setDatabaseId(this.databaseIdProvider.getDatabaseId(this.dataSource));  } catch (SQLException e) {  throw new NestedIOException("Failed getting a databaseId", e);  }  }   if (this.cache != null) {  configuration.addCache(this.cache);  }   if (xmlConfigBuilder != null) {  try {  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();  }  }   if (this.transactionFactory == null) {  this.transactionFactory = new SpringManagedTransactionFactory();  } //设置环境  configuration.setEnvironment(new Environment(this.environment, this.transactionFactory, this.dataSource));   if (!*isEmpty*(this.mapperLocations)) {  for (Resource mapperLocation : this.mapperLocations) {  if (mapperLocation == null) {  continue;  }   try {  XMLMapperBuilder xmlMapperBuilder = new XMLMapperBuilder(mapperLocation.getInputStream(),  configuration, mapperLocation.toString(), configuration.getSqlFragments());**  **// 解析映射文件  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");  }  }   return this.sqlSessionFactoryBuilder.build(configuration); }** |

# XMLMapperBuilder

## 解析标签

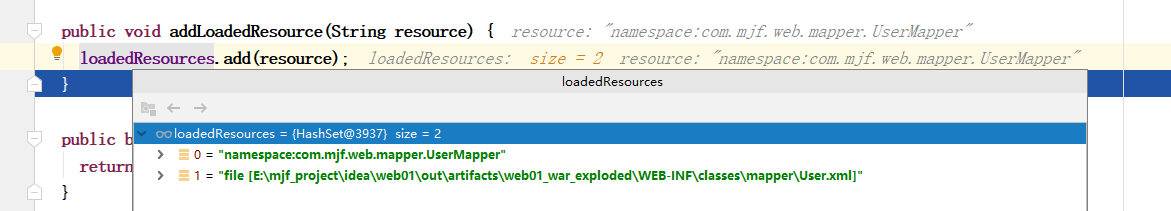
|  |
| --- |
| **public void parse() {  if (!configuration.isResourceLoaded(resource)) {  configurationElement(parser.evalNode("/mapper"));  configuration.addLoadedResource(resource);  bindMapperForNamespace();  }   parsePendingResultMaps();  parsePendingCacheRefs();  parsePendingStatements(); }** |

# Configuration

## 字段

|  |
| --- |
| **// 映射语句**  **protected final** Map<String, MappedStatement> **mappedStatements** = **new** StrictMap<MappedStatement>(**"Mapped Statements collection"**);  **protected final** Set<String> **loadedResources** = **new** HashSet<String>();  **protected final** MapperRegistry **mapperRegistry** = **new** MapperRegistry(**this**); |

## 加载资源



# MapperRegistry

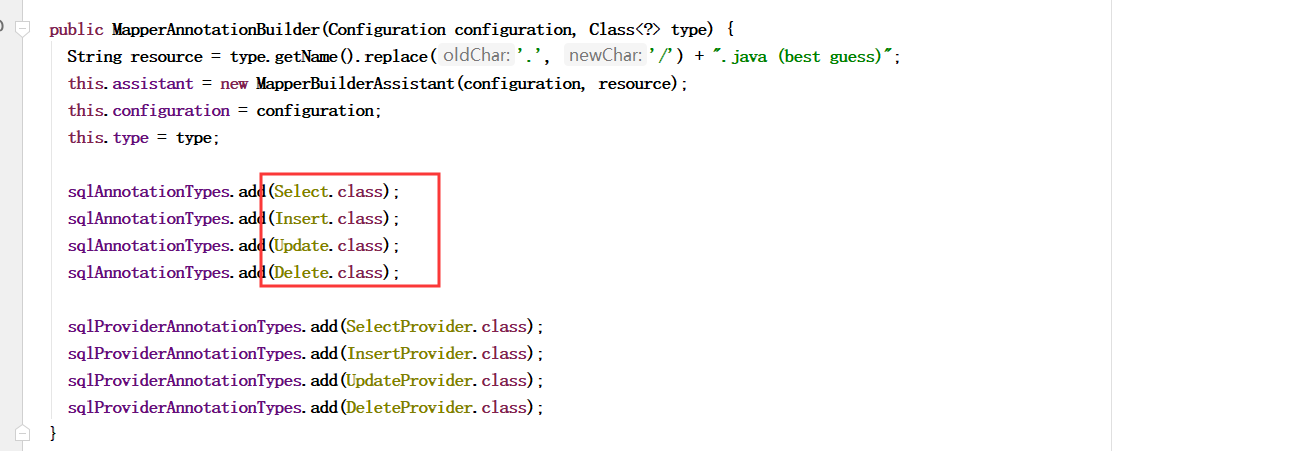
## 为映射接口添加代理工厂

|  |
| --- |
| **public <T> void addMapper(Class<T> type) {  if (type.isInterface()) {  if (hasMapper(type)) {  throw new BindingException("Type " + type + " is already known to the MapperRegistry.");  }  boolean loadCompleted = false;  try {  knownMappers.put(type, new MapperProxyFactory<T>(type));  *// It's important that the type is added before the parser is run  // otherwise the binding may automatically be attempted by the  // mapper parser. If the type is already known, it won't try.* MapperAnnotationBuilder parser = new MapperAnnotationBuilder(config, type);  parser.parse();  loadCompleted = true;  } finally {  if (!loadCompleted) {  knownMappers.remove(type);  }  }  } }** |

# MapperAnnotationBuilder

## 作用

用来处理注解的



# ClassPathBeanDefinitionScanner

## 作用

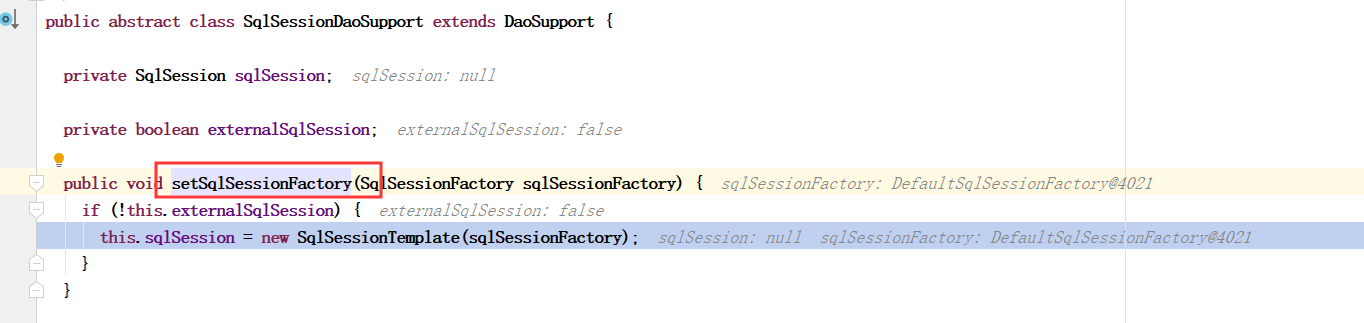
扫描映射接口的

## 核心方法

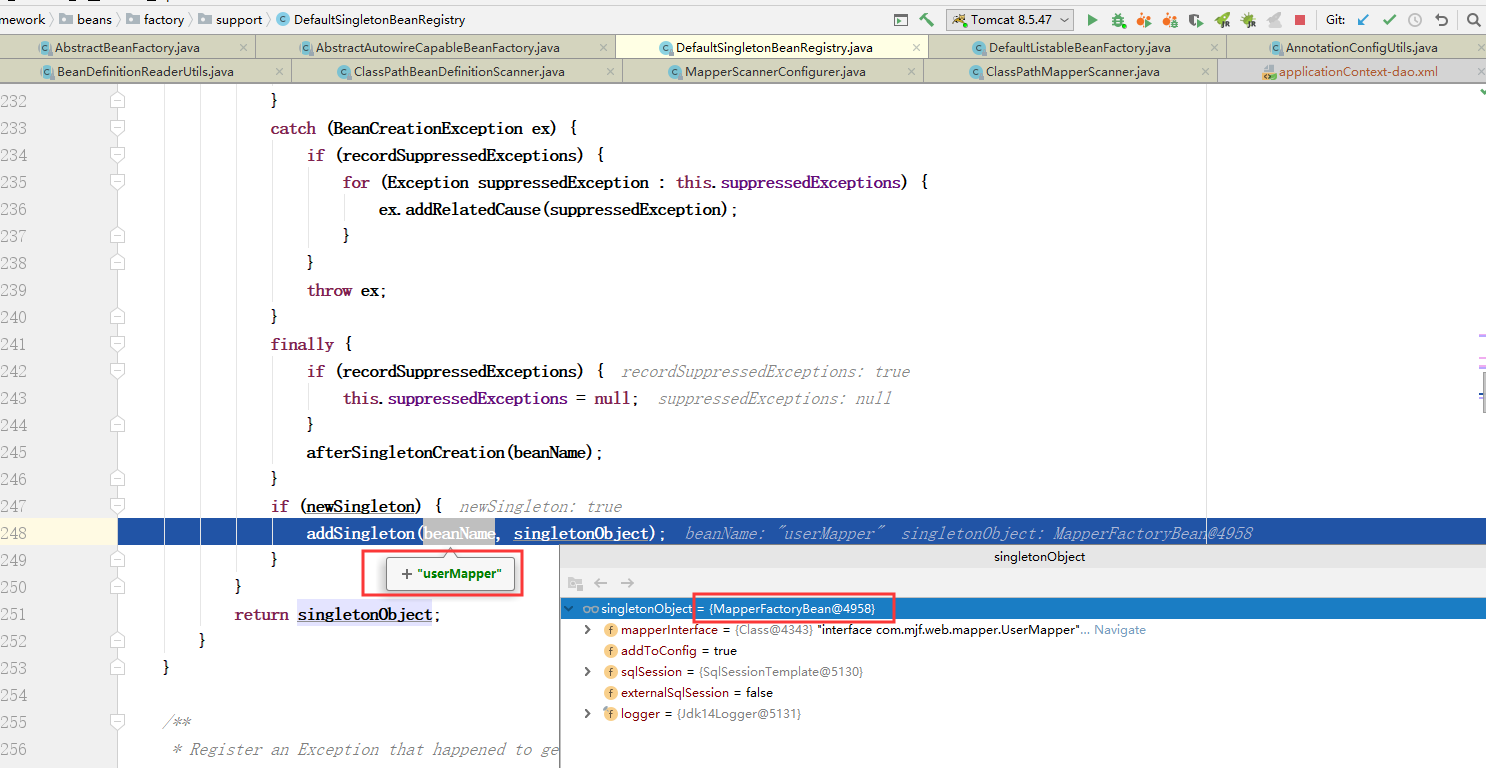
|  |
| --- |
| ***/\*\*  \* Calls the parent search that will search and register all the candidates.  \* Then the registered objects are post processed to set them as  \* MapperFactoryBeans  \*/* @Override public Set<BeanDefinitionHolder> doScan(String... basePackages) {**  **// 注册映射接口  Set<BeanDefinitionHolder> beanDefinitions = super.doScan(basePackages);   if (beanDefinitions.isEmpty()) {  logger.warn("No MyBatis mapper was found in '" + Arrays.*toString*(basePackages) + "' package. Please check your configuration.");  } else {**  **// 处理BeanDifinition  processBeanDefinitions(beanDefinitions);  }   return beanDefinitions; }** |

|  |
| --- |
| **private void processBeanDefinitions(Set<BeanDefinitionHolder> beanDefinitions) {  GenericBeanDefinition definition;  for (BeanDefinitionHolder holder : beanDefinitions) {  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* definition.getConstructorArgumentValues().addGenericArgumentValue(definition.getBeanClassName()); *// issue #59* definition.setBeanClass(this.mapperFactoryBean.getClass());   definition.getPropertyValues().add("addToConfig", this.addToConfig);   boolean explicitFactoryUsed = false;  if (StringUtils.*hasText*(this.sqlSessionFactoryBeanName)) {**  **// 添加SqlessionFactory属性  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*);  }  } }** |

## 设置sqlSessionFactory

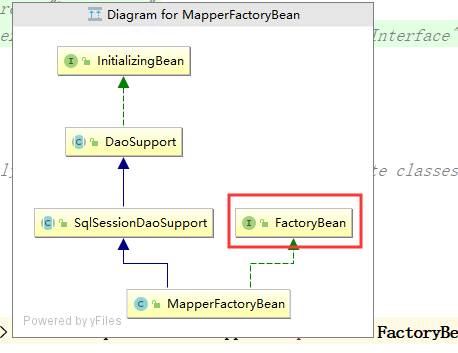


# 映射接口的真实类型



# MapperFactoryBean

## 类图



## 核心方法

|  |
| --- |
| **public T getObject() throws Exception {  return getSqlSession().getMapper(this.mapperInterface); }** |
| **@SuppressWarnings("unchecked") public <T> T getMapper(Class<T> type, SqlSession sqlSession) {**  **// 上文中放进去的对象  final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>) knownMappers.get(type);  if (mapperProxyFactory == null) {  throw new BindingException("Type " + type + " is not known to the MapperRegistry.");  }  try {  return mapperProxyFactory.newInstance(sqlSession);  } catch (Exception e) {  throw new BindingException("Error getting mapper instance. Cause: " + e, e);  } }** |

# MapperMethod

## 核心方法

|  |
| --- |
| **public Object execute(SqlSession sqlSession, Object[] args) {  Object result;  switch (command.getType()) {  case *INSERT*: {  Object param = method.convertArgsToSqlCommandParam(args);  result = rowCountResult(sqlSession.insert(command.getName(), param));  break;  }  case *UPDATE*: {  Object param = method.convertArgsToSqlCommandParam(args);  result = rowCountResult(sqlSession.update(command.getName(), param));  break;  }  case *DELETE*: {  Object param = method.convertArgsToSqlCommandParam(args);  result = rowCountResult(sqlSession.delete(command.getName(), param));  break;  }  case *SELECT*:  if (method.returnsVoid() && method.hasResultHandler()) {  executeWithResultHandler(sqlSession, args);  result = null;  } else if (method.returnsMany()) {  result = executeForMany(sqlSession, args);  } else if (method.returnsMap()) {  result = executeForMap(sqlSession, args);  } else if (method.returnsCursor()) {  result = executeForCursor(sqlSession, args);  } else {  Object param = method.convertArgsToSqlCommandParam(args);**  **// 获取一条记录  result = sqlSession.selectOne(command.getName(), param);  }  break;  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; }** |

# SimpleExecutor

## prepareStatement

|  |
| --- |
| **private Statement prepareStatement(StatementHandler handler, Log statementLog) throws SQLException {  Statement stmt;**  **// 获取连接  Connection connection = getConnection(statementLog);  stmt = handler.prepare(connection, transaction.getTimeout());**  **// 设置参数  handler.parameterize(stmt);  return stmt; }** |

# SqlSessionTemplate

|  |
| --- |
| **public SqlSessionTemplate(SqlSessionFactory sqlSessionFactory, ExecutorType executorType,  PersistenceExceptionTranslator exceptionTranslator) {   *notNull*(sqlSessionFactory, "Property 'sqlSessionFactory' is required");  *notNull*(executorType, "Property 'executorType' is required");   this.sqlSessionFactory = sqlSessionFactory;  this.executorType = executorType;  this.exceptionTranslator = exceptionTranslator;**  **// 代理  this.sqlSessionProxy = (SqlSession) *newProxyInstance*(  SqlSessionFactory.class.getClassLoader(),  new Class[] { SqlSession.class },  new SqlSessionInterceptor()); }** |

## 内部类SqlSessionInterceptor

|  |
| --- |
| **public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  SqlSession sqlSession = *getSqlSession*(  SqlSessionTemplate.this.sqlSessionFactory,  SqlSessionTemplate.this.executorType,  SqlSessionTemplate.this.exceptionTranslator);  try {  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((PersistenceException) unwrapped);  if (translated != null) {  unwrapped = translated;  }  }  throw unwrapped;  } finally {  if (sqlSession != null) {  *closeSqlSession*(sqlSession, SqlSessionTemplate.this.sqlSessionFactory);  }  }  } }** |

# DefaultSqlSessionFactory

## openSessionFromDataSource

|  |
| --- |
| **private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level, boolean autoCommit) {  Transaction tx = null;  try {  final Environment environment = configuration.getEnvironment();  final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment);**  **// 创建事务  tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);  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();  } }** |

# PreparedStatementHandler

## 初始化Statement

|  |
| --- |
| **@Override protected Statement instantiateStatement(Connection connection) throws SQLException {  String sql = boundSql.getSql();  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() != null) {  return connection.prepareStatement(sql, mappedStatement.getResultSetType().getValue(), ResultSet.*CONCUR\_READ\_ONLY*);  } else {  return connection.prepareStatement(sql);  } }** |

## 查询

|  |
| --- |
| **@Override public <E> List<E> query(Statement statement, ResultHandler resultHandler) throws SQLException {  PreparedStatement ps = (PreparedStatement) statement;  ps.execute();  return resultSetHandler.<E> handleResultSets(ps); }** |