当一个项目中使用了 Spring 和 Mybatis 时，通常会有以下配置。当然现在很多项目应该都是 SpringBoot 了，可能没有以下配置，但是究其底层原理都是类似的，无非是将扫描 bean 等一些工作通过注解来实现。

```
<!-- DAO接口所在包名，Spring会自动查找其下的类 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!--basePackage指定要扫描的包，在此包之下的映射器都会被搜索到。可指定多个包，包与包之间用逗号或分号分隔-->
    <property name="basePackage" value="com.joonwhee.open.mapper"/>
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
</bean>

<!-- spring和MyBatis完美整合，不需要mybatis的配置映射文件 -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <!-- 自动扫描mapping.xml文件 -->
    <property name="mapperLocations" value="classpath:config/mapper/*.xml"/>
    <property name="configLocation" value="classpath:config/mybatis/mybatis-config.xml"/>
    <!--Entity package -->
    <property name="typeAliasesPackage" value="com.joonwhee.open.po"/>
</bean>

<!-- dataSource -->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close">
    <property name="driverClassName" value="${driver}"/>
    <property name="url" value="${url}"/>
    <property name="username" value="${username}"/>
    <property name="password" value="${password}"/>
</bean>
```

通常我们还会有 DAO 类和 对用的 mapper 文件，如下。

```
package com.joonwhee.open.mapper;

import com.joonwhee.open.po.UserPO;

public interface UserPOMapper {
    UserPO queryByPrimaryKey(Integer id);
}
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.joonwhee.open.mapper.UserPOMapper" >
    <resultMap id="BaseResultMap" type="com.joonwhee.open.po.UserPO">
        <result column="id" property="id" jdbcType="INTEGER" />
        <result column="name" property="name" jdbcType="VARCHAR" />
    </resultMap>

    <select id="queryByPrimaryKey" resultMap="BaseResultMap"
            parameterType="java.lang.Integer">
```

```
        select id, name
        from user
        where id = #{id,jdbcType=INTEGER}
    </select>
</mapper>
```

1、解析 MapperScannerConfigurer

MapperScannerConfigurer 是一个 BeanDefinitionRegistryPostProcessor，会在 Spring 构建 IoC容器的早期被调用重写的 postProcessBeanDefinitionRegistry 方法，参考：Spring IoC：invokeBeanFactoryPostProcessors 详解

```
@Override
public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) {
  if (this.processPropertyPlaceHolders) {
    processPropertyPlaceHolders();
  }

  // 1.新建一个ClassPathMapperScanner，并填充相应属性
  ClassPathMapperScanner scanner = new ClassPathMapperScanner(registry);
  scanner.setAddToConfig(this.addToConfig);
  scanner.setAnnotationClass(this.annotationClass);
  scanner.setMarkerInterface(this.markerInterface);
  scanner.setSqlSessionFactory(this.sqlSessionFactory);
  scanner.setSqlSessionTemplate(this.sqlSessionTemplate);
  scanner.setSqlSessionFactoryBeanName(this.sqlSessionFactoryBeanName);
  scanner.setSqlSessionTemplateBeanName(this.sqlSessionTemplateBeanName);
  scanner.setResourceLoader(this.applicationContext);
  scanner.setBeanNameGenerator(this.nameGenerator);
  scanner.setMapperFactoryBeanClass(this.mapperFactoryBeanClass);
  if (StringUtils.hasText(lazyInitialization)) {
    // 2.设置mapper bean是否需要懒加载
    scanner.setLazyInitialization(Boolean.valueOf(lazyInitialization));
  }
  // 3.注册Filter，因为上面构造函数我们没有使用默认的Filter，
  // 有两种Filter，includeFilters：要扫描的；excludeFilters：要排除的
  scanner.registerFilters();
  // 4.扫描basePackage，basePackage可通过",; \t\n"来填写多个，
  // ClassPathMapperScanner重写了doScan方法
  scanner.scan(
      StringUtils.tokenizeToStringArray(this.basePackage,
ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS));
}
```

3.注册 Filter，见代码块1。

4.扫描 basePackage，这边会走到 ClassPathBeanDefinitionScanner（ClassPathMapperScanner 的父类），然后在执行 “doScan(basePackages)” 时回到 ClassPathMapperScanner 重写的方法，见代码块2。

代码块1：registerFilters

```
public void registerFilters() {
  boolean acceptAllInterfaces = true;
```

```
  // if specified, use the given annotation and / or marker interface
  // 1.如果指定了注解，则将注解添加到includeFilters
  if (this.annotationClass != null) {
    addIncludeFilter(new AnnotationTypeFilter(this.annotationClass));
    acceptAllInterfaces = false;
  }

  // override AssignableTypeFilter to ignore matches on the actual marker
interface
  // 2.如果指定了标记接口，则将标记接口添加到includeFilters，
  // 但这边重写了matchClassName方法，并返回了false，
  // 相当于忽略了标记接口上的匹配项，所以该参数目前相当于没有任何作用
  if (this.markerInterface != null) {
    addIncludeFilter(new AssignableTypeFilter(this.markerInterface) {
      @Override
      protected boolean matchClassName(String className) {
        return false;
      }
    });
    acceptAllInterfaces = false;
  }

  // 3.如果没有指定annotationClass和markerInterface，则
  // 添加默认的includeFilters，直接返回true，接受所有类
  if (acceptAllInterfaces) {
    // default include filter that accepts all classes
    addIncludeFilter((metadataReader, metadataReaderFactory) -> true);
  }

  // exclude package-info.java
  // 4.添加默认的excludeFilters，排除以package-info结尾的类
  addExcludeFilter((metadataReader, metadataReaderFactory) -> {
    String className = metadataReader.getClassMetadata().getClassName();
    return className.endsWith("package-info");
  });
}
```

通常我们都不会指定 annotationClass 和 markerInterface，也就是会添加默认的 Filter，相当于会接受除了 package-info 结尾的所有类。因此，basePackage 包下的类不需要使用 @Component 注解或 XML 中配置 bean 定义，也会被添加到 IoC 容器中。

代码块2：doScan

```
@Override
public Set<BeanDefinitionHolder> doScan(String... basePackages) {
  // 1.直接使用父类的方法扫描和注册bean定义，
  // 之前在spring中已经介绍过：Spring IoC源码学习：context:component-scan 节点详解 代码
块5
  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 {
      // 2.对扫描到的beanDefinitions进行处理，主要4件事：
      // 1）将bean的真正接口类添加到通用构造函数参数中
      // 2）将beanClass直接设置为MapperFactoryBean.class，
      //  结合1，相当于要使用的构造函数是MapperFactoryBean(java.lang.Class<T>)
      // 3）添加sqlSessionFactory属性，sqlSessionFactoryBeanName和
      //  sqlSessionFactory中，优先使用sqlSessionFactoryBeanName
      // 4）添加sqlSessionTemplate属性，同样的，sqlSessionTemplateBeanName
      //  优先于sqlSessionTemplate，
      processBeanDefinitions(beanDefinitions);
    }

    return beanDefinitions;
  }
```

小结，解析 MapperScannerConfigurer 主要是做了几件事：

1）新建扫描器 ClassPathMapperScanner；

2）使用 ClassPathMapperScanner 扫描注册 basePackage 包下的所有 bean；

3）将 basePackage 包下的所有 bean 进行一些特殊处理：beanClass 设置为 MapperFactoryBean、bean 的真正接口类作为构造函数参数传入 MapperFactoryBean、为 MapperFactoryBean 添加 sqlSessionFactory 和 sqlSessionTemplate属性。

2、解析 SqlSessionFactoryBean

对于 SqlSessionFactoryBean 来说，实现了2个接口，InitializingBean 和 FactoryBean，看过我之前 Spring 文章的同学应该对这2个接口不会陌生，简单来说：1）FactoryBean 可以自己定义创建实例对象的方法，只需要实现它的 getObject() 方法；InitializingBean 则是会在 bean 初始化阶段被调用。

SqlSessionFactoryBean 重写这两个接口的部分方法代码如下，核心代码就一个方法——“buildSqlSessionFactory()”。

```
@Override
public SqlSessionFactory getObject() throws Exception {
  if (this.sqlSessionFactory == null) {
    // 如果之前没有构建，则这边也会调用afterPropertiesSet进行构建操作
    afterPropertiesSet();
  }

  return this.sqlSessionFactory;
}

@Override
public void afterPropertiesSet() throws Exception {
  // 省略部分代码
  // 构建sqlSessionFactory
  this.sqlSessionFactory = buildSqlSessionFactory();
}
```

buildSqlSessionFactory()

主要做了几件事：1）对我们配置的参数进行相应解析；2）使用配置的参数构建一个 Configuration；3）使用 Configuration 新建一个 DefaultSqlSessionFactory。

这边的核心内容是对于 mapperLocations 的解析，如下代码。

```
protected SqlSessionFactory buildSqlSessionFactory() throws Exception {

  // 省略部分代码

  // 5.mapper处理（最重要）
  if (this.mapperLocations != null) {
    if (this.mapperLocations.length == 0) {
      LOGGER.warn(() -> "Property 'mapperLocations' was specified but matching
resources are not found.");
    } else {
      for (Resource mapperLocation : this.mapperLocations) {
        if (mapperLocation == null) {
          continue;
        }
        try {
          // 5.1 新建XMLMapperBuilder
          XMLMapperBuilder xmlMapperBuilder = new
XMLMapperBuilder(mapperLocation.getInputStream(),
              targetConfiguration, mapperLocation.toString(),
targetConfiguration.getSqlFragments());
          // 5.2 解析mapper文件
          xmlMapperBuilder.parse();
        } catch (Exception e) {
          throw new NestedIOException("Failed to parse mapping resource: '" +
mapperLocation + "'", e);
        } finally {
          ErrorContext.instance().reset();
        }
        LOGGER.debug(() -> "Parsed mapper file: '" + mapperLocation + "'");
      }
    }
  } else {
    LOGGER.debug(() -> "Property 'mapperLocations' was not specified.");
  }

  // 6.使用targetConfiguration构建DefaultSqlSessionFactory
  return this.sqlSessionFactoryBuilder.build(targetConfiguration);
}
```

5.2 解析mapper文件，见代码块3。

代码块3：parse()

```
public void parse() {
  // 1.如果resource没被加载过才进行加载
  if (!configuration.isResourceLoaded(resource)) {
    // 1.1 解析mapper文件
    configurationElement(parser.evalNode("/mapper"));
    // 1.2 将resource添加到已加载列表
    configuration.addLoadedResource(resource);
    // 1.3 绑定namespace的mapper
    bindMapperForNamespace();
  }
```

```
  parsePendingResultMaps();
  parsePendingCacheRefs();
  parsePendingStatements();
}
```

1.1 解析mapper文件，见代码4。

1.3 绑定namespace的mapper，见代码块6。

代码块4：configurationElement

```
private void configurationElement(XNode context) {
  try {
    // 1.获取namespace属性
    String namespace = context.getStringAttribute("namespace");
    if (namespace == null || namespace.isEmpty()) {
      throw new BuilderException("Mapper's namespace cannot be empty");
    }
    // 2.设置currentNamespace属性
    builderAssistant.setCurrentNamespace(namespace);
    // 3.解析parameterMap、resultMap、sql等节点
    cacheRefElement(context.evalNode("cache-ref"));
    cacheElement(context.evalNode("cache"));
    parameterMapElement(context.evalNodes("/mapper/parameterMap"));
    resultMapElements(context.evalNodes("/mapper/resultMap"));
    sqlElement(context.evalNodes("/mapper/sql"));
    // 4.解析增删改查节点，封装成Statement
    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);
  }
}

private void buildStatementFromContext(List<XNode> list) {
  if (configuration.getDatabaseId() != null) {
    buildStatementFromContext(list, configuration.getDatabaseId());
  }
  // 解析增删改查节点，封装成Statement
  buildStatementFromContext(list, null);
}

private void buildStatementFromContext(List<XNode> list, String
requiredDatabaseId) {
  for (XNode context : list) {
    // 1.构建XMLStatementBuilder
    final XMLStatementBuilder statementParser = new
XMLStatementBuilder(configuration, builderAssistant, context,
requiredDatabaseId);
    try {
      // 2.解析节点
      statementParser.parseStatementNode();
    } catch (IncompleteElementException e) {
      configuration.addIncompleteStatement(statementParser);
    }
```

```
  }
}
```

这边会一直执行到 “statementParser.parseStatementNode();”，见代码块5。

这边每个 XNode 都相当于如下的一个 SQL，下面封装的每个 MappedStatement 可以理解就是每个 SQL。

```
<select id="queryByPrimaryKey" resultMap="BaseResultMap"
        parameterType="java.lang.Integer">
    select id, name, password, age
    from user
    where id = #{id,jdbcType=INTEGER}
</select>
```

代码块5：parseStatementNode

```
public void parseStatementNode() {
  // 省略所有的属性解析
  // 将解析出来的所有参数添加到 mappedStatements 缓存
  builderAssistant.addMappedStatement(id, sqlSource, statementType,
sqlCommandType,
      fetchSize, timeout, parameterMap, parameterTypeClass, resultMap,
resultTypeClass,
      resultSetTypeEnum, flushCache, useCache, resultOrdered,
      keyGenerator, keyProperty, keyColumn, databaseId, langDriver, resultSets);
}

// MapperBuilderAssistant.java
public MappedStatement addMappedStatement(
    String id,
    SqlSource sqlSource,
    StatementType statementType,
    SqlCommandType sqlCommandType,
    Integer fetchSize,
    Integer timeout,
    String parameterMap,
    Class<?> parameterType,
    String resultMap,
    Class<?> resultType,
    ResultSetType resultSetType,
    boolean flushCache,
    boolean useCache,
    boolean resultOrdered,
    KeyGenerator keyGenerator,
    String keyProperty,
    String keyColumn,
    String databaseId,
    LanguageDriver lang,
    String resultSets) {

  if (unresolvedCacheRef) {
    throw new IncompleteElementException("Cache-ref not yet resolved");
  }
```

```
  // 1.将id填充上namespace，例如：queryByPrimaryKey变成
  // com.joonwhee.open.mapper.UserPOMapper.queryByPrimaryKey
  id = applyCurrentNamespace(id, false);
  boolean isSelect = sqlCommandType == SqlCommandType.SELECT;
  // 2.使用参数构建MappedStatement.Builder
  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);

  ParameterMap statementParameterMap = getStatementParameterMap(parameterMap,
parameterType, id);
  if (statementParameterMap != null) {
    statementBuilder.parameterMap(statementParameterMap);
  }
  // 3.使用MappedStatement.Builder构建MappedStatement
  MappedStatement statement = statementBuilder.build();
  // 4.将MappedStatement 添加到缓存
  configuration.addMappedStatement(statement);
  return statement;
}
```

该方法会将节点的属性解析后封装成 MappedStatement，放到 mappedStatements 缓存中，key 为 id，例如：com.joonwhee.open.mapper.UserPOMapper.queryByPrimaryKey，value 为 MappedStatement。

代码块6：bindMapperForNamespace

```
private void bindMapperForNamespace() {
  String namespace = builderAssistant.getCurrentNamespace();
  if (namespace != null) {
    Class<?> boundType = null;
    try {
      // 1.解析namespace对应的绑定类型
      boundType = Resources.classForName(namespace);
    } catch (ClassNotFoundException e) {
      // ignore, bound type is not required
    }
    if (boundType != null && !configuration.hasMapper(boundType)) {
      // Spring may not know the real resource name so we set a flag
      // to prevent loading again this resource from the mapper interface
```

```
      // look at MapperAnnotationBuilder#loadXmlResource
      // 2.boundType不为空，并且configuration还没有添加boundType，
      // 则将namespace添加到已加载列表，将boundType添加到knownMappers缓存
      configuration.addLoadedResource("namespace:" + namespace);
      configuration.addMapper(boundType);
    }
  }
}

public <T> void addMapper(Class<T> type) {
  mapperRegistry.addMapper(type);
}

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 {
      // 将type和以该type为参数构建的MapperProxyFactory作为键值对，
      // 放到knownMappers缓存中去
      knownMappers.put(type, new MapperProxyFactory<>(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);
      }
    }
  }
}
```

主要是将刚刚解析过的 mapper 文件的 namespace 放到 knownMappers 缓存中，key 为 namespace 对应的 class，value 为 MapperProxyFactory。

小结，解析 SqlSessionFactoryBean 主要做了几件事：

1）解析处理所有属性参数构建 Configuration ，使用 Configuration 新建 DefaultSqlSessionFactory；

2）解析 mapperLocations 属性的 mapper 文件，将 mapper 文件中的每个 SQL 封装成 MappedStatement，放到 mappedStatements 缓存中，key 为 id，例如：com.joonwhee.open.mapper.UserPOMapper.queryByPrimaryKey，value 为 MappedStatement。

3）将解析过的 mapper 文件的 namespace 放到 knownMappers 缓存中，key 为 namespace 对应的 class，value 为 MapperProxyFactory。

3、解析 DAO 文件

DAO 文件，也就是 basePackage 指定的包下的文件，也就是上文的 interface UserPOMapper 。

上文 doScan 中说过，basePackage 包下所有 bean 定义的 beanClass 会被设置成 MapperFactoryBean.class，而 MapperFactoryBean 也是 FactoryBean，因此直接看 MapperFactoryBean 的 getObject 方法。

```
@Override
public T getObject() throws Exception {
  // 1.从父类中拿到sqlSessionTemplate，这边的sqlSessionTemplate也是doScan中添加的属性
  // 2.通过mapperInterface获取mapper
  return getSqlSession().getMapper(this.mapperInterface);
}

// SqlSessionTemplate
@Override
public <T> T getMapper(Class<T> type) {
  return getConfiguration().getMapper(type, this);
}

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

// MapperRegistry.java
public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
  // 1.从knownMappers缓存中获取
  final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>)
knownMappers.get(type);
  if (mapperProxyFactory == null) {
    throw new BindingException("Type " + type + " is not known to the
MapperRegistry.");
  }
  try {
    // 2.新建实例
    return mapperProxyFactory.newInstance(sqlSession);
  } catch (Exception e) {
    throw new BindingException("Error getting mapper instance. Cause: " + e, e);
  }
}

// MapperProxyFactory.java
public T newInstance(SqlSession sqlSession) {
  // 1.构造一个MapperProxy
  final MapperProxy<T> mapperProxy = new MapperProxy<>(sqlSession,
mapperInterface, methodCache);
  // 2.使用MapperProxy来构建实例对象
  return newInstance(mapperProxy);
}

protected T newInstance(MapperProxy<T> mapperProxy) {
  // 使用JDK动态代理来代理要创建的实例对象，InvocationHandler为mapperProxy，
  // 因此当我们真正调用时，会走到mapperProxy的invoke方法
  return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new
Class[] { mapperInterface }, mapperProxy);
}
```

这边代码用到的 sqlSessionTemplate、mapperInterface 等都是之前添加的属性。

小结，解析 DAO 文件 主要做了几件事：

1）通过 mapperInterface 从 knownMappers 缓存中获取到 MapperProxyFactory 对象；

2）通过 JDK 动态代理创建 MapperProxyFactory 实例对象，InvocationHandler 为 MapperProxy。

4、DAO 接口被调用

当 DAO 中的接口被调用时，会走到 MapperProxy 的 invoke 方法。

```
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
  try {
    if (Object.class.equals(method.getDeclaringClass())) {
      return method.invoke(this, args);
    } else {
      // 1.创建MapperMethodInvoker
      // 2.将method -> MapperMethodInvoker放到methodCache缓存
      // 3.调用MapperMethodInvoker的invoke方法
      return cachedInvoker(method).invoke(proxy, method, args, sqlSession);
    }
  } catch (Throwable t) {
    throw ExceptionUtil.unwrapThrowable(t);
  }
}

// MapperProxy.java
private MapperMethodInvoker cachedInvoker(Method method) throws Throwable {
  try {
    // 1.放到methodCache缓存，key为method，value为MapperMethodInvoker
    return methodCache.computeIfAbsent(method, m -> {
      if (m.isDefault()) {
        // 2.方法为默认方法，Java8之后，接口允许有默认方法
        try {
          if (privateLookupInMethod == null) {
            return new DefaultMethodInvoker(getMethodHandleJava8(method));
          } else {
            return new DefaultMethodInvoker(getMethodHandleJava9(method));
          }
        } catch (IllegalAccessException | InstantiationException |
InvocationTargetException
            | NoSuchMethodException e) {
          throw new RuntimeException(e);
        }
      } else {
        // 3.正常接口会走这边，使用mapperInterface、method、configuration
        // 构建一个MapperMethod，封装成PlainMethodInvoker
        return new PlainMethodInvoker(new MapperMethod(mapperInterface, method,
sqlSession.getConfiguration()));
      }
    });
  } catch (RuntimeException re) {
```

```
    Throwable cause = re.getCause();
    throw cause == null ? re : cause;
  }
}
```

3.调用 MapperMethodInvoker 的 invoke 方法，见代码块7。

代码块7：invoke

```
@Override
public Object invoke(Object proxy, Method method, Object[] args, SqlSession
sqlSession) throws Throwable {
  return mapperMethod.execute(sqlSession, args);
}

// MapperMethod.java
public Object execute(SqlSession sqlSession, Object[] args) {
  Object result;
  // 1.根据命令类型执行来进行相应操作
  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);
        result = sqlSession.selectOne(command.getName(), param);
        if (method.returnsOptional()
            && (result == null ||
!method.getReturnType().equals(result.getClass()))) {
          result = Optional.ofNullable(result);
        }
      }
      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;
}
```

这边就比较简单，根据不同的操作类型执行相应的操作，最终将结果返回，见代码块8。

这边的 command 是上文 “new MapperMethod(mapperInterface, method, sqlSession.getConfiguration())” 时创建的。

代码块8：增删改查

```
// 1.insert
@Override
public int insert(String statement, Object parameter) {
  return update(statement, parameter);
}

// 2.update
@Override
public int update(String statement, Object parameter) {
  try {
    dirty = true;
    // 从mappedStatements缓存拿到对应的MappedStatement对象，执行更新操作
    MappedStatement ms = configuration.getMappedStatement(statement);
    return executor.update(ms, wrapCollection(parameter));
  } catch (Exception e) {
    throw ExceptionFactory.wrapException("Error updating database.  Cause: " +
e, e);
  } finally {
    ErrorContext.instance().reset();
  }
}

// 3.delete
@Override
public int delete(String statement, Object parameter) {
  return update(statement, parameter);
}

// 4.select，以executeForMany为例
private <E> Object executeForMany(SqlSession sqlSession, Object[] args) {
  List<E> result;
  // 1.参数转换成sql命令参数
  Object param = method.convertArgsToSqlCommandParam(args);
  if (method.hasRowBounds()) {
```

```
      RowBounds rowBounds = method.extractRowBounds(args);
      result = sqlSession.selectList(command.getName(), param, rowBounds);
    } else {
      // 2.执行查询操作
      result = sqlSession.selectList(command.getName(), param);
    }
    // 3.处理返回结果
    // issue #510 Collections & arrays support
    if (!method.getReturnType().isAssignableFrom(result.getClass())) {
      if (method.getReturnType().isArray()) {
        return convertToArray(result);
      } else {
        return convertToDeclaredCollection(sqlSession.getConfiguration(), result);
      }
    }
    return result;
  }

  @Override
  public <E> List<E> selectList(String statement, Object parameter) {
    return this.selectList(statement, parameter, RowBounds.DEFAULT);
  }

  @Override
  public <E> List<E> selectList(String statement, Object parameter, RowBounds
  rowBounds) {
    try {
      //从mappedStatements缓存中拿到对应的MappedStatement对象，执行查询操作
      MappedStatement ms = configuration.getMappedStatement(statement);
      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();
    }
  }
```

可以看出，最终都是从 mappedStatements 缓存中拿到对应的 MappedStatement 对象，执行相应的操作。

这边的增删改查不是直接调用 SqlSession 中的方法，而是调用 SqlSessionTemplate 中的方法，继而通过 sqlSessionProxy 来调用 SqlSession 中的方法。SqlSessionTemplate 中的方法主要是通过 sqlSessionProxy 做了一层动态代理，基本没差别。

总结

整个流程主要是以下几个核心步骤：

1）扫描注册 basePackage 包下的所有 bean，将 basePackage 包下的所有 bean 进行一些特殊处理：beanClass 设置为 MapperFactoryBean、bean 的真正接口类作为构造函数参数传入 MapperFactoryBean、为 MapperFactoryBean 添加 sqlSessionFactory 和 sqlSessionTemplate属性。

2）解析 mapperLocations 属性的 mapper 文件，将 mapper 文件中的每个 SQL 封装成 MappedStatement，放到 mappedStatements 缓存中，key 为 id，例如：com.joonwhee.open.mapper.UserPOMapper.queryByPrimaryKey，value 为 MappedStatement。并且将解析过的 mapper 文件的 namespace 放到 knownMappers 缓存中，key 为 namespace 对应的 class，value 为 MapperProxyFactory。

3）创建 DAO 的 bean 时，通过 mapperInterface 从 knownMappers 缓存中获取到 MapperProxyFactory 对象，通过 JDK 动态代理创建 MapperProxyFactory 实例对象，InvocationHandler 为 MapperProxy。

4）DAO 中的接口被调用时，通过动态代理，调用 MapperProxy 的 invoke 方法，最终通过 mapperInterface 从 mappedStatements 缓存中拿到对应的 MappedStatement，执行相应的操作。