1 MybatisAutoConfiguration 功能概况

- 获取 MybatisProperties 对应的配置属性
- 生成 sqlSessionFactory 并将 MybatisProperties 中的配置放入其中
- 生成 sqlSessionTemplate 并将 MybatisProperties 中的 executorType 放入其中
- 注册扫描 @Mapper 注解的 Registrar

2 源码分析

- 项目中有 SqlSessionFactory、SqlSessionFactoryBean class 文件时才解析
- 项目中只有一个 候选的 datasource Bean 时才解析（如果有多个 datasource，可以通过 @Primary 指定唯一候选者）
- 在 DataSourceAutoConfiguration 后解析，以便先生成 datasource
- 只有项目中没有 sqlSessionFactory Bean 时，才生成 sqlSessionFactory（开发者创建的优先）
- 只有项目中没有 sqlSessionTemplate Bean 时，才生成 sqlSessionTemplate（开发者创建的优先）
- **注：** 如果开发者自己定义 sqlSessionFactory 、sqlSessionTemplate，MybatisProperties **就失效了**
- 只有项目中没有 **MapperFactoryBean**（使用 @MapperScan 时会注入）时，才会注册 扫描 @Mapper 注解的 Registrar，所以 **@MapperScan 和 @Mapper 注解只有一个会生效**

```
@org.springframework.context.annotation.Configuration
@ConditionalOnClass({ SqlSessionFactory.class, SqlSessionFactoryBean.class })
@ConditionalOnSingleCandidate(DataSource.class)
@EnableConfigurationProperties(MybatisProperties.class)
@AutoConfigureAfter(DataSourceAutoConfiguration.class)
public class MybatisAutoConfiguration implements InitializingBean {
  @Bean
  @ConditionalOnMissingBean
  public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {}
  @Bean
  @ConditionalOnMissingBean
  public SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory sqlSessionFactory) {}

  @org.springframework.context.annotation.Configuration
  @Import({ AutoConfiguredMapperScannerRegistrar.class })
  @ConditionalOnMissingBean(MapperFactoryBean.class)
  public static class MapperScannerRegistrarNotFoundConfiguration {}
```

```
}
```

2.1 生成 sqlSessionFactory

- interceptors：容器中的 interceptors（mybatis 插件）
- databaseIdProvider：容器中的 databaseIdProvider（提供多数据管理）
- 其余的属性在 3 中统一介绍

```
@Bean
@ConditionalOnMissingBean
public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {
  SqlSessionFactoryBean factory = new SqlSessionFactoryBean();
  factory.setDataSource(dataSource);
  factory.setVfs(SpringBootVFS.class);//一个工具类，用来获取 Resource
  //设置 MybatisProperties 属性到 sqlSessionFactory 中，代码略
  return factory.getObject();//如果定义了 mapperLocation，则进行 xml 解析
}
//factory.getObject() 最终会走到这
protected SqlSessionFactory buildSqlSessionFactory() throws IOException {

}
```

2.2 生成 sqlSessionTemplate

- 指定 执行器类型

```
@Bean
@ConditionalOnMissingBean
public SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory sqlSessionFactory) {
  ExecutorType executorType = this.properties.getExecutorType();
  if (executorType != null) {
    return new SqlSessionTemplate(sqlSessionFactory, executorType);
  } else {
    return new SqlSessionTemplate(sqlSessionFactory);
  }
}
```

2.3 注册扫描 @Mapper 注解的 Registrar

- 当 @MapperScan 注解不存在时，就去导入 AutoConfiguredMapperScannerRegistrar，用来生成 mapper 接口对应 beanDefinition
- 扫描 @EnableAutoConfiguration 类路径下，对应的 @Mapper 注解的类，并生成 设置为 mapperFactoryBean 的 BeanDefinition

```
@org.springframework.context.annotation.Configuration
@Import({ AutoConfiguredMapperScannerRegistrar.class })
@ConditionalOnMissingBean(MapperFactoryBean.class)
public static class MapperScannerRegistrarNotFoundConfiguration {
  @Override
  public void afterPropertiesSet() {
```

```
    logger.debug("No {} found.", MapperFactoryBean.class.getName());
  }
}
public static class AutoConfiguredMapperScannerRegistrar {
    @Override
    public void registerBeanDefinitions(importingClassMetadata,registry) {
        //必须存在 @EnableAutoConfiguration 注解
        if (!AutoConfigurationPackages.has(this.beanFactory)) {
            logger.debug("Could not determine auto-configuration package, automatic
mapper scanning disabled.");
            return;
        }
        //扫描 @Mapper 注解的路径就是 @EnableAutoConfiguration 注解 指定类的路径
        List<String> packages = AutoConfigurationPackages.get(this.beanFactory);
        if (logger.isDebugEnabled()) {
            packages.forEach(pkg -> logger.debug("Using auto-configuration base package
'{}'", pkg));
        }

        ClassPathMapperScanner scanner = new ClassPathMapperScanner(registry);
        if (this.resourceLoader != null) {
            scanner.setResourceLoader(this.resourceLoader);
        }
        scanner.setAnnotationClass(Mapper.class);
        scanner.registerFilters();//设置只扫描 @Mapper 注解类的过滤器(指定 @mapper)
        //扫描并注册 mapper 类对应的 BeanDefinition,关键的是将 BeanDefinition 设置成
mapperFactoryBean 类,以便后续生成mapper 代理操作!
        scanner.doScan(StringUtils.toStringArray(packages));
    }
}
```

2.3.1 生成 @Mapper 对应的 BeanDefinition (关键)

- 在 2.3 中将 @Mapper 对应的 BeanDefinition 设置成了 mapperFactoryBean
- 又因为 mapperFactoryBean实现了 InitializingBean，所以实例化的时候会去调用 afterPropertiesSet()
- afterPropertiesSet() 中会将 xml 中的所有 sql 解析成 **MappedStatement** 放入 configuration 中
- 默认情况下 mapper.xml 要和 mapper 接口在一个路径中
- 如果不在一个路径，则必须通过 mapperLocation 指定，否则 mapper 接口中没有 sql 相关信息，运行时会报错
- 具体 addMapper() 方法介绍可参考 Mybatis 源码分析 —— MappedStatement 解析

```
public abstract class DaoSupport implements InitializingBean {
    @Override
    public final void afterPropertiesSet() {
        checkDaoConfig();//继承类 mapperFactoryBean 实现了此方法
        initDao();//空实现，钩子方法
    }
}
public class MapperFactoryBean<T> extends SqlSessionDaoSupport implements
FactoryBean<T> {
  @Override
  protected void checkDaoConfig() {
```

```
    //此时的 mapperInterface 为 @Mapper 对应类的类型
    //将 它 加入 SqlSession 中的 configuration 中，以便后续的 getObject() 做 mapper 代理！！
    Configuration configuration = getSqlSession().getConfiguration();
    if (this.addToConfig && !configuration.hasMapper(this.mapperInterface)) {
        //addMapper 的时候，做了 mapper.xml 解析 或 注解解析工作！！
        //如果设置了 MybatisProperties 中的 MapperLocation 属性，则在 初始化
sqlsessionfactory 的时候，就会做流程 configuration.addMapper(namespace)，这个 namespace 就和
mapperInterface 一样了，所以 外层的 if 条件就不满足了
        //所以，如果指定了 xml 并且 namespace 对应了，肯定能解析成功，如果不指定，
mapperInterface 对应的 xml 不在 mapperInterface 对应的路径中，则不能解析 mapper
        //默认情况下 mapper.xml 要和 mapper 接口在一个路径中
        //如果不在一个路径，则必须通过 mapperLocation 指定
        configuration.addMapper(this.mapperInterface);
    }
  }

  @Override
  //bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd); 内部调用
getObject() 方法
  //实例化完成最后，最终会调用这个方法来返回 bean。因为实现了 FactoryBean<T> 接口！
  public T getObject() throws Exception {
    //返回代理对象！！
    return getSqlSession().getMapper(this.mapperInterface);
  }
}
public class Configuration {
  //通过这个将 mapper 和 sqlSession 绑定，返回代理类
  //这样应用程序调用 mapper 类中的方法时，就会走代理类逻辑！！
  public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    return mapperRegistry.getMapper(type, sqlSession);
  }
}
```

2.4 项目中存在 @MapperScan 时的操作

- 通过在配置类上加 @MapperScan 注解，开启对 mapper 接口的解析，将 mapper 接口生成 Beandifintion

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Import(MapperScannerRegistrar.class)
@Repeatable(MapperScans.class)
public @interface MapperScan {
  //以下 3 个属性作用都一样，用来确定扫描路径
  String[] value() default {};
  String[] basePackages() default {};
  Class<?>[] basePackageClasses() default {};// class 对应包的路径

  Class<? extends BeanNameGenerator> nameGenerator() default BeanNameGenerator.class;
  //扫描过滤
  Class<? extends Annotation> annotationClass() default Annotation.class;
  Class<?> markerInterface() default Class.class;
```

```
  //如果项目中有多个 SessionTemplate、sqlSessionFactory 时，通过这个指定
  String sqlSessionTemplateRef() default "";
  String sqlSessionFactoryRef() default "";
  //指定一个 MapperFactoryBean 来做 mapper 接口的实例化功能
  Class<? extends MapperFactoryBean> factoryBean() default MapperFactoryBean.class;
}
```

- 此注解用来引入 MapperScannerRegistrar 类达到 生成 mapper 对应 beanDefinition 的目的

```
public class MapperScannerRegistrar implements ImportBeanDefinitionRegistrar {
    public void registerBeanDefinitions(importingClassMetadata, registry) {
        //获取注解属性
        AnnotationAttributes mapperScanAttrs = AnnotationAttributes
.fromMap(importingClassMetadata.getAnnotationAttributes(MapperScan.class.getName()));
        //解析注解
        if (mapperScanAttrs != null) {
            registerBeanDefinitions(mapperScanAttrs, registry);
        }
    }
    void registerBeanDefinitions(annoAttrs, registry) {
        ClassPathMapperScanner scanner = new ClassPathMapperScanner(registry);

        Class<? extends Annotation> annotationClass =
annoAttrs.getClass("annotationClass");
        //默认 annotationClass 为 Annotation，如果不是再设置
        if (!Annotation.class.equals(annotationClass)) {
            scanner.setAnnotationClass(annotationClass);
        }
        //同理 markerInterface
        //同理 BeanNameGenerator
        //同理 MapperFactoryBean

        //设置 sqlsessionTemplate、sqlsessionFactory
        //设置 basePackages，通过 value、basePackages、basePackageClasses

        //添加 过滤器：markerInterface 和 annotationClass 来过滤扫描的类
        scanner.registerFilters();
        //扫描并注册 mapper 类对应的 BeanDefinition，关键的是将 BeanDefinition 设置成
mapperFactoryBean 类，以便后续生成mapper 代理操作!
        //和 2.3 流程一样了
        scanner.doScan(StringUtils.toStringArray(basePackages));//扫描
    }
}
```

3 MybatisProperties 属性介绍

```
@ConfigurationProperties(prefix = MybatisProperties.MYBATIS_PREFIX)
public class MybatisProperties {
  //configuration 属性对应的 xml 配置文件，这个属性和 configuration 不能同时设置
  //如果没有做不能同时设置的判断，configuration 配置优先
  private String configLocation;
```

```
  //**mapper.xml 对应的路径，用来解析 sql
  private String[] mapperLocations;

  //Packages to search type aliases. (Package delimiters are ",; \t\n")
  //如，registerAlias("byte", Byte.class)，将 路径下的类解析，并放入 Map<String, Class<?>>
TYPE_ALIASES。路径下的 Alias 类，可以通过 @Alias 注解指定 别名
  private String typeAliasesPackage;

  //指定 typeAliasesPackage 扫描路径下支持的 类型，默认为 Object 类，即所有都可以放入
TYPE_ALIASES
  private Class<?> typeAliasesSuperType;

  //Packages to search for type handlers. (Package delimiters are ",; \t\n")
  //解析后放入：Map<Type, Map<JdbcType, TypeHandler<?>>> TYPE_HANDLER_MAP
  private String typeHandlersPackage;

  //判断，配置的 configLocation 资源是否存在，如果设置为 true，且不存在会报错
  private boolean checkConfigLocation = false;

  //执行器类型
  private ExecutorType executorType;

  //对 configuration 配置类中 variables 属性的扩展
  private Properties configurationProperties;

  @NestedConfigurationProperty
  private Configuration configuration;
}
```

4 Configuration 属性介绍

- 未完待续。。

```
public class Configuration {

  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;
  protected String logPrefix;
  protected Class <? extends Log> logImpl;
  protected Class <? extends VFS> vfsImpl;
  protected LocalCacheScope localCacheScope = LocalCacheScope.SESSION;
  protected JdbcType jdbcTypeForNull = JdbcType.OTHER;
```

```
  protected Set<String> lazyLoadTriggerMethods = new HashSet<>(Arrays.asList("equals",
"clone", "hashCode", "toString"));
  protected Integer defaultStatementTimeout;
  protected Integer defaultFetchSize;
  protected ExecutorType defaultExecutorType = ExecutorType.SIMPLE;
  protected AutoMappingBehavior autoMappingBehavior = AutoMappingBehavior.PARTIAL;
  protected AutoMappingUnknownColumnBehavior autoMappingUnknownColumnBehavior =
AutoMappingUnknownColumnBehavior.NONE;

  protected Properties variables = new Properties();
  protected ReflectorFactory reflectorFactory = new DefaultReflectorFactory();
  protected ObjectFactory objectFactory = new DefaultObjectFactory();
  protected ObjectWrapperFactory objectWrapperFactory = new
DefaultObjectWrapperFactory();

  protected boolean lazyLoadingEnabled = false;
  protected ProxyFactory proxyFactory = new JavassistProxyFactory(); // #224 Using
internal Javassist instead of OGNL

  protected String databaseId;
  protected Class<?> configurationFactory;

  protected final MapperRegistry mapperRegistry = new MapperRegistry(this);
  protected final InterceptorChain interceptorChain = new InterceptorChain();
  protected final TypeHandlerRegistry typeHandlerRegistry = new TypeHandlerRegistry();
  protected final TypeAliasRegistry typeAliasRegistry = new TypeAliasRegistry();
  protected final LanguageDriverRegistry languageRegistry = new
LanguageDriverRegistry();

  protected final Map<String, MappedStatement> mappedStatements = new
StrictMap<MappedStatement>("Mapped Statements collection")
      .conflictMessageProducer((savedValue, targetValue) ->
          ". please check " + savedValue.getResource() + " and " +
targetValue.getResource());
  protected final Map<String, Cache> caches = new StrictMap<>("Caches collection");
  protected final Map<String, ResultMap> resultMaps = new StrictMap<>("Result Maps
collection");
  protected final Map<String, ParameterMap> parameterMaps = new StrictMap<>("Parameter
Maps collection");
  protected final Map<String, KeyGenerator> keyGenerators = new StrictMap<>("Key
Generators collection");

  protected final Set<String> loadedResources = new HashSet<>();
  protected final Map<String, XNode> sqlFragments = new StrictMap<>("XML fragments
parsed from previous mappers");

  protected final Collection<XMLStatementBuilder> incompleteStatements = new
LinkedList<>();
  protected final Collection<CacheRefResolver> incompleteCacheRefs = new LinkedList<>
();
  protected final Collection<ResultMapResolver> incompleteResultMaps = new LinkedList<>
();
  protected final Collection<MethodResolver> incompleteMethods = new LinkedList<>();
```

```
}
```

参考

mybatis-spring-boot-autoconfigure:2.0.0 Mybatis 源码分析 —— MappedStatement 解析