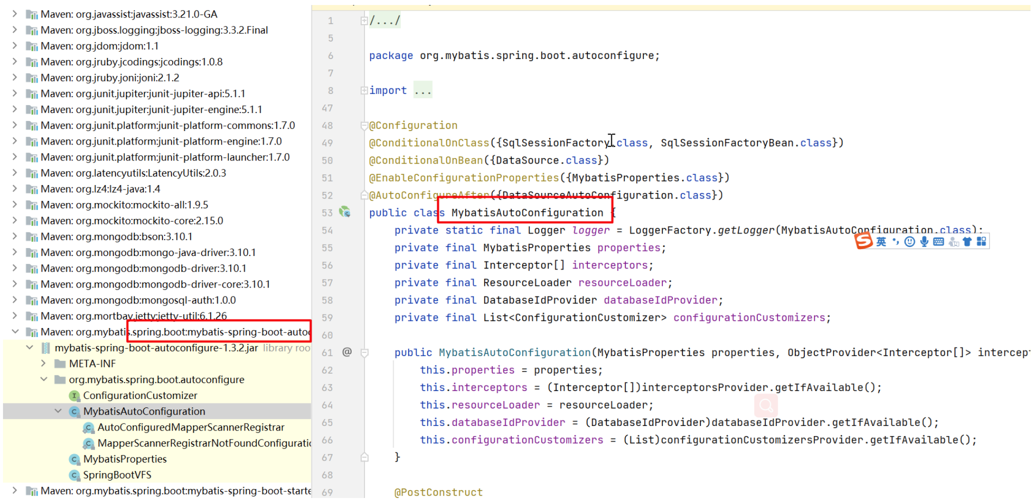
MyBatis是如何与Spring-boot整合在一起的

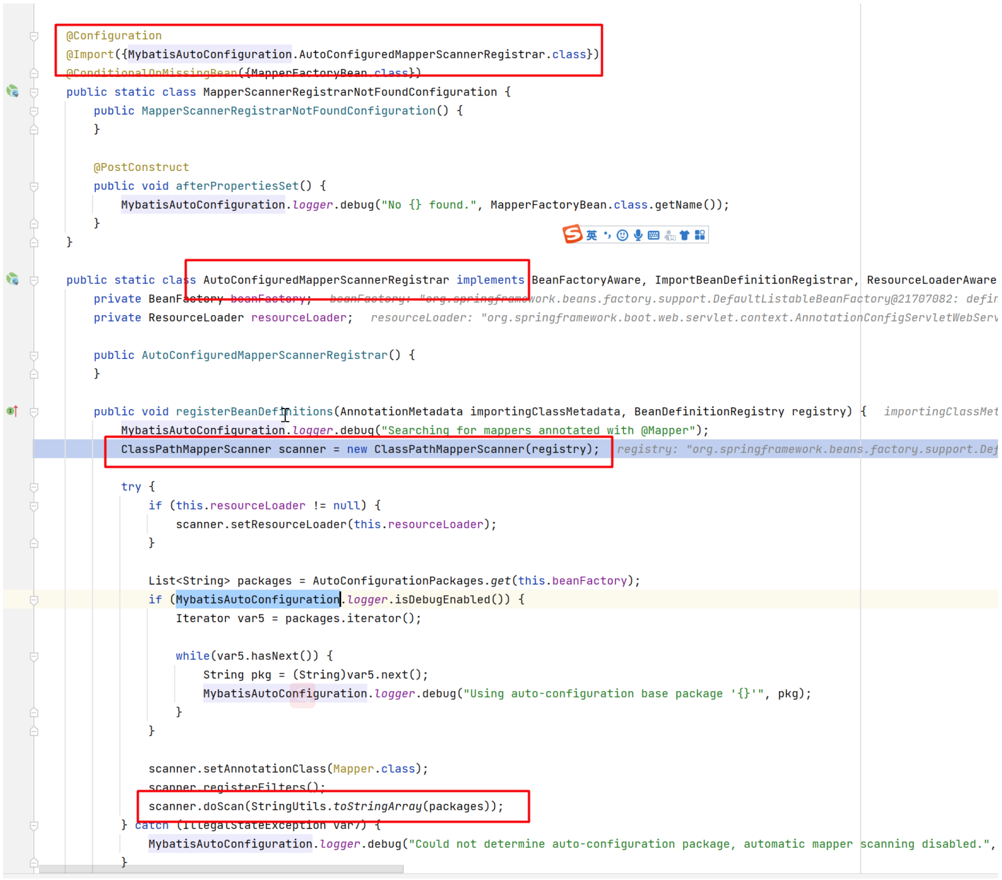
1，首先有一个AutoConfiguration



```
> Maven: org.javassist:javassist:3.21.0-GA
> Maven: org.jboss.logging:jboss-logging:3.3.2.Final
> Maven: org.jdom:jdom:1.1
> Maven: org.jruby.jcodings:jcodings:1.0.8
> Maven: org.jruby.joni:joni:2.1.2
> Maven: org.junit.jupiter:junit-jupiter-api:5.1.1
> Maven: org.junit.jupiter:junit-jupiter-engine:5.1.1
> Maven: org.junit.platform:junit-platform-commons:1.7.0
> Maven: org.junit.platform:junit-platform-engine:1.7.0
> Maven: org.junit.platform:junit-platform-launcher:1.7.0
> Maven: org.latencyutils:LatencyUtils:2.0.3
> Maven: org.lz4:lz4-java:1.4
> Maven: org.mockito:mockito-all:1.9.5
> Maven: org.mockito:mockito-core:2.15.0
> Maven: org.mongodb:bson:3.10.1
> Maven: org.mongodb:mongo-java-driver:3.10.1
> Maven: org.mongodb:mongodb-driver:3.10.1
> Maven: org.mongodb:mongodb-driver-core:3.10.1
> Maven: org.mongodb:mongosql-auth:1.0.0
> Maven: org.mortbay.jetty:jetty-util:6.1.26
v Maven: org.mybatis.spring.boot:mybatis-spring-boot-autoc
  v mybatis-spring-boot-autoconfigure-1.3.2.jar library roo
    > META-INF
    v org.mybatis.spring.boot.autoconfigure
        ConfigurationCustomizer
      v MybatisAutoConfiguration
          AutoConfiguredMapperScannerRegistrar
          MapperScannerRegistrarNotFoundConfiguratic
        MybatisProperties
        SpringBootVFS
> Maven: org.mybatis.spring.boot:mybatis-spring-boot-starte
```

```java
/.../

package org.mybatis.spring.boot.autoconfigure;

import ...

@Configuration
@ConditionalOnClass({SqlSessionFactory.class, SqlSessionFactoryBean.class})
@ConditionalOnBean({DataSource.class})
@EnableConfigurationProperties({MybatisProperties.class})
@AutoConfigureAfter({DataSourceAutoConfiguration.class})
public class MybatisAutoConfiguration {
    private static final Logger logger = LoggerFactory.getLogger(MybatisAutoConfiguration.class);
    private final MybatisProperties properties;
    private final Interceptor[] interceptors;
    private final ResourceLoader resourceLoader;
    private final DatabaseIdProvider databaseIdProvider;
    private final List<ConfigurationCustomizer> configurationCustomizers;

    public MybatisAutoConfiguration(MybatisProperties properties, ObjectProvider<Interceptor[]> intercep
        this.properties = properties;
        this.interceptors = (Interceptor[])interceptorsProvider.getIfAvailable();
        this.resourceLoader = resourceLoader;
        this.databaseIdProvider = (DatabaseIdProvider)databaseIdProvider.getIfAvailable();
        this.configurationCustomizers = (List)configurationCustomizersProvider.getIfAvailable();
    }

    @PostConstruct
```

2，然后在这个AutoConfiguration内部又有一个@Configuration，这个Configuration使用了@Import 注解，指定为AutoConfiguredMapperScanerRegister



```java
@Configuration
@Import({MybatisAutoConfiguration.AutoConfiguredMapperScannerRegistrar.class})
@ConditionalOnMissingBean({MapperFactoryBean.class})
public static class MapperScannerRegistrarNotFoundConfiguration {
    public MapperScannerRegistrarNotFoundConfiguration() {
    }

    @PostConstruct
    public void afterPropertiesSet() {
        MybatisAutoConfiguration.logger.debug("No {} found.", MapperFactoryBean.class.getName());
    }
}

public static class AutoConfiguredMapperScannerRegistrar implements BeanFactoryAware, ImportBeanDefinitionRegistrar, ResourceLoaderAware
    private BeanFactory beanFactory;   beanFactory: "org.springframework.beans.factory.support.DefaultListableBeanFactory@21707082: defir
    private ResourceLoader resourceLoader;   resourceLoader: "org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServ

    public AutoConfiguredMapperScannerRegistrar() {
    }

    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {   importingClassMet
        MybatisAutoConfiguration.logger.debug("Searching for mappers annotated with @Mapper");
        ClassPathMapperScanner scanner = new ClassPathMapperScanner(registry);   registry: "org.springframework.beans.factory.support.Def

        try {
            if (this.resourceLoader != null) {
                scanner.setResourceLoader(this.resourceLoader);
            }

            List<String> packages = AutoConfigurationPackages.get(this.beanFactory);
            if (MybatisAutoConfiguration.logger.isDebugEnabled()) {
                Iterator var5 = packages.iterator();

                while(var5.hasNext()) {
                    String pkg = (String)var5.next();
                    MybatisAutoConfiguration.logger.debug("Using auto-configuration base package '{}'", pkg);
                }
            }

            scanner.setAnnotationClass(Mapper.class);
            scanner.registerFilters();
            scanner.doScan(StringUtils.toStringArray(packages));
        } catch (IllegalStateException var7) {
            MybatisAutoConfiguration.logger.debug("Could not determine auto-configuration package, automatic mapper scanning disabled.",
        }
```

在AutoConfiguredMapperScannerRegistrar的 registerBeanDefinition方法内部会 创建一个ClassPathMapperScanner对象， 然后用这个对象的doScan方法去扫描有哪些Mapper接口需要创建代理对象。

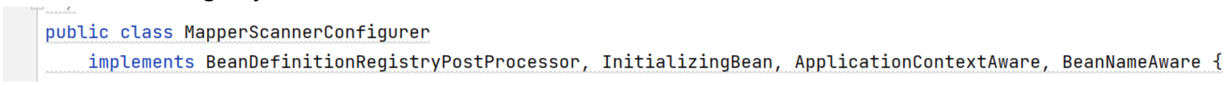
3， 如果是单独使用mybatis，也就是不使用spring-boot，而是使用spring和mybatis整合，一般我们会创建一个Mybatis-spring包中提供的MapperScannerConfigurer对象，我们会为这个配置一个basePackage，然后这个对象就会扫描指定包下的接口，为接口创建代理对象，同时我们还可以为对象MapperScannerConfigurer指定annotationClass属性，然后MapperScannerConfigurer对象就会扫描指定包下接口上使用了指定注解的接口创建代理对象。

比如下面这段测试代码

```
applicationContext = new GenericApplicationContext();
// add the mapper scanner as a bean definition rather than explicitly setting a
// postProcessor on the context so initialization follows the same code path as reading from
// an XML config file
GenericBeanDefinition definition = new GenericBeanDefinition();
definition.setBeanClass(MapperScannerConfigurer.class);
//指定扫描包
definition.getPropertyValues().add("basePackage", "org.mybatis.spring.mapper");
applicationContext.registerBeanDefinition("mapperScanner", definition);
applicationContext.getBeanFactory().registerScope("thread", new SimpleThreadScope());
//创建一个BeanDefinition
GenericBeanDefinition definition = new GenericBeanDefinition();
definition.setBeanClass(SqlSessionFactoryBean.class);
definition.getPropertyValues().add("dataSource", new MockDataSource());
applicationContext.registerBeanDefinition( "sqlSessionFactory" , definition);
applicationContext.refresh();
applicationContext.start();
```

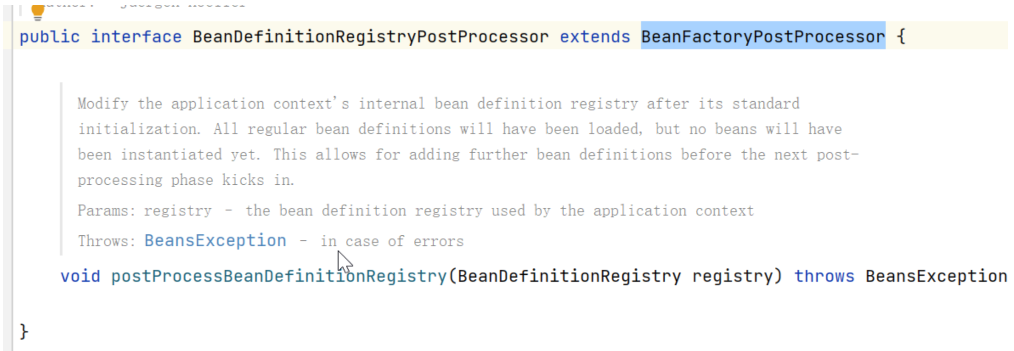
在这段代码中MapperScannerConfigurer对象就会扫描指定包下的接口，为接口创建代理对象。

4，MapperScannerConfigurer 是如何实现为接口创建代理对象的?
MapperScannerConfigurer对象的本质是spring-mybatis包中为了提供在Spring中使用Mybatis而创建的一个BeanDefinitionRegistryPostProcessor



```
public class MapperScannerConfigurer
    implements BeanDefinitionRegistryPostProcessor, InitializingBean, ApplicationContextAware, BeanNameAware {
```

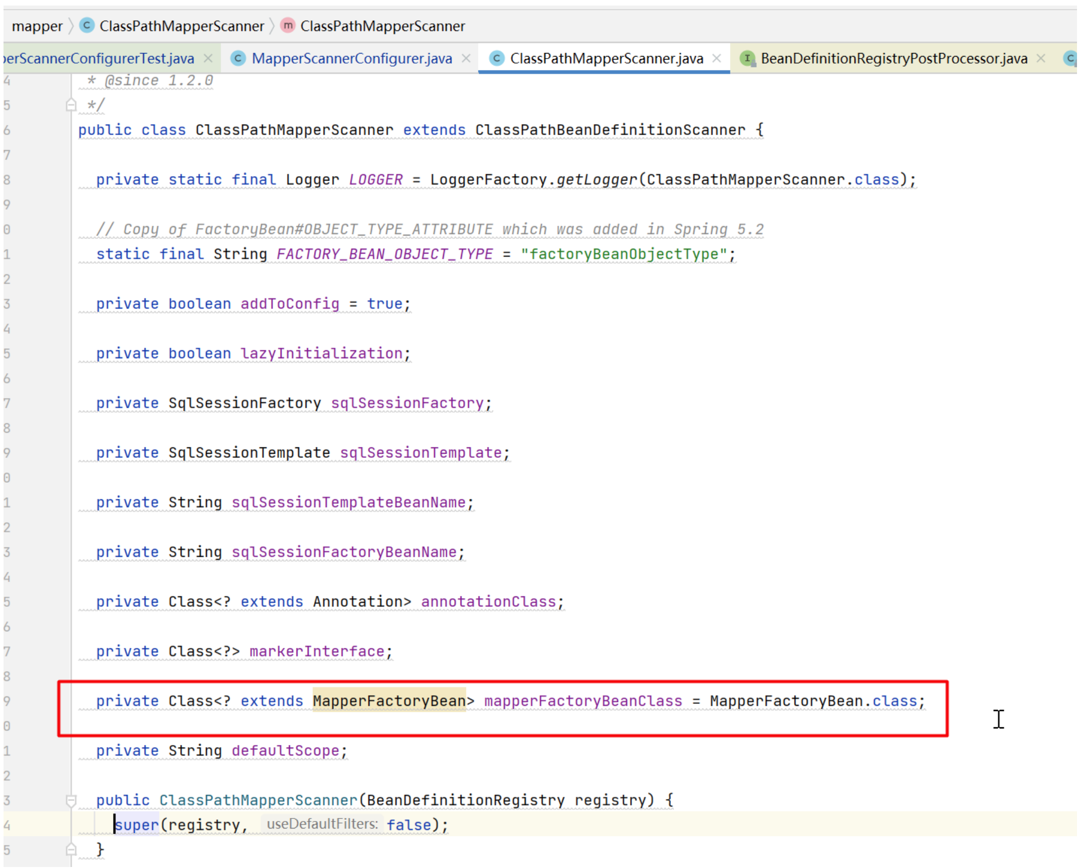
这个BeanDefinitionRegistryPostProcessor 是一个BeanDefinition相关的PostProcessor，一般而言和BeanDefinition相关的PostProcessor都是BeanFactoryPostProcessor。



```
public interface BeanDefinitionRegistryPostProcessor extends BeanFactoryPostProcessor {

    Modify the application context's internal bean definition registry after its standard
    initialization. All regular bean definitions will have been loaded, but no beans will have
    been instantiated yet. This allows for adding further bean definitions before the next post-
    processing phase kicks in.
    Params: registry – the bean definition registry used by the application context
    Throws: BeansException – in case of errors
    void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException

}
```

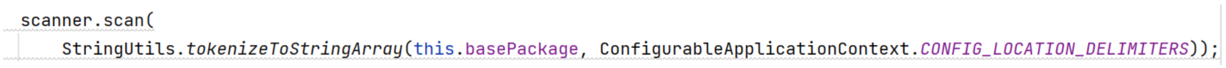
因此在容器启动的时候会执行MapperScannerConfigurer对象的postProcessBeanDefinitionRegistry方法
在这个方法我们会扫描basePackage指定的包下的接口，然后为每一个接口创建一个BeanDefinition，后续我们将会使用这个BeanDefinition创建接口的代理对象， 我们的关注点是BeanDefinition是如何创建的。

在创建ClassPathMapperScanner的时候 ClassPathMapperScanner内有一个重要的属性mapperFactoryBeanClass

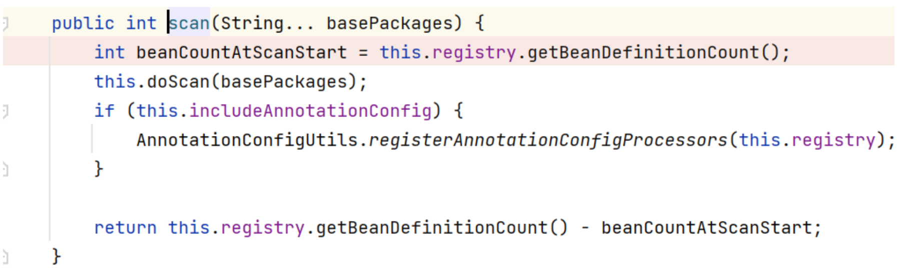

mapper 〉 ClassPathMapperScanner 〉 ClassPathMapperScanner
erScannerConfigurerTest.java
MapperScannerConfigurer.java
ClassPathMapperScanner.java
BeanDefinitionRegistryPostProcessor.java

```java
 * @since 1.2.0
 */
public class ClassPathMapperScanner extends ClassPathBeanDefinitionScanner {

  private static final Logger LOGGER = LoggerFactory.getLogger(ClassPathMapperScanner.class);

  // Copy of FactoryBean#OBJECT_TYPE_ATTRIBUTE which was added in Spring 5.2
  static final String FACTORY_BEAN_OBJECT_TYPE = "factoryBeanObjectType";

  private boolean addToConfig = true;

  private boolean lazyInitialization;

  private SqlSessionFactory sqlSessionFactory;

  private SqlSessionTemplate sqlSessionTemplate;

  private String sqlSessionTemplateBeanName;

  private String sqlSessionFactoryBeanName;

  private Class<? extends Annotation> annotationClass;

  private Class<?> markerInterface;

  private Class<? extends MapperFactoryBean> mapperFactoryBeanClass = MapperFactoryBean.class;

  private String defaultScope;

  public ClassPathMapperScanner(BeanDefinitionRegistry registry) {
    super(registry,  useDefaultFilters: false);
  }
```

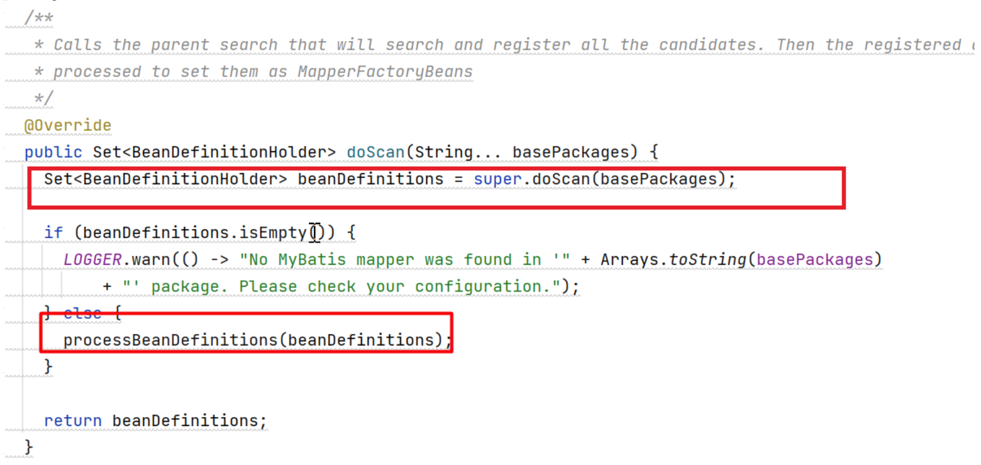
然后我们会使用ClassPathMapperScanner的scan方法进行扫描包，scan方法会调用doScan



```java
 scanner.scan(
     StringUtils.tokenizeToStringArray(this.basePackage, ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS));
```

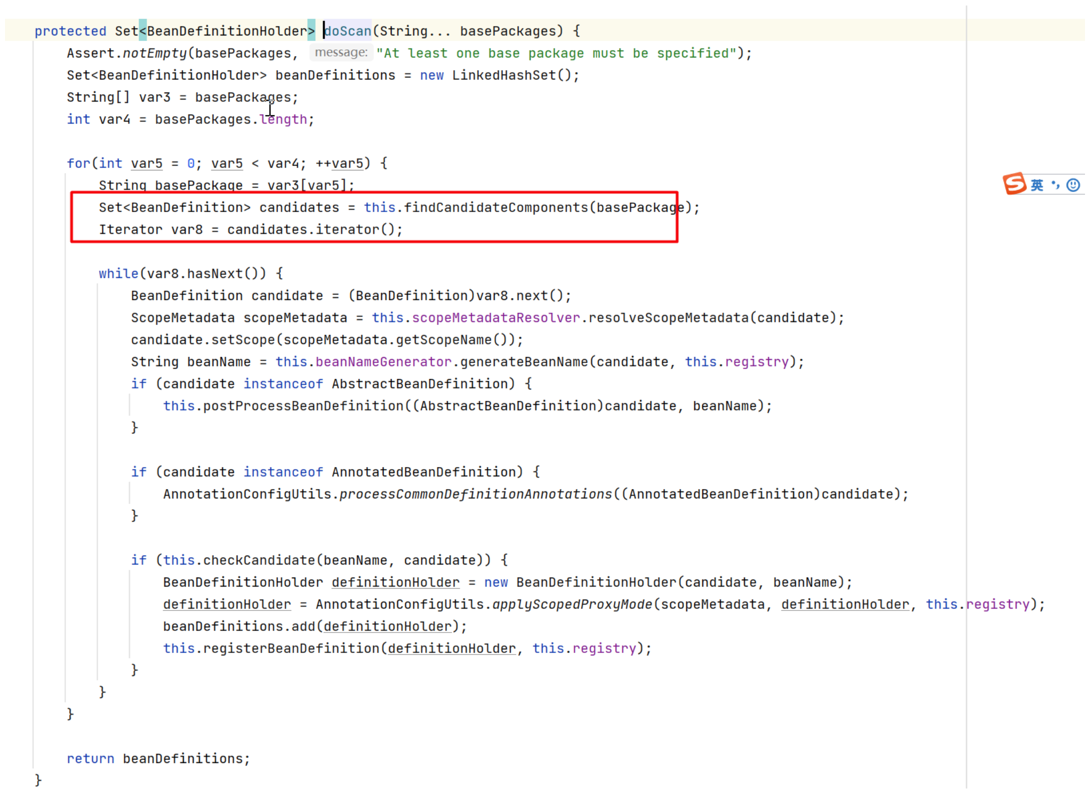


```java
    public int scan(String... basePackages) {
        int beanCountAtScanStart = this.registry.getBeanDefinitionCount();
        this.doScan(basePackages);
        if (this.includeAnnotationConfig) {
            AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
        }

        return this.registry.getBeanDefinitionCount() - beanCountAtScanStart;
    }
```

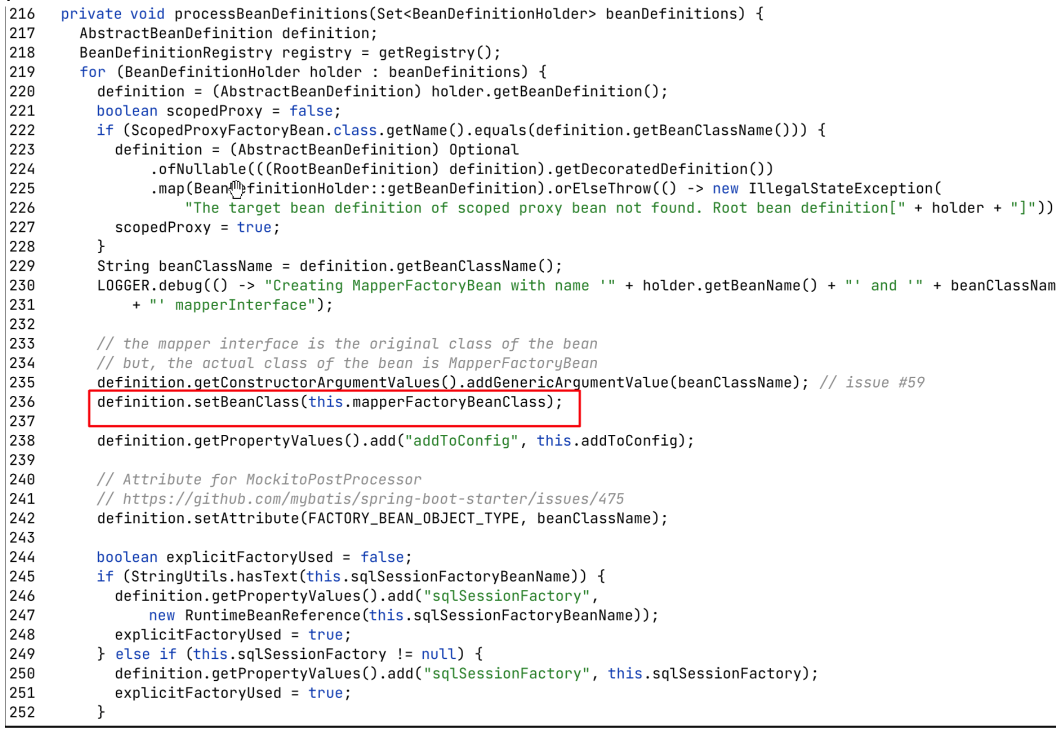


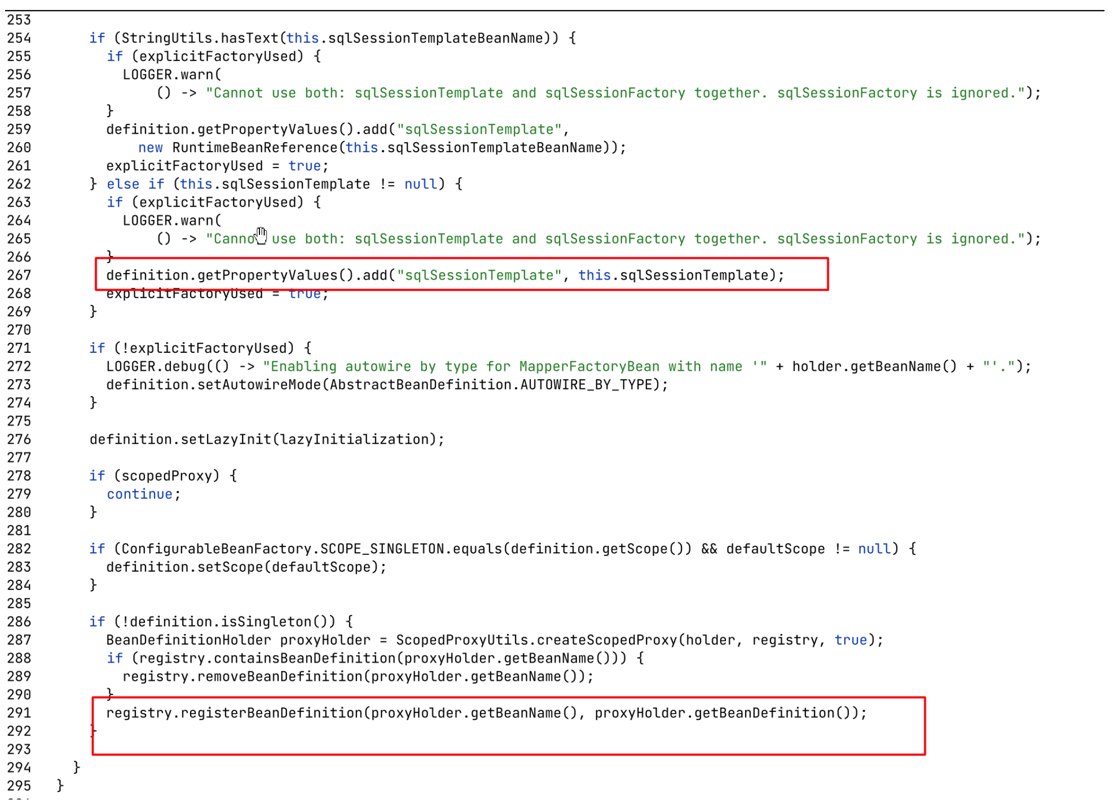
```java
  /**
   * Calls the parent search that will search and register all the candidates. Then the registered
   * processed to set them as MapperFactoryBeans
   */
  @Override
  public Set<BeanDefinitionHolder> doScan(String... basePackages) {
    Set<BeanDefinitionHolder> beanDefinitions = super.doScan(basePackages);

    if (beanDefinitions.isEmpty()) {
      LOGGER.warn(() -> "No MyBatis mapper was found in '" + Arrays.toString(basePackages)
          + "' package. Please check your configuration.");
    } else {
      processBeanDefinitions(beanDefinitions);
    }

    return beanDefinitions;
  }
```

在doScan方法内部会首先调用super.doScan 获取到符合条件的BeanDefinition，然后调用processBeanDefinitions方法


```java
protected Set<BeanDefinitionHolder> doScan(String... basePackages) {
    Assert.notEmpty(basePackages,  message: "At least one base package must be specified");
    Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet();
    String[] var3 = basePackages;
    int var4 = basePackages.length;

    for(int var5 = 0; var5 < var4; ++var5) {
        String basePackage = var3[var5];
        Set<BeanDefinition> candidates = this.findCandidateComponents(basePackage);
        Iterator var8 = candidates.iterator();

        while(var8.hasNext()) {
            BeanDefinition candidate = (BeanDefinition)var8.next();
            ScopeMetadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMetadata(candidate);
            candidate.setScope(scopeMetadata.getScopeName());
            String beanName = this.beanNameGenerator.generateBeanName(candidate, this.registry);
            if (candidate instanceof AbstractBeanDefinition) {
                this.postProcessBeanDefinition((AbstractBeanDefinition)candidate, beanName);
            }

            if (candidate instanceof AnnotatedBeanDefinition) {
                AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinition)candidate);
            }

            if (this.checkCandidate(beanName, candidate)) {
                BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(candidate, beanName);
                definitionHolder = AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder, this.registry);
                beanDefinitions.add(definitionHolder);
                this.registerBeanDefinition(definitionHolder, this.registry);
            }
        }
    }

    return beanDefinitions;
}
```
在super.doScan方法中我们看到 首先是调用findCandidateComponents方法 根据指定的basePackage 和指定的annotationClass找到指定包下符合条件的类的BeanDefinition。如果这个BeanDefinition是一个AbstractBeanDefinition，我们就调用postProcessBeanDefinition方法进行处理.

同时我们需要注意在ClassPathMapperScanner的doscan方法内部除了调用了super.doScan方法外，还调用了processBeanDefinitions方法处理获取到的BeanDefinition。


```java
private void processBeanDefinitions(Set<BeanDefinitionHolder> beanDefinitions) {
  AbstractBeanDefinition definition;
  BeanDefinitionRegistry registry = getRegistry();
  for (BeanDefinitionHolder holder : beanDefinitions) {
    definition = (AbstractBeanDefinition) holder.getBeanDefinition();
    boolean scopedProxy = false;
    if (ScopedProxyFactoryBean.class.getName().equals(definition.getBeanClassName())) {
      definition = (AbstractBeanDefinition) Optional
          .ofNullable(((RootBeanDefinition) definition).getDecoratedDefinition())
          .map(BeanDefinitionHolder::getBeanDefinition).orElseThrow(() -> new IllegalStateException(
              "The target bean definition of scoped proxy bean not found. Root bean definition[" + holder + "]"));
      scopedProxy = true;
    }
    String beanClassName = definition.getBeanClassName();
    LOGGER.debug(() -> "Creating MapperFactoryBean with name '" + holder.getBeanName() + "' and '" + beanClassNam
        + "' mapperInterface");

    // the mapper interface is the original class of the bean
    // but, the actual class of the bean is MapperFactoryBean
    definition.getConstructorArgumentValues().addGenericArgumentValue(beanClassName); // issue #59
    definition.setBeanClass(this.mapperFactoryBeanClass);

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

    // Attribute for MockitoPostProcessor
    // https://github.com/mybatis/spring-boot-starter/issues/475
    definition.setAttribute(FACTORY_BEAN_OBJECT_TYPE, beanClassName);

    boolean explicitFactoryUsed = false;
    if (StringUtils.hasText(this.sqlSessionFactoryBeanName)) {
      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) {
        LOGGER.debug(() -> "Enabling autowire by type for MapperFactoryBean with name '" + holder.getBeanName() + "'.");
        definition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_TYPE);
      }

      definition.setLazyInit(lazyInitialization);

      if (scopedProxy) {
        continue;
      }

      if (ConfigurableBeanFactory.SCOPE_SINGLETON.equals(definition.getScope()) && defaultScope != null) {
        definition.setScope(defaultScope);
      }

      if (!definition.isSingleton()) {
        BeanDefinitionHolder proxyHolder = ScopedProxyUtils.createScopedProxy(holder, registry, true);
        if (registry.containsBeanDefinition(proxyHolder.getBeanName())) {
          registry.removeBeanDefinition(proxyHolder.getBeanName());
        }
        registry.registerBeanDefinition(proxyHolder.getBeanName(), proxyHolder.getBeanDefinition());
      }

    }
  }
```

从上面的代码中我们看到我们给这个BeanDefinition设置了Beanclass为

definition.setBeanClass(this.mapperFactoryBeanClass);

这个mapperFactoryBeanClass 就是

private Class<? extends MapperFactoryBean> mapperFactoryBeanClass = MapperFactoryBean.class;

MapperFactoryBean类是spring-mybatis中定义的一个FactoryBean，也就是工厂Bean，工厂Bean的作用就是通过getObject方法创建对象，因此MapperFactoryBean的作用就是创建一个Mapper接口的代理对象

```
public class MapperFactoryBean<T> extends SqlSessionDaoSupport implements FactoryBean<T> {
private Class<T> mapperInterface;
@Override
public T getObject() throws Exception {
  return getSqlSession().getMapper(this.mapperInterface);
}
}
```

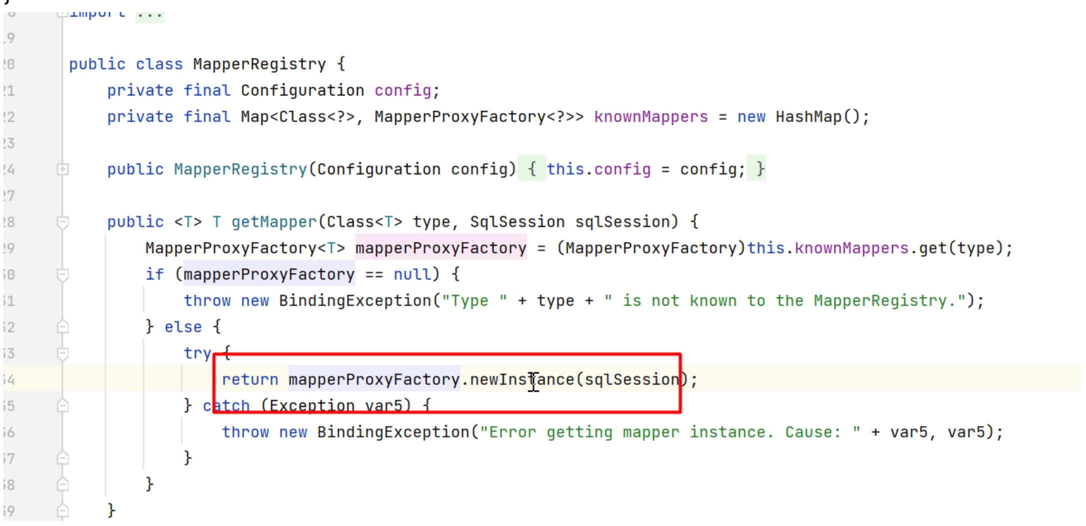
MapperFactoryBean中有一个mapperInterface属性，代表该工厂是创建哪一个Mapper接口的代理对象。MapperFactoryBean的getObject方法能够返回接口的代理对象，其内部是委托给了SqlSession对象的getMapper方法实现。这个SQLSession实际上是sqlSessionTemplate对象。

在SqlSessionTemplate的getMapper方法中我们看到getMapper实际上是委托给了Mybatis的Configuration的getMapper实现

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

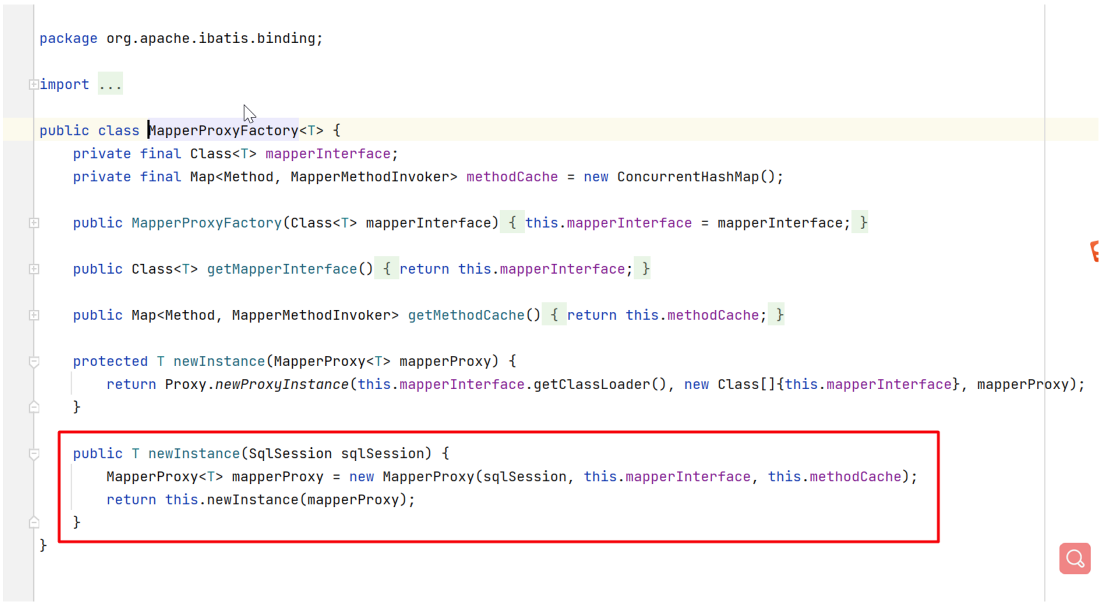
Configuration的getMapper委托给MapperRegistry的getMapper实现

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



```java
import ...

public class MapperRegistry {
    private final Configuration config;
    private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap();

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

    public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
        MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory)this.knownMappers.get(type);
        if (mapperProxyFactory == null) {
            throw new BindingException("Type " + type + " is not known to the MapperRegistry.");
        } else {
            try {
                return mapperProxyFactory.newInstance(sqlSession);
            } catch (Exception var5) {
                throw new BindingException("Error getting mapper instance. Cause: " + var5, var5);
            }
        }
    }
```

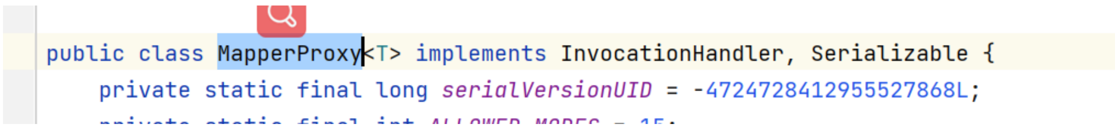
这个MappperProxyFactory是Mybatis提供的， 其newInstance方法内首先创建了一个MapperProxy对象



```java
package org.apache.ibatis.binding;

import ...

public class MapperProxyFactory<T> {
    private final Class<T> mapperInterface;
    private final Map<Method, MapperMethodInvoker> methodCache = new ConcurrentHashMap();

    public MapperProxyFactory(Class<T> mapperInterface) { this.mapperInterface = mapperInterface; }

    public Class<T> getMapperInterface() { return this.mapperInterface; }

    public Map<Method, MapperMethodInvoker> getMethodCache() { return this.methodCache; }

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

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

这个MapperProxy对象本质上是一个InvocationHandler对象，因此我们最终就是使用了MapperProxy作为InvocationHandler创建了一个JDK代理对象

Proxy.*newProxyInstance*(this.mapperInterface.getClassLoader(), new Class[]{this.mapperInterface}, mapperProxy);。 这个JDK代理对象代理的接口就是Mapper接口，使用MapperProxy作为InvocationHandler



```java
public class MapperProxy<T> implements InvocationHandler, Serializable {
    private static final long serialVersionUID = -4724728412955527868L;
```