分析之前，先看先Mybatis的通用使用方法：

```
1.  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(this.getCl
    ass().getClassLoader().getResourceAsStream("mybatis-config.xml"));
2.  SqlSession session = sqlSessionFactory.openSession();
3.  CountryMapper cm = session.getMapper(CountryMapper.class);
4.  cm.queryCountryByCode("AFG");
5.  .....
```

由上面的代码可以看到，集成的重点有两个：
1. 如何在Spring中生成Mybatis的SqlSessionFactory实例对象；
2. 如何将Mybatis的代理mapper对象交由Spring的ApplicationContext容器来管理，即如何通过Spring来获取Mybatis的mapper代理对象。

一、首先看配置：

```
1.  <!-- 数据源配置 -->
2.  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
3.  <property name="driverClassName" value="${jdbc.mysql.driverClassName}"></property>
4.  <property name="url" value="${jdbc.mysql.url}"></property>
5.  <property name="username" value="${jdbc.mysql.username}"></property>
6.  <property name="password" value="${jdbc.mysql.password}"></property>
7.  </bean>
8.
9.  <!-- 配置Mybatis sessionFactory -->
10. <bean id="mybatisSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
11. <property name="dataSource" ref="dataSource"></property>
12. <property name="configLocation" value="classpath:mybatis-config.xml"></property>
13. </bean>
14.
15. <!--  配置了Mapper接口所在的包路径，用于加载相关的mapper接口 -->
16. <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
      <property name="basePackage" value="lxy.study.core.mappers"></property>
    </bean>
```

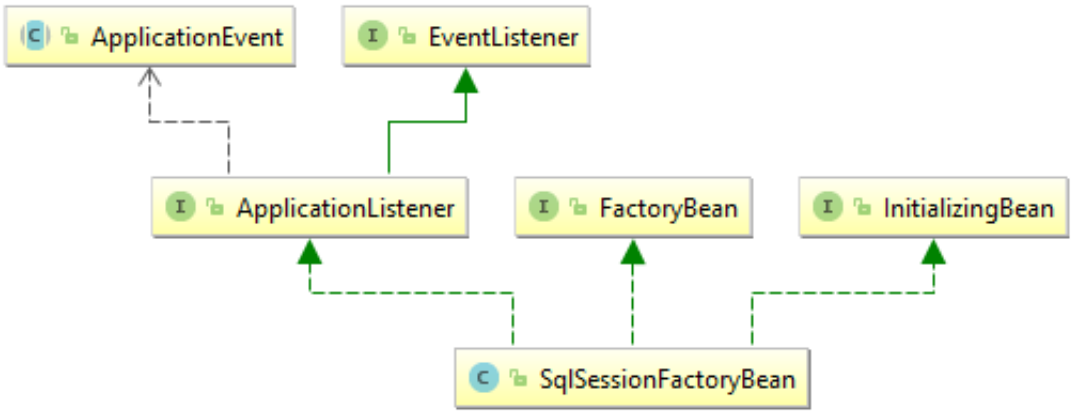
1.1 首先配置的是一个数据源，即数据库信息，这个和在单独使用Mybatis的时候是一样的，只是数据源的类型有所不一样，这里不讲；
1.2 配置Mybatis的SqlSessionFactory，这是咱们的重点。
从配置上看，很简单，就配置了一个dataSource，引用了1.1中配置的数据源，然后配置了一个Mybatis的配置文件，就算完事；具体后面会根据源码进行分析；
1.3 配置了Mapper接口所在的包路径，用于加载相关的mapper接口。
1.4 Spring在获取相关的Mapper代理对象时，具体的操作过程是怎样的？

本文会重点围绕1.2~1.4这3点重点分析，做到深入理解原理/机制。

二、生成MyBatis的SqlSessionFactory对象

从配置上看，我们首先不得不关注org.mybatis.spring.SqlSessionFactoryBean这个类，这个类的最终目标是生成Mybatis的SqlSessionFactory对象。
二话不说，我们打开源码。

2.1 首先看下其UML类图



由类图我们看到，这个SqlSessionFactoryBean实际上是实现了众多接口（如：FactoryBean）接口的对象，这里我们重点关注的是，该类是一个Spring的FactoryBean对象。至于什么是Spring的FactoryBean，什么是Spring的BeanFactory，详情大家可以自行百度一下。这里只做简单介绍
FactoryBean：实现该接口类，会被Spring当作一类特殊Java Bean对象，在Spring实例化该类Bean对象的时候，会调用该实例的getObject()方法，获取该方法返回的对象B，并将该对象B作为与配置中的id关联起来。所以，如下配置中，在Spring中实际上生成的不是SqlSessionFactoryBean这个Bean对象，而是这个SqlSessionFactoryBean类的getObject方法返回的对象。具体**Spring**的源码分析，可以参考**Spring**的**AbstractBeanFactory**的**getObjectForBeanInstance**方法。下面也做简单分析：

```
<!-- 配置Mybatis sessionFactory -->
<bean id="mybatisSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
<property name="dataSource" ref="dataSource"></property>
<property name="configLocation" value="classpath:mybatis-config.xml"></property>
</bean>
```

这就是为什么，通过这个配置，咱们能获取到Mybatis的SqlSessionFactory对象的原因了。

Spring的**AbstractBeanFactory**的**getObjectForBeanInstance**

FactoryBeanRegistrySupport.**getObjectFromFactoryBean**方法的源码

```java
/**
 * Obtain an object to expose from the given FactoryBean.
 * @param factory the FactoryBean instance
 * @param beanName the name of the bean
 * @param shouldPostProcess whether the bean is subject to post-processing
 * @return the object obtained from the FactoryBean
 * @throws BeanCreationException if FactoryBean object creation failed
 * @see org.springframework.beans.factory.FactoryBean#getObject()
 */
protected Object getObjectFromFactoryBean(FactoryBean<?> factory, String beanName, boolean shouldPostProcess) {
  if (factory.isSingleton() && containsSingleton(beanName)) {
    synchronized (getSingletonMutex()) {
      Object object = this.factoryBeanObjectCache.get(beanName);
      if (object == null) {
        object = doGetObjectFromFactoryBean(factory, beanName);
        // Only post-process and store if not put there already during getObject() call above
        // (e.g. because of circular reference processing triggered by custom getBean calls)
        Object alreadyThere = this.factoryBeanObjectCache.get(beanName);
        if (alreadyThere != null) {
          object = alreadyThere;
        }
        else {
          if (object != null && shouldPostProcess) {
            try {
              object = postProcessObjectFromFactoryBean(object, beanName);
            }
            catch (Throwable ex) {
              throw new BeanCreationException(beanName,
                  "Post-processing of FactoryBean's singleton object failed", ex);
            }
          }
          this.factoryBeanObjectCache.put(beanName, (object != null ? object : NULL_OBJECT));
        }
      }
      return (object != NULL_OBJECT ? object : null);
    }
  }
  else {
    Object object = doGetObjectFromFactoryBean(factory, beanName);
    if (object != null && shouldPostProcess) {
      try {
        object = postProcessObjectFromFactoryBean(object, beanName);
      }
      catch (Throwable ex) {
        throw new BeanCreationException(beanName, "Post-processing of FactoryBean's object failed", ex);
      }
    }
    return object;
  }
}
```

FactoryBeanRegistrySupport.doGetObjectFromFactoryBean源码：

```java
/**
 * Obtain an object to expose from the given FactoryBean.
 * @param factory the FactoryBean instance
 * @param beanName the name of the bean
 * @return the object obtained from the FactoryBean
 * @throws BeanCreationException if FactoryBean object creation failed
 * @see org.springframework.beans.factory.FactoryBean#getObject()
 */
private Object doGetObjectFromFactoryBean(final FactoryBean<?> factory, final String beanName)
    throws BeanCreationException {

  Object object;
  try {
    if (System.getSecurityManager() != null) {
      AccessControlContext acc = getAccessControlContext();
      try {
        object = AccessController.doPrivileged((PrivilegedExceptionAction) () -> {
              return factory.getObject();
            }, acc);
      }
      catch (PrivilegedActionException pae) {
        throw pae.getException();
      }
    }
    else {
      object = factory.getObject();
    }
  }
  catch (FactoryBeanNotInitializedException ex) {
    throw new BeanCurrentlyInCreationException(beanName, ex.toString());
  }
  catch (Throwable ex) {
    throw new BeanCreationException(beanName, "FactoryBean threw exception on object creation", ex);
  }

  // Do not accept a null value for a FactoryBean that's not fully
  // initialized yet: Many FactoryBeans just return null then.
  if (object == null && isSingletonCurrentlyInCreation(beanName)) {
    throw new BeanCurrentlyInCreationException(
        beanName, "FactoryBean which is currently in creation returned null from getObject");
  }
  return object;
}
```

至此，柳暗花明。

2.2 解开SqlSessionFactory对象生成的神秘面纱

分析这个问题，看源码是硬道理，我如下截取了SqlSessionFactoryBean部分核心源代码，具体的详情，大家可以自行去下载。

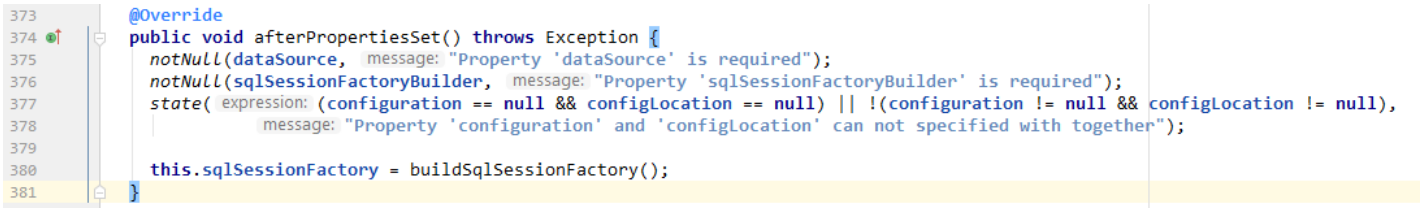
首先查看**getObject**方法的实现：

```java
  */
@Override
public SqlSessionFactory getObject() throws Exception {
  if (this.sqlSessionFactory == null) {
    afterPropertiesSet();
  }

  return this.sqlSessionFactory;
}
```

这里可以很明显的看到，这个方法返回的就是对象内部的sqlSessionFactory对象。获取逻辑，如果sqlSessionFactory为空，则会调用afterPropertiesSet方法（实际上这个方法是InitializingBean的接口方法）。

afterPropertiesSet源码



可以看到，该方法中，首先是对dataSource/sqlSessionFactoryBuilder对象判空校验，同时校验configuration/configLocation，二者既不能同时为空，也不能同时都有值（否则会使用默认的Mybatis配置信息），一句话，就是说“有且只能有一个对象是有值的”，为什么这样？留给读者自己思考（内部就是根据这两个配置去读取mybatis的配置文件的）。然后再调用buildSqlSessionFactory方法，来进行工厂的创建。

buildSqlSessionFactory源码

```
1.  /**
2.  * Build a {@code SqlSessionFactory} instance.
3.  *
4.  * The default implementation uses the standard MyBatis {@code XMLConfigBuilder} API t
    o build a
5.  * {@code SqlSessionFactory} instance based on an Reader.
6.  * Since 1.3.0, it can be specified a {@link Configuration} instance directly(without
    config file).
7.  *
8.  * @return SqlSessionFactory
9.  * @throws IOException if loading the config file failed
10. */
11. protected SqlSessionFactory buildSqlSessionFactory() throws IOException {
12.
13. Configuration configuration;
14.
15. // 构建xmlConfigBuilder对象
16. XMLConfigBuilder xmlConfigBuilder = null;
17. // 从这里我想大家都能看出来为什么configuration 和 configLocation 不能同时有值，有不能同时
    为空了吧，因为如果这样的话，
18. // 会导致程序无法加载配置，或者同时都配置了的话，会优先使用configuration 的配置文档二给使
    用者造成迷惑。
19. if (this.configuration != null) {
20.     // 使用外部配置的configuration对象，然后进行相关的参数和属性设置
21.      configuration = this.configuration;
22.      if (configuration.getVariables() == null) {
23.          configuration.setVariables(this.configurationProperties);
24.      } else if (this.configurationProperties != null) {
```

```
            configuration.getVariables().putAll(this.configurationProperties);
        }
    } else if (this.configLocation != null) {
        // 否则使用传入的configLocation，获取Mybatis的配置文件，从而生成configuration对象
        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");
        }
        // 使用默认配置
        configuration = new Configuration();
        if (this.configurationProperties != null) {
            configuration.setVariables(this.configurationProperties);
        }
    }

    // 以下进行各种参数的设置
    if (this.objectFactory != null) {
        configuration.setObjectFactory(this.objectFactory);
    }

    if (this.objectWrapperFactory != null) {
        configuration.setObjectWrapperFactory(this.objectWrapperFactory);
    }

    if (this.vfs != null) {
        configuration.setVfsImpl(this.vfs);
    }

    // 读取Alias配置
    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 + "'");
```

```
73.   }
74.   }
75.   }
76.
77.   // 读取插件配置
78.   if (!isEmpty(this.plugins)) {
79.   for (Interceptor plugin : this.plugins) {
80.   configuration.addInterceptor(plugin);
81.   if (LOGGER.isDebugEnabled()) {
82.   LOGGER.debug("Registered plugin: '" + plugin + "'");
83.   }
84.   }
85.   }
86.
87.   // 读取和设置typeHandler
88.   if (hasLength(this.typeHandlersPackage)) {
89.   String[] typeHandlersPackageArray = tokenizeToStringArray(this.typeHandlersPackage,
90.   ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS);
91.   for (String packageToScan : typeHandlersPackageArray) {
92.   configuration.getTypeHandlerRegistry().register(packageToScan);
93.   if (LOGGER.isDebugEnabled()) {
94.   LOGGER.debug("Scanned package: '" + packageToScan + "' for type handlers");
95.   }
96.   }
97.   }
98.
99.   if (!isEmpty(this.typeHandlers)) {
100.  for (TypeHandler<?> typeHandler : this.typeHandlers) {
101.  configuration.getTypeHandlerRegistry().register(typeHandler);
102.  if (LOGGER.isDebugEnabled()) {
103.  LOGGER.debug("Registered type handler: '" + typeHandler + "'");
104.  }
105.  }
106.  }
107.
108.  // 设置databaseId
109.  if (this.databaseIdProvider != null) {//fix #64 set databaseId before parse mapper xm
      ls
110.  try {
111.  configuration.setDatabaseId(this.databaseIdProvider.getDatabaseId(this.dataSource));
112.  } catch (SQLException e) {
113.  throw new NestedIOException("Failed getting a databaseId", e);
114.  }
115.  }
116.
117.  // 设置缓存
118.  if (this.cache != null) {
119.  configuration.addCache(this.cache);
120.  }
121.
```

```
122.  if (xmlConfigBuilder != null) {
123.  try {
124.  xmlConfigBuilder.parse();
125.
126.  if (LOGGER.isDebugEnabled()) {
127.  LOGGER.debug("Parsed configuration file: '" + this.configLocation + "'");
128.  }
129.  } catch (Exception ex) {
130.  throw new NestedIOException("Failed to parse config resource: " + this.configLocation
      , ex);
131.  } finally {
132.  ErrorContext.instance().reset();
133.  }
134.  }
135.
136.  // 设置事务管理工厂，用于SqlSessionFactory的事务管理对象，和Spring事务的集成还有待研究
137.  if (this.transactionFactory == null) {
138.  this.transactionFactory = new SpringManagedTransactionFactory();
139.  }
140.
141.  configuration.setEnvironment(new Environment(this.environment, this.transactionFactor
      y, this.dataSource));
142.
143.  // 读取所有的Mapper xml配置
144.  if (!isEmpty(this.mapperLocations)) {
145.  for (Resource mapperLocation : this.mapperLocations) {
146.  if (mapperLocation == null) {
147.  continue;
148.  }
149.
150.  try {
151.  XMLMapperBuilder xmlMapperBuilder = new XMLMapperBuilder(mapperLocation.getInputStrea
      m(),
152.  configuration, mapperLocation.toString(), configuration.getSqlFragments());
153.  xmlMapperBuilder.parse();
154.  } catch (Exception e) {
155.  throw new NestedIOException("Failed to parse mapping resource: '" + mapperLocation +
      "'", e);
156.  } finally {
157.  ErrorContext.instance().reset();
158.  }
159.
160.  if (LOGGER.isDebugEnabled()) {
161.  LOGGER.debug("Parsed mapper file: '" + mapperLocation + "'");
162.  }
163.  }
164.  } else {
165.  if (LOGGER.isDebugEnabled()) {
166.  LOGGER.debug("Property 'mapperLocations' was not specified or no matching resources f
      ound");
```

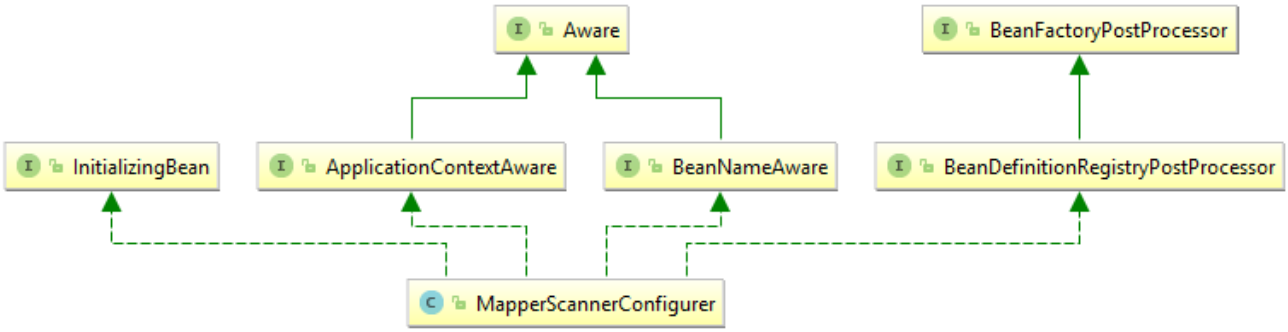
```
167.    }
168.    }
169.
170.    // 生成sqlSessionFactory对象
171.    return this.sqlSessionFactoryBuilder.build(configuration);
172.    }
```

总体过程和使用Java方式来构建Mybatis的过程是完全一样的。具体注释我写在源码里面。
经过这里的配置，Spring中就会有SqlSessionFactory的实例对象了。后续的使用过程中，可以使用自动注入的方式或者xml的配置方式来使用这个对象了。

三、MyBatis的mapper接口加载

```
43    <!-- 采用自动扫描方式创建mapper bean -->
44    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
45      <property name="basePackage" value="lxy.study.core.mappers"></property>
46    </bean>
```

由此可见，所有的mapper接口都是从这里加载的。接下来，我们看下MapperScannerConfigurer是如何实现这个功能。



从UML图看，MapperScannerConfigurer这个实现了多个接口，我们重点看下BeanDefinitionRegistryPostProcessor

BeanDefinitionRegistryPostProcessor：其官方说明是“Extension to the standard {@link BeanFactoryPostProcessor} SPI, allowing for the registration of further bean definitions before regular
BeanFactoryPostProcessor detection kicks in. In particular, BeanDefinitionRegistryPostProcessor may register further bean definitions which in turn define BeanFactoryPostProcessor instances.”
主要就是说，该接口会在所有Spring的Bean对象定义完成之后会被调用，这个时候，可以通过该接口很魔术性的改变已经定好的BeanDefinition对象相关熟悉/Class等信息，这样可以改变BeanFactory对这些BeanDefinitions实例化的行为。在Spring和Mybatis的集成当中，Mybatis正是利用了这一点来达到目的的。

接下来我们来分析一下源码，看看是如何利用这一原理来实现这一目的，我们只看重点部分：

实现的接口方法**postProcessBeanDefinitionRegistry**：

```
300     @Override
301     public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) {
302       if (this.processPropertyPlaceHolders) {
303         processPropertyPlaceHolders();
304       }
305
306       ClassPathMapperScanner scanner = new ClassPathMapperScanner(registry);
307       scanner.setAddToConfig(this.addToConfig);
308       scanner.setAnnotationClass(this.annotationClass);
309       scanner.setMarkerInterface(this.markerInterface);
310       scanner.setSqlSessionFactory(this.sqlSessionFactory);
311       scanner.setSqlSessionTemplate(this.sqlSessionTemplate);
312       scanner.setSqlSessionFactoryBeanName(this.sqlSessionFactoryBeanName);
313       scanner.setSqlSessionTemplateBeanName(this.sqlSessionTemplateBeanName);
314       scanner.setResourceLoader(this.applicationContext);
315       scanner.setBeanNameGenerator(this.nameGenerator);
316       scanner.registerFilters();
317       scanner.scan(StringUtils.tokenizeToStringArray(this.basePackage, ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS));
318     }
```

代码分析：

line 302： 判断在当前的MapperScannerConfigurer中是否有属性配置符，就是说，是否含有类似**“${basePackage}”**这种字符串，通常，这种是会配置是属性文件中的，如果存在，则需要解析出来，并且复制给MapperScannerConfigurer对象中的相关属性；当然，**this.processPropertyPlaceHolders**也是从配置中配置的。

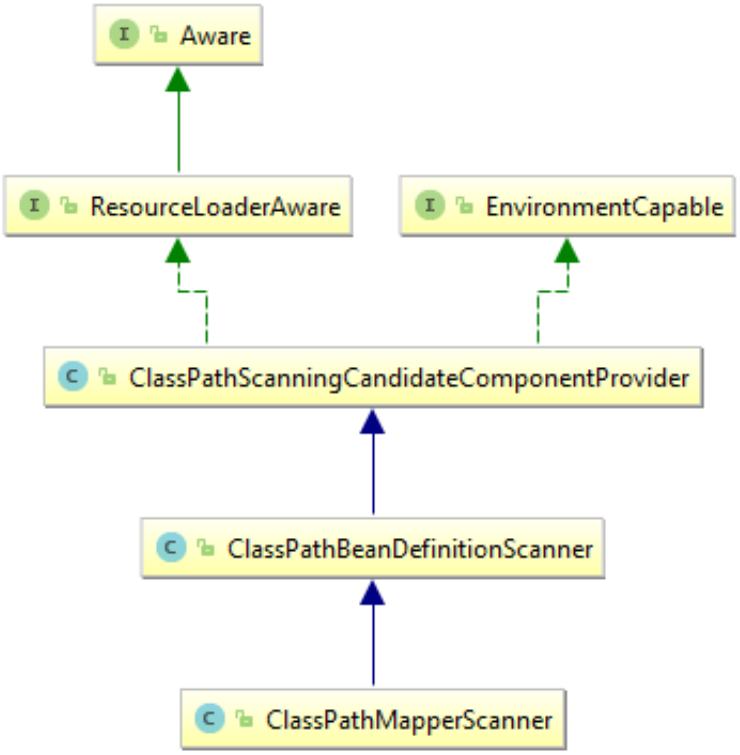
line 303： 如果需要解析属性文件中的相关通配符，则在此函数中进行操作，这里不是我们的重点，所以有兴趣的可以自己研究一下，很简单。

line 306 ~ line 315： 如代码所见，就是new出来一个扫描对象，用于扫描ClassPath下的Mapper接口，同时设置scanner的相关属性，这些属性，都可以从xml的配置文件中获得，如：basePackage，就是我们在xml中配置的唯一属性。这些值的在scanner里面的使用，我们待会就能看到。当然，如果你没配置，自然而然就是空。

line 316：这里是配置一些扫描时候的过滤器，默认情况下，会扫描basePackage下面的所有类，当然，也可以通过annotationClass/markerInterface来指定过滤规则，只扫描指定注解类型的接口，或者不扫描某种（markerInterface）接口；

line 317：核心点在这里，扫描basePackage下的所有类，并修改相关beanDefinition对象的有关属性，从而改变BeanFactory在实例化bean时的行为。

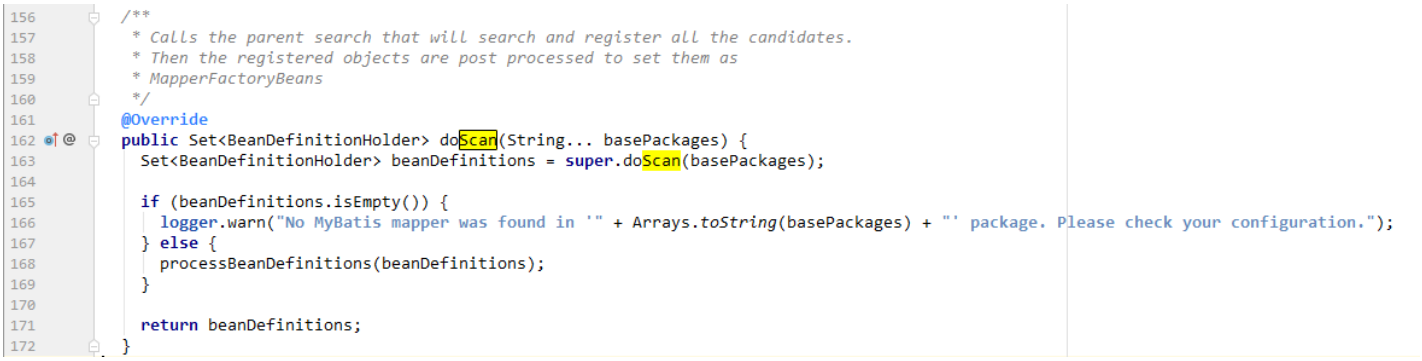
首先看下**ClassPathMapperScanner**的类图



scan的方法实现在其父类中ClassPathBeanDefinitionScanner实现

```
1.  public int scan(String... basePackages) {
2.           // 获取原有的已经加载好的bean
3.          int beanCountAtScanStart = this.registry.getBeanDefinitionCount();
4.           // 执行真正的扫描操作，在ClassPathMapperScanner中实现
5.          doScan(basePackages);
6.
7.          // 注册相关的注解处理器，用于处理可能在Mapper中存在的使用到的Spring注解，仔细查看
    里面的逻辑，可以发现，都是在没有相关注解的processor时才会注入
8.          // Register annotation config processors, if necessary.
9.          if (this.includeAnnotationConfig) {
10.             AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
11.         }
12.
13.         // 返回加载的bean数量
14.         return (this.registry.getBeanDefinitionCount() - beanCountAtScanStart);
15.     }
```

重点分析一下doScan方法，该方法被子类（ClassPathMapperScanner）覆盖。



```
156     /**
157      * Calls the parent search that will search and register all the candidates.
158      * Then the registered objects are post processed to set them as
159      * MapperFactoryBeans
160      */
161     @Override
162     public Set<BeanDefinitionHolder> doScan(String... basePackages) {
163       Set<BeanDefinitionHolder> beanDefinitions = super.doScan(basePackages);
164
165       if (beanDefinitions.isEmpty()) {
166         logger.warn("No MyBatis mapper was found in '" + Arrays.toString(basePackages) + "' package. Please check your configuration.");
167       } else {
168         processBeanDefinitions(beanDefinitions);
169       }
170
171       return beanDefinitions;
172     }
```

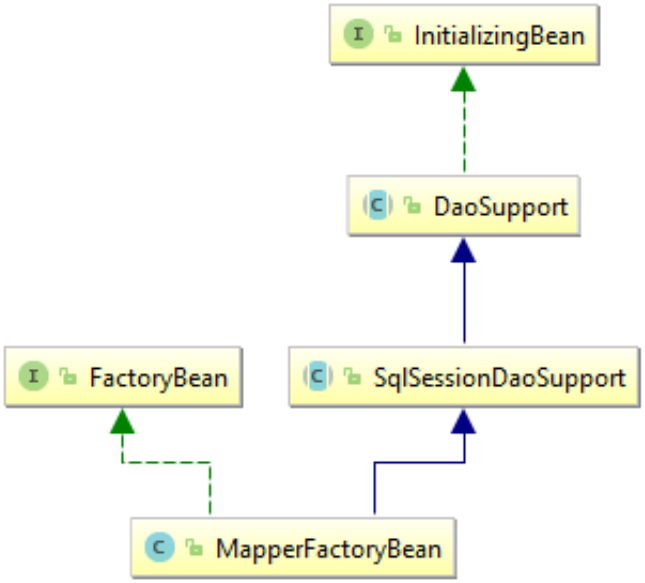
首先执行父类的扫描方法，获取当前目录下的Mapper接口，然后，如果扫描到的beanDefinitions不为空，则执行processBeanDefinitions对扫描进来的beanDefinitions做相关的处理

processBeanDefinitions源码：

```
1.  private void processBeanDefinitions(Set<BeanDefinitionHolder> beanDefinitions) {
2.      GenericBeanDefinition definition;
3.      // 遍历没一个beanDefinition对象
4.      for (BeanDefinitionHolder holder : beanDefinitions) {
5.        definition = (GenericBeanDefinition) holder.getBeanDefinition();
6.
7.        if (logger.isDebugEnabled()) {
8.          logger.debug("Creating MapperFactoryBean with name '" + holder.getBeanName()
9.              + "' and '" + definition.getBeanClassName() + "' mapperInterface");
10.       }
11.
12.       // 这里比较有趣，把当前bean定义的class对象作为了该bean class对象实例化的构造函数的入参
13.       // the mapper interface is the original class of the bean
14.       // but, the actual class of the bean is MapperFactoryBean
15.       definition.getConstructorArgumentValues().addGenericArgumentValue(definition.getBeanClassName()); // issue #59
16.       // 替换掉之前定义的bean的class为MapperFactoryBean.class
17.       definition.setBeanClass(this.mapperFactoryBean.getClass());
18.       // 设置MapperFactoryBean中的addToConfig属性为当前的对象中的addToConfig属性值，即为true
19.       definition.getPropertyValues().add("addToConfig", this.addToConfig);
20.
21.       // 接下来都是对xml配置里面的相关内容做赋值，如果都没配置，则会走到代码的49行
22.       boolean explicitFactoryUsed = false;
23.       if (StringUtils.hasText(this.sqlSessionFactoryBeanName)) {
24.         definition.getPropertyValues().add("sqlSessionFactory", new RuntimeBeanReference(this.sqlSessionFactoryBeanName));
25.         explicitFactoryUsed = true;
26.       } else if (this.sqlSessionFactory != null) {
27.         definition.getPropertyValues().add("sqlSessionFactory", this.sqlSessionFactory);
28.         explicitFactoryUsed = true;
29.       }
30.
```

```
31.       if (StringUtils.hasText(this.sqlSessionTemplateBeanName)) {
32.         if (explicitFactoryUsed) {
33.           logger.warn("Cannot use both: sqlSessionTemplate and sqlSessionFactory toge
    ther. sqlSessionFactory is ignored.");
34.         }
35.         definition.getPropertyValues().add("sqlSessionTemplate", new RuntimeBeanRefer
    ence(this.sqlSessionTemplateBeanName));
36.         explicitFactoryUsed = true;
37.       } else if (this.sqlSessionTemplate != null) {
38.         if (explicitFactoryUsed) {
39.           logger.warn("Cannot use both: sqlSessionTemplate and sqlSessionFactory toge
    ther. sqlSessionFactory is ignored.");
40.         }
41.         definition.getPropertyValues().add("sqlSessionTemplate", this.sqlSessionTempl
    ate);
42.         explicitFactoryUsed = true;
43.       }
44.
45.       if (!explicitFactoryUsed) {
46.         if (logger.isDebugEnabled()) {
47.           logger.debug("Enabling autowire by type for MapperFactoryBean with name '"
    + holder.getBeanName() + "'.");
48.         }
49.          //将当前的Bean模式设置为根据类型自动注入模式，默认为不自动注入。设置了根据类型自动
    注入的结果就是，当BeanFactory实例化该BeanDefinition的时候，对该
           //BeanDefinition的属性引用会进行自动化注入，注入的策略就是选择与引用类型同类型的b
    ean对象实例，这也是，为什么SqlSessionFactory实例对象实例化之后，会被自动注入到
50.          // MapperFactoryBean这个的类的对象的根本原因
51.         definition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_TYPE);
52.       }
53.     }
54.   }
```

然后我们看下MapperFactoryBean的相关源码，这真正产生Mapper动态代理对象的地方：



看，这个类图，是不是发现有点眼熟？SqlSessionFactoryBean是不是也是实现了接口FactoryBean？看到这里，是不是豁然开朗？对没错MapperFactoryBean也是通过getObject方法来返回真正的Mapper代理对象的。换句化说，当使用Spring的getBean方法时，首先会根据beanId找到这个的MapperFactoryBean对象，当发现这个是一个FactoryBean时，就会调用它的getObject方法来得到真正的对象（没错，就是Mybatis里面的Mapper代理对象）。现在是不是很激动的想看到MapperFactoryBean的getObject对象的实现了？

MapperFactoryBean.getObject源码：

```
91      * {@inheritDoc}
92      */
93     @Override
94     public T getObject() throws Exception {
95       return getSqlSession().getMapper(this.mapperInterface);
96     }
```

哈哈，没错，就是这样获取到的。这和开篇的那段代码是如此的一致。很多同学看到这里，一定还至少带着两个疑问：

a）getSqlSession这么实现的，又是何时何地设置的这个sqlSession？

b）this.mapperInterface又是个什么东西？

MapperFactoryBean类的成员变量

```
54  public class MapperFactoryBean<T> extends SqlSessionDaoSupport implements FactoryBean<T> {
55
56    private Class<T> mapperInterface;
57
58    private boolean addToConfig = true;
59
60    public MapperFactoryBean() {
61      //intentionally empty
62    }
63
64    public MapperFactoryBean(Class<T> mapperInterface) { this.mapperInterface = mapperInterface; }
67
```

看这里，this.mapperInterface，是从构造函数中传入的。结合上述的processBeanDefinitions方法中的15行代码，在scanner中已经对BeanDefinition中的入参类型做了定义。不难发现，该mapperInterface就是Mapper对应的class对象（如：CountryMapper.class），再结合getObject的源码，就能解释为什么能够通过它获取到Mybatis中的mapper代理对象。

接下来我们看下getSqlSession方法，该方法的实现在SqlSessionDaoSupport中

```
41  public abstract class SqlSessionDaoSupport extends DaoSupport {
42
43    private SqlSession sqlSession;
44
45    private boolean externalSqlSession;
46
47    public void setSqlSessionFactory(SqlSessionFactory sqlSessionFactory) {
48      if (!this.externalSqlSession) {
49        this.sqlSession = new SqlSessionTemplate(sqlSessionFactory);
50      }
51    }
52
53    public void setSqlSessionTemplate(SqlSessionTemplate sqlSessionTemplate) {
54      this.sqlSession = sqlSessionTemplate;
55      this.externalSqlSession = true;
56    }
57
58    /**
59     * Users should use this method to get a SqlSession to call its statement methods
60     * This is SqlSession is managed by spring. Users should not commit/rollback/close it
61     * because it will be automatically done.
62     *
63     * @return Spring managed thread safe SqlSession
64     */
65    public SqlSession getSqlSession() {
66      return this.sqlSession;
67    }
68
69    /**
70     * {@inheritDoc}
71     */
72    @Override
73    protected void checkDaoConfig() {
74      notNull(this.sqlSession, message: "Property 'sqlSessionFactory' or 'sqlSessionTemplate' are required");
75    }
76
77  }
```

实际上这里获取的是成员变量sqlSession，然而，这个变量的设置却是通过setSqlSessionFactory或setSqlSessionTemplate的setter方法设置的。还记得之前在设置beanDefinition的地方，将其属性设置为AbstractBeanDefinition.AUTOWIRE_BY_TYPE，这样在Spring的beanFactory实例化BeanDefinition（即MapperFactoryBean）时，会自动调用该setter方法，将事先已经存在spring容器中的sqlSessionFactory（由SqlSessionFactoryBean的getObject获取，前面已经细讲了）对象注入到此bean对象中，这样，在调用getSqlSession方法的时候，就能正确获取在配置文件中配置的正确对象信息，这样，Spring就成功的代理了Mybatis的mapper动态代理对象的管理。

```
48.         }
49.         //将当前的Bean模式设置为根据类型自动注入模式，默认为不自动注入。设置了根据类型自动注入的结果就是，当BeanFactory实例化该BeanDefinition的时候，对该
            //BeanDefinition的属性引用会进行自动化注入，注入的策略就是选择与引用类型同类型的bean对象实例，这也是，为什么SqlSessionFactory实例对象实例化之后，会被自动注入到
50.         // MapperFactoryBean这个的类的对象的根本原因
51.         definition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_TYPE);
52.       }
53.     }
54.   }
```

注意：在Spring的bean实例化的过程中，会对标准的get/set方法进行分析，为相关的BeanDefinit

ion对象添加**get/set**方法中存在但是在**Bean**的实际定义中没有的属性。这样的话，实例化时会调用相关的**get/set**方法，如以上截图中的**public void setSqlSessionFactory(SqlSessionFactory sqlSessionFactory)**和**public void setSqlSessionTemplate(SqlSessionTemplate sqlSessionTemplate)**，这两个方法都会在**MapperFactoryBean**实例化的时候被调用，并且成功赋值**(**具体可以参照**Spring**的**Introspector**的**getBeanInfo**方法的实现**)**。

四、**Spring**在获取相关的**Mapper**代理对象

以下是一张调用的时序图，较为详细的描述了，如何通过**Spring**容器去获取到**Mybatis**的**Mapper**代理对象，实际上是对**MapperFactoryBean**对象的获取，由于**MapperFactorrBean**对象是**FactoryBean**，故而返回的是其**getObject**方法的返回对象。前面已经分析的很清楚了。