Mybatis

1.1 搭建mybatis环境

1. 创建mybatis-config.xml
2. pom.xml加入依赖
3. Mapper.xml中写动态sql
4. 创建Mapper接口xxxMapper（以前叫Dao接口xxxDao）

1.2 SqlSession

SqlSession对象： 能够发送sql语句到数据库。SqlSession对象使 用完时，要进行close。

sqlSession.getMapper(Mapper接口.class): mybatis会创建一个代理类(实现了这个传入的Mapper接口)，然后创建这个代理类的对象并 返回。mybatis创建的这个代理类的作用是作为Mapper接口的实现类。

SqlSession是一个接口，需要传给接口中的方法的var1这个参数的就是Mapper.xml中的sql语句对应标签的id值

```
public interface SqlSession extends Closeable {
    <T> T selectOne(String var1);

    <T> T selectOne(String var1, Object var2);

    <E> List<E> selectList(String var1);

    <E> List<E> selectList(String var1, Object var2);

    <E> List<E> selectList(String var1, Object var2, RowBounds var3);

    <K, V> Map<K, V> selectMap(String var1, String var2);
```

Sqlsession这个接口的实现类只有一个就是DefaultsqlSession，DefaultsqlSession不是线程安全的。从sqlsessionfactory获得的sqlsession对象就是DefaultsqlSession对象。

在一个线程中只能有一个sqlsession对象，不然会出问题，所以要用threadLocal保证一个线程中取到的sqlsession对象一定是同一个。

DefaultsqlSession实现Sqlsession接口的时候将sqlsession接口中crud方法的第一个参数名var1都重新定义了一个新名字叫statement。

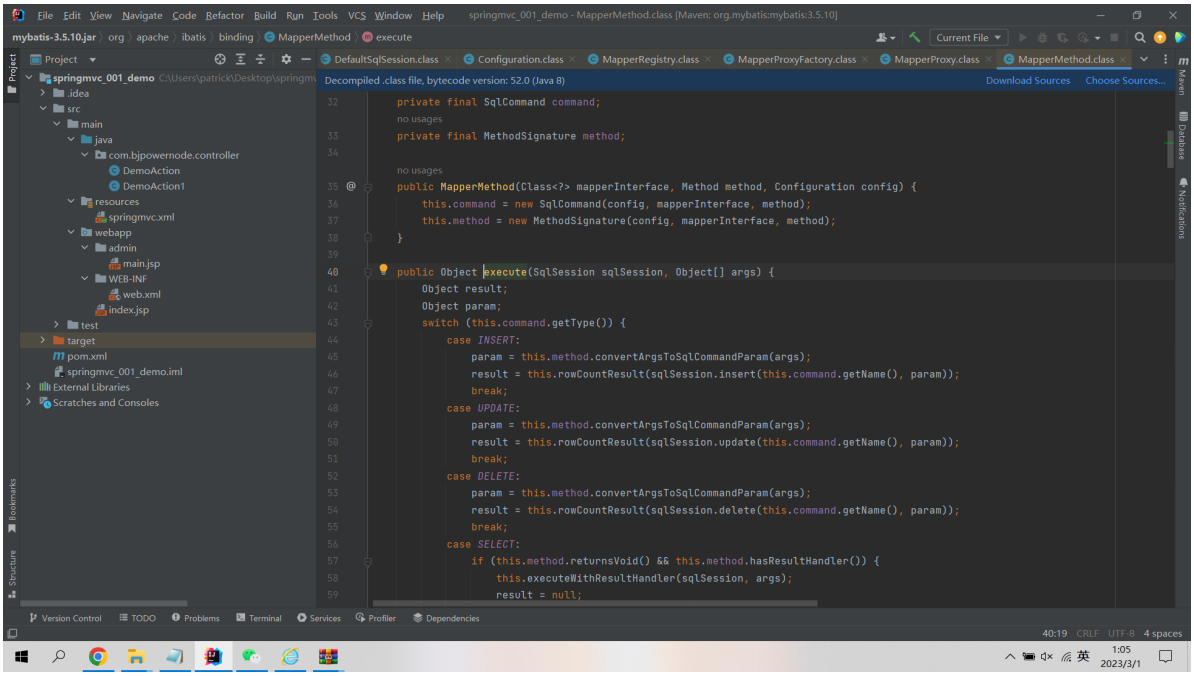
需要传给crud方法的statement参数的就是Mapper.xml中的sql语句对应标签的id值。

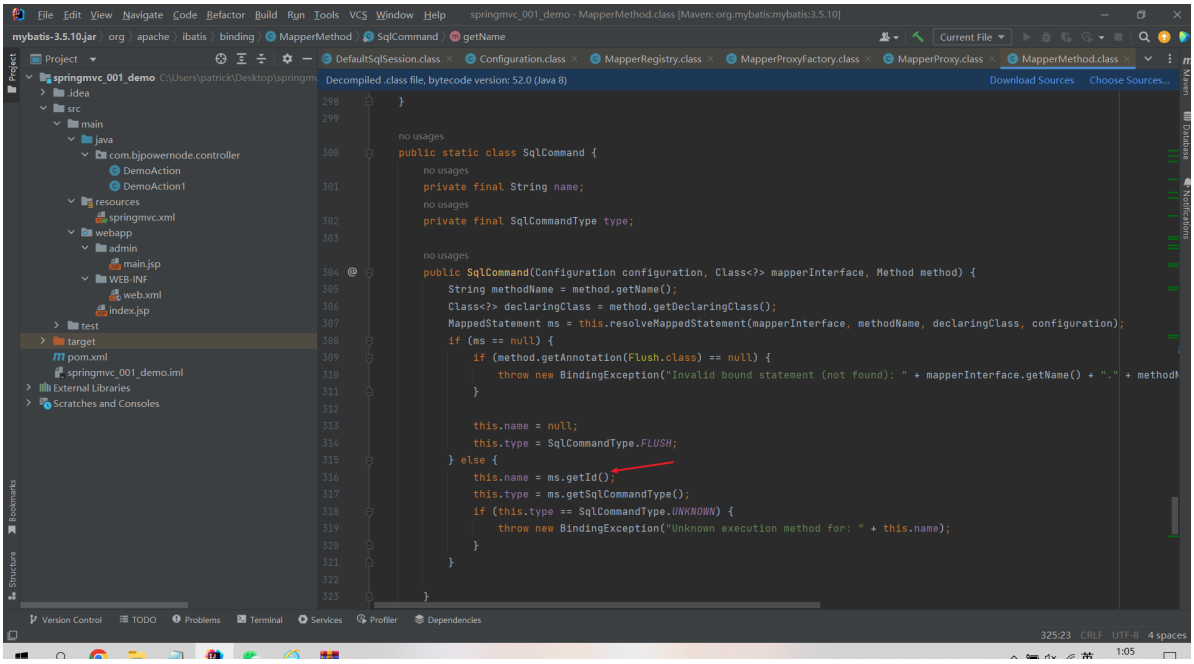
需要传给parameter参数的就是要传入动态sql的值。

```
public <T> T selectOne(String statement) {
        return this.selectOne(statement, (Object)null);
    }

    public <T> T selectOne(String statement, Object parameter) {
        List<T> list = this.selectList(statement, parameter);
        if (list.size() == 1) {
            return list.get(0);
```

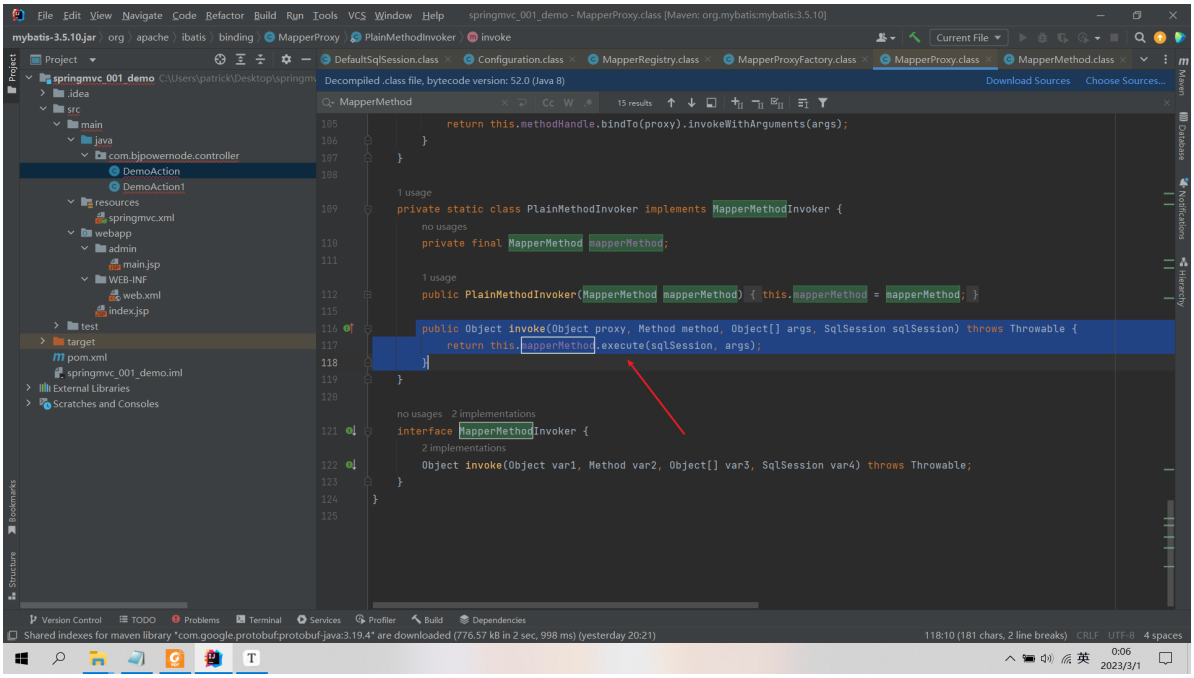
```
        } else if (list.size() > 1) {
            throw new TooManyResultsException("Expected one result (or null) to
be returned by selectOne(), but found: " + list.size());
        } else {
            return null;
        }
    }

    public <K, V> Map<K, V> selectMap(String statement, String mapKey) {
        return this.selectMap(statement, (Object)null, mapKey,
RowBounds.DEFAULT);
    }

    public <K, V> Map<K, V> selectMap(String statement, Object parameter, String
mapKey) {
        return this.selectMap(statement, parameter, mapKey, RowBounds.DEFAULT);
    }

    public <K, V> Map<K, V> selectMap(String statement, Object parameter, String
mapKey, RowBounds rowBounds) {
        List<? extends V> list = this.selectList(statement, parameter,
rowBounds);
        DefaultMapResultHandler<K, V> mapResultHandler = new
DefaultMapResultHandler(mapKey, this.configuration.getObjectFactory(),
this.configuration.getObjectWrapperFactory(),
this.configuration.getReflectorFactory());
        DefaultResultContext<V> context = new DefaultResultContext();
        Iterator var8 = list.iterator();
```

MapperMethod，会获取到mapperxml中sql标签的id值，和接口方法的方法名和参数。根据接口方法名判断返回类型？反正就是获得返回值类型，根据这个返回值类型，将id对应的结果集强制转化为返回类型？

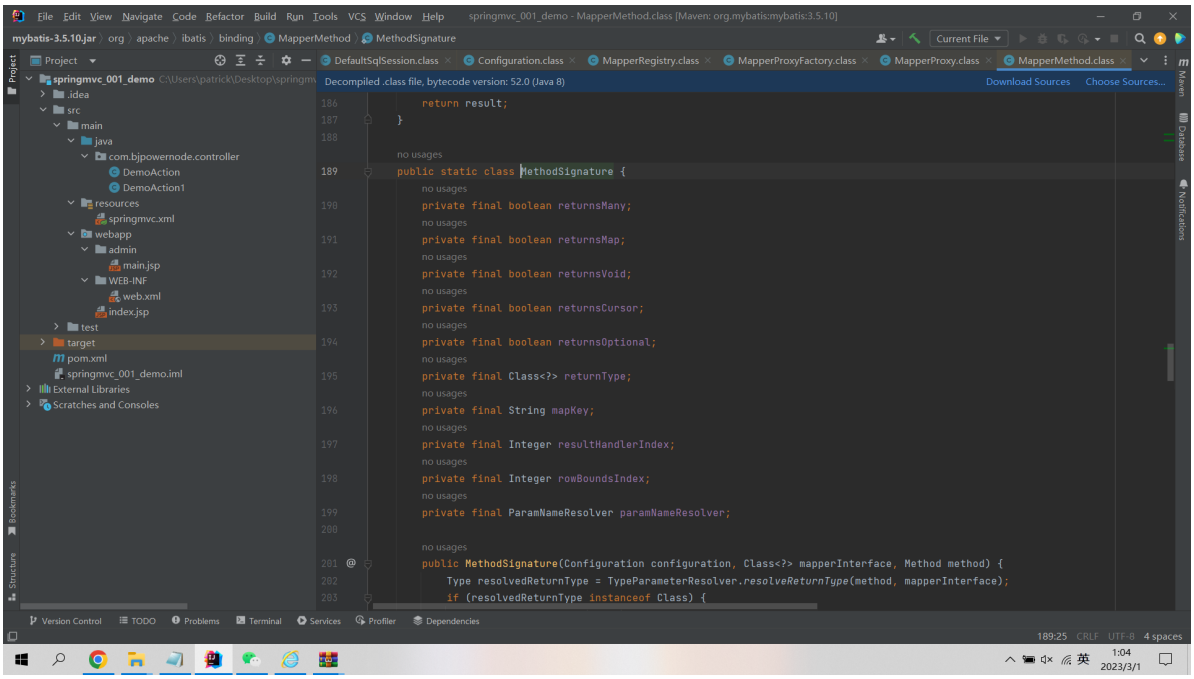




```
    }

    no usages
    public static class SqlCommand {
        no usages
        private final String name;
        no usages
        private final SqlCommandType type;

        no usages
        public SqlCommand(Configuration configuration, Class<?> mapperInterface, Method method) {
            String methodName = method.getName();
            Class<?> declaringClass = method.getDeclaringClass();
            MappedStatement ms = this.resolveMappedStatement(mapperInterface, methodName, declaringClass, configuration);
            if (ms == null) {
                if (method.getAnnotation(Flush.class) == null) {
                    throw new BindingException("Invalid bound statement (not found): " + mapperInterface.getName() + "." + methodN
                }

                this.name = null;
                this.type = SqlCommandType.FLUSH;
            } else {
                this.name = ms.getId();
                this.type = ms.getSqlCommandType();
                if (this.type == SqlCommandType.UNKNOWN) {
                    throw new BindingException("Unknown execution method for: " + this.name);
                }
            }

        }
```



```
            return this.methodHandle.bindTo(proxy).invokeWithArguments(args);
        }
    }

    1 usage
    private static class PlainMethodInvoker implements MapperMethodInvoker {
        no usages
        private final MapperMethod mapperMethod;

        1 usage
        public PlainMethodInvoker(MapperMethod mapperMethod) { this.mapperMethod = mapperMethod; }

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

    no usages  2 implementations
    interface MapperMethodInvoker {
        2 implementations
        Object invoke(Object var1, Method var2, Object[] var3, SqlSession var4) throws Throwable;
    }
}
```



```
        return result;
    }

    no usages
    public static class MethodSignature {
        no usages
        private final boolean returnsMany;
        no usages
        private final boolean returnsMap;
        no usages
        private final boolean returnsVoid;
        no usages
        private final boolean returnsCursor;
        no usages
        private final boolean returnsOptional;
        no usages
        private final Class<?> returnType;
        no usages
        private final String mapKey;
        no usages
        private final Integer resultHandlerIndex;
        no usages
        private final Integer rowBoundsIndex;
        no usages
        private final ParamNameResolver paramNameResolver;

        no usages
        public MethodSignature(Configuration configuration, Class<?> mapperInterface, Method method) {
            Type resolvedReturnType = TypeParameterResolver.resolveReturnType(method, mapperInterface);
            if (resolvedReturnType instanceof Class) {
```

1.3 Mybatis中的动态代理

Mybatis中定义了Mapper接口之后，不需要我们去实现Mapper接口，mybatis会自动生成Mapper接口的实现类（是个代理类）。

sqlSession.getMapper(Mapper接口.class): 执行了这段代码，mybatis才会生成代理类。

注意：我们用ssm的时候没有自己去创建Sqlsession对象，和调用getMapper()方法，而能在ServiceImpl类中自动注入实现了Mapper接口的代理类对象，这是因为ssm整合的时候，在配置文件中配置了相应的内容。框架读取到配置的内容之后，会产生SqlSession对象，Sqlsession对象会调用getMapper()方法，进而会产生实现了Mapper接口的代理类对象，并且把代理类对象放入spring容器中。

```
<!-- 整合MyBatis -->
<!--
    1. SqlSession对象的创建及管理

    MyBatis :  SqlSessionFactory(全局配置文件+映射文件)==>openSession()==>SqlSession
    SSM     :  SqlSessionFactoryBean
 -->
 <bean class="org.mybatis.spring.SqlSessionFactoryBean">
    <!-- 配置Mybatis的全局配置文件 -->
    <property name="configLocation" value="classpath:mybatis-config.xml"></property>
    <!-- 注入数据源 -->
    <property name="dataSource" ref="myDataSource"></property>
    <!-- 别名处理 -->
    <property name="typeAliasesPackage" value="com.atguigu.ssm.beans"></property>
    <!-- Sql映射文件 -->
    <property name="mapperLocations" value="classpath:mybatis/mapper/*.xml"></property>

 </bean>
```

```
    <!-- 2. Mapper接口代理实现类对象的创建及管理

        MyBatis : SqlSession.getMapper()==>xxxMapper的代理实现类对象
        SSM     : MapperScannerConfigurer
                  EmployeeMapper ==>代理实现类对象Proxy ==>在IOC容器中的id值: employeeMapper

    -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
       <property name="basePackage" value="com.atguigu.ssm.mapper"></property>
    </bean>
```