一、Spring

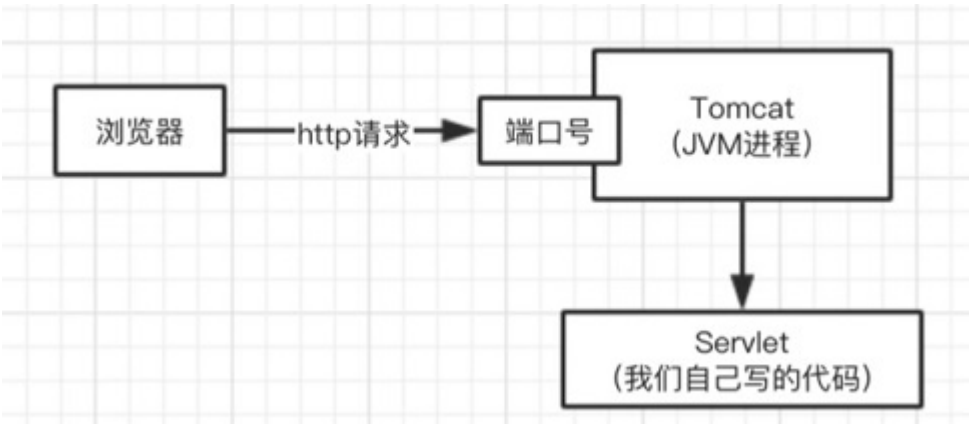
1、spring-ioc

核心：spring容器、反射

1.1 对spring-ioc的理解

传统servlet开发

```
@WebServlet("/hellohttp")
public class helloHttp extends HttpServlet {

    private static final long serialVersionUID = 1L;

    private MyService MyService = new MyServiceImpl();

    // res获取浏览器信息，ron向浏览器返回信息
    protected void doGet(HttpServletRequest res, HttpServletResponse ron) {
        MyService.handler();
    }

    protected void doPost(HttpServletRequest res, HttpServletResponse ron)  {
    }
}

public interface MyService{
  void handler();
}

public class MyServiceImpl implements MyService {
  void handler(){}
}

public class MySecondServiceImpl implements MyService {
  void handler(){}
}
```



> 问题：
>
> 一个类调用另外一个类，一般采用new的方式。
> 比如MyService = new MyServiceImpl()，而且可能在很多地方都得创建。如果MyService的实现类换成MySecondServiceImpl了，那么所有的地方都得改成MyService = new MySecondServiceImpl()。

所以：

传统的这种方式最大的缺点就是在于`类和类之间强耦合`。

哪怕一丁点功能改动，代码改动量都很大，而且容易出错。

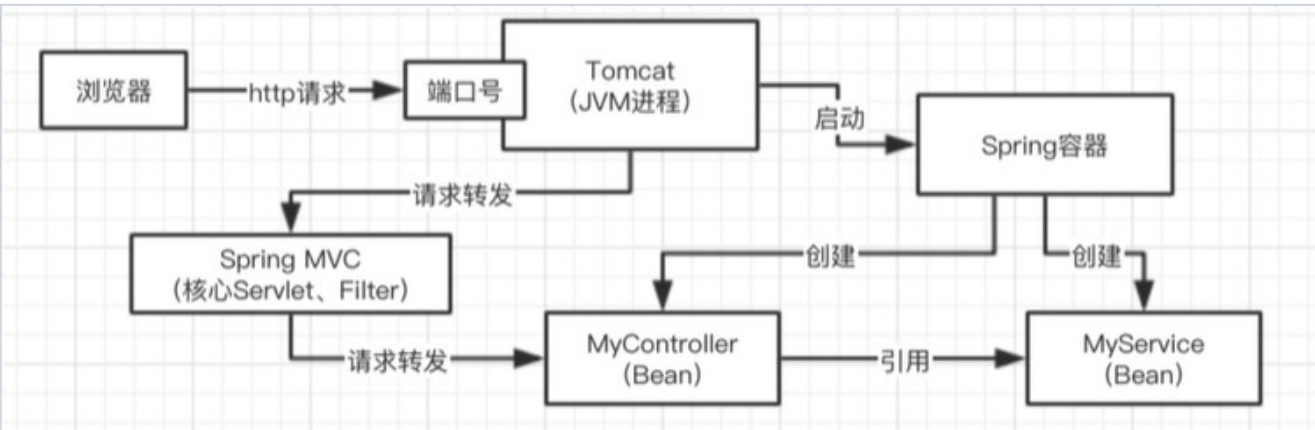
spring-ioc开发

spring容器：根据注解、xml配置，实例化一些bean对象

底层的核心技术就是反射。

> spring-ioc最大的贡献：系统的类和类之间彻底地解耦了。

```
@RestController
@RequestMapping("/hellohttp/")
public class MyController {

    @Autowired
    private MyService MyService;

    @Mapping("xxxx")
    protected Response doGet(Request res) {
        MyService.handler();
    }
}
```



2、spring-aop

核心：动态代理

```
比如spring的事务，在方法中都需要经历以下流程：
1. 开启事务
2. 多个sql语句
3.1 如果失败，回滚事务
3.2 如果成功，提交事务
```

如果直接编码，这样就需要在所有用到事务的方法中，都需要重复编写开启事务、回滚事务、提交事务的代码，很麻烦。

可以直接采用AOP，针对某些类下的所有方法中，开始、结束、抛出异常的时候都植入一些代码即可。

```java
@Component
@Scope("prototype") //创建为多例对象，防止多线程安全问题
public class AopAspectUtil {
    //注入spring的事务管理器
    @Autowired
    private DataSourceTransactionManager manager;

    //事务拦截器
    private TransactionStatus transaction;

    public TransactionStatus begin() {
        //设置为默认事务隔离级别
        transaction = manager.getTransaction(new DefaultTransactionAttribute());
        //返回事务拦截器
        return transaction;
    }

    public void commit() {
        manager.commit(transaction);
    }

    public void rollback() {
        manager.rollback(transaction);
    }
}
```

```
@Component
@Aspect
public class AopTransaction {
    @Autowired
    private AopAspectUtil transactionUtils;

    @Around("execution(* com.zbin.aop.service.UserService.add(..))")
    public void around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
        //调用方法之前执行
        System.out.println("开启事务");
        transactionUtils.begin();
        proceedingJoinPoint.proceed();
        //调用方法之后执行
        System.out.println("提交事务");
        transactionUtils.commit();
    }

    @AfterThrowing("execution(* com.zbin.aop.service.UserService.add(..))")
    public void afterThrowing() {
        System.out.println("异常通知  ");
        //获取当前事务进行回滚
        //TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
        transactionUtils.rollback();
    }
}
```

2.1 aop原理

采用spring-aop时，比如需要注入UserService

```
public class UserController{
  // 如果有切面，那么被注入的不再是UserServiceImpl实例。
  // 而是代理类ProxyUserService实例
  @Autowired
  private UserService userService;

  public void doService{
    userService.register();
  }
}
```

如果对UserService中的方法设置了切面，那么在注入的时候，就不是注入UserServiceImpl实例，而ProxyUserService实例。

```
public class ProxyUserService implements UserService {

  @Autowired
  private UserService userService; // 注入原来的实现类UserServiceImpl

  public void register{
    // 方法执行前
    userService.register();       // 执行UserServiceImpl的register方法
    // 方法执行后
  }
}
```

2.2 工程实践

MML实际应用：在controller层，都通过AOP进行日志的打印。

```
@Aspect
@Component
public class LogAspect {
    private final static Logger logger = LoggerFactory.getLogger(LogAspect.class);

    @Autowired
    private UserService userService;

    private static String requestLogWithUser = "[REQUEST]URL:%s, METHOD:%s, PARAMETER:%s,
USERID:%s, USERNAME:%s";
    private static String requestLogNoUser = "[REQUEST]URL:%s, METHOD:%s, PARAMETER:%s";
    private static String requestLogWithDebug = "[REQUEST]URL:%s, METHOD:%s, PARAMETER: to long
,view by [debug] level .";

    private static String responseLog = "[RESPONSE]URL:%s, DURATION:%s ms";

    @Around("execution(* com.wm.test.api.controller..*.*(..))")
    public Object doAroundAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{
        long begin = System.currentTimeMillis();
        Map<String,Object> userInfo = null;
        try {
            userInfo = userService.getUserInfo();
        } catch (Exception e) {
            e.printStackTrace();
        }
        HttpServletRequest request = ((ServletRequestAttributes)
RequestContextHolder.getRequestAttributes()).getRequest();
        try {
             String log=null;
             String args=JSON.toJSONString(proceedingJoinPoint.getArgs());
             if(userInfo!=null){
                  log=String.format(requestLogWithUser,
                                    request.getRequestURL(),
                                    request.getMethod(),
                                    args,
                                    userInfo.get("id").toString(),
                                    userInfo.get("name").toString());
             }else{
                  log=String.format(requestLogNoUser,
                                    request.getRequestURL(),
                                    request.getMethod(),
                                    args);
             }
             logger.info(log);
         } catch (Exception e) {
              e.printStackTrace();
         }

        Object obj = proceedingJoinPoint.proceed();
        Long delay=System.currentTimeMillis()-begin;   //耗时
        logger.info(String.format(responseLog,request.
                                  getRequestURL(),
                                  delay.toString()));
```

```
        return obj;
    }
}
```

3、动态代理

spring-aop在使用过程中，需要对设置了切面的类进行动态代理。但是有的类实现了某个接口，比如Service层，有的类没有实现接口，比如Controller层。会采用不同的动态代理方式。

通常采用的动态代理有两种形式

- jdk动态代理：被代理类必须实现了某个接口
- cglib动态代理：被代理类没有实现接口，比如对Controller层配置切面。代理类是当前类的一个子类

3.1 jdk动态代理

3.2 cglib动态代理

```
public class HelloService {
    public HelloService() {
        System.out.println("HelloService构造");
    }

    /**
     * 该方法不能被子类覆盖,Cglib是无法代理final修饰的方法的
     */
    final public String sayOthers(String name) {
        System.out.println("HelloService:sayOthers>>" + name);
        return null;
    }

    public void sayHello() {
        System.out.println("HelloService:sayHello");
    }
}
```

```
/**
 * 自定义MethodInterceptor
 */
public class MyMethodInterceptor implements MethodInterceptor {

    /**
     * sub：cglib生成的代理对象
     * method：被代理对象方法
     * objects：方法入参
     * methodProxy: 代理方法
     */
    @Override
    public Object intercept(Object sub,
                            Method method,
                            Object[] objects,
                            MethodProxy methodProxy) throws Throwable {
        System.out.println("======插入前置通知======");
        Object object = methodProxy.invokeSuper(sub, objects);
        System.out.println("======插入后者通知======");
        return object;
    }
}
```

```
public static void main(String[] args) {
    // 代理类class文件存入本地磁盘方便我们反编译查看源码
    System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY, "D:\\code");
    // 通过CGLIB动态代理获取代理对象的过程
    Enhancer enhancer = new Enhancer();
    // 设置enhancer对象的父类
    enhancer.setSuperclass(HelloService.class);
    // 设置enhancer的回调对象
    enhancer.setCallback(new MyMethodInterceptor());
    // 创建代理对象
    HelloService proxy= (HelloService)enhancer.create();
    // 通过代理对象调用目标方法
    proxy.sayHello();
}
```

将代理对象进行反编译：可以发现，生成的代理对象继承了HelloService，是它的子类

```
public class HelloService$$EnhancerByCGLIB$$4da4ebaf extends HelloService
    implements Factory
{
......
}
```

4、spring事务

使用@Transactional注解，spring就使用AOP的思想，在方法执行前开启事务，结束时提交事务，出异常时回滚事务。

4.1 事务的实现原理

4.2 事务的传播机制

包含事务的方法，调用了其他包含事务的方法。

```
// Propagation.REQUIRED 为默认的事务传播机制
@Transactional(propagation = Propagation.REQUIRED)
public void methodA(){
    doSomethingPre();
    methodB();
    doSomethingAfter();
}

@Transactional(propagation = Propagation.REQUIRED)
public void methodB(){
    // do something
}
```

事务共7种传播机制：

一般也就 `REQUIRED` 、 `REQUIRES_NEW` 、 `NESTED` 会用。

```
public enum Propagation {
    //【默认】:当前没有事务就创建事务，已有事务就加入该事务。
    REQUIRED(0),
    // 当前有事务就加入该事务，没有事务就以非事务的方式执行
    SUPPORTS(1),
    // 当前有事务就加入该事务，没有事务就报错。
    MANDATORY(2),
    // 当无论当前是否存在事务，都创建新事务。
    // 注意：此时methodB异常回滚，methodA还是能正常执行完的。回滚事务的时候互不影响了
    REQUIRES_NEW(3),
    // 强制以非事务的方式执行。如果当前存在事务，就把当前事务挂起。
    NOT_SUPPORTED(4),
    // 以非事务的方式执行。如果当前存在事务，就抛出异常。
    NEVER(5),
    // 嵌套事务。
    // 外层事务如果回滚，内层事务也会回滚；内层事务回滚仅仅回滚自己的代码
    NESTED(6);

    private final int value;

    private Propagation(int value) { this.value = value; }

    public int value() { return this.value;}
}
```

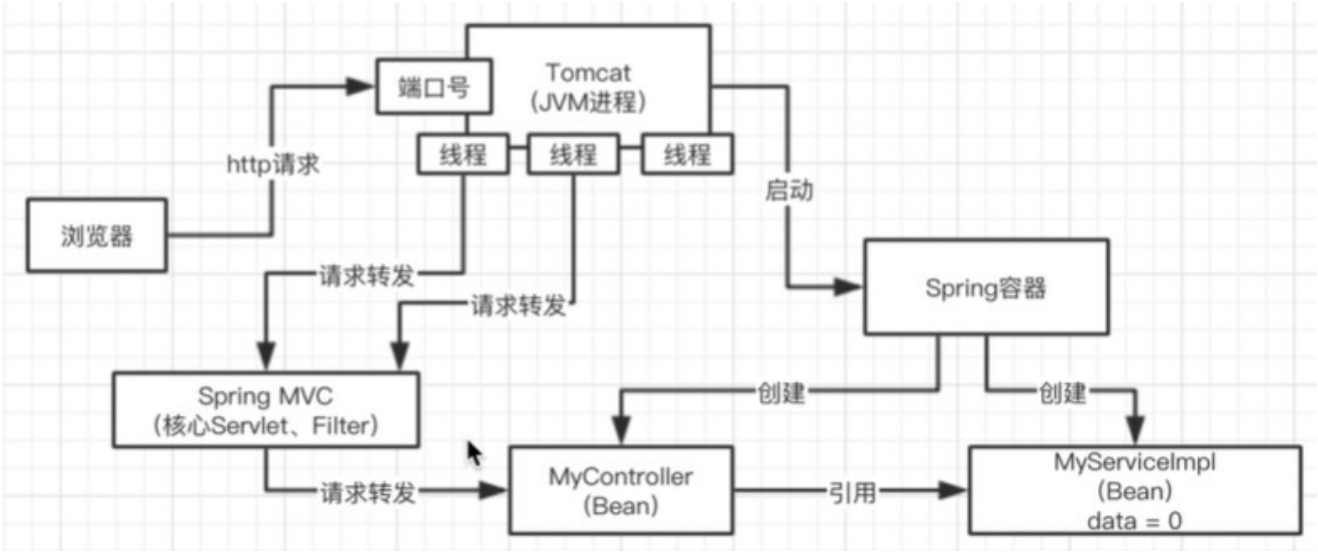
5、spring中bean是否线程安全

spring中bean的作用域：

- Singleton[默认]：每个容器中，只有一个bean实例
- prototype：为每次bean请求都提供一个实例
- request、session、global-session：都不常用

所以：像@Controller、@Service这些注解，在容器内都是只创建了一个实例对象。

> spring中bean是默认(Singleton)是线程不安全的。
>
> 比如在Service设置了一个实例变量，由于tomcat内部是多线程模型，同一时间并发访问的时候，可能是不同的线程请求Controller，然后访问同一个bean实例对象MyServiceImpl，此时进行data++之类的操作，肯定引发线程安全问题。
>
> 但是：一般来说，很少在spring bean里面放一下实例变量，而是通过多个组件互相调用，最终去访问数据库的，并不会多个线程去访问内存里面的一些共享变量。
>
> 因此：虽然spring bean是线程不安全的，但是由于多个线程并没有访问内存的共享变量，每次执行都是一个无状态的。此时并不会引发线程安全问题。



6、spring中的设计模式

6.1 工厂模式

spring ioc的核心就是工厂模式。将所有的spring bean放在spring容器中(一个大工厂)，里面包含了各种BeanDefination。

6.2 单例模式

spring bean的默认都是单例的。确保每个类只有一个实例对象。

6.3 代理模式、适配器模式

spring aop核心就是动态代理。

6.4 观察者模式

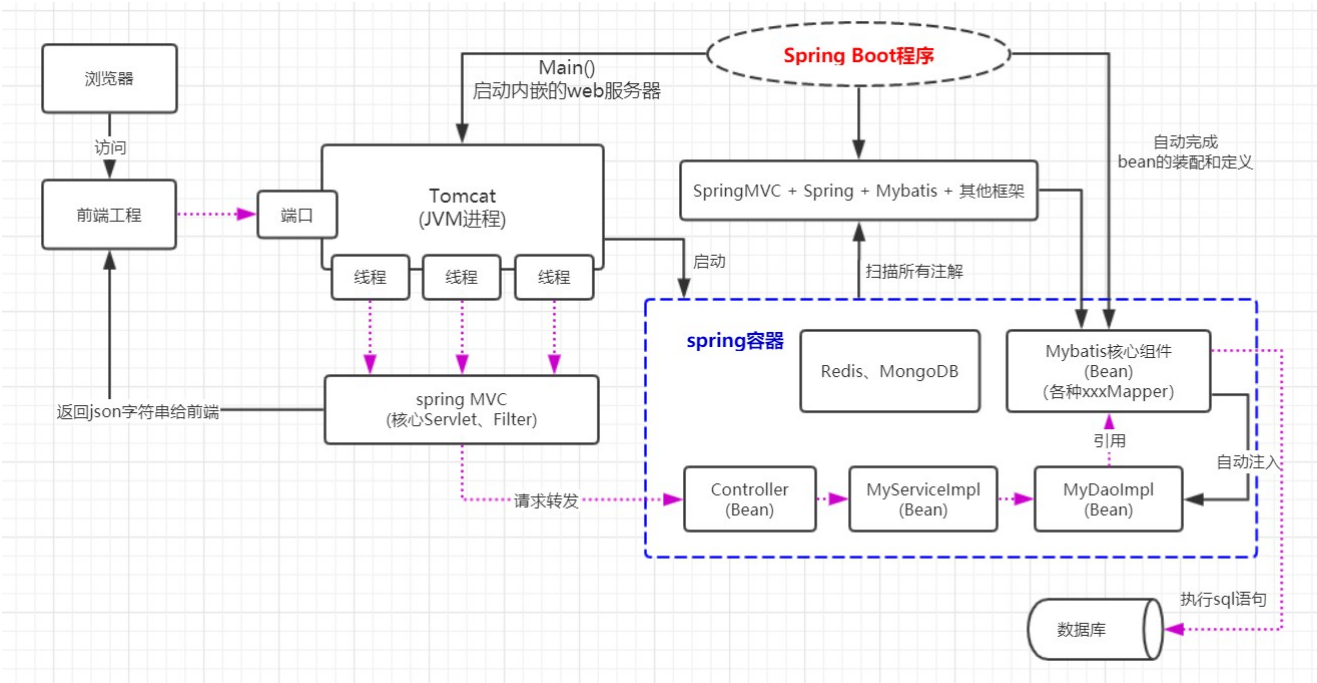
定义对象间的某种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。
spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

6.5 模板方法模式

二、Spring boot

注意以下几点：

1. tomcat内部会有多个线程，去处理client端的请求。
2. spring容器里面的对象默认是单例的。
3. spring boot支持封装Tomcat、Jetty和Undertow三种web服务器。默认采用tomcat



三、Spring MVC

四、Mybatis

1、传统jdbc写法

```java
public class JdbcTest {
    public static void main(String[] args) throws Exception {
        Connection con = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            //注册驱动。缺点：硬编码
            Class.forName("com.mysql.jdbc.Driver");
            String url = "jdbc:mysql://localhost:3306/mybatis";
            String user = "root";
            String password = "root";
            //获取连接 缺点：每次都要重新建立连接
            con = DriverManager.getConnection(url, user, password);
            String sql = "select * from tb_user where id = ? ";
            //获取数据库操作对象
            ps = con.prepareStatement(sql);
            //设置参数  缺点：需要提前知道参数类型
            ps.setLong(1, 1);
            rs = ps.executeQuery();
            //处理结果集  缺点：需要提前知道结果有哪些字段
            while(rs.next()){
                System.out.println(rs.getString("name"));
                System.out.println(rs.getInt("age"));
                System.out.println(rs.getInt("sex"));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }finally{
            //释放资源  缺点：每次使用完都必须释放资源
            try {
                if (rs!=null) {
                    rs.close();
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
            try {
                if (ps!=null) {
                    ps.close();
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
            try {
                if (con!=null) {
                    con.close();
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

2、mybatis入门小案例

mybatis-config.xml

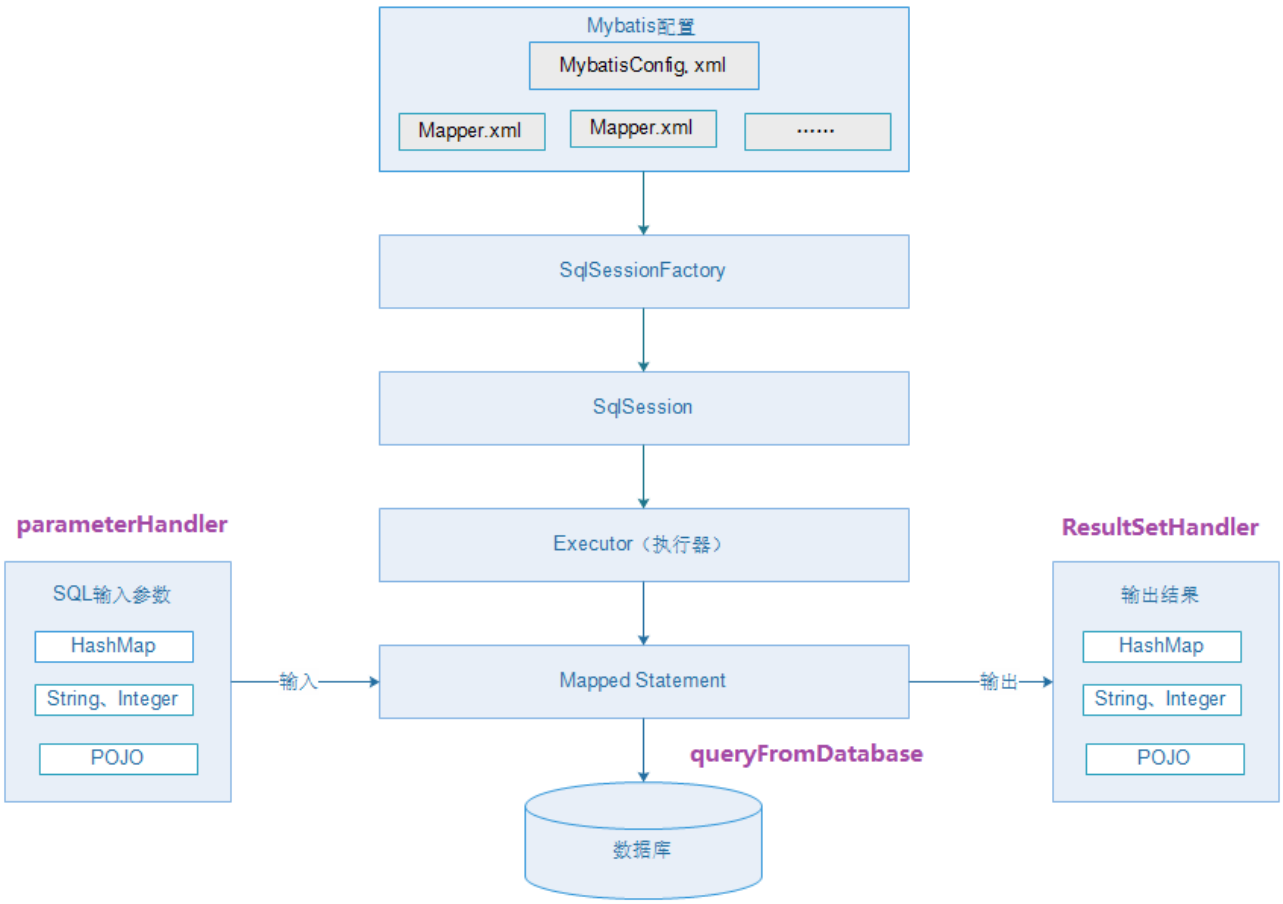
```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 配置环境：可以配置多个环境，default：配置某一个环境的唯一标识，表示默认使用哪个环境 -->
    <environments default="development">
        <!-- 配置环境,id:环境的唯一标识 -->
        <environment id="development">
            <!-- 事务管理器，type:使用jdbc的事务管理器 -->
            <transactionManager type="JDBC"/>
            <!-- 数据源，type:数据源类型，池类型的数据源 -->
            <dataSource type="POOLED">
                <!-- 配置连接信息 -->
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://172.20.3.173:3306/ds0"/>
                <property name="username" value="root"/>
                <property name="password" value="MySQL!23"/>
            </dataSource>
        </environment>
    </environments>
    <!-- 配置映射文件：用来配置sql语句和结果集类型等 -->
    <mappers>
        <mapper resource="UserMapper.xml"/>
    </mappers>
</configuration>
```

UserMapper.xml

```
<?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.wm.service.UserMapper.queryUserById">
    <!-- 开启mybatis的二级缓存 -->
    <cache/>
    <select id="queryUserById" resultType="com.wm.pojo.User">
        select *,user_name as userName from tb_user where id = #{id}
    </select>
</mapper>
```

```java
public class MybatisTest {
    public static void main(String[] args) throws IOException {
        //获取全局配置文件输入流
        InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");
        //加载全局配置文件
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
        //获取sqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();
        //第一个参数：名称空间.语句的唯一标识
        //第二个参数：sql语句传递的参数
        User user = sqlSession.selectOne("com.wm.service.UserMapper.queryUserById", 1L);
        System.out.println(user);

        // 会通过动态代理生成UserMapper的一个实现类
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        User user1 = mapper.queryUserById(1);
        System.out.println(user1);

        //释放资源
        sqlSession.close();
    }
}
```

结合上述过程，可以发现mybatis的执行流程为：



3、mybatis缓存

3.1 mybatis一级缓存

一级缓存默认是开启的！！！但是中间如果经历过增删改，也会清空一级缓存。

作用域：sqlSession期间。另外一个线程来就查询就利用不了一级缓存了

```
User user = sqlSession.selectOne("UserMapper.queryUserById", 1L);
System.out.println(user);

// 测试一级缓存
// sqlSession.clearCache()    清空一级缓存
User user1 = sqlSession.selectOne("UserMapper.queryUserById", 1L);
System.out.println(user1);

// 通过日志可以发现：只执行了一次查询。可以通过sqlSession.clearCache()方法清空一级缓存
```

3.2 mybatis二级缓存

作用域：

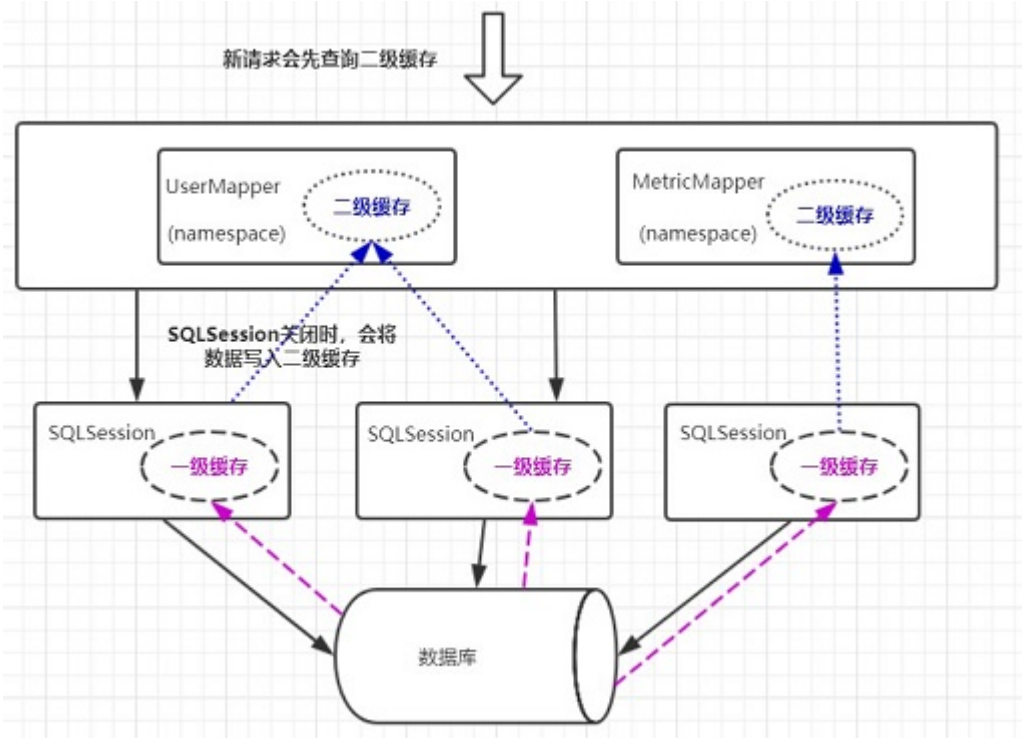
- 范围是按照每个namepace一个缓存来存贮和维护，同一个namespace放到一个缓存对象中，当这个namaspace中执行了`！isselect`语句的时候，整个namespace中的缓存全部清除掉。不同namespace互不影响。
- 跨sqlSession，不同的SqlSession可以从二级缓存中命中
- 几乎基于application为生命周期的

```
可以理解一个xxxMapper接口对应一个namespace，用接口的全路径名作为namespace的！！！
```

如何开启：

- 在映射文件中，添加标签
- 在全局配置文件中，设置cacheEnabled参数，默认已开启。

```
二级缓存只有在sqlSession被close()的时候才会将数据存入二级缓存中。
另外，使用二级缓存要求对象必须序列化

这样：第一个线程查询结束，close session后，数据存入二级缓存
      第二个线程来只要中间没有经历增删改操作，都可以利用二级缓存。
```



3.3 二级缓存注意事项

https://www.cnblogs.com/KingIceMou/p/9389872.html

缓存是以namespace为单位的，不同namespace下的操作互不影响。

insert、update、delete操作会清空所在namespace下的全部缓存。

> 第一：不只是要保证这个表在整个系统中只有单表操作，而且和该表有关的全部操作必须全部在一个namespace下。
>
> 针对一个表的某些操作不在他独立的namespace下进行。例如在UserMapper.xml中有大多数针对user表的操作。但是在一个XXXMapper.xml中，还有针对user单表的操作。
> 这会导致user在两个命名空间下的数据不一致。如果在UserMapper.xml中做了刷新缓存的操作，在XXXMapper.xml中缓存仍然有效，如果有针对user的单表查询，使用缓存的结果可能会不正确。
>
> 更危险的情况是在XXXMapper.xml做了insert,update,delete操作时，会导致UserMapper.xml中的各种操作充满未知和风险。
>
> 第二：多表操作一定不能使用缓存
> 为什么不能？首先不管多表操作写到那个namespace下，都会存在某个表不在这个namespace下的情况。
>
> 解决方案：选择一个Namespace作为主namespace，其余相关的Mapper使用cache-ref引用此Cache。

建议：采用专门的缓存

4、mybatis整合第三方缓存

mybatis本身的缓存就是一个map结构。它不是专门做缓存的，但是它给第三方缓存提供了SPI。

https://github.com/qiao-zhi/Maven_SSM

无论采用ehcache、redis都可以实现Cache接口。

这样开启mybatis二级缓存后，就可以直接将缓存信息写入相应的第三方缓存。

```
public interface Cache {
    // 获取缓存的唯一标识
    String getId();

    void putObject(Object key, Object value);

    Object getObject(Object key);

    Object removeObject(Object key);

    void clear();

    // 获取缓存系统存储的元素个数
    int getSize();

    ReadWriteLock getReadWriteLock();
}
```

5、mybatis拦截器

MyBatis 允许你在已映射语句执行过程中的某一点进行拦截调用。

默认情况下，MyBatis支持使用插件对以下方法(四大对象)的拦截。

- Executor：　　所有增加改查都是通过executor来执行的
- StatementHandler ：生成sql的statement
- ParameterHandler：对statement里的sql语句设置参数，进行参数预编译
- ResultSetHandler：　将statement的execute的结果，进行结果的封装

6、mybatis原理

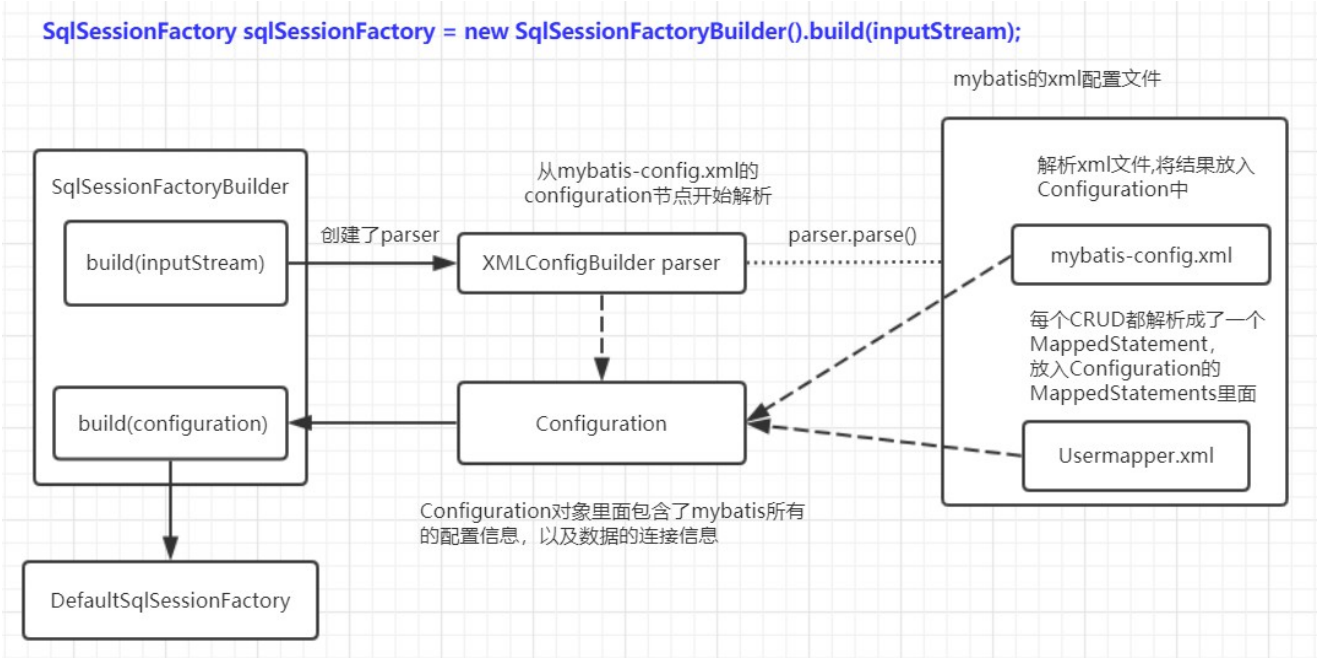
```
SqlSessionFactory sqlSessionFactory = new  SqlSessionFactoryBuilder().build(inputStream);

SqlSession sqlSession = sqlSessionFactory.openSession();

// 会通过动态代理生成UserMapper的一个实现类
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
```

6.1 SqlSessionFactory

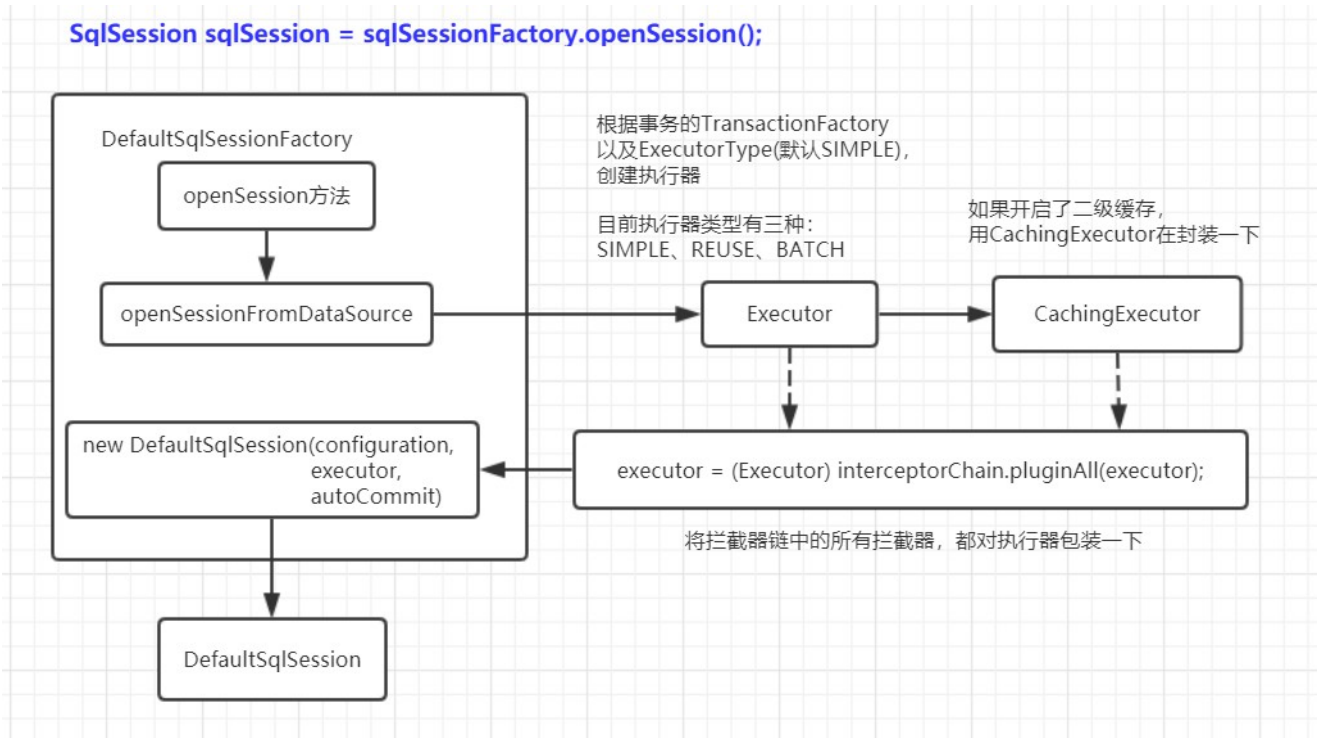
解析xml文件的所有信息，保存在Configuration对象中。

xml中的一个增删改查信息都被解析成一个MappedStatement。

所有增加改查都是通过executor来执行的。

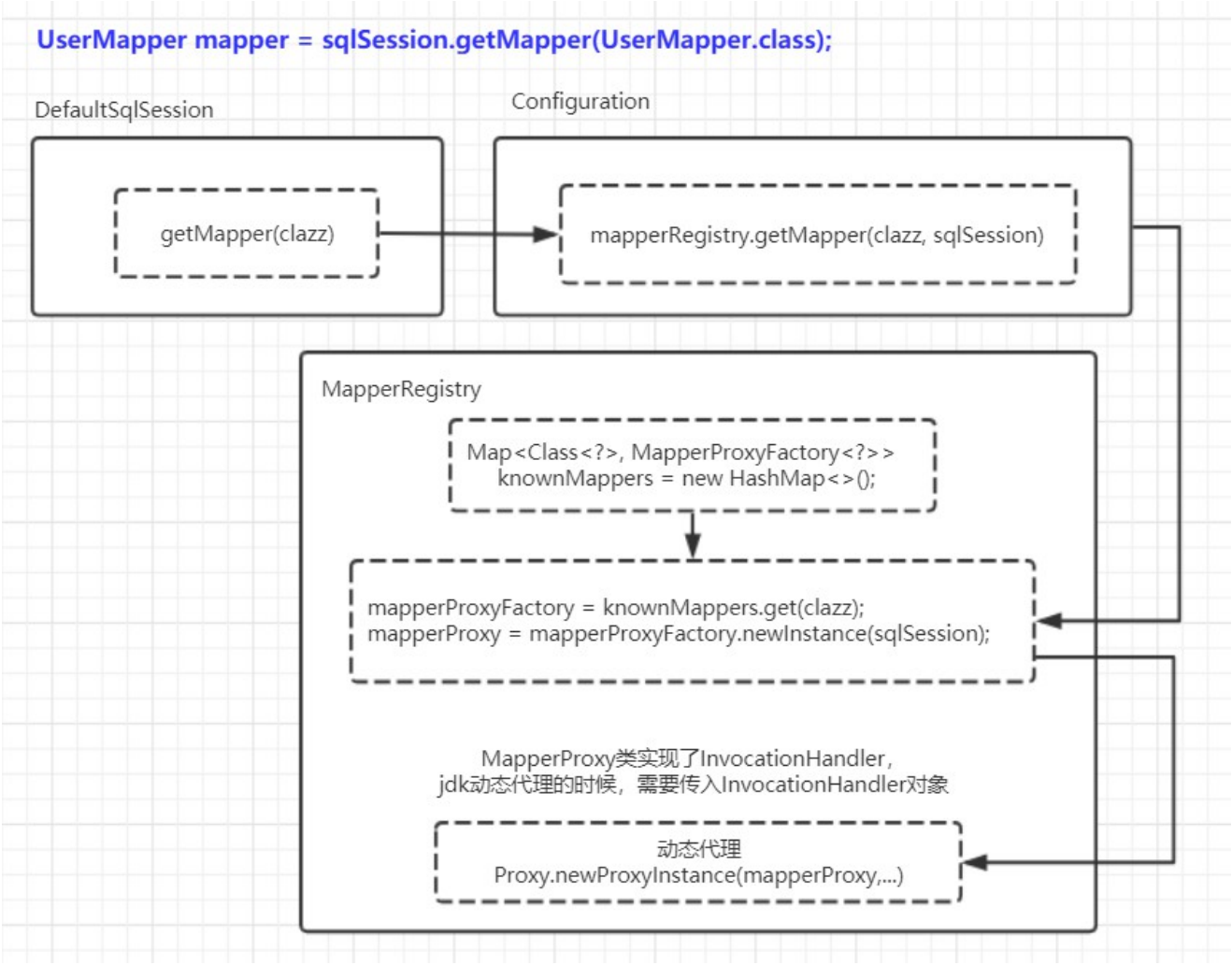


6.2 SqlSession

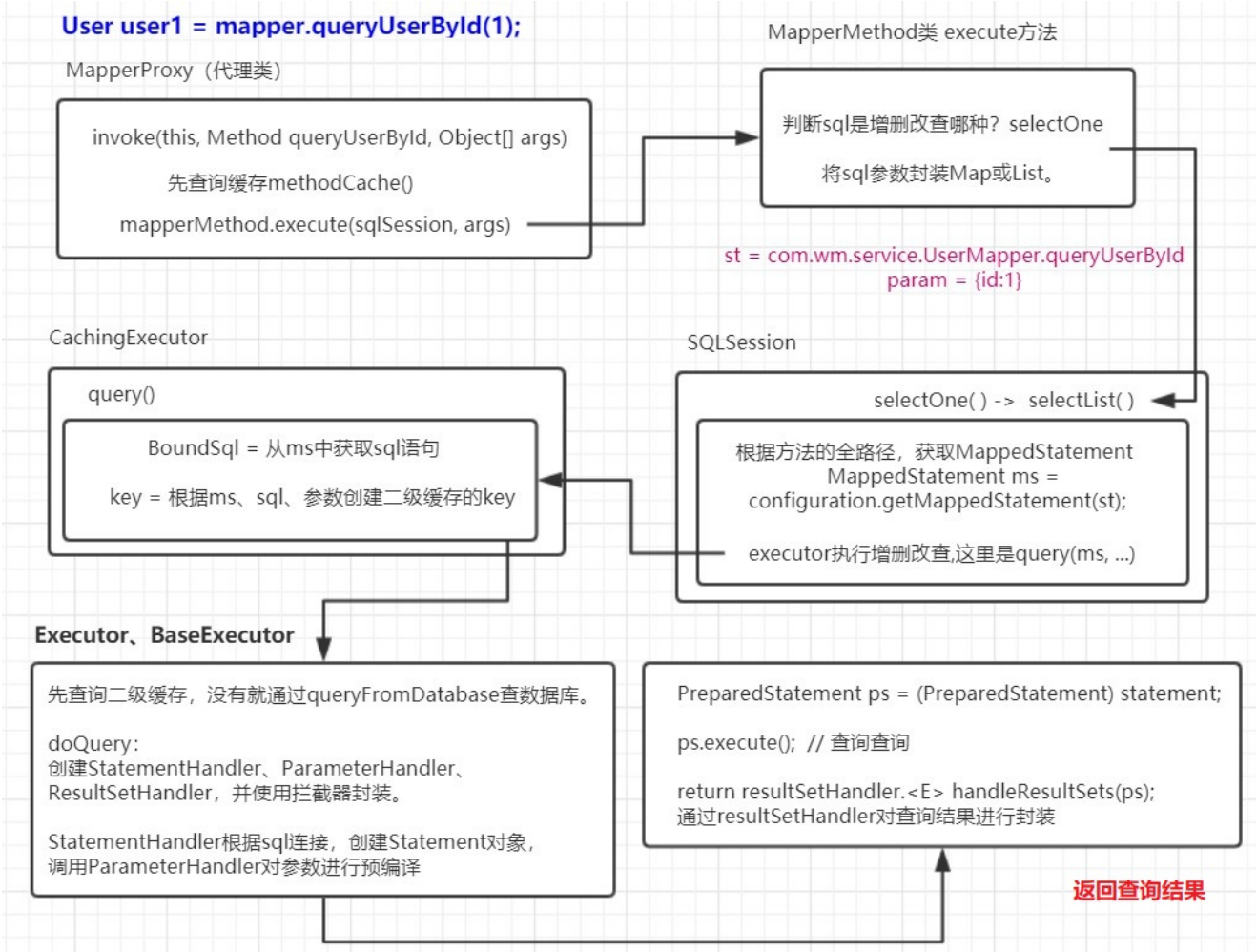


6.3 getMapper

根据自定义的xxxMapper的接口，生成接口的代理对象MapperProxy。

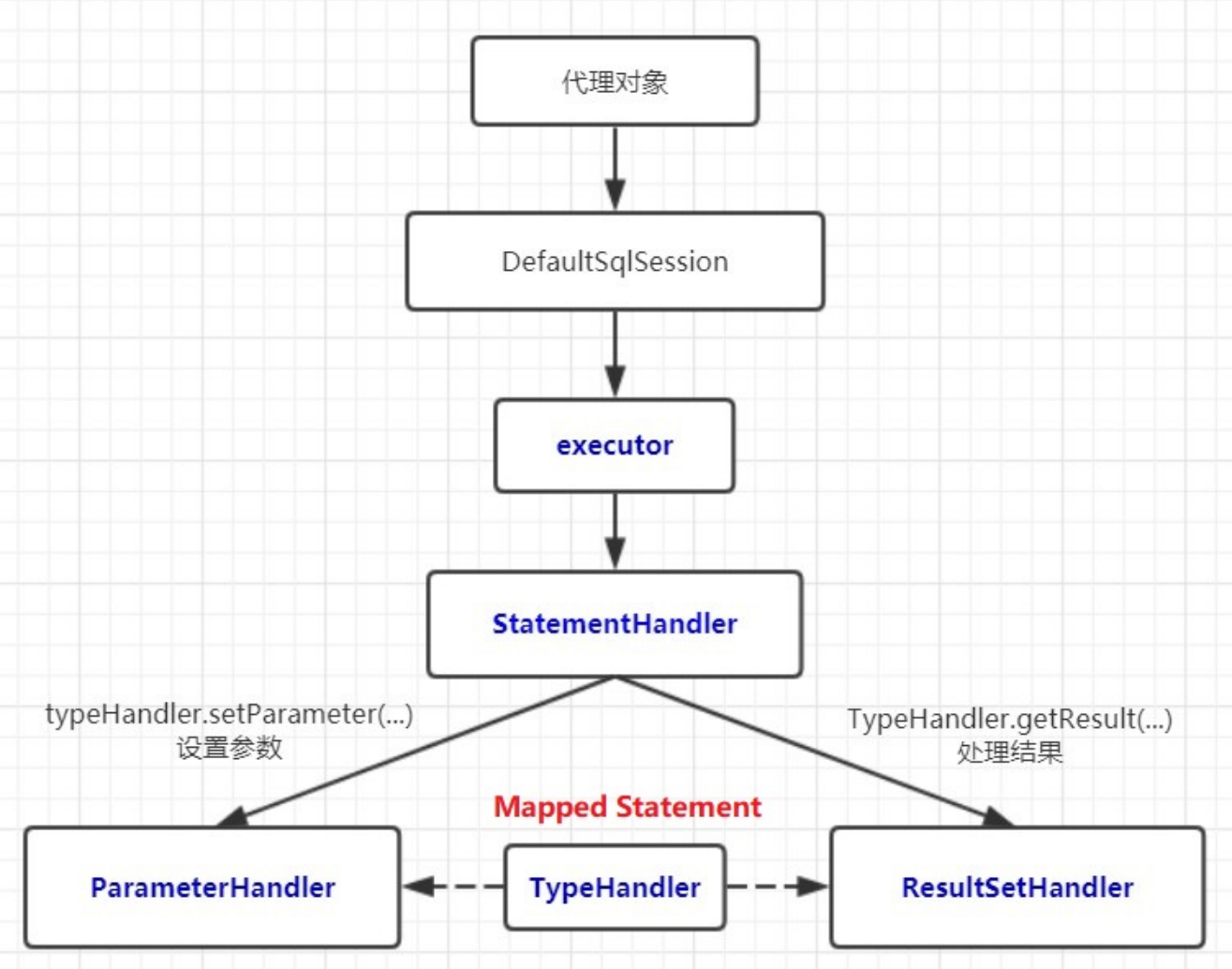


6.4 mybatis的query查询



最后做下总结：

1.根据配置文件初始化一个configuration对象
2.创建一个DefaultSQLSession对象，他里面包含了configuration、executor
3.DefaultSQLSession.getMapper(),拿到mapper接口的代理对象MapperProxy
4.实现CRUD方法
  4.1 调用DefaultSQLSession的executor
  4.2 创建一个StatementHandler，同时也会创建ParameterHandler、ResultSetHandler
  4.3 StatementHandler创建Statement对象,ParameterHandler对其进行参数设置以及参数预编译
  4.4 生成PreparedStatement，调用execute方法，查询数据库获取结果
  4.5 ResultSetHandler对结果进行封装成相应的对象并返回



7、mybatis插件原理

在四大对象创建的后：

1. 每个handler都调用了`interceptorChain.pluginAll(target)`
2. 获取所有的interceptor，执行interceptor.plugin(target)

```
public Object pluginAll(Object target) {
   for (Interceptor interceptor : interceptors) {
     target = interceptor.plugin(target);
   }
   return target;
}
```

3. 插件机制。为4大对象都创建出代理对象，这样代理对象就可以拦截对四大对象的每一个执行。

8、mybatis插件编写

```
1.编程Interceptor的实现类,继承Interceptor类
2.使用@Intercepts注解，完成插件的签名Signature
3.将写好的插件注册到全局的配置文件中（mybatis-config.xml）
```

```java
/**
 * 插件注解Intercepts的签名Signature参数:
 * type :  拦截哪个对象(Executor、StatementHandler、ParameterHandler、ResultSetHandler)
 * method: 拦截哪个方法
 * args:   拦截的方法参数类型。(有些方法是重载过的)
 */

@Intercepts({
     @Signature(type = StatementHandler.class,
                method = "parameterize",
                args = Statement.class)
})
public class MyPlugin implements Interceptor {
     /**
     * 在插件注册时，将插件配置的属性信息设置进来
     * @param properties
     */
    @Override
    public void setProperties(Properties properties) {
        System.out.println("==============》 插件的配置信息为: " + properties);
    }

    /**
     * 拦截模板对象目标方法的执行。
     * 只拦截注解签名中指定的type和method
     */
    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        System.out.println("======》 直接执行原方法: " + invocation.getMethod());
        // 执行模板方法。
        Object proceed = invocation.proceed();
        // 直接返回执行后的返回值
        return proceed;
    }

    /**
     * plugin: 包装目标对象，为目标对象创建一个代理对象
     * 四大对象在创建的时候，都会执行Interceptor的plugin方法
     * target = interceptor.plugin(target);
     */
    @Override
    public Object plugin(Object target) {
        // 使用当前的拦截器来包装目标对象
        Object proxy = Plugin.wrap(target, this);
        System.out.println("===== 执行自定义插件的 plugin方法  ==== " + target);
        // 返回为目标对象创建的动态代理对象
        return proxy;
    }
}
```

mybatis-config.xml全局配置

```
<plugins>
  <!-- 设置自定义插件-->
    <plugin interceptor="com.wm.plugin.MyPlugin">
        <!-- 配置插件的属性-->
        <property name="username" value="wangming"/>
        <property name="age" value="28"/>
    </plugin>
</plugins>
```

9、执行器executor

Mybatis 共有三种执行器:

- SIMPLE: 默认的执行器, 对每条sql进行预编译->设置参数->执行等操作
- BATCH: 批量执行器, 对相同sql进行一次预编译, 然后设置参数, 最后统一执行操作。如何采用SIMPLE会每次都会预编译
- REUSE: REUSE 执行器会重用预处理语句。(prepared statements)

```
SqlSession simpleSqlSession = sqlSessionFactory.openSession();

SqlSession  reuseSqlSession = sqlSessionFactory.openSession(ExecutorType.REUSE);

SqlSession  batchSqlSession = sqlSessionFactory.openSession(ExecutorType.BATCH);
```