1. MyBatis 源码概述

1.1 怎么下载 MyBatis 源码？

MyBatis 源码下载地址：https://github.com/MyBatis/MyBatis-3
建议直接用网盘里的源码包，老师有在里面加注释；

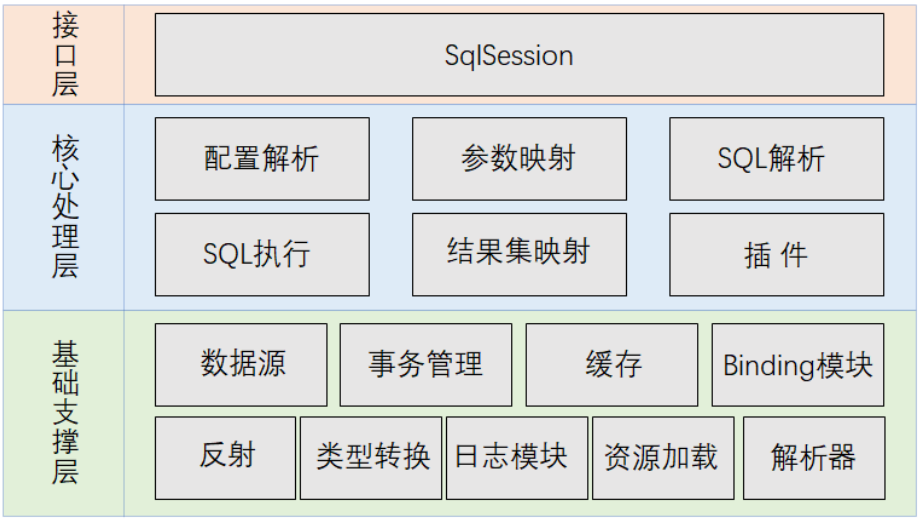
源码包导入过程：

1. 下载 MyBatis 的源码
2. 检查 maven 的版本，必须是 3.25 以上，建议使用 maven 的最新版本
3. MyBatis 的工程是 maven 工程，在开发工具中导入，工程必须使用 jdk1.8 以上版本；
4. 把 MyBatis 源码的 pom 文件中<optional>true</optional>，全部改为 false；
5. 在工程目录下执行 mvn clean install -Dmaven.test.skip=true,将当前工程安装到本地仓库（pdf 插件报错的话，需要将这个插件屏蔽）；
   注意：安装过程中会可能会有很多异常信息，只要不中断运行，请耐心等待；
6. 其他工程依赖此工程

1.2 源码架构分析

源码包模块分析见脑图（双击打开）：

MyBatis 源码共 16 个模块，可以分成三层，如下图：



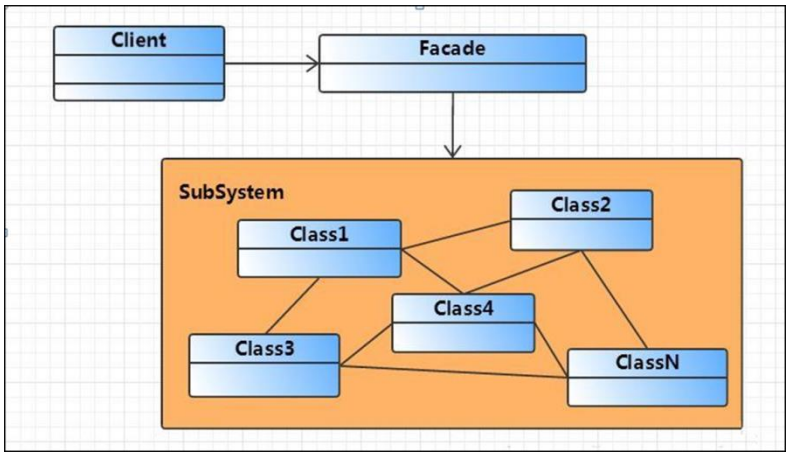
基础支撑层：技术组件专注于底层技术实现，通用性较强无业务含义；
核心处理层：业务组件专注 MyBatis 的业务流程实现，依赖于基础支撑层；
接口层：MyBatis 对外提供的访问接口，面向 SqlSession 编程；

思考题：系统为什么要分层？

1. 代码和系统的可维护性更高。系统分层之后，每个层次都有自己的定位，每个层次内部的组件都有自己的分工，系统就会变得很清晰，维护起来非常明确；
2. 方便开发团队分工和开发效率的提升；举个例子，mybatis 这么大的一个源码框架不可能是一个人开发的，他需要一个团队，团队之间肯定有分工，既然有了层次的划分，分工也会变得容易，开发人员可以专注于某一层的某一个模块的实现，专注力提升了，开发效率自然也会提升；
3. 提高系统的伸缩性和性能。系统分层之后，我们只要把层次之间的调用接口明确了，那我们就可以从逻辑上的分层变成物理上的分层。当系统并发量吞吐量上来了，怎么办？为了提高系统伸缩性和性能，我们可以把不同的层部署在不同服务器集群上，不同的组件放在不同的机器上，用多台机器去抗压力，这就提高了系统的性能。压力大的时候扩展节点加机器，压力小的时候，压缩节点减机器，系统的伸缩性就是这么来的；

1.3 外观模式（门面模式）

从源码的架构分析，特别是接口层的设计，可以看出来 MyBatis 的整体架构符合门面模式的。门面模式定义：提供了一个统一的接口，用来访问子系统中的一群接口。外观模式定义了一个高层接口，让子系统更容易使用。类图如下：



Facade 角色：提供一个外观接口，对外，它提供一个易于客户端访问的接口，对内，它可以访问子系统中的所有功能。

SubSystem（子系统）角色：子系统在整个系统中可以是一个或多个模块，每个模块都有若干类组成，这些类可能相互之间有着比较复杂的关系。

门面模式优点：使复杂子系统的接口变的简单可用，减少了客户端对子系统的依赖，达到了解耦的效果；遵循了 OO 原则中的迪米特法则，对内封装具体细节，对外只暴露必要的接口。

门面模式使用场景：

✓ 一个复杂的模块或子系统提供一个供外界访问的接口

✓ 子系统相对独立 — 外界对子系统的访问只要黑箱操作即可

1.4 面向对象设计需要遵循的六大设计原则

学习源码的目的除了学习编程的技巧、经验之外，最重要的是学习源码的设计的思想以及设计模式的灵活应用，因此在学习源码之前有必要对面向对象设计的几个原则先深入的去了解，让自己具备良好的设计思想和理念；

1. **单一职责原则**：一个类或者一个接口只负责唯一项职责，尽量设计出功能单一的接口；

2. **依赖倒转原则**：高层模块不应该依赖低层模块具体实现，解耦高层与低层。既面向接口编程，当实现发生变化时，只需提供新的实现类，不需要修改高层模块代码;
3. **开放-封闭原则**：程序对外扩展开放，对修改关闭；换句话说，当需求发生变化时，我们可以通过添加新模块来满足新需求，而不是通过修改原来的实现代码来满足新需求；
4. **迪米特法则**：一个对象应该对其他对象保持最少的了解，尽量降低类与类之间的耦合度；实现这个原则，要注意两个点，一方面在做类结构设计的时候尽量降低成员的访问权限，能用 private 的尽量用 private；另外在类之间，如果没有必要直接调用，就不要有依赖关系；这个法则强调的还是类之间的松耦合；
5. **里氏代换原则**：所有引用基类（父类）的地方必须能透明地使用其子类的对象；
6. **接口隔离原则**：客户端不应该依赖它不需要的接口，一个类对另一个类的依赖应该建立在最小的接口上；

扩展知识：Lison 老师 2019 年 8 月 6 号的公开课《这样 Code 迅速脱单》，其中讲到的代码优化技巧归根究底就是在遵循单一职责原则和迪米特法则；

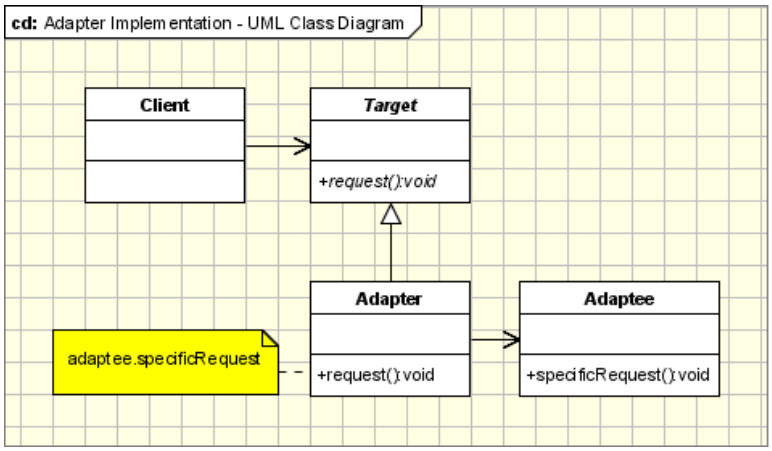
2. 日志模块分析

2.1 日志模块需求分析

1. MyBatis 没有提供日志的实现类，需要接入第三方的日志组件，但第三方日志组件都有各自的 Log 级别，且各不相同，而 MyBatis 统一提供了 trace、debug、warn、error 四个级别；
2. 自动扫描日志实现，并且第三方日志插件加载优先级如下：slf4J → commonsLoging → Log4J2 → Log4J → JdkLog;
3. 日志的使用要优雅的嵌入到主体功能中；

2.2 适配器模式

日志模块的第一个需求是一个典型的使用适配器模式的场景，**适配器模式**（Adapter Pattern）是作为两个不兼容的接口之间的桥梁，将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作；类图如下：



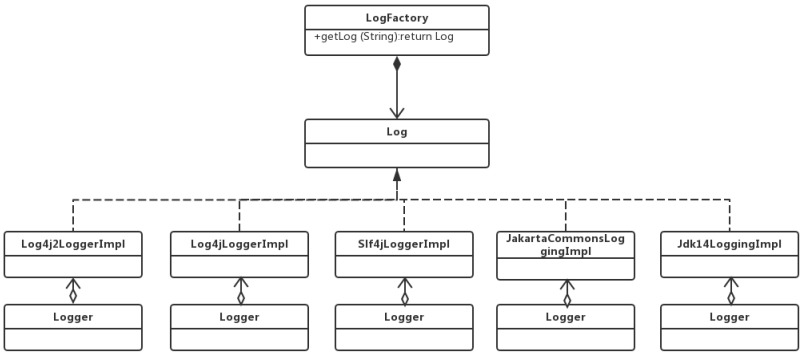
- ✓ Target：目标角色,期待得到的接口.
- ✓ Adaptee：适配者角色,被适配的接口.
- ✓ Adapter：适配器角色,将源接口转换成目标接口.

适用场景：当调用双方都不太容易修改的时候，为了复用现有组件可以使用适配器模式；在系统中接入第三方组件的时候经常被使用到；注意：如果系统中存在过多的适配器，会增加系统的复杂性，设计人员应考虑对系统进行重构；

MyBatis 日志模块是怎么使用适配器模式？实现如下：

- ✓ Target：目标角色,期待得到的接口。org.apache.ibatis.logging.Log 接口，对内提供了统一的日志接口；
- ✓ Adaptee：适配者角色,被适配的接口。其他日志组件组件如 slf4J 、commonsLoging 、Log4J2 等被包含在适配器中。
- ✓ Adapter：适配器角色,将源接口转换成目标接口。针对每个日志组件都提供了适配器，每个适配器都对特定的日志组件进行封装和转换；如 Slf4jLoggerImpl 、JakartaCommonsLoggingImpl 等；

日志模块适配器结构类图：



总结：日志模块实现采用适配器模式，日志组件（Target）、适配器以及统一接口（Log 接口）定义清晰明确符合单一职责原则；同时，客户端在使用日志时，面向 Log 接口编程，不需要关心底层日志模块的实现，符合依赖倒转原则；最为重要的是，如果需要加入其他第三方日志框架，只需要扩展新的模块满足新需求，而不需要修改原有代码，这又符合了开闭原则；

2.3 怎么实现优先加载日志组件？

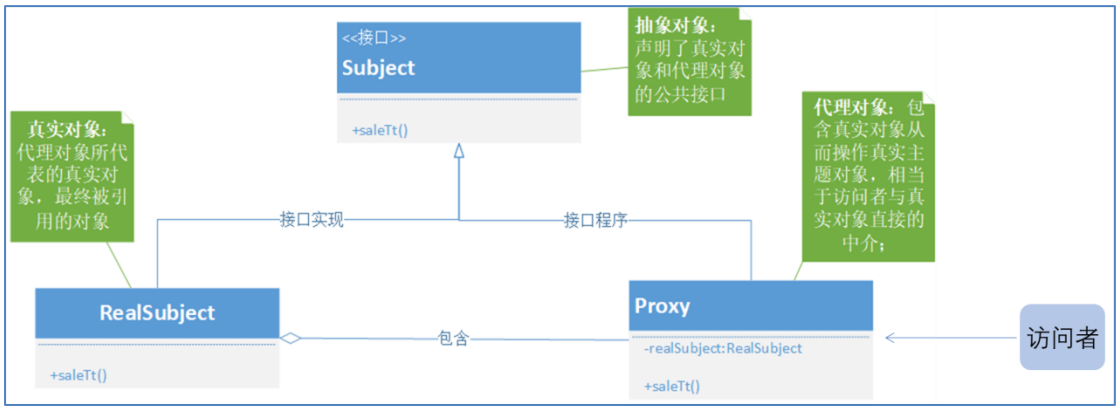
见 org.apache.ibatis.logging.LogFactory 中的静态代码块，通过静态代码块确保第三方日志插件加载优先级如下：slf4J → commonsLoging → Log4J2 → Log4J → JdkLog;

```java
public final class LogFactory {

  /**
   * Marker to be used by logging implementations that support markers
   */
  public static final String MARKER = "MYBATIS";

  //被选定的第三方日志组件适配器的构造方法
  private static Constructor<? extends Log> logConstructor;

  //自动扫描日志实现，并且第三方日志插件加载优先级如下：slf4J → commonsLoging → Log4J2 → Log4J → JdkLog
  static {
    tryImplementation(LogFactory::useSlf4jLogging);
    tryImplementation(LogFactory::useCommonsLogging);
    tryImplementation(LogFactory::useLog4J2Logging);
    tryImplementation(LogFactory::useLog4JLogging);
    tryImplementation(LogFactory::useJdkLogging);
    tryImplementation(LogFactory::useNoLogging);
  }

  private LogFactory() {
    // disable construction
  }
```

2.3 代理模式和动态代理

代理模式定义：给目标对象提供一个代理对象，并由代理对象控制对目标对象的引用；目的：（1）通过引入代理对象的方式来间接访问目标对象，防止直接访问目标对象给系统带来的不必要复杂性；（2）通过代理对象对原有的业务增强；

代理模式类图：



代理模式有静态代理和动态代理两种实现方式。

2.3.1 静态代理

这种代理方式需要代理对象和目标对象实现一样的接口。

优点：可以在不修改目标对象的前提下扩展目标对象的功能。

缺点：

- ✓ 冗余。由于代理对象要实现与目标对象一致的接口，会产生过多的代理类。
- ✓ 不易维护。一旦接口增加方法，目标对象与代理对象都要进行修改。

2.3.2 动态代理

动态代理利用了 JDK API，动态地在内存中构建代理对象，从而实现对目标对象的代理功能。

动态代理又被称为 JDK 代理或接口代理。静态代理与动态代理的区别主要在：

1. 静态代理在编译时就已经实现，编译完成后代理类是一个实际的 class 文件
2. 动态代理是在运行时动态生成的，即编译完成后没有实际的 class 文件，而是在运行时动态生成类字节码，并加载到 JVM 中

注意：动态代理对象不需要实现接口，但是要求目标对象必须实现接口，否则不能使用动态代理。

JDK 中生成代理对象主要涉及两个类，第一个类为 java.lang.reflect.Proxy，通过静态方法 newProxyInstance 生成代理对象，第二个为 java.lang.reflect.InvocationHandler 接口，通过 invoke 方法对业务进行增强；

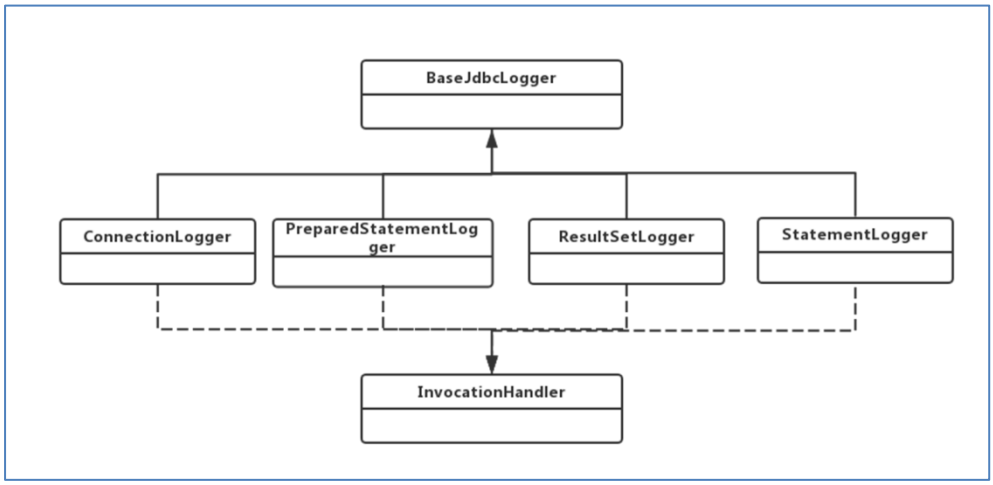
温馨提示：找班主任老师索取 lison 老师 2019 年 4 月 15 号的公开课《从动态代理来，到 Spring 源码去》，详细学习代理模式、静态代理、动态代理，以及这些技术点在 spring 事务中的运用。

2.3 优雅的增强日志功能

首先搞清楚那些地方需要打印日志？通过对日志的观察，如下几个位置需要打日志：

1. 在创建 prepareStatement 时，打印执行的 SQL 语句；
2. 访问数据库时，打印参数的类型和值
3. 查询出结构后，打印结果数据条数

因此在日志模块中有 BaseJdbcLogger、ConnectionLogger、PreparedStatementLogger 和 ResultSetLogge 通过动态代理负责在不同的位置打印日志；几个相关类的类图如下：



- ✓ BaseJdbcLogger：所有日志增强的抽象基类，用于记录 JDBC 那些方法需要增强，保存运行期间 sql 参数信息；
- ✓ ConnectionLogger：负责打印连接信息和 SQL 语句。通过动态代理，对 connection 进行增强，如果是调用 prepareStatement、prepareCall、createStatement 的方法，打印要执行的 sql 语句并返回 prepareStatement 的代理对象（PreparedStatementLogger），让 prepareStatement 也具备日志能力，打印参数；
- ✓ PreparedStatementLogger：对 prepareStatement 对象增强，增强的点如下：
  - ➢ 增强 PreparedStatement 的 setxxx 方法将参数设置到 columnMap、columnNames、columnValues，为打印参数做好准备；

- 增强 PreparedStatement 的 execute 相关方法，当方法执行时，通过动态代理打印参数,返回动态代理能力的 resultSet；
- 如果是查询，增强 PreparedStatement 的 getResultSet 方法，返回动态代理能力的 resultSet；如果是更新，直接打印影响的行数

✓ ResultSetLogge：负责打印数据结果信息；

最后一个问题：上面讲这么多，都是日志功能的实现，那日志功能是怎么加入主体功能的？答：既然在 Mybatis 中 Executor 才是访问数据库的组件，日志功能是在 Executor 中被嵌入的，具体代码在 org.apache.ibatis.executor.SimpleExecutor.prepareStatement(StatementHandler, Log) 方法中，如下图所示：

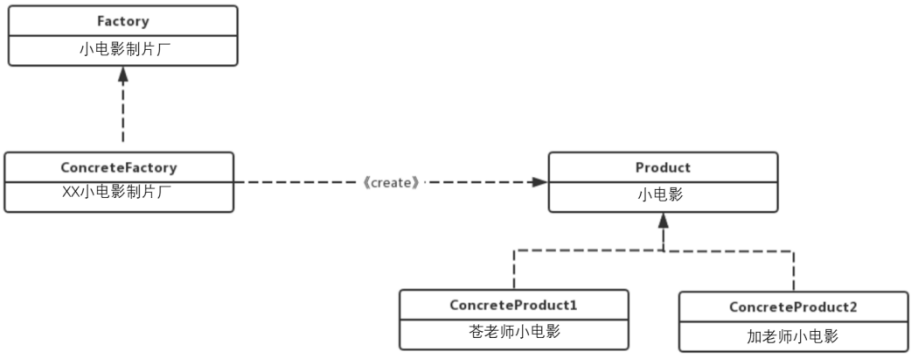
```
//创建Statement
private Statement prepareStatement(StatementHandler handler, Log statementLog) throws SQLException {
  Statement stmt;
  //获取connection对象的动态代理，添加日志能力
  Connection connection = getConnection(statementLog);
  //通过不同的StatementHandler，利用connection创建（prepare）Statement
  stmt = handler.prepare(connection, transaction.getTimeout());
  //使用parameterHandler处理占位符
  handler.parameterize(stmt);
  return stmt;
}
```

3. 数据源模块分析

数据源模块重点讲解数据源的创建和数据库连接池的源码分析；数据源创建比较负责，对于复杂对象的创建，可以考虑使用工厂模式来优化，接下来介绍下简单工厂模式和工厂模式；

3.1 简单工厂模式

简单工厂属于类的创建型设计模式，通过专门定义一个类来负责创建其它类的实例，被创建的实例通常都具有共同的父类。类图如下：



✓ 工厂接口（Factory）:简单工厂的接口，定义了创建产品的方法，具体的工厂类必须实现这个接口；

✓ 工厂角色（ConcreteFactory）：这是简单工厂模式的核心，由它负责创建全部的类的内部逻辑。工厂类被外界调用，创建所须要的产品对象。

- ✓ 抽象（Product）产品角色：简单工厂模式所创建的全部对象的父类，注意，这里的父类能够是接口也能够是抽象类，它负责描写叙述全部实例所共同拥有的公共接口。
- ✓ 详细产品（Concrete Product）角色：简单工厂所创建的详细实例对象，这些详细的产品往往都拥有共同的父类。

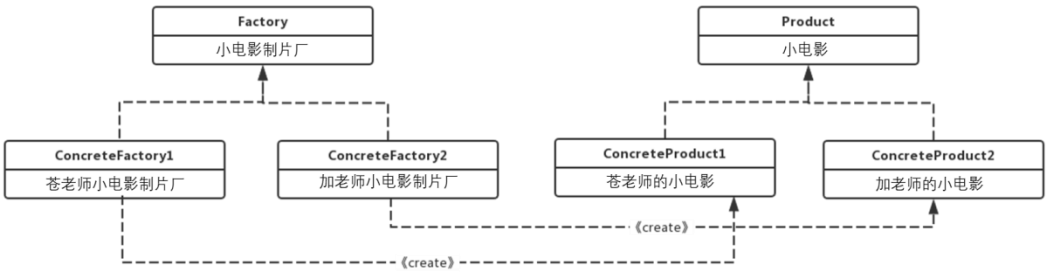
简单工厂适用场景：简单工厂模式将对象的创建和使用进行解耦，并屏蔽了创建对象可能的复杂过程，但由于创建对象的逻辑集中工厂类当中，所以简单工厂适合于产品类型不多、需求变化不频繁的场景；

简单工厂模式的缺点：工厂类负责了所有产品的实例化，违反单一职责原则，如果产品类型比较多工厂类的代码量会比较大，不利于类的可读性和扩展性；。另外当有新的产品类型加入时，必须修改工厂类原有的代码，这又违反了开闭原则；

示例代码：com.enjoylearning.mybatis.factory.simple.SimpleSmallMovieFactory

3.2 工厂模式

工厂模式属于创建型模式，它提供了一种创建对象的最佳方式。定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。类图如下：



- ✓ 产品接口（Product）：产品接口用于定义产品类的功能，具体工厂类产生的所有产品都必须实现这个接口。调用者与产品接口直接交互，这是调用者最关心的接口；
- ✓ 具体产品类（ConcreteProduct）：实现产品接口的实现类，具体产品类中定义了具体的业务逻辑；
- ✓ 工厂接口（Factory）：工厂接口是工厂方法模式的核心接口，调用者会直接和工厂接口交互用于获取具体的产品实现类；
- ✓ 具体工厂类（ConcreteFactory）:是工厂接口的实现类，用于实例化产品对象，不同的具体工厂类会根据需求实例化不同的产品实现类；

为什么要使用工厂模式？

答：对象可以通过 new 关键字、反射、clone 等方式创建，也可以通过工厂模式创建。对于复杂对象，使用 new 关键字、反射、clone 等方式创建存在如下缺点：

- ✓ 对象创建和对象使用的职责耦合在一起，违反单一原则；
- ✓ 当业务扩展时，必须修改代业务代码，违反了开闭原则；

而使用工厂模式将对象的创建和使用进行解耦，并屏蔽了创建对象可能的复杂过程，相对简单工厂模式，又具备更好的扩展性和可维护性，优点具体如下：

- ✓ 把对象的创建和使用的过程分开，对象创建和对象使用使用的职责解耦；
- ✓ 如果创建对象的过程很复杂，创建过程统一到工厂里管理，既减少了重复代码，也方便以后对创建过程的修改维护；
- ✓ 当业务扩展时，只需要增加工厂子类，符合开闭原则；

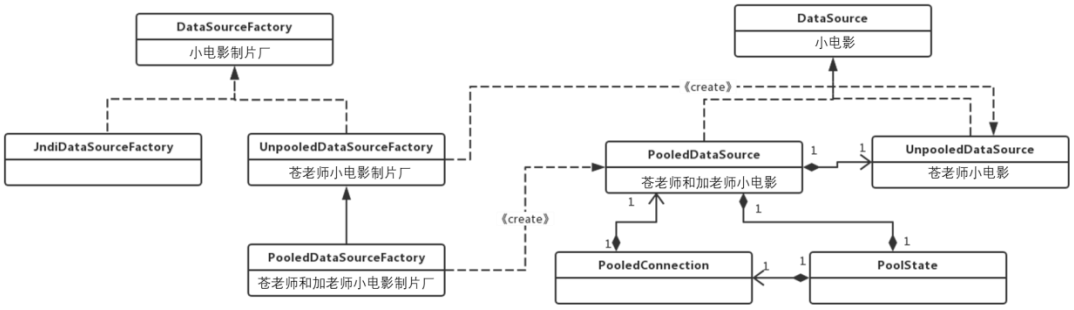
示例代码：com.enjoylearning.mybatis.factory.real.*

3.3 数据源的创建

数据源对象是比较复杂的对象，其创建过程相对比较复杂，对于 MyBatis 创建一个数据源，具体来讲有如下难点：

1. 常见的数据源组件都实现了 javax.sql.DataSource 接口；
2. MyBatis 不但要能集成第三方的数据源组件，自身也提供了数据源的实现；
3. 一般情况下，数据源的初始化过程参数较多，比较复杂；

综上所述，数据源的创建是一个典型使用工厂模式的场景，实现类图如下所示：



- ✓ DataSource：数据源接口，JDBC 标准规范之一，定义了获取获取 Connection 的方法；
- ✓ UnPooledDataSource：不带连接池的数据源，获取连接的方式和手动通过 JDBC 获取连接的方式是一样的；
- ✓ PooledDataSource：带连接池的数据源，提高连接资源的复用性，避免频繁创建、关闭连接资源带来的开销；
- ✓ DataSourceFactory：工厂接口，定义了创建 Datasource 的方法；
- ✓ UnpooledDataSourceFactory：工厂接口的实现类之一，用于创建 UnpooledDataSource(不带连接池的数据源)；
- ✓ PooledDataSourceFactory：工厂接口的实现类之一，用于创建 PooledDataSource（带连接池的数据源）；

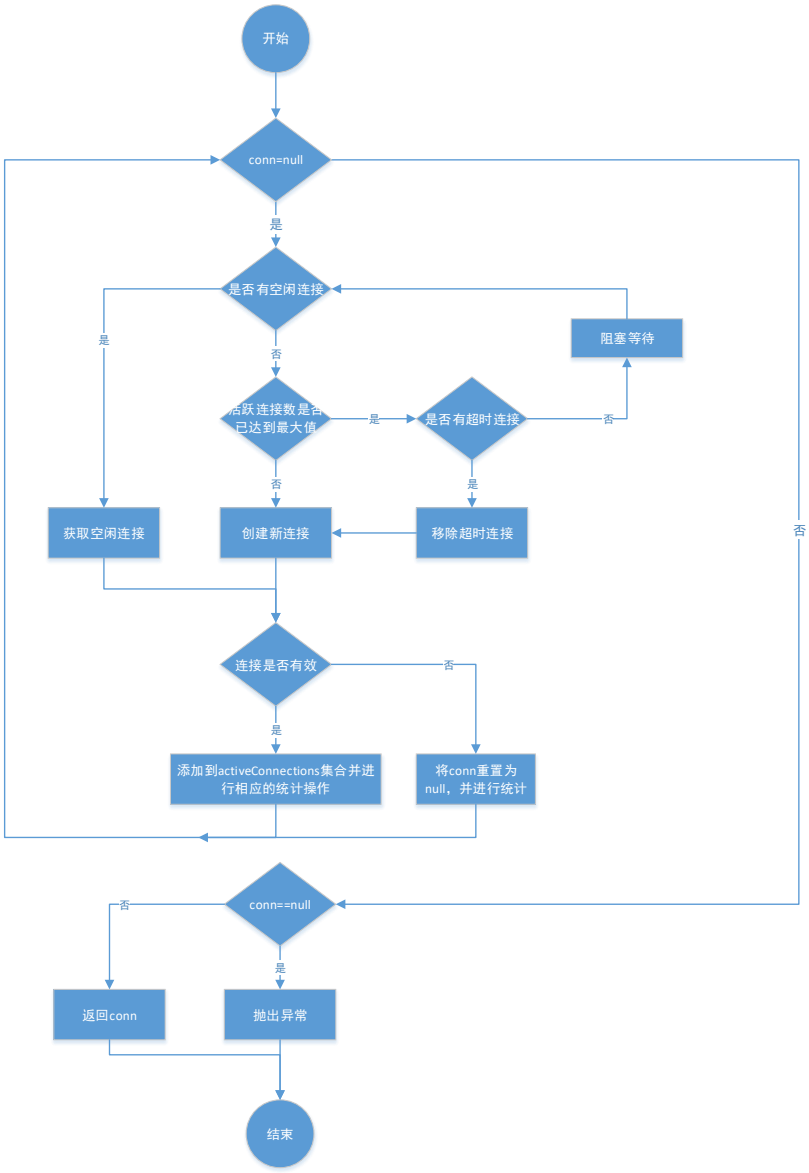
3.4 数据库连接池技术解析

数据库连接池技术是提升数据库访问效率常用的手段，使用连接池可以提高连接资源的复用性，避免频繁创建、关闭连接资源带来的开销，池化技术也是大厂高频面试题。MyBatis 内部就带了一个连接池的实现，接下来重点解析连接池技术的数据结构和算法；先重点分析下跟连接池相关的关键类：

- ✓ PooledDataSource：一个简单，同步的、线程安全的数据库连接池
- ✓ PooledConnection：使用动态代理封装了真正的数据库连接对象，在连接使用之前和关闭时进行增强；
- ✓ PoolState：用于管理 PooledConnection 对象状态的组件，通过两个 list 分别管理空闲状态的连接资源和活跃状态的连接资源，如下图，需要注意的是这两个 List 使用 ArrayList

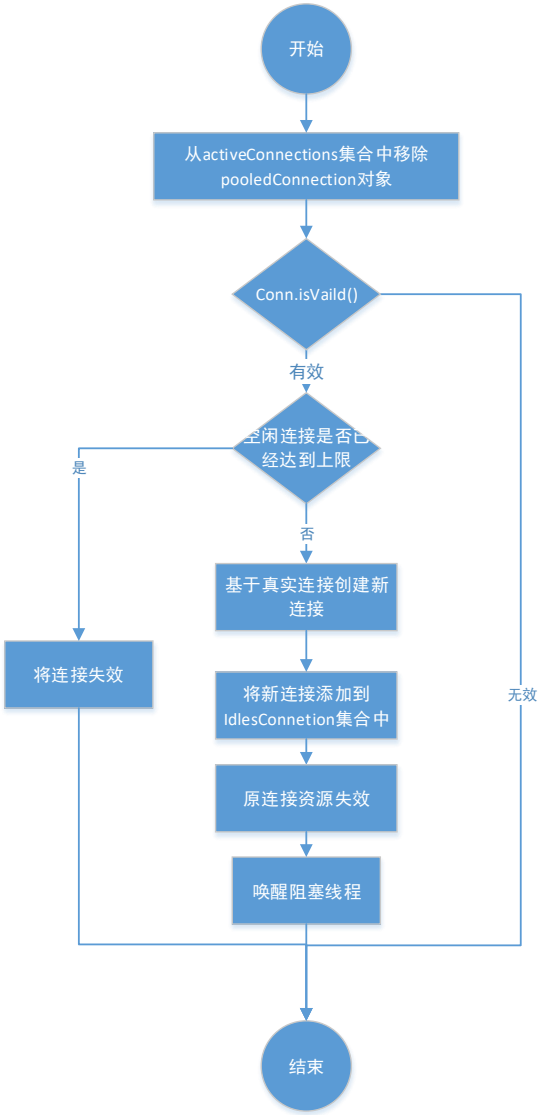
实现，存在并发安全的问题，因此在使用时，注意加上同步控制；

重点解析获取资源和回收资源的流程，获取连接资源的过程如下图：



参考代码：org.apache.ibatis.datasource.pooled.PooledDataSource.popConnection(String, String)

回收连接资源的过程如下图：



参考代码：org.apache.ibatis.datasource.pooled.PooledDataSource.pushConnection(PooledConnection)

4. 缓存模块分析

4.1 需求分析

MyBatis 缓存模块需满足如下需求：

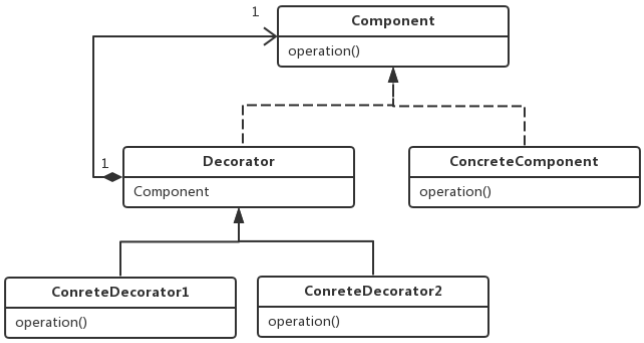
1. MyBatis 缓存的实现是基于 Map 的，从缓存里面读写数据是缓存模块的核心基础功能；
2. 除核心功能之外，有很多额外的附加功能，如：防止缓存击穿，添加缓存清空策略（fifo、lru）、序列化功能、日志能力、定时清空能力等；
3. 附加功能可以以任意的组合附加到核心基础功能之上；

基于 Map 核心缓存能力，将阻塞、清空策略、序列化、日志等等能力以任意组合的方式优雅的增强是 Mybatis 缓存模块实现最大的难题，用动态代理或者继承的方式扩展多种附加能

力的传统方式存在以下问题：这些方式是静态的，用户不能控制增加行为的方式和时机；另外，新功能的存在多种组合，使用继承可能导致大量子类存在。综上，MyBtis 缓存模块采用了装饰器模式实现了缓存模块；

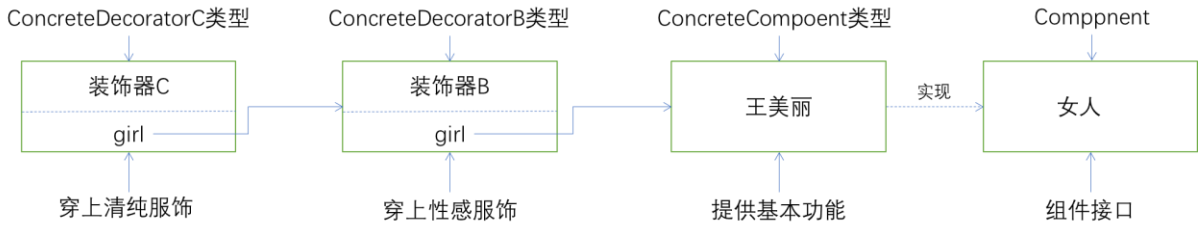
4.2 装饰器模式

装饰器模式是一种用于代替继承的技术，无需通过继承增加子类就能扩展对象的新功能。使用对象的关联关系代替继承关系，更加灵活，同时避免类型体系的快速膨胀。装饰器 UML 类图如下：



- ✓ 组件（Component）：组件接口定义了全部组件类和装饰器实现的行为；
- ✓ 组件实现类（ConcreteComponent）：实现 Component 接口，组件实现类就是被装饰器装饰的原始对象，新功能或者附加功能都是通过装饰器添加到该类的对象上的；
- ✓ 装饰器抽象类（Decorator）：实现 Component 接口的抽象类，在其中封装了一个 Component 对象，也就是被装饰的对象；
- ✓ 具体装饰器类（ConcreteDecorator）：该实现类要向被装饰的对象添加某些功能；

装饰器模式通俗易懂图示：



装饰器相对于继承，装饰器模式灵活性更强，扩展性更强：

- ✓ 灵活性：装饰器模式将功能切分成一个个独立的装饰器，在运行期可以根据需要动态的添加功能，甚至对添加的新功能进行自由的组合；
- ✓ 扩展性：当有新功能要添加的时候，只需要添加新的装饰器实现类，然后通过组合方式添加这个新装饰器，无需修改已有代码，符合开闭原则；

装饰器模式使用举例：

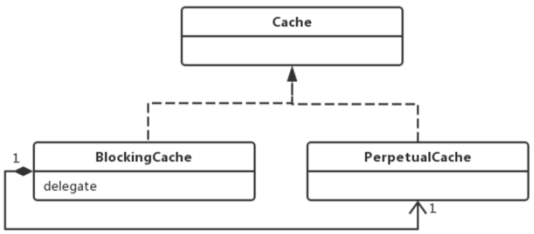
1. IO 中输入流和输出流的设计

```
BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(new FileInputStream("c://a.txt")));
```

2. 对网络爬虫的自定义增强，可增强的功能包括：多线程能力、缓存、自动生成报表、黑白名单、random 触发等

4.3 装饰器在缓存模块的使用

MyBatis 缓存模块是一个经典的使用装饰器实现的模块，类图如下：



- ✓ **Cache**：Cache 接口是缓存模块的核心接口，定义了缓存的基本操作；
- ✓ **PerpetualCache**：在缓存模块中扮演 ConcreteComponent 角色，使用 HashMap 来实现 cache 的相关操作；
- ✓ **BlockingCache**：阻塞版本的缓存装饰器，保证只有一个线程到数据库去查找指定的 key 对应的数据；

BlockingCache 是阻塞版本的缓存装饰器，这个装饰器通过 ConcurrentHashMap 对锁的粒度进行了控制，提高加锁后系统代码运行的效率（注：缓存雪崩的问题可以使用细粒度锁的方式提升锁性能），源码分析见：org.apache.ibatis.cache.decorators.BlockingCache；
除了 BlockingCache 之外，缓存模块还有其他的装饰器如：

1. LoggingCache：日志能力的缓存；
2. ScheduledCache：定时清空的缓存；
3. BlockingCache：阻塞式缓存；
4. SerializedCache：序列化能力的缓存；
5. SynchronizedCache：进行同步控制的缓存；

思考题：Mybatis 的缓存功能使用 HashMap 实现会不会出现并发安全的问题？
答：MyBatis 的缓存分为一级缓存、二级缓存。二级缓存是多个会话共享的缓存，确实会出现并发安全的问题，因此 MyBatis 在初始化二级缓存时，会给二级缓存默认加上 SynchronizedCache 装饰器的增强，在对共享数据 HashMap 操作时进行同步控制，所以二级缓存不会出现并发安全问题；而一级缓存是会话独享的，不会出现多个线程同时操作缓存数据的场景，因此一级缓存也不会出现并发安全的问题；

4.4 缓存的唯一标识 CacheKey

MyBatis 中涉及到动态 SQL 的原因，缓存项的 key 不能仅仅通过一个 String 来表示，所以通过 CacheKey 来封装缓存的 Key 值，CacheKey 可以封装多个影响缓存项的因素；判断两个 CacheKey 是否相同关键是比较两个对象的 hash 值是否一致；构成 CacheKey 对象的要素包括：

1. mappedStatment 的 id
2. 指定查询结果集的范围（分页信息）
3. 查询所使用的 SQL 语句
4. 用户传递给 SQL 语句的实际参数值

checksum、count 以及 updateList 进行更新；

- ✓ equals 方法：用于比较两个元素是否相等。首先比较 hashcode、checksum、count 是否相等，如果这三个值相等，会循环比较 updateList 中每个元素的 hashCode 是否一致；按照这种方式判断两个对象是否相等，一方面能很严格的判断是否一致避免出现误判，另外一方面能提高比较的效率；

5. 反射模块分析

反射是 Mybatis 模块中类最多的模块，通过反射实现了 POJO 对象的实例化和 POJO 的属性赋值，相对 JDK 自带的反射功能，MyBatis 的反射模块功能更为强大，性能更高；反射模块关键的几个类如下：

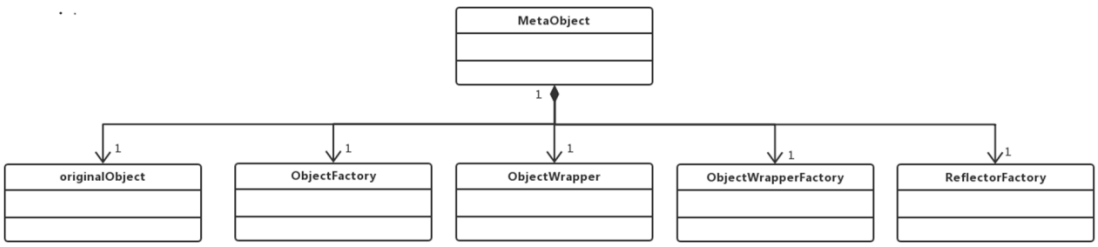
ObjectFactory：MyBatis 每次创建结果对象的新实例时，它都会使用对象工厂（ObjectFactory）去构建 POJO；

ReflectorFactory：创建 Reflector 的工厂类，Reflector 是 MyBatis 反射模块的基础，每个 Reflector 对象都对应一个类，在其中缓存了反射操作所需要的类元信息；

ObjectWrapper：对对象的包装，抽象了对象的属性信息，他定义了一系列查询对象属性信息的方法，以及更新属性的方法；

ObjectWrapperFactory： ObjectWrapper 的工厂类，用于创建 ObjectWrapper ；

MetaObject：封装了对象元信息，包装了 MyBatis 中五个核心的反射类。也是提供给外部使用的反射工具类，可以利用它可以读取或者修改对象的属性信息；MetaObject 的类结构如下所示：



示例代码：com.enjoylearning.mybatis.MybatisDemo.reflectionTest()