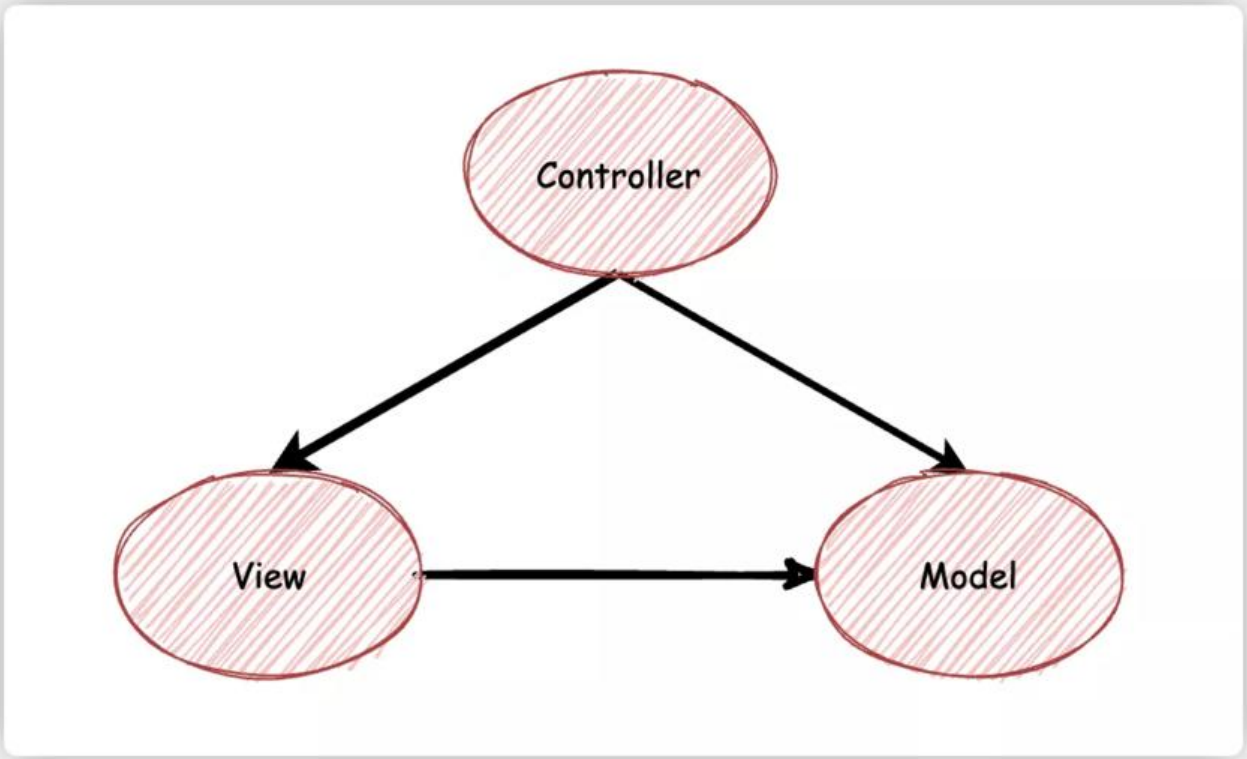
为什么要给 MVC 分层架构再加一层 Manager 层?

作者：jianzh5
原文链接：https://ld246.com/article/1634634833503
来源网站：链滴

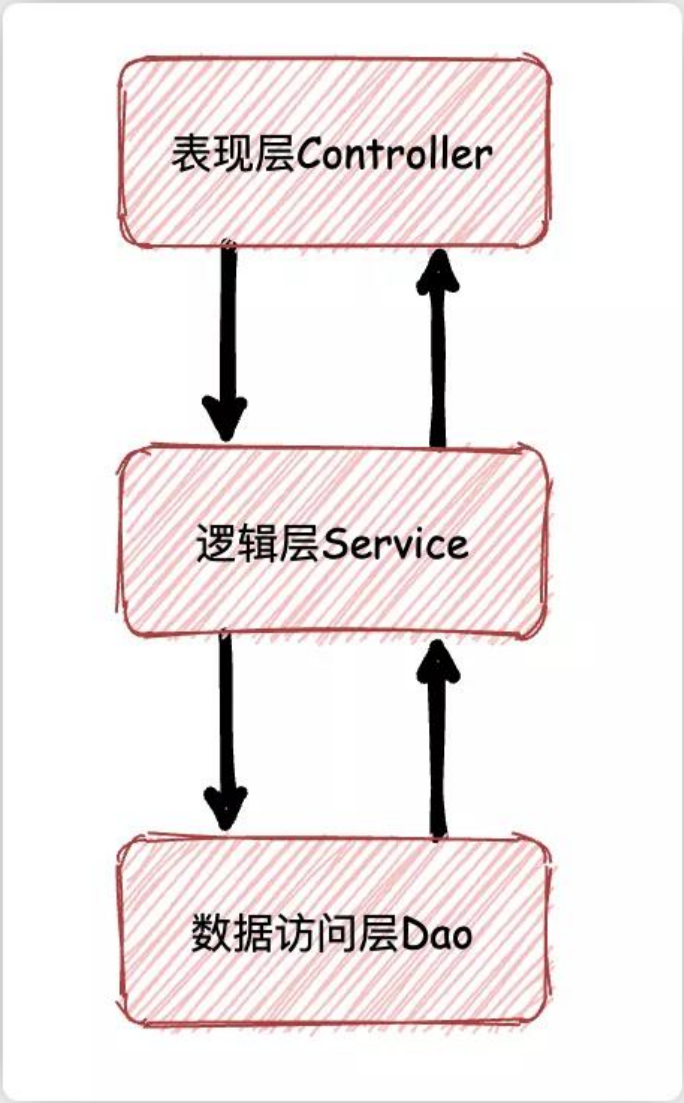
MVC三层架构

我们在刚刚成为程序员的时候，就会被前辈们 “教育” 说系统的设计要遵循 MVC（Model-View-Controller）架构。它将整体的系统分成了 Model（模型），View（视图）和 Controller（控制器）三层次，也就是将用户视图和业务处理隔离开，并且通过控制器连接起来，很好地实现了表现和逻辑的耦，是一种标准的软件分层架构。



MVC分层架构是架构上最简单的一种分层方式。为了遵循这种分层架构我们在构建项目时往往会建立样三个目录：controller、service 和 dao，它们分别对应了表现层、逻辑层还有数据访问层。

原文链接：为什么要给 MVC 分层架构再加一层 Manager 层?



每层的作用如下：

- Controller层：主要是对访问控制进行转发，各类基本参数校验，或者不复用的业务简单处理。
- Service层：主要是处理业务逻辑和事务
- Dao层：负责与底层数据库MySQL，Oracle等进行数据交互

可是随着我们的业务逻辑越来复杂，代码写的越来越多，这种简单的三层架构的问题也越来越明显。

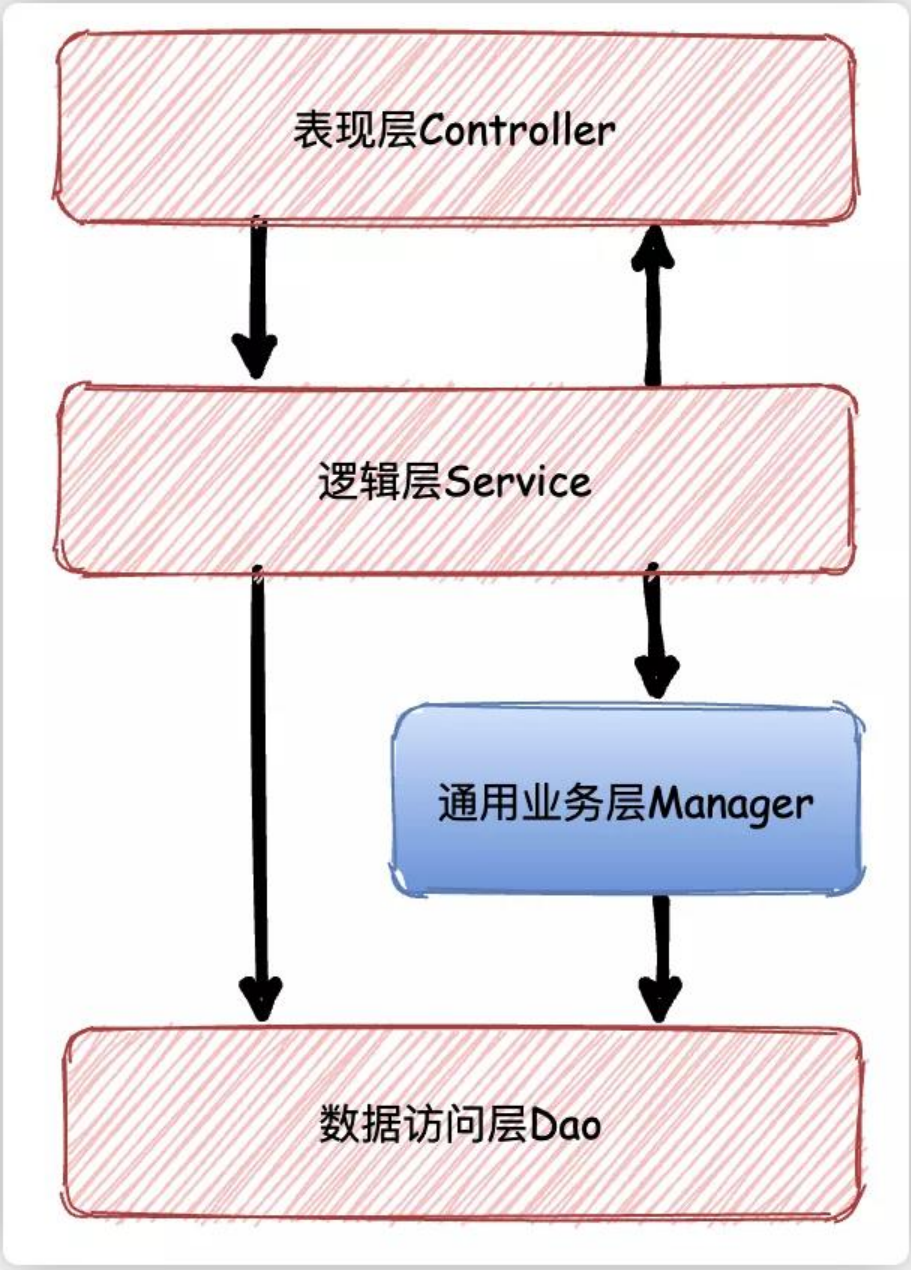
MVC架构弊端

传统的MVC分层有以下几个很明显的问题：

1. Service层代码臃肿
2. Service层很容易出现大事务，事务嵌套，导致问题很多，而且极难排查
3. dao层参杂业务逻辑
4. dao层sql语句复杂，关联查询比较多

原文链接：为什么要给 MVC 分层架构再加一层 Manager 层?

为了解决这个问题，我们参考《alibaba java开发手册》，在Service层之下再独立出一个通用业务处层（Manager层）



在这个分层架构中主要增加了 Manager 层，它与 Service 层的关系是：Manager 层提供原子的服接口，Service 层负责依据业务逻辑来编排原子接口。

Manager层的特征

在《alibaba java开发手册》中是这样描述Manager层的：

Manager 层：通用业务处理层，它有如下特征：

1. 对第三方平台封装的层，预处理返回结果及转化异常信息，适配上层接口；
2. 对 Service 层通用能力的下沉，如缓存方案、中间件通用处理；

原文链接：为什么要给 MVC 分层架构再加一层 Manager 层?

3. 与 DAO 层交互，对多个 DAO 的组合复用。

在实际开发中我们可以这样使用Manager层

1. 复杂业务，service提供数据给Manager层，负责业务编排，然后把事务下沉到Manager层，Manaer层不允许相互调用，不会出现事务嵌套。

2. 专注于不带业务sql语言，也可以在manager层进行通用业务的dao层封装。

3. 避免复杂的join查询，数据库压力比java大很多，所以要严格控制好sql，所以可以在manager层行拆分，比如复杂查询。

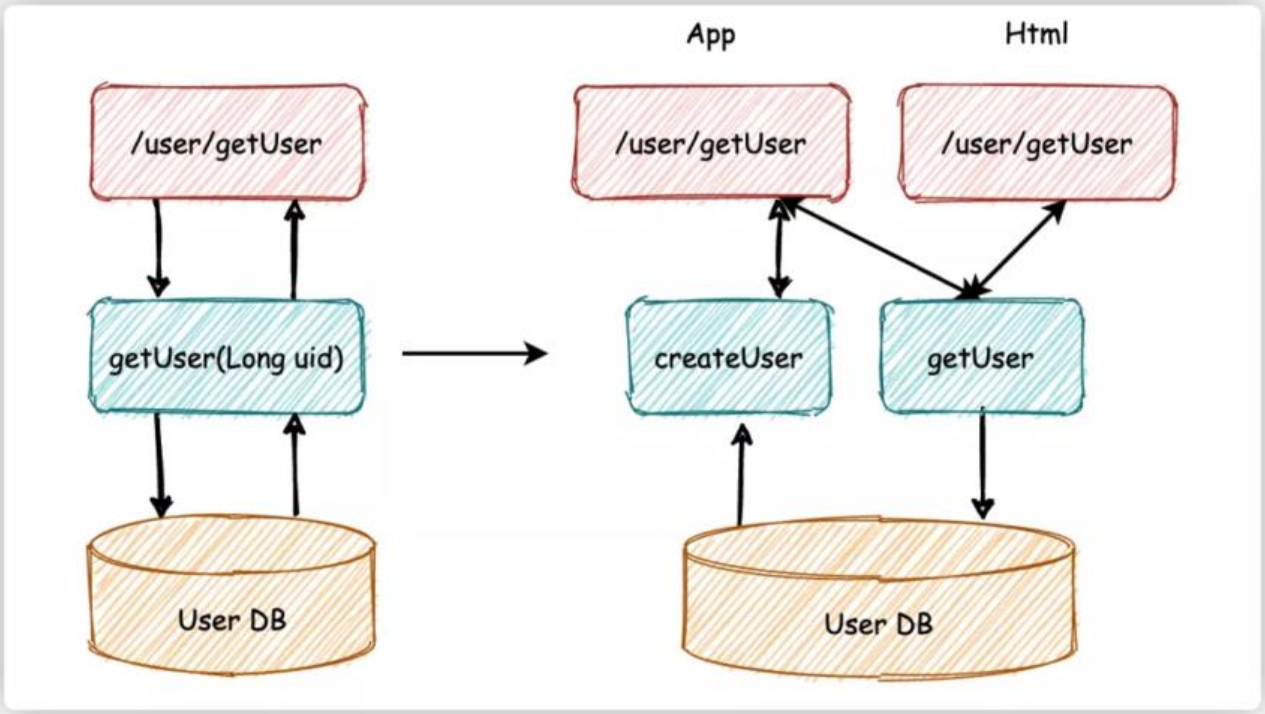
当然对于简单的业务，可以不使用Manager层。

Manager层使用案例

这里我们举个例子说明一下Manager层的使用场景：

假设你有一个用户系统，他有一个获取用户信息的接口，它调用逻辑Service层的 getUser 方法，getser方法又和 User DB 交互获取数据。如下图左边展示部分。

这时，产品提出一个需求，在 APP 中展示用户信息的时候，如果用户不存在，那么要自动给用户创一个用户。同时，要做一个 HTML5 的页面，HTML5 页面要保留之前的逻辑，也就是不需要创建用。



此时按照传统的三层架构，逻辑层的边界就变得不清晰，表现层也承担了一部分的业务逻辑，因为我往往会在表现层Controller中增加业务逻辑处理，将获取用户和创建用户接口编排起来。

而添加Manager层以后，Manager 层提供创建用户和获取用户信息的接口，而 Service 层负责将这个接口组装起来。这样就把原先散布在表现层的业务逻辑都统一到了 Service 层，每一层的边界就非清晰了。

原文链接：为什么要给 MVC 分层架构再加一层 Manager 层?

接下来我们看一段实际代码说明一下Service层与Manager层如何进行区分?

```
@Transactional(rollbackFor = Throwable.class)
public Result<String> upOrDown(Long departmentId, Long swapId) {
  // 验证 1
  DepartmentEntity departmentEntity = departmentDao.selectById(departmentId);
  if (departmentEntity == null) {
    return Result.error("部门xxx不存在");
  }
  // 验证 2
  DepartmentEntity swapEntity = departmentDao.selectById(swapId);
  if (swapEntity == null) {
    return Result.error("部门xxx不存在");
  }
  // 验证 3
  Long count = employeeDao.countByDepartmentId(departmentId);
  if (count != null && count > 0) {
    return Result.error("员工不存在");
  }
  // 操作数据库 4
  Long departmentSort = departmentEntity.getSort();
  departmentEntity.setSort(swapEntity.getSort());
  departmentDao.updateById(departmentEntity);
  swapEntity.setSort(departmentSort);
  departmentDao.updateById(swapEntity);
  return Result.OK("success");
}
```

上面代码在我们在我们采用三层架构时经常会遇到，那么它有什么问题呢?

上面的代码是典型的长事务问题（类似的还有调用第三方接口），前三步都是使用 connection 进行证操作，但是由于方法上有@Transactional 注解，所以这三个验证都是使用的同一个 connection。

若对于复杂业务、复杂的验证逻辑，会导致整个验证过程始终占用该 connection 连接，占用时间可会很长，直至方法结束，connection 才会交还给数据库连接池。

对于复杂业务的不可预计的情况，长时间占用同一个 connection 连接不是好的事情，应该尽量缩短用时间。

说明：对于@Transactional 注解，当 spring 遇到该注解时，会自动从数据库连接池中获取 connectio，并开启事务然后绑定到 ThreadLocal 上，如果业务并没有进入到最终的 操作数据库环节，那么就有必要获取连接并开启事务，应该直接将 connection 返回给数据库连接池，供其他使用。

所以我们在加入Manager层以后可以这样写:

DepartmentService.java

```
public Result<String> upOrDown(Long departmentId, Long swapId) {
  // 验证 1
  DepartmentEntity departmentEntity = departmentDao.selectById(departmentId);
  if (departmentEntity == null) {
    return Result.error("部门xxx不存在");
  }
  // 验证 2
  DepartmentEntity swapEntity = departmentDao.selectById(swapId);
```

```
  if (swapEntity == null) {
    return Result.error("部门xxx不存在");
  }
  // 验证 3
  Long count = employeeDao.countByDepartmentId(departmentId);
  if (count != null && count > 0) {
    return Result.error("员工不存在");
  }

  // 操作数据库 4
  departmentManager.upOrDown(departmentSort,swapEntity);

  return Result.OK("success");
}
```

DepartmentManager.java

```
@Transactional(rollbackFor = Throwable.class)
public void upOrDown(DepartmentEntity departmentEntity ,DepartmentEntity swapEntity){
  Long departmentSort = departmentEntity.getSort();
  departmentEntity.setSort(swapEntity.getSort());
  departmentDao.updateById(departmentEntity);
  swapEntity.setSort(departmentSort);
  departmentDao.updateById(swapEntity);
}
```

将数据在 service 层准备好，然后传递给 manager 层，由 manager 层添加 @Transactional事务注进行数据库操作。