天津大学

设计模式实践课程报告

学　　院　国际工程师学院

专　　业　电子信息（计算机技术）

组　　员　王梓蔚　2021229036

组　　员　廖　松　2021229041

组　　员　王　翔　2021229089

任课教师　李罡

实验指导　李罡

2022 年　05 月　09 日

第一章 总体设计

1.1 基本配置

（1）编译环境：在本次实验中编译器采用 Intellij IDEA 2021.2.2，编程语言采用 JAVA，IDK 版本号为 jdk 11，整体采用 Maven4 + MyBatis + SpringBoot2.4.0 的框架，数据库使用 MySQL 存储基本实验数据。

（2）操作系统：Windows 10

1.2 实现方案

1.2.1 模块划分

根据系统用户的需求，将本系统按功能划分成三大主要模块，即请假调休系统、考勤系统、后台管理模块，涉及到的用户包括普通员工，人事处员工，财务部员工，部门经理，副总经理，总经理和管理人员七大类用户

1、请假、调休系统模块

本模块的功能是在线请假以及在线调休的实现及管理，普通员工通过此功能模块进行在线请假和调休及查看请假记录信息;管理人员可以在线审批员工的请假调休信息及查看请假调休记录信息；领导在线审批员工长时间的请假及查看请假记录信息。

2、考勤系统模块

本模块的功能是员工考勤信息统计的实现、查看及管理，涉及七大类用户中的所有用户。普通员工在线查看自己所有的出勤信息;管理层人员可以在线管理学生出勤信息。

3、后台管理管理

本模块的功能实现整个系统数据的同步更新及维护，只涉及系统管理员用户。系统管理员动态的管理员工的基本信息、请假状态、权限设置等，是整个系统实现的基础。

1.2.2 模块之间交互关系

该系统模块之间的交互关系主要采用同步接口调用的方式，该方式较简单，并且采用同步调用的方式，不易出现同时对同一个数据性进行脏读的现象。并且由于考勤管理系统的接口的多样性，在接口调用时，由于接口直接更多属于上下层级关系，因此采用同步接口调用的方式会更好。

1.2.3 代码分层

根据传统的 MVC 设计模式，可以将系统大致分为 Controller 层、Service 层、Mapper 层以及实体 Entity 层。在本次实验中更加注重和管理项目中的设计模式和开发原则，因此额外将通常的 Repository 层修改为 Mapper 层隔离那些与数据库的操作。因此通过设计，可以大幅简化 Service 层的代码量，仅保留基本的接口，这样不仅从可视层面亦或是代码管理层面都具有显著的优化效果。

1. Controller 层主要负责和用户前端交互的信息的传递，留出相应与前端的接口。该层设计对于每一类不同实体的独立操作分别添加不同的响应接口，这样也能够使得代码更加符合迪米特法则所描述的只与“朋友”进行通信。

2. Mapper 主要负责与数据库交互，带有不同参数的 mapper 与响应的 service 层中的接口对应，分别通过持久化 Mybatis 执行相关的数据库操作。

3. Service 层主要负责封装具体的业务逻辑供 Controller 层调用。该层同时定义了响应的 service 接口以及他们的具体实现类。并通过相应的设计模式和原则对复用性进行了提高。

4. Entity 层主要负责存放不同实体类型的属性和接口。该层与 Service 层相互配合，在本次实验中主要存放的是基于贫血模式的 Entity 实体，其中只一种实体对应数据库中的一张表，将面对对象的操作放于 Service 层中进行编写，从而更好的面向接口编程。

第二章 详细系统设计

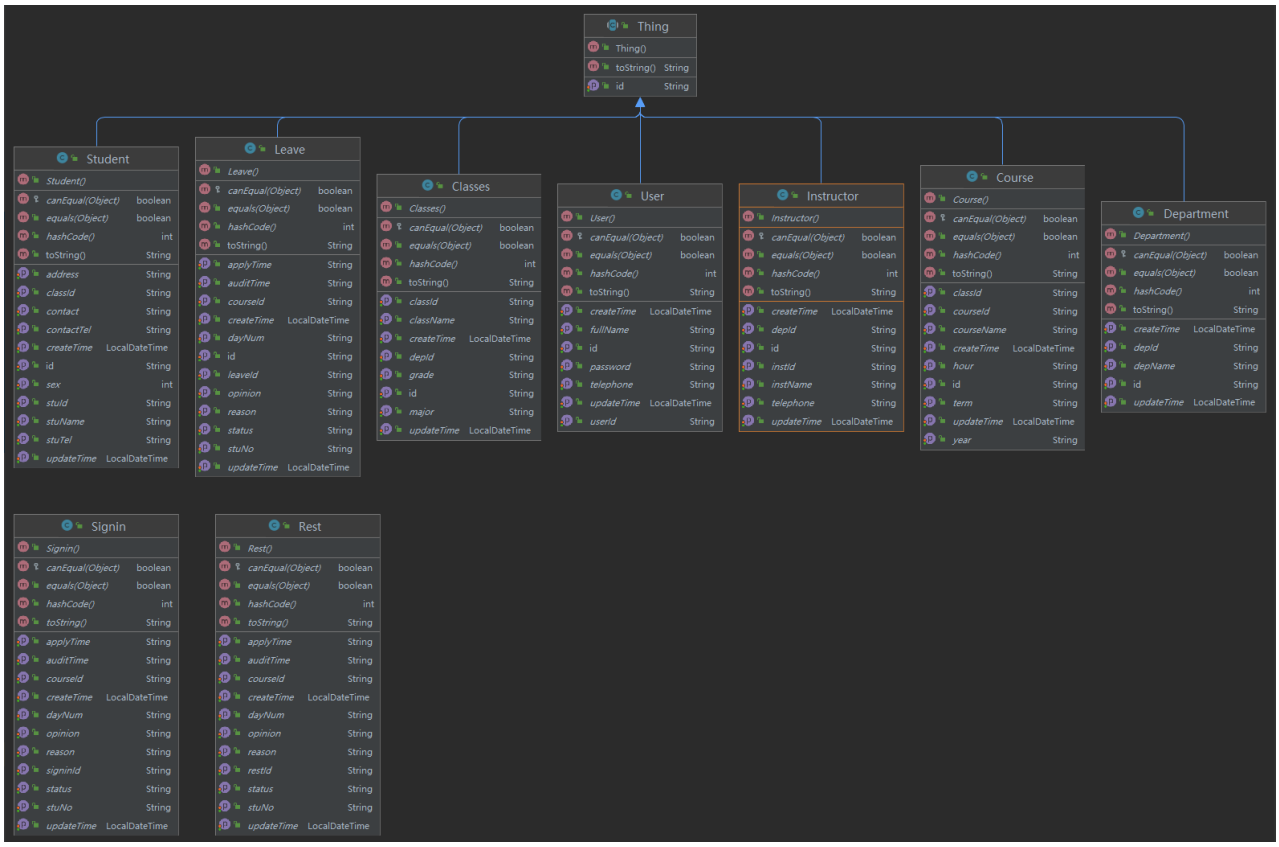
该节将在第一章的基础上更加详细的介绍本次实验中的具体设计，主要包括两个方面：将在系统结构设计中具体描述项目文件组成以及在项目代码中涉及到的基本设计原则，将在数据库设计中介绍后端数据库表的具体设计。

2.1 系统结构设计

根据上一章所述，本次实验将整体系统划分为 Controller 层、Service 层、Mapper 层以及实体 Entity 层四层，其中附加一个 Util 工具包层。

2.1.1 Entity 层

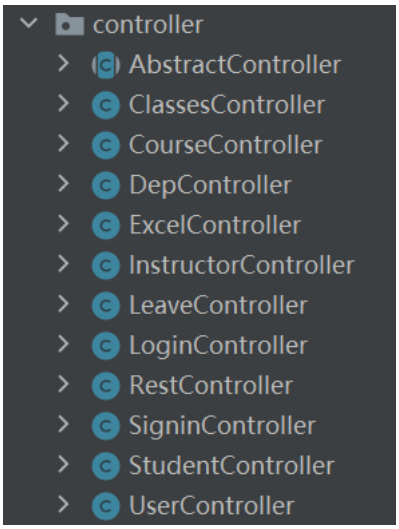
该层中的 UML 图如下图所示：



在该层中一共定义了十个类，其中 Thing 类作为抽象实体被定义。其他类通过继承该顶层父类使得我们仅需要通过定义一个 Thing 类实体便可以通过多态的不同简单操作进行复用，即便这些类中的属性不同。

2.1.2 Controller 层

Controller 层主要是通过 Restful 接口于用户界面进行交互。该层中的文件结构如下图所示：



通过单一职责原则，我们为不同的实体类分别定义了相关的 Controller 控制器，但是这样做的代价相对较大，导致了控制器的数量变得繁杂，因此采用外观模式为这些不同的控制器定义了一个公共的控制器父类 AbstractController，将这些不同控制器中相似的代码和方法进行提取，其他控制器通过继承该类以获得已有的父类中函数或是重用代码，以 nextPage 函数举例：

```
/**
     * 根据是否存在模糊查询内容跳转到不同的分页查询方法
     * @param modelAndView
     * @param thing
     * @param pageNumber
     * @param page
     * @param request
     * @return
     */
    public ModelAndView nextPage(ModelAndView modelAndView, Thing thing, Integer pageNumber,
                                 Page page, HttpServletRequest request) {
        if (request.getSession().getAttribute("result") != null) {
            request.getSession().setAttribute("result", "");
        }
        if (thing.getId() == null) {
            return findAll(modelAndView, page, pageNumber, request);
        } else {
            return findById(modelAndView, thing, pageNumber, page, request);
        }
    }
```

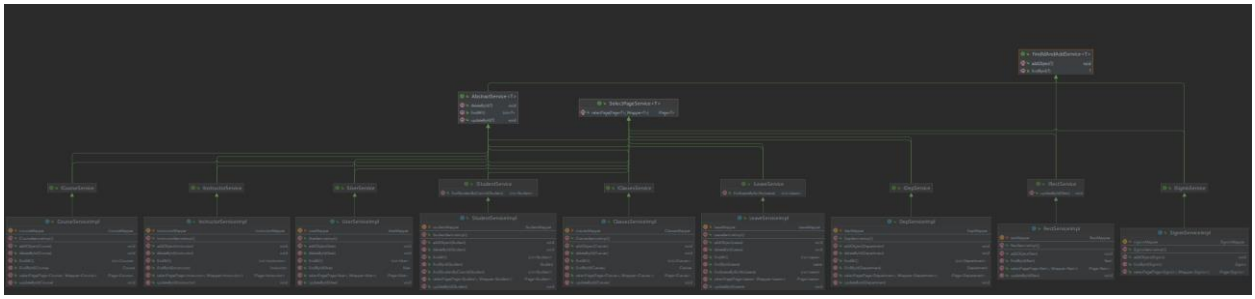
由于在各个控制器中均含有该段相似代码，因此我们将该方法提取至父类抽象控制器中，并用抽象对象 Thing 类来对具体的实例类进行改写。因此在实现具体的控制器时，并

不需要将该段代码重复编写，仅需要 super 语句便可以获取到父类中相同的操作。

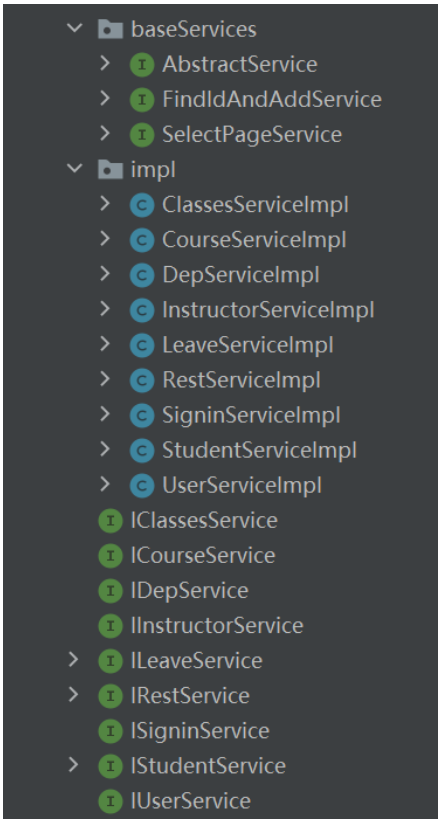
通过外观模式将一些通用的操作封装成接口进行面板展示，在实现具体的控制器时仅需要重用面板方法或是利用面板方法加上该控制器特有代码块即可。

2.1.3 Service 层

Service 层中包括了对具体代码逻辑的实现和调用，通过该层将数据库操作和前端进行传递和处理。该层中的 UML 图如下图所示：



由于 UML 不方便查看，提供该层中的文件结构如下图所示：

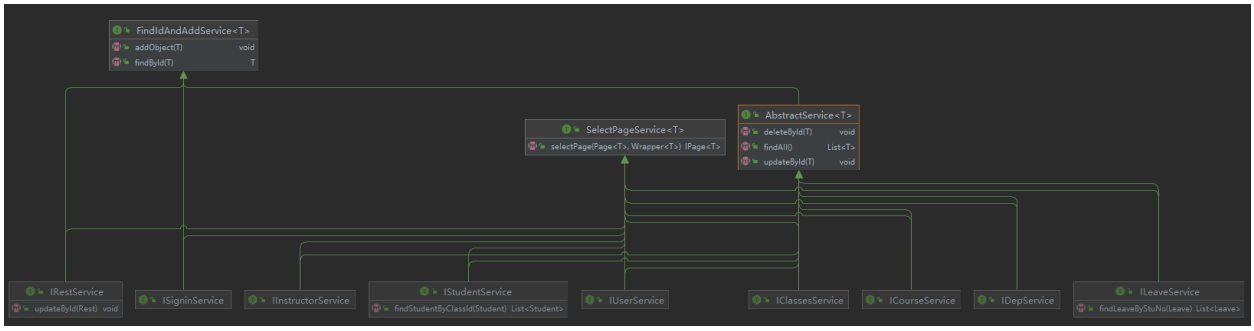


在本次实验中，为了更好的满足接口隔离原则和开闭原则，首先定义了 SelectPageService 和 FindIdAndAddService，AbstractService 三个基类接口，由于分页操作与数据库的增删改查操作不同，因此将分页功能的接口单独设置在 SelectPageService 接口

中；并且对于该实验来说，能够通过 Id 获取到对象实例以及添加功能是每一个类必需具有的功能。因此将这两个功能单独封装于 FindIdAndAddService 接口中。对于 AbstractService 接口，主要包括了对数据库通用的操作，因此采用装饰者模式，在继承了 FindIdAndAddService 接口的基础上额外添加新的接口。

有了三个基本接口后，便可以直接利用这三个接口通过组合复用的方式进行不同类对象 Service 接口的编写。

主要的接口形成了以下的类图：

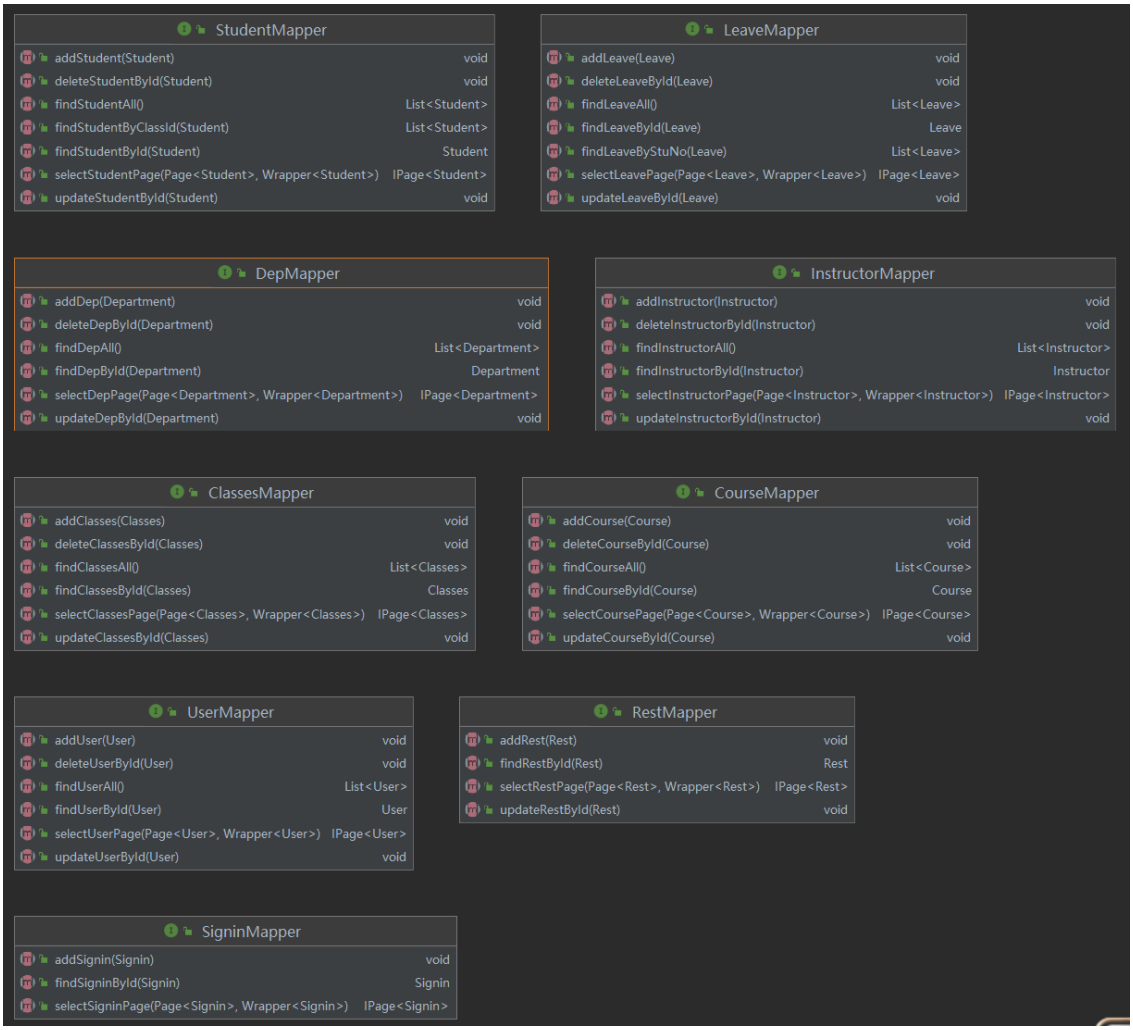


可以看到具体的接口内部的方法数量明显减少，并且符合开闭原则，如果在具体的 Service 接口中需要额外添加新的功能，无需对原有的额方法进行修改，因为接口隔离原则保证了仅对最小的额接口进行了继承，当有新方法时，在具体的接口中添加即可。

该层的最后一部分是对于接口的实现类。我们通过单一职责原则可知仅需要对当前的每一个类型的接口进行方法实现即可，直接调用 Mapper 层中相应的操作。

2.1.4 Mapper 层

该层中的 UML 图如下图所示：

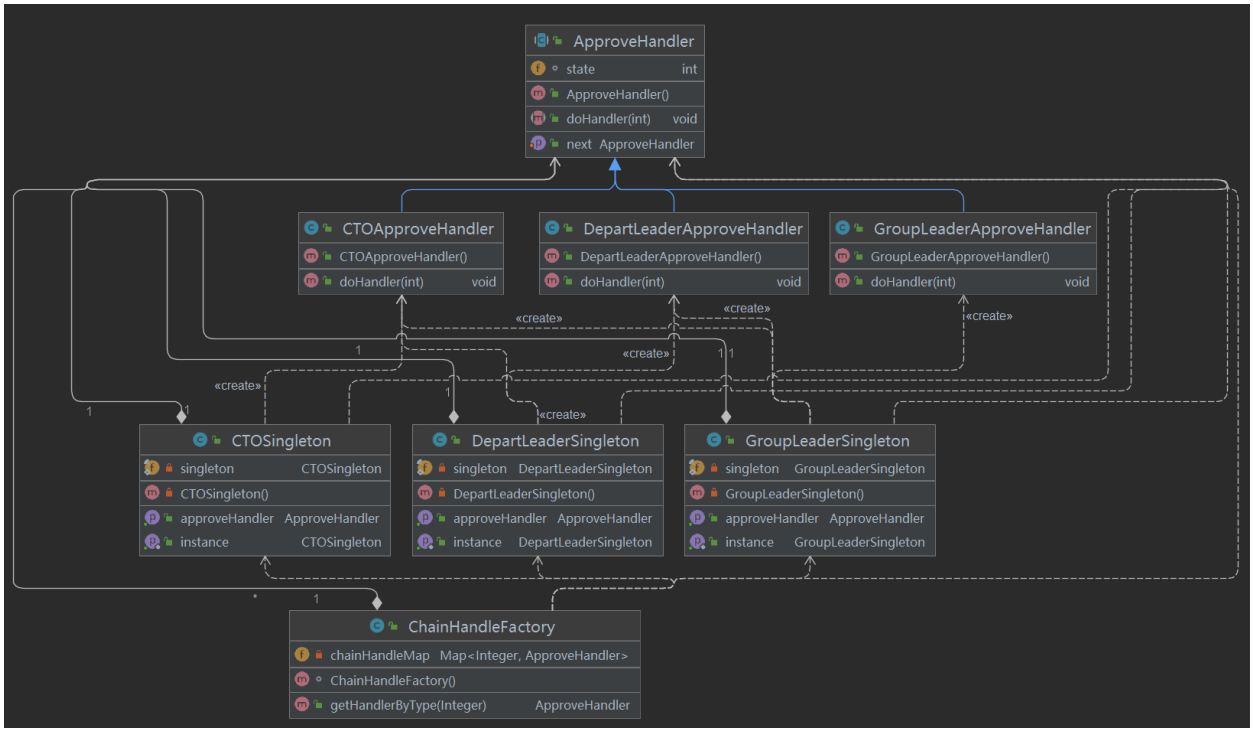


该层中不涉及相应的设计模式，由于 Mapper 对应了不同 Service 接口的实现，并且需要对特定功能进行细化和区分，因此重写每一个方法名即可，在该层通过 MyBatis 持久层于后台数据库进行交互。

2.1.5 Util 层

Util 主要负责在该项目中涉及到的工具函数的编写和存放部分，由于在该实验中并未设计过多复杂的处理逻辑，仅此目前仅有处理请假调休请求该功能，该功能主要使用在设计模式中的责任链模式、简单工厂模式和单例模式进行编写从而保证处理请假调休的逻辑是可以在不同的链式处理结构中生效并且无需重复获取处理链，从而降低运行成本。

该层中的 UML 图如下图所示：



Approve 为该责任链中的基本单位，分别通过继承定义组长，部门经理，总经理三个处理单位，又为这三个处理单位分别设置了一个单例模式，在单例对象中形成该层级下的处理链，然后存放到责任链工厂中以供 Controller 中处理逻辑时使用。这三个具体设计模式的实现将在下一章进行介绍。

2.2 数据库设计

本系统使用 MySQL 数据库，通过前期的筹划和系统设计，创建了数据库的物理表，设计规范的数据表能够完备的存储当前系统产生的数据，从而确保考勤系统的功能性设计和业务逻辑规范。

数据库的开发设计是总体构建的处于底层核心位置，数据库设计的优劣能够影响到平台的运行能力，而且还为后期的运维和系统扩展产生深远的影响。如果设计中出现大量的重复、垃圾数据会导致系统的运行变慢、用户体验变差，会提前进入系统生命周期的消亡阶段，因此数据库的设计至关重要。在构建数据库的时候要考虑和遵循科学实用的设计理念，所有表与表之间的关键数据调用都要尽量遵从一致的存储原则，这样可以保证数据的准确性和唯一性。数据输出在满足系统要求的同时不增加数据库本身的应用压力。

数据库的设计包含以下内容：信息表的创建需包含全部系统运行产生的数据；使用数据理念、存储数据使其创建规范化、正规化；数据展示简洁明确，物理表通过 KEY 和唯一标识进行管理；数据字段做到命名规范。下面展示具体的数据库物理表设计：

2.2.1 物理表设计

表 1 打卡签到信息表

字段名	类型	是否为空	索引	默认值	说明
id	int	NO	PRI	无	编码
signin_id	varchar(30)	YES	无	无	签到编号
status	varchar(3)	YES	无	无	状态
create_time	timestamp	YES	无	CURRENT_TIMESTAMP	创建时间

表 2 产品线信息表

字段名	类型	是否为空	索引	默认值	说明
id	int	NO	PRI	无	编号
course_id	varchar(30)	YES	无	无	课程名称
course_name	varchar(50)	YES	无	无	课程名称
create_time	timestamp	YES	无	CURRENT_TIMESTAMP	创建时间
update_time	timestamp	YES	无	CURRENT_TIMESTAMP	修改时间

表 3 部门信息表

字段名	类型	是否为空	索引	默认值	说明
id	int	NO	PRI	无	编码
dep_id	varchar(30)	YES	无	无	二级部门编码
dep_name	varchar(50)	YES	无	无	名称
create_time	timestamp	YES	无	CURRENT_TIMESTAMP	创建时间
update_time	timestamp	YES	无	CURRENT_TIMESTAMP	修改时间

表 4 总经理信息表

字段名	类型	是否为空	索引	默认值	说明
id	int	NO	PRI	无	编号
inst_id	varchar(30)	YES	无	无	工号
inst_name	varchar(50)	YES	无	无	姓名
dep_id	varchar(30)	YES	无	无	二级部门编号
telephone	varchar(11)	YES	无	无	联系电话
create_time	timestamp	YES	无	CURRENT_TIMESTAMP	创建时间
update_time	timestamp	YES	无	CURRENT_TIMESTAMP	修改时间

表 5 请假信息表

字段名	类型	是否为空	索引	默认值	说明
id	int	NO	PRI	无	编码
leave_id	varchar(30)	YES	无	无	请假编号
reason	varchar(150)	YES	无	无	请假理由
day_num	int	YES	无	无	天数
apply_time	datetime	YES	无	无	请假时间
status	varchar(3)	YES	无	无	状态

audit_time	datetime	YES	无	无	审核已经
opinion	varchar(150)	YES	无	无	
create_time	timestamp	YES	无	CURRENT_TIMESTAMP	创建时间
update_time	timestamp	YES	无	CURRENT_TIMESTAMP	修改时间

表 6 调休信息表

字段名	类型	是否为空	索引	默认值	说明
id	int	NO	PRI	无	编码
rest_id	varchar(30)	YES	无	无	请假编号
reason	varchar(150)	YES	无	无	请假理由
day_num	int	YES	无	无	天数
apply_time	datetime	YES	无	无	请假时间
status	varchar(3)	YES	无	无	状态
audit_time	datetime	YES	无	无	审核已经
opinion	varchar(150)	YES	无	无	
create_time	timestamp	YES	无	CURRENT_TIMESTAMP	创建时间
update_time	timestamp	YES	无	CURRENT_TIMESTAMP	修改时间

表 7 普通员工信息表

字段名	类型	是否为空	索引	默认值	说明
id	int	NO	PRI	无	编码
work_id	varchar(30)	YES	无	无	学号
class_id	varchar(30)	YES	无	无	班号
name	varchar(50)	YES	无	无	姓名
sex	int	YES	无	无	性别
address	varchar(150)	YES	无	无	通讯地址
tel	varchar(11)	YES	无	无	员工电话
contact	varchar(30)	YES	无	无	联系人
create_time	timestamp	YES	无	CURRENT_TIMESTAMP	创建时间
update_time	timestamp	YES	无	CURRENT_TIMESTAMP	修改时间

表 8 系统用户信息表

字段名	类型	是否为空	索引	默认值	说明
id	int	NO	PRI	无	编号
user_id	varchar(30)	YES	无	无	工号
full_name	varchar(50)	YES	无	无	姓名
password	varchar(50)	YES	无	无	密码
telephone	varchar(11)	YES	无	无	电话
create_time	timestamp	YES	无	CURRENT_TIMESTAMP	创建时间
update_time	timestamp	YES	无	CURRENT_TIMESTAMP	修改时间

第三章 设计模式总结

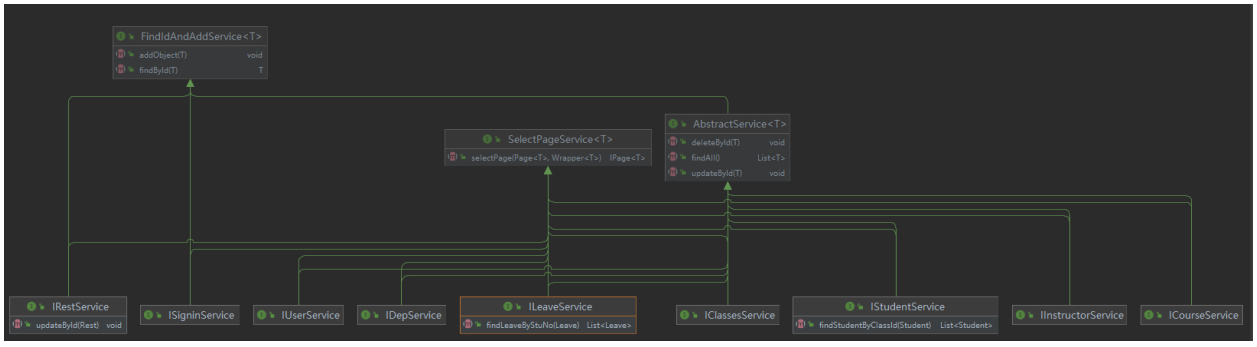
本章将着重对该次实验中涉及到的设计原则和设计模式进行总结，分析这些设计模式的使用情况以及带给系统的优势。

3.1 设计基本原则

3.1.1 单一职责原则

单一职责原则主要描述了一个类应该仅有一个引起它变化的原因。通俗来说就是一个类，应该拥有单一的职责，如果多个职责耦合在一起会造成职责的相互影响，导致设计变得很脆弱。因为很难划分职责范围，所以这个原则也是最容易在开发中被打破的。

但是在该实验中，为了从最大程度上实现单一职责原则，我们对于数据库中不同的表设计不同的实体类进行区分，其次在 MVC 框架中，我们将不同实体类型的操作进行区分开，并且将这些操作以更小粒度进行划分，在需要使用时在通过组合和继承进行实现复用。正如 Service 层中基础 service 类的设计：



将查询分页的功能与对数据库增删改查的功能分离开来，这样变能更好的实现单一职责原则。

3.1.2 开闭原则

开闭原则主要描述了一个类应该对拓展开放，对修改关闭。是我们在该次实验设计中非常核心的一个原则。要求是，类的行为可拓展，而且是在不修改现有代码的前提下进行拓展。开闭原则实现的关键在于，合理的抽象出、分离出变化与不变的部分。为变化部分预留下可拓展的方式，比如钩子方法，或动态组合对象等。

在本次实验中，采用外观模式和对 MVC 各层进行抽象都是开闭原则很好地

提现。我们使得不同的操作进行独立封装，仅保留基础接口供其他类使用，因此当项目需要添加新功能时，仅需要在现有基础上额外定义新的接口及其实现从而实现“热插拔”的效果。

3.1.3 里氏替换原则

里氏替换原则主要指父类使用的地方均能够使用子类进行替换。该原则在项目中的提现主要在于多态和泛类的使用。正如以下代码片段，我们在设计抽象控制器时，由于该段代码是对不同类中操作相同部分进行抽象的部分，因此采用了一个抽象对象 Thing 类型对数据中的泛型和特殊方法进行替换，这样在子控制器实现具体代码时，由于具体的子类如 Classes 类继承自 Thing 类，因此可以直接将 Classes 类直接赋值给 Thing 对象便可以实现。

```
……
modelAndView.addObject("pagesList", pagesList);
  // 存放 page，内有当前页数
  modelAndView.addObject("page", page);
  System.out.println("总条数" + thingIPage.getTotal());
  System.out.println("总页数" + thingIPage.getPages());
  // 存放总页数
  modelAndView.addObject("pages", (int) thingIPage.getPages());
  modelAndView.addObject("numberPages", thingIPage.getTotal());
  List<Thing> list = thingIPage.getRecords();
  System.out.println("list = " + list);
  modelAndView.addObject("list", list);

modelAndView.setViewName(this.toString()+"/"+this.toString()+"_list");
  return modelAndView;
```

因此在该实验中，更多的使用多态和泛型实现，采用父类定义，子类赋值的方式，很好地满足里氏替换原则。

3.1.4 依赖倒置原则

依赖倒置原则的主要思想是要依赖抽象，不要依赖具体类。高层模块不应该

依赖底层模块，二者都应该依赖于抽象。抽象不应该依赖于具体实现，具体实现应该依赖于抽象。

高层一般需要处理业务逻辑，因此逻辑也许会更分散，但是公共的处理需要抽象。因此底层的实现，实际上是由高层的需求所控制的，因此，高层依赖的接口，应该由高层进行抽象，底层负责实现。接口的所有权应该属于高层。

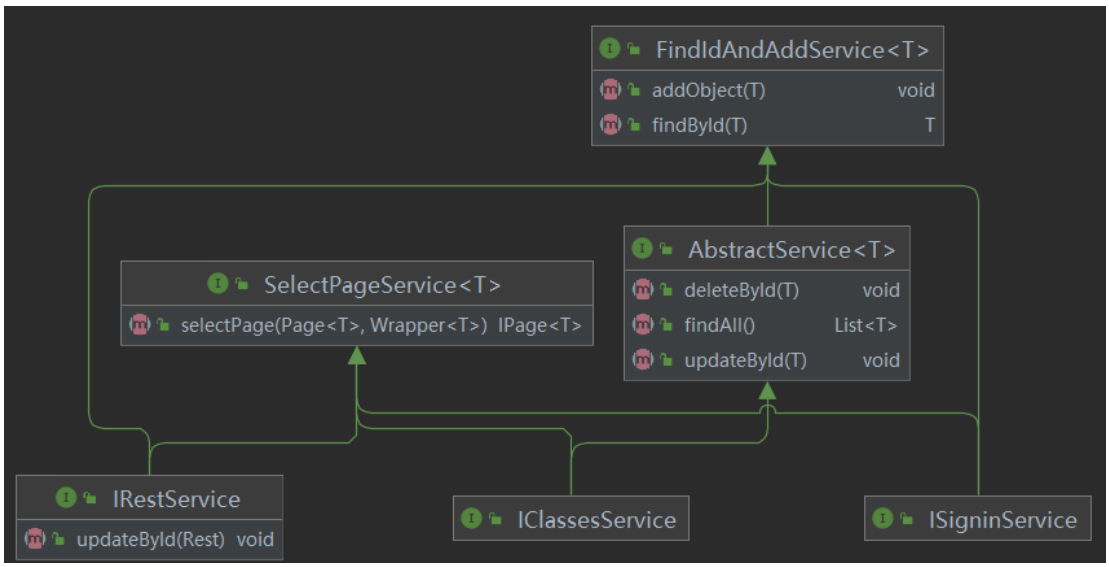
在该实验中，对 Controller 层和 Service 层都进行了抽象，具体的实现均面向接口，并且在传统的 MVC 模式中，通常在 Controller 控制器中会定义一个 Service 对象从而进行操作，但加入依赖倒转原则后，我们在抽象控制器中有以下代码：

```
private AbstractService abstractService;
private SelectPageService selectPageService;
```

由于我们已经对 Service 层进行了抽象，因此我们在抽象控制器 AbstractController 中调用是抽象的基本 Service 接口 abstractService 和 selectPageService，至于具体实现则交付于底层。一切控制权都有该抽象对象进行控制。

3.1.5 接口隔离原则

在之前已经讨论过在该实验中的 Service 层的设计，本层的设计是单一职责原则和接口隔离原则的提现。接口隔离原则主要指不应该强迫客户依赖他们不用的方法，在本次实验设计中，除去抽象 Service 类外一共包括 9 个类，其中 ISigninService 和 IRestService 接口仅仅包括了三个功能，除去在单一职责原则中抽象出的 SelectPageService 接口外，仅依赖最小接口，如下如所示：



如图所示，除了这两个接口直接调用了最小接口 FindIdAndAddService 以外，其他接口继承的最小接口是 AbstractService 接口，该接口继承自 FindIdAndAddService 接口并在此基础上增添了新的共同方法以供后期实例接口使用。

3.1.6 最少知识原则（迪米特原则）

该原则主要指的是一个类的通信只和自己的朋友对话，从而减少对象之间的交互，松散类之间的耦合，降低类之间的相互依赖。一般在实验中可以当作朋友的对象：

i、当前对象本身。

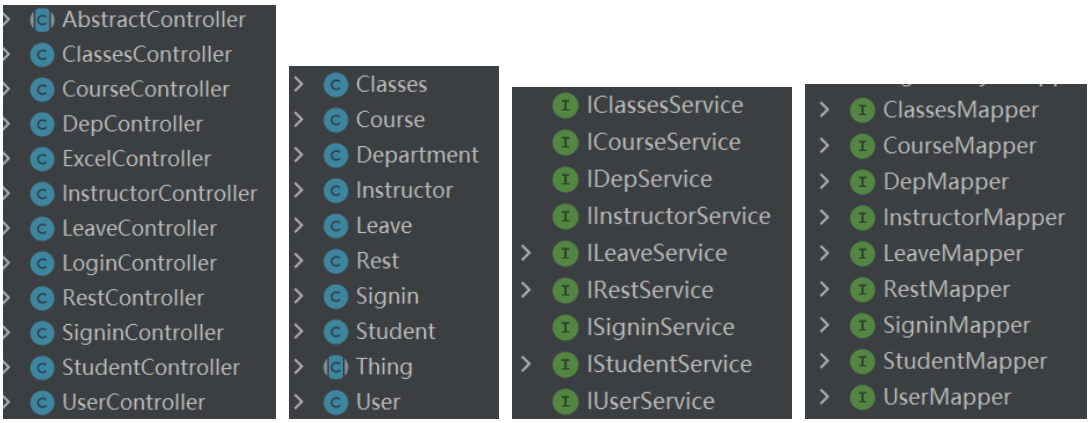
ii、通过参数传递进来的对象

iii、当前对象所创建的对象

iv、当前对象的实例变量所引用的对象

v、方法内所创建或实例化的对象。

因此再本实验中，我们为数据库中的每一张表都分别创建了 Entity 层、Service 层、Controlller 层和 Mapper 层，目的是隔离开不同对象之间的通信，仅保证一个对象仅和其朋友对话，从而实现迪米特法则以松散类之间的耦合程度。



3.2 设计模式

设计原则的指导思想往往是高度概括性和原则性，然而设计模式往往只是其中一种实现方式，而并非唯一的实现方式。通常在项目的开发过程中往往不仅仅采用一种设计原则，而是一些设计原则的整合。具体场景下突出和体现的设计原则也不尽相同。项目中合理地运用设计模式可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式都描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案。

为了重用代码、让代码更容易被他人理解、保证代码可靠性。在本次实验中或多或少地使用了很多设计模式，但是其中一部分更多的是借鉴其思想，实现并非规范，其中较为规范的主要包括了四个设计模式：装饰者模式、责任链模式、单例模式和工厂模式。

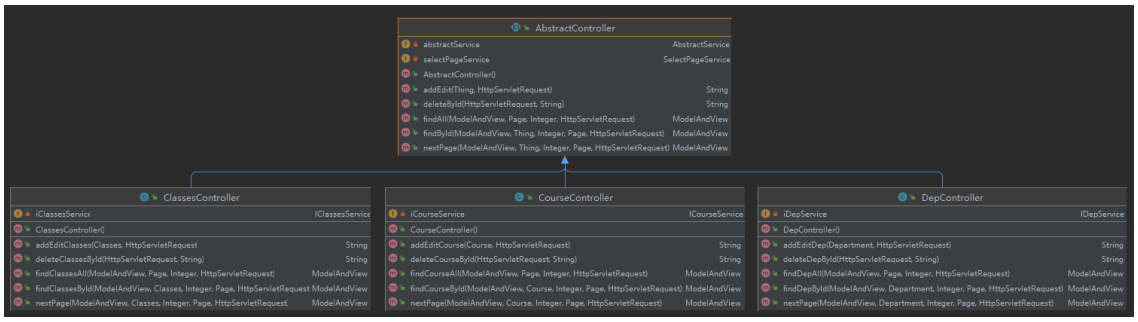
3.2.1 装饰者模式

装饰者模式作为结构型的设计模式，在项目的实际应用中更多的是提功了我们构建文件和定义类方面的帮助。其主要定义如下：

> 装饰模式是在不必改变原类和使用继承的情况下，动态地扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰来包裹真实的对象

在本次考勤请假系统中，面对不同的用户，不管是考勤模块还是请假调休模块，都存在大量的表单诺列操作，这些操作不仅包括查询分页方法，也包括了对于数据库表单的增删改查操作。尤其是在 Controller 层中需要与用户界面进行交互，提供表单的展现方法，这些方法在系统的实现过程中往往是类似的甚至是相同的，因此通过装饰者模式，将这些共同操作进行提炼，这就可以保证这些重复代码块不必多次被书写和创建，一方面优化了代码的可读性，另一方面提高了代

码的安全性和简约性。

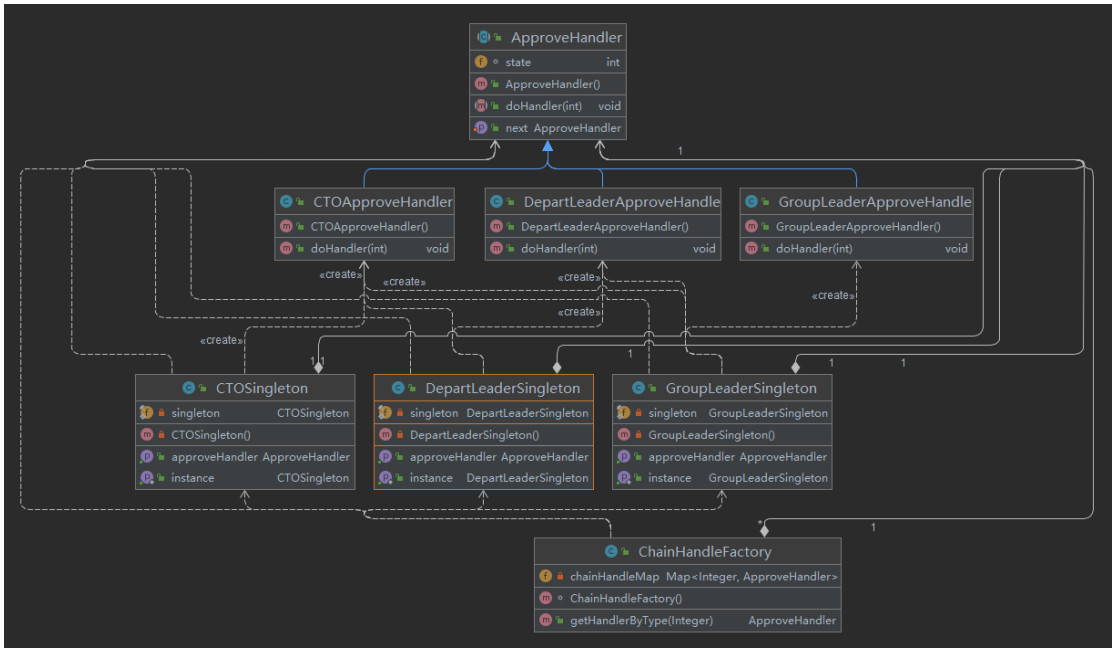


上图是本次实验项目中的部分控制器的 UML 图，我们将公共方法进行提炼，然后通过抽象对象和泛型进行改写，从而保证子类在继承这些关系的时候能够根据自身类型进行操作对象的微调。

需要注意的是，在该项目中，AbstractController 作为原始对象，提供原始接口；具体类的 Controller 如 ClassesController 便是作为具体的装饰者类。这里与通常的装饰者模式不同，并未设计单独的装饰者基类。由于在 Controller 的实现中，这些装饰者往往只是针对前端的表单数据进行特殊处理，因此，直接定义装饰者类，在这些类方法中继承来自原始类 AbstractController 的方法，并在其头尾增加特殊处理代码即可。

3.2.1 责任链模式+单例模式+工厂模式

本次实验的重点是处理好员工的请假调休信息，这些信息具有层层审批的环节，自然可以通过责任链模式实现。并且基于责任单一原则和基于接口编程的特性，这一部分的处理逻辑与整体的系统相对独立，因此我们在 Util 层中构建 LeaveList 包用于存放该功能，并且暴露出这一部分的接口供其它层使用从而减少该部分的耦合性。因此暴露接口这一功能主要通过工厂模式实现。这一部分的 UML 图如下所示：



（1）责任链模式

在本次实验中，当员工提供了请假调休申请时，需要根据请假天数依次交给项目组长、部门经理和总经理进行审批，这种层层递进的方式使用责任链模式可以很好的对应。责任链模式的定义主要如下：

> 责任链模式避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止

责任链模式主要由两部分构成：一是所有处理器的接口，第二个则是具体的处理器类。在本次实验中担任处理器接口的类是 ApproveHandler：

```
public abstract class ApproveHandler {
    protected ApproveHandler next;
    int state;
    /**
     * 指定下一个处理者
     */
    public void setNext(ApproveHandler approveHandler) {
        this.next = approveHandler;
    }
    /**
     * 处理审批请假天数
     */
    public abstract void doHandler(int size);
}
```

在这个类中只有两个函数，一个是处理审批的函数，另一个则是责任链模式中必须使用的函数用于指定当前处理器的下一级处理器。

有了处理器接口以后，在本次实验中担任具体处理器的便是项目组长、部门经理和总经理三个处理器：

```
public class GroupLeaderApproveHandler extends ApproveHandler {
    @Override
    public void doHandler(int size) {
        state = 1;
        if (size <= 1) {
            return;
        }
        next.doHandler(size);
    }
}
public class DepartLeaderApproveHandler extends ApproveHandler {
    @Override
    public void doHandler(int size) {
        state = 2;
        if (size <= 3) {
            return;
        }
        next.doHandler(size);
    }
}
public class CTOApproveHandler extends ApproveHandler {
    @Override
    public void doHandler(int size) {
        state = 3;
        if (size > 3) {
            return;
        }
    }
}
```

在定义完各层级处理器之后，仅需将这些处理器连接形成处理器链即可，具体的实现我们将通过单例模式进行辅助实现。

（2）单例模式

当我们需要使用责任链时，我们需要手动链接责任链，重复进行此操作将会变得很麻烦，在代码书写上也会造成冗余操作，因此我们使用单例模式辅助责任

链的构建。单例模式的定义如下：

> 某个类只有一个实例，且自行实例化并向整个系统提供此实例

单例模式已知有很多种写法，包括饿汉模式，懒汉模式等，在本次试验中，责任链是一直存在的，因此我们选择恶饿汉模式，在类加载时边形成单例，并且也无需使用 null 检查，加锁等复杂操作。

由于责任链的特性要求我们可以直接从不同层级开始逐层向上处理，因此我们需要为每一个处理器都构造一个单例，然后在该单例中形成从该级处理器形成的处理器链，此部分代码如下：

项目组长责任链单例：

```
public class GroupLeaderSingleton {
    private final static GroupLeaderSingleton singleton = new GroupLeaderSingleton();
    private ApproveHandler approveHandler;
    public ApproveHandler getApproveHandler(){
        return approveHandler;
    }
    private GroupLeaderSingleton() {
        approveHandler = new GroupLeaderApproveHandler();
        // 处理器初始化
        DepartLeaderApproveHandler departLeaderHandler = new
DepartLeaderApproveHandler();
        CTOApproveHandler ctoHandler = new CTOApproveHandler();
        approveHandler.setNext(departLeaderHandler);
        departLeaderHandler.setNext(ctoHandler);
    }
    public static GroupLeaderSingleton getInstance(){
        return singleton;
    }
}
```

项目经理责任链单例：

```
public class DepartLeaderSingleton {
    private final static DepartLeaderSingleton singleton = new DepartLeaderSingleton();
    private ApproveHandler approveHandler;
    public ApproveHandler getApproveHandler(){
        return approveHandler;
    }
```

```
        //接上页
         private DepartLeaderSingleton() {
              approveHandler = new DepartLeaderApproveHandler();
              // 处理器初始化
              CTOApproveHandler ctoHandler = new CTOApproveHandler();
              approveHandler.setNext(ctoHandler);
        }
        public static DepartLeaderSingleton getInstance(){
              return singleton;
        }
}
```

总经理责任链单例：

```
public class CTOSingleton {
        private final static CTOSingleton singleton = new CTOSingleton();
        private ApproveHandler approveHandler;
        public ApproveHandler getApproveHandler(){
              return approveHandler;
        }
        private CTOSingleton() {
              approveHandler = new CTOApproveHandler();
        }
        public static CTOSingleton getInstance(){
              return singleton;
        }
}
```

至此已经构造好了三条责任链，并且由于是单例模式，我们无需重复进行这三条责任链的构建，目前存在的问题就是如和隔离开以上操作与系统主要部分进行交互，因此引入工厂模式。

（3）工厂模式

由于在单例模式中已经创建好了责任链的实例，又不希望将责任链的实例在其他代码中进行创建，因此我们采用工厂模式提供给其他类工厂实例，将选择责任链的操作进行封装从而降低耦合性。工厂模式的定义如下：

```
定义一个用于创建对象的接口，让子类决定实例化哪个类
```

在本实验中，我们的责任链工厂主要负责两件事：一是将已经构造好的责任链单例进行工厂装载，二是暴露接口，根据用户传参自动选择责任链进行操作，该部分的代码相对简单：

```
import java.util.HashMap;
import java.util.Map;
public class ChainHandleFactory {
    private Map<Integer,ApproveHandler> chainHandleMap;
    ChainHandleFactory(){
        chainHandleMap = new HashMap<>();
        chainHandleMap.put(1,GroupLeaderSingleton.getInstance().getApproveHandler());
        chainHandleMap.put(2,DepartLeaderSingleton.getInstance().getApproveHandler());
        chainHandleMap.put(3,CTOSingleton.getInstance().getApproveHandler());
    }
    public ApproveHandler getHandlerByType(Integer type){
        return chainHandleMap.get(type);
    }
}
```

（4）组合效果及其优点

通过使用责任链模式、单例模式和工厂模式，我们将考勤系统请假调休模块的处理逻辑进行了独立封装和解耦。当其他类如 Controller 控制器类中需要调用责任链时，仅需要从责任链工厂中直接获取责任链即可，并且该责任链在全程只初始化一次，无需用户频繁创建，从而大大增加责任链带来的执行效率。