设计模式实验报告

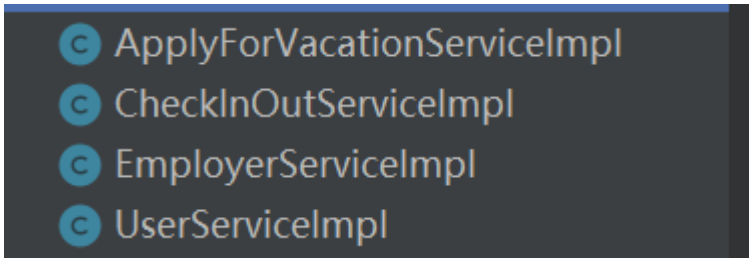
小组成员：朱亚乔 马晓春 闫浩然

设计原则

一、SRP 单一职责原则

单一职责原则表示一个类或者模块只负责完成一类职责或者功能，不要设计大而全而要设计粒度小，功能单一的类

本程序的 Servic 层跟据功能不同创建了四个不同的类



其中 ApplyForVacationService 主要实现员工请假相关功能

```
public interface ApplyForVacationService {

    /**
     * 员工申请假期，需要填写请假开始时间，请假总时长，理由，请假类型
     * @param empId
     * @param duringTime
     * @param reason
     * @param type
     * @return
     */
    2 usages  1 implementation  Juliet Jobs
    boolean applyForVacation(long empId , Date startTime , Integer duringTime , String reason , Integer type);

    /** 负责人原审批员工请假，根据申请ID号修改申请表中的状态 ...*/
    2 usages  1 implementation  Juliet Jobs
    boolean updateApplyState(long applyId , Integer state);

    /** 查询本人请假记录 ...*/
    2 usages  1 implementation  Juliet Jobs
    List<QueryApplyRecordResp> queryApplyByEmpId(long empId);

    /** 用于展示给项目经理，只需要审批state为1的审批记录并且申请天数小于3天 ...*/
    1 usage  1 implementation  Juliet Jobs
    List<QueryApplyInfoResp> queryApplyByStateForEventManager();

    /** 用于展示给总经理经理，只需要审批state为1的审批记录并且申请天数大于3天 ...*/
    1 usage  1 implementation  Juliet Jobs
    List<QueryApplyInfoResp> queryApplyByStateForBigManager();

    /** 根据用户id查询正在申请的记录 ...*/
    2 usages  1 implementation  Juliet Jobs
    List<QueryApplyInfoResp> queryApplyingById(long empId);

}
```

CheckInOutService 主要实现员工上下班打卡相关功能

```java
public interface CheckInOutService {

    /** 上下班打卡（签到），根据用户ID修改用户的状态，并添加打卡记录 ...*/
    2 usages  1 implementation  Juliet Jobs
    boolean signInOrOut(Long empId , int signType);

    /** 查询全部打卡记录 ...*/
    2 usages  1 implementation  Juliet Jobs
    List<QueryCheckRecordResp> queryAllSignRecord();

    /** 模糊查询：根据account或者type查询打卡记录 ...*/
    2 usages  1 implementation  Juliet Jobs
    List<QueryCheckRecordResp> querySignRecordByAccountOrType(String account , Integer type);

    /** 根据员工ID查询本人剩余假期时间 ...*/
    1 usage  1 implementation  Juliet Jobs
    LeftVacation queryLeftTimeByEmpId(long empId);

}
```

EmployerService 主要实现员工信息的增删改查

```java
public interface EmployerService {

    /** 查询全部员工信息 ...*/
    2 usages  1 implementation  Juliet Jobs
    List<QueryEmpInfoResp> queryAllEmployeeInfo();

    /** 根据id查询员工信息 ...*/
    2 usages  1 implementation  Juliet Jobs
    Employer queryEmpInfoById(long empId);

    /** 查询全部经理信息 ...*/
    2 usages  1 implementation  Juliet Jobs
    List<QueryEmpInfoResp> queryAllEmployerInfo();

    /** 登录，登录后返回该员工信息 ...*/
    2 usages  1 implementation  Juliet Jobs
    Employer login(String account , String password);

    /** 新增员工信息， ...*/
    2 usages  1 implementation  Juliet Jobs
    boolean insertEmpInfo(Employer employer);

    /** 查询公司全部人员的状态 ...*/
    1 usage  1 implementation  Juliet Jobs
    List<QueryEmpStateResp> queryEmpState();

    /** 查询当前用户的状态 ...*/
    1 usage  1 implementation  Juliet Jobs
    Integer queryEmpStateById(long empId);
```

二、OCP 开闭原则

开闭原则是指一个软件实体（如类、模块和函数）应该对扩展开放，对修改关闭。所谓的开闭，也正是对扩展和修改两个行为的一个原则。它强调的是用抽象构建框架，用实现扩展细节，可以提高软件系统的可复用性及可维护性。

三、LSP 里氏替换原则

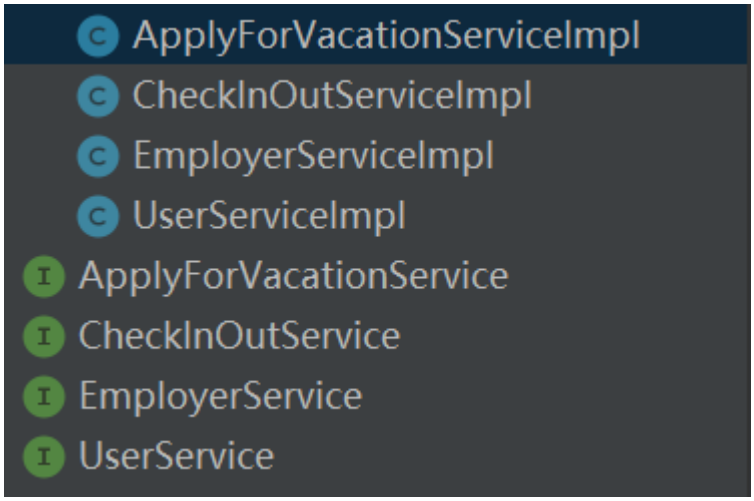
里氏替换原则是指类对象能够替换程序中父类对象出现的任何地方，并且保证原来程序的逻辑行为不变及正确性不被破坏。

四、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。概括的说就是：建立单一接口，不要建立臃肿庞大的接口。（接口尽量细化，同时接口中的方法尽量少。）

本项目的 service 层中根据功能不同定义了四个接口，然后由不同的实现类分别基层他们，保证了接口的功能尽量单一。



五、DIP 依赖反转原则

依赖反转原则是指：

A、高层次的模块不要依赖于低层次的模块，都应该依赖于抽象（接口）。

B、抽象（接口）不应该依赖于具体，而具体要依赖于抽象。

本项目使用 springboot 框架，而 springboot 中的依赖注入主要使用注解来实现

本项目中的 mapper 全部使用@Repository 注解注册到了 ioc 容器中，然后在 service 类中通过@Autowired 注解实现依赖注入。

```java
@Repository
public interface ApplyMapper extends BaseMapper<Apply> {

}
```

```java
@Repository
public interface LeftVacationMapper extends BaseMapper<LeftVacation> {

}
```

```java
@Repository
public interface EmployerMapper extends BaseMapper<Employer> {
```

```java
@Repository
public interface ApplyRecordMapper extends BaseMapper<ApplyRecord> {
```

```java
@Autowired
private ApplyMapper applyMapper;

10 usages
@Autowired
private LeftVacationMapper leftVacationMapper;

4 usages
@Autowired
private EmployerMapper employerMapper;

2 usages
@Autowired
private ApplyRecordMapper applyRecordMapper;
```

并且 service 类也通过@Service 注解注册到 ioc 容器中，在从 controller 中通过@Autowired 注解实现依赖注入

```java
@Service
public class EmployerServiceImpl implements EmployerService {
```

```java
7 usages
@Autowired
private EmployerService employerService;
```

设计模式

本项目采用了 SpringBoot 框架，下面分析 Spring 中所使用的设计模式

一、工厂方法

Spring 使用工厂模式，通过 BeanFactory 和 ApplicationContext 来创建对象。工厂模式是把创建对象的任务交给工厂，从而来降低类与类之间的耦合。Spring 最主要的两个特性就是 AOP 和 IOC，其中 IOC 就是控制反转，将对象的控制权转移给 Spring，并由 Spring 创建实例和管理各个实例之间的依赖关系，其中，对象的创建就是通过 BeanFactory 和 ApplicationContext 完成的

（1） Beanfactory 的源码为

```
1.  public interface BeanFactory {
2.      Object getBean(String name) throws BeansException;
3.
4.      <T> T getBean(String name, @Nullable Class<T> requiredType) th
    rows BeansException;
5.
6.      Object getBean(String name, Object... args) throws BeansExcept
    ion;
7.
8.      <T> T getBean(Class<T> requiredType) throws BeansException;
9.
10.     <T> T getBean(Class<T> requiredType, Object... args) throws Be
    ansException;
11. }
```

BeanFactory 是 Spring 里面最底层的接口，是 IoC 的核心，定义了 IoC 的基本功能，包含了各种 Bean 的定义、加载、实例化，依赖注入和生命周期管理。BeanFactroy 采用的是延迟加载形式来注入 Bean 的，只有在使用到某个 Bean 时(调用 getBean())，才对该 Bean 进行加载实例化。这样，我们就不能提前发现一些存在的 Spring 的配置问题。

（2）ApplicationContext 接口作为 BeanFactory 的子类，除了提供 BeanFactory 所具有的功能外，还扩展了其他更完整功能，对于 Bean 创建，ApplicationContext 在容器启动时，一次性创建了所有的 Bean。其源码为

```
1.  public interface ApplicationContext extends EnvironmentCapable, Lista
    bleBeanFactory, HierarchicalBeanFactory,
2.  MessageSource, ApplicationEventPublisher, ResourcePatternResolver {
3.      @Nullable
4.      String getId();
5.
6.      String getApplicationName();
7.
8.      String getDisplayName();
9.
10.     long getStartupDate();
11.
12.     @Nullable
13.     ApplicationContext getParent();
14.
15.     AutowireCapableBeanFactory getAutowireCapableBeanFactory() throws
     IllegalStateException;
16. }
```

二、单例模式:

在 Spring 中的 Bean 默认的作用域就是 singleton 单例的。单例模式的好处在于对一些重量级的对象，省略了重复创建对象花费的时间，减少了系统的开销，第二点是使

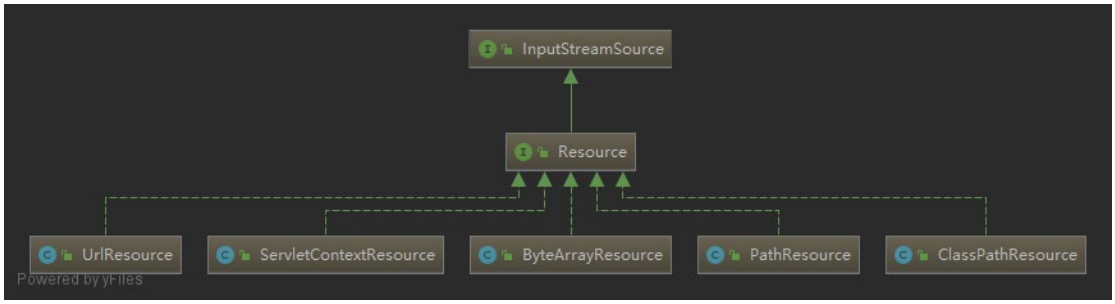
用单例可以减少 new 操作的次数，回收内存的压力。

对于单例 bean 的创建方式，主要看 DefaultSingletonBeanRegistry 的 getSingleton() 方法：

```
1.  public class DefaultSingletonBeanRegistry extends SimpleAliasRegistry
     implements SingletonBeanRegistry {
2.      /** 保存单例 Objects 的缓存集合 ConcurrentHashMap，key：beanName -
    -> value：bean 实例 */
3.      private final Map<String, Object> singletonObjects = new Concurre
    ntHashMap<>(256);
4.
5.      public Object getSingleton(String beanName, ObjectFactory<?> sing
    letonFactory) {
6.          Assert.notNull(beanName, "Bean name must not be null");
7.          synchronized (this.singletonObjects) {
8.              //检查缓存中是否有实例，如果缓存中有实例，直接返回
9.              Object singletonObject = this.singletonObjects.get(beanNa
    me);
10.             if (singletonObject == null) {
11.                 //省略...
12.                 try {
13.                     //通过 singletonFactory 获取单例
14.                     singletonObject = singletonFactory.getObject();
15.                     newSingleton = true;
16.                 }
17.                 if (newSingleton) {
18.                     addSingleton(beanName, singletonObject);
19.                 }
20.             }
21.             //返回实例
22.             return singletonObject;
23.         }
24.     }
25.
26.     protected void addSingleton(String beanName, Object singletonObje
    ct) {
27.       synchronized (this.singletonObjects) {
28.         this.singletonObjects.put(beanName, singletonObject);
29.         this.singletonFactories.remove(beanName);
30.         this.earlySingletonObjects.remove(beanName);
31.         this.registeredSingletons.add(beanName);
32.       }
33.     }
34. }
```

三、策略模式

策略模式，简单来说就是封装好一组策略算法，外部客户端根据不同的条件选择不同的策略算法解决问题。比如在 Spring 的 Resource 类，针对不同的资源，Spring 定义了不同的 Resource 类的实现类，以此实现不同的访问方式。



UrlResource：访问网络资源的实现类。

ServletContextResource：访问相对于 ServletContext 路径里的资源的实现类。

ByteArrayResource：访问字节数组资源的实现类。

PathResource：访问文件路径资源的实现类。

ClassPathResource：访问类加载路径里资源的实现类。

四、代理模式

AOP 是 Spring 的一个核心特性(面向切面编程)，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，减少系统中的重复代码，降低了模块间的耦合度，提高系统的可维护性。可用于权限认证、日志、事务处理。

Spring AOP 实现的关键在于动态代理，主要有两种方式，JDK 动态代理和 CGLIB 动态代理

DefaultAopProxyFactory 的 createAopProxy()方法，Spring 通过此方法创建动态代理类：

```
1.  public class DefaultAopProxyFactory implements AopProxyFactory, Seria
    lizable {
2.      @Override
3.      public AopProxy createAopProxy(AdvisedSupport config) throws AopC
    onfigException {
4.          if (config.isOptimize() || config.isProxyTargetClass() || has
    NoUserSuppliedProxyInterfaces(config)) {
5.              Class<?> targetClass = config.getTargetClass();
6.              if (targetClass == null) {
7.                  throw new AopConfigException("TargetSource cannot det
    ermine target class: " + "Either an interface or a target is required
     for proxy creation.");
8.              }
9.              if (targetClass.isInterface() || Proxy.isProxyClass(targe
    tClass)) {
10.                 return new JdkDynamicAopProxy(config);
11.             }
12.             return new ObjenesisCglibAopProxy(config);
```

```
        }
        else {
            return new JdkDynamicAopProxy(config);
        }
    }
}
```

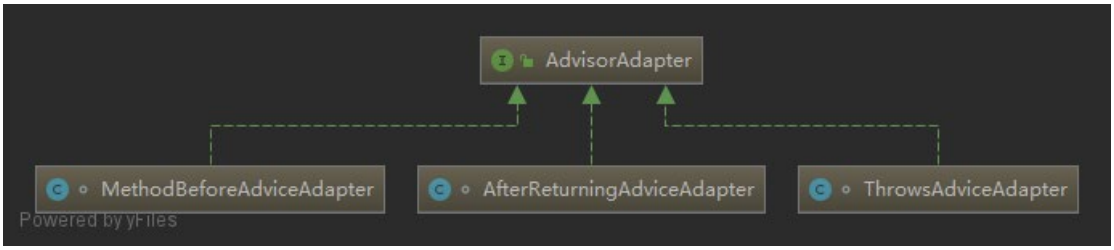
五、适配器模式与责任链模式：

适配器模式能使接口不兼容的对象能够相互合作，将一个类的接口，转换成客户期望的另外一个接口。

在 SpringAOP 中有一个很重要的功能就是使用的 Advice（通知） 来增强被代理类的功能，Advice 主要有 MethodBeforeAdvice、AfterReturningAdvice、ThrowsAdvice 这几种。每个 Advice 都有对应的拦截器，如下所示

Advice(通知)	Interceptor(拦截器)
MethodBeforeAdvice	MethodBeforeAdviceInterceptor
AfterReturningAdvice	AfterReturningAdviceInterceptor
ThrowsAdvice	ThrowsAdviceInterceptor

Spring 需要将每个 Advice 都封装成对应的拦截器类型返回给容器，所以需要使用适配器模式对 Advice 进行转换。对应的就有三个适配器。



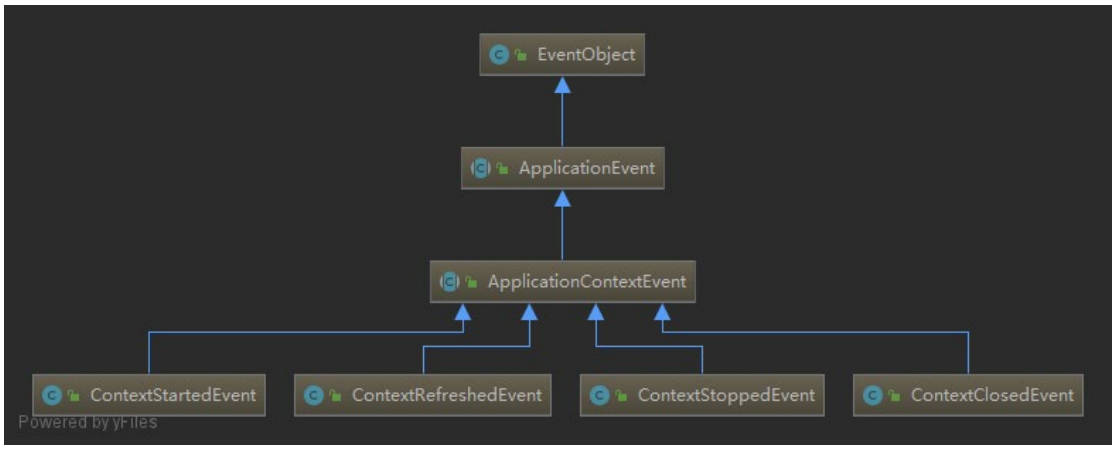
六、观察者模式

观察者模式是一种对象行为型模式，当一个对象发生变化时，这个对象所依赖的对象也会做出反应。

Spring 事件驱动模型就是观察者模式很经典的一个应用。

在 Spring 事件驱动模型中，首先有事件角色 ApplicationEvent，这是一个抽象类，抽象类下有四个实现类代表四种事件。

ContextStartedEvent：ApplicationContext 启动后触发的事件。
ContextStoppedEvent：ApplicationContext 停止后触发的事件。
ContextRefreshedEvent：ApplicationContext 初始化或刷新完成后触发的事件。
ContextClosedEvent：ApplicationContext 关闭后触发的事件。


EventObject
ApplicationEvent
ApplicationContextEvent
ContextStartedEvent
ContextRefreshedEvent
ContextStoppedEvent
ContextClosedEvent
Powered by yFiles