课程目标

1、掌握委派模式，精简程序逻辑，提升代码的可读性。

2、学会用模板方法模式梳理使用工作中流程标准化的业务场景。

内容定位

1、希望通过对委派模式的学习，让自己写出更加优雅的代码的人群。

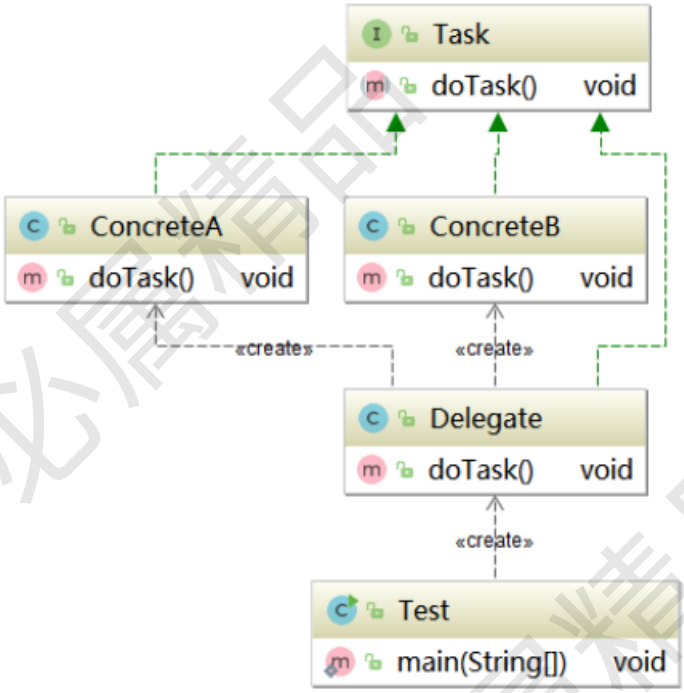
2、深刻了解模板方法模式的应用场景。

委派模式

委派模式（Delegate Pattern）又叫委托模式，是一种面向对象的设计模式，允许对象组合实现与继承相同的代码重用。它的基本作用就是负责任务的调用和分配任务，是一种特殊的静态代理，可以理解为全权代理，但是代理模式注重过程，而委派模式注重结果。委派模式属于行为型模式，不属于 GOF 23 种设计模式中。

委派模式的应用场景

委派模式在 Spring 中应用非常多，大家常用的 DispatcherServlet 其实就是用到了委派模式。先来看一下类图：



从类图中我们可以看到，委派模式有三个参与角色：

抽象任务角色（Task）：定义一个抽象接口，它有若干实现类。。

委派者角色（Delegate）：负责在各个具体角色实例之间做出决策，并判断并调用具体实现的方法。

具体任务角色（Concrete）真正执行任务的角色。

现实生活中也常有委派的场景发生，例如：老板（Boss）给项目经理（Leader）下达任务，项目经理会根据实际情况给每个员工派发工作任务，待员工把工作任务完成之后，再由项目经理汇报工作进度和结果给老板。

老板给员工下达任务

授权委托书

委派模式在业务场景中的应用

我们用代码来模拟下这个业务场景，

创建 IEmployee 员工接口：

```
public interface IEmployee {
    void doing(String task);
}
```

创建员工 EmployeeA 类：

```
public class EmployeeA implements IEmployee {
    protected String goodAt = "编程";
    public void doing(String task) {
        System.out.println("我是员工 A，我擅长" + goodAt + ",现在开始做" +task + "工作");
    }
}
```

创建员工 EmployeeB 类：

```
public class EmployeeB implements IEmployee {
    protected String goodAt = "平面设计";
    public void doing(String task) {
        System.out.println("我是员工 B，我擅长" + goodAt + ",现在开始做" +task + "工作");
    }
}
```

创建项目经理 Leader 类：

```
public class Leader implements IEmployee {

    private Map<String,IEmployee> employee = new HashMap<String,IEmployee>();

    public Leader(){
        employee.put("爬虫",new EmployeeA());
        employee.put("海报图",new EmployeeB());
    }

    public void doing(String task) {
        if(!employee.containsKey(task)){
            System.out.println("这个任务" +task + "超出我的能力范围");
            return;
        }
        employee.get(task).doing(task);
    }
}
```

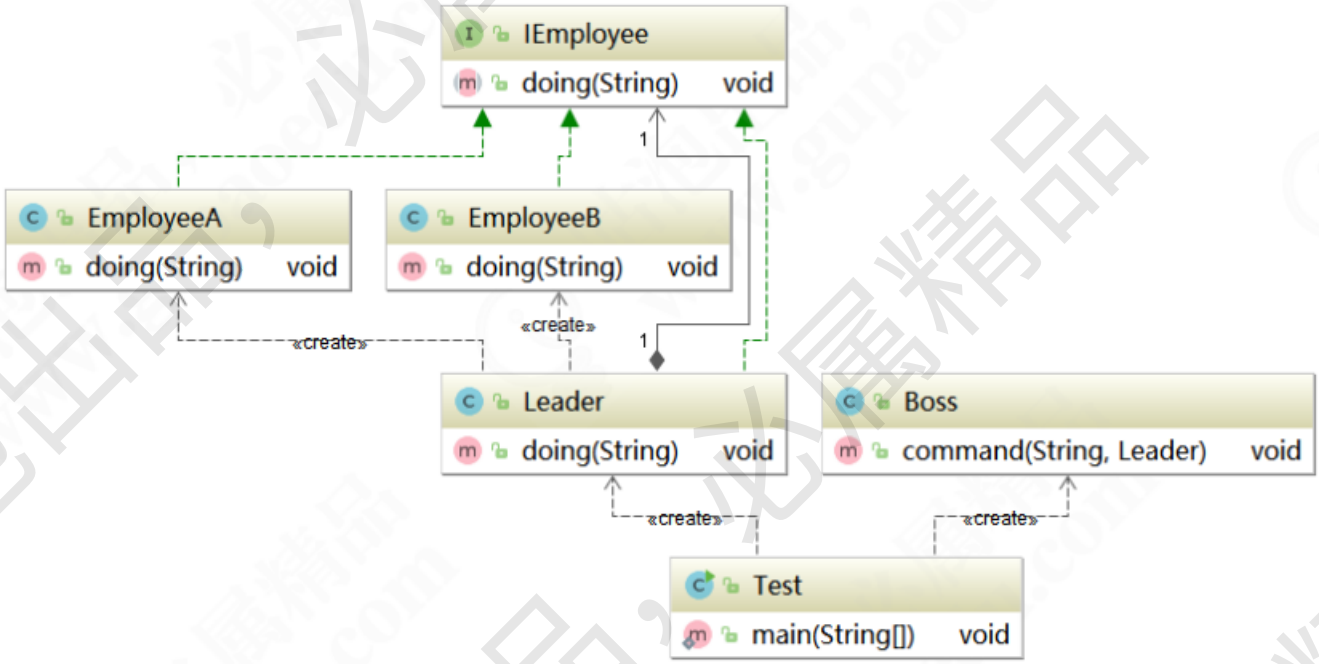
创建 Boss 类下达命令：

```
public class Boss {
    public void command(String task,Leader leader){
        leader.doing(task);
    }
}
```

测试代码：

```
public class Test {
    public static void main(String[] args) {
        new Boss().command("海报图",new Leader());
        new Boss().command("爬虫",new Leader());
        new Boss().command("卖手机",new Leader());
    }
}
```

通过上面的代码，生动地还原了项目经理分配工作的业务场景，也是委派模式的生动体现。下面来看一下类图：



委派模式在源码中的体现

JDK 中有一个典型的委派，众所周知 JVM 在加载类是用的双亲委派模型，这又是什么呢？一个类加载器在加载类时，先把这个请求委派给自己的父类加载器去执行，如果父类加载器还存在父类加载器，就继续向上委派，直到顶层的启动类加载器。如果父类加载器能够完成类加载，就成功返回，如果父类加载器无法完成加载，那么子加载器才会尝试自己去加载。从定义中可以看到双亲加载模型一个类加载器加载类时，首先不是自己加载，而是委派给父加载器。下面我们来看看 loadClass()方法的源码，此方法在 ClassLoader 中。在这个类里就定义了一个双亲，用于下面的类加载。

```
public abstract class ClassLoader {
    ...
    private final ClassLoader parent;
    ...
    protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException
```

```
    {
        synchronized (getClassLoadingLock(name)) {
            // First, check if the class has already been loaded
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }

                if (c == null) {
                    long t1 = System.nanoTime();
                    c = findClass(name);

                    sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
                    sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                    sun.misc.PerfCounter.getFindClasses().increment();
                }
            }
            if (resolve) {
                resolveClass(c);
            }
            return c;
        }
    }
     ...
}
```

同样在 Method 类里我们常用代理执行方法 invoke()也存在类似的机制。

```
public Object invoke(Object obj, Object... args)
        throws IllegalAccessException, IllegalArgumentException,
           InvocationTargetException
    {
        if (!override) {
            if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
                Class<?> caller = Reflection.getCallerClass();
                checkAccess(caller, clazz, obj, modifiers);
            }
        }
        MethodAccessor ma = methodAccessor;             // read volatile
        if (ma == null) {
            ma = acquireMethodAccessor();
        }
        return ma.invoke(obj, args);
    }
```

看完代码，相信小伙伴们对委派和代理区别搞清楚了吧。

下面来看一下委派模式在 Spring 中的应用，在 Spring IoC 模块中的 DefaultBeanDefinitionDocumentReader 类，在调用 doRegisterBeanDefinitions() 方法时即

BeanDefinition 进行注册的过程中，会设置 BeanDefinitionParserDelegate 类型的 Delegate 对象传给 this.delegate，并将这个对象作为一个参数传给：parseBeanDefinitions(root, this.delegate)中，然后主要的解析的工作就是通过 delegate 作为主要角色来完成的，可以看到下方代码：

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
   //判断节点是否属于同一命名空间，是则执行后续的解析
   if (delegate.isDefaultNamespace(root)) {
      NodeList nl = root.getChildNodes();
      for (int i = 0; i < nl.getLength(); i++) {
         Node node = nl.item(i);
         if (node instanceof Element) {
            Element ele = (Element) node;
            if (delegate.isDefaultNamespace(ele)) {
               parseDefaultElement(ele, delegate);
            }
            else {
               //注解定义的 Context 的 nameSpace 进入到这个分支中
               delegate.parseCustomElement(ele);
            }
         }
      }
   }
   else {
      delegate.parseCustomElement(root);
   }
}
```

其中最终能够走到 bean 注册部分的是，会进入到 parseDefaultElement(ele, delegate)中，然后针对不同的节点类型，针对 bean 的节点进行真正的注册操作，而在这个过程中，delegate 会对 element 进行 parseBeanDefinitionElement，得到了一个 BeanDefinitionHolder 类型的对象，之后通过这个对象完成真正的注册到 Factory 的操作。

下面我们再来还原一下 SpringMVC 的 DispatcherServlet 是如何实现委派模式的。创建业务类 MemberController：

```
/**
 * Created by Tom.
 */
public class MemberController {

    public void getMemberById(String mid){

    }
}
```

OrderController 类：

```
/**
 * Created by Tom.
 */
```

```
public class OrderController {

    public void getOrderById(String mid){

    }

}
```

SystemController 类:

```
/**
 * Created by Tom.
 */
public class SystemController {

    public void logout(){

    }
}
```

创建 DispatcherServlet 类:

```
public class DispatcherServlet extends HttpServlet {

    private Map<String,Method> handlerMapping = new HashMap<String,Method>();

    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        doDispatch(req,resp);
    }

    private void doDispatch(HttpServletRequest req, HttpServletResponse resp) {
        String url = req.getRequestURI();
        Method method = handlerMapping.get(url);
//        method.invoke();
    }

    @Override
    public void init() throws ServletException {
        try {
            handlerMapping.put("/web/getMemeberById.json",    MemberController.class.getMethod("getMemberById",    new
Class[]{String.class}));
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

配置 web.xml 文件:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:javaee="http://java.sun.com/xml/ns/javaee"
   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
   version="2.4">
   <display-name>Gupao Web Application</display-name>


   <servlet>
      <servlet-name>delegateServlet</servlet-name>
      <servlet-class>com.gupaoedu.vip.pattern.delegate.mvc.DispatcherServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
```

```
  </servlet>

  <servlet-mapping>
     <servlet-name>delegateServlet</servlet-name>
     <url-pattern>/*</url-pattern>
  </servlet-mapping>

</web-app>
```

一个完整的委派模式就实现出来了。当然，在 Spring 中运用到委派模式不仅于此，还有很多。小伙伴们可以通过命名就可以识别。在 Spring 源码中，只要以 Delegate 结尾的都是实现了委派模式。例如：BeanDefinitionParserDelegate 根据不同类型委派不同的逻辑解析 BeanDefinition。

委派模式的优缺点

优点：

通过任务委派能够将一个大型的任务细化，然后通过统一管理这些子任务的完成情况实现任务的跟进，能够加快任务执行的效率。

缺点：

任务委派方式需要根据任务的复杂程度进行不同的改变，在任务比较复杂的情况下可能需要进行多重委派，容易造成紊乱。

模板方法模式

模板方法模式（Template Method Pattern）又叫模板模式，是指定义一个操作中的算法的框架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤，属于行为型设计模式。

```
原文：Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
```

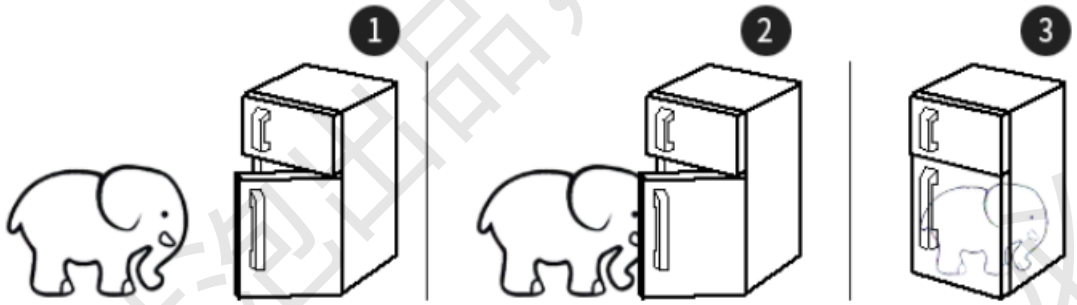
模板方法模式实际上是封装了一个固定流程，该流程由几个步骤组成，具体步骤可以由子类进行不同实现，从而让固定的流程产生不同的结果。它非常简单，其实就是类的继承机制，但它却是一个应用

非常广泛的模式。模板方法模式的本质是抽象封装流程，具体进行实现。

模板方法模式的应用场景

当完成一个操作具有固定的流程时，由抽象固定流程步骤，具体步骤交给子类进行具体实现（固定的流程，不同的实现）。

在我们日常生活中模板方法模式也很常见。比如我们平时办理入职流程填写入职登记表 → 打印简历 → 复印学历 → 复印身份证 → 签订劳动合同 → 建立花名册 → 办理工牌 → 安排工位等；再比如，我平时在家里炒菜：洗锅 → 点火 → 热锅 → 上油 → 下原料 → 翻炒 → 放调料 → 出锅；再比如有个小品，赵本山问宋丹丹:“如何把大象放进冰箱？”宋丹丹回答：“第一步：打开冰箱门，第二步：把大象塞进冰箱，第三步：关闭冰箱门”。赵本山再问：“怎么把长劲鹿放进冰箱？”宋丹丹答：“第一步：打开冰箱门，第二步：把大象拿出来，第三步：把长劲鹿塞进去，第四步：关闭冰箱门”（如下图所示），这些都是模板方法模式的体现。

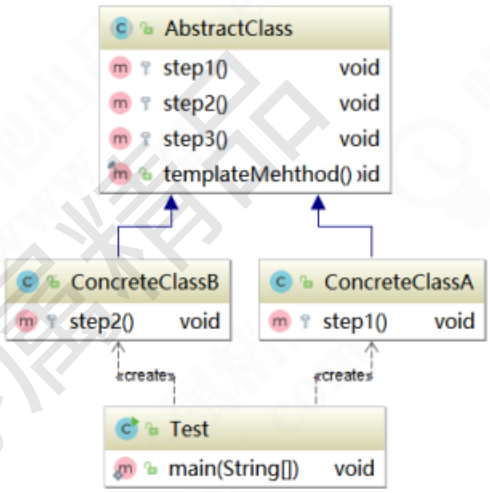


模板方法模式适用于以下应用场景：

1、一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。

2、各子类中公共的行为被提取出来并集中到一个公共的父类中，从而避免代码重复。

首先来看下模板方法模式的通用 UML 类图：



从 UML 类图中，我们可以看到，模板方法模式主要包含两种角色：

抽象模板（AbstractClass）：抽象模板类，定义了一套算法框架/流程；

具体实现（ConcreteClass）：具体实现类，对算法框架/流程的某些步骤进行了实现。

模板方法模式中的钩子方法

我们还是以咕泡学院的课程创建流程为例：发布预习资料 → 制作课件 PPT → 在线直播 → 提交课堂笔记 → 提交源码 → 布置作业 → 检查作业。首先我们来创建 AbastractCourse 抽象类：

```
public abstract class AbastractCourse {

    public final void createCourse(){
        //1、发布预习资料
        postPreResoucse();

        //2、制作课件
        createPPT();

        //3、直播授课
        liveVideo();

        //4、上传课后资料
        postResource();

        //5、布置作业
        postHomework();

        if(needCheckHomework()){
            checkHomework();
        }
    }

    protected abstract void checkHomework();
```

```
    //钩子方法
    protected boolean needCheckHomework(){return  false;}

    protected void postHomework(){
        System.out.println("布置作业");
    }

    protected void postResource(){
        System.out.println("上传课后资料");
    }

    protected void liveVideo(){
        System.out.println("直播授课");
    }

    protected void createPPT(){
        System.out.println("制作课件");
    }

    protected void postPreResoucse(){
        System.out.println("发布预习资料");
    }

}
```

上面的代码中有个钩子方法可能有些小伙伴还不是太理解，在此我稍作解释。设计钩子方法的主要目的是用来干预执行流程，使得我们控制行为流程更加灵活，更符合实际业务的需求。钩子方法的返回值一般为适合条件分支语句的返回值（如 boolean、int 等）。小伙伴们可以根据自己的业务场景来决定是否需要使用钩子方法。接下来创建 JavaCourse 类：

```
public class JavaCourse extends AbastractCourse {
    private boolean needCheckHomework = false;

    public void setNeedCheckHomework(boolean needCheckHomework) {
        this.needCheckHomework = needCheckHomework;
    }

    @Override
    protected boolean needCheckHomework() {
        return this.needCheckHomework;
    }

    protected void checkHomework() {
        System.out.println("检查 Java 作业");
    }
}
```

创建 PythonCourse 类：

```
public class PythonCourse extends AbastractCourse {
    protected void checkHomework() {
        System.out.println("检查 Python 作业");
    }
}
```

客户端测试代码：

```
    public static void main(String[] args) {
        System.out.println("=========架构师课程=========");
        JavaCourse java = new JavaCourse();
        java.setNeedCheckHomework(false);
        java.createCourse();


        System.out.println("=========Python课程=========");
        PythonCourse python = new PythonCourse();
        python.createCourse();
    }
```

通过这样一个案例，相信下伙伴们对模板方法模式有了一个基本的印象。为了加深理解，下面我们来结合一个常见的业务场景。

利用模板方法模式重构 JDBC 操作业务场景

创建一个模板类 JdbcTemplate，封装所有的 JDBC 操作。以查询为例，每次查询的表不同，返回的数据结构也就不一样。我们针对不同的数据，都要封装成不同的实体对象。而每个实体封装的逻辑都是不一样的，但封装前和封装后的处理流程是不变的，因此，我们可以使用模板方法模式来设计这样的业务场景。先创建约束 ORM 逻辑的接口 RowMapper：

```
/**
 * ORM映射定制化的接口
 * Created by Tom.
 */
public interface RowMapper<T> {
    T mapRow(ResultSet rs,int rowNum) throws Exception;
}
```

在创建封装了所有处理流程的抽象类 JdbcTemplate：

```
public abstract class JdbcTemplate {
    private DataSource dataSource;

    public JdbcTemplate(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public final List<?> executeQuery(String sql,RowMapper<?> rowMapper,Object[] values){
        try {
            //1、获取连接
            Connection conn = this.getConnection();
            //2、创建语句集
            PreparedStatement pstm = this.createPrepareStatement(conn,sql);
            //3、执行语句集
            ResultSet rs = this.executeQuery(pstm,values);
            //4、处理结果集
            List<?> result = this.parseResultSet(rs,rowMapper);
            //5、关闭结果集
            rs.close();
            //6、关闭语句集
            pstm.close();
```

```
            //7、关闭连接
            conn.close();
            return result;
        }catch (Exception e){
            e.printStackTrace();
        }
        return null;
    }

    private List<?> parseResultSet(ResultSet rs, RowMapper<?> rowMapper) throws Exception {
        List<Object> result = new ArrayList<Object>();
        int rowNum = 0;
        while (rs.next()){
            result.add(rowMapper.mapRow(rs,rowNum++));
        }
        return result;
    }


    private ResultSet executeQuery(PreparedStatement pstm, Object[] values) throws SQLException {
        for (int i = 0; i < values.length; i++) {
            pstm.setObject(i,values[i]);
        }
        return pstm.executeQuery();
    }

    private PreparedStatement createPrepareStatement(Connection conn, String sql) throws SQLException {
        return conn.prepareStatement(sql);
    }

    private Connection getConnection() throws SQLException {
        return this.dataSource.getConnection();
    }
}
```

创建实体对象 Member 类：

```
public class Member {

    private String username;
    private String password;
    private String nickname;
    private int age;
    private String addr;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getNickname() {
        return nickname;
```

```
    }

    public void setNickname(String nickname) {
        this.nickname = nickname;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getAddr() {
        return addr;
    }

    public void setAddr(String addr) {
        this.addr = addr;
    }
}
```

创建数据库操作类 MemberDao：

```
public class MemberDao extends JdbcTemplate {
    public MemberDao(DataSource dataSource) {
        super(dataSource);
    }

    public List<?> selectAll(){
        String sql = "select * from t_member";
        return super.executeQuery(sql, new RowMapper<Member>() {
            public Member mapRow(ResultSet rs, int rowNum) throws Exception {
                Member member = new Member();
                //字段过多，原型模式
                member.setUsername(rs.getString("username"));
                member.setPassword(rs.getString("password"));
                member.setAge(rs.getInt("age"));
                member.setAddr(rs.getString("addr"));
                return member;
            }
        },null);
    }
}
```

客户端测试代码：

```
    public static void main(String[] args) {
        MemberDao memberDao = new MemberDao(null);
        List<?> result = memberDao.selectAll();
        System.out.println(result);
    }
```

希望通过这两个案例的业务场景分析，能够帮助小伙们对模板方法模式有更深的理解。

模板方法模式在源码中的体现

先来看 JDK 中的 AbstractList，来看代码：

```
package java.util;
```

```
public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {
    ...
    abstract public E get(int index);
    ...
}
```

我们看到 get()是一个抽象方法，那么它的逻辑就是交给子类来实现，我们大家所熟知的 ArrayList 就是 AbstractList 的子类。同理，有 AbstractList 就有 AbstractSet 和 AbstractMap，有兴趣的小伙伴可以去看看这些的源码实现。还有一个每天都在用的 HttpServlet，有三个方法 service()和 doGet()、doPost()方法，都是模板方法的抽象实现。

在 MyBatis 框架也有一些经典的应用，我们来一下 BaseExecutor 类，它是一个基础的 SQL 执行类，实现了大部分的 SQL 执行逻辑，然后把几个方法交给子类定制化完成，源码如下：

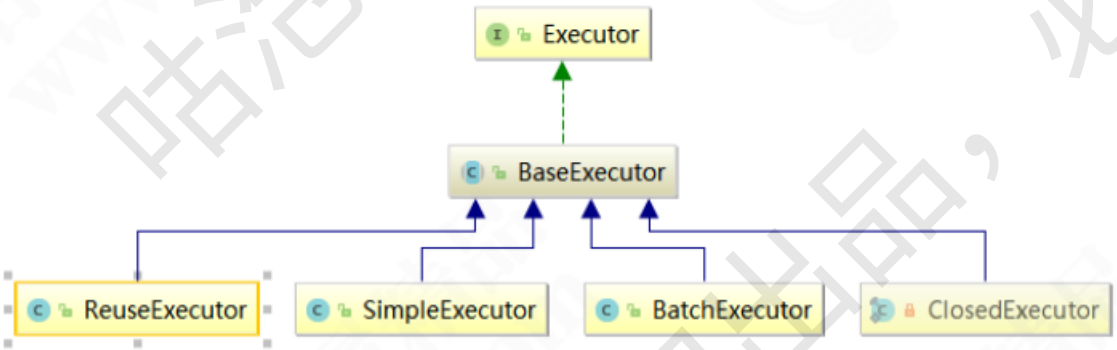
```
...
public abstract class BaseExecutor implements Executor {
    ...
    protected abstract int doUpdate(MappedStatement var1, Object var2) throws SQLException;

    protected abstract List<BatchResult> doFlushStatements(boolean var1) throws SQLException;

    protected abstract <E> List<E> doQuery(MappedStatement var1, Object var2, RowBounds var3, ResultHandler var4, BoundSql
var5) throws SQLException;

    protected abstract <E> Cursor<E> doQueryCursor(MappedStatement var1, Object var2, RowBounds var3, BoundSql var4)
throws SQLException;
...
}
```

如 doUpdate、doFlushStatements、doQuery、doQueryCursor 这几个方法就是交由子类来实现，那么 BaseExecutor 有哪些子类呢？我们来看一下它的类图：



我们一起来看一下 SimpleExecutor 的 doUpdate 实现：

```
public int doUpdate(MappedStatement ms, Object parameter) throws SQLException {
    Statement stmt = null;

    int var6;
```

```
    try {
        Configuration configuration = ms.getConfiguration();
        StatementHandler handler = configuration.newStatementHandler(this, ms, parameter, RowBounds.DEFAULT,
(ResultHandler)null, (BoundSql)null);
        stmt = this.prepareStatement(handler, ms.getStatementLog());
        var6 = handler.update(stmt);
    } finally {
        this.closeStatement(stmt);
    }

    return var6;
}
```

再来对比一下 BatchExecutor 的 doUpate 实现：

```
public int doUpdate(MappedStatement ms, Object parameterObject) throws SQLException {
    Configuration configuration = ms.getConfiguration();
    StatementHandler handler = configuration.newStatementHandler(this, ms, parameterObject, RowBounds.DEFAULT,
(ResultHandler)null, (BoundSql)null);
    BoundSql boundSql = handler.getBoundSql();
    String sql = boundSql.getSql();
    Statement stmt;
    if(sql.equals(this.currentSql) && ms.equals(this.currentStatement)) {
        int last = this.statementList.size() - 1;
        stmt = (Statement)this.statementList.get(last);
        this.applyTransactionTimeout(stmt);
        handler.parameterize(stmt);
        BatchResult batchResult = (BatchResult)this.batchResultList.get(last);
        batchResult.addParameterObject(parameterObject);
    } else {
        Connection connection = this.getConnection(ms.getStatementLog());
        stmt = handler.prepare(connection, this.transaction.getTimeout());
        handler.parameterize(stmt);
        this.currentSql = sql;
        this.currentStatement = ms;
        this.statementList.add(stmt);
        this.batchResultList.add(new BatchResult(ms, sql, parameterObject));
    }

    handler.batch(stmt);
    return -2147482646;
}
```

细心的小伙伴一定看出来了差异。当然，我们在这里就暂时不对 MyBatis 源码进行深入分析，感兴趣的小伙伴可以继续关注我们后面的课程。

模板方法模式的优缺点

优点：

1、利用模板方法将相同处理逻辑的代码放到抽象父类中，可以提高代码的复用性。

2、将不同的代码不同的子类中，通过对子类的扩展增加新的行为，提高代码的扩展性。

3、把不变的行为写在父类上，去除子类的重复代码，提供了一个很好的代码复用平台，符合开闭原则。

缺点:

1、类数目的增加，每一个抽象类都需要一个子类来实现，这样导致类的个数增加。

2、类数量的增加，间接地增加了系统实现的复杂度。

3、继承关系自身缺点，如果父类添加新的抽象方法，所有子类都要改一遍。

模板方法模式比较简单，相信小伙伴们肯定能学会，也肯定能理解好！只要勤加练习，多结合业务场景思考问题，就能够把模板方法模式运用好。