课程目标

1、掌握门面模式特征和应用场景

2、了解门面模式的优、缺点。

内容定位

1、深刻了解门面模式的应用场景。

门面模式

门面模式（Facade Pattern）又叫外观模式，提供了一个统一的接口，用来访问子系统中的一群接口。其主要特征是定义了一个高层接口，让子系统更容易使用，属于结构性模式。

原文: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

解释: 要求一个子系统的外部与其内部的通信必须通过一个同一的对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。

其实，在我们日常的编码工作中，我们都在有意无意地大量使用门面模式，但凡只要高层模块需要调度多个子系统（2 个以上类对象），我们都会自觉地创建一个新类封装这些子系统，提供精简接口，让高层模块可以更加容易间接调用这些子系统的功能。尤其是现阶段各种第三方 SDK，各种开源类库，很大概率都会使用门面模式。尤其是你觉得调用越方便的，门面模式使用的一般更多。

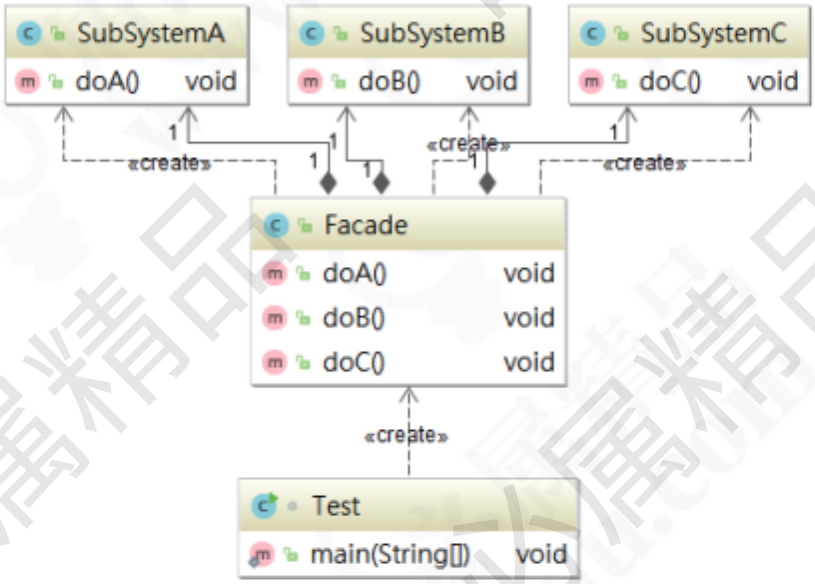
门面模式的应用场景

1、子系统越来越复杂，增加门面模式提供简单接口

2、构建多层系统结构，利用门面对象作为每层的入口，简化层间调用。

门面模式的通用写法

首先来看门面模式的 UML 类图：



门面模式主要包含 2 种角色：

外观角色（Facade）：也称 门面角色，系统对外的统一接口；

子系统角色（SubSystem）：可以同时有一个或多个 SubSystem。每个 SubSytem 都不是一个单独的类，而是一个类的集合。SubSystem 并不知道 Facade 的存在，对于 SubSystem 而言，Facade 只是另一个客户端而已（即 Facade 对 SubSystem 透明）。

下面是门面模式的通用代码，首先分别创建 3 个子系统的业务逻辑 SubSystemA、SubSystemB、SubSystemC，代码很简单：

```
public class SubSystemA {
    public void doA() {
        System.out.println("doing A stuff");
    }
}
public class SubSystemB {
    public void doB() {
        System.out.println("doing B stuff");
    }
}
public class SubSystemC {
    public void doC() {
        System.out.println("doing C stuff");
```

```
    }
}
```

然后，创建外观角色 Facade 类：

```
public class Facade {
    private SubSystemA a = new SubSystemA();
    private SubSystemB b = new SubSystemB();
    private SubSystemC c = new SubSystemC();

    // 对外接口
    public void doA() {
        this.a.doA();
    }

    // 对外接口
    public void doB() {
        this.b.doB();
    }

    // 对外接口
    public void doC() {
        this.c.doC();
    }
}
```

来看客户端代码：

```
    public static void main(String[] args) {
        Facade facade = new Facade();
        facade.doA();
        facade.doB();
        facade.doC();
    }
```

门面模式业务场景实例

Gper 社区上线了一个积分兑换礼品的商城，这礼品商城中的大部分功能并不是全部重新开发的，而是要去对接已有的各个子系统（如下图所示）：

这些子系统可能涉及到积分系统、支付系统、物流系统的接口调用。如果所有的接口调用全部由前端发送网络请求去调用现有接口的话，一则会增加前端开发人员的难度，二则会增加一些网络请求影响页面性能。这个时候就可以发挥门面模式的优势了。将所有现成的接口全部整合到一个类中，由后端提供统一的接口给前端调用，这样前端开发人员就不需要关心各接口的业务关系，只需要把精力集中在页面交互上。下面我们用代码来模拟一下这个场景。

首先，创建礼品的实体类 GiftInfo：

```
public class GiftInfo {
    private String name;

    public GiftInfo(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

然后，编写各个子系统的业务逻辑代码，分别创建积分系统 QualifyService 类：

```
public class QualifyService {
    public boolean isAvailable(GiftInfo giftInfo){
        System.out.println("校验" + giftInfo.getName() + " 积分资格通过,库存通过");
        return true;
    }
}
```

支付系统 PaymentService 类：

```
public class PaymentService {
```

```
    public boolean pay(GiftInfo pointsGift){
        //扣减积分
        System.out.println("支付" + pointsGift.getName() + " 积分成功");
        return true;
    }
}
```

物流系统 ShippingService 类：

```
public class ShippingService {

    //发货
    public String delivery(GiftInfo giftInfo){
        //物流系统的对接逻辑
        System.out.println(giftInfo.getName() + "进入物流系统");
        String shippingOrderNo = "666";
        return shippingOrderNo;
    }
}
```

然后创建外观角色 GiftFacdeService 类，对外只开放一个兑换礼物的 exchange()方法，在 exchange()方法内部整合 3 个子系统的所有功能。

```
public class GiftFacadeService {
    private QualifyService qualifyService = new QualifyService();
    private PaymentService pointsPaymentService = new PaymentService();
    private ShippingService shippingService = new ShippingService();

    //兑换
    public void exchange(GiftInfo giftInfo){
        if(qualifyService.isAvailable(giftInfo)){
            //资格校验通过
            if(pointsPaymentService.pay(giftInfo)){
                //如果支付积分成功
                String shippingOrderNo = shippingService.delivery(giftInfo);
                System.out.println("物流系统下单成功,订单号是:"+shippingOrderNo);
            }
        }
    }
}
```

最后，来看客户端代码：

```
    public static void main(String[] args) {
        GiftInfo giftInfo = new GiftInfo("《Spring 5 核心原理》");
        GiftFacadeService giftFacadeService = new GiftFacadeService();
        giftFacadeService.exchange(giftInfo);
    }
```

运行结果如下：

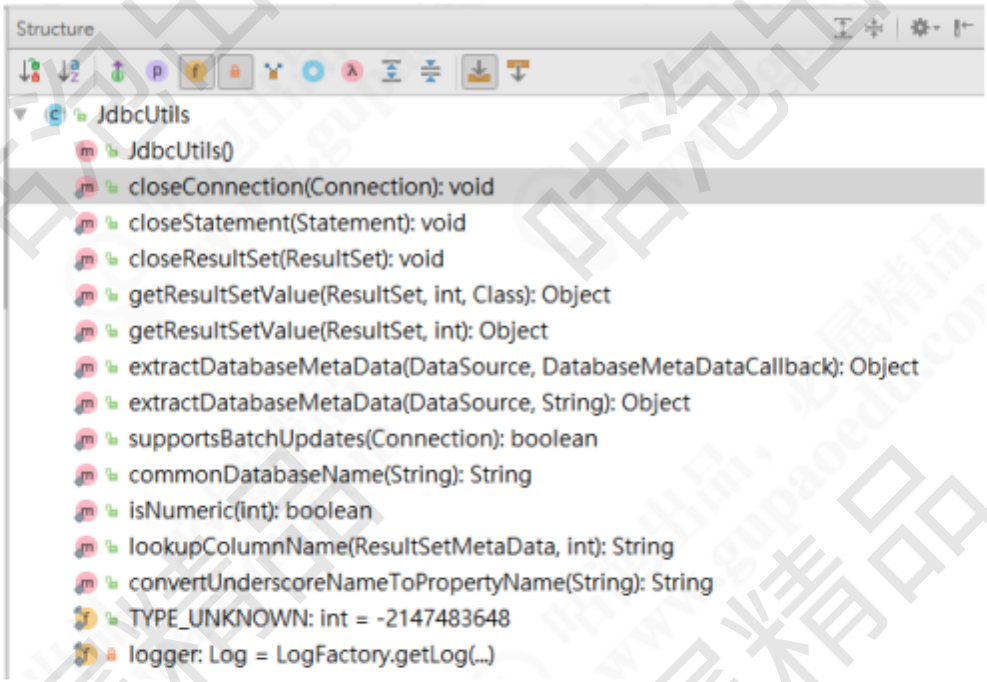
通过这样一个案例对比之后，相信大家对门面模式的印象非常深刻了。

门面模式在源码中的应用

下面我们来门面模式在源码中的应用，先来看 Spring JDBC 模块下的 JdbcUtils 类，它封装了和 JDBC 相关的所有操作，它一个代码片段：

```
public abstract class JdbcUtils {
    public static final int TYPE_UNKNOWN = -2147483648;
    private static final Log logger = LogFactory.getLog(JdbcUtils.class);

    public JdbcUtils() {
    }

    public static void closeConnection(Connection con) {
        if(con != null) {
            try {
                con.close();
            } catch (SQLException var2) {
                logger.debug("Could not close JDBC Connection", var2);
            } catch (Throwable var3) {
                logger.debug("Unexpected exception on closing JDBC Connection", var3);
            }
        }

    }

    public static void closeStatement(Statement stmt) {
        if(stmt != null) {
            try {
                stmt.close();
            } catch (SQLException var2) {
                logger.trace("Could not close JDBC Statement", var2);
            } catch (Throwable var3) {
                logger.trace("Unexpected exception on closing JDBC Statement", var3);
            }
        }

    }

    public static void closeResultSet(ResultSet rs) {
        if(rs != null) {
            try {
                rs.close();
            } catch (SQLException var2) {
                logger.trace("Could not close JDBC ResultSet", var2);
            } catch (Throwable var3) {
                logger.trace("Unexpected exception on closing JDBC ResultSet", var3);
            }
        }

    }
     ...
}
```

其他更多的操作，看它的结构就非常清楚了：



再来看一个 MyBatis 中的 Configuration 类。它其中有很多 new 开头的方法，来看一下源代码：

```
public MetaObject newMetaObject(Object object) {
        return MetaObject.forObject(object, this.objectFactory, this.objectWrapperFactory, this.reflectorFactory);
    }

    public ParameterHandler newParameterHandler(MappedStatement mappedStatement, Object parameterObject, BoundSql
boundSql) {
        ParameterHandler parameterHandler = mappedStatement.getLang().createParameterHandler(mappedStatement,
parameterObject, boundSql);
        parameterHandler = (ParameterHandler)this.interceptorChain.pluginAll(parameterHandler);
        return parameterHandler;
    }

    public ResultSetHandler newResultSetHandler(Executor executor, MappedStatement mappedStatement, RowBounds rowBounds,
ParameterHandler parameterHandler, ResultHandler resultHandler, BoundSql boundSql) {
        ResultSetHandler resultSetHandler = new DefaultResultSetHandler(executor, mappedStatement, parameterHandler,
resultHandler, boundSql, rowBounds);
        ResultSetHandler resultSetHandler = (ResultSetHandler)this.interceptorChain.pluginAll(resultSetHandler);
        return resultSetHandler;
    }

    public StatementHandler newStatementHandler(Executor executor, MappedStatement mappedStatement, Object
parameterObject, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) {
        StatementHandler statementHandler = new RoutingStatementHandler(executor, mappedStatement, parameterObject,
rowBounds, resultHandler, boundSql);
        StatementHandler statementHandler = (StatementHandler)this.interceptorChain.pluginAll(statementHandler);
        return statementHandler;
    }

    public Executor newExecutor(Transaction transaction) {
        return this.newExecutor(transaction, this.defaultExecutorType);
    }
```

上面的这些方法都是对 JDBC 中关键组件操作的封装。另外地在 Tomcat 的源码中也有体现，也非常的有意思。举个例子 RequestFacade 类，来看源码：

```
public class RequestFacade implements HttpServletRequest {
...
@Override
    public String getContentType() {

        if (request == null) {
            throw new IllegalStateException(
                            sm.getString("requestFacade.nullRequest"));
        }

        return request.getContentType();
    }


    @Override
    public ServletInputStream getInputStream() throws IOException {

        if (request == null) {
            throw new IllegalStateException(
                            sm.getString("requestFacade.nullRequest"));
        }

        return request.getInputStream();
    }


    @Override
    public String getParameter(String name) {

        if (request == null) {
            throw new IllegalStateException(
                            sm.getString("requestFacade.nullRequest"));
        }

        if (Globals.IS_SECURITY_ENABLED){
            return AccessController.doPrivileged(
                new GetParameterPrivilegedAction(name));
        } else {
            return request.getParameter(name);
        }
    }
...
}
```

我们看名字就知道它用了门面模式。它封装了非常多的 request 的操作，也整合了很多 servlet-api 以外的一些内容，给用户使用提供了很大便捷。同样，Tomcat 对 Response 和 Session 当也封装了 ResponseFacade 和 StandardSessionFacade 类，感兴趣的小伙伴可以去深入了解一下。

门面模式的优缺点

优点：

1、简化了调用过程，无需深入了解子系统，以防给子系统带来风险。

2、减少系统依赖、松散耦合

3、更好地划分访问层次，提高了安全性

4、遵循迪米特法则，即最少知道原则。

缺点:

1、当增加子系统和扩展子系统行为时，可能容易带来未知风险

2、不符合开闭原则

3、某些情况下可能违背单一职责原则。