极客时间训练营

# 系统架构设计师

## 第13章 层次架构设计理论与实践

授课：王建平

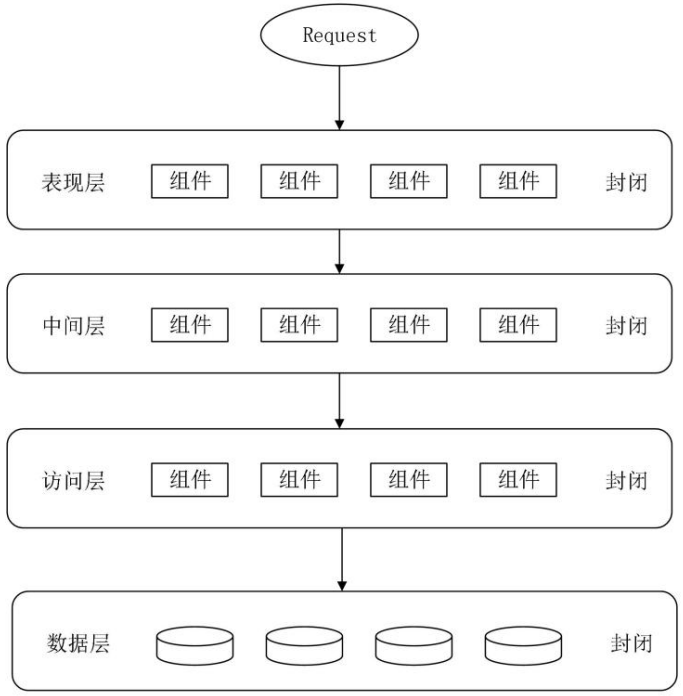
# 目录

# 目录

# 层次式体系结构概述

◆层次式体系结构设计将系统组成一个层次结构，每一层为上层服务，并作为下层客户。内部的层接口只对相邻的层可见，每一层最多只影响两层，只要给相邻层提供相同的接口，允许每层用不同的方法实现，为软件重用提供了强大的支持。（★）

◆层次式架构也称N层架构模式，分成表现层(展示层)、中间层(业务层)、数据访问层(持久层) 和数据层。（★★★）




MVC---MVP---MVVM

ORM

# 层次式体系结构概述

◆分层架构的一个特性就是关注分离。（★）
该层中的组件只负责本层的逻辑，组件的划分很容易明确组件的角色和职责，也比较容易开发、测试、管理和维护。

◆但设计时要注意以下两点：（★）
(1)要注意污水池反模式
污水池反模式，就是请求流简单地穿过几个层，每层里面基本没有做任何业务逻辑，或者做了很少的业务逻辑。如果请求超过20%，则应该考虑让一些层变成开放的。
(2)需要考虑的是分层架构可能会让你的应用变得庞大。
即使你的表现层和中间层可以独立发布，但它的确会带来一些潜在的问题，比如：分布模式复杂、健壮性下降、可靠性和性能的不足，以及代码规模的膨胀等。

# 典型真题

分层体现了（ ）思想。
A．关注点分离　B．复用　C．设计　D．场景

答案：A

层次式架构也称N层架构模式，分成表现层(展示层)、（ ）、数据访问层(持久层) 和数据层。
A．视图层　B．业务层　C．数据库层　D．方法层

答案：B

# 目录

# 表现层设计模式

◆MVC模式

MVC 强制地把一个应用的输入、处理、输出流程按照视图、控制、模型的方式进行分离，形成了控制器、模型、视图三个核心模块。（★★）

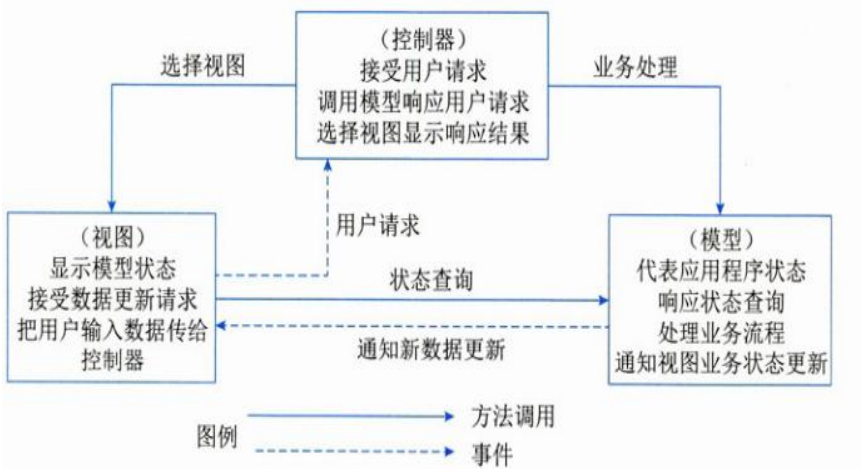
(1)视图 (View)：用户看到并与之交互的界面。视图向用户显示相关的数据，并接收用户输入的数据，但是它并不进行任何实际的业务处理。

(2)控制器(Controller)

接受用户的输入并调用模型和视图去完成用户的需求。该部分是用户界面与模型的接口。一方面它解释来自于视图的输入，将其解释成为系统能够理解的对象，同时它也识别用户动作，并将其解释为对模型特定方法的调用；另一方面，它处理来自于模型的事件和模型逻辑执行的结果，调用适当的视图为用户提供反馈。

(3)模型(Model)

应用程序的主体部分。模型表示业务数据和业务逻辑。一个模型能为多个视图提供数据。由于同一个模型可以被多个视图重用，所以提高了应用的可重用性。




◆使用MVC 模式来设计表现层，可以有以下的优点。

1、允许多种用户界面的扩展

2、易于维护

3、功能强大的用户界面。

4、增加应用的可拓展性、强壮型、灵活性。

# 表现层设计模式

以某个支付业务场景的“获取用户订单”为例。用户先获取订单详情，然后要查询订单状态并更新页面。
公司运用了MVC模式：

1、用户请求某个订单的详情

```
# Controller

function orderController.getOrder (request,  response) {
        checkUser(request.userId);
        orderInfo = orderModel.getOrder(request.userId, request.orderId);
        return '<img class="taro-img__mode-scaletofill"
src=orderInfo.image>
                <text>orderInfo.productName</text>
                <text>orderInfo.spec</text>
                <view>orderInfo.country

<text>orderInfo.priceYuan</text>.orderInfo.pricePen</text>
                <text">x orderInfo.count</text></view>
                <text>orderInfo.comment</text>}'
                <script>轮询订单状态的js代码</script>';
}
```

2、controller调用model取到订单的详情

3、controller用代码拼接了html页面视图


4、在终端渲染出页面

HTML页面视图

```
<img class="taro-img__mode-scaletofill"
src="https://img11.360buyimg.com/jdphoto/5aadf9daNd4909ddc.png">
<text>怡宝 饮用水 纯净水4.5L*4桶装水 整箱装</text>
<text>4.5L*4桶装水整箱</text>
<view>¥<text>30</text>.20</text><text">x 4</text></view>
<text>支持7天无理由退货(拆封后不支持)</text>

<script>
      # 轮询订单状态的js代码，如果订单状态有更新则实时展示在页面
上
</script>
```

```
# Model

function orderModel.getOrder (userId,  orderId) {
        orderInfo = mysql.select("select * from orders where
user_id = " + userId + " and order_id = " + orderId + ";");
        return orderInfo;
}

function orderModel.listenOrderState (userId,  orderId) {
        orderState = redis.sub(userId, orderId);
        return orderState;
}
```

5、必要的时候，需要跟踪订单状态。

# 表现层设计模式

◆MVP模式

MVP模式中Model提供数据，View负责显示，Presenter/Controller负责逻辑的处理。（★★）

MVP与MVC 有一些显著的区别，MVC模式中允许 View 和 Model直接进行"交流"，在MVP 模式中是不允许的。在MVP 中 View 并不直接使用 Model，它们之间的通信是通过 Presenter (MVC中的Controller)来进行的，所有的交互都发生在 Presenter 内部，而在MVC中View会直接从Model 中读取数据而不是通过Controller。

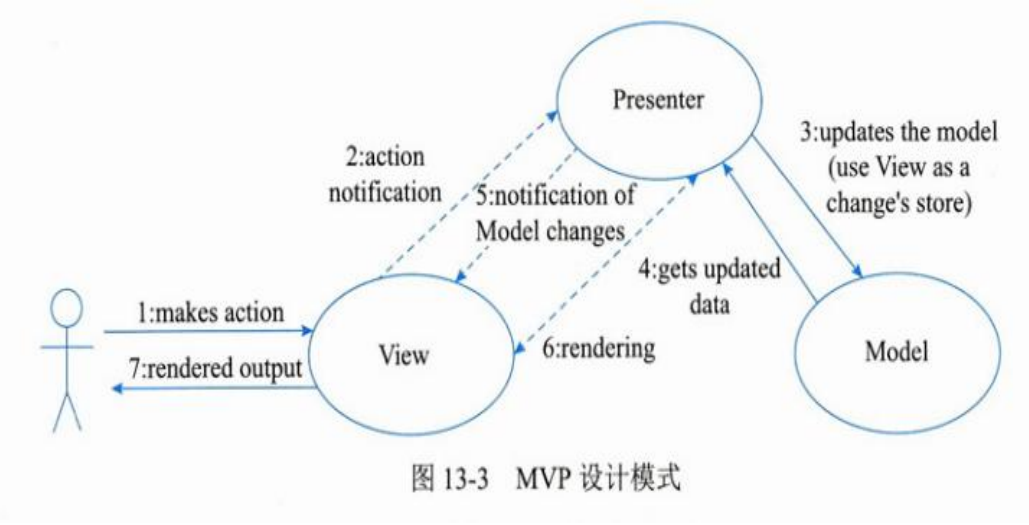



图 13-3　MVP 设计模式

◆使用MVP模式来设计表现层，可以有以下的优点。

(1 ）模型与视图完全分离，可以修改视图而不影响模型。

(2 ）可以更高效地使用模型，因为所有的交互都发生在一个地方—-Presenter 内部。

(3 ）可以将一个Presenter 用于 多个视图，而不需要改变Presenter 的逻辑。这个特性非常的
有用，因为视图的变化总是比模型的变化频繁。

(4 ）如果把逻辑放在Presenter 中，就可以脱离用户接口来测试这些逻辑（单元测试）。

目前，MVP 模式被更多地用 在 Android 开发当中。

# 表现层设计模式

改进后的做法，View不再和Model直接联系，Presenter成为唯一的直接面向用户请求的入口，统一管理请求、分配资源、必要的检查等。

1、用户请求某个订单的详情


3、Presenter根据业务需求和获取的数据，生成html页面视图

4、在终端渲染出页面

```
HTML页面

<img class="taro-img__mode-scaletofill"
src="https://img11.360buyimg.com/jdphoto/5aadf9daNd4909ddc.png"
>
<text>怡宝 饮用水 纯净水4.5L*4桶装水 整箱装</text>
<text>4.5L*4桶装水整箱</text>
<view>¥<text>30</text>.20</text><text">x 4</text></view>
<text>支持7天无理由退货(拆封后不支持)</text>
<script>
      # 轮询订单状态的js代码，如果订单状态有更新则实时展示在页面上
</script>
```

```
# Presenter

function orderController.getOrder (request,  response) {
        checkUser(request.userId);
        orderInfo = orderModel.getOrder(request.userId, request.or
        return '<img class="taro-img__mode-scaletofill" src=orderInf    age>
                <text>orderInfo.productName</text>
                <text>orderInfo.spec</text>
                <view>orderInfo.country
                <text>orderInfo.priceYuan</text>.orderInfo.priceFen</text>
                <text">x orderInfo.count</text></view>
                <text>orderInfo.comment</text>}'
                <script>轮询订单状态的js代码</script>';
}

function orderModel.listenOrderState(request, response) {
                checkUser(request.userId);
                orderState = orderModel.listenOrderState(request.userId,
request.orderId);
                return orderState;
}
```

用程序代码拼接出html的视图代码实在不是好办法，下面咱们解决这个问题。

6、Presenter访问model对应的方法来获取订单最新状态

2、Presenter访问model获取订单的详情

5、远程调用Presenter的listenOrderState，轮询订单最新状态

```
# Model
function orderModel.getOrder (userId,  orderId) {
        orderInfo = mysql.select("select * from
orders where user_id = " + userId + " and order_id
= " + orderId + ";");
        return orderInfo;
}

function orderModel.listenOrderState (userId,
orderId) {
        orderState = redis.sub(userId, orderId);
        return orderState;
}
```

同理这里的SQL拼接也不是好办法，况且还有SQL注入的问题

# 表现层设计模式

◆MVVM模式（★★）

MVVM模式正是为解决MVP中UI种类变多，接口也会不断增加的问题而提出的。MVVM模式全称是模型-视图-视图模型 (Model-View-ViewModel)，它和MVC、MVP类似，主要目的都是为了实现视图和模型的分离，不同的是 MVVM 中，View 与Model 的交互通过 ViewModel来实现。ViewModel是MVVM的核心，它通过DataBinding实现 View 与Model之间的双向绑定，其内容包括数据状态处理、数据绑定及数据转换。例如，View 中某处的状态和Model中某部分数据绑定在一起，这部分数据一旦变化将会反映到 View 层。而这个机制通过 ViewModel来实现。

ViewModel，即视图模型，是一个专门用于数据转换的控制器，它可以把对象信息转换为视图信息，将命令从视图携带到对象。View和ViewModel之间使用DataBinding及其事件进行通信。View的用户接口事件仍然由 View自身处理，并把相关事性映射到ViewModel，以实现View中的对象与视图模型内容的同步，且可通过双向数据绑定进行更新。

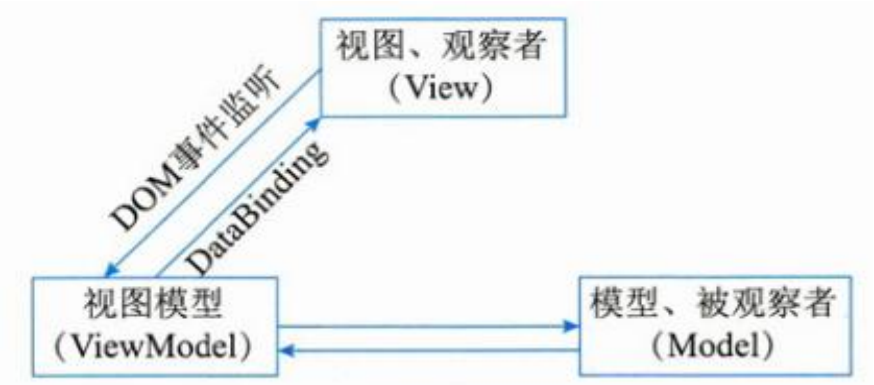
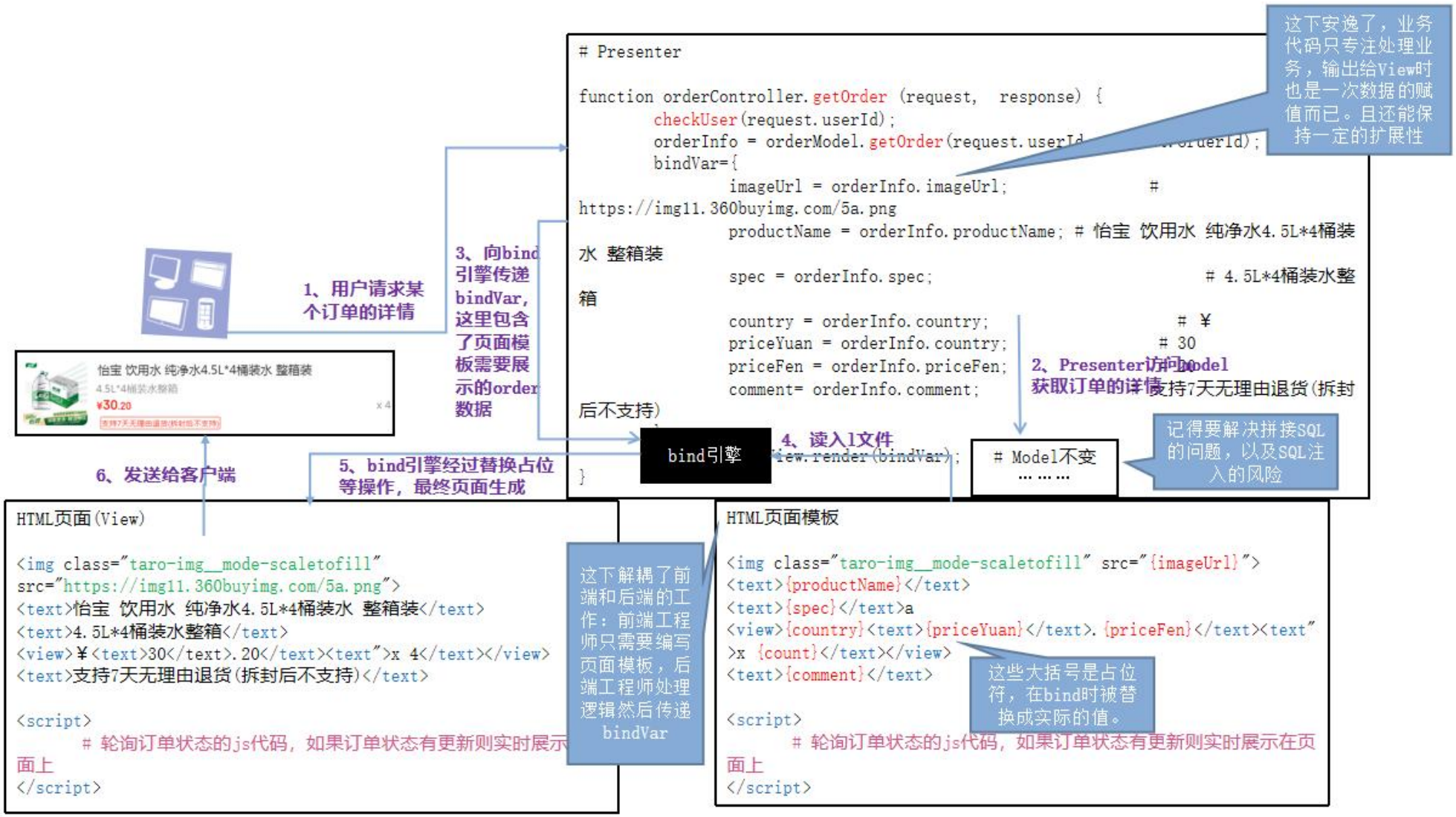



图 13-4　MVVM 设计模式

# 表现层设计模式


1、用户请求某个订单的详情
3、向bind引擎传递bindVar，这里包含了页面模板需要展示的order数据
2、Presenter访问Model获取订单的详情
4、读入1文件
5、bind引擎经过替换占位等操作，最终页面生成
6、发送给客户端
bind引擎
这下安逸了，业务代码只专注处理业务，输出给View时也是一次数据的赋值而已。且还能保持一定的扩展性
记得要解决拼接SQL的问题，以及SQL注入的风险
这下解耦了前端和后端的工作：前端工程师只需要编写页面模板，后端工程师处理逻辑然后传递bindVar
这些大括号是占位符，在bind时被替换成实际的值。
# Presenter
function orderController.getOrder (request, response) {
checkUser(request.userId);
orderInfo = orderModel.getOrder(request.userId, orderId);
bindVar={
imageUrl = orderInfo.imageUrl; # https://img11.360buyimg.com/5a.png
productName = orderInfo.productName; # 怡宝 饮用水 纯净水4.5L*4桶装水 整箱装
spec = orderInfo.spec; # 4.5L*4桶装水整箱
country = orderInfo.country; # ¥
priceYuan = orderInfo.country; # 30
priceFen = orderInfo.priceFen; # 20
comment= orderInfo.comment; # 支持7天无理由退货(拆封后不支持)
}
view.render(bindVar);
# Model不变
……
HTML页面(View)
<img class="taro-img__mode-scaletofill" src="https://img11.360buyimg.com/5a.png">
<text>怡宝 饮用水 纯净水4.5L*4桶装水 整箱装</text>
<text>4.5L*4桶装水整箱</text>
<view>¥<text>30</text>.20</text><text">x 4</text></view>
<text>支持7天无理由退货(拆封后不支持)</text>
<script>
# 轮询订单状态的js代码，如果订单状态有更新则实时展示在页面上
</script>
HTML页面模板
<img class="taro-img__mode-scaletofill" src="{imageUrl}">
<text>{productName}</text>
<text>{spec}</text>a
<view>{country}<text>{priceYuan}</text>.{priceFen}</text><text"
>x {count}</text></view>
<text>{comment}</text>
<script>
# 轮询订单状态的js代码，如果订单状态有更新则实时展示在页面上
</script>
怡宝 饮用水 纯净水4.5L*4桶装水 整箱装
4.5L*4桶装水整箱
¥30.20
x 4
支持7天无理由退货(拆封后不支持)

# 表现层设计思想

◆表现层中UIP设计思想

UIP是微软社区开发的众多Application Block 中的其中之一，它是开源的。UIP 提供了一个扩展的框架，用于简化用户界面与商业逻辑代码的分离的方法，可以用它来写复杂的用户界面导航和工作流处理，并且它能够复用在不同的场景并可以随着应用的增加而进行扩展。

◆UIP框架把表现层分为了以下二层：（★★★）

- ✓ User Interface Components: 这个组件就是原来的表现层，用户看到的和进行交互都是这个组件，它负责获取用户的数据并且返回结果。
- ✓ User Interface Process Components: 这个组件用于协调用户界面的各部分，使其配合后台的活动，例如导航和工作流控制，以及状态和视图的管理。

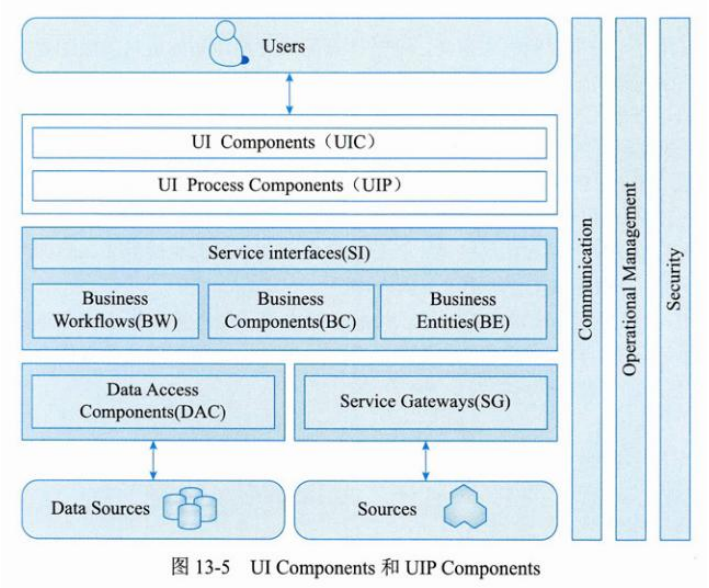



图 13-5　UI Components 和 UIP Components

# 表现层设计技术

◆基于 XML 的界面管理技术可实现灵活的界面配置、界面动态生成和界面定制。其思路是用XML 生成配置文件及界面所需的元数据，
◆按不同需求生成界面元素及软件界面。（★★★）
◆基于XML 界面管理技术，包括界面配置、界面动态生成和界面定制三部分。

- ✓ 界面配置是对用户界面的静态定义，通过读取配置文件的初始值对界面配置。由界面配置对软件功能进行裁剪、重组和扩充，以实现特殊需求。
- ✓ 界面定制是对用户界面的动态修改过程，在软件运行过程中，用户可按需求和使用习惯对界面元素(如菜单、工具栏、键盘命令) 的属性(如文字、图标、大小和位置等) 进行修改。软件运行结束，界面定制的结果被保存。

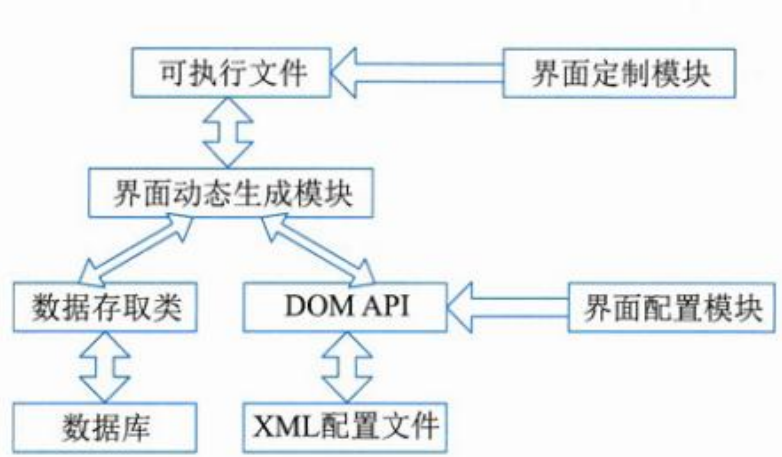



图 13-6　基于 XML 的界面管理技术框图

# 目录

# 业务逻辑层组件设计

◆业务逻辑层框架。业务框架位于系统架构的中间层，是实现系统功能的核心组件。采用容器的形式，便于系统功能的开发、代码重用和管理。在业务容器中，业务逻辑是按照 Domain Model-Service-Control 思想来实现的。（★）　　其中：

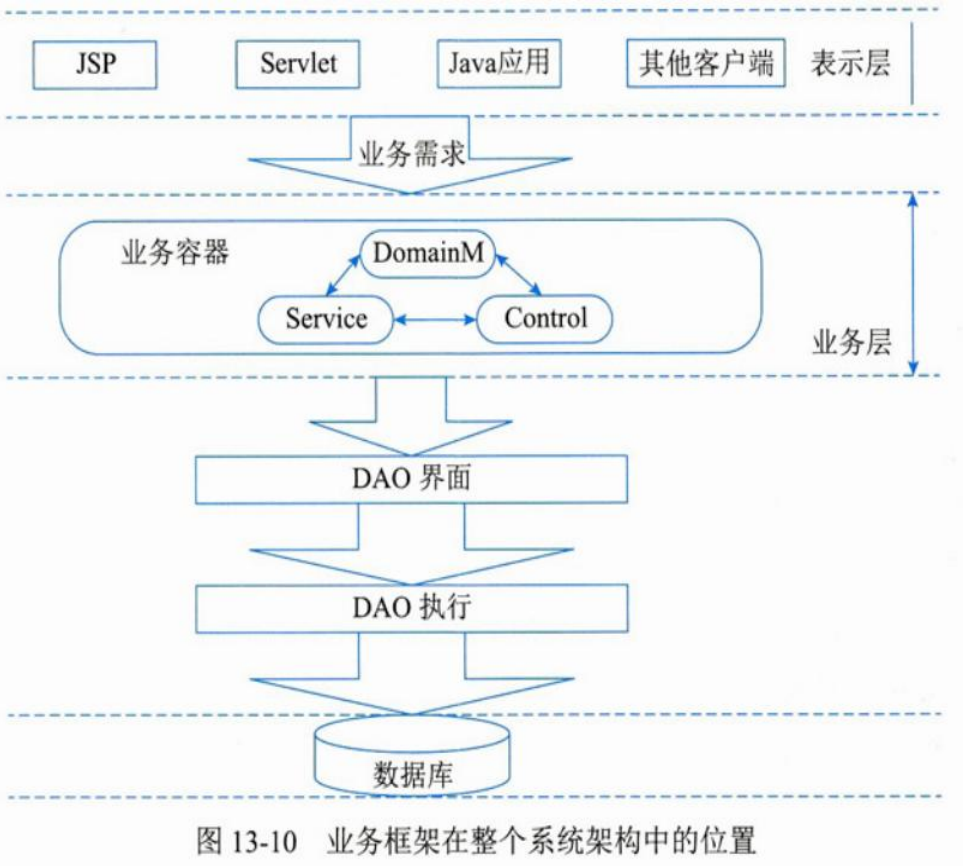



图 13-10　业务框架在整个系统架构中的位置

◆业务层采用业务容器的方式存在于整个系统当中，采用此方式可以大大降低业务层和相邻各层的耦合，表示层代码只需要将业务参数传递给业务容器，而不需要业务层多余的干预。如此一来，可以有效地防止业务层代码渗透到表示层。

◆在业务容器中，业务逻辑是按照Domain Model—Service—Control思想来实现的。（★★★）

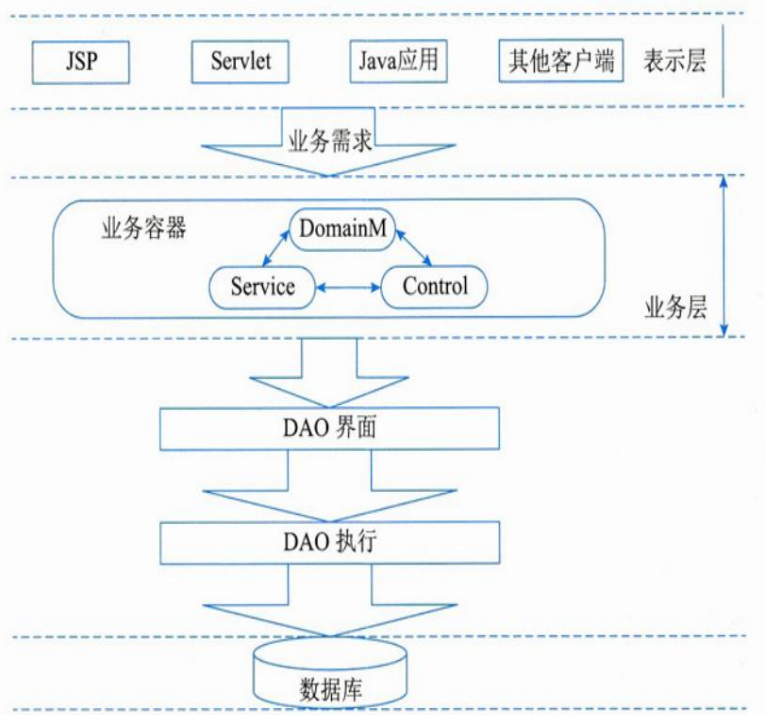
(1)Domain Model是领域层业务对象，它仅仅包含业务相关的属性。

(2)Service是业务过程实现的组成部分，是应用程序的不同功能单元，通过在这些服务之间定义良好的接口和契约联系起来。

(3)Control服务控制器，是服务之间的纽带，不同服务之间的切换就是通过它来实现的。

# 业务逻辑层组件设计

◆业务逻辑层组件设计。业务逻辑层组件分为接口和实现类两个部分。接口用于定义业务逻辑组件，定义业务逻辑组件必须实现的方法是整个系统运行的核心。通常按模块来设计业务逻辑组件，每个模块设计一个业务逻辑组件，并且每个业务逻辑组件以多个 DAO(Data Access Objet) 组件作为基础，从而实现对外提供系统的业务逻辑服务。




◆业务逻辑的内容包括四部分：（★★）

✓ 领域实体：定义了业务中的对象，对象有属性和行为；

✓ 业务规则：定义了需要完成一个动作，必须满足的条件；

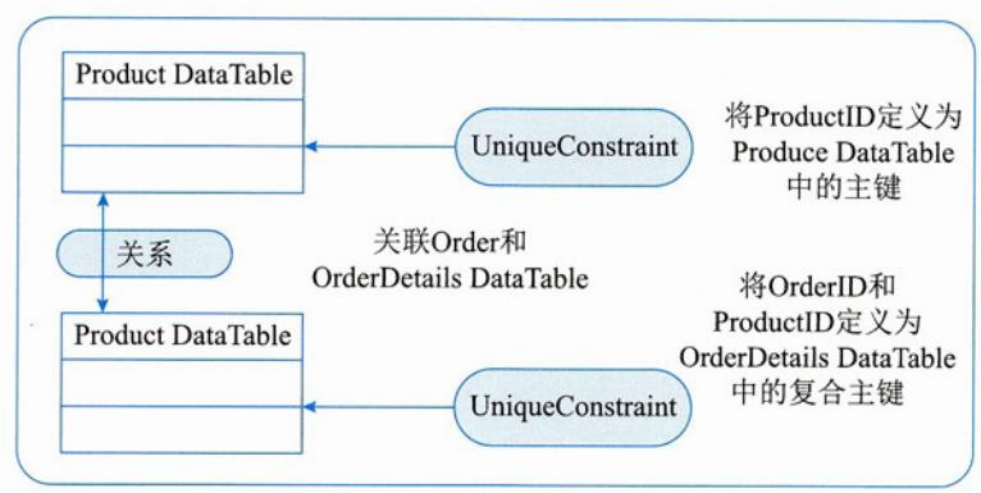
✓ 数据完整性：某些数据不可少；

✓ 工作流：业务流程的全部或部分自动化，在此过程中，文档、信息或任务按照一定的过程规则流转，实现组织成员间的协调工作以达到业务的整体目标。

# 业务逻辑层组件设计

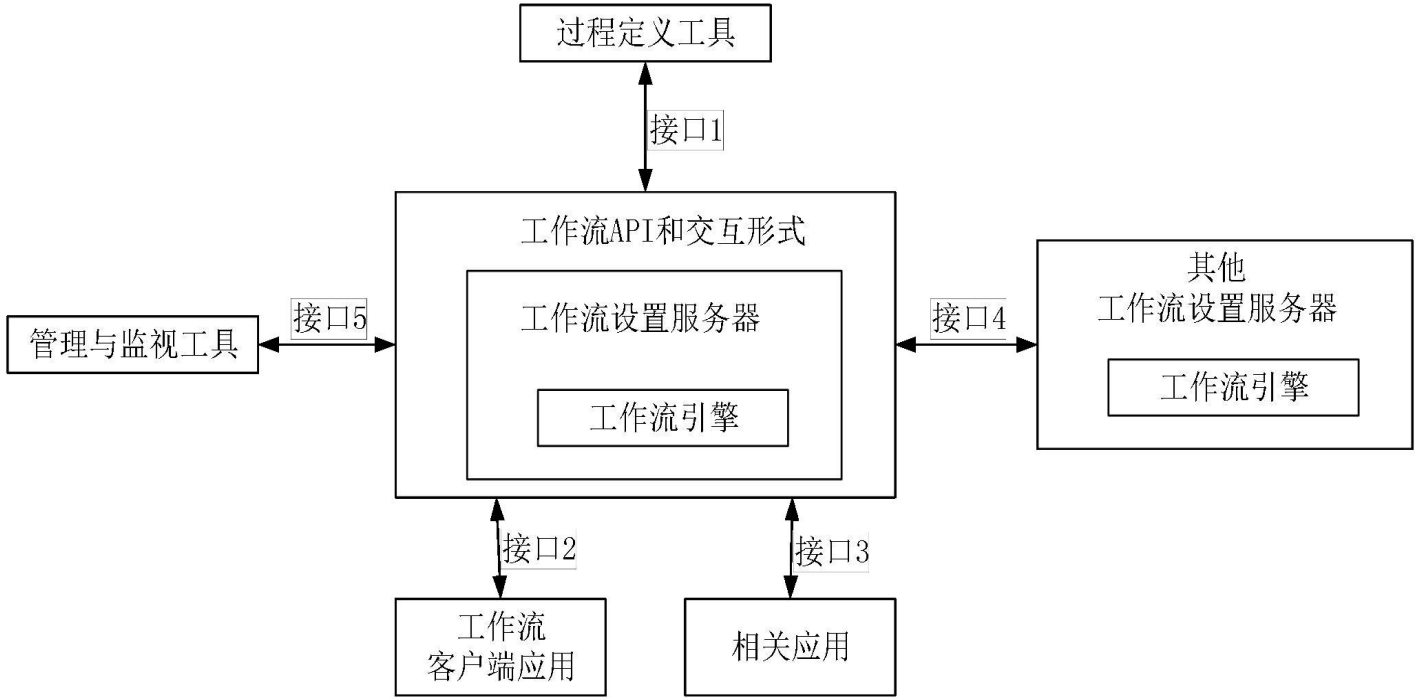
◆业务逻辑层实体设计。逻辑层实体提供对业务数据及相关功能(在某些设计中)的状态编程访问。业务逻辑层实体可以使用具有复杂架构的数据来构建，这种数据通常来自数据库中的多个相关表。业务逻辑层实体数据可以作为业务过程的部分 I/O 参数传递。业务逻辑层实体可以是可序列化的，以保持它们的当前状态。（XML和DataSet）

```
<?xml version="1.0"?>
<Product xmlns="urn:aUniqueNamespace">
<ProductID>1</ProductID>
<ProductName>Chai</ProductName>
<QuantityPerUnit>10 boxes x 20 bags</QuantityPerUnit>
<UnitPrice>18.00</UnitPrice>
<UnitsInStock>39</UnitsInStock>
<UnitsOnOrder>0</UnitsOnOrder>
<ReorderLevel>10</ReorderLevel>
</Product>
```



# 业务逻辑层组件设计

◆业务逻辑层工作流设计。工作流管理联盟 (Workow Management Coalition)将工作流定义为:业务流程的全部或部分自动化，在此过程中，文档、信息或任务按照一定的过程规则流转，实现组织成员间的协调工作以达到业务的整体目标。（★★）



# 业务逻辑层组件设计

(1)interface1:过程定义导入/导出接口。这个接口的特点是：转换格式和API调用，从而支持过程定义信息间的互相转换。这个接口也支持已完成的过程定义或过程定义的一部分之间的互相转换。早期标准是WPDL,后来发展为XPDL。
(2)interface 2:客户端应用程序接口。通过这个接口工作流机可以与任务表处理器交互，代表用户资源来组织任务。然后由任务表处理器负责，从任务表中选择、推进任务项。由任务表处理器或者终端用户来控制应用工具的活动。
(3)interface 3:应用程序调用接口。允许工作流机直接激活一个应用工具，来执行一个活动。典型的是调用以后台服务为主的应用程序，没有用户接口。当执行活动要用到的工具，需要与终端用户交互，通常是使用客户端应用程序接口来调用那个工具，这样可以为用户安排任务时间表提供更多的灵活性。
(4)interface 4:工作流机协作接口。其目标是定义相关标准，以使不同开发商的工作流系统产品相互间能够进行无缝的任务项传递。WFMC定义了4个协同工作模型，包含多种协同工作能力级别。
(5)interface 5:管理和监视接口。提供的功能包括用户管理、角色管理、审查管理、资源控制、过程管理和过程状态处理器等。
用工作流的思想组织业务逻辑，优点是：将应用逻辑与过程逻辑分离，在不修改具体功能的情况下，通过修改过程模型改变系统功能，完成对生产经营部分过程或全过程的集成管理，可有效地把人、信息和应用工具合理地组织在一起，发挥系统的最大效能。

# 目录

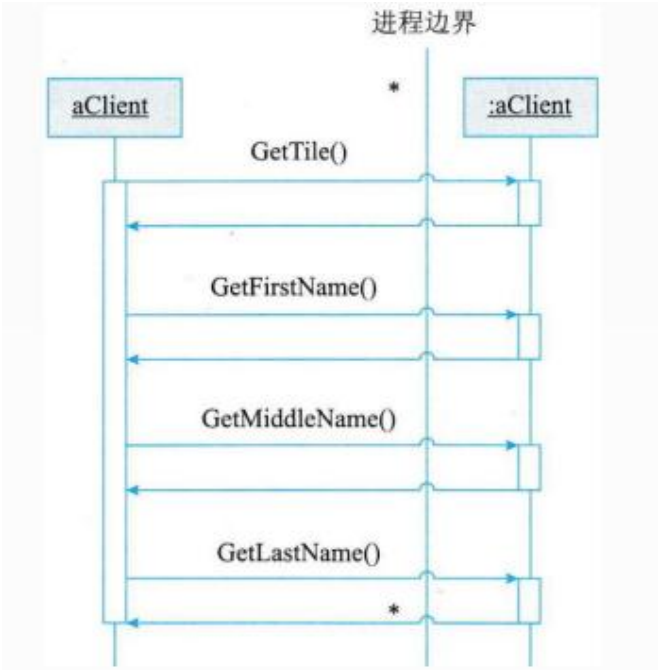
# 数据访问层设计

5种数据访问模式

◆在线访问（★★）
在线访问是最基本的数据访问模式。在线访问模式会占用一个数据库连接，读取数据，每个数据库操作都会通过这个连接不断地与后台的数据源进行交互。

◆在线访问方式的优点：
✓ 可以处理复杂的 Select 语句
✓ 性能比直接的 SQL要优越一些

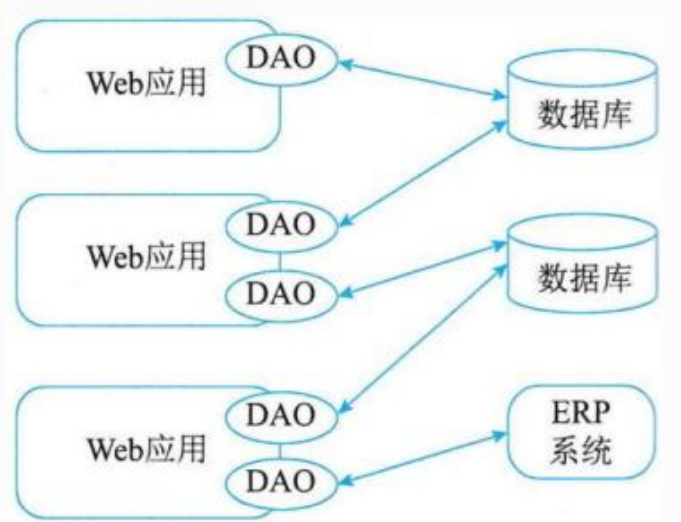
◆在线访问方式的缺点：
■ 业务对象和数据访问代码完全耦合在一起，代码混乱
■ 修改维护上相对困难
■ 开发程序员必须能看懂SQL语句



# 数据访问层设计

◆DataAccess Object（★★）
DAO(数据访问对象)模式是标准J2EE 设计模式之一，开发人员常常用这种模式将底层数据访问操作与高层业务逻辑分离开。
◆业务对象应该关注的是业务逻辑，不应该关心数据存取的细节，DAO组件只对他的客户端暴露一些非常简单的DAO外部接口，而将数据源的实现细节对客户端完全的隐藏起来。
DAO提供了访问关系型数据库所需操作的接口，将数据访问和业务逻辑分离，对上层提供面向对象的数据访问接口。
◆优点：DAO设计模式可以减少代码量，增强程序的可移植性，提高代码的可读性。

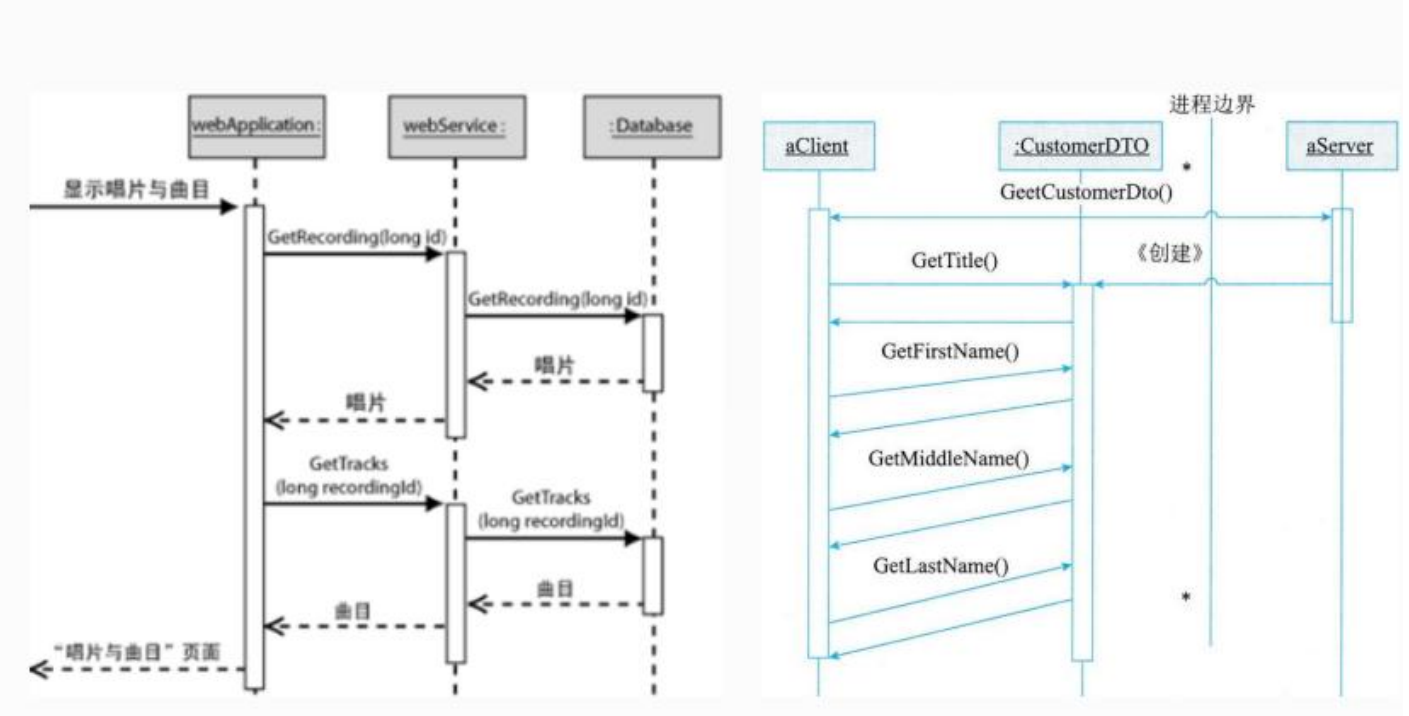
◆典型的 DAO实现包括以下组件：
✓ 一个DAO 工厂类
✓ 一个DAO 接口
✓ 一个实现了 DAO 接口的具体类
✓ 数据传输对象



# 数据访问层设计

◆Data Transfer Object（★★）

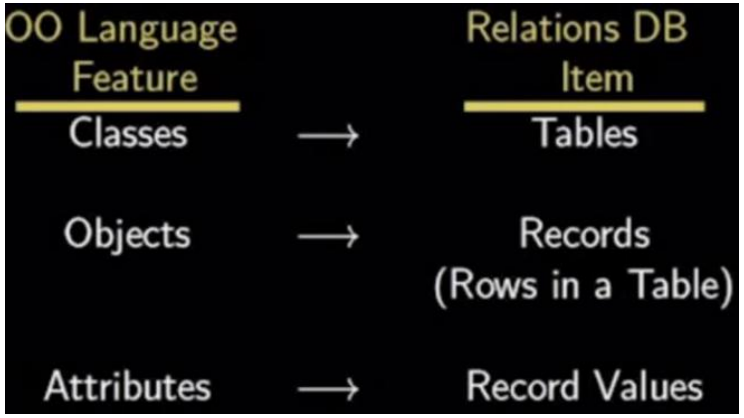
DTO(数据传输对象)是这样一组对象或是数据的容器，它需要跨不同的进程或是网络的边界来传输数据。这类对象本身应该不包含具体的业务逻辑，与业务逻辑解耦。



# 数据访问层设计

◆离线数据模式（★★）
离线数据模式是以数据为中心，数据从数据源获取之后，将按照某种预定义的结构存放在系统中，成为应用的中心。离线，对数据的各种操作独立于各种与后台数据源之间的连接或是事务；
XML集成，数据可以方便地与XML格式的文档之间互相转换；独立于数据源，离线数据模式的不同实现定义了数据的各异的存放结构和规则，这些都是独立于具体的某种数据源的。

◆对象/关系映射(O/R Mapping)（★★）
对象/关系映射的基本思想来源于这样一种现实: 多数应用的数据都是存在关系型数据库中，而这些应用程序中的数据在开发或是运行时是以对象的形式组织起来的，那么对象/关系映射就提供了这样一种工具或平台，能够将应用程序中的数据转换成关系型数据库中的记录，或是将关系数据库中的 记录转换成应用 程序中的代码便于操作的对象。



# 数据访问层设计

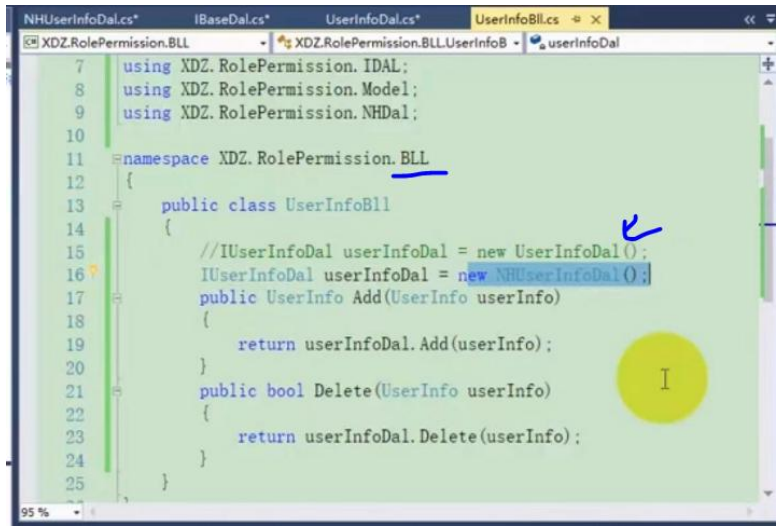
◆工厂模式定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。这里可能会处理对多种数据库的操作，因此，需要首先定义一个操纵数据库的接口，然后根据数据库的不同，由类工厂决定实例化哪个类。

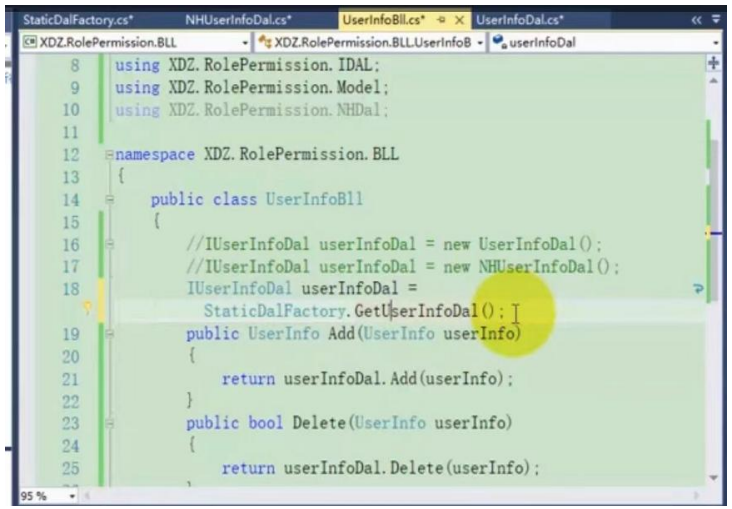
**◆工厂模式在数据库访问层的应用**

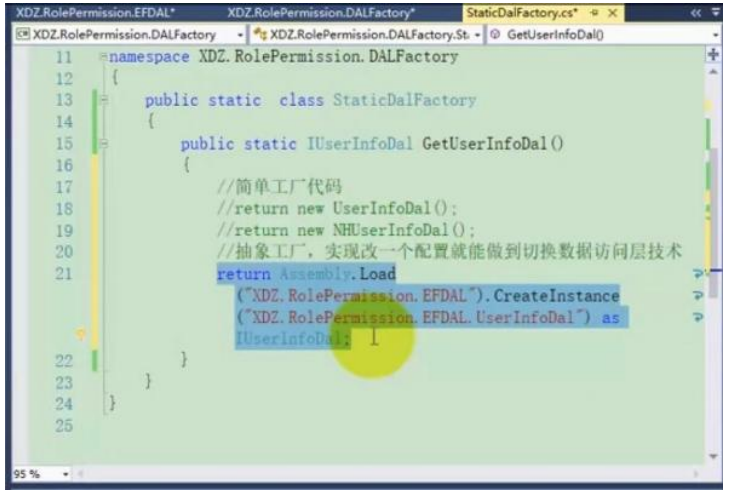
首先定义一个操纵数据库的接口DataAccess,然后根据数据库的不同，由类工厂决定实例化哪个类。

因为DataAccess的具体实现类有一些共同的方法，所以先从DataAccess实现一个抽象AbstractDataAccess类，包含一些公用方法。然后，分别为SQLServer、Oracle和OleDb数据库编写三个数据访问的具体实现类。

现在已经完成了所要的功能，下面需要创建一个Factory类，来实现自动数据库切换的管理。这个类很简单，主要的功能就是根据数据库类型，返回适当的数据库操纵类。



# 数据访问层设计

◆ORM, Hibernate 与 CMP2.0 设计思想。ORM (Object-Relation Mapping)在关系型数据库和对象之间作一个映射，这样，在具体操纵数据库时，就不需要再去和复杂的 SQL 语句打交道，只要像平时操作对象一样操作即可。Hibernate 是一个功能强大，可以有效地进行数据库数据到业务对象的 O/R 映射方案。Hibernate 推动了基于普通 Java 对象模型，用于映射底层数据结构的持久对象的开发。

◆XML Schema。用 XMLSchema 来描述 XML 文档合法结构、内容和限制，提供丰富的数据类型。

◆事务处理设计。事务必须服从 ISO/IEC 所制定的 ACID 原则。ACID 是原子性(Atomicity)、一致性(Consistency)、隔离性 (Isolation) 和持久性(Durability) 的缩写。

**事务处理设计**

JavaBean中使用JDBC方式进行事务处理：在JDBC中，打开一个连接对象Connection时，默认是auto-commit模式，每个SQL语句都被当作一个事务，即每次执行一个语句，都会自动地得到事务确认。为了能将多个SQL语句组合成一个事务，要将auto-commit模式屏蔽掉。在auto-commit模式屏蔽掉之后，如果不调用commit()方法，SQL语句不会得到事务确认。在最近一次commit()方法调用之后的所有SQL会在方法commit()调用时得到确认。

# 数据访问层设计

◆连接对象管理设计

**连接对象管理设计：**通过资源池解决资源频繁分配、释放所造成的问题。

建立连接池的第一步，就是要建立一个静态的连接池。所谓静态，是指池中的连接是在系统初始化时就分配好的，并且不能够随意关闭。Java中给我们提供了很多容器类，可以方便地用来构建连接池，如Vector、Stack等。在系统初始化时，根据配置创建连接并放置在连接池中，以后所使用的连接都是从该连接池中获取的，这样就可以避免连接随意建立、关闭造成的开销。

有了这个连接池，下面就可以提供一套自定义的分配、释放策略。当客户请求数据库连接时，首先看连接池中是否有未分配出去的连接。如果存在空闲连接则把连接分配给客户，并标记该连接为已分配。若连接池中没有空闲连接，就在已经分配出去的连接中，寻找一个合适的连接给客户，此时该连接在多个客户间复用。当客户释放数据库连接时，可以根据该连接是否被复用，进行不同的处理。如果连接没有使用者，就放入到连接池中，而不是被关闭。

# 目录

# 数据架构规划与设计

数据库设计与XML设计融合

◆XML 文档分为两类：（★★）

1、以数据为中心的文档，这种文档在结构上是规则的，在内容上是同构的，具有较少的混合内容和嵌套层次，只关心文档中的数据而并不关心数据元素的存放顺序，这种文档简称为数据文档，常用来存储和传输Web 数据。如XML文档包含销售数据、餐馆菜单。

2、以文档为中心的文档，这种文档的结构不规则，内容比较零散，具有较多的混合内容，并且元素之间的顺序是有关的，这种文档常用来在网页上发布描述性信息、产品性能介绍和 E-mail信息等。

◆XML文档的存储方式有两种：（★★）

(1)基于文件的存储方式。基于文件的存储方式是指将 XML 文档按其原始文本形式存储。这种存储方式需维护某种类型的附加索引，以建立文件之间的层次结构。

基于文件的存储方式的特点：

✓ 无法获取 XML 文档中的结构化数据；

✓ 通过附加索引可以定位具有某些关键字的 XML 文档，一旦关键字不确定，将很难定位；

✓ 查询时只能以原始文档的形式返回，即不能获取文档内部信息；

✓ 文件管理存在容量大、管理难的缺点。

# 数据架构规划与设计

(2)数据库存储方式。数据库在数据管理方面具有管理方便、存储占用空间小、检索速度快、修改效率高和安全性好等优点。采用数据库对 XML 文档进行存取和操作，这样可以利用相对成熟的数据库技术处理XML 文档内部的数据。

数据库存储方式的特点：

- 能够管理结构化和半结构化数据；
- 具有管理和控制整个文档集合本身的能力；
- 可以对文档内部的数据进行操作；
- 具有数据库技术的特性，如多用户、并发控制和一致性约束等。
- 管理方便，易于操作。

# 物联网层次架构设计

◆物联网层次架构：应用层、网络层、感知层（★★★）

1.感知层
感知层用于识别物体、采集信息。感知层包括二维码标签和识读器、RFID 标签和读写器、摄像头、GPS、传感器、M2M 终端、传感器网关等，主要功能是识别对象、采集信息。感知层解决的是数据获取问题。它首先通过传感器、数码相机等设备，采集外部物理世界的数据，然后通过 RFID、条码、工业现场总线、蓝牙、红外等短距离传输技术传递数据。感知层的关键技术包括检测技术、短距离无线通信技术等。
2.网络层
网络层将感知层获取的信息进行传递和处理，数据可以通过移动通信网、互联网、企业内部网、各类专网、小型局域网进行传输。物联网的网络层是建立在现有的移动通信网和互联网基础上的。网络层的关键技术包括长距离有线和无线通信技术、网络技术等。
3.应用层
应用层解决的是信息处理和人机交互问题。网络层传输来的数据在这一层进入各类信息系统进行处理，并通过各种设备与人进行交互。应用层分为两个子层：

- ✓ 应用程序层，进行数据处理，涵盖了国民经济和社会的每一领域，包括电力、医疗银行、交通、环保、物流、工业、农业、城市管理、家居生活等，是物联网作为深度信息化的重要体现。
- ✓ 终端设备层，提供人机接口。

# 层次架构案例分析

◆电子小票物联网架构

采用感知层、网络层和应用层的 3 层物联网体系架构模型，电子小票物联网的架构见图 13-22。

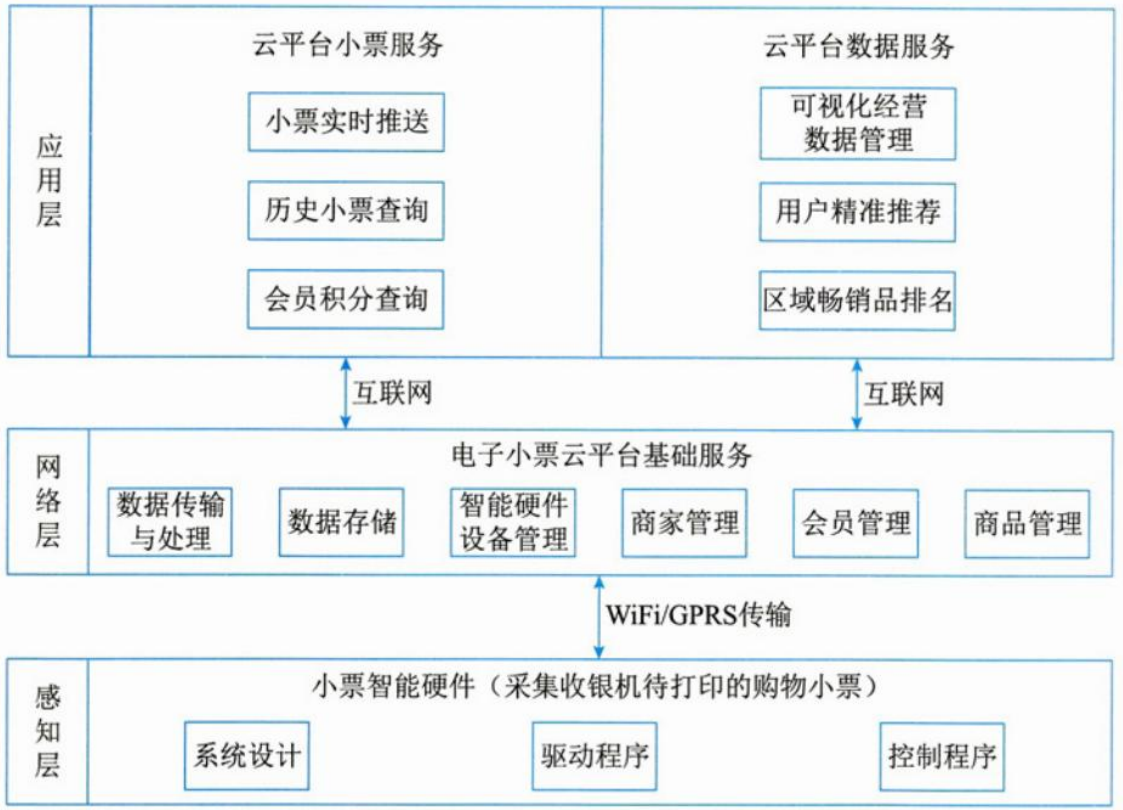



图 13-22　电子小票物联网架构

# 层次架构案例分析

◆电子商务网站(网上商店PetShop)

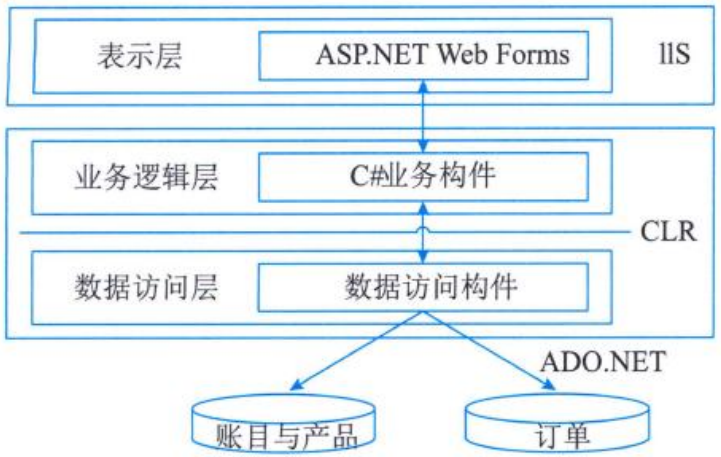



图 13-15　Net 中标准的 BS 分层式结构

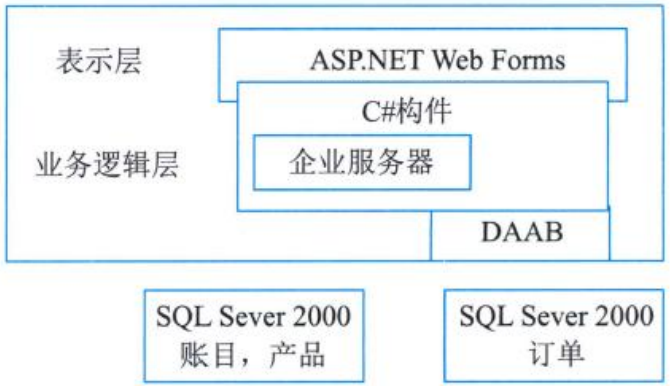



图 13-16　PetShop 2.0 的体系架构

◆从图13-16中可以看到，并没有明显的数据访问层设计。这样的设计虽然提高了数据访问的性能，但也同时导致了业务逻辑层与数据访问的职责混乱。

◆ PetShop 3.0纠正了此前层次不明的问题， 将数据访问逻辑作为单独的一层独立出来。

# 层次架构案例分析

◆ PetShop 4.0基本上延续了3.0的结构， 但在性能上作了一定的改进，引入了缓存和异步处理机制， 同时又充分利用了ASP.Net 2.0的新功能MemberShip。
◆可以看到，在数据访问层中，完全采用了“面向接口编程”思想。抽象出来的I D A L模块，脱离了与具体数据库的依赖，从而使得整个数据访问层有利于数据库迁移。DALFactory模块专门管理D A L对象的创建，便于业务逻辑层访问。SQLServerDAL 和OracleDAL模块均实现I D A L模块的接口，其中包含的逻辑就是对数据库的Select、Insert、Update 和Delete操作。因为数据库类型的不同，对数据库的操作也有所不同，代码也会因此有所区别。

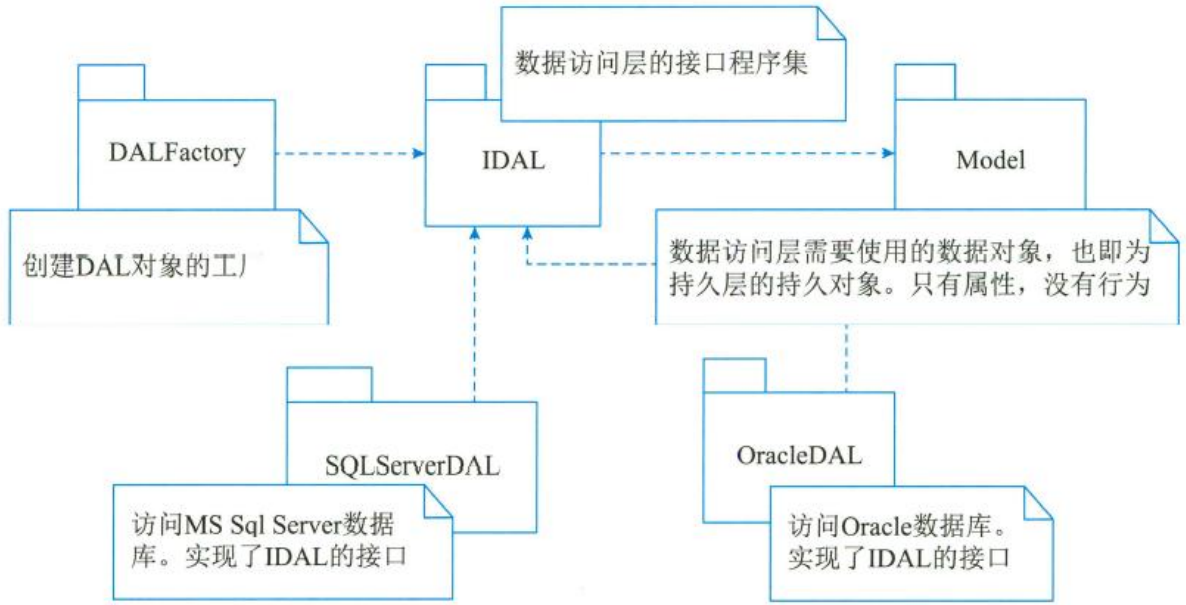



图 13-19　数据访问层的模块结构图

# 层次架构案例分析

◆此外，抽象出来的I D A L模块，除了解除了向下的依赖之外，对于其上的业务逻辑层同样仅存在弱依赖关系，如图13-20所示。

◆图13-20中， B L L 是业务逻辑层的核心模块，它包含了整个系统的核心业务。在业务逻辑层中，不能直接访问数据库，而必须通过数据访问层。注意，图13-20中对数据访问业务的调用，是通过接口模块I D A L来完成的。既然与具体的数据访问逻辑无关，则层与层之间的关系就是松散耦合的。如果此时需要修改数据访问层的具体实现，只要不涉及IDAL 的接口定义，那么业务逻辑层就不会受到任何影响。毕竟，具体实现的SQLServerDAL和OracalDAL根本就与业务逻辑层没有半点关系。

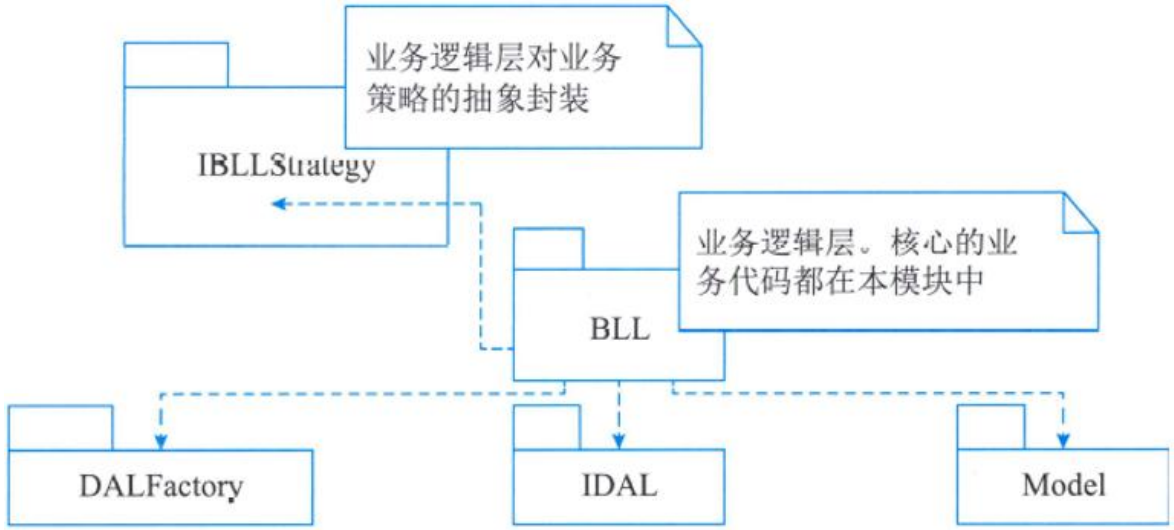



图 13-20　业务逻辑层的模块结构图

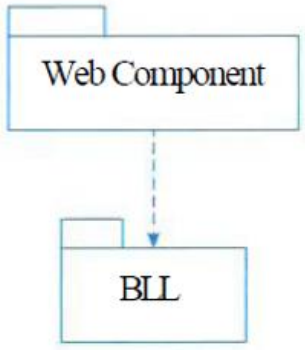



图13-21　表示层的楼块结构

# 典型真题

工作流管理联盟 (Workow Management Coalition)将工作流定义为:业务流程的全部或部分自动化，在此过程中，文档、信息或任务按照一定的过程规则流转，实现组织成员间的协调工作以达到业务的整体目标。工作流参考模型包括的组件是（ ）。
A．过程定义工具、工作流引擎、工作流客户端应用、相关应用、管理与监视工具.
B．工作流定义工具、工作流引擎、工作流客户端应用、相关应用、管理与监视工具
C．工作流定义工具、工作流引擎、工作流客户端应用、工作流 API、管理与监视工具
D．过程定义工具、工作流引擎、工作流客户端应用、工作流 API、管理与监视工具
解析：工作流参考模型包括的组件是过程定义工具、工作流引擎、工作流客户端应用、相关应用、管理与监视工具。
答案：A

物联网的感知层用于识别物体、采集信息。下列哪项不属于感知层设备（ ）。
A．摄像头　　B．GPS　　C．扫描仪　　D．指纹.
解析：感知层主要功能是识别对象、采集信息，与人体结构中皮肤和五官的作用类似。但 D选项指纹是人的特征属性，不是感知层设备。
答案：D

# 本章重点回顾

1、表现层技术
2、业务逻辑层组件
3、数据访问层五种技术
4、物联网三层

# THANKS