



软件工程
第三章 软件架构设计
3-2 基本架构风格



徐汉川
xhc@hit.edu.cn

2015年10月16日

主要内容

- 1. 什么是“架构风格”
- 2. 调用-返回风格
- 3. 以数据为中心的风格
- 4. 数据流风格
- 5. 分层结构
- 6. C/S、B/S和M/C
- 7. 事件风格
- 8. 模型-视图-控制器(MVC)
- *9. 面向服务的体系结构



1. 什么是“架构风格”

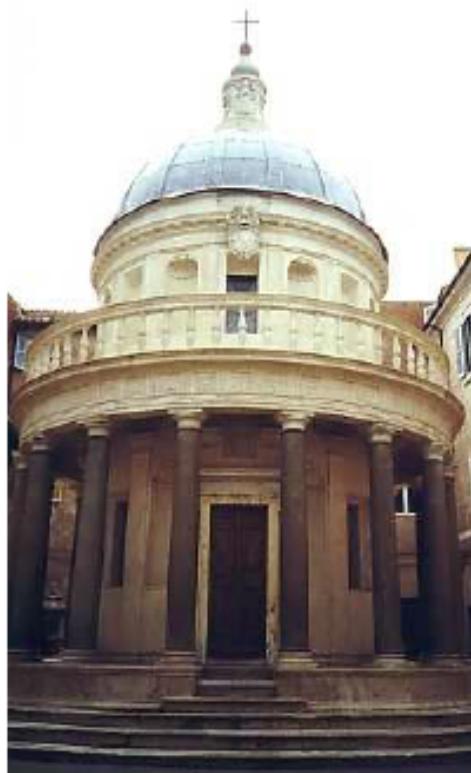


从“建筑风格”开始

- Architectural style constitutes a mode of classifying architecture largely by morphological characteristics in terms of form, techniques, materials, etc. (建筑风格等同于建筑体系结构的一种可分类的模式，通过诸如外形、技术和材料等形态上的特征加以区分)。
- 之所以称为“风格”，是因为经过长时间的实践，它们已经被证明具有良好的工艺可行性、性能与实用性，并可直接用来遵循与模仿（复用）。

Renaissance architecture (文艺复兴建筑风格)

- 结构清晰、规则，遵循简单的形状和数学比例
- 简单的圆柱、对称，区别于日后不规则的、复杂的多面建筑
- 古典风格的圆柱、完美的几何设计和半球形的屋顶



Baroque (巴洛克建筑风格)

- 起源于17世纪的意大利
- 狹窄的长条形主殿被替换为宽阔的圆形
- 较多的使用光影效果，具有强烈的明暗对比效果，或者使用均衡的光线
- 较多的使用装饰品
- 大范围的天花板壁画
- 从外部看，中央剧烈的突出
- 内部则经常只是用来绘画和雕塑



Gothic (哥特式风格)

- 强调垂直高度；石骨架结构；
- 不论是墙和塔都是越往上分划越细，装饰越多，也越玲珑，而且顶上都有锋利的、直刺苍穹的小尖顶；
- 丛生的圆柱、悬空的扶壁、有龙骨的拱顶、尖顶形状的圆弧拱门；
- 优美的彩色玻璃窗画；
- 强调雕塑的细节，多采用圆雕和接近圆雕的高浮雕；



中国古典建筑



宫殿风格



江南建筑风格



园林风格

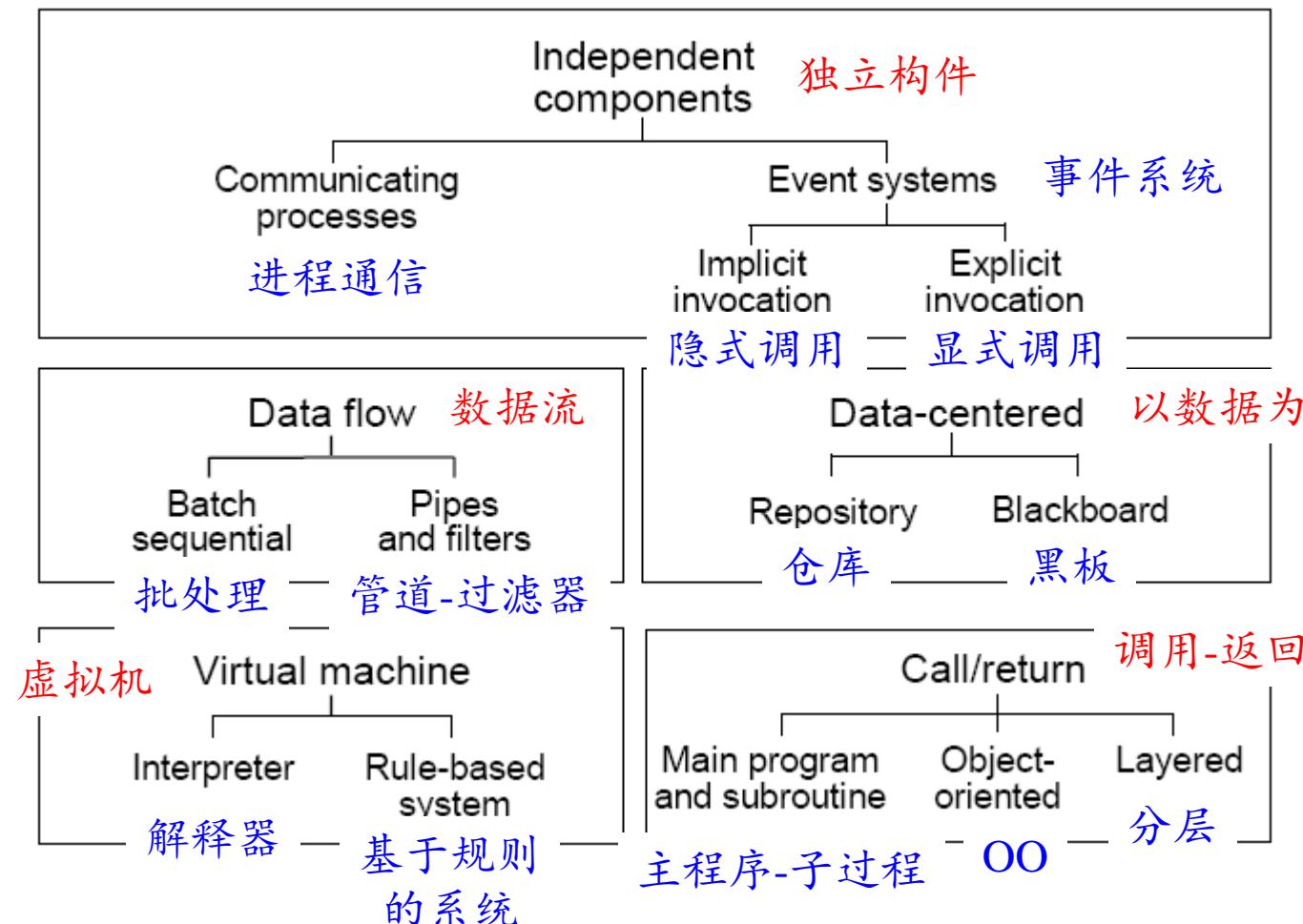
软件体系结构风格

- 软件系统同建筑一样，也具有若干特定的“风格” (software architectural style):
 - 这些风格在实践中被多次设计、应用，已被证明具有良好的性能、可行性和广泛的应用场景，可以被重复使用；
 - 实现“软件体系结构级”的复用。
- 定义：
 - 描述特定领域中软件系统家族的组织方式的惯用模式，反映了领域中众多系统所**共有的结构和语义特性**，并指导如何将各个模块和子系统有效地组织成一个完整的系统。

“软件架构风格”的组成

- A set of component types (e.g., data repository, process, object) (一组构件类型)
- A set of connector types/interaction mechanisms (e.g., subroutine call, event, pipe) (一组连接件类型/交互机制)
- A topological layout of these components (这些构件的拓扑分布)
- A set of constraints on topology and behavior (e.g., a data repository is not allowed to change stored values, pipelines are acyclic) (一组对拓扑和行为的约束)
- An informal description of the costs and benefits of the style, e.g.: “Use the pipe and filter style when reuse is desired and performance is not a top priority” (一些对风格的成本和收益的描述)

经典架构风格



新型的架构风格
点对点(P2P)
网格(grid)
Web 2.0
面向服务的架构
(SOA)
事件驱动的架构
(EDA)
软件即服务(SaaS)
云计算与虚拟化

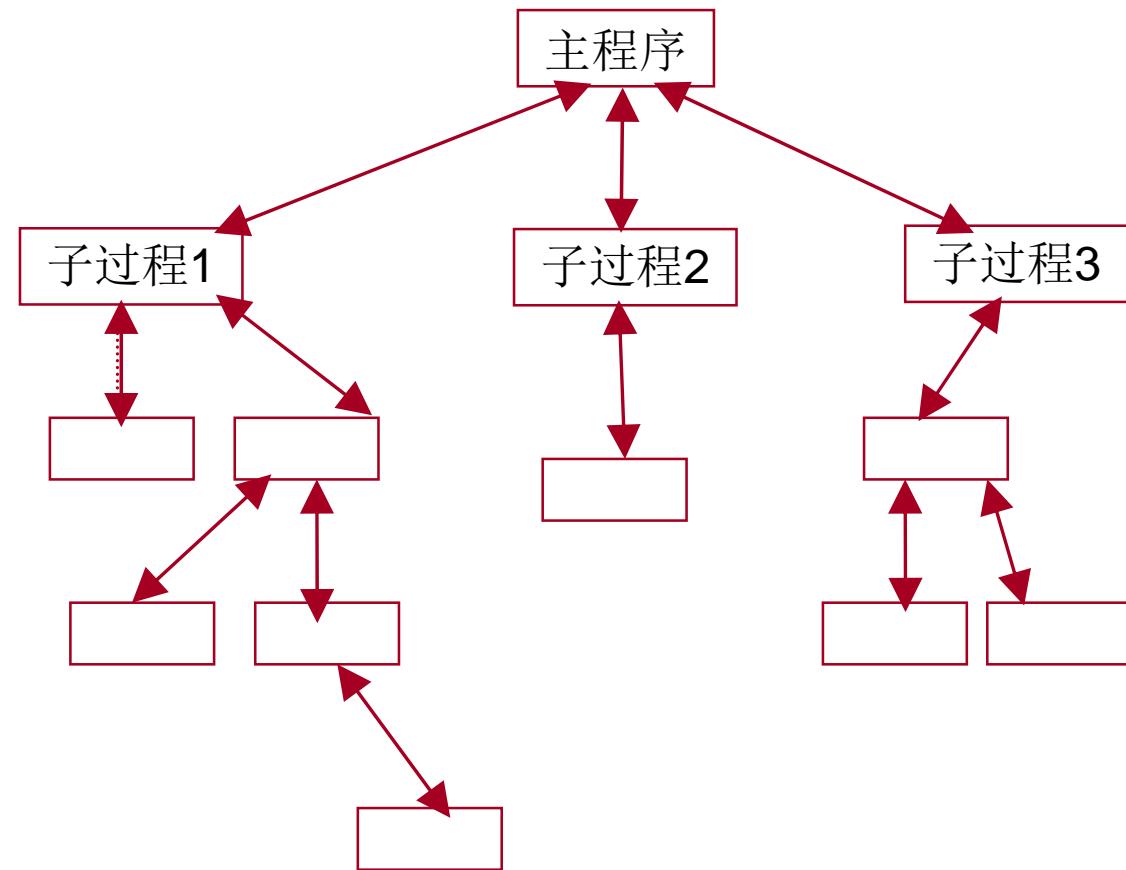


2. 调用-返回风格

以函数/对象作为基本构造单元，彼此通过call-return相互连接
特征：不考虑分层，主要遵循同步调用方式

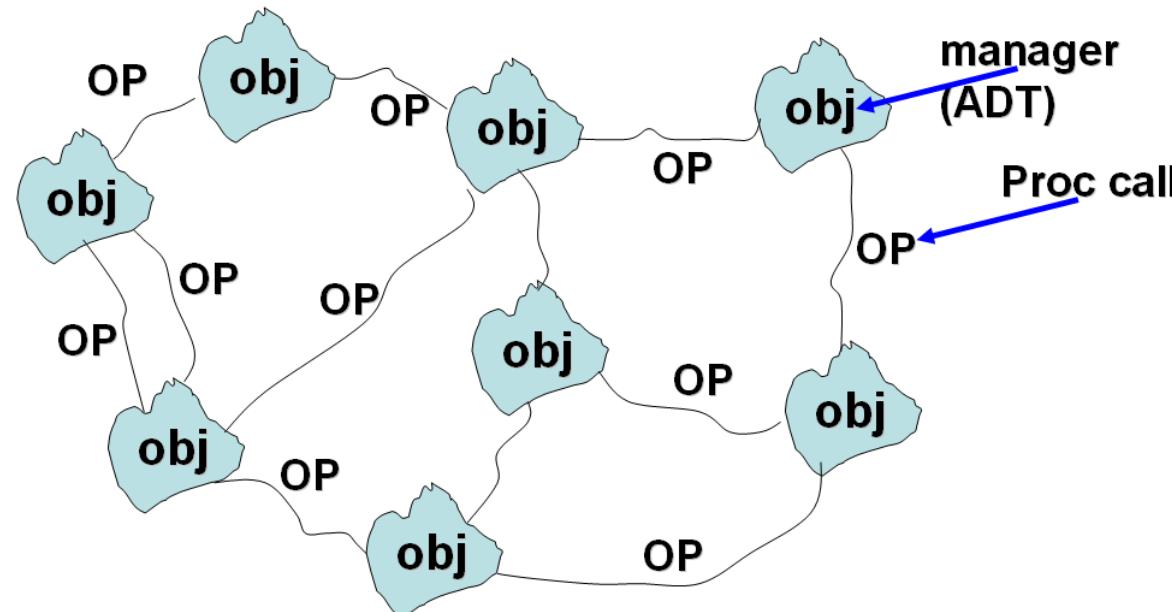
主程序-子过程

- 该风格是结构化程序设计的一种典型风格，从功能的观点设计系统，通过逐步分解和逐步细化，得到系统架构。
 - 构件：主程序、子程序
 - 连接器：调用-返回机制
 - 拓扑结构：层次化结构
- 本质：将大系统分解为若干模块(模块化)，主程序调用这些模块实现完整的系统功能。



面向对象风格

- 系统被看作对象的集合，每个对象都有一个它自己的功能集合；
- 数据及作用在数据上的操作被封装成抽象数据类型(ADT)；
- 只通过接口与外界交互，内部的设计决策则被封装起来
 - 构件：类和对象
 - 连接件：对象之间通过函数调用、消息传递实现交互

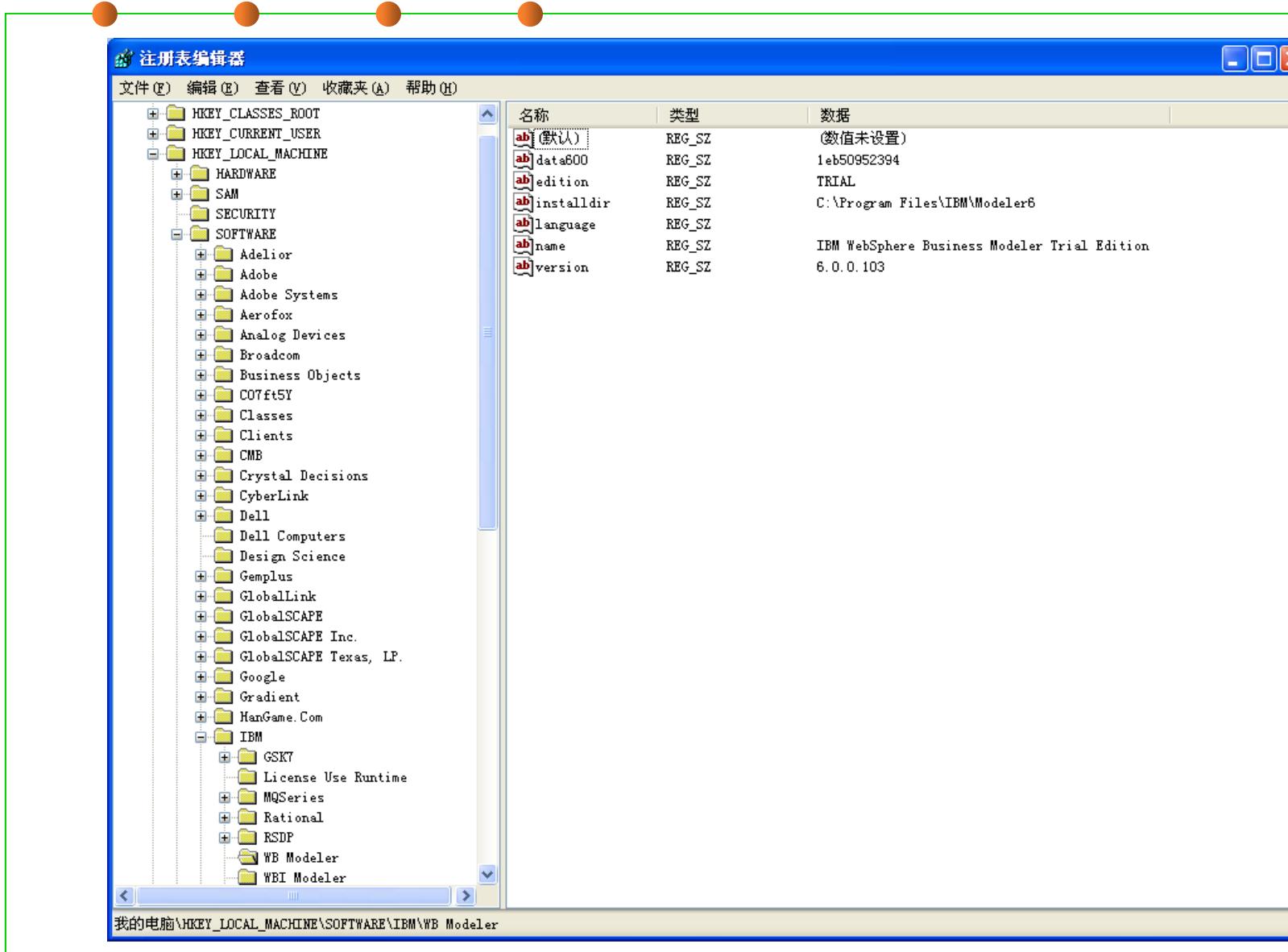




3. 以数据为中心的风格

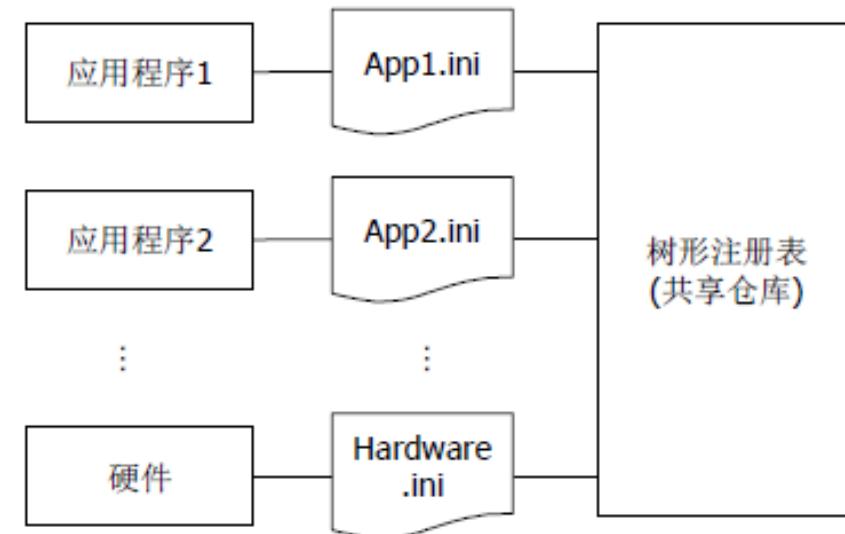
数据的持久化存储被分离出去，形成两层结构

例1：注册表(Windows Registry)



注册表的结构

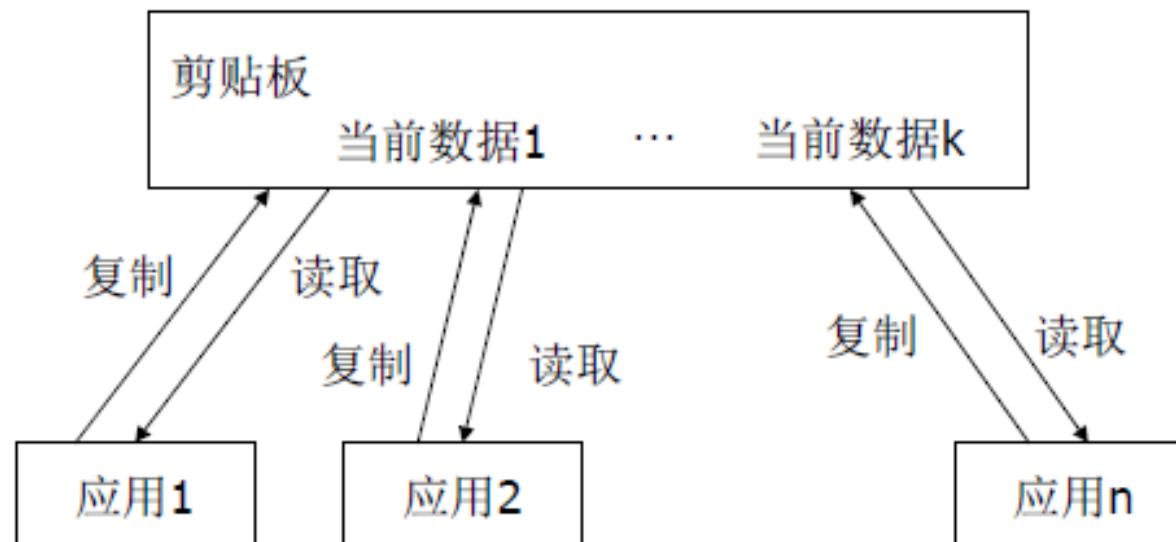
- 最初，硬件/软件系统的配置信息均被各自保存在一个配置文件中(.ini)；
- 这些文件散落在系统的各个角落，很难对其进行维护；
- 为此，引入注册表的思想，将所有.ini文件集中起来，形成共享仓库，为系统运行起到了集中的资源配置管理和控制调度的作用。



- 注册表中存在着系统的所有硬件和软件配置信息，如启动信息、用户、**BIOS**、各类硬件、网络、**INI**文件、驱动程序、应用程序等；
- 注册表信息影响或控制系统/应用软件的行为，应用软件安装/运行/卸载时对其进行添加/修改/删除信息，以达到改变系统功能和控制软件运行的目的。

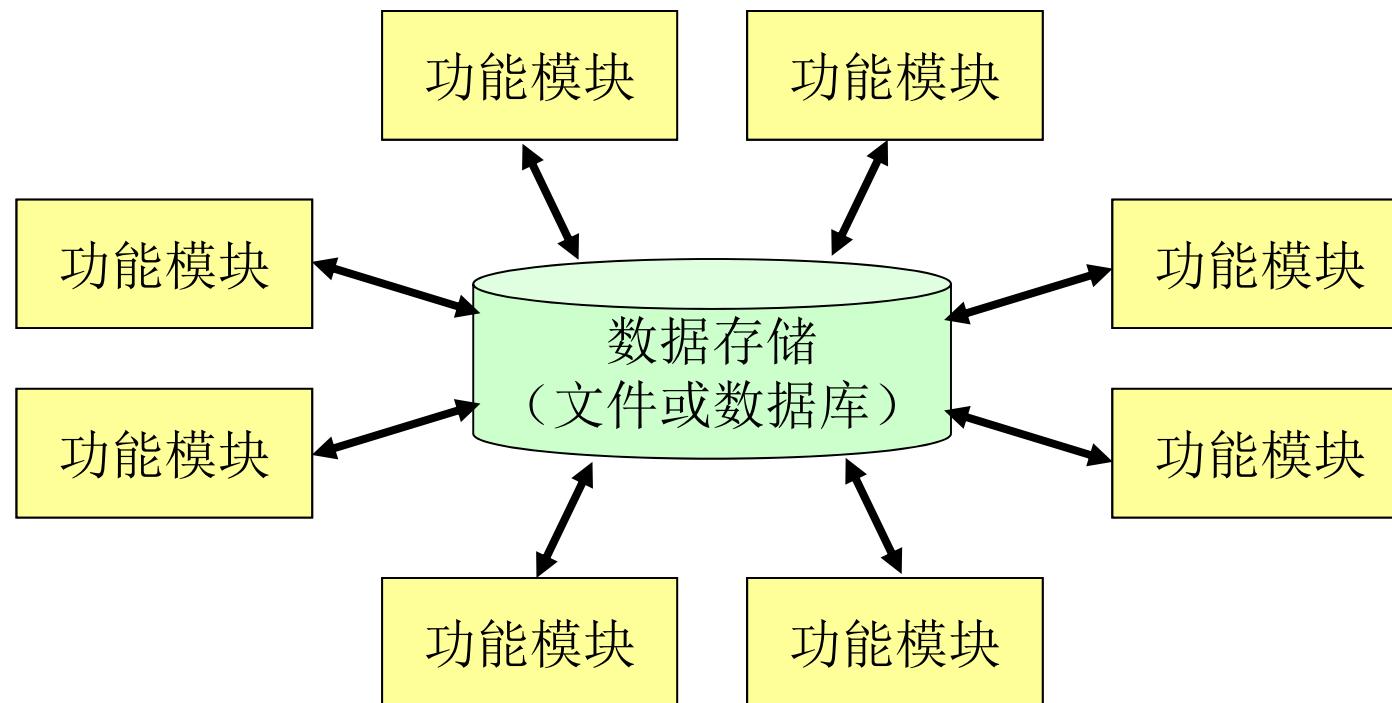
例2：剪贴板 (Clipboard)

- 剪贴板是一个用来进行短时间的数据存储并在文档/应用之间进行数据传递和交换的软件程序
 - 用来存储带传递和交换信息的公共区域(形成共享数据仓库);
 - 不同的应用程序通过该区域交换格式化的信息;
 - 访问剪贴板的方式: Copy & Paste/Cut



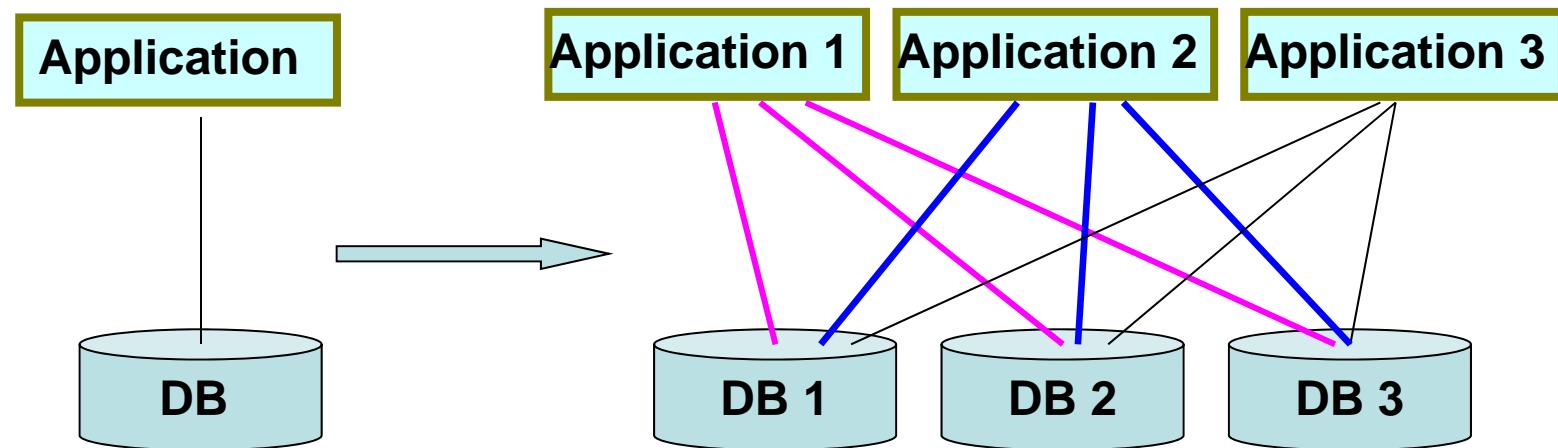
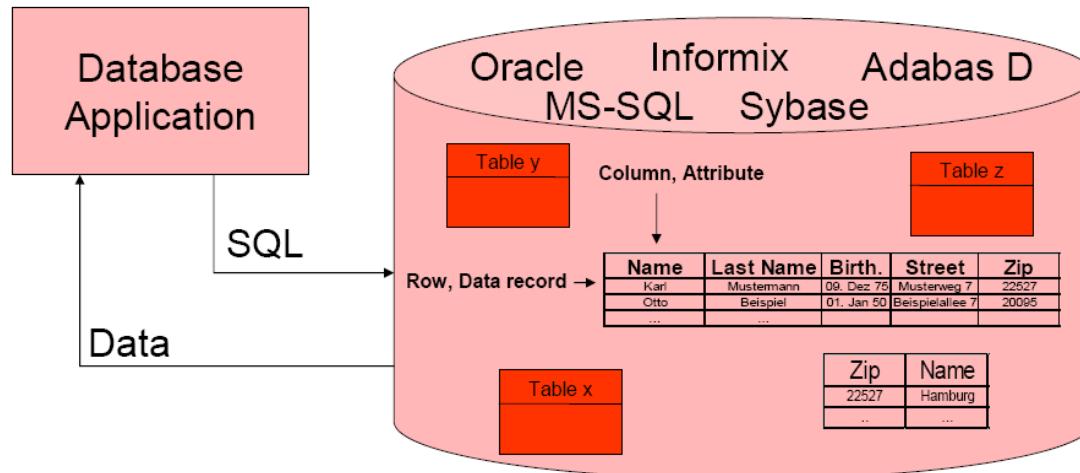
以数据为中心的体系结构风格

- 以数据为中心的体系结构风格
 - 仓库风格（数据被动存储）
 - 黑板风格（仓库风格的变体，数据发生变化后，主动通知相关用户）

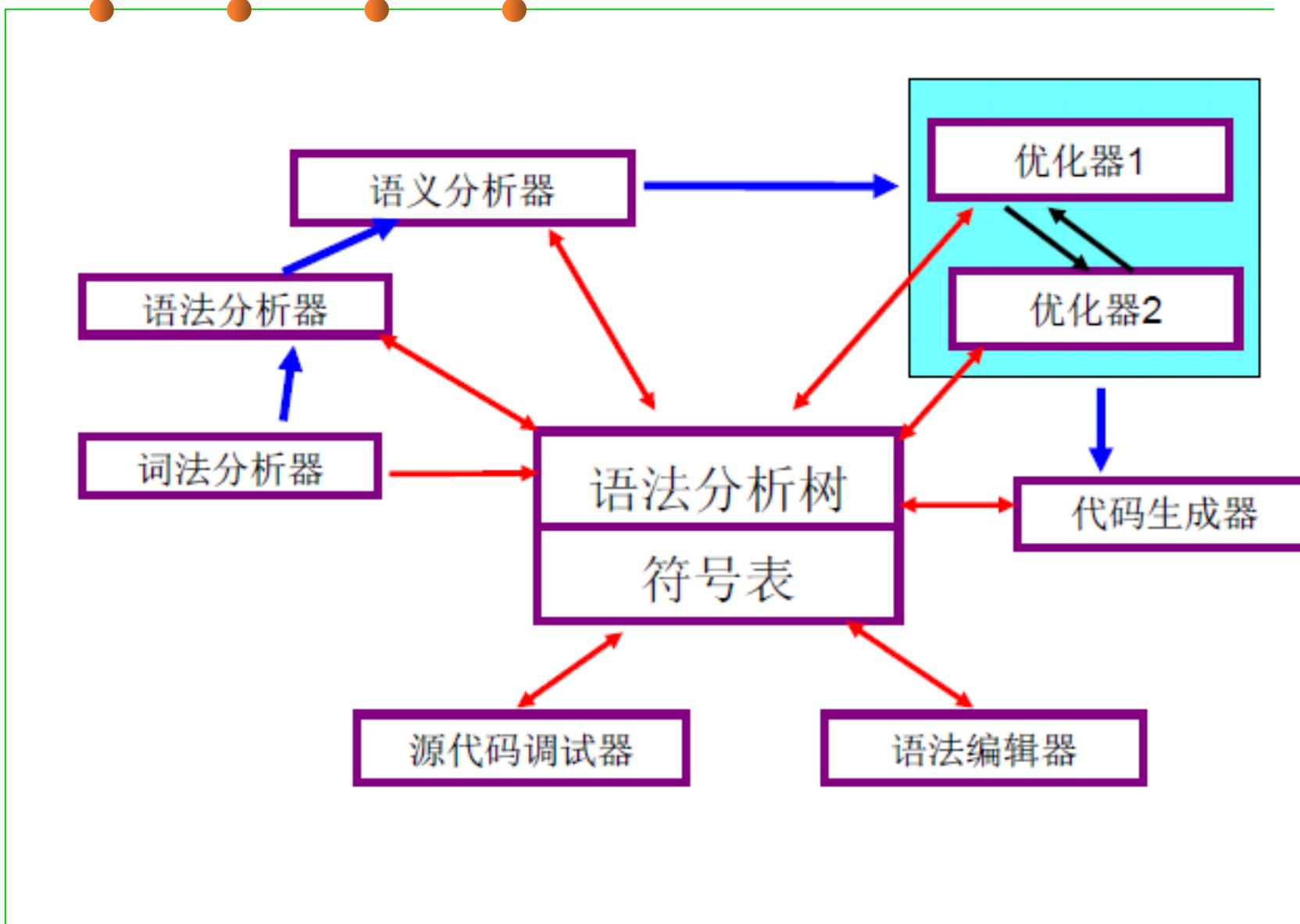


示例1：基于数据库的系统结构

VB、PB等开发的典型信息系统软件



示例2：仓库形式的编译器结构





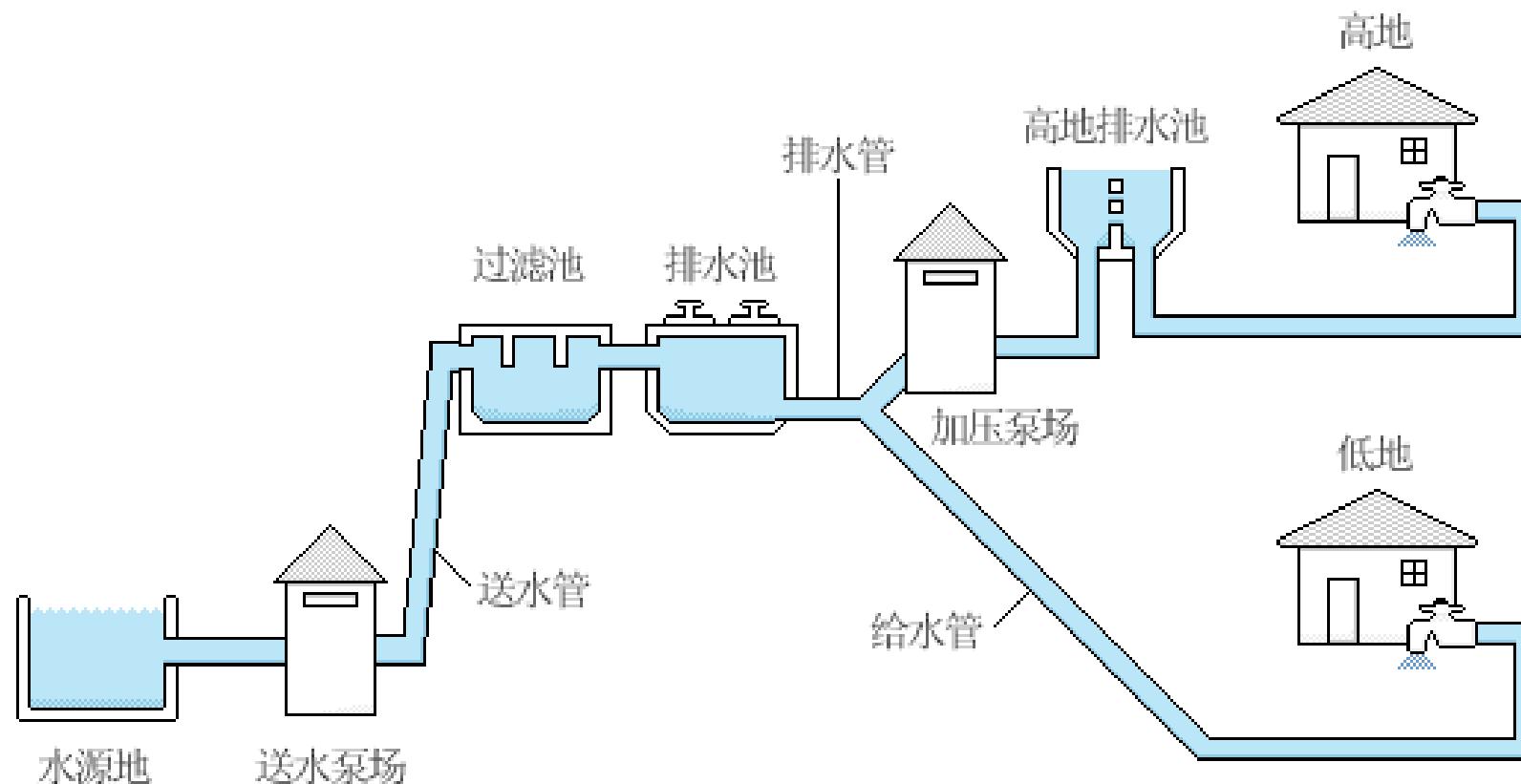
4. 数据流风格



把系统分解为几个序贯的处理步骤，这些步骤之间通过数据流连接，一个步骤的输出是另一个步骤的输入；
每个处理步骤由一个过滤器构件(Filter)实现；
处理步骤之间的数据传输由管道(Pipe)负责

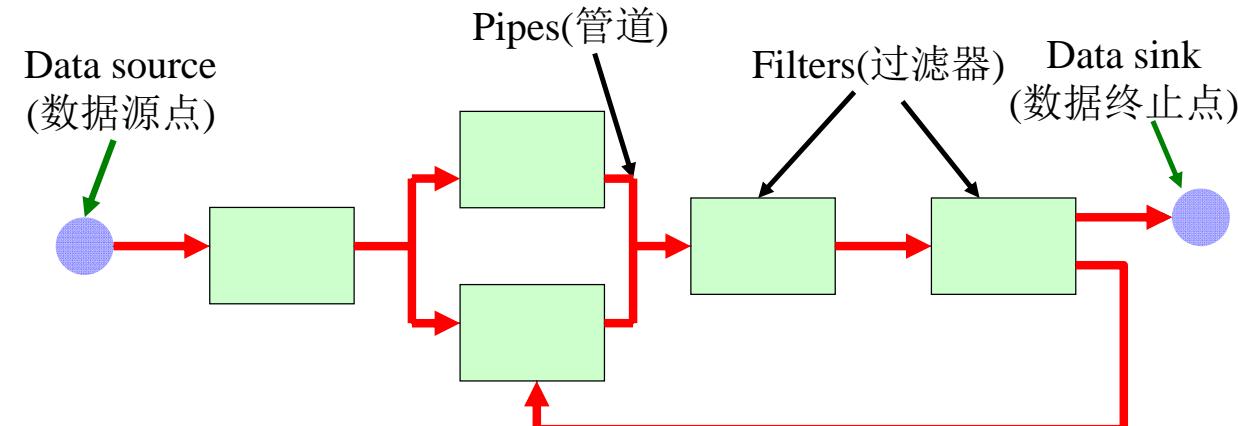
数据流风格

- 现实中的“数据流”体系结构

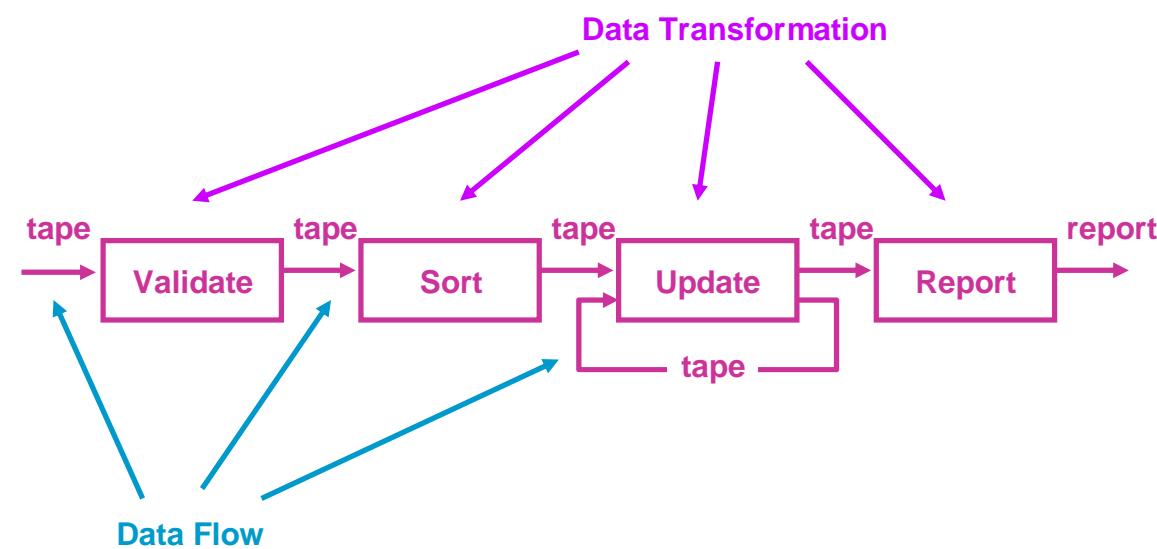


软件中的数据流风格

- 管道-过滤器风格

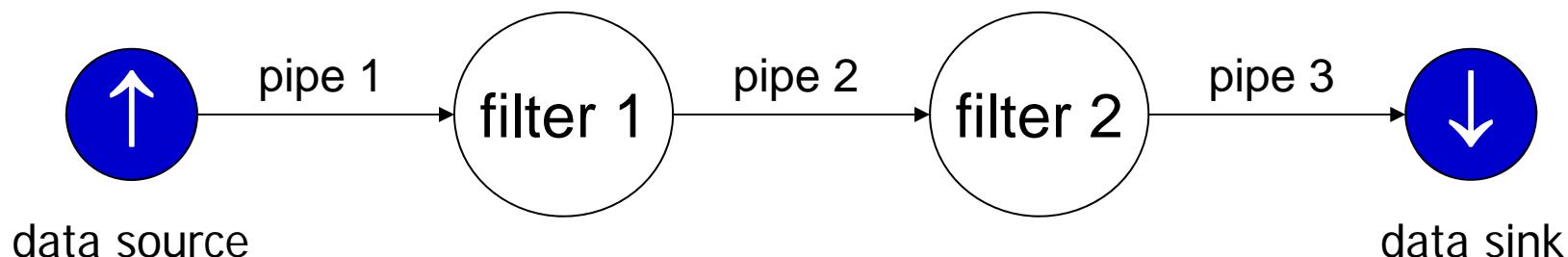


- 顺序批处理风格

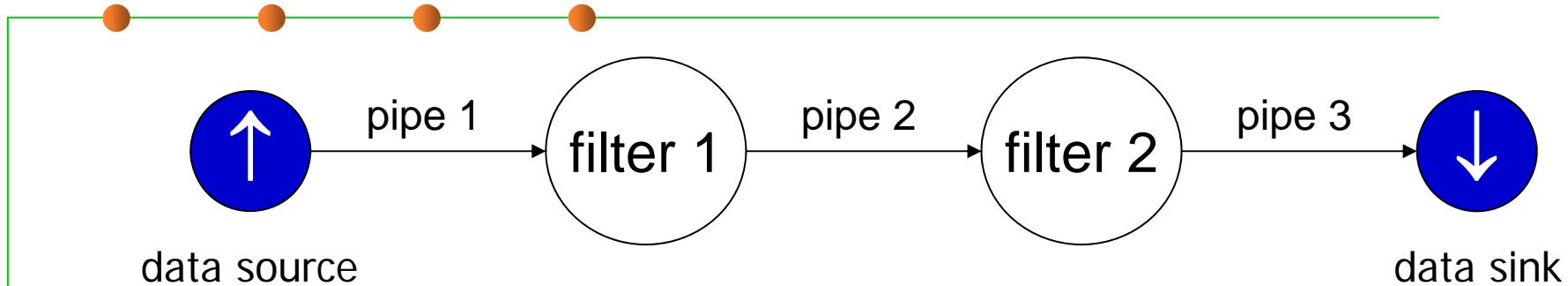


软件中的数据流风格

- 语境：数据源源不断的产生，系统需要对这些数据进行若干处理(分析、计算、转换等)。
- 解决方案：
 - 把系统分解为几个序贯的处理步骤，这些步骤之间通过数据流连接，一个步骤的输出是另一个步骤的输入；
 - 每个处理步骤由一个过滤器构件(Filter)实现；
 - 处理步骤之间的数据传输由管道(Pipe)负责。
- 每个处理步骤(过滤器)都有一组输入和输出，过滤器从管道中读取输入的数据流，经过内部处理，然后产生输出数据流并写入管道中。



软件中的数据流风格



filter1 {

 Read data d from pipe1;
 Deal with d and transform it to d' ;
 Write d' to pipe2;

}

filter2 {

 Read data d' from pipe2;
 Deal with d' and transform it to d'' ;
 Write d'' to pipe3;

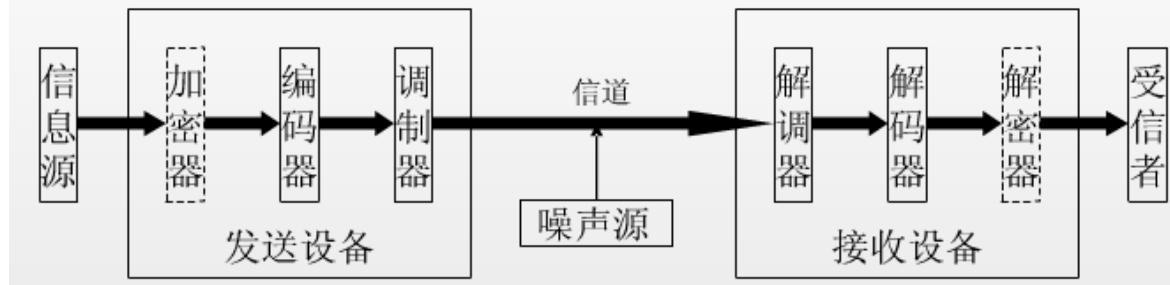
}

现实中的典型应用领域：

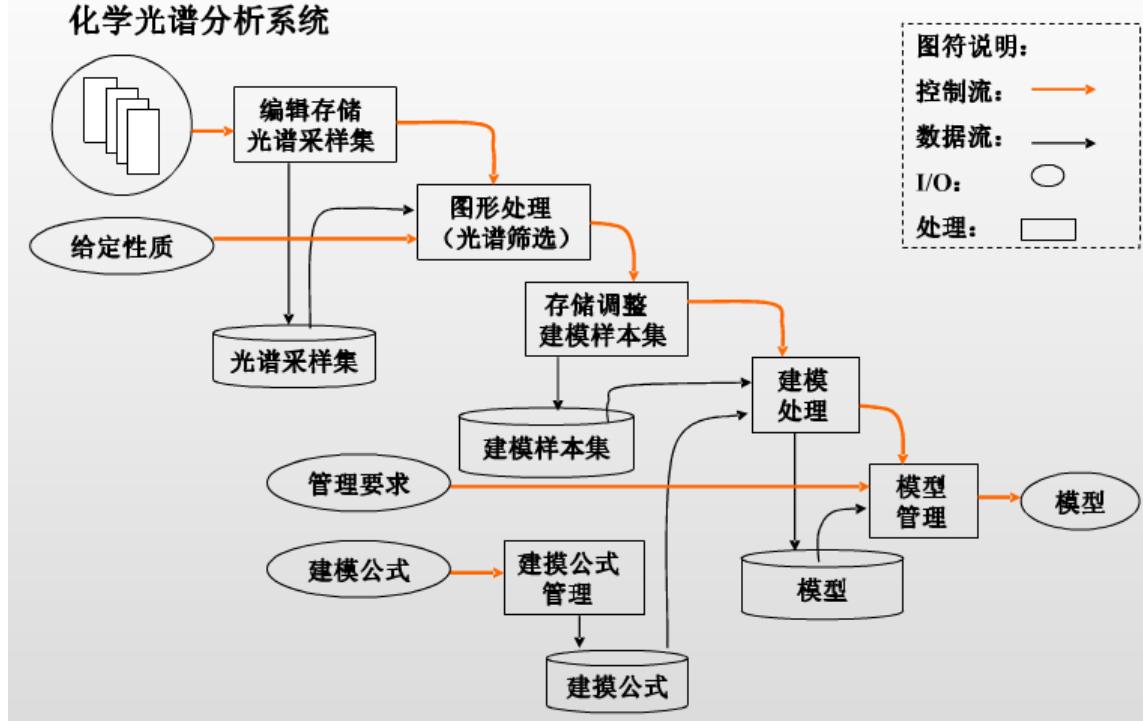
编译器、Unix管道、图像处理、信号处理、网络监控与管理等

软件中的数据流风格

数字通信系统结构简略描述:



化学光谱分析系统



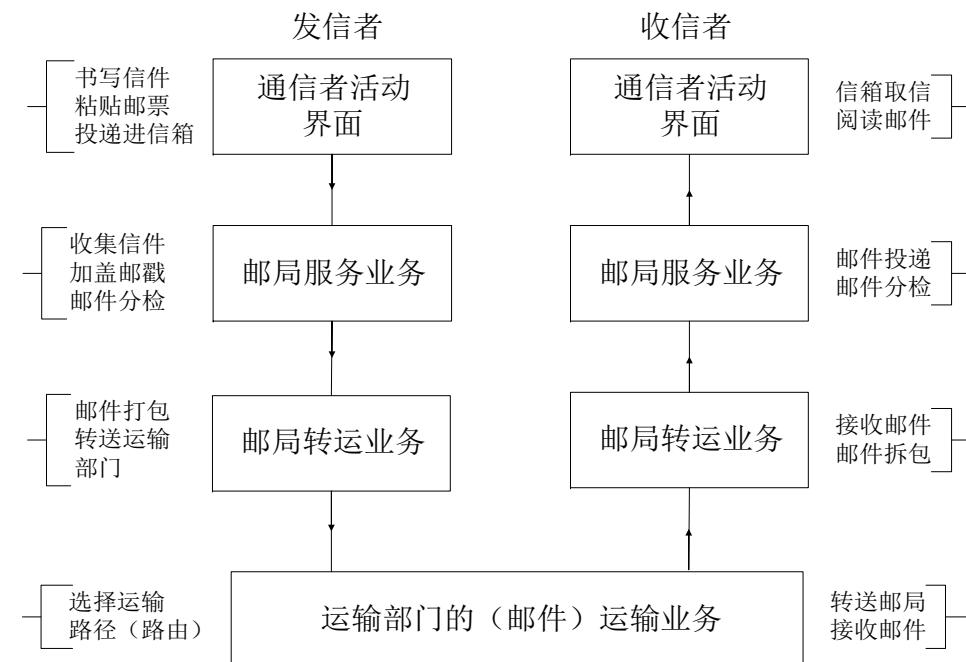


5. 层次风格

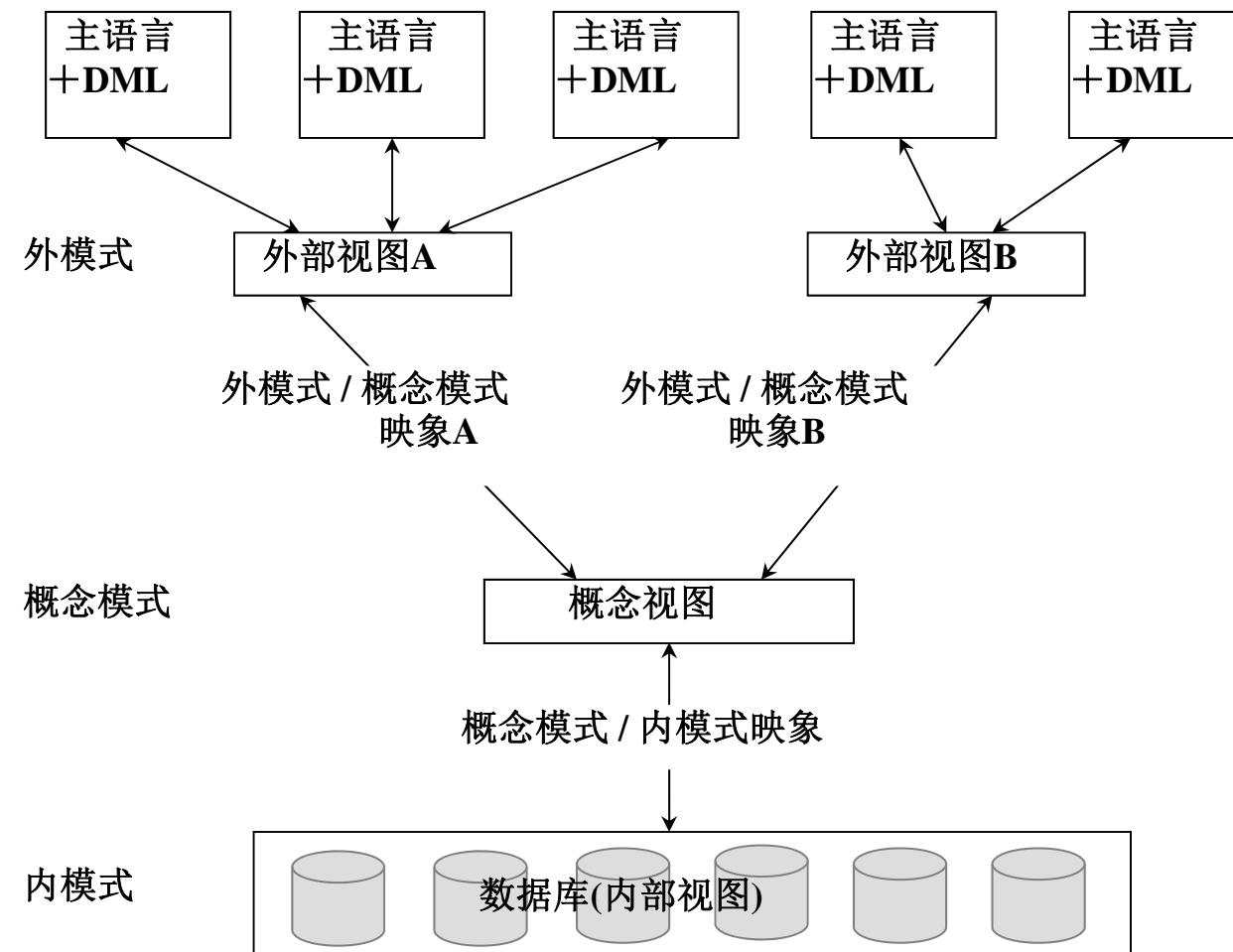
除了数据被分离出去，软件系统的其他各部分进一步分离，形成更复杂的层次结构

层次结构

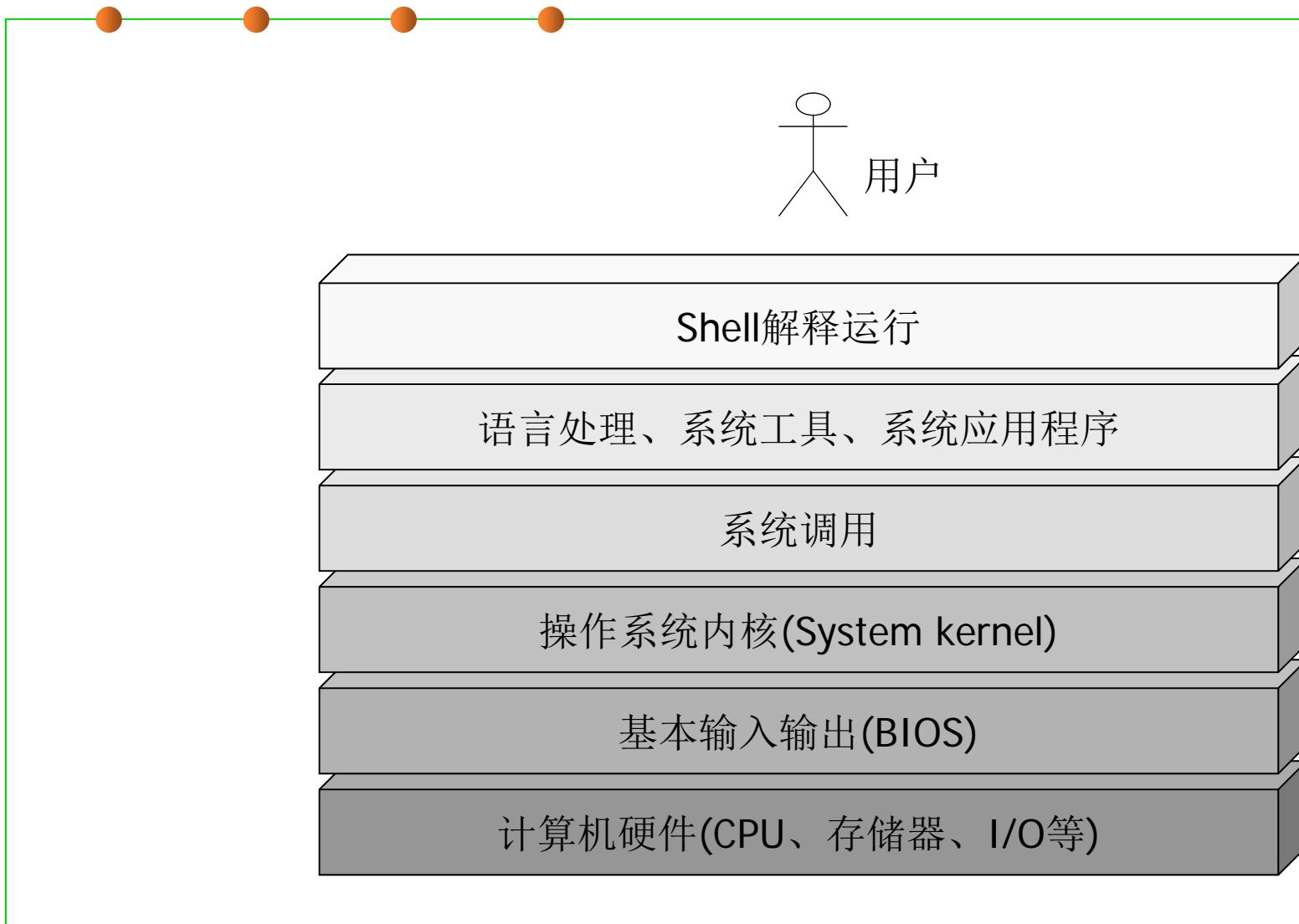
- 层次化早已经成为一种复杂系统设计的普遍性原则；
- 两个方面的原因：
 - 事物天生就是从简单的、基础的层次开始发生的；
 - 众多复杂软件设计的实践，大到操作系统，中到网络系统，小到一般应用，几乎都是以层次化结构建立起来的。



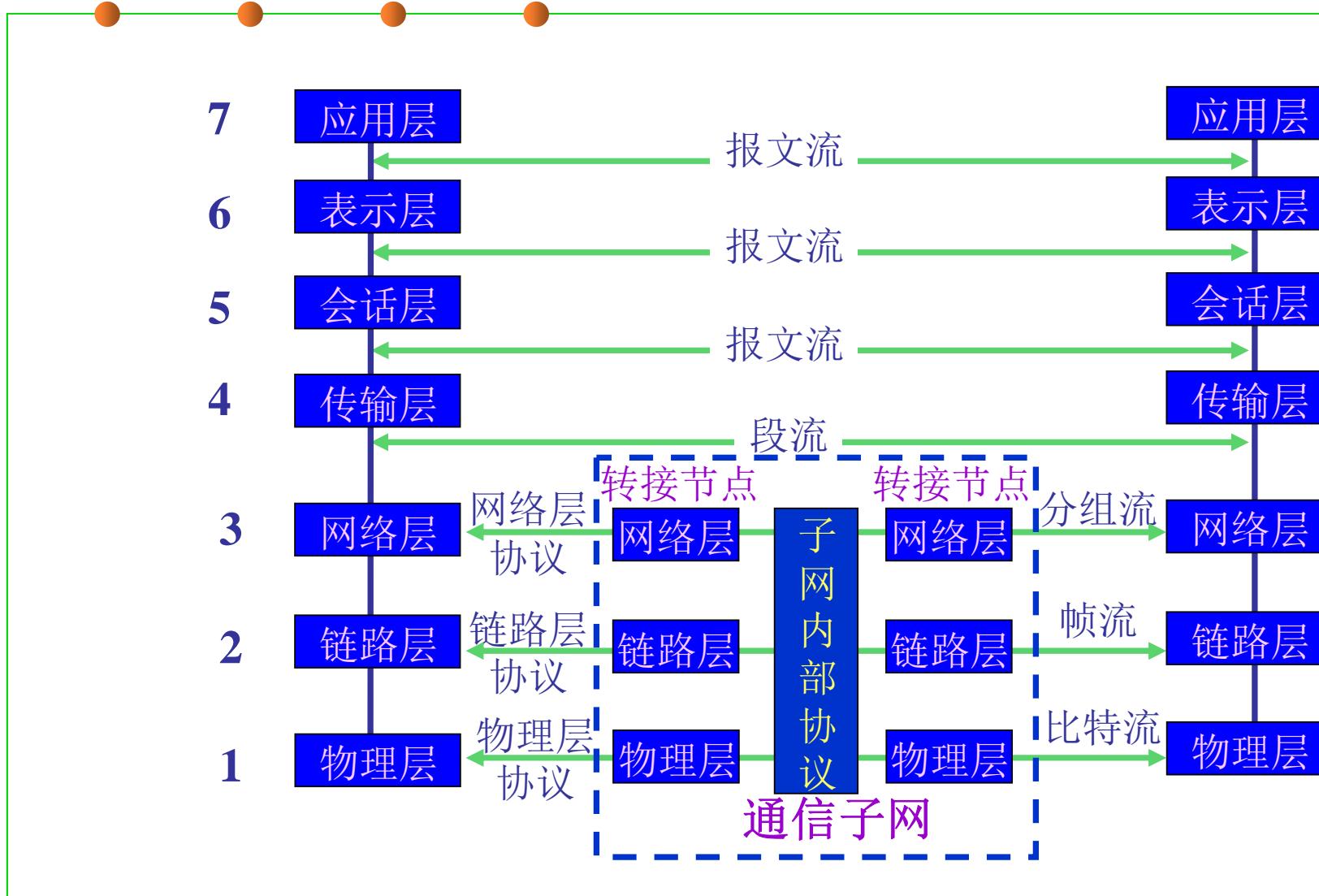
DBMS中的“三级模式-两层映像”



计算机操作系统的层次结构

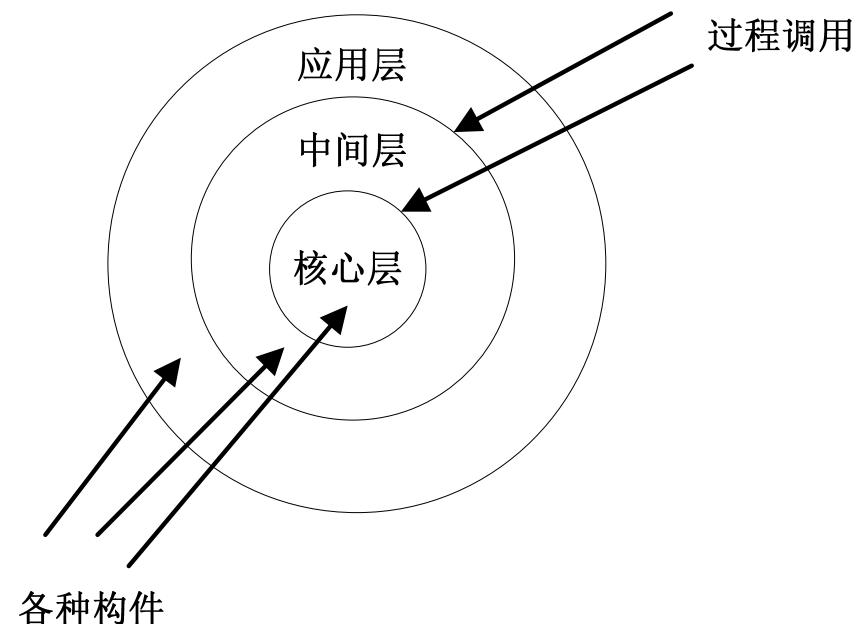


网络的分层模型



层次系统

- 在层次系统中，系统被组织成若干个层次，每个层次由一系列构件组成；
- 层次之间存在接口，通过接口形成**call/return**的关系
 - 下层构件向上层构件提供服务
 - 上层构件被看作是下层构件的客户端

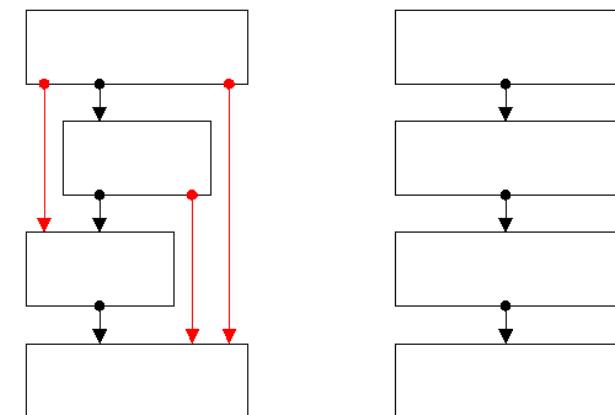


层次系统的优点

-
-
-
-
- 这种风格支持基于可增加抽象层的设计，允许将一个复杂问题分解成一个增量步骤序列的实现。
- 不同的层次处于不同的抽象级别：
 - 越靠近底层，抽象级别越高；
 - 越靠近顶层，抽象级别越低；
- 由于每一层最多只影响两层，同时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，同样为软件复用提供了强大的支持。

严格分层和松散分层

- 严格分层系统要求严格遵循分层原则，限制一层中的构件只能与对等实体以及与它紧邻的下面一层进行交互
 - 优点：修改时的简单性
 - 缺点：效率低下
- 松散的分层应用程序放宽了此限制，它允许构件与位于它下面的任意层中的组件进行交互
 - 优点：效率高
 - 缺点：修改时困难



- 任何计算机科学问题都可以通过增加一层来解决！
- 任何计算机性能问题都可以通过去掉一个间接层来解决！



6. C/S、B/S和M/C结构

几种经典的分层结构

“客户机-服务器”体系结构

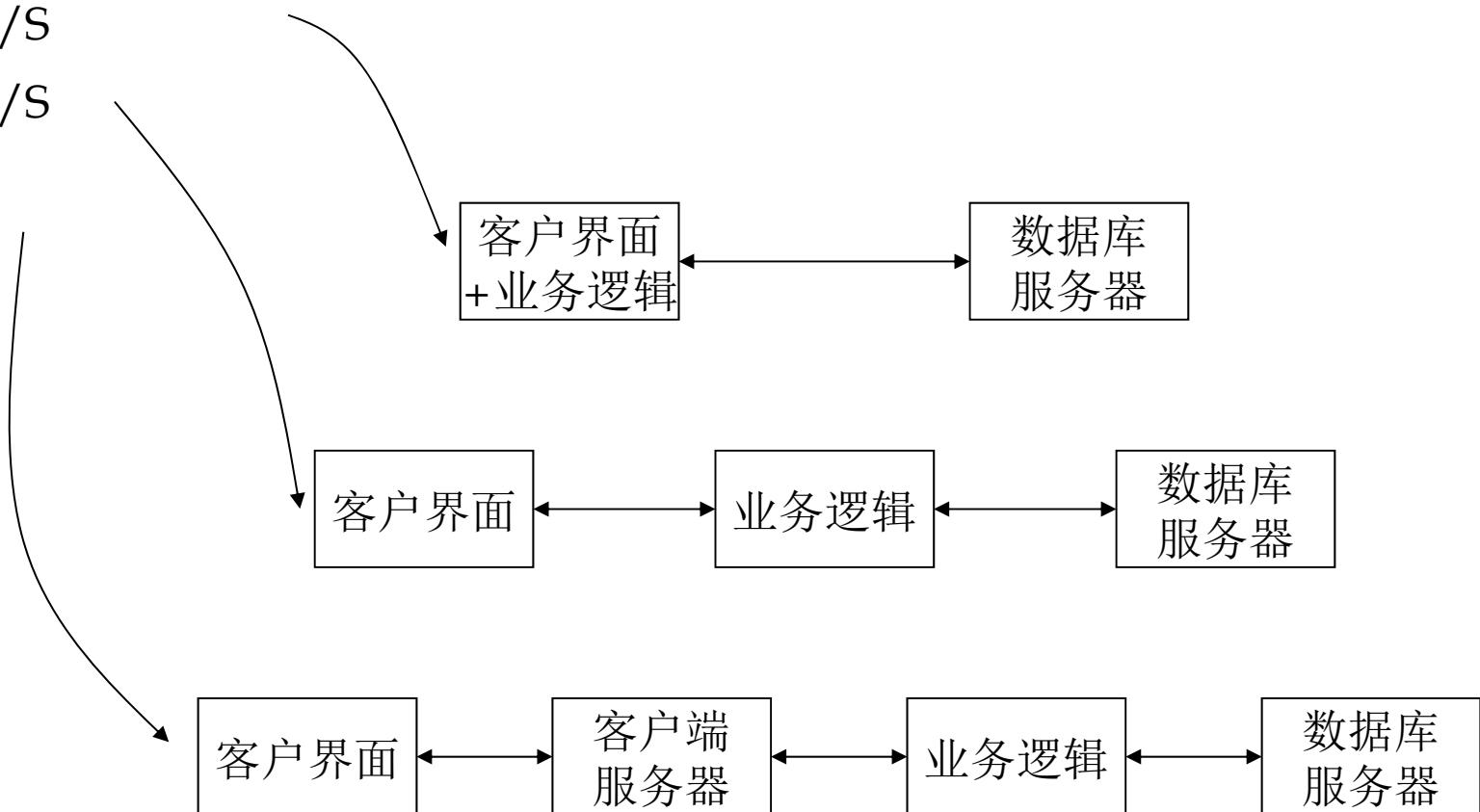
- 客户机/服务器(Client/Server, C/S): 一个应用系统被分为两个逻辑上分离的部分，每一部分充当不同的角色、完成不同的功能，多台计算机共同完成统一的任务。
 - 客户机(前端, front-end): 业务逻辑、与服务器通讯的接口；
 - 服务器(后端, back-end): 与客户机通讯的接口、业务逻辑、数据管理。
- 一般的，
 - 客户机为完成特定的工作向服务器发出请求；
 - 服务器处理客户机的请求并返回结果。



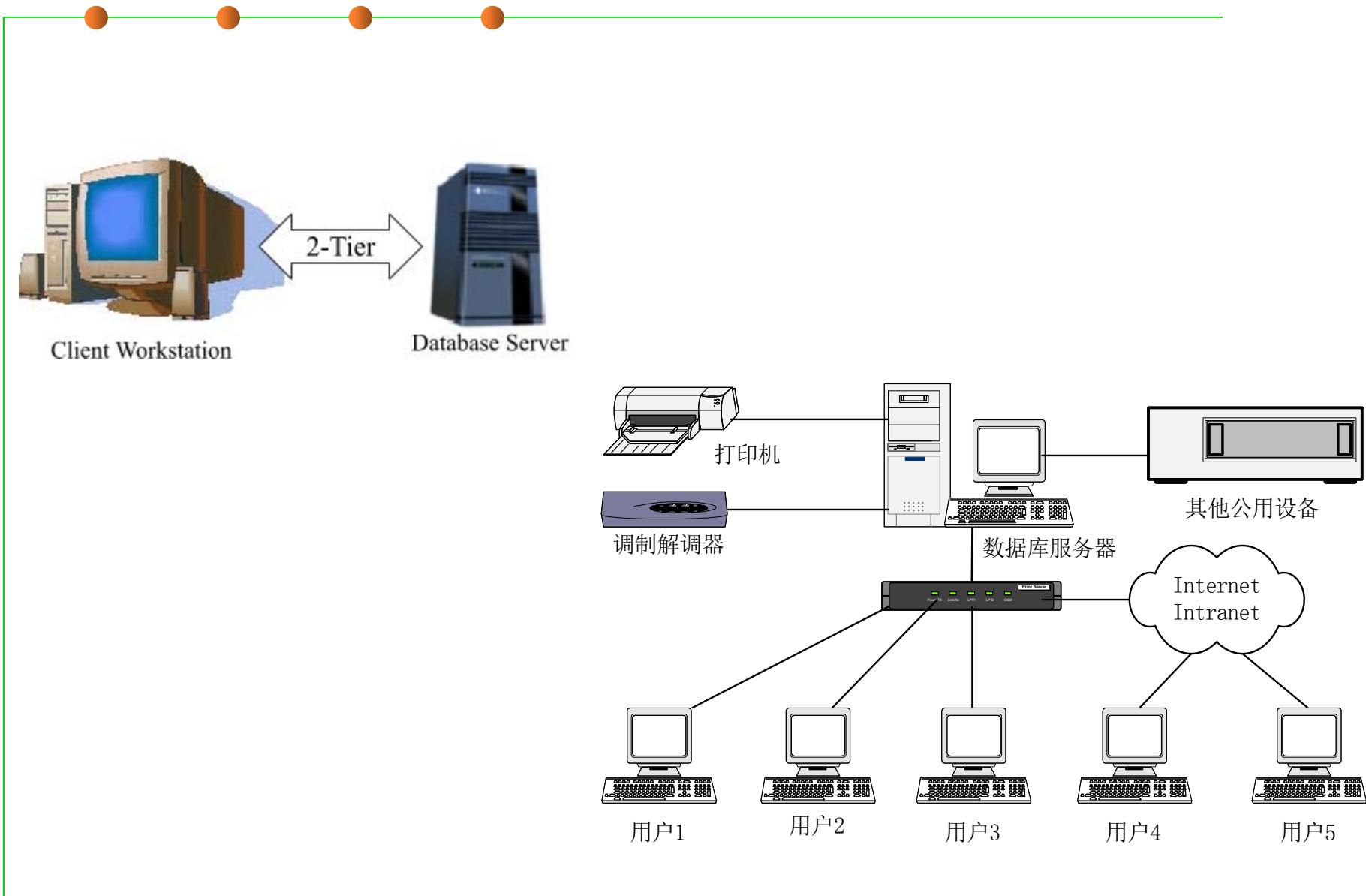
客户机/服务器的层次性

- “客户机-服务器”结构的发展历程：

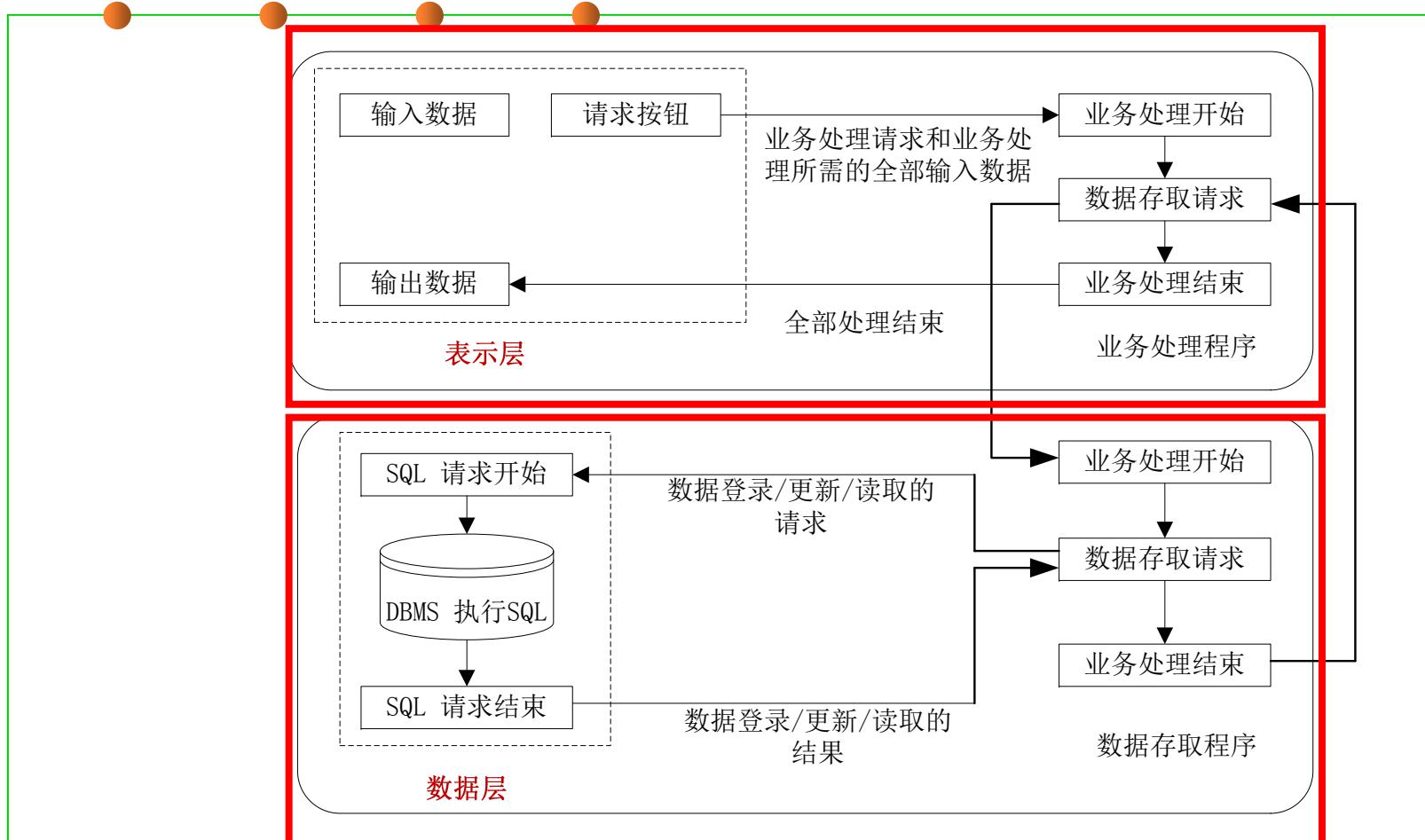
- 两层C/S（仓库体系风格）
- 三层C/S
- 多层C/S



两层C/S结构



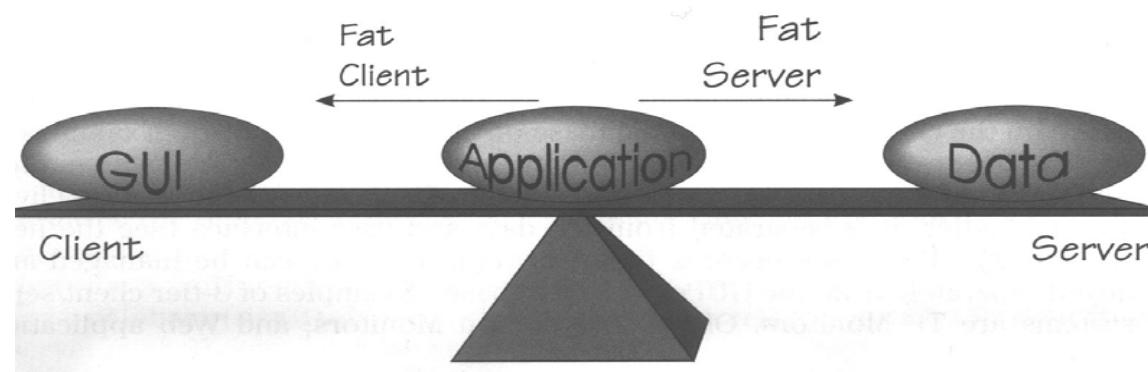
两层C/S结构



两层结构存在的问题？

胖客户端与瘦客户端

- 业务逻辑的划分比重：在客户端多一些还是在服务器端多一些？
 - 胖客户端：客户端执行大部分的数据处理操作
 - 瘦客户端：客户端具有很少或没有业务逻辑

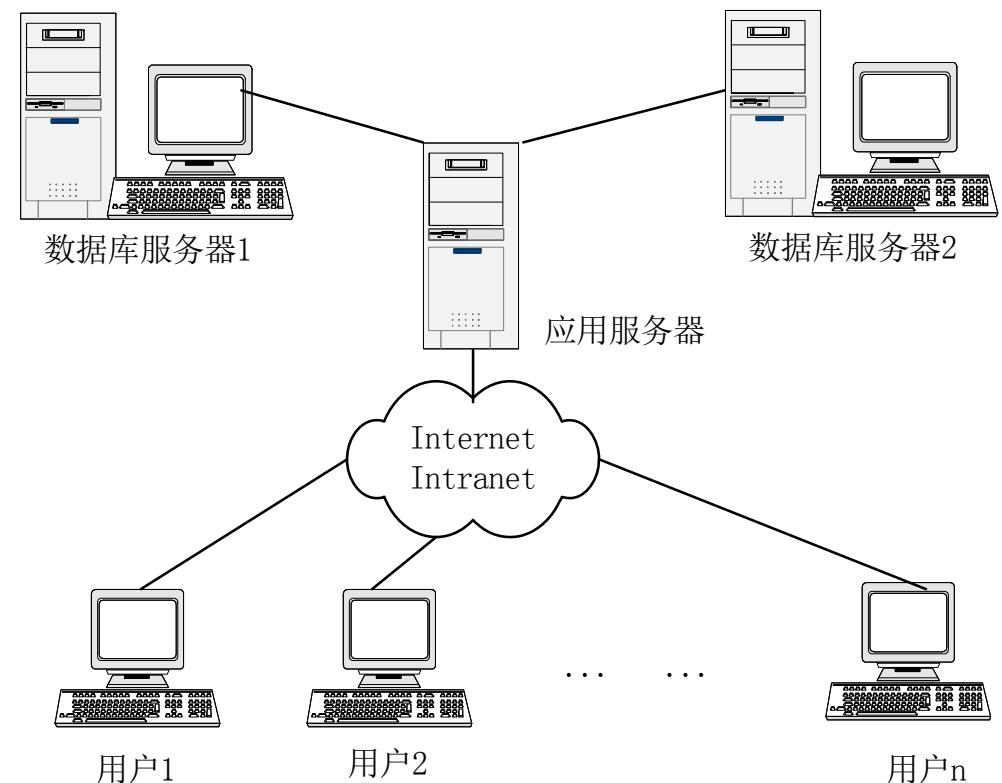


瘦客户端存在的问题？

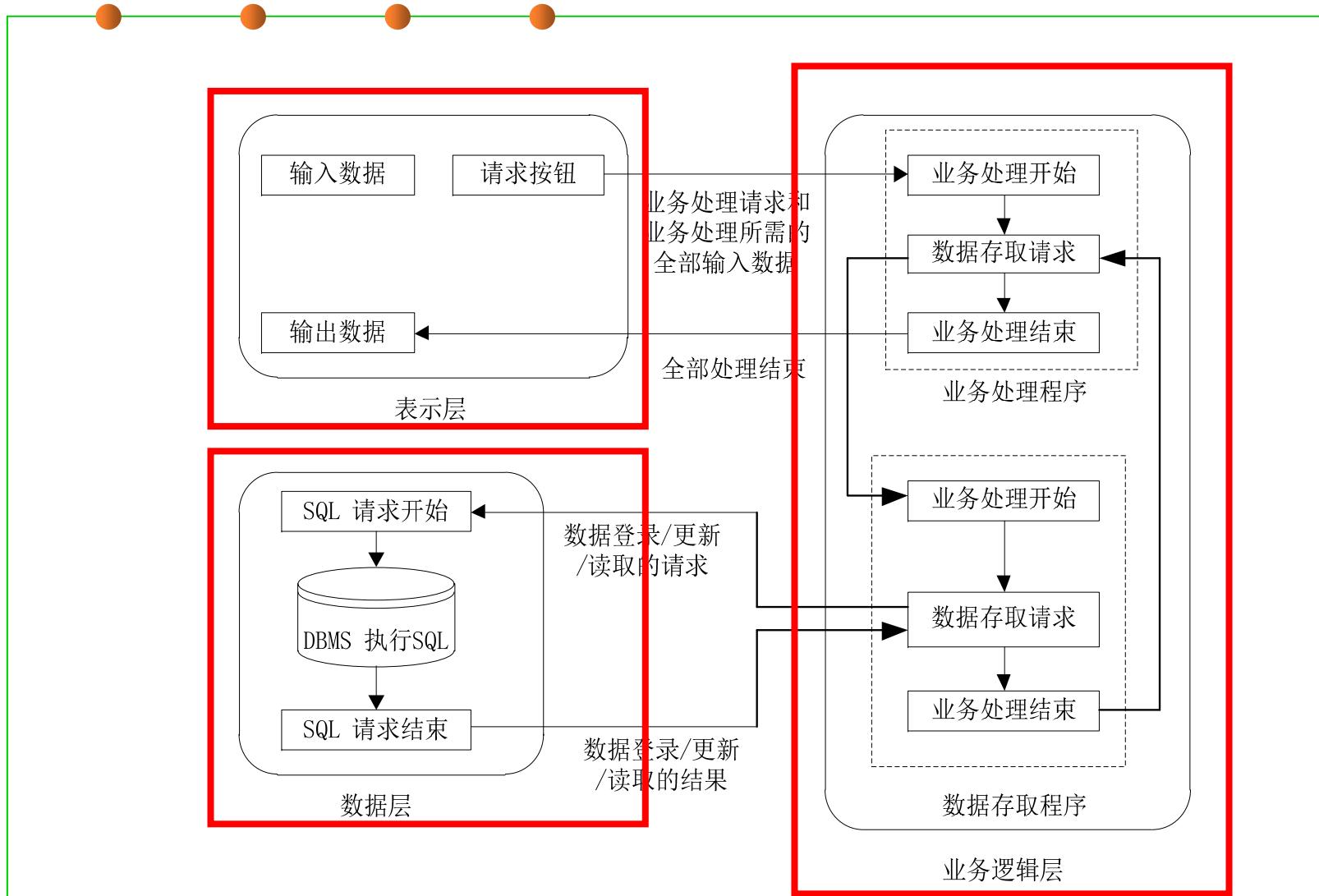
RIA (Rich Internet Application)
—富客户端

三层C/S体系结构

- 在客户端与数据库服务器之间增加了一个中间层
 - 表示层：用户界面—界面设计
 - 业务逻辑层：业务处理—程序设计
 - 数据层：数据存储—数据库设计

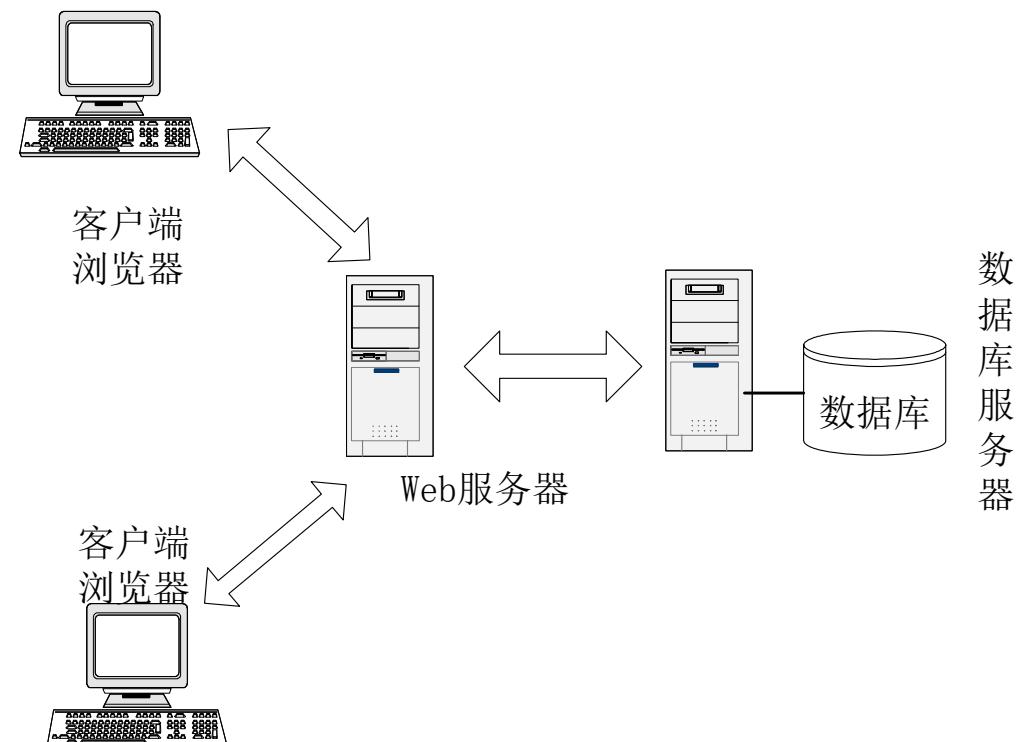


三层C/S结构

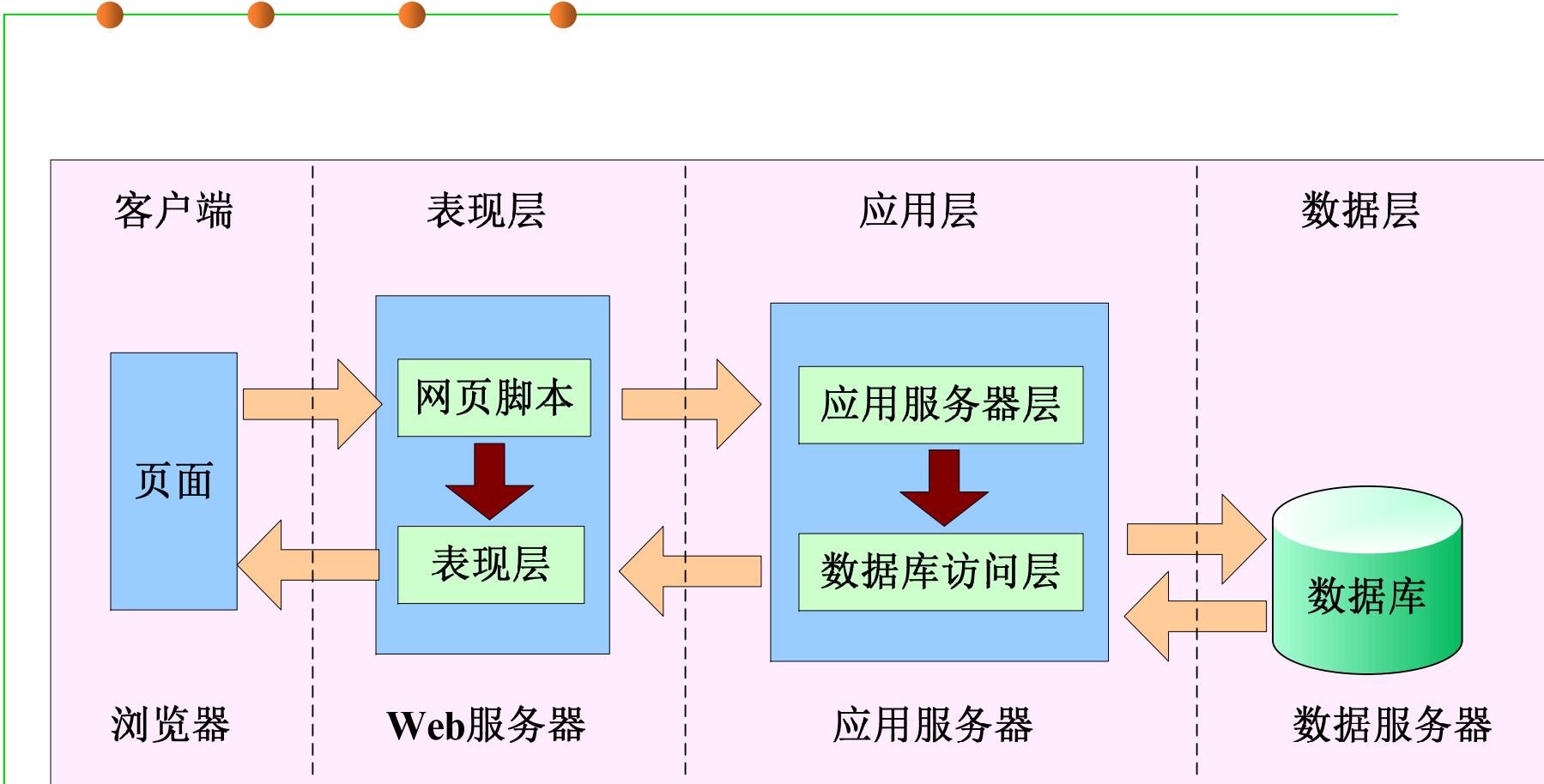


B/S结构

- 浏览器/服务器(Browser-Server,B/S)是三层C/S风格的一种实现方式。
 - 表现层：浏览器
 - 逻辑层：
 - Web服务器
 - 应用服务器
 - 数据层：数据库服务器



B/S结构



HTML

- 超文本标记语言HyperText Markup Language (HTML)
- Web页面文档(Document) = Hierarchical collection of *elements*
 - inline (headings, tables, lists...)
 - embedded (images, JavaScript code...)
 - **forms**: 用于向服务端提交数据 (text, radio/check buttons, dropdown menus...)
- 每个element具有特定的属性(*attributes*)和内容(*content*)
- 可采用大量的可视化HTML编辑软件编写(如: Dreamweaver等)

层叠样式表 Cascading Style Sheets (CSS)

- CSS: 将HTML文档中各elements的可视化显示样式与待显示的内容分开，在单独的文档(stylesheets)中加以定义，从而将页面设计师和开发者的工作分开。
- `<link rel="stylesheet" href="http://..."/>`, 在`<head>`元素内定义，用于指明该HTML页面使用哪个stylesheet;
- 在每个HTML要素内，使用`id`和`class`属性来指向CSS中的相关定义：

- `id`: 页面范围内的唯一标识;
 - `class`: 一个class可用于页面内的多个要素;

```
<div id="right" class="content">  
    <p>  
        I'm Rainy. I teach Software Engineering and do  
        research in the ICES Lab of CS/HIT.  
    </p>  
</div>
```

动态产生HTML内容

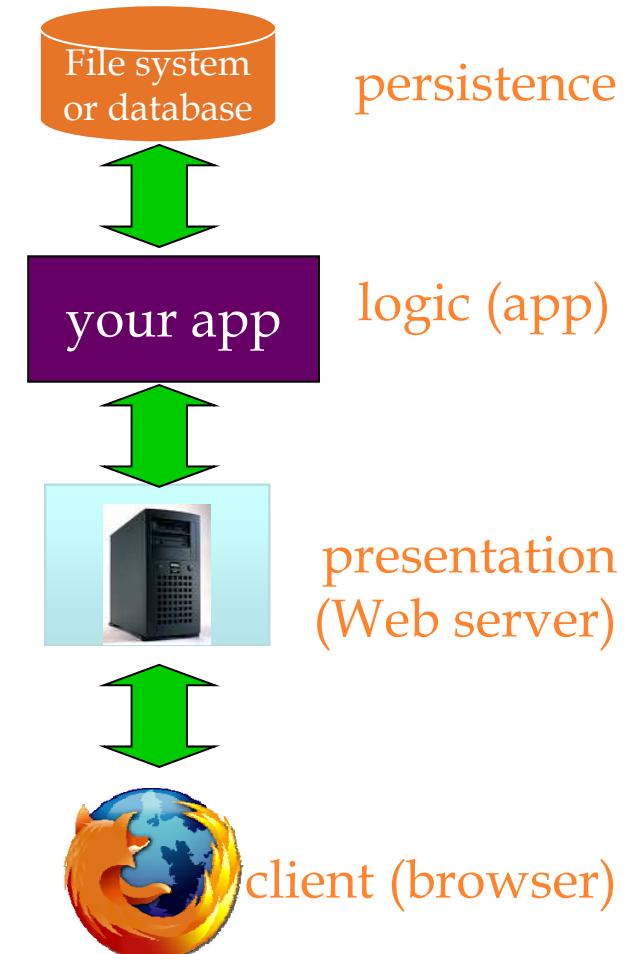
- 最初的大部分HTML页面都是静态的：直接从服务端获取，直接在HTML中展示；
- 目前的HTML：其中大部分内容是动态产生的，根据用户请求，服务端的程序执行之后产生内容，填充到HTML中，交付浏览器展示；
- 实现方式：在HTML中嵌入动态代码，如JSP、ASP、PHP、Ruby、Python等，在服务器端编译/解释后传输到浏览器端执行。
- 此外，为了扩充浏览器中HTML的能力，可使用JavaScript、...

网站==应用程序

- **How do you:**

- “map” URI to correct program & function?
- pass arguments?
- invoke program on server?
- handle persistent storage?
- handle cookies?
- handle errors?
- package output back to user?

- **Frameworks support these common tasks**

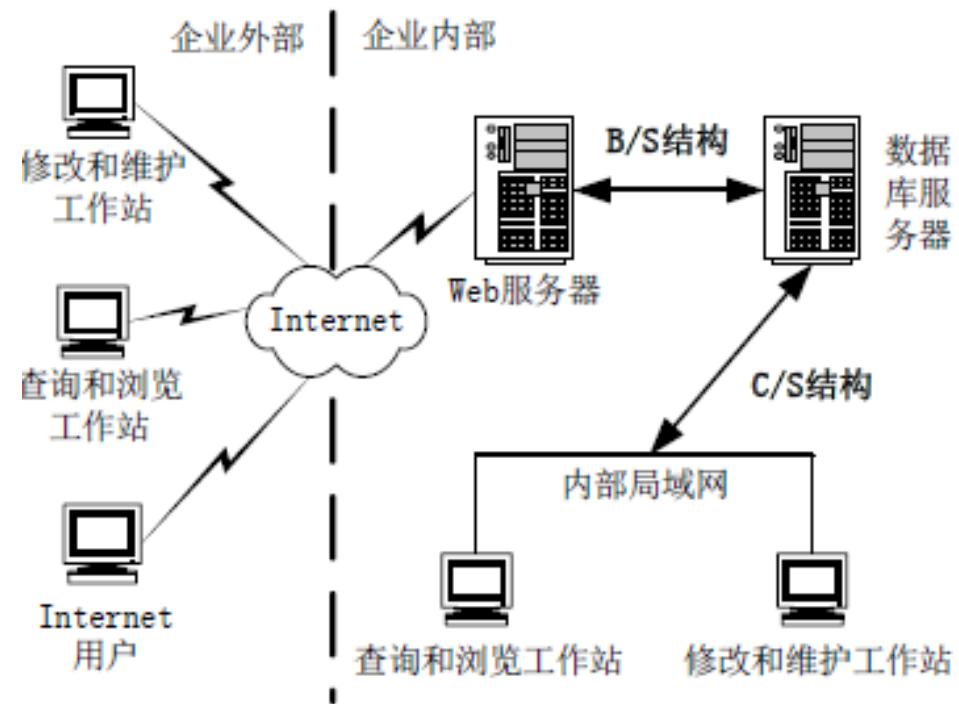


B/S结构

- 基于**B/S**体系结构的软件，系统安装、修改和维护全在服务器端解决，
系统维护成本低：
 - 客户端无任何业务逻辑，用户在使用系统时，仅仅需要一个浏览器就可运行全部的模块，真正达到了“零客户端”的功能，很容易在运行时自动升级。
 - **良好的灵活性和可扩展性：**对于环境和应用条件经常变动的情况，只要对业务逻辑层实施相应的改变，就能够达到目的。
- **B/S**成为真正意义上的“瘦客户端”，从而具备了很高的稳定性、延展性和执行效率。
- **B/S**将服务集中在一起管理，统一服务于客户端，从而具备了良好的容错能力和负载平衡能力。

C/S+B/S内外有别模式

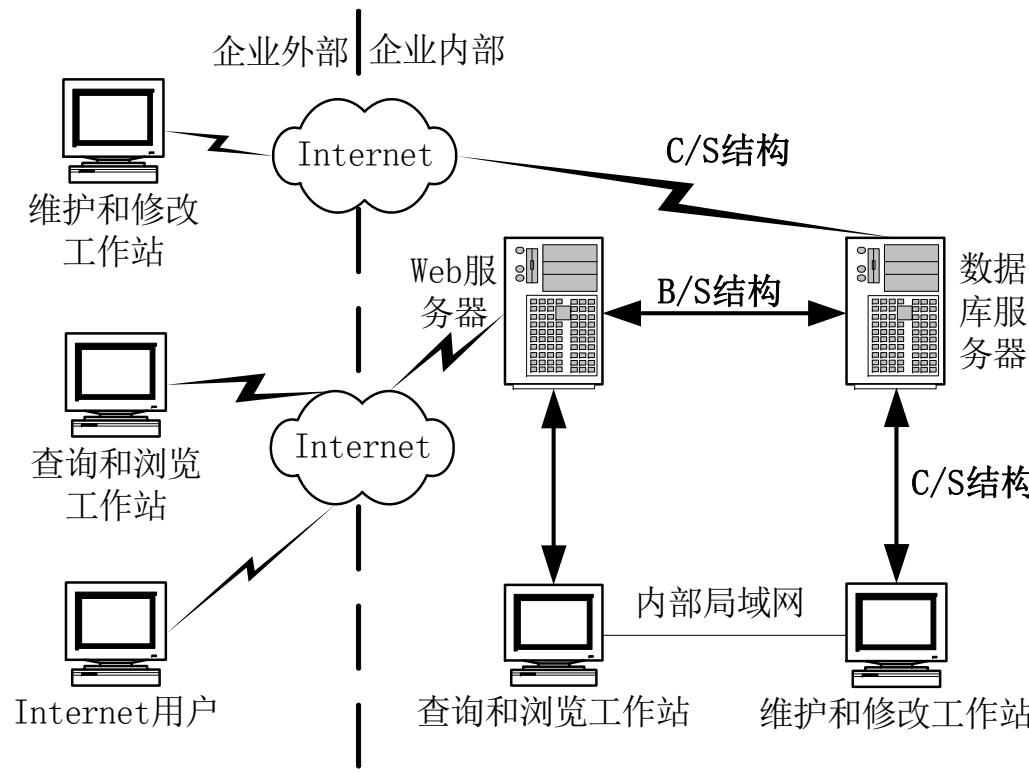
- 为了克服C/S与B/S各自的缺点，发挥各自的优点，在实际应用中，通常将二者结合起来；
- 遵循“内外有别”的原则：
 - 企业内部用户通过局域网直接访问数据库服务器
 - C/S结构；
 - 交互性增强；
 - 数据查询与修改的响应速度高；
 - 企业外部用户通过Internet访问Web服务器/应用服务器
 - B/S结构；
 - 用户不直接访问数据，数据安全



C/S+B/S混合模式

- 遵循“查改有别”的原则：

- 不管用户处于企业内外什么位置(局域网或Internet)，凡是需要对数据进行更新操作的(Add, Delete, Update)，都需要使用C/S结构；
- 如果只是执行一般的查询与浏览操作(Read/Query)，则使用B/S结构。



M/C结构

- 移动端/云端结构(**Mobile/Cloud**):
 - 客户端不是传统的客户机，而是各类移动终端设备，如智能手机、平板、智能家电、可穿戴设备等。
 - 服务端也不是传统的服务器，而是扩展到云环境下，支持高可伸缩性、按需付费等特性。
- 可以看作是C/S结构的扩展。
- 优势：移动，可以做到**anytime & anywhere**使用软件的功能。
- 客户端程序的体现形式：各类App

课堂讨论

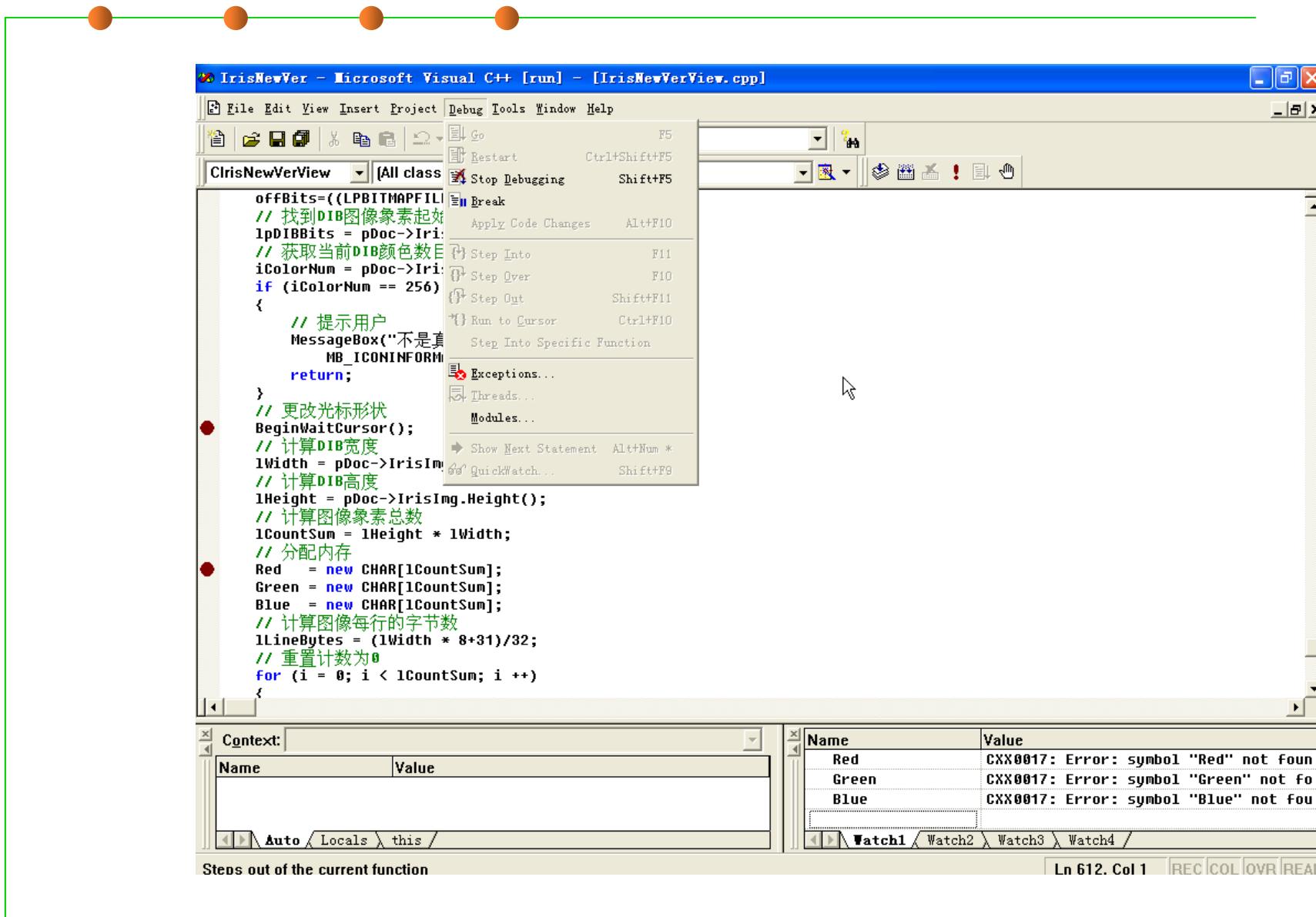
- 针对你所擅长的一个编程语言，阐述它对C/S、B/S、事件的实现机制
(1组)



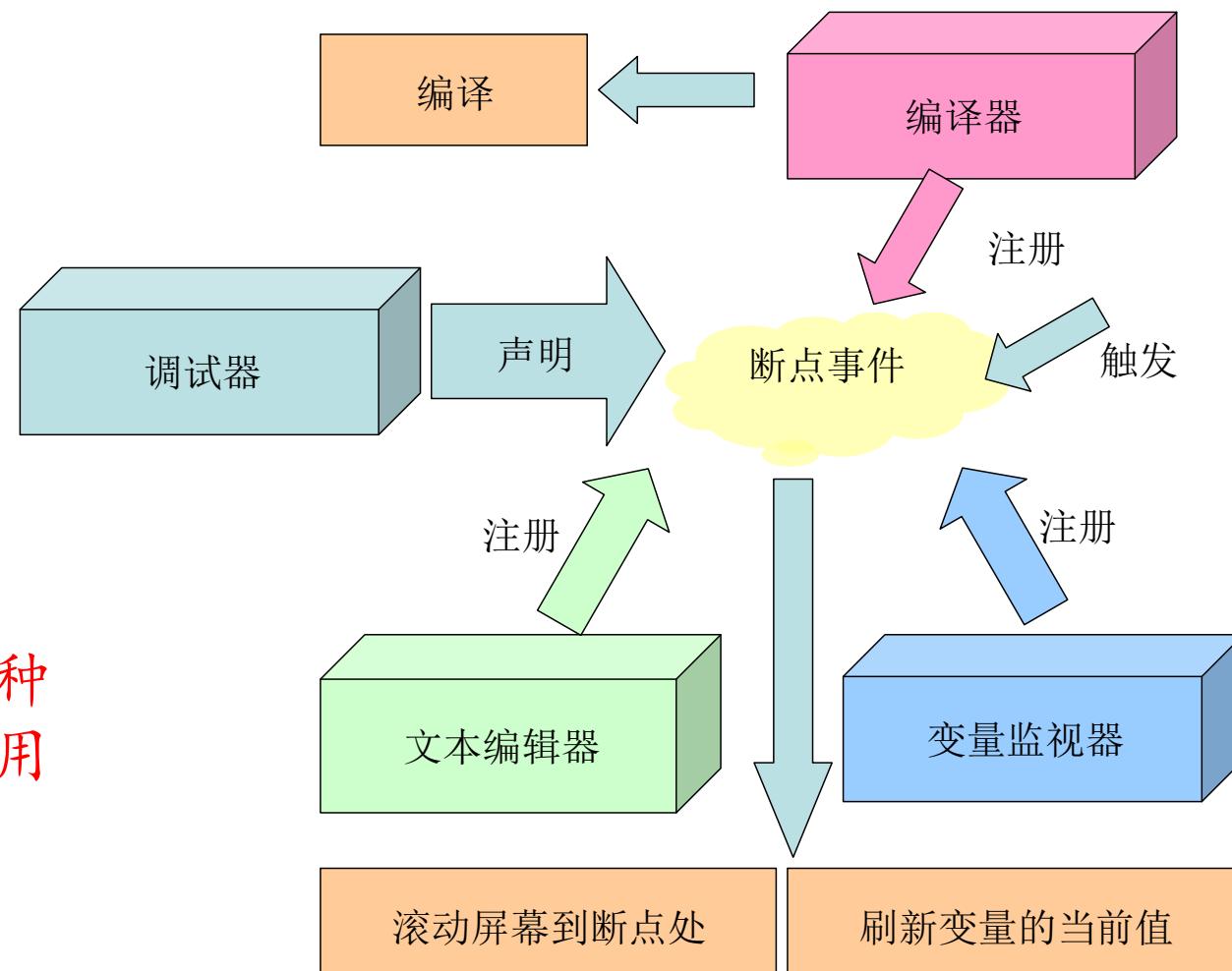
7. 事件风格

与“调用-返回”方式相对应，从同步调用变为了异步调用

例子：调试器

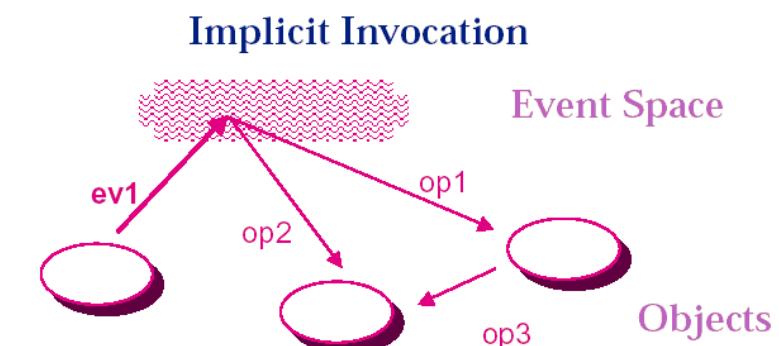
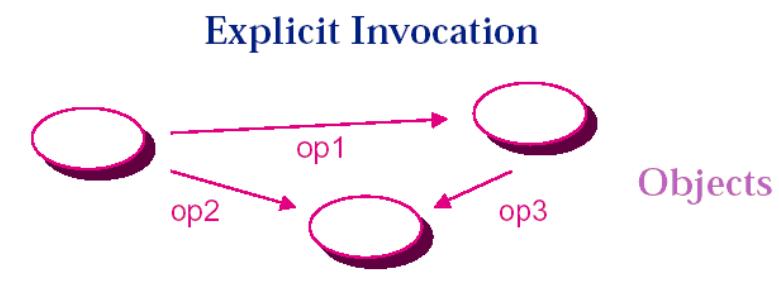


调试器的工作流程

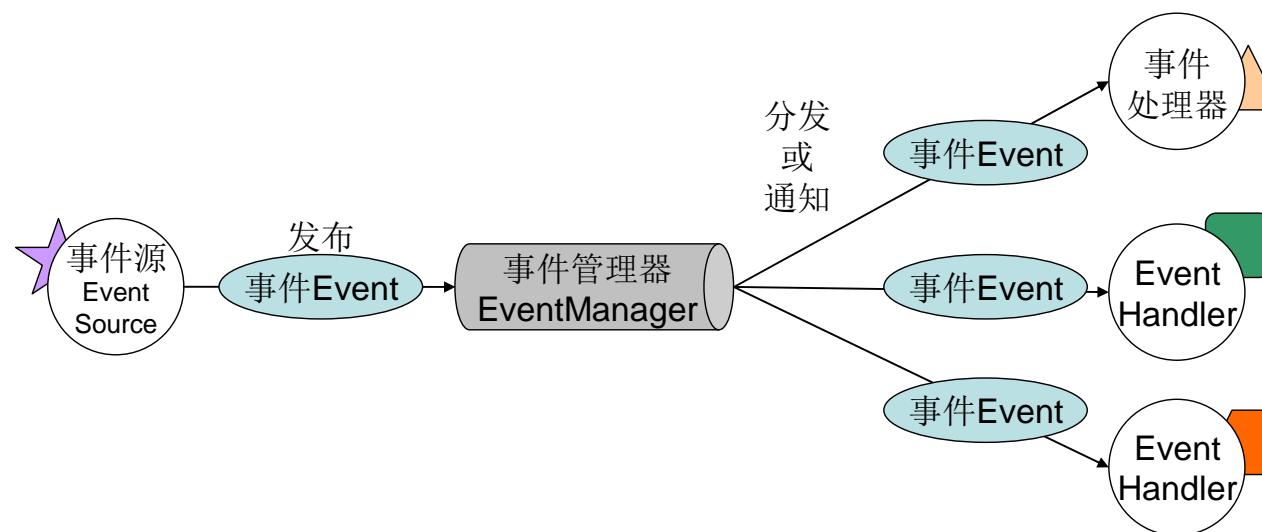


显式调用 vs. 隐式调用

- 显式调用：
 - 各个构件之间的互动是由显性调用函数或程序完成的。
 - 调用过程与次序是固定的、预先设定的。
- 隐式调用：
 - 调用过程与次序不是固定的、预先未知；
 - 各构件之间通过事件的方式进行交互；



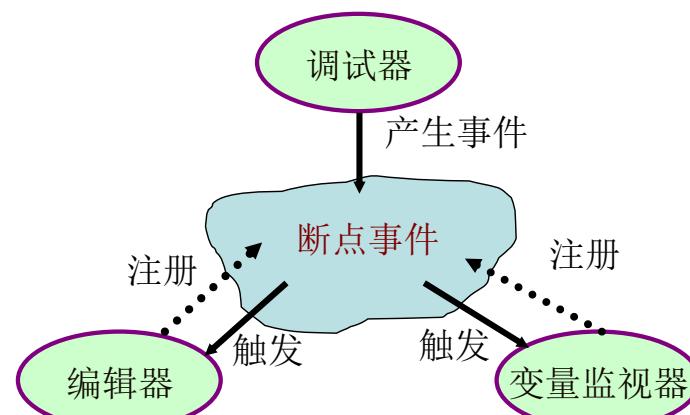
事件系统的基本构成



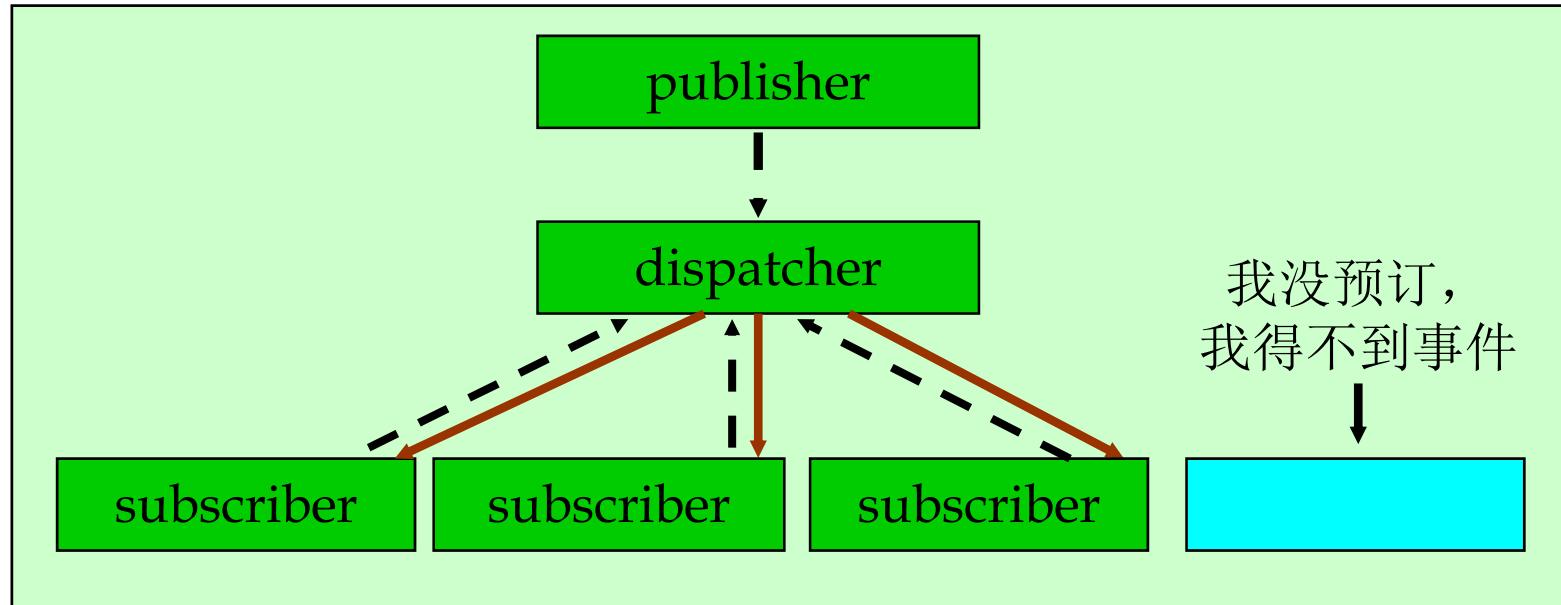
功能	描述
分离的交互	事件发布者并不会意识到事件订阅者的存在。
多对多通信	采用发布/订阅消息传递，一个特定事件可以影响多个订阅者。
基于事件的触发器	控制流由接收者确定（基于发布的事件）。
异步	通过事件消息传递支持异步操作。

回到调试器的例子

- EventSource: debugger (调试器)
- EventHandler: editor and variable monitor (编辑器与变量监视器)
- EventManager: IDE (集成开发环境)
- 编辑器与变量监视器向调试器注册，接收“断点事件”；
- 一旦遇到断点，调试器发布事件，从而触发“编辑器”与“变量监测器”；
- 编辑器将源代码滚动到断点处，变量监测器则更新当前变量值并显示出来。

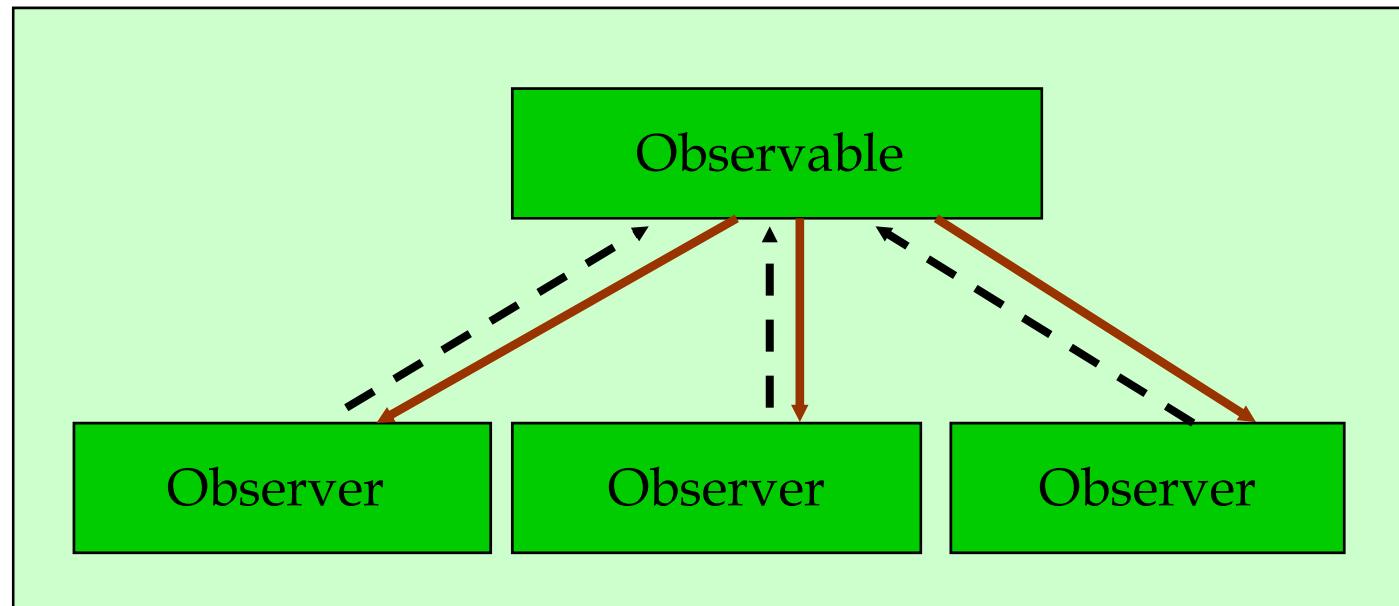


事件风格的实现策略之一：选择广播式



有目的广播，
只发送给那些已经注册过的订阅者

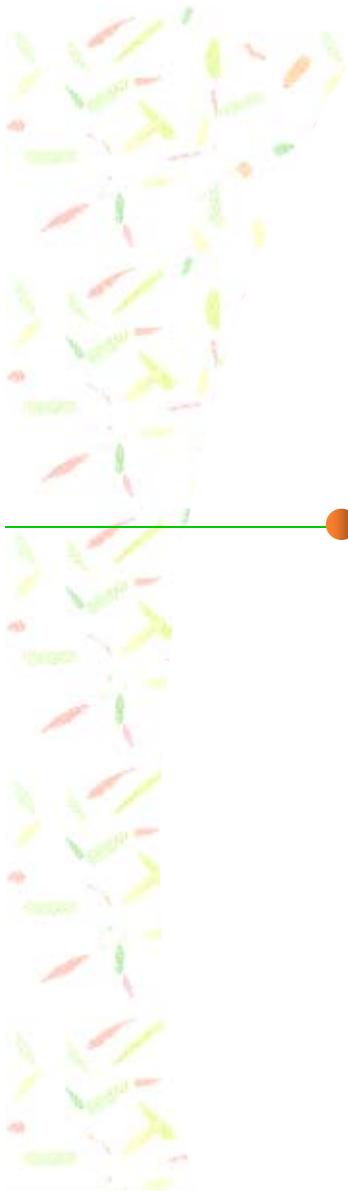
事件风格的实现策略之二：观察者模式



Legend: ► Register event → Send event



8. 模型-视图-控制器 (MVC)



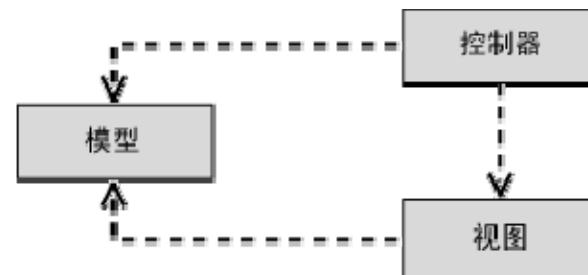
模型-视图-控制器 (MVC)

- ---- 用户界面需要频繁的修改，它是“不稳定”的。
- ---- 业务逻辑/数据 与 用户界面 之间应避免/尽量少的直接通信。

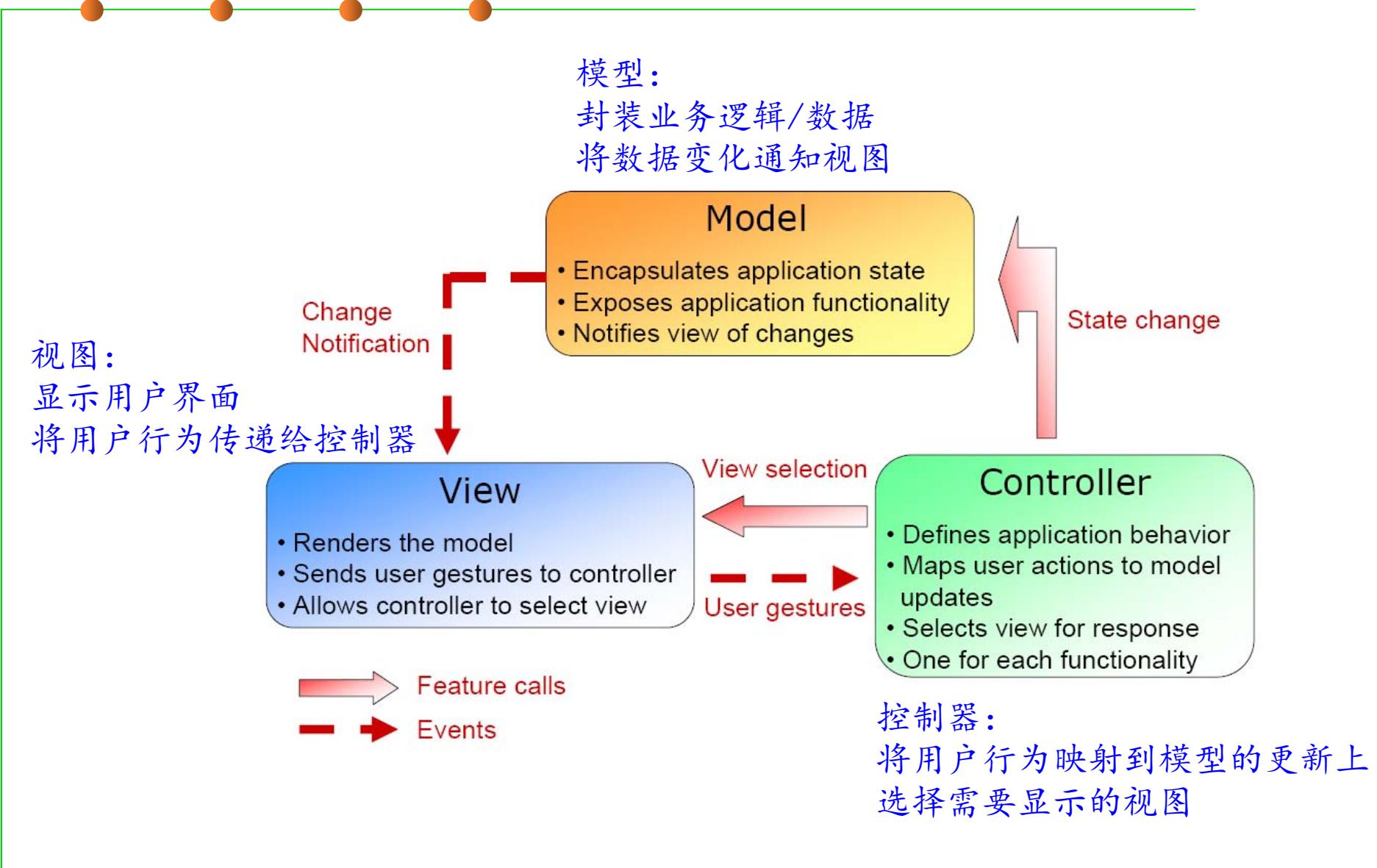
- 目标：将承担不同职责的软件实体之间清晰的分离开来，降低耦合；
 - 让SaaS程序的用户界面与业务逻辑功能实现模块化，以便使程序开发人员可以分别开发、单独修改各个部分而不影响其他部分。

解决方案：Model-View-Controller (MVC)

- MVC是一种软件体系结构，它将应用程序的数据模型/业务逻辑、用户界面分别放在独立的构件中，从而对用户界面的修改不会对数据模型/业务逻辑造成很大影响。
 - 在传统的B/S体系结构的基础上加入了一个新的元素：**控制器**，由控制器来决定视图与模型之间的依赖关系。
 - **模型(Model, M)**: 用于管理应用系统的行为和数据，并响应为获取其状态信息(通常来自视图)而发出的请求，还会响应更改状态的指令(通常来自控制器); — 对应于传统B/S中的业务逻辑和数据
 - **视图(View, V)**: 用于管理数据的显示; — 对应于传统B/S中的用户界面
 - **控制器(Controller, C)**: 用于解释用户的鼠标和键盘输入，以通知模型和视图进行相应的更改。 — 在传统B/S结构中新增的元素



MVC运行机制

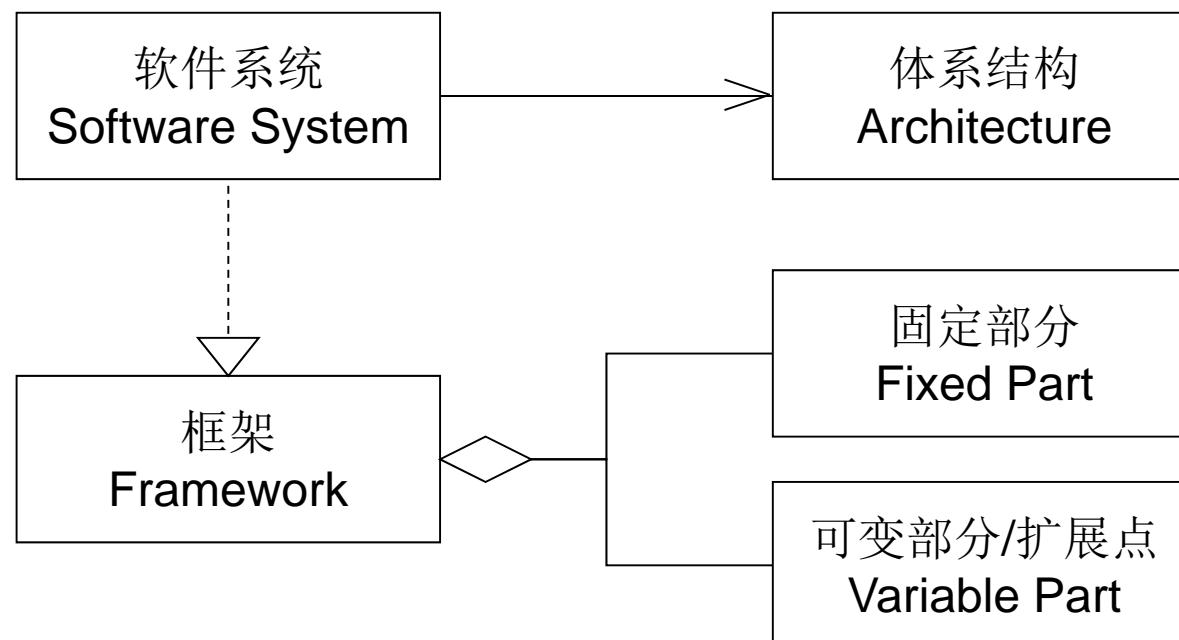


MVC各层次的实现技术

- 针对不同层次，采用不同的实现技术：
 - 用户界面层：HTML/JavaScript/CSS、jQuery、ExtJs、JSP、AJAX、Flex、HTML5、Dojo、Bootstrap、Node.js...
 - 控制层：PHP、Python、Servlet、Ruby、...
 - 业务逻辑层：JavaBean、Pojo、...
 - 持久化层：JDBC、JDO、Hibernate、iBatis、...
- **Struts、Django、CI、Rails**等以不同的编程语言(**Java、Python、PHP、Ruby**)分别实现了这一架构，提供了一个半成品，帮助开发人员迅速地开发符合**MVC**架构的应用程序，它们都是“框架**Framework**

Framework vs Architecture (框架和体系结构)

- 框架(Framework): 可实例化的、部分完成的软件系统或子系统，它为一组系统或子系统定义了统一的体系结构(architecture)，并提供了构造系统的基本构造块(building blocks)，还为实现具体功能定义了扩展点(extending points)。
- 框架实现了体系结构级别的复用。



MVC Framework

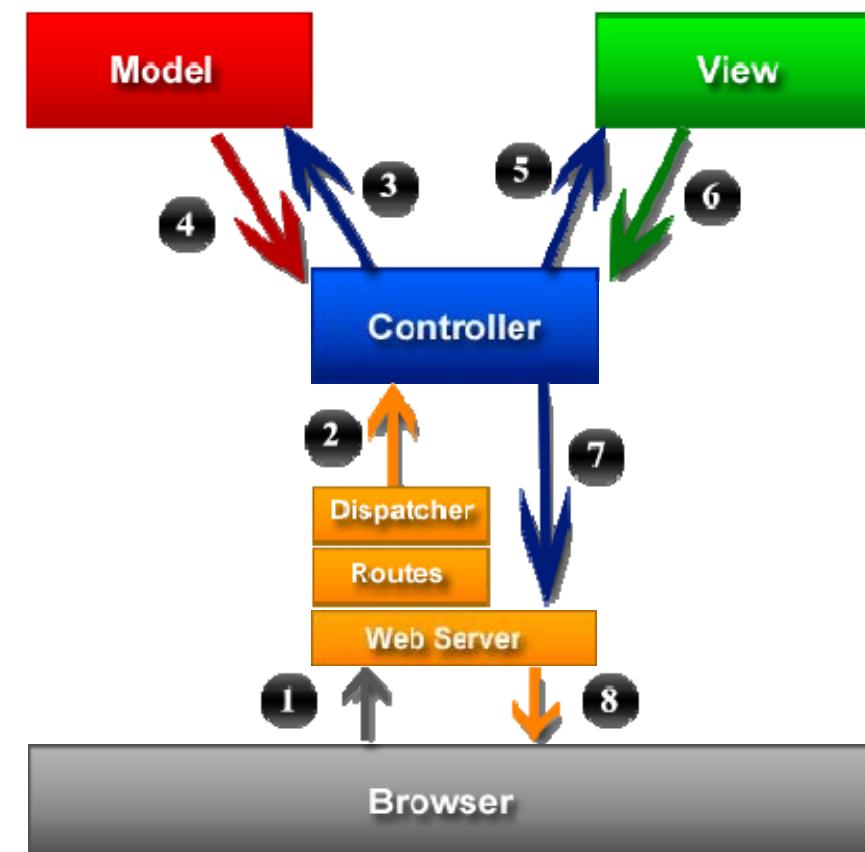
- Java: Struts、Spring
- Python: Django、Pylons、Tornado
- PHP: CodeIgniter、Zend、CakePHP、ThinkPHP、Yii
- Ruby: Rails、Sinatra
- [http://en.wikipedia.org/wiki/Comparison
of web application frameworks](http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks)
- 每个框架均覆盖了M、V、C、Persistence四部分，提供了各类基础库的支持。

Struts



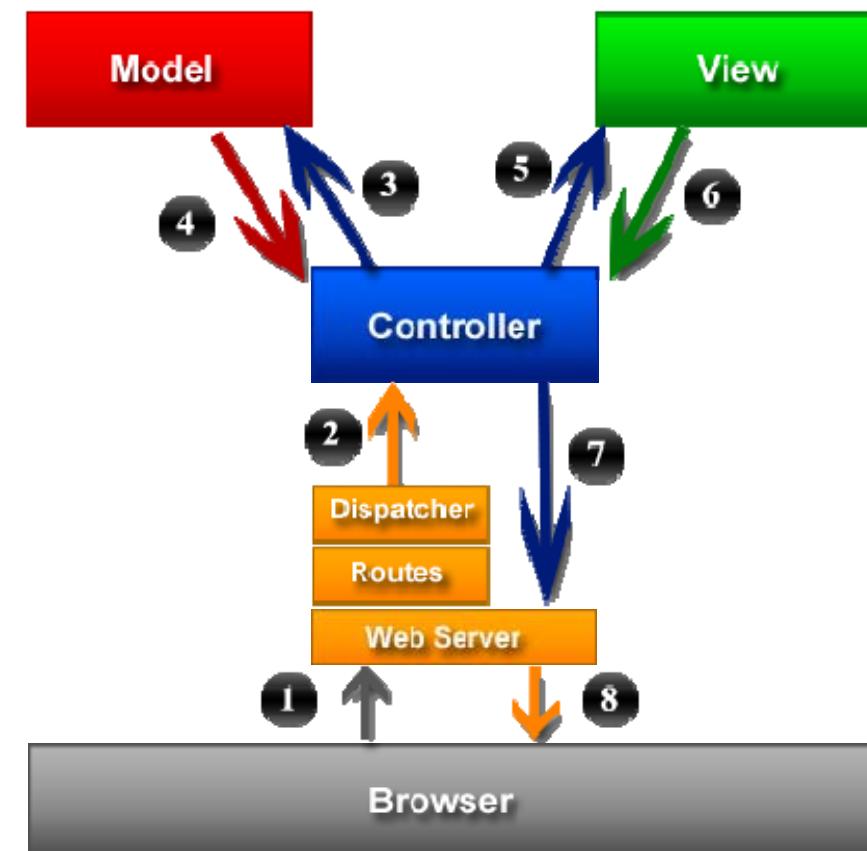
前端View

- HTML+CSS+JavaScript
- jQuery、ExtJS
- Bootstrap
- HTML5



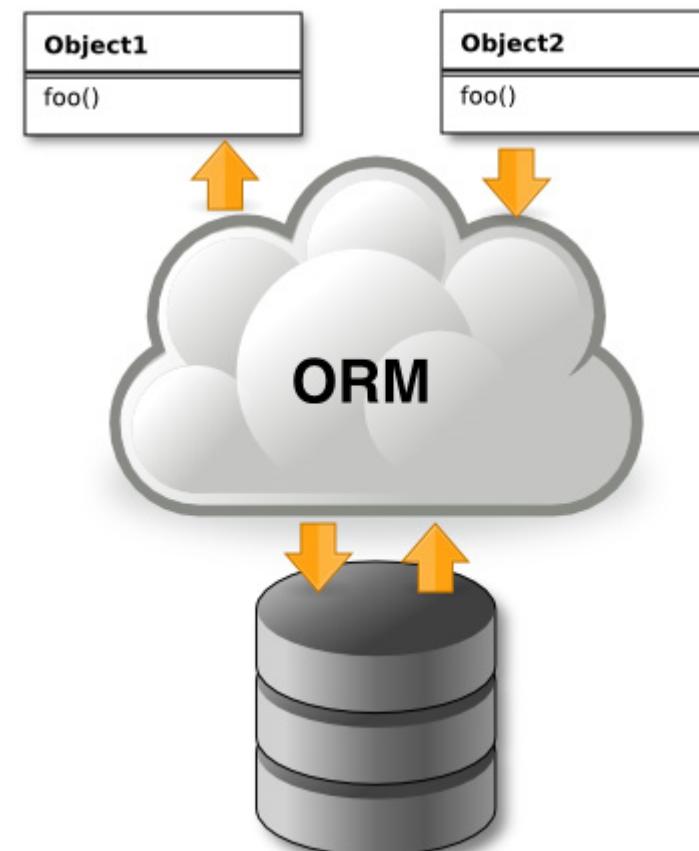
后端Controller/Model

- Java
- PHP
- Python
- Ruby
- C++
- C#
- ASP.Net



Persistence (数据持久化)

- Hibernate
- Apache OJB
- iBatis
- Jaxor
- ...



以Python和Django为例



- Django是一个开放源代码的Web应用框架，由Python写成。采用了MVC的软件设计模式，以比利时的吉普赛爵士吉他手Django Reinhardt命名。
- 设计理念
 - Django的主要目的是简便、快速地开发数据库驱动的网站——动态网站。
 - Django强调代码复用，多个组件可以方便地以“插件”形式服务于整个框架，Django有许多功能强大的第三方插件。
 - Django强调快速开发，DRY-Don't Repeat Yourself
 - 基于MVC（更确切的说是MTV）
- <https://www.djangoproject.com/> Django网站
- 《The Django Book 2.0》中文译本 CMS网站上
- 《Django Web开发指南》 CMS网站上

以Python和Django为例



■ 设计哲学

- 对象关系映射 ([ORM](#))：将模型与关系数据库连接起来，非常容易使用的数据库API，也可以使用原始的SQL语句。
- 基于正则表达式匹配的URL 分派。
- 强大而可扩展的模板语言：可以分隔设计、内容和Python代码。并且具有可继承性。
- 表单处理：可以方便的生成各种表单模型，实现表单的有效性检验。方便的从定义的模型实例生成相应的表单。
- Cache系统：可以挂在内存缓冲或其它的框架实现超级缓冲 —— 实现你所需要的粒度。
- 会话([session](#))，用户登录与权限检查，快速开发用户会话功能。
- 国际化：内置国际化系统，方便开发出多种语言的网站。
- 自动化的管理界面：不需要你花大量的工作来创建人员管理和更新内容。Django自带一个ADMIN site,类似于[内容管理系统](#)。

Django的MTV

■ M = Model

- 定义数据结构和行为，负责数据库管理
- 使用ORM技术
- models.py文件

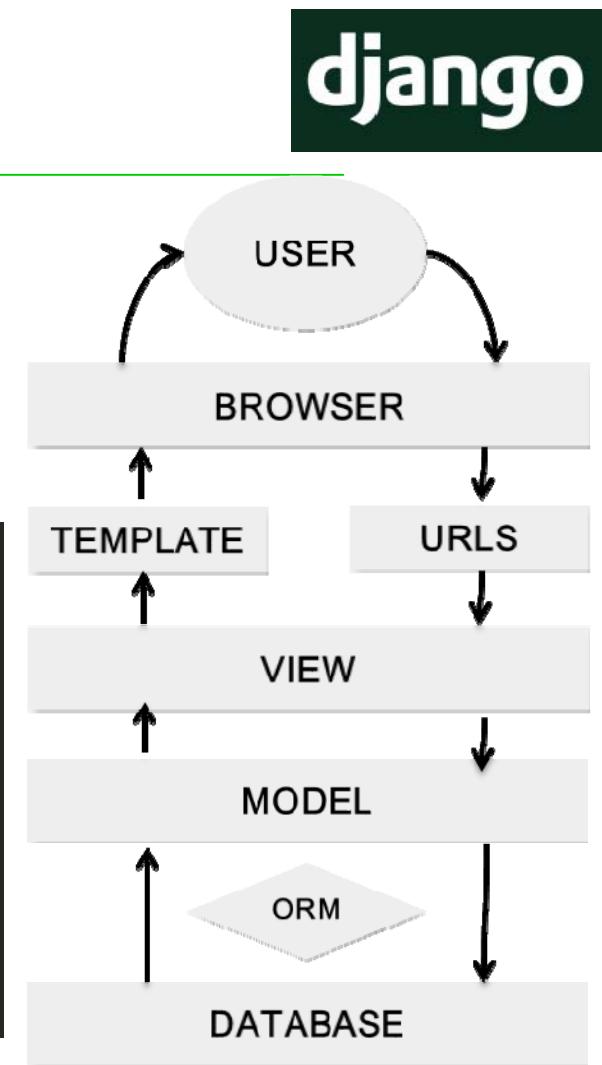
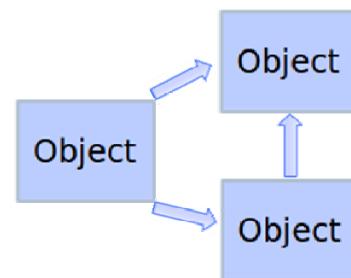
```
class Issue(models.Model):
    dms_id = models.CharField(max_length=200)
    title = models.CharField(max_length=500)
    submitter = models.CharField(max_length=100)
    rft_status = models.CharField(max_length=100, blank=True, null=True, choices=choices.MODEL_CHOICES_STATUS,
    rft_owner = models.CharField(max_length=200, blank=True, null=True)
    rft_result = models.CharField(max_length=100, blank=True, null=True, choices=choices.MODEL_CHOICES_RESULT,
    class Meta:
        app_label = "rftissue"
        abstract = True
        # ordering = ['dms_id']

    def __unicode__(self):
        return self.dms_id

    def get_basic_status(self):
        pass

    def get_specified_status(self):
        pass
```

Table		
column1	column2	column3
row1_column1	row1_column2	row1_column3
row2_column1	row2_column2	row2_column3
Table		
column1	column2	column3
row3_column1	row3_column2	row3_column3
row4_column1	row4_column2	row4_column3
row5_column1	row5_column2	row5_column3



Django的MTV



■ V = View

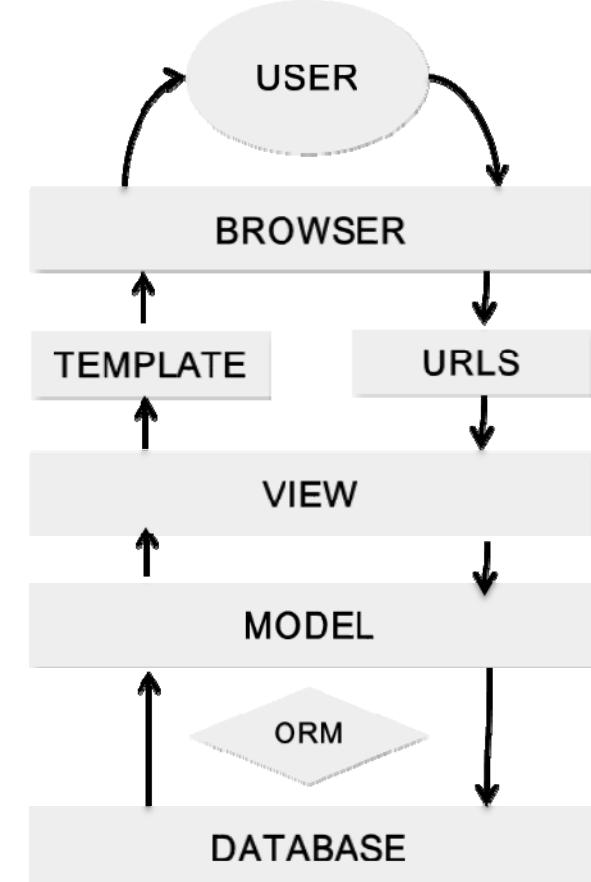
- 用于控制要显示什么数据
- 返回HTTP响应
- views.py文件

```

@ensure_csrf_cookie
@login_required(login_url='/accounts/signin/')
def home(request):
    ...
    Display home page.
    ...
    # User must be logged, otherwise the user will redirected to signin page
    user = "Guest"
    if request.user.is_authenticated():
        user = request.user.username
    return render(request, 'rft/home.html', {'user': user})

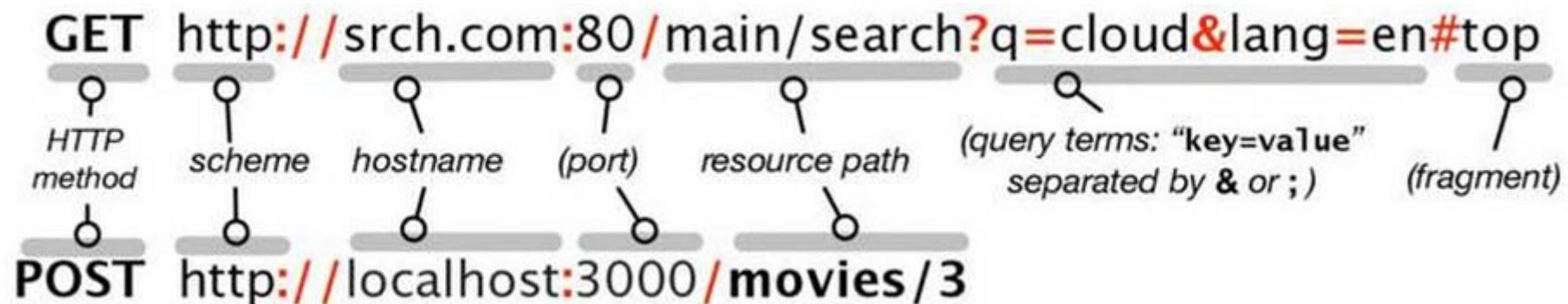
@login_required(login_url='/accounts/signin/')
def display_acm_rft(request, export='none'):#

@login_required(login_url='/accounts/signin/')
def display_sw_rft(request, export='none'):#
```



前端请求的URI (Uniform Resource Identifier)

- 用户通过浏览器发出的请求，通过HTTP协议传递到服务器端。
- 具体形式即为URI



- HTTP是无状态协议，通过浏览器端的cookies保持同一用户的多次请求，通过服务器端的session保持与用户的连接。

Django的MTV



- urls.py is a mapping between URL patterns to Python functions (Views)

```
urlpatterns = patterns('',
    url(r'^$', views.home),
    url(r'^acm/$', views.display_acm_rft),
    url(r'^acm/csv$', views.display_acm_rft, {'export': 'csv'}),
    url(r'^sw/$', views.display_sw_rft),
    url(r'^sw/csv$', views.display_sw_rft, {'export': 'csv'}),
    url(r'^acm/(?P<pk>\d+)/$', views.edit_issue, {'issue_type': 'acm'}),
    url(r'^sw/(?P<pk>\d+)/$', views.edit_issue, {'issue_type': 'sw'}),
    url(r'^export/csv/$', views.export_to_csv),
    url(r'^about/$', views.about),
    url(r'^doc/$', views.doc),
    url(r'^doc/guide/$', views.user_guide),
    url(r'^send_email_notification/$', views.send_email_notification),
    url(r'^api/acm/$', views.ACMList.as_view()),
    url(r'^api/acm/(?P<pk>[0-9]+)/$', views.ACMDetail.as_view()),
    url(r'^api/sw/$', views.SWList.as_view()),
    url(r'^api/sw/(?P<pk>[0-9]+)/$', views.SWDetail.as_view()),
)
```

Django的MTV



■ T = Template

- 负责怎样显示数据
- 利用格式化的html文件，使数据按照要求显示
- 强大而可扩展的模板语言

```

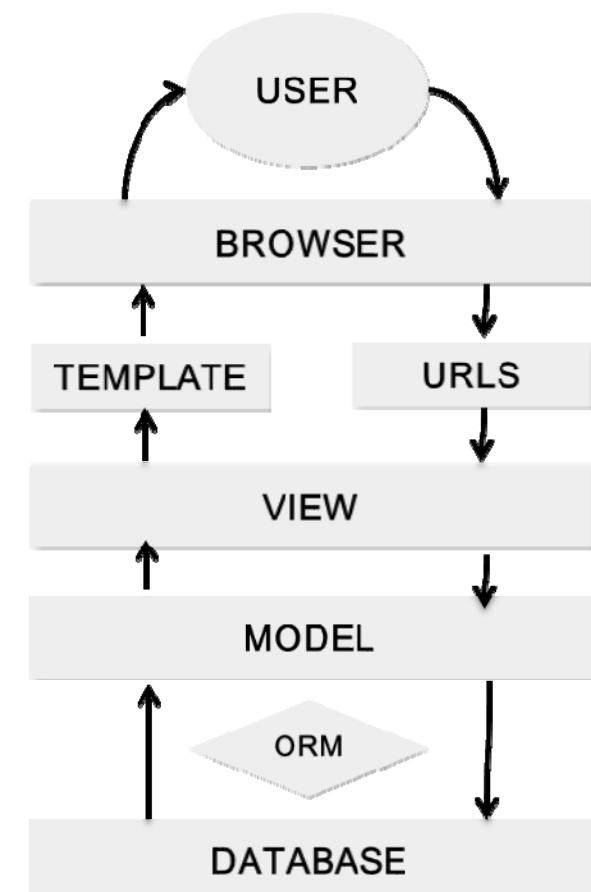
{% block title %}{% trans "Signin" %}{% endblock %}

{% block content %}
    <form action="" method="post">
        {% csrf_token %}
        <fieldset>
            <legend><strong>{% trans "Welcome to CV RFT Issues Management Site, please log in:" %}</strong></legend>
            {{ form.non_field_errors }}
            {% for field in form %}
                {{ field.errors }}
                {% comment %} Displaying checkboxes differently {% endcomment %}
                {% if field.name == 'remember_me' %}
                    <p class="checkbox">
                        <label for="id_{{ field.name }}">{{ field }} {{ field.label }}</label>
                    </p>
                {% else %}
                    <p>
                        {{ field.label_tag }}
                        {{ field }}
                    </p>
                {% endif %}
            {% endfor %}
        </fieldset>

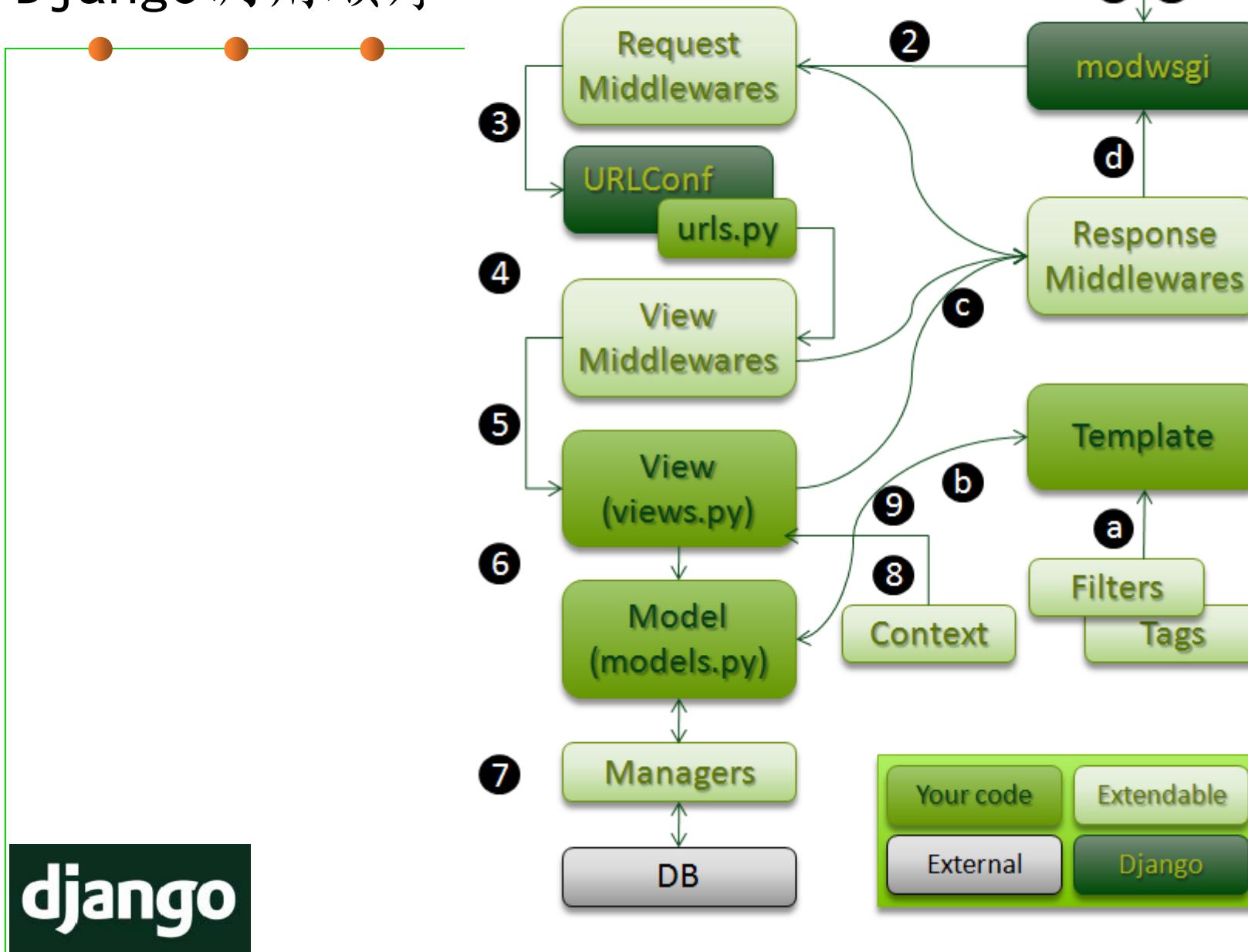
        <input type="submit" value="{{ trans "Signin" }}/>

        <p class="forgot-password"><a href="{% url 'userena_password_reset' %}" title="{{ trans 'Forgot your password?' }}>{{ trans "Forgot your password?" }}</a>
        </p>
        {% if next %}<input type="hidden" name="next" value="{{ next }}"/>{% endif %}
    </form>
{% endblock %}

```



Django调用顺序



MTV vs MVC



传统MVC	Django的MVC（MTV）
M：封装业务逻辑/数据；将数据变化通知视图	M: 数据存取
V: 负责把数据格式化后呈现给用户	V: Django将MVC中的视图进一步分解为 Django视图V和 Django模板T两个部分，分别决定“展现哪些数据”和“如何展现”，使得Django的模板可以根据需要随时替换，而不仅仅限制于内置的模板
C: 将用户行为映射到模型的更新上；选择需要显示的视图	C: 由Django框架的URLconf来实现

Settings.py



```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
)

MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
)

ROOT_URLCONF = 'demoproject.urls'

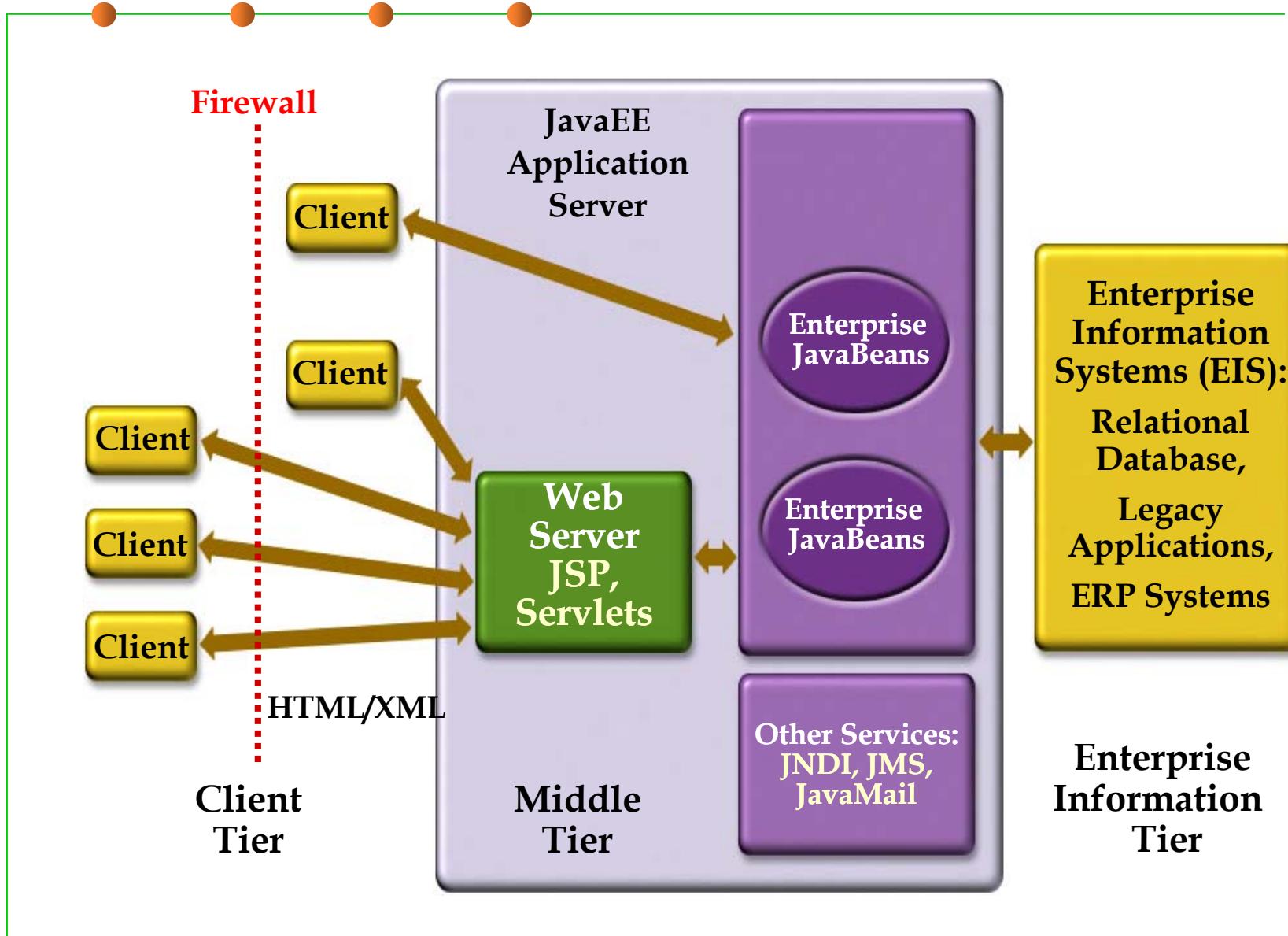
WSGI_APPLICATION = 'demoproject.wsgi.application'

# Database
# https://docs.djangoproject.com/en/1.6/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

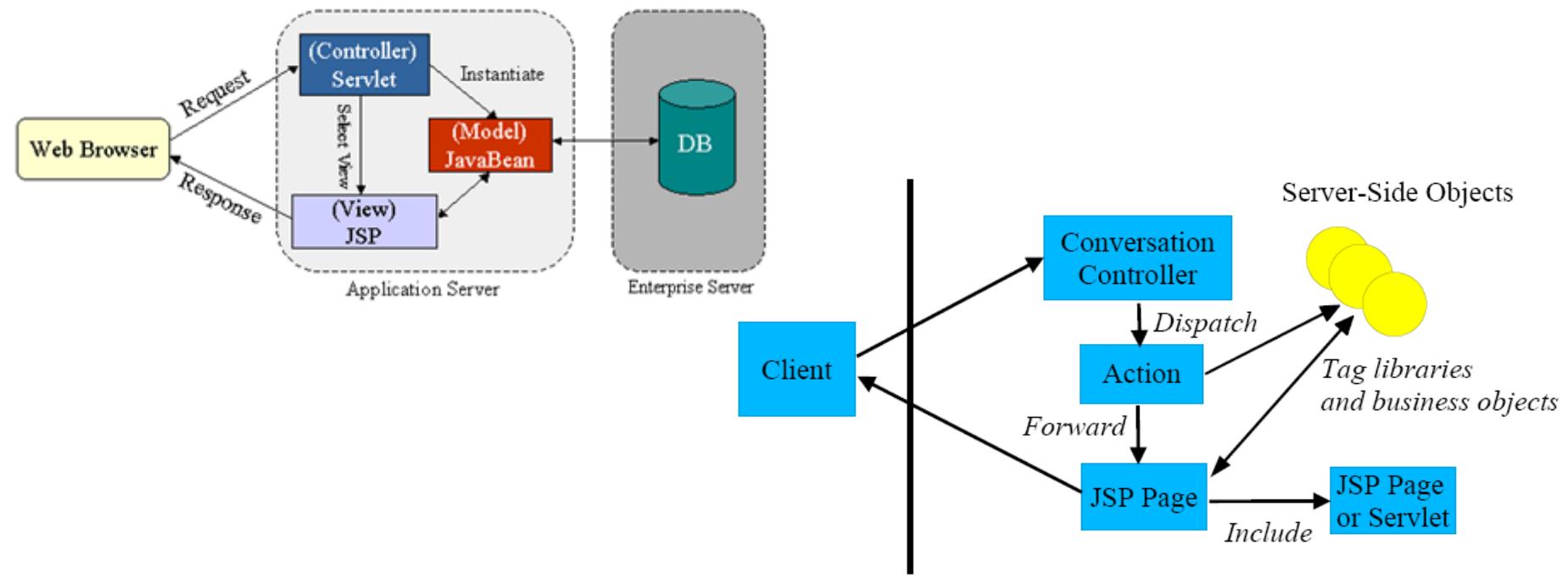
- **INSTALLED_APPS:** Enable the Django applications. App names **MUST** be unique.
- **MIDDLEWARE_CLASSES:** Enable the middleware components. The order in MIDDLEWARE_CLASSES matters.
- **ROOT_URLCONF:** Represent the full Python path to root URLconf.
- **WSGI_APPLICATION:** The full Python path of WSGI app object that built-in application will use.
- **DATABASES:** Settings for all databases to be used with Django.

**以JavaEE和Struts为例



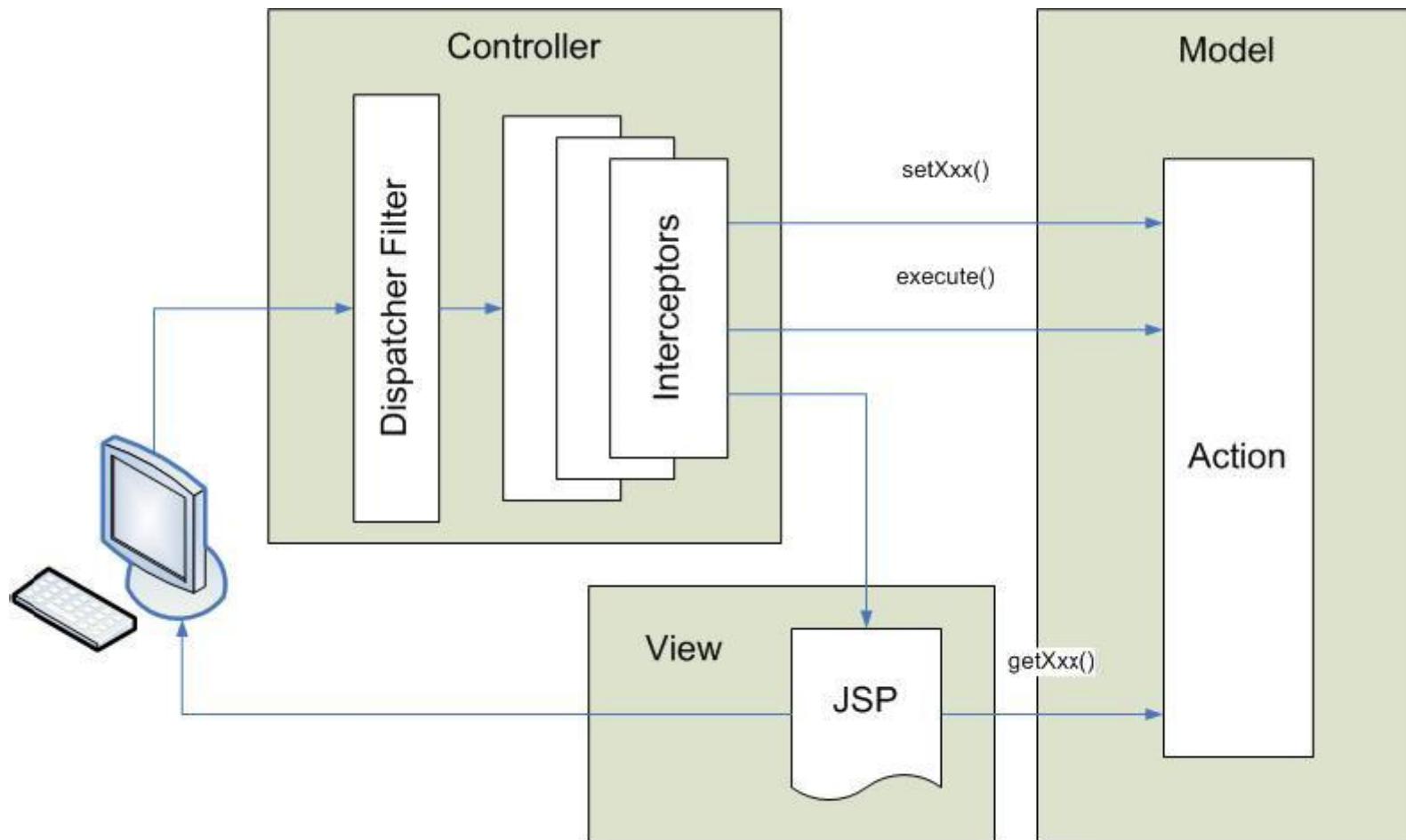
**JavaEE MVC Model

- **Servlet**作为控制器，负责处理用户请求，并将其转发给扮演**Model**角色的**JavaBean**，而**JSP**作为纯粹的**View**：
- **Servlet**还决定在处理完该请求之后，将向用户显示哪个**JSP**页面(**View**)以显示处理结果；
- **JSP**页面获取**Servlet**的处理结果并动态显示给用户。



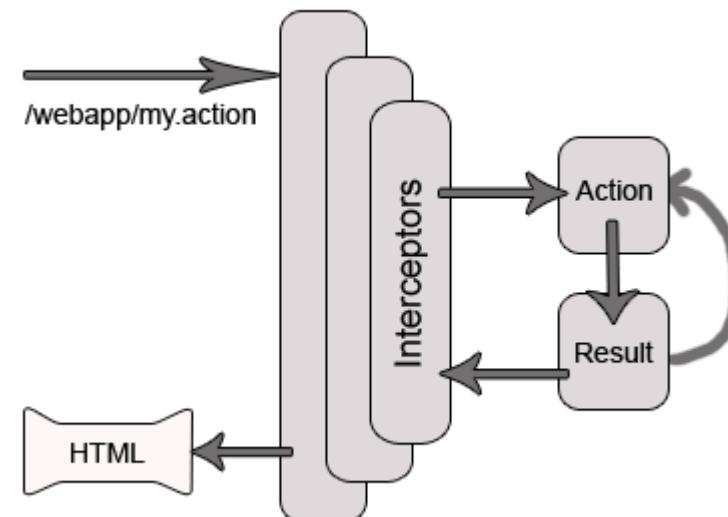
** Struts: a MVC framework of JavaEE

Struts²

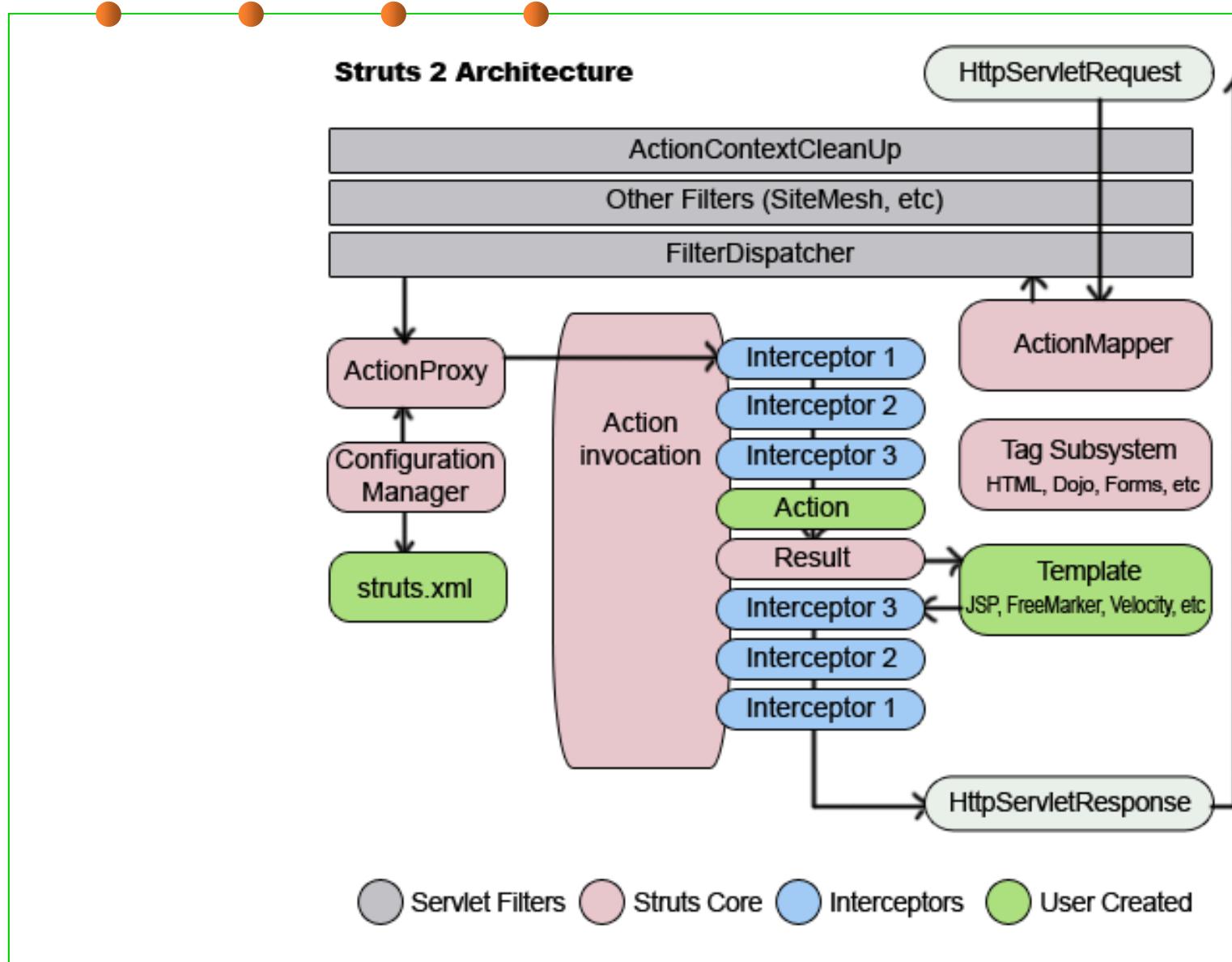


** Struts: a MVC framework of JavaEE

- User Sends Request
- Filter Dispatcher determines the appropriate action
- Interceptors are applied
- Execution of action
- Output Rendering
- Return of Request
- Display of result to user

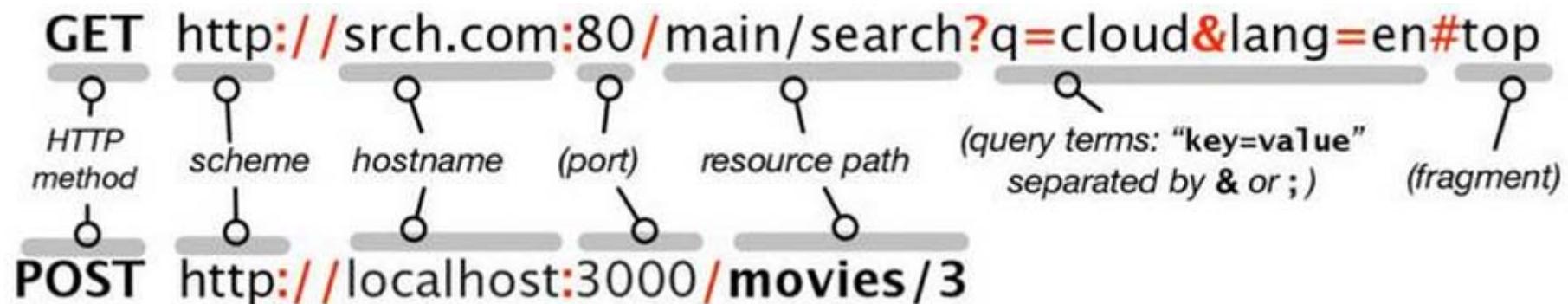


** Struts: a MVC framework of JavaEE



**前端请求的URI (Uniform Resource Identifier)

- 用户通过浏览器发出的请求，通过HTTP协议传递到服务器端。
- 具体形式即为URI



- HTTP是无状态协议，通过浏览器端的cookies保持同一用户的多次请求，通过服务器端的session保持与用户的连接。

** FilterDispatcher：前端控制器/调度器

- 前端请求的URI中包含了一个**resource path**，代表着用户的请求。
- 通过**HTTPRequest**发送至MVC的**front controller**，它相当于服务端的入口、总调度。——在Struts中，实现为**FilterDispatcher**。
- **FilterDispatcher**接收到请求之后，根据配置文件将请求转发到具体执行动作的**Model (Struts中称之为action)**。
- 配置文件**struts.xml**:

```
<struts>
    <package name="01" extends="struts-default">
        <action name="submit" class="SubmitAction" method="submit">
            <param name="param1">value1</param>
            <result>result.jsp</result>
        </action>
    </package>
</struts>
```

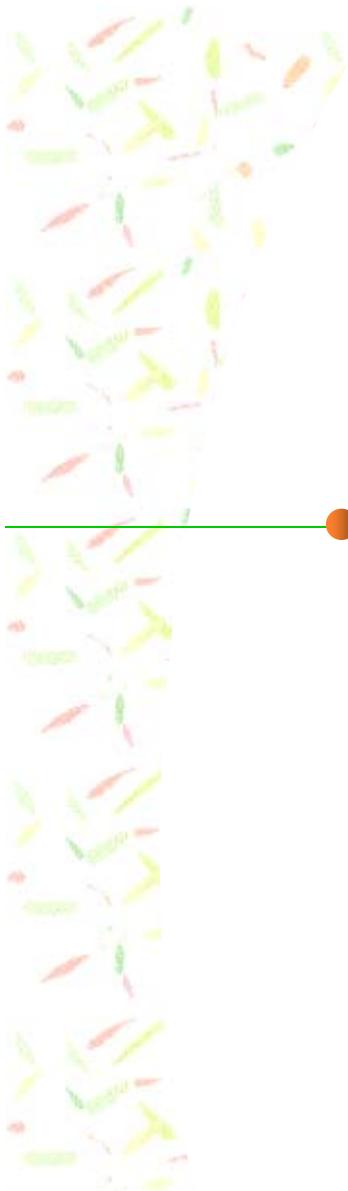
** Action(model) 和 JSP(View)

```
public class SubmitAction extends ActionSupport {  
    private String param1;  
    public String execute() throws Exception{  
        ...  
    }  
    public void submit(String param1){  
        ...  
    }  
}
```

```
<s:form action="submit" method="true">  
    <s:textfield label="Message" name="Param1" />  
    <s:submit value="submit" />  
</s:form>
```



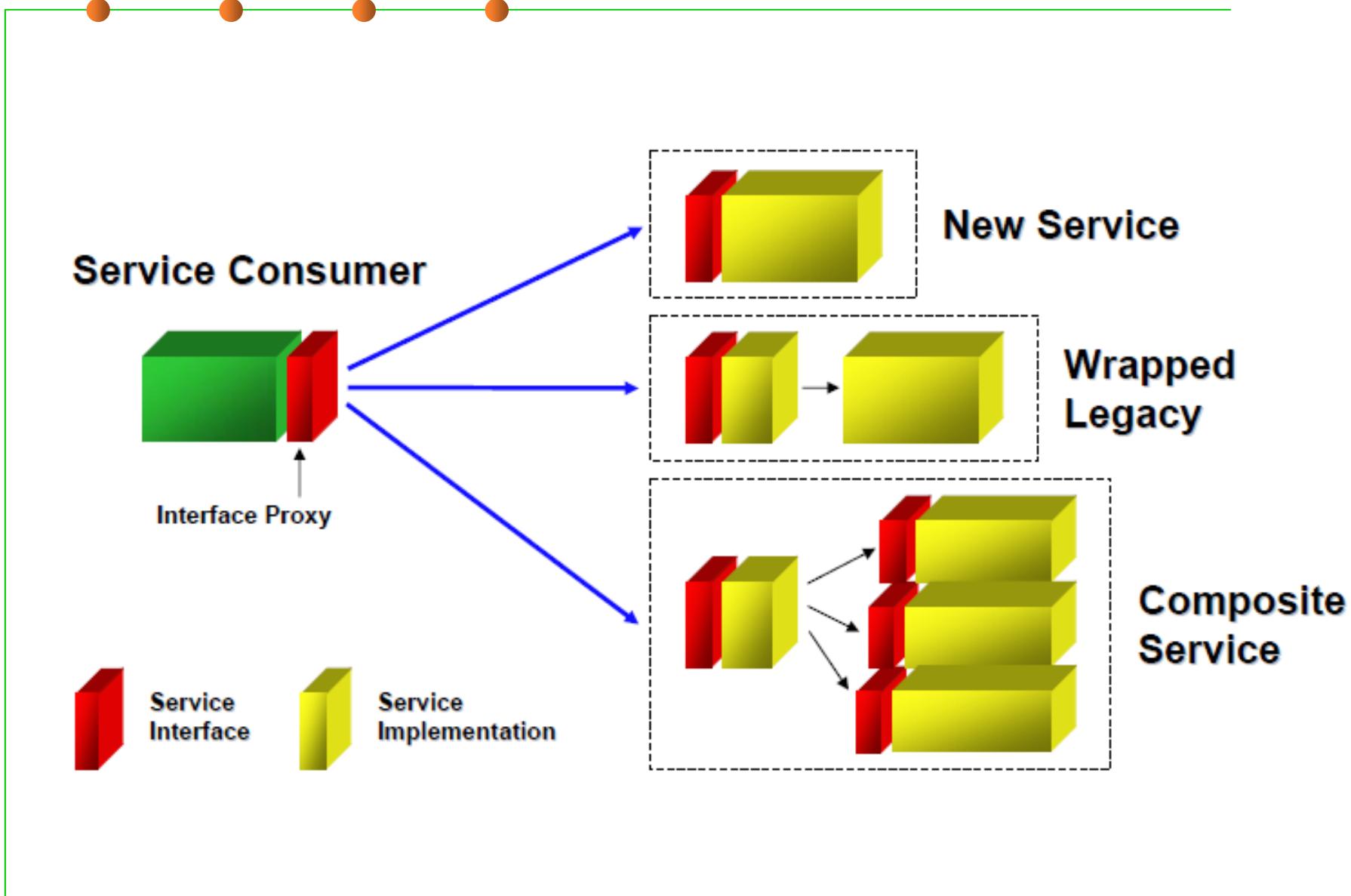
*9. 面向服务的体系结构



面向服务的体系结构

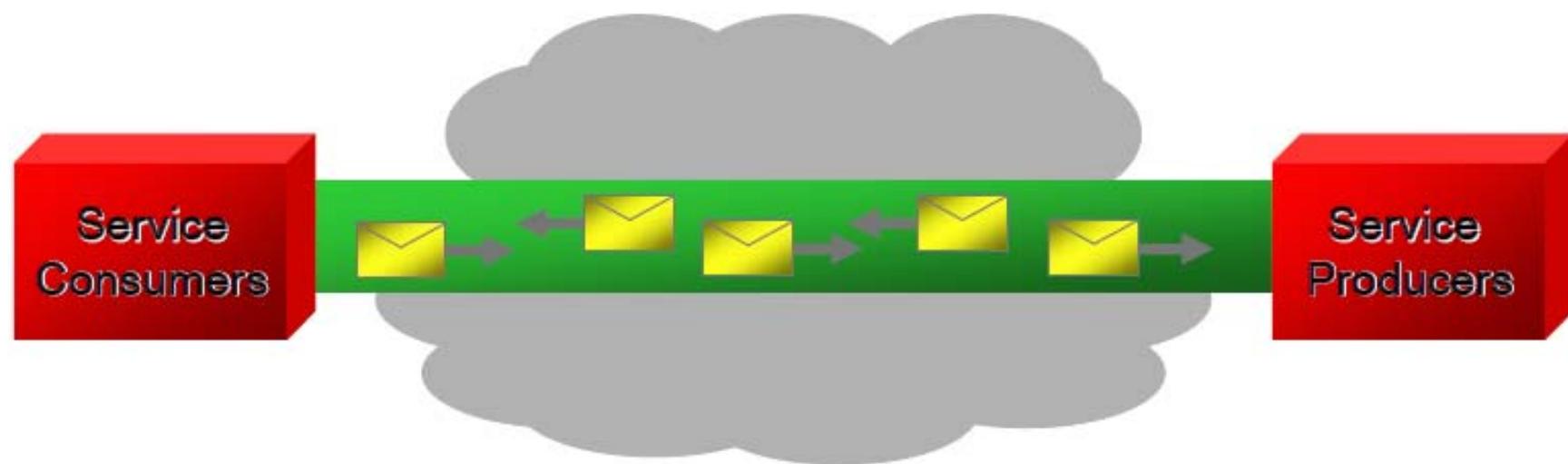
- SOA中可用的基本构件是“**服务**”：
 - 一个服务是一个服务提供者为一个服务消费者获得其想要的最终结果所提供的一个软件单元。
 - 从外特性上看，一个服务被定义为显式的、独立于服务具体实现技术细节的**接口**。
 - 从内特性上看，**服务封装了可复用的业务功能**，这些功能通常是**大粒度业务**，如业务过程、业务活动等。服务的实现可采用任何技术平台，如J2EE、.Net等。

服务(service)



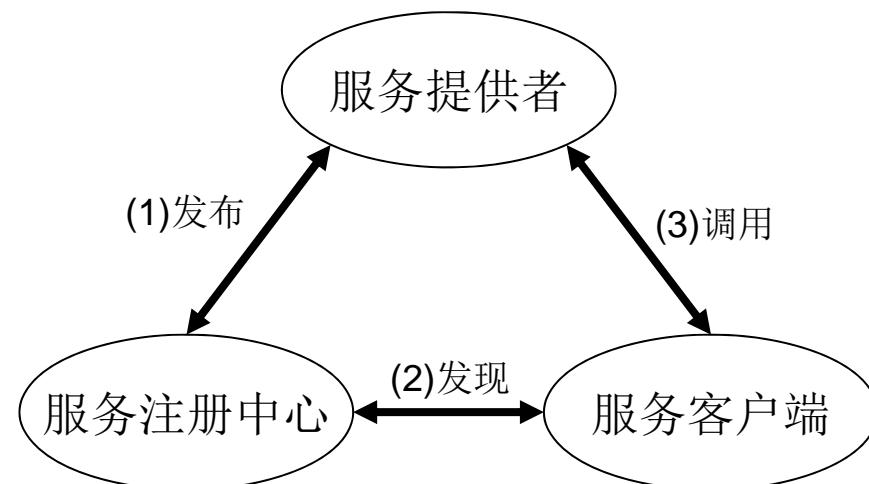
服务之间的“连接件”

- 通过接口，采用位置透明的、可互操作的协议进行调用，与客户端以“松散耦合”(loosely coupling)的方式绑定在一起。
- SOA中所有协议均是基于XML的文本文件。



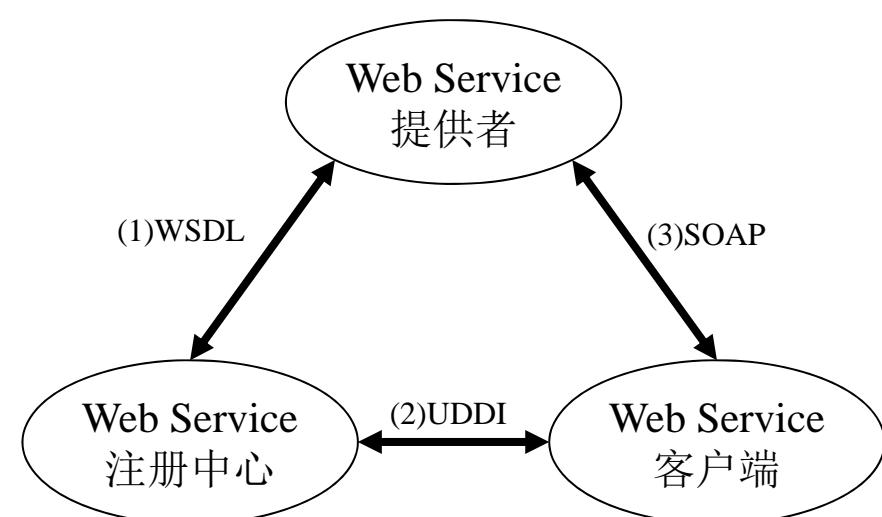
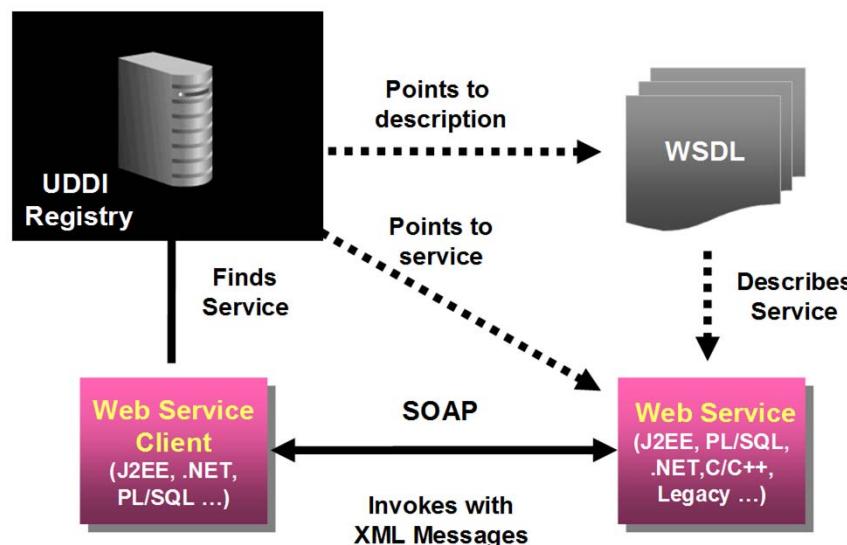
SOA的基本体系结构样式之一：发布-访问

- **发布(Publish)**: 为了使服务可访问，需要发布服务描述以使服务使用者可以发现它。
- **发现(Find)**: 服务请求者定位服务，方法是查询服务注册中心来找到满足其标准的服务。
- **调用(invoker)**: 在检索到服务描述之后，服务使用者继续根据服务描述中的信息来调用服务。

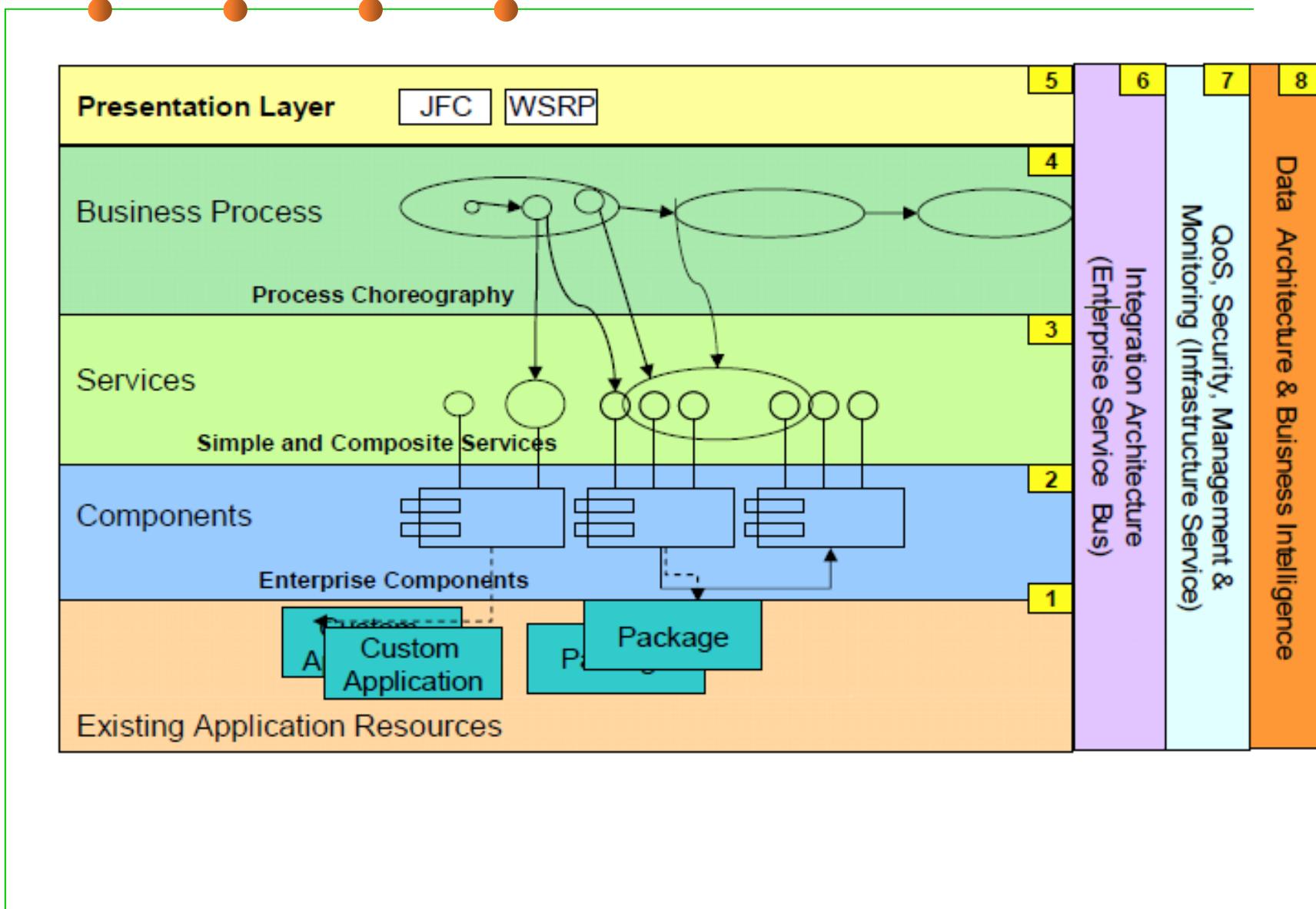


Web Service中该模式的实现机制

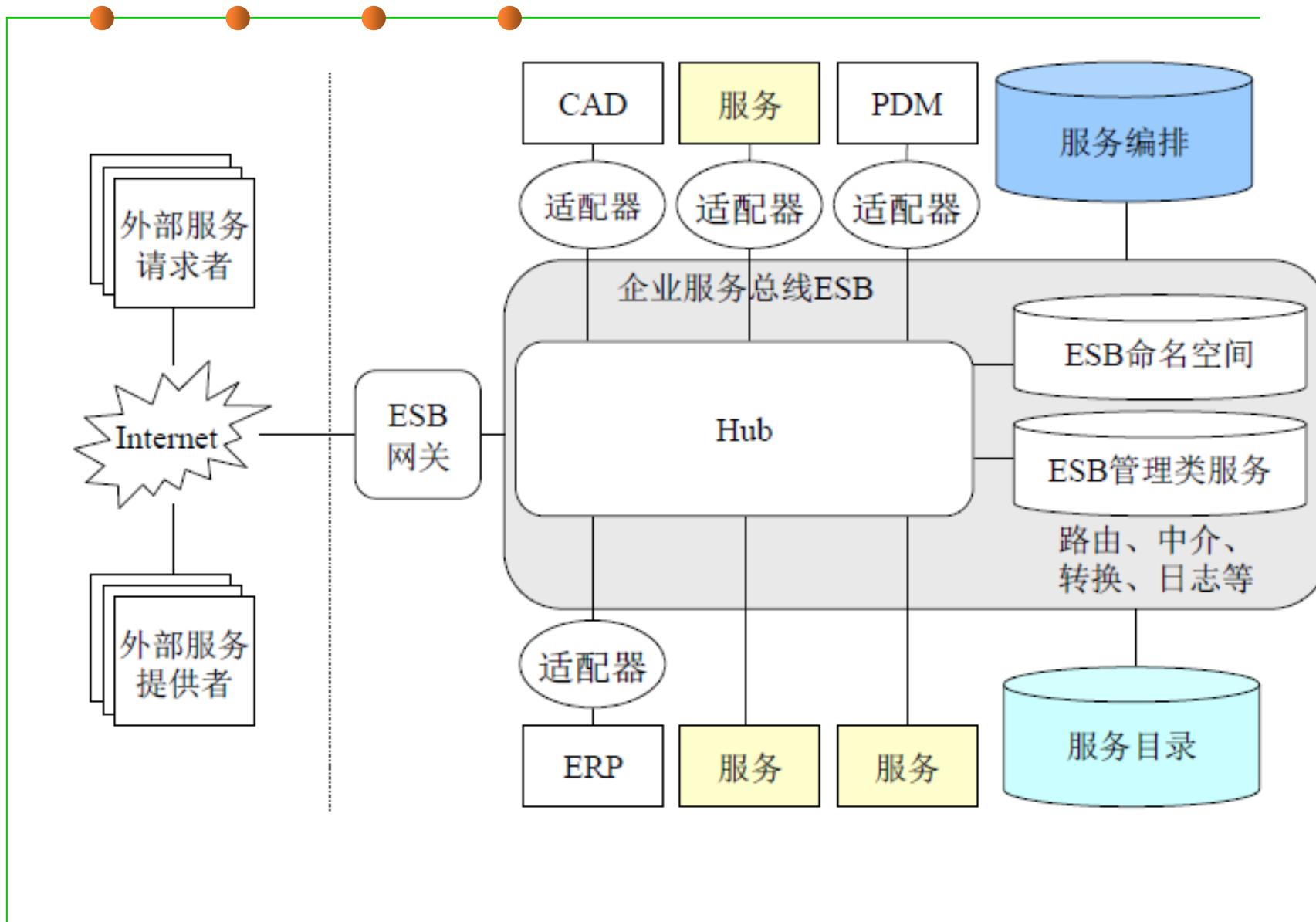
- **WSDL: Web服务描述语言**
 - 用于服务接口的描述 – What can the service do?
- **UDDI: 统一描述、发现和集成协议**
 - 服务使用者通过UDDI发现相应的服务并据此将服务集成在自身的系统中 – What kind of services are needed?
- **SOAP: 简单对象访问协议**
 - 用户在服务客户端与服务提供者之间传递信息，通过HTTP或JMS等各类基于文本的消息传递协议来运输



复杂模式：SOA分层架构



扩展结构：企业服务总线





哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

软件工程

结束

2015年10月16日