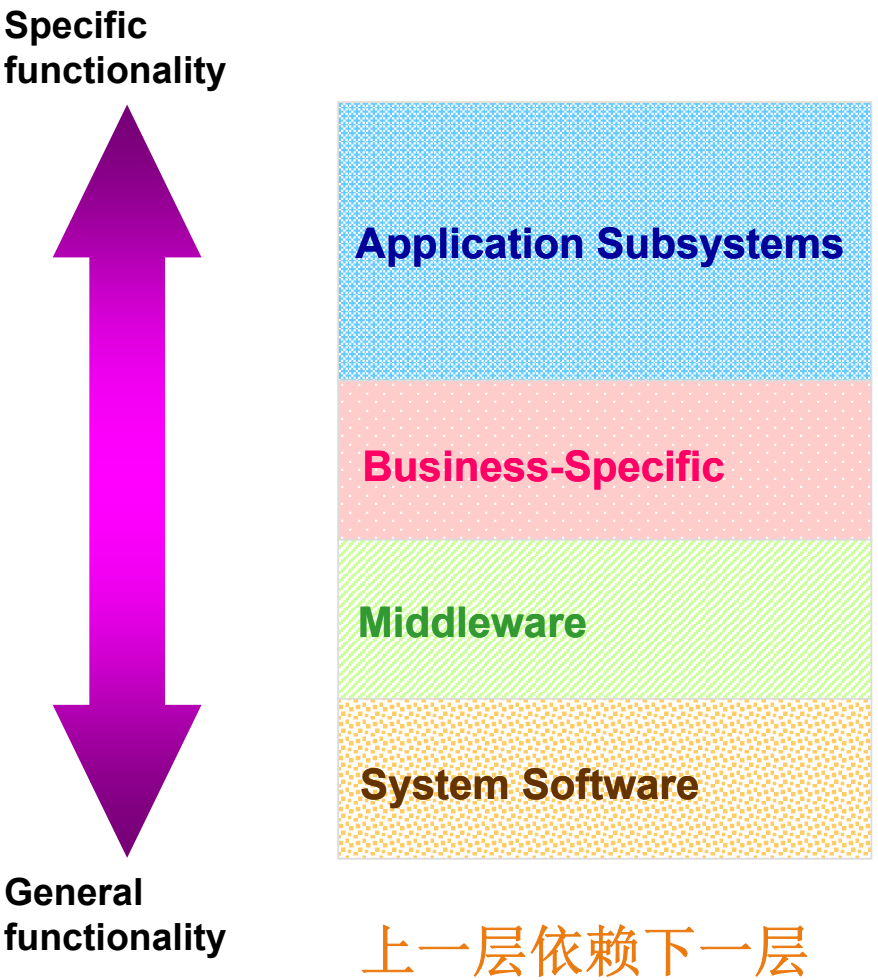
1) 层次架构（layered architecture）风格



Distinct application subsystems that make up an application — contains the value adding software developed by the organization.

Business specific — contains a number of reusable subsystems specific to the type of business.

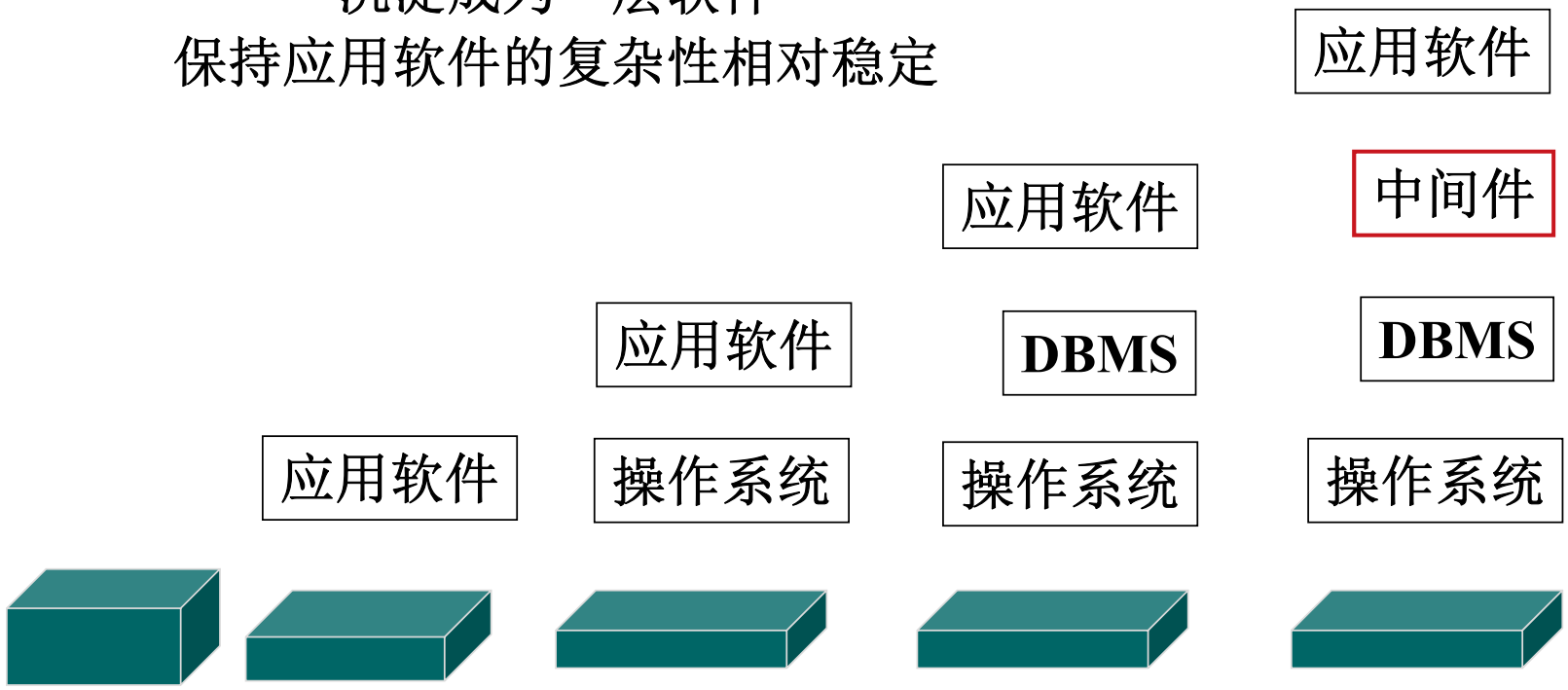
Middleware — offers subsystems for utility classes and platform-independent services for distributed object computing in heterogeneous environments and so on.

System software — contains the software for the actual infrastructure such as operating systems, interfaces to specific hardware, device drivers, and so on.

上一层依赖下一层

中间件

不断提取共性！
沉淀成为一层软件
保持应用软件的复杂性相对稳定



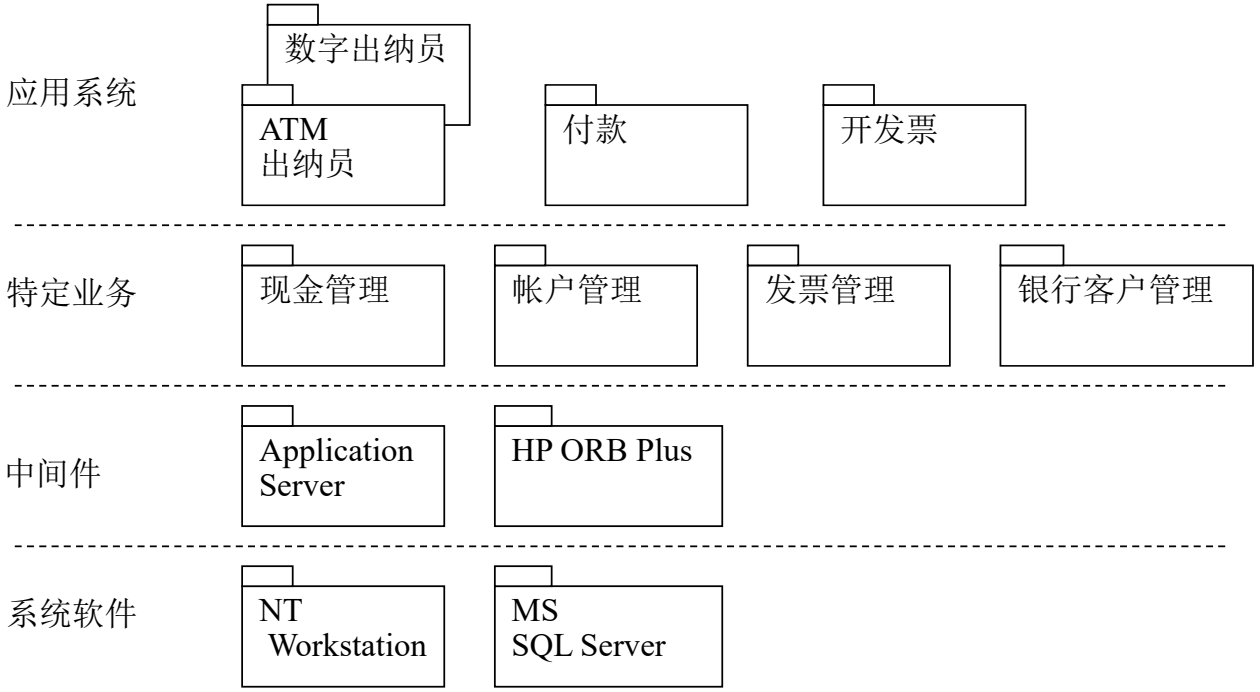
中间件的分类

- 1) 数据访问中间件
  - 允许应用程序和本地或者异地的数据库进行通信，并提供一系列的应用程序接口（如ODBC、JDBC等）。该类中间件技术上最成熟，但局限于与数据库相关的应用。
- 2) 消息中间件
  - 可以屏蔽平台和协议上的差异进行远程通信，实现应用程序之间的协同，如ActiveMQ、RabbitMQ，Kafka，RocketMQ 等，其优点在于提供高可靠的同步和异步通信，缺点在于不同的消息中间件产品之间不能互操作。
- 3) 远程过程调用RPC中间件
  - 解决了平台异构的问题，但不支持异步操作。如gRPC（Google RPC），支持多种语言（如Java、C++、C#、Go、Node、PHP、Python、Ruby）。

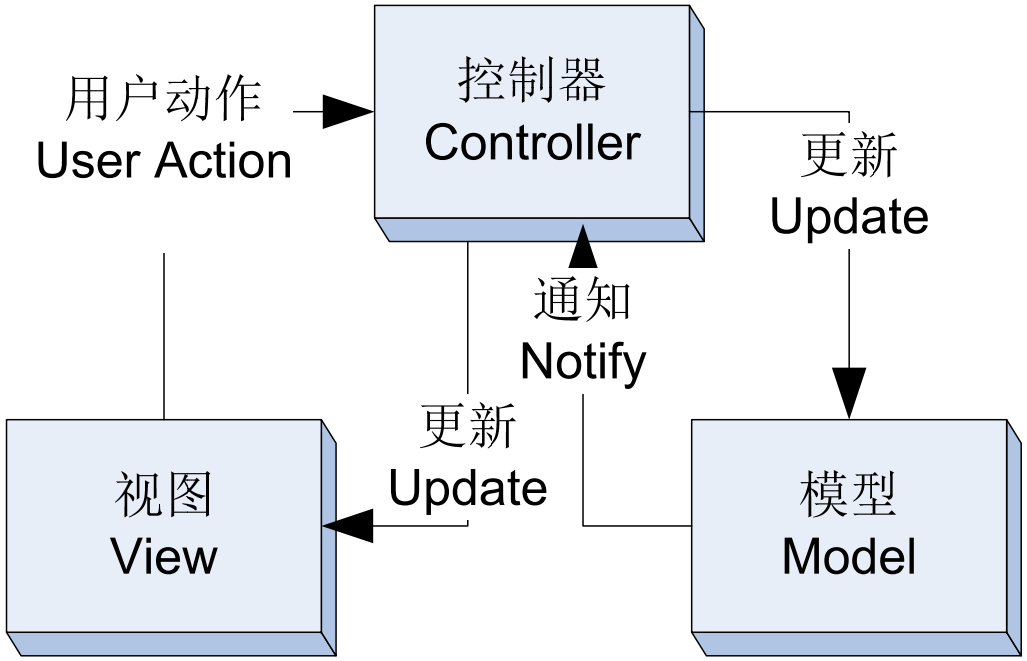
中间件的分类 （2）

- 4）事务中间件
  - 是在分布、异构环境下提供保证事务完整性和数据完整性的一种平台，如BEA的TUXEDO、IBM的CICS、微软的MTS 。其优势在于对关键业务的支持，但机制复杂、对用户要求较高。
- 5）分布对象中间件
  - 在分布、异构的网络计算环境中，可以将各种分布对象有机地结合在一起，完成系统的快速集成。主流标准有Microsoft的 DNA/COM+、OMG的OMA/CORBA、Sun的 J2EE / EJB。Weblogic, Websphere, Jboss, .Net等应用服务器都包含了分布对象中间件，也有如Orbix、HP ORB等独立产品。
- 6）分布式服务中间件
  - Web服务中间件：通过网络对松散耦合的粗粒度应用服务进行分布式部署、组合和使用，其标准是SOA，Web服务是其中的一种实现。应用服务器都包含了Web服务中间件，也有AXIS2、HP Web Services Platform等独立产品。
  - 微服务中间件：轻载的细粒度服务的中间件，如Dubbo.

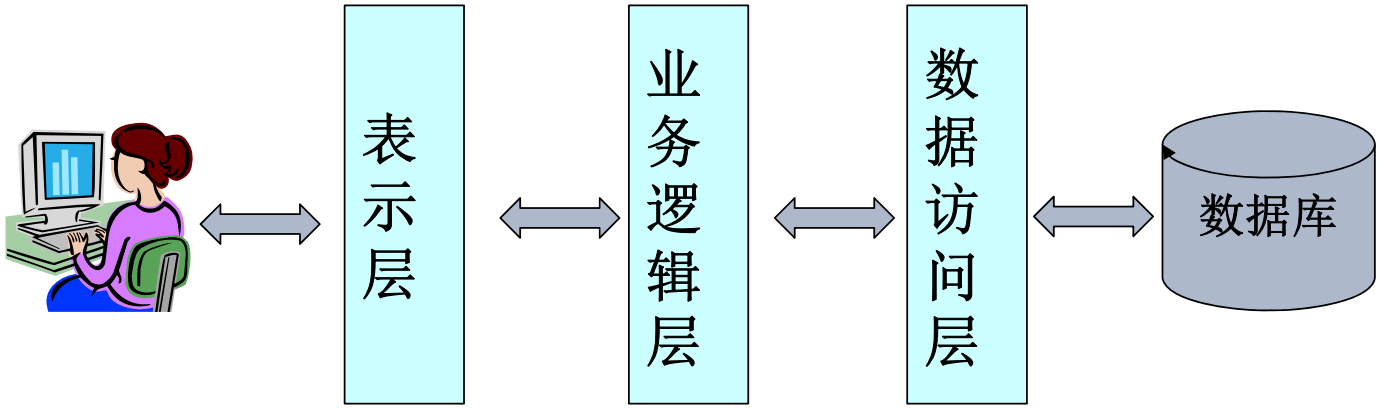
层次架构实例


应用系统
数字出纳员
ATM
出纳员
付款
开发票
特定业务
现金管理
帐户管理
发票管理
银行客户管理
中间件
Application
Server
HP ORB Plus
系统软件
NT
Workstation
MS
SQL Server

2) MVC

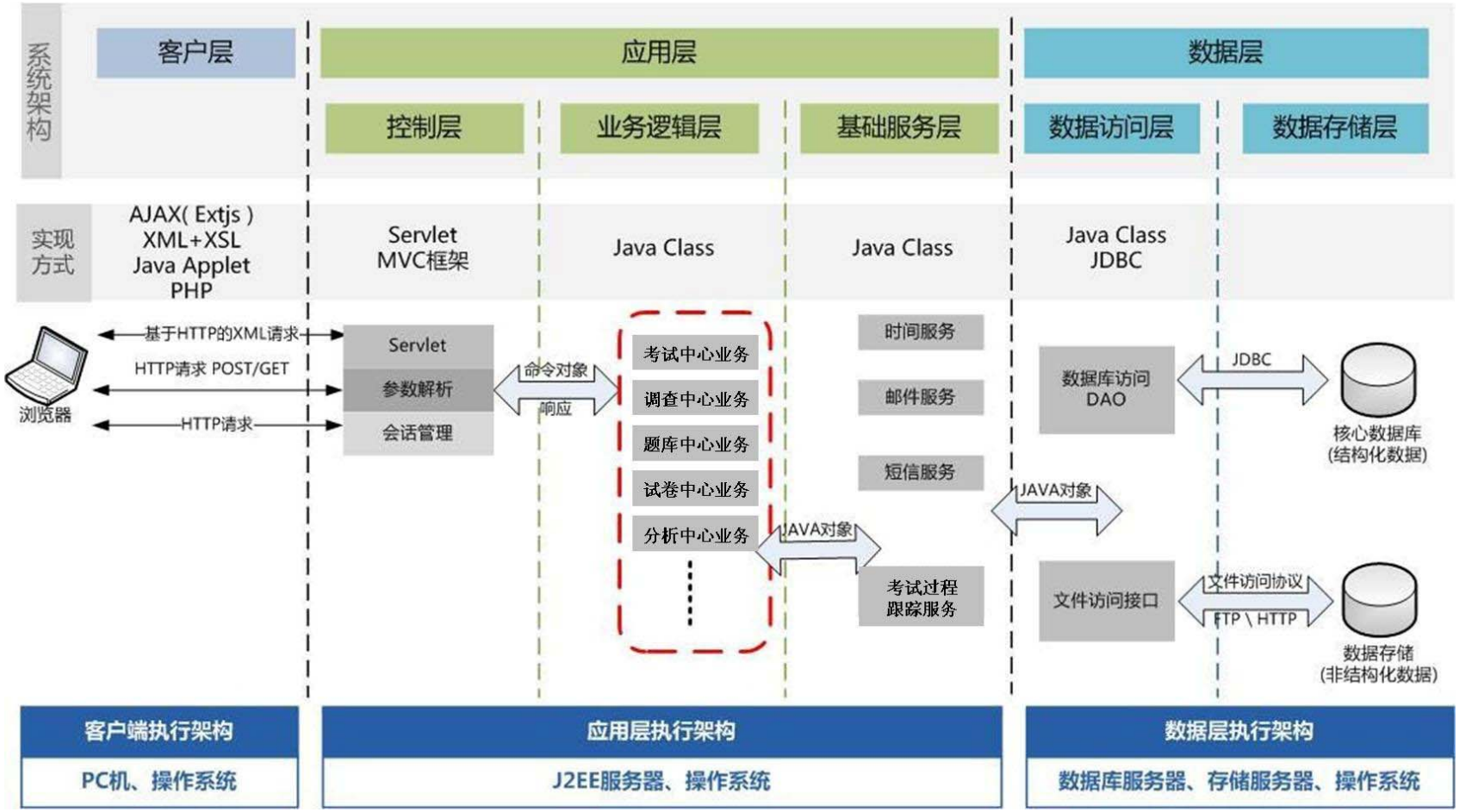

用户动作
User Action
控制器
Controller
更新
Update
通知
Notify
更新
Update
视图
View
模型
Model

3) 3 Tiers



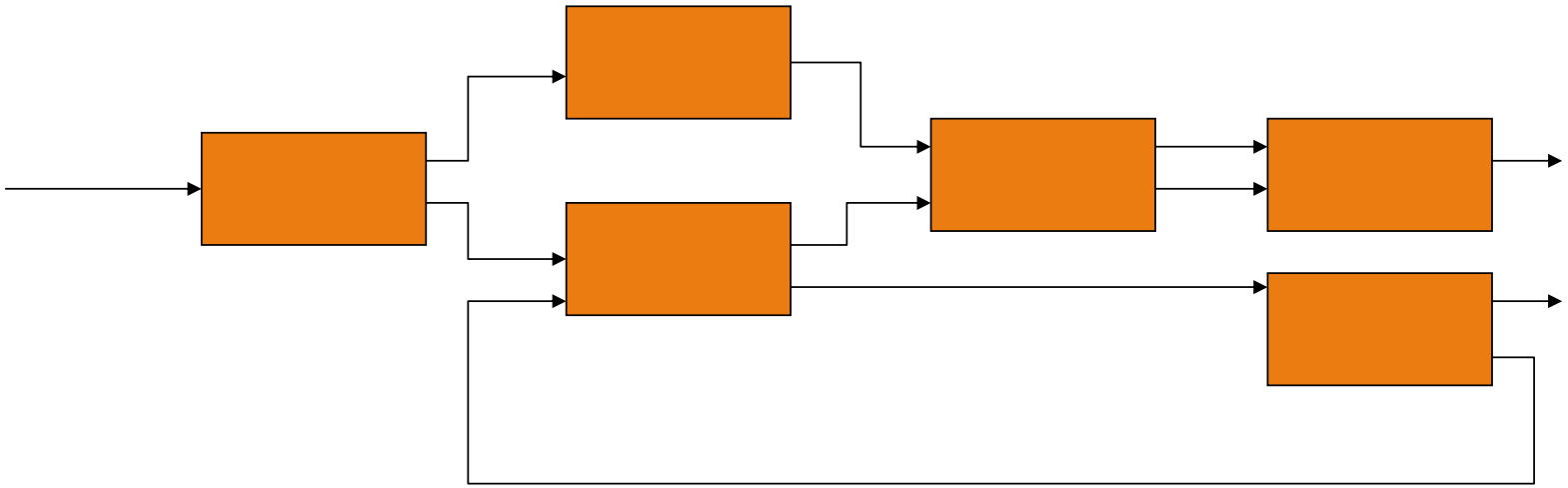
- 表示层：负责向用户呈现界面，并接收用户请求发送给业务逻辑层；
- 业务逻辑层：负责执行业务逻辑以处理用户请求，并调用数据访问层提供的持久性操作；
- 数据访问层：负责执行数据库持久性操作。

举例：基于Web的在线考试系统的架构图



4) 管道和过滤器（Pipes and Filters）

- In this style, each component has a set of inputs and a set of outputs.
- A component reads streams of data on its inputs and produces streams of data on its output.



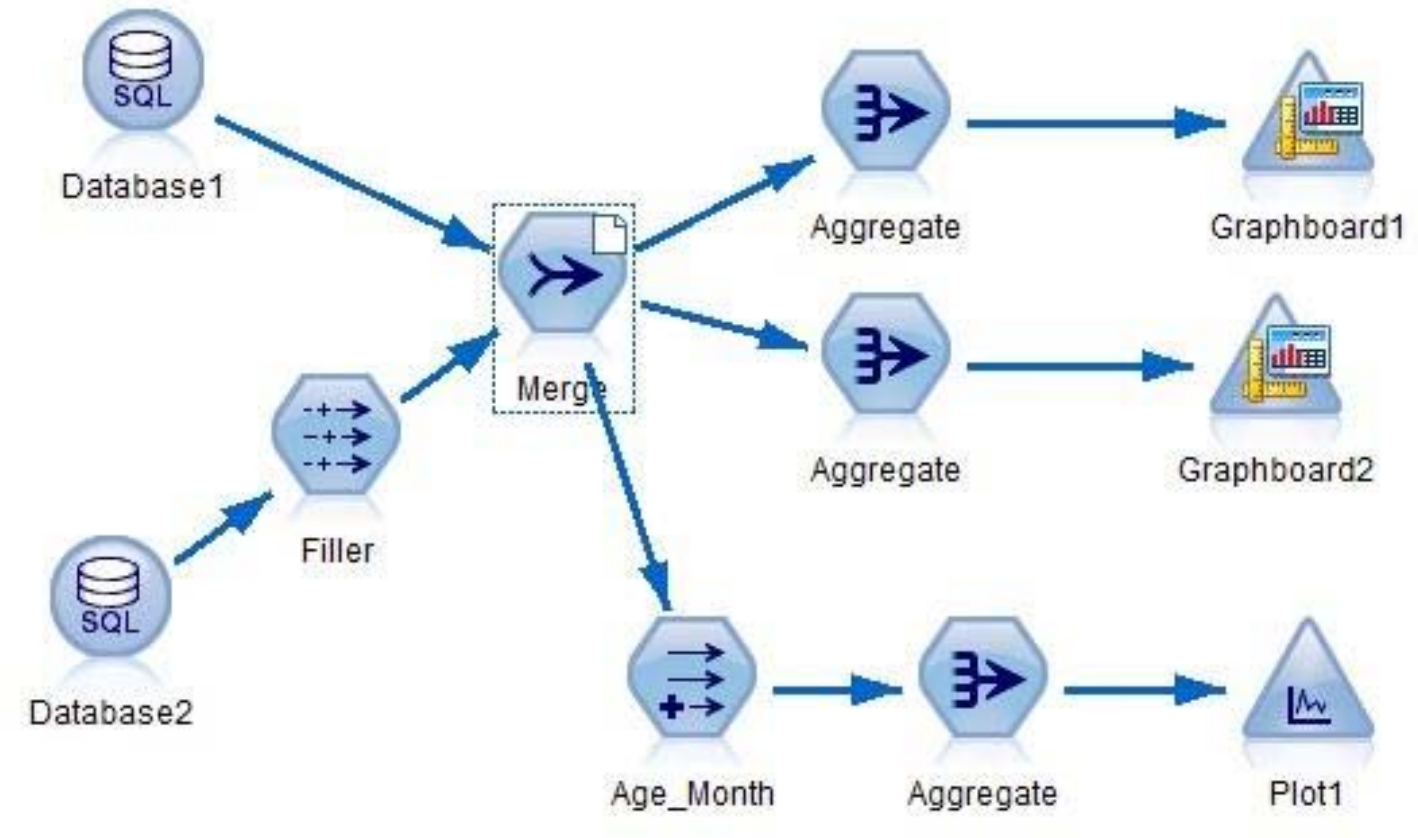
举例

- Linux的Shell程序可以看做是典型的管道与过滤器架构的例子
- 例如下面的Shell脚本：

```
$cat TestResults | sort | grep Good
```

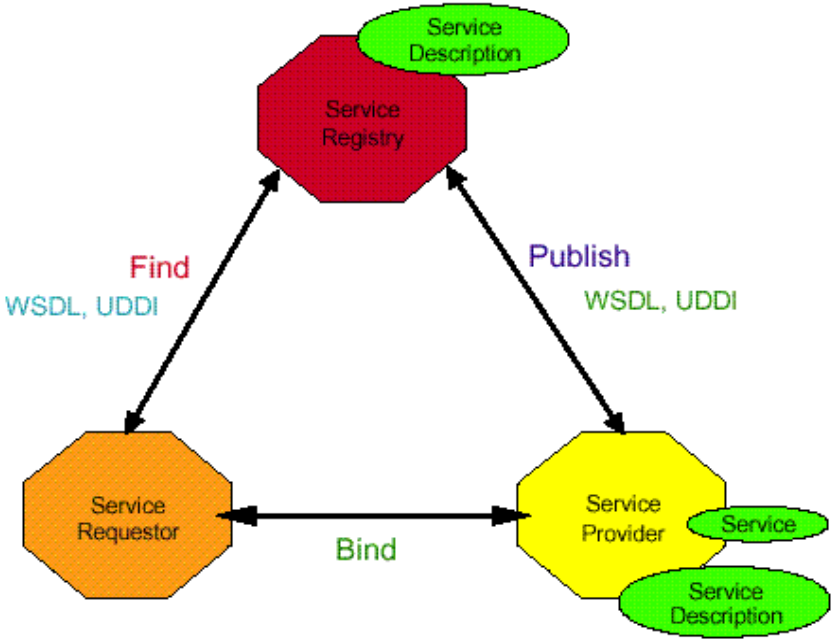
会将TestResults文件的文本进行排序，然后找出其中包含单词Good的行，并显出在屏幕上。Shell命令cat、sort和grep依次执行，就构成了一个管道-过滤器架构。

举例


SQL
Database1
SQL
Database2
Filler
Merge
Aggregate
Aggregate
Aggregate
Graphboard1
Graphboard2
Age_Month
Plot1

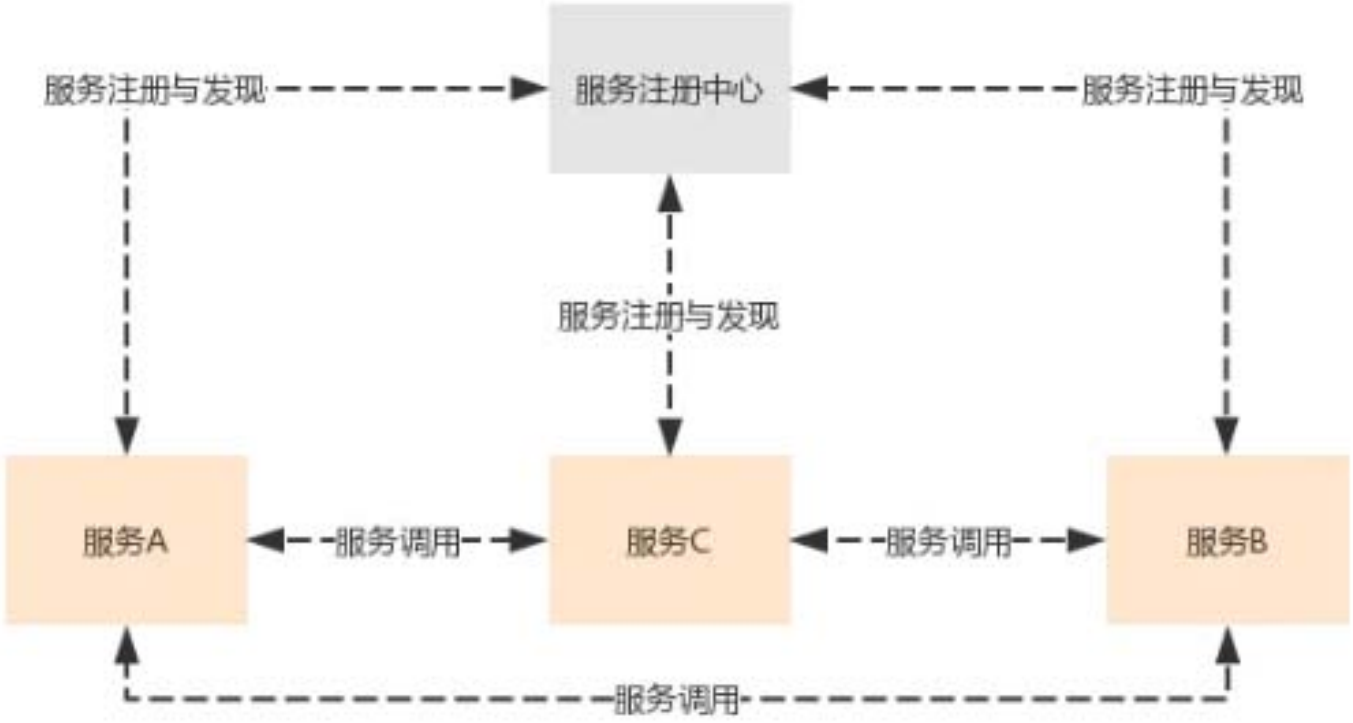
5）服务与微服务的架构风格

服务架构风格

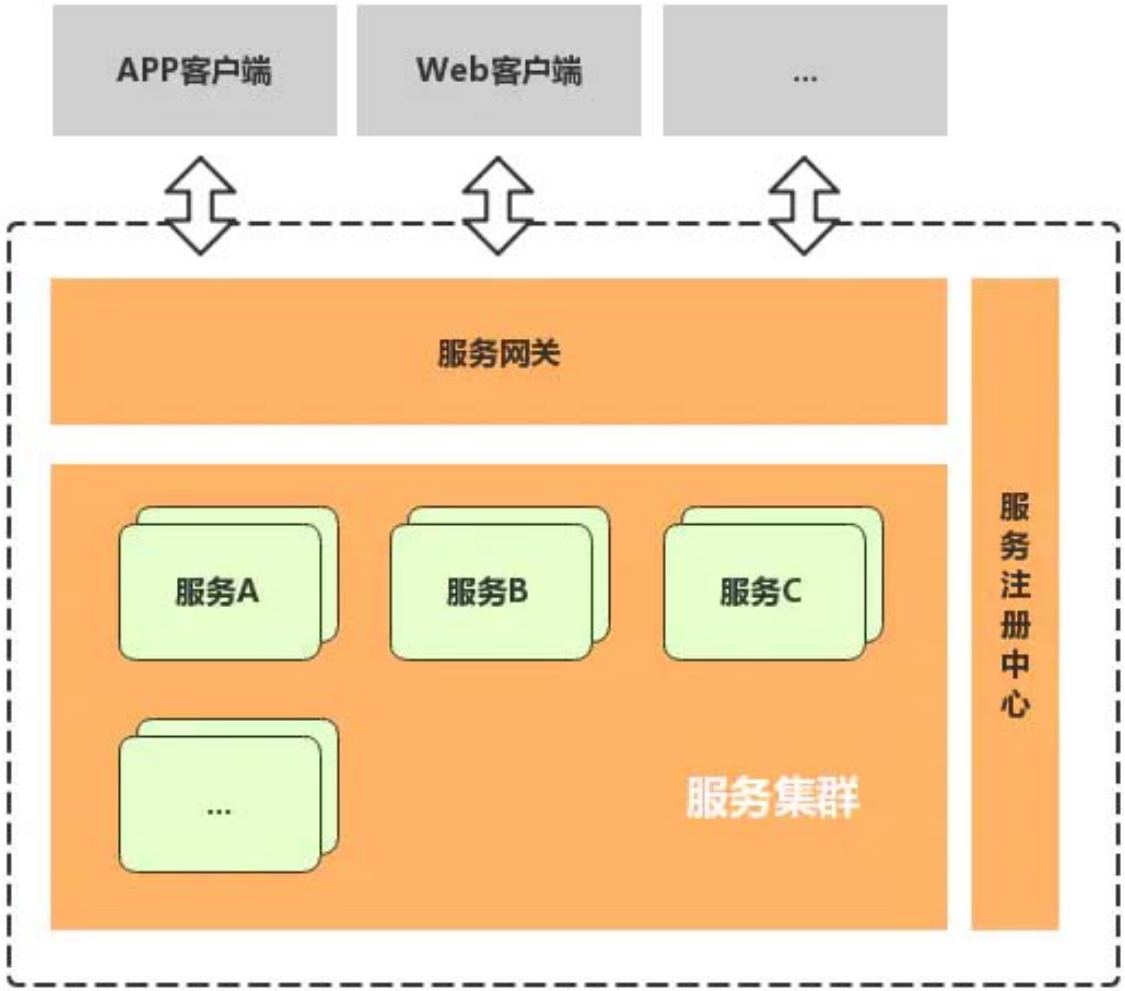


- 服务的抽象性（基于接口的编程）
- 服务的自治性（实现分布式应用）
- 服务间的松耦合式绑定，基于消息进行通信
- 服务的粗粒度

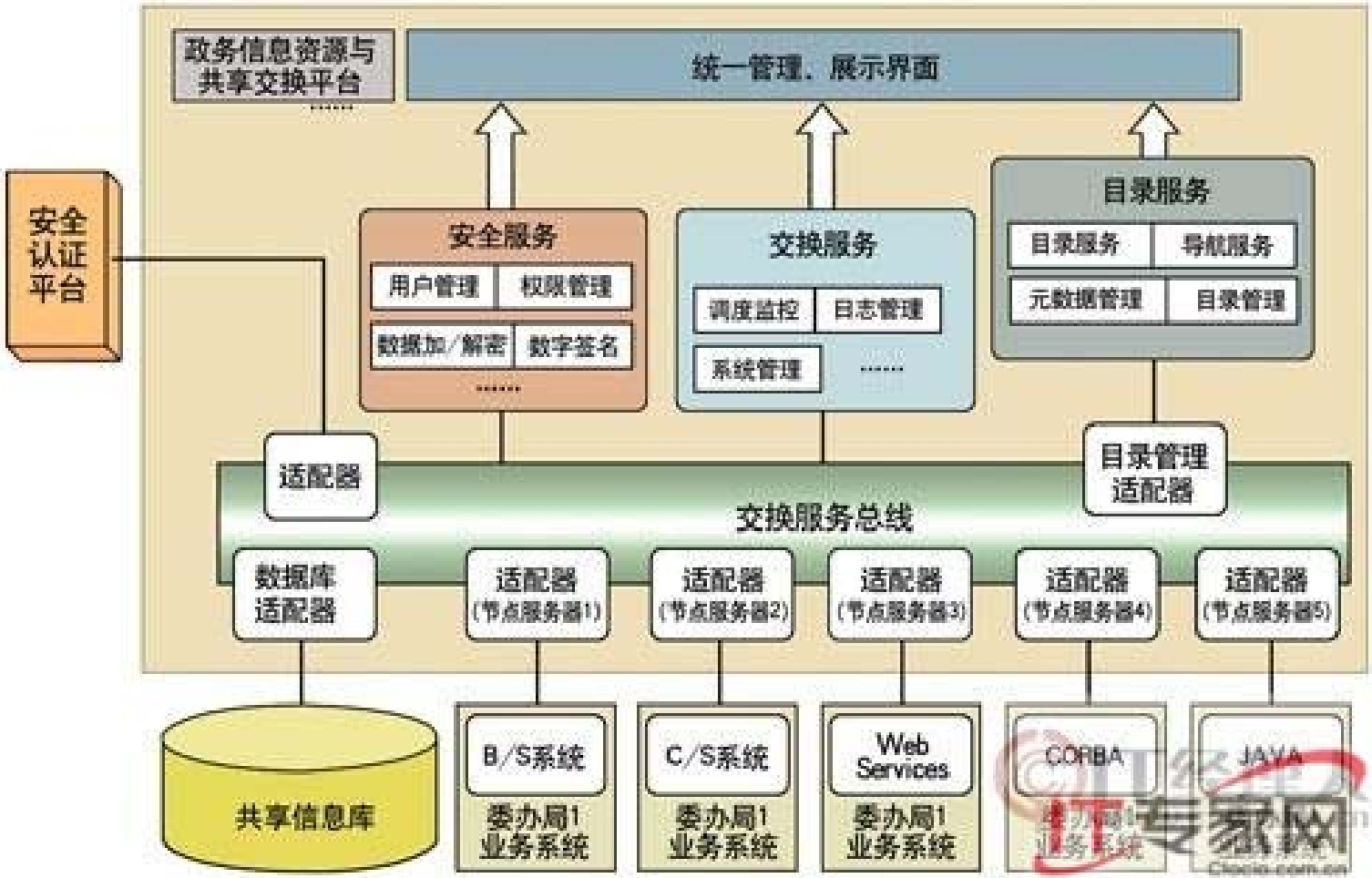
服务的注册、发现和调用



基于SOA的C/S系统的典型架构



举例（政务信息资源与共享交换平台）

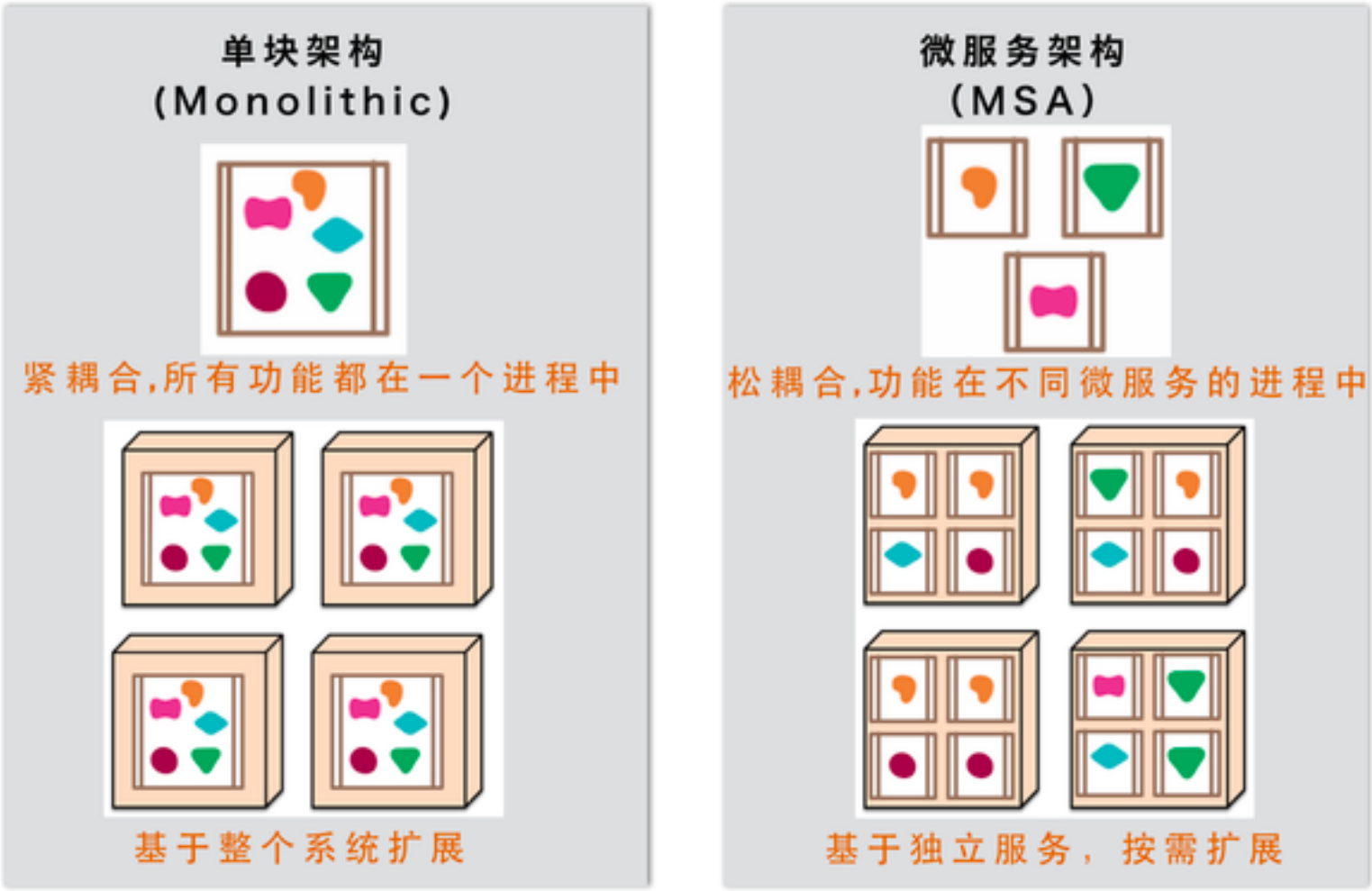


微服务架构风格

- **微服务架构风格**是一种使用一套小服务来开发单个应用的方式途径，每个服务运行在自己的进程中，通过轻量的通讯机制联系，经常是基于HTTP资源API，这些服务基于业务能力构建，能够通过自动化部署方式独立部署，这些服务自己有一些小型集中化管理，可以是使用不同的编程语言编写。

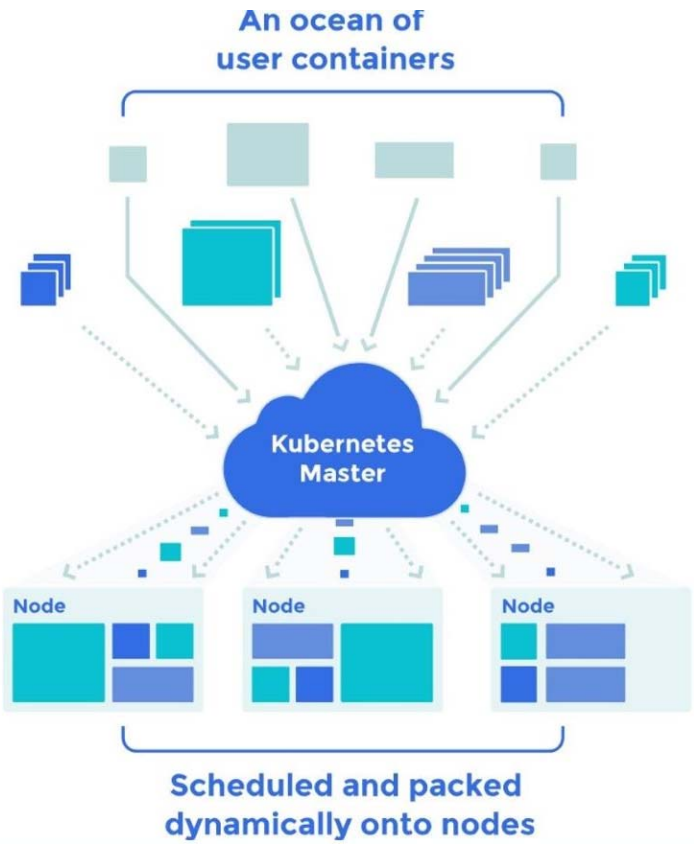
- 和SOA不同：SOA倡导粗粒度服务，而它是细粒度服务。同时，微服务采用“智能终端和哑管道”，它们拥有自己的领域逻辑，以类似Unix管道过滤方式运行，接受到一个请求，使用相应的逻辑，产生一个响应，这些都可以使用RESTful方式编排，而不是使用复杂的协议如WS-Choreography 或 BPEL或ESB指挥控制。

Monolithic application Vs. Microservices

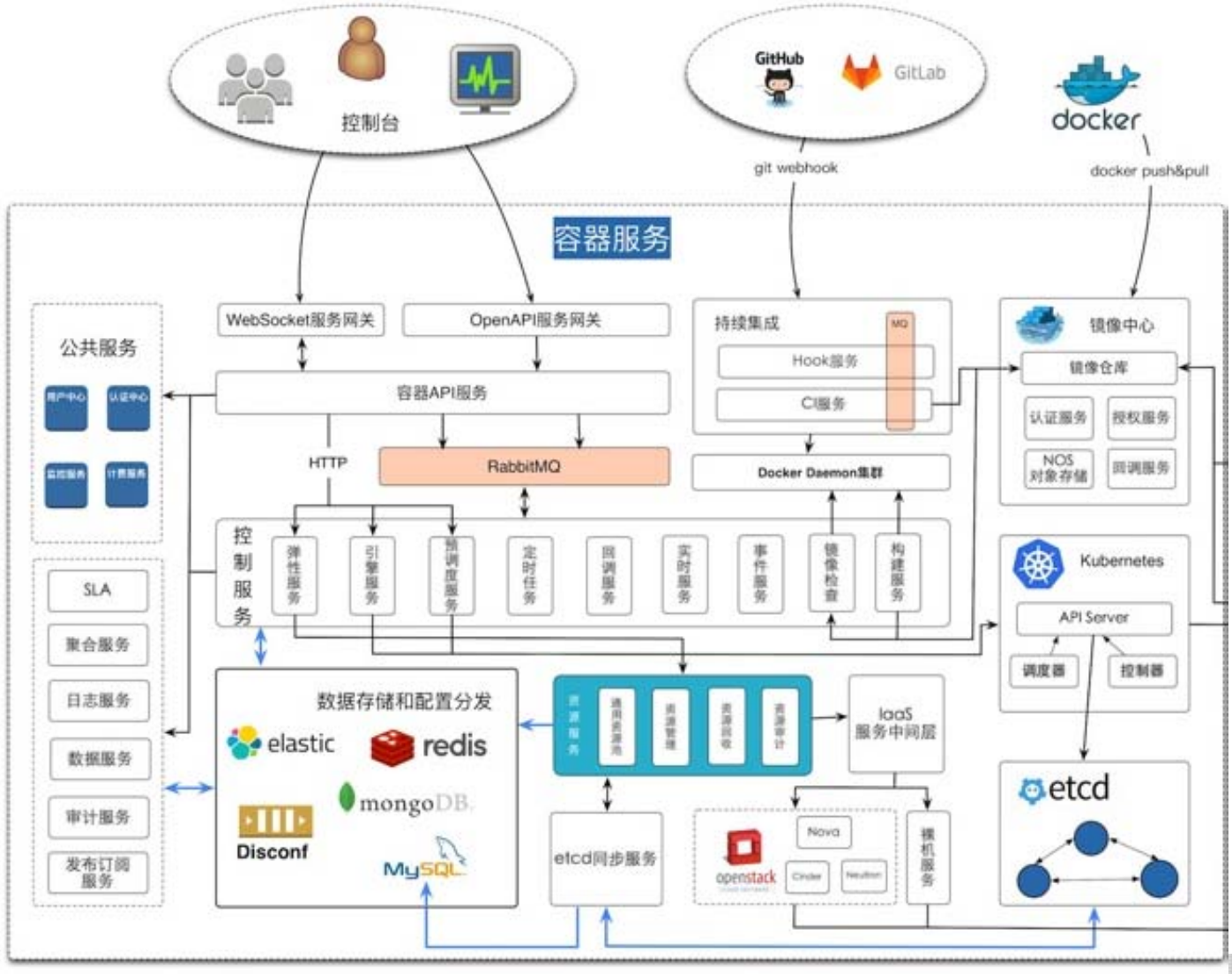


容器

- 微服务的管理问题：
  - 因为服务通常部署在多个主机上，很难持续跟踪指定服务究竟运行在某台主机上。
  - 因为微服务架构使用的主机容量往往小于Monolithic架构，随着微服务架构不停的横向扩展，主机数量将以一个非常恐怖的速度增长。
- 解决方案：容器
  - 不同容器共享相同的内核，容器的共享和发布非常简单
  - 容器之间进行了完全的隔离，简易了不同语言开发的微服务代码部署
  - 目前最流行的容器实现是 Docker
  - Docker编排和集群管理工具：Kubernetes等



举例（网易云）

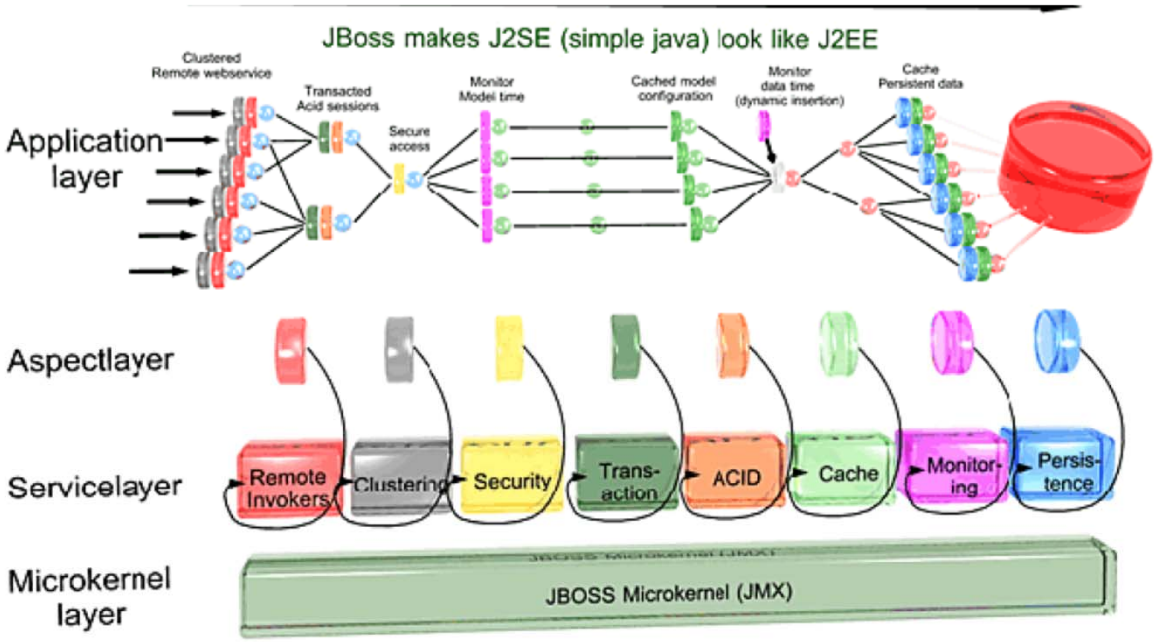

控制台
GitHub
GitLab
git webhook
docker
docker push&pull
容器服务
WebSocket服务网关
OpenAPI服务网关
公共服务
容器API服务
HTTP
RabbitMQ
持续集成
MQ
Hook服务
CI服务
Docker Daemon集群
镜像中心
镜像仓库
认证服务
授权服务
NOS
对象存储
回调服务
控制服务
弹性服务
引擎服务
调度服务
定时任务
回调服务
实时服务
事件服务
镜像检查
构建服务
SLA
聚合服务
日志服务
数据服务
审计服务
发布订阅服务
数据存储和配置分发
elastic
redis
mongoDB.
Disconf
MySQL
资源服务
通用资源池
资源管理
资源回收
资源审计
IaaS
服务中间层
etcd同步服务
openstack
Nova
Cinder
Neutron
裸机服务
Kubernetes
API Server
调度器
控制器
etcd

6) 仓库风格

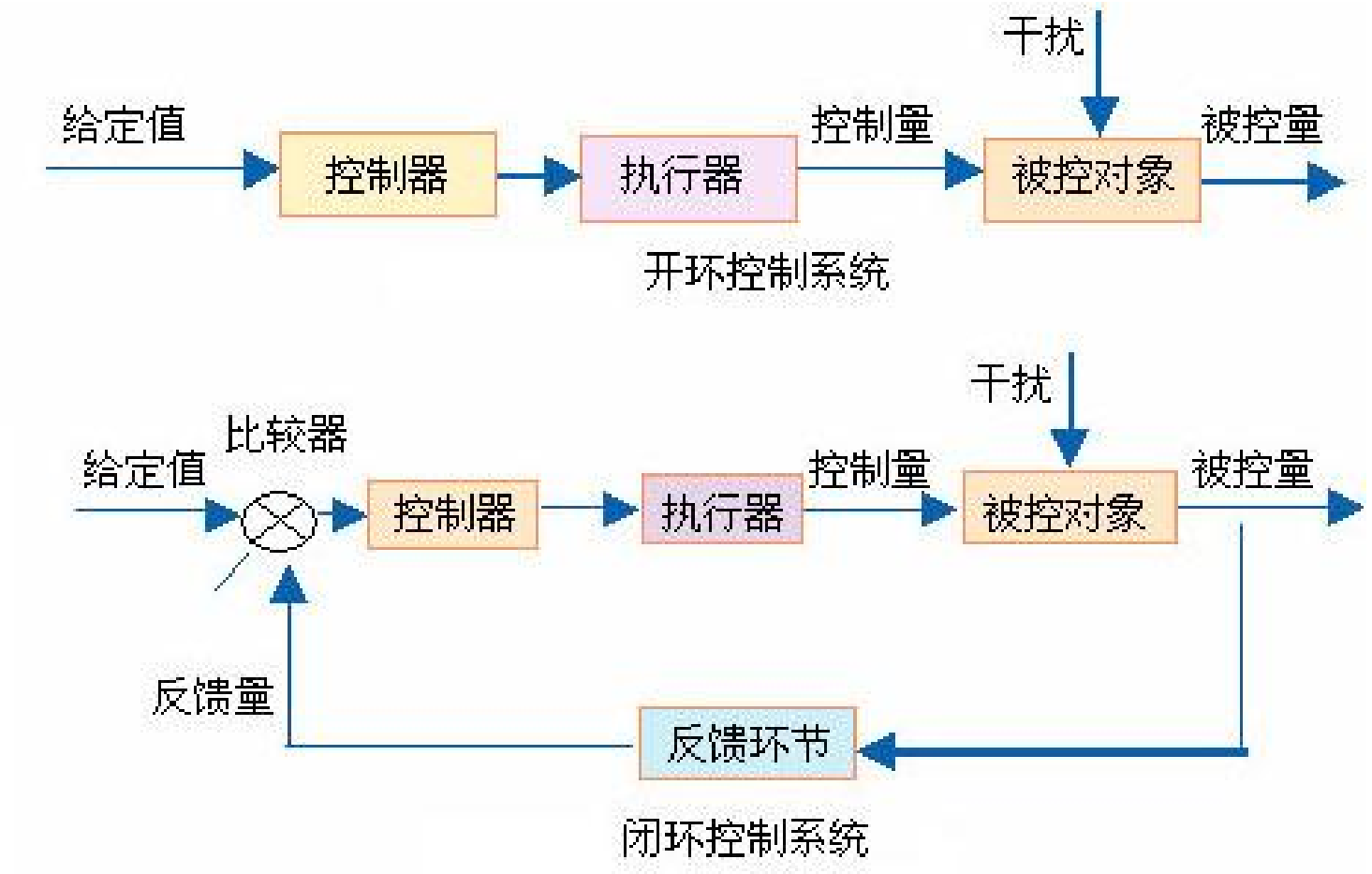
- 仓库风格是以数据为中心的系统架构，它细分为：
  - 数据库系统
    - 以数据库为核心，各个构件存取数据。
  - 超文本系统
    - 用超链接的方法，将各种不同空间的
    - 文字、图片等信息组织在一起的网状文本
  - 黑板系统
    - 为参与问题解决的知识源提供了共享的数据表示，这些数据表示是与应用相关的。在黑板系统中，控制流是由黑板数据的状态决定的，而并非按照某个固定的顺序执行。

7) 微内核风格

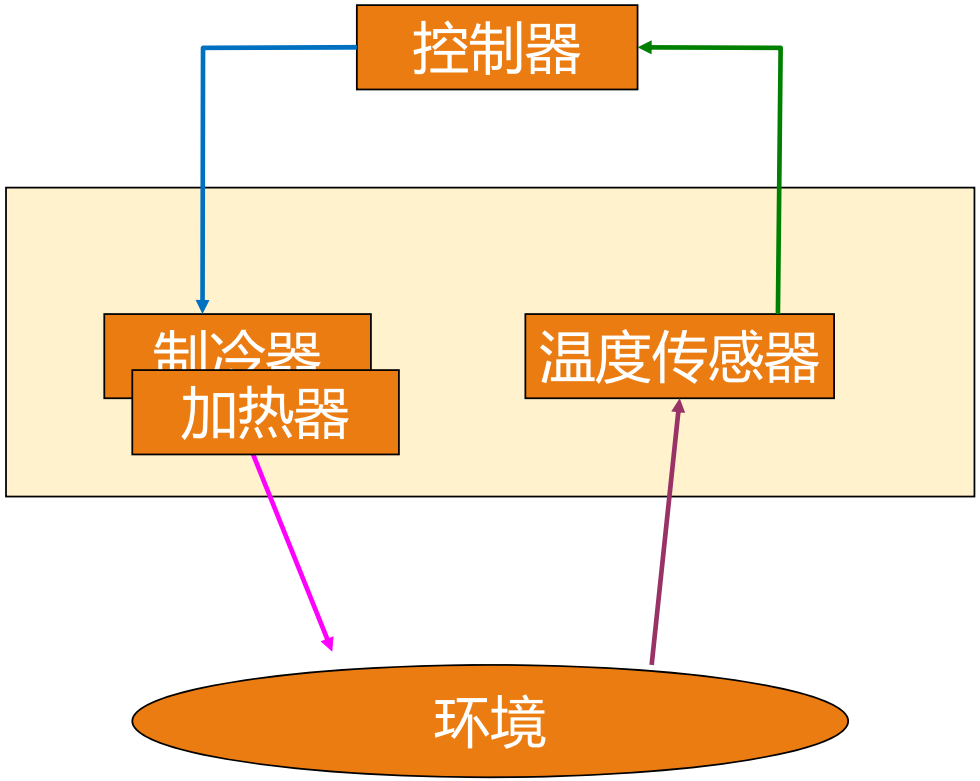
- **微内核**概念来源与操作系统领域。微内核是提供了操作系统核心功能的内核，它只需占用很小的内存空间即可启动，并向用户提供了标准接口，以使用户能够按照模块化的方式扩展其功能。现在大多数操作系统都采用了微内核架构。



8) 开环和闭环控制风格


干扰
给定值
控制器
执行器
控制量
被控对象
被控量
开环控制系统
干扰
比较器
给定值
控制器
执行器
控制量
被控对象
被控量
反馈量
反馈环节
闭环控制系统

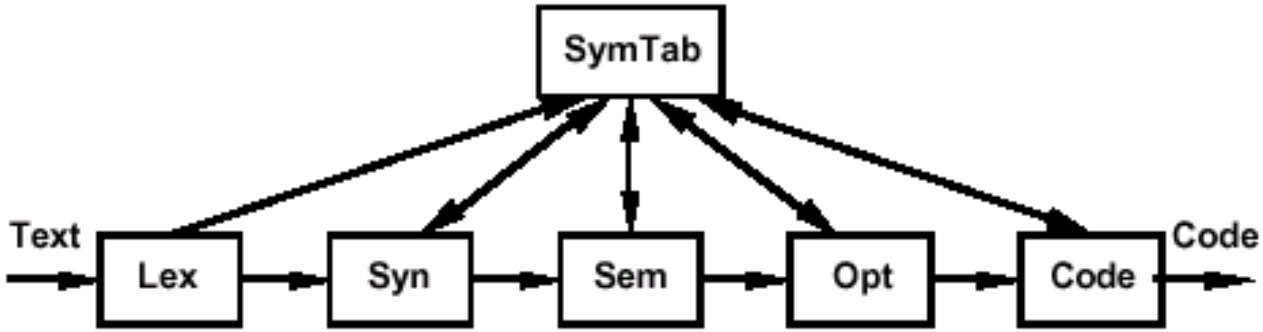
举例: 空调控制软件的逻辑架构图



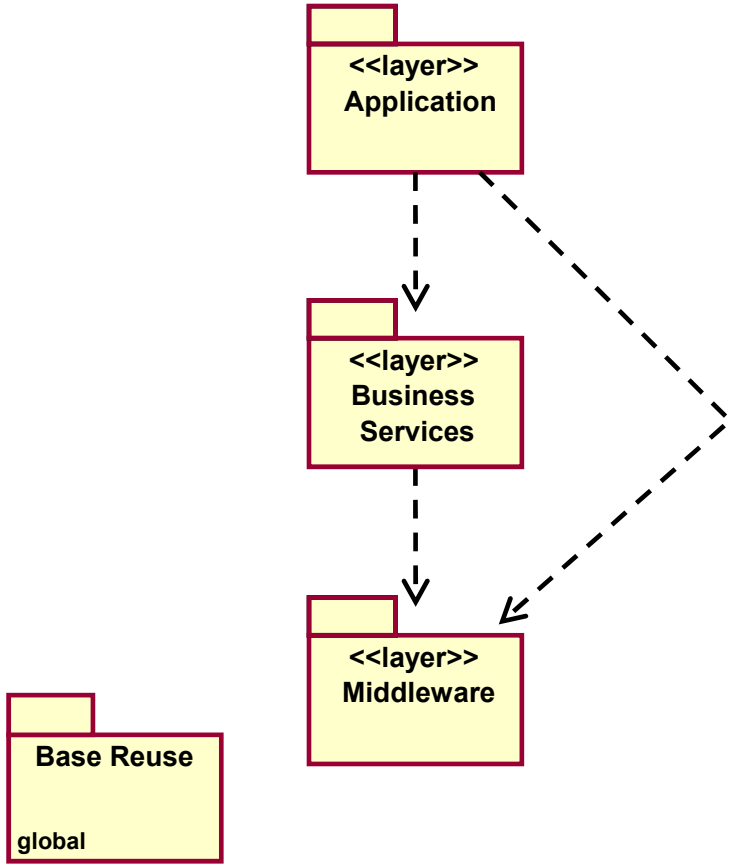
综合举例：编译器的逻辑架构图

（1）传统的编译器的架构图

（2）具有共享符号表的编译器的架构图



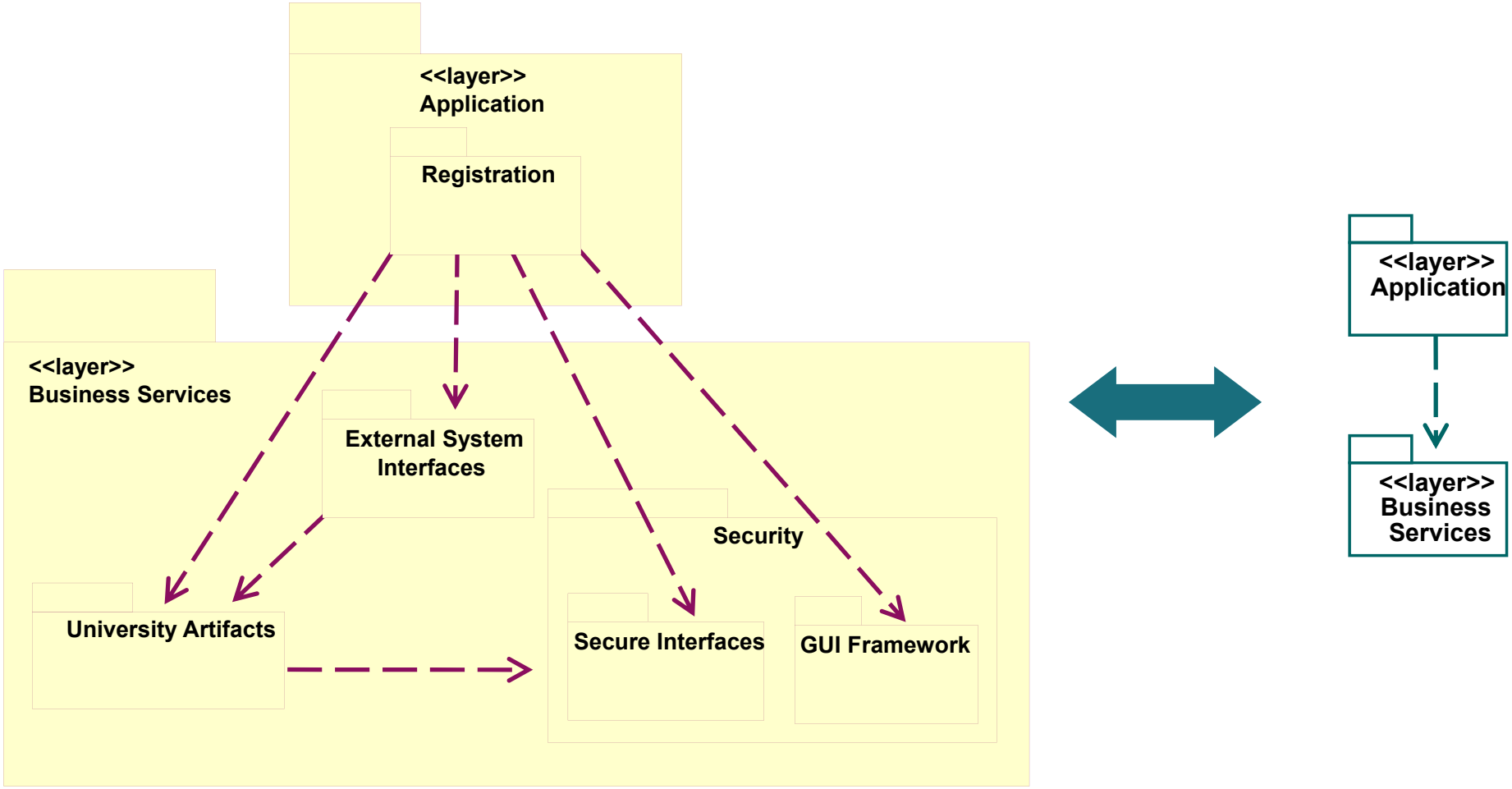
UML的逻辑视图的描述



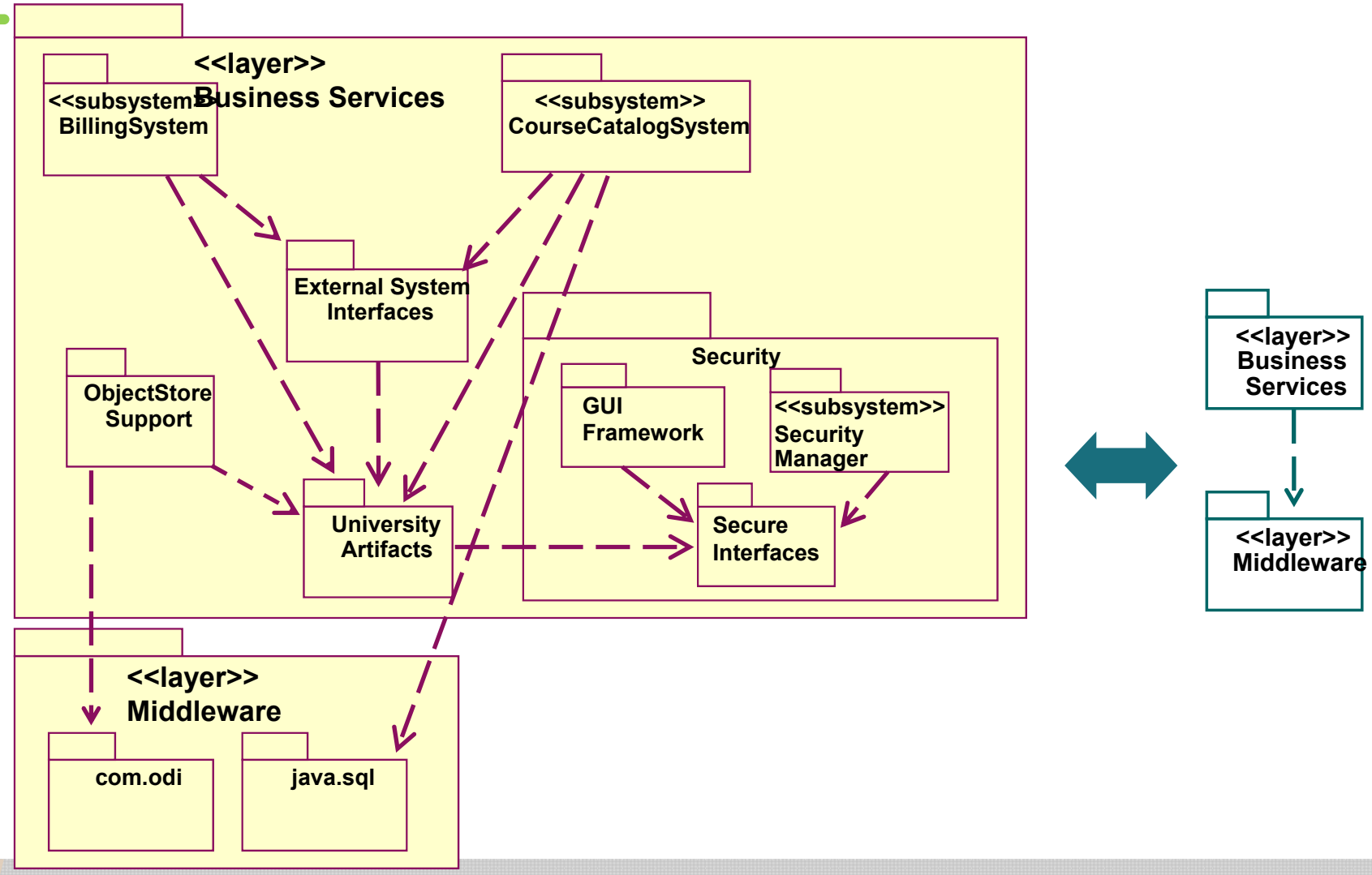
Necessary because the Application Layer must have access to the core distribution mechanisms provided with Java RMI.

选课系统的逻辑架构

Example: Application Layer

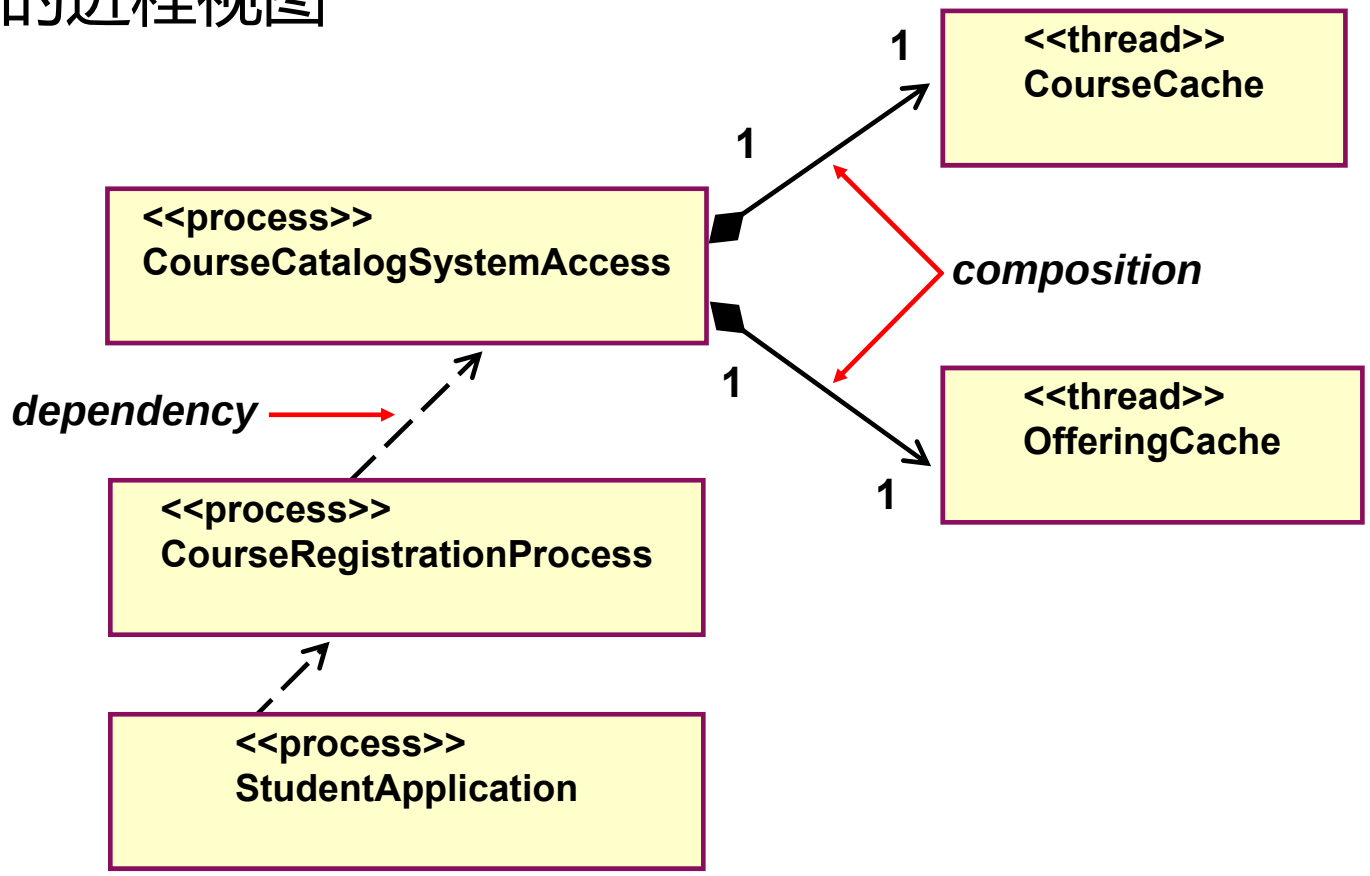


Example: Business Services Layer Context



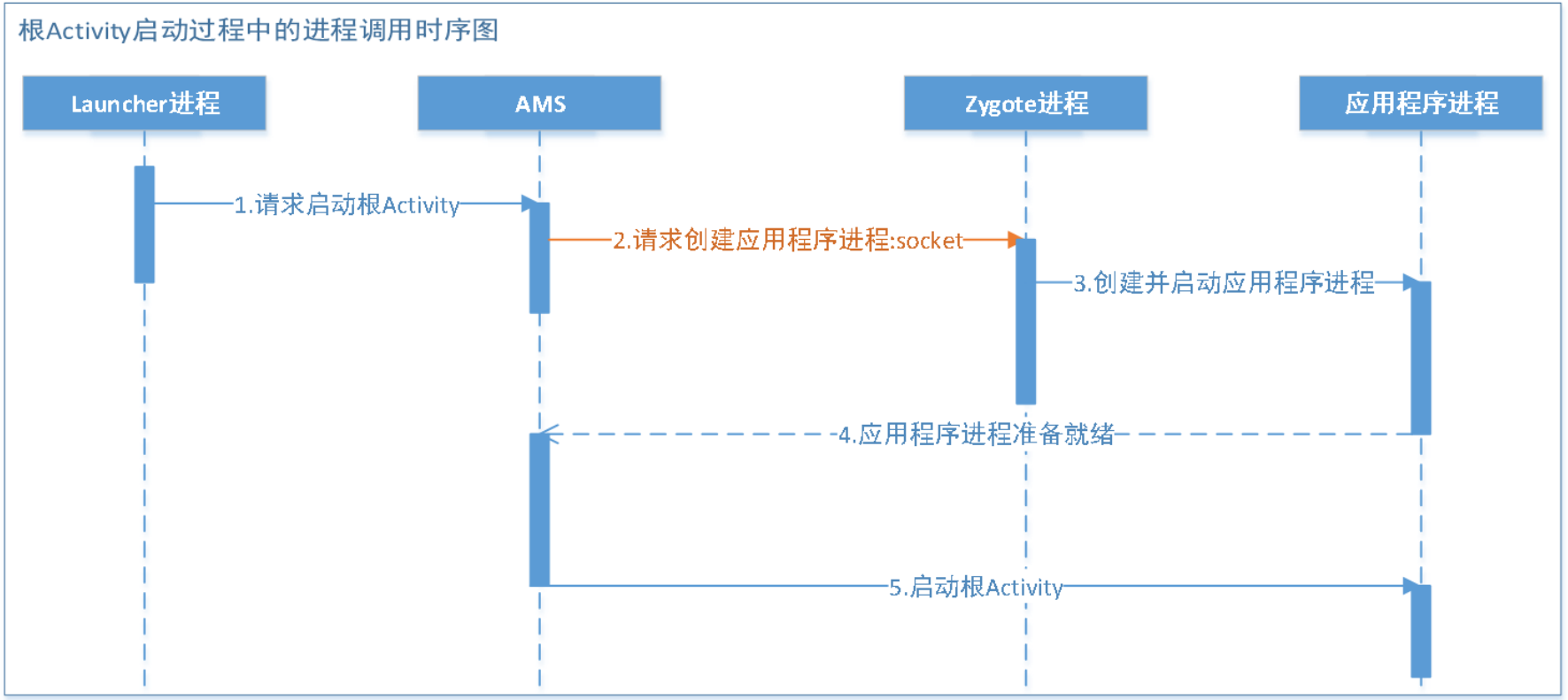
UML的进程视图的描述

选课系统的进程视图

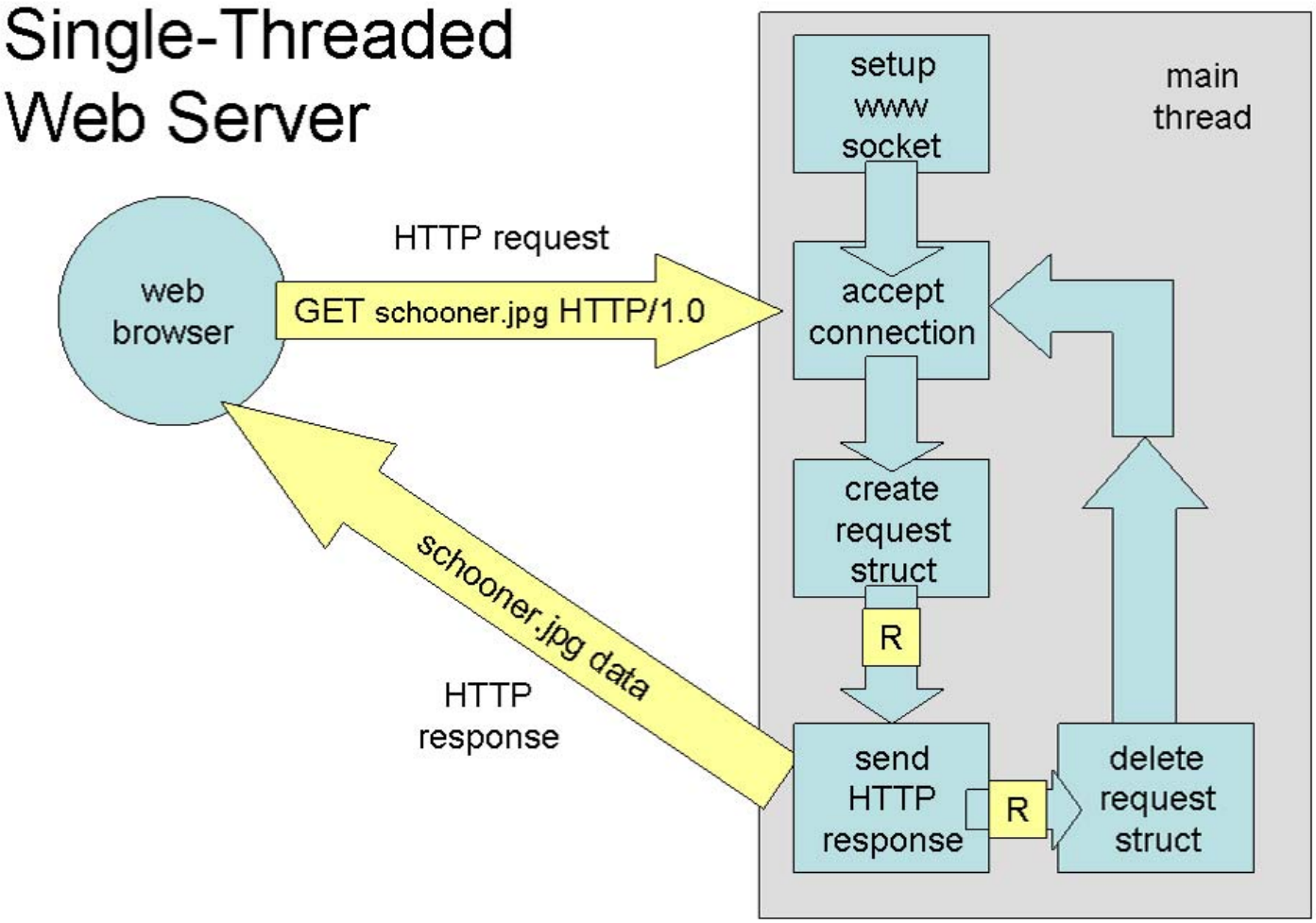


进程间交互

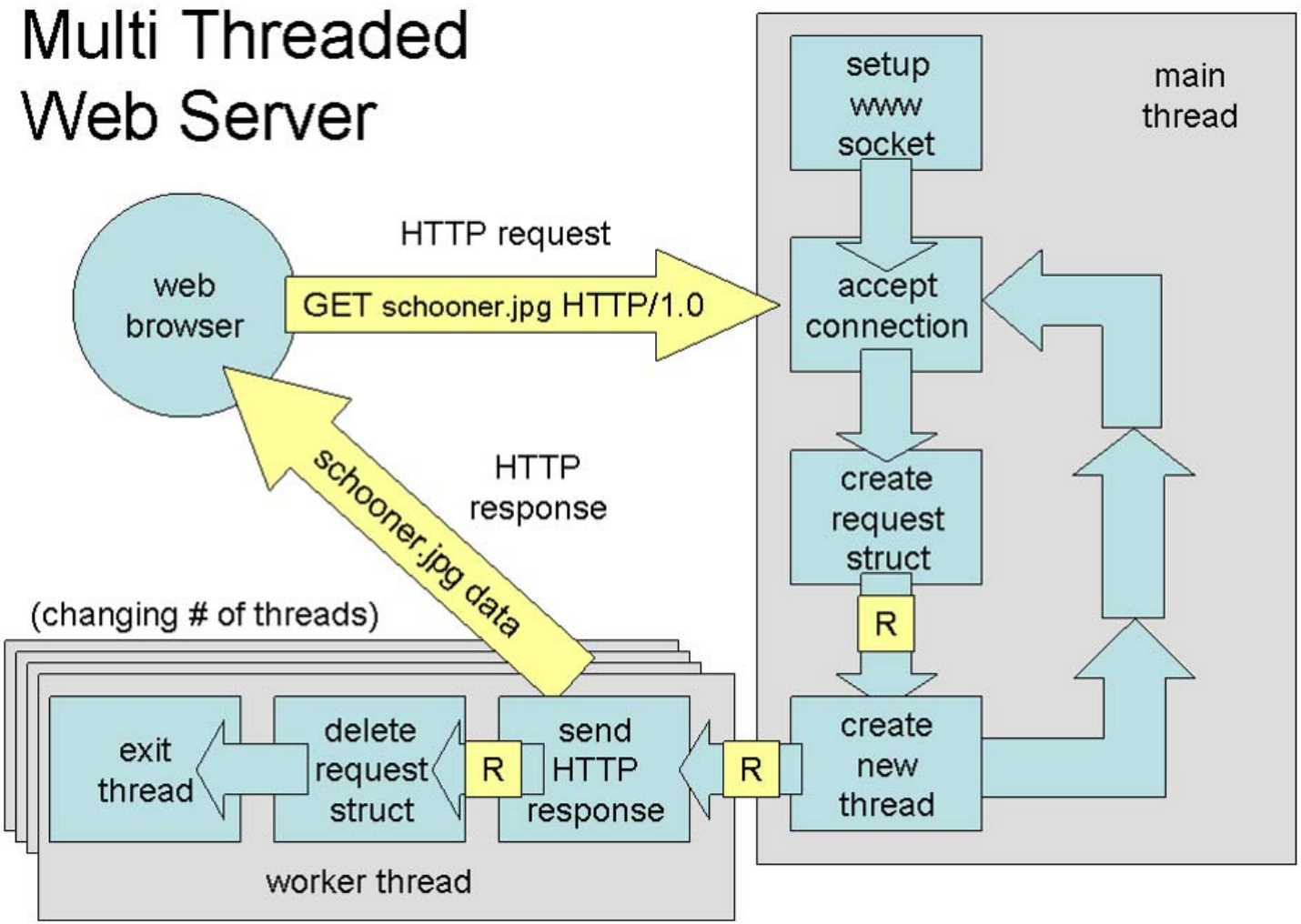
进程交互关系可以使用时序图（Sequence Diagram）来描述



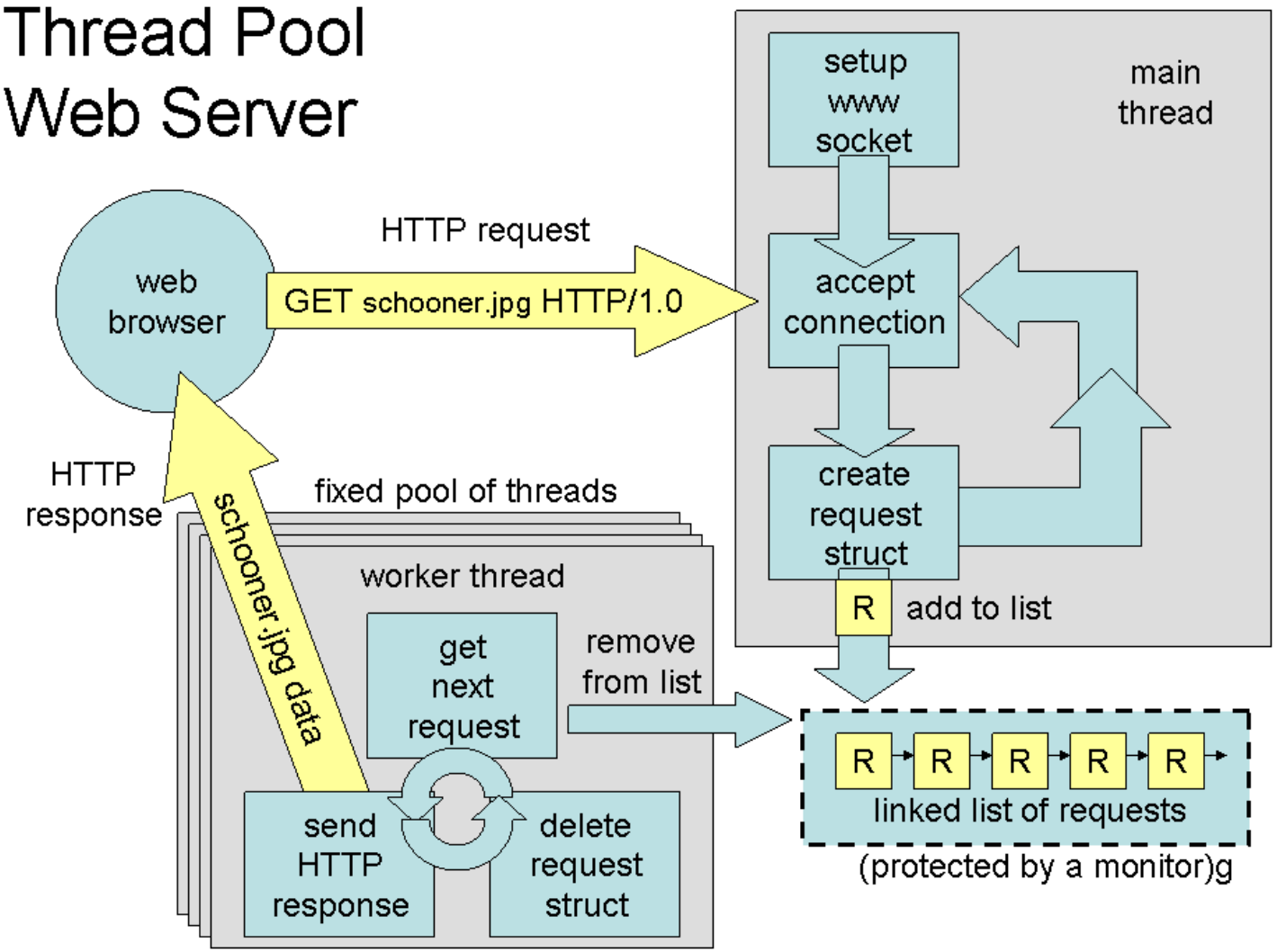
单线程的Web服务器


Single-Threaded
Web Server
setup
www
socket
main
thread
web
browser
HTTP request
GET schooner.jpg HTTP/1.0
accept
connection
create
request
struct
R
schooner.jpg data
HTTP
response
send
HTTP
response
R
delete
request
struct

多线程的Web服务器

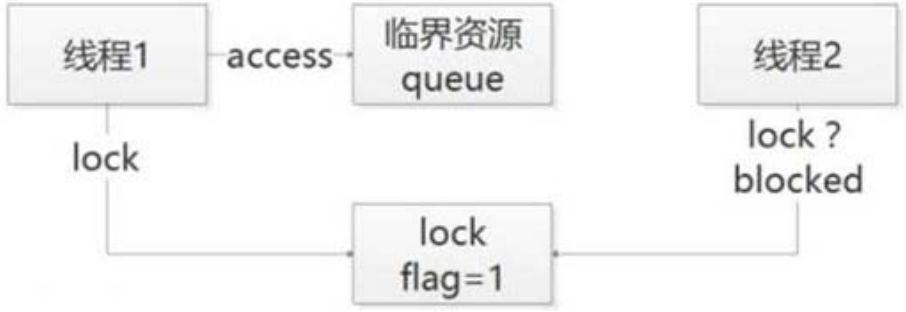

Multi Threaded
Web Server
setup
www
socket
main
thread
web
browser
HTTP request
GET schooner.jpg HTTP/1.0
accept
connection
HTTP
response
schooner.jpg data
create
request
struct
R
(changing # of threads)
exit
thread
delete
request
struct
R
send
HTTP
response
R
create
new
thread
worker thread

线程池的Web服务器

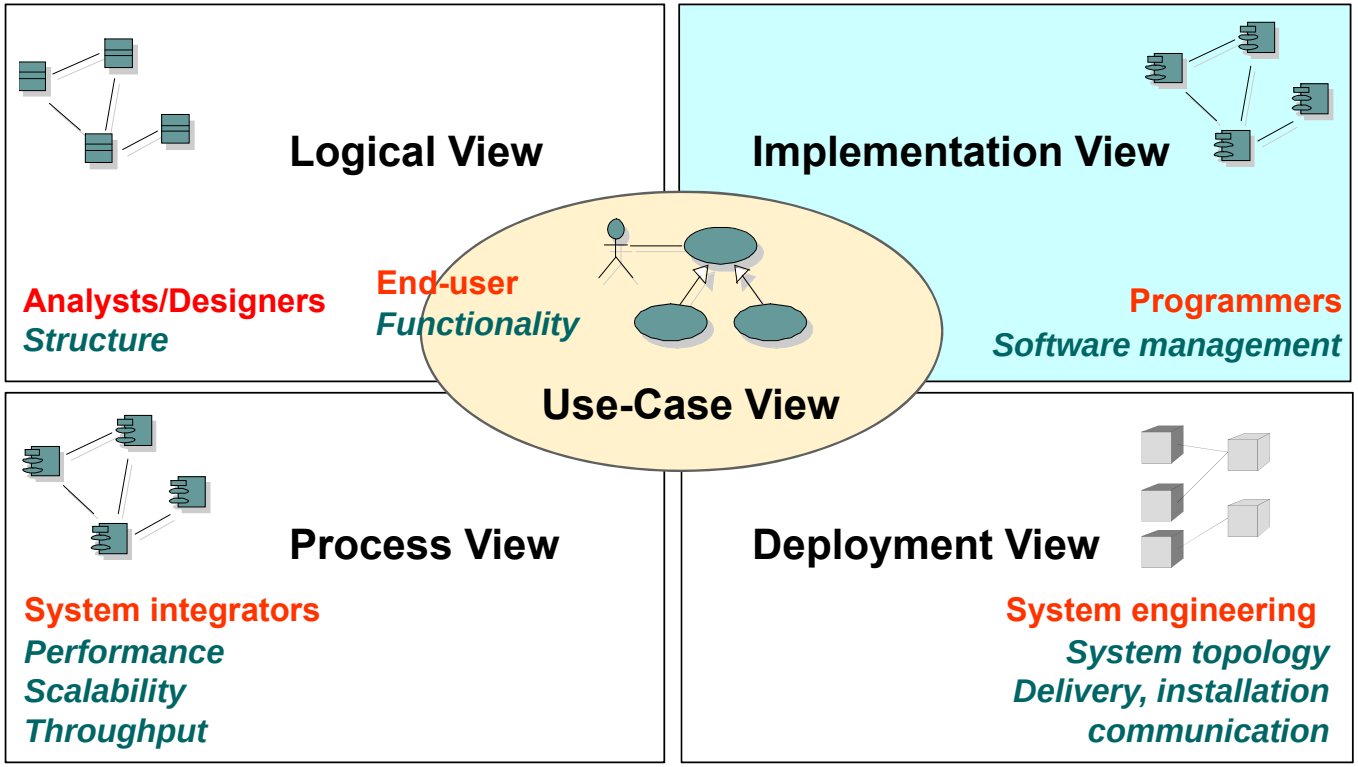

Thread Pool
Web Server
setup
www
socket
main
thread
web
browser
HTTP request
GET schooner.jpg HTTP/1.0
accept
connection
create
request
struct
R
add to list
HTTP
response
schooner.jpg data
fixed pool of threads
worker thread
get
next
request
remove
from list
send
HTTP
response
delete
request
struct
R
R
R
R
R
linked list of requests
(protected by a monitor)g

多进程/线程间的协作

- 线程同步机制
  - 互斥量
  - 读写锁
  - 条件变量
- 进程间通信机制
  - 管道
  - IPC
  - 信号



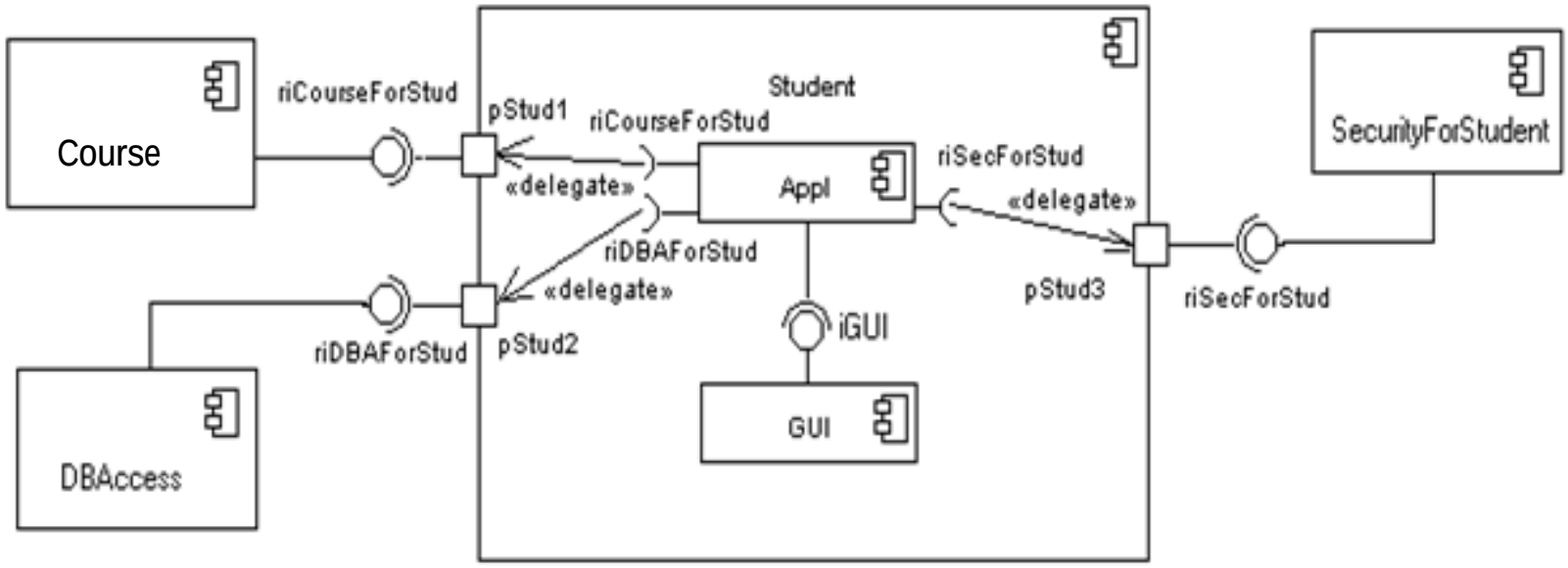
实现视图



The Implementation View is an "architecturally significant" slice of the Component Model.

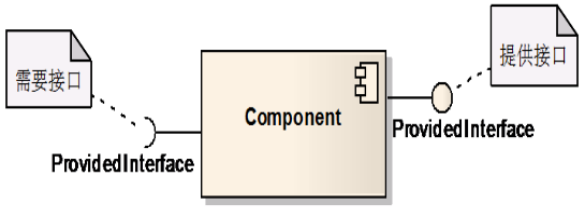
构件图Component Diagram

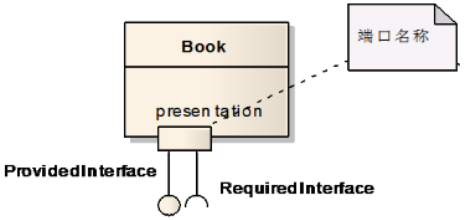
- 构件图，描述的是在软件系统中遵从并实现一组接口的物理的、可替换的软件模块。
- 构件图 = 构件(Component) + 接口(Interface) + 关系(Relationship) + 端口(Port) + 连接器(Connector)
- 针对中大型软件进行建模



构件图的主要组成

- 构件（Component）
  - 代表系统的一个物理实现模块，代表逻辑模型元素如类、接口、协同等的物理打包
- 接口（Interface）
  - 由一组操作组成，它指定了一个契约，这个契约必须由实现和使用这个接口的构件的所遵循。
  - 接口分提供接口和请求接口
- 端口（Port）
  - 描述了在构件与它的环境之间以及在构件与它的内部构件之间的一个显式的交互点





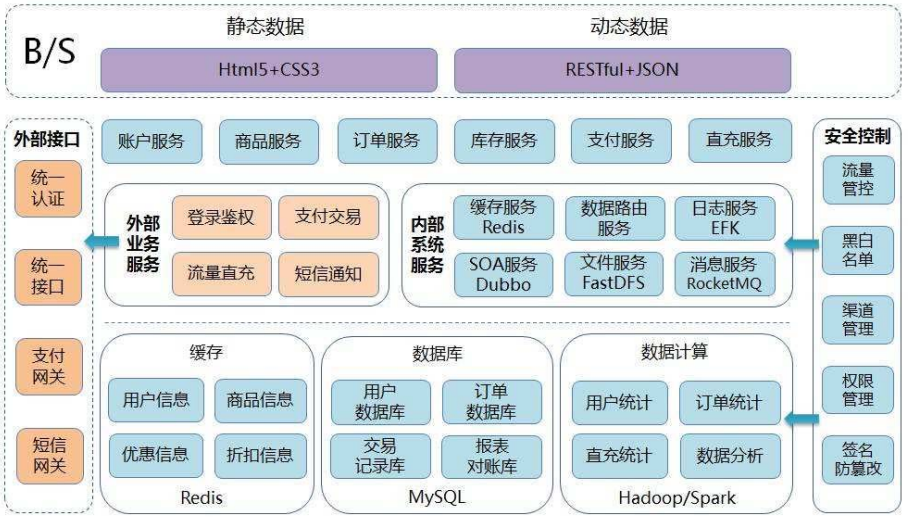
技术视图 Technical view

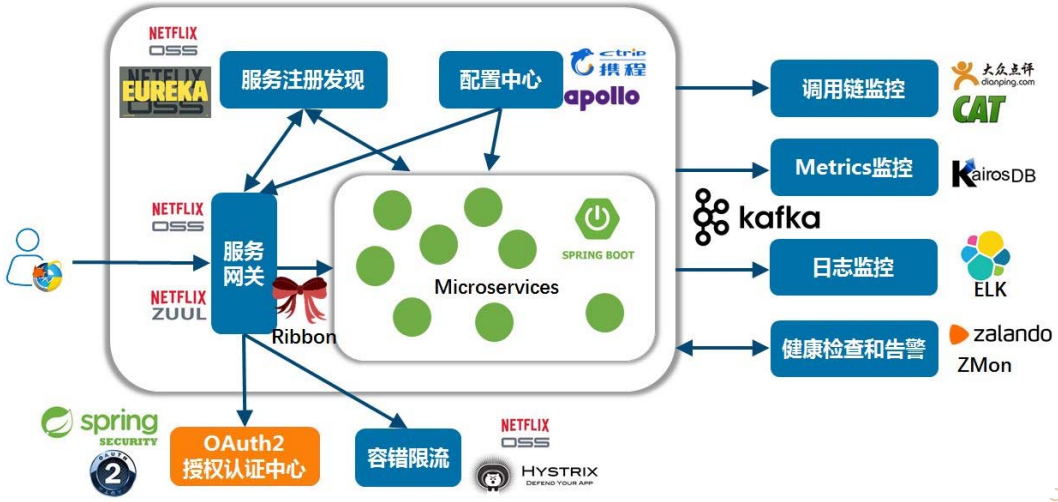
技术选型：

- 编程语言
- 操作系统
- 数据库
- 框架
- 中间件
- 库
- ....

技术视图描述方式：

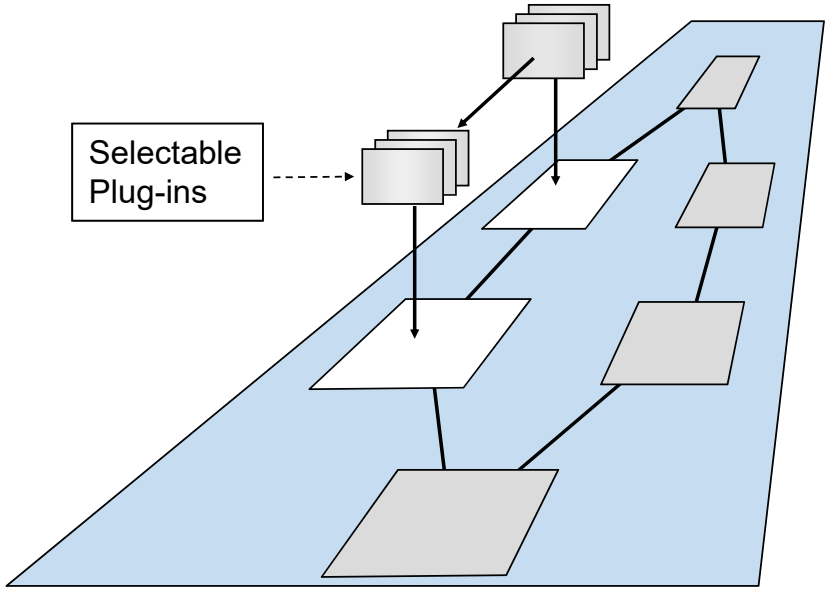
- 文本
- 技术视图
- 在逻辑视图上标注
- 在物理视图上标注
- ....





框架(Framework)

- 框架是一个代码骨架，可以使用为解决问题而设计的特定类或功能来填充这个代码骨架，使之丰满。
- 使用框架后，该框架实现的内容就不需要再进行设计与实现



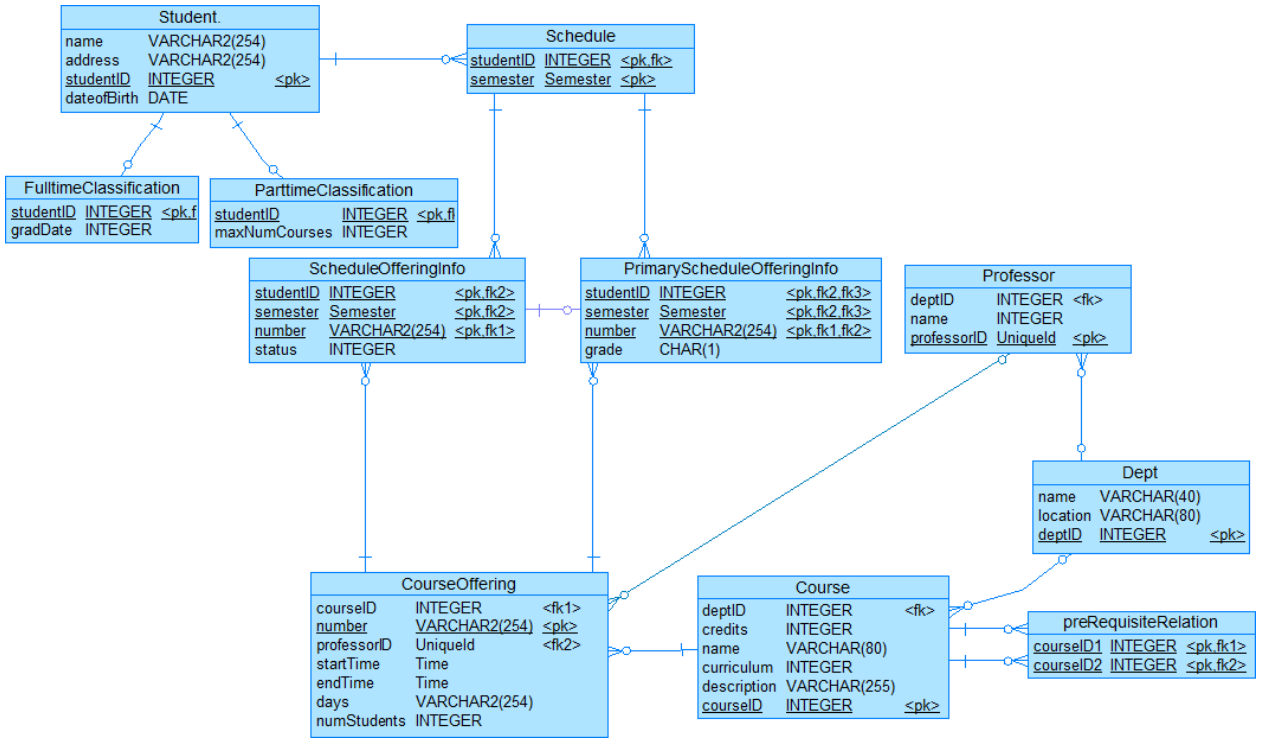
Frameworks with Plug-ins

框架举例

- 1. MVC框架
  - Spring MVC、Struts、JSF、Grails、Google Web Toolkit（GWT）
- 2. ORM 框架
  - Mybatis
  - Hibernate、Spring Data JPA
- 3. Web前端框架
  - CSS框架：Bootstrap，Fundation
  - JS框架：VUE.JS, React.js, AngularJS, Ember.js
- 4. 并行计算框架
  - Hadoop, Spark, MapReduce
- 5. 服务与微服务框架
  - Spring Cloud，Dubbo
- 6.深度学习框架
  - PyTorch, TensorFlow，Caffe
- 等等

数据视图 Data View

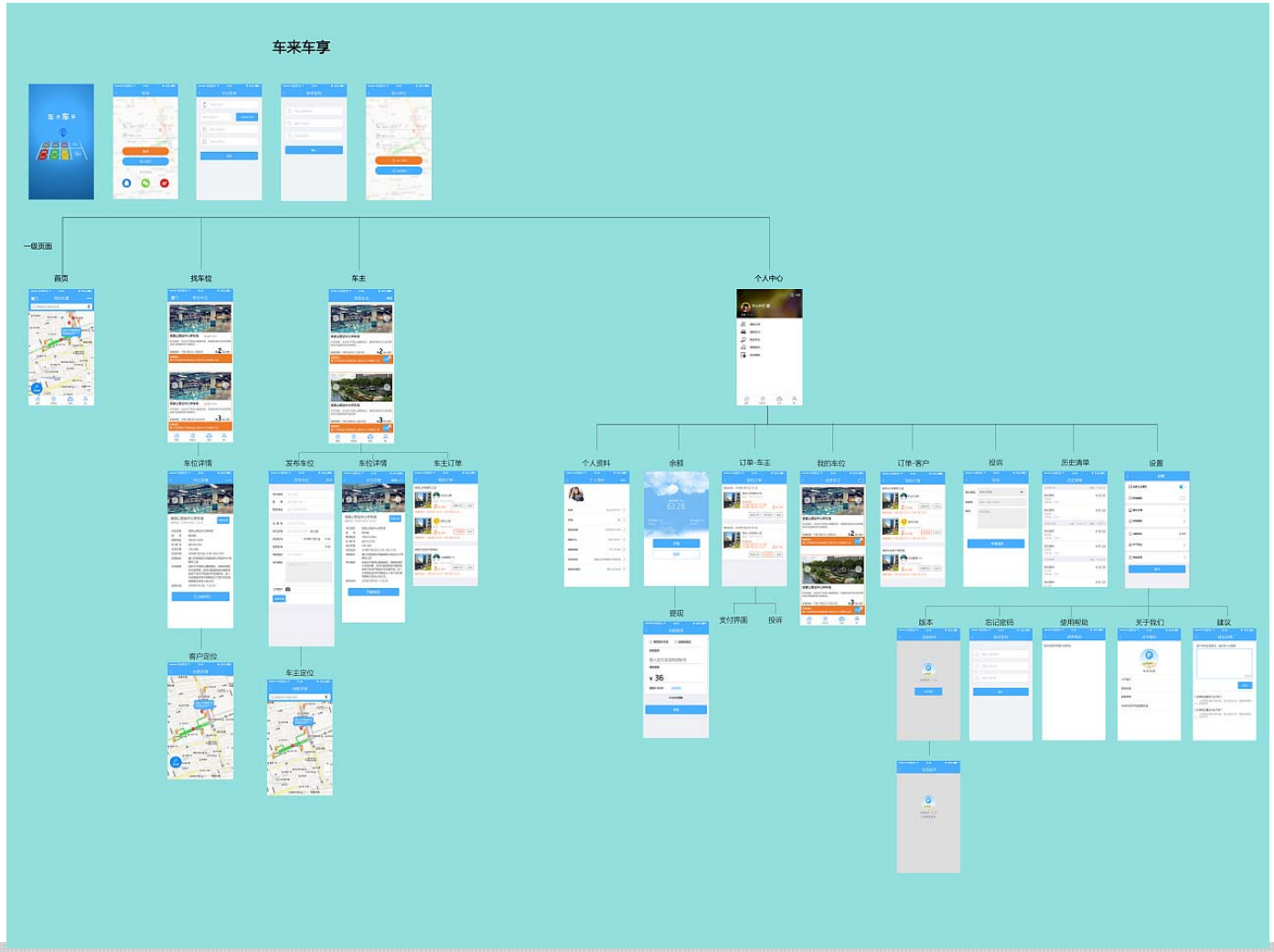
- 针对以数据为中心的软件，则需定义数据架构
- 举例：选课系统的数据视图，从概念模型生成并进行了优化。



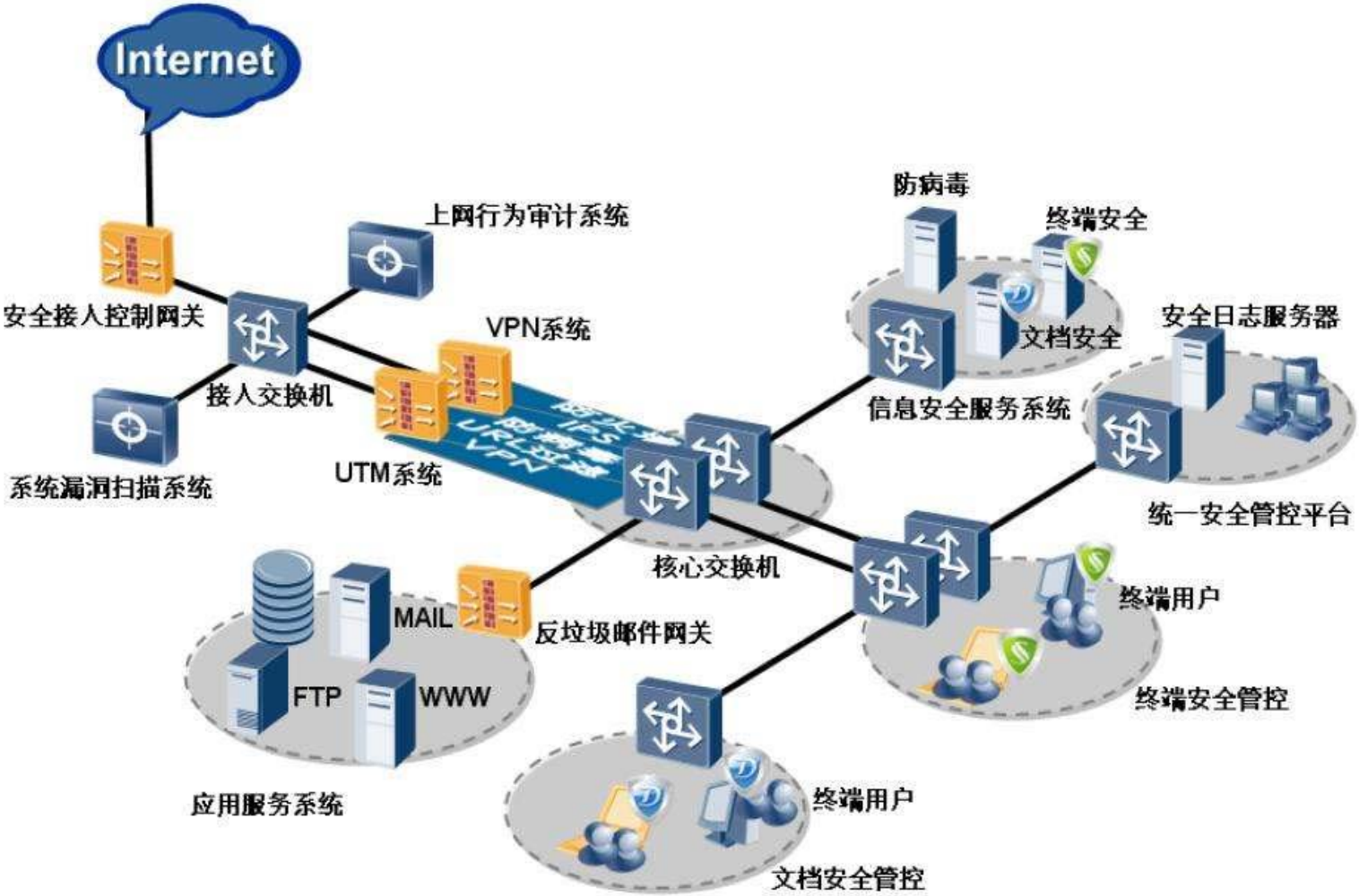
O-R Mapping

- The Problem:
  - As with relational databases, a representation mismatch exists between objects and these non-object-oriented formats.
- The Solution:
  - O-R Mapping service
    - a persistence service translate objects into records and save them in a database,
    - and translate records into objects when retrieving from a database
- O-R Mapping Middleware or Persistence Framework
  - Such as Hibernate, Spring Data JPA, MyBatis

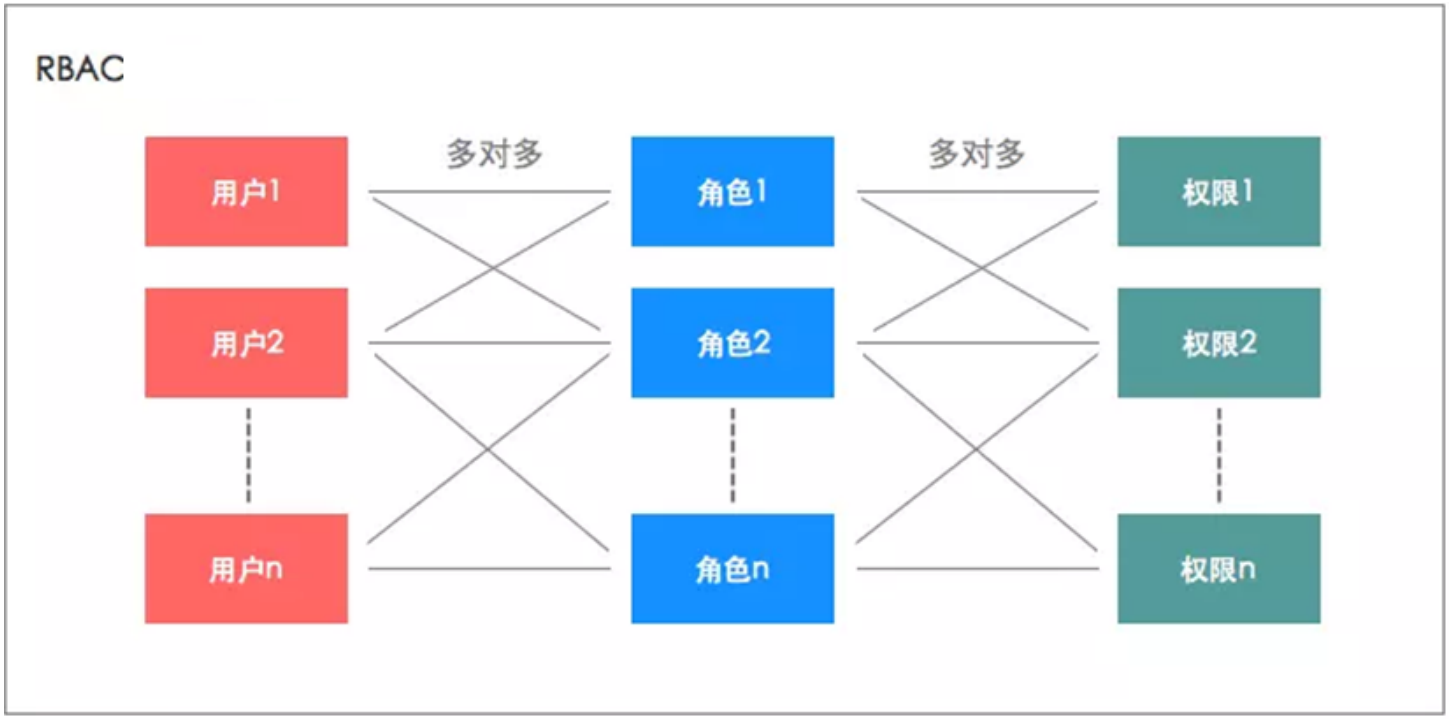
页面视图 Page view



安全视图 Security view



权限控制



- 权限包括：操作权限和数据权限

大 纲

01-软件设计的原则

02-软件设计的步骤和复用

03-软件架构设计

设计软件架构的多个视图

选择软件质量属性的设计战术

✓易用性战术
✓可靠性战术
✓性能战术
✓安全性战术
✓可测试性战术
✓可维护性战术

②选择软件质量属性的设计战术

质量因素	设计战术
易用性	1.为用户提供适当的反馈和协助 2.将用户接口与应用的其余部分分离 3.提供“取消”、“撤消”等命令 4.建立用户模型、任务模型和系统模型

- 案例讨论：用户纷纷反馈12306网站的登录的成功率低、易用性差，你有什么改进办法?

可靠性的设计战术

质量因素	设计战术
可用性 即可靠性	1.错误检测，如心跳、异常、命令/响应 2.错误恢复，如表决、冗余、备件、检查点/回滚 3.错误预防，如事务、进程监视、从服务中删除

- 案例讨论： 2011年,由北京南站开往福州站的D301次列车与杭州站开往福州南站的D3115次列车发生追尾事故，造成40人死亡，直接经济损失2亿元。原因是信号设备存在严重缺陷，遭雷击发生故障后，导致本应显示为红灯的信号机错误显示为绿灯.
- 如何提高系统的可靠性?

性能的设计战术

质量因素	设计战术
性能	1.资源需求，如提高计算效率、减少计算开销、控制采样频率、限制队列大小 2.资源管理，如引入并发、增加可用资源、维持数据或计算的多个副本 3.资源仲裁，如先进先出、优先级调度

- 案例讨论：淘宝网站在11.11面临大量并发用户和订单，
  如何保证系统的性能（响应时间）？
  如何设计按需伸缩的软件架构?

安全性的设计战术

质量因素	设计战术
安全性 Security	1.抵抗和检测攻击，如用户身份验证、用户授权、限制暴露的信息、防火墙 2.从攻击中恢复，如维持审计追踪、采用可用性中恢复战术

- 案例讨论：如何保证网上银行系统的安全性？可以应用哪些技术与措施？

可测试性与可维护性的设计战术

质量因素	设计战术
可测试性	1.测试的输入/输出，如记录/回放、将接口与实现分离、特化访问路线/接口 2.内部监视，如内置监视器
可维护性	1.局部化修改，如泛化模块、预期期望的变更、限制可能的选择 2.防止连锁反应，如信息隐藏、维持现有接口、限制通信路径、仲裁者的使用 3.推迟绑定时间

- 案例讨论：如何保证交我办系统可以方便增加各种服务？

多目标权衡

- 不同质量属性之间往往存在冲突，例如：
  - 性能 VS. 安全性
  - 可靠性 VS. 灵活性
  - 易维护性 VS. 高效性
  - ……
- 排出优先级，进行多目标权衡