



第一章

分布式系统概论

第一章、分布式系统概论

01

1.1 分布式系统概念

02

1.2 分布式系统关键技术

03

1.3 分布式系统软件体系结构

04

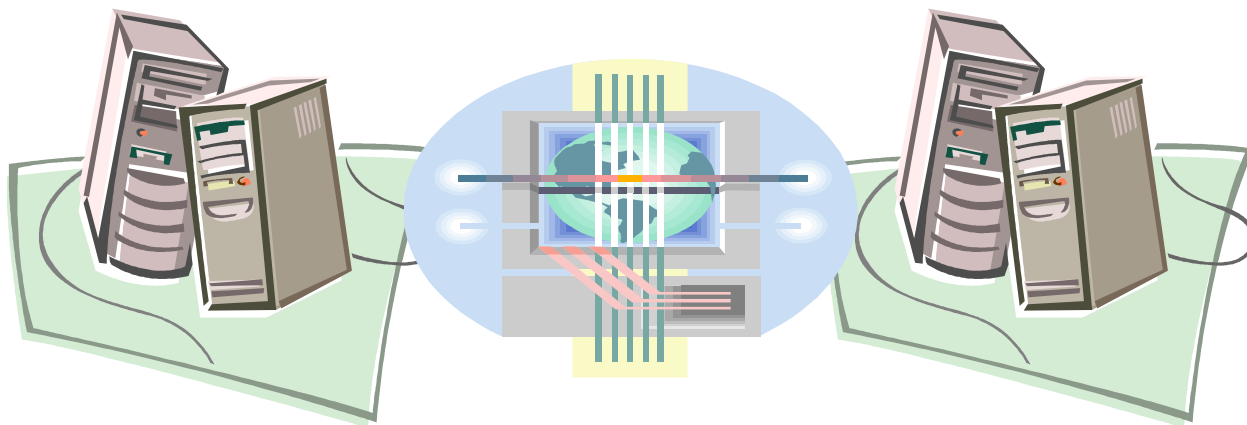
1.4 分布式系统系统体系结构



1.1 分布式系统概念

“一个分布式系统是若干个独立的计算机的集合，但是对该系统的用户来说，感觉该系统就像一台计算机一样。”

- 硬件方面：每台计算机都是独立、自主的计算机
- 软件方面：用户感觉在独占系统



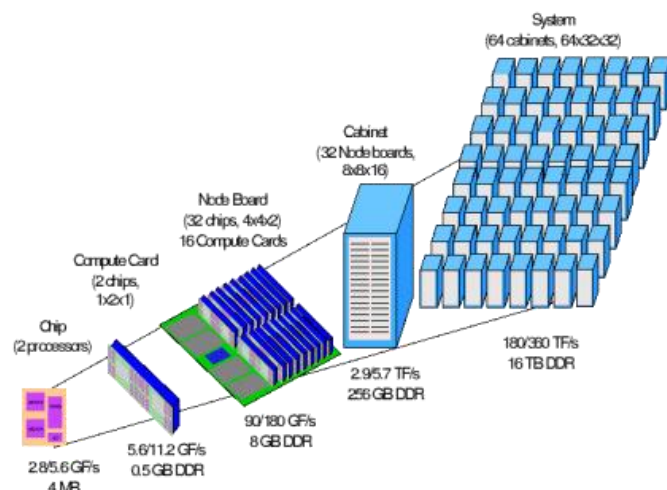
从集中式系统到分布式系统



- 高性能微型计算机(PC)的普及
 - 台式机、笔记本、平板电脑、智能手机...
- 高速计算机网络 (LAN、WAN) 的普及
 - 有线网络、WIFI无线网络
 - 4G、5G移动网络

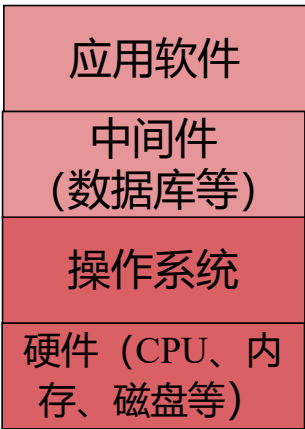
从集中式系统到分布式系统

目前中国最先进、最快的超级计算机是「灵晟」(LineShine)，部署于国家超级计算深圳中心，在 2026 年 6 月 ISC 大会第 67 期 TOP500 榜单登顶全球第一

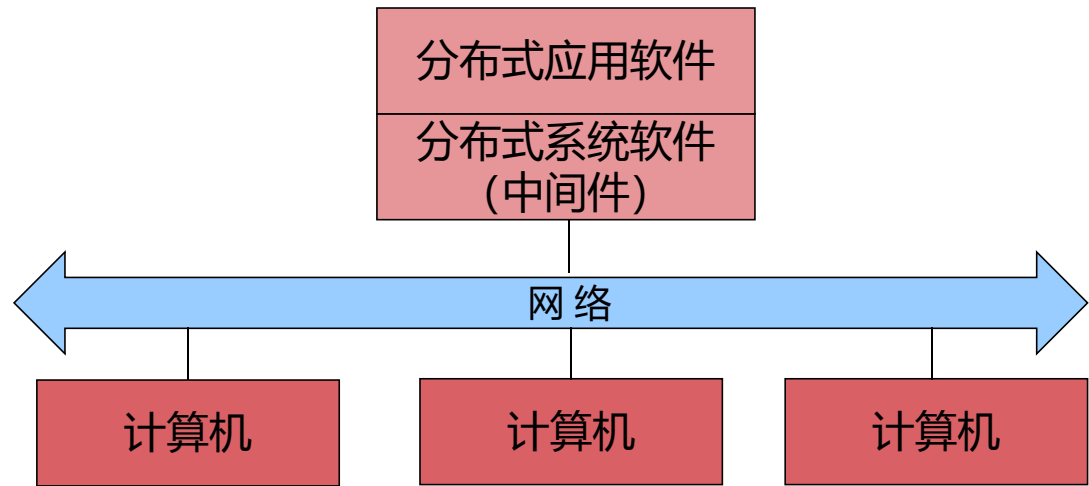


- **算力**：HPL 持续 2.198 EFlops（每秒 2198 亿亿次）；理论峰值 2.736 EFlops，利用率约 80%。
- **处理器**：自研「灵昆 LingKun」平台，LX2 处理器，单片 304 核、1.55 GHz；总核数 13,789,440（约 1379 万核）。
- **架构**：全 CPU、无 GPU/加速器；内嵌 AI 矩阵加速单元，首创 Online Acceleration 全 CPU 架构。
- **互联**：自研「灵柒 LingQi」高速互联，百万端口级交换 fabric。
- **软件**：麒麟操作系统（Kylin OS）+ 自研系统栈。
- **规模**：二期约 92 个计算机柜、~4.7 万颗 CPU；36 个网络机柜；67 个液冷存储机柜、428 存储节点、总带宽 10 TB/s。
- **能耗**：功耗约 42.2 MW，能效 52.07 GFlops/W。

集中式系统与分布式系统



(a) 集中式系统



(b) 分布式系统



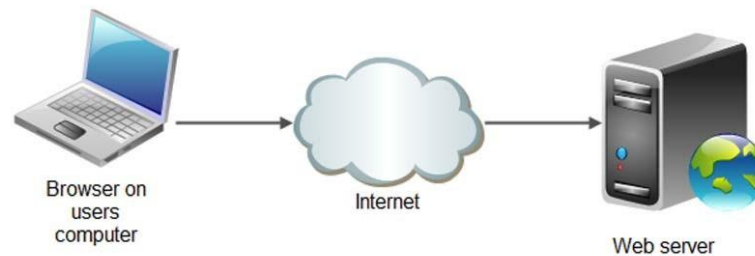
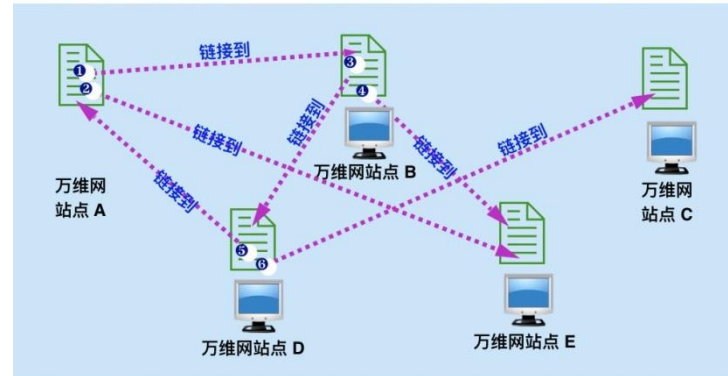
1.1.1 分布式系统的应用

- 举例：
 - WWW系统
 - 银行卡系统
 - 办公自动化系统
 - 快递物流系统
 - 企业工作流系统
 - 物联网系统
 - 区块链系统



分布式应用举例1- WWW系统

- 建立在因特网（Internet）之上的一个全球性网络，称为万维网
 - 基于超文本协议HTTP的分布式可视化信息系统
 - Web服务器和Web浏览器构成
 - 各种基于B/S模式（浏览器/服务器）的信息系统

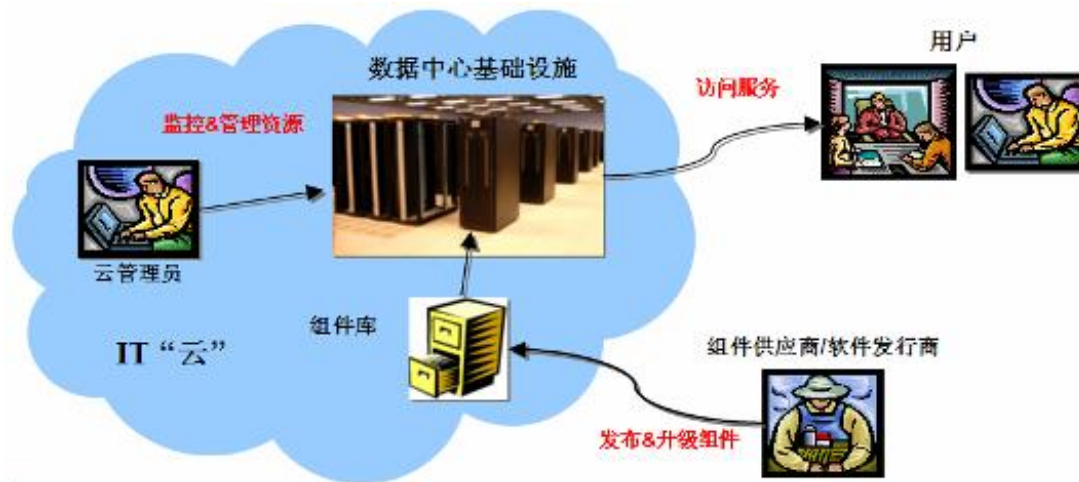


分布式应用举例2 - 云计算

- 在互联网上提供的动态可扩展的虚拟化资源的服务。
 - 成千上万台的计算机，组成了“云”。
 - 用户通过网络与“云”连接后，就可使用“云”中的计算能力和资源
 - 软件服务SaaS、平台服务PaaS、架构服务IaaS

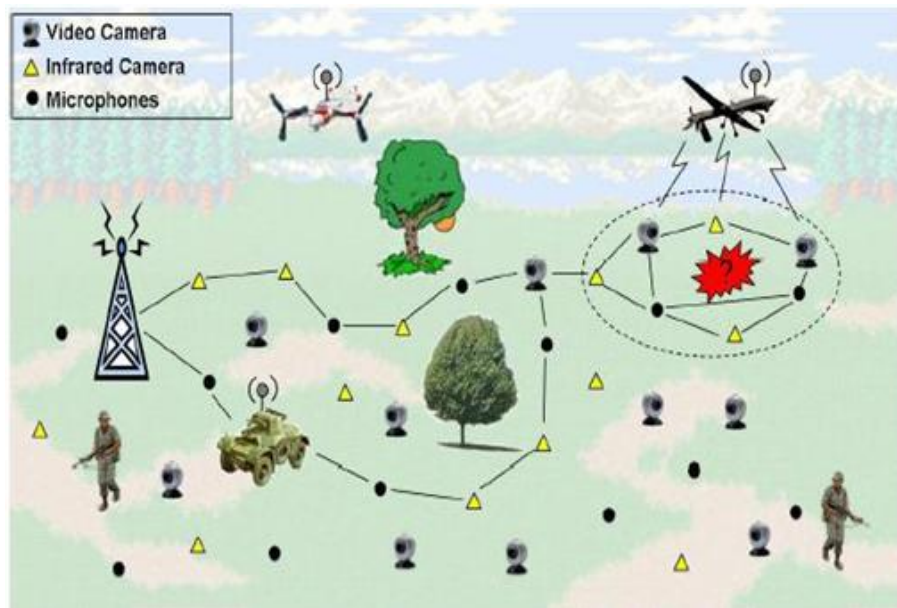
电厂→电力
云平台→算力

AI时代：算力=token



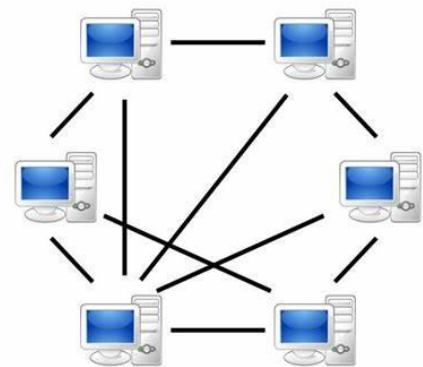
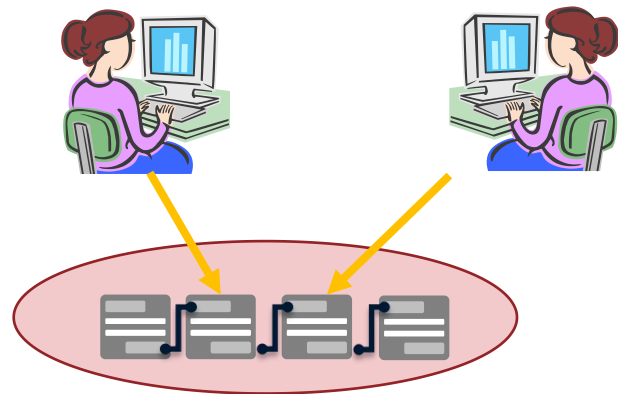
分布式系统举例3 - 物联网

- 强调计算与环境融为一体。随时随地可以进行计算
 - 例1：无线传感器网络，传感器节点组成的自组织（Ad hoc）网络
 - 例2：无人机群，感知、通信、计算



分布式系统举例4 - 区块链

- 案例3：区块链系统
- 去中心化的分布式记账系统
 - 分布式账本：多个节点共同完成，每一个节点记录完整账目（数据库）
 - 非对称加密：只有在数据拥有者授权的情况下才能访问到（安全）
 - 共识机制：记账节点之间怎么达成共识，去认定记录的有效性（分布式）
 - 智能合约：自动化的执行一些预先定义好的规则和条款



知乎 @Kaichao

分布式系统的类型

根据分布式系统的应用类型，可划分出3种分布式系统。

1. **科学计算**：高性能科学计算
2. **信息服务**：信息管理和事务处理
3. **数据密集型计算**：大数据处理

分布式计算系统

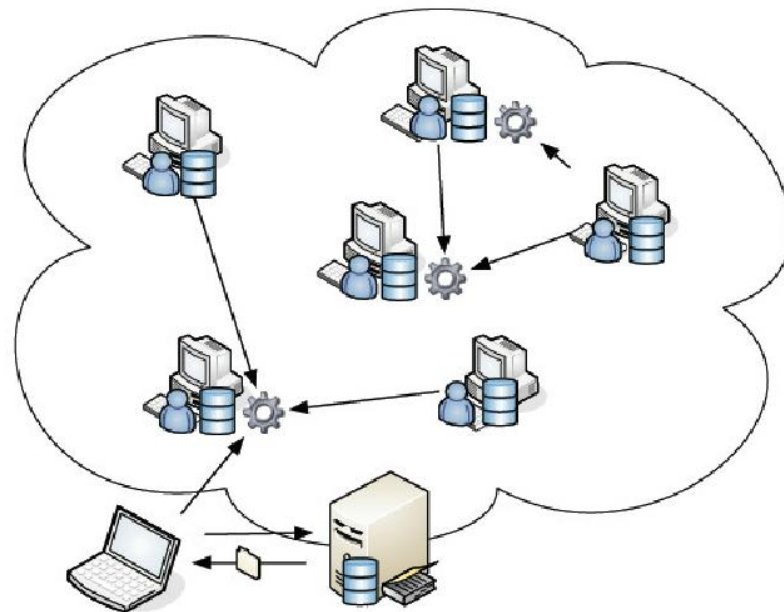
■ 案例1：计算网格系统

■ 虚拟组织

- 逻辑上统一的一组人或机构
- 资源的提供者
- 资源的使用者

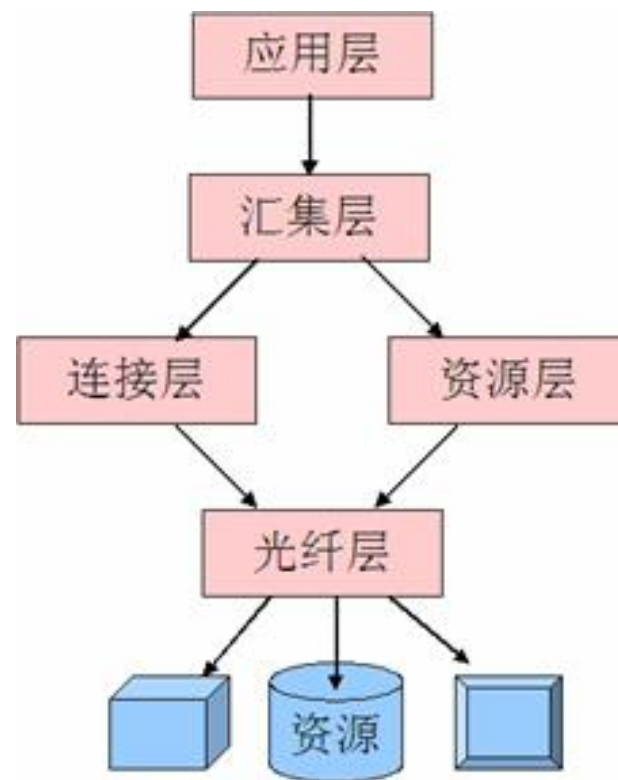
■ 共享资源

- 硬件：服务器、存储
- 软件：操作系统、数据库等



分布式计算系统

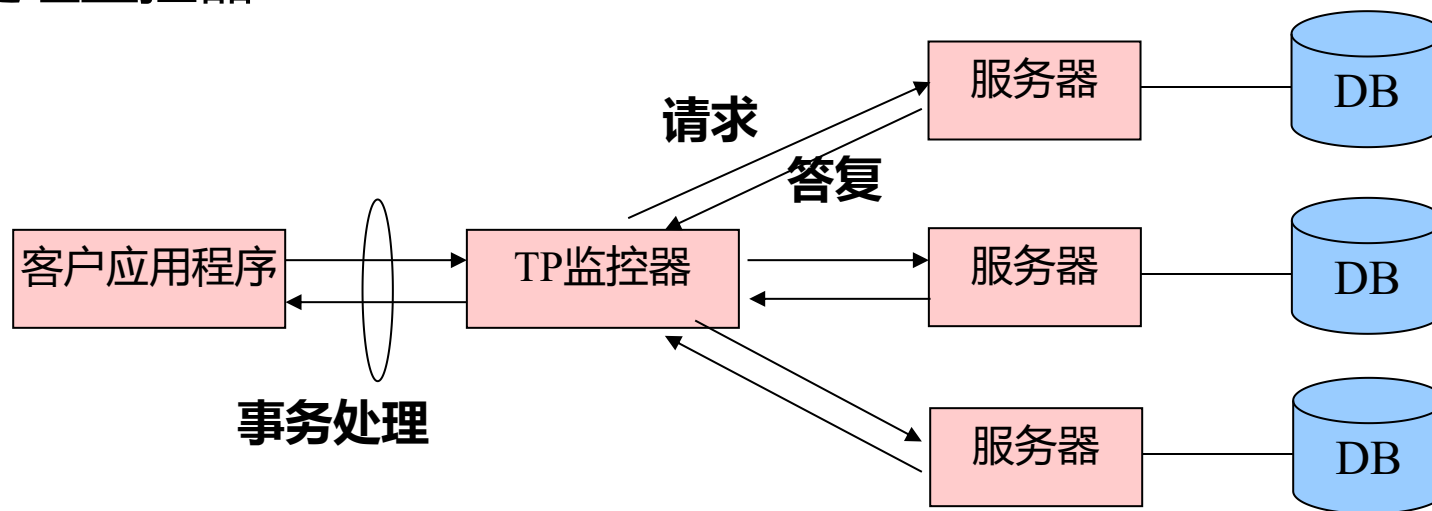
- 案例1：网格系统
 - 光纤层：资源的接口
 - 连接层：通信协议
 - 资源层：管理单个资源
 - 汇集层：对多个资源的访问
 - 应用层：虚拟组织的应用程序
 - 面向服务的体系结构（SOA）
 - 开放式网格服务体系结构（OGSA）



分布式信息系统

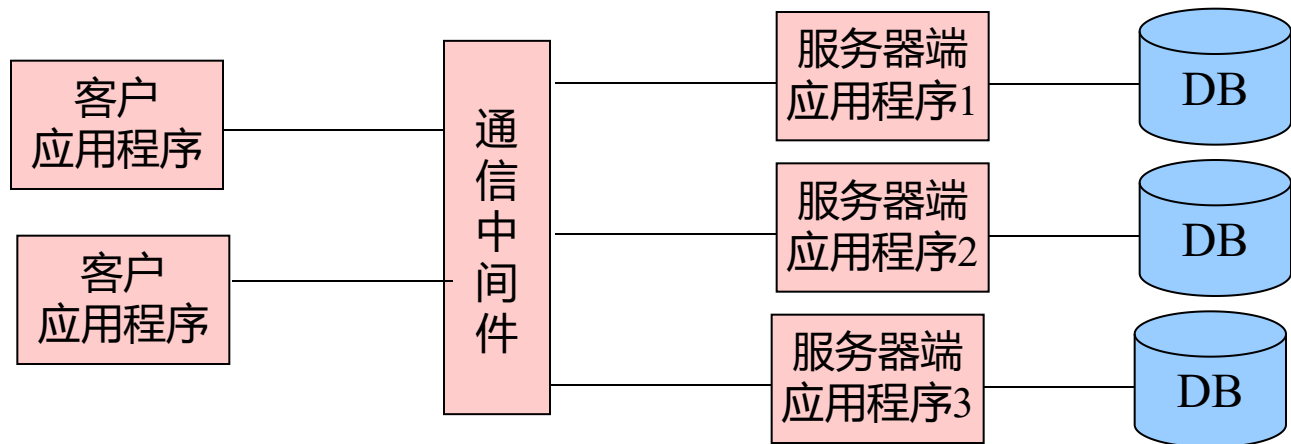
■ 案例2：事务处理系统

- 事务的ACID性质：原子性、一致性、隔离性和持久性
- 分布式或嵌套式事务：一个全局事务加上多个子事务组成
- 事务处理监控器



分布式信息系统

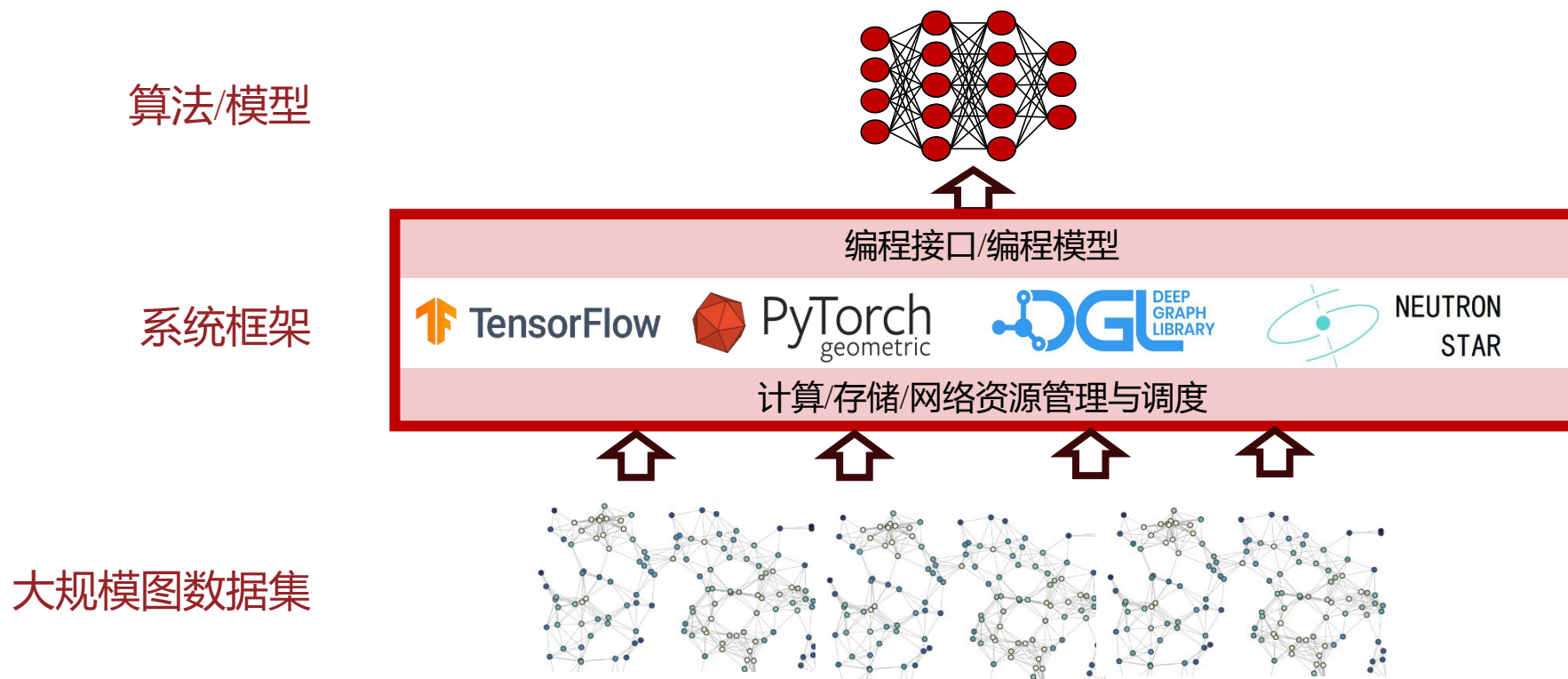
- 案例3：企业应用集成(EAI)
 - 将与企业中的应用程序进行集成
 - 提供应用程序之间的互操作。如CIMS（CAD, CAPP, CAM, PDM）
 - 通信中间件：RPC、RMI、MOM等



数据密集计算型系统

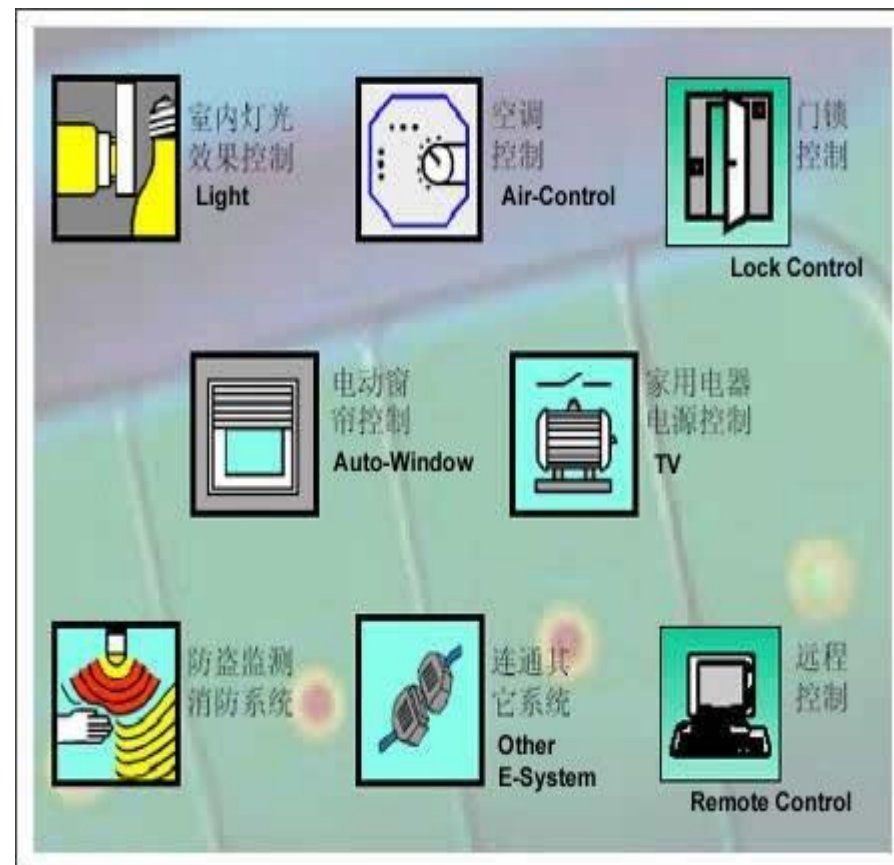
■ 案例4： 大模型训练系统

- 建立图神经网络（GNN），得到图的嵌入表示。为后续的分类、链接预测、推荐提供支持



分布式普适系统

- 案例5：智能家庭系统
 - 家庭网络，连接所有家用电器等设备
 - 支持通用即插即（UPnP）标准
 - 支持个人数据空间（data space）管理



1.1.2 分布式系统的设计目标

1. 透明性 (Transparency) : 方便用户使用
2. 开放性 (Openness) : 便于协同工作和交互
3. 可伸缩性 (Scalability) : 可持续扩展性能
4. 可靠性(Reliability)和可信性 (Trustiness) : 保持正常工作的能力
5. 高性价比 (Performance/Cost ratio) : 低代价, 高处理能力

透明性 (Transparency)

- **访问透明性**: 用户不必知道数据表示和访问方法。如数据编码
- **位置透明性**: 用户不必知道资源位于何处
- **迁移透明性**: 资源可以不改名, 而随意移动
- **重定位透明性**: 资源可在使用中移动位置
- **复制透明性**: 用户不必知道有多少个备份存在
- **并发透明性**: 多个用户可以自动地共享资源
- **故障透明性**: 用户不必了解故障恢复处理过程

开放性(openness)



- 开放式系统
 - 符合标准协议（语法和语义规则）
 - 例：IDL(接口定义语言)
- 特点：
 - 互操作性：不同系统可共同工作
 - 可移植性：应用程序可在不同系统上运行
 - 灵活性：可配置不同开发者的构件
 - 可扩展性：可增减构件

可伸缩性 (scalability)

- 指标：
 - 规模可伸缩性
 - 地理位置可伸缩性 (跨地理区域)
 - 管理可伸缩性 (跨管理区域)
- 限制规模可伸缩性的问题：
 - 集中式服务 (如高度机密的信息)
 - 集中式数据 (通信线路影响)
 - 集中式算法 (如, 最佳路由选择)



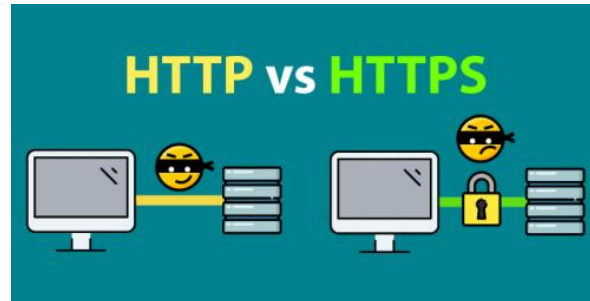
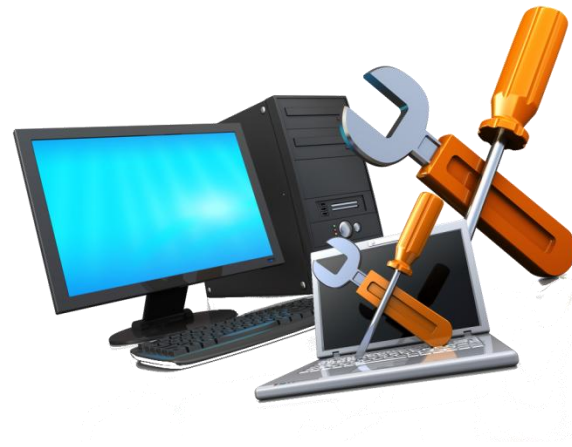
可伸缩性 (scalability)

- 限制地理可伸缩性的问题
 - 跨地理区域
 - 通信延迟
 - 不可靠通信
 - 集中式方案
- 限制管理可伸缩性的问题
 - 跨管理区域
 - 策略冲突 (收费、管理、安全性)



可靠性与安全性

- 可靠性(Reliability)和可用性(availability)
 - 保证系统正确运行
 - 计算机节点间“或”相关性、“与”相关性
- 容错性(Fault-tolerance)
 - 保证系统在发生故障后，仍能正常运行
 - 硬/软件冗余
 - 恢复和接管
- 安全性(Security)
 - 保证系统不受攻击
 - 身份认证、入侵监测



高性价比 (Performance/Cost)

高的性能/价格比

Grosch's Law: Power $\uparrow$ Price²

$$100^2 > 50^2 + 50^2$$



NOPC vs. 大型机 (mainframe)



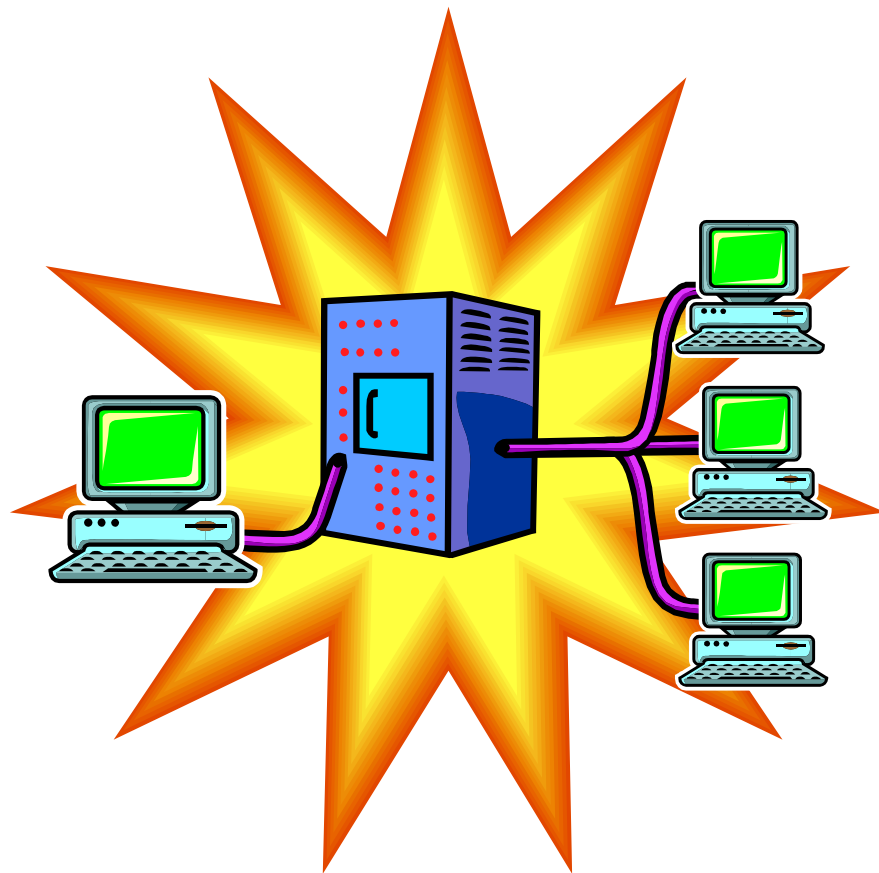
高性价比 (Performance/Cost)

- 运算能力
 - FLOPS每秒浮点运算次数
 - Linpack测试
- 运行成本
 - 造价，价格。普通电脑6000元左右。超级计算机造价一般超过1亿美元
 - 功耗。天河二号能效比每瓦特19亿次
 - 空调设备功耗
 - 维护费用。人工、备件



高性价比 (Performance/Cost)

- 吞吐率(Throughput)
 - 每秒钟处理事务的能力
 - TPC-C基准测试
- 实时性(Real-time)
 - 截止期(deadline)
 - 响应速度
- 并行性(Parallelism)
 - 数据并行性、功能并行性
 - 分区并行性、流水平行性
 - 细粒度并行性、粗粒度并行性



1.1.3 面临的挑战

1、不可靠全局时钟

- 时钟同步



2、不可靠的网络

- QoS(带宽、速度)
- 消息丢失、次序错乱

3、可靠性和安全性

- 黑客、泄密、盗用、破坏



第一章、分布式系统概论

01

1.1 分布式系统概念

02

1.2 分布式系统关键技术

03

1.3 分布式系统软件体系结构

04

1.4 分布式系统系统体系结构



1.2 分布式系统的关键技术

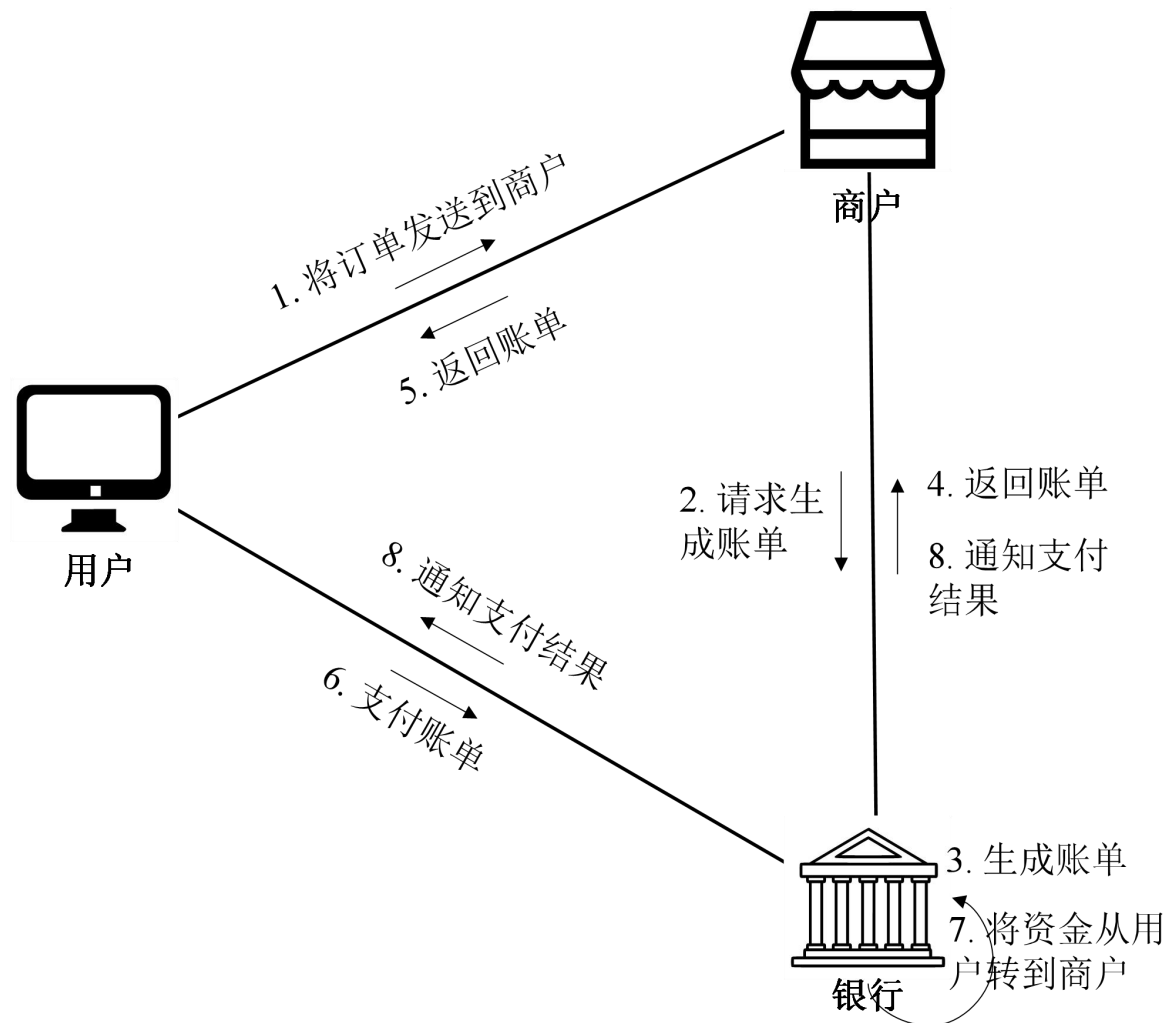
■ 关键技术

1. **任务调度**：作业管理、进程管理和线程管理
2. **通信管理**：在节点之间可靠地传送消息
3. **命名管理**：识别和定位实体（进程、文件、设备等）
4. **数据划分和定位**：分配和定位节点上的数据
5. **复制和一致性**：管理数据在节点上多个副本
6. **故障恢复与共识**：容错技术、共识协议
7. **事务处理**：分布式并发控制、分布式提交协议等

1.2 分布式系统的关键技术

举例：在线电商支付系统

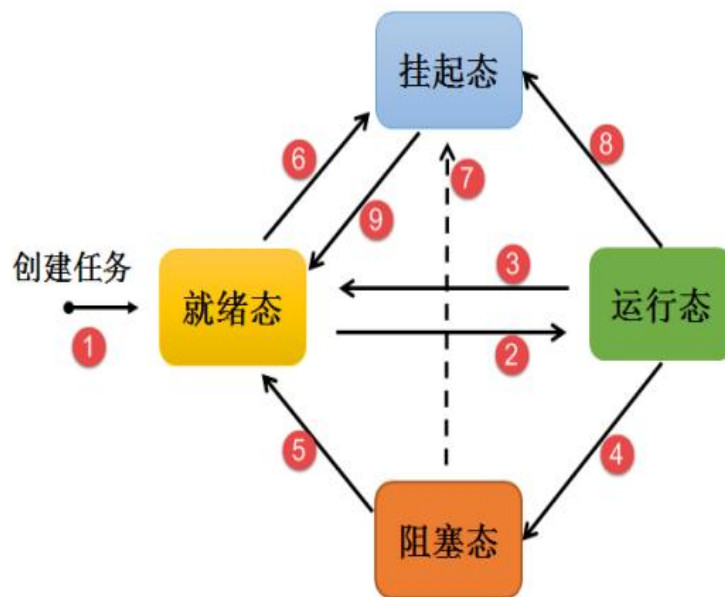
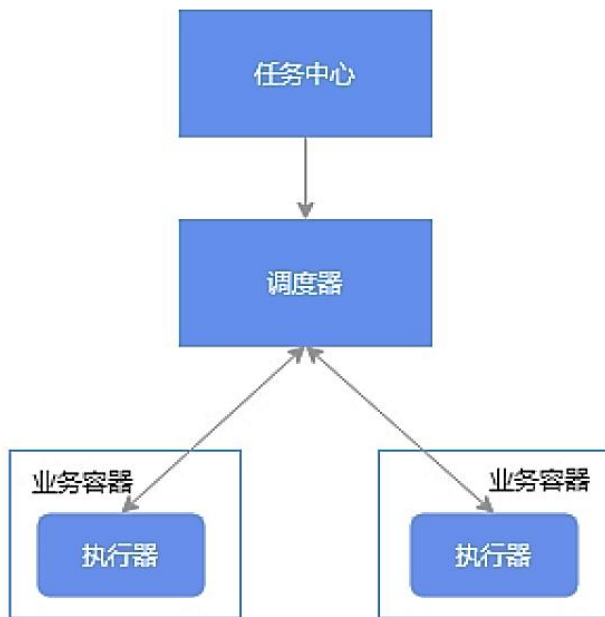
- 顾客节点、商户节点和银行节点



1.2 分布式系统的关键技术

1、任务调度

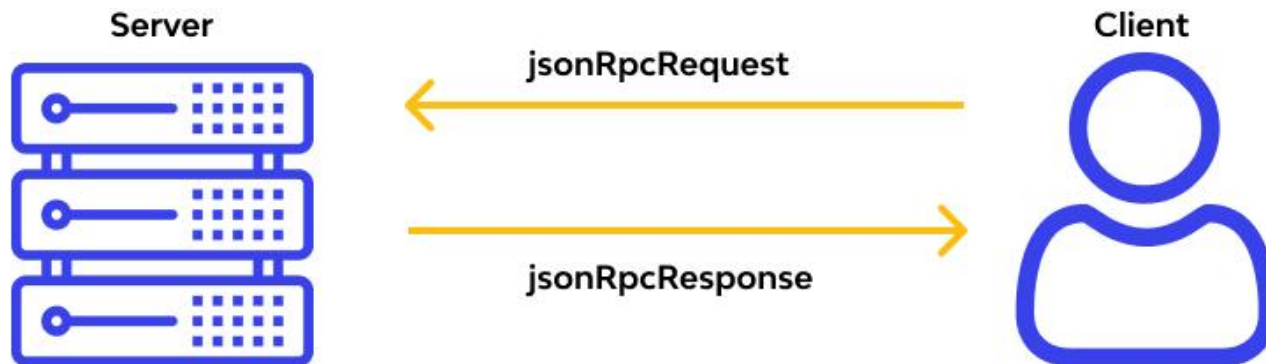
- **作业管理**：协调和管理跨多台服务器部署的商户节点或银行节点，按需分配计算资源，可随负载弹性伸缩；
- **进程管理和线程管理**：由底层操作系统完成，创建、调度和终止进程或线程



1.2 分布式系统的关键技术

2、通信管理

- **远程过程调用 (RPC)**：顾客节点将商品订单信息作为参数，向指定商户节点的地址和端口发送RPC请求
- **基于消息的通信**：商户节点可采用基于消息的通信方式，向用户节点发送评价、推荐等提醒信息



1.2 分布式系统的关键技术

3、命名管理

- 统一资源定位符 (URL) <https://www.taobao.com/>
- 域名系统 (DNS) 解析对应IP地址

4、数据划分和定位

- 账户数据通常被划分成多组（分片），分别保存在多台服务器上
- 采用基于哈希的数据划分、数据路由等策略

1.2 分布式系统的关键技术

5、复制和一致性

- 银行节点存储同一份账目数据的多个副本到多个备份节点上（如两地三中心容灾策略），提高可用性
- 商户节点使用内容分发网络（CDN）将商品信息跨地域复制，提高顾客的查询效率
- 保证各个副本的一致性和执行效率

6、故障恢复与共识

- **容错技术**：发生故障后，使用备份节点代替故障节点
- **共识协议**：确保了多个账户副本间所保存信息的一致性

1.2 分布式系统的关键技术

7、事务处理

- 保证分布式事务的ACID性质
- **分布式提交协议**：在银行节点上，扣除用户金额并增加商户金额，需要保证原子性，即要么都执行成功，要么都失败
- **低隔离级别的并发控制策略**：用于商户节点中的非关键数据（如查询商品列表），提升系统性能

第一章、分布式系统概论

01

1.1 分布式系统概念

02

1.2 分布式系统关键技术

03

1.3 分布式系统软件体系结构

04

1.4 分布式系统系统体系结构



1.3 软件体系结构

- **结构**（构成）：由多个部份组成并按特定方式组织在一起的**事物**
- **体系结构**（架构）：有关设计与构造的方式和方法，一种规范
- **框架**（骨架、基本结构、**平台**）：支撑或围住其它物体的结构，尤指用作建筑物之基础的支撑框架。



1.3 软件体系结构

- 分布式系统的逻辑组织是由软件构件构成
 - 逻辑体系结构用软件体系结构表示
- 软件体系结构 (Software Architecture)
 - 描述软件的构件, 以及构件之间的相互关系
- 基本概念
 - 构件、组件(component): 具有良好接口的程序模块单元
 - 连接器 (connector: 实现构件间通信、协作的机制。如支持远程过程调用、消息传输的工具设施

[1] Bass L., Clements P., and Kazman R. Software Architecture in Practice. Addison-Wesley, Reading, MA., 2nd edition, 2003

1.3 软件体系结构

- 软件体系结构的架构模式

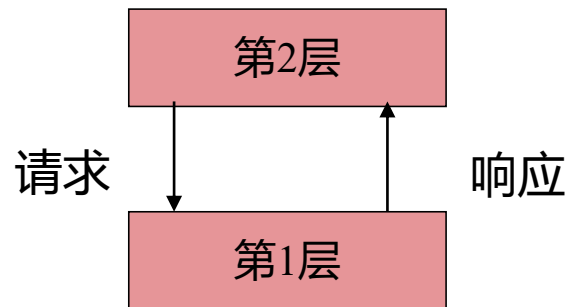
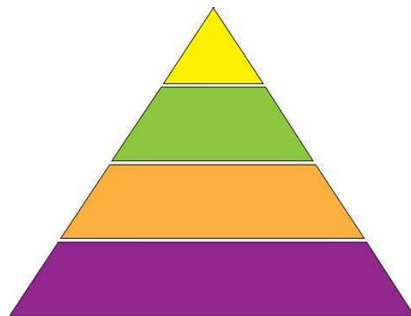
- 软件体系结构的表现形式、个性特征
- 常用的有5种模式（分层模式、对象模式、数据中心模式、事件驱动模式、资源型模式）

- 软件体系结构的要素

- 构件：封装性、可替换性（内部可修改但接口不变）
- 连接方式：连接器，将构件相互连接的方式
- 数据：在构件之间交换的数据
- 配置：如何将构件和连接器配置成一个系统

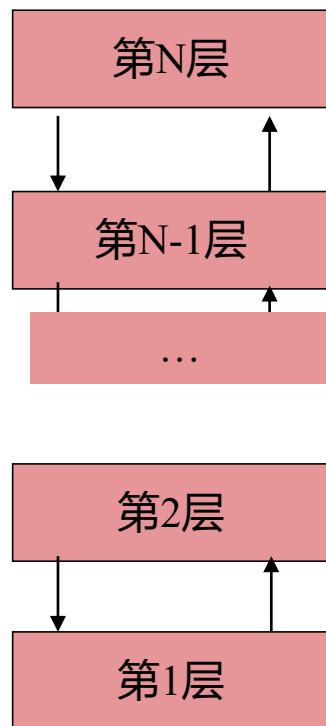
1.3.1 分层模式

- 层次 (layer, tier, level)
 - 按上下排列的水平划分。如地层、阶层、等级
- 系统由自上而下的不同层次的构件组成
 - 只有相邻的层次可以进行通信
 - 请求消息自上而下 (**向下调用**)，响应消息自下而上
- 例如：浏览器/Web服务器系统

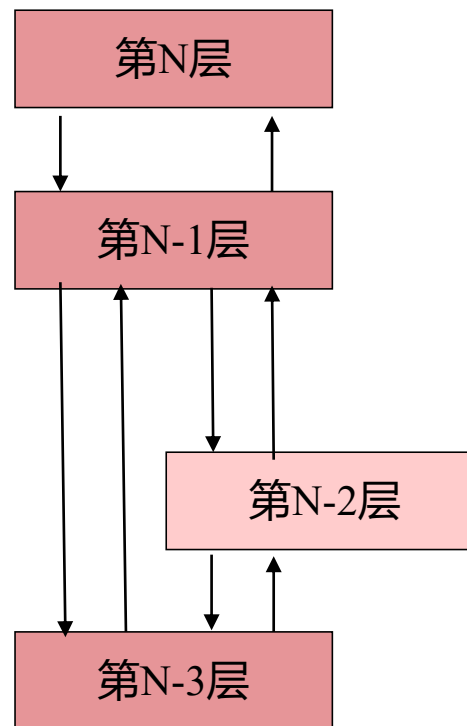


1.3.1 分层模式

- 三种典型的层次结构
 - 普通型、混合型（跨层调用）



(a) 普通型



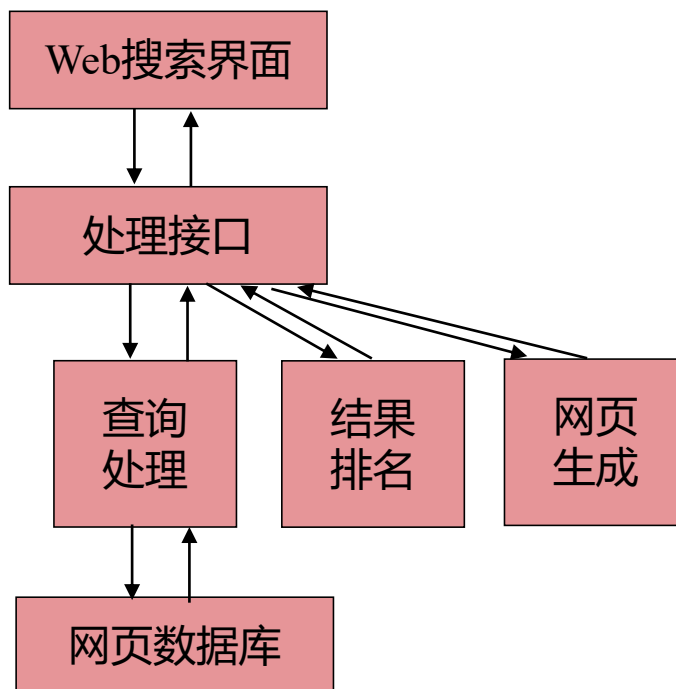
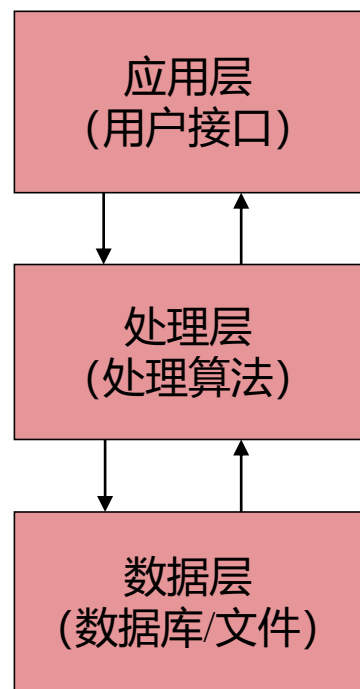
(b) 混合型

1.3.1 分层模式

- 传统的应用系统三层结构

- 应用层（用户接口）、处理层（函数）、数据层（需要应用层处理的数据）

- 例：搜索引擎（混合型）



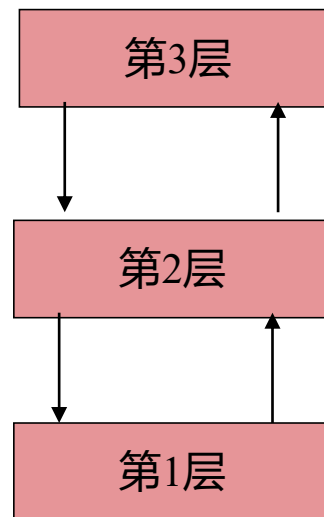
1.3.1 分层模式

■ 分层模式的优点

- 支持“逐步求精”设计，结构清晰，容易实现
- 每个层次具有局部性，容易扩展
- 按照开放性标准设计层次接口，容易系统集成用

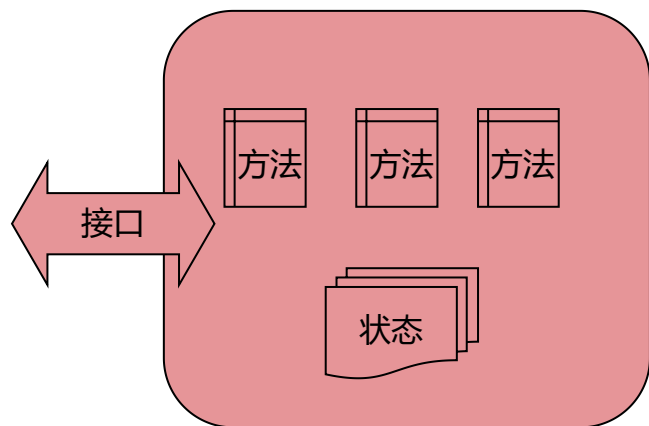
■ 分层模式的不足

- 不是所有的系统都能严格地划分成层次
- 严格的层次可能造成系统的性能下降

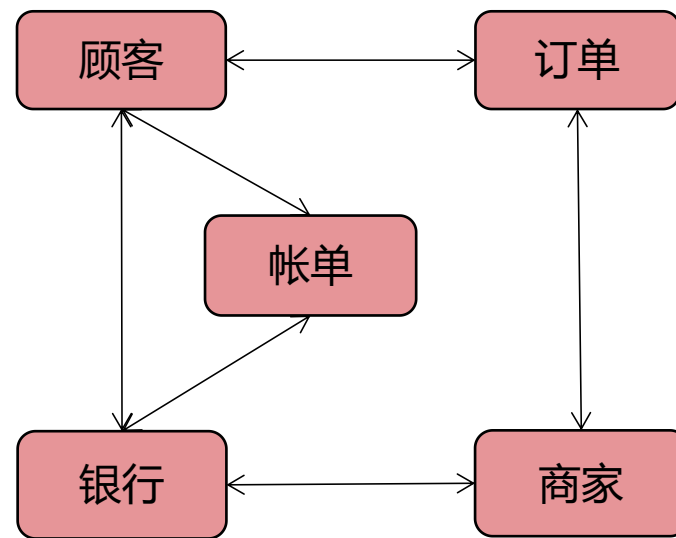


1.3.2 对象模式

- **对象模型** (Object-Oriented)
 - 外部结构：接口(interface)
 - 内部结构：状态(state)和方法(method)
 - 特点：封装性(encapsulation)，实现的细节被封装，对外不可见



(a) 面向对象模型



(b) 对象模式举例

1.3.2 对象模式

■ 主要特点

- 基于对象模型，每个对象可以调用任意其他对象
- 每个构件对应一个对象；可位于不同计算机节点上
- 对象之间通信利用远程方法调用（RMI）实现；

■ 分布式对象系统举例

- Java平台（如：JDK、Eclipse、J2EE）
- .Net系统



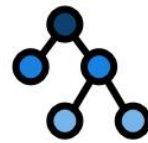
Abstraction



Encapsulation



Modularity



Hierarchy

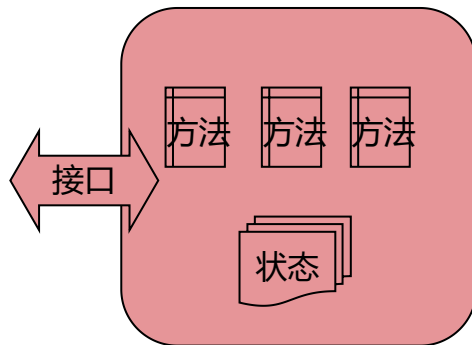
1.3.2 对象模式

■ 对象模式的主要优点

- 支持软件工程的模块化、软件复用
- 支持良好的功能扩充
- 容易维护。对象的封装性，易于检错和纠错

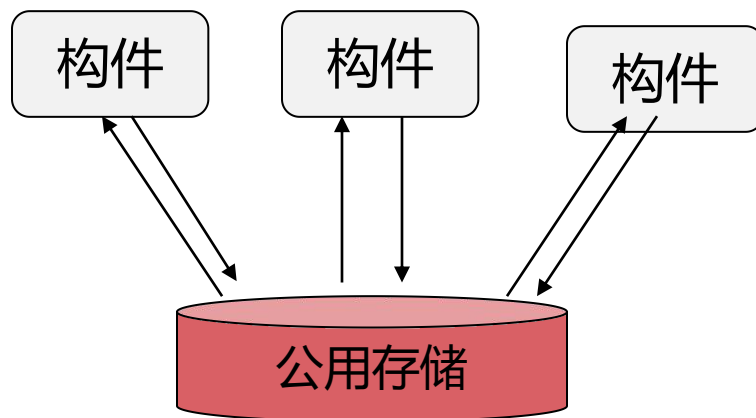
■ 对象模式的不足

- 调用关系可能会很复杂，不适于大规模系统



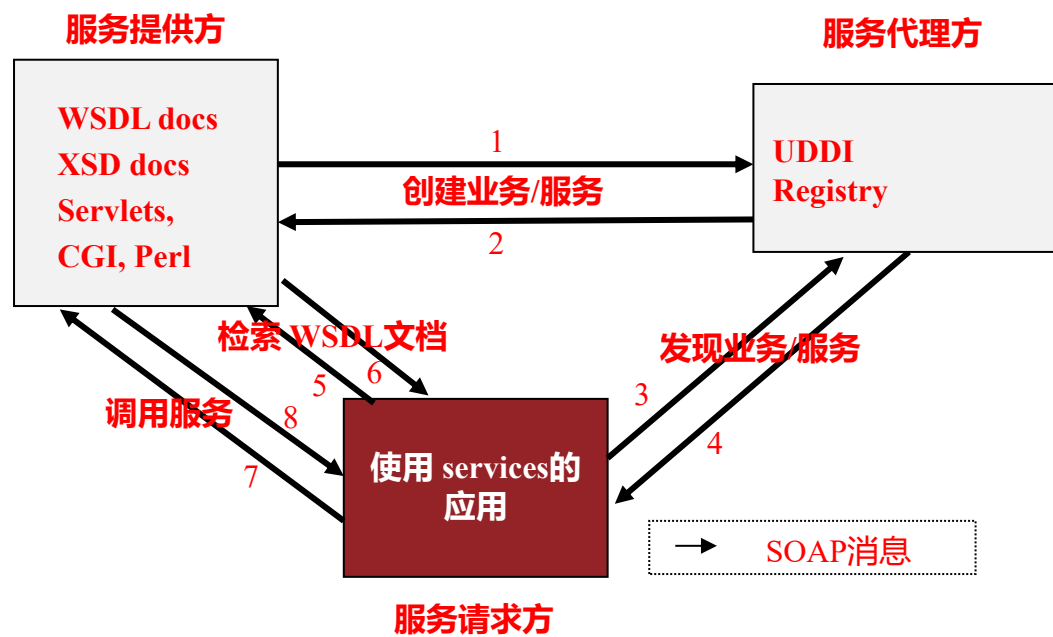
1.3.3 数据中心模式

- **构件间的通信**，通过一个公用的存储（如共享的分布式文件系统）实现
 - 例：SOA体系结构，构件使用UDDI数据服务，建立基于Web的分布式系统
 - 例：Hadoop系统，利用HDFS分布式文件系统，进行分布式计算



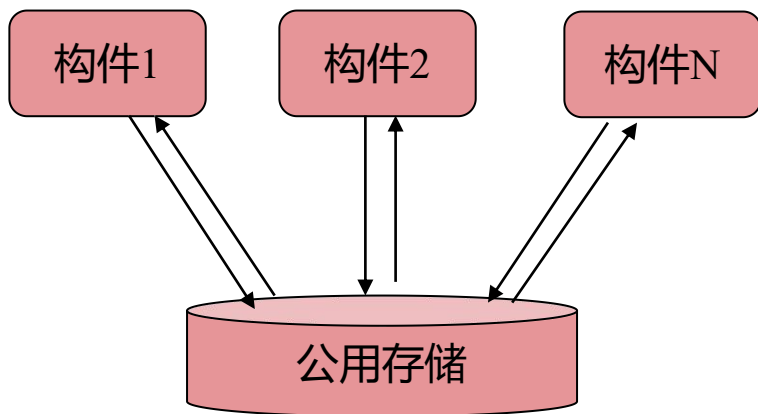
1.3.3 数据中心模式

- 举例：面向服务的体系结构 (SOA, Service-Oriented)
 - 服务(Service)：相当于对象
 - 服务合成：由多个服务组成一个大的服务
 - 应用：如EAI（企业应用集成）

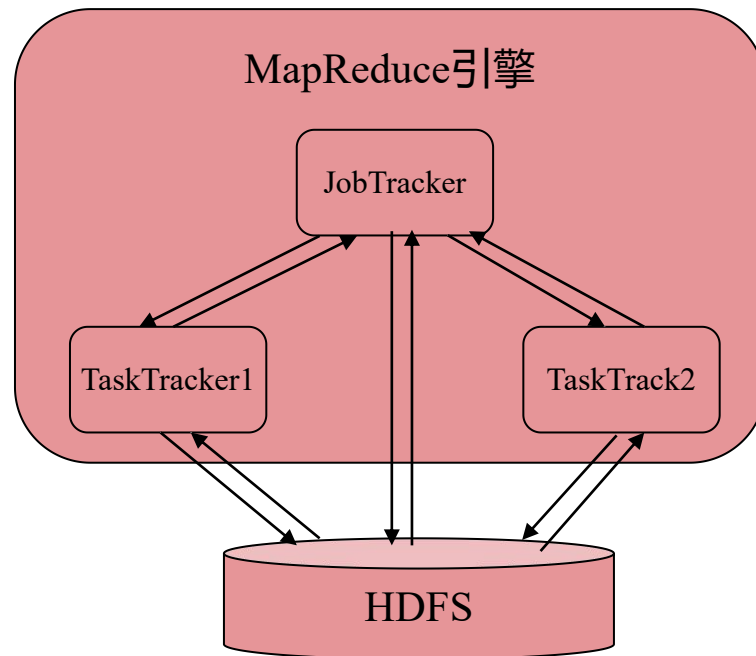


1.3.3 数据中心模式

- 举例：Hadoop系统，利用HDFS分布式文件系统，进行分布式计算



(a) 数据中心模式



(b) Hadoop的系统结构

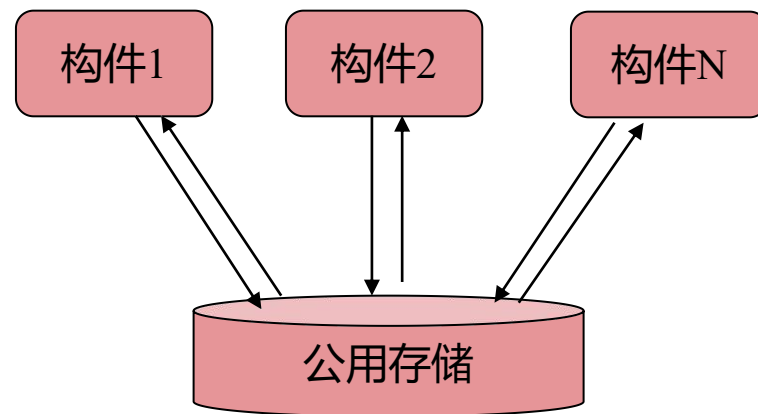
1.3.3 数据中心模式

- **数据中心模式的优点**

- 实现简单，类似共享内存(shared-memory)的并行计算模型，特别适用于数据密集型计算，减少了系统开发的复杂性。

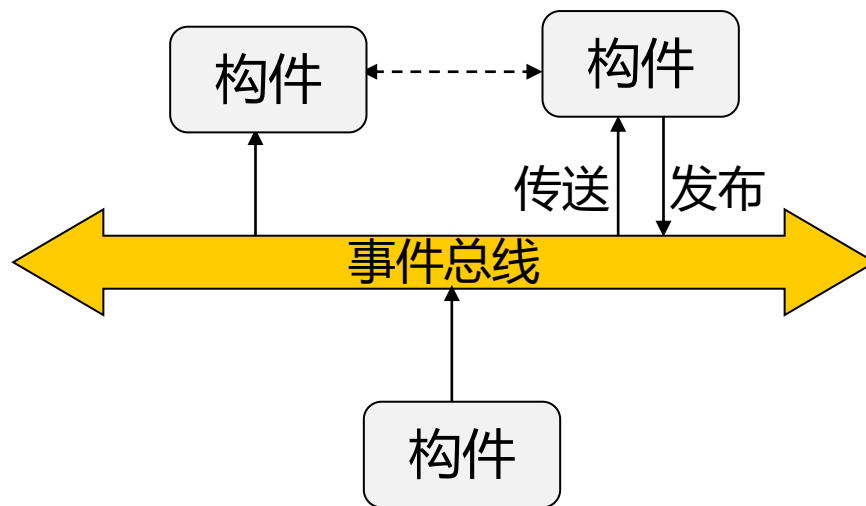
- **数据中心模式的不足**

- 底层的共享存储系统的实现比较复杂，对于大规模分布场景，如跨地理区域，性能会受到很大局限



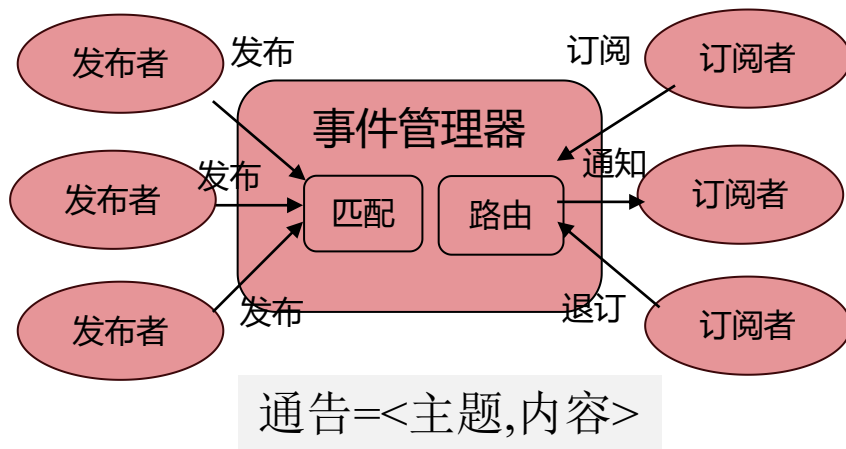
1.3.4 事件驱动模式

- 构件通过**事件总线**连接在一起
 - 构件之间的通信，通过事件（可带有数据）的传播实现
 - **事件总线**，负责在系统中传播关于事件的消息
 - 例如，微博、微信、QQ



1.3.4 事件驱动模式

- 例：基于事件驱动模式的发布/订阅系统
 - 发布者(publisher):发布通告的进程
 - 订阅者(subscriber): 订阅特定通告的进程
 - 匹配: 检查发布的内容与订阅的内容是否相同
 - 事件: 如果产生了一个匹配, 触发一个事件



1.3.4 事件驱动模式

- 事件驱动模式是对象模式的一个特例，一种**松耦合系统**
 - 处理和协调解耦，构件之间解耦
- **事件驱动模式的优点**
 - 适合事件处理应用，方便及时地处理事件
 - 具有较好的扩展性，增加一个新构件，只需在事件总线上进行注册，不影响到其它构件。
- **事件驱动模式的不足**
 - 发布事件的构件难以确定其它构件对该事件是否响应、以及响应次序等情况

1.3.5 资源型模式

- **RESTful 体系结构 (Representational State Transfer)**
 - 分布式系统看作是由构件管理的资源的集合
 - 资源可被应用添加、删除、检索和修改
- **特点**
 - 每个资源有一个命名模式进行标识, 例: URI
 - 所有资源服务提供同样的接口, 例: HTTP操作
 - 资源服务的消息是自描述的, 例: "Content-Type"
 - 在服务中执行一个操作后, 构件不保留关于其调用者的任何信息(无状态执行)



Roy T. Fielding, <https://www.ics.uci.edu/~fielding/>, *Architectural Styles and the Design of Network-based Software Architectures*, 2000

1.3.5 资源型模式

- 举例： **Amazon云存储服务 S3**
 - 对象（如文件）置于桶（如目录）
 - 命名： `http://BucketName.s3.amazonaws.com/ObjectName`
 - 创建一个桶/对象： PUT URI
 - 列表对象： GET 桶名
 - 读对象： GET URI
- **4个基本操作**



Operation	Description
PUT	Create a new resource
GET	Retrieve the state of a resource in some representation
DELETE	Delete a resource
POST	Modify a resource by transferring a new state

1.3.5 资源型模式

- 对比：Amazon S3 的SOAP接口，16个操作
- 例：创建一个桶

import bucket

bucket.create("mybucket ")

PUT "http://mybucket.s3.amazonaws.com/"

SOAP接口

RESTful接口

Bucket operations	Object operations
ListAllMyBuckets	PutObjectInline
CreateBucket	PutObject
DeleteBucket	CopyObject
ListBucket	GetObject
GetBucketAccessControlPolicy	GetObjectExtended
SetBucketAccessControlPolicy	DeleteObject
GetBucketLoggingStatus	GetObjectAccessControlPolicy
SetBucketLoggingStatus	SetObjectAccessControlPolicy

1.3.5 资源型模式

- 资源型模式是数据中心模式的一个扩展
 - 公用存储中不仅管理数据，还管理各种资源
- **资源型模式的优点**
 - 与面向服务的架构（SOA）相比，更加简单、轻量
 - 与SOAP相比，具有更高的效率和更好的易用性
- **资源型模式的不足**
 - 安全性方面不如SOAP

1.3.6 软件体系结构小结

风格	功能性	简易性	可伸缩性
分层模式	强	弱	良
对象模式	强	良	良
数据中心模式	弱	强	强
事件驱动模式	弱	良	强
资源型模式	良	强	强

第一章、分布式系统概论

01

1.1 分布式系统概念

02

1.2 分布式系统关键技术

03

1.3 分布式系统软件体系结构

04

1.4 分布式系统系统体系结构



1.4 系统体系结构

- 系统体系结构 (System Architecture)
 - 软件体系结构的具体实例 (**物理实现**)
 - 根据软件中各个构件的部署位置和交互方式，确定为不同类型的体系结构
- 共分为三种体系结构
 - **中心化(centralized)**：存在**中心构件**，其它构件必须与中心构件交互
 - **去中心化(decentralized)**：没有中心化构件，所有构件之间都可以交互
 - **混合方式**：包含以上两种方式

软件体系结构
(逻辑层)

系统体系结构
(物理层)



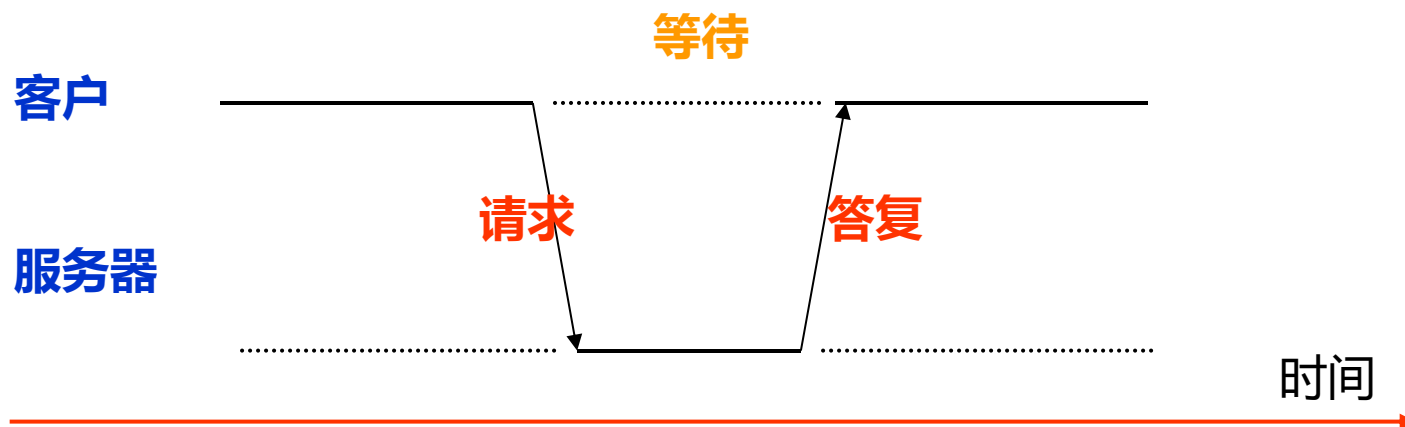
应用软件 (上层)

系统软件 (下层)

[2] Len Bass, Paul Clements, Rick Kazman Software Architecture in Practice. Addison-Wesley Professional, MA., 4nd edition, 2021

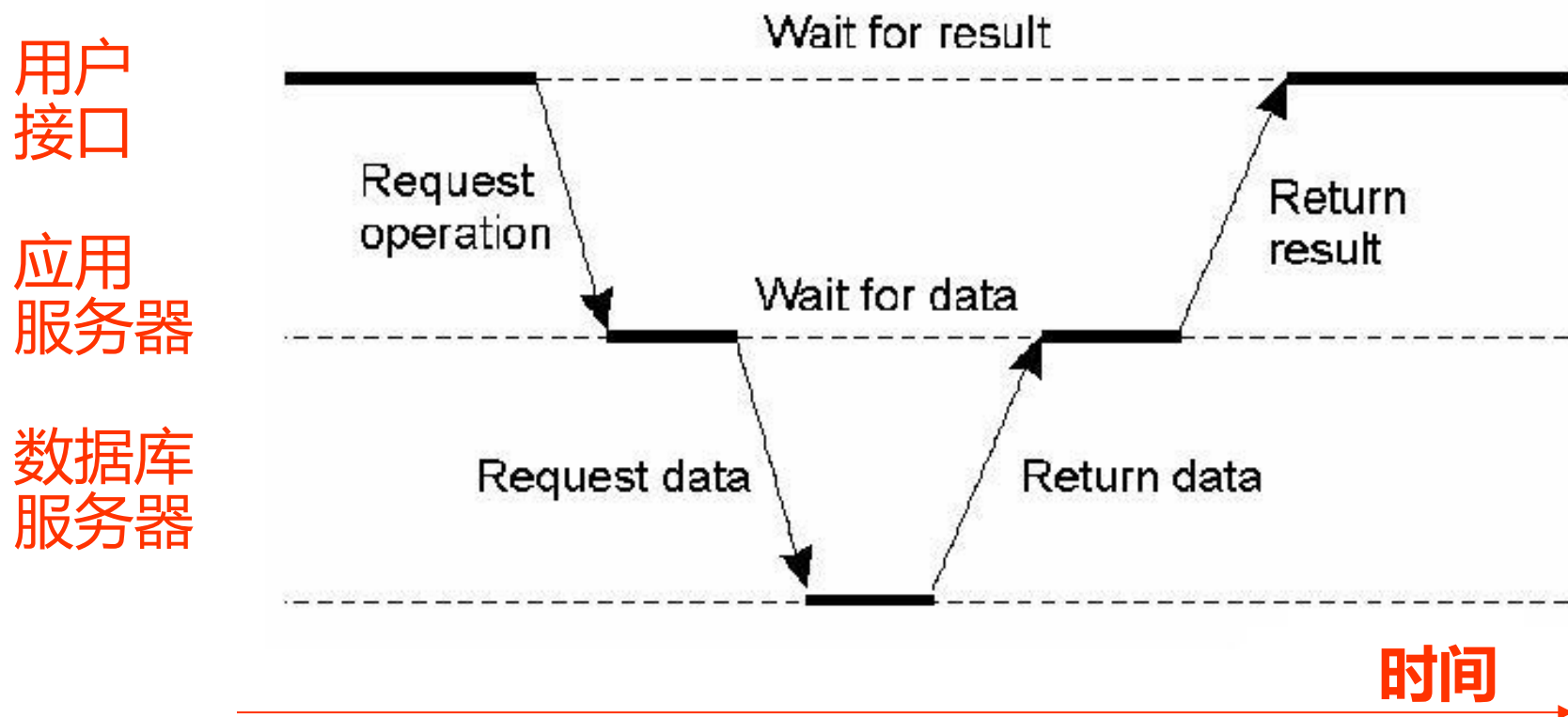
1.4.1 中心化体系结构

- 基本的客户机/服务器模型
 - **服务器(server)**: 服务方, 实现特定服务的进程
 - **客户机(client)**: 委托方, 向服务器提出服务请求的进程
 - 客户机和服务器可位于不同的计算机
 - **交互模式**: 请求/答复



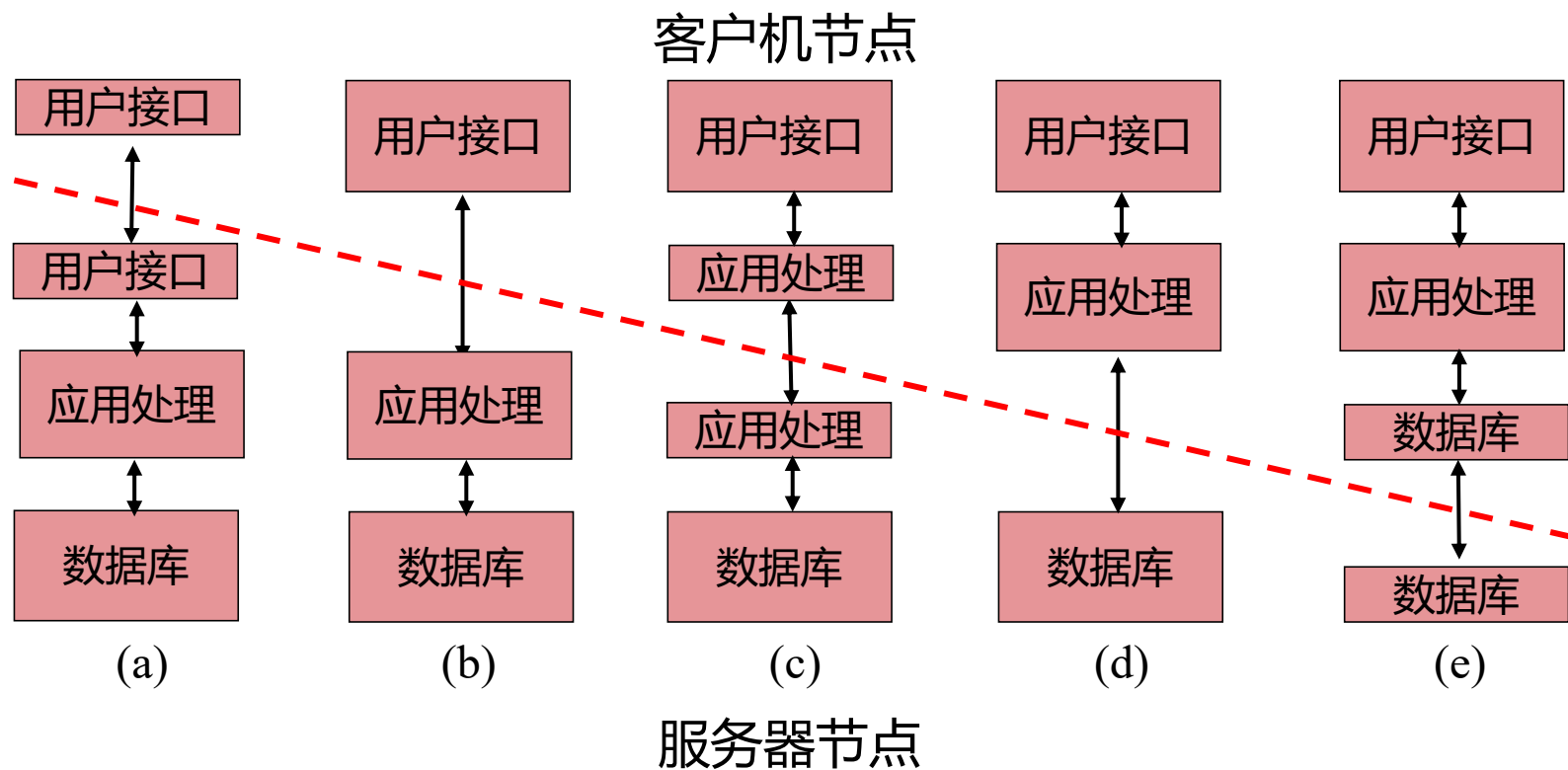
三层C/S体系结构举例

- 传统的C/S组织方式：两层和三层



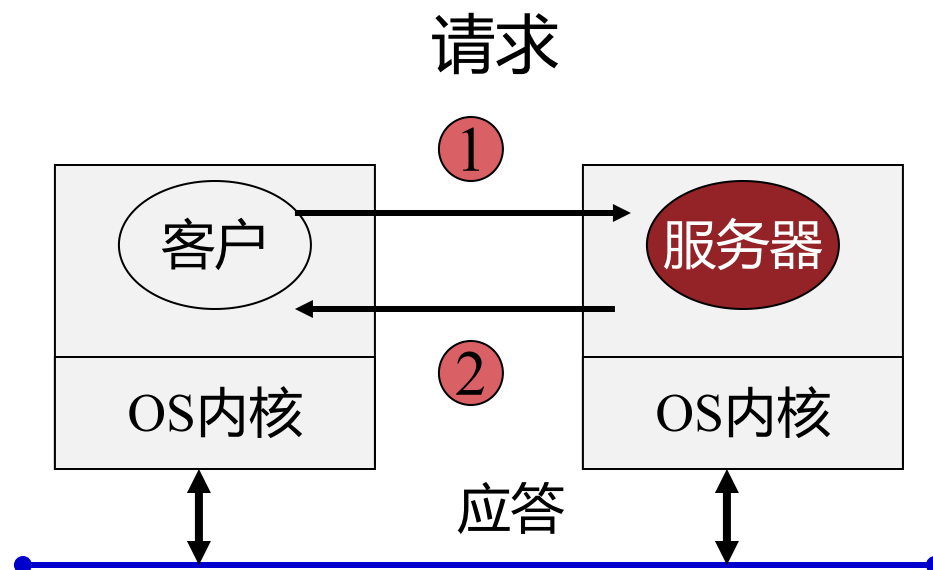
多层中心化体系结构

- 瘦客户机/胖服务器：用户接口简单，但后端负载重
- 胖客户机/瘦服务器：能提高性能，但管理困难



客户机/服务器模型的实现

- 客户机：用户进程
- 服务器机：能够提供服务的进程



通信协议

- 简化的请求/应答协议
 - 无路由、无连接
 - 两个基本操作
 - `send(dest, &mptr)`
 - `receive(addr,&mptr)`



客户机和服务器举例

- 头文件
- 客户机请求服务器进行文件的读、写、创建、删除操作
- C语言和Linux环境

```
/* Definitions needed by clients and servers. */
#define TRUE 1
#define MAX_PATH 255 /* maximum length of file name */
#define BUF_SIZE 1024 /* how much data to transfer at once */
#define FILE_SERVER 243 /* file server's network address */

/* Definitions of the allowed operations */
#define CREATE 1 /* create a new file */
#define READ 2 /* read data from a file and return it */
#define WRITE 3 /* write data to a file */
#define DELETE 4 /* delete an existing file */

/* Error codes. */
#define OK 0 /* operation performed correctly */
#define E_BAD_OPCODE -1 /* unknown operation requested */
#define E_BAD_PARAM -2 /* error in a parameter */
#define E_IO -3 /* disk error or other I/O error */

/* Definition of the message format. */
struct message {
    long source; /* sender's identity */
    long dest; /* receiver's identity */
    long opcode; /* requested operation */
    long count; /* number of bytes to transfer */
    long offset; /* position in file to start I/O */
    long result; /* result of the operation */
    char name[MAX_PATH]; /* name of file being operated on */
    char data[BUF_SIZE]; /* data to be read or written */
};
```

客户和服务器的举例

■ Server程序

```
#include <header.h>
void main(void) {
    struct message m1, m2;          /* incoming and outgoing messages */
    int r;                          /* result code */

    while(TRUE) {                  /* server runs forever */
        receive(FILE_SERVER, &m1); /* block waiting for a message */
        switch(m1.opcode) {        /* dispatch on type of request */
            case CREATE: r = do_create(&m1, &m2); break;
            case READ:   r = do_read(&m1, &m2); break;
            case WRITE:  r = do_write(&m1, &m2); break;
            case DELETE: r = do_delete(&m1, &m2); break;
            default:     r = E_BAD_OPCODE;
        }
        m2.result = r;              /* return result to client */
        send(m1.source, &m2);      /* send reply */
    }
}
```

客户和服务器的举例

■ Client程序

```
(a)
#include <header.h>
int copy(char *src, char *dst){
    struct message ml;
    long position;
    long client = 110;

    initialize( );
    position = 0;
    do {
        ml.opcode = READ;
        ml.offset = position;
        ml.count = BUF_SIZE;
        strcpy(&ml.name, src);
        send(FILESERVER, &ml);
        receive(client, &ml);

        /* Write the data just received to the destination file.
        ml.opcode = WRITE;
        ml.offset = position;
        ml.count = ml.result;
        strcpy(&ml.name, dst);
        send(FILE_SERVER, &ml);
        receive(client, &ml);
        position += ml.result;
    } while( ml.result > 0 );
    return(ml.result >= 0 ? OK : ml.result);
}

/* procedure to copy file using the server */
/* message buffer */
/* current file position */
/* client's address */

/* prepare for execution */

/* operation is a read */
/* current position in the file */
/* how many bytes to read */
/* copy name of file to be read to message */
/* send the message to the file server */
/* block waiting for the reply */

/* operation is a write */
/* current position in the file */
/* how many bytes to write */
/* copy name of file to be written to buf */
/* send the message to the file server */
/* block waiting for the reply */
/* ml.result is number of bytes written */
/* iterate until done */
/* return OK or error code */
```

消息格式举例

```
struct message {  
    long source; /* 发送者标识 */  
    long dest;   /* 接受者标识 */  
    long opcode; /* 操作码：读、写、创建、删除 */  
    long result; /* 返回结果代码：成功、失败 */  
    long offset; /* 读写位置 */  
    long count;  /* 读写计数 */  
    char filename[MAX_PATH]; /* 文件名 */  
    char data[BUF_SIZE]; /* 数据区 */  
}
```

服务器程序举例

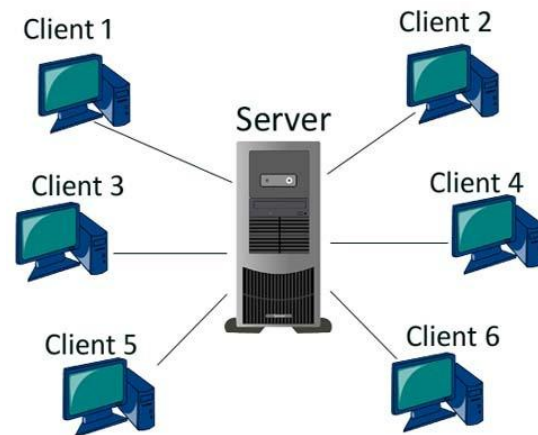
```
void main(void){
    struct message m1,m2; /* 输入、输出的消息 */
    int r; /* 返回的执行结果 */
    while (1) {
        receive(FILE_SERVER,&m1); /* 等待客户请求 */
        case(m1.opcode) { /* 执行请求的操作 */
            case READ: r = do_read(&m1,&m2); break;
            :
            default:    r = E_BAD_OPCODE;
        }
        m2.result = r;
        send(m1.source,&m2); /* 返回结果 */
    }
}
```

客户程序举例

```
int read(char *file, int position , int nbytes, char *buf){  
    struct message m1; /* 消息缓冲区 */  
  
    m1.opcode = READ;    /* 设置参数 */  
    m1.offset  = position; /* 读位置 */  
    m1.count   = nbytes;   /* 读长度 */  
    strcpy(&m1.filename,file) /* 文件名 */  
  
    send(FILE_SERVER, &m1); /* 发送请求 */  
    receive(CLIENT,&m1); /* 等待服务器应答 */  
  
    if(m1.result == OK) strcpy(buf, &m1.data); /* 置缓冲区 */  
    return(m1.result); /* 返回结果 */  
}  
}
```

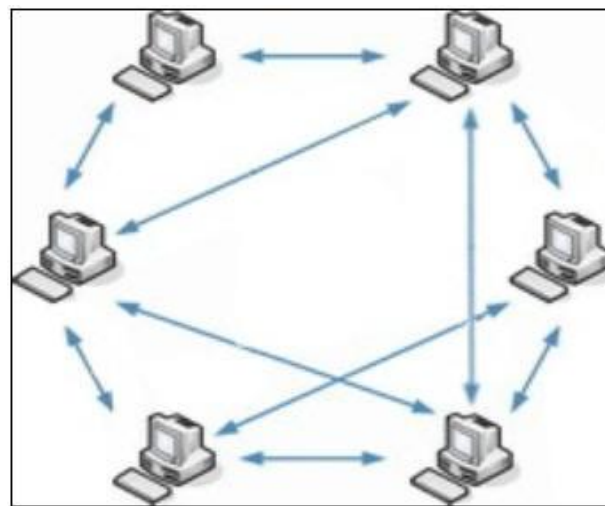
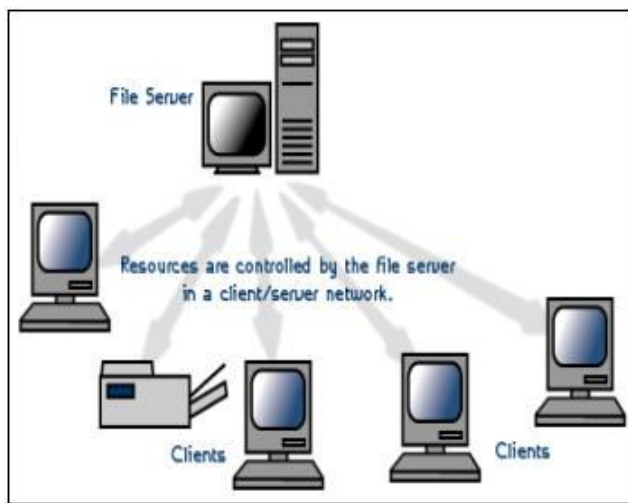
中心化体系结构

- 客户机/服务器模型特点
 - 以服务器为中心的中心化结构
 - 实现简单，管理方便
- 存在问题
 - **性能瓶颈**：服务器的性能影响到整个系统的性能
 - **单点故障**：服务器的故障导致整个系统故障
 - 系统的可伸缩性和健壮性存在很大的局限



1.4.2 去中心化体系结构

- 从C/S模式到P2P模式
 - **去中心化**: 可扩展性、健壮性
 - **高性价比**: 有效聚集计算资源和存储资源



引自：中科院计算所罗杰文 Peer to Peer 综述 <http://www.intsci.ac.cn/users/luojw/P2P/ch01.html>

P2P技术

■ P2P 应用

- 文件内容共享和下载，例如Napster、Gnutella、eDonkey、eMule、Maze、BT等；
- 计算能力和存储共享，例如[SETI@home](#)、Avaki、Popular Power等；
- 协同与服务共享平台，例如JXTA、Magi、Groove等；
- 即时通讯工具，包括ICQ、QQ、Yahoo Messenger、MSN Messenger等；
- P2P通讯与信息共享，例如[Skype](#)、Crowds、Onion Routing等；
- 网络电视：沸点、PPStream、PPLive、QQLive、SopCast等。
- 区块链系统：比特币，以太坊，Hyperledger等

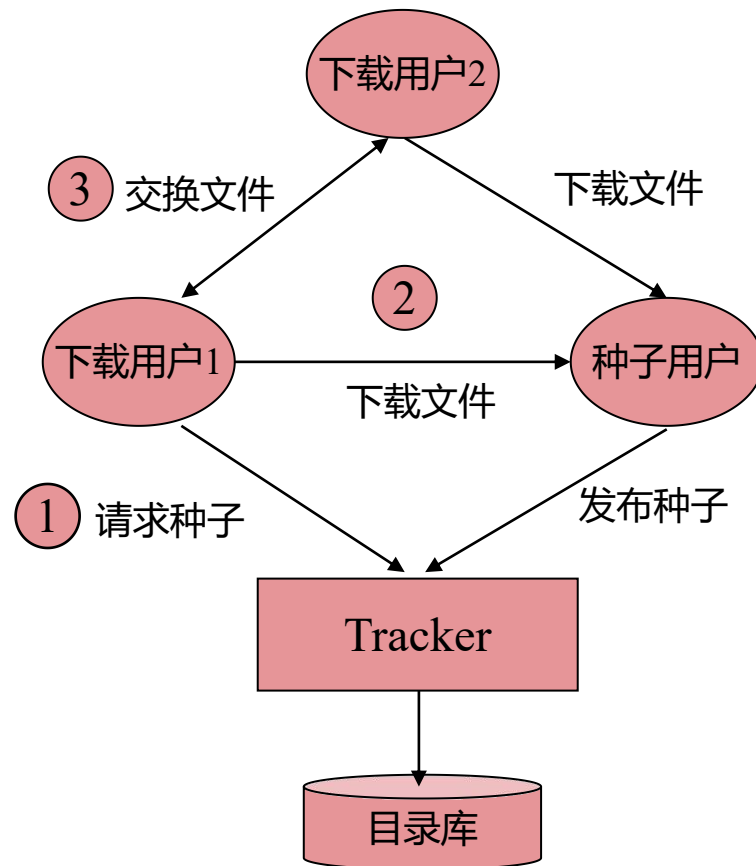


P2P技术

- **P2P网络**: 建立在Internet之上一种覆盖网络。
 - 覆盖网络(overlay network): 建立在另一个网络上的网络, 属于应用层网络, 不考虑或很少考虑网络层, 物理层的问题, 如, 无需IP地址进行通信。
- **P2P网络的拓扑结构 (Topology)**
 - 中心化 (Centralized)
 - 去中心化有结构拓扑 (Decentralized Structured, 也称作DHT网络)
 - 去中心化无结构拓扑(Decentralized Unstructured);
 - 部分去中心化拓扑 (Partially Decentralized)

中心化P2P网络

- 存在**中心节点**
 - 资源的发现依赖**中心化目录**
 - 例：文件共享软件BT（BitTorrent），通过**中央索引服务器（Tracker）**保存所有上传文件的索引和存放位置信息
- **优点**
 - 维护简单，资源发现效率高
- **存在问题**
 - 中心节点的单点失效、维护成本高



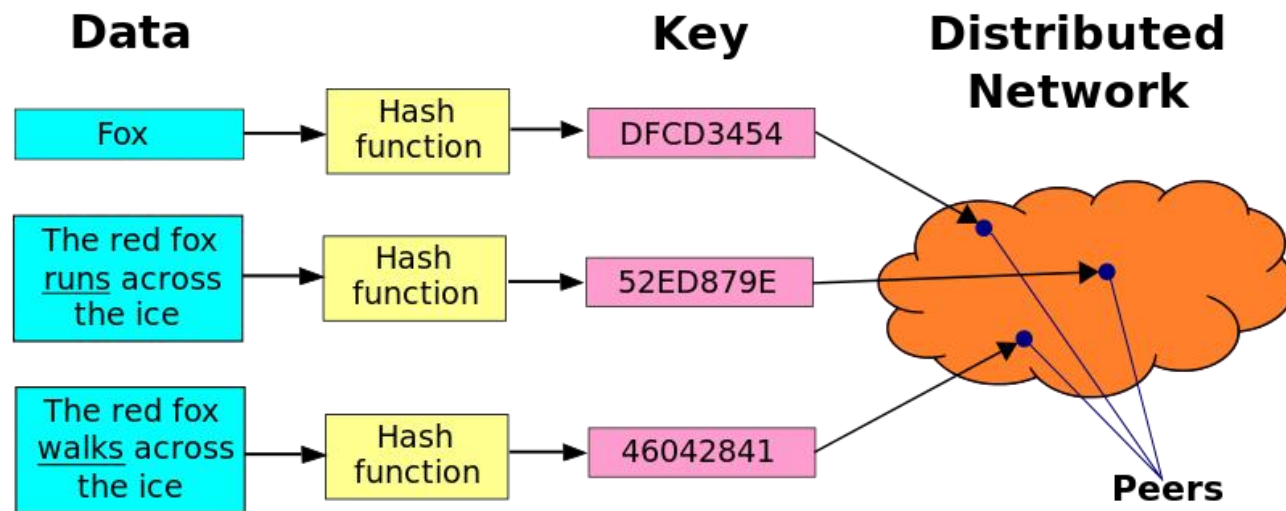
去中心化有结构P2P网络

- 无中心节点
- 将节点组织成特殊的**拓扑结构**：环、树、网格等
- **索引**：采用分布式散列表（Distributed Hash Table, DHT），组织和管理P2P网络中的节点和数据。
- 通过DHT，将数据<key,value>映射到P2P节点上，即可找到数据的存储位置。
- DHT由P2P网络中的节点共同维护
 - 散列表被分割成**不重叠的散列块**，每个节点被分配给一个属于自己的散列块，并成为这个散列块的管理者。

http://en.wikipedia.org/wiki/Distributed_hash_table

去中心化有结构P2P网络

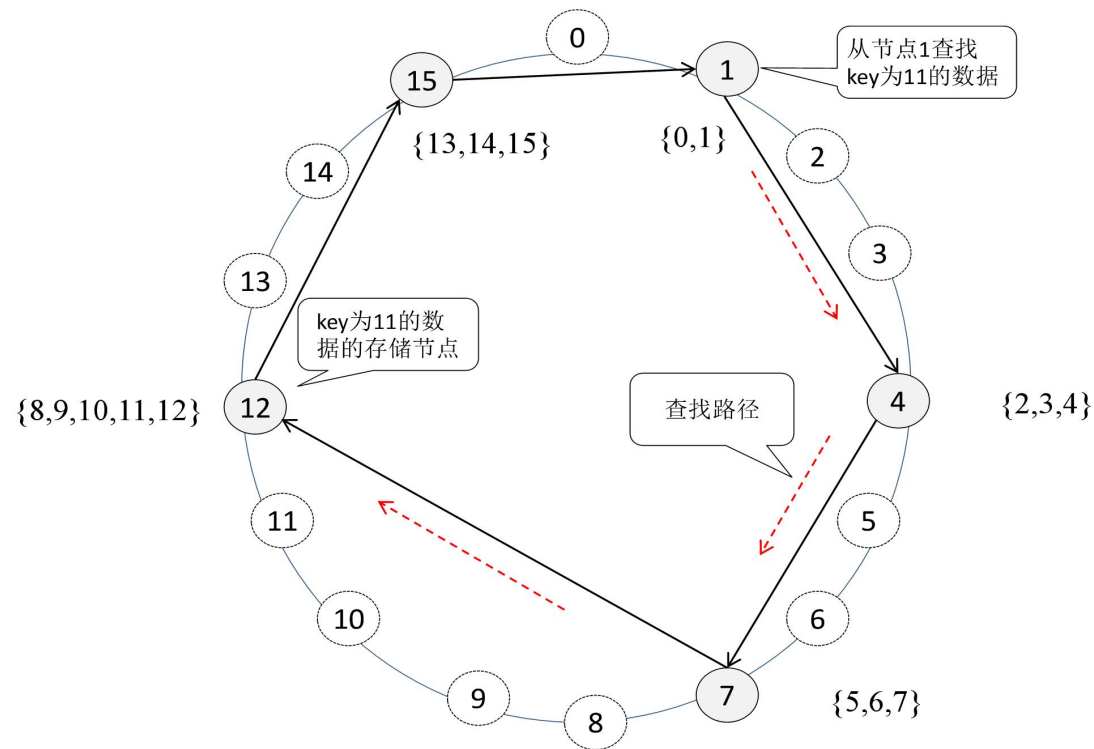
- 举例：
 - DHT加密散列函数，将一个对象的名字或关键词映射为128位或160位的散列值
 - 典型系统: Tapestry, Pastry, Chord和CAN



去中心化有结构P2P网络

举例：MIT 的 Chord 系统 – 环结构

- 网络节点按照一定的方式分配一个唯一标识符(Node ID)
- 通过散列运算为对象产生一个唯一的标识符(Object ID)
- 该对象存储在与其ID相等或相近的节点上
- **分布式查找协议**：将指定的关键字(Key)映射到对应的节点(Node)
- **时间复杂性** $O(\log(N))$

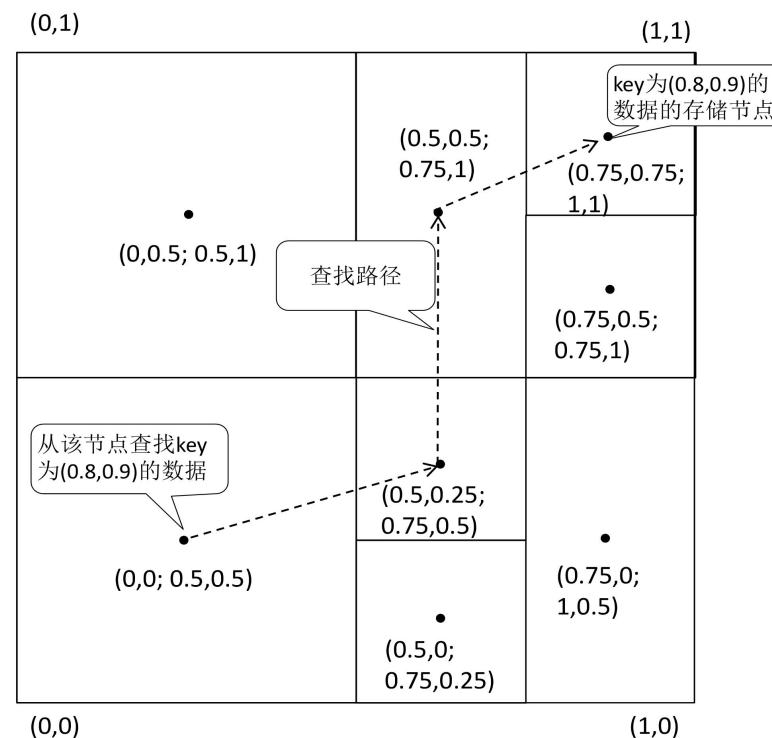


Chord: A scalable peer-to-peer lookup service for Internet applications, Ion Stoica, Robert Morris, David Karger, et al. SIGCOMM 2001.

去中心化有结构P2P网络

举例：UC Berkeley 的 CAN 系统 – 网格结构

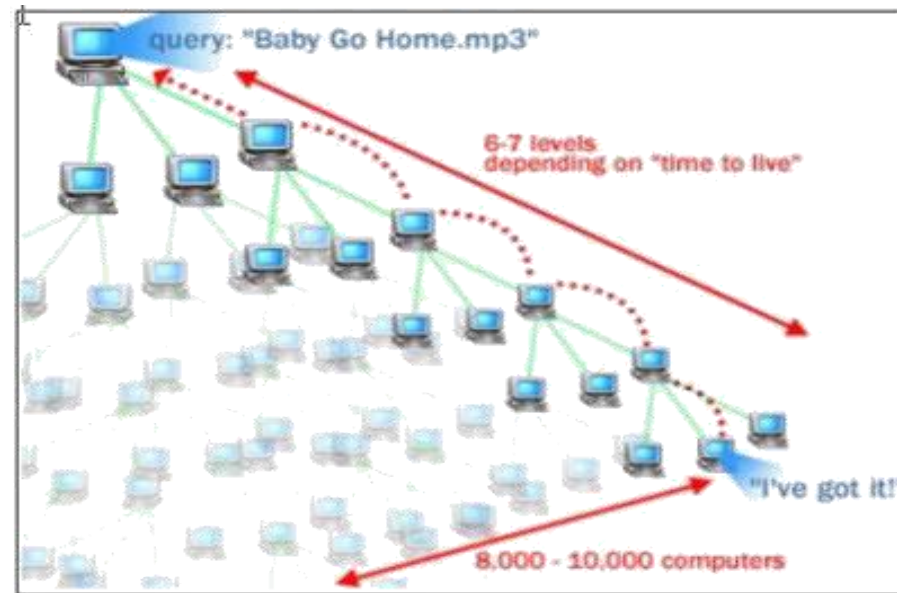
- 所有节点映射到n维笛卡尔空间
- 将<key, value> 键值对中key做散列运算, 得到一个坐标点
- 将<key, value> 键值对存储在拥有该点所在区域的节点内
- **路由**：将请求传给当前节点四邻中距离最近目标点的节点
- **路由路径长度**= $O(n/d)$, n个节点, d个维数



Ratnasamy et al.. *A Scalable Content-Addressable Network*. SIGCOMM 2001

去中心化无结构P2P网络

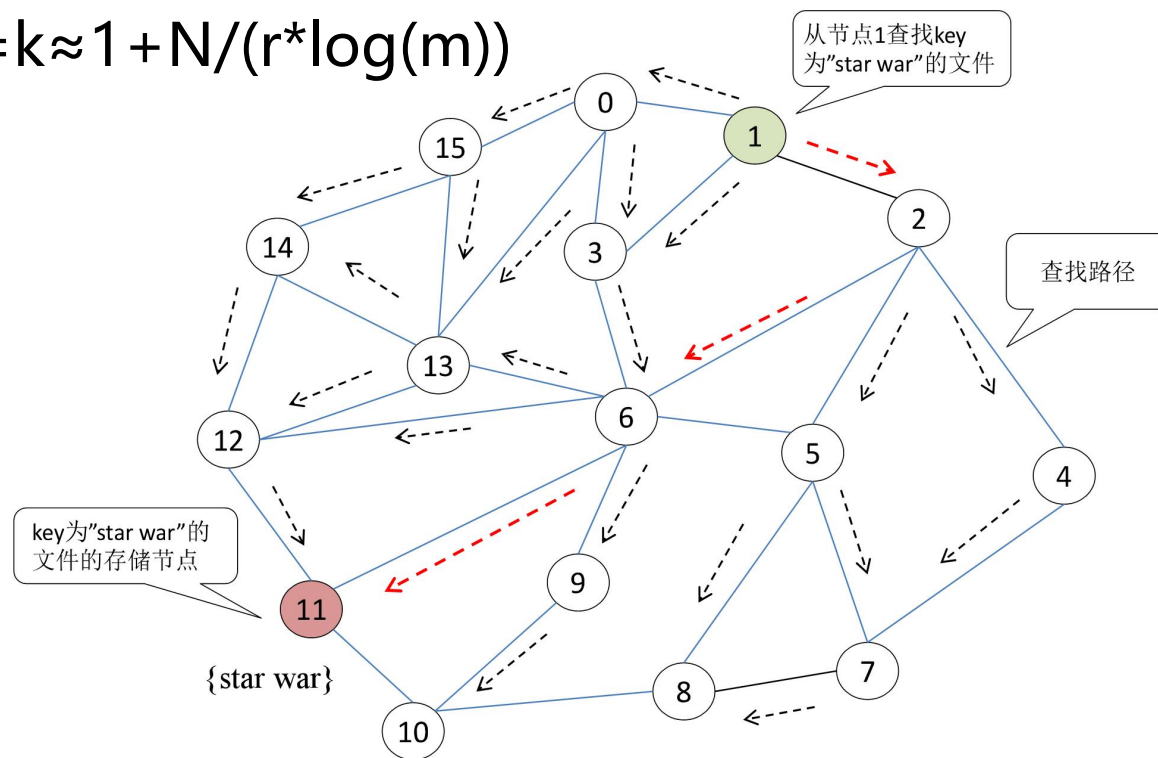
- 去中心化无结构P2P网络
 - 无中心节点、无索引结构
 - 每个节点保存有邻居列表
- 洪泛(Flooding)搜索算法
 - 节点u如果拥有请求数据d，则返回；否则向其所有的邻居节点发送请求d
 - 节点v接收到请求后，如果曾经收到过该请求，则忽略；
 - 否则,如果不超过TTL，继续搜索（递归进行）
 - TTL(Time-To-Live)：搜索最大跳数的阈值



去中心化有结构P2P网络

■ 性能分析

- 设 u 的邻居个数(出度)为 m ，在 k 步之后，搜索规模 $S \approx m * (m-1)^{k-1}$
- 如 N 个节点中有 r 个节点拥有数据 d ，当 $S \geq N/r$ 或则 $r/N * S \geq 1$ 时，则必能找到 d
- 时间复杂性 $= k \approx 1 + N/(r * \log(m))$



去中心化有结构P2P网络

■ 随机游走(Random walk) 搜索算法

- 随机图 (Random Graph) : $\langle u, v \rangle$, 存在概率 $P[\langle u, v \rangle]$

1. 节点u如果拥有请求数据d, 则返回; 否则向随机选择的一个邻居节点v发送请求数据d
2. 节点v接收到请求后, 如果曾经收到过, 则忽略;
3. 否则, 继续进行搜索 (递归进行)

■ 性能分析

- 设N个节点, 每个数据项存在于r个随机选择的节点

- 设 $P[k]$ 为k步之后搜索到d的概率

$$\mathbb{P}[k] = \frac{r}{N} \left(1 - \frac{r}{N}\right)^{k-1}.$$

- 设S为搜索规模, 为预期需要搜索的节点个数

- 设 $r/N=0.001$, 使用随机游走算法, 则 $S \approx 1000$

$$S = \sum_{k=1}^N k \cdot \mathbb{P}[k] = \sum_{k=1}^N k \cdot \frac{r}{N} \left(1 - \frac{r}{N}\right)^{k-1} \approx N/r \text{ for } 1 \ll r \leq N.$$

- 使用洪泛式算法, 设出度=10, 则需搜索4步, $S \approx 7290$

去中心化有结构P2P网络

■ P2P网络构造算法

- 关于所有节点的表，称为全局视图（global view）
- 每个节点维护一个部分视图（partial view），含有c个邻接点的列表。表项：
= <IP, age>
- 节点之间定期交换表项。由主动节点（可主动发起通信）和被动节点完成

■ 节点的加入

- 通过与任意一个已知的节点进行视图交换

■ 节点的删除

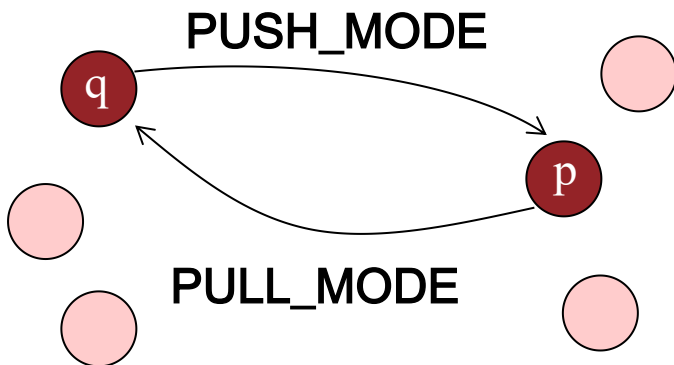
- 可自行离开，无需通知其他节点。
- 当其他节点发现某节点P不再响应时，将其从表中删除

Gossip-Based Peer Sampling, by: M. Jelasity, S. Voulgaris, R. Guerraoui, Joe Kilian, M. van Steen, ACM Transactions on Computer Systems (TOCS'07), Vol. 25, No. 3. (2007)

去中心化有结构P2P网络

■ 基于传言(gossip)的P2P网络构造算法

- 将视图数据比作传言
- 将一个固定时间间隔,定义为一个周期
- 在每个周期内,节点之间进行视图交换(View Change)



```
do forever
  wait(T time units) //等待一个周期
  p ← view.selectPeer()
  if PUSH_MODE then
    buffer ← ((MyAddress, 0)) // 0为初始年龄
    view.permute() //重新随机排列
    将最老的H个节点移到视图后部
    buffer.append(view.head(c/2-1))
    send buffer to p
  else // PULL_MODE
    send (null) to p // 发送空视图,触发响应
    receive bufferp from p
    view.select(c, H, S, bufferp)
    view.increaseAge()
```

(a) 主动节点q执行步骤

去中心化有结构P2P网络

■ P2P网络构造算法

```
do forever
  receive bufferq from q
  if PULL_MODE then
    buffer ← ((MyAddress, 0)) // 0为初始年龄
    view.permute()
    将最老的H个节点移到视图后部
    buffer.append(view.head(c/2-1))
    send buffer to q
  else // PUSH_MODE
    view.select(c, H, S, bufferq)
    view.increaseAge()
```

(b) 被动节点p的执行步骤

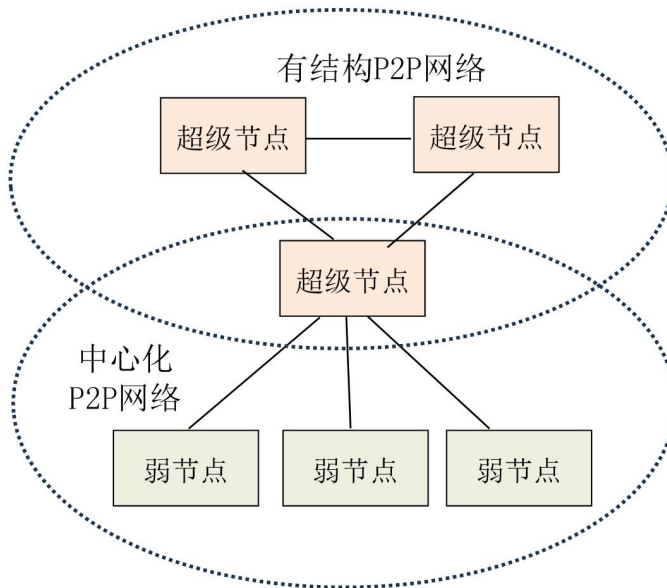
```
method view.select(c, H, S, buffer(p))
  // 执行如下操作
  view.append(buffer(p)) // 追加(合并视图)
  view.removeDuplicats() // 消重
  view.removeOldItems(min(H, view.size-c)) // 在保留c个节点前提下，去掉H个最老的节点
  view.removeHead(min(S, view.size-c)) // 在保留c个节点前提下，去掉S个最早交换来的节点
  view.removeAtRandom(view.size-c) // 随机去掉若干节点，使视图总节点个数为c
```

(c) 构建新的部分视图

PUSH_MODE=推送方式; PULL_MODE=拉取方式
c = 部分视图的大小; H=从q删除掉的旧节点数; S=删除从p交换来的节点数

分层组织的去中心化P2P网络

- 两层P2P网络结构
 - 目的：为了解决中心化P2P网络的可伸缩性问题
 - 底层：中心化P2P网络，连接一部分节点
 - 上层：基于下层，构造特定的结构化P2P网络



分层组织的去中心化P2P网络

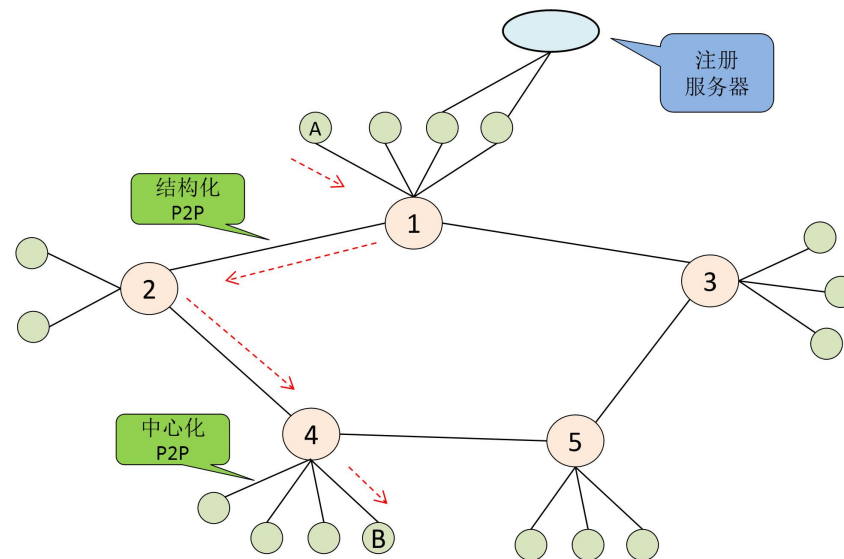
■ 两层P2P网络结构

- 超级节点(super peer): 能够维护索引或充当代理的节点
- 弱节点(weak peer): 普通节点。作为超级节点的客户端
- 弱节点与超级节点S连接方式:
 - 固定式: 绑定1个S节点; 可变式: 发现并连接新的S节点



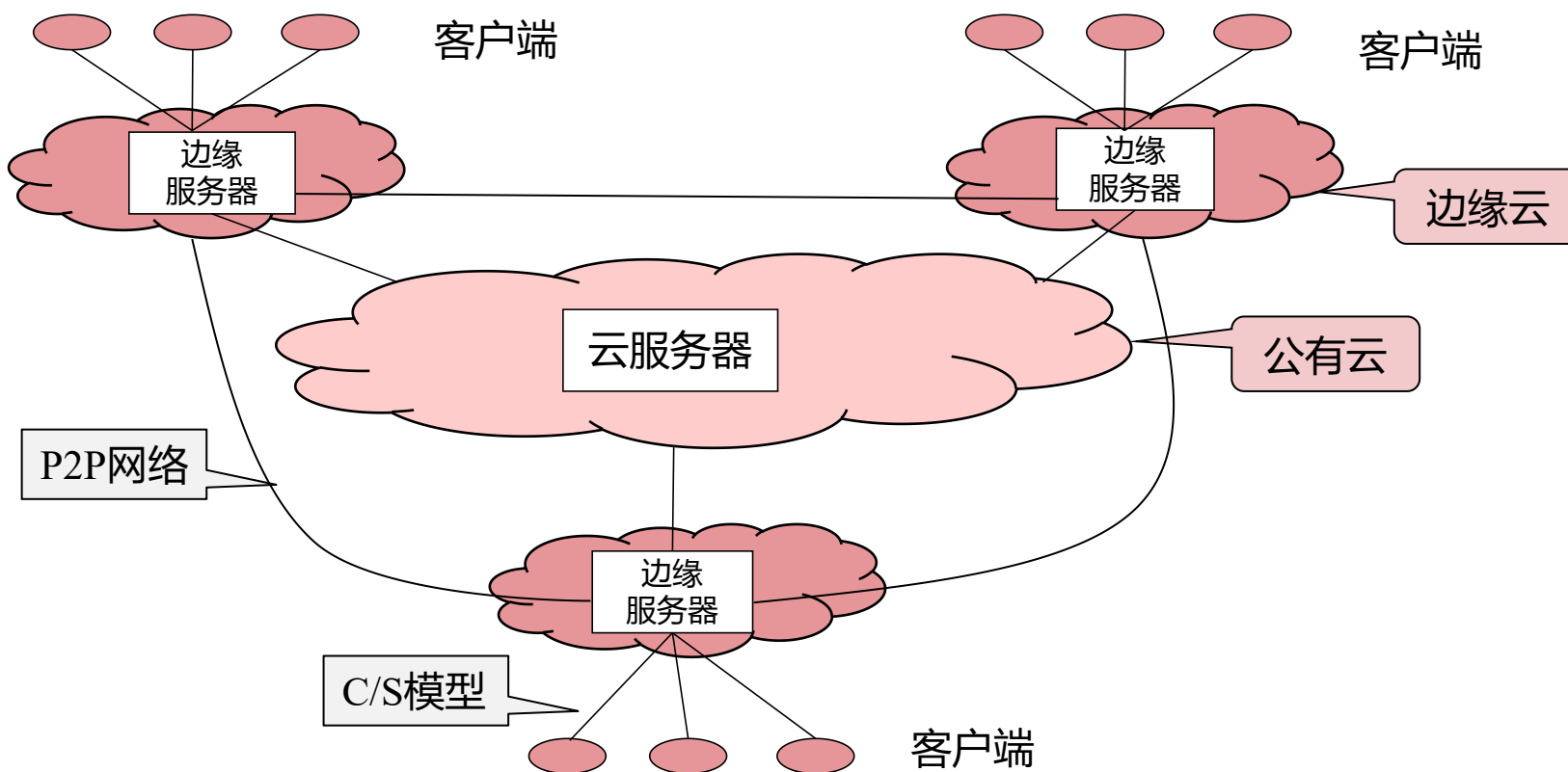
■ 举例: Skype VOIP网络

- 注册服务器: 登记合法用户ID
- 节点地址 (IP地址, 端口号)
- A查找弱节点B (用户):
 - A请求它的超级节点S,
 - S若没有, 则转发给其他超级节点S'



1.4.3 混合型体系结构

- 将客户/服务器中心化结构与P2P去中心化结构相结合
- 举例：云边端协同计算系统

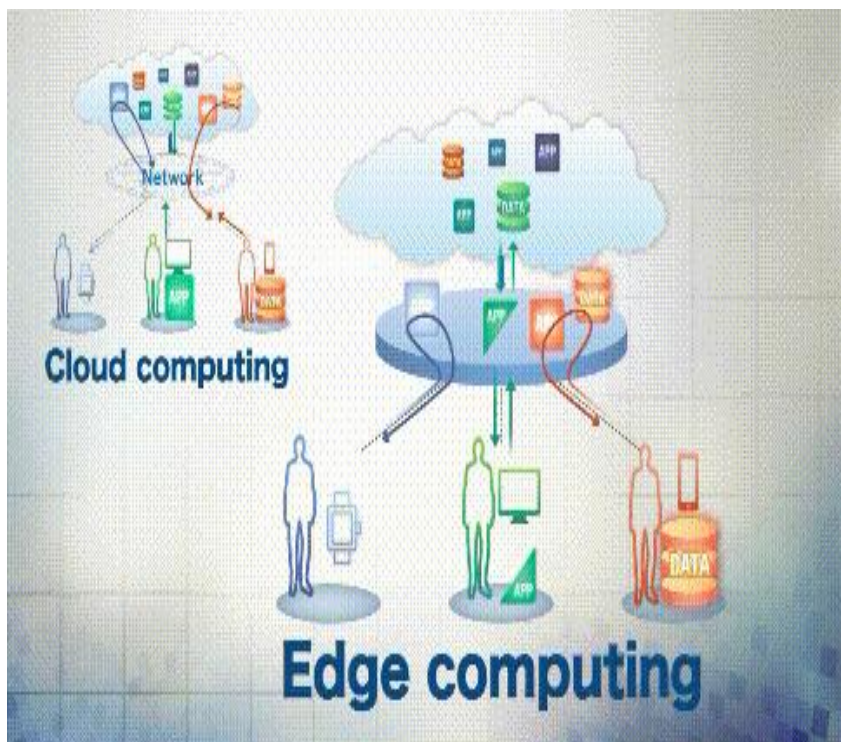


1.4.3 混合型体系结构

- 举例：云边端协同计算系统
 - 网络边缘：企业网络（内网）和互联网（外网）的边界
 - 边缘服务器系统（edge server），布署在网络边缘上
 - 内容交付网络（CDN）服务器，为邻近的客户端提供内容缓存，如提供网页、图片等。
 - 边缘计算服务器，如为物联网设备提供数据处理。
 - 边缘服务器与客户端之间采用C/S体系结构，边缘服务器之间采用P2P体系结构，以支持协同处理。

1.4.3 混合型体系结构

- 边缘计算、雾计算
 - 在接近于现场应用端提供的计算
 - 实现物联网、智能制造等



百度百科, <https://baike.baidu.com/>



小结

- 分布式系统的构成
- 分布式系统的设计目标
 - 透明性，开放性，可伸缩性，可靠性、性价比
- 分布式系统的三种应用类型
 - 科学计算、信息服务、数据密集型计算
- 软件体系结构是分布式系统的逻辑结构
 - 建立分布式系统的基本原理
 - 五种典型模式：分层型、对象模式、数据中心模式、事件驱动模式、资源型模式
- 分布式系统实现的三种典型体系结构
 - 中心化、去中心化、混合式

第一章 完