异地多活IDC机房架构

荔枝FM架构师 刘耀华

内容

- 问题与需求
- 系统理论
- 系统调研
- 架构设计
- 最佳实践
- Q&A

问题与需求分析

单一机房架构问题

数据安全	• 机房不可用时数据无法访问
业务可用性	• 机房不可用时业务停止
用户响应	• 因网络连通性，不同地域的用户的请求响应延迟不同

多机房

- 备份保留重要的用户与系统数据，以保证数据安全
- 提高系统可用性，当某一机房发生故障时能尽快地切换都另一机房继续提供服务。
- 提高系统访问性能，按用户地域来合理分配距离最近，访问最佳的机房。

系统理论

CAP理论

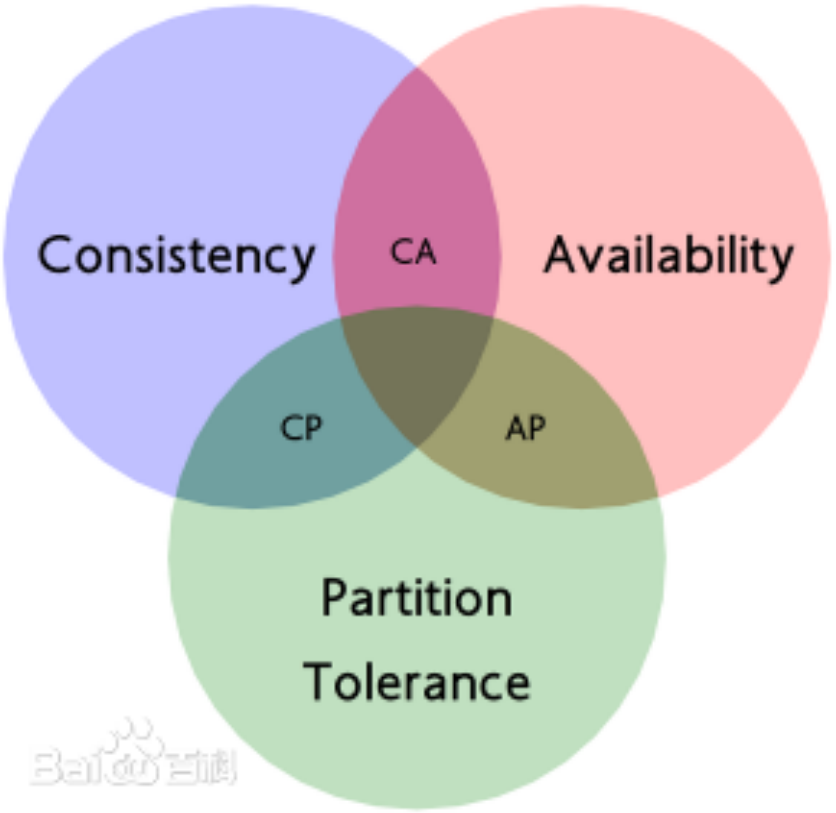
——University of California, Berkeley computer scientist Eric Brewer,

- **一致性(Consistency)**
  所有节点都能访问同一份最新的数据副本

- **可用性(Availability)**
  每个请求都能接收到一个响应，无论响应成功或失败，而不应该是网络超时、连接断开等非服务程序答复。

- **分区容忍性(Partition tolerance)**
  除了整个网络的故障外，其他的故障（集）都不能导致整个系统无法正确响应。

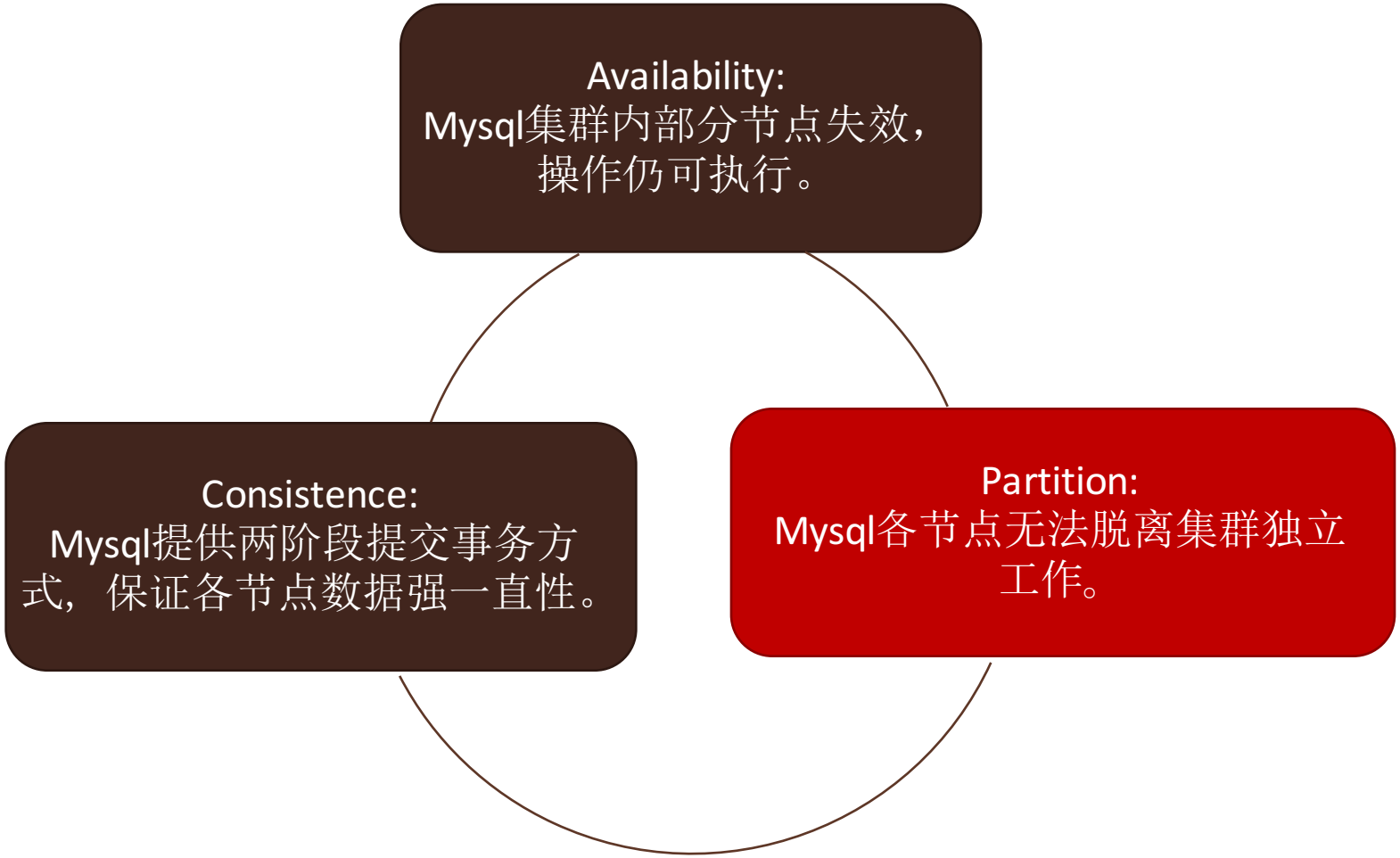
CAP三选二原则



CAP原理指的是，这三个要素最多只能同时实现两点，不可能三者兼顾。

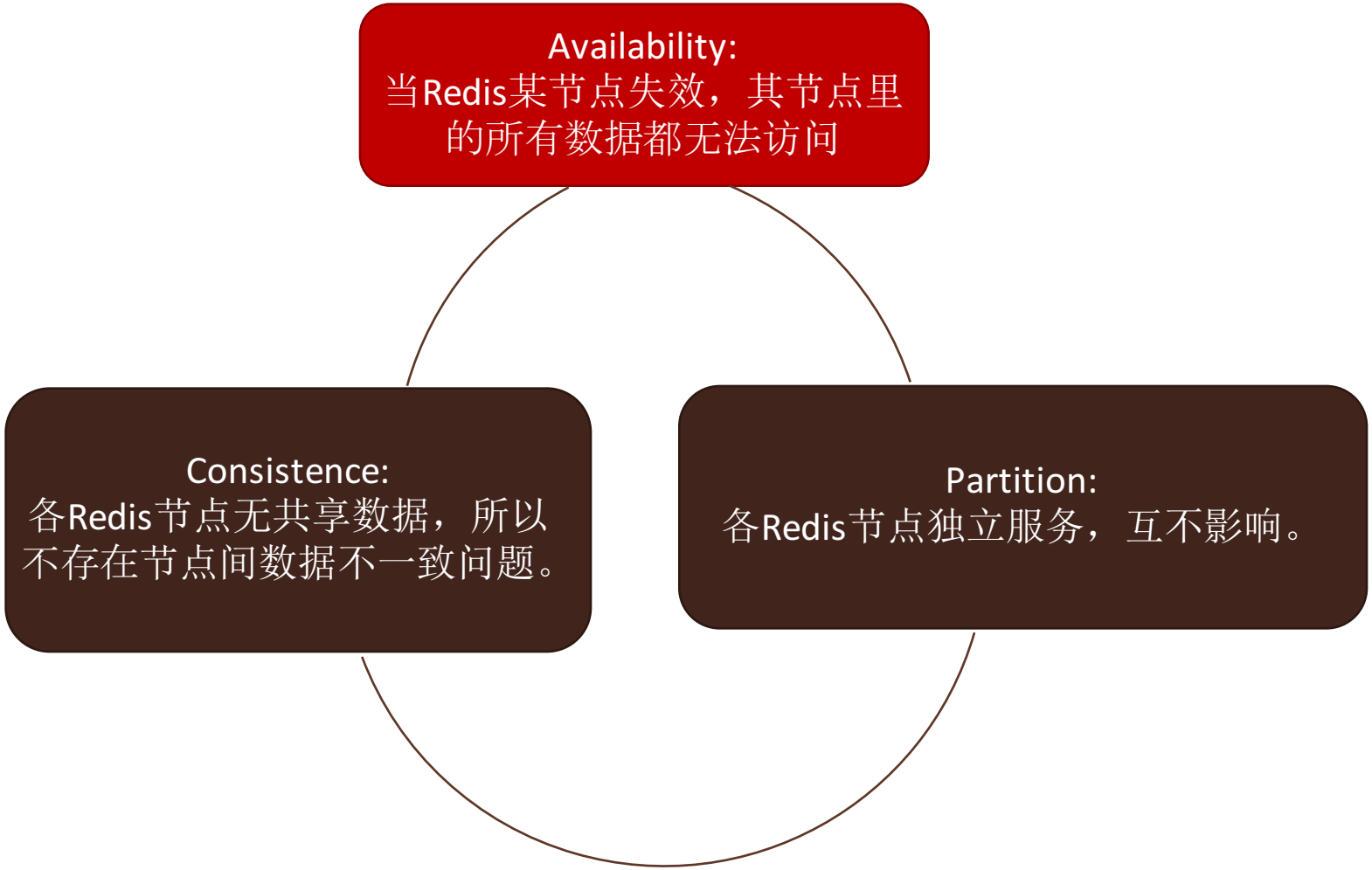
AC模型：（可用性+强一致性）-分区容忍

- MySql Cluster集群



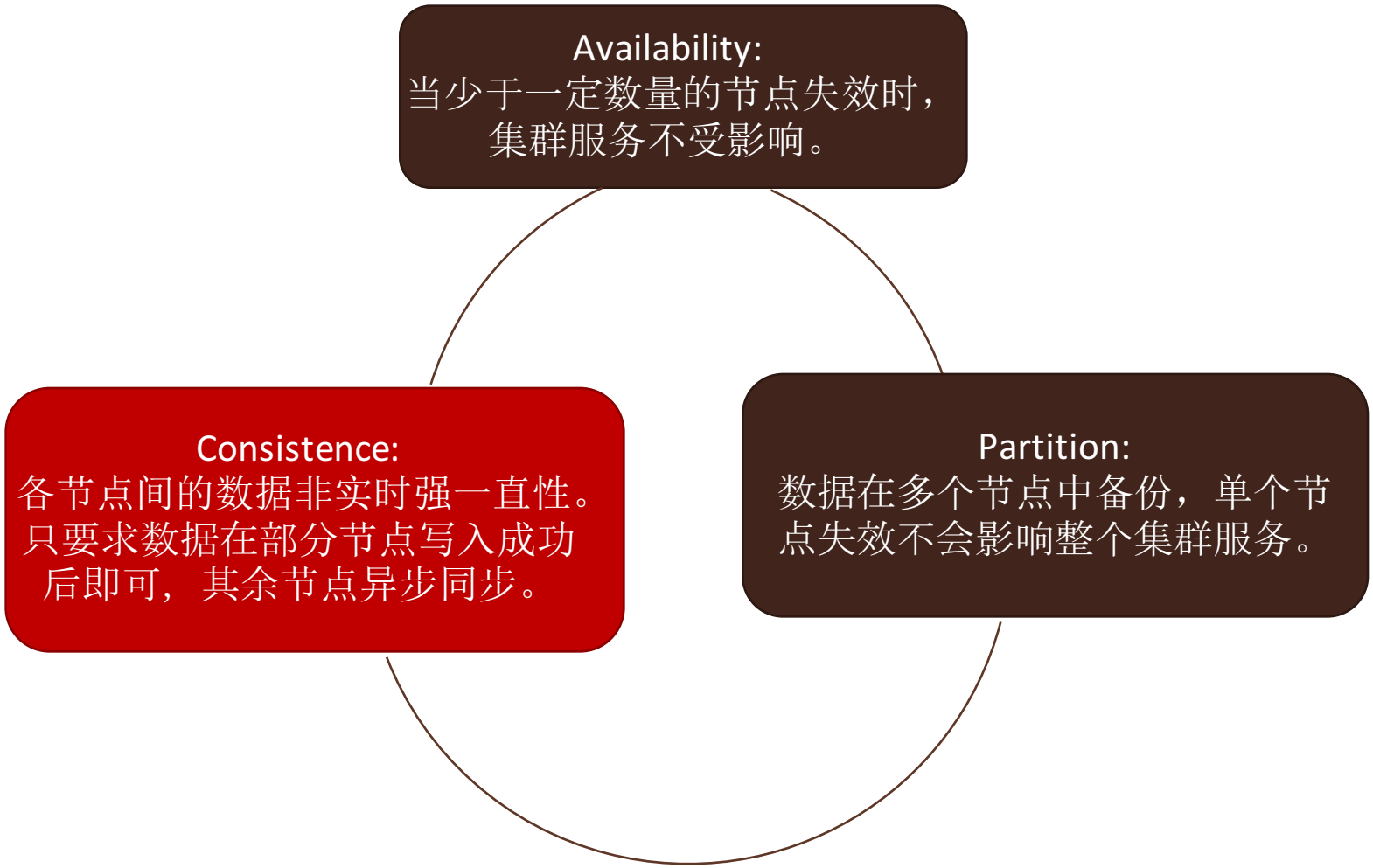
CP模型：（一致性+分区容忍）-可用性

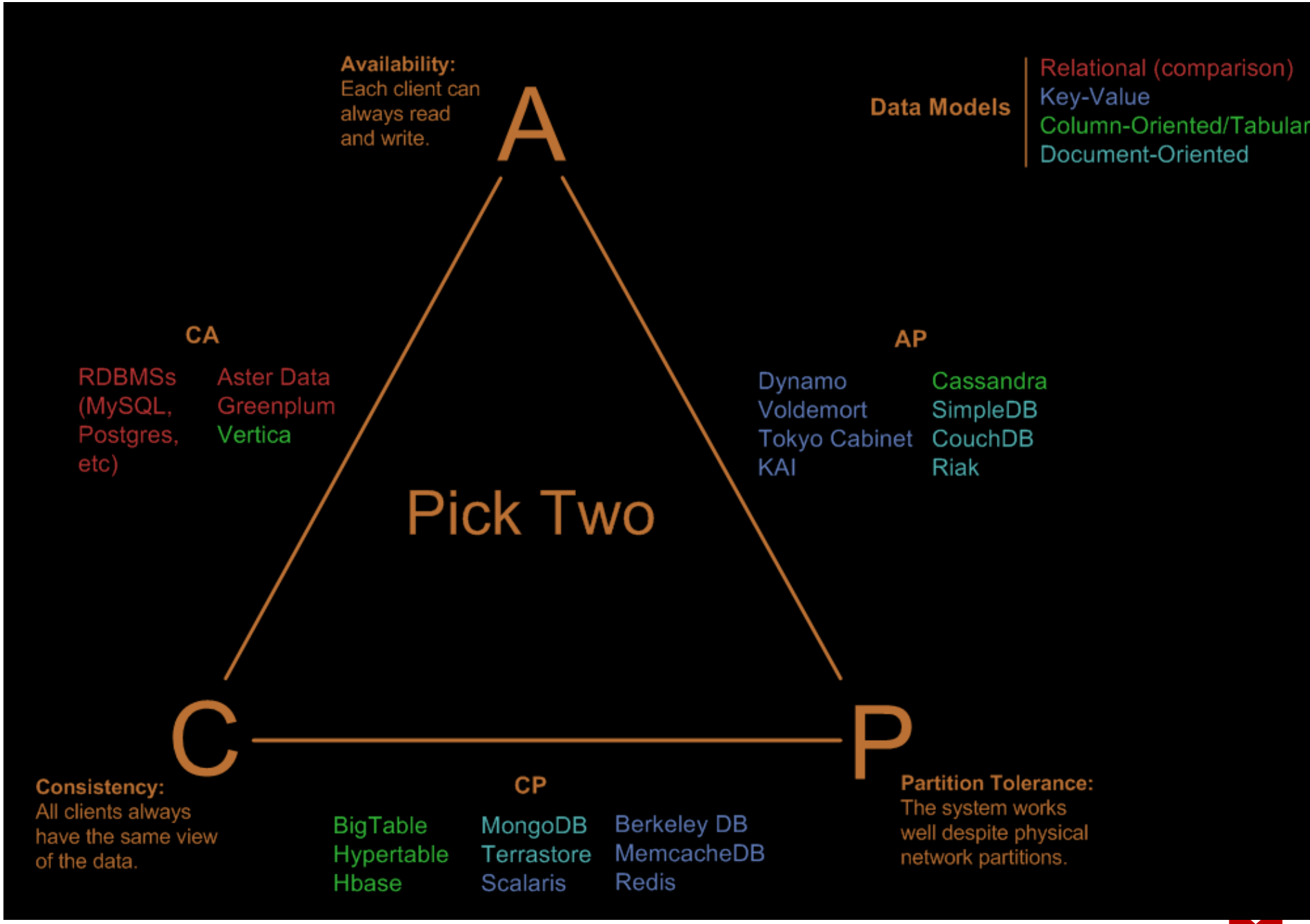
- Redis 客户端哈希/Twemproxy集群



AP模型：（可用性+分区容忍）-强一致性

- Cassandra集群




Availability:
Each client can always read and write.
A
Data Models
Relational (comparison)
Key-Value
Column-Oriented/Tabular
Document-Oriented
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Pick Two
C
P
Consistency:
All clients always have the same view of the data.
CP
BigTable
Hypertable
Hbase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
Partition Tolerance:
The system works well despite physical network partitions.

互联网行业模型

- 不同的业务类型要求不同的CAP模型：
  - CA适用于支付、交易、票务等业务要求数据强一致性的行业；（宁愿业务不可用，也不能出现脏数据或数据错乱）
  - 而互联网则对严格一致性要求不太高，但对业务可用性要求较高。因此一般都采用高可用+分区容忍+弱一致性架构。（+A+P-C）,并衍生出BASE模型。

BASE理论

- eBay的架构师Dan Pritchett源于对大规模分布式系统的实践总结，在ACM上发表文章提出BASE理论。

- BASE是指

  基本可用（Basically Available）、

  软状态（Soft State）、

  最终一致性（Eventual Consistency）。

BASE模型

基本可用（Basically Available）

基本可用是指分布式系统在出现故障的时候，允许损失部分可用性，即保证核心可用。

服务降级

BASE模型

软状态（ Soft State ）

软状态是指允许系统存在中间状态，而该中间状态不会影响系统整体可用性——临时数据不一致。

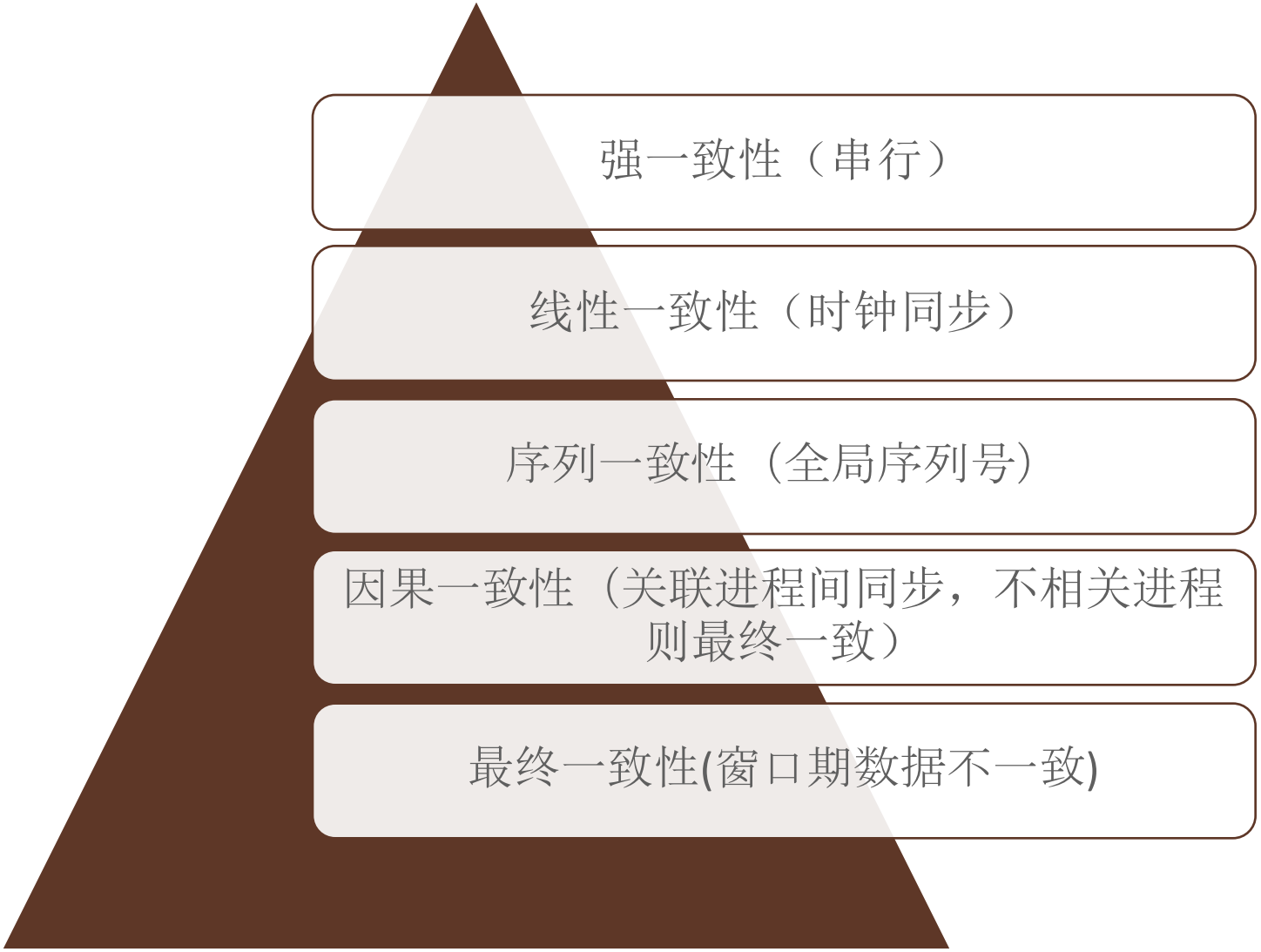
全局锁v.s数据多版本

BASE模型

最终一致性（ Eventual Consistency ）

- 最终一致性，就是不保证在任意时刻任意节点上的同一份数据都是相同的。
- 但是随着时间的迁移，不同节点上的同一份数据总是在向趋同的方向变化。

数据一致性模型

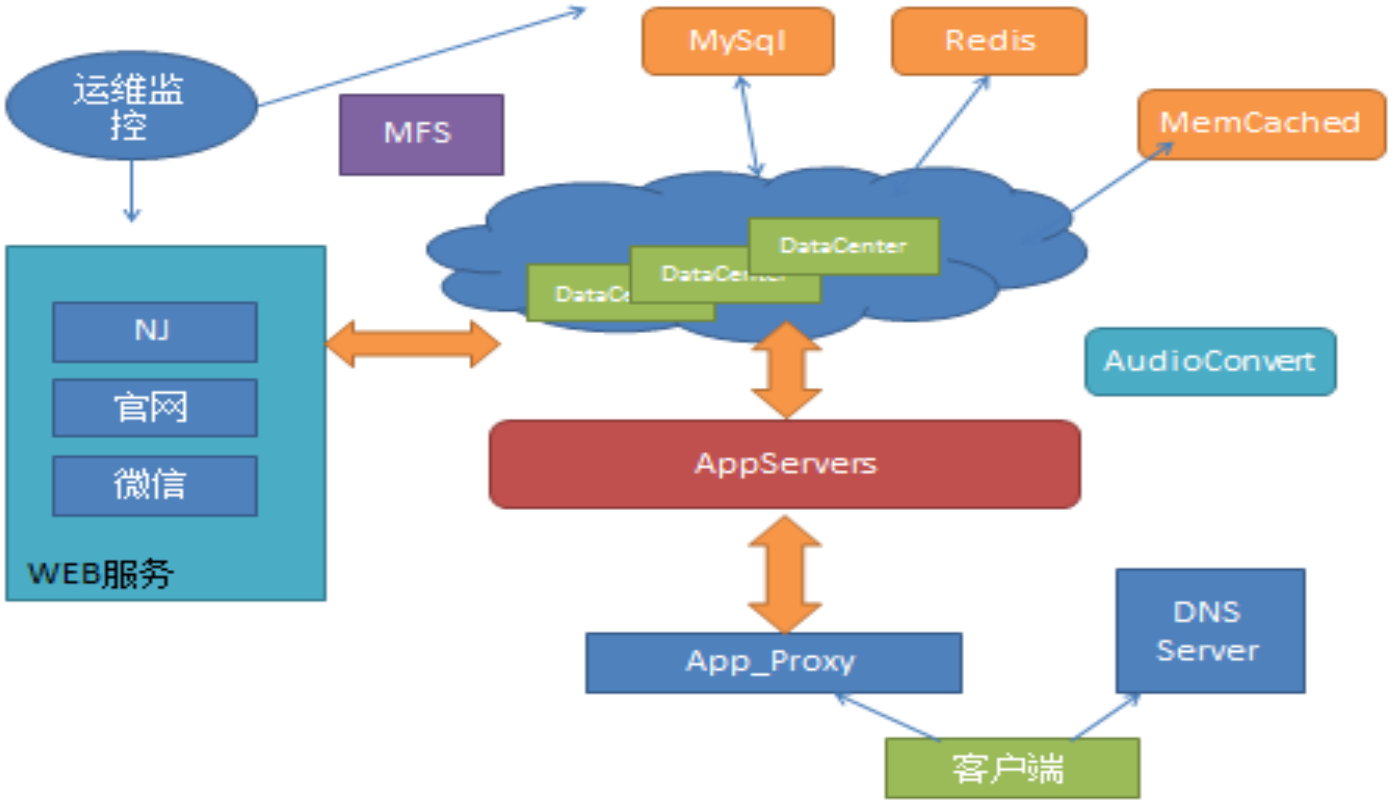

强一致性（串行）
线性一致性（时钟同步）
序列一致性（全局序列号）
因果一致性（关联进程间同步，不相关进程则最终一致）
最终一致性(窗口期数据不一致)

最终一致性

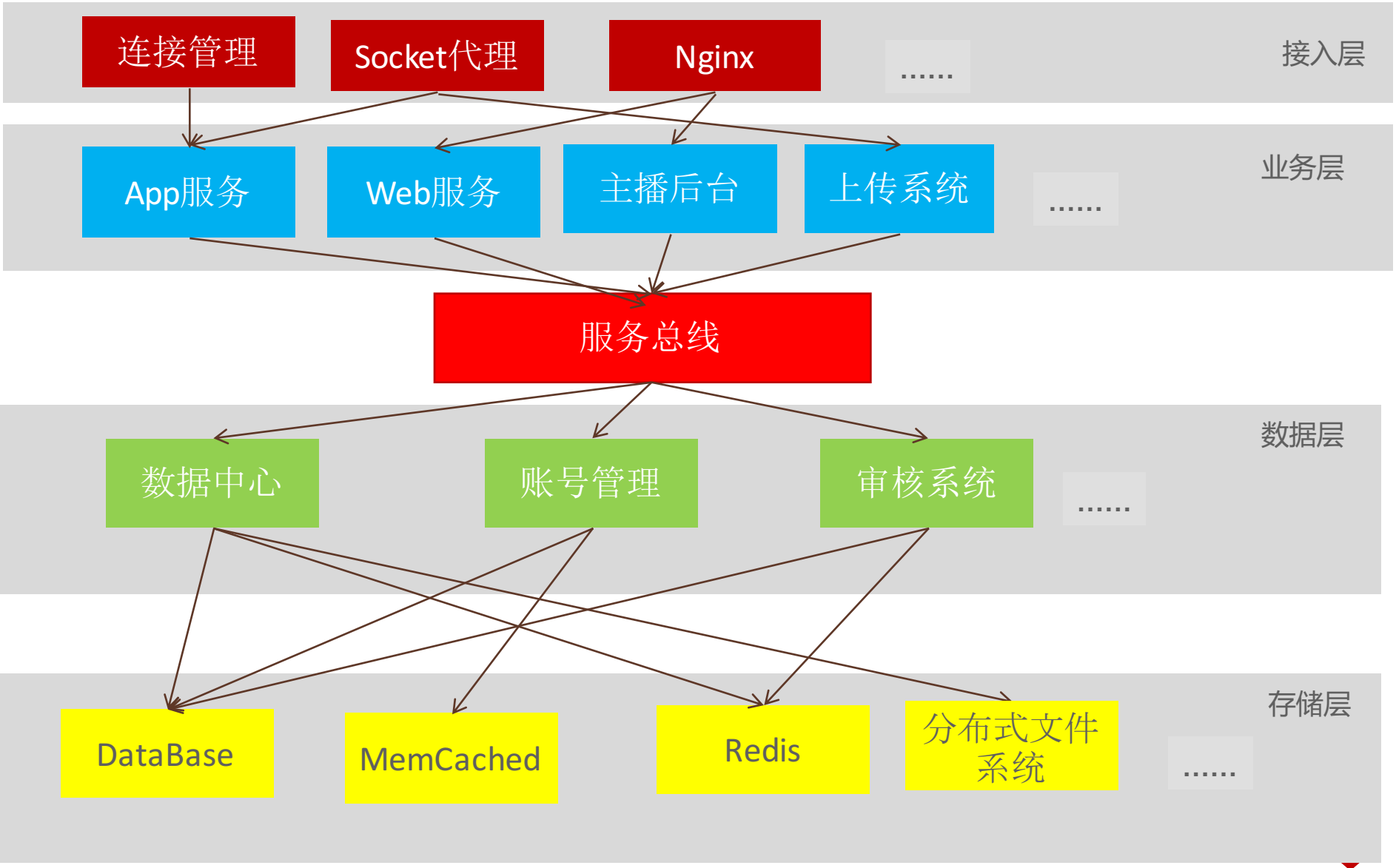
- 根据互联网业务特性，最后选择最终一致性。

系统业务调研

服务器系统


运维监控
MFS
MySql
Redis
MemCached
DataCenter
NJ
官网
微信
WEB服务
AudioConvert
AppServers
App_Proxy
DNS Server
客户端

当前服务端系统架构


连接管理
Socket代理
Nginx
.....
接入层
App服务
Web服务
主播后台
上传系统
.....
业务层
服务总线
数据中心
账号管理
审核系统
.....
数据层
DataBase
MemCached
Redis
分布式文件系统
.....
存储层

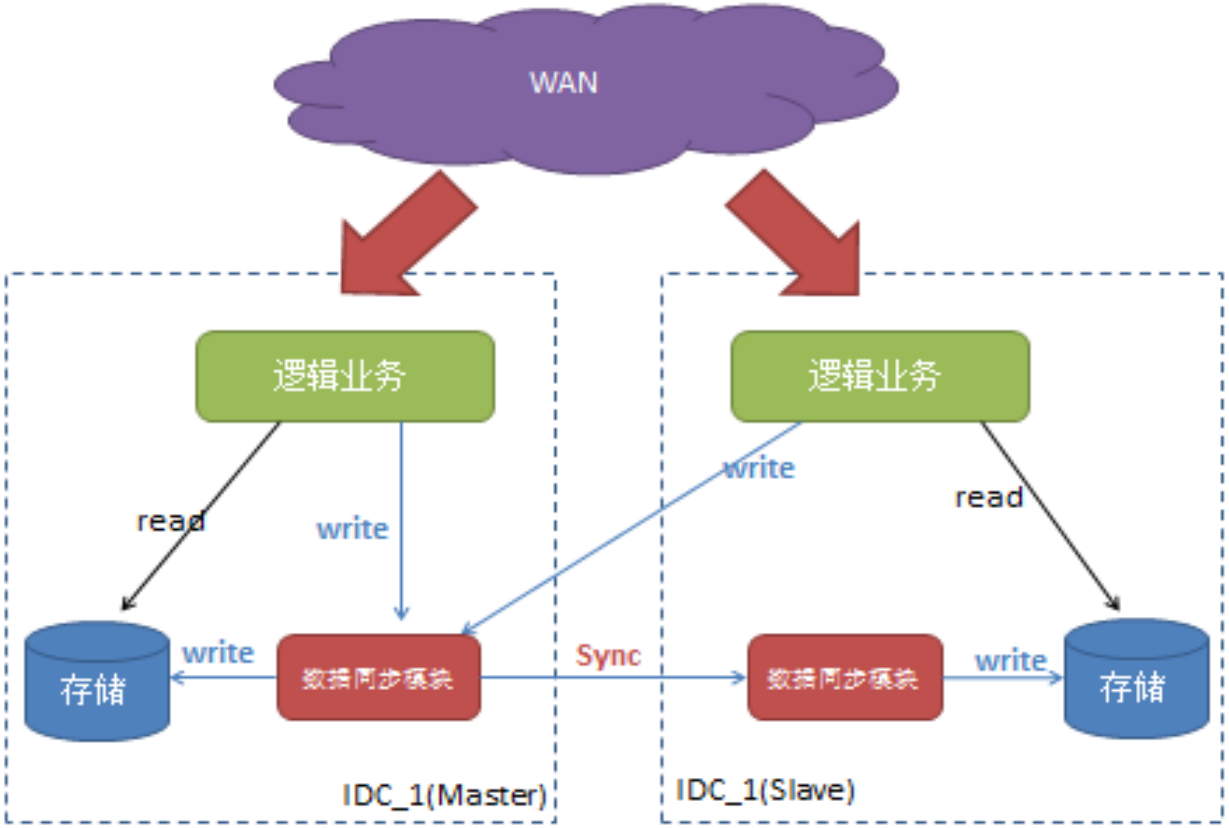
业务数据分析

类型	数据量	优先级	主传输线路	后备传输线路
资源文件	大	低	公网	机房专线
数据	小	高	机房专线	公网

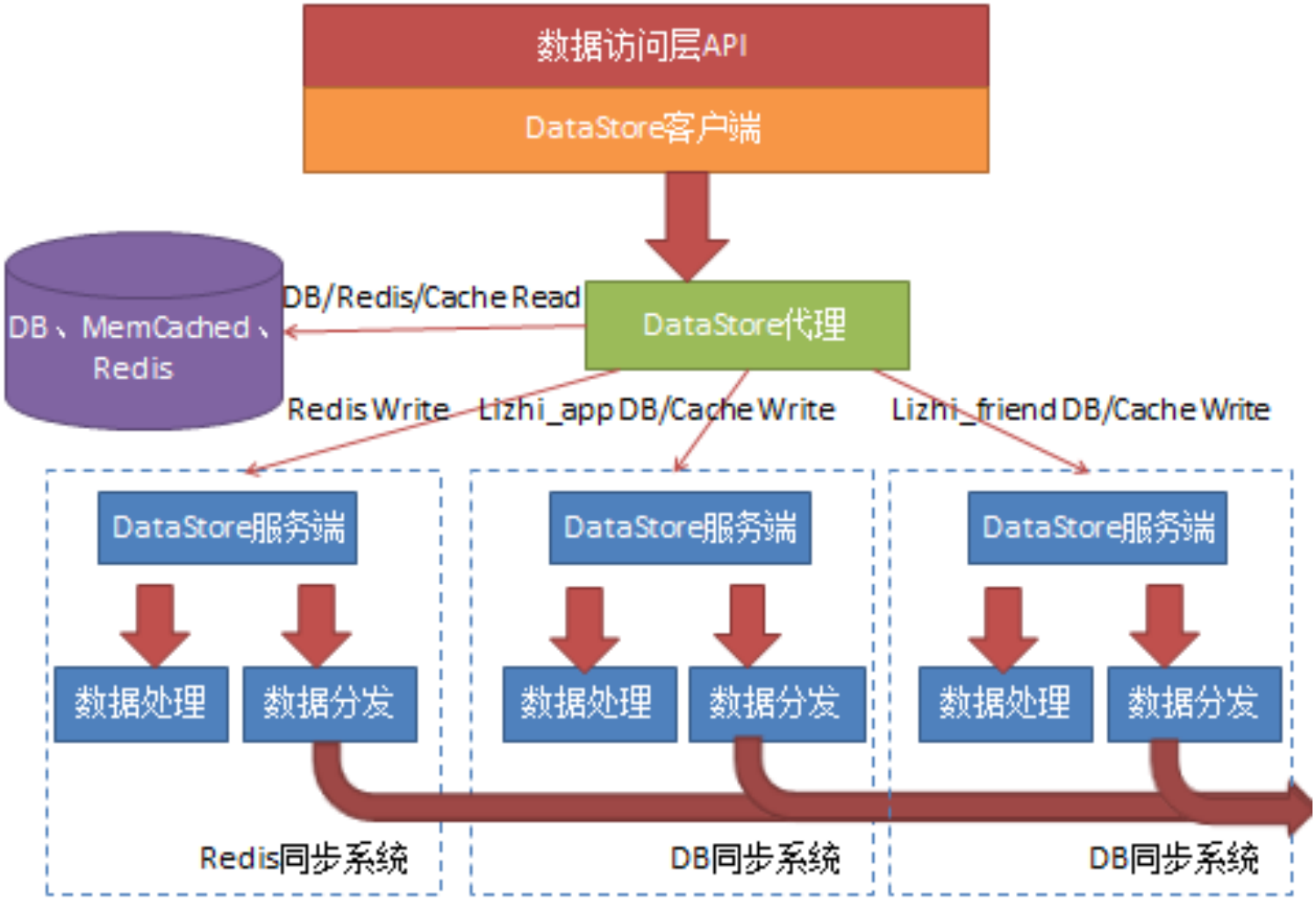
架构设计

硬件物理架构

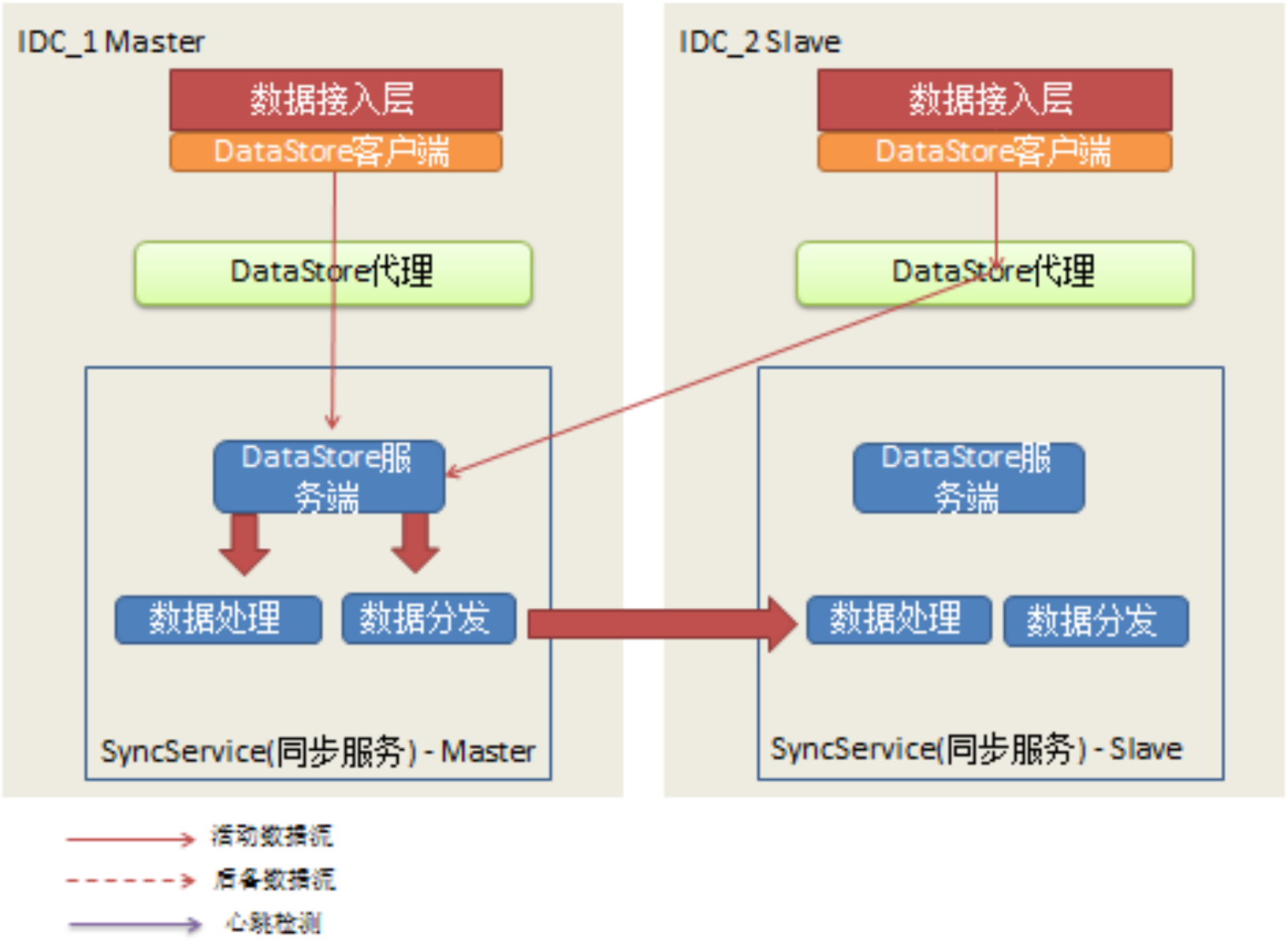
系统架构


WAN
逻辑业务
逻辑业务
read
write
write
read
write
存储
数据同步模块
Sync
数据同步模块
write
存储
IDC_1(Master)
IDC_1(Slave)

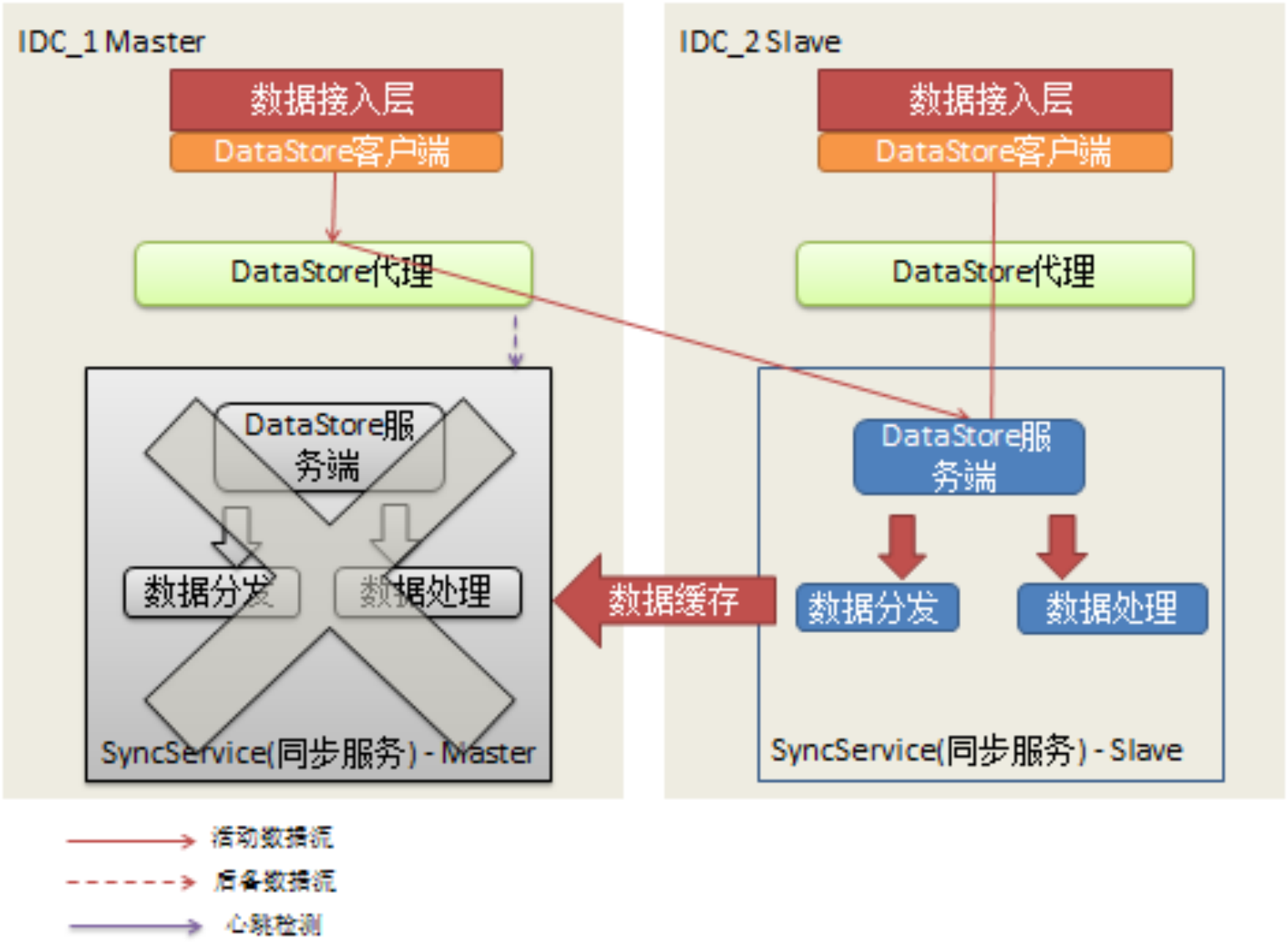
数据存储（DataStore）系统



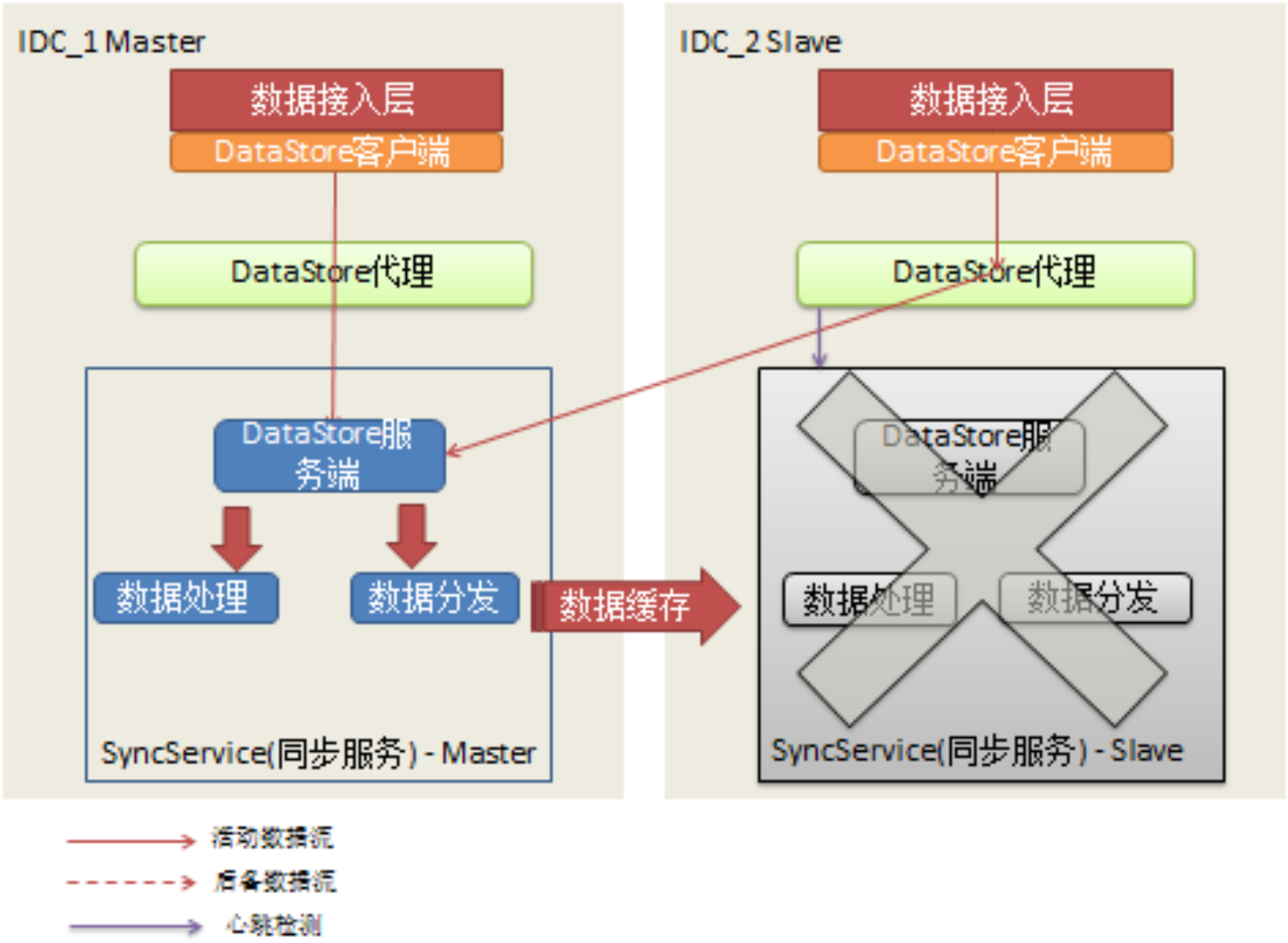
数据存储系统DataStore



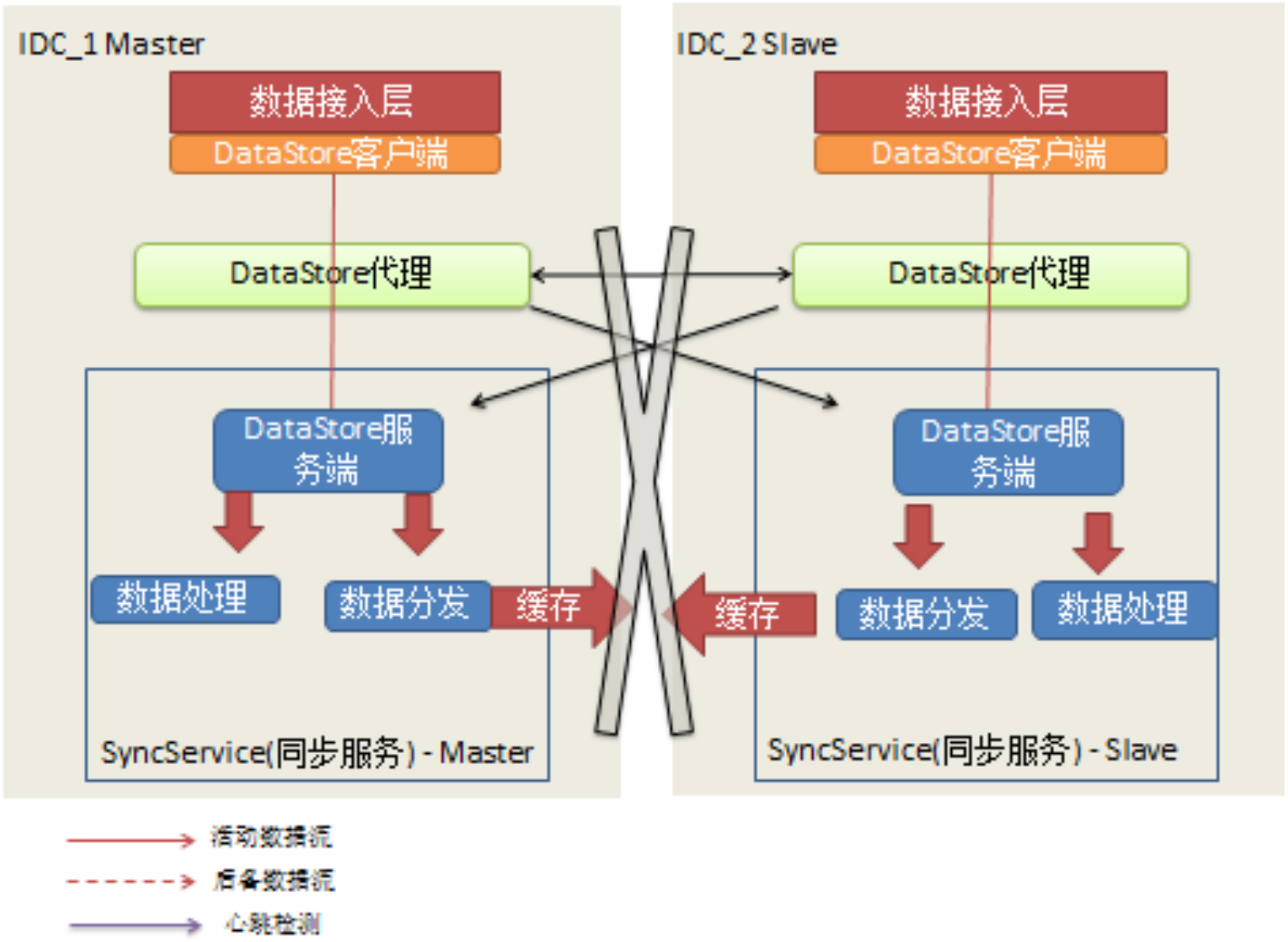
主机房DS服务器失效


IDC_1 Master
数据接入层
DataStore客户端
DataStore代理
DataStore服务端
数据分发
数据处理
SyncService(同步服务) - Master
数据缓存
IDC_2 Slave
数据接入层
DataStore客户端
DataStore代理
DataStore服务端
数据分发
数据处理
SyncService(同步服务) - Slave
活动数据流
后备数据流
心跳检测

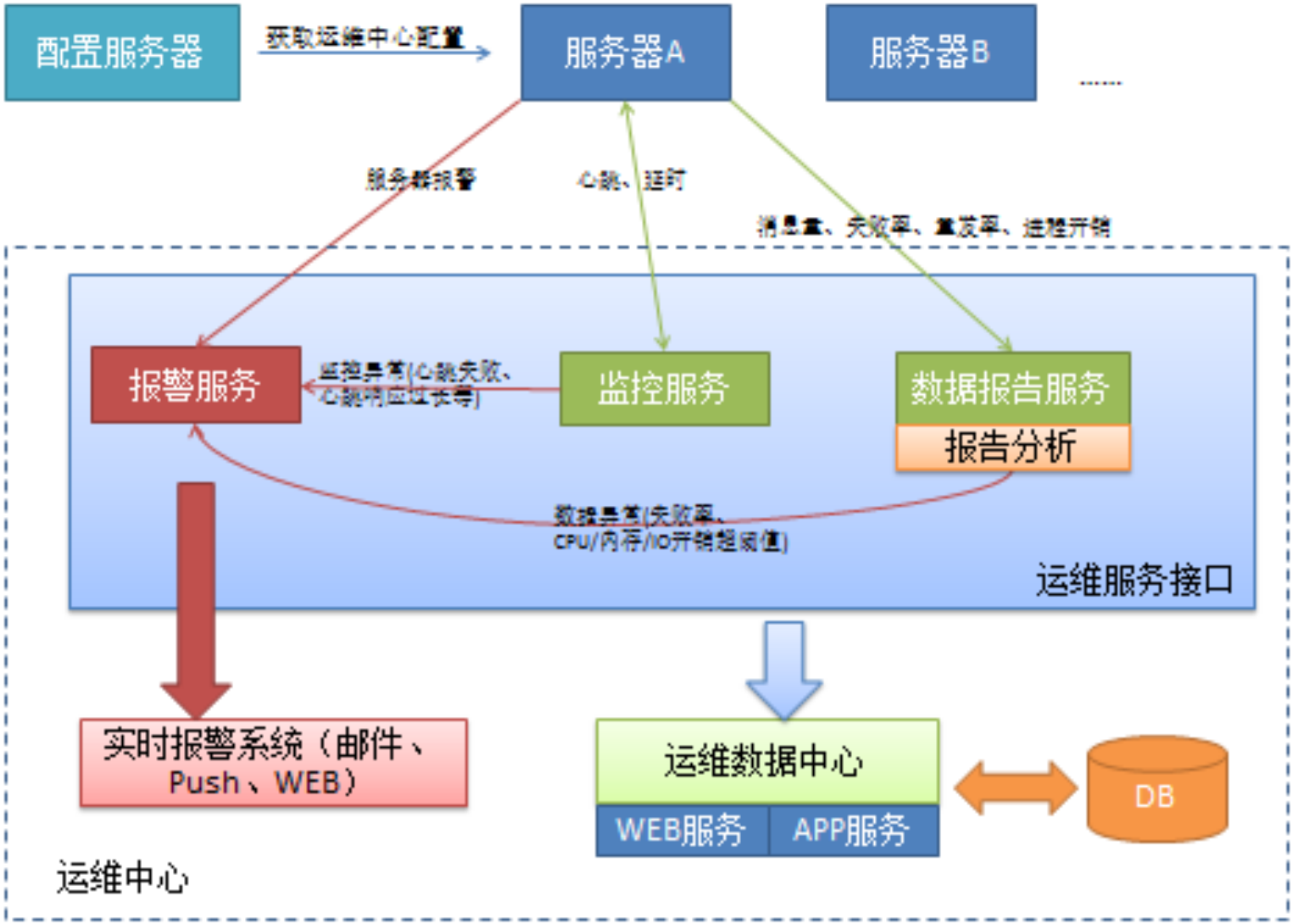
从机房DS服务器失效


IDC_1 Master
数据接入层
DataStore客户端
DataStore代理
DataStore服务端
数据处理
数据分发
SyncService(同步服务) - Master
数据缓存
IDC_2 Slave
数据接入层
DataStore客户端
DataStore代理
DataStore服务端
数据处理
数据分发
SyncService(同步服务) - Slave
活动数据流
后备数据流
心跳检测

脑裂问题


IDC_1 Master
数据接入层
DataStore客户端
DataStore代理
DataStore服务端
数据处理
数据分发
缓存
SyncService(同步服务) - Master
IDC_2 Slave
数据接入层
DataStore客户端
DataStore代理
DataStore服务端
缓存
数据分发
数据处理
SyncService(同步服务) - Slave
活动数据流
后备数据流
心跳检测

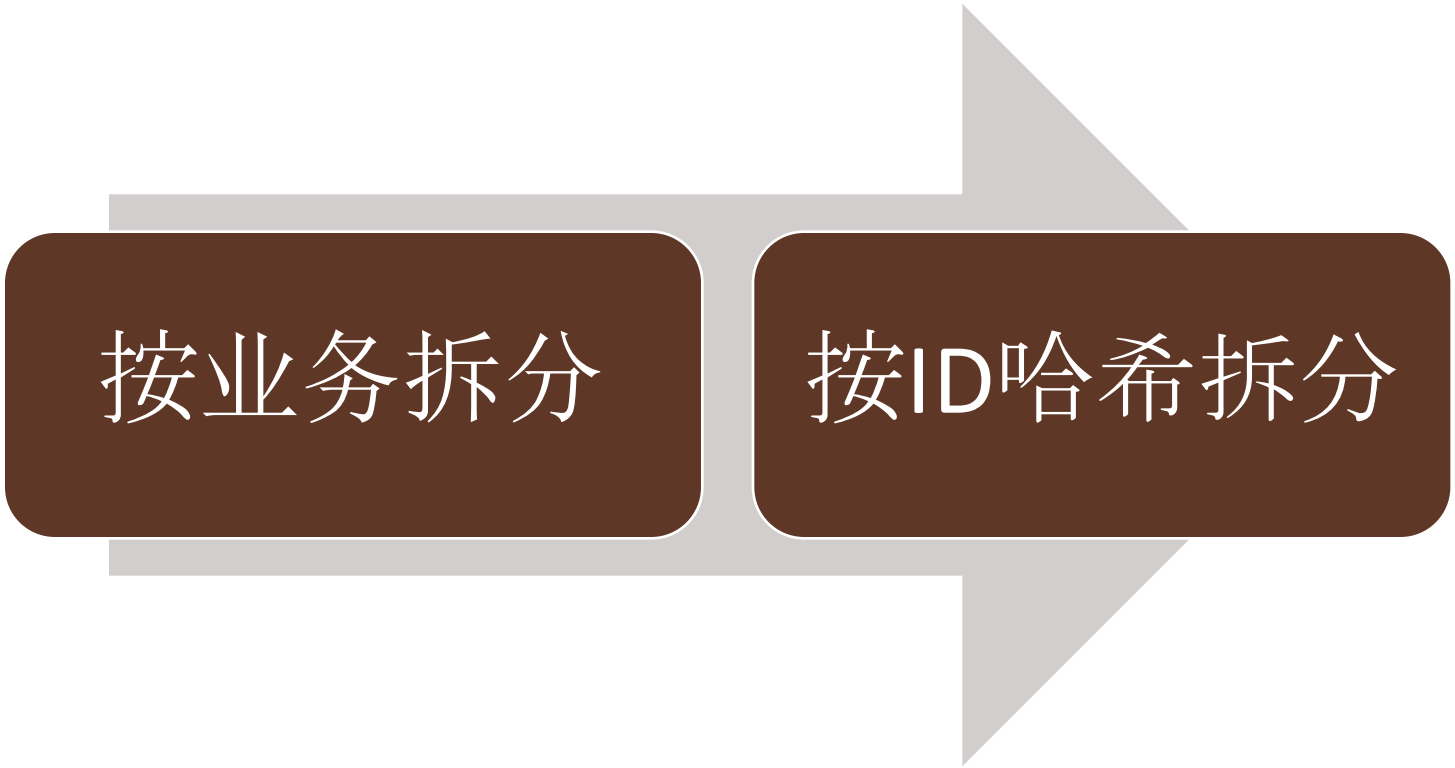
运维与监控


配置服务器
获取运维中心配置
服务器A
服务器B
.....
服务器报警
心跳、延时
消息量、失败率、重发率、连接开销
报警服务
监控异常(心跳失败、
心跳响应过长等)
监控服务
数据报告服务
报告分析
数据异常(失败率、
CPU/内存/IO开销超阈值)
运维服务接口
实时报警系统（邮件、
Push、WEB）
运维数据中心
WEB服务
APP服务
DB
运维中心

最佳实践

数据切分

- 先纵再横
  - 先按业务进行垂直拆分
  - 再按ID哈希进行水平拆分



数据只改不删

- 记录在数据库中删除后很难恢复
- 尽量做删除标记，而不是物理删除

善用异步处理

- 异步响应能提高系统使用效率，避免线程挂起死等。
- 异步会增加编程复杂度，提供简单易用的异步接口
- 所有外部的IO操作都有可能因操作缓慢而阻塞：e.g.写硬盘
- 复杂逻辑处理用异步

提高程序可测试性、可分析性

- 测试驱动，对每个单元写测试用例
- 提交修改后执行回归测试
- 运行时日志统一收集
- 统一监控

程序逻辑对一致性的容忍

- 由于架构为最终一致性，程序代码中需容忍一段时间的数据不一致窗口期。
- 如insert之后马上select刚插入的记录，由于读写分离，有可能是无法读取刚生成的记录。

幂等操作

- 由于网络、机器等不稳定因素，请求有可能会失败。
- 分布式系统的三态：成功、失败、超时。
- 其中超时最难处理，

  有可能请求包丢了，是服务端没收到；

  也有可能是服务端已收到并处理了请求，但返回包丢了。

如果操作不幂等，客户端收到超时后重试，会导致失败或数据冲突。

例如对同一id的记录应采用replace，而不是insert

推 v.s 拉

- 直接推送数据比拉取数据请求减少一个消息传递。
- 推送需要区分接收与执行线程
- 记录好接收id，并在下次重连时重新请求同步

做好监控

- 系统资源（CPU、内存、磁盘、网络、文件描述符、IOW）
- 进程监控（错误与警告日志、内存泄露、命令响应速度）
- 存储系统监控（操作量、并发量、慢查询）
- 服务监控（消息拨测、网络拨测、网络连通性）
- 实时报警（邮件、IM、短信）
- 报告（日报、分时曲线、峰值、High/Low watermark）

架构要易于随时扩展或缩减

- 当遇到性能瓶颈时，能够方便地迅速扩容调整。
- 对上层业务透明
- 支持热更新（代码、配置、架构）
- 提供一键式操作预案。
- 操作后自动测试验证

架构对程序开发友好

- 接口API简单明了
- 与业务解耦
- 减少依赖，或使用依赖管理工具（Maven）
- 配置简单，约定优于配置（COC）原则（Convention Over Configuration）

Q&A