北京航空航天大學
BEIHANG UNIVERSITY

# 数据管理技术

北京航空航天大学

周号益

2024年

# 第十二章 NoSQL和云数据库

1 分布式数据库

2 NoSQL简介

3 NoSQL的技术特点

4 云数据库概述

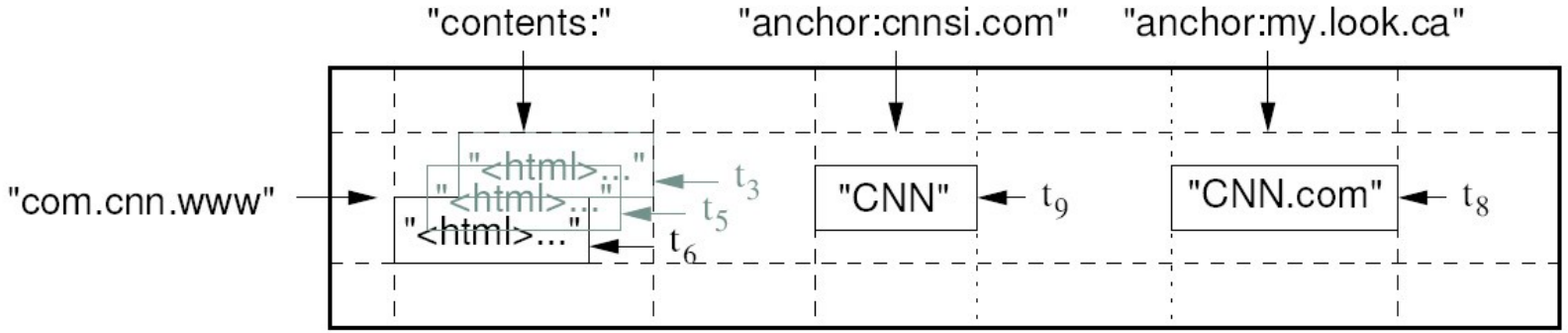
# 分布式数据库的起源——BigTable

BigTable起初用于解决Google的大规模网页搜索问题

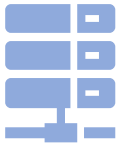
- BigTable是一个分布式存储系统

## 网页搜索任务

- **建立互联网索引**
  1. 爬虫持续抓取新页面，并每页一行地存储到BigTable里
  2. 基于MapReduce计算，生成索引，为网络搜索应用做准备
- **搜索互联网**
  1. 用户发起网络搜索请求
  2. 网络搜索应用查询索引，从BigTable得到网页地址后提交给用户



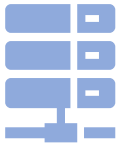

网页在BigTable中的存储样例



# 分布式数据库的起源——BigTable

- BigTable是一个分布式存储系统
- **底层数据存储：**使用谷歌分布式文件系统GFS
- **处理海量数据：**利用谷歌提出的MapReduce分布式并行计算模型
- **协同服务管理：**采用Chubby所提供的能力
- **可扩展能力：**数据到PB级、机器达上千台
- **特点：**广泛应用性、可扩展性、高性能和高可用性等

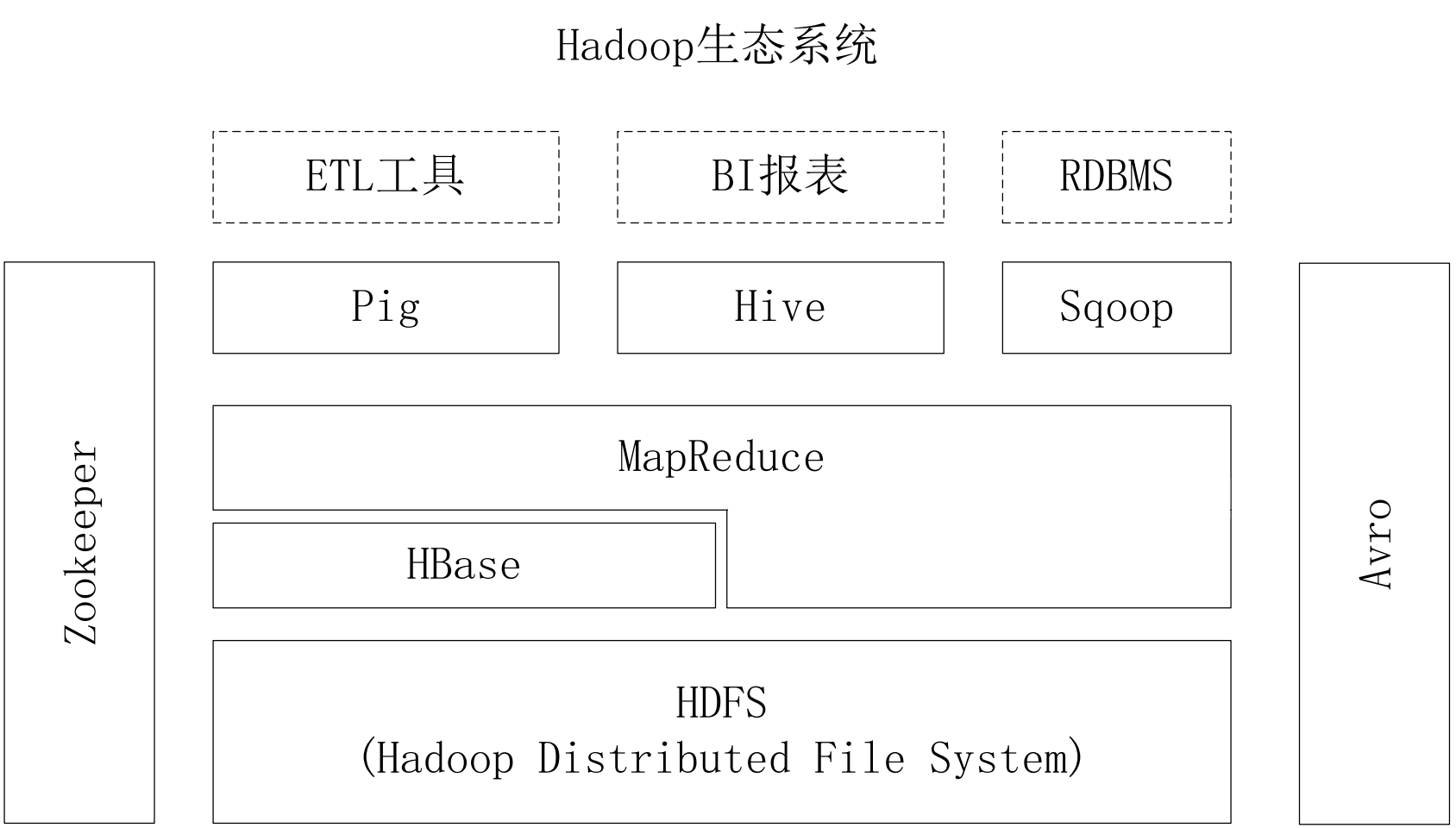
- 谷歌的许多项目都存储在BigTable中
  - 搜索、地图、财经、打印、社交网站Orkut
  - 视频共享网站YouTube、博客网站Blogger
  - ……



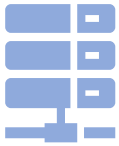
# HBase概述

- 名字来源是Hadoop database，即hadoop数据库
- 是BigTable的开源实现
- 是一个**高可靠**、**高性能**、**面向列**、**可伸缩**的分布式数据库
- 主要用来存储**非结构化**和**半结构化**的松散数据
- 目标是处理非常庞大的表
  - 可以通过水平扩展的方式，利用廉价计算机集群处理由超过10亿行数据和数百万列元素组成的数据表

# HBase概述



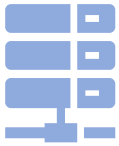

Hadoop生态系统中HBase与其他部分的关系



# HBase概述

HBase和BigTable的底层技术对应关系

| | **BigTable** | **HBase** |
|---|---|---|
| 文件存储系统 | GFS | HDFS |
| 海量数据处理 | MapReduce | Hadoop MapReduce |
| 协同服务管理 | Chubby | Zookeeper |



# HBase概述

关系数据库已经流行很多年，并且Hadoop已经有了HDFS和MapReduce，为什么需要HBase?

- 克服了HDFS无法满足大规模数据实时处理的缺陷
  - Hadoop可以很好地解决大规模数据的离线批量处理问题，但受限于Hadoop MapReduce编程框架的高延迟数据处理机制，使得HDFS面向批量访问模式，不是随机访问模式
- 克服了RDB对面大规模数据时系统扩展性和性能差问题
  - RDB采用主从、分库、分表模式也不能很好解决
  - 人工方式多，效率低下
- 克服了RDB难于处理非结构化与半结构化数据的缺陷
  - 传统RDB在数据结构变化时一般需要停机维护

# HBase与RDB的对比

| | RDB | HBase |
|---|---|---|
| **数据类型** | 采用关系模型，数据类型和存储方式丰富 | 简单的数据模型，数据存储为**未经解释的字符串** |
| **数据操作** | 丰富的操作，涉及复杂的多表连接 | **只有**简单的插入、查询、删除、清空 |
| **存储模式** | 基于行模式存储 | 基于**列存储**，每个列族都由几个文件保存，不同列族的文件是分离的 |
| **数据索引** | 针对不同列构建复杂的多个索引 | （原始设计）只有一个索引——行键 |
| **数据维护** | 记录中原来的旧值会被最新的当前值替换 | **不会删除数据旧版本**，而是生成一个新的版本 |
| **可伸缩性** | 很难实现横向扩展，纵向扩展的空间也比较有限 | **性能伸缩性好：**能轻易地通过在集群中增加硬件数量来实现 |

# HBase访问接口

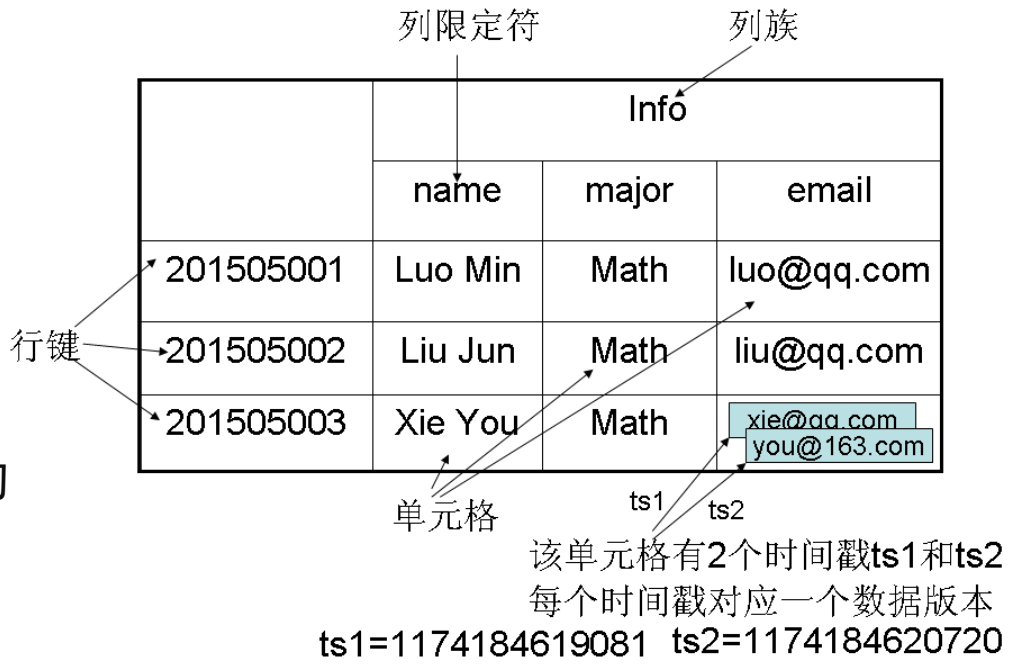
| 类型 | 特点 | 场合 |
| --- | --- | --- |
| **Native Java API** | 最常规和高效的访问方式 | 适合Hadoop MapReduce作业并行批处理HBase表数据 |
| **HBase Shell** | HBase的命令行工具，最简单的接口 | 适合HBase管理使用 |
| **Thrift Gateway** | 利用Thrift序列化技术，支持C++、PHP、Python等多种语言 | 适合其他异构系统在线访问HBase表数据 |
| **REST Gateway** | 解除了语言限制 | 支持REST风格的Http API访问HBase |
| **Pig** | 使用Pig Latin流式编程语言来处理HBase中的数据 | 适合做数据统计 |
| **Hive** | 简单 | 当需要以类似SQL语言方式来访问HBase的时候 |

# HBase数据模型

- HBase是一个稀疏、多维度、持久化存储的映射表
  - ✓这张表的索引是**行键**、**列族**、**列限定符**和**时间戳**
- 每个值是一个未经解释的字符串，没有数据类型
- 用户在表中存储数据，每一行组成：可排序的行键+任意多的列
- 表在水平方向由一个或者多个**列族**组成，一个列族中可以包含任意多个列，同一个列族里面的数据存储在一起
- 列族支持动态扩展，可以很轻松地添加一个列族或列，无需预先定义列的数量以及类型
- 列均以字符串形式存储，用户需要自行进行数据类型转换
- HBase中执行更新操作时，不会删除数据旧的版本，而是生成一个新的版本，旧有的版本仍然保留
  - ✓HBase可以对允许保留的版本的数量进行设置
  - ✓客户端可以选择获取距离某个时间最近的版本或获取所有版本

# HBase数据模型相关概念

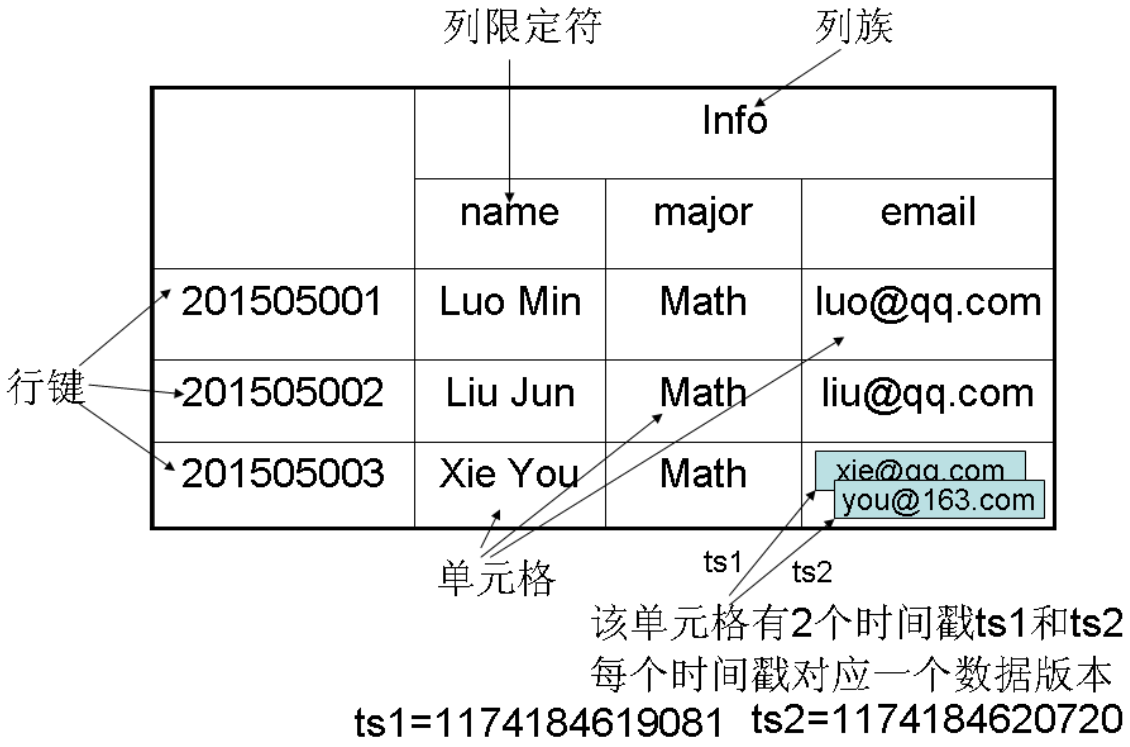
- **列**：Hbase的最基本的单位
- **列族**：列的集合，数据按列族分开存储
- **列限定符**：列族里的数据通过列限定符(列)来定位
- **行**：列族的集合，由行键来标识
- **表**：行的集合
- **单元格**：保存同一数据的多个版本,由行、列族和列限定符确定一个单元格；存储的数据没有数据类型，总被视为字节数组byte[]
- **时间戳**：标识单元格中同一数据的多个版本，这些版本用时间戳索引



# 数据坐标——定位表中的数据

- 由**四维坐标[行键，列族，列限定符(列)，时间戳]**确定一个单元格的具体值

| 键 | 值 |
| --- | --- |
| [“201505003”, “Info”, “email”, 1174184619081] | “xie@qq.com” |
| [“201505003”, “Info”, “email”, 1174184620720] | “you@163.com” |



# Hbase数据的概念视图

| 行键 | 时间戳 | 列族contents | 列族anchor |
| --- | --- | --- | --- |
| "com.cnn.www" | t5 | | anchor:cnnsi.com=”CNN” |
| | t4 | | anchor:my.look.ca="CNN.com" |
| | t3 | contents:html="<html>..." | |
| | t2 | contents:html="<html>..." | |
| | t1 | contents:html="<html>..." | |

列值：代表html列的具体值，即网页内容

列名：代表contents列族中的html列(...)

列族名：代表contents列族

时间戳：代表不同的时间点

行键：代表用于存储cnn网站信息，每个网站可以用一行来标识

# Hbase数据的概念视图

| 行键 | 时间戳 | 列族contents | 列族anchor |
|---|---|---|---|
| "com.cnn.www" | t5 | | anchor:cnnsi.com=”CNN” |
| | t4 | | anchor:my.look.ca="CNN.com" |
| | t3 | contents:html="<html>..." | |
| | t2 | contents:html="<html>..." | |
| | t1 | contents:html="<html>..." | |

- 从概念视图看，Hbase是个稀疏表，很多地方无值
- 但物理视图并不如此，是按列族为单位存储的

# HBase数据的物理视图

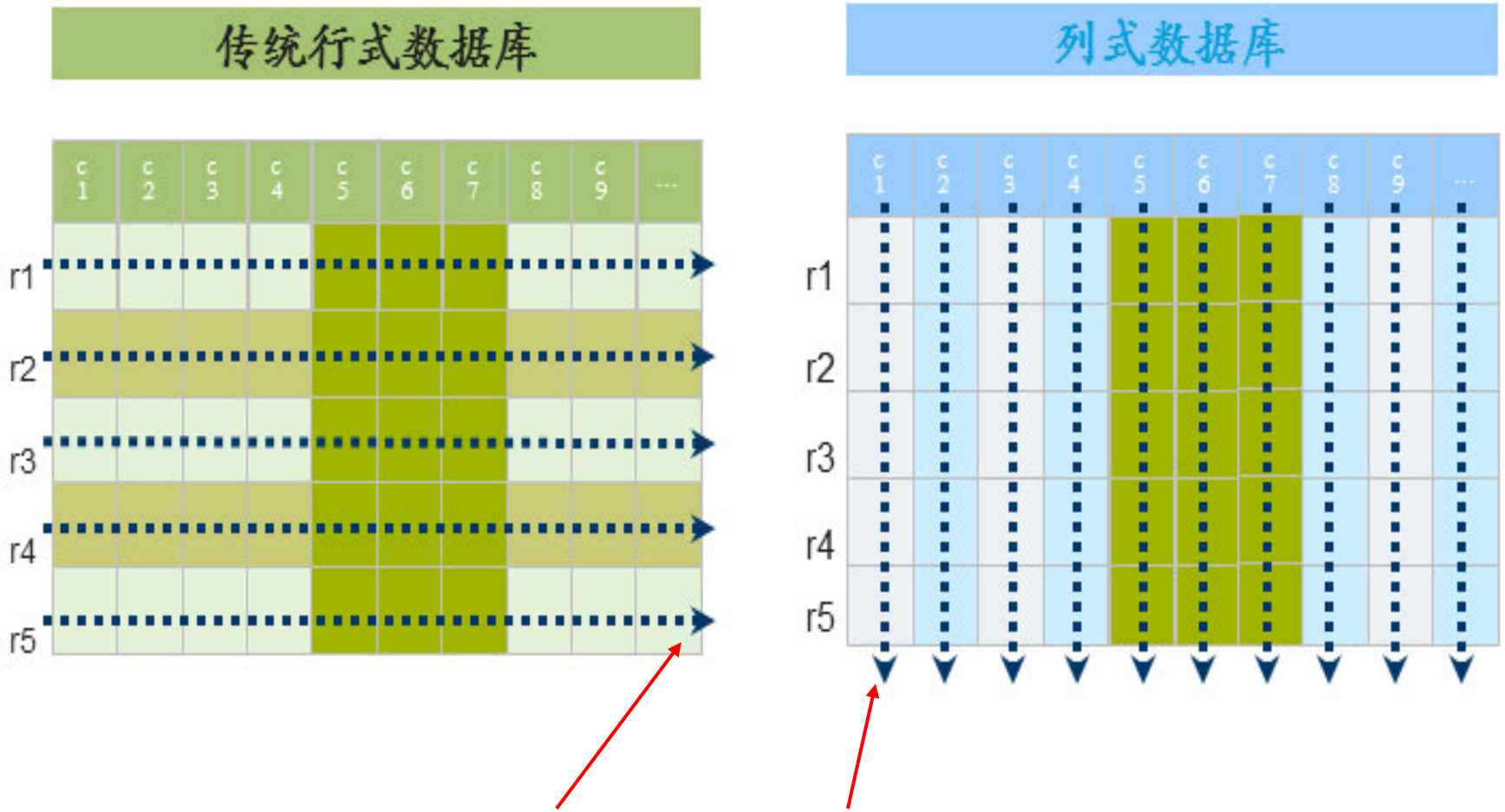
列族contents

| 行键 | 时间戳 | 列族contents |
| --- | --- | --- |
| "com.cnn.www" | t3 | contents:html="<html>..." |
| | t2 | contents:html="<html>..." |
| | t1 | contents:html="<html>..." |

列族anchor

| 行键 | 时间戳 | 列族anchor |
| --- | --- | --- |
| "com.cnn.www" | t5 | anchor:cnnsi.com=”CNN” |
| | t4 | anchor:my.look.ca="CNN.com" |

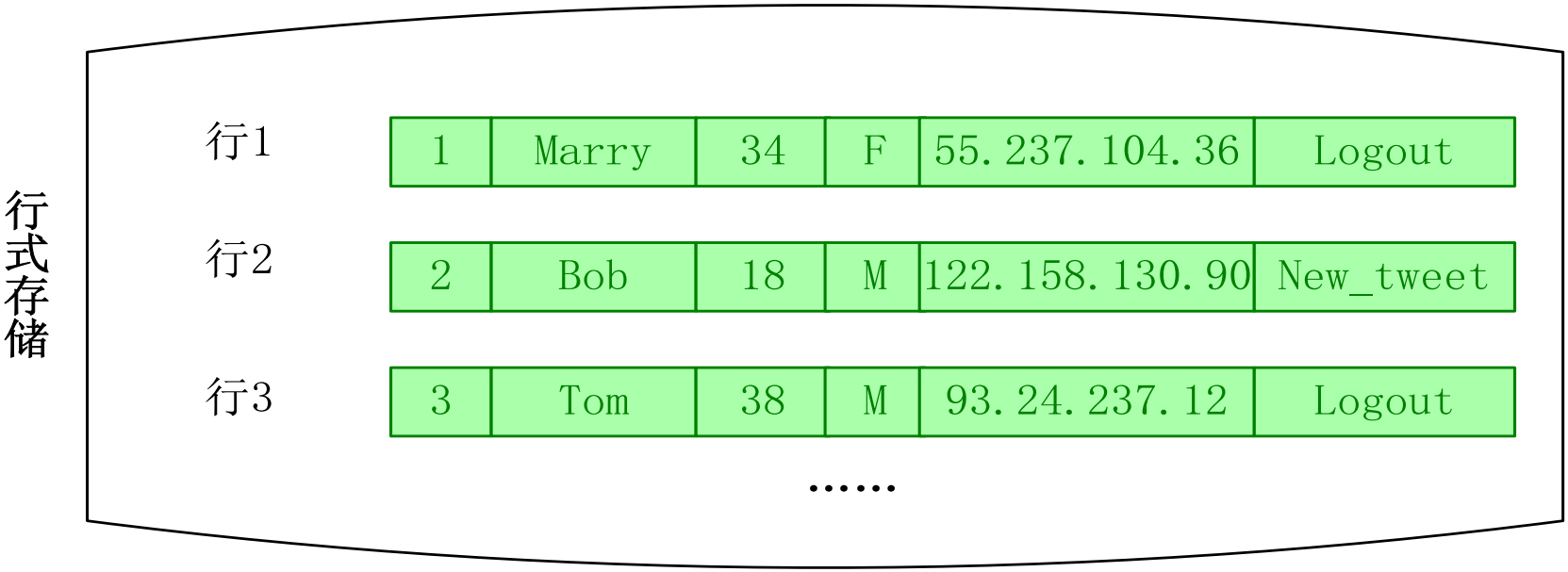
# HBase面向列的存储




行式数据库和列式数据库示意图

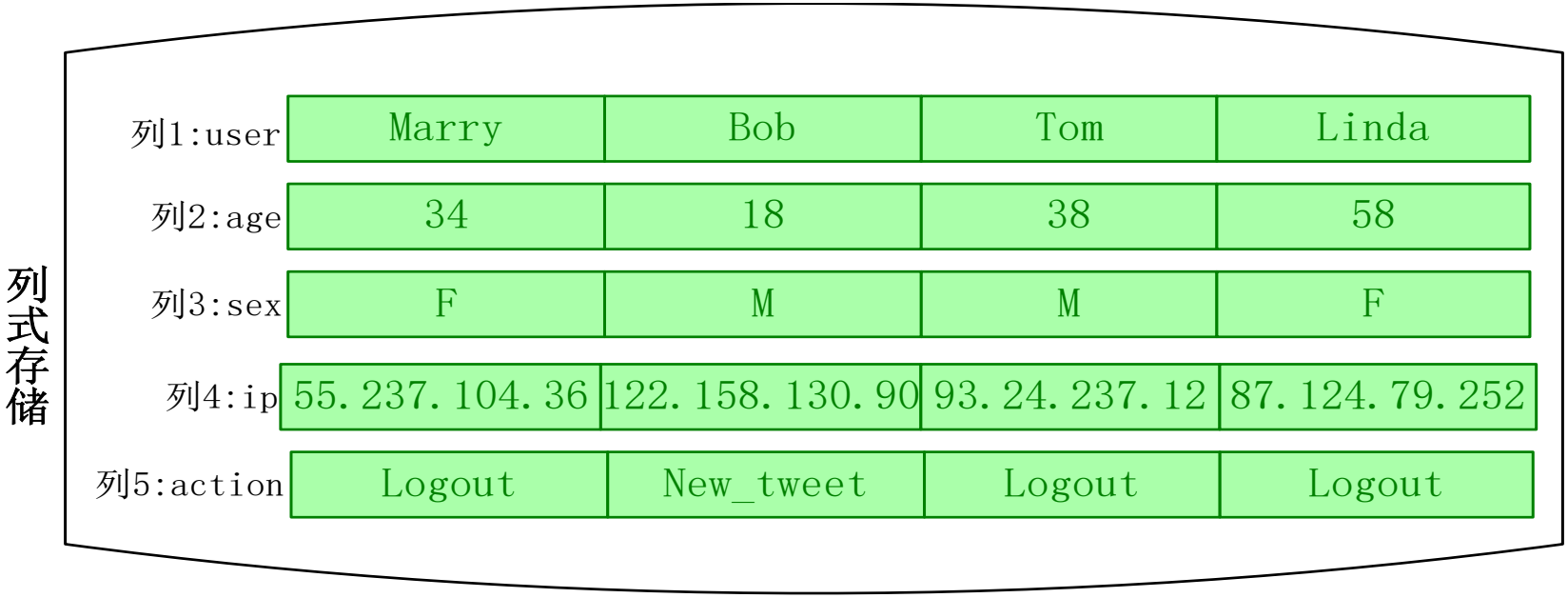
# HBase面向列的存储

| Log | | | | | |
|---|---|---|---|---|---|
| Log_id | User | Age | Sex | IP | Action |
| 1 | Marry | 34 | F | 55.237.104.36 | Logout |
| 2 | Bob | 18 | M | 122.158.130.90 | New_tweet |
| 3 | Tom | 38 | M | 93.24.237.12 | Logout |
| 4 | Linda | 58 | F | 87.124.79.252 | Logout |



# HBase面向列的存储

| Log | | | | | |
|---|---|---|---|---|---|
| Log_id | User | Age | Sex | IP | Action |
| 1 | Marry | 34 | F | 55.237.104.36 | Logout |
| 2 | Bob | 18 | M | 122.158.130.90 | New_tweet |
| 3 | Tom | 38 | M | 93.24.237.12 | Logout |
| 4 | Linda | 58 | F | 87.124.79.252 | Logout |

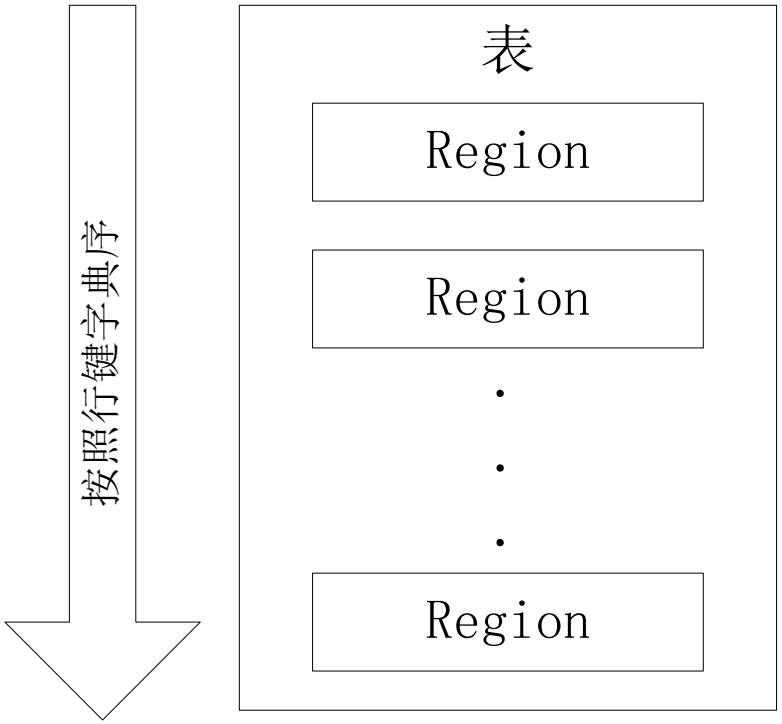


# HBase功能组件

- HBase的实现包括三个主要的功能组件：
  - ✓（1）**库函数**：链接到每个客户端
  - ✓（2）**Master主服务器：只有一个**
  - ✓（3）**Region服务器：通常有多个**
- **Master主服务器：**负责管理和维护HBase表的分区信息，维护Region服务器列表，分配Region，负载均衡
- **Region服务器：**负责存储和维护分配给自己的Region，处理来自客户端的读写请求
- **客户端：**并不是直接从Master主服务器上读取数据，而是在获得Region的存储位置信息后，直接从Region服务器上读取数据
- 客户端并不依赖Master，而是通过Zookeeper来获得Region位置信息,大多数客户端甚至从来不和Master通信，这种设计方式使得Master负载很小

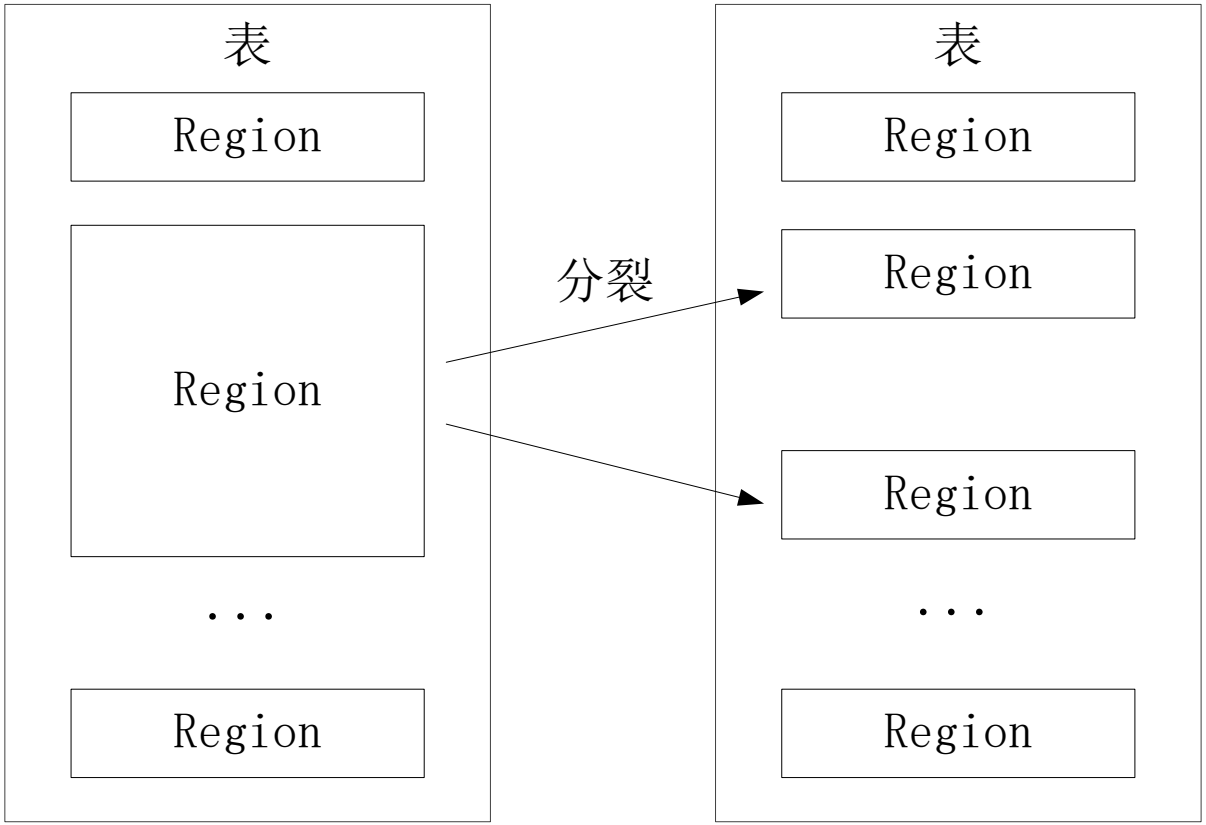
# 表和Region

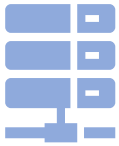
- 一个HBase表被划分成多个Region



# 表和Region

- 开始只有一个Region，后来不断分裂
- Region拆分操作非常快，接近瞬间
- 拆分之后的Region读取的仍然是原存储文件，直到“合并”过程把存储文件异步地写到独立的文件之后，才会读取新文件





# 表和Region

- 每个Region默认大小是100MB到200MB（2006年以前的硬件配置）
  - 每个Region的最佳大小取决于单台服务器的有效处理能力
  - 目前每个Region最佳大小建议1GB-2GB（2013年以后的硬件配置）
- 每个Region服务器存储10-1000个Region

注　意

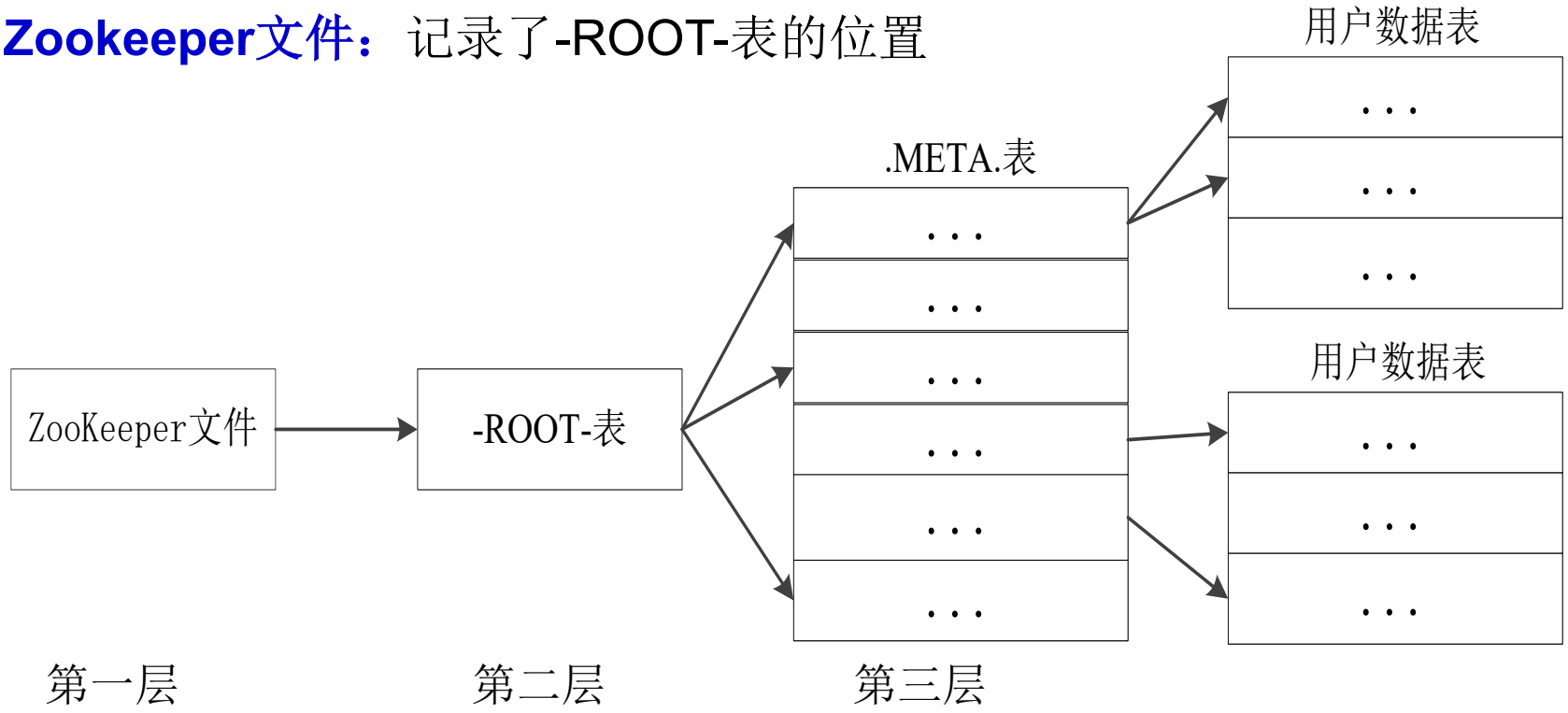
1个Region服务器←→N个Region

同一Region ←→ 只能存储在同一Region服务器上

不同的Region ←→ 可分布在不同的Region服务器上

# Region的定位——HBase的三层结构

- **.META.表**：元数据表，存储Region和Region服务器的映射关系
  - 当HBase表很大时， .META. 表也会被分裂成多个Region
- **-ROOT-表**：根数据表，记录所有元数据的具体位置
  - -ROOT-表只有唯一一个Region，名字是在程序中被写死的
- **Zookeeper文件**：记录了-ROOT-表的位置




HBase的三层结构

# Region的定位——HBase的三层结构

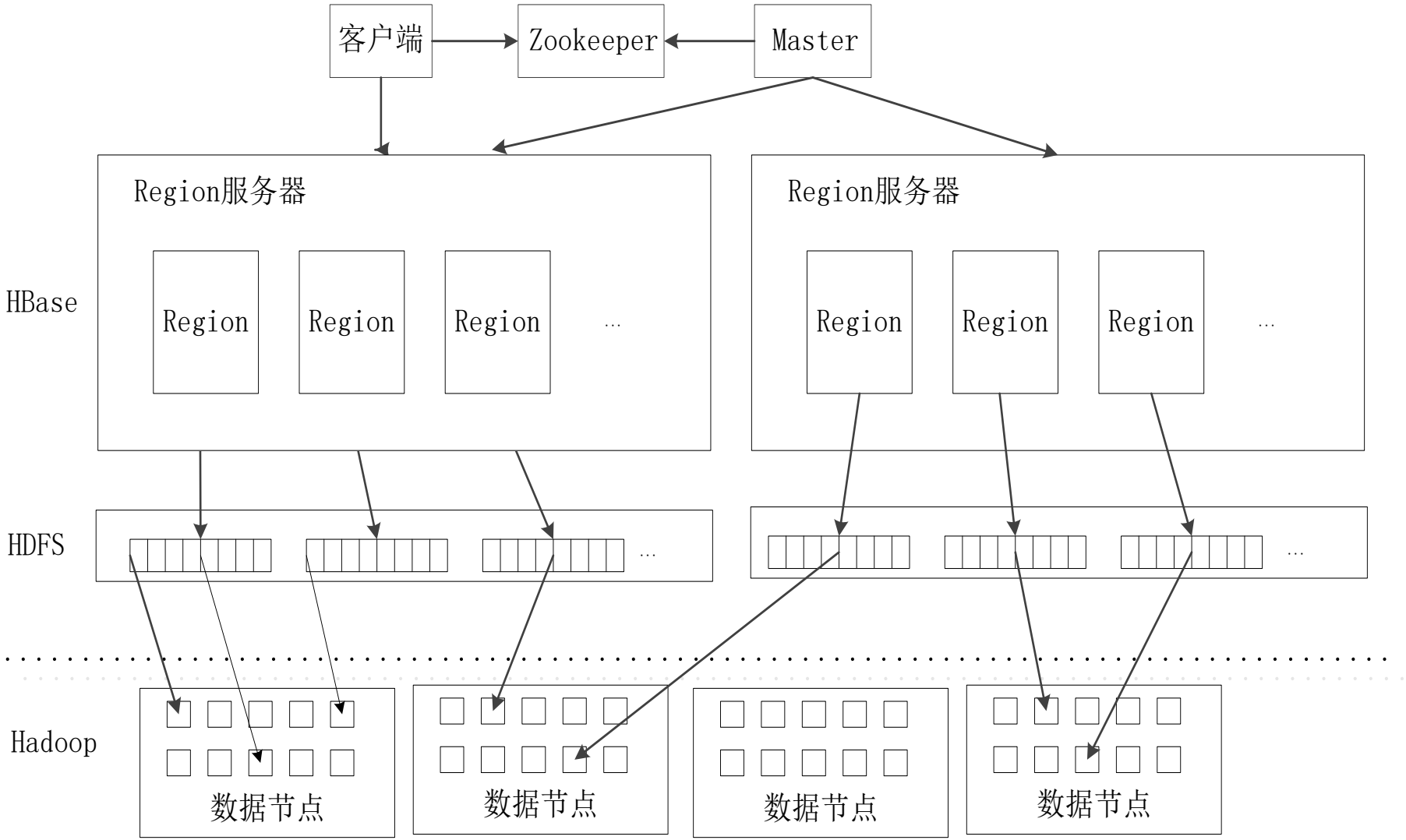
HBase的三层结构中各层次的名称和作用

| 层次 | 名称 | 作用 |
| --- | --- | --- |
| 第一层 | Zookeeper文件 | · 记录了-ROOT-表的位置信息 |
| 第二层 | -ROOT-表 | · 记录了.META.表的Region位置信息<br>· -ROOT-表只能有一个Region<br>· 通过-ROOT-表,就可以访问.META.表中的数据 |
| 第三层 | .META.表 | · 记录了用户数据表的Region位置信息<br>· .META.表可以有多个Region，保存了HBase中所有用户数据表的Region位置信息 |

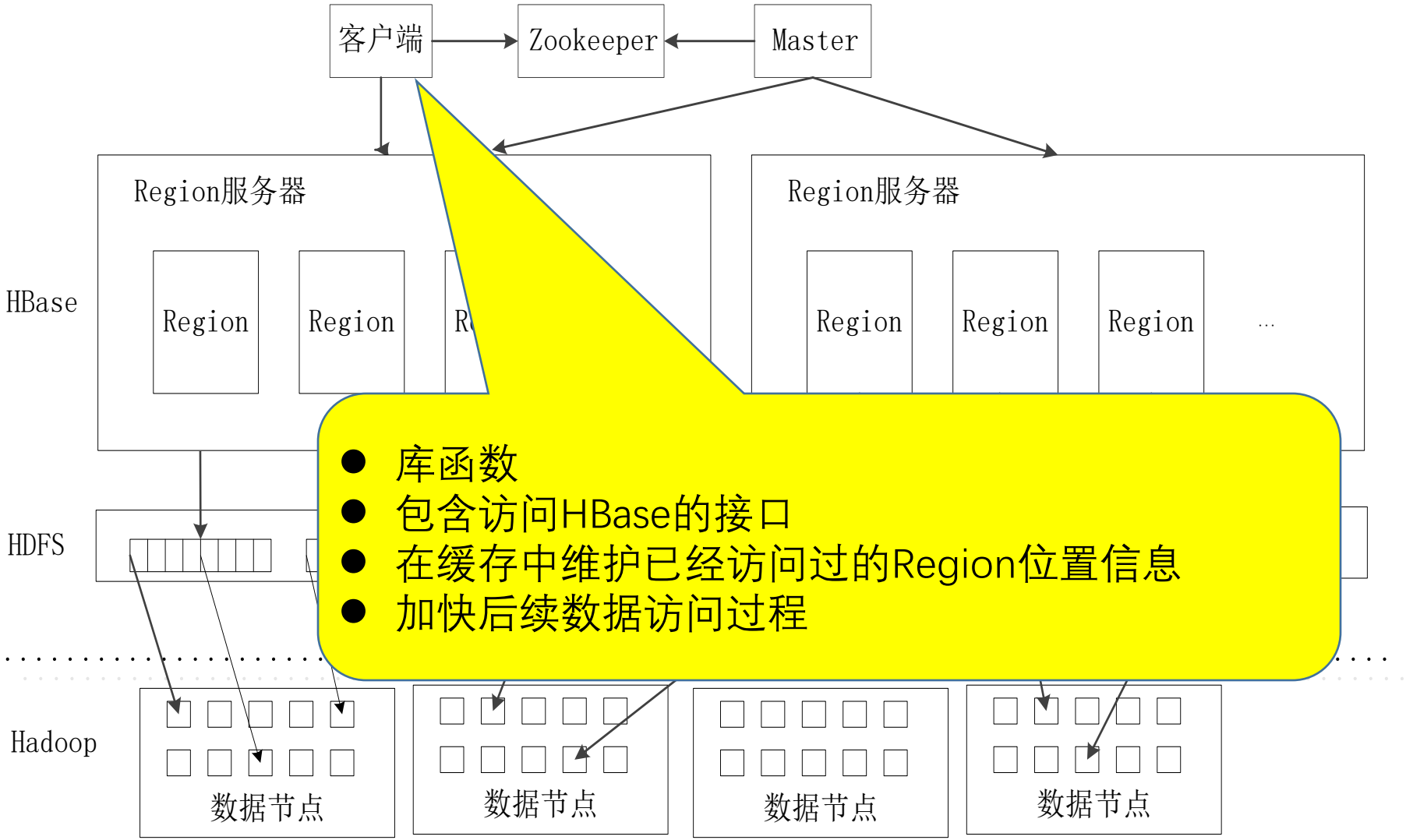
为了加快访问速度，.META.表的全部Region都会被保存在内存中

客户端访问数据时只需要“三级寻址”就可完成

# HBase系统架构-总览

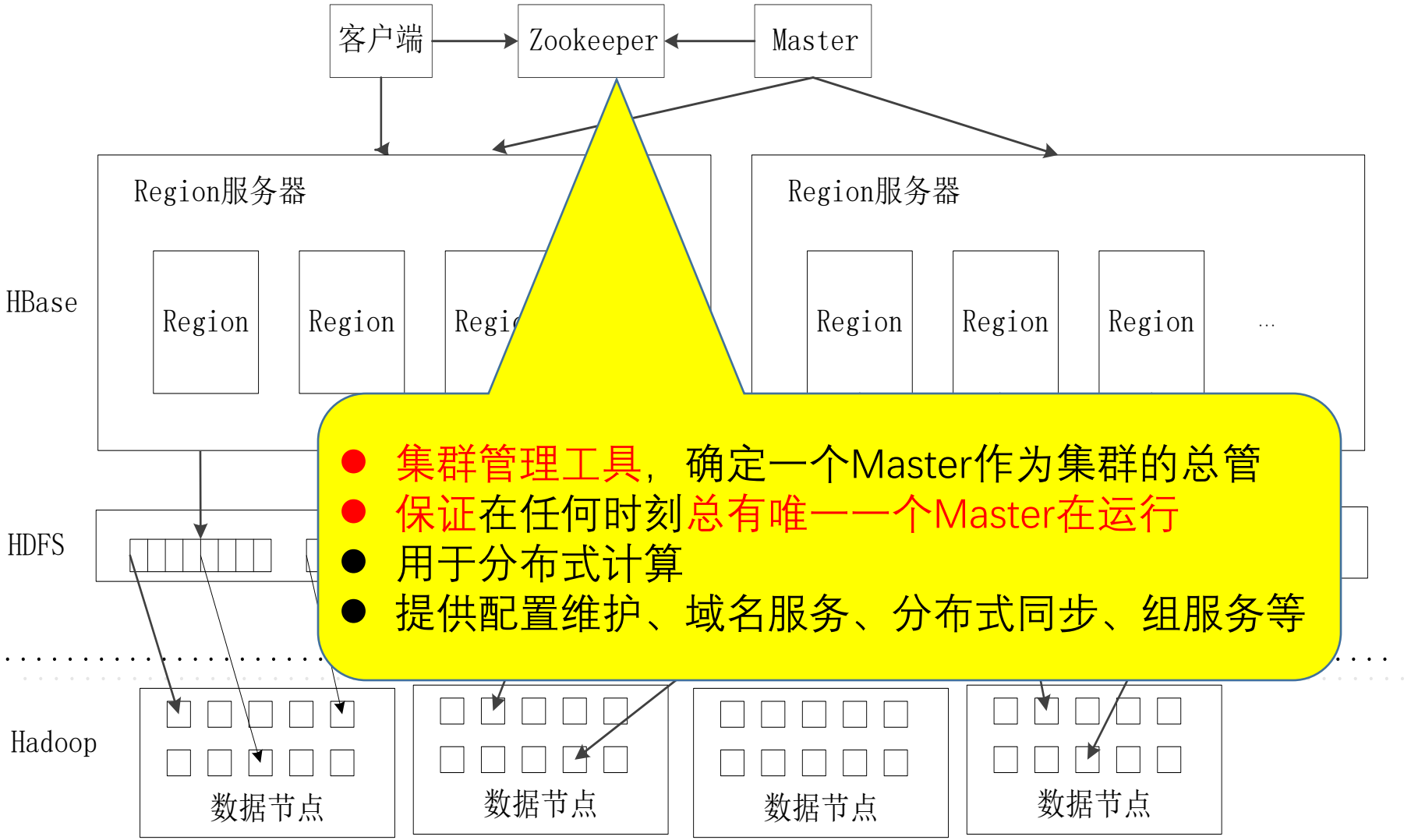


# HBase系统架构：客户端

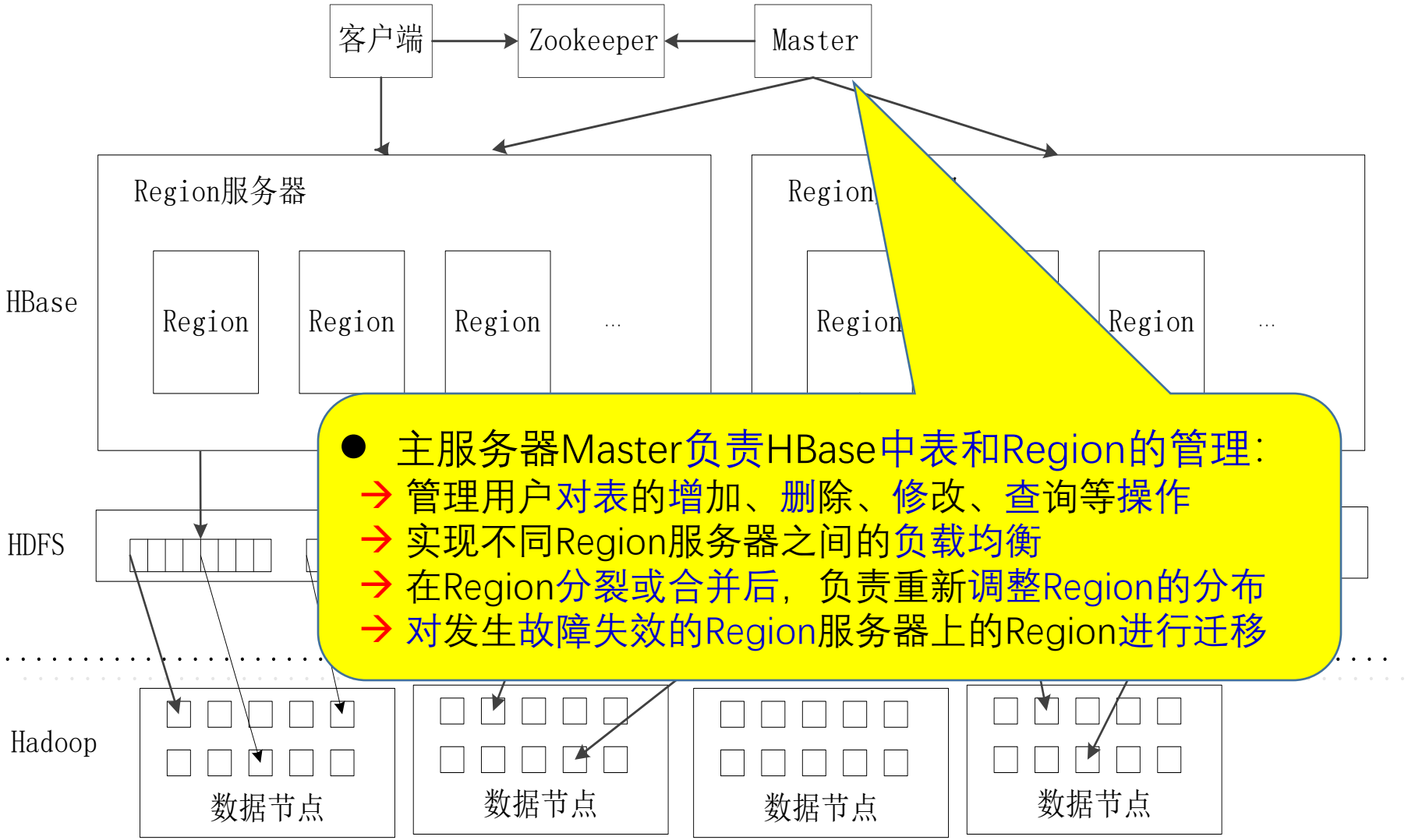



- 库函数
- 包含访问HBase的接口
- 在缓存中维护已经访问过的Region位置信息
- 加快后续数据访问过程

# HBase系统架构：Zookeeper

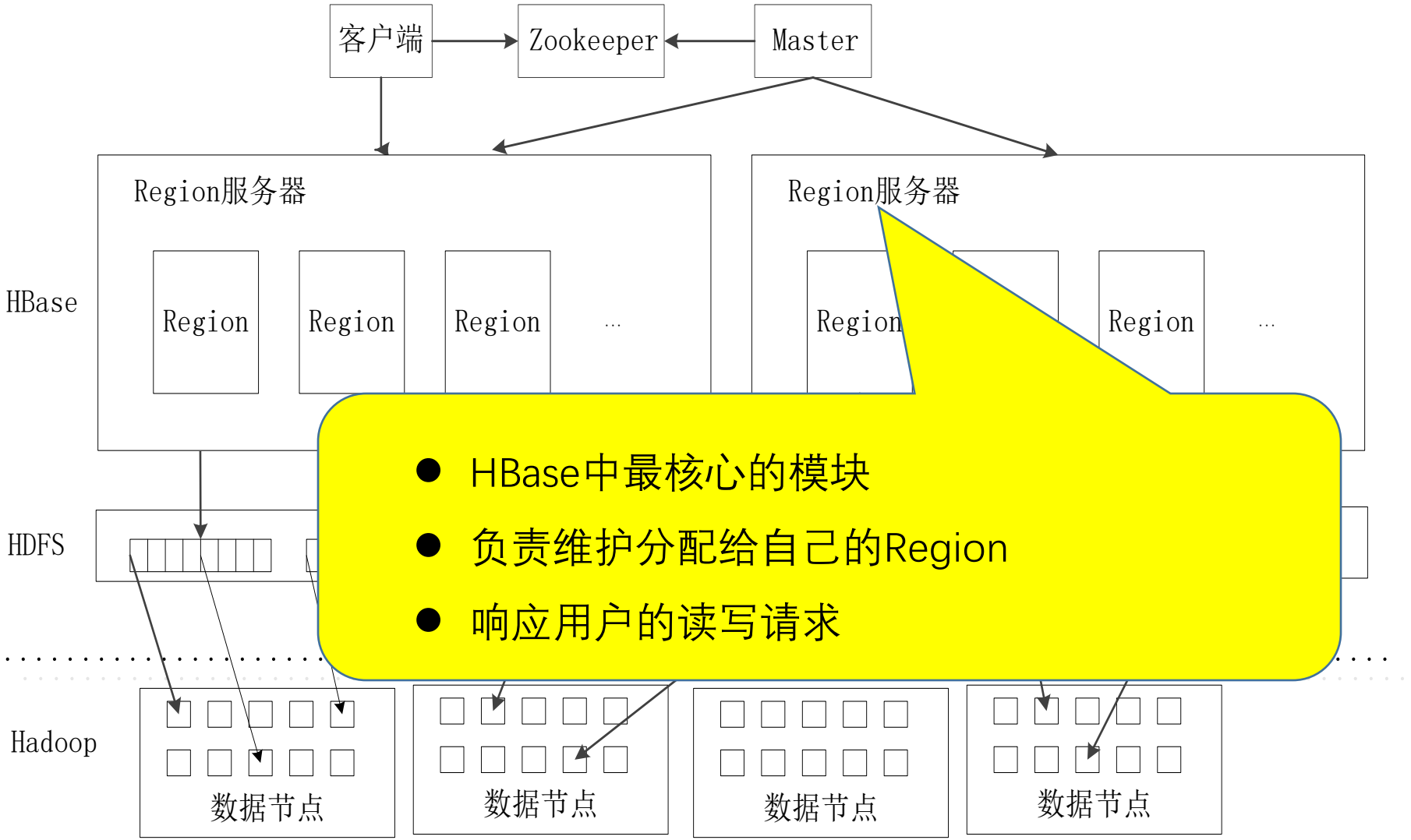


# HBase系统架构：Master



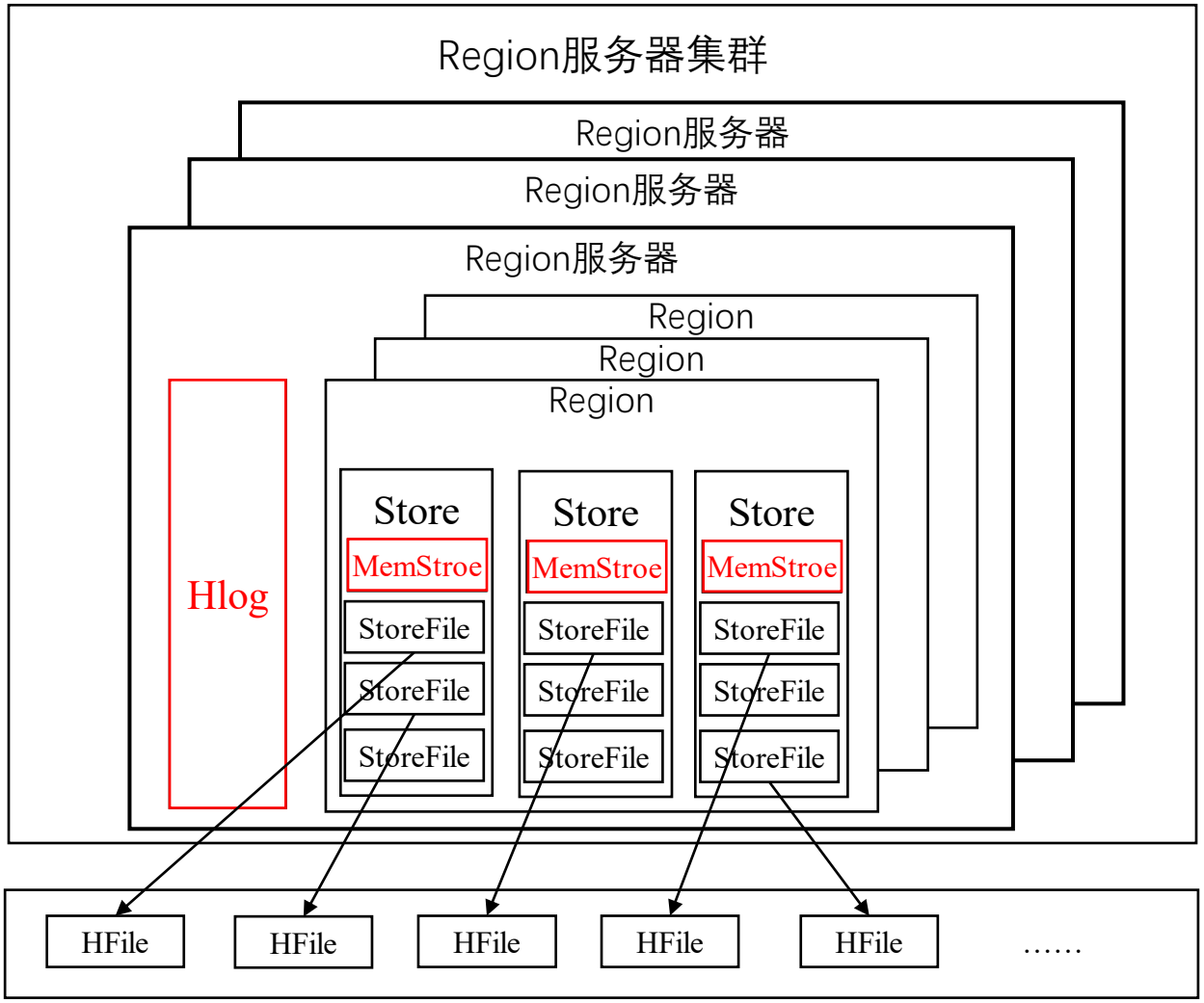

- 主服务器Master负责HBase中表和Region的管理：
  - → 管理用户对表的增加、删除、修改、查询等操作
  - → 实现不同Region服务器之间的负载均衡
  - → 在Region分裂或合并后，负责重新调整Region的分布
  - → 对发生故障失效的Region服务器上的Region进行迁移

# HBase系统架构：Region



# Region服务器工作原理

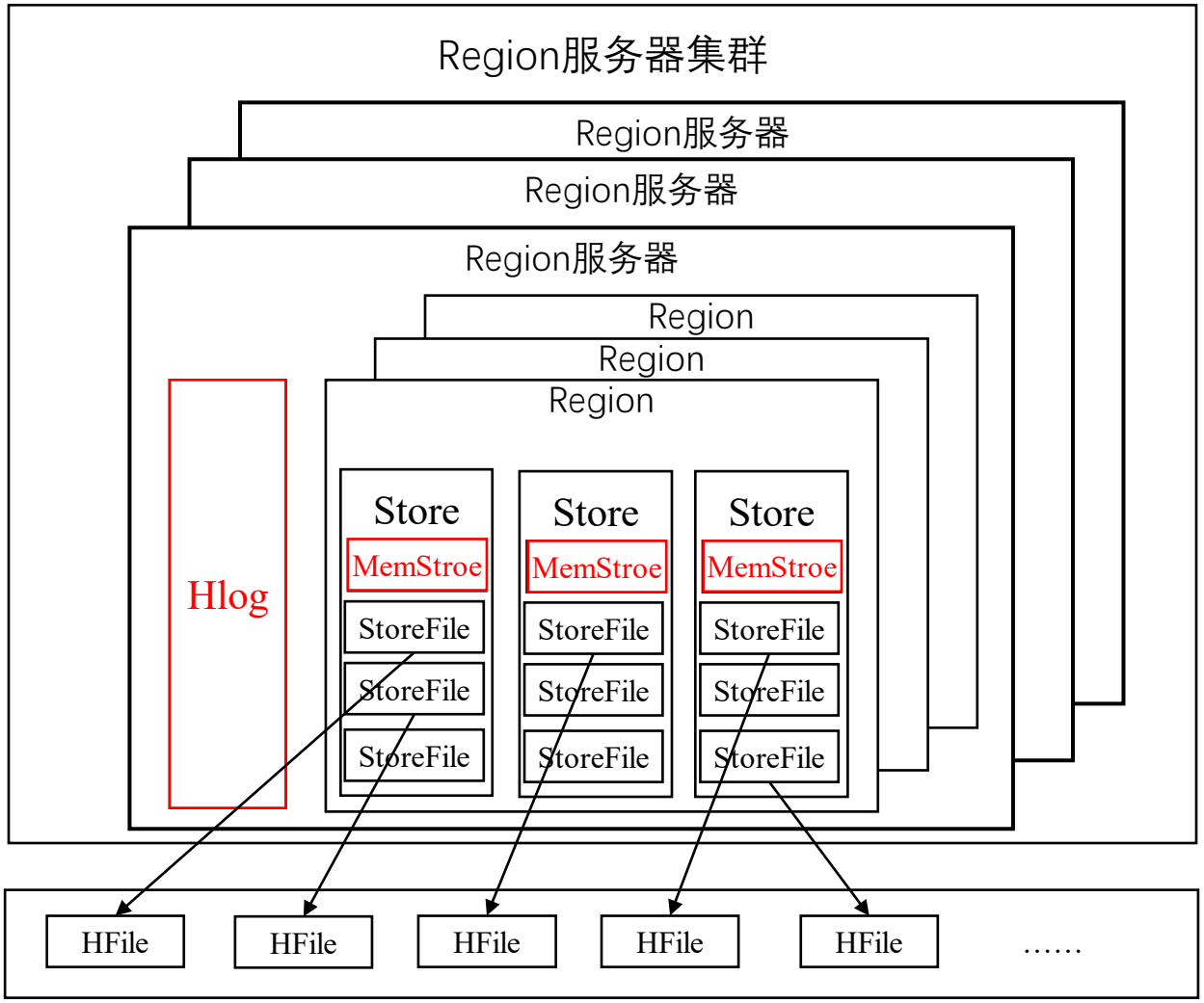
- Region服务器向HDFS文件系统中读写数据




1. 集群由多个Region服务器构成
2. 服务器由多个(10-1000个)Region构成
3. 每个服务器中的多个Region共享一个HLog
4. 每个Region会按列族切分，每个列族构成一个Store
5. 每个Store先写到Mem-Stroe缓存和Hlog中，MemStore写满后再刷写到StroeFile中(HBase的表现形式）
6. StroeFile通过Hfile去存储（Hfile是Hbase专用HDFS文件格式）

# Region服务器工作原理

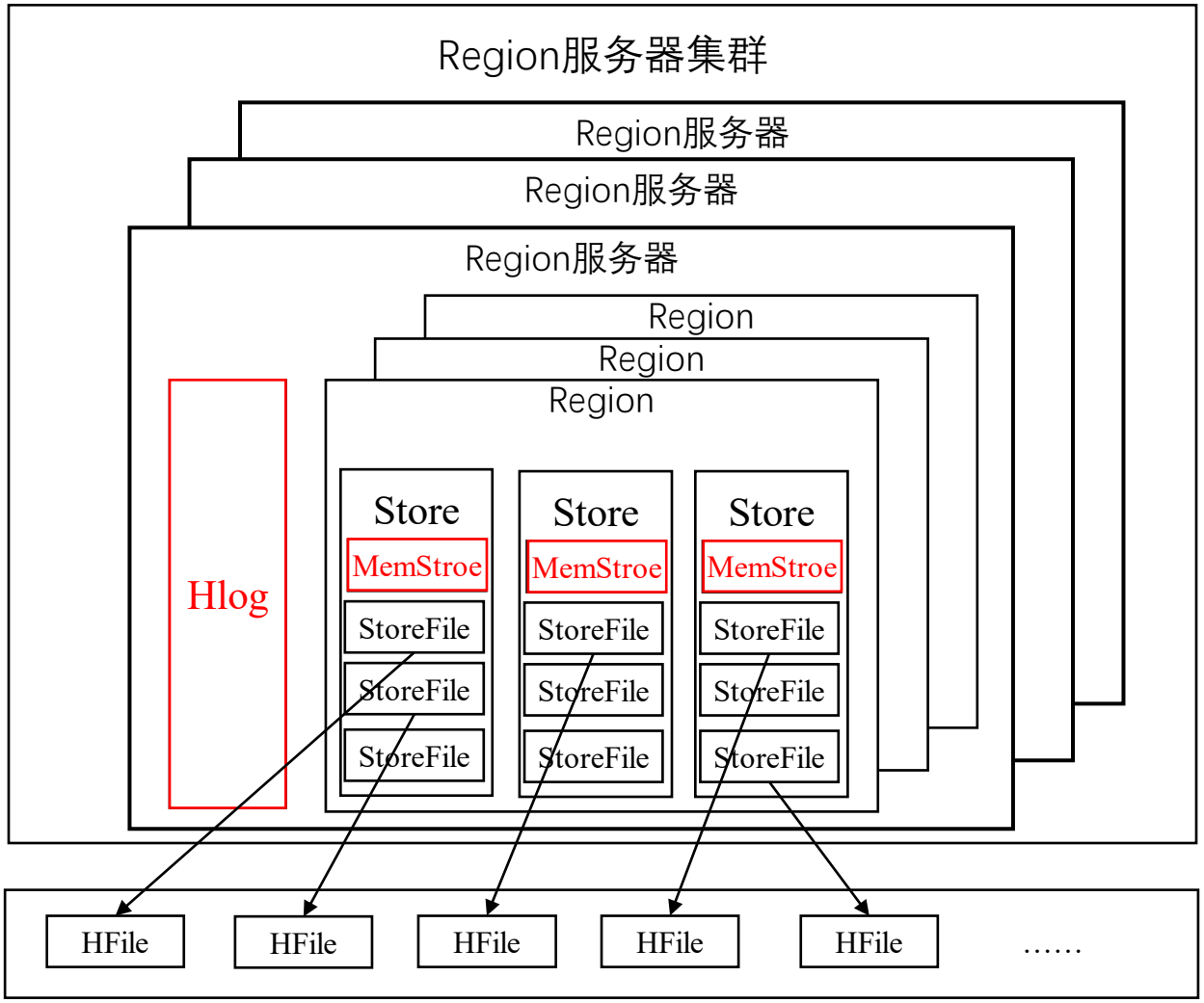
- 用户读写数据过程




- 用户写入数据时，被分配到相应Region服务器去执行
- 用户数据首先被写入到MemStore和HLog中
- 只有当操作写入HLog之后，commit()调用才会将其返回给客户端
- 当用户读取数据时，Region服务器会首先访问MemStore缓存，如果找不到，再去磁盘上面的StoreFile中寻找

# Region服务器工作原理

- 缓存的刷新




•系统会周期性地把Mem-Store缓存里的内容刷写到磁盘的StoreFile文件中

•清空缓存，并在HLog里面写入一个标记

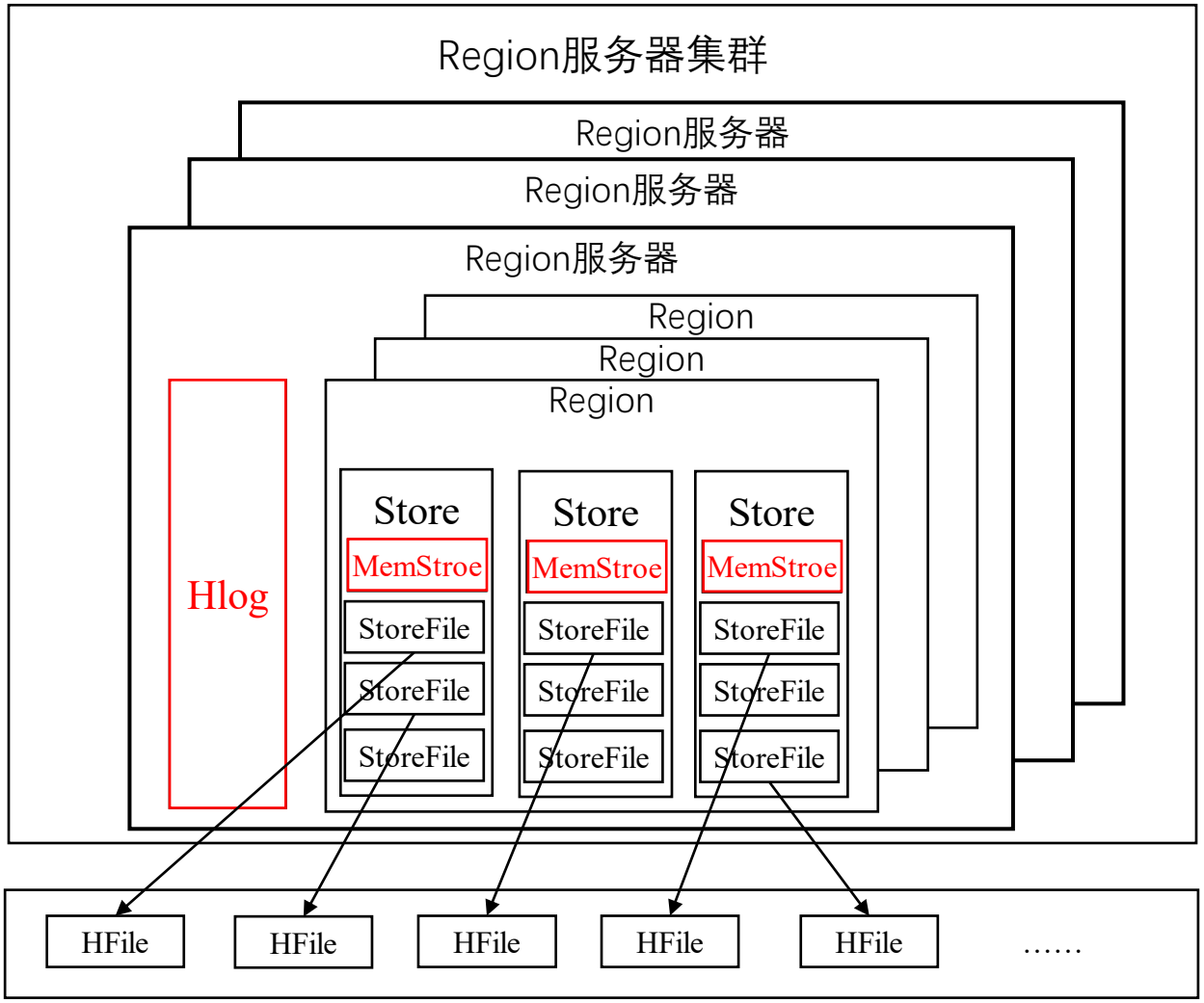
• 每次刷写都生成一个新的StoreFile文件

• 每个Region服务器都有一个自己的HLog 文件，

•每次启动都检查该文件，确认最近一次执行缓存刷新操作之后是否发生新的写入操作；如果发现更新，则先写入MemStore，再刷写到StoreFile，最后删除旧的Hlog文件，开始为用户提供服务

# Region服务器工作原理

- **StoreFile的合并**




•每次刷写都生成一个新的StoreFile，数量太多，影响查找速度

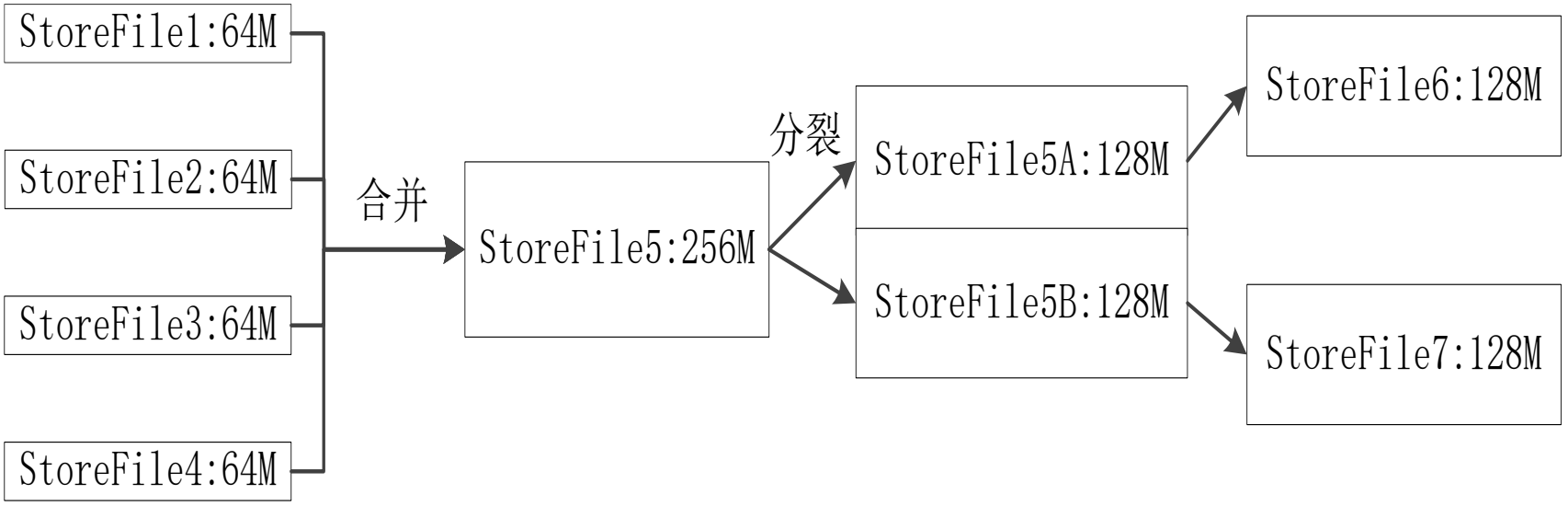
• 调用Store.compact()把多个合并成一个

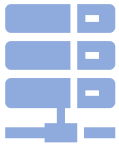
• 合并操作比较耗费资源，只有数量达到一个阈值才启动合并

# Store工作原理

StoreFile的合并和分裂过程：

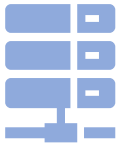
- Store是Region服务器的核心
- 多个StoreFile合并成一个
- 单个StoreFile过大时，又触发分裂操作，1个父Region被分裂成两个子Region





# HLog工作原理

- 分布式环境必须要考虑系统出错
  - ✓HBase采用HLog保证系统恢复
- HBase系统为每个Region服务器配置了一个HLog文件
  - ✓它是一种预写式日志（Write Ahead Log）
- 用户更新数据必须先写入日志后，才能写入MemStore缓存
  - ✓直到MemStore缓存内容对应的日志已经写入磁盘，该缓存内容才能被刷写到磁盘



# HLog工作原理

- Zookeeper会实时监测每个Region服务器的状态
  - 当某个Region服务器发生故障时，Zookeeper会通知Master
- Master首先会处理该故障Region服务器上面遗留的HLog文件
  - 这个遗留的HLog文件中包含了来自多个Region对象的日志记录
- 系统会根据每条日志记录所属的Region对象对HLog数据进行拆分，分别放到相应Region对象的目录下，然后，再将失效的Region重新分配到可用的Region服务器中，并把与该Region对象相关的HLog日志记录也发送给相应的Region服务器
- Region服务器领取到分配给自己的Region对象以及与之相关的HLog日志记录以后，会重新做一遍日志记录中的各种操作，把日志记录中的数据写入到MemStore缓存中，然后，刷新到磁盘的StoreFile文件中，完成数据恢复
- 共用日志优点：提高对表的写操作性能；缺点：恢复时需要分拆日志

# 第十二章 NoSQL和云数据库

1 分布式数据库

2 NoSQL简介

3 NoSQL的技术特点

4 云数据库概述

# NoSQL简介


概念演变 →

**Not only SQL**

最初表示“反SQL”运动
用新型的非关系数据库取代关系数据库

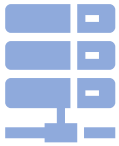
现在表示关系和非关系型数据库各有优缺点
彼此都无法互相取代

NoSQL数据库具有以下几个特点：

**（1）灵活的可扩展性**

**（2）灵活的数据模型**

**（3）与云计算紧密融合**



# NoSQL简介

现在已经有很多公司使用了NoSQL数据库：

- Google
- Facebook
- Mozilla
- Adobe
- Foursquare
- LinkedIn
- Digg
- McGraw-Hill Education
- Vermont Public Radio
- 百度、腾讯、阿里、新浪、华为.....

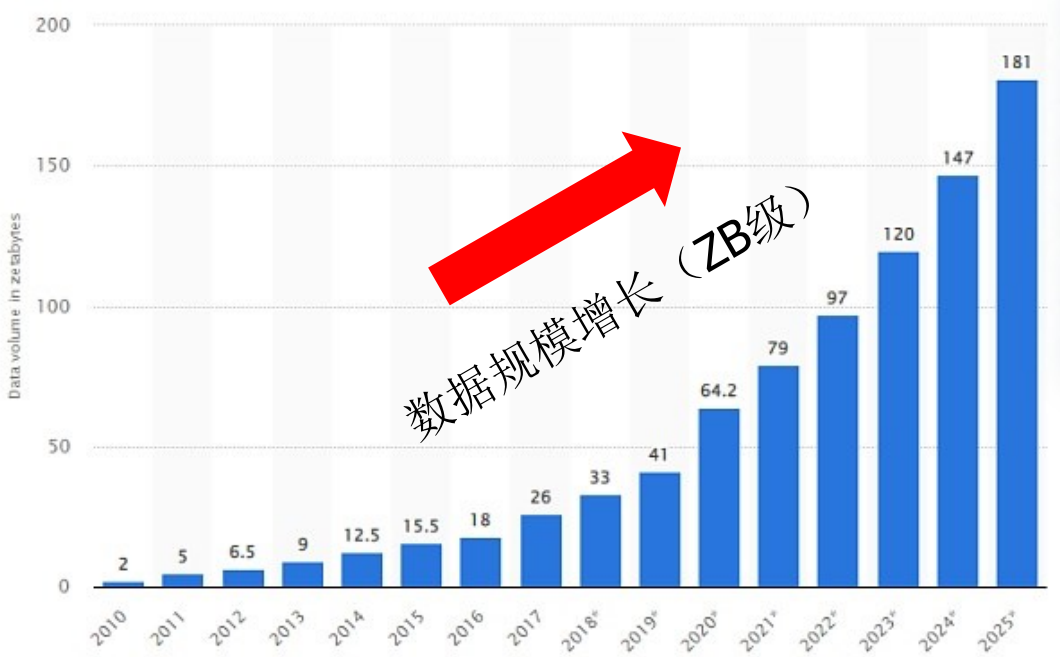
# NoSQL兴起的原因

1、关系数据库已经无法满足Web2.0的需求。主要表现在以下几个方面：

（1）无法满足海量数据的管理需求

（2）无法满足数据高并发的需求

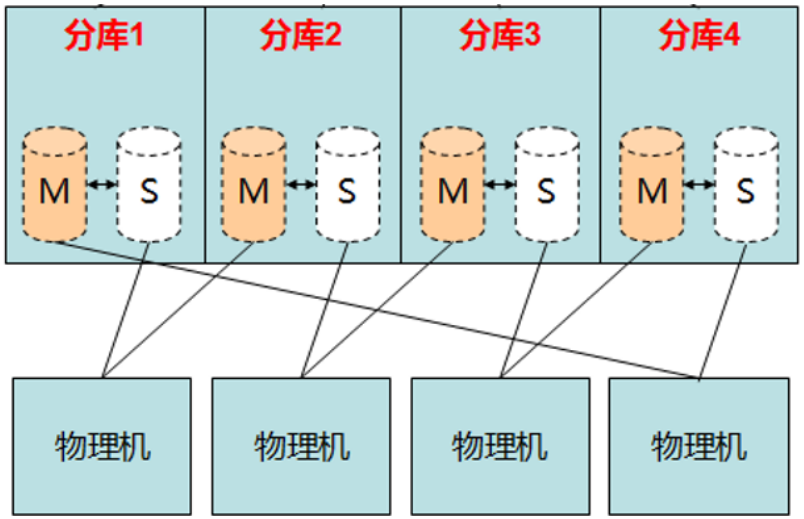
（3）无法满足高可扩展性和高可用性的需求

# NoSQL兴起的原因

**MySQL集群是否可以完全解决问题？**

- **复杂性：**
  - 部署、管理、配置很复杂
- **数据库复制：**
  - 主备之间采用异步复制
  - 当主库压力较大时常产生较大延迟
  - 主备切换可能会丢失最后一部分更新事务
  - 此时常需要人工介入，备份和恢复不方便
- **扩容问题：**
  - 需增加新服务器时，过程涉及数据重新划分
  - 整个过程比较复杂，且容易出错
- **动态数据迁移问题：**
  - 因压力需将某数据库组部分数据进行迁移时，迁移过程需要总控节点整体协调，以及数据库节点的配合
  - 过程很难自动化



# NoSQL兴起的原因

2、“One size fits all”模式很难适用于截然不同的业务场景

- 关系模型作为统一的数据模型既被用于数据分析，也被用于在线业务
- 数据分析强调高吞吐，在线业务强调低延时
- 用同一套模型来抽象不合适
- Hadoop就是针对数据分析
- MongoDB、Redis等是针对在线业务

两者都抛弃了关系模型

3、关系数据库的关键特性包括完善的事务机制和高效的查询机制，不适合Web2.0时代

（1）Web2.0网站系统通常不要求严格的数据库事务

（2）Web2.0并不要求严格的读写实时性

（3）Web2.0通常不包含大量复杂的SQL查询

- 网站设计时通常采用单表主键查询方式，已尽量减少多表连接、选择、投影操作

# NoSQL与关系数据库的比较（1）

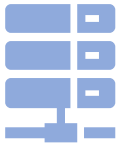
| 比较标准 | RDBMS | NoSQL | 备注 |
| --- | --- | --- | --- |
| 数据库原理 | 完全支持 | 部分支持 | · RDBMS有关系代数理论作为基础<br>· NoSQL没有统一的理论基础 |
| 数据规模 | 大 | 超大 | · RDBMS很难实现横向扩展，纵向扩展空间有限，性能会随着数据规模的增大而降低<br>· NoSQL可很容易通过添加设备来支持更大规模数据 |
| 数据库模式 | 固定 | 灵活 | · RDBMS需要定义数据库模式，严格遵守数据定义和相关约束条件<br>· NoSQL不存在数据库模式，可自由灵活定义并存储各种不同类型的数据 |
| 查询效率 | 快 | · 可高效简单查询<br>· 不具备高度结构化查询<br>· 复杂查询性能弱 | · RDBMS借助于索引机制可实现快速查询（包括记录查询和范围查询）<br>· 很多NoSQL数据库没有面向复杂查询的索引，虽然NoSQL可以使用MapReduce来加速查询，但是，在复杂查询方面的性能仍然不如RDBMS |

# NoSQL与关系数据库的比较（2）

| 比较标准 | RDBMS | NoSQL | 备注 |
| --- | --- | --- | --- |
| 一致性 | 强一致性 | 弱一致性 | · RDBMS严格遵守事务ACID模型，可保证事务强一致性<br>· 很多NoSQL数据库放松了对事务ACID四性的要求，而是遵守BASE模型，即只能保证最终一致性 |
| 数据完整性 | 容易实现 | 很难实现 | · 任何RDBMS都可以很容易实现数据完整性<br>· NoSQL数据库却无法实现 |
| 扩展性 | 一般 | 好 | · RDBMS很难实现横向扩展，纵向扩展的空间也有限<br>· NoSQL在设计之初就充分考虑了横向扩展的需求，可以很容易通过添加廉价设备实现扩展 |
| 可用性 | 好 | 很好 | · RDBMS在任何时候都以保证数据一致性为优先目标，其次才是优化系统性能，随着数据规模的增大，为保证严格的一致性，只能提供相对较弱的可用性<br>· 大多数NoSQL都能提供较高的可用性 |

# NoSQL与关系数据库的比较（3）

| 比较标准 | RDBMS | NoSQL | 备注 |
| --- | --- | --- | --- |
| 标准化 | 是 | 否 | · RDBMS已经标准化（SQL）<br>· NoSQL还没有行业标准，不同的NoSQL数据库都有自己的查询语言，很难规范应用程序接口 |
| 技术支持 | 高 | 低 | · RDBMS经过几十年的发展，已经非常成熟，通常有很好的技术支持<br>· NoSQL在技术支持方面仍然处于起步阶段，还不成熟，缺乏有力的技术支持 |
| 可维护性 | 复杂 | 复杂 | · RDBMS需要专门的数据库管理员(DBA)维护<br>· NoSQL数据库虽然没有DBMS复杂，但难以维护 |



# NoSQL与关系数据库的比较（总结）

## （1）关系数据库

优势：以完善的**关系代数理论**作为基础，有**严格的标准**，**支持事务ACID**四性，借助索引机制可以实现**高效的查询**，**技术成熟**，有专业公司的**技术支持**

劣势：**可扩展性较差**，**无法**较好支持海量数据存储，**数据模型过于死板**、**无法**较好支持Web2.0应用，事务机制影响了**系统的整体性能**等

## （2）NoSQL数据库

优势：可以**支持超大规模数据存储**，**灵活的数据模型**可以很好地支持Web2.0应用，具有**强大的横向扩展能力**等

劣势：**缺乏**数学理论基础，复杂**查询性能不高**，大都**不能实现**事务强一致性，**很难实现**数据完整性，**技术尚不成熟**，**缺乏**专业团队的技术支持，**维护较困难**等

# NoSQL与关系数据库的比较（总结）

两者各有优缺点，彼此无法取代

- **关系数据库应用场景**
  - 电信、银行等领域的关键业务系统，需要保证强事务一致性
- **NoSQL数据库应用场景**
  - 互联网企业、传统企业的非关键业务（比如数据分析）

**采用混合架构**

- 案例: 亚马逊公司就使用不同类型的数据库来支撑它的电子商务应用
- 对于“购物篮”这种临时性数据，采用键值存储会更加高效
- 当前的产品和订单信息则适合存放在关系数据库中
- 大量的历史订单信息则适合保存在类似MongoDB的文档数据库中

# 第十二章 NoSQL和云数据库

1 分布式数据库

2 NoSQL简介

3 NoSQL的技术特点

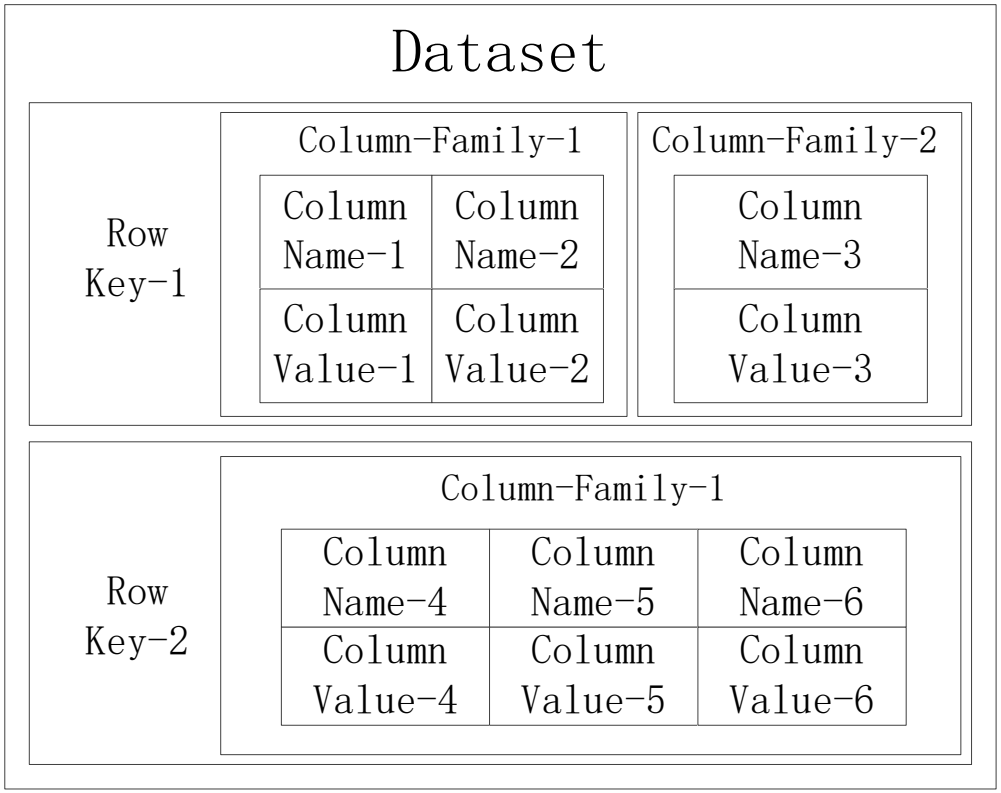
4 云数据库概述

# NoSQL的六大类型

- NoSQL数据库虽然数量众多，但**典型的NoSQL**数据库通常包括**键值数据库**、**列族数据库**、**文档数据库**和**图形数据库**

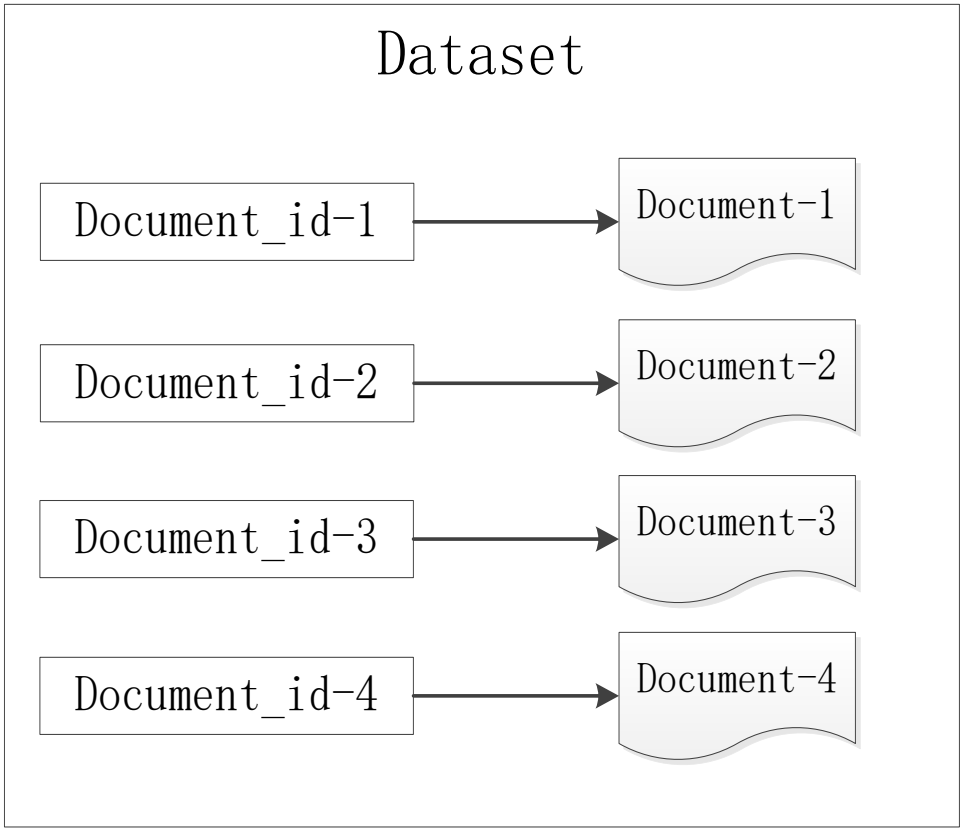
| Key_1 | Value_1 |
|---|---|
| Key_2 | Value_2 |
| Key_3 | Value_1 |
| Key_4 | Value_3 |
| Key_5 | Value_2 |
| Key_6 | Value_1 |
| Key_7 | Value_4 |
| Key_8 | Value_3 |

键值数据库

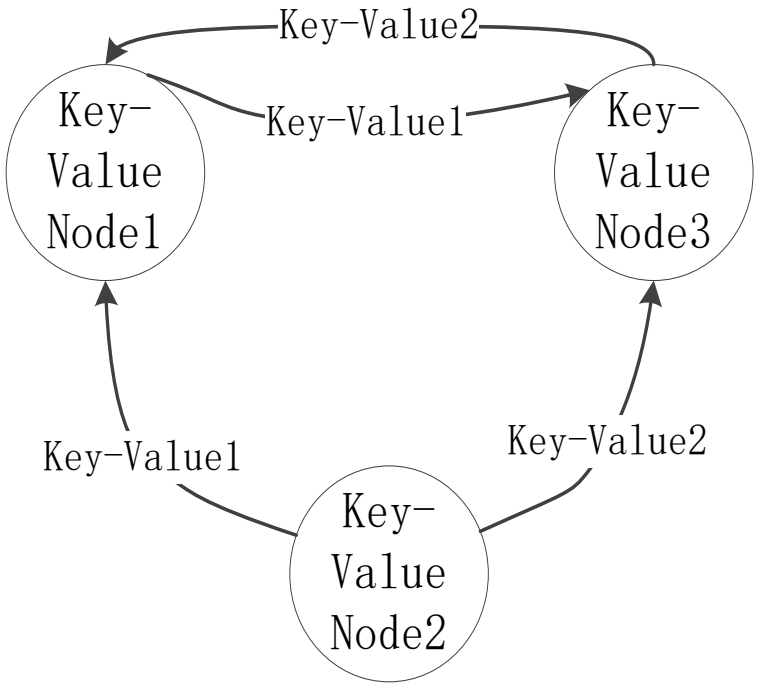



列族数据库

# NoSQL的六大类型

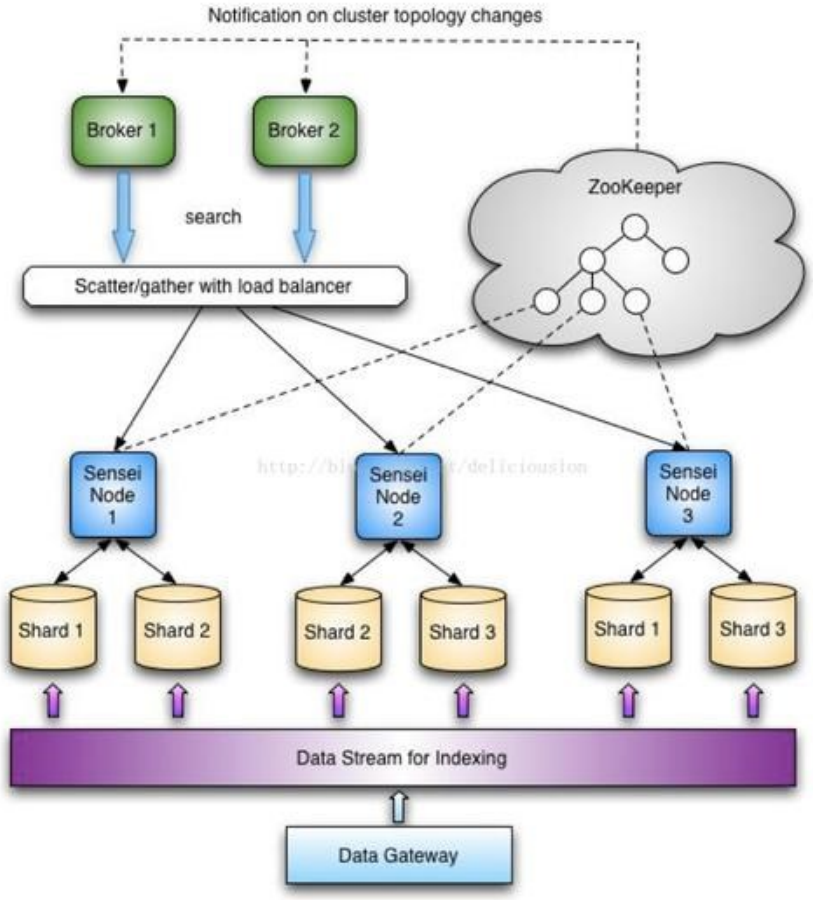



文档数据库

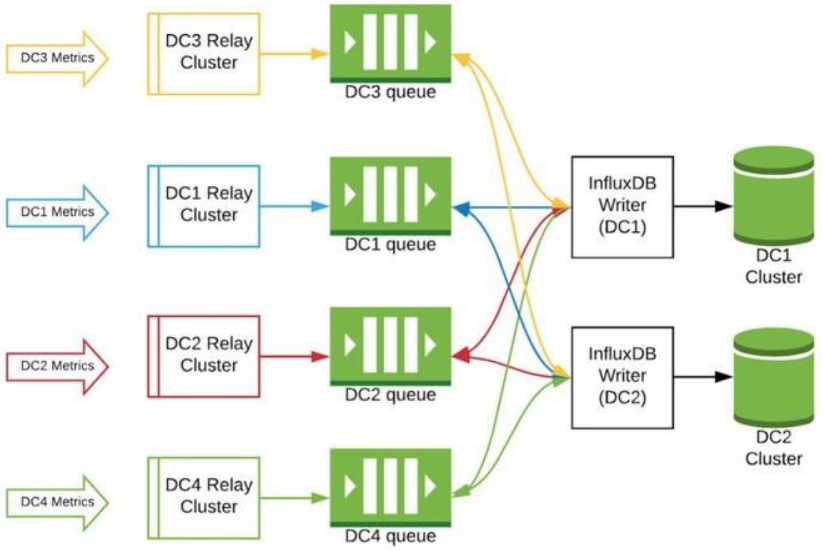



图形数据库

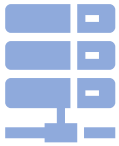
# NoSQL的六大类型




搜索数据库




时序数据库



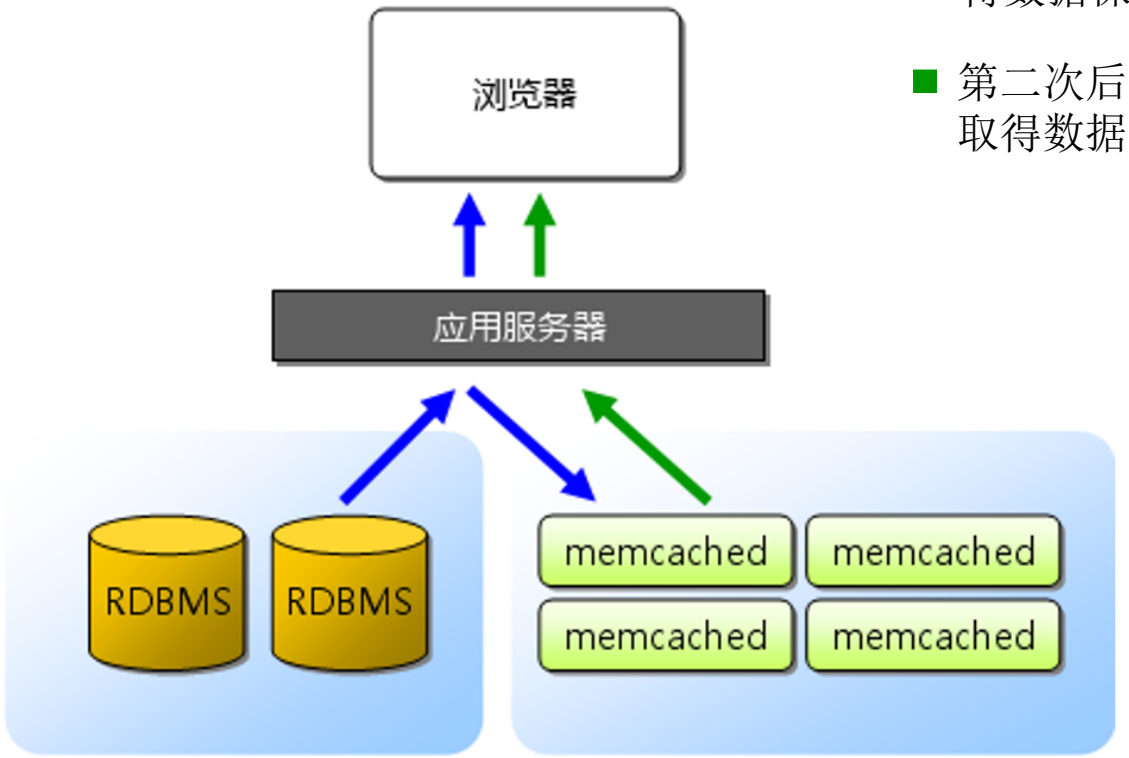
# 六大NoSQL：键值数据库

| | |
|---|---|
| 相关产品 | Redis、Riak、SimpleDB、Chordless、Scalaris、Memcached |
| 数据模型 | 键/值对。键是一个字符串对象；值可以是任意类型的数据，比如整型、字符型、数组、列表、集合等 |
| 典型应用 | 涉及频繁读写、拥有简单数据模型的应用<br>内容缓存，比如会话、配置文件、参数、购物车等<br>存储配置和用户数据信息的移动应用 |
| 优点 | 扩展性好，灵活性好，大量写操作时性能高 |
| 缺点 | 无法存储结构化信息，条件查询效率较低 |
| 不适用情形 | 通过值而非键来查：键值数据库根本没有通过值查询的途径<br>需要存储数据之间的关系：在键值数据库中，不能通过两个或两个以上的键来关联数据<br>需要事务的支持：一些键值数据库产生故障时，不可以回滚 |
| 使用者 | 百度云数据库（Redis）、GitHub（Riak）、BestBuy（Riak）、Twitter（Redis和Memcached）、StackOverFlow（Redis）、Instagram （Redis）、Youtube（Memcached）、Wikipedia（Memcached） |

# 六大NoSQL：键值数据库

- Redis又被称为“强化版的memcached”
- 支持持久化、数据恢复、更多数据类型
- 键值数据库成为理想的缓冲层解决方案

- 首次访问:从RDBMS中取得数据保存到memcached
- 第二次后:从memcached中取得数据显示页面



# 六大NoSQL：列族数据库

| | |
|---|---|
| 相关产品 | BigTable、HBase、Cassandra、HadoopDB、GreenPlum、PNUTS |
| 数据模型 | 列族 |
| 典型应用 | 分布式数据存储与管理<br>数据在地理上分布于多个数据中心的应用程序<br>可以容忍副本中存在短期不一致情况的应用程序<br>拥有动态字段的应用程序<br>拥有潜在大量数据的应用程序，大到几百TB的数据 |
| 优点 | 查找速度快，可扩展性强，容易进行分布式扩展，复杂性低 |
| 缺点 | 功能较少，大都不支持强事务一致性 |
| 不适用情形 | 需要ACID事务支持的情形，Cassandra等产品就不适用 |
| 使用者 | Ebay（Cassandra）、Instagram（Cassandra）、NASA（Cassandra）、Twitter（Cassandra and HBase）、Facebook（HBase）、Yahoo!（HBase） |

# 六大NoSQL：文档数据库

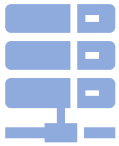
- 主要用于**存储并检索文档数据**
- **文档是**数据库的**最小单位**
  - “文档”是一个数据记录，用于对包含的数据类型和内容进行“自我描述”
- **可以包含**非常复杂的结构**而不需采用**特定的数据模式，如嵌套对象等
  - 基于加密方式的不同，可用多种格式进行解码，如XML、JSON、BSON等
- **可**通过一个键值来定位一个文档（可视为键值数据库的一个衍生品）
- **也可**基于文档内容来构建索引（区别于键值数据库之处）

例：SequoiaDB就是使用JSON格式的文档数据库，其存储方式如下：

# 六大NoSQL：文档数据库

```
{
“ID”:1,
“NAME”: “SequoiaDB”,
“Tel”: {
        “Office”: “123123”, “Mobile”: “132132132”
      }
“Addr”: “China,GZ”
}
```

- **数据是不规则的：**每一条记录包含了所有的有关“SequoiaDB”的信息而没有任何外部的引用，这条记录就是“自包含”的
- **数据迁移很容易：**因为这条记录的所有信息都包含在里面了，不需要考虑还有信息在别的表没有一起迁移走
- **ACID保证与读写速度快：**因为在移动过程中，只有被移动的那一条记录（文档）需要操作，而不像关系型中每个有关联的表都需要锁住来保证一致性



# 六大NoSQL：文档数据库

| | |
|---|---|
| 相关产品 | MongoDB、CouchDB、Terrastore、ThruDB、RavenDB、SisoDB、RaptorDB、CloudKit、Perservere、Jackrabbit |
| 数据模型 | 键 / 值　　值（value）是版本化的文档 |
| 典型应用 | 存储、索引并管理面向文档的数据或者类似的半结构化数据<br>比如，用于后台具有大量读写操作的网站、使用JSON数据结构的应用、使用嵌套结构等非规范化数据的应用程序 |
| 优点 | 性能好(高并发)，灵活性高，复杂性低，数据结构灵活<br>提供嵌入式文档功能，将经常查询的数据存储在同一文档中<br>既可以根据键来构建索引，也可以根据内容构建索引 |
| 缺点 | 缺乏统一的查询语法 |
| 不适用情形 | 在不同的文档上添加事务。文档数据库并不支持文档间的事务，如果对这方面有需求则不应该选用这个解决方案 |
| 使用者 | 百度云数据库（MongoDB）、SAP （MongoDB）、Codecademy （MongoDB）、Foursquare （MongoDB）、NBC News （RavenDB） |

# 六大NoSQL：图数据库

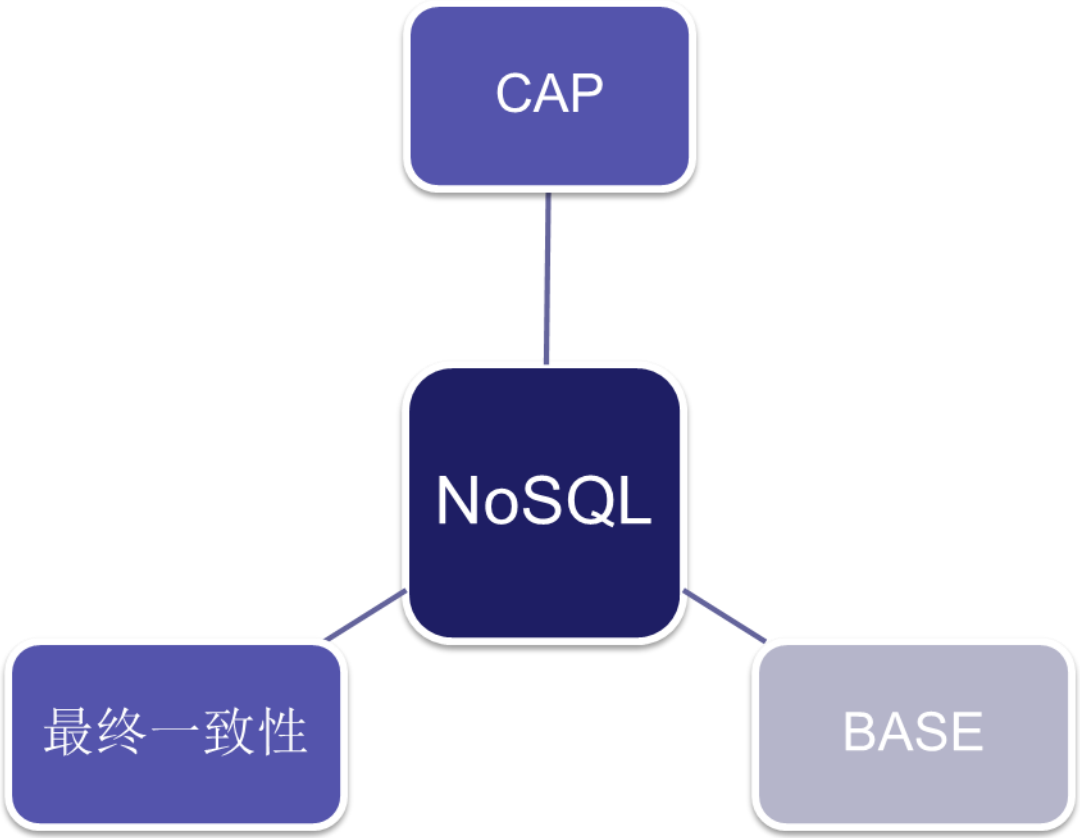
| | |
|---|---|
| 相关产品 | Neo4J、OrientDB、InfoGrid、Infinite Graph、GraphDB |
| 数据模型 | 图结构 |
| 典型应用 | 专门用于处理具有高度相互关联关系的数据，比较适合于社交网络、模式识别、依赖分析、推荐系统以及路径寻找等问题 |
| 优点 | 灵活性高，支持复杂的图形算法<br>可用于构建复杂的关系图谱 |
| 缺点 | 复杂性高，只能支持一定的数据规模 |
| 使用者 | Adobe（Neo4J）、Cisco（Neo4J）、T-Mobile（Neo4J） |

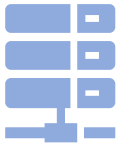
# 不同类型的数据库比较分析

| 中国菜刀 | 瑞士军刀 | 大象兵 | 金箍棒 |
| --- | --- | --- | --- |
| MySQL | MongoDB | HBase | Redis |

- **MySQL**产生年代较早，而且随着LAMP大潮得以成熟。尽管其没有什么大的改进，但是新兴的互联网使用的最多的数据库
- **MongoDB**提供更灵活的数据模型、异步提交、地理位置索引等五花十色的功能
- **HBase**是个“仗势欺人”的大象兵。依仗着Hadoop的生态环境，可以有很好的扩展性。但是就像象兵一样，使用者需要养一头大象(Hadoop),才能驱使他
- **Redis**是键值存储的代表，功能最简单。提供随机数据存储。就像一根棒子一样，没有多余的构造。但是也正是因此，它的伸缩性特别好。就像悟空手里的金箍棒，大可捅破天，小能成缩成针

# NoSQL的三大基石





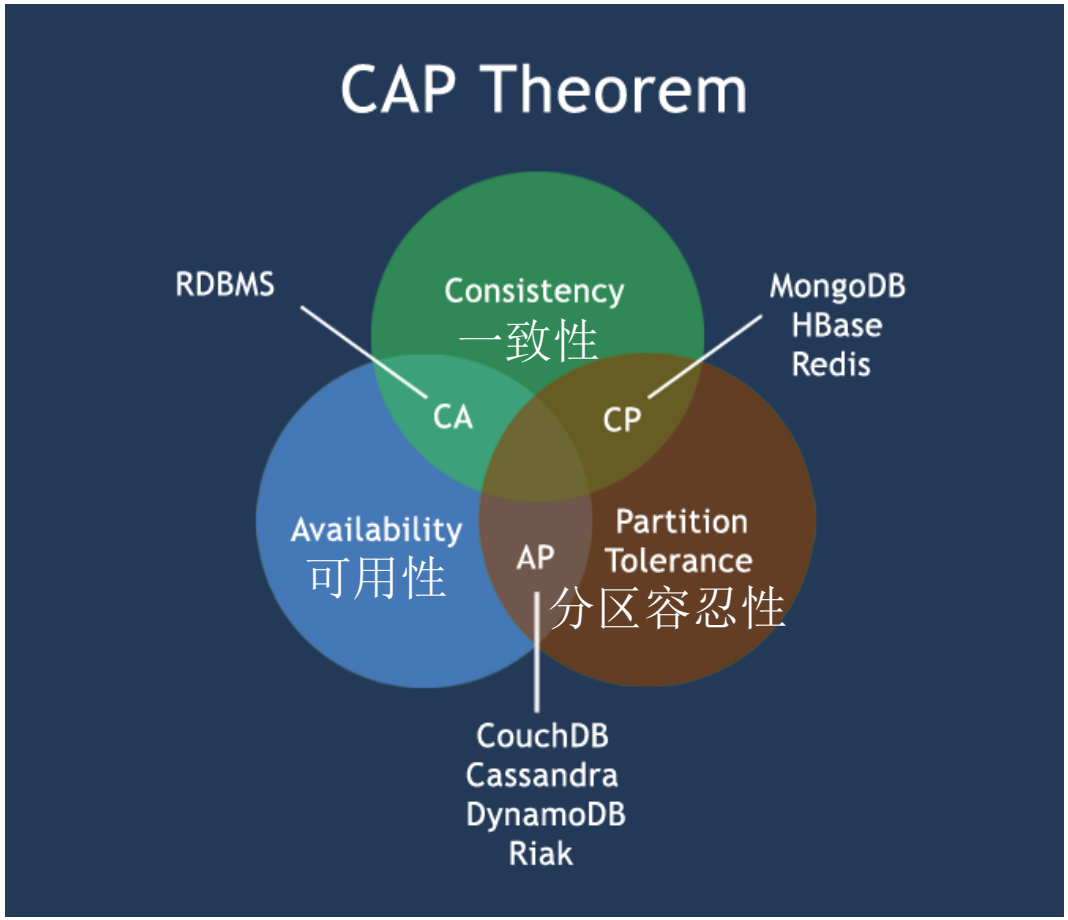
# CAP理论

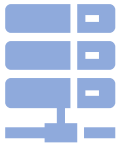
所谓的CAP指的是：

- **C（Consistency）：一致性**，是指任何一个读操作总是能够读到之前完成的写操作的结果，也就是在分布式环境中，多点的数据是一致的，或者说，所有节点在同一时间具有相同的数据
- **A（Availability）：可用性，**是指快速获取数据，可以在确定的时间内返回操作结果，保证每个请求不管成功或者失败都有响应；
- **P（Tolerance of Network Partition）：分区容忍性，**是指当出现网络分区的情况时（即系统中的一部分节点无法和其他节点进行通信），分离的系统也能够正常运行，也就是说，系统中任意信息的丢失或失败不会影响系统的继续运作。

# CAP理论

CAP理论告诉我们，一个分布式系统**不可能同时满足一致性(C)、可用性(A)和分区容忍性(P)**这三个需求，**最多只能同时满足其中两个——**“鱼和熊掌不可兼得”





# CAP理论

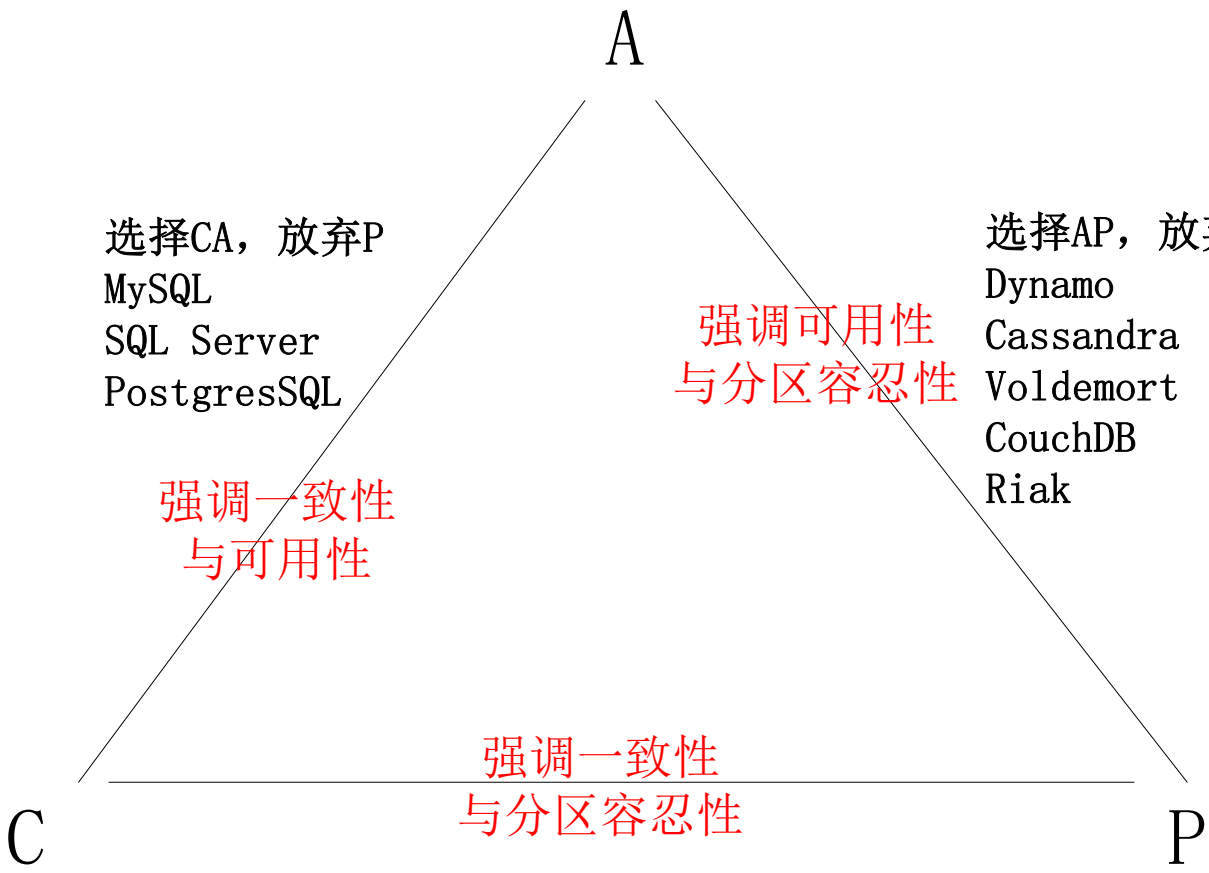
当处理CAP的问题时，可以有几个明显的选择：

**1. CA**：也就是**强调一致性（C）和可用性（A），**放弃分区容忍性（P），最简单的做法是把所有与事务相关的内容都放到同一台机器上。很显然，这种做法会严重影响系统的可扩展性。传统的关系数据库（MySQL、SQL Server和PostgreSQL），都采用了这种设计原则，因此，扩展性都比较差

**2. CP**：也就是**强调一致性（C）和分区容忍性（P），**放弃可用性（A），当出现网络分区的情况时，受影响的服务需要等待数据一致，因此在等待期间就无法对外提供服务

**3. AP**：也就是**强调可用性（A）和分区容忍性（P），**放弃一致性（C），允许系统返回不一致的数据

# CAP理论




选择CP，放弃A

Neo4J,Bigtable,MongoDB,Hbase,Hypertable,Redis

不同产品在CAP理论下的不同设计原则

# BASE理论

要理解BASE(**Basically Availble, Soft-state, Eventual consistency)**，先弄清楚ACID：

| ACID | BASE |
| --- | --- |
| 原子性(**A**tomicity) | 基本可用(**B**asically **A**vailable) |
| 一致性(**C**onsistency) | 软状态/柔性事务(**S**oft state) |
| 隔离性(**I**solation) | 最终一致性 (**E**ventual consistency) |
| 持久性 (**D**urable) | |

# BASE理论

一个数据库事务具有ACID四性：

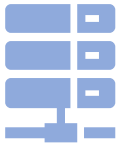
- A（Atomicity）：**原子性**，是指事务必须是原子工作单元，对于其数据修改，要么全都执行，要么全都不执行
- C（Consistency）：**一致性**，是指事务在完成时，必须使所有的数据都保持一致状态
- I（Isolation）：**隔离性**，是指由并发事务所做的修改必须与任何其它并发事务所做的修改隔离
- D（Durability）：**持久性**，是指事务完成之后，它对于系统的影响是永久性的，该修改即使出现致命的系统故障也将一直保持

# BASE理论

BASE的基本含义是**基本可用**（Basically Availble）、**软状态**（Soft-state）和**最终一致性**（Eventual consistency）：

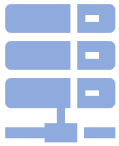
- **基本可用**
  - 指一个分布式系统的一部分发生问题变得不可用时，其他部分仍然可以正常使用，也就是允许分区失败的情形出现

- **软状态**
  - 是与“硬状态（hard-state）”相对应的一种提法
  - “硬状态” 是数据库保存的数据可以保证数据一直是正确的(一致性)
  - “软状态”是指状态可以有一段时间不同步，具有一定的滞后性



# BASE理论

- 最终一致性
  - 一致性的类型包括强一致性和弱一致性
  - 二者的主要区别在于高并发的数据访问操作下，后续操作是否能够获取最新的数据
  - 强一致性：当执行完一次更新操作后，后续其他读操作就可以保证读到更新后的最新数据
  - 弱一致性：如果不能保证后续访问读到的都是更新后的最新数据
  - 最终一致性：是弱一致性的一种特例，允许后续的访问操作可以暂时读不到更新后的数据，但是经过一段时间之后，必须最终读到更新后的数据
  - 最常见的实现最终一致性的系统是DNS（域名系统）。一个域名更新操作根据配置的形式被分发出去，并结合有过期机制的缓存；最终所有的客户端可以看到最新的值。

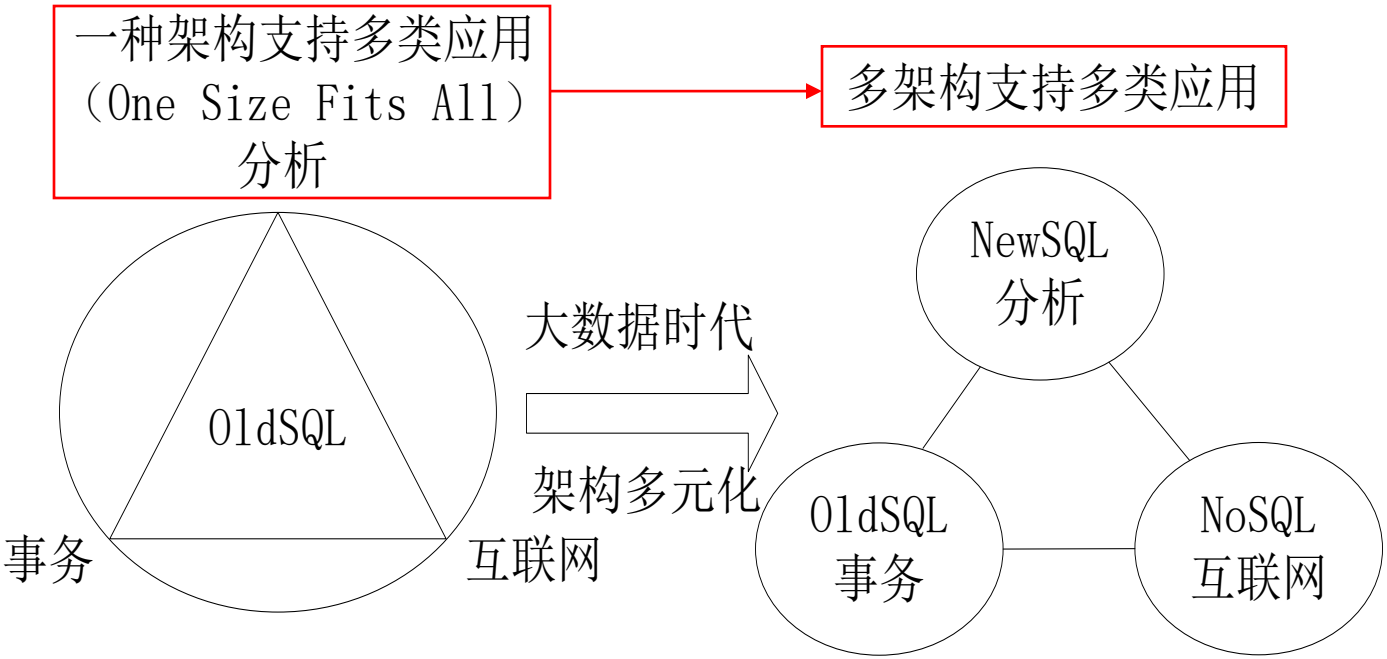


# NewSQL数据库

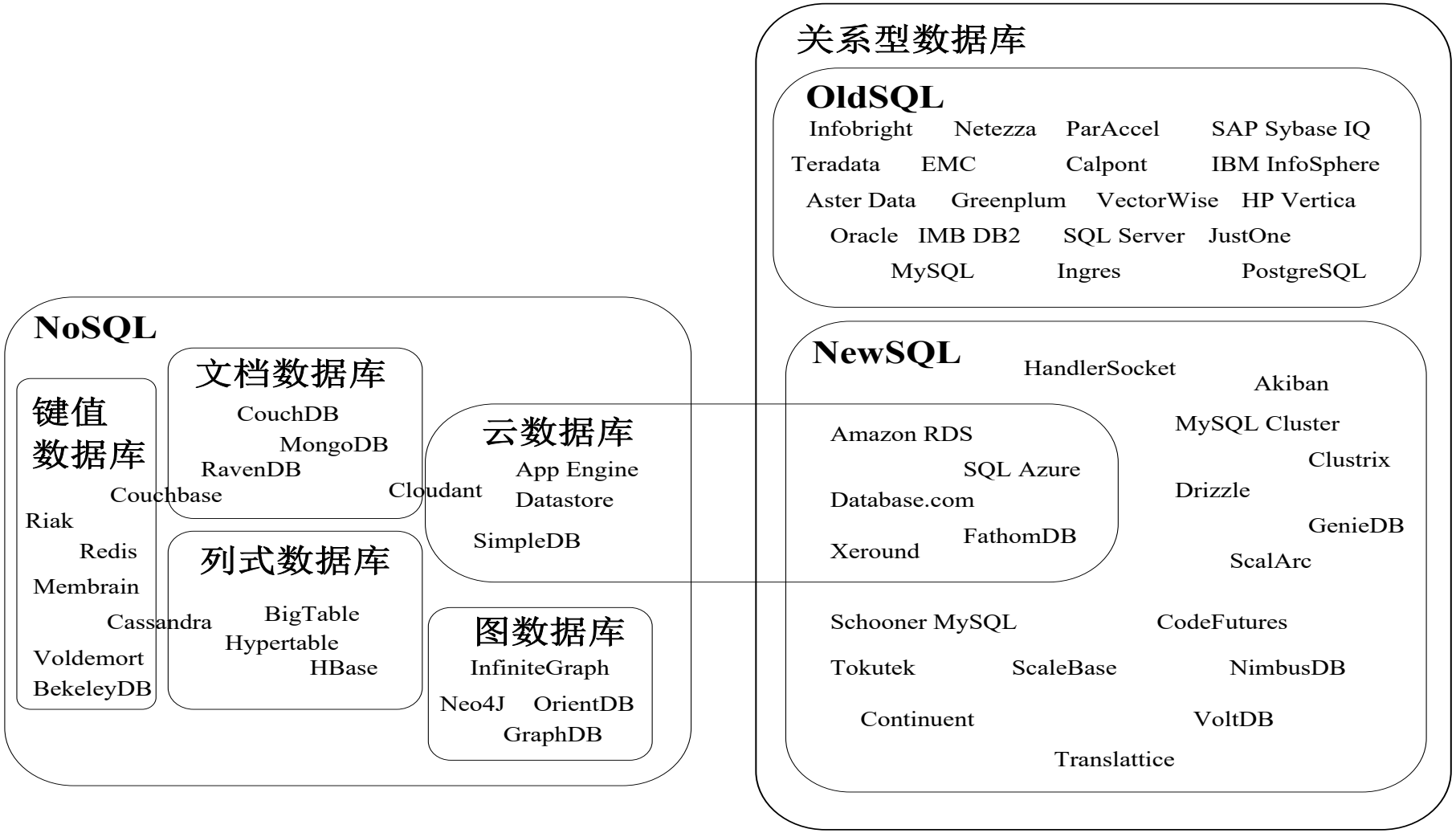
- 各种新的可扩展、高性能数据库的简称
- 不仅具有NoSQL的海量数据存储管理能力，还保持了传统DB支持ACID和SQL等特性
- 不同NewSQL内部结构差异很大，但**都有两个显著的共同特征**
  - **都支持关系数据模型**
  - **都使用SQL作为其主要接口**

# 从NoSQL到NewSQL数据库

大数据引发数据处理架构变革



# 从NoSQL到NewSQL数据库




关系数据库、NoSQL和NewSQL数据库产品分类图

# 第十二章 NoSQL和云数据库

1 分布式数据库

2 NoSQL简介

3 NoSQL的技术特点

4 云数据库概述

# 云计算是云数据库兴起的基础

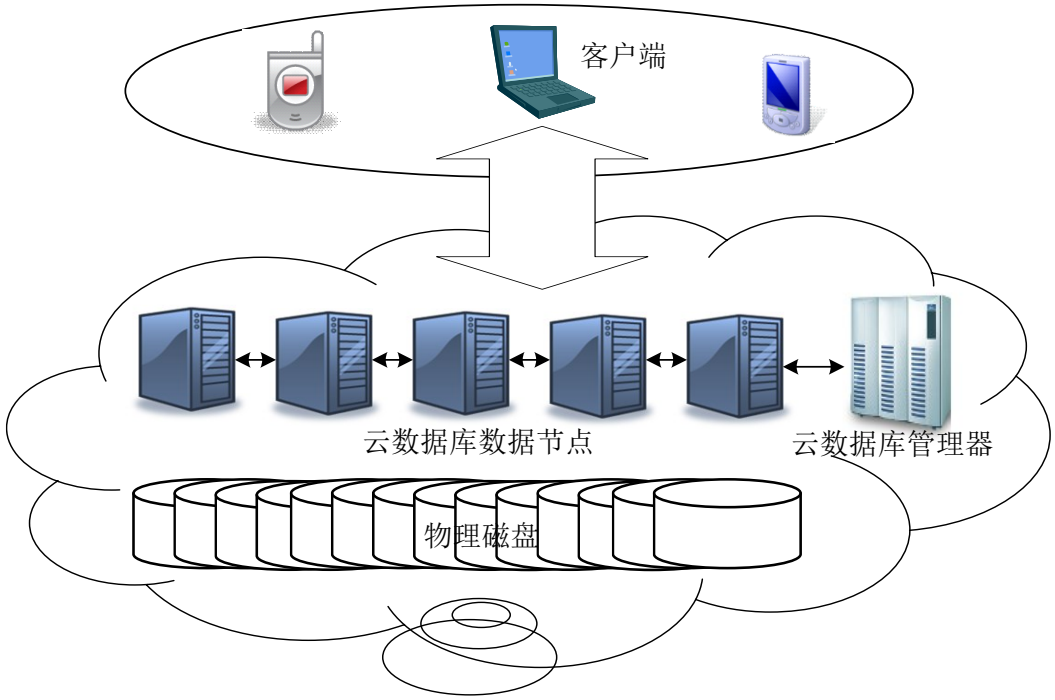

云计算示意图

## 云计算概念

•通过整合、管理、调配分布在网络各处的计算资源，通过互联网以统一界面，同时向大量的用户提供服务

## 云计算特点

# 云数据库概念




- 是部署和虚拟化在云计算环境中的数据库。
- 是在云计算的大背景下发展起来的一种**新兴的共享基础架构**的方法
- 它极大地**增强了**数据库的存储能力，**消除了**人员、硬件、软件的重复配置，让软、硬件升级变得更加容易
- **具有**高可扩展性、高可用性、采用多租形式和支持资源有效分发**等特点**

# 云数据库的特性

**云数据库特性：**

**（1）动态可扩展**

**（2）高可用性**

**（3）较低的使用代价**

**（4）易用性**

**（5）高性能**

**（6）免维护**

**（7）安全**

腾讯云数据库和自建数据库的比较

| | 自建数据库 | 腾讯云数据库 |
|---|---|---|
| 数据安全性 | 开发者自行解决，成本高昂 | 15种类型备份数据，保证数据安全 |
| 服务可用性 | | 99.99%高可靠性 |
| 数据备份 | | 0花费，系统自动多时间点数据备份 |
| 维护成本 | | 0成本，专业团队7x24小时帮助维护 |
| 实例扩容 | | 一键式直接扩容，安全可靠 |
| 资源利用率 | | 按需申请，资源利用率高达99.9% |
| 技术支持 | | 专业团队一对一指导、QQ远程协助开发者 |

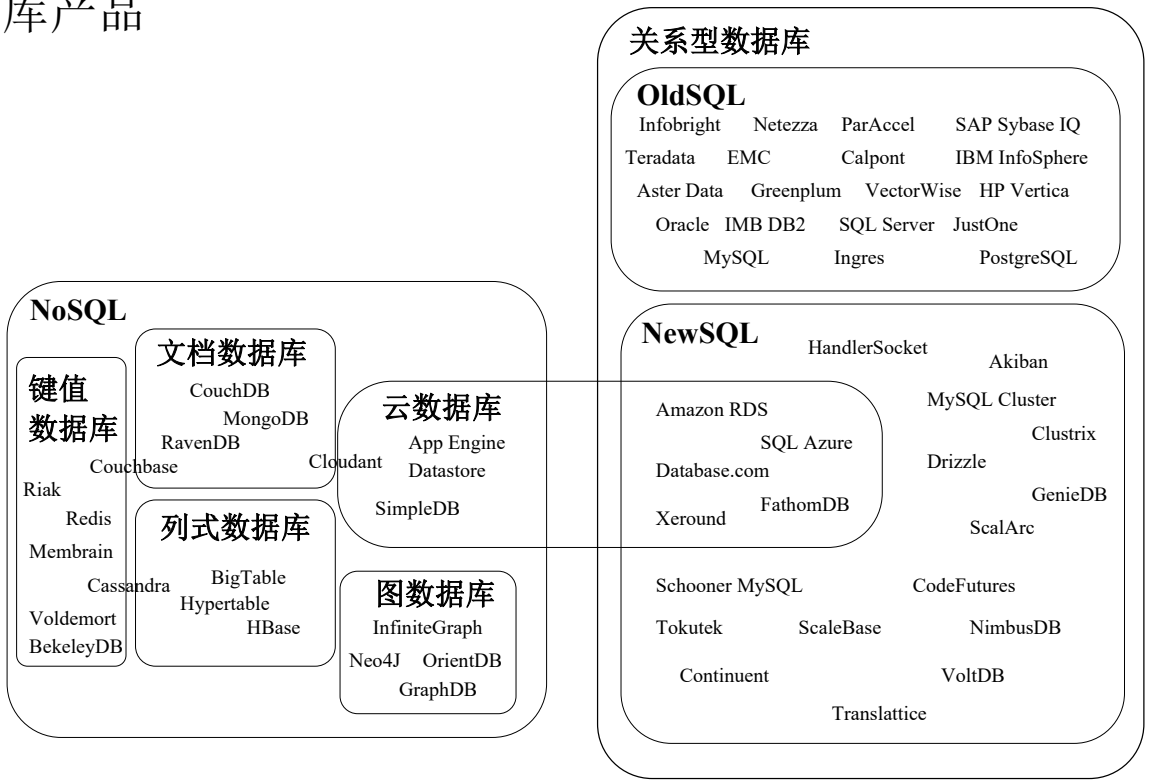
# 云数据库是个性化数据存储需求的理想选择

- 企业类型不同，对于存储的需求也千差万别，而云数据库可以很好地满足不同企业的个性化存储需求：
  1. 云数据库可以满足大企业的海量数据存储需求
  2. 云数据库可以满足中小企业的低成本数据存储需求
  3. 云数据库可以满足企业动态变化的数据存储需求

- 到底选择自建数据库还是选择云数据库，取决于企业自身的具体需求
  1. 对于一些大型企业，目前通常采用自建数据库
  2. 对于一些财力有限的中小企业而言，IT预算比较有限，云数据库这种前期零投入、后期免维护的数据库服务，可以很好满足它们的需求

# 云数据库与其他数据库的关系

- 从数据模型的角度看，云数据库并非是一种全新的数据库技术，而只是以服务的方式提供数据库功能
- 云数据库没有专属于自己的数据模型
- 同一个公司也可能提供采用不同数据模型的多种云数据库服务
- 许多公司在开发云数据库时，后端数据库都是直接使用现有的各种关系数据库或NoSQL数据库产品



# 作业

- 本次无作业，大家准备下周期末考试。