

\*\*服务器设计目标

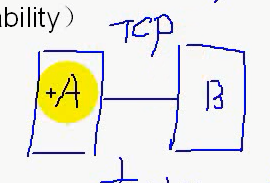


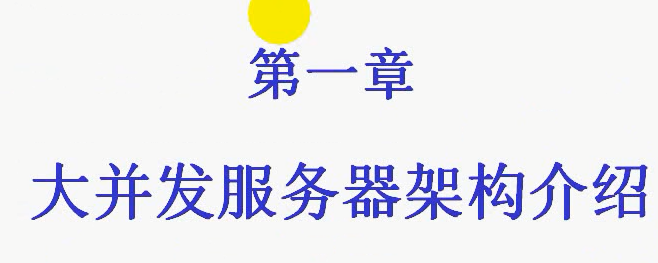
对于大量的并发请求 能及时快速的做出响应 ---- 要求编写出来的服务器 最大发挥机器性能 慢负荷情况 处理更多并发请求 并 做出快速响应

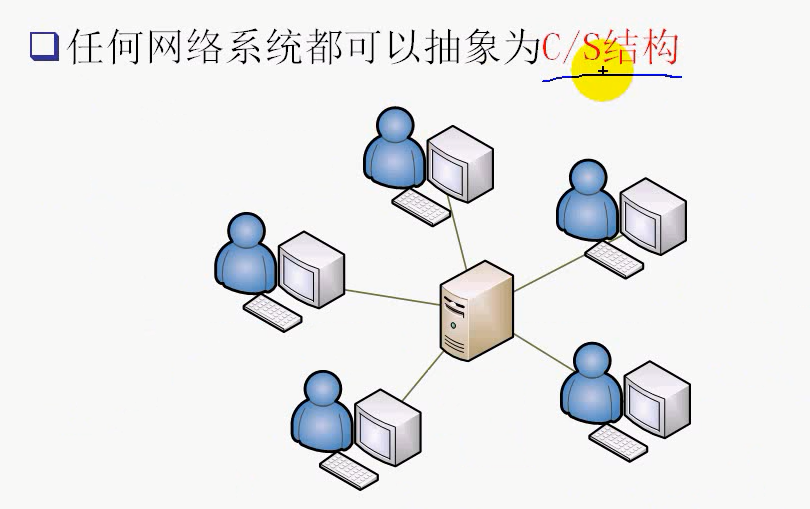
 就是保证服务器7\*24小时提供不间断服务 ---- 如果有一台服务器出现故障 也能自动转移到备用机 不需要人工的干预 这就叫failover ---故障转移

 服务器具有良好的框架 分层设计 业务分离 并可以进行灵活的部署

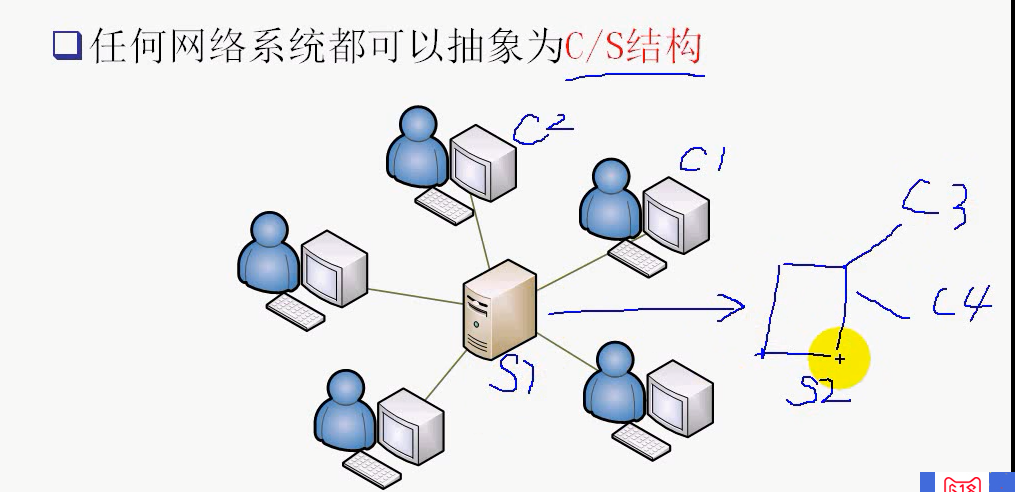
比如 某个服务具有A B两个主机 --- 这两个主机可以部署在同一台机器上 也可以部署在不同的机器上 这要求A B之间不能使用本地进程通信机制（比如共享内存） 应该使用**跨机器的进程间通信机制 --- TCP协议就可以**







一个服务端可以和另外一个服务端通信

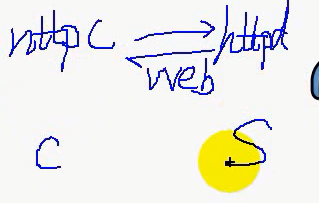


如果s1向s2发起了请求 那么 s1和s2之间就相当于c s结构

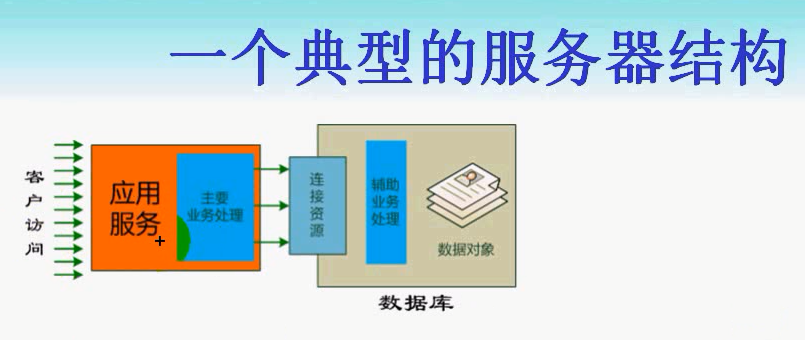
B/s结构也是特殊的c/s结构 ---- 比如IE 火狐 都是http客户端软件

http客户端软件都会向http服务器 发起请求 请求某一个web页面

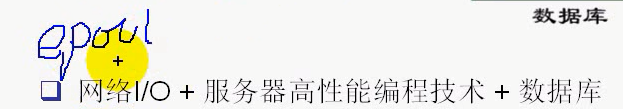
web就会在浏览器呈现出来

 通信的时候 使用的是http协议

任何网络系统都可以抽象为cs结构



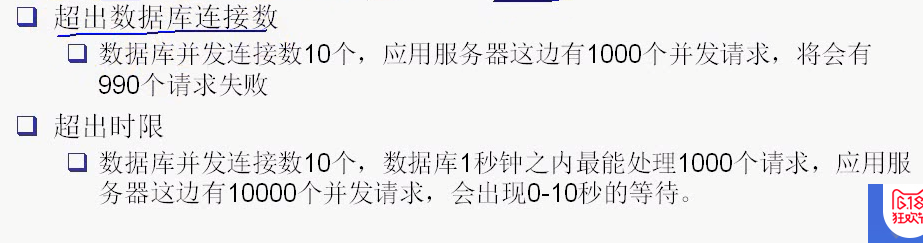
应用服务器要处理大量的网络IO

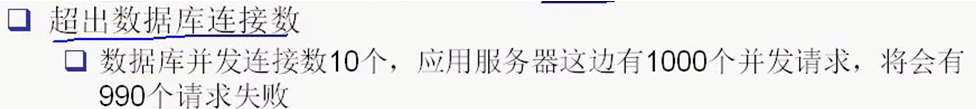


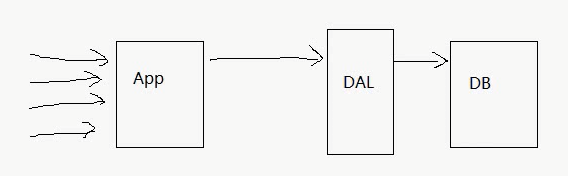
Linux中 搞笑的网络IO使用epoll编程

数据库常事瓶颈发生的地方 ------ 数据库可能出现瓶颈

比如：



 \*\*这个处理办法就是加一个队列 进行排队 --- 也就是增加一个**中间层** ---- 请求过来的时候 进行排队 中间层称为 DAL ----Data Access Layer

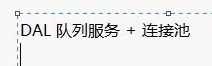


刚才说了 服务器的设计具有伸缩性 --- 这样DAL既可以和App Server在同一台机器上 也可以单独部署

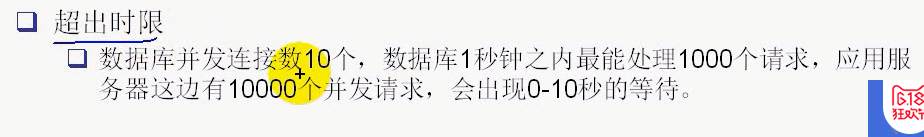
DAL 和 DB之前也是TCP进行通信

DAL 有一个连接队列 --- 大量的请求都要访问数据库 就可以进行排队

还可以做一个连接池 --- 也就是申请一些connection 放到池里面 ---下一次不需要重新创建connection 只需要从池里面找到已连接的资源进行响应 **加快响应**



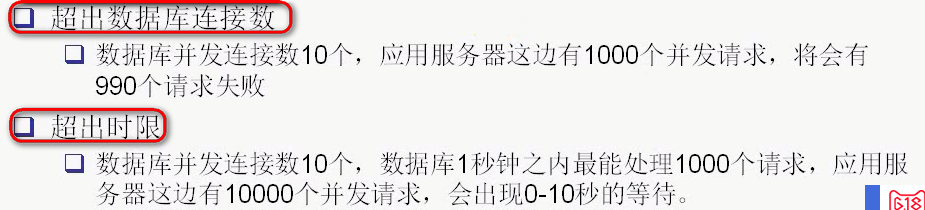
【伸缩 伸指的是部署到多台服务器 缩指的是多台能够合并到一起】



这样最后一个请求和第一个请求就相隔10s --- 如果系统规定的响应时间是5s 那么 这个系统就不能满足 --- 后面的请求会出现超时

这样 数据库的并发能力只有5000

【



首先是并发数固定 ---- 同时并发的量太大 ----- 瓶颈并发连接数不够

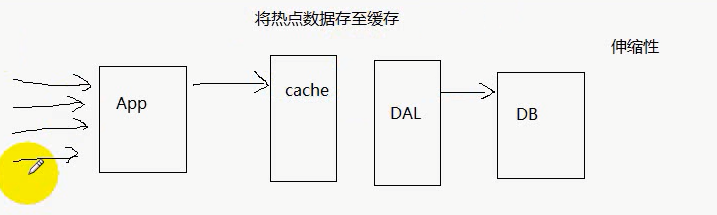
然后 就是 1s的时间段 --- 不是瞬时 ----- 更多的并发请求会有延时 延时太长 就超时】

如何降低数据库的压力？

\*\*尽可能将业务逻辑挪到应用服务那边 数据库仅仅做辅助业务处理 业务逻辑不应该太复杂 ---- 数据库上面处理逻辑---也就是占用CPU的计算 不如在OS 上业务逻辑来得高

---- 这是很有限的降低数据库的压力

\*\* 想法 是缓存 ---- 减少对数据库的访问



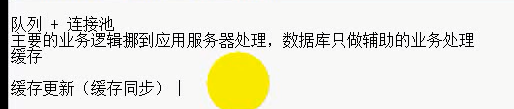
一旦使用了缓存 就有一个问题 就是缓存的更新 或者 缓存的同步问题

\*\*\*\*\*\*总结 增加数据库并发能力的办法

(1). 队列+连接池

(2). 数据库做辅助的业务处理

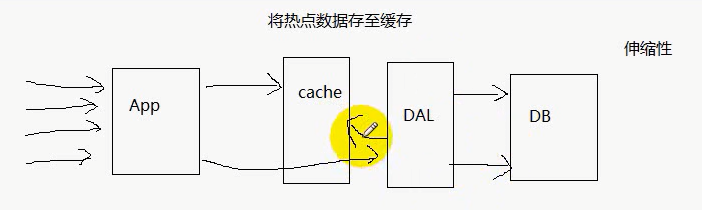
(3). 使用缓存 ---cache如何同步 或者缓存如何更新

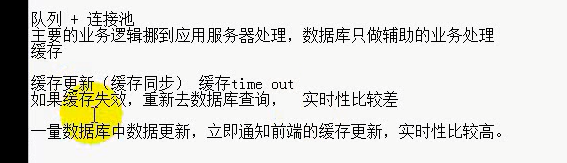


缓存同步的两种手段：

------ 时效(一定的时间会超时 如果缓存失效 需要重新到数据库中查询 查询到数据之后 再更新缓存 保持缓存和数据库中的数据同步 不是实时的同步 实时性差)

------ 某个业务改写数据库中的数据 然后更新缓存 ---这种就能够实现缓存即时更新 --- 实时性比较好 实现麻烦





---- 如果缓存存储的数据过大 就会内存不足 ----- 这时候 会把不活跃的数据放到磁盘中 这叫缓存换夜



换出的算法

先进先出

LRU--- 最近最少使用的缓存换出

LFU--- 最不频繁使用的换出



这些算法在OS课程中都有介绍过

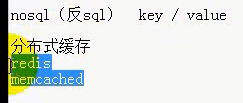
这样缓存的同步 缓存的换页 都有开源的产品 不需要自己实现

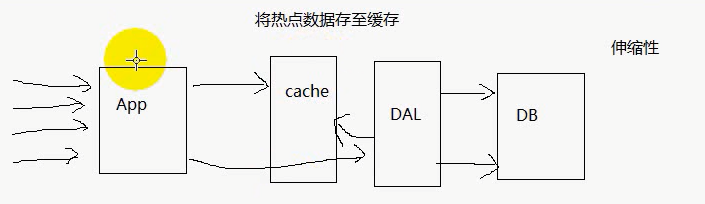
---- 比如 nosql 都具有这种功能

Nosql ---反SQL 主要用来存放非关系数据

关系数据存放的是对数据的一致性比较高的 key/value --- nosql可以当做缓存来使用 比如redis

Memcached 这些都是分布式的缓存 开源的软件

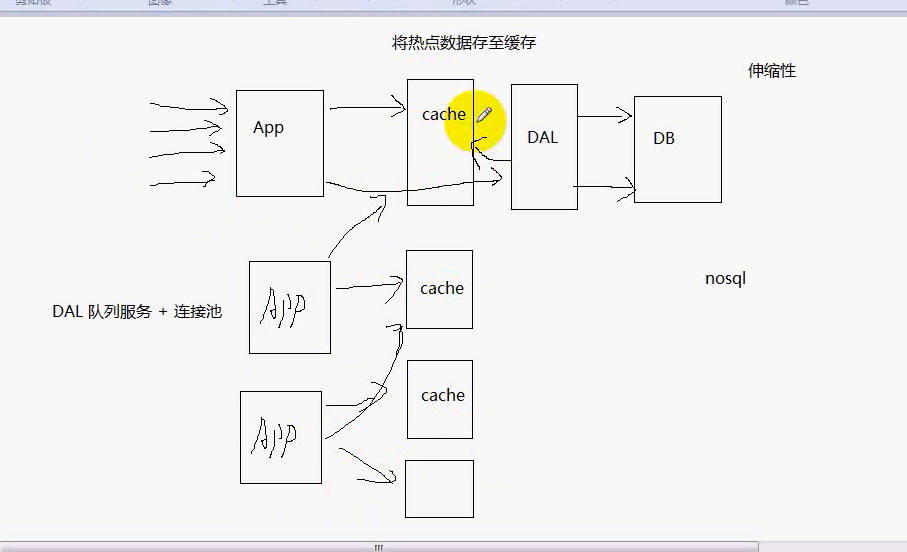




Cache可以使用开源软件

伸缩性 cache可以和app部署一台机器上 ---- 此时 缓存仅仅是局部的缓存 不是全局的缓存 --- 缓存麻烦

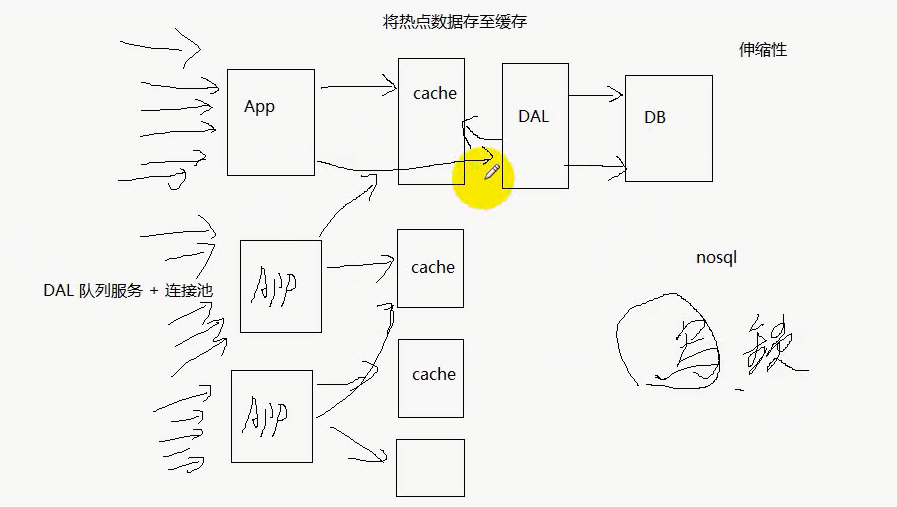
如果cache部署在独立的机器上 并且 分布式的缓存 这时候 缓存部署在独立的机器上面 各个应用服务器都能够访问

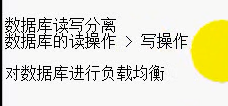


这样 就能够减少对服务器的访问 减轻服务器的负担

即使有缓存----当应用服务器足够多的时候 也就是要处理大量的并发 这样DB仍然会产生瓶颈 – 比如对数据库写的操作 把数据库锁住了 使得其他的读请求过来 需要等待 --- 读的请求被写阻赛了

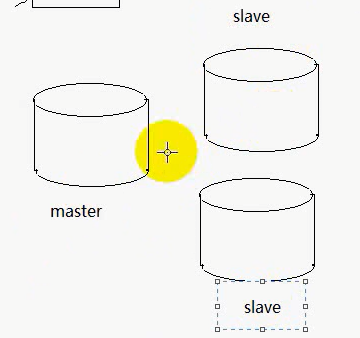
\*\*这时候该怎么办呢 ------ 这时候 进一步进行优化

 这时候 查询操作 比 更新操作来的多 所以对数据库进行读写分离

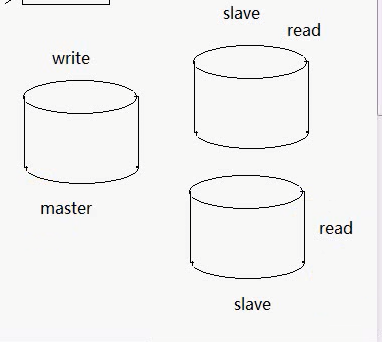


大多数数据库都有replication 机制 这个机制就可以负载均衡

采用主流数据库的replication机制 ---- 主从机制 master slave机制

 可以分配一些数据库只读 一些数据只写

主库只写 从库只读

 这样 DAL 数据访问层就复杂了

DAL 对于查询 访问读库

更新删除的请求 访问写库 ----- 一旦写库的数据发生了改变 应该同步到读库 这个就是用replication机制进行复制 ---通过日志文件进行同步

这就是replication机制

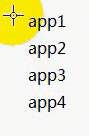
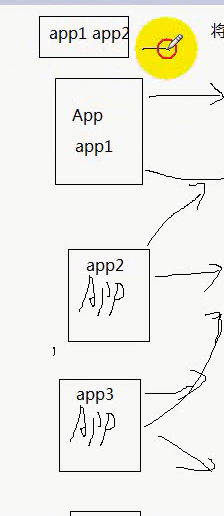
主从模式 进行读写分离 --- 一个DB服务器变成了多个服务器 ---然后 就负载均衡了

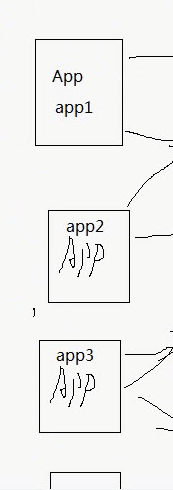
这样 DB的并发能力大大提升了

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

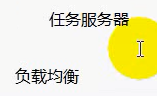
----- 这时候 应用服务器可能出现压力 这时候 增加应用服务器

 ---一个应用服务器可能有多种业务 也就是处理多种应用 app1 app2 app3 app4

 甚至可以业务分割 ---------------------  就是某个应用服务器仅仅处理固定的app

 还可以应用服务器可以处理多种应用的

这时候 前端大量的客户端 向应用服务器发起请求 ---- 这时候 需要做负载均衡 最好先经过一个任务服务器 --- **任务服务器也能够实现负载均衡**

 **任务服务能监控应用服务器的状态 在实现这些应用服务器的时候 暴露一个接口 --- 采用自定义协议 或者http协议 来暴露一个接口**

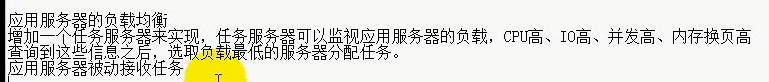
---- 也就是 这些应用服务器相当于服务端 任务服务器相当于客户端 ----- 任务服务器可以查询 或者监视当前的负载（到底是CPU高 还是IO高 还是并发高 还是内存换页高）



任务服务器查询到这些信息以后 选取负载最低的服务器分配任务(可能会有算法进行确认)

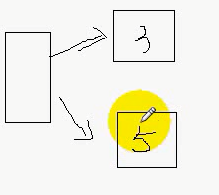


上面这种方案任务服务器主动把任务分配给应用服务器 也就是应用服务器被动执行任务

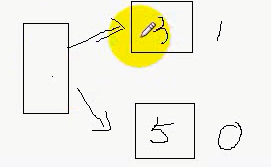


第二种 就是 应用服务器能否主动到任务服务器获取任务呢？

好处 就在于一个服务器处理完之后 说明是空闲的 --- 从任务服务器获取任务过来 这种负载均衡是最科学的

 两台应用服务器分别有3个和5个任务 ---- 按照任务服务的观点 任务服务器认为上面的应用服务器负担轻 ---- 就把任务分配给上面的应用服务器

但是 任务有各自的特点 5个任务很快处理完

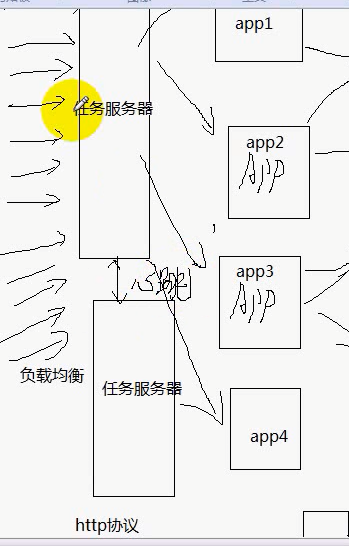
 这样 任务服务器就不能够很公平

如果应用服务器都处理相同的任务 应用服务器主动去任务服务器提取任务 这样好 否则 任务服务器实现难度大

这样都做完 就可以接受高并发了

但是 任务服务器可能出现故障 --- 所以 要能够故障转移 fail over

任务服务器有多台 ----- 任务服务器之间通过心跳来实现

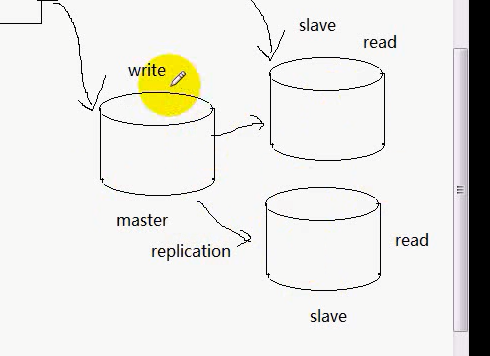
 这就是保证了服务器的高可用

----- 也就是一个服务器出现故障 能够转移到另一台服务器

----- 这里面数据库还可以进行优化

刚才做了数据库的读写分离

但是 对于大型的应用来说 每个数据库都非常庞大

 这三个数据库的量都非常庞大 三个数据库都是一样的 这时候 应该对数据库进行**分库分表 或者对数据进行分区**

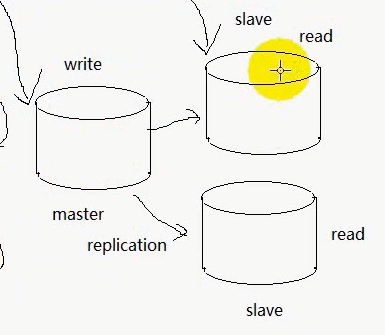
前面做了一次负载均衡 ---- 某几个做写 某一个做读 但是 如果每一个数据库的容量太大 就会影响到数据库的并发能力（数量多 查询慢 延时长 并发能力变差）

数据分区两种形式：分库 和 分表

分库：数据库可以按照一定的逻辑 把表分散到不同的数据库 或者叫 垂直分区

\*\*有用户相关的表 业务相关的表 基础信息相关的表

就可以把这三部分 分成三个库

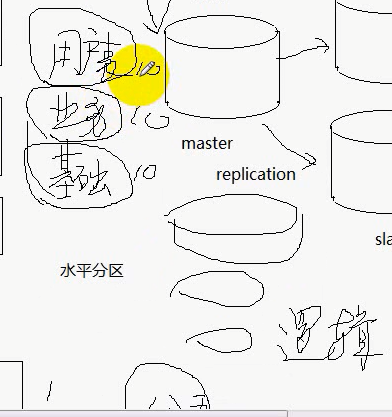
 这就相当于 有三个这样的主从库 这样数据库的容量减小了 这种切分叫垂直分区 数据访问曾的逻辑代码需要改写

更加常用的方法 不是这种垂直分区 常用的是水平分区

把数据库水平切割为若干的数据库 --- 切为若干个数据库

每个数据库都有 各种表 --- 只是记录分散到不同的数据库

每个表有10条记录 分散到10个数据库 这样 每个表中的记录仅有一条

 DAL如何访问哪个数据库 也需要进行改写代码

这种水平切分 就可以横向扩展数据库 ---- 也就是压力出现了瓶颈 就可以水平扩展 发挥更大的性能

哪一个层面出现了瓶颈 就在哪一个层面增加服务器 这就是分布式系统架构的思路

很多都可以使用开源软件搭建



服务器的高性能编程技术 有四点

服务器性能的四大杀手

**\*\*数据拷贝** --- 也就是减少内存的拷贝 ---- 同样是缓存机制

也就是应用服务器里面 还要有缓存机制 这不是分布式缓存 而是 应用服务器内部的数据缓存 --- 从os的内核拷贝数据到应用层

**\*\*环境切换** --- 针对的是线程的切换 ---- 所编写的服务器 该不该用多线程 是单线程好 还是多线程好？

==== 如果我们的服务器是单核的服务器 多线程未必能够提高性能 ---- 单核服务器采用状态机的编程机制 效率最佳

状态机编程：单核服务器 实际上是不能并行处理任务的

有大量的任务提交到服务器 这些任务实际上 即使使用了多线程处理大量的任务 实际上不能并行处理 ----- 使用了多线程 就会增加了线程间的切换开销 ---- 改成单线程有序处理完这些任务 就是每个任务占用一段时间片 一个线程处理 单线程 对单核服务器来说最佳

如果服务器是多核的 多线程可以提高威力

多线程可以充分发挥



多核服务器使用了多线程 也不能创建更多的线程 也是避免线程之间的切换

这也是性能的杀手 === 理性创建线程

**\*\*内存分配 可以增加内存池 减少向OS申请内存**

**\*\*锁竞争**  可以通过逻辑 避免锁的使用 避免锁的竞争