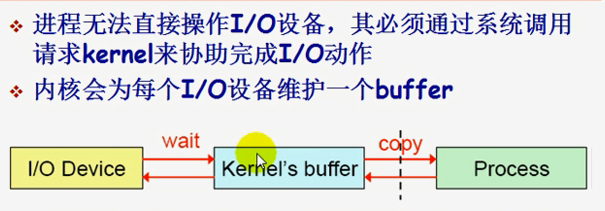
IO动作对于OS来讲 如何来执行？

为了OS安全 进程无法操作IO设备 必须通过系统调用 请求内核协助完成IO

内核在内核空间为每一个IO维护一个buffer

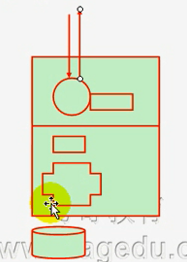


所以进程发起发起IO请求之后 请求的是内核的buffer IO设备很慢 所以 内核的buffer要等待IO的数据复制到内核的buffer

[copy指的是 我的进程无法访问内核的内存空间 必须由内核把buffer中的数据复制到进程的内存空间中]

一定是进程请求一个IO资源（WEB服务器而言 是请求一个硬盘上的页面）

进程想内核的buffer发起申请 此时 进程要处于等待状态

 从外部http请求过来 到 最后进程对http请求进行相应完成 有几个等待的状态

\*\*无论如何 用户发起请求 一定是等待状态 ---- 浏览器必须等待

\*\*进程向内核的缓冲区发起请求 在内核的缓冲区准备好之前 必须等待

\*\*内核接收到请求 由IO设备吧数据复制到内核的缓冲区中 内核必须等待（图中的wait）

\*\*最后 进程要响应给外部请求 必须有数据 所以 要等待内核把数据复制到自己的内存空间（图中的copy）

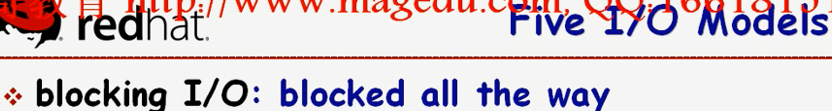
对于进程而言 就有两部分等待





等待模式的不同 这两段弧线的等待 可以有不同的等待模式





阻塞式IO



非阻塞IO



IO多路复用

IO复用的原因 ---- 就是一个进程要响应多个请求 否则无法完成此类的处理

只要有两个IO就必须要复用

比如 一个进程同时处理多个请求 必须复用

一个进程同时处理多个文件描述符 也必须复用

一个进程处理一个请求 也得复用 – 原因是用户的数据可能交互式交互 + 处理网络连接

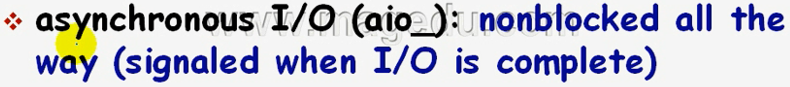
接受用户的数据 ---- 本地键盘输入的数据 ----- 本地磁盘IO

网络连接 ----- 网络IO

处理了多个IO 就必须复用



信号驱动IO

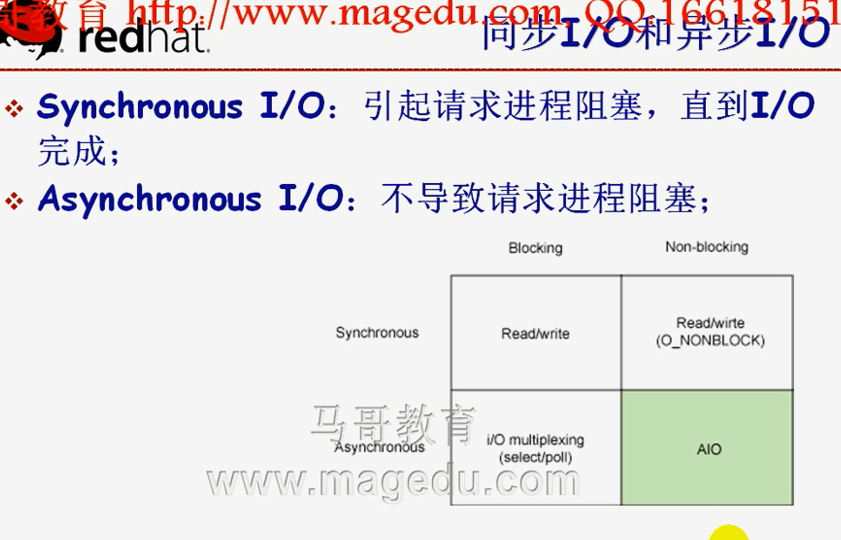


异步IO

这五种IO模型

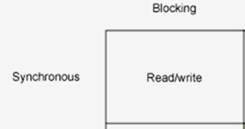
-----

下面是这四种IO的组合方式

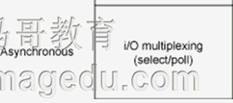


这样 一个blocking io可以以同步和异步方式执行 non-blocking io同样也可以以同步和异步方式执行

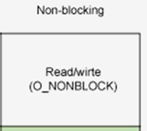
所以 同步阻塞IO 是通过系统调用 ---- 也就是read/write完成系统调用



而异步阻塞IO 就叫做IO复用



同步非阻塞IO 使用r/w 但是是O\_NONBLOCK来完成

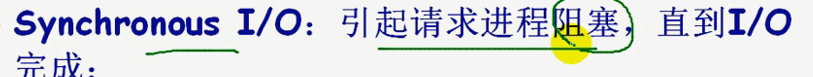


异步非阻塞IO 就称为AIO 也就是异步IO



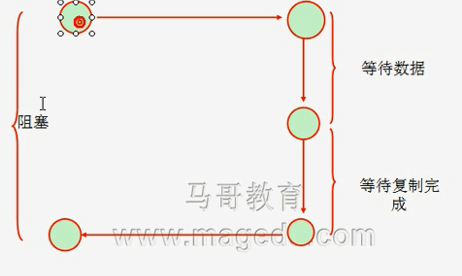
只有非阻塞IO才成为异步IO

------

---- 请求会引起请求进程阻塞 知道I/O完成



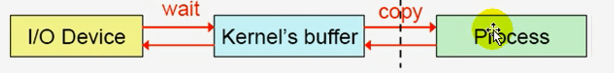
左边client 右边server

 左边的client是进程 右边的是内核 两端分别是 内核请求IO 把自己的缓冲区填满 然后把自己buffer的数据复制到进程地址空间上 这个过程 右边的进程一直没干别的 处于阻塞状态

非阻塞 ---- 在内核数据没有准备好的情况下 进程处于询问状态



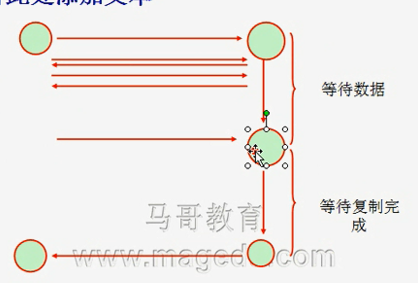
一个数据从IO设备到进程 一定分成两段



比如饭准备好了 但是 没有端上桌 还是不能吃 端上来之前 还是要处于阻塞状态

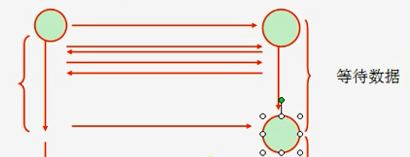
非阻塞压根就不知道数据未必准备好

所以 在数据没有准备好之前 就隔一定时间 就去问问数据是否准备好



但是数据准备好之后 这个复制过程 就不会再询问 这就好比 询问服务员 饭准备好没 为了好几遍 服务员说准备好 就不会再问了 然后 在等待复制完成 也就是饭端上桌的过程 进程就不再询问了

虽然是叫非阻塞 但是后面这段进程没做啥 就是等 所以 后面是阻塞的



这个前半段是异步IO ---- 

之所以成为异步 就是不知道对方何时准备好

同步就是 我就一直盯着你

同步就是闲等 ---- 阻塞就是盯着

异步就是忙等（一会就做点别的 到点就去问问）

这个整体叫做非阻塞

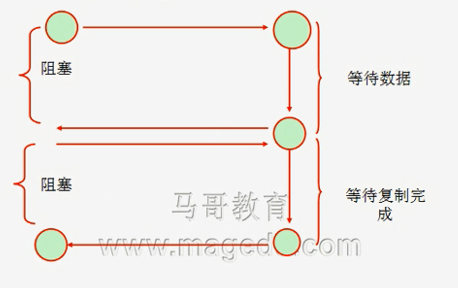
--------

IO复用 --- 也就是异步阻塞 ---- 这个叫异步的原因 就是这个两端是分开执行的

 ---- 定义说的很清楚 就是wait和copy是分别阻塞

指的是 数据通过系统调用像内核请求状态 前半段阻塞 数据准备好了 之后 再次发出一次系统调用 再阻塞一次

两端 都是独立执行 每一段都阻塞

 ---- 就是在中间 又一对箭头

别人通知我的数据完成了 我可以等一会再复制数据（等一会的过程可以干别的 这就是异步的理解）这两段并非一气呵成

这就是异步阻塞 也就是IO复用

----- 信号通知的IO



就是等待数据的过程 进程无需阻塞 完成 内核会通知给你

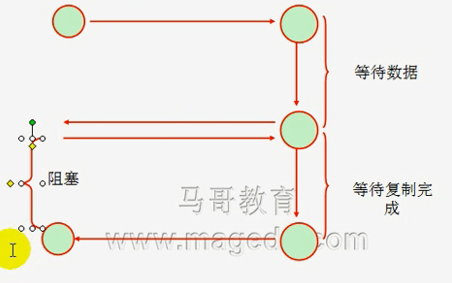
前面的非阻塞 就是要一遍遍询问 就是忙等 浪费系统大量资源

现在解决这个问题 不用忙等 该干啥干啥 有别的请求 你就去响应 但是数据准备完成 会通知你 而不会复制数据

数据准备完成 要求进程再次发起一次系统调用 完成数据的复制 ---- 进程自我主导数据的复制

事件通知的这种 就是异步的 ---- 而且前一段也是非阻塞 但是不需要隔一段时间就来询问一次

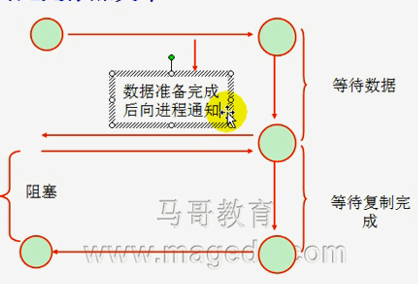
但是 复制期间 仍然是阻塞的



信号驱动IO-----解决了IO多路复用（异步阻塞）的前半段不用阻塞了

-----解决了 同步非阻塞的前半段不用轮询了

这就是事件驱动

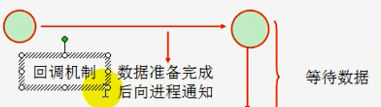


通知几次？通知多次 知道进程看到响应 ---- 这就是通知的水平触发

只通知一次 看不看到自己管理 ----- 这就是通知的边缘触发

IO复用机制简单 但是 两端阻塞 阻塞期间性能差

事件驱动 前半段不限制 性能提高了 前半段 实际上就是进程留下来一个联系方式 最后内核使用这个联系方式来通知进程数据准备好了 这个也叫回调机制

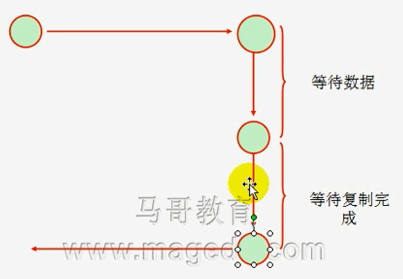


发起请求 ---- 留下联系方式 ---- 完成 ---- 进行回调（回调就可以理解为留下联系方式）

信号驱动也是一种IO复用 – event driven

最后一种是异步IO ---- 非阻塞又异步 ---- 真正意义上的异步IO

整个过程都不阻塞 数据也复制到内核缓冲区了 内核缓冲区 也送到进程地址空间 进程怎么知道？

 ---- 这个就是数据从准备好 到 最后复制到进程地址空间完成 然后发一个信号

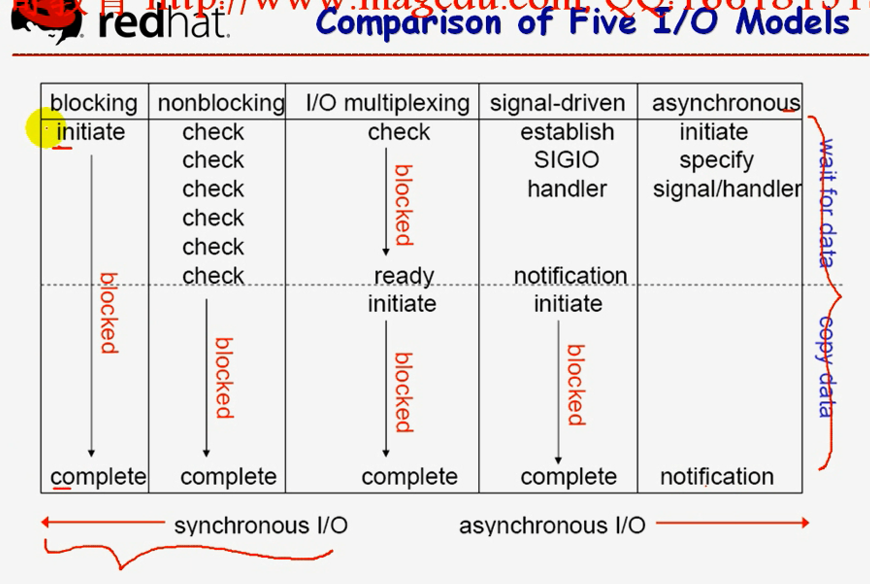
这个过程 进程爱干啥干啥

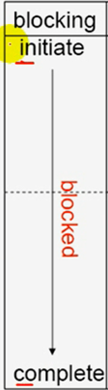
必须向对方发起通知 --- 也就是向对方传送了一个信号 --- 饭端上来上 拍桌子唤醒你

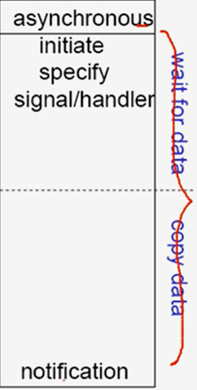
信号驱动IO和AIO存在两种信号 一个中场通知 另一个是最终通知

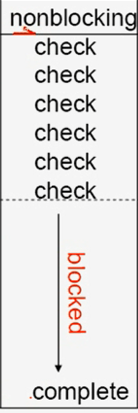
这个就是异步IO

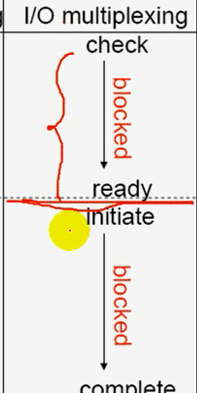
放在一个图中总结一下



 ---- 可以看到 从初始化 到 最终完成 只能阻塞 等待 ---- 同步

 ---- 初始化完成 就 指定了一个signal/handler 这个就是联系方式 最终完成 就通过这个来联系外部进程

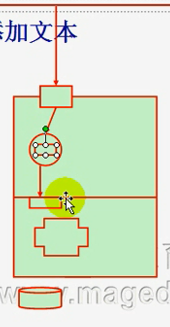
 ---- 同步非阻塞



 ---- 信号驱动 前半段会留一个联系方式

但是 整个过程 外部网络请求 一定是阻塞的 –也就是浏览器一定是阻塞的

如果进程工作同步阻塞模型下：

 ---- 从磁盘到内核空间 以及从内核空间到进程空间 进程什么都做不了 这就是同步阻塞

如果工作在同步非阻塞 ----- 非阻塞的前半段 要干别的事+check 也就是可以接受其他用户请求

如果接受了2个请求 ----- 如果两个都检查 有一个buffer完成 那么这个线程就进入第一个线程的阻塞 阻塞过程无法做别的事 完成之后 再检查第二个IO 然后 xxx 这个进程忙死了

接收了多个请求就会很差 就算响应一个 性能都没有同步阻塞方式好

第三种方式 IO复用

两段都阻塞 中间一段很短的功夫 可以自己决定 中间空闲时间不多 这样一个进程响应多个请求忙不过来 但是 可以多个进程响应多个请求 ---- 主进程监听在套接字 监听到了 分配子进程进行响应 这样就可以了

采用了IO服用这种机制 ---- 也可以一个进程多个请求 但是 两段是阻塞 所以 不好

我们的Apache在IO复用的过程 是 一个主进程 + N个子进程来响应多个请求 利用了IO复用模型 借助多个子进程来响应多个请求 --- 利用了IO复用模型

IO复用模型下 常用的函数 就是select 任何一个进程要想知道内核是否为自己的数据是否完成 ---- 头半段要阻塞 ---- 就意味着要时刻监控这个buffer ----- 打开一个文件 buffer准备好 就会通过这个文件知道准备好

父进程生成了多个子进程 每一个子进程都要打开一个文件 一个父进程自身 select最多只支持打开1024个文件 所以 最多只能接受1024个连接进来

Poll()不限制1024 但是机制和select()一样 就是阻塞 所以性能一样差 不限制 性能会越来越差

这就是IO复用机制

下面事件驱动机制 就意味着

用户请求来了 进程留下一个电话号码 该干啥干啥 这样 进程会接受多个请求 对内核留下多个联系方式

基于事件驱动可以一个进程响应多个请求

这个进程不需要切换 因为用了事件驱动 但是select/poll都实现不了

在linux中叫做epoll 在solaris是/dev/poll 在freebsd上 是kqueue

这三种都实现了事件驱动的IO复用

事件驱动的IO复用 实际就是一个线程完成了多个请求 一个线程内部维持了多个连接 而且每一个连接的资源

只是告诉那一段数据准备好 就是阻塞一下 完成copy 这个过程快

如果内核的缓冲区共享给进程 那就没有copy 这个更快 这个就是**内存间（共享）映射机制**

以前的内存映射：是MMAP

为了访问一个文件 必须把文件的数据流到 buffer 再到 进程空间

立即告诉你 准备好了

只要连接建立 关联建立 准备好了 就可以立刻来获取

异步IO

本来是分两段

异步IO就是复制完成 再打电话

这个过程 进程该干啥就干啥

这时候 数据是现成的 异步IO实现相对复杂 ---- nginx就是实实在在实现了异步IO

异步IO消耗的资源会更少

刚才的信号驱动IO 后半段的数据 还是阻塞状态

异步IO没有任何阻塞了

但是 异步是非常复杂的机制 真正实现异步是非常复杂

用户连接来的网络IO是不能异步的 本地存储的磁盘IO是可以异步的

Nginx的特性就是支持文件系统的AIO 支持内存映射mmap 支持事件驱动event-driven





使用了事件驱动的异步架构

所以 比起来传统的使用的select的Apache nginx能处理的并发连接数非常大的

解决了C10K

单台的nginx 处理3-5w没有问题

Apache在 httpd的mpm

Prefork worker event



仍然是线程处理请求 只是进程切换少了

 由于httpd有很多模块 所以比nginx大很多

Nginx不能比apache的 就是功能没有apache多

所以 这就为什么httpd还占有很高的份额的原因

Nginx不断蚕食httpd

Nginx很少做WEB服务器

基于一个线程响应多个请求 固然很好 但是有一个问题 如果线程崩溃了 N个请求都崩溃了

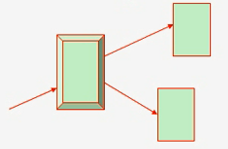
但是如果是多个进程-多线程的模型来说 一个进程崩溃了 并不影响其他线程上面用户的请求 ----- 所以apache的prefork要稳定的多得多

==🡺 所以 通常让nginx做反向代理

请求mysql-proxy本身不是mysql服务器 可以相应用户的请求 也可以工作在3306端口上 但是 可以把用户的请求转发到后端的MySQL数据库服务器上面

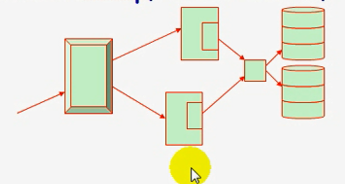
nginx就是一个web服务器的代理 nginx就是一个WEB代理服务器 只能解析HTTP协议 这个称之为**反向代理**

所以 nginx常常作为前端的反向代理 Apache作为真正的web服务器



如果提供LAMP --- 后端就安装PHP 在后端提供MySQL 一个MySQL担心不够 所以 主从+ 读写分离 ---- 读写分离就要用到MySQL proxy

由proxy把请求读写分离到后面的MySQL服务器



这个是动态的内容

我们静态的内容 web页面 放在哪里呢？

动态内容？

Html css js都是静态的 ---- 谁来响应？

Php脚本都是动态的 ---- 谁来响应？

动态静态都放在apache上面 解决两个Apache文件同步的问题 就是inotify + assync

共享存储 nus或者sun都可以 只不过使用sun 要使用集群管理系统

网页内容 不允许用户修改 也不允许用户上传附件 但是允许用户发帖 保存到MySQL

网页页面文件不需要改变 直接放到几台Apache上面就可以了 MySQL就可以了

我们有说过 Apache对静态内容的响应 没有nginx强【前面就一直讲的就是 磁盘文件 到响应 nginx比apache强】

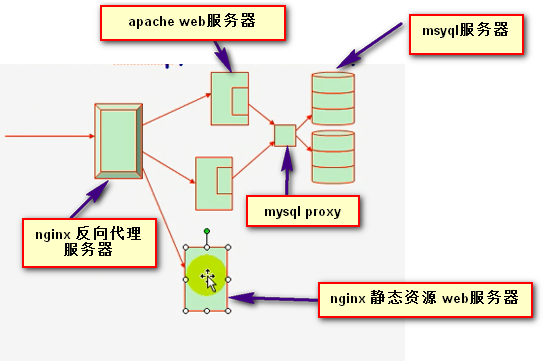
所以Apache就是处理动态内容的同时 还处理静态内容 这样会非常吃力

该如何？静态的放在nginx前端

此时nginx既做反向代理 有做静态资源的web服务器 这样nginx会很累

该怎么办？可以在后面专门放一个用户处理静态文件相应的nginx web服务器

所有的静态资源都从这个nginx web服务器上面读取

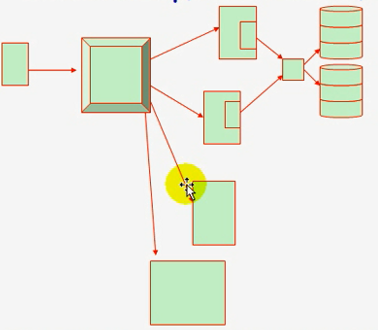


问题是 nignx如何知道 静态在下面 动态再上面？

Nginx本来就是反向代理 一定能理解用户的请求

如果请求 php就转到上面 但是 请求.jpg png css js 等 就走下面的nginxweb静态服务器

如果需要用户上传的附件 一般都是静态文件 压缩文件 或者图片文件 --- 比如论坛 此时 可以专门找一台服务器 让用户上传 工作在web协议让用户上传 工作在ftp协议让用户下载 在一个页面上传 但是传到另外一个服务器了 这正是因为可以正确理解HTTP协议

 下面那个是存放用户上传的静态文件的服务器

如果放一个lvs director 就不能区别php和静态文件了

正因为web的反向代理 能够理解用户精确执行的这种协议 所以完全可以做到对用户请求地址的重写 完全可以实现资源的重定向等各种操作 还可以实现url重写

明明请求一个站点 可以重定向到别的站点

Apache也可以由模块实现反向代理

无论哪一种web服务器 相对于动态内容而言 肯定是响应静态资源的速度要快 但是 对于动态内容 要执行脚本 要在CPU上面转N圈 才能返回html文件

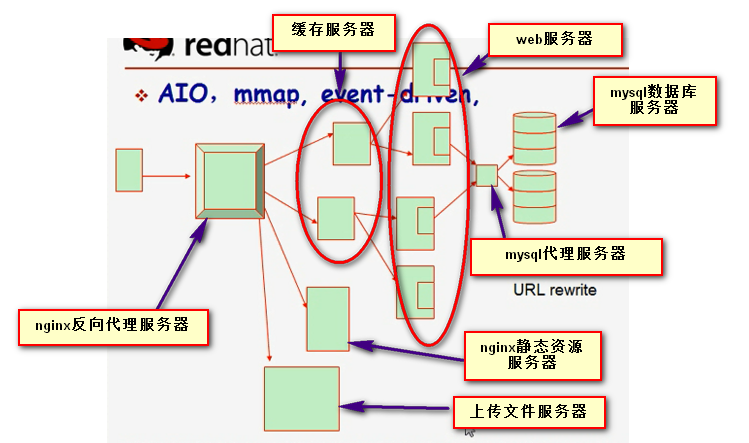
静态内容的请求 响应图片 如果图片的内容不是特别大 5-10K 可以在1s中响应5K-1w个并发 --- 取决于带宽 磁盘IO

如果用户请求的大量内容都是动态的 网站还会很慢

比如浏览一个淘宝的页面 用户多次访问 没必要一次次重复生成

所以 可以在生成之后 缓存起来 这样 以后的请求都是获取缓存中的静态内容 会快很多【同时 缓存起来的 都在内存 比磁盘IO还快】--- 这样就可以添加缓存服务器【通过360天气什么 缓存其实就是缓存字符串就是了 html也是字符串 所以 缓存可以扩展到任意格式的内容 比如缓存xml 缓存json】---- 这就可以**添加缓存服务器**

生成以后 缓存下来 就是静态内容 ---- 可以缓存web对象 可以加很多



本来每一个web服务器并发数2K 这样 每一秒动态请求的并发数是8K

但是加了缓存之后 只要缓存策略设计的好 就算后端店铺改内容 也没有改的很频繁 所以能够缓存下来 只要你的缓存空间足够大 可以降低后端的请求降低的很大

这样一来 缓存本身都是静态的 这样响应速度非常快 我们的web服务器就大大提升了

这样我们的服务器的量 就变成了四台服务器 有时候 多达几十台服务器

如果用户请求没有的话 一定走web服务器

如果并发达到5w

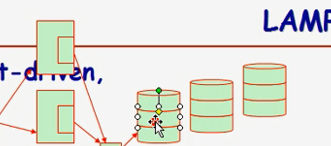
每一个请求只需要2s完成 这样 5w\*43200 =20亿个请求 这个规模就非常大了 这个是一个理论计算 带宽也未必那么大

如果我的读取量很大 很多内容要更新 所以 后端不停地到mysql读取数据 读写比例5:1 一个MySQL实在扛不住 在没有使用数据库连接池的情况下 就算是用Innodb或者Myiam引擎的情况下 能连接进来2K个就顶天了 数据连接和前端的web连接是两码事 一个select就会发送大量的数据

Select \* from table ---- table 10w行 数据很大

大量的都是读请求 该怎么处理？

可以一主多从的模型 加几个从服务器



每一个查询都要从服务器返回 性能很差 mysql执行一次查询 两个查询并行 500个查询并行 ---- 每一次查询 磁盘IO 、cpu占用大量的时间周期 ----

一个MySQL的查询可能几秒 ---- 磁盘IO一并行 都受到影响 ---- 每一次都磁盘IO会很慢

所以 MySQL也加缓存

MySQL本身就有缓存 --- mysql既要负责查询 又要负责缓存管理 ---- 这样MySQL的性能就会受到影响

更不要说 缓存要不停的回收内存+分配内存 ---- 一个组织 忙N件事 --- 怎么办？

对MySQL前面再加缓存 在MySQL读服务器前面加memcached服务器

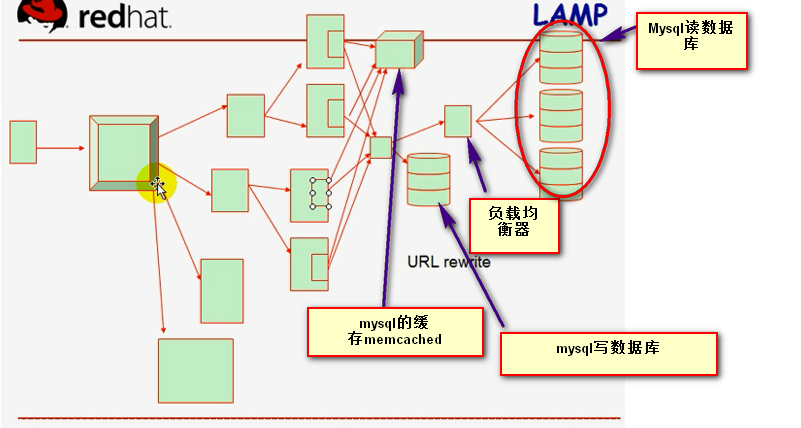
Memcached没有 就到MYSQL取 -----

如果读写分离非常多的时候 怎么办？---均衡的效果不好 –要加一个附在均衡器 ---- 又多一台主机

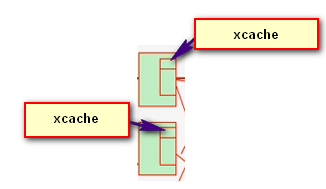
所以 我们的应用程序 提交SQL

Mysql-proxy发现是读数据 就提交给负载均衡器 负载均衡器查MySQL数据库 --- 第一次 --- 查到之后 应用程序指挥把查询出来的数据放到memcache中 ----- 自己缓存到Memcached中

第二个客户端发起请求SQL语句（也就是web服务器发出的）----- 注意 每一个应用程序先去memcached中去查数据 ----- 所以 mysql的缓存不是工作在mysql-proxy后端 而是工作在web应用程序后端 --- 如果Memcached没有 开始请求mysql-proxy --- proxy发现是读 就交给负载均衡器 ---- 负载均衡器挑选一个 ----- 计算结果逐层返回 返回结果会先缓存到memcached中 ---- 再响应



到PHP4.0之后 要先编译成opcode以后才执行 然后才能执行 所以 每次用户请求 都要先编译 这样就会很慢 ---- opcode也可以缓存 ---- 可以使用xcache ---安装到每台web服务器的上面 xcache不用服务器 可以安装成php的模块



如果要高可用 HA

前端所有请求都从nginx反向代理服务器来 所以 要高可用

Mysql proxy 高可用

Memcached不用高可用 坏了就坏了 无非加一台

当然Memcached无法高可用 因为Memcached不支持节点之间的通信 就是 缓存数据无法共享

你看看Apache用来处理动态内容 不处理静态内容

那么就不需要Apache 仅仅使用php就可以了 ---- fastcgi 自己做成服务器

Fastcgi动态进程 的生成 还有优势

修改如下

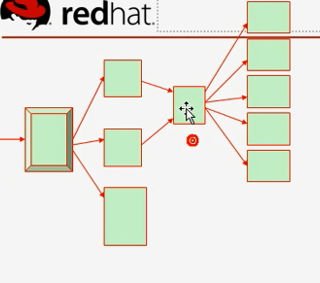
静态内容 定位到静态内容的服务器

动态内容 提供给varnish缓存服务器 缓存动态生成结构

既然后面是多个web服务器 那么 就应该让varnish转发请求到后台多个web服务器 越平均越好 ---- 使用Lvs负载均衡 varnish也可以负载均衡 但是 算法没有lvs的好

也可以使用ha-proxy 这个也是工作在7层的

将web请求交给负载均衡来



上面两个是varnish

最下面的nginx静态资源web服务器

但是 这个负载均衡要能够监控这个后面这些服务器的健康状况

后面的都是独立的应用程序服务器 --- 这样应用程序服务器响应的请求就平均了

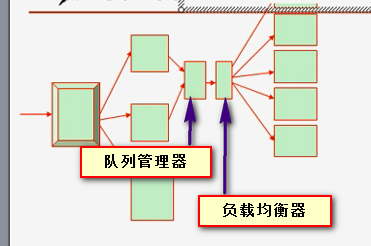
有时候有这样一种情况 就是我的应用程序服务器只能响应500个请求 现在一下子进来了5000个请求 ---- 此时服务器刚刚建立起来 就到了高峰期 ----- 我们的缓存设计的不好 命中率很低 所以 请求不断地涌进来 ----- 平均下来 一个就是1000个 这样 就会直接拒绝服务 此时该怎么办？

如果这个时候 允许所有的用户一下子涌进来 是非常不好的 所以 要考虑我们的系统的实际承载能力 --- 就算是拒绝一部分用户 也不能因为让所有的请求涌进来导致所有的请求都无法响应的这样的结果

可以做连接数限定 每一个服务支持多少个链接 --- ha-proxy支持这种设置

也可以在负载均衡和varnish之间添加一个队列管理器 如果允许异步执行 就可以加队列管理器 --- 可以接进来很多请求 ---因为是异步的 所以 不需要吧请求直接交给后端的server –于是来了5000个 但是 就给每个主机发500 ---一旦web主机空闲 再发 ---- 这就是异步消息同步的平台 --- 这个就是队列管理器 ---rabbit MQ 或者zero MQ

其实 队列管理器 就是一个连接池



不是马上给你提供服务 而是先给你缓存起来

这在异步应用程序中 这个队列管理器是非常常用的

所以 最好处理的都是静态内容

尽管都是静态的 如果都是淘宝这样的站点 一天图片的增加量是非常大的 一天几十个G或者上百G都是正常的 几年之后 多少个T

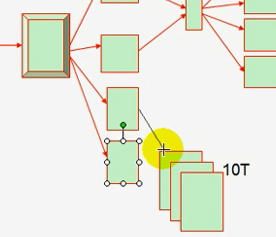
用户请求一个图片 都到这个服务器上找半天 要找多久啊！找一个图片 是非常慢的

这该怎么办？

图片量大 100-20法则 --- 2-8法则 也就是 100个里面 只有20个处于活跃的状态 ----- 既然如此 数据会有局部性 ---可以使用缓存 --- 程序 数据都有局部性

所以 在图片服务器添加varnish缓存 --- 装上很大的内存 + 很大的固态硬盘 500G硬盘+64G内存

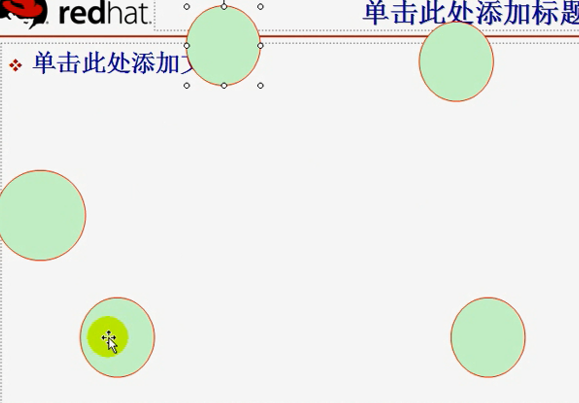
经常被访问的图片 直接缓存在varnish上

 --- 实现了用户请求大数据的快速返回

但是 面对淘宝这么多用户量的时候 平时没有问题 但是一抢购 就全挂了 一个系统是绝对扛不住

但是 只要缓存率足够高 所以 只要是静态 我们就可以使用地理位置法则

在全国各地的机房中 华北 东北 华东 西南 华南 各自建立一个缓存服务器 --- 分别都是一堆varnish集群



把图片数据 都直接缓存到用户的家门口 – 用户使用[www.taobao.com](http://www.taobao.com)去访问站点的时候 使用智能DNS解析器 比如 来自华北区的用户 直接解析到华北区的缓存服务器 90%从本地返回

但是 抢购 不是静态的 是动态的 ---- 打开页面的过程是静态的 静态化 使得打开压力不大

但是 抢购的时候 请求都过去了 ---- 所以 此时mysql集群要设置的足够大 –分区要足够好

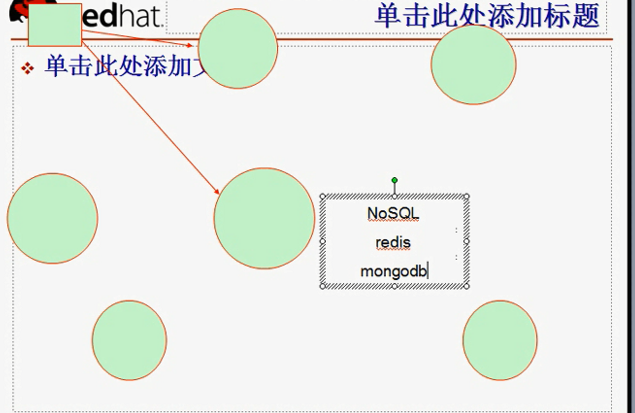
淘宝那边 抢购的时候 mysql的TPS 也就是每秒的事务量5w ---- 一个MYSQL为了查询读操作 数据量不大 一个mysql接受500个并发连接 都会扛不住

5w个写操作什么概念？问题是 一个商品是有限的 你抢完 别人就抢不到了 所以 数据要随时向前端更新 ----- 这时候利用MysQL解决不了问题

这时候 就要在前端的内存中执行事务 而且都是小事务 每一次抢购 就快速执行一个小事务 –快速反馈 商品也完成快速更新 ---- 这时候我们要借助于能够在内存中支持事物的服务器 --- 且对于约束法则不是特别严格的场景 ---- 此时NoSQL就派上用场了

Memcached本身可以缓存 但是不能保存数据

比如NoSQL中的Redis MongoDB 等等



尤其是Redis 作为**程序计数器**的情况下 就是一个产品 新浪微博 多少个人转载 都有计数 所以每操作一次 就要更新一次 所以 如果是使用 【所以 我们的360s的限量操作 不能使用Memcached 要使用redis ---- 这个盛先竟然开始让我使用Memcached】

都不是在MySQL 中操作 都是在redis 中进行 --- nosql在内存中工作 --- 还可以将数据同步到硬盘上去 –持久存储

还可以将数据转存到MYSQL上面去

所以 既有持久存储法则 又有快速响应手段

所以 计数工作 常常使用redis实现

对于 事务的支持 写操作比较多 可以基于mOngodb实现 但是有人说mongodb不是很稳定

我们的图片10T 如果分析一下图片 比如 去年增加了多少 前年增加多少 都在内存打开 空间不够

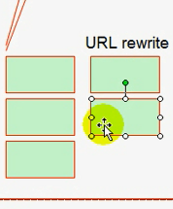
还有 我们的web服务器 有很多日志存放到日志服务器

可以将日志放到Mysql数据库中 放到有更高能力的IO上面 存放之后 可以导入到分布式FS上面

Facebook 可以每一个服务器上独立管理日志

把日志存储到分布式文件系统服务器

然后 在这个分布式文件系统服务器 可以并形程序分析

 【典型的就是HDFS MapReduce】

所以 日志量这么大 我就使用grep awk 分析大半年 也分析不完

总结不出来

这个时候 就是Hadoop就派上用场了 Hadoop可以对这些数据做批处理

但是hadoop是异步处理的 比较慢 在于短板 需要10个资源 那么最慢的一个资源将影响到Hadoop的运行速度

所以Hadoop在实时分析 性能不好

实时分析的场景：比如前段显示屏 我有多少个用户在线？多少笔交易完成了？哪个店铺在抢购时期？ 实现实时数据分析

开源界提供了实时分析的平台 ---- Spark Storm

以后 提供了machine learning 机器学习 算法 从人类的知识库 过不多久 天网就可以降临人间 比如神经网络

Hadoop节点 利用神经网络 可以完成数据进化

**这就是我们要有强大处理能力 有很好的机器学习算法**

所以**建立一个概念** 就是如果实时写操作的时候 而且操作量非常大 就需要**内存数据 或者NoSQL**

需要持久存储 或者后期分析 --- 结构化数据 ---- 关系型数据就非常好

对于非结构化数据 --- 日志 ---- 半结构 或者非结构 都放在文件系统 文件系统的分析 和Mysql不一样

MYSQL可以检索出来有效数据集

但是 这种文件系统的文件分析 就必须全量分析 ---- 这时候 要提供并行处理平台来完成性能的工作

随着网络功能的增强 主机的数量越来越大 有一天哪台主机坏了 要人工去维修

某一个服务建立在这台主机上面 如果宕机了怎办？

所以 一上来 啥也别干 就把这个机房的主机做成高可用集群 做成一个云平台 开各种虚拟机 在虚拟机上面完成各种任务 ---- 哪一个虚拟机坏了 因为是一个进程 直接kill掉 重新起来一个虚拟机进程 就没有问题了

或者坏了 支持实时迁移 迁移到其他物理平台上面 完全可以实现

一大堆虚拟机的共享存储 共享存储该怎么解决？为一大堆存储找一个解决方案 开一个虚机 创建一个虚拟磁盘 实现虚拟机进程的启动、

有一个前提 hadoop 不能够放在虚拟机上面 Hadoop需要大量的IO

但是虚拟机的IO是很差的

所以 对磁盘IO依赖很大的 就不能放在虚拟机上面

所以 前端这些 返回一个页面文件 读一个页面程序 都可以放在虚拟机上面 因为读取一个文件的IO很小

但是10T的IO 完全不一样的

所以 云 + Hadoop + Mysql结合起来使用

MySQL一大堆读 一大堆写 也不好 直接放到云里面 mysql直接提供云服务

以前讲过 云有三种类型 iaas paas saas

可以把MySQL做成 SaaS的服务 比如 数据我给你存储 至于怎么存储 你不用管

于是 可以完全实现后端MySQL数据平台模块化 hadoop平台模块化 云平台模块化 web服务模块化 每一个模块需要一个人 这样一个团队就可以了

你就是架构师

真正做架构师 网络+自动化运维编程（shell/python）+每一个服务本身的性能-特性-适用场景-优点-缺点 必须要做到通盘掌握 才能到这个层次

必须要在这些层次都能DBA 才能有

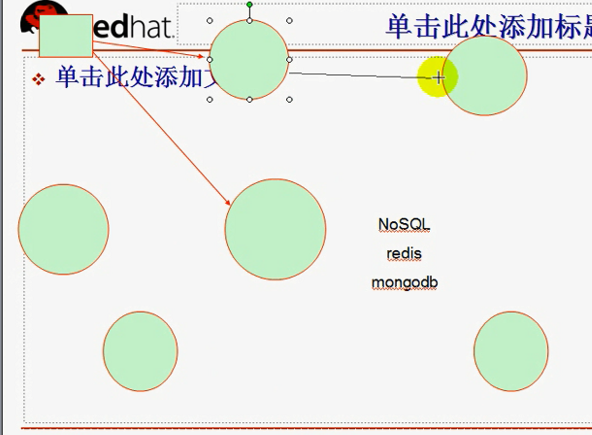
有机会接触大型系统

刚才把数据缓存在家门口 就是CDN

【我们360 root上传的文件 就是缓存在CDN上面的 --- 因为我们的用户来自于全国各地CDN的全称是Content Delivery Network，即**内容分发网络**

**CDN的基本原理是广泛采用各种缓存服务器，将这些缓存服务器分布到用户访问相对集中的地区或网络中，在用户访问网站时，利用全局负载技术将用户的访问指向距离最近的工作正常的缓存服务器上，由缓存服务器直接响应用户请求。**】

依赖于智能**DNS 把用户的请求解析到家门口去** ---- 这个依赖于后端的一个系统所生成的静态结果



如果一个用户到自己的缓存找不到数据 这个CDN还会支持这个缓存到另外一个缓存取数据

**所有的缓存都没有 最后到真正的系统取数据**

所以要能够实现内容路由 ----- 找到一个节点没找到 能够知道到哪些兄弟节点找 找不到 知道到原始服务器去找 ---- 这就是CDN网络

**CDN网络依赖于 只能DNS**

缓存服务器 缓存策略 和 缓存路由的功能

-------- 最后回到一台服务器上面 完成系统优化