一.摘要

1.网络游戏MMORPG整体服务器框架，包括早期,中期，当前的一些主流架构  
2.网络游戏网络层,包括网络协议,IO模型，网络框架，消息编码等。  
3.网络游戏的场景管理,AI，脚本的应用等。  
4.开源的网络服务器引擎  
5.参考书籍，博客

二.关键词

网络协议 网络IO 消息 广播 同步 CS TCP/UDP IP 集群 负载均衡 分布式   
网关服务器 GateServer 心跳 多线程/线程池 开源网络通讯框架/模型  
阻塞/非阻塞/同步/异步    Proactor/Reactor/Actor Select/Poll/Epoll/Iocp/Kqueue   
游戏开发中的设计模式/数据结构  
短连接和长连接 游戏安全 缓存 消息编码协议 脚本语言   
Socket Nagle/粘包/截断/TCP\_NODELAY AI/场景 分线/分地图 开源MMORPG服务器  
  
三.正文框架结构

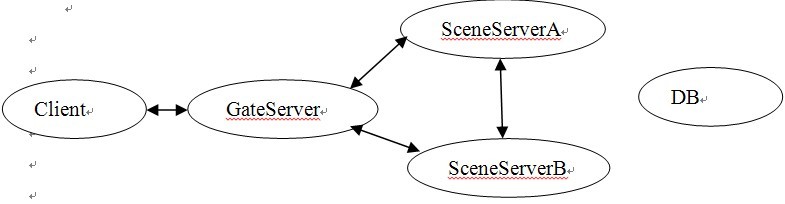
1.    早期的MMORPG服务器结构  
  
Client<->GameServer<->DB    所有业务,数据集中处理



优点:简单,快速开发  
缺点:  
    1.所有业务放在一起,系统负担大大增加.一个bug可能导致整个服务器崩溃,造成所有玩家掉线甚至丢失等严重后果。  
    2.开服一刹那,所有玩家全部堆积在同一个新手村.->>>>卡，客户端卡（同屏人数过多渲染/广播风暴） 服务器卡(处理大量同场景消息/广播风暴)

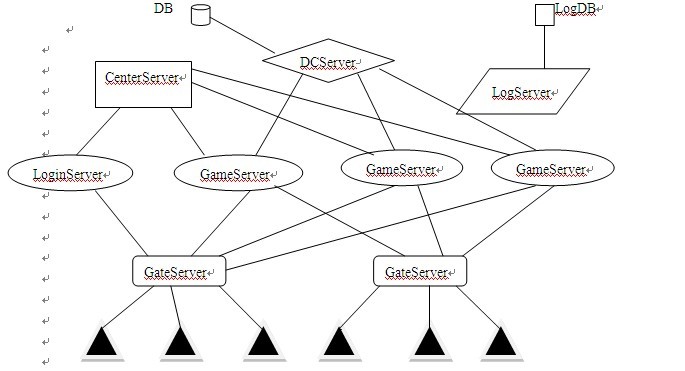
2.    中期-用户分离集群式  
  
                GameServe1  
Client            |                    DB  
                GameServer2  
  
玩家不断增多->分线->程序自动或玩家手动选择进入  
缺点:运营到后期,随着每条线玩家的减少, 互动大大减少。

3.    中后期 数据分离集群式



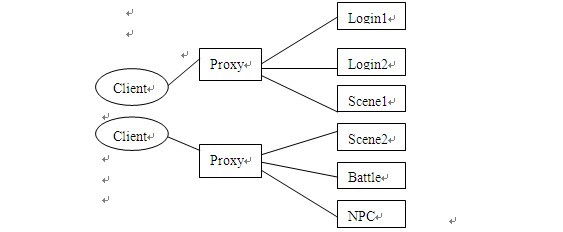
按地图划分服务器,当前主流  
    新手村问题：《天龙八部》提出了较好的解决方案，建立多个平行的新手村地图，一主多副，开服时尽可能多的同时容纳新用户的涌入，高等级玩家从其它地图回新手村只能到达主新手村。

4.    当前主流的网络游戏架构

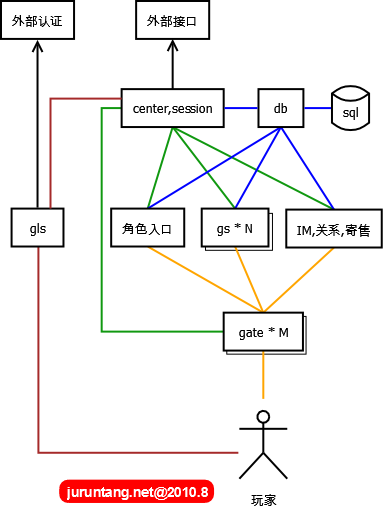
  
  
        注：在GateServer和CenterServer之间是有一条TCP连接的。而GameServer和LogServer之间的连接可以是UDP连接。这是有一个大概的图，很多地方需要细化。

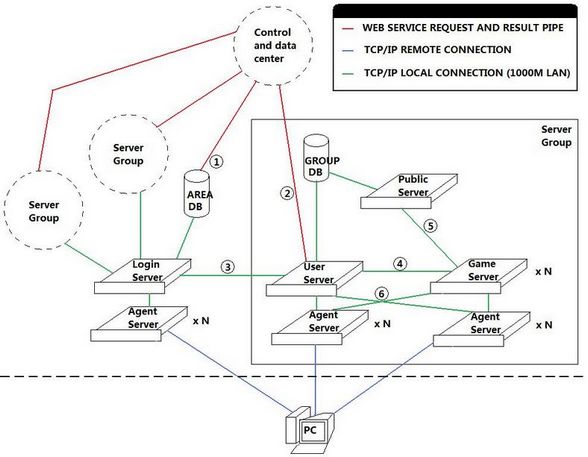
GateServer:网关服务器,AgentServer、ProxyServer  
  
 优点:   
    (1)作为网络通信的中转站，负责维护将内网和外网隔离开，使外部无法直接访问内部服务器，保障内网服务器的安全，一定程度上较少外挂的攻击。  
    (2)网关服务器负责解析数据包、加解密、超时处理和一定逻辑处理，这样可以提前过滤掉错误包和非法数据包。  
    (3)客户端程序只需建立与网关服务器的连接即可进入游戏，无需与其它游戏服务器同时建立多条连接，节省了客户端和服务器程序的网络资源开销。  
    (4)在玩家跳服务器时，不需要断开与网关服务器的连接，玩家数据在不同游戏服务器间的切换是内网切换，切换工作瞬问完成，玩家几乎察觉不到，这保证了游戏的流畅性和良好的用户体验。  
  
   缺点:   
1.网关服务器成为高负载情况下的通讯瓶颈问题  
2由于网关的单节点故障导致整组服务器无法对外提供服务的问题  
  
   解决：多网关技术。顾名思义，“多网关” 就是同时存在多个网关服务器，比如一组服务器可以配置三台GameGme。当负载较大时，可以通过增加网关服务器来增加网关的总体通讯流量，当一台网关服务器宕机时，它只会影响连接到本服务器的客户端，其它客户端不会受到任何影响。  
  
DCServer:数据中心服务器。主要的功能是缓存玩家角色数据，保证角色数据能快速的读取和保存  
CenterServer:全局服务器/中心服务器,也叫WorldServer. 主要负责维持GameServer之间数据的转发和数据广播。另外一些游戏系统也可能会放到Center上处理，比如好友系统,公会系统。  
  
    改进:将网关服务器细化为LogingateServer和多个GameGateServer.

5.    按业务分离式集群  
由于网络游戏存在很多的业务，如聊天，战斗，行走，NPC等，可以将某些业务分到单独的服务器上。这样每个服务器的程序则会精简很多。而且一些大流量业务的分离,可以有效的提高游戏服务器人数上限。

  
优点：  
      1.业务的分离使得每种服务器的程序变的简单，这样可以降低出错的几率。即使出错，也不至于影响到每一个整个游戏的进行,而且通过快速启动另一台备用服务器替换出错的服务器。  
     2.业务的分离使得流量得到了分散，进而相应速度回得到提升 。  
     3.大部分业务都分离了成了单独的服务器,所以可以动态的添加，从而提高人数上限。  
  
改进：甚至可以将登陆服务器细化拆分建角色,选择角色服务器

6.    一种简单实用的网络游戏服务器架构  
  
下图中每个方框表示一个独立的进程APP组件，每个服务进程如果发生宕机会影响部分用户，整体服务但不会全部中断。在宕机进程重启后，又可以并入整体，全部服务得以继续。



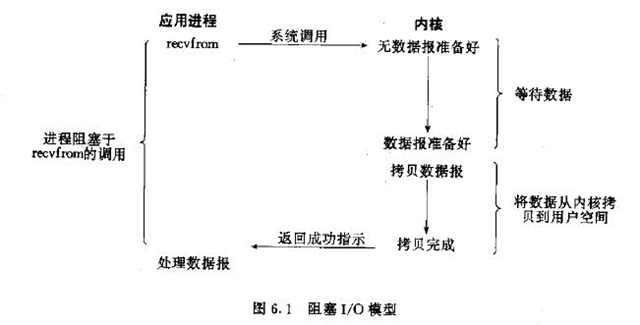
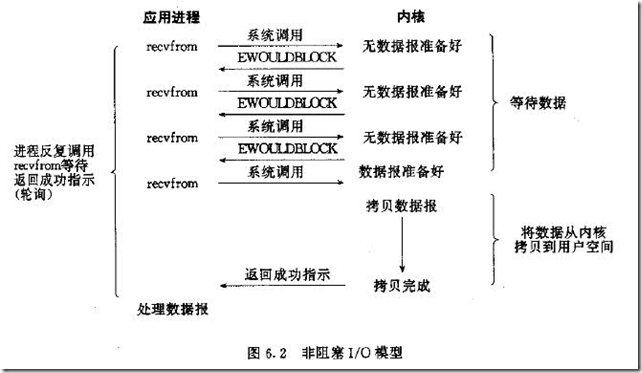
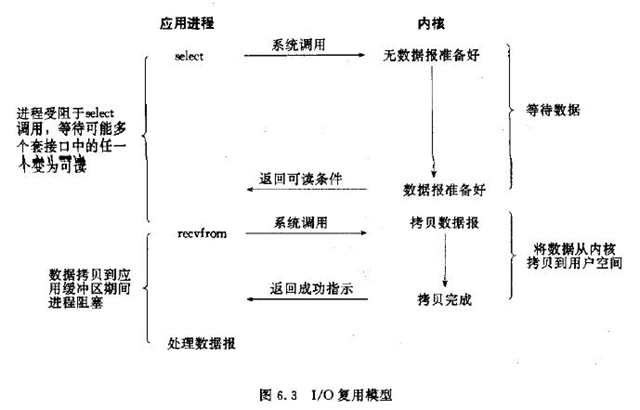
gls：game login server，游戏登录服务器，某种程序上，其不是核心组件，gls调用外部的接口，进行基本的用户名密码认证。此外需要实现很多附属的功能：登录排队（对开服非常有帮助），GM超级登录通道（GM可以不排队进入游戏），封测期间激活用户控制，限制用户登录，控制客户端版本等。  
db：实质上是后台sql的大内存缓冲，隔离了数据库操作，比较内存中的数据，只把改变的数据定时批量写入sql。系统的算法，开发稳定性都要求非常高。  
center：所有组件都要在这里注册，在线玩家的session状态都在这里集中存放，和各组件有心跳连接。所有对外的接口也全部通过这里。  
角色入口：玩家登录游戏后的选择角色  
gs：game server，最核心组件，同一地图，所有游戏逻辑相关的功能，都在这里完成。  
gate：建立和用户的常链接，主要作sockt转发，屏蔽恶意包，对gs进行保护。协议加密解密功能，一个gate共享多个gs，降低跳转地图连接不上的风险。  
IM，关系，寄售：表示其它组件，负责对应的跨地图发生全局的游戏逻辑。  
  
7.另一个架构图  
  


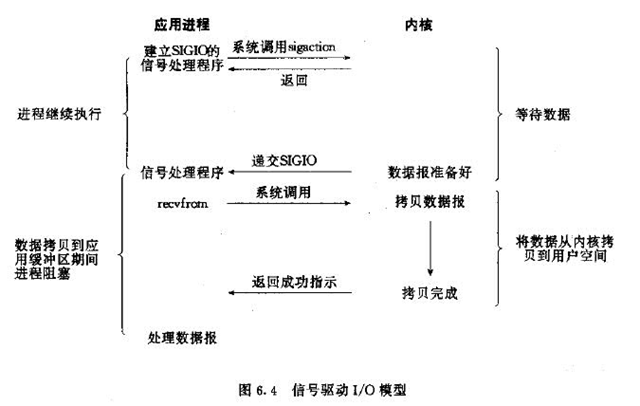
    1-   这是一条WebService的管道，在用户激活该区帐号，或者修改帐号密码的时候，通过这条通道来插入和更新用户的帐号信息。  
    2-   这也是一条WebService管道，用来获取和控制用户该该组内的角色信息，以及进行付费商城代币之类的更新操作。  
    3-   这是一条本地的TCP/IP连接，这条连接主要用来进行服务器组在登陆服务器的注册，以及登陆服务器验证帐户后，向用户服务器注册帐户登陆信息，以及进行对已经登陆的帐户角色信息进行操作（比如踢掉当前登陆的角色），还有服务器组的信息更新（当前在线玩家数量等）。  
    4-   这也是一条本地TCP/IP连接，这条连接用来对连接到GameServer的客户端进行验证，以及获取角色数据信息，还有传回GameServer上角色的数据信息改变。  
    5-   这条连接也是一条本地的TCP/IP连接，它用来进行公共信息服务器和数个游戏服务器间的交互，用来交换一些游戏世界级的信息（比如公会信息，跨服组队信息，跨服聊天频道等）。  
    6-   这里的两条连接，想表达的意思是，UserServer和GameServer的Agent是可以互换使用的，也就是玩家进入组内之后，就不需要再切换Agent。如果不怕乱套，也可以把登陆服务器的Agent也算上，这样用户整个过程里就不需要再更换Agent，减少重复连接的次数，也提高了稳定性。（毕竟连接次数少了，也降低了连不上服务器的出现几率）  
在这个架构里面，GameServer实际上是一个游戏逻辑的综合体，里面可以再去扩展成几个不同的逻辑服务器，通过PublicServer进行公共数据交换。  
    UserServer实际上扮演了一个ServerGroup的领头羊的角色，它负责向LoginServer注册和更新服务器组的信息（名字，当前人数），并且对Agent进行调度，对选择了该组的玩家提供一个用户量最少的Agent。同时，它也兼了一个角色管理服务器的功能，发送给客户端当前的角色列表，角色的创建，删除，选择等管理操作，都是在这里进行的。而且，它还是一个用户信息的验证服务器，GameServer需要通过它来进行客户端的合法性验证，以及获取玩家选择的角色数据信息。  
采用这种架构的游戏，通常有以下表现。  
    1- 用户必须激活一个大区，才能在大区内登陆自己的帐号。  
    2- 用户启动客户端的时候，弹出一个登陆器，选择大区。  
    3- 用户启动真正的客户端的时候，一开始就是输入帐号密码。  
    4- 帐号验证完成之后，进行区内的服务器选择。  
    5- 服务器选择完成之后，进入角色管理。同时，角色在不同的服务器里不能共享。

四.正文网络通讯

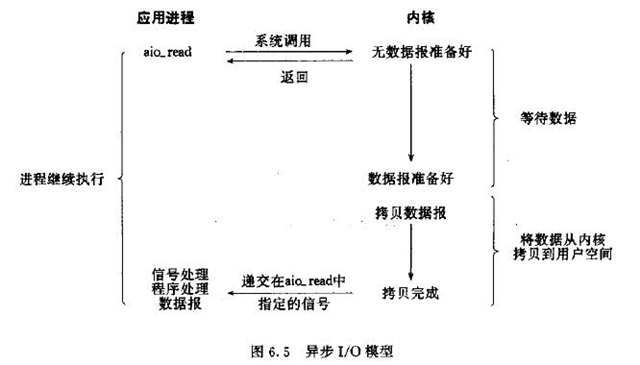
1.网络协议  
 根据游戏类型    实时性要求/是否允许丢包 来决定 TCP/UDP协议  
  
a.TCP:面向连接,可靠,保证顺序,慢,有延迟  
     TCP每次发送一个数据包后都要等待接收方发送一个应答信息，这样TCP才可以确认数据包通过因特网完整地送到了接收方。如果在一段时间内TCP没有收到接收方的应答，他就会停止发送新的数据包，转而去重新发送没有收到应答2的数据包，并且持续这种发送状态知道收到接收方的应答。所以这会造成网络数据传输的延迟，若网络情况不好，发送方会等待相当长一段时间  
       UDP:无连接,不可靠,不保证顺序,快  
  
b.长连接/短连接  
长连接，指在一个TCP连接上可以连续发送多个数据包，在TCP连接保持期间，如果没有数据包发送，需要双方发检测包以维持此连接，一般需要自己做在线维  
    连接→数据传输→保持连接(心跳)→数据传输→保持连接(心跳)→……→关闭连接  
短连接是指通信双方有数据交互时，就建立一个TCP连接，数据发送完成后，则断开此TCP连接,如Http  
    连接→数据传输→关闭连接

2.IO模型  
  
       Unix5中io模型  
1.    阻塞IO (Blocking I/O Model)  
2.    非阻塞IO (Nonblocking I/O Model)  
3.    IO复用 (I/O Multiplexing Model)  
4.    信号驱动IO (Signal-Driven I/O Model)  
5.    异步IO (Asynchronous I/O Model)

IO分两个阶段：  
1.通知内核准备数据。2.数据从内核缓冲区拷贝到应用缓冲区  
  
根据这2点IO类型可以分成：  
    1.阻塞IO，在两个阶段上面都是阻塞的。  
    2.非阻塞IO，在第1阶段，程序不断的轮询直到数据准备好，第2阶段还是阻塞的  
    3.IO复用，在第1阶段，当一个或者多个IO准备就绪时，通知程序，第2阶段还是阻塞的，在第1阶段还是轮询实现的，只是所有的IO都集中在一个地方，这个地方进行轮询  
    4.信号IO，当数据准备完毕的时候，信号通知程序数据准备完毕，第2阶段阻塞  
    5.异步IO，1，2都不阻塞  
  
  
  
  
  
  


同时阻塞多个I/O操作。而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用I/O操作函数  
Java#Selector  
  


允许套接口进行信号驱动I/O,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个SIGIO信号，可以在信号处理函数中调用I/O操作函数处理数据.



Java#NIO2  
发出系统调用后,直接返回。通知IO操作完成。  
前四种同步IO，最后一种异步IO.二者区别:第二个阶段必须要求进程主动调用recvfrom.而异步io则将io操作全部交给内核完成,完成后发信号通知。此期间,用户不需要去检查IO操作的状态，也不需要主动的去拷贝数据。

3.线程阻塞的原因:  
  
    1.Thread.sleep(),线程放弃CPU，睡眠N秒,然后恢复运行  
    2.线程要执行一段同步代码,由于无法获得相关的锁,阻塞。获得同步锁后，才可以恢复运行。  
    .线程执行了一个对象的wait方法，进入阻塞状态,只有等到其他线程执行了该对象的notify、nnotifyAll，才能将其唤醒。  
    4.IO操作,等待相关资源  
阻塞线程的共同特点是：放弃CPU,停止运行，只有等到导致阻塞的原因消除，才能恢复运行 。或者被其他线程中断，该线程会退出阻塞状态，并抛出InterruptedException.  
  
4.

阻塞/非阻塞/同步/异步  
同步/异步关注的是消息如何通知的机制。而阻塞和非阻塞关注的是处理消息。是两组完全不同的概念。

5.几个常用概念  
Select Poll  
Epoll(Linux) Kqueue(FreeBSD)

IOCP    windows

Reactor  
Dispatcher（分发器），Notifer（通知器）, 事件到来时，使用Dispatcher（分发器）对Handler进行分派，这个Dispatcher要对所有注册的Handler进行维护。同时，有一个Demultiplexer（分拣器）对多路的同步事件进行分拣。      
  
Proactor  
Proactor和Reactor都是并发编程中的设计模式.用于派发/分离IO操作事件的。这里所谓的IO事件也就是诸如read/write的IO操作。"派发/分离"就是将单独的IO事件通知到上层模块。两个模式不同的地方在于，Proactor用于异步IO，而Reactor用于同步IO。  
  
两个模式的相同点，都是对某个IO事件的事件通知(即告诉某个模块，这个IO操作可以进行或已经完成)。在结构上，两者也有相同点：demultiplexor负责提交IO操作(异步)、查询设备是否可操作(同步)，然后当条件满足时，就回调handler。  
不同点在于，异步情况下(Proactor)，当回调handler时，表示IO操作已经完成；同步情况下(Reactor)，回调handler时，表示IO设备可以进行某个操作(can read or can write)，handler这个时候开始提交操作。  
  
6.

网络通讯框架  
TCP Server框架:  
Apache MINA(Multipurpose Infrastructure for Network Applications)2.0.4  
Netty 3.5.0Final  
Grizzly 2.2  
Quickserver是一个免费的开源Java库，用于快速创建健壮的多线程、多客户端TCP服务器应用程序。使用QuickServer，用户可以只集中处理应用程序的逻辑/协议  
Cindy 强壮，可扩展，高效的异步I/O框架  
xSocket一个轻量级的基于nio的服务器框架用于开发高性能、可扩展、多线程的服务器。该框架封装了线程处理、异步读/写等方面  
ACE 6.1.0 C++ADAPTIVE CommunicationEnvironment,  
SmaxFoxServer 2.X 专门为Adobe Flash设计的跨平台socket服务器  
  
7.消息编码协议

AMF/JSON/XML/自定义/ProtocolBuffer  
  
无论是做何种网络应用，必须要解决的问题之一就是应用层从字节流中拆分出消息的问题，也就是对于 TCP 这种字节流协议，接收方应用层能够从字节流中识别发送方传输的消息.  
1.使用特殊字符或者字符串作为消息的边界，应用层解析收到的字节流时，遇见此字符或者字符串则认为收到一个完整的消息   
2.为每个消息定义一个长度，应用层收到指定长度的字节流则认为收到了一个完整的消息  
消息分隔标识（separator）、消息头（header）、消息体（body）  
 len | message\_id | data   
  
 |separator |     header   | body |  
 | len       | message\_id | data  
  
8. 粘包:  
TCP粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。   
    1.发送方引起的粘包是由TCP协议本身造成的，TCP为提高传输效率，发送方往往要收集到足够多的数据后才发送一包数据。若连续发送几次的数据都很少，通常TCP会根据优化算法把这些数据合成一包后一次发送出去，这样接收方就收到了粘包数据。  
    2.接收方引起的粘包是由于接收方用户进程不及时接收数据，从而导致粘包现象。这是因为接收方先把收到的数据放在系统接收缓冲区，用户进程从该缓冲区取数据，若下一包数据到达时前一包数据尚未被用户进程取走，则下一包数据放到系统接收缓冲区时就接到前一包数据之后，而用户进程根据预先设定的缓冲区大小从系统接收缓冲区取数据，这样就一次取到了多包数据  
  
解决措施:  
    1.对于发送方引起的粘包现象，用户可通过编程设置来避免，TCP提供了强制数据立即传送的操作指令push，TCP软件接收到该操作指令后，就立即将本段数据发送出去，而不必等待发送缓冲区满；  
TCP-NO-DELAY-关闭了优化算法,不推荐  
    2.对于接收方引起的粘包，则可通过优化程序设计、精简接收进程工作量、提高接收进程优先级等措施，使其及时接收数据，从而尽量避免出现粘包现象-当发送频率高时依然可能出现粘包  
    3.接收方控制，将一包数据按结构字段，人为控制分多次接收，然后合并，通过这种手段来避免粘包。-效率低  
    4.接收方创建一预处理线程，对接收到的数据包进行预处理，将粘连的包分开  
  
分包算法思路:  
基本思路是首先将待处理的接收数据（长度设为m）强行转换成预定的结构数据形式，并从中取出数据结构长度字段，即n，而后根据n计算得到第一包数据长度  
1) 若n<m，则表明数据流包含多包数据，从其头部截取n个字节存入临时缓冲区，剩余部分数据一次继续循环处理，直至结束。  
2) 若n=m，则表明数据流内容恰好是一完整结构数据，直接将其存入临时缓冲区即可。  
3) 若n>m，则表明数据流内容尚不够构成一个完整结构数据，需留待与下一包数据合并后再行处理。  
  
五.正文之场景管理、ai、脚本

 AOI: (Area Of Interest),广义上，AOI系统支持任何游戏世界中的物体个体对一定半径范围内发生的事件进行处理；但MMOPRG上绝大多数需求只是对半径范围内发生的物体离开/进入事件进行处理。当你进入一个游戏场景时，如果你能看到其他玩家，那背后AOI系统就正在运作.  
  
    1. 很容易想象，AOI的需求最简单的做法是全世界玩家信息全部同步给客户端。这个方案是O(n^2)的复杂度，对服务器来说是不能承受之重。但如果是超小地图十人以下的特殊需求倒可能是个简洁的方案。  
    2. 比较流行的方案是网格法，简单，高效：将地图按设定的格子大小划分为网格，设玩家移动到某坐标，我们很容易地将玩家归入该坐标所属的网格G的玩家链中，而这个玩家的可见集可以简单地将以网格G为中心的九宫格中的玩家链聚合而得到。而要获得两次移动间的可见集差异，也非难事.  
  
转自云风Blog:  
所谓 AOI ( Area Of Interest ) ，大致有两个用途。  
    一则是解决 NPC 的 AI 事件触发问题。游戏场景中有众多的 NPC ，比 PC 大致要多一个数量级。NPC 的 AI 触发条件往往是和其它 NPC 或 PC 距离接近。如果没有 AOI 模块，每个 NPC 都需要遍历场景中其它对象，判断与之距离。这个检索量是非常巨大的（复杂度 O(N\*N) ）。一般我们会设计一个 AOI 模块，统一处理，并优化比较次数，当两个对象距离接近时，以消息的形式通知它们。  
    二则用于减少向 PC 发送的同步消息数量。把离 PC 较远的物体状态变化的消息过滤掉。PC 身上可以带一个附近对象列表，由 AOI 消息来增减这个列表的内容。  
在服务器上，我们一般推荐把 AOI 模块做成一个独立服务 。场景模块通知它改变对象的位置信息。AOI 服务则发送 AOI 消息给场景  
AOI 的传统实现方法大致有三种：  
  
第一，也是最苯的方案。直接定期比较所有对象间的关系，发现能够触发 AOI 事件就发送消息。这种方案实现起来相当简洁，几乎不可能有 bug ，可以用来验证服务协议的正确性。在场景中对象不对的情况下其实也是不错的一个方案。如果我们独立出来的话，利用一个单独的核，其实可以定期处理相当大的对象数量。  
  
第二，空间切割监视的方法。把场景划分为等大的格子，在每个格子里树立灯塔。在对象进入或退出格子时，维护每个灯塔上的对象列表。对于每个灯塔还是 O(N \* N) 的复杂度，但由于把对象数据量大量降了下来，所以性能要好的多，实现也很容易。缺点是，存储空间不仅和对象数量有关，还和场景大小有关。更浪费内存。且当场景规模大过对象数量规模时，性能还会下降。因为要遍历整个场景。对大地图不太合适。这里还有一些优化技巧，比如可以把格子划分为六边形 的。  
  
第三，使用十字链表 (3d 空间则再增加一个链表维度) 保存一系列线段，当线段移动时触发 AOI 事件。算法不展开解释，这个用的很多应该搜的到。优点是可以混用于不同半径的 AOI 区域。  
  
2.AI

    1.怪物AI  
    2.NPC AI  
    3.世界环境AI  
实现方法：状态机  
 其他:  
 寻路：A\*  
 神经网络  
 遗传算法  
  
3.脚本语言的选择：

Lua/Python/Erlang  
Groovy/JRuby/Scala/Fantom/JPython-五大基于JVM的语言  
  作用:可应用于部分应用层逻辑经常发生变化的系统。如任务系统。以在不需要重新编译整个工程的情况下调整、 测试和修改游戏运行的机制和特性

六.正文之开源网络游戏服务器

魔兽世界模拟器  
mangosTrinity  TrinityCore2  
  
天堂2模拟器  
L2J  
  
永恒之塔模拟器  
  
Arianne  
  
七.正文之参考书籍，博客

1.云风Blog  ttp://codingnow.com/  
2.书籍<大型多人在线游戏开发><网络游戏服务器编程><UNIX网络编程>