tcp相关的

tcp提供的是面向连接的、可靠地字节流传输。

在一个tcp连接中，仅有双方进行彼此通信，广播和多播不能使用tcp。

tcp使用校验、确认和重传机制保证可靠传输。

tcp给数据分节进行排序，并使用累计确认机制保证数据的顺序不变和非重复。

tcp通过使用滑动窗口机制来实现流量控制，通过动态改变窗口的大小进行阻塞控制。

三次握手

服务器和客户端共发送三次包。三次握手的目的，连接服务器指定端口，建立tcp连接，并同步数据双方的序列号和确认号，交换TCP窗口大小信息。在socket编程中，客户端执行connect()触发三次握手。

第一次握手（SYN=1 seq=x）

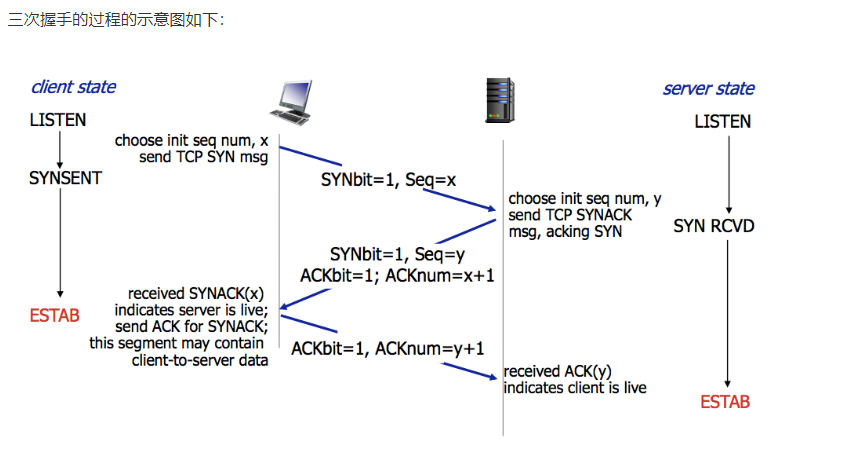
客户端发送一个TCP的SYN标志位为1的包，指明客户端打算连接的服务器端口，以及初始化序号X，保存在包头的序号（Sequence Number）字段中。

第二次握手（SYN=1 ACK=1 seq=y ACKnumber=x+1）

服务器发回确认包ACK响应，即SYN标志位和ACK标志位都为1。服务器端选择自己ISN序列号，放到Seq域里，同时将确认序号（ACKnowledgement Number）设置为客户端的ISN加1，即X+1。发送完毕后，服务器端进入SYN\_RCVD状态。

第三次握手（ACK=1 ACKnum = y+1）

客户端在此发送确认包（ACK）,SYN标志位为0，ACK标志位为1，并且把服务器发来的ACK的序号字段+1，放在确定字段中发送给对方，而且在数据字段里写ISN的+1。发送完毕后，客户端进入ESTABLISHED状态，当服务器接收到这个包时，也进入ESTABLISHED状态，TCP握手结束。



TCP的连接的拆除需要发送四个包，因此称为四次挥手。客户端或者服务器均可以主动发送挥手动作，在socket编程中，任何一方执行close()操作即可产生挥手操作。

第一次挥手（FIN=1，seq=x）

假设客户端想要关闭连接，客户端发送一个FIN标志位置为1的包，表示自己已经没有数据可以发送了，但是仍可以接收数据。

发送完毕后，客户端进入FIN\_WAIT\_1状态。

第二次握手(ACK=1, ACKnum = x + 1)

服务器端确认客户端FIN包，发送一个确认包，表明自己接收到客户端关闭连接的请求，但还没有准备好关闭连接。

发送完毕后，服务器段进入CLOSE\_WAIT状态，客户端接收到这个确认包之后，进入FIN\_WAIT\_2状态，等待服务器端关闭连接。

第三次挥手（FIN = 1， seq = y）

服务器端准备好关闭连接时，向客户端发送结束连接请求，FIN设置为1。

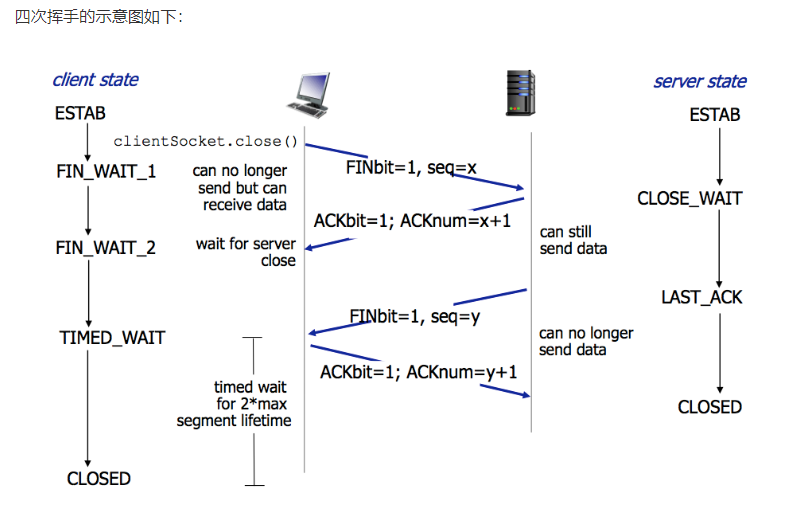
发送完毕后，服务器端进入LAST\_ACK状态，等待来自客户端的最后一个ACK。

第四次挥手（ACK = 1， ACKnum = y + 1）

客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入TIME\_WIAT状态，等待可能出现的要求重传的ACK包。

服务器端接收到这个确认包之后，关闭连接，进入close状态。

客户端等待了某个固定时间（两个最大段生命周期）之后，没有收到服务器端的ACK，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入CLOSED状态。



SYN攻击 半连接攻击

在三次握手过程中，服务器发送SYN\_ACK之后，收到客户端的ACK之前的TCP连接称为半连接。此时服务器处于SYN\_RCVD状态。当收到ACK后，服务器才能转入到ESTABLISHED状态。

SYN攻击指的是，攻击客户端在短时间内伪造大量不存在的IP地址，向服务器不断发送SYN包，服务器回复确认包，并等待客户端的确认。由于源地址不存在的，服务器需要不断的重发直至超时，这些伪造的SYN包将长时间占用未连接队列，正常的SYN请求被丢弃，导致目标系统运行缓慢，严重者会引起网络堵塞甚至系统瘫痪。

SYN攻击是一种典型的DoS/DDoS攻击。

如何检查SYN攻击？

检查SYN攻击非常的方便，当你在服务器上看到大量半连接状态时，特别是源IP地址是随机的，基本上可以判定是一次SYN攻击。在Linux/Unix上可以使用系统自带的netstats命令来检测SYN攻击。

如何防御SYN攻击？

SYN攻击不能完全被阻止，除非将TCP协议重新设计。我们所做的是尽可能的减轻SYN攻击的危害，常见的防御SYN攻击的方法有如下几种：

1.缩短超时时间。

2.增加最大半连接数

3.过滤网关防护

4.SYN cookies技术

IO复用

首先来看看服务器编程的模型，客户端发来的请求服务器会产生一个进程来对器进行服务，每当来一个客户端请求就会产生一个进程来服务，然而进程不可能无限的产生，因此为了解决大量客户端访问的问题，引入了IO复用技术。

即：一个进程可以同时对多个客户端请求进行服务。

select原理概述

调用select时，会发生以下事情：

1. 从用户空间拷贝fd\_set到内核空间；
2. 注册回调函数\_poolwait
3. 遍历所有fd，对全部指定设备做一次poll（这里的poll是一个文件操作，他有两个参数，一个是文件fd本身，一个是当设备尚未就绪是调用的回调函数\_pollwait，这个函数把设备自己的特有等待队列传给内核，让内核把当前的过程挂载到其中）；
4. 当设备就绪时，设备就会唤醒在自己特有等待队列中的所有节点，于是当前进程就获取到了完成的信号。poll文件文件操作返回的是一组标准的掩码，其中的各个位指示当前的不同的就绪状态（全0为没有任何事件触发），根据mask可对fd\_set赋值；
5. 如果所有设备返回的掩码都是没有显示任何的事件触发，就去掉回调函数的函数指针，进入有限是的睡眠状态，再恢复和不断做poll，再作有限时的睡眠，直到其中一个设备有事件触发为止。
6. 只要有事件触发，系统调用返回，将fd\_set从内核空间拷贝到用户空间，回到用户态，用户就可以相应的fd作进一步的读或者写操作了。

epoll原理概述

调用epoll\_create时，做了以下事情：

1. 内核帮我们在epoll文件系统里建了个file节点；
2. 在内核cache里建了个红黑数用于存储以后epoll\_ctl传来的socket；
3. 建立一个list链表，用于存储准备就绪的事件。

调用epoll\_ctl时，做了以下事情：

1. 把socket放到epoll文件系统里file对象对应到红黑树上；
2. 给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把他放到准备就绪的list链表里。

调用epoll\_wait时，做了以下事情：

观察list链表里有没有数据。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没有数据也返回。而且，通常情况下即使我们要监控百万计的句柄，打多一次也只返回很少量的准备就绪句柄而已，所以，epoll\_wait仅需要从内核态copy少量的句柄到用户态而已。

总结如下：

一颗红黑树，一张准备就绪句柄链表，少量的内核cache，解决了大并发下的socket处理问题。

执行epoll\_create时，创建了红黑树和就绪链表；

执行epoll\_ctl时，如果增加socket句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向内核注册回调函数，用于当中断事件来临时向准备就绪链表中插入数据；

执行epoll\_wait时立刻返回准备就绪链表里的数据即可。

两种模式的区别：

LT模式下，只要一个句柄上的事件一次没有处理完，会在以后调用epoll\_wait时重复返回这个句柄，而ET模式仅在第一次返回。

两种模式的实现：

当一个socket句柄上有事件时，内核会把该句柄插入上面所说的准备就绪list链表，这时我们调用epoll\_wait，会把准备就绪的socket拷贝到用户态内存，然后清空准备就绪list链表，最后，epoll\_wait检查这些socket，如果是LT模式，并且这些socket上确实有未处理的事件时，有把该句柄放回到刚刚清空的准备就绪链表。所以，LT模式的句柄，只要它上面还有事件，epoll\_wait每次都有返回。

对比

Select缺点：

1. 最大并发数量限制：使用32个整数的32位，即32\*32 = 1024来标识fd，虽然可以修改，但是有以下第二点的瓶颈；
2. 效率低：每次都会线性扫描整个fd\_set，集合越大速度越慢；
3. 内核/用户空间内拷贝问题。

Epoll的提升：

1. 本身没有最大并发连接的限制，仅受系统中进程能打开的最大文件数目限制；
2. 效率提升：只有活跃的socket才会主动的去调用callback函数；
3. 省去不必要的内存拷贝：epoll通过内核与用户空间mmap同一块内存实现。

当然，以上的优缺点仅仅是特定场景下的情况：高并发，且任一时间只有少数socket是活跃的。

如果在并发量低，socket都比较活跃的情况下，select就不见得比epoll慢了（就像我们常常说快排比插入排序快，但是在特定情况下这并不成立）。

Mysql通信协议

服务器启动后，会使用TCP监听一个本地端口，当客户端的连接请求到达时，就会执行三次握手以及MySql的权限验证；验证成功后，客户端开始发送请求，服务器会以响应的报文格式返回数据；档客户端发送完成后，会发送一个特殊的报文，告知服务器已经结束会话。

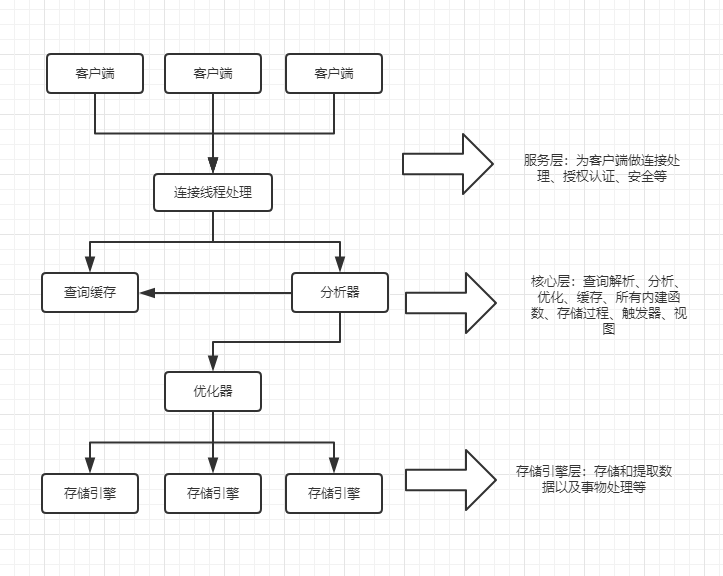
MySql定义了几种包类型，A）客户端->服务器，登录时的auth包、执行sql是的CMD包；B）服务器->客户端，登录时的握手包、数据包、数据流结束包、成功包（OK Packet）、错误信息包。

协议定义了基本的数据类型，如int、string等；数据的传送格式等。

MYSQL框架

1. MYSQL的逻辑结构

服务层、核心层、存储引擎层



synchronized实现细节

关键字synchronized可以保证在同一个时刻，只有一个线程可以执行某个方法或者某个代码块（主要是对方法或者代码块中存在共享数据的操作），同时我们还应该注意到synchronized另外一个重要的作用，synchronized可保证一个线程的变化（主要是共享数据的变化）被其他线程所看到的（保证可见性，完全可以替代Volatile功能），这点确实也是很重要的。

Synchronized的三种应用方法

Synchronized关键字最主要有以下3种应用方式，下面分别介绍

1. 修饰实例方法，作用于当前实例加锁，进入同步代码前需要获得当前实例的锁。
2. 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁。
3. 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码块前要获得给定对象的锁。

Synchronized作用于实例方法

所谓的实例对象锁就是用synchronized修饰的实例对象中的实例方法，注意是实例方法不包括静态方法，如下

public class AccountingSync implements Runnable{

//共享资源(临界资源)

static int i=0;

/\*\*

\* synchronized 修饰实例方法

\*/

public synchronized void increase(){

i++;

}

@Override

public void run() {

for(int j=0;j<1000000;j++){

increase();

}

}

public static void main(String[] args) throws InterruptedException {

AccountingSync instance=new AccountingSync();

Thread t1=new Thread(instance);

Thread t2=new Thread(instance);

t1.start();

t2.start();

t1.join();

t2.join();

System.out.println(i);

}

/\*\*

\* 输出结果:

\* 2000000

\*/

}

上述代码中，我们开启了两个线程操作同一个共享资源即变量i，由于i++操作并不具备原子性，该操作是先读取值，然后写回一个新值，相当于原来的值加上1，分两步完成，如果第二个线程在第一个线程读取旧值和写回新值期间读取i的域值，那么第二个线程就会与第一个线程一起看到同一个值，并执行相同值的加1操作，这也就造成了线程安全失败，因此对于increase方法必须使用synchronized修饰，以便保证线程安全。此时我们应该注意到synchronized修饰的是实例方法increase，在这样的情况下，当前线程的锁便是实例对象instance，注意java中的线程同步锁可以是任意对象。从代码执行结果来看确实是正确的，倘若我们没有使用synchronized关键字，其最终输出结果就很可能小于20000000，这边是synchronized关键字的作用。这里我们意识到，当一个线程正在访问一个对象synchronized实例方法，那么其他线程不能访问该对象的其他synchronized方法，毕竟一个对象只有一把锁，当一个线程获取了该对象的锁之后，其他线程无法获取该对象的锁，所以无法访问该对象的其他synchronized实例方法，但是其他线程还是可以访问该实例对象的其他非synchronized方法，当然如果是一个线程A需要访问实例对象obj1的synchronized方法f1（当前对象锁是obj1），另一个线程B需要访问实例对象obj2的synchronized方法f2（当前对象锁是obj2），这样是允许的，因为两个实例对象锁并不相同，此时如果两个线程操作数据并非共享的，线程安全是有保障的，遗憾的是如果两个线程操作的是共享数据，那么线程安全就有可能无法保证了。如下代码将演示出该现象

public class AccountingSyncBad implements Runnable{

static int i=0;

public synchronized void increase(){

i++;

}

@Override

public void run() {

for(int j=0;j<1000000;j++){

increase();

}

}

public static void main(String[] args) throws InterruptedException {

//new新实例

Thread t1=new Thread(new AccountingSyncBad());

//new新实例

Thread t2=new Thread(new AccountingSyncBad());

t1.start();

t2.start();

//join含义:当前线程A等待thread线程终止之后才能从thread.join()返回

t1.join();

t2.join();

System.out.println(i);

}

}

上述代码与前面不同的是我们同时创建了两个新实例AccountingSyncBad，然后启动两个不同的线程对共享变量i进行操作，但是遗憾操作结果是1452317而不是期望结果2000000，因为上述代码犯了严重的错误，虽然我们使用synchronized修饰了increase方法，但却new了两个不同的实例对象，这也意味着存在着两个不同的实例对象锁，因此t1和t2都会进入各自的对象锁，也就是说t1和t2线程使用的是不同的锁，因此线程安全是无法保证的。解决这种困境的方法是将synchronized作用于静态的increase方法，这样的话，对象锁就当前类对象，由于无论创建多少个实例对象，但对于的类对象拥有只有一个，所有在这样的情况下对象锁就是唯一的。

Synchronized作用于静态方法

当synchronized作用于静态方法时，其锁就是当前类的class对象锁。由于静态成员不专属于任何一个实例对象，是类成员，因此通过class对象锁可以控制静态成员的并发操作。需要注意的是如果一个线程A调用一个实例对象的非static synchronized方法，而线程B需要调用这个实例对象所属类的静态synchronized方法，是允许的，不会发生互斥现象，因为访问静态synchronized方法占用的锁是当前类的class对象，而访问非静态synchronized方法占用的锁是当前实例对象锁。

public class AccountingSyncClass implements Runnable{

static int i=0;

/\*\*

\* 作用于静态方法,锁是当前class对象,也就是

\* AccountingSyncClass类对应的class对象

\*/

public static synchronized void increase(){

i++;

}

/\*\*

\* 非静态,访问时锁不一样不会发生互斥

\*/

public synchronized void increase4Obj(){

i++;

}

@Override

public void run() {

for(int j=0;j<1000000;j++){

increase();

}

}

public static void main(String[] args) throws InterruptedException {

//new新实例

Thread t1=new Thread(new AccountingSyncClass());

//new心事了

Thread t2=new Thread(new AccountingSyncClass());

//启动线程

t1.start();t2.start();

t1.join();t2.join();

System.out.println(i);

}

}

由于synchronized关键字修饰的是静态increase方法，与修饰实例方法不同的是，其锁对象是当前类的class对象。注意代码中的increase1方法是实例对象，其对象锁是当前实例对象，如果别的线程调用该方法，将不会产生互斥现象，毕竟锁对象不同，但是我们应该意识到这样的情况下可能会发生线程安全问题（操作了共享静态变量i）。

Synchronized同步代码块

除了使用关键字修饰实例方法和静态方法外，还可以使用同步代码块，在某些情况下，我们编写的方法体可能比较大，同时存在一些比较耗时的操作，而需要同步的代码又只有一小部分，如果直接对整个方法进行同步操作，可能会得不偿失，此时我们可以使用同步代码块的方法对需要同步的代码进行包裹，这样就无需对整个方法进行同步操作了，同步代码块的使用示例如下：

public class AccountingSync implements Runnable{

static AccountingSync instance=new AccountingSync();

static int i=0;

@Override

public void run() {

//省略其他耗时操作....

//使用同步代码块对变量i进行同步操作,锁对象为instance

synchronized(instance){

for(int j=0;j<1000000;j++){

i++;

}

}

}

public static void main(String[] args) throws InterruptedException {

Thread t1=new Thread(instance);

Thread t2=new Thread(instance);

t1.start();t2.start();

t1.join();t2.join();

System.out.println(i);

}

}

从代码看出，将synchronized作用于一个给定的实例对象instance，即当前实例对象就是锁对象，每次档线程进入synchronized包裹的代码块时就会要求当前线程持有instance实例对象锁，如果当前有其他线程正持有该对象锁，那么新到的线程就必须等待，这样就会保证每次只有一个线程执行i++操作。当然除了instance作为对象外，我们还可以使用this对象或者当前累的class对象作为锁，如下代码：

//this,当前实例对象锁

synchronized(this){

for(int j=0;j<1000000;j++){

i++;

}

}

//class对象锁

synchronized(AccountingSync.class){

for(int j=0;j<1000000;j++){

i++;

}

}

Synchronized底层语义原理

Java虚拟机中的同步（sychronization）基于进入和退出管程（Monitor）对象实现，无论是显示同步（有明确的monitorenter和monitorexit指令，即同步代码块）还有隐式同步都是如此。在java语言中，同步用的最多的地方可能是被synchronized修饰的同步方法。同步方法并不是由monitorenter和monitorexit指令来实现同步的，而是由方法调用指令读取运行时常量池中方法的ACC\_SYNCHRONIZED标志来隐式实现的，关于这点，稍后详细分析。下面先来了解一个概念Java对象头，这对深入理解synchronized实现原理非常关键。

理解Java对象头与Monitor

在JVM中，对象在内存中的布局分为三块区域：对象头、实例数据和对齐填充。如下：

实例变量：存放类的属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组的长度，这部分内存按4字节对齐。

填充数据：由于虚拟机要求对象起始地址必须是8字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐，这点了解即可。

而对于顶部，则是Java头对象，它实现sunchronized的锁对象的基础，这点我们重点分析它，一般而言，synchronized使用的锁对象是存储在Java对象头里的，jvm中采用2个字来存储对象头（如果对象是数组则会分配3个字，多出来的1个字记录的是数组长度），其主要结构是由Mark Word和Class Metadata Address组成，其结构说明如下表：



在32位系统下，存放Class指针的空间大小是4字节，MarkWord是4字节，对象头为8字节。

在64位系统下，存放Class指针的空间大小是8字节，MarkWord是8字节，对象头为16字节。

64位开启指针压缩的情况下，存放Class指针的空间大小是4字节，MarkWord是8字节，对象头为12字节。

数组长度4字节+数组对象头8字节(对象引用4字节（未开启指针压缩的64位为8字节）+数组markword为4字节（64位未开启指针压缩的为8字节）)+对齐4=16字节。

静态属性不算在对象大小内。

AOI模块

一则是解决了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他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他对象，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：

1. 也是最笨的方案，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。
2. 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。在对象进入或者退出格子时，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，存储空间不仅仅和对象数据有关，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等。
3. 使用十字链表（3d空间则更增加一个链表的维度）保存一些列线段，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。有点事可以混合与不同半径的AOI区域。