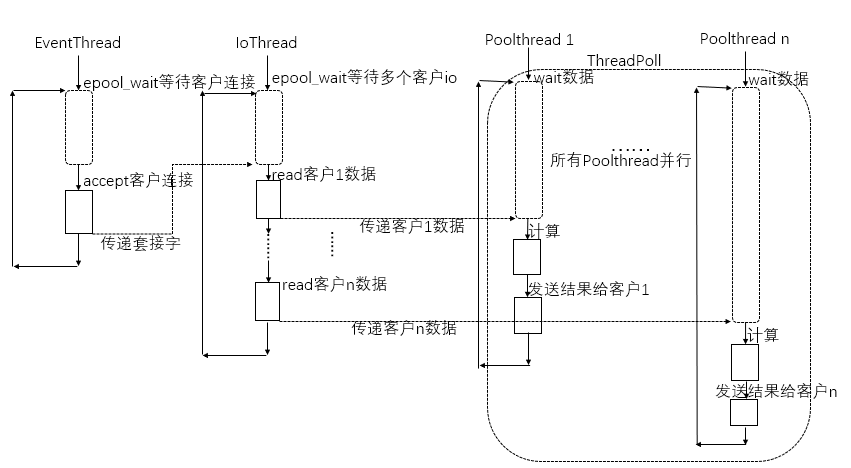
1. 整体设计

这次设计的是一个专门用来解数独的服务器程序，主要实现了 reactor +线程池的设计方法。下面是整体的设计图：



整个程序总共使用了10个线程：1个accept线程+1个io线程+8个线程池内的计算线程。

1.专门用一个accept线程和io分开，希望客户一连接就可以被处理，而io可以等待。防止io数据太慢，导致客户光是连接服务器就要等待。

2.只用一个io线程来接收所有线程的数据，因为本程序的数据量不是那么大。如果传输数据太大导致一个io线程接收的太慢，可以开多个io线程等待accept线程通知，防止用户数据等待太长时间。

3.用一个线程池来专门计算，把io和计算任务分开。本次的解数独程序，专门选了一个需要20ms的算法来解，需要一定的计算时间。相比于io，计算的时间更长。因此需要专门的计算线程，防止io阻塞后面的用户数据等待时间过长。若计算量较小，可以选择直接在io线程收完数据直接计算，因为唤醒线程池还有一个排队和唤醒的开销。

1. EventThread类

1. EventThread类主要存放了几乎所有的数据，来使其他线程共享这些数据。另外它本身就是accept线程，接受客户的连接，并通知io线程有新用户已连接。同时，通过它启动了IO线程，并初始化线程池，初始化回调等等，是整个程序的发起者。

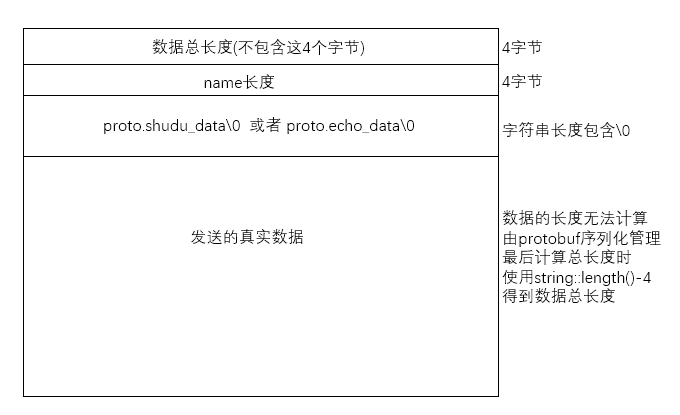
2.accept的业务逻辑在Run中实现，创建一个epoll，然后把服务器的监听套接字加入epoll中，然后进入无限循环等待监听套接字可读。（**注意，监听套接字需要设置成非阻塞的。虽然使用了epoll通知有连接才去accept，但是不能保证不会阻塞，因为如果在epoll通知后和调用accept之前，客户又断开了连接，这时再调用accept就会阻塞。对于本程序单独的accept线程似乎没有什么影响。。。**）

每收到一个连接，就往epoll中注册一个用户的数据io事件等待，然后给每个连接fd分配一个FdIoBuffer，使用一个unordered\_map关联起来，每个用户单独一个iobuf，因为我假设，每个用户可以发送多条消息过来。多个用户共用一个iobuf会导致数据混乱，管理起来很困难。

每收到一个已注册用户的io通知，则把用户的fd放入一个阻塞队列，通知io线程处理这个连接。放入队列和取队列的操作需要加锁，并且是异步通知，需要条件变量。（**注意，虽然accept和io线程是不同的线程，但是它们的事件是注册在同一个epoll中的，试想一下，如果使用不同的epoll，那么io线程阻塞队列的取操作通知就会被epoll\_wait给阻塞了，从而不能及时得到有新的用户fd已连接需要注册epoll这件事**）(一边在wait另一边又要往里面加入新东西)

3.io线程：首先是读取accept线程传入阻塞队列的io就绪客户的fd，这里**新建了一个空队列，然后在临界区内直接swap(阻塞队列，空队列)，把所有fd取出，然后退出临界区处理io任务。**这样减小了临界区的使用时间，减少临界区两边线程相互阻塞时间。

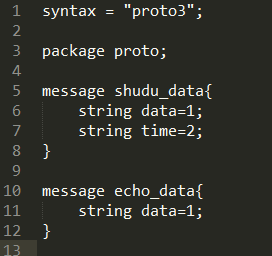
处理每个用户的io任务，即读取用户数据。考虑到用户问题或者网络问题，用户发送一条或者多条任务消息，有可能只收到半条，或者一条半数据等等不完整的情况，首先需要检测每个用户缓冲区的数据头大小（4字节），当**缓冲区收到的数据>=数据总长度+数据头大小**时才算收到一条完整的消息，然后反序列化并发送给线程池处理。每条消息我也设计了一个消息格式见下图。



收到一条完整的用户消息之后，根据消息的name注册相应的任务回调函数给threadpool。（这里我直接把回调函数写死在代码里了，后续可以改进，可以把它派生出来，变成虚函数由子类重写），然后调用ThreadPoll类的push函数把数据送入阻塞队列通知线程池有计算任务来到。若遇到未知的消息类型，则中断该客户的连接。

1. Protobuf消息格式设计

1.在我的程序中，总共设计了两种protobuf消息供客户发送：第一种是数独数据，包含一个9\*9=81长度的string，表示一个9\*9的数独矩阵。再带上一个表示时间的string。还有一种是echo消息，就是一个简单string消息，发来的什么，服务器就原样返回过去。（**protobuf的使用场景是一个结构包含许多成员，如若干个string，若干个int等等，在进行网络传输时需要序列化成一个char类型的字符流进行二进制传输（因为不同的int在不同的机器表示方式不同），转换起来非常麻烦，protobuf就提供了一种高效的序列化和反序列化的方法把它们送到你的传输buffer中。**）



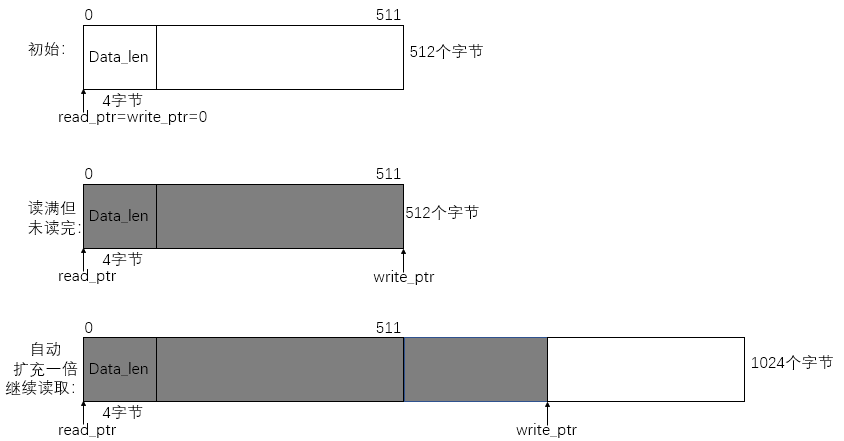
因为有两种不同的任务消息类型，所以需要有一个任务派遣功能（后续的改进），因为数据不同，消息长度也不同，因此在接收数据的时候，要自己定义消息格式（上面已经讲了），约定数据长度怎么通知对方。

在服务器收到数据的时候，需要反序列化数据，而反序列化又需要声明一个对应类型的protobuf message容器对象来从buffer中反序列化二进制数据，因此我们的消息格式还需要包含一个name字段，通过protobuf的反射功能获取对应的message对象再进行数据反序列化。

1. FdIoBuffer类

1.这个就是服务器为每个客户连接分配一个的缓存类了，用于读取客户传来的数据。一开始本来想做成接收发送通用的buffer，后来发现写回数据还用它的话，有点不方便，直接用一个临时的string来保存将要发送的数据更加方便一点，发送完就释放，下次发送重新在栈上分配一个。还有一个原因就是，反序列化的之后，数据的总长度需要用string::length()-head\_len来得到，所有数据都放在一起发送就不好求这个值了。

2.整个buffer的设计图如下



buffer主要的功能函数就是int ReadFd(int fd);，参数传入一个套接字描述符，然后读取其中的所有数据。因为是epoll通知套接字可读，所以此时去读一定会有数据而不会阻塞，**但是还是要设置fd非阻塞标志。因为read读取的数据可能超过当前剩余的缓存大小，此时read只能返回剩余缓存大小的值n==left\_size，但是我们只知道缓存已满，而不知道数据是否读完**，所以我们应该做的操作是增加缓存大小，递归的再去调用ReadFd函数继续读取剩下的数据。**此时有两种可能，一种是数据已经读完，你再去read就必须要把fd设置成非阻塞，否则就会阻塞在当前read上，其他用户的io任务无法处理，现在再去read就会立即返回一个EAGAIN错误告诉你没数据了你还来读取，我们返回上次读取的数据n即可；另一种就是数据还没读完，这时再继续读剩下的就行，如果还会满，再递归一次即可，最后返回的是所有读取的数据之和。（这里感觉需要加一个数据量控制，不然恶意用户狂发数据就会把内存占满）。**我设置的是buffer大小每满一次就扩容成两倍，初始为512字节。

3.发送的和上面的格式差不多，一开始声明一个string str(8)，前4个字节先空出来，等最后写完了再来填。从&str[4]这个位置开始按照消息格式往后装数据。



1. SolveServer类

这个就是用来存放服务器处理数据的函数类了，注册的时候只要把类里面的函数地址传递过去就可以了。

1. Dispacher类

IoThread类会负责接收客户发来的数据并反序列化解析。之后会把数据push到线程池的任务队列中唤醒compute线程来处理。而用户会发来不同的数据对应不同的任务处理函数，本程序就故意设计了两个任务a、解数独任务 b、echo任务。所以dispacher的任务就是根据不同的message类型调用不同的处理函数。处理函数可以一开始在SolveServer类中写好。然后在EventThread类的初始化函数中进行注册。**（注意，不同的消息类型并不是根据switch匹配来确定调用哪个函数，这样的设计太low了，如果加一个业务处理函数，还得在switch里添加一个case从而修改代码。我的设计方法是用了protobuf的反射功能得到具体message类型的基类descriptor指针，映射成对应的处理函数，处理函数一开始就注册好，使用模板+多态的方法：在初始状态调用注册函数自动生成不同的模板实例，然后用基类指针指向不同的派生类从而调用不同的处理函数）**

1. ThreadPoll类

1. ThreadPoll类的主要功能就是初始化若干个计算线程，本程序所有的线程都在一开始创建好，中间不会再创建线程销毁线程，所有线程的生命周期和整个程序的生命周期一样，所以也不用管资源回收之类的事情。（**初始化的计算线程数量由“阻抗匹配原则”得到。大概公式就是num=c/p，C是你的cpu核的数目，p是你的每个线程想要占据cpu的时间（也可以理解为你的计算任务的比重）。假设你的cpu有4核，线程池有一半任务在计算，则T=4/0.5=8，若p<0.2则这个公式就不再适用了，此时应该直接给一个定值最好**）

2.因为IO线程调用Push函数传过来的数据是protobuf message \*类型，你需要找到对应的回调函数来进行计算，需要一个unordered\_map映射message和回调函数。（**后面优化应该把回调函数注册在ThreadPoll里面，函数名相同，参数类型不同（不同的message），而不是对每一个反序列化得到的message注册一次，利用函数的重载特性，从而调用不同的处理函数，这样也可以少一次映射过程，传来数据直接拿来调用process(xxx\_message \*)即可**）。

处理完了计算任务，ThreadPoll还要负责把结果发回给客户，所以还需要一个unordered\_map映射message和fd。获取fd才能发送数据。

ThreadPoll内的若干个计算线程初始都是在wait睡眠，等到io线程调用push把数据传入阻塞队列才会被唤醒，因此还需要一个阻塞队列，和信号量，取数据还要一个互斥量。

