1. 环境
2. OS: Windows10 WSL, Ubuntu 18.04.4 LTS
3. 数据库：Ver 8.0.21 for Linux on x86\_64 （至少需要8.0，否则不提供C API Asynchronous Interface）
4. Lua 5.3.4
5. 内容

实现了一个简单的tcp服务器，完成对客户端的请求从数据库完成数据抓取和计算，然后将结果发回客户端。支持多个客户端同时连接和处理。意义有这几点：

1. 节省了用户自己编写SQL和计算代码的时间
2. 使无数据库访问权限的的用户也可以进行特定的数据查询
3. 将所有用户的请求整合至一处，从而可以充分利用计算资源（CPU、GPU等），同时服务器在合适的时候可以重用用户请求的计算结果，避免冗余计算
4. 程序

目前项目由两部分组成：服务器端（server.cpp）和测试用客户端（client.cpp）。除此之外，代码还包括register.lua（管理脚本并提供公用的lua function），scripts文件夹（包含对应不同请求的.lua脚本），mylib.cpp（c++ library，为lua脚本常用逻辑及多线程计算提供c++函数实现），bind.h（结合lunar.h绑定自定义c++ object到lua），read\_buffer.lua（处理buffer收到的request的逻辑）。

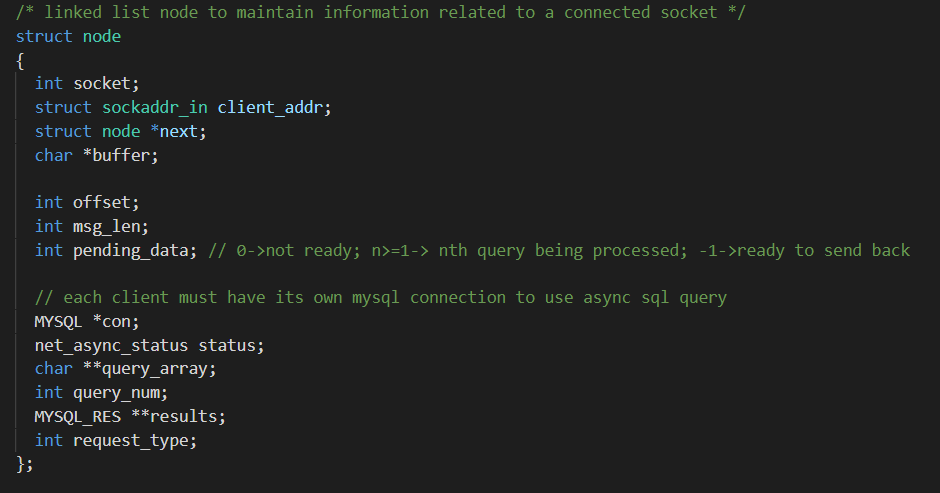
服务器端接受一个参数：运行端口；客户端接受两个参数：服务器域名和端口。

客户端链接到服务器后会连续多次（目前为10次）发送请求并接收结果，计算从发出到接收的平均时延。客户端发送的请求前两bytes代表请求的类别，后面的bytes为对应这个请求的特殊格式。

服务器利用lua管理对不同请求应对逻辑。Lua维护一个全局的table，key是query type，value是对其后续处理的lua脚本文件，这样对后续逻辑统一接口执行lua\_scripts[query type]脚本。lua脚本再在适合的情况下调用c++（如split等公用函数，或数据量大需要pthread、CUDA协助计算）。

1. 服务器结构

为了处理concurrency的问题，服务器使用linked list保存所有连接的客户端，每个node保存对应的信息，如下所示：



前四个不言而喻。因为send和recv不能保证一次完成，因此需要记录信息长度和offset用以继续未完成的send/recv。Pending\_data作为状态位，记录目前状态。因为要从数以亿计的record中查询记录，向mysql发送的query可能花费比较长的时间，为了提高并发效率，我使用了异步的mysql c api，结合non-blocking的send/recv，自然的将状态划分为三类：

0代表未收到请求或正在接受请求；

n>=1代表数据库正在执行第n个query；

-1代表已经完成计算，可以向客户端发送结果

按照mysql官方文档，一个connection不能同时发送多个异步query，因此每个node都要建立并保存自己的connection。Status则是用于判断异步query的执行情况。Query array保存了所有需要向mysql发送的query，而query num代表queryarray保存的query的总数。在每完成一个query后，结果会存在results里。request\_type用于完成全部query后调用对应lua脚本。

服务器在while(1)的循环中，首先清空fd set，将listening socket加入read set。然后遍历链表将所有connected socket加入read set，并在过程中检查状态位，若为-1则加入write set，若大于0则检查status，根据结果重新执行当前query或执行下一个query并将状态位+1，若无下一个query进行计算，完成后将状态位置于-1。

然后执行select监听read set和write set。若select有监听到，先查看是否有来自listening socket的输入，是则建立新链接并加入linked list。再遍历链表看是否

1. 有未完成的write，有则继续向客户端send结果
2. 有客户端的输入，有则recv。如收到全部信息则调用read\_buffer.lua解包，并根据输入构建需要执行全部query，保存到node中，然后执行第一个query
3. 请求
4. 数据库中有两张表，分别记录用户登录登出，格式为(UID BIGINT, dtEventTime DATETIME, ……)。若要查询某用户某月登录时间和次数的请求，请求格式为：

|-------------------------------------------------------------------------------|

|2 bytes |2 bytes | 8 bytes | 2 bytes |

|-----------------------------------------------------------------------------|

| request长度 |request type = 1 | UID | 月份 |

|------------------------------------------------------------------------------|

服务器返回的结果格式为：

|---------------------------------------------------------------|

|2 bytes | 2 bytes | 4 bytes |

|---------------------------------------------------------------|

| request type = 1 | 登录次数| 登录时长/s|

|---------------------------------------------------------------|

因为一个用户一个月的登录记录数量有限，最多在几百的数量级，所以计算过程是lua通过重复调用c++ library里面的time\_gap函数实现的，并未加入多线程或者GPU计算。

1. 数据库中有一个比赛报名信息表，包含四个队员的号码信息，格式为

(TeamID CHAR(32), OpenID1 CHAR(64), OpenID2 CHAR(64),OpenID3 CHAR(64),OpenID4 CHAR(64))。另有一比赛队伍战绩表，格式为

(dtEventTime DATETIME, OpenID1 CHAR(64), OpenID2 CHAR(64),OpenID3 CHAR(64),OpenID4 CHAR(64), TeamKill INT, TeamRank INT, …….)

要按照参赛队伍中最近30场中20场积分较高的比赛的平均积分，从报名的参赛队伍中筛选出20支队伍。每场比赛的积分由TeamRank 和TeamKill 共同决定。该请求的格式为：

|-------------------------------------------|

|2 bytes |2 bytes |

|--------------------------------------------|

| request长度 |request type = 2 |

|--------------------------------------------|

服务器返回的结果格式为：

|--------------------------------------------------------------------------------------------|

|2 bytes | 4 bytes | 4 bytes | ……. | 4 bytes |

|--------------------------------------------------------------------------------------------|

| request type = 2 | TeamID1 | TeamID2 | ……. | TeamID20 |

|--------------------------------------------------------------------------------------------|

对加入并行计算的探索：

本项目最开始的意义之一就在整合请求可以充分利用计算资源，同时我也想实践一下lua与C++多线程的结合。因此对此请求的部分计算用pthread实现并行计算。并行计算重视速度，而lua重视方便灵活，这两者似乎是有冲突的。但是如果将并行计算作为lua计算逻辑中的添加剂，对于计算过程中时间复杂度最高的瓶颈部分，如果其复杂度足够高，使用并行计算实现即可在不损失整体框架灵活性的前提下，以较少的额外代码大幅提高部分请求的处理速度。

本请求其实并不是很适合加入并行计算。因为如果设有M支队伍，每个队伍有N场比赛记录，输入的数据量为M\*N。计算过程包括使用hashmap分开队伍（复杂度O(MN)，且需要锁读写hashmap，不利于并行化），对每队所有比赛按日期排序（复杂度O(M\*NlogN)），对最近30场计算分数并排序（复杂度O(M)）。可以看到最高复杂度不过O(M\*NlogN)，并不高且基本等同于lua与C++传输数据的量（M\*N），可以预见并行计算的加入甚至会减慢总计算速度，因为lua本身不支持多线程，总是需要用O(M\*N)的时间把数据单线程的从lua传给C++（可以通过把通讯层和lua层接口从string改为mysql res结构体指针的userdata解决，但因时间有限且不影响本质，并未实际进行修改）。但是将来如果有大数据量高计算复杂度的请求，并行计算的加入还是有价值的。

如上所述，我在项目中将“对所有队伍的比赛按日期排序”这一步使用并行计算实现，本质是对一个value是table(array)的table，对value的元素（也是array）按元素的日期排序。相关代码在mylib.cpp中。实现本身除了开始使用lua C API将数据读入C++，和最后使用lua C API将数据写入Lua，其余部分就是正常的C++ pthread多线程编程。考虑到只需要最近的30场比赛，这里使用priority queue进行了优化，将每队的排序复杂度从O(NlogN)降低到了O(Nlog30)= O(N)。

6. 测试

打开服务器

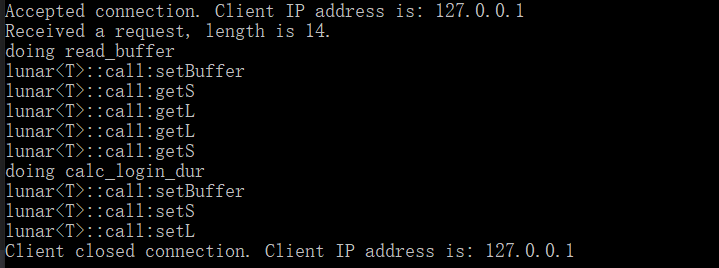


客户端连接服务器

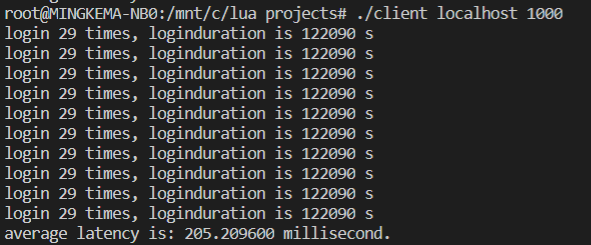


请求a：

服务器收到请求

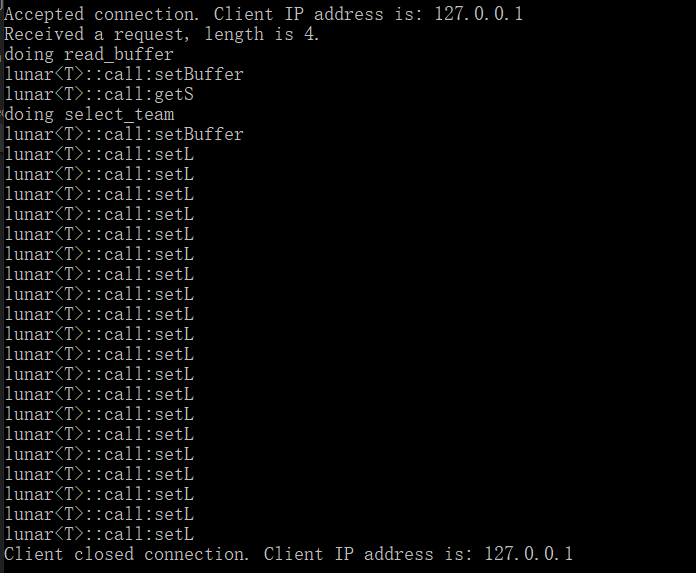


客户端收到结果



请求b：

服务器收到请求



客户端收到结果

