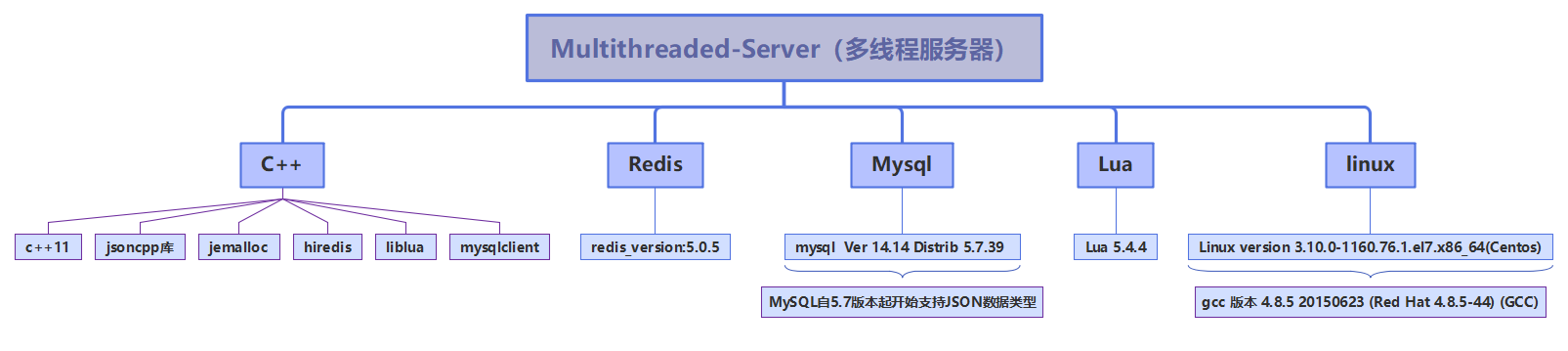
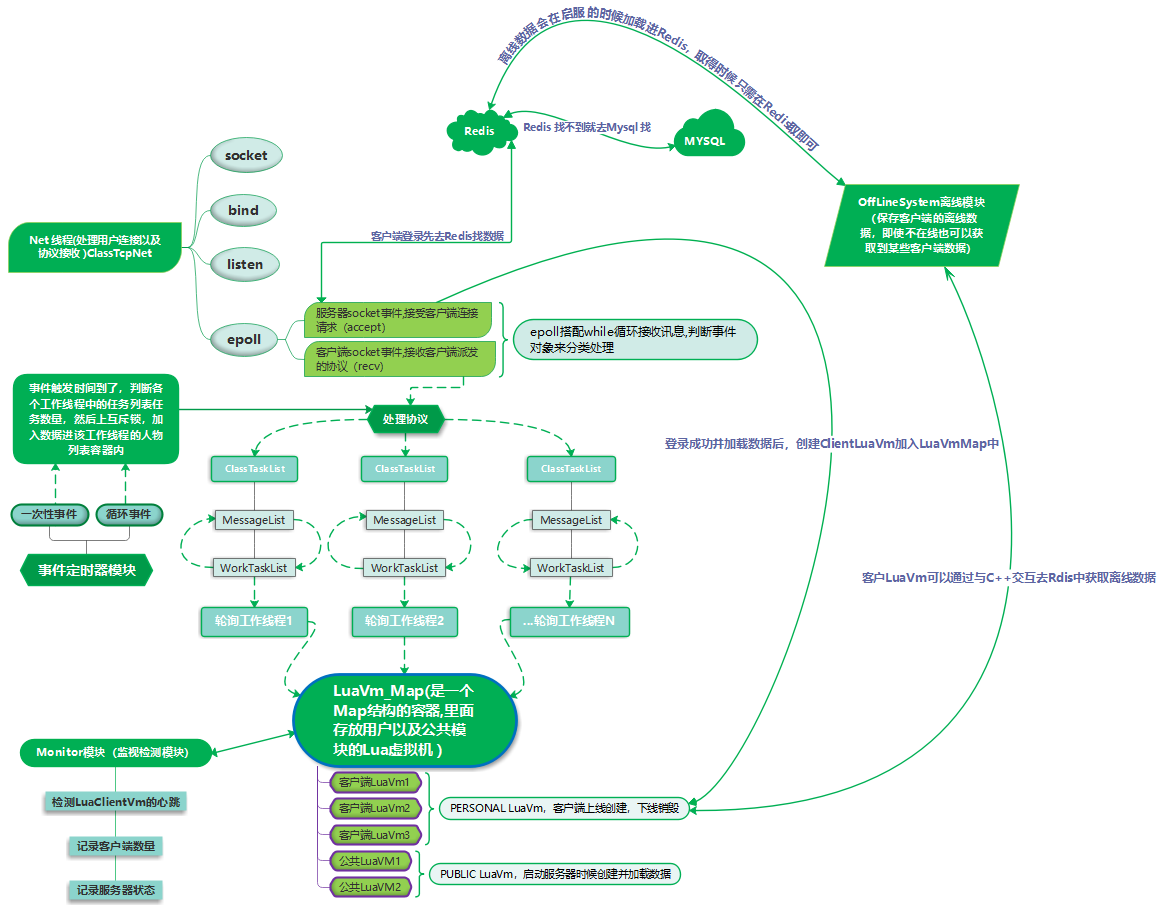
## **Multithreaded-Server**

## 技术栈以及环境：



## **组成结构：**



## **文件概述：**

1. **ClassDB :** 里面封装了Mysql 以及 Redis的类, 并提供了各自的数据库调用接口

项目使用Mysql来存储持久化数据, 但是由于Mysql存取数据实在硬盘或者磁盘上操作的, 因此, 存取的速度较慢, 而Redis用于存储使用较为频繁的数据到缓存中, 所以存取速度快, 所以用户上线连接成功后会先查看Redis中是否有数据留存, 没有就去Mysql里加载进入Reids中缓存起来。

由于Mysql数据的可持久化存储, 项目里,数据在**关服**或者用**户下线**又或者**通过定时推送**来存放在Mysql中, 而其余时间,获取数据可以在Redis 中查找以及存放,若查找不到则从Mysql中加载进Redis中。这样可以提高服务器存取用户数据的效率, 而且把数据保存在Redis也可以不用担心崩服后数据的丢失。

Mysql据库的存放用了JSON格式来存储

因为Lua的table数据结构和C++交互实现便捷性不高,所以可以用JSON作为统一格式

数据库中的每个模块,都以用户的ID作为索引,script字段作为数据来存储在Mysql中

虽然可观性不好，但是可以很好的处理C++和Lua数据交互.

对于可观性,可以通过创建视图等函数之类的DB操作来额外生成可观性高的视图来查看数据

所有在C++调用的语句都定义在名为DBCommand.h的头文件中（方便统一管理）.

加载过程：

用户在C++登录后,创建虚拟机,然后根据用户Uid 先在Redis检测用户数据是否存在, 假若不存在, 才根据Uid和模块名在指定的Mysql数据库加载用户数据存进Redis中。

保存过程：

数据保存有三种方式，可以在etc/Config中配置

①：用户下线保存,这种保存方式会直接写入Mysql中，速度慢, 但是不用担心Redis和服务器的崩溃。

②：关服保存，这样的话可以减少C++程序和Mysql的交互存取操作,仅仅是登录的时候判断不在Redis缓存中才取Mysql加载数据，而且下线的时候也没和Mysql交互，而是停服的时候才把数据存进Mysql中保存。好处是减少了程序和Mysql交互，存取数据在Redis中，速度快且方便，缺点是假如Redis崩溃会造成数据丢失。

③：定时推送这种方式可以避免在停服的时候才保存数据的问题，虽然也可能造成数据丢失，但是丢失程度相对关服保存会小一点。但是坏处是,交互相对频繁,每次都要保存很多数据

1. **ClassPthreadMgr :** 该文档里定义了工作线程管理类, 以及任务池类

该项目中,使用到多个工作线程来同时处理其各自任务池的任务列表里面的任务，对比单线程来说提高了处理业务的效率。

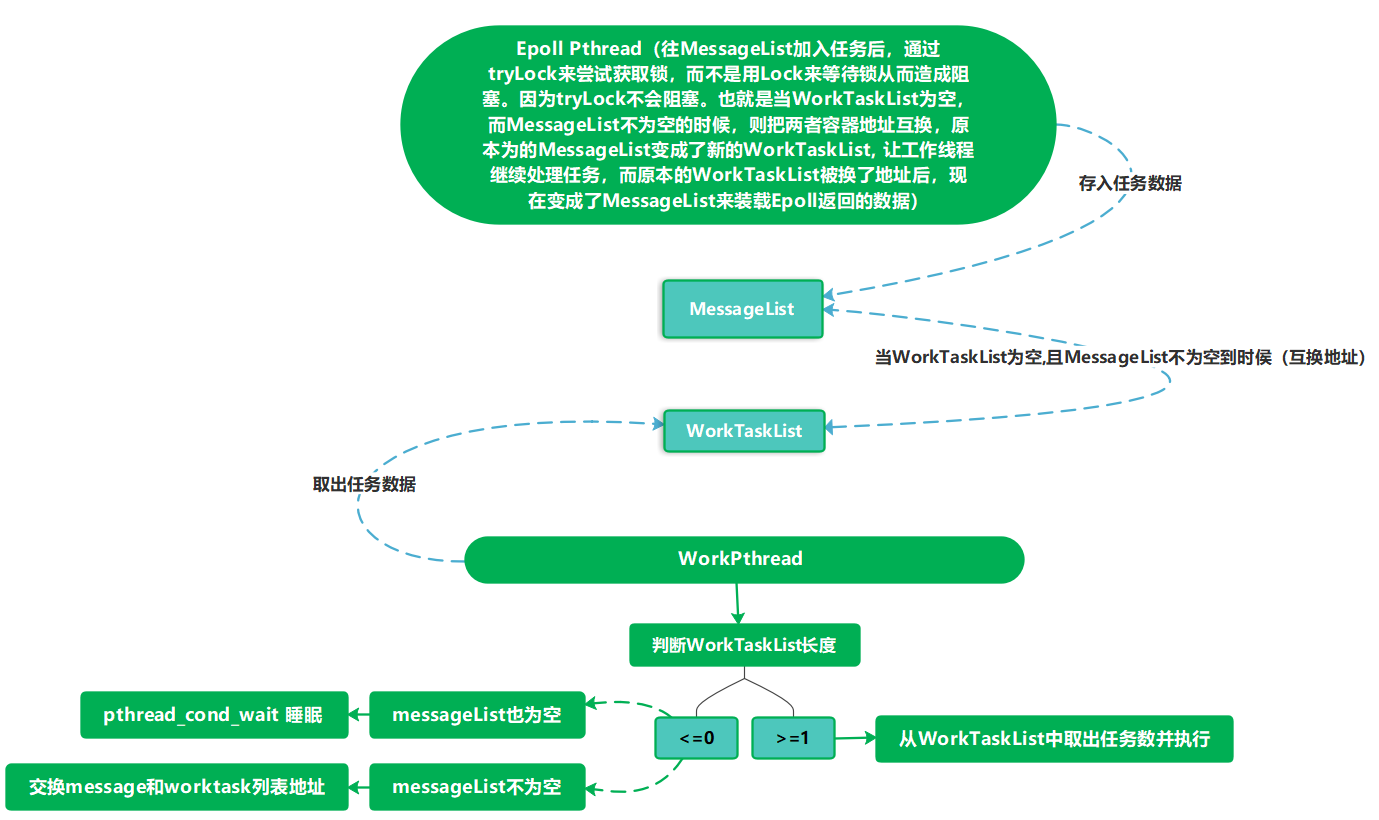
**工作线程和任务池关联思路：**由于是多线程的环境下, 假如用一个任务池存放任务, 则是多个工作线程来争夺一个任务池的资源,然后进行任务删减以及处理, 这样会存在数据的不一致性, 而且任务池列表用的结构是std::list ,该数据结构非线程安全, 也就是说需要用到锁来确保同一时间只有一个工作线程来获取任务池的任务列表的处理权。加了锁就意味着多个线程争抢一个锁以致于工作线程处理业务的效率降低, 这样多线程的优势会大大减少, 所以最后采用了**一个工作线程对应一个任务池**的方式, 这样一来，每个工作线程都有自己的任务池, 不需要所有工作线程进行加互斥锁来争抢同一个任务池。

**任务池：**任务池是由两个List结构组成的, 一个用来接收Epoll线程返回的的任务数据, 另一个则是工作线程从中获取数据。这样做的理由是避免任务数列表加锁而降低效率, 因为Epoll线程返回的任务需要加入到任务列表中,而工作线程则需要从任务列表中取出数据, 这样的话存在容器同时存取的问题, 而上面说过了list结构为非线程安全, 所以只有一个任务列表是需要加上互斥锁的, 保证同一时间只有工作线程或者是Epoll线程来操作任务列表。但是这样加锁解锁太过频繁了,会降低服务器性能, 所以,我们可以分两个任务列表来分别**存放任务数据**和**取出任务数据。**

**遇到的问题：**

假如工作线程所取的任务列表(WorkTaskList)为空了（任务处理完了）, 怎么从另一个接收了任务数据的列表中获取任务呢？

如下图：



（1）MessageList为空的时候, 两者为空，则直接睡眠（pthread\_cond\_wait会自动解锁再进行睡眠），降低CPU的使用率，避免线程一致占据资源, 直到有新的任务数据被假如MessageList中。

我们把Epoll线程存放数据进入MessageList作为条件变量。条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。

所以当Epoll放入任务的时候，就会触发pthread\_cond\_signal来唤醒正在睡眠的工作线程，假如多个工作线程都共同作用于一个变量条件,那么唤醒的时候则是多个工作线程之间相互争夺唤醒条件变量所关联的锁。

1. MessageList不为空的时候，则直接交换地址，让空的(workTaskList)容器变成MessageList来接收来自Epoll线程处理后的数据，而装有任务数据的任务列表（来自原本Epoll线程存放任务的容器MessageList），则变成了工作线程所遍历的任务容器(workTaskList)了。

总结：

1. 通过两个任务列表，使得解锁和上锁的频率降低，比起使用单个任务列表（加入和取走任务的时候都需要上锁）避免了每次都要操作锁才能进行任务的存取。
2. 当两个容器都为空的时候，工作线程会休眠，降低了CPU占用率，当有新的数据加入MessageList的时候才会唤醒沉睡的工作线程，避免了工作线程做无意义的循环。
3. **ClassTimer :** 定时器类

该类定义了两种事件类型 ①单次事件 ②循环事件

该线程会循环单次事件容器 以及 循环事件容器，达到触发事件条件边加入任务列表给予工作线程处理。

事件类包含：触发时间以及事件字符串

1. **Email :** 自定义的发邮件处理，可以发送QQ邮件（用于发送注册码，有需要可以自己接短信或者邮箱接口）
2. **Logger :** 自定义日志（最基础简单的日志模块，加锁输出，避免打印混乱，主要是用来打印测试，以及输出报错）
3. **LuaServer :** 该内容是Lua层的部分

①LuaLib内存放Lua的工具库

②LuaSystem存放Lua业务逻辑代码

③LuaVmMgr存放C++封装而成的LuaVm类和Lua虚拟机管理类等等，以及C++和Lua交互的接口函数

Lua模块部分是通过Lua虚拟机来与C++交互的

Lua虚拟机分为 “**个人类型 , LuaPersonalVm**” 以及 “**公共类型 , LuaPublicVm**”

区别如下:

①“个人类型”：个人类型的Lua虚拟机用来加载个人lua模块,这些模块每个用户登录成功后都会有自己的模块字段。不用模块的个人模块直接可以相互调用, 以用户个体为单位。

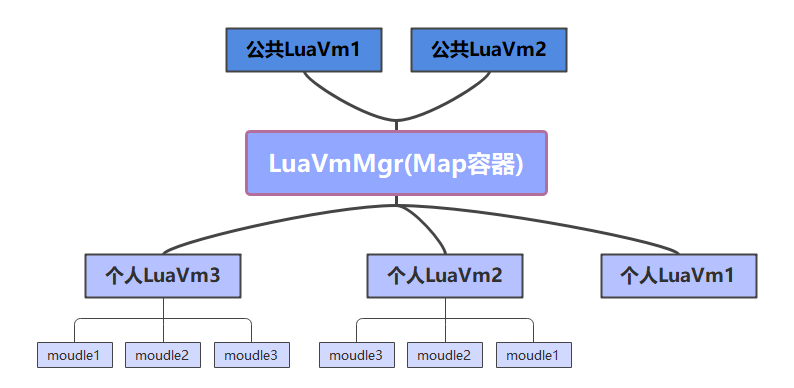
例如：每个用户都有自己Bag模块,个人虚拟机之间相互独立。

（该虚拟机会在玩家登录,加载数据成功后进行创建, C++ 以及Mysql和Redis中交互的格式都是JSON格式,在Lua中则是Table类型数据结构）

②“公共类型”：该类型的虚拟机是公用的,每个用户都可以对公共虚拟机进行操作,公共模块为每个用户创建了一个协程（corotinue）来进行操作, 假如在公共虚拟机要调用其他虚拟机,则协程会挂起, 知道下次该虚拟机被同一个用户回调后,被挂起的协程则继续完成调度。

若不同虚拟机之间想交互,则要通过返回特定的数据,来调用别的虚拟机.

（该类虚拟机会在服务器启动的时候进行创建,停服的时候保存）



模块定义在该文件同目录下的名为“MoudleFilesLoad.txt”的文本文件内，只需要按规定填写上新建的模块信息即可：

格式：{模块名（全大写）,模块的文件路径}

例：{ROOM,LuaServer/LuaSystem/Room/roomMgr.lua}

Lua虚拟机与C++之间传递的参数数据类型为：JSON格式,因为Mysql5.7.xxx后支持JSON格式数据的存储

为了统一性,协议,数据库,以及Lua信息的传递均用JSON格式处理.

Lua虚拟机之间若要传递信息,则返回固定的格式作为corotinue的挂起返回值

如：

coroutine.yield(xxx,xxx,xx,xx,xx,x)

返回参数分别是：

std::string arg = ""; //参数

std::string fun = ""; //调用函数/协议

std::string called = ""; //被调用的模块VM名字

std::string caller = ""; //调用者,一开始调用者模块默认为空

std::string uid = ""; //调用者id

1. **Monitor :** 定义了监视类

主要是用于检测服务的连接人数，以及服务器的流畅程度，还有连接者的心跳（避免掉线无通知）

1. **Net :** 该部包含了

ClassMsgPackage类:用于把要处理的协议以及事件封装成任务数据包, 待该数据包到了工作线程处再进行拆解并处理任务

Client是客户端类：里面保存了客户端的数据，每当有客户端连接成功，就根据协议参数生成一个Client类对象。其中包含了两个特别的成员变量，分别是：**用户当前任务数量**以及**当前所处任务线程索引**。

①**用户当前任务数量**:该变量代表该用户当前未处理的的任务数量，当为0的时候Epoll线程会判断所有的工作线程，哪个工作线程所对应的任务列表中的任务少，就往该个工作线程所对应的MessageList中存放任务。

②**当前所处任务线程索引**:假如用户当前任务数量大于0，则要根据当前任务所处线程索引来存放任务了。因为同一个客户端假如原本就有任务A尚未完成的话，该任务在某个工作线程任务列表中，这时候客户端收到了新的协议，打包成了任务B，这时候就要把B任务包塞进A任务所在的工作线程所对应的任务列表中。否则，就会存在B任务比A任务先完成的情况。

例如：客户端a，收到协议A打包后塞入工作线程1所对应的任务列表中，这时候又接收到了协议B，打包后塞入了工作线程2，这时候，A在1工作线程的任务列表中，B在2工作线程人物列表中，这个时候并不能保证A一定比B任务先被执行，所以该成员变量的意义就是某用户任务在尚未全部被执行的情况下，有新任务到来后，保证只能被一个工作线程所执行，确保先后顺序。执行完所有任务后，方可把该成员变量恢复，下次有新数据的时候，才根据哪个工作线程的任务列表数量少的来存放新任务。

ClassTcpNet类: 其定义了一个网络TCP类,主要功能除了Socket Bind Listen的网络基本操作设置外 , 然后还用了Epoll来处理**客户端的连接**,以及**客户端发送过来的数据**. Epoll使用了边沿触发,边沿触发是当有数据变动的时候,套接字会被检测并且返回, 通过while循环搭配Epoll来不断处理接收到的信息。接收到的数据存在了一个名为**epoll\_event**的事件结构里面了,由于Epoll的监听返回是一个Int值,也就是有多少套接字有数据变动的意思,所以通过for循环来遍历每个有变动的套接字,便可读取到接收到的每个套接字的事件数据。

由于TCP是流数据,所以我们读取流数据的时候会存在着**粘包**的情况, 这种情况我们可以通过给协议**加分隔符**以及给协议**设置一个包头指定长度**来解决协议读取问题。

①:假如有变化的套接字是服务器的监听套接字,也就是有客户端请求连接,我们可以把新连接的客户端通过**epoll\_ctl**来加入Epoll监听。

②:假如有变化的套接字是客户端的套接字,我们就通过while循环一次性读取完数,因为使用的是边沿触发,不读取完,残留数据要等待到该套接字下次有数据变化的时候才会发现。读取完数据后，封装成任务包，根据客户端类对象的**用户当前任务数量**以及**当前所处任务线程索引**来派发任务。

1. **OffLineSystem :** 该模块主要是用于提供离线数据的存取

启服的时候，程序会从Mysql中加载用户离线数据进Redis，然后用户需要获取离线数据的时候，直接调用该模块的函数即可获取到需要的离线数据，停服的时候也会从Redis中把所有离线数据存回进入Mysql中。

1. **Protocol :** 该文档内是所有协议的文件，作用是主要用来查看,具体细节查看文件夹中

名为\_README的文件内容(需要注意的是协议要用JSON格式)。

1. **Test :** 内存放着测试用例，可以运行测试用例来验证服务器是否运行正常。
2. **其余文件 :**

①ClassServer定义了一个服务器类，主要保存一些服务器相关的数据供全局使用

②GlobalVariable里面提供一些全局函数

③Server文件是整个项目的入口文件，也是服务器的启流程。

④Link.sh 是编译文件，把项目编译为可执行文件