JstChat5

2025年6月2日 9:28

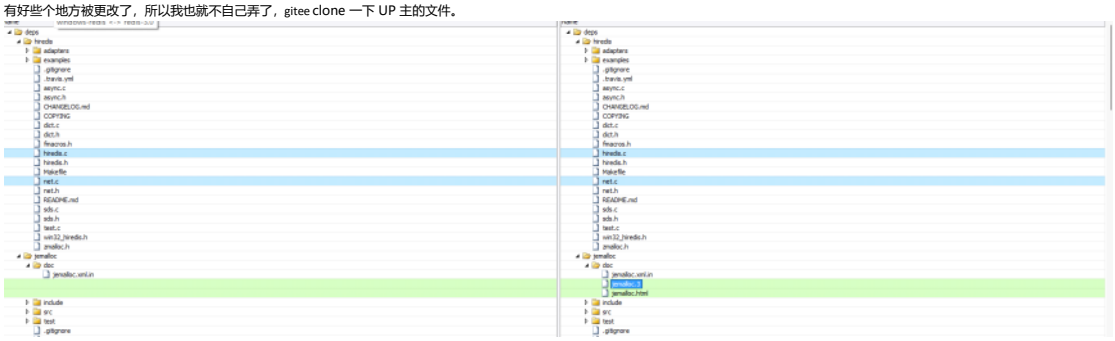
---------- Ep11 Redis 的配置 ----------

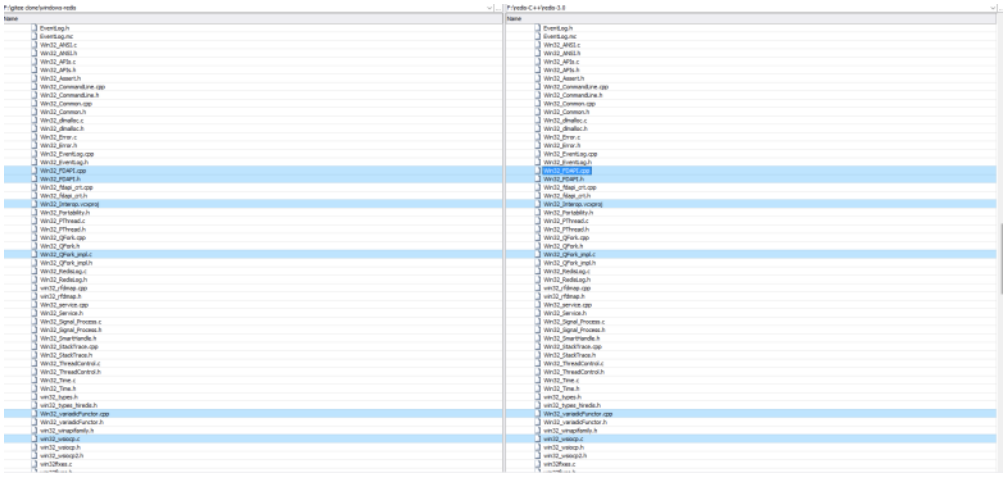
》》》》UP主的 redis 服务好像是搭建在自己私有的腾讯云服务器上: 81.68.86.146，并使用了 81.68.86.146 这个端口（redis 默认端口 6379）
但是 UP 主在启动 redis server 时没有特别标明私有的服务器，而且 redis 的配置文件中也是默认 127.0.0.1 的，所以我在想，如果要是使用 UP 主服务器上的 redis，是不是应该修改配置文件？

UP 主在C++的测试代码中直接使用了81.68.86.146 : 6380，在使用 redis desktop manager 创建连接时也直接使用了这个 IP。

```
# By default, if no "bind" configuration directive is specified, Redis listens
# for connections from all the network interfaces available on the server.
# It is possible to listen to just one or multiple selected interfaces using
# the "bind" configuration directive, followed by one or more IP addresses.
#
# Examples:
#
# bind 192.168.1.100 10.0.0.1
# bind 127.0.0.1 ::1
#
# ~~~ WARNING ~~~ If the computer running Redis is directly exposed to the
# internet, binding to all the interfaces is dangerous and will expose the
# instance to everybody on the internet. So by default we uncomment the
# following bind directive, that will force Redis to listen only into
# the IPv4 loopback interface address (this means Redis will be able to
# accept connections only from clients running into the same computer it
# is running).
#
# IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
# JUST COMMENT THE FOLLOWING LINE.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
bind 127.0.0.1

# Protected mode is a layer of security protection, in order to avoid that
# Redis instances left open on the internet are accessed and exploited.
#
# When protected mode is on and if:
#
# 1) The server is not binding explicitly to a set of addresses using the
#    "bind" directive.
# 2) No password is configured.
#
# The server only accepts connections from clients connecting from the
# IPv4 and IPv6 loopback addresses 127.0.0.1 and ::1, and from Unix domain
# sockets.
#
```

》》》》对于UP对 win32_FDAPI.h 等文件所做的更改:
有好些个地方被更改了，所以我也就不自己弄了，gitee clone 一下 UP 主的文件。

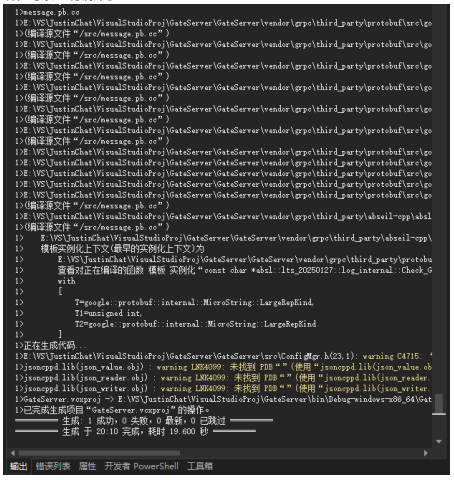


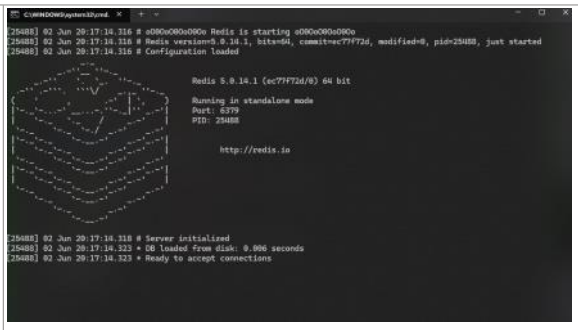
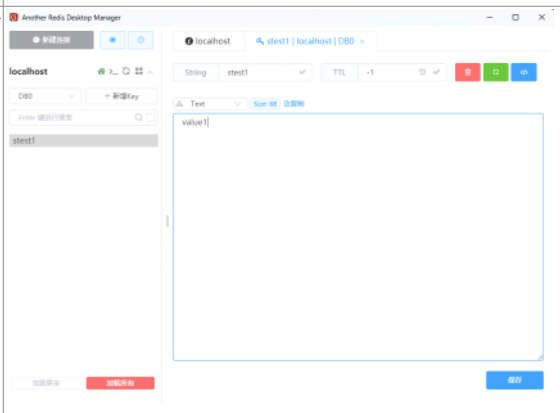


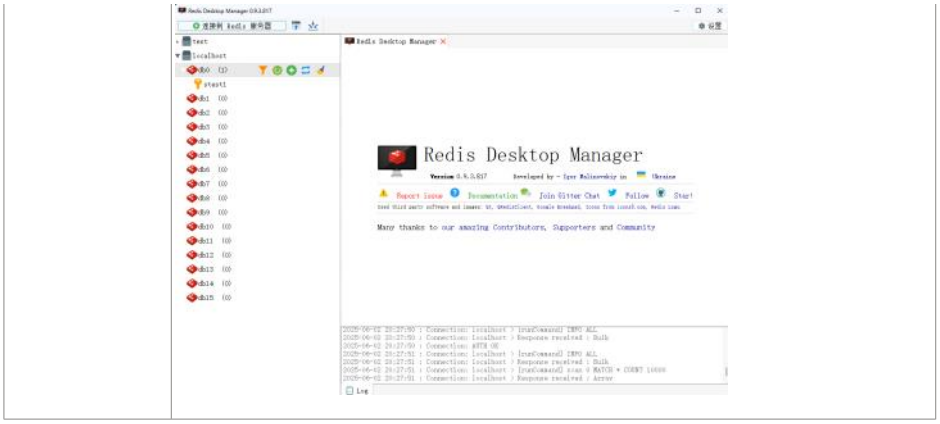
随便找个地方 clone 一下文件：（指令）git clone https://gitee.com/secondtonone1/windows-redis.git
然后将克隆后的文件全选，清除之前下载的 redis 库中的所有文件，然后将 UP 主提供的放置进去。
同时我们打开 .sln 文件，手动重新编译一下，生成 Lib 库。

我们将新生成的 lib 库放在项目中，替换之前的 Lib 库。
同时 dep 和 src 中的文件也要更新。

现在可以正确编译了:



启动 redis server 之后	
我们运行代码，并得到结果	
查看一下 DesktopManager	 或者这个软件:



OK

》》》》为了不用手动输入命令去启动 redis server，我特地写了一个 bat 文件。这个文件可以放在桌面，需要启动 redis server 时，双击该文件即可。

```
@echo off
cd /d F:\RedisServer-x64-5.0.14.1
.\redis-server.exe .\redis.windows.conf
```

这个文件的作用就是去F盘的目录下运行指令 .\redis-server.exe .\redis.window.conf

---------- Ep 12 Redis 的封装与 Redis 连接池 ----------

》》》》this 指针的问题

UP 主通篇都采用 this 来调用成员变量，我怎么觉得没有这个必要？直接写 _connect 也行吧。

```
bool RedisMgr::Connect(const std::string &host, int port)
{
    this->_connect = redisConnect(host.c_str(), port);
    if (this->_connect != NULL && this->_connect->err)
    {
        cout << "connect error " << this->_connect->errstr << std::endl;
        return false;
    }
    return true;
}
```

》》》》封装的函数

除了 Connect 函数，所有封装的函数都是一样的设计思路：

```
bool RedisMgr::Get(const std::string& key, std::string& value)
{
    m_Reply = (redisReply*)redisCommand(m_Connect, "GET %s", key.c_str());
    if (m_Reply == NULL) {
        JC_CORE_ERROR("[ GET {}] failed", key);
        freeReplyObject(m_Reply);
        return false;
    }

    if (m_Reply->type != REDIS_REPLY_STRING) {
        JC_CORE_ERROR("[ GET {}] failed", key);
        freeReplyObject(m_Reply);
        return false;
    }

    value = m_Reply->str;
    freeReplyObject(m_Reply);
    JC_CORE_TRACE("Succeed to execute command [ GET {}]", key);
    return true;
}
```

使用 redisCommand 执行数据库语句，并将返回值赋予给 m_Reply;

使用 m_Reply 的值进行判断，如果执行失败，则输出日志，并释放 m_Reply 的内存

如果没有失败，则输出日志（执行成功），并释放 m_Reply 的内存，然后将函数 RedisMgr::Get 的返回值标记为 true

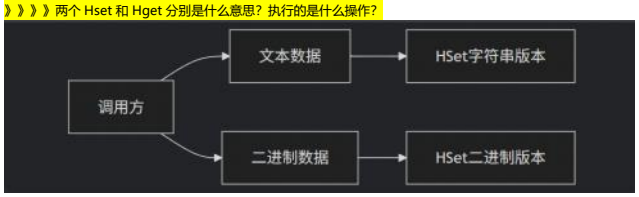
Connect 函数只需要通过 redisConnect 来判断是否连接成功，并适时输出日志

```
bool RedisMgr::Connect(const std::string& host, int port)
{
    m_Connect = redisConnect(host.c_str(), port);
    if (m_Connect != NULL && m_Connect->err)
    {
        JC_CORE_ERROR("connect error: {}", m_Connect->errstr);
        return false;
    }
    return true;
}
```

这里的 m_Connect 会在 RedisMgr::Close() 函数中被销毁。

```
void RedisMgr::Close()
{
    redisFree(m_Connect);
}
```

》》》》两个 Hset 和 Hget 分别是什么意思？执行的是什么操作？



1. HSet

HSet 函数用于在 Redis 中设置哈希表（hash）中的一个字段的值。如果指定的哈希表不存在，Redis 会自动创建它。

第一种重载：	bool RedisMgr::HSet(const std::string& key, const std::string& hkey, const std::string& value)
作用：	通过 redisCommand 函数发送 HSET 命令到 Redis，格式为 HSET key hkey value。 这种方式适合使用字符串作为参数（比如传输文本数据 Json ...）。
参数：	key 是哈希表的名称，hkey 是字段名，value 是字段对应的值。
返回值：	如果执行失败，返回 false；成功则返回 true。

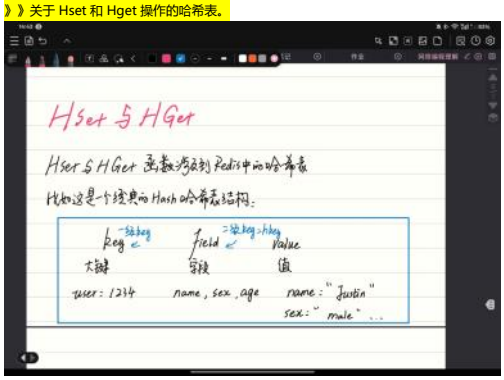
第二种重载：	bool RedisMgr::HSet(const char* key, const char* hkey, const char* hvalue, size_t hvaluelen)
作用：	通过 redisCommandArgv 使用更灵活的方式来发送命令。 这种方式适用于不直接使用 std::string 类型的数据。但是由于参数传递方式不同，适合处理二进制数据，能够指定每个参数的长度。

2. Hget

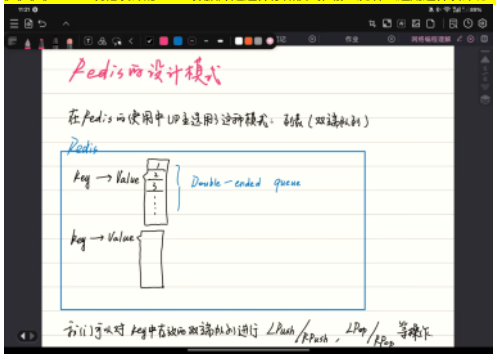
HGet 函数用于从 Redis 中获取哈希表某个字段的值。

函数签名：	std::string RedisMgr::HGet(const std::string& key, const std::string& hkey)
作用：	使用 redisCommandArgv 发送 HGET key hkey 命令到 Redis，尝试获取指定 key（哈希表）的 hkey（字段）对应的值。 这种方法适用于从哈希表中获取值，返回类型为 std::string，方便后续处理
返回值：	如果该字段存在，返回该字段的值；如果字段不存在，则返回一个空字符串，并打印错误日志

》》关于 Hset 和 Hget 操作的哈希表。



》》》》UP 主说他设计的 redis 数据库是这种存储模式，那么为什么选用这种设计呢？



在网络编程中，使用 Redis 列表结构（尤其是通过 LPUSH 和 RPUSH 命令操作队列）非常适合处理以下几种情况和需求：

1. 消息队列（Message Queue）

- 应用场景：在分布式系统中，组件之间常常需要传递消息。Redis 的队列结构（列表）非常适合实现轻量级的消息队列。生产者将消息通过 LPUSH 或 RPUSH 放入队列，消费者从队列中通过 LPOP 或 RPOP 获取消息并进行处理。

- 原因：Redis 提供了高性能的队列操作，支持多客户端并发读取与写入，并且 LPUSH 和 RPUSH 能够以常数时间复杂度 O(1) 执行，因此对于频繁的消息传递与处理非常高效。

2. 任务调度
   - 应用场景：在一些任务调度系统中，可以通过队列来管理任务。每个任务可以是一个处理单元，任务生产者通过 LPUSH 将任务加入队列，而任务消费者（例如工作线程）通过 LPOP 取出并执行任务。
   - 原因：列表结构保证了任务的先进先出（FIFO）顺序，可以确保任务按顺序被处理。而 Redis 列表的高效读写特性，使得其非常适合用作实时任务调度系统中的队列。
3. Web 请求队列（HTTP 请求排队）
   - 应用场景：在一些 Web 服务中，可以使用 Redis 列表来实现请求排队。客户端请求可以通过 LPUSH 加入队列，后端服务可以按顺序从队列中取出请求并处理。
   - 原因：由于 Redis 列表支持高效的插入和删除操作，它非常适合用于高并发环境下的请求排队和负载均衡。
4. 分布式锁（Distributed Lock）
   - 应用场景：分布式系统中常常需要对资源进行并发控制。Redis 列表可以被用作实现分布式锁的队列。例如，可以使用一个 Redis 列表来存储等待获取锁的客户端 ID，当锁释放时，通过 LPOP 操作从队列中取出下一个等待的客户端。
   - 原因：Redis 提供了高效的列表操作，可以确保在多个进程或服务之间进行分布式锁控制时，队列中的元素按顺序被处理。

为什么选用这种队列方式?

Redis 的队列（列表）结构非常适合需要高并发、低延迟、顺序处理的场景，广泛应用于消息队列、任务调度、流量控制、日志记录等领域。选择 Redis 列表作为队列方式，主要是基于其高效的插入和删除操作、持久化功能、以及分布式支持等优势，使得其在实际应用中非常有价值。

》》》》封装的函数中，有一些代码是什么意思？？

这个 value 感觉一点用没有	`bool RedisMgr::Get(const std::string& key, std::string& value)`
	`{`
	`  m_Reply = (redisReply*)redisCommand(m_Connect, "GET %s", key.c_str());`
	`  if (m_Reply == NULL) {`
	`  JC_CORE_ERROR("[ GET {}] failed", key);`
	`  freeReplyObject(m_Reply);`
	`  return false;`
	`  }`
	`  if (m_Reply->type != REDIS_REPLY_STRING) {`
	`  JC_CORE_ERROR("[ GET {}] failed", key);`
	`  freeReplyObject(m_Reply);`
	`  return false;`
	`  }`
	`  value = m_Reply->str;`
	`  freeReplyObject(m_Reply);`
	`  JC_CORE_TRACE("Succeed to execute command [ GET {}]", key);`
	`  return true;`
	`}`

这个 value 感觉一点用没有	`bool RedisMgr::LPop(const std::string& key, std::string& value) {`
	`  m_Reply = (redisReply*)redisCommand(m_Connect, "LPOP %s ", key.c_str());`
	`  if (m_Reply == nullptr || m_Reply->type == REDIS_REPLY_NIL)`
	`  {`
	`  JC_CORE_ERROR("Execut command [ LPOP {}] failure ! ", key);`
	`  freeReplyObject(m_Reply);`
	`  return false;`
	`  }`
	`  value = m_Reply->str;`
	`  JC_CORE_TRACE("Execut command [ LPOP {}] success ! ", key);`
	`  freeReplyObject(m_Reply);`
	`  return true;`
	`}`

这个 value 感觉一点用没有	`bool RedisMgr::RPop(const std::string& key, std::string& value) {`
	`  m_Reply = (redisReply*)redisCommand(m_Connect, "RPOP %s ", key.c_str());`
	`  if (m_Reply == nullptr || m_Reply->type == REDIS_REPLY_NIL) {`
	`  JC_CORE_ERROR("Execut command [ RPOP {}] failure ! ", key);`
	`  freeReplyObject(m_Reply);`
	`  return false;`
	`  }`
	`  value = m_Reply->str;`
	`  JC_CORE_TRACE("Execut command [ RPOP {}] success ! ", key);`
	`  freeReplyObject(m_Reply);`
	`  return true;`
	`}`

想了一会，觉得应该是这样：

示例:	`std::string value;`
	`RedisMgr redisMgr;`
	`bool success = redisMgr.Get("user:1001", value);`

调用 GET 命令：	当你调用 redisMgr.Get("user:1001", value) 时，redisCommand 向 Redis 发送了一个 GET user:1001 命令。 Redis 返回响应: Redis 处理这个 GET 命令并返回一个响应，其中包含了与 user:1001 关联的值（即 "Alice"）。这个响应被存储在 this->_reply 中。 赋值操作: this->_reply->str 就是 Redis 返回的那个值，它是一个 C 风格字符串。 在 value = this->_reply->str; 这一行中，你将这个字符串值（即 "Alice"）赋给了 value 变量。 返回值: 函数最后会将 true 返回，表示成功从 Redis 获取到了值，调用者就可以在 value 中看到 "Alice"。

也就是说，你填入的参数在函数中会被隐式的赋值，这个参数将会获取的值就是 m_Reply->str

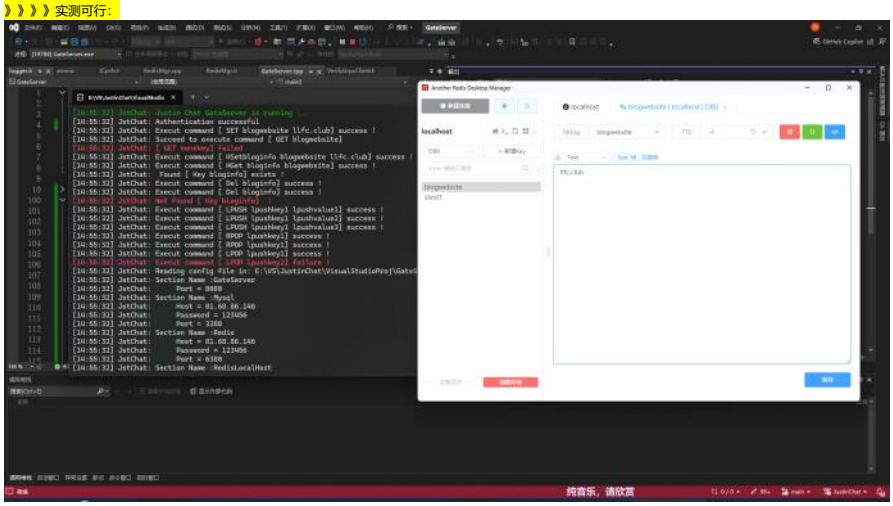
》》那这里的 **m_Reply->str** 包含的是什么信息?

这里的 value 就是从 key 中查询出来的值（"user":"1234"，value 就是这个 string “1234”）

》》还有一些条件判断： strcmp 的作用是进行字符串的比较操作

```cpp
bool RedisMgr::Set(const std::string& key, const std::string& value) {
    //执行redis命令行
    m_Reply = (redisReply*)redisCommand(m_Connect, "SET %s %s", key.c_str(), value.c_str());

    //如果返回NULL则说明执行失败
    if (NULL == m_Reply)
    {
        JC_CORE_ERROR("Execut command [ SET {} {}] failure ! ", key, value);
        freeReplyObject(m_Reply);
        return false;
    }

    //如果执行失败则释放连接
    if (!(m_Reply->type == REDIS_REPLY_STATUS && (strcmp(m_Reply->str, "OK") == 0 || strcmp(m_Reply->str, "ok") == 0)))
    {
        JC_CORE_ERROR("Execut command [ SET {} {}] failure ! ", key, value);
        freeReplyObject(m_Reply);
        return false;
    }

    //执行成功 释放redisCommand执行后返回的redisReply所占用的内存
    freeReplyObject(m_Reply);
    JC_CORE_TRACE("Execut command [ SET {} {}] success ! ", key, value);
    return true;
}
```

》》》》实测可行：



》》》》redis 连接池《《《《

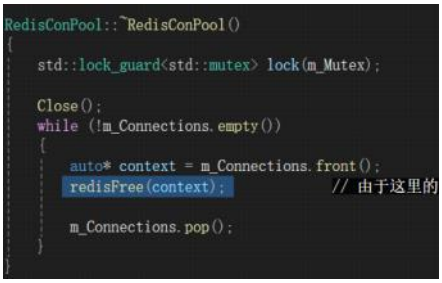
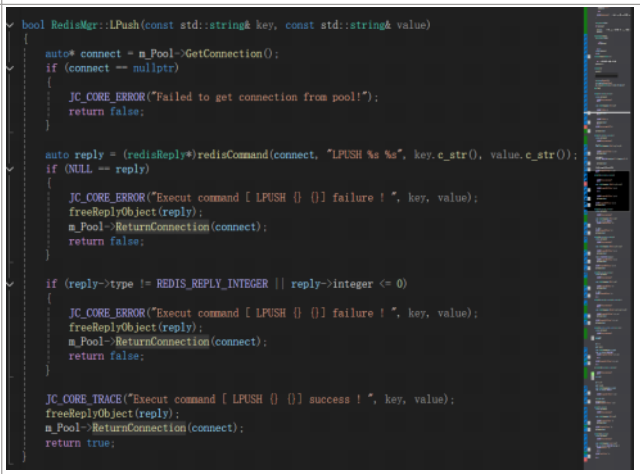
》》》》关于队列的 pop 成员函数

```cpp
RedisConPool::~RedisConPool()
{
    std::unique_lock<std::mutex> lock(m_Mutex);

    Close();
    while(!m_Connections.empty())
    {
        m_Connections.pop();
    }
}
```

队列的成员函数 pop() 没有参数，默认删除队列的第一个元素。（队首元素）

》》》》没啥要记的了

》》》》有几个问题注意一下：

需要再连接池的析构函数中，手动销毁一下 context	
设计了 RedisConPool 之后， 其实 redisMgr 中封装的 Auth 和 Connect 函数可以删除了，因为创建 RedisConPool 时已经做过了这样的操作（并验证过了连接的正确性）	
每一个redis封装的函数，都要做修改。 不仅 connect 和 reply 要改成临时变量。 **还要记得使用完连接之后，要将连接返回池中。** （这里使用 LPush 举例， 每一个函数都要更改）	 由于一些步骤比较重复（比如每一个函数中都需要检查从连接池获取的连接是否有效），也可以考虑写成宏定义。 我就懒得弄了。
我看评论区中有人说 singleton 会发生问题，我倒也遇到了，不过是因为我尝试将 configMgr.h 包含在预编译头文件中。 后面我只在使用 configMgr 的 .h文件中包含了 configMgr.h，就没啥问题。	

---------- Ep 13----------

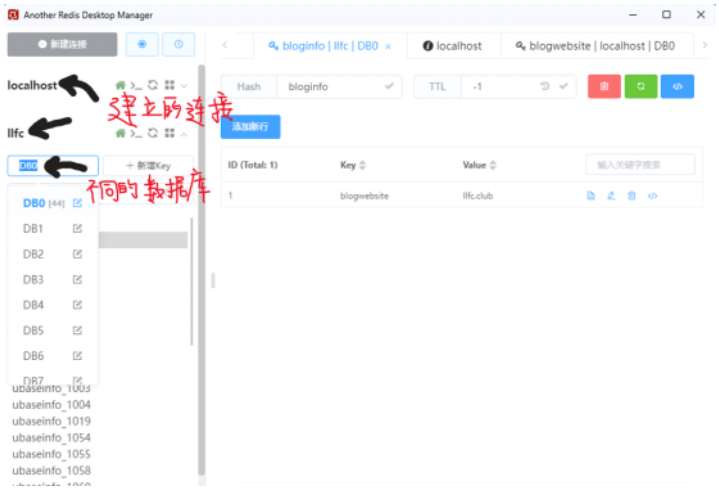
》》》》在对 redis 数据库存储 verifyCode 的时候，我发现 UP 并没有特别指定存储在某一个表中，于是我查阅了一下：
host: config_module.redis_host, // Redis服务器主机名
port: config_module.redis_port, // Redis服务器端口号
password: config_module.redis_passwd, // Redis密码
整个代码中只获取了：host,port,password这三个信息。

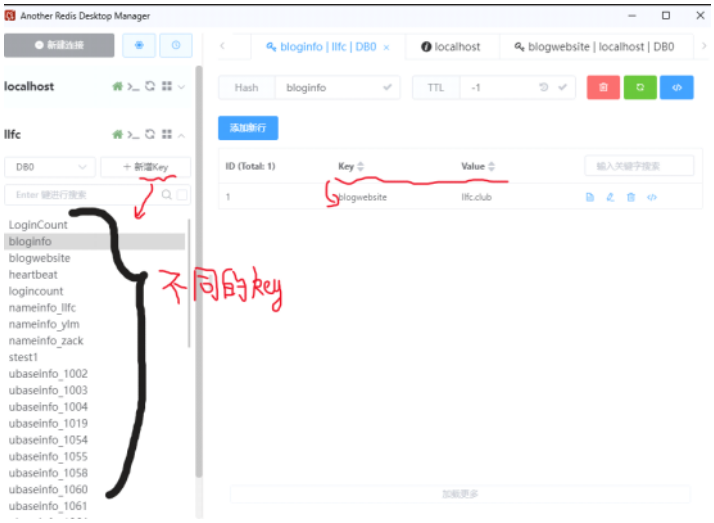
理解：
在 Redis 中，数据并不像关系型数据库（例如 MySQL 或 PostgreSQL）那样被存储在不同的“表”中。Redis 数据库是基于内存的，并且没有传统的表、行等结构。

Redis 是一个键值数据库（Key-Value Store），它通过键（key）来访问数据，而不需要显式指定表的概念。
在 Redis 中，你可以将数据存储在默认的数据库中，或者指定不同的数据库编号（0-15，Redis 默认有 16 个数据库，编号从 0 到 15）。因此 Redis 客户端的配置一般不需要显式地指定数据库名称，而是通过数据库编号来选择数据库。
（默认情况下，Redis 默认会连接到数据库编号为 0 的数据库。如果你希望使用其他数据库，可以在客户端配置中使用 select 方法来切换数据库。）

例如：

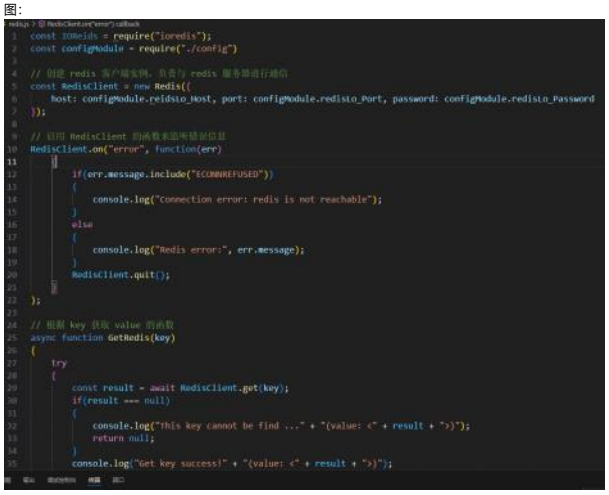
```
// 切换到数据库编号为 1 的数据库
await RedisCli.select(1);
```





》》》》关于 redis.js 文件被调用的流程理解

图：



```
const IORedis = require("ioredis");
const configModule = require("./config")

// 创建 redis 客户端实例，负责与 redis 服务器进行通信
const RedisClient = new Redis({
    host: configModule.redisto_Host, port: configModule.redisto_Port, password: configModule.redisto_Password
});

// 启用 RedisClient 的函数来监听错误信息
RedisClient.on("error", function(err)
{
    if(err.message.include("ECONNREFUSED"))
    {
        console.log("Connection error: redis is not reachable");
    }
    else
    {
        console.log("Redis error:", err.message);
    }
    RedisClient.quit();
});

// 根据 key 获取 value 的函数
async function GetRedis(key)
{
    try
    {
        const result = await RedisClient.get(key);
        if(result === null)
        {
            console.log("This key cannot be find ..." + "(value: <" + result + ">)");
            return null;
        }
        console.log("Get key success!" + "(value: <" + result + ">)");
```

让我们仔细分析一下这个文件被包含时发生的操作：

1.模块加载和执行顺序、RedisCli 实例和事件监听

• 在 Node.js 中，当你使用 require() 加载一个模块时，该模块中的代码会被立即执行。 也就是说，所有顶层的代码（包括变量定义、函数声明、对象初始化等）都会在模块被加载时执行。 • 在 redis.js 中，顶层的代码包括： ○ 引入 config_module 和 ioredis。 ○ 创建 RedisCli 实例，并配置 Redis 连接。 const RedisCli = new Redis({...}) 这一行代码在模块加载时立即创建了一个 Redis 客户端实例。 ○ 监听 Redis 错误事件。 RedisCli.on("error", function (err) {...}) 这行代码设置了一个事件监听器，它会在 Redis 客户端发生错误时触发。事件监听器会被立即注册。	**这些操作是在加载模块时完成的，当另一个文件 require() 这个模块时，这些操作会立即执行，即使这些操作的结果没有直接被导出或返回。** **但它们并不会在外部文件中被“再次执行”。**

2. 导出的函数

你定义了 GetRedis、QueryRedis、SetRedisExpire 和 Quit 函数，并将它们通过 module.exports 导出。这意味着，外部文件可以使用 require() 加载这个模块并调用这些函数。	**这些函数 不会在模块加载时自动执行，只有在外部文件显式调用时才会执行。例如，GetRedis 函数会在外部文件调用时执行，SetRedisExpire 函数会在外部调用时执行。**

总结

当另一个文件使用 require() 引入这个模块时，

以下操作会立即发生：	○ Redis 客户端实例 RedisCli 被创建。 ○ 错误监听器通过 RedisCli.on() 注册。
不会立即发生的：	导出的函数（例如 GetRedis、SetRedisExpire 等）并不会在模块加载时执行，只有当另一个文件调用这些函数时，它们才会被执行。

》》》》优化回调函数中的报错逻辑

```
// 启用 RedisClient 的函数来监听错误信息
RedisClient.on("error", function(err)
    {
        if(err.message.includes("ECONNREFUSED"))
        {
            console.log("Connection error: redis is not reachable");
        }
        else
        {
            console.log("Redis error:", err.message);
        }
        RedisClient.quit();
    }
);
```

》》》》" === "是 java script 中的什么操作符？有什么作用?

"==="

定义：	在 JavaScript 中，=== 是 严格相等 操作符（Strict Equality Operator）。
作用：	=== 用来比较两个值是否 完全相等，不仅要求值相等，还要求它们的类型相同。如果两个值 类型不同，即使它们的值看起来相同，=== 也会返回 false。

例子：	相等且类型相同：5 === 5 // true
	'hello' === 'hello' // true
	类型不同：5 === '5' // false，类型不同（一个是数字，一个是字符串）
	true === 1 // false，类型不同（一个是布尔值，一个是数字）
	null 和 undefined 的比较：null === undefined // false，因为它们的类型不同

"==" 和 "===" 的区别？？	==：比较时会做 类型转换，即如果两个值类型不同，会尝试将它们转换成相同类型后再比较。 ===：不会做类型转换，要求值和类型都完全相同。 例如：'5' == 5 // true，字符串会被转换成数字 '5' === 5 // false，类型不同（一个是字符串，一个是数字）

C++ 中的 ==（C++中没有 ===，只有 ==）

C++ 中的 == ：	C++ 是一种 **强类型语言**，它的比较操作符 == 会考虑 **隐式类型转换**

》》》》如果要使用 js 中的模板占位符，需要用反引号标记字符串：

```
    }
    console.log(`Value is null, regenerating verification code... (Key:${call.request.email})`);
```

${} 是占位符的标识，中间是变量。输出时变量会变化为对应值：

```
This key cannot be find ...(value: <null>)
Value is null, regenerating verification code... (Key:          ).com)
This key cannot be find ...(value: <null>)
```

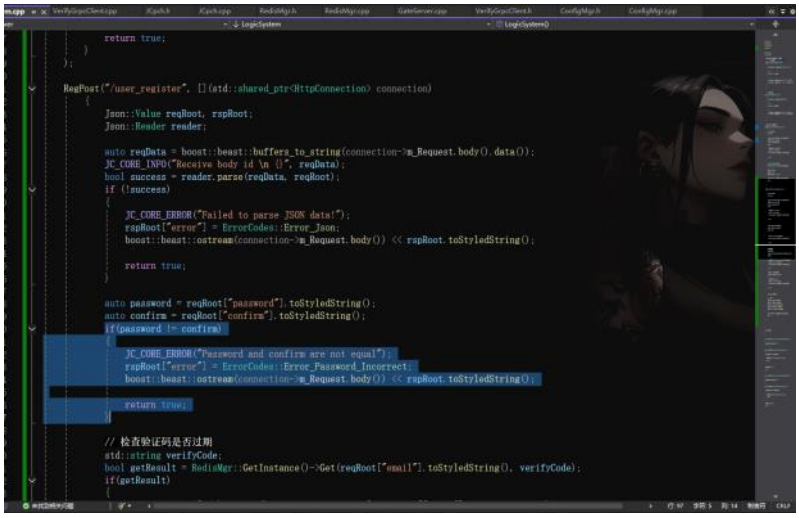
反引号这个标点的键位在ESC键位之下。（Shift + ` => ~）

》》》》在 QT 中的 password 与 confirm 的比较

》》》》在 GateServer 中的 password 与 confirm 的比较

》》》》TODO: 这两段代码有什么不同？（没想出来？）

```cpp
void RegisterDialog::on_confirm_button_clicked()
{
    if(ui->user_edit->text() == "")
    {
        ShowTip(tr("用户名不能为空"), false);
        return;
    }

    if(ui->email_edit->text() == "")
    {
        ShowTip(tr("邮箱不能为空"), false);
        return;
    }

    if(ui->pass_edit->text() == "")
    {
        ShowTip(tr("密码不能为空"), false);
        return;
    }

    if(ui->confirm_edit->text() == "")
    {
        ShowTip(tr("确认密码不能为空"), false);
        return;
    }

    if(ui->confirm_edit->text() != ui->pass_edit->text())
    {
        ShowTip(tr("两次密码不相同，请确认后重新尝试"), false);
        return;
    }

    if(ui->verify_edit->text() == "")
```



```cpp
        return true;
    }
);

RegPost("/user_register", [](std::shared_ptr<HttpConnection> connection)
    {
        Json::Value reqRoot, rspRoot;
        Json::Reader reader;

        auto reqData = boost::beast::buffers_to_string(connection->m_Request.body().data());
        JC_CORE_INFO("Receive body is \n {}", reqData);
        bool success = reader.parse(reqData, reqRoot);
        if (!success)
        {
            JC_CORE_ERROR("Failed to parse JSON data!");
            rspRoot["error"] = ErrorCodes::Error_Json;
            boost::beast::ostream(connection->m_Request.body()) << rspRoot.toStyledString();

            return true;
        }

        auto password = reqRoot["password"].toStyledString();
        auto confirm = reqRoot["confirm"].toStyledString();
        if(password != confirm)
        {
            JC_CORE_ERROR("Password and confirm are not equal");
            rspRoot["error"] = ErrorCodes::Error_Password_Incorrect;
            boost::beast::ostream(connection->m_Request.body()) << rspRoot.toStyledString();

            return true;
        }

        // 检查验证码是否过期
        std::string verifyCode;
        bool getResult = RedisMgr::GetInstance()->Get(reqRoot["email"].toStyledString(), verifyCode);
        if(getResult)
        {
```

》》》》注意：不要错用成 toStyleString() 了

```cpp
    // 检查验证码是否过期
    std::string verifyCode;
    bool getResult = RedisMgr::GetInstance()->Get(reqRoot["email"].asString(), verifyCode);
    if(!getResult)
    {
        JC_CORE_ERROR("Can't find {} in redis database!", reqRoot["email"].toStyledString());
        rspRoot["error"] = ErrorCodes::Error_Verify_Expired;
        boost::beast::ostream(connection->m_Response.body()) << rspRoot.toStyledString();

        return true;
    }
    // 检查验证码与 redis 数据库中的数据是否匹配
    if (reqRoot["email"].asString() != verifyCode)
    {
        JC_CORE_ERROR("Verify code error!");
        rspRoot["error"] = ErrorCodes::Error_Verify_Code;
        boost::beast::ostream(connection->m_Response.body()) << rspRoot.toStyledString();

        return true;
    }
```

---------- Ep14 ----------

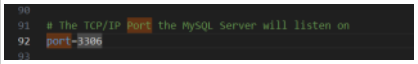
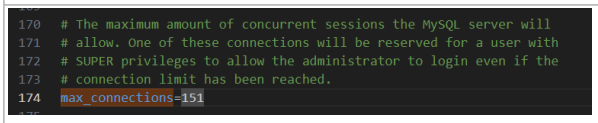
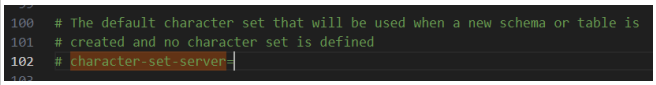
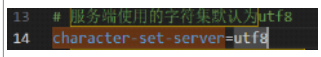
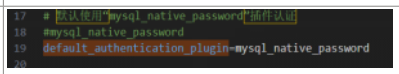
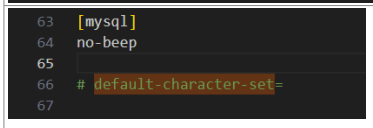
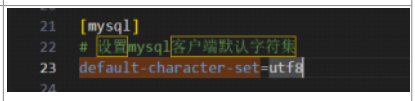
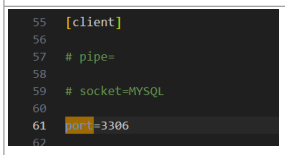
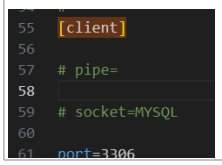
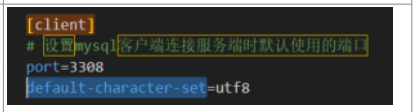
》》》》今天早上才知道可以手动向 cmd 中直接拖动文件：

向cmd拖动文件

》》》》检测一下电脑上安装的 mysql 版本：（因为我之前安装过一次）

```
Microsoft Windows [版本 10.0.22631.5335]
(c) Microsoft Corporation，保留所有权利。

C:\Users\JustinBieber>mysql
ERROR 2003 (HY000): Can't connect to MySQL server on 'localhost:3306' (10061)

C:\Users\JustinBieber>mysql --version
mysql  Ver 8.0.37 for Win64 on x86_64 (MySQL Community Server - GPL)
```

》》》》我发现在我使用过的 mysql 文件中，已经存在一个 my.ini 了，这个应该是默认的。我把一些不同的地方摘出来

Ini 文件默认内容	自定义 ini 文件中的内容
	
	
	
	
（[mysqld]中没有找到这个关键字） 	
	
	
（[client]中没有找到这个关键字） 	

》》》》mysql 中的 ini 文件加载规则

MySQL默认会按特定顺序搜索配置文件，通常只需保留一个主配置文件即可。

加载规则：	MySQL按以下顺序查找 my.ini/my.cnf： 1.%PROGRAMDATA%\MySQL\MySQL Server X.Y\my.ini（Windows默认位置） 2.%MYSQL_HOME%\my.ini（需设置环境变量 MYSQL_HOME） 3.安装目录下的 my.ini（如 E:\mysql\install\my.ini） 4.数据目录下的 my.ini（如 E:\mysql\database\my.ini） 如果两个目录均存在配置文件，MySQL会按顺序加载并合并配置，后加载的配置会覆盖前者。
操作方式	• 可以运行 sql 语句查看当前加载的文件路径： 命令： SHOW VARIABLES LIKE 'config_file'; • 也可以使用 cmd 启动 mysql 服务时，使用命令强制指定配置文件的路径： （在安装 mysql 时系统会询问你，是否需要开机自启 sql 服务，如果你选择否，则需要在使用 sql 之前手动启动 sql 服务。 你可以在 window 提供的操作界面启动 sql 服务，也可以自己手动运行命令） 命令： mysqld --defaults-file=E:\mysql\database\my.ini

我就把那些在我的 ini 文件中没出现的关键字添加一下。

》》》》未进行的操作：

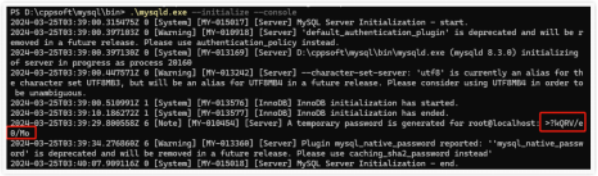
》》》》获取 mysql 的随机密码：

》》》》修改 mysql 密码：

（由于 mysql 是我之前下载的，也就这么一直使用下来了，也不知道当时有没有进行如下操作，这次我就先不执行，看看会不会造成什么问题）

```
//安装mysql 安装完成后mysql会有一个随机密码
.\mysqld.exe --initialize --console
```

如下图，随机密码要记住，以后我们改密码会用到



接下来在cmd执行第二条命令

```
//安装mysql服务并启动
.\mysqld.exe --install mysql
```

如果出现以下情况，说明cmd不是以管理员形式执行的，改用为管理员权限执行即可。

继续在cmd上执行以下命令

```
mysql -uroot -p
```

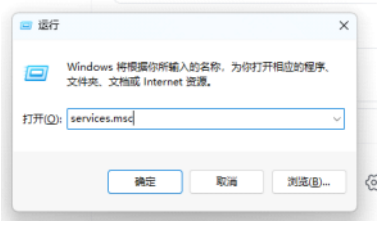
回车后输入上面安装时保存的初始密码，进入mysql里面：

在mysql里面继续执行以下命令：

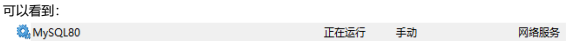
```
//修改密码为:123456
ALTER USER 'root'@'localhost' IDENTIFIED BY '123456';
```

回车按照指引执行完后，代表密码修改成功，再输入exit;退出即可

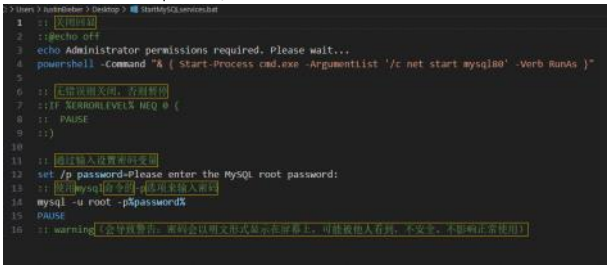
》》》》查看/运行 mysql 服务



可以看到：



我选择使用 .bat 文件启动 sql 服务：（写一个 xx.bat 文件，放置在桌面上，双击运行。以下是 bat 文件启动 sql 服务的代码）



```
:: 以管理员
::@echo off
echo Administrator permissions required. Please wait...
powershell -Command "& { Start-Process cmd.exe -ArgumentList '/c net start mysql80' -Verb RunAs }"

:: 无错误则关闭，否则暂停
::IF %ERRORLEVEL% NEQ 0 (
::  PAUSE
::)

:: 通过输入设置密码变量
set /p password=Please enter the MySQL root password:
:: 使用mysql命令行-p或用变量输入密码
mysql -u root -p%password%
PAUSE
:: warning 不会导致警告，密码会以明文形式显示在屏幕上，可能被他人看到，不安全，不影响正常使用!
```

》》》》mysql 的 desktop manager

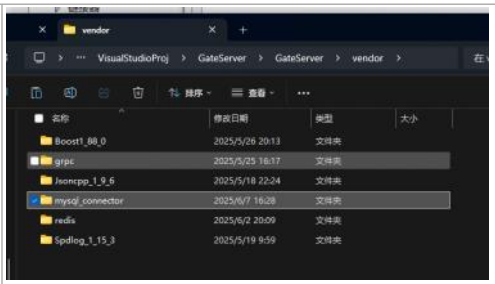
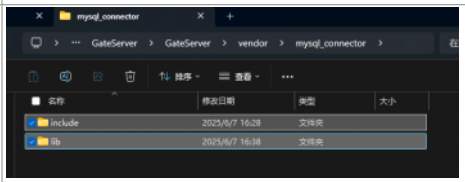
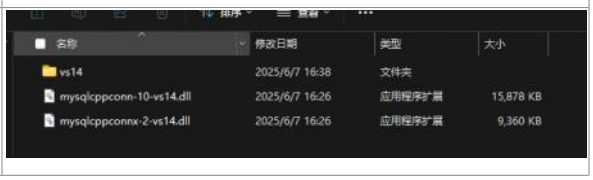
我没有选择 navicat，而是选择了 SQLyog 社区版，免费开源的。

启动 sql 服务之后，在 sqlyog 中测试连接：

》》》》Docker

》》》》暂时没有配置

》》》》Mysql Connector C++

我们在项目的 vendor 目录中新建一个 mysql_connector 文件夹:	
然后在该文件夹中创建两个子文件夹:	
Include 中存放:	
Lib 中存放:	

然后我们在 premake 脚本中包含并附加一下这些文件:

添加包含目录:	

```
-- 添加包含目录
includedirs
{
    "%{prj.name}/src",
    "%{prj.name}/vendor/Boost1_88_0/include",
    "%{prj.name}/vendor/Jsoncpp_1_9_6/include",
    "%{prj.name}/vendor/Spdlog_1_15_3/include",

    "%{prj.name}/vendor/grpc/include",
    "%{prj.name}/vendor/grpc/third_party/abseil-cpp",
    "%{prj.name}/vendor/grpc/third_party/address_sorting/include",
    "%{prj.name}/vendor/grpc/third_party/protobuf/src",
    "%{prj.name}/vendor/grpc/third_party/re2",

    "%{prj.name}/vendor/redis/include/hiredis",

    "%{prj.name}/vendor/mysql_connector/include"
}
```

包含依赖项

```
libdirs
{
    "%{prj.name}/vendor/jsoncpp_1_9_6/lib/Debug",

    "%{prj.name}/vendor/grpc/visual_pro/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/re2/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/base/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/container/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/crc/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/debugging/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/flags/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/hash/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/log/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/numeric/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/profiling/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/random/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/status/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/strings/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/synchronization/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/types/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/abseil-cpp/absl/time/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/boringssl-with-bazel/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/cares/cares/lib/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/protobuf/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/protobuf/third_party/utf8_range/Debug",
    "%{prj.name}/vendor/grpc/visual_pro/third_party/zlib/Debug",

    "%{prj.name}/vendor/redis/lib/Debug",

    "%{prj.name}/vendor/mysql_connector/lib/debug/vs14"
}
```

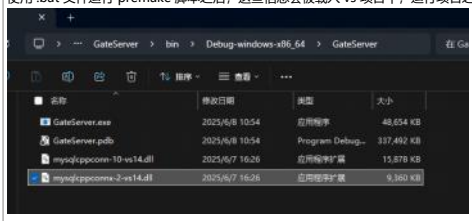
附加依赖项：

```
        "hiredis.lib",
        "Win32_Interop.lib",

        "mysqlcppconn.lib",
        "mysqlcppconn-static.lib",
        "mysqlcppconnx.lib",
        "mysqlcppconnx-static.lib"
    }
```

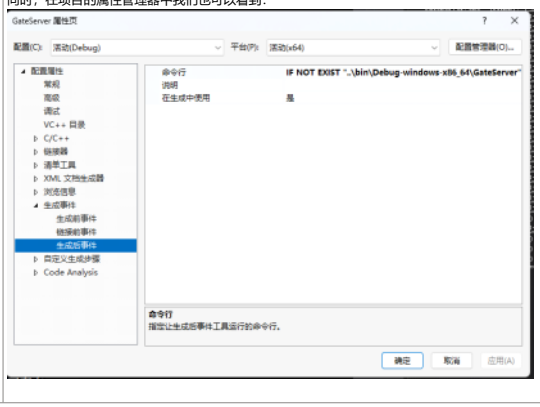
构建项目完成后执行的指令：

```
-- 添加DLL拷贝命令
filter "configurations:Debug"
    postbuildcommands           --构建项目完成后执行的指令
    {
        '{MKDIR} "%{cfg.targetdir}"',
        '(COPY) "E:/VS/JustinChat/VisualStudioProj/GateServer/GateServer/vendor/mysql_connector/lib/debug/mysqlcppconn-10-vs14.dll" "%{cfg.targetdir}"',
        '(COPY) "E:/VS/JustinChat/VisualStudioProj/GateServer/GateServer/vendor/mysql_connector/lib/debug/mysqlcppconnx-2-vs14.dll" "%{cfg.targetdir}"'
    }
```

使用 .bat 文件运行 premake 脚本之后，这些信息会被载入 vs 项目中，运行项目之后我们能够看到文件的确被移动到这里：



同时，在项目的属性管理器中我们也可以看到：



》》》封装 mysql 连接池

》》封装过几次 mysql，但从没想过 Dao 在这里一般是什么意思？？

DAO 是 "Data Access Object"（数据访问对象）的缩写。	它是一个设计模式，用于封装对数据源（如数据库）的访问。 通过 DAO 类，程序的其他部分可以不直接与数据库交互，而是通过 DAO 来进行增、删、改、查等操作。

》》什么是数据库中的 **schema** ??

在数据库中，schema（模式）是数据库对象的组织结构，用来定义数据库中的表、视图、索引、存储过程、用户等对象的结构。它不仅描述了数据库对象的排列方式，还能为不同的用户提供权限控制。

通常包括：

表结构：	定义了表的名称、字段名、数据类型以及各字段的约束（如主键、外键、唯一性等）。
视图：	虚拟表，通过查询语句从一个或多个表中提取数据。
索引：	用于加速数据检索的结构。
存储过程和触发器：	用于封装一组 SQL 操作的程序。
权限管理：	控制哪些用户可以访问或修改数据库中的哪些对象。

》》》》UP 主视频中的代码与文档中的不同，这里记录一下差异：

difference

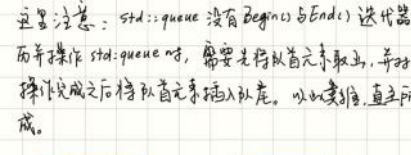
① document doesn't have this class:

② document dose not claim that Config.ini was changed.

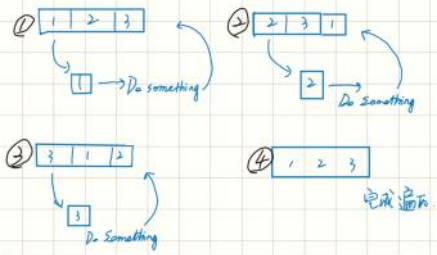
③ document doesn't have this private Variable: _check_thread

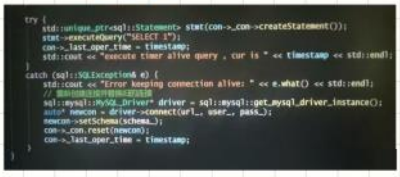
And because of that, the Constructor of MysqlPool was different also.

④ Document doesn't have this member function: checkConnection

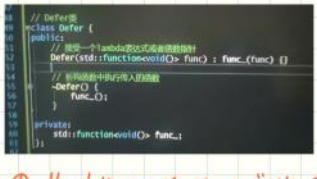


这里注意：std::queue 没有 Begin() 与 End() 迭代器，所以如果要遍历并操作 std::queue 时，需要先将队首元素取出，并对其进行操作，操作完成之后将队首元素插入队尾。以此类推，直至所有元素操作完成。

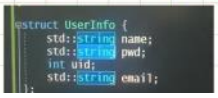




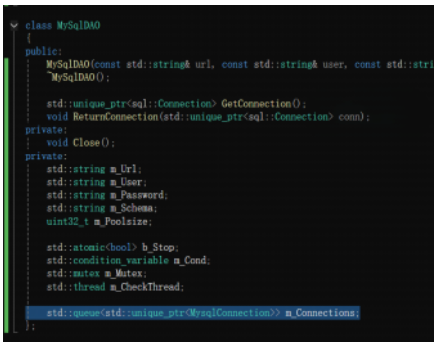
⑤ the defination of "Defer"



⑥ the defination of struct: "UserInfo"

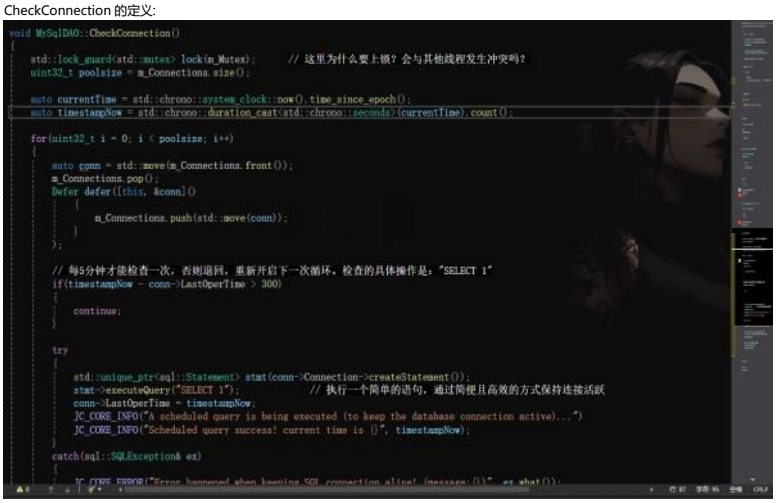


关于 MysqlConnection 这个类涉及到的更改：

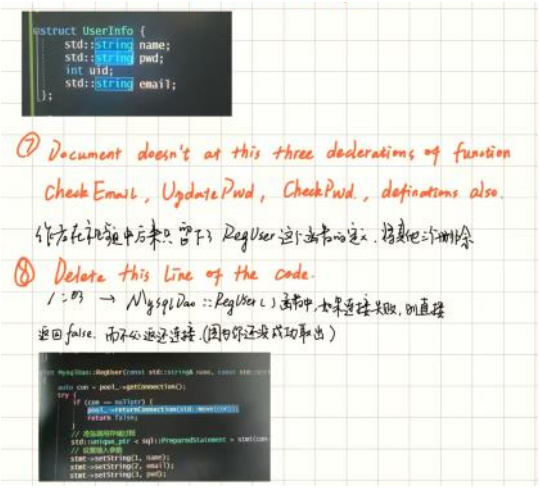


Return Connection() 还有 GetConnection() 两个函数的更新：

CheckConnection 的定义：



关于 CheckThread 涉及到的更改：



关于 CheckThread 涉及到的更改：

```
private:
    void Close();
private:
    std::string m_Url;
    std::string m_User;
    std::string m_Password;
    std::string m_Schema;
    uint32_t m_Poolsize;

    std::atomic<bool> b_Stop;
    std::condition_variable m_Cond;
    std::mutex m_Mutex;
    std::thread m_CheckThread;

    std::queue<std::unique_ptr<MysqlConnection>> m_Connections;
};
```

```
m_CheckThread = std::thread([this]()
    {
        while(!b_Stop)
        {
            CheckConnection();
            std::this_thread::sleep_for(std::chrono::seconds(60));
        }
    }
);
```

》》》》timestamp 代码的分析，以及这里获取的时间戳作用体现在哪里？

》》》》我们通过 MysqlConnection 的构造函数，将时间戳的作为参数，存入 MysqlConnection 类的第二个成员变量：last_oper_time 中，那我们将在哪里、如何使用它呢？

```
MySqlDAO::MySqlDAO(const std::string& url, const std::string& user, const std::string& password
    :m_Url(url), m_User(user), m_Password(password), m_Schema(schema), m_Poolsize(poolsize)
{
    try {
        for (uint32_t i = 0; i < m_Poolsize; i++)
        {
            sql::mysql::MySQL_Driver* driver = sql::mysql::get_mysql_driver_instance();
            std::unique_ptr<sql::Connection> con(driver->connect(m_Url, m_User, m_Password));
            con->setSchema(m_Schema);

            auto currentTime = std::chrono::system_clock::now().time_since_epoch();
            auto timestamp = std::chrono::duration_cast<std::chrono::seconds>(currentTime).count();

            m_Connections.push(std::make_unique<MysqlConnection>(con, timestamp));
        }
```

时间戳的代码分析：

代码分析：	`auto currentTime = std::chrono::system_clock::now().time_since_epoch();` `auto timestamp = std::chrono::duration_cast<std::chrono::seconds>(currentTime).count();`
1. std::chrono::system_clock::now()	std::chrono::system_clock 是 C++11 提供的时间类之一，表示系统时间（通常是 Unix 时间），也就是当前的日期和时间。 调用 system_clock::now() 返回一个 std::chrono::time_point 类型的对象，表示当前的时间点。 • system_clock::now() 返回一个 std::chrono::time_point 对象，它代表了当前的时间（与系统时钟同步）。
2. time_since_epoch()	time_since_epoch() 是 std::chrono::time_point 的成员函数，它返回自纪元（epoch）以来经过的时间量。 对于 std::chrono::system_clock，纪元通常是 Unix 纪元，即 1970 年 1 月 1 日 00:00:00 UTC。 • time_since_epoch() 返回一个 std::chrono::duration 对象，表示从 Unix 纪元开始到当前时间的时间段。
3. std::chrono::duration_cast<std::chrono::seconds>(currentTime)	std::chrono::duration_cast 是用来将一种时间单位的持续时间转换为另一种时间单位的工具。 这里，currentTime 是一个 std::chrono::duration 类型的对象，表示自 Unix 纪元以来经过的时间。你希望将这个时间转换成秒数。 • std::chrono::duration_cast<std::chrono::seconds>(currentTime) 会将 currentTime 转换为秒（即 std::chrono::seconds），并返回一个表示秒数的持续时间对象。
4. count()	count() 是 std::chrono::duration 的成员函数，用来获取其中存储的具体数值（即持续时间的数值）。 count() 返回的是一个整数，表示经过的时间量（在这种情况下是秒数）。 • count() 返回一个 long long 类型的值，表示从 Unix 纪元以来的秒数。
数据的演变：	**1. 获取当前的系统时间点（std::chrono::system_clock::now()）。** **2. 计算从 Unix 纪元（1970 年 1 月 1 日 00:00:00 UTC）到当前时刻经过的时间（time_since_epoch()）。** **3. 将经过的时间转换为秒数（std::chrono::duration_cast<std::chrono::seconds>）。** **4. 获取秒数的具体数值（count()）。**

MysqlConnection 类的设计以及使用：

关于 MysqlConnection 类：	

关于 MysqlPool 中的成员变量：	为了使用 MysqlConnection 中的 last_oper_time 这个成员变量，我们需要做一些更改。
	`std::queue<std::unique_ptr<sql::Connection>> m_Connections;`
	`std::queue<std::unique_ptr<MysqlConnection>> m_Connections;`

```
std::queue<std::unique_ptr<sql::Connection>> m_Connections;
std::queue<std::unique_ptr<MysqlConnection>> m_Connections;
```

我们在 CheckConnection() 函数中使用了 MysqlConnection 中的成员变量：
（具体细节：以 LastOperTime 为例）

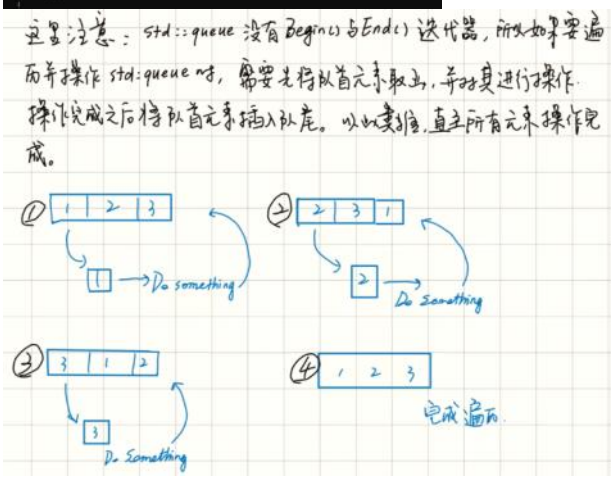
```
void MySqlDAO::CheckConnection()
{
    std::lock_guard<std::mutex> lock(m_Mutex);      // 这里为什么要上锁？会与其他线程发生冲突吗？
    uint32_t poolsize = m_Connections.size();

    auto currentTime = std::chrono::system_clock::now().time_since_epoch();
    auto timestampNow = std::chrono::duration_cast<std::chrono::seconds>(currentTime).count();

    for(uint32_t i = 0; i < poolsize; i++)
    {
        auto conn = std::move(m_Connections.front());
        m_Connections.pop();
        Defer defer([this, &conn]()
            {
                m_Connections.push(std::move(conn));
            }
        );

        // 每5分钟才能检查一次，否则退回，重新开启下一次循环。检查的具体操作是："SELECT 1"
        if(timestampNow - conn->LastOperTime > 300)
        {
            continue;
        }

        try
        {
            std::unique_ptr<sql::Statement> stmt(conn->Connection->createStatement());
            stmt->executeQuery("SELECT 1");             // 执行一个简单的语句，通过简便且高效的方式保持连接活跃
            conn->LastOperTime = timestampNow;
            JC_CORE_INFO("A scheduled query is being executed (to keep the database connection active)...")
            JC_CORE_INFO("Scheduled query success! current time is {}", timestampNow);
        }
        catch(sql::SQLException& ex)
        {
            JC_CORE_ERROR("Error happened when keeping SQL connection alive! (message:{})", ex.what());
```

》》》》check thread 如何作用于程序？有什么意义？

```
    m_CheckThread = std::thread([this]()
        {
            while(!b_Stop)
            {
                CheckConnection();
                std::this_thread::sleep_for(std::chrono::seconds(60));
            }
        }
    );

    m_CheckThread.detach();
}
```

》》》》如何安全的遍历队列？？？ std::queue<std::unique_ptr<sql::Connection>> m_Connections;

```
for(uint32_t i = 0; i < poolsize; i++)
{
    auto conn = std::move(m_Connections.front());
    m_Connections.pop();
    Defer defer([this, &conn]()
        {
            m_Connections.push(std::move(conn));
        }
    );
```



》》关于 Defer 的细节：

```cpp
// Defer 类实现了类似 RAII 的功能。
// 比如在使用时，我们先创建一个 Defer 类型对象，然后在初始化构造函数中填入一个 lambda 或者 std::function<>，
// 一旦声明该 Defer 类型变量的作用域将要终止生命周期，即将要被销毁，该作用域则会自动调用 Defer 的析构函数，
// 而 Defer 的析构函数会运行我们传入的函数。
//
// eg:
//  auto conn = std::move(m_Connections.front());
//  m_Connections.pop();
//  Defer defer([this, &conn]()
//      {
//          m_Connections.push(std::move(conn));
//      }
//  );
//  在这里，一旦存储 defer 的作用域将要被销毁，则会自动调用我们传入的 .push() 函数，实现一些目的。
class Defer
{
public:
    Defer(std::function<void()> func)
        :m_Function(func)
    {
    }

    ~Defer()
    {
        m_Function();
    }
private:
    std::function<void()> m_Function;
};
```

》》》》关于 CheckConnetion() 中的 catch() 语句：为什么在这里检查连接失败之后，则需要重新创建新连接并替换旧连接？

在数据库应用中，尤其是长时间运行的程序中，数据库连接有可能会因为网络问题、数据库服务重启、超时等原因失效。

出现 sql::SQLException 可能是因为连接已经无法使用了，因此必须通过重新建立连接来恢复与数据库的正常通信。

```cpp
catch(sql::SQLException& ex)
{
    JC_CORE_ERROR("Error happened when keeping SQL connection alive! (message:{})", ex.what());

    // 为什么在这里检查连接失败之后，则需要重新创建新连接并替换旧连接？
    sql::mysql::MySQL_Driver* driver = sql::mysql::get_mysql_driver_instance();
    std::unique_ptr<sql::Connection> newConn(driver->connect(m_Url, m_User, m_Password));
    newConn->setSchema(m_Schema);

    // 刷新 conn 中的信息（建立新连接并传入、刷新时间戳）
    conn->Connection.reset(newConn.release());
    conn->LastOperTime = timestampNow;
}
```

》》》》声明：

```cpp
class MysqlDao
{
public:
    MysqlDao();
    ~MysqlDao();
    int RegUser(const std::string& name, const std::string& email, const std::string& pwd);
    bool CheckEmail(const std::string& name, const std::string& email);
    bool UpdatePwd(const std::string& name, const std::string& newpwd);
    bool CheckPwd(const std::string& name, const std::string& pwd, UserInfo& userInfo);
private:
    std::unique_ptr<MySqlPool> pool_;
};
```

》》》》RegUser() 细节：

```cpp
int MySqlDAO::RegUser(const std::string& name, const std::string& password, const std::string& email, uint32_t uid)
{
    std::unique_ptr<MysqlConnection> conn = m_Pool->GetConnection();

    if(conn == nullptr)
    {
        JC_CORE_ERROR("Failed to get the connection from MySql connection pool!");
        return false;
    }
    try
    {
        // 创建一个预处理语句
        std::unique_ptr<sql::PreparedStatement> stmt(conn->Connection->prepareStatement("CALL reg_user(?,?,?,@result)"));
        // 并通过语句为存储过程的参数填充对应值
        stmt->setString(1, name);
        stmt->setString(2, password);
        stmt->setString(3, email);
        // 执行存储过程
        stmt->execute();

        // 由于 prepatedStatement 不直接支持输出参数这样的功能，我们需要使用会话变量或者其他方法来间接获取这些值（这里选择了使用会话变量 @result 来传递输出值）
        // 使用 createStatement 创建普通的 SQL 语句执行器
        std::unique_ptr<sql::Statement> resultStmt(conn->Connection->createStatement());
        // 如果存储过程中设置了会话变量，或者其他方式获取输出参数的值，可以使用会话变量执行查询结果
        std::unique_ptr<sql::ResultSet> resultSet(resultStmt->executeQuery("SELECT @result AS result"));
        if(resultSet->next())
        {
            int resultInt = resultSet->getInt("result");
            JC_CORE_INFO("Result:{} (function{})", resultInt, __FUNCSIG__);

            m_Pool->ReturnConnection(std::move(conn));
            return resultInt;
        }
        m_Pool->ReturnConnection(std::move(conn));
        return -1;
    }
    catch (sql::SQLException& ex)
    {
        m_Pool->ReturnConnection(std::move(conn));
        JC_CORE_ERROR("(SQL exception message:{})", ex.what());
        JC_CORE_ERROR("(SQL error code:{})", ex.getErrorCode());
        JC_CORE_ERROR("(SQL state:{})", ex.getSQLState());
```

》》》》 return 0 和 return -1 的区别?

区别:	return 0：在大多数情况下，它表示程序的成功执行或一个操作的成功完成。如果是从main()函数返回0，它表示程序执行成功没有问题。 return -1：通常作为错误的标志，表示函数或者程序在执行过程中出现了错误或不符合预期的情况。它的含义通常由开发者或系统预先定义。
终止程序的方式:	return 0：通常没有错误发生，程序顺利结束。 return -1：虽然程序也会终止，但它通常伴随着错误消息或某种错误处理机制，以帮助程序调用者或开发者知道出错了。

》》》》一些疑问: **关于（prepareStatement、createStatement、Statement、ResultSet的定义）**

```cpp
// 创建一个预处理语句
std::unique_ptr<sql::PreparedStatement> stmt(conn->Connection->prepareStatement("CALL reg_user(?,?,?,@result)"));
// 并通过语句为存储过程的参数填充对应值
stmt->setString(1, name);
stmt->setString(2, password);
stmt->setString(3, email);
// 执行存储过程
stmt->execute();

// 由于 prepatedStatement 不直接支持输出参数这样的功能，我们需要使用会话变量或者其他方法来间接获取这些值（这里选择了使用会话变量 @result 来传递输出值）
// 使用 createStatement 创建普通的 SQL 语句执行器
std::unique_ptr<sql::Statement> resultStmt(conn->Connection->createStatement());
// 如果存储过程中设置了会话变量，或者其他方式获取输出参数的值，可以使用会话变量执行查询结果
std::unique_ptr<sql::ResultSet> result(resultStmt->executeQuery("SELECT @result AS result"));
```

prepareStatement:	用于创建带有参数的预处理 SQL 语句，主要用于执行有参数的 SQL 查询或更新。
createStatement:	用于创建普通的 SQL 语句执行器，主要用于执行不带参数的 SQL 查询或更新。
sql::Statement:	表示一个 SQL 语句执行器的类型，这个类型的变量能够执行 SQL 语句（如查询、更新等）。
sql::ResultSet:	表示查询的结果集合，包含了从数据库查询返回的数据。

prepareStatement 是什么函数?

prepareStatement 是 MySQL Connector/C++ 提供的一个函数，用于创建一个预处理语句（PreparedStatement）。
这个函数主要用于执行带有参数的 SQL 语句（如 INSERT, SELECT, UPDATE, DELETE 等）。

预处理语句:	预处理语句是一种SQL查询方式，可以用来执行包含参数占位符（如 ?）的SQL语句。 在执行时，实际的参数值会被绑定到这些占位符上。与普通的 SQL 查询不同，预处理语句可以提高执行效率并防止 SQL 注入攻击。
语法:	`unique_ptr<sql::PreparedStatement> stmt(con->prepareStatement("SQL语句"));` • prepareStatement 的参数是一个 SQL 语句，可以包含参数占位符（例如 ?）。 • stmt 是一个指向 PreparedStatement 对象的智能指针。
功能:	prepareStatement 用于创建一个带参数的 SQL 语句对象，之后可以通过 setXXX 方法设置参数值并执行该语句。

例子:	`unique_ptr<sql::PreparedStatement> stmt(con->prepareStatement("SELECT * FROM users WHERE email = ?"));`
	`stmt->setString(1, "example@email.com");`
	在这个例子中，`prepareStatement` 准备了一个查询，查询条件是根据 `email` 字段。我们用 `setString` 方法将 `email` 的值绑定到查询语句中的 `?` 占位符。

createStatement 是什么函数?

createStatement 是 MySQL Connector/C++ 提供的一个函数，用于创建一个普通的 SQL 语句对象（Statement）。
与 PreparedStatement 不同，Statement 用于执行不包含参数的 SQL 语句（如 SELECT, INSERT 等）。createStatement 用于创建一个用于执行 SQL 查询或更新的对象。

语法:	`unique_ptr<sql::Statement> stmt(con->createStatement());` ○ createStatement 不需要任何参数，返回一个 Statement 对象。 ○ stmt 是一个指向 Statement 对象的智能指针。
功能:	createStatement 用于创建一个 SQL 语句执行器，通常用于执行不需要绑定参数的简单 SQL 语句。
例子:	`unique_ptr<sql::Statement> stmt(con->createStatement());` `unique_ptr<sql::ResultSet> res(stmt->executeQuery("SELECT * FROM users"));`

》》》》 这里的 res->next() 有什么作用？？

```
if (res->next()) {
     int result = res->getInt("result");
    cout << "Result: " << result << endl;
    pool_->returnConnection(std::move(con));
    return result;
}
```

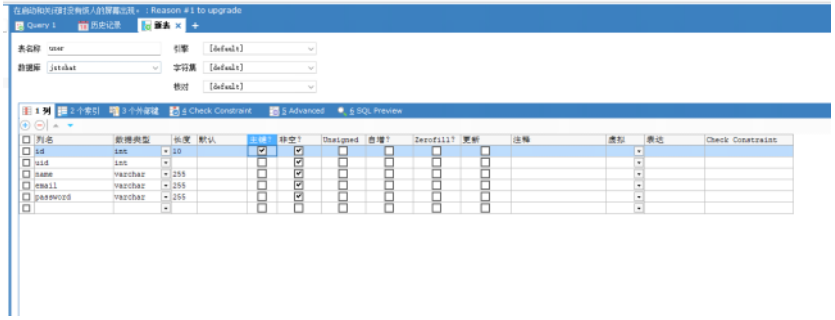
遍历结果集：	res->next() 用于将结果集中的游标（指针）移动到下一个记录/行。
返回布尔值：	它返回一个布尔值，表示是否还有更多的数据行。如果游标成功移动到下一行，next() 返回 true；如果没有更多的行，返回 false。

》》》》 std::cerr

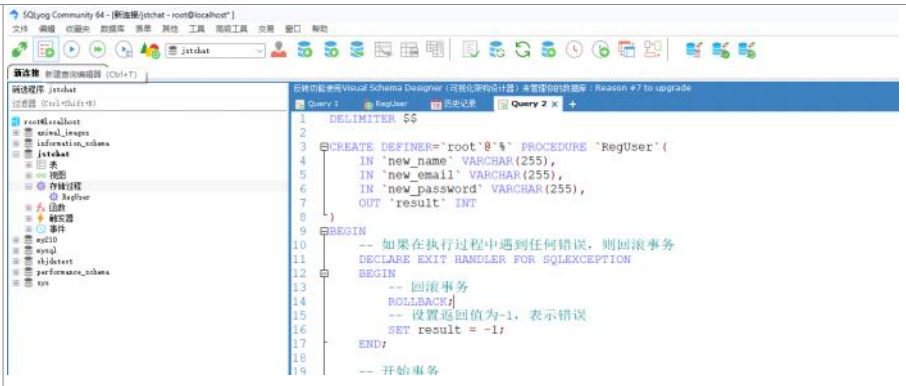
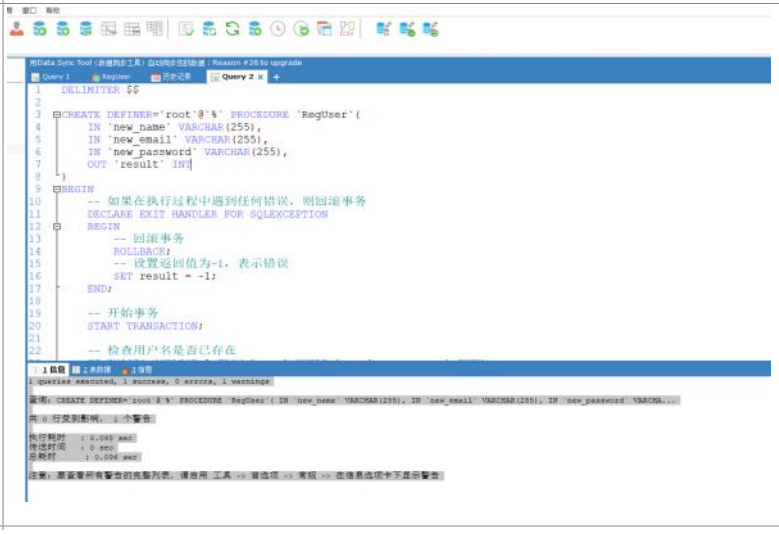
std::cerr 是 C++ 中标准库的一部分，用于输出错误信息。它是一个输出流对象，通常与标准错误流（stderr）相关联，专门用于显示错误信息和诊断消息。

- std::cerr 会向控制台或终端输出数据，和 std::cout 类似，但 std::cerr 输出的是错误信息或警告信息，目的是与正常输出区分开。而且
  输出到 std::cerr 的内容不会被缓冲，这意味着错误信息会立即显示出来，不会像 std::cout 那样先放到缓冲区再输出。

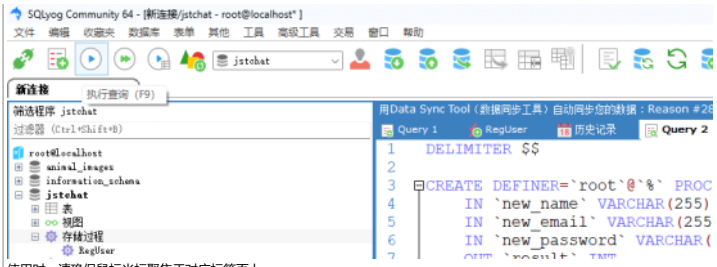
》》》》建表



》》》》创建存储过程：

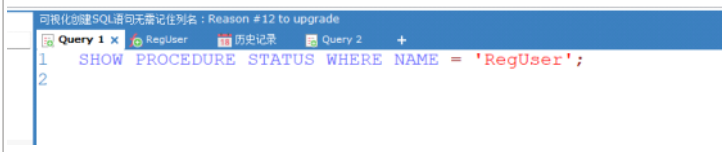
第一步：在 desktop maneger 中创建一个新查询（或者使用快捷键 Ctrl + T）	
第二步：在新建查询中编写存储过程的代码：	

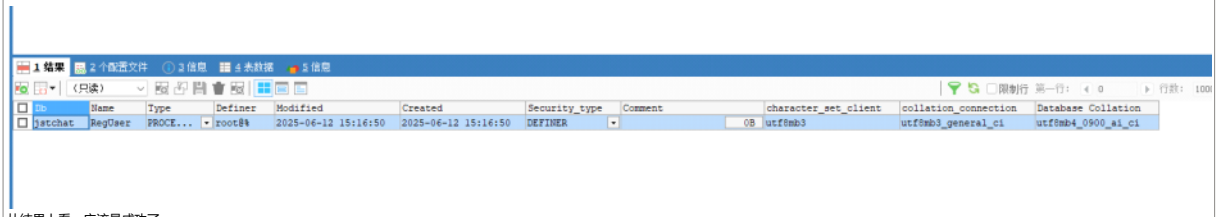
第三步：运行该存储过程（或者使用 F9）



使用时，请确保鼠标光标聚焦于对应标签页上。

第四步：执行完成之后，可以开启一个新查询，执行以查看存储过程是否保存成功？





从结果上看，应该是成功了。

或者你也可以刷新对象浏览器，查看 ”存储过程 “这个选项中是否有刚才存放的存储过程。