JstChat5

2025年6月2日 9:28

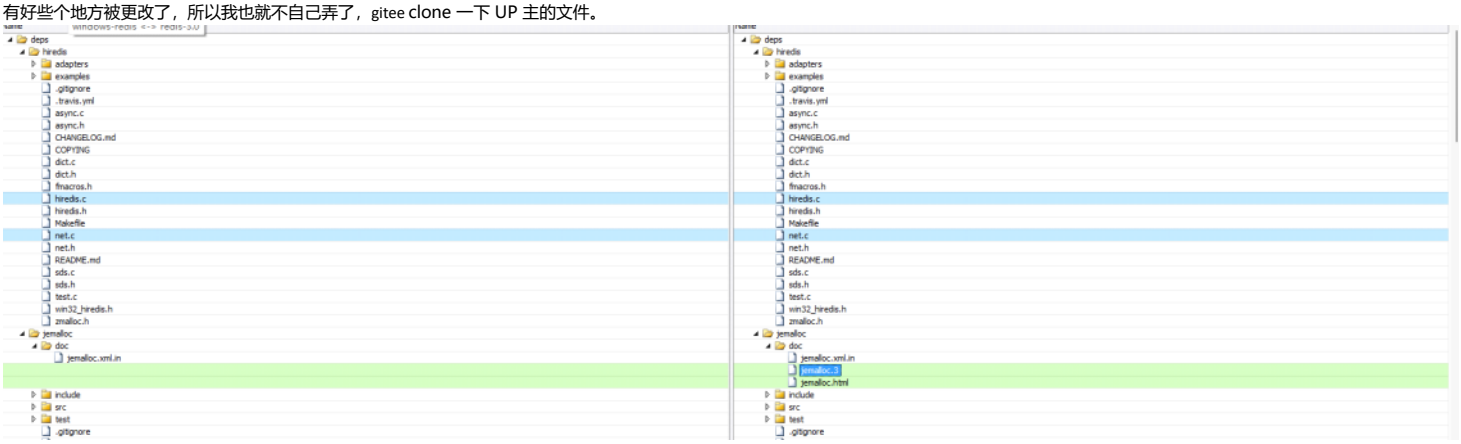
---------- Ep11 Redis 的配置 ----------

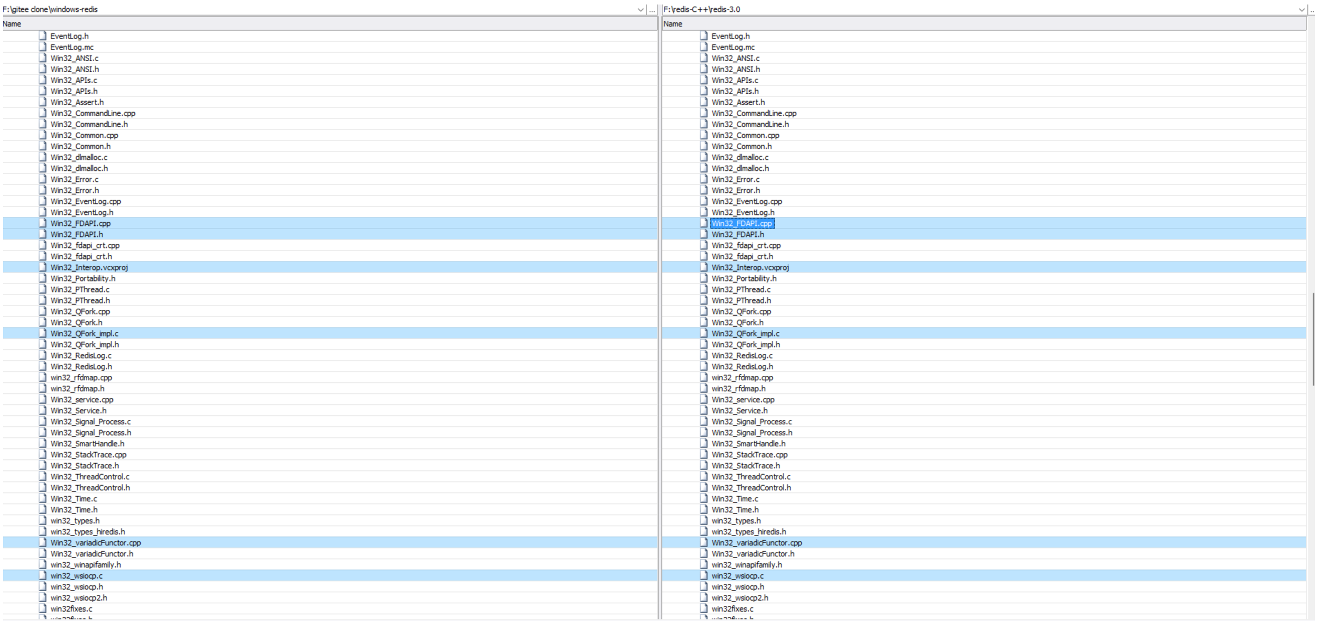
》》》》UP主的 redis 服务好像是搭建在自己私有的腾讯云服务器上: 81.68.86.146，并使用了 81.68.86.146 这个端口（redis 默认端口 6379）
但是 UP 主在启动 redis server 时没有特别标明私有的服务器，而且 redis 的配置文件中也是默认 127.0.0.1 的，所以我在想，如果要是使用 UP 主服务器上的 redis，是不是应该修改配置文件？

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。

```
42
43  # By default, if no "bind" configuration directive is specified, Redis listens
44  # for connections from all the network interfaces available on the server.
45  # It is possible to listen to just one or multiple selected interfaces using
46  # the "bind" configuration directive, followed by one or more IP addresses.
47  #
48  # Examples:
49  #
50  # bind 192.168.1.100 10.0.0.1
51  # bind 127.0.0.1 ::1
52  #
53  # ~~~ WARNING ~~~ If the computer running Redis is directly exposed to the
54  # internet, binding to all the interfaces is dangerous and will expose the
55  # instance to everybody on the internet. So by default we uncomment the
56  # following bind directive, that will force Redis to listen only into
57  # the IPv4 loopback interface address (this means Redis will be able to
58  # accept connections only from clients running into the same computer it
59  # is running).
60  #
61  # IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
62  # JUST COMMENT THE FOLLOWING LINE.
63  # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
64  bind 127.0.0.1
65
66  # Protected mode is a layer of security protection, in order to avoid that
67  # Redis instances left open on the internet are accessed and exploited.
68  #
69  # When protected mode is on and if:
70  #
71  # 1) The server is not binding explicitly to a set of addresses using the
72  #    "bind" directive.
73  # 2) No password is configured.
74  #
75  # The server only accepts connections from clients connecting from the
76  # IPv4 and IPv6 loopback addresses 127.0.0.1 and ::1, and from Unix domain
77  # sockets.
78  #
```

》》》》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。

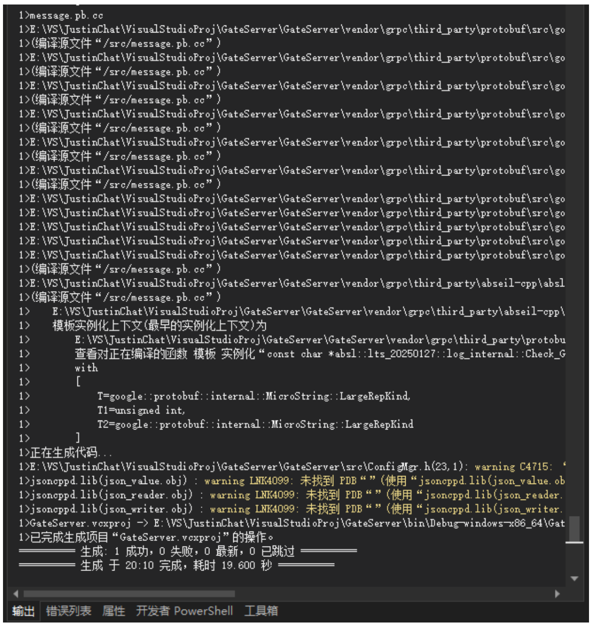


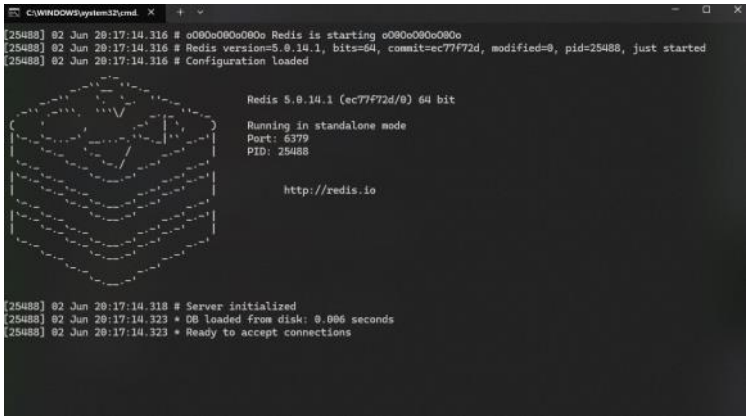
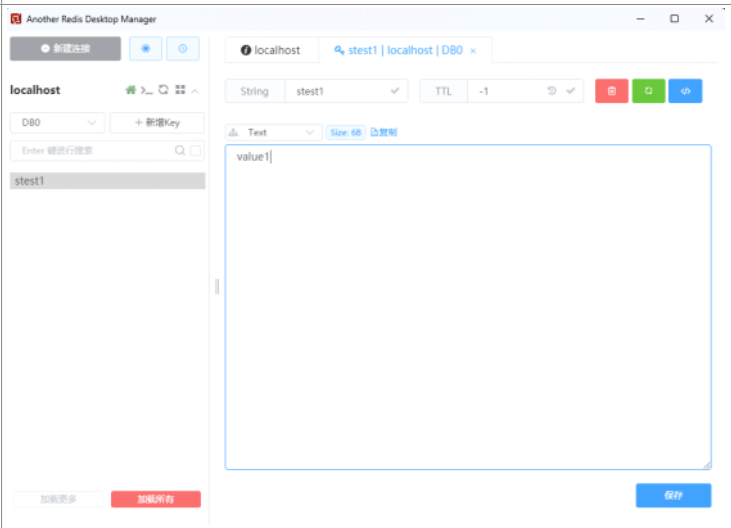
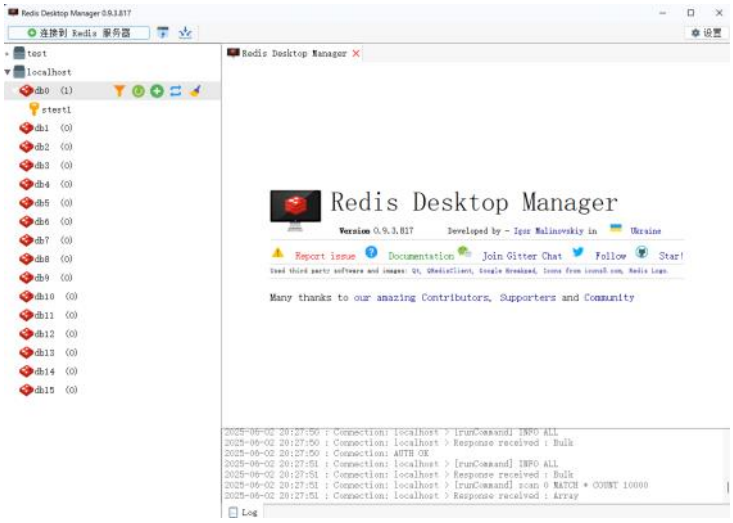


随便找个地方 clone 一下文件：（指令）git clone https://gitee.com/secondtonone1/windows-redis.git
然后将克隆后的文件全选，清除之前下载的 redis 库中的所有文件，然后将 UP 主提供的放置进去。
同时我们打开 .sln 文件，手动重新编译一下，生成 Lib 库。

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)
    {
        std::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

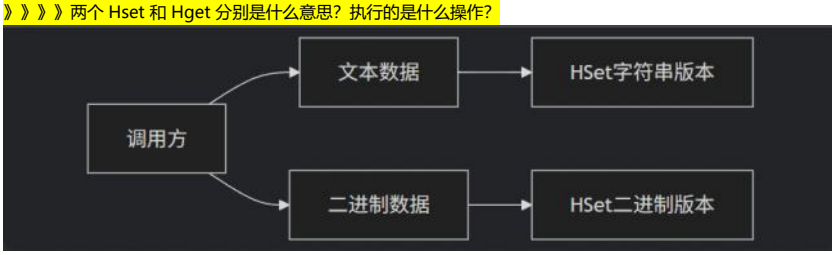
Connect 函数只需要通过 redisConnect 来判断是否连接成功，并适时输出日志

```
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。

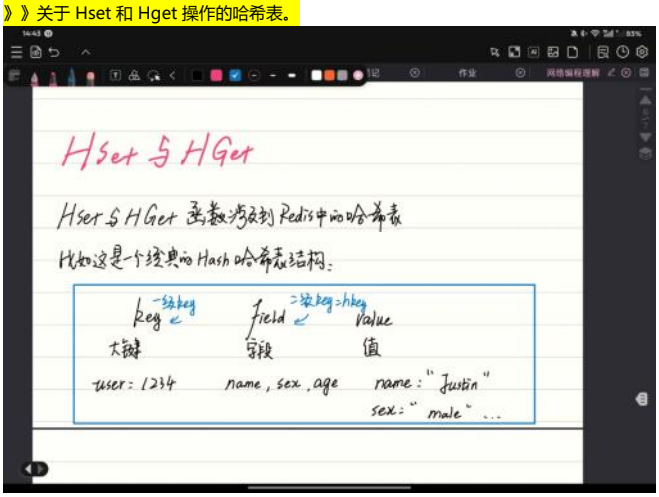
第二种重载：	bool RedisMgr::HSet(const char* key, const char* hkey, const char* hvalue, size_t hvaluelen)
作用：	通过 redisCommandArgv 使用更灵活的方式来发送命令。 这种方式适用于不直接使用 std::string 类型的数据。但是由于参数传递方式不同，适合处理二进制数据，能够指定每个参数的长度。

2. Hget

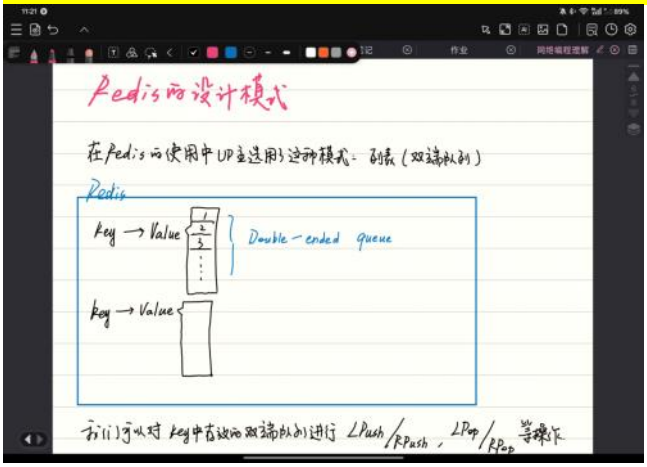
HGet 函数用于从 Redis 中获取哈希表某个字段的值。

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

》》那么这里的 **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

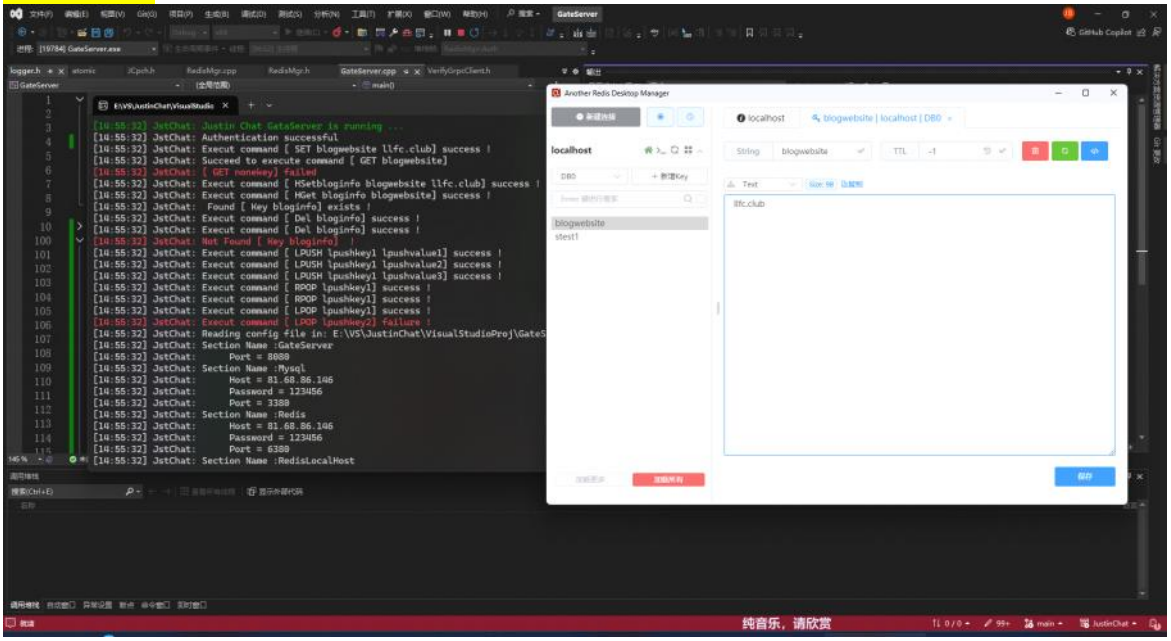
```
bool RedisMgr::Set(const std::string& key, const std::string& value) {
    //执行redis命令行
    m_Reply = (redisReply*)redisCommand(m_Connect, "SET %s %s", key.c_str(), value.c_str());

    //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

```
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

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

    Close();
    while (!m_Connections.empty())
    {
        auto* context = m_Connections.front();
        redisFree(context);                    // 由于这里的

        m_Connections.pop();
    }
}
```

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）

```
bool RedisMgr::LPush(const std::string& key, const std::string& value)
{
    auto* connect = m_Pool->GetConnection();
    if (connect == nullptr)
    {
        JC_CORE_ERROR("Failed to get connection from pool!");
        return false;
    }

    auto reply = (redisReply*)redisCommand(connect, "LPUSH %s %s", key.c_str(), value.c_str());
    if (NULL == reply)
    {
        JC_CORE_ERROR("Execut command [ LPUSH {} {}] failure ! ", key, value);
        freeReplyObject(reply);
        m_Pool->ReturnConnection(connect);
        return false;
    }

    if (reply->type != REDIS_REPLY_INTEGER || reply->integer <= 0)
    {
        JC_CORE_ERROR("Execut command [ LPUSH {} {}] failure ! ", key, value);
        freeReplyObject(reply);
        m_Pool->ReturnConnection(connect);
        return false;
    }

    JC_CORE_TRACE("Execut command [ LPUSH {} {}] success ! ", key, value);
    freeReplyObject(reply);
    m_Pool->ReturnConnection(connect);
    return true;
}
```

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。

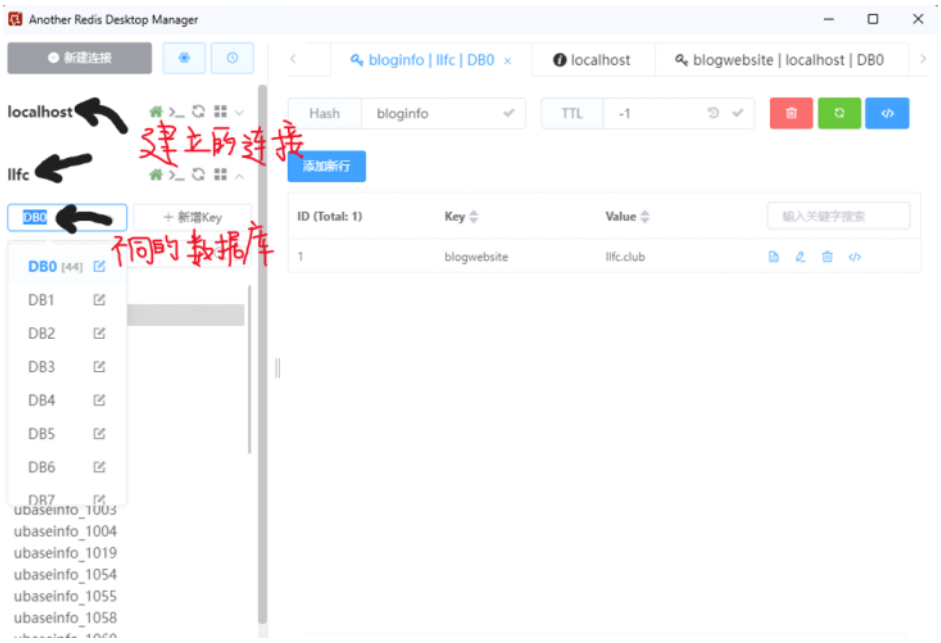
理解：
在 Redis 中，数据并不像关系型数据库（例如 MySQL 或 PostgreSQL）那样被存储在不同的“表”中。Redis 数据库是基于内存的，并且没有传统的表、行等结构。

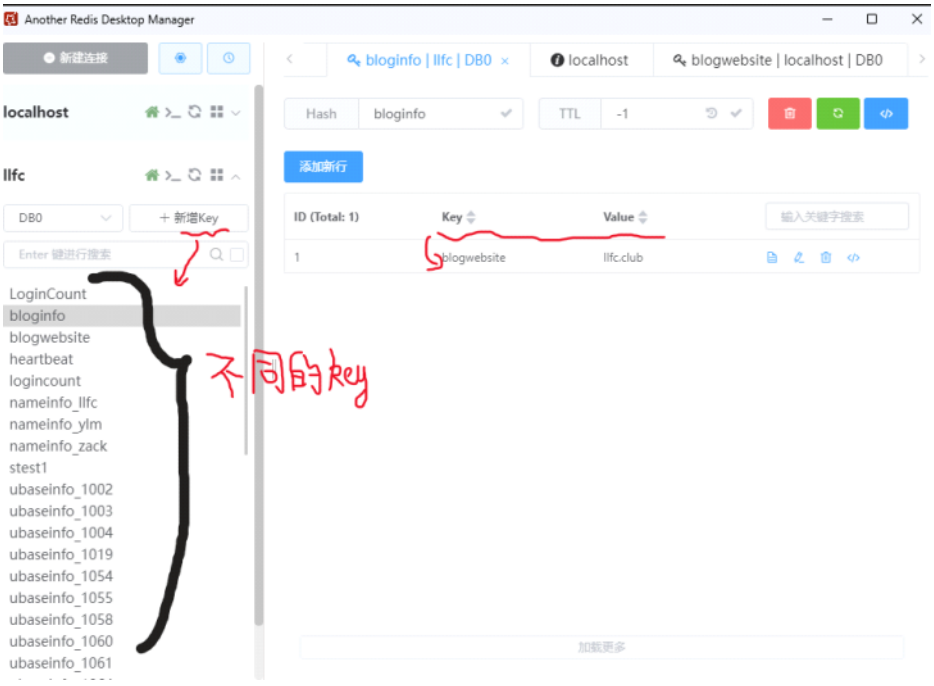
Redis 是一个键值数据库（Key-Value Store），它通过键（key）来访问数据，而不需要显式指定表的概念。
在 Redis 中，你可以将数据存储在默认的数据库中，或者指定不同的数据库编号（0-15，Redis 默认有 16 个数据库，编号从 0 到 15）。因此 Redis 客户端的配置一般不需要显式地指定数据库名称，而是通过数据库编号来选择数据库。

（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：

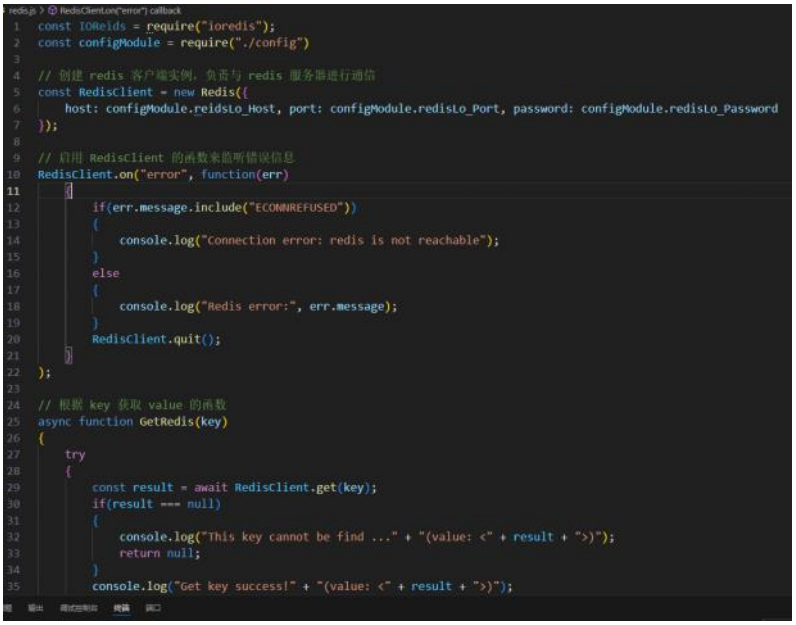
```
// 切换到数据库编号为 1 的数据库
await RedisCli.select(1);
```





》》》》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.reidsLo_Host, port: configModule.redisLo_Port, password: configModule.redisLo_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 + ` => ~）