分布式任务调度与消息派发系统

一. 前言

这篇博客是介绍我在学习C++、Linux系统与网络时所做的一个项目。该项目主要功能代码量在3800行左右，加上辅助的demo与test部分代码的话，总代码量会在4600行左右。

该项目主要还是精简RabbitMQ的核心功能，只支持单虚拟机，所以不包含虚拟机的管理模块。网络部分由陈硕大佬的muduo库支持，使用Protobuf自定义协议，SQLite3轻量数据库提供数据持久化功能，Gtest作为模块以及总体的测试框架。

我写这篇博客的目的也是为了加深印象，并且重温主要模块的构建思想。

二. 项目简介

开发环境：C++17、VSCode、Linux、Ubuntu、gdb、Makefile、Protobuf、muduo 库、SQLite3、Gtest

项目介绍：为解决分布式系统中服务解耦、异步通信和流量削峰的核心需求，基于 C++实现了轻量级消息队列中间件。项目借鉴了 RabbitMQ 的核心理念，通过 Broker 中心代理、灵活的路由机制与数据持久化，构建了高可靠的生产-消费者通信模型，有效提升了系统的可扩展性与稳定性。

项目流程：

1. 基于生产消费者模型，生产消费者作为客户端，分别实现创建消息并投递与获取消息并异步处理。

2. 消息队列作为服务端，主体由 Broker 来管理，其中包含虚拟机、数据库、内存中消息管理、数据持久化等。

3. 生产者发布消息：连接 Broker，声明交换机，发送消息给该交换机，并声明一个路由键。

4. 交换机路由消息：根据自身类型和消息路由键检查绑定规则，根据匹配结果投递到一个或多个匹配队列中。

5. 队列暂存消息：消息在对应队列中等待被消费，如果队列和消息都设置为持久化，则消息会被保存到磁盘。

6. 消费者接受处理：消费者连接 Broker 并订阅一个队列，消息根据推送模式交给消费者处理，并发送确认。

三. 第三方库介绍

Protobuf

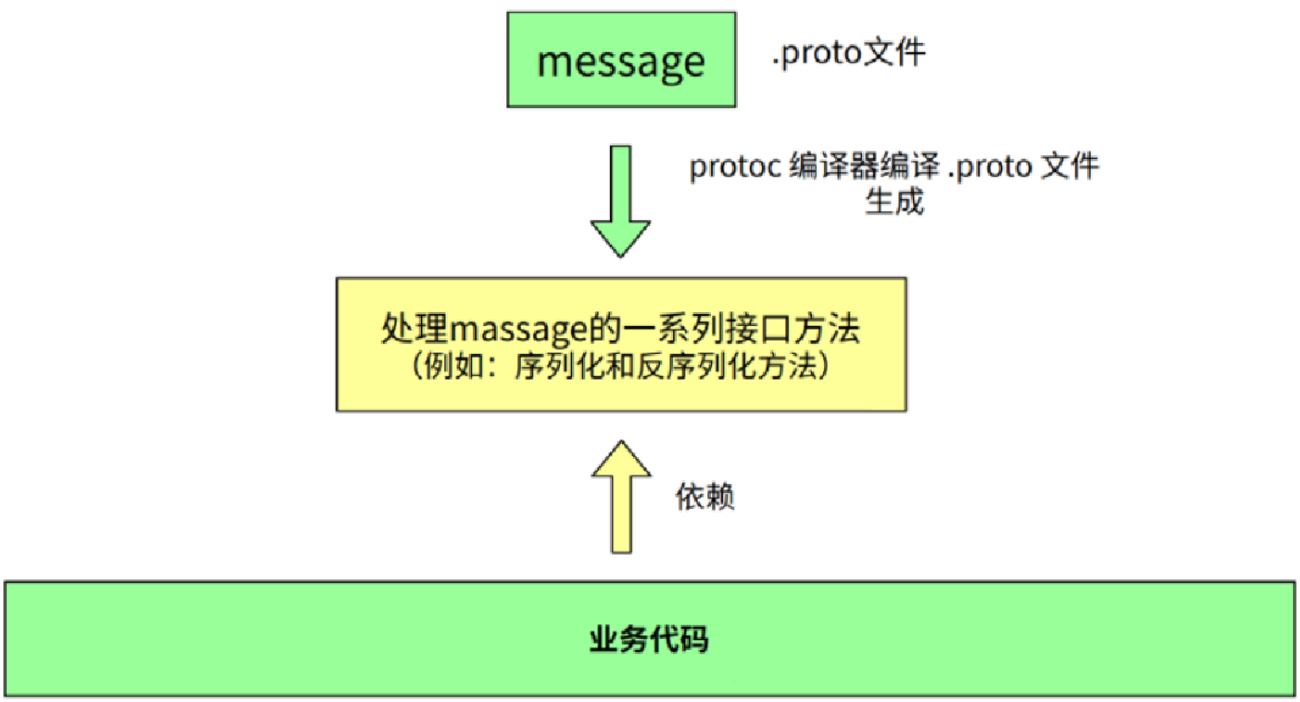
ProtoBuf（全称 Protocol Buffer）是数据结构序列化和反序列化框架，它具有以下特点：

· **语言无关、平台无关：**即 ProtoBuf 支持 Java、C++、Python 等多种语言，支持多个平台。

· **高效：**即比 XML 更小、更快、更为简单。

· **扩展性、兼容性好：**你可以更新数据结构，而不影响和破坏原有的旧程序。

Protobuf 使用流程介绍



- 编写 .proto 文件，目的是为了定义结构对象（message）及属性内容。
- 使用 protoc 编译器编译 .proto 文件，生成一系列接口代码，存放在新生成头文件和源文件中。
- 依赖生成的接口，将编译生成的头文件包含进我们的代码中，实现对 .proto 文件中定义的字段进行设置和获取，和对 message 对象进行序列化和反序列化。

ProtoBuf 基本功能

指定 proto3 语法

Protocol Buffers 语言版本 3，简称 proto3，是 .proto 文件最新的语法版本。proto3 简化了 Protocol Buffers 语言，既易于使用，又可以在更广泛的编程语言中使用。它允许你使用 Java，C++，Python 等多种语言生成 protocol buffer 代码。在 .proto 文件中，要使用 syntax = "proto3"; 来指定文件语法为 proto3，并且必须写在除去注释内容的第一行。 如果没有指定，编译器会使用 proto2 语法。

可以为文件指定 proto3 语法，内容如下：

```
syntax = "proto3";
```

package 声明符

package 是一个可选的声明符，能表示 .proto 文件的命名空间，在项目中要有唯一性。它的作用是为了避免我们定义的消息出现冲突。

```
syntax = "proto3";
package contacts;
```

定义消息（message）

消息（message）: 要定义的结构化对象，我们可以给这个结构化对象中定义其对应的属性内容。在网络传输中，我们需要为传输双方定制协议。定制协议说白了就是定义结构体或者结构化数据，比如，tcp，udp 报文就是结构化的。再比如将数据持久化存储到数据库时，会将一系列元数据统一用对象组织起来，再进行存储。ProtoBuf 就是以 message 的方式来支持我们定制协议字段，后期帮助我们形成类和方法来使用。

```
syntax = "proto3";
package contacts;
message 消息类型名
{

```

> }
>
> 消息类型命名规范：使用驼峰命名法，首字母大写

定义消息字段

在 message 中我们可以定义其属性字段，字段定义格式为：**字段类型 字段名 = 字段唯一编号；**

· 字段名称命名规范：全小写字母，多个字母之间用 _ 连接。

· 字段类型分为：标量数据类型 和 特殊类型（包括枚举、其他消息类型等）。

· 字段唯一编号：用来标识字段，一旦开始使用就不能够再改变。

该表格展示了定义于消息体中的**标量数据类型**，以及编译 .proto 文件之后自动生成的类中与之对应的字段类型。在这里展示了与 C++ 语言对应的类型。

.proto Type	Notes	C++ Type
double		double
float		float
int32	使用变长编码 [1]。负数的编码效率较低 ——若字段可能为负值，应使用 sint32 代替。	int32
int64	使用变长编码 [1]。负数的编码效率较低 ——若字段可能为负值，应使用 sint64 代替。	int64
uint32	使用变长编码 [1]。	uint32
uint64	使用变长编码 [1]。	uint64
sint32	使用变长编码 [1]。符号整型。负值的编码效率高于常规的 int32 类型。	int32
sint64	使用变长编码 [1]。符号整型。负值的编 码效率高于常规的 int64 类型。	int64
fixed32	定长 4 字节。若值常大于 2^28 则会比 uint32 更高效。	uint32

fixed64	定长 8 字节。若值常大于 2^56 则会比 uint64 更高效。	uint64
sfixed32	定长 4 字节。	int32
sfixed64	定长 8 字节。	int64
bool		bool
string	包含 UTF-8 和 ASCII 编码的字符串，长度不能超过 2^32。	string
bytes	可包含任意字节序列但长度不能超过 2^32。	string

> [1] 变长编码是指：经过 protobuf 编码后，原本 4 字节或 8 字节的数可能会被变为其他字节数。

· 新增姓名、年龄字段：

```
syntax = "proto3";
package contacts;
message PeopleInfo
{
  string name = 1;
  int32 age = 2;
}
```

注：这里还要特别讲解一下**字段唯一编号**的范围：

1 ~ 536,870,911 (2^29 - 1) ，其中 19000 ~ 19999 不可用。

19000 ~ 19999 不可用是因为：在 Protobuf 协议的实现中，对这些数进行了预留。如果非在.proto 文件中使用这些预留标识号，例如将 name 字段的编号设置为 19000，编译时就会报警：

```
// 消息中定义了如下编号，代码会告警：
// Field numbers 19,000 through 19,999 are reserved for the protobuf implementation
string name = 19000;
```

值得一提的是，范围为 1 ~ 15 的字段编号需要一个字节进行编码， 16 ~ 2047 内的数字需要两个字节进行编码。编码后的字节不仅只包含了编号，还包含了字段类型。所以 **1 ~ 15 要用来标记出现非常频繁的字段，要为将来有可能添加的、频繁出现的字段预留一些出来**。

编译 .proto 文件

编译命令行格式为：

> **protoc [--proto_path=IMPORT_PATH] --cpp_out=DST_DIR**
>
> **path/to/file.proto**
>
> **protoc** 是 Protocol Buffer 提供的命令行编译工具。
>
> **--proto_path** 指定 被编译的.proto 文件所在目录，可多次指定。可简写成 -I IMPORT_PATH 。如不指 定该参数，则在当前目录进行搜索。当某个.proto 文件import 其他 .proto 文件时， 或需要编译的 .proto 文件不在当前目录下，这时就要 用-I 来指定搜索目录。
>
> **--cpp_out=** 指编译后的文件为 C++ 文件。
>
> **OUT_DIR** 编译后生成文件的目标路径。
>
> **path/to/file.proto** 要编译的.proto 文件。

编译 contacts.proto 文件命令如下：

```
代码块
1   protoc --cpp_out=. contacts.proto
```

编译 contacts.proto 文件后，会生成所选择语言的代码，我们选择的是 C++，所以编 译后生成了两个文件：contacts.pb.h contacts.pb.cc。

对于编译生成的 C++ 代码，包含了以下内容：

- 对于每个 message ，都会生成一个对应的消息类。
- 在消息类中，编译器为每个字段提供了获取和设置方法，以及一下其他能够操作字段的方法。
- 编辑器会针对于每个 .proto 文件生成.h 和 .cc 文件，分别用来存放类的声明与类的实现。

序列化与反序列化的使用

- 创建一个测试文件 info.cc，方法中我们实现：
  - 对一个联系人的信息使用 PB 进行序列化，并将序列化结果打印出来
  - 对序列化后的内容使用 PB 进行反序列，解析出联系人信息并打印出来

```
代码块
1   #include <iostream> #include "contacts.pb.h" // 引入编译生成的头文件using
    namespace std;
```

```
2
3   int main()
4   {
5       string people_str;
6       // 序列化
7       {
8           // .proto文件声明的package，通过protoc编译后，会为编译生成的C++代码声明同名的命名空间// 其范围是在.proto 文件中定义的内容
9           contacts::PeopleInfo people;
10          people.set_age(20);
11          people.set_name("张珊");
12          // 调用序列化方法，将序列化后的二进制序列存入 string 中if (!people.SerializeToString(&people_str))
13          {
14              cout << "序列化联系人失败." << endl;
15          }
16
17          // 打印序列化结果
18          cout << "序列化后的 people_str: " << people_str.size() << endl;
19      }
20
21      // 反序列化
22      {
23          contacts::PeopleInfo people;
24          // 调用反序列化方法，读取 string 中存放的二进制序列，并反序列化出对象if (!people.ParseFromString(people_str))
25          {
26              cout << "反序列化出联系人失败." << endl;
27          }
28
29          // 打印结果
30          cout << "Parse age: " << people.age() << endl;
31          cout << "Parse name: " << people.name() << endl;
32      }
33
34      return 0;
35  }
```

· 代码书写完成后，编译 info.cc，生成可执行程序

代码块

```
1   g++ info.cc contacts.pb.cc -o info -std=c++11 -lprotobuf
```

- -lprotobuf：链接 protobuf 库文件
- -std=c++11：支持 C++11

· 执行可执行程序，可以看见 people 经过序列化和反序列化后的结果

代码块

```
序列化后的 people_str: 10
Parse age: 20
Parse name: 张珊
```

由于 ProtoBuf 是把联系人对象序列化成了二进制序列，这里用 string 来作为接收二进制序列的容器。所以在终端打印的时候会有换行等一些乱码显示。另外相对于 xml 和JSON 来说，因为 PB 被编码成二进制，破解成本增大，ProtoBuf 编码是相对安全的。

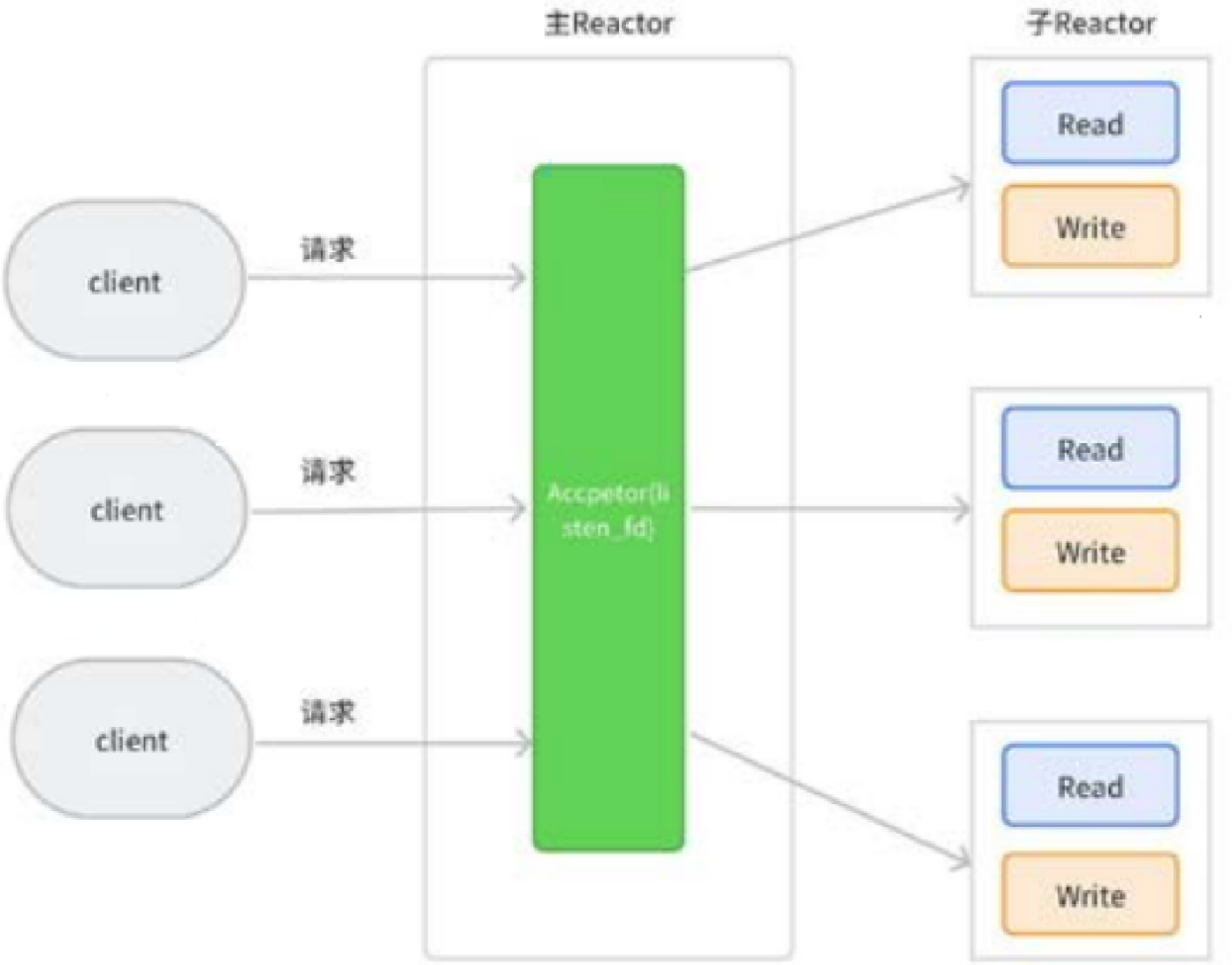
Muduo

Muduo 库是什么

Muduo 由陈硕大佬开发，是一个基于**非阻塞 IO** 和**事件驱动**的 C++高并发 **TCP 网络编程库**。它是一款基于**主从 Reactor** 模型的网络库，其使用的线程模型是 **one loop per thread,** 所谓 **one loop per thread** 指的是：

· 一个线程只能有一个事件循环（EventLoop）， 用于响应计时器和 IO 事件

· 一个文件描述符只能由一个线程进行读写，换句话说就是一个 TCP 连接必须归属于某个 EventLoop 管理



Muduo 库常见接口介绍

muduo::net::TcpServer 类基础介绍

代码块

```
typedef std::shared_ptr<TcpConnection> TcpConnectionPtr;
typedef std::function<void (const TcpConnectionPtr&)>
ConnectionCallback;
typedef std::function<void (const TcpConnectionPtr&,
 Buffer*,
 Timestamp)> MessageCallback;

class InetAddress : public muduo::copyable
{
public:
    InetAddress(StringArg ip, uint16_t port, bool ipv6 = false);
};
```

```
13
14  class TcpServer : noncopyable
15  {
16  public:
17       enum Option
18       {
19           kNoReusePort,
20           kReusePort,
21       };
22       TcpServer(EventLoop* loop,
23                 const InetAddress& listenAddr,
24                 const string& nameArg,
25                 Option option = kNoReusePort);
26       void setThreadNum(int numThreads);
27       void start();
28       /// 当一个新连接建立成功的时候被调用void setConnectionCallback(const
    ConnectionCallback& cb){ connectionCallback_ = cb; }
29       /// 消息的业务处理回调函数---这是收到新连接消息的时候被调用的函数void
    setMessageCallback(const MessageCallback& cb){ messageCallback_ = cb; }
30  };
```

muduo::net::EventLoop 类基础介绍

代码块

```
1   class EventLoop : noncopyable
2   {
3   public:
4       /// Loops forever./// Must be called in the same thread as creation of the
    object.void loop();
5       /// Quits loop./// This is not 100% thread safe, if you call through a raw
    pointer,/// better to call through shared_ptr<EventLoop> for 100% safety.void
    quit();
6       TimerId runAt(Timestamp time, TimerCallback cb);
7       /// Runs callback after @c delay seconds./// Safe to call from other
    threads.TimerId runAfter(double delay, TimerCallback cb);
8       /// Runs callback every @c interval seconds./// Safe to call from other
    threads.TimerId runEvery(double interval, TimerCallback cb);
9       /// Cancels the timer./// Safe to call from other threads.void
    cancel(TimerId timerId);
10
11  private:
12      std::atomic<bool> quit_;
13      std::unique_ptr<Poller> poller_;
```

```
    mutable MutexLock mutex_;
    std::vector<Functor> pendingFunctors_ GUARDED_BY(mutex_);
};
```

muduo::net::TcpConnection 类基础介绍

代码块

```
class TcpConnection : noncopyable,
                      public
std::enable_shared_from_this<TcpConnection>
{
public:
    /// Constructs a TcpConnection with a connected sockfd////// User should not create this object.TcpConnection(EventLoop* loop,
    const string& name,
    int sockfd,
    const InetAddress& localAddr,
    const InetAddress& peerAddr);
    bool connected() const { return state_ == kConnected; }
    bool disconnected() const { return state_ == kDisconnected; }


    void send(string&& message); // C++11void send(const void* message, int len);
    void send(const StringPiece& message);
    // void send(Buffer&& message); // C++11void send(Buffer* message); // this one will swap datavoid shutdown(); // NOT thread safe, no simultaneous callingvoid setContext(const boost::any& context){ context_ = context; }
    const boost::any& getContext() const{ return context_; }
    boost::any* getMutableContext(){ return &context_; }
    void setConnectionCallback(const ConnectionCallback& cb){ connectionCallback_ = cb; }
    void setMessageCallback(const MessageCallback& cb){ messageCallback_ = cb; }

private:
    enum StateE { kDisconnected, kConnecting, kConnected, kDisconnecting };
    EventLoop* loop_;
    ConnectionCallback connectionCallback_;
    MessageCallback messageCallback_;
    WriteCompleteCallback writeCompleteCallback_;
    boost::any context_;
};
```

muduo::net::TcpClient 类基础介绍

代码块

```
class TcpClient : noncopyable
{
public:
    // TcpClient(EventLoop* loop);// TcpClient(EventLoop* loop, const string& host, uint16_t port);TcpClient(EventLoop* loop,
    const InetAddress& serverAddr,
    const string& nameArg);

    ~TcpClient(); // force out-line dtor, for std::unique_ptr members.void connect();//连接服务器void disconnect();//关闭连接void stop();
    //获取客户端对应的通信连接 Connection 对象的接口，发起 connect 后，有可能还没有连接建立成功TcpConnectionPtr connection() const{
        MutexLockGuard lock(mutex_);
        return connection_;
    }
    /// 连接服务器成功时的回调函数void setConnectionCallback(ConnectionCallback cb){ connectionCallback_ = std::move(cb); }
    /// 收到服务器发送的消息时的回调函数void setMessageCallback(MessageCallback cb){ messageCallback_ = std::move(cb); }
private:
    EventLoop* loop_;
    ConnectionCallback connectionCallback_;
    MessageCallback messageCallback_;
    WriteCompleteCallback writeCompleteCallback_;
    TcpConnectionPtr connection_ GUARDED_BY(mutex_);
};

/*
需要注意的是，因为 muduo 库不管是服务端还是客户端都是异步操作，
对于客户端来说如果我们在连接还没有完全建立成功的时候发送数据，这是不被允
许的。
因此我们可以使用内置的 CountDownLatch 类进行同步控制
*/class CountDownLatch : noncopyable
{
public:
    explicit CountDownLatch(int count);
    void wait(){
        MutexLockGuard lock(mutex_);
        while (count_ > 0)
        {
```

```
            condition_.wait();
        }
    }

    void countDown(){
        MutexLockGuard lock(mutex_);
        --count_;
        if (count_ == 0)
        {
            condition_.notifyAll();
        }
    }
    int getCount() const;

private:
    mutable MutexLock mutex_;
    Condition condition_ GUARDED_BY(mutex_);
    int count_ GUARDED_BY(mutex_);
};
```

muduo::net::Buffer 类基础介绍

代码块

```
class Buffer : public muduo::copyable
{
public:
    static const size_t kCheapPrepend = 8;
    static const size_t kInitialSize = 1024;
    explicit Buffer(size_t initialSize = kInitialSize)
        : buffer_(kCheapPrepend + initialSize),
          readerIndex_(kCheapPrepend),
          writerIndex_(kCheapPrepend);void swap(Buffer& rhs)size_t readableBytes() consts
ize_t writableBytes() constconst char* peek() constconst char* findEOL() constconst char* findEOL(const char* start) constvoid retrieve(size_t len)void retrieveInt64()void retrieveInt32()void retrieveInt16()void retrieveInt8()
    string retrieveAllAsString()
    string retrieveAsString(size_t len)void append(const StringPiece& str)void append(const char* /*restrict*/ data, size_t len)void append(const void* /*restrict*/ data, size_t len)char* beginWrite()const char* beginWrite() constvoid hasWritten(size_t len)void appendInt64(int64_t x)void appendInt32(int32_t x)void appendInt16(int16_t x)void appendInt8(int8_t x)int64_t readInt64()int32_t readInt32()int16_t readInt16()int8_t
```

```
    readInt8()int64_t peekInt64() constint32_t peekInt32() constint16_t
    peekInt16() constint8_t peekInt8() constvoid prependInt64(int64_t x)void
    prependInt32(int32_t x)void prependInt16(int16_t x)void prependInt8(int8_t
    x)void prepend(const void* /*restrict*/ data, size_t len)private:
12      std::vector<char> buffer_;size_t readerIndex_;
13      size_t writerIndex_;
14      static const char kCRLF[];
15  };
```

Muduo 库快速上手

使用 Muduo 网络库来实现一个简单英译汉服务器和客户端快速上手 Muduo 库。英译汉 TCP 服务器。

客户端：

代码块

```
1   #include "include/muduo/net/TcpClient.h"#include
    "include/muduo/net/EventLoopThread.h"#include
    "include/muduo/net/TcpConnection.h"#include
    "include/muduo/base/CountDownLatch.h"#include <iostream>#include
    <string>#include <functional>class TranslateClient
2   {
3   public:
4       TranslateClient(const std::string& ip, uint16_t port)
5           :_latch(1), _client(_loopthread.startLoop(),
    muduo::net::InetAddress(ip, port), "TranslateClient")
6       {
7
    _client.setConnectionCallback(std::bind(&TranslateClient::onConnection, this,
    std::placeholders::_1));
8           _client.setMessageCallback(std::bind(&TranslateClient::onMessage,
    this,
9           std::placeholders::_1, std::placeholders::_2, std::placeholders::_3));
10
11      }
12
13      // 连接服务器 -- 需要等待连接建立成功之后才返回void Connect()
14      {
15          _client.connect();
16          _latch.wait(); // 阻塞等待直到连接建立成功
17      }
18
```

```
19      bool Send(const std::string& message){
20          if(_conn->connected())
21          {
22              _conn->send(message);
23              return true;
24          }
25
26          return false;
27      }
28
29
30  private:
31      // 连接建立成功时的回调函数，连接建立成功后，唤醒连接的阻塞void
    onConnection(const muduo::net::TcpConnectionPtr& conn){
32          if(conn->connected())
33          {
34              _latch.countDown(); // 唤醒主线程中的阻塞
35              _conn = conn;
36          }
37
38          else
39          {
40              // 连接关闭
41              _conn.reset();
42          }
43      }
44
45      // 收到消息时的回调函数void onMessage(const muduo::net::TcpConnectionPtr&
    conn, muduo::net::Buffer* buf, muduo::Timestamp){
46          std::cout << buf->retrieveAllAsString() << std::endl;
47      }
48
49
50  private:
51      muduo::CountDownLatch _latch;
52      muduo::net::EventLoopThread _loopthread; // 实例化后会自动启动
53      muduo::net::TcpClient _client;
54      muduo::net::TcpConnectionPtr _conn;
55  };
56
57  int main(){
58      TranslateClient client("127.0.0.1", 8085);
59      client.Connect();
60
61      while(true)
62      {
63          std::string buf;
```

```
        std::cin >> buf;
        client.Send(buf);
    }

    return 0;
}
```

服务端：

代码块

```
#include "include/muduo/net/TcpServer.h"#include
"include/muduo/net/EventLoop.h"#include
"include/muduo/net/TcpConnection.h"#include <iostream>#include <string>#include
 <functional>#include <unordered_map>class TranslateServer
{
public:
    TranslateServer(uint16_t port)
        :_server(&_baseloop, muduo::net::InetAddress("0.0.0.0", port),
         "TranslateServer", muduo::net::TcpServer::kReusePort)
        {

_server.setConnectionCallback(std::bind(&TranslateServer::onConnection, this,
std::placeholders::_1));
            _server.setMessageCallback(std::bind(&TranslateServer::onMessage,
this,
            std::placeholders::_1, std::placeholders::_2,
std::placeholders::_3));
        }

    // 启动服务器void Start(){
        _server.start(); // 开始监听
        _baseloop.loop(); // 开始事件监控，这是一个死循环阻塞接口
    }

private:
    // onConnection，应该是在一个连接建立成功和关闭时被调用void onConnection(const
muduo::net::TcpConnectionPtr& conn){
        // 新连接建立成功时的回调函数if(conn->connected() == true)
        {
            std::cout << "新连接建立成功" << std::endl;
        }

        else
```

```
        {
            std::cout << "新连接关闭" <<std::endl;
        }
    }

    std::string Translate(const std::string& str){
        static std::unordered_map<std::string, std::string> dict_map = {
            {"hello", "你好"},
            {"cat", "哈基米"},
            {"曼波", "难买绿豆"}
        };

        auto it = dict_map.find(str);
        if(it == dict_map.end()) return "不再曼波";
        std::string ret =  "翻译结果: " + dict_map[str];
        return ret;
    }

    void onMessage(const muduo::net::TcpConnectionPtr& conn,
muduo::net::Buffer* buf, muduo::Timestamp){
        // 通信连接收到请求时的回调函数// 1. 从buf中把请求的数据取出来
        std::string str = buf->retrieveAllAsString();

        // 2. 调用translate接口进行翻译
        std::string resp = Translate(str);

        // 3. 对客户端进行响应
        conn->send(resp);
    }

private:
    // _baseloop是epoll的事件监控，会进行描述符的事件监控，出发事件后进行IO操作
    muduo::net::EventLoop _baseloop;

    // 这个server对象主要用于设置回调函数，用于告诉服务器收到什么请求该如何处理
    muduo::net::TcpServer _server;
};

int main(){
    TranslateServer server(8085);
    server.Start();

    return 0;
}
```

基于 muduo 库函数实现 protobuf 协议的通信

1. 先定义具体的业务请求类型

代码块

```
syntax = "proto3";

package mq;

enum ExchangeType
{
    UNKNOWTYPE = 0;
    DIRECT = 1;
    FANOUT = 2;
    TOPIC = 3;
};

enum DeliveryMode
{
    UNDURABLE = 0;
    DURABLE = 1;
};

message BasicProperties
{
    string id = 1;
    DeliveryMode delivery_mode = 2;
    string routing_key = 3;
};

message Message
{
    message Payload
    {
        BasicProperties properties = 1;
        string body = 2;
        string valid = 3;
    };

    Payload payload = 1;
    uint32 offset = 2;
    uint32 length = 3;
};
```

2. 实现服务端提供的服务

在实现具体服务前， 先介绍一下 muduo 库中内部实现的关于简单的基于 protobuf 的接口类 ProtobufCodec 类是 muduo 库中对于 protobuf 协议的处理类，其内部实现了 onMessage 回调接口，对于接收到的数据进行基于 protobuf 协议的请求处理，然后将解析出的信息，存放到对应请求的 protobuf 请求类对象中，然后最终调用设置进去的消息处理回调函数进行对应请求的处理。

代码块

```
/*muduo-master/examples/protobuf/codec.h*/typedef
std::shared_ptr<google::protobuf::Message> MessagePtr;
//// FIXME: merge with RpcCodec//class ProtobufCodec : muduo::noncopyable
{
public:
    enum ErrorCode
    {
        kNoError = 0,
        kInvalidLength,
        kCheckSumError,
        kInvalidNameLen,
        kUnknownMessageType,
        kParseError,
    };
typedef std::function<void (const muduo::net::TcpConnectionPtr&,
 const MessagePtr&,
 muduo::Timestamp)>
    ProtobufMessageCallback;
    // 这里的messageCb是针对 protobuf 请求进行处理的函数，它声明在dispatcher.h 中的
  ProtobufDispatcher 类explicit ProtobufCodec(const ProtobufMessageCallback&
messageCb)
     : messageCallback_(messageCb),//这就是设置的请求处理回调函数

 errorCallback_(defaultErrorCallback)
 {
 }

    //它的功能就是接收消息，进行解析，得到了proto中定义的请求后调用设置的
messageCallback_进行处理void onMessage(const muduo::net::TcpConnectionPtr& conn,
    muduo::net::Buffer* buf,
    muduo::Timestamp receiveTime);
    //通过 conn 对象发送响应的接口void send(const muduo::net::TcpConnectionPtr&
conn,
    const google::protobuf::Message& message){
        // FIXME: serialize to TcpConnection::outputBuffer()
        muduo::net::Buffer buf;
        fillEmptyBuffer(&buf, message);
```

```
33          conn->send(&buf);
34      }
35      static const muduo::string& errorCodeToString(ErrorCode errorCode);
36      static void fillEmptyBuffer(muduo::net::Buffer* buf,
37      const google::protobuf::Message& message);
38      static google::protobuf::Message* createMessage(const
39      std::string& type_name);
40      static MessagePtr parse(const char* buf, int len, ErrorCode* errorCode);
41
42  private:
43      static void defaultErrorCallback(const
44      muduo::net::TcpConnectionPtr&,
45      muduo::net::Buffer*,
46      muduo::Timestamp,
47      ErrorCode);
48      ProtobufMessageCallback messageCallback_;
49      ErrorCallback errorCallback_;
50      const static int kHeaderLen = sizeof(int32_t);
51      const static int kMinMessageLen = 2*kHeaderLen + 2; // nameLen + typeName
    + checkSumconst static int kMaxMessageLen = 64*1024*1024; // same as
52      codec_stream.h kDefaultTotalBytesLimit
53  };
```

ProtobufDispatcher 类，这个类就比较重要了，这是一个 protobuf 请求的分发处理类，我们用户在使用的时候，就是在这个类对象中注册哪个请求应该用哪个业务函数进行处理。

它内部的 onProtobufMessage 接口就是给上边 ProtobufCodec::messageCallback_设置的回调函数，相当于 ProtobufCodec 中 onMessage 接口会设置给服务器作为消息回调函数，其内部对于接收到的数据进行基于 protobuf 协议的解析，得到请求后，通过 **ProtobufDispatcher::onProtobufMessage** 接口进行请求分发处理，也就是确定当前请求应该用哪一个注册的业务函数进行处。

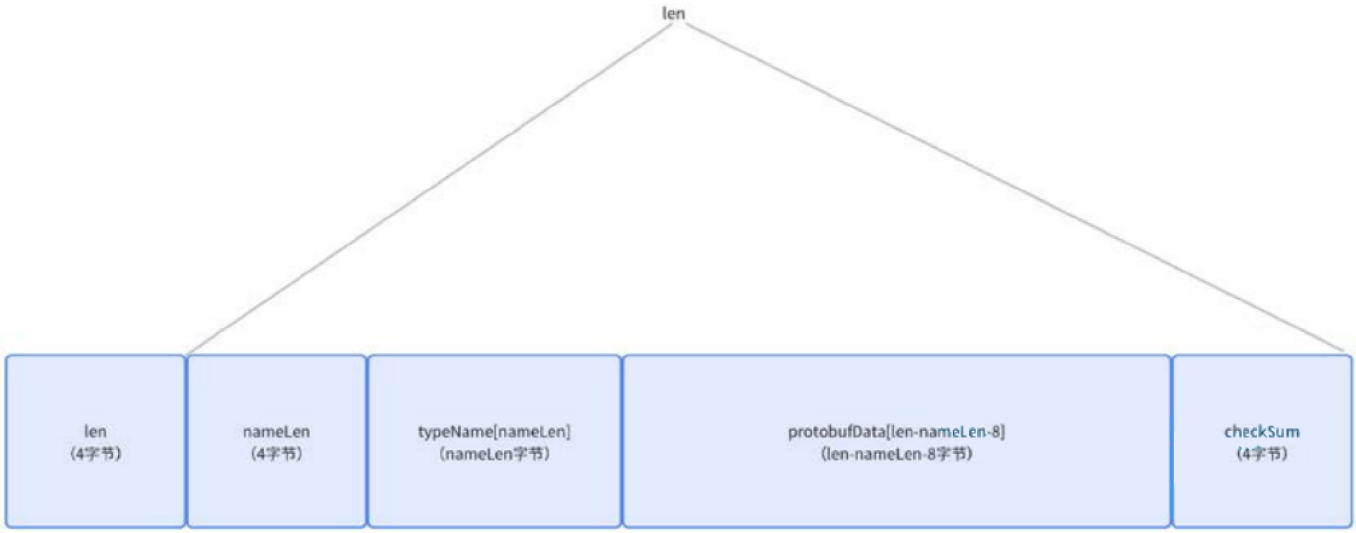
代码块

```
1   typedef std::shared_ptr<google::protobuf::Message> MessagePtr;
2   class Callback : muduo::noncopyable
3   {
4   public:
5       virtual ~Callback() = default;
6       virtual void onMessage(const muduo::net::TcpConnectionPtr&,
7       const MessagePtr& message,
8       muduo::Timestamp) const = 0;
```

```
};

//这是一个对函数接口进行二次封装生成一个统一类型对象的类template <typename T>
class CallbackT : public Callback
{
    static_assert(std::is_base_of<google::protobuf::Message, T>::value,
    "T must be derived from gpb::Message.");
public:
    typedef std::function<void (const muduo::net::TcpConnectionPtr&,
    const std::shared_ptr<T>& message,muduo::Timestamp)>
    ProtobufMessageTCallback;
    CallbackT(const ProtobufMessageTCallback& callback)
    : callback_(callback)
    {
    }
    void onMessage(const muduo::net::TcpConnectionPtr& conn,
    const MessagePtr& message,
    muduo::Timestamp receiveTime) const override{
        std::shared_ptr<T> concrete =
        muduo::down_pointer_cast<T>(message);
        assert(concrete != NULL);
        callback_(conn, concrete, receiveTime);
    }

private:
    ProtobufMessageTCallback callback_;
};

//这是一个 protobuf 请求分发器类，需要用户注册不同请求的不同处理函数，//注册完毕后，服务器收到指定请求就会使用对应接口进行处理class ProtobufDispatcher
{
public:
    typedef std::function<void (const muduo::net::TcpConnectionPtr&,
    const MessagePtr& message,
    muduo::Timestamp)>
    ProtobufMessageCallback;
    //构造对象时需要传入一个默认的业务处理函数，以便于找不到对应请求的处理函数时调用。
explicit ProtobufDispatcher(const ProtobufMessageCallback& defaultCb)
    : defaultCallback_(defaultCb)
    {
    }
    //这个是人家实现的针对 proto 中定义的类型请求进行处理的函数，内部会调用
    我们自己传入的业务处理函数
    void onProtobufMessage(const muduo::net::TcpConnectionPtr& conn,
    const MessagePtr& message,
    muduo::Timestamp receiveTime) const{
```

```
        CallbackMap::const_iterator it = callbacks_.find(message->GetDescriptor());
        if (it != callbacks_.end())
        {
            it->second->onMessage(conn, message, receiveTime);
        }

        else
        {
            defaultCallback_(conn, message, receiveTime);
        }
    }
 /*
 这个接口非常巧妙，基于 proto 中的请求类型将我们自己的业务处理函数与对应的请求给关联起来了
 相当于通过这个成员变量中的 CallbackMap 能够知道收到什么请求后应该用什么处理函数进行处理
 简单理解就是注册针对哪种请求--应该用哪个我们自己的函数进行处理的映射关系

 但是我们自己实现的函数中，参数类型都是不一样的比如翻译有翻译的请求类型，加法有加法请求类型
 而 map 需要统一的类型，这样就不好整了，所以用 CallbackT 对我们传入的接口进行了二次封装。
 */template<typename T>
 void registerMessageCallback(const typename
 CallbackT<T>::ProtobufMessageTCallback& callback){
     std::shared_ptr<CallbackT<T> > pd(new CallbackT<T>(callback));
     callbacks_[T::descriptor()] = pd;
 }

private:
    typedef std::map<const google::protobuf::Descriptor*,
    std::shared_ptr<Callback> > CallbackMap;
    CallbackMap callbacks_;
    ProtobufMessageCallback defaultCallback_;
};
```

而能实现请求与函数之间的映射，还有一个非常重要的元素：那就是应用层协议

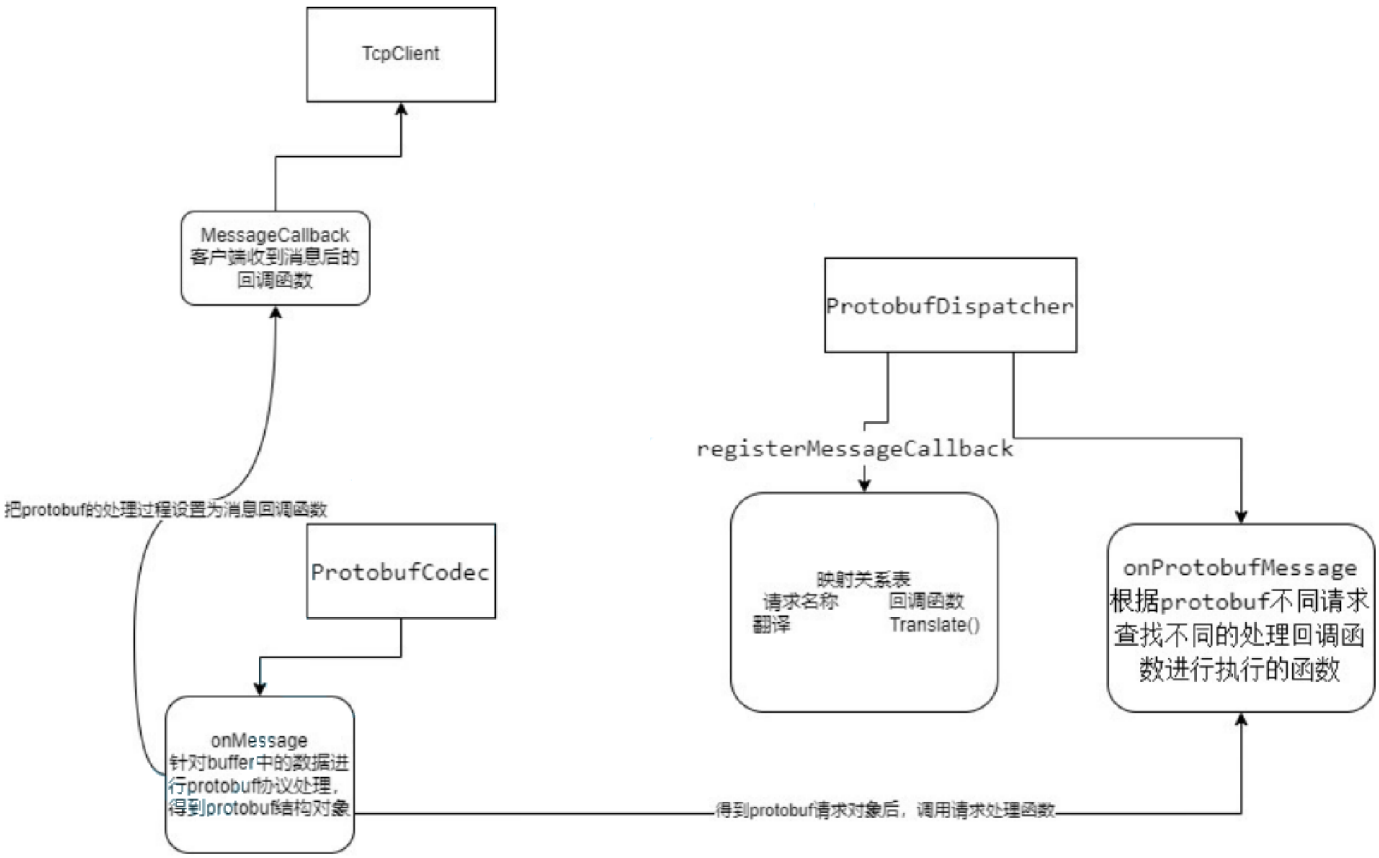


protobuf 根据我们的 **proto** 文件生成的代码中，会生成对应类型的类，比如 **TranslateRequest** 对应了一个 **TranslateRequest** 类，而不仅仅如此，**protobuf** 比我们想象中做的事情更多，每个对应的类中，都包含有一个描述结构的指针：

代码块

```
1   static const ::google::protobuf::Descriptor* descriptor();
```

这个描述结构非常重要，其内部可以获取到当前对应类类型名称，以及各项成员的名称，因此通过这些名称，加上协议中的 typename 字段，就可以实现完美的对应关系了。



了解了上述关系，接下来就可以通过 muduo 库中陈硕大佬提供的接口来编写我们的客户端/服务器端通信了，其最为简便之处就在于我们可以把更多的精力放到业务处理函数的实现上，而不是服务器的搭建或者协议的解析处理上了。

服务端：

代码块

```
#include "muduo/proto/codec.h"#include "muduo/proto/dispatcher.h"#include "muduo/base/Logging.h"#include "muduo/base/Mutex.h"#include "muduo/net/EventLoop.h"#include "muduo/net/TcpServer.h"#include "Request.pb.h"#include <iostream>#include <memory>#include <unordered_map>class Server
{
public:
    typedef std::shared_ptr<google::protobuf::Message> MessagePtr;
    typedef std::shared_ptr<pkg::TranslateQuest> TranslateRequestPtr;
    typedef std::shared_ptr<pkg::TranslateResponse> TranslateResponsePtr;
    typedef std::shared_ptr<pkg::AddQuest> AddQuestPtr;
    typedef std::shared_ptr<pkg::AddResponse> AddResponsePtr;

```

```
    Server(uint16_t port) :_server(&_baseloop, muduo::net::InetAddress("0.0.0.0", port),
        "Server", muduo::net::TcpServer::kReusePort)
        , _dispatcher(std::bind(&Server::onUnknownMessage, this,
            std::placeholders::_1, std::placeholders::_2, std::placeholders::_3))
        , _codec(std::bind(&ProtobufDispatcher::onProtobufMessage, &_dispatcher,
            std::placeholders::_1, std::placeholders::_2, std::placeholders::_3))
        {
            // 注册请求处理函数
            _dispatcher.registerMessageCallback<pkg::TranslateQuest>(std::bind(&Server::onTranslate, this,
                std::placeholders::_1, std::placeholders::_2, std::placeholders::_3));

            _dispatcher.registerMessageCallback<pkg::AddQuest>(std::bind(&Server::onAdd, this,
                std::placeholders::_1, std::placeholders::_2, std::placeholders::_3));

            _server.setMessageCallback(std::bind(&ProtobufCodec::onMessage, &_codec,
                std::placeholders::_1, std::placeholders::_2, std::placeholders::_3));

            _server.setConnectionCallback(std::bind(&Server::onConnection, this, std::placeholders::_1));
        }

        void Start(){
            _server.start();
            _baseloop.loop();
        }

private:
    std::string Translate(const std::string& str){
        static std::unordered_map<std::string, std::string> dict_map = {
            {"hello", "你好"},
            {"cat", "哈基米"},
            {"曼波", "难买绿豆"}
        };

        auto it = dict_map.find(str);
        if(it == dict_map.end()) return "不再曼波";
        std::string ret =  "翻译结果: " + dict_map[str];
```

```
46          return ret;
47      }
48
49      void onTranslate(const muduo::net::TcpConnectionPtr& conn, const
    TranslateRequestPtr& message, muduo::Timestamp){
50          // 1. 提取msg中有效消息， 也就是所需翻译内容
51          std::string req_msg = message->msg();
52
53          // 2. 进行翻译得到结果
54          std::string rsp_msg = Translate(req_msg);
55
56          // 3. 组织protobuf的响应
57          pkg::TranslateResponse resp;
58          resp.set_msg(rsp_msg);
59
60          // 4. 发送响应
61          _codec.send(conn, resp);
62      }
63
64      void onAdd(const muduo::net::TcpConnectionPtr& conn, const AddQuestPtr&
    message, muduo::Timestamp){
65          int num1 = message->num1();
66          int num2 = message->num2();
67          int result = num1 + num2;
68
69          pkg::AddResponse resp;
70          resp.set_result(result);
71
72          _codec.send(conn, resp);
73      }
74
75      void onUnknownMessage(const muduo::net::TcpConnectionPtr& conn, const
    MessagePtr& message, muduo::Timestamp){
76          LOG_INFO << "onUnknownMessage: " << message->GetTypeName();
77          conn->shutdown();
78      }
79
80      void onConnection(const muduo::net::TcpConnectionPtr& conn){
81          if(conn->connected())
82          {
83              LOG_INFO << "新连接建立成功！ ";
84          }
85
86          else
87          {
88              LOG_INFO << "连接即将关闭！ ";
89          }
```

```
    }

private:
    muduo::net::EventLoop _baseloop;
    muduo::net::TcpServer _server;   // 服务器对象
    ProtobufDispatcher _dispatcher;  // 请求分发器对象 -- 用于注册请求处理函数
    ProtobufCodec _codec;            // protobuf协议处理器 -- 针对收到的请求数据进行protobuf协议处理
};

int main(){
    Server server(8085);
    server.Start();
    return 0;
}
```

客户端：

代码块

```
#include "muduo/proto/dispatcher.h"#include "muduo/proto/codec.h"#include "muduo/base/Logging.h"#include "muduo/base/Mutex.h"#include "muduo/net/EventLoop.h"#include "muduo/net/TcpClient.h"#include "muduo/net/EventLoopThread.h"#include "muduo/base/CountDownLatch.h"#include "Request.pb.h"#include <iostream>#include <string>class Client
{
public:
    typedef std::shared_ptr<google::protobuf::Message> MessagePtr;
    typedef std::shared_ptr<pkg::AddResponse> AddResponsePtr;
    typedef std::shared_ptr<pkg::TranslateResponse> TranslateResponsePtr;


    Client(const std::string& ip, uint16_t port)
        : _latch(1), _client(_loopthread.startLoop(), muduo::net::InetAddress(ip, port), "Client")
        , _dispatcher(std::bind(&Client::onUnknownMessage, this,
            std::placeholders::_1, std::placeholders::_2, std::placeholders::_3))
        , _codec(std::bind(&ProtobufDispatcher::onProtobufMessage, &_dispatcher,
            std::placeholders::_1, std::placeholders::_2, std::placeholders::_3))
    {
        // 注册请求处理函数
```

```
17          _dispatcher.registerMessageCallback<pkg::TranslateResponse>
    (std::bind(&Client::onTranslate, this,
18              std::placeholders::_1, std::placeholders::_2,
    std::placeholders::_3));
19
20          _dispatcher.registerMessageCallback<pkg::AddResponse>
    (std::bind(&Client::onAdd, this,
21              std::placeholders::_1, std::placeholders::_2,
    std::placeholders::_3));
22
23          _client.setMessageCallback(std::bind(&ProtobufCodec::onMessage,
    &_codec,
24              std::placeholders::_1, std::placeholders::_2,
    std::placeholders::_3));
25
26          _client.setConnectionCallback(std::bind(&Client::onConnection, this,
    std::placeholders::_1));
27      }
28
29      // 连接服务器 -- 需要等待连接建立成功之后才返回void Connect()
30      {
31          _client.connect();
32          _latch.wait(); // 阻塞等待直到连接建立成功
33      }
34
35      void Translate(const std::string &msg){
36          pkg::TranslateQuest req;
37          req.set_msg(msg);
38          Send(&req);
39      }
40
41      void Add(int num1, int num2){
42          pkg::AddQuest req;
43          req.set_num1(num1);
44          req.set_num2(num2);
45          Send(&req);
46      }
47
48  private:
49      bool Send(const google::protobuf::Message* message){
50          if(_conn->connected())
51          {
52              _codec.send(_conn, *message);
53              return true;
54          }
55
56          return false;
```

```
57      }
58
59      void onUnknownMessage(const muduo::net::TcpConnectionPtr& conn, const
    MessagePtr& message, muduo::Timestamp){
60          LOG_INFO << "onUnknownMessage: " << message->GetTypeName();
61          conn->shutdown();
62      }
63
64      void onTranslate(const muduo::net::TcpConnectionPtr& conn, const
    TranslateResponsePtr& message, muduo::Timestamp){
65          std::cout << "翻译结果: " << message->msg() << std::endl;
66      }
67
68      void onAdd(const muduo::net::TcpConnectionPtr& conn, const AddResponsePtr&
    message, muduo::Timestamp){
69          std::cout << "加法结果: " << message->result() << std::endl;
70      }
71
72      void onConnection(const muduo::net::TcpConnectionPtr& conn){
73          if(conn->connected())
74          {
75              _latch.countDown(); // 唤醒主线程中的阻塞
76              _conn = conn;
77          }
78
79          else
80          {
81              // 连接关闭
82              _conn.reset();
83          }
84      }
85
86  private:
87      muduo::CountDownLatch _latch;               // 实现同步
88      muduo::net::EventLoopThread _loopthread;    // 异步循环处理线程
89      muduo::net::TcpConnectionPtr _conn;         // 客户端对应的连接
90      muduo::net::TcpClient _client;              // 客户端
91      ProtobufDispatcher _dispatcher;             // 请求分发器
92      ProtobufCodec _codec;                       // 协议处理器
93  };
94
95  int main(){
96      Client client("127.0.0.1", 8085);
97      client.Connect();
98
99      client.Translate("曼波");
100     client.Add(11,22);
```

```
101
102         sleep(1);
103
104         return 0;
105     }
```

SQLite3

什么是 SQLite

SQLite 是一个进程内的轻量级数据库，它实现了自给自足的、无服务器的、零配置的、事务性的 SQL 数据库引擎。它是一个零配置的数据库，这意味着与其他数据库不一样，我们不需要在系统中配置。像其他数据库，SQLite 引擎不是一个独立的进程，可以按应用程序需求进行静态或动态连接，SQLite 直接访问其存储文件

为什么要用 SQLite

· 不需要一个单独的服务器进程或操作的系统（无服务器的）

· SQLite 不需要配置

· 一个完整的 SQLite 数据库是存储在一个单一的跨平台的磁盘文件

· SQLite 是非常小的，是轻量级的，完全配置时小于 400KiB，省略可选功能配置时小于 250KiB

· SQLite 是自给自足的，这意味着不需要任何外部的依赖

· SQLite 事务是完全兼容 ACID 的，允许从多个进程或线程安全访问

· SQLite 支持 SQL92（SQL2）标准的大多数查询语言的功能

· SQLite 使用 ANSI-C 编写的，并提供了简单和易于使用的 API

· SQLite 可在 UNIX（Linux, Mac OS-X, Android, iOS）和 Windows（Win32, WinCE, WinRT）中运行

SQLite3 C/C++ API 介绍

C/C++ API 是 SQLite3 数据库的一个客户端， 提供一种用 C/C++操作数据库的方法。

下面介绍一下常见的几个接口：

SQLite3 官方文档：https://www.sqlite.org/c3ref/funclist.html

代码块

```
1   sqlite3 操作流程:
```

```
0. 查看当前数据库在编译阶段是否启动了线程安全
    int sqlite3_threadsafe();  0-未启用;  1-启用
    需要注意的是 sqlite3 是有三种安全等级的:
    1. 非线程安全模式
    2. 线程安全模式（不同的连接在不同的线程/进程间是安全的，即一个句柄不能用于多线程间）
    3. 串行化模式（可以在不同的线程/进程间使用同一个句柄）
1. 创建/打开数据库文件，并返回操作句柄
    int sqlite3_open(const char *filename, sqlite3 **ppDb) 成功返回SQLITE_OK
    //若在编译阶段启动了线程安全，则在程序运行阶段可以通过参数选择线程安全等级int sqlite3_open_v2(const char *filename, sqlite3 **ppDb, int flags, const char *zVfs );
    flag:
    SQLITE_OPEN_READWRITE -- 以可读可写方式打开数据库文件
    SQLITE_OPEN_CREATE -- 不存在数据库文件则创建
    SQLITE_OPEN_NOMUTEX--多线程模式，只要不同的线程使用不同的连接即可保证线程安全
    SQLITE_OPEN_FULLMUTEX--串行化模式返回: SQLITE_OK 表示成功
2. 执行语句
    int sqlite3_exec(sqlite3*, char *sql, int (*callback)(void*,int,char**,char**),
    void* arg, char **err)int (*callback)(void*,int,char**,char**)void* : 是设置的在回调时传入的 arg 参数
    int: 一行中数据的列数
    char**: 存储一行数据的字符指针数组
    char**: 每一列的字段名称
    这个回调函数有个 int 返回值，成功处理的情况下必须返回 0，返回非 0会触发 ABORT 退出程序返回: SQLITE_OK 表示成功
3. 销毁句柄
    int sqlite3_close(sqlite3* db); 成功返回 SQLITE_OK
    int sqlite3_close_v2(sqlite3*); 推荐使用--无论如何都会返回SQLITE_OK获取错误信息
    const char *sqlite3_errmsg(sqlite3* db);
```

SQLite3 C/C++ API 使用

下面我们将这几个接口封装成一个类，快速上手这几个接口

代码块

```
class SqliteHelper
{
public:
    typedef int (*sqlite_callback)(void *, int, char **, char **);
    SqliteHelper() : _db_handler(nullptr) {}
    ~SqliteHelper() { close(); }
    bool connect(const std::string &filename,
```

```
                  uint32_t thread_safe_level = SQLITE_OPEN_FULLMUTEX){
        uint32_t flag = SQLITE_OPEN_READWRITE |
                        SQLITE_OPEN_CREATE | thread_safe_level;
        if (sqlite3_open_v2(filename.c_str(),
                            &_db_handler, flag, nullptr) != SQLITE_OK)
        {
            std::cout << "open db file " << filename << "
                failed !\n ";
                std::cout
                      << sqlite3_errmsg(_db_handler) << std::endl;
            sqlite3_close(_db_handler);
            return false;
        }
        return true;
    }
    void close(){
        if (_db_handler)
            sqlite3_close(_db_handler);
        _db_handler = nullptr;
    }
    bool begin(){
        int ret = sqlite3_exec(_db_handler, "begin;",
                               nullptr, nullptr, nullptr);
        if (ret != SQLITE_OK)
        {
            std::cout << "begin transaction failed reason: ";
            std::cout << sqlite3_errmsg(_db_handler) << std::endl;
            return false;
        }
        return true;
    }
    bool commit(){
        int ret = sqlite3_exec(_db_handler, "commit;",
                               nullptr, nullptr, nullptr);
        if (ret != SQLITE_OK)
        {
            std::cout << "commit transaction failed reason: ";
            std::cout << sqlite3_errmsg(_db_handler) << std::endl;
            return false;
        }
        return true;
    }
    bool rollback(const std::string &point = ""){
        char sql[1024];
        if (point.empty())
        {
            sprintf(sql, "%s;", "rollback");
```

```
55          }
56          else
57          {
58              sprintf(sql, "rollback to %s;", point.c_str());
59          }
60          int ret = sqlite3_exec(_db_handler, sql, nullptr,
61                                 nullptr, nullptr);
62          if (ret != SQLITE_OK)
63          {
64              std::cout << "rollback transaction failed reason:
65                           ";
66                  std::cout
67                        << sqlite3_errmsg(_db_handler) << std::endl;
68              return false;
69          }
70          return true;
71      }
72      bool savepoint(const std::string &point){
73          char sql[1024];
74          sprintf(sql, "savepoint %s;", point.c_str());
75          int ret = sqlite3_exec(_db_handler, sql, nullptr,
76                                 nullptr, nullptr);
77          if (ret != SQLITE_OK)
78          {
79              std::cout << "savepoint failed reason: ";
80              std::cout << sqlite3_errmsg(_db_handler) << std::endl;
81              return false;
82          }
83          return true;
84      }
85      bool excute(const std::string &sql, sqlite_callback cb = nullptr,
86                  void *arg = nullptr){
87          int ret = sqlite3_exec(_db_handler, sql.c_str(), cb,
88                                 arg, nullptr);
89          if (ret != SQLITE_OK)
90          {
91              std::cout << "excute sql: " << sql << "
92                  failed !\n ";
93                  std::cout
94                        << "reason: " << sqlite3_errmsg(_db_handler) <<
    std::endl;
95              return false;
96          }
97          return true;
98      }
99
100 private:
```

```
101     sqlite3 *_db_handler;
102 };
```

测试程序：

代码块

```
1  #include "db.hpp"#include <cassert>int get_sn_and_name(void *arg, int count,
   char **row, char **name){
2      for (int i = 0; i < count && row[i] && name[i]; i++)
3      {
4          std::cout << name[i] << "=" << row[i] << "\t";
5      }
6      std::cout << std::endl;
7      return 0;
8  }
9
10 void student_table(){
11     SqliteHelper sh;
12     assert(sh.connect("./student.db"));
13     const std::string sql1 = "create table if not exists student(""sn int
   primary key,""name varchar(32));";
14     assert(sh.excute(sql1));
15     assert(sh.begin());
16     const std::string sql2 = "insert into student values(1, 'null');";
17     assert(sh.excute(sql2));
18     assert(sh.savepoint("point1"));
19     const std::string sql3 = "insert into student values(2, null);";
20     assert(sh.excute(sql3));
21     assert(sh.rollback("point1"));
22     assert(sh.commit());
23     const std::string sql4 = "select sn, name from student;";
24     assert(sh.excute(sql4, get_sn_and_name));
25 }
26
27 int main(){
28     student_table();
29     return 0;
30 }
```

GTest

GTest 是什么

GTest 是一个**跨平台的** C++单元测试框架，由 google 公司发布。gtest 是为了在不同平台上为编写 C++单元测试而生成的。它提供了丰富的断言、致命和非致命判断、参数化等等。

GTest 使用

TEST 宏

代码块

```
TEST(test_case_name, test_name)
TEST_F(test_fixture,test_name)
```

· **TEST**：主要用来创建一个简单测试， 它定义了一个测试函数， 在这个函数中可以使用任何C++代码并且使用框架提供的断言进行检查。

· **TEST_F**：主要用来进行多样测试，适用于多个测试场景如果需要相同的数据配置的情况， 即相同的数据测不同的行为。

断言

GTest 中的断言的宏可以分为两类：

· ASSERT_系列：如果当前点检测失败则退出当前函数

· EXPECT_系列：如果当前点检测失败则继续往下执行

下面是经常使用的断言介绍：

代码块

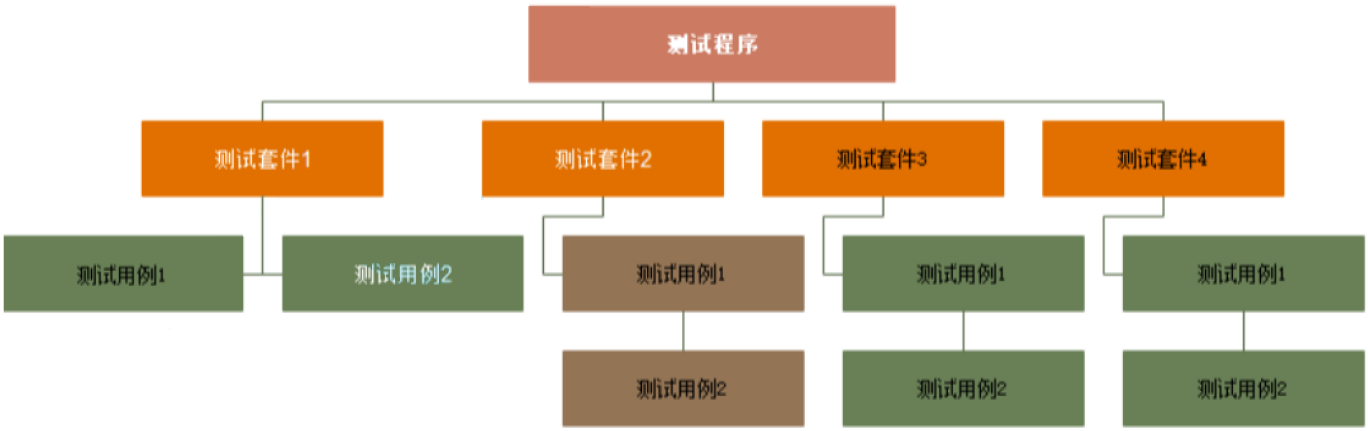
```
// bool 值检查ASSERT_TRUE(参数)，期待结果是 trueASSERT_FALSE(参数)，期待结果是 false//数值型数据检查ASSERT_EQ(参数 1，参数 2)，传入的是需要比较的两个数 equal
ASSERT_NE(参数 1，参数 2)，not equal，不等于才返回 trueASSERT_LT(参数 1，参数 2)，less than，小于才返回 trueASSERT_GT(参数 1，参数 2)，greater than，大于才返回 trueASSERT_LE(参数 1，参数 2)，less equal，小于等于才返回 trueASSERT_GE(参数 1，参数 2)，greater equal，大于等于才返回 true
```

注：如果你对自动输出的错误信息不满意的话，也可以通过 operator<< 在失败的时候打印自

定义日志。

事件机制

GTest 中的事件机制是指在测试前和测试后提供给用户自行添加操作的机制，而且该机制也可以让同一测试套件下的测试用例共享数据。GTest 框架中事件的结构层次：



· 测试程序：一个测试程序只有一个 main 函数，也可以说是一个可执行程序是一个测试程序。该级别的事件机制是在程序的开始和结束执行。

· 测试套件：代表一个测试用例的集合体，该级别的事件机制是在整体的测试案例开始和结束执行

· 测试用例：该级别的事件机制是在每个测试用例开始和结束都执行事件机制的最大好处就是能够为我们各个测试用例提前准备好测试环境，并在测试完毕后用于销毁环境，这样有个好处就是如果我们有一端代码需要进行多种不同方法的测试，则可以通过测试机制在每个测试用例进行之前初始化测试环境和数据，并在测试完毕后清理测试造成的影响。

GTest 提供了三种常见的的事件：

· 全局事件：针对整个测试程序。实现全局的事件机制，需要创建一个自己的类，然后继承 **testing::Environment** 类，然后分别实现成员函数 SetUp 和 TearDown，同时在 main 函数内进行调用 **testing::AddGlobalTestEnvironment(new MyEnvironment)**; 函数添加全局的事件机制

代码块

```
#include <iostream>#include <gtest/gtest.h>// 全局事件:针对整个测试程序,提供全局事件机制,能够在测试之前配置测试环境数据,测试完毕后清理数据// 先定义环境类,通过继承 testing::Environment 的派生类来完成// 重写的虚函数接口 SetUp 会在测试之前被调用; TearDown 会在测试完毕后调用.
std::unordered_map<std::string, std::string> dict;
class HashTestEnv : public testing::Environment
{
```

```
public:
    virtual void SetUp() override{
        std::cout << "测试前:提前准备数据!!\n";
        dict.insert(std::make_pair("Hello", "你好"));
        dict.insert(std::make_pair("hello", "你好"));
        dict.insert(std::make_pair("雷吼", "你好"));
    }

    virtual void TearDown() override{
        std::cout << "测试结束后:清理数据!!\n";
        dict.clear();
    }
};

TEST(hash_case_test, find_test)
{
    auto it = dict.find("hello");
    ASSERT_NE(it, dict.end());
}

TEST(hash_case_test, size_test)
{
    ASSERT_GT(dict.size(), 0);
}

int main(int argc, char *argv[]){
    testing::AddGlobalTestEnvironment(new HashTestEnv);
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

运行结果：

代码块

```
./event
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
测试前:提前准备数据!!
[----------] 2 tests from hash_case_test
[ RUN ] hash_case_test.find_test
[ OK ] hash_case_test.find_test (0 ms)
[ RUN ] hash_case_test.size_test
[ OK ] hash_case_test.size_test (0 ms)
```

```
[----------] 2 tests from hash_case_test (0 ms total)
[----------] Global test environment tear-down
测试结束后:清理数据!!
[==========] 2 tests from 1 test case ran. (0 ms total)
[ PASSED ] 2 tests
```

· TestSuite 事件：针对一个个测试套件。测试套件的事件机制我们同样需要去创建一个类，继承自 testing::Test，实现两个静态函数 SetUpTestCase 和 TearDownTestCase，测试套件的事件机制不需要像全局事件机制一样在 main 注册，而是需要将我们平时使用的 TEST 宏改为 TEST_F 宏。

- SetUpTestCase() 函数是在测试套件第一个测试用例开始前执行
- TearDownTestCase() 函数是在测试套件最后一个测试用例结束后执行
- 需要注意 TEST_F 的第一个参数是我们创建的类名，也就是当前测试套件的名称，这样在 TEST_F 宏的测试套件中就可以访问类中的成员了。

代码块

```
#include <iostream>#include <gtest/gtest.h>// TestSuite:测试套件/集合进行单元测试,即,将多个相关测试归入一组的方式进行测试, 为这组测试用例进行环境配置和清理// 概念: 对一个功能的验证往往需要很多测试用例, 测试套件就是针对一组相关测试用例进行环境配置的事件机制/*用法: 先定义环境类,继承于 testing::Test 基类,
    重写两个静态函数SetUpTestCase /TearDownTestCase
    进行环境的配置和清理 */class HashTestEnv1 : public testing::Test
{
public:
    static void SetUpTestCase(){
        std::cout << "环境 1 第一个 TEST 之前调用\n";
    }
    static void TearDownTestCase(){
        std::cout << "环境 1 最后一个 TEST 之后调用\n";
    }

public:
    std::unordered_map<std::string, std::string> dict;
};

// 注意,测试套件使用的不是 TEST 了,而是 TEST_F, 而第一个参数名称就是测试套件环境类名称// main 函数中不需要再注册环境了,而是在 TEST_F 中可以直接访问类的成员变量和成员函数TEST_F(HashTestEnv1, insert_test)
{
    std::cout << "环境 1,中间 insert 测试\n";
    dict.insert(std::make_pair("Hello", "你好"));
```

```
    dict.insert(std::make_pair("hello", "你好"));
    dict.insert(std::make_pair("雷吼", "你好"));
    auto it = dict.find("hello");
    ASSERT_NE(it, dict.end());
}

TEST_F(HashTestEnv1, sizeof)
{
    std::cout << "环境 1,中间 size 测试\n";
    ASSERT_GT(dict.size(), 0);
}

int main(int argc, char *argv[]){
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

运行结果：

代码块

```
./event
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from HashTestEnv1
环境 1 第一个 TEST 之前调用
[ RUN ] HashTestEnv1.insert_test
环境 1,中间 insert 测试
[ OK ] HashTestEnv1.insert_test (0 ms)
[ RUN ] HashTestEnv1.sizeof
环境 1,中间 size 测试
event.cpp:81: Failure
Expected: (dict.size()) > (0), actual: 0 vs 0
[ FAILED ] HashTestEnv1.sizeof (0 ms)
环境 1 最后一个 TEST 之后调用
[----------] 2 tests from HashTestEnv1 (0 ms total)
[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (1 ms total)
[ PASSED ] 1 test.
[ FAILED ] 1 test, listed below:
[ FAILED ] HashTestEnv1.sizeof
1 FAILED TEST
```

能够看到在上例中，有一个好处，就是将数据与测试结合到同一个测试环境类中了，这样与外界的耦合度更低，代码也更清晰。但是同样的，我们发现在两个测试用例中第二个测试用例失败了，这是为什么呢？这就涉及到了 **TestCase** 事件的机制。

· TestCase 事件: 针对一个个测试用例。测试用例的事件机制的创建和测试套件的基本一样，不同地方在于测试用例实现的两个函数分别是 SetUp 和 TearDown, 这两个函数也不是静态函数。

◦ SetUp()函数是在一个测试用例的开始前执行

◦ TearDown()函数是在一个测试用例的结束后执行

也就是说，在 TestSuite/TestCase 事件中，每个测试用例，虽然它们同用同一个事件环境类，可以访问其中的资源，但是本质上每个测试用例的环境都是独立的，这样我们就不用担心不同的测试用例之间会有数据上的影响了，保证所有的测试用例都使用相同的测试环境进行测试。

代码块

```
// TestCase:测试用例的单元测试,即针对每一个测试用例都使用独立的测试环境数据进行测试// 概念:它是针对测试用例进行环境配置的一种事件机制// 用法:先定义环境类,继承于 testing::Test 基类, 在环境类内重写SetUp / TearDown 接口 class HashTestEnv2 : public testing::Test
{
public:
    static void SetUpTestCase(){
        std::cout << "环境 2 第一个 TEST 之前被调用,进行总体环境配置\n ";
    }

    static void TearDownTestCase(){
        std::cout << "环境 2 最后一个 TEST 之后被调用,进行总体环境清理\n ";
    }

    virtual void SetUp() override{
        std::cout << "环境 2 测试前:提前准备数据!!\n";
        dict.insert(std::make_pair("bye", "再见"));
        dict.insert(std::make_pair("see you", "再见"));
    }

    virtual void TearDown() override{
        std::cout << "环境 2 测试结束后:清理数据!!\n";
        dict.clear();
    }

public:
    std::unordered_map<std::string, std::string> dict;
```

```
};

TEST_F(HashTestEnv2, insert_test)
{
    std::cout << "环境 2,中间测试\n";
    dict.insert(std::make_pair("hello", "你好"));
    ASSERT_EQ(dict.size(), 3);
}

TEST_F(HashTestEnv2, size_test)
{
    std::cout << "环境 2,中间 size 测试\n";
    auto it = dict.find("hello");
    ASSERT_EQ(it, dict.end());
    ASSERT_EQ(dict.size(), 2);
}

int main(int argc, char *argv[]){
    testing::InitGoogleTest(&argc, argv);
    RUN_ALL_TESTS();
    return 0;
}
```

运行结果：

代码块

```
./event
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from HashTestEnv2
环境 2 第一个 TEST 之前被调用,进行总体环境配置
[ RUN ] HashTestEnv2.insert_test
环境 2 测试前:提前准备数据!!
环境 2,中间测试
环境 2 测试结束后:清理数据!!
[ OK ] HashTestEnv2.insert_test (1 ms)
[ RUN ] HashTestEnv2.size_test
环境 2 测试前:提前准备数据!!
环境 2,中间 size 测试
环境 2 测试结束后:清理数据!!
[ OK ] HashTestEnv2.size_test (0 ms)
环境 2 最后一个 TEST 之后被调用,进行总体环境清理
[----------] 2 tests from HashTestEnv2 (1 ms total)
```

```
[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (1 ms total)
[ PASSED ] 2 tests
```

C++11 异步线程池

std::future

std::future 是 C++11 标准库中的一个模板类，它表示一个**异步操作的结果**。当我们在多线程编程中使用异步任务时，**std::future** 可以帮助我们在需要的时候获取任务的执行结果。**std::future** 的一个重要特性是能够阻塞当前线程，直到异步操作完成，从而确保我们在获取结果时不会遇到未完成的操作。

应用场景

· 异步任务： 当我们需要在后台执行一些耗时操作时，如网络请求或计算密集型任务等 **std::future** 可以用来表示这些异步任务的结果。通过将任务与主线程分离，我们可以实现任务的并行处理，从而提高程序的执行效率。

· 并发控制： 在多线程编程中，我们可能需要等待某些任务完成后才能继续执行其他操作。通过使用 **std::future**，我们可以实现线程之间的同步，确保任务完成后再获取结果并继续执行后续操作。

· 结果获取：**std::future** 提供了一种安全的方式来获取异步任务的结果。我们可以使用 **std::future::get()** 函数来获取任务的结果，此函数会阻塞当前线程，直到异步操作完成。这样，在调用 **get()** 函数时，我们可以确保已经获取了所需的结果。

用法示例

· 使用 **std::async** 关联异步任务

std::async 是一种将任务与 **std::future** 关联的简单方法。它创建并运行一个异步任务，并返回一个与该任务结果关联的 **std::future** 对象。默认情况下，**std::async** 是否启动一个新线程，或者在等待 **future** 时，任务是否同步运行都取决于你给的 参数。这个参数为 **std::launch** 类型：

- **std::launch::deferred** 表明该函数会被延迟调用，直到在 **future** 上调用 **get()** 或者 **wait()** 才会开始执行任务。

- **std::launch::async** 表明函数会在自己创建的线程上运行。
- **std::launch::deferred | std::launch::async** 内部通过系统等条件自动选择策略。

代码块

```
#include <iostream>#include <future>#include <chrono>int aysnc_task(){
    std::this_thread::sleep_for(std::chrono::seconds(3));
    return 2;
}

int main(){
    // 关联异步任务 aysnc_task 和 futrue
    std::future<int> result_future =
    std::async(std::launch::async, aysnc_task);
    // 此处可执行其他操作，无需等待
    std::cout << "hello world!" << std::endl;
    // 获取异步任务结果int result = result_future.get();
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

· 使用 std::packaged_task 和 std::future 配合

std::packaged_task 就是将任务和 **std::future** 绑定在一起的模板，是一种对任务的封装。我们可以通过 **std::packaged_task** 对象获取任务相关联的 std::future 对象，通过调用 **get_future()** 方法获得。**std::packaged_task** 的模板参数是**函数签名。**

可以把 **std::future** 和 **std::async** 看成是分开的， 而 **std::packaged_task** 则是一个整体。

代码块

```
#include <iostream>#include <future>#include <chrono>int add(int num1, int
num2){
    return num1 + num2;
}

int main(){
    // 封装任务std::packaged_task<int(int, int)> task(add);
    // 此处可执行其他操作，无需等待
    std::cout << "hello world!" << std::endl;
    std::future<int> result_future = task.get_future();

    // 这里必须要让任务执行，否则在 get()获取 future 的值时会一直阻塞task(1, 2);

```

```
    // 获取异步任务结果int result = result_future.get();
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

异步执行 std::packaged_task 任务：

代码块

```
#include <iostream>#include <future>#include <chrono>#include <memory>int add(int num1, int num2){
    std::this_thread::sleep_for(std::chrono::seconds(3));
    return num1 + num2;
}

int main(){
    // 封装任务// std::packaged_task<int(int, int)> task(add);// 此处可执行其他操作，无需等待// std::cout << "hello bit!" << std::endl;// std::future<int> result_future = task.get_future();// 需要注意的是，task 虽然重载了()运算符，但 task 并不是一个函数，// std::async(std::launch::async, task, 1, 2); //--错误用法// 所以导致它作为线程的入口函数时，语法上看没有问题，但是实际编译的时候会报错// std::thread(task, 1, 2); //---错误用法// 而 packaged_task 禁止了拷贝构造，// 且因为每个 packaged_task 所封装的函数签名都有可能不同，因此也无法当作参数一样传递// 传引用不可取，毕竟任务在多线程下执行存在局部变量声明周期的问题，因此不能传引用// 因此想要将一个 packaged_task 进行异步调用，// 简单方法就只能是 new packaged_task，封装函数传地址进行解引用调用了// 而类型不同的问题，在使用的时候可以使用类型推导来解决auto task = std::make_shared<std::packaged_task<int(int, int)>>(add);
    std::future<int> result_future = task->get_future();
    std::thread thr([task]() { (*task)(1, 2); });
    thr.detach();
    // 获取异步任务结果int result = result_future.get();
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

· 使用 std::promise 和 std::future 配合

std::promise 提供了一种设置值的方式，它可以在设置之后通过相关联的 **std::future** 对象进行读取。换种说法就是之前说过 **std::future** 可以读取一个异步函数的返回值了， 但是要等待就绪，而 **std::promise** 就提供一种 方式手动让 **std::future** 就绪。

```
代码块
#include <iostream>#include <future>#include <chrono>void task(std::promise<int> result_promise){
    int result = 2;
    std::cout << "task result:" << result << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(3));
    result_promise.set_value(result);
}

int main(){
    // 创建 promise
    std::promise<int> result_promise;
    std::future<int> result_future = result_promise.get_future();

    // 创建一个新线程， 执行长时间运行的任务std::thread task_thread(task, std::move(result_promise));

    // 此处可执行其他操作，无需等待
    std::cout << "hello world!" << std::endl;

    // 获取异步任务结果int result = result_future.get();
    std::cout << "Result: " << result << std::endl;
    task_thread.join();

    return 0;
}
```

c++11 线程池实现

基于线程池执行任务的时候，入口函数内部执行逻辑是固定的，因此选择 **std::packaged_task** 加上 **std::future** 的组合来实现。

线程池的工作思想：

a. 用户传入要执行的函数，以及需要处理的数据（函数的参数），由线程池中的 工作线程来执行函数完成任务。

实现：

- 管理的成员
  - 任务池：用 vector 维护的一个函数任务池子
  - 互斥锁 & 条件变量： 实现同步互斥
  - 一定数量的工作线程：用于不断从任务池取出任务执行任务
  - 结束运行标志：以便于控制线程池的结束。

- 管理的操作：
  - 入队任务：入队一个函数和参数
  - 停止运行：终止线程池

代码块

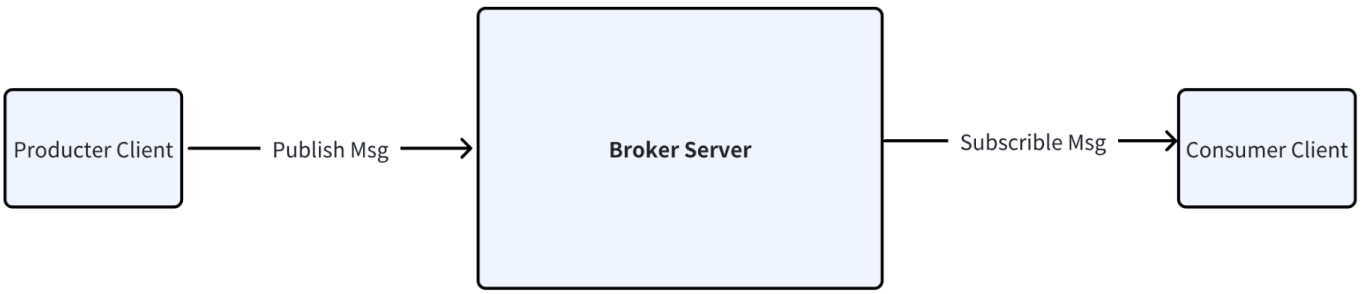
```
#pragma once#include<iostream>#include<vector>#include<functional>#include<memory>#include<thread>#include<future>#include<condition_variable>#include<mutex>namespace ThreadPoolModule
{
    class ThreadPool
    {
    public:
        using ptr = std::shared_ptr<ThreadPool>;
        using Functor = std::function<void(void)>;

        ThreadPool(int thr_count = 1) :_stop(false)
        {
            for(int i = 0; i < thr_count; i++)
            {
                _threads.emplace_back(&ThreadPool::Entry, this);
            }
        }

        ~ThreadPool()
        {
            Stop();
        }

        // push传入的是首先有一个函数--用户要执行的函数， 接下来是不定参，表示要处理的数据也就是要传入到函数中的参数// push函数内部，会将这个传入的函数封装成一个异步任务(packaged_task)，// 使用lamada生成可调用对象（内部执行异步任务），抛入到任务池中，由工作线程取出进行执行template<typename F, typename ...Args>
        auto Push(F&& func, Args&& ...args) -> std::future<decltype(func(args...))> {

            // 1. 将传入函数封装成一个packaged_task任务using return_type = decltype(func(args...));
            auto tmp_func = std::bind(std::forward<F>(func), std::forward<Args>(args)...);
            auto task = std::make_shared<std::packaged_task<return_type()>>(tmp_func);
            std::future<return_type> fu = task->get_future();

            // 2. 构造一个lamada匿名函数(捕获任务对象)，函数内执行任务对象
```

```
31              {
32                  std::unique_lock<std::mutex> lock(_mutex); // RAII生命周期结束自动释放锁// 3. 将构造出来的匿名函数对象抛入任务池中
33                  _taskpool.push_back( [task]() {(*task)();} ); // 这里其实就是tmp_func
34                  _cv.notify_one();
35              }
36
37              return fu;
38          }
39
40          void Stop(){
41              if(_stop == true) return;
42              _stop = true;
43              _cv.notify_all();
44              for(auto &thread : _threads) thread.join();
45          }
46
47      private:
48          // 线程入口函数 --- 内部不断地从任务池中拿任务进行执行void Entry(){
49              while(!_stop)
50              {
51                  // 临时变量并不涉及线程安全问题，无锁执行提高并发性能
52                  std::vector<Functor> tmp_taskpool;
53                  {
54                      // 加锁std::unique_lock<std::mutex> lock(_mutex);
55
56                      // 等待任务池不为空，或者_stop被置位返回
57                      _cv.wait(lock, [this](){ return _stop || !_taskpool.empty(); });
58
59                      // 取出任务进行执行（清空避免重复执行）
60                      tmp_taskpool.swap(_taskpool);
61                  }
62
63                  for(auto& task : tmp_taskpool) task();
64              }
65          }
66
67      private:
68          std::atomic<bool> _stop;
69          std::vector<Functor> _taskpool;
70          std::mutex _mutex;
71          std::condition_variable _cv;
72          std::vector<std::thread> _threads;
73      };
74  }
```

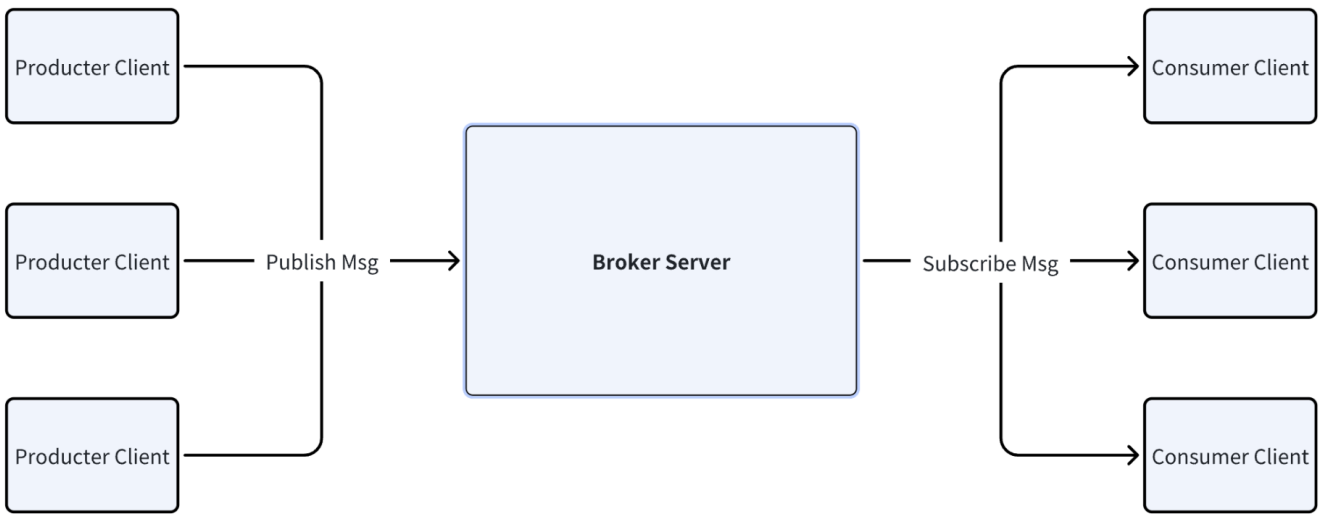
四. 项目需求分析

核心需求

- 生产者 (Producer)
- 消费者 (Consumer)
- 中间人 (Broker)
- 发布 (Publish)
- 订阅 (Subscribe)
- 一个生产者， 一个消费者



- N个生产者， N个消费者



其中， Broker Server 是最核心的部分, 负责消息的存储和转发。

而在 **AMQP**(Advanced Message Queuing Protocol-高级消息队列协议，一个提供统一消息服务的应用层标准高级消息队列协议，为面向消息的中间件设计，使得遵从该规范的客户端应用和消息中间件服务器的全功能互操作成为可能)模型中，也就是消息中间件服务器 Broker 中，又存在以下概念：

· 虚拟机 (**VirtualHost**): 类似于 **MySQL** 的 "**database**", 是一个逻辑上的集合。一个 **BrokerServer** 上可以存在多个 **VirtualHost** 。

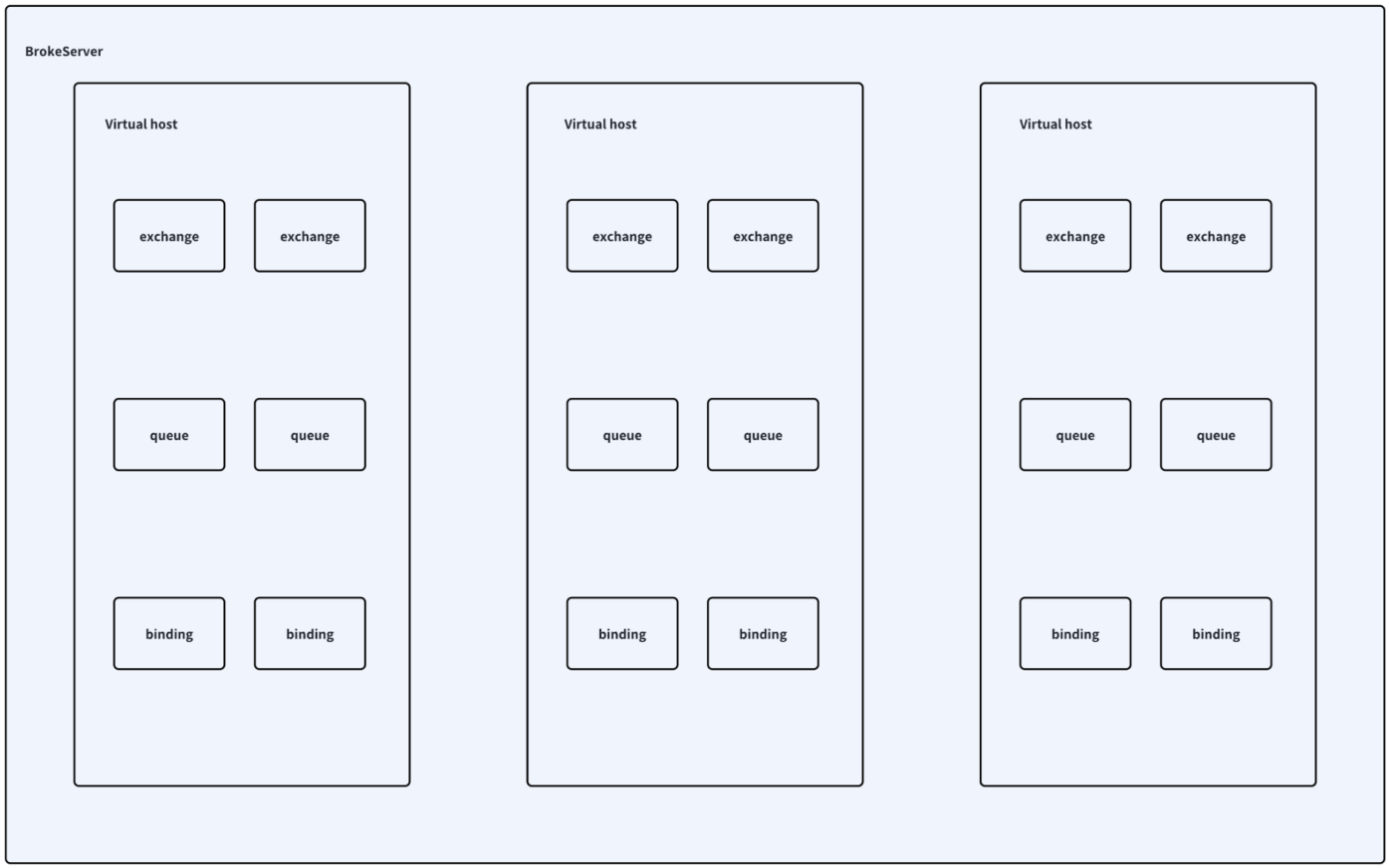
· 交换机 (**Exchange**): 生产者把消息先发送到 **Broker** 的 **Exchange** 上，再根据不同的规则, 把消息转发给不同的 **Queue**。

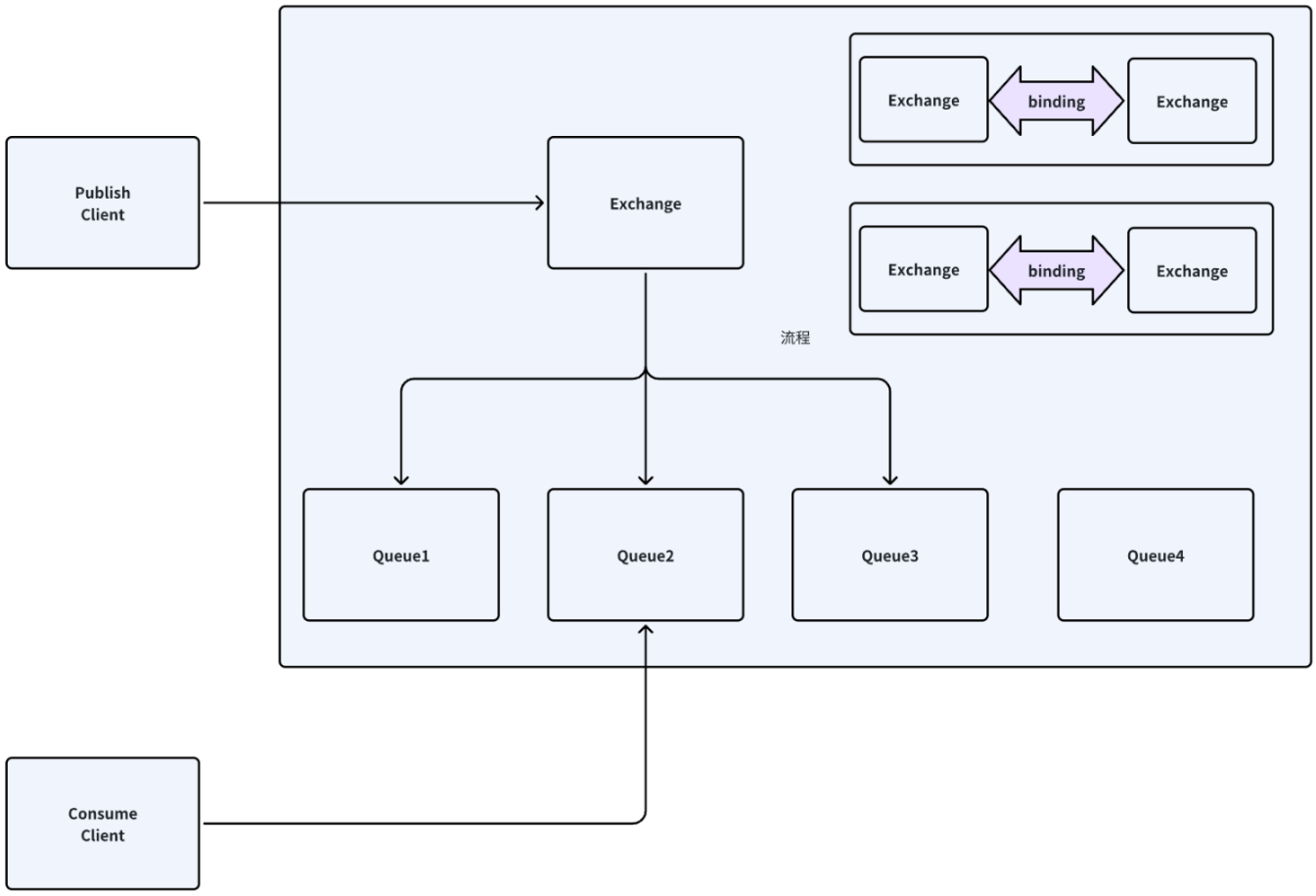
· 队列 (**Queue**): 真正用来存储消息的部分， 每个消费者决定自己从哪个 **Queue** 上读取消息。

· 绑定 (**Binding**): **Exchange** 和 **Queue** 之间的关联关系，**Exchange** 和 **Queue** 可以理解成 "多对多" 关系，使用一个关联表就可以把这两个概念联系起来。

· 消息 (Message): 传递的内容。

> 所谓的 **Exchange** 和 **Queue** 可以理解成 "多对多" 关系, 和数据库中的 "多对多" 一样， 意思是:
> 一个 **Exchange** 可以绑定多个 **Queue** (可以向多个 **Queue** 中转发消息)
> 一个 **Queue** 也可以被多个 **Exchange** 绑定 (一个 **Queue** 中的消息可以来自于多个 **Exchange**)





上述数据结构， 既需要在内存中存储， 也需要在硬盘中存储

- 内存存储： 方便使用
- 硬盘存储： 重启数据不丢失

核心API

对于 Broker 来说, 要实现以下核心 API，通过这些 API 来实现消息队列的基本功能。

3. 创建交换机 (exchangeDeclare)
4. 销毁交换机 (exchangeDelete)
5. 创建队列 (queueDeclare)
6. 销毁队列 (queueDelete)
7. 创建绑定 (queueBind)
8. 解除绑定 (queueUnbind)
9. 发布消息 (basicPublish)
10. 订阅消息 (basicConsume)
11. 确认消息 (basicAck)
12. 取消订阅 (basicCancel)

另一方面, Producer 和 Consumer 则通过网络的方式, 远程调用这些 API, 实现 **生产者消费者模型**

交换机类型

对于 RabbitMQ 来说, 主要支持四种交换机类型：

- Direct
- Fanout
- Topic
- Header

其中 Header 这种方式比较复杂, 比较少见。常用的是前三种交换机类型，项目中也主要实现三种

- Direct: 生产者发送消息时, 直接指定被该交换机绑定的队列名
- Fanout: 生产者发送的消息会被复制到该交换机的所有队列中
- Topic: 绑定队列到交换机上时, 指定一个字符串为 **bindingKey**。发送消息指定一个字符串为 **routingKey**。当 **routingKey** 和 **bindingKey** 满足一定的匹配条件的时候, 则把消息投递到指定队列。

> 这三种操作就像给 qq 群发红包。
>
> - Direct 是发一个专属红包, 只有指定的人能领。
> - Fanout 是使用了魔法, 发一个 10 块钱红包, 群里的每个人都能领 10 块钱。

> · Topic 是发一个画图红包, 发 10 块钱红包, 同时出个题, 得画的像的人, 才能领。也是每个领到的人都能领 10 块钱.

持久化

Exchange, **Queue**, **Binding**, **Message** 等数据都有持久化需求当程序重启 / 主机重启, 保证上述内容不丢失。

网络通信

生产者和消费者都是客户端程序, Broker 则是作为服务器，通过网络进行通信。

在网络通信的过程中, 客户端部分要提供对应的 api, 来实现对服务器的操作。

13. 创建 Connection
14. 关闭 Connection
15. 创建 Channel
16. 关闭 Channel
17. 创建队列 (queueDeclare)
18. 销毁队列 (queueDelete)
19. 创建交换机 (exchangeDeclare)
20. 销毁交换机 (exchangeDelete)
21. 创建绑定 (queueBind)
22. 解除绑定 (queueUnbind)
23. 发布消息 (basicPublish)
24. 订阅消息 (basicConsume)
25. 确认消息 (basicAck)
26. 取消订阅（basicCancel）

可以看到, 在 Broker 的基础上, 客户端还要增加 Connection 操作和 Channel 操作

· Connection 对应一个 TCP 连接

· Channel 则是 Connection 中的逻辑通道

一个 **Connection** 中可以包含多个 **Channel**。**Channel** 和 **Channel** 之间的数据是独立的，不会相互干扰。这样做主要是为了能够更好的复用 **TCP** 连接, 达到长连接的效果, 避免频繁的创建关闭 **TCP** 连接。

> Connection 可以理解成一根网线. Channel 则是网线里具体的线缆.

消息应答

被消费的消息, 需要进行应答。应答模式分成两种：

- 自动应答: 消费者只要消费了消息, 就算应答完毕了，Broker 直接删除这个消息。
- 手动应答: 消费者手动调用应答接口, Broker 收到应答请求之后, 才真正删除这个消息。

> 手动应答的目的, 是为了保证消息确实被消费者处理成功了. 在一些对于数据可靠性要求高的场景, 比较常见。

五. 模块划分

服务端模块

持久化数据管理中心模块

在数据管理模块中管理交换机，队列，队列绑定，消息等部分数据数据。

27. 交换机管理：
    a. 管理信息：名称，类型，是否持久化标志，是否(无人使用时)自动删除标志，其他参数....
    b. 管理操作：恢复历史信息，声明，删除，获取，判断是否存在。
28. 队列管理：
    a. 管理信息：名称，是否持久化标志，是否独有标志，是否(无人使用时)自动删除标志，其他参数....
    b. 管理操作：恢复历史信息，声明，删除，获取，判断是否存在。
29. 绑定管理：
    a. 管理信息：交换机名称，队列名称，绑定主题。
    b. 管理操作：恢复历史信息，绑定，解绑，解除交换机关联绑定信息，解除队列关联绑定信息，获取交换机关联绑定信息。
30. 消息管理：
    a. 管理信息
        i. 属性：消息 ID， 路由主题，持久化模式标志
        ii. 消息内容
        iii. 有效标志（持久化需要）
        iv. 持久化位置（内存中）
        v. 持久化消息长度（内存中）
    b. 管理操作：恢复历史信息，向指定队列新增消息，获取指定队列队首消息，确认移除消息

这几个核心概念数据都需要在内存和硬盘中存储的。

- 以内存存储为主，主要是保证快速查找信息进行处理。
- 以硬盘存储为辅，主要是保证服务器重启之后，之前的信息都可以正常保持。

虚拟机管理模块

因为交换机/队列/绑定都是基于虚拟机为单元整体进行操作的，因此虚拟机是对以上数据管理模块的整合模块。

31. 虚拟机管理信息：
    a. 交换机数据管理模块句柄
    b. 队列数据管理模块句柄
    c. 绑定数据管理模块句柄
    d. 消息数据管理模块句柄
32. 虚拟机对外操作：
    a. 提供虚拟机内交换机声明，交换机删除操作。
    b. 提供虚拟机内队列声明，队列删除操作。
    c. 提供虚拟机内交换机-队列绑定，解除绑定操作。
    d. 获取交换机相关绑定信息
33. 虚拟机管理操作：
    a. 创建虚拟机
    b. 查询虚拟机
        c. 删除虚拟机

交换机路由模块

当客户端发布一条消息到交换机后，这条消息，应该被入队到该交换机绑定的哪些队列中？交换路由模块就是决定这件事情的。

在绑定信息中有一个 binding_key，而每条发布的消息中有一个 routing_key，能否入队取决于两个要素：交换机类型和 key。

34. 广播：将消息入队到该交换机的所有绑定队列中。
35. 直接：将消息入队到绑定信息中 binding_key 与消息 routing_key 一致的队列中。
36. 主题：将消息入队到绑定信息中 binding_key 与 routing_key 是匹配成功的队列中。

binding_key

是由数字字母下划线构成的, 并且使用 . 分成若干部分。

例如：news.music.#，这用于表示交换机绑定的当前队列是一个用于发布音乐新闻的队列。

- 支持 * 和 # 两种通配符， 但是 * # 只能作为 . 切分出来的独立部分, 不能和其他数字字母混用,
    - 比如 a.*.b 是合法的， a.*a.b 是不合法的

- * 可以匹配任意一个单词（注意是单词不是字母）
- # 可以匹配任意零个或者多个单词（注意是单词不是字母）

routing_key

是由数据、字母和下划线构成， 并且可以使用 . 划分成若干部分。

例如：news.music.pop，这用于表示当前发布的消息是一个流行音乐的新闻。

消费者管理模块

消费者管理是以队列为单元的，因为每个消费者都会在开始的时候订阅一个队列的消息，当队列中有消息后，会将队列消息轮询推送给订阅了该队列的消费者。

因此操作流程通常是，从队列关联的消息管理中取出消息，从队列关联的消费者中取出一个消费者，然后将消息推送给消费者（这就是发布订阅中负载均衡的用法）。

37. 消费者信息：
    a. 标识
    b. 订阅队列名称
    c. 自动应答标志（决定了一条消息推送给消费者后，是否需要等待收到确认后再删除消息）
    d. 消息处理回调函数指针（一个消息发布后调用回调，选择消费者进行推送....）
        i. void(const std::string& tag, const BasicProperties& p, const std::string& body)
38. 消费者管理：添加，删除，轮询获取指定队列的消费者，移除队列所有消费者等操作。

信道管理模块

本质上，在 **AMQP** 模型中，除了通信连接 **Connection** 概念外，还有一个 **Channel** 的概念，**Channel** 是针对 **Connection** 连接的一个更细粒度的通信信道，多个 **Channel** 可以使用同一个通信连接 **Connection** 进行通信，但是同一个 **Connection** 的 **Channel** 之间相互独立。

而信道模块就是再次将上述模块进行整合提供服务的模块

39. 管理信息：
    a. 信道 ID
    b. 信道关联的消费者
    c. 信道关联的连接
    d. 信道关联的虚拟机
    e. 工作线程池（一条消息被发布到队列后，需要将消息推送给订阅了对应队列的消费者，过程由线程池完成）

40. 管理操作：

a. 提供声明&删除交换机操作（删除交换机的同时删除交换机关联的绑定信息）

b. 提供声明&删除队列操作（删除队列的同时，删除队列关联的绑定信息，消息，消费者信息）

c. 提供绑定&解绑队列操作

d. 提供订阅&取消订阅队列消息操作

e. 提供发布&确认消息操作

连接管理模块

本质上，仿照实现的服务器是通过 **muduo** 库来实现底层通信的，而这里的连接管理，更多的是对 **muduo** 库中的 **Connection** 进行二次封装管理，并额外提供项目所需操作。

41. 管理信息：

a. 连接关联的信道

b. 连接关联的 muduo 库 Connection

42. 管理操作：新增连接，删除连接，获取连接，打开信道，关闭信道。

Broker 服务器模块

整合以上所有模块，并搭建网络通信服务器，实现与客户端网络通信，能够识别客户端请求，并提供客户端请求的处理服务。

管理信息：

a. 虚拟机管理模块句柄

b. 消费者管理模块句柄

c. 连接管理模块句柄

d. 工作线程池句柄

e. muduo 库通信所需元素...

客户端模块

消费者管理模块

消费者在客户端的存在感比较低，因为在用户的使用角度中，只要创建一个信道后，就可以通过信道完成所有的操作，因此对于消费者的感官更多是在订阅的时候传入了一个消费者标识，且当前的简单实现也仅仅是一个信道只能创建订阅一个队列，也就是只能创建一个消费者，它们一一对应，因此更是弱化了消费者的存在。

43. 消费者信息：

a. 标识

b. 订阅队列名称

c. 自动应答标志（决定了一条消息推送给消费者后，是否需要等待收到确认后再删除消息）

d. 消息处理回调函数指针（一个消息发布后调用回调，选择消费者进行推送....）

44. 消费者管理：添加，删除，轮询获取指定队列的消费者，移除队列所有消费者等操作。

信道请求模块

与服务端的信道类似，客户端这边在 **AMQP** 模型中，也是除了通信连接 **Connection**概念外，还有一个 **Channel** 的概念，**Channel** 是针对 **Connection** 连接的一个更细粒度的通信信道，多个 **Channel** 可以使用同一个通信连接 **Connection** 进行通信，但是同一个 **Connection** 的 **Channel** 之间相互独立。

45. 信道管理信息：

a. 信道 ID

b. 信道关联的通信连接

c. 信道关联的消费者

d. 请求对应的响应信息队列（这里队列使用 hash 表，以便于查找指定的响应）

e. 互斥锁&条件变量（大部分的请求都是阻塞操作，发送请求后需要等到响应才能继续，但是 muduo 库的通信是异步的，因此需要我们自己在收到响应后，通过判断是否是等待的指定响应来进行同步）

46. 信道管理操作：

a. 提供创建信道操作

b. 提供删除信道操作

c. 提供声明交换机操作（强断言-有则 OK，没有则创建）

d. 提供删除交换机

e. 提供创建队列操作（强断言-有则 OK，没有则创建）

f. 提供删除队列操作

g. 提供交换机-队列绑定操作

h. 提供交换机-队列解除绑定操作

i. 提供添加订阅操作

j. 提供取消订阅操作

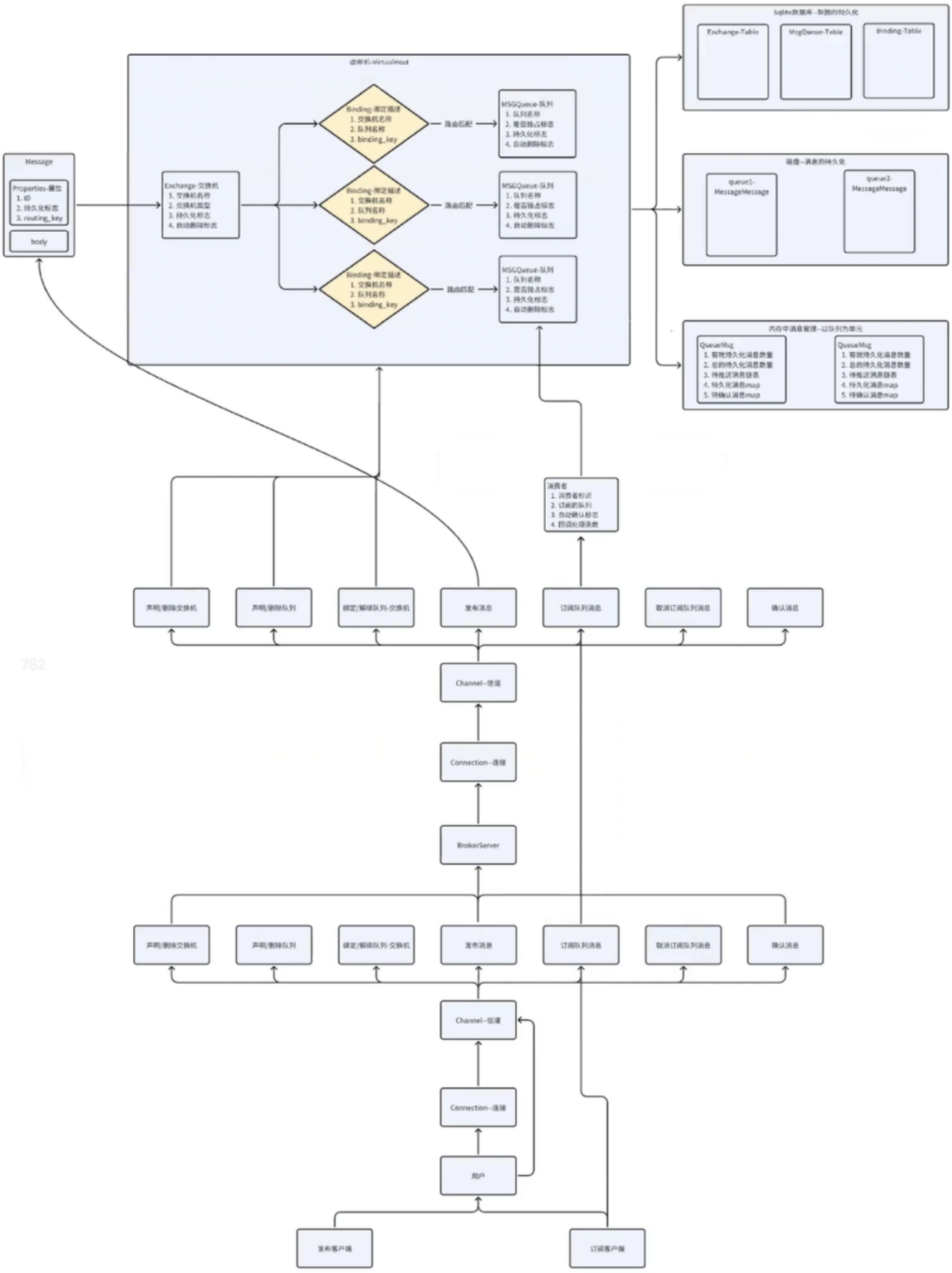
k. 提供发布消息操作

通信连接模块

向用户提供一个用于实现网络通信的 **Connection** 对象，从其内部可创建出粒度更轻的 **Channel** 对象，用于与服务端进行网络通信。

47. 管理信息：
    a. 连接关联的实际用于通信的 **muduo::net::Connection** 连接
    b. 连接关联的信管理句柄（实现信道的增删查）
    c. 连接关联的 **EventLoop** 异步循环工作线程
    d. 异步工作线程池（用于对收到服务器推送过来的消息进行处理的线程池）
48. 管理操作：
    a. 提供创建 **Channel** 信道的操作
    b. 提供删除 **Channel** 信道的操作

项目模块关系图



六. 部分模块实现

因为总代码量有点大，而且本文到此也有些长了，所以这个地方只会给一些功能类及函数的编写，

主体部分代码日后有机会的话，我会另写一文在总体介绍。

日志打印类

为了便于编写项目中能够快速定位程序的错误位置，因此编写一个日志打印类，进行简单的日志打印。

因为该项目是在 **Linux OS** 下实现的，所以日志类用的是我当初学习 **Linux** 时编写的一个日志类。该日志内用的锁是封装 **Linux** 锁的系统调用，所以这个日志没考虑跨平台的可靠性😅，有机会可以把这个锁换成 **C++** 风格的，但是这里我就不改了。

Mutex.hpp

采用 **RAII** 的思想，在封装系统调用的前提下，又实现了一个 **LockGuard** 类，保证锁的生命周期随作用域自动释放，避免死锁问题。

代码块

```
#pragma once
#include <iostream>
#include <pthread.h>
namespace MutexModule
{
    class Mutex
    {
    public:
        Mutex()
        {
            pthread_mutex_init(&_mutex, nullptr);
        }

        pthread_mutex_t* Get(){
            return &_mutex;
        }

        void Lock(){
            pthread_mutex_lock(&_mutex);
        }

        void Unlock(){
            pthread_mutex_unlock(&_mutex);
        }

        ~Mutex()
```

```
27          {
28              pthread_mutex_destroy(&_mutex);
29          }
30
31      private:
32          pthread_mutex_t _mutex;
33      };
34
35      class LockGuard
36      {
37          public:
38              LockGuard(Mutex& mutex) : _mutex(mutex)
39              {
40                  _mutex.Lock();
41              }
42
43              ~LockGuard()
44              {
45                  _mutex.Unlock();
46              }
47          private:
48              Mutex& _mutex;
49      };
50  }
```

Log.hpp

这个日志类通过**策略模式**有向文件输出和显示器打印两种功能，且支持日志文件自定位置与文件名，在选择上述两种功能之一时，需要在**main**函数中指定 **Enable_Console_Log_Strategy()** 或是 **Enable_File_Log_Strategy(path, file)** 。这里的文件宏中的两个参数是缺省的，如果不传的话默认在 **.hpp** 同路径下生成一个 **log.txt** 文件。

该日志在声明日志输出方式后，它在各个文件中的使用方式是

LOG(LogLevel:: 日志类型) << "你对该位置的描述";

[2025-11-19 22:10:01][DEBUG][3167485][Message.hpp][774]- 加载到无效消息：Hello World4

这就是一条输出日志的样子，包含时间、日志类型、进程 pid 、输出文件名、行数、描述。

```
代码块
```

```cpp
#pragma once#include <iostream>#include <string>#include <fstream>#include <filesystem>#include <pthread.h>#include <time.h>#include <sys/types.h>#include <unistd.h>#include "Mutex.hpp"namespace LogModule
{
    using namespace MutexModule;

    const std::string defaultpath = "./log";
    const std::string defaultfile = "my.log";

    const std::string& gsep = "\r\n";
    class LogStrategy
    {
    public:
        virtual void SyncLog(const std::string& message) = 0;
        ~LogStrategy() {}
    };

    class ConsoleLogStrategy : public LogStrategy
    {
    public:
        ConsoleLogStrategy() {}
        ~ConsoleLogStrategy() {}
        void SyncLog(const std::string& message) override{
            LockGuard lockguard(_mutex);
            std::cout << message << gsep;
        }
    private:
        Mutex _mutex;
    };

    class FileStrategy : public LogStrategy
    {
    public:
        FileStrategy(const std::string& path = defaultpath,
            const std::string& file = defaultfile)
            : _path(path), _file(file)
            {
                // 创建路径LockGuard lockguard(_mutex);
                if(std::filesystem::exists(_path)) return;
                else{
                    try{
                        std::filesystem::create_directories(_path);
                    }
                    catch(const std::filesystem::filesystem_error& e){
                        std::cerr << e.what() << '\n';
                    }
                }
```

```
46              }
47
48          void SyncLog(const std::string& message) override{
49              LockGuard lockguard(_mutex);
50              // 写入文件
51              std::string filename = _path + (_path.back() == '/' ? "" : "/") +
    _file;
52              std::ofstream out(filename, std::ios::app);
53
54              if(!out.is_open()){
55                  std::cerr << "Failed open file: " << filename << std::endl;
56              }
57
58              out << message << gsep;
59              out.close();
60          }
61
62          ~FileStrategy() {}
63      private:
64          Mutex _mutex;
65          const std::string _path;
66          const std::string _file;
67      };
68
69      enum class LogLevel{
70          DEBUG, INFO, WARNING, ERROR, FATAL
71      };
72
73      std::string LeveltoStr(LogLevel level){
74          switch (level)
75          {
76          case LogLevel::DEBUG:
77              return "DEBUG";
78          case LogLevel::INFO:
79              return "INFO";
80          case LogLevel::WARNING:
81              return "WARNING";
82          case LogLevel::ERROR:
83              return "ERROR";
84          case LogLevel::FATAL:
85              return "FATAL";
86          default:
87              return "UNKNOWN ID";
88          }
89      }
90
91      std::string GetTimeStamp(){
```

```
92          time_t curr = time(nullptr);
93          struct tm curr_tm;
94          localtime_r(&curr, &curr_tm);
95          char timebuffer[128];
96          snprintf(timebuffer, sizeof(timebuffer), "%4d-%02d-%02d
    %02d:%02d:%02d",
97              curr_tm.tm_year + 1900,
98              curr_tm.tm_mon + 1,
99              curr_tm.tm_mday,
100             curr_tm.tm_hour,
101             curr_tm.tm_min,
102             curr_tm.tm_sec
103         );
104
105         return (std::string)timebuffer;
106     }
107
108     // 1. 形成日志 && 2. 根据不同的策略，完成刷新class Logger
109     {
110     public:
111         Logger()
112         {
113             EnableConsoleLogStrategy();
114         }
115
116         void EnableFileLogStrategy(const std::string& path = defaultpath,
117             const std::string& file = defaultfile){
118             _fflush_strategy = std::make_unique<FileStrategy>(path, file);
119         }
120
121         void EnableConsoleLogStrategy(){
122             _fflush_strategy = std::make_unique<ConsoleLogStrategy>();
123         }
124
125         class LogMessage
126         {
127         public:
128             LogMessage(LogLevel level, std::string src_name,
129                 size_t line_number, Logger& logger)
130                 : _curr_time(GetTimeStamp()), _level(level), _pid(getpid()),
    _src_name(src_name)
131                 , _line_number(line_number), _logger(logger)
132             {
133                 std::stringstream ss;
134                 ss << "[" << _curr_time << "]"
135                 << "[" << LeveltoStr(_level) << "]"
136                 << "[" << _pid << "]"
```

```
                << "[" << _src_name << "]"
                << "[" << _line_number << "]"
                << "- ";
                _log_message = ss.str();
            }

            template <class T>
            LogMessage& operator<<(const T& info) // 日志的内容信息
            {
                std::stringstream ss;
                ss << info;
                _log_message += ss.str();
                return *this;
            }

            // 在用户输入 << 与内容后，决定输出策略
            ~LogMessage()
            {
                if(_logger._fflush_strategy)
                {
                    _logger._fflush_strategy->SyncLog(_log_message);
                }
            }

        private:
            std::string _curr_time;
            LogLevel _level;
            pid_t _pid;
            std::string _src_name;
            size_t _line_number;
            std::string _log_message;
            Logger& _logger;
        };

        LogMessage operator()(LogLevel level, std::string name, size_t line){
            return LogMessage(level, name, line, *this);
        }

        ~Logger() {}

    private:
        std::unique_ptr<LogStrategy> _fflush_strategy;
    };

    Logger logger;

```

```
183  #define LOG(level) logger(level, __FILE__, __LINE__)#define
     Enable_Console_Log_Strategy() logger.EnableConsoleLogStrategy()#define
     Enable_File_Log_Strategy(path, file) logger.EnableFileLogStrategy(path, file)
184  }
```

SQLite类

该类主要就是封装 **sqlite3** 在 **C++** 上的接口，更方便为我们的项目服务。

主要上就是两个函数，**Open** 与 **Exec** ，分别用于打开数据库与执行语句，因为我们要在后面实现消息的可持续化，需要把消息存入数据库，保证网络问题或是掉电数据仍旧存在 。

代码块

```
1   class SqliteHelper
2       {
3       public:
4           typedef int(*SqliteCallback)(void*, int, char**, char**);
5
6           SqliteHelper(const std::string& dbfile) : _dbfile(dbfile),
    _handler(nullptr) {}
7
8           bool Open(int safe_level = SQLITE_OPEN_FULLMUTEX){
9               int ret = sqlite3_open_v2(_dbfile.c_str(), &_handler
10                  , SQLITE_OPEN_CREATE | SQLITE_OPEN_READWRITE | safe_level,
    nullptr);
11              if(ret != SQLITE_OK)
12              {
13                  LOG(LogLevel::ERROR) << "打开sqlite数据库失败: " <<
    sqlite3_errmsg(_handler);
14                  return false;
15              }
16
17              return true;
18          }
19
20          bool Exec(const std::string& sql, SqliteCallback cb, void* arg){
21              int ret = sqlite3_exec(_handler, sql.c_str(), cb, arg, nullptr);
22              if(ret != SQLITE_OK)
23              {
24                  std::cout << sql << std::endl;
25                  LOG(LogLevel::ERROR) << "执行语句失败: " <<
    sqlite3_errmsg(_handler);
26                  return false;
```

```
            }

            return true;
        }

        void Close(){
            if(_handler) sqlite3_close_v2(_handler);
        }

    private:
        std::string _dbfile;
        sqlite3* _handler;
    };
```

Str类

这个类就是我们后面用来分割字符串的一个类。是为了从 **routing_key** 和 **binding_key** 取出特定部分匹配的。

没啥可讲的，算法就是一个双指针。

代码块

```
class StrHelper
    {
    public:
        static size_t Splite(const std::string& str, const std::string& sep, std::vector<std::string>& result)
        {
            result.clear();
            if (str.empty()) return 0;

            if (sep.empty())
            {
                result.push_back(str);
                return 1;
            }

            size_t start = 0;
            size_t end = 0;

            while ((end = str.find(sep, start)) != std::string::npos)
            {
                result.push_back(str.substr(start, end - start));
```

```
                start = end + sep.length();
            }

            result.push_back(str.substr(start));
            return result.size();
        }
    };
```

UUID生成类

这个类是生成UUid的，用于给我们的消息做唯一性区分。

在这里，**UUid** 生成，我们采用生成 8 个随机数字，加上 8 字节序号，共 16 字节数组生成 32 位 16 进制字符的组合形式来确保全局唯一的同时能够根据序号来分辨数据（随机数肉眼分辨起来真是太难了....），为了区分不同主机，还在最后添加主机的 **Mac** 地址。

代码块

```
class UUIDHelper
    {
    private:
        static std::string GetMacAddress(const std::string& interface = "eth0")

        {
            std::string path = "/sys/class/net/" + interface + "/address";
            std::ifstream file(path);

            if (file.is_open())
            {
                std::string mac;
                std::getline(file, mac);
                return mac;
            }

            return "";
        }

    public:
        static std::string Uuid(){
            std::random_device rd;
            std::mt19937_64 gernator(rd());
            std::uniform_int_distribution<int> distribution(0, 255);
```

```
24
25              std::stringstream ss;
26              for(int i = 0; i < 8; i++)
27              {
28                  ss << std::setw(2) << std::setfill('0') << std::hex <<
    distribution(gernator);
29                  if(i == 3 || i == 5 || i == 7) ss << "-";
30              }
31
32              ss << GetMacAddress() << "-";
33
34              static std::atomic<size_t> seq(1);
35              size_t num = seq.fetch_add(1);
36
37              for(int i = 7; i >= 0; i--)
38              {
39                  ss << std::setw(2) << std::setfill('0') << std::hex <<
    ((num>>i*8) & 0xff);
40                  if(i == 6) ss << "-";
41              }
42
43              return ss.str();
44          }
45      };
```

文件操作类

a. 文件是否存在判断

b. 文件大小获取

c. 文件读/写

d. 文件创建/删除

e. 目录创建/删除

代码块

```
1   class FileHelper
2       {
3       public:
4           FileHelper(const std::string& filename) :_filename(filename) {}
5
```

```
6           bool Exists()
7           {
8               struct stat st;
9               return (stat(_filename.c_str(), &st) == 0);
10          }
11
12           size_t Size()
13           {
14              struct stat st;
15              int ret = stat(_filename.c_str(), &st);
16
17              if (ret < 0)
18              {
19                  return 0;
20              }
21              return st.st_size;
22          }
23
24          bool Read(char* body, size_t offset, size_t len)
25          {
26              //1. 打开文件std::ifstream ifs(_filename, std::ios::binary |
    std::ios::in);
27              if (ifs.is_open() == false)
28              {
29                  LOG(LogLevel::ERROR) << "文件打开失败" << _filename.c_str();
30                  return false;
31              }
32
33              //2. 跳转文件读写位置
34              ifs.seekg(offset, std::ios::beg);
35
36              //3. 读取文件数据
37              ifs.read(body, len);
38              if (ifs.good() == false)
39              {
40                  LOG(LogLevel::ERROR) << "文件读取失败" << _filename.c_str();
41                  ifs.close();
42                  return false;
43              }
44
45              //4. 关闭文件
46              ifs.close();
47              return true;
48          }
49
50          bool Read(std::string& body)
51          {
```

```
52              // 获取文件大小，利用重载调整body空间size_t fsize = this->Size();
53
54              // 取首地址拿到char*，string的c_str返回值不能在此修改
55              body.resize(fsize);
56              return Read(&body[0], 0, fsize);
57          }
58
59          bool Write(const char* body, size_t offset, size_t len)
60          {
61              //1. 打开文件std::fstream fs(_filename, std::ios::binary |
    std::ios::in | std::ios::out);
62              if (fs.is_open() == false)
63              {
64                  LOG(LogLevel::ERROR) << "文件打开失败" << _filename.c_str();
65                  return false;
66              }
67
68              //2. 跳转到文件指定位置
69              fs.seekp(offset, std::ios::beg);
70
71              //3. 写入数据
72              fs.write(body, len);
73              if (fs.good() == false)
74              {
75                  LOG(LogLevel::ERROR) << "文件写入失败" << _filename.c_str();
76                  fs.close();
77                  return false;
78              }
79
80              //4. 关闭文件
81              fs.close();
82              return true;
83          }
84
85          bool Write(const std::string& body)
86          {
87              return Write(body.c_str(), 0, body.size());
88          }
89
90          static std::string ParentDirectory(const std::string& filename)
91          {
92              // /aaa/bb/ccc/ddd/test.txtsize_t pos = filename.find_last_of("/");
93              if (pos == std::string::npos)
94              {
95                  // test.txtreturn "./";
96              }
97
```

```
98              std::string path = filename.substr(0, pos);
99              return path;
100         }
101
102         bool Rename(const std::string& nname)
103         {
104             return (::rename(_filename.c_str(), nname.c_str()) == 0);
105         }
106
107         static bool CreateFile(const std::string& filename)
108         {
109             std::fstream ofs(filename, std::ios::binary | std::ios::out);
110             if (ofs.is_open() == false)
111             {
112                 LOG(LogLevel::ERROR) << "文件打开失败" << filename.c_str();
113                 return false;
114             }
115
116             ofs.close();
117             return true;
118         }
119
120         static bool RemoveFile(const std::string& filename)
121         {
122             return (::remove(filename.c_str()) == 0);
123         }
124
125         static bool CreateDirectory(const std::string& path)
126         {
127             //  aaa/bbb/ccc    cccc// 在多级路径创建中，我们需要从第一个父级目录开始
    创建size_t pos, idx = 0;
128             while (idx < path.size())
129             {
130                 pos = path.find("/", idx);
131                 if (pos == std::string::npos)
132                 {
133                     return (mkdir(path.c_str(), 0775) == 0);
134                 }
135
136                 std::string subpath = path.substr(0, pos);
137                 int ret = mkdir(subpath.c_str(), 0775);
138                 if (ret != 0 && errno != EEXIST)
139                 {
140                     LOG(LogLevel::ERROR) << "创建目录: " << subpath.c_str() <<
    "失败!";
141                     return false;
142                 }
```

```
143
144                 idx = pos + 1;
145             }
146
147             return true;
148         }
149
150         static bool RemoveDirectory(const std::string& path)
151         {
152             // rm -rf path// system()
153             std::string cmd = "rm -rf " + path;
154             return (system(cmd.c_str()) != -1);
155         }
156
157     private:
158         std::string _filename;
159     };
```

这就是该项目用到的所有工具类，有机会我会把项目正式功能代码模块再写一文，敬请期待。