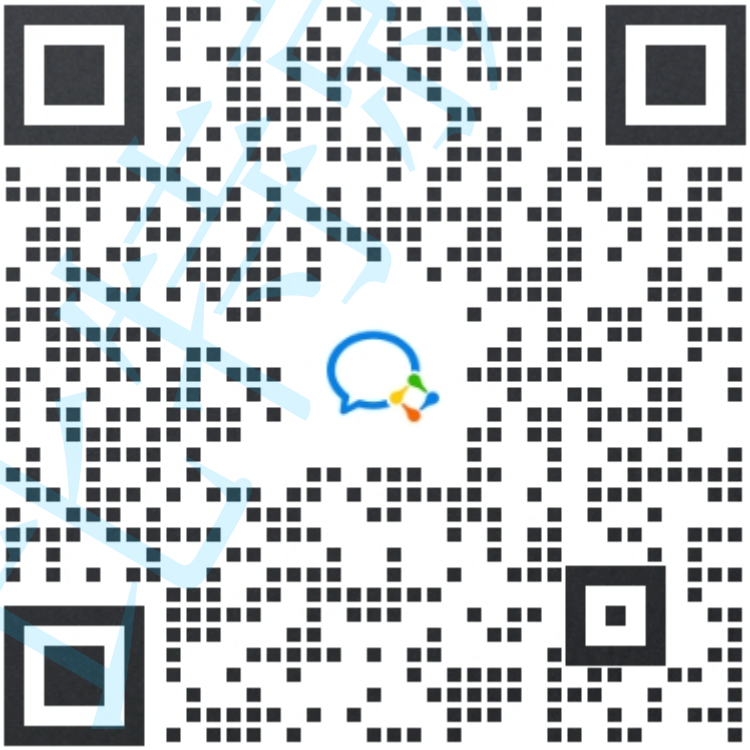
RabbitMq C++客户端的使用

版权说明

对比特项目感兴趣，可以联系这个微信。



代码 & 板书链接

https://gitee.com/bitedu-tech/cpp-chatsystem

1. 安装 RabbitMQ

1.1 安装教程

```Shell
sudo apt install rabbitmq-server
```

1.2 RabbitMQ 的简单使用

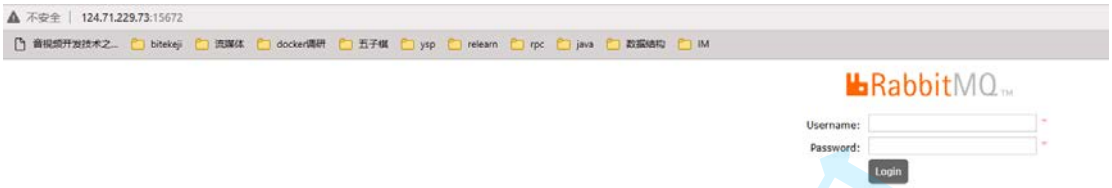
```Shell
 # 启动服务
sudo systemctl start rabbitmq-server.service
 # 查看服务状态
sudo systemctl status rabbitmq-server.service

# 安装完成的时候默认有个用户 guest ，但是权限不够，要创建一个 administrator 用户，才可以做为远程登录和发表订阅消息：
 #添加用户
sudo rabbitmqctl add_user root 123456
 #设置用户 tag
sudo rabbitmqctl set_user_tags root administrator
 #设置用户权限
sudo rabbitmqctl set_permissions -p / root "." "." ".*"

# RabbitMQ 自带了 web 管理界面,执行下面命令开启
sudo rabbitmq-plugins enable rabbitmq_management
```

查看 rabbitmq-server 的状态.

访问 webUI 界面， 默认端口为 15672.



至此 rabbitmq 安装成功。

2. 安装 RabbitMQ 客户端库

如果需要在其他应用程序中使用 RabbitMQ，则需要安装 RabbitMQ 的客户端库。

可以使用以下命令安装 librabbitmq 库：

```Shell
sudo apt-get install librabbitmq-dev
```

这将在系统上安装 RabbitMQ 的客户端库，包括头文件和静态库文件。

3. 安装 RabbitMQ 的 C++客户端库

- C 语言库：https://github.com/alanxz/rabbitmq-c
- C++库: https://github.com/CopernicaMarketingSoftware/AMQP-CPP/tree/master

我们这里使用 AMQP-CPP 库来编写客户端程序。

3.1 安装 AMQP-CPP

```Shell
sudo apt install libev-dev      #libev 网络库组件
git clone https://github.com/CopernicaMarketingSoftware/AMQP-
CPP.git
cd AMQP-CPP/
make
make install
```

```
root@hcss-ecs-2618:/home/zsc/AMQP-CPP# make install
mkdir -p /usr/include/amqpcpp
mkdir -p /usr/include/amqpcpp/linux_tcp
mkdir -p /usr/lib
cp -f include/amqpcpp.h /usr/include
cp -f include/amqpcpp/*.h /usr/include/amqpcpp
cp -f include/amqpcpp/linux_tcp/*.h /usr/include/amqpcpp/linux_tcp
cp -f src/libamqpcpp.so.4.3.26 /usr/lib
cp -f src/libamqpcpp.a.4.3.26 /usr/lib
ln -s -f libamqpcpp.so.4.3.26 /usr/lib/libamqpcpp.so.4.3
ln -s -f libamqpcpp.so.4.3.26 /usr/lib/libamqpcpp.so
ln -s -f libamqpcpp.a.4.3.26 /usr/lib/libamqpcpp.a
```

至此可以通过 AMQP-CPP 来操作 rabbitmq。

安装报错:

```
C++
/usr/include/openssl/macros.h:147:4: error: #error
"OPENSSL_API_COMPAT expresses an impossible API compatibility
level"
  147 | #  error "OPENSSL_API_COMPAT expresses an impossible API
compatibility level"
      |    ^~~~~
In file included from /usr/include/openssl/ssl.h:18,
                 from linux_tcp/openssl.h:20,
                 from linux_tcp/openssl.cpp:12:
/usr/include/openssl/bio.h:687:1: error: expected constructor,
destructor, or type conversion before ‘DEPRECATEDIN_1_1_0’
  687 | DEPRECATEDIN_1_1_0(int BIO_get_port(const char *str,
unsigned short *port_ptr))
```

这种错误，表示 ssl 版本出现问题。

解决方案：卸载当前的 ssl 库，重新进行修复安装

```
C++
dpkg -l |grep ssl
ii  erlang-ssl
ii  libevent-openssl-2.1-7:amd64
pi  libgnutls-openssl27:amd64
ii  libssl-dev:amd64
ii  libssl3:amd64
ii  libxmlsec1-openssl:amd64
ii  libzstd-dev:amd64
ii  libzstd1:amd64
ii  openssl
ii  python3-openssl
ii  zstd
```

```
sudo dpkg -P --force-all libevent-openssl-2.1-7
sudo dpkg -P --force-all openssl
sudo dpkg -P --force-all libssl-dev

sudo apt --fix-broken install
```

修复后，重新进行 make

ev 相关接口：

4. AMQP-CPP 库的简单使用

4.1 介绍

- AMQP-CPP 是用于与 RabbitMq 消息中间件通信的 c++库。它能解析从 RabbitMq 服务发送来的数据，也可以生成发向 RabbitMq 的数据包。AMQP-CPP 库不会向 RabbitMq 建立网络连接，所有的网络 io 由用户完成。
- 当然，AMQP-CPP 提供了可选的网络层接口，它预定义了 TCP 模块，用户就不用自己实现网络 io，我们也可以选择 libevent、libev、libuv、asio 等异步通信组件，需要手动安装对应的组件。
- AMQP-CPP 完全异步，没有阻塞式的系统调用，不使用线程就能够应用在高性能应用中。
- 注意：它需要 c++17 的支持。

4.2 使用

AMQP-CPP 的使用有两种模式：

- 使用默认的 TCP 模块进行网络通信
- 使用扩展的 libevent、libev、libuv、asio 异步通信组件进行通信

4.2.1 TCP 模式

- 实现一个类继承自 AMQP::TcpHandler 类， 它负责网络层的 TCP 连接
- 重写相关函数， 其中必须重写 monitor 函数
- 在 monitor 函数中需要实现的是将 fd 放入 eventloop(select、epoll)中监控， 当 fd 可写可读就绪之后， 调用 AMQP-CPP 的 connection->process(fd, flags)方法

```C++
#include <amqpcpp.h>
#include <amqpcpp/linux_tcp.h>

class MyTcpHandler : public AMQP::TcpHandler
{
    /**
        *AMQP 库在创建新连接时调用的方法
        *与处理程序相关联。这是对处理程序的第一次调用
        *@param connection 附加到处理程序的连接
    */
    virtual void onAttached(AMQP::TcpConnection *connection)
override
    {
        //@todo
        //添加您自己的实现，例如初始化事物
        //以处理连接。
    }

    /**
        *当 TCP 连接时由 AMQP 库调用的方法
        *已经建立。调用此方法后，库
        *仍然需要设置可选的 TLS 层和
        *在 TCP 层的顶部建立 AMQP 连接。，这种方法
        *总是与稍后对 onLost（）的调用配对。
        *@param connection 现在可以使用的连接
    */
    virtual void onConnected(AMQP::TcpConnection *connection)
override
    {
        //@todo
        //添加您自己的实现（可能不需要）
    }

    /**
        *在建立安全 TLS 连接时调用的方法。
        *这只对 amqps://连接调用。它允许您检查连接是否足够安全，以满足
您的喜好
        *（例如，您可以检查服务器证书）。AMQP 协议仍然需要启动。
        *@param connection 已被保护的连接
        *@param ssl 来自 openssl 库的 ssl 结构
        *@return bool 如果可以使用连接，则为 True
```

```
    */
    virtual bool onSecured(AMQP::TcpConnection *connection, const
SSL *ssl) override
    {
        //@todo
        //添加您自己的实现，例如读取证书并检查它是否确实是您的
        return true;
    }

    /**
        *当登录尝试成功时由 AMQP 库调用的方法。在此之后，连接就可以使用
了。
        *@param connection 现在可以使用的连接
    */
    virtual void onReady(AMQP::TcpConnection *connection) override
    {
        //@todo
        //添加您自己的实现，例如通过创建一个通道实例，然后开始发布或使用
    }

    /**
        *该方法在服务器尝试协商检测信号间隔时调用，
        *并被覆盖以摆脱默认实现（否决建议的检测信号间隔），转而接受该间
隔。
        *@param connection 发生错误的连接
        *@param interval 建议的间隔（秒）
    */
    virtual uint16_t onNegotiate(AMQP::TcpConnection *connection,
uint16_t interval)
    {
        //我们接受服务器的建议，但如果间隔小于一分钟，我们将使用一分钟的
间隔
        if (interval < 60) interval = 60;

        //@todo
        //在事件循环中设置一个计时器，
        //如果在这段时间内没有发送其他指令，
        //请确保每隔 interval 秒调用 connection->heartbeat（）。
        //返回我们要使用的间隔
        return interval;
    }

    /**
```

```
        *发生致命错误时由 AMQP 库调用的方法
        例如，因为无法识别从 RabbitMQ 接收的数据，或者基础连接丢失。
        此调用之后通常会调用 onLost（）（如果错误发生在 TCP 连接建立之后）和 onDetached（）。
        *@param connection 发生错误的连接
        *@param message 一条人类可读的错误消息
    */
    virtual void onError(AMQP::TcpConnection *connection, const char *message) override
    {
        //@todo
        //添加您自己的实现，例如，通过向程序的用户报告错误并记录错误
    }

    /**
        *该方法在 AMQP 协议结束时调用的方法。这是调用 connection.close（）以正常关闭连接的计数器部分。请注意，TCP 连接此时仍处于活动状态，您还将收到对 onLost（）和 onDetached（）的调用
        @param connection AMQP 协议结束的连接
    */
    virtual void onClosed(AMQP::TcpConnection *connection) override
    {
        //@todo
        //添加您自己的实现，可能没有必要，
        //但如果您想在 amqp 连接结束后立即执行某些操作，
        //又不想等待 tcp 连接关闭，则这可能会很有用
    }
    /**
        *当 TCP 连接关闭或丢失时调用的方法。
        *如果同时调用了 onConnected（），则始终调用此方法
        *@param connection 已关闭但现在无法使用的连接
    */
    virtual void onLost(AMQP::TcpConnection *connection) override
    {
        //@todo
        //添加您自己的实现（可能没有必要）
    }

    /**
        *调用的最终方法。这表示将不再对处理程序进行有关连接的进一步调用。
```

```
         *@param connection 可以被破坏的连接
     */
    virtual void onDetached(AMQP::TcpConnection *connection)
override
    {
        //@todo
        //添加您自己的实现，如清理资源或退出应用程序
    }

   /**
    *当 AMQP-CPP 库想要与主事件循环交互时，它会调用该方法。
    *AMQP-CPP 库是完全不阻塞的，
    *并且只有在事先知道这些调用不会阻塞时才进行“write()”或“read()”系统
调用。
    *要在事件循环中注册文件描述符，它会调用这个“monitor()”方法，
    *该方法带有一个文件描述符和指示是否该检查文件描述符的可读性或可写性
的标志。
    *
    *@param connection 想要与事件循环交互的连接
    *@param fd 应该检查的文件描述符
    *@param 标记位或 AMQP:: 可读和/或 AMQP:: 可写
    */
    virtual void monitor(AMQP::TcpConnection *connection, int fd,
int flags) override
    {
        //@todo
        //添加您自己的实现，
        //例如将文件描述符添加到主应用程序事件循环(如 select()或 poll()
循环)。
        //当事件循环报告描述符变为可读和或可写时，
        //由您通过调用 connection->process(fd，flags)方法
        //通知 AMQP-CPP 库文件描述符处于活动状态。
    }
};
```

4.2.2 扩展模式

以 libev 为例， 我们不必要自己实现 monitor 函数， 可以直接使用 `AMQP::LibEvHandler`

4.3 常用类与接口介绍

4.3.1 Channel

channel 是一个虚拟连接，一个连接上可以建立多个通道。并且所有的 RabbitMq 指令都是通过 channel 传输，所以连接建立后的第一步，就是建立 channel。因为所有操作是异步的，所以在 channel 上执行指令的返回值并不能作为操作执行结果，实际上它返回的是 Deferred 类，可以使用它安装处理函数。

```
C++
namespace AMQP {
/**
 *  Generic callbacks that are used by many deferred objects
 */
using SuccessCallback   =   std::function<void()>;
using ErrorCallback     =   std::function<void(const char
*message)>;
using FinalizeCallback  =   std::function<void()>;
/**
 *  Declaring and deleting a queue
 */
using QueueCallback     =   std::function<void(const std::string
&name,
         uint32_t messagecount, uint32_t consumercount)>;
using DeleteCallback    =   std::function<void(
        uint32_t deletedmessages)>;

using MessageCallback   =   std::function<void(
    const Message &message,
    uint64_t deliveryTag,
    bool redelivered)>;

//当使用发布者确认时，当服务器确认消息已被接收和处理时，将调用
AckCallback
using AckCallback       =   std::function<void(
    uint64_t deliveryTag,
    bool multiple)>;
//使用确认包裹通道时，当消息被 ack/nacked 时，会调用这些回调
using PublishAckCallback       =   std::function<void()>;
using PublishNackCallback      =   std::function<void()>;
using PublishLostCallback      =   std::function<void()>;

class Channel {
    Channel(Connection *connection);
```

```
    bool connected()
    /**
        *声明交换机
        *如果提供了一个空名称，则服务器将分配一个名称。
        *以下 flags 可用于交换机：
        *
        *-durable      持久化，重启后交换机依然有效
        *-autodelete   删除所有连接的队列后，自动删除交换
        *-passive      仅被动检查交换机是否存在
        *-internal     创建内部交换
        *
        *@param name     交换机的名称
        *@param-type     交换类型
            enum ExchangeType
            {
                fanout,   广播交换，绑定的队列都能拿到消息
                direct,   直接交换，只将消息交给 routingkey 一致的队列
                topic,    主题交换，将消息交给符合 bindingkey 规则的队
列
                headers,
                consistent_hash,
                message_deduplication
            };
        *@param flags     交换机标志
        *@param arguments 其他参数
        *
        *此函数返回一个延迟处理程序。可以安装回调
        using onSuccess(), onError() and onFinalize() methods.
    */
    Deferred &declareExchange(
        const std::string_view &name,
        ExchangeType type,
        int flags,
        const Table &arguments)
    /**
        *声明队列
        *如果不提供名称，服务器将分配一个名称。
        *flags 可以是以下值的组合：
        *
        *-durable 持久队列在代理重新启动后仍然有效
        *-autodelete 当所有连接的使用者都离开时，自动删除队列
        *-passive 仅被动检查队列是否存在
```

```
    *-exclusive 队列仅存在于此连接，并且在连接断开时自动删除
    *
    *@param name         队列的名称
    *@param flags        标志组合
    *@param arguments    可选参数
    *
    *此函数返回一个延迟处理程序。可以安装回调
    *使用 onSuccess（）、onError（）和 onFinalize（）方法。
    *
    Deferred &onError(const char *message)
    *
    *可以安装的 onSuccess（）回调应该具有以下签名：
    void myCallback(const std::string &name,
        uint32_t messageCount,
        uint32_t consumerCount);
    例如：
    channel.declareQueue("myqueue").onSuccess(
        [](const std::string &name,
            uint32_t messageCount,
            uint32_t consumerCount) {
               std::cout << "Queue '" << name << "' ";
               std::cout << "has been declared with ";
               std::cout << messageCount;
               std::cout << " messages and ";
               std::cout << consumerCount;
               std::cout << " consumers" << std::endl;
 *  });
*/
DeferredQueue &declareQueue(
    const std::string_view &name,
    int flags,
    const Table &arguments)
/**
    *将队列绑定到交换机
    *
    *@param exchange     源交换机
    *@param queue        目标队列
    *@param routingkey   路由密钥
    *@param arguments    其他绑定参数
    *
    *此函数返回一个延迟处理程序。可以安装回调
    *使用 onSuccess（）、onError（）和 onFinalize（）方法。
*/
```

```
    Deferred &bindQueue(
        const std::string_view &exchange,
        const std::string_view &queue,
        const std::string_view &routingkey,
        const Table &arguments)
    /**
        *将消息发布到 exchange
        *您必须提供交换机的名称和路由密钥。
        然后，RabbitMQ 将尝试将消息发送到一个或多个队列。
        使用可选的 flags 参数，可以指定如果消息无法路由到队列时应该发生的情况。
        默认情况下，不可更改的消息将被静默地丢弃。
        *
        *如果设置了'mandatory'或'immediate'标志，
        则无法处理的消息将返回到应用程序。
        在开始发布之前，请确保您已经调用了 recall（）-方法，
        并设置了所有适当的处理程序来处理这些返回的消息。
        *
        *可以提供以下 flags：
        *
        *-mandatory 如果设置，服务器将返回未发送到队列的消息
        *-immediate 如果设置，服务器将返回无法立即转发给使用者的消息。

        *@param exchange 要发布到的交易所
        *@param routingkey 路由密钥
        *@param envelope 要发送的完整信封
        *@param message 要发送的消息
        *@param size 消息的大小
        *@param flags 可选标志
    */
    bool publish(
        const std::string_view &exchange,
        const std::string_view &routingKey,
        const std::string &message,
        int flags = 0)

    /**
        *告诉 RabbitMQ 服务器我们已准备好使用消息-也就是订阅队列消息
        *
        *调用此方法后，RabbitMQ 开始向客户端应用程序传递消息。
        consumer tag 是一个字符串标识符，
        如果您以后想通过 channel:: cancel（）调用停止它，
```

```
        可以使用它来标识使用者。
        *如果您没有指定使用者 tag，服务器将为您分配一个。
        *
        *支持以下 flags：
        *
        *-nolocal     如果设置了，则不会同时消耗在此通道上发布的消息
        *-noack       如果设置了，则不必对已消费的消息进行确认
        *-exclusive   请求独占访问，只有此使用者可以访问队列
        *
        *@param queue     您要使用的队列
        *@param tag       将与此消费操作关联的消费者标记
        *@param flags     其他标记
        *@param arguments 其他参数
        *
        *此函数返回一个延迟处理程序。
        可以使用 onSuccess（）、onError（）和 onFinalize（）方法安装回调。

        可以安装的 onSuccess（）回调应该具有以下格式：
            void myCallback（const std:: string_view&tag）;
        样例：
        channel.consume("myqueue").onSuccess(
            [](const std::string_view& tag) {
                std::cout << "Started consuming under tag ";
                std::cout << tag << std::endl;
        });
    */
    DeferredConsumer &consume(
        const std::string_view &queue,
        const std::string_view &tag,
        int flags,
        const Table &arguments)
    /**
        *确认接收到的消息
        *
        *当在 DeferredConsumer:: onReceived（）方法中接收到消息时，
        必须确认该消息，
        以便 RabbitMQ 将其从队列中删除（除非使用 noack 选项消费）。
        *
        *支持以下标志：
        *
        *-多条确认多条消息：之前传递的所有未确认消息也会得到确认
```

```
         *
         *@param deliveryTag     消息的唯一 delivery 标签
         *@param flags           可选标志
         *@return bool
     */
    bool ack(uint64_t deliveryTag, int flags=0)
}

class DeferredConsumer {
    /*
        注册一个回调函数，该函数在消费者启动时被调用。
        void onSuccess(const std::string &consumertag)
    */
    DeferredConsumer &onSuccess(const ConsumeCallback& callback)
    /*
        注册回调函数，用于接收到一个完整消息的时候被调用
        void MessageCallback(const AMQP::Message &message,
            uint64_t deliveryTag, bool redelivered)
    */
    DeferredConsumer &onReceived(const MessageCallback& callback)
    /* Alias for onReceived() */
    DeferredConsumer &onMessage(const MessageCallback& callback)

    /*
        注册要在服务器取消消费者时调用的函数
        void CancelCallback(const std::string &tag)
    */
    DeferredConsumer &onCancelled(const CancelCallback& callback)
}


class Message : public Envelope{
    const std::string &exchange()
    const std::string &routingkey(): q
}
class Envelope : public MetaData{
    const char *body()
    uint64_t bodySize()
}
}
```

4.3.2 ev

```C++
typedef struct ev_async
{
  EV_WATCHER (ev_async)

  EV_ATOMIC_T sent; /* private */
} ev_async;

//break type
enum {
  EVBREAK_CANCEL = 0, /* undo unloop */
  EVBREAK_ONE    = 1, /* unloop once */
  EVBREAK_ALL    = 2  /* unloop all loops */
};

struct ev_loop *ev_default_loop (unsigned int flags EV_CPP (= 0))
# define EV_DEFAULT  ev_default_loop (0)

int  ev_run (struct ev_loop *loop);
/* break out of the loop */
void ev_break (struct ev_loop *loop, int32_t break_type) ;

void (*callback)(struct ev_loop *loop, ev_async *watcher, int32_t
revents)

void ev_async_init(ev_async *w, callback cb);
void ev_async_start(struct ev_loop *loop, ev_async *w) ;
void ev_async_send(struct ev_loop *loop, ev_async *w) ;
```

第三方库链接:

```C++
g++ -o example example.cpp  -lamqpcpp -lev
```

5. 二次封装思想：

在项目中使用 rabbitmq 的时候，我们目前只需要交换机与队列的直接交换，实现一台主机将消息发布给另一台主机进行处理的功能，因此在这里可以对 mq 的操作进行简单的封装，使 mq 的操作在项目中更加简便:

封装一个 MQClient:

- 提供声明指定交换机与队列，并进行绑定的功能;
- 提供向指定交换机发布消息的功能

- 提供订阅指定队列消息，并设置回调函数进行消息消费处理的功能