一、 Producer/Comsumer：同步API接口

头文件：/home/w/include/kmq/api.h

Consumer

```
class Comsumer {
public:
virtual ~Comsumer() {}
virtual int Connect(const string &appname, const string &apphost) = 0;
virtual int Close() = 0;
virtual int Recv(string &msg, string &rt) = 0;
virtual int Send(const string &msg, const string &rt) = 0;
virtual int Send(const char *data, uint32_t len, const string &rt) = 0;
virtual int SetOption(int opt, ...) = 0;
virtual int GetOption(int opt, ...) = 0;
};
Comsumer *NewComsumer();
```

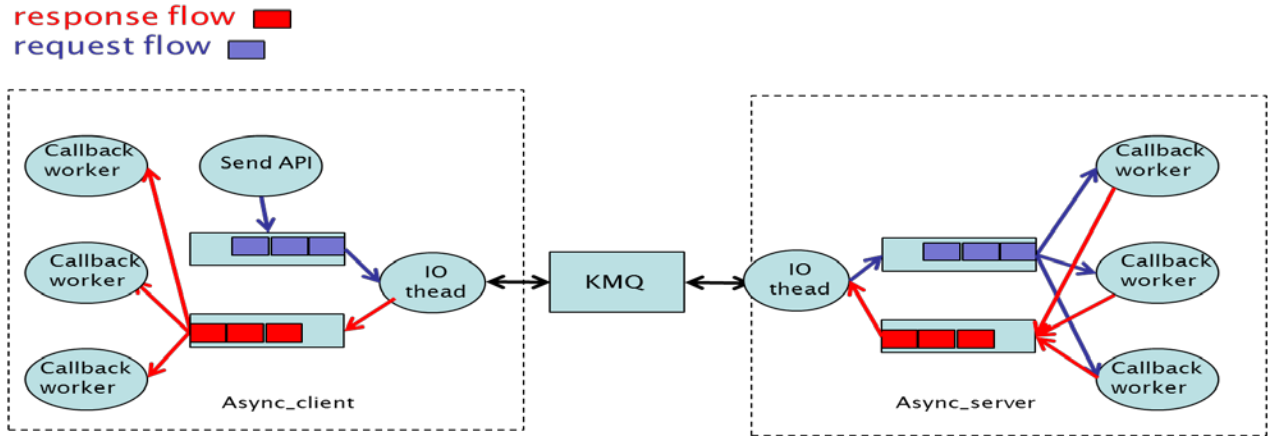
producer

```
class Producer {
public:
virtual ~Producer() {}
virtual int Connect(const string &appname, const string &apphost) = 0;
virtual int Close() = 0;
virtual int Send(const string &msg) = 0;
virtual int Send(const char *data, uint32_t len) = 0;
virtual int Recv(string &msg) = 0;
virtual int SetOption(int opt, ...) = 0;
virtual int GetOption(int opt, ...) = 0;
};
```

二、 AsyncComsumer/AsyncProducer ：异步API接口

头文件 /home/w/include/kmq/async_api.h

1）异步api的线程模型



2）异步api的配置

struct async_conf是异步api初始化setup时需要准备的参数，结构如下：

async_conf

```
class async_conf {
public:
async_conf() {
max_workers = queue_cap = max_trip_time = 0;
}
void set(string &app, string &host, int w, int cap, int rtt) {
appname = app;
apphost = host;
max_workers = w;
queue_cap = cap;
max_trip_time = rtt;
}
string appname, apphost;
int max_workers, queue_cap, max_trip_time;
};
```

注释：

1. appname表示应用所属的group名字，apphost表示连接到kmq的主机地址
2. max_workers 后台处理callback的线程数，默认值1
   a. 对于async_client，max_workers表示多线程处理响应队列里的数据包
   b. 对于async_server，max_workers表示多线程处理请求队列里的数据包
3. queue_cap 表示内部队列的大小，默认值20000
   a. 对于async_client
      i. 发送数据包时如果队列满，返回-1， errno设置为KMQ_EQUEUEFULL
      ii.io线程在接收响应时，如果响应队列满，则丢弃响应队列中等待时间最长的数据包
   b. 对于async_server
      i. io线程发现请求队列满，丢弃请求队列中等待时间最长的数据包，并构造响应的kmq_icmp报文返回，async_client收到kmq_icmp报文
      ii.callback线程在SendResponse时如果队列满，返回-1，errno设置为KMQ_EQUEUEFULL
4. max_trip_time，默认值为0
   a. 对于async_client，该值没有意义，async_client发送数据包时可以独立指定数据包的超时时间，超时后，async_client触发HandleError for TIMEOUT
   b. 对于async_server，该值仅表示数据包从client端到server端之间最大的传输时间，当io线程收到超过此值的数据包时，丢弃，并构造响应的kmq_
   c. 为0表示不在api端判断数据包的传输时间

3）异步api的错误处理

数据包在投递过程中可能会因为某个节点队列满了，或者数据包超时而被丢弃，此时相应的节点会构造kmq_icmp报文返回，async_client收到报文. 又或者数据包不明原因，既没有正常的消息返回，又没有kmq_icmp控制报文返回，此时会同样会触发HandleError函数，报告数据包TIMEOUT。

Error

```
class Error {
public:
virtual ~Error() {}
virtual string Str() = 0;
};
```

目前Error并没有暴露太多的接口，声明为一个虚基，并提供Str函数，将错误信息以字符串形式打印出来，所以通常async_client的Handle在实现HandleError函数。

```
int MyResponseHandler::HandleError(Error &ed, void *pridata) {
LOG_ERROR("request deliver with error {%s}", ed.Str().c_str());
return 0;
}
```

Error.Str()的结构通常有两种：

1. {seqid:307 is timeout of 2675ms}
2. {seqid:159 ttl:3 errno:211 msg_time:116ms icmp_time:53ms [go:0ms to 127.0.0.1:1510 cost:3ms stay:2ms] [go:5ms to 127.0.0.1:1520 cost:2ms stay:3ms] [go:10ms to 127.0.0.1:49406 cost:0ms stay:0ms]}

对于第一种类型的消息，只会出现在以下情况：

1. 数据包被丢弃
2. kmq_icmp包被丢弃，async_client在指定的max_trip_time内没有收到相应的kmq_icmp包

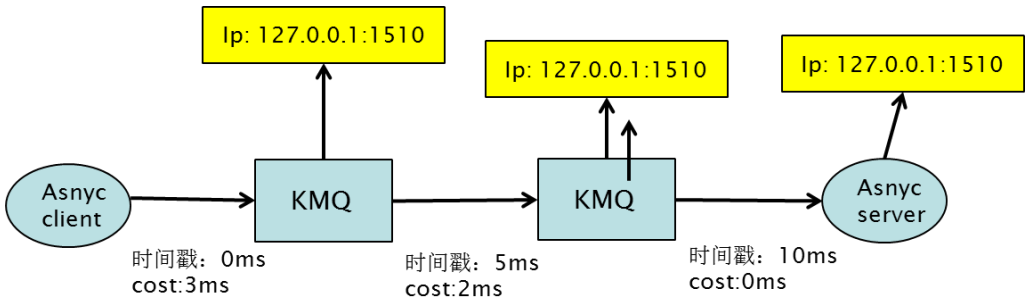
对于第二种消息，出现在：

1. 数据包被丢弃
2. async_client在max_trip_time内收到对应的kmq_icmp包

注释：

1. seqid：表示数据包的序列id，意义不大
2. ttl：表示数据包在传输过程中经过的节点数
3. errno：数据包被丢弃的原因
4. msg_time：数据包被丢弃时，在网络上的总传输时间
5. icmp_time：数据包被丢弃后，对应的kmq_icmp报文在网络上返回时的总传输时间
6. [go:xx to xx cost:xx stay:xx] 是一个元组，每个元组对应通信网络中的一个节点，它的意义如下：
   a. go 表示从当前节点出发的时间，go的时间是以数据包进网通信网络的时间为基准的一个相对时间
   b. to 表示目的地，下一个节点
   c. cost 表示从当前节点传输到下一个节点的时间消耗
   d. stay 表示到达下一个节点时停留的时间

以 {seqid:159 ttl:3 errno:211 msg_time:116ms icmp_time:53ms [go:0ms to 127.0.0.1:1510 cost:3ms stay:2ms] [go:5ms to 127.0.0.1:1520 cost:2ms stay:3ms] [go:10ms to 127.0.0.1:49406 cost:0ms stay:0ms]} 为例，如下图示数据包在网络上传输的整个过程：



4）AsyncProducer接口

用于业务层的client端，属于消息的生产者

AsyncProducer

```
class ResponseHandler {
public:
virtual ~ResponseHandler() {}
virtual int HandleError(Error &ed, void *pridata) = 0;
virtual int HandleResponse(const char *data, uint32_t len, void *pridata) = 0;

};
```

```
class AsyncProducer {
public:
virtual ~AsyncProducer() {}
virtual int Setup(async_conf &conf, ResponseHandler *h) = 0;
virtual int Stop() = 0;
virtual int StartServe() = 0;
virtual int SendRequest(const char *data, int len, void *pridata, int to_msec = 0) = 0;
};
AsyncProducer *NewAsyncProducer();
```

注释：

1. ResponseHandler
   是响应数据包的处理函数，也就是上图中的callback_worker线程的执行体，callback_worker从响应队列里读取响应数据包，然后触发此Handler
2. AsyncProducer::StartServe
   启动后台的io线程，启动后台的callback线程，此时用户可以在自己的线程里执行线程安全的SendRequest操作
3. AsyncProducer::Stop 停止所有后台的io线程，callback线程（Stop之后任何线程都不应该再调用SendRequest发送数据包）

5）AsyncComsumer

用于业务层的server端，表示消息的消费者

AsyncComsumer

```
class ResponseWriter {
public:
int Send(const char *data, uint32_t len);
string route;
AsyncComsumer *async_comsumer;
};
class RequestHandler {
public:
virtual ~RequestHandler() {}
virtual int HandleRequest(const char *data, uint32_t len, ResponseWriter &rw) = 0;
};
class AsyncComsumer {
public:
virtual ~AsyncComsumer() {}
virtual int Setup(async_conf &conf, RequestHandler *h) = 0;
virtual int Stop() = 0;
virtual int StartServe() = 0;
// apiRequestHandlerResponseWriter::Send
virtual int SendResponse(const char *data, int len, const string &rt) = 0;
};
AsyncComsumer *NewAsyncComsumer();
```

注释：

1. RequestHandler
   为请求数据包的处理函数，也就是上图中的callback_worker线程的执行体，callback_worker从请求队列里读取数据包，然后触发此Handler
2. StartServe 启动后台的io线程，启动后台的callback线程
3. Stop 停止所有后台的io线程，callback线程

三、 流水线编程接口

multiio = async_comsumer + N * async_producer，假定：
async_comsumer负责接收来自app1网络的请求并可通过 N * async_producer 向 app2,app3,... 应用发出子请求multiio的应用场景是：请求依赖（一个请求依赖于多个子请求的完成）
因此，multiio的目的，就是要简化在复杂的多应用网络环境下的混合编程，让开发人员更加专注于业务本身：消息的处理。而不需要过多关心消息在网络中的传递。

mutiio

```
class reqresp_ctx {
public:
virtual ~reqresp_ctx();
virtual int request_come(const char *data, uint32_t len, MultiSender *s) = 0;
virtual int back_response(ResponseWriter &w, bool bad) = 0;
virtual int one_response_bad(string who, Error &ed) = 0;
virtual int one_response_done(string who, const char *data, uint32_t len) = 0;
};
class reqresp_ctxfactor {
public:
virtual ~reqresp_ctxfactor() {}
// requestreqresp_ctx
virtual reqresp_ctx *new_reqresp_ctx(const char *req, uint32_t len) = 0;
};
class Multiio {
public:
Multiio();
~Multiio();
int Init(reqresp_ctxfactor *f, int nthreads, async_conf &inapp);
int AddBackendServer(async_conf &outapp);
int Start();
int Stop();
private:
};
```

注释：

1. reqresp_ctx是一个请求响应上下文，当有一个请求到达时，它自动被创建，当所有子请求完成时，它自动被销毁。reqresp_ctx实质上定义了一种契约：
   a. request_come，表示接受到来自app1网络的请求
   b. back_response，表示将结果返回给app1网络
   c. one_response_bad，表示发送app2，app3，app4,...的某个子请求失败
   d. one_response_done，表示发送给app2，app3，app4... 的某个子请求成功
2. reqresp_ctxfactor 开发者需要实现自己的reqresp_ctx，通过reqresp_ctxfactor，kmqapi才能知道收到新请求时如何创建reqresp_ctx
3. Multiio负责描述业务网络：
   a. Init()需要3个参数，其中nthreads是指后台callback线程的数量，而inapp是async_comsumer的配置，该async_comsumer负责接受来自app1应用网络的请求
   b. AddBackendServer()函数用来添加async_producer