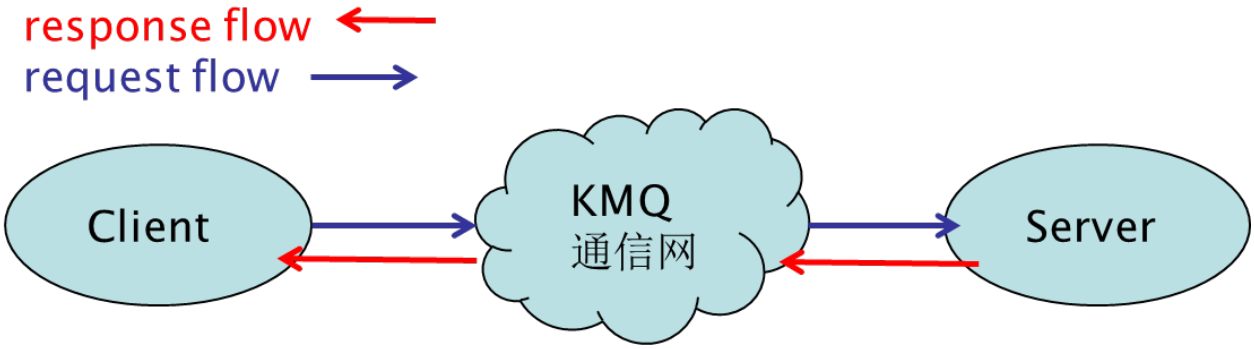
一、 请求相应模型



以下示例代码在目录 /home/w/share/kmq/example/ckmqapi/ 下

同步模式下的client端

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <kmq/compat_api.h>
static string appname = "testapp";
static string apphost = "127.0.0.1:1510";
int main(int argc, char **argv) {
CKmqApi client;
string msg("i am client"), resp;
int ret = 0;
client.init(appname);
client.join_client(apphost);
while (1) {
sleep(1);
// Read request msg
if ((ret = client.send(msg, 5000)) == 0 && (ret = client.recv(resp, 5000)) == 0)
fprintf(stdout, "client send: %s\n", msg.c_str());
else
fprintf(stderr, "client send with errno %d\n", errno);
}
return 0;
}
```

同步模式下的server端

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <kmq/compat_api.h>
using namespace kmq;
static string appname = "testapp";
static string apphost = "127.0.0.1:1520";
int main(int argc, char **argv) {
int ret = 0;
string msg;
CKmqApi server;
server.init(appname);
server.join_server(apphost);
while (1) {
sleep(1);
// Read request msg
```

```
if ((ret = server.recv(msg, 5000)) == 0 && (ret = server.send(msg, 5000)) == 0)
fprintf(stdout, "server recv %s\n", msg.c_str());
else
fprintf(stderr, "server recv with errno %d\n", errno);
}
return 0;
}
```

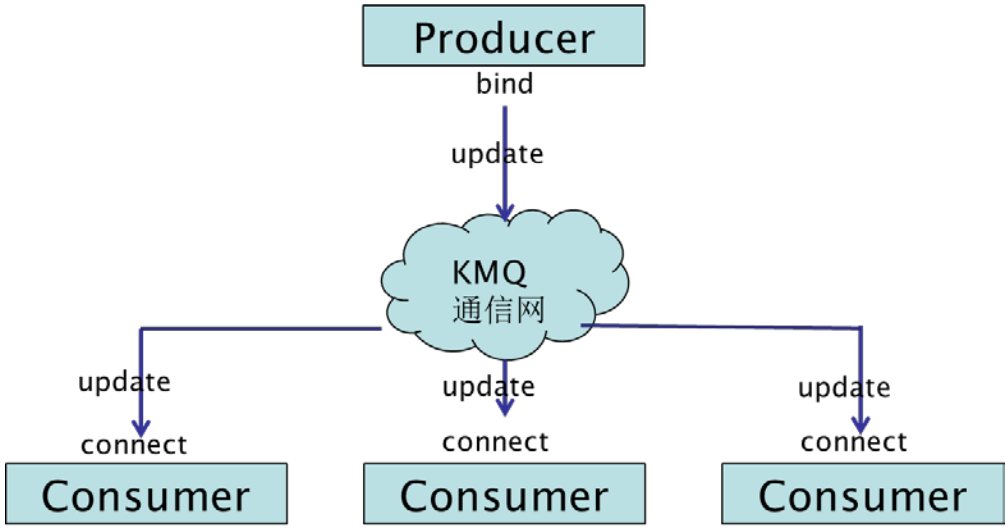
注释：

1. CKmqApi::init 不会失败
2. CKmqApi::join_xx 会一直尝试连接kmq服务器，直到连接成功，否则hang住

对于recv/send两个API，CKmqApi默认处理了网络连接错误，开发者不需要关心，例如：

1. 如果send失败，返回-1，并设置errno，CKmqApi内部会自动处理错误，例如出错的原因与是否需要重连kmq服务器等等
2. 如果recv失败，返回-1，并设置errno，同1
3. 对于client，CKmqApi能够保证每次recv到的响应必定对应于上一次send的请求

二、 发布订阅模型示例



场景说明：

producer_client: 注册callback处理response，异步发送request，在callback处理response时，
检查是否request == response
consmer_server: 注册callback处理request，对于接受到的所有request，不做任何修改，将request返回（亦即response == request）
下面示例代码在 /home/w/share/kmq/example/async_api/ 目录里：

1）消息生产者端示例

首先定义一个ResponseHandler，用来处理每个请求返回的结果，在这里我们只是检查request是否等于response，如下：

Producer端句柄的实现

```
class EchoResponseHandler : public ResponseHandler {
public:
int HandleError(Error &ed, void *req);
int HandleResponse(const char *data, uint32_t len, void *req);
private:
};
int EchoResponseHandler::HandleError(Error &ed, void *req) {
cout << "request deliver with error " << ed.Str() << endl;
free(req);
return 0;
}
int EchoResponseHandler::HandleResponse(const char *data, uint32_t len, void *req) {
if (memcmp(data, (char *)req, len) != 0)
cout << "recv response with error request != response" << endl;
free(req);
return 0;
}
```

注释：

在main函数里，我们启动一个AsyncProducer，将上面的ResponseHandler设置进去，然后不断的SendRequest操作，当有请求返回时，会自动触发ResponseHandler

producer_client

```
int main(int argc, char **argv) {
char *req = NULL;
string msg("i am async client");
EchoResponseHandler erh;
async_conf conf;
AsyncProducer *asp = NewAsyncProducer();
conf.appname = appname;
conf.apphost = apphost;
asp->Setup(conf, &erh);
asp->StartServe();
while (1) {
req = strndup(msg.data(), msg.size());
if (asp->SendRequest(msg.data(), msg.size(), req, 2 /* 2ms timeout */) < 0)
cout << "async client send request with errno " << errno << endl;
else
cout << "async client send request {" << msg << "}" << endl;
sleep(1);
}
asp->Stop();
delete asp;
return 0;
}
```

2）消息消费者端示例

服务端的逻辑很简单，收到什么请求，直接将请求返回，同样，首先我们需要实现一个RequestHandler，如下：

Consumer端句柄的实现

```
class EchoRequestHandler : public RequestHandler {
public:
int HandleRequest(const char *data, uint32_t len, ResponseWriter &rw);
};
int EchoRequestHandler::HandleRequest(const char *data, uint32_t len, ResponseWriter
&rw) {
string msg(data, len);
rw.Send(data, len);
cout << "async server recv request {" << msg << "}" << endl;
return 0;
}
```

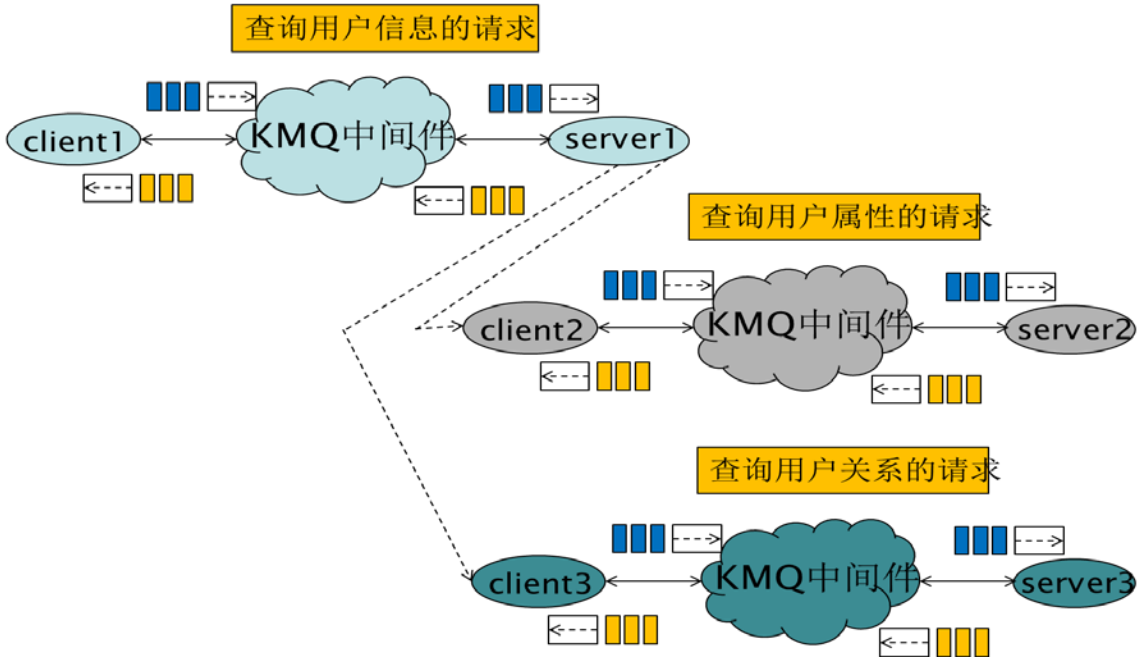
注释：

RequestHandler实际上是一个callback，当每个请求到达时，都会触发HandleRequest方法，在main函数里，我们创建一个AsyncProducer，再将Request发送出去。

consmer_server

```
int main(int argc, char **argv) {
EchoRequestHandler erh;
async_conf conf;
AsyncComsumer *asc = NewAsyncComsumer();
conf.appname = appname;
conf.apphost = apphost;
asc->Setup(conf, &erh);
asc->StartServe();
while (1) {
sleep(1);
}
asc->Stop();
delete asc;
return 0;
}
```

三、 流水线编程示例



一个查询用户信息的服务，涉及多种应用

网络1承载前端服务通信
- Client1代码：/home/w/share/kmq/example/async_pipeline/client.cc
- Server1代码：/home/w/share/kmq/example/async_pipeline/front_server.cc

网络2承载用户基本属性服务通信
- Client2代码：/home/w/share/kmq/example/async_pipeline/front_server.cc
- Server2代码：/home/w/share/kmq/example/async_pipeline/uattr_server.cc

网络3承载用户关系服务通信
- Client3代码：/home/w/share/kmq/example/async_pipeline/front_server.cc
- Server3代码：/home/w/share/kmq/example/async_pipeline/urelation_server.cc

场景角色的职责说明：
- client1向server1发送请求

server1过滤掉一部分非法请求，并查询server2和server3，得到用户的信息，聚合处理，返回给client1
- server2负责处理基本属性查询请求
- server3负责处理用户关系查询请求

场景假设，为了简单期间：
1. client1只会发出请求 {username:sina}
2. server2对于任意请求，均返回 " mock_age:18 mock_sex:female "，伪造的年龄和性别
3. server3对于任意请求，均返回 " mock_relationship:nothing "，不存在关系
4. server1拿到server2/server3返回的结果，合并字符

因此client/uattr_server/urelation_server的逻辑很简单，先简单show一下各自的实现，为清晰显示处理逻辑，忽略部分代码，详细可参考源文件。

1）client实现

client的逻辑很简单，每秒发送一个请求，拿到返回的结果，打印出来，如下：

home/w/share/kmq/example/async_pipeline/client.cc

```
class UserInfo_ResponseHandler : public ResponseHandler {
public:
int HandleError(Error &ed, void *cb);
int HandleResponse(const char *data, uint32_t len, void *cb);
};
int UserInfo_ResponseHandler::HandleError(Error &ed, void *cb) {
char *req = (char *)cb;
cout << "query with error " << ed.Str() << endl;
free(req);
return 0;
}
int UserInfo_ResponseHandler::HandleResponse(const char *data, uint32_t len, void *cb)
{
char *req = (char *)cb;
string response(data, len);
cout << "query result {" << response << "}" << endl;
free(req);
return 0;
}
int main(int argc, char **argv) {
/* ... */
while (1) {
```

```
cb = strndup(msg.data(), msg.size());
if (asp->SendRequest(msg.data(), msg.size(), cb, 2 /* 2ms timeout */) < 0)
cout << "query with errno " << errno << endl;
else
cout << "query {" << msg << "}" << endl;
sleep(1);
}
/* .... */
}
```

2）uattr_server实现

/home/w/share/kmq/example/async_pipeline/uattr_server.cc

```
/* ... UattrServer */
int UserAttr_RequestHandler::HandleRequest(const char *data, uint32_t len,
ResponseWriter &rw) {
string msg(data, len);
string response;
response = queryDB(msg);
rw.Send(response.data(), response.size());
cout << "uattr query req {" << msg << "}" << endl;
return 0;
}
/* ...main*/
```

3）urelation_server实现

/home/w/share/kmq/example/async_pipeline/urelation_server.cc

```
/* ... RequestHandler */
int UserRelation_RequestHandler::HandleRequest(const char *data, uint32_t len,
ResponseWriter &rw)
{
string msg(data, len);
string response;
response = queryDB(msg);
rw.Send(response.data(), response.size());
cout << "urelation query req {" << msg << "}" << endl;
return 0;
}
```

4）front_server实现

重点在这里，由于front_server在收到client1的请求后，需要到uattr_server/urelation_server查询相应的信息，此种场景非常适合使用流水线模型编程。

```
class query_result : public reqresp_ctx {
public:
query_result() : db(NULL)
{}
~query_result() {}
void init(mysql_db *db_engine) {
db = db_engine;
}
// client
int request_come(const char *data, uint32_t len) {
user.assign(data, len);
if (db->query_exist(user)) {
// uattr
sndr->Send(uattrhost, data, len, this, 10);
```

```
// urelation
sndr->Send(urelationhost, data, len, this, 10);
}
return 0;
}
// uattr_serverurelation_serveror
int back_response(ResponseWriter &w, bool bad) {
if (!bad)
w.Send(response.data(), response.size());
return 0;
}
// uattr_server or urelation_server
int one_response_bad(string who, Error &ed) {
return 0;
}
// uattr_server or urelation_server
int one_response_done(string who, const char *data, uint32_t len) {
response.append(data, len);
return 0;
}
private:
mysql_db *db;
string user, response;
};
```

query_result是一个reqresp_ctx，所以我们还需要一个定义一个reqresp_ctxfactor，让api知道如何创建这个实际的reqresp_ctx(query_result)

```
class ms_reqresp_ctxfactor : public reqresp_ctxfactor {
public:
reqresp_ctx *new_reqresp_ctx(const char *req, uint32_t len) {
query_result *qr = new (std::nothrow) query_result();
qr->init(&db_engine);
return qr;
}
private:
mysql_db db_engine;
};
```

最后，在main函数里，我们将整个流水线启动起来：

```
int main(int argc, char **argv) {
Multiio mio;
ms_reqresp_ctxfactor f;
async_conf inapp, outapp;
// multiiofront
inapp.set(frontname, fronthost, 1, 10, 10);
mio.Init(&f, 2, inapp);
// multiiouattr
outapp.set(uattrname, uattrhost, 1, 10, 10);
mio.AddBackendServer(outapp);
// multiiourelation
outapp.set(urelationname, urelationhost, 1, 10, 10);
mio.AddBackendServer(outapp);
mio.Start();
while (1)
sleep(1);
mio.Stop();
return 0;
}
```