TNODE 接口

一、应用场景

1. 同一进程内，不同TNODE实例可依据不同配置连接不同的Rabbitmq 实例。

server-2

server-1

1. TNODE 发送到单个订阅者的事件通知消息
2. Q1 是consumer在rabbitmq上的队列名，对应consumer eventname,队列属性AMQP::autodelete
3. Q1 绑定的 routingKey 名字同样是 Q1
4. TNODE-1 上发送消息接口参数eventname 对应 routingKey Q1。消息类型：TNODE\_EVENT\_MSG\_TYPE，消息头：struct TNodeCnEventHeader。 具体定义参见下面消息接口章节
5. TNODE 要求订阅者立即回应的请求消息
6. Q1 是consumer在rabbitmq上的队列名，对应consumer eventname,队列属性AMQP::autodelete
7. Q1 绑定的 routingKey 名字同样是 Q1
8. TNode 上channel 利用UUID 在 rabbitmq上生成唯一的队列，队列属性AMQP::autodelete， routingKey 同队列名
9. TNODE-1 上请求接口中eventname 对应 routingKey Q1, 消息类型对应 TNODE\_SERVICE\_MSG\_TYPE，消息头：struct TNodeCnServiceHeader。 消息结构参见下面消息接口定义，注意消息中携带了UUID，对应消息结构中的 queuename
10. TNODE-2 consumer 收到消息后，通过回调函数立即回应应答消息。应答消息指定的队列为queuename。消息类型：TNODE\_EVENT\_MSG\_TYPE，消息头：struct TNodeCnEventHeader。
11. TNODE 发送范围受控的广播消息
12. TNODE 通过组来控制广播消息的发送范围
13. 需要接收相同广播消息的consumer 加入到同一个组
14. 组内的consumer 对应 rabbitmq 上的队列名的命名规则为： "groupname" + "." + "eventname".以Q1队列举例，其consumer在组A中，所以队列名为 A.Q1, 队列属性AMQP::autodelete
15. Q1 绑定的 routingKey 名字同样是 Q1队列名，但需要额外再绑定一个 routingKey，命名规则为: "groupname" + ".$" 。
16. Q2 绑定的 routingKey 名字同样是 Q2队列名，但需要额外再绑定一个 routingKey，命名规则为: "groupname" + ".$" 。
17. Q3 绑定的 routingKey 名字同样是 Q3 队列名，但需要额外再绑定一个 routingKey，命名规则为: "groupname" + ".$" 。
18. TNODE-1 channel上PublishGroup接口中groupname 指定为A, eventname 指定为$, 对应 rabbitmq routingKey 为 A.$, 消息类型对应 TNODE\_GROUP\_MSG\_TYPE，消息头：struct TNodeCnEventHeader。 消息结构参见下面消息接口定义
19. TNODE 发送消息到组内的单一consumer
20. TNODE-1 channel上PublishGroup接口中groupname 指定为A, eventname 指定为Q2, 对应 rabbitmq routingKey 为 A.Q2, 消息类型对应 TNODE\_GROUP\_MSG\_TYPE，消息头：struct TNodeCnEventHeader。 消息结构参见下面消息接口定义
21. 其它部分同场景 “TNODE 发送范围受控的广播消息”
22. 多个TNODE consumer的负载均衡

message2

messge1,message2

message1

1. rabbitmq 天然支持，只需要两个TNODE 实例中的consumer订阅同一eventname，对应rabbitmq同一队列。
2. 消息类型：TNODE\_EVENT\_MSG\_TYPE，消息头：struct TNodeCnEventHeader。
3. 跨 TNODE组消息和 组内consumer的负载均衡

messge1,message2

messge1,message2

messge1,message2

messge1,message2

messge2

messge1

1. rabbitmq 天然支持，只需要两个TNODE 实例中保持相同的组名，有订阅同一 eventname的consumer.
2. 消息类型对应 TNODE\_GROUP\_MSG\_TYPE，消息头：struct TNodeCnEventHeader

二、消息接口

enum TNodeChannelMsgType {

TNODE\_EVENT\_MSG\_TYPE = 0,

TNODE\_SERVICE\_MSG\_TYPE,

TNODE\_GROUP\_MSG\_TYPE

};

struct TNodeCnEventHeader {

uint8 flag; //校验位，固定为 0x8F

uint8 msg\_type; //消息类型，enum TNodeChannelMsgType 枚举值

uint16 tag; //用户附加参数，TNODE负责透传

uint32 msg\_len; //消息体长度，不包含消息头

uint32 pad; //补码，固定为0

};

struct TNodeCnServiceHeader {

uint8 flag; //校验位，固定为 0x8F

uint8 msg\_type; //消息类型，enum TNodeChannelMsgType 枚举值

uint16 tag; //用户附加参数，TNODE负责透传

uint32 queue\_name\_len; //发送端UUID 长度，C字符串，包含结束符"\0"

uint32 msg\_len; //消息体长度，不包含消息头

};

TNODE\_EVENT\_MSG\_TYPE 消息格式：

msg body

pad

msg\_len

tag

msg\_type

flag

TNODE\_SERVICE\_MSG\_TYPE 消息格式：

queue\_name

msg body

msg\_len

queue\_name\_len

tag

msg\_type

flag

queue\_name 是C字符串，包含结束符

TNODE\_GROUP\_MSG\_TYPE 消息格式 与 TNODE\_EVENT\_MSG\_TYPE 消息格式 相同

三、API接口

class TNode {

public:

TNode();

~TNode();

virtual void Config(TNodeConfig &cfg); //配置TNODE 参数

virtual void Bind(uint32 id); //指定ID生成一个TNodeRabbit实 例，若ID 对应的TNodeRabbit实例已存 在，则该TNODE interface指向同一个 TNodeRabbit实例

virtual void unBind(uint32 id); //释放TNodeRabbit实例，TNodeRabbit 实例有引用计数，最后一个unBind将释 放TNodeRabbit实例(目前暂未做引用计 数)

virtual void Run(); //run TNodeRabbit实例，若TNodeRabbit 实例已经 run, 该函数直接返回

virtual void Stop(); //停止TNodeRabbit实例，谨慎使用，将 影响其他引用TNodeRabbit实例的 TNODE

virtual bool Check(); //检查TNode是否绑定了有效 TNodeRabbit实例

virtual bool IsReady(); //TNodeRabbit实例是否就绪

virtual TNodeChannel GetChannel(); //获得TNodeChannel interface

virtual TNodeChannel GetChannel(uint32 channel\_id); //指定ID获得TNodeChannel interface，ID 相同的 TNodeChannel interface 将 指向同一个 TNodeRabbitChannel实例

private:

uint32 id\_;

RabbitConfig\* config\_;

TNodeRabbit\* rabbit\_;

};

class TNodeChannel {

public:

TNodeChannel(TNodeRabbitChannel\* rabbitChannel);

~TNodeChannel();

void SetServiceHandle(TNodeHandle \*response\_handle); //设置处理对象，处 理RequestService 调用后， 接收到的应答消息。若重复 设置，将替换前一个处理对 象。不设置，将不能处理 RequestService应答消息

void ClearServiceHandle(); //清除处理对象，注意，清除后将不能处理 RequestService应答消息，RequestService应答消息 将被丢弃

char\* AllocDataBuff(uint32 data\_len); //申请数据BUFFER保存待发送数据，调用 PublishEvent、RequestService、PublishGroup 后将自动释放，若没有调用PublishEvent、 RequestService、PublishGroup，必须调用 FreeDataBuff释放BUFFER，否则有内存泄漏

void FreeDataBuff(char\* data); //释放通过AllocDataBuff申请的数据BUFFER， 注意，调用过PublishEvent、RequestService、 PublishGroup后，不需要调用FreeDataBuff释 放数据BUFFER，数据BUFFER将自动释放

//推送事件消息，eventname 为事件名，data为通过AllocDataBuff申请的，保存待 发送数据的BUFFER，datalen 为数据BUFFER长度， tag 透传用户参数

void PublishEvent(std::string &eventname, char \*data, uint32 datalen, uint16 tag);

//发送请求服务消息，eventname 为事件名，data为通过AllocDataBuff申请的，保 存待发送数据的BUFFER，datalen 为数据BUFFER长度，tag 透传用户参数。回应 的应答消息将通过SetServiceHandle 设置的对象处理。

void RequestService(std::string &eventname, char \*data, uint32 datalen, uint16 tag);

//往组里推送事件消息，eventname 为事件名，data为通过AllocDataBuff申请的， 保存待发送数据的BUFFER，datalen 为数据BUFFER长度，tag 透传用户参数。

eventname 为$ ,则往组里推送广播消息。eventname 非$,则发送到组内的某一个事 件consumer.

void PublishGroup(std::string &groupname, std::string &eventname, char \*data, uint32 datalen, uint16 tag);

//在chinnel上创建一个组，注意 #,\* 为TNODE保留字，请暂时不要使用#,\*命名 组名

void CreateGroup(std::string &groupname);

//在chinnel上删除一个组，组内的consumer将不能收到发往这个组的消息

void DeleteGroup(std::string &groupname);

//往组内添加一个consumer，consumer为继承TNodeConsumer接口的class的一 个实例，添加后，该实例将可以收到组内的消息。 注意 #,\* 为TNODE保留字， 请暂时不要使用#,\*命名事件

void AddConsumerToGroup(std::string &groupname, TNodeConsumer \*consumer);

//删除组内一个consumer，consumer为继承TNodeConsumer接口的class的一 个实例，删除后，该实例将不能收到组内的消息

void DeleteConsumerFromGroup(std::string &groupname, TNodeConsumer \*consumer);

//往chnanel上添加一个consumer，consumer为继承TNodeConsumer接口的class 的一个实例，添加后，该实例将可以收到chnanel上的消息。注意 #,\* 为TNODE 保留字，请暂时不要使用#,\*命名事件

void AddConsumer(TNodeConsumer \*consumer);

//在chnanel上删除一个consumer，consumer为继承TNodeConsumer接口的class 的一个实例，删除后，该实例将不能收到消息

void DeleteConsumer(TNodeConsumer \*consumer);

注意：同一个consumer既添加到chennel上，同时又添加到一个组内，将同时收到来自chennel和组内的消息，需要保证是线程安全的

using TNodeDataSend = std::function<void(uint16\_t tag, char \*data, uint32\_t size)>;

//consumer interface

class TNodeConsumer {

public:

virtual std::string GetEventName() = 0; //获取订阅的事件名

//处理收到的事件消息，tag为透传用户的参数

virtual uint32 EventHandle(uint16 tag, char \*event, uint32 eventlen) = 0;

//处理收到的请求服务消息，处理消息后，需要调用send 回调函数回复应答

//注意回复应答时使用的数据buff需要是同channel 的AllocDataBuff函数分配的

//数据Buffer，tag为透传用户的参数

virtual uint32 ServiceHandle(uint16 tag, char \*reqbody, uint32 reqlen, TNodeDataSend send) = 0;

};

class TNodeHandle {

public:

//处理收到的服务应答消息，tag为透传用户的参数

virtual uint32 Handle(uint16 tag, char \*data, uint32 len) = 0;

};

三、USE CASE

1. 对应应用场景1) USE CASE:

void UnitTest::TestCase1(){

TNodeConfigV0 cfg1;

cfg1.VHost("/demo")

.Exchange("t\_node")

.ExchangeGroup("t\_node\_grp")

.Address("192.168.10.125")

.Port("5672")

.User("demo")

.Password("demo");

TNode tn1;

tn1.Config(cfg1);

tn1.Bind(1);

tn1.Run();

TNodeConfigV0 cfg2;

cfg2.VHost("/demo")

.Exchange("t\_node")

.ExchangeGroup("t\_node\_grp")

.Address("192.168.10.126")

.Port("5672")

.User("demo")

.Password("demo");

TNode tn2;

tn2.Config(cfg2);

tn2.Bind(2);

tn2.Run();

}

1. 对应应用场景2) USE CASE:

void TestCase\_Consumer(){

std::string test\_event = "testEvent";

TestConsumer\* consumer;

std::string msg = "this is test msg";

char \* buff;

TNodeChannel channel = pnode->GetChannel();

consumer = new TestConsumer(test\_event);

channel.AddConsumer(consumer);

sleep(1);

buff = channel.AllocDataBuff((uint32)msg.length() + 1);

memcpy(buff, msg.c\_str(), (uint32)(msg.length() + 1));

channel.PublishEvent(test\_event, buff, (uint32)(msg.length() + 1), 50);

}

1. 对应应用场景3) USE CASE:

void TestCase\_Service(){

std::string test\_service = "TestServiceEvent";

TestConsumer\* consumer;

std::string msg = "this is test service msg";

char \* buff;

TNodeChannel channel = pnode->GetChannel();

consumer = new TestConsumer(test\_service);

channel.AddConsumer(consumer);

TestService\* service = new TestService();

channel.SetServiceHandle(service);

sleep(1);

while (1){

buff = channel.AllocDataBuff((uint32)msg.length() + 1);

memcpy(buff, msg.c\_str(), (uint32)(msg.length() + 1));

channel.RequestService(test\_service, buff, (uint32)(msg.length() + 1), 60);

sleep(1);

}

}

1. 对应应用场景4) USE CASE:

void TestCase\_Group\_TO\_ONE(){

std::string test\_event\_1 = "TestEvent\_1";

std::string test\_event\_2 = "TestEvent\_2";

std::string test\_event\_3 = "TestEvent\_3";

std::string test\_group = "test\_group";

TestGroupConsumer\* consumer\_one;

TestGroupConsumer\* consumer\_two;

TestGroupConsumer\* consumer\_three;

std::string msg = "this is test group two msg";

char \* buff;

TNodeChannel channel = pnode->GetChannel();

consumer\_one = new TestGroupConsumer(test\_event\_1);

consumer\_two = new TestGroupConsumer(test\_event\_2);

consumer\_three = new TestGroupConsumer(test\_event\_3);

channel.CreateGroup(test\_group);

channel.AddConsumerToGroup(test\_group, consumer\_one);

channel.AddConsumerToGroup(test\_group, consumer\_two);

channel.AddConsumerToGroup(test\_group, consumer\_three);

sleep(1);

while (1){

buff = channel.AllocDataBuff((uint32)msg.length() + 1);

memcpy(buff, msg.c\_str(), (uint32)(msg.length() + 1));

channel.PublishGroup(test\_group, test\_event\_2, buff, (uint32)(msg.length() + 1), 70);

sleep(1);

}

}

1. 对应应用场景5) USE CASE:

void TestCase\_Group\_TO\_ALL(){

std::string test\_event\_1 = "TestEvent\_1";

std::string test\_event\_2 = "TestEvent\_2";

std::string test\_event\_3 = "TestEvent\_3";

std::string test\_event\_all = "$";

std::string test\_group = "test\_group\_all";

TestGroupConsumer\* consumer\_one;

TestGroupConsumer\* consumer\_two;

TestGroupConsumer\* consumer\_three;

std::string msg = "this is test group all msg";

char \* buff;

TNodeChannel& channel = pnode->GetChannel();

consumer\_one = new TestGroupConsumer(test\_event\_1);

consumer\_two = new TestGroupConsumer(test\_event\_2);

consumer\_three = new TestGroupConsumer(test\_event\_3);

channel.CreateGroup(test\_group);

channel.AddConsumerToGroup(test\_group, consumer\_one);

channel.AddConsumerToGroup(test\_group, consumer\_two);

channel.AddConsumerToGroup(test\_group, consumer\_three);

sleep(1);

while (1){

buff = channel.AllocDataBuff((uint32)msg.length() + 1);

memcpy(buff, msg.c\_str(), (uint32)(msg.length() + 1));

channel.PublishGroup(test\_group, test\_event\_all, buff, (uint32)(msg.length() + 1), 80);

sleep(1);

}

}

1. 对应应用场景6) USE CASE:

void TestCase\_Consumer\_balance(int cmd){

std::string test\_event = "test\_Balance\_Event";

TestConsumer\* consumer;

std::string msg = "this is test balance msg";

char \* buff;

int count = 10;

TNodeChannel& channel = pnode->GetChannel();

if (1 == cmd){

consumer = new TestConsumer(test\_event);

channel.AddConsumer(consumer);

}else if (2 == cmd){

while(count--){

buff = channel.AllocDataBuff((uint32)msg.length() + 1);

memcpy(buff, msg.c\_str(), (uint32)(msg.length() + 1));

channel.PublishEvent(test\_event, buff, (uint32)(msg.length() + 1), 90);

}

}

}

三个窗口分别运行下面三个命令：

./Tnode 1

./Tnode 1

./Tnode 2

1. 对应应用场景7) USE CASE:

void TestCase\_Group\_Consumer\_balance(int cmd){

std::string test\_event\_1 = "Test\_Group\_Balance\_Event\_1";

std::string test\_event\_2 = "Test\_Group\_Balance\_Event\_2";

std::string test\_event\_all = "$";

std::string test\_group = "test\_group\_Balance\_all";

TestGroupConsumer\* consumer\_one;

TestGroupConsumer\* consumer\_two;

std::string msg = "this is test group balance msg";

char \* buff;

int count = 10;

TNodeChannel& channel = pnode->GetChannel();

channel.CreateGroup(test\_group);

if (3 == cmd){

consumer\_one = new TestGroupConsumer(test\_event\_1);

channel.AddConsumerToGroup(test\_group, consumer\_one);

}else if (4 == cmd){

consumer\_two = new TestGroupConsumer(test\_event\_2);

channel.AddConsumerToGroup(test\_group, consumer\_two);

}else if (5 == cmd){

while (count--){

buff = channel.AllocDataBuff((uint32)msg.length() + 1);

memcpy(buff, msg.c\_str(), (uint32)(msg.length() + 1));

channel.PublishGroup(test\_group, test\_event\_all, buff, (uint32)(msg.length() + 1), 100);

}

}

}

四个窗口分别运行下面四个命令：

./Tnode 3

./Tnode 4

./Tnode 4

./Tnode 5