rocketmq cpp manual doc

本文主要介绍如何使用rocketmq cpp，包括如何设置相关参数和如何使用发送消费接口，其中所有实例均在./example目录下。另外，本文还对发送和消费过程中可能出现的问题提供了原理说明和排查方法；

## rocketmq cpp install&setup

### 1.1 rocketmq cpp build&install

#### 1.1.1 Linux platform

##### 1>. Dependency Installation

* Install libevent 2.0.22 dependency
  + Download [libevent 2.0.22](https://github.com/libevent/libevent/releases/download/release-2.0.22-stable/libevent-2.0.22-stable.tar.gz)
  + Build and install libevent
    - ./configure
    - make
    - make install
* Install JsonCPP 0.7.0 dependency
  + Download [jsoncpp 0.7.0](https://github.com/open-source-parsers/jsoncpp/archive/0.10.6.zip)
  + Build and install jsoncpp
    - cmake .
    - make
    - make install
* Install boost 1.56.0 dependency
  + Download [boost 1.56.0](http://www.boost.org/users/history/version_1_56_0.html)
  + Build and install boost
    - cd path/to/boost\_1\_56\_0
    - config boost: ./bootstrap.sh
    - build boost:
      * build static boost lib: ./b2 link=static runtime-link=static
      * build dynamic boost lib: ./b2 link=shared runtime-link=shared
    - install boost: ./b2 install

##### 2>. Make and Install

* Default install path:
  + header files: /usr/local/include
  + lib: /usr/local/lib
* Make and install using cmake
  + cmake will auto find\_package, if failes, change BOOST\_INCLUDEDIR/LIBEVENT\_INCLUDE\_DIR/JSONCPP\_INCLUDE\_DIR in CMakeList.txt, according to its real install path
  + make
  + make install

##### 3>. Check verion

* strings librocketmq.so |grep VERSION

#### 1.1.2 Windows platform:

##### 1>. Dependency Installation

* Install libevent 2.0.22 dependency
  + dowload libevent 2.0.22
    - <https://github.com/libevent/libevent/releases/download/release-2.0.22-stable/libevent-2.0.22-stable.tar.gz>
  + build and install libevent
    - extract libevent to C:/libevent
    - open VirtualStudio command line tools, go to dir: C:/libevent
    - execute cmd: nmake /f Makefile.nmake
    - cp libevent.lib, libevent\_extras.lib and libevent\_core.lib to C:/libevent/lib
* install JsonCPP 0.10.6 dependency
  + download jsoncpp 0.10.6
    - <https://github.com/open-source-parsers/jsoncpp/archive/0.10.6.zip>
  + build and install jsoncpp
    - extract jsoncpp to C:/jsoncpp
    - download cmake windows tool(<https://cmake.org/files/v3.9/cmake-3.9.3-win64-x64.zip>) and extract
    - run cmake-gui.exe, choose your source code dir and build dir, then click generate which will let you choose VirtualStudio version
    - open project by VirtualStudio, and build jsoncpp, and jsoncpp.lib will be got
* install boost 1.56.0 dependency
  + dowload boost 1.56.0
    - <http://www.boost.org/users/history/version_1_56_0.html>
  + build and install boost 1.56.0
    - according to following discription: <http://www.boost.org/doc/libs/1_56_0/more/getting_started/windows.html>
    - following build options are needed to be set when run bjam.exe: msvc architecture=x86 address-model=64 link=static runtime-link=static stage
    - all lib will be generated except boost\_zlib:
      * download zlib source: <http://gnuwin32.sourceforge.net/downlinks/zlib-src-zip.php> and extract to directory C:/zlib
      * run cmd:bjam.exe msvc architecture=x86 address-model=64 link=static runtime-link=static --with-iostreams -s ZLIB\_SOURCE=C:\zlib\src\zlib\1.2.3\zlib-1.2.3 stage

##### 2>. Make and Install

* generate project solution by cmake automatically
  + download cmake windows tool(<https://cmake.org/files/v3.9/cmake-3.9.3-win64-x64.zip>) and extract
  + run cmake-gui.exe, choose your source code dir and build dir, then click generate which will let you choose VirtualStudio version
  + if generate project solution fails, change BOOST\_INCLUDEDIR/LIBEVENT\_INCLUDE\_DIR/JSONCPP\_INCLUDE\_DIR in CMakeList.txt, according to its real install path
* open&build&run project by VirtualStudio

### 1.2 common setup introduction

Common configure parameters for both Producer and Consumer are in include/MQClient.h.

#### 1.2.1 log setup

Default log path: $HOME/logs/rocketmq-cpp,

Default log level: LOG\_LEVEL\_INFO

Default log file num and size: 3, 100M;

1. Get and Set log level

elogLevel getLogLevel();

void setMetaqLogLevel(elogLevel inputLevel)

1. Set log file size and file num；

void setMetaqLogFileSizeAndNum(int fileNum, long perFileSize)

#### 1.2.2 network threadpool setup

Interface

void setTcpTransportPullThreadNum(int num);

const int getTcpTransportPullThreadNum() const;

1> Default value is CPU num, it must be setted before producer/consumer start, minimum value is CPU num;

2> The pullThread num must be tested on your environment to find the best value for RT of sendMsg or delay time of consume msg before you change it;

3> Producer and consumer need different pullThread num, if set this num, producer and consumer must set different instanceName.

4> Configuration suggestion:

<1>. Minimum RT of sendMsg:

pullThreadNum = brokerNum\*2

#### 1.2.3 TCP connect timeout setup

Interface:

void setTcpTransportConnectTimeout(long long timeout);//ms

const long long getTcpTransportConnectTimeout() const;

1. Default value is 3000ms

2. Input parameter could only be milliSecond, suggestion value is 1000-3000ms;

#### 1.2.4 tryLockTime setup when getting TcpConnection

Interface:

void setTcpTransportTryLockTimeout(long long timeout);//ms

const long long getTcpTransportTryLockTimeout() const;

1. Parameter unit is ms, default value is 3000ms, the minimun value is 1000ms，suggestion value is 3000ms;

2. If configured with value smaller than 1000ms, the tryLockTimeout value will be setted to 1000ms;

#### 1.2.5 instanceName setup

instanceName决定多个producer或者consumer，是否共享服务资源(定时处理，TCP连接等);

1. 如果多个producer和consumer设置了相同的instanceName，会共享定时处理，TCP连接等，有利于节省线程数和定时任务，还能减少网络IO流量和socket连接数开销；
2. 对于异步发送和异步消费，如果多个producer和consumer设置了相同的instance name，producer和consumer的释放顺序应该是：

pProducer1->shutdown();

pProducer2->shutdown();

pConsumer1->shutdown();

pConsumer2->shutdown();

deleteAndZero(pProducer1);

deleteAndZero(pProducer2);

deleteAndZero(pConsumer1);

deleteAndZero(pConsumer2);

任意producer或者consumer的删除必须等待所有producer和consumer shutdown之后，否则会引起coredump问题

请参见：example/ WhileTestAsyncPushConsumer.cpp

### 1.3 example parameters introduction

- -n : nameserver addr

- -i : nameserver domain name

- -g : groupName, contains producer groupName and consumer groupName

- -t : msg topic

- -m : message count(default value:1)

- -c : msg content(default value: only test)

- -b : consume model(default value: CLUSTER)

- -a : set sync push(default value: async)

- -r : setup retry times(default value:5 times)

- -u : select active broker to send msg(default value: false)

- -d : use AutoDeleteSendcallback by cpp client(defalut value: false)

- -T : thread count of send msg or consume msg(defalut value: system cpu core number)

- -v : print more details information

- Example:

- sync producer: ./SyncProducer -g producerGroup -t topic -c msgContent -m msgCount -n nameServerAddr

- async producer: ./AsyncProducer -g producerGroup -t topic -c msgContent -m msgCount -n nameServerAddr

- send delay msg: ./SendDelayMsg -g producerGroup -t topic -c msgContent -n nameServerAddr

- sync pushConsumer: ./PushConsumer -g producerGroup -t topic -c msgContent -m msgCount -n nameServerAddr -s sync

- async pushConsumer: ./AsyncPushConsumer -g producerGroup -t topic -c msgContent -m msgCount -n nameServerAddr

- orderly async pushConsumer: ./AsyncOrderlyPushConsumer -g producerGroup -t topic -c msgContent -m msgCount -n nameServerAddr

- sync pullConsumer:./PullConsumer -g producerGroup -t topic -c msgContent -m msgCount -n nameServerAddr

## Producer introduction

Producer是发送MQ消息的主体，按照发送方式可以分为同步发送和异步发送。

### 2.1 发送接口和实例说明

Producer接口位于include/MQProducer.h, 所有发送接口失败后，会报exception，请务必捕捉exception；

#### 2.1.1同步并发发送

1. 实例：rpm包中 example/SyncProducer.cpp

2. 接口

SendResult send(const MQMessage& msg, bool bSelectActiveBroker=false)

SendResult send(const MQMessage& msg, const MQMessageQueue& mq)

3. 发送自动重试5次；

4. 如果设置bSelectActiveBroker为true，优先选择ActiveBroker发送；

#### 2.2.2异步并发发送

1. 实例：rpm包中 example/AsyncProducer.cpp

2. 接口：

void send(const MQMessage& msg, SendCallback\* sendCallback, bool bSelectActiveBroker=false)

void send(const MQMessage& msg, const MQMessageQueue& mq, SendCallback\* sendCallback)

3. 发送自动重试3次；

4. 如果设置bSelectActiveBroker为true，优先选择ActiveBroker发送；

#### 2.2.3 oneway并发发送

1. 接口

void sendOneway(const MQMessage& msg, bool bSelectActiveBroker=false)

void sendOneway(const MQMessage& msg, const MQMessageQueue& mq)

2. 发送自动重试3次；

3. 如果设置bSelectActiveBroker为true，优先选择ActiveBroker发送；

#### 同步顺序发送

1. 实例：rpm包中example/OrderProducer.cpp
2. 严格顺序消息

SendResult send(const MQMessage& msg, MessageQueueSelector\* selector, void\* arg)

1>. 无自动重试，发送失败报exception;

1. 非严格顺序消息

SendResult send(const MQMessage& msg, MessageQueueSelector\* selector, void\* arg, int autoRetryTimes, bool bActiveBroker=false)

1>. 适应于非严格顺序消息，发送失败后，自动重试autoRetryTimes，此时不能保证消息的严格有序；

2>. 如果设置bSelectActiveBroker为true，首先尝试指定的队列发送，然后优先选择ActiveBroker发送；

#### 2.2.5 oneway顺序发送

void sendOneway(const MQMessage& msg, MessageQueueSelector\* selector, void\* arg)

#### 2.2.6 延时发送

实例：rpm包中的example/SendDelayMsg.cpp

### 2.2 发送参数设置

#### 2.2.1 发送超时时间

1. 接口位于include/DefaultMQProducer.h

int getSendMsgTimeout() const;

void setSendMsgTimeout(int sendMsgTimeout);

2. 接口单位为ms，默认超时时间为3000ms，建议设置时间为2000ms-3000ms；

3. 如果超时时间设置的过小，会导致消息重复的可能性(服务端已经收到消息并在存储，还未来得及回复，但是客户端因为超时重发消息)

### 2.3 发送性能测试

发送性能主要由客户端机器处理能力决定，请参考性能测试报告：

16个队列，4个发送线程的情况下，发送TPS:3W

## 3 Consumer功能说明

消费按消费方式分为集群和广播消费，按拉取方式分为pull和push两种；

消费tps主要由业务方对回调函数的处理速度决定，因为rocketmq cpp本身消费tps很高，不会成为性能瓶颈；

本章主要介绍pushConsumer的用法和设置；

### 3.1 消费方式设置

#### 3.1.1 pull/push consumer设置

consumer按拉取方式分为pull和push两种；

pushConsumer：应用通常向Consumer对象注册一个Listener接口，一旦收到消息Consumer对象立刻回调Listener接口方法。

pullConsume：应用通常主动调用Consumer的拉消息方法从Broker拉消息，主动权由应用控制。

对于pullConsumer，并不区分集群和广播消费，而且只有同步消费方式，而且业务方完全控制拉取的时间和维护消费offset，rocketmq cpp客户端和服务端都不维护pullConsumer的消费offset，实例详见: example/PullConsumer.cpp

#### 3.1.2 pushConsumer集群/广播消费设置

消费按消费方式分为集群和广播消费，默认为集群消费；

集群消费是指各个consumer实例平均消费所有消息队列，广播消费是指所有consumer实例都消费全量消息队列；

接口位于include/MQConsumer.h，默认为集群消费(clustering)

MessageModel getMessageModel() const

void setMessageModel(MessageModel messageModel)

### 3.2 pushConsumer消费参数设置

#### 3.2.1 消费线程池设置

消费线程池：决定线程池里有多少线程向业务方上报拉取到的消息，对同步和异步消费都生效，默认线程数为cpu核数；

接口：

void setConsumeThreadCount(int threadCount);

#### 3.2.2 拉取线程池设置

1.本接口仅决定同步消费拉取的线程池有多少线程工作，接口位于include/DefaultMQConsumer.h；

void setPullMsgThreadPoolCount(int threadCount);

int getPullMsgThreadPoolCount();

2. 拉取线程池的默认线程数为CPU核数，可设置；

#### 3.2.3 顺序消费及其批量消息设置

1. 顺序消费和并发消费的不同在于：msgListener继承的基类不同

顺序消费的msgListener继承MessageListenerOrderly

2. 本接口用于设置顺序消费每次返回给业务方的消息数，默认为1；

int getConsumeMessageBatchMaxSize();

void setConsumeMessageBatchMaxSize(int consumeMessageBatchMaxSize);

#### 3.2.4 消费起始拉取节点设置

1. 接口：setConsumeFromWhere

2. 说明：

所有consumer的拉取节点，请设置为：CONSUME\_FROM\_LAST\_OFFSET，否则，当向服务端请求拉取offset失败后，会从最小节点开始拉取

### 3.3 pushConsumer消费

#### 3.3.1 pushConsumer 集群消费

##### 3.3.1.1异步并发消费

1. 实例：rpm中的example/AsyncPushConsumer.cpp

##### 3.3.1.2异步顺序消费

1. 实例：rpm包中的example/AsyncOrderlyPushConsumer.cpp

#### 3.3.2 pushConsumer 广播消费

广播消费是指所有消费实例会接收到topic的全量消息，所以服务端不能维护每个消费实例的consume offset，需要客户端维护本地offset；

rocketmq cpp客户端会维护本地offset的更新：

1. 文件目录

[$HOME/.rocketmq\_offsets/IPAddr@consumerInstanceName/consumerGroupName/offsets.Json](mailto:$HOME/.rocketmq_offsets/IPAddr@consumerInstanceName/consumerGroupName/offsets.Json)

文件目录按照consumer instanceName和groupName来区分，所以不同consumer的实例需要设置不同的instance name，防止相互覆盖；

2. consumer第一次启动，rocketmq cpp会自动创建offset.Json文件

3. 如果offset.Json文件的内容被手动清除，那么下次启动consumer会报如下exception：

THROW\_MQEXCEPTION(MQClientException, "open offset store file failed, please check whether offsets.Json is cleared by operator, if so, delete this offsets.Json file and then restart consumer", -1);

此时需要用户删除offset.Json文件，再重新启动consumer

#### 3.3.3 pushConsumer线程池说明

1. 并发消费线程池：consumeTP

并发消费是默认的消费模式，通过查看consumeTP线程池里线程的调用栈，可以显示出当前消费线程的状态，对于查看消费堆积和消费卡住等问题非常有用；

其设置方法请参见3.2.2；

2. 顺序消费线程池：orderlyConsumeTP

顺序消费是特指对顺序消息的严格顺序消费，通过查看orderlyConsumeTP线程池里线程的调用栈，可以显示出当前消费线程的状态，对于查看消费堆积和消费卡住等问题非常有用；

其设置方法请参见3.2.2；

3. 拉取消息线程池：pullMsgTP

拉取消息线程池，一般情况下请使用默认设置，否则可能会引起cpu负载较高等问题；

其设置方法请参见3.2.3；

4. 网络处理线程池：networkTP

网络处理线程池默认为CPU核数，不允许设置，可以理解为网络work线程；

### 3.4 consumer消费原理说明

#### 3.4.1 消费原则说明

##### 3.4.1.1 集群消费多个进程的相同consumer订阅关系必须保持一致

多个进程的同一个consumer如果订阅关系不一致，比如不同进程订阅了不同的tag，不同的topic，都会出现某些进程接收不到消息等错误；

##### 3.4.1.2 同一个进程只能启动一个相同名字的consumer，即使instanceName不一样

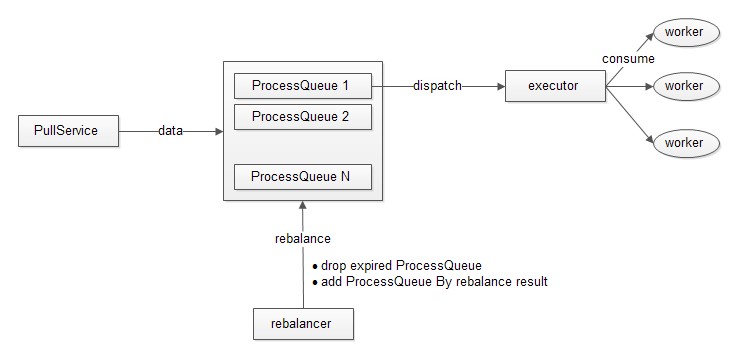
##### 3.4.1.3 广播消费没有消费回溯功能

##### 3.4.1.4 PullConsumer需要业务方处理容灾问题

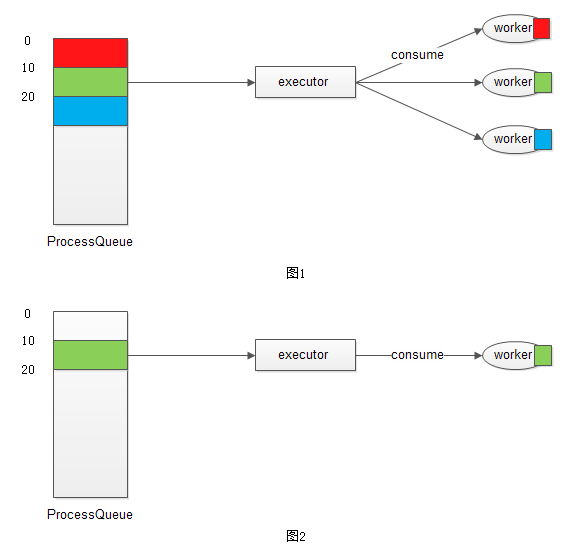
实例：rpm包中的example/PullConsumer.cpp

##### 3.4.1.5 PullConsumer不支持消费堆积报警

#### 3.4.2 非顺序消息如何实现队列内并行

* 并行消费主流程  
    
   PullService根据rebalance结果从对应的队列中获取数据，并将其缓存到客户端本地内存，每次拉取到的数据通过不同的消费线程回调上层应用。

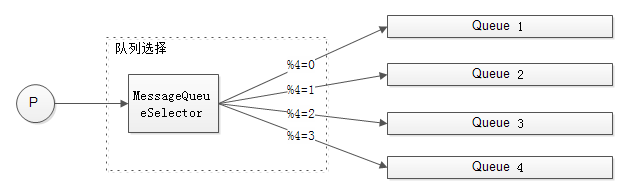
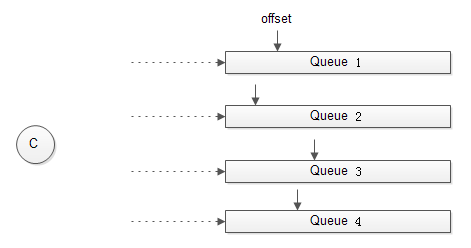
NOTE:拉消息和消费处理是异步的，不需要等待之前消息处理完，再去拉消息

* 数据提交  
   消费端一个队列只维护一个offset，消费成功后只提交最小的offset。  
    
   如图示例：  
   将拉回来的数据分成了三段（0-10,10-20,20-30）分别派发给消费线程。 0-10,20-30这两部分数据消费完成，但10-20这段数据还未消费完成。提交服务端消费位点则为10，知道这段数据消费完成提交位点才会为30。  
   NOTE：所以并发消费有可能出现重复消费的问题。如中间有部分数据一直没有被成功消费，此时重新负载，别的消费端拿到当前队列就会导致重复消费。

#### 3.4.3 如何确保消息顺序

Rocketmq cpp队列模型使得其能够提供顺序消息这一特性。顺序，是指单个队列内部能够保证消息的有序。

如何确保消息的有序：

* 发送端将需要保证有序的消息发送端一个队列(参考2.2.4)  
  发送端通过‘队列选择’将需要顺序的消息发送到一个队列里面（[发送端示例](http://gitlab.alibaba-inc.com/middleware/metaq3/blob/master/metaq-example/src/main/java/com/taobao/metaq/example/ordermessage/Producer.java)）。如下图所示：  
    
  PS：不同规则的数据可以发送到不同的队列中已提高并行度
* 接收端顺序消费(参考3.2.3和3.2.4)  
  消费端顺序消费，保证接收到的数据顺序和发送的顺序一致（[接收端示例](http://gitlab.alibaba-inc.com/middleware/metaq3/blob/master/metaq-example/src/main/java/com/taobao/metaq/example/ordermessage/Consumer.java)）。如下图所示：  
    
  依赖队列的特性保证队列内数据的有序性。

#### 3.4.4 收不到消息如何处理

* 消息是否有发出（ops按topic查询）
* 查看消费端机器端口号9876(nameServer)和10911(Broker)有没有连接上；

netstat -ant|grep 9876、netstat -ant|grep 10911，

如果显示和某一台机器ESTABLISHED，那说明和服务端正常连接；

* 通过ops上的连接查询，查询下当前Group下是否有其他consumer，如果组名相同，有可能被别的consumer收走
* 确认tag是否和发送的一样
* 确认订阅关系是否一致，即一个消费组下面所有消费端都要订阅同样数量的topic、同样的topic名字以及同样的tag；
* 显示的消费组确认是否有订阅权限，没有的话需要申请；
* 换一个订阅组重新接收

#### 3.4.5 重复消费怎么办

1. 重复消息产生的原因：

* 发送端发送超时，重新发送，有可能上一次消息已经发送端server端，由于一些其他原因响应客户端超时
* 接收端费征程结束，比如直接kill -9
* 多进程集群消费，各个进程启动和关闭的瞬间

所以broker很难避免消息的重复，Rocketmq cpp不能保证消息绝对不重，Rocketmq cpp保证的是“at least once”

2. 如何去重

* 根据MsgId去重【不推荐】 接收端可以根据msgId去重保证收到的消息不重复，但无法避免发送端发重复消息的情况
* 根据业务key去重 根据业务的一个主键去重，主键可以作为消息的一个属性

3. 如何排查

* 消费端是否消费失败，导致消息重试 可以在ops消费进度查询里面看到重试的次数%RETRY%
* 收到的消息msgId是否一样，不一样则表示消息发送端可能发送重复消息

#### 3.4.6 消费重试

##### 3.4.6.1 Rocketmq cpp并发消费方式，消费失败如何重试

* 返回RECONSUME\_LATER, Rocketmq cpp会自动重试
* 会自动重试5次，每次间隔时间依次如下（s:秒 m:分钟）

10s 30s 1m 2m 3m

* 超过5次还失败，则消息进入死信队列，不再提供重试机制。死信队列的消息应用不可以读取。
* 每次消费失败后，应用可以设置消息下次间隔多长时间到达Consumer，正常情况下，间隔时间会比较准确，在Broker压力大时，间隔时间可能会比实际设置的时间长。

注意：只能设置定时Level，不能设置具体时间

* 失败的消息存储到服务器，重试可靠性较高。

##### 3.4.6.2 Rocketmq cpp顺序消费方式，消费失败如何重试

* 返回RECONSUME\_LATER, Rocketmq cpp会自动重试，直到返回成功才停止重试。
* 因为要保证消费的顺序性，所以一旦某个队列的某条消息消费失败，那么这个队列暂停消费。其他队列不受影响
* 重试行为完全在客户端执行，不需要服务器参与。

#### 3.4.7 延迟消费时间级别设置

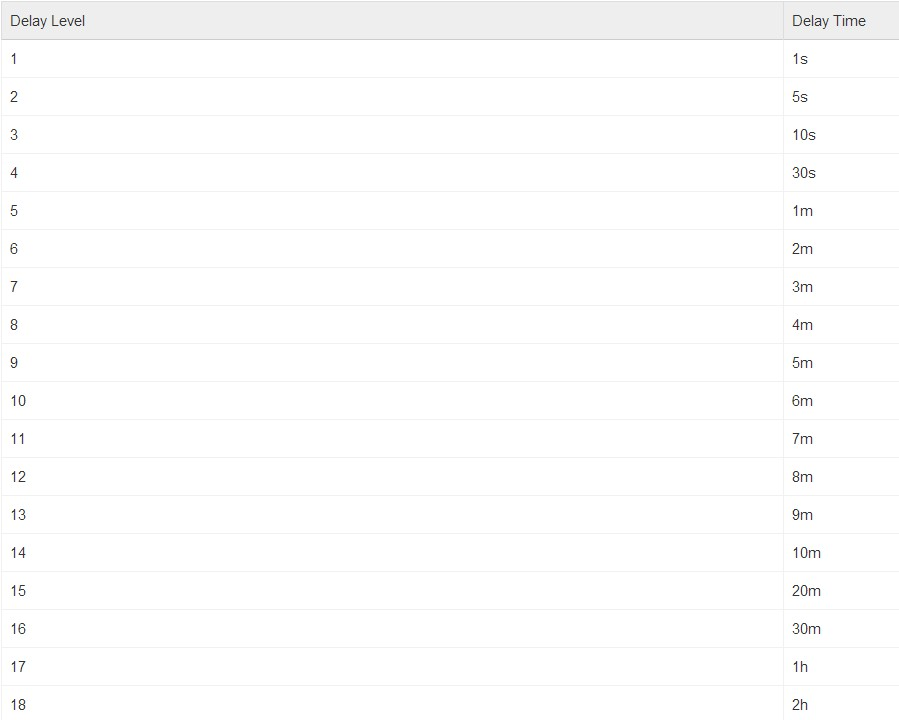
1. 定时消息

发送端生产处消息后一段时间消费端在接收到消息，可用于消息延迟消费、消息定时重试的场景。

2. 使用方式

消息发送方设置消息延时级别： Message.setDelayTimeLevel()

3. 延时级别



#### 3.4.8 消费堆积怎么办

1. 查看消费线程池的线程调用栈(gstack)

通过查看consumeTP(并发消费)/orderlyConsumeTP(顺序消费)线程池里线程的调用栈，可以显示出当前消费线程的状态，对于查看消费堆积和消费卡住等问题非常有用；

如果消费线程被卡住，rocketmq cpp不能获得消费结果，那么就不能更新offset到服务端，就会显示消费堆积；

### 3.5 消费性能分析

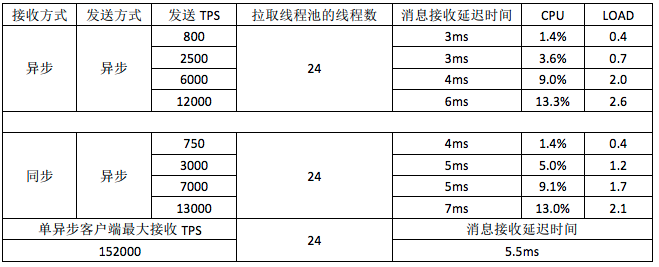
#### 3.5.1 consumer启动迅速

从consumer start到接收到消息，仅需500ms左右；

#### 3.5.2 消费TPS和消费延时

任何系统下，同步和异步消费的平均消费延时都低于10ms

16个队列的消费TPS和消费延时如下：



### 3.6 集群消费不可避免的消息重复分析

rocketmq cpp的设计原则保证消息绝对不会丢失，但是异常情况下存在重复的可能性；

#### 3.6.1 多个消费进程上下线引发的消息重复；

1. 测试场景：3个进程相继启动，消费同一个topic(40W条消息)，

2. 测试结果：

发现三个进程消费消息数为：  
 204568+49687+152111 = 406366，总共重复消费： 6366  
 3. 按时间顺序列出发生的事件，并分析重复消费的不可避免：  
 1>. 进程21440首先注册成功，并占据Q0-Q3三个消费队列  
 2>. 进程21413和进程21386注册成功  
 3>. 进程21413做rebalance，获取到队列Q2，从broker获取的消费进度为：23963，实际上进程21440已经消费到：24855

Q2重复消费： 24855-23963 = 892  
 4>. 进程21386做rebalance，获取到队列Q0-Q1，从broker获取的消费进度为：22134和22103，实际上进程21440已经消费到：24859和24852  
 Q0重复消费：24859-22134 = 2725  
 Q1重复消费：24852-22103 = 2749  
 5>. 进程21440做rebalance，发现自身只能消费队列Q3，

在remove：Q0-Q2之前，更新Q0-Q2的消费进度到broker，但已经不能阻止3和4的重复消费

6>. 重复消费的总数量6366 = Q0重复消费2725 + Q1重复消费2749 + Q2重复消费892  
 4. 综上分析可以得出，集群消费的多个进程启动瞬间会产生不可避免的重复消费，所以业务方要做去重工作，当然去重也不能依赖msgId;