

消息中间件RocketMQ （C++） API规范说明

V0.1

拟制: 汲洋弘康

审核：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

批准：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

南京中兴软创科技有限责任公司

修改记录

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **文件编号** | **版本号** | **拟制人/修改人** | **拟制/修改日期** | **更改理由** | **主要更改内容**  **（写要点即可）** |
|  | V0.1 | 汲洋弘康 | 2015-8 | 文档创建 |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| 注1：每次更改归档文件，需填写此表.  注2：文件第一次归档时，“主要更改内容”栏写无. | | | | | |

# 引言

## 编写目的

介绍RocketMQ （C++版本）的API使用

## 参考资料

《RocketMQ规范》

《消息中间件API规范说明（java版本）》

# 术语、定义和缩略语

## 术语、定义

专用的术语、定义见下表：

|  |  |  |
| --- | --- | --- |
| 1. 术语/定义 | 1. 英文 | 1. 说明 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# API

## 概述

RocketMQ是一个开源的消息中间件，本文主要针对其发送和push消费的接口实现作出说明。

## 发送

### Message

Message规定了通用的消息结构，对外提供用于构造消息。

主要构造方法：

Message::Message(const std::string& topic, const std::string& tags,const std::string& keys, const char\* body,int len)

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| topic | String | 是 | 消息主题 |
| tags | String | 否 | 消息标签 |
| key | String | 否 | 消息关键字 |
| body | Char\* | 是 | 所发消息具体内容，消息体 |
| len | int | 是 | 消息体长度 |

### DefaultMQProducer : public ClientConfig ,public MQProducer

DefaultMQProducer规定了RocketMQ通用的消息发送接口。

主要方法：

* 构造producer对象

DefaultMQProducer::DefaultMQProducer(const std::string& producerGroup)

作用：该函数用于构造producer对象，指定生产组

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| producerGroup | string | 否 | 生产组。如不给定，则采用默认生产组。 |

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| Producer | DefaultMQProducer | Producer对象 |

* 设置NameServer地址

ClientConfig::setNamesrvAddr(const std::string& namesrvAddr)

**入参列表**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| namesrvAddr | string | nameserver的地址 |

* Producer启动函数

DefaultMQProducer::start()

函数作用：启动Producer，从NS上取得默认topic对应的broker，开启定时线程任务。

* 发送（使用同步发送一般消息的方法）

DefaultMQProducer::send(Message& msg)

作用：该函数用于同步发送一般消息

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| msg | Message | 是 | 消息 |

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| sendResult | SendResult | 发送消息结果 |

* 发送（指定消息队列同步发送消息的方法）

DefaultMQProducer::send(Message& msg, MessageQueue& mq)

作用：该函数用于对指定消息队列，同步发送一般消息

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| msg | Message | 是 | 消息 |
| mq | MessageQueue | 是 | 消息队列 |

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| sendResult | SendResult | 发送消息结果 |

* 发送（使用异步发送消息的方法）

DefaultMQProducerImpl::send(Message& msg, SendCallback\* pSendCallback)

作用：该函数用于异步发送消息

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| msg | Message | 是 | 消息 |
| pSendCallback | SendCallback | 是 | 异步发送消息回调接口 |

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| sendResult | SendResult | 发送消息结果 |

* 发送（指定消息队列异步发送消息的方法）

DefaultMQProducer::send(Message& msg, MessageQueue& mq, SendCallback\* pSendCallback)作用：该函数用于对指定消息队列，异步发送消息

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| msg | Message | 是 | 消息 |
| mq | MessageQueue | 是 | 消息队列 |
| pSendCallback | SendCallback | 是 | 异步发送消息回调接口 |

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| sendResult | SendResult | 发送消息结果 |

* 发送（使用oneway方式发送消息的方法，考虑最大发送性能）

DefaultMQProducer::sendOneway(Message& msg)

作用：该函数用于oneway方式发送消息

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| msg | Message | 是 | 消息 |

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| sendResult | SendResult | 发送消息结果 |

* 发送（指定消息队列使用oneway方式发送消息的方法，考虑最大发送性能）

DefaultMQProducer::sendOneway(Message& msg, MessageQueue& mq)

作用：该函数用于对指定消息队列，采用oneway方式发送消息

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| msg | Message | 是 | 消息 |
| mq | MessageQueue | 是 | 消息队列 |

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| sendResult | SendResult | 发送消息结果 |

* 发送（构造队列选择器，同步发送消息的方法）

SendResult DefaultMQProducer::send(Message& msg, MessageQueueSelector\* pSelector, void\* arg)

作用：该函数用于外部构造队列选择器，向选择的队列同步发送一般消息

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| msg | Message | 是 | 消息 |
| pSelector | MessageQueueSelector | 是 | 队列选择器 |
| arg | Void | 是 |  |

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| sendResult | SendResult | 发送消息结果 |

### sendResult

RocketMQ发一般消息所返回的结果

**主要参数**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| m\_sendStatus | SendStatus  （是一个枚举类型enum SendStatus） | 发送状态，有四种状态：  SEND\_OK:消息发送成功  FLUSH\_DISK\_TIMEOUT: 消息发送成功，但是服务器刷盘超时，消息已经进入服务器队列，只有此时服务器宕机，消息才会丢失 (仅Rocketmq)  FLUSH\_SLAVE\_TIMEOUT: 消息发送成功，但是服务器同步到Slave时超时，消息已经进入服务器队列，只有此时服务器宕机，消息才会丢失(仅Rocketmq)  SLAVE\_NOT\_AVAILABLE: 消息发送成功，但是此时slave不可用，消息已经进入服务器队列，只有此时服务器宕机，消息才会丢失(仅Rocketmq) |
| m\_msgId | String | 消息ID，由Broker生成 |
| m\_messageQueue | MessageQueue | 消息队列，消息所发到的消息队列 |
| m\_queueOffset | long long | 队列偏移值 |

主要方法：

* 取发送消息返回结果状态

SendResult::getSendStatus()

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| m\_sendStatus | SendStatus | 同上 |

* 取发送消息返回消息ID

SendResult::getMsgId()

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| m\_msgId | String | 消息ID，由Broker生成 |

* 取发送消息返回消息队列

SendResult::getMessageQueue()

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| m\_messageQueue | MessageQueue | 消息队列 |

### MessageQueue

对外提供的消息队列数据结构。

**主要参数**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| m\_topic | string | 消息主题 |
| m\_brokerName | string | Broker名称 |
| queueId | Int | 队列ID |

主要方法：

* 构造消息队列

MessageQueue::MessageQueue(const std::string& topic, const std::string& brokerName, int queueId) :m\_topic(topic),m\_brokerName(brokerName),m\_queueId(queueId)

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| m\_messageQueue | MessageQueue | 消息队列指针 |

### MessageQueueSelector

选择队列进行发送消息，只提供接口，需要用户实现

主要方法：

* virtual MessageQueue\* select(std::vector<MessageQueue\*>& mqs, const Message& msg, void\* arg)=0;

### SendCallback

异步发送消息，对外提供回调接口

主要方法：

* virtual void onSuccess(SendResult& sendResult)=0;
* virtual void onException(MQException& e)=0

### 发送举例

具体举例请参见附件。附件中对各个接口的调用都有详尽解释。

## 消费（被动push方式）

RocketMQ提供了监听接收消息的方法，本文主要针对这种取消息的接口进行说明，被动拉取消息采用长轮询的方式实时拉去消息。

### DefaultMQPushConsumer : public ClientConfig ,public MQPushConsumer

DefaultMQPushConsumer规定了通用的消息接收接口

主要方法：

* 构造被动消费对象Pushconsumer

DefaultMQPushConsumer::DefaultMQPushConsumer(const std::string& consumerGroup)

**入参列表**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| consumerGroup | string | 消费组 |

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| consumer | DefaultMQPushConsumer | Pushconsumer对象 |

* 设置NameServer地址

ClientConfig::setNamesrvAddr(const std::string& namesrvAddr)

**入参列表**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| namesrvAddr | string | nameserver的地址 |

* 设置实例名

ClientConfig::setInstanceName(const std::string& instanceName)

**入参列表**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| instanceName | string | 实例名 |

* 设置消息订阅关系

DefaultMQPushConsumer::subscribe(const std::string& topic, const std::string& subExpression)

函数作用：指定所发消息的消息主题和tags。

**入参列表**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| topic | string | 消息主题 |
| subExpression | string | 消息tags |

* 注册消息监听器

DefaultMQPushConsumer::registerMessageListener(MessageListener\* pMessageListener)

函数作用：注册外部构造的消息监听器，回调实现消息消费。

**入参列表**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| pMessageListener | MessageListener | 外部消息监听器回调函数，需要用户实现。继承了相关接口。 |

* Pushconsumer启动

DefaultMQPushConsumer::start()

函数作用：启动pushconsumer，根据订阅关系从NS上取得对应的broker，长轮询这些broker接收消息。

### MessageListenerConcurrently : public MessageListener

消息监听器类，并发消费消息。被动方式订阅消息使用，具体使用方式需要用户实现。用户根据需要，在外部构造监听器继承此类，并实现消费消息的方法。

主要方法：

* 并发消费消息（需要用户实现）

virtual ConsumeConcurrentlyStatus consumeMessage(std::list<MessageExt\*>& msgs,

ConsumeConcurrentlyContext& context)=0

函数作用：本身是虚函数，没有实现，只提供接口。需要用户在外部构造消息监听器时，将该方法实现。

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| msgs | MessageExt | 是 | 消费端消息结构 |
| context | ConsumeConcurrentlyContext | 是 | 并发消费消息的环境结构 |

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| Consume\_status | ConsumeConcurrentlyStatus | CONSUME\_SUCCESS ：表示消费成功  RECONSUME\_LATER ：表示消费失败，但是稍后还会重新消费这批消息 |

### MessageListenerOrderly : public MessageListener

消息监听器类，顺序消费消息。被动方式订阅消息使用，具体使用方式需要用户实现。用户根据需要，在外部构造监听器继承此类，并实现消费消息的方法。

主要方法：

* 顺序消费消息（需要用户实现）

virtual ConsumeOrderlyStatus consumeMessage(std::list<MessageExt\*>& msgs,

ConsumeOrderlyContext& context)=0

函数作用：本身是虚函数，没有实现，只提供接口。需要用户在外部构造消息监听器时，将该方法实现。

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| msgs | MessageExt | 是 | 消费端消息结构 |
| context | ConsumeOrderlyContext | 是 | 顺序消费消息的环境结构 |

**返回结果**

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| Consume\_status | ConsumeOrderlyStatus | SUCCESS：消息处理成功  ROLLBACK：回滚消息  COMMIT：提交消息  SUSPEND\_CURRENT\_QUEUE\_A\_MOMENT：将当前队列挂起一小会儿 |

### 接收举例

具体举例见附件。附件中对各个接口的调用都有做注释。

# 客户端配置

## 概述

客户端配置主要分为三部分：客户端公共配置，producer端配置和pushconsumer端配置。对于客户端公共配置，主要的应用在ClientConfig类中，而其他对应配置也会继承该类。

客户端重要参数的配置接口都已经封装好，设置配置以set的形式，获取配置以get的形式。下面会介绍具体参数的配置方法。

## 客户端公共配置（ClientConfig）

### 参数列表

|  |  |  |
| --- | --- | --- |
| 参数名 | 默认值 | 说明 |
| m\_namesrvAddr | 读环境变量 | NameServer地址，多个地址用分号隔开 |
| m\_clientIP | 本机IP | 客户端IP地址，系统内部调用函数赋初值。若无法获取，则通过接口赋值 |
| m\_clientCallbackExecutorThreads | 4 | 通信层异步回调线程数 |
| m\_instanceName | DEFAULT | 实例名 |
| m\_pollNameServerInteval | 1000 \* 30 | 轮询NameServer间隔时间，单位毫秒 |
| m\_heartbeatBrokerInterval | 1000 \* 30 | 向broker发心跳间隔时间，单位毫秒 |
| m\_persistConsumerOffsetInterval | 1000 \* 30 | 持久化间隔时间，单位毫秒 |
| G\_MQLOGLEVEL | 2 | 日志级别，默认MQLOG\_NOTICE，数字越小日志越详细 |

### 配置方法

公共配置既能被producer端使用也能被consumer端使用。实际上，公共配置封装的类ClientConfig被DefaultMQProducer和DefaultMQPushConsumer继承。**因此，在使用时并不直接构造ClientConfig对象，而是通过DefaultMQProducer/DefaultMQPushConsumer调用相关赋值接口。**具体的配置方法如下：

* m\_namesrvAddr

1. 环境变量中配置：对应环境变量中的值为“NAMESRV\_ADDR”。
2. void ClientConfig::setNamesrvAddr(const std::string& namesrvAddr)

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| namesrvAddr | string | 是 | 需要设置的nameserver，多个nameserver以分号分割 |

1. 也可以调用get接口取出已经配置的量：

std::string ClientConfig::getNamesrvAddr()

* m\_clientIP

1. void ClientConfig::setClientIP(const std::string& clientIP)

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| clientIP | string | 是 | 需要设置的客户端IP |

1. 也可以调用get接口取出已经配置的量：

std::string ClientConfig::getClientIP()

* m\_clientCallbackExecutorThreads

1. void ClientConfig::setClientCallbackExecutorThreads(int clientCallbackExecutorThreads)

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| clientCallbackExecutorThreads | int | 是 | 通信层异步回调线程数 |

1. 也可以调用get接口取出已经配置的量：

int ClientConfig::getClientCallbackExecutorThreads()

* 参数配置都是采取简单的set/get形式，具体形式如上。后面只列出对应接口函数名称，set与get一一对应，前者表示设置，后者表示获取。

|  |  |
| --- | --- |
| 参数名 | 接口名 |
| m\_namesrvAddr | setNamesrvAddr(const std::string& namesrvAddr)  getNamesrvAddr() |
| m\_clientIP | setClientIP(const std::string& clientIP)  getClientIP() |
| m\_clientCallbackExecutorThreads | setClientCallbackExecutorThreads(int clientCallbackExecutorThreads)  getClientCallbackExecutorThreads() |
| m\_instanceName | setInstanceName(const std::string& instanceName)  getInstanceName() |
| m\_pollNameServerInteval | setPollNameServerInteval(int pollNameServerInteval)  getPollNameServerInteval() |
| m\_heartbeatBrokerInterval | setHeartbeatBrokerInterval(int heartbeatBrokerInterval)  getHeartbeatBrokerInterval() |
| m\_persistConsumerOffsetInterval | setPersistConsumerOffsetInterval(int persistConsumerOffsetInterval)  getPersistConsumerOffsetInterval() |
| G\_MQLOGLEVEL | setLogLevel(int iLevel)  getLogLevel() |

### 举例说明

//在producer端程序中设置公共配置中的nameserver

#头文件

int main(int argc, char\* argv[])

｛

······

DefaultMQProducer producer("MyProducerGroup"); //构造producer对象，继承了公共配置类

producer.setNamesrvAddr(argv[1]); //调用接口setNamesrvAddr，将传入参数设置为公共配置中的nameserver地址

······

return 0

｝

## producer端配置（DefaultMQProducer）

### 参数列表

|  |  |  |
| --- | --- | --- |
| 参数名 | 默认值 | 说明 |
| m\_producerGroup | DEFAULT  \_PRODUCER | producer组名，多个producer如果属于一个应用，发送同样的消息，则应该将它们归为一组 |
| m\_createTopicKey | TBW102 | 在发送消息时自动创建服务器不存在的topic，但需要指定key |
| m\_defaultTopicQueueNums | 4 | 默认创建的队列数 |
| m\_sendMsgTimeout | 3000ms | 发送消息超时时间 |
| m\_compressMsgBodyOverHowmuch | 4Kb | 消息压缩阀值，超过压缩 |
| m\_maxMessageSize | 128Kb | 最大消息大小 |
| m\_retryAnotherBrokerWhenNotStoreOK | false | 发送失败是否重试发送 |

### 配置方法

producer端的参数配置方法，依然是采用类封装的set/get函数，具体形式请参照[4.2.2节](#_配置方法)所列。在配置producer端的参数时，只需要构造出DefaultMQProducer对象，再调用对应接口进行对应赋值。

下面列出对应接口函数名称，set与get一一对应，前者表示设置，后者表示获取。

|  |  |
| --- | --- |
| 参数名 | 接口名 |
| m\_producerGroup | setProducerGroup(const std::string& producerGroup)  getProducerGroup() |
| m\_createTopicKey | setCreateTopicKey(const std::string& createTopicKey)  getCreateTopicKey() |
| m\_defaultTopicQueueNums | setDefaultTopicQueueNums(int defaultTopicQueueNums)  getDefaultTopicQueueNums() |
| m\_sendMsgTimeout | setSendMsgTimeout(int sendMsgTimeout)  getSendMsgTimeout() |
| m\_compressMsgBodyOverHowmuch | setCompressMsgBodyOverHowmuch(int compressMsgBodyOverHowmuch)  getCompressMsgBodyOverHowmuch() |
| m\_maxMessageSize | setMaxMessageSize(int maxMessageSize)  getMaxMessageSize() |
| m\_retryAnotherBroker  WhenNotStoreOK | setRetryAnotherBrokerWhenNotStoreOK(bool retryAnotherBrokerWhenNotStoreOK)  isRetryAnotherBrokerWhenNotStoreOK() |

### 举例说明

//在producer端程序中设置producer端配置的m\_sendMsgTimeout

#头文件

int main(int argc, char\* argv[])

｛

······

DefaultMQProducer producer("MyProducerGroup"); //构造producer对象

producer. setSendMsgTimeout (5000); //调用接口setSendMsgTimeout，将传入参数设置为m\_sendMsgTimeout的值

······

return 0

｝

## pushconsumer端配置（DefaultMQPushConsumer）

### 参数列表

|  |  |  |
| --- | --- | --- |
| 参数名 | 默认值 | 说明 |
| m\_consumerGroup | DEFAULT  \_CONSUMER | consumer组名，多个consumer如果属于一个应用，订阅同样的消息，且消费逻辑一致，则应该将它们归为一组 |
| m\_messageModel | CLUSTERING | 消费模型：  支持集群消费和广播消费 |
| m\_consumeFromWhere | CONSUME\_FROM\_LAST\_OFFSET | consumer启动消费位置 |
| m\_pAllocateMessageQueueStrategy | 平均分配 | consumer队列自动分配策略：  平均分配；按照配置分配；按照机房来分配 |
| m\_pMessageListener | 无 | 消息监听器 |
| m\_consumeThreadMin | 10 | 消费线程池最少线程数量 |
| m\_consumeThreadMax | 20 | 消费线程池最大线程数量 |
| m\_consumeConcurrentlyMaxSpan | 2000 | 单队列并行消费允许最大跨度 |
| m\_pullThresholdForQueue | 1000 | 本地队列缓存消息最大数 |
| m\_pullInterval | 0 | 拉消息间隔 |
| m\_consumeMessageBatchMaxSize | 1 | 批量消费，一次消费的消息的最大数量 |
| m\_pullBatchSize | 32 | 批量拉消息，一次拉多少消息 |
| m\_subscription | 无 | 订阅关系。topic与tags对应存入map容器 |

### 配置方法

Pushconsumer的参数配置较为复杂，部分参数涉及函数调用，部分参数定义了宏。针对涉及函数调用的将详细说明；针对定义了多个宏作为参数的将列举出可选参数，并作出说明。其余简单参数配置只给定接口。

* m\_messageModel 可选参数

1. BROADCASTING。表示广播消费
2. CLUSTERING。表示集群消费

* m\_consumeFromWhere可选参数

1. CONSUME\_FROM\_LAST\_OFFSET。每次启动都从上次记录的位点开始消费，如果是第一次启动则从最大位点开始消费，建议在生产环境使用；
2. CONSUME\_FROM\_FIRST\_OFFSET。一个新的订阅组第一次启动从队列的最前位置开始消费，后续再启动接着上次消费的进度开始消费；

* m\_pAllocateMessageQueueStrategy队列分配算法，应用可重写。配置中可调用三种固有算法，也可以重写这些算法。

1. DefaultMQPushConsumer::setAllocateMessageQueueStrategy(AllocateMessageQueueStrategy\* pAllocateMessageQueueStrategy)

**入参列表**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **是否必须** | **描述** |
| pAllocateMessageQueueStrategy | AllocateMessageQueueStrategy | 是 | 该参数是队列分配算法指针，可以构造的指针有三种，后面会列举。 |

1. 系统中包含的三种写好的分配消息队列算法：

算法基类AllocateMessageQueueStrategy

平均分配队列算法class AllocateMessageQueueAveragely

按照配置分配队列算法class AllocateMessageQueueByConfig

按照机房分配队列算法class AllocateMessageQueueByMachineRoom

1. 配置方法

外部直接构造算法指针，通过封装的接口set将参数配置进去。

#头文件

int main(int argc, char\* argv[])

｛

······

DefaultMQPushConsumer consumer("MyConsumerGroup "); //构造consumer对象  
//构建算法指针，并赋值对应算法

AllocateMessageQueueStrategy\* m\_pAllocateMessageQueueStrategy；

m\_pAllocateMessageQueueStrategy = new AllocateMessageQueueAveragely()；

//通过set函数配置参数

consumer. setAllocateMessageQueueStrategy（m\_pAllocateMessageQueueStrategy）；

······

return 0

｝

* m\_pMessageListener消息监听器指针，用于回调消费消息。

配置方法：

1. 消息监听器是由用户在外部实现的，用于消费的类
2. 消息监听器继承父类MessageListenerOrderly 或 MessageListenerConcurrently
3. 在主流程中构造出消息监听器指针后，需要使用下面的接口注册

DefaultMQPushConsumer::registerMessageListener(MessageListener\* pMessageListener)

该接口在[3.3节](#_消费（被动push方式）)中已经说明。

* 下面列出对应接口函数名称，set与get一一对应，前者表示设置，后者表示获取。

|  |  |
| --- | --- |
| 参数名 | 接口名 |
| m\_consumerGroup | setConsumerGroup(const std::string& consumerGroup) getConsumerGroup() |
| m\_messageModel | setMessageModel(MessageModel messageModel)  getMessageModel() |
| m\_consumeFromWhere | setConsumeFromWhere(ConsumeFromWhere consumeFromWhere)  getConsumeFromWhere() |
| m\_pAllocateMessageQueueStrategy | setAllocateMessageQueueStrategy(AllocateMessageQueueStrategy\* pAllocateMessageQueueStrategy)  getAllocateMessageQueueStrategy() |
| m\_pMessageListener | setMessageListener(MessageListener\* pMessageListener)  getMessageListener() |
| m\_consumeThreadMin | setConsumeThreadMin(int consumeThreadMin)  getConsumeThreadMin() |
| m\_consumeThreadMax | setConsumeThreadMax(int consumeThreadMax)  getConsumeThreadMax() |
| m\_consumeConcurrentlyMaxSpan | setConsumeConcurrentlyMaxSpan(int consumeConcurrentlyMaxSpan)  getConsumeConcurrentlyMaxSpan() |
| m\_pullThresholdForQueue | setPullThresholdForQueue(int pullThresholdForQueue)  getPullThresholdForQueue() |
| m\_pullInterval | setPullInterval(long pullInterval)  getPullInterval() |
| m\_consumeMessageBatchMaxSize | setConsumeMessageBatchMaxSize(int consumeMessageBatchMaxSize)  getConsumeMessageBatchMaxSize() |
| m\_pullBatchSize | setPullBatchSize(int pullBatchSize)  getPullBatchSize() |
| m\_subscription | setSubscription(const std::map<std::string, std::string>& subscription)  getSubscription() |

### 举例说明

//在consumer端程序中设置consumer端配置的m\_consumeThreadMin

#头文件

int main(int argc, char\* argv[])

｛

······

DefaultMQPushConsumer consumer("MyConsumerGroup "); //构造consumer对象

consumer. setConsumeThreadMin(15); //调用接口setConsumeThreadMin，将15传入参数设置为m\_consumeThreadMin的值

······

return 0

｝