

Distributed Message Service for RocketMQ

Best Practices

Issue 01
Date 2025-05-06



Copyright © Huawei Cloud Computing Technologies Co., Ltd. 2025. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Cloud Computing Technologies Co., Ltd.

Trademarks and Permissions



HUAWEI and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Overview.....1

2 Deduplicating Messages Through Message Idempotence.....2

3 Classifying Messages with Topic and Tag.....4

4 Ensuring Subscription Consistency.....7

5 Handling Message Accumulation.....10

6 Configuring Message Accumulation Monitoring.....13

1 Overview

This section summarizes best practices of RocketMQ in common scenarios. Each practice is given a description and procedure.

Table 1-1 RocketMQ best practices

Best Practice	Description
Deduplicating Messages Through Message Idempotence	To avoid service exceptions caused by repeated message consumption, RocketMQ consumers receive a message and perform idempotent processing based on a unique key. This document contains the concept, scenario, and implementation of idempotent messages.
Classifying Messages with Topic and Tag	RocketMQ classifies services by topic or tag. Topics are superior to tags. This document describes how to filter messages by topic and tag.
Ensuring Subscription Consistency	Messages may be consumed repeatedly or missed due to disordered consumption logic caused by inconsistent subscriptions. This document describes the concept, principle, and implementation of subscription consistency.
Handling Message Accumulation	This document describes the causes and solutions of message stack.
Configuring Message Accumulation Monitoring	This document describes how to configure RocketMQ alarm rules on Cloud Eye. Cloud Eye can monitor the real-time instance running status and key service metrics, and notify exceptions in time. In this way, you are aware of risks in the production environment.

2 Deduplicating Messages Through Message Idempotence

Overview

In RocketMQ service processes, an idempotent message process refers to a situation where a message is re-sent and consumed for multiple times and each consumption result is the same, having no negative effects on services. Idempotent messages ensure consistency in the final processing results. Services are not affected no matter how many times a message is re-sent.

Take paying as an example. Assume that a user selects a product, makes payment, and receives multiple bills due to unstable Internet connection. The bills are all paid. However, the billing should take place only once and the merchant should generate only one order placement. In this case, idempotent messages can be used to avoid the repetition.

In actual applications, messages are re-sent because of intermittent network disconnections and client faults during message production or consumption. Message repetition can be classified into two scenarios.

- A producer repeatedly sends a message:
If a producer successfully sends a message to the server but does not receive a successful response due to an intermittent network disconnection, the producer determines that the message failed to be sent and tries resending the message. In this case, the server receives two messages of the same content. Consumers consume two messages of the same content.
- A consumer repeatedly consumes a message:
A message is successfully delivered to a consumer and processed. If the server does not receive a response from the consumer due to an intermittent network disconnection, the server determines that the message failed to be delivered. To ensure that the message is consumed at least once, the server retries delivering the message. As a result, the consumer consumes two messages of the same content.

Implementation

Messages with different IDs may have the same content, so the ID cannot be used as the unique identifier. RocketMQ supports idempotent messages by using the

message key (unique service identifier) to identify messages. The sample code for configuring a message key is as follows:

```
Message message = new Message();  
message.setKey("Order_id"); // Set the message key, which can be the unique service identifier such as  
the order placement ID.  
SentResult sendResult = mqProducer.send(message);
```

When a producer sends a message, the message has a unique key. When consuming the message, a consumer reads the unique message identifier (such as the order placement ID) with **getKeys()**. The service logic can implement idempotence with the unique identifier.

3

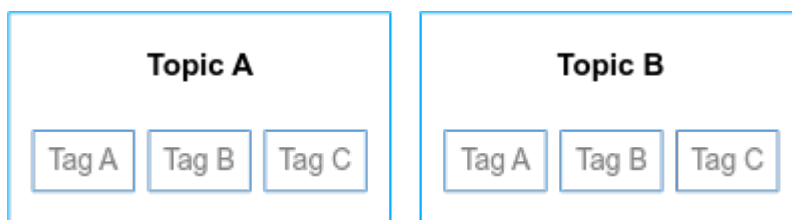
Classifying Messages with Topic and Tag

Overview

Topics are the basic logical unit of messages in message production and consumption. Each topic contains several messages and each message belongs to only one topic.

Tags are used to identify message of different types. Messages for different purposes in the same business unit can have different tags in the same topic. Tags ensure the clarity and coherency of code and facilitate query in RocketMQ. Consumers can implement different consumption logic for different topics based on tags to achieve better scalability.

Messages are first classified into topics and then with tags as shown in the following figure.



Scenario

Use topics and tags properly to ensure clear and efficient service structure. You can decide how to use topics and tags based on your needs.

- **Message type:** RocketMQ messages include normal, ordered, scheduled/delayed, and transactional messages. Different types of messages should be classified with topics, not tags.
- **Message priority:** Messages of a high priority should be in topics different from those with a low priority.
- **Service relationship:** Messages from unrelated services should be classified in topics. Messages from closely related services should be sent to the same topic, and classified with tags based on subtypes or sequence.

Implementation

Take logistics transportation as an example. Order messages of fresh goods and other goods are of different types, so they can be classified by two topics:

Topic_Common and **Topic_Fresh**. For each message type, you can use different tags to identify order destination provinces.

- Topic: Topic_Common
 - Tag = Province_A
 - Tag = Province_B
- Topic: Topic_Fresh
 - Tag = Province_A
 - Tag = Province_B

The following is message production sample code for a common goods order sent to province A:

```
Message msg = new Message("Topic_Common", "Province_A" /* Tag */, ("Order_id " +  
i).getBytes(RemotingHelper.DEFAULT_CHARSET));
```

The following is subscription sample code for a fresh goods order sent to province A and province B:

```
consumer.subscribe("Topic_Fresh", "Province_A || Province_B");
```

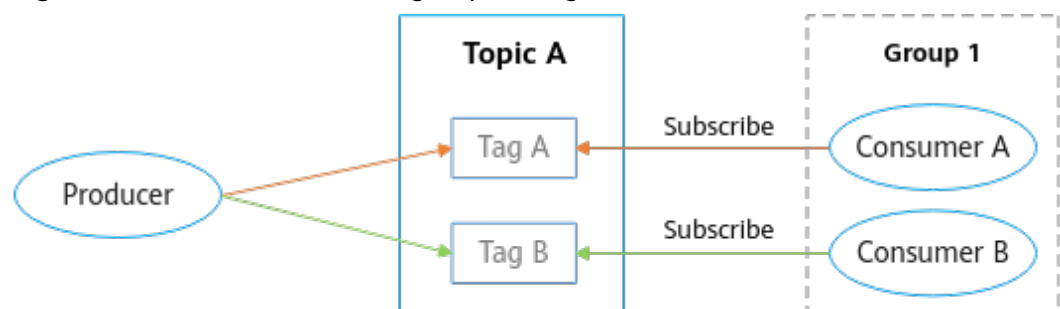
Different Consumers Consume Different Tags

Different consumers may consume messages with different tags in the same topic. For different tags in the same topic, improper consumer group settings lead to chaotic consumption.

For example, there are Tags A and B in Topic A. Consumer A subscribes to Tag A. Consumer B subscribes to Tag B.

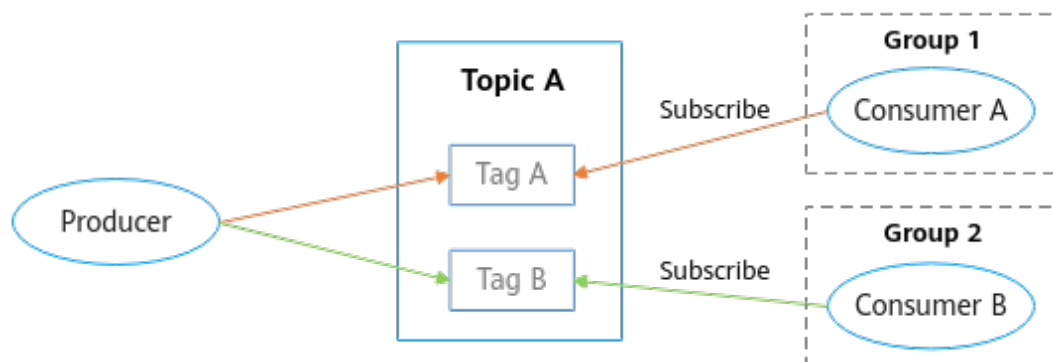
If Consumers A and B are in the same consumer group, messages with Tag A are evenly sent to Consumers A and B. Consumer B did not subscribe to Tag A, so it filters out messages with Tag A. As a result, some Tag A messages are not consumed.

Figure 3-1 Incorrect consumer group settings



To solve this problem, configure Consumers A and B with different consumer groups.

Figure 3-2 Correct consumer group settings



4 Ensuring Subscription Consistency

Overview

A consistent subscription indicates that all topics and tags subscribed by all consumers in the same consumer group are the same. An inconsistent subscription causes disordered consumption logic and even message losses.

Principle

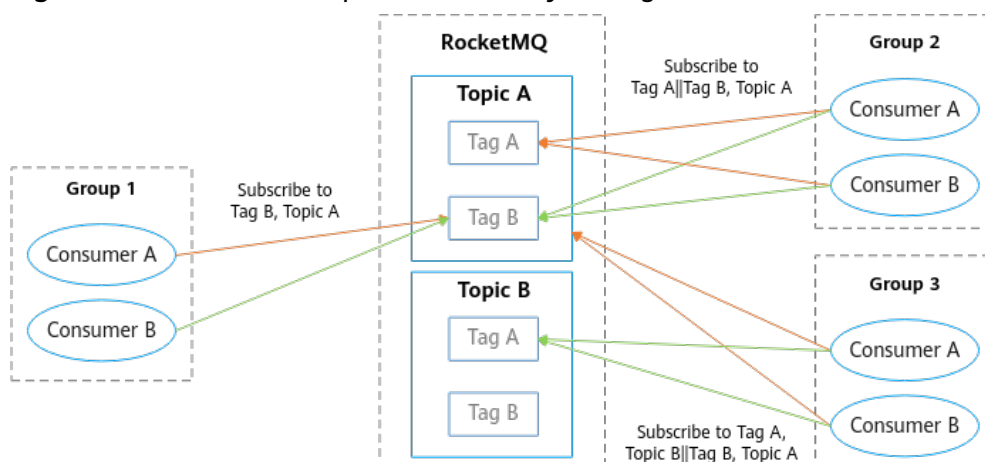
RocketMQ assigns message queues for each topic. The more queues, the higher the consumption concurrency. In distributed application scenarios, multiple consumers in the same consumer group jointly consume messages from all queues in a topic. Queues are assigned by consumer group and evenly assigned to the consumers in a consumer group, regardless of whether a consumer has subscribed to the topic. Each consumer is assigned some queues of a topic. Each queue is assigned to only one consumer.

Correct Subscription

In distributed application scenarios, all the consumers in a consumer group have the same consumer group ID. They must subscribe to the same topic and tag (consistent subscription) to ensure correct consumption logic and no message losses.

- Consumers in the same consumer group must subscribe to the same topic. For example, assume that Consumers A and B are in Consumer Group 1 and Consumer A subscribes to Topics A and B. Then, Consumer B must also subscribe to both Topics A and B, and cannot subscribe to only Topic A or B or even Topic C.
- The tags in the topic subscribed by consumers in the same consumer group must be the same, including the tag quantity and sequence. For example, assume that Consumers A and B are in Consumer Group 2. Consumer A subscribes to Tag A||Tag B in Topic A. Then, when subscribing to Topic A, Consumer B must also subscribe to Tag A||Tag B, and cannot subscribe only to Tag A or B or Tag B||Tag A.

Figure 4-1 Correct subscription consistency setting

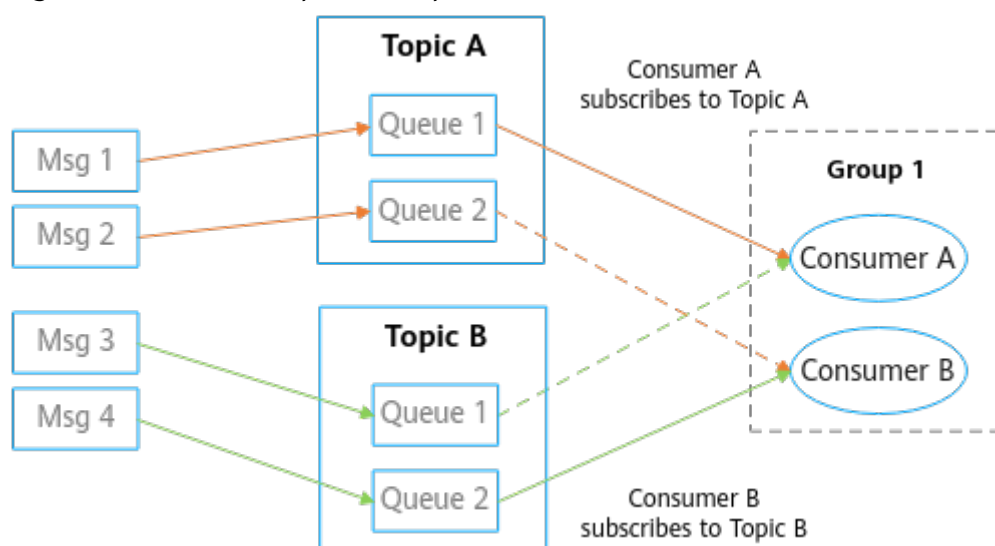


A consistent subscription ensures message consumption in the same consumer group work properly, avoiding disordered message logic or message losses. Producers should classify messages properly for consumers to correct subscription to tags. Consumers should ensure consistent subscriptions.

Incorrect Subscription

- Consumers in the same consumer group subscribe to different topics.
For example, assume that Consumers A and B are in Consumer Group 1. Consumer A subscribes to Topic A but Consumer B subscribes to Topic B. When producers send messages to Topic A, the messages are evenly sent to Consumers A and B by queue. Consumer B has not subscribed to Topic A, so it filters out messages from Topic A (Queue 2 in Topic A in [Figure 4-2](#)), leaving them unconsumed.

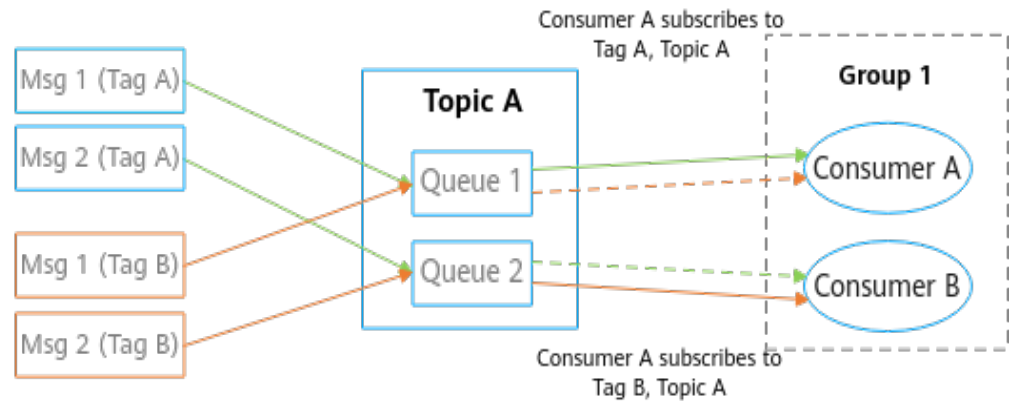
Figure 4-2 Incorrect topic subscriptions



- Consumers in the same consumer group subscribe to different tags of the same topic.
For example, assume that Consumers A and B are in Consumer Group 1. Consumer A subscribes to Tag A and Topic A. Consumer B subscribes to Tag B

and Topic A. When producer A sends messages to Tag A in Topic A, messages with Tag A are evenly sent to Consumers A and B by queue. Consumer B has not subscribed to Tag A, so it filters out messages with Tag A (Tag A in Queue 2 in [Figure 4-3](#)), leaving them unconsumed.

Figure 4-3 Incorrect tag subscriptions



Implementation

- **Subscriptions to One Tag of One Topic**

Consumers 1, 2, and 3 in Consumer Group 1 all subscribe to Tag_A and Topic_A. They have consistent subscriptions, meaning that their subscription code is the same. The sample code is as follows:

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("Group1");  
consumer.subscribe("Topic_A", "Tag_A");
```

- **Subscriptions to Multiple Tags of One Topic**

Consumers 1, 2, and 3 in Consumer Group 1 all subscribe to Tag_A and Tag_B of Topic_A. The sequence is Tag_A||Tag_B. The consumers have consistent subscriptions, meaning that their subscription code is the same. The sample code is as follows:

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("Group1");  
consumer.subscribe("Topic_A", "Tag_A||Tag_B");
```

- **Subscriptions to Multiple Tags of Multiple Topics**

Consumers 1, 2, and 3 in Consumer Group 1 all subscribe to Topic_A (no specified tag) and Topic_B (Tag_A and Tag_B). The sequence is Tag_A||Tag_B. The consumers have consistent subscriptions, meaning that their subscription code is the same. The sample code is as follows:

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("Group1");  
consumer.subscribe("Topic_A", "*");  
consumer.subscribe("Topic_B", "Tag_A||Tag_B");
```

5 Handling Message Accumulation

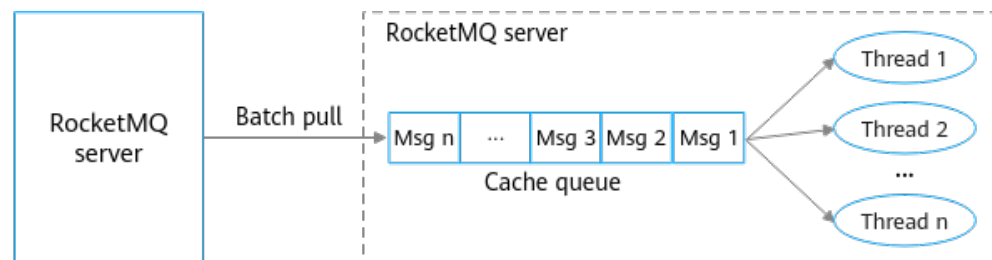
Overview

Message accumulation is common in RocketMQ services. Unprocessed messages accumulate if the client's consumption is slower than the server's sending. Accumulated messages cannot be consumed in time. Service systems with high requirements on real-time consumption cannot afford even a short message delay caused by message accumulation. Message accumulation causes are as follows:

- Messages are not consumed in time because message production is faster than consumption. Messages accumulate and consumption cannot be restored automatically.
- The service system logic is time-consuming, causing low consumption efficiency.

Message Consumption Process

Figure 5-1 Message consumption process



The message consumption process consists of two phases:

- **Message pull**
Clients pull messages from servers in batches and store the messages to local cache queues. In this phase, no messages accumulate because throughput is high on the intranet.
- **Message consumption**
Clients submit the cached messages to consumption threads, wait for the service consumption logic to process the messages, and receive the processing result. The consumption capability in this phase depends on the consumption duration and concurrency. The overall message throughput is affected if the

service logic is complicated and spends a long time on a single message. Low message throughput causes local cache queues on the client to reach the upper limit. Messages are no longer pulled from the server, resulting in accumulation.

Therefore, whether messages accumulate depends on the consumption capability of the client, and the consumption capability depends on the consumption duration and concurrency. Consumption time is prior to its concurrency. Users should ensure timely consumption before considering its concurrency.

Consumption Duration

Consumption duration is mainly affected by the service code, specially, the internal CPU computational code and the external I/O operational code. If there is no complex recursion or loop code, internal CPU computing duration can be ignored. Instead, you should focus on external I/O operations.

External I/O operations are as follows:

- Read/Write operations on external databases such as remote MySQL databases.
- Read/Write operations on external caches such as remote Redis.
- Invocations of downstream systems. For example, Dubbo invokes remote RPC and Spring Cloud invokes downstream HTTP APIs.

Learning about the downstream invoking logic helps you understand the duration of each invocation to determine whether the I/O operation duration in the service logic is proper. In general, faulty services or limited capacity in downstream systems causes longer consumption duration. Service faults can arise from network bandwidth issues as well as system errors.

Consumption Concurrency

The consumption concurrency on the client depends on number of clients (or consumers in a consumer group) and number of threads per client. The consumption concurrency of normal, scheduled/delayed, transactional, and ordered messages is calculated as follows.

Message Type	Concurrency Formula
Normal	Number of threads per client × Number of clients
Scheduled/Delayed	
Transactional	
Ordered	Min (Number of threads per client × Number of clients, Number of queues)

Note: The number of threads per client should be adjusted carefully. A large number of threads increases thread switch overhead.

An ideal calculation model for optimal number of threads per client: $C \times (T1+T2)/T1$.

C indicates the number of vCPUs per broker. T1 indicates the internal CPU computation duration. T2 indicates the external I/O operation duration. Thread switch overhead is ignored. I/O operations consume no CPU resources. A thread should have sufficient messages and memory for processing.

The model of calculating the maximum number of threads is only an ideal scenario. In actual scenarios, gradually increase threads based on the actual effect.

Implementation

To avoid unexpected message accumulation, the consumption duration should be accounted for and concurrency should be set properly in the design of service logic.

- **Accounting for consumption duration**

Perform pressure test to obtain the consumption duration. Analyze and optimize time-consuming service logic code. Pay attention to:

- Whether the computation of the consumption logic is too complex, and whether any complex recursions or loops exist in the code.
- Whether I/O operations are necessary in the consumption logic and whether local caches can be used instead.
- Whether the complicated, time-consuming operations in the consumption logic can be asynchronously processed.

- **Setting consumption concurrency**

Consumption concurrency calculation can be adjusted with the following methods:

- a. Obtain the ideal number of threads using a formula. Then, select a number smaller than the ideal number as the initial value. Increase threads per client gradually to find an optimal number of consumption threads and message throughput per client.
- b. Calculate the number of clients needed based on the upstream and downstream traffic peaks: $\text{Number of clients} = \text{Traffic peak} / \text{Message throughput per client}$.

6 Configuring Message Accumulation Monitoring

Overview

Unprocessed messages accumulate if the client's consumption is slower than the server's sending. Accumulated messages cannot be consumed in time.

Configure alarm rules so that you will be notified when the number of accumulated messages in a consumer group exceeds the threshold. The procedure described in this section can also be applied to setting alarm rules for [other metrics](#).

Prerequisites

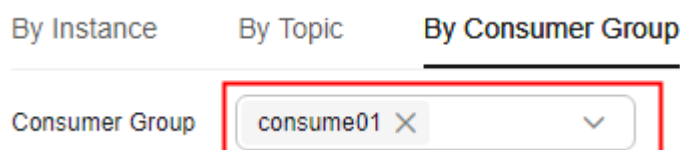
You have [purchased a RocketMQ instance](#), [created a topic](#), and there are available messages.

Procedure

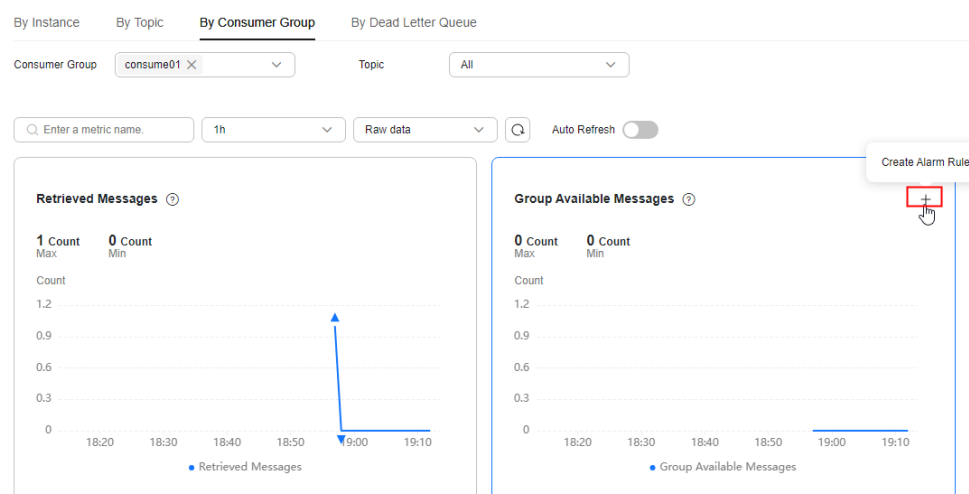
The screenshots in this section use the RocketMQ 5.x basic edition as an example.

- Step 1** Log in to the DMS for RocketMQ console and click the desired instance.
- Step 2** In the left navigation pane, choose **Monitoring And Alarm > Monitoring**.
- Step 3** On the **By Consumer Group** tab page, select the consumer group for which you want to create an alarm rule.

Figure 6-1 Selecting a consumer group



- Step 4** Hover the mouse pointer over **Consumer Available Messages** and click .

Figure 6-2 Consumer available messages chart


Step 5 On the **Create Alarm Rule** page, configure the basic information of the alarm rule. The alarm name can contain only letters, digits, underscores (_), and hyphens (-).

Figure 6-3 Configuring the basic information of the alarm rule


Step 6 Configure the alarm policy.

Alarm policy: A major alarm is generated if the number of raw data records is greater than or equal to 10,000 for one consecutive time. The alarm is notified once a day.

Figure 6-4 Configuring the alarm policy

Step 7 Configure the alarm notification.

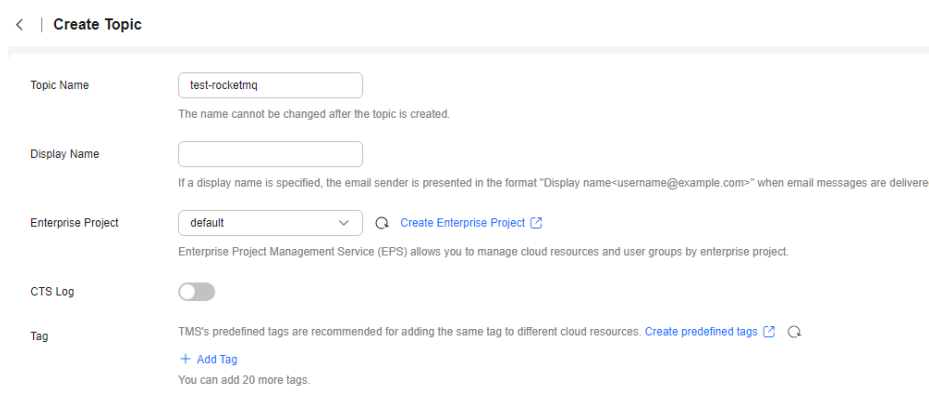
- **Alarm Notification:** Enable this option.
- **Notification Object:** Select a cloud account contact or a created alarm notification topic. An alarm notification topic contains the mobile number or email address receiving the notification.

If no alarm notification topics are available, click **Create an SMN topic**. On the SMN console, [create a topic](#) and [add a subscription](#). After the alarm notification topic is created, go back to the **Create Alarm Rule** page, click  to make the created topic available for selection.

NOTE

After the subscription is added, the corresponding subscription endpoint will receive a subscription notification. You need to confirm the subscription so that the endpoint can receive alarm notifications.

Figure 6-5 Creating an alarm notification topic



< | Create Topic

Topic Name
The name cannot be changed after the topic is created.

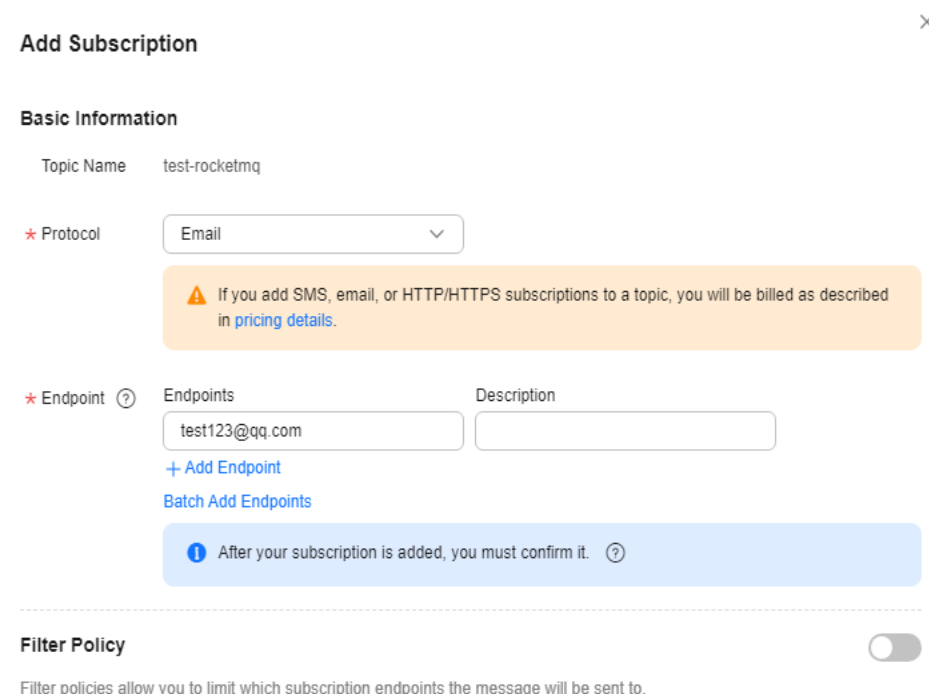
Display Name
If a display name is specified, the email sender is presented in the format "Display name<username@example.com>" when email messages are delivered.

Enterprise Project [Create Enterprise Project](#)
Enterprise Project Management Service (EPS) allows you to manage cloud resources and user groups by enterprise project.

CTS Log ☐

Tag
TMS's predefined tags are recommended for adding the same tag to different cloud resources. [Create predefined tags](#)
[+ Add Tag](#)
You can add 20 more tags.

Figure 6-6 Adding a subscription





Add Subscription

Basic Information

Topic Name test-rocketmq



* Protocol

 If you add SMS, email, or HTTP/HTTPS subscriptions to a topic, you will be billed as described in [pricing details](#).

* Endpoint  Endpoints Description

[+ Add Endpoint](#)

[Batch Add Endpoints](#)

 After your subscription is added, you must confirm it. 

Filter Policy ☐

Filter policies allow you to limit which subscription endpoints the message will be sent to.

- **Validity Period:** Cloud Eye sends notifications only within the validity period specified in the alarm rule.
- **Trigger Condition:** condition for triggering an alarm notification. **Generated alarm** and **Cleared alarm** are available.

Step 8 Configure the enterprise project and tag.

Figure 6-7 Configuring the enterprise project and tag.

The screenshot shows the 'Advanced Settings' configuration page for an alarm rule. The page has a breadcrumb 'Enterprise Project | Tag'. Under the 'Enterprise Project' section, there is a dropdown menu currently set to 'default' and a link 'Create Enterprise Project'. Below this, a text label says 'The enterprise project the alarm rule belongs to.' The 'Tag' section contains a recommendation to use predefined tags and a large text input area. Below the input area are three buttons: 'Enter a tag key', 'Enter a tag value', and 'Add'. At the bottom, it says 'Tags you can still add: 20'.

- **Enterprise Project:** enterprise project with which the alarm rule is associated. Only users who have the permissions of the enterprise project can view and manage the alarm rule.
- **Tag:** tags are used to identify cloud resources. When you have many cloud resources of the same type, you can use tags to classify cloud resources by dimension (for example, usage, owner, or environment).

Step 9 Click **Create**.

After the alarm rule is created, you can view it on the **Alarm Management > Alarm Rules** page.

-----End