# What is RabbitMQ 4

RabbitMQ 4 is a powerful **open source message broker software** (also known as message-oriented middleware), which implements the specification of the Advanced Message Queuing Protocol (AMQP) and has made many extensions on this basis.

The following is a detailed introduction to RabbitMQ 4:

## Overview

RabbitMQ is written in Erlang language, and its clustering and failover mechanisms are built on the open telecommunications platform framework. It supports multiple programming languages and provides a client library for communicating with the broker interface.


**What is RabbitMQ 4?**

RabbitMQ has been widely used in the field of messaging and middleware with its high reliability, flexible routing, message clustering, high-availability queues, multiple protocol support, multiple language clients, management interface, tracking mechanism and plug-in mechanism.

# Core components and architecture

The architecture of RabbitMQ consists of multiple core components, including producers (Publisher), switches (Exchange), queues (Queues), consumers (Consumer) and bindings (Bind).

## 1. Publisher:

- The source of the message.
- The producer sends the message to the exchange instead of directly to the queue.

## 2. Exchange:

- It can be understood as a routing rule with a routing table.
- Messages are routed to the corresponding queue according to the routing key.
- RabbitMQ provides multiple types of exchanges, such as fanout, direct, topic, etc.

## 3. Queues:

- A cache container for loading messages.
- Consumers take messages from the queue for processing.
- Queues can be persistent to ensure that messages are not lost after the RabbitMQ service stops or crashes.

## 4. Consumer:

- A client that connects to a queue and takes messages.
- Consumers can subscribe to one or more queues and process messages received from the queues.

## 5. Bind:

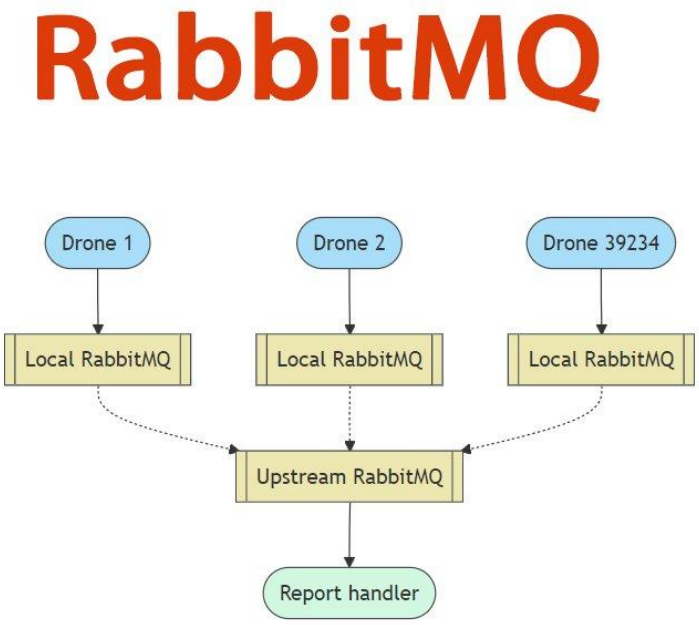
- It can actually be understood as the routing rule of the exchange.
- Each binding binds the switch, routing key and message delivery queue together to form a routing rule.

# Message delivery process

In RabbitMQ, the message delivery process is as follows:

1. The producer sends the message to the switch.
2. The switch routes the message to the corresponding queue according to the routing key.
3. The consumer takes the message from the queue for processing.

In this process, RabbitMQ provides a variety of mechanisms to ensure the reliable delivery of messages, such as persistence, transmission confirmation, and release confirmation.




**Mainstream messaging middleware RabbitMQ**

# Features and functions

## 1. Reliability:

- RabbitMQ uses some mechanisms to ensure reliability, such as persistence, transmission confirmation, and release confirmation.
- The persistence mechanism can store messages and queues on disk to ensure that messages are not lost after the RabbitMQ service stops or crashes.

## 2. Flexible Routing:

- RabbitMQ provides multiple types of switches to implement flexible routing functions.
- The switch can route messages to the corresponding queue according to the routing key.

## 3. Message Clustering:

- Multiple RabbitMQ servers can form a cluster to form a logical Broker.
- Servers in the cluster can share information such as queues, switches, and bindings.

## 4. Highly Available Queues:

- Queues can be mirrored on machines in the cluster so that queues are still available when some nodes have problems.
- Mirrored queues can improve the reliability and availability of the system.

## 5. Multi-protocol:

- RabbitMQ supports multiple message queue protocols, such as STOMP, MQTT, etc.
- This allows RabbitMQ to interoperate with multiple messaging systems and clients.

## 6. Many Clients:

- RabbitMQ supports almost all common languages, such as Java, .NET, Ruby, etc.
- This allows developers to write producers and consumers in languages they are familiar with.

## 7. Management UI:

- RabbitMQ provides an easy-to-use user interface that allows users to monitor and manage many aspects of the message broker.
- Users can view queues, switches, bindings and other information through the management interface, and can monitor the real-time status of message delivery.

## 8. Tracing:

- If the message is abnormal, RabbitMQ provides a message tracing mechanism so that users can find out what happened.
- This helps developers quickly locate and solve problems in message delivery.

## 9. Plugin System:

- RabbitMQ provides many plugins to extend it in many ways.
- Users can also edit their own plugins to meet specific needs.

**Rabbitmq 3.13.0 download**

# Queue Structure

The queue structure of RabbitMQ usually consists of two parts: AMQQueue and BackingQueue.

## 1. AMQQueue:

- Responsible for message processing related to the AMQP protocol, such as receiving messages published by producers, delivering messages to consumers, processing message confirms, acknowledges, etc.

## 2. BackingQueue:

- Provides relevant interfaces for AMQQueue to call, complete message storage and possible persistence, etc.
- BackingQueue is composed of multiple subqueues, such as Q1, Q2, Delta, Q3 and Q4. These subqueues play different roles in the message delivery process.

# Message Confirmation Mechanism

RabbitMQ provides a message confirmation mechanism to ensure that the message is successfully received and correctly processed by the consumer. When the consumer subscribes to the queue, the message confirmation mechanism can be controlled by setting the autoAck parameter.

## 1. Auto Acknowledgement:

- When the autoAck parameter is set to true, the consumer will automatically send a confirmation signal to RabbitMQ after processing the message.
- In this case, RabbitMQ will delete the message from the queue immediately after sending the message to the consumer.

### 2. Manual Acknowledgement:

- When the autoAck parameter is set to false, the consumer needs to manually send a confirmation signal to RabbitMQ.
- In this case, RabbitMQ will keep the message in the queue until it receives a confirmation signal from the consumer.
- If RabbitMQ does not receive a confirmation signal from the consumer for a long time and the consumer has been disconnected, RabbitMQ will put the message back into the queue and wait for redelivery to the next consumer.

## RabbitMQ architecture core components

### 1. Broker:

- A RabbitMQ Broker can be regarded as a RabbitMQ server for providing services to the outside world.
- Clients (producers and consumers) using RabbitMQ message middleware need to connect to Broker and use Rabbit's message queue service.

### 2. Virtual Host:

- Virtual Host is the virtual machine of Broker, which provides multi-tenant capabilities and realizes the separation of tenant permissions.

### 3. Publisher (message producer):

- The producer is the party that generates the message and sends the message to the exchange through the channel.

### 4. Connection:

- Connection is a TCP connection between the client and the RabbitMQ server.
- In the process of generating and consuming messages, the client does not need to establish multiple TCP connections with the server. It can reduce performance consumption by sharing the Channel (virtual TCP connection) in the Connection.

### 5. Exchange (message exchanger):

- The exchanger is the core component of RabbitMQ message routing.

- It receives messages sent by producers and forwards messages to the specified queue according to certain routing rules.
- There are many types of exchanges, such as Direct, Fanout, Topic, etc. Different types of exchanges have different routing strategies.

### 6. Binding:

- Binding is the rule for associating an exchanger with a message queue.
- Through binding, RabbitMQ can know how to route messages from the exchanger to the queue.

### 7. Queue (message queue):

- A queue is a container for storing messages and follows the first-in-first-out rule.
- Messages wait in the queue for consumers to consume.

### 8. Consumer:

- Consumers are the party that receives and processes messages.
- Consumers receive messages by subscribing to queues and get messages from queues for processing.

## Application scenarios

RabbitMQ is widely used in various messaging and middleware scenarios, such as asynchronous processing, application decoupling, traffic peak shaving, etc.

### 1. Asynchronous processing:

- RabbitMQ can process time-consuming tasks asynchronously, thereby improving the response speed and throughput of the system.

### 2. Application decoupling:

- RabbitMQ can decouple different application modules so that they can be developed and deployed independently.

### 3. Traffic peak shaving:

- RabbitMQ can cache a large number of messages and process them when the system load is low, thereby avoiding system crashes due to overload.

# Installation and configuration

The installation and configuration of RabbitMQ is relatively simple. Users can download and install the version suitable for their operating system from the RabbitMQ official website. After the installation is complete, users also need to perform operations such as environment configuration and plug-in activation.

## 1. Download and install:

- Users can download the installation package for their operating system from the RabbitMQ official website.
- Follow the prompts of the installation wizard to install.

## 2. Environment configuration:

- Configure system environment variables to conveniently run RabbitMQ related commands in the command line.
- Configure RabbitMQ configuration files, such as rabbitmq.conf, to meet specific needs.

## 3. Plugin enable:

- RabbitMQ provides many plugins to extend its functionality. Users can enable the corresponding plugins according to their needs.
- For example, you can enable the rabbitmq_management plugin to enable the management interface function.

**RabbitMQ Architecture**

# Usage example

The following is a simple RabbitMQ usage example, including the code implementation of the producer and consumer.

## 1. Producer code:

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class Producer {
private final static String QUEUE_NAME = "hello";

public static void main(String[] argv) throws Exception {
// Create a connection factory
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
try (Connection connection = factory.newConnection();
Channel channel = connection.createChannel()) {
// Declare a queue
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
// Send a message
String message = "Hello World!";
channel.basicPublish("", QUEUE_NAME, null, message.getBytes("UTF-8"));
System.out.println(" [x] Sent '" + message + "'");
}
}
}
```

## 2. Consumer code:

```
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer {
private final static String QUEUE_NAME = "hello";

public static void main(String[] argv) throws Exception {
// Create a connection factory
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
try (Connection connection = factory.newConnection();
Channel channel = connection.createChannel()) {
// Declare a queue
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
// Create a consumer and subscribe to a queue
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
String message = new String(delivery.getBody(), "UTF-8");
System.out.println(" [x] Received '" + message + "'");
```

```
};
channel.basicConsume(QUEUE_NAME, true, deliverCallback, consumerTag ->
{ });
}
}
}
```

In this example, the producer sends messages to a queue named "hello", and the consumer subscribes to the queue and processes the messages received from the queue.

# What is RabbitMQ 4 good for?

Its main uses can be summarized as follows:

## Application decoupling

RabbitMQ implements a loosely coupled architecture between applications through message queues. Each application can send and receive messages through message queues without knowing the specific implementation details of other applications. This decoupling method improves the maintainability, scalability, and fault tolerance of the system.

For example, in an order processing system, the order system can write information about reconstructed inventory to RabbitMQ. Even if the inventory system is temporarily unavailable, the order system can continue to process orders and consume messages from RabbitMQ to update inventory after the inventory system is restored.

## Asynchronous speed-up

RabbitMQ supports asynchronous message processing, which can significantly improve the response speed and throughput of the system. By converting time-consuming operations into asynchronous execution, applications can continue to perform other tasks while waiting for an operation to complete, without blocking and waiting for the result to return. For example, in a network request scenario, the synchronous method will cause the application to be blocked until the response returns before continuing to perform other tasks; while the asynchronous method allows the application to continue to perform other tasks, thereby improving user experience and system performance.

## Peak-shaving and valley-filling

RabbitMQ achieves the effect of peak-shaving and valley-filling by balancing the system load. When the system load fluctuates greatly, RabbitMQ can temporarily store messages generated during the peak period and continue to process these messages for a period of time after the peak period, thereby making rational use of resources and ensuring the stability and efficiency of the system. This mechanism is of great significance for handling sudden traffic and avoiding system crashes.

## Message Distribution

RabbitMQ provides a flexible message distribution mechanism that allows different components or modules to communicate through messages. The publisher sends messages to RabbitMQ without knowing the specific recipient of the message; the subscriber receives the message by subscribing to a specific message or message type. This mechanism achieves loose coupling and scalability of the system, making it easier for the system to adapt to changes.

## High Reliability

RabbitMQ ensures the reliability of message delivery through mechanisms such as persistence, transmission confirmation, and publishing confirmation. Data persistence ensures that messages will not be lost when the server goes down, and continue to be received and processed by consumers after the server is restored. In addition, RabbitMQ also supports cluster mode, achieving high availability and load balancing, further improving the reliability of the system.

## Flexible Routing

RabbitMQ provides a variety of exchange types and binding rules, allowing messages to be flexibly routed to specified queues. This includes direct exchanges (Direct Exchange), fanout exchanges (Fanout Exchange) and topic exchanges (Topic Exchange). By configuring different exchange types and binding rules, complex message routing logic can be implemented to meet different business needs.

## Multi-language support

RabbitMQ supports clients in multiple programming languages, including Java, Python, Ruby, C#, Node.js, etc. This makes it easier for developers to integrate RabbitMQ into various applications, improving development efficiency and system flexibility.

# Rabbitmq 4 python

RabbitMQ is an open source implementation of AMQP (Advanced Message Queue Protocol) developed by Erlang language, which provides a reliable messaging mechanism.

## RabbitMQ working mode

RabbitMQ can be used in Python through the `pika` library. Its common working modes include:

1. **Simple mode**: A producer sends a message to a queue, and a consumer receives messages from the queue. This is the most basic messaging mode.
2. **Work queue mode**: Distribute tasks (messages) to multiple consumers (work threads). This mode is suitable for scenarios where a large number of tasks need to be processed in parallel.
3. **Publish/subscribe mode**: A producer sends a message to a switch, and all queues bound to this switch will receive the message. This mode is suitable for scenarios where messages need to be broadcast to multiple consumers.
4. **Routing mode**: A routing key is added on the basis of the publish/subscribe mode. The producer adds the routing key when publishing a message, and the consumer also adds the routing key when binding the queue to the switch, so that the corresponding message can be received. This mode allows messages to be sent to the specified consumer more accurately.

## RabbitMQ usage examples in Python

Here is a Python example using RabbitMQ routing mode:

```
import pika

# Establish connection and channel
connection =
pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

# Declare a direct type exchange
channel.exchange_declare(exchange='direct_logs',
exchange_type='direct')

# Producer sends message
severity = ['info', 'warning', 'error']
```

```
for i in range(20):
message = f'{i} Hello World! {severity[i % 3]}'
channel.basic_publish(exchange='direct_logs', routing_key=severity[i %
3], body=message)
print(f"[x] Sent: {message}")

# Close the connection
connection.close()
```

For the consumer part, take receiving `info` level messages as an example:

```
import pika

# Establish a connection and channel
connection =
pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

# Declare the exchange (if the producer has already declared it, this step
can be omitted)
channel.exchange_declare(exchange='direct_logs',
exchange_type='direct')

# Declare a queue and set it to exclusive (exclusive=True)
result = channel.queue_declare(exclusive=True)
queue_name = result.method.queue

# Bind the queue to the exchange and specify the routing key as 'info'
channel.queue_bind(exchange='direct_logs', queue=queue_name,
routing_key='info')

# Define a callback function to process the received message
def callback(ch, method, properties, body):
print(f"[x] {method.routing_key}:{body}")

# Start consuming messages
channel.basic_consume(callback, queue=queue_name, no_ack=True)
print('[*] Waiting for logs. To exit press CTRL+C')
channel.start_consuming()
```

In this example, the producer sends messages containing different levels (info, warning, error) to the `direct_logs` exchange, while the consumer only receives and processes messages at the `info` level.

## Features of RabbitMQ

1. **Reliability**: RabbitMQ uses mechanisms such as persistence, transmission confirmation, and publication confirmation to ensure the reliability of messages.
2. **Flexible routing**: The message routing function is implemented through Exchange, supporting multiple routing modes.
3. **Message cluster**: Multiple RabbitMQ servers can form a cluster to form a logical Broker.
4. **High availability**: Queues can be mirrored on machines in the cluster, so that queues are still available when some nodes have problems.
5. **Multiple protocol support**: RabbitMQ supports multiple message queue protocols, such as STOMP, MQTT, etc.
6. **Multi-language client support**: RabbitMQ supports almost all common languages, such as Java, .NET, Ruby, Python, etc.
7. **Management interface**: RabbitMQ provides an easy-to-use user interface that allows users to monitor and manage many aspects of the message broker.

# Rabbitmq vs kafka

RabbitMQ and Kafka are both popular message middleware, each with unique characteristics and applicable scenarios.


Rabbitmq Vs kafka

The following is a detailed introduction to the two:

## RabbitMQ

### Overview:

- RabbitMQ is a reusable enterprise messaging system implemented on the basis of AMQP (Advanced Message Queuing Protocol).
- It supports high concurrency, scalability, and provides multiple client support (such as Python, Java, C#, etc.) and persistence capabilities.

### Core components:

- **Broker**: message queue server entity.
- **Exchange**: message switch, specifies the rules by which messages are routed to which queue.
- **Queue**: message queue carrier, each message will be put into one or more queues.
- **Binding**: bind Exchange and Queue according to routing rules.
- **Routing Key**: routing keyword, Exchange delivers messages based on this keyword.

### Features:

- **Reliability**: ensure message reliability through mirror queue, persistence and other functions.
- **Flexible routing**: support multiple switch types and binding rules to achieve complex message routing.
- **Scalability**: support cluster deployment, can be expanded horizontally or vertically.
- **High availability**: ensure high availability of queues through mirror queue mechanism.

### Applicable scenarios:

- Applicable to real-time message delivery scenarios with high reliability requirements.
- Applicable to scenarios that require complex message routing and distribution strategies.
- Suitable for small and medium-sized projects, scenarios with small message volume, low throughput, and delay sensitivity.

## Kafka

### Overview:

- Kafka is a distributed stream processing framework developed in Scala language, mainly used to process active streaming data and large data volumes.
- It adopts a publish-subscribe model and supports batch processing and high throughput of messages.

### Core components:

- **Producer**: message producer, responsible for sending messages to Kafka topics.
- **Broker**: nodes in the Kafka cluster, responsible for storing and forwarding messages.
- **Consumer**: message consumer, pulls messages from Kafka topics for processing.
- **Topic**: logical message classification, a Topic can be divided into multiple partitions.
- **Partition**: the smallest unit for storing messages in Kafka, each Partition is an ordered, immutable sequence of messages.

### Features:

- **High throughput**: Data storage and retrieval are local disk sequential batch operations with huge throughput.
- **Persistence**: Messages are persisted to disk, ensuring data integrity even if the system crashes.
- **Partition and copy mechanism**: Horizontal expansion is achieved through partitioning, and data reliability is guaranteed through copy mechanism.
- **Real-time**: Supports real-time data stream processing and has low latency characteristics.

### Applicable scenarios:

- Applicable to large-scale data stream processing and real-time data processing scenarios.
- Applicable to scenarios that require high throughput and persistence characteristics.
- Applicable to scenarios such as log collection, user activity tracking, and operational indicator monitoring.

## Comparative analysis

### Architecture model:

- RabbitMQ is Broker-centric and adopts a push model.
- Kafka is Consumer-centric and adopts a pull model.

### Message confirmation mechanism:

- RabbitMQ has a producer confirm mechanism and a consumer message response mechanism (ack), which can ensure reliable message delivery.
- Kafka does not have a response mechanism. Consumers maintain offsets to record messages that have been consumed.

### Message order:

- In RabbitMQ, messages in a queue are strictly ordered (FIFO).

- In Kafka, messages in each Partition are ordered, but overall (across Partitions) are unordered.

### Throughput:

- RabbitMQ's throughput is relatively small, limited by the performance and capacity of a single node.
- Kafka has a huge throughput, which can reach millions per second.

### Persistence:

- RabbitMQ can persist data to memory or hard disk.
- Kafka persists data to disk and supports data compression and batch transmission to improve performance.

In summary, RabbitMQ and Kafka have significant differences in architecture model, message confirmation mechanism, message order, throughput and persistence. When choosing to use, you need to weigh the two based on your own needs and the characteristics and limitations of the two.

## RabbitMQ alternatives

As a complete and highly reliable enterprise messaging system based on the AMQP standard, RabbitMQ has a wide user base in the market.

However, with the continuous development of technology and the diversification of application scenarios, some alternatives have gradually emerged. The following is a detailed introduction to RabbitMQ alternatives:

### Mosquitto

- **Overview**: Mosquitto is an open source message broker that implements the MQTT protocol. It is lightweight and suitable for all devices from low-power single-board computers to full-featured servers.
- **Features**:
- Supports MQTT protocol versions 5.0, 3.1.1, and 3.1.
- Cross-platform support, including Windows, Mac, and Linux.
- Open source and free.
- **Applicable scenarios**: Applicable to scenarios that require lightweight messaging and MQTT protocol support.

### Apache Pulsar

- **Overview**: Apache Pulsar is a distributed, open source publish-subscribe messaging and stream processing platform for real-time workloads.
- **Features**:
- Supports publish-subscribe and queue messaging modes.
- Provides stream processing capabilities for real-time data analysis.
- Self-hosted, with high flexibility.
- **Applicable scenarios**: Applicable to scenarios that require real-time data processing and stream processing capabilities.

## NSQ

- **Overview**: NSQ is a real-time distributed messaging platform designed to run at scale.
- **Features**:
- Distributed architecture, supports horizontal expansion.
- High throughput, processing billions of messages per day.
- Open source and free.
- **Applicable scenarios**: Applicable to scenarios that require large-scale messaging and high throughput.

## ØMQ (ZeroMQ)

- **Overview**: ØMQ is a high-performance asynchronous messaging library designed for distributed or concurrent applications.
- **Features**:
- Provides message queue capabilities without the need for a dedicated message broker.
- Supports multiple programming languages and platforms.
- Has features such as distribution, multicast, and scalability.
- **Applicable scenarios**: Applicable to scenarios that require high-performance asynchronous messaging and distributed system integration.

## Apache ActiveMQ

- **Overview**: Apache ActiveMQ is a popular and powerful open source messaging and integration mode server.
- **Features**:
- Supports multiple messaging protocols (such as JMS, AMQP, STOMP, etc.).
- Mature community and good stability.
- Applicable to small projects or resource-sensitive scenarios.
- **Applicable scenarios**: Applicable to scenarios that require support for multiple messaging protocols and community support.

### Redis (in specific scenarios)

- **Overview**: Redis is a high-performance key-value storage database that also supports message queue functions.
- **Features**:
- Simplified deployment and operation and maintenance, especially in projects that already use Redis.
- Relatively low resource consumption and low deployment difficulty.
- Applicable to simple message publishing and subscription functions.
- **Applicable scenarios**: Redis can be used as a substitute for RabbitMQ when the message queue usage scenario is relatively simple and the message reliability requirement is not high.

## About IoT cloud platform

IoT Cloud Platform (blog.iotcloudplatform.com) focuses on the Internet of Things middleware, IoT solution, IoT design, IoT programming, security IoT, industrial IoT, military Internet of Things, the best Internet of Things project, IoT company, IoT company, IoT company , Chinese IoT Corporation, IoT Corporation, Top IoT, IoT Module, Embedded Development, IoT Circuit Board, Raspberry Pi Development Design, Arduino Programming, Programming Language, RFID, LoRa Equipment, IoT System, Sensor, temperature and humidity Sensor, liquid level sensor, sensor equipment, artificial intelligence, blockchain, machine arm, smart home, smart city, smart agricultural factory, marginal computing, big data, cloud computing, brain machine interface, machine learning, robotics, VR/AR, VR/AR , AI simulation technology, exercise control, new energy, photovoltaic solar energy, lithium battery, silicon brain SBB, unmanned aerospace navigation, driverless, AGI, chip, semiconductor, intelligent hardware and other technological knowledge and technology products.

## FAQs

Here are some common questions and answers about RabbitMQ:

### What is RabbitMQ?

RabbitMQ is a message queue technology that uses the AMQP advanced message queue protocol. Its feature is that consumption does not need to ensure the existence of the provider, which achieves a high degree of decoupling between services.

### What language is the RabbitMQ server written in?

RabbitMQ server is written in **Erlang language**.

## Why use RabbitMQ?

RabbitMQ has a series of advanced features such as asynchrony, peak shaving, and load balancing in a distributed system; it has a persistence mechanism, and process messages and information in the queue can also be saved; it achieves decoupling between consumers and producers; for high-concurrency scenarios, using message queues can turn synchronous access into serial access, achieve a certain amount of current limiting, and facilitate database operations.

## How does RabbitMQ ensure that messages are sent and received correctly?

Set the channel to confirm mode (sender confirmation mode), and all messages published on the channel will be assigned a unique ID. After the message is delivered to the destination queue or the message is written to disk (persistent messages), the channel will send a confirmation to the producer (including the message unique ID). Consumers must confirm each message after receiving it. Only when consumers confirm the message can RabbitMQ safely delete the message from the queue.

## How does RabbitMQ ensure that messages are not lost?

RabbitMQ ensures that persistent messages can recover from server restarts by writing them to a persistent log file on disk. When publishing a persistent message to a persistent exchange, RabbitMQ will send a response only after the message is committed to the log file. In addition, configuring mirrored queues and mirroring queues to multiple nodes can also improve the reliability of messages.

## What are the working modes of RabbitMQ?

RabbitMQ's working modes include Simple mode (the simplest sending and receiving mode), Work mode (one queue corresponds to multiple consumers), Publish/Subscribe mode (one exchange corresponds to multiple queues), Routing mode (delivering messages to one or more queues according to routing keys), Topics mode (similar to routing mode, but supports fuzzy matching routing keys), etc.

## How does RabbitMQ handle the order of messages?

RabbitMQ itself does not provide the function of sequential messages. If you want to ensure the order of messages, you can ensure that a group of ordered messages are sent to the same queue in sequence at the sending end, and then when the consumer consumes this queue, it is guaranteed that only a single consumer instance consumes the messages on this queue. This is usually achieved through a single producer instance + a single queue + a single consumer instance, but it will

seriously affect the performance and throughput of MQ and needs to be carefully considered.

## What are the disadvantages of RabbitMQ?

The disadvantages of RabbitMQ include reduced system availability (the more external dependencies are introduced, the easier it is for the system to crash), increased system complexity (it is necessary to deal with issues such as repeated message consumption, message loss, and message delivery order), and consistency issues (such as system A returns success directly after processing, but one of the three systems BCD fails to write to the database, resulting in inconsistent data).

## What are the possible reasons for RabbitMQ connection failure?

Possible reasons for RabbitMQ connection failure include RabbitMQ service not started or stopped, firewall or network configuration blocking RabbitMQ port, RabbitMQ service crash or abnormality resulting in inability to accept connection, incorrect listeners settings in RabbitMQ configuration, etc.

## What is rabbitmq 4 server?

RabbitMQ 4 server is a powerful, flexible and reliable message broker software suitable for various distributed systems and application scenarios. By using RabbitMQ reasonably, the performance, scalability and maintainability of the system can be significantly improved.

## What is the use of RabbitMQ?

**RabbitMQ is mainly used in four aspects:** application decoupling, asynchronous speed-up, peak-to-valley filling, and message distribution. Through application decoupling, the direct dependency between applications can be reduced, and the maintainability, scalability and fault tolerance of the system can be improved. Asynchronous speed-up is to improve the response speed and throughput of the system by converting time-consuming operations into asynchronous execution. Peak-to-valley filling can balance the system load and ensure the stability and efficiency of the system. Message distribution achieves decoupling and flexibility, allowing different components or modules to communicate through messages.

## What is the difference between RabbitMQ and Kafka?

The main differences between RabbitMQ and Kafka are as follows:
**Throughput**: Kafka's throughput is usually higher, thanks to its optimizations such as Zero Copy mechanism, disk sequential read and write, batch processing mechanism, and partition mechanism.
**Reliability**: RabbitMQ has better reliability. It provides multiple message

confirmation mechanisms, such as transactions and sender confirmation, and also supports message persistence.
**Model**: RabbitMQ follows the AMQP protocol, is broker-centric, and has multiple message models such as simple queue mode and work queue mode. Kafka does not follow AMQP, is consumer-centric, and messages are nominally stored permanently.
**Responsible for balancing**: Kafka achieves load balancing through zk and partitioning mechanisms, while RabbitMQ achieves high availability and load balancing through clusters and mirror queues.

## What are the advantages and disadvantages of RabbitMQ and Redis?

**Advantages and disadvantages of RabbitMQ**:
**Advantages**:
High reliability, providing multiple message confirmation mechanisms and message persistence.
Flexible message model, supporting multiple message delivery modes.
High concurrency performance, able to handle a large number of concurrent messages.
Cluster and distributed support, achieving high availability and load balancing of message queues.
**Disadvantages**:
Relatively high complexity, complex configuration and use.
High performance overhead, rich functions will bring performance overhead to a certain extent.
Cluster maintenance is difficult, and network communication and message synchronization between nodes need to be considered.
**Advantages and Disadvantages of Redis**:
**Advantages**:
Excellent read and write performance, with a read speed of 110,000 times/s and a write speed of 81,000 times/s.
Supports a variety of data types, such as list, set, sorted set, hash, etc.
Supports transactions, all operations are atomic.
Supports data persistence, and can save data in memory to disk through AOF and RDB.
**Disadvantages**:
Limited database capacity, limited by physical memory, cannot be used for high-performance reading and writing of massive data.
No automatic fault tolerance and recovery functions, when the host or slave crashes, some read and write requests will fail.
It is difficult to support online expansion, and online expansion becomes very complicated when the cluster capacity reaches the upper limit.

## What are the common commands of RabbitMQ?

Common commands of RabbitMQ include:
**rabbitmqctl**: This is the core management tool of RabbitMQ, used for managing nodes, users, queues, permissions, etc. For example, start and stop RabbitMQ applications (rabbitmqctl start_app, rabbitmqctl stop_app), view cluster status (rabbitmqctl cluster_status), list all queues (rabbitmqctl list_queues), etc.
**rabbitmq-diagnostics**: used to diagnose the health status and performance problems of RabbitMQ nodes. For example, view node status (rabbitmq-diagnostics status), check cluster status (rabbitmq-diagnostics cluster_status), view process memory usage (rabbitmq-diagnostics memory_breakdown), etc.
**rabbitmq-plugins**: used to manage RabbitMQ plug-ins, which can extend the functions of RabbitMQ. For example, enable or disable a specified plugin (rabbitmq-plugins enable plugin_name, rabbitmq-plugins disable plugin_name).
**rabbitmqadmin**: A tool based on HTTP API for managing RabbitMQ resources. For example, declare queues, exchanges, bindings, etc. (rabbitmqadmin declare queue, rabbitmqadmin declare exchange, rabbitmqadmin declare binding), publish messages to specified exchanges and routing keys (rabbitmqadmin publish), get messages from queues (rabbitmqadmin get queue), etc.