Team Group Name

Team Member： Bo Tian

Student Number： 20211348

Synopsis:

Describe the system you intend to create:

What I want to build is an ordering system. The service objects of this system include members who appear in the ordering and delivery scene, including customers, third-party food delivery platforms, restaurants, delivery people, and settlement centers.

What is the application domain?

This is a system that is common in the catering industry and is used in ordering and delivery scenarios. Common software includes Deliveroo, UberEates, etc.

What will the application do?

Generally speaking, this application connects several roles that appear in the takeaway scene. In the whole system, each participating member performs their own duties, and finally realizes the user's remote order.

Specifically, the user orders the restaurant on the app, and the restaurant notifies the delivery man after confirming the information. When the delivery man successfully delivers the product, the user can accumulate points, and the settlement center will settle the order at the same time.

I want this system to have the following characteristics:

(1) Users can order smoothly

(2) The application requires less resource overhead

(3) The application can still maintain good performance during the peak period

(4) The order will not go wrong

Technology Stack

List of the main distribution technologies you will use

- *Main Tech : RabbitMQ*

Middleware is described as providing applications with services other than those provided by the operating system, simplifying application communication, input and output development, and allowing them to focus on their own business logic.

Messaging middleware is suitable for distributed environments that require reliable data transmission. In a system using the message middleware mechanism, different objects activate each other's events by passing messages to complete corresponding operations. The sender sends the message to the message server, and the message server stores the message in several queues, and then forwards the message to the receiver when appropriate. Message middleware can communicate between different platforms. It is often used to shield the characteristics between various platforms and protocols, and realize the collaboration between applications. Its advantage is that it can

provide synchronous and asynchronous between clients and servers. connection, and messages can be transmitted or stored and forwarded at any time.

Highlight why you used the chosen set of technologies and what was it about each technology that made you want to use it.

Reasons for using messaging middleware:

Let's first look at the comparison of several business processes.

(1) Synchronous direct call

The flow of this method is as follows,

User—>order service—>restaurant service—>rider service—>settlement service—>reward service—>user (successful order).

Its problem is:

A. The business call chain is too long, so the waiting time for users is long. The user must wait for all services to successfully process the information before getting feedback.

B. Partial component failure can paralyze the entire business.

c. There is no buffer during peak business hours.

(2) Asynchronous call

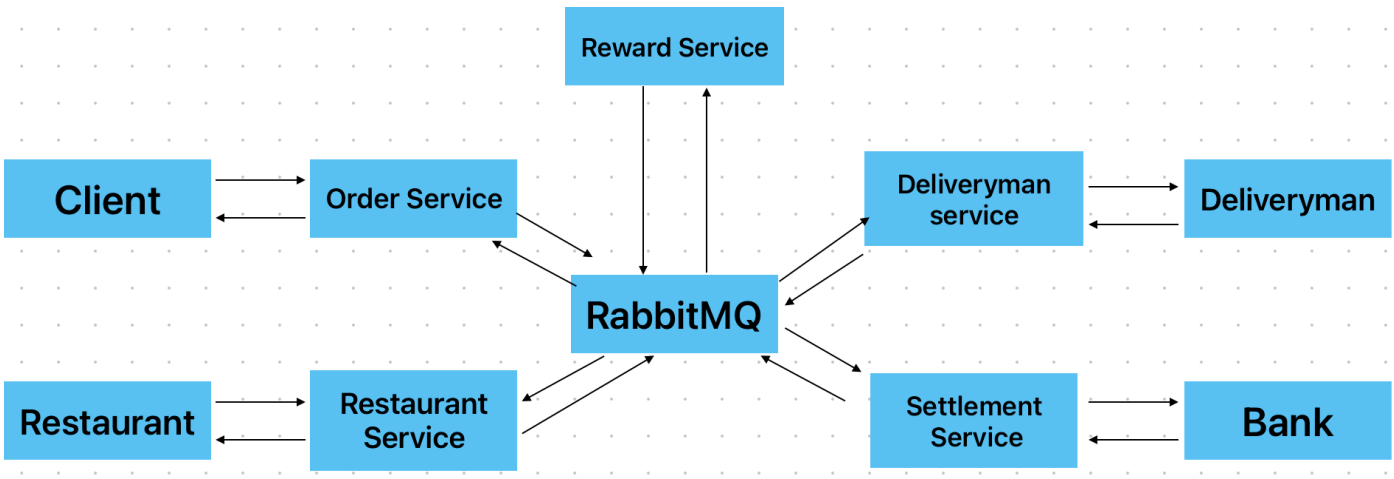
The process of this method may be like this, user -> order service -> user (successful order),

Order service—>rider service—>settlement service—>point service—>user.

In this way, the user does not need to wait too long, and the order service will directly feedback the user. In addition, if we adopt an asynchronous approach, the failure of some components may not paralyze the entire business. There is also a certain buffer during the peak period of business, and our orders can be cached in asynchronous threads.

It sort of solves the above ABC problem. But it also brings new problems, that is, a large number of asynchronous threads will be generated during peak business hours. If the thread processing is not timely and there is a large number of queuing situations, the thread pool will not be enough or the memory will be full.

(3) Use message middleware



All services do not pass messages directly, but pass through message middleware. For example, when the user sends a request to the Order service, the Order Service will directly return a message to the user. At this time, the thread has ended, and the management of the message is handed over to rabbitmq for management. In this way, a large number of asynchronous threads will not be generated.

In summary, the advantages of using message middleware are:

(1) Asynchronous processing: It can be understood from the above description that each service does not need to depend on the operation of other services, but only needs to process its own business. This makes the service call chain short and the user's waiting time short.

(2) System decoupling: As can be seen from our business flow chart, we have sufficiently divided the entire system into several independent service modules, which is very beneficial for development. If there are multiple members on the team, each member only needs to focus on his own part of the business. And the failure of some components will not paralyze the entire business.

(3) Traffic peak clipping: In the case of no need to consume a lot of system resources (a large number of threads), each thread ends after processing a message without waiting.

(4) Message broadcast: When a service needs to broadcast a message to multiple services, it does not need to actually connect with other services, but directly sends the message to the middleware, and allows the middleware to broadcast through certain configurations. In a sense, this is also good for the decoupling of the system.

(5) Message collection: When multiple microservices send information to a service, we don't need to actually establish a connection between the services, but hand it over to the message middleware for management. In a sense, this is also good for the decoupling of the system.

(6) Final consistency: Although we split the system into many modules, we don't need to worry about the final consistency of the data because the messages are managed by the middleware.

Reasons for choosing rabbitmq:

Comparison of mainstream message middleware technologies:

1. *Apache ActiveMQ*

features：

(1) Produced by Apache, developed by java.

(2) Support jms1.1 protocol and j2 er er 1.4 specification.

(3) Easy to manage, easy to configure agent group agent

advantages：

(1) Run on the basis of java Moqua platform

(2) You can use jdbc to connect to various databases

(3) There is a complete interface, monitoring and security mechanism

(4) Automatic reconnection and error retry

shortcoming:

(1) The community is less active than RabbitMQ

(2) Not suitable for application scenarios with thousands of queues

2. *Apache RocketMQ*

features:

(1) Can guarantee strict message order

(2) Hundreds of millions of message accumulation capabilities

(3) Rich message pull mode

advantages:

(1) Based on java, it is convenient for secondary development

(2) A single machine supports more than 10,000 persistent queues

(3) Both memory and disk have a copy of data to ensure performance + high availability

shortcoming:

(1) There are not many types of clients

(2) There is no web management interface, a CLI is provided

(3) Community attention and maturity are not as good as RabbitMQ

3. *Apache Kafka*

features:

(1) Unique partition feature, suitable for big data systems

(2) Efficient performance and good scalability

(3) Reproducible and fault-tolerant

advantages:

(1) Zero-copy technology reduces IO operation steps and improves system throughput

(2) Fast persistence, message persistence can be performed under O(1) system overhead

(3) Support data batch sending and pulling

shortcoming:

(1) When a single machine has more than 64 queues, the performance is obviously degraded

(2) Using the short polling method, the real-time performance depends on the polling interval

(3) Consumption failure does not support retry

(4) Poor reliability

4. *RabbitMQ*

Features:

(1) It is currently the most mainstream message middleware

(2) It has high reliability and supports sending confirmation, delivery confirmation and other characteristics

(3) It is highly available and supports mirrored queues

(4) It supports many plugins

advantages:

(1) It is based on erlang and supports high concurrency

(2) It supports multiple platforms, multiple clients, and complete documentation

(3) It has high reliability

(4) It has a large-scale application in Internet companies, and the community is highly active

shortcoming

(1) Erlang is relatively small, which is not conducive to secondary development

(2) Under the proxy architecture, the central node increases the delay and affects performance

(3) Using the AMQP protocol, there is a learning cost for using it

Generally speaking, rabbitmq is the most widely used, with excellent performance, suitable for large and small companies, and suitable for a wide range of scenarios.

System Overview

Describe the main components of your system.

Order Fetching and Fulfillment → Order Microservice

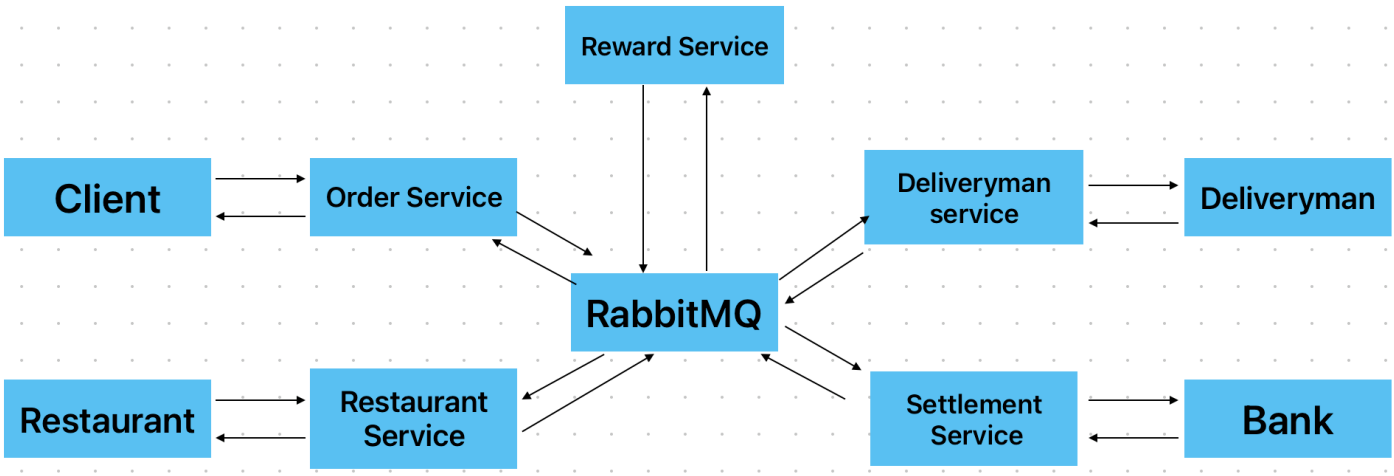
Vendor and Product Management → Merchant Microservices

Meal delivery, rider management → deliveryman microservice

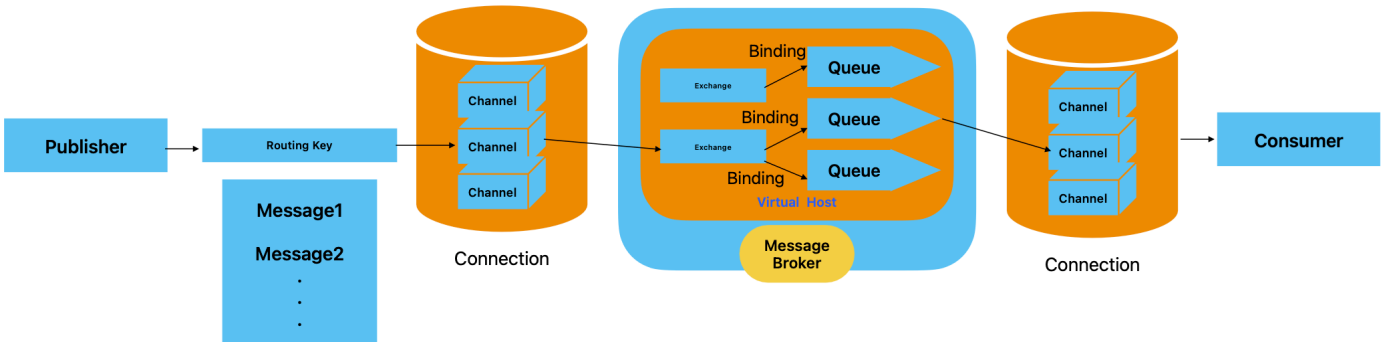
Bookkeeping and Settlement → Settlement Microservice

Integral Management → Integral Microservice

Include your system architecture diagram in this section.



architecture diagram



RabbitMQ message processing flow chart.

Explain how your system works based on the diagram.

Application Workflow:

1. *The user sends a message to the order service. After the order service performs certain processing, a connection between the order service and RabbitMQ is established, and the message is sent to RabbitMQ through the channel. RabbitMQ will send the message to the specified queue according to the exchange information of the message. (The exchange will specify the connection relationship between itself and one or more queues. This relationship is the Binding Key, which is configured by us. The message will contain the Routing Key information. When RabbitMQ receives the message, it will check the Routing Key. Then match the Routing Key and Binding Key, and finally send the information to the specified queue.) The message will first enter the Restaurant Service. After the Restaurant Service receives it, it will process, persist and confirm the information, and then send the information to RabbitMQ .*
2. *At this time, the status variable of the message has changed, and its routing information will have some changes, and then sent to the Deliverman Service. Deliverman Service performs some processing, persistence and confirmation on the information, and then sends the information to RabbitMQ.*
3. *Similar to the above process, as long as the message is confirmed by the server, it will enter the Reward Service and the Settlement Service in turn.*
4. *When the settlement Service processes the information, the message will be sent back to the Order Service, informing the user that the order has been created, and the information will be stored in the database.*

Each user needs to acknowledge the message.

Explain how your system is designed to support scalability and fault tolerance.

Scalability:

The existence of RabbitMQ allows us to divide the entire system into modules with different functions, which makes our application more scalable, and we can basically add new services into our application without large-scale code changes. For example, if our system only has Order Service, Restaurant Service, Deliveryman Service, and Settlement Service, this system can also operate normally. At this point, we can independently develop another Reward Service without making a lot of changes to the source code. We only need to add an exchange and a Case in the business code.

Fault tolerance:

ACID is often aimed at traditional local transactions, and distributed transactions cannot satisfy atomicity and isolation, so it is necessary to abandon the traditional ACID theory. Therefore, based on the BASE theory, the business status does not need to be strongly consistent in the microservice system; the order status can be final consistent. Therefore, in order to achieve final consistency, it is necessary to ensure that messages are not lost, and the sending-processing process must have a retry mechanism, and a warning must be given after multiple failed retries.

Here I have adopted some functions of RabbitMQ including: sending failure retry, consumption failure retry and dead letter warning. At the same time, because of the need to retry on failure, we will temporarily store the message in the database.

Contributions

Provide a sub section for each team member that describes their contribution to the project. Descriptions should be short and to the point.

Completed by Bo Tian alone.

Reflections

What were the key challenges you have faced in completing the project? How did you overcome them?

1. *Version dependency issues. Here I started by manually looking up what dependencies are available on the website. In the end, I used Spring Initialize to solve this problem, which can automatically configure the correct Maven dependencies for me.*
2. *The code in the project development is not standardized, and the development framework is chaotic. Here I refer to some java development specifications, and divide the code according to the functions of the package. The data-related part of the code is designed hierarchically, including DAO (data access object), DTO (data trancefer object), PO (persistant object), VO (value object).*

DAO: It contains objects for operations such as adding, deleting, modifying, querying, etc. of the DB.

PO: It is an object used to represent the persistence of the database.

DTO: It is used for data transmission.

VO: corresponds to the data object displayed on the interface. According to the interface display output requirements and page information collection input requirements. (I didn't do the front-end content here)

In addition, I distinguished RabbitMQ configuration, enumeration and service. In this way, each package has a content that it needs to be responsible for separately. And our code also has a certain template function, as long as a service is developed, the content of other services will become very easy.

What would you have done differently if you could start again?

1. *I should have used Springboot from the start, it would have saved me a ton of time. Because I don't have to think about the version of each dependency, and when using RabbitMQ, Springboot-related packages can save me a lot of code, and they are easier to use.*
2. *I figured I'd build a simple front end so that no matter how rough the project was it would look more complete.*

3. I should consider the clustering and containerized deployment of RabbitMQ, which should be closer to the real project.

What have you learnt about the technologies you have used? Limitations? Benefits?

In the process of using RabbitMQ, my distributed framework design and code development capabilities have been improved. The existence of message middleware makes it necessary for me to develop projects according to modular thinking. Firstly, this makes my code style more and more standardized. Secondly, it also makes my design ideas for an application system more and more clear. In addition, during the development of the project, my theory of distributed systems has been strengthened, because during the development process I must know why I am doing this.

The regret of this project is that I did not form a team to develop it. On the one hand, this makes the front-end and clustering of the project lacking. On the other hand, I did not practice the necessary skills for teamwork in this assignment, and my collaboration ability was not exercised.