A Cloud Native Microservice System for Ticket Booking

19126278 郭子斌 19126280 郝梓腾 19126304 李雨欣 19126334 杨本艳

Catalogue

1 Introduction

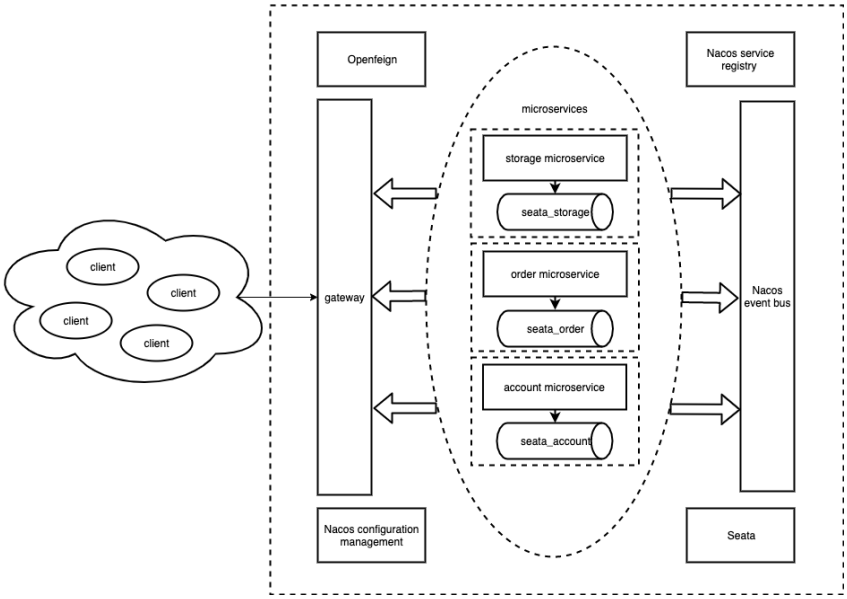


Figure (1) System architecture

Our project as a whole is designed based on the SpringCloud Alibaba microservice framework, using the technology Nacos + SEATA + OpenFeign. Nacos is committed to help our discover, configure, and manage microservices. It provides a set of simple and useful features enabling us to realize dynamic service discovery, service configuration, service metadata and traffic management. Seata is used to solve the global data consistency of distributed transactions. OpenFeign is a lightweight Restful HTTP service client. OpenFeign's @feignclient can parse the interface under the annotation of SpringMVC's @requestmapping, and generate the implementation class through dynamic proxy, so as to achieve load balancing in the class and invoke other services.

Here we will create three services, an order service, an inventory service, and an account service. When the user places an order, an order is created in the order service, the inventory service is called remotely to deduct the inventory of the order, the account service is called remotely to deduct the balance in the user's account, and the order status is changed to complete in the order service. And there are functions to cancel unpaid orders and refund already paid orders. The ordering operation spans three databases and there are two remote calls. Obviously, there will be distributed transaction problems.

2 Architecture

We used DDD structure in our project, the purpose is to deal with the problem of software complexity caused by excessive system scale. The process is that the development team and domain experts work together to understand and digest the domain knowledge through a common language, extract and divide the domain knowledge into subdomains, and build a model on the subdomain. A set of models in line with the current field.

2.1 Layered architecture

DDD divides the system into UI layer, application layer, domain layer and infrastructure layer. In the figure(1) above, the application layer is a very thin layer, because it is only responsible for receiving the parameters and routing from the UI layer to the corresponding domain model, and it is not responsible for handling specific business logic. The business logic of the system is placed in the domain layer, so the domain layer occupies a large area in the system architecture.

In the hierarchical structure, the upper module calls the services provided by the lower module, there will be a dependency relationship: the upper module should not depend on the lower module, both should rely on abstraction：abstraction should not depend on implementation, implementation should depend on abstraction.

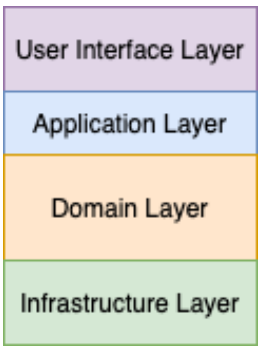


Figure (2) Layered architecture

2.2 DDD Modelling

When designing a system using DDD, it mainly includes Entity, Value Object, Service, Aggregate, Repository and other elements.

In modeling, Entity can be used to represent a thing. Since Entity is used to represent a thing, then it should have a unique value to identify this thing, at the same time, he can also record changes in the state of this thing.

Value Object is used to describe the characteristics of a certain aspect of things, so it is a stateless and an object without an identifier, which is the essential difference from Entity.

The operation provided by the service is that it is provided to clients that use it, and highlights the relationship between domain objects. All services are only responsible for coordinating and delegating business logic to the domain objects for processing. It actually implements the business logic itself, and most of the business logic is carried and implemented by the domain objects. The service can interact with a variety of components, these components include: other services, domain objects and repository or dao.

Aggregate is a collection of related objects. It serves as the basic unit of data modification and provides a boundary for data modification. Each aggregation has a root and a boundary. The root is also an entity. Objects outside the aggregation boundary can only operate on the internal elements of the aggregation through the root. Aggregation brings together a group of related objects, thereby reducing the complexity of the system.

The repository is used to store aggregates, which means that each aggregate should have a repository instance. Both Entity and Value Object should have behaviors, and some behaviors are semantically unsuitable to be placed in these two objects, so a Service object is abstracted separately to store these behaviors. Factory is used to generate aggregates. When the steps to generate an aggregate are too complicated, you can put the generation process in the factory.

2.3 Ticket booking system

The workflow for our system:

A user reserves tickets from ticket order system,the ticket order system generates an orderID for the reservation,user can pay for the reservation by the orderID. When the user places an order, the ticket inventory decreases by the corresponding amount， and the user's remaining amount also decreases. The user can cancel the unpaid order, or refund the purchased ticket, the storage becomes the original unpurchased.

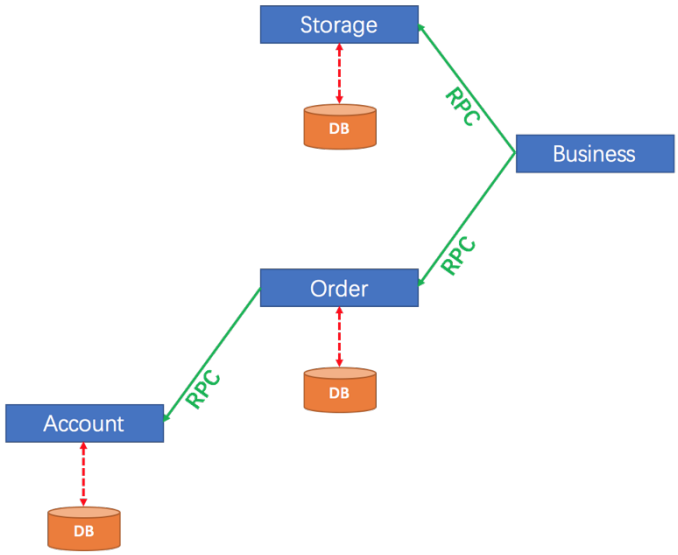


Figure (3) Workflow

3 Design Patterns for Microservice

3.1 Distributed Transaction Management

A business operation needs to cross multiple data sources or need to make remote calls across multiple systems, which will cause distributed transaction problems.

We chose to use SEATA to maintain data consistency in distributed transactions. SEATA is an open source distributed transaction solution dedicated to providing high performance and easy to use distributed transaction services. SEATA will provide users with AT, TCC, SAGA, and XA transaction models to create a one-stop distributed solution for users. AT mode focuses on data consistency in multiple databases. So we choose this one.

Here is our distributed transaction solution with SEATA.

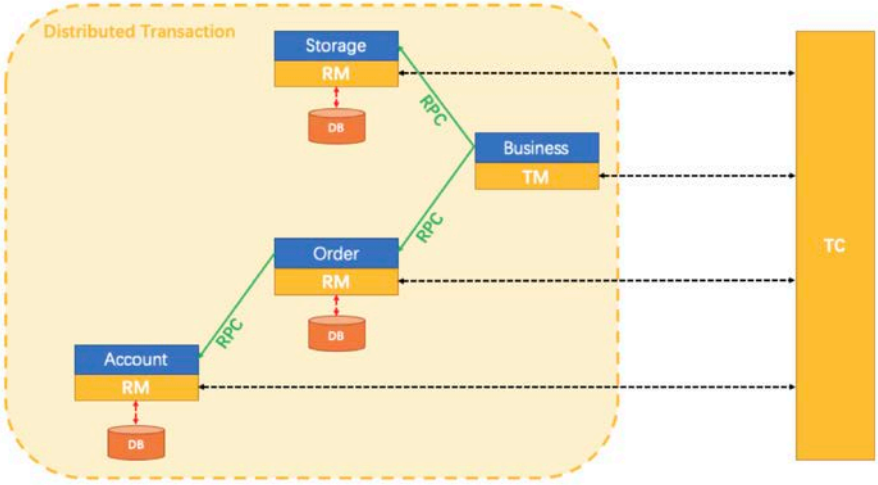


Figure (4) transaction Management with SEATA

There are 3 basic components in SEATA:

a) Transaction Coordinator(TC): Maintain status of global and branch transactions, drive the global commit or rollback.

b) Transaction Manager(TM): Define the scope of global transaction: begin a global transaction, commit or rollback a global transaction.

c) Resource Manager(RM): Manage resources that branch transactions working on, talk to TC for registering branch transactions and reporting status of branch transactions, and drive the branch transaction commit or rollback.

We just need an annotation @GlobalTransactional on business method, like this:

```
@GlobalTransactional(name = "fsp-create-order",rollbackFor = Exception.class)
public Order create(Order order)
{
    log.info("------>开始新建订单");
    //1 新建订单
    orderDao.create(order);
   return orderDao.orderfindByid(order.getId());
}
```

Figure (5) An example of using SEATA

3.2 API Gateway

This project provides an API Gateway built on top of the Spring Ecosystem, including: Spring 5, Spring Boot 2 and Project Reactor. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.

3.2.1 Key Concept

The gateway provides fully managed API services, rich API management functions, and assists enterprises to manage large-scale APIs to reduce management costs and security risks, including protocol adaptation, protocol forwarding, security policies, anti-brushing, traffic, and monitoring logs. Here are some important concepts in Spring Cloud Gateway.

Route: The basic building block of the gateway. It is defined by an ID, a destination URI, a

collection of predicates, and a collection of filters. A route is matched if the aggregate predicate is true.

Predicate: This is a Java 8 Function Predicate. The input type is a Spring Framework ServerWebExchange. This lets you match on anything from the HTTP request, such as headers or parameters.

Filter: These are instances of Spring Framework GatewayFilter that have been constructed with a specific factory. Here, you can modify requests and responses before or after sending the downstream request.

3.2.2 Work Process

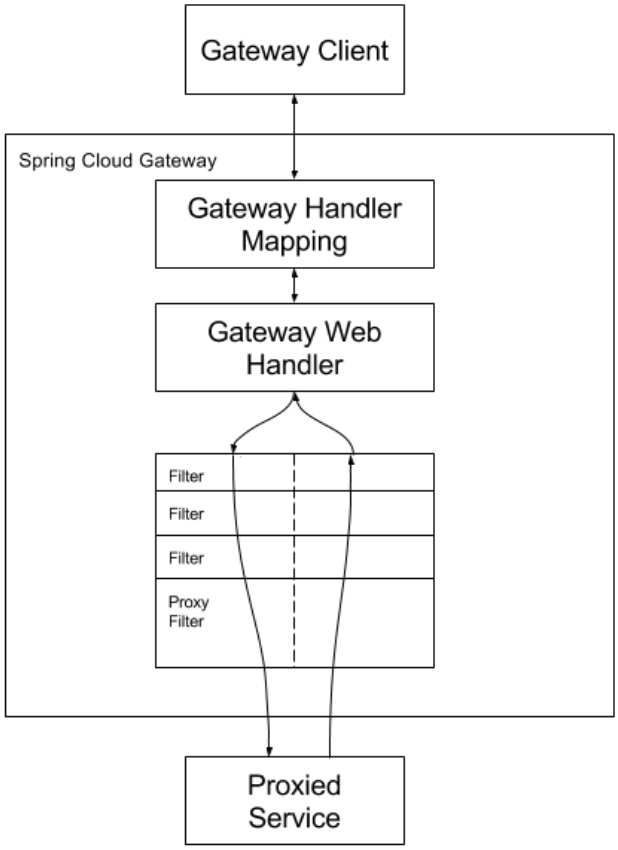


Figure (6) Gateway work Process

Clients make requests to Spring Cloud Gateway. If the Gateway Handler Mapping determines that a request matches a route, it is sent to the Gateway Web Handler. This handler runs the request through a filter chain that is specific to the request. The reason the filters are divided by the dotted line is that filters can run logic both before and after the proxy request is sent. All “pre” filter logic is executed. Then the proxy request is made. After the proxy request is made, the “post” filter logic is run.

3.2.3 Position in the Microservices Architecture

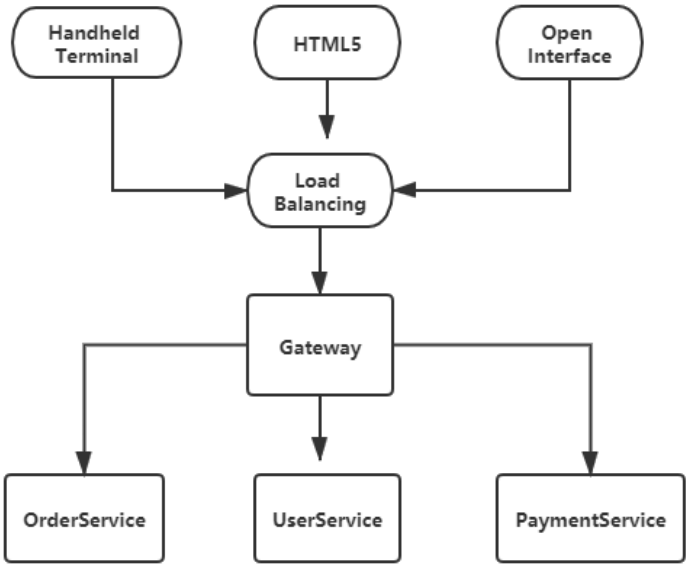


Figure (7) Gateway in the Microservices Architecture

4 The deployment guide

4.1 Basic information:

Development environment :WINDOWS 10 pro
Development tool: IDEA
Development framework: spring-cloud-alibaba
Middleware: NACOS+SEATA+GATEWAY+OPENFEIGN

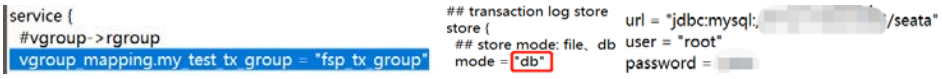
4.2 Deployment environment: docker

(1) deployment of mysql environment
Exposed port: 3306 3307 3307 is used to bind to the host port
Docker run --name mysql_one-e MYSQL_ROOT_PASSWORD= 123 -p 3307: 3307-d
Mysql: 5.7
Database creation includes seata library, order library, storage library, and user library.Sql statements are uploaded to github
(2) deployment of Nacos environment
Exposed port :8848 is bound to host port 8848
Docker run-d --name nacos-p 8848: 8848-e MODE= standalone-e nacos
(3) deployment of Seata environment
Exposed port :8091 is bound to host port 8091
Docker run-d --name seata-p 8091:8091-e SEATA_PORT=8091 seataio/seata-server:latest
To configure seata, first create a seata database in mysql for the logging of seata, create SQL

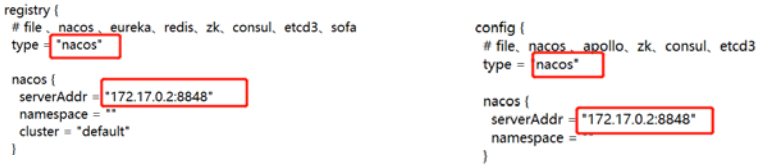
and upload it to github, and then locally modify nacos-config, file.conf, and regist.conf in the seata folder.

Nacos-config modification location:

File. Conf modification location:



Rigistry. Conf modification location::



Finally, copy the three files to the docker environment in turn:

Docker cp/your path/register.conf seata:seata-server/resources/

Docker restart seata

4.3 Micro-service deployment

Dockerfile as follows:

```
FROM java:8
VOLUME /tmp
ADD order-1.0-SNAPSHOT.jar app.jar
RUN    bash -c 'touch /app.jar'
EXPOSE 2001
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

(1) order microservice deployment: the exposed port is 2001 and the host binding port is 2001:2001

(2) inventory centric micro service deployment

The exposed port is 2002 and the host binding port is 2002:2002. Dockerfile is consistent with the order microservice except jar package and exposed port

(3) user centric micro service deployment

The exposed port is 2003 and the host binding port is 2003:2003. Dockerfile is consistent with the order microservice except for the jar package and exposed port

(4) gateway center:

The exposed port is 80, which is bound to the host port at 80:80. Dockerfile is consistent with the order microservice except for the jar package and the exposed port

5 The test results

Test tool : postman

A. Order creation interface test

http://localhost:2001/order/create? UserId = 1 & ticketTypeId = 1 & count = 1

Test results:

```
{"code":200,"message":"订单创建成功 ","data":{"id":48,"userId":1,"ticketTypeId":1,"count":1,"money":10,"status":0}}
```

B. Order cancellation interface test

http://localhost:2001/order/confirm? OrderId = 48 & pay = 0

Test results:

```
{"code":200,"message":"取消成功","data":{"id":48,"userId":1,"ticketTypeId":1,"count":1,"money":10,"status":2}}
```

C. Gateway test, where the gateway binding port is 80, you can no longer enter the port number

http://localhost:/order/create? UserId = 1 & ticketTypeId = 1 & count = 1

Test results:

```
{"code":200,"message":"订单创建成功 ","data":{"id":49,"userId":1,"ticketTypeId":1,"count":1,"money":10,"status":0}}
```

D. Order payment interface test

http://localhost:2001/order/confirm? OrderId = 49 & pay = 1

Test results:

```
{"code":200,"message":"交易成功","data":{"id":49,"userId":1,"ticketTypeId":1,"count":1,"money":10,"status":1}}
```

E. Order refund interface test

http://localhost:2001/order/refund? OrderId = 49

The test results

```
{"code":200,"message":"订单退款成功 ","data":null}
```

F. Post test for all interfaces of order query

http://localhost:/order/findAll

Test results:

```
{"code":200,"message":"查询成功","data":[{"id":1,"userId":1,"ticketTypeId":1,"count":10,"money":100,"status":3},{"id":10,"userId":1,
"ticketTypeId":1,"count":10,"money":100,"status":3},{"id":11,"userId":1,"ticketTypeId":1,"count":10,"money":100,"status":1},
```

G. Inventory query for all interface tests

http://localhost:/storage/findAll

Test results:

```
{"code":200,"message":"库存查询成功！","data":[{"id":1,"ticketTypeId":1,"total":100,"used":75,"residue":25,"price":10}]}
```

H. Inventory increase, decrease interface test is used for the order microservice to be invoked.

Http://localhost:2002/storage/decrease?TicketTypeId = 1 & count = 1

Test results:

```
{"code":200,"message":"扣减库存成功！","data":null}
```

I. User account query all interface tests

http://localhost:/account/findAll

```
{"code":200,"message":"查询成功","data":[{"id":1,"userId":1,"total":1000,"used":750,"residue":250}]}
```

J. The user account debit interface test is also invoked by the order microservice

http://localhost:2003/account/decrease? UserId = 1 & money = 1

```
{"code":200,"message":"扣减账户余额成功！","data":null}
```


K. Correctness proof of order payment transaction

a) Check the current account for money


```
{"code":200,"message":"查询成功","data":[{"id":1,"userId":1,"total":1000,"used":751,"residue":249}]}
```


b) Query current inventory


```
{"code":200,"message":"库存查询成功！","data":[{"id":1,"ticketTypeId":1,"total":100,"used":76,"residue":24,"price":10}]}
```


c) Create and pay orders


```
{"code":200,"message":"交易成功","data":{"id":50,"userId":1,"ticketTypeId":1,"count":1,"money":10,"status":1}}
```


d) Check current inventory and account for money


```
{"code":200,"message":"库存查询成功！","data":[{"id":1,"ticketTypeId":1,"total":100,"used":77,"residue":23,"price":10}]}
```

```
{"code":200,"message":"查询成功","data":[{"id":1,"userId":1,"total":1000,"used":761,"residue":239}]}
```


6 Link to the source code

https://github.com/a626411464/A-Cloud-Native-Microservice-System-for-Ticket-Booking