Example of Microservices using Zuul, Eureka, Kafka, Spring cloud/boot and Global TX

By Daping Huang

**Introduction**

As the high volume, high scalability in cloud container technologies get more and more popular, Microservice architectures will become next mainstream for business-based distributed applications development due to microservice architecture have below advantages:

* Independent Development and Deployment
* Fault isolation
* Highly Scalability by container image copy.
* Highly reusable
* Easy integrated as applications

In order to demonstrate how to build microservices using most up to date technologies like with Zuul gateway load balancer, Eureka service registration and discovery, Spring cloud/boot, Kafka messaging and weak consistent global transactions, I setup an example project. In example project I also demonstrate how to track and discover “dirty transaction data” that microservices system may encounter.

This example builds a sale system with order, account, payment, inventory and audit microservices.

The example codes are available in Bitbucket.

**Design Principle of Domain Driven**

Unlike SOA architecture, which is coarse grain service, microservice is fine grain service. Microservices design should follow couple principles:

* Domain driven, represented by a serial of models
* Strong cohesion/bonded model(s) should be in same microservice.
* Loose coupling/bonded

I introduce a model bonding analyse mechanism; all relationships (include dependency and state) of models are categorised into 4 levels.

* Conjugate model bonded, represented by solid double line; Strong cohesion model relationship that should not be separated.
* Ionic model bonded, represented by solid single line; Fragile coupling model relationship that can be traded-off for separation.
* Molecular model bonded, represented by dash single line; Loosest coupling model relationship that always can be separated
* No bonded, without relationship.

Fig. Model bonding.

A picture containing drawing

Description automatically generated

Model bonding analyses may result in above diagram, for example. Each circle is a model. This diagram has 2 clear “molecular” A and B, those can be considered as 2 (A and B) microservices. Then here is a trade-off you have to make whether model 9 separated from 7, 8 to be a single microservice. If yes, then we have A, B, C, D microservices. (Note: our example is too simple to apply this rule)

**Synchronous or Asynchronous Communications**

The asynchronous communication is event driven via messaging broker like RabitMQ or Kafka etc. Saga compensation is weak consistency persistent persistence.

The synchronous communication is direct call via http/rest (usually) via web client or Feign Client between 2 microservices. It can apply strong consistency or week consistency persistence.

Fig. Example Sale System microservices via synchronous service call.

A close up of a map

Description automatically generated

The Sale Application is a “product” that integrates a serial fine grain microservices like Order, Inventory, Account, Payment booking etc. together. As a demo, the examples may not be business correct.

Two levels of microservice calls are ideal design, that means all fine grain microservices are in flat structure; Three levels of microservice calls are also acceptable but not a good practice. More than three level calls should avoid, it will make system management complex, and may result in circular dependency.

The example microservice deploys spring MVC with restful, JPA/Hibernate, and Atomikos local transaction service, it looks like:

A picture containing clock

Description automatically generated

**Microservice registration**

The microservice host is hidden from client, and each microservice required register in server(s), like Eureka. In the examples code, go to eureka-server folder and run *../gradlew build bootRun*. Open <http://localhost:8010/>, you will see Eureka server is up.

Fig. Eureka view when all microservices up and registered

Table

Description automatically generated

To register the microservice in Eureka, add dependency in to build.gradle.

Then, add below 2 lines to *application.properties* of microservice application.

eureka.client.service-url.defaultZone=http://localhost:8010/eureka/

eureka.instance.metadata-map.service-name=account-service

To run microservice, go to application folder and using command line *“../gradlew build bootRun*”

The client lookup microservice via an @Autowired proxy. It is interface like this:  
@FeignClient(name="order-service)

@RequestMapping("/api")

public interface OrderApiClient {

@PostMapping(path = "/order", produces = MediaType.APPLICATION\_JSON\_VALUE)

public Order order(@RequestBody Order order);

}

**Microservices Distributed Tracking System**

Distributed Tracking system helps to track and debug latency of microservices. Zipkin is one of of the popular one. The installation to docker is pretty simple: “*docker run -d -p 9411:9411 openzipkin/zipkin*”, here 9411 is port number. Registration for the service also simple, in application.properties add one line: spring.zipkin.base-url=http://localhost:9411/ . Also, add dependency in to build.gradle.

Open <http://localhost:9411/>, you will see the call graphs and time spent in each span if there are activities going on.

Timeline

Description automatically generated

**Gateway and load balancing**

You may scale up microservices by adding new instance and require load balancing.  
Netflix Zuul gateway and Robin load balancing is popular service for this purpose. The full configure is in Zuul-Server of the examples. It is run as Spring book application. All microservices opened for gateway and load balance should be configure in *application.properties* of Zuul-Server project.

Chart

Description automatically generated

Add @RibbonClient to client proxy

@RibbonClient (name="order-service)

@FeignClient(name="order-service)

@RequestMapping("/api")

public interface OrderApiClient {

@PostMapping(path = "/order", produces = MediaType.APPLICATION\_JSON\_VALUE)

public Order order(@RequestBody Order order);

}

**Strong or weak Consistency?**

Microservices are distributed deployed, the ACID data operations may not go well. The isolated data operation is obviously not suitable, therefore ACID become ACD.

* Strong data consistency will block data accessing and down-size the put-through volume of services. It is big shortage to microservices.
* Week data consistency with ACD in local and roll back in global is the main strategy for microservices, this strategy also introduces another issue, hard to debug data consistency.

There are some cross microservices weak consistency transaction (also called Glogal Transaction) providers:

* Bitronix, only run in Wildfly Jboss Application Service
* Atomikos, TransactionsEssential™ free but not officially recommended (may need to write 2 confgures by yourself); ExtrameTransactions™ is official recommended but not free.
* Seata, Alibaba open source and free. The example application uses Seata transactions.

Diagram

Description automatically generated

Seata Global Transaction generates Global TX ID and propagates over individual microservice, each microservice has its own ACID transaction that can use any JTA provider. Seata records the data before data updatied or after data created, upon failure, Seata recovers or deletes the data based on record.

All weak consistent persistent transaction has a hard issue, if a row of data is updated during another failure transaction, it results a dirty row of data. Though, this case rarely happens in real world, the debug is hard. I introduce an audit log concept for dirty transaction detection.

Download Seata transaction server from <http://seata.io/en-us/blog/download.html> unzip it. The file structure like this:

A screenshot of a cell phone

Description automatically generated

Before you run seata server, some configure are required:

The *File.conf* need to configure to fit your own application

service {

vgroupMapping.dph\_tx\_group = "seata-server"

seata-server.grouplist = "127.0.0.1:8091"

disableGlobalTransaction = false

}

If select file mode, not more configure required; if select db mode, it needs below configure, and 3 Database tables that the db url point to. The DDL scripts are in *setup-seata-server-db.sql* of examples codes.

store {

## store mode: file、db、redis

mode = "db"

db {

## the implement of javax.sql.DataSource, such as DruidDataSource(druid)/BasicDataSource(dbcp)/HikariDataSource(hikari) etc.

datasource = "druid"

## mysql/oracle/postgresql/h2/oceanbase etc.

dbType = "mysql"

driverClassName = "com.mysql.cj.jdbc.Driver"

url = "jdbc:mysql://127.0.0.1:3307/seata\_db?useUnicode=true&useSSL=false&character\_set\_server=utf8mb4"

user = "dph"

password = "N0Passw0rd"

- - - --- - - -

}

}

Another configure file is *registry.conf* for registry couple of services.

registry {

type = "eureka"

eureka {

serviceUrl = "http://localhost:8010/eureka"

application = "seata-server"

weight = "1"

}

* - - - - -

}

Here it selects Eureka microservice lookup service.

The *application.properties* which application uses Seata Global Transaction is required configure, add

spring.cloud.alibaba.seata.tx-service-group=dph\_tx\_group

Copy *File.conf* file to the same folder as application.properties. The “service” block is only required in *File.conf* file.

Finally, add *@GlobalTransactional* to restful API method.

@GlobalTransactional(rollbackFor = Exception.class, timeoutMills = 10000)

@PostMapping(path = "debit", produces = MediaType.APPLICATION\_JSON\_VALUE)

public @ResponseBody Account debit(@RequestBody Debit debit) {

return accountService.debit(debit);

}  
To activate Seata service, the database and Eureka should be in up and running state and run *seata-server.sh* in bin folder. Now the Seata Transaction service is registered in Eureka and ready to serve.

A screenshot of a cell phone

Description automatically generated

**Kafka Streaming**

In our example, it uses Kafka as logging system for AuditLog. The configure files are under *kafka-server /config* folder.

The similar streaming open source like Kafka is RabitMQ.

To run the server, enter *kafka-server* folder, invoke both Kafka server and Zookeeper server:

./bin/zookeeper-server-start.sh ./config/zookeeper.properties

./bin/kafka-server-start.sh ./config/server.properties

**AuditLog Design for Distributed Microservices**

As we have mentioned, Weak consistency Global Transaction may result in dirty row of transaction data upon the case that a row data is rolling back while another transaction updates the same row of data. Here is the dirty come from, the second update will be erased and replaced by rolled back one. This situation rarely happens though.

Without the proper audit log, we never know:

* Dose Dirty transaction happen?
* How and when it happens?
* How to recover?

To achieve above 3 items, we at least audit:

Each global transaction start time, end time and roll back time, participant entities and their data.

**Audit Design**

Audit log design consideration:

* Not block any microservices.
* Generate a global request unique Id at request entry point of the system for each global transaction and propagate to participant microservices silently.
* Log Audit to DB via an microservice.

Diagram

Description automatically generated

The detail codes are available in this examples. At this moment, it still required adding a line of code to send audit message. Potentially, it can be fully automatic via annotation like “@AuditLog”.

**Dirty Global Transaction Analysis:**

The dirty transaction only happens in case of rollback while other transaction update the same row of data. It rarely occurs. However, it is very easy to siminulate by starting the microservice as debug mode, start a transacion via that microservice and on hold before return, make second transaction done to upate same row data via second instance of microservices. Then rollback first transaction.

You make get a log like this:

Table

Description automatically generated

Transaction 1 has record 15,16,17,18, and roll back at 20, and Transaction 2 has record 19. Both records 16 and 19 modified same *Item* entity and same id, means both modified the same row of data. But the Transaction 1 rollbacks at record 20, then the transaction 2 data of *Item* entity were overridden. Now, based on record, the dirty data can be cleaned.

In the example codes, we already have a simple dirty data detection api in audit log microservice.

**About the example**