Method of Refactoring a Monolith into Micro-services

Qing-hui Ren*, Sheng-lin Li, Han Qiao
Army Logistical University of PLA, Chongqing, China.

* Corresponding author. Tel.: +8613290006306; email: renqinghui007@163.com
Manuscript submitted September 29, 2018; accepted November 2, 2018.
doi: 10.17706/jsw.13.12.646-653

Abstract: Micro-service architecture, a new cloud deployment technique for applications and services, provides an effective means to build an integrated system and helps to realize the universality and compatibility between information systems. Targeted at refactoring a monolithic system, this paper proposes Spring Cloud Framework as a solution to the micro-service architecture. The construction process of the system was detailed with an example of inventory management system. The system was constructed in the following steps: 1) service classification and seam recognition; 2) service development and packaging; 3) Spring Cloud Framework configuration; 4) service deployment. Through the verification of the system functionality, it is confirmed that the micro-service architecture system has stronger robustness than the monolithic system.

Key words: Micro-service architecture, integrated system, spring cloud framework, monolithic system.

1. Introduction

The micro-service architecture structures an application as a collection of loosely coupled services, and defines services corresponding to business capabilities [1], [2]. Featuring effective management complexity, independent service deployment, and flexible technology stacks [3], micro-service architecture has been adopted by many enterprises, e.g. Amazon and Netflix, as the development framework for their information system [4]. Despite these advantages, it is not easy for enterprises still using monolithic system to develop a new micro-service architecture system [5]. To solve this problem, this paper proposes the use of Spring Cloud Framework as a solution to the micro-service architecture [6].

Compared with prior efforts, this paper makes the following contributions: 1) the creation of a complete splitting of a monolithic system. The division between program and database makes it possible to construct the system at a granular level; 2) the application of advanced technology such as Spring Cloud Framework, docker container, etc. These technologies simplify the entire development and 3) the verification of the robustness of micro-service architecture system. Through single node load monitoring and overall testing of the system, the robustness was verified based on the simulated delay of service response.

The remainder of this paper is organized as follows: Section 2 shows the splitting process of a monolithic system. Section 3 constructs the micro-service architecture system. Section 4 verifies the system; finally, Section 5 draws some meaningful conclusions.

2. Splitting a Monolithic System

Taking inventory management system as an example, this section shows the splitting process of a monolithic system. The main steps are as follows:

Step 1 Physical layering of three-tier architecture

This step realizes the transformation of logical layer to the physical layer. The decoupling of coarse granularity can lay the foundation for the functional decomposition and the elimination of integrated database.

Step 2 Identifying bounded context

With the use of seams, developers are allowed to extract code blocks from the original system. The blocks can relatively independently implement modifications with little effect on other parts of the code [7]. Before building a new system, it must accurately identify the capabilities of each business module. Table 1 shows the main features and details of the system.

Table 1. Features and Details of the System

Name	Function	Description
Account	GET/PUT/POST	Account data management
Iomanagement	GET/PUT/POST	Service of issue and receipt
Details	GET	Data query service.
Inventory	GET/PUT/POST	Inventory management.
Notification	GET/PUT	Notification service
Auth	GET/PUT	Authentication mechanism
Gateway	--	Service routing
Monitoring	--	Monitoring service
Config	--	Configuration center
Registry	--	Service registration

In Table 1, the function GET is used to read data information, PUT is used to upload the data information, and POST is used to update the data information [8].

Step 3 Elimination of integrated database

The key to eliminate the integrated database lies in breaking the foreign key relationship between the tables. In contrast, the service-oriented splitting of the integrated database is effectively enabling each service to form its own database [9]. Other services can access the data through the API exposed by one service instead of accessing the database directly.

3. Construction of Micro-service Architecture System

Spring Cloud Frame is a comprehensive solution to the implementation of micro-service architecture. The solution is grounded on spring boot for configuration management, service governance, circuit breaker, intelligent routing, control bus, global lock, decision making, distributed session, and cluster state management, etc. In view of the above characteristics and the separation of the monolithic system, this section creates the docker image and deploys the services into the docker container, aiming to use the Spring Cloud Framework to construct the micro-service architecture system.

3.1. Configuration of the Spring Cloud Framework

This system uses some of the core components of the Spring Cloud Framework [10].

Spring Cloud Config: It is a distributed configuration center component that provides scalable configuration services and uses the configuration center to centrally manage various configuration files for all services.

Eureka: The service governance component, including the service registry, mainly implements the registration and discovery mechanism of the service.

Ribbon: The client operated a load balancing service invocation component.

Hystrix: Fault tolerant management component can realize circuit breaker mode and provide good fault tolerance for the delay.

Feign: The declarative service invocation component is based on the Ribbon and Hystrix.
Zuul: Gateway components provide intelligent routing, access filtering and other functions.
Turbine: Monitoring the metrics of Hystrix on each node in the cluster environment.
Figure 1 shows the component architecture of the system.

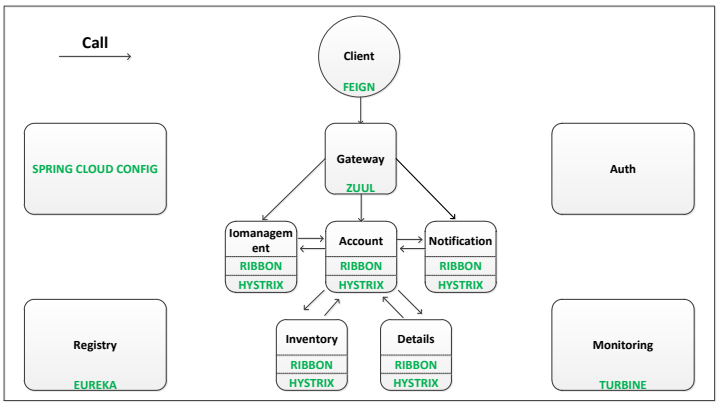


Fig. 1. System component architecture.

It can be seen from Figure 1 that: 1) the account service calls other services through the remote client Feign. The services realize each other through the registration centre Eureka. Then the unified gateway provided by Zuul and all open circuit information is unified through the aggregator Turbine according to the business logic; 2) the services use ribbon aims to realize load balancing and to realize circuit breaker function with Hystrix; 3) Spring Cloud Config as the configuration centre manages all the configuration files in the business process; 4) Auth service operates the basic authentication mechanism.

3.2. Deploy Services to Docker Container

All the services of the micro-service architecture system run independently in their respective docker containers; each service is accessed through the process of inter-process communication [11]. So the service needs to be deployed to the container. Taking Account service as an example, the process is described in details as follows.

1. Service packaging

Figure 2 shows the Account service packaged as a jar package.

```
ren@ren-X550LD:~/DigitalBarracks/account-service$  mvn package -DskipTests
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building account-service 0.0.2-SNAPSHOT
[INFO] ------------------------------------------------------------------------
.....
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 21.767 s
[INFO] Finished at: 2017-09-25T09:17:52+08:00
[INFO] Final Memory: 31M/299M
```

Fig. 2. Service packaging.

2. Pulling docker images

The docker image, similar to the virtual machine image, is a read-only template, including the file system that satisfies the data required for the container. A docker image can be built on top of another docker image; each container is an image instance; starting the container is equivalent to an increase in the image on a writable layer. With theses, it can write applications [12]. Therefore, it should first pull the underlying images

needed to run the container. Figure 3 shows the Account service base image from Docker Hub (https://hub.docker.com).

```
root@ren-X550LD:/home/ren/DigitalBarracks/account-service# docker pull
digitalbarracks0/account-service
Using default tag: latest
latest: Pulling from digitalbarracks0/account-service
......
Digest:
sha256:b271b32d4de2a973994cd5e2ff419e875cc2af1edea5f9954e53ea7af0551a1a
Status: Downloaded newer image for digitalbarracks0/account-service:latest
```

Fig. 3. Pulling Docker Images.

3. Creation of custom images

With successful underlying images pulling, custom images are created and the container is operated on top of these images. There are two ways to create a custom image. One is to manually start the container with the docker run command, and then to commit the newly configured container as an image [13]. The method is complex for each modification needs to be committed again. The second method is much easier, by applying the Dockerfile to generate the images [14]. Figure 4 shows Dockerfile configuration.

```
FROM java:8-jre
MAINTAINER SZYQ_YZGL <xxx@xxx.com>
ADD ./target/account-service.jar /app/
CMD ["java", "-Xmx200m", "-jar", "/app/account-service.jar"]
EXPOSE 6000
```

Fig. 4. Dockerfile Configuration.

The FROM keyword specifies the image template for the custom image; the MAINTAINER keyword indicates the author and contact information of the document; the ADD keyword copies the previously packaged jar package to the / app directory of the container; the CMD keyword indicates the time when starting the container running the jar package; the EXPOSE keyword specifies the port in the container.

According to the above analysis, the relationship between basic image, custom image, container, and account service can be obtained. Custom image is built on top of the basic mirror with the container as a running instance, and the account service is operated in the container. With the configured Dockerfile, the image can be created and the container can be operated simply by the docker build command.

4. Configuration of multiple containers

A micro-service architecture system usually consists of multiple services. The one-by-one docker build command manually is time-consuming and laborious. Therefore, it uses the Docker Compose service orchestration tool to manage multiple containers, the command is as follows.

docker-compose -f docker-compose.yml -f docker-compose.dev.yml up –d

Among them, *docker-compose* is the base command; *-f* specifies the optional yaml configuration file, which are *docker-compose.yml* and *docker-compose.dev.yml*; *up* represents the boot; *-d* represents the background operation.

In the two yaml files, *docker-compose.dev.yml* writes all service build commands and open ports, and *docker-compose.yml* specifies the boot mode based image, custom environment variable, container and log

file attribute.

With the execution of the *docker-compose* command; Figure 5 shows the building process of the account service.

```
root@ren-X550LD:/home/ren/DigitalBarracks# docker-compose -f docker-compose.yml -f
docker-compose.dev.yml up -d
Building account-service
Step 1/5 : FROM java:8-jre
 ---> e44d62cf8862
Step 2/5 : MAINTAINER SZYQ <xxx@xxx.com>
 ---> Running in 3d312b392196
 ---> 9a5b31cf5b97
Removing intermediate container 3d312b392196
Step 3/5 : ADD ./target/account-service.jar /app/
 ---> 1ebe6096c304
Step 4/5 : CMD java -Xmx200m -jar /app/account-service.jar
 ---> Running in 3399b2e6d443
 ---> 5c7bc4fa72f7
Removing intermediate container 3399b2e6d443
Step 5/5 : EXPOSE 6000
 ---> Running in f398695c9b21
 ---> 8aff21abe965
Removing intermediate container f398695c9b21
Successfully built 8aff21abe965
Successfully tagged digitalbarracks0/account-service:latest
```

Fig. 5. The Execution Process of the docker-compose Command.

Figure 5 shows the build process of the service executing the statements in the Dockerfile one by one. So far, the whole micro-service architecture system development, deployment and operation have been completed.

4. Micro-Service Architecture System Testing

To make the system meet the requirements, it is necessary to verify mainly system function and simulation service delay.

4.1. Function Test

When starting all containers, it can see all the running containers through the *docker ps* command. Figure 6 shows the running container.

```
root@ren-X550LD:/home/ren/DigitalBarracks# docker ps
CONTAINER ID        IMAGE                                COMMAND                  CREATED             STATUS              PORTS
NAMES
7067afd305c1        digitalbarracks0/account-service     "java -Xmx200m -ja..."   3 weeks ago         Up Less than a second
0.0.0.0:6000-
>6000/tcp                                                digitalbarracks_account-service_1
6e2ac945b4cd        digitalbarracks0/auth-service        "java -Xmx200m -ja..."   3 weeks ago         Up Less than a second
0.0.0.0:5000->5000/tcp                                   digitalbarracks_auth-service_1
6d0f8595b0ed        digitalbarracks0/mongodb             "/init.sh"               3 weeks ago         Up Less than a second
0.0.0.0:25000->27017/tcp                                 digitalbarracks_auth-mongodb_1
6a1e6fae43f9        digitalbarracks0/inventory-service   "java -Xmx200m -ja..."   3 weeks ago         Up Less than a second
0.0.0.0:7000->7000/tcp                                   digitalbarracks_inventory-service_1
b8573233d675        digitalbarracks0/details             "java -Xmx200m -ja..."   3 weeks ago         Up Less than a second
0.0.0.0:3000->3000/tcp                                   digitalbarracks_details_1
f46fb1dec520        digitalbarracks0/iomanagement        "java -Xmx200m -ja..."   3 weeks ago         Up Less than a second
0.0.0.0:2000->2000/tcp                                   digitalbarracks_iomanagement_1
39cb9e14835f        digitalbarracks0/gateway             "java -Xmx200m -ja..."   3 weeks ago         Up Less than a second
0.0.0.0:80->4000/tcp                                     digitalbarracks_gateway_1
5369b88c8b3e        digitalbarracks0/config              "java -Xmx200m -ja..."   3 weeks ago         Up Less than a second
0.0.0.0:8888->8888/tcp                                   digitalbarracks_config_1
b59bb648119e        digitalbarracks0/monitoring          "java -Xmx200m -ja..."   3 weeks ago         Up Less than a second
0.0.0.0:8989->8989/tcp, 0.0.0.0:9000->8080/tcp           digitalbarracks_monitoring_1
25c609eddb0a        digitalbarracks0/registry            "java -Xmx200m -ja..."   3 weeks ago         Up Less than a second
0.0.0.0:8761->8761/tcp                                   digitalbarracks_registry_1
1251888c2867        rabbitmq:3-management                "docker-entrypoint..."   3 weeks ago         Up Less than a second
4369/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp, 15671/tcp, 25672/tcp, 0.0.0.0:15672->15672/tcp   digitalbarracks_rabbitmq_1
bc403c49eb44        digitalbarracks0/mongodb             "/init.sh"               3 weeks ago         Up Less than a second
0.0.0.0:26000->27017/tcp                                 digitalbarracks_account-mongodb_1
7b197c47e2d8        digitalbarracks0/mongodb             "/init.sh"               3 weeks ago         Up Less than a second
0.0.0.0:27000->27017/tcp                                 digitalbarracks_inventory-mongodb_1
443ff0c46ef5        digitalbarracks0/mongodb             "/init.sh"               3 weeks ago         Up Less than a second
0.0.0.0:28000->27017/tcp                                 digitalbarracks_notification-mongodb_1
de7bb36e7968        digitalbarracks0/mongodb             "/init.sh"               3 weeks ago         Up Less than a second
0.0.0.0:24000->27017/tcp                                 digitalbarracks_details-mongodb_1
a63b4a5597de        digitalbarracks0/mongodb             "/init.sh"               3 weeks ago         Up Less than a second
0.0.0.0:23000->27017/tcp                                 digitalbarracks_iomanagement-mongodb_1
b25d7a787535        digitalbarracks0/notification-service   "java -Xmx200m -ja..."   3 weeks ago         Up Less than a second
0.0.0.0:8000->8000/tcp                                   digitalbarracks_notification-service_1
```

Fig. 6. Running containers.

Fig. 6 shows that all containers are started normally. CONTAINER ID is the number of container, IMAGE is the basic image, COMMAND is the operation instruction, CREATED is the time container created, NAMES is the container name, and PORTS is the port number. In addition to the business logic and database, there is a digitalbarracks_rabbitmq_1 with its basic image is rabbitmq: 3-management. RabbitMQ is an advanced message queuing protocol (AMQP) based on the complete, reusable enterprise information system [15], which can be used for efficient communication between modules of large software.

4.2. Delay Simulation

The new system and the original monolithic system are basically consistent with the business functions provided. It is difficult to differentiate the difference for the users. But people engaged in development, deployment and management can easily tell the difference between the two systems. This paper describes the development and deployment process in details. What's more, the business function and configuration information of the new system is verified. From the perspective of service management, the differences between the two systems are further elaborated, and the behavior characteristics of the system under load are tested.

Hystrix can push the monitoring indexes of each service to Turbine, and display the system behavior through the Hystrix dashboard. Fig. 7 shows the timeout threshold for setting the request response is 1000ms. When the different simulation responses are delayed, the system behavior of the account service invokes the inventory service.

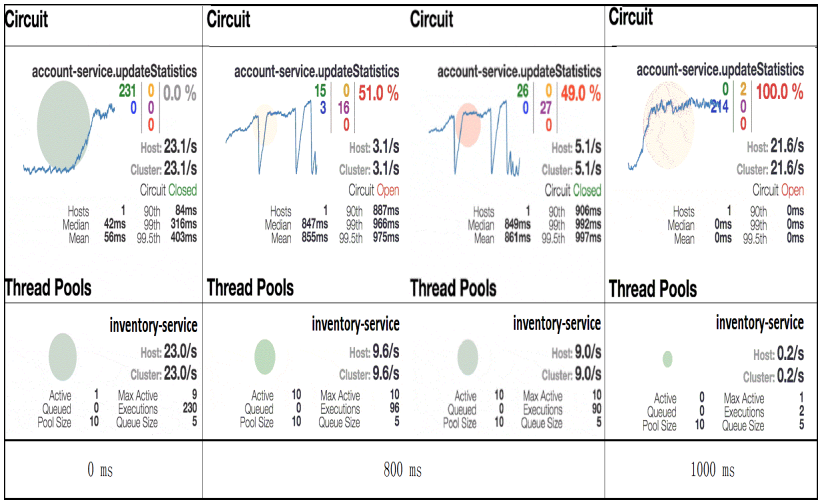


Fig. 7. System behavior under simulated delay.

In Fig. 7, when the analogue delay is 0ms, the median response time is about 50ms, the throughput is about 23 requests per second, with the inventory service in the small number of active threads and the circuit breaker closed. So, the request is in good condition. When the analogue delay is 800ms, the median time response is about 850ms, the ratio of fault request reached 50%, the inventory service is in a large number of active threads and the circuit breaker is of frequent switching between the shutdown and startup state. So, the request response is in poor state. When the analogue delay is 1000ms, 100% of the request fails with no active thread for Inventory service and the permanently open circuit breaker. As a result inventory service no longer processes the request. At this time, the request failure information can be timely returned, which is basically the same throughput and 0ms.

The above results show that: 1) when the system runs, if the response delay of the service is too high, the circuit breaker will automatically start to return the request failure information quickly to prevent cascading

failure; 2) when the response delay reaches a certain critical point, the circuit breaker controls the number of requests allowed through frequent opening/closing; 3) the failure of one service in the micro-service architecture system does not affect the normal operation of other services.

5. Conclusion

This paper probes into the functional characteristics and architecture characteristics of the inventory system. First, the split method of the monolithic system was proposed and the process was summarized into three steps: physical layering of three-tier architecture, identification of bounded context and elimination of integrated database. Then, the micro-service architecture system is constructed through four steps: service packaging, pulling docker images, configuration of multiple containers and creation of custom images. Finally, the new system was simulated and tested. The results show that the micro-service architecture system works normally with a stronger robustness than the monolithic system.

References

[1] Newman, S. (2015). Microservices, building microservices, loukides M., MacDonald B. (Eds.): O'Reilly Media, Inc., Sebastopol, 1-11.

[2] Dragoni N., Giallorenzo S., & Lafuente A. L. (2016). Microservices: yesterday, today, and tomorrow, 1-16.

[3] Namiot D., & Sneps-Sneppe, M. (2014). On micro-services architecture. *International Journal of Open Information Technologies*, *2(9)*, 24-27.

[4] Thönes, J. (2015). Microservices, software IEEE.

[5] Levcovitz, A., Terra, R., & Valente, M. T. (2016). Towards a technique for extracting microservices from monolithic enterprise systems, 1-8.

[6] Dautov, R., Paraskakis, I., & Stannett, M. (2014). Utilising stream reasoning techniques to underpin an autonomous framework for cloud application platforms. *Journal of Cloud Computing*.

[7] Uddin, I., Haque, H. M. U., Rakib, A., & Rahmat, M. R. S. (2016). Resource-bounded context-aware applications: A survey and early experiment. *International Conference on Nature of Computation and Communication*.

[8] Quiter, B. J., Ramakrishnan, L., & Bandstra, M. S. (2015). Grdc. a collaborative framework for radiological background and contextual data analysis.

[9] Verma, V., & Bhaskar, R. (2012). The research paper published by ijser journal is about an analysis of vertical splitting algorithms in telecom databases. *International Journal of Computer Applications*, 30-36.

[10] Cosmina, I. (2017). Spring microservices with spring cloud. Pivotal Certified Professional Spring Developer Exam. Apress.

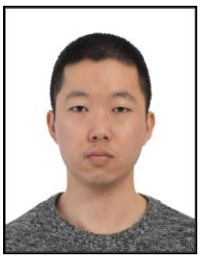
[11] Lamport, L. (2016). On interprocess communication. Part I: Basic formalism. *Distributed Computing*.

[12] Marwick, B. (2017). Computational reproducibility in archaeological research: Basic principles and a case study of their implementation. *Journal of Archaeological Method & Theory*.

[13] Xu, Q., Jin, C., Rasid, M. F. B. M., Veeravalli, B., & Aung, K. M. M. (2017). Decentralized content trust for docker images. *International Conference on Internet of Things, Big Data and Security*, 431-437.

[14] Boettiger, C. (2015). An introduction to docker for reproducible research. *Acm Sigops Operating Systems Review*, 71-79.

[15] Ionescu, V. M. (2015). The analysis of the performance of RabbitMQ and ActiveMQ. Roedunet.



Qinghui Ren received his master's degree in PLA University of Science and Technology, China, in 2016, and studying for a Ph. D. degree in science of military logistics. He is interested in software engineering & big data and computer science.

Shenglin Li received his B.S. degree in mathematics from Southwest University, China, in 1986, and Ph.D. degree in logistical engineering University of P.L.A China, in 2008. He is interested in information management engineering and computer science.

Han Qiao received his B. S. degree in PLA University of Science and Technology, China, in 2013, and studying for a Master's degree in Science of Military Logistics. He is interested in Computer Science.