DS Master

AnYi Huang	Minghan Ta	Qi Gao	Xiaoyan Cao
20202580	19209435	19209389	20203458

Synopsis:

We simulate a backend system of an online shopping site, which is a growing business. The site currently provides four services: displaying a homepage with bestsellers, displaying products from a specific category, displaying the content of a specific product and a search bar.

Due to the growth in the business, we must enable the system to handle the increasing workload. For example, it can serve more users or provide more services in the future. As a result, scalability must be considered when designing the architecture of this system.

As the system will get more complex and bigger in the future, it could be easier to have failure at some point in some components. Therefore, we want to make our system fault-tolerant, which means that it can experience failure (or multiple failures) in its components, but still continue operating properly.

Technology Stack

- SpringBoot

  It is mainly used for its auto-configuration functions for our project, which is done through adding annotations to our classes.

  i. @Component

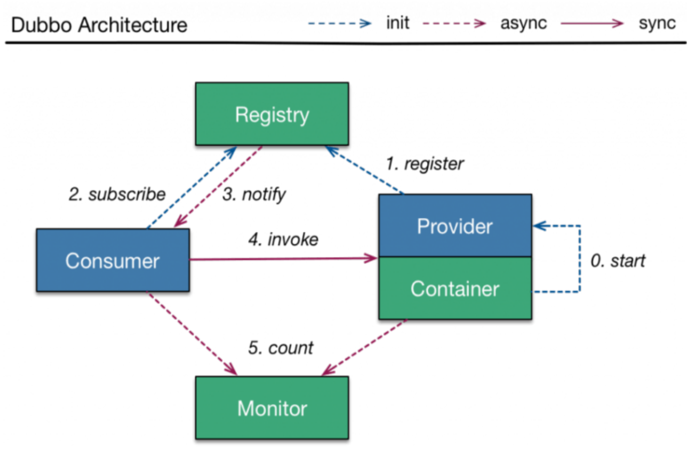
  We annotate our implementation classes of the services with this annotation to mark them as Spring beans so that they could be detected by Spring.

  ii. @RestController & @RequestMapping

  They enable us to create RESTFUL APIs in our controller. @RestController is used to map URIs to methods and associate them with a specific HTTP verb. @RequestMapping specifies the URI path and the verb of each API. For our project, all our APIs are linked with GET.

- Dubbo

  Dubbo is an open-source RPC and microservice framework, which perfectly suits our need to create a scalable, fault tolerant distributed system.



Our system follows the Dubbo architecture as shown in the diagram above (We did not set up the Monitor), where the provider corresponds to the servers of our four services and the consumer corresponds to our controller. We use ZooKeeper as the registry center.

Each service server is set up to register in ZooKeeper and Zookeeper will keep a list of the corresponding server addresses of each service. It will constantly update its list in case there are changes in the servers and inform the consumer of the latest server list so that when the consumer needs a certain service, it will use the load balance algorithm provided by dubbo to decide which server to invoke in the list.

- Docker

  Docker is a containerization technology that we use to deploy our project and simulate the connection of the different components. We created one container for the controller, one for ZooKeeper, and two containers for each service.

System Overview

- Main components:

  i. Core

     Core package contains the information that will be used by the other components, including the data class that represents product info, the data class that contains all the information on the products (equivalent to our database) and the four interfaces containing the methods to be implemented by the services.

  ii. Four services

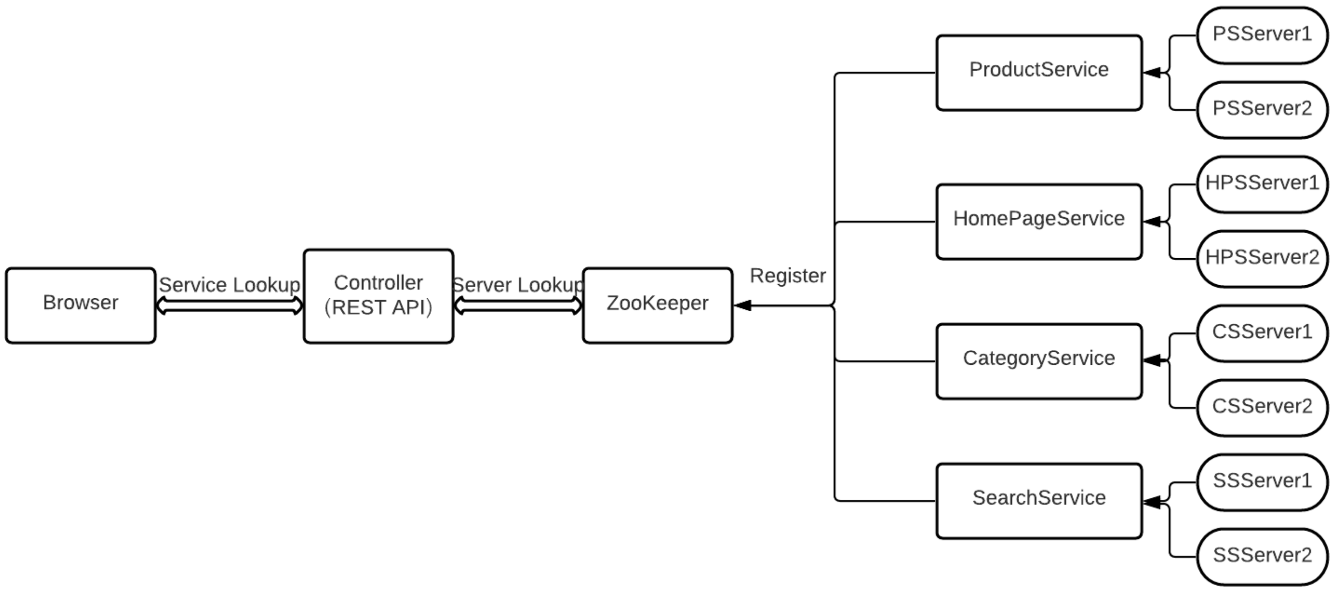
     These four packages each implement a service interface declared in the core package, including returning bestsellers, returning products from a specific category, returning the content of a specific product and returning the products according to the search information.

  iii. ZooKeeper

     Zookeeper serves as the registry center of the services. It maintains an active connection with all the servers of the services using a heartbeat mechanism. Therefore it could send the latest list of service servers to the consumer registered with this ZooKeeper.

  iv. Controller

     It subscribes to the services registered in the ZooKeeper and exposes them through different urls to the frontend of the website. It would invoke the server from the server list sent by ZooKeeper based on the configured load balance algorithm.



- Workflow of the System

  First, run a ZooKeeper instance. After that, run two servers for each service and register them through port 2181 of the ZooKeeper address. Therefore ZooKeeper will establish connections with all the servers and keep a record of their IP address and the service they correspond to. Then we execute the controller which serves as a consumer in the Dubbo architecture. It provides an API gateway for all the four services of our website backend and encapsulates them into RESTful APIs which can be accessed by the frontend through URLs. It will also be registered in ZooKeeper so that it could get the latest server list and save it. When the frontend sends a request, the controller will look up the server list of the corresponding service and choose one to invoke using the default random load balancing algorithm. At last, the controller will return the result of the invoked server to the frontend.

- Design for scalability

  As there are a lot of users we need to serve, our website needs to be able to cope with a large number of demands while maintaining high performance. In other words, good scalability is the priority of the system design. To do this, we introduce the mechanism of load balance. For each service, we would run multiple servers. Each server will register itself in the registry center so that they can expose their address to the controller. When the controller receives the request for a particular service, it would invoke one of the servers providing this service. This technique would distribute incoming traffic across a group of servers efficiently, so as to avoid the situation when some servers are overloaded while others are left idle.

  The algorithm we use for our system is the random load balancing algorithm which chooses the server on a random basis. It is able to distribute the requests evenly to the nodes when large traffic comes.

- Design for fault tolerance

  Fault Tolerance means a system's ability to continue operating uninterrupted despite the failure of one or more of its components. It is certainly one of our major concerns when designing the system. There are mainly two kinds of failures that our system manages to handle.

  The first one is crash failure which occurs when a server crashes or any other hardware related problem occurs. Our solution is to open multiple servers for one service so that when one server is down, ZooKeeper will lose its connection and take its address off the server list and report the latest list to the consumer. In this way, the failed server will not be invoked when a new request comes. In the meantime, our maintenance team would discover the change of the server list change in ZooKeeper and fix the errored server. When the server is restarted, it could be recognized by ZooKeeper again and works as normal.

The second failure is omission and timing failure. Omission occurs when a server does not receive incoming requests from clients or fails to send messages in response to clients' requests. Timing occurs when a server fails to respond in a particular amount of time. The Dubbo architecture comes with a timeout and reconnection mechanism that could solve this problem. If the provider does not respond within a given amount of time, the call is regarded to have failed, and the retry mechanism is invoked once more. The request is considered abnormal and an exception is thrown if it fails within the set number of calls. Moreover, when invoking the retry mechanism, the request will be directed to other servers instead of trying on the same server which guarantees the service quality.

Contributions

- AnYi Huang: Contributes to the product content service, the routing of controllers and the dockerization of the project. Also responsible for writing the group report.
- Minghan Ta (HankDa): Contributes to the category service.
- Qi Gao (Neo): Contributes to the establishment of the Zookeeper and SpringBoot framework and the searching service.
- Xiaoyan Cao: Contributes to the home page service, the routing of controllers. Also responsible for the group report and video.

Reflections

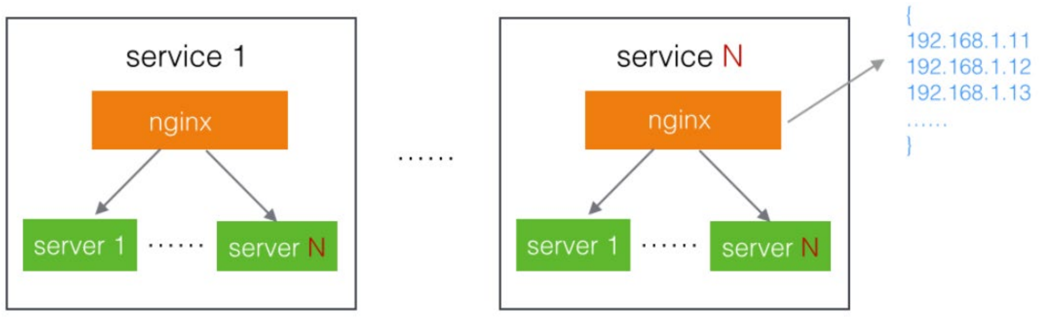
- The challenges and solutions

The challenge we have faced was making the SpringBoot system scalable and fault-tolerant. We find that using SpringBoot alone as we did in the quoco-rest lab cannot achieve this goal. The reason is that the broker component itself does not update the URL list of the service provider, once the system is already started. We cannot add a new service provider to the system without rebooting the whole system, which must reduce its availability. Also, when any of the service providers is not functioning, the whole system will be broken because the broker component cannot handle it.

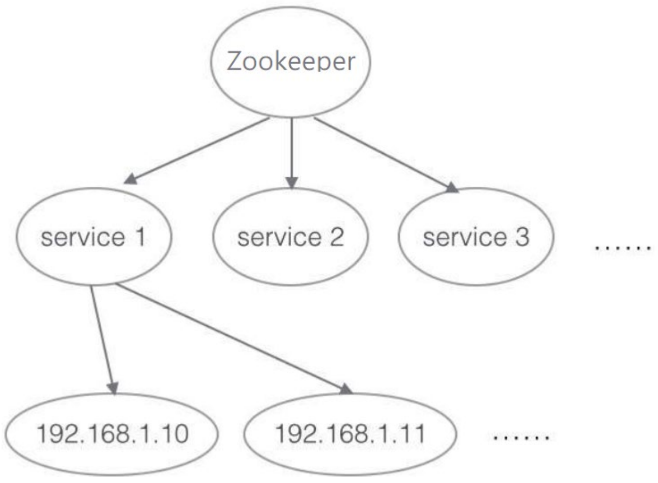
To overcome this problem, we decide to integrate a load balancer into our system. We compared the two solutions: Nginx and Zookeeper.

The Nginx solution is one of the commonly used software-based load balancers. The main advantage of it is that the configuration for Nginx is simple. We only need to set up the config file and run the Nginx server, then it can work as a load balancer to make our system scalable and fault-tolerant. Apart from the config file, no codes would be affected.

Although the Nginx seems to be simple, we did not choose to use this solution. There are two reasons. First, if any Nginx server is not functioning, the whole service will not be operating. Second, in our case, we expect to have multiple services in our system. This means that we will need multiple Nginx servers (one for each service) as the following diagram shows. The increase of the nodes in our system will cause an increase in the complexity of this system.



The Zookeeper, on the other hand, can be used as a registration centre. Every service provider itself has to send the registry query to Zookeeper and specify which service it belongs to. One Zookeeper server can maintain the address list for multiple services with multiple servers. Therefore, the number of nodes in this solution can be less than that of the Nginx solution. Another advantage is that since Zookeeper is only a registration centre, which helps update the service list and does not involve load balancing, the users (or consumers) of the services can still connect to the services based on the last update cache, even if the Zookeeper is accidentally shut down.



- What we could have done differently

  In a distributed system, one of the most important issues that developers have to deal with is the CAP theorem. This theorem states that, in a distributed system (a collection of interconnected nodes that share data.), you can only have two out of the following three guarantees across a write/read pair: Consistency, Availability, and Partition Tolerance.

  Because our system is a shopping site, shared data/databases must be used among the service providers. This means that we have to face the CAP problem in this aspect. However, we were focusing on solving the scaling and fault tolerance problem, so the data for the shopping site is only hard coded in our system at the moment, the CAP issue of it was not considered. If we have more time, what we could have done is imitate the system taking this CAP issue into account.

- Benefits and limitation of Zookeeper

  In conclusion, one benefit of the Zookeeper solution is that the failure of a single or a few systems does not make the whole system fail. Another benefit is that performance can be increased as and when needed by adding more machines with a minor change in the configuration of the application with no downtime. Also, it hides the complexity of the system and shows itself as a single entity/application.

  However, having only one Zookeeper server in a system is not ideal in the aspect of availability. When that server is not functioning, the whole system will be affected. Therefore, a cluster of Zookeepers is usually built to serve the system. To decide the leader of a Zookeeper cluster, a leader election must be performed. The process can be slow, and during the election, none of the Zookeeper servers is able to provide service to their clients. This could be a limitation of Zookeeper.

  Another limitation is consistency. In real-world situations, Zookeeper clients might not always be able to query data from Zookeeper, sometimes a cache mechanism will be used. This makes it difficult to have high consistency in the system.

  Furthermore, the authentication system of Zookeeper could not be satisfactory enough. Some developers even have to integrate an extra authentication tool into the system to strengthen the security. This could cause more maintenance costs on this system.