Team Group Name

Boshen Fan	Qiyan Xuan	Cheng Zhang	Xing Liu
<20211152>	<20211258>	<20211342>	<21201906>

Synopsis

With the popularity of e-commerce, more and more people are choosing to buy items on the Internet. Traditional shopping malls will discount items based on specific dates, during which mall traffic will increase. An example is Black Friday. The same scenario can be seen in the e-commerce sales model. When the products are on sale, more users will visit the website to buy the products. It is obvious that a stand-alone deployment is not able to carry the project. The project takes this scenario as a premise, and implements the scenario that will occur when the goods are on sale through technologies such as redis and message queues. Firstly, we use redis to solve the problem of inconsistent session in different servers after distributed deployment, use message queue to relieve the pressure on the server during the peak hours of specific user requests, and use redis to reduce the number of interactions between the server and mysql to improve the speed of the system.

Technology Stack

- *Java 8*

This project is written using the eighth version of java, which has better stability compared to other higher versions. And for lower versions of java, some of the new features of java8 are more suitable for this project. For example, stream computing, multi-threaded classes and so on

- *Springboot*

SpringBoot is widely used as a microservices development framework for java, it makes development more concise and fast, using automatic assembly technology eliminates the need for SpringMVC to receive the trouble of managing the Spring context

- *RabbitMQ*

RabbitMq is used to process requests asynchronously, so users will get faster replies and can distribute message handlers to more computers, improving the user experience.

- *Redis*

Redis, as a non-relational database, can be used to find the data users need from the computer more quickly by means of indexing.

- *Mysql*

Mysql being a relational database, it stores all the key information in the project. For example, users, products, product orders, etc.

- *Mybatis Plus*

MybatisPlus is used as a java connection to mysql database. And its reverse engineering helped us to automatically generate the code for the mapper, controller, and service layers.

- *Dockor*

Used to run multiple containers for distributed deployment

- *swagger*

Used to summarize and test interface functionality for team development.

- *lombok*

Constructors, get and set methods for auto-completion classes

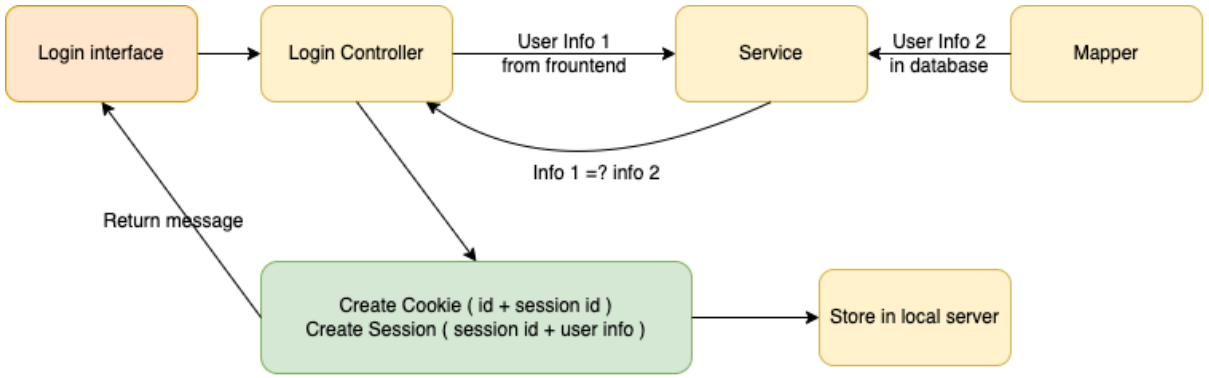
System Overview

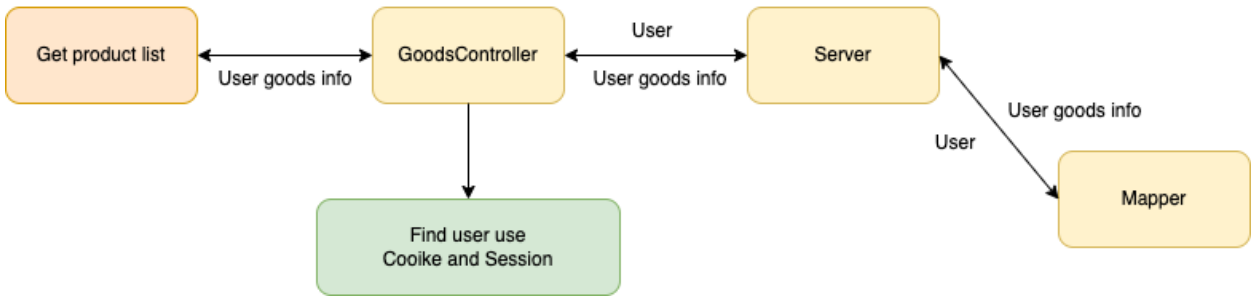
This project aims to improve the reliability of the system in the case of e-commerce spike campaigns by combining distributed technologies. For e-commerce systems, the single server service delivery model is obviously not applicable when the website has a large number of users and concurrent requests. The reason is that a single server has limited ability to handle requests, limited response time, and limited query time of mysql database, which makes the server less reliable. And a single server crash also faces problems in disaster recovery. This is where the combination of distributed technology is necessary.

1. *Redis implements distributed session - Traditional Steps*

   First of all, the login function is necessary for e-commerce systems. When a single server provides the service, session and cookie technologies are often used jointly to store the user's login status and information. This allows the user to access all interfaces within the site with the corresponding rights after a single login. This is due to the fact that all information about the user, including basic information about permissions, is recorded in the session. And the session is stored in the server. Imagine if we use distributed technology to let multiple servers provide services at the same time, when the user requests one of the servers, the server has the corresponding user information recorded locally. For the other servers in the cluster, the user is still not logged in. Imagine if there are 100 servers providing the service, in the worst case, the user will log in 100 times when requesting 100 interfaces in order to have the session of all the servers record that the user has logged in. This is obviously not what we want to see in a distributed system. To solve this problem I used a Redis database. The database is deployed on a separate server. When a user logs in for the first time, only the appropriate key-value pairs are stored on this Redis server. For the other servers, the redis servers are equal. They can all find the key in the key-value pair through a cookie and get the information stored in Redis.

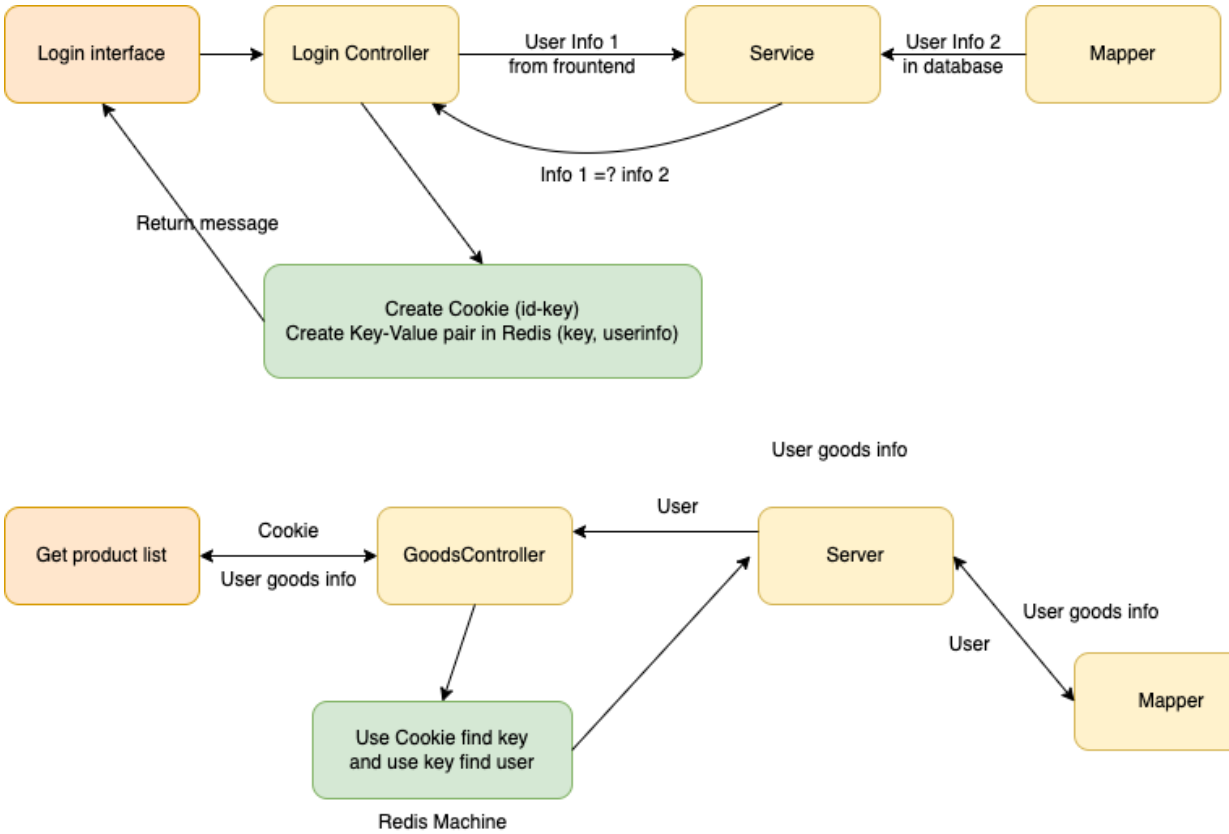
   When a user logs in, he or she fills in the username and password on the front-end page, and the front-end collects this information and sends a request to the controller, which calls the service to check the user's password, both through the mapper and by querying the password of the corresponding username in mysql. And compare the password passed by the front-end and the database query to see if the password is the same. If the password is the same, a cookie and a session are created in the controller, where the session includes the sessionid and the user's real information, and the cookie stores the cookieid and sessionid.



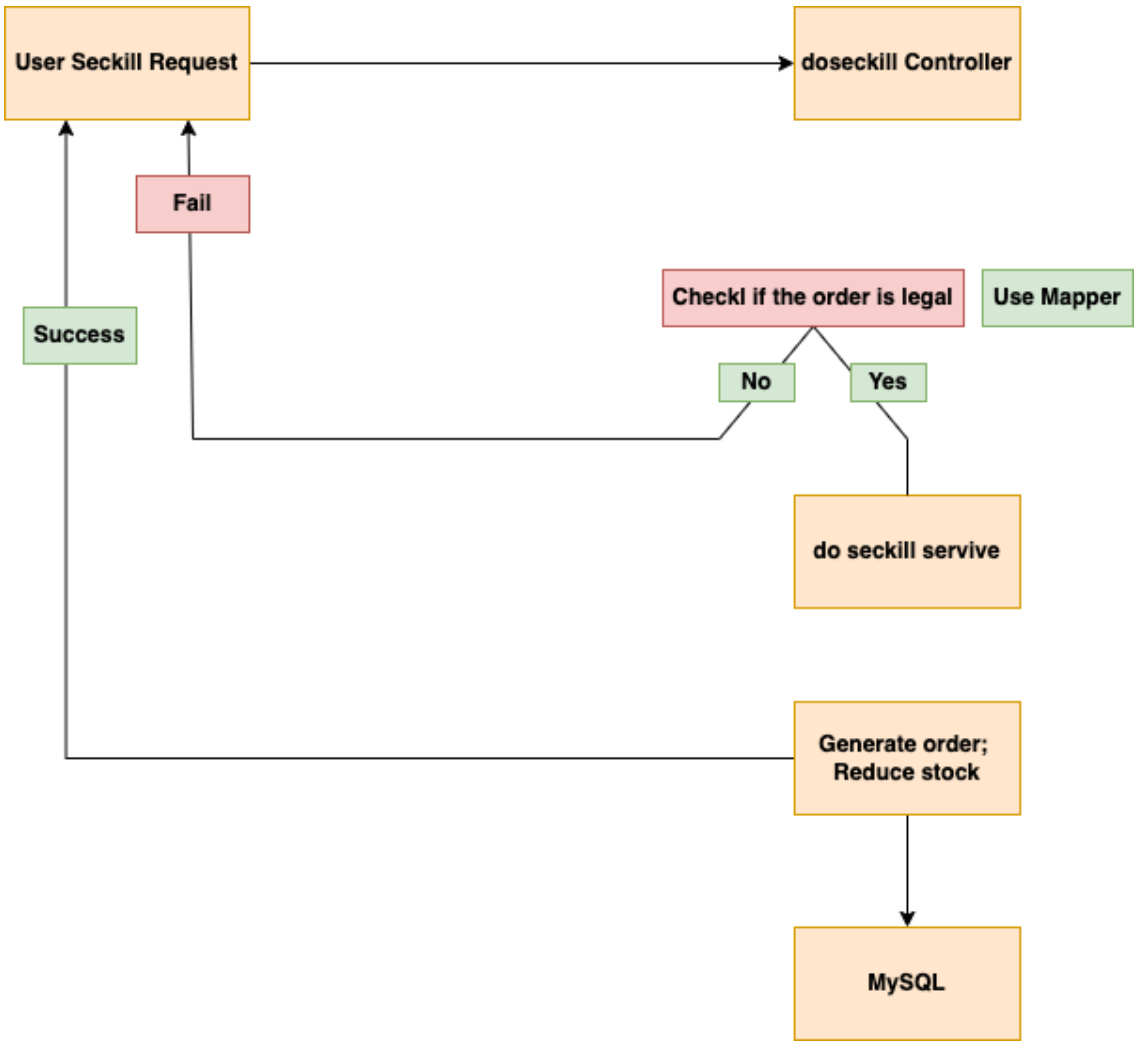


Redis implements distributed session - Improved version

The cookie is still generated when the user first logs in, but the session is replaced with a redis key-value pair and stored in the redis database. So the cookie stores the cookieid and the key in its redis key-value pair. When the second process occurs, the cookie is sent to the corresponding controller, which opens a thread to access the redis database to get the corresponding key-value pair. The value (i.e., the user's information) is obtained. In this way, session and cookie functions are implemented and redis can be shared among multiple servers. This solves the problem of distributed sessions.

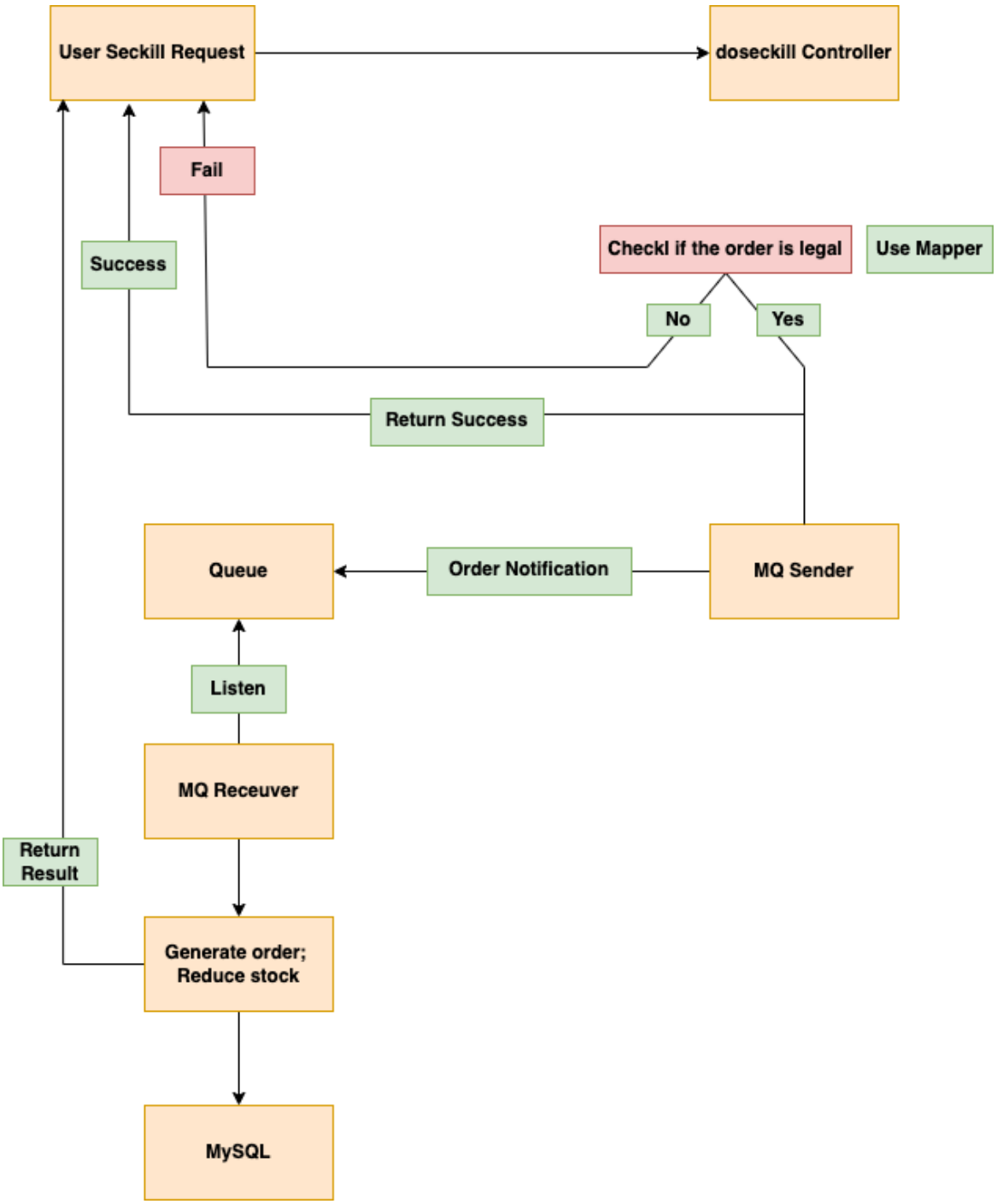


2. *Seckill Interface - Basic version*



As shown in the diagram, when the user logs in to the page, the server will send a request like the login controller just introduced, and the request will be sent with cookie and session information, and the backend will return the product list directly after verification, and the user will select the corresponding product that he wants to kill by clicking on the second request doseckillcontroller interface, in this interface will use the mapper to determine whether the order can be placed. The check logic is whether there is still a corresponding inventory, and whether the user has previously generated an order to buy the product, if the inventory is sufficient and the user is entering the order for the first time, then the inventory will be modified through the mapper and the order will be generated and stored in mysql. Everything is processed and then returned to the front end. In this traditional process we can see that the user does not get the feedback directly after clicking the second button, but only after the database is processed in the backend. If there are more and more users and the processing logic becomes complicated, the user will be stuck on the page waiting for feedback. This is not good for user experience. So in this section, our design is to see if the user can get feedback immediately after clicking the second button.

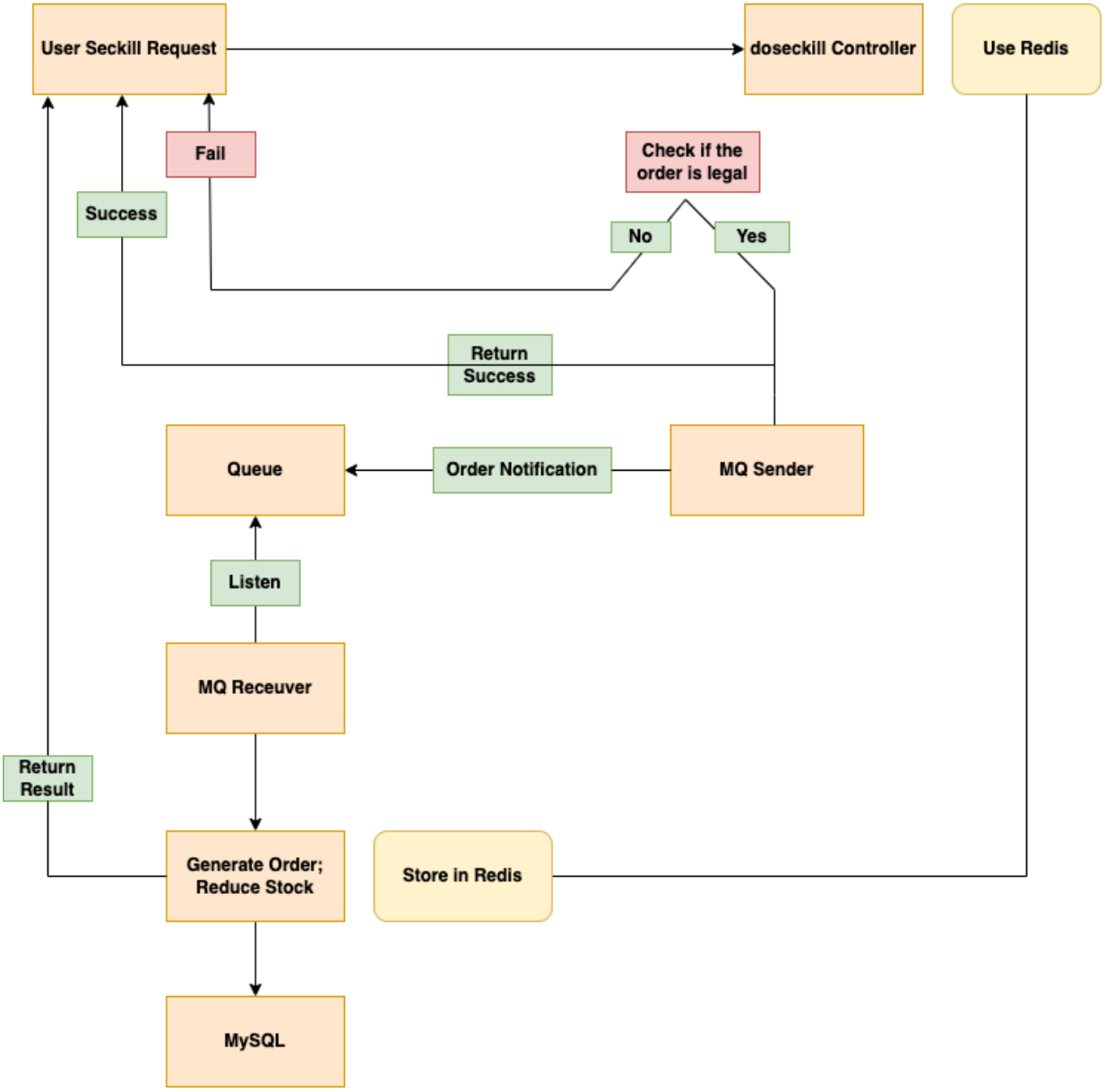
Seckill Interface - After optimization



To implement this feature we use RibbitMQ and decouple the above process by means of a message queue. When the user clicks on the doseckillcontroller, we need to add this event to the message queue immediately, and return the message immediately after it is added, while the spike is in progress. A thread will be opened in the background, and this thread will monitor the queue for messages to be generated. If there are any, they will be processed immediately. The processing logic is the same as in the traditional way. This is done by checking the legitimacy of the operation, manipulating the database, etc. Compared to the traditional way, the message queue allows the user to get immediate feedback on the operation and enhances the user experience. This is done through the asynchronous nature of the message queue. In subsequent development, we can use the consumer not as a thread, but as a separate server for operation. This also reduces the service pressure on the same server.

3. *Redis VS MySQL Performance Comparison*

Redis can be used not only for distributed sessions, but also for improving overall system performance. In this project redis is also used to reduce the pressure on the consumer thread to communicate with mysql. In the absence of redis, every user request, whether it is a legitimacy check or a real operation, requires frequent access to mysql, which does not have a high performance compared to our controller. So this makes the user wait longer for a response.



In this project, Redis is used to pre-reduce the stock as shown in the diagram, recording the user's order directly through redis when the user first generates it and loading the mysql inventory into redis when the system is first started. In the previous section, we discussed the exact flow of the spike process. The process now is that all operations are performed on redis. The data from redis is flushed to mysql at regular intervals. When the system starts, all the inventory of all the products is added to redis. Then the consumer thread for each task first looks up the remaining quantity of the corresponding item in redis. It also queries the redis to see if the user already has an order for that item. If the check passes, the quantity of the corresponding product stored in redis is modified directly. The order is generated and stored in redis. If the user's next request is satisfied, the order is checked. This solves the problem of slow queries in relational databases like

mysql through the fast access speed of redis. This further improves the availability and reliability of the system.

4. *Swagger*

   We introduced the swagger module to better manage all the rest style api . It is necessary to manage the interfaces in order to avoid the concentration of problems in the development process. In distributed projects, due to the complexity and size of the services provided by the interfaces, an effective interface management tool is the basis for project implementation.

   Swagger provides real-time updated API instead of using git and word to replace the development process for project planning documents. This can be especially useful during development and testing stages, as you can easily send requests and see the responses without having to use a separate tool like Postman.

Contributions

Boshen Fan

- *Project Architecture Design*
- *Apply RabbitMq to provide asynchronous processing to the system*
- *Write a consumer-producer to complete the user's order placement operation*
- *Seckill interface security optimization*
- *tool class writing*

Qiyan Xuan

- *Integrating Redis in project*
- *Implementing a Distributed System Login Module*
- *Distributed Sessions with Redis*

Cheng Zhang

- *Optimizing System Performance with Redis instead of MySQL*
- *Pre-reducing stock with Redis*
- *Caching order information with Redis*

Xing Liu

- *design controller*
- *implement swagger API*

Reflections

Difficulty 1: Controller layer optimization

The controller layer, as an important component to receive information from and feed back to the front-end, we found a lot of code redundancy during the development process. For example, when we need to feed a fixed error to the front-end, we need to write this error n times. To solve this problem we use enumeration classes. This is used to specify the message code for each message returned to the front-end and to standardize the message return format.

Difficulty 2: Different systems have different session in distributed system

In the process of testing, we found that when a user connects to a server, the session will be stored in the server that the user visits. But because of the distributed system, the server that the user accesses for the second time and the server that the user accesses for the first time are not guaranteed to be the same. So when the user accesses for the second time, the login will be invalid. To solve this problem, we introduced redis, a solution that stores all the information that would otherwise be stored in the session in Redis.

Difficulty 3: Mysql's slow query

When the database has a lot of data, since Mysql's search is based on the B+ tree, the time complexity of finding a record is O(LogN), but in extreme cases, when the paging size is large or the index is not appropriate, the search time is huge. So we copy as much of the Mysql data as possible into Redis, such as the number of items, user logins, etc. Redis uses indexed storage to optimize the speed of lookups.

The project we used Rest style api but due to time constraints we did not write the front-end pages using the front-end engineering framework. Instead, we used theleft framework for writing. In future projects we can further decouple the project as much as possible to achieve the separation of front and back ends. The front-end project is deployed separately. This can further reduce the pressure on the server. In other areas we need to consider the deployment strategy of the project in different situations. Currently we deploy Redis, RabbitMq, Mysql and the backend of the project on separate computers (replacing them with docker containers) but the backend project is only deployed on one server. We need to use cluster management tools such as K8S to dynamically allocate nodes to improve the speed of the system.