CS237 Project Report - Group 5

Distributed SecKill Service Based on Redis

Qirui Yang
Mingyuan Yang
Jiannan Tan

1 Introduction

The rapid development of e-commerce prompted the birth of the SecKill activity. In a SecKill activity, a limited number of products will be sold at varying degrees of discount, which brings a huge temptation for customers [1]. The discounted products are usually sold out in seconds, which means high concurrency and high availability in a short time. It can be a huge challenge for ecommerce systems. In this case, a SecKill system with high concurrency and high availability has very practical significance and has gradually become one of the core modules of a large-scale e-commerce platform.

In this project, we aim to design and implement a SecKill system based on Redis to mock the scenario like Black Friday where a large number of customers snap up a small number of super-cheap products successfully at a specified time. In addition, in order to make sure the reliability and availability of the entire system, we will do the function test and performance test of the entire system.

Obviously, when designing a Seckill system, we need to control purchase traffic requests and only allow a small number of requests to enter the backend service, add cache technology to convert instantaneous high traffic to stable traffic, and adopt asynchronous processing mode to improve system concurrency. Therefore, the key point to achieve Seckill system is to control the thread's competition for resource.

In a stand-alone architecture, we can use "synchronized" to achieve mutual exclusion of goods. However, in a distributed system, there will be multiple machines in parallel to achieve the same function. In other words, if processes locks are used to concurrently access database resources in multiple processes, the oversold situation will occur. Therefore, we need to introduce distributed locks. In this project, we implement distributed locks based on Redis. More specifically, we improve the distribution of workload to maximize throughput and improve efficiency through implementing load balancer. In addition, we use message queues to place orders asynchronously to reduce the processing time of a single request and improving throughput.

The rest of the report is organized as follows. Section 2 describes an overview of the related works. Section 3 introduces the methods and techniques used in our project. Section 4 evaluates the feasibility of the projects and analyzes the results. Section 5 introduces the conclusion of this project and future work.

2 Related efforts

Considering implementing distributed lock is the core part of this project, in order to better understand distributed locks and choose the better method to design our SecKill system, we research three popular distributed locks which are widely used in industry, including distributed locks base on MySQL, distributed locks base on Redis, and distributed locks base on Zookeeper. In addition, we also research load balancing, which is used to solve with the problems of high concurrency and high availability in the internet architecture. It is necessary to explore such technology to meet the concurrency requirement of our project.

2.1 Distributed locks

The first method we researched is implementing distributed locks based on MySQL. One feasible method to implement distributed locks in MySQL is storing data in a specific table, where method_name and value will be stored. However, MySQL distributed locks are generally applicable to the specific situation where resource does not exist in database. MySQL distributed locks have some advantages, like simple to understand, no need to maintain additional middleware. However, this kind lock also has limitations, including we need to consider lock timeout and add transactions manually. More importantly, the performance of this method is limited to the database and it is not suitable for high concurrency scenarios, which do not obey our aim of this project.

The second method we explored is implementing distributed locks based on Zookeeper. Zookeeper create a number of nodes before each lock operation. And it needs to release nodes when completion, which will waste a lot of time. We explored how to implement distributed lock service using Zookeeper in this paper [2]. And we know that there are two methods which make Zookeeper work. The first one is using the uniqueness of node name to realize the shared lock. In this method, if many clients are waiting for a lock, when the lock is released and all clients are awakened, only one client gets the lock, which is very inefficient. The second one is using the temporary sequential node to realize the shared lock [3].

In this method, there are a lot of repeated and useless work during the whole competition, which has damage impact on the server when the cluster is large.

The third method we explored is implementing distributed locks based on Redis. We know that Redis is a high-performance key-value database [4], which supports the persistence of data. Redis not only supports simple key-value type data, but also provides storage of data structures such as list, set, string, and hash, which is more flexible in implementing. Through research, we know that Redis has extremely high performance: Redis can read at 110,000 times/s and write at 81,000 times/s [5]. Compared to MySQL and Zookeeper, Redis has better performance in distributed system. Therefore, our group implement the distribute locks based on Redis.

2.2 Load balancing

Load balancing refers to efficiently distributing incoming network traffic across a group of backend servers. More specifically, a load balancer is responsible for routing client requests

across all servers capable of fulfilling those requests in a manner that maximizes speed and capacity utilization and ensures that no one server is overworked, which could degrade performance. Popular websites cannot rely on a single powerful server [6]. If a single server or machine goes wrong, the load balancer redirects traffic to the remaining online servers. Therefore, load balancing is very important in a SecKill system.

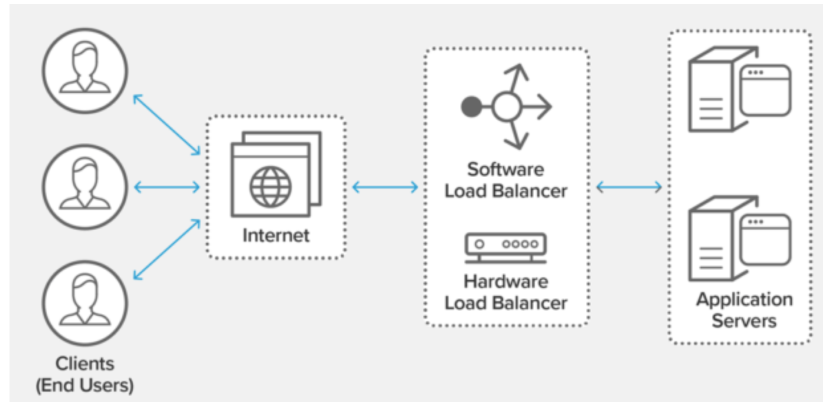

Figure 2-1: load balancing diagram

There are some load balancing algorithms which provide different benefits. For example, Round Robin algorithm refers to requests are distributed across the group of servers sequentially. Least Connections algorithm refers to a new request is sent to the server with the fewest current connections to clients, and the relative computing capacity of each server is factored into determining which one has the least connections. Hash algorithm refers to distributes requests based on a key you define, such as the client IP address or the request URL, NGINX Plus can optionally apply a consistent hash to minimize redistribution of loads if the set of upstream server changes [7].

3 Design architecture

3.1 techniques

Spring Cloud [8] offers a full set of distributed system solutions. Spring Cloud encapsulates multiple open source components Netflix in the microservices infrastructure framework, while integrating with the cloud platform and with the Spring Boot development framework. Spring Cloud provides developers with the tools to build distributed systems quickly. Developers can quickly launch services or build applications, and quickly connect with cloud platform resources.

In the Spring Cloud we use here, we use Eureka and Ribbon. Eureka is a service discovery component developed by Netflix and is a REST based service. Spring Cloud integrate it in its subproject spring-cloud-Netflix to achieve Spring Cloud service discovery functionality.

Eureka is a client service discovery mode that provides both Server and Client components. Eureka Server, as the role of the service registry, provides registration and queries of REST API to manage service instances. POST request is used for service registration, PUT request is used to implement heartbeat mechanism, DELETE request service registry removes instance

information. Eureka Client is Java implemented Eureka client, in addition to convenient integration, but also provides a relatively simple Round-Robin Balance. In this project, we use Eureka and Netflix Ribbon [9], to realize the complex balance strategies based on traffic, resource occupancy and provide more reliable elastic guarantee for the system.

Spring Boot is an open source Java-based framework used to create a micro Service. It is developed by Pivotal Team and is used to build stand-alone and production ready spring applications. Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need minimal Spring configuration. So, we user Spring Boot to implement our project.

As we discuss before, According to Redis homepage, Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports various data structures such as Strings, Hashes, Lists, Sets etc. It is very fast. So we use it to implement SecKill this function.

Above all, we use Eureka to implement as register center, Ribbon to realize load balance, Spring boot to realize our system structure, MySQL as the database and Redis as the middleware between back-end service and database.

3.2 simulation platforms

The project is a high concurrent website program. So we need to test its performance. The Apache JMeter [10] is pure Java open source software, which is firstly designed to load test functional behavior and measure performance. We can use JMeter to analyze and measure the performance of web application or a variety of services. Performance Testing means testing a web application against heavy load, multiple and concurrent user traffic. JMeter originally is used for testing Web Application or FTP application. Nowadays, it is used for a functional test, database server test etc.

It's not feasible to arrange 100 people with PC and internet access simultaneously accessing google.com Think of the infrastructure requirement when you test for 10000 users (a small number for a site like google). Hence you need a software tool like JMeter that will simulate real-user behaviors and performance/load test your site.

3.3 frameworks

The first step is to check sold-out map in SecKill provider. If the answer is true, it needs to check if this client can buy this item in Redis. If this can be done, we need to try to get lock of this item and then but it and lock. At the end, we still need to unlock this value in Redis.

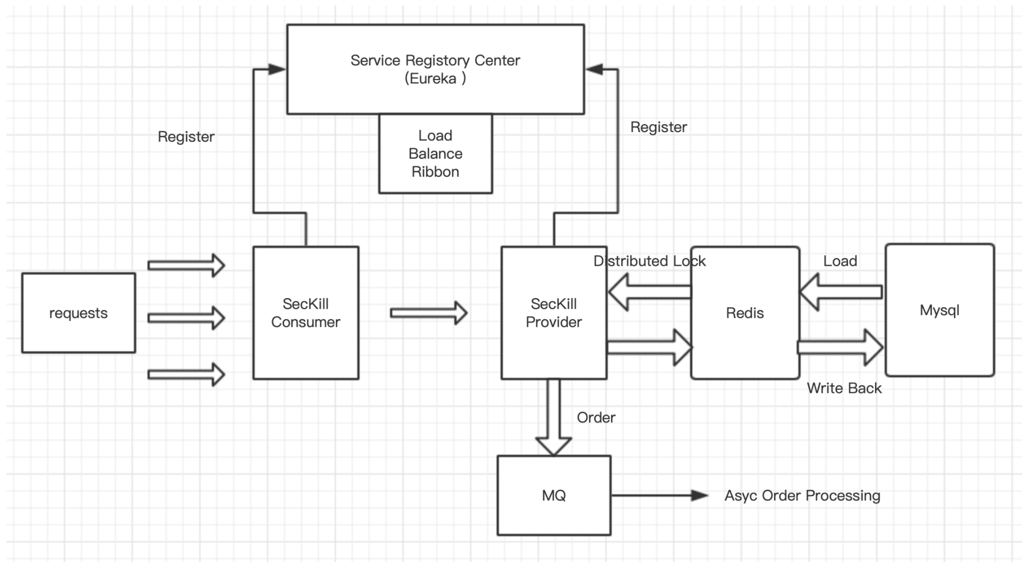

Figure 3-1: SecKill system diagram

From the figure above, SecKill consumer(requests) and SecKill provider register in the service Registering Center (Eureka). When requests arrive firstly at Eureka, Eureka will send information of SecKill provider to SecKill consumer. And SecKill consumer use Ribbon to choose which server will serve this request, which is a local load balance. In this way, requests are sent to the backend. The clients' order evenly allocated to distributed servers.

These requests would be processed by controller and service-level code. In order to respond to such high concurrent real-time requests, we can use Redis as the cache of the MySQL database, and use the single-threaded feature of Redis to prevent the situation of selling more or less in distributed scenarios. Also, because Redis runs in memory, it would speed up the access of database. At the same time, JVM-level cache flags can also be introduced to speed up. We can design this flag in the SecKill provider. When no more item in Redis, we can update the value of this flag. In the next time, when request for this item arrives, it will check this flag firstly and will not access Redis, which will cost less time. After getting the message from stock database and Redis, finally, using message queues to complete asynchronous order services can further reduce the pressure on the database, which means getting goods successfully did not directly write data into the order database. In this way, we can implement a SecKill system efficiently.

4 Evaluation

4.1 Evaluation plan

As we mentioned before, we use Jmeter to evaluate our project. Because our project is a high concurrent project, we need a lot of requests to test the performance of the project. We set the parameter as 30000 requests and 200 items.

But also, we use different ports to simulate Distributed Servers because we only have one computer.

4.2 Evaluation Result and Problem

When we did not finish our project and we do not use Redis, we use Jmeter to test the result as following:

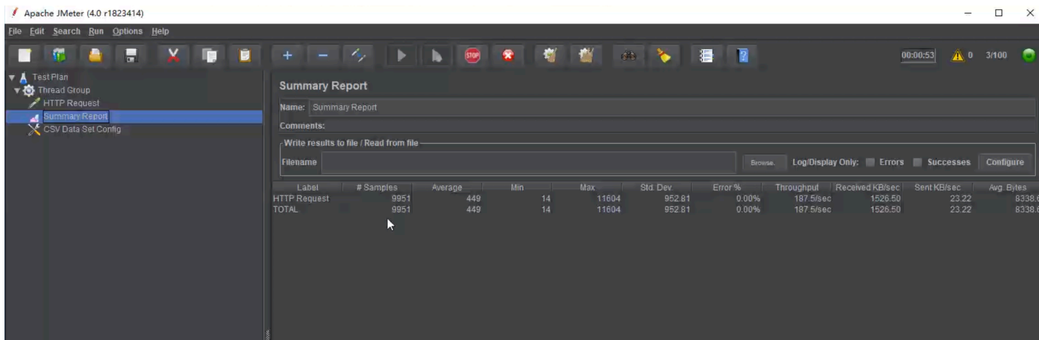

Figure 4-1: test result

From Figure 4-1, We can see that the throughput is 187.5/sec. It is not an efficient value. If we have 1875 requests, it will take us 10 seconds to finish service these requests. So we need to use Redis to improve our project.

From the Figure 4-2 and Figure 4-3, we find that 200 items are sold out. So we realize the function of Seckill project. There are 30000 requests but only 200 of them get the good successfully.

And from the analysis of Jmeter, we can see that the new throughput is 343/sec. Therefore, we have made an improvement. And For the response time of requests, the high one is narrow, which means most response time is short.

To summarize, the project has the ability to meet functional requirements.

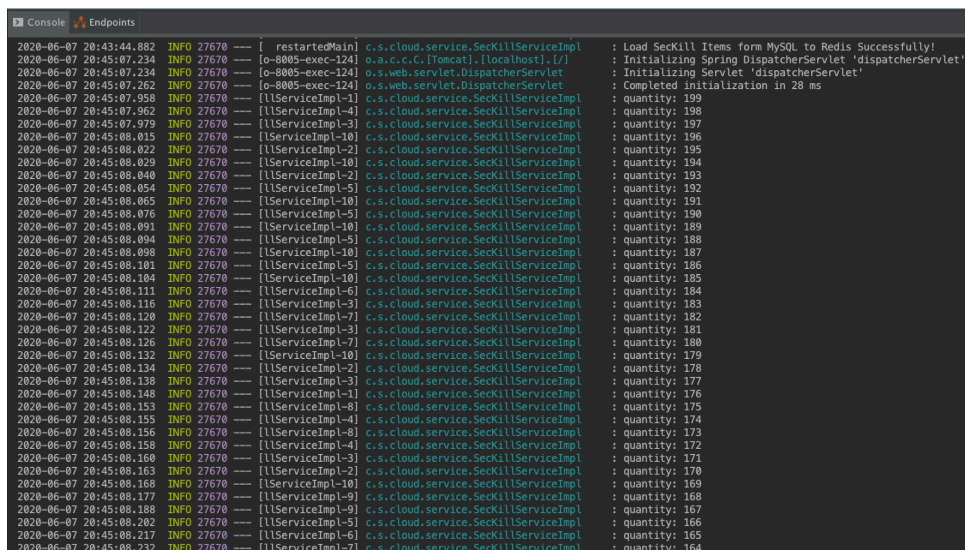

Figure 4-2: test result

```
Console  Endpoints
2020-06-07 20:45:14.578  INFO 27670 --- [llServiceImpl-2] c.s.cloud.service.SecKillServiceImpl     : quantity: 23
2020-06-07 20:45:14.579  INFO 27670 --- [llServiceImpl-6] c.s.cloud.service.SecKillServiceImpl     : quantity: 22
2020-06-07 20:45:14.582  INFO 27670 --- [llServiceImpl-1] c.s.cloud.service.SecKillServiceImpl     : quantity: 21
2020-06-07 20:45:14.583  INFO 27670 --- [lServiceImpl-10] c.s.cloud.service.SecKillServiceImpl     : quantity: 20
2020-06-07 20:45:14.585  INFO 27670 --- [llServiceImpl-5] c.s.cloud.service.SecKillServiceImpl     : quantity: 19
2020-06-07 20:45:14.586  INFO 27670 --- [llServiceImpl-4] c.s.cloud.service.SecKillServiceImpl     : quantity: 18
2020-06-07 20:45:14.589  INFO 27670 --- [llServiceImpl-9] c.s.cloud.service.SecKillServiceImpl     : quantity: 17
2020-06-07 20:45:14.591  INFO 27670 --- [llServiceImpl-7] c.s.cloud.service.SecKillServiceImpl     : quantity: 16
2020-06-07 20:45:14.593  INFO 27670 --- [llServiceImpl-2] c.s.cloud.service.SecKillServiceImpl     : quantity: 15
2020-06-07 20:45:14.594  INFO 27670 --- [llServiceImpl-6] c.s.cloud.service.SecKillServiceImpl     : quantity: 14
2020-06-07 20:45:14.595  INFO 27670 --- [llServiceImpl-1] c.s.cloud.service.SecKillServiceImpl     : quantity: 13
2020-06-07 20:45:14.597  INFO 27670 --- [lServiceImpl-10] c.s.cloud.service.SecKillServiceImpl     : quantity: 12
2020-06-07 20:45:14.599  INFO 27670 --- [llServiceImpl-5] c.s.cloud.service.SecKillServiceImpl     : quantity: 11
2020-06-07 20:45:14.600  INFO 27670 --- [llServiceImpl-4] c.s.cloud.service.SecKillServiceImpl     : quantity: 10
2020-06-07 20:45:14.603  INFO 27670 --- [llServiceImpl-9] c.s.cloud.service.SecKillServiceImpl     : quantity: 9
2020-06-07 20:45:14.604  INFO 27670 --- [llServiceImpl-7] c.s.cloud.service.SecKillServiceImpl     : quantity: 8
2020-06-07 20:45:14.607  INFO 27670 --- [llServiceImpl-2] c.s.cloud.service.SecKillServiceImpl     : quantity: 7
2020-06-07 20:45:14.609  INFO 27670 --- [llServiceImpl-6] c.s.cloud.service.SecKillServiceImpl     : quantity: 6
2020-06-07 20:45:14.610  INFO 27670 --- [llServiceImpl-1] c.s.cloud.service.SecKillServiceImpl     : quantity: 5
2020-06-07 20:45:14.612  INFO 27670 --- [lServiceImpl-10] c.s.cloud.service.SecKillServiceImpl     : quantity: 4
2020-06-07 20:45:14.614  INFO 27670 --- [llServiceImpl-5] c.s.cloud.service.SecKillServiceImpl     : quantity: 3
2020-06-07 20:45:14.616  INFO 27670 --- [llServiceImpl-4] c.s.cloud.service.SecKillServiceImpl     : quantity: 2
2020-06-07 20:45:14.618  INFO 27670 --- [llServiceImpl-9] c.s.cloud.service.SecKillServiceImpl     : quantity: 1
2020-06-07 20:45:14.619  INFO 27670 --- [llServiceImpl-7] c.s.cloud.service.SecKillServiceImpl     : quantity: 0
2020-06-07 20:45:14.621  INFO 27670 --- [llServiceImpl-2] c.s.cloud.service.SecKillServiceImpl     : sold out
2020-06-07 20:45:14.622  INFO 27670 --- [llServiceImpl-7] c.s.cloud.service.SecKillServiceImpl     : sold out
2020-06-07 20:45:14.622  INFO 27670 --- [llServiceImpl-2] c.s.cloud.service.SecKillServiceImpl     : sold out
2020-06-07 20:45:14.623  INFO 27670 --- [llServiceImpl-1] c.s.cloud.service.SecKillServiceImpl     : sold out
2020-06-07 20:45:14.623  INFO 27670 --- [llServiceImpl-2] c.s.cloud.service.SecKillServiceImpl     : sold out
```

Figure 4-3: test result

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/sec	Sent KB/sec
HTTP Request	64678	1958	1565	4002	4214	5902	1	10818	4.20%	343.9/sec	137.77	46.97
TOTAL	64678	1958	1565	4002	4214	5902	1	10818	4.20%	343.9/sec	137.77	46.97

Figure 4-4: test result

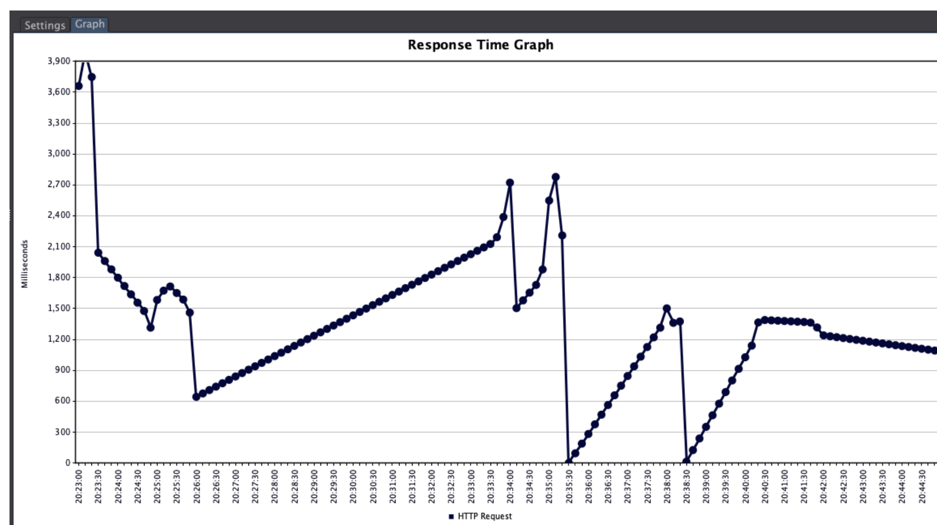

Figure 4-5: test result

5 Conclusion and Future work

In this project, we explore how to design and implement a SecKill system. We research and compare several distributed locks methods, and finally use the Redis to complete this project. In addition, we also implement the load balancing function by introducing the Ribbon algorithm. Through observing the test data and doing jmeter analysis, our SecKill system meets the requirement

For the future work, there are still some areas could be improved. As a commercial project, our current throughput is not high enough to meet the commercial requirement, so we can optimize our project to improve the performance of entire system. In addition, researching the test data from the current system, we know that our average processing time is a little long, and the difference between the shortest time and the longest time is a little big. For users to have a good shopping experience, we will shorten the average request processing time in the future work.

6 Reference

[1] Zhu, Liye. "Configurable e-commerce-oriented distributed seckill system with high availability." *AIP Conference Proceedings*. Vol. 1955. No. 1. AIP Publishing LLC, 2018.

[2] Dongmei Chen, Guangyan Chang. ZooKeeper development and application. 2017(21):35-36+42.

[3] Paksula, Matti. "Introduction to store data in Redis, a persistent and fast key-value database." Proceedings of AMICT 2010-2011 Advances in Methods of Information and Communication Technology (2010): 39.

[4] Nguyen, Thanh Trung, and Minh Hieu Nguyen. "Zing database: high-performance key-value store for large-scale storage service." *Vietnam Journal of Computer Science* 2.1 (2015): 13-23.

[5] Zhanlin Ji, Ivan Ganchev, Máirtín O'Droma, Tiefan Ding, "A Distributed Redis Framework for Use in the UCWW", Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC) 2014 International Conference on, pp. 241-244, 2014.

[6] Cardellini, Valeria, Michele Colajanni, and Philip S. Yu. "Dynamic load balancing on web-server systems." *IEEE Internet computing* 3.3 (1999): 28-39.

[7] Stoica I, Morris R, Karger D, et al. Chord: A scalable peer-to-peer lookup service for internet applications[J]. ACM SIGCOMM Computer Communication Review, 2001, 31(4): 149-160.

[8] Zhang Mingsen, Huang Hongmin, Zhan Sweden. The Design and Implementation of Advertising System Based on Spring Cloud Micro-service Architecture [J]. Electronic World ,2020(08):165-166.

[9] Chi Temple Committee. Realization of Ribbon Based Micro Service Communication and Load Balancing [J].] Computer and Information Technology ,2019,27(05):25-27.

[10] Bian Nai-zheng, Zhao Dongxu. JMeter Web Service Automation Testing Integration Framework [J].] Computer Applications and Software ,2016,33(05):8-12 16.