Distributed Transaction Settlement System

516030910082 冯二虎

516030910293 姚子航

515015910005　丁丁

516030910375 蔡一凡

目录

1. 系统环境概述

我们的分布式事务处理系统（DTSS）部署在一个由多台虚拟主机组成的集群中，这个集群通过openstack平台统一创建与管理。

以下是该集群的具体配置:

- 节点数及名称: 4 （vm1 vm2 vm3 vm4）
- 节点实例资源：4vCPU + 8GB内存空间 + 80GB硬盘空间
- 四个节点之间可通过集群网络相互通信，节点VM1绑定了浮动ip，外部网络可通过端口30430-30439与VM1通信

分布式事务处理系统由以下多个组件组成:

- **Hadoop集群**

Hadoop集群搭建在四个节点上，vm1为namenode，vm2，vm3，vm4为datanode，为Spark集群提供分布式存储以及yarn调度的支持。

- **Spark集群**

Spark集群搭建在四个节点上, vm1为spark-master, vm2, vm3, vm4为spark-worker，本系统使用Spark平台的Spark Streaming组件完成订单数据流的处理及计算。

- **Kafka消息队列**

Kafka集群搭建在vm2，vm3，vm4三个节点上，Kafka消息队列连接web API服务与Spark集群，缓冲订单消息。

- **Zookeeper集群**

Zookeeper集群搭建在vm2，vm3，vm4三个节点上，为Kafka集群提供协调服务，为事务处理系统提供分布式锁服务及配置管理服务。

- **Mysql集群**

Mysql集群搭建在四个节点上，vm1为管理节点，vm2，vm3，vm4为存储节点及SQL节点，为事务处理系统提供数据库存储服务。

- **Order Receiver Web服务**

事务处理系统对用户开放Http接口，receiver服务运行在vm2，vm3，vm4节点上，vm1运行Nginx作为网关及负载均衡器

2. 环境搭建

2.1 基础环境

Hadoop，Spark这些平台的运行都需要一些基础环境，这里不再详述安装过程，基础环境如下：

- Java 1.8.0_212
- Scala 2.11.0

2.2 节点间SSH免密通信

事务处理系统中的大部分组件都搭建在分布式集群上，这些集群启动时往往需要通过SSH进行数据传输，执行指令等操作，所以我们首先需要配置四个节点之间SSH的通信。

- 编辑`/etc/host`文件使得我们不需要用ip来识别节点。

```
10.0.0.77   vm1
10.0.0.154  vm2
10.0.0.137  vm3
10.0.0.115  vm4
```

- 在每个节点上使用指令`ssh-keygen -t rsa`为每个节点生成密钥对，再将所有节点的公钥一起拷贝到每个节点的`.ssh/authorized_keys`文件中，这样就实现了四个节点之间SSH免密登录。

2.3 搭建Hadoop集群

- 从官网下载`hadoop-3.1.2.tar.gz`安装包，解压至`/usr/local/hadoop`文件夹。
- 编辑`hadoop-env.sh`，`yarn-env.sh`，`core-site.xml`，`hdfs-site.xml`，`maprd-site.xml`，`yarn-site.xml`等配置文件，配置文件内容不在此展示。
- 编写`slaves`文件，添加vm2，vm3，vm4作为datanode。
- 将配置好的hadoop文件夹通过scp分发给vm2，vm3，vm4。
- 使用命令`bin/hadoop namenode -format`格式化namenode。
- 使用命令`sbin/start-all.sh`启动hadoop集群，通过`jps`命令可以看到namenode,datanode正常启动。

```
master (vm1)
[centos@vm1 ~]$ jps
5714 SecondaryNameNode
5963 ResourceManager
5455 NameNode
```

```
slaves (vm2, vm3, vm4)
[centos@vm2 ~]$ jps
5255 NodeManager
5131 DataNode
```

2.4 搭建Spark集群

- 从官网上下载`spark-2.4.3-bin-hadoop2.7.tar.gz`安装包。
- 在`conf/spark-env.sh`添加配置配置hadoop目录及Spark集群配置。

```
export SCALA_HOME=/usr/lcoal/scala-2.11.0
export JAVA_HOME=/usr/local/java
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
SPARK_MASTER_IP=vm1
SPARK_LOCAL_DIRS=/home/centos/spark-2.2.1-bin-hadoop2.7
```

- 类似Hadoop集群，填写slaves文件配置vm2，vm3，vm4为spark-worker节点。
- 执行命令`sbin/start-all.sh`启动Spark集群，通过`jps`命令可以看到master,worker正常启动。

```
master (vm1)
[centos@vm1 ~]$ jps
5714 SecondaryNameNode
28259 Master
5963 ResourceManager
5455 NameNode
```

```
slaves (vm2, vm3, vm4)
[centos@vm2 ~]$ jps
5255 NodeManager
5131 DataNode
19101 Worker
```

- 浏览器输入ip `http://202.120.40.8:30431/`可以看到三个Worker节点正常存活。

2.5 搭建Kafka & Zookeeper集群

由于最新版本的Kafka内置了Zookeeper，所以我们将这两个组件的搭建过程放在一起。

- 从官网下载`kafka_2.12-2.2.0.tgz`安装包，解压。
- 创建`~/kafka-logs`和`~/data/zk`目录分别用来存储kafka log及zookeeper数据。
- 修改`config/server.properties`配置kafka，修改内容如下：

```
broker.id=1       #broker id
num.partitions=3      #分区数量，一般与broker数量保持一致
listeners=PLAINTEXT://localhost:9092     #修改为本机ip
zookeeper.connect=vm2:2181,vm3:2181,vm4:2181      #三台服务zookeeper连接地址
host.name={vm2/vm3/vm4}      #根据自己的ip设置
log.dirs=/home/centos/kafka-logs/       #logs目录
```

- 修改`config/zookeeper.properties`配置Zookeeper，修改内容如下：

```
#数据目录
dataDir=/home/centos/data/zk
#设置连接参数
tickTime=2000
initLimit=10
syncLimit=5
#broker Id的服务地址
server.0=vm2:2888:3888
server.1=vm3:2888:3888
server.2=vm4:2888:3888
```

- 在每个节点的Zookeeper目录下添加myid文件，依次填写broker.id。
- 使用命令`bin/zookeeper-server-start.sh config/zookeeper.properties &`启动每个节点的Zookeeper。
- 使用命令`bin/kafka-server-start.sh config/server.properties &`启动每个节点的Kafka，通过`jps`命令可以看到Kafka,Zookeeper（QuorumPeerMain）正常启动。

```
[centos@vm2 ~]$ jps
4199 QuorumPeerMain
5255 NodeManager
4712 Kafka
5131 DataNode
19101 Worker
```

- 创建系统所需topic：
  `~/kafka_2.12-2.2.0/bin/kafka-topics.sh -create --zookeeper`
  `vm2:2181,vm3:2181,vm4:2181 -replication-factor 3 --partitions 4 --topic dsgroup`

2.6 搭建MySQL Cluster

首先在在官网下载`mysql-cluster-gpl-7.5.15-linux-glibc2.12-x86_64.tar.gz`解压至`/usr/local/mysql-cluster`文件夹。

Management节点

- 执行命令`cp bin/ndb_mgm* /usr/local/bin`将管理节点程序拷贝到PATH目录下。
- 执行命令`chmod +x ndb_mgm*` 添加可执行权限。
- 创建`/var/lib/mysql-cluster/config.ini`配置文件，填写Mysql 集群配置,可以看到我们将vm2,vm3,vm4配置为data节点及SQL节点。

```
[ndb_mgmd]
# Management process options:
HostName=10.0.0.77          # Hostname or IP address of MGM node
NodeId=1
DataDir=/var/lib/mysql-cluster  # Directory for MGM node log files

[ndbd]
# Options for data node "A":
                                # (one [ndbd] section per data node)
HostName=10.0.0.154          # Hostname or IP address
NodeId=2                        # Node ID for this data node
DataDir=/usr/local/mysql/data   # Directory for this data node's data files

[ndbd]
# Options for data node "B":
HostName=10.0.0.137          # Hostname or IP address
NodeId=3                        # Node ID for this data node
DataDir=/usr/local/mysql/data   # Directory for this data node's data files

[ndbd]
# Options for data node "B":
HostName=10.0.0.115          # Hostname or IP address
NodeId=4                        # Node ID for this data node
DataDir=/usr/local/mysql/data   # Directory for this data node's data files

[mysqld]
# SQL node options:
NodeId=5
HostName=10.0.0.154

[mysqld]
# SQL node options:
NodeId=6
HostName=10.0.0.137

[mysqld]
# SQL node options:
NodeId=7
HostName=10.0.0.115
```

SQL节点

- 执行命令`groupadd mysql useradd -g mysql -s /bin/false mysql`创建MySQL用户及用户组。
- 执行命令`chown -R root . && chown -R mysql data && chgrp -R mysql .`添加必要权限。
- 执行命令`cp support-files/mysql.server /etc/rc.d/init.d/ && chmod +x /etc/rc.d/init.d/mysql.server && chkconfig --add mysql.server`添加MySQL服务自启动。
- 编辑`/etc/my.cnf`配置文件，添加SQL配置。

```
mysqld]
# Options for mysqld process:
ndbcluster                      # run NDB storage engine

[mysql_cluster]
# Options for NDB Cluster processes:
ndb-connectstring=vm1  # location of management server
```

- 执行命令`mysqld --initialize sudo systemctl mysql start`启动Mysql服务。

Data节点

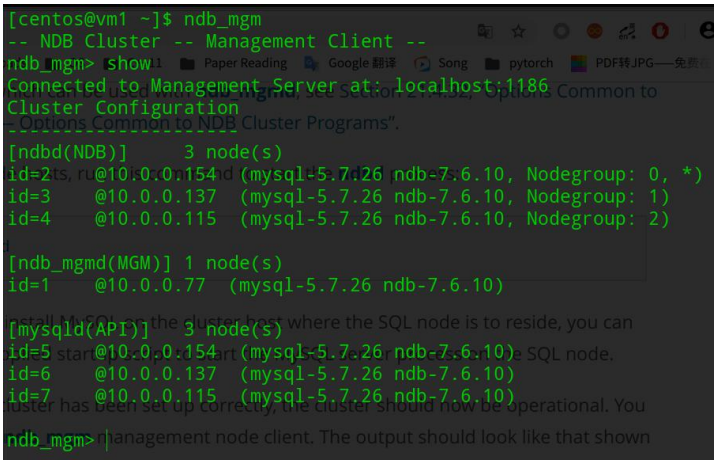
- 执行命令`cp bin/ndbd /usr/local/bin/ndbd && cp bin/ndbmtd /usr/local/bin/ndbmtd`拷贝可执行文件到PATH目录下。
- 执行命令`cd /usr/local/bin && chmod +x ndb*`添加可执行权限。
- 编辑`/etc/my.cnf`配置文件，添加SQL配置。

```
mysqld]
# Options for mysqld process:
ndbcluster                      # run NDB storage engine

[mysql_cluster]
# Options for NDB Cluster processes:
ndb-connectstring=vm1  # location of management server
```

启动MySQL Cluster

- 进入Management节点，执行命令`ndb_mgmd -f /var/lib/mysql-cluster/config.ini`启动管理节点。
- 进入Data节点，执行命令`ndbd`启动数据节点。
- 进入Management节点，执行命令`ndb_mgm`之后`show`可以看到已经成功启动MySQL集群。



2.7 Nginx网关及负载均衡

- 执行命令`sudo yum -y install nginx`安装Nginx
- 编辑`/etc/nginx/nginx.conf`文件配置Web服务节点

```
upstream serviceservers{
        server 10.0.0.154:8080;
        server 10.0.0.137:8080;
        server 10.0.0.115:8080;
}

server {
    listen        30438;
    server_name  localhost;
    root         /usr/share/nginx/html;

    # Load configuration files for the default server block.
    include /etc/nginx/default.d/*.conf;

    location / {
        proxy_pass http://serviceservers;
    }

    error_page 404 /404.html;
        location = /40x.html {
    }

    error_page 500 502 503 504 /50x.html;
        location = /50x.html {
    }
}
```

3. 系统架构设计

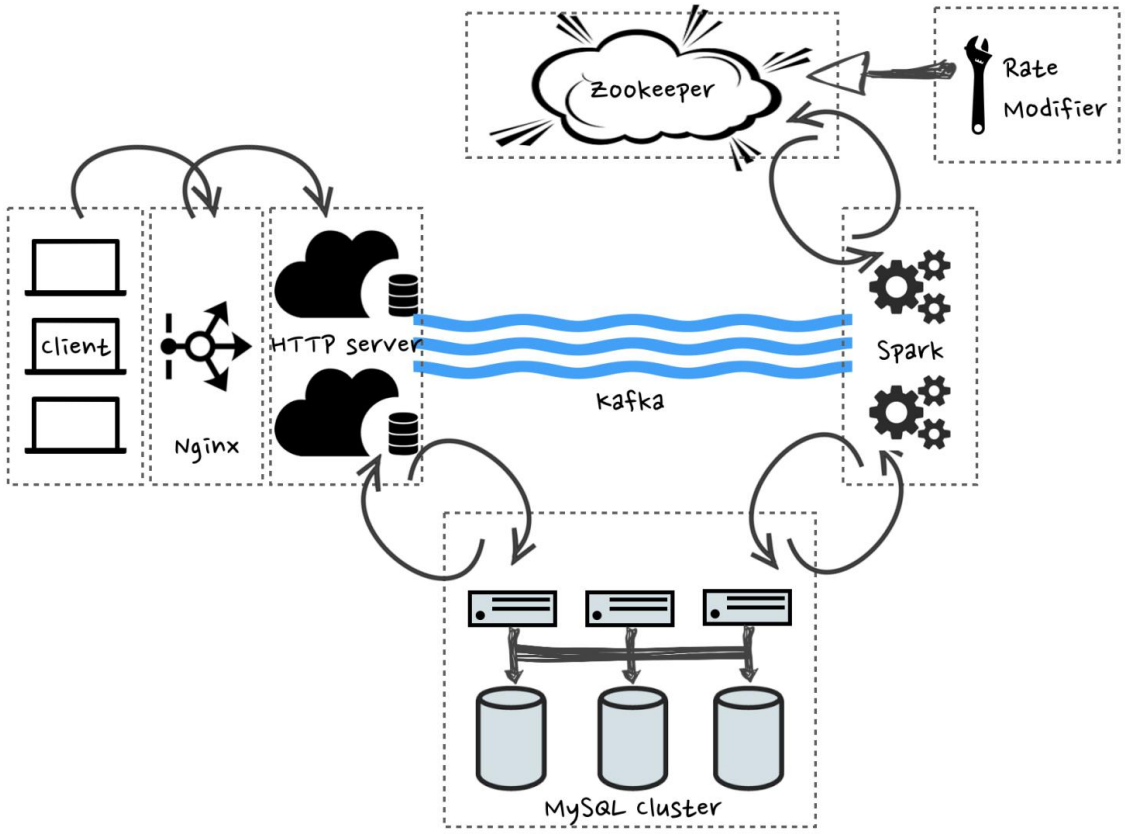
3.1 系统过程描述 & 架构图

整个系统的执行流程如下：

1. Client发送HTTP请求提交订单。
2. Nginx对Client发送的请求进行负载调节，发送给HTTP server。
3. 对于查询订单请求，HTTP server直接向MySQL Cluster查询订单结果。
4. 对于下单请求，HTTP server将请求生成订单并传递给Kafka。
5. Spark streaming从Kafka中不断读出订单消息。
6. Spark streaming向Zookeeper请求汇率数据。
7. Spark执行订单处理逻辑，并将结果写入MySQL Cluster。

在整个系统运行的同时，Rate Modifier不断请求修改Zookeeper中的汇率数据。

系统架构图如下：



3.2 HTTP Receiver

HTTP server主要负责接受来自用户的订单创建请求，并将这些订单信息添加至Kafka的缓存中，以待Spark从中读取并处理。HTTP server部署在多个节点以提供高效的服务，开放或使用和前端，Kafka以及MySQL的接口。我们使用了Nginx来实现请求的调度。

对于创建订单的要求，HTTP server对前端暴露了“/create_order”的HTTP请求接口，接收一个JSON格式的字符串。JSON中包含user_id，initiator，time，item的信息。当HTTP server收到了订单的信息后，会以UUID的形式创建一个唯一的order_id。将以上所有信息打包为一个JSON对象后，调用Kafka的producer接口，并将信息添加至Kafka指定的topic的缓存中。添加成功后，将返回order_id给前端，作为订单提交成功的返回信息。

对于查询订单的请求，HTTP server通过连接MySQL集群，并从中读取数据返回给前端。用户需要指定一个order_id，HTTP server通过order_id在数据库中查询后，返回给前端用户。

在连接Kafka的过程中，我们发现如果对每一个请求动态创建Producer，则会在创建和销毁的过程中消耗大量的时间；但如果只维持一个静态的Producer，则又无法达到预期的并发性。因此，我们维持了一个阻塞队列。HTTP server会尝试从队列中获取已经建立连接的Producer，如果一段时间内无法获得，则创建一个新的连接。当HTTP server向Kafka缓存中发送数据完毕后，会将使用到的Producer类重新添加至队列中。

3.3 用户接口

为了产生request，我们设置了两种途径：适用于用户使用的前端和模拟用户发送订单请求的python脚本。

前端提供了两种功能：创建订单、查询订单。

在创建订单中，用户需要输入自己的用户id，已经订单适用的货币，并添加想要购买的商品id和数量，随后点击“提交订单”即可。前端会发送HTTP POST请求交给后端，后端读取body中的参数并交给kafka等待处理。

在查询订单中，用户需要输入订单id，并点击“查询”。前端会发送HTTP GET请求交给后端，后端读取body中的参数，从数据库中读取订单内容并返回。

python脚本会随机生成某个订单，并发送至HTTP server。每次运行脚本会随机发送1~4个订单请求。

在发送请求后，Nginx会均衡各个HTTP server的负载。

3.4 定时修改汇率

我们实现一个java程序`ExchangeSimulator.java`去控制更改汇率。该程序新建一个zkclient来向zookeeper server请求更改汇率。

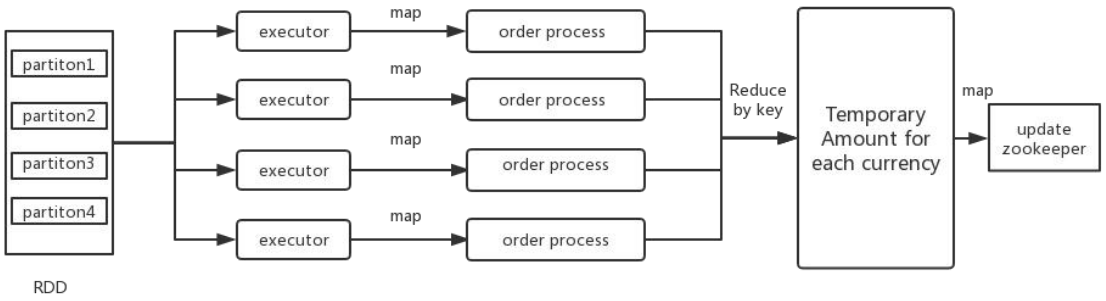
程序每分钟修改1次汇率，每次随机涨跌不超过2个百分点，同时控制汇率在初始设定的范围内波动。

并发控制成为修改汇率的一个问题。我们通过在修改汇率前向zookeeper server索要汇率对应的锁来解决问题。

3.5 Spark处理订单

Spark处理订单通过Spark streaming技术实现。Spark会从kafka中定时获取订单，通过Map-Reduce的方式进行处理，同时将结果写入MySQL cluster中。

在数据处理过程中，Spark需要从Zookeeper中读取汇率。为了进行并发控制，也需要对其进行加锁处理。



上图为Spark streaming处理订单的流程的大致流程。Spark从kafka采用direct的方式读取订单数据，给每个partition分配一个executor处理订单。处理订单包括解析订单内容，从zookeeper中获取锁，读取汇率信息，判断该订单是否能够完成，将result和修改后的库存信息重新写会数据库。将改订单的交

易额和货币类型以<key,value>对的形式存下来。之后使用reduce的方法统计改rdd中每种货币的交易额，然后修改zookeeper中保存的总交易额。

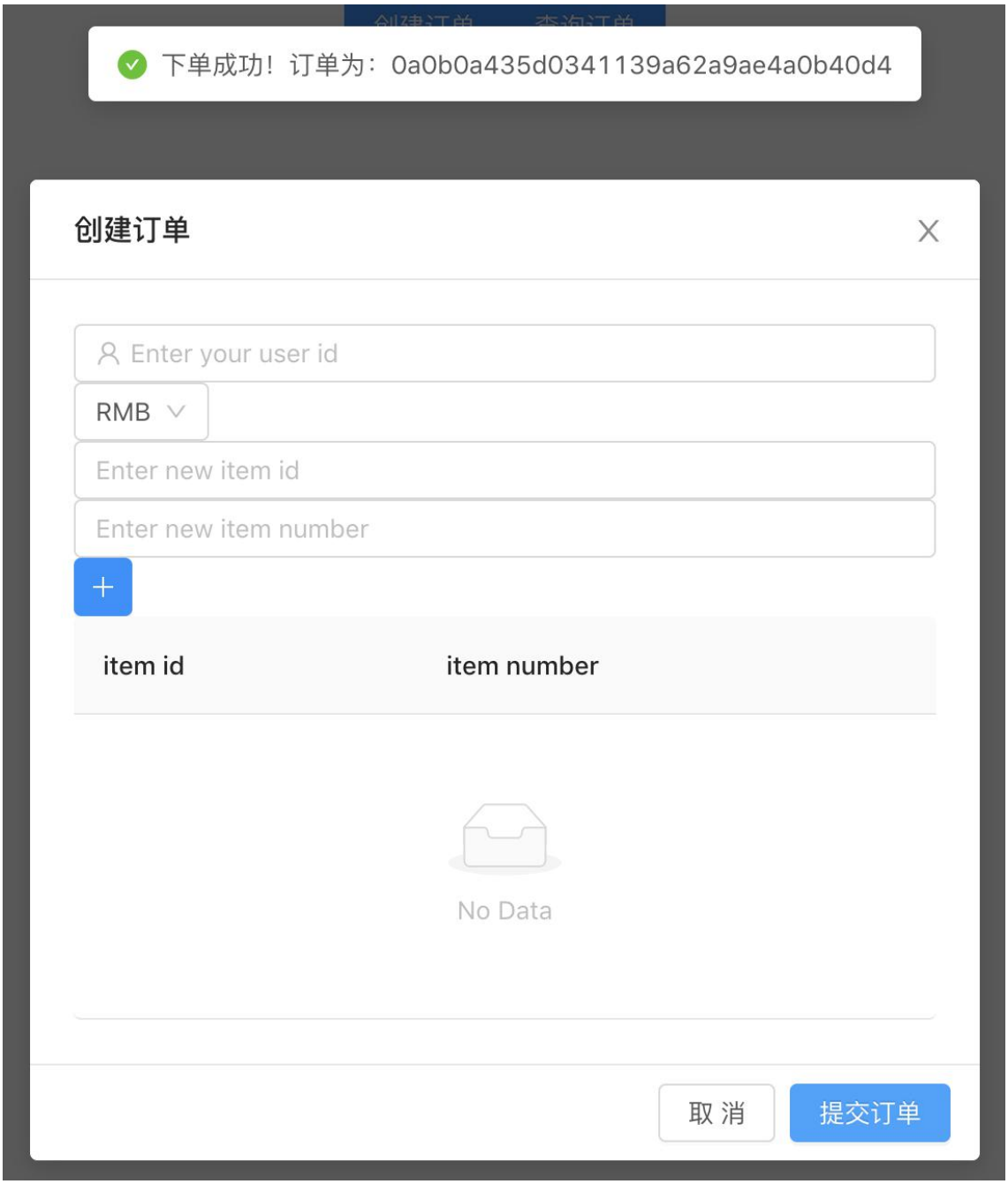
3.6 读取总交易额

获取Amount的读锁，然后从zookeeper中读取相应货币的总成交额。该操作为系统管理员执行所以不需要将该接口返回给用户。操作员可以通过给定的指令读取当前的总交易额。

3.7 程序截图

从前端发起创建订单请求，程序截图如下：

编辑订单


下单成功！订单为：0a0b0a435d0341139a62a9ae4a0b40d4
创建订单
Enter your user id
RMB
Enter new item id
Enter new item number
item id
item number
No Data
取 消
提交订单

下单成功

4. 项目中遇到的问题

4.1 配置环境

- `start.all hadoop`失败。

  ERROR:

```
ERROR: Attempting to operate on hdfs namenode as root
ERROR: but there is no HDFS_NAMENODE_USER defined. Aborting operation.
Starting datanodes
ERROR: Attempting to operate on hdfs datanode as root
ERROR: but there is no HDFS_DATANODE_USER defined. Aborting operation.
Starting secondary namenodes [vm1.novalocal]
ERROR: Attempting to operate on hdfs secondarynamenode as root
ERROR: but there is no HDFS_SECONDARYNAMENODE_USER defined. Aborting
operation.
WARNING: HADOOP_PREFIX has been replaced by HADOOP_HOME. Using value of
HADOOP_PREFIX.
WARNING: HADOOP_PREFIX has been replaced by HADOOP_HOME. Using value of
HADOOP_PREFIX.
Starting resourcemanager
ERROR: Attempting to operate on yarn resourcemanager as root
ERROR: but there is no YARN_RESOURCEMANAGER_USER defined. Aborting operation.
Starting nodemanagers
ERROR: Attempting to operate on yarn nodemanager as root
ERROR: but there is no YARN_NODEMANAGER_USER defined. Aborting operation.
```

- scp及ssh频繁出现permission denied（已设置免密登录）。

  解决：把用户改为centos 并执行了 `chmod -R 777 /usr/local/hadoop`。

- 解压Spark安装包出现失败。

  ERROR:

```
gzip: stdin: not in gzip format
tar: Child returned status 1
tar: Error is not recoverable: exiting now
```

  解决：这个压缩文件实际上是一个html，可使用`file`命令查看。

- Spark启动后jps可以看到worker启动，但是web界面没有worker信息

  ERROR:

解决：参考https://blog.csdn.net/qq1187239259/article/details/79489800

- Nginx服务启动失败

  ERROR：

```
7月 09 02:09:56 vm1.novalocal systemd[1]: Starting The nginx HTTP and reverse
proxy server...
7月 09 02:09:56 vm1.novalocal nginx[14445]: nginx: the configuration file
/etc/nginx/nginx.conf syntax is ok
7月 09 02:09:56 vm1.novalocal nginx[14445]: nginx: [emerg] bind() to
0.0.0.0:30438 failed (13: Permission denied)
7月 09 02:09:56 vm1.novalocal nginx[14445]: nginx: configuration file
/etc/nginx/nginx.conf test failed
7月 09 02:09:56 vm1.novalocal systemd[1]: nginx.service: control process
exited, code=exited status=1
7月 09 02:09:56 vm1.novalocal systemd[1]: Failed to start The nginx HTTP and
reverse proxy server.
7月 09 02:09:56 vm1.novalocal systemd[1]: Unit nginx.service entered failed
state.
7月 09 02:09:56 vm1.novalocal systemd[1]: nginx.service failed.
```

  原因：centos的selinux导致

  解决：

  If you use a port bigger than 1024 and root privilege, then still get this problem, that's may cause by SELinux:

  Check this port, say 8024, in segange port

```
sudo semanage port -l | grep http_port_t
```

  If the port list no 8024, then add it into segange port

```
sudo semanage port -a -t http_port_t  -p tcp 8024
```

4.2 系统性能优化

4.2.1 性能挑战

1.最初性能描述

a) 吞吐量：在最初的版本下，我们的吞吐量大约只在10 orders/s左右。

b) 延迟：Spark处理一个订单的时间需要300-400ms左右。

2.探究性能瓶颈

a) 吞吐量

吞吐量的限制来自于其中某一部分达到了性能瓶颈，导致了整个系统运作的吞吐量无法提升。我们对于系统中的各个部分进行了分析：

MySQL Cluster

由于我们使用了3个SQL服务器，并且启用了负载均衡，所以在只读请求上不会成为吞吐量的瓶颈。对于写请求，我们使用了MySQL Cluster提供的benchmark工具Mysqlslap来进行模拟，吞吐量达每秒1000个transaction以上（延迟~=1ms/transaction），虽然我们只是进行简单的update操作，但是这个数据也和总吞吐量10 orders/s相差太多。我们认为MySQL Cluster不应该成为整个系统吞吐量的瓶颈所在。

HTTP server

由于我们部署了3个HTTP server，同时使用了Nginx进行负载均衡，而且网络的bandwidth应该远远不止10 request/s，所以我们认为HTTP server也不应是系统的吞吐量瓶颈。

Spark streaming

最终我们发现，在spark-submit提交任务后，其性能和单机并没有差太多，这是违反直觉的。为此我们查看了log，发现只有1个消费者的记录，而我们默认有4个partition。经过对log的分析我们发现这是由于我们在生产消息的时候制定了单一的key导致的，所有的消息都被分到了同一个partition里。

b) 延迟

我们对单个消息的处理过程时间做了细粒度的划分，得到整个Spark处理的延迟大约来自于以下几个部分：

	建立session factory	Zookeeper client连接	索要分布式锁
latency(ms)	200-300	20-30	20-30

4.2.2 性能优化手段

通过对于系统的性能分析，我们主要通过以下几种方式优化系统性能：

优化latency的三个重要组成部分

- 使用单例的思想解决建立session factory消耗时间过长的问题。
  我们利用单例思想，建立了可重用的唯一的连接池，这样就可以将session factory的初始化时间降为0，大大减少了latency。
- 使用mapPartition计算接口降低Zookeeper client连接延迟
  我们发现之前编写的代码只简单地使用了map方法，在处理每个订单的时候都要与Zookeeper建立连接。之后我们又查阅了Spark文档，发现可以使用mapPartition这个接口，这个接口可以对每个RDD的本地partition建立唯一的Zookeeper client连接，这样就将Zookeeper连接的时间分摊，使得延迟基本可以忽略。
- 在索要分布式锁的部分，我们发现Zookeeper中的forcesync属性会使得每次Zookeeper server处理锁都要进行持久化操作，十分耗时。于是我们考虑能否关闭这一属性以提升性能。**接下来我们将重点讨论我们在forcesync方面的一些工作与思考。**

forcesync对性能的影响分析

由于Zookeeper的forcesync属性会在操作commit时默认将其值持久化到硬盘，导致了非常高的延迟和阻塞，成为了整个系统的一个bottleneck。尤其是在目前的场景下，Zookeeper会对每个拿锁、放锁操作都进行持久化，大大降低了拿锁的效率，从而降低了整个系统的性能。

关闭forcesync对系统造成的影响

持久化是分布式系统容错的一个重要部分，简单关闭forcesync，将会对整个系统的容错造成影

响。我们对关闭forcesync后Zookeeper的容错状态进行了分析，并给出了可能的解决方案。

Zookeeper中需要持久化的部分

我们首先对Zookeeper在系统中负责的功能进行了分析：

1.存储汇率 2.分发分布式锁。

在容错场景下，汇率和对分布式锁的操作都是需要持久化的，否则可能会出现汇率不一致、同时将锁给了两个zkclient的情况。为了保证程序的正确性，必须要保证以下两个属性：

- **属性1**：可以恢复出最新的汇率值。
- **属性2**：保证不会有两个zkclient，它们同时拿到了同1把锁。

在当前的实现下，为了能够容忍Zookeeper的crash，需要将汇率值和分布式锁都进行持久化。

可能的解决方案

对于属性1，由于汇率每分钟修改1次，对汇率的持久化并不会影响太多性能，我们通过手动将汇率持久化到硬盘来保证属性1。

对于属性2，我们在查询资料后，提出了以下两种可能的解决方案，但都需要对Zookeeper的源代码进行修改：

利用内存中数据来进行recovery

Zookeeper使用ZAB这一协议来保证数据的一致性和recovery，而disk durable是ZAB保证没有data loss和availability的重要部分（事实上，Paxos，Viewstamped Replication, Raft等一致性协议均是如此）。

而实际上，为了提升性能，许多现实应用场景都会关闭forcesync，利用内存replication来进行recovery。这么做显然会大幅提升性能，因为replication的操作变为了in-memory的操作，但是带来了潜在的data loss和unavailability的问题。

为了解决这个问题，我们尝试去搜索了相关资料和论文。我们发现在论文*Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems(OSDI’ 18)*中，针对这个问题提出了解决办法。

此论文提出了SAUCR（situation-aware updates and crash recovery）系统，该系统基于以下假设：

*1.分布式系统中的failure很少会**同时**发生，即便是**correlated**的failure，在individual failure出现之间也会存在一个时间gap，大致在几十毫秒到几秒之间。*

2.SAUCR可以在正常运行的内存操作下解决independent failure和non-simultaneously correlated failure下的data loss问题。这两种failure的共同点是在节点fail之间存在time gap。SAUCR会即时感知可能的failure并利用这样的gap进行数据持久化，以解决data loss的问题。

*3.SAUCR无法解决simultaneously correlated failure的场景，但是论文声称这样的场景是**extremely rare**的。*

我们认为SAUCR为在内存操作的场景下避免data loss提供了一种解决办法，可以**实现在没有forcesync的情况下，依旧保证了没有data loss的出现**（在绝大多数情况下）。

使用lease概念实现分布式锁

上一种解决方法从整个一致性协议角度，将正常运作情况下的持久化磁盘操作优化为内存操作。

而对于修改汇率，由于其频率不高，我们是可以接受对汇率的持久化操作的，目前我们只需要解决如何确保属性2这一问题即可。为了确保属性2，我们只需满足在存在failure的情况下，有且仅有1个zkclient可以拿到锁。

我们同样查询了关于分布式锁实现的资料，发现一篇经典论文提供了***lease***的概念：*Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency。*我们认为这可能成为解决此问题的又一方式。

在原先没有forcesync的情况下，Zookeeper服务器fail可能会导致重启之后，服务器无法获得是否有人已经拿过锁的信息，错误地将锁又传递给另一个zkclient。

而在使用lease lock的场景下，重启的Zookeeper服务器只需等待一个lease time后，即可放心

地将锁给任意一个zkclient（因为此时，在宕机前服务器所给出的lease lock必然都已经过期了）。

使用lease lock的执行流如下：

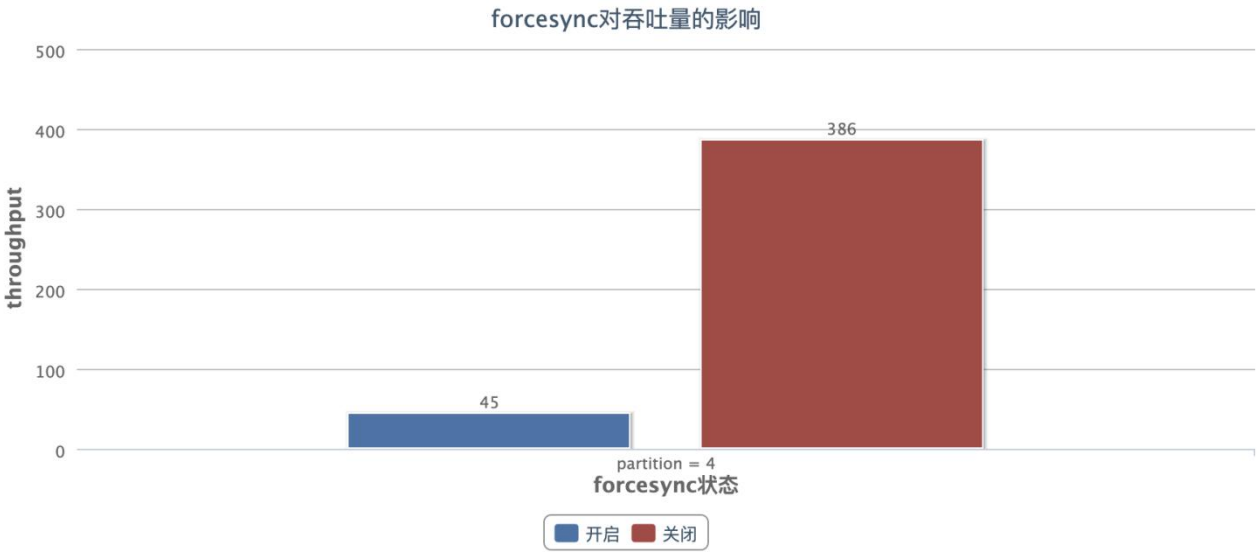
1. zkclient向Zookeeper server索要lease lock。
2. Zookeeper server返回一把lease lock，同时携带lease time（租赁有效时间）。
3. zkclient在执行完操作后，将锁返还Zookeeper server。
4. 倘若zkclient发生宕机，无法正确返还lease lock，Zookeeper server在lease time后可将锁在授权给别的zkclient。
5. 倘若Zookeeper server发生宕机，在重启后，等待lease time后即可将锁授权给zkclient。

通过lease lock的时效机制，我们可以轻松保证Zookeeper服务器不会同时将锁给不同的zkclient，由此保证了属性2。

然而以上两种方法都需要修改Zookeeper server。方法1需要修改Zookeeper server基于的一致性协议，修改工程量较大。方法2需要修改Zookeeper分布式锁的实现。

我们尝试进行了lease lock的实现，最后由于Zookeeper server代码过于复杂而中途放弃 了。

最终我们发现，关闭forcesync在降低latency的同时也极大地提升了吞吐量，由于关闭forcesync大大提升了拿锁　的效率，所以这是显而易见的。我们测试了关闭前后吞吐量的大致提升：



解决吞吐量过低的问题

如4.2.1所述，当前吞吐量的主要瓶颈在于Spark内部的并发程度不够，我们主要通过以下几种方式提升Spark内部的并行度：

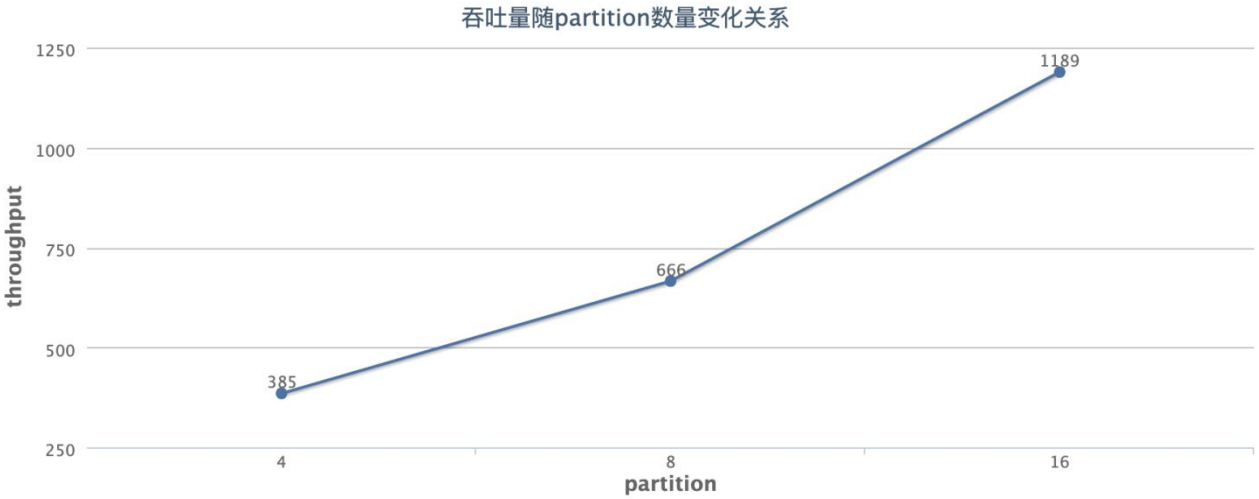
随机生成消息key

生产消息时使用随机key而非单一key，使得消息可以尽可能平均地分配到各个partition上。

增加partition数目

在完成上一步后，我们发现系统的资源仍未充分利用完全，说明4个partition的并发还未达到系统的瓶颈，我们还可以继续增加partition的数目增加并发度，从而提升系统性能。需要注意的是，由于系统的资源有限，过多增加partition会导致严重的资源竞争问题，反而会降低系统的整体性能。经过调参测试后，我们将partition数目定为了10。

改变partition，吞吐量的变化大致如下：



将消费方式由receiver改为direct

在查阅资料后，我们发现Spark streaming有两种消费方式：receive和direct。由于在我们设计的系统中，并不会出现1个group内有多个worker的情况，所以改为direct方式不会增加维offset带来的overhead。而direct方式有以下特性：

a.zero-copy

b.batch处理

c.减少资源使用

以上特性均能够带来吞吐量的提升。

针对batch特性融入Map-Reduce的思想处理totalAmount

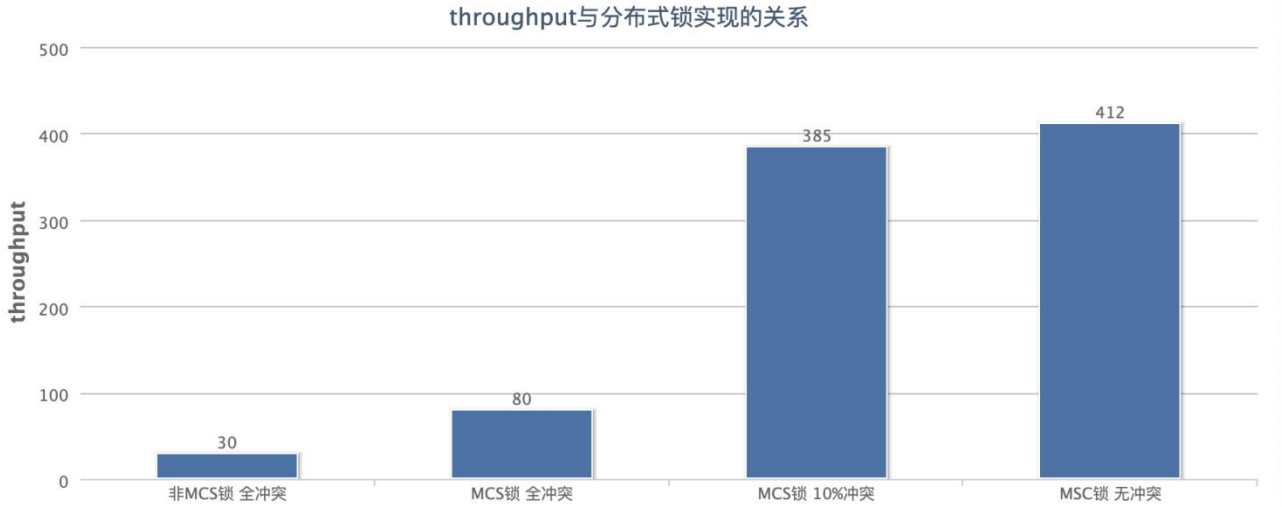
在查询文档的过程中，我们发现Spark支持reduce操作，在原本的实现中，我们在每次处理订单之后都要更改totalAmount。而使用reduce操作，我们可以对RDD partition中的所有订单做batch处理，计算每个partition的totalAmount之后，再Reduce回真正的 totalAmount。

优化分布式锁设计

在锁的设计上，由于非可扩展锁（如传统的spinlock）会导致多个锁请求者频繁读取同1个变量，其性能在跨核场景下已经很不乐观，在分布式场景下更是需要通过网络连接来获得最新的锁的状态，我们估测这样的性能应该是不可接受的。于是我们采用了可扩展锁MCS来实现分布式锁。

同时，我们分析了汇率的使用场景为1个写者、多个读者，同时写的频率并不是非常高（1 time /minute），所以我们将MCS实现为读写锁的形式，可以大大减少Spark在处理订单时读取汇率的锁冲突，提升性能。

我们对MCS锁和非可扩展锁进行了性能对比测试，测试结果如下：



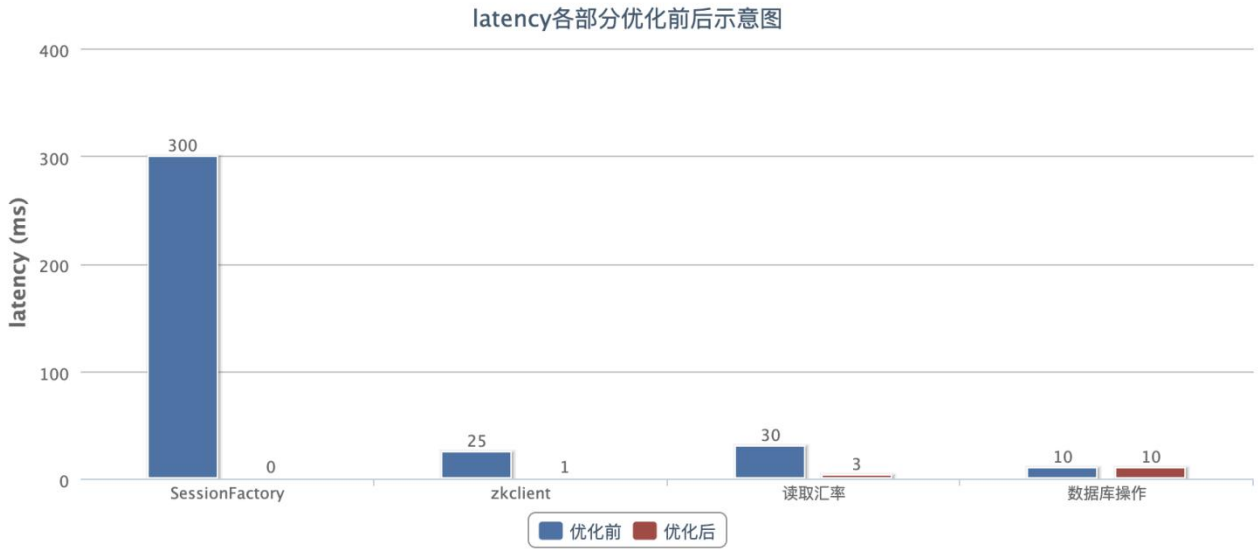
4.2.3 前后性能对比

latency性能对比

在进行上述的优化后，latency部分的性能提升大致如下：

	建立session factory	Zookeeper client连接	索要分布式锁	总计
before (ms)	200-300	20-30	20-30	300-400

after (ms)	~0	~0	~0	~10

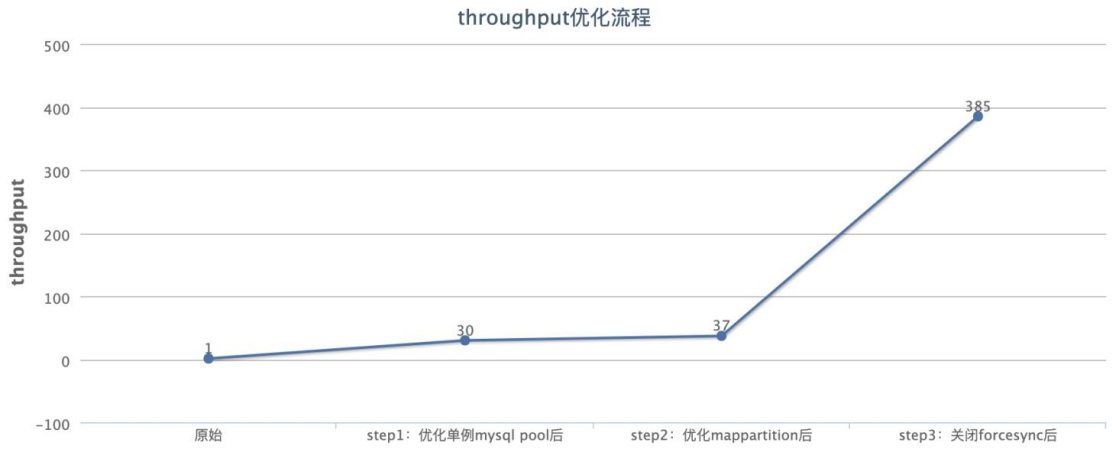


throughput性能对比

在进行上述的优化后，throughput部分的性能提升大致如下：

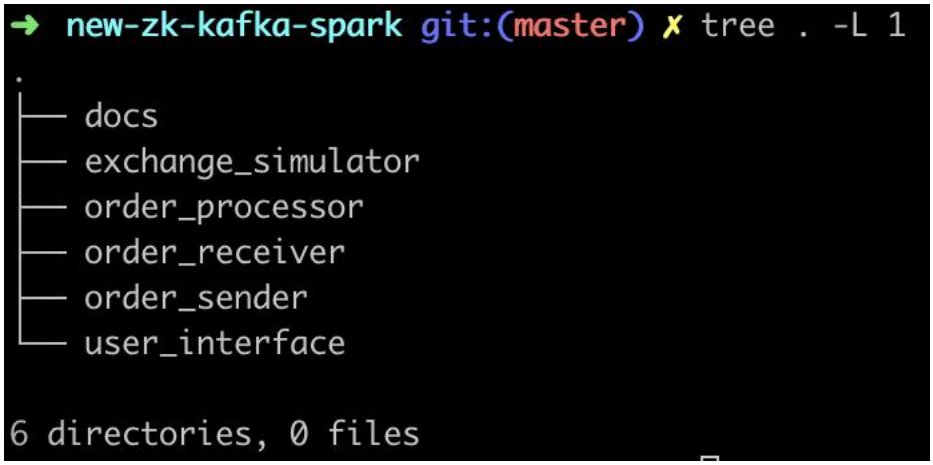
	throughput
before (orders/s)	~10
after (orders/s)	785

其中，除增加partition外，我们大致有3个步骤逐步提升throughput，其提升曲线如下：



5. 项目结构

整个项目的项目结构如下:



- `./docs`目录包含该项目的相关文档
- `./exchange_simulator`目录包含负责修改汇率的java程序
- `/order_processor`目录包含处理订单的Spark程序
- `/order_receiver`目录包含接收订单的server程序
- `/order_sender`目录包含发送订单请求的脚本程序
- `/user_interface`目录包含用于和用户交互的前端程序

6. 分工

冯二虎: zookeeper、spark、kafka代码，订单处理，系统性能测试（26%）

姚子航: 系统环境搭建, http server后端, 汇率修改程序，发送订单脚本（26%）

丁丁: 文档撰写，ppt制作，前端，系统性能优化，订单总量查询（26%）

蔡一凡: 文档撰写，ppt制作，http server后端，kafka producer（22%）

7. 参考资料

[1] Hadoop cluster.https://hadoop.apache.org/

[2] Kafka: https://kafka.apache.org/

[3] Zookeeper: http://zookeeper.apache.org/

[4] MySQL Cluster: https://dev.mysql.com/

[5] Spark: https://spark.apache.org/

[6] Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems(OSDI '18): https://www.usenix.org/system/files/osdi18-alagappan.pdf

[7] Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency: https://web.stanford.edu/class/cs240/readings/89-leases.pdf