基于分库分表中间件的高并发秒杀系统设计与实现

汇报人：黄勖

时间：2024.06.06

小组成员

- 黄勖:
DB-router中间件编写，工程基本框架搭建，部分测试

- 胡翼翔:
Sharding-JDBC中间件连接，文档编写

- 张靖源:
Jmeter测试，PPT制作

利用三次实验完成，之前的小实验均完成。

项目简介

随着电子商务的迅猛发展，促销活动如“秒杀”已成为吸引客户和增加销售额的重要手段。秒杀活动特点是商品数量有限、折扣深，且购买时间限制严格，这常常在极短的时间内引发大量用户的并发访问。这种高并发场景对电商平台的数据处理能力和系统响应速度提出了极高的要求。

我们在本项目中自己尝试编写中间件，并通过实施中间件技术——Sharding-JDBC，分别来实现数据的有效分片和读写分离，提高数据库处理能力，确保系统在高并发条件下的稳定性和高效性。经过对比，Sharding-JDBC作为一个轻量级的中间件集成方案，不仅可以减少系统的复杂性，还能提高数据操作的灵活性和效率。

项目目标

高并发处理能力

通过分库分表中间件，系统能够在多数据库实例间分散请求压力，提高数据操作的并行度，从而有效应对秒杀场景下的高并发请求

读写分离

通过配置Sharding-JDBC的读写分离策略，系统可以将查询操作和事务性写操作分别路由到不同的数据库节点，进一步优化性能

数据一致性保证

尽管数据分布在多个数据库实例中，分库分表中间件确保跨分片的数据操作维持一致性，对外提供一致的数据视图

容错与可扩展性

系统设计考虑了容错机制，如数据库实例宕机的情况下自动切换到备用实例。同时，支持动态扩展数据库实例以应对业务增长

项目背景和项目架构

分库分表中间件项目旨在解决传统单体数据库在高并发、大数据量场景下的性能瓶颈和可扩展性问题。通过将数据分散到多个数据库实例和表中，该中间件能够显著提高系统的读写性能和扩展能力。本项目主要实现了动态数据源切换和分库分表的功能，通过自定义的哈希路由策略确保数据的均匀分布。
本项目借鉴了 Sharding-jdbc 的分库分表思想，结合了 Spring Boot 的配置和管理机制，提供了灵活的分库分表解决方案。

项目基于Spring Boot框架，采用动态数据源和MyBatis插件实现数据的分库分表功能。核心组件包括动态数据源切换、路由策略实现和SQL拦截器。

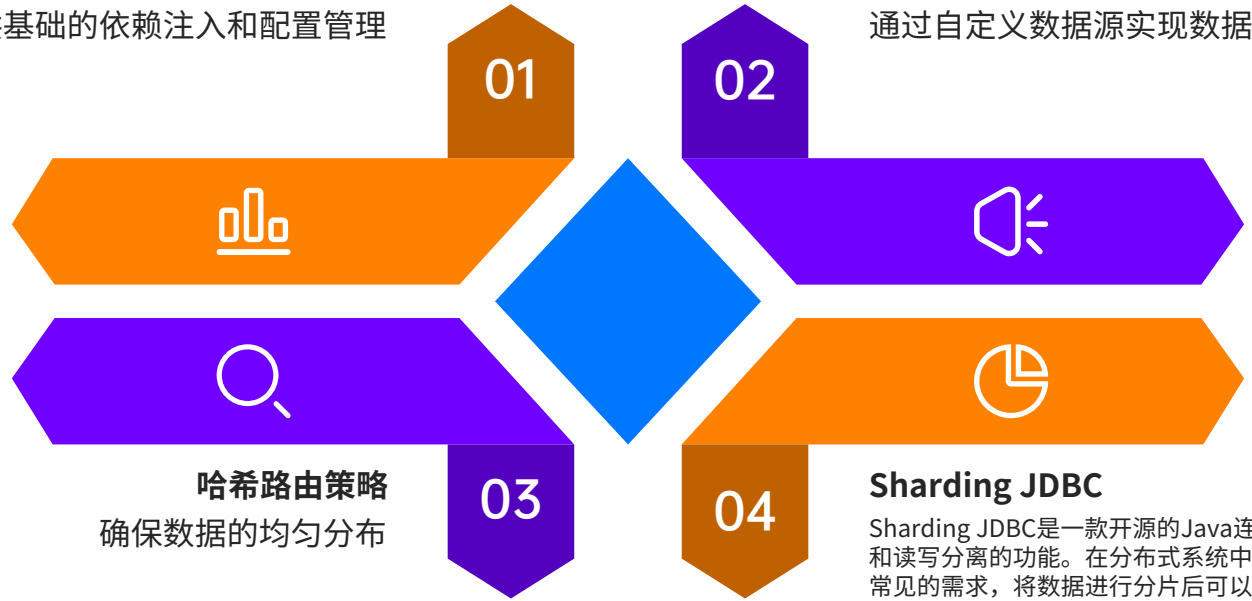
核心技术

Spring Boot

提供基础的依赖注入和配置管理

动态数据源切换

通过自定义数据源实现数据源的动态切换



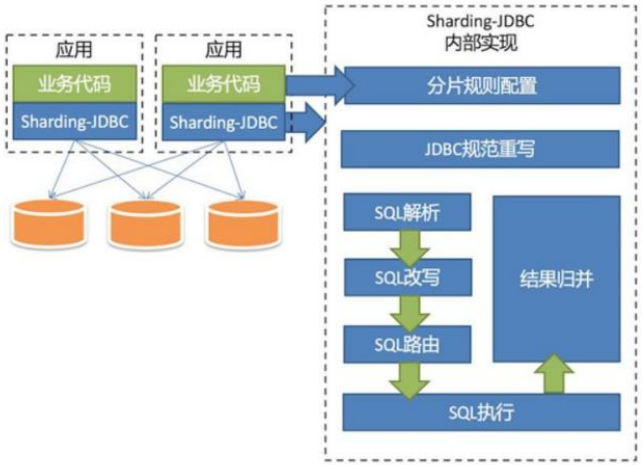
哈希路由策略

确保数据的均匀分布

Sharding JDBC

Sharding JDBC是一款开源的Java连接池库，提供了分片和读写分离的功能。在分布式系统中，数据的分片是非常常见的需求，将数据进行分片后可以有效的提高系统性能和可拓展性。Sharding JDBC 以 JDBC接口为规范，可以被所有基于JDBC的应用程序所使用。

Sharding JDBC的整体架构和主要功能



1.分片以及读写分离

支持垂直拆分和水平拆分两种分片方式，同时支持读写分离，可以将读操作和写操作分别指向不同的数据库实例，有效的分担数据库的负载，提高了系统的性能和可靠性。

2.基于SQL的路由

Sharding JDBC 支持基于SQL语句的路由，通过解析SQL语句中的分片键来将SQL语句路由到相应的分片数据库实例进行执行。这种路由方式相对于基于数据源的路由更加灵活和高效，可以支持更多的分片场景。

3.事务和连接管理

Sharding JDBC支持分布式事务，通过在应用程序中使用XA接口来实现。同时，它也提供了连接管理功能，可以对连接进行池化和复用，降低了系统开销。

4.简单易用的配置

Sharding JDBC 提供了简单易用的配置方式，用户只需要在配置文件中指定分片规则和读写分离规则即可

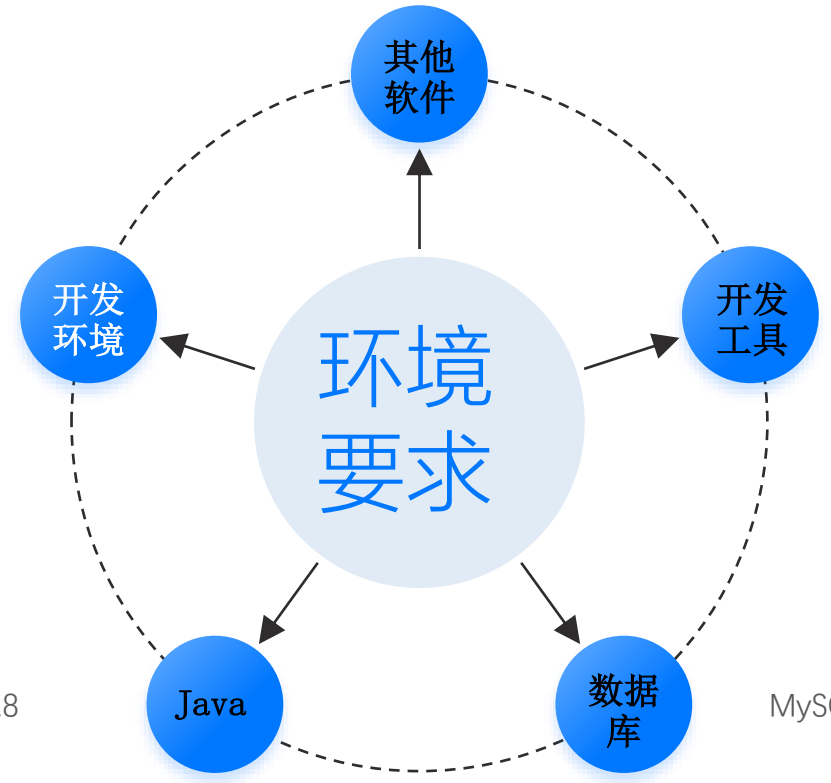
5.负载均衡和故障检测

Sharding JDBC 支持负载均衡和故障检测功能，可以自动检测数据库节点的情况，保证了系统的可用性和容错性

环境要求

Git（用于代码管理和版本控制）

处理器
3.19 GHz AMD Ryzen 7 5800H
内存
15.4GB 3200 MHz DDR4
版本
Windows10 9045.4412

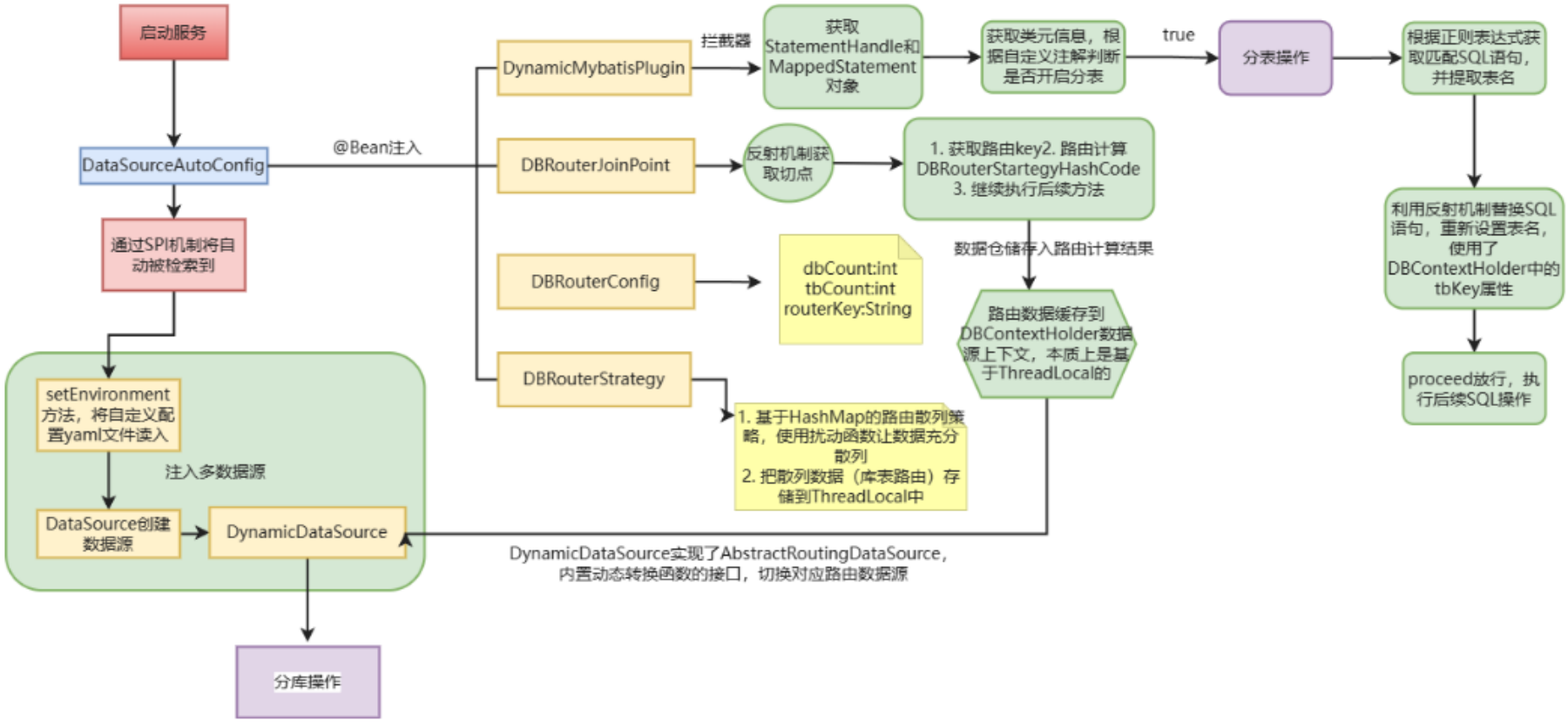


IntelliJ IDEA 或 Eclipse（带Spring Boot插件）

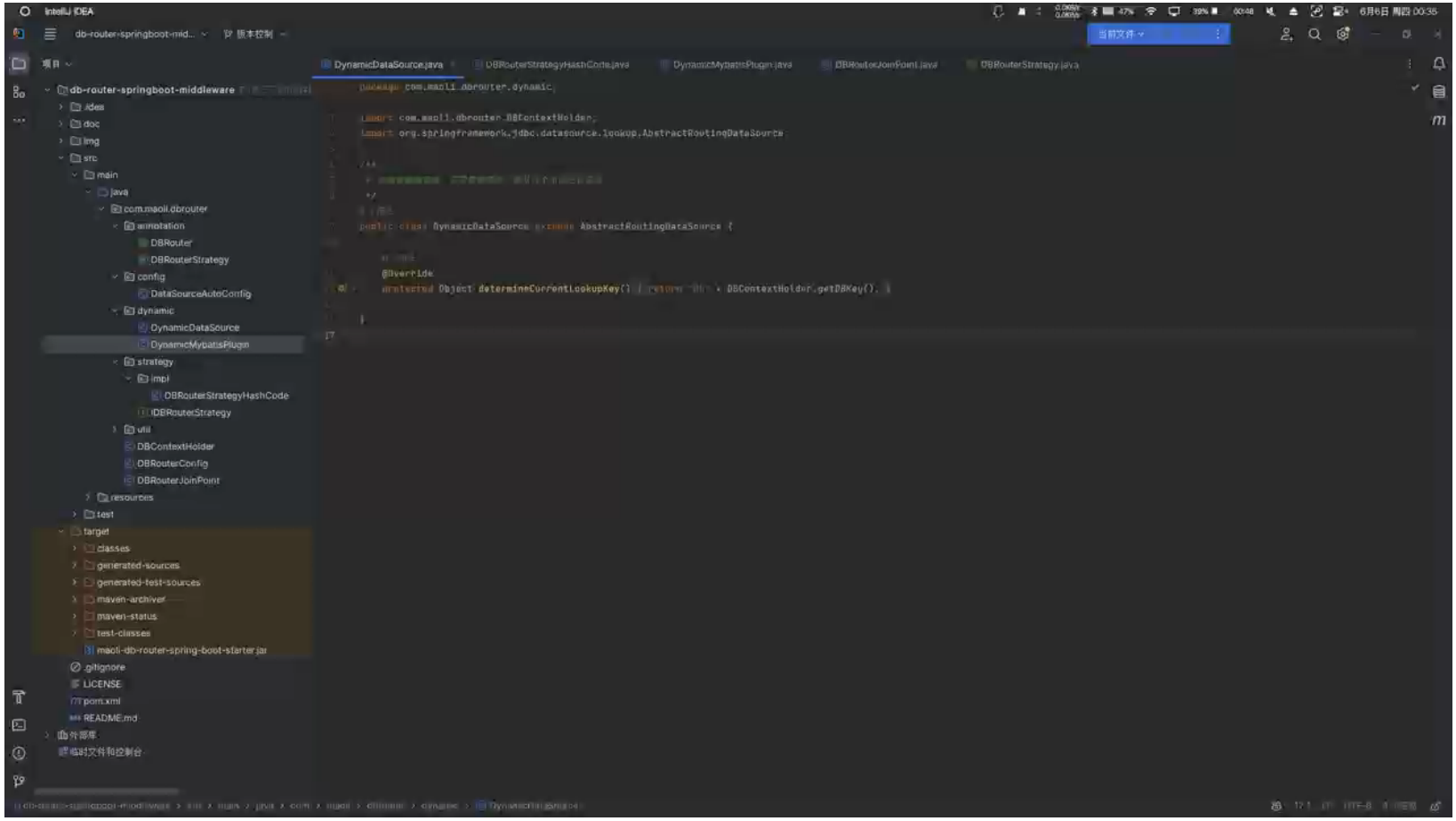
Java JDK 1.8

MySQL 8.0 或更高版本

分库分表功能


启动服务
DataSourceAutoConfig
@Bean注入
通过SPI机制将自动被检索到
setEnvironment方法，将自定义配置yaml文件读入
注入多数据源
DataSource创建数据源
DynamicDataSource
分库操作
DynamicMybatisPlugin
拦截器
获取StatementHandle和MappedStatement对象
获取类元信息，根据自定义注解判断是否开启分表
true
分表操作
根据正则表达式获取匹配SQL语句，并提取表名
利用反射机制替换SQL语句，重新设置表名，使用了DBContextHolder中的tbKey属性
proceed放行，执行后续SQL操作
DBRouterJoinPoint
反射机制获取切点
1. 获取路由key2. 路由计算DBRouterStartegyHashCode
3. 继续执行后续方法
数据仓储存入路由计算结果
DBRouterConfig
dbCount:int
tbCount:int
routerKey:String
路由数据缓存到DBContextHolder数据源上下文，本质上是基于ThreadLocal的
DBRouterStrategy
1. 基于HashMap的路由散列策略，使用扰动函数让数据充分散列
2. 把散列数据（库表路由）存储到ThreadLocal中
DynamicDataSource实现了AbstractRoutingDataSource，
内置动态转换函数的接口，切换对应路由数据源

代码讲解


IntelliJ IDEA
db-router-springboot-middleware
DynamicDataSource.java
DBRouterStrategyHashCode.java
DynamicMybatisPlugin.java
DBRouterJoinPoint.java
DBRouterStrategy.java
db-router-springboot-middleware
.idea
doc
img
src
main
java
com.maoli.dbrouter
annotation
DBRouter
DBRouterStrategy
config
DataSourceAutoConfig
dynamic
DynamicDataSource
DynamicMybatisPlugin
strategy
impl
DBRouterStrategyHashCode
IDBRouterStrategy
util
DBContextHolder
DBRouterConfig
DBRouterJoinPoint
resources
test
target
classes
generated-sources
generated-test-sources
maven-archiver
maven-status
test-classes
maoli-db-router-spring-boot-starter.jar
.gitignore
LICENSE
pom.xml
README.md
外部库
临时文件和控制台
package com.maoli.dbrouter.dynamic;
import com.maoli.dbrouter.DBContextHolder;
import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;
public class DynamicDataSource extends AbstractRoutingDataSource {
@Override
protected Object determineCurrentLookupKey() { return DBContextHolder.getDBKey(); }
}

安装步骤

将项目打成jar包放入ma

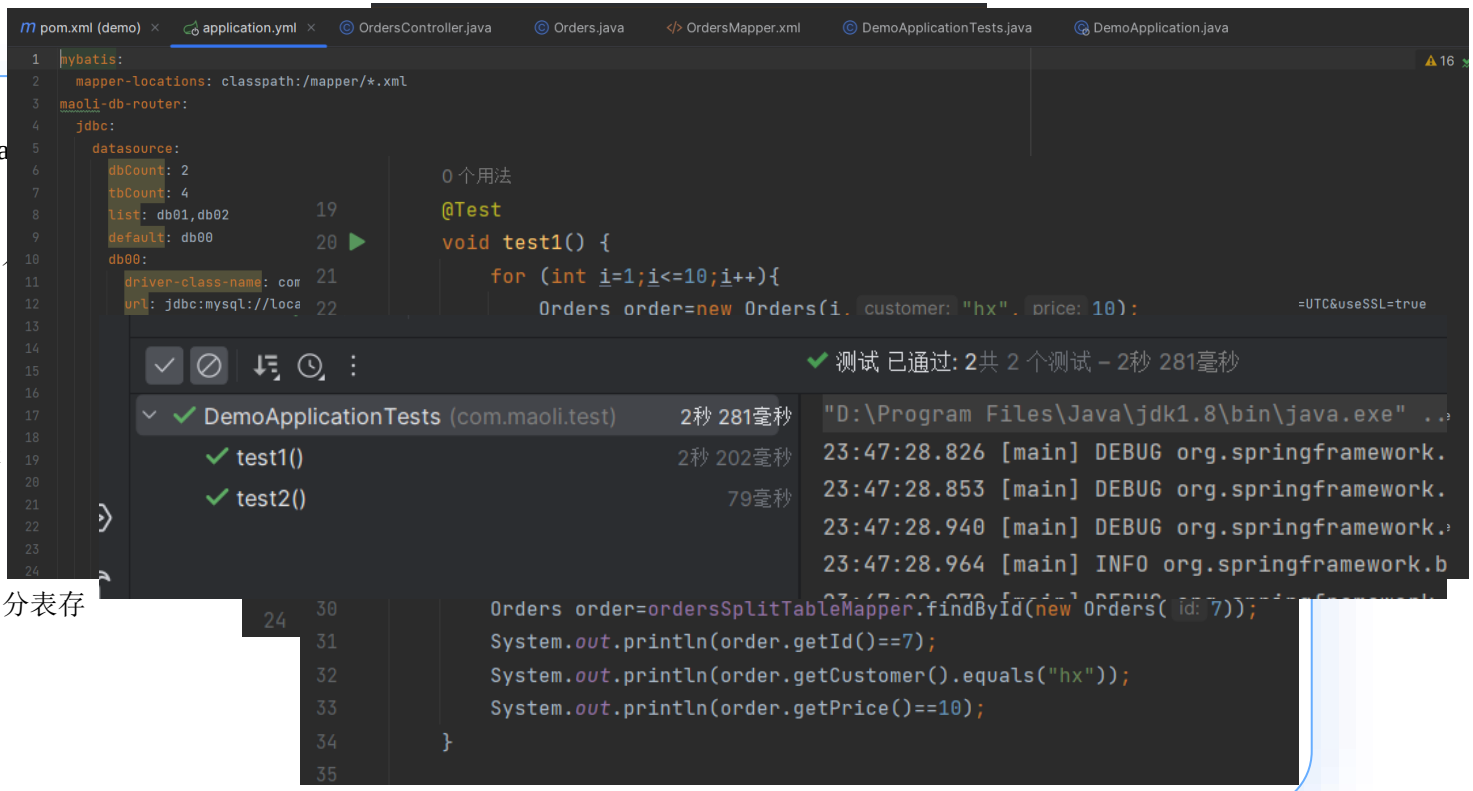
在demo项目的pom中引

配置数据源

使用注解进行分库分表

测试用例

十条数据按照设定规则分表存入两库八表之中



Sharding-jdbc安装

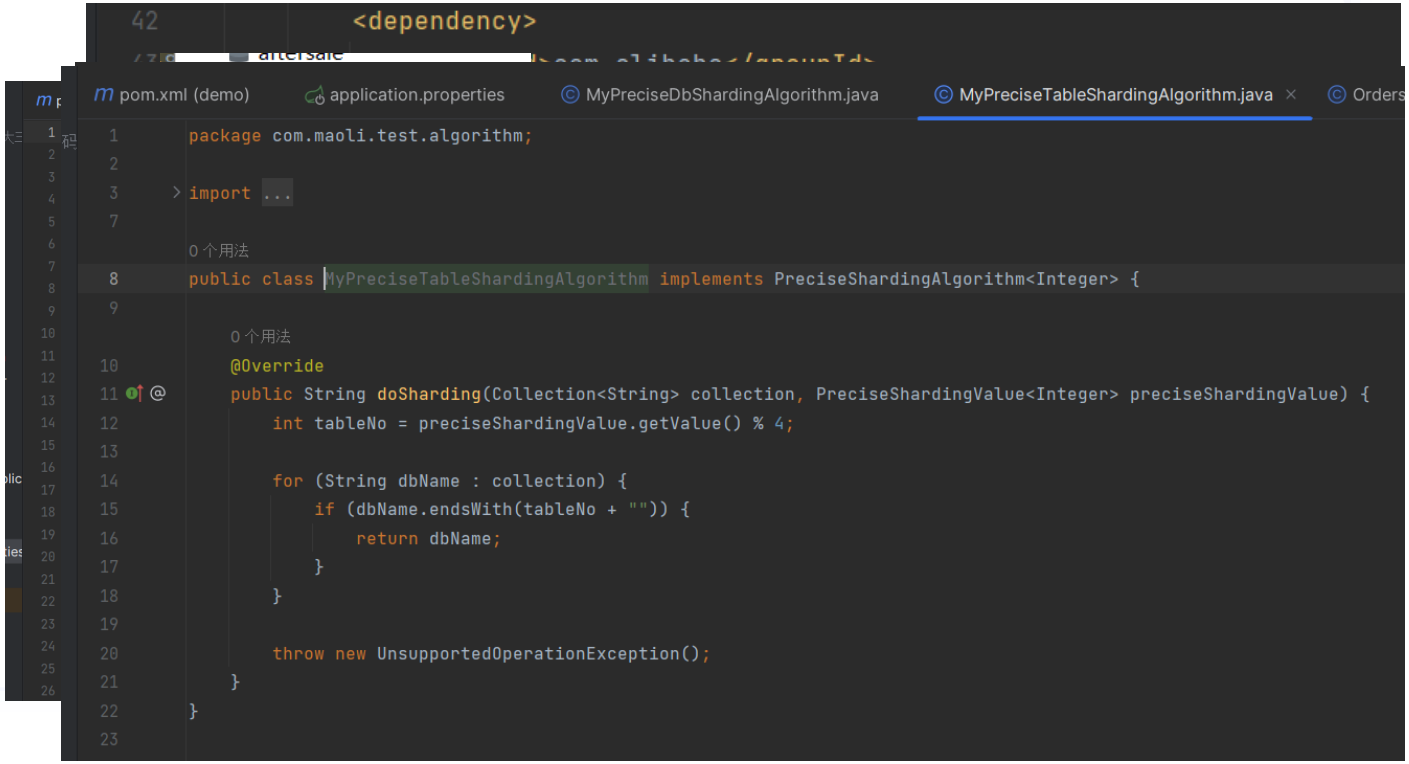
1.导入依赖

2.创建数据库db0 db1、分别创建四张表

3.引入依赖

4.分库算法

5.分表算法



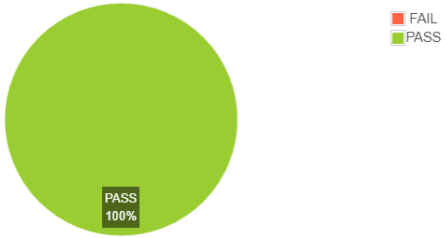
测试结果分析：

低负载：
500线程/s：

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.934	500 ms	1 sec 500 ms	Total
0.934	500 ms	1 sec 500 ms	POST http://localhost:8080/order

Requests Summary



demo

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**5000**	**0**	**0.00%**	**138.94**	**2**	**1850**	**4.00**	**713.90**	**964.00**	**1229.99**	**499.50**	**59.02**	**270.03**
POST http://localhost:8080/order	5000	0	0.00%	138.94	2	1850	4.00	713.90	964.00	1229.99	499.50	59.02	270.03

自行编写的分库分表中间件

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**5000**	**0**	**0.00%**	**18.52**	**10**	**124**	**17.00**	**24.00**	**32.95**	**56.00**	**498.75**	**58.93**	**269.63**
POST http://localhost:8080/order	5000	0	0.00%	18.52	10	124	17.00	24.00	32.95	56.00	498.75	58.93	269.63

Sharding-JDBC中间件

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**5000**	**0**	**0.00%**	**3.31**	**1**	**45**	**3.00**	**4.00**	**4.00**	**13.00**	**499.50**	**59.02**	**270.03**
POST http://localhost:8080/order	5000	0	0.00%	3.31	1	45	3.00	4.00	4.00	13.00	499.50	59.02	270.03

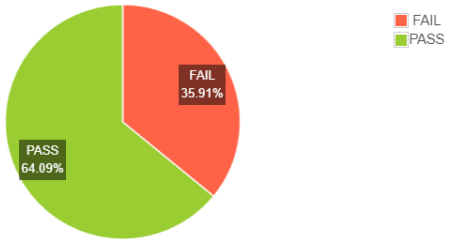
测试结果分析：

中负载：
1000线程/s：

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.608	500 ms	1 sec 500 ms	Total
0.608	500 ms	1 sec 500 ms	POST http://localhost:8080/order

Requests Summary

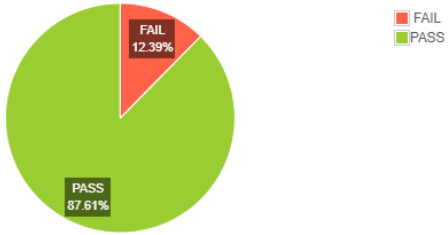


demo

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.156	500 ms	1 sec 500 ms	Total
0.156	500 ms	1 sec 500 ms	POST http://localhost:8080/order

Requests Summary

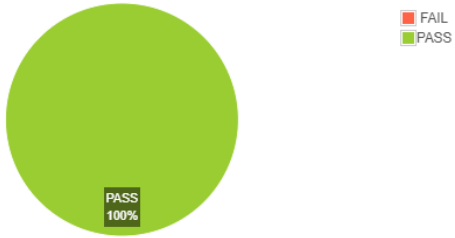


分库分表中间件

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.937	500 ms	1 sec 500 ms	Total
0.937	500 ms	1 sec 500 ms	POST http://localhost:8080/order

Requests Summary



Sharding-JDBC中间件

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	10000	0	0.00%	126.59	1	1422	3.00	685.00	913.00	1083.00	999.90	118.15	540.27
POST http://localhost:8080/order	10000	0	0.00%	126.59	1	1422	3.00	685.00	913.00	1083.00	999.90	118.15	540.27

测试结果分析：

高负载：
1500线程/s：

demo

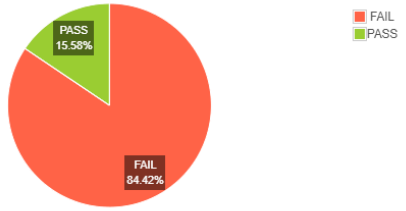
分库分表中间件

Sharding-JDBC中间件

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.156	500 ms	1 sec 500 ms	Total
0.156	500 ms	1 sec 500 ms	POST http://localhost:8080/order

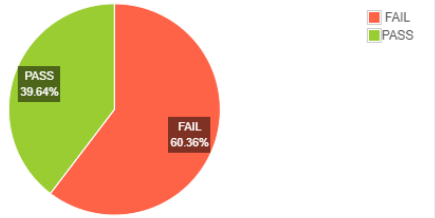
Requests Summary



APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.313	500 ms	1 sec 500 ms	Total
0.313	500 ms	1 sec 500 ms	POST http://localhost:8080/order

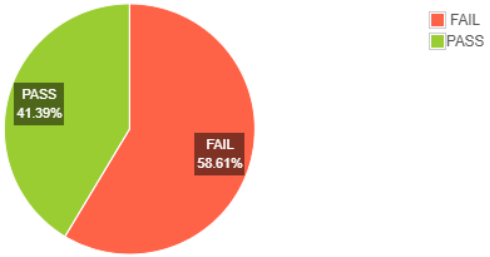
Requests Summary



APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.414	500 ms	1 sec 500 ms	Total
0.414	500 ms	1 sec 500 ms	POST http://localhost:8080/order

Requests Summary



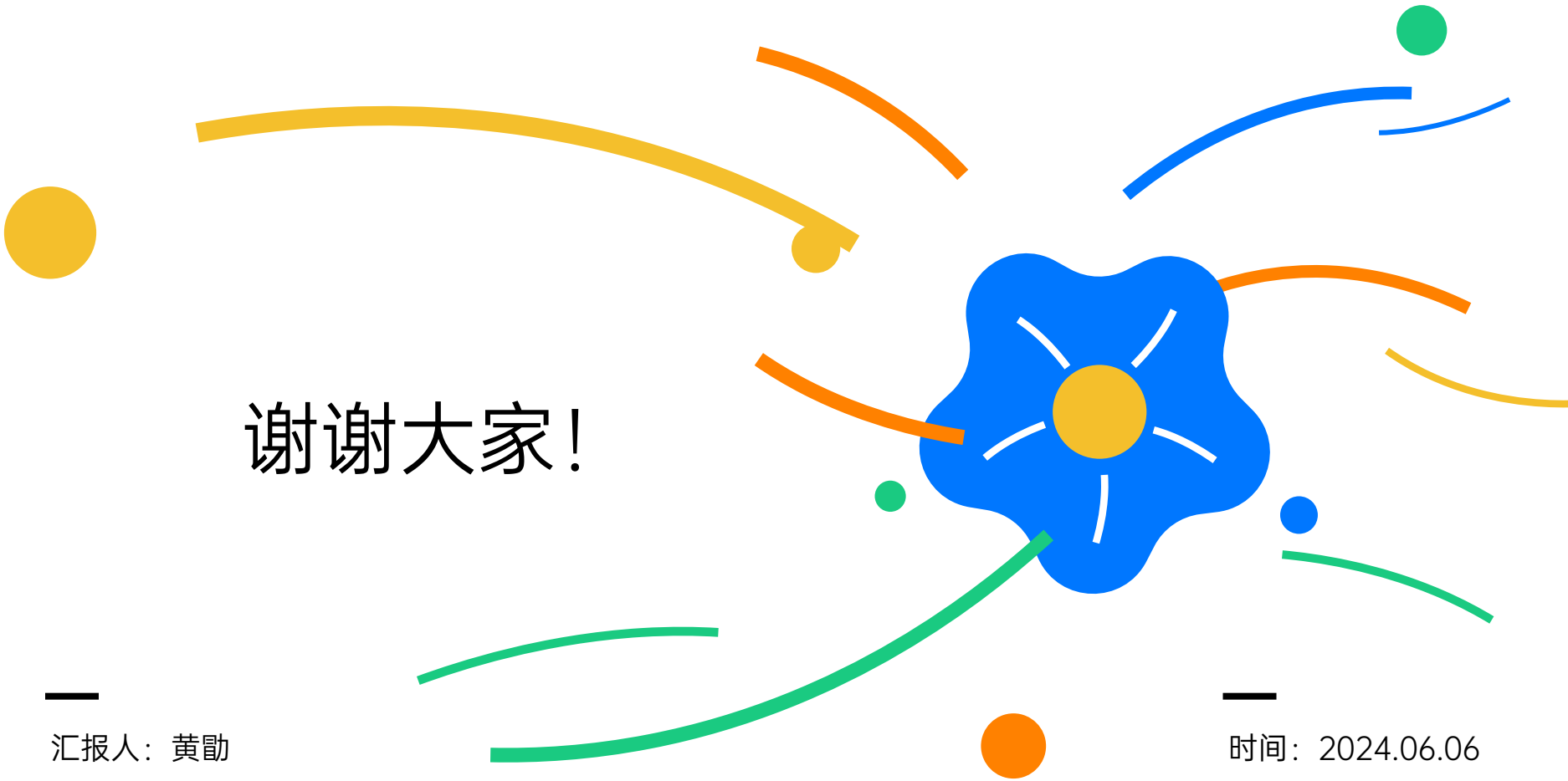
Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	15000	8792	58.61%	69.56	1	734	7.00	276.00	358.00	550.99	999.20	1379.19	223.54
POST http://localhost:8080/order	15000	8792	58.61%	69.56	1	734	7.00	276.00	358.00	550.99	999.20	1379.19	223.54

结论和展望

结论：
在并发量较小时，大部分情况下都能完美通过，在并发量逐渐增大的过程中，没有使用中间件的demo开始出现大量的错误，自己实现的中间件会少些，sharding-jdbc依旧稳定。而在并发量超大时，我们实现的中间件稳定性与Sharding-JDBC中间件相似，相差不到2%。

展望：
1.可以发现自己实现的中间件还存在效率上的问题，这可能是我们在aop 的实现上还存在一部分的冗余操作，可以在后续学习 shardingjdbc 的思路继续实现。
2.还可以尝试不同的分库分表策略，设置不同的算法来进行分库分表。



谢谢大家！

汇报人：黄勖

时间：2024.06.06