# NSRS号卡资源管理系统性能测试方案

## 1. 概述

### 1.1 测试目标

NSRS（Network SIM Resource System）号卡资源管理系统是一个专业的电信级SIM卡资源管理平台，本性能测试方案旨在验证系统在高并发、大数据量场景下的性能表现，确保系统满足电信运营商的业务需求。

### 1.2 系统架构概览

┌─────────────────────────────────────────────────────────────┐  
│ NSRS系统架构 │  
├─────────────────────────────────────────────────────────────┤  
│ 前端层: React 19 + TypeScript + Ant Design + Vite │  
├─────────────────────────────────────────────────────────────┤  
│ 网关层: Spring Cloud Gateway + 负载均衡 │  
├─────────────────────────────────────────────────────────────┤  
│ 应用层: Spring Boot 2.x + MyBatis-Plus │  
│ ├── nsrs-msisdn (号码资源管理) │  
│ ├── nsrs-simcard (SIM卡资源管理) │  
│ ├── nsrs-binding (号码IMSI绑定) │  
│ └── nsrs-busacc (业务受理) │  
├─────────────────────────────────────────────────────────────┤  
│ 缓存层: Redis Cluster │  
├─────────────────────────────────────────────────────────────┤  
│ 数据层: MySQL 8.0 + ShardingSphere (分库分表) │  
└─────────────────────────────────────────────────────────────┘

### 1.3 核心业务模块

* **号码资源管理**: 号码段管理、号码分配、状态变更
* **SIM卡资源管理**: SIM卡入库、分配、激活、回收
* **IMSI资源管理**: IMSI资源池管理、分配策略
* **绑定关系管理**: 号码与IMSI绑定、批量绑定、解绑操作
* **库存预警**: 实时库存监控、阈值预警

## 2. 性能测试策略

### 2.1 测试分类

#### 2.1.1 负载测试（Load Testing）

* **目标**: 验证系统在预期负载下的性能表现
* **场景**: 模拟正常业务高峰期的用户访问量
* **指标**: 响应时间、吞吐量、资源利用率

#### 2.1.2 压力测试（Stress Testing）

* **目标**: 确定系统的性能瓶颈和最大承载能力
* **场景**: 逐步增加负载直到系统性能显著下降
* **指标**: 最大并发用户数、系统崩溃点

#### 2.1.3 峰值测试（Spike Testing）

* **目标**: 验证系统在突发流量下的稳定性
* **场景**: 短时间内大量用户同时访问
* **指标**: 系统恢复时间、错误率

#### 2.1.4 容量测试（Volume Testing）

* **目标**: 验证系统在大数据量下的处理能力
* **场景**: 大量数据的批量操作
* **指标**: 数据处理速度、存储性能

#### 2.1.5 稳定性测试（Endurance Testing）

* **目标**: 验证系统长时间运行的稳定性
* **场景**: 持续负载运行24-72小时
* **指标**: 内存泄漏、性能衰减

### 2.2 测试环境规划

#### 2.2.1 硬件环境

# 应用服务器配置  
app\_servers:  
 count: 3  
 cpu: 8 cores  
 memory: 16GB  
 disk: 500GB SSD  
 network: 1Gbps  
  
# 数据库服务器配置  
db\_servers:  
 master:  
 cpu: 16 cores  
 memory: 32GB  
 disk: 1TB NVMe SSD  
 slaves:  
 count: 2  
 cpu: 8 cores  
 memory: 16GB  
 disk: 500GB SSD  
  
# Redis集群配置  
redis\_cluster:  
 nodes: 6 (3 master + 3 slave)  
 cpu: 4 cores per node  
 memory: 8GB per node  
 disk: 100GB SSD per node  
  
# 负载均衡器  
load\_balancer:  
 cpu: 4 cores  
 memory: 8GB  
 network: 10Gbps

#### 2.2.2 软件环境

# 操作系统  
os: CentOS 7.9 / Ubuntu 20.04 LTS  
  
# 应用运行时  
java\_version: OpenJDK 11  
node\_version: Node.js 18.x  
  
# 数据库  
mysql\_version: 8.0.35  
redis\_version: 7.0.x  
  
# 容器化  
docker\_version: 24.x  
kubernetes\_version: 1.28.x  
  
# 监控工具  
prometheus: 2.45.x  
grafana: 10.x  
jaeger: 1.49.x

### 2.3 测试数据准备

#### 2.3.1 基础数据量规划

-- 号码资源数据  
INSERT INTO number\_resource   
SELECT   
 get\_next\_number\_resource\_id() as number\_id,  
 CONCAT('138', LPAD(ROW\_NUMBER() OVER(), 8, '0')) as number,  
 2 as number\_type,  
 1 as segment\_id,  
 1 as level\_id,  
 1 as pattern\_id,  
 1 as hlr\_id,  
 NULL as iccid,  
 1 as status,  
 0.00 as charge,  
 'TEST\_ORG' as attributive\_org,  
 'Performance Test Data' as remark,  
 NOW() as create\_time,  
 NOW() as update\_time,  
 1 as create\_user\_id,  
 1 as update\_user\_id  
FROM (  
 SELECT @row\_number := @row\_number + 1 as rn  
 FROM information\_schema.tables t1, information\_schema.tables t2,  
 (SELECT @row\_number := 0) r  
 LIMIT 10000000 -- 1000万号码资源  
) numbers;  
  
-- IMSI资源数据  
INSERT INTO imsi\_resource   
SELECT   
 get\_next\_sequence\_value('imsi\_resource\_id\_seq') as imsi\_id,  
 CONCAT('460001', LPAD(ROW\_NUMBER() OVER(), 10, '0')) as imsi,  
 1 as imsi\_type,  
 1 as group\_id,  
 1 as supplier\_id,  
 'test123' as password,  
 NULL as bill\_id,  
 1 as status,  
 NOW() as create\_time,  
 NOW() as update\_time,  
 1 as create\_user\_id,  
 1 as update\_user\_id  
FROM (  
 SELECT @row\_number := @row\_number + 1 as rn  
 FROM information\_schema.tables t1, information\_schema.tables t2,  
 (SELECT @row\_number := 0) r  
 LIMIT 5000000 -- 500万IMSI资源  
) imsis;  
  
-- SIM卡数据  
INSERT INTO sim\_card   
SELECT   
 get\_next\_sequence\_value('sim\_card\_id\_seq') as card\_id,  
 CONCAT('89860001', LPAD(ROW\_NUMBER() OVER(), 12, '0')) as iccid,  
 CONCAT('460001', LPAD(ROW\_NUMBER() OVER(), 10, '0')) as imsi,  
 1 as batch\_id,  
 1 as card\_type\_id,  
 1 as spec\_id,  
 1 as data\_type,  
 1 as supplier\_id,  
 1 as org\_id,  
 1 as status,  
 'Performance Test SIM Card' as remark,  
 NOW() as create\_time,  
 NOW() as update\_time,  
 1 as create\_user\_id,  
 1 as update\_user\_id  
FROM (  
 SELECT @row\_number := @row\_number + 1 as rn  
 FROM information\_schema.tables t1, information\_schema.tables t2,  
 (SELECT @row\_number := 0) r  
 LIMIT 5000000 -- 500万SIM卡  
) cards;

#### 2.3.2 测试用户数据

# 并发用户配置  
concurrent\_users:  
 low\_load: 100 # 低负载  
 normal\_load: 500 # 正常负载  
 high\_load: 1000 # 高负载  
 peak\_load: 2000 # 峰值负载  
 stress\_load: 5000 # 压力测试  
  
# 用户行为模拟  
user\_scenarios:  
 - name: "号码查询"  
 weight: 30%  
 think\_time: 1-3s  
 - name: "号码分配"  
 weight: 20%  
 think\_time: 2-5s  
 - name: "SIM卡管理"  
 weight: 25%  
 think\_time: 1-4s  
 - name: "绑定操作"  
 weight: 15%  
 think\_time: 3-6s  
 - name: "批量操作"  
 weight: 10%  
 think\_time: 5-10s

## 3. 核心业务场景测试

### 3.1 号码资源管理性能测试

#### 3.1.1 号码查询性能测试

// JMeter测试脚本示例  
const numberQueryTest = {  
 testPlan: "号码查询性能测试",  
 threadGroups: [  
 {  
 name: "号码查询并发测试",  
 threads: 500,  
 rampUp: 60,  
 duration: 300,  
 requests: [  
 {  
 name: "按号码查询",  
 method: "GET",  
 url: "/api/numbers/search",  
 parameters: {  
 number: "${\_\_Random(13800000000,13899999999)}",  
 pageSize: 20,  
 pageNum: 1  
 },  
 assertions: [  
 { responseCode: 200 },  
 { responseTime: "< 500ms" },  
 { jsonPath: "$.success", expectedValue: true }  
 ]  
 },  
 {  
 name: "按状态查询",  
 method: "GET",  
 url: "/api/numbers/search",  
 parameters: {  
 status: "${\_\_Random(1,7)}",  
 pageSize: 50,  
 pageNum: "${\_\_Random(1,100)}"  
 },  
 assertions: [  
 { responseCode: 200 },  
 { responseTime: "< 800ms" }  
 ]  
 }  
 ]  
 }  
 ],  
 performanceTargets: {  
 averageResponseTime: "< 300ms",  
 p95ResponseTime: "< 800ms",  
 p99ResponseTime: "< 1500ms",  
 throughput: "> 1000 TPS",  
 errorRate: "< 0.1%"  
 }  
};

#### 3.1.2 号码分配性能测试

# 号码分配测试场景  
number\_allocation\_test:  
 scenario: "号码分配压力测试"  
 concurrent\_users: 200  
 test\_duration: 600s  
 ramp\_up\_time: 120s  
   
 test\_cases:  
 - name: "单个号码分配"  
 weight: 70%  
 request:  
 method: POST  
 url: "/api/numbers/allocate"  
 body:  
 levelId: 1  
 quantity: 1  
 orgId: "${orgId}"  
 remark: "Performance Test"  
 assertions:  
 - response\_code: 200  
 - response\_time: "< 1000ms"  
 - json\_path: "$.data.allocatedNumbers.length == 1"  
   
 - name: "批量号码分配"  
 weight: 30%  
 request:  
 method: POST  
 url: "/api/numbers/batch-allocate"  
 body:  
 levelId: 1  
 quantity: "${\_\_Random(10,100)}"  
 orgId: "${orgId}"  
 remark: "Batch Performance Test"  
 assertions:  
 - response\_code: 200  
 - response\_time: "< 5000ms"  
 - json\_path: "$.data.allocatedNumbers.length > 0"  
  
 performance\_targets:  
 average\_response\_time: "< 800ms"  
 p95\_response\_time: "< 2000ms"  
 throughput: "> 200 TPS"  
 error\_rate: "< 0.5%"  
 database\_cpu: "< 80%"  
 application\_memory: "< 12GB"

### 3.2 SIM卡资源管理性能测试

#### 3.2.1 SIM卡批量入库性能测试

# Python性能测试脚本  
import asyncio  
import aiohttp  
import time  
import json  
from concurrent.futures import ThreadPoolExecutor  
  
class SimCardBatchImportTest:  
 def \_\_init\_\_(self, base\_url, concurrent\_users=100):  
 self.base\_url = base\_url  
 self.concurrent\_users = concurrent\_users  
 self.results = []  
   
 async def batch\_import\_test(self, session, batch\_size=1000):  
 """SIM卡批量入库测试"""  
 start\_time = time.time()  
   
 # 生成测试数据  
 sim\_cards = []  
 for i in range(batch\_size):  
 sim\_cards.append({  
 "iccid": f"89860001{str(int(time.time() \* 1000000) + i).zfill(12)}",  
 "imsi": f"460001{str(int(time.time() \* 1000) + i).zfill(10)}",  
 "batchId": 1,  
 "cardTypeId": 1,  
 "specId": 1,  
 "dataType": 1,  
 "supplierId": 1,  
 "orgId": 1,  
 "remark": "Performance Test Batch Import"  
 })  
   
 try:  
 async with session.post(  
 f"{self.base\_url}/api/sim-cards/batch-import",  
 json={"simCards": sim\_cards},  
 headers={"Content-Type": "application/json"}  
 ) as response:  
 response\_time = time.time() - start\_time  
 result = {  
 "status\_code": response.status,  
 "response\_time": response\_time,  
 "batch\_size": batch\_size,  
 "success": response.status == 200  
 }  
   
 if response.status == 200:  
 data = await response.json()  
 result["imported\_count"] = data.get("data", {}).get("importedCount", 0)  
   
 self.results.append(result)  
 return result  
   
 except Exception as e:  
 self.results.append({  
 "status\_code": 0,  
 "response\_time": time.time() - start\_time,  
 "batch\_size": batch\_size,  
 "success": False,  
 "error": str(e)  
 })  
   
 async def run\_concurrent\_test(self, test\_duration=300):  
 """运行并发测试"""  
 async with aiohttp.ClientSession() as session:  
 tasks = []  
 start\_time = time.time()  
   
 while time.time() - start\_time < test\_duration:  
 if len(tasks) < self.concurrent\_users:  
 task = asyncio.create\_task(  
 self.batch\_import\_test(session, batch\_size=500)  
 )  
 tasks.append(task)  
   
 # 清理完成的任务  
 tasks = [task for task in tasks if not task.done()]  
 await asyncio.sleep(0.1)  
   
 # 等待所有任务完成  
 await asyncio.gather(\*tasks, return\_exceptions=True)  
   
 def analyze\_results(self):  
 """分析测试结果"""  
 if not self.results:  
 return  
   
 successful\_results = [r for r in self.results if r["success"]]  
 response\_times = [r["response\_time"] for r in successful\_results]  
   
 print(f"\n=== SIM卡批量入库性能测试结果 ===")  
 print(f"总请求数: {len(self.results)}")  
 print(f"成功请求数: {len(successful\_results)}")  
 print(f"成功率: {len(successful\_results)/len(self.results)\*100:.2f}%")  
   
 if response\_times:  
 print(f"平均响应时间: {sum(response\_times)/len(response\_times):.3f}s")  
 print(f"最小响应时间: {min(response\_times):.3f}s")  
 print(f"最大响应时间: {max(response\_times):.3f}s")  
   
 response\_times.sort()  
 p95\_index = int(len(response\_times) \* 0.95)  
 p99\_index = int(len(response\_times) \* 0.99)  
 print(f"P95响应时间: {response\_times[p95\_index]:.3f}s")  
 print(f"P99响应时间: {response\_times[p99\_index]:.3f}s")  
   
 total\_imported = sum(r.get("imported\_count", 0) for r in successful\_results)  
 total\_time = max(response\_times) if response\_times else 0  
 if total\_time > 0:  
 throughput = total\_imported / total\_time  
 print(f"吞吐量: {throughput:.2f} records/second")  
  
# 运行测试  
if \_\_name\_\_ == "\_\_main\_\_":  
 test = SimCardBatchImportTest("http://localhost:8088", concurrent\_users=50)  
 asyncio.run(test.run\_concurrent\_test(test\_duration=600))  
 test.analyze\_results()

### 3.3 绑定关系管理性能测试

#### 3.3.1 批量绑定性能测试

#!/bin/bash  
# 批量绑定性能测试脚本  
  
BASE\_URL="http://localhost:8088"  
CONCURRENT\_USERS=100  
TEST\_DURATION=600  
BATCH\_SIZE=1000  
  
echo "=== NSRS批量绑定性能测试 ==="  
echo "测试参数:"  
echo " 并发用户数: $CONCURRENT\_USERS"  
echo " 测试时长: ${TEST\_DURATION}秒"  
echo " 批量大小: $BATCH\_SIZE"  
echo " 目标URL: $BASE\_URL"  
echo ""  
  
# 创建测试数据文件  
echo "生成测试数据..."  
cat > batch\_binding\_data.json << EOF  
{  
 "orderId": $(date +%s),  
 "bindingType": 2,  
 "operatorUserId": 1,  
 "remark": "Performance Test Batch Binding",  
 "bindings": [  
EOF  
  
# 生成绑定数据  
for i in $(seq 1 $BATCH\_SIZE); do  
 number="138$(printf "%08d" $i)"  
 imsi="460001$(printf "%010d" $i)"  
 iccid="89860001$(printf "%012d" $i)"  
   
 echo " {" >> batch\_binding\_data.json  
 echo " \"number\": \"$number\"," >> batch\_binding\_data.json  
 echo " \"imsi\": \"$imsi\"," >> batch\_binding\_data.json  
 echo " \"iccid\": \"$iccid\"" >> batch\_binding\_data.json  
   
 if [ $i -eq $BATCH\_SIZE ]; then  
 echo " }" >> batch\_binding\_data.json  
 else  
 echo " }," >> batch\_binding\_data.json  
 fi  
done  
  
echo " ]" >> batch\_binding\_data.json  
echo "}" >> batch\_binding\_data.json  
  
echo "测试数据生成完成，开始性能测试..."  
echo ""  
  
# 使用Apache Bench进行并发测试  
echo "执行批量绑定性能测试..."  
ab -n $((CONCURRENT\_USERS \* 10)) -c $CONCURRENT\_USERS \  
 -T "application/json" \  
 -p batch\_binding\_data.json \  
 -g batch\_binding\_results.tsv \  
 "$BASE\_URL/api/bindings/batch-bind"  
  
echo ""  
echo "=== 测试结果分析 ==="  
  
# 分析结果  
if [ -f batch\_binding\_results.tsv ]; then  
 echo "响应时间分布:"  
 awk 'NR>1 {sum+=$9; count++; if($9>max) max=$9; if(min=="" || $9<min) min=$9}   
 END {print "平均响应时间: " sum/count "ms";   
 print "最小响应时间: " min "ms";   
 print "最大响应时间: " max "ms"}' batch\_binding\_results.tsv  
   
 echo ""  
 echo "成功率统计:"  
 awk 'NR>1 {total++; if($4==200) success++}   
 END {print "总请求数: " total;   
 print "成功请求数: " success;   
 print "成功率: " (success/total)\*100 "%"}' batch\_binding\_results.tsv  
fi  
  
# 清理临时文件  
rm -f batch\_binding\_data.json batch\_binding\_results.tsv  
  
echo ""  
echo "批量绑定性能测试完成！"

## 4. 数据库性能测试

### 4.1 MySQL性能基准测试

#!/bin/bash  
# MySQL性能基准测试脚本  
  
DB\_HOST="mysql-master"  
DB\_PORT="3306"  
DB\_USER="nsrs\_user"  
DB\_PASSWORD="password"  
DB\_NAME="nsrs"  
  
echo "=== MySQL性能基准测试 ==="  
  
# 1. 准备测试数据  
echo "1. 准备测试数据..."  
sysbench --db-driver=mysql \  
 --mysql-host=$DB\_HOST \  
 --mysql-port=$DB\_PORT \  
 --mysql-user=$DB\_USER \  
 --mysql-password=$DB\_PASSWORD \  
 --mysql-db=$DB\_NAME \  
 --table-size=1000000 \  
 --tables=10 \  
 --threads=16 \  
 oltp\_read\_write prepare  
  
# 2. 读写混合测试  
echo "\n2. 读写混合性能测试..."  
sysbench --db-driver=mysql \  
 --mysql-host=$DB\_HOST \  
 --mysql-port=$DB\_PORT \  
 --mysql-user=$DB\_USER \  
 --mysql-password=$DB\_PASSWORD \  
 --mysql-db=$DB\_NAME \  
 --table-size=1000000 \  
 --tables=10 \  
 --threads=32 \  
 --time=300 \  
 --report-interval=10 \  
 oltp\_read\_write run  
  
# 3. 只读性能测试  
echo "\n3. 只读性能测试..."  
sysbench --db-driver=mysql \  
 --mysql-host=$DB\_HOST \  
 --mysql-port=$DB\_PORT \  
 --mysql-user=$DB\_USER \  
 --mysql-password=$DB\_PASSWORD \  
 --mysql-db=$DB\_NAME \  
 --table-size=1000000 \  
 --tables=10 \  
 --threads=64 \  
 --time=300 \  
 --report-interval=10 \  
 oltp\_read\_only run  
  
# 4. 只写性能测试  
echo "\n4. 只写性能测试..."  
sysbench --db-driver=mysql \  
 --mysql-host=$DB\_HOST \  
 --mysql-port=$DB\_PORT \  
 --mysql-user=$DB\_USER \  
 --mysql-password=$DB\_PASSWORD \  
 --mysql-db=$DB\_NAME \  
 --table-size=1000000 \  
 --tables=10 \  
 --threads=16 \  
 --time=300 \  
 --report-interval=10 \  
 oltp\_write\_only run  
  
# 5. 清理测试数据  
echo "\n5. 清理测试数据..."  
sysbench --db-driver=mysql \  
 --mysql-host=$DB\_HOST \  
 --mysql-port=$DB\_PORT \  
 --mysql-user=$DB\_USER \  
 --mysql-password=$DB\_PASSWORD \  
 --mysql-db=$DB\_NAME \  
 --tables=10 \  
 oltp\_read\_write cleanup  
  
echo "\nMySQL性能基准测试完成！"

### 4.2 分库分表性能验证

-- 分库分表性能验证SQL脚本  
  
-- 1. 测试IMSI资源表分表查询性能  
EXPLAIN SELECT \* FROM imsi\_resource\_0 WHERE imsi = '460001000000001';  
EXPLAIN SELECT \* FROM imsi\_resource WHERE imsi = '460001000000001';  
  
-- 2. 测试跨分表统计查询性能  
SELECT   
 COUNT(\*) as total\_count,  
 SUM(CASE WHEN status = 1 THEN 1 ELSE 0 END) as idle\_count,  
 SUM(CASE WHEN status = 2 THEN 1 ELSE 0 END) as bound\_count,  
 AVG(CASE WHEN status = 1 THEN 1 ELSE 0 END) \* 100 as idle\_percentage  
FROM imsi\_resource;  
  
-- 3. 测试批量插入性能  
INSERT INTO imsi\_resource (imsi\_id, imsi, imsi\_type, group\_id, supplier\_id, status)  
SELECT   
 get\_next\_sequence\_value('imsi\_resource\_id\_seq'),  
 CONCAT('460001', LPAD(@row\_number := @row\_number + 1, 10, '0')),  
 1,  
 1,  
 1,  
 1  
FROM   
 (SELECT @row\_number := 0) r,  
 information\_schema.tables t1  
LIMIT 100000;  
  
-- 4. 测试分表JOIN查询性能  
SELECT   
 n.number,  
 i.imsi,  
 b.binding\_time,  
 b.binding\_status  
FROM number\_resource n  
JOIN number\_imsi\_binding b ON n.number = b.number  
JOIN imsi\_resource i ON b.imsi = i.imsi  
WHERE n.status = 3 AND b.binding\_status = 1  
LIMIT 1000;  
  
-- 5. 测试分区表查询性能（号码操作日志表）  
SELECT   
 operation\_type,  
 COUNT(\*) as operation\_count,  
 AVG(CASE WHEN result\_status = 1 THEN 1 ELSE 0 END) \* 100 as success\_rate  
FROM number\_operation\_log   
WHERE operation\_time >= DATE\_SUB(NOW(), INTERVAL 30 DAY)  
GROUP BY operation\_type;

## 5. 缓存性能测试

### 5.1 Redis集群性能测试

#!/bin/bash  
# Redis集群性能测试脚本  
  
REDIS\_HOST="redis-cluster"  
REDIS\_PORT="7000"  
  
echo "=== Redis集群性能测试 ==="  
  
# 1. 基础性能测试  
echo "1. Redis基础性能测试..."  
redis-benchmark -h $REDIS\_HOST -p $REDIS\_PORT \  
 -n 1000000 -c 100 -d 1024 \  
 -t set,get,incr,lpush,rpush,lpop,rpop,sadd,hset,spop,zadd,zpopmin,lrange  
  
# 2. 管道性能测试  
echo "\n2. Redis管道性能测试..."  
redis-benchmark -h $REDIS\_HOST -p $REDIS\_PORT \  
 -n 1000000 -c 100 -P 16 \  
 -t set,get  
  
# 3. 大数据量测试  
echo "\n3. Redis大数据量测试..."  
redis-benchmark -h $REDIS\_HOST -p $REDIS\_PORT \  
 -n 100000 -c 50 -d 10240 \  
 -t set,get  
  
# 4. 集群模式特定测试  
echo "\n4. Redis集群模式测试..."  
for i in {1..10}; do  
 echo "测试轮次 $i:"  
 redis-benchmark -h $REDIS\_HOST -p $REDIS\_PORT \  
 -n 100000 -c 50 \  
 --cluster \  
 -t set,get  
 sleep 2  
done  
  
echo "\nRedis集群性能测试完成！"

### 5.2 缓存命中率测试

# 缓存命中率测试脚本  
import redis  
import random  
import time  
import threading  
from concurrent.futures import ThreadPoolExecutor  
  
class CacheHitRateTest:  
 def \_\_init\_\_(self, redis\_host='localhost', redis\_port=6379):  
 self.redis\_client = redis.Redis(  
 host=redis\_host,   
 port=redis\_port,   
 decode\_responses=True  
 )  
 self.hit\_count = 0  
 self.miss\_count = 0  
 self.total\_requests = 0  
 self.lock = threading.Lock()  
   
 def simulate\_number\_query(self, number):  
 """模拟号码查询缓存"""  
 cache\_key = f"number:{number}"  
   
 # 尝试从缓存获取  
 cached\_data = self.redis\_client.get(cache\_key)  
   
 with self.lock:  
 self.total\_requests += 1  
   
 if cached\_data:  
 self.hit\_count += 1  
 return {"source": "cache", "data": cached\_data}  
 else:  
 self.miss\_count += 1  
 # 模拟数据库查询  
 time.sleep(0.01) # 模拟数据库查询延迟  
   
 # 模拟数据库返回的数据  
 db\_data = {  
 "number": number,  
 "status": random.choice([1, 2, 3, 4, 5]),  
 "level\_id": random.choice([1, 2, 3]),  
 "create\_time": int(time.time())  
 }  
   
 # 写入缓存，TTL 5分钟  
 self.redis\_client.setex(  
 cache\_key,   
 300,   
 str(db\_data)  
 )  
   
 return {"source": "database", "data": db\_data}  
   
 def simulate\_sim\_card\_query(self, iccid):  
 """模拟SIM卡查询缓存"""  
 cache\_key = f"sim\_card:{iccid}"  
   
 cached\_data = self.redis\_client.hgetall(cache\_key)  
   
 with self.lock:  
 self.total\_requests += 1  
   
 if cached\_data:  
 self.hit\_count += 1  
 return {"source": "cache", "data": cached\_data}  
 else:  
 self.miss\_count += 1  
 time.sleep(0.015) # 模拟数据库查询延迟  
   
 db\_data = {  
 "iccid": iccid,  
 "imsi": f"460001{random.randint(1000000000, 9999999999)}",  
 "status": random.choice([1, 2, 3, 4, 5]),  
 "card\_type\_id": random.choice([1, 2, 3]),  
 "supplier\_id": random.choice([1, 2, 3, 4])  
 }  
   
 # 使用Hash存储，TTL 10分钟  
 pipe = self.redis\_client.pipeline()  
 pipe.hset(cache\_key, mapping=db\_data)  
 pipe.expire(cache\_key, 600)  
 pipe.execute()  
   
 return {"source": "database", "data": db\_data}  
   
 def worker\_thread(self, thread\_id, duration=300):  
 """工作线程"""  
 start\_time = time.time()  
   
 while time.time() - start\_time < duration:  
 # 随机选择查询类型  
 if random.random() < 0.6: # 60%概率查询号码  
 number = f"138{random.randint(10000000, 99999999)}"  
 self.simulate\_number\_query(number)  
 else: # 40%概率查询SIM卡  
 iccid = f"89860001{random.randint(100000000000, 999999999999)}"  
 self.simulate\_sim\_card\_query(iccid)  
   
 # 模拟用户思考时间  
 time.sleep(random.uniform(0.1, 0.5))  
   
 def run\_test(self, num\_threads=50, duration=300):  
 """运行缓存命中率测试"""  
 print(f"开始缓存命中率测试...")  
 print(f"线程数: {num\_threads}")  
 print(f"测试时长: {duration}秒")  
 print("")  
   
 # 预热缓存  
 print("预热缓存...")  
 for i in range(1000):  
 number = f"138{str(i).zfill(8)}"  
 self.simulate\_number\_query(number)  
   
 # 重置计数器  
 self.hit\_count = 0  
 self.miss\_count = 0  
 self.total\_requests = 0  
   
 # 启动测试线程  
 with ThreadPoolExecutor(max\_workers=num\_threads) as executor:  
 futures = []  
 for i in range(num\_threads):  
 future = executor.submit(self.worker\_thread, i, duration)  
 futures.append(future)  
   
 # 等待所有线程完成  
 for future in futures:  
 future.result()  
   
 # 输出结果  
 self.print\_results()  
   
 def print\_results(self):  
 """输出测试结果"""  
 hit\_rate = (self.hit\_count / self.total\_requests) \* 100 if self.total\_requests > 0 else 0  
   
 print("\n=== 缓存命中率测试结果 ===")  
 print(f"总请求数: {self.total\_requests}")  
 print(f"缓存命中数: {self.hit\_count}")  
 print(f"缓存未命中数: {self.miss\_count}")  
 print(f"缓存命中率: {hit\_rate:.2f}%")  
   
 # 获取Redis统计信息  
 info = self.redis\_client.info()  
 print(f"\nRedis统计信息:")  
 print(f"已用内存: {info.get('used\_memory\_human', 'N/A')}")  
 print(f"键总数: {info.get('db0', {}).get('keys', 0)}")  
 print(f"命令处理总数: {info.get('total\_commands\_processed', 0)}")  
 print(f"键空间命中率: {info.get('keyspace\_hit\_rate', 'N/A')}")  
  
# 运行测试  
if \_\_name\_\_ == "\_\_main\_\_":  
 test = CacheHitRateTest(redis\_host='localhost', redis\_port=6379)  
 test.run\_test(num\_threads=100, duration=600)

## 6. 性能监控与分析

### 6.1 监控指标体系

#### 6.1.1 应用层监控指标

# Prometheus监控配置  
application\_metrics:  
 # HTTP请求指标  
 http\_metrics:  
 - name: http\_requests\_total  
 type: counter  
 labels: [method, endpoint, status]  
 description: "HTTP请求总数"  
   
 - name: http\_request\_duration\_seconds  
 type: histogram  
 labels: [method, endpoint]  
 buckets: [0.1, 0.25, 0.5, 1, 2.5, 5, 10]  
 description: "HTTP请求响应时间"  
   
 - name: http\_request\_size\_bytes  
 type: histogram  
 labels: [method, endpoint]  
 description: "HTTP请求大小"  
   
 # 业务指标  
 business\_metrics:  
 - name: number\_allocation\_total  
 type: counter  
 labels: [level, org, result]  
 description: "号码分配总数"  
   
 - name: sim\_card\_operations\_total  
 type: counter  
 labels: [operation\_type, result]  
 description: "SIM卡操作总数"  
   
 - name: binding\_operations\_total  
 type: counter  
 labels: [binding\_type, result]  
 description: "绑定操作总数"  
   
 - name: batch\_operation\_duration\_seconds  
 type: histogram  
 labels: [operation\_type]  
 buckets: [1, 5, 10, 30, 60, 300]  
 description: "批量操作耗时"  
   
 # JVM指标  
 jvm\_metrics:  
 - name: jvm\_memory\_used\_bytes  
 type: gauge  
 labels: [area]  
 description: "JVM内存使用量"  
   
 - name: jvm\_gc\_collection\_seconds  
 type: summary  
 labels: [gc]  
 description: "GC耗时"  
   
 - name: jvm\_threads\_current  
 type: gauge  
 description: "当前线程数"  
  
# 性能阈值配置  
performance\_thresholds:  
 response\_time:  
 p50: 200ms  
 p95: 800ms  
 p99: 2000ms  
   
 throughput:  
 number\_query: 1000 TPS  
 number\_allocation: 200 TPS  
 sim\_card\_import: 500 records/second  
 batch\_binding: 100 TPS  
   
 error\_rate:  
 max\_error\_rate: 0.1%  
 max\_timeout\_rate: 0.05%  
   
 resource\_usage:  
 cpu\_usage: 80%  
 memory\_usage: 85%  
 disk\_usage: 90%  
 network\_usage: 80%

#### 6.1.2 数据库监控指标

-- MySQL性能监控查询  
  
-- 1. 查询性能统计  
SELECT   
 schema\_name,  
 digest\_text,  
 count\_star as exec\_count,  
 avg\_timer\_wait/1000000000 as avg\_latency\_ms,  
 max\_timer\_wait/1000000000 as max\_latency\_ms,  
 sum\_rows\_examined/count\_star as avg\_rows\_examined,  
 sum\_rows\_sent/count\_star as avg\_rows\_sent  
FROM performance\_schema.events\_statements\_summary\_by\_digest   
WHERE schema\_name = 'nsrs'  
ORDER BY avg\_timer\_wait DESC  
LIMIT 20;  
  
-- 2. 表访问统计  
SELECT   
 object\_schema,  
 object\_name,  
 count\_read,  
 count\_write,  
 count\_fetch,  
 count\_insert,  
 count\_update,  
 count\_delete,  
 sum\_timer\_wait/1000000000 as total\_latency\_ms  
FROM performance\_schema.table\_io\_waits\_summary\_by\_table   
WHERE object\_schema = 'nsrs'  
ORDER BY sum\_timer\_wait DESC  
LIMIT 20;  
  
-- 3. 索引使用统计  
SELECT   
 object\_schema,  
 object\_name,  
 index\_name,  
 count\_fetch,  
 count\_insert,  
 count\_update,  
 count\_delete,  
 sum\_timer\_wait/1000000000 as total\_latency\_ms  
FROM performance\_schema.table\_io\_waits\_summary\_by\_index\_usage   
WHERE object\_schema = 'nsrs'  
ORDER BY sum\_timer\_wait DESC  
LIMIT 20;  
  
-- 4. 连接统计  
SELECT   
 user,  
 host,  
 current\_connections,  
 total\_connections,  
 max\_session\_controlled\_memory,  
 max\_session\_total\_memory  
FROM performance\_schema.accounts  
WHERE user IS NOT NULL  
ORDER BY current\_connections DESC;  
  
-- 5. 锁等待统计  
SELECT   
 object\_schema,  
 object\_name,  
 index\_name,  
 lock\_type,  
 lock\_mode,  
 count\_star as lock\_count,  
 sum\_timer\_wait/1000000000 as total\_wait\_time\_ms,  
 avg\_timer\_wait/1000000000 as avg\_wait\_time\_ms  
FROM performance\_schema.events\_waits\_summary\_by\_instance  
WHERE event\_name LIKE 'wait/lock%'  
AND object\_schema = 'nsrs'  
ORDER BY sum\_timer\_wait DESC  
LIMIT 20;

### 6.2 性能分析工具

#### 6.2.1 APM集成配置

# application.yml - APM配置  
spring:  
 application:  
 name: nsrs-application  
   
# Micrometer配置  
management:  
 endpoints:  
 web:  
 exposure:  
 include: "\*"  
 endpoint:  
 health:  
 show-details: always  
 metrics:  
 enabled: true  
 metrics:  
 export:  
 prometheus:  
 enabled: true  
 step: 10s  
 influx:  
 enabled: false  
 distribution:  
 percentiles-histogram:  
 http.server.requests: true  
 percentiles:  
 http.server.requests: 0.5, 0.95, 0.99  
 sla:  
 http.server.requests: 100ms, 500ms, 1s, 2s  
  
# Jaeger链路追踪配置  
opentracing:  
 jaeger:  
 enabled: true  
 service-name: ${spring.application.name}  
 sampler:  
 type: probabilistic  
 param: 0.1 # 10%采样率  
 reporter:  
 log-spans: false  
 max-queue-size: 10000  
 flush-interval: 1000  
 sender:  
 endpoint: http://jaeger-collector:14268/api/traces  
  
# 自定义性能监控配置  
nsrs:  
 monitoring:  
 enabled: true  
 slow-query-threshold: 1000 # 慢查询阈值(ms)  
 batch-operation-threshold: 5000 # 批量操作阈值(ms)  
 cache-hit-rate-threshold: 0.8 # 缓存命中率阈值  
 error-rate-threshold: 0.001 # 错误率阈值

#### 6.2.2 性能分析脚本

# 性能分析脚本  
import requests  
import json  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns  
from datetime import datetime, timedelta  
import numpy as np  
  
class PerformanceAnalyzer:  
 def \_\_init\_\_(self, prometheus\_url, grafana\_url=None):  
 self.prometheus\_url = prometheus\_url  
 self.grafana\_url = grafana\_url  
   
 def query\_prometheus(self, query, start\_time=None, end\_time=None, step='1m'):  
 """查询Prometheus指标"""  
 if start\_time is None:  
 start\_time = datetime.now() - timedelta(hours=1)  
 if end\_time is None:  
 end\_time = datetime.now()  
   
 params = {  
 'query': query,  
 'start': start\_time.isoformat(),  
 'end': end\_time.isoformat(),  
 'step': step  
 }  
   
 response = requests.get(  
 f"{self.prometheus\_url}/api/v1/query\_range",  
 params=params  
 )  
   
 if response.status\_code == 200:  
 return response.json()['data']['result']  
 else:  
 raise Exception(f"Prometheus查询失败: {response.text}")  
   
 def analyze\_response\_time(self, hours=24):  
 """分析响应时间趋势"""  
 end\_time = datetime.now()  
 start\_time = end\_time - timedelta(hours=hours)  
   
 # 查询P50, P95, P99响应时间  
 queries = {  
 'P50': 'histogram\_quantile(0.5, rate(http\_request\_duration\_seconds\_bucket[5m]))',  
 'P95': 'histogram\_quantile(0.95, rate(http\_request\_duration\_seconds\_bucket[5m]))',  
 'P99': 'histogram\_quantile(0.99, rate(http\_request\_duration\_seconds\_bucket[5m]))'  
 }  
   
 results = {}  
 for name, query in queries.items():  
 data = self.query\_prometheus(query, start\_time, end\_time)  
 if data:  
 timestamps = [float(point[0]) for point in data[0]['values']]  
 values = [float(point[1]) \* 1000 for point in data[0]['values']] # 转换为ms  
 results[name] = pd.DataFrame({  
 'timestamp': pd.to\_datetime(timestamps, unit='s'),  
 'response\_time\_ms': values  
 })  
   
 # 绘制响应时间趋势图  
 plt.figure(figsize=(15, 8))  
 for name, df in results.items():  
 plt.plot(df['timestamp'], df['response\_time\_ms'], label=name, linewidth=2)  
   
 plt.title('NSRS系统响应时间趋势分析', fontsize=16)  
 plt.xlabel('时间', fontsize=12)  
 plt.ylabel('响应时间 (ms)', fontsize=12)  
 plt.legend()  
 plt.grid(True, alpha=0.3)  
 plt.xticks(rotation=45)  
 plt.tight\_layout()  
 plt.savefig('response\_time\_trend.png', dpi=300, bbox\_inches='tight')  
 plt.show()  
   
 return results  
   
 def analyze\_throughput(self, hours=24):  
 """分析吞吐量趋势"""  
 end\_time = datetime.now()  
 start\_time = end\_time - timedelta(hours=hours)  
   
 # 查询各接口的QPS  
 query = 'rate(http\_requests\_total[5m])'  
 data = self.query\_prometheus(query, start\_time, end\_time)  
   
 # 按endpoint分组分析  
 endpoint\_data = {}  
 for series in data:  
 endpoint = series['metric'].get('endpoint', 'unknown')  
 if endpoint not in endpoint\_data:  
 endpoint\_data[endpoint] = []  
   
 for timestamp, value in series['values']:  
 endpoint\_data[endpoint].append({  
 'timestamp': pd.to\_datetime(float(timestamp), unit='s'),  
 'qps': float(value)  
 })  
   
 # 绘制吞吐量趋势图  
 plt.figure(figsize=(15, 10))  
   
 for i, (endpoint, points) in enumerate(endpoint\_data.items()):  
 if len(points) > 0:  
 df = pd.DataFrame(points)  
 plt.subplot(len(endpoint\_data), 1, i+1)  
 plt.plot(df['timestamp'], df['qps'], label=endpoint, linewidth=2)  
 plt.title(f'{endpoint} - QPS趋势')  
 plt.ylabel('QPS')  
 plt.grid(True, alpha=0.3)  
 if i == len(endpoint\_data) - 1:  
 plt.xlabel('时间')  
 plt.xticks(rotation=45)  
   
 plt.tight\_layout()  
 plt.savefig('throughput\_trend.png', dpi=300, bbox\_inches='tight')  
 plt.show()  
   
 return endpoint\_data  
   
 def analyze\_error\_rate(self, hours=24):  
 """分析错误率趋势"""  
 end\_time = datetime.now()  
 start\_time = end\_time - timedelta(hours=hours)  
   
 # 查询总请求数和错误请求数  
 total\_query = 'rate(http\_requests\_total[5m])'  
 error\_query = 'rate(http\_requests\_total{status=~"4..|5.."}[5m])'  
   
 total\_data = self.query\_prometheus(total\_query, start\_time, end\_time)  
 error\_data = self.query\_prometheus(error\_query, start\_time, end\_time)  
   
 # 计算错误率  
 error\_rates = []  
 if total\_data and error\_data:  
 for i, (total\_point, error\_point) in enumerate(zip(total\_data[0]['values'], error\_data[0]['values'])):  
 timestamp = float(total\_point[0])  
 total\_rate = float(total\_point[1])  
 error\_rate = float(error\_point[1]) if error\_data else 0  
   
 error\_percentage = (error\_rate / total\_rate \* 100) if total\_rate > 0 else 0  
 error\_rates.append({  
 'timestamp': pd.to\_datetime(timestamp, unit='s'),  
 'error\_rate': error\_percentage  
 })  
   
 if error\_rates:  
 df = pd.DataFrame(error\_rates)  
   
 plt.figure(figsize=(15, 6))  
 plt.plot(df['timestamp'], df['error\_rate'], color='red', linewidth=2)  
 plt.title('NSRS系统错误率趋势分析', fontsize=16)  
 plt.xlabel('时间', fontsize=12)  
 plt.ylabel('错误率 (%)', fontsize=12)  
 plt.grid(True, alpha=0.3)  
 plt.xticks(rotation=45)  
 plt.tight\_layout()  
 plt.savefig('error\_rate\_trend.png', dpi=300, bbox\_inches='tight')  
 plt.show()  
   
 return error\_rates  
   
 def analyze\_resource\_usage(self, hours=24):  
 """分析资源使用率"""  
 end\_time = datetime.now()  
 start\_time = end\_time - timedelta(hours=hours)  
   
 # 查询CPU、内存、磁盘使用率  
 queries = {  
 'CPU使用率': 'rate(process\_cpu\_seconds\_total[5m]) \* 100',  
 'JVM内存使用率': 'jvm\_memory\_used\_bytes{area="heap"} / jvm\_memory\_max\_bytes{area="heap"} \* 100',  
 '磁盘使用率': '(1 - node\_filesystem\_avail\_bytes / node\_filesystem\_size\_bytes) \* 100'  
 }  
   
 plt.figure(figsize=(15, 12))  
   
 for i, (name, query) in enumerate(queries.items()):  
 data = self.query\_prometheus(query, start\_time, end\_time)  
 if data:  
 timestamps = [float(point[0]) for point in data[0]['values']]  
 values = [float(point[1]) for point in data[0]['values']]  
   
 plt.subplot(3, 1, i+1)  
 plt.plot(pd.to\_datetime(timestamps, unit='s'), values, linewidth=2)  
 plt.title(f'{name}趋势')  
 plt.ylabel('使用率 (%)')  
 plt.grid(True, alpha=0.3)  
 if i == 2:  
 plt.xlabel('时间')  
 plt.xticks(rotation=45)  
   
 plt.tight\_layout()  
 plt.savefig('resource\_usage\_trend.png', dpi=300, bbox\_inches='tight')  
 plt.show()  
   
 def generate\_performance\_report(self, hours=24):  
 """生成性能分析报告"""  
 print("=== NSRS系统性能分析报告 ===")  
 print(f"分析时间范围: 最近{hours}小时")  
 print(f"报告生成时间: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")  
 print("\n" + "="\*50)  
   
 # 分析各项指标  
 print("\n1. 响应时间分析...")  
 response\_data = self.analyze\_response\_time(hours)  
   
 print("\n2. 吞吐量分析...")  
 throughput\_data = self.analyze\_throughput(hours)  
   
 print("\n3. 错误率分析...")  
 error\_data = self.analyze\_error\_rate(hours)  
   
 print("\n4. 资源使用率分析...")  
 self.analyze\_resource\_usage(hours)  
   
 print("\n=== 性能分析报告生成完成 ===")  
 print("图表已保存到当前目录:")  
 print("- response\_time\_trend.png")  
 print("- throughput\_trend.png")  
 print("- error\_rate\_trend.png")  
 print("- resource\_usage\_trend.png")  
  
# 使用示例  
if \_\_name\_\_ == "\_\_main\_\_":  
 analyzer = PerformanceAnalyzer(  
 prometheus\_url="http://prometheus:9090",  
 grafana\_url="http://grafana:3000"  
 )  
   
 # 生成24小时性能分析报告  
 analyzer.generate\_performance\_report(hours=24)

## 7. 性能优化建议

### 7.1 应用层优化

#### 7.1.1 代码层面优化

// 批量操作优化示例  
@Service  
public class NumberResourceOptimizedService {  
   
 @Autowired  
 private NumberResourceMapper numberResourceMapper;  
   
 @Autowired  
 private RedisTemplate<String, Object> redisTemplate;  
   
 /\*\*  
 \* 优化的批量号码分配  
 \*/  
 @Transactional(rollbackFor = Exception.class)  
 public BatchAllocationResult batchAllocateNumbers(BatchAllocationRequest request) {  
 // 1. 参数验证  
 validateBatchRequest(request);  
   
 // 2. 预检查可用号码数量  
 long availableCount = numberResourceMapper.countAvailableNumbers(  
 request.getLevelId(), request.getOrgId());  
   
 if (availableCount < request.getQuantity()) {  
 throw new BusinessException("可用号码数量不足");  
 }  
   
 // 3. 批量查询可用号码（使用LIMIT优化）  
 List<NumberResource> availableNumbers = numberResourceMapper  
 .selectAvailableNumbersWithLimit(  
 request.getLevelId(),   
 request.getOrgId(),   
 request.getQuantity()  
 );  
   
 // 4. 批量更新状态（使用批量SQL）  
 List<Long> numberIds = availableNumbers.stream()  
 .map(NumberResource::getNumberId)  
 .collect(Collectors.toList());  
   
 int updatedCount = numberResourceMapper.batchUpdateStatus(  
 numberIds,   
 NumberStatusEnum.ALLOCATED.getCode(),  
 request.getOperatorUserId()  
 );  
   
 // 5. 批量插入操作日志（异步处理）  
 asyncLogBatchOperation(numberIds, request);  
   
 // 6. 更新缓存  
 updateNumberCache(availableNumbers);  
   
 return BatchAllocationResult.builder()  
 .allocatedNumbers(availableNumbers)  
 .allocatedCount(updatedCount)  
 .build();  
 }  
   
 /\*\*  
 \* 异步记录批量操作日志  
 \*/  
 @Async("taskExecutor")  
 public void asyncLogBatchOperation(List<Long> numberIds, BatchAllocationRequest request) {  
 List<NumberOperationLog> logs = numberIds.stream()  
 .map(numberId -> NumberOperationLog.builder()  
 .numberId(numberId)  
 .operationType(OperationTypeEnum.ALLOCATION.getCode())  
 .operatorUserId(request.getOperatorUserId())  
 .operationTime(LocalDateTime.now())  
 .remark(request.getRemark())  
 .build())  
 .collect(Collectors.toList());  
   
 // 批量插入日志  
 numberOperationLogMapper.batchInsert(logs);  
 }  
   
 /\*\*  
 \* 更新号码缓存  
 \*/  
 private void updateNumberCache(List<NumberResource> numbers) {  
 Pipeline pipeline = redisTemplate.executePipelined(new RedisCallback<Object>() {  
 @Override  
 public Object doInRedis(RedisConnection connection) throws DataAccessException {  
 for (NumberResource number : numbers) {  
 String key = "number:" + number.getNumber();  
 connection.setEx(  
 key.getBytes(),   
 300, // 5分钟TTL  
 JSON.toJSONString(number).getBytes()  
 );  
 }  
 return null;  
 }  
 });  
 }  
}

#### 7.1.2 数据库连接池优化

# application.yml - 数据库连接池优化配置  
spring:  
 datasource:  
 type: com.zaxxer.hikari.HikariDataSource  
 hikari:  
 # 连接池配置  
 minimum-idle: 10  
 maximum-pool-size: 50  
 connection-timeout: 30000  
 idle-timeout: 600000  
 max-lifetime: 1800000  
 leak-detection-threshold: 60000  
   
 # 性能优化配置  
 cache-prep-stmts: true  
 prep-stmt-cache-size: 250  
 prep-stmt-cache-sql-limit: 2048  
 use-server-prep-stmts: true  
 use-local-session-state: true  
 rewrite-batched-statements: true  
 cache-result-set-metadata: true  
 cache-server-configuration: true  
 elide-set-auto-commits: true  
 maintain-time-stats: false  
  
# MyBatis-Plus优化配置  
mybatis-plus:  
 configuration:  
 # 开启二级缓存  
 cache-enabled: true  
 # 延迟加载  
 lazy-loading-enabled: true  
 aggressive-lazy-loading: false  
 # 批量执行器  
 default-executor-type: batch  
 global-config:  
 db-config:  
 # 逻辑删除  
 logic-delete-field: deleted  
 logic-delete-value: 1  
 logic-not-delete-value: 0

### 7.2 数据库优化

#### 7.2.1 索引优化策略

-- 号码资源表索引优化  
-- 1. 复合索引优化查询  
CREATE INDEX idx\_number\_resource\_status\_level\_org   
ON number\_resource\_0 (status, level\_id, attributive\_org);  
  
CREATE INDEX idx\_number\_resource\_status\_level\_org   
ON number\_resource\_1 (status, level\_id, attributive\_org);  
  
CREATE INDEX idx\_number\_resource\_status\_level\_org   
ON number\_resource\_2 (status, level\_id, attributive\_org);  
  
-- 2. 覆盖索引优化  
CREATE INDEX idx\_number\_resource\_cover   
ON number\_resource\_0 (status, level\_id, number\_id, number, create\_time);  
  
-- 3. IMSI资源表索引优化  
CREATE INDEX idx\_imsi\_resource\_status\_group   
ON imsi\_resource\_0 (status, group\_id, imsi\_id);  
  
CREATE INDEX idx\_imsi\_resource\_status\_group   
ON imsi\_resource\_1 (status, group\_id, imsi\_id);  
  
-- 4. 绑定关系表索引优化  
CREATE INDEX idx\_binding\_number\_status   
ON number\_imsi\_binding (number, binding\_status, binding\_time);  
  
CREATE INDEX idx\_binding\_imsi\_status   
ON number\_imsi\_binding (imsi, binding\_status, binding\_time);  
  
-- 5. 操作日志表分区索引  
ALTER TABLE number\_operation\_log\_202401   
ADD INDEX idx\_operation\_time\_type (operation\_time, operation\_type);  
  
ALTER TABLE number\_operation\_log\_202402   
ADD INDEX idx\_operation\_time\_type (operation\_time, operation\_type);

#### 7.2.2 查询优化

-- 优化前的查询（性能较差）  
SELECT \* FROM number\_resource   
WHERE status = 1   
AND level\_id = 2   
AND attributive\_org = 'ORG001'  
ORDER BY create\_time DESC   
LIMIT 100;  
  
-- 优化后的查询（使用覆盖索引）  
SELECT number\_id, number, status, level\_id, create\_time   
FROM number\_resource   
WHERE status = 1   
AND level\_id = 2   
AND attributive\_org = 'ORG001'  
ORDER BY create\_time DESC   
LIMIT 100;  
  
-- 批量查询优化  
-- 优化前（N+1查询问题）  
SELECT \* FROM number\_resource WHERE number\_id = ?;  
-- 对每个number\_id执行一次查询  
  
-- 优化后（批量查询）  
SELECT \* FROM number\_resource   
WHERE number\_id IN (?, ?, ?, ...);  
  
-- 分页查询优化  
-- 优化前（深分页性能差）  
SELECT \* FROM number\_resource   
WHERE status = 1   
ORDER BY number\_id   
LIMIT 100000, 20;  
  
-- 优化后（使用游标分页）  
SELECT \* FROM number\_resource   
WHERE status = 1   
AND number\_id > 100000   
ORDER BY number\_id   
LIMIT 20;

### 7.3 缓存优化

#### 7.3.1 多级缓存架构

@Component  
public class MultiLevelCacheManager {  
   
 @Autowired  
 private RedisTemplate<String, Object> redisTemplate;  
   
 // L1缓存：本地缓存（Caffeine）  
 private final Cache<String, Object> localCache = Caffeine.newBuilder()  
 .maximumSize(10000)  
 .expireAfterWrite(5, TimeUnit.MINUTES)  
 .recordStats()  
 .build();  
   
 // L2缓存：Redis分布式缓存  
   
 /\*\*  
 \* 获取缓存数据  
 \*/  
 public <T> T get(String key, Class<T> type, Supplier<T> dataLoader) {  
 // 1. 尝试从L1缓存获取  
 Object cached = localCache.getIfPresent(key);  
 if (cached != null) {  
 return type.cast(cached);  
 }  
   
 // 2. 尝试从L2缓存获取  
 String redisValue = (String) redisTemplate.opsForValue().get(key);  
 if (StringUtils.hasText(redisValue)) {  
 T data = JSON.parseObject(redisValue, type);  
 // 回填L1缓存  
 localCache.put(key, data);  
 return data;  
 }  
   
 // 3. 从数据源加载  
 T data = dataLoader.get();  
 if (data != null) {  
 // 写入L2缓存  
 redisTemplate.opsForValue().set(key, JSON.toJSONString(data),   
 Duration.ofMinutes(10));  
 // 写入L1缓存  
 localCache.put(key, data);  
 }  
   
 return data;  
 }  
   
 /\*\*  
 \* 删除缓存  
 \*/  
 public void evict(String key) {  
 localCache.invalidate(key);  
 redisTemplate.delete(key);  
 }  
   
 /\*\*  
 \* 批量删除缓存  
 \*/  
 public void evictBatch(Collection<String> keys) {  
 keys.forEach(localCache::invalidate);  
 redisTemplate.delete(keys);  
 }  
   
 /\*\*  
 \* 获取缓存统计信息  
 \*/  
 public CacheStats getLocalCacheStats() {  
 return localCache.stats();  
 }  
}

#### 7.3.2 缓存预热策略

@Component  
public class CacheWarmupService {  
   
 @Autowired  
 private NumberResourceService numberResourceService;  
   
 @Autowired  
 private MultiLevelCacheManager cacheManager;  
   
 @EventListener(ApplicationReadyEvent.class)  
 public void warmupCache() {  
 log.info("开始缓存预热...");  
   
 // 预热热点号码数据  
 warmupHotNumbers();  
   
 // 预热配置数据  
 warmupConfigData();  
   
 // 预热统计数据  
 warmupStatisticsData();  
   
 log.info("缓存预热完成");  
 }  
   
 private void warmupHotNumbers() {  
 // 查询最近访问频率高的号码  
 List<String> hotNumbers = numberResourceService.getHotNumbers(1000);  
   
 hotNumbers.parallelStream().forEach(number -> {  
 try {  
 String cacheKey = "number:" + number;  
 NumberResource numberResource = numberResourceService.getByNumber(number);  
 if (numberResource != null) {  
 cacheManager.put(cacheKey, numberResource);  
 }  
 } catch (Exception e) {  
 log.warn("预热号码缓存失败: {}", number, e);  
 }  
 });  
 }  
   
 private void warmupConfigData() {  
 // 预热系统配置  
 List<SystemConfig> configs = systemConfigService.getAllConfigs();  
 configs.forEach(config -> {  
 String cacheKey = "config:" + config.getConfigKey();  
 cacheManager.put(cacheKey, config.getConfigValue());  
 });  
 }  
   
 private void warmupStatisticsData() {  
 // 预热统计数据  
 Map<String, Object> stats = statisticsService.getDashboardStats();  
 cacheManager.put("dashboard:stats", stats);  
 }  
}

## 8. 性能测试执行计划

### 8.1 测试阶段规划

#### 8.1.1 第一阶段：基准测试（1周）

phase\_1\_baseline\_testing:  
 duration: 1周  
 objectives:  
 - 建立性能基准线  
 - 验证测试环境  
 - 确认测试工具  
   
 test\_scenarios:  
 day\_1\_2:  
 - 环境搭建和配置  
 - 测试数据准备  
 - 监控工具部署  
   
 day\_3\_4:  
 - 单接口性能测试  
 - 数据库基准测试  
 - 缓存性能测试  
   
 day\_5\_7:  
 - 基准测试报告  
 - 性能瓶颈初步分析  
 - 测试计划调整  
   
 deliverables:  
 - 性能基准报告  
 - 测试环境文档  
 - 监控仪表板

#### 8.1.2 第二阶段：负载测试（2周）

phase\_2\_load\_testing:  
 duration: 2周  
 objectives:  
 - 验证系统在预期负载下的性能  
 - 识别性能瓶颈  
 - 优化系统配置  
   
 week\_1:  
 - 号码资源管理负载测试  
 - SIM卡管理负载测试  
 - 绑定操作负载测试  
   
 week\_2:  
 - 批量操作负载测试  
 - 混合场景负载测试  
 - 性能优化实施  
   
 test\_matrix:  
 concurrent\_users: [100, 300, 500, 800, 1000]  
 test\_duration: [30min, 1hour, 2hours]  
 scenarios:  
 - 号码查询: 40%  
 - 号码分配: 25%  
 - SIM卡操作: 20%  
 - 绑定操作: 15%

#### 8.1.3 第三阶段：压力测试（1周）

phase\_3\_stress\_testing:  
 duration: 1周  
 objectives:  
 - 确定系统最大承载能力  
 - 验证系统在极限负载下的表现  
 - 测试系统恢复能力  
   
 test\_approach:  
 - 逐步增加负载直到系统性能显著下降  
 - 测试系统在资源耗尽情况下的行为  
 - 验证系统的故障恢复机制  
   
 stress\_scenarios:  
 - CPU密集型操作压力测试  
 - 内存密集型操作压力测试  
 - 数据库连接池压力测试  
 - 网络带宽压力测试

### 8.2 测试执行流程

#### 8.2.1 测试前准备

#!/bin/bash  
# 性能测试环境准备脚本  
  
echo "=== NSRS性能测试环境准备 ==="  
  
# 1. 检查测试环境  
echo "1. 检查测试环境状态..."  
curl -f http://nsrs-app:8088/actuator/health || exit 1  
curl -f http://mysql-master:3306 || exit 1  
redis-cli -h redis-cluster -p 7000 ping || exit 1  
  
# 2. 清理历史数据  
echo "2. 清理历史测试数据..."  
mysql -h mysql-master -u nsrs\_user -p$DB\_PASSWORD nsrs << EOF  
DELETE FROM number\_operation\_log WHERE remark LIKE '%Performance Test%';  
DELETE FROM number\_imsi\_binding WHERE remark LIKE '%Performance Test%';  
UPDATE number\_resource SET status = 1 WHERE remark LIKE '%Performance Test%';  
UPDATE sim\_card SET status = 1 WHERE remark LIKE '%Performance Test%';  
UPDATE imsi\_resource SET status = 1 WHERE remark LIKE '%Performance Test%';  
EOF  
  
# 3. 重置Redis缓存  
echo "3. 清理Redis缓存..."  
redis-cli -h redis-cluster -p 7000 FLUSHALL  
  
# 4. 重启应用服务  
echo "4. 重启应用服务..."  
kubectl rollout restart deployment/nsrs-app  
kubectl rollout status deployment/nsrs-app  
  
# 5. 预热系统  
echo "5. 系统预热..."  
for i in {1..100}; do  
 curl -s "http://nsrs-app:8088/api/numbers/search?pageSize=10&pageNum=1" > /dev/null  
 curl -s "http://nsrs-app:8088/api/sim-cards/search?pageSize=10&pageNum=1" > /dev/null  
done  
  
# 6. 验证监控系统  
echo "6. 验证监控系统..."  
curl -f http://prometheus:9090/-/healthy || exit 1  
curl -f http://grafana:3000/api/health || exit 1  
  
echo "测试环境准备完成！"

#### 8.2.2 测试执行脚本

#!/usr/bin/env python3  
# 性能测试执行脚本  
  
import subprocess  
import time  
import json  
import os  
from datetime import datetime  
  
class PerformanceTestExecutor:  
 def \_\_init\_\_(self, config\_file='test\_config.json'):  
 with open(config\_file, 'r') as f:  
 self.config = json.load(f)  
   
 self.results\_dir = f"results\_{datetime.now().strftime('%Y%m%d\_%H%M%S')}"  
 os.makedirs(self.results\_dir, exist\_ok=True)  
   
 def execute\_jmeter\_test(self, test\_plan, test\_name):  
 """执行JMeter测试"""  
 print(f"执行测试: {test\_name}")  
   
 cmd = [  
 'jmeter',  
 '-n', # 非GUI模式  
 '-t', test\_plan, # 测试计划文件  
 '-l', f"{self.results\_dir}/{test\_name}\_results.jtl", # 结果文件  
 '-e', # 生成HTML报告  
 '-o', f"{self.results\_dir}/{test\_name}\_report", # 报告目录  
 '-J', f"threads={self.config['concurrent\_users']}",  
 '-J', f"duration={self.config['test\_duration']}",  
 '-J', f"rampup={self.config['ramp\_up\_time']}"  
 ]  
   
 start\_time = time.time()  
 result = subprocess.run(cmd, capture\_output=True, text=True)  
 end\_time = time.time()  
   
 if result.returncode == 0:  
 print(f"测试 {test\_name} 完成，耗时: {end\_time - start\_time:.2f}秒")  
 return True  
 else:  
 print(f"测试 {test\_name} 失败: {result.stderr}")  
 return False  
   
 def monitor\_system\_resources(self, duration):  
 """监控系统资源"""  
 print(f"开始监控系统资源，持续{duration}秒...")  
   
 # 启动资源监控脚本  
 monitor\_cmd = [  
 'python3', 'monitor\_resources.py',  
 '--duration', str(duration),  
 '--output', f"{self.results\_dir}/system\_resources.json"  
 ]  
   
 subprocess.Popen(monitor\_cmd)  
   
 def run\_test\_suite(self):  
 """运行完整测试套件"""  
 print("=== 开始NSRS性能测试套件 ===")  
 print(f"结果目录: {self.results\_dir}")  
   
 test\_plans = [  
 ('number\_query\_test.jmx', '号码查询性能测试'),  
 ('number\_allocation\_test.jmx', '号码分配性能测试'),  
 ('sim\_card\_test.jmx', 'SIM卡管理性能测试'),  
 ('binding\_test.jmx', '绑定操作性能测试'),  
 ('batch\_operation\_test.jmx', '批量操作性能测试'),  
 ('mixed\_scenario\_test.jmx', '混合场景性能测试')  
 ]  
   
 for test\_plan, test\_name in test\_plans:  
 print(f"\n--- {test\_name} ---")  
   
 # 启动系统监控  
 self.monitor\_system\_resources(self.config['test\_duration'] + 60)  
   
 # 执行测试  
 success = self.execute\_jmeter\_test(test\_plan, test\_name.replace(' ', '\_'))  
   
 if success:  
 print(f"✓ {test\_name} 执行成功")  
 else:  
 print(f"✗ {test\_name} 执行失败")  
   
 # 测试间隔  
 print(f"等待{self.config['test\_interval']}秒后执行下一个测试...")  
 time.sleep(self.config['test\_interval'])  
   
 print("\n=== 性能测试套件执行完成 ===")  
 self.generate\_summary\_report()  
   
 def generate\_summary\_report(self):  
 """生成汇总报告"""  
 print("生成汇总报告...")  
   
 summary\_cmd = [  
 'python3', 'generate\_summary.py',  
 '--results-dir', self.results\_dir,  
 '--output', f"{self.results\_dir}/summary\_report.html"  
 ]  
   
 subprocess.run(summary\_cmd)  
 print(f"汇总报告已生成: {self.results\_dir}/summary\_report.html")  
  
# 执行测试  
if \_\_name\_\_ == "\_\_main\_\_":  
 executor = PerformanceTestExecutor()  
 executor.run\_test\_suite()

## 9. 性能测试报告模板

### 9.1 测试结果评估标准

performance\_criteria:  
 response\_time:  
 excellent: "< 200ms"  
 good: "200ms - 500ms"  
 acceptable: "500ms - 1000ms"  
 poor: "> 1000ms"  
   
 throughput:  
 number\_query:  
 excellent: "> 2000 TPS"  
 good: "1000 - 2000 TPS"  
 acceptable: "500 - 1000 TPS"  
 poor: "< 500 TPS"  
   
 number\_allocation:  
 excellent: "> 500 TPS"  
 good: "200 - 500 TPS"  
 acceptable: "100 - 200 TPS"  
 poor: "< 100 TPS"  
   
 error\_rate:  
 excellent: "< 0.01%"  
 good: "0.01% - 0.1%"  
 acceptable: "0.1% - 0.5%"  
 poor: "> 0.5%"  
   
 resource\_usage:  
 cpu:  
 excellent: "< 60%"  
 good: "60% - 75%"  
 acceptable: "75% - 85%"  
 poor: "> 85%"  
   
 memory:  
 excellent: "< 70%"  
 good: "70% - 80%"  
 acceptable: "80% - 90%"  
 poor: "> 90%"

### 9.2 报告生成脚本

# 性能测试报告生成脚本  
import json  
import pandas as pd  
from jinja2 import Template  
from datetime import datetime  
import matplotlib.pyplot as plt  
import seaborn as sns  
  
class PerformanceReportGenerator:  
 def \_\_init\_\_(self, results\_dir):  
 self.results\_dir = results\_dir  
 self.report\_data = {}  
   
 def parse\_jmeter\_results(self, jtl\_file):  
 """解析JMeter结果文件"""  
 df = pd.read\_csv(jtl\_file)  
   
 # 计算统计指标  
 stats = {  
 'total\_requests': len(df),  
 'successful\_requests': len(df[df['success'] == True]),  
 'failed\_requests': len(df[df['success'] == False]),  
 'error\_rate': len(df[df['success'] == False]) / len(df) \* 100,  
 'avg\_response\_time': df['elapsed'].mean(),  
 'min\_response\_time': df['elapsed'].min(),  
 'max\_response\_time': df['elapsed'].max(),  
 'p50\_response\_time': df['elapsed'].quantile(0.5),  
 'p95\_response\_time': df['elapsed'].quantile(0.95),  
 'p99\_response\_time': df['elapsed'].quantile(0.99),  
 'throughput': len(df) / (df['timeStamp'].max() - df['timeStamp'].min()) \* 1000  
 }  
   
 return stats  
   
 def generate\_charts(self, test\_name, df):  
 """生成性能图表"""  
 fig, axes = plt.subplots(2, 2, figsize=(15, 10))  
 fig.suptitle(f'{test\_name} - 性能分析图表', fontsize=16)  
   
 # 响应时间分布  
 axes[0, 0].hist(df['elapsed'], bins=50, alpha=0.7)  
 axes[0, 0].set\_title('响应时间分布')  
 axes[0, 0].set\_xlabel('响应时间 (ms)')  
 axes[0, 0].set\_ylabel('请求数量')  
   
 # 响应时间趋势  
 df['timestamp'] = pd.to\_datetime(df['timeStamp'], unit='ms')  
 df\_resampled = df.set\_index('timestamp').resample('1min')['elapsed'].mean()  
 axes[0, 1].plot(df\_resampled.index, df\_resampled.values)  
 axes[0, 1].set\_title('响应时间趋势')  
 axes[0, 1].set\_xlabel('时间')  
 axes[0, 1].set\_ylabel('平均响应时间 (ms)')  
 axes[0, 1].tick\_params(axis='x', rotation=45)  
   
 # 吞吐量趋势  
 throughput = df.set\_index('timestamp').resample('1min').size()  
 axes[1, 0].plot(throughput.index, throughput.values)  
 axes[1, 0].set\_title('吞吐量趋势')  
 axes[1, 0].set\_xlabel('时间')  
 axes[1, 0].set\_ylabel('TPS')  
 axes[1, 0].tick\_params(axis='x', rotation=45)  
   
 # 错误率趋势  
 error\_rate = df.set\_index('timestamp').resample('1min')['success'].apply(lambda x: (1 - x.mean()) \* 100)  
 axes[1, 1].plot(error\_rate.index, error\_rate.values, color='red')  
 axes[1, 1].set\_title('错误率趋势')  
 axes[1, 1].set\_xlabel('时间')  
 axes[1, 1].set\_ylabel('错误率 (%)')  
 axes[1, 1].tick\_params(axis='x', rotation=45)  
   
 plt.tight\_layout()  
 chart\_file = f"{self.results\_dir}/{test\_name}\_charts.png"  
 plt.savefig(chart\_file, dpi=300, bbox\_inches='tight')  
 plt.close()  
   
 return chart\_file  
   
 def generate\_html\_report(self):  
 """生成HTML报告"""  
 template\_str = """  
<!DOCTYPE html>  
<html>  
<head>  
 <title>NSRS系统性能测试报告</title>  
 <meta charset="UTF-8">  
 <style>  
 body { font-family: Arial, sans-serif; margin: 20px; }  
 .header { background-color: #f0f0f0; padding: 20px; border-radius: 5px; }  
 .summary { margin: 20px 0; }  
 .test-result { margin: 20px 0; border: 1px solid #ddd; padding: 15px; border-radius: 5px; }  
 .metrics { display: flex; flex-wrap: wrap; gap: 20px; }  
 .metric { background-color: #f9f9f9; padding: 10px; border-radius: 3px; min-width: 200px; }  
 .excellent { background-color: #d4edda; }  
 .good { background-color: #d1ecf1; }  
 .acceptable { background-color: #fff3cd; }  
 .poor { background-color: #f8d7da; }  
 table { width: 100%; border-collapse: collapse; margin: 10px 0; }  
 th, td { border: 1px solid #ddd; padding: 8px; text-align: left; }  
 th { background-color: #f2f2f2; }  
 .chart { text-align: center; margin: 20px 0; }  
 </style>  
</head>  
<body>  
 <div class="header">  
 <h1>NSRS号卡资源管理系统性能测试报告</h1>  
 <p><strong>测试时间:</strong> {{ test\_date }}</p>  
 <p><strong>测试环境:</strong> {{ test\_environment }}</p>  
 <p><strong>测试版本:</strong> {{ system\_version }}</p>  
 </div>  
   
 <div class="summary">  
 <h2>测试概要</h2>  
 <div class="metrics">  
 <div class="metric {{ overall\_rating }}">  
 <h4>总体评级</h4>  
 <p>{{ overall\_rating.upper() }}</p>  
 </div>  
 <div class="metric">  
 <h4>测试场景数</h4>  
 <p>{{ total\_scenarios }}</p>  
 </div>  
 <div class="metric">  
 <h4>总请求数</h4>  
 <p>{{ total\_requests }}</p>  
 </div>  
 <div class="metric">  
 <h4>平均成功率</h4>  
 <p>{{ avg\_success\_rate }}%</p>  
 </div>  
 </div>  
 </div>  
   
 {% for test in test\_results %}  
 <div class="test-result">  
 <h3>{{ test.name }}</h3>  
 <div class="metrics">  
 <div class="metric {{ test.response\_time\_rating }}">  
 <h4>平均响应时间</h4>  
 <p>{{ test.avg\_response\_time }}ms</p>  
 </div>  
 <div class="metric {{ test.throughput\_rating }}">  
 <h4>吞吐量</h4>  
 <p>{{ test.throughput }} TPS</p>  
 </div>  
 <div class="metric {{ test.error\_rate\_rating }}">  
 <h4>错误率</h4>  
 <p>{{ test.error\_rate }}%</p>  
 </div>  
 </div>  
   
 <table>  
 <tr>  
 <th>指标</th>  
 <th>值</th>  
 <th>评级</th>  
 </tr>  
 <tr>  
 <td>总请求数</td>  
 <td>{{ test.total\_requests }}</td>  
 <td>-</td>  
 </tr>  
 <tr>  
 <td>成功请求数</td>  
 <td>{{ test.successful\_requests }}</td>  
 <td>-</td>  
 </tr>  
 <tr>  
 <td>P50响应时间</td>  
 <td>{{ test.p50\_response\_time }}ms</td>  
 <td>{{ test.p50\_rating }}</td>  
 </tr>  
 <tr>  
 <td>P95响应时间</td>  
 <td>{{ test.p95\_response\_time }}ms</td>  
 <td>{{ test.p95\_rating }}</td>  
 </tr>  
 <tr>  
 <td>P99响应时间</td>  
 <td>{{ test.p99\_response\_time }}ms</td>  
 <td>{{ test.p99\_rating }}</td>  
 </tr>  
 </table>  
   
 {% if test.chart\_file %}  
 <div class="chart">  
 <img src="{{ test.chart\_file }}" alt="{{ test.name }} 性能图表" style="max-width: 100%;">  
 </div>  
 {% endif %}  
 </div>  
 {% endfor %}  
   
 <div class="summary">  
 <h2>性能优化建议</h2>  
 <ul>  
 {% for recommendation in recommendations %}  
 <li>{{ recommendation }}</li>  
 {% endfor %}  
 </ul>  
 </div>  
</body>  
</html>  
 """  
   
 template = Template(template\_str)  
 html\_content = template.render(\*\*self.report\_data)  
   
 report\_file = f"{self.results\_dir}/performance\_report.html"  
 with open(report\_file, 'w', encoding='utf-8') as f:  
 f.write(html\_content)  
   
 return report\_file  
  
# 使用示例  
if \_\_name\_\_ == "\_\_main\_\_":  
 generator = PerformanceReportGenerator("results\_20241201\_143000")  
 report\_file = generator.generate\_html\_report()  
 print(f"性能测试报告已生成: {report\_file}")

## 10. 总结

本性能测试方案为NSRS号卡资源管理系统提供了全面的性能测试框架，涵盖了从测试策略制定到结果分析的完整流程。通过系统化的性能测试，可以：

1. **建立性能基准**: 为系统性能建立可量化的基准指标
2. **识别性能瓶颈**: 及时发现系统的性能瓶颈和优化点
3. **验证系统容量**: 确保系统能够满足业务增长需求
4. **指导优化方向**: 为系统优化提供数据支撑和方向指导
5. **保障服务质量**: 确保系统在生产环境中的稳定性和可靠性

建议在系统上线前、重大版本发布前以及定期维护时执行本测试方案，持续监控和优化系统性能，确保NSRS系统能够为电信运营商提供高效、稳定的号卡资源管理服务。