

Java 无状态系统的优化实践

赵思焰

爱迪德技术(北京)有限公司, 北京 100125

摘 要: Java 无状态系统越来越成为系统 API 平台间无缝连接的主要方式, 例如跨平台 REST API、SOAP 调用等; 但平台间的安全性和性能根据业务的要求都有着一定的标准, 事无巨细, 对某一复杂逻辑重要单 API 的优化需要经过从测试分析到检查瓶颈, 到优化方法一系列过程, 从而反推到设计问题, 例如架构的选择和表结构设计等。

关键词: Spring Security; Sardine; Java 无状态系统; 性能优化

中图分类号: TP311

文献标识码: A

DOI: 10.19414/j.cnki.1005-1228.2017.01.010

Java Stateless System Performance Optimizations

ZHAO Si-yan

Irdeto Technology(Beijing) Co Ltd. Beijing 100125, China

Abstract: Java stateless systems becomes more and more main connection as a kind of API seamless access to other platforms such as REST API, SOAP invoke etc. But security and performance need to have a high level according to business requirements. From details for a complex logic business API optimization, we need to experience a process from analysis by testing, checking and finding bottle-neck reasons and finding optimization strategies so can re-think design problems such as framework selection and table design etc.

Key words: spring Security; sardine; Java stateless system; performance optimization

随着业务逻辑的多层次要求, 跨平台跨系统间的架构及耦合实现变得越来越多, 一般采用无状态 Java 系统作为 portal 比较常见, 所以当系统瓶颈出现, 就需要通过业务逻辑, 系统构成, 程序逻辑, 语言机制, 设计

3) 处理业务时, 需要对 DB, 其他平台如 .net soap, 和 webdav server 同时访问;

4) 问题, 由于在 api 上连接的模块比较多, 且每次又有安全校验, 所以开始单个 api 的吞吐量只在 20 左

右, 而业务需求需要提升到 200, 怎样才能达到要求, 首先还是从应用的技术模块分别入手, 并且利用 Jmeter 等测试工具结合找到瓶颈调优。

1 系统结构

通常由于业务的需要, 经常采用跨平台 API 访问所以无状态访问 API, 这就涉及到多线程, 多系统间高性能并发访问的要求问题, 如图 1 所示, 目前的系统架构为:

1) Webserver 采用并发连接访问和丰富的组件化的 Jetty;

2) 由于跨平台业务, 所以在安全性上采用不同策略, 例如 openid 或 oauth2 协议, 或 basic-authentication 对于设定低 level api;



图 1 系统结构

2 技术应用模块

如图 2 所示, 为各应用模块之间的简要连接。

收稿日期: 2016-10-20

作者简介: 赵思焰 (1973-), 男, 北京人, 首席工程师, 主要研究方向: 大数据及 Java 在大型系统中性能提升研究。

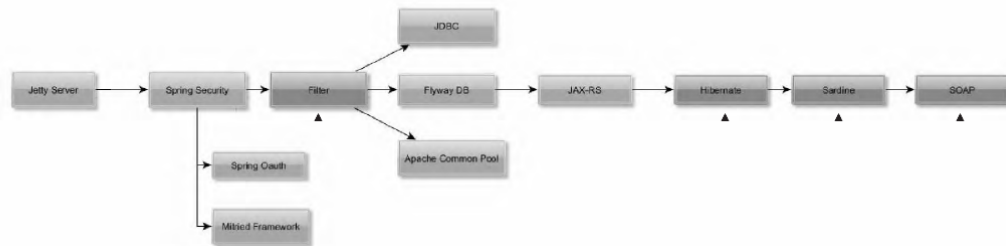


图 2 各系统实现模块构成

- (1) 利用 spring security 可以灵活注入各种安全机制, Mitted 框架可支持 openid 和 oauth2 协议, 这里仅对 basic authentication 说明。
- (2) Flyway DB 用于各数据库 Migration, 对性能无

利用 VisualVM 诊断需要注意的地方

- (1) 利用 monitor 窗口查看在并发量增大时 CPU 和 Heap 的变化, 因为本系统现象为 CPU 长时间增高不下降。需要对其进一步采样

影响, 并已设置连接池属性。

- (3) 对于不同平台调用, sardine 用于 Webdav 客户端, 经研究未能很好实现多线程

- (2) 利用 Sampler 窗口可以观察到哪些类所消耗时间最大, 对其在程序中进行跟踪分析, 例如

如图 3 所示, 可以判断 JDBC 对 SQL 进行处理和加密过程消耗时间较大。

3 JMeter 和 VisualVM 配合性能诊断及调优

Hot Spots - Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
com.microsoft.sqlserver.jdbc.TDSChannel.read()	42.57%	42,576 ms (33.4%)	42,576 ms	42,576 ms	42,576 ms
org.springframework.security.oauth2.jwt.JwtKey()	28.67%	28,676 ms (22.5%)	28,676 ms	28,676 ms	28,676 ms
org.springframework.jdbc.datasource.DataSourceUtils.getConnection()	24.63%	24,630 ms (19.4%)	24,630 ms	24,630 ms	24,630 ms
org.hibernate.engine.internal.EntityContext.addInitiator()	8.61%	8,616 ms (6.8%)	8,616 ms	8,616 ms	8,616 ms
org.springframework.util.concurrent.AbstractFuture.poll()	8.01%	8,011 ms (6.3%)	8,011 ms	8,011 ms	8,011 ms
org.apache.http.impl.client.HttpClientConnManager.getConnection()	4.23%	4,233 ms (3.3%)	4,233 ms	4,233 ms	4,233 ms
org.apache.http.impl.client.HttpClientConnManager.getConnection()	2.89%	2,895 ms (2.3%)	2,895 ms	2,895 ms	2,895 ms
org.springframework.util.concurrent.AbstractFuture.await()	951 ms (0.7%)	951 ms (0.7%)	951 ms	951 ms	951 ms
org.hibernate.engine.internal.SessionFactoryImpl.initializeEntity()	446 ms (0.4%)	446 ms	12,550 ms	12,550 ms	12,550 ms
org.apache.http.conn.socket.PlainConnectionSocketFactory.connectSocket()	412 ms (0.3%)	412 ms	412 ms	412 ms	412 ms
com.microsoft.sqlserver.jdbc.TDSChannel.write()	402 ms (0.3%)	402 ms	402 ms	402 ms	402 ms
ch.qos.logback.classic.Logger.isDebugEnabled()	340 ms (0.3%)	340 ms	340 ms	340 ms	340 ms
org.hibernate.type.AbstractStandardBasicType.isEqual()	322 ms (0.3%)	322 ms	322 ms	322 ms	322 ms
org.hibernate.property.DirectPropertyAccessor\$DirectGetter.get()	213 ms (0.2%)	213 ms	213 ms	213 ms	213 ms
com.github.sardine.util.SardineUtil.unmarshal()	297 ms (0.2%)	297 ms	601 ms	601 ms	601 ms
org.hibernate.tuple.entity.AbstractEntityTuplizer.setPropertyValues()	291 ms (0.2%)	291 ms	291 ms	291 ms	291 ms
org.hibernate.engine.internal.SessionFactoryImpl.postLoad()	246 ms (0.2%)	246 ms	246 ms	246 ms	246 ms
org.hibernate.type.AbstractStandardBasicType.decrypt()	211 ms (0.2%)	211 ms	409 ms	409 ms	409 ms
org.springframework.common.datasource.DataSourceUtils.getConnection()	201 ms (0.2%)	201 ms	201 ms	201 ms	201 ms
org.springframework.web.util.WebAppClassLoader.isTest()	190 ms (0.2%)	190 ms	190 ms	190 ms	190 ms
ch.qos.logback.classic.Logger.getLogger()	197 ms (0.2%)	197 ms	197 ms	197 ms	197 ms
org.hibernate.type.AbstractStandardBasicType.getStabilityPlan()	197 ms (0.2%)	197 ms	197 ms	197 ms	197 ms
org.hibernate.engine.jdbc.internal.JdbcCoordinatorImpl.register()	192 ms (0.2%)	192 ms	192 ms	192 ms	192 ms
org.hibernate.engine.internal.EntityContext.clear()	176 ms (0.1%)	176 ms	176 ms	176 ms	176 ms
com.microsoft.sqlserver.jdbc.SQLServerConnection.beginTransaction()	176 ms (0.1%)	176 ms	176 ms	176 ms	176 ms
ch.qos.logback.core.joran.spi.ConsoleTarget.write()	151 ms (0.1%)	151 ms	151 ms	151 ms	151 ms
com.microsoft.sqlserver.jdbc.TDSReader.readBytes()	151 ms (0.1%)	151 ms	8,241 ms	8,241 ms	8,241 ms
org.springframework.security.authentication.TrustResolverImpl.isAnonymous()	128 ms (0.1%)	128 ms	128 ms	128 ms	128 ms
org.apache.http.impl.client.HttpClientConnManager.getConnection()	103 ms (0.1%)	103 ms	103 ms	103 ms	103 ms
org.springframework.jdbc.datasource.DataSourceUtils.getConnection()	103 ms (0.1%)	103 ms	103 ms	103 ms	103 ms
org.springframework.jdbc.datasource.DataSourceUtils.getConnection()	102 ms (0.1%)	102 ms	102 ms	102 ms	102 ms
org.springframework.jdbc.datasource.DataSourceUtils.getConnection()	102 ms (0.1%)	102 ms	102 ms	102 ms	102 ms
org.hibernate.internal.util.comparison.EqualityHelper.equals()	101 ms (0.1%)	101 ms	101 ms	101 ms	101 ms
org.springframework.jdbc.datasource.DataSourceUtils.getConnection()	100 ms (0.1%)	100 ms	100 ms	100 ms	100 ms

图 3 VisualVM 的测试分析结果

- (3) 在优化后, 往往瓶颈都是相对的, 例如解决了 JDBC 问题, 则另一个问题如加密算法问题突出, 可继续对应 Sampler 窗口观察涉及哪些类, 以求进一步优化跟踪。

运行 JMeter 测试吞吐量注意:

- (1) 由于程序中运用大量注入机制, 所以建议第一次手动运行, 以确保正常初始化

- (2) User 线程数最好有少增多, Ramp-up period 也需要适当调节, 便于观察系统中的负载峰值和配合

VM 查看出错原因

- (3) JMeter 也是利用多线程机制测试吞吐量, 根据错误量进行统计, 所以需要稳定一段时间才可看到最大吞吐量

- (4) 当超过最大承载量时, 利用 View ResultTree 查看错误日志。

4 模块优化策略

图 2 中带“▲”标记的为需要优化模块

(1) 现象: 从 JMeter 中不断发出错误处理 Bad Request

分析: 错误日志中并没有执行到 REST API 逻辑, 而是在 Servlet 时报错, 因为系统中所有 API 需要认证, 所以锁定在 Spring Security 处理认证时发生瓶颈解决: 我们在 Filter 模块中加入 Cache, 将反复认证的

```
<!-- 配置 CacheFilter 举例?
<http-authentication-manager-ref="authenticationManager" create-session="never" entry-point-ref="entryPoint"/>
<!-- 设置 cacheFilter 作为前端 Filter?
<custom-filter ref="cacheFilter" position="BasicAuthFilter" />
<csrf-disabled="true"/>
<port-mappings>
<port-mapping http="8080" https="443"/>
</port-mappings>
</http>
```

<!--配置 Basic 秘钥-->

```
<authentication-manager id="authenticationManager">
<authentication-provider>
<password-encoder ref="bcryptEncoder"/>
<user-service>
<username="user" password="..."
authorities="ROLE_USER"/>
</user-service>
</authentication-provider>
</authentication-manager>
```

因为 CacheFilter 继承 org.springframework.security.web.authentication.www.BasicAuthenticationFilter 所以需要 override 的 doFilterInternal 方法

```
@Override
protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain chain) throws IOException, ServletException {
```

```
String header = request.getHeader("Authorization");//得到header
```

```
if(header != null && header.startsWith("Basic ")) {
try {
```

```
if (cached.contains(header)) { //static ArrayList<String>
cached = new ArrayList<>();
```

```
... //处理出 user,password
UsernamePasswordAuthenticationToken authRequest = new
UsernamePasswordAuthenticationToken(user, password);
Authentication authResult = this.getAuthenticationManager().authenticate(authRequest);
//这里用 spring security 认证方法
if(authResult.isAuthenticated()) {
cached.add(header); //cache 住认证
}
else
throw new BaseException ("Invalid authentication token or wrong username, password");
} catch (Exception ex) {
...
}
...
chain.doFilter(request, response);
}
```

(2) 现象: 经过 cache 处理后, Bad Request 消失, ThroughPut 略有增加, 但效果不大, 对方法体内函数进一步跟踪发现, WebDav 的连接操作产生大量网络等待并且占用时间很大。于是研究 SardineClientFramework

分析: Sardine 以其简洁的编程调用语法便于使用, 但内部方法没有对多线程处理, 然而大量的网络等待原因, 突出的问题是连接问题, 而 Sardine 有其自身的连接 http pool, 可以用来优化。

```
private static final HttpClientBuilder builder = HttpClientBuilder.create();
builder.setConnectionManager (new PoolingHttpClientConnectionManager(100, MAX_VALUE,
TimeUnit.MILLISECONDS));
sardine = new SardineImpl(builder, user, password);
```

(3) 现象: 经过加入连接池后, 吞吐量已达到 120 以上, 从 VisualVM 的 Sampler 仍然看到很多的 JDBC 处理, 说明 SQL 需要进一步优化

分析: 由于数据库采用的是 Flyway 和 Apache Common Pool, 在连接池上没有问题, 就需要查看表定义, 发现在表 A 与表 B 之间有外键关系, 且一对多, 但从 API 中能看到查询方法只需一个表 A 的内容, 而查询语句是表 A 和 B 的连接查询, 所以改表为独立表, 并修改相应级联删除的 API 逻辑。

(4) JDBC 下 PostgreSQL 和微软 SQL Server Driver

性能比较

Postgres 的性能会高很多, 对于多线程处理优化明显

(5) 一个小且容易遗忘的步骤

关闭控制台日志输出大量 INFO

5 结束语

综上所述, 系统优化不仅仅是靠某一工具或者结论能够放之四海而皆准, 而是需要更多的调研和大量的实验得出的坚实的结论, 每个系统都有着不同的特

点, 尤其在跨平台跨系统调用上找到更多有效的办法。从而对以后系统合理的设计和系统更新有更明确的解决方案。

参考文献:

- [1] Spring security[B/OL]. <http://projects.spring.io/spring-security/>
- [2] Sardine client[B/OL]. <https://github.com/lookfirst/sardine>.
- [3] JMeter[B/OL]. <http://jmeter.apache.org/>
- [4] VisualVM [B/OL]. <https://visualvm.github.io/>