哈尔滨工业大学

**计算学部**

**2024年秋季学期**

**《软件架构与中间件》课程**

**实验报告**

**Lab 3：系统整合与质量测试**

|  |  |  |
| --- | --- | --- |
| **姓名** | **学号** | **联系方式** |
|  |  | phantasia\_march@outlook.com |
|  |  |  |

目 录

[1 实验概述 1](#_Toc181716769)

[1.1 实验目的 1](#_Toc181716770)

[1.2 实验要求 1](#_Toc181716771)

[2 实验内容与过程 1](#_Toc181716772)

[2.1 制定目标和分析系统 1](#_Toc181716773)

[2.2 选择测试度量的方法 2](#_Toc181716774)

[2.3 选择相关技术和工具 3](#_Toc181716775)

[a) 负载与压力测试工具 3](#_Toc181716776)

[b) 缓存测试工具 4](#_Toc181716777)

[2.4 制定评估标准 4](#_Toc181716778)

[2.5 设计测试用例 4](#_Toc181716779)

[2.6 运行测试用例（步骤） 4](#_Toc181716780)

[2.7 分析测试结果 4](#_Toc181716781)

[2.8 系统改造前改造前： 5](#_Toc181716782)

[2.8.1 负载测试： 5](#_Toc181716783)

[2.8.2 压力测试： 6](#_Toc181716784)

[2.8.3 缓存测试： 7](#_Toc181716785)

[2.8.4 数据库测试： 8](#_Toc181716786)

[2.8.5 问题总结： 8](#_Toc181716787)

[2.9 改造后： 9](#_Toc181716788)

[2.9.1 负载测试： 9](#_Toc181716789)

[2.9.2 压力测试： 10](#_Toc181716790)

[2.9.3 缓存策略测试： 11](#_Toc181716791)

[2.9.4 数据库分库分表测试： 16](#_Toc181716792)

[2.9.5 数据库状态监控 19](#_Toc181716793)

[3 测试报告 20](#_Toc181716794)

[4 结对开发过程记录 22](#_Toc181716795)

[5 实验总结 23](#_Toc181716796)

[6 教师评语 24](#_Toc181716797)

[文档全部完成之后，请在上述区域点击右键，选择“更新域”，在打开的对话框中选择“更新整个目录”]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **学号：** |  | **姓名：** |  | |
| **学号：** |  | **姓名** |  | |
| 实验概述 | | | |  |
| 实验目的  1. 学习针对分布式系统进行质量测试的方法、工具 2. 学习使用诸如Jmeter的系统质量测试工具，掌握其安装与配置的方法 3. 能够根据性能分析与评估报告对系统的架构细节进行调优或整体升级 4. 学习如何撰写测试报告  实验要求  1. 2人结对成组 2. 按照课程中7步骤测试法，应用实验二中技术对系统进行改造并执行质量测试。 3. 分析改造前后系统的质量指标(包括常规性能指标) 4. 关注压力测试 5. 给出测试报告 | | | | |
| 实验内容与过程 | | | | |
| 制定目标和分析系统 **目标** 评估系统在高并发下的稳定性、响应速度以及消息队列、缓存、数据库分表的性能表现，包括nginx负载均衡、Redis缓存、分库分表的数据库系统、消息队列。我们的项目主要使用python进行编程，代码执行效率受限，将关注以下几个关键指标来确保系统的高可用和较高的效率：   * + **响应时间**：在高并发请求下，系统处理请求的平均时间，目标设定为<=200ms。   + **吞吐量：**单位时间内系统能够处理的请求数量，目标设定为>=2000 TPS（每秒事务数）。   + **资源使用率：**包括 CPU、内存和网络带宽的使用情况，目标是保持 CPU 使用率<=70%，内存使用率<=80%。   + **Redis缓存命中率：**缓存命中率的目标设定为>=90%，以确保频繁请求的数据能够快速访问。   + **数据库查询效率：**对分库分表的查询响应时间目标为<=100ms。   **系统分析** 本系统主要由以下组件构成：   1. **Nginx负载均衡**：用于将请求分发到后端服务。分析将集中在其负载均衡策略（如轮询、最少连接、IP哈希等）对响应时间和负载分配的影响。 2. **Redis缓存**：用于加速数据访问，减轻数据库负担。我们将测试Redis的缓存命中率以及在高负载下的过期策略对系统性能的影响。 3. **数据库（分库分表）**：采用分库分表技术以提高查询性能。我们将分析在高并发条件下，分库分表策略对数据库查询效率的影响，尤其是在多个用户同时访问时。 4. **消息队列**：用于异步处理和解耦系统组件。测试将包括消息队列的处理能力（如最大并发处理数）以及在高负载下的延迟表现。   **分析方式** 在进行性能评估时，我们将采用以下方法：   * + **负载测试：**使用工具（如 JMeter 或 Locust）模拟高并发场景，收集响应时间、吞吐量和资源使用率的数据。   + **监控和日志分析：**通过监控工具（如 Prometheus 和 Grafana）实时跟踪系统的性能指标，并分析日志以识别潜在瓶颈。   + **数据比较：**将高并发测试结果与低负载情况进行对比，分析系统在不同负载下的性能变化。  选择测试度量的方法 在本实验中，我们将通过多种测试度量方法来评估系统性能，以确保各项指标达到预期目标。主要的性能指标包括吞吐量（TPS）、并发用户数、Redis缓存命中率和数据库响应时间。以下是对这些指标的详细说明及其测量方法：   1. **吞吐量（TPS）**   定义：吞吐量是指单位时间内系统能够处理的请求数量，通常以TPS（Transactions Per Second）表示。  测量方法：在负载测试中，我们将使用测试工具（如 JMeter 或 Locust）模拟大量并发请求，并记录在一定时间内成功处理的请求总数。测试将分阶段进行，以不同的并发级别评估系统在不同负载下的吞吐量。     1. **并发用户数**   定义：并发用户数是指同时访问系统的用户数量，影响系统的负载和性能表现。  测量方法：在负载测试中，我们将逐步增加并发用户数 （通过修改测试工具中一段时间内的线程数量来模拟用户的负载），监测系统的响应时间和资源使用情况。通过分析在不同并发用户数下的性能变化，确定系统的承载能力。     1. **Redis缓存命中率**   定义：Redis缓存命中率是指成功从缓存中获取数据的请求占总请求的比例。高命中率意味着系统在访问频繁数据时能够减少对数据库的请求，从而提高响应速度。  测量方法：在使用Redis作为缓存时，我们将记录缓存的总请求数和命中请求数。命中率计算公式为：  命中率=总请求数/命中请求数​×100%    在高并发场景下，我们将监测并分析命中率的变化。   1. **数据库响应时间**   定义：数据库响应时间是指从发送查询请求到接收到查询结果的时间。该指标直接影响用户体验。  测量方法：在负载测试中，我们将记录每个数据库查询的响应时间，包括查询开始和结束的时间戳。我们将计算平均响应时间，并分析在高并发情况下的变化趋势，以评估分库分表策略的有效性。 选择相关技术和工具负载与压力测试工具  * **工具名称**：Apache JMeter * **用途**：JMeter是一个开源的性能测试工具，广泛用于模拟高并发请求，评估系统在不同负载条件下的性能表现。 * **特点**：   + 能够模拟多个用户同时发起请求，以测试系统的吞吐量和响应时间。   + 支持多种协议，包括HTTP、HTTPS、JDBC等，适合多种应用场景。   + 具备图形化界面，便于设计测试计划、查看实时数据和分析结果。   使用方法：   * 创建测试计划，配置线程组以定义并发用户数和请求频率。 * 设计请求，并设置断言以验证系统的响应。 * 运行测试并收集性能数据，包括响应时间、吞吐量等指标。  缓存测试工具 缓存通过编写测试用例与脚本进行测试 制定评估标准  * + 1. 吞吐量   **标准**：系统应支持至少2000 TPS（每秒事务数）。  **评估方法**：在Jmeter负载测试过程中，监测系统在不同并发用户数下的吞吐量数据。   * + 1. 错误率   **标准：**系统的错误率应低于1%。  **评估方法：**记录所有错误响应的数量，计算错误率（JMeter会自动计算错误率）。   * + 1. 数据库查询效率   **标准：**数据库的查询响应时间应小于100ms。  **评估方法：**在测试中记录每个数据库查询的响应时间，并计算平均响应时间。 设计测试用例 **压力测试：**逐步增加并发用户数，记录系统在不同负载下的响应时间和资源使用情况，直到达到系统的崩溃点。  **负载测试：**模拟高峰负载下的真实业务场景，分析系统在持续负载下的稳定性，重点测试nginx对负载的均衡和消息对大量请求的处理，以及分布式数据库的读写能力  **缓存策略测试：**设计空值返回和缓存过期策略的测试，观察多线程并发请求是否会导致重复查询。  **数据库分库分表测试：**测试数据库分片的查询插入效率，检查是否能在不同的表中快速获取数据，验证分库分表的效果和一致性。 运行测试用例（步骤）  * 使用Jmeter进行测试，逐步提高并发量和请求量，记录系统的响应时间、错误率、缓存命中率、互斥锁表现和资源使用情况。 * 针对Redis缓存的过期策略和空值返回，观察高并发下的缓存更新是否会出现数据延迟或缓存穿透现象。 * 对分库分表的数据库进行质量测试（使用Jmeter的JDBC Sampler），验证其分布式处理能力和一致性。  分析测试结果  1. **吞吐量**：根据不同并发量下的吞吐量变化，确定系统的负载承受能力。 2. **缓存性能分析**：分析Redis缓存的命中率、过期策略，判断缓存是否有效减轻数据库压力。 3. **数据库分库分表表现**：对比数据库在不同负载下的查询效率，分析分表是否有效分散了查询压力。 4. **优化建议**：基于分析结果，识别性能瓶颈，提出提升负载能力、优化缓存策略和数据库查询性能的建议。  系统改造前改造前：负载测试： 测试工具：Jmeter  测试用例：逐步增加并发数，直到出现错误，发送随机conversation\_id的请求，通过中间件进行处理并通过订阅者插入到分布式数据库中。  测试截图：      测试结果：并发数1000/s，吞吐率942.9/sec，错误率0%  测试分析：在未加入负载均衡时，单机使用了消息队列（数据库瓶颈不计），可以实现一个稳定1000左右的TPS 压力测试： 测试工具：Jmeter  测试用例：以最大并发进行测试，观察错误率和数据丢失情况，来测试中间件在最大负荷运行情况  测试截图：      **测试结果**：并发数1500，吞吐率1397/sec，错误率29.14%  **测试分析**：经过日志查看，分析问题可能有以下两点：1. http请求处理受限制，我们采用的是flask的轻量级服务器框架，其http请求处理量较小 2. 中间件能力不足，在线程不足时造成数据丢失  **优化：**我们首先更换了商业级别的服务器框架gunicorn，基于flask但对于http请求处理能力更强。其次我们对中间件实现了缓存队列和优先级阻塞，优先处理publish发布消息请求，并阻塞发送给subscriber的请求等待优先请求处理完成，实现更强的消息处理能力（详情见系统改造部分）      优化后测试结果在2000/s并发下，吞吐率1900/sec，错误率为0% 缓存测试： 改造前未实现缓存，本处展示原有功能及逻辑，可与改造后对比    可对话得到回复，并且具有记忆功能，标题总结和打字机效果，可在左侧选择历史对话加载记忆并继续聊天。  原有逻辑：完全从数据库中加载历史聊天记录，不具有缓存功能与分页查询功能，在大量数据的情况下可导致用户体验差，加载缓慢。 数据库测试： 改造前未实现数据库分库分表与分布式部署，本处测试单数据库单表性能  测试工具：Jmeter  测试用例：配置JDBC，向数据库中插入数据      测试截图：    测试结果：50w并发，吞吐率稳定在6k5，错误率0%  测试分析：MySQL本身为商业数据库，性能强劲，其能力上限高，在300w左右才会出现查询插入问题，但受限于机器性能。 问题总结：  1. 数据库连接数有限：使用数据库连接池优化连接，使用shardingsphere-proxy进行分布式数据库部署，水平分片，并增大sharding-proxy的最大连接数。 2. HTTP请求数量受限：通过改用基于轻量级服务器Flask的商业级别服务器waitress或gunicorn，并且设置处理请求的线程数 3. Nginx负载均衡能力：Nginx通过设置更大的最大连接数，使用Nginx最少连接数策略进行负载均衡。并且nginx配置了请求排队，当请求数量超过了突发请求的上限时，将请求放入缓存的队列阻塞。  改造后：负载测试： 测试工具：Jmeter  测试用例：逐步增加并发数，直到出现错误，发送随机conversation\_id的请求，通过中间件进行处理并通过订阅者插入到分布式数据库中。  测试截图：        **测试结果**：并发数5000，错误率0.02%，吞吐量达到4762.3/sec，数据库丢失13条数据。  **测试分析**：在由单机性能加入**nginx负载均衡**后（途中可以看出nginx的上流均衡负载成功启用），使原有中间件系统的能力更加强悍，可以将请求分散到三台机器的中间件进行处理，然后插入分布式数据库，在有限的硬件能力内达到一个比较好的水平。 压力测试： 测试工具：Jmeter  测试用例：以最大并发进行测试，观察错误率和数据丢失情况，来测试中间件和nginx在实现负载均衡情况下的最大负荷运行情况  测试截图：      **测试结果**：在每秒1w并发下，吞吐率6555.6/sec，错误率86%，数据大量丢失  **测试分析**：经日志排查，主要问题为HTTP请求数量受限，服务器能力无法调用更多线程处理并发请求。因为数据库中无明显报错，而中间件与nginx连接中出现大量丢失，这是因为nginx发送的http请求及http处理达到极限。 缓存策略测试： 测试工具：使用redis自带的info指令查看数据库key命中和未命中数量  测试用例：缓存使用allkeys-lru策略，模拟真实情况下，用户查询历史消息的情况，并测试缓存命中率。 编写测试脚本，验证不存在的数据是否在缓存中返回空值，并且只有一个线程请求了数据库：    说明：查询不存在的值时，应该返回空值并将kv设置为coversation\_id:empty=ture    说明：查询不存在的值时，应该返回空值并将kv设置为coversation\_id:empty=ture，并且只有一个线程查询了数据库，其余线程读取缓存中的空kv标记  注意：我们的查询不具有随机性，即用户通过分页查询实现将数据查询一页存入缓存，而用户只能在其查询到的分页内容中选择历史记录并查询详细信息，因此命中率必然为100%，此处测试该功能的有效性和过期策略，以及缓存穿透应对策略。  测试过程与截图：   1. **验证缓存命中**： 首先，向数据库中插入50条数据，分页查询设置为10，即每次用户加载更多历史记录，会查询50条内容，但此时缓存还未更新内容。     可以看到redis中数据仍只有5条，分页数据为1条，然后点击加载更多对话      现在客户端显示了20条内容，并且redis中分页为2条key，而对话数据有25条key，可以看到设置过期时间为1小时，最近更新的内容距离过期时间更长一些     1. **测试缓存穿透策略**：   首先测试缓存空标记和空值  测试结果：    测试分析：当查询11时，数据库和redis中都不存在，会查询数据库，得到空查询结果，函数会设置空标记并返回None。测试成功   1. 测试互斥锁防止大量请求空数据：     测试分析：当查询数据库时，函数会返回None值。如果redis中有这个key，则会返回键值对。通过结果可看到只有第一个None表明查询了数据库，其余读取缓存。 数据库分库分表测试： 测试工具：Jmeter  测试用例：通过向shardingsphere-proxy实现的代理数据库中发送大量插入请求，观察吞吐率和数据分库分表情况    测试截图：      测试结果：100w并发长时间压力测试，吞吐量稳定在5800/s，错误率0%，数据库中实现按conversation\_id进行分布式存储到两个MySQL服务器中。        测试分析：经过观察，发送到数据库的数据是有被缓存并逐步存入数据库的，shardingsphere实现了强大的分布式数据库支持，并且进行缓存来缓解数据库压力，效率很高，该吞吐率应受到机器和docker产生的硬件能力的限制 数据库状态监控 在配置好监控工具Prometheus + Grafana + mysqld\_exporter 我们可以监控到压测时数据库的状态，下图是100000线程/s 压力测试过程中监控工具的dashboard，其中可以查看到一些数据库的监控信息，数据过多，不做更多的展示。包括了数据库当前的连接数，数据库线程活动情况，数据库的请求量，线程的缓存以及数据库的网络监控等。     测试报告 **一、测试内容**   1. **测试目的：**分析系统改造前后的性能与质量提升，主要为以下方面：    * 请求吞吐量    * 请求并发量    * 错误率    * 缓存策略功能测试    * 数据库吞吐量 2. **性能测试方法**：    * **负载测试**：使用 JMeter 模拟不同并发场景，评估系统的响应时间、吞吐量和资源使用情况。    * **压力测试：**使用 JMeter 模拟最大负载场景，评估系统的吞吐量和错误率水平，分析弱点。    * **缓存测试**：分析 Redis 的命中率、过期策略和穿透应对能力，以降低数据库压力。    * **数据库分库分表测试**：测试高并发条件下的事务处理能力。 3. **测试工具**：    * **JMeter**：用于负载测试，监测系统性能变化。    * **Python测试脚本**：实现缓存命中情况，缓存过期和互斥锁等功能测试。   **二、测试用例设计**   1. **压力测试**：逐步增加并发用户数，监测系统资源和响应时间，直至崩溃。 2. **负载测试**：模拟高峰负载，重点分析nginx负载均衡和数据库分表。 3. **缓存策略测试**：验证空值返回和缓存过期策略的效果。 4. **数据库分库分表测试**：检查不同表的数据查询效率和一致性。   **三、测试结果**   1. **改造前测试结果**：    * **负载测试**：1000并发，吞吐率942.9/sec，错误率0%。    * **压力测试**：1500并发，吞吐率1397/sec，错误率29.14%，主要问题在于HTTP请求处理能力和中间件性能不足。 2. **改造后测试结果**：    * **负载测试**：5000并发，吞吐率4762.3/sec，错误率0.02%。    * **压力测试**：每秒1w并发，吞吐率6555.6/sec，但错误率达到86%。    * **缓存策略测试**：缓存命中率达到100%，有效防止缓存穿透。    * **数据库分库分表测试**：100w并发长时间测试，吞吐率稳定在5800/s，错误率0%。   **四、测试结论**   * **系统负载能力显著提升**：在加入nginx负载均衡和中间件优化后，系统的负载承受能力明显增强。 * **缓存策略优化显著**：Redis缓存提高了系统响应速度和稳定性，并且有效防止大量请求穿透到数据库。 * **分库分表方案有效**：数据库分库分表策略生效，并且在高并发下保持了良好的读写吞吐量。   **五、优化建议**   * **提升HTTP处理能力**：进一步优化服务器的HTTP请求处理能力。 * **改进中间件性能**：增加线程和优化中间件的消息处理逻辑，以减少数据丢失。 * **扩展缓存策略**：根据实际应用场景，调整缓存过期时间和策略，进一步降低数据库压力。 * **分库分表策略：**根据实际应用场景，在数据更复杂的时候，可选择加入垂直分表以进一步加强处理能力。 | | | | |
| 结对开发过程记录 | | | |  |
| **（1）角色切换与任务分工** | | | | |
| 表1-1结对开发角色与任务分工 | | | | |
| | 日期 | 时间(HH:MM - HH:MM) | 驾驶员角色 | 领航员角色 | 本段时间的任务 | | --- | --- | --- | --- | --- | | 2024/11/01 | 09:00 - 11:00 | 蔡志宇 | 杨昊岩 | 制定测试方案，确定负载测试和压力测试的具体指标 | | 2024/11/01 | 11:15 - 13:00 | 杨昊岩 | 蔡志宇 | 使用 JMeter 设置负载测试环境，并编写测试脚本 | | 2024/11/02 | 14:00 - 16:00 | 蔡志宇 | 杨昊岩 | 调试缓存策略测试脚本，确保 Redis 缓存命中率记录正确 | | 2024/11/02 | 16:15 - 18:00 | 杨昊岩 | 蔡志宇 | 分析负载测试初步结果，并针对高错误率调整系统配置 | | 2024/11/03 | 09:00 - 11:00 | 蔡志宇 | 杨昊岩 | 实施分库分表策略，配置 ShardingSphere 实现数据库分片 | | 2024/11/03 | 13:00 - 15:00 | 杨昊岩 | 蔡志宇 | 运行数据库测试并监控系统在高并发下的分片效果 | | | | | |
| **（2）工作日志** | | | | |
| 由领航员负责记录，记录结对开发期间的遇到的问题、两人如何通过交流合作解决每个问题的。 | | | | |
| 表1-2 结对开发工作日志 | | | | |
| | 日期/时间 | 问题描述 | 最终解决方法 | 交流过程 | | --- | --- | --- | --- | | 2024/11/01 | 负载测试中错误率偏高，影响系统稳定性 | 调整 Nginx 负载均衡策略，减少请求压力 | 蔡志宇提出可能是由于请求分配不均，杨昊岩同意并一起调试负载均衡设置，尝试多种策略最终找到最优方案。 | | 2024/11/02 | Redis 缓存未命中率较高，导致数据库压力大 | 调整缓存过期策略和增加缓存容量 | 蔡志宇观察到高频数据未缓存，建议增大缓存容量，杨昊岩配合修改缓存策略，重新测试后问题得到解决。 | | 2024/11/03 | 数据库查询响应时间过长，分库分表策略未见明显改善 | 引入查询优化索引，提升数据访问效率 | 杨昊岩提议优化索引结构，蔡志宇协助进行数据库重构和索引设置，验证后数据库查询效率显著提升。 | | 2024/11/03 | 高并发条件下数据库插入操作丢失数据 | 通过引入事务处理和多线程同步机制减少数据丢失 | 蔡志宇注意到数据丢失现象，杨昊岩提出用事务处理解决，通过讨论和测试，成功改善数据一致性问题。 |   【注意】该表格可自行增加更多的行 | | | | |
| **（3）结对开发工作现场照片** | | | | |
| 请其他同学帮助拍摄结对开发现场照片至少2张。 | | | | |
| 结对开发现场照片1 结对开发现场照片2 | | | | |
| 实验总结 | | | |  |
| 通过本次实验，我们深入学习并实践了事件驱动系统的架构风格、观测者模式和消息中间件的应用。在实验中，通过使用Nginx进行负载均衡、Redis进行缓存处理，以及ShardingSphere进行数据库分库分表，我们进一步理解了分布式架构在高并发场景下的性能表现和优化方法。这些技术帮助我们在系统设计中实现了模块解耦，提高了系统的可扩展性和响应速度，使我们能够应对大规模请求的压力，达成了预期的吞吐量和低错误率的目标。  实验中，我们使用了观测者模式来实现不同模块间的松耦合设计，使得消息的发布和订阅流程更具灵活性。通过引入消息中间件，系统的异步处理能力得以提升，实现了组件间的解耦与高效通信。尽管取得了一定的成果，但我们也发现中间件的处理能力和HTTP请求处理能力仍然是性能的瓶颈。未来我们可以通过进一步优化中间件性能、引入更高效的消息处理机制，以及扩展缓存策略来提升系统的整体稳定性和响应能力。  本实验不仅增强了我们对分布式架构和相关技术的理解，还让我们体会到在实际开发中进行性能调优的重要性。 | | | | |
| 教师评语 | | | |  |
|  | | | | |