作业验收

hw1

2. 所使用的 Scope 属性值的原因：
    - TimerService 使用了@Scope(value = “session”)，因为要求记录的是用户每次登录 E-BookStore 后的会话保持的时间，而在不同浏览器登录时、会话是不同的。这个要求是需要维护多个用户间独立的状态。因此需要设置为“session”，使得 TimerService 对于每个会话能够创建对应的实例，持续维护登录时间的变量。
    - LoginController 中、虽然不维护登录时间的变量，但是需要注入对应的 TimerService 实例，如果设置 LoginController 为 singleton，会报出实例依赖注入错误的异常。因此在这里也设置为@Scope(value = “session”)，保证成功注入。
    - （注：这里询问过助教、也可以通过设置 LoginController 为 singleton，同时在 TimerService 设置代理模式 proxyMode 保证每次调用 LoginController 的时候、注入对应 session 的 TimerService 实例，可以节约资源）

hw2

2. 所使用的 Transactional 传播属性值的原因：
   最终在 Service 的购买函数 purchaseFromCart，OrderDao 的 save order 函数和 OrderItemDao 的 save orderItem 函数中都使用了 REQUIRED 作为传播属性值（即不加额外设置时的默认传播属性值）。

   其中 REQUIRED 属性值表示：如果目前进程中有事务，那么就加入到这个事务中，如果没有事务，那么就新建一个事务。而 REQUIRES_NEW 属性值表示：不管有没有事务，都新建一个事务。（即使外层有事务，也会新建一个事务）即新事务 TX2 的成功或者回滚都不会影响到外层事务 TX1 的成功或者回滚。

   作业中的要求是：必须保证 orders 和 order_items 两个表的数据要么都插入成功，要么都不插入，即这两个表的插入操作必须在一个事务中完成。使用 REQUIRED 属性值满足这一要求。而如果使用 REQUIRES_NEW 属性的话、不管在 save order 或者 save orderItem 上使用，都不满足这一要求。此外，如果在 save orderItem 上使用的话，还可能会导致作业 ppt 中所述的 SQL 异常。

   情况 5，9 的事务回滚是由于插入 orderItem 时的异常导致。这个异常是因为 save orderItem 使用了 REQUIRES_NEW 属性，导致其在执行时、从包含 purchase 和 save order 的原事务 TX1 里脱离，新建了一个独立的事务 TX2，导致 TX1 被暂时挂起。然而，此时 TX1 没有 commit 提交，因此 TX1 中的 order 数据并未落盘，所以在 TX2 中尝试保存以 orderId 作为外键的 orderItem 记录时，TX2 的 orderitemDao 无法看见没有提交和落盘的 orderID，无法直接 save，无法插入 orderItem。因此最终抛出 Unable to find com.example.demo.models.Order with id 6 的异常，并使得 TX2 直接回滚。而又因为在 TX1 中，TX2 发生的异常被 try&catch 捕获处理、因此 TX1 仍继续运行。所以在这两种情况下、TX1 正常运行至结束、并提交 save order 的记录。（情况 9 中尚未运行到 result = 10 / 0 一行）因此导致 order 表中有数据、但 orderItem 表中无数据。

3. 对比分析：
    1. 在前端工程中，使用 JavaScript 监听订单处理结果消息发送到的 Topic，然后刷新页面。

       • 优点：实时性：能够立即响应消息，无需等待用户手动操作。
       • 缺点 1：安全性问题：在前端直接连接 Kafka，可能暴露服务端的细节，增加安全风险。
    2. 在前端发送 Ajax 请求获取订单的最新状态，后端接收到请求后将订单状态返回给前端去显示。

• 优点 1：易于实现：基于 RESTful API 的设计，易于理解和开发，且与传统的 Web 开发模式一致。
• 优点 2：兼容性好：Ajax 在各大主流浏览器中都有良好的支持，适合广泛的用户群体。
• 缺点 1：延迟问题：需要定期轮询获取状态，可能导致延迟，以及造成额外的开销。如果未设置合理的轮询间隔，可能会导致用户看到过时的信息。
• 缺点 2：并发性差：当大量用户下订单的时候、订单可能无法很快地处理完，如果用户要尽快知道订单的处理结果，会造成很多次请求还不能获取到最新的订单状态，这时的性能远不如直接通过 websocket 把处理结果推回给前端。

3. 采用 WebSocket 方式，后端的消息监听器类监听到消息处理结果 Topic 中的消息后，通过 WebSocket 发送给前端。

   • 优点 1：高实时性，实现了双向通信，解决了使用 kafka 时告知前端订单什么时候被处理完的问题，能够立即将订单处理成功的消息推送给前端，用户体验极佳。
   • 优点 2：减少流量：只在有订单处理成功的新消息时才进行数据传输，降低了不必要的网络请求。
   • 缺点 1：兼容性问题：虽然大多数现代浏览器支持 WebSocket，但部分老旧浏览器支持 WebSocket。
   • 缺点 2：连接管理：引入了 ws 协议，需要额外处理连接的生命周期、重连逻辑等问题，增加了项目复杂度。

hw3

2. 为什么要选择线程安全的集合类型来维护客户端 Session：
   因为在 web 应用服务中，不知道同时会有多少客户端对服务端进行连接，如果在不同线程中对一个非线程安全的类型进行读写操作，会出现并发错误。
   使用线程安全的 ConcurrentHashMap 保证所有函数对这个类型的操作都是原子性的，因此在多个客户端同时 OnOpen，OnClose 的时候、保证了对 ConcurrentHashMap 操作的原子性，避免并发错误。

   选择的类型为什么是线程安全的：
   ConcurrentHashMap 在内部使用了分段锁的机制。整个哈希表被划分为多个段（segments），每个段都有自己的锁。当线程要访问某个段时，只需要锁住该段，而不是锁住整个哈希表。这种设计保证了线程安全，也提高了并发性能，因为多个线程可以同时访问不同的段。

3. 在数据库事务管理中，持久性是指事务提交后即使数据库产生故障，事务提交的结果仍然可以在数据库中访问到，不会丢失。请编写文档回答下面的问题：

   1. 如果数据库系统在事务执行过程中不断地将事务操作的结果执行落盘操作，会带来什么潜在问题？可以如何处理？
      - 潜在问题：如果该事务尚未执行完时、发生了系统故障，由于该事务先前的操作结果已经落盘、但是该事务后续的操作尚未进行，因此会影响事务原子性
      - 如何处理：系统重启时需要根据 undo 日志中的内容回滚该事务
   2. 如果数据库系统在事务执行提交后再将事务操作的结果执行落盘操作，会带来什么潜在问题？可以如何处理？
      - 潜在问题：如果在事务执行提交之后、在结果落盘之前、发生了系统故障，相当于已完成的事务尚未将操作写入磁盘，即影响了数据库的持久性，因为此时事务提交后由于数据库产生故障，事务提交的结果不能在数据库中访问到，数据丢失。
      - 如何处理：系统重启时需要根据 redo 日志中的内容重做该事务

   - 原子性：撤销未结束（不带 Commit、Abort 标记）的事务
   - 持久性：重做已经结束（带 Commit 或 Abort 标记）的事务
   - 原子性

- ① 事务运行期间不刷盘，故障系统重启后自动保证原子性；
- ② 事务运行期间刷盘，故障系统重启需回滚该事务
- 持久性
- ① 事务完成（commit、abort）时刷盘，故障系统重启后自动保证持久性；
- ② 事务完成时不刷盘，故障系统重启后需重做该事务

hw4

1. opsForList()是一个操作，可以用来操作 list，比如 rightpush 和 leftpop。opsForValue()是一个操作，可以用来操作 value（普通的 key-value），比如 set 和 get。
2. 解释日志
    - 首次读写：
        - 首次打开 redis 容器时，登录用户 1 后自动直接跳转至图书 1 详情页面，显示 "从数据库中获取了图书 1"，因为 redis 中尚未有该条图书的记录，需要从数据库中先拿到后、存储到 redis
    - 后续读写
        - 在图书 1 详情页面刷新，即后续读写，显示 "从 Redis 中获取了图书 1"，因为数据已经从数据库中拿到并存储到了 redis 中、因此后续读写的时候、直接从 redis 读到数据、而非再需要到数据库去获取一次数据
    - 关掉 Redis
        - 在处理 redis 未连接的异常前、关掉 redis 后、在书籍详情页面刷新、报错“Redis command timed out”、报出 Reconnecting 中，不能正常运行。
        - 在引入处理 redis 未连接异常函数后、关掉 redis、在书籍详情页面刷新、系统可以正常运行。（无法操作 redis 报出了自定义 catch 中的 log 的 error 日志、但从数据库中获取了图书，前端仍能正常显示）

hw5

2. Gateway 和 Service Registry 在微服务架构中都起到了什么作用
    - Gateway
    - API Gateway 是一个管理微服务请求的入口点，它充当前端与后端微服务之间的中介，同时在多实例情况下还能有负载均衡的效果。在我编写的这个微服务的使用中、前端的所有请求都会先发送到 Gateway 的 url。然后，Gateway 从 eureka 注册中心找到各个实际注册的服务的位置后再进行访问。这样做可以让前端不需要知道具体的服务地址，只需要知道 Gateway 的 url 就可以了。（一种例子是当每个微服务或者容器自动重启时、其 ip:port 可能会发送变化，但是这种通过 Gateway 的访问方式可以屏蔽这种变化。client 通过 Gateway 调用 service 的方式，使得使用服务和 service 的实际位置解耦）（此外还易于使用配置文件进行访问方法的配置）
    - Service Registry
    - 在作业中使用了 eureka 作为 Service Registry，当一个服务启动时、它会注册到这个注册中心里，让注册中心统一保存管理这个服务的实际位置。和上面提到的例子类似、当该服务进行重启时、其位置可能会发生变化。但只要将服务注册进去 eureka，就不需要改变通过 Gateway 访问服务的代码。提高了可维护性。
3. 解释函数式服务的无状态指的是什么?

- 对于“状态”的定义：一种是在内存里保存的状态、比如 session、还有一种是在数据库里、比如订单的数据信息等
- 而函数式服务的“无状态”的意思是：函数式服务的输出只与输入有关、不依赖于其他的内存里或者是数据库里的状态。而且执行完一个函数之后、不会对后续的函数服务产生结果上的影响

函数式服务为什么容易扩展？

- 因为函数式服务是无状态的、所以这个服务器就不需要进行状态的维护（不需要配数据源之类的）意味着每次调用都是独立的，且不会受到之前调用的影响。使得横向扩展变得简单，因为可以随时增加或减少实例以应对变化的负载。同时、函数可以轻松组合成更复杂的函数服务或工作流，这种模块化设计使得整体系统更容易扩展和维护。与微服务架构相结合，每个函数可以实现一个特定的业务逻辑，使得各个功能模块可以独立扩展，便于在需要时快速调整。

hw6

3. 在聚簇索引上执行查询，尤其是范围查询时，它的执行效率会非常高。请问，为什么聚簇索引的效率高？
    - 聚簇索引指的是按照每张表的主键构造一颗 B+树，同时叶子节点中存放的就是整张表的行记录数据，也将聚集索引的叶子节点称为数据页，树上的叶子顺序和在磁盘上的顺序是一样的，而从磁盘中每次读取数据、是按照连续的页来进行读取的，因此每次读取页的时候、会把更多的这个表中的数据在这个页中读到，因此减少了磁盘 IO 次数，提高了效率。
4. 如果要将你的图书封面以 Base64 的形式存储在 MySQL 中，那么你在数据表的设计中，是以 VARCHAR 类型来存储，还是以 LONGBLOB 形式存储？请详细解释你的理由。
    - VARCHAR 的最大存储限制是 64KB；
    - LONGBLOB 是 BLOB 类型中的一种，最大可以存储 4GB 的二进制数据。对于大的数据、会存储一个指向实际数据的指针。
    - 通过观察，我的原始图片数据的大小全部大于 64KB、而通过 Base64 的形式转化后，其大小会增加 33%，肯定超过了 VARCHAR 的存储限制，如果用 VARCHAR 存储，会报错。因此应该使用 LONGBLOB 形式进行存储。
5. 当你的图书表上建立复合索引以加速涉及多个字段的查询时，请用 SQL 语句给出你建立该复合索引的方式，并详细解释你为什么构建这样的索引，包括字段顺序、升降序排序等因素是如何确定的。
    - 建立如下两个复合索引，不同点是 price 的排序方式不同：

```sql
CREATE INDEX idx_book_title_price_asc
ON book (title, price ASC);
CREATE INDEX idx_book_title_price_desc
ON book (title, price DESC);
```

    - 字段顺序：先 title 后 price
    - 升降序排序：price 有升序有降序
    - 为什么构建这样的索引：因为在具体的电子书店的业务中、会涉及到对书名的后方模糊查找、并让查找的书本结果按照其 price 大小按照升序或者降序进行排序。过程中会涉及到诸如下方的 SQL 语句 SELECT * FROM book WHERE title LIKE 'Java%' ORDER BY price ASC; （或者 ASC 变成 DESC）
    - 字段顺序原因：这类查询中的 title 是主要的筛选条件（使用了 LIKE），而 price 只是用于排序。而且 title 是查询的最重要字段，并且会常常出现在 WHERE 子句中（即使不需要排序、只按照书名搜索也会出现），应该把它放在索引的前面。

- 升降序排序原因：因为有具体的按照 price 的升序和降序进行排序的业务，使用不同的升降序索引提高效率。

6. 你认为你的订单表的主键使用自增主键还是 UUID 好？为什么？
   - 我认为我的订单表的主键使用自增主键更好。因为对于电子书店来说、订单的数据是会比较大的，而使用自增的数值主键所需要的空间比 UUID 所占用的空间要少，可以减少空间的占用。同时，当订单数据比较大时，由于 UUID 生成的结果是无序的，新行的值不一定要比之前的主键的值要大，因此 innodb 无法做到像自增主键那样，总是把新行插入到索引的最后，而是需要为新行寻找新的合适的位置从而来分配新的空间、会提高维护 UUID 主键索引节点分裂和调整的开销，因此造成性能也较自增主键差。
   - 而 UUID 可以用在分布式数据库的情况下、保证每个主键的唯一性，也在高并发的情况下的性能比自增主键更好。不过在电子书店的业务中、尚未出现这两种业务情况、因此还是认为用自增主键更好。
7. 请你搜索参考文献，总结 InnoDB 和 MyISAM 两种存储引擎的主要差异。
   1. 事务支持：InnoDB 支持事务，而 MyISAM 不支持事务。
   2. 外键支持：InnoDB 支持外键约束，MyISAM 则不支持。
   3. 锁机制：MyISAM 只支持表锁，效率相对较低。而 InnoDB 支持行锁，效率会高一些。
   4. 主键观念：MyISAM 可以不定义主键，如果定义了主键，则会是主键索引。而 InnoDB 必须要有主键，就算没有定义，那么会自动创建一个隐藏的 6byte 的 int 型的索引。
   5. 索引结构：InnoDB 使用聚集索引，而 MyISAM 使用非聚集索引。
   6. 性能差异：MyISAM 会比 Innodb 的查询速度快。InnoDB 在做 SELECT 的时候，要维护的内容比 MyISAM 引擎多很多，比如 MVCC
   7. 查询表的行数不同：
      MyISAM：select count*() from table,MyISAM 可以简单读出保存好的行数*
      InnoDB ： InnoDB 中不保存表的具体行数，因此，执行 select count() from table 时，InnoDB 要扫描一遍整个表来计算有多少行
   8. 应用场景：
      MyISAM 适合：(1)做很多 count 的计算；(2)插入不频繁，查询非常频繁；(3)没有事务。
      InnoDB 适合：(1)可靠性要求比较高，或者要求事务；(2)表更新和查询都相当的频繁，并且行锁定的机会比较大的情况。

hw7

1. 请你详细叙述物理备份和逻辑备份各自的优缺点。
   - 物理备份：直接备份文件 raw copies
   - 优点：适合比较大、比较重要的数据库，恢复也会比较快，
   - 缺点：不能适用版本迁移，备份文件较大
   - 逻辑备份：将数据库的表结构和表中的数据转换成 SQL 语句的形式备份
   - 优点：适合数据量比较小的情况，可以迁移到不同的机器和版本的数据库上
   - 缺点：执行恢复的时候会比较慢
2. 请你参照上课的举例，详细描述如何通过全量备份和增量备份来实现系统状态的恢复。
   首先、开启全量备份和增量备份，使用 mysqldump 进行定期的全量备份，使用 MySQL 的 binlog 功能实现增量备份，在启动 mysql 时加上--log-bin 参数即可启动 binlog 功能进行增量备份
   - 假设在周三 8AM 的时候，发生崩溃、需要从备份恢复数据库状态

- 首先、我们先从我们最近的 full backup（全量备份）开始恢复，执行
  shell> mysql < backup_sunday_1_PM.sql 后，数据库恢复到了周日 1PM 的状态
- 然后、如果 gbichot2-bin.000007 和 gbichot2-bin.000008 文件存储了直到周二 1PM 的时候的增量备份，我们可以执行
  shell> mysqlbinlog gbichot2-bin.000007 gbichot2-bin.000008 | mysql
  将数据库通过增量备份恢复到周二 1PM 时候的状态。
- 注意这里 mysqlbinlog 命令需要在同一行进行执行、不能分开执行、不然会出现问题。比如如果在 bin007 里创建了一个临时表，然后在 bin008 里使用到这个临时表的话、如果分开执行、通过 bin008 恢复的时候会报错“unknown table”
- 最后、对于最新的增量备份，我们会有 gbichot2-bin.000009 文件 (和若干跟随的文件)，因为这个 binlog 在崩溃的时候还没有被截断、因此我们需要执行
  shell> mysqlbinlog gbichot2-bin.000009 ... | mysql
- 将数据库状态恢复至崩溃时的状态（其中...代表的是其他若干相关文件）
  注意我们为了能使用以上的文件进行最后的恢复，我们需要把这个 binlog 和 data 文件存在不同的硬盘上，防止一块硬盘坏了，binlog 和 data 同时丢失。

3. 请你按照你的理解，阐述 Partition 机制有什么好处?
   - Partition 使得单个的一张表的不同部分可以分布在不同的机器上存储，当一个表在一台机器上放不下了的时候、可以分区让数据能够放得下。
   1. 加速查询。分区之后、因为已经相当于做了一次分区的范围限制、因此在对应分区上的搜索等操作都会变快。
   2. 方便删除批量数据。可以直接把一个分区 drop 掉。而不是一条一条 delete
   3. 也可以做分区的替换，比如使用 EXCHANGE PARTITION 方便地对某个分区进行批量的处理、比如替换不同的学生信息。
4. 如果数据文件在一台机器上有足够的存储空间存储，是否还需要进行 Partition？为什么?
   - 也可以做 Partition，视需求而定。当数据分区可以提升对应的业务能力时，就可以使用，并非只有数据存储空间不足时才能使用 partitioning。
   - 分区做搜索、会把搜索的范围限制到一个区域里，即缩小了搜索范围、所以搜索会更快。当数据量庞大且用户查询通常集中在某个特定字段的连续范围或离散值时，比如频繁查询某一特定年份的数据，可以考虑通过对该字段（如时间字段）进行分区，从而优化查询性能。类似地，对于其他具有特定查询需求的字段，也可以根据其访问模式选择合适的分区策略，以提升查询效率。
   - 对于那些具有时效性并需要频繁插入的数据，例如天气数据，通常需要持续更新，同时还需要定期归档过期的历史数据。在这种情况下，采用基于时间的分区策略可以有效地将更新和归档操作分散到不同的分区中，避免在大量数据迁移、删除和插入时对系统性能造成过大影响。
   - 也可以做分区的替换，比如使用 EXCHANGE PARTITION 方便地对某个分区进行批量的处理、比如替换不同的学生信息。

hw9

2. 请你用 Word 文档回答下面的问题:
   - 请阐述日志结构数据库中的读放大和写放大分别是什么意思？(2 分)
     - 读放大：由于在 LSM-tree 中、是分层存储数据的，有时候读取某个数据项时需要读取比原数据规模更大的的数据内容。比如如果不使用布隆过滤器去做优化、当读一个数据库中不存在的值的时候，会需要从 L0 层的 SSTable 一直读到最后一层的 SSTable、直到检查完了所有的 SSTable、导致读的开销巨大。

    - 写放大：每次写操作导致实际进行写操作的总数据量超过了原始写入的数据量。当进行单次写操作的时候，会涉及到 SSTable 的合并（L0 层满时将阻塞内存到磁盘的 Flush 过程），如果这种合并的情况可能会一直延续到最后一层的 SSTable、导致这次单次写操作的开销就变得巨大。
- 请阐述向量数据库中两种以上不同的相似度计算方法中所采用的具体计算方式？(1 分)
    - 向量数据库中的相似度计算方法用于衡量向量之间的相似度，常见的计算方式包括**余弦相似度 (Cosine Similarity)**、**欧几里得距离 (Euclidean Distance)** 和**点积 (Dot Product)**，每种方法采用不同的计算方式。以下是它们的具体计算方式：

1. 余弦相似度 (Cosine Similarity)

余弦相似度衡量的是两个向量在角度上的相似度，忽略它们的大小，仅关注方向。它的计算方式如下：

$$
\text{Cosine Similarity}(A, B) = \frac{A \cdot B}{|A| |B|}
$$

其中：

- (A $\cdot$ B) 是向量 (A) 和 (B) 的点积。
- (|A|) 和 (|B|) 分别是向量 (A) 和 (B) 的模（即它们的长度）。

范围：余弦相似度的值范围是从 -1 到 1：

- 1 表示两个向量完全相同（方向相同）。
- 0 表示两个向量正交（无关）。
- -1 表示两个向量完全相反（方向相反）。

2. 欧几里得距离 (Euclidean Distance)

欧几里得距离用于计算两个向量之间的直线距离，衡量它们在空间中的"相离"程度。其计算方式为：

$$
\text{Euclidean Distance}(A, B) = \sqrt{\sum_{i=1}^{n} (A_i - B_i)^2}
$$

其中：

- (A_i) 和 (B_i) 是向量 (A) 和 (B) 在第 (i) 个维度上的分量。

范围：欧几里得距离的值从 0 到正无穷大：

- 0 表示两个向量完全相同。
- 越大的值表示向量之间的差异越大。

3. 点积 (Dot Product)

点积是一种简单的相似度度量，表示两个向量在某种程度上的"投影"关系。其计算方式为：

$$\text{Dot Product}(A, B) = \sum_{i=1}^{n} A_i B_i$$

其中：

- (A_i) 和 (B_i) 是向量 (A) 和 (B) 在第 (i) 个维度上的分量。

范围：点积的值范围是从负无穷大到正无穷大：

- 正值表示两个向量的方向相似（指向相同或相似方向）。

- 0 表示两个向量正交（方向无关）。
- 负值表示两个向量方向相反。

总结：

- **余弦相似度**：衡量向量之间的角度差异，范围从 -1 到 1。
- **欧几里得距离**：衡量向量之间的直线距离，范围从 0 到正无穷。
- **点积**：衡量向量在同一方向上的投影大小，范围从负无穷大到正无穷。

不同的相似度计算方法适用于不同的应用场景，例如，余弦相似度适用于文本数据中的向量相似性比较，欧几里得距离常用于几何计算，而点积则常用于机器学习中的特征向量相似度计算。