01 Stateful and Stateless

1. 本地缓存比较小、所以需要再有分布式缓存
2. 大部分数据是只读数据、所以可以使用缓存，而不是每次都去数据库查询
3. 主从复制的服务器可以每隔一段时间切换一下角色，保证多机器的冷热均衡
4. CDN 服务器：内容分发网络，可以缓存静态资源，减少服务器压力，在不同地区有不同的服务器节点，可以加速访问
5. 反向代理服务器：用于分发请求。“反向”这个词强调的是代理的作用是代表后端服务器来接收客户端的请求。反向代理则是指客户端直接向代理服务器发送请求，而代理服务器再将请求转发到后端的目标服务器。（让音视频可以在线点播）
6. 数据访问网关：封装数据访问的细节，提供统一的数据访问接口，对外提供服务（比如数据到底是从数据库还是缓存中获取）
7. 消息服务器：把服务变成基础设施
8. 微服务：抽离出来公用的服务，比如 A 应用和 B 应用都需要用到支付服务，那么就可以把支付服务抽离出来；或者比如说登录服务、注册服务等等
9. http 是无状态的协议，如果有状态，内存会爆
10. 实例池：限制每种实例的数量，防止内存爆炸，然后用 LRU 策略来淘汰，但是会有 swap out，硬盘读写，会影响性能
11. springboot 中也可以通过配置@Scope("prototype")，@Scope("session")等来控制实例的创建。prototype 即每次请求都会创建一个新的实例（任意情况下都是新的），session 即每个会话创建一个实例（见 ppt 表格）。singleton 即单例模式，即整个应用只有一个实例，这个是默认情况下的。request 即每个请求创建一个实例。
12. 如果需要不需要维护多个用户间独立的状态的话、就用成 singleton，如果需要维护多个用户间独立的状态的话、就用成 prototype。（状态即程序变量，比如在前端显示用户已经在线了多久，这个时间就是状态，需要在后端维护，而且每个用户的状态是不一样的）
13. 最好是有状态的内容尽量少（压缩、但是有些东西必须要靠状态实现），因为有状态的内容会占用内存，而且会有并发问题。无状态的内容尽量多。极致就是函数式编程，没有状态，只有函数。好处就是后端不用维护那么多对象，消耗资源就很少。
14. 连接的本质是一个线程，线程之间需要切换。所以连接池不是越大越好，因为线程上下文切换也是有开销的。所以连接池的大小要根据服务器的性能来调整。连接池的大小设置和用户并发数没有关系。如果高并发之后性能不行，那么就是服务器性能不行，不是连接池参数设置的问题。
15. connections = (2 * core_count) + effective_spindle_count;（线程去做 IO 的时候，cpu 是空闲的，所以可以*2 作为等待线程的数量）链接 【遗留问题】effective_spindle_count 是什么?

首先可能形成瓶颈的是硬盘，网络 IO 和内存。然后 MySQL 是半双工的网络模型，半双工就是在同一时刻只能进行收发的其中一个动作。所以有了 core_count*2 的说法，一半在写数据，一半在收数据。effective_spindle_count 是在不能命中的时候需要硬盘去寻址，读页数据，比较耗时，这时候 CPU 不会一直等着硬盘，可以去处理其他，也就是说如果只有一个机械硬盘，我们需要多加 1 根线程来处理等待硬盘读的事务。

那为什么文中会说是在这个值附近呢？而不是说很精确的就是这个值呢？我的理解是这个也和其他一些比如事务类型，缓存，内存计算有关，抛开硬盘部分不说，如果事务类型都是很简单的计算，同时缓存能够 100 命中的话，那这个连接池数量还可以加，因为内存这时候可能网络 IO 成了瓶颈，可以多加一些链接来满足内存的运算量。如果本身事务比较复杂，内存计算不及时，那么这时候再加链接数量反倒增加了上下文切换的压力。

02 Messaging (JMS, Kafka)

1. 同步通信的问题：因为我们是面对接口编程的。有大量实例，可能调用不成功。或者网络不佳，也会失败。（没有送达保证）同步调用也没有请求缓冲（忙了也会失败）。同步还是紧耦合的，即调用函数的时候会需要根据函数名重新修改代码。同时会比较难维护。即过于强调了发送了一个 request，就要等待 response，这样就会导致请求的阻塞。所以同步通信不适合高并发的场景。（例子 打电话和发短信的例子）
2. 如果不用消息队列，就没有 request buffer，会在 controller 或者 service 或者 dao 里面阻塞，导致可能请求会丢掉。而且前端也不知道请求的状态。这几层之间都可以通过消息中间件进行交互。比如，kafka 中有很多 Topic 的缓冲。在消息队列的场景下，client 发送请求，然后 controller 直接返回一个确认收到的响应给 client。那怎么 client 如何最终确定这个请求处理完了呢：一种是再发一次 ajax 请求，一种是 websocket，一种是把结果重新发给另外一个 B 的消息队列，然后 client 从这个 B 的消息队列里面去拿结果。（类似于发邮箱）对应于 ajax ws 和 messaging （消息队列解决一段时间内请求高峰，请求波动的问题，服务器资源不够）
3. 问题在于异步的返回值应该是什么，service.m()还可能抛出一个异常。所以也要把这个异常组成一个消息发给 B 这个结果的消息队列。
4. 消息系统是 peer to peer 的通信方式。是松耦合。在这个 topic 里面的是数据。这个是 data-driven 的，即不指明调用的方法。 SOAP 和 HTML。
5. jms 是标准的 java 里的消息队列。让通信变得异步和可靠。
6. 什么时候要用消息通信？比如下订单。（可以在闲时处理）。
7. 消息的格式：1.header（必有） 2.body（可选） 3.property （可选）
8. 消息队列的 S，C 和 P 是什么意思：S 是 send ， C 是 consumer（client 设定的），P 是 producer（server 设定的）
9. 有时候 A - Kafka - B 之间，会根据设置的 Destination 或者 DeliveryMode 来决定消息的发送和接收。比如说，如果设置了 Destination 为持久化，那么即使 B 没有启动，A 发送的消息也会被保存在磁盘上，等 B 启动之后，会把这个消息发送给 B。如果设置了 DeliveryMode 为持久化，那么即使 Kafka 挂了，消息也不会丢失。如果设置了 DeliveryMode 为非持久化，那么 Kafka 挂了，消息也会丢失，不会发到 B。（比如股市的例子，要保证时序的顺序）
10. 消息头的例子 Clients can not extend the fields
    - JMSDestination (S)
    - JMSDeliveryMode (S)
    - JMSMessageID (S)
    - JMSTimestamp (S)
    - JMSCorrelationID (C)
    - JMSReplyTo (C)
    - JMSRedelivered (P)
    - JMSType (C)
    - JMSExpiration (S)
    - JMSPriority (S)
11. 消息的属性（Property） Clients can extend the fields。比如，扩展属性，给消息加上 selector：name=“B”，就能起到筛选的作用。只能加简单类型。Property name: follows the rule of naming selectors；Property value: boolean, byte, short, int, long, float, double, String
12. 消息体：见 ppt 表格

13. java 对象能够被序列化的意思：实现一个 java 类的接口，使得能变成适合在网络上传播的数据。
14. jms：destination 分为 topic 和 queue（那个图是什么意思）都需要通过链接工厂建立一个到 jms server 的链接，然后找到数据源 datasource。（JNDI namespace）
15. 点对点的消息：client 1 的消息只会被 client 2 接收到，而不会被 client 3 接收到（queue）。而发布订阅的消息：client 1 的消息会被 client 2 和 client 3 接收到。即点对点是一对一的，发布订阅是一对多的（广播 topic）。
16. Durable 和 Non-Durable 的区别：Durable 是持久化的，即消息不会丢失。（consumer 不在线的时候，消息不会丢失）Non-Durable 是非持久化的，即消息会丢失。（不在线的时候，消息会丢失）
17. browser 只浏览
18. 重要的是类的 toString 方法的重写
19. kafka 基于日志通信，只追加（append-only），顺序写，速度快。
20. 对每个用户维护一个 offset，即每个用户都有一个 offset，这个 offset 会记录这个用户读到了哪里。这个 offset 会被保存在 zookeeper 中。如果 kafka 挂了，那么 zookeeper 会把这个 offset 保存下来，等 kafka 恢复之后，会把这个 offset 读取出来，然后继续读取。（用 zookeeper 来管理）
21. 一个 topic 里面可以放相同类型的对象（长度易于管理），也可以放不同类型的对象（缺点是要记录多个偏移量，好处是只用一个队列）。
22. topic 可以分区，比如按照用户 id 来分区，这样可以保证一个用户的消息是有序的。但是这样会导致一个用户的消息只能被一个消费者消费，所以要根据业务需求来决定是否要分区。
23. 分区的目的：可以并发，同时可以集群部署。也可以加上分区备份，保证数据不丢失（在不同机器上，保证可靠性）。
24. 会发送心跳包，如果一个 consumer 挂了，那么会把这个分区的消息发送给另外一个 consumer。
25. 最好在事务里发送消息，配置 transaction 的原因：如果一个消息发送失败，那么就会导致这个消息丢失。所以要配置 transaction，保证消息的可靠性。
26. broker 是一个 kafka server，一个集群里面有多个 broker，每个 broker 里面有多个 topic，每个 topic 里面有多个 partition，每个 partition 里面有多个 offset。
27. 在 Kafka 中，`groupId` 用于将多个消费者组织成一个消费组。将两个 `@KafkaListener` 的 `groupId` 设置为相同的含义如下：
    - **同一组中的多个消费者**：当多个消费者属于同一个消费组时，Kafka 会在这些消费者之间分配消息。这意味着在这个消费组中的每个消费者只能处理自己分配到的消息，而不会重复处理。
    - **负载均衡**：如果你有多个实例的应用程序运行并且它们都使用相同的 `groupId`，Kafka 将会自动在这些实例之间分配消息，从而实现负载均衡。
    - **消息确认**：同一消费组的消费者共同维护消息的偏移量（offset），这意味着当组内的某个消费者成功处理了消息后，偏移量会更新，其他消费者就不会再处理这些消息。
    - 在例子中，虽然两个监听器的 `groupId` 相同，但它们监听的是不同的主题（`topic1` 和 `topic2`），因此它们不会相互影响。`topic1Listener` 处理来自 `topic1` 的消息，而 `topic2Listener` 处理来自 `topic2` 的消息。不过，它们共享相同的 `groupId` 并没有实际意义，因为它们不在同一个主题上工作。

03 Websocket

1. 全双工的通信，即客户端和服务端都可以发送消息（双边都在工作）。没有响应 请求的概念了。
2. 底层仍然是 TCP 协议。（一定要保证收到，会重发）（UDP 是不保证收到的，但是速度快）

3. 前端和后端都运行着 websocket 程序
4. 称为一个 websocket 的 endpoint，有统一定位符
5. 所以需要握手（即注册）。websocket 由握手和数据传输两部分组成。握手是基于 http 的，数据传输是基于 tcp 的。
6. 握手时，如果两个 key 能匹配上、即握手成功。
7. ws://是不加密的，wss://是加密的 443 端口。（ws+ssl）
8. 握手握上了、server 会专门为每一个客户端开一个 session。（server 端维护了一个 list）
9. session.getOpenSessions() 可以获取所有的 session 的 list （当规模大了之后，单纯的 for 循环会有性能问题）
10. ConcurrentLinkedQueue 是线程安全的 （注意要用）
11. 可以增加 websocket 的编码器和解码器，来处理消息的编码和解码
12. 前端要用一个 STOMP 的库
13. 下面的代码是一个简单的例子，即前端发送一个消息给后端，后端再把这个消息发送给前端。其中，`@MessageMapping("/hello")` 是前端发送消息的目标路径，`@SendTo("/topic/greetings")` 是后端发送消息的目标路径。

```
@MessageMapping("/hello")
@SendTo("/topic/greetings")
```

14. 也可以设置 broker 代理。
15. ajax 通过 callback 函数来实现前端的异步通信。消息队列处理前端请求过多的问题（返回给前端订单已接受的响应）。（后端异步，如果是同步的话、数据在后端就直接丢掉了）。websocket 解决订单什么时候处理完的问题。（主动推回给前端）
16. 也不是什么都要用消息队列。比如 login（高优先级的任务）就不用消息队列。
17. service 下的应该是同一层逻辑，不然会有很多代码重复。
18. http session 和 websocket session 是不同的 session
19. react 使用 需要先 npm install websocket ; npm install sockjs-client ; npm install stompjs ; npm install --save-dev @types/sockjs-client ; npm i --save-dev @types/stompjs

```
//vite.config.ts
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";

export default defineConfig({
  plugins: [react()],
  define: {
    // Some libraries use the global object, even though it doesn't exist in the
browser.
    // Alternatively, we could add `<script>window.global = window;</script>` to
index.html.
    // https://github.com/vitejs/vite/discussions/5912
    global: {},
  },
});
// 这里有一个问题、如果在vite中像上面这样使用、尝试使用sockjs的时候、会导致前端控制台没有报错信息，所以也可以用下面的方法，在index.html中加入
```

```
<script>const global = globalThis;</script>;
```

20. SockJS 需要使用 http 或 https 协议进行连接，内部会自动转化为 ws 或 wss 协议
21. npm 设置代理

```
npm config set proxy http://127.0.0.1:7890
npm config set https-proxy http://127.0.0.1:7890
npm config list
```

22. 使用 sockjs 之后、还可能需要配置跨域访问和拦截器的设置，因为是通过 http 协议连接的。
23. 找到一个问题、如果在 vite 里面设置 global 变量、会导致前端 sockjs 没有报错信息。
24. 但是后端需要额外设置 sockjs 的 CORS 策略，而且要分清楚是 patterns 还是 origins。如果是 patterns 的话，那么就是正则表达式，
    如果是 origins 的话，那么就是一个字符串数组。
25. 前端 react 使用严格模式的时候、在调试环境的时候、会自动触发两次 useEffect ，关掉 strict mode 就不会触发两次了。
26. @ServerEndpoint 注解好像不能用于 sockjs 的情况，只能用于 websocket 的情况。自定义 sockjs 的 endpoint 需要其他方式。
27. 注意下面的 patterns ...

```
registry.addEndpoint("/ws").setAllowedOriginPatterns("*").withSockJS();
```

04 transaction

1. OrderDao 和 OrderItemDao ?
2. 问题例子 1：如果两个用户同时对只剩 1 本书的库存下单。（并发情况下需要保证库存的一致性）
3. 问题例子 2：在转账操作中，分批写数据库，如果第一次写成功，第二次写失败，那么就会导致数据不一致。（要看起来像原子性的）
4. 所以有了事务，要么全部执行，要么回滚。（注意是只在数据库（资源管理器）中进行回滚，而不是在代码中进行回滚）
5. EJB 是 JavaEE 的一部分，是一种企业级的 JavaBean。EJB 是一种服务器端组件模型，
   用于构建分布式应用程序。EJB 有三种类型：Session Bean、Entity Bean 和 Message-Driven Bean。
   Session Bean 又分为 Stateful Session Bean 和 Stateless Session Bean。
   Stateful Session Bean 是有状态的，Stateless Session Bean 是无状态的。
6. 注解@TransactionAttribute 是用来控制事务的属性的。
   - TransactionAttributeType.REQUIRED：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。（默认）
     这里的有和没有事务是相对于父子函数的 txattribute 来说的（见 ppt 上 methodA 和 methodB 的例子）
   - TransactionAttributeType.REQUIRES_NEW：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。（比如 TX2 是 log，TX1 是转账，TX2 不稳定，TX1 稳定）

- TransactionAttributeType.SUPPORTS：如果有事务，那么就加入到这个事务中，如果没有事务，那么就不用事务。
  （比如 A 和 B 都是 support，那么两个都没有事务）
- TransactionAttributeType.NOT_SUPPORTED：不管有没有事务，都不用事务。（非事务状态下运行，会挂起父事务）
- TransactionAttributeType.MANDATORY：必须父函数有事务，如果没有事务，就会抛出异常。（必须有事务，否则就会抛异常）
- TransactionAttributeType.NEVER：必须父函数没有事务，如果有事务，就会抛出异常。（必须没有事务，否则就会抛异常）

7. Spring 里面用@Transactional 来控制事务的属性。即事务传播行为。
8. String... 是可变参数，可以传入多个参数，但是要放在最后一个参数。（就是可以相当于是数组）
9. ACID
   - Atomicity：原子性，要么全部执行，要么全部回滚
   - Consistency：一致性，事务执行前后，数据的完整性约束没有被破坏
   - Isolation：隔离性，多个事务并发执行的时候，一个事务的执行不应该影响到其他事务的执行
   - Durability：持久性，事务执行成功之后，数据的改变是持久的
10. dirty reads：脏读，一个事务读取到了另一个事务未提交的数据（进行中的状态，可能会被 rollback）（没有任何隔离）
    non-repeatable reads：不可重复读，切断了事务间内存的交流，但是可能会对数据库同时写。
    phantom reads：幻读，在上面的基础上再对数据库加锁，但是会有不断有新的满足条件的记录进去。（通过对表的加锁定来解决）
11. 所以有了@Transactional isolation 注释
    - Isolation.DEFAULT：使用数据库的默认隔离级别
    - Isolation.READ_UNCOMMITTED：读未提交，会出现脏读、不可重复读、幻读
    - Isolation.READ_COMMITTED：读已提交，会出现不可重复读、幻读
    - Isolation.REPEATABLE_READ：可重复读，会出现幻读
    - Isolation.SERIALIZABLE：串行化，不会出现脏读、不可重复读、幻读
12. read locks：读锁，多个事务可以同时读取数据，但是不能同时写入数据
13. write locks：写锁，只有一个事务可以写入数据，其他事务不能读取和写入数据
14. 这个注释是通过连接在数据库中完成的，而不是在应用程序中完成的。即数据库管理锁。（即不同的隔离级别）
15. 管理分布式的事务：因为多个数据库之间是互相隔离的（spring 在监测到多数据源时会自动管理，不需要代码上的额外操作）
    - 实现：2 phase commit（2PC）
    - Stage 1：prepare（准备阶段）：每个数据库都会把事务的结果写到一个 log 里面，然后返回给协调者
    - Stage 2：commit（提交阶段）：如果所有的数据库都返回了 prepare，那么就会提交，否则就会回滚
16. 但是也有问题、如果 rollback 的时候、到其中一个数据库的网络不通。会导致这个数据库进行启发式的操作，即自己决定是 commit 还是 rollback。
    所以这里只能人工介入了。（就是不能避免第二阶段的问题）

17. 能支持这种事务的，都叫 Resource Manager。比如说，数据库、消息队列、邮件等等。（见 ppt 图）
18. optimistic offline lock：多线程可以读取，但是只有一个线程可以写入。即读取的时候不加锁，写入的时候检查版本号 version。
    （即乐观锁，判断是基于旧数据进行的操作就行，预计并发写的情况不多，即读的概率很高，写的概率不高）
19. pessimistic offline lock：读取的时候加锁，写入的时候也加锁。即悲观锁，判断是基于新数据进行的操作就行，预计并发写的情况很多。
20. Coarse-grained lock：粗粒度锁，即把和一张表关联的所有数据都锁住。（比如说，一个订单的所有商品都锁住）
    也存在乐观锁和悲观锁的区别。（通过 version 表，这里的悲观锁需要对 version 表，版本号加锁）
21. mysql 的默认隔离级别是 REPEATABLE READ。作业中发现一个问题、如果让 orderItem 的 Save 变成 REQUIRES_NEW，会导致死锁，报错如下。SQL 错误代码 1205 和 SQLState 40001 通常表示一个 死锁（Deadlock）错误。注意这里抛出异常的是父事务、可能是因为父事务先加入等待序列、而子事务后加入等待序列、其定时器先到期，所以父事务先抛出异常。

```
2024-10-09T16:46:32.239+08:00  WARN 36108 --- [ntainer#0-0-C-1]
o.h.engine.jdbc.spi.SqlExceptionHelper   : SQL Error: 1205, SQLState: 40001
2024-10-09T16:46:32.239+08:00 ERROR 36108 --- [ntainer#0-0-C-1]
o.h.engine.jdbc.spi.SqlExceptionHelper   : Lock wait timeout exceeded; try
restarting transaction
Order processing failed !
```

05 事务管理

1. 做 websocket 连接的时候 其实需要用到 JWT 鉴权。
2. 回滚：rollback 的时候、需要用到 undo log，即把之前的操作反过来执行一遍。
3. redo log：解决从内存写到磁盘时，数据还没落盘就挂了的问题。即把数据先写到 redo log 里面，然后再写到磁盘里面。数据库重启的时候，会把 redo log 里面的数据重新写到磁盘里面。
4. 两种写硬盘方式，一种是直接落盘，一种是先写 dirty page，然后再写到磁盘里面。后者的好处是速度快，但是有可能会丢失数据。
5. 日志分为两种，一种是记录 sql 语句的，一种是记录 bit 位的。
6. 隔离性：
    - 写读冲突：导致脏读（dirty read）
    - 读写冲突：导致不可重复读（non-repeatable read）（在同一个事务中，读取到的数据不一样，因为在事务中，数据可能被修改了）（可以通过加锁来解决）
    - 写写冲突：导致脏写（dirty write）
7. 幻读：读到其他事务插入或者删除的数据。（和不可重复读的区别是，不可重复读是读到了其他事务修改的数据，而幻读是读到了其他事务插入或者删除的数据）
8. 调度：即多个事务的读写顺序安排。会有串行调度和并发调度。（串行调度一定是正确的）
9. 可串行化调度：一个并发调度 S 如果存在一个串行调度 S'，在任何数据库状态下，S'和 S 产生的结果是一样的，那么就称 S 是可串行化的。
10. 实现方法：找到一个解就行（因为一共的解是很多的）

11. 冲突可串行化调度：：冲突可串行化调度，是从冲突的角度出发，针对一个调度 S 去发现其等价的串行调度 S’ 来确定 S 是一个可串行化调度。
12. 一次操作交换定义为交换事务调度序列中相邻的两个操作，一个交换操作可以将一个调度 A 变成另外一个调度 B。
13. 当交换不会影响两个调度的一致性时，定义该交换得到的两个调度是等价的，该交换为等价交换。
14. 等价操作
    - 交换连续两个相同数据读取操作的顺序:RT1(A) RT2(A)
    - 交换连续两个不同数据读写操作的顺序:RT1(A) WT2(B)；WT1(A) RT2(B)；WT1(A) WT2(B)
15. 非等价操作– 交换连续两个相同数据的读写、写写操作: RT1(A) WT2(A)；WT1(A) RT2(A)；WT1(A) WT2(A)
16. 优先图：利用冲突关系，可以将并发的事务构建成一个图来验证事务的并发调度 S’是否是冲突可串行化调度。
    - 若 Ti 与事务 Tj 存在冲突（读写冲突、写写冲突、写读冲突）（注意是这三种冲突，没有读读冲突），且 Ti 的冲突操作在 Tj 冲突操作之前，则 Ti 向 Tj 连一条有向边（Ti，Tj）。
17. 冲突可串行化的充分必要条件：优先图无环
18. 原子性：运行期间刷盘的优势：内存占用少、但是重启的时候会慢一点。运行期间不刷盘的优势：重启的速度快，但是内存占用大。
19. 持久性： 事务完成（commit、abort）时刷盘，故障系统重启后自动保证持久性；事务完成时不刷盘，故障系统重启后需重做该事务
20. 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
21. ppt 上有例子。
22. 保持脏页的缺点就是内存占用大。写入的缺点就是 IO 操作多，速度慢。
23. FORCE 条件：事务完成强制刷脏 vs 不强制刷脏
    - Force 缺点：每次事务提交都必须刷新脏页，消耗大量 IO 读写资源
    - 解决方法：使用 NO-FORCE 模式，利用 Redo 日志重做事务
    - Redo 日志：记录事务对数据库的所有影响
24. NO-STEAL 条件：事务中间可以刷脏 vs 不可以刷脏
    - No-Steal 缺点：事务执行过程中不能刷新磁盘，必须占有较大的缓冲区空间，不利于多个事务的并发执行
    - 解决方法：使用 STEAL 模式，利用 Undo 日志撤销事务
    - Undo 日志：记录撤销事务所需的内
25. 要解决原子性和持久性问题，各分别存在两种方法，两两组合起来产生四种数据库恢复算法
26. 日志的内容在写入磁盘以后是不会被修改的，，因此所有的日志内容可以顺序写入磁盘，这保证了高效的写入速度。该特性也是建立高可用恢复机制的前提。即数据的随机读写转换为日志的连续读写。
27. 逻辑日志和物理日志的区别：逻辑日志是记录的是 sql 语句，物理日志是记录的是 bit 位。物理日志可以是幂等的，即多次执行不会有影响。（比如一个在逻辑日志中的 insert 操作，物理日志满足幂等性；逻辑日志不满足）物理日志通常还小一点。
28. 有关数据库日志的三个重要性质（见 ppt）
    - 幂等性：一条日志记录无论执行一次或多次，得到的结果都是一致的。物理日志满足幂等性；逻辑日志不满足

    - 失败可重做性：：一条日志执行失败后，是否可以重做一遍达成恢复目的。物理日志满足失败可重做性；逻辑日志不满足：例如插入数据页面成功，而插入索引失败，重做插入这个逻辑日志失败。
    - 操作可逆性：逆向执行日志记录的操作，可以恢复原来状态（未执行这批操作时的状态）物理日志不可逆（页面偏移量位置可能被后续记录修改），逻辑日志可逆。
29. 所以重做 redo 日志是物理日志（因为逻辑日志不具有幂等性，逻辑日志不具有失败可重做性，一条逻辑日志记录可能对应多项数据修改（表、索引），崩溃时 X 对索引 造成影响，但没有影响数据页面。），而撤销 undo 日志是逻辑日志或者物理逻辑日志（ 物理日志不具有可逆性，无法处理数据项位置变化的情况，页面分裂后 A 的地址发生了变化，撤销的时候无法定位数据项，因此物理日志一般不能用于回滚，Undo 日志须用逻辑日志）。
30. 有一个影子拷贝的方法，事务修改在拷贝数据库上(或者影子拷贝页面上)
    - 优点：影子页面，仅拷贝修改的页面
    - 缺点：效率低，难并发
31. 补偿日志（Undo 日志的 redo 日志，基于 undo/redo 日志的恢复，每次执行 undo 日志记录后，数据库需要向日志中写入一条补偿日志记录 （compensation log record，CLR），记录撤销的动作，记录已经 undo 的日志，保证 undo 不被重复执行）、还有 redo 日志和 undo 日志的实际流程在对应的 ppt 中。
32. undo 里面就是每次操作完的旧值（因为要撤销）、redo 里面就是每次操作完的新值（因为要重做）

06 Multi-threading

1. 问题：进程间的内存是隔离的。可以通过 RMI 来实现进程间的通信。RMI：Remote Method Invocation，即远程方法调用。
2. 但是整个 JVM 是一个进程，里面的所有比如 spring 都是线程。
3. 有两种方法，一个是实现 Runnable object，一个是当继承 thread 的子类。但是推荐是前者，因为 java 是单继承的。（即每个类只能有一个子继承的类）
4. sleep 可能会有在多台机器上精度不一致的问题。
5. 多线程下会存在同步性和内存一致性的问题，比如操作一个类里的同一个对象
6. 所以有了 synchronized 关键字，即同步锁。synchronized 修饰的方法，只能有一个线程进入。synchronized 修饰的代码块，只能有一个线程进入。（而且是多个函数之间会共享一把锁，每一个对象会有一个内部锁，用 1bit，这个对象里面所有的 synchronized 方法都会共享这个锁）（static 的也需要这个锁）
7. 也可以用 synchronized(this) 来锁代码块
8. 也可以用一个对象当作一个锁，即 synchronized(object) 来锁代码块
9. reentrant synchronization 可重入锁：即一个线程可以多次进入同一个 synchronized 方法，但是要注意，如果是多个 synchronized 方法，那么就会有问题。（即一个线程可以多次进入同一个 synchronized 方法，但是不能进入不同的 synchronized 方法）
10. java 中的原子性操作：Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double). Reads and writes are atomic for all variables declared volatile (including long and double variables).
    - 读和写是原子性的，即一个线程读到的是另一个线程写的值，而不是自己的缓存值。（除了 long 和 double 之外的所有类型）
    - 如果设置为 volatile，那么读和写都是原子性的。（包括 long 和 double）
11. 会有死锁、活锁和 starvation 的问题。死锁解决方法可以是超时。

12. 可以会有 spin lock，会阻塞，但是不会有线程切换。但是会有问题，比如一个线程一直占用 CPU。（如果线程切换的开销大于了 spin lock 的开销，那么就适合用 spin lock）
13. 所以也可以用 wait 和 notify 来解决。wait 会释放锁，notify 会唤醒一个线程。notifyAll 会唤醒所有线程。
14. java 的自动垃圾回收通过 generation 来实现的，即新生代和老年代。新生代又分为 Eden 和两个 Survivor。老年代是存活时间长的对象。（就是一个 generation 满了之后，就会被移到下一个 generation，然后做垃圾回收）
15. Immutable Objects 就是不可变对象，即一旦创建就不能被修改。比如 String 类。这样可以避免多线程的问题。如果要修改、就要重新创建一个对象然后赋值。
16. 单单对函数进行 sync 还是会有问题，因为可能会有多个线程按不同的时序进入 sync 函数。（ppt 例子）
17. 所以需要写 final private 的方法。然后不提供 setter，因为不存在多线程写的情况，所以一定是线程安全的。
18. 有一个锁对象 Lock，分为可重入和不可重入。
19. 可重入的锁：ReentrantLock，即一个线程可以多次进入同一个锁。（即一个线程可以多次进入同一个锁，但是不能进入不同的锁）
20. 也有一个 executor，就相当于有一个线程池给所有的线程使用，不然的话就是会造成对象的垃圾回收。线程池会保持内存存在，只是值恢复初始化
21. executor interface 分为 3 种，即 Executor、ExecutorService 和 ScheduledExecutorService。Executor 是最基本的接口，接收 Runnable 类。ExecutorService 可以接受 runnable 和 callback。ScheduledExecutorService 可以延迟执行任务。
22. blur 操作，不停地将任务分解，直到任务可以被执行。即任务分解的过程。（fork / join task），在 java 里面叫做 RecursiveAction 和 RecursiveTask。这样就可以切成多线程多处理器并发执行了。
23. 这个 recursivetask 通过 override 的 compute 方法来实现任务的分解和合并。即分解成子任务，然后合并子任务。
24. 上面会涉及到 invokeAll 方法，把新建的线程通过线程池进行管理，即通过线程池的管理来执行任务。（已经封装好了）
25. 为什么要并发安全、因为不知道同时会有多少客户端连接，所以需要用线程安全的集合，同时写有问题。
26. atomic object 这种对象也可以通过非基本类型的原子性操作
27. 用途：每个线程生成不同的伪随机数
28. java 新特性，virtual thread，即轻量级线程，不需要操作系统的线程，而是在 JVM 里面实现的。这样就可以避免线程的切换开销。（相当于 n 个虚拟线程可以在一个真实线程里面运行）（本质就是要复用）（例子：服务器给每一个客户端分配一个虚拟线程）（有一个 try-resolve block，即先获取，再执行，try（）{}catch（）{}）

```
Thread.Builder builder =
   Thread.ofVirtual().name("worker-", 0);
```

29. 与之对应的是 platform thread，即绑定着操作系统的线程（平台线程）。会存在线程栈，持有很多资源，开销大。（操作系统里面的线程数是有限的）（linux 线程调用用的是红黑树，线程调度性能受树高影响）

07 cache

1. memcached，就是以键值对的形式存储数据在内存里面。
2. DBMS 是数据库管理系统。比如 mysql 里面也有 cache；spring 这里 JPA 里面也带有 cache，为什么还需要缓存 memcache 呢?
3. mysql 的 cache 是根据 sql 操作来进行存储 cache 的，他做不到。MySQL 内部的缓存主要包括多个不同层次和类型的缓存，旨在提高数据库性能和响应速度。比如查询缓存、查询结果缓存、InnoDB 缓冲池、表缓存、键缓存等。
4. 进程间通信要 martial（即序列化），因为内存是隔离的。（也会耗时）
5. 也可能 spring，memcache 和 db server 在不同的机器上。所以需要单独分离出一个 memcache。（memcache 甚至可以既缓存 mysql 的，又缓存 mongodb 的，而且存的是一个处理后的结果，而不是原始数据）
6. InnoDB 是 MySQL 的一个存储引擎，专门设计用于支持高性能和高可靠性的数据库应用。它是 MySQL 默认的存储引擎，并具有多个关键特性，使其适用于需要事务处理和高并发访问的应用场景。
7. InnoDB 的聚簇索引（Clustered Index）是该存储引擎的一种特殊索引类型，它将表的数据存储与索引结构结合在一起，优化了数据的检索效率。聚簇索引是将表中的数据行与索引存储在一起，数据的物理顺序与索引的顺序一致。换句话说，表的实际数据按照聚簇索引的顺序存储在磁盘上。聚簇索引使用 B+树结构，这意味着所有的叶子节点（leaf node）都包含了实际的数据行，而非仅仅是指向数据行的指针。这样可以通过索引直接访问数据行，提高查询性能。
8. 经常要写的数据、可以不用放在 memcache 里面
9. 书本的 isbn 和 title 可以放在 cache 里面，因为不经常变化。
10. 这个 cacheable 的注解可以写在 repository 的方法上。比如 getByIsbn，会先在 cache 里面看有没有对应的 isbn，如果有的话，就直接返回，如果没有的话，就去数据库里面找。（springcache，也要在 main 函数里面加上 @EnableCaching）
11. 比如 book 只有 1k，一个 person 因为含有 img，所以有 5M，怎么在 memcache 里面存储高效? 在 memcache 里面分成了很多种大小的 chunks。然后这个注解的名字空间可以辅助剪枝，减少找到内存块的时间。
12. 不同的 cache server 间的 node 不需要通信。（因为有哈希）
13. 如果有了多个 server 节点，需要做一致性哈希来存储和 get（解决增减服务器后数据位置的问题）（会涉及到一个环中的 chord 的概念，环中可能还有环）（好处、总体浪费空间不多，而且可以增加搜索性能）
14. 如果 server 上的 cache 满了，那么通过 LRU 来写到硬盘上。（然后这个硬盘是 memcache server 同一个的硬盘上）（然后硬盘也可能被写满，所以要限制硬盘写的空间大小）
15. redis：redis 也可以做主从备份，redis 里面的 list 是 link list，因为避免复制（所以顺序查找比较慢）
16. 为什么需要数据访问网关？data access gateway。（减少冗余代码，多种 daoimpl 调用同一个模板类）
17. redis 除了放类，还可以放消息（不需要 kafka，因为运行在 spring 外，所以实现异步通信，也不一定要用消息队列中间件，可以直接种 redis 缓存中间件）
18. 可以有不同的 redis 操作，比如 rightpush 和 set，可以用不同的操作来实现不同的功能（数据结构）
19. 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。

20. 但是这个 list 上面提到是链表、所以不用

08 Full-text Searching

1. 场景：搜索书评（非结构化数据）（remark 用 varchar 放、可变长，char 是定长）（或者用表外部的 text/blob 文件进行存储，用指针指向）
2. 如果用 sql 的 like 加%%，本质还是一个一个读出来再对比，效率低，会很慢。
3. 还有一种方法是结构化部分的关键词、作为列或者行。（但是看似能做结构支撑、但是还需要做结构化）
4. 所以需要做一个反向定位的索引，比如搜关键词然后返回评论的位置。
5. 所以有四个问题，一个是 keyword 进去之后、如何返回哪些 doc；还有一个是返回的 doc 如何排序。（相似度和频率）；还有一个是返回的数据为什么是 title+url+部分内容+author，而不是全部（即索引维护的内容究竟是哪些部分，可以是只对 title 进行索引，其他文本内容只是为了快速返回内容）；这个索引在做维护的时候、能不能 incremental 地做维护。
6. Lucene 是一个全文检索引擎，可以用来做搜索。（主要做非结构化数据文本的搜索，比如可以是来自文件、数据库、网页等等）
7. 因为有各种各样的数据、所以需要一个适配器 adapter，来解析各种数据。
8. 为了变快，需要将索引建成支持快速扫描的结构。
9. .fnm 文件：有一个索引的域（field），可以是（支持 index 的） subject、content、author 等等。（或者是支持向量化的 vectored 的 subject，用余弦相似度计算）
10. .tis 文件：维护域里的 value 在几个文档里面出现过。
11. .frq 文件：维护域里的 value 在每个文档里面出现的次数。
12. .prx 文件：维护.frq 文件中的 frequency 在文档中的位置。
13. 以上这个就是一种反向索引的结构。即不需要通过文档的一行行匹配来对比。可以直接通过关键词定位到文档的位置。
14. 会涉及到 precision 和 recall 的 balance 的问题，即精确度和召回率。精确度是指检索出来的结果中有多少是正确的，召回率是指所有正确的结果中有多少被检索出来了。
15. 有些内容是不需要参与索引的构建的（优化）
16. lucene analyzer 是用来分词的，比如中文分词器、英文分词器等等，把 text 分成一个个的 token。其中也有 stop analyzer，即停用词分析器，可以把一些无意义的词去掉。
17. lucene codecs： 是用来对索引进行编码的，比如可以用来压缩索引。
18. lucene docuemnt 是用来存储索引的，即一个 document 里面有很多个 field，每个 field 里面有一个 value。
19. 给的示例代码的例子、即对应概念中的域
20. 其中的数值索引、可以支持范围搜索或者精确匹配
21. 建完索引之后会有一个 segment 文件生成，相当于就把 lucene 的索引分成了很多建好的索引段（索引的索引）（因为索引可能会比较大、所以就支持了在不同的机器上面存）
22. document number 是 lucene 里面的一个概念，即每个 document 有一个编号，可以通过这个编号来找到 document。（这个编号是唯一的）
23. 相似度是为了最终进行排序
24. Similarity 是用来计算相似度的，即计算文档和查询的相似度。（比如 BM25Similarity）

25. TF-IDF 是一种计算相似度的方法，即 term frequency 和 inverse document frequency。即一个词在文档中出现的次数和在所有文档中出现的次数的比值。(即出现多的次数可能是在很多文档中出现了很多次、说明是一个常用词，所以权重就会降低)
26. 由于 lucene 使用比较复杂，所以有了 solr，即一个基于 lucene 的搜索服务器，可以用来搜索、存储、索引文档。
27. 除此之外、还有 elasticsearch。是近实时的。（不用写 sql，建立反向索引）
28. cache 里面的对象可以被多个进程读
29. **索引的基本概念**:
    搜索引擎的核心概念是**索引**。索引是一个高效的交叉引用查找结构，用来加速搜索过程。通过将原始数据（如大量文本文件）转换为索引，可以避免逐个扫描文件，从而快速找到包含某个单词或短语的文件。
30. **原始搜索方法的局限性**:
    如果没有索引，搜索引擎可能需要逐个扫描每个文件来查找特定的单词或短语，这种方法在处理大量文件或大文件时非常低效，无法满足快速搜索的需求。
31. **索引的作用**:
    索引就是将文本数据转化成一种能够快速访问单词的格式，这样就能避免逐一扫描每个文件。它是一种数据结构，允许我们快速随机访问存储其中的单词，并进行搜索。
32. **搜索过程**:
    搜索就是通过查找索引中的单词，找到包含这些单词的文档。这大大提高了搜索效率。
33. **搜索质量的衡量标准**:
    搜索的效果通常通过**精确度（Precision）**和**召回率（Recall）**来衡量:
    - **召回率（Recall）**：衡量搜索系统能否找到所有相关文档。
    - **精确度（Precision）**：衡量搜索系统能否排除无关的文档。
34. **其他搜索考虑因素**:
    除了精确度和召回率，搜索系统还需要快速地处理大量文本，支持单词查询、多个词查询、短语查询、通配符、结果排名和排序等功能。此外，用户输入查询时的语法友好性也是重要的考虑因素。
35. **Lucene API 的包结构**:
    Lucene API 被划分为多个包，每个包都有特定的功能:
    - **org.apache.lucene.analysis**：负责文本分析。分析的核心是通过将文本从一个 `Reader` 转换成 `TokenStream`。`TokenStream` 是一个包含多个标记（tokens）的序列，标记代表文本中的基本单位（如单词、符号等）。`TokenStream` 通过 `Tokenizer`（分词器）和 `TokenFilter`（过滤器）来构建。`Tokenizer` 将文本拆分成初步的标记，`TokenFilter` 可以进一步修改这些标记。
      - **分析器（Analyzer）** 是将这些分词器和过滤器组合在一起的工具。
      - `analysis-common` 包含了一些常见的分析器实现，比如 `StopAnalyzer`（停用词分析器）和基于语法的 `StandardAnalyzer`（标准分析器）。
    - **org.apache.lucene.codecs**：提供了对倒排索引结构的编码和解码的抽象。它为索引的存储和读取提供不同的实现方式，可以根据应用需求进行选择。
    - **org.apache.lucene.document**：提供了 `Document` 类。`Document` 是一组命名的字段（`Field`），这些字段的值可以是字符串或 `Reader` 类型的实例。
    - **org.apache.lucene.index**：提供了两个主要类:
      - `IndexWriter`：用于创建索引并将文档添加到索引中。
      - `IndexReader`：用于访问索引中的数据。

- **org.apache.lucene.search**：提供了表示查询的结构（如 `TermQuery` 用于单个单词查询，`PhraseQuery` 用于短语查询，`BooleanQuery` 用于布尔查询组合）以及 `IndexSearcher`，它将查询转换为 `TopDocs`，即搜索结果。
  - `QueryParser` 帮助将字符串或 XML 格式的查询转换为查询结构。
- **org.apache.lucene.store**：定义了一个抽象类 `Directory`，用于存储持久化数据。`Directory` 是一个由 `IndexOutput` 写入并由 `IndexInput` 读取的命名文件集合。
  - 不同的存储实现可以使用，`FSDirectory` 是最常用的，它高效地使用操作系统的磁盘缓存。
- **org.apache.lucene.util**：包含一些有用的数据结构和工具类，例如 `FixedBitSet` 和 `PriorityQueue`。

36. **如何使用 Lucene：**
    使用 Lucene 时，应用程序通常需要以下几个步骤：
    1. **创建文档**：通过添加字段（`Fields`）来创建 `Document`。
    2. **创建 IndexWriter**：使用 `IndexWriter` 将文档添加到索引中，使用 `addDocument()` 方法。
    3. **创建查询**：通过 `QueryParser.parse()` 从字符串构建查询。
    4. **执行搜索**：创建 `IndexSearcher` 并调用其 `search()` 方法执行查询。
37. **基本概念：**
    - **索引（Index）**：一个索引包含一系列文档。
    - **文档（Document）**：一个文档是多个字段的集合。
    - **字段（Field）**：一个字段是一个命名的术语序列。
    - **术语（Term）**：术语是一个字节序列。术语可以是文档中某个字段的内容，经过分析处理后得到。

    术语的区别：同样的字节序列，如果出现在不同的字段中，Lucene 会将它们视为不同的术语。因此，术语在 Lucene 中是以一个“字段名称”和“字段内字节”的组合来表示的。
38. **倒排索引（Inverted Indexing）：**
    - Lucene 的索引通过存储术语及其相关统计信息来提高基于术语的搜索效率。Lucene 的术语索引属于 **倒排索引** 的一种类型。
    - **倒排索引**：它能够列出包含某个术语的所有文档，而不是列出一个文档包含的所有术语。这种方式是自然文档中列出术语的关系的反向。
39. **字段的类型：**
    Lucene 中的字段分为两类：存储字段和索引字段：
    - **存储字段（Stored）**：字段的内容原样存储在索引中，以便在搜索结果中返回。
    - **索引字段（Indexed）**：字段的内容经过分析处理并转化为术语，便于进行搜索。

    一个字段可以同时是 **存储** 和 **索引** 的，即同时保留原始数据和经过分析处理的数据。

    字段的文本可以通过 **分词（Tokenization）** 转化为术语进行索引，也可以直接用文本作为术语进行索引。大多数字段会被分词处理，但对于某些标识符类型的字段，有时需要原样索引。
40. **Lucene 中的字段类型：**
    这部分列举了许多专用的字段类型，它们的用途各不相同，例如：
    - **TextField**：适合用于全文搜索，索引整个文本内容。
    - **StringField**：将字符串按字面索引，作为单一的术语。

- **IntPoint, LongPoint, FloatPoint, DoublePoint**：分别用于精确和范围查询的整数、长整数、浮动点数和双精度浮动点数。
- **SortedDocValuesField**：用于排序或分面查询的字节数组字段。
- **StoredField**：仅存储字段的值，以便在结果中返回。

每个字段都有三个部分：名称、类型和值。字段的值可以是文本（如 `String`、`Reader` 或预分析的 `TokenStream`）、二进制数据（如 `byte[]`）或数字（如 `Number`）。

41. **段（Segments）**
    Lucene 索引通常由多个 **子索引**（即"段"）组成。每个段是一个 **独立的索引**，它可以单独进行查询。
    - **索引的演变**：
      - **创建新的段**：当添加新的文档时，Lucene 会为其创建新的段。
      - **合并现有的段**：Lucene 会定期合并多个段，以优化性能。段的合并可以减少查询时涉及的段的数量，从而提升查询效率。
    - **搜索可能涉及多个段和/或多个索引**：每个索引可能由多个段组成，搜索时可能需要同时查询多个段或多个索引。
42. **文档编号（Document Numbers）**
    在 Lucene 中，每个文档都有一个 **整数编号**。编号是按顺序分配的：
    - 第一个文档的编号是 **0**，第二个文档是 **1**，依此类推。
43. **段索引的内容**
    每个段的索引包含以下信息：
    - **Segment info（段信息）**：包含关于该段的元数据，如该段包含的文档数量、使用的文件、以及该段的排序信息。
    - **Field names（字段名）**：该段索引中使用的所有字段的名称元数据。
    - **Stored Field values（存储字段值）**：每个文档的存储数据，包括字段名及其对应的值（例如，文档的标题、URL 或数据库访问标识符）。这些存储的字段值是搜索结果中返回的数据。每个存储字段都通过文档编号来访问。
    - **Term dictionary（术语字典）**：包含所有文档中所有字段使用的术语。字典中还包含每个术语出现的文档数量，并指向术语的频率和接近度数据。
    - **Term Frequency data（术语频率数据）**：对于字典中的每个术语，记录包含该术语的文档数量以及该术语在每个文档中的频率。这个数据是可选的，具体取决于索引时是否启用频率数据（通过 `IndexOptions.DOCS` 配置）。
    - **Term Proximity data（术语接近度数据）**：记录术语在每个文档中的位置。这个数据对于文本分析和搜索排序很重要，特别是在短语匹配时。若所有文档的字段没有位置数据，接近度数据将不会存在。
    - **Normalization factors（归一化因子）**：每个字段在每个文档中的值，这些值在搜索时会被乘入得分计算，以影响该字段的相关性评分。
    - **Term Vectors（术语向量）**：每个文档的每个字段可以包含术语向量。术语向量记录了每个术语及其频率。术语向量通常用于支持高效的文本分析和查询。
    - **Per-document values（每文档值）**：这些值类似于存储字段值，但它们通常加载到内存中以便快速访问。与存储字段值不同，它们更多用于快速访问特定文档的信息，尤其是当这些值影响文档评分时。
    - **Live documents（活跃文档）**：这是一个可选的文件，指示哪些文档是"活跃"的，意味着它们没有被删除或标记为过时。

- **Point values（点值）**：用于存储维度索引的字段，以便快速进行数字范围过滤和处理大型数字值（如 `BigInteger` 和 `BigDecimal`）。这也适用于地理数据的处理（如 2D 或 3D 地理位置数据）。
- **Vector values（向量值）**：用于存储数值向量的格式，优化了随机访问和计算，支持高维的近邻搜索。这通常用于机器学习和自然语言处理应用中。

44. **搜索基本概念**：
    - **IndexSearcher.search(Query, int)**：应用程序通过调用此方法来提交查询，Lucene 会开始对文档进行评分和检索。
    - 查询处理的关键部分是通过**Weight**实现类及其**Scorer**或**BulkScorer**实例来执行。
45. **查询类**：
    - **TermQuery**：最简单且最常用的查询类型。它会匹配包含特定术语（Term）的文档，术语指的是某个字段中的一个词。构建时非常简单，例如：`TermQuery tq = new TermQuery(new Term("fieldName", "term"));`，它会查找所有字段名为“fieldName”且包含“term”词的文档。
    - **BooleanQuery**：可以将多个**TermQuery**组合成一个布尔查询，支持不同的逻辑操作：
        - **SHOULD**：匹配文档中可以出现但不强制的子查询。
        - **MUST**：匹配文档中必须出现的子查询。
        - **FILTER**：匹配文档中必须出现的子查询，但不影响评分。
        - **MUST NOT**：匹配文档中不能出现的子查询。

      如果布尔查询中包含过多的子查询（即布尔子句），可能会导致**TooManyClauses**异常。
    - **PhraseQuery**：用于查找包含特定短语的文档。短语匹配使用**slop**（偏移量）来允许词语之间有一定的间距，默认情况下 slop 为 0，表示短语必须完全匹配。
    - **MultiPhraseQuery**：更一般化的短语查询，允许在短语中的某个位置使用多个词语，支持像同义词的替代。
    - **PointRangeQuery**：用于匹配数字范围内的文档，支持整数、长整型、浮点型或双精度型字段。
    - **PrefixQuery, WildcardQuery, RegexpQuery**：
        - **PrefixQuery**：查找以特定字符串开头的词。
        - **WildcardQuery**：支持使用`*`（匹配 0 个或多个字符）和`?`（匹配一个字符）作为通配符。
        - **RegexpQuery**：更通用的正则表达式查询，支持复杂的模式匹配。
    - **FuzzyQuery**：通过**Levenshtein 距离**（即编辑距离）来匹配与指定术语相似的文档，适用于处理拼写错误或变异的情况。
46. **Lucene 支持的评分模型**：
    - **向量空间模型（VSM）**：通过将文本表示为向量来计算相似度。
    - **概率模型**：如 Okapi BM25 和 DFR，它们使用概率来评估文档的相关性。
    - **语言模型**：基于统计的语言生成模型来衡量文本的相关性。

    这些模型可以通过**Similarity API**插入，并提供扩展钩子和调优参数以进行优化。
47. **评分计算依赖于索引方式**：
    评分是基于文档的索引方式，因此理解索引过程非常重要。可以使用**IndexSearcher.explain(Query, doc)**方法查看某个匹配文档的评分是如何计算出来的。

48. **查询与相似度的作用**：
    通常，**Query**决定哪些文档匹配（这是一个二元决策），而**Similarity**决定如何为这些匹配的文档分配评分。
49. **字段和文档**：
    在 Lucene 中，我们评分的对象是**文档（Document）**。每个文档由多个**字段（Field）**组成。Lucene 的评分是基于字段的，然后将各个字段的评分结果合并，最终返回文档。即使两个文档内容相同，如果一个文档的内容分布在两个字段中，另一个文档只在一个字段中，它们的评分也可能不同，因为 Lucene 在计算评分时会考虑**长度归一化（length normalization）**。
50. **评分提升**：
    Lucene 允许通过使用**BoostQuery**来调整查询中不同部分的评分贡献，进而影响最终的评分结果。
51. **改变评分公式**：
    更改**Similarity**是影响评分的一个简单方法，可以在索引时通过**IndexWriterConfig.setSimilarity(Similarity)**进行设置，也可以在查询时通过**IndexSearcher.setSimilarity(Similarity)**来实现。要确保查询时使用与索引时相同的**Similarity**，以确保规范化参数（norms）在编码和解码时的一致性，Lucene 不会自动验证这一点。
52. **内置的 Similarity 实现**：
    - **BM25Similarity**：这是 Okapi BM25 模型的优化实现。
    - **ClassicSimilarity**：这是 Lucene 最初的评分函数，基于向量空间模型（Vector Space Model）。
    - **SimilarityBase**：提供了 Similarity 契约的基本实现，并暴露了简化的接口，非常适合用作新排名函数的起点。
53. **Elasticsearch 是一个高度可扩展的开源全文搜索和分析引擎**：Elasticsearch 是一个非常强大且开源的搜索和分析工具，能够快速处理大量数据，适用于大规模的搜索和分析任务。
54. **它允许你快速并接近实时地存储、搜索和分析大量数据**：Elasticsearch 能够高效地存储和查询海量数据，而且搜索结果几乎是实时的。也就是说，数据存入之后，一般在 1 秒钟内就可以被搜索到。
55. **它通常作为支持具有复杂搜索功能和需求的应用程序的底层引擎/技术**：Elasticsearch 常用于构建那些需要复杂搜索功能的应用系统，是这些应用的技术核心。
56. **近实时（Near Real Time）**：Elasticsearch 是一个近实时的搜索平台。意思是从你将一个文档索引（存入）到它可以被搜索到之间，通常会有约 1 秒钟的延迟。
57. **文档（Document）**：文档是可以被索引的基本信息单元。每个文档以 JSON 格式表示，JSON 是一种广泛使用的互联网数据交换格式，便于数据存储和交换。
58. **索引（Index）**：索引是具有相似特征的文档集合。每个索引都有一个名字（必须全部小写），这个名字在执行索引、搜索、更新和删除操作时用来引用该索引中的文档。

09 Web Services

1. 原始的前后端交互通过 restcontroller、但是 CRUD 每个方法都有一个 url pattern，这样好不好?
2. 第一个问题：是否只能 http 协议？（有很多协议、比如 smtp、ftp、sftp）还是说用 method 或者用 data 驱动?
3. 在跨语言的时候应该用什么时候调用？（同 java 语言的时候用 rmi 调用）
4. 在 www 的网络之下、仍然能访问到 alipay（在广域网上，对于 RMI 完全不可能）
5. service 不仅仅是一个 method 或者是一个 function（实现解决跨语言、操作系统、数据库的问题、通过 service 进行屏蔽）
6. REST：传递数据

7. SOAP：传递方法的调用（方法的签名、方法的参数列表）Simple Object Access Protocol（基于 xml）
8. 主要的内容在 body 里面、包含了对一个方法的描述

```
public interface TravelAgent {
   public String makeReservation(int cruiseID,
         int cabinID, int customerId, double price);
}
```

9. WSDL: Web Service Description Language（描述一个 web service 的语言）, 通过 wsdl 可以生成客户端代码 （可以自动生成）
10. port 可以看作一个接口类、然后可以暴露出不同协议的方法

```
<message name="RequestMessage">
        <part name="cruiseId" type="xsd:int" />
        <part name="cabinId" type="xsd:int" />
        <part name="customerId" type="xsd:int" />
        <part name="price" type="xsd:double" />
</message>
<message name="ResponseMessage">
        <part name="reservationId" type="xsd:string" />
</message>
<portType name="TravelAgent">
   <operation name="makeReservation">
      <input message="titan:RequestMessage"/>
      <output message="titan:ResponseMessage"/>
   </operation>
</portType>
<!--binding element tells us which protocols and encoding styles are used -->
<binding name="TravelAgentBinding" type="titan:TravelAgent">
 <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="makeReservation">
     <soap:operation soapAction="" />
     <input>
      <soap:body use="literal" namespace="http://www.titan.com/TravelAgent"/>
     </input>
     <output>
      <soap:body use="literal" namespace="http://www.titan.com/TravelAgent"/>
     </output>
 </operation>
</binding>
```

11. 这个 style="rpc" 也有很多种，比如 document、rpc、literal、encoded
12. 通过 binding 可以指定协议、比如 http、ftp、smtp
13. 这个 wsdl 文件有什么作用？比如说有一个接口的实现类，可以对着接口生成一个 wsdl 文件，这个文件是其他端，比如说 c# 可以访问的，然后 c# 拿到这个 wsdl 文件之后还会有一个实现类会把这个 wsdl 文件转换成一个类（这个类并不知道业务逻辑，相当于做的是封装和解析 request 和 response 消息的逻辑），然后这个类就可以调用这个接口的方法，然后通过 soap 把这个方法调用传递给 java 端，就可以通信了。
14. 但是要知道有哪些服务可用、所以会有一个第三方的注册表、比如 java 注册 wsdl、然后 c# 从注册表里面找到对应的服务。（屏蔽掉具体应用的位置）（ppt 上有图）

15. header 里面要指定为 xml type
16. 示例代码中、producer 暴露服务、consumer 调用服务
17. 有不同的方法、jaxb 和 jax-ws
18. SOAP-based Web Services （耦合度高）

- Coupling with the message format
- Coupling with the encoding of WS
- Parse and assemble SOAP （解析有开销）
- Need a WSDL to describe the details of WS（需要额外的文件去描述）
- Need a proxy generated from WSDL（需要代理去解析）

It is a time-cost way to implement Web Service with SOAP

- We should find a new way to implement WS

19. REST：REpresentational State Transfer（数据驱动）

Representational:

- All data are resources. Representation for client. （所有数据在互联网上唯一标识、可以用 uri 拿到数据）
- Each resource can have different representations （拿到的数据应该显示的是什么、怎么去呈现、由客户端决定）
- Each resource has its own unique identity(URI)

State:

- It refers to state of client. Server is stateless. （服务器端无状态）
- The representation of resource is a state of client.

Transfer: （在访问不同资源的时候、前面的状态会被传递）

- Client’s representation will be transferred when client access different resources by different URI.
- It means the states of client are also transferred.
- That is Representation State Transfer

应该叫/person/1，而不是/findPerson/1 (这个是一个动作)（怎么去改 CRUD 呢）

20. 这个 CRUD 的方法应该根据 http 的 method 来区分，比如说 get、post、put、delete
21. 所以是靠 url 做设计、然后用 method 去区分 CRUD ，这样才算是 restful，按照实例来。
22. 这个只是类似 标记，可以自定义。
23. 实现解耦、前端对修改无感知（这个和前面的 soap 作比较，前后端绑定）
24. 因为是 restful、所有返回的是 json
25. 所以对于不同语言交互的解决方法
    - 第一种是 xml 描述 request 和 response，和方法
    - 第二种是通过 http 的 crud 方法，然后返回 json
26. 对于广域网的解决方法
    - 上面两个方法就能解决。

27. alipay 是 int 到分的金额
28. Advantages:
    - Across platforms （跨平台）
      XML-based, independent of vendors
    - Self-described （自描述）
      WSDL: operations, parameters, types and return values
    - Modulization （模块化）
      Encapsulate components
    - Across Firewall （跨防火墙）
      HTTP

    Disadvantages:
    - Lower productivity
      Not suitable for stand-alone applications （单体不行）
    - Lower performance （解析有开销，同一个进程就不用 http 了）
      Parse and assembly
    - Security （安全性需要额外的机制）
      Depend on other mechanism, such as HTTP+SSL
29. 最后还讲了 ssl 的非对称加密。

10 Microservices & Serverless

1. 不应该将搜索结果的 key 和 result 存在 redis 里、因为会有维护一致性时的问题。
2. 前端来了 query 之后、不是马上执行的、会先把 query 变成一个 query plan、然后再执行。（可能 query1 和 query2 显式地不一样、但是生成的 query plan 是一样的，抽象语法树是一样的、才会使用 mysql cache）
3. 将 ip 和端口注册到注册中心里，然后在注册中心里通过服务名找到对应的 ip 和端口，而不是直接访问 ip 和端口。
4. 然后需要一个 gateway、通过 gateway 以 name 的方式访问服务，位置由 gateway 去查注册中心得到
5. 让 client 和其位置解耦。
6. gateway 还会有一个负载均衡的策略、比如说轮询、随机、权重、最少连接数。
7. 独立数据库可以提高容错性，但是一个流程可能需要访问到多个数据库。通信的开销变大。（性能受损）
    - 所以引入消息队列、前后端采用异步通信的方式去弥补。（只是能马上得到响应、但不是结果）
    - 也能屏蔽位置、因为通过消息队列中的 topic 通信。
8. 系统分离部件越多、越不可靠（越容易出错）
    - 所以需要事前监控、事后分析
    - 需要容错和服务降级
9. 定位问题：链路追踪
10. 其实还是从 eureka 里面拿到服务注册在里面的位置、拿到这个位置之后、再去用这个位置直接调用这个服务。
11. 任何服务默认注册进去是一个集群。

12. 前端发请求发到 gateway、gateway 通过 eureka 拿到服务的位置、然后调用服务。
13. gateway 提供统一的访问网关入口，eureka 提供和位置解耦（如果重启了，位置会变，eureka 会自动把服务注册进去，但是访问的代码不需要改动）
14. 没有 gateway 也可以、但是有了之后、只需要改写配置的 yml 文件就可以了，而不需要修改代码。
15. 函数式服务的无状态指的是什么？函数式服务为什么容易扩展？
    - 状态：一种是在内存里、比如 session、还有一种是在数据库里、比如订单状态
    - 对于有状态的、比较难维护
    - 无状态的：它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、所以可以很容易地扩展
    - 而且执行完一个函数之后、不会对后续的函数进行影响
    - 所以不需要知道之前执行了什么、服务器端就不需要维护状态（不需要配数据源之类的）
16. 一种方案是状态推到客户端去维护、一种是集中在服务端维护
17. 无状态的服务是事件驱动的。不需要关联服务器基础设施的状态。
18. spring cloud function 是一个函数式编程的框架、可以用来开发无状态的服务。
19. 需要包在 function 和 flux 里面、然后通过 spring cloud stream 来调用。（直接通过函数名访问）
20. 也可以通过 compose 进行函数的组合
21. 这个配置的时候、每个都需要配置一下 cloud starter eureka 包。
22. 这个参数如果是一个原生类的话、貌似在 requestbody 里面不需要用 json 格式、比如说就一个字符串、直接传就可以了。但是需要加上@requestbody
23. 这个 gateway 需要配置 cors、在配置文件里设置
24. 由于每次都会处理一次跨域、所以每一轮处理的时候会在响应头里面再加一层 cors 的头，正常情况下基于以上配置即可，但是由于目前的项目的下游微服务也配置了可以跨域的相关配置，这就导致返回的 ResponseHeader 中有多重属性，这个多重属性浏览器是不认的。所以基于此的处理方法，把下游的所有 cors 配置都取消就可以了。
25. Function 和 Flux 是来自于 Java 和 Spring 生态系统中的重要概念，特别是在响应式编程中。下面是对它们的详细解释：

    1. Function
       Function<T, R> 是 Java 8 中的一个函数式接口，它表示一个接受一个输入参数并生成一个结果的函数。在你的代码中，Function<Flux, Flux> 和 Function<Flux, Flux> 代表接受一个 Flux 类型的输入，并返回另一个 Flux 类型的输出。

    Flux：表示一个异步流，可以发出 0 到 N 个字符串值。
    Flux：表示一个异步流，可以发出 0 到 N 个整数值。 2. Flux
    Flux 是来自 Project Reactor 的一个类，表示一个响应式流，可以包含多个元素。它是响应式编程的核心概念之一，支持流式数据处理。Flux 可以处理从 0 到 N 个元素的异步序列，通常用于处理流式数据或事件。

    在你的代码中，flux.map(value -> value.toUpperCase()) 和 e.map(value -> value * 2) 等操作是对流中每个元素进行变换的过程。

26. Flux 和 Mono 都是 Reactor 框架中的响应式类型，但它们的用途有所不同。

    主要区别：
    数据数量：

Mono：表示零或一个元素的异步序列，适用于单一结果（如单个用户查询）。
Flux：表示零或多个元素的异步序列，适用于多个结果（如查询多个用户）。
使用场景：

使用 Mono 时，你期待的是一个单一的响应，比如从数据库获取一个用户信息。
使用 Flux 时，你可能在处理列表、流或者多项事件，比如从数据库获取所有用户或实时消息流。

27. 它这个 maven 默认是不带 web 的、所以需要加上 web 的依赖
28. 貌似这里会有一个问题、使用 redis 的时候、这个下订单的操作没法正常进行。这个错误通常表示在使用 JPA（Java Persistence API）时，您尝试删除或修改一个实体集合，但该集合的拥有者实例已经被分离（detached）或者没有正确管理。具体来说，当您使用 cascade="all-delete-orphan" 时，JPA 期望对拥有者实体（如 Book）的集合（如 cartItems）进行正确的引用和管理。
29. 这个函数计算应该在前端调用还是在后端服务里调用呢？貌似应该在前端调用啊
30. 这个函数容器在 cloud 被调用的时候、会看有没有这个容器、如果没有会马上创建一个容器、调用完之后会被马上关掉，生命周期很短
31. 或者可以把容器的操作系统层的进行保留、然后根据倒计时只更换函数层部分的代码、或者只更换语言层的环境，可以节省容器重启，做初始化环境的时间。
32. 由 spring cloud 管理需要多少个函数服务实例去处理请求（因为请求可能会有很多个）
33. 说是 gateway 和 register 不是必须的、如果重启之后位置不会变的话。
34. 因为需要通过高层的 http 协议、所以微服务的性能会受影响。
35. session 首先未必需要维护在后端、也可以维护在前端的 kvmap 和 cookie 里面。还有一种方法是通过额外的 session server 去维护

11 MySQL Optimization

0. 写进去的 sql 语句会通过 query plan 进行类似编译的优化、然后再进行查询，会将查询的结果存到 cache 里面。
1. 下面的一些是数据库层面的优化。
    - 存储的内容会有优化关系，比如学号应该存 Integer 还是 String，如果存 Integer，这个就会节省内存。或者是 char 和 varchar 的区别
    - 存储的索引应该建立到哪个列上面、还是建立几个列的索引?
    - InnoDB 的存储引擎是什么？不是 mysql 的原生引擎
    - 行格式的问题？比如 varchar（10），那就是 0-10 都可以，如果它里面只存了 2 bytes，那么剩下的 8 bytes 怎么处理？是应该等长的还是节省空间？或者如果学号的前缀都是相同的、是不是就可以压缩一下，是不是动态的。（空间和时间的权衡）（取决于经常发的 query 是什么样）
    - 锁级别?
    - cache 的大小？而且这里的 cache 也不是一整块的、而是分开来的。
2. 还可以做一些硬件层面的优化
    - Disk seeks. （如何寻找空闲区域）
    - Disk reading and writing. （ssd 和 hdd）
    - CPU cycles.
    - Memory bandwidth.
3. 还有 sql 语言的可移植性，需要包装

4. 索引用 B，B+树，因为读的内容大小和 pagesize 差不多，而且可以范围查询（有指向兄弟节点的指针）
5. 当更改索引指向的内容的话、需要花费额外的时间去维护索引。
6. 如果 scan 全表的时间比通过索引找到、然后再改动的时间还要少的话、那就不需要做索引，扫描一下就可以。（因为会有建索引和维护索引的开销）
7. 大多数 mysql 索引都在 B-树里面，除了：
    - 像空间索引这样的、就会用 R-trees
    - 内存表也支持哈希索引
    - 全文索引用的是反向列表
8. 聚簇索引：每个表只有一个（默认上是在主键上建立）（可以在经常做搜索的内容上面设置聚簇索引）
9. 主键优化：
    - 如果主键由多个列构成、那么会让这个索引的尺寸增长得很快。
    - 如果有空的内容作为索引、会从空的那个地方、延长出一个链表，降低索引效率，所以最好不要为空。是否为空的实现是通过最后一个 bit 的 0 和 1 来实现的、每次都会检查，会消耗时间。
10. 空间索引：
    - 把空间分为了很多个区域、对区域去做索引（八叉树）
11. 外键关联：
    - 为什么需要外键：会有多张表，之间有关联
    - 为什么会有多张表？因为一张表中每一列的详情使用频率会比其他的更高，比如 book 的信息
    - 第一个好处、可以从 hdd 上面一次读一个会被经常使用的 page
    - 第二个好处、不经常使用的数据可以存储在其他地方
    - 第三个好处、可以复用数据、冗余数据会比较少，表比较小、搜索速度就比较快（IO 更少）
12. 索引：
    - 如果这个字段的长度太长了怎么办，可以把这个前面的 n 个字符作为索引（对于 blob 和 text 是强制需要在创建索引的时候确定的）
    - 会有四种格式：
        - REDUNDANT （留下多余的空间）
        - COMPACT （压缩空闲的空间）
        - DYNAMIC
        - COMPRESSED (只存不一样的)
    - 如果 search 的内容超过了索引的长度、可以先去根据 prefix 过滤、然后扫描剩下的
    - TF 和 IDF （频率和频率的倒数）
    - LIKE 的时候也可以用到
    - 哈希索引对范围查找不太行，但是判断等或者不等很快
    - 也可以做升序降序索引，虽然只有两个字段、可以根据升降做不同的索引组合（可以有 4 种）
13. 多列索引：
    - 索引建立的顺序也会有影响，决定因素应该是到时候这个索引会被使用的方式有关
    - INDEX name (last_name,first_name)

- 如果建立了上面这种索引，可以先 select lastname 在前面的，会利用到这种索引（ppt 上有例子，除了 lastname or firstname）

14. You can get better performance for a table and minimize storage space by using the techniques listed here:
    - Table Columns （比如 Int 和 Mediumint，或者设置为非空、因为会有额外的位和检查的开销）
    - Row Format（上面提到的四种 ROW Format，Redundant 已经不用了）（而且会有编码方式的不同、用 unicode 还是 utf8）
    - Indexes （主键应该越短越好）（一种主键是 Int 递增、8bytes，一种是 UUID，32 位 Hex。需要后面一种的原因是、因为前面保证主键递增需要额外的开销，比如再来一个线程，当用户大量的时候、就会有问题。实现是通过两个 agent，每次给 agent 分配一段的 id，比如分配 1000-1999 和 2000-2999，直到用完的时候再请求新的）（但是 UUID 是通过 IP:port:timestamp:random 的方式产生的，生成策略简单，好处是唯一的、但是坏处是太长、而且不知道插入的顺序）（维护索引节点的分裂和调整会有开销、要减少这个操作数）（分列做索引、效率不一定高、不如单列）（如果用了多列索引、那么最好把能筛选掉多的内容把它放到前面）（为什么要用 prefix 做索引、因为短的索引更快，因为读的 page 里面包含的索引更多 Shorter indexes are faster, not only because they require less disk space, but because they also give you more hits in the index cache, and thus fewer disk seeks. ）
    - Joins （如果需要对两个表做 join、而且两个表里的内容和含义和数据都是一样的、那么列名和数据类型应该相同、不然 join 之后就会慢 For example, in a table named customer, use a column name of name instead of customer_name. ）
    - Normalization（范式化、在 3 范式和 BCNF 范式里面选择，4 范式会做频繁的 join，范式低会提高维护数据一致性的开销、而且造成数据冗余）
    - 使用数值型的类型去存(Since large numeric values can be stored in fewer bytes than the corresponding strings, it is faster and **takes less memory** to transfer and compare them.If you are using numeric data, it is **faster** in many cases to access information from a database (using a live connection) than to access a text file. Information in the database is likely to be stored in a more compact format than in the text file, so accessing it involves **fewer disk accesses**. You also save code in your application because you can **avoid parsing the text file to find line and column boundaries**.)
    - 使用字符时、需要用相同的字符集（unicode、utf8）（尺寸小于 8kb，用 varchar，大于 8kb，用 blob，varchar 是存在一个 page 的相邻的内存块里、如果大于一个 page 的一半的话、就会报错，但是 blob 是存一个指针、然后在另外的地方存数据 ）（临时的内存表里面不能存 BLOB）（如果有很多记录不需要读、放到另外的表里面、保证单条记录小）

15. 对于打开的表的优化：
    - 打开的很多表里面会有默认的系统的表
    - 会有多版本的数据控制，通过 MVCC（多版本并发控制）来实现
    - table_open_cache 是打开的表的数量，根据 Max_connections 来设置(For example, for 200 concurrent running connections, specify a table cache size of at least 200 * N, where N is the maximum number of tables per join in any of the queries which you execute. You must also reserve some extra file descriptors for temporary tables and files.)
    - If table_open_cache is set too high, MySQL may run out of file descriptors and exhibit symptoms such as refusing connections or failing to perform queries.(如果设置太高、会有文件描述符的问题)

- 怎么做表的打开和关闭？用的不是 LRU，是把内容放到中间？（MySQL closes an unused table and removes it from the table cache under the following circumstances:When the cache is full and a thread tries to open a table that is not in the cache.When the cache contains more than table_open_cache entries and a table in the cache is no longer being used by any threads.When a table-flushing operation occurs. This happens when someone issues a FLUSH TABLES statement or executes a mysqladmin flush-tables ormysqladmin refresh command.）
- 可以展示当前打开的表的数量（会有多版本的控制、实际没有那么多）SHOW GLOBAL STATUS LIKE 'Opened_tables'；（If the value is very large or increases rapidly, even when you have not issued many FLUSH TABLES statements, increase the table_open_cache value at server startup.）
- 为什么会建立临时表？比如 union 的时候、会建立临时表，因为 sql 就是进行关系运算，需要额外的存储内存
- mysql 对数据库个数和表的个数都没有上限。（但是操作系统会对目录数有限制，进而限制数据库个数）（表的大小由操作系统规定的最大文件大小限制）
- Row Size Limits 65535 （ppt 第 51 页上的是什么意思？）（ppt p52 开始有例子，因为 text 和 blob 存的只是指针、只存 9-12 字节）（为什么 varchar 65535 直接设置是会报错的，因为还会有额外的几个 byte 去存 varchar 的它的长度信息、但是 65533 就是可以的了）（同样地、如果是可以为空的，那么也需要额外的 bit 去存储）（最后一个报错是因为 char，判断超过了页的一半，就失败了，这个貌似是规定、否则效率会低、同样的对于最后一个 row size 的例子也是一样、一行所有的不能超过半页？varchar 可以、char 不行、因为 varchar 是可以压缩的）（而且是定义的时候是可以的、但是如果真的插入 varchar 里面那么多内容、貌似也会报错）（char 是对齐的、varchar 是压缩的）
- innodb_page_size 设置页大小、默认是 16kb。8KB 就是 65536 bits

16. Optimizing for BLOB Types
    When storing a large blob containing textual data, consider compressing it first.

For a table with several columns, to reduce memory requirements for queries that do not use the BLOB column,
consider splitting the BLOB column into a separate table and referencing it with a join query when needed.

You could put the BLOB-specific table on a different storage device or even a separate database instance.
Rather than testing for equality against a very long text string, you can store a hash of the column value in a separate column, index that column, and test the hashed value in queries.

Since hash functions can produce duplicate results for different inputs, you still include a clause AND blob_column = long_string_value in the query to guard against false matches; the performance benefit comes from the smaller, easily scanned index for the hashed values.

12 MySQL Optimization

1. InnoDB table 表优化:
    - OPTIMIZE TABLE : 可以让表的碎片整理（把很多小块整理成一大块），然后重建索引，提高性能。（当数据量大小稳定，而且尺寸达到一定程度很大的时候）（本质是复制碎片然后调整索引，所以会浪费时间，不能频繁做）
    - 如果主键比较长的话、那么占的磁盘空间会比较多（因为默认会在主键上面做索引）

- 尽量用 varchar，不要用 char（因为 varchar 可以压缩）（特别是对于很多空的字段，如果空的话、就可以压缩）（除了定长和都有的字段、用 char）
- COMPRESSED 行格式可以对有很多重复内容的数据进行压缩（比如前缀相同的、可以压缩，数值和字符串都行，字符串有重复、或者数值很接近）（但是缺点是解析会慢，所以只有当这个表的空间很大的时候才用，小的时候不用）

2. 事务管理的优化：
   - 默认每个 sql 语句都是在一个事务里面的，可以通过 autocommit 来设置多久提交一次。也可以设置 autocommmit 为 0，然后手动提交。
   - 如果一个事务里面只有类似于 select 的操作、那么 innodb 会对这个只读事务进行优化。
   - 要避免很长的事务，因为会涉及到很多的 inserting、updating 很多行，而且会有加很多锁的问题。（这个问题还不能通过重启解决、因为重启的时候会有 rollback 的操作，又加锁了）
   - 可以通过调整 buffer pool 的大小来优化性能，因为 buffer pool 是 innodb 的内存缓存，可以存储数据和索引，减少磁盘 I/O 操作。（一般是脏页到了 buffer 的 10%的时候会写、因为如果超过 10%，恢复的时候开销会很大） innodb_buffer_pool_size option.
   - 可以通过设置 innodb_change_buffering 来设置 innodb 的写入策略，可以设置为 all、inserts、none。（all 是所有的都缓存，inserts 是只缓存插入的，none 是不缓存）（而且实际业务种一般不删除，是通过标记删除的方式）
   - 避免大规模的数据操作事务，会导致数据不一致的问题
   - 也要避免长时间运行的事务、因为会有锁的问题、而且如果 non-steal、会占用很多内存（产生冲突之后不好恢复）
   - If database write performance is an issue, conduct benchmarks with the innodb_flush_method parameter set to O_DSYNC.

3. 大量数据导入的时候的优化：
   - importing data into InnoDB, turn off autocommit mode，最后统一 commit
   - 同样地、也可以关掉 UNIQUE constraints 的检查，然后最后再检查一遍
   - 外键关联也是一样、可以关掉 foreign key checks，然后最后再检查一遍
   - with auto-increment columns, set innodb_autoinc_lock_mode to 2 (interleaved 穿插) instead of 1 (consecutive 连续).（自增的数据维护在一个系统表里）（因为大量数据导入的时候、肯定不是一个线程，设置为 interveaved 的时候、当有多个线程的时候、读走的是一个区域的数值，然后分别去插入）
   - 当多行插入的时候、用 multiple-row INSERT syntax 、避免额外的通信开销

4. Queries 的优化：
   - 聚簇索引：指的是 B 树上的叶子顺序和在磁盘上的顺序是一样的，所以可以减少磁盘的读取
   - 不要让主键太长，因为会占用很多的空间在索引里面
   - 所以主键应该是经常搜索的内容，而且会涉及到批量读取的内容
   - 要根据查询的需求做复合索引（根据支持的功能），不要在每一列上做单列索引
   - 索引列本身不要包含空的值（从 null 开始一堆的链表）

5. 磁盘 IO 的优化：
   - P10 有 3 个例子
   - fsync threshold 是什么?
   - 会有不同的存储数据、如果需要随机访问、那么放在 ssd 上面、如果是顺序访问、比如 logfile、那么放在 hdd 上面（根据磁盘的特性）

1. **识别磁盘 I/O 问题**：
    - 如果你已经按照最佳实践进行数据库设计和 SQL 操作调优，但数据库依然因磁盘 I/O 活动过重而变慢，你可能需要考虑进一步优化磁盘 I/O。
    - 使用 **Unix top** 工具或 **Windows Task Manager** 检查 CPU 使用率，如果在你的工作负载下 CPU 使用率低于 70%，说明你的数据库可能是受磁盘 I/O 限制的（即 **磁盘瓶颈**）。
2. **增加缓冲池大小**：
    - 当表数据被缓存到 **InnoDB 缓冲池** 时，查询可以反复访问这些数据，而无需进行磁盘 I/O 操作。通过设置 **innodb_buffer_pool_size** 选项，你可以指定缓冲池的大小。
    - 增加缓冲池大小能减少磁盘 I/O，提高数据库性能。
3. **调整刷新方法**：
    - 如果数据库的写入性能是一个问题，你可以通过将 **innodb_flush_method** 参数设置为 **O_DSYNC** 来进行性能基准测试，查看此设置是否能提升性能。
4. **配置 fsync 阈值**：
    - 你可以使用 **innodb_fsync_threshold** 变量来定义一个阈值（以字节为单位）。
    - 指定一个阈值，强制进行较小、周期性的刷新，对于多个 MySQL 实例共享同一存储设备的情况可能特别有用。这样可以减少频繁的磁盘 I/O 操作，提高性能。
5. **增加 I/O 容量以避免积压**：
    - 如果你注意到由于 **InnoDB 检查点操作**，吞吐量周期性下降，建议增大 **innodb_io_capacity** 配置选项的值。
    - 设置更高的 **innodb_io_capacity** 值将使得 **InnoDB** 执行更频繁的刷新操作，从而避免因工作积压而导致吞吐量下降。
    - P11 Increase I/O capacity to avoid backlogs 这个是什么，checkpoint 需要提高磁盘 io 空间?
6. DDL 操作，比如创建表、修改表结构、删除表等操作：
    - Use TRUNCATE TABLE to empty a table, not DELETE FROM tbl_name.
    - 不要修改主键、因为会涉及到很多的索引的重建
7. 内存表的优化：
    - 应该放常用的而且不重要的数据（而且只读、尽量少写）
    - 一般使用 hash index、也可以指定用 b 树，b 树对范围查找比较好，哈希对等值查找比较好
8. buffer pool 的优化：
    - 会有 chunk size 和 pool size 的区别
    - chunk size 默认是 128MB，代表 buffer pool 里面的一个块
    - 这个 buffer pool size 必须要是 chunk size 的整数倍，因为 chunk 是最小的单位
    - 可以设置多个实例、为了多个线程并行读取一个 table、减少竞争 For systems with buffer pools in the multi-gigabyte range, dividing the buffer pool into separate instances can improve concurrency, by reducing contention as different threads read and write to cached pages. （对于大的 buffer pool，可以分成多个实例，通过减少每个线程读写缓存页的竞争来提高并发性）
    - Newly read blocks are inserted into the middle of the LRU list.（新读取的块会插入到 LRU 列表的中间，一般是 3/8 的位置， 3/8 from the tail of the LRU list）
    - LRU list into two segments 也会有类似 generation 的旧代和新代的概念
    - Prefetching (Read-Ahead)来减少磁盘 io 的次数、提高性能

- 也可以设置同时最多有几个线程去把 dirty page 写到磁盘上 innodb_page_cleaners
- the low water mark value 10% 也可以调整
- 缓存的内容也可以做恢复、下次再启动的时候、可以减少 warm-up 的时间（通过记录 buffer pool 里面最常用的页）

9. **多缓冲池实例的优势**：
   - 对于 **缓冲池内存大小达到多吉字节（multi-gigabyte）** 的系统，将缓冲池分成多个实例可以提高 **并发性**，即允许更多的线程同时访问缓冲池中的数据。这样做的目的是减少线程之间的 **竞争**（contention），因为多个线程可以同时访问不同的缓冲池实例，避免了在同一个缓冲池内存区域的竞争。
10. **配置多缓冲池实例**：
    - 配置多个缓冲池实例可以通过 `innodb_buffer_pool_instances` 配置选项来完成。
    - 你还可以调整 `innodb_buffer_pool_size` 来设置总的缓冲池大小。两者的配置是相辅相成的。
11. **大缓冲池的好处**：
    - 当 **InnoDB** 的缓冲池很大时，系统可以从 **内存中直接** 提供数据，减少磁盘访问。很多数据请求可以通过从内存中检索数据来得到满足，而不需要频繁地访问磁盘，从而显著提升性能。
    - 对于内存较大的系统，分割缓冲池为多个实例有助于提升并发性能。
    - 通过合理配置 `innodb_buffer_pool_instances` 和 `innodb_buffer_pool_size`，可以优化 **InnoDB** 的内存利用率和减少线程之间的竞争，进而提高数据库的处理效率。
    - 更大的缓冲池意味着更多的数据可以在内存中直接访问，减少磁盘 I/O，从而提高数据库的整体性能。
12. **启用多个缓冲池实例**：
    - 要启用多个 **InnoDB** 缓冲池实例，需要将配置选项 `innodb_buffer_pool_instances` 设置为大于 1 的值（默认为 1），最大值为 64。
    - 缓冲池实例用于管理数据库缓存，从而提高多核处理器系统的并发性能。
13. **条件要求**：
    - 该配置选项只有在你将 `innodb_buffer_pool_size` 设置为 **1GB 或更大** 时才有效。
    - `innodb_buffer_pool_size` 是指为 InnoDB 缓冲池分配的总内存大小。
14. **内存划分**：
    - 当你设置了 `innodb_buffer_pool_instances` 和 `innodb_buffer_pool_size` 后，指定的总缓冲池内存大小将会被 **平均分配** 给所有的缓冲池实例。
    - 为了实现最佳效率，建议选择 `innodb_buffer_pool_instances` 和 `innodb_buffer_pool_size` 的组合，使得每个缓冲池实例至少有 **1GB** 的内存。
15. **缓冲池扫描抗性**：
    InnoDB 不使用严格的 LRU（最久未使用）算法，而是通过将新读入的块插入 LRU 列表的中间位置来优化缓冲池管理。这样可以确保频繁访问的"热"页面不会被替换掉，而不再访问的页面更容易被清理，从而提高缓冲池的效率。
16. **InnoDB 缓冲池预读取（Read-Ahead）**：
    预读取是 InnoDB 通过异步 I/O 请求提前加载多个页面到缓冲池中，目的是提前准备即将需要的页面。InnoDB 使用两种预读取算法：
    - **线性预读取（Linear Read-Ahead）**：基于顺序访问的页面来预测即将需要的页面。

- **随机预读取（Random Read-Ahead）**：基于缓冲池中已加载的页面来预测即将需要的页面，不管这些页面的访问顺序如何。
  `SHOW ENGINE INNODB STATUS` 命令可以显示预读取算法的统计信息，帮助调优 `innodb_random_read_ahead` 配置。

17. **缓冲池刷新（Buffer Pool Flushing）**：
    在 MySQL 8.0 中，InnoDB 通过后台的页面清理线程（Page Cleaner Threads）来执行缓冲池的刷新。页面清理线程将修改过的页面写回磁盘。`innodb_page_cleaners` 变量控制线程数，`innodb_max_dirty_pages_pct_lwm` 变量定义了脏页面达到的低水位阈值，当脏页面比例超过该阈值时，会触发刷新操作。
18. **缓冲池状态的保存和恢复**：
    为了减少服务器重启后的热启动时间，InnoDB 会在服务器关闭时保存一部分最近使用的缓冲池页面，并在启动时恢复这些页面。通过 `innodb_buffer_pool_dump_pct` 配置项控制保存的页面百分比。
19. **预处理语句和存储程序的缓存**：
    InnoDB 还会缓存预处理语句和存储程序（例如存储过程、触发器等）。
    `max_prepared_stmt_count` 控制服务器缓存的最大预处理语句数量，而 `stored_program_cache` 则控制每个会话缓存的存储程序的数量。每个会话的缓存是独立的，其他会话无法访问其缓存的语句。

13 MySQL Backup & Recovery

1. 备份：
   - 一个是为了数据库崩溃时的恢复
   - 还有一个是防止 mysql 升级后的数据丢失
2. 备份的种类：
   - 逻辑备份：通过 sql 语句去备份（版本迁移，比较慢、适合数据量比较小的情况，可以移植到不同的机器的数据库上）
   - 物理备份：直接备份文件（适合比较大、比较重要的数据库，恢复也会比较快，不能适用版本迁移）
   - 在线备份：不停机的备份（会有不一致的问题，用户体验较好，但是要加锁）（hot）
   - 离线备份：停机的备份（会有停机的问题）（cold）
   - warm backup：只允许读，不允许写（warm）
   - 远程备份和本地备份：只是远程操控和本地操控备份操作的情况、备份的位置还是在数据库的机器上
   - 快照备份（增量式备份）：用逻辑备份备份变化的部分。（mysql 不支持快照备份）
   - incremental backup：只备份变化的部分（mysql 使用 binlog 实现）
   - Backup Scheduling, Compression, and Encryption
   - Backup scheduling is valuable for automating backup procedures.
   - Compression of backup output reduces space requirements, and
   - encryption of the output provides better security against unauthorized access of backed-up data.
3. mysql 备份方法：
   - 企业版的 mysql 有自己的备份工具
   - Innodb 用的是 hot backup，其他引擎的表用的是 warm backup

- For InnoDB tables, it is possible to perform an online backup that takes no locks on tables using the --single-transaction option to mysqldump.
- 也可以保存成不同格式的文件
- 也可以关掉 binlog、但是就没法实现增量式备份了（binlog 也是 append）
- 也可以做主从备份（然后做负载均衡、比如读操作可以到所有节点、然后写操作只到主节点）

4. **使用 MySQL Enterprise Backup 进行热备份**
   - MySQL 企业版用户可以使用 MySQL Enterprise Backup 产品对整个实例或选定的数据库、表进行物理备份。该产品支持增量备份和压缩备份功能。物理备份比 mysqldump 等逻辑备份恢复速度更快。
   - 对于 InnoDB 表，使用热备份机制进行备份。（理想情况下，InnoDB 表应占据大部分数据。）
   - 对于其他存储引擎的表，使用冷备份机制进行备份。
5. **使用 mysqldump 进行备份**
   - mysqldump 工具可以进行备份，支持备份各种表类型。对于 InnoDB 表，可以使用 `--single-transaction` 选项执行在线备份，这种方式不需要对表进行锁定。
6. **通过复制表文件进行备份**
   - 可以通过复制 MyISAM 表的文件（如 *.MYD*、.MYI 文件和相关的 *.sdi 文件）来进行备份。为了获得一致性的备份，可以在复制表文件之前锁定并刷新相关表：

     ```
     FLUSH TABLES tbl_list WITH READ LOCK;
     ```

   - 只需要读取锁定，这样其他客户端仍然可以查询表，但在开始复制表文件之前，所有活动的索引页需要写入磁盘。
   - 也可以通过简单地复制表文件进行二进制备份，前提是服务器没有进行任何更新。但需要注意，表文件复制方法不适用于 InnoDB 表，即使服务器没有更新数据，InnoDB 可能仍然会有修改的数据缓存尚未写入磁盘。
7. **创建分隔文本文件备份**
   - 通过 `SELECT * INTO OUTFILE 'file_name' FROM tbl_name` 可以创建包含表数据的文本文件。该文件会在 MySQL 服务器主机上创建，而不是客户端主机上。需要注意，输出文件不能已存在，因为允许覆盖文件存在安全风险。
   - 这种方法适用于任何类型的数据文件，但仅保存表的数据，而不保存表的结构。
   - 另一种方法是使用 mysqldump 的 `--tab` 选项，创建包含表结构和数据的分隔文本文件。恢复时可以使用 `LOAD DATA` 或 `mysqlimport` 命令。
8. **通过启用二进制日志进行增量备份**
   - MySQL 支持使用二进制日志进行增量备份。二进制日志文件记录了自上次备份以来数据库的所有更改信息。
   - 在进行增量备份时，应使用 `FLUSH LOGS` 来轮换二进制日志。下一次进行完全备份时，您还应使用 `FLUSH LOGS` 或 `mysqldump --flush-logs` 来轮换二进制日志。
9. **使用复制服务器进行备份**
   - 如果在备份过程中服务器性能较差，您可以设置复制，并在复制服务器上进行备份，而不是在主服务器上备份。
   - 在备份复制服务器时，除了备份复制的数据库外，还应备份其连接元数据存储库和应用元数据存储库，这些信息在恢复复制服务器数据后是恢复复制所必需的。

- 如果复制服务器正在复制 `LOAD DATA` 语句，还应备份复制服务器用于该操作的 SQL_LOAD-* 文件，以便在恢复后继续复制任何中断的 `LOAD DATA` 操作。

10. **恢复损坏的 MyISAM 表**
    - 如果您的 MyISAM 表损坏，首先可以尝试使用 `REPAIR TABLE` 或 `myisamchk -r` 来恢复表数据。这种方法在 99.9% 的情况下都能有效恢复数据。
11. **使用文件系统快照进行备份**
    - 如果您使用的是 Veritas 文件系统，可以按照以下步骤创建备份：
      - 从客户端程序中执行 `FLUSH TABLES WITH READ LOCK`，该命令会锁定表以防止数据更新。
      - 从另一个终端执行 `mount vxfs snapshot`，挂载文件系统快照。
      - 在客户端中执行 `UNLOCK TABLES`，解除表的锁定。
      - 从快照中复制文件。
      - 卸载快照。
    - 其他文件系统（如 LVM 或 ZFS）也可能提供类似的快照功能，可以使用相似的步骤进行备份。
12. 数据库恢复方法：
    - For cases of operating system crashes or power failures, we can assume that MySQL's disk data is available after a restart.（因为有 log，可以直接恢复）
    - For the cases of file system crashes or hardware problems, we can assume that the MySQL disk data is not available after a restart.（所以会需要备份）
    - This backup operation acquires a global read lock on all tables at the beginning of the dump.（备份策略：备份的时候会加锁，这里 ppt 上有一个例子）
    - 对于没有被截断的 binlog 文件，要用 mysqlbinlog gbichot2-bin.000009 ... | mysql 里的 ...，说明还没有被截断
    - log 文件和数据文件可以分别存在不同的硬盘上、这样就避免了一个硬盘坏了之后、数据和日志都丢失的问题
    - tab-delimited text 用 tab 分隔的文本文件
13. 可以用 merkle 树来存储哈希码，然后通过对比这棵树来判断保存和恢复的数据是否一致（dump 和 recovery 的时候同时建树）
14. 下面的这种方法进行多线程地恢复是不对的，因为可能第一个 binlog 会创建一个临时表、然后第二个 binlog 会用到这个临时表，但是这个临时表在 binlog001 运行完之后、会被删除，就会在第二个 binlog 里报“unknown table”

```
shell> mysqlbinlog binlog.000001 | mysql -u root -p # DANGER!!
shell> mysqlbinlog binlog.000002 | mysql -u root -p # DANGER!!
```

应该用

```
shell> mysqlbinlog binlog.000001 binlog.000002 | mysql -u root -p
```

7. 也可以指定时间点去做 binlog 的恢复（可以跳过 file 中的一些操作）
8. 做完 full backup 时候、会把先前的所有的 binlog 都删除，然后重新开始
9. binlog 为了系统崩溃的时候恢复。

10. **备份策略总结**
    在操作系统崩溃或电源故障的情况下，InnoDB 本身会负责数据恢复。但为了确保可以放心，建议遵循以下指南：

11. **启用二进制日志**
    始终启用二进制日志（MySQL 8.0 的默认设置）。二进制日志有助于记录所有的数据库更改，从而支持恢复和数据复制等功能。

12. **磁盘负载平衡**
    如果您有安全的存储介质，这种技术对于磁盘负载平衡也有帮助，从而提升性能。

13. **定期进行全备份**
    使用 `mysqldump` 命令进行定期的全量备份，这种备份方式是在线的、非阻塞的。

14. **定期进行增量备份**
    通过 `FLUSH LOGS` 或 `mysqladmin flush-logs` 命令定期进行增量备份。

15. **考虑使用 MySQL Shell 转储工具**
    使用 MySQL Shell 的转储工具提供并行转储、多线程、文件压缩和进度信息显示等功能，还支持 Oracle 云基础设施对象存储流媒体以及 MySQL 数据库服务的兼容性检查和修改。
    转储文件可以轻松导入到 MySQL Server 实例或 MySQL 数据库服务 DB 系统中。

16. **转储文件的多种用途**
    转储文件可以用于：
    - 数据丢失时的数据恢复备份。
    - 设置复制的源数据。
    - 实验用途：
      - 创建数据库的副本，用于不改变原始数据的测试。
      - 测试潜在的升级兼容性问题。

17. **mysqldump 的输出格式**
    `mysqldump` 产生两种类型的输出，取决于是否使用了 `--tab` 选项：
    - **没有 `--tab` 选项时**，`mysqldump` 将 SQL 语句写入标准输出。这些输出包括创建数据库、表、存储程序等的 `CREATE` 语句，以及将数据加载到表中的 `INSERT` 语句。输出可以保存在文件中，稍后使用 `mysql` 命令重新加载，以重新创建转储的对象。
    - **使用 `--tab` 选项时**，`mysqldump` 为每个转储的表产生两个输出文件。服务器会将一个文件写为制表符分隔的文本，每行一个表行，文件名为 `tbl_name.txt`，另一个文件是包含 `CREATE TABLE` 语句的 SQL 文件，文件名为 `tbl_name.sql`。

18. **如何复制数据库**
    - 如何从一个服务器复制数据库到另一个服务器。
    - 如何转储存储程序（存储过程和函数、触发器和事件）。
    - 如何分别转储数据库的定义和数据。

19. **备份和恢复单个数据库**：
    - 使用 `mysqldump` 导出数据库内容：`shell> mysqldump db1 > dump.sql`
    - 创建一个新的数据库：`shell> mysqladmin create db2`
    - 导入备份文件到新数据库：`shell> mysql db2 < dump.sql`
    - 不要在 `mysqldump` 命令中使用 `--databases`，因为这会导致在备份文件中包含 `USE db1`，会覆盖在 `mysql` 命令行中指定的数据库。

20. **从一个服务器到另一个服务器复制数据库**：

- 在服务器 1 上：
  - 使用 `--databases` 选项导出数据库：`shell> mysqldump --databases db1 > dump.sql`
  - 将 `dump.sql` 文件从服务器 1 复制到服务器 2。
- 在服务器 2 上：
  - 使用 `mysql < dump.sql` 导入数据。

另一种方式是：

- 在服务器 1 上：
  - 不使用 `--databases` 选项：`shell> mysqldump db1 > dump.sql`
- 在服务器 2 上：
  - 创建数据库：`shell> mysqladmin create db1`
  - 导入数据：`shell> mysql db1 < dump.sql`

21. **导出存储程序（存储过程、函数、触发器、事件）**：
    - 使用以下选项来控制 `mysqldump` 如何处理存储程序：
      - `--events`：导出事件调度器事件。
      - `--routines`：导出存储过程和函数。
      - `--triggers`：导出表的触发器。
    - 默认情况下，`--triggers` 选项是启用的，其他选项需要显式指定来导出相应的对象。
    - 如果不希望导出某些内容，可以使用其跳过形式：`--skip-events`、`--skip-routines` 或 `--skip-triggers`。
22. **分别导出表结构和数据**：
    - 使用 `--no-data` 选项，只导出表的结构（不包含数据）：`shell> mysqldump --no-data test > dump-defs.sql`
    - 使用 `--no-create-info` 选项，只导出表的数据（不包含结构）：`shell> mysqldump --no-create-info test > dump-data.sql`
    - 如果需要导出表结构并包含存储程序和事件定义，可以使用以下命令：
      - `shell> mysqldump --no-data --routines --events test > dump-defs.sql`
23. **使用 `mysqldump` 测试 MySQL 升级兼容性**：
    - 在进行 MySQL 升级时，建议先在新版本的服务器上安装，并从当前生产环境中导出数据库定义，导入新服务器以检查兼容性（也适用于测试降级）。
      - 在生产服务器上：`shell> mysqldump --all-databases --no-data --routines --events > dump-defs.sql`
      - 在升级后的服务器上：`shell> mysql < dump-defs.sql`
    - 因为备份文件不包含表数据，可以很快处理，帮助快速检测潜在的不兼容问题。
24. **使用 `mysqldump` 测试升级数据加载**：
    - 确保数据库定义没有问题后，接下来可以导出数据并尝试将其加载到升级后的服务器：
      - 在生产服务器上：`shell> mysqldump --all-databases --no-create-info > dump-data.sql`
      - 在升级后的服务器上：`shell> mysql < dump-data.sql`

- 导入数据后，可以检查表内容并运行一些测试查询。

25. **Point-in-time recovery（时间点恢复）**：
    时间点恢复是指恢复数据到某个指定的时间点，通常在恢复了一个完整的备份后进行，该备份将服务器恢复到备份时的状态。然后，时间点恢复通过增量的方式将服务器恢复到从完整备份时间到更近时间的状态。

26. **使用二进制日志进行时间点恢复**：
    要从二进制日志恢复数据，必须知道当前二进制日志文件的名称和位置。可以使用以下命令查看：

    ```
    mysql> SHOW BINARY LOGS;
    ```

    要确定当前二进制日志文件的名称，可以执行：

    ```
    mysql> SHOW MASTER STATUS;
    ```

27. **应用二进制日志中的事件**：
    要应用二进制日志中的事件，可以使用 `mysqlbinlog` 输出通过 `mysql` 客户端处理：

    ```
    shell> mysqlbinlog binlog_files | mysql -u root -p
    ```

28. **处理加密的二进制日志**：
    从 MySQL 8.0.14 版本开始，二进制日志可以被加密。在这种情况下，`mysqlbinlog` 不能像上述示例那样直接读取加密的日志，但可以使用 `--read-from-remote-server` 选项通过服务器读取。例如：

    ```
    shell> mysqlbinlog --read-from-remote-server --host=host_name --port=3306 --
    user=root --password --ssl-mode=required binlog_files | mysql -u root -p
    ```

29. **查看二进制日志中的事件**：
    要查看日志中的事件，可以将 `mysqlbinlog` 输出传递到分页程序中：

    ```
    shell> mysqlbinlog binlog_files | more
    ```

    或者将输出保存到一个文件中，并用文本编辑器查看：

    ```
    shell> mysqlbinlog binlog_files > tmpfile
    shell> ... edit tmpfile ...
    ```

30. **编辑日志文件并执行**：
    将输出保存到文件中是处理日志的预备步骤，特别是当需要删除某些事件（如意外的 `DROP TABLE`）时。你可以从文件中删除不想执行的语句，编辑完成后，通过以下命令应用文件内容：

    ```
    shell> mysql -u root -p < tmpfile
    ```

31. **多个二进制日志的安全恢复**：
    如果有多个二进制日志要应用，安全的方法是使用单个连接来处理它们。以下是一个可能不安全的示例：

```
shell> mysqlbinlog binlog.000001 | mysql -u root -p # 危险!!
shell> mysqlbinlog binlog.000002 | mysql -u root -p # 危险!!
```

这种方式存在问题，如果第一个日志文件中包含 `CREATE TEMPORARY TABLE` 语句，而第二个日志文件包含使用该临时表的语句，当第一个 `mysql` 进程终止时，服务器会丢弃临时表。第二个 `mysql` 进程尝试使用该表时，服务器会报告“未知的表”。

32. **恢复最后的完整备份**：
    假设你在某个特定时间点（例如 2020 年 3 月 11 日 20:06:00）执行了删除表的 SQL 语句，你需要进行时间点恢复来恢复到该时间点之前的数据库状态。
    首先，恢复在该时间点之前创建的最后一个完整备份（假设时间点是 `tp`，即 2020 年 3 月 11 日 20:06:00）。恢复完成后，记下恢复到的二进制日志位置，以便稍后使用，并重启服务器。
33. **查找精确的二进制日志事件位置**：
    接下来，需要找到对应时间点（`tp`）的二进制日志事件位置。在这个例子中，假设你知道大致的时间（`tp`），可以通过使用 `mysqlbinlog` 工具来检查该时间附近的日志内容，找到精确的事件位置。
    使用 `--start-datetime` 和 `--stop-datetime` 选项来指定一个短时间段，围绕 `tp` 时间点进行查询。然后在输出的日志中查找相关的事件（如删除表的 SQL 语句）。
34. **应用二进制日志中的事件到服务器**：
    首先，将步骤 1 中找到的二进制日志位置（假设是 155）作为起始位置，步骤 2 中找到的、在你感兴趣的时间点（例如 `tp`）之前的位置（假设是 232）作为结束位置，将二进制日志事件应用到服务器。
    使用 `mysqlbinlog` 工具，指定 `--start-position` 和 `--stop-position`，例如：

    ```
    mysqlbinlog --start-position=155 --stop-position=232
    /var/lib/mysql/bin.123456 | mysql -u root -p
    ```

    执行该命令后，你的数据库将恢复到感兴趣的时间点 `tp`，即在删除 `pets.cats` 表之前的状态。
35. **恢复时间点之后的所有语句**：
    如果你还希望重新执行恢复后的时间点 `tp` 之后的所有 SQL 语句，可以再次使用 `mysqlbinlog` 工具，将时间点之后的所有事件应用到服务器。
    在步骤 2 中，我们注意到在需要跳过的语句后，日志的位置为 355，可以使用这个位置作为 `--start-position` 参数，包含所有此位置之后的语句。例如：

    ```
    mysqlbinlog --start-position=355 /var/lib/mysql/bin.123456 | mysql -u root -p
    ```

    这样可以将时间点 `tp` 之后的所有事件重新应用到服务器。

14 MySQL Partitioning

1. 所有数据库都有类似的问题、所以解决方法也有通用性
2. 一个表在一个机器上放不下、所以分区
3. 是不是放不下了才去分？也不是、分区做搜索、会把搜索的范围限制到一个区域里，即缩小了、所以搜索会更快
4. 分区：单个的一张表的不同部分可以分布在不同的机器上存储
5. 都是水平分区支持，mysql 垂直分区不支持（不如变成两张表、然后做外键关联，而且也会需要水平分区）
6. 有了分区之后、也可以逐个增加分区

7. 也可以指定分区进行搜索以及其他操作
8. 有四种分区的方式：
    - RANGE 分区：根据某个列的范围进行分区（连续的值）
    - LIST 分区：根据某个列的值进行分区（离散的值，eg 名字）
    - HASH 分区：根据某个列的 hash 值进行分区
    - KEY 分区：根据某个列的 hash 值进行分区（except that only one or more columns to be evaluated are supplied）
9. range partition：
    - （按照小于来表达，因为如果是限制左右范围的话、那么删去一个分区、就会有很多的分区不连续，而且会有重叠）
    - 也可以不用 less than maxvalue、相当于做了一个筛选
    - 这个 less than value 要严格升序（比如在 UNIX_TIMESTAMP 里面统一的时候），不然就会报错
    - 多列 range partition 也是可以的，是按照列的顺序来依次判断的
    - 如果有 6-8-aaa 会存在哪里?
10. List partition：
    - 是根据 id 的值来分区的，如果 id 的值不在、插入的时候会报错
    - 也可以在单次插入多个值的时候、用 ignore 来忽略掉不在的值，避免报错
11. 列的字段
    - range 不支持：浮点数、TEXT and BLOB columns
    - 使用 RANGE COLUMN 指定多行
12. hash partition：
    - 可以指定 hash 的方法，比如指定日期里的年份做 hash
    - 也有一个 LINEAR HASH 的方法，即不是用取余，而是用一个 2 的幂次方来做 hash，为了就是当有分区变化的时候、变化和调整小一些（一个公式、ppt 有例子、会分布得比较均匀）
13. key partition：
    - 这个 hash 函数没法指定、是 mysql 自己决定的
14. 可以做 subpartition，比如在一个分区里按照日期再分一次
    - 每一个分区里面的子分区也可以不一样，也可以给对应的子分区的名字
15. 设置分区也会让删除更快，可以直接把一个分区 drop 掉了
16. 解决 null 的值
    - 空值 null 会存到 range 的最小的分区里面
    - 可以把 null 加到某个分区 list 里面
    - hash 的时候会把这个值看成 0
17. 分区管理
    - 可以改变分区方式
    - 可以指定操作的分区
    - 分区的数字需要有意义、比如对应有意义的年份、对应经常的查询操作等、不然意义不大

    - 追加 range 分区只能加在最后的分区那边、放在前面会报错、因为会涉及到很多数据的转移（但是也可以指定重新组织 reorganize table）
    - 新加 list 分区不能有重复的值
    - hash 和 key 可以合并分区、但是不能删除分区（但是合并的数值不能超过原来的数值）
    - 这个分区和表是可以双向进行替换的（结构要一样、没有外键关联，分区条件要满足）（也可以 without validation，结果是啥应该是就是插入了）
18. 也有 maintenance partitions，用来解决分区里的数值的问题
19. 分区的逻辑要和查询的业务相关、比如如果要查整 5 整 10 的 sql、那么就要按照这个来分区（因为 query plan 的优化）

15 NoSQL & MongoDB

1. 为什么要非关系型数据库、因为数据量太大了
2. mysql 处理结构化数据，而 nosql 处理非结构化数据
3. 有结构化数据、半结构化数据和非结构化数据
4. ppt 上有传统关系型数据库和 map-reduce 的区别
    - 也可以会有 dynamic schema，动态调整结构和字段
    - map reduce 对分布式做了优化、锁竞争被减弱了、所以可以达到线性的扩展
5. mongodb 是文档型数据库
    - 支持全索引
    - 通过备份提供高可用性
    - 有 auto-sharding，可以自动做的（类似于分区）
    - 就地修改会快
6. 文档 document 的定义
    - A document is the basic unit of data for MongoDB
    - A collection can be thought of as the schema-free equivalent of a table, the documents in the same collection could have different shapes or types. （就相当于在一个文档中的数据的结构可以不一样，很多文档构成了一个 collection，是 schema-free 的）
    - 默认在每个文档都有的一个特殊的键_id 上建索引（id 是唯一表示的，但是不是主键的概念，因为在数据库表里有很多候选键）
    - 很多 collection 构成了一个 database
7. 文档的特点
    - 他里面的键值对是有序的，然后不同的序的数据可以存在一个 collection 里面
    - 里面的数据是类型敏感和大小写敏感的
    - 里面的 key 不能重复
    - schema-free
8. 与 mysql 的优势、比如对于下面这个数据、在 mysql 里面就要定义两张表，然后用外键关联，所以在一个 collection 里面可以表示两个外键关联的表的关系、所以形式比较简单

```
{
  "name": "John Doe",
  "address": {
    "street": "123 Park Street",
    "city": "Anytown",
    "state": "NY"
  }
}
```

9. collection
    - 所有的数据不能存在一个 collection 里面，首先会更快，而且可以利用数据局部性、只拿出自己想要的数据、避免拿出数据的时候拿出多余的不想要的数据，然后和索引也有关系
    - subcollection 相当于对字段做了分割，但是两个 subcollection 直接没有直接的关系（比如外键关联）
10. database
    - admin 是 root database、用来鉴权
    - local: This database will never be replicated and can be used to store any collections that should be local to a single server.
    - config 的服务器会做这个操作、当某个请求来了、根据_id 的范围来去哪个 sharding 的服务器上找对应的数据
11. mongoDB 里面有一个聚集操作、可以做流水线式的操作、即有不同的操作阶段
    - $unwind 做摊平操作
12. 注解与查询字段绑定：@Param("name") 注解用于明确告诉 Spring Data，将方法参数 name 映射到查询中的对应字段值（在 MongoDB 查询中，如果你通过 findByLastName 查询，则会根据 lastName 字段进行查询）。

```
public interface PersonRepository extends MongoRepository<Person, String> {

    List<Person> findByLastName(@Param("name") String name);
    List<Person> findByFirstName(@Param("name") String name);

}
```

13. 下面的@Transient 注解用于告诉 Spring JPA，不要将这个字段持久化到数据库中

```
    private PersonIcon icon;
    @Transient
    public PersonIcon getPersonIcon(){
        return icon;
    }

    public void setIcon(PersonIcon icon) {
        this.icon = icon;
    }
```

14. 这里把图片存到 mongoDB 里面只是为了演示、实际上用 blob 也可以

15. 有一个 normal index，还有一个 geospatial index，就可以根据坐标的范围来求最近点之类的内容了，但是这里的$maxdistance 代表的是一个单位，就是需要提前设置这个搜索的单位精度，然后根据这个单位来搜索
16. **创建地理空间索引**：
    可以使用 `createIndex` 函数创建地理空间索引，但要传递 `"2d"` 作为值，而不是 1 或 -1。
    这意味着索引类型为二维地理空间索引。
17. **改变默认精度**：
    要改变默认的精度，可以在创建二维索引时指定 `bits` 值。
    你可以指定一个介于 1 和 32 之间的 `bits` 值（包含 1 和 32）。
    `bits` 值越大，索引的精度越高，查询的效率也可能有所变化。
18. 这里有一个删除索引的事情？回放？
19. 这个貌似有一个就是不加自定义语句的话、就是按照经纬度去计算（貌似只要是一对数值就可以自动进行计算了）、如果加了、就按照二维坐标系里的距离去计算（相当于是二维索引的范围搜索）
20. 注解里的 fields 可以用来做结果的返回的过滤
21. $nearSphere 也可以不止是一个球、比如说正方形之类的
22. 进一步地、这个非关系型数据库就可以解决下面这种很多种枚举的情况、不可能枚举完、而且也不能在关系型数据库里用很多张表去维护（相当于 schema 不一样的时候、做某些操作、比如说搜索的时候、即使有不存在的字段、也不会报错）

```
INSERT INTO `mobile_params` (`id`, `mobile_id`, `name`, `value`) VALUES
   (1, 1, 'Standby time', '200'),
   (2, 1, 'Screen', 'OLED'),
   (3, 1, 'Quality', 'SSS'),
   (4, 2, 'Standby time', '300'),
   (5, 2, 'Screen', 'Curve'),
   (6, 2, 'Price', 'Attractive');
```

20. 貌似这个 repository interface 只能返回对应管理的类，而不是自定义的什么比如字符串的返回值
21. 在这段代码中，我们首先通过 ID 查找用户，如果找到则更新用户的 email 字段，并保存回数据库。这种方式会替换整个文档，但如果仅修改了一个字段，MongoDB 会进行内部优化以只更新变化的部分。
22. 我服了、搞了半天原来是数据源里的数据库的名称没配置对。。。应该是 mongodb://localhost:27017/ebookstore
23. 通过 spring 操作完之后、里面多了一个 class 是表示这个对象的类类型信息，它是 Spring 框架（特别是 Spring Data 或类似的框架）序列化对象时添加的额外字段。

```
{
  "id": 1,
  "description": "This book provides a comprehensive introduction to the field of
computer science. It covers topics such as algorithms, data structures,
programming languages, and computer architecture. The book is suitable for
beginners and does not require any prior knowledge of programming. 123123",
  "class": "com.example.backend.entity.BookInfo"
}
```

24. 这里下完订单保存的时候、会把原来的 description 给覆盖掉、需要修改

25. sharding，把一个大的 collection 切成很多小的 chunks，然后可以分布到不同的服务器上（autosharding 是自动的）
    - 然后会按照索引在不同的 chunk 里找（索引也是 B+树）
    - 会使用一个 router 来找到对应的 chunk（router 在 mongodb 里的）
    - 客户端里的 driver 会先和 router mongos 联系、根据 router 里的 config 找到对应的集群里的 mongod
26. 什么时候需要做 shard
    - 磁盘空间不够了
    - 要增加写数据的速度（比单一个 mongod 能承受的）（在这个 nosql 的情况下、写是一次性写入的）
    - 也有可能是内存不够用，需要怎加机器作为集群
27. 增加机器的时候、会有一个 balancer 来做数据的迁移（到不同的服务器上）
    - 因为一个 shard 里面会有不同的 chunk
    - 如何实现的：就是 router 会去 primary db 里面找一个 config、这个 config 保存了其中对应的每个 chunk 保存的_id key 的 range，当每个 chunk 超过一定大小、比如 128MB 的时候、就会切开这个 chunk
    - 当每个 shard 直接的 chunk 树相差小于等于 2 的时候可以接受、如果超过了、就会自动迁移（是不是 2 是可以配置的，也可以把这个自动分配关掉）
    - 均匀分布的意义在于：数据量访问的频繁次数，前提是数据是对等的（每个数据的访问量差不多）（反例就是新闻，光靠数据量均匀、不能提高性能）（所以在这种情况下可以根据数据访问的情况去做均衡）
    - client 是不知道这个的，无感知的
28. **分片是 MongoDB 的扩展方式**：
    分片是 MongoDB 用来扩展处理能力的方式，它允许你通过添加更多机器来应对不断增加的负载和数据量，而不影响应用程序的运行。
29. **手动分片的挑战**：
    手动分片虽然能起作用，但在添加或移除节点，或者数据分布和负载模式发生变化时，会变得难以维护。
30. **自动分片的优势**：
    MongoDB 支持自动分片（autosharding），这减少了手动分片所带来的一些管理难题。
31. **MongoDB 分片的基本概念**：
    MongoDB 的分片概念是将集合（collections）分成小块（chunks）。这些小块便于分布在多个分片上。
32. **应用程序和路由器的工作方式**：
    我们不需要知道每个分片上存储了哪些数据，因此在应用程序前面运行一个路由器。路由器知道数据的位置，应用程序可以正常连接到它并发出请求。
33. **路由器如何工作**：
    应用程序会连接到一个普通的 `mongod` 实例，路由器知道哪个分片存储了哪些数据，它会将请求转发到适当的分片。
34. **分片的适用情况**：
    一般来说，应该从非分片配置开始，当有需要时再转换为分片配置。适合分片的情况包括：
    - 当前机器的磁盘空间已用尽。
    - 你希望比单个 `mongod` 实例处理数据的速度更快。

    - 你希望将更多的数据保存在内存中以提高性能。
35. **分片键的选择**：
    在设置分片时，你需要选择一个集合中的键，并使用该键的值来拆分数据。这个键被称为**分片键（shard key）**。
36. **分片键示例**：
    例如，如果选择 "name" 作为分片键，那么一个分片可能存储名字以 A–F 开头的文档，另一个分片存储名字以 G–P 开头的文档，最后一个分片存储名字以 Q–Z 开头的文档。
37. **动态平衡数据**：
    当添加或移除分片时，MongoDB 会重新平衡数据，使每个分片的流量和数据量合理分配。
38. **分片和数据块**：
    假设添加一个新的分片。一旦该分片启动并运行，MongoDB 会将集合拆分成两个数据块（chunks）。
39. **数据块的定义**：
    一个数据块包含了某个范围的分片键值对应的所有文档。例如，如果使用 "timestamp" 作为分片键，一个数据块可能包含所有时间戳在-∞ 到 2003 年 6 月 26 日之间的文档，另一个数据块则包含时间戳在 2003 年 6 月 27 日到 ∞ 之间的文档。

16 Neo4J & Graph Computing

1. 图就是节点和边的集合，每个节点也可以有不同的含义，边也是
2. 这个 graph DBMS 引擎类似的，会有两层
    - The underlying storage：用于存储结构之类的
    - The processing engine：用于处理查询的优化之类的
3. 问题：在关系型数据库里缺乏 relationship 的表示，需要做多次 join，而且 join 会有很多的性能问题（费时）（还是可以做的），在 ppt 的例子里、反过来做、如果问谁买了土豆、可能更难做。（join 的频繁会导致很差的性能，对应于 ppt 上的第二个例子，即使对于简单的需求，也需要做多次 join）
4. nosql 也缺乏 relationship 的表示、对于 ppt 上的例子，如果正向还是比较好做的、但是如果反向了就不好做了（比如要知道谁买了一个 item ，必须得扫）
5. 图数据库的区别：如果找单个节点的所有关系、只需要从一个节点开始、然后沿着边找就可以了（不需要 join）（就不需要访问所有的 nodes 和所有的边）
6. A labeled property graph is made up of nodes, relationships, properties, and labels.
    - Nodes contain properties.
    - nodes 上面是可以有 label 属性的. Labels group nodes together, and indicate the roles they play within the dataset.
    - A relationship always has a direction, a single name, and a start node and an end node
7. 图查询语言：Cypher
    - ASCII 的表示：(emil:Person {name:'Emil'}) <-[:KNOWS]-(jim:Person {name:'Jim'}) -[:KNOWS]->(ian:Person {name:'Ian'}) -[:KNOWS]->(emil)
    - 图的描述方式不唯一
    - 基于上面这个用 cypher 描述图的方式、就可以用 cypher 去进行 query

```
MATCH (a:Person)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
WHERE a.name = 'Jim'
RETURN b, c
```

8. 对于 ppt 上的例子，如果用关系型数据库来表示的话，会有需要不同张表示一对多关系这些的表

```
MATCH (user:User)-[*1..5]-(asset:Asset) //通过1条到5条边找到所有的 asset
WHERE user.name = 'User 3' AND asset.status = 'down' RETURN DISTINCT asset //返回所有的user3可以访问的down的asset
```

9. 对于上面的查询语句、会把上面转化成下面这样的很多的搜索路径(比关系型数据库里的很多 join 操作要好)

```
(user)-[:USER_OF]->(app)
(user)-[:USER_OF]->(app)-[:USES]->(database)
(user)-[:USER_OF]->(app)-[:USES]->(database)-[:SLAVE_OF]->(another-database)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)-[:HOSTED_BY]->(server)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)-[:HOSTED_BY]->(server)-[:IN]->(rack)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)-[:HOSTED_BY]->(server)-[:IN]->(rack) <-[:IN]-(load-balancer)
```

9. Use nodes to represent entities—that is, the things in our domain that are of interest to us, and which can be labeled and grouped.
    - Use relationships both to express the connections between entities and to establish semantic context for each entity, thereby structuring the domain.
    - Use node properties to represent entity attributes, plus any necessary entity metadata, such as timestamps, version numbers, etc.
10. 有一个例子、比如说可以做一个简单的推荐系统、比如说一本书被几个读者所喜欢、可以给其中一个读者推荐喜欢这本书的其他人的看过的其他书推荐给这个读者
11. neo4j 可以作为嵌入式进程地去跑、也可也作为一个独立的服务器进程地去跑
    - 可以做分布式存储，也可以做主从备份
12. 所有节点和边都是等长的
    - node： 15 bytes，里面只存第一条边
    - relationship： 34 bytes：里面存了节点的下一条边和这条边的属性
    - 见 ppt 上有一个存储结构的例子
13. 需要注意的是、这里的 neo4j 类里的 Id 注解应该是从 neo4j 包里导入的，而不是从 Jakarta persistence 里面导入的、不然会在启动的时候报错找不到对应的 neo4j 主键的错误
14. 貌似这里有几页 ppt 没听，可以听一下回放，貌似后面的都是一些例子

17 Log-Structured Database & Vector Database

1. 先讲了需要给 redis 的信息加入一个超时的时间，如果一开始的时候、就把所有书都放入到 redis 里，当 3600s 到了之后，那么这些 book 对象全部失效了，那么大量的访问书的请求就需要重新去数据库里拿。这个情况就是雪崩的情况。如何解决呢、比如说可以把这个 10K 的分成 10 个 1K 的，然后让他们逐个失效。第二个情况是、如果 10w 个请求同时读一个不在 redis 里的数据、不加处理的话、会导致 10w 个全部往数据库里去拿、这个就是击穿的情况。如何解决呢、可以加一个

锁，当所有请求都访问一个数据的时候、只有一个请求去数据库里拿。第三个情况是，大量访问数据库中不存在的数据，那么全都会访问数据库，这个就是缓存的穿透现象。如何解决呢，可以在前面做一个过滤、比如用布隆过滤器，可以马上告诉你这个数据不在数据库里。（穿透其实是一种攻击）

2. 日志结构数据库：先前的 mysql、mongo 和 neo4j，都有一个隐含的前提、就是所有的数据访问的次数都是差不多的（体现不出新订单和旧订单访问次数的差异（没法处理热点数据的处理情况）（可以用 partition 做、但是还是需要用 timestamp 做、再根据其他字段做子分区也不方便）），所以就有了日志结构数据库。
3. 日志结构数据库用了 LSM-tree
    - Log 需要提高写入性能
    - Log 以 Append 的模式追加写入，不存在删除和修改
    - 这种结构虽然大大提升了数据的写入能力，却是以牺牲部分读取性能为代价，故此这种结构通常适合于写多读少的场景
    - C0 树（常驻内存）
    - C1-N 树(位于磁盘)
4. 因为要落盘的时间、所以会有 immutable memtable （rockdb 做了优化、这个 immutable table 可能会有多个）
5. **Immutable Memtable 的概念**
   Immutable Memtable 是一种在数据写入后变为不可修改的 Memtable。它通常是在以下情况下使用的：
    - 当内存中的 Memtable 达到一定大小时，它会变为不可变的（Immutable）。
    - 不可变的 Memtable 会被标记为“不可修改”状态，任何新的写入操作都不会影响这个 Memtable。新的数据会被写入到一个新的 **Memtable** 中。
    - 一旦这个不可变的 Memtable 被生成，它会被异步地刷写到磁盘上的 SSTable（Sorted String Table）文件中。通过这种方式，数据库可以持续地将内存中的数据持久化到磁盘，同时又能避免频繁的磁盘写入操作。
6. **为什么使用 Immutable Memtable**
   使用 **Immutable Memtable** 有几个优点：
    - **提高写入性能**：由于 Memtable 是内存中的数据结构，当数据被写入到 Memtable 后，它不会立即写入磁盘，而是会先暂存到内存中。当 Memtable 满了之后，它会变成 Immutable，避免了频繁的写入磁盘。新的写入操作会被加入到新的 Memtable 中。
    - **减少锁的争用**：当 Memtable 一旦变为 Immutable 后，它不再允许写入，因此可以避免多个线程在写操作时产生锁的争用，提升并发性能。
    - **异步持久化**：通过将数据从 Immutable Memtable 批量地刷新到磁盘，数据库可以通过异步方式持久化数据，避免每次写入都发生磁盘 I/O 操作，从而提高系统性能。
7. **工作流程**
    1. **写操作**：当用户执行写操作时，数据首先被写入到 Memtable 中。如果 Memtable 是不可变的，那么数据会被写入到一个新的 Memtable 中。
    2. **转为 Immutable Memtable**：当 Memtable 达到一定大小时，它会变成 Immutable Memtable。
    3. **刷写到磁盘**：Immutable Memtable 被异步刷写到磁盘，通常是以 **SSTable** 格式进行持久化。
    4. **清理**：一旦 Immutable Memtable 被成功写入磁盘，旧的 Memtable 会被释放或清空，以为新的 Memtable 留出空间。

8. **总结**

   - **Immutable Memtable** 是内存表的一种优化形式，它在数据写入后变为不可变，并且在满了之后会将数据批量写入磁盘。它主要用于提升数据库的写入性能，减少内存操作和磁盘操作之间的冲突，提供异步持久化的能力，从而实现高效的数据存储和查询。

9. 要先写到 WAL log （write ahead log）里面、才能再写到内存里面

   在 **LSM-Tree (Log-Structured Merge-Tree)** 中，**WAL (Write-Ahead Log)** 的作用与在传统的数据库系统中类似，它是一种确保数据持久性和一致性的重要机制。LSM-Tree 是一种专为优化写入和读取负载设计的数据结构，通常用于 NoSQL 数据库（例如：LevelDB、RocksDB 等）。在 LSM-Tree 中，WAL 主要是用来处理写入操作，并确保即使在系统崩溃时，数据也不会丢失。

LSM-Tree 中的 WAL 工作原理

LSM-Tree 的设计理念是通过将所有写操作先写入内存中的 **MemTable**，然后定期将 MemTable 的内容以批量的形式合并到磁盘上的 **SSTable**（Sorted String Table）中，来优化写入性能并降低磁盘 I/O。但是为了确保数据的可靠性，所有对 MemTable 的修改都会先记录到 WAL 中。

WAL 在 LSM-Tree 中的流程

1. **写入 WAL**：

   - 每当有新的写操作（如插入、更新或删除）时，LSM-Tree 会先将这些操作记录到 WAL 中，而不是直接修改 MemTable 或磁盘上的 SSTable。
   - 这种写操作会以 **顺序日志** 的形式保存在 WAL 中，这样可以减少随机写入的开销，提高性能。

2. **MemTable 更新**：

   - 在写入 WAL 后，数据会被写入到 MemTable 中。MemTable 是一个内存中的结构，它会不断地接受写入操作，直到 MemTable 满。

3. **刷新 MemTable 到 SSTable**：

   - 当 MemTable 满了之后，它会被刷新到磁盘上的 SSTable 中。这个过程涉及将 MemTable 中的数据持久化到磁盘，并将其与现有的 SSTable 合并。这里的合并过程可能会产生新的 SSTable 文件。

4. **故障恢复**：

   - 如果在将 MemTable 中的数据刷新到磁盘之前，系统发生崩溃或故障，WAL 中的日志记录可以用于恢复数据。系统会在启动时读取 WAL 日志，重新执行日志中的写操作，并将这些数据恢复到 MemTable 中。这样，即使系统在数据写入磁盘之前崩溃，也能保证数据的完整性。

5. **删除旧的 WAL**：

   - 当 MemTable 被成功写入到 SSTable 后，WAL 文件就可以被丢弃或重用。这是因为 MemTable 中的内容已经被永久保存到磁盘。

WAL 在 LSM-Tree 中的优势

1. **保证数据持久性**：

   - WAL 提供了故障恢复机制，即使在系统崩溃的情况下，未持久化到磁盘的数据仍然可以通过 WAL 恢复。

2. **提高写入性能**：

   - 写入 WAL 是顺序操作，减少了磁盘的随机写入，提升了性能。这与 LSM-Tree 的设计哲学一致，旨在优化写入性能。

3. **避免数据丢失**：

- 即使 MemTable 在刷写前发生了崩溃，WAL 中的日志可以确保这些数据在崩溃后不会丢失。

4. **延迟合并操作**：

- LSM-Tree 延迟将 MemTable 中的数据合并到 SSTable 中，这有助于减少写入操作的频率。WAL 记录使得这一过程在后台进行，不影响数据的一致性。

LSM-Tree 中 WAL 和传统数据库 WAL 的区别

- **传统数据库的 WAL**：在传统的关系型数据库中，WAL 通常是为了保证每个单独的事务一致性和持久性。而 LSM-Tree 中的 WAL 更多的是作为一个写入缓存，确保数据能够在崩溃时恢复，避免丢失数据。
- **写入方式**：传统数据库系统在每次数据修改时会先写 WAL，再进行磁盘更新；而在 LSM-Tree 中，写操作通常会先写入 WAL，并写入 MemTable（内存中），然后定期将数据批量刷写到磁盘的 SSTable 中。

LSM-Tree 中的 WAL 需要注意的问题

1. **WAL 大小**：WAL 文件可能会变得很大，特别是在系统大量写入时。通常系统会定期清理 WAL 文件（例如，在 MemTable 被成功持久化之后）。
2. **恢复效率**：恢复过程中，如果 WAL 中有大量未写入磁盘的记录，可能会影响恢复时间。不过，LSM-Tree 的合并策略（例如，通过 `compaction`）可以有效地减少 WAL 恢复时的工作量。

10. 优点
    - 大幅度提高插入（修改、删除）性能
    - 空间放大率降低（可能会产生不同版本的数据，一个数据多个版本在里面的话、空间放大率就会变大）
    - 访问新数据更快，适合时序、实时存储（适合存双十一的订单）
    - 数据热度分布和 level 相关（经常不被访问到的数据就会沉到底层）
11. 缺点
    - 牺牲了读性能（一次可能访问多个层）
    - 读、写放大率提升
12. ppt p6 有 sstable 的定义
13. 会有一个 shared key 的概念、就是 abcd、abcd、abcf 前面共同的前缀 abc 就是 shared key（一个同样的 shared key 会存在一个 record group 里面）
14. 写放大：一直要合并到最后一层的时候，会涉及到很多层的合并
15. L0 是无序的，L1-Ln 是有序的
16. 读放大：比如读一个 key=61 的，但是不存在这个数据库里，如果每一层都找不到、就要一路找到最底层（如果不存在布隆过滤器的时候）
17. 老的数据也会在 compaction 的时候被 remove 掉（所有的 sstable 都不可更改、都是在 compaction 的时候重新写进行处理的）
18. 给出的 rocksdb 的例子是嵌入式执行的
19. 热点数据不是一个大量的数据，只需要尽可能快地访问到热点数据就可以了。如果热点数据不再成为热点数据之后、应该把高层的数据放到其他的数据库里（因为会有读放大和写放大）

20. HTAP Hybrid Transactional/Analytical Processing 混合事务分析处理 （这里提到了数据是应该按行存储还是按列存储）（因为在一个系统里面，下面这两种需求都会有，最好不要只有一种、应该混合，或者也可以按行存一轮、按列存一轮（但是也会有缺点、就是空间的浪费和维护一致性的两个问题））
21. Online Transaction Processing(OLTP) （要对随机的读写做支持，要做索引的维护、随机访问数据，可以按行存）（这里就不能按列存、因为需要在不同列地方存同一个订单的信息）
    - 在线事务处理（订单之类的）
    - 低延迟
    - 高并发
    - 数据量小
22. Online Analytical Processing(OLAP)
    - 在线分析处理（统计之类的）（这个如果按行存储的话、导致每一行的长度不一样、会造成很多的随机访问、所以可以按列存储，这样按顺序读就很快）
    - 高延迟
    - 低并发
    - 大量数据
23. LSM-Tree 抽象结构
    - 分层结构
    - 提供写优化
    - TP 事务友好
24. 这里有一个列族的概念（相当于垂直分区，但是管理起来比较麻烦、但是 LSM-tree 在 rocksDB 里面可以支持（可以共用一个 WAL 日志和 manifest files）
    - 比如说可以对和 sale 相关的信息用列存、然后做 OLAP、放在一个列族里面，用 lsm-tree 来存储列信息
    - 然后其他的数据用 row 存、然后支持 OLTP、存在另一个列族里面，用 lsm-tree 来存储行信息
25. RocksDB 写流程
    - 优 - 低延迟插入
    - 写入内存后直接返回
    - 后台异步写入磁盘
    - 劣 - 写阻塞问题
    - L0 层满时将阻塞内存到磁盘的 Flush 过程
    - L0 层下沉 Compaction 过程无法多任务执行
    - 异步写写放大严重，容易磁盘变成瓶颈
26. RocksDB 读流程
    - 劣 - 读放大问题
    - 不同层级存储着不同版本的数据
    - 需要访问所有可能的数据文件
27. 写阻塞问题降低了 TP 事务的可用性，读放大问题限制了 AP 查询的性能

28. 解决写阻塞问题，在 RocksDB 和数据源之间增加一个收集分发层，写阻塞时缓存数据（collector）（基于 apache flink）

- 采用中心化设计：
  Master 采用主从备份
  监控集群负载，
  调控负载均衡。
- 负载均衡策略基于剩余内存大小，
  即分配到某个节点的概率与剩余内存大小正比
- 在 RocksDB 和数据源之间增加一个 **收集分发层 (collector)**，并基于 **Apache Flink** 来缓存数据，旨在解决写阻塞问题。这个 **collector** 的作用是管理、缓存和转发写入数据，以减少写操作对数据库性能的影响。具体来说，它会在以下几个方面起到作用：

1. **缓解写阻塞（Write Backpressure）**

- 在传统的写入过程中，如果 RocksDB 因为写入压力过大或者硬盘 I/O 瓶颈导致的写阻塞，数据的写入就会变得非常缓慢，甚至可能导致系统崩溃。
- **Collector** 层将充当一个缓冲区，接收从数据源发来的写请求。当 RocksDB 无法立即处理写入时，Collector 会先缓存这些写入数据。缓存区的存在避免了直接的写阻塞现象，从而减少了系统的压力。

2. **流量控制与数据流优化**

- 基于 **Apache Flink** 这样的流处理框架，Collector 可以实时地收集数据，并通过 Flink 的流处理机制来优化数据的流向和批处理过程。
- Flink 能够提供 **流量控制**，当 RocksDB 压力过大时，Flink 可以减缓写入速度，或者通过批量处理来提升写入的效率，从而避免数据堆积。
- **Flink** 的时间窗口和流式操作（如批处理）可以帮助以更高效的方式将数据写入 RocksDB，从而缓解单个请求造成的阻塞问题。

3. **数据预处理与转换**

- 在数据写入 RocksDB 之前，**Collector** 层不仅仅是一个简单的缓存，还可以执行一些数据预处理和转换任务。例如，对数据进行格式化、过滤、聚合等操作，以减少数据写入时的计算压力。
- **Flink** 提供强大的流处理能力，可以对数据进行实时计算和预处理，然后将处理后的数据推送到 RocksDB。

4. **提高写入吞吐量（Throughput）**

- 当数据源产生大量数据时，Collector 层能够缓存这些数据，并通过 Flink 的批处理机制来优化写入过程。即使 RocksDB 一时无法承载所有写入请求，Collector 也能按照一定的批次逐步将数据写入 RocksDB，确保系统的吞吐量不受过多影响。
- Flink 的并行处理能力可以显著提高写入效率，确保即使在高并发的情况下，写操作也不会造成明显的延迟或阻塞。

5. **容错性与数据可靠性**

- Flink 提供内建的容错机制（如 **checkpointing**），使得 Collector 在出现故障时能够恢复数据，防止数据丢失。
- 通过这种机制，Collector 能够确保在面对系统故障或者写阻塞时，数据不会丢失，并且能够继续流畅地处理后续写入操作。

6. **流量平滑（Backpressure Handling）**

    - 在数据流入系统时，**backpressure** 可能会发生，尤其是当写操作过多，数据积压在 Collector 中时，Collector 可以通过 Flink 的流控制机制进行流量平滑，减缓数据的写入速度，避免系统崩溃。
29. 解决读放大问题：在 RocksDB 中增加列式存储，列式存储在访问少量列时磁盘读取量更小，可以减少读放大的开销（列式存储也是按照数据块的）
30. 提出混合存储策略
    - 常做事务的数据以行式存储，
    - 常做查询的数据以列式存储
    - 以磁盘读写开销为格式转换的指标
31. 然后讲向量数据库
    - 会有 one-hot 编码（所有的单词的点积为 0，每个单词都无关、但是这个缺点就是需要的维数太大）
    - 也会有一个对一个人的各种特征做向量化的处理（降维，然后可以计算两个向量之间的相似性，最后得到的数字都是数值型的，然后找最相似的时候，就去找和向量最相似的（最 naive 的方法，其实需要做更多预处理））
    - 所以向量数据库就是先存、然后做搜索（范围内的搜索，区别就是不是精确匹配、找的是最相似的，和 mongo 的 2d 的区别就是向量数据库支持更高维的）
32. 数据插入向量数据库之后也会需要做索引
33. 随机投影：为了降低数据的维数
    - 用一个随机的矩阵 random matrix generator，然后把原来的数据乘上这个矩阵，就可以得到一个降维或者升维的数据
34. 建立索引的时候、还可以做一个 product quantization，就是把向量分成几个部分，然后每个部分再做一个聚类（通过 codebook generator 转化成 code，然后查找的时候也把查找的 vector 变成多个 code 进行匹配）（比如就是把 500 种组合压缩成 10 种 code，这样就可以做近似匹配（因为不需要做精确匹配））
35. 还可以做位置敏感哈希 locality sensitive hashing，通过 hash 值决定它在哪个 bucket 里面，然后 query 的时候还是找最近的
36. HNSW（hierarchical navigable small world）就是多层的索引
37. 相似度计算：
    - 余弦相似度 Cosine similarity：It ranges from -1 to 1, where 1 represents identical vectors, 0 represents orthogonal vectors, and -1 represents vectors that are diametrically opposed. $\text{Cosine Similarity}(A, B) = \frac{A \cdot B}{|A| |B|}$
    - Euclidean distance:
      It ranges from 0 to infinity, where 0 represents identical vectors, and larger values represent increasingly dissimilar vectors.
    - Dot product:
      It ranges from -∞ to ∞, where a positive value represents vectors that point in the same direction, 0 represents orthogonal vectors, and a negative value represents vectors that point in opposite directions.
38. 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。

32. 也可以通过向量的元数据进行过滤，然后再进行匹配
33. **随机投影（Random Projection）**
    随机投影的基本思想是使用随机投影矩阵将高维向量投影到低维空间。
34. **（Product Quantization）**
    这种方法将原始向量分解为更小的块，通过为每个块创建代表性的“代码”来简化每个块的表示，然后将所有块重新组合在一起。
35. **局部敏感哈希（Locality-Sensitive Hashing, LSH）**
    局部敏感哈希（LSH）是一种在近似最近邻搜索中用于索引的技术。
36. **（Hierarchical Navigable Small World, HNSW）**
    （HNSW）创建一个层次化的树状结构，每个树的节点代表一组向量。节点之间的边表示向量之间的相似性。
37. **元数据（Metadata）**
    每个存储在数据库中的向量还包括元数据。除了能够查询相似向量外，向量数据库还可以根据元数据查询来过滤结果。
38. **向量索引和元数据索引**
    为了实现基于元数据的过滤，向量数据库通常维护两个索引：一个是向量索引，另一个是元数据索引。

18 Timeseries Database

1. 云数据库走 serverless 的模式
2. 时序数据库的特点
    1. 时序数据库里的数据是有一定的存活时间的，比如一个小时之内电脑的状态
    2. 数据的格式简单，数值在一定范围内波动
    3. 所以不一定需要存原值、可以存差值（difference），好处是当数值很大的时候、差值会比较少、可以节省空间
    4. 所有的数据都带 timestamp（和 sstable 一样，存的时候可以带一个前缀，比如前几个 time 是一样的、就可以合并）（也可以存差值）
3. 时序数据库的数据叫做度量，是通过监控和下采样得到的
    - 特点是数据会源源不断地来
    - 需要做数据的生命周期管理
    - 关心的不是单点的数据、而是一段时间的总结性的数据（比如平均利用率，总和、标准差）
    - 所以也需要扫描一大堆的数据（经常做的操作）
4. serverless 的会涉及到冷启动的 docker 的问题、一开始的体验不会很好
5. influxdb 用 telegraf 来采集数据
    - flux 是类似于 sql 的查询语言

6. 课上监控了 cpu 的状态
7. bucket 相当于数据库表的名字，但是没有库的概念
    - _time 是时间戳
    - _measurement 是一个 bucket 里面，在度量什么
    - _field 是字段的名字，key
    - _value 是值
    - 然后也会有 tag key 和 tag value，用来表示后面的值的特征
8. 时序数据库的索引
    - _field 是不参与索引的，因为只能在 value 上做索引、但是没有意义，这个是搜索的结果，而且取值不是唯一的，所以建索引没有意义
    - _tag 是参与索引的，因为我们经常在 tag 这一列上去做搜索和筛选，而且 tag 的值是固定的，所以建索引有意义（写时有额外开销、但是搜索快）
    - 所以其实 tag-field 之间在设计的时候是可以互相转化的，如果操作上不需要做全表的 scan、可以在根据搜索过滤的条件、把对应的数据转化成 tag，然后再去做搜索
9. series
    - 有时候查询出来的数据是一系列的，这个就是 series
    - A series key is a collection of points that share a measurement, tag set, and field key
10. point：A point includes the series key, a field value, and a timestamp.
11. bucket：A bucket combines the concept of a database and a retention period (the duration of time that each data point persists). A bucket belongs to an organization.
    - 超过一定时间会删掉数据
12. organization：An InfluxDB organization is a workspace for a group of users.
    All dashboards, tasks, buckets, and users belong to an organization.
13. InfluxDB design principle
    1. 为了提升性能、这个数据是按照时间升序排列的、所以是可以 append 来做插入的
    2. 数据是一次性写入、不允许再修改（而且不存在新版本的数据、只存在新数据，每个数据时间戳都是不一样的）（这个不能修改和日志合并树很类似）
    3. 因为有可能会在做 query 的时候、不断地有数据进来、Therefore, if the ingest rate is high (multiple writes per ms), query results may not include the most recent data.
    4. Schemaless design：Time series data are often ephemeral, meaning the data appears for a few hours and then goes away.For example, a new host that gets started and reports for a while and then gets shut down.
    5. 没有 ID（和关系型数据库不同）Points are differentiated by timestamp and series, so don’t have IDs in the traditional sense.
    6. 对于 Duplicate data，InfluxDB assumes data sent multiple times is duplicate data. Identical points aren’t stored twice.（好处就是记录的数据会进一步减少）
14. The storage engine includes the following components:
    - Write Ahead Log (WAL)
    - Cache
    - Time-Structed Merge Tree (TSM)
    - Time Series Index (TSI)

15. Writing data from API to disk
    - 数据点如果非常多地进来、会需要先进行压缩，否则 WAL 会很快地变得很大
    - points 写进来会先写到 cache 里面
    - 然后 cache 会周期性地通过 TSM 文件写到磁盘里面
    - （和 LSM 很像） As TSM files accumulate, the storage engine combines and compacts accumulated them into higher level TSM files.
16. cache
    - 通过组织一组值的 key (measurement, tag set, and unique field) （series）
17. TSM
    - Column-oriented storage lets the engine read by series key and omit extraneous data.
    - 用的是列存、因为经常要做 scan、所以用列存会更快
    - 也分为 shared key 和 non-shared key
18. TSI：time series index
    - 基于 series key 做的索引
    - The TSI stores series keys grouped by measurement, tag, and field
19. binlog 不应该和 MySQL 放在一起、因为如果遇到的是硬盘损坏、那么数据和 binlog 都会丢失、所以应该放在不同的地方
20. InfluxDB file structure
    - blotdb 用来存小的用户数据
    - Configs path 用来存 token 之类的内容
21. InfluxDB shards and shard groups
    - 大量的数据进来、存不下、就用 shard
    - A shard contains encoded and compressed time series data for a given time range defined by the shard group duration.（压缩过的数据）
    - 会根据不同的时间段来存不同的 Shard group （设置 duration）
    - 只有这个时序数据库会自动帮你删除、其他种类的数据库都需要自己去删除（主要是对于原始的数据点不太关心、关心的是一段时间的总结性数据）
22. shard life circle
    - 可以预先做 Shard precreation，然后之后写的时候就会快一点
    - InfluxDB writes time series data to un-compacted or “hot” shards. When a shard is no longer actively written to, InfluxDB compacts shard data, resulting in a “cold” shard.
    - Shard compaction（Level 1 (L1)（在内存里）: InfluxDB flushes all newly written data held in an in-memory cache to disk. Level 2 (L2): InfluxDB compacts up to eight L1-compacted files into one or more L2 files ）
    - 也可以设置 retention forever、然后就永远不会删除数据
23. 这里的代码例子上课的时候没听、复习的时候可以听一下
24. **InfluxDB 是时间序列平台**
    InfluxDB 是一个为处理时间序列数据而设计的平台。它能够帮助你在更短的时间内、用更少的代码构建实时分析、物联网（IoT）和云原生服务应用程序。它的核心是一个专门为处理由传感器、应用程序和基础设施生成的海量时间戳数据而优化的数据库。

25. **时间戳（Timestamp）**
    所有存储在 InfluxDB 中的数据都包含一个 _time 列，该列存储时间戳。时间戳在磁盘上以“纪元纳秒”（epoch nanosecond）格式存储，而 InfluxDB 格式化时间戳以显示与数据相关的日期和时间，采用 RFC3339 UTC 格式。写入数据时，时间戳的精度非常重要。

26. **度量（Measurement）**
    _measurement 列显示度量的名称。例如，census（人口普查）。度量作为一个容器，存储标签、字段和值以及时间戳。度量名称应描述数据的内容。比如，census 这个名称表明字段值记录的是蜜蜂和蚂蚁的数量。

27. **字段（Fields）**
    字段包含字段键（field key）和值（field value）。字段键存储在 _field 列中，而字段值存储在 _value 列中。字段值可以是字符串、浮动值、整数或布尔值。

    - **字段键（Field Key）**：字段键是一个字符串，表示字段的名称。
    - **字段值（Field Value）**：字段值表示与字段相关联的具体值。

    字段集合（Field Set）是一个在特定时间戳下与多个字段键值对相关联的数据集合。

28. **标签（Tags）**
    标签包括标签键（tag key）和标签值（tag value），标签键值对是字符串类型的元数据。标签用于为数据添加上下文信息，通常用于筛选、查询和索引数据。

    - **标签键（Tag Key）**：标签键是标签的名称。
    - **标签值（Tag Value）**：标签值是标签键的具体值。

    例如，标签键 “location” 有两个标签值：klamath 和 portland。标签键 “scientist” 也有两个标签值：anderson 和 mullen。

    标签集合（Tag Set）是由标签键值对组成的数据集合。

29. **字段不被索引**
    字段在 InfluxDB 中是必须的，但它们不会被索引。如果查询中需要过滤字段值，InfluxDB 将必须扫描所有字段值，以匹配查询条件。因此，基于字段的查询性能比基于标签的查询差。为了优化性能，应该将常用的元数据存储在标签中。

30. **标签被索引**
    标签是可选的。你可以选择不在数据结构中使用标签，但通常最好包含标签。因为标签是被索引的，所以基于标签的查询比基于字段的查询更快。这使得标签非常适合存储常用的元数据。

31. **你的数据模式很重要**
    如果大部分查询关注的是字段中的值，例如查询在何时计数到 23 只蜜蜂，InfluxDB 会扫描数据集中所有字段值来寻找符合“蜜蜂”的条件，然后返回结果。因此，查询字段值时可能会导致性能下降。

32. **如何优化数据模式**
    如果你的数据量增长到数百万行，为了优化查询，你可以重新安排数据模式，将字段（例如蜜蜂和蚂蚁）变成标签，将标签（例如位置和科学家）变成字段。这种方式可以提升查询性能，因为查询基于标签时会更快。

33. **桶模式（Bucket schema）**
    在 InfluxDB Cloud 中，带有明确模式类型的桶（bucket）需要为每个度量（measurement）指定明确的模式。每个度量包含标签、字段和时间戳。明确的模式会限制可以写入该度量的数据结构。例如，对于“人口普查（census）”数据，明确的模式会约束该数据的格式。

34. **Series（系列）**
    一个系列键（series key）是由一组共享相同度量（measurement）、标签集合（tag set）和字段键（field key）的点（point）组成的。例如，示例数据包含两个唯一的系列键：

- 一个系列包括与该系列键相关的时间戳和字段值。从示例数据来看，这里有一个系列键及其对应的系列数据。

35. **Point（点）**
    一个点（point）包含了系列键、字段值和时间戳。例如，从示例数据中，一个单独的点的样式如下所示：
    - 一个点可以看作是数据记录的基本单位，它由一个系列键、一个字段值和一个时间戳组成。
36. **Bucket（桶）**
    所有的 InfluxDB 数据都会存储在一个桶（bucket）中。桶结合了数据库和数据保留时间（每个数据点存在的时长）的概念。每个桶都属于一个组织（organization）。
    - 例如，你可以理解为桶是数据存储的容器，它将数据按照组织划分，同时还规定了数据的保留期限。
37. **Organization（组织）**
    一个 InfluxDB 组织是一个用于一组用户的工作空间。所有的仪表板（dashboard）、任务（task）、桶（bucket）和用户（user）都属于某个组织。
    - 组织是 InfluxDB 中管理和分配资源的单位，它将用户、数据和任务等关联起来。
38. **时间顺序的数据**
    为了提高性能，InfluxDB 要求数据按时间升序写入。
    这意味着数据会按照时间顺序进行写入，从而优化写入操作的效率。
39. **严格的更新和删除权限**
    为了提高查询和写入性能，InfluxDB 严格限制了更新和删除操作的权限。
    时间序列数据主要是新增数据，不会进行更新。删除操作通常只会影响那些不再写入的数据，而冲突的更新操作很少发生。
40. **优先处理读写查询**
    InfluxDB 优先处理读写请求，而不是强一致性。
    在执行查询时，InfluxDB 会立即返回结果，查询数据后涉及的数据事务会被随后处理，以确保数据最终一致。因此，如果写入速率很高（每毫秒有多个写入），查询结果可能不会包含最新的数据。
41. **无模式设计**
    InfluxDB 使用无模式设计来更好地管理不连续的数据。
    时间序列数据通常是短暂的，可能只存在几个小时。例如，一个新的主机启动并报告数据一段时间后再关闭。
42. **数据集比单个点更重要**
    因为数据集比单个数据点更重要，InfluxDB 实现了强大的工具来聚合数据和处理大规模数据集。
    数据点是通过时间戳和系列来区分的，而不是通过传统的 ID。
43. **重复数据**
    为了简化冲突解决并提高写入性能，InfluxDB 假设多次发送的数据是重复数据。
    相同的数据点不会被存储两次。如果一个数据点的字段值发生变化，InfluxDB 会用最新的字段值更新该点。在极少数情况下，数据可能会被覆盖。
44. **存储引擎确保**：
    - 数据被安全地写入磁盘。
    - 查询的数据是完整且正确的。
    - 数据的准确性优先，其次是性能。
45. **存储引擎的组成部分**：
    - 写前日志（WAL）
    - 缓存

- 时间结构合并树（TSM）
- 时间序列索引（TSI）

46. **从 API 写入数据到磁盘**：
    - 存储引擎处理从接收到 API 写入请求到将数据写入物理磁盘的整个过程。
    - 数据使用行协议（line protocol）通过 HTTP POST 请求发送到 `/write` 端点。
    - 数据批量发送到 InfluxDB，被压缩后写入 WAL，以确保数据即时持久化。
    - 数据点还会被写入内存缓存，并立即可查询。
    - 内存缓存定期写入磁盘，形成 TSM 文件。
    - 随着 TSM 文件的积累，存储引擎会将这些文件合并并压缩成更高级别的 TSM 文件。
    - 尽管数据点可以单独发送，但为了提高效率，大多数应用程序会批量发送数据点。
47. **写前日志（WAL）**：
    - 写前日志（WAL）保证了 InfluxDB 数据的持久性，即使存储引擎重启，也能保留数据。
    - 当存储引擎收到写请求时，以下步骤会发生：
        - 写请求会追加到 WAL 文件的末尾。
        - 数据会通过 `fsync()` 写入磁盘。
        - 内存缓存会更新。
        - 数据成功写入磁盘后，会返回一个确认响应，表示写入成功。
48. **缓存**：
    - 缓存是 WAL 中当前存储的数据点的内存副本。
    - 缓存的功能：
        - 按键（测量、标签集和唯一字段）组织数据点，每个字段按照时间顺序存储在自己的范围内。
        - 存储未压缩的数据。
        - 每次存储引擎重启时，缓存会从 WAL 中获取更新数据。
        - 缓存会在运行时被查询，并与存储在 TSM 文件中的数据合并。
49. **时间结构合并树（TSM）**：
    - 为了有效地压缩和存储数据，存储引擎将字段值按系列键分组，并按时间对这些字段值进行排序。
    - 存储引擎使用时间结构合并树（TSM）数据格式。
    - TSM 文件以列式格式存储压缩后的系列数据。
    - 为了提高效率，存储引擎仅存储系列中值的差异（即增量）。
    - 列式存储使得引擎可以按系列键读取数据，省略不必要的数据。
50. **时间序列索引（TSI）**：
    - 随着数据的基数（系列的数量）增加，查询读取的系列键增多，查询变得更加缓慢。
    - 时间序列索引（TSI）保证在数据基数增长时，查询仍然保持高效。
    - TSI 按测量、标签和字段将系列键分组存储，这使得数据库能够很好地回答两个问题：
        - 存在什么样的测量、标签、字段？（发生在元数据查询中）
        - 给定一个测量、标签和字段，存在哪些系列键？
51. **Shard（分片）**：

- 分片包含指定时间范围内的编码和压缩后的时间序列数据，时间范围由**分片组时长**（shard group duration）定义。
- 同一时间段内的所有数据点都会存储在同一个分片中。
- 每个分片包含多个系列（series），一个或多个 TSM 文件，并且属于某个**分片组**。

52. **Shard Group（分片组）**：
    - 分片组属于一个 InfluxDB 的桶（bucket），包含特定时间范围内的时间序列数据，时间范围由**分片组时长**定义。
    - 在**InfluxDB OSS**中，通常每个分片组只包含一个分片。
    - 在**InfluxDB Enterprise 1.x**集群中，分片组会包含多个分片，分布在多个数据节点上。
53. **Shard Group Duration（分片组时长）**：
    - 分片组时长指定了每个分片组的时间范围，并决定了创建新分片组的频率。默认情况下，InfluxDB 会根据桶的**数据保留周期**（retention period）设置分片组时长。
54. **Shard Group Diagram（分片组示意图）**：
    - 下面的示意图展示了一个保留周期为 4 天、分片组时长为 1 天的桶结构。
55. **Shard Precreation（分片预创建）**：
    - InfluxDB 的分片预创建服务根据分片组时长为每个分片组预先创建未来的分片，定义了分片的开始和结束时间。
    - 该预创建服务不会为过去的时间范围预创建分片。
    - 在回填历史数据时，InfluxDB 会按需创建历史时间范围的分片，这可能会导致写入吞吐量暂时下降。
56. **Shard Writes（分片写入）**：
    - InfluxDB 将时间序列数据写入未压缩的“热”分片。当一个分片不再被写入时，InfluxDB 会对该分片的数据进行压缩，形成“冷”分片。
    - 通常，InfluxDB 会写入最新的分片组，但在回填历史数据时，InfluxDB 会写入老的分片，这些分片需要先进行解压缩。回填完成后，InfluxDB 会重新压缩这些老的分片。
57. **Shard Compaction（分片压缩）**：
    - InfluxDB 会定期压缩分片，以压缩时间序列数据并优化磁盘使用。InfluxDB 有四个压缩级别：
        - **Level 1 (L1)**：将所有新写入的数据从内存缓存写入磁盘。
        - **Level 2 (L2)**：将最多八个 L1 压缩后的文件合并成一个或多个 L2 文件，通过将包含相同系列的多个块合并为更少的块来优化存储。
        - **Level 3 (L3)**：对 L2 压缩文件中的块进行处理，将多个包含相同系列的块合并成一个新块。
        - **Level 4 (L4)**：完全压缩——对 L3 压缩的文件块进行迭代，将多个包含相同系列的块合并成一个新块。
58. **Shard Deletion（分片删除）**：
    - InfluxDB 的**数据保留服务**定期检查是否有分片组的时间超出了桶的保留周期。
    - 一旦分片组的开始时间超过桶的保留周期，InfluxDB 会删除该分片组及其相关的分片和 TSM 文件。
    - 对于具有无限保留周期的桶，分片将永久保留在磁盘上。

19 GaussDB

1. 不是自己写的、是基于 postgresql 等数据库的，主要是为了做分布式（比如分布式里会有 2PC 之类的）
2. GaussDB 技术架构
    - 分布式优化
    1. 分布式近数计算：数据的计算的位置应该尽量靠近数据节点的位置，效率能够提高
    2. 全链路编译执行：类似于 prepared statement，可以提前编译好、然后把参数传进去再执行、这样可以提高效率（用的是 JIT）
    3. 大规模并发事务处理
    4. sql 解析：类似于编译器的优化（query plan），
    5. RBO：比如可以基于一些数据库的规则、把一些查询优化成更好的查询（基于规则的优化）根据预定义的启发式规则对 SQL 语句进行优化（比如贪心算法，有一个全局最优解、但是算不出来）（还有一个 meta-heristic，可以跳出局部最优解找出全局最优解）
    6. CBO：基于查询的代价来做优化
    7. Hint：辅助优化器的工具
    8. postgres 没有多线程、gaussdb 做了线程池
    9. 向量化：提高多核效率（充分利用多核的效率、比如处理 tensor 的时候，一次一批做（batch））
    - 多地容灾
    1. 多地多活容灾（两地三中心，另一个城市用于容灾）
    - 云原生弹性伸缩
    1. 纵向：增加进程可用的内存、cpu
    2. 横向：增加计算节点服务器
    3. 多租户（重要）
    - 智能优化
    1. ABO：AI based optimization（基于怎么样的 ai 模型能做出更好的优化）
    - 安全隐私
    1. 防篡改：比如用了区块链技术
3. coordinator node 用来协调（比如分布式事务），data node 用来存数据和执行计算
4. 优化器优化出来的查询可能对某些数据库表现好、有些数据库表现不好
5. 会有缓存层和存储层之间的 RDMA（Remote Direct Memory Access）（不需要 CPU 的参与、直接在内存和硬盘之间传输数据）
6. 多租户：主要问题在于维护后期的差异、比如不同的租户会有不同的数据表的要求，有四个方案
    - 可以用两个分别的小的数据库
    - 可以用一个大的数据库，然后在一个大数据库里面维护两张表
    - 可以在一张表里加一个字段是租户 id，然后在查询的时候加一个条件，然后增加修改字段的时候可以用外键关联（相同的列存在一张表里）
    - 可以把列转成行去做，这样就可以维护多租户（主键+列名+取值），缺点是这张表可能会很大
    - GaussDB 实现了多租户、而且可以选择方式（但是不同的方式代价不一样）

    - 可能形式上有 6 个数据库、但是实际上只有 5 个
    - 相同的数据库最好只有一个实例
7. 查询重写中的谓词下推：
    - 有两个 data node、一种方法是通过把数据表上推到 coordinator node 做计算，另一种方法是把计算下推到 data node 上去做（比如把 t1 和 t2 表的连接操作推到 data node 上去做）
8. GaussDB 计划生成：GaussDB 主要采用自底向上与随机搜索相结合的方式
9. 全局计划缓存：每个 session 可以复用全局其他 session 已经执行过的缓存、提高效率
10. 页面结构：行存储以页面为单位
    - 因为可能会有 varchar 这样每个元素不一样的字段、所以需要 pointer 的槽位去指向这些元组的位置
    - 元组信息前面四个比较重要，需要做 MVCC（因为每个元组的版本号是不一样的）
    - xmin 和 xmax 决定了元组的生命期，亦即该版本的可见性窗口。（ppt 上有一个表格）
11. ppt 例子是在同一个事务里做两次更新，同一个事务里对一个记录改两次、就会改动 t_cid。而 t_ctid 指向的是最新的记录，页面中的第几个元组，有新的变动的时候就会更新这个值。
    - 保存历史数据：为了做回滚和做快照
12. CSNLog：记录 XID 与 CSN 的映射关系，为每个事务生成一个唯一递增的 CSN，用于将事务与其可见性进行关联
    - Clog：记录事务 ID 的运行状态：运行中/提交/回滚
    - 当事务结束后，使用 CLOG 记录是否提交，使用 CSNLOG 记录该事务提交的序列
13. 有一个 Ring Buffer？环形缓冲区
14. 列存里面可以用 min 和 max 做 index
15. 预分片：预先分配好 shard，然后在写入的时候就不用再去分配 shard 了
16. 热备：请求同时发到两个节点上，然后一个节点挂了，另一个节点可以继续工作，用户无感知（能实现 RPO=0，但是是有资源的浪费的代价）
17. 暖备：请求只发到一个节点上，然后另一个节点挂了，用户会感知到（fail 了，需要重发）
18. 定期切换冷备中的主从节点：为了知道这个冷备的节点是不是崩了
19. 讲了一个 sql 注入的例子和一个跨站脚本攻击的例子

20 Data Lake

1. 如果有很多数据分布在不同种类的数据库中、需要抽取到一个统一的地方然后，访问的时候对这个地方做 OLAP。但是会有很多 ETL 的操作（抽取、转换、加载）
2. 所以数据湖就是为了解决上面的问题，直接存储 raw data（各种不同的数据源的不同格式的数据）
3. 所以数据湖需要很强的数据接入能力（通过 flink 工具）
4. 可以通过 metadata 来发现需要操作哪些类的数据库，然后通过 calcite 来做针对不同的查询语言的转换
5. 数据湖和数据仓库有不同功能和特点的对比
6. filter 需要进行实时的流式处理（比如推文的立即过滤）、原来的 hdfs （批处理）对于这种实时处理支持不太好，因此产生了 lambda 架构（batch 和 stream 两种方式）
7. 也可以控制流计算的时间窗口、比如 5s 一个窗口，然后统一处理（流批一体）
8. 更新的时候不做 ETL、萃取、沉淀的时候做 ETL

9. datalake 的意义还在于可以支撑更多的数据分析、比如机器学习、数据挖掘，而不仅仅是直接做完 etl 之后进入 data warehouse 的数据分析
10. incremental ETL：按需去做对应数据的 ETL，而不是全部的数据都做 ETL
11. 边缘计算就是基站里的这些服务器里也有 cpu 和资源，可以利用这些资源做一些计算（但是计算能力有限）
12. 比如一辆车开、它会不断地把数据发到相邻的边缘服务器里面（因为总是在连接到边缘节点里，压缩后传输可以减少 edge 到 cloud 的带宽占用），然后定期地从边缘节点同步数据到云中心节点里面（所以有些 query 既需要在中心 cloud server 里面找、也要在 edge server 里面找）
13. 边缘存储架构比较简单、是 tag 或者是 key-value、因为计算资源有限
14. 边缘节点会不断地发数据流元数据到云数据中心，然后之后的查询就会知道在哪些边缘节点去查找了
15. Rosetta 过滤器：多层 bloom 过滤器，减少假阳性
16. 这里的下推是指把计算下推到边缘节点去做、比如先在边缘节点做过滤和聚合（比如做 group by），然后再把数据传到云中心节点去做 union 和 aggregate（这里再做 count）
    - 好处就是原来是把整表拿上来、造成数据传输很多，现在是过滤了一下，所以数据传输就会少
    - 然后还有一个好处就是利用了节点的计算能力，减少了云中心节点的压力，整体性能也会提高
17. 核心问题两个：
    - 定位数据
    - 并行处理
18. **数据湖定义**：数据湖是一个系统或数据仓库，用于以其原始或自然格式存储数据，通常以对象存储（blobs）或文件的形式存在。
19. **数据湖的内容**：数据湖通常包含一个数据存储库，内含源系统数据的原始副本、传感器数据、社交数据等，以及经过转换用于报告、可视化、高级分析和机器学习的处理数据。
20. **数据类型**：数据湖可以包含多种类型的数据，包括结构化数据（关系型数据库中的行和列）、半结构化数据（如 CSV、日志、XML、JSON）、非结构化数据（如电子邮件、文档、PDF）和二进制数据（如图像、音频、视频）。
21. **部署方式**：数据湖可以部署在“本地”数据中心（组织内部）或“云端”使用云服务（如亚马逊、微软、Oracle Cloud 或谷歌云等提供商）。
22. **数据湖的功能**：数据湖是一个集中式存储库，旨在存储、处理和保护大量的结构化、半结构化和非结构化数据。它可以以原始格式存储数据，并处理各种类型的数据，不受数据大小的限制。
23. **数据湖的优势**：数据湖提供一个可扩展和安全的平台，允许企业：
    - 从任何系统以任何速度摄取数据——无论数据来自本地、云端还是边缘计算系统；
    - 完整保留任何类型或体量的数据；
    - 实时或批处理模式下处理数据；
    - 使用 SQL、Python、R 或任何其他语言、第三方数据或分析应用程序进行数据分析。

21 Clustering

0. The main principle behind clustering is that of redundancy.
1. 集群的目的就是提高系统的可靠性和可扩展性
2. 负载均衡的至少 3 种不同策略
    - 轮询策略：依次选择一个节点

    - 选择最少连接的节点
    - hash 策略：根据请求的 ip 地址或者请求的 url 去 hash，然后把请求发到对应的节点上去
3. session stickiness 会话粘滞性问题：比如用户登录之后、会话会一直保持在一个节点上，这样就会造成负载不均衡
    - 可以用 ip-hash 的方法，把同一个 ip 的请求都发到同一个节点上去
4. Request-level failover 和 Session failover
    - 因为服务器会崩、需要重做（容错就是重做）（所以操作需要幂等性）
    - 一种方法是广播
    - 还有一种方法是专门的节点是设置一个 session 服务器（缺点是这个服务器就变成了单一故障节点）（好处是其他节点都变成了无状态的节点）
5. 正向代理：相当于服务器视所有使用代理的客户端请求为同一个客户端的请求
6. 反向代理：相当于一个 load balancer、把请求分发到不同的服务器上去
7. 还可以加 Weighted load balancing
8. **负载均衡**：负载均衡是指将请求分配到集群节点中，以优化整个系统的性能。负载均衡器使用的算法可以是有系统的，也可以是随机的。负载均衡器还可以通过监控集群中各节点的负载情况，选择负载较轻的节点来处理请求。
9. **会话粘性**：对于 Web 负载均衡器来说，一个重要的特性是会话粘性（Session Stickiness）。这意味着客户端的所有请求都会被指向同一台服务器，保证会话的一致性。
10. **故障转移**：为了提供比单个服务器更高的可用性，集群必须具备故障转移能力，当主服务器发生故障时，能够切换到备用服务器继续提供服务。
    - **请求级别故障转移**：当请求指向的节点无法处理请求时，负载均衡器会将请求重定向到其他节点处理。
    - **会话故障转移**：如果会话状态在客户端和服务器之间共享，单纯的请求级别故障转移可能无法保证继续运行。在这种情况下，服务器节点必须重建会话状态。
11. **幂等方法**：幂等方法是指一个方法可以多次调用，并且每次使用相同的参数时都会得到相同的结果。例如，HTTP 的 GET 方法通常是幂等的。
12. **非幂等方法**：一般来说，任何根据当前状态修改持久存储的操作方法都不是幂等的，因为相同的方法调用两次会改变持久存储两次。
13. **请求失败的发生位置**：请求失败可能发生在以下三个阶段：
    - 在请求发起后、服务器方法执行开始之前；
    - 在服务器方法执行开始后、但方法尚未完成之前；
    - 在服务器方法执行完成后，但响应尚未成功传输到客户端之前。
14. **负载均衡方法**：在使用 **轮询（round-robin）** 或 **最少连接（least-connected）** 负载均衡时，客户端的每一个请求都可能被分配到不同的服务器。因此，无法保证每次请求都被定向到同一台服务器。
15. **会话保持需求**：如果需要确保同一个客户端的所有请求始终指向相同的应用服务器，即需要保证客户端的会话是"粘性"或"持久"的，可以使用 **ip-hash** 负载均衡机制。
16. **ip-hash 机制**：通过 **ip-hash** 负载均衡机制，客户端的请求将根据客户端的 IP 地址进行哈希运算，从而始终将该客户端的请求定向到相同的服务器。
17. mysql 的 innoDB 也可以配置集群（mysql shell + router）
    - 使用的时候、是前端连接到 router
    - 就可以做到写请求只写一个、然后读请求会做负载均衡

18. 12 月 9 日这里还讲了一个配置 mysql 集群的操作，不知道在干什么，可以再回看一下
19. Only when the joining instance has recovered all of the transactions previously processed by the cluster can it then join as an online instance and begin processing transactions.
20. You need a minimum of three instances in the cluster to make it tolerant to the failure of one instance.
21. 做 configurationInstance 的时候、几个实例需要完全一样（mysql 版本、配置、数据），而且需要至少 3 个实例
22. 返回结果的时候、是通过 nginx 进行返回的，还是不通过 nginx 进行返回的？（如果是，还能不能那么高效？可以通过 nginx 集群增加数量）
    - 是经过 nginx 统一中转再返回到前端的，客户端访问和看到的是 Nginx 而不是 实际 server(自己看一下 HTTP Header 中的 Server 头就能确认，确实是 nginx)
    - 屏蔽了后端的真实 IP，增加了安全性，也避免了差异性
    - nginx 做了优化
    - Nginx 的非阻塞事件驱动模型：Nginx 采用的是基于事件驱动的非阻塞模型（即单线程异步 I/O），能够高效地处理大量并发请求。这使得 Nginx 即使在高并发环境下也能保持高效的响应速度。
    - Nginx 内存效率高：Nginx 在处理请求时，使用的是少量的内存，并且它的处理模型使得它可以在不占用过多资源的情况下，处理大量的并发连接。

22 Cloud Computing

1. 网格计算：每个人可以把自己的计算资源共享，然后使用计算资源的时候不用在意是谁的计算资源
2. 云计算的特点
    - 弹性伸缩
    - 按需付费
    - 服务化
    - 无限扩展性
    - 快速虚拟化
3. 也会涉及到多租户的情况
    - 最简单的相当于在门户网站上定制自己的 ui，但是本质上是一个服务
    - 然后云服务把所有软件和硬件都提供为服务
4. 会有三种情况：
    - long-term
    - short-term
    - onspot（拍卖）：灵活定价
    - 可以根据需求去选择不同的云服务商（课上的曲线）以降低成本
5. 垂直扩展和水平扩展
    - 垂直扩展就是增加单个节点的资源
    - 水平扩展就是增加节点的数量
    - 云提供商应该都支持这些操作
6. google 提出了 mapreduce

    - 使得云上的操作系统可以进行 job scheduling
    - 首先输入
    - map phase：把原文映射成了很多个键值对（然后每次一个 worker 会处理一个 split）
    - 然后存在了一个 intermediate file 里面（等到所有 split 都被处理完之后、就会进行 reduce）
    - reduce phase：把相同 key 的 value 进行合并（可以会有不同的区间范围规则。比如 A-N 由 worker1 处理，O-Z 由 worker2 处理）
    - 然后输出
    - 这个是一个批式处理的过程
7. 然后需要一个分布式的文件系统 DFS，谷歌提出了 GFS
    - 有一个 master 节点，然后有很多 chunk server
    - master 节点负责管理 chunk server，chunk server 负责存储 chunk
    - master 节点会记录 chunk 的 metadata
    - 有一个 shadow master 节点，用来备份 master 节点的信息
    - 有一个 chunk server 会有很多个 chunk，但是一个 chunk 只会存在于一个 chunk server 上
    - 返回的是 handler
8. 文件系统在多个用户使用的时候、需要考虑并发问题
    - 并发控制
    - 还需要应对错误
    - 分离的服务器越多、可靠性越差
    - 需要使用 Write Control and Data Flow 增加副本，提高可靠性（解决一致性问题）
    - 增加 replica 的数量、可以提高系统的可用性，但是会增加系统的负担（所以写操作的时候性能会下降）（如果不一致、这个写操作就失败）（也是必须要做的）
9. 数据存储的时候需要用到 bigtable
    - 如果用传统的关系型数据库、会有很多的外键关联和检查（而且会需要很多不同机器上的 join 操作和不同进程间通信）
    - 如果横着存、很多时候会实现不了（如果采用切开等价类，采用语义等价的情况）
    - bigtable 就把很多表之间的关系关联都切除、就可以把数据存储在不同的机器上
    - 可以通过再加一列提高它的表现性（比如把几个列合并加成列族，然后再动态加一列新的，允许字段为空，所以对结构化的要更灵活）
10. 所有 table 和 tablet 都可以进行切分
    - 然后数据库的入口就在了 root tablet
    - chubby file 能够维护一个锁，保证并行安全
11. OS：
    - FS：GFS
    - job scheduling：mapreduce
    - DB：Bigtable
    - mem（内存管理）：这个就没有统一架构了、每个系统有自己的实现（可以像 gaussdb 一样看作是一个大的内存整体）
12. hadoop 就是上面这些内容的开源实现
13. 云原生：cloud native

    - microservice -> serverless
    - VM -> container
    - CI/CD（持续开发、持续集成、持续部署，增量式开发）
14. 边缘计算
    - 比如有很多基站，里面其实也是有服务器的
    - 边缘设备和云不能持续性连接
    - 所以需要节省带宽，可以间歇性地把数据发到最近的基站上，然后选择性地把数据发到云上去
15. 边缘计算的实际场景（云边端融合的场景）：
    - 比如说视频分析，会在摄像头里直接使用计算资源进行计算
    - 如果超过了摄像头的处理能力、那么就发到云里去做处理（动态变化的）
    - 或者说智能家居（边缘操作系统实现互联）
    - 或者说如果实现手机的 ai 计算、可以把一些计算放到附近的 MEC（Mobile Edge Computing）上去做
    - 然后逐层地动态决定在哪一层上进行处理
16. 不同的模式名称
    - Local Execution （全部本地处理）
    - Full offloading （全部发送）
    - Partial offloading （部分本地处理，部分发送）
17. 下面讲了 graphql
    - 能不能对客户端更友好，客户端可以根据查询的语句只获取自己需要的数据、而不是整个数据，比如搜索 id 返回 id
    - graphql 在后端直接根据前端的 query 进行组装，然后返回结果给前端（和 restful 不同）
18. 也需要写 resource 里的 schema
19. **GraphQL 是一种查询语言**：GraphQL 是一种用于 API 的查询语言，也是一个用于通过定义的数据类型系统来执行这些查询的运行时环境。
20. **提供完整的数据描述**：GraphQL 为 API 提供了一个完整且易于理解的数据描述，客户端可以准确地请求它们需要的数据，而不是更多，减少了数据的冗余传输。
21. **支持 API 演化**：GraphQL 可以方便地随着时间的推移演化 API，允许添加新字段或类型，而不破坏现有客户端的使用。
22. **与数据库无关**：GraphQL 并不依赖于特定的数据库或存储引擎，它是通过现有的代码和数据来实现的。
23. **GraphQL 服务的创建**：GraphQL 服务是通过定义类型（types）和这些类型的字段（fields）来创建的，并为每个字段提供相应的函数。
24. **GraphQL 服务运行**：GraphQL 服务运行后，通常会有一个 URL，能够接收并验证查询，确保查询只引用已定义的类型和字段，并执行相应的函数来返回结果。
25. **字段（Fields）**：GraphQL 查询允许请求特定的字段，查询结果中可以包含所需字段的数据，而不是获取整个对象。
26. **查询相关对象的字段**：GraphQL 查询不仅能够请求字段，还能遍历与之相关的对象及其字段，这样可以在一次请求中获取多个相关的数据，而不需要像传统的 REST 架构中那样做多次请求。
27. **参数（Arguments）**：在 GraphQL 中，每个字段和嵌套对象都可以有自己的参数，这使得 GraphQL 完全可以替代多个 API 请求。

28. **别名（Aliases）**：如果有多个字段会产生冲突，可以通过别名为这些字段命名不同的名字，从而在一个请求中获取多个不同的结果。
29. **片段（Fragments）**：片段允许你构造字段的集合，然后在需要的地方包含这些集合，从而避免重复写相同的字段选择。
30. **变量（Variables）**：GraphQL 提供了第一类变量的支持，允许将动态值从查询中提取，并作为一个单独的字典传递，这样查询可以更加灵活和通用。
31. **变更（Mutations）**：GraphQL 中的变更类似于查询，但它们会导致数据的修改（写操作）。例如，创建评论的字段 `createReview` 会返回新创建的评论的评分和评论内容字段。
32. **内联片段（Inline Fragments）**：当查询的字段返回一个接口或联合类型时，你需要使用内联片段来访问底层具体类型的数据。

23 docker

1. 一开始讲了一个例子、就是在一个表上如果有一个复合主键（a 和 b）、会自动地先在这个复合主键上面做索引，然后再新建了一个 b 和 c 两个键的索引（默认都是升序的）
    - 查询的时候、如果在最前面加一个 explain、会解释这个查询是怎么执行的
    - 比如说如果做 EXPLAIN SELECT * FROM table WHERE a = 1 AND b = 2 AND c > 1 ，会返回 possible_keys 是这两个都可以，就是 PRIMARY 和 b_c，然后这里会先用 PRIMARY（然后 key_len 是 8，因为是两个 int，a 和 b，所以是 4+4=8）
    - EXPLAIN SELECT * FROM table WHERE a = 1 AND b > 2 AND c < 1 也用 PRIMARY , key_len 是 8
    - EXPLAIN SELECT * FROM table WHERE b = 2 AND c > 1 也用 b_c（先用 b 搜然后搜出来之后再用 c 去筛选, key_len 是 8）
    - 但是有一个情况不一样、就是 EXPLAIN SELECT * FROM table WHERE b > 2 AND c > 1，key_len 是 4，虽然仍然是在用这个索引，但是只用了 b 这个索引，没有用到 c 这个索引因为相当于是在 b=8 和 b=9 上、分别有不同的关于 c 的记录，所以找到对应的大于的 b 之后、需要进一步地在子树 c 里面去做遍历、而且它的 c 的值可能不连续
    - 所以在范围查找的时候、只有在第一个碰到的时候才会用到这个索引、之后就不会用到了，会在这个索引上做遍历 scan （b > 2 的时候、就会用到这个索引，但是 c > 1 的时候就会遍历）
    - 所以设计索引的时候需要考虑到很多是不是会有范围查找的情况，因为碰到第一个范围查找之后、这个索引就不会再被使用了
2. image 就是对整个运行环境进行了打包。docker 是在一个 linux 的 os 环境的上层进行打包运行的
    - 分层的目的是什么、比如我们用 spring boot 开发一个项目、别人用 c#开发、其中有一个共同的层、那么这个共同的层就可以抽取出来做复用。（pull 的时候可以节省空间）
    - 就是不同的容器可能会共享相同的层
    - 用的是 cgroup 和 namespace 将进程隔离开
    - 通过 cgroup 保证不同容器的文件系统是隔离开的
    - 虚拟机是每一个虚拟机都有一个独立的操作系统，而容器是共享一个操作系统
3. dockerfile 主要负责怎么打包这个 image
4. 能不能不要每次修改原始文件的时候都重新打包这个镜像呢?
    - 也可以在环境中动态添加文件
    - 可以用 volume，把本地的文件映射到容器里面去
    - 这个卷的名称好像也可以指定固定的路径+名称、而不用每次都用路径

    - 有差别、一个是容器虚拟机内部管理、还有一个是本地管理（挂载）
5. 下面的每一行代码的意思是

```
$ docker run -d -p 127.0.0.1:3000:3000 \
    -w /app --mount type=bind,src="$(pwd)",target=/app \
    node:18-alpine \
    sh -c "yarn install && yarn run dev"
```

```
- -d 是后台运行
- -p 是端口映射
- -w 是工作目录
- --mount 是挂载,这里挂载了当前目录到 /app
- node:18-alpine 是镜像
```

6. 容器间要能互相通信、需要加入共同的网络
7. docker compose 一次起多个容器
8. **什么是容器**：
    - 容器是运行在主机上的一个沙箱化进程，与主机上其他进程相互隔离。
    - 这种隔离利用了 Linux 内核中的命名空间（namespace）和控制组（cgroups）技术，且这些特性在 Linux 中已经存在很长时间。
    - Docker 使得这些功能变得更加易于使用。
9. **容器的特点**：
    - 容器是一个镜像的可运行实例。你可以使用 Docker 的 API 或命令行界面（CLI）创建、启动、停止、移动或删除容器。
    - 容器可以在本地机器、虚拟机上运行，或者部署到云端。
    - 容器是可移植的（可以在任何操作系统上运行）。
    - 容器与其他容器隔离，运行自己的软件、二进制文件、配置等。
10. **容器和 chroot 的比较**：
    - 如果你熟悉 `chroot`，可以将容器看作是 `chroot` 的扩展版。容器的文件系统来源于镜像，但容器提供了更多的隔离功能，这是 `chroot` 无法做到的。
11. **什么是镜像**：
    - 运行中的容器使用一个隔离的文件系统，这个文件系统是由镜像提供的。镜像包含了运行应用程序所需的所有内容，包括所有依赖项、配置、脚本、二进制文件等。
    - 镜像还包含容器的其他配置，例如环境变量、默认执行的命令和其他元数据。
12. **容器和虚拟机的比较**：
    - 容器和虚拟机在资源隔离和分配上有类似的优点，但工作原理不同。容器虚拟化的是操作系统，而虚拟机虚拟化的是硬件。容器比虚拟机更具可移植性和高效性。
13. **Docker 的实现**：
    - Docker 是用 Go 编程语言编写的，并利用了 Linux 内核中的多个特性来实现其功能。
    - Docker 使用名为 "命名空间"（namespace）的技术来提供容器的隔离工作空间。
14. **命名空间的作用**：
    - 当运行容器时，Docker 会为该容器创建一组命名空间。命名空间为容器提供隔离层。

- 容器的每个方面（如网络、进程、文件系统等）都运行在各自的命名空间内，且访问被限制在各自的命名空间中。

15. **容器的数据持久化问题**：
    - 在之前的实验中，我们看到每次容器启动时都从镜像定义中开始。
    - 容器可以创建、更新和删除文件，但当容器被移除时，这些更改会丢失，且这些更改仅限于容器内部。
    - 使用卷（volumes）可以解决这个问题，使得容器中的数据可以持久化。
16. **卷的作用**：
    - 卷提供了将容器内特定文件系统路径连接到主机机器的能力。
    - 如果容器中的某个目录被挂载到主机，容器内的更改也会在主机上反映出来。
    - 如果将同一个目录挂载到多个容器上，每次容器重启时，都会看到相同的文件。
17. **卷的类型**：
    - 卷有两种主要类型，本文中将先使用命名卷（named volumes）。
18. **示例应用**：
    - 默认情况下，待办事项应用会将数据存储在容器文件系统中的 SQLite 数据库 `/etc/todos/todo.db` 文件中。
    - 如果我们能够将这个数据库文件持久化到主机上，并且确保它在下一个容器中可用，新的容器应该能够继续上一个容器的工作。
19. **使用命名卷**：
    - 命名卷可以理解为一个数据存储桶。Docker 会管理物理存储位置，用户只需要记住卷的名称。
    - 每次使用该卷时，Docker 会确保提供正确的数据。
20. **创建命名卷**：
    - 使用 `docker volume create` 命令创建一个卷。例如，创建名为 `todo-db` 的卷：

      ```
      $ docker volume create todo-db
      ```

21. **启动容器并挂载卷**：
    - 停止并删除当前运行的待办事项应用容器（因为它没有使用持久化卷）。
    - 使用 `docker run` 命令启动容器，并添加 `-v` 标志来指定卷挂载。将命名卷挂载到容器的 `/etc/todos` 目录下，以便保存所有在该路径下创建的文件：

      ```
      $ docker run -dp 127.0.0.1:3000:3000 --mount type=volume,src=todo-db,target=/etc/todos getting-started
      ```

22. **验证数据持久化**：
    - 启动容器后，打开应用并在待办事项列表中添加一些项目。
    - 停止并删除容器，可以通过 Dashboard 或 `docker ps` 获取容器 ID，使用 `docker rm -f <id>` 删除容器。
    - 使用相同的命令再次启动一个新的容器。
    - 打开应用，应该会看到你之前添加的待办事项仍然在列表中。
23. **很多人常常问：“Docker 实际上在使用命名卷时将数据存储在哪里？”**

- 如果你想知道，可以使用 `docker volume inspect` 命令来查看详细信息。

24. **Mountpoint 是数据存储的实际位置**
    - `Mountpoint` 是 Docker 存储数据的实际路径。在大多数机器上，你可能需要具有 root 权限才能从主机访问这个目录。但这就是数据存储的位置！
25. **绑定挂载（Bind Mounts）**
    - 对于绑定挂载（bind mounts），我们控制主机上挂载的具体位置。绑定挂载通常用于将数据从主机传入容器，也可以用来将源代码挂载到容器中，以便容器能够看到代码更改并即时响应。
26. **卷的类型**
    - Docker 引擎支持两种主要的卷类型：绑定挂载（bind mounts）和命名卷（named volumes）。
    - 绑定挂载和命名卷是最常见的类型，但 Docker 还支持其他一些额外的卷驱动（如 SFTP、Ceph、NetApp、S3 等），以支持不同的使用场景。
27. **Host location（主机位置）**：
    - **命名卷（Named Volumes）**：主机的位置由 Docker 自动选择。你不需要指定存储路径，Docker 会管理这个位置。
    - **绑定挂载（Bind Mounts）**：你自己决定主机上的路径。你需要明确指定挂载点路径。
28. **Mount example（挂载示例）**：
    - **命名卷（Named Volumes）**：

      ```
      --mount type=volume,src=my-volume,target=/usr/local/data
      ```

      这里，`src=my-volume` 表示使用名为 `my-volume` 的卷，`target=/usr/local/data` 是容器内部的挂载点。
    - **绑定挂载（Bind Mounts）**：

      ```
      --mount type=bind,src=/path/to/data,target=/usr/local/data
      ```

      这里，`src=/path/to/data` 是主机上的路径，`target=/usr/local/data` 是容器内部的挂载点。
29. **Populates new volume with container contents（用容器内容填充新卷）**：
    - **命名卷（Named Volumes）**：当创建一个新的卷时，Docker 会自动将容器的内容填充到新的卷中。
    - **绑定挂载（Bind Mounts）**：不会自动用容器内容填充绑定挂载。绑定挂载依赖于你指定的主机路径，容器不会修改这个路径的内容。
30. **Supports Volume Drivers（支持卷驱动）**：
    - **命名卷（Named Volumes）**：支持卷驱动（Volume Drivers），允许你使用不同类型的存储后端，如 Ceph、S3 等。
    - **绑定挂载（Bind Mounts）**：不支持卷驱动。绑定挂载仅仅是将主机上的指定路径挂载到容器内，并不涉及不同的存储后端。
    - **命名卷**由 Docker 管理，适用于容器间共享数据并支持使用不同存储后端的情况。
    - **绑定挂载**则直接映射主机的指定路径，适用于需要直接访问或修改主机文件的场景。
31. bind-mount 可以实时 develop 更新

    - Using bind mounts is very common for local development setups.
    - The advantage is that the dev machine doesn't need to have all of the build tools and environments installed.
    - With a single docker run command, the dev environment is pulled and ready to go.
32. **Docker Compose 是一个帮助定义和共享多容器应用的工具**:
    Docker Compose 使得我们可以通过定义 YAML 文件来描述多个服务，并通过单个命令来启动或停止这些服务。
33. **使用 Compose 的一个重要优势是**:
    我们可以在一个文件中定义应用栈，并将该文件保存在项目仓库的根目录下（现在它是版本控制的），这使得其他人也能轻松参与到项目中。
34. **只需要克隆你的仓库并启动 Compose 应用**:
    这样，其他开发人员只需要克隆你的项目仓库并启动 Compose 应用，就能轻松复现你的开发环境和应用设置。
35. **GitHub/GitLab 上的许多项目都在这样做**:
    现在，你可能会发现 GitHub 或 GitLab 上有很多项目正是通过 Docker Compose 来实现这种便捷的开发和部署方式。
36. **一般情况下，每个容器应该做一件事并做到最好**:
    这样设计的几个原因包括：
    - 你可能需要根据不同的需求对 API 和前端与数据库进行不同的扩展。
    - 将容器分开能够让你在隔离的环境中进行版本管理和更新。
    - 在本地开发时，你可能会使用容器来运行数据库，但在生产环境中你可能更倾向于使用托管的数据库服务。在这种情况下，你不希望将数据库引擎与应用一同部署。
    - 如果你在容器中运行多个进程，需要使用进程管理器（因为容器默认只启动一个进程），这会增加容器启动和关闭的复杂性。
37. **容器网络**:
    - 默认情况下，容器是相互隔离的，它们不知道同一台机器上的其他进程或容器。那么，如何让一个容器与另一个容器通信呢？
    - 答案是：**网络。**
38. **注意**:
    - 如果两个容器在同一个网络中，它们可以相互通信。
    - 如果不在同一个网络中，它们无法通信。
39. **启动 MySQL**:
    - 有两种方法可以将容器连接到网络：
        1. 在启动容器时指定网络。
        2. 将已运行的容器连接到网络。
    - 在本节中，我们将先创建网络，然后在启动时将 MySQL 容器连接到该网络。

24 Hadoop

1. 一个操作系统需要
    - 文件系统 GFS
    - job scheduling mapreduce
    - memory management 各个不一样

    - database bigtable
    - hadoop 包含了上面的所有
2. 在配置文件里面、一定要有下面的内容，指定了 hdfs 的地址和端口

```
etc/hadoop/core-site.xml:
     <configuration>
         <property>
             <name>fs.defaultFS</name>
             <value>hdfs://localhost:9000</value>
         </property>
      </configuration>
```

3. 下面配置副本数

```
etc/hadoop/hdfs-site.xml:
     <configuration>
         <property>
             <name>dfs.replication</name>
             <value>1</value>
         </property>
     </configuration>
```

3. hdfs 是基于 linux fs 上面运行的、而不是纯基于底层硬件的
4. 文件分区表存在下面的 namenode 里面

```
Format the filesystem:
$ bin/hdfs namenode -format
```

5. namenode 是客户端每次访问的必经之路
    - 存储了每个 file 的所有 block 底层上存在了 linuxFS 中的哪里
6. 可以在 localhost:9870 查看 namenode 的状态
7. 实际的运行流程就是先启动 hdfs 集群、然后在上面指定地运行 mapreduce 操作、比如丢一个 jar 包，然后等待结果
8. 和云计算那一节和 cse 中提到的差不多
9. 例子里的时候、是会对两个 string 1 合并成 string [1,1]，然后统计总数
10. 第一次做 reduce、会在 mapper worker 结束之后的本地去做（这次操作叫做 combine，把本地的次数合并），第二次做 reduce 是在 reduce worker 里面去做（是对多个本地合并后的结果再去做 reduce）
11. 然后这个 intermediate file、图里有两列、那么第一列永远是 reducer 1 的 partition、第二列是 reducer 2 的 partition
12. 需要进行 shuffle（就是 sort 排序），然后再进行合并（为了减少随机 io）
13. combine 和 reduce 可以用不同的函数、也可以用相同的函数
14. 还有一个例子是找不同年份中温度的最大值
15. combiner 也可以让在网络上传递的数据变少
16. 有些时候也不需要 combiner、比如传输的数据不多的时候
17. 但是有时候不能用 combiner、比如求平均值的时候、除非每个本地的数目都相同

18. 所以要具体问题具体分析
19. 导入这个库后、貌似这个进程会跑一个 hadoop 集群
20. 一种常见的情况是、一个 outputfile 包含了 A-N 的、还有一个包含了 O-Z 的、那么就需要两个 reduce worker 去处理，最后存到不同的 replica 里面
21. 也可以没有 reduce、做完 map 之后就存
22. 首先是 job client 根据程序生成一个 job、然后给 jobtracker
23. jobtracker 会做资源的管理、他会记录每个机器的状态和任务、然后记录心跳包、然后如果发现某个节点挂了、那么就会重新分配任务到节点
24. jobtracker 会是一个瓶颈
25. 给定一个时间段里、会需要接收到固定数量的心跳、不然就会认为这个节点挂了（因为 job 的时长可能会不同）
26. reducer 的结果可能不是有序的
27. 这个排序的比较器也可以自己指定
28. 然后讲了一个问题、就是因为每个 split 可能也不是足够小的、所以每个 mapper 在做 map 的过程中、如果内存满了、会把内容以 spill 的形式写到硬盘上（overflow 了）
    - mapReduce 之中会使用大量的磁盘 IO、在所有的步骤中都会涉及到（都是在磁盘中实现的）（每个 part 的步骤都有）
    - 而且会涉及到可靠性的问题、如果机器崩了、那么前面的任务就白做了
    - 可以做一个优化、比如直接把 spill 放到待 reduce 的队列中
29. 由于 job tracker 是把资源管理和 job scheduling 都放在一起、所以可能会有瓶颈
30. yarn 就是 mapreduce 的升级版，每个应用都有一个 applicationmaster（每一个 job 都有一个 tracker 相当于），然后有一个全局的 resource manager
31. 相当于管理一个 job 的所有容器的状态都丢给了 application master（每一个任务一个），然后请求新资源的时候会通过向给 resource manager 去请求，这就实现了解耦、职责分离了，然后一个 job 崩了、另外的 job 还可以正常进行
32. mappers
    - Mapper maps input key/value pairs to a set of intermediate key/value pairs.
    - The Hadoop MapReduce framework spawns one map task for each InputSplit generated by the InputFormat for the job.
    - Output pairs do not need to be of the same types as input pairs.
    - All intermediate values associated with a given output key are subsequently grouped by the framework,
    - The Mapper outputs are sorted and then partitioned per Reducer.
    - 用户也可以自己设置 combiner
33. mapper 的数量可以设置和输入的文件 block 一样就可以了
34. 这里的 number of reduces 如果希望容器间切换的开销小一点、那么就设的小一点，如果处理得快，那么就可以调高一点（1.75），然后就可以利用磁盘 IO 的空闲时间，能充分利用资源
35. 这里的 shuffle 和 sort 其实是一体的，保证了数据的有序性，也可以通过 secondary sort 自定义排序方法
    - The shuffle and sort phases occur simultaneously; while map-outputs are being fetched they are merged.

36. Reducer NONE
    It is legal to set the number of reduce-tasks to zero if no reduction is desired.
37. Partitioner
    Partitioner partitions the key space.
    Partitioner controls the partitioning of the keys of the intermediate map-outputs.
    The key (or a subset of the key) is used to derive the partition, typically by a hash function.
38. reducer 包含的结果是非排好序的
39. **Map 任务数量与输入数据大小的关系**：
    - Map 任务的数量通常由输入数据的总大小决定，具体来说，是由输入文件的块数量决定。
40. **Map 任务的并行度设置**：
    - 合适的并行度通常在每个节点 10 到 100 个 map 任务之间，尤其是当 map 任务比较轻时，可以设置为每节点 300 个 map 任务。
    - 设置的 map 任务应该确保每个任务至少执行一分钟，这样可以避免任务设置的开销过大。
41. **具体例子**：
    - 例如，如果你有 10TB 的输入数据，块大小为 128MB，那么最终会得到约 82,000 个 map 任务，除非你使用 `Configuration.set(MRJobConfig.NUM_MAPS, int)` 进行调整，这个配置仅提供一个提示，框架会根据该提示来设置更高的任务数。
42. **Reduce 任务数量的设定**：
    - 合适的 reduce 任务数量是 0.95 或 1.75 倍于（节点数 × 每个节点最大容器数）。
    - 如果设置为 0.95，所有 reduce 任务可以立即启动，并开始在 map 完成时传输 map 输出数据。
    - 如果设置为 1.75，较快的节点会在完成第一次 reduce 后启动第二轮 reduce，从而提高负载均衡效果。
43. **增加 reduce 任务数量的影响**：
    - 增加 reduce 任务的数量会增加框架的开销，但可以提高负载均衡性，减少失败任务的成本。
    - 上述 scaling factor 设置略小于整数值，是为了给框架保留一些 reduce 槽位，用于处理推测性任务和失败任务。
44. **MapReduce 2.0（MRv2）或 YARN**：
    - 在 Hadoop 0.23 中，MapReduce 进行了完全的改造，推出了新版本的 MapReduce，称为 MapReduce 2.0 或 YARN。
45. **MRv2 的基本思想**：
    - MRv2 的核心思想是将 JobTracker 的两个主要功能：资源管理和作业调度/监控，分成独立的守护进程。
46. **架构概念**：
    - 引入了 **ResourceManager（RM）** 和每个应用程序的 **ApplicationMaster（AM）**。
    - 一个应用程序可以是一个单一的作业或一组作业的有向无环图（DAG）。
47. **ResourceManager（RM） 的组成**：
    - ResourceManager 有两个主要组件：
        - **Scheduler（调度器）**：负责根据资源需求为正在运行的应用程序分配资源，受容量、队列等约束。

        - **ApplicationsManager（应用程序管理器）**：负责接收作业提交，协商执行应用程序特定的 ApplicationMaster 容器，并提供在失败时重启 ApplicationMaster 容器的服务。
48. **Scheduler 的工作方式**：
    - Scheduler 根据应用程序的资源需求进行调度，基于抽象的资源容器（Container）进行资源分配。容器包括内存、CPU、磁盘、网络等资源。
    - 在最初版本中，Scheduler 只支持内存。
    - Scheduler 使用可插拔的策略插件，负责将集群资源划分给各个队列、应用程序等。比如，当前的 MapReduce 调度器，如 CapacityScheduler 和 FairScheduler，就是插件的示例。
49. **NodeManager 的职责**：
    - NodeManager 是每台机器的框架代理，负责管理容器，监控它们的资源使用（CPU、内存、磁盘、网络），并将这些信息报告给 ResourceManager 和 Scheduler。
50. **ApplicationMaster 的职责**：
    - 每个应用程序的 ApplicationMaster 负责：
        - 从 Scheduler 协商合适的资源容器。
        - 跟踪容器的状态并监控作业的进度。
51. 这样就没有了性能瓶颈和单一故障节点
52. 但是都是对用户无感知的
53. **Apache™ Hadoop® 项目**：
    - 该项目开发了开源软件，用于可靠、可扩展的分布式计算。
54. **Apache Hadoop 软件库**：
    - Hadoop 是一个框架，允许在计算机集群上以简单的编程模型分布式处理大规模数据集。
    - 它设计成可以从单台服务器扩展到成千上万的机器，每台机器提供本地计算和存储能力。
    - Hadoop 不依赖硬件提供高可用性，而是设计了能够在应用层检测和处理故障的机制，因此可以在每台可能会发生故障的计算机集群上提供高可用的服务。

25 Spark

1. 也是对大规模的数据做分析、但是是放在内存里做的（和 hadoop 最大的差异）
2. hadoop 会涉及到很多 IO、所以性能比较差
3. 例子的目的是在一个很大的 log 中、过滤出来一个小部分的报错的信息
4. 这个 lines 就是 RDD 的一个实例、一旦读到了内存里之后这个值就不能被修改了、这也意味着这个值可以被复用（多线程的竞争就可以免除掉）
5. filter 可以使得一个 RDD lines 变成另一个 RDD errors
6. 同样地，map 也是一样的、可以把一个 RDD 变成另一个 RDD
7. 为什么要 cache 住呢，因为内存可能会满，所以需要涉及 LRU，除了 LRU、还有 swap、就是把内存里的内容写到硬盘上。但是 cache 住了之后、这个值就不会被释放掉了（无论做 LRU 还是 swap 都不会被释放，一直在内存里，但是这只是一个愿望，如果全都满了，还是会进行替换）
8. 然后其实本质上等到 cache 这行的时候、才会反推执行前面这三步一起执行、内存里会先将它们进入计算图再执行。（通过 stage 执行，惰性）（所以抛错误是在 cache 这一行才报的错）
9. 同样地直到执行.count 的时候才会执行前面的.filter。这些操作叫做 action（比如.count，.cache），才会反推进行前面的计算（前面的叫做 transformation）。然后这个 filter 和 count 都是在多个节点上进行的、然后再合并成一个的。

10. spark 也是分为 worker 和 manager 的
11. 先起一个 spark 集群、然后使用 spark-submit 提交一个脚本、然后就能执行任务了。
12. 这里需要用 spark submit exe 去运行一个 jar 包才能运行对应的 java 代码
13. scala 也可以变成一个 jar 包去运行
14. RDD 是一个弹性分布式数据集（Resilient Distributed Dataset），可以并行地去操作。也可以把内存里不是 RDD 的内容通过 sparkcontent 变成 RDD
15. RDD 一般 load 之后都是分 partition 的
16. 所有 transformation 操作都是 lazy 的。比如上面的这些 filter 操作
17. 中间可以看一下回放
18. 这个 An Easy Example 的例子、当没有 partition 的时候。如果要生成一个结果、那么这个 join 就需要读所有的 userDate 和所有的 events
19. 如果已经用 hash 的方式把 userData 做了分区、那么久可以每个 join 对应一个 userData 的块去做操作
20. 窄依赖就意味着性能会比较高（只需要读一部分、而且会在同一台机器上，而且如果崩了、可以只重新拿一部分）、宽依赖就意味着性能会比较差
21. Narrow Dependencies VS. Wide Dependencies
    - 宽依赖往往意味着 Shuffle 操作，可能涉及多个节点的数据传输
    - 当 RDD 分区丢失时，Spark 会对数据进行重算
    - 窄依赖只需计算丢失 RDD 的父分区，不同节点间可以并行计算，能更有效地进行节点的恢复
    - 宽依赖中，重算的子 RDD 分区往往来源自多个父 RDD 分区，其中只有一部分数据用于恢复，造成了不必要的冗余，甚至需要整体重新计算
22. 下面这个例子是只有当发生宽依赖的时候、才进行计算、然后再进行前面还没有进行的计算
    - 好处是可以节省很多中间结果的 RDD、可以节省内存，而且可以并行化
    - 而且每个 stage 内部的计算会很快、因为可能这个数据和计算发生在同一台机器上（lazy）
    - 就是一个 stage 结束之后、才回推做 stage 前面还没做的很多操作
    - 每个阶段 stage 内部尽可能多地包含一组具有窄依赖关系的 transformations 操作，以便将它们流水线并行化（pipeline）
    - 而且可以在同一台机器里算、算得也比较快
    - 边界有两种情况：一是宽依赖上的 Shuffle 操作；二是已缓存分区
    - 黑色代表一个 cache 的动作、所以要划分处理
    - 这个 E 为什么又 cache 的内容、还是不是一个单独的分区呢、因为是拿来前面用的 cache
23. **RDD 支持两种操作类型**：
    - RDD 支持两种主要的操作：**转换操作**（transformations）和**动作操作**（actions）。
24. **转换操作（Transformations）**：
    - 转换操作是从现有数据集创建一个新的数据集。
    - 例如，`map` 是一种转换操作，它将数据集中的每个元素通过一个函数进行处理，然后返回一个新的 RDD，其中包含处理结果。
25. **动作操作（Actions）**：
    - 动作操作是在数据集上运行计算后，返回一个值给驱动程序（driver program）。

    - 例如，`reduce` 是一种动作操作，它通过某个函数聚合 RDD 中的所有元素，最终返回一个结果给驱动程序。
26. **Spark 中的 Shuffle 事件**：
    - 在 Spark 中，某些操作会触发一个被称为 **shuffle** 的事件。
27. **Shuffle 的作用**：
    - Shuffle 是 Spark 用来重新分配数据的机制，使得数据在不同的分区之间重新分组。
    - 通常，这个过程涉及将数据从一个执行节点（executor）复制到另一个执行节点，甚至跨机器进行数据传输。
    - Shuffle 操作通常在进行类似 groupBy、reduceByKey、join 等需要将数据按某种规则重新分组或聚合的操作时发生。
28. **Shuffle 操作的复杂性和成本**：
29. Narrow Dependencies
    - 窄依赖
    - 父 RDD 的每个分区只被子 RDD 的一个分区所使用
    - map, union, ...
30. Wide Dependencies
    - 宽依赖
    - 父 RDD 的每个分区都可能被多个子 RDD 分区所使用
    - Shuffle
    - Join with inputs not co-partitioned, ...
31. Removing Data
    Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion.
    If you would like to manually remove an RDD instead of waiting for it to fall out of the cache, use the RDD.unpersist() method.
32. cache 也有不同的等级、比如 MEMORY_ONLY、MEMORY_AND_DISK、MEMORY_AND_DISK_SER、DISK_ONLY
33. dataframe 是有列名的数据、结构化更好，可以用语句去查询，也可以用 sql 语句去查询
34. 微观上是批处理、宏观上是流处理。这个例子是处理增量进入的数据、进行数据的统计。（就是 spark streaming）（可以通过 kafka 里读出来）
35. 也可以做机器学习和图计算（把顶点和边分别放在两个 RDD 里）（不是最优选择）
36. spark 可以做流批一体的计算

26 Storm

1. 这个 Storm 就是专门基于流数据的处理做的框架
2. 处理的中间结果叫做 Bolt（后面的都是 bolt）
3. 产生数据的源是 Spout
4. 一样是在微观上是批处理、宏观上是流处理
5. 同样也需要集群管理、默认用的是 zookeeper
6. 一个 worker 可以有多个 slots、就是多个进程用于运行多个不同的任务
7. supervisor 就是监控它的状态

8. 在 storm 上会运行 topology，但是这个任务是不会终止的（和 mapreduce 不同）
9. nimbus 管理机器节点的重启和工作等
10. 执行的时候和 spark 类似、需要通过 storm 命令运行一个 jar 包
11. 这里的例子里到底是实例数还是其他的?
12. 后面的 zookeeper 都没怎么再讲

27 hdfs

1. 适合 Very large files “Very large” in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. 因为存的 block 大，目标就是要存很大的文件
2. Hadoop doesn’t require expensive, highly reliable hardware to run on.所以不适合并行写（因为是存在了廉价的硬件上面、所以写同步的代价比较大）
3. write-once, read-many-times pattern， the most efficient data processing pattern 最好只写入一次、不太适合做修改、因为定位会比较难，而且存储会比较麻烦
4. HDFS is not a good fit when:
    - Low-latency data access. Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency.（读写的性能比较低、因为它多了一层操作）
    - Lots of small files .（小文件不如直接用自己的文件系统） Since the namenode holds file system metadata in memory, the limit to the number of files in a file system is governed by the amount of memory on the namenode. （节省 metadata 内存空间）
    - Multiple writers, arbitrary file modifications （如果需要做修改的话、就不适合用 hdfs，因为它是做追加）
5. 所以比如说存很多视频的时候、就适合用 HDFS，因为视频文件很大，而且不会经常修改
6. metadata 存在 namenode 里面
7. datanode 存储实际的数据
8. 所有的数据都不经过 namenode、只有 metadata 经过 namenode（减少瓶颈）
9. 副本数为 3 的话、可能在一个机架上放两个副本、一个在另一个机架上、可以做负载均衡，如果规模更大的话、放到另外一个中心机房里
10. 写操作是流水线式地写入每个 replica，写完每个之后才完成（流水作业的）
11. 集群起：先起 namenode、然后 datanode、datanode 会给 namenode 发心跳包，然后就能加入集群、可扩展性就会变强、不需要重启
12. 通过 datanode 发 blocks report 告诉 namenode 自己有哪些 block，然后要保证每个 block 都有 3 个 replica，然后每台机器上的 block 数量都要均衡（？）A Blockreport contains a list of all blocks on a DataNode.（因为 The DataNode has no knowledge about HDFS files.）
13. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes.
14. namenode 专门用一台机器跑（相当于文件分区表），和 datanode 分离、因为每次都是从 namenode 读取 metadata，所以要保证 namenode 的高可用性，要干净
15. 一个机器上跑多个 datanode 没有意义（因为两个 datanode 可能会在同一台机器上的时候、做冗余就没有意义了）（物理上就应该做到分开，和没有做冗余是一样的）
16. 可以创建目录、不支持硬/软链接

17. have strictly one writer at any time. （不能并行地写多个 replica，否则不能保证数据一致性）（流式地一个个写，为了保证数据一致性）
18. 副本数是每个文件单独设置的、设置多、可以保证可靠性变高、但是也让这个写入的代价变高
19. 会有一个 rack-aware replica placement policy 机架感知策略，但是会减少性能（具体是根据网络延迟的距离来判别的，距离越大、写入的代价越大）（示意图）
20. (replicas - 1) / racks + 2 公式保证了每个 rack 上的 block 数量上限都是均衡的（一般副本是 3）
21. 多个副本的时候、读操作可以做负载均衡（primary 写、secondary 读）
22. On startup, the NameNode enters a special state called Safemode. （在等 datanode 给它发心跳包，因为没有 datanode 的时候，它是不能工作的，所以处于安全模式）
    - 如果数量太少、就会进入安全模式，等待 block 数量平衡
    - 可扩展性就强、运行时就能动态地加入 datanode
23. EditLog 记录了所有操作，写完日志才会做文件操作
24. 这个日志文件不是 hdfs 的一部分、而是本地文件系统的一部分
25. 同样地、fsimage 也是本地文件系统的一部分，因为是启动 hdfs 必须的文件
26. 启动的时候也需要做对齐，有些 editlog 里的操作还没写入 fsimage 里面。
27. 这个会有一个 checkpoint、可以检查 fsimage 和 editlog 的一致性。也可以设置 checkpoint 的时间间隔，但是不能太大也不能太小，太小会影响性能，太大会影响空间开销
    - 和 binlog 是差不多的动作
28. 如果一个收不到一个 datanode 的心跳包，那么就不会再往这个 datanode 上做 io 了、而且会给其他 datanode 发需要产生其他 replica 的请求
29. 如果是网络问题、也是不可用的
30. 如果存的数据太多、就会需要做数据迁移 Cluster Rebalancing
31. 每台机器数据量不一样的话、会做自动的平衡的迁移
32. 会对所有的生成的文件生成一个校验码 checksum，保证 Data Integrity（不被篡改）
33. FsImage and the EditLog 也有多个副本、保证能够启动 hdfs，而且这两个文件的副本必须要实时同步
34. 数据块大小是 128MB，写一次多次读
35. 如果没有 ack 的话、就是会相当于没有写成功。但是还是有一定优化、比如如果就差 1MB 没写的话、还是会有快照之类的机制让它不要重试的时候重复再写一次

28 HBase

1. 在云计算那讲中的 Bigtable 的 hadoop 中的实现
    - 可以随机读写、但是建议是顺序读写
    - Apache HBase is an open-source, distributed, versioned, non-relational database
2. 本身是一个大表、可以切开用 hdfs 存在不同的机器上（分布式）
3. 通过 root 找到要找的表的 metadata、然后根据 metadata 找到对应的 tablet
4. 像一个立方体的数据?
5. 它是按列存的
6. mysql 不做垂直分区、是因为 mysql 默认按行存、而且每个切开的行都要带主键，很麻烦
7. hbase 是按列存、所以可以做垂直分区，而且还支持列族的概念。

8. 典型的应用是 webtable、可以记住在不同时间点上的某个字段的不同的数据
9. Features
    - Linear and modular scalability. （可扩展性好）
    - Strictly consistent reads and writes. （数据一致性 同步执行）
    - Automatic and configurable sharding of tables （tablets 可以自动做 shard）
    - Automatic failover support between RegionServers.
    - Convenient base classes for backing Hadoop MapReduce jobs with Apache HBase tables.
    - Easy to use Java API for client access.
    - Block cache and Bloom Filters for real-time queries.
    - Query predicate push down via server side Filters （查询谓词 query 下推）
    - Thrift gateway and a REST-ful Web service that supports XML, Protobuf, and binary data encoding options
    - Extensible jruby-based (JIRB) shell
    - Support for exporting metrics via the Hadoop metrics subsystem to files or Ganglia; or via JMX
10. Tables are made of rows and columns.
    - Table cells—the intersection of row and column coordinates—are versioned. （所有 cell 都是是带版本的）
    - 和关系型数据库的差异，Hbase 里面的主键是完全有序地排列的（每次插入的时候），因为要记住每个 tablet 里面的键的最小和最大值（和 mysql 的多副本时不一样）（因为通过 key range 来指定一个 tablet）
    - Table row keys are also byte arrays
11. 一个行是由很多个列族组成的、每个列族里面有很多列
12. 列族的意义是：可以动态地加入一个表里（可以隔一段时间再加入）（就是垂直分区的一个依据）（列族的数量不适合超过 3 个，因为会减少性能）（就是一个半结构化的数据）
    - 就可以切开来存到不同的地方去
13. 要通过 namesever 改 metadata，需要先通过 chubby lock 获取锁，就是不是所有人都能改那个 metadata 的表
14. Tables are automatically partitioned horizontally by HBase into regions.
    - 横着切就是 tablet 或 region
15. 这个图就是和前面的差不多、从 master 进去、然后 regionserver 管理这个文件在哪、然后存到 hdfs 里面
16. 没有库的概念、用 namespace 这个概念去替代
17. A common row key pattern is a website domain.这个网址可以倒过来存储
18. 不同的列族里可以有相同的列名、因为是通过两个的组合来定位的
19. 这个会一直往时间戳之前读、直到读到一个不为空的值（所以 hbase 是稀疏的表）
    - 空的就是代表这些个时间里没有发生变化
    - 在每个时间戳里只存了发生变化的值
20. 因为是要保证它的稀疏性、所以用了一个类似 json 的结构去存（时间戳是倒序的、保证直接读就是最新值）

21. 如果在一个先前的时间戳去读的话、有时候会读到一个空值
22. 版本在读取的时候可以指定（时间戳是倒序（降序）的、保证直接读就是最新值）
23. A namespace is a logical grouping of tables analogous to a database in relation database systems.
24. 要先 disable table 然后再能做增加列族的操作（schema 的修改）
25. 所以是三维立方体、有不同版本的维度，更灵活、没有要求每个表有严格一直的 schema
26. 关注的是大量的数据如何存储、而不是关注关系

29 Hive

1. hive 是在 hadoop 基础上发展起来的数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供完整的 sql 查询功能，可以将 sql 语句转换为 MapReduce 任务进行运行。
2. 要在低版本的 java8 上运行
3. 很多数据仓库仍然用的是 hive 去做 metadata 的 store
4. 数据仓库和数据库最大的区别就是它是 schema on read 的状态。在 hive 里，大量的数据仍然是以原始文件的状态存储在 hdfs 上面、然后 hive 在上面去读取这个 hdfs 里的内容。
5. 然后 sql 不可能直接执行在 csv 这种文件格式上、所以，hive 会先将这些文件做处理、然后转成一种中间格式、最常见的是 parquet 格式。
6. csv 加载进来的 dataframe 之后、有些内容字段会为空、所以会在进行 LOAD DATA 操作的时候、不会先做预处理（不做 ETL 操作、不做校验、数据清洗转换和加载）、直到 SQL（做在这个数据上面的分析时）的时候才会执行变成 parquet 的操作。这个是 schema on read 的特性。（好处就是因为数据会非常大量、源源不断地进来、没有足够的计算资源去做 ETL 操作）
7. 初始化 hive 的时候、可以填进去一个，默认是 derby、存的是 metadata 的信息。如果是 derby 的话、就是存储在内存里面、也可以存储在 mysql 里面，但是要告诉具体的位置在哪。
8. hive 默认是分区的、可以按照 range 或者 key 去分区
9. 也可以做分区、之后做搜索的时候就可以定位到一个分区里了
10. parquet 为什么是列存的？因为做的是 OLAP、就是在线的分析处理、都是会对同一列的数据进行操作（比如统计订单数量）、所以列存储的效率会更高。parquet 支持用 sql 操作
11. 同样是 sql、为什么要存到 hive 里而不是 mysql 的？因为 hive 支持很多数据格式。而且 hive 执行 sql 表格的来源是多样的、不仅仅是 mysql table，还可以是 csv 等等。
12. Metastore 是 hive 的元数据存储的地方、hive 的元数据存储在 metastore 里面、metastore 是一个数据库、存储了 hive 的元数据信息、比如表的信息、表的结构、表的列、表的分区等等。可以分成独立的进程、也可以和 hive 在一起。（可以两个 hive server 共用一个 metadata server）
13. 是 schema on read 的状态、所以在 hive 里面、不会对数据做任何的校验和处理、只有在 query 查询的时候才会做处理。（好处就是一开始只需要直接拷过来、然后就很快）（而且数据仓库的场景是大量的数据、不可能一开始就做处理，多种数据源情况下的一个选择）
14. schema on write 的状态、就是在写入的时候就会做处理、比如 mysql、写入的时候就会做校验、数据清洗、转换等等。（查询的数据块，因为查询的时候就不用校验了）
15. （会把 tables 或者 partitions 进一步地划分成小的 buckets）用 buckets、查询方便，进一步减少查询的数据量。而且会做分布存储、不同的 bucket 会存储在不同的地方、这样查询的时候就会更快。也可以保证机器学习中采样的效率、保证整体的采样和部分的采样的一致性。（如果是按照性别区分的 buckets）

16. 存储的时候、会把表拆成不同的列族，然后按列单独去存，这个是 HBase。然后 RCFile 是会先把几个行组织成 row group、然后在这个 row group 里面再按照列进行存储。就有助于 OLAP 的查询。
17. ORCFile 就是优化过的 RCFile。会有 Index data、row data、stripe footer。这样就可以快速定位到数据的位置、然后直接读取数据。（index data 存的就是 row data 中的 block（就是 column 数据） 的 offset）
18. 重复数据多的时候、同样会有优化、比如只存变化量
19. 压缩率高、存储的时候会压缩、查询的时候会解压缩，所以会需要做权衡

30 Flink

1. 事实上的流式数据处理的框架。
2. 在 spark 里、在微观上会把时间轴切成很小的时间窗、然后宏观上是流处理、微观上是批处理。
3. flink 要支持有状态的处理、就是处理后面事件的时候、要知道前面的状态是什么。比如统计一个小时内的数据、就要知道前面的数据是什么。
4. 它可以支持有界或者无界的流式数据
5. 几个概念：streams
    - Bounded and unbounded streams
    - Real-time and recorded streams
6. state
    - 事件和事件之间是有关联的，除非处理单个事件、否则都是有状态的
7. time
    - 所有的事件都有一个时间戳。
    - 有一个产生事件的时间、有一个处理的时间。
    - event-time and processing-time.
8. 有 layered api
    - 如果关心业务、可以用高层的 table api
    - 如果需要从流的角度去处理、可以用中间的 datastream api
    - 如果需要更底层的操作、可以用底层的 process function（流是由一堆事件组成的、可以对这些事件进行操作）
9. 先演示了一个最底层的 api、process function，这个例子是说要计算一个事件的开始时间和结束时间之间的时间差，需要在 start 的时候开始记录并等待、然后在 end 的时候计算时间差。如果等待超过了 4h、就会停止等待这个事件。
10. 这个 datastream api 的例子、就是分一段固定的时间时、做 map reduce
11. 状态是怎么保存的、是放在内存或硬盘上、处理的时候会要读这个状态、内存不大的时候、需要存到硬盘上、会影响性能。每个程序只能访问自己的本地的 state、不能访问其他设备上的 state。为了保证一致性（而且这个可能会丢）、需要做 state 的 snapshot 快照，那么核心问题就是要给哪些东西做快照。
12. 这个 counter 是要有状态的
13. flink 在做 word count 的时候、会有两个流程、就是先 input、然后再做 sink（count）。为了保证不发乱，需要把同样的单词发到同样的地方去。所以就根据 hash 发到不同的机器上去。

14. 这个就是 keyed state、就是根据 key 来存储 state。这个 key 是根据 hash 来生成的、所以相同的 key 会发到同一个地方去。这个 counter 必须是有状态的、这个 counter 是 1 还是 2，是有状态的、基于之前的状态来做的。source 发送的时候就需要做指定目标的发送。
15. 为了提高可靠性、需要每隔一段时间做一个 checkpoint，只会丢掉最后一个 checkpoint 之后的数据。记下来之后、已经做过 checkpoint 的事件、如果还没来得及处理、可以通过 replay checkpoint 来处理。
16. 这个 checkpoint barrier 是什么呢、也是相当于流里的一个事件。比如一个 operator 会接受有两个流、每个流都会接受到一个 checkpoint barrier，然后这个 operator 会等待两个流都接受到 checkpoint barrier 之后、才会继续往下走。（这里就在做对齐操作了，然后写到硬盘上之后就记录完了）
17. 能不能不对齐也能做呢？也是可以的、就是要把这个栅栏之后还没处理完的内容、也都算在这个 checkpoint 的 state 里面、好处就是没有这个等的过程，性能更高。坏处就是记录的状态会更多。
18. 这个存这个 snapshot 状态的时候用的是 RocksDB、所以新的状态会更快地被拿到、老的数据就沉下去了。（taskmanager 写入硬盘）
19. 为了做同步、还有用 watermarks 水印来表示的时间，本身也是一个事件。意义表达了、如果这个流是按照顺序来的、那么 watermarks 前面的数据都是小的、watermarks 之后的时间都是大的、那么就可以去除掉这个不符合 watermarks 规则的事件
20. 后面的示意图里、黄色的事件时间就代表当前这个 operator 实际处理过的时间（就是黄色的时间戳只是记录经过的 watermark 的时间、可能会出现虽然处理了 35 的事件、但是还是黄色的 33 时间戳、因为只过去了 33 的 watermark）、白色的就是事件的时间。就是它经过一个 watermark 的时候、代表这个时间点之前的所有事件都已经处理完了、可能处理不是顺序处理、但是都处理了
21. 防止多个流时间不一样 跑乱了
22. 对于这个例子、如果一个 operator 有两个输入、它还是会受限于那个时间戳慢的输入事件。水位线的目的是就是、因为这个事件可能会来得比较乱、所以要有一个规则来保证这个事件是大致有序的。（就是在这个 watermark 之后不会出现比这个数字更小的时间了、但是在这个 watermark 之前、可能会出现比这个数字更大的数字）
23. 单挑水位线出来、保证了严格有序。但是之间的值可能是乱序（相当于一个上界）
24. 这个时间窗 window 应该如何去确定？
    - 第一种是按照相同的时间段（缺陷：当一个时间段内的数据量很大的时候，会导致一个时间段内的数据量很大，或者当一个时间段内的数据量很小的时候，会导致一个时间段内的数据量很小）
    - 第二种是按照相同的数据量（缺陷：会需要等待时间）
25. 管理和 hadoop、yarn 和 spark、storm 差不多、就是有 worker 和 job manager 管理。每一个 worker 上面有一个 task manager、然后 job manager 会把任务分配到 task manager 上去。（也可以多线程地去跑等多个任务）