知道 mysql 的事务隔离吗？/事务隔离分哪几个级别？/事务隔离性怎么保证/...

数据库支持事务本身需要有具备四个特性：原子性、一致性、隔离性和持久性，也即 ACID。其中事务的隔离性简单来说就是并发执行的多个事务之间互不干扰。

事务隔离总体分为四个级别：第一个是 RU(未提交读)，该级别事务内容易出现脏读的情况，即事务 A 读到了事务 B 没有提交的数据；为了解决脏读问题，可以提高事务隔离级别到 RC(已提交读)，此时事务 A 不会读到其他事务未提交的数据，但又产生了一个新的现象：事务 A 执行的过程中，有可能另外一个事务 B 提交了数据，此时事务读取的数据和之前不一致，即出现了不可重复读的问题；所以 mysql 的 InnoDB 本身默认采用了第三个事务隔离级别 RR(可重复读)，该级别使用 MVCC(多版本并发控制)解决重复读的问题(select 的时候采用的方式是快照度, 即将数据照张相, 以后都是读这张照片上的数据, 数据可以重复读)，一般的 RR 级别会出现幻读的问题，即同一个事务多次执行同一个 select，读取到的数据行发生了改变，这是因为数据行发生了行数减少或者新增, 但 mysql 中因为引入了 Gap Lock(间隙锁)的概念，所以 mysql 的 RR 级别同时也解决了幻读的问题；而最高的事务隔离级别是 SE(可序列化)，该方式下事务相当于串行执行，解决了脏读、不可重复读、幻读等问题，但对性能和效率的影响很大，生产环境中很少会使用该隔离级别.

MySQL 和 MongoDB 的区别?

MySQL 是关系型数据库的一种，其存储引擎有 MyISam, InnoDB，Merge 等，目前在业界中大多使用的支持事务 InnoDB 存储引擎，从 MySQL5.7 开始，MySQL 开始支持 json 格式（开始向 nosql 数据库靠近）；而 MongoDB 去年 8 月份发布的最新版本 4.0 开始支持多文档事务，当前其默认的存储引擎 wiredtiger 性能非常卓越.

mysql 和 MongoDB 数据库发展越来越类似，都在取其精华，在具体的数据库选型时，两种 DB 底层的实现机制可能是一大考虑因素。mysql 索引底层是用 B+树实现的，而 MongoDB 则是采取的 B 树，B 树的结构也就决定了 MongoDB 在海量读写的情况下性能比 mysql 卓越（时间复杂度是 O(1)-O(logN)）, 而 MySQL 的 B+树的特性也决定了 MySQL 更适合多区间范围查询的业务需求(时间复杂度为 O(logn))。

当然，在具体的项目 DB 选择过程中，我们还需要考虑到成本、团队成员的意愿、技术栈等情况。总的来说，在当前的情况下，如果是海量数据高并发读写，从技术的角度推荐使用 MongoDB，如果数据结构相对统一，同时对于事务有较高要求，个人倾向于 MySQL。

什么是 IO 多路复用？/epoll 和 select,poll 有什么区别？

I/O 多路复用，I/O 就是指的我们网络 I/O, 多路指多个 TCP 连接(或多个 Channel)，复用指复用一个或少量线程。IO 多路复用指的就是在有很多个网络 I/O 时用一个或少量的线程来处理这些连接.

IO 多路复用最早是在 select 机制中体现的，即在内核中轮询所有文件描述符, 一旦某个描述符就绪, 能够通知程序进行相应的读写操作。在很长的时间内 select 都满足多路复用的各类需求，但随着技术和社会的发展，select 本身 1024 个连接上限逐渐开始不够用了，于是 poll 机制应运而生，poll 机制去掉了 select 的很多问题，比如 1024 链接数限制，但同时本身也存在着缺陷, 比如以轮询的方式在很多场景下会造成性能和资源的浪费。后来一种新的多用复用机制 epoll 随之产生，epoll 采用的是事件触发的机制，每个文件描述符都定义一个回调函数, 只有准备就绪的文件描述符才会执行回调函数, 进行读写操作, epoll 方式放弃了 select 和 poll 轮询的实现方式，大大提高了实际场景中的并发处理能力；但 epoll 目前也存在一些不足的地方，比如只有 Linux 系统才支持，同时在一些特定的场景下，比如绝大部分 TCP 链接都处于就绪的状态下，或连接数较少, 此时比较适合轮询的方式。

同步异步

针对应用程序来，关注的是程序中间的协作关系

阻塞非阻塞

关注的是单个进程的执行状态。

什么时候使用 Python/你对 Python 怎么看/为什么使用 Python?

Python 语言和 C++, Java 等其他编译型语言不一样，它是一种动态解释型语言，代码在执行时会一行一行的解释成 CPU 能理解的机器码。同时它又是跨平台的，可以允许在 windows, linux，mac 等系统上，同样的程序逻辑，可能 C 语言需要 1000 行代码，java 有 100 行，而 Python 实现可能只需要 20 行，这极大程度上提升了企业的项目开发效率。

记得之前看到一篇报道说，到 2014 年为止，哈佛、麻省理工、伯克利等顶尖高校都已经使用 Python 作为教学语言，而最近这一两年来 Python 也多次被各大机构评为年度最佳编程语言。对于企业项目后台开发而言，不是说一定需要使用 Python 开发，但至少应该是一个首选考虑项，当然 Python 本身从根本上来说性能是比不上 C/C++这些编译型语言，但现在语言本身的性能差距在实际开发的过程中，相对业务功能合理性设计、代码编写架构质量等等，语言底层本身造成的性能损失越来越可以忽略不计；针对于一些特定的功能模块，Python 当前也出现了 pypy，JIT 等大幅提高 Python 程序运行效率的相关技术，能够满足大多数功能需求业务场景。

具体选用 Python 后，新项目推荐使用 python3，2008 年 Python3 发布后，十几年来 Python3 生态圈三方库已经非常完备和齐全，而官方已宣布 Python2 在 2020 年将不再维护，所以推荐使用 Python3 版本.

什么是协程

协程又称为微线程, 它是单线程下的并发, 是由用户程序自己控制和调度的.

20 世纪 60 年代，进程的概念被引入，进程作为操作系统资源分配和调度的基本单位，多进程的方式在很长时间内大大提高了系统运行的效率，但进程的频繁创建和销毁代价较大、资源的大量复制和分配耗时仍然较高，于是后来又出现了能够独立运行的单位--线程。多线程之间可以直接共享资源，同时线程之间的通信效率又远高于进程间，将任务并发的性能再次向前推进了一大步。不过多线程也有其不足的地方，虽然线程之间切换代价相较多进程小了很多，但由于 Python 中 GIL 锁的存在，使得多线程无法利用多核的优势, 无法提高运行效率，于是协程的概念又开始发挥了作用，是一个线程在执行，只有当该子程序内部发生中断或阻塞时，才会交出线程的执行权交给其他子程序，在适当的时候再返回来接着执行。这省去了线程间频繁切换的时间开销同时也解决了多线程加锁造成的相关问题.

具体的生产环境中，Python 项目经常会使用多进程+协程的方式，规避 GIL 锁的问题，充分利用多核的同时又充分发挥协程高效的特性。

了解过 tornado 或 sanic 吗？/有木有接触过其他 web 框架？

有了解过 tornado 和 sanic 框架，不过之前公司的项目中大多框架选用的是 django，少部分项目使用的 flask。其中 django 设计哲学是简便、快速，强调代码复用，大而全是代表特性，而 flask 相对而言小而精一点。tornado 和 sanic 是异步框架，性能都非常卓越，tornado 是 Facebook 开源的一个项目，目前应用亦颇为广泛；sanic 是基于 Python3.5 的近些年兴起的框架，用到了很多 Python3 的新特性。

一般在框架选型过程中，如果希望开发过程中框架有丰富的三方插件，推荐使用 django 和 flask，然后部署时使用 NGINX+uwsgi 提高并发；如果纯粹的后端项目，更加追求性能，可以考虑使用 tornado 或者 sanic 等异步框架，像 tornado、sanic 这些支持协程的框架确实比较适合大多数的高并发网络 IO 处理。

消息队列

如果不需要立即获得结果, 但是并发量有需要进行控制时候, 就可以考虑使用消息队列.

消息队列的使用场景

应用耦合: 多个应用程序之间的耦合度较高, 可以通过消息队列实现解耦合, 避免调用接口失败而导致整个服务失败;

异步调用: 多个应用程序对消息队列中的某一消息进行处理, 应用间并发处理消息, 相比串行处理, 减少处理时间(比如注册完成后, 系统需要发送注册邮件, 发送短信并进行短信验证, 发送邮件和发送短信可以并发完成);

限流削峰: 广泛应用于秒杀或抢购活动中，避免流量过大导致应用系统挂掉的情况(比如在双 11 活动中, 由于瞬时访问量过大，服务器接收过大，会导致流量暴增，相关系统无法处理请求甚至崩溃。这时候就可以使用消息队列. 请求先进入消息队列，而不是由业务处理系统直接处理，做了一次缓冲, 极大地减少了业务处理系统的压力；队列长度可以做限制，秒杀时，后入队列的用户无法秒杀到商品，这些请求可以直接被抛弃，返回活动已结束或商品已售完等信息)

消息驱动: 多个子系统间由消息队列连接起来，前一个阶段的处理结果放入队列中，后一个阶段从队列中获取消息继续处理(即所谓的生产者消费者模型)

队列的两种模式

点对点: 一条消息只能被一个消费者接收, 接收者需要应答

发布/订阅: 发布者发布到 topic 的消息，所有订阅了此 topic 的订阅者都会收到

消息队列的实现

RabbitMQ

优点:

1. 性能好，高并发, 单机 QPS(每秒处理的请求量)在万数量级；
2. 可靠性高，有消息确认机制和持久化机制；
3. 可定制路由；
4. 社区活跃，管理界面丰富；

缺陷：

1. 不利于二次开发和维护（erlang 语言）；
2. 接口和协议较复杂，学习成本较高；

Kafka

使用场景: 对消息顺序不依赖，且不是那么实时的系统; 对消息丢失并不那么敏感的系统。

优点:

1. 客户端语言丰富；
2. 性能极佳，单机 QPS 百万级别；
3. 分布式架构，拥有高可用性和可靠性，理论消息存储无上限；
4. 消费者采用 Pull 方式获取消息, 消息有序, 通过控制能够保证所有消息被消费且仅被消费一次;

5. 有优秀的第三方 Kafka Web 管理界面 Kafka-Manager；
6. 在日志领域比较成熟，被多家公司和多个开源项目使用；
7. 支持批量操作；

缺点:

1. 目前支持的功能没有 RabbitMQ 丰富；
2. 消息传递只支持 pull 模式，不支持 push；
3. 一台代理宕机后，消息会发生乱序；

DevOps

它是一组过程、方法与系统的统称, 用于促进开发（应用程序/软件工程）、技术运营和质量保障（QA）部门之间的沟通、协作与整合, 使得各个团队减少时间损耗，更加高效地协同工作。

团队协作工具是什么？

teambition

Zabbix

基于 WEB 界面的提供分布式系统监视

ELK Stack

日志分析系统

三次握手四次挥手

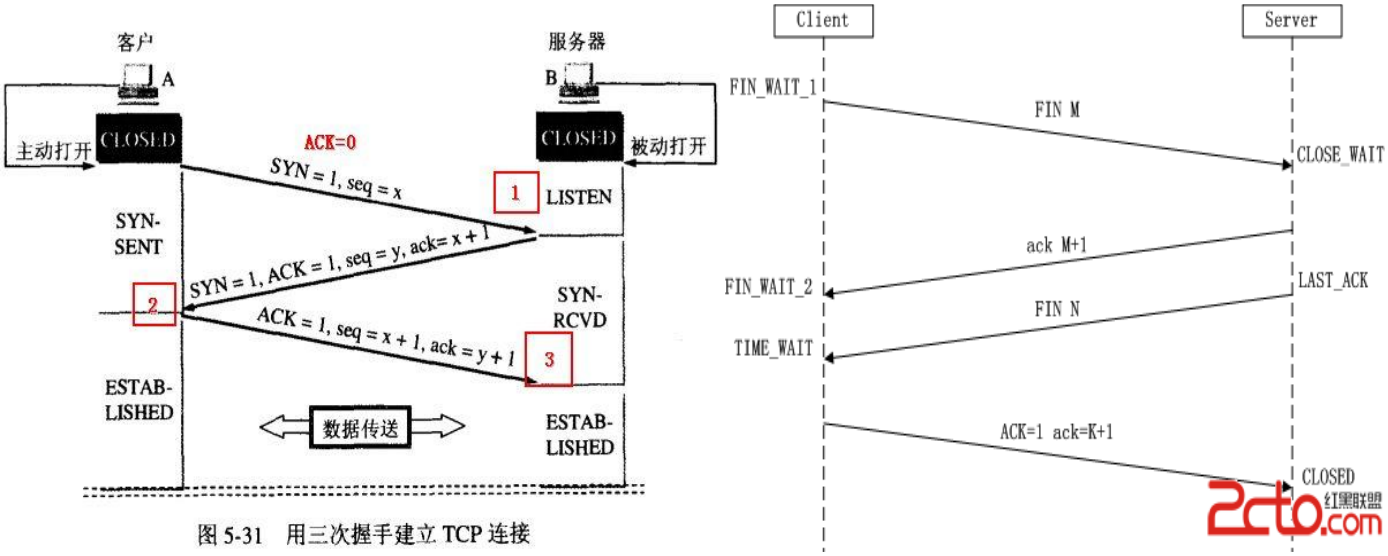

图 5-31　用三次握手建立 TCP 连接

为什么 A 还要发送一次确认呢？这主要是为了防止已失效的连接请求报文段突然又传送到了 B，因而产生错误。

所谓“已失效的连接请求报文段”是这样产生的。考虑一种正常情况。A 发出连接请求，但因连接请求报文丢失而未收到确认。于是 A 再重传一次连接请求。后来收到了确认，建立了连接。数据传输完毕后，就释放了连接。A 共发送了两个连接请求报文段，其中第一个丢失，第二个到达了 B。没有“已失效的连接请求报文段”。

现假定出现一种异常情况，即 A 发出的第一个连接请求报文段并没有丢失，而是在某些网络结点长时间滞留了，以致延误到连接释放以后的某个时间才到达 B。本来这是一个早已失效的报文段。但 B 收到此失效的连接请求报文段后，就误认为是 A 又发出一次新的连接请求。于是就向 A 发出确认报文段，同意建立连接。假定不采用三次握手，那么只要 B 发出确认，新的连接就建立了。

由于现在 A 并没有发出建立连接的请求，因此不会理睬 B 的确认，也不会向 B 发送数据。但 B 却以为新的运输连接已经建立了，并一直等待 A 发来数据。B 的许多资源就这样白白浪费了。

采用三次握手的办法可以防止上述现象的发生。例如在刚才的情况下，A 不会向 B 的确认发出确认。B 由于收不到确认，就知道 A 并没有要求建立连接。