> 乐信

1.画出项目的架构图

2.所处自己负责的业务模块，其中用到了哪些技术点?

3.如何实现最终一致性分布式事务?

1. 二阶段提交：

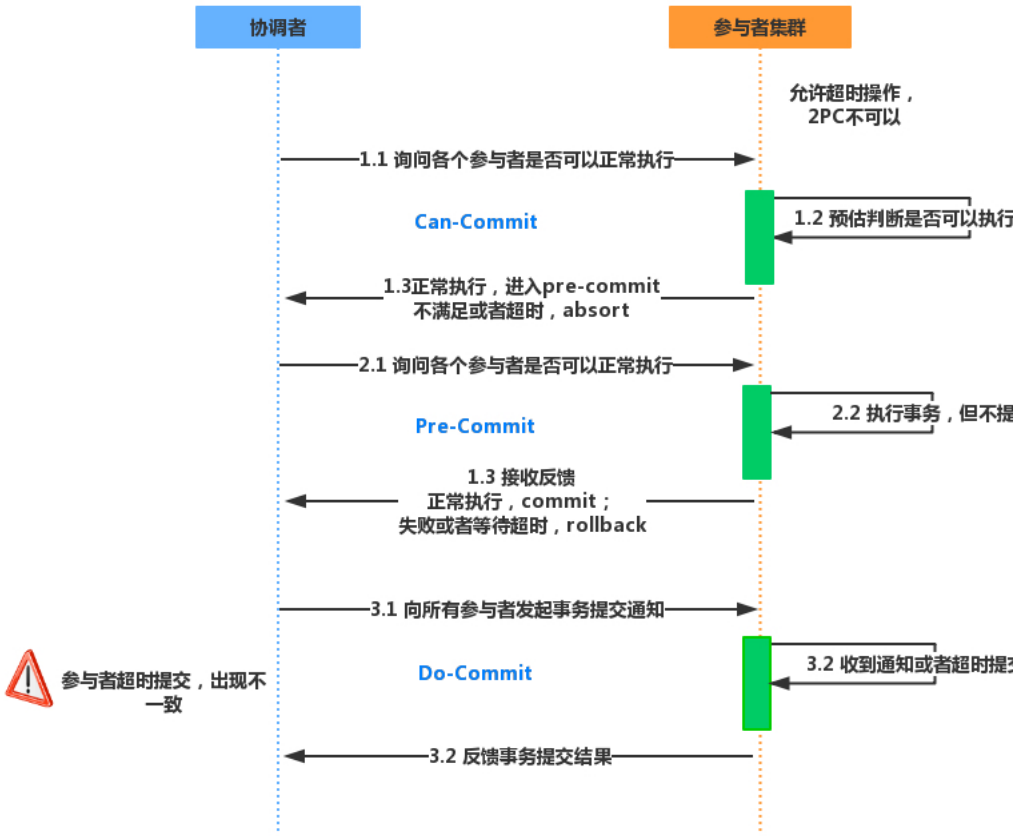
   a. 概念：参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。

   b. 作用：主要保证了分布式事务的原子性；第一阶段为准备阶段，第二阶段为提交阶段；

   c. 缺点：不仅要锁住参与者的所有资源，而且要锁住协调者资源，开销大。一句话总结就是：2PC效率很低，对高并发很不友好。

2. 三阶段提交：

   a. 概念：三阶段提交协议在协调者和参与者中都引入超时机制，并且把两阶段提交协议的第一个阶段拆分成了两步：询问，然后再锁资源，最后真正提交。这样三阶段提交就有CanCommit、PreCommit、DoCommit三个阶段。



b. 缺点：如果进入PreCommit后，Coordinator发出的是abort请求，假设只有一个Cohort收到并进行了abort操作，
而其他对于系统状态未知的Cohort会根据3PC选择继续Commit，此时系统状态发生不一致性。

3. 柔性事务：

a. 概念：所谓柔性事务是相对强制锁表的刚性事务而言。流程入下：服务器A的事务如果执行顺利，那么事务A就先行提交，如果事务B也执行顺利，则事务B也提交，整个事务就算完成。但是如果事务B执行失败，事务B本身回滚，这时事务A已经被提交，所以需要执行一个补偿操作，将已经提交的事务A执行的操作作反操作，恢复到未执行前事务A的状态。

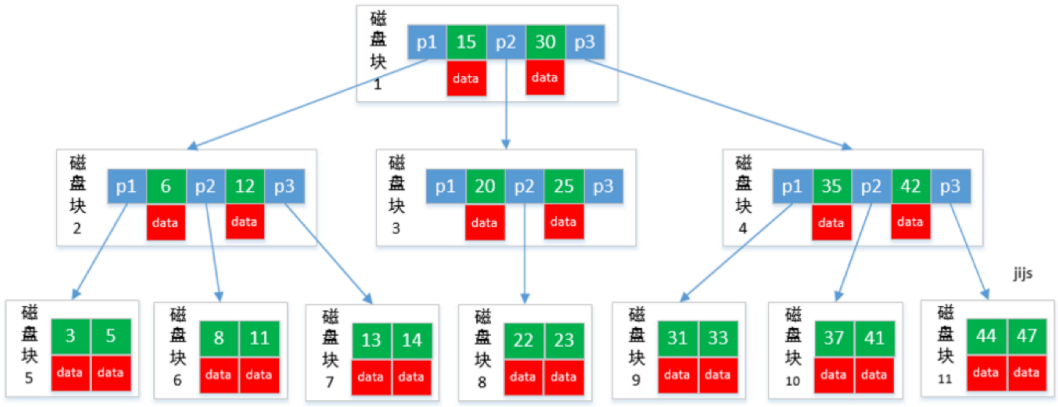
b. 缺点：业务侵入性太强，还要补偿操作，缺乏普遍性，没法大规模推广。

4. 消息最终一致性解决方案之RabbitMQ实现：

a. 实现：发送方确认+消息持久化+消费者确认。

4.索引的B+树结构是咋样的?

1. B-tree：

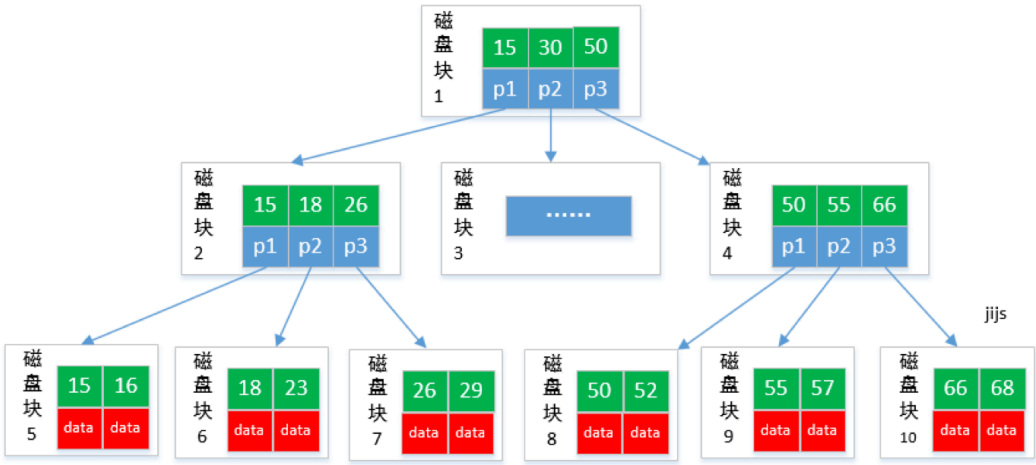


B-tree 利用了磁盘块的特性进行构建的树。每个磁盘块一个节点，每个节点包含了很关键字。把树的节点关键字增多后树的层级比原来的二叉树少了，减少数据查找的次数和复杂度。

B-tree巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页（每页为4K），这样每个节点只需要一次I/O就可以完全载入。

B-tree 的数据可以存在任何节点中。

2. B+tree：



B+tree 是 B-tree 的变种，B+tree 数据只存储在叶子节点中。这样在B树的基础上每个节点存储的关键字数更多，树的层级更少所以查询数据更快，所有指关键字指针都存在叶子节点，所以每次查找的次数都相同所以查询速度更稳定;

5.哪些情况下索引会失效？除了加索引优化查询，还有哪些方法?

6.说说自己了解的设计模式？Spring中用到了哪些设计模式？自己有用过哪些设计模式吗？

1. spring中的设计模式：

a. 简单工厂：spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

b. 单例模式：Spring下默认的bean均为singleton。

c. 代理模式：为其他对象提供一种代理以控制对这个对象的访问。 从结构上来看和Decorator模式类似，但Proxy是控制，更像是一种对功能的限制，而Decorator是增加职责。 spring的Proxy模式在aop中有体现，比如JdkDynamicAopProxy和Cglib2AopProxy。

d. 观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

7.TCP的三次握手四次挥手机制？

1. TCP建立连接的过程。

三次握手：

1. 第一次握手(客户端发送syn包到服务器端)：客户端发送syn包到服务器端，进入syn_send状态，等待服务器端的确认；

2. 第二次握手(服务器返回syn+ack包给客户端)：服务器端收到客户端的syn包，发送syn+ack包给客户端，进入syn_recv状态；

3. 第三次握手(客服端返回ack包给服务端)：客户端收到服务器端的syn+ack包，发送个ack包到服务器端，至此，客户端与服务器端进入established状态；

4. 握手过程中传送的包不包含任何数据，连接建立后才会开始传送数据，理想状态下，TCP连接一旦建立，在通信双方的任何一方主动关闭连接前，TCP连接都会一直保持下去。

2. TCP断开连接的过程。

四次挥手：

1. 第一次挥手：主动关闭方发送fin包到被动关闭方，告诉被动关闭方我不会再给你发送数据了；

2. 第二次挥手：被动关闭方收到syn包，发送ack给对方，确认序号为收到序号+1；

3. 第三次挥手：被动关闭方也也发送fin包给主动关闭方，告诉对方我也

不会给你发送数据了；

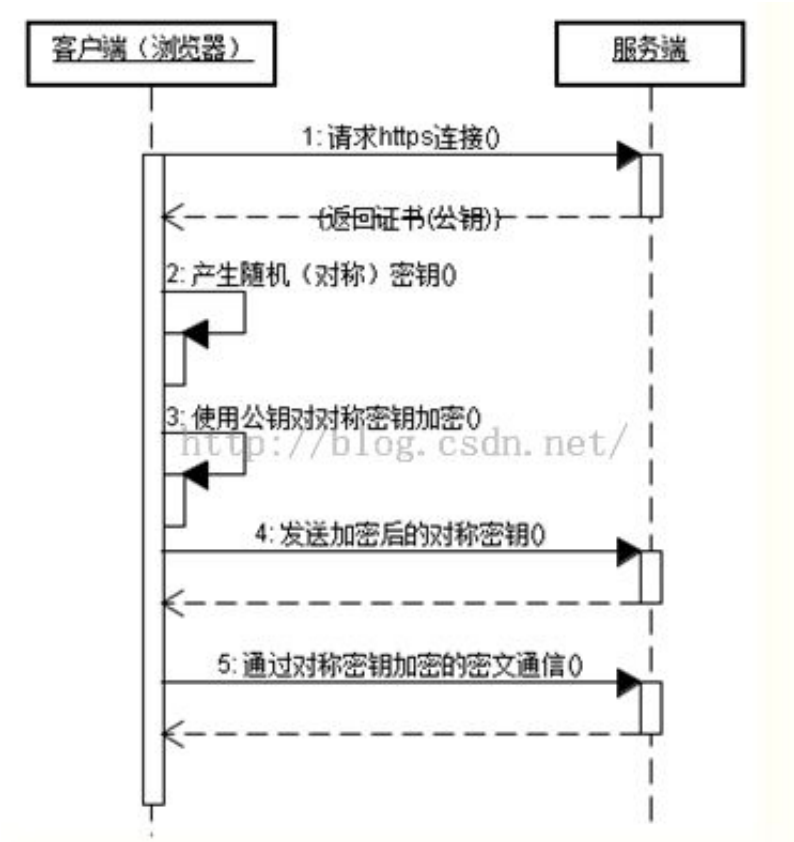
4. 第四次挥手：主动关闭方收到syn包，发送ack给对方，至此，完成四次挥手；

8.Https原理?

HTTPS协议就是基于SSL的HTTP协议

HTTPS使用与HTTP不同的端口（HTTP:80， HTTPS:443）

提供了身份验证与加密通信方法，被广泛用于互联网上安全敏感的通信。



1、客户端请求SSL连接，并将自己支持的加密规则发给网站。

2、服务器端将自己的身份信息以证书形式发回给客户端。证书里面包含了网站地址，加密公钥，以及证书的颁发机构。

3、获得证书后，客户要做以下工作

- 验证证书合法性
- 如果证书受信任，客户端会生成一串随机数的密码，并用证书提供的公钥进行加密。
- 将加密好的随机数发给服务器。

4、获得到客户端发的加密了的随机数之后，服务器用自己的私钥

进行解密，得到这个随机数，把这个随机数作为对称加密的密钥。（利用非对称加密传输对称加密的密钥）

5、之后服务器与客户之间就可以用随机数对各自的信息进行加密，解密。

注意的是：证书是一个公钥，这个公钥是进行加密用的。而私钥是进行解密用的。公钥任何都知道，私钥只有自己知道。这是非对称加密。

而对称加密就是钥匙只有一把，我们都知道。

之所以用到对称加密，是因为对称加密的速度更快。而非对称加密的可靠性更高。

客户端请求--服务端发送证书（公钥）--客户端验证证书，并生成随机数，通过公钥加密后发送给服务端--服务端用私钥解密出随机数--对称加密传输数据。

9.Redis的数据类型有哪些？与Memcached的区别?

Redis目前支持5种数据类型，分别是：

String（字符串）

List（列表）

Hash（字典）

Set（集合）

Sorted Set（有序集合）

区别

1、memcache支持k/v类型数据；

2、redis除了缓存k/v类型数据之外，还能缓存list、set、hash等数据结构的数据；

3.redis的持久化,事务,master/salver这些虽然也是redis的优势,但实际应用中发现这些其实是会托累服务器的性能,而我们全部都不能用.我们得自已做；memcached以上都不管,我们也得自已做；

4.如果需要在服务器端做一些聚合的运算,用redis；如果只是做缓存,redis虽然可以,但性能很差.在要求高性能的环境下使用memcached更合适；

10.消息队列有用到吗？具体在项目中是怎么用的？如何保证消息的可靠传递?

详见“面试题库/RabbitMQ”

> 0304 乐信
>
> 微服务框架用的dubbo

1. 说说java集合，每个集合下面有哪些实现类，及其数据结构?

   深入理解这篇：https://www.jianshu.com/p/63e76826e852

2. 介绍一下红黑树、二叉平衡树。

   理解这篇：https://juejin.im/post/5a27c6946fb9a04509096248

3. jdk1.8中ConcurrentHashMap size大于8时会转化成红黑树，请问有什么作用，如果通过remove操作，size小于8了，会发生什么?

4. 说说java同步机制，java有哪些锁，每个锁的特性?

   看这篇：https://blog.csdn.net/vking_wang/article/details/9952063

5. 说说volatile如何保证可见性，从cpu层面分析。

   需深入理解：https://juejin.im/post/5ae9b41b518825670b33e6c4

6. spring加载bean的顺序?

   spring容器及bean加载机制源码解读：https://blog.csdn.net/songyang19871115/article/details/54342242

7. 哪些对象会被存放到老年代?

   1. 新生代对象每次经历一次minor gc，年龄会加1，当达到年龄阈值（默认为15岁）会直接进入老年代；
   2. 大对象直接进入老年代；
   3. 新生代复制算法需要一个survivor区进行轮换备份，如果出现大量对象在minor gc后仍然存活的情况时，就需要老年代进行分配担保，让survivor无法容纳的对象直接进入老年代；
   4. 如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入年老代。

8. 什么时候触发full gc?

   （1）调用System.gc时，系统建议执行Full GC，但是不必然执行

   （2）老年代空间不足

   （3）方法去空间不足

   （4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

   （5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

9. jvm中哪些地方会出现oom？分别说说oom的可能原因?

jvm发生oom的四种情

况：https://blog.csdn.net/QQ578473688/article/details/77752080

10. 我们如何发现oom来自jvm中哪个区域?
11. 有没有jvm调优经验？调优方案有哪些?
    1. 调优时机：
        a. heap 内存（老年代）持续上涨达到设置的最大内存值；
        b. Full GC 次数频繁；
        c. GC 停顿时间过长（超过1秒）；
        d. 应用出现OutOfMemory 等内存异常；
        e. 应用中有使用本地缓存且占用大量内存空间；
        f. 系统吞吐量与响应性能不高或下降。
    2. 调优原则：
        a. 多数的Java应用不需要在服务器上进行JVM优化；
        b. 多数导致GC问题的Java应用，都不是因为我们参数设置错误，而是代码问题；
        c. 在应用上线之前，先考虑将机器的JVM参数设置到最优（最适合）；
        d. 减少创建对象的数量；
        e. 减少使用全局变量和大对象；
        f. JVM优化是到最后不得已才采用的手段；
        g. 在实际使用中，分析GC情况优化代码比优化JVM参数更好；
    3. 调优目标：
        a. GC低停顿；
        b. GC低频率；
        c. 低内存占用；
        d. 高吞吐量；
    4. 调优步骤：
        a. 分析GC日志及dump文件，判断是否需要优化，确定瓶颈问题点；
        b. 确定jvm调优量化目标；
        c. 确定jvm调优参数（根据历史jvm参数来调整）；
        d. 调优一台服务器，对比观察调优前后的差异；
        e. 不断的分析和调整，知道找到合适的jvm参数配置；
        f. 找到最合适的参数，将这些参数应用到所有服务器，并进行后续

跟踪。

12. 平时有没有看过什么源码，请画出来。

    深入理解：https://juejin.im/post/5caef238e51d456e27504b83

13. 有没有写过或者看过custom classloader?

    了解一下即可：https://www.jianshu.com/p/3036b46f1188

14. 介绍你最近做的一个项目，画出框架图并分析业务流程。

15. 平时看过那些书?