Java内存模型

线程公有

方法区：

Java方法区和堆一样，方法区是一块所有线程共享的内存区域，他保存系统的类信息。比如类的字段、方法、常量池等。方法区的大小决定系统可以保存多少个类。如果系统定义太多的类，导致方法区溢出。虚拟机同样会抛出内存溢出的错误。方法区可以理解为永久区。

堆

虚拟机栈

局部变量表

运行时常量池

操作数栈

主要保存计算过程的中间结果，同时作为计算过程中的变量临时的存储空间

动态连接

方法返回地址

本地方法区

程序计数器

堆讲解

有限分配Eden区-->空间分配担保机制

大对象直接进入老年代-->避免在Eden区和Survivor区之间产生大量的内存复制

对象晋升

年龄阀值 --> -XX:MaxTenuringThreshold, 默认15

动态年龄判定 --> 然而VM并不总是要求对象的年龄必须达到MaxTenuringThreshold才能晋升老年代:

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老年代, 而无须等到晋升年龄.

对象生死判定

可达性分析算法

方法区: 类静态属性引用的对象;

方法区: 常量引用的对象。

虚拟机栈(本地变量表)中引用的对象。

本地方法栈JNI(Native方法)中引用的对象。

垃圾回算法4种

1、引用计数

2、复制算法

3、标记清楚算法

4、标记整理算法

垃圾回收器

目前 没有完美的垃圾收集器，更没有万能的垃圾回收器，只有针对具体应用最适合的垃圾收集器

1、serial 串行

2、paranew 并行

3、cms 并法

4、G1

垃圾回收器

年轻代

Serial收集器---Serial Old收集器：简单高效

一个线程的收集器，在进行垃圾收集的时候，必须暂停其他所有的线程直到他结束收集为止。

设置参数：-XX:+UseSerialGC

开启后会使用Serial(Young区用)+ Serial Old(Old区用)的收集器组合

表示：新生代、老年代都会使用串行垃圾收集器，新生代使用复制算法

ParNew收集器

使用多线程进行垃圾回收，在垃圾收集时，会stw暂停其他所有的工作线程

ParNew收集器其实就是Serial收集器新生代的并行多线程版本，最常见的应用场景时老年代配合CMS工作

设置参数：-XX:+UserParNewGC 启用ParNew收集器，只影响新生代的收集，不影响老年代

开启后ParNew(Young区用)+Serial Old收集器组合，新生代使用复制算法，老年代使用标记-整理算法

备注：-XX:ParallelGCThreads 限制线程数量，默认开启和CPU数目相同的线程数

Parallel Scavenge收集器：jdk1.8默认垃圾回收器

Parallel Scavenge收集器类似ParNew也是一个新生代垃圾回收器，使用复制算法，也是一个并行的多线程

的垃圾回收器，俗称吞吐量优先的垃圾收集器，一句话，穿行收集器在新生代和老年代的并行化

他最关注的是

可控的吞吐量：用户代码运行的时间/（用户代码运行的时间+垃圾收集的时间），高吞吐量意味着高效利用

cpu的时间，他多用户后台运算而不需要太多的交互任务

自适应调节策略：这也是ParallelScavenge收集器与ParNew收集器的一个重要区别（自适应调节策略，虚拟机

会根据当前运行情况收集性能控制信息，动态调整这些参数以提供最合适的停顿时间（-XX:MaxGCPauseMillis）

或最大吞吐量）

设置参数：-XX:+UseParallelGC或-XX:+UseParllelOldGC(相互激活)，开启该参数后，新生代使用复制算法

老年代使用标记-整理算法

老年代

Serial Old收集器

Parallel Old收集器

CMS收集器

优点

并发标记清楚，并发收集地停顿，并发指的是与用户线程一起执行

缺点

CMS在收集与应用线程会同事增加对对内存的占用，也就是CMS必须在老年代用尽之前完成垃圾回收，否则CMS回收失败时

会出发担保机制，串行老年代收集器将会以STW的方式进行一次GC，从而造成较大的停顿时间

标记-清除：产生大量空间碎片，-XX:CMSFullGCsBeForeCompaction来设置多少次full gc之后进行一次整理，

参数设置：-XX:+UseConcMarkSweepGC 开启该参数后，会自动将-XX:+UserParNewGC打开

开启参数后：ParNew(Young区用)+CMS(Old区用)+Serial Old的收集器组合（CMS出错后的备用收集器）

四个阶段

初始标记---> STW. 只是标记一下GC Roots能直接关联的对象，速度很快，仍然需要暂停所有的工作线程

并发标记---> 和用户线程一起工作，不需要暂停工作线程，主要标记过程，标记全部对象

重新标记---> STW. 对之前标记的对象进行二次确认

并发清除---> 和并发标记这两个阶段是最耗时的

特点：

CMS收集器是一种以获取最短回收停顿时间为目标的收集器，这类应用尤其重视服务器的响应速度，希望停顿时间最最短，

CMS非常适合对内存大，CPU核数多的服务器端应用，也是G1出现之前大型应用首选的垃圾回收器

缺点

无法处理浮动垃圾

标记-清除：产生大量空间碎片，当然可以设置多少次full gc之后进行一次整理，=0的情况下，就变成了标记-整理

分区收集

G1收集器 --> G1将整个Java堆划分为多个大小相等的独立区域(Region),

虽然还保留有新生代和老年代的概念, 但新生代和老年代不再是物理隔离的了,

它们都是一部分Region(不需要连续)的集合.

分区：

E：Eden区

S：Survivor区

O：Old区

H：超大对象区,如果一个对象占用的空间超过了分区容量的50%以上，G1收集器就认为这是一个巨型对象，这些巨型对象默认会直接分配到老年到

但是如果他是一个短期存在的巨型对象，就会对垃圾收集器产生负面影响，为了解决这个问题，G1划分了多了个H区，专门用来存储巨型对象

如果一个H区存不下，会寻找多个连续的H区来存储，有时候没有连续的空间，就进行full GC

并发标记阶段

在与应用程序并发执行的过程中会计算活跃度信息.

这些活跃度信息标识出那些regions最适合在STW期间回收

不像CMS有清理阶段.

再次标记阶段

使用Snapshot-at-the-Beginning(SATB)算法比CMS快得多.

空region直接被回收.

拷贝/清理阶段

年轻代与老年代同时回收.

老年代内存回收会基于他的活跃度信息.

G1收集器相对于CMS收集器的优点

G1在压缩空间方面有优势

G1通过将内存空间分成区域（Region）的方式避免内存碎片问题

Eden, Survivor, Old区不再固定、在内存使用效率上来说更灵活

G1可以通过设置预期停顿时间（Pause Time）来控制垃圾收集时间避免应用雪崩现象

G1在回收内存后会马上同时做合并空闲内存的工作、而CMS默认是在STW（stop the world）的时候做

G1会在Young GC中使用、而CMS只能在O区使用

jvm小工具

jps 打印Hotspot VM进程 VMID、JVM参数、main()函数参数、主类名/Jar路径

jstat 查看Hotspot VM 运行时信息 类加载、内存、GC[可分代查看]、JIT编译

jinfo 查看和修改虚拟机各项配置

jmap heapdump: 生成VM堆转储快照、查询finalize执行队列、Java堆和永久代详细信息

jstack 查看VM当前时刻的线程快照: 当前VM内每一条线程正在执行的方法堆栈集合

javap 查看经javac之后产生的JVM字节码代码

jcmd 一个多功能工具, 可以用来导出堆, 查看Java进程、导出线程信息、 执行GC、查看性能相关数据等

jconsole 可以查看内存、线程、类、CPU信息, 以及对JMX MBean进行管理

jvisualvm 可以监控内存泄露、跟踪垃圾回收、执行时内存分析、CPU分析、线程分析…

java.lang.StackOverflowError

递归调用，容易导致栈溢出

java.lang.OutOfMemoryError:Java heap space

堆空间不足导致Java heap spacke

java.lang.OutOfMemoryError:Direct buffer memory

nio磁盘不足导致direct buffer memory

java.lang.OutOfMemoryError:GC overhead limit exceeded

频繁垃圾回收，却没有释放空间，导致Gc overhead limit execpte

java.lang.OutOfMemoryError:unable to create new native thread

高并发unable create new native thread

java.lang.OutOfMemoryError:Metaspace

原空间大小，只受本地内存限制

java.lang.OutOfMemoryError: PermGen space

类或者方法不能被加载到老年代。它可能出现在一个程序加载很多类的时候，比如引用了很多第三方的库

java.lang.OutOfMemoryError: Requested array size exceeds VM limit

创建的数组大于堆内存的空间

java.lang.OutOfMemoryError: request <size> bytes for <reason>. Out of swap space?

同样是本地方法内存分配失败，只不过是JNI或者本地方法或者Java虚拟机发现；

redis数据结构实现

二进制

缓存穿透：缓存中没有，数据库中也没有（解决方案布隆过滤器，数据库中的数据提前加载到布隆过滤器中，存在1，不存在0）

布隆过滤器，多个hash算法，把数据库中的数据映射到布隆过滤器中，保证布隆过滤器中有的，数据库中不一定有，

布隆过滤器中没有的数据库中肯定没有

缓存击穿：缓存中没有，数据库中有（针对并发而言）

分布式锁来解决，限流，拒绝策略

缓存雪崩

因为缓存失效导致数据未加载到缓存中，或者缓存同一时间大面积的实效，从而导致所有的请求都去查询数据库，

导致数据库cpu和内存负载过高，甚至宕机

分布式锁：三个步骤，加锁、解锁、锁超时

1、加锁：lua脚本实现SET lock_key random_value NX PX 5000

2、解锁：删除key

3、伴随线程定时检查过期时间，不断延期，自旋锁

volitile

在变量改变之后，会立刻从cpu高速缓存写到内存

会通知其他cpu缓存中的该变量的值设置成无效，用到该变量时会到内存中重新读取该变量的值。

Lock指令保证了缓存一致性原理。

指令重排

synchronized

对象锁:方法锁（默认锁对象为 this 当前实例对象）,同步代码块锁（自己指定锁对象）

类锁:修饰静态方法，锁 class 对象

可重入：可重入的原理，就和前面提到的 monitor 计数器类似。对象本身有一把锁，JVM 会跟踪对象被加锁的次数：

避免死锁、提升封装性

粒度是线程而不是调用

不可中断

特点

效率低

锁的释放情况少

试图获得锁时不能设定超时时间

不能中断一个正在试图获得锁的线程

不够灵活

加锁和释放锁的时机单一

每个锁仅有单一的条件(某个对象)，可能不够

无法知道是否成功获取到锁

Java线程池

corePoolSize：核心线程数

maximumPoolSize：最大线程数

keepAliveTime：线程存活时间

unit：存活单位

workQueue：阻塞队列

count < corePoolSize: 创建新线程执行任务

count > corePoolSize: workQueue是否满，否：添加到阻塞队列等待执行，是：判断count > maximumPoolSize拒绝策略，创建新线程执行任务

常见线程池

固定容量线程池：newFixedThreadPool(int nThreads, ThreadFactory threadFactory)

Cached线程池：newCachedThreadPool()降低资源消耗

单线程线程池: newSingleThreadExecutor()

定时线程池: newScheduledThreadPool()

优点

降低资源消耗

提高响应速度

提高线程的可管理性

限流算法

固定窗口算法：就是通过维护一个单位时间内的计数值，每当一个请求通过时，就将计数值加1，当计数值超过预先设定的阈值时，

就拒绝单位时间内的其他请求。如果单位时间已经结束，则将计数器清零，开启下一轮的计数。

滑动窗口算法

漏桶算法：特点

漏桶具有固定容量，出口流量速率是固定常量（流出请求）

入口流量可以以任意速率流入到漏桶中（流入请求）

如果入口流量超出了桶的容量，则流入流量会溢出（新请求被拒绝）

令牌桶算法：特点

最多可以存发b个令牌。如果令牌到达时令牌桶已经满了，那么这个令牌会被丢弃

请求到来时，如果令牌桶中少于n个令牌，那么不会删除令牌。该请求会被限流（阻塞活拒绝）

算法允许最大b(令牌桶大小)个请求的突发

MQ

消息丢失

事物机制

此时可以选择用 RabbitMQ 提供的事务功能，就是生产者发送数据之前开启 RabbitMQ 事务channel.txSelect，然后发送消息，

如果消息没有成功被 RabbitMQ 接收到，那么生产者会收到异常报错，此时就可以回滚事务channel.txRollback，然后重试发送消息；

如果收到了消息，那么可以提交事务channel.txCommit。

cnofirm机制

如果你要确保说写 RabbitMQ 的消息别丢，可以开启 confirm 模式，在生产者那里设置开启 confirm 模式之后，

你每次写的消息都会分配一个唯一的 id，然后如果写入了 RabbitMQ 中，RabbitMQ 会给你回传一个 ack 消息，

告诉你说这个消息 ok 了。如果 RabbitMQ 没能处理这个消息，会回调你的一个 nack 接口，告诉你这个消息接收失败，你可以重试。

而且你可以结合这个机制自己在内存里维护每个消息 id 的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

组合方案

生产者

开启RabbktMQ事务---->（同步，不推荐）

开启confirm模式---->（异步，推荐）

MQ：开启RabbitMq持久化

消费者：关闭RabbitMq自动化ACK

RabbitMQ

1、消息持久化：

Exchange 设置持久化：durable:true

Queue 设置持久化

Message持久化发送

2、ACK确认机制

消息发送确认

消息接收确认

3、设置集群镜像模式

4、消息补偿机制

消息必达

消息落地

上半场

MQ-client将消息发送给MQ-server

MQ-server将消息落地，落地后即为发送成功

MQ-server将应答发送给MQ-client（此时回调业务方是API：SendCallback）

下半场

MQ-server将消息发送给MQ-client（此时回调业务方是API：RecvCallback）

MQ-client回复应答给MQ-server（此时业务方主动调用API：SendAck）

MQ-server收到ack，将之前已经落地的消息删除，完成消息的可靠投递

MQ消息投递的上下半场，都可以出现消息丢失，为了降低消息丢失的概率，MQ需要进行超时和重传。

消息重试、重传、反向确认

消息顺序性到达

RabbitMQ

拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确实是麻烦点；

或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。

消息重复性消费

消息做幂等性处理，同一条消息设置一条唯一的uuid，或者唯一值，当服务端消费消息的时候先去查询库中这条消息是否已经存在，如果已经存在则不消费，如果不存在测消费

mysql索引

索引定义

就是按照用户任意指定的字段对数据进行排序的一种数据结构

sql调优

最左前缀原则

架构

第一层：处理客户端连接、授权认证等。

第二层：服务器层，负责查询语句的解析、优化、缓存以及内置函数的实现、存储过程等

第三层：存储引擎，负责 MySQL 中数据的存储和提取。MySQL 中服务器层不管理事务，事务是由存储引擎实现的。

MyISAM：表级锁

Innodb：表级锁+行级锁

redolog（重做日志）

redo log 用于保证事务持久性

InnoDB 作为 MySQL 的存储引擎，数据是存放在磁盘中的，但如果每次读写数据都需要磁盘 IO，效率会很低。

为此，InnoDB 提供了缓存(Buffer Pool)，Buffer Pool 中包含了磁盘中部分数据页的映射，作为访问数据库的缓冲：

当从数据库读取数据时，会首先从 Buffer Pool 中读取，如果 Buffer Pool 中没有，则从磁盘读取后放入 Buffer Pool。

当向数据库写入数据时，会首先写入 Buffer Pool，Buffer Pool 中修改的数据会定期刷新到磁盘中(这一过程称为刷脏)。

Buffer Pool 的使用大大提高了读写数据的效率，但是也带来了新的问题：如果 MySQL 宕机，而此时 Buffer Pool 中修改的数据还没有刷新到磁盘，就会导致数据的丢失，事务的持久性无法保证。

于是，redo log 被引入来解决这个问题：当数据修改时，除了修改 Buffer Pool 中的数据，还会在 redo log 记录这次操作;当事务提交时，会调用 fsync 接口对

redo log 进行刷盘。

如果 MySQL 宕机，重启时可以读取 redo log 中的数据，对数据库进行恢复。

undolog（回滚日志）

undo log 则是事务原子性和隔离性实现的基础

InnoDB 实现回滚，靠的是 undo log：

当事务对数据库进行修改时，InnoDB 会生成对应的 undo log。

如果事务执行失败或调用了 rollback，导致事务需要回滚，便可以利用 undo log 中的信息将数据回滚到修改之前的样子。

binlog

binlog 是用于 point-in-time recovery 的，保证服务器可以基于时间点恢复数据，此外 binlog 还用于主从复制。

事务隔离级别

Read Uncommitted:脏读+可重复读+幻读

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读

Read Committed:幻读+不可重复读

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别 也支持所谓的不可重复读

Repeatable Read:幻读

这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读 （Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题

Serializable：

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

经典问题

脏读：

当前事务(A)中可以读到其他事务(B)未提交的数据(脏数据)，这种现象是脏读。

不可重复读

在事务 A 中先后两次读取同一个数据，两次读取的结果不一样，这种现象称为不可重复读。

幻读

在事务 A 中按照某个条件先后两次查询数据库，两次查询结果的条数不同，这种现象称为幻读。

MVCC

RR 解决脏读、不可重复读、幻读等问题，使用的是 MVCC：MVCC 全称 Multi-Version Concurrency Control，即多版本的并发控制协议。

MVCC 的特点：在同一时刻，不同的事务读取到的数据可能是不同的(即多版本)；

MVCC 最大的优点是读不加锁，因此读写不冲突，并发性能好。InnoDB 实现 MVCC，多个版本的数据可以共存，主要是依靠数据的隐藏列(也可以称之为标记位)和 undo log。

其中数据的隐藏列包括了该行数据的版本号、删除时间、指向 undo log 的指针等等。

Innodb死锁

死锁一般是两个或两个以上事务相互等待对方释放锁，形成死循环所造成的

可能的几种死锁

多个事务按不同的顺序锁定相同的数据集导致的死锁：如果多个事务按不同的顺序锁定相同的数据集，此时事务之间就会形成循环等待造成死锁，这是一种最常见也比较容易理解的死锁。

索引不合理导致的死锁：由于InnoDB的锁是加在索引上的，因此索引不合理将直接导致锁定范围增大，发生锁冲突和死锁的的概率也随着增加。

唯一键值冲突导致的死锁：这个场景主要发生在三个或三个以上的事务同时进行唯一键值相同的记录插入操作

死锁预防策略

减少事务操作的记录数

合理设置索引（索引的粒度为一条记录）

对事务中要操作的记录进行排序

避免使用唯一键值约

总结

原子性：

语句要么全执行，要么全不执行，是事务最核心的特性。事务本身就是以原子性来定义的;实现主要基于 undo log。

持久性：

保证事务提交后不会因为宕机等原因导致数据丢失;实现主要基于 redo log。

隔离性：

保证事务执行尽可能不受其他事务影响;InnoDB 默认的隔离级别是 RR，RR 的实现主要基于锁机制、数据的隐藏列、undo log 和类 next-key lock 机制。

一致性：

事务追求的最终目标，一致性的实现既需要数据库层面的保障，也需要应用层面的保障。

数据结构对比

hash表

范围查询非常不方便，需要遍历所有元素

AVL绝对平衡树

等值查询要比hash表慢一点，支持范围查询

B+树

一个节点有多个元素，叶子节点保存了全部数据，父节点存储了子节点的范围，子节点间有指针，范围查询根据指针就可以查出来

叶子节点具有相同的深度，叶节点得指针为空

所有索引元素不重复

节点中的数据索引从左到右递增排列

特点

非叶子节点不存储data，只存储索引，可以存放更多的索引

也子节点包含所有索引字段

也子节点用指针链接，提高区间访问的性能

红黑树

数据在一定程度上回会产生偏移，造成树得高度特别高，造成多次io，耗费系统性能

主键：一级索引，非主键索引称为：二级索引

MyISAM

索引文件盒数据文件是分离得（非聚集得）

InnoDB

表数据文件本身就是按B+ Tree组织的一个索引结构文件

聚集索引-也子节点包含了完整的数据记录

为什么InnoDB表必须有主见，并且土建使用整型得自增主键？

为什么非主键索引结构也子节点存储得是主键值？(一致性和节省存储空间)

如果用户没有创建索引，mysql会隐藏的创建一列。

ACID: 原子性，一致性，隔离型，持久性

分布式事务

两阶段提交协议：

概念

二阶段提交的算法思路可以概括为: 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。

请求阶段(表决)：

事务协调者通知每个参与者准备提交或取消事务，然后进入表决过程，参与者要么在本地执行事务，写本地的redo和undo日志，但不提交，到达一种"万事俱备，只欠东风"的状态。请求阶段，参与者将告知协调者自己的决策: 同意(事务参与者本地作业执行成功)或取消（本地作业执行故障）

提交阶段(执行):

在该阶段，写调整将基于第一个阶段的投票结果进行决策: 提交或取消

缺点

1、同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。

2、由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。

3、数据不一致.在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。

三阶段提交协议:

改进：

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。

执行步骤

CanCommit阶段：协调者向参与者发送commit请求，参与者如果可以提交就返回Yes响应，否则返回No响应。Coordinator根据Cohort的反应情况来决定是否可以继续事务的PreCommit操作。

PreCommit阶段：协调者向参与者发送准备提交的信息

DoCommit阶段：根据第二阶段的反馈，来最终确认是提交事务还是会滚事务

BASE理论：

BASE是指基本可用（Basically Available）、软状态（ Soft State）、最终一致性（ Eventual Consistency）。

BASE理论是对CAP理论的延伸，思想是即使无法做到强一致性（CAP的一致性就是强一致性），但可以采用适当的采取弱一致性，即最终一致性。

CAP定理：

CAP是一个已经经过证实的理论：一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项。

柔性事务中的服务模式：

1、可查询操作：服务操作具有全局唯一的标识，操作唯一的确定的时间。

2、幂等操作：重复调用多次产生的业务结果与调用一次产生的结果相同。一是通过业务操作实现幂等性，二是系统缓存所有请求　与处理的结果，最后是检测到重复请求之后，自动返回之前的处理结果。

3、TCC操作：Try阶段，尝试执行业务，完成所有业务的检查，实现一致性；预留必须的业务资源，实现准隔离性。Confirm阶段：真正的去执行业务，不做任何检查，仅适用Try阶段预留的业务资源，Confirm操作还要满足幂等性。Cancel阶段：取消执行业务，释放Try阶段预留的业务资源，Cancel操作要满足幂等性。TCC与2PC(两阶段提交)协议的区别：TCC位于业务服务层而不是资源层，TCC没有单独准备阶段，Try操作兼备资源操作与准备的能力，TCC中Try操作可以灵活的选择业务资源，锁定粒度。TCC的开发成本比2PC高。实际上TCC也属于两阶段操作，但是TCC不等同于2PC操作。

4、可补偿操作：Do阶段：真正的执行业务处理，业务处理结果外部可见。Compensate阶段：抵消或者部分撤销正向业务操作的业务结果，补偿操作满足幂等性。约束：补偿操作在业务上可行，由于业务执行结果未隔离或者补偿不完整带来的风险与成本可控。实际上，TCC的Confirm和Cancel操作可以看做是补偿操作

聚集索引：数据与索引存储到一块就是聚集索引

生成条件

1、一般会选择主键列作为聚集索引列。

2、如果没有主键，会选择唯一键。

3、如果都没有。

结构

1、按照聚集索引列的值得顺序，存储数据页

Spring

spring Aop

涉及名词

1、切面（Aspect）：

共有功能的实现。如日志切面、权限切面、验签切面等。在实际开发中通常是一个存放共有功能实现的标准Java类。当Java类使用了@Aspect注解修饰时，就能被AOP容器识别为切面。

2、通知（Advice）：

切面的具体实现。就是要给目标对象织入的事情。以目标方法为参照点，根据放置的地方不同，可分为前置通知（Before）、后置通知（AfterReturning）、异常通知（AfterThrowing）、最终通知（After）与环绕通知（Around）5种。在实际开发中通常是切面类中的一个方法，具体属于哪类通知，通过方法上的注解区分。

3、连接点（JoinPoint）：

程序在运行过程中能够插入切面的地点。例如，方法调用、异常抛出等。Spring只支持方法级的连接点。一个类的所有方法前、后、抛出异常时等都是连接点。

4、切入点（Pointcut）

用于定义通知应该切入到哪些连接点上。不同的通知通常需要切入到不同的连接点上，这种精准的匹配是由切入点的正则表达式来定义的。

5、目标对象(Target)

那些即将切入切面的对象，也就是那些被通知的对象。这些对象专注业务本身的逻辑，所有的共有功能等待AOP容器的切入

6、代理对象（Proxy）

将通知应用到目标对象之后被动态创建的对象。可以简单地理解为，代理对象的功能等于目标对象本身业务逻辑加上共有功能。代理对象对于使用者而言是透明的，是程序运行过程中的产物。目标对象被织入共有功能后产生的对象。

7、织入(Weaving)

将切面应用到目标对象从而创建一个新的代理对象的过程。这个过程可以发生在编译时、类加载时、运行时。Spring是在运行时完成织入，运行时织入通过Java语言的反射机制与动态代理机制来动态实现。

优点

1、降低模块的耦合度

2、使系统更加容易扩展

3、提高代码复用性

JDK动态代理：只能代理接口

CGLIG动态代理：字节码技术完成代理，不能代理final类

spring ioc

优点：

1、bean能够实现集中管理，实现累的可配置和易管理

2、降低类与类之间的耦合度

流程:(循环依赖-currentHashMap,Lasy)

1、实例化一个ApplicationContext的对象；

2：调用bean工厂后置处理器完成扫描；

3：循环解析扫描出来的类信息；

4、实例化一个BeanDefinition对象来存储解析出来的信息；

5、把实例化好的beanDefinition对象put到beanDefinitionMap当中缓存起来，以便后面实例化bean；

6、再次调用bean工厂后置处理器；

7：当然spring还会干很多事情，比如国际化，比如注册BeanPostProcessor等等，如果我们只关心如何实例化一个bean的话那么这一步就是spring调用finishBeanFactoryInitialization方法来实例化单例的bean，实例化之前spring要做验证，需要遍历所有扫描出来的类，依次判断这个bean是否Lazy，是否prototype，是否abstract等等；

8：如果验证完成spring在实例化一个bean之前需要推断构造方法，因为spring实例化对象是通过构造方法反射，故而需要知道用哪个构造方法；

9：推断完构造方法之后spring调用构造方法反射实例化一个对象；注意我这里说的是对象、对象、对象；这个时候对象已经实例化出来了，但是并不是一个完整的bean，最简单的体现是这个时候实例化出来的对象属性是没有注入，所以不是一个完整的bean；

10：spring处理合并后的beanDefinition(合并？是spring当中非常重要的一块内容，后面的文章我会分析)；

11：判断是否需要完成属性注入

12：如果需要完成属性注入，则开始注入属性

13、判断bean的类型回调Aware接口

14、调用生命周期回调方法

15、如果需要代理则完成代理

16、put到单例池——bean完成——存在spring容器当中

事务传播机制

required:支持当前事务，如果当前没有事务，就新建一个事物，这就是最常用得选择

supports:支持当前事务，如果当前没有事务，就以非事物方式执行

mandatory:支持当前事物，如果当前没有事务，就抛出异常

required_new:新建事务，如果存在事务，把当前事务挂起来

not_supported:以非实物方式执行操作，如果当前存在事务，就把当前事务挂起来

never:以非事务得方式运行，如果当前存在事务，则抛出异常

nested:如果当前存在事务，则在嵌套事务内执行，如果当前没有事物，则运行与PROPAGATION_REQUIRED类似的操作。

Java锁

乐观锁

顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS(Compare and Swap 比较并交换)实现的。

乐观锁在Java中的使用，是无锁编程，常常采用的是CAS算法，典型的例子就是原子类，通过CAS自旋实现原子操作的更新。

悲观锁

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。比如Java里面的同步原语synchronized关键字的实现就是悲观锁。悲观锁适合写操作非常多的场景，乐观锁适合读操作非常多的场景，不加锁会带来大量的性能提升。

独享锁

独享锁是指该锁一次只能被一个线程所持有（ReadWriteLock读锁是共享锁）

共享锁

共享锁是指该锁可被多个线程所持有（ReadWriteLock写锁，synchronized是共享锁）

互斥锁

互斥锁在Java中的具体实现就是ReentrantLock。

读写锁

读写锁在Java中的具体实现就是ReadWriteLock。

可重入锁

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。说的有点抽象，下面会有一个代码的示例。

对于Java ReetrantLock而言，从名字就可以看出是一个重入锁，其名字是Re entrant Lock 重新进入锁。

对于Synchronized而言，也是一个可重入锁。可重入锁的一个好处是可一定程度避免死锁。

公平锁

公平锁是指多个线程按照申请锁的顺序来获取锁。

非公平锁

非公平锁是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。有可能，会造成优先级反转或者饥饿现象

对于Java ReetrantLock而言，通过构造函数指定该锁是否是公平锁，默认是非公平锁。非公平锁的优点在于吞吐量比公平锁大。

分段锁

分段锁其实是一种锁的设计，并不是具体的一种锁，对于ConcurrentHashMap而言，其并发的实现就是通过分段锁的形式来实现高效的并发操作。

偏向锁

偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。降低获取锁的代价。

轻量级锁

轻量级锁是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。

重量级锁

重量级锁是指当锁为轻量级锁的时候，另一个线程虽然是自旋，但自旋不会一直持续下去，当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。

重量级锁会让他申请的线程进入阻塞，性能降低。

自旋锁

自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。

dubbo

1、Provider --> 暴露服务的服务提供方

2、Consumer --> 调用远程服务的服务消费方

3、Registry --> 服务注册与发现的注册中心

4、Monitor --> 统计服务的调用次数和调用时间的监控中心

5、Container --> 服务运行容器

dubbo协议

dubbo

http

rest

redis

memcached

负载均衡策略

1、random loadbalance：安权重设置随机概率（默认）；

2、roundrobin loadbalance：轮寻，按照公约后权重设置轮训比例；

3、lastactive loadbalance：最少活跃调用数，若相同则随机；

4、consistenthash：一致性Hash

容错方案

Failover Cluster:失败自动切换，自动重试其他服务器

Failfast Cluster:快速失败，立即报错，直发起一次调用

Failsafe Cluster:失败安全，出现异常时，直接忽略

Failback Cluster：失败自动回复，记录失败请求，定时重发

Forking Cluster：并行多个服务器，只要一个成功立即返回

Broadcast Cluster：广播逐个调用所有提供这，任意一个报错则报错

Dubbo服务之间的调用是阻塞的吗？

默认是同步等待结果阻塞的，支持异步调用。

Dubbo 是基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对

多线程开销较小，异步调用会返回一个 Future 对象。

Dubbo暂时不支持分布式事务。

Dubbo的管理控制台能做什么？

管理控制台主要包含：路由规则，动态配置，服务降级，访问控制，权重调整，负载均衡，等管理功能。

注：dubbo源码中的dubbo-admin模块打成war包，发布运行即可得到dubbo控制管理界面。

Dubbo 服务暴露的过程

Dubbo 会在 Spring 实例化完 bean 之后，在刷新容器最后一步发布 ContextRefreshEvent 事件的时候，通知

实现了 ApplicationListener 的 ServiceBean 类进行回调 onApplicationEvent 事件方法，Dubbo 会在这个方法

中调用 ServiceBean 父类 ServiceConfig 的 export 方法，而该方法真正实现了服务的（异步或者非异步）发

布。

当一个服务接口有多种实现时怎么做？

当一个接口有多种实现时，可以用 group 属性来分组，服务提供方和消费方都指定同一个 group 即可。

服务上线怎么兼容旧版本

可以用版本号（version）过渡，多个不同版本的服务注册到注册中心，版本号不同的服务相互间不引用。这个和服务分组的概念有一点类似。

你觉得用 Dubbo 好还是 Spring Cloud 好？

扩展性的问题，没有好坏，只有适合不适合，我更倾向于使用 Dubbo, Spring Cloud 版本升级太快，组件更新替换太频繁，配置太繁琐。

ZooKeeper

概念

是一个典型的分布式数据一致性解决方案，分布式应用程序可以基于 ZooKeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

特点

顺序一致性： 从同一客户端发起的事务请求，最终将会严格地按照顺序被应用到 ZooKeeper 中去。

原子性： 所有事务请求的处理结果在整个集群中所有机器上的应用情况是一致的，也就是说，要么整个集群中所有的机器都成功应用了某一个事务，要么都没有应用。

单一系统映像： 无论客户端连到哪一个 ZooKeeper 服务器上，其看到的服务端数据模型都是一致的。

可靠性： 一旦一次更改请求被应用，更改的结果就会被持久化，直到被下一次更改覆盖

ZAB 协议&Paxos算法

Paxos 算法应该可以说是 ZooKeeper 的灵魂了。但是，ZooKeeper 并没有完全采用 Paxos算法 ，而是使用 ZAB 协议作为其保证数据一致性的核心算法。另外，在ZooKeeper的官方文档中也指出，ZAB协议并不像 Paxos 算法那样，是一种通用的分布式一致性算法，它是一种特别为Zookeeper设计的崩溃可恢复的原子消息广播算法。

AB 协议介绍

ZAB（ZooKeeper Atomic Broadcast 原子广播） 协议是为分布式协调服务 ZooKeeper 专门设计的一种支持崩溃恢复的原子广播协议。 在 ZooKeeper 中，主要依赖 ZAB 协议来实现分布式数据一致性，基于该协议，ZooKeeper 实现了一种主备模式的系统架构来保持集群中各个副本之间的数据一致性。

ZAB 协议两种基本的模式：崩溃恢复和消息广播

Paxos算法

1、是一种基于消息传递得一致性算法

权限(ACL)

create：创建子节点权限

read：获取节点数据和子节点列表的权限

write：更新节点数据的权限

delete：删除子节点的权限

admin：设置节点acl得权限

节点类型

PERSISTENT：永久节点：节点创建后会被持久化，只有主动调用delete方法的时候才可以删除节点

PERSISTENT_SEQUENTIAL：持久化排序节点

EPHEMERAL：临时节点：节点创建后在创建者超时连接或失去连接的时候，节点会被删除。临时节点下不能存在字节点

EPHEMERAL_SEQUENTIAL：临时排序节点

系统运行速度变得越来越慢

1、之前怀疑是网络，I/O问题，观察网络和磁盘io占用很小

2、怀疑是由于内存占用太高导致的，然后机器内存剩余80G左右，应用内存占用均正常。机器CPU占用70%以下

3、分析heapdump和gc，jstack日志没有发现明显的异常，应用也没有任何报错，无解。

4、对出问题的应用使用jmap -histo:live [pid]命令，应用会恢复正常，目前测试了出问题的几个应用，都有效果，但是目前还不清楚原因。

5、怀疑内存泄露，但是从dump来看并没有什么异常，出问题的应用堆栈基本都在执行数据库操作，就是慢，而且出问题的应用CPU占用比较高达到90%以上。

线程死锁条件

互斥条件：一个资源每次只能被一个进程使用。

请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

不剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺。

循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

避免死锁最简单的方法就是阻止循环等待条件

Mybatis