[一、java基础 3](#_Toc95489576)

[1、arraylist/linkedlist/lru/HashMap的原理---红黑树 3](#_Toc95489577)

[2、synchronized和ReentrantLock原理、可重入性（valatile aqs） 3](#_Toc95489578)

[3、读写锁 3](#_Toc95489579)

[4、乐观锁、悲观锁 4](#_Toc95489580)

[5、cas 4](#_Toc95489581)

[6、自旋锁 4](#_Toc95489582)

[7、无锁 VS 偏向锁 VS 轻量级锁 VS 重量级锁（锁优化、锁升级、锁消除、锁粗化  锁升级的过程） 5](#_Toc95489583)

[8、公平锁 VS 非公平锁 6](#_Toc95489584)

[9、线程池参数及常用的四种类型的线程池，线程创建的流程 7](#_Toc95489585)

[11、Java类加载的过程、为什么使用双亲委派模型 7](#_Toc95489586)

[12、Java的内存模型、那些是共享、哪些私有 8](#_Toc95489587)

[13、垃圾回收器的类型及区别 8](#_Toc95489588)

[14、判断对象存活 8](#_Toc95489589)

[15、垃圾回收器算法 9](#_Toc95489590)

[16、CMS和G1区别 10](#_Toc95489591)

[17、threadLocal如何解决并发问题，会带来什么问题  （删除key就可以了？） 12](#_Toc95489592)

[二、spring基础 13](#_Toc95489593)

[1、循环依赖问题的解决 13](#_Toc95489594)

[2、Spring的IoC理解 13](#_Toc95489595)

[3、Spring的AOP理解 14](#_Toc95489596)

[4、Spring容器的启动流程 14](#_Toc95489597)

[5、Spring Bean的生命周期？ 15](#_Toc95489598)

[6、BeanFactory和ApplicationContext有什么区别？ 17](#_Toc95489599)

[7、Spring中bean的作用域： 17](#_Toc95489600)

[8、Spring的自动装配 17](#_Toc95489601)

[9、 spring的事务传播机制 18](#_Toc95489602)

[10、Spring 框架中都用到了哪些设计模式？ 19](#_Toc95489603)

[11、Spring框架中有哪些不同类型的事件？ 19](#_Toc95489604)

[12、springboot启动过程 19](#_Toc95489605)

[三、常用中间件 20](#_Toc95489606)

[1、redis 20](#_Toc95489607)

[1）5种类型及底层实现 sds 、压缩表、跳表 20](#_Toc95489608)

[2）分布式锁的实现及问题 21](#_Toc95489609)

[3）zookeeper分布式锁特点 22](#_Toc95489610)

[4）redis单线程原理 22](#_Toc95489611)

[5）多线程之lazy free（redis 4.0） 22](#_Toc95489612)

[6) 多线程IO（redis 6.0） 22](#_Toc95489613)

[7）redis与mysql数据一致性 23](#_Toc95489614)

[8）缓存穿透、击穿、雪崩 23](#_Toc95489615)

[9）reactor模型 24](#_Toc95489616)

[10）布隆过滤器 24](#_Toc95489617)

[11）Redis 4.0 对于持久化机制的优化 25](#_Toc95489618)

[12）Redis集群搭建的三种方式 25](#_Toc95489619)

[2、mysql 27](#_Toc95489620)

[1）mysql的逻辑架构&查询流程 27](#_Toc95489621)

[2）缓冲池 (buffer pool) 27](#_Toc95489622)

[3）Innodb的存储格式 28](#_Toc95489623)

[4）索引 29](#_Toc95489624)

[5) 索引的使用 29](#_Toc95489625)

[6）事务 29](#_Toc95489626)

[7）死锁 30](#_Toc95489627)

[8）日志类型 31](#_Toc95489628)

[9）主从复制 31](#_Toc95489629)

[3、mq 32](#_Toc95489630)

[1）基础知识 32](#_Toc95489631)

[2）kafka-基础知识 33](#_Toc95489632)

[3）kafka-保持高可用 33](#_Toc95489633)

[4）kafka-leader副本到follower副本数据同步机制 34](#_Toc95489634)

[5）kafka为什么快 34](#_Toc95489635)

[6）rocketMQ和kafka的区别 35](#_Toc95489636)

[7) rocketMQ-基本原理 35](#_Toc95489637)

[8) rocketMQ为什么不使用zookeeper作为注册中心 36](#_Toc95489638)

[9）rocketMQ-事务消息 36](#_Toc95489639)

[4、zookeeper 37](#_Toc95489640)

[1) 基础结构 37](#_Toc95489641)

[2）Zab 协议中的 Zxid 37](#_Toc95489642)

[3) Zab 协议来保证分布式事务的最终一致性 37](#_Toc95489643)

[4) Zab 与 Paxos 算法的联系与区别 38](#_Toc95489644)

[5、rpc 39](#_Toc95489645)

[1）rmi和rpc的区别 39](#_Toc95489646)

[2）如何实现简单的rpc 39](#_Toc95489647)

[3）为什么实现序列化 39](#_Toc95489648)

[6、mybatis 39](#_Toc95489649)

[四、分布式 39](#_Toc95489650)

[1）CAP和BASE理论 39](#_Toc95489651)

[2）分布式事务算法-2PC 40](#_Toc95489652)

[3）分布式事务算法-3PC 41](#_Toc95489653)

[4）分布式一致性算法-paxos\raft\zab 41](#_Toc95489654)

[5）接口故障应对方案 41](#_Toc95489655)

[6）Eureka 42](#_Toc95489656)

[五、算法 43](#_Toc95489657)

# 一、java基础

## 1、arraylist/linkedlist/lru/HashMap的原理---红黑树

## 2、synchronized和ReentrantLock原理、可重入性（valatile aqs）

ReentrantLock里面有一个内部类Sync，Sync继承AQS（AbstractQueuedSynchronizer），添加锁和释放锁的大部分操作实际上都是在Sync中实现的。它有公平锁FairSync和非公平锁NonfairSync两个子类。ReentrantLock默认使用非公平锁，也可以通过构造器来显示的指定使用公平锁。

Java中ReentrantLock和synchronized都是可重入锁，可重入锁的一个优点是可一定程度避免死锁。

首先ReentrantLock和NonReentrantLock都继承父类AQS，其父类AQS中维护了一个同步状态status来计数重入次数，status初始值为0。

当线程尝试获取锁时，可重入锁先尝试获取并更新status值，如果status == 0表示没有其他线程在执行同步代码，则把status置为1，当前线程开始执行。如果status != 0，则判断当前线程是否是获取到这个锁的线程，如果是的话执行status+1，且当前线程可以再次获取锁。而非可重入锁是直接去获取并尝试更新当前status的值，如果status != 0的话会导致其获取锁失败，当前线程阻塞。

释放锁时，可重入锁同样先获取当前status的值，在当前线程是持有锁的线程的前提下。如果status-1 == 0，则表示当前线程所有重复获取锁的操作都已经执行完毕，然后该线程才会真正释放锁。而非可重入锁则是在确定当前线程是持有锁的线程之后，直接将status置为0，将锁释放。

## 3、读写锁

ReentrantReadWriteLock有两把锁：ReadLock和WriteLock，一个读锁一个写锁，合称“读写锁”。 ReadLock和WriteLock是靠内部类Sync实现的锁。Sync是AQS的一个子类，这种结构在CountDownLatch、ReentrantLock、Semaphore里面也都存在。

使用了AQS的 state字段（int类型，32位），该字段用来描述有多少线程获持有锁。

在独享锁中这个值通常是0或者1（如果是重入锁的话state值就是重入的次数），在共享锁中state就是持有锁的数量。但是在ReentrantReadWriteLock中有读、写两把锁，所以需要在一个整型变量state上分别描述读锁和写锁的数量（或者也可以叫状态）。于是将state变量“按位切割”切分成了两个部分，高16位表示读锁状态（读锁个数），低16位表示写锁状态（写锁个数）

tryAcquire()除了重入条件（当前线程为获取了写锁的线程）之外，增加了一个读锁是否存在的判断。如果存在读锁，则写锁不能被获取，原因在于：必须确保写锁的操作对读锁可见，如果允许读锁在已被获取的情况下对写锁的获取，那么正在运行的其他读线程就无法感知到当前写线程的操作。

因此，只有等待其他读线程都释放了读锁，写锁才能被当前线程获取，而写锁一旦被获取，则其他读写线程的后续访问均被阻塞。

## 4、乐观锁、悲观锁

悲观锁适合写操作多的场景，先加锁可以保证写操作时数据正确。

乐观锁适合读操作多的场景，不加锁的特点能够使其读操作的性能大幅提升

## 5、cas

三个操作值： 需要读写的内存值 V、进行比较的值 A、要写入的新值 B。

三大问题：

1） \*\*ABA问题\*\*。CAS需要在操作值的时候检查内存值是否发生变化，没有发生变化才会更新内存值。但是如果内存值原来是A，后来变成了B，然后又变成了A，那么CAS进行检查时会发现值没有发生变化，但是实际上是有变化的。ABA问题的解决思路就是在变量前面添加版本号，每次变量更新的时候都把版本号加一，这样变化过程就从“A－B－A”变成了“1A－2B－3A”。

JDK从1.5开始提供了AtomicStampedReference类来解决ABA问题，具体操作封装在compareAndSet()中。compareAndSet()首先检查当前引用和当前标志与预期引用和预期标志是否相等，如果都相等，则以原子方式将引用值和标志的值设置为给定的更新值。

2） \*\*循环时间长开销大\*\*。CAS操作如果长时间不成功，会导致其一直自旋，给CPU带来非常大的开销。

3） \*\*只能保证一个共享变量的原子操作\*\*。对一个共享变量执行操作时，CAS能够保证原子操作，但是对多个共享变量操作时，CAS是无法保证操作的原子性的。

Java从1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，可以把多个变量放在一个对象里来进行CAS操作。

## 6、自旋锁

自旋锁本身是有缺点的，它不能代替阻塞。自旋等待虽然避免了线程切换的开销，以及恢复现场的消耗，但它要占用处理器时间。如果锁被占用的时间很短，自旋等待的效果就会非常好。反之，如果锁被占用的时间很长，那么自旋的线程只会白浪费处理器资源。所以，自旋等待的时间必须要有一定的限度，如果自旋超过了限定次数（默认是10次，可以使用-XX:PreBlockSpin来更改）没有成功获得锁，就应当挂起线程。

## 7、无锁 VS 偏向锁 VS 轻量级锁 VS 重量级锁（锁优化、锁升级、锁消除、锁粗化  锁升级的过程）

四种锁是指锁的状态，专门针对synchronized的，需理解：“Java对象头”、“Monitor”。

1）对象头主要包括两部分数据：Mark Word（标记字段）、Klass Pointer（类型指针）。

\*\*Mark Word\*\*：默认存储对象的HashCode，分代年龄和锁标志位信息

\*\*Klass Point\*\*：对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

2）Monitor可以理解为一个同步工具或一种同步机制，通常被描述为一个对象。Monitor是线程私有的数据结构，每一个线程都有一个可用monitor record列表，同时还有一个全局的可用列表。每一个被锁住的对象都会和一个monitor关联，同时monitor中有一个Owner字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用。



无锁没有对资源进行锁定，所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功。

**无锁**

无锁没有对资源进行锁定，所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功。

**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。

在大多数情况下，锁总是由同一线程多次获得，不存在多线程竞争，所以出现了偏向锁。其目标就是在只有一个线程执行同步代码块时能够提高性能。

当一个线程访问同步代码块并获取锁时，会在Mark Word里存储锁偏向的线程ID。在线程进入和退出同步块时不再通过CAS操作来加锁和解锁，而是检测Mark Word里是否存储着指向当前线程的偏向锁。引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令即可。

偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动释放偏向锁。

**轻量级锁**

是指当锁是偏向锁的时候，被另外的线程所访问，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，从而提高性能。

在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，然后拷贝对象头中的Mark Word复制到锁记录中。

拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针，并将Lock Record里的owner指针指向对象的Mark Word。

如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，表示此对象处于轻量级锁定状态。

若当前只有一个等待线程，则该线程通过自旋进行等待。但是当自旋超过一定的次数，或者一个线程在持有锁，一个在自旋，又有第三个来访时，轻量级锁升级为重量级锁。

**重量级锁**

升级为重量级锁时，锁标志的状态值变为“10”，此时Mark Word中存储的是指向重量级锁的指针，此时等待锁的线程都会进入阻塞状态。

## 8、公平锁 VS 非公平锁

公平锁是指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程才能获得锁。公平锁的优点是等待锁的线程不会饿死。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU唤醒阻塞线程的开销比非公平锁大。

非公平锁是多个线程加锁时直接尝试获取锁，获取不到才会到等待队列的队尾等待。但如果此时锁刚好可用，那么这个线程可以无需阻塞直接获取到锁，所以非公平锁有可能出现后申请锁的线程先获取锁的场景。非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU不必唤醒所有线程。缺点是处于等待队列中的线程可能会饿死，或者等很久才会获得锁。

## 9、线程池参数及常用的四种类型的线程池，线程创建的流程

## 11、Java类加载的过程、为什么使用双亲委派模型

1）Java语言系统中支持以下4种类加载器：

- Bootstrap ClassLoader ，主要负责加载Java核心类库，%JRE\_HOME%\lib下的rt.jar、resources.jar、charsets.jar和class等。

- Extention ClassLoader，主要负责加载目录%JRE\_HOME%\lib\ext目录下的jar包和class文件。

- Application ClassLoader ，主要负责加载当前应用的classpath下的所有类

- User ClassLoader ， 用户自定义的类加载器,可加载指定路径的class文件

当一个类加载器收到了类加载的请求的时候，他不会直接去加载指定的类，而是把这个请求委托给自己的父加载器去加载。只有父加载器无法加载这个类的时候，才会由当前这个加载器来负责类的加载。

2）机制的好处：

首先，\*\*通过委派的方式，可以避免类的重复加载\*\*，当父加载器已经加载过某一个类时，子加载器就不会再重新加载这个类。

另外，\*\*通过双亲委派的方式，还保证了安全性\*\*。因为Bootstrap ClassLoader在加载的时候，只会加载JAVA\_HOME中的jar包里面的类，如java.lang.Integer，那么这个类是不会被随意替换的，除非有人跑到你的机器上， 破坏你的JDK。

双亲委派模型中，类加载器之间的父子关系一般不会以继承（Inheritance）的关系来实现，而是都使用组合（Composition）关系来复用父加载器的代码的

3）流程

a、先检查类是否已经被加载过

b、若没有加载则调用父加载器的loadClass()方法进行加载

c、若父加载器为空则默认使用启动类加载器作为父加载器。

d、如果父类加载失败，抛出ClassNotFoundException异常后，再调用自己的findClass()方法进行加载。

4）方法

loadClass（）、findClass（）、defineClass（）区别

ClassLoader中和类加载有关的方法有很多，前面提到了loadClass，除此之外，还有findClass和defineClass等，那么这几个方法有什么区别呢？

- loadClass()

- 就是主要进行类加载的方法，默认的双亲委派机制就实现在这个方法中。

- findClass()

- 根据名称或位置加载.class字节码

- definclass()

- 把字节码转化为Class

这里面需要展开讲一下loadClass和findClass，我们前面说过，当我们想要自定义一个类加载器的时候，并且像破坏双亲委派原则时，我们会重写loadClass方法。

5）为什么Tomcat要破坏双亲委派

Tomcat是web容器，那么一个web容器可能需要部署多个应用程序。

不同的应用程序可能会依赖同一个第三方类库的不同版本，但是不同版本的类库中某一个类的全路径名可能是一样的。

如多个应用都要依赖hollis.jar，但是A应用需要依赖1.0.0版本，但是B应用需要依赖1.0.1版本。这两个版本中都有一个类是com.hollis.Test.class。

\*\*如果采用默认的双亲委派类加载机制，那么是无法加载多个相同的类。\*\*

所以，\*\*Tomcat破坏双亲委派原则，提供隔离的机制，为每个web容器单独提供一个WebAppClassLoader加载器。\*\*

## 12、Java的内存模型、那些是共享、哪些私有

## 13、垃圾回收器的类型及区别

1）G1垃圾回收器

G1最大的优势就在于可预测的停顿时间模型

面向整个堆内存进行回收，把内存划分为多个大小相等的独立区域Region。

一共有4种Region：

1. 自由分区Free Region

2. 年轻代分区Young Region，年轻代还是会存在Eden和Survivor的区分

3. 老年代分区Old Region

4. 大对象分区Humongous Region

每个Region的大小通过`-XX:G1HeapRegionSize`来设置，大小为1~32MB，默认最多可以有2048个Region，那么按照默认值计算G1能管理的最大内存就是32MB\*2048=64G。

2）G1的回收过程分为以下四个步骤：

1. 初始标记：标记GC ROOT能关联到的对象，需要STW（仅标记gc roots能关联到的对象）

2. 并发标记：从GCRoots的直接关联对象开始遍历整个对象图的过程，扫描完成后还会重新处理并发标记过程中产生变动的对象（可达性分析）

3. 最终标记：短暂暂停用户线程，再处理一次，需要STW（修正并发标记期间的变化，变化被记录在log中）

4. 筛选回收：更新Region的统计数据，对每个Region的回收价值和成本排序，根据用户设置的停顿时间制定回收计划。再把需要回收的Region中存活对象复制到空的Region，同时清理旧的Region。需要STW。（将region回收价值排序，根据用户期望进行选择回收计划）

## 14、判断对象存活

1）引用计数算法

2）可达性分析算法

基本思路就是，以一系列被称为\*\*GC Roots\*\*的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为\*\*引用链\*\*，当一个对象到GC Roots没有任何引用链相连时，证明该对象不再存活，可以作为垃圾被回收。

## 15、垃圾回收器算法

**1）标记-清除算法**

当堆中的有效内存空间耗尽时，会触发STW（`stop the world`），然后分\*\*标记\*\*和\*\*清除\*\*两阶段来进行垃圾收集工作：

- 标记：从GC Roots的节点开始进行扫描，对所有存活的对象进行标记，将其记录为可达对象

- 清除：对整个堆内存空间进行扫描，如果发现某个对象未被标记为可达对象，那么将其回收

算法会带来几个问题：

- 在进行GC时会产生STW，停止整个应用程序，造成用户体验较差

- 标记和清除两个阶段的效率都比较低，标记阶段需要从根集合进行扫描，清除阶段需要对堆内所有的对象进行遍历

- 仅对非存活的对象进行处理，清除之后会产生大量不连续的内存碎片。导致之后程序在运行时需要分配较大的对象时，无法找到足够的连续内存，会再触发一次新的垃圾收集动作

此外，jvm并不是真正的把垃圾对象进行了遍历，把内部的数据都删除了，而是把垃圾对象的首地址和尾地址进行了保存，等到再次分配内存时，直接去地址列表中分配，通过这一措施提高了一些标记清除算法的效率。

**2）复制算法**

主要被应用于新生代，它将内存分为大小相同的两块，每次只使用其中的一块。在任意时间点，所有动态分配的对象都只能分配在其中一个内存空间，而另外一个内存空间则是空闲的。**复制算法两个步骤：**

- 当其中一块内存的有效内存空间耗尽后，jvm会停止应用程序运行，开启复制算法的gc线程，将还存活的对象复制到另一块空闲的内存空间。复制后的对象会严格按照内存地址依次排列，同时gc线程会更新存活对象的内存引用地址，指向新的内存地址

- 在复制完成后，再把使用过的空间一次性清理掉，这样就完成了使用的内存空间和空闲内存空间的对调，使每次的内存回收都是对内存空间的一半进行回收

**复制算法的优缺点：**

**优点：**

弥补了标记清除算法中，会出现内存碎片的缺点

**缺点：**

- 只使用了一半的内存，所以内存的利用率较低，造成了浪费

- 如果对象的存活率很高，那么需要将很多对象复制一遍，并且更新它们的应用地址，这一过程花费的时间会非常的长，因此用于新生代更合适.

**3）标记-整理算法**

标记整理算法和标记清除算法非常的类似，主要被应用于老年代中。

**分两个步骤：**

- 标记：和标记清除算法一样，先进行对象的标记，通过GC Roots节点扫描存活对象进行标记

- 整理：将所有存活对象往一端空闲空间移动，按照内存地址依次排序，并更新对应引用的指针，然后清理末端内存地址以外的全部内存空间。

**优缺点：**

标记整理算法对前面的两种算法进行了改进

**优点：**

- 相对于标记清除算法，弥补了出现内存空间碎片的缺点

- 相对于复制算法，弥补了浪费一半内存空间的缺点

**缺点：**

一方面它要标记所有存活对象，另一方面还添加了对象的移动操作以及更新引用地址的操作，因此标记整理算法具有更高的使用成本。

**4）分代收集算法**

java中的垃圾回收器并不是只使用的一种垃圾收集算法，当前大多采用的都是分代收集算法。jvm一般根据对象存活周期的不同，将内存分为几块，一般是把堆内存分为新生代和老年代，再根据各个年代的特点选择最佳的垃圾收集算法。

主要思想如下：

- 新生代中，每次收集都会有大量对象死去，所以可以选择\*\*复制\*\*算法，只需要复制少量对象以及更改引用，就可以完成垃圾收集

- 老年代中，对象存活率比较高，使用复制算法不能很好的提高性能和效率。另外，没有额外的空间对它进行分配担保，因此选择\*\*标记清除\*\*或\*\*标记整理\*\*算法进行垃圾收集

## 16、CMS和G1区别

**1）CMS（Concurrent Mark Sweep）收集器**

是一种以获取最短回收停顿时间为目标的收集器，基于并发“标记清理”实现,在标记清理过程中不会导致用户线程无法定位引用对象。仅作用于老年代收集。

步骤如下：

1. 初始标记（CMS initial mark）：独占CPU，stop-the-world, 仅标记GCroots能直接关联的对象,速度比较快；

2. 并发标记（CMS concurrent mark）:可以和用户线程并发执行，通过GCRoots Tracing 标记所有可达对象；

3. 重新标记（CMS remark）：独占CPU，stop-the-world, 对并发标记阶段用户线程运行产生的垃圾对象进行标记修正,以及更新逃逸对象；

4. 并发清理（CMS concurrent sweep）：可以和用户线程并发执行，清理在重复标记中被标记为可回收的对象。

**优缺点**

**优点：**

- 支持并发收集.

- 低停顿,因为CMS可以控制将耗时的两个stop-the-world操作保持与用户线程恰当的时机并发执行，并且能保证在短时间执行完成，这样就达到了近似并发的目的.

**缺点：**

a、 CMS收集器对CPU资源非常敏感,在并发阶段虽然不会导致用户线程停顿，但是会因为占用了一部分CPU资源，如果在CPU资源不足的情况下应用会有明显的卡顿。

b、无法处理浮动垃圾：在执行‘并发清理’步骤时，用户线程也会同时产生一部分可回收对象，但是这部分可回收对象只能在下次执行清理是才会被回收。如果在清理过程中预留给用户线程的内存不足就会出现‘Concurrent Mode Failure’,一旦出现此错误时便会切换到SerialOld收集方式。

c、CMS清理后会产生大量的内存碎片，当有不足以提供整块连续的空间给新对象/晋升为老年代对象时又会触发FullGC。且在1.9后将其废除。

**使用场景**

它关注的是垃圾回收最短的停顿时间（低停顿），在老年代并不频繁GC的场景下，是比较适用的。

**2）G1**

弱化了CMS原有的分代模型（分代可以是不连续的空间），将堆内存划分成一个个Region（1MB~32MB, 默认2048个分区）,这么做的目的是在进行收集时不必在全堆范围内进行。它主要特点在于达到可控的停顿时间，用户可以指定收集操作在多长时间内完成，即G1提供了接近实时的收集特性。

**步骤：**

1. 初始标记（Initial Marking）：标记一下GC Roots能直接关联到的对象，伴随着一次普通的Young GC发生，并修改NTAMS（Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用的Region中创建新对象，此阶段是stop-the-world操作。

根区间扫描，标记所有幸存者区间的对象引用，扫描 Survivor到老年代的引用，该阶段必须在下一次Young GC 发生前结束。

2. 并发标记（Concurrent Marking）：是从GC Roots开始堆中对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可与用户程序并发执行，该阶段可以被Young GC中断。

3. 最终标记（Final Marking）：是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程Remembered Set Logs里面，最终标记阶段需要把Remembered Set Logs的数据合并到Remembered Set中，此阶段是stop-the-world操作，使用snapshot-at-the-beginning (SATB) 算法。

4. 筛选回收：筛选回收（Live Data Counting and Evacuation）：首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划,回收没有存活对象的Region并加入可用Region队列。这个阶段也可以做到与用户程序一起并发执行，但是因为只回收一部分Region，时间是用户可控制的，而且停顿用户线程将大幅提高收集效率。

**特点：**

- 并行与并发：G1充分发挥多核性能，使用多CPU来缩短Stop-The-world的时间，

- 分代收集：G1能够自己管理不同分代内已创建对象和新对象的收集。

- 空间整合：G1从整体上来看是基于‘标记-整理’算法实现，从局部（相关的两块Region）上来看是基于‘复制’算法实现，这两种算法都不会产生内存空间碎片。

- 可预测的停顿：它可以自定义停顿时间模型，可以指定一段时间内消耗在垃圾回收商的时间不大于预期设定值。

**使用场景**

G1 GC切分堆内存为多个区间（Region），从而避免很多GC操作在整个Java堆或者整个年轻代进行。G1 GC只关注你有没有存货对象，都会被回收并放入可用的Region队列。G1 GC是基于Region的GC，适用于大内存机器。即使内存很大，Region扫描，性能还是很高的。

**内部原理**

**查找Region，使用Remembered Set**

G1在回收每个Region上的垃圾时，每个Region之间又有相互依赖引用关系，想要做到对全部Region进行扫描清理，那么不得不做一次全堆扫描。这样就降低了垃圾回收的效率。所以HotSpot引入了Remembered Set来专门存储于管理对象的引用依赖关系，这样当每次回收时，只需要根据Remembered Set上面的对应关系找到相对的区域进行清理，这样就可以避免扫描整个堆内存又不会遗漏某一个区域。

**查找对象**

GC之前要做一次GC Roots来查找对象的存活情况，以便在GC时候正确的回收。那么每次GC时候遍历所有的引用是不现实的，那么这之后就引入了OopMap，它里面记录了一些类加载时候的类型与偏移量地址等信息生成一张映射表放在OopMap中。GC开始的时候，就通过OopMap这样的一个映射表知道，在对象内的什么偏移量上是什么类型的数据，而且特定的位置记录下栈和寄存器中哪些位置是引用。

**关联对象变化-安全点/安全区域(Safepoint/Safe Region)**

对象引用关系发生变化，更新OopMap是不现实的，这时候就引入了安全点的概念。OopMap的作用就是在每次GC前保证是最新的就可以了。在程序中寻找一个安全点，当GC触发时，为了线程状态和数据的一致性，让线程都跑到这个安全点停顿下来后再执行GC。

**安全点中断GC的方式有两种**：

1. 抢先式中断（Preemptive Suspension）：抢先式中断不需要线程的执行代码主动去配合，在GC发生时，首先把所有线程全部中断，如果发现有线程中断的地方不在安全点上，就恢复线程，让它“跑”到安全点上。现在几乎没有虚拟机采用这种方式来暂停线程从而响应GC事件。

2. 主动式中断（Voluntary Suspension）：主动式中断的思想是当GC需要中断线程的时候，不直接对线程操作，仅仅简单地设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自己中断挂起。轮询标志的地方和安全点是重合的，另外再加上创建对象需要分配内存的地方。

**老年代对象引用年轻代对象情况—卡表、写屏障**

老年代的对象可能引用新生代的对象，由于新生代的垃圾收集通常很频繁，那标记存活对象的时候，需要扫描从老年代到新生代的所有引用对象。因为该对象拥有对新生代对象的引用，那么这个引用也会被称为GC Roots。为防止做全堆扫描,对于HotSpot JVM，使用了卡标记（Card Marking）技术来解决老年代到新生代的引用问题。具体是，使用卡表（Card Table）和写屏障（Write Barrier）来进行标记并加快对GC Roots的扫描。卡表将堆内存平均分成2的N次方大小（默认512字节）个卡，并且维护一个卡表，用来储存每个卡的标识位。当对一个对象引用进行写操作时（对象引用改变），写屏障逻辑将会标记对象所在的卡页为脏页。在YGC只需要扫描卡表中的脏卡，将脏中的对象加入到YGC的GC Roots里面。当完成所有脏卡扫描时候，虚拟机会将卡表的脏卡标志位清空。

## 17、threadLocal如何解决并发问题，会带来什么问题  （删除key就可以了？）

#### 4、  ConcurrentHashmap的

#### 原理1.8及以前的区别

# 二、spring基础

## 1、循环依赖问题的解决

循环依赖问题在Spring中主要有三种情况：

- （1）通过构造方法进行依赖注入时产生的循环依赖问题。

- （2）通过setter方法进行依赖注入且是在多例（原型）模式下产生的循环依赖问题。

- （3）通过setter方法进行依赖注入且是在单例模式下产生的循环依赖问题。

在Spring中，只有第（3）种方式的循环依赖问题被解决了，其他两种方式在遇到循环依赖问题时都会产生异常。

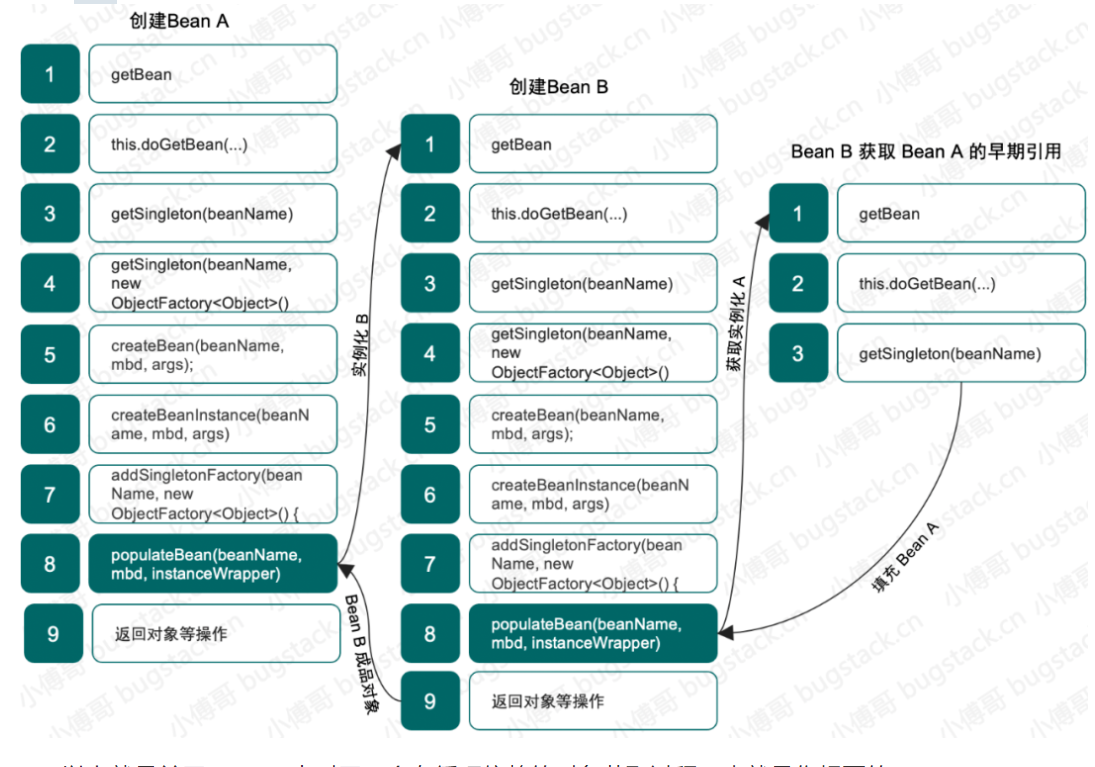
1）Spring 是如何通过三级缓存提前暴露对象来解决循环依赖

对象的创建分成两步：实例化、初始化，实例化指的是从堆中申请内存空间，完成 JVM 层面的对象创建，初始化指的是给属性值赋值

一级缓存里存的是成品对象，实例化和初始化都完成了，我们的应用中使用的对象就是一级缓存中的

二级缓存中存的是半成品，用来解决对象创建过程中的循环依赖问题

三级缓存中存的是 ObjectFactory<?> 类型的 lambda 表达式，用于处理存在 AOP 时能将代理对象提前创建（严格来讲，第三级缓存并非缺它不可，因为可以提前创建代理对象）



## 2、Spring的IoC理解

（1）IOC就是控制反转，指创建对象的控制权转移给Spring框架进行管理，并由Spring根据配置文件去创建实例和管理各个实例之间的依赖关系，对象与对象之间松散耦合，也利于功能的复用。DI依赖注入，和控制反转是同一个概念的不同角度的描述，即 应用程序在运行时依赖IoC容器来动态注入对象需要的外部依赖。

（2）最直观的表达就是，以前创建对象的主动权和时机都是由自己把控的，IOC让对象的创建不用去new了，可以由spring自动生产，使用java的反射机制，根据配置文件在运行时动态的去创建对象以及管理对象，并调用对象的方法的。

（3）Spring的IOC有三种注入方式 ：构造器注入、setter方法注入、接口注入~~根据注解注入~~。

## 3、Spring的AOP理解

AOP，一般称为面向切面，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，提高系统的可维护性。可用于权限认证、日志、事务处理。

AOP实现的关键在于 代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

（1）AspectJ是静态代理，也称为编译时增强，AOP框架会在编译阶段生成AOP代理类，并将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。

（2）Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

​ ① JDK动态代理只提供接口的代理，不支持类的代理，要求被代理类实现接口。JDK动态代理的核心是InvocationHandler接口和Proxy类，在获取代理对象时，使用Proxy类来动态创建目标类的代理类（即最终真正的代理类，这个类继承自Proxy并实现了我们定义的接口），当代理对象调用真实对象的方法时， InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；

> InvocationHandler 的 invoke(Object proxy,Method method,Object[] args)：proxy是最终生成的代理对象; method 是被代理目标实例的某个具体方法; args 是被代理目标实例某个方法的具体入参, 在方法反射调用时使用。

​ ② 如果被代理类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

（3）静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

## 4、Spring容器的启动流程

（1）初始化Spring容器，注册内置的BeanPostProcessor的BeanDefinition到容器中：

> - ① 实例化BeanFactory【DefaultListableBeanFactory】工厂，用于生成Bean对象

> - ② 实例化BeanDefinitionReader注解配置读取器，用于对特定注解（如@Service、@Repository）的类进行读取转化成 BeanDefinition 对象，（BeanDefinition 是 Spring 中极其重要的一个概念，它存储了 bean 对象的所有特征信息，如是否单例，是否懒加载，factoryBeanName 等）

> - ③ 实例化ClassPathBeanDefinitionScanner路径扫描器，用于对指定的包目录进行扫描查找 bean 对象

（2）将配置类的BeanDefinition注册到容器中：

（3）调用refresh()方法刷新容器：

> - ① prepareRefresh()刷新前的预处理：

> - ② obtainFreshBeanFactory()：获取在容器初始化时创建的BeanFactory：

> - ③ prepareBeanFactory(beanFactory)：BeanFactory的预处理工作，向容器中添加一些组件：

> - ④ postProcessBeanFactory(beanFactory)：子类重写该方法，可以实现在BeanFactory创建并预处理完成以后做进一步的设置

> - ⑤ invokeBeanFactoryPostProcessors(beanFactory)：在BeanFactory标准初始化之后执行BeanFactoryPostProcessor的方法，即BeanFactory的后置处理器：

> - ⑥ registerBeanPostProcessors(beanFactory)：向容器中注册Bean的后置处理器BeanPostProcessor，它的主要作用是干预Spring初始化bean的流程，从而完成代理、自动注入、循环依赖等功能

> - ⑦ initMessageSource()：初始化MessageSource组件，主要用于做国际化功能，消息绑定与消息解析：

> - ⑧ initApplicationEventMulticaster()：初始化事件派发器，在注册监听器时会用到：

> - ⑨ onRefresh()：留给子容器、子类重写这个方法，在容器刷新的时候可以自定义逻辑

> - ⑩ registerListeners()：注册监听器：将容器中所有的ApplicationListener注册到事件派发器中，并派发之前步骤产生的事件：

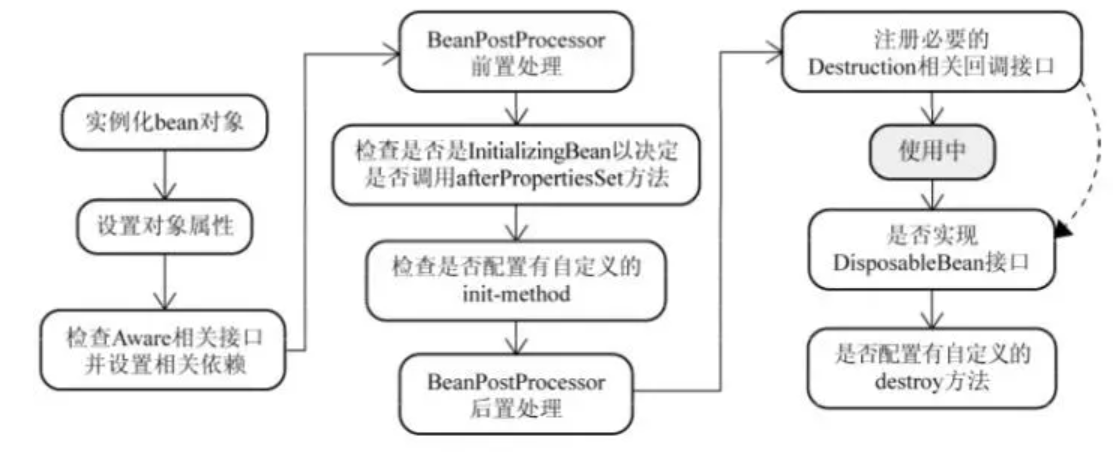
> - ⑪ finishBeanFactoryInitialization(beanFactory)：初始化所有剩下的单实例bean，核心方法是preInstantiateSingletons()，会调用getBean()方法创建对象；

> - ⑫ finishRefresh()：发布BeanFactory容器刷新完成事件

## 5、Spring Bean的生命周期？

简单来说，Spring Bean的生命周期只有四个阶段：实例化 Instantiation --> 属性赋值 Populate --> 初始化 Initialization --> 销毁 Destruction

但具体来说，Spring Bean的生命周期包含下图的流程：



（1）实例化Bean：

对于BeanFactory容器，当客户向容器请求一个尚未初始化的bean时，或初始化bean的时候需要注入另一个尚未初始化的依赖时，容器就会调用createBean进行实例化。

对于ApplicationContext容器，当容器启动结束后，通过获取BeanDefinition对象中的信息，实例化所有的bean。

（2）设置对象属性（依赖注入）：实例化后的对象被封装在BeanWrapper对象中，紧接着，Spring根据BeanDefinition中的信息 以及 通过BeanWrapper提供的设置属性的接口完成属性设置与依赖注入。

（3）处理Aware接口：Spring会检测该对象是否实现了xxxAware接口，通过Aware类型的接口，可以让我们拿到Spring容器的一些资源：

- ①如果这个Bean实现了BeanNameAware接口，会调用它实现的setBeanName(String beanId)方法，传入Bean的名字；

- ②如果这个Bean实现了BeanClassLoaderAware接口，调用setBeanClassLoader()方法，传入ClassLoader对象的实例。

- ②如果这个Bean实现了BeanFactoryAware接口，会调用它实现的setBeanFactory()方法，传递的是Spring工厂自身。

- ③如果这个Bean实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文；

（4）BeanPostProcessor前置处理：如果想对Bean进行一些自定义的前置处理，那么可以让Bean实现了BeanPostProcessor接口，那将会调用postProcessBeforeInitialization(Object obj, String s)方法。

（5）InitializingBean：如果Bean实现了InitializingBean接口，执行afeterPropertiesSet()方法。

（6）init-method：如果Bean在Spring配置文件中配置了 init-method 属性，则会自动调用其配置的初始化方法。

（7）BeanPostProcessor后置处理：如果这个Bean实现了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj, String s)方法；由于这个方法是在Bean初始化结束时调用的，所以可以被应用于内存或缓存技术；

> 以上几个步骤完成后，Bean就已经被正确创建了，之后就可以使用这个Bean了。

（8）DisposableBean：当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，会调用其实现的destroy()方法；

（9）destroy-method：最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法。

## 6、BeanFactory和ApplicationContext有什么区别？

​ BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。

（1）BeanFactory是Spring里面最底层的接口，是IoC的核心，定义了IoC的基本功能，包含了各种Bean的定义、加载、实例化，依赖注入和生命周期管理。ApplicationContext接口作为BeanFactory的子类，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

- 继承MessageSource，因此支持国际化。

- 资源文件访问，如URL和文件（ResourceLoader）。

- 载入多个（有继承关系）上下文（即同时加载多个配置文件） ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

- 提供在监听器中注册bean的事件。

（2）①BeanFactory采用的是延迟加载形式来注入Bean的，只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能提前发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

​ ②ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。

​ ③ApplicationContext启动后预载入所有的单实例Bean，所以在运行的时候速度比较快，因为它们已经创建好了。相对于BeanFactory，ApplicationContext 唯一的不足是占用内存空间，当应用程序配置Bean较多时，程序启动较慢。

(3)BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

（4）BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

## 7、Spring中bean的作用域：

（1）singleton：默认作用域，单例bean，每个容器中只有一个bean的实例。

（2）prototype：为每一个bean请求创建一个实例。

（3）request：为每一个request请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。

（4）session：与request范围类似，同一个session会话共享一个实例，不同会话使用不同的实例。

（5）global-session：全局作用域，所有会话共享一个实例。如果想要声明让所有会话共享的存储变量的话，那么这全局变量需要存储在global-session中。

## 8、Spring的自动装配

在spring中，使用autowire来配置自动装载模式，对象无需自己查找或创建与其关联的其他对象，由容器负责把需要相互协作的对象引用赋予各个对象。

（1）在Spring框架xml配置中共有5种自动装配：

- no：默认的方式是不进行自动装配的，通过手工设置ref属性来进行装配bean。

- byName：通过bean的名称进行自动装配，如果一个bean的 property 与另一bean 的name 相同，就进行自动装配。

- byType：通过参数的数据类型进行自动装配。

- constructor：利用构造函数进行装配，并且构造函数的参数通过byType进行装配。

- autodetect：自动探测，如果有构造方法，通过 construct的方式自动装配，否则使用 byType的方式自动装配。

（2）基于注解的自动装配方式：

使用@Autowired、@Resource注解来自动装配指定的bean。在使用@Autowired注解之前需要在Spring配置文件进行配置，<context:annotation-config />。在启动spring IoC时，容器自动装载了一个AutowiredAnnotationBeanPostProcessor后置处理器，当容器扫描到@Autowied、@Resource或@Inject时，就会在IoC容器自动查找需要的bean，并装配给该对象的属性。在使用@Autowired时，首先在容器中查询对应类型的bean：

如果查询结果刚好为一个，就将该bean装配给@Autowired指定的数据；

如果查询的结果不止一个，那么@Autowired会根据名称来查找；

如果上述查找的结果为空，那么会抛出异常。解决方法时，使用required=false。

> @Autowired可用于：构造函数、成员变量、Setter方法

> 注：@Autowired和@Resource之间的区别：

> (1) @Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。

> (2) @Resource默认是按照名称来装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入。

## 9、 spring的事务传播机制

spring事务的传播机制说的是，当多个事务同时存在的时候，spring如何处理这些事务的行为。事务传播机制实际上是使用简单的ThreadLocal实现的，所以，如果调用的方法是在新线程调用的，事务传播实际上是会失效的。

> ① PROPAGATION\_REQUIRED：（默认传播行为）如果当前没有事务，就创建一个新事务；如果当前存在事务，就加入该事务。

> ② PROPAGATION\_REQUIRES\_NEW：无论当前存不存在事务，都创建新事务进行执行。

> ③ PROPAGATION\_SUPPORTS：如果当前存在事务，就加入该事务；如果当前不存在事务，就以非事务执行。

> ④ PROPAGATION\_NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

> ⑤ PROPAGATION\_NESTED：如果当前存在事务，则在嵌套事务内执行；如果当前没有事务，则按REQUIRED属性执行。

> ⑥ PROPAGATION\_MANDATORY：如果当前存在事务，就加入该事务；如果当前不存在事务，就抛出异常。

> ⑦ PROPAGATION\_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

## 10、Spring 框架中都用到了哪些设计模式？

（1）工厂模式：Spring使用工厂模式，通过BeanFactory和ApplicationContext来创建对象

（2）单例模式：Bean默认为单例模式

（3）策略模式：例如Resource的实现类，针对不同的资源文件，实现了不同方式的资源获取策略

（4）代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术

（5）模板方法：可以将相同部分的代码放在父类中，而将不同的代码放入不同的子类中，用来解决代码重复的问题。比如RestTemplate, JmsTemplate, JpaTemplate

（6）适配器模式：Spring AOP的增强或通知（Advice）使用到了适配器模式，Spring MVC中也是用到了适配器模式适配Controller

（7）观察者模式：Spring事件驱动模型就是观察者模式的一个经典应用。

（8）桥接模式：可以根据客户的需求能够动态切换不同的数据源。比如我们的项目需要连接多个数据库，客户在每次访问中根据需要会去访问不同的数据库

## 11、Spring框架中有哪些不同类型的事件？

Spring 提供了以下5种标准的事件：

（1）上下文更新事件（ContextRefreshedEvent）：在调用ConfigurableApplicationContext 接口中的refresh()方法时被触发。

（2）上下文开始事件（ContextStartedEvent）：当容器调用ConfigurableApplicationContext的Start()方法开始/重新开始容器时触发该事件。

（3）上下文停止事件（ContextStoppedEvent）：当容器调用ConfigurableApplicationContext的Stop()方法停止容器时触发该事件。

（4）上下文关闭事件（ContextClosedEvent）：当ApplicationContext被关闭时触发该事件。容器被关闭时，其管理的所有单例Bean都被销毁。

（5）请求处理事件（RequestHandledEvent）：在Web应用中，当一个http请求（request）结束触发该事件。

如果一个bean实现了ApplicationListener接口，当一个ApplicationEvent 被发布以后，bean会自动被通知。

## 12、springboot启动过程

SpringBoot启动用的是两种设计策略：开箱即用和约定大于配置。

开箱即用：在开发过程中，通过maven项目的pom文件中添加相关依赖包，然后通过相应的注解来代替繁琐的XML配置以管理对象的生命周期。

约定大于配置：由SpringBoot本身来配置目标结构，由开发者在结构中添加信息的软件设计范式。一般分为依赖包的配置文件和本工程的配置文件两部分。

**开箱即用原理：**

SpringBoot项目启动的核心就是@SpringBootApplication注解，它是一个组合注解，核心是由@SpringBootConfiguration@EnableAutoConfiguration再加上@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class)组成。

@SpringBootConfiguration其实就携带了一个@Configuration注解，用于加载自身的Spring的配置类。

@AutoConfigurationPackage 主要包括两个注解，@AutoConfigurationPackage //自动配置包@Import(AutoConfigurationImportSelector.class)//自动配置导入选择。

AutoConfigurationImportSelector，这个类中getCandidateConfigurations()可以帮我们获取所有的配置，通过loadFactoryNames()方法，获取到配置EnableAutoConfiguration注解类下的所有包。而配置了这个注解类的路径就存放在spring.factories这个文件中。但是这个spring.factories文件中的配置并不会全部加载，而是通过@ConditionalOnXXX:如果其中的条件都满足，该类才会生效。最终获取到配置类后，在上层方法中循环封装properties供我们使用。

**约定大于配置：**

就是将yml配置文件加载出来进行自动配置,主要是用了@ConfigurationProperties注解，指定文件，然后将yml文件中的属性赋值给config类进行使用。

⑥    Springmvc流程

# 三、常用中间件

## 1、redis

### 1）5种类型及底层实现 sds 、压缩表、跳表

#### a.字符串对象

SDS(简单动态字符串) 与 C 字符串的优势：

1. 记录字符串长度，降低获取字符串长度时间复杂度；

2. sds扩容时，先检查当前字节数组长度是否充足，若不满足，则先扩容后进行字符串操作，杜绝缓冲区溢出（c字符串可能导致数据覆盖）；

3. 使用free 属性记录空闲空间，实现空间预分配和惰性空间释放两种优化策略，减少修改字符串时带来的内存重分配次数；

预分配的策略：

- len 小于 1MB 时：每次重分配时会多分配同样大小的空闲空间；

- len 大于等于 1MB 时：每次重分配时会多分配 1MB 大小的空闲空间。

惰性空间释放

当字符串缩短时，并不立即使用内存重分配来回收多出来的字节，而是用 free 属性记录，等待将来使用。

4. 二进制安全。

C 字符串只能保存文本数据。

#### b.列表对象

列表对象的编码可以是 linkedlist 或者 ziplist，对应的底层数据结构是双端无环双向链表和压缩列表。

默认情况下，当列表对象保存的所有字符串元素的长度都小于 64 字节，且元素个数小于 512 个时，列表对象采用的是 ziplist 编码，否则使用 linkedlist 编码。

#### c哈希对象

哈希对象的编码可以是 ziplist 或者 hashtable。

#### d集合对象

集合对象的编码可以是 intset 或者 hashtable。当集合对象保存的元素都是整数，并且个数不超过 512 个时，使用 intset 编码，否则使用 hashtable 编码

#### e有序集合对象

有序集合的编码可以是 ziplist 或者 skiplist。当有序集合保存的元素个数小于 128 个，且所有元素成员长度都小于 64 字节时，使用 ziplist 编码，否则使用 skiplist 编码。

### 2）分布式锁的实现及问题

redis分布式锁可能出现的问题

1. 过期时间原子性保证
2. 设置合理过期时间，及锁续期
3. 锁删除判断及原子性保证
4. 主从+哨兵模式，如果主节点宕机，如何保证分布式锁可用（可用RedLock）

Redlock 具体如何使用呢？

整体的流程是这样的，一共分为 5 步：

1. 客户端先获取「当前时间戳T1」

2. 客户端依次向这 5 个 Redis 实例发起加锁请求（用前面讲到的 SET 命令），且每个请求会设置超时时间（毫秒级，要远小于锁的有效时间），如果某一个实例加锁失败（包括网络超时、锁被其它人持有等各种异常情况），就立即向下一个 Redis 实例申请加锁

3. 如果客户端从 >=3 个（大多数）以上 Redis 实例加锁成功，则再次获取「当前时间戳T2」，如果 T2 - T1 < 锁的过期时间，此时，认为客户端加锁成功，否则认为加锁失败

4. 加锁成功，去操作共享资源（例如修改 MySQL 某一行，或发起一个 API 请求）

5. 加锁失败，向「全部节点」发起释放锁请求（前面讲到的 Lua 脚本释放锁）

RedLock缺点：

硬件需要保持时钟正确，并且部署成本比较高，性能较低

### 3）zookeeper分布式锁特点

Zookeeper 通过创建临时节点，来实现分布式锁：

优点：

1. 不需要考虑锁的过期时间

2. watch 机制，加锁失败，可以 watch 等待锁释放，实现乐观锁

劣势：

1. 性能不如 Redis

2. 部署和运维成本高

3. 客户端与 Zookeeper 的长时间失联，锁被释放问题(zookeeper是通过心跳维持链接，保证双端交互session信息，如果因为gc导致心跳暂停，那么session被清除，临时节点也相应的被清除)

### 4）redis单线程原理

Redis将文件事件和时间事件进行抽象，时间轮训器（定时器）会监听I/O事件表，一旦有文件事件就绪，Redis就会优先处理文件事件，接着处理时间事件。Redis都是以`单线程`形式处理，所以说Redis是单线程的。Redis基于Reactor模式开发了自己的I/O事件处理器，也就是文件事件处理器，Redis在I/O事件处理上，采用了I/O多路复用技术（一个线程内同时处理多个socket的IO请求），同时监听多个套接字，并为套接字关联不同的事件处理函数，通过一个线程实现了多客户端并发处理。

Redis单线程只是指其在事件处理上，实际上，Redis也并不是单线程的，比如生成RDB文件，就会fork一个子进程来实现。

### 5）多线程之lazy free（redis 4.0）

Redis 4.0版本引入了`Lazy Free`，将`慢操作`异步化，这也是在事件处理上向多线程迈进了一步。比如Redis实现大键的删除操作异步化，采用非阻塞删除（对应命令`UNLINK`），大键的空间回收交由单独线程实现，主线程只做关系解除，可以快速返回，继续处理其他事件，避免服务器长时间阻塞。

### 6) 多线程IO（redis 6.0）

redis的性能瓶颈并不在CPU上，而是在内存和网络上。因此6.0发布的多线程并未将事件处理改成多线程，而是在I/O上，此外，如果把事件处理改成多线程，不但会导致锁竞争，而且会有频繁的上下文切换，即使用分段锁来减少竞争，对Redis内核也会有较大改动，性能也不一定有明显提升。

Redis实现的多线程部分，利用多核来分担I/O读写负荷。在`事件处理线程`每次获取到可读事件时，会将所有就绪的读事件分配给`I/O线程`，并进行等待，在所有`I/O线程`完成读操作后，`事件处理线程`开始执行任务处理，在处理结束后，同样将写事件分配给`I/O线程`，等待所有`I/O`线程完成写操作。

6.0版本的多线程并非彻底的多线程，`I/O线程`只能同时执行读或者同时执行写操作，期间`事件处理线程`一直处于等待状态，并非流水线模型，有很多轮训等待开销

### 7）redis与mysql数据一致性

**基本流程：**更新数据库-》删除缓存

不要先删除缓存，因为在更新数据库期间有可能会有请求过来，导致缓存重新被赋值旧值。

不要更新缓存，可能因为网络延迟导致数据更新有问题。

**保持数据一致性方案：**延迟双删+删除缓存重试机制

1. 先删除缓存（目的：防止其他线程在更新数据库后再清除缓存期间会拿到旧数据）

2. 写请求更新数据库

3. 休眠一会（比如1秒），再次删除缓存

4. 缓存因为某些原因，删除失败

5. 把删除失败的key放到消息队列

6. 消费消息队列的消息，获取要删除的key

7. 重试删除缓存操作

或读取binlog异步删除缓存

- 可以使用阿里的canal将binlog日志采集发送到MQ队列里面

- 然后通过ACK机制确认处理这条更新消息，删除缓存，保证数据缓存一致性

### 8）缓存穿透、击穿、雪崩

**缓存穿透**

缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id为“-1”的数据或id为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大。

解决方案：

1）采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被 这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

2) 接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；

3) 从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击

**缓存击穿**

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力

解决方案：

1) 设置热点数据永远不过期。

2) 加互斥锁。

**缓存雪崩**

缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。和缓存击穿不同的是， 缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案：

1）缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。

2）如果缓存数据库是分布式部署，将热点数据均匀分布在不同缓存数据库中。

3）设置热点数据永远不过期。

### 9）reactor模型

IO多路复用技术，它的核心就是reactor模型又叫的dispatcher模型，一般分为三种类型：单reactor单进程\单线程模型，单reactor多线程模型和多reactor多进程\多线程模型.

**单reactor单进程\单线程模型**

主要包括reactor、acceptor、handler三个部分.

1）reactor对象通过select监控连接事件，收到事件后通过dispatch分发出去;

2）如果需要建立新连接的事件，那就交由acceptor进行处理，acceptor会创建一个handler处理后续的各种事件

3）如果不是建立连接，那么reactor就直接会调用handler

4）handler读取数据->业务处理->发送数据，完成整个流程

所有这些操作都是在一个线程中完成的，redis中使用的就是这个模型。

**单reactor多线程模型**

在以上基础上，将handler业务处理部分创建一个processor子线程进行异步处理，process处理完毕之后，将相应结果返还给主线程的handler，最终将数据返回给客户端.

单reactor多线程的一个模型，它能够利用cpu多核的能力，但是线程之间进行数据交互、数据共享比较复杂。

**多reactor多线程/多进程模型**

主要包括mainReactor、subReactor、acceptor、handler三个部分.

1）mainReactor对象通过select监控连接事件，收到新的事件后就交由acceptor进行接收，然后分配给subReactor子进程。

2）subReactor将被分配的链接加入链接队列并进行监听，创建一个handler处理后续的各种事件

3）如果不是建立新连接，那么subReactor就直接会调用handler

4）handler读取数据->业务处理->发送数据，完成整个流程

Nginx使用多reactor多进程模型，netty使用多reactor多线程模型

### 10）布隆过滤器

布隆过滤器说某个元素存在，小概率会误判。布隆过滤器说某个元素不在，那么这个元素一定不在。

为什么会出现误判的情况呢?

当一个元素加入布隆过滤器中时：

1）使用布隆过滤器中的哈希函数对元素值进行计算，得到哈希值（有几个哈希函数得到几个哈希值）。

2）根据得到的哈希值，在位数组中把对应下标的值置为 1。

当我们需要判断一个元素是否存在于布隆过滤器的时：

1）对给定元素再次进行相同的哈希计算；

2）得到值之后判断位数组中的每个元素是否都为 1，如果值都为 1，那么说明这个值在布隆过滤器中，如果存在一个值不为 1，说明该元素不在布隆过滤器中。

一定会出现这样一种情况：不同的字符串可能哈希出来的位置相同。 （可以适当增加位数组大小或者调整我们的哈希函数来降低概率）

### 11）Redis 4.0 对于持久化机制的优化&AOF重写

Redis 4.0 开始支持 RDB 和 AOF 的混合持久化（默认关闭，可以通过配置项 aof-use-rdb-preamble 开启）。 如果把混合持久化打开，AOF 重写的时候就直接把 RDB 的内容写到 AOF 文件开头。

优点:是可以结合 RDB 和 AOF 的优点, 快速加载同时避免丢失过多的数据。

缺点:AOF 里面的 RDB 部分是压缩格式不再是 AOF 格式，可读性较差。

AOF 重写

AOF 重写可以产生一个新的 AOF 文件，这个新的 AOF 文件和原有的 AOF 文件所保存的数据库状态一样，但体积更小。

该功能是通过读取数据库中的键值对来实现的，程序无须对现有 AOF 文件进行任何读入、分析或者写入操作。 在执行 BGREWRITEAOF 命令时，Redis 服务器会维护一个 AOF 重写缓冲区，Redis服务器主进程在执行完写命令之后，会同时将这个写命令追加到AOF缓冲区和AOF重写缓冲区。

当子进程完成创建新 AOF 文件的工作之后，服务器会将重写缓冲区中的所有内容追加到新 AOF 文件的末尾，使得新的 AOF 文件保存的数据库状态与现有的数据库状态一致。对新的AOF文件进行改名，原子的覆盖原有的AOF文件；完成新旧两个AOF文件的替换。

### 12）Redis集群搭建的三种方式

**一、Redis主从**

**1.1 Redis主从原理**

和MySQL需要主从复制的原因一样，Redis虽然读取写入的速度都特别快，但是也会产生性能瓶颈，特别是在读压力上，为了分担压力，Redis支持主从复制。Redis的主从结构一主一从，一主多从或级联结构，复制类型可以根据是否是全量而分为全量同步和增量同步。

**1.2Redis主从同步的策略**

主从同步刚连接的时候进行全量同步；全量同步结束后开始增量同步。如果有需要，slave在任何时候都可以发起全量同步，其主要策略就是无论如何首先会尝试进行增量同步，如果同步成功，则会要求slave进行全量同步，之后再进行增量同步。

注意：如果多个slave同时断线需要重启的时候，因为只要slave启动，就会和master建立连接发送SYNC请求和主机全量同步，如果多个同时发送SYNC请求，可能导致master IO突增而发送宕机。

**二、Redis哨兵**

**2.1 Redis哨兵机制**

在主从复制实现之后，如果想对master进行监控，Redis提供了一种哨兵机制，哨兵的含义就是监控Redis系统的运行状态，并做相应的响应。

**2.2 哨兵的任务**

哨兵主要用于管理多个Redis服务器，主要有以下三个任务：

（1）、监控：哨兵会不断的检测master和slave之间是否运行正常；

（2）、提醒：当监控的某个Redis出现问题，哨兵可以通过API向管理员或其他应用程序发送通知；

（3）、故障迁移：当一个master不能正常工作时，哨兵会开始一次自动故障迁移操作，它会将失效master的其中一个slave提升为master，并让失效master和其他slave该为复制新的master，当客户端试图连接失效的master时，集群也会向客户端返回新的master地址，使得集群可以使用新的master代替失效的master。

在一个一主多从的集群中，可以启用多个哨兵进行监控以保证集群足够稳健，这种情况下，哨兵不仅监控主从服务，哨兵之间也会相互监控，建议哨兵至少3个并且是奇数。

**三、Redis集群**

**3.1 Redis集群的特征**

Redis集群有以下几个重要的特征：

（1）、Redis集群的分片特征在于将空间拆分为16384个槽位，某一个节点负责其中一些槽位；

（2）、Redis集群提供一定程度的可用性，可以在某个节点宕机或者不可达的情况继续处理命令；

（3）、Redis集群不存在中心节点或代理节点，集群的其中一个最重要的设计目标是达到线性可扩展性；

**3.2 Redis集群的原理**

正常工作的时候，Redis Cluster中的每个Master节点都会负责一部分的槽，当有某个key被映射到某个Master负责的槽，那么这个Master负责为这个key提供服务。只有Master才拥有槽的所有权，如果是某个Master的slave只负责槽的使用，但是没有所有权。

**3.3 master或集群不可用**

(1)、投票过程是集群中所有master参与,如果半数以上master节点与master节点通信超时(cluster-node-timeout)认为当前master节点挂掉.

(2)、什么时候整个集群不可用(cluster\_state:fail)

a、如果集群任意master挂掉,且当前master没有slave.集群进入fail状态,也可以理解成集群的slot映射[0-16383]不完整时进入fail状态.

b、如果集群超过半数以上master挂掉，无论是否有slave，集群进入fail状态。

①    使用场景、缓存和数据库一致性

②    持久化机制、数据恢复时机

③    5种类型及底层实现 sds 、压缩表、跳表

④    分布式锁的实现（多集群和使用流程）

⑤

⑥    主从复制流程、复制效率问题

⑦    Sentinel作用、下线、选举

⑧    Cluster 槽概念，数据查询流程

⑨    事务ACID与数据库区别

⑩    过期淘汰策略

## 2、mysql

### 1）mysql的逻辑架构&查询流程

第一层是客户端层，它主要是做链接处理，身份验证，安全性功能

这一层主要是有一个连接池，预先在缓冲池里放一定连接，当需要建立连接的时候，就在连接池拿出一个链接，用完之后再还回去。他的作用就是身份验证，线程重用，连接限制，缓存处理等等

第二层就是sql层，包括缓存、解析器、查询优化器。

Querycache缓存主要是对将所有查询结果缓存在内存里，并且结果与query的哈希值对应，但是query查询条件发生变化或表结果或表内容发生变化，缓存就会失效。如果是写多读少的情况，就会导致cpu增高，mysql性能下降。

解析器主要是做sql验证和解析，对sql语句进行解析验证，比如校验语法等，最终生成一个解析树。

查询优化器主要是对查询语句进行一个优化，生成一个最优的策略。比如说where语句，它并不是说将所有的数据都查出来,然后进行数据筛选，而是根据他的条件进行查询，并且查询字段也不是将所有字段都查出来，而是做的字段映射。

最后一层是可插拔的一个存储引擎，存储引擎分为三种，innoDB，MYISAM和memory

### 2）缓冲池 (buffer pool)&LRU数据管理

缓存表数据与索引数据，把磁盘上的数据加载到缓冲池，避免每次访问都进行磁盘 IO，起到加速访问的作用。

**采用预读机制：**

磁盘读写，并不是按需读取，而是按页读取，一次至少读一页数据（一般是 4K），如果未来要读取的数据就在页中，就能够省去后续的磁盘 IO，提高效率。

数据访问，通常都遵循 “集中读写” 的原则，使用一些数据，大概率会使用附近的数据，这就是所谓的“局部性原理”，它表明提前加载是有效的，确实能够减少磁盘 IO。

（1）磁盘访问按页读取能够提高性能，所以缓冲池一般也是按页缓存数据；

（2）预读机制启示了我们，能把一些 “可能要访问” 的页提前加入缓冲池，避免未来的磁盘 IO 操作；

InnoDB使用改造过的LRU算法管理缓存页\*（解决预读失效问题，即预读的页并没有真正被访问）：

（1）将 LRU 分为两个部分：

- 新生代 (new sublist)

- 老生代 (old sublist)

（2）新老生代首尾相连，即：新生代的尾 (tail) 连接着老生代的头 (head)；

（3）新页（例如被预读的页）加入缓冲池时，只加入到老生代头部：

- 如果数据真正被读取（预读成功），才会加入到新生代的头部

- 如果数据没有被读取，则会比新生代里的 “热数据页” 更早被淘汰出缓冲池

MySQL 缓冲池加入了一个 “老生代停留时间窗口” 的机制（解决缓冲池污染问题，即批量访问数据，导致大量热数据被淘汰问题）：

（1）假设 T = 老生代停留时间窗口；

（2）插入老生代头部的页，即使立刻被访问，并不会立刻放入新生代头部；

（3）只有\*\*满足\*\* “被访问” 并且 “在老生代停留时间” 大于 T，才会被放入新生代头部；

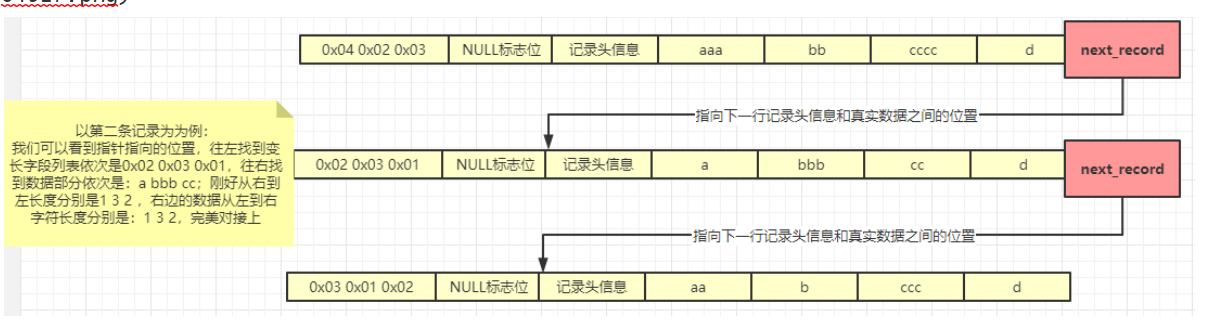
### 3）Innodb的存储格式

InnoDB 储存引擎支持有四种行储存格式：COMPACT、Redundant、Dynamic 和 COMPRESSED。默认为COMPACT。

1. varchar存储

当有多个变长字段的时候，`MySql`在 compact 行格式中，把所有变长类型的长度存放在行记录的开头部位形成一个列表（这个列表就是刚刚上面说的变长字段列表），按照列的逆序存放，每行记录的都有一个 `next\_record`指针 指向下一行 `记录头信息`和 `真实数据` 之间的位置。因为这个位置刚刚好，向左读取就是行描述相关信息，向右读取就是真实数据，方便寻址。

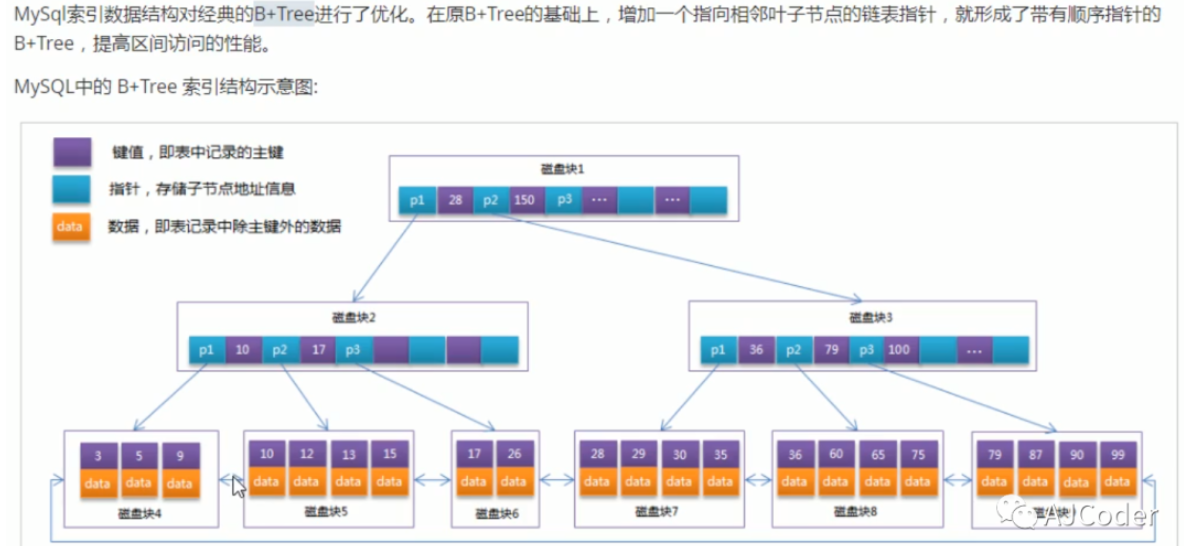
(备注0x04,代表物理偏移量，因为在磁盘中是按照顺序存储的)



1. null存储

Compact 格式数据中的【NULL标志位（也可以叫NULL列表）】就是用来存储NULL值的。若有某个字段值为 null，将将其 bit 位置为 1 说明值为 NULL，bit为 0 说明该字段值不为空

### 4）索引



聚簇索引指索引和数据保存在一起，因此不需要回表就能一次得到相应的数据；而非聚簇索引则需要先得到索引再回表得到对应的数据\*\*。

### 5) 索引的使用

1）全值匹配。对索引中所有列都指定具体值。该情况下，索引生效，执行效率高。

2）\*\*最左前缀法则\*\*。如果索引了多列，要遵守最左前缀法则。指的是查询从索引的最左前列开始，并且不跳过索引中的列。

3）范围查询右边的列，不能使用索引。

4）不要在索引列上进行运算操作，否则索引将失效。

5）字符串不加单引号，索引将失效。

6）尽量使用\*\*覆盖索引\*\*（只访问联合索引其中的字段），避免使用 `select \*`

7）用or分隔开的条件，如果or前的条件中的列有索引，而后面的列中没有索引，那么涉及的索引都不会被用到。

8）以%开头的模糊查询，索引失效。（使用覆盖索引可以解决问题）

9）如果MySQL评估使用索引比全表更慢，则不使用索引。

10）is NULL，is not NULL有时索引失效。

11）in走索引，not in索引失效。

12）单列索引和复合索引。尽量使用复合索引，而少使用单列索引。因为数据库会选择一个最优的索引（辨识度最高的索引）来使用，并不会使用全部索引。

### 6）事务

事务原则（ACID）

原子性（Atomicity）：要么都成功，要么都失败。

一致性（Consistency）：事务前后的数据完整性要保证一致。

隔离性（Isolation）：多个用户并发访问数据库时，数据库为每个用户开启的事务不能被其他事务的操作所干扰。

持久性（Durability）：事务一旦提交则不可逆，被持久化到数据库中。

事务的隔离性是通过锁实现，而事务的原子性、一致性和持久性是通过事务的日志实现的。读未提交:不加锁。

串行化:读的时候加共享锁，可以并发读，但是不能写。写的时候加排它锁，其他事务不能并发写也不能并发读。

可重复读：MySQL 采用了 MVCC (多版本并发控制) 的方式

InnoDB会给数据库中的每一行增加三个字段，它们分别是`DB\_TRX\_ID（6个字节，事务ID）`、`DB\_ROLL\_PTR（7个字节，回滚指针）`、`DB\_ROW\_ID（6个字节，用于聚集索引中的id）。

在InnoDB中，给每行增加两个隐藏字段来实现MVCC，一个用来记录数据行的创建时间，另一个用来记录行的过期时间（删除时间）。在实际操作中，存储的并不是时间，而是事务的版本号，每开启一个新事务，事务的版本号就会递增。（MVCC在mysql中的实现依赖的是undo log与read view

- undo log :undo log 中记录某行数据的多个版本的数据。

- read view :用来判断当前版本数据的可见性）

读已提交则是每次执行select语句的时候都重新生成一次快照，而可重复读是在事务开始的时候生成一个当前事务全局性的快照。所以在读已提交每次查询都可能会读取不同的副本，导致数据不一致。

MySQL 已经在可重复读隔离级别下解决了幻读的问题。解决幻读用的也是锁，叫做间隙锁，MySQL 把行锁和间隙锁合并在一起，解决了并发写和幻读的问题，这个锁叫做Next-Key锁。

普通的SELECT就是快照读，而 `UPDATE、DELETE、INSERT、SELECT ... LOCK IN SHARE MODE、SELECT ... FOR UPDATE` 是当前读。

### 7）死锁

死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统产生了死锁。

如何处理死锁？

答：1）等待，直到超时（innodb\_lock\_wait\_timeout=50s） 2）发起死锁检测，发现死锁后，主动回滚死锁中的某一个事务，让其他事务继续进行。

如何避免死锁？

答：1）在事务中，如果要更新记录，应该直接申请足够级别的锁，即排他锁，而不应该先申请共享锁，更新时再申请排他锁 2）如果事务需要修改或锁定多个表，则应在每个事务中以相同的顺序使用加锁语句。

优化锁？

答：

1）选择合适的事务大小，小的事务发生锁冲突的几率也更小 2）可以适当降低隔离级别 3）创建索引，并尽量使用索引访问数据库，使锁更精确，从而减少锁冲突 4）除非必须，查询时不要显示加锁。MySQL的MVCC可以实现事务中的查询不用加锁，优化事务性能 5）不同的程序访问一组表时，尽量约定以相同的顺序访问各表，对一个表而言，尽可能以固定的顺序存取表中的行，这样可以大大减少死锁的机会

行锁锁的是什么？

答：\*\*存在唯一性索引和主键索引的时候锁的是指定的记录\*\*；\*\*而没有索引的时候锁的是全表\*\*。

不声明主键创建表的危害？

答：

1）不声明主键的情况下，当主键id达到2^32-1时，归0，重新计算，相同的主键直接覆盖记录。声明了主键的情况下，当主键达到最大值时，报错主键冲突。（int改为bigint）

2）行锁锁的是全表。

3）每行记录附加一个隐藏字段叫做\_rowid（6个byte），相较于我们自己声明的4个byte，造成了资源的浪费。

### 8）日志类型

- BinLog:是mysql服务层产生的日志，常用来进行数据恢复、数据库复制，常见的mysql主从架构，就是采用slave同步master的binlog实现的。

binlog刷盘时机

对于InnoDB存储引擎而言，只有在事务提交时才会记录binlog，此时记录还在内存中，那么binlog是什么时候刷到磁盘中的呢？mysql通过sync\_binlog参数控制binlog的刷盘时机，取值范围是0-N：

- 0：不去强制要求，由系统自行判断何时写入磁盘；

- 1：每次commit的时候都要将binlog写入磁盘；

- N：每N个事务，才会将binlog写入磁盘。

- Redo Log:记录了数据操作在物理层面的修改，mysql 为了提升性能不会把每次的修改都实时同步到磁盘，而是会先存到Buffer Pool(缓冲池)里头，把这个当作缓存来用。然后使用后台线程去做缓冲池和磁盘之间的同步。事务进行中时会不断的产生redo log，在事务提交时进行一次flush操作，保存到磁盘中。当数据库或主机失效重启时，会根据redo log进行数据的恢复。redo log是使用ib\_logfile0和ib\_logfile1两个文件循环写入，一个写满就使用另外一个。

redo log包括两部分：一个是内存中的日志缓冲(redo log buffer)，另一个是磁盘上的日志文件(redo log file)。mysql每执行一条DML语句，先将记录写入redo log buffer，后续某个时间点再一次性将多个操作记录写到redo log file。这种先写日志，再写磁盘的技术就是MySQL里经常说到的WAL(Write-Ahead Logging) 技术。

redo log实际上记录数据页的变更，而这种变更记录是没必要全部保存，因此redo log实现上采用了大小固定，循环写入的方式，当写到结尾时，会回到开头循环写日志。

- Undo Log: 除了记录redo log外，当进行数据修改时还会记录undo log，undo log用于数据的撤回操作，它记录了修改的反向操作，比如，插入对应删除，修改对应修改为原来的数据，通过undo log可以实现事务回滚，并且可以根据undo log回溯到某个特定的版本的数据，实现MVCC，保证原子性。

<https://jishuin.proginn.com/p/763bfbd5ce13>

### 9）主从复制

主节点会生成一个发送bin log的线程，并且从节点也会产生一个连接主节点的IO线程，此IO线程会分批的获取bin\_log指定位置后的日志信息，包括bin\_log的二进制名和位置信息。

新版本mysql使用GTID(全局事务标识)完成主从复制：

1）GTID是在主节点在更新数据的时候，会在事务前产生GTID，然后一同记录的binlog日志

2）从节点的io线程将bin\_log日志写入到本地的relay\_log里。

3）sql线程将relay\_log里获取的GTID，然后对比一下本地的bin\_log中是否具备GTID，如果是有的话，那么就忽略本次同步，如果没有的话，那么就开始执行GTID事物，并记录到本地bin\_log里

4）使用GTID比传统主从复制更加的方便，他只需要知道GTID号就可完成主从复制，并不依赖原先bin\_log二进制名称以及position点

①常用的优化策略，平常是怎么使用的

②    数据库隔离级别，原理、MVCC

③    索引的原理、回表的原理、索引失效

④    数据库的锁的种类及使用场景(行锁、表锁、乐观、悲观)

⑤    存储引擎的区别

⑥    持久性原理

⑦ 原子性原理

## 3、mq

### 1）基础知识

**a）优势**

- 削峰填谷（主要解决瞬时写压力大于应用服务能力导致消息丢失、系统崩溃等问题）

- 系统解耦（解决不同重要程度、不同能力级别系统之间依赖导致一死全死）

- 提升性能（当存在一对多调用时，可以发一条消息给消息系统，让消息系统通知相关系统）

- 蓄流压测（线上有些链路不好压测，可以通过堆积一定量消息再放开来压测）

**b) 如何保证消息不丢失**

从三个方面考虑

1.生产者

那生产者可以从自身的机制和我们业务逻辑保证消息不丢失。MQ自身机制比如kafka的ACK机制，包括0、1、-1三种选择，相应的RocketMQ也有相应的机制。从业务逻辑上，一般采用发送消息失败后，将数据打标，使用定时器重新发送进行托底，再加上报警+人工处理等方法处理。

2.Broker存储

Broker消息存储是在消息发送成功后，将数据发送到master，然后master将数据同步slave，比如说rocketMQ的话，同步刷盘和异步刷盘机制，保证消息不丢失。

3.消费者

消费者端，其实考虑两种情况，一个是宕机，另一个就是发生异常。消费者宕机情况，像kafka默认使用自动提交机制，如果在消费过程中突然宕机，消息没有消费，但是自动提交offset，此时消息不会再进行重新消费，消息丢失了，可以采用手动提交的一个方案。

发生异常情况一般是我们的业务处理中发生异常，我们可以采取多种方案处理，比如说自产自销的情况下发生异常，那么就可以将消息重新扔到消息队列重新消费。或者是采用异常向外抛出方案，像jmq，消息消费异常的话，自动把它重新放重试队列之中。而且我们应该消费者端和生产端建立一个ACK机制，也就是说消费后将状态同步给生产端，若中途异常，采用消息重发托底。

### 2）kafka-基础知识

Partitions 分区：把 Topic 看做一个队列，实际上，一个 Topic 是由多个队列组成的，被称为【Partition（分区）】。这样可以便于 Topic 的扩展,防止数据太多使单台broker无法容纳。

生产者发送消息时，默认是面向 Topic 的，由 Topic 决定放在哪个 Partition，默认使用轮询策略。

也可以配置 Topic，让同类型的消息都在同一个 Partition。保持消息的【有序性】

ZooKeeper 管理者所有的 Topic 和 Partition。Topic 和 Partition 存储在 Node 物理节点中，ZooKeeper负责维护这些 Node。

Topic A 的 Partition #1 有3份，分布在各个 Node 上。这样可以增加 Kafka 的可靠性和系统弹性。3个 Partition #1 中，ZooKeeper 会指定一个 Leader，负责接收生产者发来的消息。

### 3）kafka-保持高可用

①备份机制

Kafka允许同一个Partition存在多个消息副本，每个Partition的副本通常由1个Leader及0个以上的Follower组成，生产者将消息直接发往对应Partition的Leader，Follower会周期地向Leader发送同步请求

同一Partition的Replica不应存储在同一个Broker上

②ISR机制（in-sync Replica）

ISR 中的副本都是与 Leader 同步的副本，相反，不在 ISR 中的追随者副本就被认为是与 Leader 不同步，保持同步不是指与Leader数据保持完全一致，只需在`replica.lag.time.max.ms`时间内与Leader保持有效连接，各Partition的Leader负责维护ISR列表并将ISR的变更同步至ZooKeeper，被移出ISR的Follower会继续向Leader发FetchRequest请求，试图再次跟上Leader重新进入ISR。

③ACK机制

「acks=0」

生产者无需等待服务端的任何确认，消息被添加到生产者套接字缓冲区后就视为已发送，因此acks=0不能保证服务端已收到消息

「acks=1」

只要 `Partition Leader` 接收到消息而且写入本地磁盘了，就认为成功了，不管它其他的 Follower 有没有同步过去这条消息了

「acks=all」

Leader将等待ISR中的所有副本确认后再做出应答，因此只要ISR中任何一个副本还存活着，这条应答过的消息就不会丢失，也即如果只有leader存活，也会出现丢失情况

④故障恢复机制

Leader选举及失败恢复机制

### 4）kafka-leader副本到follower副本数据同步机制

①副本中的高水位（High Watermark）

I定义消息的可见性，即用来标识分区下的哪些消息是可以被消费的，比如某个分区的HW（leader的HW）是8，那么这个分区只有 < 8 这些位置上消息可以被消费。

II 帮助Kafka完成副本同步。位移值等于高水位的消息也属于未提交消息。也就是说，高水位上的消息是不能被消费者消费的.。

日志末端位移，Log End Offset，缩写是LEO。它表示副本写入下一条消息的位移值。同一个副本对象，其高水位值不会大于LEO值。

数据同步时，先同步hw,然后同步leo，此时就会导致两个值不一致，如果此时宕机就会发生日志截断，导致数据丢失。

leader副本的高水位更新和follower副本高水位更新在时间上是存在一定延迟的，这样会导致数据丢失或者数据不一致。也就是如果数据还没同步完成，leader宕机，flower成为leader，这时候会根据leader的hw进行日志截断，主要是HW和LEO（log end offset，日志偏移量要对齐），就可能会导致数据丢失。

②Leader Epoch保持数据不丢失

Leader Epoch<0, 0> 和 <1, 120>会在每个节点上记录leader版本号和偏移量。每次有 Leader 变更时，新的 Leader 副本会查询这部分缓存，取出对应的 Leader Epoch 的起始位移，以避免数据丢失和不一致的情况。其实就是每次节点重启后，会检查版本号和偏移量，不发生日志截断，就不会导致数据丢失。

### 5）kafka为什么快

「顺序读写」

kafka的消息是不断追加到文件中的，这个特性使kafka可以充分利用磁盘的顺序读写性能

顺序读写不需要硬盘磁头的寻道时间，只需很少的扇区旋转时间，所以速度远快于随机读写

「零拷贝」

服务器先将文件从复制到内核空间，再复制到用户空间，最后再复制到内核空间并通过网卡发送出去，而零拷贝则是直接从内核到内核再到网卡，省去了用户空间的复制

Zero copy对应的是Linux中sendfile函数，这个函数会接受一个offsize来确定从哪里开始读取。现实中，不可能将整个文件全部发给消费者，他通过消费者传递过来的偏移量来使用零拷贝读取指定内容的数据返回给消费者

「分区」

kafka中的topic中的内容可以被分为多分partition存在,每个partition又分为多个段segment,所以每次操作都是针对一小部分做操作，很轻便，并且增加`并行操作`的能力

「批量发送」

kafka允许进行批量发送消息，producter发送消息的时候，可以将消息缓存在本地,等到了固定条件发送到kafka

1. 等消息条数到固定条数

2. 一段时间发送一次

「数据压缩」

Kafka还支持对消息集合进行压缩，Producer可以通过GZIP或Snappy格式对消息集合进行压缩。

压缩的好处就是减少传输的数据量，减轻对网络传输的压力。

Producer压缩之后，在Consumer需进行解压，虽然增加了CPU的工作，但在对大数据处理上，瓶颈在网络上而不是CPU，所以这个成本很值得

### 6）rocketMQ和kafka的区别

1.rocketMQ的同步刷盘和异步刷盘和kafaka的ack机制相似

2.rocketMQ默认需要消费者回复ack确认，如果没有确认，那就进入死信队列等候人工处理；kafaka需要手动关闭自动offset配置；

3.使用页缓存和内存映射

页缓存其实就是操作系统对文件的缓存，用来加速文件的读写，也就是说对文件的写入先写到页缓存中，操作系统会不定期刷盘（时间不可控），对文件的读会先加载到页缓存中，并且根据局部性原理还会预读临近块的内容。而内存映射也只是做了映射，只有当真正读取页面的时候产生缺页中断，才会将数据真正加载到内存中。

rocketMQ和其他的MQ对比：

• 支持事务型消息（消息发送和DB操作保持两方的最终一致性，rabbitmq和kafka不支持）

• 支持结合rocketmq的多个系统之间数据最终一致性（多方事务，二方事务是前提）

• 支持18个级别的延迟消息（rabbitmq和kafka不支持）

• 支持指定次数和时间间隔的失败消息重发（kafka不支持，rabbitmq需要手动确认）

• 支持consumer端tag过滤，减少不必要的网络传输（rabbitmq和kafka不支持）

• 支持重复消费（rabbitmq不支持，kafka支持）

### 7) rocketMQ-基本原理

RocketMQ由NameServer注册中心集群、Producer生产者集群、Consumer消费者集群和若干Broker（RocketMQ进程）组成，它的架构原理是这样的：

1. Broker在启动的时候去向所有的NameServer注册，并保持长连接，每30s发送一次心跳

2. Producer在发送消息的时候从NameServer获取Broker服务器地址，根据负载均衡算法选择一台服务器来发送消息

3. Conusmer消费消息的时候同样从NameServer获取Broker地址，然后主动拉取消息来消费

4．NameServer之间不会互相通信，都具备完整的路由信息，即无状态。所有的broker都需注册在每个NameServer中，并通过30s发送心跳进行通信。

RocketMQ主要的存储文件包括commitlog文件、consumequeue文件、indexfile文件。

CommitLog：单个 CommitLog 默认 1G，并且文件名以起始偏移量命名，固定 20 位，不足则前面补 0，这样通过偏移量就能找到文件。

ConsumeQueue 消息消费队列，可以认为是 CommitLog 中消息的索引，因为 CommitLog 是糅合了所有主题的消息，所以通过索引才能更加高效的查找消息，ConsumeQueue 对应的是一个Topic 下的某个 Queue。

IndexFile 就是索引文件，通过 Key 或者时间区间来查询对应的消息，文件名以创建时间戳命名，固定的单个 IndexFile 文件大小约为400M，一个 IndexFile 存储 2000W个索引。

Broker在收到消息之后，会把消息保存到commitlog的文件当中，而同时在分布式的存储当中，每个broker都会保存一部分topic的数据，同时，每个topic对应的messagequeue下都会生成consumequeue文件用于保存commitlog的物理位置偏移量offset，indexfile中会保存key和offset的对应关系。

由于同一个topic的消息并不是连续的存储在commitlog中，消费者如果直接从commitlog获取消息效率非常低，所以通过consumequeue保存commitlog中消息的偏移量的物理地址，这样消费者在消费的时候先从consumequeue中根据偏移量定位到具体的commitlog物理文件，然后根据一定的规则（offset和文件大小取模）在commitlog中快速定位。

### 8) **rocket**MQ为什么不使用zookeeper作为注册中心

1.zookeeper的可用性相对低，如果leader宕机，选举时间太长，会导致注册中心一段时间不可用。

2. zookeeper使用zap协议实现方案太重，会在每个节点上保留事务日志，并且不支持写横向拓展

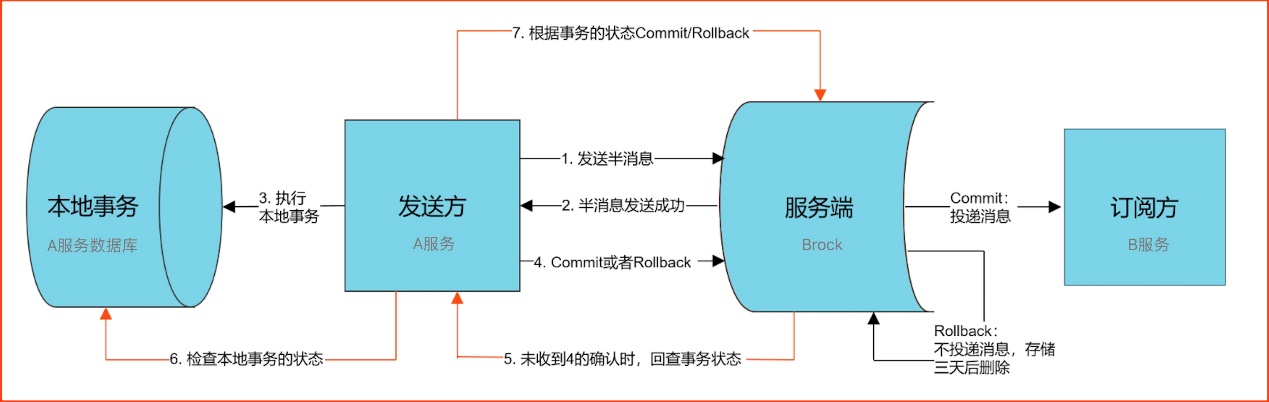
### 9）**rocket**MQ-事务消息

事务消息是针对于生产端，对于消费者端是通过消息重试&托底方案实现数据最终一致性。

`Half Message`：预处理消息，当broker收到此类消息后，会存储到`RMQ\_SYS\_TRANS\_HALF\_TOPIC`的消息消费队列中，并不是将消息投递到服务订阅的`topic`。

`检查事务状态`：Broker会开启一个定时任务，消费`RMQ\_SYS\_TRANS\_HALF\_TOPIC`队列中的消息，每次执行任务会向消息发送者确认事务执行状态（提交、回滚、未知），如果是未知，Broker会定时去回调在重新检查。

超时：如果超过回查次数，默认回滚消息。 也就是他并未真正进入Topic的queue，而是用了临时queue来放所谓的`half message`，等提交事务后才会真正的将half message转移到topic下的queue。



## 4、zookeeper

### 1) 基础结构

提供了分布式集群管理，分布式锁，分布式队列，发布订阅这样的一个能力。

实现注册中心用的节点树的一个结构，也就说第一层是APPKEY，代表是应用命名空间，第二层是一个类信息，第三层就是服务的提供者和消费者，第四层就是它的一个IP和端口号。

实现服务注册功能，那么就在服务在启动过程中会将自己的IP、端口和类信息传给zookeeper生成一个临时的znode节点。

服务的一个消费者启动之后，就会对相应命名空间下相应类的znode节点，注册监听器，并将映射关系存到本地缓存。当发起服务调用的时候，就会根据某种路由侧率从本地缓存中的路由关系去找寻相应节点，获取到相应的服务者的一个相应信息，完成调用。

服务下线、服务的扩容，都是相同的原理。

### 

### 2）Zab 协议中的 Zxid

Zxid 是 Zab 协议的一个64 位的事务编号，其中低 32 位是一个简单的单调递增计数器，针对客户端每一个事务请求，计数器加 1；而高 32 位则代表 Leader 的epoch周期年代的编号。

每当有一个新的 Leader 选举出现时，就会从这个 Leader 服务器上取出其本地日志中最大事务的 Zxid，并从中读取 epoch 值，然后加 1，以此作为新的周期 ID。高 32 位代表了每代 Leader 的唯一性，低 32 位则代表了每代 Leader 中事务的唯一性。

### 3) Zab 协议来保证分布式事务的最终一致性

**消息广播阶段**

在 ZooKeeper 中所有的事务请求都由 Leader 节点来处理，其他服务器为 Follower，Leader 将客户端的事务请求转换为事务 Proposal，并且将 Proposal 分发给集群中其他所有的 Follower。

完成广播之后，Leader 等待 Follwer 反馈，当有过半数的 Follower 反馈信息后，Leader 将再次向集群内 Follower 广播 Commit 信息，Commit 信息就是确认将之前的 Proposal 提交。

Leader 节点的写入也是一个两步操作，第一步是广播事务操作，第二步是广播提交操作，其中 过半数指的是反馈的节点数 >=N/2+1，N 是全部的 Follower 节点数量。

**崩溃恢复阶段**

下面的几种情况都会进入崩溃恢复阶段：

- 初始化集群，刚刚启动的时候

- Leader 崩溃，因为故障宕机

- Leader 失去了半数的机器支持，与集群中超过一半的节点断连

崩溃恢复模式将会开启新的一轮选举，选举产生的 Leader 会与过半的 Follower 进行同步，使数据一致，当与过半的机器同步完成后，就退出恢复模式， 然后进入消息广播模式。

选举过程如下：

1.各个节点变更状态，变更为 Looking

- ZooKeeper 中除了 Leader 和 Follower，还有 Observer 节点，Observer 不参与选举， Leader 挂后，余下的 Follower 节点都会将自己的状态变更为 Looking，然后开始进入 Leader 选举过程。

2.各个 Server 节点都会发出一个投票，参与选举

- 在第一次投票中，所有的 Server 都会投自己，然后各自将投票发送给集群中所有机器，在运行期间，每个服务器上的 Zxid 大概率不同。

3.集群接收来自各个服务器的投票，开始处理投票和选举

- 处理投票的过程就是对比 Zxid 的过程，假定 Server3 的 Zxid 最大，Server1判断Server3可以成为 Leader，那么Server1就投票给Server3，判断的依据如下：首先选举 epoch 最大的，如果 epoch 相等，则选 zxid 最大的，若 epoch 和 zxid 都相等，则选择 server id 最大的，就是配置 zoo.cfg 中的 myid；在选举过程中，如果有节点获得超过半数的投票数，则会成为 Leader 节点，反之则重新投票选举。

**数据同步**

在选举过程中，通过投票已经确认 Leader 服务器是最大Zxid 的节点，同步阶段就是利用 Leader 前一阶段获得的最新Proposal历史，同步集群中所有的副本。

### 4) Zab 与 Paxos 算法的联系与区别

Zab 协议可以认为是基于 Paxos 算法实现的，先来看下两者之间的联系：

- 都存在一个 Leader 进程的角色，负责协调多个 Follower 进程的运行

- 都应用 Quorum 机制，Leader 进程都会等待超过半数的 Follower 做出正确的反馈后，才会将一个提案进行提交

- 在 Zab 协议中，Zxid 中通过 epoch 来代表当前 Leader 周期，在 Paxos 算法中，同样存在这样一个标识，叫做 Ballot Number

两者之间的区别是，Paxos 是理论，Zab 是实践，Paxos 是论文性质的，目的是设计一种通用的分布式一致性算法，而 Zab 协议应用在 ZooKeeper 中，是一个特别设计的崩溃可恢复的原子消息广播算法。

Zab 协议增加了崩溃恢复的功能，当 Leader 服务器不可用，或者已经半数以上节点失去联系时，ZooKeeper 会进入恢复模式选举新的 Leader 服务器，使集群达到一个一致的状态。

## 5、rpc

### 1）rmi和rpc的区别

rmi是远程方法调用，RPC的是远程过程调用，rmi基于java版本的一个RPC框架。它是利用的Java原生的serializable接口实现一个序列化和反序列化，只能适用于java版并且序列化和反序列化效率较低，不满足于高性能要求。

### 2）如何实现简单的rpc

实现一个简单的RPC框架，基本上有这么几个步骤，一个是需要客户端和服务端，然后客户端和服务端需要进行数据通信，那么数据通信的时候需要序列化反序列化，序列化反序列化分为两种实现机制，一种是文本，使用XML或者json，另一种就是二进制， thrift或者Protobuff、msg、hession之类的一些框架去做。区别在于序列化反序列化效率，以及序列化后码流大小是有区别的。数据传输模块的话，一般使用socket，或者说使用更加成熟的框架netty进行一个数据传输 。数据传送过来之后，一般需要proxy去做一些代理，屏蔽一些底层数据传输以及数据转换的一些复杂逻辑。

### 3）为什么实现序列化

序列化和反序列化，就是将对象和字节码进行互相转换的过程。那么，使用序列化的原因就是不共享内存的系统能通过网络连接进行一个对象传输。JavaBean->IDL（接口定义语言）文件->byte[]--socket--byte[]->IDL（接口定义语言）文件->JavaBean.

## 6、mybatis

①分页原理

②如何防止sql注入

③一二级缓存

# 四、分布式

## 1）CAP和BASE理论

**CAP理论**

C表示的是一致性，A是可用性，P分区容错性

P是表示硬件出现问题，比如网络异常，而不是指应用出现问题。

一般认为认为CAP3种状态是无法同时保持，也就是只能保持CP和AP

CP：保持数据一致性和分区容错性，比如说AB两个节点之间进行数据复制，如果是在P发生异常的情况下，比如说网络不通，那么在此过程中，AB2个节点之间，它的数据是不一致的，此时客户端访问的话，只能返回数据错误

AP：保持服务可用性和分区容错性，在上述流程中，即便发生问题，那么应该把旧的数据返回回去，保持服务可用。

**base理论**

即基本可用、软状态和最终数据一致性

基本可用的就是说，分布式系统在发生故障的时候，允许非核心部分失去功能，但是仍然保持核心功能可用，比如说登录注册，像注册的这种非核心功能，就可以出现一些问题，体现了功能解耦思想。

软状态，就是允许出现中间状态，即出现数据不一致的状态。

最终一致性，即保持数据最终一致。

## 2）分布式事务算法-2PC

分布式事务算法分为2PC、3PC

**2PC：二阶段提交**

它分为协调者和参与者两个角色，协调者和参与者之间是可以进行一个网络通讯的，所有节点都采用预写式日志的方法，且日志保存在存储设备。

2PC由两个阶段组成，一是提交请求阶段，又称为投票阶段，二是提交执行阶段

1）提交请求阶段：协调者向所有的参与者发送请求提交消息，寻味是否可以提交事务，并等待各个参与者的响应，然后参与者执行询问发起为止的所有的事物操作，并将undo信息写入日志。如果执行成功，则参与着发送yes给协调者，如果发送失败，则发送NO.

2)提交执行阶段：若所有参与者都回答的YES，则协调者向所有的参与者发送commit消息，所有参与者完成commit操作，并释放整个事物所占用的资源。最后参与者向协调者发送ack，完成整个事物。

如果参与者发送NO或者说任意一个参与者在超时时间内没有回馈的话，则协调者会向所有的参与者发送rollBack请求。所有的参与者数据进行回滚，然后并释放整个事物过程中占用的资源。

最后参与者向协调者发送ACK信息，协调者收到所有的ACK信息之后，取消整个事物。

**2PC 优点/缺点**

优点是实现比较简单，缺点的时候也非常明显：

1）同步阻塞，性能比较差。事务执行过程中，等待过程是堵塞状态，不能做其他事情，如果某个节点反应比较慢的话，就会拖垮整个系统。

2）可能出现状态不一致问题。在第二阶段执行过程中，如果协调者发出commit后，某个参与者没有收到这份消息，其他的参与者收到这个消息了，则收到消息的参与者会提交事务，而未收到的一个参与者会超时回滚事务，那么就导致一个事物状态不一致。虽然参与者最终还会进行数据回滚，但在这个过程中会出现数据不一致。

3）单点故障问题。协调者是整个算法的单点，如果协调者在第二阶段发生问题的话，那么参与者就会一直阻塞下去，如果协调者因为故障不能正常发送请求或者回复通知的话，那么就一直处于阻塞状态，整个集群就无法提供服务

## 3）分布式事务算法-3PC

**3PC算法**

是针对二阶段提交算法单点故障而提出的解决方案，在2PC算法中，在第一阶段和第二阶段之间插入一个新的阶段，准备阶段。当协调者发生故障之后，参与者可以通过超时时间来避免一直阻塞。

第一阶段（提交判断阶段）

1）协调者向参与者向参与者发送canCommit消息，然后询问参与者是否可以提交事务

2）参与者收到canCommit消息之后，判断是否能够提交事务，如果可以执行事务，则返回YES,否则返回No。

3）如果协调者收到任何一个no，或者是参与者执行超时就终止事物，否则的话就会进行第二阶段

第二阶段（准备提交阶段）

1）协调者发送preCommit给所有的参与者，然后通知参与者准备提交

2）参与者收到preCommit消息之后，开始执行实事务操作，记录undoLog，然后返回ACK信息

第三阶段（提交执行阶段）

1)协调者收到所有的ACK信息之后，会发送doCommit，告诉参与者正式提交

2)参与者收到doCommit消息之后提交事务，然后返回一个haveCommitted消息

3)如果参与者收到一个preCommit的消息之后，并返回ACK，但等待doCommit的时候消息超时，参与者会在超时后自动提交事务

3PC虽然避免了2PC中协调者单点故障导致的系统阻塞问题，但是没有解决数据不一致问题。

## 4）分布式一致性算法-paxos\raft\zab

分布式事务算法主要保证分散在多个节点上的数据统一的提交或回滚，为了满足ACID原则。

分布式一致性算法是为了保证同一份数据在多个节点之上数据的一致性，满足的是CAP中的CP原则。

## 5）接口故障应对方案

接口故障应对的方案包括降级、熔断、限流、排队

1）降级指系统中某些业务接口的功能降低，只提供部分功能或全部停掉,主要是对于系统本身的故障，采用降级方案。

2）熔断指应对依赖外部系统的故障。比如说A调用B系统，B系统发生问题，如果A一直调用B，则可能导致A被拖垮

熔断机制一般就是设定一个阀值，比如响应时间超过x,就执行托底方案。

3）限流指只允许系统能够承受的访问量进来，超出系统承受能力则被限制、

分为基于请求的限流，基于资源的限流两种方案：

基于请求的限流：一般就是限制总量，或者限制时间量

基于资源限流，一般和硬件有关系，根据cpu、磁盘等占用多少，然后进行限流。

4）排队是限流的变种，限流是直接拒绝接收，而排队的话就让长时间进行等待

## 6）Eureka

**1、客户端注册到服务端**

Eureka客户端在启动时，首先会创建一个心跳的定时任务，定时向服务端发送心跳信息，服务端会对客户端心跳做出响应，如果响应状态码为404时，表示服务端没有该客户端的服务信息，那么客户端则会向服务端发送注册请求，注册信息包括服务名、ip、端口、唯一实例ID等信息。

**2、服务端如何保存客户端服务信息**

客户端将服务实例信息发送到服务端，服务端将客户端信息放在一个ConcurrentHashMap对象中。

**3、客户端拉取服务端已保存的服务信息**

客户端拉取服务端服务信息是通过一个定时任务定时拉取的，每次拉取后刷新本地已保存的信息，需要使用时直接从本地直接获取。

**4、构建高可用的Eureka集群**

搭建高可用的Eureka集群，只需要在注册中心的配置文件中配置其他注册中心的地址

注册中心收到注册信息后会判断是否是其他注册中心同步的信息还是客户端注册的信息，如果是客户端注册的信息，那么他将会将该客户端信息同步到其他注册中心去；否则收到信息后不作任何操作。通过此机制避免集群中信息同步的死循环。

一级缓存：registry，类型ConcurrentHashMap 实时更新，又名注册表，UI界面从这里获取服务注册信息；

三级缓存策略保持服务高并发可用：

readWriteCacheMap

二级缓存：readOnlyCacheMap，类型Guava Cache（LoadingCache）实时更新，缓存时间180秒；

三级缓存：readOnlyCacheMap，类型ConcurrentHashMap，周期更新，默认每30s从二级缓存readWriteCacheMap中同步数据更新；

Eureka client第一次向Eureka Server注册服务或者发送心跳续约时，往其registry字段中添加服务注册信息，并会将其二级缓存appName下ip地址清空/失效；

使用定时器进行一级-》二级-》三级缓存同步：

会设置初始化二级缓存readWriteCacheMap（过期时间180s），设置二级缓存往三级缓存readOnlyCacheMap同步的时间间隔（默认30s）。

设置是否使用三级缓存（默认使用），如果使用则启动一个定时任务，默认每隔30s从二级缓存中同步数据到三级缓存（只更新三级缓存中已存在的key）；

**5、心跳和服务剔除机制**

心跳机制：

客户端启动后，就会启动一个定时任务，定时向服务端发送心跳数据，告知服务端自己还活着，默认的心跳时间间隔是30秒。

服务剔除机制：

如果开启了自我保护机制，那么所有的服务，包括长时间没有收到心跳的服务（即已过期的服务）都不会被剔除；

如果未开启自我保护机制，那么将判断最后一分钟收到的心跳数与一分钟收到心跳数临界值（计算方法参考5.1节）比较，如果前者大于后者，且后者大于0的话，则启用服务剔除机制；

一旦服务剔除机制开启，则Eureka服务端并不会直接剔除所有已过期的服务，而是通过随机数的方式进行剔除，避免自我保护开启之前将所有的服务（包括正常的服务）给剔除。

**6、Eureka自我保护机制**

当网络发生分区时，客户端和服务端的通讯将会终止，那么服务端在一定的时间内将收不到大部分的客户端的一个心跳，如果这个时候将这些收不到心跳的服务剔除，那可能会将可用的客户端剔除了，这就不符合AP理论。

①    注册发现

优点：

1）提供透明化服务路由，也就是说服务调用者和服务提供者之间互相解耦，不需要知道对方的一个IP和端口都可能实现数据通讯。2) 提供横向拓展能力，也就说无论机器的增减都不影响正常使用。3）那可以动态的监控依赖，服务运行状况，为服务治理提供基础。

②    负载均衡

③    限流、容灾

④    幂等

⑤    Cap理论

⑥    服务治理（服务依赖梳理、负载均衡、服务分组灰度发布、链路监控、服务质量统计、服务注册发现、服务下线）

# 五、算法

①    判断链表是否含环

②    旋转链表

③    判断树是否对称

④    topK

⑤    数组中出现次数超过一半的数字

⑥    排序-快排、冒泡

六、其他

①    TCP/IP

② bio\nio\aio

BIO和NIO区别就是NIO主要做了几大改变，一是使用buffer缓冲区，BIO使用stream流，第二就是channel通道，全双工，可以同时进行读写，而stream流是单向的，第三就是selector选择器，channer注册selector上面，然后不断地进行轮询，监听Channel的读写变化

AIO,异步IO

③非阻塞和异步区别

非阻塞其实就是调用发起一个调用请求的时候，当前线程不用等待立马返回。但是调用方采用的是轮询的手段去获取返回结果

异步模型，调用发起一个调用请求的时候，当前线程不用等待立马返回，等到真正处理完后，被调用方通过回调方式去将结果返回。

区别就是非阻塞，主动的去判断,异步是被调用方主动通知。

IO复用模型，介于非阻塞io和异步IO之间，在调用方调用时候是将线程阻塞在select模型，然后由select模型轮询数据是否准备完毕，如果准备完毕后使用回调将数据返回。

④netty