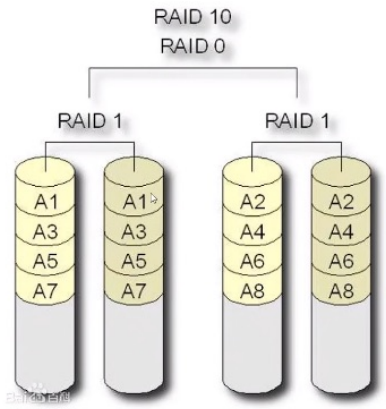
91.MQ的存储介质损坏，该如何处理？

单机可使用RAID10方案存储。整个RAID0包含了所有数据，每一块数据又有相同的备份。



RAID0是将所有数据分散存储到两块不同的硬盘上。每个RAID1硬盘大小性能相同，两块硬盘数据完全相同，一块一盘坏了可以去另一块补上。

92.本地缓存和分布式缓存优缺点？

缓存：

在服务端编程当中，缓存主要是指将数据库的数据加载到内存中，之后对该数据的访问都在内存中完成，从而减少了对数据库的访问。

以及基于内存的访问速度高于磁盘的访问速度的原理，提高了数据的访问速度和程序性能。

根据缓存是否与应用进程属于同一进程，可以将内存分为本地缓存和分布式缓存。

本地缓存是在同一个进程内的内存空间中缓存数据，数据读写都是在同一个进程内完成；

而分布式缓存是一个独立部署的进程并且一般都是与应用进程部署在不同的机器，故需要通过网络来完成分布式缓存数据读写操作的数据传输。

I、本地缓存的优缺点：

1. 访问速度快，但无法进行大数据存储

本地缓存相对于分布式缓存的好处是，由于数据不需要跨网络传输，故性能更好，但是由于占用了应用进程的内存空间，

如 Java 进程的 JVM 内存空间，故不能进行大数据量的数据存储。

2. 集群的数据更新问题

与此同时，本地缓存只支持被该应用进程访问，一般无法被其他应用进程访问，故在应用进程的集群部署当中，如果对应的数据库数据，

存在数据更新，则需要同步更新不同部署节点的本地缓存的数据来包保证数据一致性，复杂度较高并且容易出错，如基于 Redis 的发布订阅机制来同步更新各个部署节点。

3. 数据随应用进程的重启而丢失

由于本地缓存的数据是存储在应用进程的内存空间的，所以当应用进程重启时，本地缓存的数据会丢失。

所以对于需要持久化的数据，需要注意及时保存，否则可能会造成数据丢失。

适用场景：

所以本地缓存一般适合于缓存只读数据，如统计类数据。或者每个部署节点独立的数据，如长连接服务中，每个部署节点由于都是维护了不同的连接，

每个连接的数据都是独立的，并且随着连接的断开而删除。如果数据在集群的不同部署节点需要共享和保持一致，则需要使用分布式缓存来统一存储，

实现应用集群的所有应用进程都在该统一的分布式缓存中进行数据存取即可。

本地缓存的实现

缓存一般是一种key-value的键值对数据结构，所以需要使用字典数据结构来实现，与此同时，本地缓存由于需要被不同的服务端线程并发读写，故需要保证线程安全。

故一般会使用 ConcurrentHashMap 来作为 Java 编程中的本地缓存实现。除此之外，也有其他更加智能的本地缓存实现，如可以定时失效，访问重新加载等特性，

典型实现包括 Google 的 guava 工具包的 Cache 实现，这些也是线程安全的。

II、分布式缓存的优缺点

1. 支持大数据量存储，不受应用进程重启影响

分布式缓存由于是独立部署的进程，拥有自身独立的内存空间，不会受到应用进程重启的影响，在应用进程重启时，分布式缓存的数据依然存在。

同时对于数据量而言，由于不需要占用应用进程的内存空间，并且一般支持以集群的方式拓展，故可以进行大数据量的数据缓存。

2. 数据集中存储，保证数据一致性

当应用进程采用集群方式部署时，集群的每个部署节点都通过一个统一的分布式缓存进行数据存取操作，故不存在本地缓存中的数据更新问题，保证了不同节点的应用进程的数据一致性问题。

3. 数据读写分离，高性能，高可用

分布式缓存一般支持数据副本机制，可以实现读写分离，故可以解决高并发场景中的数据读写性能问题。并且由于在多个缓存节点冗余存储数据，提高了缓存数据的可用性。

3. 数据跨网络传输，性能低于本地缓存

由于分布式缓存是独立部署的进程，并且一般都是与应用进程位于不同的机器，故需要通过网络来进行数据传输，这样相对于本地缓存的进程内部的数据读取操作，性能会较低。

分布式缓存的实现

分布式缓存的典型实现包括 MemCached 和 Redis。

MemCached

MemCached 相对于本地缓存的主要差别是以独立进程方式存在，数据集中存储，数据不随应用程序的重启而丢失。

而 key-value 键值对的 value 也是一个简单的对象类型。

所以 MemCached 进程相当于是在内存维护了一个非常大的哈希表来存储数据，对应的数据操作复杂度都是 O(1)，即常量级别，这也是 MemCached 高性能的一个实现方式，键值对存取速度都非常快。

Redis

Redis，更一步丰富了数据结构类型。并且 Redis 是单线程的，不存在并发数据读写的线程安全问题，以及更重要的是保证的数据读写操作的顺序性。

除此之外，Redis 支持主从同步（读写分离）、集群分片拓展、数据持久化等特性，这也是 MemCached 不支持的。

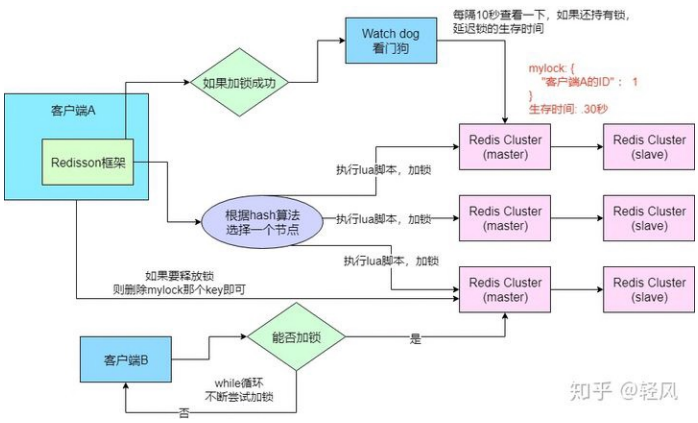
所以在高并发场景并且数据能够容忍极端情况下的少量丢失，或者说丢失后可以恢复，如通过日志或者重新计算等， Redis 也可以作为数据库来使用，提高高并发场景中的访问性能。

93.Redis分布式锁的实现原理？

```
RLock lock = redisson.getLock("myLock");
lock.lock();
lock.unlock();
```

知乎 @轻风

目前基于Redis实现的分布式锁常用的框架是Redisson,它的使用比较简单，在项目中引入Redisson的依赖，然后基于Redis实现分布式锁的加锁与释放锁。



实现原理：

1.客户端首先根据hash结点选择一台机器，紧接着就会发送一段lua脚本到redis上。

2.lua脚本，把一大堆业务逻辑通过封装在lua脚本发送给redis，保证这段赋值业务逻辑执行的原子性。

3.watch dog看门狗，是一个后台线程，会每隔10s检查一下，如果客户端A还持有锁key，那么就会不断延长锁key的生存时间。

4.解锁时，执行lock.unlock()，每次对锁计数减一，如果锁计数为零了，就会使用“del MyLock”命令，从redis里删除key。

5.客户端B尝试加锁，第一个if判断锁是否存在，第二个if判断锁id是否是自己，加锁失败就会进入自旋状态。

6.缺点：

异步数据丢失问题：客户端A对redis master实例写入锁，此时会异步复制给对应的master slave实例，这个过程中如果redis master宕机，

主备切换，redis slave变成了新的redis master。这时候客户端B来尝试加锁，在新的redis master上完成了加锁，但是客户端A也自以为成功加锁。

此时就会导致多个客户端对一个分布式锁完成了加锁，这时就会导致各种脏数据的产生。

解决：

红锁（Redlock）：客户端用相同的key和随机值在所有节点上请求锁，请求锁的超时时间应小于锁自动释放时间。

超过半数N/2+1，才算是真正获取到了锁。这样就算其中某个redis节点挂掉，锁也不会被其他客户端获取到。

如果没有获取到锁，则把部分已锁的redis释放掉。

94.限流算法？

1、漏桶算法

往漏斗里面倒水，不论倒多少水，下面出水的速率是恒定的。当漏斗满了，多余的水就被直接丢弃了。

类比流量，每秒处理的速率是恒定的，如果有大量的流量过来就先放到漏斗里面。当漏斗也满了，请求则被丢弃

2、令牌桶算法

1）、所有的请求在处理之前都需要拿到一个可用的令牌才会被处理；

2）、根据限流大小，设置按照一定的速率往桶里添加令牌；

3）、桶设置最大的放置令牌限制，当桶满时、新添加的令牌就被丢弃或者拒绝；

4）、请求达到后首先要获取令牌桶中的令牌，拿着令牌才可以进行其他的业务逻辑，处理完业务逻辑之后，将令牌直接删除；

5）、因为桶中有存的令牌，所以令牌桶算法支持突发流量。

对比：

漏桶算法：请求都按固定的速度取走处理，常用于网络的流量整流。

令牌桶算法：token放入速度是确定的，如果桶中存有token支持突发流量。

实现：用ScheduledThreadPoolExecutor来定时从队列中取请求来执行/定时放令牌

95.spring IOC的底层实现

反射，工厂。

1、先通过creatBeanFactory创建出一个Bean工厂（DefaultListableBeanFactory）

2、开始循环创建对象，容器中bean默认都是单例的，所以优先通过getBean、doGetBean从容器中查找

3、若查不到，通过creatBean，doCreatBean的方法，以反射的方式创建对象，一般情况使用无参构造方法（getDeclaredConstructor.newInstance）

4、进行对象的属性填充populateBean

5、进行其他的初始化操作（initializingBean）

96.Spring事务底层如何实现？

通过aop生成代理对象，通过代理对象生成事务拦截链TransactionInterceptor，调用invoke实现具体逻辑。

```
1.根据事务传播等级等属性来判断是否开启事务。
2.获取数据库连接，关闭自动提交，开启事务。
3.执行具体的sql
4.如果执行失败，通过completeTransationAfterThrowing来完成事务回滚，具体逻辑是通过
doRollBack实现，先获取连接对象，通过连接对象回滚
5.如果执行正常，通过commitTransactionAfterReturning来完成事务提交，具体逻辑是通过
doCommit实现，先获取连接对象，通过连接对象提交
```

97.一次请求涉及的网络协议

一、发请求：

网址----------（IP）----------DNS

↓

请求----------HTTP/HTTPS----------进程(tomcat)--------RPC---------专门的进程

↓ ↑

传输----------TCP----------传输层（tcp会确认回复，不行就ARQ）

↓ ↑

网络----------IP----------网络层

↓ ↑

网关（默认192.108.1.1）----------ARP协议获取MAC----------目标服务器

二、传输 ↑

网关----------路由器，路由协议（RIP/OSPF/BGP）----------目标网关（会在局域网广播）

98.CSRF攻击？

原理：攻击者引诱用户访问第三方页面B，这个页面B有请求被攻击网站A的URL，利用浏览器默认携带网站A的cookie，进而发起跨站请求。

防御: 要抵御 CSRF，关键在于在请求中放入黑客所不能伪造的信息。

因为伪造网站无法读取cookie，所以用户请求参数中带上cookie的值作为CSRF_token。

1. 服务端生成一个随机的，不可预测的Token，放在用户的Session中，浏览器的Cookie中。

2. 用户在页面表单附带上Token参数。

3. 用户提交请求后， 服务端验证表单中的Token是否与用户Session中的Token一致，一致为合法请求，不是则非法请求。

99.HTTPS的混合加密？会有什么问题？

1某网站拥有用于非对称加密的公钥A、私钥A’。

2浏览器向网站服务器请求，服务器把公钥A明文给传输浏览器。

3浏览器随机生成一个用于对称加密的密钥X，用公钥A加密后传给服务器。

4服务器拿到后用私钥A’解密得到密钥X。

5这样双方就都拥有密钥X了，且别人无法知道它。之后双方所有数据都通过密钥X加密解密即可。

中间人攻击：

如果在数据传输过程中，中间人劫持到了数据，此时他的确无法得到浏览器生成的密钥X，这个密钥本身被公钥A加密了，

只有服务器才有私钥A’解开它，然而中间人却完全不需要拿到私钥A’就能干坏事了。请看：

1某网站有用于非对称加密的公钥A、私钥A’。

2浏览器向网站服务器请求，服务器把公钥A明文给传输浏览器。

3中间人劫持到公钥A，保存下来，把数据包中的公钥A替换成自己伪造的公钥B（它当然也拥有公钥B对应的私钥B’）。

4浏览器生成一个用于对称加密的密钥X，用公钥B（浏览器无法得知公钥被替换了）加密后传给服务器。

5中间人劫持后用私钥B’解密得到密钥X，再用公钥A加密后传给服务器。

6服务器拿到后用私钥A’解密得到密钥X。

这样在双方都不会发现异常的情况下，中间人掉包了服务器传来的公钥，进而得到了密钥X。

根本原因是浏览器无法确认收到的公钥是不是网站自己的，因为公钥本身是明文传输的，解决办法是CA数字证书。

100.CA怎么解决中间人攻击问题？

网站在使用HTTPS前，需要向CA机构申领一份数字证书，数字证书里含有证书持有者信息、公钥信息等。

服务器把证书传输给浏览器，浏览器从证书里获取公钥就行了，证书就如身份证，证明“该公钥对应该网站”。

而这里又有一个显而易见的问题，“证书本身的传输过程中，如何防止被篡改”？数字证书怎么防伪呢？解决这个问题我们就接近胜利了！

我们把证书原本的内容散列生成一份“数字签名”，比对证书内容和签名是否一致就能判别是否被篡改。这就是数字证书的“防伪技术”。