Mysql篇

1. mysql数据库哈希索引B+树索引：
   1. 时间复杂度：哈希索引是O（1）因为底层是哈希表。B+树是O（logn），底层是B+树。所以查询的话哈希索引是比B+树快的，
   2. 实现方式：哈希索引是**对索引列计算哈希值，然后将记录映射到哈希槽中。**B+树索引:是通过比较索引列值大小，然后将**数据记录在B+树的叶子节点（B+树只有叶子节点才记录信息）**，并且还可以直接遍历叶子节点，例如链表的方式遍历，所以**B+树索引支持范围查询。**
   3. 优缺点：

B+树优点：

* + - 1. **支持范围查询**
      2. **查询删除等操作效率稳定都是O（logn）。**
      3. **读取时，内存单次从磁盘加载的索引节点会更多，因为只有叶子节点才存储数据，其他节点不存储数据。IO效率更高。**

B+树缺点：

1. **高并发情况下整树的结构（产生页分裂）**

哈希索引优点：

1.**查询时，效率非常高。**

哈希索引的缺点**：**

1. **不能进行模糊查询、范围查询。**
2. **若经常发生哈希冲突的话，查询效率会大大下降。**
   1. 新型索引结构：**LSM-Tree（日志结构合并树）**很多中间件都使用的一种索引结构，大量使用在分布式的数据库中如HBase。 Kafka中也有。 LSM-Tree索引的写吞吐量很高， 顺序写入日志的方式。还会合并索引相同的日志。
3. 索引失效场景：
   1. 不满足最左前缀原则：
      1. where条件后的查询字段，最好和**复合索引的顺序一致**（虽然有时不一致，引擎也会优化）。
      2. 如果**复合索引出现断层**，那么断层前的字段还是会走索引，断层后的字段是不会走索引的直接失效了。
   2. 索引列上有计算或者聚合函数：
      1. Where查询条件后的字段出现了计算、或者出现了聚合函数（Max、sum），索引也会失效。
   3. 模糊查询like左边出现%符号，也会导致索引失效。
4. 行锁是如何实现的（以innodb为例）
   1. 是通过给索引上的索引项加锁来实现的，**只有通过索引条件来检索数据，innodb才会使用行锁，否则使用表锁。**
5. 什么是聚簇索引和非聚簇索引
   1. 聚簇索引：单来说就是**主键索引。**非聚簇索引：就是辅助索引、复合索引，就是其他列组成的索引。
   2. 相同点：都是使用B+树

不同点：**主键索引的B+树的叶子节点一行完整的数据，而非主键索引的B+树的叶子节点存储的是不完整的数据（只有索引列的数据）,且叶子节点还存储了主键索引的Key（便于回表）。**

1. 什么是回表，如何避免
   1. 什么是回表：select查询的数据不在where条件使用的索引中，所以当根据索引查询到数据后，要根据主键索引key回到聚簇索引的b+树中回表查询。
   2. 如何避免：使用覆盖索引，覆盖索引就是指复合索引中包含查询的所有字段，这样就不需要回表了。
2. Mysql的MVCC机制是如何实现的
   1. MVCC是什么？
      1. 多版本并发控制，mvcc在innodb中主要是为了**提高数据库并发效率，用更好的方式解决读-写问题，实现读-写冲突时，也不需要加锁。MVCC 就是为了实现读-写冲突不加锁，而这个读指的就是快照读**
   2. 什么是**当前读**和**快照读**
      1. 当前读：就是读取最新的数据，通过加锁的方式读取（selec lock in share mode、select for update），**读取时保证其他事务不能修改当前正在读取的记录，所以通过加锁的方式读取**
      2. 快照读：即不加锁的读取，**前提是隔离级别不能是串行化。**Innodb实现快照读的方式就是多版本并发控制（mvcc）。避免了加锁，降低了开销，但是读取的数据不一定是最新的数据，而可能是历史版本。
   3. MVCC实现方式：**三个隐藏字段、undo日志、rede view**
      1. 三个隐藏字段：回滚指针（某条undo日志的上一条日志位置）、本条undo日志的事务id（DB\_TRX\_ID）、隐式主键
      2. Undo日志：某事务对某条记录修改后（update），就会产生一条日志，存在于undo log日志链表中通过回滚指针连接。
      3. Read view：就是当事务执行快照读时产生的一个读视图，记录并维护当前活跃的事务id（事务id遵循自增的规律，最新的事务id最大）。
   4. 如何通过read view判断可见性
      1. 获得需要读取的记录的最新被修改的事务id（DB\_TRX\_ID）
      2. 如果DB\_TRX\_ID小于read view中正在活跃的事务的最小id，那么说明需要读取最新的记录早就提交了，是可见的，反之进入下一步。
      3. 如果DB\_TRX\_ID大于了read view中正在活跃的事务的最大id，说明该需要读取最新的记录是在**发生快照读后才修改的，所以是不可见的。需要通过undo log查找旧记录的**DB\_TRX\_ID**重新开始判断。知道可见**
      4. 最后判断DB\_TRX\_ID是否在read view活跃的事务id中**，如果不在那么是可见的，如果在的话那就是不可见的，那么同样需要通过undo log查找就旧记录的**DB\_TRX\_ID
   5. MVCC机制下的RC（read commited）、RR（read repeatable）两个隔离级别是如何实现的
      1. RC：某个事务每次读取同一条记录都会重新生成一个快照读（read view）。同一个事务中，每次同一条记录时，读取的记录可能不一样
      2. RR:某个事务只会在第一次读取同一条记录时生成一个快照读（read view），以后都是使用这个read view。同一个事务中，每次读取同一个记录都是一样的
3. Mysql是如何实现事务的
   1. 通过四个特性来实现事务的
      1. 原子性：同一个事务的操作要么同时成功要么同时失败回滚
      2. 一致性：事务执行后，数据库的一致性为发生改变
      3. 持久性：一旦事务提交完成，就会被持久化到磁盘中
      4. 隔离性：事务和事务之间是隔离的，互不干扰的
         1. 通过锁来实现
         2. **读-写通过mvcc**来实现，可以不加锁的情况下实现事务间隔离。
4. Inoodb和Myisam两个引擎的区别
   1. Inoodb支持事务，而Myisam不支持事务。如果考虑事务的话，首选Inoodb，相反考虑查询速度的话，Myisam比Inoodb快。
   2. Inoodb支持行锁、间隙锁、页锁、表锁等，**而mysiam只支持表锁。**
   3. Inoodb支持聚簇索引和非聚簇索引，**数据和索引在一个文件中的。**而myisam只支持非聚簇索引，且数据和索引是分开两个文件的，索引文件存放指向数据的指针。
   4. **Inoodb每条数据必须有主键，如果没有指定也会生成默认的主键。**相反myisam可以没有主键。
   5. Inoodb不支持全文检索，而myisam支持全文检索。
5. 隔离级别和解决的问题
   1. 问题
      1. 脏写：某个事务修改了数据还未提交，另一个事务修改了并提交了，先前的事务再提交，那么第二个事务的修改被覆盖了。 （写-写发生的冲突）
      2. 脏读：某个事务修改了数据，没有提交，另一个事务读取了，然后先前那个事务又修改了，再提交，那么读取的数据就是错误的。
      3. 不可重复读：某个事务读取了某一条记录，然后又来读取这条记录发现两次读取不一样。
      4. 幻读：幻读和不可重复读很想，幻读是读取一个区间内的数据，然后又读取这个区间的数据，发现前后两次读取的数量不同。
   2. 隔离级别：
      1. 读未提交：写-写问题解决了，解决了：脏写
      2. 读已提交：只能读取提交了的数据，解决了：脏写、脏读
      3. 可重复读：同一个事务中，多次读取同一条数据结果是一样的（靠mvcc解决的），解决了：脏写、脏读、不可重复读
      4. 可串行化：加锁，解决了所有问题 。
6. 什么是分库分表
   1. 由于数据量越来越大，单个库无法承受，单张表数据量太大无法承受，需要才分多个表和库出来。
   2. 分表
      1. 垂直分表：将原来表的**数据列拆分为开**，分别保存在多张表上。**每张表的数据列不相同。表结构变了**
      2. 水平分表：将一张很大的表，才分为多张表保存，表结构不变。
   3. 分库
      1. 垂直分库：将原来的库的模块才分为多个模块保存在不同的库中，库结构变了。优点：**业务逻辑更清晰**
      2. 水平分库：一般和水平分表结合，库结构不变。用多个库保存原来库中数据，
7. Mysql有哪些日志
   1. Bin log
      1. Binlog日志，记录的是对mysql写操作，以二进制的方式记录的。且每种数据库引擎都支持，binlog是server层面的，不是数据库引擎层面的。
      2. Binlog日志的作用：
         1. 在mysql主从复制集群中，master服务器会将binlog日志同步slave服务器，达到主从一致。
         2. Canal可以手机mysql的binlog日志，然后异步通知redis与mysql达到数据一致性。
   2. Redo log
      1. Redo log日志，记录了事务对数据做了哪些修改。**只有innodb才有redo log**
      2. Redo log作用
         1. 它是一种Write-Ahead Logging（写前日志），主要是保证了数据库的**持久性（事务对数据库修改保存后，不能回到原来的状态）。**Redo log就可以保证，一旦事务提交后，数据库状态就会确定下来。**主要用于崩溃回复**
         2. Redo log有两个部分：
            1. **Redo log buffer：内存的日志缓存**
            2. **Redo logfile：磁盘日志文件**
         3. 对数据库的操作一旦保存，机会先将操作保存到缓存中，最后写到磁盘。
   3. Undo log
      1. 重做日志，主要是保存了对某一条数据历史版本
      2. Undolog作用
         1. 保存**原子性，一个事务期间的操作，要么全部成功要么全部失败，undo log日志用于回滚。**
         2. **Undo log 还用于mvcc。**
   4. Relay log
      1. 中继日志，内容和bin log日志相同，
      2. 作用主要是主从复制中，slava保存主服务器同步过来的binlog日志。
8. 那些情况下需要索引，那些情况不需要索引
   1. 适合
      1. 频繁作为查询条件的列，需要加上索引
      2. 需要参与排序的字段，加上索引可以加快排序速度。
      3. 需要参与分组的列，加上索引。
   2. 不适合
      1. 经常需要更新的列，不能加上索引。因为更新列，会更新索引，会导致update速度很慢。
      2. Where条件中用不到的字段，不要加上索引。

计算机网络篇

1. session和cookie的区别

a) 保存位置不同：cookie保存在浏览器端（用户本地磁盘），session保存在服务器中。

b) 使用方式不同：cookie保存在磁盘中，设置有时间限制，时间限制内就可以通过浏览器读取，时间到了就会被删除。session：浏览器向服务器发送请求是会携带一个sessionId,服务器会根据id查询相应的session对象。SessionId存放在cookie中。**Session离不开cookie**

c） 存储内容不同：cookie存储字符串，session存储结构类似hashtable，可以存储任何内容

d）存储数据大小不同：cookie最大4k（因为在浏览器上），session没有限制（因为存储在服务器端）

e）使用场景：cookie一般存储用户登入信息。Session在服务器端保存了客户的信息（如客户的购物车信息、客户信息等），通过cookie中的sessionId来区分不同客户的session。

2. TCP怎样实现可靠性

a）**效验和**：解决了数据丢失、破坏、篡改、乱序等问题。发送放回填充最后16位效验和，接收方会检验效验和。

b）**序列号和确认应答（ack）**：tcp每发送的**报都会有一个序列号，序列号是递增的**。接收方会转到自己接收了那些报，那些报丢失了，重复的报有哪些，并且接收方会为每一个包返回一个确认应答（ack）。**如接收到了序列号为2的报文，接收方就会返回ack-3表示我需要接收序列号为3的报文了。**

c）**重传机制：**

**1.超时重传：**发送方在发送信息后，会设定一个定时器（时间大小是RTO,一般略大于RTT）,超时时间到，还没有收到接收方的ack，那么就会重新发送该报文。

2.**快速重传：**超时重传问题是在时间到后才会重新发送报文，如果接收方收到了乱序的报文（序列号断了），那么接收方会重新发送最近一次确认报文信号，称之为**冗余ACK，**发送方连续3次收到相同的ack后，就会重新发送接收方需要的序列号对应的报文信息。

d）滑动窗口机制：tcp每发送一次报文，都需要等待对方的ack确认应答，显然会影响速率，最好的方式是将数据一起发送出去，然后一起确认，这样就引入了**滑动窗口**，发送方有一个缓冲区，发送出去的数据都在缓冲区里保存着的，必须等到接受方返回ack后才能将缓冲区中的数据清除，若未接到ack这需要将缓冲区的数据重新发送。**缓冲区的大小一般由接受方确定，主要是为了保证接受方的接收能力，毕竟太多了接受方无法处理。**

e）拥塞控制：

1.慢启动：在不知道网络拥塞情况下，tcp刚建立完成会一点一点的发送报文。

2.快速恢复：**快速恢复是指在发生网络拥塞后对慢启动的优化**。快速恢复不会像慢启动那样刚开始一点一点的发送报文，而是直接将拥塞窗口减半而不会减为1（慢开始就是从1开始。

1. tcp的三次握手和四次挥手
   * + - 1. **三次握手**

流程原理：

首先最开始客户端处于Closed状态，服务器处于Listended状态

第一次握手：客户端想服务端发送syn报文，表示想建立连接。

第二次握手：服务端接收到了客户端的syn报文，会以自己的syn报文作为回答，**并且还会携带ack，此时服务器处于SYN\_RCVD状态**

第三次握手：客户端机接收到了服务端的syn后，**会发送一个ack报文段，然后客户端进入established状态，最后服务端接收到了ack报文段后也进入了established状态。**

**三次握手是否携带有数据？**

第一次和第二次都不可以携带数据，第三次握手是可以携带数据的。

**为什么需要三次握手，两次不行吗。**

三次握手目的是为了确定客户端、服务端，他们的发送和接收能力都是正常的。第一次握手可以说明：客户端的发送能力是正常的，服务端的接收能力是正常的。第二次握手可以说明：服务端的发送能力是正常的，客户端的接收能力是正常的，**但是无法服务端无法确定客户端的接收能力是否正常。**第三次握手说明：服务端、客户端双方都知道对方的接收和发送能力是正常的。

如果只有两次握手是危险的：第一次握手，客户端给服务端发送syn报文，网络延迟导致服务端没有收到从而没有第二次握手，然后客户端因为超时没有收到服务端的第二次握手报文，然后客户端又重新发送syn报文，这一次服务端收到了并返回了第二次握手报文，**然后服务端立马进入了established状态，客户端也是。**双方断开连接后，由于客户端第一个丢失的syn报文成功到达了服务端，然后服务端又进入了established状态，等待客户端报文，这样浪费了服务端的资源。

* + - * 1. 四次挥手

最开始双方都处于established状态

第一次挥手：客户端想要断开连接，**于是给服务器发送FIN报文，客户端处于FIN\_WAIT状态**

第二次挥手：服务端接收到了FIN报文，会向客户端发送ac确认接收到了FIN报文，服务端处于CLOSE\_WAIT状态。

第三次挥手：服务端要是也想关闭连接，**会向客户端发送FIN报文，然后服务端处于LAST\_ACK状态**

第四次挥手：客户端接收到了服务端的FIN报文后，会向服务端发送ack确认报文。**然后最重要的：客户端并不会直接处于CLOSED状态，而是会进入TIME\_WAIT状态，会等待2MSL时间后再进入CLOSED状态。服务端接收到ack后直接进入CLOSED状态**

* + - * 1. **第四次为什么客户端不直接进入CLOSED状态：万一服务端未收到最后客户端发送的ack，会再次发送fin给客户端，然后客户端再发送ack给服务端。如果客户端第四次握手发送ack后直接进入CLOSED，那么此时网络问题，第四次握手没有抵达服务端，服务端重新发送fin给客户端，但客户端已经CLOSED,那么服务端会永远无法关闭，浪费资源。所以客户端不会立马进入CLOSED状态。**

4.TCP和UDP区别

* 1. TCP是面向连接的，有严格的三次握手、四次挥手。而UDP是无连接的，发送消息前是不需要建立连接的。
  2. TCP是可靠的，因为TCP协议可以通过效验和、重传机制、流量控制等方式保证消息安全、不会丢失的到达对方。而UDP将会以最大速率传输数据，不保证可靠交付，会出现消息丢失、重复等问题。
  3. TCP是面向字节传输的，tcp把数据看作是无结构的字节流，每个字节都有编号。而UDP是面向报文的，不会因为网络波动而降低报文发送速率。
  4. TCP是点到点的通讯，而UDP可以是一对一通讯、一对多、多对一。

5.为什么tcp四次挥手，最后客户端处于TIME\_WAIT需要等待2MSL后，客户端才关闭

因为第三次握手时，服务端发送FIN给客户端，客户端收到了FIN后，返回ack给服务端。如果此时ack因为网络原因没有到达服务端，服务端不知道客户端是否收到自己的FIN报文，所以会重新发送FIN报文给客户端。**所以客户端必须等待2个“最大报文存活时间”，来确定是否服务端因为没有收到ack而重新发送fin报文。**

**6.**HTTP和HTTPS的区别

https是就是安全版的http

主要区别：

1. **http是超文本传输协议，信息是明文传输，https是通过ssl加密了的传输协议**
2. 端口不同：http是80端口， https是443端口。

Redis篇

1. redis为什么这么快
   1. **内存数据库：**相比于mysql这种数据存储在磁盘中，**redis是基于内存的数据库（为了安全数据也会持久化在磁盘）。**
   2. **合理的线程模型：**redis处理请求是C写的I/O多路复用（epoll实现的），一个线程可以同时处理多个请求。
   3. **每一种数据结构都很简单，操作数据很高效。**
   4. 处理请求的模块是**单线程，**所以没有上下文切换。
2. 为什么redis处理请求的模块使用的单线程的（其他模块还是多线程的）
   1. 因为redis的瓶颈不是cpu，而是内存的速度，所以单线程完全够用。
   2. 而且单线程还少去了线程间切换的时间。
3. 什么是缓存击穿、缓存雪崩、缓存穿透
   1. 缓存击穿：**查询一个数据库都不存在的数据时**，首先查询缓存未命中，再查询数据库同样没有，每次查询这个不存在的数据都会请求数据库，这就是缓存击穿。解决方法：若数据库没有的值，可以在缓存中设置一个默认值。或者**通过布隆过滤器快速判断是否存在该值。**
   2. 缓存雪崩：同时大量的缓存失效，导致许多请求到达数据库。解决方式：采用较大的随机值，设置过期时间。
   3. 缓存穿透：某个**热点key失效，**大量的请求到达数据库。解决方法：**“永不过期”**设置一个永远不过期的时间、**“使用互斥锁方式redis的setnx”**，只有一个请求能够到达数据库，然后将热点数据待会缓存中
4. Redis的持久化机制
   1. 为什么需要持久化？
      1. Redis虽然是内存数据库，redis服务器运行时数据存储在内存中的，但是为了保证**redis的安全和可用性，我们必须包redis的数据持久化到磁盘中，避免redis宕机恢复后数据丢失的情况**。
   2. RDB方式
      1. rdb就是把内存中的数据**以快照的方式保存到磁盘上**，在指定的时间间隔内，将内存中的数据以快照的方式保存在磁盘中，**是redis默认的持久化方式。** 最后会生成dump.rdb文件
         1. 优点:适合大规模的数据全量恢复、备份，毕竟是快照。启动恢复数据比aof快。
         2. 缺点：因为不是实时的持久化，某一时间没有执行快照保存，会导致redis中部分内存数据丢失。
   3. AOF方式
      1. Aof持久化方式是通过**追加日志**的方式持久化内存数据，**aof解决的问题是实时性问题。默认是不开启的。**
         1. 优点:因为是实时性的，所以保存redis数据是完整的。
         2. 缺点：因为记录的是日志，日志不仅仅包含数据，还有一些其他的信息，所以文件更大，数据恢复慢。
   4. Redis启动恢复数据的流程
      1. 如果开启了aof机制，那么首先会读取aof文件，**因为aof文件实时性高数据更新，**若没有开启aof则是以rdb文件恢复数据。
5. 为什么redis采用进程来做aof重写而不是线程
   1. Aof是通过保存日志的方式记录了redis的操作，随着服务器的运行，aof文件越来越来，重写aof会浪费很多时间，**可能会引起stop the world，如果通过处理请求的进程一起执行重写aof，会导致处理请求效率低。**
   2. 通过fork的方式，创建子进程来完成aof重写，这样主进程可以继续处理请求。
6. 如何保证mysql和redis的一致性
   1. 首先：不管任何方式，都不看保证强一致性，都是最终一致性。
   2. 不太好的方式
      1. **先写redis在写mysql**：A用户修改redis值为A，B用户再修改redis值变为了B。但是A用户到达mysql请求比B用户到达mysql请求慢（本来应该快），那么最后mysql的值是A，而redis的值是B；**redis和mysql就不一致了。**
      2. **先写mysql再写redis**：和上一中方式一样，也会导致redis和mysql不一致。
      3. **先删除redis再写mysql：**用户A删除redis数据，同时去修改Mysql数据。此时用户B查询数据发现redis没有，直接去mysql查询就旧数据（因为用户A修改mysql的请求还没到），然后用户B将查询的旧数据又写会redis，最后用户A请求到达mysql修改为新数据。**最后导致mysql和redis不一致。**
   3. 比较ok的方式：
      1. **先删redis，再写mysql，再删redis**（**“缓存双删”**）: **先删除redis再写mysql会导致mysql和redis不一致，但是再次删除redis，可以删除redis的旧数据。**
      2. **先写mysql，再删redis：先更新mysql的数据，再将redis中的旧数据删除，这样可以保证接下来查询的数据全是新的。**
         1. 存在一个不容易发现的问题：就是redis中的数据早已经失效了，然后用户B查询redis数据发现没有，会去mysql查询。用户A先修改mysql数据，然后删除redis（其实已经失效了）。最后用户B查询到数据后，更新redis还是旧数据。
      3. **先写mysql，然后通过canal监听mysql的binlog日志，然后通过消息队列异步更新redis。**
7. Redis淘汰策略
   1. 过期删除：是在存储key时，给key设置一个过期时间。当过期时间到时，**该 key就不可用了（但是还是存在内存中的）**，然后利用**过期策略对其进行删除**。Redis有两种过期策略：定期删除、惰性删除
      1. 定期删除
         1. 定期删除类似于一个**守护线程，定期扫描redis内存数据，随机删除一些过期的key。**
         2. 随机删除**一些，所以异常定期删除后，并不会将所有的过期key删除。原因是，如果过期的key太多的话，那么一次性删除，会给cpu带来很大的负担。**
      2. 惰性删除
         1. 当查询redis内存 中某一个key时，如果已经过期了，那么 才会删除。
      3. 总结：无论是定期删除还是惰性删除，都无法避免内存中有过期的key，并且，无法删除没有设置过期时间的key。**所以redis提供了内存淘汰策略来解决没有设置过期时间的key无法删除的问题。**
   2. 内存淘汰：过期删除是指，过期时间到了正常删除过期的可以。而内存淘汰是指超过最大内存是的保护机制。
      1. LRU(Least Recently Used 最近最长时间未使用)：当内存不够时，每次添加一个key时，都会删除很久没有使用的旧数据。（key上一次的访问时间存储在key对象中的）。
         1. LRU是使用链表结构实现的，最新被操作的key会放到链表头。所以使用LRU淘汰旧数据时，会淘汰链表尾部的数据。
         2. Redis并没有使用标准的LRU算法，而是**使用近似的LRU算法**。
         3. 缺点：LRU看似保存了热点的key，但还是有一点小问题。假设一个**很久没有被访问的key最近被访问了**，按照LRU策略，这个key会保留下来，但是这显然不是我们想要保存下来的热点key，于是就有了LFU。
      2. LFU(Least Frequently Used 最经常不使用)：LFU和key使用次数有关，根据key最近被访问的频率进行淘汰。
         1. 相比于LRU算法，LFU增加了访问频率这样一个维度来统计热点key。
         2. 首先，LFU使用一个维度的链表，按照最近使用频率从大到小保存key，然后再同频率下，LRU算法使用一个维度的链表，按照最近很长时间未使用，从最近使用到最近未使用时间排序。
         3. 所以，也就避免了，偶尔一下访问很长时间没有访问的key，然后将它保存下来的情况。
         4. 淘汰时，首先判断最少使用频率的，然后在判断相同使用频率下，最长时间不访问的。
8. Redis高可用
   1. 主从复制
      1. 解决了，单机下，redis服务器崩溃后，数据丢失问题。主从复制中，**读写分离，**主服务器处理写操作，从服务器处理读操作。
      2. 主从复制流程：当有从服务器加入到主服务器下
         1. 从服务器和主服务器建立连接、协商同步。
         2. **全量复制：**主服务器同步数据给从服务器（第一次）：主服务器通过bgsave命令生成RDB快照文件，发给从服务器。从服务器开始载入RDB快照文件。**主服务器将RDB发送给从服务器时，还是能正常处理写操作的，并且把写操作写入到replication buffer缓冲区中。**
         3. 后主从服务器建立TCP连接，主服务不断将replication buffer的写操作同步给从服务器。
      3. 主服务器在第一次同步RDB文件时，**主服务器其实是fork()了一个进程**来完成这个操作的，所以不影响主服务器处理写操作。
      4. **分摊主服务器的压力**：如果从服务器突然增加很多的话，**已经加入的redis从服务器器，会帮助主服务器同步RDB文件给新加入的从服务器。**
      5. **增量复制：**如果从服务器突然和主服务器断开连接了，然后后面又恢复了，但是断连时，从服务器数据不一致了。为了保持主从数据一致，主服务器必须将数据同步给从服务器，此时主服务器不会通过**全量复制**将所有数据同步给从服务器，而是通过**增量复制**把断联间主服务器收到的写操作同步给从服务器。 增量复制数据太大了，开销太大。
      6. 主从复制的缺点：
         1. 主服务器崩溃后，没有选举leader功能，必须通过人工去重新选取leader。也就是说，主从复制下，主服务器崩溃，那么redis就不可用了。
   2. 哨兵模式：**所实话我感觉哨兵不就是zookeeper这样的分布式协调服务器吗**
      1. 哨兵是一个独立的进程，其原理就是哨兵通过发送命令给redis服务器，等待redis服务器的回应。**哨兵也是集群的，哨兵实现了高可用了。**
      2. 哨兵主要作用：
         1. 监控主从节点是否监控master、slave是否能够正常工作。
         2. 故障转移：发现master节点崩溃后，选举一个salve节点成为新的master节点，然后通知其他slave节点修改配置信息。
      3. 实现原理：**哨兵集群中通过gossip一致性算法，**传播master是否下线信息，并投票是否选举新leader，以及投票哪一个slave称为leader。
      4. 哨兵模式的缺点：哨兵模式实现了redis的高可用，**但是只有一个主节点能够写操作。**
   3. cluster模式（分片）：
      1. 个人理解，就是将redis数据进行分片处理，然后每个分片都是主从复制实现的。最终，每个分片都可以进行写操作，大大的提高了写吞吐量，且每个分片的数据都实现了备份操作。
      2. 分片原理
         1. 一共有16383个哈希槽节点，然后每个分片平均分配这些哈希槽节点。
      3. Cluster复制原理：
         1. 一个主节点对应一个或多个从节点，如果主节点失效了，从节点或变为主节点。
         2. 所有分片的主节点相互监督，**如果半数的主节点都和某一个主节点的通信超时了，那么这个主节点一定失效了。**
      4. **Cluster模式：实现了真正的分布式存储，有点像kafka这些分区了。**
9. Redis分布式锁
   1. 方案一：
      1. 利用setnx进行加锁，当前进程setnx如果成功获得锁，则返回1，并且将key值设置为了value。如果key已经存在了，那么返回0，获得锁失败，无法进入临界区。
      2. 利用del释放锁，进程执行完临界区后，利用del函数将key删除，就可以释放锁。
      3. 缺点：如果进程获得锁，执行时，崩溃了，**那么无法即时释放锁，那么就导致死锁出现了。**
         1. 所以此时就需要加锁并且设置过期时间（不能setnx后，在执行expire操作，因为这两个操作不是原子操作，如果setnx后，崩溃了，没有执行expire操作，还是会造成死锁问题）。
         2. 解决方案是：redis的set命令同时可以设置key的过期时间（但还是有问题，如果进程没有执行完临界区的操作因为慢了一点，然后锁过期了，其他进程就抢到锁了，导致严重的安全问题）。
   2. 方案二：续期锁：类似于redisson
      1. 为了解决，抢到锁的进程没有执行完，锁就过期了。可以设置一个**守护线程，**用来给锁“续期”。
      2. 如果设置过期时间为30秒，守护线程发现29秒还没有执行完，那么就可以给锁续期20秒。
         1. 情况一：抢到锁的进程执行完了，会关掉守护线程，然后释放锁。
         2. 情况二：如果抢到锁的服务器挂了，因为守护线程和任务进程在同一个进程中，所以守护线程也会同时挂掉。那么时间到了，锁也会超时。其他进程判断超时后，可以抢到锁。
   3. **方案三：红锁redLock**
      1. 前两种方案，很明显**没有做到高可用，**如果redis服务器崩了，那么分布式锁也就没有了。
         1. 此时我们肯定需要用到主从复制、哨兵模式，两种方法都可以实现高可用。
         2. 但是又有问题，也就是数据一致性问题。如果进程A抢到了redis的锁，key写到了redis集群的master中，但是没来得及同步给slave，master挂了。然后重新选举了一个slave成为master节点，进程B也可以抢到锁，因为新的master节点中没有key值。**（红锁问题）**
      2. **基于redlock的分布式锁**
         1. 首先redis是高可用的（服务器大于等于3个），并且不能同时挂掉。
         2. 进程对redis集群加锁时，master必须保证集群中成功写入key的节点个数大于一半。
         3. 然后，master如果挂了，重新选取的master节点，必须拥有最新的数据。（这样已看，不就是raft、paxos、zab算法吗）。
         4. 最后保证了，master挂了，也不会有多个进程同时拥有锁的情况出现。
10. 问题：快速获得top100的电影，如何设计呢。
    1. 首先既然是top100点电影，那么一定是热点数据，所以一定会永远存在redis中。那么用什么数据结构在存放top100的电影呢。
    2. 肯定需要排序，从redis中查询时，返回热度从高到低的top100电影。所以需要用到zset数据结构。Zset是可以支持排序的，可以通过电影访问量作为排序规则。
11. Redis作为计数器
    1. 例如文章点赞量、微博点赞、抖音视频点赞。记录这些数量
    2. 首先，不要每次点赞后，直接更新数据库，因为点赞可能是非常频繁的事，有非常多的视频、微博条目，显然会导致数据库直接挂掉。
    3. 应该先记录的redis这样的缓存数据库中，然后周期性的更新数据库，这样可以大大的降低数据库的压力。
12. Redis作为抽奖工具。
    1. 可以利用set数据结构，通过spop（redis操作set集合的命令，可以删除指定key的set集合中的一个或多个元素，随机删除，然后返回被删除的元素）。

JAVA基础篇

* + - 1. threadLocal
         1. threadLocal叫作线程变量，只能被当前线程所使用，所以不存在多个线程间共享问题。
         2. 实现原理：每个**线程thread都有自己的threadLocalMap对象**，threadLocalMap是threadLocal类的一个静态内部类。ThreadLocalMap通过以threadLocal为key，（threadLocal，value）的方式保存了线程的变量value。

每次通过threadLocal.get方法获得value时，都要通过当前线程的currentThread然后再jvm中得到threadLocalMap，然后以threadLocal对象就可以在threadLocalMap的到value了。 threadLocal.set方法也是类似的。

c）**同一个线程中可以有多个ThreadLocal对象**，这些ThreadLcoal对象都会存放在当前线程拥有的ThreadLocalMap中，以ThreadLocal为key保存着value的。

d）**单例模式下的ThreadLocal可以保证线程安全**

1.单例模式我们都知道如果在多线程环境下，通过懒汉式创建单例时，需要加锁，避免多个线程在同一时间同时创建单例。

2.但是如果用ThreadLocal保存单例，ThreadLocal<singleton>，那么就可以保证每个线程都会有一个自己的单例对象，这样就可以保证线程安全了，

3.具体实现方式可以csdn一下，单例下的ThreadLocal。

* + - 1. ForkJoinPool
         1. ForkJoinPool是一个用于**并行执行任务的框架，类似map-reduce框架，区别就是forkjoinPool只能在一个jvm里运行，而map-reduce可以在集群里面运行。**
         2. ForkJoinPool基本思想原理

**分治法：**将大任务分解为小任务，然后继续分解直到小任务能够被直接执行，最后将任务合并一直向上合并。

**任务窃取：**顾名思义就是forkJoinPool中的线程窃取其他线程的任务。每个工作线程都对应一个工作队列（双端deque队列），某个工作线程的队列任务执行完成后，就会去窃取其他工作线程的任务，目的是避免线程空闲。

* + - * 1. ForkJoinPool相关源码

ForkJoinPool是ExecutorService的一个实现类，一个特殊的线程池。

ForkJoinPool是线程池，需要传入任务（ForkJoinTask的实现类），forkJoinTask有两个实现类，需要实现forkJoinTask的方法（compute）。

RecursiveAction：没有返回值，也就是compute方法没有返回值

RecursiveTask：有返回值，通过泛型确定返回值类型，也就是确定compute方法的返回值

* + - 1. String、StringBuilder、StringBuffer
         1. String

**String是被final修饰的**，所以不可以被继承

String底层是private final char value[]数组实现的，代表**value引用地址是不能改变的，也就是value指向的地址，但是value数组中的内容是可以改变的**。

如果修改String字符串的话，那么就**会创建一个String新对象，所以 浪费空间且浪费时间。**

* + - * 1. StringBuilder

StringBuilder底层也是char[]数组，但是数组是可变的，当创建一个StringBuilder对象是，并且传入一个字符串时，会初始化数组长度，在接下来append方法调用中，因为数组引用·是可变的，所以**不需要重新创建StringBuilder**对象，如果**数组长度不够，会对数组长度进行扩容**。

如果StringBuilder中char数组不够的话，会重新创建一个新数组，然后数组长度扩容是2n + 2;

但是**StringBuilder是线程不安全的，没有被sync修饰**

* + - * 1. StringBuffer

基本和StringBuilder一模一样

**但是StringBuffer是线程安全的，因为其方法被sync修饰了**

* + - 1. **CAS和ABA问题**
         1. CAS(compare and swap)比较并交换

Cas是一种**不需要加锁实现了线程同步的方法，加锁的方式是悲观加锁，而cas是乐观锁。**

实现原理

通过三个变量控制：**内存值、期望值、新值**

在while循环中，不停的获取内存值，然后与期望值比较，如果相同，说明没有其他线程修改，然后就可以将该值修改为新值，如果不同那么就while循环直到相同。

在java中，因为需要直接获得内存的值，所以一般方法都是native修饰的，**这些方法都是有c++实现的。所以都是Unsave不安全的类，（如compareAndSwapInt），**所以一般都是有jdk开发人员使用。

Java中一些类就用到了cas乐观锁机制：AtomicInteger类的方法getAndAdd()就用到了compareAndSwapInt这个不安全的 方法。

* + - * 1. ABA问题

ABA问题出现在 CAS机制中，**如果值A修改为值B，然后又被修改为值A**，那么其他线程不知道该值是被修改过的，所以**内存值和期望值是相同的，**然后就swap成功了。

解决方法：一般是在加一个变量进行控制，**Version版本号，当值被修改了一次，那么version就会增加1**，cas判断的时候就可以判断version是否改变过。

* + - * 1. Cas的缺点：cpu的消耗很高，如果变量频繁被多个线程操作时，那么cas机制中的while就会一直消耗cpu。
      1. 抽象类和接口的区别
         1. 抽象类：对某类东西进行抽象，是具体的一种东西。

抽象类被abstract修饰

抽象类中可以有变量、方法。抽象类中的**抽象方法必须是被abstract修饰且被public或者protect的**，因为抽象类就是为了被继承而设计的。

抽象类不能实例化对象，**必须被子类继承，并且子类必须实现抽象类中的抽象方法后才能够实例化子类对象。**

* + - * 1. 接口：对某一种行为进行抽象。

接口中的**变量被隐式定义为且只能为public static final。**

接口中的**方法被隐式定义为且只能为public abstract。**

* + - * 1. **区别：**

抽象类中的方法可以不是抽象方法，非抽象方法可以被实现（没有被abstract修饰的）。**而接口中的方法一定是public abstract修饰的，所以不能被实现。**

抽象类中的变量可以是任何类型的，而接口中的变量一定是public static final类型，且一定要被初始化。

抽象类中可以包含静态代码块、静态方法。**但是接口中不能有静态代码块、静态方法。**

* + - 1. **HashMap相关**
         1. 什么是负载因子：负载因子是hashMap扩容的一个阈值，size/capacity。当hashMap元素个数超过总容量的负载因子，就会进行扩容。

为什么需要扩容：扩容是为了**减少哈希冲突，负载因子越低，冲突越小，但是更耗费内存。越高，冲突越大**

* + - * 1. HashMap初始化：**初始化大小是16，负载因子是0.75。**
        2. **HashMap扩容机制：**插入后如果超过了负载因子，那么**数组长度会变成原来的两倍。**

resize()方法可以实现扩容，但是此时就会伴随**重新hash，这是一个非常耗时的操作。**

**所以要尽量避免resize()方法。而是要使用rehash()方法。因为每次都是都是翻倍，所以原来的hash值比扩容后的hash值少一个bit而已，**所以原来数据的位置在扩容后的map中要么是原来的位置要么是【原来的位置+就数组长度】。

相比于简单的resize()后，重新计算旧数据的hash值，rehash()方法回快很多。

* + - * 1. HashMap链表升级为红黑树的条件

链表长度大于8

且数组长度大于等于64

* + - * 1. **hashMap底层如何存储的**

通过数组+链表+红黑树组成的

为什么最后链表会退化成红黑树：因为经常发生hash冲突的话，**哈希槽的链表会很长，链表的查询非常慢，所以当达到阈值的时候，最后链表会退化为红黑树，红黑树的查询很稳定。**

* + - * 1. **HashMap不是线程安全的**

HashTable是线程安全的

HashTable如何保证线程安全的

hashTable类中的方法都**被synchronize修饰了的，所以效率不高。**

hashTable初始化时

初始化的容量是11

负载因子是0.75，和hashMap相同

**ConcurrentHashMap是线程安全的**

ConcurrentHashMap如何保证线程安全

ConcurrentHashMap底层如何实现的

Jdk1.7前：采用的Segment数组+链表的方式，Segment类中有HashEntry（和HashMap的Entry一样），Segment类继承了ReentrantLock。**所以采用的分段锁技术，Segment数组某一个segmeng被锁不会影响Segment数组中其他的segment，效率很高。**

JDK1.8后：**抛弃了Segment分段锁，采用CAS+synchronize保证线程安全。底层数据结构采用数组+链表+红黑树。**

放弃Segment分段锁的原因是，因为分段锁占用内存。

* + - * 1. 还可以采用Collections.SynchronizedMap这个方法将HashMap对象转为线程安全的map容器。

保证线程安全的方式：每个方法都被synchronize修饰。

* + - 1. hashMap扩容为什么是两倍
         1. 尽可能的减少元素位置的移动：当hashMap中数组扩容后，元素在新数组中要么位置不变，要么有规律的变化（新位置：原数组位置+原数组大小）。
         2. 可是使元素均匀的散布在hashMap数组中，减少hash碰撞。在hashMap中，n表示容量大小，有一个操作时(n-1) & hash这个操作，n是2的幂次方，n-1就是11111…111的形式，所以最后(n-1) & hash会均匀散布在map中。为什么11111这样会导致均匀，可以csdn一下。
      2. hashMap遍历方式
         1. 第一种就是map.entrySet()，获得一个entry集合，然后通过增强for遍历entry集合
         2. 第二种就是map.entrySet().iterator()，获得一个迭代器，然后遍历迭代器即可。通过while(iterator.hasNext())遍历。
      3. 为什么ConcurrentHashMap不能存方法null值，而HashMap可以
         1. **如果ConcurrentHashMap对象、HashTable对象，在put时，key或者value为null的话，会判断然后抛出空指针问题。Jdk就是这样写的。**
         2. 简单来说，是为了避免并发过程带来的歧义问题。

对于HashMap来说，为什么可以插入null值是因为**HashMap就是为了单线程设计使用的。**

单线程A对HashMap对象进行了get(key)返回null，我们虽然不知道是因为对象中**到底是没有key还是所key对应的value是null**。但是我们可以通过containsKey判断是否有该key，然后得出结论。

但是对于多线程下的ConcurrentHashMap(HashTable也是一样)，假能够存入null。假如，concurrentHashMap对象本来没有某个key，线程A执行了get(key)返回了null，此时线程A不知道到底是因为对象中没有key还是因为该key的value是null，然后线程A调用containsKey(key)，但是在此之前，线程B调用了put(key, null)。最后线程A的containsKey方法返回true。 这种情况明显造成了歧义，本来对象中没有该key，理应线程A调用containsKey返回false，可以最后线程B的介入，导致发生了歧义。

* + - 1. HashMap的put插入过程
         1. 首先，根据key的哈希值(hash(key.getHashCode()))，找到在hashMap中数组的位置
         2. 然后，如果该位置没有元素，那么直接插入。如果有元素，那么遍历该位置的链表，如果存在相同的key，那么直接替换value为最新的。如果没有相同的key的节点，则将新节点加入。
         3. 如果判断，数组中该位置的链表长度大于了8，并且数组长度大于等于64，那么链表会**退化为红黑树。**
         4. 当哈希表中负载因子大于了0.75，那么会对数组进行扩容。扩容为原来的两倍大小。然后执行rehash()函数，对原来的数据进行重新hash到新数组中的位置。
      2. HashSet相关
         1. HashSet类是Set接口的实现类，底层存储数据是靠HashMap对象。
         2. HashSet特点：

存储元素时唯一的（靠元素对象的引用地址hashcode判断是否是同一个对象）。所以，如果两个不同的对象（对象引用地址不同），但是两个对象数据相同，也会都存储到hashSet集合中（**这个问题需要依靠重写equals方法和hashcode方法来解决**）。

存储数据是无序的（put的顺序和遍历顺序是不一样的，**LinkedHashSet可以实现put顺序和遍历顺序一致**）。

线程不安全的。

* + - * 1. HashSet的实现原理：

HashSet如何存储数据的：

HashSet的所有构造函数，都会创建一个HashMap对象，hashMap对象用于存储真实数据。

HashMap对象的每个entry的key都存放了HashSet的一个值（真实的值）。每个entry的value没有任何意义的，HashSet类中定义了一个Object final对象PERSENT,就是用于填充value位置的。

为什么HashSet能够保证元素不重复

底层存储数据的是HashMap对象。HashMap保证了每一个key只能有一个。（具体为什么hashMap保证了key只有一个，可以看hashMap的put过程）

HashSet的方法

add方法其实就是调用Hashmap对象的put方法

remove方法其实也是调用HashMap对象的remove方法。

* + - * 1. 最后提一句：HashSet为什么是无序的

因为HashMap是无序的。

* + - 1. LinkedHashMap
         1. 为什么会有LinkedHashMap这个类出现呢？

因为HashMap是不保证顺序的。也就是说，添加数据到HashMap对象的顺序，和遍历HashMap数据的顺序是不同的。 无法实现LRU这些功能。

而LinkedHashMap保证了添加顺序和遍历数据的顺序是相同的。 **实现原理就是HashMap+双向链表**

* + - * 1. LinkedHashMap具体如何保证顺序的呢？

LinkedHashMap继承了HashMap，**但是LinkedHashMap重新定义了HashMap中的Entry静态内部类。**在原来的Entry类基础上，**增加了两个属性before、after都是Entry类型。**实现了双向链表的功能。

LinkedHashMap中也增加了一个属性，header是Entry类型。 也就是链表的头结点（header会在LinkedHashMap构造函数时被初始化）。

LinkedHashMap的构造函数，其实基本就是调用super()方法，也就是执行HashMap的构造方法。（HashMap构造方法回调用一个方法init()），init()方法在HashMap中是一个没有body的方法，**而在LinkedHashMap中被重写了。 Init方法被重写用于初始化header节点（是不是前后呼应了）。**

最后分析：LinkedHashMap的put方法，就是用的HashMap中的put方法，但是put方法中调用了一个方法addEntry，所以**关键就是LinkedHashMap重写了addEntry方法，和addEntry中的createEntry方法。**重写后的createEntry方法就会**创建一个entry并且添加到双向链表中**

* + - 1. LinkedHashSet相关
         1. 有了LinkedHashMap，LinkedHashSet就是利用LinkedHashMap实现了有序性。
      2. TreeMap和TreeSet
         1. 它们底层是通过红黑树维护的，也就是利用搜索树实现的Map和Set
         2. TreeSet依然是靠TreeMap来存储数据的，所以我们只需要搞懂TreeMap就可以了。
         3. TreeMap如何实现的

TreeMap类中有一个属性，compator用于比较元素的大小实现有序。 （通过比较key来实现有序的），如果key的类型本来就实现了Compator接口，那么TreeMap不需要传入Compaor对象。

TreeMap中还有一个属性root，是Entry类型，也就是红黑树的根节点。

* + - 1. ArrayList
         1. ArrayList的扩容机制

**ArrayList就相当于一个动态数组**

ArrayList初始化时，数组容量为10。

扩容流程：当ArrayList容量不够时，会创建一个新数组，**长度是原来数组的1.5倍**，然后调用**Arrays.copyOf方法将原来数组内容复制到新数组中**，在将新添加的数据加入到新数组中。

* + - * 1. ArrayList是线程不安全的，可以使用Vector代替，但是Vector通过synchronized加锁，所以很慢
        2. 可以用CopyOnWriteArrayList代替，**写时复制，有写操作是，会创建一个新的副本，不会在原数组上写，修改完后将原数组指针指向副本。读时不需要，所以适合读多写少。如果写很多的话，频繁复制很消耗内存和时间**
        3. 比如add()方法，首先直接加锁。然后创建一个新数组，将原来的旧数据复制到新数组中，最后将新值填进去。
      1. LinkedList的扩容机制
         1. LinkedList是没有扩容机制的，它底层实现是链表，所以一直向后增加就可以了。
      2. **线程池**
         1. 线程池的优势：

降低内存的消耗：通过重复使用已经创建好并放入线程池的线程，减少不断常见线程消耗的内存。

降低处理任务时间：创建线程需要一定的时间，直接从线程池中获取线程，很好的减低反应时间。

合理的管理线程：将线程集中在一起，更好的管理监控线程

* + - * 1. Java中用的最多的线程类**ThreadPoolExecutor**

7个核心参数

corePoolSize：核心线程数。线程池中核心的线程数，即使核心线程空闲时间很长，也永远不会被销毁。

maximumPoolSize：最大线程数，线程池中允许存在的最多的线程数，

keepaliveTime：最大空闲时间，多余核心线程数的线程，空闲时间到了后，会被销毁

TimeUnit：时间单位，描述空闲时间的时间单位，可以是秒、毫秒等。

WorkQueue：任务队列，通过线程池的execute（）方法提交的Runable任务，会被存放到任务队列中。

**阻塞队列，有许多的实现类可选择**

ThreadFactor：线程工厂，指定创建线程的方式。

默认**用jdk自带的DefaultThreadFactory。**

Handler：拒绝策略，当任务到达了最大线程数无法执行时，拒绝任务的策略。（任务队列无法再放入任务时）。

拒绝策略**有四种**

AbortPolicy：任务添加失败的话，直接丢弃任务，并抛出“拒绝异常”

CallerRunPolicy：任务添加失败的话，由主线程自己执行该任务

DiscardPolicy：添加失败的话，就直接丢弃，不会抛出任何异常。

DiscardOdiestPolicy：添加失败的话，就丢弃任务队列中最早的任务。

* + - * 1. 还有一些线程池，但是一般不会用

FixedThreadPool：定长线程池

ScheduleThreadPool：定时线程池，核心线程数固定，**最大线程数不固定，容易出现OOM**

CacheThreadPool：可缓存线程池，**线程数不固定，容易出现OOM**

SingleThreadExecutor：单线程化线程池，只有一个线程，任务时按顺序执行的，不许需要考虑线程同步问题。

* + - 1. 阻塞队列BlockingQueue
         1. 阻塞队列也是一种双端操作的队列。但是提供了超时、阻塞功能。普通的add()方法添加元素时，如果队列满了会抛出异常。普通的remove()方法取元素时，如果队列没有元素，那么也会抛出异常。

阻塞

取元素（利用take()方法取时），如果队列中没有元素，那么**会阻塞当前线程，**直到阻塞队列中有元素时。

想队列尾部添加元素时（利用put()方法添加时），如果队列满了，**那么当前线程会阻塞，直到阻塞队列中元素被其他线程取出时。**

超时

取元素（利用poll()方法时），如果当前阻塞队列没有元素，并且传入了时间，那么超时时间后，poll()方法回返回null; 如果在超时时间内，队列中被添加了元素，那么poll()方法会放回取到的元素。

添加元素（利用offer()方法），如果阻塞队列满了，并且传入了超时间，在超时时间内，队列还是满的，那么会返回false; 如果在超时时间内，队列不满，那么offer方法添加成功返回true;

* + - * 1. 阻塞队列常用场景：消费者和生产者模型。
        2. 常见的实现类：

ArrayBlockingQueue（常用）:定长的

内部维护了一个**定长**的数组（生产者和消费者不可以并行执行）。**消费者和生产者公用一个锁，似乎吞吐量不太高**，可以消费者和生产者各用一个锁。

LinkedBlockingQueue：定长的

内部维护了一个链表（链表大小在构造函数中可以指定，若不指定大小是MAX）。生产者和消费者都有各自的锁（所以生产者和消费者可以并行执行）。

DelayQueue：无限长

只有当元素的延迟时间到了，才能被消费者取出。

PriorityBlockingQueue：定长的

元素的取出具有优先级（在构造函数时，传入一个Compator对象来觉得优先级）。

* + - 1. 乐观锁和悲观锁
         1. 悲观锁：某一线程在访问资源时，认为一定会有其他线程修改该资源，**所以一定会显示加锁。比如java中的synchronized、Lock类。**
         2. 乐观锁：某一线程访问资源时，认为一定没有其他线程会访问修改该资源，所以不会加锁，只是在更新的时候会判断之前有没有线程更新该数据。**常见乐观锁实现：CAS机制**
      2. 自旋锁和非自旋锁
         1. 自旋锁：线程在没有获得资源时，不会导致线程休眠（cpu上下文切换到其他线程），该线程是不会放弃cpu的时间片的，而是该线程自己自旋等待。 **自旋锁的实现原理同样也是CAS。自旋锁的缺点：如果自旋时间太长，浪费cpu时间会很长。优点：如果自旋时间非常短就获得了资源，那么减少了cpu上下文切换时间。**
         2. 非自旋锁： 当线程不能获取某一资源时，就会被cpu挂起，线程发生状态变化。Cpu时间片段会让给其他线程。
      3. Synchronized和Lock的区别
         1. Synchronized

是java关键字，是java**内置锁/监视器锁（monitor）**。java每个对象都有内置锁（监视器锁），synchronized就是使用java内置锁将代码块（方法）加锁（**jvm基于进入和退出Monitor对象来实现方法同步和代码块同步**）。

Synchronized是非公平锁（不会按照线程阻塞顺序唤醒线程）。

Synchronized是隐式锁，不需要程序员主动去加锁和释放锁，当发生异常时，不需要程序员释放锁（不会出现死锁问题）。

Synchronized可以锁**普通方法（锁对象）、静态方法（锁类）、代码块（锁对象）。**

**Synchronized是可重入锁。**

Synchronized就是排他锁（独占锁），**不能实现读共享、写排斥**

* + - * 1. Lock类的锁

Lock是一个接口，这个接口下有很多实现类（比如ReentrantLock）。需要程序员显示创建锁、然后显示加锁和释放锁。

Lock锁需要程序员主动释放锁，**在finally代码块中，不然程序发生异常会导致阻塞（因为lock锁不会主动释放，必须显示调用unlock方法）。**

Lock可以实现共享锁（读共享），Lock实现类ReadWriteLock可以实现读共享、写排斥。

* + - 1. AQS（Abstrac Queued Synchronizer 抽象队列同步器）
         1. 它是juc下的一个重要的类。许多juc下的类都继承了它。比如ReentrantLock中的Sync继承了AQS。
         2. AQS中有两个重要的概念：state（资源状态）、同步队列（CLH）。

state：资源状态。

初始值为0表示没有线程再使用资源。

当某个线程t尝试获得锁成功，state就变成了1。表示有一个线程正在访问资源。

当另一个线程尝试获得锁，发现状态为1，就会加入**同步队列中（CLH）**。

当线程t再次获得锁，state会加1，这就是ReentrantLock可重入锁的实现。当线程t释放一次锁，state就会减1.

CLH（同步队列）：队列是一个**双向链表，**链表中的元素是Node对象（AQS的内部类）。

Node类中有几个属性：prev当前节点的前置节点、next当前节点的后置节点、Thread当前节点代表的线程、waitStatus当前节点的状态、nextWaiter。

waitStatus取值：SIGNAL(-1)表示当前节点在**同步队列（CLH）**上等待、CONDITION(-2)表示当前节点在**条件队列**上等待。

* + - 1. ReentrantLock底层实现
         1. reentrantLock底层是靠AQS实现的。Abstract Queued Synchronizer（抽象队列同步器）

ReentrantLock类中有一个抽象类Sync继承了AQS，他有两个实现NonfairSync非公平锁和FairSync公平锁。

默认创建ReentrantLock是非公平锁，可以构造函数中传入true表示公平锁。

* + - * 1. （FairSync）公平锁的实现

Lock过程（获取锁的过程）

lock（）方法中调用acquire()方法（该方法是AQS中的final方法不能被子类实现）。acquire()中调用了tryAcquire()方法（该方法是AQS中的抽象方法，被子类Sync的NonfairSync和FairSync分别实现）。

如果tryAcquire()方法返回true表示获得锁成功，那么不用再向下执行了。

**如果tryAcquire()返回false，表示当前线程获得锁失败。那么就会执行acquireQueued(addWaiter(Node.EXCLUSIVE,arg))方法。**

tryAcquire()方法，目的是尝试去获得一下锁。

判断如果state是0的话，说明表面上没有其他线程获得锁。**但是并不意味着当前线程可以获得锁。**还需要满足两个条件，hasQueuedPredecessors()方法返回false，然后compareAndSetState(0, 1)返回true。

hasQueuedPredecessors()方法很巧妙。如果上一个持有该锁的线程释放锁，其后继节点（线程a）还没来得及获得锁，然后当前线程b就抢到了锁。这不代表当前线程b能获得锁，有资格获得锁的线程应该是线程a，b应该去排队。**不然不符合公平。也就是说如果CLH队列中有线程等待的话，那么hasQueuedPredecessors()方法回返回true，如果没有就返回true。**

compareAndSetState()，通过cas将state改为1。

如果hasQueuedPredecessors()返回false和compareAndSetState()返回true，那么表示当前线程可以获得锁。然后执行setExclusiveOwnerThread(current)方法，将持有锁的线程设置为当前线程。**tryAcquire()方法返回true，表示当前线程获得锁成功。**

如果state不等于0时。**不代表当前线程不能获得锁。（因为是可重入的）**

判断持有锁的线程是否是当前线程，如果是就将state加1。**然后tryAcquire()方法返回true，表示当前线程获得锁成功。**

如果持有锁的线程不是当前线程，那么tryAcquire()返回false，表示获得锁失败。

如果tryAcquire方法返回false，表示当前线程获取锁失败，那么久调用addWaitre方法，将当前线程的Node节点**添加到同步队列的尾部。（一定能添加成功）。**

最后调用acquireQueued方法，循环。判断当前线程是否是头结点、或者被当先节点的前置节点释放锁时唤醒。

Unlock（释放锁的过程） 释放锁过程比较简单了

unlock()方法中调用release()方法（release方法时AQS类中的一个final方法，和acquire方法时一对）。release方法中调用**tryRelease方法（公平锁和非公平锁公用同一个tryRelease方法。该方法被ReentrantLock类实现。不像tryAcquire方法FairSync和NonFairSync各自有自己的实现）。**

tryRelease方法中，获得state，然后将state减1。

然后判断state是否为0，如果为零表示当前线程已经不存在重入了。然后执行setExclusiveOwnerThread(null)将获得锁的线程设置为null。然后tryRelease方法返回true。

如果state不为0，表示当前线程还有重入。tryRelease就返回false，不会唤醒后继线程。

如果tryRelease方法返回true，那么就会执行unparkSuccessor方法唤醒后继线程。

* + - * 1. FairSync（非公平锁）实现

非公平锁和公平锁的区别就在lock获得锁时不同。

非公平锁会lock方法中首先直接执行compareAndSetState方法直接将state设置为1，然后执行acquire方法。而公平锁的lock方法直接执行acquire。**所以造成的结果就是：当前线程不需要排队，可以直接获得锁，不公平**

还有就是：nonFairTryAcquire方法中，判断如果state是0的话，不会执行hasQueuedPredecessors()方法，判断是否同步队列中是否有等待的线程，而是自己compareAndSetState获得锁。 而公平锁的tryAcquire方法为了保证公平会执行hasQueuedPredecessors方法。

* + - 1. Semaphore
         1. 允许一定数量的线程一起运行。类似与操作系统重的信号量
         2. 实现原理：其实和ReentrantLock差不多。

Semaphore类中也有一个Sync类继承了AQS类。同步对state的增加和减少。

然后acquire获得锁的操作、release释放锁的操作和ReeantrantLock差不多。

* + - 1. CountDownLatch
         1. 主要使用场景是：某个线程必须等待某些线程执行完后，才能开始执行。

CountDownLatch构造函数，传染int，表示计数。

当任务线程执行完后，然后执行latch.countDown()，然后计数就会减一。

主线程执行latch.await()方法然后会阻塞，等待计数变成0，然后主线程才会继续执行。

* + - * 1. 实现原理：CountDownLatch类中也同样有一个Sync类，然后Sync类继承了AQS类。

所以最终，countDown()方法，还是对AQS类中的state减1操作。

* + - 1. Volatile关键字的作用
         1. 并发编程中通常会遇到三个问题：**原子性、可见性、有序性。**
         2. 可以保证共享变量的**可见性，它会保证修改的值会立马更新到主存中，其他线程需要时取的最新值。**
         3. 可以保证一定的“有序性”（synchronizied和lock也可以保证有序性）。volatile也可以**禁止指令重排**，不能将被volatile变量后面的语句放到其前面执行，
         4. **不能保证原子性（只有lock、synchronized可以保证原子性）。**
      2. 什么是偏向锁、锁升级是什么（轻量级锁、重量级锁）
         1. 什么是偏向锁：偏向锁是java中，为了解决加锁导致性能下降而设计的锁。大部分情况下，某个对象是没有竞争的，所以，可以通过偏向锁来提高性能。

偏向锁实现就是，将对象头中的Mark标记设置为偏向，当前线程id写到对象头Mark中。线程是不会自动释放偏向锁的，除非遇到其他线程竞争（也就是对象头中的mark标志记录的线程id是不会自动删除或修改的）。

当下一次该线程再次想要获取该对象锁时，对比直接的线程id和对象头中的线程id是否相同，相同的话，那么获取成功。

如果另外的线程也来获取该对象锁时，对比线程id发现不同，**那么此时锁发生了竞争，偏向锁会升级为轻量级锁。**

Jvm启动时可以选择关闭或开始偏向锁机制，如果开启了偏向锁机制的话，性能大概能提高5%。

* + - * 1. 轻量级锁：jvm会为每个线程的栈帧中创建用于存放**锁记录空间（Displaced Mark Word），并将对象头中的Mark Work复制到锁记录中。然后线程通过cas尝试将对象头中的Mark Work替换为自己锁记录空间的指针。**
        2. 重量级锁：没有抢到锁的线程直接被阻塞，等待唤醒。减少了cpu自旋带来的资源损耗。
        3. 什么是锁升级

承接上面的偏向锁升级为轻量级锁。另外一个线程竞争锁时，**如果原来的线程还在执行没有放弃锁的话，竞争失败的线程会陷入cas自旋状态，等待获取轻量级锁。**

自旋达到一定时间，还没有获得轻量级锁时，锁会进入重**量级锁。**

* + - 1. 重写equals方法，为什么还要重写hashcode方法。
         1. 为什么有hashcode这个方法，

**为了加快效率：**比如像set这种集合，不能有重复的元素，那么如果已经有1000个不同的对象在set集合中，再添加第1001个对象时，如果只有equal方法时，那么通过equal判断1000次的话，效率太低了。所以直接通过hashcode方法回快很多。

Set集合去重的步骤：

首先判断两个对象的hashcode是否相同（没有重写的Object类的hashcode方法，是通过对象引用地址获得的hashcode）。

如果hashcode不相同，那么就直接插入不用判断equal方法了。

如果hashcode方法相同（比如重写了hashcode方法，那么hashcode值就不是通过引用地址hash得来的）。那么就会判断两个对象的equals方法是否相同，如果不同那么就插入。如果相同那么久修改旧值为新值。

* + - * 1. 原始的equals方法判断的是两个对象的引用地址是否相同。所以即时两个对象的属性完全相同，那么equals也是不同的。比如Person类（两个属性age、name），没有重写equals方法。p1=Person(1, “xiaoming”)和p2=Person(1, “xiaoming”);这两个对象很明显几乎是同一个对象吧（毕竟属性是相同的），所以我们想set集合中存的话，一定要去除其中一个。所以此时我们要重写equals方法，通过判断age和name是否相同。

但是问题来了，set集合首先判断的hashcode方法，很明显hashcode方法没有重写，那么hashcode值是不相同的，所以p1和p2都被存储到了set集合中。不符合我们的期望。

所以我们重写hashcode方法，hashcode方法就通过hash(age)+hash(name),获得对象的hashcode值。最后p1和p2比较hashcode相同，那么再会判断p1和p2的equals方法是否相同，很明显p1和p2的equals方法相同，所有最后去重了。

如果凑巧，有一种情况p1和p2,的age不相同，name也不相同，但是hash(age)+hash(name)相同，然后，再判断p1和p2的equals方法，发现equals方法不同，最后p1和p2都会存储进去。 （这符合我们的期望，毕竟age和name都不同，所以坑定不是同一个对象。）

* + - 1. Java中Exception和Error
         1. Excepton和Error这两个类都是继承自Throwable类。
         2. Error：是**不可预料**的错误，比如OOM、NoClassDefFound这些致命的错误。会直接导致JVM不可处理或者不可恢复。所以这些情况是不可以被try catch到的。
         3. Exception:**可预料**的错误

**检查性异常**：比如IOException、或自定义的Exception类，都是可以被try catch到的。

**非检查性异常**：空指针异常、数组越界异常。这些错误在代码编写时可以避免。而且**直接有程序自己捕获并抛出。**

**Kafka篇**

kafka如何保证消息顺序

kafka不能保证全局消息顺序（不能保存同一个topic内所有消息的顺序）

kafka只能保证topic中某一个partition分区的消息顺序。

kafka为了增加吞吐量**，设计了一个topic下有多个partition，这样的设计增加了并发量。**如果还要保证同一个topic内消息的顺序，那么某一个partition分区在被消费时，**其他的分区一定是被锁住了。违背了分区的意义。**

具体的原理。

生产时：消息会根据key值，或是hash算法或是其他算法被分配到一个partition中。

消费时：某一个消费则组订阅了一个topic，那么某一个分区只能又一个消费者组中的某一个消费者消费，所以也保证了partition内消息顺序。

Kafka消息重复消费以及解决方案

Kafka消费模型：

Kafka中一个消费者组消费一个topic信息。发布到topic中某一个partition的数据只能被消费者组中一个消费者消费。

消费者消费完信息，会提交offset偏移量。

所以，kafka出现重复消费的原因是：消费者消费了topic中某一个partition消息，但是还没来得及提交offset，发生了以外。

**强行kill消费者（消费者突然挂了）**：导致消费者消费数据没来及提交。比如，消费者poll了500条消息，消费了200条，但是此时消费者挂了。那么kafka会触发rebalance，那么该partition会分配给其他的消费者消费，但是新消费者不知道offset，会导致重复消费200条数据。

**消费这消费时间过长**：消费者消费数据很耗时，到时session timeout，然后触发rebalance，然后新消费者poll数据，发送重复消费。max.poll.interval.ms，这个参数设置了consumer在规定时间内没有消费完上一次数据，那么consumer就会主动离开group，然后发生rebanlance。

解决方法：

提高消费能力，可以使用多线程方式提高消费者处理消息的速度。

max.poll.interval.ms，超时时间设置长一些，避免session timeout，然后发生rebalance。

kafka保证消息不丢失

生产端阶段：ack机制保证，发送到broker的每个信息，都会有反馈机制，保证消息正常被接受。Ack类型可以设置

Ack=0时，不需要等待broker相应，producer发送了就不管了。丢失了就真丢失了。

Ack=1时，需要等待leader写入成功后，返回相应。如果leader没有同步给任何一个follower就崩溃了，那么消息也会丢失。

Ack=-1时，表示等待所有broker写入成功后，返回相应。**不是严格意义上的所有broker，而是ISR集合，ISR集合中有leader和若干个follower，ISR集合中的所有broker写入这个后返回相应。如果ISR集合只有一个leader，那么和ack=1情况相同。**

Broker存储端：发送端将消息发送到broker后，broker会将消息持久化到磁盘。所以，持久化就涉及到**刷盘机制。**

同步刷盘：发送端将消息发送到broker后，只有broker将消息持久化到磁盘后，才返回ack。能保证消息不回在broker端丢失，但是很影响性能。

异步刷盘：发送端将消息发送到broker端后，只要写入page cache缓存，就返回一个ack。这样提高了mq性能，但是如果broker挂了，pagecache缓存没有被操作系统刷到磁盘，那么消息回丢失。

Broker一般是集群部署，有master和slave，所以通过副本保证了高可用。

消费端消息丢失：

消费者回提交offset消息偏移量，如果设置了autoconmmit=true，那么消费者poll消息没有消费完就自己提交了offset，然后挂了，那么消息没有消费，就丢失了。Rebalance后，新消费者从offset开始poll消息。

解决方法：

autocommit=false，手动提交

设置ack=-1，保证发送消息不回丢失。

Broker端，利用复制机制，保证多slave。

Kafka中的coordinator

Coordinator作用是管理consumer group中的元数据，如分配分区数、重平衡。

Coordinator分类：

Group Coordinator：负责管理消费者组的生命周期，包括consumer加入、心跳、重平衡、分区等。

Transaction Coordinator：负责事务性的Producer的协调工作，包括TransactionId管理，prepare阶段的启动和Commit/Rollback等操作。

Kafka rebalance

发生重平衡情况

消费者组中消费者数量发生变化，比如新增消费者，最常见是某个消费者宕机了。会发生rebalance。

某个主题的分区数发生变化，但是目前kafka只支持分区数增加。

重平衡的缺点：

当kafka发生重平衡事，消费者组是不可用的，应该尽量避免发生重平衡。

重平衡流程：两个步骤分别是加入组和等待领导消费者的分配方案。这两个步骤的请求分别是：JoinGroup和SyncGroup

当组内有新消费者成员加入是，新消费者会向coordinator发送joinGroup请求。该请求后，每个组员都要将自己订阅的主题上报给coordinator。

Coordinator会将这些信息发给消费者组中的领导（一般是第一个加入消费者组的消费者）。然后由消费者领导做出统一的分配方案。

然后领导者分配方案放到syncGroup中，发送给coordinator。同时，消费者组内的其他消费者也会向coordinator发送syncGroup请求（但是不携带信息），目的是为了让协调者将分配方案发送给消费者。

最后coordinator通过syncGroup请求，将分配方案发送给各个消费者。

Rebalance算法

Range：这种分配算法基于**每个主题分配的**。消费者组可以订阅多个主题，每个主题都会有多个分区数，如果某个主题的分区数不能平均分配给消费者，那么多余的分区会分配给某一个消费者。

RoundRobin：是kakfa某人的分配算法。这种分配算法是基于**全部主题的分区数**的。会平均分配该消费者组订阅的所有的主题的所有分区给所有的消费者。

Sticty：该分配算法是为了一定程度上解决重平衡非要重新分配全部分区的问题。保证目前的分配情况不要发生变化，只移动尽可能少的分区。

Kafka中的事务

场景：

Kafka为什么高吞吐量

顺序写入：kafka支持顺序写入日志的操作，而不是随机写入像b+树一样。顺序写入不需要硬盘的寻道时间，只需要扇区旋转的短暂时间。随机写入，需要硬盘寻道，会很慢。

D MA（直接存储器访问）技术。它允许不同的硬件之间沟通传输数据，不需要cpu参与，cpu可以干其他事情。通常网卡、磁盘设备等都支持DMA技术。

零拷贝技术：

传统的文件拷贝到网络中发送出去，首先需要通过内核态将硬盘数据DMA方式加载到内核的**内核缓存 ，**然后cpu切换调用read加载到用户态的**应用层buffer**也就是代码设置的数组缓冲中。最后通过应用层代码将数据然后cpu切换调用write加载到**socket buffer**中，然后DMA方式发送到**网卡**上。 需要4次copy。 Cpu切换4次。

第一种mmap零拷贝。少了read和write两个操作。用户程序通过mmap()函数，将磁盘数据通过DMA拷贝到内核缓冲区。然后操作系统会将这段数据和用户程序映射（这样就不需要将内核数据拷贝到用户程序中）。然后用户程序调用write直接通过cpu空隙将内核数据拷贝到socket 缓存中。最后socket缓存通过DMA将数据拷贝给网卡。 如此相较于传统的拷贝方式，CPU上下文切换从4次减少到了2次，数据copy重4次减少到了3次。

Sendfile方式：linux2.4将零拷贝技术推向顶峰。Sendfile函数，用于将内核缓存数据直接通过DMA方式copy到网卡。所以最后只有两次copy：第一次是磁盘到内核缓存、第二次是内核缓存到网卡。 Cpu不需要参与，没有上下文切换。

分区：每个主题topic都有若干个分区。消费者组订阅一个topic后，每个消费者消费指定的分区数据。很明显并发量大大增加（只不过不能保证全局消费顺序）。

批量发送：producer发送消息回先缓存在本地，等到固定条件到后在发送到broker。

操作系统篇

1. 死锁产生的四大条件
   1. 存在互斥条件
   2. 请求并等待
   3. 不可被剥夺
   4. 循环等待
2. 处理死锁的方法
   1. 预防死锁

预防死锁是通过破坏死锁产生的条件来达到处理死锁的问题（破坏“请求并保存”、破坏“不可剥夺”、破坏“循环等待”）

* 1. 死锁避免
     1. 避免死锁：将系统分为安全和不安全状态，**运行的进程申请资源，先有计算机判断此次资源分配是否安全，若不会导致不安全状态，这回将资源分配给该进程。**
     2. 典型的避免死锁的算法：**银行家算法**
  2. 检测死锁
     1. 检测死锁就是通过**银行家算法，**判断系统是否安全
  3. 解除死锁：
     1. 剥夺资源：从其他进程剥夺资源给当前死锁的进程。
     2. 终止或撤销进程：可以直接终止当前进程，等待资源足够时在开启进程。

1. 什么是进程
   1. 进程是操作系统分配资源的最小单位。一个进程就是一个程序的执行，进程分配有独立的空间。
   2. 进程状态：创建态、运行态、就绪态、阻塞态、终止态。
   3. 进程的特点：
      1. 独立性：进程拥有自己的内存空间
      2. 并发性：同一时间有多个进程运行在操作系统中
      3. 异步性：进程之间相互隔离
      4. 动态性：进程是程序运行时的状态。
2. 进程和线程的区别
   1. 线程是操作系统最小的调度单元，进程是操作系统最小分配资源的单位。
   2. 进程在操作系统中拥有独立的空间（堆、栈等）。线程归进程管理，一个进程拥有多个线程，同一个进程中的线程共享堆区数据，但同时线程们都有自己独立的栈空间，互相隔离。

Spring篇

1. Spring是什么
   1. Spring是java的一个IOC、AOP轻量级框架。**目的是为了让程序员更注重业务代码。**
2. 什么是IOC
   1. IOC概念
      1. 控制反转：旨在将对象交给spring容器管理（对象的创建、销毁）。并且有spring来管理对象之间的依赖。简单莱索就是：**以前我们需要通过new来创建对象，而现在由spring容器创建、查找对象，最后通过依赖注入将对象给我们用。**
   2. IOC容器的初始化（非常重要）
      1. **Ioc中一些概念**
         1. BeanFactory：Spring的本质就是bean工厂（beanFactory），beanFactory会在bean生命周期的各个阶段中，对bean进行管理。有很多的实现类（比如**DefaultLisableBeanFactory**）。
         2. ApplicationContext:
            1. 它是BeanFactory派生而来的。**ApplicationContext实际上是代理增强了BeanFactory。**
            2. 如果说BeanFactory是ioc容器，那么ApplicationContext是Spring的容器，spring的上下文。**ApplicationContext内持有BeanFactory对象（会new一个DefaultLisableBeanFactory）。**
            3. ApplicationContext可以直接通过String路径读取到BeanDefinition信息，而BeanFactory需要通过reader来读取很麻烦。
         3. BeanDefinitionReader：它可以从bean.xml文件、被@Component系列注解的类、@Configuration注解的配置类等，获得BeanDefiniton，然后注册到BeanFactory中。有很多的实现类：xml的、annotation的、configuration的等。**处理不同情况下的bean信息读取。**
         4. BeanDefinitionRegistry：它可以将beanDefinition注册到BeanFactory中，**存储在BeanFactory的一个ConCurrentHashMap中。**
         5. BeanDefinition：它是String Bean的建模对象，就相当于类和对象的关系。它包含bean的一些元数据信息。Beandefinition从xml文件中读取并解析后的到。
      2. Ioc容器初始化包括三个过程：beanDefinition的Resource定位、beanDefinition的载入、beanDefinition的注册。
      3. BeanDefinition的Resource定位
         1. **第一种方式：**编程式使用DefaultListableBeanFactory（beanFactory的实现类也就是ioc容器的一个实现类）。首先BeanDefinition的信息保存在xml文件中，我们需要一个ClassPathResource定位，但是defaultListableBeanFactory不能直接读取，**需要Resource读取器XmlBeanDefinitionReader处理。**
         2. **第二种方式**：不使用DefaultLisableBeanFactory，而是使用ApplicationContext。**ApplictionContext可以直接通过String类型的路径获得xml文件路径。而BeanFactory需要reader来读取。**
      4. BeanDefinition的加载和注册
         1. ClassPathApplicationContext是ApplicationContext的一个子类。
         2. 以ClassPathApplicationContext为例。其构造函数内会调用**setConfigurations()方法完成对beanDefinition的加载，并且会解析为一个个的BeanDefinition**，然后**会执行核心方法refresh()。**
            1. refresh()方法标志着ioc容器正式启动。
            2. 然后refresh()方法中调用核心方法obtainFreshBeanFactory()，
            3. 然后obtainFreshBeanFactory()中调用refreshBeanFactory()方法
            4. refreshBeanFactory()方法时最关键的。这个方法里面：如果已经存在beanFactory就先销毁，如果没有就会调用createBeanFactor()方法，创建一个DefaultListableBeanFactory对象。**然后调用loabBeanDefinitions()方法将beanDefinition注册到beanFactory中。到这里就结束了。**
3. ApplicationContext和BeanFactory的区别
   1. ApplicationContext和BeanFactory都是ioc容器，只不过APPC初始化ioc容器时不需要向BeanFactory那样麻烦。
   2. APPC其实就是通过代理模式增强的BeanFactory，本质上在APPC类中，refresh()方法也创建了BeanFactory对象。
4. Spring bean的生命周期
   1. 只有当程序员主动调用getBean(name)方法获得一个bean时，才会触发bean实例化阶段的活动。（或者这些注解@Autowired自动注入时）
   2. Bean的生命周期主要包含五个阶段：**实例化Bean、Bean属性填充、初始化Bean、使用bean、销毁Bean**
      1. 实例化Bean：当客户向容器申请一个容器尚未拥有的bean时（或者初始化bean时，该bean需要另一个bean也未初始化时），容器就会调用doCreateBean()方法进行实例化，**实际就是通过反射的方式创建出一个bean对象。Spring在实例化bean时，使用的是对象默认的空参构造函数。**
         1. **BeanFactoryPostProcessor：**实现BeanFactoryPostProcessor接口的类，可以用于定制bean Factory，所有的bean definition被加载后，都还没有对应的bean都还没有被实例化，**所以简言之实现了这个接口的类，可以在对bean进行实例化之前完成对bean的改动。**
      2. Bean属性填充：bean实例化后，需要对bean的属性进行填充，同时注入bean依赖的其他bean对象。
      3. Bean初始化：bean初始化阶段又有很多事儿
         1. 执行Aware接口的方法
            1. **什么是Aware系列接口**：这些接口可以让我们，拿到Spring容器的一些信息。比如BeanNameAware接口中的getBeanName()方法可以获得该bean在容器中的名字、BeanFactoryAware接口中的getBeanFactory可以获得工厂对象BeanFactory。
            2. Spring会检测该bean对象对应的类，是否实现了xxxAware接口，如果实现了，就可以拿到Spring容器的一些资源。
         2. 执行前置处理器postProcessorBeforeInitialization方法。
            1. **BeanPostProcessor是Spring IOC容器提供给我们的扩展接口，让我们在bean实例化、属性填充后，在执行bean的初始化方法前，允许我们修改bean的属性等。**
            2. 如果有实现BeanPostProcessor接口（同时要注册到bean.xml中）。就会执行前置处理方法postProcessBeforeInitialization()，对bean进行一些自定义的前置处理。
         3. 调用了bean的初始化方法（init-method），如果有指定的话。
         4. 执行后置处理器postProcessorAfterInitialization方法：如果有实现BeanPostProcessor接口，就会执行后置处理器的postProcessAfterInitialization方法。
      4. 使用bean对象。
      5. 销毁bean：
         1. 如果有实现DisposableBean接口的话，并且有写在bean.xml，那么在销毁bean时，会调用DisposableBean接口中的destroy方法。
            1. DisposableBean接口和BeanPostProcessor接口一样，为bean提供了释放资源的方式。
         2. 如果指定了destroy-method等方法的话，则会执行这些销毁方法。
5. Spring下有哪些容器
   1. 原始的BeanFactory：有很多子类，加载XML的、加载注解的
   2. ApplicationContext：它相当于是通过代理增强版的BeanFactory。容器启动流程没有BeanFactory那样麻烦。
6. Spring循环依赖是什么，如何解决
   1. Spring循环依赖的相关概念
      1. 就是对象A依赖对象B，并且对象B依赖对象A
      2. **不考虑Spring的话，单纯new对象A和对象B，即时AB相互依赖，已没有任何问题，**因为new不像Spring中getBean那样获得一个对象那样麻烦。Spring中bean生命周期就很麻烦。
      3. Spring生成一个对象的步骤：
         1. 扫描class的BeanDefinition文件，然后通过反射得到一个**原始对象（这个概念很重要）。**
         2. 然后对原始对象进行依赖注入
         3. 如果原始对象的某个方法被**AOP了，那么需要根据原始对象生成一个代理对象**
         4. **最后将最终对象放入singletonObject中，下次getBean可以直接获得**
      4. 属性依赖注入环节：**getBean(A)时，生成一个A原始对象。**然后对A进行属性依赖注入，此时有一个属性是B类属性。
         1. **如果BeanFactory中有B，那么直接给A注入**
         2. **如果BeanFactory中没有B，那么需要先生成B对象。**
         3. **如果是第二种情况的话，那么就导致AB都无法被创建出来。**
   2. 三级缓存解决Spring循环依赖
      1. 三个缓存：
         1. SingletonObject：放已经经历完整生命周期的对象（大概就是已经完成了依赖注入的对象）。
         2. EarlySingletonObject：大概就是放**原始对象（没有进行依赖注入的对象）。**
         3. SingletonFactory：缓存ObjectFactory，也就是生成原始bean的工厂（这个缓存主要是为了服务解决循环依赖中的AOP问题）。
      2. 普通循环依赖（不包含aop）解决思路：
         1. Spirng生成了原始对象A后，将原始对象A放入earlySingletonObject缓存中。然后进行依赖注入
         2. A对象依赖注入时，发现依赖了B对象。**首先，Spring会去singletonObject缓存中找，如果没有，再去earlySingletonObject中找，如果还没有那么此时Spring再生成一个原始对象B。**
         3. 然后对B进行依赖注入，发现原始对象B依赖A类对象，那么首先去singleObject中找，如果没有，再去将earlySingleObject中将原始对象A取出，注入B对象的属性A。
         4. 然后对象B就执行了整个生命周期了，就会被放到singletonObject中
         5. 最后Spring会将对象B注入到A对象中，A也完成了生命周期，也会被放到singletonObjective中。
   3. 存在AOP代理的Spring循环依赖解决方法：
      1. 普通循环依赖只需要一个earlySingleObject就可以完美解决了，为什么还需要singletonFactory呢？
      2. 如果遇到了aop情况的话，A还是将原始对象放入earlySingletonObject中，然后B直接获取A原始对象进行依赖注入。**最后A的原始对象进行了AOP代理后生成了了代理对象**，此时会出现A的真正对象是代理对象，而B中A的依赖不是A的代理对象。
      3. 解决方法：
         1. 在bean的生命周期中，如果生成了原始对象后，同时需要构造一个ObjectFactory放入singletonFactory中。
         2. ObjectFacotyr是一个函数式接口，**() ->getEarlyBeanReference(beanName, mbd, bean)。**
         3. **加入其他对象依赖了A对象，且A的原始对象已经在earlySingtonObject中，那么此时并不是直接返回原始对象，而是通过在获得A原始对象的ObjectFactory，然后返回代理对象（如果有A对象存在AOP代理的话）。**
7. **Spring 中，基于 xml文件的注入bean方式有几种**
   1. **Setter注入、构造器注入、静态工厂注入、实例工厂**
   2. **这四种方式，前两种用的较多。**
8. **Spring中的事务**
   1. 事务分类：Spring提供了两种事务实现方式
      1. 编程式事务，它允许客户在**自己的代码中，精准的确定事务的范围**，需要手动提交和手动回滚。**编程式事务具有代码侵入性，但是提供了更详细的事务管理。**
      2. 声明式事务：它利用**@Transaction注解实现的，底层机制是AOP机制**。帮用户将事务和业务解耦，不需要用户手动commit或者rollback。
   2. 下面主要讨论声明式事务
      1. Spring事务基本流程。
         1. @Transaction主要的作用：当在某一个类中的方法上使用了@Transaction注解后，**Spring会基于这个类生成一个代理对象，将这个代理对象作为bean放在容器中。**
         2. 当使用这个代理对象时，如果这个方法加上了@Transaction注解，那么代理对象会利用事务管理器创建一个数据库连接，**将数据库中的autocommit修改为false，禁止自动提交。**
      2. Spring事务相关概念
         1. 事务管理器
            1. Spring不是直接管理事务的，**Spring提供了事务管理接口** org.springframework.transaction.PlatformTransactionManager。**具体的实现由jdbc、hibernate等这些持久化机制提供。**
            2. **以jdbc为例**

**Jdbc给Spring提供的实现是**org.springframework.jdbc.datasource.DataSourceTransactionManagers（也就是最开始学习，总是要添加的一个bean）。

* + - * 1. **事务定义属性TransactionDefinition**

**Spring通过**PlatformTransactionManag接口中的getTransaction(TransactionDefinition definition)获得事务。

**TransactionDefinition就定义了事务的一些属性。**

**通过在@Transaction注解中定义这些事务属性，最后会交给TransactionDefinition。**

* + - * 1. **什么是事务属性呢？有五个方面（很重要）**

**传播行为**

就是一个A事务方法被另一个B事务方法调用时，被调用的事务方法应该如何进行（是继续在A方法的事务中进行，还是B方法自己开启一个新事务）。

传播行为可以在@Transaction中自己设置，有很多选择。**Spirng默认选择是：PROPAGATION\_REQUIRED如果当前方法存在事务，那么被调用的事务方法，会在当前事务中进行进行，如果调用方没有事务，那么被调用的事务方法回自己开启一个新事务（有一个事务失败，则全部回滚）。**

**隔离级别**

可以在@Transaction中定义事务隔离级别，

Spring默认是默认隔离级别选择是：和数据库使用的相同

只读

可以在@Transaction中设置当前事务为**只读事务。**

**如果设置为了只读，那么如果当前事务有增删改的话，会报异常。**

**事务超时**

可以在@Transaction中设置事务超时时间。

**回滚规则**

在@Transaction中设置，当前事务回滚的规则（定义那些异常会导致事务回滚）。默认是任何事务都会回滚。

1. **Spring事务中，那些情况下回事务会失效**
   1. 数据库不支持事务：比如innodb引擎支持事务，而myisam不支持事务。
   2. 事务方法没有被Spring管理：类上没有加@Component、@Service等注解，那么该**类就不是Spring容器中的bean，**那么事务也就失效了。
   3. 方法没有被public修饰：私有方法，即使加了@Transaction注解也会失效。
   4. **同一个类中，非事务方法调用事务方法**：若同一个类中，A方法没有事务修饰，但是B方法有事务，如果A方法调用B方法，那么B方法事务会失效。
   5. 未配置事务管理器：比如使用mybatis，没有增加jdbc的事务管理器DataSourceTransactionManager。
   6. 不正确的异常捕获：事务方法中，异常被try catch捕获了，那么事务也会失效。
   7. 错误的标注异常类型：@Transaction注解中可以标注，那些异常需要回滚，默认是RuntimeExeption，如果是Exeception异常，事务就失效了。
2. Spring AOP
   1. AOP是什么
      1. 翻译过来是“面向切面编程”，主要是最oop“面向对象编程”的补充。实现在不修改源代码的情况下，对方法添加新功能（比如Spring事务就是通过@Transaction注解，通过aop生成一个代理对象）。
   2. AOP实现原理
      1. Aop通过拦截方法，然后对方法进行增强。
      2. AOP底层是利用**动态代理（jdk动态代理、cglib动态代理）去实现。**
   3. AOP中几个概念：
      1. JointPoint：**连接点，**指需要切入的点，就是那些去要被代理的类的方法。
      2. Pointcut：**切入点，**相当于一个匹配规则，匹配那些JointPoint需要被拦截代理增强。
      3. Advice：**通知，**指被拦截jointPoint需要做的事，也就是对拦截方法增强的内容。
      4. Target：**目标，**指代理对象
      5. Weaving：**织入，**指把增强代码引用到目标上，生成代理对象的过 程。
      6. Proxy：**生成的代理对象，**
      7. **Aspect：切面，也就是切入点和通知的结合。**
   4. Aop织入时期
      1. 编译期
      2. 类加载期
      3. 运行期
   5. Advice类型
      1. Before：前置通知
      2. After：后置通知
      3. Around：环绕通知
      4. AfterReturn：返回后通知
      5. AfterThrowing：异常后通知

SpringMVC篇

* + - 1. SpringMVC执行流程
         1. SpringMVC中重要组件

DispatcherServlet：**前端控制器**，用于接收前端发来的请求、相应结果，相当于转发器。帮助其他组件解耦。

HandlerMapping：**处理器映射器**，根据DispatcherServlet接收到请求中的url寻找对应的Handler。

HandlerAdapter：**处理器适配器**，负责执行Handler。

Handler：**处理器**，其实就是我们写的@Controller类。

ViewResolver：**视图解析器**，将ModelAndView解析为View。

View：View是一个接口，它的实现类，支持不同的视图（jsp、freemark）。

* + - * 1. SpringMVC的执行流程

前端用户发送请求到DispatcherServlet

DispatcherServlet接收前端request请求，然后调用HandlerMapping，查询对应的Handler

HandlerMapping根据请求中的url查询对应的handler，并且生成对应的拦截器（如果有的话），然后交给DispatcherServlet。

DispatcherServlet调用HandlerAdapter。

HandlerAdapter调用处理器Controller。

自定义处理器Controller将得到的参数处理，并且返回结果给HandlerAdapter。

HandlerAdapter将得到的结果交给DispathcerServlet

DispatcherServlet将ModelView交给ViewResolver

ViewResolver解析MovelView后，将View交给DispatcherServlet

DispatcherServlet调用物理视图将view进行渲染并返回。

最后DispatcherServlet将结果返回给前端用户。

**SpringBoot**

SpringBoot自动装配

概念：自动装配，它是starter的基础，也是SpringBoot的核心。

什么叫装配：把bean托管给Spring IOC管理，就叫做装配。

@import注解：这个注解可以把多个分开的容器配置合并到一个配置中加载。@import注解里面可以填写类（.class）、importSelector、importBeanDefinitionRegisor

ImportSelector：第一种方式，如果导入的是一个配置类（@Configuration注解的类），那么该类中的所有Bean都会被导入。为了更灵活，那么就可以导入ImportSelector的实现类。实现Importelector接口中的selectImport()方法，这个方法返回String[]，这个String[]数组中，就可以灵活的传入自己想要导入的类的名字（.class.getName()）。

ImportBeanDefinitionRegistor：这种方式和ImportSelector方式很像。同样要实现ImportBeanDefinitionRegistor接口中的ImportBeanDefinitions()方法。

深入分析@EnableAutoConfiguration注解

@EnableAutoConfiguration注解，在SpringBoot启动类上的，当SpringBoot启动时，这个注解就会生效

@EnableAutoConfiguration注解的实现上，[又有一个@Import(EnableAutoConfigurationImportSelector.class)](mailto:又有一个@Import(EnableAutoConfigurationImportSelector.class)),学了前面的@Import注解引入ImportSelector接口实现类就懂了。

EnableAutoConfigurationImportSelector类中的**selectImorts()方法就是关键，返回String[]就是SpringBoot启动时需要加载的所有三方库中的bean了。**

selectImorts()方法中，有很多逻辑处理（**主要就是帮springboot应用把符合@Configuration配置都加载到当前SpringBoot创建的IOC容器中）。**

**SpringFactoriesLoader：它的作用就是从classpath/META-INF/spring.factories文件中，根据key来加载对应的类到spring ioc容器中。**

JVM篇

1. 分代gc算法
   1. 流程
      1. 对象回收一般出现在堆区。Jvm将堆区分为**新生代和老年代。新生代又分为：伊甸园区、存活1区、存活2区。**
      2. 新对象一般在伊甸园区被创建，如果伊甸园区空间满后，会触发gc算法，将被标记的垃圾回收后，最后将伊甸园数据复制到存活1区或者存活2区（非空的一个）。
      3. 如果存活1区或者存活2区满后（非空的一个），又会触发gc算法。然后没有被清除的对象，对象头的gc次数会加一。然后将非空存活区的数据复制到空存活区。
      4. 如果gc次数达到一个阈值，就会被复制到老年代区。
   2. 性能：因为大部分的对象都会在新生代的伊甸园区被回收掉，所有在存活区复制是少量的，不会很慢。
2. gc如何确定回收对象
   1. 引用计数算法：对象被引用后，计数加一。但是无法解决对象互相引用，导致内存泄漏。
   2. 可达性分析算法：从gc roots出发，类似一颗树遍历每一个可达的对象。当一个对象没有被引用，那么就会被gc回收。
      1. gc roots如何确定：栈中的局部变量引用对象、方法区静态变量引用的对象、方法区常量引用的对象。
3. gc如何清理垃圾对象
   1. 标记-清除算法：直接将垃圾清理，会造成碎片内存。
   2. 标记-复制：堆区分两块区域，一块区域用于复制数据（空的），清理另一块有数据的区域，然后将存货对象复制到空的区域。（复制时间很长）。
   3. 标记-整理：直接清楚垃圾对象，然后整理内存，解决内存碎片问题。
   4. Gc分代算法（上面有）
4. 垃圾收集器：
   1. Serial：最老的垃圾收集器了，**单线程的垃圾收集器**，**工作在新生代**，采用**标记-复制算法**。在gc时，**会导致用户线程暂停，stop the world；**
   2. Serial Old：工作在老年代的Serial垃圾收集器，**采用标记-整理算法**。
   3. ParNew：多线程版本的Serial垃圾收集器，充分利用cpu，但是会导致stop the world；
   4. Parallel Scavenage：也叫做吞吐量最优的垃圾收集器。什么时吞吐量就是100分钟，有99分钟用户线程在执行，gc线程只有1分钟执行，你们吞吐量就时99%，**工作在新生代**，**采用标记-复制算法**。
   5. Parallel Old：时Parallel Scavenage的老年代收集器，采用**标记-整理算法**
   6. CMS：Concurrent Mark Sweep（并发标记清除算法），实现用户线程和gc线程同时工作，目的是：**降低用户线程停顿干，减少STW，注重响应速度。**
      1. 包括四个阶段
         1. 初始标记：标记线程标记所有gc roots可达的对象，很快但是回发生STW；
         2. 并发标记：用户线程和垃圾回收线程同时运行，会标记gc roots的可达对象，但是可能会漏标对象，因为用户线程可能会修改地址。
         3. 重新标记：垃圾回收多线程修正并发标记过程中，被用户线程修改标记的对象，速度很快，但是会发生STW；
         4. 并发清除：用户线程和垃圾回收线程同时执行，清理垃圾对象。
      2. 缺点：使用的标记-清理算法，会有内存碎片。
      3. 优点：用户线程会和垃圾清线程部分同时进行，STW发生少。
   7. Garbage First：G1垃圾收集器，是JDK9默认垃圾收集器，**同时关注低延迟和高吞吐量。**
      1. G1收集器，将堆分为多个大小相同的region，并且维护了一个优先列表（跟踪每个region回收需要的时间以及回收能释放的内存大小），如果时间越少并且释放内存越多，那么G1下次回收会优先考虑这个region。
      2. G1进行收集时，也会经历**初始标记、并发标记、重新标记、并发清理**四个阶段，这种方式下，STW时间很少。
      3. 优点：新生代和老年代都可以回收。STW时间少，并且吞吐量高。
5. Young GC和Full GC
   1. Young GC
      1. 指年轻代的垃圾回收，通常是Eden区满了后，会触发。
   2. Full GC
      1. 指整个堆区的垃圾回收，比较耗时
      2. 触发条件：
         1. 老年代满了，或者新生代对象晋升为老年代时，老年代空间不足。
6. jvm内存结构：**一般研究的是运行时数据区**
   1. 程序计数器：是一块很小的内存空间。每个线程都有自己的程序计数器。程序计数器作用就是指向下一条需要执行的指令。不会发生oom。
   2. 栈区
      1. Java虚拟机栈
         1. 栈中保存什么：栈帧是栈中的元素。每个方法在执行时都会有栈帧。
         2. 栈帧中保存了什么：局部变量表、操作数栈、动态连接、返回信息
            1. 局部变量表：它用于存储局部变量包括：基本数据类型、对象引用、
            2. 操作数栈：栈中数据可以是任意java类型。它是栈帧中用于保存计算的临时数据存储区。**它是保证了先进后出。**
            3. **动态链接：**每个栈帧中都包含了指向运行时常量池中该栈帧的变量、方法的引用（栈帧中调用其他方法）。**动态链接就是将方法的符号引用直接转化为方法的直接引用。**
      2. 本地方法栈：
         1. 被native修饰的方法都是本地方法。这些方法都是有c或c++完成的。可以说是直接和操作系统交互。
   3. 堆区
      1. 堆区概念：是jvm内存最大的一块地方，主要是存放new出来的对象，所以是gc的主要区域。
      2. 堆区分为：年轻代、老年代。 年轻代分为：伊甸园区、存活1区、存活2区。
      3. 老年代存放长期存活的对象，如果老年代满后，会触发full gc，会清理这个堆区，清理后如果还是无法存放新对象，那么就会触发oom。
   4. 方法区（元空间）
      1. 在jdk1.8后，jvm用元空间代替了方法区。**而且不是使用的jvm内存，而是使用本地内存。**
      2. 方法区存放了：加载类信息、常量（字符串）、静态变量、字面量。
      3. 说简单一点就是：常量+静态变量+类信息（版本、方法、字段等）+运行时常量池。
         1. 运行时常量池：存放了各种字面量（基本数据类型的值、字符串）和符号引用（类和接口的全类名、方法名、字段名）
         2. 常量池作用：**避免了对象频繁销毁和创建产生对性能的影响**
7. 指针碰撞，空闲列表。
   1. 当new一个对象是，需要堆区的内存空间，如何分配空闲内存给这个对象也就引出这两个概念。
   2. 指针碰撞：就是内存的空间是连续的，一边是已经使用的内存，另一边是空闲的内存，中间有个分割指针。当需要分配空间时，就直接将指针向空闲内存移动需要的内存大小。
   3. 空闲链表：当内存时不连续时，空闲的区域在空闲链表中记录着的，需要内存时，直接查看空闲链表即可。
8. 类加载过程
   1. 类文件结构：
      1. .class文件包括：魔数、类信息、接口信息、方法信息、字段信息、**常量池。**
      2. 常量池包括：字面量、符号引用（在类加载时，解析过程会将符号引用替换为真实的地址）
   2. 类加载过程：加载、连接（验证、准备、解析）、初始化、
      1. 加载：.class问价只有加载到方法区，才能被使用。会在方法区生成java.lang.Class对象，通过Class对象，就可以访问到方法区的field、method等消息。
      2. 连接：验证安全信息。准备给**static静态变量**分配空间并赋予初始零值（如果是常量，就直接赋予指定值）。解析是将常量池中的符号引用替换为真实的引用地址。
      3. 初始化：收集从上到下的**静态代码块、静态变量赋值，合并成一个cinit()方法**执行。
9. new一个对象的过程
   1. 类加载：首先判断new的参数这个类是否已经被加载到了方法区，如果没有首先需要现将类加载到方法区，然后生成一个Class对象，最后解析符号引用、初始化静态代码。
   2. 为对象在堆区分配内存：指针碰撞、空闲链表两种方式。
   3. 初始化数据：
      1. 对象头数据：gc分代年龄、synchronized锁monitor、类型指针（指向该对象在方法区的类Class对象）。
      2. 实例数据：程序中对该对象中字段的赋值
      3. 内存对齐
10. 双亲委派机制
    1. 就是在类加载器加载class文件时，类加载器不会直接加载，而是将这个class文件加载任务交给上层的类加载器加载。
    2. ApplicationClassLoader回交给ExtentClassLoader加载，
    3. 然后extendClassLoader又交给BootstrapClassLoader加载，
    4. 如果上层无法加载，那么就返回给下层加载。
       1. 这也导致了用户自定义String类，是无法使用的，因为BootstrapClassLaoder会加载核心类库，核心类库中的String才是最终被加载到jvm中的String类。

SpringCloud相关微服务篇

* + - 1. SpringCloud和Dubbo的区别：
         1. Dubbo是RPC调用框架，而SpringCloud是restFul风格的调用框架。
         2. Dubbo的注册中心是zookeeper，而SpringCloud一般利用Eureka作为注册中心。
         3. Dubbo是比较轻量级的，也就是说微服务中的降级、限流、网关等功能dubbo并没有实现，必须通过三方技术整合。而SpirngCloud是微服务一站式解决方案，支持zuul网关、hytrix降级等这些功能。
      2. 有哪些服务注册中心
         1. Zookeeper：是一个分布式协调服务

一致性协议：zab协议

Zookeeper集群节点有主从之分，主节点挂了需要leader选举。

CAP支持：zab支持CP。也就是强一致性、分区容错性。

缺点：不能支持高可用，主节点挂了，leader选举很浪费时间。因为是强一致性的，所以leader挂了，那么zookeeper集群是不可用的。

* + - * 1. Eureka：

一致性协议：eureka并没有使用一致性协议，而是利用p2p模式进行集群通讯。

Eureka没有主节点和从节点之分，集群中的节点都是平等的。

CAP支持：eureka支持AP。也就是高可用、分区容错性。

相较于zookeeper，eureka中只要有一台节点，那么就是可用的（因为eureka是高可用的）。

* + - * 1. Etcd：利用go语言编写的

一致性协议：raft协议

Etcd有主从节点之分，毕竟是通过raft协议完成的。

CAP支持：支持CP，也就是强一致性、分区容错性。

* + - * 1. Consul：利用go编写的

一致性算法：raft算法

CAP支持：支持CP，强一致性、分区容错性。

* + - 1. Ribbon和Feign的区别：
         1. Ribbon和feign都是Netflix的SpringCloud提供的两种实现了负载均衡的restFul风格的远程调用组件。
         2. Ribbon：是最原始的负载均衡远程调用，通过restTmplate模拟http请求，调用远程其他微服务的接口。步骤很繁琐。
         3. Feign：是在Ribbon基础上进行了改进，是的服务调用非常便捷。

采用接口的方式，将其他微服务上，需要调用的方法定义为接口的形式就好了，然后加上@FeignClient注解即可。

注意：**接口的定义必须一样。**

* + - * 1. **Spring-cloud-starter-feign中包含了spring-cloud-starter-ribbon模块了。（也就是，feign是基于ribbon实现的）**
      1. 服务降级和服务熔断
         1. 服务降级：面对当前系统压力剧增的情况下，**对部分不重要的服务不处理、或者简单处理**，从而释放资源保证**重要的服务**正常运行。

降级的情况：

超时降级：主要配置好了超时时间和超时重试机制，如果某个请求到达某一个微服务处理时间达到了超时时间、或者重试次数，那么该微服务就会被降级。

故障降级：如果调用远程其他服务失败（rpc调用），那么可以直接降级。（也就是服务熔断，触发降级）

* + - * 1. 服务熔断：在分布式微服务环境下，浏览器一个http请求，到后端处理可能经过多个微服务之间的调用，**如果某个微服务的某个api突然不可用了，那么会导致这一条调用链直接服务雪崩。**

通过路断器的故障监控，发现某个服务单元不可用后，直接熔断整个服务。**想调用方返回一个符合预期的fallback，而不是长时间的等待导致服务雪崩。**

* + - * 1. 共同点：

都是为了实现系统的高可用，防止系统缓慢甚至崩溃。

* + - * 1. 不同点

触发条件不通：服务降级是对整个系统来说，为了保证重要服务正常运行，而将不重要的服务暂时降级。 服务熔断是对某一条微服务调用链来说，调用链上某个服务不可用，可以直接熔断，想调用方返回fallback，以免导致服务雪崩。

* + - * 1. 总结：

服务熔断一定伴随服务降级（出现故障的服务），熔断也是降级情况的一种。熔断是对链路的保护，而降级是对整个系统的保护。qinliang