1. 让你来写一个消息队列，该如何进行架构设计

分布式消息

采用哈希算法，取模，一致性哈希，对2^32取模，沿着顺时针方向的最近的一个位置，服务器映射到哈希环中，引入虚拟节点避免服务器映射不均匀的问题，一般将虚拟节点数设为32

消息队列：提高并发量，但不需要立即获得结果的情景

使用场景

* 应用耦合：多应用间通过消息队列进行处理，避免调用接口失败导致整个过程失败
* 异步处理：应用间并发处理消息，减少处理时间
* 限流削峰：应用于秒杀或抢购活动中，避免流量过大导致应用系统崩溃
* 消息驱动系统：

异步处理：比如注册过程种，系统需要发送注册邮件和并验证短信

应用耦合：用户使用QQ相册上传一张图片，人脸识别系统会对该图片进行人脸识别

限流削峰：秒杀活动种，大量用户请求，业务处理来不及

消息驱动系统：多个调用

消息队列设计的考虑：可扩展性、持久化存储、可用性、数据丢失

消息队列的设计涉及到：生产者（producer）、消息服务端（broker）、消费者（consumer）

服务端：把消息发送到接收端，涉及到消息存储、通讯机制

Broker的设计重点

* 消息转储：合适的时间投递，通过某些手段保证消息最终达到消费者
* 规范——通用模型：保证最终一致性，错峰
* 消息转发器，把一次RPC做成两次RPC，发送者把消息投递到Broker，Broker再将消息转发给接收端，两次RPC+1次转储
* 保证消息可达：需要做消费确认，则需要三次RPC

三个基础功能：消息传输、存储、消费

涉及到三个方面的设计：通讯协议、存储选择、消息关系的维护

传统的通讯协议标准由XMPP和AMQP，为了提高性能也可以使用自己设计的通信协议

* JMS：Java MessageService，为Java提供了同一个消息操作，包括创建消息、发送消息和接收消息等。
  + 提供了两种模型：点对点、发布订阅模型
  + 点对点模型，消息将发送到一个队列，该队列的消息只能被一个消费者消费
  + 发布订阅模型，生产者和消费者独立
* AMQP：高级消息队列协议，是一个标准化的消息中间件协议，目标是让不同系统的应用相互通信，并提供了简单的统一模型和编程接口。目前ActiveMQ和RabbitMQ都支持AMQP协议
* Kafka通信协议，基于TCP的协议

存储选择：内存、本地文件系统、分布式文件系统、nosql、DB

内存最快，但消息会丢失，消息堆积能力不高的场景

DB是最简单的实现可靠存储的方案，最终一致性

理论上，速度，文件系统>分布式KV（持久化）>分布式文件系统>数据库，而可靠性相反

还需要支持事务、顺序消息、消息重试等功能

1. CAP原则和BASE理论

CAP原则是指Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可兼得

最终一致性

CA without P：不要求P，不允许分区，则强一致性和可用性可以保证，但是分布式节点受限

CP without A：不要求A可用性，则会导致同步时间很长，当网络故障时，会牺牲用户体验

AP without C：高可用并允许分区，需要放弃一致性。会导致全局数据的不一致性，比如抢购系统，常见的mysql主从复制保证了最终一致性

BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的简写

Mysql主从复制保证了CAP中的AP

基本可用是指在系统发生故障时，允许损失部分可用性

软状态：允许系统中数据存在中间状态，且中间状态不会影响系统的整体可用性

最终一致性：在经过一段时间的同步后，数据最终能够达到一个一致的状态

1. 爬虫与反爬机制

* Headers and referer 反爬机制
* ip限制，限制ip访问频率和次数进行反爬，解决措施：构造IP代理池，后每次访问时随机选择代理
* UA限制：UA使用户访问网站时候的浏览器标识，可以在requests 设置timeout
* 验证码反爬虫或者模拟登陆
* Ajax动态加载
* cookie限制：打开网页会生成随机cookie，如果再次打开网页这个cookie不存在，那么再次设置，第三次打开仍然不存在，这就非常有可能是爬虫在工作了

1. select ... for update加锁时，宕机时锁会释放吗？会的话，原理是什么？

会释放，原因是这时宕机后，根据事务的一致性，该事务会回退，因此会释放这个锁。

1. mysql 间隙锁（gap lock）

在以前的mysql在Repetable read（可重复读）级别下，还会出现幻读，但是最新的版本不会了，原因就是多了个gap锁

Gap锁会锁住扫描的范围

幻读是指在一个事务中，读取到另一个事务插入的数据，但是另一个事务还没有提交。

当事务需要更新一条记录的时候，GAP对这条记录之前和之后加锁。

Mysql中的insert、update、select … in share mode、select … for update，当前读会在所有扫描到的索引记录上加锁，不管它后面的where条件到底有没有命中对应的行记录。当前读可能会引起死锁。

意向锁：

InnoDB的意向锁主要用于多粒度锁并存的情况。比如事务A要在表上加S锁，如果表中的一个行被事务B加了X锁，那么该锁的申请也会被阻塞。如果表中的数据很多，逐行检查所标志的开销很大，系统的性能将会受到影响。因此引出了意向锁概念，解决这种情况。

意向锁的作用：事务A在更新记录前，先加意向锁，再加X锁。事务B先检查该表上是否存在意向锁，存在的意向锁是否与自己准备加的锁冲突。如果冲突，则等待直到事务A释放，而无需逐条记录去检测，事务B更新表时，无需直到到底哪行被锁了，它只要知道反正有一行被锁了就行。

1. mysql中select \* for update中锁表的问题

InnoDB默认使用行级锁，只有明确的指定主键，Mysql才会执行Row lock（只锁住被选取的行），否则Mysql将会使用表级锁。

Select \* for update在RR隔离级别下的潜在死锁问题

1. 数据库的隔离级别是怎么实现的？

数据库隔离级别是通过锁来实现的

* 读未提交：事务在数据库修改数据的时候只对数据增加行级共享锁，读数据的时候不加锁。会导致脏读。
* 读已提交：事务在读取数据时使用行级共享锁，读完就释放；写数据时使用行级排他锁，事务结束才释放。会出现不可重复读现象
* 可重复读：事务在读数据的时候对行加行级共享锁，直到事务结束才释放；写数据的时候使用行级排他锁，直到事务结束才释放。能解决不可重复读，但是不能解决幻读问题。
* 可序列化：事务在读数据的时候，先加表级共享锁，直到事务结束才释放；事务在更新数据的时候，必须对其加表级排他锁，直到事务结束才释放。能解决幻读问题。

1. Mysql数据引擎：MyISAM和InnoDB两者之间的区别与选择

* MyISAM：默认类型，基于传统的ISAM类型，是存储记录和文件的标准方法，不是事务安全的，不支持外键，占用资源较少。如果不需要事务，且读数据较多或者写数据比较多，读写并发访问相对较低，对数据一致性要求不高，硬件资源比较差。
* InnoDB：支持事务安全引擎，支持外键、行锁，事务是InnoDB的最大特点，占用资源较高。如果有大量的update和insert，建议使用InnoDB，特别是针对多并非和QPS较高的情况。
* 表锁的差异
  + Myisam只支持表级锁，select、update、delete、insert都会给表自动加锁。
  + InnoDB支持事务和行级锁，能够保证事务的ACID特性，并且带来了几个问题。
* 索引差异
  + Mysiam允许没有主键和主键索引，InnoDB如果没有主键或非空唯一索引，会自动生成一个主键。
  + MyISAM支持fulltext类型的全文索引，InnoDB不支持full text类型的全文索引，但是InnoDB可以使用sphinx插件支持全文索引。

1. mysql主键索引和唯一索引的区别

主键索引一定是唯一索引，唯一索引并不一定就是主键。

一个表中可以有多个唯一索引，但是只能有一个主键

主键不允许空值，唯一索引列允许空值

索引可以提高查询速度

主键索引比较快

1. Mysql有哪些索引类型？

* 全文索引（fulltext），只有MyISAM引擎支持，用于解决like这类模糊查询的索引
* 哈希索引（hash），在=和in条件下高效，对范围查询、排序及组合索引效率仍然不高，hash值的唯一以及类似键值对的形式，可以一次定位，具有极高的效率
* B树索引（BTREE），一种将索引值按一定算法，存入一个树形结构的数据结构（B+树）中，每次查询都是从树的入口root开始，一次遍历node，获取leaf，最常用索引类型。
* RTREE：仅支持geometry数据类型，支持的存储引擎有MyISAM、InnoDB等。优势在于范围查询，多维数据，如空间数据，如地理位置。

索引种类：

* 唯一索引：加速查询+列值唯一，可以有null
* 主键索引：加速查询+列值唯一，表中只有一个
* 组合索引：多列组成一个索引，专门用于组合搜索，其效率远大于索引合并
* 全文索引：对文本的内容进行分词，进行搜索

Limit分页 Limit a,b; 表示仅获取第a条开始的b条数据

1. 红黑树与AVL树

红黑树的定义

* 是一棵搜索二叉树
* 每个节点不是黑色就是红色
* 根节点是黑色的，叶子节点是黑色的
* 如果一个节点是红色的，那么他的两个子节点必须是黑色的（保证没有连续的红色节点）
* 从根接节点到任意一个叶子节点的简单路径上，均包含相同数目的黑色节点

红黑树的性质：

* 任意简单路径上的红色节点数不超过黑色节点数
* 能保证最长路径长度不超过最短路径长度的2倍，任意两个路径的黑色节点数相同，而红色节点数不超过黑色节点数，从而保证任意路径长度不超过黑色节点数的2倍
* 由于红黑树是相对平衡的，因此红黑树查找时间复杂度是O(logn)
* 插入和删除的时间复杂度都是O(logn)

红黑树的插入

* 插入的节点必须是红色节点，插入后对树进行调整，按照不同情况对树进行调整
* 插入的位置必然是叶子节点的位置，不妨设插入的节点为cur，即插入后cur的左右子节点都是null，设其父节点为parent，祖父节点为pParent，叔节点为uncle
* 如果parent为黑色，则不需要调整；如果parent为红色，则pParent为黑色
* 如果uncle为红色，则只需要将parent和uncle改为黑，pParent改为红色，然后将cur设置为pParent，继续调整
* 如果uncle为黑色，且parent为pParent的左儿子，则进行右旋，即
* 如果uncle为黑色，且parent为pParent的右儿子，则进行左旋

红黑树的删除

* 红黑树要删除的节点有3种情况
  + 1 叶子节点：直接删除
  + 2 只有左子树或者右子树，直接删除
  + 3 既有左子树又有右子树，找到右子树的最左节点，替换后删除（3最终转换成1或2）
* 在情况2中，设要删除的节点为cur，则cur的儿子节点必为红色，否则无法满足性质5。从而cur的必为黑色
* 情况1 cur为红色，则cur的必为叶子节点，则可以直接删除cur，无需调整
* 情况2 cur为黑色，只有左子树或者只有右子树，此时cur的儿子节点必为红色，则只需要将cur的儿子节点改为黑色，并且替换cur即可。
* 情况3 cur为黑色，且cur为叶子节点，cur的兄弟节点bro为红色，替换p和bro颜色进行旋转
* 情况4 cur为黑色，且cur 为叶子节点，cur的兄弟节点为黑色，且兄弟节点的一个孩子节点为红色，
* 情况5 cur为黑色，且cur为叶子节点，cur的兄弟节点为黑色，且兄弟节点的两个孩子节点都为黑色，将兄弟节点变成红色，cur改为cur的父节点

1. B+树和B树

* B+树：所有key分布在叶子节点，只有叶子节点中有key和数据信息，所有的叶子节点形成一个链表，全局扫描只需要扫描整个链表。在非叶子节点中能存放更多的key，非叶子节点更小，数据存放更紧密，具有更好的空间局部性，因此访问叶子节点具有更高的缓存命中率。同时，能很好的支持范围查询
* B树：非叶子节点（内部节点）中也保存key和数据信息，不利于全局扫描。内部节点较大，空间局部性较差，不能很好的支持范围查询。
* B+树的优点：因为内部节点可以存更多的key，使得IO次数更少，从B+树的查询效率比B树高；B+树查询稳定，便于范围查询。

1. MySQL锁粒度分类？级别分类？

按粒度分：

* **行级锁**：是mysql中粒度最细的一种锁，只针对当前操作的行进行加锁。行级锁能大大减小数据库操作的冲突，其加锁粒度最小，但是加锁开销也最大。行级锁分为共享锁和排他锁。会出现死锁，发生冲突概率最低，并发度最高
* **表级锁**：mysql中粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分mysql引擎支持。表级锁分为表共享读锁和表独占写锁（排他锁）。
* **页级锁**：是mysql中介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。因此，采取了折衷的页级锁，一次锁定相邻的一组记录。并发度一般。

按级别分类：

* 共享锁（Share Lock）：又称为读锁，读取操作创建的锁。其它用户可以并发读数据，但是任何事务都不能对数据进行修改（获取数据上的排他锁），直到已释放所有共享锁。获取共享锁的事务只能读数据，不能修改数据。
* 排他锁（Exclusive Lock）：又称写锁，独占锁，如果事务T对数据A加上排他锁后，则其它事务不能再对A加任何类和的锁。获取排他锁的事务既能读数据，又能修改数据。
* 意向锁（Intention Lock）：是表级锁，其目的是为了在一个事务中揭示下一行将要被请求的锁类型。
  + 意向共享锁（IS）：表示事务准备给数据行加入共享锁，也就是说下一个数据行加共享锁前，必须先取得该表的IS锁；
  + 意向排他锁（IX）：表示事务准备给数据行加入排他锁。

MyISAM中是不会产生死锁的，因为MyISAM总是一次性获得所需的全部锁，要么全满足，要么全等待。而InnoDB中，锁是逐步获得的，造成了死锁的可能。Mysql中行级锁，不是直接锁记录，而是锁索引。若SQL语句操作了主键索引，Mysql就会锁住这条主键索引；如果SQL语句操作了非主键索引，MySQL就会先锁住该非主键索引，然后再锁住相关的主键索引。

1. 事务及ACID特性？

事务是指满足ACID特性的一些列操作，可以通过commit提交一个事务，也可以使用Rollback进行回滚。

* **原子性**（Atomicity）：事务被视为不可分割的最小单元，要么全部提交完成，要么全部回滚失败。
* **一致性**（Consistency）：数据库在事务执行前后保持一致状态。在一致性状态下，所有事务对一个数据的读取结果是相同的。
* **隔离性**（Isolation）：一个事务所做的修改在最终提交以前，对其他事务不可见。
* **持久性**（Durability）：一个事务一旦提交，则其所做的修改将永远保存到数据库中。

Mysql默认采用自动提交模式，也就是说如果不显示使用start transaction语句来开始一个事务，那么每一个查询都会被当做一个事务自动提交。

1. 数据库的三种不一致性与四种事务隔离级别？

并发事务带来的问题：

* 脏读（Dirty read）：当一个事务正在访问数据，并且对数据进行了修改，但是这个事务还没有提交，另一个事务也访问了这个数据，并使用了这个数据，即访问到了还没有提交的数据。
* 丢失修改（Lost to modify）：指一个事务读取一个数据时，另一个数据事务也读取了这个数据，那么第一个事务修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务修改的结果就被丢失了。如A=20，事务1读取A=20，事务2读取数据A=20，事务1修改数据A=A-1，事务2修改数据A=A-1，最终结果为19，事务的修改被丢失。
* 不可重复读（Unrepeatable read）：指在一个事务内多次读取同一个数据，在这个事务还没结束时，另一个事务也修改了这个数据。那么在第一个事务中的两次读数据之间，由于第二个事务修改导致了第一个事务两次读取不一致的情况。
* 幻读（Phantom read）：与不可重复读类似。在T1读取一些数据后，事务T2插入了一些数据，在随后的查询中事务T1发现一些原本不存在的记录。

数据库事务的隔离级别

* 读未提交（READ-UNCOMMITTED）：事务中的修改，即使没有提交，对其他事务也是可见的
* 读已提交（READ-COMMITTED）：一个事务只能读取已经提交的事务所作的修改，解决了脏读
* 可重复度（REPEATABLE-READ）：保证同一个事务中多次读取同一数据的结果是一样的，解决了不可重复读问题，MySql默认隔离级别
* 串行化（SERIALIZABLE）：强制事务串行执行，解决了幻读问题

1. Mysql调优技巧（腾讯一面）

* 选择合适的存储引擎：InnoDB
* 增大内存缓存
* 数据预热
* 定期优化重建数据库
* 降低磁盘写入操作
* 充分使用索引
* 分析查询日志和慢查询日志

1. OSI与TCP/IP各层的结构与功能，都有哪些协议？

* **应用层**：通过应用进程间的交互来完成特定的网络应用。常见的应用层协议有，DNS域名系统，HTTP协议（超文本传输协议），支持邮件SMTP协议，FTP文件传输协议
* 表示层：
* 会话层：SSL安全套接字层协议
* **传输层**：负责为两台主机进程之间的通信提供数据传输服务。多种应用可以使用同一个传输层服务。传输层具有复用和分用功能。传输层主要使用以下两种协议
  + 传输控制协议**TCP**：面向连接的，可靠的数据传输服务
  + 用户数据协议**UDP**：无连接的，不可靠的数据传输服务
* **网络层**：负责两个计算机直接的的通信。使用**IP协议**，分组叫做IP数据包。也叫IP层，网际层。网络层设备，路由器。最复杂的一层。ICMP协议
* **数据链路层**：通常称为链路层，数据链路层把IP数据包组装成帧，在相邻节点的链路上传送帧。每一帧包括必要的控制信息（如同步信息，地址信息，差错控制等）。**ARP协议**，数据链路层设备：网桥、交换机
* 物理层：实现相邻计算机节点之间比特流的透明传送，屏蔽具体传输介质和物理设备的差异。物理层设备，网卡，网线，集线器，中继器，调制解调器

交换机和路由器的区别

* 工作层次不同：路由器工作在网络层，交换机工作在数据链路层
* 数据转发依据的对象不同：交换机转发利用的是MAC地址，而路由器是利用IP地址
* 分割区域：传统交换机只能分割冲突域，不能分割广播域；而路由器可以分割广播域。VLAN可以分割广播域，但是各个子广播域之间是不能通信交流的，它们之间的交流仍然需要路由器。
* 路由器提供了防火墙功能，可以防止广播风暴。

1. TCP三次握手和四次挥手？

三次握手过程

* 客户端发送带有SYN标志的数据包——一次握手—服务端
* 服务端发送带有SYN/ACK标志的数据包——二次握手—客户端
* 客户端发送带有ACK标志的数据包——三次握手—服务端

为什么要三次握手?

三次握手的目的是为了建立可靠的通信信道，双方确认自己与对方的发送与接收时正常的，同时双方同步序列号。

有SYN，为什么还要ACK？服务端告知客户端，自己已经收到了客户端的数据。

四次挥手过程

* 客户端发送FIN，用来关闭客户端到服务器的数据传送
* 服务器收到FIN，它发回一个ACK，并关闭自己的数据接收
* 服务器发送FIN，关闭服务器发送数据的服务
* 客户端发回ACK，确认。

为什么要四次挥手？当一方在数据传送结束后发出释放的通知，等待对方确认后进入半关闭状态。这个过程需要两次握手。而当另一方也没有数据再发送时，则发出连接释放的通知，对方确认后就完全关闭TCP连接。这个过程也需要两次握手，因此需要四次挥手。

挥手比握手多一次的原因时FIN和ACK没有一起发送，因为此时服务器可能还有数据要发送给客户端。

1. TCP和UDP协议的区别？

* TCP：面向连接，传输可靠，传输形式是字节流。在性能上，TCP传输效率慢，所需要的资源多。首部字节较长，20-60个字节。在需要可靠数据传输的场景中应用，比如FTP、HTTP、SMTP等
* UDP：无连接，不可靠，传输形式是数据报文段。在性能上，UDP协议传输效率高，所需资源少。首部字节较短，8个字节。在需求通信速度的场景中应用，比如DNS、DHCP、ICMP、即时通信（如QQ语音、QQ视频、直播等）等场景。

1. TCP协议是如何保证可靠传输的？

* 应用数据被分割成TCP认为最适合发送的数据块。
* TCP给发送的每一个包进行编号，接收方对数据包进行排序，从而保证数据有序的传送给应用层
* **校验和**：TCP将保持首部和数据的校验和。这是端到端的校验和，保证数据在传输过程中没有任何变化。
* TCP的接受端回丢弃重复的报文
* **超时重传**：当TCP发出一个数据报后，它会启动一个定时器，等待接收端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。
* **流量控制**：TCP连接的每一方都有一个固定大小的缓冲区，TCP接收端，只允许发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据时，能提示发送方及时降低发送速率，防止丢包。TCP使用的流量控制协议是可变窗口的滑动窗口协议。（TCP利用滑动窗口实现流量控制）。
* **拥塞控制**：当网络拥塞时，减少数据的发送。
* **ARQ协议**：为了实现可靠传输，基本原理是，每发完一个分组就停止发送，等待对方确认。在收到确认后，再发送下一个分组。

1. ARQ协议

自动重传协议（Automatic Repeat-reQuest，ARQ）是OSI模型中数据链路层和传输层的错误纠正协议之一。它使用确认和超时这两个机制，在不可靠服务的基础上实现可靠的信息传输。如果发送方在一段时间后没有收到确认帧，它通常会重新发送。ARQ协议包括停止等待ARQ协议和连续ARQ协议。

* 停止等待ARQ协议
  + 停止等待协议的基本原理是每发完一个分组就停止发送，等待对方确认（回复ACK）。如果过了一段时间（超时后）还没收到ACK，就需要重新发送，直到收到确认后再发送下一个分组。
  + 停止协议中，若接收方收到重复分组，则丢弃该分组，但同时还要发送确认。
  + 优点：简单
  + 缺点：信道利用率低，等待时间长
* 连续ARQ协议（GBN协议，Go-back-N）
  + 连续ARQ协议的基本原理是，发送方维持一个发送窗口，凡是位于发送窗口内的分组可以连续的发送出去，不需要等待对方确认。接收方一般采用累计确认，对按序到达的最后一个分组发送确认，表明到这个分组位置的分组都已经正确收到了。
  + 优点：信道利用率高，容易实现，即使确认丢失也不必重传
  + 缺点：不能向发送方反映出已经正确收到的所有分组的信息。比如：发送方发送了5个分组，中间第3条丢失，接收方只能对前两个发送确认。发送方不知道后三个分组的达到情况，只好把最后三个全部重传一次。这叫做Go-Back-N，表示需要退回来重传已经发送的N个消息。

1. 滑动窗口和流量控制

TCP利用滑动窗口来实现流量控制。流量控制的目的是，控制发送方的发送速率，保证接收方来得及接收。接收方发送的确认报文中的窗口字段可以用来控制发送方发送窗口的大小，从而影响发送方的发送速率。

1. 拥塞控制

目的是防止过多的数据发送到网络中，从而避免网络中的链路或者路由器过载。拥塞控制是一个全局的过程，涉及到所有的主机、路由器以及与降低网络传输性能的有关的所有因素。

为了进行拥塞控制，TCP的发送方要维持一个**拥塞窗口**(cwnd)的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态变化。发送方的发送窗口取拥塞窗口和接收方的接收窗口中较小的一个。

TCP的拥塞控制采用了四种算法，即**慢开始**、**拥塞避免**、**快重传**和**快恢复**。在网络层也可以使路由器采取适当的分组丢弃策略，以减少网络拥塞的发生。

* **慢开始**：慢开始算法的思想是当主机开始发送数据的时候，如果立即把大量数据发送到网络，那么可能会引起网络阻塞，因为现在还不知道网络的拥塞情况。因此，先探测以下，即由小到大逐渐增大发送窗口，即由小到大主键增大拥塞窗口的数值。Cwnd初值设为1，每经过一个传播轮次，cwnd加倍。只有在网络开始时和超时重传时，才把cwnd值设为1.
* **拥塞避免**：拥塞避免的算法思路是，让拥塞窗口cwnd缓慢增大，即每经过一个往返时间RTT就把发送cwnd加1.
* **快重传与快恢复**：在TCP/IP中，快重传与恢复（fast retransmit and recovery，FRR）是一种拥塞控制算法，它能够快速恢复丢失的数据包。FRR要求接收方每收到一个不按顺序的数据报文段就立即发出重复确认。FFR规定接收方，如果连续收到三个重复的确认就立即重传对方尚未收到的报文段，而不必继续等待重传的计时器到期。FFR规定，当发送方连续收到三个重复确认后，就执行**乘法减小**，把cwnd减半。

1. 在浏览器中输入[www.qq.com](http://www.qq.com)地址到显示主页的过程

总体来说可以分成以下几个部分：

* DNS解析过程
* TCP建立连接（三次握手）过程
* 客户端发送HTTP请求
* 服务器处理请求并返回HTTP报文
* 浏览器解析数据并渲染
* 与目标主机断开连接（四次挥手）

**DNS解析**

DNS解析也就是域名解析，域名到IP地址的转换过程，DNS解析的工作由DNS服务器完成。

DNS解析的过程是按照浏览器缓存、系统缓存、路由器缓存、ISP（运营商）DNS缓存、根域名服务器、顶级域名服务器、根域名服务器的顺序，逐步读取缓存直到拿到IP地址。

1. Java中线程安全的collections

* Java中大多数Collections都不是线程安全的，比如：ArrayList, LinkedList, HashMap, HashSet, TreeMap, TreeSet等，Vector和HashTable、StringBuffer是线程安全的。Java中线程安全的集合主要在java.util.concurrent包中，其中包括ConcurrentHashMap、ConcurrentSkipListMap、ConcurrentSkipListSet、CopyOnWriteArrayList

1. Java四种引用类型

Java引用类型分为两大类：值类型、引用类型，值类型就是基本数据类型，引用类型就是除了基本数据类型之外的所有类型，基本的变量类型的储存空间被分配到栈中，引用类型有两块储存空间：一块在栈中，一块在堆中

* 1 简单类型（基本数据类型）是按值传递的
* 2 引用类型是引用传递，引用其实就像是一个对象的名字或者别名
* 引用是什么？

引用是一种数据类型，保存在stack中，保存了对象在内存（heap，堆空间）中的地址，这种类型即不是我们平时所说的简单数据类型也不是类实例（对象）。不同的引用可能指向同一个对象，一个对象可以有多个引用。

基本数据类型是按值传递，自定义数据类型是按引用传递

对引用判断的方式（判断一个对象是否要回收的方法）：引用计数法、可达性分析法

强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）4 种，这 4 种引用的强度依次减弱。

* **强引用**：默认声明就是强引用，强引用存在的对象永远不会被回收，哪怕内存不足，JVM也会直接抛出OutOfMemoryError。
* **软引用**：在内存足够的时候，软引用对象不会被回收，只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常。这种特性常常被用来实现缓存技术。
* **弱引用**：无论内存是否足够，只要 JVM 开始进行垃圾回收，那些被弱引用关联的对象都会被回收。
* **虚引用**：如果一个对象仅持有虚引用，那么它就和没有任何引用一样，它随时可能会被回收。虚引用必须要和 ReferenceQueue 引用队列一起使用

1. Java和C++的区别

* 语言层面
  + Java是纯面向对象的语言，所有变量和方法都在类中声明和定义。C++则保留了C语言中面向过程的特性。
  + C++是编译型类型强语言，由编译器直接编译成本地机器识别的机器语言。Java是半编译半解释型语言，由编译器得到class字节码文件，然后由虚拟机解释执行字节码，由JIT即时编译器进行热点代码探测，直接翻译成机器码。
  + Java没有指针，虽然虚拟机底层实现依赖指针，但不对外开放，更加安全。C++存在各种指针、常量指针、指针的指针、还有与指针对应的取址符号&，C++的指针使用不当肯能存在野指针。
* 类与继承
  + C++可以实现多继承抽象类，Java只能实现单继承，但可以进行接口实现和接口多继承接口
  + C++中的抽象类用virtual，Java用abstract，接口用interface
  + C++还有结构题、联合体，其成员变量默认访问权限为private，Java只有类型，类的成员变量默认访问权限为default，是一种包可见的访问权限。
* 操作层面
  + C++可以进行函数和运算符的重载，Java只允许函数重载
  + C++中有goto和const，Java中作为保留字。Java用final来定义常量，synchronized来用作同步
  + Java引入了真正的数组，将指针去掉，C++采用的是伪数组，需要关注数组越界问题
* 内存管理
  + Java有自动内存管理，而C++需要手动分配和释放内存
  + C++区分代码段、数据段、栈，其中代码段存储程序代码，数据段存储全局静态变量数据、堆用于动态内存分配、栈用来存储局部变量函数参数。Java内部分为程序计数器、堆、栈、方法区，栈分为本地栈和虚拟机栈，分别用来存储本地方法和Java方法的局部变量，程序计数器存储当前线程执行的字节码的行号，堆用来存储对象实例，方法区用来存储常量，静态变量，编译后的机器代码等信息。

1. Java垃圾回收宣判对象死亡的过程？

可达性分析算法中，未到达的对象并非是非死不可的，一个对象的死亡宣判需要经历两次标记阶段。

* 如果对象在进行可达性分析后，发现没有与GCRoots相连的引用链，则该对象被第一次标记并进行一次筛选，筛选条件为是否有必要执行该对象的finalize方法，若对象没有覆盖finalize方法或者该finalize方法已经被虚拟机执行过了，则均视作不必要执行该对象的finalize方法，即该对象将会被回收。反之，若对象覆盖了finalize方法并且该finalize方法并没有被执行过，那么，这个对象会被放置在一个叫F-Queue的队列中，之后会由虚拟机自动建立的、优先级低的Finalizer线程去执行，而虚拟机不必要等待该线程执行结束，即虚拟机只负责建立线程，其他的事情交给此线程去处理。
* 对F-Queue中对象进行第二次标记，如果对象在finalize方法中拯救了自己，即关联上了GCRoots引用链，如把this关键字赋值给其他变量，那么在第二次标记的时候该对象将从“即将回收”的集合中移除，如果对象还是没有拯救自己，那就会被回收。

示例代码：

public class FinalizeEscapeGC{

public String name;

public static FinalizeEscapeGC SAVE\_HOOK = null;

public FinalizeEscapeGC(String name){

this.name = name;

}

public void isAlive(){

System.out.println("yes, i am still alive :("+name+")");

}

@Override

protected void finalize() throws Throwable {

super.finalize();

System.out.println("finalize method excuted!");

System.out.println(this);

FinalizeEscapeGC.SAVE\_HOOK = this;

}

@Override

public String toString(){

return name;

}

public static void main(String[] args) throws InterruptedException {

SAVE\_HOOK = new FinalizeEscapeGC("leesf");

System.out.println(SAVE\_HOOK);

//对象第一次拯救自己

SAVE\_HOOK = null;

System.out.println(SAVE\_HOOK);

System.gc();

// 因为finalize方法优先级很低，所以暂停0.5秒以等待它

Thread.sleep(500);

if(SAVE\_HOOK != null){

SAVE\_HOOK.isAlive();

} else {

System.out.println("no, i am dead: (");

}

// 下面这段代码与上面的完全相同,但是这一次自救却失败了

// 一个对象的finalize方法只会被调用一次

SAVE\_HOOK = null;

System.gc();

// 因为finalize方法优先级很低，所以暂停0.5秒以等待它

Thread.sleep(500);

if (SAVE\_HOOK != null) {

SAVE\_HOOK.isAlive();

} else {

System.out.println("no, i am dead : (");

}

}

}

1. Java 的类加载过程？

Java类加载过程

* 加载：加载指的是将类的class文件读入到内存，并为之创建一个java.lang.Class对象，也就是说，当程序中使用任何类时，系统都会为之建立一个java.lang.Class对象。
  + 类的加载由类加载器完成，类加载器通常由JVM提供，JVM提供的类加载器通常被称为系统类加载器。除此外，可以通过继承ClassLoader基类来创建自己的类加载器。通过使用不同的类加载器，可以从不同来源加载类的二进制数据。有以下几种来源
    - 从本地文件系统加载class文件
    - 从jar包加载class文件，如JDBC编程时用到的数据库驱动类
    - 通过网络加载class文件
    - Java源文件动态编译，并执行加载
* 链接：当类被加载之后，系统为之生成一个对应的Class对象，接着将会进入连接阶段，连接阶段负责把类的二进制数据合并到JRE中。链接分为三个阶段
  + 验证：验证阶段用于检验被加载的类是否有正确的内部结构，并和其他类协调一致。验证的目的在于确保Class文件的字节流中包含信息符合当前虚拟机要求，不会危害虚拟机自身安全。其主要包括四种验证，文件格式验证，元数据验证，字节码验证，符号引用验证。
    - 文件格式验证：主要验证字节流是否符合Class文件格式规范，并且能被当前的虚拟机加载处理。例如：主，次版本号是否在当前虚拟机处理的范围之内。常量池中是否有不被支持的常量类型。指向常量的中的索引值是否存在不存在的常量或不符合类型的常量。
    - 元数据验证：对字节码描述的信息进行语义的分析，分析是否符合java的语言语法的规范。
    - 字节码验证：最重要的验证环节，分析数据流和控制，确定语义是合法的，符合逻辑的。主要的针对元数据验证后对方法体的验证。保证类方法在运行时不会有危害出现。
    - 符号引用验证：主要是针对符号引用转换为直接引用的时候，是会延伸到第三解析阶段，主要去确定访问类型等涉及到引用的情况，主要是要保证引用一定会被访问到，不会出现类等无法访问的问题。
  + 准备：类准备阶段负责为类的静态变量分配内存，并设置默认初始值。
  + 解析：将类的二进制数据中的符号引用替换成直接引用。
* 初始化： 初始化是为类的静态变量赋予正确的初始值，它的执行过程是这样的，首先字节码文件被加载到内存后，先进行链接的验证这一步骤，验证通过后准备阶段，给a分配内存，因为变量a是static的，所以此时a等于int类型的默认初始值0，即a=0,然后到解析（后面在说），到初始化这一步骤时，才把a的真正的值10赋给a,此时a=10。

1. Java 类加载时机？

类加载时机

* 创建类的实例时，也就是new一个对象
* 访问某个类或接口的静态变量，或者对该静态变量赋值
* 调用类的静态方法
* 反射（Class.forName()）
* 初始化一个类的子类（会首先初始化子类的父类）
* JVM启动时标明的启动类，即文件名和类名相同的那个类

对于final类型的静态变量，如果该变量的值在编译时就已经确定下来，那么这个变量相当于“宏变量”。Java编译器会在编译时直接把这个变量出现的地方替换成它的值，因此即使程序用该静态变量也不会导致该类初始化。反之，如果final类型的静态Field的值不能在编译时确定下来，则必须等到运行时才可以确定该变量的值，如果通过该类来访问它的静态变量，则会导致该类被初始化。

1. Java 类加载器有哪些？

类加载器负责加载所有的类，为所有被载入内存的类生成一个java.lang.Class实例对象。一旦一个类被加载到JVM中，同一个类就不会被再次载入了。一个载入JVM的类有唯一的标识。在Java中一个类用其全限定类名（包括包名和类名）作为标识；在JVM中，一个类用其全限定类名和其加载器作为唯一标识。JVM预定义三种类加载器，

**启动类加载器**（Bootstrap ClassLoader）:也称为根类加载器，负责将存放在<JAVA\_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如rt.jar,名字不符合的类库即使放在lib目录中也不会被加载）类库加载到虚拟机内存中。

**扩展类加载器**（Extension ClassLoader）：负责加载<JAVA\_HOME>\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

**应用程序类加载器**（Application ClassLoader）：也称为系统类加载器。负责加载用户类路径（classpath）上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

用户自定义加载器

1. Java 类加载机制？

* **全盘负责**：当一个类加载器负责加载某个class时，该class所依赖和引用的其它class也将由该类加载器负责载入
* **双亲委派模型**工作过程是：如果一个类加载器收到类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器完成。每个类加载器都是如此，只有当父加载器在自己的搜索范围内找不到指定的类时（即ClassNotFoundException），子加载器才会尝试自己去加载。保证类的唯一性，避免重复加载
* **缓存机制**：缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区中搜寻该Class，只有当缓存区中不存在该Class对象时，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓冲区中。这就是为很么修改了Class后，必须重新启动JVM，程序所做的修改才会生效的原因。

1. Java为什么要用锁？

在多线程环境中保证线程安全

1. 什么是线程安全？如何实现保证线程安全？

线程安全是指多个线程对共享变量方位时的结果和单线程下顺序对共享变量访问的结果是一致的，是符合预期的；而线程不安全是指多个线程对共享变量访问的结果每次都是不确定的。为了保证线程安全，第一个不用共享变量，第二个采用加锁机制，第三个使用并发容器（本质还是锁实现的）。

1. Volatile关键字的原理及作用？

这个关键字的作用是保证多线程中变量的可见性和禁止指令重排序。在多线程的环境中，每个线程都有自己的私有缓存区域，访问数据时，会先将数据存储到缓存中如果下次访问时，这个数据在缓存中已经存在，则直接从缓存中读取，写入也是一样，但是不确定什么时候将缓存中的数据更新到内存中。而被volatile修饰的变量，线程在读取这个变量时，会直接从内存中读取数据，同时在写数据时也直接写入到内存中，而不经过缓存。这个关键字保证了可见性，不是保证原子性，所以并不能保证线程安全。要保证线程安全还是要加锁。Volatile的还有一个作用是禁止指令重排序。

1. synchronized关键字的原理及作用？

* 对于普通方法，锁的是当前实例对象
* 对于静态方法，锁的是当前类
* 对于同步方法块，锁的是括号里配置的对象。

当一个线程试图访问同步代码块时，它首先必须得到锁，退出或者抛出异常时必须释放锁。

Synchronized用的锁保存在Java的对象头里，Java对象头里的Mark Word默认存储对象的HashCode、分代年龄标记和锁标记位。在运行期间，Mark Word里存储的数据会随着锁标志位的变化而变化。



锁一共有四种状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态，几个锁状态随着竞争情况逐渐升级。为了提高获得锁和释放锁的效率，锁可以升级但不能降级。

* 1偏向锁

当一个线程访问同步块并获取锁时，会在对象头和栈帧的锁记录里存储偏向的线程ID，以后该线程在进入和退出同步块时不需要进行CAS操作来加锁和解锁，只需测试Mark Word里线程ID是否为当前线程。如果测试成功，表示线程已经获得了锁。如果测试失败，则需要判断偏向锁的标识。如果标识被设置为0（表示当前是无锁状态），则使用CAS竞争锁；如果标识设置成1（表示当前是偏向锁状态），则尝试使用CAS将对象头的偏向锁指向当前线程，触发偏向锁的撤销。偏向锁只有在竞争出现才会释放锁。当其他线程尝试竞争偏向锁时，程序到达全局安全点后（没有正在执行的代码），它会查看Java对象头中记录的线程是否存活，如果没有存活，那么锁对象被重置为无锁状态，其它线程可以竞争将其设置为偏向锁；如果存活，那么立刻查找该线程的栈帧信息，如果还是需要继续持有这个锁对象，那么暂停当前线程，撤销偏向锁，升级为轻量级锁，如果线程1不再使用该锁对象，那么将锁对象状态设为无锁状态，重新偏向新的线程。

* 2轻量级锁

线程在执行同步块之前，JVM会先在当前线程的栈栈帧中创建用于存储锁记录的空间，并将对象头的MarkWord复制到锁记录中，即Displaced Mark Word。然后线程会尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁。如果失败，表示其他线程在竞争锁，当前线程使用自旋来获取锁。当自旋次数达到一定次数时，锁就会升级为重量级锁。

轻量级锁解锁时，会使用CAS操作将Displaced Mark Word替换回到对象头，如果成功，表示没有竞争发生。如果失败，表示当前锁存在竞争，锁已经被升级为重量级锁，则会释放锁并唤醒等待的线程。

* 3重量级锁

重量级锁依赖于操作系统互斥量实现，重量级锁会导致进程从用户态于内核态之间的切换。

原理：

Java对象头及Monitor

在JVM中，对象在内存中的布局分为三块区域：对象头、实例数据和对齐填充。

* 实例变量：存放类的属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组的长度，这部分内存按4字节对齐。
* 填充数据：由于虚拟机要求对象起始地址必须是8字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐

Java对象头是实现synchronized的锁对象的基础，synchronized使用的锁对象是存储在Java对象头里的，jvm采用2个字来存储对象头（对象是数组则会分配3个字，多出来的一个字记录数组长度）。其主要结构是Mark Word和Class Metadata Address组成。

* Mark Word：存储对象的hashCode、锁信息或分代年龄或GC标志信息
* Class Metadata Address：类型指针指向对象的类元数据，JVM通过这个指针确定该对象是哪个类的实例。

锁升级过程：

1. equals与hashcode？

Equals()和hashcode()的作用其实是一样的，在Java中都是用来比较两个对象是否相等一致的。因为在实际应用中，equal()函数比较全面和复杂，效率较低。而利用hashcode()进行对比，则只要生成一个hash值进行比较就可以了，效率很高。但是hashcode并不是完全可靠的，有时不同的对象，它们生成的hashcode也会一样，hashcode在大部分时候可靠，并不是绝对可靠。

两个equal()的对象，他们的hashcode一定相等。但两个hashcode相等的对象，它们不一定equal()。两个hashcode不同的对象，它们一定不equal()

对于需要大量且快速的对比的对象，用equal的效率太低，所以解决方式是，每当需要对比的时候先用hashcode去对比，如果hashcode不一样，则表示这两个对象肯定不一样，不必再用equal了。如果hashcode相同，此时再对比它们的equal。这样既能大大提高效率，也保证了对比的绝对可靠性。

hashCode的作用是更好的支持基于哈希机制的Java集合类

1. synchronized的缺陷

synchronized释放锁只有两种情况：

* 获取锁的线程执行完了代码块，线程释放对锁的占有
* 线程执行发生异常，此时JVM会让线程自动释放锁

但是，当获取锁的线程由于需要等待IO或者其它原因被阻塞了，又没有释放锁，其它线程无法获取锁，这严重影响程序执行效率。因此需要一种机制不让等待的线程无限期的等待下去。

除此外，在多个线程读写文件的时候，读操作和写操作会发生冲突，写操作和写操作也会发生冲突，但是读操作和读操作不会发生冲突，使用synchronized不能使多个线程并发的读，导致效率很低。

因此需要一种机制来使多个线程都只进行读操作时，线程不会发生冲突，通过Lock可以办到。

1. Java并发之Lock

Lock不是Java内置的，synchronized是Java语言的关键字，因此是内置特性。Lock是一个接口，通过这个类可以实现同步访问。

Lock的缺点在于需要用户手动释放锁，如果没有主动释放可能导致死锁。而synchronized则会自动释放锁。

* Lock，是一个接口
  + lock()、tryLock()、tryLock(long time, TimeUnit unit)和lockInterruptibly()是用来获取锁的
  + lock获取锁，如果锁已经被其它线程获取，则进行等待。且lock的代码块必须在try{}catch{}中进行，并且将释放锁的操作放在finally中，以保证锁一定被释放，防止死锁的发生。
  + tryLock()方法有返回值，表示尝试获取锁，获取成功则返回true，获取失败则放回false，方法会立即返回，不会一直等待。
  + tryLock(long time, TimeUnit unit)和tryLock()类似，不过在这个方法在拿不到锁时会等待一定时间，在时间期限内拿不到锁就返回false，否则返回true。
  + lockInterruptibly()，线程正在获取锁，如果线程在等待锁，则这个线程能够响应中断，即中断线程的等待状态。即，当两个线程同时通过lockInterruptibly()想获取某个锁时，此时线程A获取到了锁，线程B只有在等待状态，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。
  + lock.lockInterruptibly()必须放在try块中，或者在调用lockInterruptibly()的方法外声明抛出InterruptedException。
* ReentrantLock，意思是可重入锁，ReentrantLock是唯一实现了Lock接口的类。
* ReadWriteLock，一个接口，读写锁
* ReentrantReadWriteLock，实现了ReadWriteLock接口，主要有两个方法：readLock()和writeLock()用来获取读锁和写锁。

Lock和synchronized的选择

* Lock是一个接口；synchronized是关键字，内置的语言实现
* Synchronized在发生异常时会自动释放锁，不会导致死锁发生；Lock发生异常时，如果没有主动unLock释放锁，则很可能造成死锁现象。
* Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；
* 通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。
* Lock可以提高多个线程进行读操作的效率。
* 竞争不激烈两者性能差不多，竞争激烈则Lock效率远优于synchronized。在JDK1.6后，优化了synchronized关键字的实现，效率大大提高。

1. Java常用的锁及其分类

* **可重入锁**

如果锁具备可重入性，则称为可重入锁。其中synchronized和ReentrantLock都是可重入锁，可重入锁表明了锁的分配机制：基于线程的分配，而不是基于方法调用的分配。

* **可中断锁**

可以响应中断的锁，在Java中，synchronized不是可中断锁，而Lock是可中断锁。

* **公平锁**

公平锁即尽量以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该所，这种就是公平锁。

Java中，synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。

而对于ReentrantLock和ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。

* **读写锁**

读写锁将对一个资源（比如文件）的访问分成了2个锁，一个读锁和一个写锁。正因为有了读写锁，才使得多个线程之间的读操作不会发生冲突。ReadWriteLock就是读写锁，它是一个接口，ReentrantReadWriteLock实现了这个接口。

1. Java主流的锁

* 线程要不要锁住同步资源
  + 锁住：悲观锁
  + 不锁住：乐观锁
* 锁住同步资源失效，线程要不要阻塞
  + 阻塞：
  + 不阻塞：自旋锁、适应性自旋锁
* 多个线程竞争同步资源的细节有没有区别
  + 不锁住资源，多个线程中只有一个能修改资源成功，其它线程会重试——无锁
  + 同一个线程执行同步资源时自动获取锁——偏向锁
  + 多个线程竞争同步资源时，没有获取资源的线程自旋等待释放——轻量级
  + 多个线程竞争同步资源时，没有获取资源的线程阻塞等待唤醒——重量级
* 多个线程竞争时要不要排队？
  + 排队——公平锁
  + 先尝试插队，插队失败再排队——非公平锁
* 一个线程中的多个流程能不能获取同一把锁
  + 能——可重入锁
  + 不能——不可重入
* 多个线程能不能共享一把锁
  + 能——共享锁
  + 不能——排他锁
* Condition接口

JDK1.5之后才有的，等待通知机制

* CAS：compare and swap比较和交换，是一种又名的无锁算法。无锁编程，即不用锁的情况下实现多线程间的变量同步，也叫做非阻塞同步。三个操作数，需要读取的内存值V，进行比较的值A，拟写的值B。当且仅当V的值等于A时，CAS操作通过原子方式用新值B来更新V的值，否则不会执行任何操作，一般是一个自旋操作，即不断的重试。
  + CAS锁的问题：ABA问题，循环时间开销太大，只能保证一个共享变量的原子操作。适用于竞争少的情况
* 自旋锁：轻量级的锁，通过空操作来使当前线程等待一段时间，从而避免切换到CPU内核态的开销。自旋锁的实现原理同样是CAS。
  + 自适应自旋锁：自旋的时间不再固定，而实由上一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也是很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。如果对于某个锁，自旋很少成功获得过，那在以后尝试获取这个锁时将可能省略掉自旋过程，直接阻塞线程，避免浪费处理器资源。
  + 自旋锁中：常见的三种形式的锁使TicketLock、CLHlock和MCSlock
  + 自旋锁在单CPU环境下是无效的，
    - 在不支持抢占的系统中，只有一个线程，自旋锁什么也不用做
    - 在支持抢占的系统中，自旋锁加锁的作用是禁止抢占，解锁的作用是启用抢占。

1. Java原子类

Java提供的原子类是基于CAS实现的，CAS是一种乐观锁。

* AtomicInteger

AtomicInteger是一个支持原子操作的Integer类，保证对AtomicInteger类型变量的增加和减少是原子性的，不会出现多个线程下的数据不一致问题。

声明：private volatile int value;

Volatile修饰value变量，保证了变量的可见性

incrementAndGet() 方法，下面是具体的代码

public final int incrementAndGet(){

for(;;){

int current = get();

int next = current+1;

if(compareAndSet(current,next))

return next;

}

}

compareAndSet()方法调用compareAndSwapInt()方法的声明如下，是一个native方法。compareAndSwapInt 基于的是 CPU 的 CAS 指令来实现的。所以基于 CAS 的操作可认为是无阻塞的，一个线程的失败或挂起不会引起其它线程也失败或挂起。并且由于 CAS 操作是 CPU 原语，所以性能比较好。

1. 什么是线程安全？

线程安全是指多个线程对共享变量访问时的结果和单线程下顺序对共享变量访问的结果是一致的，是符合预期的；而线程不安全是指多个线程对共享变量访问的结果每次都是不确定的。为了保证线程安全，第一个不用共享变量，第二个采用加锁机制，第三个使用并发容器（本质还是锁实现的）。

1. Java Object九大方法？

* protected Object clone() 创建并返回此对象的一个副本。
* boolean equals(Object obj) 指示某个其他对象是否与此对象“相等”。
* protected void finalize() 当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。
* Class<? extendsObject> getClass() 返回一个对象的运行时类。
* int hashCode() 返回该对象的哈希码值。
* void notify() 唤醒在此对象监视器上等待的单个线程。
* void notifyAll() 唤醒在此对象监视器上等待的所有线程。
* String toString() 返回该对象的字符串表示。
* void wait() 导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。
* void wait(long timeout) 导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量。
* void wait(long timeout, int nanos) 导致当前的线程等待，直到其他线程调用此对象的 notify()

1. Java单例模式

单例模式的实现方法很多，下面依次列出各种方法，并分析优缺点

* 懒汉模式（线程不安全）

public class Singleton{

private static Singleton instance = null;

private Singleton(){

}

public static Singleton getInstance(){

if(instance==null){

instance = new Singleton();

}

return instance;

}

}

* 线程安全的懒汉模式，缺点：效率低

public class Singleton{

private static Singleton instance = null;

private Singleton(){

}

public static synchronized Singleton getInstance(){

if(instance==null)

instance = new Singleton();

return instance;

}

}

* 饿汉模式（线程安全），缺点：没有起到lazy loading的作用

public class Singleton{

private static Singleton instance = new Singleton();

private Singleton(){}

public static Singleton getInstance(){

return instance;

}

}

* 静态类内部加载（线程安全），推荐

静态内部类不会在单例加载时就加载，而是在调用getInstance()方法时才进行加载，达到了类似懒汉模式的效果，而这种方法又是线程安全的，由JVM保证线程安全性。

public class Singleton{

private static class SingletonHolder{

private static Singleton instance = new Singleton();

}

private Singleton{

}

public static Singleton getInstance(){

return SingletonHolder.instance;

}

}

* 枚举方法（线程安全），Josh Bloch提倡的方式

解决了三个问题：自由串行化，保证只有一个实例，线程安全，但是在需要继承的情况下不适用

enum Singleton{

INSTANCE;

public void otherMethods(){

System.out.println("Something");

}

}

* 双重校验锁(volatile关键字)

public class Singleton {

private volatile static Singleton instance = null;

private Singleton(){

}

public static Singleton getInstance(){

if(instance==null){

synchronized(Singleton.class){

if(instance==null){

instance = new Singleton();

}

}

}

return instance;

}

}

第一次判断instance是否为null

第一次判断是在同步块之外判断，由于单例模式只会创建一个实例，并通过getInstance方法返回Singleton对象，所以第一次判断是为了在Singleton对象已经创建的情况下，避免进入同步代码块，提升效率。

第二次判断instance是否为null

第二次判断是为了避免以下情况发生。

STEP1. 线程A已经经过了第一次判断，判断instance为null，准备进入代码块

STEP2. 此时线程B在同步代码块中，创建了一个实例。并离开同步块

STEP3. 线程A此时获取时间片，再进入同步代码块，又创建一个实例，显然不满足要求。

为什么要加volatile关键字？

volatile关键字的作用是保证可见性，同时禁止指令重排序。

在instance=new Singleton();这行代码中，需要三个指令

- 指令1：获取instance的地址，分配内存空间

- 指令2：初始化Singleton对象

- 指令3：将这块内存空间的地址指向引用变量Singleton

由于指令重排序，上面的过程可能会变成1-3-2的过程，如果是1-3-2的过程，在多线程下，线程A先完成了1-3，但是2未完成，此时线程B判断instance不为空，但实际对象仍为空，造成了错误。

同时，synchronized锁的是Singleton.class对象，只能保证Singleton.class对象的可见性，但是不能保证Singleton对象的可见性。避免线程A创建实例还只在自己的工作内存，未更新到主内存。线程2判断对象为空，创建实例。

不用volatile的缺陷在于：

STEP1. 线程A访问getInstance()方法，因为单例没有实例化，所以进入了锁定状态

STEP2. 线程B访问getInstance()方法，因为单例还没有实例化，得以访问接下来的代码块，而接下来的代码块已经被线程A锁定

STEP3. 线程A进入下一步判断，因为单例还没有实例化，所以进行实例化，成功实例化后退出代码，解除锁定。

STEP4. 线程B进入代码块，锁定线程，进入下一步判断，因为已经实例化，退出代码块，解除锁定。

STEP5. 线程A获取到了单例实例并返回，而线程B没有获取到单例实例并返回NULL。

1. Java内存模型(JMM)

Java内存模型特别容易和JVM虚拟机运行时数据区域混淆。这里详细讲述什么是Java内存模型。

Java内存模型是Java虚拟机规范定义的，用来屏蔽掉java程序在各种不同的硬件和操作系统对内存访问的差异，从而实现Java程序在各种不同平台上内存访问的一致性。主要目标是定义程序中变量的访问规则。即在虚拟机中将变量存储到贮存或者从变量从主存取出的底层细节。

主内存：所有变量都必须在主内存中产生，所有线程共享，

工作内存：每个线程都有自己的工作内存，工作内存中保存了线程需要的变量在主内存中的副本，线程对主内存的变量修改是在工作内存中的进行的，不能直接读写主内存的变量。不同线程之间不能互相访问对方的工作内存。

Java内存模型是指：运行在Java虚拟机里的每个线程都拥有自己的线程栈，线程栈是私有的。所有线程共享Java堆。这个线程栈里包含了这个线程调用的方法当前执行点相关的信息。线程栈是线程私有的，线程栈里的变量对其它线程不可见。

线程栈中包含：原始类型、指向对象的引用（对象本身存放在堆上），对象方法里的本地变量存放在线程栈上。

对象的成员变量可能存在堆上，静态变量存放在堆上。

存放在堆上的对象可以被所有持有对这个对象引用的线程访问。当一个线程可以访问一个对象时，它也可以访问这个对象的成员变量。如果两个线程同时调用一个对象上的方法，它们都会访问这个对象的成员变量，但是每个线程拥有这个本地变量的私有拷贝。

首先是并发编程需要处理的两个关键问题是：线程间的通信和线程间的同步。

* 通信：是指线程间交换信息的机制。在命令式编程中，线程的通信机制有两种：共享内存和消息传递。
* 在共享内存的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信。
* 在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信。
* 同步：指程序用于控制不同线程之间操作发生的相对顺序的机制
  + 在共享内存模型里，同步是显示进行的。程序员必须显示的指定某个方法或者代码段需要在线程间互斥执行。
  + 在消息传递的并发模型里，由于消息的发送必须在消息接收之前，因此同步是隐式进行的。

Java的并发采用的是**共享内存模型**，Java线程间的通信总是隐式进行，整个通信过程对程序员完全透明。

* Java内存模型的抽象

Java中左右的实例域、静态域和数组元素存储在堆内存中，堆内存在线程间共享。局部变量、方法定义参数和异常处理参数不会在线程间共享。

Java间的通信由Java内存模型（JMM）控制。JMM决定了对共享变量的写入何时对另一个线程可见。JMM的线程与主内存的抽象关系：线程之间的共享变量存储在主内存中，每个线程都有一个私有的本地内存，本地内存中存储了该变量以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。JMM通过控制主内存与每个线程的本地内存的交换，来为Java程序员提供内存可见性保证。

* 指令重排序：在程序执行时，为了提高性能，编译器和处理器会对执行做重排序，分为三类
  + 编译器优化的重排序，不改变单线程语义前提下，重新安排语句执行顺序
  + 指令集并行的排序，如果不存在数据依赖性，处理器可以改变语句对应机器执行的执行顺序
  + 内存系统的重排序，由于处理器使用读/写缓冲区
* 顺序一致性内存模型
* Volatile
  + 可见性：对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入
  + 原子性：对任意单个volatile变量的读/写具有原子性
* Volatile写-读内存的定义
  + 当写一个volatile变量时，JMM会把该线程对本地内存种共享变量值刷新到主内存
  + 当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效，线程接下来将从主内存读取共享变量。

1. Java中异常的分类

Java中异常类分为两大类型：Error类代表编译和系统错误，不允许捕获；Exception代表标准Java库方法所激发的异常。Exception还包含运行时异常RuntimeException和非运行时异常类，Non\_RuntimeException这两个子类

运行异常类对应于编译错误，由Java程序在运行时产生的由解释器引发的各种异常。出现运行错误往往表示代码有错误，如：算数异常（被0除）、下标异常

非运行时异常：可检测异常。在Java标准包java.lang和java.util和java.net中定义的异常都是非运行时异常。

* 算术异常类：ArithmeticExecption
* 空指针异常类：NullPointerException
* 类型强制转换异常：ClassCastException
* 数组负下标异常：NegativeArrayException
* 数组下标越界异常：ArrayIndexOutOfBoundsException
* 违背安全原则异常：SecturityException
* 文件已结束异常：EOFException
* 文件未找到异常：FileNotFoundException
* 字符串转换为数字异常：NumberFormatException
* 操作数据库异常：SQLException
* 输入输出异常：IOException

1. Integer.valueOf(100) == new Integer(100)？ true or false

False，

1. Java线程池

* 线程池的作用：限制系统中线程的数量
* 为什么使用线程池：减少创建和销毁线程次数，减少资源消耗，提高响应速度，提高线程可管理性（如定时执行、并发数控制）
* 五种线程池
  + newCachedThreadPool 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程
  + newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
  + newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
  + newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。
  + newWorkStealingPool：一个拥有多个任务队列的线程池，可以减少连接数，创建当前可用cpu数量的线程来并行执行。
* 线程池的参数
  + corePoolSize就是线程池中的核心线程数量，这几个核心线程，只是在没有用的时候，也不会被回收
  + maximumPoolSize就是线程池中可以容纳的最大线程的数量
  + keepAliveTime， 就是线程池中除了核心线程之外的其他的最长可以保留的时间，因为在线程池中，除了核心线程即使在无任务的情况下也不能被清 除，其余的都是有存活时间的，意思就是非核心线程可以保留的最长的空闲时间，
  + util，就是计算这个时间的一个单位。
  + workQueue，就是等待队列，任务可以储存在任务队列中等待被执行，执行的是FIFIO原则（先进先出）。
  + threadFactory，就是创建线程的线程工厂。
  + handler,是一种拒绝策略，我们可以在任务满了之后，拒绝执行某些任务
* 拒绝策略
  + AbortPolicy策略：该策略会直接抛出异常，阻止系统正常工作。
  + CallerRunsPolicy 策略：只要线程池未关闭，该策略直接在调用者线程中，运行当前的被丢弃的任务。
  + DiscardOleddestPolicy策略： 该策略将丢弃最老的一个请求，也就是即将被执行的任务，并尝试再次提交当前任务。
  + DiscardPolicy策略：该策略默默的丢弃无法处理的任务，不予任何处理。
* 执行执行execute()方法和submit()方法的区别是什么？
  + execute()方法适用于提交不需要返回值的任务，无法判断任务是否被线程池执行成功与否；
  + submit()方法用于提交需要返回值的任务。线程池会返回一个future类型的对象，通过这个future对象可以判断任务是否执行成功，并且可以通过future的get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，而使用get(long timeout,TimeUnit unit)方法则会阻塞当前线程一段时间后立即返回，这时有可能任务没有执行完。、
* 实现Runnable接口和Callable接口的区别
  + 如果想让线程池执行任务需要实现Runnable接口或Callable接口。区别在于Runnable接口不会返回结果，但是Callable接口可以返回结果

1. ArrayList和LinkedList的区别？插入、删除、访问的时间复杂度？
2. ArrayList是怎么扩容的？初始化时给定ArrayList的size，数组大小一定就是给定值么？
3. HashMap初始大小，为什么要这么设计？
4. JVM虚拟机运行时数据区域

JVM运行时数据区域大致可以分为5个区域，分别是：方法区、堆、虚拟机栈、本地方法栈、程序计数器。其中方法区和堆随着虚拟机启动而存在，线程间共享；而其它三个则是线程私有的。

* 程序计数器：占用较小的内存空间，是当前线程正在执行的字节码的行号指示器，JVM中通过改变这个值来选择下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复都需要依赖这个计数器。唯一在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。
* Java虚拟机栈：线程私有，用于存储局部变量、操作数、动态链接、方法出口信息等。每个方法的调用对应这个栈帧在虚拟机栈中入栈到出栈的过程。两种错误：访问的栈深度大于允许的深度，将会抛出StackOverFlowError异常；如果虚拟机动态扩展栈时，无法申请到足够内存，则会抛出OutOfMemoryError异常。
* 本地方法栈：为虚拟机中使用到的native方法（Java调用非Java代码的接口）服务。
* Java堆：是JVM中内存最大的一块。Java堆被所有线程共享。唯一作用就是存放对象实例，几乎所有的对象实例都在这里分配内存。也是垃圾收集管理的主要区域，因此很多时候也被称为GC堆。
* 方法区（也称为永久代）：主要用于存储已被JVM加载的类信息、常量、静态变量、即时编译后的代码等数据。包含常量池。

1. Java 中String存放的位置

* 每次new String对象存放在堆中，其中String s1 = new String(“helloworld”);会生成两个对象，一个是在常量池中的”helloworld”，另一个是堆中的String
* 而String s=”helloworld”，存放在常量池中

1. String、StringBuffer、StringBuilder的区别

* String
  + 字符串常量，被创建就不能被修改，而对已存在的String对象的修改都是重新创建新对象，把新值保存进去
  + String是final类，不能被继承
  + String对象不是基本类型
  + String覆盖了equals方法和hashCode方法
* StringBuffer
  + 字符串可变对象，可以对字符串进行操作，修改字符串不会创建新对象
  + 执行效率较慢，但是线程安全
  + 没有覆盖equals和hashCode方法
  + 可以动态构造字符串，append方法
* StringBuilder
  + 字符串可变对象，可以对字符串进行操作，也不会创建新对象
  + 执行效率高，但线程不安全

1. GC算法，垃圾回收器

常用GC算法有4种，分别是**标记清除算法、复制算法、标记整理算法、分代收集算法**（JVM主流算法）。常用的垃圾回收器有7种，分别是**Serial串行收集器、ParNew多线程收集器、Parallel Scavenge收集器、Serial Old收集器、Parallel Old收集器、CMS收集器、G1收集器**。

* 标记清除算法：从根集合（GC Roots）进行扫描，对存活的对象进行标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收，此算法没有虚拟机采用。其优点是不需要移动对象，效率较高，不需要额外的空间。缺点：扫描了两次，效率较低，收集暂停时间长，会产生不连续的内存碎片。
* 复制算法：将内存分成两块大小相等的区域，每次只使用其中一块，当这块内存用完了，就会将所有存活对象复制到另一块内存，然后清除前一块内存。优点：不易产生内存碎片问题。缺点：复制代价很高，适合新生代（因为新生代中对象存活率较低），而且总是有一块内存空闲。
* 标记整理算法：在标记后，将存活对象往一端移动，然后清理掉边界以外的内存。不会产生碎片，但是移动对象的成本大，常用于老年代。
* 分代收集算法：JVM的主流算法。核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），在堆区之外还有一个代就是永久代（Parmanet Generation）。老年代的特点是每次垃圾收集时只有少量的对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点采取最适合的收集算法。
* 内存被分成三个区域
  + 新生代：分成三部分，Eden，S1，S2，8：1：1
  + 老年代：Tenured
  + 永久代：方法区
* 大部分对象都是在新生代中的Eden区域生成的，回收时，先将Eden区域的存活对象复制到S0区，然后清空Eden区。
* Serial(串行)收集器：针对新生代，采用复制算法，单线程收集，进行收集时，必须暂停所有工作线程，直到完成。
* ParNew收集器：多线程，特点和Serial收集器一样。除了Serial收集器外，只有它能与CMS收集器配合工作。
* Parallel Scavenge收集器：新生代收集器、复制算法、多线程收集，目标提高吞吐量
* Serial Old收集器：Serial收集器的老年代版本，针对老年代，标记-整理算法，单线程收集，与Parallel Scavenge收集器搭配使用，作为CMS的后备预案
* Parallel Old收集器：Parallel Scavenge收集器的老年代版本，针对老年代，采用标记-整理算法，多线程收集
* CMS收集器：针对老年代，基于标记-清除算法，以获取最短回收停顿时间为目标，并发收集、低停顿，需要更多内存
* G1收集器：JDK1.7后的商用收集器，充分利用多CPU、多核环境下的硬件优势，可以缩短Stop the world的时间，也可以让并发让垃圾收集和用户程序同时执行。分代收集，收集范围包括新生代和老年代，能独立管理整个GC堆，不需要其它收集器搭配，可预测的停顿：低停顿的同时实现高吞吐量

1. 介绍一下集合里的hashmap？1.7和1.8的区别？hashmap的扩容机制？如何处理哈希冲突（碰撞）？

在JDK1.7中，hashmap的本质是**数组+链表**，使用一个Entry数组来存储数据，用key的hashcode取模来决定key放在数组中的未知，如果hashcode相同，或者hashcode取模后结果相同，那么这些key会被定位到Entry数组的同一个未知，这些key会形成一个链表。当key足够多，或者hash函数较差时，这个链表可能很长，put/get需要遍历整个链表，最差情况下复杂度为O(n)。

在JDK1.8中，使用Node**数组**来存储数据，但这个Node可能是**链表结构**，也可能是**红黑树**结构；如果元素与key的hashcode相同，那么这些key也会被定位到Node数组的同一个未知中，在不超过8个的时，使用链表存储，超过8个，会调用treeifyBin函数，将链表转换成红黑树，用于红黑树的特点，查找某个特定元素只需要O(logn)的开销。

HashCode扰动处理：JDK1.8中，1次位运算+1次异或运算；JDK1.7 4次位运算+5次异或运算

插入数据方式：JDK1.8 直接插入到数据尾部，尾插法，不会出现逆序环形链表死循环问题；JDK1.7 头插法，会出现逆序，环形链表死循环问题。

扩容后存储位置计算方式：JDK1.8，扩容后位置=原位置+旧容量；JDK1.7 全部按原来的方法重新计算，hashcode & length-1。

扩容：JDK1.8 扩容前插入，转移数据时统一计算；JDK1.7 扩容后插入，单独计算，转移数据时无法统一计算。

* 为什么直接用hashCode()作为下标位置？容易出现哈希码与数组大小范围不匹配的情况，即哈希码超出数组大小范围。为了解决这个问题，采用hashcode & (length-1)。
* 为什么采用hashcod & (length-1)计算数组下标？根据HashMap的数组长度，按需取哈希码一定数量的低位作为存储数组的下标位置，解决哈希码与数组大小范围不匹配的问题。
* 再计算下标前，为什么需要对哈希码进行二次处理：扰动处理？加大哈希码的随机性，使得分布更均匀，从而提高对应数组存储下标位置的随机性 & 均匀性，最终减少Hash冲突。

1. HashMap与concurrenthashmap的区别？concurrenthashmap的线程安全是怎么实现的，1.7和1.8有什么区别？为什么要这样升级锁机制？

HashMap是线程不安全的，而concurrenthashmap是线程安全的。Concurrenthashmap的线程安全是通过加锁实现的。

* 在JDK1.7中，concurrenthashmap是通过**数组+segment+分段锁**实现的。ConcurrentHashMap中的分段锁称为Segment，类似于HashMap的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表,同时又是一个ReentrantLock。定位一个元素的过程需要进行两次Hash操作。
* 在JDK1.8中，**数组+链表+红黑树**的实现方式来设计，内部采用大量**CAS**操作。CAS是一种基于锁的操作，而且是乐观锁。V，A和新值B。JDK8中彻底放弃了Segment转而采用Node，其设计思想也不再是分段锁思想。

1. ConcurrentHashMap如何实现size的

JDK1.7 之前的方法是，进行一次遍历，segments数组，将每个segment的count加起来作为总数，期间把modCount加起来

连续左两遍循环统计segment得到count，比较两次的modCount，如果相同，意味着segments没有改动，是正确的

否则，对所有segment加锁，重新统计

在concurrentHashMap中

1. Java NIO
2. 操作系统内存管理——段页式管理

管理方法：虚地址、地址变换、内存分配和回收、内存扩充、内存共享和保护

连续内存分配管理方式：单一连续存储管理和分区式储管理两种方式

* 单一连续存储管理

内存被分为两个区域：系统区和用户区。应用程序装入到用户区，可使用用户区全部空间。其特点是简单。

* 分区存储管理

把内存分成不同的区，按区进行分配。分区管理引入了两个新问题：内碎片和外碎片

固定分区：把内存分成若干固定大小的连续分区，分区大小可以相等。优点：易于实现、开销小，缺点：内碎片浪费，分区数固定，限制并发执行程序数

动态分区：在装入时按照初始要求分配，优点：没有内碎片，但是有外碎片。

内存紧缩：将各个占用分区的内存向一端移动，然后各个空闲分区合并成为一个空闲分区

交换技术：暂停内存中执行的的进程，将进程地址空间保存到外存的交换区，而将外存中的由阻塞态变为就绪态的进程读入到内存。

* 页式内存管理

程序的逻辑空间分为固定大小的页，而物理内存划分为同样大小的页框。程序加载时，可以将任意一页放入内存中的任意一个页框，需要CPU硬件支持，来实现逻辑地址和物理地址的映射转换。

优点：没有外部碎片，程序不必连续存放

数据结构：

进程的页表：

操作系统的物理页表，可以采用位示图和空闲页链表。

* 地址变换

逻辑页号和页内地址。逻辑页号+页内偏移->进程页表，物理页号，->物理地址

原理：CPU中的MMU（内存管理单元）按照逻辑页号通过查进程页表得到物理框号，将物理页号与页内地址相加形成物理地址。

* 段式内存管理

进程段、系统段、空闲段

1. C语言程序在内存中的分布

可执行文件，其存储空间包括：

* 代码区：存放CPU执行的集器指令
* 全局初始化区域/静态数据区：包含程序中明确被初始化的全局变量、静态变量和常量
* 未初始化区域，BSS区，存入的是全局未初始化的变量
* 栈区：由编译器自动分配释放，存放函数的参数值、局部变量
* 堆区：动态内存分配，堆在内存中位于BSS区和栈区之间。

内存从上往下：栈区（向下增长）、堆区（向上增长）、BSS区（用0初始化）、数据区（已初始化全局变量、静态变量和常量数据）、代码区（可执行代码）

1. 简单介绍一下tcp协议和ip协议

TCP协议是面向连接的可靠的传输层协议，TCP协议会进行包排序和错误检查，TCP数据包中包括序号和确认，通过重传和ACK机制保证数据传输的可靠性。

IP协议

1. tcp 的可靠性是如何保证的？有哪些因素会影响 tcp 的传输速率？tcp 的流量控制怎么实现？关于 tcp 的拥塞控制和流量控制能具体展开说吗？
2. tcp 的握手为什么是三次不是两次？四次？
3. ip 层的主要作用是什么？ip 层从发出到接收的过程是怎样的？目标主机接收到之后怎么知道这是发给自己的？通过 ip 协议拿到之后怎么找到具体的 mac 地址（ arp 和 rarp ）？
4. MySQL 数据库的索引有哪些？
5. 数组里找两数之和（要求调到最优时间复杂度）。剑指原题。
6. 输入一个正整数 N，返回 N 个 '(' 和 N 个 ')' 的所有可能情况。如N=2，输出)(()，(())等。
7. 只返回上题中括号合法的情况，比如 (())，()() 即为合法。
8. B+ 树了解吗，有哪些特点？为什么用 B+ 树而不用其他树？这个和磁盘预读有什么关系？
9. 输入一个 url 回车到获得响应，经过哪些层，分别涉及什么协议？
10. tcp 和 udp 的区别？
11. tcp 协议里的流量控制和拥塞控制的区别是什么？
12. http 和 https 的区别？
13. https 的加密过程（ SSL 四次握手）？对称加密和非对称加密有什么区别？
14. md5 是加密算法吗？它是怎么解密的？(这里是陷阱，md5 是单向的，不存在解密，某种程度上应该不算一种加密算法。)你刚刚提到彩虹表，你能告诉我彩虹表是怎么破解 md5 的吗？
15. Linux 系统里一个文件在被使用的时候可以被删除吗？
16. 乐观锁和悲观锁是什么？现在让你实现一个乐观锁说说思路？
17. Table 表中有字段 id, name, sex, age, city，求男性平均年龄最大的城市？先说说如何建立索引？写一下这个 sql。

select city from Table where sex='男' group by city order by avg(age) desc limit 1;