

## \*针对项目的问题

### 1.为什么使用在shiro中基于redis

如果使用nginx进行代理，可能会出现在两次登录过程中，第二次所访问的服务器不是第一次访问的服务器，导致无法认证当前状态是否属于已经登录，那么后续的操作也将无法访问呢。

### 2.数据库怎么确保传进去的是当前数据

根据当前用户与序号对应的字段

### 3.怎么响应客户端数据传输成功。

( 同步非阻塞 ) 应该是每一个请求会打开一个线程，如果buffer内不存在数据时，返回给对应的请求已完成

### 4.docker的内存与cpu的区别？以及两台主机分布式A.ip与B.ip怎么交互。

在这个过程中每个服务都在独立的容器里运行，每台机器上都运行着相互不关联的容器，所有容器共享宿主机的cpu、磁盘、网络、内存等，即实现了进程隔离（每个服务独立运行）、文件系统隔离（容器目录修改不影响主机目录）、资源隔离（CPU内存磁盘网络资源独立）。

Docker容器的实现原理就是通过Namespace命名空间实现进程隔离、

在底层的setNamespaces方法中传入进程、用户、网络等，创建新docker容器时把对应的隔离参数传递进去，从而实现了与宿主机、与各个容器的进程、用户、网络隔离。

- 到node1和node2宿主机都是可以正常通信的，由于我们都是设置的不同网段，所以可以在每个宿主机上单独添加一条静态路由，指定数据包的流向：

node1:

```
route add -net 10.10.0.0 /16 gw 192.168.56.12
```

node2:

```
route add -net 172.17.0.0 /16 gw 192.168.56.11
```

这样不同宿主机上的容器就可以互联了。但是维护成本太高了。

- 新建overlay network 使用集群部署

```
member bd93686a68a54c2d is healthy: got healthy result from
http://10.211.55.11:2379
member e5230093897f552c is healthy: got healthy result from
http://10.211.55.9:2379
cluster is healthy
```

## 5.springboot如果一个接口有多个子类，会出现问题？

可以 1、@Autowired 是通过 byType 的方式去注入的，使用该注解，要求接口只能有一个实现类。 2、@Resource 可以通过 byName 和 byType的方式注入，默认先按 byName的方式进行匹配，如果匹配不到，再按 byType的方式进行匹配。 3、@Qualifier 注解可以按名称注入（bean的名称）

## 6.SpringBoot和单纯Spring的区别

SpringBoot基本上是 Spring框架的扩展，它消除了设置 Spring应用程序所需的 XML配置，为更快，更高效的发生态系统铺平了道路。

SpringBoot中的一些特征：

- 1、创建独立的 Spring应用。
- 2、嵌入式 Tomcat、Jetty、Undertow容器（无需部署war文件）。
- 3、提供的 starters 简化构建配置
- 4、尽可能自动配置 spring应用。
- 5、提供生产指标,例如指标、健壮检查和外部化配置
- 6、完全没有代码生成和 XML配置要求

## 第一章 数据库

### 1.数据库中的索引

数据库的索引是用于加快数据的查询速度，索引分为聚簇索引与非聚簇索引。我们通常使用的InnoDB引擎，属于聚簇索引，他的数据查询主要依靠主键索引以及辅助索引，其底层的实现是B+树,叶子节点存储的是数据，通过主键索引可以直接查询到当前的数据，辅助索引的叶子节点存储了主键的key，如果想查询其他数据，他需要借助主键索引；而MyISAM的索引结构数据非聚簇索引，他的仍是B+树为底层，只不过其叶子节点存储的是数据的物理地址，其需要通过地址访问数据。

### 2.Mysql实现在A表不在B表的语句

```
select * from A where id not in (select id from B)
```

3.学生成绩表 table1, 学生、课程、成绩，(name,sbuject,score)，查询出所有课程都大于80分的学生的平均成绩。

```
select avg(score), name from table1 where name not in (select distinct name from table1 where score < 80)
group by name ;
```

### 4.数据库中的范式



- 第一范式：属性不可分
- 第二范式：每个非主属性完全函数依赖键码

- 第三范式：非主属性不传递函数依赖于键码

## 5.Mysql性能优化-索引

1. 在查询的语句的时候，尽量使用全部的复合索引
2. 最佳左前缀法则
3. 不要做以下操作 计算，如：+、-、\*、/、!=、<>、is null、is not null、or 函数，如：sum()、round()等等 手动/自动类型转换，如：id = "1"，本来是数字，给写成字符串了
4. 索引不要放在范围查询的右边
5. 减少使用select \*，使用覆盖索引查询。即：select 查询字段和 where 中使用的索引字段一致。
6. 对于like模糊查询
  - 失效情况
    1. like "%张三%"
    2. like "%张三"
  - 解决方案 使用复合索引，即 like 字段是 select 的查询字段，如：select name from table where name like "%张三%"使用 like "张三%"

## 6.innodb默认的事务隔离级别

 mvcc1  mvcc2 可重复读。(读已提交与可重复读的快照读都是基于MVCC实现的)

- 它的实现原理主要是版本链，undo日志，Read View来实现的RC、RR级别下的InnoDB快照读区别
- 在RR级别下的某个事务的对某条记录的第一次快照读会创建一个快照及Read View，将当前系统活跃的其他事务记录起来，此后在调用快照读的时候，还是使用的是同一个Read View，所以只要当前事务在其他事务提交更新之前使用过快照读，那么之后的快照读使用的都是同一个Read View，所以对之后的修改不可见；
- 即RR级别下，快照读生成Read View时，Read View会记录此时所有其他活动事务的快照，这些事务的修改对于当前事务都是不可见的。而早于Read View创建的事务所做的修改均是可见
- 而在RC级别下的，事务中，每次快照读都会新生成一个快照和Read View，这就是我们在RC级别下的事务中可以看到别的事务提交的更新的原因 此外原子性undolog实现，持久性redolog实现，隔离性通过加锁与Mvvc

## 7.写了redo log，但事务还没提交，突然系统崩溃了会怎么样？

Mysql的事务提交是两阶段的，先写入redolog prepare 再写binlog最后commit,如果崩溃了

- 1.如果 redo log 里面的事务是完整的，也就是已经有了 commit 标识，则直接提交；
- 2.如果 redo log 里面的事务只有完整的 prepare，通过XID去binlog找对应事务, 判断对应的事务 binlog 是否存在并完整 a.如果是，则提交事务； b.否则，回滚事务。

因为 只有redolog其实并没有一开始就写到磁盘 而是写到redologbuffer里面 commit才会写到磁盘;一个事务的binlog 是有完整格式的：1.statement 格式的 binlog，最后会有 COMMIT；2.row 格式的 binlog，最后会有一个 XID event。

## 8.ACID的意思

- 原子性：事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚；

- 一致性：数据库在事务执行前后都保持一致性状态
- 隔离性：一个事务所做的修改在最终提交以前，对其它事务是不可见的。
- 持久性：一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失。

## 9.数据库server层

mysql逻辑分层: 1.client =>连接层=>服务层=>引擎层=>存储层 server 2.连接层: 提供与客户端连接的服务 3.服务层: 1.提供各种用户使用的接口(增删改查),sql解析 sql的解析过程比如: from ... on ... where ... group by ... having ... select ... order by ... limit 2.提供SQL优化器(MySQL Query Optimizer),重写查询,决定表的读取顺序,选择合适的索引 mysql的hint关键字有很多比如:SQL\_NO\_CACHE FORCE\_INDEX SQL\_BUFFER\_RESULT 4.引擎层:innoDB和MyISAM 1.innoDB:事务优先(适合高并发修改操作;行锁) 2.MyISAM:读性能优先 3.show engines;查询支持哪些引擎 4.查看当前默认的引擎 show variables like '%storage\_engine%';default\_storage\_engine

## 10.limit的使用

1.语法：

- limit [offset,] rows

一般是用于select语句中用以从结果集中拿出特定的一部分数据。

offset是偏移量，表示我们现在需要的数据是跳过多少行数据之后的，可以忽略；rows表示我们现在要拿多少行数据。

2.例子:

- select \* from mytbl limit 10000,100 上边SQL语句表示从表mytbl中拿数据，跳过10000行之后，拿100行
- select \* from mytbl limit 0,100 表示从表mytbl拿数据，跳过0行之后，拿取100行
- select \* from mytbl limit 100 这条SQL跟第二条的效果是完全一样的，表示拿前100条数据
- 为了检索从某一个偏移量到记录集的结束所有的记录行，可以指定第二个参数为 -1：mysql> SELECT \* FROM table LIMIT 95,-1; // 检索记录行 96-last

3.用处：前台要展示数据库中数据，需要后台实现分页，传入数据要有“页码page”跟“每页数据条数nums”。对应SQL大概是这样子：select \* from mytbl order by id limit (page-1)\*nums,nums

5.问题分析：

原因出在Limit的偏移量offset上，比如limit 100000,10虽然最后只返回10条数据，但是偏移量却高达100000，数据库的操作其实是拿到100010数据，然后返回最后10条。

那么解决思路就是，我能不能跳过100000条数据然后读取10条，而不是读取100010条数据然后返回10条数据。

6.问题解决实现： select \* from mytbl where id >= (select id from mytbl order by id limit 100000,1) limit 10 注：假设id是主键索引，那么里层走的是索引，外层也是走的索引，所以性能大大提高

## 11.innodb和myisam在什么情况下选择哪个，为什么

- 处理大数据容量的数据库系统。MySQL 运行的时候，InnoDB 会在内存中建立缓冲池，用于缓冲数据和索引；此外innodb属于行锁，锁的的粒度小，写操作不会锁住全表，并发度较高的场景下使用会提升效率的。
- MySQL 的默认引擎，但不提供事务的支持，也不支持行级锁和外键。因此当执行插入和更新语句时，即执行写操作的时候需要锁定这个表，所以会导致效率会降低。如果表的读操作远远多于写操作时，并且不需要事务的支持的，可以将 MyIASM 作为数据库引擎的首选。

## 12.创建索引的方法

Create index indexName on tableName(fieldName)

## 13.数据库怎么保证一致性

在数据库层面主要通过保证原子性、隔离性、持久性来保证数据的一致性，也就是我们要基于undolog、加锁与MVCC、redolog

## 14.为什么binlog代替不了redolog

Mysql的事务提交是两阶段的，先写入redolog prepare 再写binlog最后commit,如果只有binlog,就无法知道日志是否已经在磁盘还是在内存中，在引擎层发生了数据更新，事务在binlog写完之后奔溃而为提交，回滚再次执行binlog就会产生两次不一样的结果（此时 binlog 来恢复的时候就多了一个事务出来，所恢复的值与原来的值不一样）

## 15.redolog undolog特点是啥

- redolog通常是物理日志，记录的是数据页的物理修改，而不是某一行或者某几行的修改，它用来恢复提交后的物理数据页。它基于磁盘上的重做日志文件（redo log file）来实现事务的持久化。
- undolog用来回滚行记录到的某个版本。undolog一般是逻辑日志，根据每行记录进行记录。它用来提供回滚和多个行版本控制(MVCC)。为了满足事务的原子性，在操作任何数据之前，首先将数据备份到一个地方（这个存储数据备份的地方称为Undo Log）。然后进行数据的修改。如果出现了错误或者用户执行了ROLLBACK语句，系统可以利用Undo Log中的备份将数据恢复到事务开始之前的状态。

## 16.binlog的用处；从服务器主动拉binlog，还是主服务器推binlog

binlog是一个二进制格式的文件，记录所有对数据库表结构产生变更的操作；

- 1.数据恢复：可用于回档或故障迁移恢复。比如腾讯云MySQL实例的回档功能基于冷备数据+binlog实现，可以将云数据库或表回档到指定时间。
- 2.主从复制：MySQL通过主从复制保持所有主从节点的数据一致性，这个过程依赖于binlog的传输回放等。具体流程在下面有详细介绍。
- 3.数据审计：通过binlog记录的SQL语句，可用来审计追踪用户操作。

主从复制模式:

- 1. 异步复制:从服务器主动拉binlog;这种模式下主节点将所有更新写入binlog文件之后就返回客户端结果，而不会主动推送binlog到从节点。一旦主节点宕机，从节点会因为同步到最新binlog导致数据丢失。相对地，该模式处理效率更高，响应速度快。mysql 主从复制默认是异步模式。

- 2. 半同步复制:这种模式下主节点在本地写入binlog后，会主动推送binlog到从节点。在接收到其中一台从节点的返回信息后，再返回给客户端，否则需要等待直到超时时间然后切换成异步模式再提交。相比异步复制，提高了数据安全性，也降低了主从延迟。但性能上会有一定的降低，响应时间会变长。
- 3. 全同步复制:全同步复制是指主节点和全部从节点都执行了commit并确认才会向客户端返回成功。在牺牲了效率的前提下，保证了数据的强一致性。

## 17.那你平时连接数据库使用的是BIO还是NIO呢？为什么不用NIO呢？

- BIO
- 1.JDBC似乎不支持NIO的来连接方式 2.假设一个采用该技术的dbserver，使用单线程监听。当有多个数据库的操作过来，也就是发生了多个事件，对于每个事件假设都是向db中写数据，那采用io复用逐一的写入。注意，仍然是单线程的，万一不幸的是第一个fd的写入操作长时间阻塞，这得导致后面多少事件超时啊。

## 18.数据库索引有没有什么办法A、C与A、B、C同时走索引

能不能把新建索引 ( Create index indexName on tableName(A,C,B))？注意：数据库会进行自动优化 (1) select \* from myTest where a=3 and b=5 and c=4; ---- abc顺序 abc三个索引都在where条件里面用到了，而且都发挥了作用

(2) select \* from myTest where c=4 and b=6 and a=3; where里面的条件顺序在查询之前会被mysql自动优化，效果跟上一句一样

## 19.防止数据库的注入

- 1、检查变量数据类型和格式
- 2、过滤特殊符号(使用正则表达式过滤传入的参数)；字符串过滤
- 3、绑定变量，使用预编译语句(PreparedStatement)

## 20.数据库隔离性的实现

- 写-写操作：锁
- 写-读操作：MVCC ( 读会出现脏读、不可重复读、幻读的情况 )

## 21.什么是事务，什么场景下用事务

事务指的是满足 ACID 特性的一组操作，可以通过 Commit 提交一个事务，也可以使用 Rollback 进行回滚。事务的使用场景: 在日常生活中，有时我们需要进行银行转账，这个银行转账操作背后就是需要执行多个SQL语句，假如这些SQL执行到一半突然停电了，那么就会导致这个功能只完成了一半，这种情况是不允许出现，要想解决这个问题就需要通过事务来完成。

## 22.读写分离

主从复制、读写分离一般是一起使用的。目的很简单，就是为了提高数据库的并发性能。你想，假设是单机，读写都在一台MySQL上面完成，性能肯定不高。如果有三台MySQL，一台mater只负责写操作，两台salve只负责读操作，性能不就能大大提高了吗？

- 实现的方式有很多，可以采用AOP的方式，通过方法名判断，方法名中有get、select、query开头的则连接slave，其他的则连接master数据库。
- Apache ShardingSphere 是一套开源的分布式数据库中间件解决方案组成的生态圈，它由 JDBC、Proxy两部分组成。

### 23.左连接、右连接、内连接、外连接

- left join（左连接，左外连接）：返回包括左表中的所有记录和右表中连接字段相等的记录。

```
select a.name,b.job from A a left join B b on a.id=b.A_id
```

- right join（右连接，右外连接）：返回包括右表中的所有记录和左表中连接字段相等的记录。

```
select a.name,b.job from A a right join B b on a.id=b.A_id
```

- inner join（等值连接或者叫内连接）：只返回两个表中连接字段相等的行。

```
select a.name,b.job from A a inner join B b on a.id=b.A_id
```

- full join（全外连接）：返回左右表中所有的记录和左右表中连接字段相等的记录。

```
select a.name,b.job from A a full join B b on a.id=b.A_id
```

### 24.数据库中锁的分类？间隙锁的实现原理

- 行级锁：是一种排他锁，防止其他事务修改此行；行级锁是Mysql中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。行级锁分为共享锁和排他锁。特点：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。
- 页级锁：页级锁是MySQL中锁定粒度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。所以取了折衷的页级，一次锁定相邻的一组记录。BDB支持页级锁。特点：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。
- 表级锁：锁定粒度大，发生锁冲突的概率最高,并发度最低。特点：开销小，加锁快；不会出现死锁(因为MyISAM会一次性获得SQL所需的全部锁)；

当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB 会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙 (GAP)”，InnoDB 也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁 (Next-Key 锁)。举例来说，假如 user 表中只有 101 条记录，其 userid 的值分别是 1,2,...,100,101，下面的 SQL：

```
Select * from user where userid > 100 for update;
```

是一个范围条件的检索，InnoDB 不仅会对符合条件的 `userid` 值为 101 的记录加锁，也会对 `userid` 大于 101（但是这些记录并不存在）的“间隙”加锁，防止其它事务在表的末尾增加数据。

（间隙锁，是在索引的间隙之间加上锁，这是为什么 **Repeatable Read** 隔离级别下能防止幻读的主要原因。）  
InnoDB 使用间隙锁的目的，为了防止幻读，以满足串行化隔离级别的要求，对于上面的例子，要是不使用间隙锁，如果其他事务插入了 `userid` 大于 100 的任何记录，那么本事务如果再次执行上述语句，就会发生幻读。

## 第二种分类方法 锁级别分类

共享锁 & 排他锁 & 意向锁

### 1 共享锁 ( Share Lock)

共享锁又称读锁，是读取操作创建的锁。其他用户可以并发读取数据，但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。

如果事务T对数据A加上共享锁后，则其他事务只能对A再加共享锁，不能加排他锁。获准共享锁的事务只能读数据，不能修改数据。

用法

```
SELECT ... LOCK IN SHARE MODE;
```

在查询语句后面增加 `LOCK IN SHARE MODE`，MySQL 就会对查询结果中的每行都加共享锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请共享锁，否则会被阻塞。其他线程也可以读取使用了共享锁的表，而且这些线程读取的是同一个版本的数据。

### 2 排他锁 ( Exclusive Lock)

排他锁又称写锁、独占锁，如果事务T对数据A加上排他锁后，则其他事务不能再对A加任何类型的封锁。获准排他锁的事务既能读数据，又能修改数据。

用法

```
SELECT ... FOR UPDATE;
```

在查询语句后面增加 `FOR UPDATE`，MySQL 就会对查询结果中的每行都加排他锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请排他锁，否则会被阻塞。

### 3 意向锁 ( Intention Lock)

意向锁是表级锁，其设计目的主要是为了在一个事务中揭示下一行将要被请求锁的类型。InnoDB 中的两个表锁：

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 )：类似上面，表示事务准备给数据行加入排他锁，说明事务在一个数据行加排他锁前必须先取得该表的IX锁。

意向锁是 InnoDB 自动加的，不需要用户干预。

对于 `INSERT`、`UPDATE` 和 `DELETE`，InnoDB 会自动给涉及的数据加排他锁；对于一般的 `SELECT` 语句，InnoDB 不会加任何锁，事务可以通过以下语句显式加共享锁或排他锁。



共享锁：SELECT ... LOCK IN SHARE MODE;

排他锁：SELECT ... FOR UPDATE;

## 25.从数据库表查询语文成绩及格但是平均分不及格的学生

```
select student.id,student.score,a.avg from student
inner JOIN (
    select id,avg(score) as avg from student
    group by id having avg(score) < 60
) a
on student.id = a.id where student.name='Chinese' and score >= 60
```

## 26.什么是幻读，数据库是怎么保证不出现幻读

- T1 读取某个范围的数据，T2 在这个范围内插入新的数据，T1 再次读取这个范围的数据，此时读取的结果和第一次读取的结果不同。
- （快照读：）MVCC中将readview中包含了多个查询，其数据结果要么同时提交，要么同时回滚。（当前读：行锁+间隙锁）

## 27.数据库随机获取10条数据

```
SELECT id FROM user ORDER BY RAND() LIMIT 10;
```

## 28.数据库如何避免死锁

- 1) 以固定的顺序访问表和行。即按顺序申请锁，这样就不会造成互相等待的场面。
- 2) 大事务拆小。大事务更倾向于死锁，如果业务允许，将大事务拆小。
- 3) 在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁概率。
- 4) 降低隔离级别。如果业务允许，将隔离级别调低也是较好的选择，比如将隔离级别从RR调整为RC，可以避免掉很多因为gap锁造成的死锁。
- 5) 为表添加合理的索引。如果不走索引将会为表的每一行记录添加上锁，死锁的概率大大增大。

## 29.当前读与快照读

当前读：它读取的数据库记录们都是当前最新的版本，会对当前读取的数据进行加锁，防止其他是修改数据，是悲观锁的一种操作。

- select lock in share mode(共享锁)
- select for update(排他锁)
- update(排他锁)
- insert(排他锁)
- delete(排他锁)

- 串行化事务隔离级别 快照读：快照读的实现是基于多版本并发控制，快照读读到的数据不一定是当前最新的数据，可能是之前那的历史版本的数据 不加锁的select操作（注：事务级别不是串行化）

## 第二章 Redis

### 1.缓存穿透，怎么解决

- 数据库无论查询出什么结果，都缓存到redis中
- 把ip拉黑
- 对参数进行合法校验，对不合法参数直接return
- 布隆过滤器

### 2.缓存雪崩，怎么解决

- 不设置缓存时间
- 跑定时任务，定时刷新这个缓存

### 3.缓存的击穿

- 缓存永远不过期（不太好）
- 使用分布式锁，单体应用可以使用：互斥锁；（有一个线程有锁，去使用数据库查询，让其他线程处于等待几毫秒，这时将数据库的查询到的数据放入缓存中，其他线程访问时redis就可以查到这些数据了）

### 4.缓存与数据库数据一致性

- 先删除缓存，再更新数据库。解决方案是使用延迟双删。
- 先更新数据库，再删除缓存。解决方案是消息队列或者其他binlog同步，引入消息队列会带来更多的问题，并不推荐直接使用。

### 5.redis的持久化存储

- AOF 日志：文件的内容是操作命令
- RDB 快照：文件的内容是二进制数据（RDB 快照就是记录某一个瞬间的内存数据，记录的是实际数据，而 AOF 文件记录的是命令操作的日志，而不是实际的数据。）

因此在 Redis 恢复数据时，RDB 恢复数据的效率会比 AOF 高些，因为直接将 RDB 文件读入内存就可以，不需要像 AOF 那样还需要额外执行操作命令的步骤才能恢复数据。

## 第三章 Java基础

### 1.内存分配的方式

- 程序计数器:记录正在执行的虚拟机字节码指令的地址（如果正在执行的是本地方法则为空）。
- Java 虚拟机栈:每个 Java 方法在运行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。
- 本地方法栈：本地方法栈与 Java 虚拟机栈类似，它们之间的区别只不过是本地方法栈为本地方法服务。本地方法一般是用其它语言（C、C++ 或汇编语言等）编写的，并且被编译为基于本机硬件和操作系统的程序，对待这些方法需要特别处理

- 堆：所有对象都在这里分配内存，是垃圾收集的主要区域（"GC 堆"）。现代的垃圾收集器基本都是采用分代收集算法，其主要的思想是针对不同类型的对象采取不同的垃圾回收算法。
- 方法区：用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。和堆一样不需要连续的内存，并且可以动态扩展，动态扩展失败一样会抛出 `OutOfMemoryError` 异常。

## 2.为什么HashMap的扩容机制是2倍

因为是通过除留余数法获取桶号，而Hash表的大小始终为2的n次幂，因此可以将取模转为位运算提高效率容量n为2的幂次方，n-1的二进制会全为1，位运算时可以充分散列，避免不必要的哈希冲突，这也就是为什么要按照2倍方式扩容的一个原因

## 3.Hashmap的查询

put操作的流程：

第一步：`key.hashCode()`，时间复杂度 $O(1)$ 。

第二步：找到桶以后，判断桶里是否有元素，如果没有，直接new一个entey节点插入到数组中。时间复杂度 $O(1)$ 。

第三步：如果桶里有元素，并且元素个数小于6，则调用equals方法，比较是否存在相同名字的key，不存在则new一个entry插入都链表尾部。时间复杂度 $O(1)+O(n)=O(n)$ 。

第四步：如果桶里有元素，并且元素个数大于6，则调用equals方法，比较是否存在相同名字的key，不存在则new一个entry插入都链表尾部。时间复杂度 $O(1)+O(\log n)=O(\log n)$ 。红黑树查询的时间复杂度是 $\log n$ 。（此时桶内为红黑树，虽然小于8个数量，但是原来属于红黑树，只有当小于6时才从红黑树退化为链表）

通过上面的分析，我们可以得出结论，HashMap新增元素的时间复杂度是不固定的，可能的值有 $O(1)$ 、 $O(\log n)$ 、 $O(n)$ 。。

## 4.JVM的CMS的步骤？为什么要执行第二步的并发标记？

- 初始标记：仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要停顿。
- 并发标记：进行 GC Roots Tracing 的过程，它在整个回收过程中耗时最长，不需要停顿。
- 并发预清理：是减少重标记（Remark）步骤Stop-the-World的时间。这一步同样也是并发的，不会停止用户应用线程。在前面的并发标记中，一些引用被改变了。当某一块块（Card）中的对象引用发生改变时，JVM会标记这个空间为“脏块”（Dirty Card）
- 重新标记：为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，需要停顿。
- 并发清除：不需要停顿

## 5.GCRoot的对象有哪些？

目前的gC主要是基于可达性分析算法，通过一系列的名为“GC Root”的对象作为起点，从这些节点向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到GC Root没有任何引用链相连时，则该对象不可达，该对象是不可使用的，垃圾收集器将回收其所占的内存。GCroot的对象包括以下几种：

- java虚拟机栈（栈帧中的本地变量表）中的引用的对象
- 方法区中的静态属性引用的对象
- 方法区中的常量引用的对象

- 本地方法栈中JNI本地方法的引用对象。（GC管理的主要区域是Java堆，一般情况下只针对堆进行垃圾回收。方法区、栈和本地方法区不被GC所管理,因而选择这些区域内的对象作为GC roots,被GC roots引用的对象不被GC回收。）

## 6.G1收集器的收集算法

- 初始标记
- 并发标记
- 最终标记：为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程的 Remembered Set Logs 里面，最终标记阶段需要把 Remembered Set Logs的数据合并到 Remembered Set 中。这阶段需要停顿线程，但是可并行执行。
- 筛选回收：首先对各个 Region 中的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划。此阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率。

空间整合：整体来看是基于“标记 - 整理”算法实现的收集器，从局部（两个 Region 之间）上来看是基于“复制”算法实现的，这意味着运行期间不会产生内存空间碎片。

## 7.聊一下AQS

- AQS的核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并将共享资源设置为锁定状态，如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制
- 实现了AQS的锁有：自旋锁、互斥锁、读锁写锁、条件产量、信号量、栅栏都是AQS的衍生物
- 1.Exclusive：独占，只有一个线程能执行，如ReentrantLock 2.Share：共享，多个线程可以同时执行，如Semaphore、CountDownLatch、ReadWriteLock、CyclicBarrier

## 8.聊一下异常？为什么异常分为运行时异常和非运行时异常



- Error 一般是指java虚拟机相关的问题，如系统崩溃、虚拟机出错误、动态链接失败等，这种错误无法恢复或不可能捕获，将导致应用程序中断，通常应用程序无法处理这些错误，因此应用程序不应该捕获Error对象，也无须在其throws子句中声明该方法抛出任何Error或其子类。
- (1)运行时异常都是RuntimeException类及其子类异常，如NullPointerException、IndexOutOfBoundsException等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。
- (2)非运行时异常是RuntimeException以外的异常，类型上都属于Exception类及其子类。如IOException、SQLException等以及用户自定义的Exception异常。对于这种异常，JAVA编译器强制要求我们必需对出现的这些异常进行catch并处理，否则程序就不能编译通过。所以，面对这种异常不管我们是否愿意，只能自己去写一大堆catch块去处理可能的异常。

## 9.cas的ABA问题怎么解决的

- ABA问题：当有多个线程对一个原子类进行操作的时候，某个线程在短时间内将原子类的值A修改为B，又马上将其修改为A，此时其他线程不感知，还是会修改原子类的值A成功，而这和原本设计目的相悖。
- 解决：1.使用数据库乐观锁解决ABA问题 实现思路：表中加一个VNO版本号字段，每次修改VNO = VNO + 1，则下一次修改则修改失败。2.使用AtomicStampedReference解决ABA问题 实现思路：本质是

有一个 int 值作为版本号，每次更改前先取到这个int值的版本号，等到修改的时候，比较当前版本号与当前线程持有的版本号是否一致，如果一致，则进行修改，并将版本号+1（当然加多少或减多少都是可以自己定义的），在zookeeper中保持数据的一致性也是用的这种方式。

## 10.JAVA SDK起到的作用

SDK软件开发工具；助开发某一类软件的相关文档、API必需资料、范例和工具的集合都可以叫做 "SDK"

## 11.Arrays.sort底层的排序算法

- 快速排序
- 插入排序
- 归并排序

一个数组进来遇到第一个阈值QUICKSORT\_THRESHOLD (286) < length < INSERTION\_SORT\_THRESHOLD (47) 使用快速排序；如果小于INSERTION\_SORT\_THRESHOLD (47) 则使用插入排序；

对于大于QUICKSORT\_THRESHOLD (286)；需要进行数组结构判断。实际逻辑是分组排序，每降序为一个组，像1,9,8,7,6,8。9到6是降序，为一个组，然后把降序的一组排成升序：1,6,7,8,9,8。然后最后的8后面继续往后面找。。。

每遇到这样一个降序组，++count，当count大于MAX\_RUN\_COUNT (67)，被判断为这个数组不具备结构（也就是这数据时而升时而降），然后送给之前的sort(里面的快速排序)的方法（The array is not highly structured,use Quicksort instead of merge sort.）。

如果count少于MAX\_RUN\_COUNT (67) 的，说明这个数组还有点结构，就继续往下走下面的归并排序：

## 12.java类加载过程；

- 加载=》验证=》准备=》解析=》初始化

出现以下几种情况的时候需要我们在自己写自定义Java的类加载器应用到加载阶段

- 我们需要的类不一定存放在已经设置好的classPath下(有系统类加载器AppClassLoader加载的路径)，对于自定义路径中的class类文件的加载，我们需要自己的ClassLoader
- 有时我们不一定是从类文件中读取类，可能是从网络的输入流中读取类，这就需要做一些加密和解密操作，这就需要自己实现加载类的逻辑，当然其他的特殊处理也同样适用
- 可以定义类的实现机制，实现类的热部署,如OSGi中的bundle模块就是通过实现自己的ClassLoader实现的。

## 13.双亲委派原则的机制

该模型要求除了顶层的启动类加载器外，其它的类加载器都要有自己的父类加载器。这里的父子关系一般通过组合关系（Composition）来实现，而不是继承关系（Inheritance）。

1. 工作过程 一个类加载器首先将类加载请求转发到父类加载器，只有当父类加载器无法完成时才尝试自己加载。
2. 好处 使得 Java 类随着它的类加载器一起具有一种带有优先级的层次关系，从而使得基础类得到统一。  
例如 java.lang.Object 存放在 rt.jar 中，如果编写另外一个 java.lang.Object 并放到 ClassPath 中，程序可

以编译通过。由于双亲委派模型的存在，所以在 `rt.jar` 中的 `Object` 比在 `ClassPath` 中的 `Object` 优先级更高，这是因为 `rt.jar` 中的 `Object` 使用的是启动类加载器，而 `ClassPath` 中的 `Object` 使用的是应用程序类加载器。`rt.jar` 中的 `Object` 优先级更高，那么程序中所有的 `Object` 都是这个 `Object`。

3. 实现 以下是抽象类 `java.lang.ClassLoader` 的代码片段，其中的 `loadClass()` 方法运行过程如下：先检查类是否已经加载过，如果没有则让父类加载器去加载。当父类加载器加载失败时抛出 `ClassNotFoundException`，此时尝试自己去加载。

## 14.类加载器的分类

- 启动类加载器 ( `Bootstrap ClassLoader` ) 此类加载器负责将存放在 `<JRE_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。启动类加载器无法被 Java 程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给启动类加载器，直接使用 `null` 代替即可。
- 扩展类加载器 ( `Extension ClassLoader` ) 这个类加载器是由 `ExtClassLoader` ( `sun.misc.Launcher$ExtClassLoader` ) 实现的。它负责将 `<JAVA_HOME>/lib/ext` 或者被 `java.ext.dir` 系统变量所指定路径中的所有类库加载到内存中，开发者可以直接使用扩展类加载器。
- 应用程序类加载器 ( `Application ClassLoader` ) 这个类加载器是由 `AppClassLoader` ( `sun.misc.Launcher$AppClassLoader` ) 实现的。由于这个类加载器是 `ClassLoader` 中的 `getSystemClassLoader()` 方法的返回值，因此一般称为系统类加载器。它负责加载用户类路径 ( `ClassPath` ) 上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认类加载器。

## 15.HashMap为什么小于6是链表，大于8变成红黑树

`treeify` 过程会把原本的 `Node` 对象转化为 `TreeNode` 对象，而 `TreeNode` 大小是 `Node` 的两倍；除去内存的损耗，`treeify` 本身也是一个耗时的过程，并且在红黑树节点数小于等于 `UNTREEIFY_THRESHOLD` ( 默认为 6 ) 时，红黑树又会重新转换为链表，如果出现频繁的相互转化，这是一笔不小的开销。通过设置阈值更多的体现了时间和空间平衡的思想

## 16.HashMap是否线程安全，例子

不安全，如果存在两个线程，其中一个线程出现扩容，会出现链表的顺序被反转，此时另一个线程被调度回来，数据会出现环形链。

## 17.ConcurrentHashMap和HashTable的区别

- 在JDK1.7的时候，`ConcurrentHashMap` ( 分段锁 ) 对整个桶数组进行了分割分段(`Segment`)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。( 默认分配16个`Segment`，比`Hashtable`效率提高16倍。 ) 到了JDK1.8的时候已经摒弃了`Segment`的概念，而是直接用 `Node` 数组+链表+红黑树的数据结构来实现，并发控制使用 `synchronized` 和 `CAS` 来操作。摒弃了分段锁的概念，启用 `node + CAS + Synchronized` 代替`Segment`。当前的 `table[(n - 1) & hash]` == `null` 时，采用`CAS`操作；当产生`hash`冲突时，采用`synchronized`关键字
- `Hashtable`(同一把锁):使用 `synchronized` 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 `put` 添加元素，另一个线程不能使用 `put` 添加元素，也不能使用 `get`，竞争会越来越激烈效率越低。

## 18.CAS能保证线程安全吗，volatile关键字能保证线程安全吗；

如果从原子性上来说的话是安全的，cas是原子的，但是可能会存在ABA问题，这个问题可以基于（原子的int类记录每一次的修改得到解决），但是cas存在可见性的问题，CAS操作自己是保证可见行的，并采用了内存偏移量的独特方式。这类修改结果对于其他并发的普通get操作不会立即可见，还是要靠volatile修饰变量、让读操作从主存读取最新值。

### 19.ArrayList怎么扩容，时间复杂度O(n)？插尾部O(1)，平均是多少，答案O(2)需要考虑扩容

当数组A满了则会触发数据扩容，即创建一个新的数组B，将数组A中所有的元素复制到数组B中，这个操作的复杂度为O(n)。🖼️arraylist 上图中扩容到2N为笔者任意设定，读者也可自定（至少要保证倍数大于1）。（实际扩容为1.5倍）不论扩容的倍数为多少，均摊复杂度都为O(1)。

### 20.Java对编译做了哪些优化？

- 泛型与类型的擦除
- 自动装箱、拆箱与遍历循环
- 逃逸分析

### 21.OOM (“Out of Memory”)

1. 年老代溢出，表现为：java.lang.OutOfMemoryError:Javaheap space 这是最常见的情况，产生的原因可能是：设置的内存参数Xmx过小或程序的内存泄露及使用不当问题。例如循环上万余次的字符串处理、创建上千万个对象、在一段代码内申请上百M甚至上G的内存。还有的时候虽然不会报内存溢出，却会使系统不间断的垃圾回收，也无法处理其它请求。这种情况下除了检查程序、打印堆内存等方法排查，还可以借助一些内存分析工具，比如MAT就很不错。
2. 持久代溢出，表现为：java.lang.OutOfMemoryError:PermGenspace 通常由于持久代设置过小，动态加载了大量Java类而导致溢出，解决办法唯有将参数 -XX:MaxPermSize 调大（一般256m能满足绝大多数应用程序需求）。将部分Java类放到容器共享区（例如Tomcat share lib）去加载的办法也是一个思路，但前提是容器里部署了多个应用，且这些应用有大量的共享类库。
3. 默认情况下，并不是等堆内存耗尽，才会报 OutOfMemoryError，而是如果 JVM 觉得 GC 效率不高，也会报这个错误。
4. 这个在创建太多的线程，超过系统配置的极限。如Linux默认允许单个进程可以创建的线程数是1024个。
5. 向系统申请直接内存时，如果系统可用内存不足，就会抛出这个异常，

### 22.Java的线程池知道吗？说一下执行流程，又给了一个场景现在没有执行的线程了，线程池是什么样的状态？

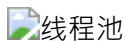
线程池：是一种基于池化思想管理线程的工具，经常出现在多线程服务器中，线程过多会带来额外的开销，其中包括创建销毁线程的开销、调度线程的开销等等，同时也降低了计算机的整体性能。线程池维护多个线程，等待监督管理者分配可并发执行的任务。这种做法，一方面避免了处理任务时创建销毁线程开销的代价，另一方面避免了线程数量膨胀导致的过分调度问题，保证了对内核的充分利用。

- 降低资源消耗：通过池化技术重复利用已创建的线程，降低线程创建和销毁造成的损耗。
- 提高响应速度：任务到达时，无需等待线程创建即可立即执行。
- 提高线程的可管理性：线程是稀缺资源，如果无限制创建，不仅会消耗系统资源，还会因为线程的不合理分布导致资源调度失衡，降低系统的稳定性。使用线程池可以进行统一的分配、调优和监控。

- 提供更多更强大的功能：线程池具备可拓展性，允许开发人员向其中增加更多的功能。比如延时定时线程池ScheduledThreadPoolExecutor，就允许任务延期执行或定期执行。

执行的过程：

- 首先检测线程池运行状态，如果不是RUNNING，则直接拒绝，线程池要保证在RUNNING的状态下执行任务。
- 如果workerCount < corePoolSize，则创建并启动一个线程来执行新提交的任务。
- 如果workerCount >= corePoolSize，且线程池内的阻塞队列未满，则将任务添加到该阻塞队列中。
- 如果workerCount >= corePoolSize && workerCount < maximumPoolSize，且线程池内的阻塞队列已满，则创建并启动一个线程来执行新提交的任务。
- 如果workerCount >= maximumPoolSize，并且线程池内的阻塞队列已满，则根据拒绝策略来处理该任务，默认的处理方式是直接抛异常。



线程的状态

- RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。
- SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。
- STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。
- TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为TIDYING 状态时，会执行钩子方法 terminated()。
- TERMINATED：terminated()方法结束后，线程池的状态就会变成这个。

## 23.maven命令如何指定加载某个jar包而避免冲突

- version ?

1、Maven默认处理策略 最短路径优先：Maven 面对 D1 和 D2 时，会默认选择最短路径的那个 jar 包，即 D2。E->F->D2 比 A->B->C->D1 路径短 1。

最先声明优先：如果路径一样的话，如：A->B->C1, E->F->C2，两个依赖路径长度都是 2，那么就选择最先声明。

2、移除依赖：用于排除某项依赖的依赖jar包 或者手动在pom.xml中使用标签去排除冲突的jar包

## 24.双亲加载的好处？破坏双亲加载的几种方式，破坏双亲加载是为了什么

1.好处：

- 首先，==通过委派的方式，可以避免类的重复加载==，当父加载器已经加载过某一个类时，子加载器就不会再重新加载这个类。
- 另外，==通过双亲委派的方式，还保证了安全性==。因为Bootstrap ClassLoader在加载的时候，只会加载JAVA\_HOME中的jar包里面的类，如java.lang.Integer，那么这个类是不会被随意替换的，除非有人跑到你的机器上，破坏你的JDK。

2.破坏加载的几种方式：

- 自定义一个类加载器，重写其中的loadClass方法，使其不进行双亲委派即可。



( 这里面需要展开讲一下loadClass和findClass，我们前面说过，当我们想要自定义一个类加载器的时候，并且像破坏双亲委派原则时，我们会重写loadClass方法。那么，如果我们想定义一个类加载器，但是不想破坏双亲委派模型的时候呢？这时候，就可以继承ClassLoader，并且重写findClass方法。findClass()方法是JDK1.2之后的ClassLoader新添加的一个方法。 )

### 3.为什么破坏双亲加载

- 比如典型的JDBC服务，DriverManager会先被类加载器加载，因为java.sql.DriverManager类是位于rt.jar下面的，所以他会根加载器加载。==DriverManager是被根加载器加载的，那么在加载时遇到以上代码，会尝试加载所有Driver的实现类，但是这些实现类基本都是第三方提供的，根据双亲委派原则，第三方的类不能被根加载器加载。于是，就在JDBC中通过引入ThreadContextClassLoader（线程上下文加载器，默认情况下是AppClassLoader）的方式破坏了双亲委派原则。==
- Tomcat是web容器，那么一个web容器可能需要部署多个应用程序。不同的应用程序可能会依赖同一个第三方类库的不同版本，但是不同版本的类库中某一个类的全路径名可能是一样的。如多个应用都要依赖hollis.jar，但是A应用需要依赖1.0.0版本，但是B应用需要依赖1.0.1版本。这两个版本中都有一个类是com.hollis.Test.class。==如果采用默认的双亲委派类加载机制，那么是无法加载多个相同的类。所以，Tomcat破坏双亲委派原则，提供隔离的机制，为每个web容器单独提供一个WebAppClassLoader加载器。==
- 模块化技术与类加载机制 近几年模块化技术已经很成熟了，在JDK 9中已经应用了模块化的技术。其实早在JDK 9之前，==OSGI这种框架已经是模块化的了，而OSGI之所以能够实现模块热插拔和模块内部可见性的精准控制都归结于其特殊的类加载机制，加载器之间的关系不再是双亲委派模型的树状结构，而是发展成复杂的网状结。在JDK中，双亲委派也不是绝对的了==。在JDK9之前，JVM的基础类以前都是在rt.jar这个包里，这个包也是JRE运行的基石。这不仅是违反了单一职责原则，同样程序在编译的时候会将很多无用的类也一并打包，造成臃肿。==在JDK9中，整个JDK都基于模块化进行构建，以前的rt.jar, tool.jar被拆分成数十个模块，编译的时候只编译实际用到的模块，同时各个类加载器各司其职，只加载自己负责的模块。==

## 25.java nio模型介绍



- NIO支持面向缓冲区的、基于通道的IO操作。NIO将以更加高效的方式进行文件的读写操作。NIO可以理解为非阻塞IO,传统的IO的read和write只能阻塞执行，线程在读写IO期间不能干其他事情，比如调用socket.read()时，如果服务器一直没有数据传输过来，线程就一直阻塞，而NIO中可以配置socket为非阻塞模式。
- NIO 有三大核心部分：**Channel( 通道)**，**Buffer( 缓冲区)**，**Selector( 选择器)**
- Java NIO 的非阻塞模式，使一个线程从某通道发送请求或者读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取，而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。非阻塞写也是如此，一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。
- 通俗理解：NIO 是可以做到用一个线程来处理多个操作的。假设有 1000 个请求过来,根据实际情况，可以分配20 或者 80个线程来处理。不像之前的阻塞 IO 那样，非得分配 1000 个。

NIO	BIO
面向缓冲区 ( Buffer)	面向流 ( Stream)
非阻塞 ( Non Blocking IO)	阻塞IO(Blocking IO)

**NIO****BIO**

选择器 ( Selectors)

**26.spring的自动装载机制 ( spi机制 )**

是使用的@SpringApplication注解，然后使用@EnableAutoConfiguration以及@ComponentScan自动装配，注解@EnableAutoConfiguration使用了@Import加载，最后使用了SpringFactoriesLoader反射出maven中META-INF下spring.factories。说的很对，的确是这样的。

**27.SpringBoot和单纯Spring的区别**

Spring框架为开发 Java应用程序提供了全面的基础架构支持。它包含一些很好的功能，如依赖注入和开箱即用的模块，如：SpringJDBC、SpringMVC、SpringSecurity、SpringAOP、SpringORM、SpringTest，这些模块缩短应用程序的开发时间，提高了应用开发的效率例如，在 JavaWeb开发的早期阶段，我们需要编写大量的代码来将记录插入到数据库中。


SpringBoot基本上是 Spring框架的扩展，它消除了设置 Spring应用程序所需的 XML配置，为更快，更高效的开发生态系统铺平了道路。SpringBoot中的一些特征：

- 1、创建独立的 Spring应用。
- 2、嵌入式 Tomcat、Jetty、Undertow容器（无需部署war文件）。
- 3、提供的 starters 简化构建配置
- 4、尽可能自动配置 spring应用。
- 5、提供生产指标,例如指标、健壮检查和外部化配置
- 6、完全没有代码生成和 XML配置要求

**28.Spring IOC AOP原理，Spring loc中 bean的顶层接口是什么**

IOC的原理？好处？原理：“控制反转”：借助于“第三方”实现具有依赖关系的对象之间的解耦，通过容器，是两个对象之间失去直接的联系，假设当对象A运行到需要对象B的时候，容器会主动创建一个对象B注入到对象A需要的地方。好处：1.各个类之间只有与容器连接时才具有相关性，所以任何一方出问题都不会影响到另一方的运行，增加了维护性与单元测试性，便于调试程序。2.由于容器与类之间的无关性，在保证接口标准的情况下，可以将一个大中型项目分割为多个子项目，团队成员分工明确，提高产品的开发效率；3.具有可重用性

AOP原理：通过动态代理的方式进行实现，主要包括JDK动态代理（代理对象必须必须是接口，核心·InvocationHandler和Proxy）和cglib动态代理（cglib是一个代码生成的类库，可以在运行时动态生成某个类的子类）。好处：降低模块之间的耦合，2.使系统更容易扩展。3.避免修改业务代码，避免引入重复的代码，有更好的重用性。

顶层接口：BeanFactory，使用了简单工厂模式. 通常都是根据一个名字生产一个实例, 根据传入的唯一的标志来获得bean对象, 但具体是穿入参数后创建, 还是穿入参数前创建, 这个要根据 具体情况而定, 根据名字或类型生产不同的bean. 一句话总结: **BeanFactory的责任就是生产Bean**  IOC

**29.JVM分区，对象如何到老年代**

主要有下面三种方式：大对象，长期存活的对象，动态对象年龄判定

1：大对象直接进入老年代。比如很长的字符串，或者很大的数组等，参数-  
XX:PretenureSizeThreshold=3145728设置，超过这个参数设置的值就直接进入老年代

2：长期存活的对象进入老年代。在堆中分配内存的对象，其内存布局的对象头中（Header）包含了GC分代年龄标记信息。如果对象在eden区出生，那么它的GC分代年龄会初始值为1，每熬过一次Minor GC而不被回收，这个值就会增加1岁。当它的年龄到达一定的数值时，就会晋升到老年代中，可以通过参数-  
XX:MaxTenuringThreshold设置年龄阈值（默认是15岁）

3：当Survivor空间中相同年龄所有对象的大小总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，而不需要达到默认的分代年龄

### 30.BIO、NIO、AIO：

- Java BIO：同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。
- Java NIO：同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。
- Java AIO(NIO.2)：异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。

### 31.BIO、NIO、AIO适用场景分析:

- BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。
- NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。
- AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持

### 32.自旋锁

自旋锁（spinlock）：是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。获取锁的线程一直处于活跃状态，但是并没有执行任何有效的任务，使用这种锁会造成busy-waiting。而轻量级锁是在加锁的时候，如何使用一种更高效的方式来加锁。先解释自旋锁：未使用自旋锁的线程状态：运行-》阻塞-》运行被挂起 被唤醒使用自旋锁线程状态：运行-》运行-》运行自旋等待锁被释放

### 33.线程阻塞几种情况？如何自己实现阻塞队列？

阻塞的几种情况：

- 1.当线程执行Thread.sleep（）时，它一直阻塞到指定的毫秒时间之后，或者阻塞被另一个线程打断；
- 2.当线程碰到一条wait（）语句时，它会一直阻塞到接到通知（notify（））、被中断或经过了指定毫秒时间为止（若制定了超时值的话）
- 3.线程阻塞与不同I/O的方式有多种。常见的一种方式是通过InputStream的read（）方法，该方法一直阻塞到从流中读取一个字节的的数据为止，它可以无限阻塞，因此不能指定超时时间；
- 4.线程也可以阻塞等待获取某个对象锁的排他性访问权限（即等待获得synchronized语句必须的锁时阻塞）。

如何实现阻塞队列：基于一个Object[]数组存放数据，分别定义takeIndex与putIndex,当下标到达末尾时会被设置为0，从数组的第一个下标位置重新开始向后增长，形成一个不断循环的过程。一个判断count是否达到了items.length（队列大小）的if语句，如果count不等于items.length，那么就表示队列还没有满，随后就直接调用了enqueue方法对元素进行了入队。但是如果这时候队列已满，那么count的值就会等于items.length，这将会导致我们调用Thread.sleep(200L)使当前线程休眠200毫秒。当线程从休眠中恢复时，又会进入下一次循环，重新判断条件count != items.length。也就是说，如果队列没有弹出元素使我们可以完成插入操作，那么线程就会一直处于“判断 -> 休眠”的循环而无法从put()方法中返回，也就是进入了“阻塞”状态。

### 34.HashMap和HashTable的区别

- 1、HashMap继承自AbstractMap类。但二者都实现了Map接口。Hashtable继承自Dictionary类，Dictionary类是一个已经被废弃的类（见其源码中的注释）。父类都被废弃，自然而然也没人用它的子类Hashtable了。
- 2、HashMap线程不安全,Hashtable线程安全
- 3.包含的contains方法不同 HashMap是没有contains方法的，而包括containsValue和containsKey方法；hashtable则保留了contains方法，效果同containsValue,还包括containsValue和containsKey方法。
- 4.是否允许null值 Hashmap是允许key和value为null值的，用containsValue和containsKey方法判断是否包含对应键值对；HashTable键值对都不能为空，否则包空指针异常。
- 5.计算hash值方式不同
- 6.扩容方式不同（容量不够）
- 7.解决hash冲突方式不同（地址冲突）

### 35.Java 的 I/O 大概可以分成以下几类：

磁盘操作：File 字节操作：InputStream 和 OutputStream 字符操作：Reader 和 Writer 对象操作：Serializable  
网络操作：Socket 新的输入/输出：NIO

### 36.线程池的拒绝策略

- CallerRunsPolicy（调用者运行策略）：如果添加到线程池失败，那么主线程自己会去执行该任务；如果执行程序已关闭（主线程运行结束），则会丢弃该任务，但是由于是调用者线程自己执行的，当多次提交任务时，就会阻塞后续任务执行，性能和效率自然就慢了。
- AbortPolicy（中止策略）：默认，队列满了丢任务抛出异常。中止策略的意思也就是打断当前执行流程
- DiscardPolicy（丢弃策略）：队列满了丢任务不抛出异常。直接静悄悄的丢弃这个任务，不触发任何动作。所以这个策略基本不用了。
- DiscardOldestPolicy（弃老策略）：将最早进入队列的任务删除（弹出队列头部的元素），之后尝试加入队列。

### 37.出现异常时，jvm怎么操作的

- 1.JVM会在当前出现异常的方法中，查找异常表，是否有合适的处理者来处理
- 2.如果当前方法异常表不为空，并且异常符合处理者的from和to节点，并且type也匹配，则JVM调用位于target的调用者来处理。
- 3.如果上一条未找到合理的处理者，则继续查找异常表中的剩余条目
- 4.如果当前方法的异常表无法处理，则向上查找（弹栈处理）刚刚调用该方法的调用处，并重复上面的操作。

5.如果所有的栈帧被弹出，仍然没有处理，则抛给当前的Thread，Thread则会终止。

6.如果当前Thread为最后一个非守护线程，且未处理异常，则会导致JVM终止运行。（finally代码块是如何去实现的？在编译阶段对finally代码块进行处理 当前版本Java编译器的做法，是复制finally代码块的内容，分别放到所有正常执行路径，以及异常执行路径的出口中。）

### 38.年轻代回收以及full GC，持久代能回收？

- 年轻代回收：大多数情况下，对象在新生代 Eden 上分配，当 Eden 空间不够时，发起 Minor GC。
- full GC回收
  1. 调用 System.gc() 只是建议虚拟机执行 Full GC，但是虚拟机不一定真正去执行。不建议使用这种方式，而是让虚拟机管理内存。
  2. 老年代空间不足 老年代空间不足的常见场景为前文所讲的大对象直接进入老年代、长期存活的对象进入老年代等。为了避免以上原因引起的 Full GC，应当尽量不要创建过大的对象以及数组。除此之外，可以通过 -Xmn 虚拟机参数 调大新生代的大小，让对象尽量在新生代被回收掉，不进入老年代。还可以通过 -XX:MaxTenuringThreshold 调大对象进入老年代的年龄，让对象在新生代多存活一段时间。
  3. 空间分配担保失败 使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果担保失败会执行一次 Full GC。。
  4. JDK 1.7 及以前的永久代空间不足 在 JDK 1.7 及以前，HotSpot 虚拟机中的方法区是用永久代实现的，永久代中存放的为一些 Class 的信息、常量、静态变量等数据。当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果经过 Full GC 仍然回收不了，那么虚拟机会抛出 java.lang.OutOfMemoryError。为避免以上原因引起的 Full GC，可采用的方法为增大永久代空间或转为使用 CMS GC。
  5. Concurrent Mode Failure 执行 CMS GC 的过程中同时对对象要放入老年代，而此时老年代空间不足（可能是 GC 过程中浮动垃圾过多导致暂时性的空间不足），便会报 Concurrent Mode Failure 错误，并触发 Full G
- 永生代也是可以回收的，条件是 1.该类的实例都被回收。 2.加载该类的classLoader已经被回收 3.该类不能通过反射访问到其方法，而且该类的java.lang.class没有被引用 当满足这3个条件时，是可以回收，但回不回收还得看jvm。（会触发full gc）

### 39.检查性异常与非检查性异常

1.非检查性异常：Error 和 RuntimeException 以及他们的子类。Java语言在编译时，不会提示和发现这样的异常，不要求在程序中处理这些异常。所以我们可以 在程序中编写代码来处理（使用try...catch...finally）这样的异常，也可以不做任何处理。对于这些错误或异常，我们应该修正代码，而不是去通过异常处理器处理。这样的异常发生的原因多半是由于我们的代码逻辑出现了问题。

2.Java语言强制要求程序员为这样的异常做预备处理工作（使用try...catch...finally或者throws）。在方法中要么用try-catch语句捕获它并处理，要么用throws子句声明抛出它，否则编译不会通过。这样的异常一般是由程序的运行环境导致的。因为程序可能被运行在各种未知的环境下，而程序员无法干预用户如何使用他编写的程序，于是程序员就应该为这样的异常时刻准备着。如SQLException，IOException，ClassNotFoundException 等。

## 第四章 网络


### 1.tcp的概念？UDP的概念

- tcp属于传输层，是一种面向连接的传输协议，提供可靠的交付，能够进行流量控制，拥塞控制，提供全双工通信，面向字节流，每一条tcp连接只能是点对点。
- udp用户数据报协议 UDP (User Datagram Protocol) 是无连接的，尽最大可能交付，没有拥塞控制，面向报文 (对于应用程序传下来的报文不合并也不拆分，只是添加 UDP 首部)，支持一对一、一对多、多对一和多对多的交互通信。

## 2.三次握手 (三次握手改成二次或者四次会怎么样)

- 当A向B发送请求，此时会将同步位SYN为1，并选择序号seq=x,代表传输的数据第一个数据字节的序号为x；当B接受到报文请求后，会将SYN=1、确认值 ACK=1置为1，确认号ack=x+1,选择序号seq=y;A收到此报文后会向B发送请求，此时ACK=1,ack=y+1;seq=x+1,当确认结束后将SYN=0;
- 1.两次握手无法判断当前连接是否是历史连接 (序列号过期或者超时)。如果是历史连接 (序列号过期或超时)，则第三次握手发送的报文是 RST 报文，以此中止历史连接；如果不是历史连接，则第三次发送的报文是 ACK 报文，通信双方就会成功建立连接；2.(无法同步双方初始序列号) 只保证了一方的初始序列号能被对方成功接收，没办法保证双方的初始序列号都能被确认接收。3.由于没有第三次握手，服务器不清楚客户端是否收到了自己发送的建立连接的 ACK 确认信号，所以每收到一个 SYN 就只能先主动建立一个连接，即两次握手会造成消息滞留情况下，服务器重复接受无用的连接请求 SYN 报文，而造成重复分配资源。
- 四次握手其实也能够可靠的同步双方的初始化序号，但由于第二步和第三步可以优化成一步，所以就成了「三次握手」。

## 3.四次挥手

四次挥手 A向B发出请求，此时会将FIN=1,seq=u;此时B收到请求，将会向A发送报文，其中ACK=1,ack=U+1, seq=v;这时TCP服务器进程通知高层应用进程，从A到B这个放向的连接就释放了，Tcp的连接处于半关闭状态。B若发送数据，A仍要接收。若B已经没有向A发送的数据，其应用进程就通知TCP释放连接。此时B向A发送释放连接的请求，ACK=1,FIN=1,seq=w,ack=u+1;当A收到这段报文后，必须发出确认。ACK=1,ack=w+1;seq=u+1;同时要注意此时A要等待2MSL (60s) 的时间，确保A发送的最后一个请求能够到达A端，还能防止“已失效的连接请求报文段”出现在下一次请求中。

## 4.Web的网页请求流程

- 通过DHCP动态主机配置协议，层层访问默认网关路由器，为主机申请IP。
- 基于ARP地址转换协议，查找默认网关路由器的MAC地址
- 基于DNS域名系统，去查找目的域名的IP，首先：1.先从浏览器缓存里找IP,因为浏览器会缓存DNS记录一段时间。2.如没找到,再从Hosts文件查找是否有该域名和对应IP。3.如没找到,再从路由器缓存找4.如没找到,再从DNS缓存查找5.如果都没找到,浏览器域名服务器向根域名服务器(http://baidu.com)查找域名对应IP,还没找到就把请求转发到下一级,直到找到IP
- 建立TCP三次握手的请求连接，将请求报文放入套接字，网络层、数据链路层，层层封装，最终经过以太网路由转发给目的服务器
- 服务器收到请求后，返回响应报文，本地主机接收到响应报文之后，经由浏览器渲染展示。

## 5.ARP协议的作用以及使用方法

- ARP协议是根据IP地址获取物理地址的一个TCP/IP协议。主机将包含目标主机ip地址的ARP请求发送到网络上的所有主机，接受返回的消息，用于确定目标的物理地址；收到返回消息后将ARP与MAC地址映射到本机的ARP缓存中，以便下次访问节省时间。

6.TCP协议的滑动窗口

窗口是缓存的一部分，用来暂时存放字节流。发送方和接收方各有一个窗口，接收方通过TCP报文字段中的窗口字段告诉发送方自己窗口大小，而发送方根据这个值和其他数值对自己的窗口大小进行设置。当发送方的一部分数据被发送时，他的滑动窗口向右移动一部分，接受窗口也是同样的道理，当异步数据接收被确认时，窗口向右滑动，同时要注意接受窗口内只会对最后一个按序到达的字节进行确认，比如{31, 34, 35}，接收方只对31进行确认，发送方得到一个字节后，可以指导这个字节之前的数据已经全部被接收。

7.Select和poll以及epoll使用场景,以及epoll中的ET和LT

	select	poll	epoll
FD数量	1024	无限制	无限制
FD状态感知	轮询	轮询	事件通知
重置数据源	需要	不需要 ( event/revent)	通知就绪的
运行模式	条件触发 ( LT)	条件触发 ( LT)	边缘触发 ( ET) /条件触发(LT)

poll:Struct pollfd select:bitMap epoll:

- `epoll_create()`:内核态创建epoll实例 ( 红黑树与就绪队列rdlist)
- `epoll_ctl`:对红黑树操作，添加所有socket节点
- `epoll_wait`:1.阻塞线程2.内核查找红黑树ready的socket，放入就绪列表3.将就绪列表内容复制到events。
- `events[i].data.fd`:处理socket 条件触发：当读取一半数据时，干别的事情，条件触发在下次会重新读取，但是边缘处触发不会，只会读取一次。条件触发：不会触发遗漏事件，系统资源占用大 边缘触发：速度快，但会出现遗漏事件；nginx:ET;redis:LT 简单来讲，LT是epoll的默认操作模式，当`epoll_wait`函数检测到有事件发生并将通知应用程序，而应用程序不一定必须立即进行处理，这样`epoll_wait`函数再次检测到此事件的时候还会通知应用程序，直到事件被处理。

而ET模式，只要`epoll_wait`函数检测到事件发生，通知应用程序立即进行处理，后续的`epoll_wait`函数将不再检测此事件。因此ET模式在很大程度上降低了同一个事件被epoll触发的次数，因此效率比LT模式高。

解释为什么epoll默认是LT的原因

LT(level triggered)：LT是缺省的工作方式，并且同时支持block和no-block socket。在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表。

8.DNS服务器原理

DNS 是一个分布式数据库，提供了主机名和 IP 地址之间相互转换的服务。这里的分布式数据库是指，每个站点只保留它自己的那部分数据。域名具有层次结构，从上到下依次为：根域名、顶级域名、二级域名。DNS 可以使用 UDP 或者 TCP 进行传输，使用的端口号都为 53。大多数情况下 DNS 使用UDP进行传输，这就要求域名解析器和域名服务器都必须自己处理超时和重传从而保证可靠性。在两种情况下会使用 TCP 进行传输：

- 如果返回的响应超过的 512 字节 ( UDP 最大只支持 512 字节的数据 )。
- 区域传送 ( 区域传送是主域名服务器向辅助域名服务器传送变化的那部分数据 )。

9.如何保证数据的实时性



- 降低排队时延：分组在路由器的输入队列和输出队列中排队等待的时间，取决于网络当前的通信量。
- 处理时延：主机或路由器收到分组时进行处理所需要的时间，例如分析首部、从分组中提取数据、进行差错检验或查找适当的路由等。
- 传输时延：主机或路由器传输数据帧所需要的时间。
- 传播时延：电磁波在信道中传播所需要花费的时间，电磁波传播的速度接近光速

## 10.http协议的错误码

400 Bad Request：请求语法出错 401 Unauthorized:认证未通过 403 Forbidden:请求被拒绝 404 Not Found 500 Internal Server Error:服务器正在执行请求时发生错误。 503 Service Unavailable：服务器暂时处于超负载或正在进行停机维护，现在无法处理请求。

## 11.对https的理解

HTTPS 在 HTTP 与 TCP 层之间加入了 TLS 协议，来解决上述的风险

TLS 协议是解决 HTTP 的风险的方式：

- 信息加密：HTTP 交互信息是被加密的，第三方就无法被窃取；
- 校验机制：校验信息传输过程中是否有被第三方篡改过，如果被篡改过，则会有警告提示；
- 身份证书：证明淘宝是真的淘宝网；

## 12.TCP、IP、HTTP各自在网络的那一层

- TCP属于传输层，IP属于网络层，HTTP属于应用层

## 13.介绍http和https

- HTTP 是超文本传输协议，信息是明文传输，存在安全风险的问题。HTTPS 则解决 HTTP 不安全的缺陷，在 TCP 和 HTTP 网络层之间加入了 SSL/TLS 安全协议，使得报文能够加密传输。
- HTTP 连接建立相对简单，TCP 三次握手之后便可进行 HTTP 的报文传输。而 HTTPS 在 TCP 三次握手之后，还需进行 SSL/TLS 的握手过程，才可进入加密报文传输。
- HTTP 的端口号是 80，HTTPS 的端口号是 443。
- HTTPS 协议需要向 CA (证书权威机构) 申请数字证书，来保证服务器的身份是可信的。

## 14.介绍http各个版本(1.0、1.1、1.x、2.0、3.0)说清楚每个版本在之前版本上的主要功能升级；http3.0具体解决了什么问题？使用udp的可靠性如何保障(QUIC?)

1.HTTP/1.1 相比 HTTP/1.0 性能上的改进：

- 使用 TCP 长连接的方式改善了 HTTP/1.0 短连接造成的性能开销。
- 支持 管道 ( pipeline ) 网络传输，只要第一个请求发出去了，不必等其回来，就可以发第二个请求出去，可以减少整体的响应时间。

2.那上面的 HTTP/1.1 的性能瓶颈，HTTP/2 做了什么优化？

- 头部压缩：压缩头 ( Header ) 如果你同时发出多个请求，他们的头是一样的或是相似的，那么，协议会帮你消除重复的分。



- 二进制格式：HTTP/2 不再像 HTTP/1.1 里的纯文本形式的报文，而是全面采用了二进制格式。
- 数据流：的数据包不是按顺序发送的，同一个连接里面连续的数据包，可能属于不同的回应。因此，必须对数据包做标记，指出它属于哪个回应。每个请求或回应的所有数据包，称为一个数据流（Stream）
- 多路复用：HTTP/2 是可以在一个连接中并发多个请求或回应，而不用按照顺序一一对应。移除了 HTTP/1.1 中的串行请求，不需要排队等待，也就不会再出现「队头阻塞」问题，降低了延迟，大幅度提高了连接的利用率
- 服务器推送：还在一定程度上改善了传统的「请求 - 应答」工作模式，服务不再是被动地响应，也可以主动向客户端发送消息。举例来说，在浏览器刚请求 HTML 的时候，就提前把可能会用到的 JS、CSS 文件等静态资源主动发给客户端，减少延时的等待，也就是服务器推送（Server Push，也叫 Cache Push）。

3.HTTP/2 有哪些缺陷？HTTP/3 做了哪些优化？HTTP/2 主要的问题在于：多个 HTTP 请求在复用同一个 TCP 连接，下层的 TCP 协议是不知道有多少个 HTTP 请求的。

所以一旦发生了丢包现象，就会触发 TCP 的重传机制，这样在一个 TCP 连接中的所有的 HTTP 请求都必须等待这个丢了的包被重传回来。

- HTTP/1.1 中的管道（pipeline）传输中如果有一个请求阻塞了，那么队列后请求也统统被阻塞住了
- HTTP/2 多请求复用同一个TCP连接，一旦发生丢包，就会阻塞住所有的 HTTP 请求

**HTTP/3 把 HTTP 下层的 TCP 协议改成了 UDP！**

大家都知道 UDP 是不可靠传输的，基于 UDP 的 QUIC 协议 可以实现类似 TCP 的可靠性传输。QUIC 是一个在 UDP 之上的伪 TCP + TLS + HTTP/2 的多路复用的协议。

## 15.Get和post的区别

- get:方法的含义是请求从服务器获取资源，这个资源可以是静态的文本、页面、图片视频等。
- post:它向 URI 指定的资源提交数据，数据就放在报文的 body 里。

## 16.GET 和 POST 方法都是安全和幂等的吗？

先说明下安全和幂等的概念：

- 在 HTTP 协议里，所谓的「安全」是指请求方法不会「破坏」服务器上的资源。
- 所谓的「幂等」，意思是多次执行相同的操作，结果都是「相同」的。

那么很明显 GET 方法就是安全且幂等的，因为它是「只读」操作，无论操作多少次，服务器上的数据都是安全的，且每次的结果都是相同的。

POST 因为是「新增或提交数据」的操作，会修改服务器上的资源，所以是不安全的，且多次提交数据就会创建多个资源，所以不是幂等的。

## 17.对称加密算法：

- AES：密钥的长度可以为128、192和256位，也就是16个字节、24个字节和32个字节
- CHACHA20：

- DES：密钥的长度64位，8个字节

## 18.HTTP协议的消息是什么样的？

一个HTTP请求报文由四个部分组成：请求行、请求头部、空行、请求数据。

## 19.tcp这么保证数据的可靠性

- 校验和
- 序列号
- 确认应答
- 超时重传
- 连接管理
- 流量控制
- 拥塞控制

## 20.HTTPS怎么保证可靠的？

SSL/TLS

- SSL：（Secure Socket Layer，安全套接字层），位于可靠的面向连接的网络层协议和应用层协议之间的一种协议层。SSL通过互相认证、使用数字签名确保完整性、使用加密确保私密性，以实现客户端和服务端之间的安全通讯。该协议由两层组成：SSL记录协议和SSL握手协议。
- TLS：(Transport Layer Security，传输层安全协议)，用于两个应用程序之间提供保密性和数据完整性。该协议由两层组成：TLS 记录协议和 TLS 握手协议。

## 21.如果请求过大，如何优化

1.对请求数据进行压缩：Accept-Encoding: gzip,compress 2.缓存

## 22.三次握手的阶段对应Java网络编程的哪个流程？服务器端的Accpet对应哪个流程？



1. 服务器 listen 时，计算了全/半连接队列的长度，还申请了相关内存并初始化。
2. 客户端 connect 时，把本地 socket 状态设置成了 TCP\_SYN\_SENT，选则一个可用的端口，发出 SYN 握手请求并启动重传定时器。
3. 服务器响应 ack 时，会判断下接收队列是否满了，满的话可能会丢弃该请求。否则发出 synack，申请 request\_sock 添加到半连接队列中，同时启动定时器。
4. 客户端响应 synack 时，清除了 connect 时设置的重传定时器，把当前 socket 状态设置为 ESTABLISHED，开启保活计时器后发出第三次握手的 ack 确认。
5. 服务器响应 ack 时，把对应半连接对象删除，创建了新的 sock 后加入到全连接队列中，最后将新连接状态设置为 ESTABLISHED。
6. accept 从已经建立好的全连接队列中取出一个返回给用户进程

## 23.keep-alive是怎么实现的

- 基于TCP的保活机制

- 如果两端的 TCP 连接一直没有数据交互，达到了触发 TCP 保活机制的条件，那么内核里的 TCP 协议栈就会发送探测报文。1.如果对端程序是正常工作的。当 TCP 保活的探测报文发送给对端, 对端会正常响应，这样 TCP 保活时间会被重置，等待下一个 TCP 保活时间的到来。2.如果对端主机崩溃，或对端由于其他原因导致报文不可达。当 TCP 保活的探测报文发送给对端后，石沉大海，没有响应，连续几次，达到保活探测次数后，TCP 会报告该 TCP 连接已经死亡。所以，TCP 保活机制可以在双方没有数据交互的情况，通过探测报文，来确定对方的 TCP 连接是否存活，这个工作是在内核完成的。

## 24.http2.0与http1.1的区别

- 头部压缩：压缩头 ( Header ) 如果你同时发出多个请求，他们的头是一样的或是相似的，那么，协议会帮你消除重复的分。
- 二进制格式：HTTP/2 不再像 HTTP/1.1 里的纯文本形式的报文，而是全面采用了二进制格式。
- 数据流：的数据包不是按顺序发送的，同一个连接里面连续的数据包，可能属于不同的回应。因此，必须要对数据包做标记，指出它属于哪个回应。每个请求或回应的所有数据包，称为一个数据流 (Stream)
- 多路复用：HTTP/2 是可以在一个连接中并发多个请求或回应，而不用按照顺序一一对应。移除了 HTTP/1.1 中的串行请求，不需要排队等待，也就不会再出现「队头阻塞」问题，降低了延迟，大幅度提高了连接的利用率
- 服务器推送：还在一定程度上改善了传统的「请求 - 应答」工作模式，服务不再是被动地响应，也可以主动向客户端发送消息。举例来说，在浏览器刚请求 HTML 的时候，就提前把可能会用到的 JS、CSS 文件等静态资源主动发给客户端，减少延时的等待，也就是服务器推送 ( Server Push，也叫 Cache Push) 。

## 25.cookie与session的区别，然后它们之间怎么通讯

- Cookie 只能存储 ASCII 码字符串，而 Session 则可以存储任何类型的数据，因此在考虑数据复杂性时首选 Session；
- Cookie 存储在浏览器中，容易被恶意查看。如果非要将一些隐私数据存在 Cookie 中，可以将 Cookie 值进行加密，然后在服务器进行解密；
- 对于大型网站，如果用户所有的信息都存储在 Session 中，那么开销是非常大的，因此不建议将所有的用户信息都存储到 Session 中
- 容量和个数限制：cookie 有容量限制，每个站点下的 cookie 也有个数限制。
- 存储的多样性：session 可以存储在 Redis 中、数据库中、应用程序中；而 cookie 只能存储在浏览器中。
- 存储位置不同：session 存储在服务器端；cookie 存储在浏览器端。
- 安全性不同：cookie 安全性一般，在浏览器存储，可以被伪造和修改。

通讯：1.服务器返回的响应报文的 Set-Cookie 首部字段包含了这个 Session ID，客户端收到响应报文之后将该 Cookie值存入浏览器中；2.客户端之后对同一个服务器进行请求时会包含该 Cookie 值，服务器收到之后提取出 Session ID，从 Redis 中取出用户信息，继续之前的业务操作。

## 26.一个tcp报文最大是多少？udp呢？

- UDP 包的大小应该是  $1500 - \text{IP头}(20) - \text{UDP头}(8) = 1472(\text{Bytes})$
- TCP 包的大小应该是  $1500 - \text{IP头}(20) - \text{TCP头}(20) = 1460(\text{Bytes})$

## 27.拥塞控制

- 慢开始与拥塞避免 发送的最初执行慢开始，令  $cwnd = 1$ ，发送方只能发送 1 个报文段；当收到确认后，将  $cwnd$  加倍，因此之后发送方能够发送的报文段数量为：2、4、8 ... 注意到慢开始每个轮次都将  $cwnd$  加倍，这样会让  $cwnd$  增长速度非常快，从而使得发送方发送的速度增长速度过快，网络拥塞的可能性也就更高。设置一个慢开始门限  $ssthresh$ ，当  $cwnd \geq ssthresh$  时，进入拥塞避免，每个轮次只将  $cwnd$  加 1。如果出现了超时，则令  $ssthresh = cwnd / 2$ ，然后重新执行慢开始。
- 快重传与快恢复 在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。例如已经接收到 M1 和 M2，此时收到 M4，应当发送对 M2 的确认。在发送方，如果收到三个重复确认，那么可以知道下一个报文段丢失，此时执行快重传，立即重传下一个报文段。例如收到三个 M2，则 M3 丢失，立即重传 M3。在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此执行快恢复，令  $ssthresh = cwnd / 2$ ， $cwnd = ssthresh$ ，注意到此时直接进入拥塞避免。慢开始和快恢复的快慢指的是  $cwnd$  的设定值，而不是  $cwnd$  的增长速率。慢开始  $cwnd$  设定为 1，而快恢复  $cwnd$  设定为  $ssthresh$ 。

## 28.time\_wait等待时间内核参数怎么修改

有个 `sysctl` 参数貌似可以使用，它是 `/proc/sys/net/ipv4/tcp_fin_timeout`，缺省值是 60，也就是 60 秒，很多网上的资料都说将这个数值设置低一些就可以减少 `netstat` 里面的 `TIME_WAIT` 状态，但是这个说法不是很准确的。经过认真阅读 Linux 的内核源代码，我们发现这个数值其实是输出用的，修改之后并没有真正的读回内核中进行使用，而内核中真正管用的是一个宏定义，在 `$KERNEL/include/net/tcp.h` 里面，有下面的行：

```
#define TCP_TIMEWAIT_LEN (60HZ) / how long to wait to destroy TIME-WAIT * state, about 60 seconds / 而
```

这个宏是真正控制 `TCP TIME_WAIT` 状态的超时时间的。如果我们希望减少 `TIME_WAIT` 状态的数目(从而节省一点点内核操作时间)，那么可以把这个数值设置低一些，根据我们的测试，设置为 10 秒比较合适，也就是把上面的修改为：

```
# define TCP_TIMEWAIT_LEN (10HZ) /* how long to wait to destroy TIME-WAIT
```

## 29.tcp打开连接的上限

- 用户级别：用户最大打开的文件限制 `ulimit -n65535`
- Linux 系统级别 `sysctl -a | grep file-max fs.file-max = 65535` 这表明这台 Linux 系统最多允许同时打开(即包含所有用户打开文件数总和)65535 个文件，是 Linux 系统级硬限制，所有用户级的打开文件数限制都不会超过这个数值。通常这个系统级硬限制是 Linux 系统在启动时根据系统硬件资源状况计算出来的最佳的最大的同时打开文件数限制
- 网络端口限制 `sysctl -a | grep ipv4.ip_conntrack_max net.ipv4.ip_conntrack_max = 20000` 这表明系统将对最大跟踪的 TCP 连接数限制默认为 20000。

## 30.HTTP状态码 301和302什么区别

301 和 302 状态码都表示重定向，就是说浏览器在拿到服务器返回的这个状态码后会自动跳转到一个新的 URL 地址，这个地址可以从响应的 `Location` 首部中获取（用户看到的效果就是他输入的地址 A 瞬间变成了另一个地址 B）——这是它们的共同点。他们的不同在于。

- 301 表示旧地址 A 的资源已经被永久地移除了（这个资源不可访问了），搜索引擎在抓取新内容的同时也将旧的网址交换为重定向之后的网址；
- 302 表示旧地址 A 的资源还在（仍然可以访问），这个重定向只是临时地从旧地址 A 跳转到地址 B，搜索引擎会抓取新的内容而保存旧的网址。

## 31.Https的优化手段

https 产生性能消耗主要包括两个环节：

- 第一环节，TLS协议握手过程；
- 第二环节，握手后的对称加密报文传输；

为此有以下手段进行优化：

- 硬件优化：
- 软件优化：
- 协议优化：1.使用ECDHE密钥交换算法替换RSA算法，其支持[False start] (抢跑的意思)，能够将TLS握手的消息往返有2RTT减少到1RTT。2.直接把 TLS 1.2 升级成 TLS 1.3，TLS 1.3 大幅度简化了握手的步骤，完成 TLS 握手只要 1 RTT，而且安全性更高。
- 证书优化：优化方向为1.一个是证书传输：要让证书更便于传输，那必然是减少证书的大小，这样可以节约带宽，也能减少客户端的运算量。所以，对于服务器的证书应该选择 椭圆曲线 ( ECDSA ) 证书，而不是 RSA 证书，因为在相同安全强度下，ECC 密钥长度比 RSA 短的多。2.一个是证书验证；服务器应该开启 OCSP Stapling 功能，由服务器预先获得 OCSP 的响应，并把响应结果缓存起来，这样 TLS 握手的时候就不用再访问 CA 服务器，减少了网络通信的开销，提高了证书验证的效率；
- 会话复用：我们可以使用一些技术让客户端和服务端使用上一次 HTTPS 连接使用的会话密钥，直接恢复会话，而不用再重新走完整的 TLS 握手过程。常见的会话重用技术有 Session ID 和 Session Ticket，用了会话重用技术，当再次重连 HTTPS 时，只需要 1 RTT 就可以恢复会话。对于 TLS1.3 使用 Pre-shared Key 会话重用技术，只需要 0 RTT 就可以恢复会话。

### 32.A B C类网络地址范围

- A:1-126
- B:128.1-191.255
- C:192.0.1-223.255.255

### 33.路由器转发的过程中怎么指定端口 以及转发ip数据报的过程？

IP报文中的目的IP地址往往是主机地址，而路由表中的目的地址往往为网络地址，怎么让二者匹配呢？这里面有个底层的操作：首先将IP报文中的目的地址和路由表项中的子网掩码进行"逻辑与"操作，得到一个网络地址，然后拿此网络地址与路由项中的网络地址做比较，如果一致就认为匹配，否则认为不匹配。

如果路由项匹配，则路由器查看所匹配的路由项的下一跳地址是否在直连的链路上。如果在直连的链路上，则根据此下一跳转发；如果不在直连的链路上，则需要在路由表中再次查找此下一跳地址所匹配的路由项。

确定了下一跳地址后，路由器将此报文送往对应的接口，接口进行相应的地址解析，解析出对应的链路层地址后，对IP报文进行数据封装并转发。

### 34.https为什么要先非对称再对称

首先：非对称加密的加解密效率是非常低的，而 http 的应用场景中通常端与端之间存在大量的交互，非对称加密的效率是无法接受的。

另外：在 HTTPS 的场景中只有服务端保存了私钥，一对公私钥只能实现单向的加解密，所以HTTPS 中内容传输加密采取的是对称加密，而不是非对称加密。

### 35.怎么抵挡syn攻击

- syn攻击：我们都知道 TCP 连接建立是需要三次握手，假设攻击者短时间伪造不同 IP 地址的 SYN 报文，服务端每接收到一个 SYN 报文，就进入SYN\_RCVD 状态，但服务端发送出去的 ACK + SYN 报文，无法得到未知 IP 主机的 ACK 应答，久而久之就会占满服务端的 SYN 接收队列（未连接队列），使得服务器不能为正常用户服务。
- 通过修改 Linux 内核参数，控制队列大小和当队列满时应做什么处理。

1.当网卡接收数据包的速度大于内核处理的速度时，会有一个队列保存这些数据包。控制该队列的最大值如下参数：

```
net.core.netdev_max_backlog
```

2.SYN\_RCVD 状态连接的最大个数：

```
net.ipv4.tcp_max_syn_backlog
```

3.超出处理能时，对新的 SYN 直接回 RST，丢弃连接：

```
net.ipv4.tcp_abort_on_overflow
```

- 1.当「SYN 队列」满之后，后续服务器收到 SYN 包，不进入「SYN 队列」；2.计算出一个 cookie 值，再以 SYN + ACK 中的「序列号」返回客户端，3.服务端接收到客户端的应答报文时，服务器会检查这个 ACK 包的合法性。如果合法，直接放入到「Accept 队列」。4.最后应用通过调用 `accept()` socket 接口，从「Accept 队列」取出的连接。

### 36.什么是XSS攻击，如何避免。

XSS 攻击：即跨站脚本攻击，它是 Web 程序中常见的漏洞。原理是攻击者往 Web 页面里插入恶意的脚本代码（css 代码、Javascript 代码等），当用户浏览该页面时，嵌入其中的脚本代码会被执行，从而达到恶意攻击用户的目的，如盗取用户 cookie、破坏页面结构、重定向到其他网站等。

预防 XSS 的核心是必须对输入的数据做过滤处理。

### 37.什么是 CSRF 攻击，如何避免？

CSRF：Cross-Site Request Forgery（中文：跨站请求伪造），可以理解为攻击者盗用了你的身份，以你的名义发送恶意请求，比如：以你名义发送邮件、发消息、购买商品，虚拟货币转账等。防御手段：

- 验证请求来源地址；
- 关键操作添加验证码；
- 在请求地址添加 token 并验证。

### 38.TCP头的参数

- 源端口号

- 目标端口号
- 序列号
- 确认应答号
- 窗口大小
- 校验和
- 紧急指针

### 39.请求头的参数

- Content-Length 字段 服务器在返回数据时，会有 Content-Length 字段，表明本次回应的数据长度。
- Connection 字段 Connection 字段最常用于客户端要求服务器使用 TCP 持久连接，以便其他请求复用
- Content-Type 字段 Content-Type 字段用于服务器回应时，告诉客户端，本次数据是什么格式。
- Content-Encoding 字段 Content-Encoding 字段说明数据的压缩方法。表示服务器返回的数据使用了什么压缩格式

### 40.tcp与udp使用的场景

TCP适合对传输效率要求低，但准确率要求高的应用场景，比如万维网(HTTP)、文件传输(FTP)、电子邮件(SMTP)等。

UDP是无连接的，不可靠传输，尽最大努力交付数据，协议简单、资源要求少、传输速度快、实时性高的特点，适用于对传输效率要求高，但准确率要求低的应用场景，比如域名转换(DNS)、远程文件服务器(NFS)等。

### 41.https加密过程

SSL/TLS 协议建立的详细流程：

#### 1. ClientHello

首先，由客户端向服务器发起加密通信请求，也就是 ClientHello 请求。

在这一步，客户端主要向服务器发送以下信息：

- (1) 客户端支持的 SSL/TLS 协议版本，如 TLS 1.2 版本。
- (2) 客户端生产的随机数 ( Client Random )，后面用于生产「会话密钥」。
- (3) 客户端支持的密码套件列表，如 RSA 加密算法。

#### 2. SeverHello

服务器收到客户端请求后，向客户端发出响应，也就是 SeverHello。服务器回应的内容有如下内容：

- (1) 确认 SSL/ TLS 协议版本，如果浏览器不支持，则关闭加密通信。
- (2) 服务器生产的随机数 ( Server Random )，后面用于生产「会话密钥」。
- (3) 确认的密码套件列表，如 RSA 加密算法。
- (4) 服务器的数字证书。

#### 3.客户端回应

客户端收到服务器的回应之后，首先通过浏览器或者操作系统中的 CA 公钥，确认服务器的数字证书的真实性。

如果证书没有问题，客户端会从数字证书中取出服务器的公钥，然后使用它加密报文，向服务器发送如下信息：

- (1) 一个随机数 ( pre-master key )。该随机数会被服务器公钥加密。
- (2) 加密通信算法改变通知，表示随后的信息都将用「会话密钥」加密通信。
- (3) 客户端握手结束通知，表示客户端的握手阶段已经结束。这一项同时把之前所有内容的发生的数据做个摘要，用来供服务端校验。

上面第一项的随机数是整个握手阶段的第三个随机数，这样服务器和客户端就同时有三个随机数，接着就用双方协商的加密算法，各自生成本次通信的「会话密钥」。

#### 4. 服务器的最后回应

服务器收到客户端的第三个随机数 ( pre-master key ) 之后，通过协商的加密算法，计算出本次通信的「会话密钥」。然后，向客户端发送最后的信息：

- (1) 加密通信算法改变通知，表示随后的信息都将用「会话密钥」加密通信。
- (2) 服务器握手结束通知，表示服务器的握手阶段已经结束。这一项同时把之前所有内容的发生的数据做个摘要，用来供客户端校验。

至此，整个 SSL/TLS 的握手阶段全部结束。接下来，客户端与服务器进入加密通信，就完全是使用普通的 HTTP 协议，只不过用「会话密钥」加密内容。

## 第五章 操作系统

### 1.操作系统的内存管理

包括：虚拟内存、内存分配、地址映射、内存保护与共享；操作系统为了管理内存，其需要将内存抽象为地址空间。每个程序拥有自己的地址空间，地址空间又被分为多个页，这些页被映射到物理内存，但不需要将全部的存储到物理内存中，当程序引用到的内存不在物理内存中存在时，由硬件执行必要的映射，将缺失的部分装入到物理内存中，进行重新执行。从而使在有限的内存中运行大程序成为了可能性。（虚拟内存：用于让物理内存扩充成更大的逻辑内存，从而让程序获得更多可用的内存。）

### 2.操作系统的文件管理系统，Linux文件系统的数据结构是如何组织的？

文件系统是操作系统中负责管理持久数据的子系统，说简单点，就是负责把用户的文件存到磁盘硬件中，因为即使计算机断电了，磁盘里的数据并不会丢失，所以可以持久化的保存文件。文件系统的基本单位是文件，它的目的是对磁盘上的文件进行组织管理，那组织的方式不同，就会形成不同的文件系统。对于Linux系统会为文件分配两个数据结构：索引节点和目录项。


- 索引节点，也就是 **inode**，用来记录文件的元信息，比如 **inode** 编号、文件大小、访问权限、创建时间、修改时间、数据在磁盘的位置等等。索引节点是文件的唯一标识，它们之间一一对应，也同样都会被存储在硬盘中，所以索引节点同样占用磁盘空间。
- 目录项，也就是 **dentry**，用来记录文件的名称、索引节点指针以及与其他目录项的层级关联关系。多个目录项关联起来，就会形成目录结构，但它与索引节点不同的是，目录项是由内核维护的一个数据结



构，不存放于磁盘，而是缓存在内存。

由于索引节点唯一标识一个文件，而目录项记录着文件的名，所以目录项和索引节点的关系是多对一，也就是说，一个文件可以有多个别字。比如，硬链接的实现就是多个目录项中的索引节点指向同一个文件。

文件系统的结构：

Ext2整个文件系统的结构和块组：文件系统的结构

- 超级块，包含的是文件系统的重要信息，比如 inode 总个数、块总个数、每个块组的 inode 个数、每个块组的块个数等等。
- 块组描述符，包含文件系统中各个块组的状态，比如块组中空闲块和 inode 的数目等，每个块组都包含了文件系统中「所有块组的组描述符信息」
- 数据位图和 inode 位图，用于表示对应的数据块或 inode 是空闲的，还是被使用中。
- inode 列表，包含了块组中所有的 inode，inode 用于保存文件系统中与各个文件和目录相关的所有元数据。
- 数据块，包含文件的有用数据。

可以会发现每个块组里有很多重复的信息，比如超级块和块组描述符表，这两个都是全局信息，而且非常的重要，这么做是有两个原因：

如果系统崩溃破坏了超级块或块组描述符，有关文件系统结构和内容的所有信息都会丢失。如果有冗余的副本，该信息是可能恢复的。

通过使文件和管理数据尽可能接近，减少了磁头寻道和旋转，这可以提高文件系统的性能。

不过，Ext2 的后续版本采用了稀疏技术。该做法是，超级块和块组描述符表不再存储到文件系统的每个块组中，而是只写入到块组 0、块组 1 和其他 ID 可以表示为 3、5、7 的幂的块组中。

### 3.进程通讯的常用方式

- 管道
- FIFO命名管道：去除了管道只能在父子进程中使用的限制。
- 消息队列：1.消息队列可以独立于读写进程存在，从而避免了 FIFO 中同步管道的打开和关闭时可能产生的困难；2.避免了FIFO的同步阻塞问题，不需要进程自己提供同步方法；3.读进程可以根据消息类型有选择地接收消息，而不像 FIFO 那样只能默认地接收。
- 信号量
- 共享存储
- 套接字：与其它通信机制不同的是，它可用于不同机器间的进程通信。

### 4.操作系统中常用的调度算法

1. 批处理系统 批处理系统没有太多的用户操作，在该系统中，调度算法目标是保证吞吐量和周转时间（从提交到终止的时间）。
- 针对批处理系统：先来先服务、短作业优先、最短剩余时间优先
2. 交互式系统 交互式系统有大量的用户交互操作，在该系统中调度算法的目标是快速地进行响应。
- 针对交互式系统：时间片轮转、优先级调度、多级反馈队列

3. 实时系统 实时系统要求一个请求在一个确定时间内得到响应。分为硬实时和软实时，前者必须满足绝对的截止时间，后者可以容忍一定的超时。

## 5.如何处理死锁

- 鸵鸟策略：操作系统当作没看到
- 死锁检测与死锁恢复：1.死锁检测：每种类型一个资源的死锁检测；每种类型多个资源的死锁检测 2.死锁恢复：利用抢占恢复；利用回滚恢复；通过杀死进程恢复
- 死锁预防 1.破坏互斥 2.破坏占有和等待 3.破坏不可抢占 4.破坏循环等待
- 死锁避免 1.安全状态 2.单个资源的银行家算法 3.多个资源的银行家算法

## 6.进程、线程

- 进程是资源分配的基本单位。进程控制块 (Process Control Block, PCB) 描述进程的基本信息和运行态，所谓的创建进程和撤销进程，都是指对PCB 的操作。
- 线程是独立调度的基本单位。一个进程中可以有多个线程，它们共享进程资源。QQ 和浏览器是两个进程，浏览器进程里面有很多线程，例如 HTTP 请求线程、事件响应线程、渲染线程等等，线程 的并发执行使得在浏览器中点击一个新链接从而发起 HTTP 请求时，浏览器还可以响应用户的其它事件。
- 区别 I 拥有资源 进程是资源分配的基本单位，但是线程不拥有资源，线程可以访问隶属进程的资源。 II 调度 线程是独立调度的基本单位，在同一进程中，线程的切换不会引起进程切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程切换。 III 系统开销 由于创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，所付出的开销远大于创建或撤销线程时的开销。类似地，在进行进程切换时，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境的设置，而线程切换时只需保存和设置少量寄存器内容，开销很小。 IV 通信方面 线程间可以通过直接读写同一进程中的数据进行通信，但是进程通信需要借助 IPC

## 7.页存储器、段式存储器、段页式存储器以及分配原理

分页与分段的比较:

- 对程序员的透明性：分页透明，但是分段需要程序员显式划分每个段。
- 地址空间的维度：分页是一维地址空间，分段是二维的。
- 大小是否可以改变：页的大小不可变，段的大小可以动态改变。
- 出现的原因：分页主要用于实现虚拟内存，从而获得更大的地址空间；分段主要是为了使程序和数据可以被划分为逻辑上独立的地址空间并且有助于共享和保护

段页式：

- 程序的地址空间划分成多个拥有独立地址空间的段，每个段上的地址空间划分成大小相同的页。这样既拥有分段系统的共享和保护，又拥有分页系统的虚拟内存功能。段页式内存管理实现的方式：

先将程序划分为多个有逻辑意义的段，也就是前面提到的分段机制；

接着再把每个段划分为多个页，也就是对分段划分出来的连续空间，再划分固定大小的页；

这样，地址结构就由段号、段内页号和页内位移三部分组成。

段页式地址变换中要得到物理地址须经过三次内存访问：

第一次访问段表，得到页表起始地址；

第二次访问页表，得到物理页号；

第三次将物理页号与页内位移组合，得到物理地址。

可用软、硬件相结合的方法实现段页式地址变换，这样虽然增加了硬件成本和系统开销，但提高了内存的利用率。

## 8.页面置换算法

- 最佳算法
- LRU
- NRU：维护一个R与M状态位，其中R=1是最近访问过，M=1是修改过
- FIFO：缺陷可能会把经常访问的页面置换出去
- 第二次机会算法：结合了FIFO与NRU,如果该页面的R=1则将其R=0,放入末尾，否则的R=0时直接置换。
- 时钟算法：第二次机会算法需要在页表中移动页面，其采用环形链表将页面连接起来，在通过指针指向最老的页面

## 9.linux分配堆内存的算法？

- 空闲表法
- 空闲链表法
- 位图法

## 10.用户空间申请信号量的时候，是如何陷入到内核空间的？

- 系统调用：这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作，比如前例中fork()实际上就是执行了一个创建新进程的系统调用。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如Linux的int 80h中断。
- 异常：当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。
- 外围设备的中断：当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

这3种方式是系统在运行时由用户态转到内核态的最主要方式，其中系统调用可以认为是用户进程主动发起的，异常和外围设备中断则是被动的。

## 11.线程的地址空间什么的？

一般来说线程是不拥有系统资源的，地址空间都是按进程进行分配的，但在地址空间里有专属于线程的线程栈。线程在创建的时候，加上了CLOSE\_VM标记，线程的内存描述符将直接指向父进程的内存描述符，从一方面来说线程的地址空间和进程一样，线程栈不能动态增长，一旦用尽就没了，这是和生成进程的fork不同的地方。由于线程栈是从进程的地址空间中map出来的一块内存区域，原则上是线程私有的。但是同一个进程的所有线程生成的时候浅拷贝生成者的task\_struct的很多字段，其中包括所有的vma，如果愿意，其它线程也还是可以访问到的，于是一定要注意。

## 12.物理内存和虚拟内存区别？物理地址和虚拟地址区别

- 物理内存是指由于安装内存条而获得的临时储存空间。主要作用是在计算机运行时为操作系统和各种程序提供临时储存。常见的物理内存规格有256M、512M、1G、2G等，当物理内存不足时，可以用虚拟内存代替。
- 虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续可用的内存（一个连续完整的地址空间），它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要进行数据交换。虚拟内存就是为了满足物理内存的不足而提出的策略，它是利用磁盘空间虚拟出的一块逻辑内存，用作虚拟内存的磁盘空间被称为交换空间(Swap Space)。

===== 地址是针对程序的说法，本质上属于内存

- 虚拟地址：操作系统为了管理内存，其需要将内存抽象为地址空间。每个程序拥有自己的地址空间，地址空间又被分为多个页。
- 物理地址：由物理内存抽象出来

我们程序所使用的内存地址叫做虚拟内存地址（Virtual Memory Address） 实际存在硬件里面的空间地址叫物理内存地址（Physical Memory Address）

### 13.为什么使用虚拟地址

- 在支持多进程的系统中，如果各个进程的镜像文件都使用物理地址，则在加载到同一物理内存空间的时候，可能发生冲突。
- 直接使用物理地址，不便于进行进程地址空间的隔离。
- 物理内存是有限的，在物理内存整体吃紧的时候，可以让多个进程通过分时复用的方法共享一个物理页面（某个进程需要保存的内容可以暂时swap到外部的disk/flash），这有点类似于多线程分时复用共享CPU的方式。

### 14.僵尸进程

一个子进程的进程描述符在子进程退出时不会释放，只有当父进程通过 `wait()` 或 `waitpid()` 获取了子进程信息后才会释放。如果子进程退出，而父进程并没有调用 `wait()` 或 `waitpid()`，那么子进程的进程描述符仍然保存在系统中，这种进程称之为僵尸进程。

僵尸进程通过 `ps` 命令显示出来的状态为 `Z (zombie)`。

系统所能使用的进程号是有限的，如果产生大量僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程。要消灭系统中大量的僵尸进程，只需要将其父进程杀死，此时僵尸进程就会变成孤儿进程，从而被 `init` 进程所收养，这样 `init` 进程就会释放所有的僵尸进程所占有的资源，从而结束僵尸进程。

### 15.文件描述符是什么

文件描述符（file descriptor）是内核为了高效管理已被打开的文件所创建的索引，用于指代被打开的文件，对文件所有 I/O 操作相关的系统调用都需要通过文件描述符。

- 进程级别的文件描述符表：内核为每个进程维护一个文件描述符表，该表记录了文件描述符的相关信息，包括文件描述符、指向打开文件表中记录的指针。
- 系统级别的打开文件表：内核对所有打开文件维护的一个进程共享的打开文件描述表，表中存储了处于打开状态文件的相关信息，包括文件类型、访问权限、文件操作函数(file\_operations)等。

- 系统级别的 i-node 表：i-node 结构体记录了文件相关的信息，包括文件长度，文件所在设备，文件物理位置，创建、修改和更新时间等，"ls -li" 命令可以查看文件 i-node 节点

## 16.说下JMM吧?JMM解决了什么问题?知道happend before吗?

- JMM规定了内存主要划分为主内存和工作内存两种，规定所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了该线程中用到的变量的主内存的副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。
- Java内存模型定义了共享内存系统中多线程程序读写操作行为的规范，Java内存模型也就是为了解决这个并发编程问题而存在的。
- 如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。JMM可以通过happens-before关系向程序员提供跨线程的内存可见性保证（如果A线程的写操作a与B线程的读操作b之间存在happens-before关系，尽管a操作和b操作在不同的线程中执行，但JMM向程序员保证a操作将对b操作可见）

## 17.buffer与cache的区别

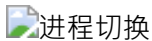
- buffer 的存在原因是生产者和消费者对资源的生产/效率速率不一致;
- cache 的存在原因是对资源调用的空间局部性

1.Buffer不是缓存，国内常用的翻译是缓冲区。2.其次，大部分场景中，Buffer是特指内存中临时存放的IO设备数据——包括读取和写入；而Cache的用处很多——很多IO设备（例如硬盘、RAID卡）上都有Cache，CPU内部也有Cache，浏览器也有Cache。3.Buffer并非用于提高性能，而Cache的目的则是提高性能。4.涉及到IO设备读写的场景中，Cache的一部分本身就是Buffer的一种。如果说某些场合Buffer可以提升IO设备的读写性能，只不过是因为Buffer本身是Cache系统的一部分，性能提升来自于Cache机制。5.Buffer占用的内存不能回收，如果被强行回收会出现IO错误。Cache占用的内存，除实现Buffer的部分外都可以回收，代价则是下一次读取需要从数据的原始位置（通常是性能更低的设备）读取。6.在IO读写过程中，任何数据的读写都必然会产生Buffer，但根据Cache算法，可能会有相当部分数据不会被Cache。

总结 第一，cache和buffer的根本区别在于它们解决的问题不同，cache解决的是性能问题，利用了访问局部性，buffer解决的是异步并发问题，第二，cache上下两端不是对等的，而buffer两端本质上是对等的agent，因此buffer类似生产者消费者队列，第三，具体实现是你中有我的，但一般是cache的实现需要内藏buffer

## 18.进程切换会发生什么

进程是由内核来管理和调度的，进程的切换只能发生在内核态。所以，进程的上下文不仅包括了**虚拟内存、栈、全局变量等用户空间的资源**，还包括了**内核堆栈、寄存器等内核空间的状态**。因此，进程的上下文切换就比系统调用时多了一步：在保存内核态资源（当前进程的内核状态和 CPU 寄存器）之前，需要先把该进程的用户态资源（虚拟内存、栈等）保存下来；而加载了下一进程的内核态后，还需要刷新进程的虚拟内存和用户栈。



## 19.内核是怎么回收的socket

- close(fd)调用会将描述符的引用计数减1，只有当socket描述符的引用计数为0时，才关闭socket,
- shutdown:方式有三种分别是 SHUT\_RD (0)：关闭sockfd上的读功能，此选项将不允许sockfd进行读操作。SHUT\_WR (1)：关闭sockfd的写功能，此选项将不允许sockfd进行写操作。SHUT\_RDWR (2)：关闭sockfd的读写功能。

## 20.内核态和用户态

一般的操作系统对执行权限进行分级，分别为用户态和内核态。用户态相较于内核态有较低的执行权限，很多操作是不被操作系统允许的，原因简单来说就是用户态出现问题（自己写的bug），也不能让操作系统崩溃呀。内核态：当一个任务（进程）执行系统调用而陷入内核代码中执行时，我们就称进程处于内核运行态（或简称为内核态）。其他的属于用户态。用户程序运行在用户态，操作系统运行在内核态。（操作系统内核运行在内核态，而服务器运行在用户态）。用户态不能干扰内核态，所以CPU指令就有两种，特权指令和非特权指令。不同的状态对应不同的指令。特权指令：只能由操作系统内核部分使用，不允许用户直接使用的指令。如，I/O指令、置终端屏蔽指令、清内存、建存储保护、设置时钟指令（这几种记好，属于内核态）。非特权指令：所有程序均可直接使用。

## 21.操作系统线程哪些状态？运行态之后到什么状态

- 初始，可运行，运行，阻塞，终止
- 阻塞，终止

## 22.线程有哪些状态，进程 有哪些状态，区别

线程：初始，可运行，运行，阻塞，终止 进程：创建、就绪、执行、阻塞、终止

## 第六章 Linux

### 1.查询端口号的方法/进程PID

#### 1. 查询端口号的方法

- `lsof -i:端口号`
- `netstat -nltp | grep 端口号`

#### 2. 进程

- `ps -ef | grep 16923`

### 2.信号与信号量的区别

- 信号：是由用户、系统或者进程发送给目标进程的信息，以通知目标进程某个状态的改变或者系统异常
- 信号量：信号量是一个特殊的变量，他的本质是计数器信号量里面记录了临界资源的数目，有多少数目，信号量的值就为多少，进程对其访问都是原子操作。他的作用就是，协调进程对共享资源的访问，让一个临界区同一时间只有一个进程在访问他，主要用于实现进程间的互斥与同步。

### 3.为什么Linux系统中存在这么多的编码

主要是每个国家有自己的语言，就像GBK是国家标准GB2312的基础上扩容后兼容GB2312的标准。GBK的文字编码是用双字节来表示的，即不论中、英文字符均使用双字节来表示，为了区分中文，将其最高位都设定成1。GBK包含全部中文字符，是国家编码，通用性比UTF8差，不过UTF8占用的数据库比GBK大。

GBK、GB2312等与UTF8之间都必须通过Unicode编码才能相互转换：

GBK、GB2312<===>Unicode<===>UTF8

而UTF-8是用以解决国际上字符的一种多字节编码，它对英文使用8位（即一个字节），中文使用24为（三个字节）来编码。UTF-8包含全世界所有国家需要用到的字符，是国际编码，通用性强。

需要注意的是，Unicode只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储，互联网的普及，强烈要求出现一种统一的编码方式。UTF-8就是在互联网上使用最广的一种unicode的实现方式。其他实现方式还包括UTF-16和UTF-32，不过在互联网上基本不用。重复一遍，这里的关系是，UTF-8是Unicode的实现方式之一。

#### 4.Linux中压缩解压命令

- 解压gzip
- 打包tar

#### 5.linux，操作系统的开机流程

- 1. BIOS上电自检(POST)
- 2. 引导装载程序 ( GRUB2)
- 3. 内核初始化
- 4. 启动systemd，其为所有进程之父

### 第七章 设计模式

#### 1.设计模式的六大原则

- 开闭原则
- 单一原则
- 里氏替换原则
- 依赖倒置原则
- 迪米特原则
- 接口隔离原则

#### 2.常用的设计模式

单例模式、策略模式、代理模式等

```
//单例模式
public class Singleton{
    private static volatile Singleton instance=null;
    private Singleton(){

    }
    public static Singleton getInstance(){
        if(instance == null){
            synchronized(Singleton.class){
                if(instance == null){
                    instance=new Singleton();
                }
            }
        }
    }
}
```

```
        return instace;
    }
}
```

## 第八章 JAVA多线程

### 1.多线程编程中如何实现资源同步

- synchronized与同步代码块
- Lock
- cas与Volatile
- TreadLocal

### 2.线程间的同步和通信怎么实现的

线程之间是共享内存的，他们之间是可以直接通讯的，同步有临界区、自旋锁、等待与唤醒，屏障

- synchronized wait/notify
- sleep join/yield
- park/unpark
- threadLocal

### 3.volatile的特性？Synchronized的锁升级过程？Synchronized是重量级锁，重量体现在哪里，轻量级锁轻量体现在哪里？

- volatile能够实现可见性与有序性
- synchronized 锁升级的过程：（无锁->偏向锁->轻量级锁->重量级锁）1.在锁对象的对象头里面有一个 threadid 字段，未访问时 threadid 为空 2.第一次访问 jvm 让其持有偏向锁，并将 threadid 设置为其线程 id 3.再次访问时会先判断 threadid 是否与其线程 id 一致。如果一致则可以直接使用此对象；如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁 4.执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁
- 当一个线程获得重量级锁之后，其余所有等待获取该锁的线程都会进入到阻塞状态
- 当锁是偏向锁的时候，此时线程B对该锁进行访问，此时偏向锁会升级为轻量级锁，线程B会通过自旋的形式获取该锁，线程不会阻塞，从而提高性能。🖼️锁区别

### 4.锁的分类

无锁：没有对资源进行锁定，所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功，其他修改失败的线程会不断重试直到修改成功。

偏向锁：对象的代码一直被同一线程执行，不存在多个线程竞争，该线程在后续的执行中自动获取锁，降低获取锁带来的性能开销。偏向锁，指的就是偏向第一个加锁线程，该线程是不会主动释放偏向锁的，只有当其他线程尝试竞争偏向锁才会被释放。

偏向锁的撤销，需要在某个时间点上没有字节码正在执行时，先暂停拥有偏向锁的线程，然后判断锁对象是否处于被锁定状态。如果线程不处于活动状态，则将对象头设置成无锁状态，并撤销偏向锁；

如果线程处于活动状态，升级为轻量级锁的状态。



轻量级锁：轻量级锁是指当锁是偏向锁的时候，被第二个线程 B 所访问，此时偏向锁就会升级为轻量级锁，线程 B 会通过自旋的形式尝试获取锁，线程不会阻塞，从而提高性能。

当前只有一个等待线程，则该线程将通过自旋进行等待。但是当自旋超过一定的次数时，轻量级锁便会升级为重量级锁；当一个线程已持有锁，另一个线程在自旋，而此时又有第三个线程来访时，轻量级锁也会升级为重量级锁。

重量级锁：指当有一个线程获取锁之后，其余所有等待获取该锁的线程都会处于阻塞状态。

重量级锁通过对象内部的监视器 ( monitor ) 实现，而其中 monitor 的本质是依赖于底层操作系统的 Mutex Lock 实现，操作系统实现线程之间的切换需要从用户态切换到内核态，切换成本非常高

## 5.线程池 ( todo )

- ThreadPoolExecutor()：最原始的线程池
- execute()：只能执行 Runnable 类型的任务。
- submit()：可以执行 Runnable 和 Callable 类型的任务

## 6.线程池都有哪些状态？

- RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。
- SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。
- STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。
- TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为TIDYING 状态时，会执行钩子方法 terminated()。
- TERMINATED：terminated()方法结束后，线程池的状态就会变成这个。

## 7.synchronized的原理？

synchronized 是由一对 monitorenter/monitorexit 指令实现的，monitor 对象是同步的基本实现单元。在 Java 6 之前，monitor 的实现完全是依靠操作系统内部的互斥锁，因为需要进行用户态到内核态的切换，所以同步操作是一个无差别的重量级操作，性能也很低。但在 Java 6 的时候，Java 虚拟机 对此进行了大刀阔斧地改进，提供了三种不同的 monitor 实现，也就是常说的三种不同的锁：偏向锁 ( Biased Locking )、轻量级锁和重量级锁，大大改进了其性能。


## 8.公平锁、非公平锁解释一下

- 非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么就获取到锁返回了。
- 非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了 ( state == 0 )，非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

公平锁和非公平锁就这两点区别，如果这两次 CAS 都不成功，那么后面非公平锁和公平锁是一样的，都要进入到阻塞队列等待唤醒。

相对来说，非公平锁会有更好的性能，因为它的吞吐量比较大。当然，非公平锁让获取锁的时间变得更加不确定，可能会导致在阻塞队列中的线程长期处于饥饿状态。

## 9.ThreadLocal原理是什么？

 ThreadLocal ThreadLocalMap 是 ThreadLocal 类中的内部静态类。每个线程都有一个 ThreadLocalMap 类变量，该类存放了键值对，其中键值对的 key 就是 ThreadLocal 的一个引用，value 就是 ThreadLocal 对应的值。属于entry[]数组，ThreadLocal 实例本身并没有存储值，值都是存放在 ThreadLocalMap 中，ThreadLocal 的作用就是作为键值对中的一个 key。其中key是ThreadLocal本身弱引用，value对应的线程变量副本。下一次GC时，key弱引用会被回收，但是value强引用，如果线程一直存在，value会一直存在，造成内存泄漏。解决方法：调用get,set,remove方法都会对key为null的数据进行清楚。

## 10.Reentrantlock默认公平还是非公平？synchronized?

都属于非公平锁

## 11.wait、notifyAll的底层实现，以及可重入锁

- Owner 线程发现条件不满足，调用 wait 方法，即可进入 WaitSet 变为 WAITING 状态
- BLOCKED 和 WAITING 的线程都处于阻塞状态，不占用 CPU 时间片
- BLOCKED 线程会在 Owner 线程释放锁时唤醒
- WAITING 线程会在 Owner 线程调用 notify 或 notifyAll 时唤醒，但唤醒后并不意味着立刻获得锁，仍需进入 EntryList 重新竞争
- 重入锁实现可重入性原理或机制是：每一个锁关联一个线程持有者和计数器，当计数器为 0 时表示该锁没有被任何线程持有，那么任何线程都可能获得该锁而调用相应的方法；当某一线程请求成功后，JVM 会记下锁的持有线程，并且将计数器置为 1；此时其它线程请求该锁，则必须等待；而该持有锁的线程如果再次请求这个锁，就可以再次拿到这个锁，同时计数器会递增；当线程退出同步代码块时，计数器会递减，如果计数器为 0，则释放该锁。

## 第九章 算法

### 1.堆排序与快速排序区别以及各自的使用场景

堆排序与快速排序的时间复杂度都是 $n\log n$ ，但是快排的最差时间复杂度位 ( $n^2$ )，相对于堆排序稍微差了一点。但是从综合性能上分析还是快速排序的性能更好，因为当数据量过大时，堆排序需要对顶节点进行堆化，而不是像快速排序对局部进行顺序访问，此外堆排序在建堆过程中数据交换次数要远大于快速排序的次数。

### 2.B+ 树和红黑树的区别

- 红黑树：它是一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是红或黑（非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍（这里是和平衡二叉树的主要区别），因此，红黑树是一种弱平衡二叉树（由于是弱平衡，可以看到，在相同的节点情况下，AVL树的高度低于红黑树），相对于要求严格的AVL树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，我们就用红黑树。如果应用场景中对插入删除不频繁，只是对查找要求较高，那么AVL还是较优于红黑树。
1. java1.8+，桶上的链表长度大于等于8时，链表转化成红黑树并且只有当数组容量大于64时，链表才会转化成红黑树
  2. java1.8+，桶上的红黑树大小小于等于6时，红黑树转化成链表

- B+的叶子节点以链表的形式，且b+的树的高度一般比较低

### 3.有序数组截断、交换后的查找算法

先查找到截断的位置，拆分成两部分，在这两部分里面使用二分查找？这是肯定可以的，并且找截断位置也使用二分法

4.给定1个二维字符数组m和1个单词w，搜索w是否在m中。搜索的定义是从m的任意位置开始，可以上下左右移动，依次和w每个字符匹配，如果w能匹配完，则存在，否则不存在。并且二维数组中的元素不能被重复使用。

```
public class Main {

    public int[][] dir={{1,0},{-1,0},{0,1},{0,-1}};

    public boolean exist(char[][] board, String word) {

        if (word==null) return false;
        boolean[][] dp=new boolean[board.length][board[0].length];
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
                boolean ans=help(board,word,i,j,0,dp);
                if (ans) return true;
            }
        }

        return false;
    }

    private boolean help(char[][] board, String word, int i, int j, int index,
        boolean[][] dp) {

        if (board[i][j]!=word.charAt(index)) return false;
        if (index==word.length()-1) return true;
        dp[i][j]=true;
        boolean result=false;
        for (int K = 0; K < dir.length; K++) {
            int newi=dir[K][0]+i;
            int newj=dir[K][1]+j;
            if (newi>=0&&newj>=0&&newi<board.length&&newj<board[0].length){
                if (!dp[newi][newj]){
                    boolean flag=help(board,word,newi,newj,index+1,dp);
                    if (flag){
                        result=true;
                        break;
                    }
                }
            }
        }

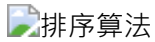
        return result;
    }
}
```

```

        dp[i][j]=false;
        return result;
    }
}

```

## 5.常用排序算法及时间复杂度



排序方法	时间复杂度 ( 平均 )	时间复杂度 ( 最坏 )	空间复杂度	稳定性
堆排序	$n \log n$	$n \log n$	1	不稳定
快速排序	$n \log n$	$n^2$	$n \log n$	不稳定
冒泡排序	$n^2$	$n^2$	1	稳定
归并排序	$n \log n$	$n \log n$	$n$	稳定
桶排序	$n+k$	$n^2$	$n*k$	稳定

## 6.单例模式

```

//单例模式
public class Singleton{
    private static volatile Singleton instace=null;
    private Singleton(){

    }
    public static Singleton getInstance(){
        if(instance == null){
            synchronized(Singleton.class){
                if(instance == null){
                    instace=new Singleton();
                }
            }
        }
        return instace;
    }
}

```

7.学生成绩表 **table1**, 学生、课程、成绩 · (name,sbuject,score) · 查询出所有课程都大于80分的学生的平均成绩。

```

select avg(score), name from table1 where name not in (select distinct name from table1 where score < 80)
group by name ;

```

8.有一个 **1GB** 大小的文件，文件里每一行是一个词，每个词的大小不超过 **16B**，内存大小限制是 **1MB**，要求返回频数最高的 **100** 个词

9.平均延迟最大的调用链(力扣743)

10.堆排序基本思路

- 1 将待排序序列构造成为一个大顶堆。
- 2 此时，整个序列的最大值就是堆顶的根节点。
- 3 将其与末尾元素进行交换，此时末尾就为最大值。
- 4 然后将剩余  $n-1$  个元素重新构造成为一个堆，这样会得到  $n$  个元素的次小值。如此反复执行，便能得到一个有序序列了。

11. LeetCode 31题

```
public void nextPermutation(int[] nums) {
    if(nums==null||nums.length==0) return;
    int firstIndex=-1;
    for (int i = nums.length-2; i>=0; i--) {
        if (nums[i]<nums[i+1]){
            firstIndex=i;
            break;
        }
    }
    if (firstIndex== -1) {
        Arrays.sort(nums);
        return;
    }
    int second=-1;
    for (int i = nums.length-1; i >= 0; i--) {
        if (nums[firstIndex]<nums[i]){
            second=i;
            break;
        }
    }
    swap(nums,firstIndex,second);
    reverse(nums,firstIndex+1,nums.length-1);
    return;
}

public void reverse(int[] nums,int i,int j){
    while (i<j){
        swap(nums,i++,j--);
    }
}

public void swap(int[] nums,int i,int j){
    int temp=nums[i];
    nums[i]=nums[j];
    nums[j]=temp;
}
```

## 12.从数据库表查询语文成绩及格但是平均分不及格的学生

```
select student.id,student.score,a.avg from student
inner JOIN (
    select id,avg(score) as avg from student
    group by id having avg(score) < 60
) a
on student.id = a.id where student.name='Chinese' and score >= 60
```

## 消息队列

### 1.消息队列的交换机

直连交换机：Direct exchange 扇形交换机：Fanout exchange 主题交换机：Topic exchange 首部交换机：Headers exchange