数据库

1. 数据库三范式是什么？

第一范式：表中每个字段都不能再分。

第二范式：满足第一范式并且表中的非主键字段都依赖于主键字段。

第三范式：满足第二范式并且表中的非主键字段必须不传递依赖于主键字段，每一列数据和主键直接相关。

2.什么是数据库事务？

事务具有四大特性：一致性、原子性、隔离性、持久性。

原子性：原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚。

一致性：事务开始前和结束后，数据库的完整性约束没有被破坏。比如A向B转账，不可能A扣了钱，B却没收到。

隔离性：隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

持久性：持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的

数据库事务是指：几个SQL语句，要么全部执行成功，要么全部执行失败。比如银行转账就是事务的典型场景。

数据库事务的三个常用命令：Begin Transaction、Commit Transaction、RollBack Transaction。

3.什么是视图？

视图实际上是在数据库中通过Select查询语句从多张表中提取的多个表字段所组成的虚拟表。

l 视图并不占据物理空间，所以通过视图查询出的记录并非保存在视图中，而是保存在原表中。

l 通过视图可以对指定用户隐藏相应的表字段，起到保护数据的作用。

l 在满足一定条件时，可以通过视图对原表中的记录进行增删改操作。

l 创建视图时，只能使用单条select查询语句。

4.什么是索引？

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。

l 索引分为：聚集索引、非聚集索引、唯一索引等。

l 一张表可以有多个唯一索引和非聚集索引，但最多只能有一个聚集索引。

l 索引可以包含多列。

l 合理的创建索引能够提升查询语句的执行效率，但降低了新增、删除操作的速度，同时也会消耗一定的数据库物理空间。

是一种快速查询表中内容的机制，类似于字典。

运用在表中某个些字段上，但存储时，独立于表外。
索引一旦建立，Oracle管理系统会对其进行自动维护，而且由Oracle管理系统决定何时使用索引。
用户不用再查询语句中指定使用哪个索引。
在定义primary key或unique约束后系统自动在想印的列上创建索引
用户也能按自己的需求，对指定单个字段或多个字段，添加索引。

什么时候要创建索引

- （1）表经常进行 SELECT 操作
- （2）表很大(记录超多)，记录内容分布范围很广
- （3）列名经常在 WHERE 子句或连接条件中出现

什么时候不要创建索引

- （1）表经常进行 INSERT/UPDATE/DELETE 操作
- （2）表很小(记录超少)
- （3）列名不经常作为连接条件或出现在 WHERE 子句中

索引优缺点：

- 索引加快数据库的检索速度
- 索引降低了插入、删除、修改等维护任务的速度(虽然索引可以提高查询速度，但是它们也会导致数据库系统更新数据的性能下降，**因为大部分数据更新需要同时更新索引**)
- 唯一索引可以确保每一行数据的唯一性，通过使用索引，可以在查询的过程中使用优化隐藏器，提高系统的性能
- 索引需要占物理和数据空间

索引分类：

- **唯一索引**：唯一索引不允许两行具有相同的索引值
- **主键索引**：为表定义一个主键将自动创建主键索引，主键索引是唯一索引的特殊类型。主键索引要求主键中的每个值是唯一的，并且不能为空
- **聚集索引**(Clustered)：表中各行的物理顺序与键值的逻辑（索引）顺序相同，每个表只能有一个
- **非聚集索引**(Non-clustered)：非聚集索引指定表的逻辑顺序。数据存储在一个位置，索引存储在另一个位置，索引中包含指向数据存储位置的指针。可以有多个，小于249个

5.什么是存储过程？

存储过程是一个预编译的SQL语句，优点是允许模块化的设计，就是说只需创建一次，以后在该程序中就可以调用多次。如果某次操作需要执行多次SQL，使用存储过程比单纯SQL语句执行要快。

存储过程的优点：

能够将代码封装起来
保存在数据库之中
让编程语言进行调用
存储过程是一个预编译的代码块，执行效率比较高
一个存储过程替代大量T_SQL语句，可以降低网络通信量，提高通信速率

存储过程的缺点：

每个数据库的存储过程语法几乎都不一样，十分难以维护（不通用）
业务逻辑放在数据库上，难以迭代

预编译又称为预处理，是做些代码文本的替换工作。
处理#开头的指令，比如拷贝#include包含的文件代码，#define宏定义的替换,条件编译等就是为编译做的预备工作的阶段主要处理#开始的预编译指令

6.什么是触发器？
触发器是一中特殊的存储过程，主要是通过事件来触发而被执行的。它可以强化约束，来维护数据的完整性和一致性，可以跟踪数据库内的操作从而不允许未经许可的更新和变化。可以联级运算。如，某表上的触发器上包含对另一个表的数据操作，而该操作又会导致该表触发器被触发。

7.写出一条Sql语句：取出表A中第31到第40记录 （MS-SQLServer）
解1：select top 10 * from A where id not in (select top 30 id from A)
解2：select top 10 * from A where id > (select max(id) from (select top 30 id from A)as A)
解3：select * from (select *, Row_Number() OVER (ORDER BY id asc) rowid FROM A) as A where rowid between 31 and 40

8.写出一条Sql语句：取出表A中第31到第40记录 （Mysql）
```
select * from A limit 30, 10
```

9.写出一条Sql语句：取出表A中第31到第40记录 （Oracle）
```
select *
from (select A.*,
row_number() over (order by id asc) rank
FROM A)
where rank >=31 AND rank<=40;
```

7.在关系型数据库中如何描述多对多的关系?
在关系型数据库中描述多对多的关系,需要建立第三张数据表。比如学生选课,需要在学生信息表和课程信息表的基础上,再建立选课信息表,该表中存放学生Id和课程Id。

8.什么是数据库约束,常见的约束有哪几种?
数据库约束用于保证数据库表数据的完整性（正确性和一致性）。可以通过定义约束\索引\触发器来保证数据的完整性。
总体来讲,约束可以分为:
主键约束：primary key；
外键约束：foreign key；
唯一约束：unique；
检查约束：check；
空值约束：not null；用于控制字段的值范围。
默认值约束：default

9.列举几种常用的聚合函数?
Sum:求和\ Avg:求平均数\ Max:求最大值\ Min:求最小值\ Count:求记录数

10.什么是内联接、左外联接、右外联接？
l 内联接（Inner Join）：匹配2张表中相关联的记录。
l 左外联接（Left Outer Join）：除了匹配2张表中相关联的记录外，还会匹配左表中剩余的记录，右表中未匹配到的字段用NULL表示。
l 右外联接（Right Outer Join）：除了匹配2张表中相关联的记录外，还会匹配右表中剩余的记录，左表中未匹配到的字段用NULL表示。
在判定左表和右表时，要根据表名出现在Outer Join的左右位置关系。

11.如何在删除主表记录时，一并删除从表相关联的记录？
如果两张表存在主外键关系，那么在删除主键表的记录时，如果从表有相关联的记录，那么将导致删除失败。
在定义外键约束时，可以同时指定3种删除策略：一是将从表记录一并删除（级联删除）；二是将从表记录外键字段设置为NULL；三是将从表记录外键字段设置为默认值。
级联是用来设计一对多关系的。例如一个表存放老师的信息:表A（姓名，性别，年龄），姓名为主键。还有一张表存放老师所教的班级信息:表B（姓名，班级）。他们通过姓名来级联。级联的操作有级联更新，级联删除。
在启用一个级联更新选项后，就可在存在相匹配的外键值的前提下更改一个主键值。系统会相应地更新所有匹配的外键值。如果在表A中将姓名为张三的记录改为李四，那么表B中的姓名为张三的所有记录也会随着改为李四。级联删除与更新相类似。如果在表A中将姓名为张三的记录删除，那么表B中的姓名为张三的所有记录也将删除。
级联删除示例：
alter table 从表名
add constraint 外键名

foreign key(字段名) references 主表名(字段名)
on delete cascade

12.超键，候选键，主键，外键分别是什么

超键：**在关系中能唯一标识元组的属性集称为关系模式的超键**。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。**超键包含候选键和主键**。
候选键(候选码)：是最小超键，即没有冗余元素的超键。
主键(主码)：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。
外键：在一个表中存在的另一个表的主键称此表的外键。

候选码和主码：

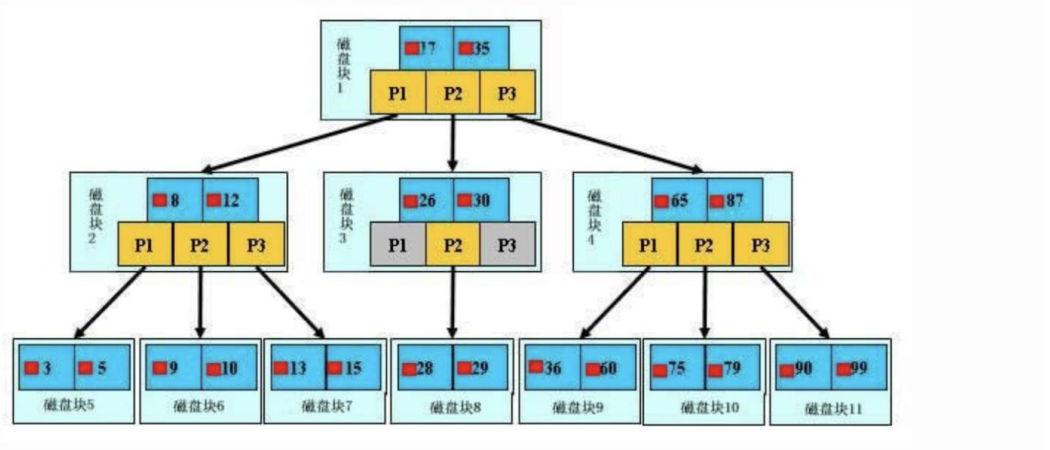
例子：邮寄地址（城市名，街道名，邮政编码，单位名，收件人）

- **它有两个候选键:{城市名，街道名} 和 {街道名，邮政编码}**
- **如果我选取{城市名，街道名}作为唯一标识实体的属性，那么{城市名，街道名}就是主码(主键)**

13.varchar和char的区别

Char是一种固定长度的类型，varchar是一种可变长度的类型
如果存进去的是‘csdn’,那么char所占的长度依然为10，除了字符‘csdn’外，后面跟六个空格，varchar就立马把长度变为4了，取数据的时候，char类型的要用trim()去掉多余的空格，而varchar是不需要的。
char的存取数度还是要比varchar要快得多，因为其长度固定，方便程序的存储与查找。

14.B+树索引



b+树的查找过程

如图所示，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。

b+树性质

1.通过上面的分析，我们知道IO次数取决于b+数的高度h，假设当前数据表的数据为N，每个磁盘块的数据项的数量是m，则有h=㏒(m+1)N，当数据量N一定的情况下，m越大，h越小；而m = 磁盘块的大小 / 数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的，如果数据项占的空间越小，数据项的数量越多，树的高度越低。这就是为什么每个数据项，即索引字段要尽量的小，比如int占4字节，要比bigint8字节少一半。这也是为什么b+树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。当数据项等于1时将会退化成线性表。

当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+数是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

B+树的表示要比B树要“胖”，原因在于B+树中的非叶子节点会冗余一份在叶子节点中，并且叶子节点之间用指针相连。

15.聚集函数

SELECT city FROM weather WHERE temp_lo = max(temp_lo); *WRONG*不过这个方法不能运转，因为聚集 max 不能用于 WHERE 子句中。（存在这个限制是因为 WHERE 子句决定哪些行可以进入聚集阶段；因此它必需在聚集函数之前计算。） 不过，我们通常都可以用其它方法实现我们的目的；这里我们就可以使用*子查询*：

SELECT city FROM weather WHERE temp_lo = (SELECT max(temp_lo) FROM weather);

理解聚集和SQL的 WHERE 以及 HAVING 子句之间的关系对我们非常重要。WHERE 和 HAVING 的基本区别如下： WHERE 在分组和聚集计算之前选取输入行（因此，它控制哪些行进入聚集计算）， 而 HAVING 在分组和聚集之后选取分组的行。 因此，WHERE 子句不能包含聚集函数； 因为试图用聚集函数判断那些行输入给

聚集运算是没有意义的。 相反，HAVING 子句总是包含聚集函数。 （严格说来，你可以写不使用聚集的 HAVING 子句， 但这样做只是白费劲。同样的条件可以更有效地用于 WHERE 阶段。）

当应用于空集时，SUM、AVG、MAX 和 MIN 函数可以返回 null 值。当 COUNT 函数应用于空集时，它返回零（0）。如果返回值可能是 NULL，那么使用包装器类型；否则，容器会显示 ObjectNotFound 异常。

drop、truncate、 delete区别

最基本：

- drop直接删掉表。
- truncate删除表中数据，再插入时自增长id又从1开始。
- delete删除表中数据，可以加where字句。

16.非关系型数据库和关系型数据库区别，优势比较?

非关系型数据库的优势：

- **性能：**NOSQL是基于键值对的，可以想象成表中的主键和值的对应关系，而且不需要经过SQL层的解析，所以性能非常高。
- **可扩展性：**同样也是因为基于键值对，数据之间没有耦合性，所以非常容易水平扩展。

关系型数据库的优势：

- **复杂查询：**可以用SQL语句方便的在一个表以及多个表之间做非常复杂的数据查询。
- **事务支持：**使得对于安全性能很高的数据访问要求得以实现。

like %和-的区别

通配符的分类:
%百分号通配符:表示任何字符出现任意次数(可以是0次).
_下划线通配符:表示只能匹配单个字符,不能多也不能少,就是一个字符.

17.最左前缀原则

多列索引：
ALTER TABLE people ADD INDEX lname_fname_age (lame,fname,age);
为了提高搜索效率，我们需要考虑运用多列索引,由于索引文件以B－Tree格式保存，所以我们不用扫描任何记录，即可得到最终结果。
注：在mysql中执行查询时，只能使用一个索引，如果我们在lname,fname,age上分别建索引,执行查询时，只能使用一个索引，mysql会选择一个最严格(获得结果集记录数最少)的索引。
最左前缀原则：顾名思义，就是最左优先，上例中我们创建了lname_fname_age多列索引,相当于创建了(lname)单列索引，(lname,fname)组合索引以及(lname,fname,age)组合索引。
ALTER TABLE 'table_name' ADD INDEX index_name('col1','col2','col3')

https://blog.csdn.net/qq_19557947/article/details/76951912

18.MySQL B+Tree索引和Hash索引的区别?

Hash索引和B+树索引的特点：

- Hash索引结构的特殊性，其检索效率非常高，索引的检索可以一次定位;
- B+树索引需要从根节点到枝节点，最后才能访问到页节点这样多次的IO访问;

19.为什么不都用Hash索引而使用B+树索引？

Hash索引仅仅能满足"=","IN"和""查询，不能使用范围查询

Hash索引遇到大量Hash值相等的情况后性能并不一定就会比B+树索引高

为什么说B+比B树更适合实际应用中操作系统的文件索引和数据库索引？

(1) B+的磁盘读写代价更低

B+的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。

(2) B+tree的查询效率更加稳定

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

20. 聚集索引和非聚集索引区别?

聚集索引**表记录的排列顺序和索引的排列顺序一致，所以查询效率快。**

聚集索引**指定了表中记录的逻辑顺序，但是记录的物理和索引不一定一致。**

聚集索引和非聚集索引的根本区别是表记录的排列顺序和与索引的排列顺序是否一致。

21. 锁的基本概念

当并发事务同时访问一个资源时，有可能导致数据不一致，因此需要一种机制来将数据访问顺序化，以保证数据库数据的一致性。多个事务同时读取一个对象的时候，是不会有冲突的。同时读和写，或者同时写才会产生冲突。

共享锁（Shared Lock，也叫S锁）

共享锁(S)表示对数据进行读操作。因此多个事务可以同时为一个对象加共享锁。

排他锁(Exclusive Lock，也叫X锁)

排他锁表示对数据进行写操作。如果一个事务对对象加了排他锁，其他事务就不能再给它加任何锁了。

锁的粒度

就是通常我们所说的锁级别。**MySQL有三种锁的级别：页级、表级、行级。**

表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低。

行级锁：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。

页面锁：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。

行锁(Row Lock)

对一行记录加锁，只影响一条记录。
通常用在DML语句中，如INSERT, UPDATE, DELETE等。

表锁(Table Lock)

对整个表加锁，影响标准的所有记录。通常用在DDL语句中，如DELETE TABLE,ALTER TABLE等。
很明显，表锁影响整个表的数据，因此并发性不如行锁好。
https://www.cnblogs.com/wenxiaofei/p/9853682.html

乐观锁
总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量。
悲观锁
总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。
脏读
所谓脏读是指一个事务中访问到了另外一个事务未提交的数据，获得更新前的值
幻读
一个事务读取2次，得到的记录条数不一致
不可重复读
一个事务读取同一条记录2次，得到的结果不一致
解决方法

22. 数据库的主从复制
主从复制的几种方式:
同步复制:

所谓的同步复制，意思是master的变化，必须等待slave-1,slave-2,...,slave-n完成后才能返回。 这样，显然不可取，也不是MySQL复制的默认设置。比如，在WEB前端页面上，用户增加了条记录，需要等待很长时间。
异步复制:
如同AJAX请求一样。master只需要完成自己的数据库操作即可。至于slaves是否收到二进制日志，是否完成操作，不用关心,MySQL的默认设置。
半同步复制:
master只保证slaves中的一个操作成功，就返回，其他slave不管。 这个功能，是由google为MySQL引入的。

master的写操作，slaves被动的进行一样的操作，保持数据一致性。如果slave可以主动的进行写操作，slave又无法通知master，这样就导致了master和slave数据不一致了。
主从复制中，可以有N个slave，实现数据备份，异地容灾。

插入数据 insert into t1 values(5,'xiaoming',null); insert into t1 (id,name) values (2,'aa'); insert into t1 values(5,'xiaoming',null),(5,'xiaoming',null),(5,'xiaoming',null); insert into t1 (id,name) values (2,'aa'),(2,'aa'),(2,'aa');
查询 select * from t1; select name from t1; select * from t1 where id=10;
修改 update t1 set age=100 where id=10;
删除 delete from t1 where id=10;

修改表名 rename table t1 to t2;
修改表属性 alter table t1 engine=myisam/innodb charset=utf8/gbk;
添加表字段 alter table t1 add age int first/after xxx;
删除表字段 alter table t1 drop age;
修改表字段名和类型 alter table t1 change age newAge int;
修改表的类型和位置 alter table t1 modify age int first/after xx;
删除表 drop table t1;

23.MySQL主从复制原理
MySQL 主从复制概念
MySQL 主从复制是指数据可以从一个MySQL数据库服务器主节点复制到一个或多个从节点。MySQL 默认采用异步复制方式，这样从节点不用一直访问主服务器来更新自己的数据，数据的更新可以在远程连接上进行，从节点可以复制主数据库中的所有数据库或者特定的数据库，或者特定的表。

MySQL 主从复制主要用途
l 读写分离
在开发工作中，有时候会遇见某个sql 语句需要锁表，导致暂时不能使用读的服务，这样就会影响现有业务，使用主从复制，让主库负责写，从库负责读，这样，即使主库出现了锁表的情景，通过读从库也可以保证业务的正常运作。

l **数据实时备份**，当系统中某个节点发生故障时，可以方便的故障切换
l 高可用HA
l 架构扩展
随着系统中业务访问量的增大，如果是单机部署数据库，就会导致I/O访问频率过高。有了主从复制，增加多个数据存储节点，将负载分布在多个从节点上，降低单机磁盘I/O访问的频率，提高单个机器的I/O性能。

MySQL 主从形式
一主一从，一主多从，提高系统的读性能。一主一从和一主多从是最常见的主从架构，实施起来简单并且有效，不仅可以实现HA，而且还能读写分离，进而提升集群的并发能力。
多主一从 （从5.7开始支持）多主一从可以将多个mysql数据库备份到一台存储性能比较好的服务器上。
双主复制，也就是互做主从复制，每个master既是master，又是另外一台服务器的slave。这样任何一方所做的变更，都会通过复制应用到另外一方的数据库中。
级联复制。级联复制模式下，部分slave的数据同步不连接主节点，而是连接从节点。因为如果主节点有太多的从节点，就会损耗一部分性能用复制，那么我们可以让3~5个从节点连接主节点，其它从节点作为二级或者三级与从节点连接，这样不仅可以缓解主节点的压力，并且对数据一致性没有负面影响。

MySQL 主从复制原理
MySQL主从复制涉及到三个线程，一个运行在主节点（log dump thread），其余两个(I/O thread, SQL thread)运行在从节点。
主节点 binary log dump 线程
当从节点连接主节点时，主节点会创建一个log dump 线程，用于发送bin-log的内容。在读取bin-log中的操作时，此线程会对主节点上的bin-log加锁，当读取完成，甚至在发动给从节点之前，锁会被释放。
从节点I/O线程
当从节点上执行`start slave`命令之后，从节点会创建一个I/O线程用来连接主节点，请求主库中更新的bin-log。I/O线程接收到主节点binlog dump 进程发来的更新之后，保存在本地relay-log中。
从节点SQL线程
SQL线程负责读取relay log中的内容，解析成具体的操作并执行，最终保证主从数据的一致性。

复制的基本过程如下：
从节点上的I/O 进程连接主节点，并请求从指定日志文件的指定位置（或者从最开始的日志）之后的日志内容；
主节点接收到来自从节点的I/O请求后，通过负责复制的I/O进程根据请求信息读取指定日志指定位置之后的日志信息，返回给从节点。返回信息中除了日志所包含的信息之外，还包括本次返回的信息的bin-log file 的以及bin-log position；从节点的I/O进程接收到内容后，将接收到的日志内容更新到本机的relay log中，并将读取到的binary

log文件名和位置保存到master-info 文件中，以便在下一次读取的时候能够清楚的告诉Master“我需要从某个bin-log 的哪个位置开始往后的日志内容，请发给我”；
Slave 的 SQL线程检测到relay-log 中新增加了内容后，会将relay-log的内容解析成在祝节点上实际执行过的操作，并在本数据库中执行。
https://zhuanlan.zhihu.com/p/50597960

数据结构

1. 数组和链表的区别

从逻辑结构来看：

a) 数组必须事先定义固定的长度（元素个数），不能适应数据动态地增减的情况。当数据增加时，可能超出原先定义的元素个数；当数据减少时，造成内存浪费；数组可以根据下标直接存取。

b) 链表动态地进行存储分配，可以适应数据动态地增减的情况，且可以方便地插入、删除数据项。（数组中插入、删除数据项时，需要移动其它数据项，非常繁琐）链表必须根据next指针找到下一个元素

从内存存储来看：

a) (静态)数组从栈中分配空间, 对于程序员方便快速,但是自由度小

b) 链表从堆中分配空间, 自由度大但是申请管理比较麻烦

从上面的比较可以看出，如果需要快速访问数据，很少或不插入和删除元素，就应该用数组；相反，如果需要经常插入和删除元素就需要用链表数据结构了。

常用的排序方法
插入排序，冒泡排序，选择排序，快速排序，堆排序，归并排序，基数排序，希尔排序等。

冒泡排序法 插入排序法 堆排序法 二叉树排序法

O(n^2） O(n^2) O(nlog2 n) 最差O(n2)平均O(n*log2n)

快速排序法 希尔排序法

最差O(n2)平均O(n*log2n) O(nlog n)不稳定

文件系统和数据库系统一般都采用树（特别是B树）的数据结构数据，主要为排序和检索的效率。二叉树是一种最基本最典型的排序树，链表。

1. 磁盘IO比内存操作性是相差千级数量级的差异，因此应该尽量减少io操作

2. 磁盘组织形式是扇区，因为数据的局部性，因此一次磁盘io会把一个扇区的都预读出来

索引的目标是要找到数据所在的物理位置，因此用树去实现搜索数据所在物理位置，每个节点对应一次IO，因此结合知识点1为了减少搜索时间，就需要控制树的高度，那这样的话二叉树明显不行，因为二叉树插入的话树的高度是没办法控制的，因此采用B+树的形式，每个节点对应很多子节点，插入节点时增加子节点而不是增加树高度。更进一步，采用B+树时在相同数据量的情况下如何降低树的高度？当然是增加每一层的数据量，而考虑到知识点2，一个节点对应一个扇区大小存储多个数据项，既可以降低索引文件大小，又可以在相同数据量的情况下减少每层节点数，提高性能。

链表(Linked List)也是线性结构，它与数组看起来非常像，但是它们的内存分配方式、内部结构和插入删除操作方式都不一样。链表是一系列节点组成的链，每一个节点保存了数据以及指向下一个节点的指针。链表头指针指向第一个节点，如果链表为空，则头指针为空或者为 null。

树
N 叉树(N-ary Tree)，平衡树(Balanced Tree)，二叉树(Binary Tree)，二叉查找树(Binary Search Tree)，平衡二叉树(AVL Tree)，红黑树(Red Black Tree)

哈希表
哈希(Hash)将某个对象变换为唯一标识符，该标识符通常用一个短的随机字母和数字组成的字符串来代表。哈希可以用来实现各种数据结构，其中最常用的就是**哈希表(hash table)**。

大小顶堆
构造时间复杂度：n
插入/删除时间复杂度：最好O(1) 最坏worst O(logn) 平均O(1)
搜索O(n)

map, set, multimap, and multiset

上述四种容器采用红黑树实现，红黑树是平衡二叉树的一种。不同操作的时间复杂度近似为:
插入: O(logN)
查看:O(logN)
删除:O(logN)

hash_map, hash_set, hash_multimap, and hash_multiset
上述四种容器采用哈希表实现，不同操作的时间复杂度为：
插入:O(1)，最坏情况O(N)。
查看:O(1)，最坏情况O(N)。

删除:O(1)，最坏情况O(N)。

Vector
vector（连续的( )空间存储，可以使用[ ]操作符）可以快速的访问随机的元素，快速的在末尾插入元素，但是在序列中间随机的插入、删除元素要慢。而且，如果一开始分配的空间不够时，有一个重新分配更大空间的过程。
连续存储结构，每个元素在内存上是连续的；支持高效的随机访问和在尾端插入/删除操作，但其他位置的插入/删除操作效率低下；相当于一个数组，但是与数组的区别为：内存空间的扩展。vector支持不指定vector大小的存储，但是数组的扩展需要程序员自己写。
STL内部实现时，首先分配（assign）一个非常大的内存空间预备进行存储，即capacity（）函数返回的大小，当超过此分配的空间时再整体重新放分配一块内存存储
扩充空间（不论多大）都应该这样做：
（1）配置一块新空间
（2）将旧元素一一搬往新址
（3）把原来的空间释放还给系统
insert和erase时间复杂度O(1)

Deque
连续存储结构，即其每个元素在内存上也是连续的，类似于vector，不同之处在于，deque提供了两级数组结构， 第一级完全类似于vector，代表实际容器；另一级维护容器的首位地址。这样，deque除了具有vector的所有功能外，还支持高效的首/尾端插入/删除操作。在deque中不存在容量的概念，因为它维护的是一段动态的存储空间

deque 采用一块连续的空间存放指针，每个指针都指向一个连续的存储空间，成为缓冲区，这些缓冲区才是元素真正存储的空间。每个缓冲区的大小是相同的，所以当计算deque的size时，不需要一个一个遍历（traverse）元素，是需要计算指针数组中一共使用了多少个指针，和最后一个指针指向缓冲区使用的个数，就可以计算出来。
deque中的元素就存在于这一段一段的缓冲区中。如果当存放指针的连续空间不足时，就需要像操作vector空间不足一样，开辟一段更大的内存空间，将原来的指针数组拷贝到新的空间中去。

insert操作和erase操作相似，首先检查插入或者删除的位置，如果前面的元素较少，那么向前方扩张（收缩）；如果后面元素较少，反之。假设想前方扩张（收缩），将之前的元素一一移动，如果是插入，将插入的元素赋值给腾出的空间，若是删除，则pop_front第一个元素即可。

deque　双端队列 double-end queue
deque是在功能上合并了vector和list。
优点：(1) 随机访问方便，即支持[ ]操作符和vector.at()
(2) 在内部方便的进行插入和删除操作

(3) 可在两端进行push、pop
缺点：占用内存多
使用区别：
（1）如果你需要高效的随即存取，而不在乎插入和删除的效率，使用vector
（2）如果你需要大量的插入和删除，而不关心随机存取，则应使用list
（3）如果你需要随机存取，而且关心两端数据的插入和删除，则应使用deque

List

C++11中，针对顺序容器(如vector、deque、list)，新标准引入了三个新成员：emplace_front、emplace和emplace_back，这些操作构造而不是拷贝元素。这些操作分别对应push_front、insert和push_back，允许我们将元素放置在容器头部、一个指定位置之前或容器尾部。

当调用push或insert成员函数时，我们将元素类型的对象传递给它们，这些对象被拷贝到容器中。而当我们调用一个emplace成员函数时，则是将参数传递给元素类型的构造函数。emplace成员使用这些参数在容器管理的内存空间中直接构造元素。

非连续存储结构，具有双链表结构，每个元素维护一对前向和后向指针，因此支持前向/后向遍历。支持高效的随机插入/删除操作，但随机访问效率低下，且由于需要额外维护指针，开销也比较大。每一个结点都包括一个信息快Info、一个前驱指针Pre、一个后驱指针Post。可以不分配必须的内存大小方便的进行添加和删除操作。使用的是非连续的内存空间进行存储。
Insert erase O（1）
优点：(1) 不使用连续内存完成动态操作。
(2) 在内部方便的进行插入和删除操作
(3) 可在两端进行push、pop
缺点：(1) 不能进行内部的随机访问，即不支持[ ]操作符和vector.at()
(2) 相对于verctor占用内存多
使用区别：
（1）如果你需要高效的随即存取，而不在乎插入和删除的效率，使用vector
（2）如果你需要大量的插入和删除，而不关心随机存取，则应使用list
（3）如果你需要随机存取，而且关心两端数据的插入和删除，则应使用deque

vector VS. list VS. deque：
a、若需要随机访问操作，则选择vector；
b、若已经知道需要存储元素的数目，则选择vector；
c、若需要随机插入/删除（不仅仅在两端），则选择list
d、只有需要在首端进行插入/删除操作的时候，还要兼顾随机访问效率，才选择deque，否则都选择vector。
e、若既需要随机插入/删除，又需要随机访问，则需要在vector与list间做个折中-deque。

f、当要存储的是大型负责类对象时，list要优于vector；当然这时候也可以用vector来存储指向对象的指针，
同样会取得较高的效率，但是指针的维护非常容易出错，因此不推荐使用。

Free和delete区别
free释放内存的和delete可以说是两套代码，它们的逻辑不同，不能混用。用new申请的就yu要用delete翻译，用malloc申请的就要用free释放。

free 只是告诉操作系统回收recycle内存，而delete会先调用类的析构函数，然后才告诉操作系统回收内存。

首先，C++中的explicit关键字只能用于修饰只有一个参数的类构造函数，它的作用是表明该构造函数是显示的，而非隐式的，跟它相对应的另一个关键字是implicit，意思是隐藏的,类构造函数默认情况下即声明为implicit(隐式).
Implicit type conversion

递归
优点：

1. 简洁

2.在树的前序，中序，后序遍历算法中，递归的实现明显要比循环简单得多。

缺点：

1.递归由于是函数调用自身，而函数调用是有时间和空间的消耗的：每一次函数调用，都需要在内存栈中分配空间以保存参数、返回地址以及临时变量，而往栈中压入数据和弹出数据都需要时间。->效率
2.递归中很多计算都是重复的，由于其本质是把一个问题分解成两个或者多个小问题，多个小问题存在相互重叠的部分，则存在重复计算，如fibonacci斐波那契数列的递归实现。->效率

3.调用栈可能会溢出，其实每一次函数调用会在内存栈中分配空间，而每个进程的栈的容量是有限的，当调用的层次太多时，就会超出栈的容量，从而导致栈溢出。->性能

size_t是一种数据相关的无符号类型，它被设计得足够大以便能够内存中任意对象的大小在C++中，设计 size_t 就是为了适应多个平台的 。size_t的引入增强了程序在不同平台上的可移植性。

动态规划和贪心算法都是一种递推算法 均用局部最优解来推导全局最优解
不同点：
贪心算法：
1.贪心算法中，作出的每步贪心决策都无法改变，因为贪心策略是由上一步的最优解推导下一步的最优解，而上一部之前的最优解则不作保留。
2.由（1）中的介绍，可以知道贪心法正确的条件是：每一步的最优解一定包含上一步的最优解。

动态规划算法：
1.全局最优解中一定包含某个局部最优解，但不一定包含前一个局部最优解，因此需要记录之前的所有最优解
2.动态规划的关键是状态转移方程，即如何由以求出的局部最优解来推导全局最优解
3.边界条件：即最简单的，可以直接得出的局部最优解

BST
Algorithm Average Worst **case**
Space O(n) O(n)
Search O(log n) O(n)

Insert O(log n) O(n)
Delete O(log n) O(n)
The major advantage of binary search trees over other data structures is that the related sorting algorithms **and** search algorithms such as in-order traversal can be very efficient

快排
nlog(n)
Sort the array **using** the dichotomy,
in each recursion, we choose a certain value, **and**
put the value smaller than the certain value to the left **and** larger ons in the right

满二叉树 full binary tree
a binary tree T is full if each node is either a leaf or possesses exactly two child nodes.
2^k-1 k is height
which all interior nodes have two children

Perfect binary tree
A perfect binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.
A perfect binary tree has 2n+1-1 nodes, n is height
若设二叉树的深度为h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。
In each layer except the most bottom layer, the number of nodes will reach the maximum number. And in the most bottom layer, all nodes are consecutive gathered from left to right.

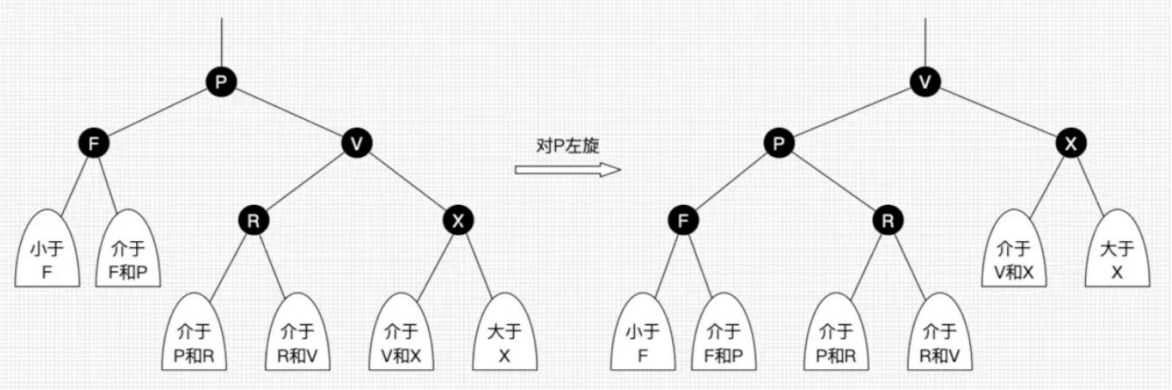
平衡二叉树avl -> red-black tree -> insert search erase is logn
Empty tree
The difference between left subtree's height and right subtree's height is not large 1.
It is an Absolute value
Also it left subtree and right subtree both are balance binary tree
Rotate clockwise anticlockwise
Rotate clockwise from left to right, so left child node will become root node and cur root node will become right child node

Also left node less than root right node larger than root value.

B-tree Balance tree
Multipe tree
Have large more than 2 search path.

红黑树（Red Black Tree） 是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。红黑树和AVL树类似，都是在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。
红黑树能自平衡

- **左旋**：以某个结点作为支点(旋转结点)，其右子结点变为旋转结点的父结点，右子结点的左子结点变为旋转结点的右子结点，左子结点保持不变。如图3。
- **右旋**：以某个结点作为支点(旋转结点)，其左子结点变为旋转结点的父结点，左子结点的右子结点变为旋转结点的左子结点，右子结点保持不变。如图4。
- **变色**：结点的颜色由红变黑或由黑变红。



左旋只影响旋转结点和其**右子树**的结构，把右子树的结点往左子树挪了。

右旋只影响旋转结点和其**左子树**的结构，把左子树的结点往右子树挪了。

所以旋转操作是**局部**的。另外可以看出旋转能保持红黑树平衡的一些端详了：当一边子树的结点少了，那么向另外一边子树“借”一些结点；当一边子树的结点多了，那么向另外一边子树“租”一些结点。

红黑树插入：插入操作包括两部分工作：一查找插入的位置；二插入后自平衡。查找插入的父结点很简单。

除最后一层无任何子节点外，每一层上的所有结点都有两个子结点的二叉树。

完全二叉树是效率很高的数据结构，完全二叉树是由满二叉树而引出来的。对于深度为K的，有n个结点的二叉树，当且仅当其每一个结点都与深度为K的满二叉树中编号从1至n的结点一一对应时称之为完全二叉树。

满二叉树

该节点要么是叶子结点，要么是有两个字节点的节点。

完全二叉树

对于一颗二叉树，假设其深度为d（d>1）。除第d层外的所有节点构成满二叉树，且第d层所有节点从左向右连续地紧密排列，这样的二叉树被称为完全二叉树；

平衡二叉树（AVL）

它是一 棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树，同时，平衡二叉树必定是二叉搜索树。

红黑树
红黑树是一颗特殊的二叉查找树，除了二叉查找树的要求外，它还具有以下特性：

每个节点或者是黑色，或者是红色。
根节点是黑色。
每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]
如果一个节点是红色的，则它的子节点必须是黑色的。
从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

B树
B树和平衡二叉树稍有不同的是B树属于多叉树又名平衡多路查找树（查找路径不只两个），数据库索引技术里大量使用者B树和B+树的数据结构

B+

B+跟B树不同B+树的非叶子节点不保存关键字记录的指针，只进行数据索引，这样使得B+树每个非叶子节点所能保存的关键字大大增加；

B+树叶子节点保存了父节点的所有关键字记录的指针，所有数据地址必须要到叶子节点才能获取到。所以每次数据查询的次数都一样；

B+树叶子节点的关键字从小到大有序排列，左边结尾数据都会保存右边节点开始数据的指针。

https://zhuanlan.zhihu.com/p/27700617

网络

1. TCP 三次握手

第一次握手：建立连接时，客户端发送syn包（syn=j）到服务器，并进入SYN_SENT状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）
第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN_RECV状态；
第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=k+1），此包发送完毕，客户端和服务器进入ESTABLISHED（TCP连接成功）状态，完成三次握手。

2.“三次握手”的目的是为了解决“网络中存在延迟的重复分组”的问题

client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。

3.TCP四次挥手

1.客户端进程发出连接释放报文，并且停止发送数据。FIN=1。客户端进入FIN-WAIT-1（终止等待1）状态。

2.服务器收到连接释放报文，发出确认报文，此时，服务端就进入了CLOSE-WAIT（关闭等待）状态。TCP服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个CLOSE-WAIT状态持续的时间。

3.客户端收到服务器的确认请求后，此时，客户端就进入FIN-WAIT-2（终止等待2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。

4. 服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，此时，服务器就进入了LAST-ACK（最后确认）状态，等待客户端的确认。

5.客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，客户端就进入了TIME-WAIT（时间等待）状态。注意此时TCP连接还没有释放，必须经过2**MSL（最长报文段寿命）的时间后，当客户端撤销相应的TCB后，才进入CLOSED状态。

6.服务器只要收到了客户端发出的确认，立即进入CLOSED状态。同样，撤销TCB后，就结束了这次的TCP连接。可以看到，服务器结束TCP连接的时间要比客户端早一些。

4.为什么连接的时候是三次握手，关闭的时候却是四次握手？

答：因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

https://blog.csdn.net/qq_38950316/article/details/81087809

5.TCP超时重传

原理是在发送某一个数据以后就开启一个计时器，在一定时间内如果没有得到发送的数据报的ACK报文，那么就重新发送数据，直到发送成功为止。影响超时重传机制协议效率的一个关键参数是重传超时时间（RTO，Retransmission TimeOut）。RTO的值被设置过大过小都会对协议造成不利影响。

（1）RTO设长了，重发就慢，没有效率，性能差。

（2）RTO设短了，重发的就快，会增加网络拥塞，导致更多的超时，更多的超时导致更多的重发。 连接往返时间（RTT，Round Trip Time），指发送端从发送TCP包开始到接收它的立即响应所消耗的时间。

6.TCP滑动窗口

TCP的滑动窗口的可靠性也是建立在“确认重传”基础上的。
发送窗口只有收到对端对于本端发送窗口内字节的ACK确认，才会移动发送窗口的左边界。 接收端可以根据自己的状况通告窗口大小，从而控制发送端的接收，进行流量控制。滑动窗口协议是传输层进行流控的一种措施，接收方通过通告发送方自己的窗口大小，从而控制发送方的发送速度，从而达到防止发送方发送速度过快而导致自己被淹没的目的。拥塞窗口是发送方使用的流量控制，而滑动窗口则是接收方使用的流量控制。

7.TCP的拥塞控制

拥塞控制是一个全局性的过程； 流量控制是点对点通信量的控制TCP拥塞控制4个核心算法：慢开始（slow start）、拥塞避免（Congestion Avoidance）、快速重传（fastretransmit）、快速回复（fast recovery）拥塞窗口（cwnd，congestion window），其大小取决于网络的拥塞程度，并且动态地在变化。

慢开始算法的思路就是，不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。
为了防止cwnd增长过大引起网络拥塞，还需设置一个慢开始门限ssthresh状态变量。
ssthresh的用法如下：当cwnd<ssthresh时，使用慢开始算法。
当cwnd>ssthresh时，改用拥塞避免算法。
当cwnd=ssthresh时，慢开始与拥塞避免算法任意
拥塞避免算法让拥塞窗口缓慢增长，即每经过一个往返时间RTT就把发送发的拥塞窗口cwnd加1，而不是加倍。无论是在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞，就把慢开始门限设置为出现拥塞时的发送窗口大小的一半。然后把拥塞窗口设置为1，执行慢开始算法。
快速重传(Fast retransmit)要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方），而不要等到自己发送数据时捎

带确认。快重传算法规定，发送方只要一连收到3个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计数器时间到期。

快速恢复(Fast Recovery)

（1）当发送方连续收到三个重复确认，就执行“乘法减小”算法，把慢开始门限ssthresh减半。这是为了预防网络发生拥塞。请注意：接下去不执行慢开始算法。

（2）由于发送方现在认为网络很可能没有发生拥塞，因此与慢开始不同之处是现在不执行慢开始算法（即拥塞窗口cwnd现在不设置为1），而是把cwnd值设置为慢开始门限ssthresh减半后的数值，然后开始执行拥塞避免算法（“加法增大”），使拥塞窗口缓慢地线性增大。

“乘法减小”指的是无论是在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞，就把慢开始门限ssthresh设置为出现拥塞时的发送窗口大小的一半，并执行慢开始算法（指数增长），所以当网络频繁出现拥塞时，ssthresh下降的很快，以大大减少注入到网络中的分组数。“加法增大”是指执行拥塞避免算法后，使拥塞窗口缓慢增大，以防止过早出现拥塞。常合起来成为AIMD算法。

8.TCP UDP HTTP

物理层-数据链路-网络（IP协议）传输 （TCP协议）会话-表示层和应用层（HTTP协议）

TCP

可靠，稳定

TCP的可靠体现在TCP在传递数据之前，会有三次握手来建立连接，而且在数据传递时，有确认、窗口、重传、拥塞控制机制，在数据传完后，还会断开连接用来节约系统资源。

TCP的缺点：

慢，效率低，占用系统资源高，易被攻击

TCP在传递数据之前，要先建连接，这会消耗时间，而且在数据传递时，确认机制、重传机制、拥塞控制机制等都会消耗大量的时间，而且要在每台设备上维护所有的传输连接，事实上，每个连接都会占用系统的CPU、内存等硬件资源。

而且，因为TCP有确认机制、三次握手机制，这些也导致TCP容易被人利用，实现DOS、DDOS、CC等攻击。

UDP

快，比TCP稍安全

UDP没有TCP的握手、确认、窗口、重传、拥塞控制等机制，UDP是一个无状态的传输协议，所以它在传递数据时非常快。没有TCP的这些机制，UDP较TCP被攻击者利用的漏洞就要少一些。但UDP也是无法避免攻击的，比如：UDP Flood攻击……

UDP的缺点：

不可靠，不稳定

因为UDP没有TCP那些可靠的机制，在数据传递时，如果网络质量不好，就会很容易丢包。
UDP 用户数据报协议，是一个面向无连接的协议。采用该协议不需要两个应用程序先建立连接。UDP协议不提供差错恢复，不能提供数据重传，因此该协议传输数据安全性差。
（1）、TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接
（2）、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付

SYN攻击原理
SYN攻击属于DOS攻击的一种，它利用TCP协议缺陷，通过发送大量的半连接请求，耗费CPU和内存资源。SYN攻击除了能影响主机外，还可以危害路由器、防火墙等网络系统，事实上SYN攻击并不管目标是什么系统，只要这些系统打开TCP服务就可以实施。从上图可看到，服务器接收到连接请求（syn=j），将此信息加入未连接队列，并发送请求包给客户（syn=k,ack=j+1），此时进入SYN_RECV状态。当服务器未收到客户端的确认包时，重发请求包，一直到超时，才将此条目从未连接队列删除。配合IP欺骗，SYN攻击能达到很好的效果，通常，客户端在短时间内伪造大量不存在的IP地址，向服务器不断地发送syn包，服务器回复确认包，并等待客户的确认，由于源地址是不存在的，
服务器需要不断的重发直至超时，这些伪造的SYN包将长时间占用未连接队列，正常的SYN请求被丢弃，目标系统运行缓慢，严重者引起网络堵塞甚至系统瘫痪。

9.HTTP各个版本区别
HTTP连接最显著的特点是客户端发送的每次请求都需要服务器回送响应，在请求结束后，会主动释放连接。从建立连接到关闭连接的过程称为“一次连接”
1）在HTTP 1.0中，客户端的每次请求都要求建立一次单独的连接，在处理完本次请求后，就自动释放连接。
2）在HTTP 1.1中则可以在一次连接中处理多个请求，并且多个请求可以重叠进行，不需要等待一个请求结束后再发送下一个请求。
3）HTTP 2通过支持请求与相应的多路重用来减少延迟，通过压缩HTTP头字段将协议开销降到最低，同时增加了对请求优先级和服务器端推送的支持。HTTP/2有三大特性：头部压缩（原因：HTTP 请求和响应都是由「状态行、请求 / 响应头部、消息主体」三部分组成。一般而言，消息主体都会经过 gzip 压缩，或者本身传输的就是压缩过后的二进制文件（例如图片、音频），但状态行和头部却没有经过任何压缩，直接以纯文本传输。随着 Web 功能越来越复杂，每个页面产生的请求数也越来越多，导致头部会很长）、Server Push、多路复用。Server Push就是在某次流中，可以返回客户端并没有主动要的数据。

由于HTTP在每次请求结束后都会主动释放连接，因此HTTP连接是一种“短连接”，要保持客户端程序的在线状态，需要不断地向服务器发起连接请求。通常的做法是即时

不需要获得任何数据，客户端也保持每隔一段固定的时间向服务器发送一次“保持连接”的请求，服务器在收到该请求后对客户端进行回复，表明知道客户端“在线”。若服务器长时间无法收到客户端的请求，则认为客户端“下线”，若客户端长时间无法收到服务器的回复，则认为网络已经断开（http1.0）。
Keep-Alive解决的核心问题：一定时间内，同一域名多次请求数据，只建立一次HTTP请求，其他请求可复用每一次建立的连接通道，以达到提高请求效率的问题。在HTTP1.1中是默认开启了Keep-Alive，他解决了多次连接的问题，但是依然有两个效率上的问题：
1.串行的文件传输。当请求a文件时，b文件只能等待，等待a连接到服务器、服务器处理文件、服务器返回文件，这三个步骤。
2.连接数过多，就需要等待前面某个请求处理完成。
Http2的多路复用，HTTP/2的多路复用就是为了解决上述的两个性能问题。
1. 解决第一个：在HTTP1.1的协议中，我们传输的request和response都是基本于文本的，这样就会引发一个问题：所有的数据必须按顺序传输，比如需要传输：hello world，只能从h到d一个一个的传输，不能并行传输，因为接收端并不知道这些字符的顺序，所以并行传输在HTTP1.1是不能实现的。HTTP/2引入二进制数据帧和流的概念，其中帧对数据进行顺序标识。这样浏览器收到数据之后，就可以按照序列对数据进行合并，而不会出现合并后数据错乱的情况。同样是因为有了序列，服务器就可以并行的传输数据，这就是流所做的事情。
2. HTTP/2对同一域名下所有请求都是基于流，也就是说同一域名不管访问多少文件，也只建立一路连接。同样Apache的最大连接数为300，因为有了这个新特性，最大的并发就可以提升到300，比原来提升了6倍！

HTTPS：是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。HTTPS协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

10.Http请求方式，请求头
GET请求会显示请求指定的资源。一般来说GET方法应该只用于数据的读取
POST请求会 向指定资源提交数据，请求服务器进行处理，如：表单数据提交
HEAD方法与GET方法一样，都是向服务器发出指定资源的请求。但是，服务器在响应HEAD请求时不会回传资源的内容部分，即：响应主体。这样，我们可以不传输全部内容的情况下，就可以获取服务器的响应头信息。HEAD方法常被用于客户端查看服务器的性能
PUT请求会身向指定资源位置上传其最新内容，PUT方法是幂等的方法。通过该方法客户端可以将指定资源的最新数据传送给服务器取代指定的资源的内容。

11.HTTPS和HTTP的区别主要如下：

1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。服务端发送证书，也就是公钥私钥中的公钥给客户端，客户端通过该公钥计算得到在本次通信中才使用的私钥，传递给服务端，服务端如果能正常解码该消息，则之后的通信都是用该次通信对应的密钥进行

2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

12.SOCKET

套接字（socket）是通信的基石，是支持TCP/IP协议的网络通信的基本操作单元。包含进行网络通信必须的五种信息：连接使用的协议，本地主机的IP地址，本地进程的协议端口，远地主机的IP地址，远地进程的协议端口。

应用层通过传输层进行数据通信时，TCP会遇到同时为多个应用程序进程提供并发服务的问题。多个TCP连接或多个应用程序进程可能需要通过同一个 TCP协议端口传输数据。为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与TCP／IP协议交互提供了套接字(Socket)接口。**应用层可以和传输层通过Socket接口，区分来自不同应用程序进程或网络连接的通信**，实现数据传输的并发服务。

建立socket连接

建立Socket连接至少需要一对套接字，其中一个运行于客户端，称为ClientSocket，另一个运行于服务器端，称为ServerSocket 。

套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

服务器监听：服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求。

客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

连接确认：当服务器端套接字监听到或者说接收到客户端套接字的连接请求时，就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，双方就正式建立连接。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

socket通信流程

服务器根据地址类型（ipv4,ipv6）、socket类型、协议创建socket

服务器为socket绑定ip地址和端口号
服务器socket监听端口号请求，随时准备接收客户端发来的连接，这时候服务器的socket并没有被打开
客户端创建socket
客户端打开socket，根据服务器ip地址和端口号试图连接服务器socket
服务器socket接收到客户端socket请求，被动打开，开始接收客户端请求，直到客户端返回连接信息。这时候socket进入阻塞状态，所谓阻塞即accept()方法一直到客户端返回连接信息后才返回，开始接收下一个客户端谅解请求
客户端连接成功，向服务器发送连接状态信息
服务器accept方法返回，连接成功
客户端向socket写入信息
服务器读取信息
客户端关闭
服务器端关闭

Socket连接与HTTP连接
由于通常情况下Socket连接就是TCP连接，因此Socket连接一旦建立，通信双方即可开始相互发送数据内容，直到双方连接断开。但在实际网络应用中，客户端到服务器之间的通信往往需要穿越多个中间节点，例如路由器、网关、防火墙等，大部分防火墙默认会关闭长时间处于非活跃状态的连接而导致 Socket 连接断连，因此需要通过轮询告诉网络，该连接处于活跃状态。
而HTTP连接使用的是“请求—响应”的方式，不仅在请求时需要先建立连接，而且需要客户端向服务器发出请求后，服务器端才能回复数据。

很多情况下，需要服务器端主动向客户端推送数据，保持客户端与服务器数据的实时与同步。此时若双方建立的是Socket连接，服务器就可以直接将数据传送给客户端；若双方建立的是HTTP连接，则服务器需要等到客户端发送一次请求后才能将数据传回给客户端，因此，客户端定时向服务器端发送连接请求，不仅可以保持在线，同时也是在“询问”服务器是否有新的数据，如果有就将数据传给客户端。

13. Cookie and Session，Token

Cookies是服务器在本地机器上存储的小段文本并随每一个请求发送至同一个服务器cookie的内容主要包括：名字，值，过期时间，路径和域。路径与域一起构成cookie的作用范围。若不设置过期时间，则表示这个cookie的生命期为浏览器会话期间，关闭浏览器窗口，cookie就消失。这种生命期为浏览器会话期的cookie被称为会话cookie。会话cookie一般不存储在硬盘上而是保存在内存里，当然这种行为并不是规

范规定的。若设置了过期时间，浏览器就会把cookie保存到硬盘上，关闭后再次打开浏览器，这些cookie仍然有效直到超过设定的过期时间。存储在硬盘上的cookie可以在不同的浏览器进程间共享，比如两个IE窗口。

Session机制是一种服务器端的机制，服务器使用一种类似于散列表的结构（也可能就是使用散列表）来保存信息。

区别
1.存取方式的不同，Cookie中只能保管ASCII字符串，而Session中能够存取任何类型的数据，包括而不限于String、Integer、List、Map等。
2.隐私策略的不同，Cookie存储在客户端阅读器中，对客户端是可见的，而Session存储在服务器上，对客户端是透明的。假如选用Cookie，比较好的方法是，敏感的信息如账号密码等尽量不要写到Cookie中。最好是像Google、Baidu那样将Cookie信息加密，提交到服务器后再进行解密，保证Cookie中的信息只要本人能读得懂。
3.有效期上的不同，Cookie可以方便的设置。Session依赖于名为JSESSIONID的Cookie，而且假如设置Session的超时时间过长，服务器累计的Session就会越多，越容易招致内存溢出。
4.服务器压力的不同，Session是保管在服务器端的，每个用户都会产生一个Session，耗费大量的内存。而Cookie保管在客户端，不占用服务器资源
5.跨域支持上的不同，Cookie支持跨域名访问，而Session则不会支持跨域名访问。Session仅在他所在的域名内有效。

Token：

token是用户身份的验证方式，我们通常叫它：令牌。最简单的token组成:uid(用户唯一的身份标识)、time(当前时间的时间戳)、sign(签名，由token的前几位+盐以哈希算法压缩成一定长的十六进制字符串，可以防止恶意第三方拼接token请求服务器)。还可以把不变的参数也放进token，避免多次查库。

应用场景：

A：当用户首次登录成功（注册也是一种可以适用的场景）之后, 服务器端就会生成一个 token 值，这个值，会在服务器保存token值(保存在数据库中)，再将这个token值返回给客户端.

B：客户端拿到 token 值之后,进行本地保存。（SP存储是大家能够比较支持和易于理解操作的存储）

C：当客户端再次发送网络请求(一般不是登录请求)的时候,就会将这个 token 值附带到参数中发送给服务器.

D：服务器接收到客户端的请求之后,会取出token值与保存在本地(数据库)中的token值做对比

对比一：如果两个 token 值相同， 说明用户登录成功过!当前用户处于登录状态!

对比二：如果没有这个 token 值, 则说明没有登录成功.

对比三：如果 token 值不同: 说明原来的登录信息已经失效,让用户重新登录.

session和 token并不矛盾，作为身份认证token安全性比session好，因为每个请求都有签名还能防止监听以及重放攻击，而session就必须靠链路层来保障通讯安全了

14. OSI七层模型

OSI七层协议模型主要是：应用层（Application）、表示层（Presentation）、会话层（Session）、传输层（Transport）、网络层（Network）、数据链路层（Data Link）、物理层（Physical）。

应用层 文件传输，电子邮件，文件服务，虚拟终端 TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet

表示层 数据格式化，代码转换，数据加密 没有协议

会话层 解除或建立与别的接点的联系 没有协议

传输层 提供端对端的接口 TCP，UDP

网络层 为数据包选择路由 IP，ICMP，RIP，OSPF，BGP，IGMP

数据链路层 传输有地址的帧以及错误检测功能 SLIP，CSLIP，PPP，ARP，RARP，MTU

物理层 以二进制数据形式在物理媒体上传输数据 ISO2110，IEEE802，IEEE802.2

在OSI七层模型中ARP属于数据链路层， 在TCP/IP模型中ARP属于网络层

五层体系结构

五层体系结构包括：应用层、运输层、网络层、数据链路层和物理层。

网际层协议：包括：IP协议、ICMP协议、ARP协议、RARP协议。
传输层协议：TCP协议、UDP协议。
应用层协议：FTP、Telnet、SMTP、HTTP、RIP、NFS、DNS

ARP协议，全称“Address Resolution Protocol”,中文名是地址解析协议，使用ARP协议可实现通过IP地址获得对应主机的物理地址（MAC地址）
ICMP：一个新搭建好的网络，往往需要先进行一个简单的测试，来验证网络是否畅通；但是IP协议并不提供可靠传输。如果丢包了，IP协议并不能通知传输层是否丢包以及丢包的原因。所以我们就需要一种协议来完成这样的功能–ICMP协议。

15. HTTP过程

1.对www.baidu.com这个网址进行DNS域名解析(将主机名和域名转换为IP地址)，得到对应的IP地址
2.根据这个IP，找到对应的服务器，发起TCP的三次握手
3.建立TCP连接后发起HTTP请求
4.服务器响应HTTP请求，浏览器得到html代码。Responds
5.浏览器解析html代码，并请求html代码中的资源（如js、css图片等）（先得到html代码，才能去找这些资源）parsing
6.浏览器对页面进行渲染呈现给用户

HTTP请求过程
建立连接完毕以后客户端会发送响应给服务端
服务端接受请求并且做出响应发送给客户端
客户端收到响应并且解析响应给用户

HTTPS过程
客户端发送请求到服务器端
服务器端返回证书和公开密钥，公开密钥作为证书的一部分而存在
客户端验证证书和公开密钥的有效性，如果有效，则生成共享密钥并使用公开密钥加密发送到服务器端
服务器端使用私有密钥解密数据，并使用收到的共享密钥加密数据，发送到客户端
客户端使用共享密钥解密数据
SSL加密建立……

DNS过程
1.用户主机上运行着DNS的客户端，就是我们的PC机或者手机客户端运行着DNS客户端了
2.浏览器将接收到的url中抽取出域名字段，就是访问的主机名，，并将这个主机名传送给DNS应用的客户端。
3.DNS客户机端向DNS服务器端发送一份查询报文，报文中包含着要访问的主机名字段（中间包括一些列缓存查询以及分布式DNS集群的工作）
4.该DNS客户机最终会收到一份回答报文，其中包含有该主机名对应的IP地址。
5.一旦该浏览器收到来自DNS的IP地址，就可以向该IP地址定位的HTTP服务器发起TCP连接。

16.TCP流量控制、拥塞控制

什么是流量控制？流量控制的目的？

如果发送者发送数据过快，接收者来不及接收，那么就会有分组丢失。为了避免分组丢失，控制发送者的发送速度，使得接收者来得及接收，这就是流量控制。流量控制根本目的是防止分组丢失，它是构成TCP可靠性的一方面。

如何实现流量控制？

由滑动窗口协议（连续ARQ协议）实现。滑动窗口协议既保证了分组无差错、有序接收，也实现了流量控制。主要的方式就是接收方返回的 ACK 中会包含自己的接收窗口的大小，并且利用大小来控制发送方的数据发送。

流量控制引发的死锁？怎么避免死锁的发生？

当发送者收到了一个窗口为0的应答，发送者便停止发送，等待接收者的下一个应答。但是如果这个窗口不为0的应答在传输过程丢失，发送者一直等待下去，而接收者以为发送者已经收到该应答，等待接收新数据，这样双方就相互等待，从而产生死锁。
为了避免流量控制引发的死锁，TCP使用了持续计时器。每当发送者收到一个零窗口的应答后就启动该计时器。时间一到便主动发送报文询问接收者的窗口大小。若接收者仍然返回零窗口，则重置该计时器继续等待；若窗口不为0，则表示应答报文丢失了，此时重置发送窗口后开始发送，这样就避免了死锁的产生。

拥塞控制和流量控制的区别
拥塞控制：拥塞控制是作用于网络的，它是防止过多的数据注入到网络中，避免出现网络负载过大的情况；常用的方法就是：（1）慢开始、拥塞避免（2）快重传、快恢复。

流量控制：流量控制是作用于接收者的，它是控制发送者的发送速度从而使接收者来得及接收，防止分组丢失的。

17.HTTP的报文格式
HTTP报文是面向文本的，报文中的每一个字段都是一些ASCII码串，各个字段的长度是不确定的。HTTP有两类报文：请求报文和响应报文。
请求报文
一个HTTP请求报文由请求行（request line）、请求头部（header）、空行和请求数据4个部分组成。
请求行，请求行由请求方法字段、URL字段和HTTP协议版本字段3个字段组成，它们用空格分隔。例如，GET /index.html HTTP/1.1。
请求头部，请求头部由关键字/值对组成，每行一对，关键字和值用英文冒号“:”分隔。请求头部通知服务器有关于客户端请求的信息，典型的请求头有：
空行，最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器以下不再有请求头。对于一个完整的http请求来说空行是必须的，否则服务器会认为本次请求的数据尚未完全发送到服务器，处于等待状态。
请求数据，请求数据不在GET方法中使用，而是在POST方法中使用。

Http中 header主要来存放cookie，token等信息的 body主要用来存放post的一些数据

语言

Overload 重载

在C++程序中，可以将语义、功能相似的几个函数用同一个名字表示，但参数不同（包括类型、顺序不同），即函数重载。
（1）相同的范围（在同一个类中）；
（2）函数名字相同；
（3）参数不同；

请注意，重载解析中不考虑返回类型，而且在不同的作用域里声明的函数也不算是重载。重载可以理解为一个类内部的函数重载，较好理解，此处不举例。

Override 覆盖

是指派生类函数覆盖基类函数，特征是：
（1）不同的范围（分别位于派生类与基类）；
（2）函数名字相同；
（3）参数相同；
（4）基类函数必须有virtual 关键字。

Overwrite(重写)：是指派生类的函数屏蔽了与其同名的基类函数，规则如下：
（1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual关键字，基类的函数将被隐藏（注意别与重载混淆）。

（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

Polymorphism
不能通过基类的指针 调用 派生类特有的方法
多态绑定：编译时绑定(compile-time Binding)：函数重载 操作符重载

动态绑定(Runtime Binding): 以封装和继承为基础：
必须使用virtual function (虚函数)
只有在继承下才有虚函数
必须是 public 继承
声明在 父类的方法上
声明: virtual 函数原型;

多态中存在的问题
内存泄漏:
在父类指针操作子类对象的成员时, 不会出现问题, 但delete时,会出现问题:
父类指针的delete释放资源, 调用的是父类的析构函数, 子类的析构函数却没有调用. 这会导致内存泄漏（memory leak）
解决方法
调用子类的的delete, 即delete 子类指针, 此时子类和父类的析构函数都调用了
析构虚函数
这样父类指针指向的是哪个对象，哪个对象的构造函数就会先执行，然后执行父类的构造函数。销毁的时候子类的析构函数也会执行(子类虚构函数会调用父类虚构函数)

缓存的优缺点：
缓存可以减少访问内存数据的时间，减少了对数据库的读操作，数据库的压力降低。但是需要为缓存开辟额外的存储空间。因为内存断电就清空数据，存放到内存中的数据可能丢失

Hash bucket 怎么增长
几种哈希冲突解confilct决方法 承载因子load factor：
1.找下一个空桶,
2. 再哈希再哈希法又叫双哈希法，有多个不同的Hash函数，当发生冲突时，使用第二个，第三个，….,
3. 建立公共溢出区：
这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表
4.链接法（拉链法）。将具有同一散列地址的记录存储在一条线性链表中

主要讲了四个参数，size大小,capacity容量,loadFactor装载因子和threshold临界值
size是指当前hashmap中数据的个数，capacity是hashmap当前最大容纳个数
hashmap有自动扩容机制，但是不是到达数据占满的时候才扩容的，在达到临界值*装载因子时就扩容extand storage
比如当前临界值是16，装载因子是0.75（通常都是0.75）的时候，当map中个数达到13就会扩容

Ajax 异步通信Asynchronous communication
Ajax 在浏览器与 Web 服务器之间使用异步数据传输（HTTP 请求），这样就可使网页从服务器请求少量的信息，而不是整个页面。

Ajax uses asynchronous data transfer between browser and web server (HTTP request)
Ajax可使因特网应用程序更小、更快，更友好

C++封装继承多态
面向对象的三个基本特征是：封装、继承、多态。其中，封装可以隐藏实现细节，使得代码模块化；继承可以扩展已存在的代码模块（类）；它们的目的都是为了——代码重用。而多态则是为了实现另一个目的——接口重用！
封装：在面向对象编程上可理解为：把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。
继承：继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。其继承的过程，就是从一般到特殊的过程。
多态性（polymorphisn）是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。

指针和引用的定义和性质区别
指针：指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名而已。
可以有const指针，但是没有const引用。指针可以有多级，但是引用只能是一级。指针的值可以为空，但是引用的值不能为NULL，并且引用在定义的时候必须初始化。
指针的值在初始化后可以改变，即指向其它的存储单元，而引用在进行初始化后就不会再改变了。
"sizeof引用"得到的是所指向的变量(对象)的大小，而"sizeof指针"得到的是指针本身的大小。

协程
协程（coroutine）跟具有操作系统概念的线程不一样，实际上协程就是类函数一样的程序组件，你可以在一个线程里面轻松创建数十万个协程，就像数十万次函数调用一样。只不过函数只有一个调用入口起始点，返回之后就结束了，而协程入口既可以是起始点，又可以从上一个返回点继续执行，也就是说协程之间可以通过 yield 方式转移执行权，对称（symmetric）、平级地调用对方，而不是像函数那样上下级调用关系。当然 协程也可以模拟函数那样实现上下级调用关系，这就叫非对称协程（asymmetric coroutines）。

- 非对称式协程，协程之间有调用链关系，一个协程A释放控制权有2种方式
- 通过调用yield，将控制权返还给协程A的创建协程
- 通过调用resume，将控制权交给一个子协程

- 对称式协程，与非对称式协程不同，各个协程之间可以互相转移控制权，类似于goto语句，这种方式，即使非常有经验的程序员也很难理清调用流程。同时该协程方式实现困难，性能不高。

Static 和 Const

const定义的常量在超出其作用域之后其空间会被释放，而**static定义的静态常量在函数执行后不会释放其存储空间**。

static静态成员变量不能在类的内部初始化，在类的内部只是声明，定义必须在类定义体的外部，通常在类的实现文件中初始化。const成员变量也不能在类定义处初始化，只能通过构造函数初始化列表进行，并且必须有构造函数。

静态数据成员被类的所有对象所共享，包括该类派生类的对象。即派生类对象与基类对象共享基类的静态数据成员。

静态数据成员可以成为成员函数的可选参数，**而普通数据成员则不可以。**

静态数据成员的类型可以是所属类的类型，而普通数据成员则不可以。普通数据成员的只能声明为所属类类型的指针或引用。

静态成员函数不可以调用类的非静态成员。

https://blog.csdn.net/u012864854/article/details/79777991

new 和 malloc

new从堆上分配内存，malloc从自由存储区上分配内存；new/delete会调用构造函数/析构函数对对象进行初始化与销毁。new可以认为是malloc加构造函数的执行。new出来的指针是直接带类型信息的。而malloc返回的都是void指针。

缓冲区溢出（栈溢出）

程序为了临时存取数据的需要，一般会分配一些内存空间称为缓冲区。如果向缓冲区中写入缓冲区无法容纳的数据，机会造成缓冲区以外的存储单元被改写，称为缓冲区溢出。而栈溢出是缓冲区溢出的一种，原理也是相同的。分为上溢出和下溢出。其中，上溢出是指栈满而又向其增加新的数据，导致数据溢出；下溢出是指空栈而又进行删除操作等，导致空间溢出。

类型安全以及C++中的类型转换

类型安全很大程度上可以等价于内存安全，类型安全的代码不会试图访问自己没被授权的内存区域。

static_cast <T*> (content) 静态转换.在编译期间处理，可以实现C++中内置基本数据类型之间的相互转换。用来强迫隐式转换如non-const对象转为const对象，编译时检查，用于非多态的转换

dynamic_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换（cross cast）。在类层次间进行上行转换时（父类->子类），dynamic_cast和static_cast的效果是一样的；在进行下行转换时，dynamic_cast具有类型检查的功能，比static_cast更安全。

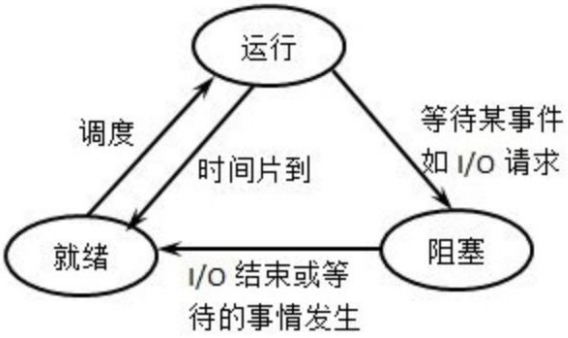
33.C++中的公有，私有，保护的问题?

	类	类对象	公有继承派生类	私有继承派生类	保护继承派生类	公有继承派生类对象	私有继承派生类对象	保护继承派生类对象
公有成员	√	√	√	√	√	√	X	X
私有成员	√	X	X	X	X	X	X	X
保护成员	√	X	√	√	√	X	X	X

操作系统

1.进程的常见状态？以及各种状态之间的转换条件？

- 就绪：进程已处于准备好运行的状态，即进程已分配到除CPU外的所有必要资源后，只要再获得CPU，便可立即执行。
- 执行：进程已经获得CPU，程序正在执行状态。
- 阻塞：正在执行的进程由于发生某事件（如I/O请求、申请缓冲区失败等）暂时无法继续执行的状态。



2.进程同步

进程同步的主要任务：是对多个相关进程在执行次序上进行协调，以使并发执行的诸进程之间能有效地共享资源和相互合作，从而使程序的执行具有可再现性。

同步机制遵循的原则：

（1）空闲让进；

（2）忙则等待（保证对**临界区**的互斥访问）；

（3）有限等待（有限代表有限的时间，避免死等）；

（4）让权等待，（当进程不能进入自己的临界区时，应该释放处理机，以免陷入忙等状态）。

3. 进程同步

线程同步是指多线程通过特定的设置（如互斥量，事件对象，临界区）来控制线程之间的执行顺序（即所谓的同步）也可以说是在线程之间通过同步建立起执行顺序的关系。

界区（Critical Section）、互斥对象（Mutex）：主要用于互斥控制；都具有拥有权的控制方法，只有拥有该对象的线程才能执行任务，所以拥有，执行完任务后一定要释放该对象。

信号量（Semaphore）、事件对象（Event）：事件对象是以通知的方式进行控制，主要用于同步控制！

临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。在任意时刻只允许一个线程对共享资源进行访问，如果有多个线程试图访问公共资源，那么在有一个线程进入后，其他试图访问公共资源的线程将被挂起，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。

互斥对象：互斥对象和临界区很像，采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程同时访问。

信号量：信号量也是内核对象。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目

事件对象：通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作

3.进程的通信方式有哪些？

在介绍进程的时候，我们提起过一个进程不能直接读写另一个进程的数据，两者之间的通信需要通过进程间通信(inter-process communication, IPC)进行。进程通信的方式通常遵从生产者消费者模型，需要实现数据交换和同步两大功能。

1) Shared-memory + semaphore

不同进程通过读写操作系统中特殊的共享内存进行数据交换，进程之间用semaphore实现同步。

2) Message passing

进程在操作系统内部注册一个port，并且监测有没有数据，其他进程直接写数据到该port。该通信方式更加接近于网络通信方式。事实上，网络通信也是一种IPC，只是进程分布在不同机器上而已。

1. 管道：管道是单向的、先进先出的、无结构的、固定大小的字节流，它把一个进程的标准输出和另一个进程的标准输入连接在一起。写进程在管道的尾端写入数据，读进程在管道的道端读出数据。数据读出后将从管道中移走，其它读进程都不能再读到这些数据。管道提供了简单的流控制机制。进程试图读空管道时，在有数据写入管道前，进程将一直阻塞。同样地，管道已经满时，进程再试图写管道，在其它进程从管道中移走数据之前，写进程将一直阻塞。
    - 无名管道：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系（通常是指父子进程关系）的进程间使用。
    - 命名管道：命名管道也是半双工的通信方式，在文件系统中作为一个特殊的设备文件而存在，但是它允许无亲缘关系进程间的通信。当共享管道的进程执行完所有的I/O操作以后，命名管道将继续保存在文件系统中以便以后使用。
2. 信号量：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其它进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
3. 消息队列：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
4. 信号：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。
5. 共享内存：共享内存就是映射一段能被其它进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的IPC方式，它是针对其它进程间通信方式运行效率低而专门设计的。它往往与其它通信机制（如信号量）配合使用，来实现进程间的同步和通信。
6. 套接字：套接字也是一种进程间通信机制，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.死锁

在引入锁的同时，我们遇到了一个新的问题：死锁(Deadlock)。死锁是指两个或多个线程／进程之间相互阻塞，以至于任何一个都不能继续运行，因此也不能解锁其他线程／进程。例如，线程A占有lock A，并且尝试获取lock B；而线程2占有lock B，尝试获取lock A。此时，两者相互阻塞，都无法继续运行。

总结产生死锁的四个条件(只有当四个条件同时满足时才会产生死锁)：

1. Mutual Exclusion – Only one process may use a resource at a time
2. Hold-and-Wait – Process holds resource while waiting for another

3. No Preemption – Can't take a resource away from a process
4. Circular Wait – The waiting processes form a cycle

死锁条件如何处理

1. 互斥条件(Mutual exclusion)：资源不能被共享，只能由一个进程使用。
2. 请求与保持条件(Hold and wait)：已经得到资源的进程可以再次申请新的资源。
3. 非剥夺条件(No pre-emption)：已经分配的资源不能从相应的进程中被强制地剥夺。
4. 循环等待条件(Circular wait)：系统中若干进程组成环路，该环路中每个进程都在等待相邻进程正占用的资源。

如何处理死锁问题：

1. 忽略该问题。例如鸵鸟算法，该算法可以应用在极少发生死锁的的情况下。为什么叫鸵鸟算法呢，因为传说中鸵鸟看到危险就把头埋在地底下，可能鸵鸟觉得看不到危险也就没危险了吧。跟掩耳盗铃有点像。
2. 检测死锁并且恢复。
3. 仔细地对资源进行动态分配，以避免死锁。
4. 通过破除死锁四个必要条件之一，来防止死锁产生。

5.生产消费者

生产者消费者模型是一种常见的通信模型：生产者和消费者共享一个数据管道，生产者将数据写入buffer，消费者从另一头读取数据。对于数据管道，需要考虑为空和溢出的情况。同时，通常还需要将这部分共享内存用mutex加锁。在只有一个生产者一个消费者的情况下，可以设计无锁队列(lockless queue)，线程安全地直接读写数据。

生产者-消费者问题，实际上主要是包含了两类线程，一种是生产者线程用于生产数据，另一种是消费者线程用于消费数据，为了解耦生产者和消费者的关系，通常会采用共享的数据区域，就像是一个仓库，生产者生产数据之后直接放置在共享数据区中，并不需要关心消费者的行为；而消费者只需要从共享数据区中去获取数据，就不再需要关心生产者的行为。但是，这个共享数据区域中应该具备这样的线程间并发协作的功能：如果共享数据区已满的话，阻塞生产者继续生产数据放置入内；如果共享数据区为空的话，阻塞消费者继续消费数据；

5.进程与线程的区别

- 从概念上：

    - 进程：一个程序对一个数据集的动态执行过程，是分配资源的基本单位。
    - 线程：一个进程内的基本调度单位。线程的划分尺度小于进程，一个进程包含一个或者更多的线程。
- 从执行过程中来看：
    - 进程：拥有独立的内存单元，而多个线程共享内存，从而提高了应用程序的运行效率。
    - 线程：每一个独立的线程，都有一个程序运行的入口、顺序执行序列、和程序的出口。但是线程不能够独立的执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。线程依托于进程存在，在进程之下，可以共享进程的内存，而且还拥有一个属于自己的内存空间，这段内存空间也叫做线程栈，是在建立线程时由系统分配的，主要用来保存线程内部所使用的数据。
- 从逻辑角度来看（重要区别）：
    - 多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但是，操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理及资源分配。

线程执行开销小，但不利于资源的管理和保护；而进程正相反。

进程和线程

1.定义：

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

２.关系：

进程在执行过程中拥有独立的内存单元，而多个线程共享进程的内存。

一个线程可以创建和撤销另一个线程;同一个进程中的多个线程之间可以并发执行.

相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

并发

1.微观角度：所有的并发处理都有排队等候，唤醒，执行等这样的步骤，在微观上他们都是序列被处理的，如果是同一时刻到达的请求（或线程）也会根据优先级的不同，而先后进入队列排队等候执行。

2.宏观角度：多个几乎同时到达的请求（或线程）在宏观上看就像是同时在被处理。

并行

1. Parallelism，即并行，指两个或两个以上事件（或线程）在同一时刻发生，是真正意义上的不同事件或线程在同一时刻，在不同CPU资源呢上（多核），同时执行。

2.并行，不存在像并发那样竞争，等待的概念。

通过多线程实现并发，并行：

1. java中的Thread类定义了多线程，通过多线程可以实现并发或并行。
2.在CPU比较繁忙，资源不足的时候（开启了很多进程），操作系统只为一个含有多线程的进程分配仅有的CPU资源，这些线程就会为自己尽量多抢时间片，这就是通过多线程实现并发，线程之间会竞争CPU资源争取执行机会。
3.在CPU资源比较充足的时候，一个进程内的多线程，可以被分配到不同的CPU资源，这就是通过多线程实现并行。
4.至于多线程实现的是并发还是并行？上面所说，所写多线程可能被分配到一个CPU内核中执行，也可能被分配到不同CPU执行，分配过程是操作系统所为，不可人为控制。所有，如果有人问我我所写的多线程是并发还是并行的？我会说，都有可能。
5.不管并发还是并行，都提高了程序对CPU资源的利用率，最大限度地利用CPU资源。

6.进程/线程间同步机制
1.**临界区**（Critical Section）:通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。

优点：保证在某一时刻只有一个线程能访问数据的简便办法

缺点：虽然临界区同步速度很快，但却只能用来同步本进程内的线程，而不可用来同步多个进程中的线程。

2. **互斥量**（Mutex）:为协调共同对一个共享资源的单独访问而设计的。

互斥量跟临界区很相似，比临界区复杂，互斥对象只有一个，只有拥有互斥对象的线程才具有访问资源的权限。

优点：使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。

缺点：互斥量是可以命名的，也就是说它可以跨越进程使用，所以创建互斥量需要的资源更多，所以如果只为了在进程内部是用的话使用临界区会带来速度上的优势并能

够减少资源占用量。因为互斥量是跨进程的互斥量一旦被创建，就可以通过名字打开它。

3、信号量（Semaphore）:为控制一个具有有限数量用户资源而设计。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。互斥量是信号量的一种特殊情况，当信号量的最大资源数=1就是互斥量了。

优点：适用于对Socket（套接字）程序中线程的同步。（例如，网络上的HTTP服务器要对同一时间内访问同一页面的用户数加以限制，只有不大于设定的最大用户数目的线程能够进行访问，而其他的访问企图则被挂起，只有在有用户退出对此页面的访问后才有可能进入。）

缺点：①信号量机制必须有公共内存，不能用于分布式操作系统，这是它最大的弱点

②信号量机制功能强大，但使用时对信号量的操作分散， 而且难以控制，读写和维护都很困难，加重了程序员的编码负担；

③核心操作P-V分散在各用户程序的代码中，不易控制和管理，一旦错误，后果严重，且不易发现和纠正。

4、事件（Event）：用来通知线程有一些事件已发生，从而启动后继任务的开始。

优点：事件对象通过通知操作的方式来保持线程的同步，并且可以实现不同进程中的线程同步操作。

总结：

①临界区不是内核对象，只能用于进程内部的线程同步，是用户方式的同步。互斥、信号量是内核对象可以用于不同进程之间的线程同步（跨进程同步）。
②互斥其实是信号量的一种特殊形式。互斥可以保证在某一时刻只有一个线程可以拥有临界资源。信号量可以保证在某一时刻有指定数目的线程可以拥有临界资源。

7.调度算法：

FIFO或First Come, First Served (FCFS)先来先服务

- 调度的顺序就是任务到达就绪队列的顺序。
- 公平、简单(FIFO队列)、非抢占、不适合交互式。

- 未考虑任务特性，平均等待时间可以缩短。

Shortest Job First (SJF)

- 最短的作业(CPU区间长度最小)最先调度。
- SJF可以保证最小的平均等待时间。

Shortest Remaining Job First (SRJF)

- SJF的可抢占版本，比SJF更有优势。
- SJF(SRJF): 如何知道下一CPU区间大小？根据历史进行预测: 指数平均法。

优先权调度

- 每个任务关联一个优先权，调度优先权最高的任务。
- 注意：优先权太低的任务一直就绪，得不到运行，出现“饥饿”现象。

Round-Robin(RR)轮转调度算法

- 设置一个时间片，按时间片来轮转调度（“轮叫”算法）
- 优点: 定时有响应，等待时间较短；缺点: 上下文切换次数较多；
- 时间片太大，响应时间太长；吞吐量变小，周转时间变长；当时间片过长时，退化为FCFS。

8.页面置换算法

操作系统将内存按照页面进行管理，在需要的时候才把进程相应的部分调入内存。当产生缺页中断时，需要选择一个页面写入。如果要换出的页面在内存中被修改过，变成了“脏”页面，那就需要先写会到磁盘。页面置换算法，就是要选出最合适的一个页面，使得置换的效率最高。页面置换算法有很多，简单介绍几个，重点介绍比较重要的LRU及其实现算法。

一、最优页面置换算法

最理想的状态下，我们给页面做个标记，挑选一个最远才会被再次用到的页面调出。当然，这样的算法不可能实现，因为不确定一个页面在何时会被用到。

二、先进先出页面置换算法（FIFO）及其改进

这种算法的思想和队列是一样的，该算法总是淘汰最先进入内存的页面，即选择在内存中驻留时间最久的页面予淘汰。实现：把一个进程已调入内存的页面按先后次序链接成一个队列，并且设置一个指针总是指向最老的页面。缺点：对于有些经常被访问的页面如含有全局变量、常用函数、例程等的页面，不能保证这些不被淘汰。

三、最近最少使用页面置换算法LRU（Least Recently Used）

根据页面调入内存后的使用情况做出决策。LRU置换算法是选择最近最久未使用的页面进行淘汰。

1.为每个在内存中的页面配置一个移位寄存器。（P165）定时信号将每隔一段时间将寄存器右移一位。最小数值的寄存器对应页面就是最久未使用页面。

2.利用一个特殊的栈保存当前使用的各个页面的页面号。每当进程访问某页面时，便将该页面的页面号从栈中移出，将它压入栈顶。因此，栈顶永远是最新被访问的页面号，栈底是最近最久未被访问的页面号。

9.逻辑地址 Vs 物理地址 Vs 虚拟内存

- **所谓的逻辑地址，是指计算机用户(例如程序开发者)，看到的地址。**例如，当创建一个长度为100的整型数组时，操作系统返回一个逻辑上的连续空间：指针指向数组第一个元素的内存地址。由于整型元素的大小为4个字节，故第二个元素的地址时起始地址加4，以此类推。事实上，**逻辑地址并不一定是元素存储的真实地址，即数组元素的物理地址(在内存条中所处的位置)，并非是连续的，只是操作系统通过地址映射，将逻辑地址映射成连续的，这样更符合人们的直观思维**。
- 另一个重要概念是虚拟内存。操作系统读写内存的速度可以比读写磁盘的速度快几个量级。但是，内存价格也相对较高，不能大规模扩展。于是，**操作系统可以通过将部分不太常用的数据移出内存，“存放到价格相对较低的磁盘缓存，以实现内存扩展**。操作系统还可以通过算法预测哪部分存储到磁盘缓存的数据需要进行读写，提前把这部分数据读回内存。**虚拟内存空间相对磁盘而言要小很多，因此，即使搜索虚拟内存空间也比直接搜索磁盘要快。唯一慢于磁盘的可能是，内存、虚拟内存中都没有所需要的数据，最终还需要从硬盘中直接读取。**这就是为什么内存和虚拟内存中需要存储会被重复读写的数据，否则就失去了缓存的意义。现代计算机中有一个专门的**转译缓冲区(Translation Lookaside Buffer，TLB)**，用来实现虚拟地址到物理地址的快速转换。

10.同步和互斥的区别

当有多个线程的时候，经常需要去同步这些线程以访问同一个数据或资源。例如，假设有一个程序，其中一个线程用于把文件读到内存，而另一个线程用于统计文件中的字符数。当然，在把整个文件调入内存之前，统计它的计数是没有意义的。但是，由于每个操作都有自己的线程，操作系统会把两个线程当作是互不相干的任务分别执行，这样就可能在没有把整个文件装入内存时统计字数。为解决此问题，你必须使两个线程同步工作。

所谓**同步**，是指散步在不同进程之间的若干程序片断，它们的运行必须严格按照规定的某种先后次序来运行，这种先后次序依赖于要完成的特定的任务。如果**用对资源的访问来定义的话，同步是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。**

所谓**互斥**，是指散布在不同进程之间的若干程序片断，当某个进程运行其中一个程序片段时，其它进程就不能运行它们之中的任一程序片段，只能等到该进程运行完这个程序片段后才可以运行。如果**用对资源的访问来定义的话，互斥某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。**

11.什么是线程安全

如果多线程的程序运行结果是可预期的，而且与单线程的程序运行结果一样，那么说明是“线程安全”的。

12.同步与异步

同步：

- 同步的定义：是指一个进程在执行某个请求的时候，若该请求需要一段时间才能返回信息，那么，这个进程将会一直等待下去，直到收到返回信息才继续执行下去。
- 特点：

1. 同步是阻塞模式；
2. 同步是按顺序执行，执行完一个再执行下一个，需要等待，协调运行；

异步：

- 是指进程不需要一直等下去，而是继续执行下面的操作，不管其他进程的状态。当有消息返回时系统会通知进程进行处理，这样可以提高执行的效率。
- 特点：

1. 异步是非阻塞模式，无需等待；
2. 异步是彼此独立，在等待某事件的过程中，继续做自己的事，不需要等待这一事件完成后再工作。线程是异步实现的一个方式。

同步与异步的优缺点：

- 同步可以避免出现死锁，读脏数据的发生。一般共享某一资源的时候，如果每个人都有修改权限，同时修改一个文件，有可能使一个读取另一个人已经删除了内容，就会出错，同步就不会出错。但，同步需要等待资源访问结束，浪费时间，效率低。
- 异步可以提高效率，但，安全性较低。

13.守护、僵尸、孤儿进程的概念

- **守护进程**：运行在后台的一种特殊进程，**独立于控制终端并周期性地执行某些任务**。
- **僵尸进程**：一个进程 fork 子进程，子进程退出，而父进程没有wait/waitpid子进程，那么**子进程的进程描述符仍保存在系统中**，这样的进程称为僵尸进程。僵尸进程是当子进程比父进程先结束，而父进程又没有回收子进程，释放子进程占用的资源，此时子进程将成为一个僵尸进程。

- **孤儿进程**：一个**父进程退出，而它的一个或多个子进程还在运行**，这些子进程称为孤儿进程。（孤儿进程将由 init 进程收养并对它们完成状态收集i工作）

同步阻塞形式
效率是最低的，如果这个线程在等待当前函数返回时，没有执行其他消息处理，而是处于挂起等待状态，那这种情况就叫做同步阻塞；
异步阻塞形式
异步操作是可以被阻塞住的，只不过它不是在处理消息时阻塞，而是在等待消息通知时被阻塞。
同步非阻塞形式
实际上是效率低下的，程序需要在这两种不同的行为之间来回的切换，效率可想而知是低下的。如果这个线程在等待当前函数返回时，仍在执行其他消息处理，那这种情况就叫做同步非阻塞；
异步非阻塞形式
效率更高，程序没有在两种不同的操作中来回切换。

进程切换

为了控制进程的执行，内核必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行。这种行为被称为进程切换。因此可以说，任何进程都是在操作系统内核的支持下运行的，是与内核紧密相关的。

从一个进程的运行转到另一个进程上运行，这个过程中经过下面这些变化：

1. 保存处理机上下文，包括程序计数器和其他寄存器。
2. 更新PCB信息。
3. 把进程的PCB移入相应的队列，如就绪、在某事件阻塞等队列。
4. 选择另一个进程执行，并更新其PCB。
5. 更新内存管理的数据结构。
6. 恢复处理机上下文。

https://www.jianshu.com/p/486b0965c296

进程调度方式
非剥夺调度方式

剥夺调度方式
进程上下文

所谓的进程上下文，就是一个进程在执行的时候，CPU的所有寄存器中的值、进程的状态以及堆栈上的内容，当内核需要切换到另一个进程时，它 需要保存当前进程的所有状态，即保存当前进程的进程上下文，以便再次执行该进程时，能够恢复切换时的状态，继续执行。

一个进程的上下文可以分为三个部分:用户级上下文、寄存器上下文以及系统级上下文。

用户级上下文: 正文、数据、用户堆栈以及共享存储区；

寄存器上下文: 通用寄存器、程序寄存器(IP)、处理器状态寄存器(EFLAGS)、栈指针(ESP)；

系统级上下文: 进程控制块task_struct、内存管理信息(mm_struct、vm_area_struct、pgd、pte)、内核栈。

当发生进程调度时，进行进程切换就是上下文切换。操作系统必须对上面提到的全部信息进行切换，新调度的进程才能运行。内核之所以进入进程上下文是因为进程自身的一些工作需要在内核中做。

一个由C/C++编译的程序占用的内存分为以下几个部分
1、栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其
操作方式类似于数据结构中的栈。
2、堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回
收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。
3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的
全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另
一块区域。 - 程序结束后由系统释放。
4、文字常量区 —常量字符串就是放在这里的。 程序结束后由系统释放
5、程序代码区—存放函数体的二进制代码。

上下文切换就是从当前执行任务切换到另一个任务执行的过程。但是，为了确保下次能从正确的位置继续执行，在切换之前，会保存上一个任务的状态。

阻塞/非阻塞，同步/异步的概念要注意讨论的上下文：
在进程通信层面，阻塞/非阻塞，同步/异步基本是同义词，但是需要注意区分讨论的对象是发送方还是接收方。
发送方阻塞/非阻塞（同步/异步）和接收方的阻塞/非阻塞（同步/异步） 是互不影响的。
在 IO 系统调用层面（ IO system call ）层面，非阻塞 IO 系统调用 和 异步 IO 系统调用存在着一定的差别，它们都不会阻塞进程，但是返回结果的方式和内容有所差别，但是都属于非阻塞系统调用（ non-blocing system call ）
非阻塞系统调用（non-blocking I/O system call 与 asynchronous I/O system call） 的存在可以用来实现线程级别的 I/O 并发，与通过多进程实现的 I/O 并发相比可以减少内存消耗以及进程切换的开销。

14. 用户态和内核态的区别

内核态：cpu可以访问内存的所有数据，包括外围设备，例如硬盘，网卡，cpu也可以将自己从一个程序切换到另一个程序。

用户态：只能受限的访问内存，且不允许访问外围设备，占用cpu的能力被剥夺，cpu资源可以被其他程序获取。

为什么要有用户态和内核态？

由于需要限制不同的程序之间的访问能力, 防止他们获取别的程序的内存数据, 或者获取外围设备的数据, 并发送到网络, CPU划分出两个权限等级 -- 用户态和内核态。

设计模式

工厂方法模式和抽象工厂模式

工厂方法模式，又称工厂模式、多态工厂模式和虚拟构造器模式，通过定义工厂父类负责定义创建对象的公共接口，而子类则负责生成具体的对象。

将类的实例化（具体产品的创建）延迟到工厂类的子类（具体工厂）中完成，即由子类来决定应该实例化（创建）哪一个类。

步骤

步骤1： 创建抽象工厂类，定义具体工厂的公共接口；

步骤2： 创建抽象产品类， 定义具体产品的公共接口；

步骤3： 创建具体产品类（继承抽象产品类） & 定义生产的具体产品；

步骤4：创建具体工厂类（继承抽象工厂类）， 定义创建对应具体产品实例的方法；

步骤5：外界通过调用具体工厂类的方法，从而创建不同具体产品类的实例

优点

更符合开-闭原则 新增一种产品时，只需要增加相应的具体产品类和相应的工厂子类即可

符合单一职责原则 每个具体工厂类只负责创建对应的产品

不使用静态工厂方法，可以形成基于继承的等级结构。

缺点

添加新产品时，除了增加新产品类外，还要提供与之对应的具体工厂类，系统类的个数将成对增加，在一定程度上增加了系统的复杂度；同时，有更多的类需要编译和运行，会给系统带来一些额外的开销；

由于考虑到系统的可扩展性，需要引入抽象层，在客户端代码中均使用抽象层进行定义，增加了系统的抽象性和理解难度，且在实现时可能需要用到DOM、反射等技术，增加了系统的实现难度。

虽然保证了工厂方法内的对修改关闭，但对于使用工厂方法的类，如果要更换另外一种产品，仍然需要修改实例化的具体工厂类；

一个具体工厂只能创建一种具体产品

控制反转（IoC）与依赖注入（DI）

控制反转（IoC

控制反转（Inversion of Control）是一种是面向对象编程中的一种设计原则，用来减低计算机代码之间的耦合度。其基本思想是：借助于“第三方”实现具有依赖关系的对象之间的解耦。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试

依赖注入

依赖注入就是将实例变量传入到一个对象中去

解耦，将依赖之间解耦。因为已经解耦，所以方便做单元测试，尤其是 Mock 测试。

观察者模式

在观察者模式中的Subject就像一个发布者（Publisher），观察者（Observer）完全和订阅者（Subscriber）关联。subject通知观察者就像一个发布者通知他的订阅者。

发布-订阅设计模式

发布者和订阅者不知道对方的存在。需要一个第三方组件，叫做信息中介，它将订阅者和发布者串联起来，它过滤和分配所有输入的消息。换句话说，发布-订阅模式用来处理不同系统组件的信息交流，即使这些组件不知道对方的存在。

区别

在观察者模式中，观察者是知道Subject的，Subject一直保持对观察者进行记录。然而，在发布订阅模式中，发布者和订阅者不知道对方的存在。它们只有通过消息代理进行通信。
在发布订阅模式中，组件是松散耦合的，正好和观察者模式相反。
观察者模式大多数时候是同步的，比如当事件触发，Subject就会去调用观察者的方法。而发布-订阅模式大多数时候是异步的（使用消息队列）。
观察者 模式需要在单个应用程序地址空间中实现，而发布-订阅更像交叉应用模式

观察者模式：当对象间有一对多的依赖关系时，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知并自动更新。

单例模式

确保一个类只有一个实例，并提供了一个全局访问点。单例大约有两种实现方法：懒汉与饿汉。
懒汉：故名思义，不到万不得已就不会去实例化类，也就是说在第一次用到类实例的时候才会去实例化，所以上边的经典方法被归为懒汉实现
饿汉：饿了肯定要饥不择食。所以在单例类定义的时候就进行实例化。
由于要进行线程同步，所以在访问量比较大，或者可能访问的线程比较多时，采用饿汉实现，可以实现更好的性能。这是以空间换时间。
在访问量较小时，采用懒汉实现。这是以时间换空间。

```
class singleton
{
protected:
    singleton() {
        cout << 1 << endl;
        pthread_mutex_init(&mutex, NULL);
    }
private:
    static singleton* p;
public:
    static pthread_mutex_t mutex;
    static singleton* initance() {
        //因为每次判断是否为空都需要被锁定，如果有很多线程的话，就会造成大量线程的阻塞。
        //于是出现了双重锁定。
        if (p == NULL) {
            pthread_mutex_lock(&mutex);
            if (p == NULL) {
                p = new singleton();
            }
            pthread_mutex_unlock(&mutex);
        }
    }
};

pthread_mutex_t singleton::mutex;
singleton* singleton::p = NULL;
```

内部静态变量版本

```
class singleton {
protected:
    singleton() {
        pthread_mutex_init(&mutex, NULL);
    }
public:
    static pthread_mutex_t mutex;
    static singleton* initance() {
        pthread_mutex_lock(&mutex);
        static singleton obj;
        pthread_mutex_unlock(&mutex);
        return &obj;
    }
    int a;
};

pthread_mutex_t singleton::mutex;
```

饿汉实现

因为饿汉实现本来就是线程安全的，不用加锁。

```
//假如有一个全局对象A 构造函数里引用上文中饿汉形式的指针，
//若在A构造函数构造之前以上单例并未构造出来，那就会有问题。
class singleton {
protected:
    singleton(){}
private:
    static singleton* p;
public:
    static singleton* initance() {
        return p;
    }
};
singleton* singleton::p = new singleton;
```

只能实例化没有参数的类型，其它带参数的类型就不行了

```
class Singleton{
public:
    static Singleton* getInstance() {
        return instance;
    }

private:
    Singleton();
    //把复制构造函数和=操作符也设为私有,防止被复制
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);

    static Singleton* instance;
};
//在此处初始化
Singleton* Singleton::instance = new Singleton();
```

使用场景
单例模式常常与工厂模式结合使用，因为工厂只需要创建产品实例就可以了，在多线程的环境下也不会造成任何的冲突，因此只需要一个工厂实例就可以了。

工厂模式

工厂模式包括三种：简单工厂模式、工厂方法模式、抽象工厂模式。

工厂模式的主要作用是封装对象的创建，分离对象的创建和操作过程，用于批量管理对象的创建过程，便于程序的维护和扩展。

简单工厂模式

简单工厂是工厂模式最简单的一种实现，对于不同产品的创建定义一个工厂类，将产品的类型作为参数传入到工厂的创建函数，根据类型分支选择不同的产品构造函数。

工厂方法模式
其实这才是正宗的工厂模式，简单工厂模式只是一个简单的对创建过程封装。工厂方法模式在简单工厂模式的基础上增加对工厂的基类抽象，不同的产品创建采用不同的工厂创建（从工厂的抽象基类派生），这样创建不同的产品过程就由不同的工厂分工解决。

抽象工厂模式

抽象工厂模式对工厂方法模式进行了更加一般化的描述。工厂方法模式适用于产品种类结构单一的场合，为一类产品提供创建的接口；而抽象工厂方法适用于产品种类结构多的场合，就是当具有多个抽象产品类型时，抽象工厂便可以派上用场。

https://www.baidu.com/link?url=ksfaANGbxhBvZEJS9L7cQeGLLfZKa-ozEsVCI0llBIk4aBUVBF6LsxVTc0yhbQ8ZUWp9BCa6O8qK27xV495Ye3oWgDHrXZ4TfYYn__Tg6FC&wd=&eqid=fe1117e200028110000000065e52e402

观察者模式
观察者模式：定义了一种一对多的依赖关系，让多个观察者对象同时监听某一主题对象，在主题对象的状态发生变化时,会通知所有的观察者。
https://www.cnblogs.com/yangjj08/p/10533250.html
MVC
Model（模型） - 模型代表一个存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。
View（视图） - 视图代表模型包含的数据的可视化。
Controller（控制器） - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

你能解释下MVC的完整流程吗？

下面是MVC（模型、视图、控制器）架构的控制流程：

所有的终端用户请求被发送到控制器。

控制器依赖请求去选择加载哪个模型，并把模型附加到对应的视图。

附加了模型数据的最终视图做为响应发送给终端用户。

杂乱

1. Redis

redis是一个key-value存储系统。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis支持各种不同方式的排序。与memcached一样，为了保证效率，数据都是缓存在内存中。区别的是**redis会周期性异步的把更新的数据写入磁盘或者把修改操作写入追加的记录文件**，并且在此基础上实现了master-slave(主从)同步。

Redis本质上是一个Key-Value类型的内存数据库，很像memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据flush到硬盘上进行保存。因为是纯内存操作，Redis的性能非常出色。

Key-Value Store 更加注重对海量数据存取的性能、分布式、扩展性支持上，并不需要传统关系数据库的一些特征，例如：Schema、事务、完整 SQL 查询支持等等，因此在分布式环境下的性能相对于传统的关系数据库有较大的提升。

为什么使用 Key-Value Store

云存储

如果说上一个问题还有可以替代的解决方案（切割数据库）的话，那么对于云存储来说，也许 key-value 的 store 就是唯一的解决方案了。云存储简单点说就是构建一个大型的存储平台给别人用，这也就意味着在这上面运行的应用其实是不可控的。如果其中某个客户的应用随着用户的增长而不断增长时，云存储供应商是没有办法通过数据库的切割来达到 scale 的，因为这个数据是客户的，供应商不了解这个数据自然就没法作出切割。在这种情况下，key-value 的 store 就是唯一的选择了，因为这种条件下的 scalability 必须是自动完成的，不能有人工干预。这也是为什么几乎所有的现有的云存储都是 key-value 形式的，例如 Amazon的 smipleDB，底层实现就是 key-value，还有 google 的 GoogleAppEngine，采用的是 BigTable的存储形式。

Key-Value

键值数据库是一种非关系数据库，它使用简单的键值方法来存储数据。键值数据库将数据存储为键值对集合，其中键作为唯一标识符。键和值都可以是从简单对象到复杂复合对象的任何内容。键值数据库是高度可分区的，并且允许以其他类型的数据库无法实现的规模进行水平扩展。 例如，如果现有分区填满了容量，并且需要更多的存储空间，Amazon DynamoDB 就会将额外的分区分配给表。

Redis数据类型

String，string表示的是一个可变的字节数组，我们初始化字符串的内容、可以拿到字符串的长度，可以获取string的子串，可以覆盖string的子串内容，可以追加子串。采用预分配冗余空间的方式来减少内存的频繁分配。字符串最大长度为512M。**初始化字符串 需要提供「变量名称」和「变量的内容」set ireader beijing**

List 列表的存储结构用的是链表, 而且链表还是双向链表。因为它是链表，所以随机定位性能较弱，首尾插入删除性能较优。如果list的列表长度很长，使用时我们一定要关注链表相关操作的时间复杂度。负下标 链表元素的位置使用自然数0,1,2,....n-1表示，还可以使用负数-1,-2,...-n来表示，-1表示「倒数第一」，-2表示「倒数第二」，那么-n就表示第一个元素，对应的下标为0。 队列／堆栈 链表可以从表头和表尾追加和移除元素，结合使用rpush/rpop/lpush/lpop四条指令，可以将链表作为队列或堆栈使用，左向右向进行都可以

快速链表，首先在列表元素较少的情况下会使用一块连续的内存存储，这个结构是ziplist，也即是压缩列表。它将所有的元素紧挨着一起存储，分配的是一块连续的内存。当数据量比较多的时候才会改成quicklist。Redis将链表和ziplist结合起来组成了quicklist。也就是将多个ziplist使用双向指针串起来使用。这样既满足了快速的插入删除性能，又不会出现太大的空间冗余。

Hash，在实现结构上它使用二维结构，第一维是数组，第二维是链表，hash的内容key和value存放在链表中，数组里存放的是链表的头指针。通过key查找元素时，先计算key的hashcode，然后用hashcode对数组的长度进行取模定位到链表的表头，再对链表进行遍历获取到相应的value值，链表的作用就是用来将产生了「hash碰撞」的元素串起来。

扩容 当hash内部的元素比较拥挤时(hash碰撞比较频繁)，就需要进行扩容。扩容需要申请新的两倍大小的数组，然后将所有的键值对重新分配到新的数组下标对应的链表中(rehash)。如果hash结构很大，比如有上百万个键值对，那么一次完整rehash的过

程就会耗时很长。这对于单线程的Redis里来说有点压力山大。所以Redis采用了渐进式rehash的方案。它会同时保留两个新旧hash结构，在后续的定时任务以及hash结构的读写指令中将旧结构的元素逐渐迁移到新的结构中。这样就可以避免因扩容导致的线程卡顿现象。

缩容 Redis的hash结构不但有扩容还有缩容，缩容的原理和扩容是一致的，只不过新的数组大小要比旧数组小一倍。

Set，Java程序员都知道HashSet的内部实现使用的是HashMap，只不过所有的value都指向同一个对象。Redis的set结构也是一样，它的内部也使用hash结构，所有的value都指向同一个内部值。

SortedSet(zset)，zset底层实现使用了两个数据结构，第一个是hash，第二个是跳跃列表，hash的作用就是关联元素value和权重score，保障元素value的唯一性，可以通过元素value找到相应的score值。跳跃列表的目的在于给元素value排序，根据score的范围获取元素列表。我们需要这个链表按照score值进行排序。这意味着当有新元素需要插入时，需要定位到特定位置的插入点，这样才可以继续保证链表是有序的。跳跃列表使用类似于这种层级制，最下面一层所有的元素都会串起来。然后每隔几个元素挑选出一个代表来，再将这几个代表使用另外一级指针串起来。然后在这些代表里再挑出二级代表，再串起来。最终就形成了金字塔结构。定位插入点时，先在顶层进行定位，然后下潜到下一级定位，一直下潜到最底层找到合适的位置，将新元素插进去。

Redis有哪几种数据淘汰策略？

noeviction:返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令（大部分的写入指令，但DEL和几个例外）

allkeys-lru: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。

volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。

allkeys-random: 回收随机的键使得新添加的数据有空间存放。

volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。

volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

Redis的LRU实现

如果按照HashMap和双向链表实现，需要额外的存储存放 next 和 prev 指针，牺牲比较大的存储空间，显然是不划算的。所以Redis采用了一个近似的做法，就是随机取出若干个key，然后按照访问时间排序后，淘汰掉最不经常使用的

MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据？redis内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

Redis如何设置密码及验证密码？

设置密码：config set requirepass 123456

授权密码：auth 123456

Redis key的过期时间和永久有效分别怎么设置？EXPIRE和PERSIST命令。

Redis集群之间是如何复制的？ 异步复制

Redis集群最大节点个数是多少？ 16384个。

怎么测试Redis的连通性？ping

Redis回收使用的是什么算法？LRU算法

Redis如何做大量数据插入？

Redis2.6开始redis-cli支持一种新的被称之为pipe mode的新模式用于执行大量数据插入工作。

如果有大量的 key 需要设置同一时间过期，一般需要注意什么？

如果大量的 key 过期时间设置的过于集中，到过期的那个时间点，redis 可能会出现短暂的卡顿现象。一般需要在时间上加一个随机值，使得过期时间分散一些。

Redis如何做内存优化？

尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。比如你的web系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的key,而是应该把这个用户的所有信息存储到一张散列表里面.

为什么Redis需要把所有数据放到内存中？

Redis为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以redis具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘I/O速度为严重影响redis的性能。

Redis的并发竞争问题如何解决?

Redis为单进程单线程模式，采用队列模式将并发访问变为串行访问。Redis本身没有锁的概念，Redis对于多个客户端连接并不存在竞争

Redis为什么这么快？

1.完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。

2.数据结构简单，对数据操作也简单。

3.采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗。

4.使用多路I/O复用模型，非阻塞IO。

多路 I/O 复用模型

多路I/O复用模型是利用 select、poll、epoll 可以同时监察多个流的 I/O 事件的能力，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有 I/O 事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll 是只轮询那些真正发出了事件的流），并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。

这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络 IO 的时间消耗），且 Redis 在内存中操作数据的速度非常快，也就是说内存内的操作不会成为影响Redis性能的瓶颈，主要由以上几点造就了 Redis 具有很高的吞吐量。

那么为什么Redis是单线程的

官方FAQ表示，因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）。

https处理的一个过程，对称加密和非对称加密

对称加密

所谓对称加密，就是它们在编码时使用的密钥e和解码时一样d(e=d)，我们就将其统称为密钥k。

对称加解密的过程如下：

发送端和接收端首先要共享相同的密钥k(即通信前双方都需要知道对应的密钥)才能进行通信。发送端用共享密钥k对明文p进行加密，得到密文c，并将得到的密文发送给接收端，接收端收到密文后，并用其相同的共享密钥k对密文进行解密，得出明文p。

一般加密和解密的算法是公开的，需要保持隐秘的是密钥k，流行的对称加密算法有：DES，Triple-DES，RC2和RC4

对称加密的不足主要有两点：

1，发送方和接收方首先需要共享相同的密钥，即存在密钥k的分发问题，如何安全的把共享密钥在双方进行分享，这本身也是一个如何安全通信的问题。

2，密钥管理的复杂度问题。由于对称加密的密钥是一对一的使用方式，若一方要跟n方通信，则需要维护n对密钥。

对称加密的好处是：

加密和解密的速度要比非对称加密快很多，因此常用非对称加密建立的安全信道进行共享密钥的分享，完成后，具体的加解密则使用对称加密。即混合加密系统。另外一个点需要重点说明的是，密钥k的长度对解密破解的难度有很重大的影响，k的长度越长，对应的密码空间就越大，遭到暴力破解或者词典破解的难度就更大，就更加安全。

二，非对称加密

所谓非对称加密技术是指加密的密钥e和解密的密钥d是不同的（e!=d），并且加密的密钥e是公开的，叫做公钥，而解密的密钥d是保密的，叫私钥。

非对称加解密的过程如下：

加密一方找到接收方的公钥e(如何找到呢？大部分的公钥查找工作实际上都是通过数字证书来实现的)，然后用公钥e对明文p进行加密后得到密文c，并将得到的密文发送给接收方，接收方收到密文后，用自己保留的私钥d进行解密，得到明文p，需要注意的是：用公钥加密的密文，只有拥有私钥的一方才能解密，这样就可以解决加密的各方可以统一使用一个公钥即可。

常用的非对称加密算法有：RSA

非对称加密的优点是：

1，不存在密钥分发的问题，解码方可以自己生成密钥对，一个做私钥存起来，另外一个作为公钥进行发布。

2，解决了密钥管理的复杂度问题，多个加密方都可以使用一个已知的公钥进行加密，但只有拥有私钥的一方才能解密。

非对称加密不足的地方是加解密的速度没有对称加密快。

消息队列

消息队列可以简单理解为：把要传输的数据放在队列中。 把数据放到消息队列叫做生产者，从消息队列里边取数据叫做消费者。

为什么要用消息队列？

解耦

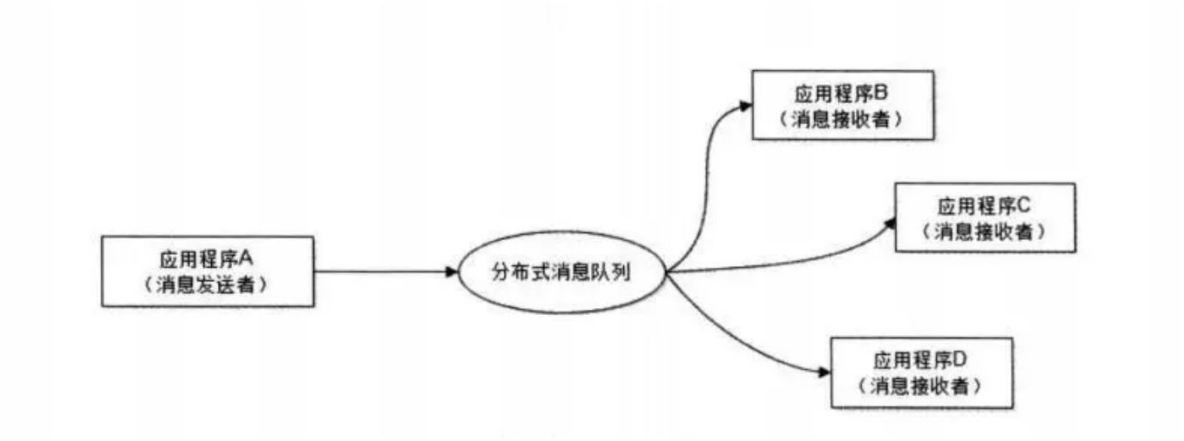
我们知道如果模块之间不存在直接调用，那么新增模块或者修改模块就对其他模块影响较小，这样系统的可扩展性无疑更好一些。

削峰/限流

消息队列具有很好的削峰作用的功能——即通过异步处理，将短时间高并发产生的事务消息存储在消息队列中，从而削平高峰期的并发事务。

高可用

无论是我们使用消息队列来做解耦、异步还是削峰，消息队列肯定不能是单机的。试着想一下，如果是单机的消息队列，万一这台机器挂了，那我们整个系统几乎就是不可用了。所以，当我们项目中使用消息队列，都是得集群/分布式的。要做集群/分布式就必然希望该消息队列能够提供现成的支持，而不是自己写代码手动去实现。



利用消息队列实现事件驱动结构

消息队列使利用发布-订阅模式工作，消息发送者（生产者）发布消息，一个或多个消息接受者（消费者）订阅消息。 从上图可以看到**消息发送者（生产者）和消息接受者（消费者）之间没有直接耦合**，消息发送者将消息发送至分布式消息队列即结束对消息的处理，消息接受者从分布式消息队列获取该消息后进行后续处理，并不需要知道该消息从何而来。**对新增业务，只要对该类消息感兴趣，即可订阅该消息，对原有系统和业务没有任何影响，从而实现网站业务的可扩展性设计。**

另外为了避免消息队列服务器宕机造成消息丢失，会将成功发送到消息队列的消息存储在消息生产者服务器上，等消息真正被消费者服务器处理后才删除消息。在消息队列服务器宕机后，生产者服务器会选择分布式消息队列服务器集群中的其他服务器发布消息。

不要认为消息队列只能利用发布-订阅模式工作，只不过在解耦这个特定业务环境下是使用发布-订阅模式的。除了发布-订阅模式，还有点对点订阅模式（一个消息只有一个消费者），我们比较常用的是发布-订阅模式。

使用消息队列带来的一些问题

- **系统可用性降低**：系统可用性在某种程度上降低，为什么这样说呢？在加入MQ之前，你不用考虑消息丢失或者说MQ挂掉等等的情况，但是，引入MQ之后你就需要去考虑了！
- **系统复杂性提高**：加入MQ之后，你需要保证消息没有被重复消费、处理消息丢失的情况、保证消息传递的顺序性等等问题！
- **一致性问题**：我上面讲了消息队列可以实现异步，消息队列带来的异步确实可以提高系统响应速度。但是，万一消息的真正消费者并没有正确消费消息怎么办？这样就会导致数据不一致的情况了!

缓存

https://www.cnblogs.com/llzhang123/p/9037346.html

缓存能解决的问题

提升性能

绝大多数情况下，select 是出现性能问题最大的地方。一方面，select 会有很多像 join 、group、order、like 等这样丰富的语义，而这些语义是非常耗性能的；另一方面，大多 数应用都是读多写少，所以加剧了慢查询的问题。

分布式系统中远程调用也会耗很多性能，因为有网络开销，会导致整体的响应时间下降。为了挽救这样的性能开销，在业务允许的情况（不需要太实时的数据）下，使用缓存是非常必要的事情。

缓解数据库压力

当用户请求增多时，数据库的压力将大大增加，通过缓存能够大大降低数据库的压力。

缓存的适用场景

对于数据实时性要求不高，对于一些经常访问但是很少改变的数据，读明显多于写，适用缓存就很有必要。比如一些网站配置项。对于性能要求高。比如一些秒杀活动场景。

缓存三种模式

Cache Aside 更新模式

这是最常用的缓存模式了，具体的流程是：

失效：应用程序先从 cache 取数据，没有得到，则从数据库中取数据，成功后，放到缓存中。

命中：应用程序从 cache 中取数据，取到后返回。

更新：先把数据存到数据库中，成功后，再让缓存失效。

Read/Write Through 更新模式

在上面的 Cache Aside 更新模式中，应用代码需要维护两个数据存储，一个是缓存（Cache），一个是数据库（Repository）。而在Read/Write Through 更新模式中，应用程序只需要维护缓存，数据库的维护工作由缓存代理了。

Read Through

Read Through 模式就是在查询操作中更新缓存，也就是说，当缓存失效的时候，Cache Aside 模式是由调用方负责把数据加载入缓存，而 Read Through 则用缓存服务自己来加载。

Write Through

Write Through 模式和 Read Through 相仿，不过是在更新数据时发生。当有数据更新的时候，如果没有命中缓存，直接更新数据库，然后返回。如果命中了缓存，则更新缓存，然后由缓存自己更新数据库（这是一个同步操作）。

Write Behind Caching 更新模式

Write Behind Caching 更新模式就是在更新数据的时候，只更新缓存，不更新数据库，而我们的缓存会异步地批量更新数据库。这个设计的好处就是直接操作内存速度快。因为异步，Write Behind Caching 更新模式还可以合并对同一个数据的多次操作到数据库，所以性能的提高是相当可观的。

但其带来的问题是，数据不是强一致性的，而且可能会丢失。另外，Write Behind Caching 更新模式实现逻辑比较复杂，因为它需要确认有哪些数据是被更新了的，哪些数据需要刷到持久层上。只有在缓存需要失效的时候，才会把它真正持久起来。

反向代理

正向代理的过程，它隐藏了真实的请求客户端，服务端不知道真实的客户端是谁，客户端请求的服务都被代理服务器代替来请求，知名的科学上网工具shadowsocks 扮演的就是典型的正向代理角色。在大陆用浏览器访问 www.google.com 时，会被GFW被残忍的拒绝，于是你可以在国外搭建一台代理服务器，让代理帮我去请求google.com，代理把请求返回的相应结构再返回给我。

而反向代理隐藏了真实的服务端，当我们请求 www.google.com 的时候，其实背后有成千上万台服务器为我们服务，但具体是哪一台，你不知道，也不需要知道，你只需要知道反向代理服务器是谁就好了，www.google.com 就是我们的反向代理服务器，反向代理服务器会帮我们把请求转发到真实的服务器那里去。

两者的区别在于代理的对象不一样：正向代理代理的对象是客户端，反向代理代理的对象是服务端。