随学随记，现学现卖

目录

智能指针 3
auto_ptr: 3
unique_ptr: 3
share_ptr: 3
weak_ptr: 4
数据库 ACID 和隔离级别 5
事务 Transaction: 5
四大特性:ACID 5
隔离级别： 5
脏读、不可重复读、幻读： 5
数据库的锁： 6
行级锁： 6
表级锁： 6
页级锁： 6
分享锁/读锁（Shared locks）: 6
排它锁/写锁（Exclusive Locks）: 6
意向锁： 7
间隙锁（Next-Key)： 7
悲观锁： 7
乐观锁： 7
MVCC(Multi-Version Concurrency Control)多版本并发控制 7
内连接、外连接： 8
inner join（内连接)： 8
left join（左外连接)： 8
right join(右外连接) ： 8
full join(全外连接)： 9
数据库的存储过程： 9
存储过程的优点： 9
索引： 10
索引的优点： 10
索引的缺点： 10
B-Tree 10
B+Tree 11
聚集索引和非聚集索引的区别： 11
Hash 索引和 B+Tree 索引？ 11
Innodb 索引文件（聚集索引)： 11
MyISAM 索引文件和数据文件是分离的（非聚集索引） 12
为什么 InnoDB（聚集）表必须有主键，并且推荐使用整形的自增主键？ 12
数据库的分库分表？（垂直拆分和水平拆分） 12

垂直拆分：12
水平拆分：13
为什么非主键索引结构叶子节点存储的是主键值？13
红黑树14
二叉查找树、平衡树与红黑树的关系？14
红黑树的性质：14
红黑树的三种变换：14
红黑树的变换规则：（ 所有插入的点默认为红色）14
哈希：16
哈希函数的构造方法：16
处理冲突的方法：16
HTTP 和 HTTPS 协议16
HTTP 与 HTTPS 的区别？16
对称加密和非对称加密以及常用协议？17
数字签名和数字证书的区别？17
数字认证的过程：17
客户端使用 HTTPS 与服务器通信混合加密过程：17
怎么保证服务器给客户端下发的公钥是真正的公钥,而不是中间人伪造的公钥？18
TCP 协议19
为什么需要三次握手？19
为什么连接的时候是三次握手，关闭的时候却是四次握手？19
为什么 TIME_WAIT 状态需要经过 2MSL(最大报文段生存时间)才能返回到 CLOSE 状态？19
如果已经建立了连接，但是客户端突然出现故障了怎么办？20
黑盒测试和白盒测试21
黑盒测试定义：21
白盒测试定义：21
黑盒测试方法：21
白盒测试方法：21

智能指针

#include <memory>

C++不支持垃圾自动回收机制，程序员必须手动释放动态申请的空间，否则会发生内存泄漏；智能指针就可以保证我们申请的资源，最后忘记释放的问题，防止内存泄露。

智能指针是一个类模板，**对普通指针进行封装，模板参数为指针指向的类型**，使用时智能指针申请在栈空间，在函数结束时，栈上的变量会释放空间，调用智能指针的析构函数，而这个智能指针析构函数的内部实现了对传入指针的释放操作，从而实现了内存的智能释放。

auto_ptr:

auto_ptr<int> p(new myclass);

1、不能用于数组

2、支持所有权概念，当一个 auto_ptr 对象被用于另一个对象初始化或者赋值时，左边对象获得所有权，右边对象不在拥有所有权。(指针独占问题)

3、auto_ptr 对象不能用于非 new 动态分配对象

4、两个 auto_ptr 不能指向同一个对象

当 auto_ptr 指针作为函数的值传递时，主调函数内部的智能指针所有权被转移，智能指针失效，只能用引用传递。

unique_ptr:

最接近 auto_ptr 的指针，在此基础上提高了犯错误的成本，例如值传递和拷贝构造私有，在写的时候就报错了

1.unique_ptr 持有对对象的独有权—两个 unique_ptr 不能指向一个对象，不能进行复制操作只能进行移动操作

2.提供删除器释放对象，允许用户自定义删除器

3.添加了对象数组的偏特化实现，new[]，delete[]

4.使用 C++ 11 的右值引用特性，实现所有权转移 std::move()

5.线程不安全，但是效率高，一般线程安全用 share_ptr，单线程首选 unique_ptr

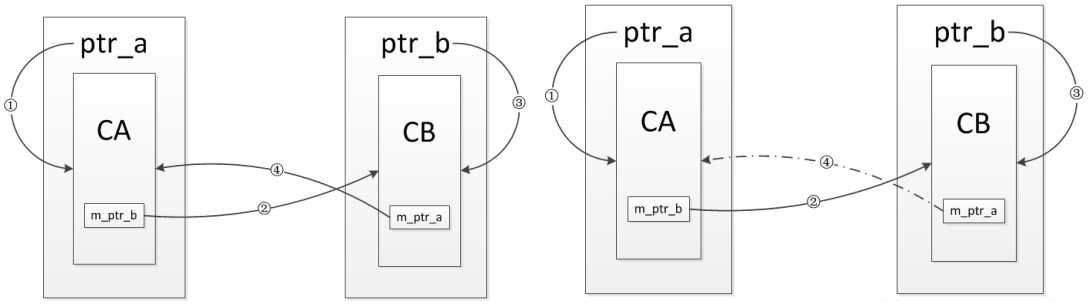
share_ptr:

1.通过引用计数解决了 auto_ptr 的指针独占问题，当引用计数为 0 时，析构对象。

2.会存在一个引用成环的问题。当 A 类有一个成员变量是 B 类的智能指针引用，B 类有一个也有成员变量是 A 类的智能指针引用。当声明两个 share_ptr

智能指针分别指向 A 类和 B 类时，同时将 A 的成员变量指向 B，B 的指向 A。当释放 A,B 类的智能指针时，引用计数并不等于 0，导致无法释放，内存泄露。

3.解决办法则是引入 weak_ptr 指正，将 A B 类内部的成员变量引用其中一个改为 weak_ptr，打破这个环路。



weak_ptr:

1. weak_ptr 虽然是一个模板类，但是不能用来直接定义指向原始指针的对象。

2. weak_ptr 接受 shared_ptr 类型的变量赋值，但是反过来是行不通的，需要使用 lock 函数。

3. weak_ptr 设计之初就是为了服务于 shared_ptr 的，所以不增加引用计数就是它的核心功能。

4. 由于不知道什么之后 weak_ptr 所指向的对象就会被析构掉，所以使用之前请先使用 expired 函数检测一下。

数据库 ACID 和隔离级别

事务 Transaction:

事务其实是一个最小的不可分割的工作单元，事务能够保证一个业务的完整性。

四大特性:ACID

原子性（Atomicity）：事务是最小的单位，必须保证同时成功或者同时失败。。

一致性（Consistency）：指在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。如 A 给 B 转账，不论转账的事务操作是否成功，其两者的存款总额不变。

隔离性（Isolation）：并发的事务是相互隔离的，一个事务内部的操作对另一个事务不可见。

持久性（Durability）：事务一旦执行结束，对数据库中的数据的改变是永久性的。

隔离级别：

读未提交(Read Uncommitted): 一个事务中的读操作可能读到另一个事务中未提交修改的数据，如果事务发生回滚就可能造成错误。避免这些事情的发生就需要我们在写操作的时候加锁，使读写分离，保证读数据的时候，数据不被修改，写数据的时候，数据不被读取。从而保证写的同时不能被另个事务写和读。

提交读(Read Committed): 写数据时增加写锁，就可以保证不出现脏读，也就是保证读的都是提交之后的数据，但是会造成不可重读，即读的时候不加锁，一个读的事务过程中，如果读取数据两次，在两次之间有写事务修改了数据，将会导致两次读取的结果不一致，从而导致逻辑错误。

可重复读(Repeatable Read): 解决不可重复读问题，一个事务中如果有多次读取操作，读取结果需要一致（指的是固定一条数据的一致，幻读指的是查询出的数量不一致）。 这就牵涉到事务中是否加读锁，并且读操作加锁后是否在事务 commit 之前持有锁的问题，如果不加读锁，必然出现不可重复读，如果加锁读完立即释放，不持有，那么就可能在其他事务中被修改，若其他事务已经执行完成，此时该事务中再次读取就会出现不可重复读。

可串行化（Serializable）：解决幻读问题，在同一个事务中，同一个查询多次返回的结果不一致。事务 A 新增了一条记录，事务 B 在事务 A 提交前后各执行了一次查询操作，发现后一次比前一次多了一条记录。幻读是由于并发事务增加记录导致的，这个不能像不可重复读通过记录加锁解决，因为对于新增的记录根本无法加锁。需要将事务串行化，才能避免幻读。

脏读、不可重复读、幻读：

脏读（Dirty Read）:事务 A 在执行的过程中的未提交操作在事务 B 中可见，当事务 A 执行回滚操作，事务 B 将读取到两次不一样的结果。

不可重复读（Nonrepeatable Read）:事务 A 在修改数据库并提交的前后，事务 B 分别执行了读取操作，两次读取到的结果不一致。

幻读（Phantom Read）：不同于不可重复读之处在于，事务 A 执行的是插入操作，幻读强调的是事务 A 插入前后，事务 B 读到的总条目不一致。

数据库的锁：

按锁的粒度划分：表级锁、行级锁、页级锁
按锁级别划分：共享锁、排它锁、意向锁
按加锁方式划分：自动锁、显示锁
按使用方式划分：乐观锁、悲观锁

行级锁：

行级锁分为共享锁和排他锁。行级锁是 MySQL 中锁定粒度最细的锁。InnoDB 引擎支持行级锁和表级锁，只有在通过索引条件检索数据的时候，才使用行级锁，否则使用表级锁。行级锁开销大，加锁慢，锁定粒度最小，发生锁冲突的概率最低，并发度高。

表级锁：

表级锁分为表共享锁和表独占锁，表级锁开销小，加锁快，锁定粒度大，发生锁冲突最高，并发度最低。

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，所以取折中的页级锁，一次锁定相邻的一组记录。BDB 支持页级锁，开销和加锁时间介于表锁和行锁之间，会出现死锁。锁定粒度界于表锁和行锁之间，并发一般。

分享锁/读锁（Shared locks）：

共享锁是针对同一份数据，多个读操作可以同时进行，简单来说即读加锁，不能写并且可并行读；

排它锁/写锁（Exclusive Locks）：

排它锁针对写操作，假如当前写操作没有完成，那么它会阻断其它的写锁和读锁，即写加锁，其它读写都阻塞 。

意向锁：

当事务 A 对一行数据加读锁，而这时候事务 B 又要对这个行数据所在的表加写锁，这是存在冲突，如果系统逐行检查存在的行锁效率很低，所以存在了意向锁。当事务 A 申请行锁额的时候，数据库会自动先开始申请表的意向锁，如果事务 B 发现表中有意向锁时，就会阻塞等到锁释放。

间隙锁（Next-Key）：

当使用范围条件而不是相等条件检索数据，并请求共享或排它锁时，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”，InnoDB 也会对这个间隙加锁，这种锁机制就是所谓的间隙锁。InnoDB 使用间隙锁一方面是为了防止幻读；另一方面是为了满足恢复和复制的需求。显然，在使用范围条件检索并锁定记录时，InnoDB 这种加锁机制会阻塞符合条件范围内键值的并发插入，往往造成严重的锁等待，因此在实际应用开发过程中，尤其是并发插入较多的应用，要尽量优化业务逻辑，尽量使用相等条件来访问更新数据，避免使用范围条件。

悲观锁：

悲观锁认为被它保护的数据是极其不安全的，每时每刻都有可能变动，一个事务拿到悲观锁后（可以理解为一个用户），其他任何事务都不能对该数据进行修改，只能等待锁被释放才可以执行。

乐观锁：

乐观锁的“乐观情绪”体现在认为数据的变动不会太频繁。因此，它允许多个事务同时对数据进行变动。 当事务从数据库中提取一个数据时，会同时提取其时间戳，在事务提交时，对比之前提取的时间戳与现在数据库中该数据的时间戳是否一样，如果一样就认为这段时间没有其它事务操作数据，可以提交。如果不一样，则交给用户处理。

对于 UPDATE、DELETE、INSERT 语句会自动给涉及数据集加排它锁；对于普通的 SELECT 语句不会加任何锁。

MVCC(Multi-Version Concurrency Control)多版本并发控制

MySQL、Oracle 等数据库系统的大多数事务型存储引擎实现的都不是简单的行级锁。基于提高并发性能的考虑，一般都同时实现了 MVCC。MVCC 是通过在每行记录的后面保存两个隐藏的列来实现的，这两个列一个保存了行的创建时间，一个保存了行的过期时间（删除时间）。当然存储的不是实际的时间值，而是系统的版本号。每开始一个新的事务，

系统版本号都会自动递增。事务开始时刻的系统版本号会作为事务的版本号，用来和查询到的每行记录的版本号进行比较。

保存额外的系统版本号，使大多数读操作都可以不用加锁，MVCC 使得读取数据操作很简单，性能好，而且保证只会读取到符合标准的行，不足之处是每行都需要额外的存储空间和更多的检查工作，有一些额外的维护工作。

内连接、外连接：

inner join（内连接）：

两个表的交集，其实就是两张表中的数据通过某个字段相对，查询出相关记录数据。

```
select * from person inner join card on person.cardId=card.id;

mysql> select * from person inner join card on person.cardId=card.id;
+------+------+--------+------+--------+
| id   | name | cardId | id   | name   |
+------+------+--------+------+--------+
|    1 | 张三 |      1 |    1 | 饭卡   |
|    2 | 李四 |      3 |    3 | 农行卡 |
+------+------+--------+------+--------+
2 rows in set (0.00 sec)
```

left join（左外连接）：

会把左边表里面的所有数据取出来，而右边表中的数据，如果有相等的，就显示出来，如果没有，就补 null。

```
select * from person left join card on person.cardId=card.id;

mysql> select * from person left join card on person.cardId=card.id;
+------+------+--------+------+--------+
| id   | name | cardId | id   | name   |
+------+------+--------+------+--------+
|    1 | 张三 |      1 |    1 | 饭卡   |
|    2 | 李四 |      3 |    3 | 农行卡 |
|    3 | 王五 |      6 | NULL | NULL   |
+------+------+--------+------+--------+
3 rows in set (0.00 sec)
```

right join(右外连接) ：

右外连接，会把右边表里面的所有数据取出来，而左边表中的数据，如果有相等的，就显示出来，如果没有，就补 null。

```
select * from person right join card on person.cardId=card.id;

mysql> select * from person right join card on person.cardId=card.id;
+------+------+--------+------+----------+
| id   | name | cardId | id   | name     |
+------+------+--------+------+----------+
|    1 | 张三 |      1 |    1 | 饭卡     |
| NULL | NULL |   NULL |    2 | 建行卡   |
|    2 | 李四 |      3 |    3 | 农行卡   |
| NULL | NULL |   NULL |    4 | 工商行卡 |
| NULL | NULL |   NULL |    5 | 邮政卡   |
+------+------+--------+------+----------+
5 rows in set (0.00 sec)
```

full join(全外连接):

(mysql 不支持)　全连接并集

```
select * from person full join card on person.cardId=card.id;
等价于
select * from person right join card on person.cardId=card.id
union
select * from person left join card on person.cardId=card.id;
```

id	name	cardId	id	name
1	张三	1	1	饭卡
NULL	NULL	NULL	2	建行卡
2	李四	3	3	农行卡
NULL	NULL	NULL	4	工商行卡
NULL	NULL	NULL	5	邮政卡
3	王五	6	NULL	NULL

6 rows in set (0.00 sec)

数据库的存储过程:

SQL 语句需要先编译然后执行，而存储过程 Stored Procedure 是一组为了完成特点功能的 SQL 语句集，经编译后存储在数据库中，用户通过制定存储过程的名字并给定参数（如果该存储过程带有参数）来调用执行它。

存储过程是可编程的函数，在数据库中创建并保存，可以由 SQL 语句和控制结构组成。当想要在不同的应用程序或平台上执行相同的函数或者封装特定的功能时，存储过程非常有用。数据库中的存储过程可以看做是对编程中面向对象方法的模拟，允许控制数据的访问。

存储过程的优点:

1、 增强 SQL 语言的功能和灵活性：存储过程可以用控制语句编写，有很强的灵活性，可以完成复杂的判断和较复杂的运算。
2、 标准组件式编程：存储过程被创建后，可以在程序中多次调用，不需要重新编写该存储过程的 SQL 语句。数据库维护人员修改存储过程的内容并不会影响服务器的程序源码。
3、 较快的执行速度：如果某一操作包含大量的 Transaction-SQL 代码或分别被多次执行，那么存储过程要比批处理的执行速度快。因为存储过程是预编译的，在首次运行一个存储过程时查询，优化器对其进行分析优化，并且给出最终被存储在系统表中的执行计划。而批处理的语句每次执行都要重新进行编译和优化，速度较慢。
4、 减少网络流量：针对同一个数据库对象的操作，存储过程只需要传输调用语句名，存储过程的逻辑代码在数据库中，传输数据量小。
5、 安全机制：通过对执行某一存储过程的权限进行限制，能够实现对相应的数据的访问权限的限制，避免了非授权用户对数据的访问，保证了数据的安全性。

索引：

帮助数据库高效获取数据的**排好序**的**数据结构**。
数据库的索引：数据库中的表中，取一个字段作为索引，就会将这个字段的数据排序放到一种数据结构里，一般以 key_value 的形式存储，该字段为 key，value 为对应条目磁盘中的首地址。

索引的优点：

1. 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
2. 可以大大加快 数据的检索速度，这也是创建索引的最主要的原因。
3. 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。
4. 在使用分组和排序 子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。
5. 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。

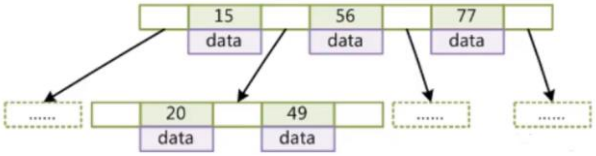
索引的缺点：

1. 创建索引和维护索引要耗费时间，这种时间随着数据 量的增加而增加。
2. 索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。
3. 当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

CURD 工程师，哈哈哈哈哈哈

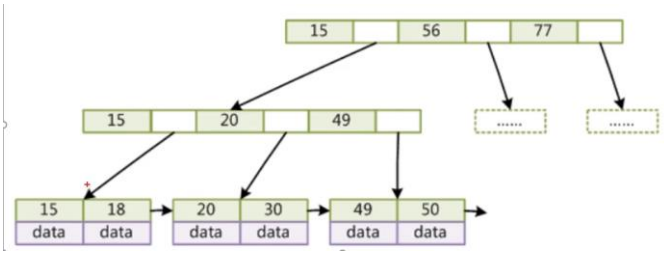
B-Tree

1、 叶子节点具有相同的深度，叶子节点的指针为空
2、 所有索引元素不重复
3、 节点中的数据索引从左到右递增排列



B+Tree

1、 非叶子节点不存储 data，只存储索引，则可以放更多的索引（因为一次读取的空间大小相同，而 B 树的索引还包含 data 的内容，所以放的索引小）
2、 叶子节点包含所有的索引字段，查询性能稳定
3、 叶子节点用指针连接，提高区间访问的性能



聚集索引和非聚集索引的区别：

（索引和数据存在一起就是聚集索引，分开存储就是非聚集索引，聚集索引的叶子节点包含完整的数据记录）
聚集索引（聚簇索引）：数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引。
非聚集索引：该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同，一个表中可以拥有多个非聚集索引。

Hash 索引和 B+Tree 索引？

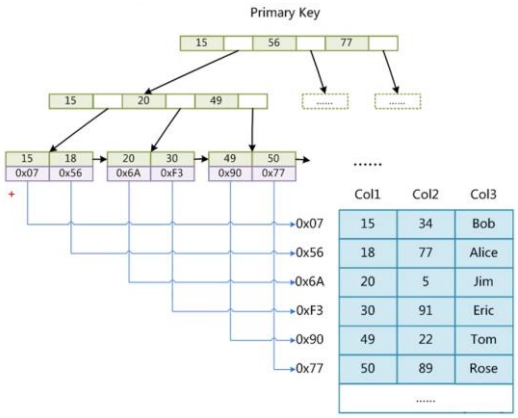
Hash 索引单个查找只用进行一次哈希运算，查找速度比 B+Tree 快，但是 Hash 索引不支持范围查找，对范围查找比较头疼。B+Tree 每个叶子节点之间用指针连接，范围查找性能非常高。

Innodb 索引文件（聚集索引）：

表数据文件本身就是按照 B+Tree 组织一个索引结构文件
*.frm: 存储的表的结构
*ibd: 存储的是索引和数据，就是一个 B+树，数据和索引是存在一起的，而不是地址指针

MyISAM 索引文件和数据文件是分离的（非聚集索引）

*.frm：存储的是表的结构，字段名之类的
*.MYD：存储的是数据 Data
*.MYI：存储的是索引
查找过程：select * from test where col==49
先在表结构中查询 col 在不在 frm 文件中，找到后在 MYI 文件中查找 col=49 的索引，索引的叶子节点挂载着 data 数据的磁盘地址指针，根据这个指针在 MYD 文件中找出数据。



为什么 InnoDB（聚集）表必须有主键，并且推荐使用整形的自增主键？

innoDB 的底层实现就是默认有主键的，如果出现没有主键的情况，系统首先会选一个适合作为主键的列作为默认主键，如果没有合适的，就会自动生成一列来作为索引。
主键使用整形而不使用 UUID 是因为整形所占的空间小，而且在大小比较的过程中计算开销较小。
使用自增主键是当主键不是自增时，就可能向前面已经有序的数据做一个插入操作，而这个插入操作，可能会让 B+树分叉，改变层数，重新平衡等大量操作。而自增有序会顺序插入。

数据库的分库分表？（垂直拆分和水平拆分）

垂直拆分：

垂直分库：当一个数据库中表中的数据量过大时，将其中不同的表建立不同的数据库存放，这样有的业务只依赖其中几张表就可以把这几张表分出来做成相应服务。
垂直分表：适用于字段比较多的表中，例如可以将常用的 20%字段和不常用的 80%字段分开，然后使用主键进行联合。垂直分表可解决跨页问题。跨页问题是一行数据过大，存

储空间超过一页，而需要用到的也就其中几个常用字段，垂直分表可以减少磁盘 IO 次数，读取跟多的数据到内存。

垂直拆分的优缺点：

优点：1、跟随业务进行分割，和微服务概念相似，方便解耦之后的管理及扩展。

2、高并发的场景下，垂直拆分使用多台服务器的 CPU、I/O、内存能提升性能，同时对单机数据库连接数、一些资源限制也得到了提升。

3、能实现冷热数据的分离。

缺点：1、部分业务表无法 join，应用层需要很大的改造，只能通过聚合的方式实现。增加了开发难度。

2、当单库中的表的数据量增大的时候依然没有得到有效解决。

3、分布式事务也是难点。

水平拆分：

库内分表：例如 Order 表中的数据达到 50000 万行，对读写效率影响非常大。所以考虑按照订单编号进行 range 划分，分为五段区间存储在不同的表中。

分库分表：上面的库内分表虽然提高了读写效率，但是这些表仍然在一个数据库内，CPU 内存 IO 都还存在限制。所以在库内分表的基础上，将分表挪动到不同的主机和数据库上。缺陷是当不根据订单编号查询时，就不知道要查询的内容在哪个库，就需要广播查询，每个库都检索，系统要开的线程较多。

优点：1、水平扩展能够无限扩展，不存在某个库某个表过大的情况。

2、能够较好的应对高并发，同时将热点数据打散。

3、应用侧的改动较小，不需要根据业务来拆分。

缺点：1、需要增加一层路由计算，不带分片键的查询会产生广播

2、跨库 join 的性能较差

3、需要处理分布式事务的一致性问题

物理表逻辑表：物理表就是实际划分的存储到不同库或者表里面的表，而逻辑表则是被划分的表在应用层看来仍然是一个整个的表。

Nginx：高性能的 Web 服务器。

为什么非主键索引结构叶子节点存储的是主键值？

一致性

当数据库表进行 DML 操作时，同一行记录的页地址会发生改变，因非主键索引保存的是主键的值，无需进行更改。

节省存储空间

Innodb 数据本身就已经汇聚到主键索引所在的 B+树上了， 如果普通索引还继续再保存一份数据，就会导致有多少索引就要存多少份数据。

红黑树

二叉查找树、平衡树与红黑树的关系？

虽然平衡树解决了二叉查找树退化为近似链表的缺点，能够把查找时间控制在 O(logn)，不过却不是最佳的，因为平衡树要求每个节点的左子树和右子树的高度差至多等于 1，这个要求实在是太严了，导致每次进行插入/删除节点的时候，几乎都会破坏平衡树的第二个规则，进而我们都需要通过左旋和右旋来进行调整，使之再次成为一颗符合要求的平衡树。

显然，如果在那种插入、删除很频繁的场景中，平衡树需要频繁着进行调整，这会使平衡树的性能大打折扣，为了解决这个问题，于是有了红黑树。

与平衡树不同的是，红黑树在插入、删除等操作，不会像平衡树那样，频繁着破坏红黑树的规则，所以不需要频繁着调整，这也是我们为什么大多数情况下使用红黑树的原因。但单单在查找方面的效率的话，平衡树比红黑树快。

所以，我们也可以说，红黑树是一种不大严格的平衡树。也可以说是一个折中发方案。

平衡树是为了解决二叉查找树退化为链表的情况，而红黑树是为了解决平衡树在插入、删除等操作需要频繁调整的情况。

红黑树的性质：

1) 每个结点不是红色就是黑色。
2) 不可能有连在一起的红色结点。
3) 根节点都是黑色 root。
4) 每个红色结点的两个子结点都是黑色。叶子节点都是黑色：出度为 0.

红黑树的三种变换：

1) 颜色变换
2) 左旋
3) 右旋

红黑树的变换规则：（ 所有插入的点默认为红色）

1)变颜色的情况，当前结点的父亲是红色，且它的爷爷结点的另一个子结点也是红色（叔叔结点）。

1- 把父结点变为黑色
2- 把叔叔结点变为黑色
3- 把爷爷结点变为红色
4- 把指针定义到爷爷结点设为当前要操作的结点

2) 左旋：当前父结点是红色，叔叔是黑色的时候，且当前的结点是右子树，以父结点进行左旋。把指针定位到刚才的父结点。

3)右旋：当前父结点是红色，叔叔是黑色的时候，且当前的结点是左子树。右旋 。

1- 把父结点变为黑色

2- 把爷爷结点变为红色

3- 以爷爷结点右旋

哈希：

当查询条件不会变，而且没有部分查询也没有范围查询的情况下可以使用哈希。

哈希函数的构造方法：

①数字分析法
②平方取中法
③除留取余法
④分段叠加法

处理冲突的方法：

①开放地址法（包括线性探测法、二次探测法、伪随机探测法）
②链地址法
③再散列法
④建立一个公共溢出区

HTTP 和 HTTPS 协议

HTTP 端口号：80　　HTTPS 端口号：443
HTTP: 超文本传输协议，为了提供一种发布和接收 HTML 页面的方法。
HTTP 的特点？

1、 以明文的方式发送信息，如果传输的报文被截取，就可以直接获取传输的内容。
2、 HTTP/1.1 之前，默认短连接，每次请求都需要重新进行 TCP 连接。
3、 无状态，协议对客户端没有状态存储，访问一个网站需要反复登录。

HTTPS:是以安全为目标的 HTTP 通道，HTTPS 增加 SSL 协议，该协议位于 TCP/TP 协议与各种应用层协议之间，为数据通讯提供安全支持。

HTTP 与 HTTPS 的区别？

1、 HTTPS 协议需要到 CA(Certificate Authority，证书颁发机构)申请证书，需要费用。
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 使用完全不同的连接方式，前者 80 端口，后者 443 端口。
4、 HTTPS 增加了内容加密、验证身份、保证数据完整性不被篡改的特点。

对称加密和非对称加密以及常用协议？

对称加密的特征：

1、 加密方和解密方使用同一个密钥；

2、 加密解密的速度比较快，适合数据比较长时的使用；

3、 密钥传输过程不安全，且容易破解，密钥管理麻烦。

常用协议：DES、3DES、AES

非对称加密的特征：

1、 加密和解密使用不同的密钥和加密算法。

2、 公钥和私钥是一对存在的，如果用公钥对数据进行加密，只有对应的私钥才能解密。

3、 非对称加密算法强度复杂，安全性依赖于算法与密钥，加密解密速度慢。

常用协议：RSA、DSA、ECC　　SHA-1、MD5

数字签名和数字证书的区别？

数据签名是使用数字证书与信息加密技术，用于鉴别电子数据信息的技术，可理解为加盖在电子文件上的数字指纹。

数字证书是由权威机构公认的第三方认证机构负责签发和管理的个人或者企业的网络数字身份证明。

数字证书是数字签名的基础。

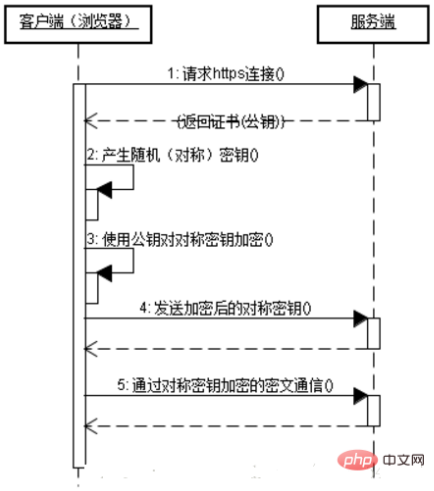
数字认证的过程：

某网站服务器创建公钥和私钥后，将公钥发送请求给数字认证中心，让其验证它的身份。认证中心确认服务器的身份后，用自己的私钥给其证书签名，所以拥有认证中心公钥的人都能验证数字证书的正确性。

客户端使用 HTTPS 与服务器通信混合加密过程：

客户端输入 URL 请求访问一个服务器，先由 DNS 解析域名获取 IP 地址建立连接。服务器回复客户端自己的有权威认证机构签字的数字证书里面包含服务器的公开密钥，客户端获得服务器数字证书后，使用认证机构的公钥进行解密，验证证书的真实性。证书有效之后，客户端创建一个随机密钥，并且用服务器的公钥进行加密。服务器使用自己私钥对数据进行解密，获取到客户端创建的随机密钥。接下来的通讯过程就使用这个随机密钥进行对称加密。

使用非对称加密过程传输客户端创建的随机密钥，使用该随机密钥进行对称加密通信。



怎么保证服务器给客户端下发的公钥是真正的公钥，而不是中间人伪造的公钥？

数字证书中除了包含加密之后的服务器公钥、权威机构的信息之外，还包含了 CA 对证书内容的签名、签名计算方法、证书的对应域名。（签名：Hash 函数计算得到证书数字的摘要，然后权威机构使用自己的私钥加密数字摘要得到数字签名）。

使用浏览器内置的权威机构的公钥解密数字证书，证书的内容（服务器公钥）和证书的数字签名。根据证书上描述的证书签名的计算方法和证书内容计算当前证书的签名与证书里包含的签名一致，表示证书一定是服务器下发，没有被篡改。因为中间人虽然有权威机构的公钥，能够解析服务器下发的证书，但是篡改后中间人需要将证书重新加密，而加密需要权威机构的私钥，无法加密。如果强行加密只会导致客户端无法解密。

考虑证书被掉包的情况，中间人也向权威机构申请了一份证书，然后在服务器给客户端下发证书的时候劫持证书，将自己的假证书下发给客户端，客户端收到之后，依然能够使用权威机构的公钥解密，并且证书签名也一致，但这个时候客户端还会检查证书中的域名是否和当前访问的域名一致，如果不一致会发出警告。

TCP 协议

为什么需要三次握手？

答：为了防止已经失效的连接请求报文段突然传到服务器端，使服务器浪费资源。

当客户端发出的第一个连接请求报文没有丢失，而是在某个网络结点滞留了很长时间，以至于延误到连接释放以后的某个时间才到达服务器。这本来已经是一个失效的报文，但是服务器在接收到连接请求之后，误以为是客户端新的建立连接请求，给该请求分配资源，而客户端根本不会应答，导致了服务器资源的占用浪费。

为什么连接的时候是三次握手，关闭的时候却是四次握手？

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。

为什么 TIME_WAIT 状态需要经过 2MSL(最大报文段生存时间)才能返回到 CLOSE 状态？

虽然按道理，四个报文都发送完毕，我们可以直接进入 CLOSE 状态了，但是我们必须假象网络是不可靠的，有可以最后一个 ACK 丢失。所以 TIME_WAIT 状态就是用来重发可能丢失的 ACK 报文。在 Client 发送出最后的 ACK 回复，但该 ACK 可能丢失。Server 如果没有收到 ACK，将不断重复发送 FIN 片段。所以 Client 不能立即关闭，它必须确认 Server 接收到了该 ACK。Client 会在发送出 ACK 之后进入到 TIME_WAIT 状态。Client 会设置一个计时器，等待 2MSL 的时间。如果在该时间内再次收到 FIN，那么 Client 会重发 ACK 并再次等待 2MSL。所谓的 2MSL 是两倍的 MSL(Maximum Segment Lifetime)。MSL 指一个片段在网络中最大的存活时间，2MSL 就是一个发送和一个回复所需的最大时间。如果直到 2MSL，Client 都没有再次收到 FIN，那么 Client 推断 ACK 已经被成功接收，则结束 TCP 连接。

如果已经建立了连接，但是客户端突然出现故障了怎么办？

TCP 还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为 2 小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔 75 秒钟发送一次。若一连发送 10 个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

黑盒测试和白盒测试

黑盒测试定义：

是通过使用整个软件或某种软件功能来严格地测试，而并没有通过检查程序的源代码或者很清楚地了解该软件的源代码程序具体是怎样设计的。测试人员通过输入他们的数据然后看输出的结果从而了解软件怎样工作。在测试时，把程序看作一个不能打开的黑盆子，在完全不考虑程序内部结构和内部特性的情况下，测试者在程序接口进行测试，它只检查程序功能是否按照需求

白盒测试定义：

是通过程序的源代码进行测试而不使用用户界面。这种类型的测试需要从代码句法发现内部代码在算法，溢出，路径，条件等等中的缺点或者错误，进而加以修正。

黑盒测试方法：

1. 等价类划分：把程序的输入域划分成数据类，据此可以导出测试用例；
2. 边界值分析：经验表明，处理边界情况时程序最容易发生错误，使用边界值分析(Boundary Value Analysis)方法设计测试方案首先应该确定边界情况，这需要经验和创造性，通常输入等价类和输出等价类的边界，就是应该着重测试的程序边界情况。
3. 错误推测：对于程序中可能存在哪类错误的推测，是挑选测试方案的一个重要因素，错误推测法在很大程度上靠直觉和经验进行。它的基本想法是列举出程序中可能有的错误和容易发生错误的特殊情况，并且根据它们选择测试方案，已经发现的错误数目往往和尚未发现的错误数目成正比，因此，在进一步测试时要着重测试那些已经发现了较多错误的程序段。
4. 因果图法：如果在测试时必须考虑输入条件的各种组合，可使用一种适合于描述对于多种条件的组合，相应产生多个动作的形式来设计测试用例，这时就需要利用因果图，因果图方法最终生成的就是判定表，它适合于检查程序输入条件的各种组合情况。
5. 判定表法
6. 正交表法

白盒测试方法：

1. 语句覆盖：选择足够多的测试数据，使得被测程序中每个语句至少被执行一次。
2. 判定覆盖：保证程序中每个语句都被执行一次，并且保证每种可能的结果都应该至少执行一次。
3. 条件覆盖：保证每个语句都至少执行一次，并且保证每个判定表达式中的每个条件都取

到相应的可能值。
4. 判定/条件覆盖：选取足够多的测试数据，使得判定表达式中的每个条件都能取到各种可能值，并且判定表达式也都取到各种可能的结构。
5. 条件组合覆盖：选取足够多的测试数据，使得每个判定表达式条件的各种可能组合都至少出现一次。
6. 路径覆盖：选取足够多的测试数据，使得程序中每条可能的路径都执行一次，是白盒测试中覆盖程度最高的测试方法。