

目录

简介

项目简介	1.1
------	-----

数据库

索引使用说明&注意事项	2.1
InnoDB&MyIsam的区别	2.2
取第一天登录&第二天没有登录的用户	2.3
Mysql的索引类型	2.4
Mysql聚集索引&非聚集索引	2.5
Mysql事物隔离级别	2.6
mysql锁机制详解	2.7
Mysql主从同步原理	2.8

Linux

Crontab表达式详解	3.1
Linux文件权限说明	3.2

其他

TCP的三次握手&四次挥手	4.1
POST&GET区别	4.2
Http状态码	4.3
OSI七层&TCP五层	4.4
栈和队列的区别	4.5

PHP

Nginx&PHP通信	5.1
守护进程的原理与实现	5.2
Cookie&Session区别	5.3
PHP安全处理机制	5.4
require&include的区别	5.5
PHP翻转数组	5.6

缓存

消息队列

RabbitMQ消息一致性怎么保证

7.1

RabbitMQ是如何运转的

7.2

RabbitMQ集群实现方式

7.3

算法实现

冒泡排序

8.1

快速排序

8.2

面试问题集合

[!NOTE|label:插件使用例子] 这是插件使用例子

[!NOTE|style:flat|label:Mylabel|iconVisibility:hidden] "ltype": NOTE 、 "style": flat 、 "label": 自定义标签 、图标不可见

This text is highlighted ! This text is highlighted with **markdown**! This text is highlighted in green! This text is highlighted in red! This text is **highlighted with a custom color!**

数据库创建索引能够大大提高系统的性能。

- 通过创建唯一性的索引，可以保证数据库表中每一行数据的唯一性。
- 可以大大加快数据的检索速度，这也使创建索引的最主要的原因。
- 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。
- 在使用分组和排序子句进行数据检索时，同样可以显著的减少查询中查询中分组和排序的时间。
- 通过使用索引，可以在查询的过程中，使用优化器，提高系统的性能。

增加索引也有许多不利的方面。

- 创建索引和维护索引需要消耗时间，这种时间随着数量的增加而增加。
- 索引需要占物理空间，除了数据表占据数据空间之外，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。
- 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。

应该对如下的列建立索引

- 在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构。
- 在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度。
- 在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的。
- 在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间。
- 在经常使用在where子句中的列上面创建索引，加快条件的判断速度。

有些列不应该创建索引

- 在查询中很少使用或者作为参考的列不应该创建索引。
- 对于那些只有很少数据值的列也不应该增加索引（比如性别，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度）。
- 对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。
- 当修改性能远远大于检索性能时，不应该创建索引，因为修改性能和检索性能是矛盾的。

创建索引的方法

- 直接创建和间接创建（在表中定义主键约束或者唯一性约束时，同时也创建了索引）。

索引的特征

唯一性索引和复合索引。唯一性索引保证在索引列中的全部数据是唯一的，不会包含冗余数据。复合索引就是一个索引创建在两个列或者多个列上。可以减少一在一个表中所创建的索引数量。

- InnoDB支持事物，外键等高级的数据库功能，MyISAM不支持。需要注意的是，InnoDB行级锁也不是绝对的，例如mysql执行一个未定范围的sql时，也还是会锁表，例如sql中like的使用
- 效率，明显MyISAM在插入数据的表现是InnoDB所远远不及的，在删改查，随着InnoDB的优化，差距渐渐变小
- 行数查询，InnoDB不保存行数，也就是select的时候，要扫描全表，MyISAM只需读取保存的行数即可，这也是MyISAM查询速度快的一个因素。
- 锁的支持。**MyISAM只支持表锁。InnoDB支持表锁、行锁 行锁大幅度提高了多用户并发操作的新能。但是InnoDB的行锁，只是在WHERE的主键是有效的，非主键的WHERE都会锁全表的
- 索引，InnoDB会自动创建Auto_Increment类型字段的索引，一般习惯应用于主键，即主键索引（只包含该字段），而MyISAM可以和其他字段创建联合索引。除此之外，MyISAM还支持全文索引（FULLTEXT_INDEX），压缩索引，InnoDB不支持(5.7以后的InnoDB支持全文索引了)

备注：MyISAM的索引和数据是分开的，并且索引是有压缩的，内存使用率就对应提高了不少。能加载更多索引，而Innodb是索引和数据是紧密捆绑的，没有使用压缩从而会造成Innodb比MyISAM体积庞大不小。

InnoDB存储引擎被完全与MySQL服务器整合，InnoDB存储引擎为在主内存中缓存数据和索引而维持它自己的缓冲池。InnoDB存储它的表&索引在一个表空间中，表空间可以包含数个文件（或原始磁盘分区）。这与MyISAM表不同，比如在MyISAM表中每个表被存在分离的文件中。InnoDB表可以是任何尺寸，即使在文件尺寸被限制为2GB的操作系统上。

- 服务器数据备份。InnoDB必须导出SQL来备份，LOAD TABLE FROM MASTER操作对InnoDB是不起作用的，（解决方法是首先把InnoDB表改成MyISAM表，导入数据后再改成InnoDB表，但是对于使用的额外的InnoDB特性（例如外键）的表不适用。）

备注：而且MyISAM应对错误编码导致的数据恢复速度快。MyISAM的数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。InnoDB是拷贝数据文件、备份binlog，或者用mysqldump，支持灾难恢复（仅需几分钟），MyISAM不支持，遇到数据崩溃，基本上很难恢复，所以要经常进行数据备份。

MyISAM索引结构

- MyISAM索引用的B+ tree来储存数据，MyISAM索引的指针指向的是键值的地址，地址存储的是数据。B+Tree的数据域存储的内容为实际数据的地址，也就是说它的索引和实际的数据是分开的，只不过是用索引指向了实际的数据，这种索引就是所谓的（**非聚集索引**）
- 因此，过程为：MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，根据data域的值去读取相应数据记录。

InnoDB索引结构

- 用的也是B+Tree索引结构。Innodb的索引文件本身就是数据文件，即B+Tree的数据域存储的就是实际的数据，这种索引就是（**聚集索引**）。这个索引的key就是数据表的主键，因此InnoDB表数据文件本身就是主索引。
- InnoDB的辅助索引数据域存储的也是相应记录主键的值而不是地址，所以当以辅助索引查找时，会先根据辅助索引找到主键，再根据主键索引找到实际的数据。所以Innodb不建议使用过长的主键，否则会使辅助索引变得过大。
- 与MyISAM索引的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。
- 叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。
- 建议使用自增的字段作为主键，这样B+Tree的每一个结点都会被顺序的填满，而不会频繁的分裂调整，会有效的提升插入数据的效率。

- 过程为：将主键组织到一棵B+树中，而行数据就储存在叶子节点上，若使用“where id = 13”这样的条件查找主键，则按照B+树的检索算法即可查找到对应的叶节点，之后获得行数据。若对Name列进行条件搜索，则需要两个步骤：第一步在辅助索引B+树中检索Name，到达其叶子节点获取对应的主键。第二步使用主键在主索引B+树中再执行一次B+树检索操作，最终到达叶子节点即可获取整行数据。

InnoDB为什么推荐使用自增ID作为主键

- 自增ID可以保证每次插入时B+索引是从右边扩展的，可以避免B+树和频繁合并和分裂（对比使用UUID）。如果使用字符串主键和随机主键，会使得数据随机插入，效率比较差。

innodb引擎的4大特性

- 插入缓冲(insert buffer),二次写(double write),自适应哈希索引(ahi),预读(read ahead)

如何选择？

- 是否要支持事务，如果要请选择innodb，如果不需要可以考虑MyISAM；
- 如果表中绝大多数都只是读查询，可以考虑MyISAM，如果既有读也有写，请使用InnoDB。
- 系统崩溃后，MyISAM恢复起来更困难，能否接受；
- MySQL5.5版本开始Innodb已经成为Mysql的默认引擎(之前是MyISAM)，说明其优势是有目共睹的，如果你不知道用什么，那就用InnoDB，至少不会差。

为什么使用索引会快？

索引就相当于一本书的目录，通过目录来找书中的某一页，确实是很快的，如果没有目录，就需要一页一页的去翻书了，大大降低了效率

- 当没有可用的索引时只能走全表扫描，即把主键索引上的叶子节点从头到尾都扫描一遍，然后每扫描到一行把字段x的值拿出来再比对一下，筛选出满足条件的记录，这个查询是非常低效的。
- 当有可以使用索引x时，该查询会先到二级索引x这个B+树上，快速找到满足要求的叶子节点，而这里的叶子节点上只保存了主键的值，所以还需要通过获得的主键ID值再回到主键索引上查出所有字段的值
- 参考[深入理解MySQL索引原理和实现——为什么索引可以加速查询？](#)

- 在 user_log 表中取出第一天登录第二天没有登录的用户ID

id	time
1	2019-06-01
1	2019-06-02
2	2019-06-01
3	2019-06-01
3	2019-06-02
4	2019-06-01

```
select id from user_log u1 where u1.time="2019-06-01" and not exists(select 1 from user_login u2 where u1.id = u2.id and u2.time = '2019-06-02')
```

B-Tree索引

最常见的索引类型，基于B-Tree数据结构。B-Tree的基本思想是，所有值（被索引的列）都是排过序的，每个叶节点到跟节点距离相等。所以B-Tree适合用来查找某一范围内的数据，而且可以直接支持数据排序（ORDER BY）。但是当索引多列时，列的顺序特别重要，需要格外注意。InnoDB和MyISAM都支持B-Tree索引。InnoDB用的是一个变种B+Tree，而MyISAM为了节省空间对索引进行了压缩，从而牺牲了性能。

Hash索引

[!TIP][label:HASH索引说明] 基于hash表。所以这种索引只支持精确查找，不支持范围查找，不支持排序。这意味着范围查找或ORDER BY都要依赖server层的额外工作。目前只有Memory引擎支持显式的hash索引（但是它的hash是nonunique的，冲突太多时也会影响查找性能）。Memory引擎默认的索引类型即是Hash索引，虽然它也支持B-Tree索引

- 哈希索引只包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行（即不能使用哈希索引来做覆盖索引扫描），不过，访问内存中的行的速度很快（因为memory引擎的数据都保存在内存里），所以大部分情况下这一点对性能的影响并不明显。
- 哈希索引数据并不是按照索引列的值顺序存储的，所以也就无法用于排序
- 哈希索引也不支持部分索引列匹配查找，因为哈希索引始终是使用索引的全部列值内容来计算哈希值的。如：数据列(a,b)上建立哈希索引，如果只查询数据列a，则无法使用该索引。
- 哈希索引只支持等值比较查询，如：=,in(),<=>（注意，<>和<=>是不同的操作），不支持任何范围查询（必须给定具体的where条件值来计算hash值，所以不支持范围查询）。
- 访问哈希索引的数据非常快，除非有很多哈希冲突，当出现哈希冲突的时候，存储引擎必须遍历链表中所有的行指针，逐行进行比较，直到找到所有符合条件的行。
- 如果哈希冲突很多的话，一些索引维护操作的代价也很高，如：如果在某个选择性很低的列上建立哈希索引（即很多重复值的列），那么当从表中删除一行时，存储引擎需要遍历对应哈希值的链表中的每一行，找到并删除对应的引用，冲突越多，代价越大。

Spatial (R-Tree) (空间) 索引

只有MyISAM引擎支持，并且支持的不好。可以忽略。

Full-text索引

主要用来查找文本中的关键字，而不是直接与索引中的值相比较。Full-text索引跟其它索引大不相同，它更像是一个搜索引擎，而不是简单的WHERE语句的参数匹配。你可以对某列分别进行full-text索引和B-Tree索引，两者互不冲突。Full-text索引配合MATCH AGAINST操作使用，而不是一般的WHERE语句加LIKE。

参考

- MySQL 的索引理解
- btree与hash索引的适用场景和限制
- MySQL有哪些索引类型

聚集索引

[!TIP][label:说明] 聚集索引表记录的排列顺序与索引的排列顺序一致

非聚集索引

[!TIP][label:说明] 非聚集索引指定了表中记录的逻辑顺序，但记录的物理顺序和索引的顺序不一致

区别

- 聚集索引一个表只能有一个，而非聚集索引一个表可以存在多个。(因为这个聚簇索引的顺序就决定了数据的顺序)
- 聚集索引存储记录是物理上连续存在，而非聚集索引是逻辑上的连续，物理存储并不连续。
- 聚集索引查询数据速度快，插入数据速度慢(时间花费在“物理存储的排序”上，也就是首先要找到位置然后插入);非聚集索引反之。

说明

- 聚集索引表记录的排列顺序与索引的排列顺序一致，优点是查询速度快，因为一旦具有第一个索引值的纪录被找到，具有连续索引值的纪录也一定物理的紧跟其后。聚集索引的缺点是对表进行修改速度较慢，这是为了保持表中的记录的物理顺序与索引的顺序一致，而把记录插入到数据页的相应位置，必须在数据页中进行数据重排，降低了执行速度。插入数据时速度要慢(时间花费在“物理存储的排序”上，也就是首先要找到位置然后插入)。
- 非聚集索引指定了表中记录的逻辑顺序，但记录的物理顺序和索引的顺序不一致，聚集索引和非聚集索引都采用了B+树的结构，但非聚集索引的叶子层并不与实际的数据页相重叠，而采用叶子层包含一个指向表中的记录在数据页中的指针的方式。非聚集索引比聚集索引层次多，添加记录不会引起数据顺序的重组。
- 索引是通过二叉树的数据结构来描述的，我们可以这么理解聚簇索引：索引的叶节点就是数据节点。而非聚簇索引的叶节点仍然是索引节点，只不过有一个指针指向对应的数据块

事务的基本要素

- 原子性 (Atomicity) : 事务开始后所有操作，要么全部做完，要么全部不做，不可能停滞在中间环节。事务执行过程中出错，会回滚到事务开始前的状态，所有的操作就像没有发生一样。也就是说事务是一个不可分割的整体，就像化学中学过的原子，是物质构成的基本单位。
- 一致性 (Consistency) : 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 (Isolation) : 同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。
- 持久性 (Durability) : 事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚。

事务的并发问题

不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

- 脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据
- 不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。
- 幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

MySQL事务隔离级别

mysql默认的事务隔离级别为 可重复读(repeatable-read)

事务隔离级别	脏读	不可重复读	幻读
读未提交 (read-uncommitted)	是	是	是
读已提交 (read-committed)	否	是	是
可重复读 (repeatable-read)	否	否	是
串行化 (serializable)	否	否	否

Mysql锁说明

- 共享锁(S锁) 用于只读操作(SELECT)，锁定共享的资源。共享锁不会阻止其他用户读，但是阻止其他的用户写和修改。
- 更新锁(U锁) 用于可更新的资源中。防止当多个会话在读取、锁定以及随后可能进行的资源更新时发生常见形式的死锁。
- 独占锁(X锁，也叫排他锁) 一次只能有一个独占锁用在一个资源上，并且阻止其他所有的锁包括共享锁。写是独占锁，可以有效的防止“脏读”。

事物隔离实现说明

- 读未提交 如果一个事务已经开始写数据，则另外一个数据则不允许同时进行写操作，但允许其他事务读此行数据。该隔离级别可以通过“排他写锁”实现。
- 读已提交 读取数据的事务允许其他事务继续访问该行数据，但是未提交的写事务将会禁止其他事务访问该行。可以通过“瞬间共享读锁”和“排他写锁”实现。
- 可重复读 读取数据的事务将会禁止写事务（但允许读事务），写事务则禁止任何其他事务。可以通过“共享读锁”和“排他写锁”实现。
- 串行化 读加共享锁，写加排他锁，读写互斥。

参考

- MySQL的四种事务隔离级别
- 面试官：谈谈Mysql事务隔离级别？
- mysql/mariadb知识点总结（21）：事务隔离级别（事务总结之三）

按锁的粒度划分

- 表锁：意向锁(IS锁、IX锁)、自增锁；
- 行锁：记录锁、间隙锁、临键锁、插入意向锁；

共享/排它锁(Shared and Exclusive Locks)

- 共享锁 (Share Locks, 记为S锁)，读取数据时加S锁 (共享锁之间不互斥，简记为：读读可以并行)
- 排他锁 (eXclusive Locks, 记为X锁)，修改数据时加X锁 (排他锁与任何锁互斥，简记为：写读，写写不可以并行)
- 排他锁，一旦写数据的任务没有完成，数据是不能被其他任务读取的，这对并发度有较大的影响。对应到数据库，可以理解为，写事务没有提交，读相关数据的select也会被阻塞，这里的select是指加了锁的，普通的select仍然可以读到数据(快照读)。

意向锁(Intention Locks)

InnoDB为了支持多粒度锁机制(multiple granularity locking)，即允许行级锁与表级锁共存，而引入了意向锁(intention locks)。意向锁是指，未来的某个时刻，事务可能要加共享/排它锁了，先提前声明一个意向。

1. 意向锁是一个表级别的锁(table-level locking)；
2. 意向锁又分为：
 - 意向共享锁(intention shared lock, IS)，它预示着，事务有意向对表中的某些行加共享S锁；
 - 意向排它锁(intention exclusive lock, IX)，它预示着，事务有意向对表中的某些行加排它X锁；

记录锁(Record Locks)

- 记录锁，它封锁索引记录

间隙锁(Gap Locks)

- 间隙锁的主要目的，就是为了防止其他事务在间隔中插入数据，以导致“不可重复读”。如果把事务的隔离级别降级为读提交(Read Committed, RC)，间隙锁则会自动失效。
- 间隙锁，它封锁索引记录中的间隔，或者第一条索引记录之前的范围，又或者最后一条索引记录之后的范围

临键锁(Next-key Locks)

- 临键锁，是记录锁与间隙锁的组合，它的封锁范围，既包含索引记录，又包含索引区间。
- 默认情况下，innodb使用next-key locks来锁定记录。但当查询的索引含有唯一属性的时候，Next-Key Lock 会进行优化，将其降级为Record Lock，即仅锁住索引本身，不是范围。

插入意向锁(Insert Intention Locks)

- 对已有数据行的修改与删除，必须加强互斥锁(X锁)，那么对于数据的插入，是否还需要加这么强的锁，来实施互

斥呢？插入意向锁，孕育而生。

- 插入意向锁，是间隙锁(Gap Locks)的一种（所以，也是实施在索引上的），它是专门针对insert操作的。多个事务，在同一个索引，同一个范围区间插入记录时，如果插入的位置不冲突，不会阻塞彼此。

自增锁(Auto-inc Locks)

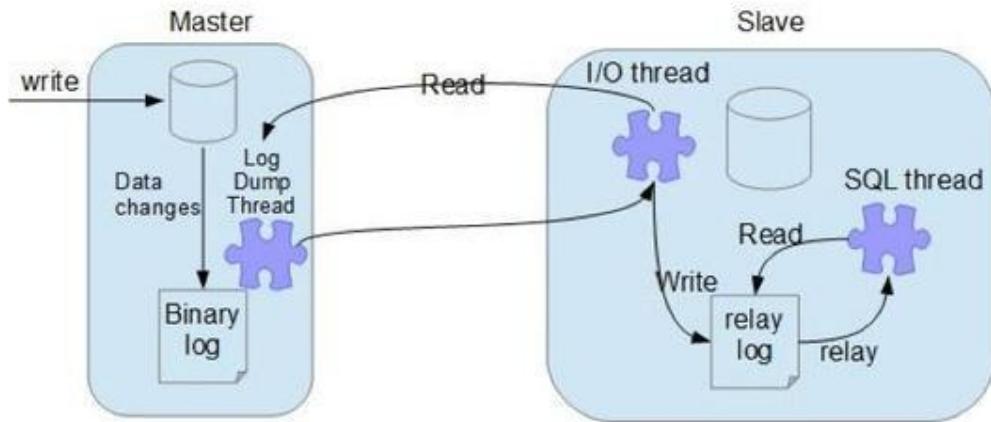
- 自增锁是一种特殊的表级别锁(table-level lock)，专门针对事务插入AUTO_INCREMENT类型的列。最简单的情况，如果一个事务正在往表中插入记录，所有其他事务的插入必须等待，以便第一个事务插入的行，是连续的主键值。

参考

- [mysql锁机制详解](#)
- [MySQL锁机制与用法分析](#)
- [MySQL InnoDB锁机制全面解析分享](#)
- [乐观锁与悲观锁——解决并发问题](#)
- [MySQL学习笔记（四）悲观锁与乐观锁](#)

MySQL 主从复制原理

[!TIP][label:说明] 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。当主节点有多个从节点时，对于每一个主从连接，主节点会为每一个当前连接的从节点建一个binary log dump 进程



1. 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。
2. 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。
3. 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。

流程

数据库有个bin-log二进制文件，记录了所有sql语句。

1. binlog输出线程：每当有从库连接到主库的时候，主库都会创建一个线程然后发送binlog内容到从库。在从库里，当复制开始的时候，从库就会创建两个线程进行处理：
2. 从库创建一个I/O线程，该线程连接到主库并请求主库发送binlog里面的更新记录到从库上。从库I/O线程读取主库的binlog输出线程发送的更新并拷贝这些更新到本地文件，其中包括relay log文件。
3. 从库的SQL线程：从库创建一个SQL线程，这个线程读取从库I/O线程写到relay log的更新事件并执行。

问题&解决

- 主库宕机后，数据丢失

半同步复制

- 主库写压力大，因从库只有一个sql 线程来持久化，复制可能延迟

并行复制

- 复制出错处理 常见：1062（主键冲突）， 1032（记录不存在）

手动处理 OR 跳过复制错误：`set global sql_slave_skip_counter=1`

主从复制延迟

1. 主节点如果执行一个很大的事务(更新千万行语句， 总之执行很长时间的事务)， 那么就会对主从延迟产生较大的影响
2. 网络延迟， 日志较大， slave数量过多。
3. 主上多线程写入， 从节点只有单线程恢复

处理办法：

1. 大事务：将大事务分为小事务， 分批更新数据。
2. 减少Slave的数量， 不要超过5个， 减少单次事务的大小。
3. MySQL 5.7之后， 可以使用多线程复制， 使用MGR复制架构

半同步复制

原理:事务在主库写完binlog后需要从库返回一个已接受， 才放回给客户端

- 5.5集成到mysql， 以插件的形式存在， 需要单独安装
- 确保事务提交后binlog至少传输到一个从库
- 不保证从库应用完成这个事务的binlog
- 性能有一定的降低
- 网络异常或从库宕机， 卡主库， 直到超时或从库恢复

并行复制

原理：从库多线程apply binlog

- 在社区5.6中新增
- 库级别并行应用binlog， 同一个库数据更改还是串行的
- 5.7版本并行复制基于事务组

联级复制（部分数据复制）

- A->B->C
- B中添加参数log_slave_updates
- B将把A的binlog记录到自己的binlog日志中

图解



特殊字符

特殊字符	代表意义
星号(*)	表示所有可能的值，可以理解为每。
逗号(,)	用逗号隔开的值表示一个列表范围，如1,2,3 每天每小时的第一、第二、第三分钟。
中杠(-)	用中杠隔开的值表示一个数值范围，如1-10 每天每小时的1到10分钟。
正斜线(/)	指定执行任务的间隔频率，如 0 10-18/2 *每天的十点到十八点间隔2小时执行。

实例

```
# 每分钟执行一次
* * * * *

# 每小时的第3和第15分钟执行
3,15 * * * *

#在上午的8点到11点的第3和第15分钟执行
3,15 8-11 * * *

#在每隔2天的上午8点和11点的第3和第15分钟执行
3,15 8-11 */2 * *

#每个星期一的上午8点到11点的第3和第15分钟执行
3,15 8-11 * * 1
```

```
#每晚的21: 30执行  
30 21 * * *  
#每月1、10、22日的4:30执行  
30 4 1,10,22 * *  
#每周六、日1:10执行  
10 1 * * 6,7  
#每天18:00到23:00之间每隔30分钟执行  
0/30 18-23 * * *  
#星期六的23:00执行  
0 23 * * 6  
#每小时执行一次  
* */1 * * *  
#晚上11点到早上7点之间，每小时执行一次  
* 23-7/1 * * *  
#每月的4号与每周一到周三的11点  
0 11 4 * 1-3  
#一月一号的4点  
0 4 1 1 *
```

例子

```
→ ~ ls -l
total 24
drwx----- 5 zhanglinyu staff 170 1 3 2017 Applications
drwx-----+ 28 zhanglinyu staff 952 9 14 16:12 Desktop
drwx-----+ 22 zhanglinyu staff 748 9 12 19:31 Documents
drwx-----+ 27 zhanglinyu staff 918 9 18 10:55 Downloads
drwx-----@ 74 zhanglinyu staff 2516 8 10 12:43 Library
drwx-----+ 3 zhanglinyu staff 102 9 30 2016 Movies
drwx-----+ 7 zhanglinyu staff 238 7 5 11:22 Music
drwxr-xr-x 3 zhanglinyu staff 102 8 22 13:31 New_dir
drwx-----+ 4 zhanglinyu staff 136 9 30 2016 Pictures
drwxr-xr-x+ 6 zhanglinyu staff 204 1 4 2017 Public
drwxr-xr-x 13 zhanglinyu staff 442 3 31 16:05 jsk
drwxr-xr-x 16 zhanglinyu staff 544 8 23 16:45 linxiwork
--w--w-rw- 1 zhanglinyu staff 32 9 18 09:58 pu.sh
-rw----- 1 zhanglinyu staff 0 9 18 11:25 pus.sh
----- 1 zhanglinyu staff 0 9 18 11:25 push.sh
-rw-r--r-- 1 zhanglinyu staff 0 9 18 11:26 puss.sh
-rw-r--r-- 1 zhanglinyu staff 39 8 23 19:11 test.gz
-rw-r--r-- 1 zhanglinyu staff 1674 8 31 19:56 test.txt.gz
drwxr-xr-x 2 zhanglinyu staff 68 8 23 19:06 test1
-rw-r--r-- 1 zhanglinyu staff 0 9 18 09:46 txt
drwxr-xr-x 12 zhanglinyu staff 408 7 26 17:03 work
→ ~
```

位置说明

-rwxrwxr-x 1 rich rich 4882 2010-09-18 13:58 myprog

文件属主的权限
属组成员的权限
其他用户的权限

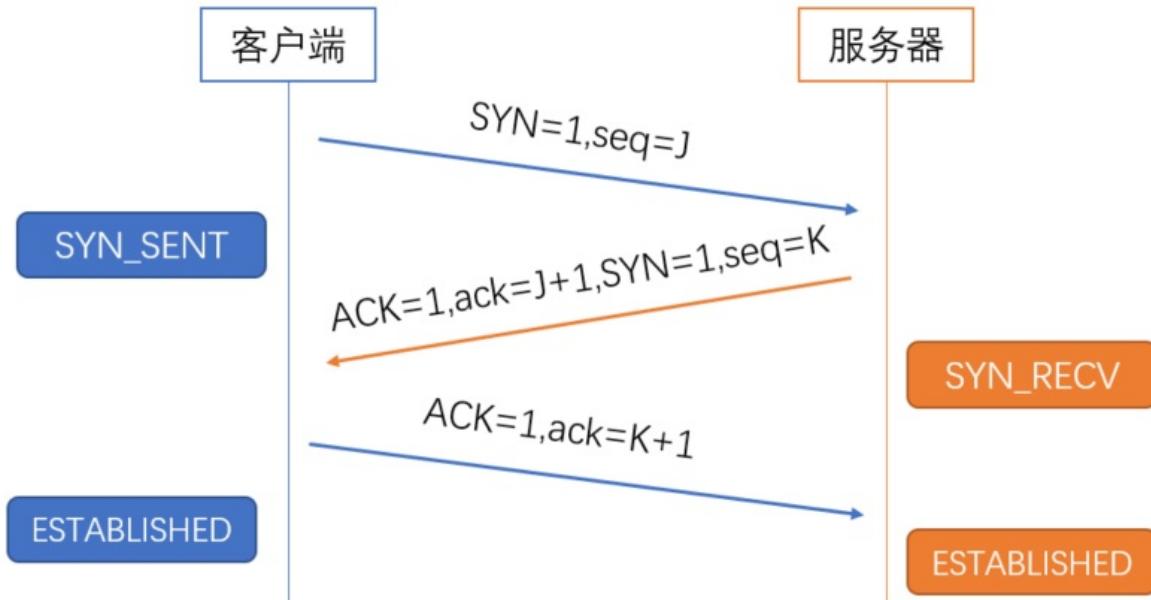
第一个字段

- - 代表文件
- d 代表目录
- l 代表链接

- c 代表字符型设备
- b 代表块设备
- n 代表网络设备

权限值说明

权限	权限数值	二进制	具体作用
r	4	00000100	read, 读取。当前用户可以读取文件内容，当前用户可以浏览目录。
w	2	00000010	write, 写入。当前用户可以新增或修改文件内容，当前用户可以删除、移动目录或目录内文件。
x	1	00000001	execute, 执行。当前用户可以执行文件，当前用户可以进入目录。



TCP三次握手示意

第一次握手(**SYN=1, seq=x**):

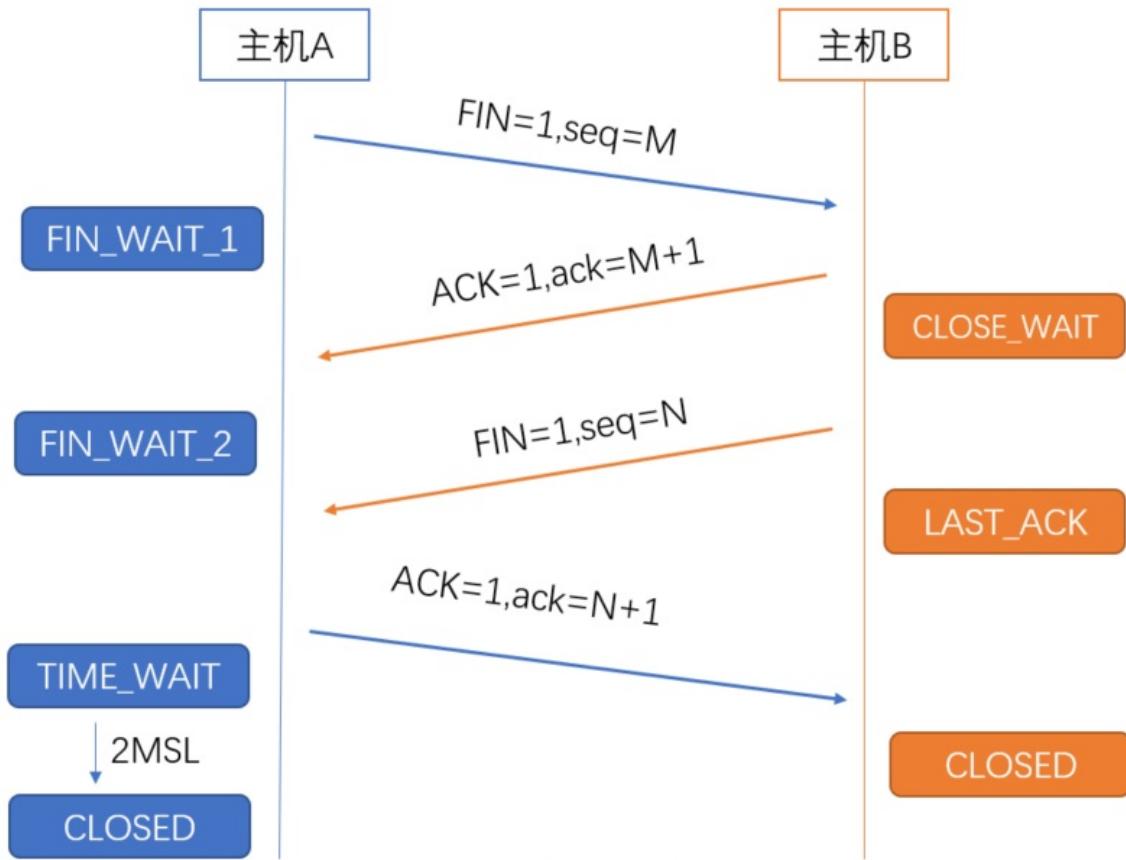
- 客户端发送一个 TCP 的 SYN 标志位置1的包，指明客户端打算连接的服务器的端口，以及初始序号 X,保存在包头的序列号(Sequence Number)字段里。
- 发送完毕后，客户端进入 SYN_SEND 状态。

第二次握手(**SYN=1, ACK=1, seq=y, ACKnum=x+1**):

- 服务器发回确认包(ACK)应答。即 SYN 标志位和 ACK 标志位均为1。服务器端选择自己 ISN 序列号，放到 Seq 域里，同时将确认序号(Acknowledgement Number)设置为客户的 ISN 加1，即 $X+1$ 。 - 发送完毕后，服务器端进入 SYN_RECV 状态。

第三次握手(**ACK=1, ACKnum=y+1**)

- 客户端再次发送确认包(ACK)，SYN 标志位为0，ACK 标志位为1，并且把服务器发来 ACK 的序号字段+1，放在确定字段中发送给对方，并且在数据段放写ISN的+1
- 发送完毕后，客户端进入 ESTABLISHED 状态，当服务器端接收到这个包时，也进入 ESTABLISHED 状态，TCP 握手结束。



第一次挥手($\text{FIN}=1, \text{seq}=x$)

- 假设客户端想要关闭连接，客户端发送一个 FIN 标志位置为1的包，表示自己已经没有数据可以发送了，但是仍然可以接受数据。
- 发送完毕后，客户端进入 **FIN_WAIT_1** 状态。

第二次挥手($\text{ACK}=1, \text{ACKnum}=x+1$)

- 服务器端确认客户端的 FIN 包，发送一个确认包，表明自己接受到了客户端关闭连接的请求，但还没有准备好关闭连接。
- 发送完毕后，服务器端进入 **CLOSE_WAIT** 状态，客户端接收到这个确认包之后，进入 **FIN_WAIT_2** 状态，等待服务器端关闭连接。

第三次挥手($\text{FIN}=1, \text{seq}=y$)

- 服务器端准备好关闭连接时，向客户端发送结束连接请求，FIN 置为1。
- 发送完毕后，服务器端进入 **LAST_ACK** 状态，等待来自客户端的最后一个ACK。

第四次挥手($\text{ACK}=1, \text{ACKnum}=y+1$)

- 客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入 TIME_WAIT 状态，等待可能出现的要求重传的 ACK 包。
- 服务器端接收到这个确认包之后，关闭连接，进入 CLOSED 状态。
- 客户端等待了某个固定时间（两个最大段生命周期，2MSL，2 Maximum Segment Lifetime）之后，没有收到服务器端的 ACK，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入 CLOSED 状态。

GET - 从指定的资源请求数据。

- GET 请求可被缓存
- GET 请求保留在浏览器历史记录中
- GET 请求可被收藏为书签
- GET 请求不应在处理敏感数据时使用
- GET 请求有长度限制
- GET 请求只应当用于取回数据
- GET 只允许 ASCII 字符。

POST - 向指定的资源提交要被处理的数据

- POST 请求不会被缓存
- POST 请求不会保留在浏览器历史记录中
- POST 不能被收藏为书签
- POST 请求对数据长度没有要求
- POST 数据类型没有限制

对比

区别	GET	POST
后退按钮&刷新	无害	数据会被重新提交(浏览器应该告知用户数据会被重新提交)
书签	可收藏为书签	不可收藏为书签
缓存	能被缓存	不能缓存
编码类型	application/x-www-form-urlencoded	application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码。
历史	参数保留在浏览器历史中。	参数不会保存在浏览器历史中。
对数据长度的限制	是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。	无限制。
对数据类型的限制	只允许 ASCII 字符。	没有限制。也允许二进制数据。
安全性	所发送的数据是 URL 的一部分。	数据不会显示在 URL 中

其他 HTTP 请求方法

方法	描述

HEAD	与 GET 相同，但只返回 HTTP 报头，不返回文档主体。
PUT	上传指定的 URI 表示。
DELETE	删除指定资源。
OPTIONS	返回服务器支持的 HTTP 方法。
CONNECT	把请求连接转换到透明的 TCP/IP 通道。

1XX : 信息状态码

- 100 Continue 继续，一般在发送 post 请求时，已发送了 http header 之后服务端 将返回此信息，表示确认，之后发送具体参数信息

2XX : 成功状态码

- 200 OK 正常返回信息 201 Created 请求成功并且服务器创建了新的资源
- 202 Accepted 服务器已接受请求，但尚未处理

3XX : 重定向

- 301 Moved Permanently 请求的网页已永久移动到新位置。
- 302 Found 临时性重定向。 303 See Other 临时性重定向，且总是使用 GET 请求新的 URI 。
- 304 Not Modified 自从上次请求后，请求的网页未修改过。

4XX : 客户端错误

- 400 Bad Request 服务器无法理解请求的格式，客户端不应当尝试再次使用相同的内容发起请求。
- 401 Unauthorized 请求未授权。
- 403 Forbidden 禁止访问。
- 404 Not Found 找不到如何与 URI 相匹配的资源。

5XX: 服务器错误

- 500 Internal Server Error 最常见的服务器端错误。
- 503 Service Unavailable 服务器端暂时无法处理请求（可能是过载或维护）。

参考列表

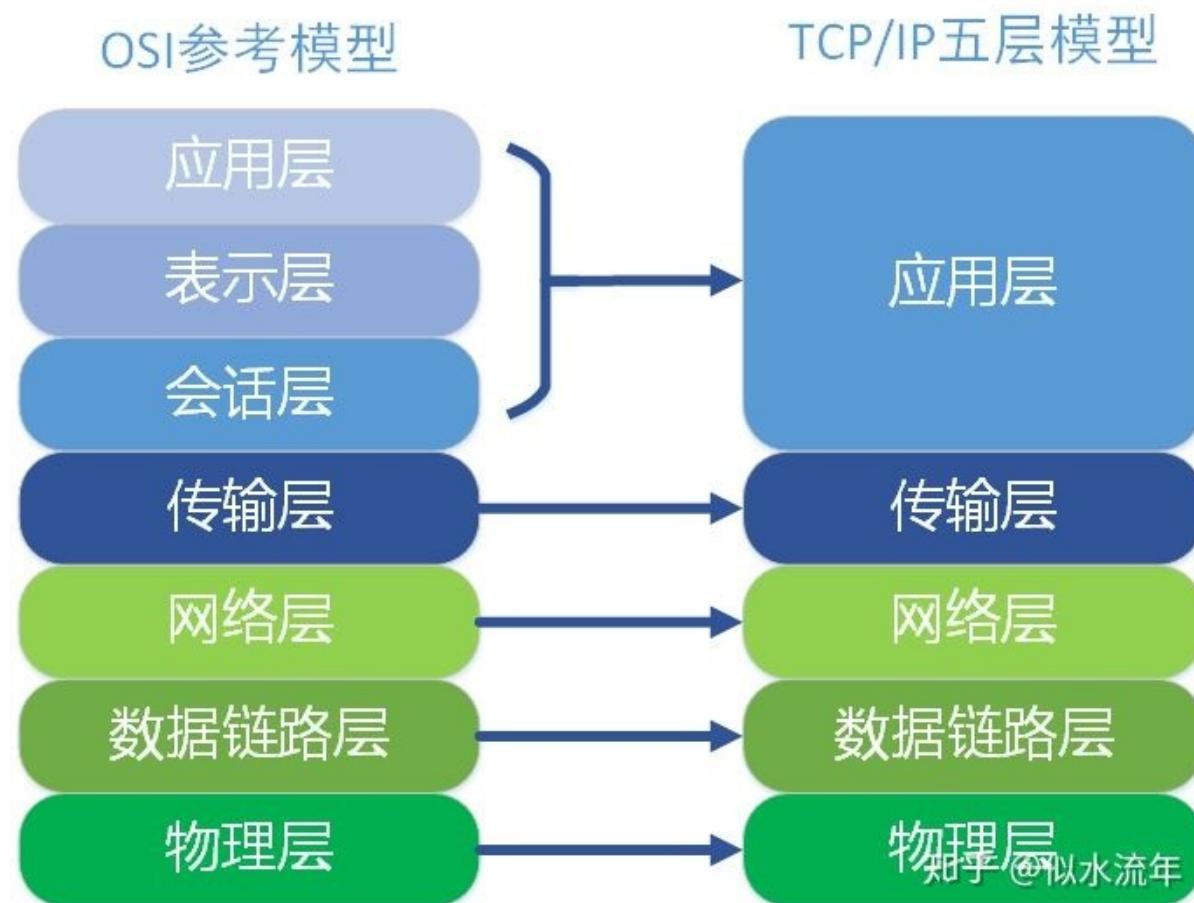
状态码	含义
100	客户端应当继续发送请求。这个临时响应是用来通知客户端它的部分请求已经被服务器接收，且仍未被拒绝。客户端应当继续发送请求的剩余部分，或者如果请求已经完成，忽略这个响应。服务器必须在请求完成后向客户端发送一个最终响应。
101	服务器已经理解了客户端的请求，并将通过Upgrade 消息头通知客户端采用不同的协议来完成这个请求。在发送完这个响应最后的空行后，服务器将会切换到在Upgrade 消息头中定义的那些协议。 只有在切换新的协议更有好处的时候才应该采取类似措施。例如，切换到新的HTTP 版本比旧版本更有优势，或者切换到一个实时且同步的协议以传送利用此类特性的资源。
102	由WebDAV (RFC 2518) 扩展的状态码，代表处理将被继续执行。
200	请求已成功，请求所希望的响应头或数据体将随此响应返回。
	请求已经被实现，而且有一个新的资源已经依据请求的需要而建立，且其 URI 已经随Location 头信息返

202	服务器已接受请求，但尚未处理。正如它可能被拒绝一样，最终该请求可能会也可能不会被执行。在异步操作的场合下，没有比发送这个状态码更方便的做法了。返回202状态码的响应的目的是允许服务器接受其他过程的请求（例如某个每天只执行一次的基于批处理的操作），而不必让客户端一直保持与服务器的连接直到批处理操作全部完成。在接受请求处理并返回202状态码的响应应当在返回的实体中包含一些指示处理当前状态的信息，以及指向处理状态监视器或状态预测的指针，以便用户能够估计操作是否已经完成。	
203	服务器已成功处理了请求，但返回的实体头部元信息不是在原始服务器上有效的确定集合，而是来自本地或者第三方的拷贝。当前的信息可能是原始版本的子集或者超集。例如，包含资源的元数据可能导致原始服务器知道元信息的超级。使用此状态码不是必须的，而且只有在响应不使用此状态码便会返回200 OK的情况下才是合适的。	
204	服务器成功处理了请求，但不需要返回任何实体内容，并且希望返回更新了的元信息。响应可能通过实体头部的形式，返回新的或更新后的元信息。如果存在这些头部信息，则应当与所请求的变量相呼应。如果客户端是浏览器的话，那么用户浏览器应保留发送了该请求的页面，而不产生任何文档视图上的变化，即使按照规范新的或更新后的元信息应当被应用到用户浏览器活动视图中的文档。由于204响应被禁止包含任何消息体，因此它始终以消息头后的第一个空行结尾。	
205	服务器成功处理了请求，且没有返回任何内容。但是与204响应不同，返回此状态码的响应要求请求者重置文档视图。该响应主要是被用于接受用户输入后，立即重置表单，以便用户能够轻松地开始另一次输入。与204响应一样，该响应也被禁止包含任何消息体，且以消息头后的第一个空行结束。	
206	服务器已经成功处理了部分 GET 请求。类似于 FlashGet 或者迅雷这类的 HTTP 下载工具都是使用此类响应实现断点续传或者将一个大文档分解为多个下载段同时下载。该请求必须包含 Range 头信息来指示客户端希望得到的内容范围，并且可能包含 If-Range 来作为请求条件。响应必须包含如下的头部域：Content-Range 用以指示本次响应中返回的内容的范围；如果是 Content-Type 为 multipart/byteranges 的多段下载，则每一 multipart 段中都应包含 Content-Range 域用以指示本段的内容范围。假如响应中包含 Content-Length，那么它的数值必须匹配它返回的内容范围的真实字节数。Date ETag 和/或 Content-Location，假如同样的请求本应该返回200响应。Expires, Cache-Control, 和/或 Vary，假如其值可能与之前相同变量的其他响应对应的值不同的话。假如本响应请求使用了 If-Range 强缓存验证，那么本次响应不应该包含其他实体头；假如本响应的请求使用了 If-Range 弱缓存验证，那么本次响应禁止包含其他实体头；这避免了缓存的实体内容和更新了的实体头信息之间的不一致。否则，本响应就应当包含所有本应该返回200响应中应当返回的所有实体头部域。假如 ETag 或 Last-Modified 头部不能精确匹配的话，则客户端缓存应禁止将206响应返回的内容与之前任何缓存过的内容组合在一起。任何不支持 Range 以及 Content-Range 头的缓存都禁止缓存206响应返回的内容。	
207	由WebDAV(RFC 2518)扩展的状态码，代表之后的消息体将是一个XML消息，并且可能依照之前子请求数量的不同，包含一系列独立的响应代码。	
300	被请求的资源有一系列可供选择的回馈信息，每个都有自己特定的地址和浏览器驱动的商议信息。用户或浏览器能够自行选择一个首选的地址进行重定向。除非这是一个 HEAD 请求，否则该响应应当包括一个资源特性及地址的列表的实体，以便用户或浏览器从中选择最合适的选择。这个实体的格式由 Content-Type 定义的格式所决定。浏览器可能根据响应的格式以及浏览器自身能力，自动作出最合适的选择。当然，RFC 2616规范并没有规定这样的自动选择该如何进行。如果服务器本身已经有了首选的回馈选择，那么在 Location 中应当指明这个回馈的 URI；浏览器可能会将这个 Location 值作为自动重定向的地址。此外，除非额外指定，否则这个响应也是可缓存的。	
301	被请求的资源已永久移动到新位置，并且将来任何对此资源的引用都应该使用本响应返回的若干个 URI 之一。如果可能，拥有链接编辑功能的客户端应当自动把请求的地址修改为从服务器反馈回来的地址。除非额外指定，否则这个响应也是可缓存的。新的永久性的 URI 应当在响应的 Location 域中返回。除非这是一个 HEAD 请求，否则响应的实体中应当包含指向新的 URI 的超链接及简短说明。如果这不是一个 GET 或者 HEAD 请求，因此浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。注意：对于某些使用 HTTP/1.0 协议的浏览器，当它们发送的 POST 请求得到了一个301响应的话，接下来的重定向请求将会变成 GET 方式。	
302	请求的资源现在临时从不同的 URI 响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。只有在Cache-Control或Expires中进行了指定的情况下，这个响应才是可缓存的。新的临时性的 URI 应当在响应的 Location 域中返回。除非这是一个 HEAD 请求，否则响应的实体中应当包含指向新的 URI 的超链接及简短说明。如果这不是一个 GET 或者 HEAD 请求，那么浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。注意：虽然 RFC 1945和RFC 2068规范不允许客户端在重定向时改变请求的方法，但是很多现存的浏览器将302响应视作为303响应，并且使用 GET 方式访问在 Location 中规定的 URI，而无视原先请求的方法。状态码303和307被添加了进来，用以明确服务器期待客户端进行何种反应。	
	对应当前请求的响应可以在另一个 URI 上被找到，而且客户端应当采用 GET 的方式访问那个资源。这个方法的存在主要是为了允许由脚本激活的POST请求输出重定向到一个新的资源。这个新的 URI 不是原始资源的替代引用。同时，303响应禁止被缓存。当然，第二个请求（重定向）可能被缓存。新的 URI 应当在响应的 Location 域中返回。除非这是一个 HEAD 请求，否则响应的实体中应当包含指向新的	

	URI 的超链接及简短说明。 注意：许多 HTTP/1.1 版以前的 浏览器不能正确理解303状态。如果需要考虑与这些浏览器之间的互动，302状态码应该可以胜任，因为大多数的浏览器处理302响应时的方式恰恰就是上述规范要求客户端处理303响应时应当做的。
304	如果客户端发送了一个带条件的 GET 请求且该请求已被允许，而文档的内容（自上次访问以来或者根据请求的条件）并没有改变，则服务器应当返回这个状态码。304响应禁止包含消息体，因此始终以消息头后的第一个空行结尾。 该响应必须包含以下的头信息： Date，除非这个服务器没有时钟。假如没有时钟的服务器也遵守这些规则，那么代理服务器以及客户端可以自行将 Date 字段添加到接收到的响应头中去（正如RFC 2068中规定的一样），缓存机制将会正常工作。 ETag 和/或 Content-Location，假如同样的请求本应返回200响应。 Expires, Cache-Control, 和/或 Vary，假如其值可能与之前相同变量的其他响应对应的值不同的话。 假如本响应请求使用了强缓存验证，那么本次响应不应该包含其他实体头；否则（例如，某个带条件的 GET 请求使用了弱缓存验证），本次响应禁止包含其他实体头；这避免了缓存了的实体内容和更新了的实体头信息之间的不一致。 假如某个304响应指明了当前某个实体没有缓存，那么缓存系统必须忽视这个响应，并且重复发送不包含限制条件的请求。 假如接收到一个要求更新某个缓存条目的304响应，那么缓存系统必须更新整个条目以反映所有在响应中被更新的字段的值。
305	被请求的资源必须通过指定的代理才能被访问。Location 域中将给出指定的代理所在的 URI 信息，接收者需要重复发送一个单独的请求，通过这个代理才能访问相应资源。只有原始服务器才能建立305响应。 注意：RFC 2068中没有明确305响应是为了重定向一个单独的请求，而且只能被原始服务器建立。忽视这些限制可能导致严重的安全后果。
306	在最新版的规范中，306状态码已经不再被使用。
307	请求的资源现在临时从不同的URI 响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。只有在Cache-Control或Expires中进行了指定的情况下，这个响应才是可缓存的。 新的临时性的URI 应当在响应的 Location 域中返回。除非这是一个HEAD 请求，否则响应的实体中应当包含指向新的URI 的超链接及简短说明。因为部分浏览器不能识别307响应，因此需要添加上述必要信息以便用户能够理解并向新的 URI 发出访问请求。 如果这不是一个GET 或者 HEAD 请求，那么浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。
400	1、语义有误，当前请求无法被服务器理解。除非进行修改，否则客户端不应该重复提交这个请求。 2、请求参数有误。
401	当前请求需要用户验证。该响应必须包含一个适用于被请求资源的 WWW-Authenticate 信息头用以询问用户信息。客户端可以重复提交一个包含恰当的 Authorization 头信息的请求。如果当前请求已经包含了 Authorization 证书，那么401响应代表着服务器验证已经拒绝了那些证书。如果401响应包含了与前一个响应相同的身份验证询问，且浏览器已经至少尝试了一次验证，那么浏览器应当向用户展示响应中包含的实体信息，因为这个实体信息中可能包含了相关诊断信息。参见RFC 2617。
402	该状态码是为了将来可能的需求而预留的。
403	服务器已经理解请求，但是拒绝执行它。与401响应不同的是，身份验证并不能提供任何帮助，而且这个请求也不应该被重复提交。如果这不是一个 HEAD 请求，而且服务器希望能够讲清楚为何请求不能被执行，那么就应该在实体内描述拒绝的原因。当然服务器也可以返回一个404响应，假如它不希望让客户端获得任何信息。
404	请求失败，请求所希望得到的资源未被在服务器上发现。没有信息能够告诉用户这个状况到底是暂时的还是永久的。假如服务器知道情况的话，应当使用410状态码来告知旧资源因为某些内部的配置机制问题，已经永久的不可用，而且没有任何可以跳转的地址。404这个状态码被广泛应用于当服务器不想揭示到底为何请求被拒绝或者没有其他适合的响应可用的情况下。
405	请求行中指定的请求方法不能被用于请求相应的资源。该响应必须返回一个Allow 头信息用以表示出当前资源能够接受的请求方法的列表。 鉴于 PUT, DELETE 方法会对服务器上的资源进行写操作，因而绝大部分的网页服务器都不支持或者在默认配置下不允许上述请求方法，对于此类请求均会返回405错误。
406	请求的资源的内容特性无法满足请求头中的条件，因而无法生成响应实体。 除非这是一个 HEAD 请求，否则该响应就应当返回一个包含可以让用户或者浏览器从中选择最合适的实体特性以及地址列表的实体。实体的格式由 Content-Type 头中定义的媒体类型决定。浏览器可以根据格式及自身能力自行作出最佳选择。但是，规范中并没有定义任何作出此类自动选择的标准。
407	与401响应类似，只不过客户端必须在代理服务器上进行身份验证。代理服务器必须返回一个 Proxy-Authenticate 用以进行身份询问。客户端可以返回一个 Proxy-Authorization 信息头用以验证。参见RFC 2617。
408	请求超时。客户端没有在服务器预备等待的时间内完成一个请求的发送。客户端可以随时再次提交这一请求而无需进行任何更改。
	由于和被请求的资源的当前状态之间存在冲突，请求无法完成。这个代码只允许用在这样的情况下才能

409	被使用：用户被认为能够解决冲突，并且会重新提交新的请求。该响应应当包含足够的信息以便用户发现冲突的源头。冲突通常发生于对 PUT 请求的处理中。例如，在采用版本检查的环境下，某次 PUT 提交的对特定资源的修改请求所附带的版本信息与之前的某个（第三方）请求向冲突，那么此时服务器就应该返回一个409错误，告知用户请求无法完成。此时，响应实体中很可能会包含两个冲突版本之间的差异比较，以便用户重新提交归并以后的新版本。
410	被请求的资源在服务器上已经不再可用，而且没有任何已知的转发地址。这样的状况应当被认为是永久性的。如果可能，拥有链接编辑功能的客户端应当在获得用户许可后删除所有指向这个地址的引用。如果服务器不知道或者无法确定这个状况是否是永久的，那么就应该使用404状态码。除非额外说明，否则这个响应是可缓存的。410响应的主要目的是帮助网站管理员维护网站，通知用户该资源已经不再可用，并且服务器拥有者希望所有指向这个资源的远端连接也被删除。这类事件在限时、增值服务中很普遍。同样，410响应也被用于通知客户端在当前服务器站点上，原本属于某个个人的资源已经不再可用。当然，是否需要把所有永久不可用的资源标记为'410 Gone'，以及是否需要保持此标记多长时间，完全取决于服务器拥有者。
411	服务器拒绝在没有定义 Content-Length 头的情况下接受请求。在添加了表明请求消息体长度的有效 Content-Length 头之后，客户端可以再次提交该请求。
412	服务器在验证在请求的头字段中给出先决条件时，没能满足其中的一个或多个。这个状态码允许客户端在获取资源时在请求的元信息（请求头字段数据）中设置先决条件，以此避免该请求方法被应用到其希望的内容以外的资源上。
413	服务器拒绝处理当前请求，因为该请求提交的实体数据大小超过了服务器愿意或者能够处理的范围。此种情况下，服务器可以关闭连接以免客户端继续发送此请求。如果这个状况是临时的，服务器应当返回一个 Retry-After 的响应头，以告知客户端可以在多少时间以后重新尝试。
414	请求的URI长度超过了服务器能够解释的长度，因此服务器拒绝对该请求提供服务。这比较少见，通常的情况包括：本应使用POST方法的表单提交变成了GET方法，导致查询字符串（Query String）过长。重定向URI“黑洞”，例如每次重定向把旧的 URI 作为新的 URI 的一部分，导致在若干次重定向后 URI 超长。客户端正在尝试利用某些服务器中存在的安全漏洞攻击服务器。这类服务器使用固定长度的缓冲读取或操作请求的 URI，当 GET 后的参数超过某个数值后，可能会产生缓冲区溢出，导致任意代码被执行[1]。没有此类漏洞的服务器，应当返回414状态码。
415	对于当前请求的方法和所请求的资源，请求中提交的实体并不是服务器中所支持的格式，因此请求被拒绝。
416	如果请求中包含了 Range 请求头，并且 Range 中指定的任何数据范围都与当前资源的可用范围不重合，同时请求中又没有定义 If-Range 请求头，那么服务器就应当返回416状态码。假如 Range 使用的是字节范围，那么这种情况就是指请求指定的所有数据范围的首字节位置都超过了当前资源的长度。服务器也应当在返回416状态码的同时，包含一个 Content-Range 实体头，用以指明当前资源的长度。这个响应也被禁止使用 multipart/byteranges 作为其 Content-Type。
417	在请求头 Expect 中指定的预期内容无法被服务器满足，或者这个服务器是一个代理服务器，它有明显的证据证明在当前路由的下一个节点上，Expect 的内容无法被满足。
421	从当前客户端所在的IP地址到服务器的连接数超过了服务器许可的最大范围。通常，这里的IP地址指的是从服务器上看到的客户端地址（比如用户的网关或者代理服务器地址）。在这种情况下，连接数的计算可能涉及到不止一个终端用户。
422	从当前客户端所在的IP地址到服务器的连接数超过了服务器许可的最大范围。通常，这里的IP地址指的是从服务器上看到的客户端地址（比如用户的网关或者代理服务器地址）。在这种情况下，连接数的计算可能涉及到不止一个终端用户。
423	请求格式正确，但是由于含有语义错误，无法响应。（RFC 4918 WebDAV）423 Locked 当前资源被锁定。（RFC 4918 WebDAV）
424	由于之前的某个请求发生的错误，导致当前请求失败，例如 PROPPATCH。（RFC 4918 WebDAV）
425	在WebDav Advanced Collections 草案中定义，但是未出现在《WebDAV 顺序集协议》（RFC 3658）中。
426	客户端应当切换到TLS/1.0。（RFC 2817）
449	由微软扩展，代表请求应当在执行完适当的操作后进行重试。
500	服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。一般来说，这个问题都会在服务器的程序码出错时出现。
501	服务器不支持当前请求所需要的某个功能。当服务器无法识别请求的方法，并且无法支持其对任何资源的请求。

502	作为网关或者代理工作的服务器尝试执行请求时，从上游服务器接收到无效的响应。
503	由于临时的服务器维护或者过载，服务器当前无法处理请求。这个状况是临时的，并且将在一段时间以后恢复。如果能够预计延迟时间，那么响应中可以包含一个 <code>Retry-After</code> 头用以标明这个延迟时间。如果没有给出这个 <code>Retry-After</code> 信息，那么客户端应当以处理500响应的方式处理它。注意：503状态码的存在并不意味着服务器在过载的时候必须使用它。某些服务器只不过是希望拒绝客户端的连接。
504	作为网关或者代理工作的服务器尝试执行请求时，未能及时从上游服务器（URI标识出的服务器，例如 HTTP、FTP、LDAP）或者辅助服务器（例如DNS）收到响应。注意：某些代理服务器在DNS查询超时时会返回400或者500错误
505	服务器不支持，或者拒绝支持在请求中使用的 HTTP 版本。这暗示着服务器不能或不愿使用与客户端相同的版本。响应中应当包含一个描述了为何版本不被支持以及服务器支持哪些协议的实体。
506	由《透明内容协商协议》（RFC 2295）扩展，代表服务器存在内部配置错误：被请求的协商变元资源被配置为在透明内容协商中使用自己，因此在一个协商处理中不是一个合适的重点。
507	服务器无法存储完成请求所必须的内容。这个状况被认为是临时的。WebDAV (RFC 4918)
509	服务器达到带宽限制。这不是一个官方的状态码，但是仍被广泛使用。
510	获取资源所需要的策略并没有满足。 (RFC 2774)



OSI分层(7层)

- 物理层
- 数据链路层
- 网络层
- 传输层
- 会话层
- 表示层
- 应用层。

TCP/IP分层(4层)

- 网络接口层
- 网际层
- 运输层
- 应用层

五层协议(5层)

- 物理层
- 数据链路层

- 网络层
- 运输层
- 应用层

栈（Stack）和队列（Queue）是两种操作受限的线性表。

相同点

- 都是线性结构。
- 都可以通过顺序结构和链式结构实现。
- 插入与删除的时间复杂度都是 $O(1)$,在空间复杂度上两者也一样。

不同点

- 栈的插入和删除操作都是在一端进行的，而队列的操作却是在两端进行的。
- 队列先进先出，栈先进后出。
- 栈只允许在表尾一端进行插入和删除，而队列只允许在表尾一端进行插入，在表头一端进行删除
- 应用场景不同；常见栈的应用场景包括括号问题的求解，表达式的转换和求值，函数调用和递归实现，深度优先搜索遍历等；常见的队列的应用场景包括计算机系统中各种资源的管理，消息缓冲器的管理和广度优先搜索遍历等。
- 顺序栈能够实现多栈空间共享，而顺序队列不能。

PHP-FPM

PHP-FPM 即 PHP-FastCGI Process Manager，它是 FastCGI 的实现，并提供了进程管理的功能。进程包含 master 进程和 worker 进程两种；master 进程只有一个，负责监听端口，接收来自服务器的请求，而 worker 进程则一般有多个（具体数量根据实际需要进行配置），每个进程内部都会嵌入一个 PHP 解释器，是代码真正执行的地方。

nginx与php-fpm通信方式

在 Linux 上，nginx 与 php-fpm 的通信有 tcp socket 和 unix socket 两种方式。

tcp socket

tcp socket 的优点是可以跨服务器，当 nginx 和 php-fpm 不在同一台机器上时，只能使用这种方式

unix socket

[!NOTE|label:注意] 在使用 unix socket 方式连接时，由于 socket 文件本质上是一个文件，存在权限控制的问题，所以需要注意 nginx 进程的权限与 php-fpm 的权限问题，不然会提示无权限访问。（在各自的配置文件里设置用户）

Unix socket 又叫 IPC(inter-process communication 进程间通信) socket，用于实现同一主机上的进程间通信，这种方式需要在 nginx 配置文件中填写 php-fpm 的 socket 文件位置。

区别

由于 Unix socket 不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。所以其效率比 tcp socket 的方式要高，可减少不必要的 tcp 开销。不过，unix socket 高并发时不稳定，连接数爆发时，会产生大量的长时缓存，在没有面向连接协议的支撑下，大数据包可能会直接出错不返回异常。而 tcp 这样的面向连接的协议，可以更好的保证通信的正确性和完整性。

在应用中的选择

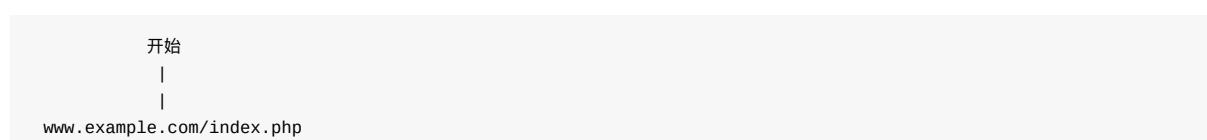
如果是在同一台服务器上运行的 nginx 和 php-fpm，且并发量不高（不超过1000），选择 unix socket，以提高 nginx 和 php-fpm 的通信效率。如果是面临高并发业务，则考虑选择使用更可靠的 tcp socket，以负载均衡、内核优化等运维手段维持效率。

若并发较高但仍想用 unix socket 时，可通过以下方式提高 unix socket 的稳定性。

- 1) 将sock文件放在 /dev/shm 目录下，此目录下将 sock 文件放在内存里面，内存的读写更快。
- 2) 提高 backlog

backlog 默认值 128，1024 这个值最好换算成自己正常的 QPS，配置如下。

Nginx&FPM处理流程



```
|  
|  
nginx 80 端口  
|  
|  
nginx 加载 fastcgi 模块  
|  
|  
反向代理到 fpm 监听的 9000 端口  
|  
|  
fpm 处理请求并返回至 nginx  
|  
|  
nginx 接收并返回客户端  
|  
|  
结束
```

在linux或者unix操作系统中在系统的引导的时候会开启很多服务，这些服务就叫做守护进程。为了增加灵活性，root可以选择系统开启的模式，这些模式叫做运行级别，每一种运行级别以一定的方式配置系统。守护进程是脱离于终端并且在后台运行的进程。守护进程脱离于终端是为了避免进程在执行过程中的信息在任何终端上显示并且进程也不会被任何终端所产生的终端信息所打断。

守护进程及其特性

守护进程最重要的特性是后台运行。在这一点上DOS下的常驻内存程序TSR与之相似。其次，守护进程必须与其运行前的环境隔离开来。这些环境包括未关闭的文件描述符，控制终端，会话和进程组，工作目录以及文件创建掩模等。这些环境通常是守护进程从执行它的父进程（特别是shell）中继承下来的。最后，守护进程的启动方式有其特殊之处。它可以在Linux系统启动时从启动脚本/etc/rc.d中启动，可以由作业规划进程crond启动，还可以由用户终端（通常也是shell）执行。

守护进程流程处理

- 创建子进程，父进程退出
- 在子进程中创建新会话
- 改变当前目录为根目录
- 重设文件权限掩码
- 关闭文件描述符
- 守护进程退出处理

PHP参考

```
<?php
$fatherFile = fopen('test.txt');
// 重设文件创建掩码
umask( 0 );
// 创建子进程，终止父进程
$pid = pcntl_fork();
if( $pid < 0 ){
    exit('fork error.');
} else if( $pid > 0 ) {
    exit();
}
// 在子进程中创建新会话
if( !posix_setsid() ){
    exit('setsid error.');
}
$pid = pcntl_fork();
if( $pid < 0 ){
    exit('fork error.');
} else if( $pid > 0 ) {
    exit();
}
// 改变工作目录
chdir( '/tmp' );

// 关闭文件描述符
fclose($fatherFile);
```

C参考

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<fcntl.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include <signal.h>
#define MAXFILE 65535

volatile sig_atomic_t _running = 1;

void sigterm_handler(int arg)
{
    _running = 0;
}

int main()
{
    pid_t pc, pid;
    int i, fd, len;
    char *buf = "this is a Dameon\n";
    len = strlen(buf);
    //第一步
    pc = fork();
    if(pc < 0)
    {
        printf("error fork\n");
        exit(1);
    }
    else if(pc > 0)
    {
        exit(0);
    }
    //第二步
    setsid();
    pid = fork(); //与终端完全脱离[1]
    if (pid < 0)
    {
        perror("fork error");
    }
    if (pid > 0)
    {
        exit(0);
    }
    //第三步
    chdir("/");
    //第四步
    umask(0);
    //第五步
    for(i = 0;i < MAXFILE; i++)
    {
        close(i);
    }
    signal(SIGTERM, sigterm_handler);
    while( _running )
    {
        if((fd = open("/tmp/dameon.log", O_CREAT | O_WRONLY | O_APPEND, 0600)) < 0)
        {
            perror("open");
            exit(1);
        }
        write(fd, buf, len);
        close(fd);
        usleep(10 * 1000); //10毫秒
    }
}

```

- [Linux守护进程的原理与实现](#)
- [Linux守护进程的编程方法](#)
- [PHP实现守护进程](#)

1、存储位置不同

- cookie的数据信息存放在客户端浏览器上。
- session的数据信息存放在服务器上。

2、存储容量不同

- 单个cookie保存的数据<=4KB，一个站点最多保存20个Cookie。
- 对于session来说并没有上限，但出于对服务器端的性能考虑，session内不要存放过多的东西，并且设置session删除机制。

3、存储方式不同

- cookie中只能保管ASCII字符串，并需要通过编码方式存储为Unicode字符或者二进制数据。
- session中能够存储任何类型的数据，包括且不限于string, integer, list, map等。

4、隐私策略不同

- cookie对客户端是可见的，别有用心的人可以分析存放在本地的cookie并进行cookie欺骗，所以它是不安全的。
- session存储在服务器上，对客户端是透明的，不存在敏感信息泄漏的风险。

5、有效期上不同

- 开发可以通过设置cookie的属性，达到使cookie长期有效的效果。
- session依赖于名为JSESSIONID的cookie，而cookie JSESSIONID的过期时间默认为-1，只需关闭窗口该session就会失效，因而session不能达到长期有效的效果。

6、服务器压力不同

- cookie保管在客户端，不占用服务器资源。对于并发用户十分多的网站，cookie是很好的选择。
- session是保管在服务器端的，每个用户都会产生一个session。假如并发访问的用户十分多，会产生十分多的session，耗费大量的内存。

7、浏览器支持不同

1. 假如客户端浏览器不支持cookie：

- cookie是需要客户端浏览器支持的，假如客户端禁用了cookie，或者不支持cookie，则会话跟踪会失效。关于WAP上的应用，常规的cookie就派不上用场了。
- 运用session需要使用URL地址重写的方式。一切用到session程序的URL都要进行URL地址重写，否则session会话跟踪还会失效。

2. 假如客户端支持cookie：

- cookie既能够设为本浏览器窗口以及子窗口内有效，也能够设为一切窗口内有效。

- session只能在本窗口以及子窗口内有效。

8、跨域支持上不同

- cookie支持跨域名访问。
- session不支持跨域名访问。

Cookie的工作机制

1. 服务器向客户端响应请求的时候，会在响应头中设置set-cookie的值，其值的格式通常是name = value的格式
2. 浏览器将 cookie 保存下来
3. 每次请求浏览器都会自动将 cookie 发向服务器
4. cookie最初是在客户端用于存储会话信息的。

Session的工作机制

1. 当客户端第一次请求session对象时，服务器会创建一个session，并通过特殊算法算出一个session的ID，用来标识该session对象，然后将这个session序列放置到set-cookie中发送给浏览器
2. 浏览器下次发请求的时候，这个sessionId会被放置在请求头中，和cookie一起发送回来
3. 服务器再通过内存中保存的sessionId跟cookie中保存的sessionId进行比较，并根据ID在内存中找到之前创建的session对象，提供给请求使用，也就是服务器会通过session保存一个状态记录，浏览器会通过cookie保存状态记录，服务器通过两者的对比实现跟踪状态，这样的做，也极大的避免了cookie被篡改而带来的安全性问题
4. 由于cookie可以被人为的禁止，必须有其他机制以便在cookie被禁止时仍然能够把session id传递回服务器。经常被使用的一种技术叫做URL重写，就是把session id直接附加在URL路径的后面，附加方式也有两种，一种是作为URL路径的附加信息，另一种是作为查询字符串附加在URL后面

问题列表

- SQL 注入

主要就是一些数据没有经过严格的验证，然后直接拼接 SQL 去查询。导致漏洞产生，注入者可提交任何类型的数据，比如 " and 1= 1 or " 等不安全的数据

- 文件系统安全

PHP 可以直接访问文件系统、执行 Shell 命令，这给程序开发提供了强大的支持的同时也可能会带来危险。同样，恰当的过滤和编码可以避免危险。

- 命令行注入

PHP 提供了 exec(), system(), passthru(), shell_exec() 等函数，以及 ` (反引号) 运算符。这些函数可以直接调用命令行系统指令，例如 system('dir c:'); 可以显示 C 盘符下的目录内容，system("ls -al|cat/etc/passwd"); 可以获得 passwd 的内容。假如攻击者能将这些命令注入你的代码并运行，将给系统带来巨大的危害。

- XSS

XSS 又叫 CSS (Cross Site Script)，跨站脚本攻击。它指的是恶意攻击者往 Web 页面里插入恶意 html 代码，当用户浏览该页之时，嵌入其中 Web 里面的 html 代码会被执行，从而达到恶意攻击用户的特殊目的。

- CSRF

CSRF 是跨站请求伪造的缩写，它是攻击者通过一些技术手段欺骗用户去访问曾经认证过的网站并运行一些操作。

- 在生产环境中不正确的错误报告暴露敏感数据

如果你不小心，可能会在生产环境中因为不正确的错误报告泄露了敏感信息，例如：文件夹结构、数据库结构、连接信息与用户信息。

PHP 预编译工作原理

预处理：创建 SQL 语句模板并发送到数据库。预留的值使用参数 "?" 标记。例如：

INSERT INTO MyGuests (firstname, lastname, email) VALUES(?, ?, ?) 数据库解析，编译，对 SQL 语句模板执行查询优化，并存储结果不输出。

执行：最后，将应用绑定的值传递给参数 ("?" 标记)，数据库执行语句。应用可以多次执行语句，如果参数的值不一样。

相比于直接执行 SQL 语句，预处理语句有两个主要优点：

预处理语句大大减少了分析时间，只做了一次查询（虽然语句多次执行）。

绑定参数减少了服务器带宽，你只需要发送查询的参数，而不是整个语句。

预处理语句针对 SQL 注入是非常有用的，因为参数值发送后使用不同的协议，保证了数据的合法性。

处理文件错误的机制上面不同

- require() :如果文件不存在，会报出一个fatal error.脚本停止执行;
- include() : 如果文件不存在，会给出一个 warning，但脚本会继续执行;

php性能

- 对include()来说，在include()执行时文件每次都要进行读取和评估;
- 对require()来说，文件只处理一次(实际上，文件内容替换了require()语句)。

不同的使用弹性

- require的使用方法如 require("./inc.php"); 。通常放在PHP程式的最前面，PHP程式在执行前，就会先读入require所指定引入的档案，使它变成PHP 程式网页的一部份。
- include使用方法如 include("./inc.php"); 。一般是放在流程控制的处理区段中。PHP程式网页在读到 include的档案时，才将它读进来。这种方式，可以把程式执行时的流程简单化。

once

- 带once和不带once的区别主要是:带once的会判断你在加载这个文件之前是否已经加载过了文件，避免重复加载。

- [array-reverse](#)
- [array-flip](#)

方案1

```

class Lock
{
    private $redis=''; #存储redis对象

    public function __construct($host,$port=6379)
    {
        $this->redis=new Redis();
        $this->redis->connect($host,$port);
    }

    /**
     * @desc 加锁方法
     *
     * @param $lockName string | 锁的名字
     * @param $timeout int | 锁的过期时间
     *
     * @return 成功返回identifier/失败返回false
     */
    public function getLock($lockName, $timeout=2)
    {
        $identifier=uniqid();          #获取唯一标识符
        $timeout=ceil($timeout);       #确保是整数
        $end=time()+$timeout;
        while(time()<$end)           #循环获取锁
        {
            if($this->redis->setnx($lockName, $identifier)) #查看$lockName是否被上锁
            {
                $this->redis->expire($lockName, $timeout);
                return $identifier;
            } elseif ($this->redis->ttl($lockName)==-1) {
                $this->redis->expire($lockName, $timeout);
                #检测是否有设置过期时间，没有则加上（假设，客户端A上一步没
                能设置时间就进程奔溃了，客户端B就可检测出来，并设置时间）
            }
            usleep(0.001);             #停止0.001ms
        }
        return false;
    }

    /**
     * @desc 释放锁
     *
     * @param $lockName string | 锁名
     * @param $identifier string | 锁的唯一值
     *
     * @param bool
     */
    public function releaseLock($lockName,$identifier)
    {
        if($this->redis->get($lockName)==$identifier) #判断是锁有没有被其他客户端修改
        {
            $this->redis->multi();
            $this->redis->del($lockName);   #释放锁
            $this->redis->exec();
            return true;
        }
        else
        {
            return false;   #其他客户端修改了锁，不能删除别人的锁
        }
    }

    /**
     * @desc 测试
     */
}

```

```

/*
 * @param $lockName string | 锁名
 */
public function test($lockName)
{
    $start=time();
    for ($i=0; $i < 10000; $i++)
    {
        $identifier=$this->getLock($lockName);
        if($identifier)
        {
            $count=$this->redis->get('count');
            $count=$count+1;
            $this->redis->set('count',$count);
            $this->releaseLock($lockName,$identifier);
        }
    }
    $end=time();
    echo "this OK<br/>";
    echo "执行时间为: ".($end-$start);
}
}

```

方案2

```

class RedisMutexLock
{
    /**
     * 缓存 Redis 连接。
     *
     * @return void
     */
    public static function getRedis()
    {
        // 这行代码请根据自己项目替换为自己的获取 Redis 连接。
        return YCache::getRedisClient();
    }

    /**
     * 获得锁,如果锁被占用,阻塞,直到获得锁或者超时。
     * -- 1、如果 $timeout 参数为 0,则立即返回锁。
     * -- 2、建议 timeout 设置为 0,避免 redis 因为阻塞导致性能下降。请根据实际需求进行设置。
     *
     * @param string $key      缓存KEY。
     * @param int    $timeout   取锁超时时间。单位(秒)。等于0,如果当前锁被占用,则立即返回失败。如果大于0,则反复尝试获取锁
     * 直到达该超时时间。
     * @param int    $lockSecond 锁定时间。单位(秒)。
     * @param int    $sleep      取锁间隔时间。单位(微秒)。当锁为占用状态时。每隔多久尝试去取锁。默认 0.1 秒一次取锁。
     * @return bool 成功:true、失败:false
     */
    public static function lock($key, $timeout = 0, $lockSecond = 20, $sleep = 100000)
    {
        if (strlen($key) === 0) {
            // 请更换为自己项目抛异常的方法。
            YCore::exception(500, '缓存KEY没有设置');
        }
        $start = self::getMicroTime();
        $redis = self::getRedis();
        do {
            // [1] 锁的 KEY 不存在时设置其值并把过期时间设置为指定的时间。锁的值并不重要。重要的是利用 Redis 的特性。
            $acquired = $redis->set("Lock:$key", 1, ['NX', 'EX' => $lockSecond]);
            if ($acquired) {
                break;
            }
            if ($timeout === 0) {

```

```
        break;
    }
    usleep($sleep);
} while (!is_numeric($timeout) || ($self::getMicroTime() < ($start + ($timeout * 1000000))));
return $acquired ? true : false;
}

/**
 * 释放锁
 *
 * @param mixed $key 被加锁的KEY。
 * @return void
 */
public static function release($key)
{
    if (strlen($key) === 0) {
        // 请更换为自己项目抛异常的方法。
        YCore::exception(500, '缓存KEY没有设置');
    }
    $redis = self::getRedis();
    $redis->del("Lock:$key");
}

/**
 * 获取当前微秒。
 *
 * @return bigint
 */
protected static function getMicroTime()
{
    return bcmul(microtime(true), 1000000);
}
}
```

生产者丢数据

确认机制相对于事务机制，最大的好处就是可以异步处理提高吞吐量，不需要额外等待消耗资源。但是两者时候不能同时共存的。

事务机制

[!TIP][label:说明] 与事务机制相关的有三种方法，分别是channel.txSelect设置当前信道为事务模式、channel.txCommit提交事务和channel.txRollback事务回滚。如果事务提交成功，则消息一定是到达了RabbitMQ中，如果事务提交之前由于发送异常或者其他原因，捕获后可以进行channel.txRollback回滚。

```
// 将信道设置为事务模式，开启事务
channel.txSelect();
// 发送持久化消息
channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY, MessageProperties.PERSISTENT_TEXT_PLAIN, "transaction messages"
    .getBytes());
// 事务提交
channel.txCommit();
// 事物回滚
channel.txRollback();
```

消息确认机制

[!TIP][label:说明]

- 生产者将信道设置为confirm确认模式，确认之后所有在信道上的消息将会被指派一个唯一的从1开始的ID，一旦消息被正确匹配到所有队列后，RabbitMQ就会发送一个确认Basic.Ack给生产者（包含消息的唯一ID），生产者便知晓消息是否正确到达目的地了。
- 消息如果是持久化的，那么确认消息会在消息写入磁盘之后发出。RabbitMQ中的deliveryTag包含了确认消息序号，还可以设置multiple参数，表示到这个序号之前的所有消息都已经得到处理。确认机制相对事务机制来说，相比较代码来说比较复杂，但会经常使用，主要有单条确认、批量确认、异步批量确认三种方式。

单条确认

```
// 设置 channel 消息推送为 单条确认模式
channel.confirmSelect();
// 推送消息
channel.basicPublish("exchange", "routingkey", null, "publisher confirm test".getBytes());
// 消息推送失败处理
if (!channel.waitForConfirms()) {
    //publisher confirm failed handle
}
```

批量确认

[!TIP][label:问题] 批量确认comfirm需要解决出现返回的Basic.Nack或者超时情况的消息全部重发怎么解决

```
# 增加一个缓存，将发送成功并且确认Ack之后的消息去除，剩下Nack或者超时的消息
// take ArrayList or BlockingQueue as a cache
List<Object> cache = new ArrayList<>();
// set channel publisher confirm mode
```

```

channel.confirmSelect();
for (int i=0; i < 20; i++) {
    // publish message
    String message = "publisher message["+ i +"]";
    cache.add(message);
    channel.basicPublish("exchange", "routingkey", null, message.getBytes());
    if (channel.waitForConfirms()) {
        // remove message publisher confirm
        cache.remove(i);
    }
    // TODO handle Nack message: republish
}

```

异步批量确认

[!NOTE]lable:说明] 异步确认方式通过在客户端addConfirmListener增加ConfirmListener回调接口，包括handleAck与handleNack处理方法

```

// take as a cache
SortedSet cache = new TreeSet();
// set channel publisher confirm mode
channel.confirmSelect();
for (int i = 0; i < 20; i++) {
    // publish message
    long nextSeqNo = channel.getNextPublishSeqNo();
    String message = "publisher message[" + i + "]";
    cache.add(message);
    channel.basicPublish("exchange", "routingkey", MessageProperties.PERSISTENT_TEXT_PLAIN, message.getBytes());
;
    cache.add(nextSeqNo);
}
// add confirmCalback: handleAck, handleNack
channel.addConfirmListener(new ConfirmListener() {
    @Override
    public void handleAck(long deliveryTag, boolean multiple) {
        if (multiple) {
            // batch remove ack message
            cache.headSet(deliveryTag - 1).clear();
        } else {
            // remove ack message
            cache.remove(deliveryTag);
        }
    }
    @Override
    public void handleNack(long deliveryTag, boolean multiple) {
        // TODO handle Nack message: republish
    }
});

```

参考

- 四种途径提高RabbitMQ传输消息数据的可靠性（一）
- 四种途径提高RabbitMQ传输数据的可靠性（二）
- RabbitMQ如何解决各种情况下丢数据的问题

MQ的作用

1. 解耦：可以很好地屏蔽应用程序及平台之间的特性，充当中间者，松散耦合应用程序及平台，它们彼此不需要了解远程过程调用RPC与网络协议的细节；
2. 异步通信：能提供C/S之间同步与异步连接，在任何时刻都可以将消息进行传送或者存储转发；
3. 可恢复性：当消息接收方宕机或网络不通的情况下，消息转储于MQ中，直到网络恢复或接收方恢复再进行转发；
4. 扩展性：提高消息入队列和处理效率是容易的，只需要另外增加处理过程即可，不需要改变代码，也不需要调节参数。
5. 顺序性：由于大部分MQ支持队列模式，自然也就能保证一定的数据处理顺序（事实上是不准确的，很有局限性。
消息的顺序性取决于很多因素）
6. 缓冲：MQ通过一个缓冲层来帮助任务最高效率执行，写入MQ的处理会尽可能快速。

RabbitMQ的消息处理过程



RabbitMQ的四种交换机

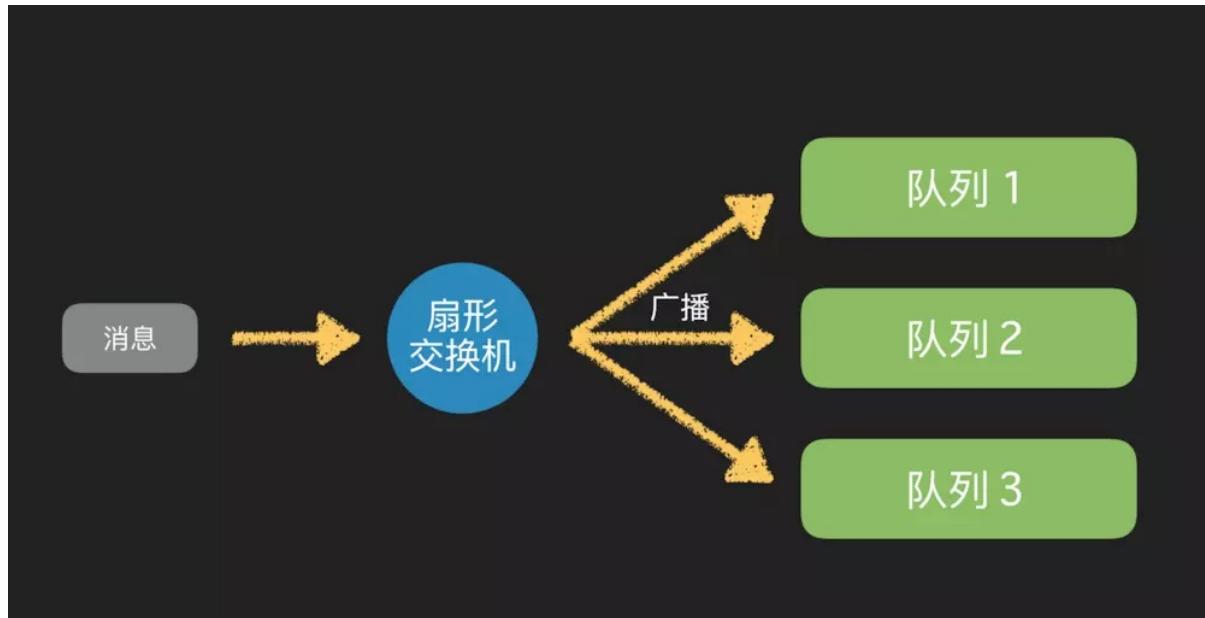
直连交换机：Direct exchange

[!TIP|label:适用场景] 有优先级的任务，根据任务的优先级把消息发送到对应的队列，这样可以指派更多的资源去处理高优先级的队列。



扇形交换机：Fanout exchange

广播消息。扇形交换机会把能接收到的消息全部发送给绑定在自己身上的队列。因为广播不需要“思考”，所以扇形交换机处理消息的速度也是所有的交换机类型里面最快的。



主题交换机：Topic exchange

RabbitMQ提供了一种主题交换机，发送到主题交换机上的消息需要携带指定规则的routing_key，主题交换机会根据这个规则将数据发送到对应的(多个)队列上。主题交换机的routing_key需要有一定的规则，交换机和队列的binding_key需要采用#.……的格式，每个部分用.分开

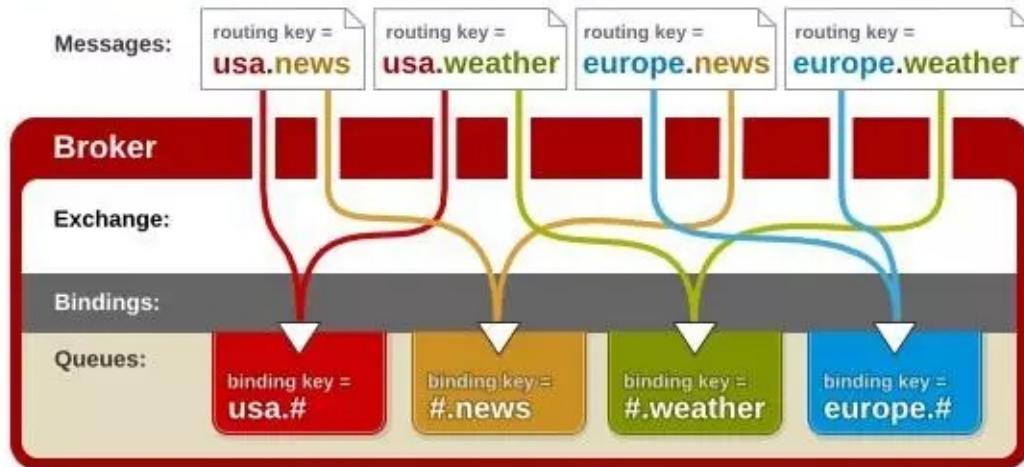
- *表示一个单词

- 表示任意数量（零个或多个）单词。

- *表示一个单词

• 表示任意数量（零个或多个）单词。

Topic Exchange



首部交换机：Headers exchange

- 首部交换机是忽略routing_key的一种路由方式。路由器和交换机路由的规则是通过Headers信息来交换的，这个有点像HTTP的Headers。将一个交换机声明成首部交换机，绑定一个队列的时候，定义一个Hash的数据结构，消息发送的时候，会携带一组hash数据结构的信息，当Hash的内容匹配上的时候，消息就会被写入队列。
- 绑定交换机和队列的时候，Hash结构中要求携带一个键“x-match”，这个键的Value可以是any或者all，这代表消息携带的Hash是需要全部匹配(all)，还是仅匹配一个键(any)就可以了。相比直连交换机，首部交换机的优势是匹配的规则不被限定为字符串(string)。

生产者流程

1. 生产者连接到RabbitMQ Broker，建立Connection，开启信道Channel (Connection与Channel概念下面会介绍)
2. 生产者声明一个交换器，设置相关属性。
3. 生产者声明一个队列并设置相关属性
4. 生产者通过路由键将交换器和队列绑定起来
5. 生产者发送消息到RabbitMQ Broker，包括路由键、交换器信息等
6. 相应的交换器根据路由键查找匹配的队列
7. 如果找到则消息存入相应队列中
8. 如果没找到则根据配置的属性丢弃或者回退给生产者
9. 关闭信道
10. 关闭连接

消费者流程

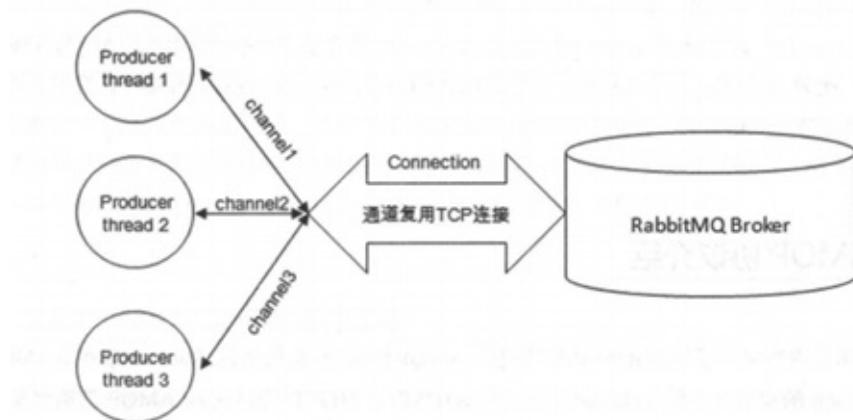
1. 消费者连接到RabbitMQ Broker，建立Connection，开启Channel
2. 消费者向RabbitMQ Broker请求消费相应队列中消息，可能会设置相应的回调函数。

Connection与Channel概念

- Connection: 实际就是一条TCP连接，TCP一旦建立起来，客户端紧接着可以创建AMQP信道。
- Channel: 每个Channel都有唯一的ID，都是建立在Connection上的虚拟连接，RabbitMQ处理每条AMQP指令都是通过信道完成的

[!TIP|label:单TCP复用连接与多信道的优势]

1. 为什么TCP连接只有一条，而每个生产者都会创建一条唯一的信道呢？想象下，实际情况，会有很多的生产者生产消息，多个消费者消费消息，那么就不得不创建多个线程，建立多个TCP连接。多个TCP连接的建立必然会对操作系统性能消耗较高，也不方便管理。从而选择一种类似于NIO（非阻塞I/O, Non-blocking I/O）技术是很有必要的，多信道的在TCP基础上的建立就是这么实现的。
2. 每个线程都有自己的一个信道，复用了Connection的TCP连接，信道之间相互独立，相互保持神秘，节约TCP连接资源，当然本身信道的流量很大的话，也可以创建多个适当的Connection的TCP连接，需要根据具体业务情况制定。



参考

- RabbitMQ的四种交换机
- RabbitMQ是如何运转的？
- 认识RabbitMQ交换机模型

RabbitMQ集群实现方式

主备模式(cluster)

- 不支持跨网段，用于同一个网段内的局域网
- 可以随意的动态增加或者减少
- 节点之间需要运行相同版本的RabbitMQ和Erlang

也称为 Warren (兔子窝) 模式。实现 rabbitMQ 的高可用集群，一般在并发和数据量不高的情况下，这种模式非常的好用且简单。也是一个主/备方案，主节点提供读写，备用节点不提供读写。如果主节点挂了，就切换到备用节点，原来的备用节点升级为主节点提供读写服务，当原来的主节点恢复运行后，原来的主节点就变成备用节点，和 activeMQ 利用 zookeeper 做主/备一样，也可以一主多备。

镜像模式(mirror)

- RabbitMQ的Cluster集群模式一般分为两种，普通模式和镜像模式。

将需要消费的队列变为镜像队列，存在于多个节点，这样就可以实现RabbitMQ的HA高可用性。作用就是消息实体会主动在镜像节点之间实现同步，而不是像普通模式那样，在consumer消费数据时临时读取。缺点就是，集群内部的同步通讯会占用大量的网络带宽。

RabbitMQ主从之间的数据复制是异步的，但是在rabbitmq中不会出现mysql那种丢数据的情况，这是因为rabbitmq的接口也是异步的，主收到一条消息写入本地存储，然后在发起写入从的请求。当所有从写入成功后，主才会给client返回ack说这次写入成功了。所以可以看出，虽然rabbitmq的主从复制是异步的，但是并且不会出现mysql丢数据的场景。只要客户端收到ack，就说明这条消息已经写入主和从了。

多活模式(federation)

- 应用于广域网，允许单台服务器上的交换机或队列接收发布到另一台服务器上交换机或队列的消息，可以是单独机器或集群。federation队列类似于单向点对点连接，消息会在联盟队列之间转发任意次，直到被消费者接受。通常使用federation来连接internet上的中间服务器，用作订阅分发消息或工作队列。

federation 插件是一个不需要构建 cluster，而在 brokers 之间传输消息的高性能插件，federation 插件可以在 brokers 或者 cluster 之间传输消息，连接的双方可以使用不同的 users 和 virtual hosts，双方也可以使用不同版本的 rabbitMQ 和 erlang。federation 插件使用 AMQP 协议通信，可以接受不连续的传输。federation 不是建立在集群上的，而是建立在单个节点上的

远程模式(shovel)

- 连接方式与federation的连接方式类似，但它工作在更低层次。可以应用于广域网。

远程模式可以实现双活的一种模式，简称 shovel 模式，所谓的 shovel 就是把消息进行不同数据中心的复制工作，可以跨地域的让两个 MQ 集群互联，远距离通信和复制。Shovel 就是我们可以把消息进行数据中心的复制工作，我们可以跨地域的让两个 MQ 集群互联。

参考

- RabbitMQ 的4种集群架构
- RabbitMQ高可用原理
- RabbitMQ主备复制是异步还是同步?
- RabbitMQ集群搭建-镜像模式

```
$demo_array = array(23,15,43,25,54,2,6,82,11,5,21,32,65);
// 第一层for循环可以理解为从数组中键为0开始循环到最后一个
for ($i=0;$i<count($demo_array);$i++) {
    // 第二层将从键为$i的地方循环到数组最后
    for ($j=$i+1;$j<count($demo_array);$j++) {
        // 比较数组中相邻两个值的大小
        if ($demo_array[$i] > $demo_array[$j]) {
            $tmp          = $demo_array[$i]; // 这里的$tmp是临时变量
            $demo_array[$i] = $demo_array[$j]; // 第一次更换位置
            $demo_array[$j] = $tmp;           // 完成位置互换
        }
    }
}
```

```
function quick_sort($arr){  
    //先判断是否需要继续进行  
    $length = count($arr);  
    if($length <= 1) {  
        return $arr;  
    }  
    //选择第一个元素作为基准  
    $base_num = $arr[0];  
    //遍历除了标尺外的所有元素，按照大小关系放入两个数组内  
    //初始化两个数组  
    $left_array = array(); //小于基准的  
    $right_array = array(); //大于基准的  
    for($i=1; $i<$length; $i++) {  
        if($base_num > $arr[$i]) {  
            //放入左边数组  
            $left_array[] = $arr[$i];  
        } else {  
            //放入右边  
            $right_array[] = $arr[$i];  
        }  
    }  
    //再分别对左边和右边的数组进行相同的排序处理方式递归调用这个函数  
    $left_array = quick_sort($left_array);  
    $right_array = quick_sort($right_array);  
    //合并  
    return array_merge($left_array, array($base_num), $right_array);;  
}
```