redis

redis为什么高效?

1. 纯内存缓存数据库
2. 使用单线程，避免了多线程带来的频繁的上下文切换
3. 使用了高效的数据结构体
4. 客户端和服务端使用了非阻塞的IO多路复用模型

redis支持哪些数据结构?

1. 字符串 string
2. 列表 list
3. 哈希表 hash
4. 集合 set
5. 有序集合 Sorted Set

高级数据结构有：HyperLogLog/BitMap/BloomFilter/GeoHash

redis典型使用场景有哪些?

1. 队列

    1. 使用lpush/brpop或者blpop/rpush

        优点：使用简单， 缺点：若消费过慢，容易出现热点数据，不支持消费确认，不支持重复消费和多次消费

    2. 使用SUBSCRIBE/PUBLISH 发布订阅模式

        优点：使用简单，缺点：不支持持久化，容易丢数据。若消息者异常，则在异常这段时间，消息是无法被该消费者消费到的。

    3. 使用有序集合（Sorted-Set）实现延迟队列

    4. 使用Stream类型

        优点：redis5.0支持，近乎完美。1.支持阻塞或非阻塞式读取 2. 支持消费组模式，从而支持多次消费 2. 支持消息队列监控

2. 排行榜

3. 自动补全

4. 分布式锁

    为了避免单点故障，可以使用redlock。

    redlock原理：不能只在一个redis实例上创建锁，应该是在多个redis实例上创建锁，n/2 + 1，必须在大多数redis节点上都成功创建锁，才能算这个整体的RedLock加锁成功，避免说仅仅在一个redis实例

上加锁而带来的问题。

5. UV统计

   使用HyperLogLog数据结构实现：

```
redis 127.0.0.1:6379> PFADD runoobkey "redis"
1) (integer) 1
redis 127.0.0.1:6379> PFADD runoobkey "mongodb"
1) (integer) 1
redis 127.0.0.1:6379> PFADD runoobkey "mysql"
1) (integer) 1
redis 127.0.0.1:6379> PFCOUNT runoobkey
(integer) 3
```

6. 去重过滤

   基于bitmap实现布隆过滤器

7. 限流器

8. 用户签到/用户在线状态/活跃用户统计等统计功能

   基于位图bitmap实现

redis中key有哪些过期策略?

1. 惰性删除策略

   当访问key时候，再检查key是否过期

2. 定期删除策略

   Redis 会将每个设置了过期时间的 key 放入到一个独立的字典中，默认每 100ms 进行一次过期扫描：

   1. 从过期key字典中随机 20 个 key，删除这 20 个 key 中已经过期的 key
   2. 如果过期的 key 比率超过 1/4，那就重复步骤 1
   3. 扫描处理时间默认不会超过 25ms

redis中最大内存策略有哪些?

redis中最大内存是由maxmemory参数配置的，当内存使用达到设置的最大内存上限时候，会执行最大内存策略：

1. noeviction 一直写，直到可用内存使用完，写不进数据
2. volatile
    - volatile-ttl 设置了过期时间，ttl时间越短的key越优先被淘汰
    - volatile-lru 基于LRU规则淘汰删除设置了过期时间的key
    - volatile-random 随机淘汰过期集合中的key
3. allkeys
    - allkeys-lru 基于LRU规则淘汰所有key
    - allkeys-random 随机淘汰

redis中持久化方式有哪些?

RDB

RDB是Snapshot快照存储，是redis默认的持久化方式，即按照一定的策略周期性的将数据保存到磁盘上。配置文件中的save参数来定义快照的周期。

优点：

1. 压缩的二进制文件，非常适合备份和灾难恢复

缺点：

1. 备份操作需要fork一个进程（使用bgsave命令）属于重操作
2. 不能最大限度避免丢失数据

AOF

AOF(Append-Only File)，Redis会将每一个收到的写命令都通过Write函数追加到文件中（默认appendonly.aof）。

优点：

1. 以文本形式保存，易读
2. 能最大限度避免丢失数据

缺点：

1. 文件体积过大（可以使用bgrewriteaof重写aof)
2. 相比rdb，aof恢复数据较慢

AOF支持三种同步策略：

1. always

每条Redis写命令都同步写入硬盘

2. everysec

每秒执行一次同步，将多个命令写入硬盘

3. no

由操作系统决定何时同步。

Redis重启的时候优先加载AOF文件，如果AOF文件不存在再去加载RDB文件。 如果AOF文件和RDB文件都不存在，那么直接启动。 不论加载AOF文件还是RDB文件，只要发生错误都会打印错误信息，并且启动失败。

redis键的数据结构是什么样的?

整个Redis 数据库的所有key 和value 也组成了一个全局字典，还有带过期时间的key 集合也是一个字典。

```
struct RedisDb {
    dict    * dict;         /* all keys key=>value */
    dict    * expires;      /* all expired keys key=>long(timestamp) */
    ...
}
```

zset底层数据结构是什么样的?

Redis 的zset 是一个复合结构，一方面它需要一个hash结构（字典）来存储value（成员）和score 的对应关系， 另一方面需要提供按照score 来排序的功能，还需要能够指定score 的范围来获取value 列表的功能，这个时候通过跳跃列表实现。

```
typedef struct zset {
    dict *dict; // 字典
    zskiplist *zsl; // 跳表
} zset;

typedef struct zskiplist {
    struct zskiplistNode *header, *tail;
    unsigned long length;
    int level;
} zskiplist;

typedef struct zskiplistNode {
    sds ele;
    double score;
    struct zskiplistNode *backward;
    struct zskiplistLevel {
        struct zskiplistNode *forward;
        unsigned long span;
    } level[];
} zskiplistNode;
```

set底层数据结构是什么样的?

set底层数据结构有两种，即intset和dict。当满足以下条件时候，使用intset，否则使用dict：

1. 结合对象保存的所有元素都是整数值
2. 集合对象保存的元素数量不超过512个

当使用dict时候，即hashTable，哈希表的key值是set集合元素，value值是nil。

资料：https://juejin.cn/post/6844904198019137550

Zrank的复杂度是O(log(n))，为什么?

Redis 在skiplist 的forward 指针上进行了优化，给每一个forward 指针都增加了span 属性，span 是「跨度」的意思， 表示从前一个节点沿着当前层的forward 指针跳到当前这个节点中间会跳过多少个节点 这样计算一个元素的排名时，只需要将「搜索路径」上的经过的所有节点的跨度span 值进行叠加就可以算出元素的最终 rank 值。

zrange 的复杂度是 O(log(N)+M)， N 为有序集的基数，而 M 为结果集的基数。为什么是这个复杂度呢?

ZRANGE key start stop [WITHSCORES]，zrange 就是返回有序集 key 中，指定区间内的成员，而跳表中的元素最下面的一层是有序的(上面的几层就是跳表的索引)，按照分数排序，我们只要找出 start 代表的元素，然后向前或者向后遍历 M 次拉出所有数据即可，而找出 start 代表的元素，其实就是在跳表中找一个元素的时间复杂度。**跳表中每个节点每一层都会保存到下一个节点的跨度，在寻找过程中可以根据跨度和来求当前的排名**，所以查找过程是 O(log(N) 过程，加上遍历 M 个元素，就是 O(log(N)+M)，所以 redis 的 zrange 不会像 mysql 的 offset 有比较严重的性能问题。

redis中哪些操作时间复杂度O(n)?

Key

```
keys *
```

List

```
lindex // n为列表长度
lset
linsert
```

Hash

```
hgetall // n为哈希表大小
hkeys
hvals
```

Set

```
smembers // 返回所有集合成员，n为集合成员元素
sunion/sunionstore // 并集， N 是所有给定集合的成员数量之和
sinter/sinterstore // 交集，O(N * M)， N 为给定集合当中基数最小的集合， M 为给定集合的个数
sdiff/sdiffstore // 差集， N 是所有给定集合的成员数量之和
Sorted Set:

zrange/zrevrange/zrangebyscore/zrevrangebyscore/zremrangebyrank/zremrangebyscore // O(m) + O(log(n)) // N 为有序集的基数，而 M 为结果集的基数
```

redis哨兵模式介绍?

redis的哨兵模式（sentinel）**为了保证redis主从的高可用**。主要功能：

1. 监控：它会监听主服务器和从服务器之间是否在正常工作。
2. 故障转移：它在主节点出了问题的情况下，会在所有的从节点中竞选出一个节点，并将其作为新的主节点。

从从节点选出一个主节点流程是：

1. 首先去掉所有断线或下线的节点，获取所有监控节点
2. 然后选择**复制偏移量**最大的节点，复制偏移量代表其从主节点成功复制了多少数据，越大说明越与主节点最接近同步。
3. 若步骤选出了多个节点，那比较每个节点的唯一标识uid，选择最小的那个。

将从节点变成主节点操作时候，需要哨兵来操作，由于为了保证哨兵高可用性，哨兵存在多个，那就需要选主出来一个哨兵头领来处理这个操作，这个过程涉及到Raft算法。

redis cluster分片原理?

采用虚拟槽进行数据分片，总共2^14个虚拟槽，有几个节点就把2^14分成几个范围，按照crc16(key)%2^14确定key在哪个槽，每个节点保存了集群中的槽对应的节点信息，如果一个请求过来发现key不在这个节点上，这个节点会回复一个mov的消息指向新节点，彼此节点间定时通过ping来检测故障。

redis rehash过程?

redis采用渐进式hash方式。redis会同时维持两个hash表：ht[0] 和 ht[1] 两个哈希表。 要在字典里面查找一个键的话， 程序会先在 ht[0] 里面进行查找， 如果没找到的话， 就会继续到 ht[1] 里面进行查找， 诸如此类。在渐进式 rehash 执行期间， 新添加到字典的键值对一律会被保存到 ht[1] 里面， 而 ht[0] 则不再进行任何添加操作

mysql

mysql存储引擎Myisam和innodb区别?

对比项	MyISAM	InnoDB
主外键	不支持	支持
事务	不支持	支持
行表锁	表锁，即使操作一条记录也会锁住整个表，不适合高并发的操作	行锁,操作时只锁某一行，不对其它行有影响， 适合高并发的操作
缓存	只缓存索引，不缓存真实数据	不仅缓存索引还要缓存真实数据，对内存要求较高，而且内存大小对性能有决定性的影响
表空间	小	大
关注点	性能	事务
默认安装	Y	Y

1. InnoDB支持事务，MyISAM不支持
2. InnoDB支持外键，而MyISAM不支持
3. InoDB支持行级别锁，有更高的并发性，而MyISAM只支持表级别锁
4. InnoDB是聚集索引，数据文件是和索引绑在一起的，必须要有主键，通过主键索引效率很高

mysql是ACID是如何保证的?

mysql事务特征有：

- 原子性(Atomicity)： 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
- 一致性(Consistency)： 执行事务前后，数据保持一致；
- 隔离性(Isolation)： 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；
- 持久性(Durability): 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库 发生故障也不应该对其有任何影响。

保证措施：

- A 原子性由undo log日志保证，它记录了需要回滚的日志信息，事务回滚时撤销已经执行成功的sql
- C 一致性一般由代码层面来保证
- I 隔离性由MVCC来保证
- D 持久性由内存+redo log来保证，mysql修改数据同时在内存和redo log记录这次操作，事务提交的时候通过redo log刷盘，宕机的时候可以从redo log恢复。

mysql事务隔离级别有哪些?

mysql支持四种隔离级别，默认是可重复读的 (REPEATABLE READ)。

隔离级别	脏读	不可重复读取	幻影数据行
READ UNCOMMITTED(RU) - 读未提交	X	X	X

隔离级别	脏读	不可重复读取	幻影数据行
READ COMMITTED(RC) - 读已提交	√	X	X
REPEATABLE READ(RR) - 可重复读	√	√	X
SERIALIZABLE(SZ) - 串行化	√	√	√

不可重复读与幻读的区别是?

1. 不可重复读的重点是修改:同样的条件, 你读取过的数据, 再次读取出来发现值不一样了，其只需要锁住满足条件的记录
2. 幻读的重点在于新增或者删除：同样的条件, 第1次和第2次读出来的记录数不一样，要锁住满足条件及其相近的记录

mysql在可重复读模式下，如何解决不可重复读的问题?

mvcc+undo log解决了快照读可能会导致的不可重复读的问题。Mysql默认隔离级别是RR（可重复读），是通过“行锁+MVCC”来实现的，正常读时不加锁，写时加锁，MVCC的实现依赖于：三个隐藏字段，Read View、Undo log 来实现。

MVCC的目的就是多版本并发控制，在数据库中的实现，就是为了解决读写冲突，它的实现原理主要是依赖记录中的 3个隐式字段，undo log ，Read View 来实现的。 InnoDB MVCC的实现基于undo log，通过回滚指针来构建需要的版本记录。通过ReadView来判断哪些版本的数据可见。同时Purge线程是通过ReadView来清理旧版本数据。

mysql的当前读和快照读是什么回事?

快照读（snapshot read）能看到别的事务生成快照前提交的数据，而不能看到别的事务生成快照后提交的数据或者未提交的数据。快照读是repeatable-read 和 read-committed 级别下，默认的查询模式，好处是：读不加锁，读写不冲突，这个对于 MySQL 并发访问提升很大。

快照读的实现方式：undolog和多版本并发控制MVCC。

使用快照读的场景：单纯的select操作，不包括上述 select ... lock in share mode、select ... for update

Read Committed隔离级别下快照读：每次select都生成一个快照读

Read Repeatable隔离级别下快照读：开启事务后第一个select语句才是快照读的地方，而不是一开启事务就快照读

在 read-committed 隔离级别下，事务中的快照读，总是以最新的快照为基准进行查询的。

在 repeatable-read 隔离级别下，快照读是以事务开始时的快照为基准进行查询的，如果想以最新的快照为基准进行查询，可以先把事务提交完再进行查询。

在 repeatable-read 隔离级别下，别的事务在你生成快照后进行的删除、更新、新增，快照读是看不到的。

当前读读取的是最新版本, 并且对读取的记录加锁，保证其他事务不会再并发的修改这条记录，避免出现安全问题。

使用当前读的场景：

- select...lock in share mode (共享读锁)
- select...for update
- update
- delete
- insert

当前读的实现方式：next-key锁(行记录锁+Gap间隙锁)

innodb三种行锁算法

InnoDB有三种行锁的算法：

1. Record Lock：单个行记录上的锁。
2. Gap Lock：间隙锁，锁定一个范围，但不包括记录本身。GAP锁的目的，是为了防止同一事务的两次当前读，出现幻读的情况。
3. Next-Key Lock：1+2，锁定一个范围，并且锁定记录本身。对于行的查询，都是采用该方法，主要目的是解决幻读的问题。

truncate,delete,drop区别?

- truncate 会删除所有数据
- delete 可以删除部分数据，会触发触发器
- drop会删除整个表和数据

mysql索引分类

从物理存储角度

1. 聚簇索引

InnoDB中主键索引是聚集索引，索引跟数据在一起的。其他索引是非聚集索引，索引指向的是主键索引。为MyISAM存储引擎数据文件和索引文件是分离的，不存在聚集索引的概念。

2. 非聚簇索引

从数据结构角度

1. B+树索引
2. hash索引

   基于哈希表实现，只有全值匹配才有效

3. 全文索引

   查找的是文本中的关键词，而不是直接比较索引中的值，类似于搜索引擎做的事情。

从逻辑角度

1. 唯一索引，唯一索引也是一种约束。唯一索引的属性列不能出现重复的数据，但是**允许数据为NULL**，一张表允许创建多个唯一索引。UNIQUE (column)
2. 主键索引，数据表的主键列使用的就是主键索引PRIMARY KEY (column)
3. 联合索引，指多个字段上创建的索引 INDEX index_name (column1, ...)，使用时最左匹配原则
4. 普通 /单列索引，普通索引的唯一作用就是为了快速查询数据，一张表允许创建多个普通索引，并允许数据重复和NULL。INDEX index_name (column)。
5. 全文索引，查找的是文本中的关键词，而不是直接比较索引中的值，类似于搜索引擎做的事情。FULLTEXT (column)

资料：https://juejin.cn/post/6907966385394515975

mysql为什么使用b+树?

B+树是一个平衡的多叉树，B+树中的B不是代表二叉（binary），而是代表平衡（balance）。

B+树和B树区别：

1. B +树中的非叶子节点不存储数据，并且存储在叶节点中的所有数据使得查询时间复杂度固定为log n。
2. B树查询时间的复杂度不是固定的，它与键在树中的位置有关，最好是O（1）。
3. 由于B+树的叶子节点是通过双向链表链接的，所以支持范围查询，且效率比B树高
4. B树每个节点的键和数据是一起的

哈希索引的优势与劣势?

优点：

1. 等值查询，哈希索引具有绝对优势，时间复杂度为O(1)

缺点：

1. 不支持范围查询
2. 不支持索引完成排序
3. 不支持联合索引的最左前缀匹配规则

为什么uuid不适合做innodb主键?

mysql调优几种方式

1. 打开慢查询日志，查看慢查询
2. explain看一下执行计划，查看①key字段是否使用到索引，使用到什么索引。②type字段是否为ALL全表扫描。③row字段扫描的行数是否过大，估计值。MySQL数据单位都是页，使用采样统计方法。④extra字段是否需要额外排序，就是不能通过索引顺序达到排序效果；是否需要使用临时表等。⑤如果是组合索引的话通过key_len字段判断是否被完全使用。
3. 使用覆盖索引。

4. 注意最左前缀原则
5. 使用前缀索引。当要给字符串加索引时，可以使用前缀索引，节省资源占用。如果前缀区分度不高可以倒序存储或者是存储hash。
6. 索引下推
7. 注意隐式类型转换，防止索引实现
8. 区分度不大的字段避免使用索引，比如性别字段

mysql语句性能评测?

使用explain分析语句执行，主要看select_type、type、key、possiable keys、extra列。

查询类型 - select_type

- SIMPLE : 简单的select查询,查询中不包含子查询或者UNION
- PRIMARY: 查询中若包含复杂的子查询,最外层的查询则标记为PRIMARY
- SUBQUERY : 在SELECT或者WHERE列表中包含子查询
- DERIVED : 在from列表中包含子查询被标记为DRIVED衍生,MYSQL会递归执行这些子查询,把结果放到临时表中
- UNION: 若第二个SELECT出现在union之后,则被标记为UNION, 若union包含在from子句的子查询中,外层select被标记为:derived
- UNION RESULT: 从union表获取结果的select

连接类型 - type

有如下几种取值，性能从好到坏排序 如下：

- const

  针对主键或唯一索引的等值查询扫描, 最多只返回一行数据. const 查询速度非常快, 因为它仅仅读取一次即可

- ref

  当满足索引的**最左前缀规则**，或者索引不是主键也不是唯一索引时才会发生。如果使用的索引只会匹配到少量的行，性能也是不错的

- range

  范围扫描，表示检索了指定范围的行，主要用于**有限制的索引扫描**。比较常见的范围扫描是带有BETWEEN子句或WHERE子句里有>、>=、<、<=、IS NULL、<=>、BETWEEN、LIKE、IN()等操作符。

- index

  全索引扫描，和ALL类似，只不过index是全盘扫描了索引的数据。当查询仅使用索引中的一部分列时，可使用此类型。有两种场景会触发： 如果索引是查询的覆盖索引，并且索引查询的数据就可以满

足查询中所需的所有数据，则只扫描索引树。此时，explain的Extra 列的结果是Using index。index通常比ALL快，因为索引的大小通常小于表数据。

- ALL

  全表扫描，性能最差

key

表示MySQL实际选择的索引

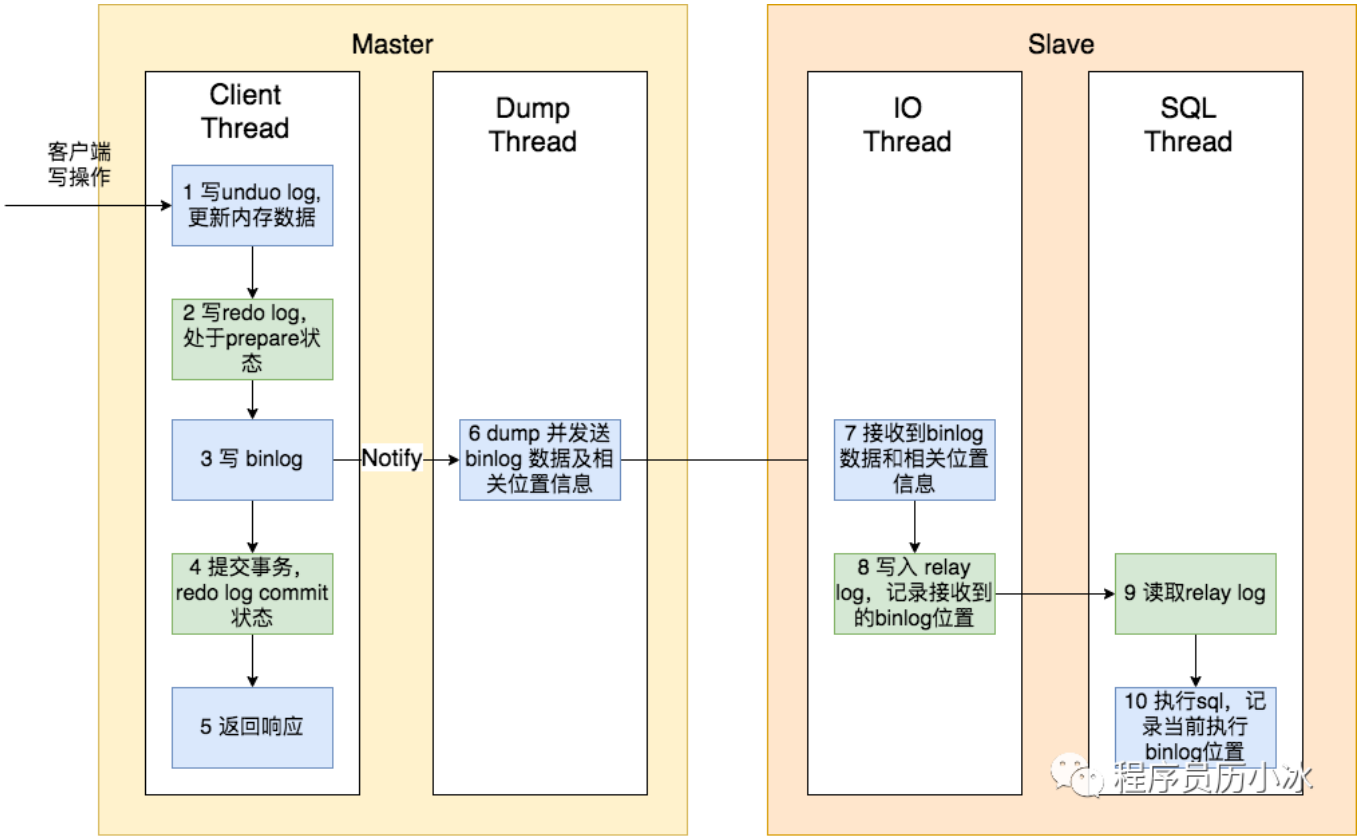
possiable keys

MYSQL可能用到的key

rows

MySQL估算会扫描的行数，数值越小越好。

mysql主从复制流程?



mysql slave节点连接master节点时，master节点会新建一个binlog dump线程，当master数据有更新时写入到binlog，dump线程通知slave的io线程接收后写入到本地的relaylog，然后通过sql线程重放

mysql日志类型

mysql日志主要包括错误日志、查询日志、慢查询日志、事务日志、二进制日志几大类。

binlog

binlog 用于记录数据库执行的写入性操作(不包括查询)信息，以二进制的形式保存在磁盘中。binlog 的主要使用场景有两个，分别是 主从复制 和 数据恢复。

- 主从复制 ： 在 Master 端开启 binlog ，然后将 binlog发送到各个 Slave 端， Slave 端重放 binlog 从而达到主从数据一致。
- 数据恢复 ： 通过使用 mysqlbinlog 工具来恢复数据。

redo log

当有一条记录要更新时，InnoDB先记录日志再更新内存，然后在比较空闲的时候将操作更新到磁盘，有了redolog即使MySQL崩溃也不会丢失数据，这个能力称为crash-safe。redo log是事务持久性的保证。

Undo log

undo log用于回滚操作，保证事务的原子性。

slow log

slow log用来记录慢查询

查询语句不同元素（where、jion、limit、group by、having等等）执行先后顺序?

where字句在聚合前先筛选记录，也就是说作用在group by和having字句前。而 having子句在聚合后对组记录进行筛选。

Innodb为什么一定需要一个主键，且必须自增列作为主键?

如果我们定义了主键(PRIMARY KEY)，那么InnoDB会选择主键作为聚集索引。如果没有显式定义主键，则InnoDB会选择第一个不包含有NULL值的唯一索引作为主键索引。如果也没有这样的唯一索引，则InnoDB会选择内置6字节长的ROWID作为隐含的聚集索引(ROWID随着行记录的写入而主键递增，这个ROWID不像ORACLE的ROWID那样可引用，是隐含的)。

总之Innodb一定需要一个主键。

1. 这是因为数据记录本身被存于主索引（一颗B+Tree）的叶子节点上，这就要求同一个叶子节点内（大小为一个内存页或磁盘页）的各条数据记录按主键顺序存放
2. 如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页（这样的页称为叶子页)
3. 如果使用非自增主键（如果身份证号或学号等），由于每次插入主键的值近似于随机，因此每次新记录都要被插到现有索引页得中间某个位置

在MVCC并发控制中，读操作可以分成哪两类?

快照读 (snapshot read)：读取的是记录的可见版本 (有可能是历史版本)，不用加锁（共享读锁s锁也不加，所以不会阻塞其他事务的写）。

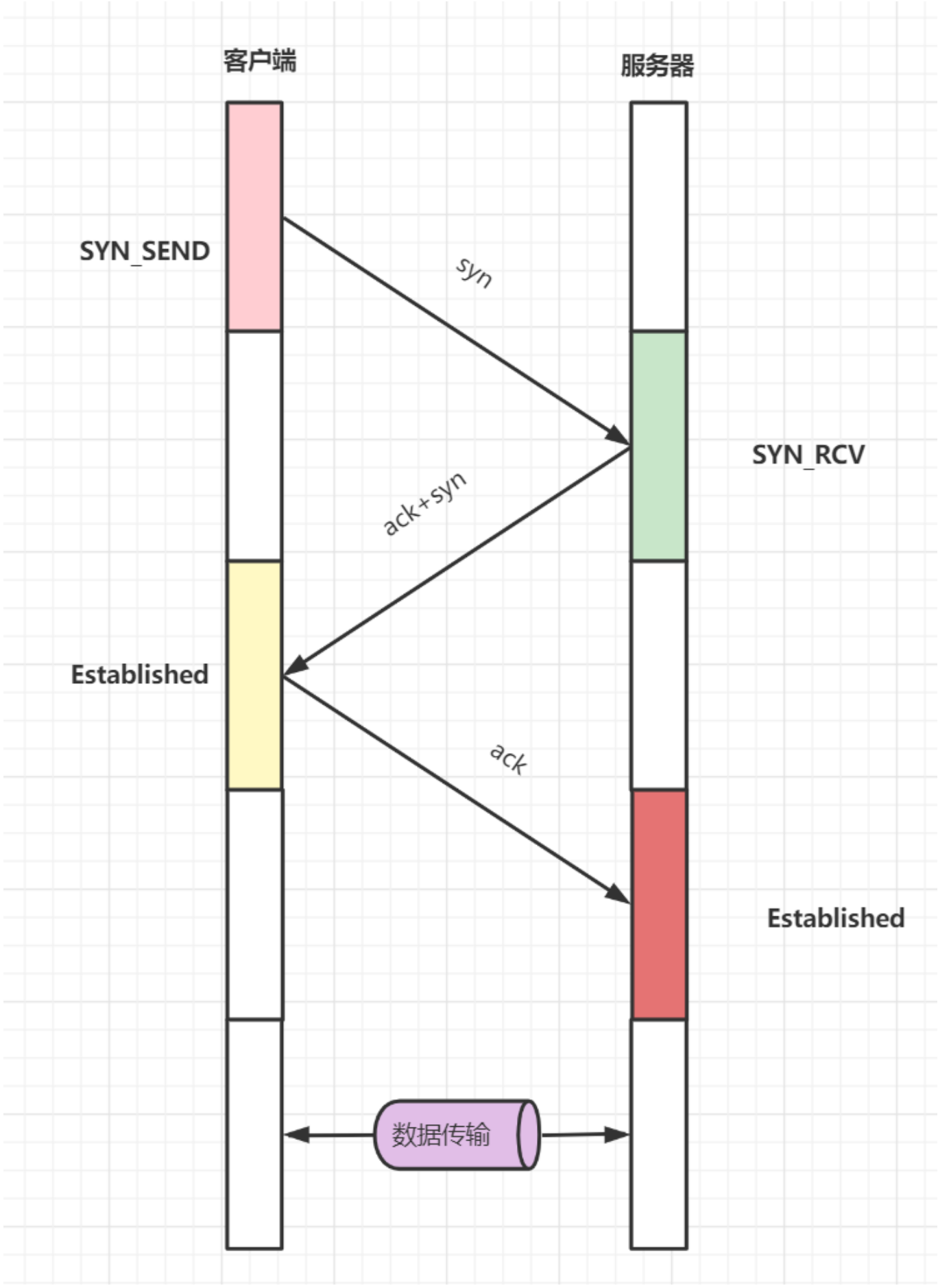
当前读 (current read)：读取的是记录的最新版本，并且，当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。

TCP

什么是 TCP？

TCP 是面向连接的、可靠的、基于字节流的传输层通信协议。

TCP三次握手流程?



开始客户端和服务器都处于CLOSED状态，然后服务端开始监听某个端口，进入LISTEN状态

1. 第一次握手(SYN=1, seq=x)，发送完毕后，客户端进入 SYN_SEND 状态

2. 第二次握手(SYN=1, ACK=1, seq=y, ACKnum=x+1)， 发送完毕后，服务器端进入 SYN_RCVD 状态。
3. 第三次握手(ACK=1，ACKnum=y+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，即可以开始数据传输。

当TCP连接一个不存在的端口时候，会不会有三次握手过程?

不会有。因为当服务器收到没有监听端口的连接请求时会返回RST包。

TCP四次挥手过程?

1. 第一次挥手(FIN=1，seq=u)，发送完毕后，客户端进入FIN_WAIT_1 状态
2. 第二次挥手(ACK=1，ack=u+1,seq=v)，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
3. 第三次挥手(FIN=1，ACK1,seq=w,ack=u+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。
4. 第四次挥手(ACK=1，seq=u+1,ack=w+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。服务器端接收到这个确认包之后，关闭连接，进入 CLOSED 状态。

TCP挥手为什么需要四次?

数据传输是双向传输的，一方告诉对方数据传输完成，需要两次挥手：一次发送通知给对方说我已经传输完成，一次需要接收到对方确认收到通知。也可以这么说每个方向都需要一个 FIN 和一个 ACK，因此通常被称为四次挥手。

TIME-WAIT 状态为什么需要等待 2MSL?

MSL 是 Maximum Segment Lifetime，报文最大生存时间，它是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。

主动发起关闭连接的一方，才会有 TIME-WAIT 状态。

- 1个 MSL 保证四次挥手中主动关闭方最后的 ACK 报文能最终到达对端
- 1个 MSL 保证对端没有收到 ACK 那么进行重传的 FIN 报文能够到达

TIME_WAIT 过多有什么危害?

如果服务器有处于 TIME-WAIT 状态的 TCP，则说明是由服务器方主动发起的断开请求。过多的 TIME-WAIT 状态主要的危害有两种：

- 第一是内存资源占用；
- 第二是对端口资源的占用，一个 TCP 连接至少消耗一个本地端口，导致无法创建链接

端口资源有限，一般可以开启的端口为 32768～61000，也可以通过如下参数设置指定

```
net.ipv4.ip_local_port_range
```

如何优化 TIME_WAIT?

1. 复用处于 TIME_WAIT 的 socket 为新的连接所用

```
net.ipv4.tcp_tw_reuse = 1
```

如何唯一确定一个 TCP 连接呢?

1. 源地址
2. 源端口
3. 目的地址
4. 目的端口

源地址和目的地址的字段（32位）是在 IP 头部中，作用是通过 IP 协议发送报文给对方主机。源端口和目的端口的字段（16位）是在 TCP 头部中，作用是告诉 TCP 协议应该把报文发给哪个进程。

有一个 IP 的服务器监听了一个端口，它的 TCP 的最大连接数是多少?

服务端最大并发 TCP 连接数远不能达到理论上限:

- 首先主要是文件描述符限制，Socket 都是文件，所以首先要通过 ulimit 配置文件描述符的数目；
- 另一个是内存限制，每个 TCP 连接都要占用一定内存，操作系统是有限的。

TCP 和 UDP 的区别有哪些?

- **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。
- TCP要求安全性，**提供可靠的服务**，通过TCP连接传送的数据，不丢失、不重复、安全可靠。而UDP尽最大努力交付，即不保证可靠交付。
- TCP是点对点连接的，UDP一对一，一对多，多对多都可以
- TCP传输效率相对较低,而UDP传输效率高，它适用于对高速传输和实时性有较高的通信或广播通信。
- TCP适合用于网页，邮件等;UDP适合用于视频，语音广播等
- **TCP面向字节流，UDP面向报文**

TCP 是如何保证可靠性的?

1. 连接的可靠性：TCP的连接是基于三次握手，而断开则是四次挥手。确保连接和断开的可靠性
2. 数据传输的可靠性和可控性：支持报文校验、报文确认应答、超时重传、流量控制（滑动窗口）

超时重传机制是什么样的?

超时重传指的是**在发送数据报文时，设定一个定时器，每间隔一段时间，没有收到对方的ACK确认应答报文，就会重发该报文**。超时重传强调的是客户端。

快速重传机制是什么样的?

它基于接收端的反馈信息来引发重传。接收方通过发送三次重复的ACK确认引发客户端快速重传延迟或丢失的报文。

举例子，发送端发送了 1，2，3，4，5,6 份数据:

- 第一份 Seq=1 先送到了，于是就 Ack 回 2；
- 第二份 Seq=2 也送到了，假设也正常，于是ACK 回 3；
- 第三份 Seq=3 由于网络等其他原因，没送到；
- 第四份 Seq=4 也送到了，但是因为Seq3没收到。所以ACK回3；

- 后面的 Seq=4,5的也送到了，但是ACK还是回复3，因为Seq=3没收到。
- 发送端连着收到三个重复冗余ACK=3的确认（实际上是4个，但是前面一个是正常的ACK，后面三个才是重复冗余的），便知道哪个报文段在传输过程中丢失了，**于是在定时器过期之前（发送方的超时重传机制）**，重传该报文段。
- 最后，接收到收到了 Seq3，此时因为 Seq=4，5，6都收到了，于是ACK回7.

但快速重传还可能会有个问题：ACK只向发送端告知最大的有序报文段，到底是哪个报文丢失了呢？并不确定！那到底该重传多少个包呢？是重传 Seq3 呢？还是重传 Seq3、Seq4、Seq5、Seq6 呢？因为发送端并不清楚这三个连续的 ACK3 是谁传回来的。

解决上面问题，有两个办法：

1. 带选择确认的重传（SACK）
2. D-SACK

TCP 滑动窗口是怎么回事?

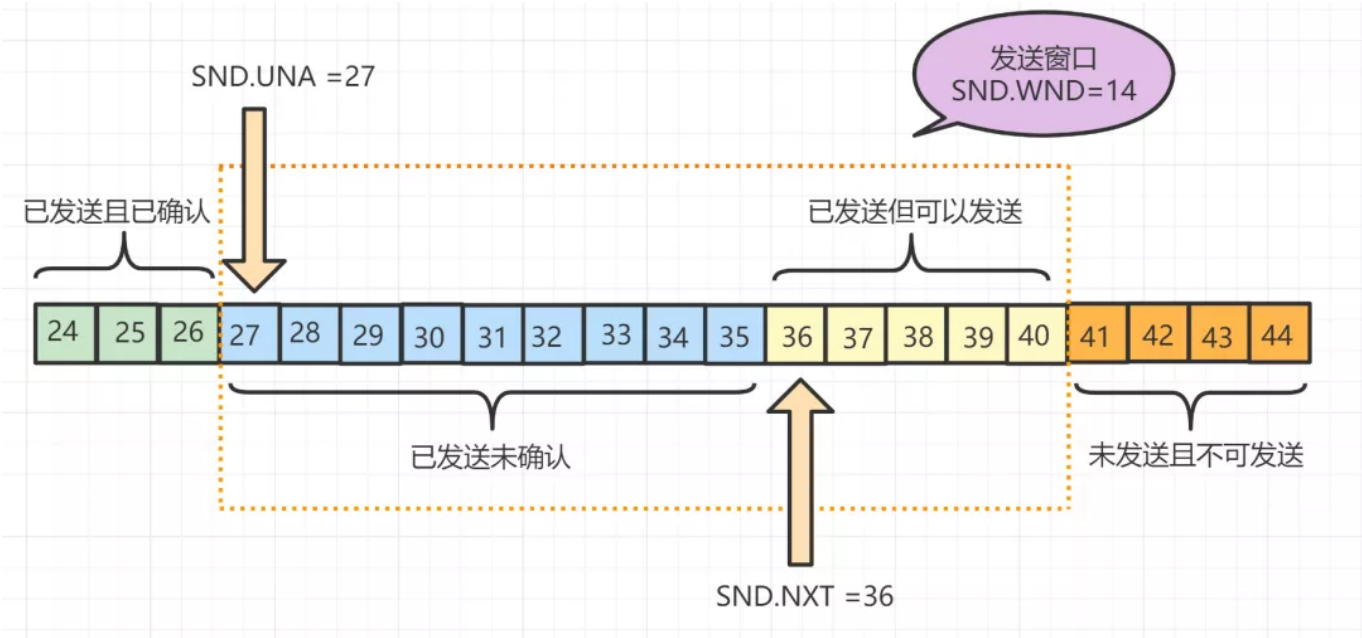
TCP 发送一个数据，需要收到确认应答，才会发送下一个数据。这样有个缺点，就是效率会比较低。

为了解决这个问题，TCP引入了窗口，它是操作系统开辟的一个缓存空间。窗口大小值表示无需等待确认应答，而可以继续发送数据的最大值。

TCP头部有个字段叫win，也即那个16位的窗口大小，它告诉对方本端的TCP接收缓冲区还能容纳多少字节的数据，这样对方就可以控制发送数据的速度，从而达到流量控制的目的。

TCP 滑动窗口分为两种: 发送窗口和接收窗口。发送端的滑动窗口包含四大部分，如下：

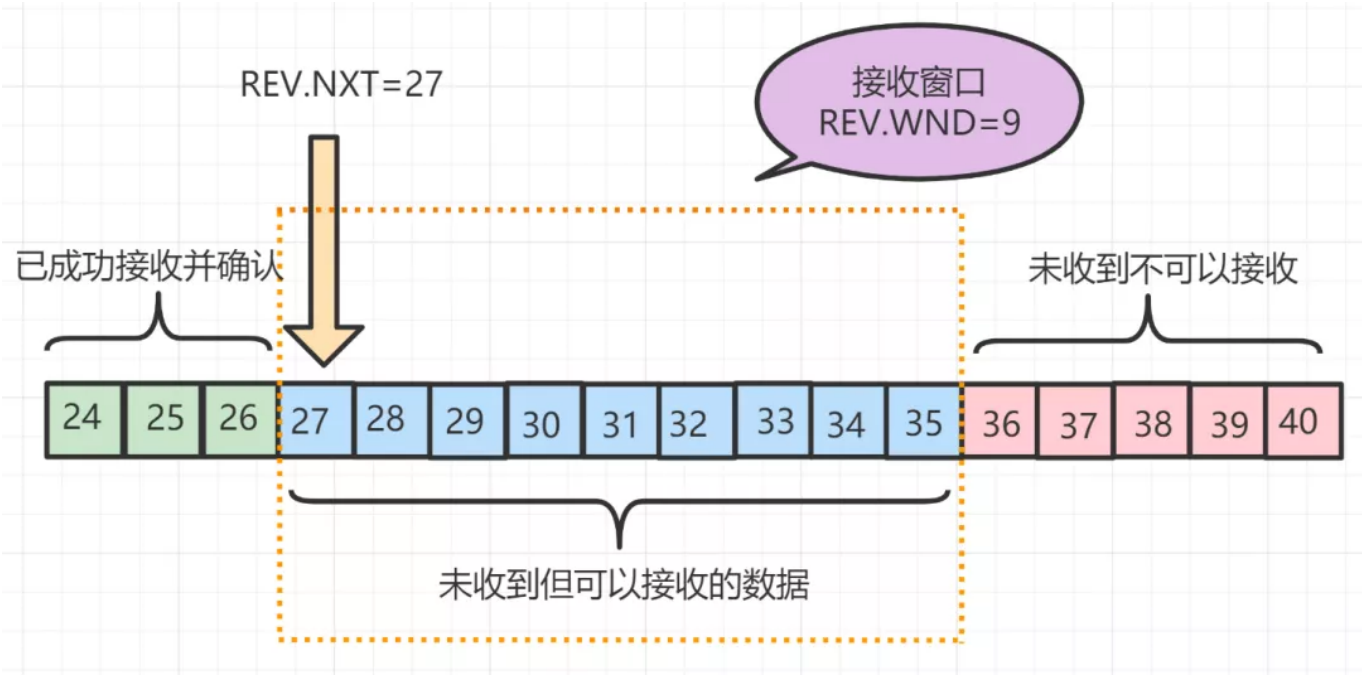
- 已发送且已收到ACK确认
- 已发送但未收到ACK确认
- 未发送但可以发送
- 未发送也不可以发送



接收方的滑动窗口包含三大部分，如下：

- 已成功接收并确认

- 未收到数据但可以接收
- 未收到数据并不可以接收的数据



TCP的流量控制是怎么回事?

TCP 提供一种机制可以让发送端根据接收端的实际接收能力控制发送的数据量，这就是流量控制。流量控制是作用于接收者的，根据接收端的实际接收能力控制发送速度，防止分组丢失的。

注意TCP的拥塞控制作用于网络的，防止过多的数据包注入到网络中，避免出现网络负载过大的情况。

TCP的粘包和拆包是如何实现的?

TCP是面向流，没有界限的一串数据。TCP底层并不了解上层业务数据的具体含义，它会根据TCP缓冲区的实际情况进行包的划分，所以在业务上认为，一个完整的包可能会被TCP拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这就是所谓的TCP粘包和拆包问题。

为什么会产生粘包和拆包呢?

- 要发送的数据小于TCP发送缓冲区的大小，TCP将多次写入缓冲区的数据一次发送出去，将会发生粘包；
- 接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包；
- 要发送的数据大于TCP发送缓冲区剩余空间大小，将会发生拆包；
- 待发送数据大于MSS（最大报文长度），TCP在传输前将进行拆包。即TCP报文长度-TCP头部长度>MSS。

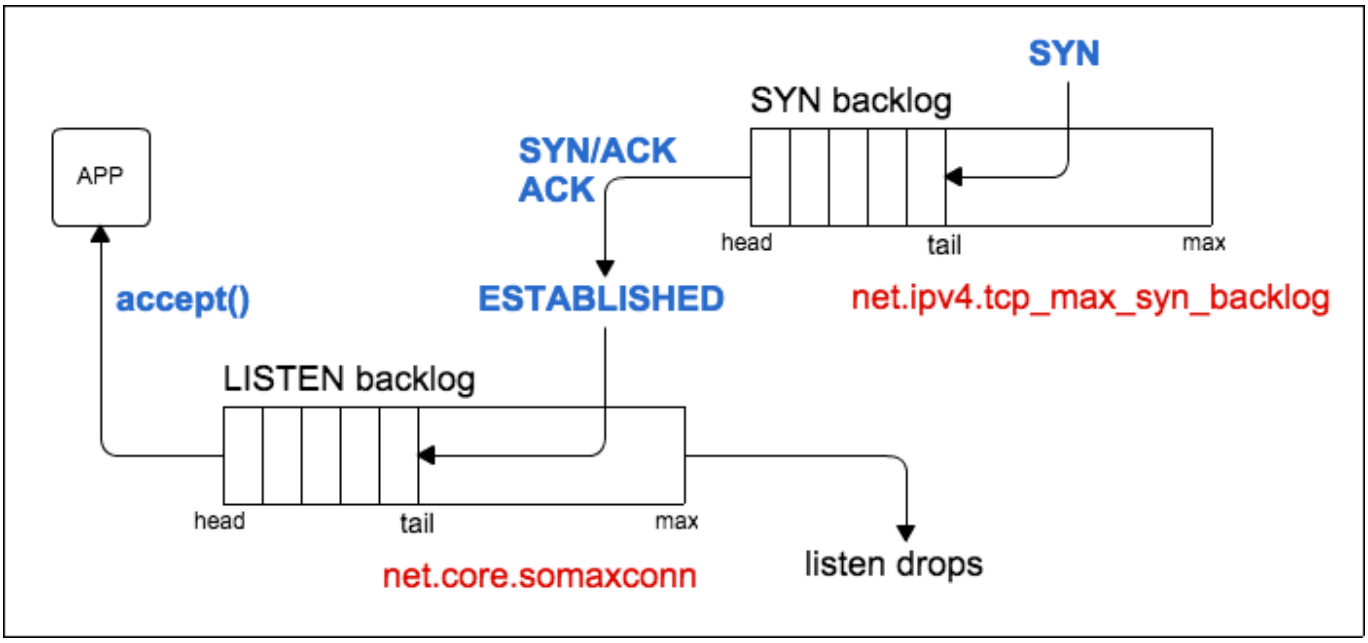
解决方案

- 在数据尾部增加特殊字符进行分割
- 将数据分为两部分，一部分是头部，一部分是内容体；其中头部结构大小固定，且有一个字段声明内容体的大小

半连接队列和 SYN Flood 攻击的关系?

一个完整的连接建立过程，服务器会经历 2 种 TCP 状态：SYN_REVD, ESTABELLISHED。对应也会维护两个队列：

- 一个存放SYN的队列（半连接队列，也成SYN队列）
- 一个存放已经完成连接的队列（全连接队列， 也称Accept队列）



SYN Flood是一种典型的DoS (Denial of Service，拒绝服务) 攻击，它在短时间内，伪造不存在的IP地址,向服务器大量发起SYN报文。当服务器回复SYN+ACK报文后，不会收到ACK回应报文，导致服务器上建立大量的半连接半连接队列满了，这就无法处理正常的TCP请求。

解决办法是：

- syn cookie：在收到SYN包后，服务器根据一定的方法，以数据包的源地址、端口等信息为参数计算出一个cookie值作为自己的SYN ACK包的序列号，回复SYN+ACK后，服务器并不立即分配资源进行处理，等收到发送方的ACK包后，重新根据数据包的源地址、端口计算该包中的确认序列号是否正确，如果正确则建立连接，否则丢弃该包。

以太网下，TCP包最大负载是多少?

最大负载大小(MSS) = MTU(1500) - IP头大小(20) - TCP头大小(20) = 1460

IP和TCP头大小不固定，最小是20字节

HTTP

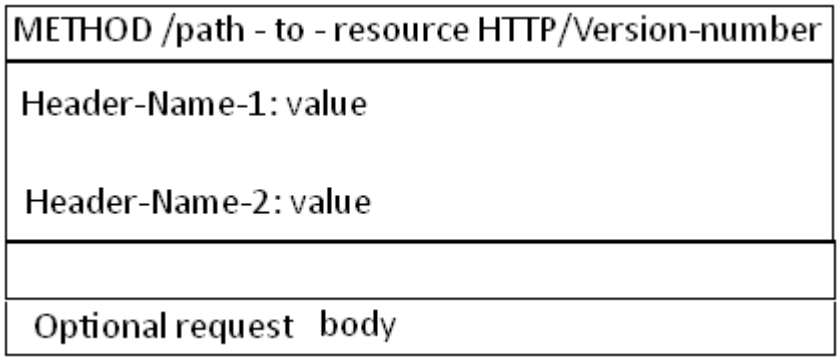
什么是HTTP协议?

超文本传输协议(HTTP)是一种通信协议，它允许将超文本标记语言(HTML)文档从Web服务器传送到客户端的浏览器。目前广泛使用的是HTTP/1.1 版本。

HTTP请求消息和响应消息格式是?

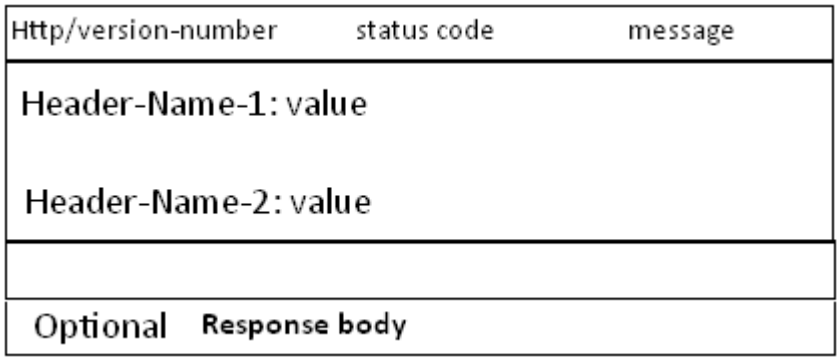
请求消息

Request 消息分为3部分，第一部分叫Request line, 第二部分叫Request header, 第三部分是body. header和body之间有个空行， 结构如下图

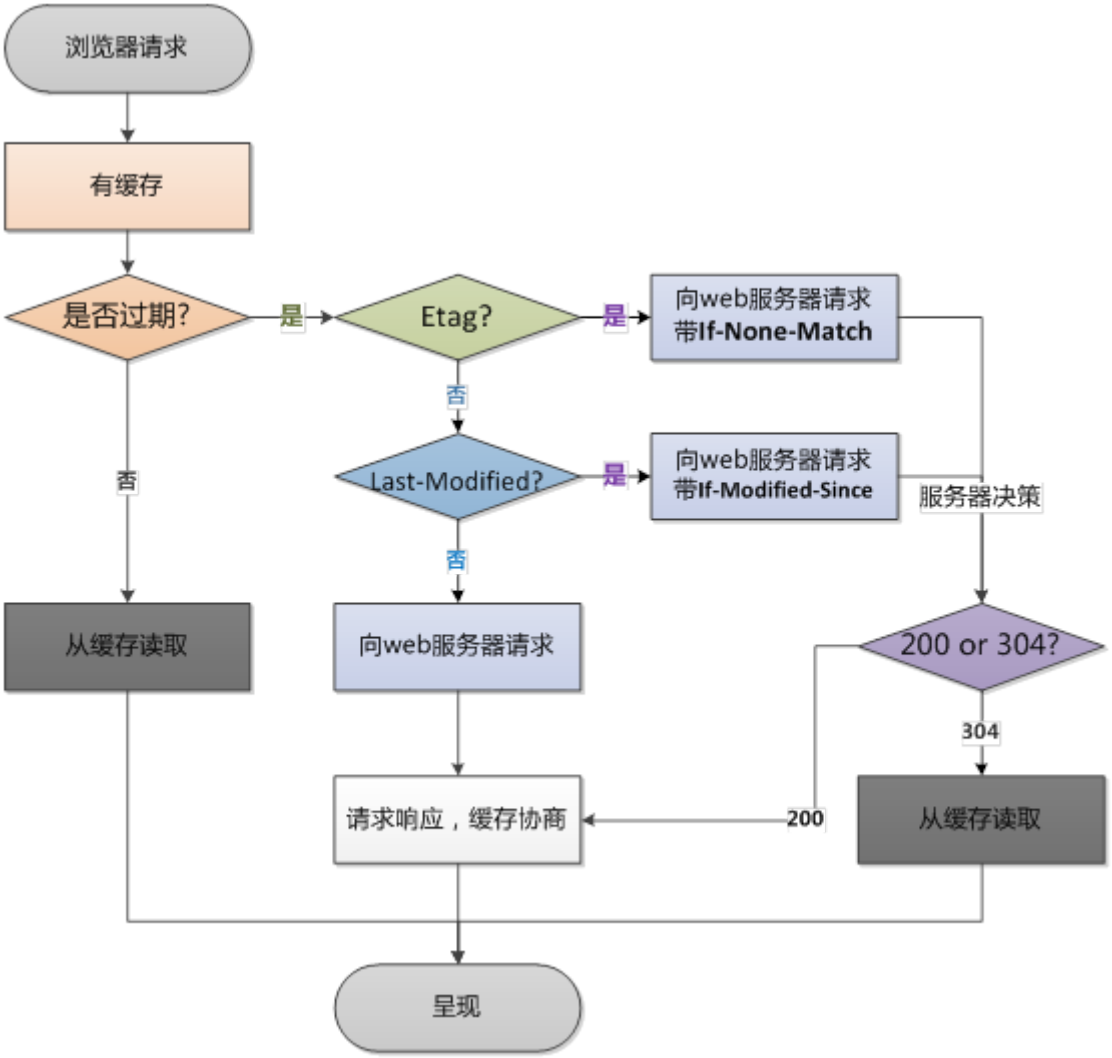


响应消息

第一部分叫Response line, 第二部分叫Response header，第三部分是body. header字段之间要有空行（\r\n)，header和body之间也有个空行, 结构如下图:



HTTP Cache流程是怎么样的?



HTTP Cache有哪些重要的头?

Cache-Control

选项值有:

- Public： 所有内容都将被缓存，在响应头中设置
- Private： 内容只缓存到私有服务器中，在响应头中设置
- no-cache： **不是不缓存，而是缓存需要校验**。
- no-store： 所有内容都不会被缓存到缓存或Internet临时文件中，在响应头中设置 must-revalidation/proxy-revalidation： 如果缓存的内容失效，请求必须发送到服务器/代理以进行重新验证，在请求头中设置
- max-age=xxx： 缓存的内容将在xxx秒后失效，这个选项只在HTTP1.1中可用，和Last-Modified一起使用时优先级较高，在响应头中设置

Expires

它通常的使用格式是Expires:Fri ,24 Dec 2027 04:24:07 GMT，后面跟的是日期和时间，超过这个时间后，缓存的内容将失效，浏览器在发出请求之前会先检查这个页面的这个字段，查看页面是否已经过期，过期了就重新向服务器发起请求

Last-Modified / If-Modified

它一般用于表示一个服务器上的资源最后的修改时间，资源可以是静态或动态的内容， 通过这个最后修改时间可以判断当前请求的资源是否是最新的。 一般服务端在响应头中返回一个Last-Modified字段，告诉浏览器这个页面的最后修改时间， 浏览器再次请求时会在请求头中增加一个If-Modified字段，询问当前缓存的页面是否是最新的， 如果是最新的就返回304状态码，告诉浏览器是最新的，服务器也不会传输新的数据

Etag/If-None-Match

一般用于当Cache-Control:no-cache时，用于验证缓存有效性。

它的作用是让服务端给每个页面分配一个唯一 的编号，然后通过这个编号来区分当前这个页面是否是最新的， 这种方式更加灵活，但是后端如果有多台Web服务器时不太好处理，因为每个Web服务器都要记住网站的所有资源，否则浏览器返回这个编号就没有意义了

HTTP跨域有哪些?

CORS跨域访问的请求分三种:

- simple request

  如果一个请求没有包含任何自定义请求头，而且它所使用HTTP动词是GET，HEAD或POST之一，那么它就是一个Simple Request。但是在使用POST作为请求的动词时，该请求的Content-Type需要是application/x-www-form-urlencoded，multipart/form-data或text/plain之一。

- preflighted request(预请求)

  如果一个请求包含了任何自定义请求头，或者它所使用的HTTP动词是GET，HEAD或POST之外的任何一个动词，那么它就是一个Preflighted Request。如果POST请求的Content-Type并不是application/x-www-form-urlencoded，multipart/form-data或text/plain之一，那么其也是Preflighted Request。

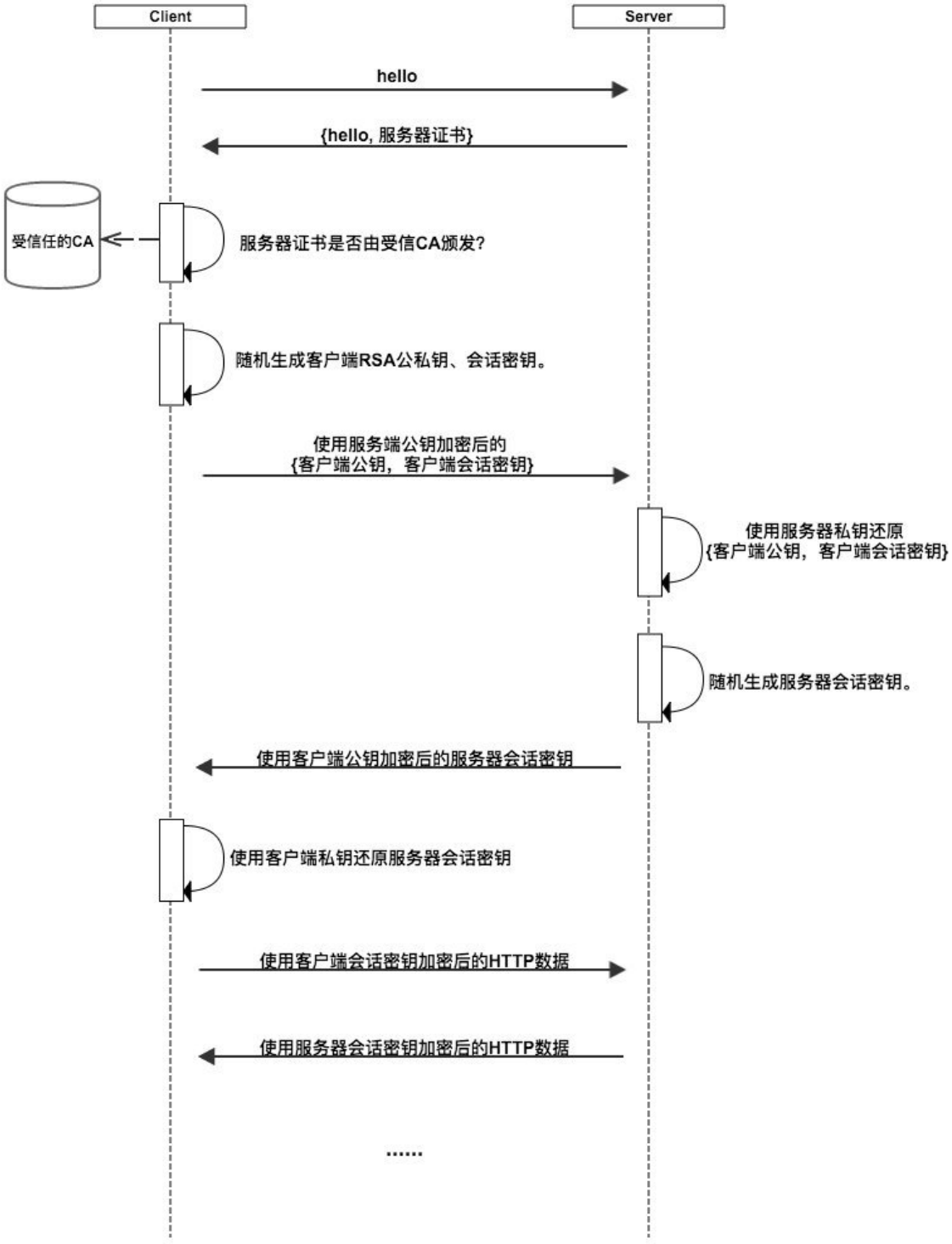
- requests with credential

  一般情况下，一个跨域请求不会包含当前页面的用户凭证。一旦一个跨域请求包含了当前页面的用户凭证，那么其就属于Requests with Credential。

对于simple request 只需要在后端程序处理时候设Access-Control-Allow-Orgin头就可以了。

对于preflighted request 每次都会请求2次，第一次options（firefox下看不到这次请求，chrome可以看见)。如果只能跟simple request 一样只设置access-control-allow-orgin是不行的。 还必须处理$_SERVER[‘REQUEST_METHOD’] == ‘OPTIONS’，2者都必须处理

HTTPS的四次握手过程是什么样的?



https握手过程分为两步：

通过CA验证服务端的证书是否真实,交换客户端和服务端的对称加密秘钥，以后数据传输，靠这两个进行加密。引入CA目的是为了防止中间人攻击。即攻击者伪造成服务端，然后发送假的证书。

现代浏览器在与服务器建立了一个 TCP 连接后是否会在一个 HTTP 请求完成后断开?什么情况下会断开?

默认情况下建立 TCP 连接不会断开，只有在请求报头中声明 Connection: close 才会在请求完成后关闭连接。在 HTTP/1.0 中，一个服务器在发送完一个 HTTP 响应后，会断开 TCP 链接。但是这样每次请求都会重新建立和断开 TCP 连接，代价过大。所以虽然标准中没有设定，某些服务器对 Connection: keep-alive 的 Header 进行了支持。

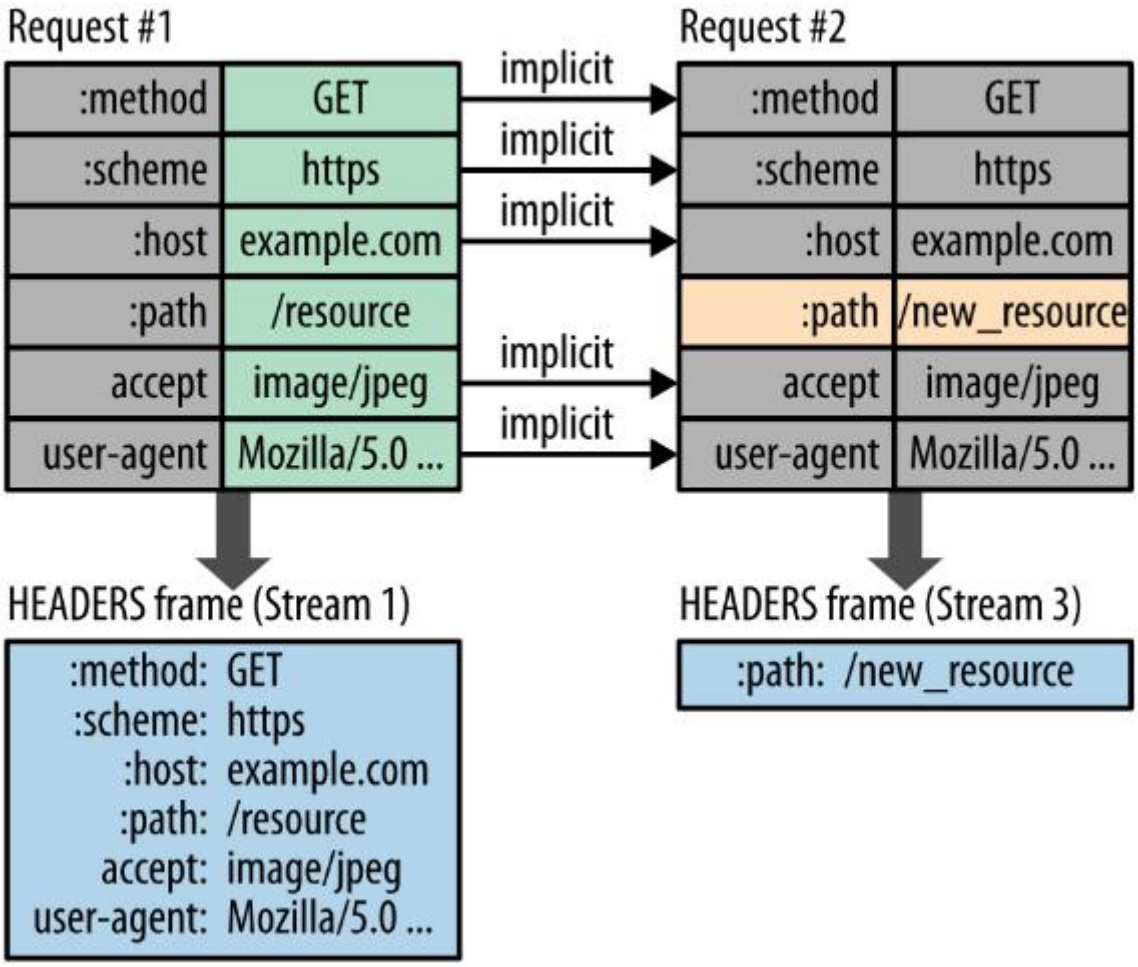
相比HTTP1，HTTP2有哪些优点？

1. 多路复用

   HTTP/2在一个TCP连接上可以并行的发送多个请求。这是HTTP/2协议最重要的特性，因为这允许你可以异步的从服务器上下载网络资源。许多主流的浏览器都会限制一个服务器的TCP连接数量

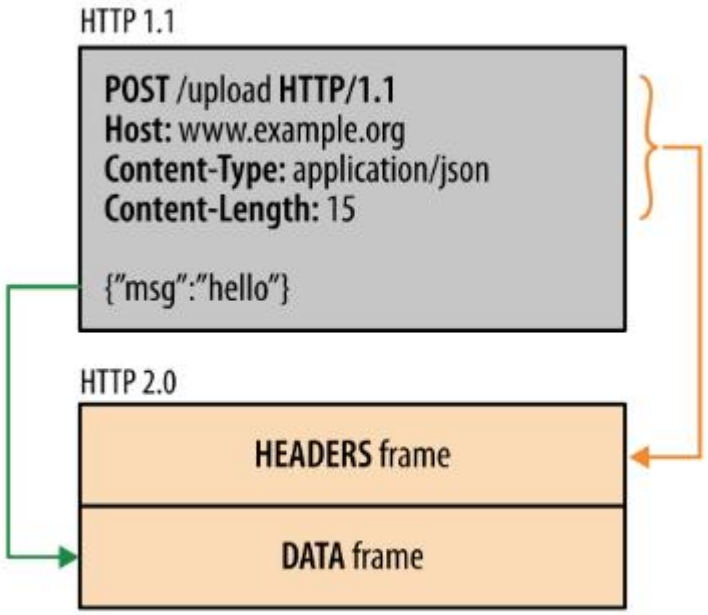
2. 请求头压缩

HTTP2.0可以维护一个字典，差量更新HTTP头部，大大降低因头部传输产生的流量。HTTP/1.1 的首部带有大量信息，而且每次都要重复发送。HTTP/2.0 要求客户端和服务器同时维护和更新一个包含之前见过的首部字段表，从而避免了重复传输。不仅如此，HTTP/2.0 也使用 Huffman 编码对首部字段进行压缩。

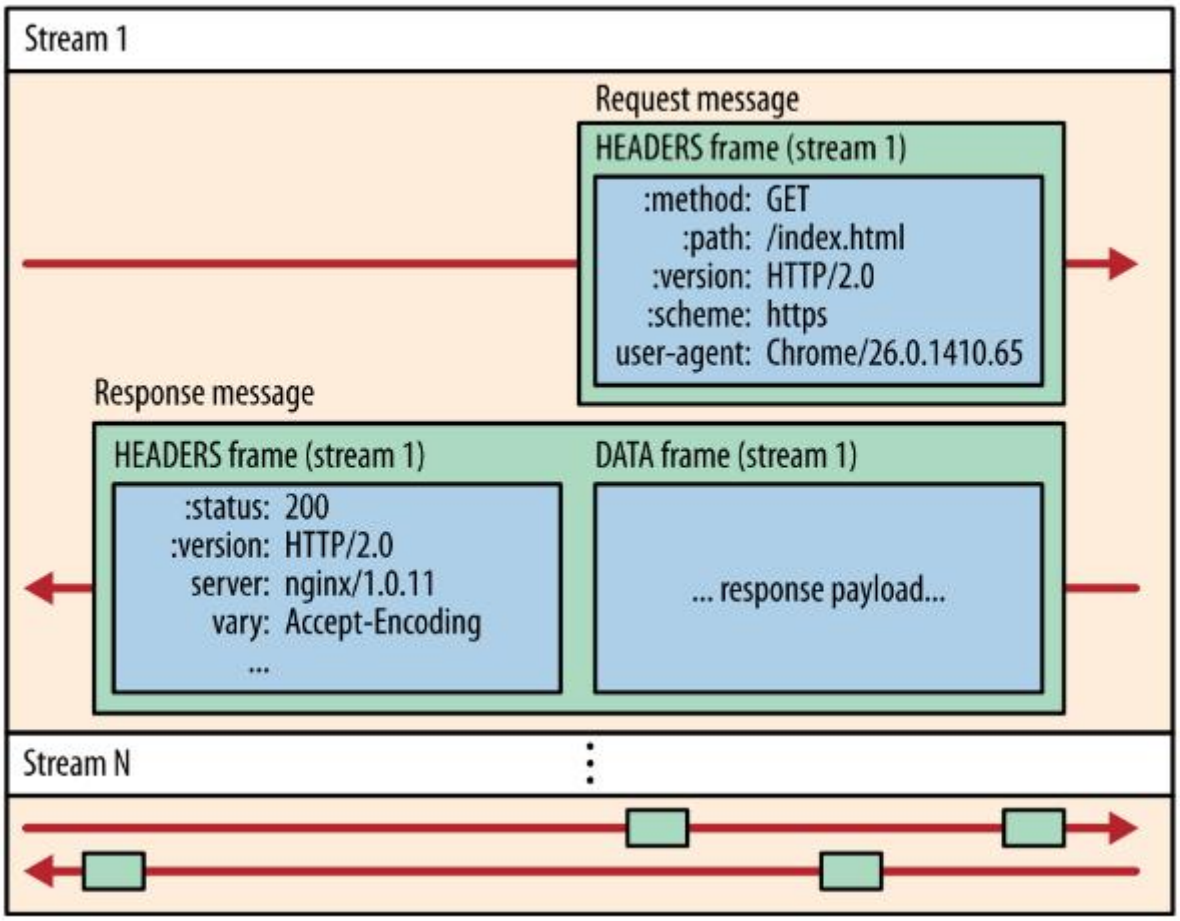


3. 二级制分帧层

TTP/2.0 将报文分成 HEADERS 帧和 DATA 帧，它们都是二进制格式的。

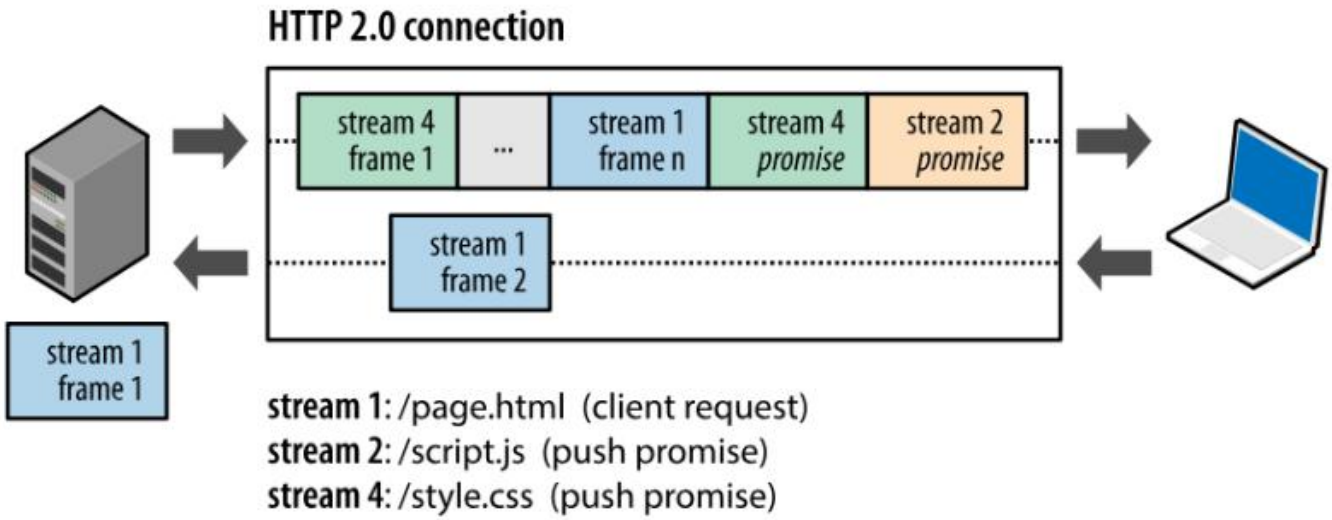


在通信过程中，只会有一个 TCP 连接存在，它承载了任意数量的双向数据流（Stream）。一个数据流都有一个唯一标识符和可选的优先级信息，用于承载双向信息。消息（Message）是与逻辑请求或响应消息对应的完整的一系列帧。帧（Fram）是最小的通信单位，来自不同数据流的帧可以交错发送，然后再根据每个帧头的数据流标识符重新组装。



4. 服务端推送

HTTP/2.0 在客户端请求一个资源时，会把相关的资源一起发送给客户端，客户端就不需要再次发起请求了。例如客户端请求 page.html 页面，服务端就把 script.js 和 style.css 等与之相关的资源一起发给客户端。



缓存

缓存有3大问题，以及如何解决?

缓存雪崩

缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。和缓存击穿不同的是， 缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决办法:

- 缓存过期时间设置成随机
- 热点数据考虑永不过期（定时刷新）
- 使用分布式缓存，防止单点故障缓存全部丢失

缓存穿透

缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id为“-1”的数据或id为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大。

解决办法:

- 空对象
- 布隆过滤器

缓存击穿

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力

解决办法:

- 热点数据永不过期(后台进程定时刷新)
- 加互斥锁

缓存有哪些淘汰策略?

- 先进先出策略 FIFO（First In，First Out）

  如果一个数据最先进入缓存中，则应该最早淘汰掉

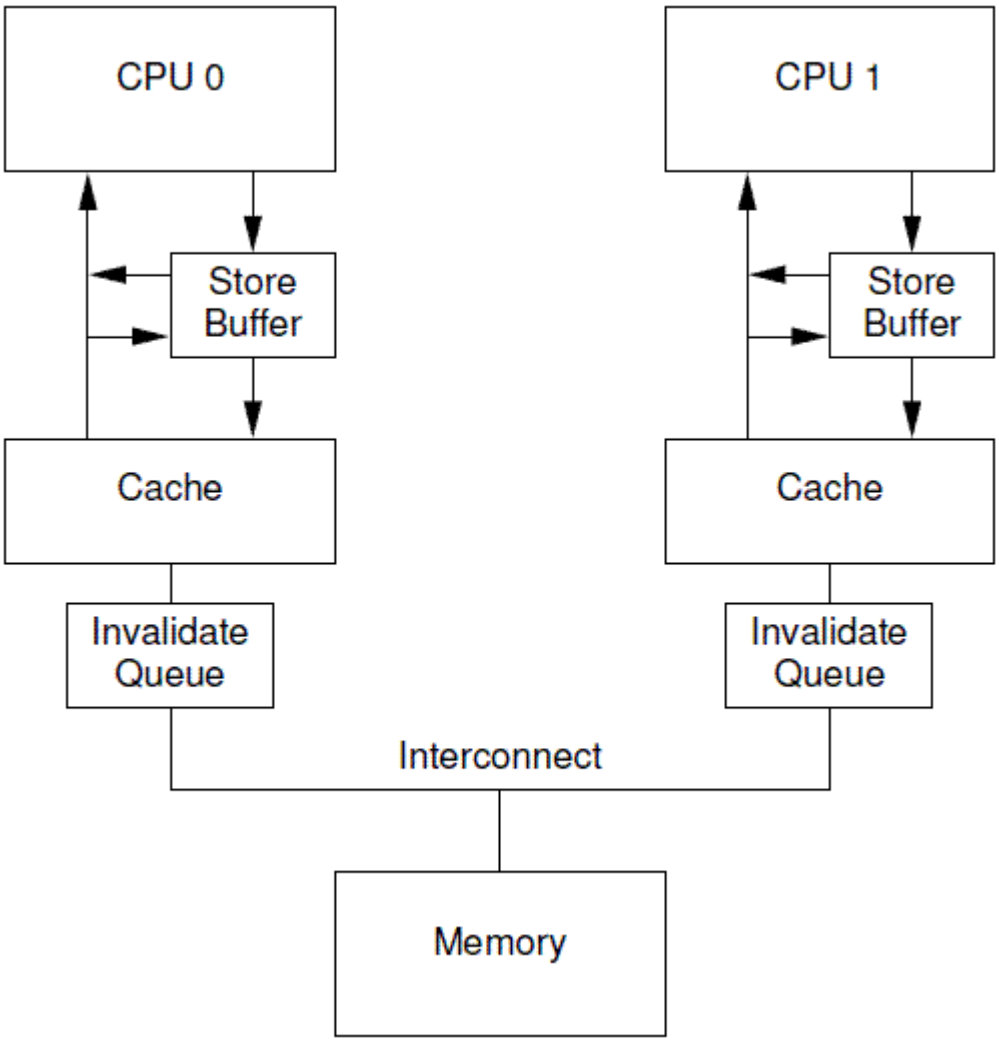
- 最近最少使用策略 LRU（Least Recently Used）

  如果数据最近被访问过，那么将来被访问的几率也更高。对于循环出现的数据，缓存命中不高。实际实现时候一般可采用双向链表，将最近访问过得缓存key放在链表首部，删除尾部的缓存key，再加上hash表来记录key-value，实现快速访问缓存

- 最少使用策略 LFU（Least Frequently Used）

  如果一个数据在最近一段时间内使用次数很少，那么在将来一段时间内被使用的可能性也很小。对于交替出现的数据，缓存命中不高

读写屏障是怎么回事?



内存屏障分为读屏障(rmb)与写屏障(wmb)。写屏障主要保证在写屏障之前的在Store buffer中的指令都真正的写入了缓存。读屏障主要保证了在读屏障之前所有Invalidate queue中所有的无效化指令都执行。有了读写屏障的配合，那么在不同的核心上，缓存可以得到强同步。

Nginx

Nginx工作模式?

Nginx采用master-worker模式，nginx启动成功后，会有一个master进程和至少一个worker进程；master进程负责处理系统信号、加载配置、管理worker进程；worker进程负责处理具体的业务逻辑。

nginx采用了异步非阻塞的工作方式，epoll模型：当有i/o事件产生时，epoll就会告诉进程哪个连接由i/o事件产生，然后进程就会处理这个事件。nginx配置use epoll后，以异步非阻塞的方式工作，能够处理百万计的并发连接。

才有master-worker模式的优缺点：

稳定性高：一个worker进程挂掉后master进程会立即启动一个新的worker进程，保证worker进程数量不变，降低服务中断的概率; 高性能： Nginx 启动 N 个 worker, 并将 worker 和 cpu 进行绑定，每个 worker 有自己的 epoll 和 定时器，由于没有进程、线程切换开销，性能非常好。配合Linux的cpu亲和性的匹配中，可以充分利用多核cpu的优势，提升性能; 支持平滑重启：处理信号、配置重新加载等可以做到尽可能不中断服务;

Nginx Location 路径匹配规则是怎么样的?

对于请求： http://example.com/static/img/logo.jpg

1. 如果命中精确匹配，例如：

```
location = /static/img/logo.jpg {

}
```

则优先精确匹配，并终止匹配。

2. 如果命中多个前缀匹配，例如：

```
location /static/ {

}

location /static/img/ {

}
```

则记住最长的前缀匹配，即上例中的 /static/img/，并继续匹配

3. 如果最长的前缀匹配是优先前缀匹配，即：

```
location /static/ {

}
```

```
location ^~ /static/img/ {

}
```

4. 否则，如果命中多个正则匹配，即：

```
location /static/ {

}

location /static/img/ {

}

location ~* /static/ {

}

location ~* /static/img/ {

}
```

则忘记上述 2 中的最长前缀匹配，使用第一个命中的正则匹配，即上例中的 location ~* /static/ ，并终止匹配（命中多个正则匹配，优先使用配置文件中出现次序的第一个）

5. 否则，命中上述 2 中记住的最长前缀匹配

Nginx的负载均衡策略有哪些?

负载均衡策略	说明
轮询(rr)	负载均衡默认策略
weight	权重方式，权重越高分配到需要处理的请求越多，此策略比较适合服务器的硬件配置差别比较大的情况。此策略可以与least_conn和ip_hash结合使用。
ip_hash	依据ip分配方式，基于客户端IP的分配方式，确保了相同的客户端的请求一直发送到相同的服务器，实现会话粘滞目的
least_conn	最少连接方式，把请求转发给连接数较少的后端服务器，可以达到更好的负载均衡效果
fair（第三方）	响应时间方式
url_hash（第三方）	依据URL分配方式

nginx基于权重轮询平滑算法是怎么实现的?

常规的基于权重的轮询调度算，假定a, b, c三台机器的负载能力分别是4:2:1，则可以给它们分配的权限为4, 2, 1。 这样轮询完一次后，a被调用4次，b被调用2次，c被调用1次。

对于普通的基于权重的轮询算法，可能会产生以下的调度顺序{a, a, a, a, b, b, c}。这样的调度顺序其实并不友好，它会一下子把大压力压到同一台机器上，这样会产生一个机器一下子很忙的情况。 于是乎，就有了平滑的基于权重的轮询算法。

所谓平滑就是调度不会集中压在同一台权重比较高的机器上。这样对所有机器都更加公平。 比如，对于{a:5, b:1, c:1}，产生{a, a, b, a, c, a, a}的调度序列就比{c, b, a, a, a, a, a} 更加平滑。

算法逻辑：

算法执行2步，选择出1个当前节点。

1. 用上次选择后的权重加上每个节点配置的权重，作为节点当前权重值，第一选择的时候，上次选择后的权重都是0
2. 选择当前权重值最大的节点为选中节点，并把它的当前值减去所有节点的权重总和，作为选择后的权重值

例如{a:5, b:1, c:1}三个节点。一开始我们初始化三个节点的当前值为{0, 0, 0}。 选择过程如下表：

轮数	当前权重	选择节点	选择后的权重
0	-	-	{0, 0, 0}
1	{5, 1, 1}	a	{-2, 1, 1}
2	{3, 2, 2}	a	{-4, 2, 2}
3	{1, 3, 3}	b	{1, -4, 3}
4	{6, -3, 4}	a	{-1, -3, 4}
5	{4, -2, 5}	c	{4, -2, -2}
6	{9, -1, -1}	a	{2, -1, -1}
7	{7, 0, 0}	a	{0, 0, 0}

我们可以发现，a, b, c选择的次数符合5:1:1，而且权重大的不会被连接选择。7轮选择后， 当前值又回到{0, 0, 0}，以上操作可以一直循环，一样符合平滑和基于权重。

nginx四层、七层负载均衡的有什么区别?

四层就是基于IP+端口的负载均衡，通过虚拟IP+端口接收请求，然后再分配到真实的服务器；

七层通过虚拟的URL或主机名接收请求，然后再分配到真实的服务器。七层就是基于URL等应用层信息的负载均衡。

七层负载均衡是：

```
# cat /etc/nginx/conf.d/test.conf
upstream phpserver {
    server192.168.2.3;
```

```
    server192.168.2.4;
}
upstream htmlserver {
    server192.168.2.1;
    server192.168.2.2;
}

# /etc/nginx/nginx.conf
location / {
    root  /usr/share/nginx/html;
    index  index.html index.htm;
    if ($request_uri ~*\.html$){
            proxy_pass http://htmlserver;
    }
    if ($request_uri~* \.php$){
            proxy_pass http://phpserver;
    }
}
```

四层负载均衡:

```
# vim nginx.conf
worker_processes  1;
events {
    worker_connections  1024;
}
stream {  # 类似于7层的http段
    upstream ssh_proxy {
        hash $remote_addr consistent;
        server 192.168.56.2:22;
        server 192.168.56.3:22;
    }
    server {
        listen 2222;
        proxy_connect_timeout 1s;
        proxy_timeout 3s;
        proxy_pass ssh_proxy;
    }
}
```

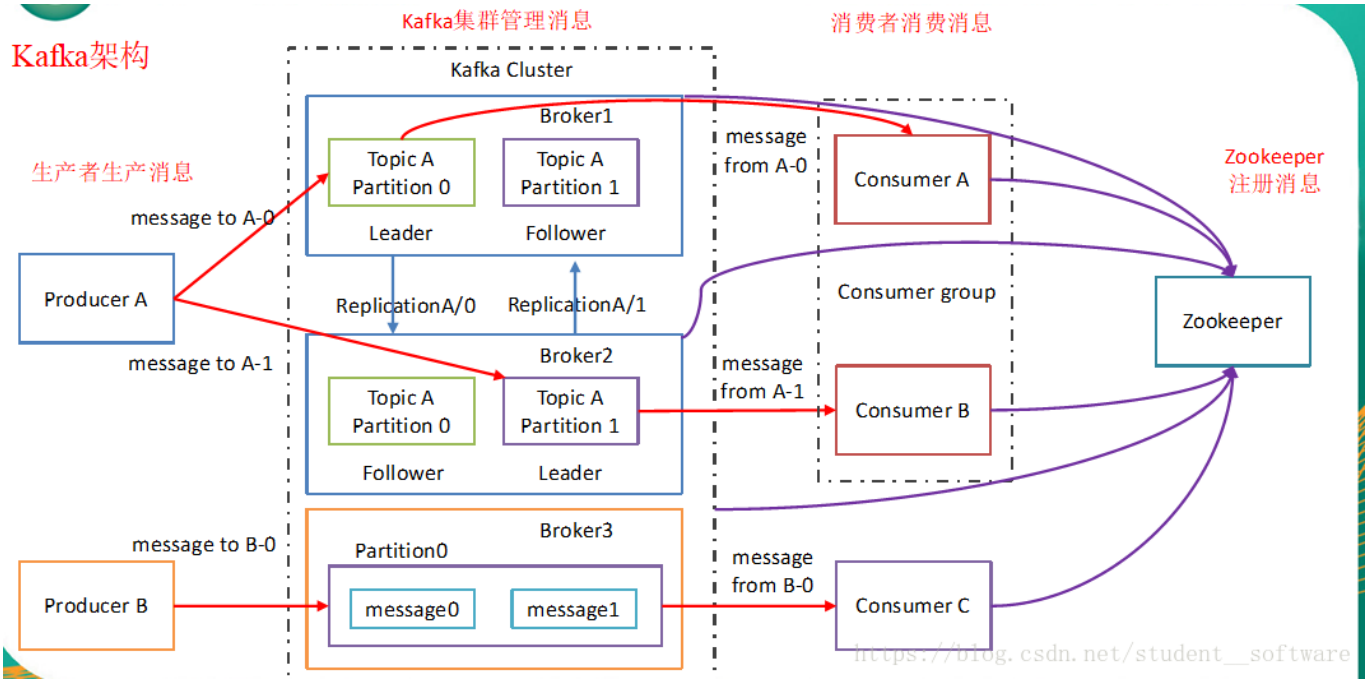
消息队列

为什么要使用消息队列?

1. 业务解耦
2. 异步处理
3. 流量削峰

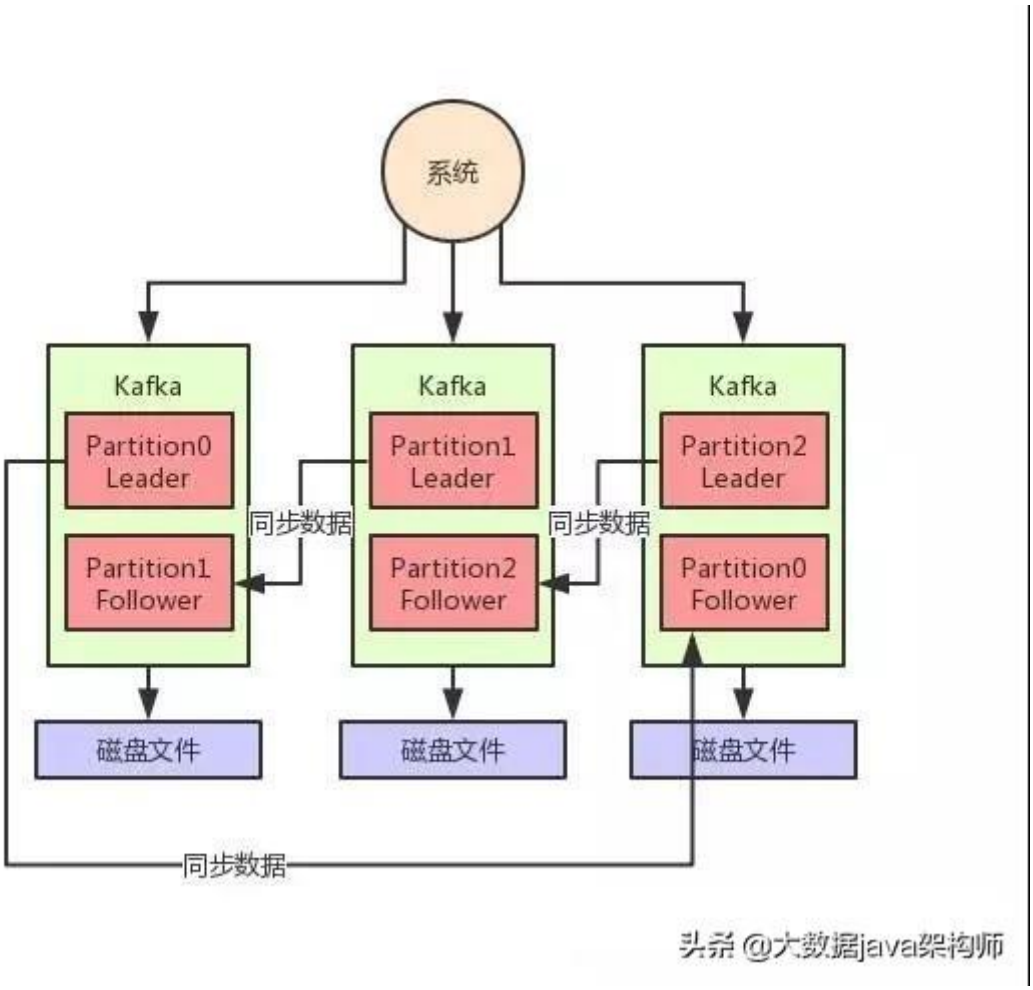
kafka是什么?

Kafka是高吞吐低延迟的高并发、高性能的消息中间件，配置良好的Kafka集群甚至可以做到每秒几十万、上百万的超高并发写入。Kafka是**一个分布式消息队列**。Kafka对消息保存时根据Topic进行归类，发送消息者称为Producer，消息接受者称为Consumer，此外kafka集群有多个kafka实例组成，每个实例(server)称为broker。无论是kafka集群，还是consumer都依赖于zookeeper集群保存一些meta信息，来保证系统可用性。



kafka如何做到高可用的?

从topic的Partition的副本来看：



上图中只有一个Topic，它有3个Partition。

Kafka 为什么不支持读写分离?

自 Kafka 2.4 之后，Kafka 提供了有限度的读写分离，也就是说，Follower 副本能够对外提供读服务。

1. 业务场景不适用。读写分离适用于那种读负载很大，而写操作相对不频繁的场景，可 Kafka 不属于这样的场景。
2. 同步机制。Kafka 采用 PULL 方式实现 Follower 的同步，因此，Follower 与 Leader 存 在不一致性窗口。如果允许读 Follower 副本，就势必要处理消息滞后(Lagging)的问题。

如何解决kafka消息重复消费问题?

将消息的唯一标识保存到外部介质中，每次消费时判断是否处理过即可。这个解决办法适合其他消息系统。

Kafka消息是采用Pull模式，还是Push模式?

kafka遵循了一种大部分消息系统共同的传统的设计：producer将消息推送到broker，consumer从broker拉取消息。同redis的bpop命令类似，Kafka有个参数可以让consumer阻塞知道新消息到达，可以防止consumer不断在循环中轮询。

kafka中如何防止消息丢失?

Kafka消息发送有两种方式：同步（sync）和异步（async），默认是同步方式，可通过producer.type属性进行配置。Kafka通过配置request.required.acks属性来确认消息的生产：

- 0---表示不进行消息接收是否成功的确认；
- 1---表示当Leader接收成功时确认，默认状态
- -1---表示Leader和Follower都接收成功时确认；

综上所述，有6种消息生产的情况，下面分情况来分析消息丢失的场景：

（1）acks=0，不和Kafka集群进行消息接收确认，则当网络异常、缓冲区满了等情况时，消息可能丢失；

（2）acks=1、同步模式下，只有Leader确认接收成功后但挂掉了，副本没有同步，数据可能丢失；

可见在同步模式下，ack=-1时候，可以防止消息丢失。但这牺牲了吞吐量。

kafka中的 zookeeper 起到什么作用，可以不用zookeeper吗?

早期版本的kafka用zk做meta信息存储，consumer的消费状态，group的管理以及 offset的值。新的consumer使用了kafka内部的group coordination协议，也减少了对zookeeper的依赖，但是broker依然依赖于ZK，zookeeper 在kafka中还用来选举controller 和 检测broker是否存活等等。

kafka 为什么那么快?

1. Page cache技术

   Kafka每次接收到数据都会往磁盘上去写。但并不是直接写入磁盘的，而是写入OS cache上面，然后在写到磁盘

2. 顺序读写磁盘

   磁盘读写时候，是顺序读写的。此时数据在磁盘上存取代价为O(1)。

3. 零拷贝技术

   Customer从broker读取数据，采用零拷贝技术。将磁盘文件读到OS内核缓冲区后，直接转到socket buffer进行网络发送。

   传统的数据发送需要发送4次上下文切换，采用sendfile系统调用之后，数据直接在内核态交换，系统上下文切换减少为2次。

Kafka中是怎么体现消息顺序性的?

kafka每个partition中的消息在写入时都是有序的，消费时，每个partition只能被每一个group中的一个消费者消费，保证了消费时也是有序的。整个topic不保证有序。如果为了保证topic整个有序，那么将partition调整为1。

kakafka如何实现延迟队列?

kafka基于**时间轮**可以将插入和删除操作的时间复杂度都降为O(1)。

kafka中consumer group 是什么概念?

consumer group是Kafka实现单播和广播两种消息模型的手段。同一个topic的数据，会广播给不同的group；同一个group中的worker，只有一个worker能拿到这个数据。换句话说，对于同一个topic，每个group都可以拿到同样的所有数据，但是数据进入group后只能被其中的一个worker消费。group内的worker

可以使用多线程或多进程来实现，也可以将进程分散在多台机器上，worker的数量通常不超过partition的数量，且二者最好保持整数倍关系，因为Kafka在设计时假定了一个partition只能被一个worker消费（同一group内）。

Kafka 中位移(offset)的作用?

在 Kafka 中，每个 主题分区下的每条消息都被赋予了一个唯一的 ID 数值，用于标识它在分区中的位置。这个 ID 数值，就被称为位移，或者叫偏移量。一旦消息被写入到分区日志，它的位移值将不能 被修改。

阐述下Kafka 中的领导者副本(Leader Replica)和追随者副本 (Follower Replica)的区别?

Kafka 副本当前分为领导者副本和追随者副本。只有 Leader 副本才能 对外提供读写服务，响应 Clients 端的请求。Follower 副本只是采用拉(PULL)的方 式，被动地同步 Leader 副本中的数据，并且在 Leader 副本所在的 Broker 宕机后，随时准备应聘 Leader 副本。

自 Kafka 2.4 版本开始，社区通过引入新的 Broker 端参数，允许 Follower 副本有限度地提供读服务。

消息传递语义是什么概念?

message delivery semantic 也就是消息传递语义。通用的概念，也就是消息传递过程中消息传递的保证性。分为三种：

- 最多一次（at most once）

  消息可能丢失也可能被处理，但最多只会被处理一次。

  可能丢失 不会重复

- 至少一次（at least once）

  消息不会丢失，但可能被处理多次。

  可能重复 不会丢失

- 精确传递一次（exactly once）

  消息被处理且只会被处理一次。

  不丢失 不重复 就一次

IO

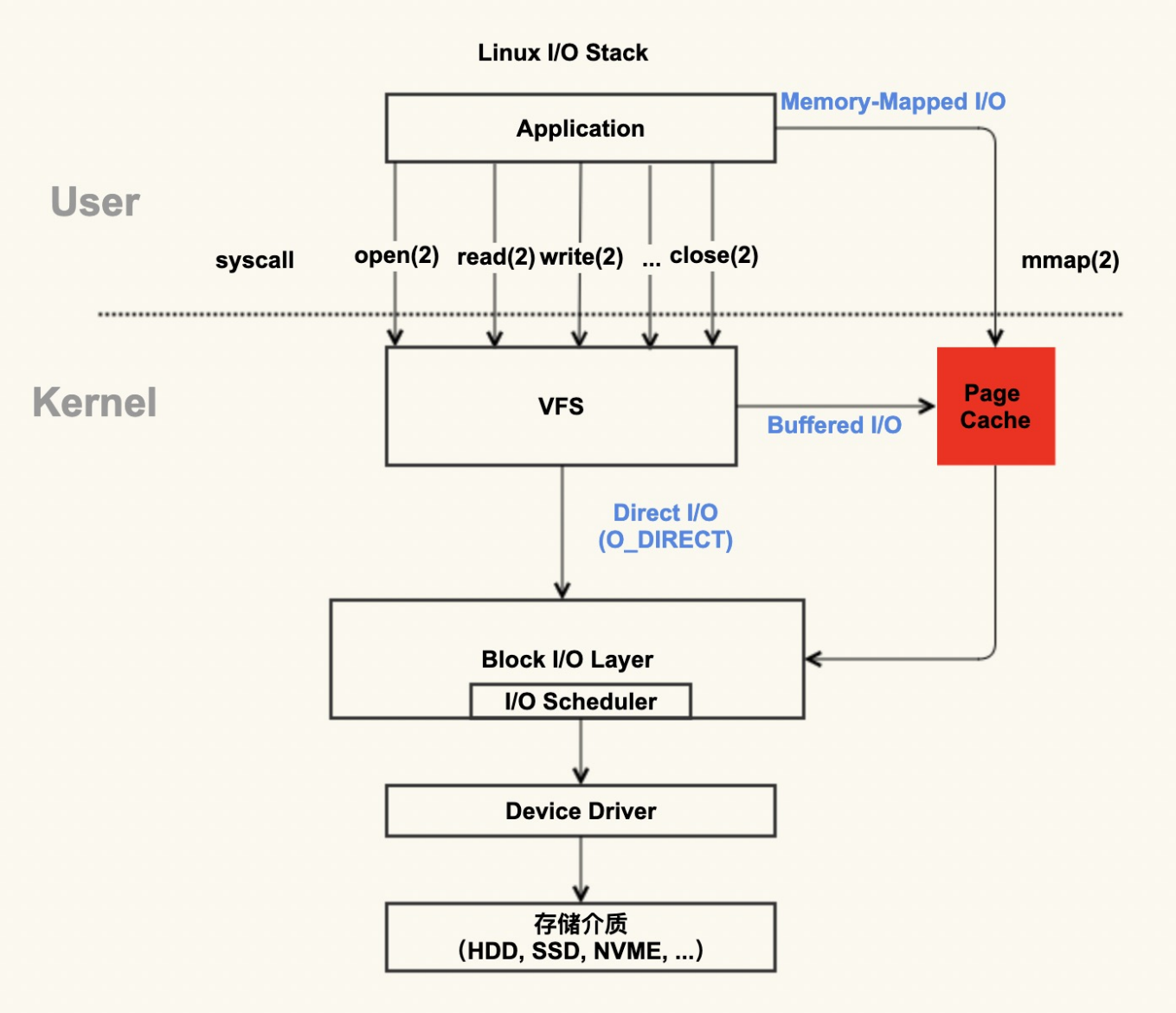
Page Cache是怎么回事?

Page cache是通过将磁盘中的数据缓存到内存中，从而减少磁盘I/O操作，从而提高性能，这个内存就是Page Cache。Page Cache中被修改的内存页称之为脏页（Dirty Page），脏页在特定的时候被一个叫做pdflush(Page Dirty Flush)的内核线程写入磁盘。

Page Cache中写入方式称为Write back（写回），属于异步方式，即写入内存中即返回，它属于buffered I/O。若写入磁盘之后才返回就是Write Through（写穿），它属于direct I/O。

Page Cache缺点就是会导致数据丢失。为了解决这个问题可以使用WAL技术（Write-Ahead Log）,在数据库中一般又称之为redo log。WAL日志是append模式，写入速度是O(1)。

Page Cache与mmap(memory-maped I/O)区别：



mmap是怎么回事?

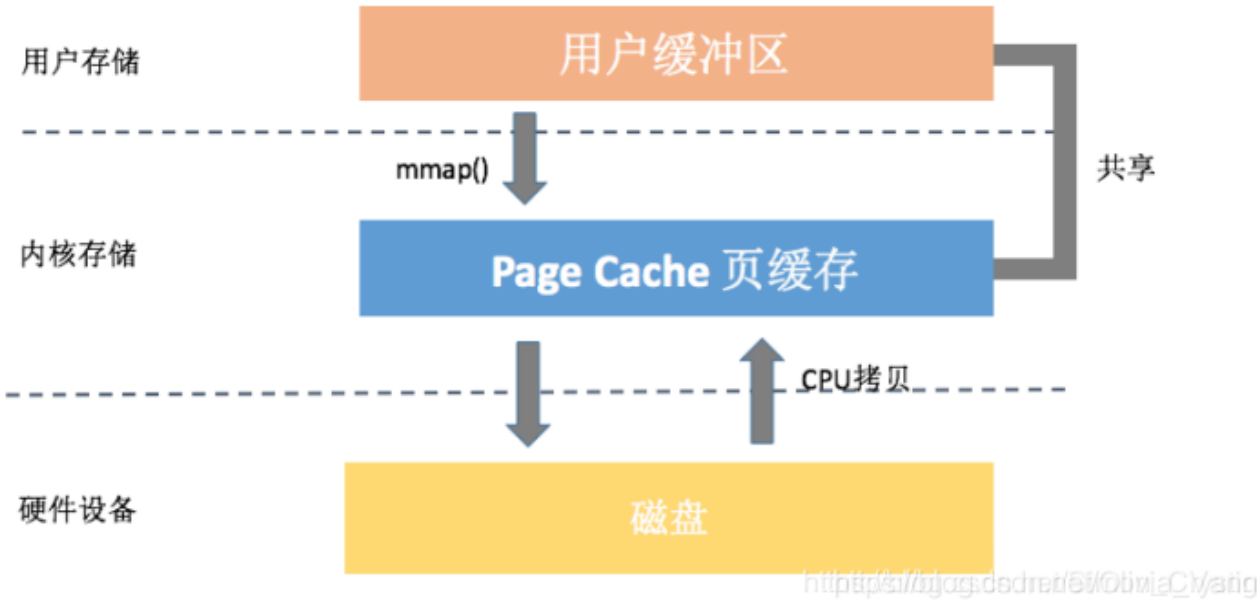
Page Cache的写入时候，需要从应用缓冲区拷贝到内核缓冲区(即Page Cache)。为了避免这样，可以使用mmap技术。

mmap 把文件映射到用户空间里的虚拟地址空间，实现文件和进程虚拟地址空间中一段虚拟地址的一一对映关系。它省去了从内核缓冲区复制到用户空间的过程，进程就可以采用指针的方式读写操作这一段内存（文件 / page cache）。而系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作。相反，内核空间对这段区域的修改也直接反映到用户空间，从而可以实现用户态和内核态对此内存区域的共享。

但在真正使用到这些数据前却不会消耗物理内存，也不会有读写磁盘的操作，只有真正使用这些数据时，虚拟内存管理系统 VMS 才根据缺页加载的机制从磁盘加载对应的数据块到内核态的Page Cache。这样的文件读写文件方式少了数据从内核缓存到用户空间的拷贝，效率很高。

mmap有以下特点：

- 文件（page cache）直接映射到用户虚拟地址空间，内核态和用户态共享一片page cache，避免了一次数据拷贝
- 建立mmap之后，并不会立马加载数据到内存，只有真正使用数据时，才会引发缺页异常并加载数据到内存



网络IO模型有哪些?

阻塞I/O

从发起系统调用（比如read)时候，从等待数据到复制到内核和从内核复制到用户态，全程阻塞。

非阻塞I/O

从发起系统调用之后，无需等待，通过轮询方式获取状态，数据准备阶段是fe非阻塞的，而从内核拷贝到用户空间是阻塞的

I/O多路复用

监听多个IO对象，当IO对象有数据时候，通知用户进程。

异步I/O

发起系统调用后等待数据到达和数据从内核复制到用户态两个io阶段都是非阻塞的

I/O多路复用中select/poll/epoll的区别?

select/poll属于一类，传给一组文件描述符数组，返回准备就绪的文件描述符数组，select有大小限制(1024)，poll则没有，他们每次都要传文件描述符数组，比较低效，epoll在内核开辟一个空间存放描述符，无须频繁的从用户空间传递给内核

僵尸进程与孤儿进程区别?

僵尸进程指的是一个父进程利用fork创建子进程，如果子进程退出，而父进程没有利用wait或者 waitpid来获取子进程的状态信息，那么子进程的状态描述符依然保存在系统中。

孤儿进程指的是一个父进程退出， 而它的一个或几个子进程仍然还在运行，那么这些子进程就会变成孤儿进程，孤儿进程将被init进程（进程号为1）所收养，并由init进程对它们完成状态收集的工作

多进程通信方式有哪些?

1. 共享内存
2. 消息队列
3. 信号
4. 管道
5. socket

Protobuf

Protobuf编码相比Json的优点有哪些?

1. 具有强一致性
2. 占用空间小，更高效

Protobuf编码存储方式是?

Wire Type 值	编码方式	编码长度	存储方式	代表的数据类型
0	Varint （负数时以Zigzag辅助编码）	变长（1–10个字节）	T – V	• int32, int64, uint32, uint64,bool, enum • sint32, sint64(负数时使用)
1	64–bit	固定8个字节		fixed64, sfixed64, double
2	Length–delimi	变长	T – L – V	string, bytes, embedded messages, packed repeated fields
3	Start group	已弃用	已弃用	Groups (已弃用)
4	End group			
5	32–bit	固定4个字节	T – V	fixed32, sfixed32, float

Protobuf采用的是`Tag – Length – Value`，即`标识 – 长度 – 字段值`

varint是一种变长编码方式，用来字节编码数字，每个字节的高位表示后面的那个字节是否是数字的一部分。

varint缺点是如果用来编码一个负数，一定需要5个byte。因为负数最高位是1，会被当做很大的整数去处理。解决办法是采用zigzag编码，即将负数转换成正数，然后再采用varint编码。

Tag是标识，是将字段编号左移三位之后与字段类型或运算之后产生：

```
Tag  = (field_number << 3) | wire_type
```

Go

互斥锁(Mutex)有哪两种模式?

Mutex 可能处于两种操作模式下：正常模式和饥饿模式。

正常模式下，waiter 都是进入先入先出队列，被唤醒的 waiter 并不会直接持有锁，而是要和新来的 goroutine 进行竞争。新来的 goroutine 有先天的优势，它们正在 CPU 中运行，可能它们的数量还不少，所以，在高并发情况下，被唤醒的 waiter 可能比较悲剧地获取不到锁， 这时，它会被插入到队列的前面。如果 waiter 获取不到锁的时间超过阈值 1 毫秒，

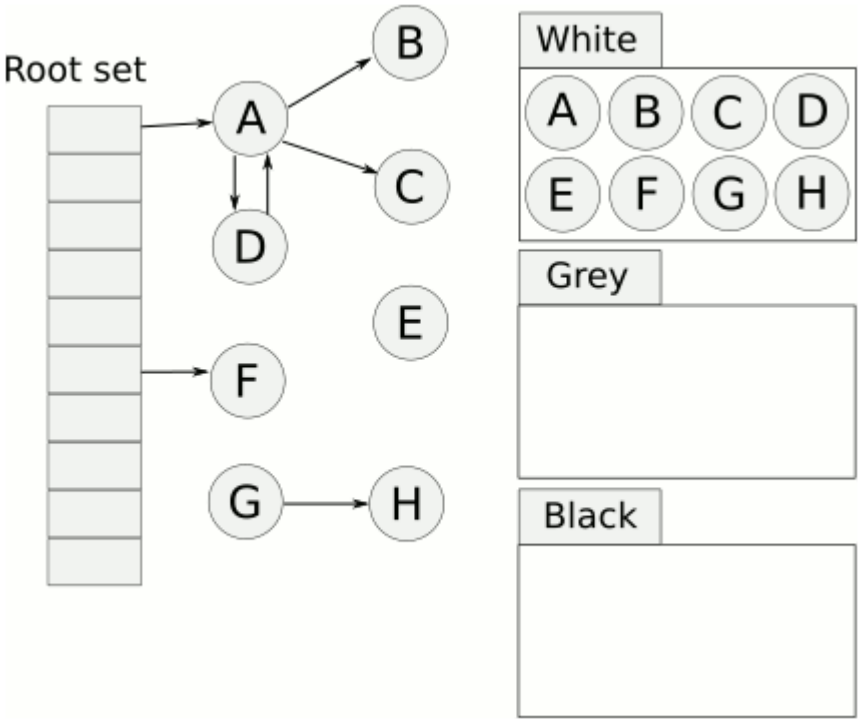
那么，这个 Mutex 就进入到了饥饿模式。在饥饿模式下，Mutex 的拥有者将直接把锁交给队列最前面的 waiter。新来的 goroutine 不会尝试获取锁，即使看起来锁没有被持有，它也不会去抢，也不会 spin，它会乖乖地加入到等待队列的尾部。

如果拥有 Mutex 的 waiter 发现下面两种情况的其中之一，它就会把这个 Mutex 转换成正常模式:

- 此 waiter 已经是队列中的最后一个 waiter 了，没有其它的等待锁的 goroutine 了；
- 此 waiter 的等待时间小于 1 毫秒。

饥饿模式是对公平性和性能的一种平衡，它避免了某些 goroutine 长时间的等待锁。在饥饿模式下，优先对待的是那些一直在等待的 waiter

Go垃圾清理的三色标记法?



三色标记法是传统 Mark-Sweep 的一个改进，它是一个并发的 GC 算法。 原理如下，

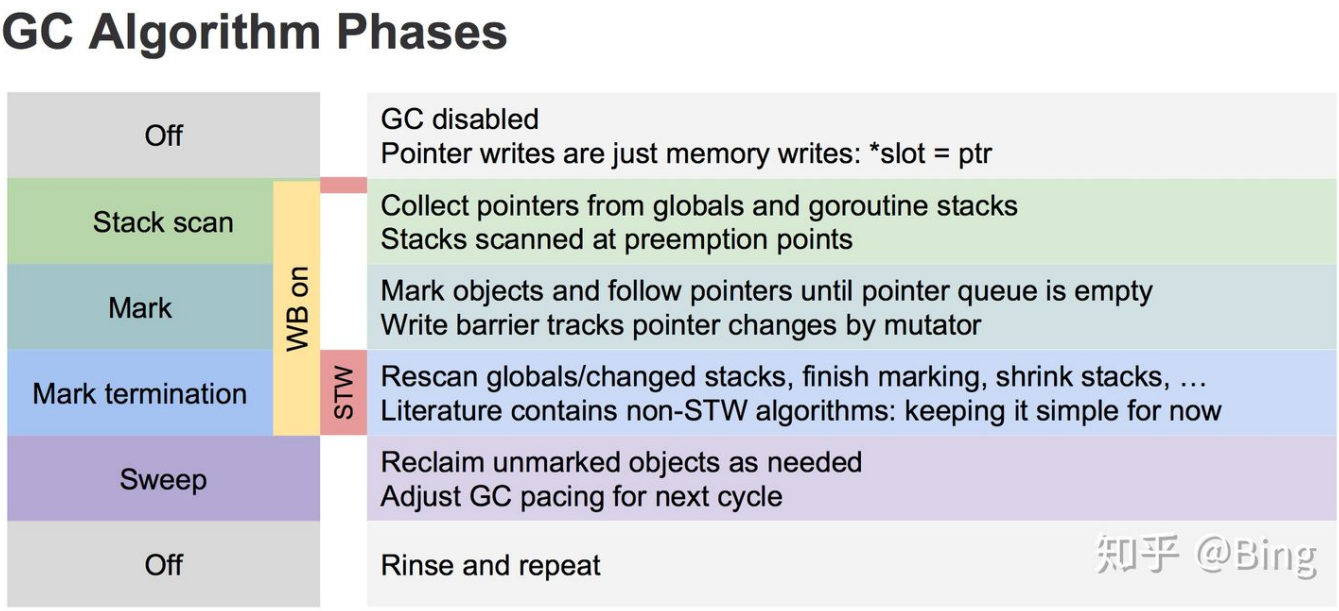
- 首先创建三个集合：白、灰、黑。
- 将所有对象放入白色集合中。

- 然后从根节点开始遍历所有对象（注意这里并不递归遍历），把遍历到的对象从白色集合放入灰色集合。
- 之后遍历灰色集合，将灰色对象引用的对象从白色集合放入灰色集合，之后将此灰色对象放入黑色集合 重复 4 直到灰色中无任何对象
- 通过write-barrier检测对象有变化，重复以上操作
- 收集所有白色对象（垃圾）

Golang什么时候会触发GC?

- 阈值：默认内存扩大一倍，启动gc
- 定期：默认2min触发一次gc，src/runtime/proc.go:forcegcperiod
- 手动：runtime.gc()

Golang进行GC时候会不会STW?



Golang使用的是三色标记法方案，并且支持并行GC，即用户代码何以和GC代码同时运行。具体来讲，Golang GC分为几个阶段:Mark阶段该阶段又分为两个部分：

- Mark Prepare：初始化GC任务，包括开启写屏障(write barrier)和辅助GC(mutator assist)，统计root对象的任务数量等，这个过程需要STW。
- GC Drains: 扫描所有root对象，包括全局指针和goroutine(G)栈上的指针（扫描对应G栈时需停止该G)，将其加入标记队列(灰色队列)，并循环处理灰色队列的对象，直到灰色队列为空。该过程后台并行执行。
- Mark Termination阶段：该阶段主要是完成标记工作，重新扫描(re-scan)全局指针和栈。因为Mark和用户程序是并行的，所以在Mark过程中可能会有新的对象分配和指针赋值，这个时候就需要通过写屏障（write barrier）记录下来，re-scan 再检查一下，这个过程也是会STW的。Sweep: 按照标记结果回收所有的白色对象，该过程后台并行执行。
- Sweep Termination: 对未清扫的span进行清扫, 只有上一轮的GC的清扫工作完成才可以开始新一轮的GC。

总结一下，Golang的GC过程有两次STW:第一次STW会准备根对象的扫描, 启动写屏障(Write Barrier)和辅助GC(mutator assist).第二次STW会重新扫描部分根对象, 禁用写屏障(Write Barrier)和辅助GC(mutator assist).

Go内存分配策略是什么样子的?

Golang内存分配管理策略是根据对象大小区分和不同的内存分配层级来分配管理内存。Golang中内存分配管理的对象按照大小可以分为：

类别	大小
微对象 tiny object	(0, 16B)
小对象 small object	[16B, 32KB]
大对象 large object	(32KB, +∞)

Golang中内存管理的层级从最下到最上可以分为：mspan -> mcache -> mcentral -> mheap -> heapArena。golang中对象的内存分配流程如下：

- 小于16个字节的对象使用mcache的微对象分配器进行分配内存
- 大小在16个字节到32k字节之间的对象，首先计算出需要使用的span大小规格，然后使用mcache中相同大小规格的mspan分配
- 如果对应的大小规格在mcache中没有可用的mspan，则向mcentral申请
- 如果mcentral中没有可用的mspan，则向mheap申请，并根据BestFit算法找到最合适的mspan。如果申请到的mspan超出申请大小，将会根据需求进行切分，以返回用户所需的页数，剩余的页构成一个新的mspan放回mheap的空闲列表
- 如果mheap中没有可用span，则向操作系统申请一系列新的页（最小 1MB）
- 对于大于32K的大对象直接从mheap分配

new和make的区别?

传递给 new 函数的是一个类型，不是一个值。返回值是指向这个新分配的零值的指针。

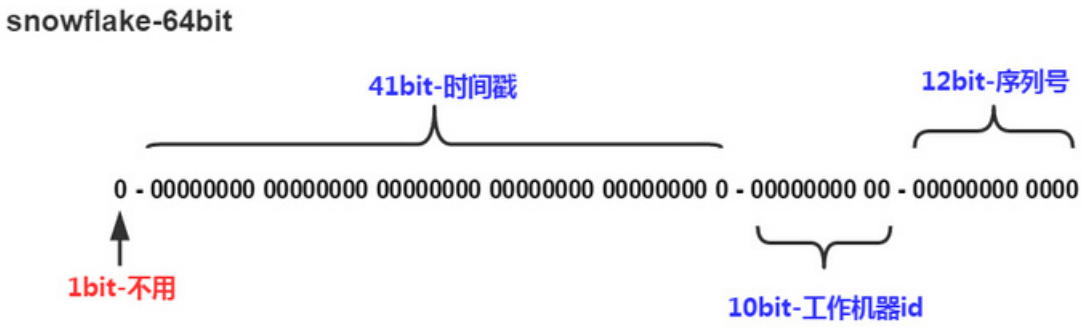
make 的作用是为创建slice，map 或 chan。

分布式

分布式ID生成方案有哪些?

- Redis Incr命令
- Mongodb ObjectID
- 雪花算法

  Snowflake算法(雪花算法)是由Twitter提出的一个分布式全局唯一ID生成算法，该算法生成一个64bit大小的长整数。64bit位ID结构如下：



一致性Hash算法

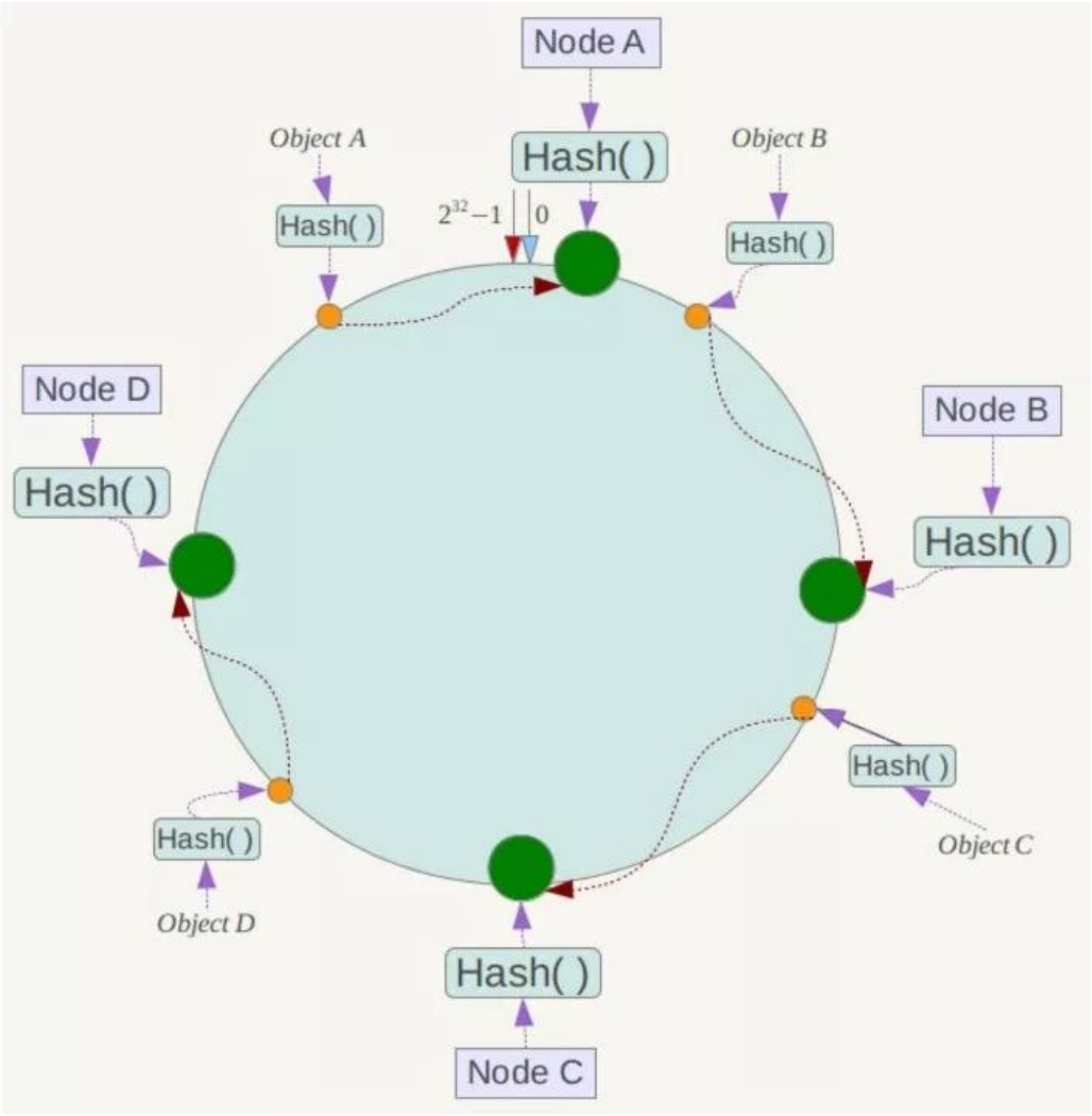
一致性Hash算法是为了解决传统Hash算法（比如取余运算）时候，由于添加或删删除节点时候，导致过多数据进行迁移问题而引入的新算法。

假定有几台Redis服务器，假定是N，要存储key-value数据，传统模式是根据hash(key)%N服务器数量来定位出来存放在第几台服务器上面。

一致性Hash算法，会使用2^32个虚拟hash槽位，可以想象成在一个圆环（也叫hash环）上面有0到2^32-1编号的槽位。首先我们确定每台Redis服务器在这个环上槽位，可以用服务器IP或ID或者name进行hash：

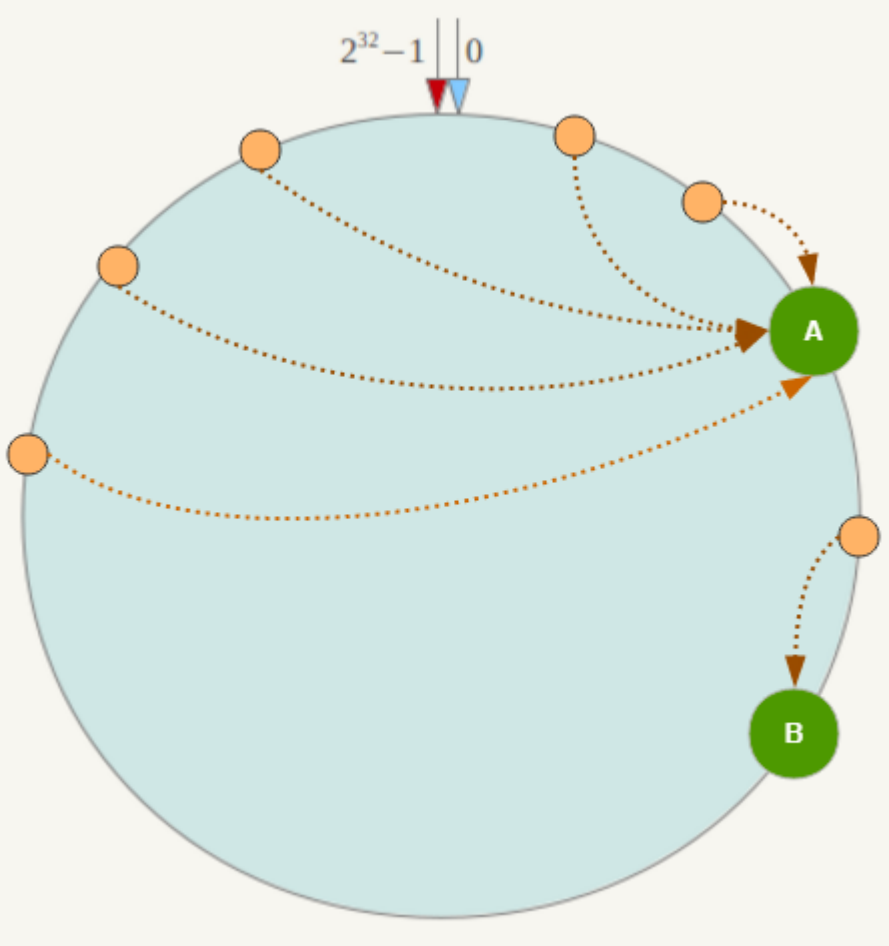
> 槽位 = hash(Redis服务器IP)%2^32

当一个key-value数据过来时候，根据key计算出其在环上槽位，然后沿着环顺时针行走，遇到的第一个服务器就是要存放的服务器。



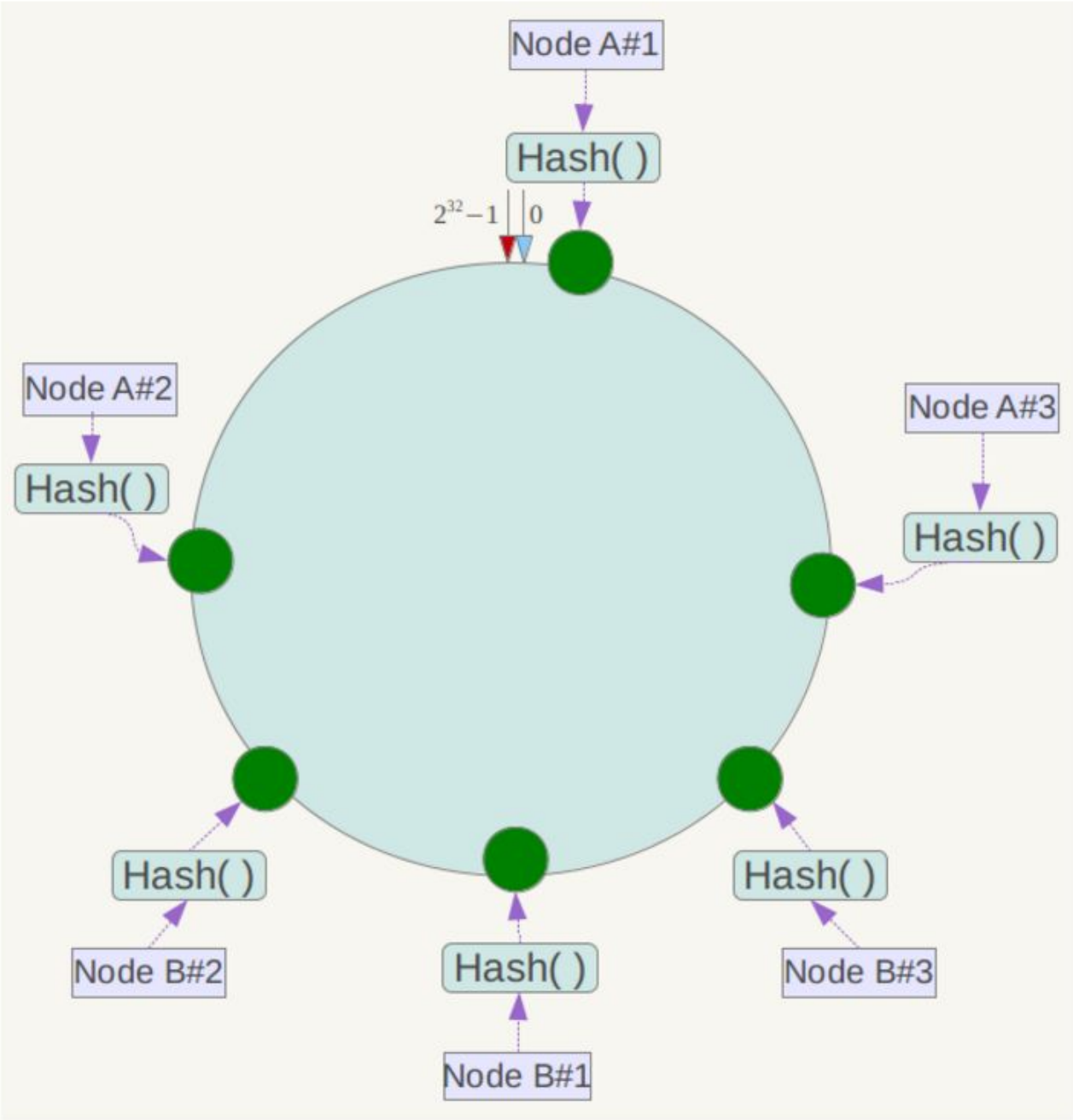
如上图所示：根据一致性Hash算法，数据A会被定为到Node A上，B被定为到Node B上，C被定为到Node C上，D被定为到Node D上。

生倾斜。一致性Hash算法在服务节点太少时，容易因为节点分部不均匀而造成**数据倾斜**（被缓存的对象大部分集中缓存在某一台服务器上）问题，例如系统中只有两台服务器，其环分布如下：



此时必然造成大量数据集中到Node A上，而只有极少量会定位到Node B上。为了解决这种数据倾斜问题，一致性Hash算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。具体做法可以在服务器IP或主机名的后面增加编号来实现。

例如上面的情况，可以为每台服务器计算三个虚拟节点，于是可以分别计算 “Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点：

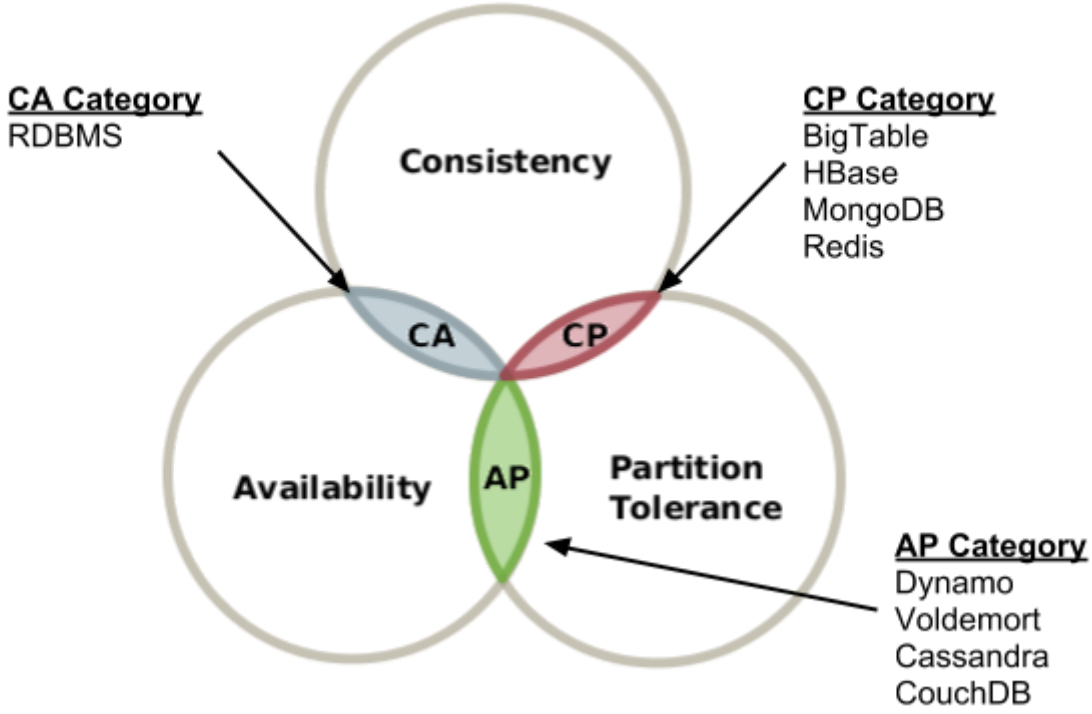


什么是分布式一致性?

分布式一致性（Distributed Consensus）简单来说就是在多个节点组成系统中各个节点的数据保持一致，并且可以承受某些节点数据不一致或操作失败造成的影响。分布式一致性是分布式系统的基石。

Cap理论是什么?

Cap理论中C代表一致性，A代表可用性，P代表分区容忍性。CAP理论下分布式系统在满足P情况下，需要在C和A之间找到平衡。



根据C的情况，数据一致性模型分为：

- 强一致性

  新的数据一旦写入，在任意副本任意时刻都能读到新值。强一致性使用的是同步复制，即某节点受到请求之后，必须保证其他所有节点也全部完成同样操作，才算这次请求成功完成。

- 弱一致性

  不同副本上的值有新有旧，这需要应用方做更多的工作获取最新值

- 最终一致性

  各副本的数据最终将达到一致。一般使用异步复制，这意味有更好的性能，但需要更复杂的状态控制

什么是Raft算法?

Raft算法属于强一致性算法实现。在Raft中，每个节点在同一时刻只能处于以下三种状态之一：

- 领导者(Leader)
- 候选者(Candidate)
- 跟随者(Follower)

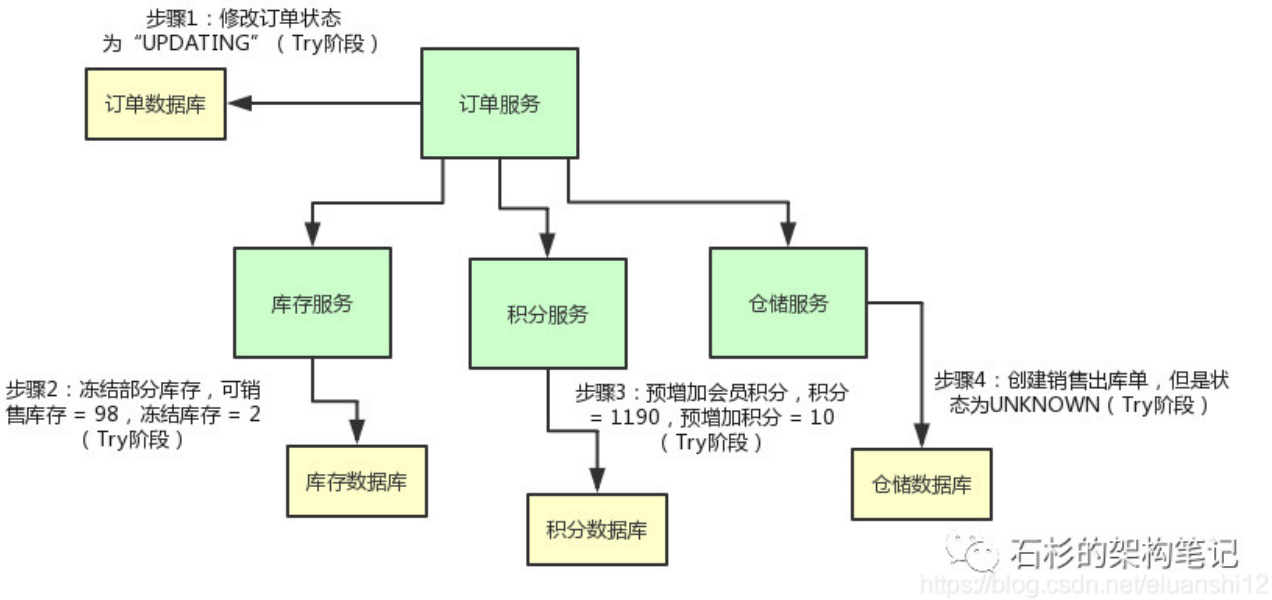
在Raft中Leader负责所有数据的读写，Follower只能用来接受Leader的Replication log。当一个节点刚开始启动时候默认都是Follwer状态，若在一段时间内，没有收到Leader的心跳，就会开始Leader Election过程：该节点会变成Canidate，将当前term技术加+1，同时投个自己一票，然后向其他节点发起投票请求，若获得集群中半数以上投票，则该节点会成Leader，然后开始向集群中发布心跳。

如何实现分布式事务?

分布式事务可以采用TCC来处理分布式事务。TCC核心思想就是针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。分为三个阶段：

- Try 阶段:主要是对业务系统做检测（一致性）及资源预留（准隔离性）
- Confirm 阶段:主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶段时，默认Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。(Confirm 操作满足幂等性。要求具备幂等设计，Confirm 失败后需要进行重试。)
- Cancel 阶段：主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。(Cancel 操作满足幂等性)

Try阶段进行资源预留示意图：



国内有一些关于TCC方案介绍的文章中，把TCC分成三种类型：

- 通用型TCC
    - 服务需要提供try、confirm、cancel
- 补偿性TCC 业务服务只需要提供 Do 和 Compensate 两个接口
- 异步确保型TCC

  主业务服务的直接从业务服务是可靠消息服务，而真正的从业务服务则通过消息服务解耦，作为消息服务的消费端，异步地执行。异步确保型 TCC中业务服务不需要提供try、confirm、cancel三个接口

限流算法怎么实现?

熔断器工作原理是?

什么是Sidecar模式?

想象一下假如你有6个微服务相互通信以确定一个包裹的成本。

每个微服务都需要具有可观察性、监控、日志记录、配置、断路器等功能。所有这些功能都是根据一些行业标准的第三方库在每个微服务中实现的。

但再想一想，这不是多余吗？它不会增加应用程序的整体复杂性吗?

这时候可以使用Sidecar模式，将单个服务的所有传入和传出网络流量都流经 Sidecar 代理。 因此，Sidecar 能够管理微服务之间的流量，收集遥测数据并实施相关策略。从某种意义上说，该服务不了解整个网络，只知道附加的 Sidecar 代理。这实际上就是 Sidecar 模式如何工作的本质——将网络依赖性抽象为 Sidecar。

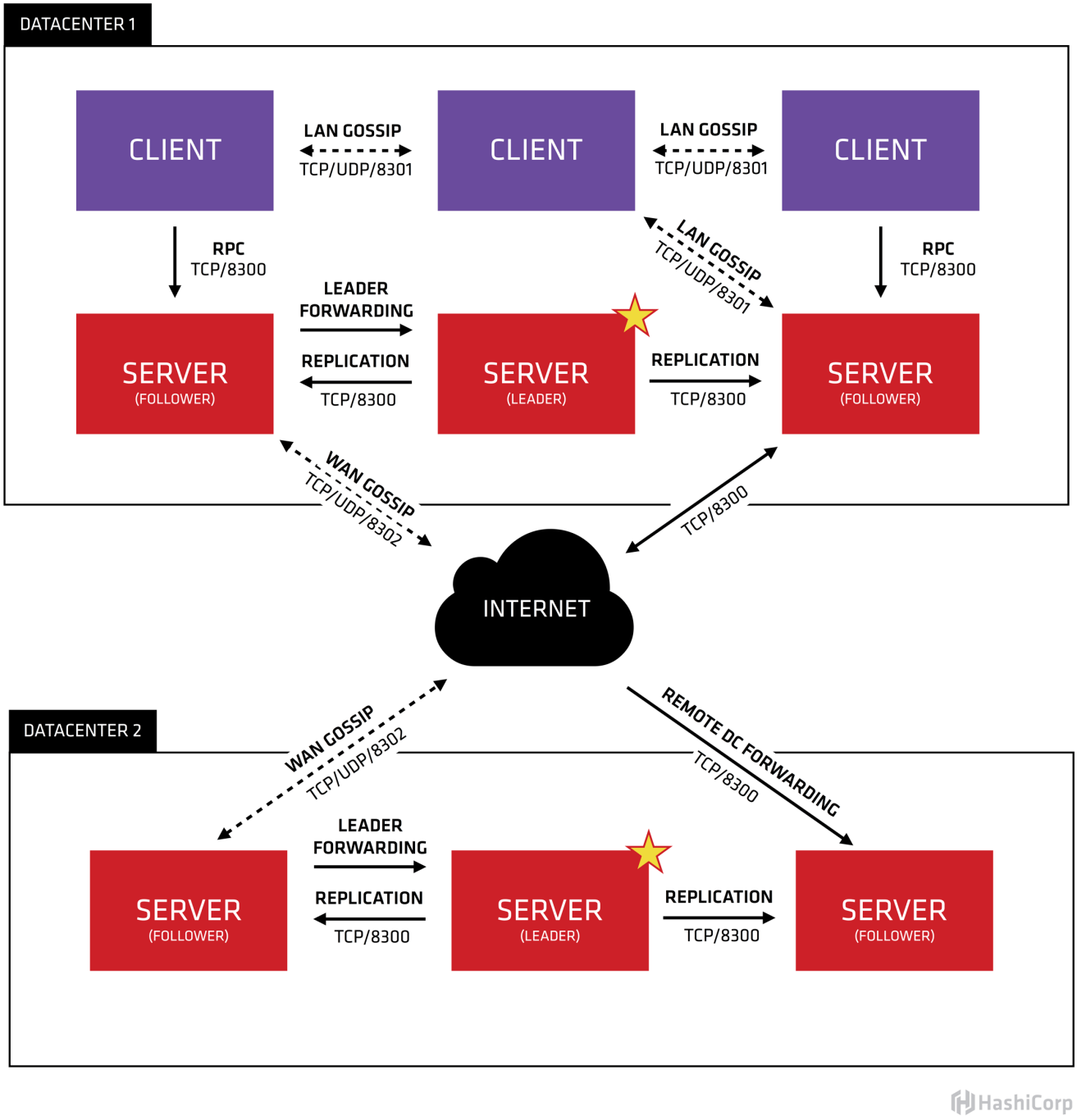
在sidecar上，可以把日志、微服务注册、调用链、限流熔断降级等功能都实现，基于sidecar，抽象出servicemesh。

什么是Service mesh

服务网格（Service Mesh）是一种在分布式软件系统中管理服务对服务（service-to-service）通信的技术。服务网格管理东西向类型的网络通信（East-west traffic）。服务网格中有数据平面和控制平面的概念：

- 数据平面的职责是处理网格内部服务之间的通信，并负责服务发现、负载均衡、流量管理、健康检查等功能。
- 控制平面的职责是管理和配置 Sidecar 代理以实施策略并收集遥测

Consul架构是怎么样的?



- Agent是Consul集群中的守护进程。它的生命周期从启动Consul agent开始。Agent可以以client或是server模式运行
- Client：Client是转发所有RPC请求道Server的Agent。Client相对来说是无状态的，它的唯一后台活动是参与LAN gossip pool。它的资源开销很小，只消耗少量的网络带宽
- Server：Server是负责参与Raft quorum，维护集群状态，响应RPC查询，和其他datacenter交换WAN gossip信息并转发查询到leader或是远程datacenter的Agent
- Gossip：Consul是建立在处于多种目的提供了完整gossip 协议的Serf之上的。Serf提供了成员管理、错误检测、时间传播等特性。我们只需要知道gossip会触发随机的点到点通信，主要基于UDP协议
- LAN Gossip：位于相同本地网络区域或是datacenter的节点之间的LAN gossip pool

- WAN Gossip：只包含server的WAN gossip pool。这些server主要存在于不同的datacenter中，并且通常通过internet或广域网进行通信

ElasticSearch

什么是ElasticSearch?

Elasticsearch是基于Apace Lunence构建的开源，分布式，具有高可用性和高拓展性的全文检索引擎。Elasticsearch具有开箱即用的特性，提供RESTful接口，是面向文档的数据库，文档存储格式为JSON，可以水平扩展至数以百计的服务器存储来实现处理PB级别的数据。

ES有哪些节点类型?

一个节点就是一个Elasticsearch的实例，每个节点需要显示指定节点名称，可以通过配置文件配置，或者启动时候`-E node.name=node1`指定

节点类型

每个节点在集群承担承担不同的角色，也可以称为节点类型。

候选主节点(Master-eligible nodes)和主节点(Master Node)

- 每个节点启动之后，默认就是一个Master eligible节点，Master-eligible节点可以参加选主流程，成为Master节点
- 当第一个节点启动时候，它会将自己选举成为Master节点
- 在每个节点上都保存了集群的状态信息，但**只有Master节点才能修改集群的状态信息**。集群状态(Cluster State)中必要信息包含
  - 所有节点的信息
  - 所有的索引，以及其Mapping与Setting信息
  - 分片的路由信息

数据节点(Data Node)和协调节点(Coordinating Node)和Ingest节点

- Data Node
  - **用于保存数据的节点。负责保存分片的数据，在数据拓展上起到至关重要的作用**
- Coorination Node
  - **负责接受Client的请求，将请求分发到合适的节点，最终把结果汇集到一起**
  - **每个节点默认都起到Cooridinating Node的职责**，这就意味着如果一个node，将node.master，node.data，node.ingest全部设置为false，那么它就是一个纯粹的coordinating Node node，仅仅用于接收客户端的请求，同时进行请求的转发和合并
- Ingest节点
  - 用于预处理，可以运行pipeline脚本，用来对document写入索引文件之前进行预处理的

在生产环境部署上可以部署独立(dedicate)的 Ingest Node 和 Coordinate node，在前端的Load Balance前面增加转发规则把读分发到coording node，写分发到 ingest node。 如果集群负载不高，可以配置一些节点同时具备coording和ingest的能力。然后将读写全部路由到这些节点。不仅配置简单，还节约硬件成本

其他类型节点

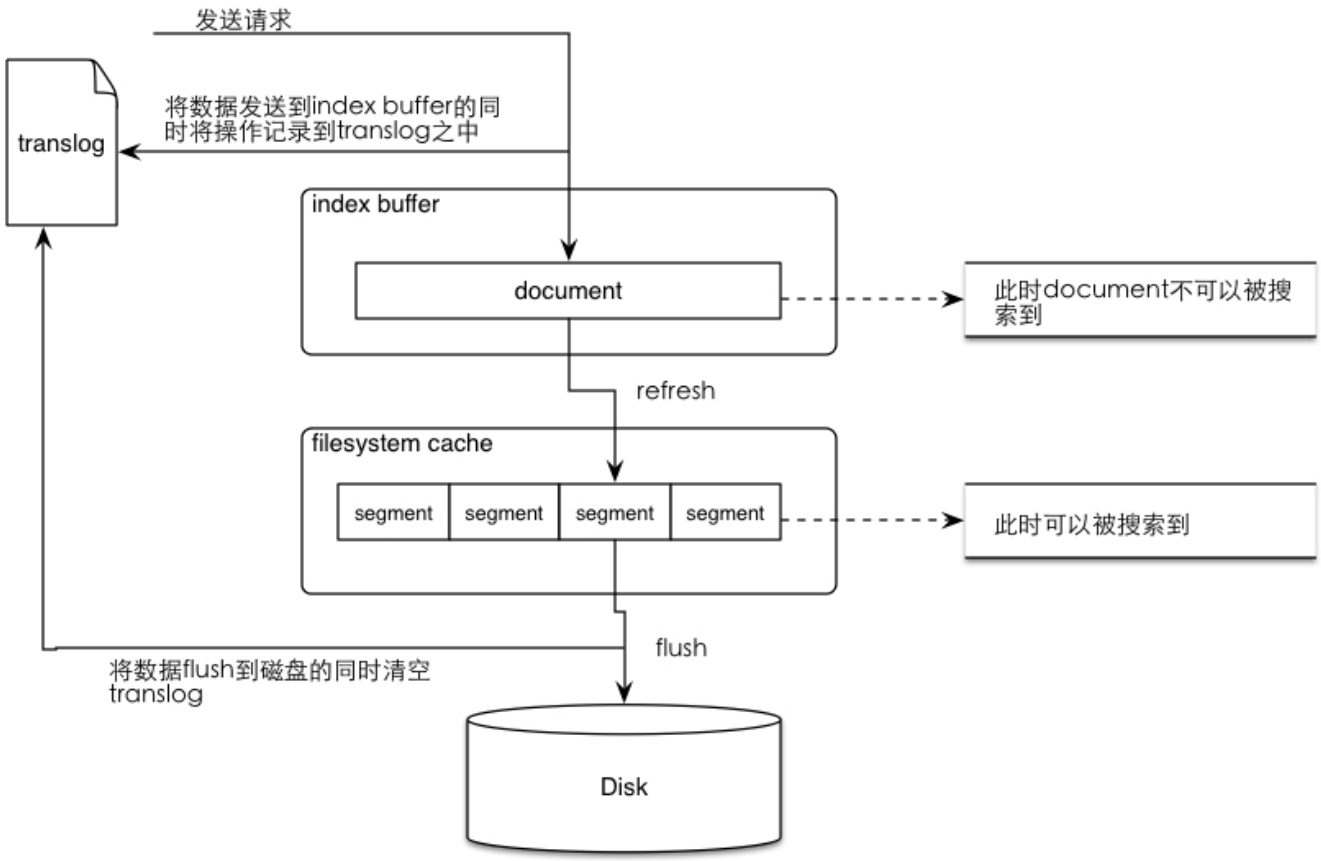
- 冷热节点(Hot & Warm Node)
    - 不同硬件配置的Data Node，用来实现Hot & Warm架构，降低集群部署的成本。通过设置节点属性来实现
- 机器学习节点(Machine Learning Node)
    - 负责跑机器学习的Job，用来异常检测

配置原则：

- 开发环境一个节点可以承担多种角色，节省服务器资源
- 生产环境中，应该设置单一的角色的节点，即dedicated node

节点类型	配置参数	默认值
候选主节点	node.master	true
数据节点	node.data	true
ingest节点	node.ingest	true
协调节点	无	每个节点默认都是协调节点
机器学习节点	node.ml	true

为什么说ES是准实时的?



Document首先写入到Indexing buffer中，当 buffer 中的数据每隔index.refresh_interval秒（或者Indexing buffer满了)缓存 refresh 到filesystem cache 中时，此时文档是可以检索到了

怎么解决ES深度分页问题?

深度分页指的是假设我们请求第 1000 页—结果从 10001 到 10010 。所有都以相同的方式工作除了每个分片不得不产生前10010个结果以外。 然后协调节点对全部 50050 个结果排序最后丢弃掉这些结果中的 50040 个结果。在分布式系统中，对结果排序的成本随分页的深度成指数上升。

解决办法就是业务上面避免。

doc values为了解决什么?

doc_values是为了解决排序和聚合问题。doc_values不适合text类型字段，对于text类型字段需要使用Fielddata

```
PUT /myindex/_mapping/doc
{
  "properties": {
      "myfield": {
      "type": "text",
      "fields":{
        "keyword":{
          "type":"keyword",
          "ignore_above":256
        }
      }
    }
  }
}
```

上面myfield是不可以聚合的，但是myfield.keyword是可以聚合的

ES中副本分片的目的是做什么?

1. 副本分片的主要目的就是为了故障转移，如果持有主分片的节点挂掉了，一个副本分片就会晋升为主分片的角色。
2. 副本分片可以服务于读请求，可以通过增加副本的数目来提升查询性能

怎样在不停机情况下，进行索引重建?

索引别名

资料

- nginx平滑的基于权重轮询算法分析
- elasticsearch中 refresh 和flush区别
- 一文带你彻底弄懂ES中的doc_values和fielddata