

# 第 I 章 SQL 关系型数据库

## 1. 索引

### 1.1. 索引是什么？

加快 SQL 查询速度的数据结构，引来的缺点（降低更新表的速度，保存索引占用空间）

### 1.2. 索引采用那些数据结构？

1. HASH 索引：底层是哈希表，存储 KV，在进行查找时，调用一次 hash 函数就可以找到相应的键值
2. B+Tree 索引：B+树，多路平衡查找树，每次查询都是从根节点出发，查找到叶子节点就可以获得所查的值

MySQL 中的数据一般是放在磁盘中的，读取数据的时候肯定会有访问磁盘的操作，定位是磁盘的存取中花费时间比较大的一块
B+Tree 是专门为磁盘 IO 设计的一种多路平衡查找树，它的高度远远小于其他数据结构，因此访问磁盘的数量极小，磁盘 IO 所花的时间少
B+树为什么比 B 树更适合？ <ol style="list-style-type: none"><li>1. B+树的内部节点不存放数据，因此其内部节点相对 B 树更小</li><li>2. B+树的查询效率更加稳定：B+树的数据都存储在叶子结点中，所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。</li><li>3. 由于 B+树的数据都存储在叶子结点中，且叶子节点形成链表，便于区间查询</li></ol>
B 树，什么时候会分裂、合并

#### 1.2.1. HASH 索引和 B+Tree 索引对比

1. HASH 索引等值查询更快，但是不支持采用 key 排序/范围查询/最左匹配原则/模糊查询
2. HASH 函数选择不好时，会发生 HASH 碰撞，导致查询效率降低
3. HASH 索引任何时候都避免不了回表查询，B+索引在覆盖查询时，可以避免回表查询
4. HASH 索引是存放在内存中的，占用内存资源太大

#### 1.2.2. 为什么底层使用 B+Tree，不使用二叉树、BST、AVL、RBT

二叉树/BST 会退化成链表、AVL 树旋转代价太高、RBT 树太高

B+Tree 与 BTree 相比有什么优势？中间节点只存放索引、只有叶子节点存放数据；叶子节点连成链表，便于范围查询

### 1.3. 索引种类

1. 普通索引：最基本的索引，没有任何限制
2. 唯一索引：列值唯一，可以有 NULL。“唯一”：加入在 name 上建立唯一索引，那么，整个表就不能有两个行 name 相同的情况
3. 组合索引，又叫联合索引
4. 全文索引：FULLTEXT，仅适用于 MYISAM 引擎的数据表。通过关键字的匹配来进行过滤查询
5. 聚簇索引、非聚簇索引；主键索引、非主键索引
6. 覆盖索引

#### 1.3.1. 聚簇索引、非聚簇索引

**[核心区别]** 聚簇索引/非聚簇索引的区别是 B+树的叶子节点存放的是数据？还是指向数据的指针(一般存放的是主键值)？

**[对比]** 聚簇索引查询速度更快

聚簇索引：索引 B+Tree 的叶子节点上存放了数据行的物理地址

非聚簇索引：非聚簇索引 B+Tree 树的叶子节点存储的不再是行的物理位置，而是主键值；索引数据时，需

要先查找到主键值，再通过二次索引查找到数据行的物理地址

面试：聚簇索引可以有多个吗？

答：不可以，聚簇索引只能有一个。

**聚簇索引**的顺序就是数据的物理存储顺序,而非**聚簇索引**的解释是:索引顺序与数据物理排列顺序无关。正式因为如此,所以一个表最多只能有一个**聚簇索引**

### 1.3.2. 主键索引、非主键索引

1. 主键索引(聚簇索引): 的叶子节点上存放的是整行数据。查询时只需要查询一张表就可以得到结果
2. 非主键索引(非聚簇索引): 叶子节点上存放的是主键的值。查询时, 需要扫描两个索引树(1)第一遍先通过普通索引定位到主键值 id=5 (2)然后第二遍再通过聚集索引定位到具体行记录, 这就是所谓的回表查询

### 1.3.3. 回表查询/覆盖索引

**[问题]** 非聚簇索引一定都会通过回表查找多次么? 答案: 不是, 原因是“覆盖索引”。

覆盖索引: 当 sql 语句的所求查询字段 (select 列) 和查询条件字段 (where 子句) 全都包含在一个索引中, 就可以直接使用索引查询而不需要回表!

**[问题]** 怎么通过覆盖索引优化回表查询?

建立联合索引, 使要查找的列都在索引中, 避免回表查询。

### 1.3.4. Index Condition PushDown 索引下推

select \* from where name like 'zhang%' and age>18

因为是 select \* , 所以一定会触发回表查询, 以下有 2 种查法

1. 查找 zhang 开头的主键, 然后回表查询所有的记录, 再过滤 age>18 的行
  2. 查找 zhang 开头的记录, 再筛选出 age>18 的记录, 再回表查询所有数据
- 优化器会选择第 2 中, 因为 2 先通过两个条件过滤会得到更少的信息, 再回表查询

### 1.3.5. 组合索引(联合索引)、最左匹配原则

**[问题]** 什么是联合索引? 为什么需要注意联合索引中的顺序?

组合索引: 可以使用多个列建立一个索引, 满足最左前缀匹配原则

## 1.4. 优索引口诀

<https://www.cnblogs.com/pdun/p/11343318.html>

全值匹配我最爱, 最左前缀要遵守  
带头大哥不能丢, 中间兄弟不能断  
索引列上少计算, 范围之后全失效  
like 百分写最右, 覆盖索引不写星  
不等空值还有 or, 索引失效要少用  
var 引号不能丢, SQL 高级也不难

1. 查询频率高的列、经常需要排序、分组、联合的字段建立索引
2. 创建索引的数目不宜过多, 过多会占用空间, 且影响表的更新速度
3. 选择唯一性索引(如学生的学号)
4. 不在索引上做运算符操作
5. 范围条件放最后: 因为范围条件后的索引都会失效
6. 字符类型要加双引号
7. or 替换为 union: A or B, 如果 A 建立了索引, B 没有建立索引, 则索引通通不走
8. like 查询要当心: like %keyword 索引失效, like keyword%索引有效
9. 不等于!=要慎用: 索引失效
10. 考虑在 where 或 order by 或 group by 涉及的列建立索引

## 2. 存储引擎 innodb/myisam

### 2.1. 区别

	InnoDB	myisam	InnoDB	myisam
事务	支持		支持事务, 可靠性要求高	不支持事务
锁级别	行锁	表锁	表更新较频繁	查询多, 插入和删除少
是否支持外键	支持			做很多 count(*) 运算
查询 插入和更新	更快	更快	两种类型最主要的差别就是 InnoDB 支持事务处理与外键和行级锁. 而 MyISAM 不支持.	
全文索引		支持		

1. innodb 支持事务、外键、行锁(默认)/表锁, 不支持全文所以

2. innodb 必须有主键, 没有显示指定主键, mysql 会默认创建主键\_rowid; 而 myisam 可以没有主键

3. innodb 是主键索引/聚集索引, myisam 是非主键索引/非聚集索引。

也就是说: InnoDB 的 B+树主键索引的叶子节点就是数据文件, 辅助索引的叶子节点是主键的值; 而 MyISAM 的 B+树叶子节点都是数据文件的地址指针

4. 存储文件

innodb: frm 表结构文件、ibd 数据文件(包括索引/数据)

Myisam: frm 表结构文件、MYD 数据文件、MYI 索引文件

### 2.2. 如何选择?

innodb: 支持事务、行锁、读写频繁

myisam: 不支持事务、具有大量的读操作、写操作很少

### 2.3. 扩展问题/知识点

#### 2.3.1. InnoDB 为什么推荐使用自增 ID 作为主键, 不用 UUID?

B+Tree 底层结构: 在插入的时候, (1)自增 ID 可以保证每次插入时 B+索引是从右边扩展的(2)UUID 是随机生成的, 不一定 key 值就比之前的数大, 会导致 B+树和频繁合并和分裂。

另外, UUID 占用 16 个字节, 占用内存较大

#### 2.3.2. innodb 无索引 or 索引失效时, 行锁会升级为表锁

## 3. 三范式

腾讯比较喜欢考这个题, 要重视!

第一范式: 列不可拆分

第二范式: 非主键列完全依赖于主键, 而不依赖于主键的一部分

第三范式: 非主键列只依赖于主键, 不依赖于其他非主键

## 4. 事务

### 4.1. 前言

多用户、多程序、多线程, 保证数据一致性, 引出事务

### 4.2. 特性: ACID

原子性: 最小单元, 整个事务的所有操作要么做, 要么都不做

一致性: 从一种一致性状态转换为另一种一致性状态, 事务开始/结束都保证完整性

隔离性: 并发执行的各个事务之间不相互干扰

持久性: 事务一旦提交, 结果将永久保存在数据库中

### 4.3. 事务并发问题

脏读(读取未提交数据): B 修改某个数据后, 未提交, 被 A 读到; 之后 B 回滚修改数据操作, A 之前读到的数据就是脏数据

**不可重复读(在一个事务中前后读取的数据不一致)**: A 读取同一个数据经历的时间很长, 第一次读时, 该数据为 Val1, 之后, 该数据被 B 修改, 之后 A 再去读该数据, 结果为 Val2, 这就叫做不可重复读

**幻读(前后多次读取, 数据总量不一致)**: 与不可重复读类似, 都是在一个事务中, 两次读取结果不一样。区别在于幻读是在一个事务中读取到数据的条数不一致, 如: 事务 A 在执行读取操作, 需要两次统计数据的总量, 前一次查询数据总量后, 此时事务 B 执行了新增数据的操作并提交后, 这个时候事务 A 读取的数据总量和之前统计的不一样, 就像产生了幻觉一样, 平白无故的多了几条数据, 称为幻读

#### 4.4. 事务的四种隔离级别

**读未提交**: 所有事务都能够读取其他事务未提交的数据, 会导致脏读、不可重复读、幻读

**读已提交**: 所有事务只能读取其他事务已经提交的数据, 可以解决脏读! 但是会出现在一个事务中前后读取内容不一致的问题, 即不可重复读、幻读

**可重复读**: 在一个事务中, 不允许 Update 操作, 允许 Add 操作, 因此能保证在一个事务中读取数据内容是一致的(能解决不可重复读), 但是不能保证读取到数据条目数一致(会发生幻读)

**可串行化**: 所有的事务都顺序串行执行, 不存在冲突

#### 4.5. 扩展问题/知识点

**4.5.1. innodb 默认是可重复读级别(次高级, 不是最高级)**

#### 4.6. 事务的原理

##### 4.6.1. 事务日志: redo/undo (仅仅 innodb)

redo log 重做日志: 保证事务持久性	undo log 回滚日志: 保证事务原子性
在事务提交之前, 先写入到 redo log 中, 如果某时刻系统宕机, 重启后, 可以通过 redo log 恢复之前的数据	数据更新时, 会写入 undo log (该操作和数据更新执行操作相反, 即如果是插入数据, 则 undo log 是删除数据)。当事务失败或回滚时, 可以通过 undo log 回滚到更新前的状态
<b>数据库宕机恢复过程</b>	
先从 redo log 中把未落盘的脏页数据恢复回来, 重新写入磁盘, 保证用户数据不丢失。	
把未提交的事务, 根据 undo log 执行回滚操作, 恢复事务	

##### 4.6.2. redo 日志写入时机

<b>redolog 写入真实的物理磁盘</b>
① 先写入 redo log buffer (redo log 缓存)
② 之后调用 fsync, 刷新写入 redo log 物理磁盘
它的写入是可进行参数配置的。

##### 4.6.3. 二进制日志: binlog (所有存储引擎都有)

binlog 与 redo log 类似, 它也是在数据提交前, 保存更新日志, 但是二者还是有本质的区别

	redo log	binlog
场景	crash-recovery 宕机恢复	point-time-recovery 恢复某个时间点
	只有 innodb 支持事务的存储引擎有	所有 SQL 都支持
写入时机	写入时机根据配置文件	事务提交时写入, 记录数据库更新操作
		实现主从复制(下面会介绍)

##### 4.6.4. 分布式之两/三阶段提交

背景	在分布式环境中, 会有多个节点, 常出现这种场景, 多副本, 更新一个数据时, 要使得所有节点上的数据全部更新, 才认为该更新成功。但是某个节点只能知道自己节点上的事务是否执行成功, 不能知道其他节点的事务是否更新。
因此	引出了一个中间人(即协调者), 它用于协调事务执行过程

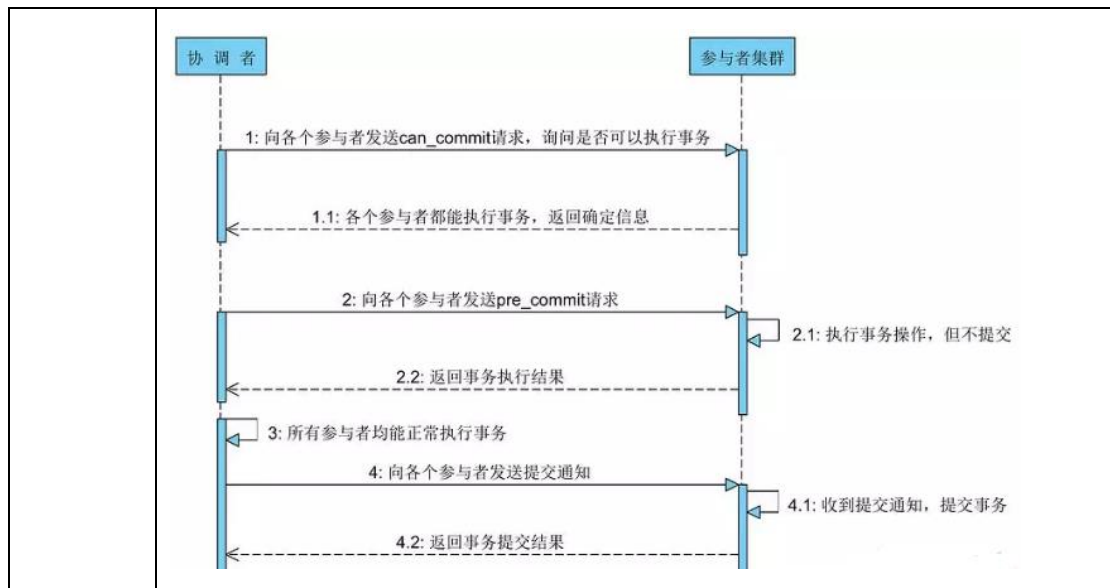
##### 4.6.4.1. 二阶段提交

<b>二阶</b>	① 协调者向所有节点上的参与者发送“开始事务”的信号
-----------	----------------------------

段提交运作过程	<p>② <b>（阶段 1：投票）</b> 所有参与者执行事务，执行成功后，会应答结果给协调者（事务成功/事务失败）</p> <p>③ <b>（阶段 2：执行）</b> 协调者收到信号后，会根据结果，再次向参与者发送通知：全部成功，就发送“提交事务”信号；有一个失败，就发送回滚事务信号，放弃事务</p> <pre>sequenceDiagram     participant C as 协调者     participant P as 参与者集群     Note over C: 1: 询问各参与者事务是否可以正常执行     C-&gt;&gt;P: 1: 询问各参与者事务是否可以正常执行     Note over P: 1.1: 执行事务，但不提交     P--&gt;&gt;C: 1.2: 所有参与者均能成功执行事务     Note over C: 2: 向所有参与者发起事务提交通知     C-&gt;&gt;P: 2: 向所有参与者发起事务提交通知     Note over P: 2.1: 提交事务，并释放资源     P--&gt;&gt;C: 2.2: 反馈事务提交结果</pre>
<b>两阶段提交的缺点</b>	
单点故障	只有一个协调者，它宕机后，将会造成单点故障
阻塞	协调者故障，所有的参与者将阻塞等待它通知，此时会一直持有事务资源
不一致性	协调者应答“提交事务”时，可能由于网络原因，导致某个节点没收到，该节点不会提交，造成数据不一致

4.6.4.2. 三阶段提交

与两阶段提交的改进点	
超时机制	加入超时，当超时机制，即参与者/协调者接收信号时，都设置超时，防止一直阻塞
三阶段	将两阶段种的第一个阶段（投票阶段），拆分成 2 个阶段：预投票、准备
	<p>① <b>（阶段 1 预投票）</b>：协调者向参与者询问，是不是都能开始事务（如果全部都能开始事务，则进入下一个阶段；如果有一个不能开始事务，则放弃事务）。</p> <p>② <b>（阶段 2）</b> 假如，协调者向参与者发送“开始执行事务”的信号</p> <p>③ 对方执行事务，并返回应答该协调者</p> <p>④ <b>（阶段 3）</b> 协调者收到信号后，会根据结果，再次向参与者发送通知：全部成功，就发送“提交事务”信号；有一个失败，就发送回滚事务信号，放弃事务</p>



## 5. 锁

当数据库有并发事务时, 可能会产生数据不一致, 锁可以保证访问次序。

### 5.1. 共享锁(读锁), 独占锁(写锁)

- (1) 共享锁(读锁): 可以被多个事务同时读, 但是加了读锁, 不允许加写锁
- (2) 独占锁(写锁): 加了写锁, 不允许加读锁 or 写锁

### 5.2. 乐观锁, 悲观锁

(1) 乐观锁: MySQL 最经常使用的乐观锁是进行[版本控制], 也就是在数据库表中增加一列, 记为 **version**。当我们将数据读出时, 将版本号一并读出; 当数据进行更新时, 会对这个版本号进行加 1; 当我们提交数据时, 会判断数据库表中当前的 **version** 列值和当时读出的 **version** 是否相同, 若相同说明没有进行更新的操作, 不然, 则取消这次的操作。

- (2) 悲观锁: 只允许一个锁进入

### 5.3. 表锁、行锁

### 5.4. 多版本并发控制 MVCC \*\*

<https://www.cnblogs.com/shujiying/p/11347632.html>

## 6. SQL 优化

### 6.1. 数据插入优化

1.插入前, 禁用索引

2.修改事务的提交方式 (变多次提交为一次提交)

```
insert into test values(1,2);
```

```
insert into test values(1,3);
```

```
insert into test values(1,4);
```

```
//合并多条为一条
```

```
insert into test values(1,2),(1,3),(1,4)
```

3.插入后, 不禁用索引

### 6.2. 单机优化

#### 1. 慢查询

- (1) 慢查询配置: `slow_query_log`、`slow_query_log_file`、`long_query_time`

(2) 慢查询日志分析工具 mysqldumpslow: 捕获前 10 条查询较慢的 mysqldumpslow -s at -t 5 xxx.log

2. SQL 语句优化

消除子查询，改为关联查询

3. 没建立索引，就建立索引

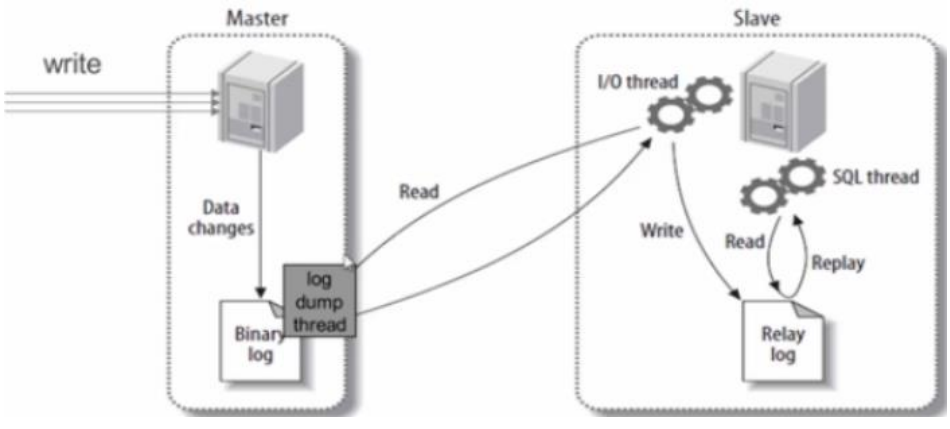
4. 对于已经建立的索引，可能存在索引失效

explain 查看 SQL 语句的执行计划: possible\_key,key,key\_len,using 等字段，分别表示可能使用的索引，实际使用的索引，使用索引的总字节数，using index(覆盖索引)/where(回表)/filesort(order by)/temporary(group by 临时表)

Using 字段含义	
where	回表查询
index	覆盖索引，直接通过索引就可以获取到所有要查询的数据，无需回表查询
filesort	并不是说通过磁盘文件进行排序，而只是告诉我们进行了一个排序操作而已(只有在 order by 数据列的时候才可能会出现 using filesort) (1)修改逻辑，不在 mysql 中使用 order by 而是在应用中自己进行排序 (2)使用 mysql 索引，将待排序的内容放到索引中，直接利用索引的排序
temporary	group by 临时表

6.3. 集群优化

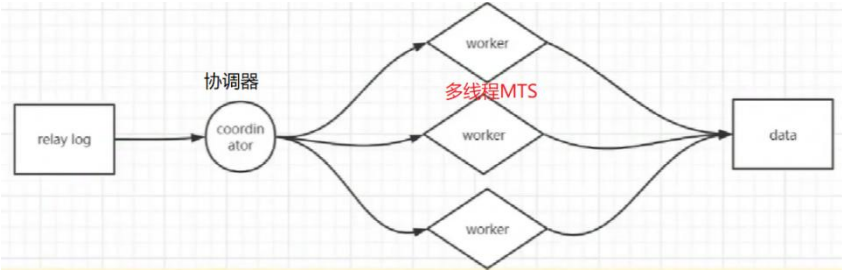
1. SQL/Redis 主从复制

定义	将一台 Redis 服务器的数据，复制到其他的 Redis 服务器，前者后者分别称之为 master/slaver，数据的复制是单向的，只能由主节点到从节点
作用	数据冗余：从节点保存了和主节点一样的数据 故障恢复：主节点出现问题时，从节点可以提供服务，实现故障恢复 负载均衡：在主从复制的基础上，配合读写分离，主节点提供写服务，从节点提供读服务 高可用基石：哨兵、集群实现高可用
主从复制原理	<div></div> <p>上面已经知道，数据库的 DML 操作，都会保存在 binlog 中；如何将主节点的 binlog 同步到各个从节点上？</p> <p>答：</p> <p>①拉去日志：从数据库开启 IO 线程，主动拉取 binlog，保存为中继日志 Relay log</p> <p>②日志回放：从数据库开启 SQL 线程，将中继日志在从服务器上重新执行（执行完成后，主从数据库数据一致）。</p>

重点：主从复制中延迟问题

定义	从服务器的两个线程执行速度不一致，可能会造成延迟问题。 因为：IO 线程从主服务器读取日志速度很快（顺序读），而 SQL 线程重放 SQL 速度慢，
----	---



	这就会造成从服务器同步数据远远落后于主服务器，导致从服务器数据落后于主服务器（主从数据库长期处于不一致的状态），这种现象就是延迟更新。
主从复制中延时产生的原因	1. 备库性能 << 主库性能 2. 主库经常会开多个线程去写，从库只有一个线程在工作，导致从库效率<<主库效率 3. 根本原因：主库写 Binlog、从库 IO 线程读 Binlog 都是顺序操作，执行速度很快； <b>但是从库 SQL 线程重放操作是随机操作</b> ，很慢 4. 主库一直在执行大事务（每个事务执行 10min），而 Binlog 的写入必须要等待事务执行完成之后，才会传入备库，那么此时在开始执行时就已经延迟了 10min
解决方案：  从服务器的数据重放过程采用多线程	 <p>MTS：要遵循两个规则，即对于 2 种情况，应该必须分发到同一个 worker 线程  <b>同一个事务</b>中的 MDL，必须分发到同一个 worker 线程  <b>MDL 同一行</b>的多个事务，必须分发到同一个 worker 线程</p>

## 2. 读写分离

	主库只进行更新写操作，从库进行查询读操作
主库	增删改更新操作，即：更新操作，一直在主服务器
从库	查询操作，即：查询操作，一直在从服务器

## 3. 分库分表：水平/垂直

场景	<p>随着公司业务的发展，数据库中的数据量猛增，访问性能也变慢了，优化迫在眉睫。当数据达到 100W 或 100G 后，由于查询的维度较多，即使添加从库、优化索引，很多操作仍然性能下降很大。</p> <p>分库分表是为了解决数据量过大导致数据库性能降低的问题！将原来独立的数据库、表，拆分成多个数据库、表，使得单个数据库、表的数据量变小，从而达到性能优化的目的。</p>
垂直分表	将表按照 <b>属性列</b> 划分成多个表
水平分表	数据量行数过大时，按照行分成多个表
垂直分库	按照 <b>业务</b> 将表分不到不同的数据库，每个库可以放在不同的服务器
水平分库	

## 6.4. 缓存 redis

防止每次请求都发到数据库上，使用缓存，降低连接数据库操作、数据库处理操作次数，提高数据库性能
--

## 7. 分页查询：limit

### 7.1. 超大分页怎么处理？

答：超大的分页一般从两个方向上来解决。

① 数据库层面,这也是我们主要集中关注的(虽然收效没那么大), 类似于 `select * from table where age > 20 limit 1000000,10` 这种查询其实也是有可以优化的余地的. 这条语句需要 load 1000000 数据然后基本上全部丢弃,只取 10 条当然比较慢。我们可以修改为 `select * from table where id in (select id from table where age > 20 limit 1000000,10)`。这样虽然也 load 了一百万的数据,但是由于**索引覆盖**,要查询的所有字段都在索引



引中,所以速度会很快.

② 同时如果 ID 从 0 开始且连续递增的话, 我们还可以 `select * from table where id > 1000000 limit 10`,效率也是不错的

## 7.2. 结论: 优化的可能性有许多种,但核心思想是减少 load 数据量

分页查询: 如何快速定位起始位置 offset、减少无用数据缓存

从需求的角度减少这种请求....主要是不做类似的需求(直接跳转到几百万页之后的具体某一页.只允许逐页查看或者按照给定的路线走,这样可预测,可缓存)以及防止 ID 泄漏且连续被人恶意攻击.

解决超大分页,其实主要是靠缓存,可预测性的提前查到内容,缓存至 redis 等 k-v 数据库中,直接返回即可. 在阿里巴巴《Java 开发手册》中,对超大分页的解决办法是类似于上面提到的第一种.

## 8. 关联查询 join (待整理) \*\*

### 8.1. 内连接; 左连接/右连接; 自身连接

自身连接	<code>select FIRST.Cno,SECOND.Cpon</code> #查询每门课的先修课的先修课 <code>form course FIRST, course SECOND</code> #重命名 <code>where FIRST.Cpon=SECOND.Cno</code>
左连接/ 左外连接	<code>from A left join B on (连接条件)</code> #以A的行为主行,B没有的补NULL <code>from A right join B on (连接条件)</code> #以B的行为主行,A没有的补NULL
内连接	<code>from A inner join B on (连接条件)</code> #A和B的交集

## 9. 其他常见问题

### 9.1. SQL 常用语法&&执行顺序

	<code>select</code> [ALL DISTANCE] <目标列表达式> <code>from</code> <表明或视图名> <code>where</code> <条件表达式> <code>group by</code> <列名> <code>having</code> <条件表达式> <code>order by</code> <列名> ASC DESC		
谓词	IN ; LIKE	IS NULL ; IS NOT NULL	AND ;OR
	<code>Order by</code> <属性列> #按照<属性列>排序		
	<code>Group by</code> <属性列>	<code>Having</code> 条件表达式	#<属性列>值相等的分在一组
书写顺序: <code>select...from...where...group by...having...order by..</code> 执行顺序: <code>from...where...group by...having...select...order by...</code>			

### 9.2. 什么是存储过程? 有哪些优缺点?

定义: 多个 SQL 语句的集合, 就像是函数, 但是它没返回值

优点: 一次连接, 执行存储过程中所有的 SQL 语句, 效率高

1. 只在创建时编译一次, 之后不编译; 可以重复使用, 提高开发效率
2. 安全性高: 可以设定某个用户是否具有某个存储过程的使用权限

```
create procedure insert_student_process(name varchar(50),age int,out_id ing)
```

```
//创建存储过程
```

```
begin:
```

```
insert into student value(null,name,age)
```

```
select max(stuId) into id from student
```

```
end;
```

```
call insert_student_process('gjw',26,@id); //调用存储过程
```

```
select @id;
```

### 9.3. 触发器：触发条件 条件满足

触发器：需要有触发条件，当条件满足以后做什么操作

例如 1：校内网，开心网，facebook，你发一个日志，自动通知好友，其实

就是增加日志时做的一个后触发，再向"通知表"写入条目。--->触发器的效率高

### 9.4. [select \*] 和 [select 全部字段] 2 种写法有什么优缺点

	select *	select 全部字段
是否需要解析数据字典	是	否
结果输出顺序	于建表列顺序相同	按指定字段顺序
表字段改名	无需修改	需要修改
是否可以建立索引优化	否	是
可读性	低	高

### 9.5. varchar、char 区别

(1) 定长/变长：是否由实际存储内容决定

char 是定长字段，假如申请了 char(10)的空间,那么无论实际存储多少内容.该 字段都占用 10 个字符

varchar 是变长的,也就是说申请的只是最大长度,占用 的空间为实际字符长度+1,最后一个字符存储使用了多长的空间.

(2) char 查询效率更快

## 10. 面试：海量数据

（面试）海量数据存入表：将 10W 数据导入数据库，要求实时查看导入进度

将 10W 数据导入数据库：数据分批处理——>多线程	
1.	将 10W 数据分成 10 份，给 10 个线程分别处理（每个线程处理 1W 个数据）
2.	分批次插入：将 1W 条数据，分块（每块 100 条数据）；一批一批插入
实时查看导入进度——>触发器	
	创建触发器，每次插入时，对全局变量加锁，更新进度

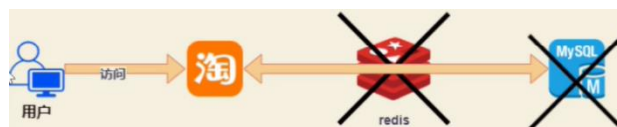
## 第 II 章 Redis 缓存

### 0. 缓存雪崩/穿透/击穿

#### 0.1. 缓存雪崩

大量的 Redis 缓存在某一时刻失效（就像雪崩来了一样），导致大量请求全部打到数据库上，导致数据库压力过大挂掉。

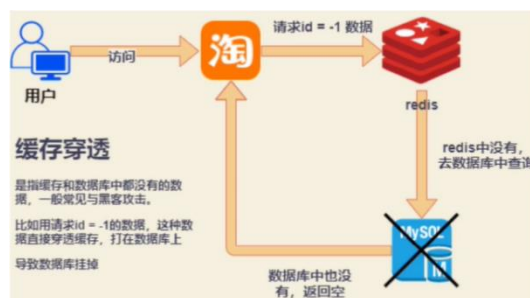
解决方案：熔断、降级、限流。（即：“熔断”，一旦发现当前服务的请求失败率达到预设的值，就拒绝随后该服务的所有请求。当经过一段时间后，会放行该服务的一部分请求，再次统计它的请求失败率。如果此时请求失败率符合预设值，则完全打开限流开关；如果请求失败率仍然很高，那么继续拒绝该服务的所有请求，这就是所谓的“限流”。而向那些被拒绝的请求直接返回一个预设结果，被称为“降级”）



#### 0.2. 缓存穿透

恶意的用户，频繁访问数据库中不存在的数据，导致 Redis 查不到该数据就去数据库中查，导致缓存被穿透！

解决方案：对查询内容的合法性校验；布隆过滤器[过滤掉数据库中不存在的 key]

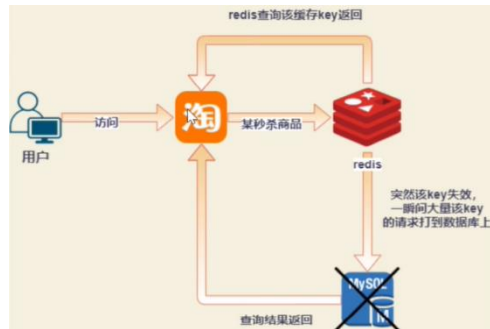


#### 0.3. 缓存击穿

双 11 秒杀某个商品的时候，某个商品的缓存 key 突然失效，导致一瞬间大量查询该 key 的请求打到数

数据库上，导致数据库挂掉。

解决方案：分布式锁（先对该商品加上锁，使得其他请求不能进来，然后将 key 写入缓存中，其他后来的请求就能从缓存中找到该 key）



## 1. 底层数据结构

### 1.1. 压缩列表 ziplist

压缩列表使用场景：含有少数的键，且键是“短整型”、“短字符串”	
优点：节省内存，实现简单，是连续内存块的顺序存储（有点像变长数组，它通过长度划分每个节点）	
每个节点构成	1. 前一个节点的长度 pre_len 2. 当前节点的长度、类型 3. 当前节点的数据内容
连锁更新	当插入和删除元素时，可能会导致连锁更新。 ①（全 small）原 ziplist 节点都是长度小于 256：当在 idx 插入大于 256 的节点时，idx+1 后面的节点 e1 的成员 pre_len 无法保存前一个节点的长度，因此，要重分配内存。这样 e1 内存就扩增了，因为是顺序存储，所以 e2、e3 后面的元素都要向后移动（更新） ②（big1、small、big2）：当删除 small 时，将会引起 big2 后面的节点连锁更新
场景	列表键、哈希键

### 1.2. 跳跃表

时间	最好 $O(\lg N)$ ，最差 $O(N)$ 性能可以和 AVL 树媲美，且实现简单
	是一个多层次的链表 每层节点的 next 跨度大小都不同，从上到下依次减小
场景	有序集合键

### 1.3. 哈希表

渐进式 rehash 过程	rehash（重新散列）：哈希键数量增多/减少，会造成 rehash，保证哈希表负载因子维持在一个合理的范围。 ① 分配新的哈希内存空间 hash[1] ② 将原哈希 hash[0]，通过 rehash（重新计算哈希值），散列/迁移到新哈希 hash[1]中 rehash 过程不是一次执行的，而是分批次、渐进式执行的。
场景	哈希键

## 2. Redis 五种数据结构

字符串键 string	字符串类型的 value 最大能容纳 512M
列表键	ziplist、linklist
哈希键	ziplist、hash-table
有序集合键	ziplist、跳跃表

位图键	bitmap
-----	--------

### 3. 对象的内存回收机制/对象共享

就是采用了引用计数，维护对象的声明周期
① 当引用计数减为零时，释放该对象
② 当引用计数大于 1 时，说明此时该对象至少被别人持有 2 次，即：共享

### 4. 过期时间\过期删除机制

过期时间	当一个元素过期时，将会被删除，即：该 key-value 变为无效
过期删除机制	<p>① 惰性删除：只有访问时，采取查看该元素是否过期，过期才真正的删除。</p> <p>优点：不需要频繁的删除，消耗 CPU 资源少</p> <p>缺点：占用内存</p> <p>② 定时删除：开启定时器，定时器到时间时，删除过期数据</p> <p>优点缺点和方案①完全相反</p>

### 5. 持久化 RDB/AOF

面试题法：Redis 持久化机制介绍，各自优缺点。	
RDB	在指定时间间隔内，将内存中的数据和操作，通过【快照】的方式保存到 RDB 文件，可以配置在 N 秒内进行一次快照。
触发条件	<p>1.手动触发：save 命令和 bgsave 命令都可以生成 RDB 文件</p> <p>save 会阻塞 Redis 服务器进程，直到 RDB 文件创建完毕为止</p> <p>bgsave 会创建一个子进程，更新 RDB 文件，父进程可以继续处理请求</p> <p>2.自动触发：redis.conf 配置文件 save m n，表示当 m 秒内发生 n 次变化时，触发 bgsave</p>
优点	加载数据快，直接读取 RDB 文件内容加载到内存
缺点	1.不安全，会丢失时间间隔内的数据；2.每次保存 RDB 文件时，都需要 fork 一个子进程来持久化，性能开销较大；3.加载 RDB 文件期间，会一直处于阻塞状态，直到载入工作完成位置
AOF	将每次更新写操作，都写到 AOF 文件，重启时，只需要从头到尾执行一次 AOF 中的指令，可以恢复数据
优点	比较安全，可以配置一次更新写，就写 AOF 文件
缺点	一直更新写操作，会使 AOF 文件激增，极端场景下，会对硬盘空间造成压力
Rewrite	将对同一个 key 的操作，合并成一个语句，写入 AOF 文件
恢复顺序	
	当有 AOF 时，将会先执行 AOF，再执行 RDB

### 6. 高可用

#### 6.1. 高可用介绍

高可用：当一台服务器宕机停止服务后，对业务及用户毫无影响
主备方式：正常情况下，主机提供服务，并将数据同步到备份机器，当主机宕机后，备机立即开始服务
主从方式：一主多从方式，主从之间数据同步。Master 宕机后，通过选举投票方式选择出新的 master，继续提供服务。主从方式主要实现读写分离，提高并发性，通过协调器将请求分发到主从节点上。
主从切换技术：当 master 服务器宕机后，slave 切换为 master

#### 6.2. 哨兵

原理	哨兵将会监控集群中所有的节点；一般为了防止单哨兵节点故障，将配置多个哨兵协同合作。
----	---

切换过程	1.（主观下线）哨兵 A 检测到主节点下线后，将不会立即切换主节点，而是认为它客观下线 2.（询问）哨兵 A 会询问监听该主节点的其他哨兵，收集汇总信息，当有足够多的主观下线信息时，判断是否为客观下线 3.（选取领头哨兵）当有一个哨兵判断为客观下线后，将会选举出领头哨兵，由它进行切换主节点操作（即：故障转移操作） 4. 会选择一个“好”的主从节点作为新的主节点，进行切换
缺点	运维复杂；哨兵选举期间，不能对外提供服务（因此如果 Master 宕机后，不支持并发）

### 6.3. 集群

	一个集群通常有多个服务器节点组成。
	最开始时，各个服务器节点是相互独立的；之后，将各个节点连接起来

#### 6.3.1. 槽指派

	整个数据库，被划分为 N 个槽，每个槽记录了一个元组（键值对）
	当所有的槽都被分配/指派给节点处理后，Redis 服务器才会进入上线状态，此时才可以接收和处理客户端发来的数据请求

#### 6.3.2. 复制和故障转移

集群中主节点/从节点和主从复制中的主节点/从节点不是一个概念	
主节点	当前集群中已经存在的节点
从节点	当前集群中新加入的节点（新加入的节点将会从原来的主节点中选择一个，复制它的槽）
如果某个主节点故障了，因为从节点已经复制了它的槽，所以该从节点将会升级为主节点，继续服务。	

## 7. 其他特性

### 7.1. 发布订阅模式

客户端订阅服务端的频道；当服务端向该频道中发送消息时，频道中所有的客户端都会收到该消息，执行相应的动作
---

还有其他的特性，不一一介绍了，如：事务 ACID \ 慢查询日志

### 7.2. 事务

#### 7.2.1. MySQL 和 Redis 的事务的区别

结论：二者在事务上有本质的区别	
Redis 事务原理	使用乐观锁，只负责监听 key 有没有被改动。即： 采用 watch 监听某个 key；在执行命令时，检查该被监视的 key 是否已经被修改；如果该 key 时被改动，那么事务将会被打断
	与 SQL 事务最大的区别，Redis 不支持事务回滚，即使事务在执行过程中出错，也不会回滚，将会一直执行下去，直到事务结束。 因为，开发者认为，事务执行失败，很少会在实际生产环境中出现。

### 7.3. 索引

Redis 没有实现索引，如果需要索引，需要用户自己设置并实现之
----------------------------------

## 8. Redis 面试题汇总

### 8.1. 开发中如何使用缓存（伪代码）

先查询缓存，①如果命中，就返回；②不命中，就查询数据库
-----------------------------

### 8.2. 怎么保证数据库和缓存一致性问题 \*\*

<https://blog.csdn.net/diweikang/article/details/94406186>



8.3. Redis 是单线程还是多线程？为什么采用单线程，效率却如此之高？

Redis 是单线程还是多线程要从两个方面谈起
①服务器接收客户端发来的请求命令是单线程的，它采用了 IO 多路复用，与 libevent 事件驱动库一样，它是一种反应堆模式 Reactor，服务器单线程同时监听多个事件的到来，当事件到来后，会将就绪事件挂到激活队列中
②对于每一个就绪队列中任务的处理是多线程的，这也是 Reactor 模式高性能的体现，Reactor 会事先建立一个线程池，线程池中的线程是消费者，当有任务到来时，线程将会获取任务执行它

8.4. 为什么 redis 不能代替 mysql 进行数据存储

MYSQL 是关系型数据库，大部分数据是存储在磁盘上的，以二维表格存储
Redis 是非关系型数据库，存放在内存中，以 KV 存储
一般，当并发量大时，且读很多时，采用 Redis 降低读压力，Redis 一般存放热点数据

9. Redis 实现秒杀系统设计方案

1.1. 秒杀的时候，不同地域的客户端到达服务器的时间不同，怎么保证公平性？

--	--

10. Redis 使用场景

10.1. String 字符串对象

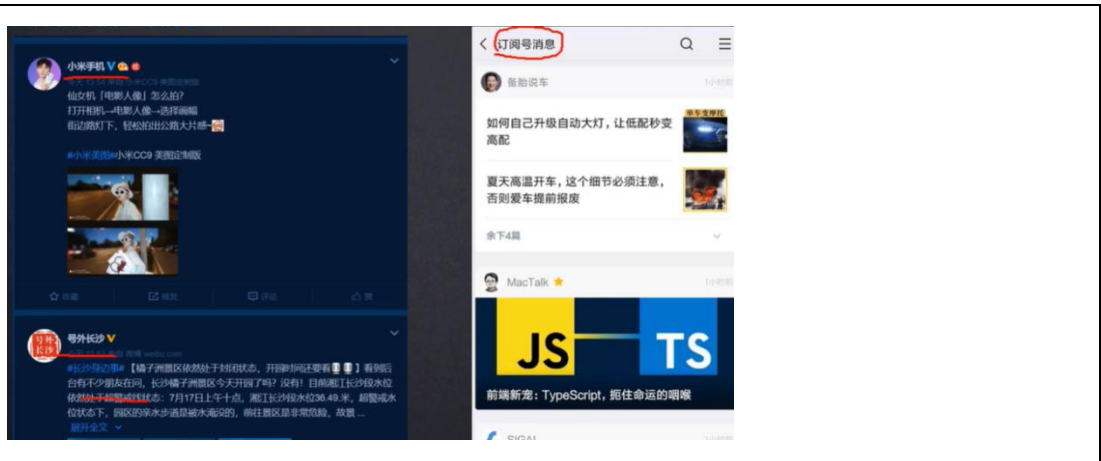

场景：统计网页/贴吧/文章阅读/浏览次数
Incr article:readcnt:1001     对文章 1001 的读次数+1

10.2. Hash 之淘宝商城购物车

<ul style="list-style-type: none"><li>• 电商购物车</li><li>1) 以用户id为key</li><li>2) 商品id为field</li><li>3) 商品数量为value</li></ul>	
--	--

10.3. List 之微博公众号消息流

订阅了某个公众号，当该公众号发文章后，会推送给你！
特点：消息的发送是有一个时间线的



LPAUSH msg:{小明微信号 ID} 10018	发消息
LPAUSH msg:{小明微信号 ID} 10011	发消息
LRANGE msg:{小明微信号 ID} 0 -1	查看最新的消息流列表

#### 10.4. Bit 位之日活量

场景：统计 2020/10/03，登录用户数。现在系统有千万级活跃用户，如何实现日活统计，为了增强用户粘性，要上线一个连续打卡发放积分的功能，怎么实现连续打卡用户统计？
将 20201003Login 作为 key，offset 作为每个用户，value 0/1 表示都否登录，即：
20201003Login Data     1   0   0   0   1   1   0 Offset   0   1   2   3   4   5   ... .. Setbit 20201003Login 0 1 Setbit 20201003Login 4 5 统计日活: bitcount 20201003Login 0 -1 统计连续几日登录量: 将 20201003Login, 20201004Login,..., 20201007Login 相与，之后再统计 1 的总个数

#### 10.5. Set 之微信抽奖小程序、微博点赞列表、微博关注模型

无序不重复，放相同的元素，将会被去重（每个人点击多次抽奖，将会被去重，只视为一次抽奖） <b>场景 1：微信抽奖小程序</b> 用户点击抽奖按钮后，将加入 set: SADD activity:10086 {用户 ID} 查看参与抽奖的所有用户: SMEMBERS activity:10086 随机抽取 count 名中奖者: SRANDMEMBER activity:10086 2 SPOP activity:10086 2	
<b>场景 2：微博点赞</b> 点赞: SADD like:{消息 ID} {用户 ID} 取消点赞: SREM like:{消息 ID} {用户 ID} 检查用户是否点过赞: SISMEMBER like:{消息 ID} {用户 ID} 获取点赞的用户列表: SMEMBERS like:{消息 ID} {用户 ID} 获取点赞用户总数: SCARD like:{消息 ID}	
<b>场景 3：微博关注模型</b> James --> {A,B,C} Kobe-->{A,C,D,R} James、Kobe 是两个集合 Set SINTER James Kobe 取出 James 和 Kobe 共同关注的人（交集） SISMEMBER James Jordan 判断 James 集合中是否有 Jordan SDIFF James Kobe James 可能认识的人	

#### 场景 4: Set 实现电商商品类型的筛选

SADD brand:huawei P30

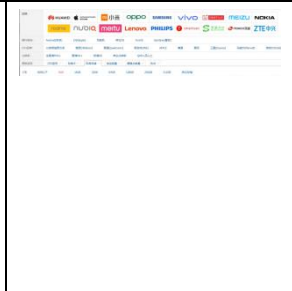
SADD brand:xiaomi 6X

SADD os:android P30 6X

SADD cpu:brand:intel P30 6X

SADD ram:8G P30 6X

SINTER os:android cpu:brand:intel ram:8G -->{P30, 6X}



#### 10.6. ZSet 有序集合之微博热点排行榜

2. 点击新闻 ZINCRBY hotNews:20190722 1 双宋离婚

3. 展示当日排行前十 ZREVRANGE hotNews:20190722 0 10 WITHSCORES

4. 七日搜索榜单计算

ZUNIONSTORE hotNews:20190716-20190722 7 hotNews:20190716

hotNews:20190717 hotNews:20190718 ... hotNews:20190722

5. 展示七日排行前十

ZREVRANGE hotNews:20190722 0 10 WITHSCORES



#### Redis 面试总结

<https://www.bilibili.com/read/cv4042105/>