数据库相关知识

SQL基础

NOSQL和SQL的区别

ACID和BASE

- ACID 即原子性，一致性，隔离性和持续性。
- BASE 即基本可用，软状态和最终一致性。
- 从实用的角度出发，我们需要考虑对于面对的应用场景，ACID 是否是必须的。比如银行应用就必须保证 ACID，否则一笔钱可能被使用两次；又比如社交软件不必保证 ACID，因为一条状态的更新对于所有用户读取先后时间有数秒不同并不影响使用。

扩展性

- NoSQL数据之间无关系，这样就非常容易扩展，也无形之间，在架构的层面上带来了可扩展的能力。比如 redis 自带主从复制模式、哨兵模式、切片集群模式。
- 相反关系型数据库的数据之间存在关联性，水平扩展较难 ，需要解决跨服务器 JOIN，分布式事务等问题。

三大范式

- 1NF：要求数据库表的每一列都是不可分割的原子数据项。
- 2NF：在1NF的基础上，非码属性必须完全依赖于候选码（在1NF基础上消除非主属性对主码的部分函数依赖）
    - 需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。
- 3NF：在2NF基础上，任何非主属性 (opens new window)不依赖于其它非主属性（在2NF基础上消除传递依赖）
    - 第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

MySQL连表查询

内连接

- 返回两个表中有匹配关系的行

左外连接

- 左外连接返回左表中的所有行，即使在右表中没有匹配的行。未匹配的右表列会包含NULL。

右外连接

- 右外连接返回右表中的所有行，即使左表中没有匹配的行。未匹配的左表列会包含NULL。

全外连接（MySQL不支持）

- 全外连接返回两个表中所有行，包括非匹配行，在MySQL中，FULL JOIN 需要使用左外连接 UNION 右外连接来实现，因为 MySQL 不直接支持 FULL JOIN。

```sql
SELECT c.id, c.name, o.product
FROM customers c
LEFT JOIN orders o ON c.id = o.customer_id
UNION
SELECT c.id, c.name, o.product
FROM customers c
RIGHT JOIN orders o ON c.id = o.customer_id;
```

MySQL如何避免插入重复数据

使用UNIQUE约束

- 在表的相关列上添加UNIQUE约束，确保每个值在该列中唯一。

```sql
CREATE TABLE users (
    id INT PRIMARY KEY AUTO_INCREMENT,
    email VARCHAR(255) UNIQUE,
);
```

使用INSERT ... ON DUPLICATE KEY UPDATE

- 这种语句允许在插入记录时处理重复键的情况。如果插入的记录与现有记录冲突，可以选择更新现有记录：

```sql
INSERT INTO users (email, name)
VALUES ('example@example.com', 'John Doe')
ON DUPLICATE KEY UPDATE name = VALUES(name);
```

使用INSERT IGNORE

- 该语句会在插入记录时忽略那些因重复键而导致的插入错误。

```sql
INSERT IGNORE INTO users (email, name)
VALUES ('example@example.com', 'John Doe');
```

CHAR 和 VARCHAR有什么区别

- CHAR固定长度，定义时需要指定长度，存储时会在末尾补足空格
- VARCHAR可变长第，定义时需要指定最大长度

INT(1)和INT(10)区别

- 区别为显示宽度，常与ZEROFILL配合使用，比如，字段类型为 INT(4) ZEROFILL，实际存入 5 → 显示为 0005，实际存入 12345 → 显示仍为 12345（宽度超限时不截断）。

说一下外键约束

- 外键约束的作用是维护表与表之间的关系，确保数据的完整性和一致性。若A表中有一个B表的外键，则保证了A表中的每一个B字段都存在于B表中，从而维护了数据的完整性和一致性。

MySQL的关键字in和exist

- IN：用于检查左边的表达式是否存在于右边的列表或子查询的结果集中。如果存在，则IN 返回TRUE，否则返回FALSE。
- EXIST：用于判断子查询是否至少能返回一行数据。它不关心子查询返回什么数据，只关心是否有结果。如果子查询有结果，则EXISTS 返回TRUE，否则返回FALSE。
- 在很多情况下，EXISTS 的性能优于 IN，特别是当子查询的表很大时。这是因为EXISTS 一旦找到匹配项就会立即停止查询，而IN可能会扫描整个子查询结果集。
- IN 能够正确处理子查询中包含NULL值的情况，而EXISTS 不受子查询结果中NULL值的影响，因为它关注的是行的存在性，而不是具体值。

MySQL中的一些函数

字符串函数

- CONCAT(str1, str2, ...)：连接多个字符串，返回一个合并后的字符串。
- LENGTH(str)：返回字符串的长度（字符数）。
- SUBSTRING(str, pos, len)：从指定位置开始，截取指定长度的子字符串。
- REPLACE(str, from_str, to_str)：将字符串中的某部分替换为另一个字符串。

数值函数

- ABS(num)：返回数字的绝对值。
- POWER(num, exponent)：返回指定数字的指定幂次方。

日期和时间函数

- NOW()：返回当前日期和时间。
- CURDATE()：返回当前日期。

聚合函数

- COUNT(column)：计算指定列中的非NULL值的个数。
- SUM(column)：计算指定列的总和。
- AVG(column)：计算指定列的平均值。
- MAX(column)：返回指定列的最大值。
- MIN(column)：返回指定列的最小值。

存储引擎

执行一条SQL请求的过程是什么?

```
客户端 → 连接器 → 查询缓存 → 分析器 → 优化器 → 执行器 → 存储引擎 → 返回结果
```

连接器

- 建立客户端与 MySQL 服务器的 TCP 连接，并进行身份认证和权限验证。

查询缓存

- MySQL 8.0 已移除该功能，但在旧版本中存在。
- 如果缓存中有完全相同的 SQL 结果，直接返回，跳过后续步骤。

分析器

- 词法分析：识别 SQL 中的关键字、表名、列名等。
- 语法分析：检查 SQL 是否符合语法规则，不合法会直接报错。
- 语义检查：确认表、列是否存在，数据类型是否匹配。

优化器

- 决定执行计划：选择使用哪个索引、表的连接顺序等。
- 目标是成本最小化（I/O + CPU）。

执行器

- 检查执行权限（SELECT、UPDATE 等）。
- 调用存储引擎 API（如 InnoDB、MyISAM）去读取或写入数据。
- 对于写操作，会记录到 binlog（归档日志）和 redo log（事务日志）。

讲一讲MySQL的引擎

- InnoDB：InnoDB是MySQL的默认存储引擎，具有ACID事务支持、行级锁、外键约束等特性。它适用于高并发的读写操作，支持较好的数据完整性和并发控制。
- MyISAM：MyISAM是MySQL的另一种常见的存储引擎，具有较低的存储空间和内存消耗，适用于大量读操作的场景。然而，MyISAM不支持事务、行级锁和外键约束，因此在并发写入和数据完整性方面有一定的限制。
- Memory：Memory引擎将数据存储在内存中，适用于对性能要求较高的读操作，但是在服务器重启或崩溃时数据会丢失。它不支持事务、行级锁和外键约束。

MySQL为什么InnoDB是默认引擎

- 事务支持：InnoDB引擎提供了对事务的支持，可以进行ACID（原子性、一致性、隔离性、持久性）属性的操作。Myisam存储引擎是不支持事务的。
- 并发性能：InnoDB引擎采用了行级锁定的机制，可以提供更好的并发性能，Myisam存储引擎只支持表锁，锁的粒度比较大。
- 崩溃恢复：InnoDB引引擎通过 redolog 日志实现了崩溃恢复，可以在数据库发生异常情况（如断电）时，通过日志文件进行恢复，保证数据的持久性和一致性。Myisam是不支持崩溃恢复的。

说一下MySQL的innodb与MyISAM的区别

- 事务：InnoDB 支持事务，MyISAM 不支持事务，这是 MySQL 将默认存储引擎从 MyISAM 变成 InnoDB 的重要原因之一。
- 索引结构：InnoDB 是聚簇索引，MyISAM 是非聚簇索引。聚簇索引的文件存放在主键索引的叶子节点上，因此 InnoDB 必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。而 MyISAM 是非聚簇索引，数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。
- 锁粒度：InnoDB 最小的锁粒度是行锁，MyISAM 最小的锁粒度是表锁。一个更新语句会锁住整张表，导致其他查询和更新都会被阻塞，因此并发访问受限。

- count 的效率：InnoDB 不保存表的具体行数，执行 select count(*) from table 时需要全表扫描。而MyISAM 用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快。

索引

索引分类

所有分类

- 按「数据结构」分类：B+tree索引、Hash索引、Full-text索引。
- 按「物理存储」分类：聚簇索引（主键索引）、二级索引（辅助索引）。
- 按「字段特性」分类：主键索引、唯一索引、普通索引、前缀索引。
- 按「字段个数」分类：单列索引、联合索引。

按数据结构分类

索引类型	InnoDB 引擎	MyISAM 引擎	Memory 引擎
B+Tree 索引	Yes	Yes	Yes
HASH 索引	No (不支持hash索引，但是在内存结构中有一个自适应hash索引)	No	Yes
Full-Text 索引	Yes (MySQL 5.6 版本后支持)	Yes	No

- InnoDB 是在 MySQL 5.5 之后成为默认的 MySQL 存储引擎，B+Tree 索引类型也是 MySQL 存储引擎采用最多的索引类型。
- 在创建表时，InnoDB 存储引擎会根据不同的场景选择不同的列作为索引：
  - 如果有主键，默认会使用主键作为聚簇索引的索引键（key）；
  - 如果没有主键，就选择第一个不包含 NULL 值的唯一列作为聚簇索引的索引键（key）；
  - 在上面两个都没有的情况下，InnoDB 将自动生成一个隐式自增 id 列作为聚簇索引的索引键（key）；
- 其它索引都属于辅助索引（Secondary Index），也被称为二级索引或非聚簇索引。创建的主键索引和二级索引默认使用的是 B+Tree 索引。

按物理存储分类

- 从物理存储的角度来看，索引分为聚簇索引（主键索引）、二级索引（辅助索引）。
  - 主键索引的 B+Tree 的叶子节点存放的是实际数据，所有完整的用户记录都存放在主键索引的 B+Tree 的叶子节点里；
  - 二级索引的 B+Tree 的叶子节点存放的是主键值，而不是实际数据。

- 所以，在查询时使用了二级索引，如果查询的数据能在二级索引里查询的到，那么就不需要回表，这个过程就是覆盖索引。如果查询的数据不在二级索引里，就会先检索二级索引，找到对应的叶子节点，获取到主键值后，然后再检索主键索引，就能查询到数据了，这个过程就是回表。

按字段特性分类

- 主键索引
  - 主键索引就是建立在主键字段上的索引，通常在创建表的时候一起创建，一张表最多只有一个主键索引，索引列的值不允许有空值。
  - 在创建表时，创建主键索引的方式如下：

```
CREATE TABLE table_name  (
  PRIMARY KEY (index_column_1) USING BTREE
);
```

- 唯一索引
  - 唯一索引建立在 UNIQUE 字段上的索引，一张表可以有多个唯一索引，索引列的值必须唯一，但是允许有空值。
  - 在创建表时，创建唯一索引的方式如下：

```
CREATE TABLE table_name  (
  ....
  UNIQUE KEY(index_column_1,index_column_2,...)
);
```

- 建表后，如果要创建唯一索引，可以使用这面这条命令：

```
CREATE UNIQUE INDEX index_name
ON table_name(index_column_1,index_column_2,...);
```

- 普通索引
  - 普通索引就是建立在普通字段上的索引，既不要求字段为主键，也不要求字段为 UNIQUE。
  - 在创建表时，创建普通索引的方式如下：

```
CREATE TABLE table_name  (
  INDEX(index_column_1,index_column_2,...)
);
```

- 建表后，如果要创建普通索引，可以使用这面这条命令：

```
CREATE INDEX index_name
ON table_name(index_column_1,index_column_2,...);
```

- 前缀索引
  - 前缀索引是指对字符类型字段的前几个字符建立的索引，而不是在整个字段上建立的索引，前缀索引可以建立在字段类型为 char、 varchar、binary、varbinary 的列上。
  - 使用前缀索引的目的是为了减少索引占用的存储空间，提升查询效率。
  - 在创建表时，创建前缀索引的方式如下：

```
CREATE TABLE table_name(
    column_list,
    INDEX(column_name(length))
);
```

- 建表后，如果要创建前缀索引，可以使用这面这条命令：

```
CREATE INDEX index_name
ON table_name(column_name(length));
```

按字段个数分类

- 从字段个数的角度来看，索引分为单列索引、联合索引（复合索引）。
    - 建立在单列上的索引称为单列索引，比如主键索引；
    - 建立在多列上的索引称为联合索引；

MySQL聚簇索引和非聚簇索引的区别是什么

- 数据存储：在聚簇索引中，数据行按照索引键值的顺序存储，也就是说，索引的叶子节点包含了实际的数据行。这意味着索引结构本身就是数据的物理存储结构。非聚簇索引的叶子节点不包含完整的数据行，而是包含指向数据行的指针或主键值。数据行本身存储在聚簇索引中。
- 索引与数据关系：由于数据与索引紧密相连，当通过聚簇索引查找数据时，可以直接从索引中获得数据行，而不需要额外的步骤去查找数据所在的位置。当通过非聚簇索引查找数据时，首先在非聚簇索引中找到对应的主键值，然后通过这个主键值回溯到聚簇索引中查找实际的数据行，这个过程称为“回表”。
- 唯一性：聚簇索引通常是基于主键构建的，因此每个表只能有一个聚簇索引，因为数据只能有一种物理排序方式。一个表可以有多个非聚簇索引，因为它们不直接影响数据的物理存储位置。
- 效率：对于范围查询和排序查询，聚簇索引通常更有效率，因为它避免了额外的寻址开销。非聚簇索引在使用覆盖索引进行查询时效率更高，因为它不需要读取完整的数据行。但是需要进行回表的操作，使用非聚簇索引效率比较低，因为需要进行额外的回表操作。

如果聚簇索引的数据更新，它的存储要不要变化

- 如果更新的数据是非索引数据，也就是普通的用户记录，那么存储结构是不会发生变化
- 如果更新的数据是索引数据，那么存储结构是有变化的，因为要维护 b+树的有序性

MySQL主键是聚簇索引吗

- 在MySQL的InnoDB存储引擎中，主键确实是以聚簇索引的形式存储的。
- InnoDB将数据存储在B+树的结构中，其中主键索引的B+树就是所谓的聚簇索引。这意味着表中的数据行在物理上是按照主键的顺序排列的，聚簇索引的叶节点包含了实际的数据行。
- InnoDB 在创建聚簇索引时，会根据不同的场景选择不同的列作为索引：
    - 如果有主键，默认会使用主键作为聚簇索引的索引键；
    - 如果没有主键，就选择第一个不包含 NULL 值的唯一列作为聚簇索引的索引键；
    - 在上面两个都没有的情况下，InnoDB 将自动生成一个隐式自增 id 列作为聚簇索引的索引键；
- 一张表只能有一个聚簇索引，那为了实现非主键字段的快速搜索，就引出了二级索引（非聚簇索引/辅助索引），它也是利用了 B+ 树的数据结构，但是二级索引的叶子节点存放的是主键值，不是实际数据。

表中十个字段，你主键用自增ID还是UUID，为什么?

- 用的是自增 id。
- 因为 uuid 相对顺序的自增 id 来说是毫无规律可言的，新行的值不一定要比之前的主键的值要大，所以 innodb 无法做到总是把新行插入到索引的最后，而是需要为新行寻找新的合适的位置从而来分配新的空间。
- 这个过程需要做很多额外的操作，数据的毫无顺序会导致数据分布散乱，将会导致以下的问题：
    - 写入的目标页很可能已经刷新到磁盘上并且从缓存上移除，或者还没有被加载到缓存中，innodb 在插入之前不得不先找到并从磁盘读取目标页到内存中，这将导致大量的随机 IO。
    - 因为写入是乱序的，innodb 不得不频繁的做页分裂操作，以便为新的行分配空间，页分裂导致移动大量的数据，影响性能。
    - 由于频繁的页分裂，页会变得稀疏并被不规则的填充，最终会导致数据会有碎片。
- 结论：使用 InnoDB 应该尽可能的按主键的自增顺序插入，并且尽可能使用单调的增加的聚簇键的值来插入新行。

什么自增ID更快一些，UUID不快吗，它在B+树里面存储是有序的吗

- 自增的主键的值是顺序的，所以 Innodb 把每一条记录都存储在一条记录的后面，所以自增 id 更快的原因：
    - 下一条记录就会写入新的页中，一旦数据按照这种顺序的方式加载，主键页就会近乎于顺序的记录填满，提升了页面的最大填充率，不会有页的浪费
    - 新插入的行一定会在原有的最大数据行下一行，mysql定位和寻址很快，不会为计算新行的位置而做出额外的消耗
    - 减少了页分裂和碎片的产生
- 但是 UUID 不是递增的，MySQL 中索引的数据结构是 B+Tree，这种数据结构的特点是索引树上的节点的数据是有序的，而如果使用 UUID 作为主键，那么每次插入数据时，因为无法保证每次产生的 UUID 有序，所以就会出现新的 UUID 需要插入到索引树的中间去，这样可能会频繁地导致页分裂，使性能下降。
- 而且，UUID 太占用内存。每个 UUID 由 36 个字符组成，在字符串进行比较时，需要从前往后比较，字符串越长，性能越差。另外字符串越长，占用的内存越大，由于页的大小是固定的，这样一个页上能存放的关键字数量就会越少，这样最终就会导致索引树的高度越大，在索引搜索的时候，发生的磁盘 IO 次数越多，性能越差。

Mysql中的索引是怎么实现的

- MySQL InnoDB 引擎是用了B+树作为了索引的数据结构。
- B+Tree 是一种多叉树，叶子节点才存放数据，非叶子节点只存放索引，而且每个节点里的数据是按主键顺序存放的。每一层父节点的索引值都会出现在下层子节点的索引值中，因此在叶子节点中，包括了所有的索引值信息，并且每一个叶子节点都有两个指针，分别指向下一个叶子节点和上一个叶子节点，形成一个双向链表。

B+树的特性是什么

- 所有叶子节点都在同一层：这是B+树的一个重要特性，确保了所有数据项的检索都具有相同的I/O延迟，提高了搜索效率。每个叶子节点都包含指向相邻叶子节点的指针，形成一个链表，由于叶子节点之间的链接，B+树非常适合进行范围查询和排序扫描。可以沿着叶子节点的链表顺序访问数据，而无需进行多次随机访问。
- 非叶子节点存储键值：非叶子节点仅存储键值和指向子节点的指针，不包含数据记录。这些键值用于指导搜索路径，帮助快速定位到正确的叶子节点。并且，由于非叶子节点只存放键值，当数据量比较大时，相对于B树，B+树的层高更少，查找效率也就更高。
- 叶子节点存储数据记录：与B树不同，B+树的叶子节点存储实际的数据记录或指向数据记录的指针。这意味着每次搜索都会到达叶子节点，才能找到所需数据。

- 自平衡：B+树在插入、删除和更新操作后会自动重新平衡，确保树的高度保持相对稳定，从而保持良好的搜索性能。每个节点最多可以有M个子节点，最少可以有ceil(M/2，M为阶数)个子节点（除了根节点），这里的M是树的阶数。

说说B+树和B树的区别

- 在B+树中，数据都存储在叶子节点上，而非叶子节点只存储索引信息；而B树的非叶子节点既存储索引信息也存储部分数据。
- B+树的叶子节点使用链表相连，便于范围查询和顺序访问；B树的叶子节点没有链表连接。
- B+树的查找性能更稳定，每次查找都需要查找到叶子节点；而B树的查找可能会在非叶子节点找到数据，性能相对不稳定。

B+树的好处是什么

- B+ 树的非叶子节点不存放实际的记录数据，仅存放索引，因此数据量相同的情况下，相比存储即存索引又存记录的 B 树，B+树的非叶子节点可以存放更多的索引，因此 B+ 树可以比 B 树更「矮胖」，查询底层节点的磁盘I/O次数会更少。
- B+ 树有大量的冗余节点（所有非叶子节点都是冗余索引），这些冗余索引让 B+ 树在插入、删除的效率都更高，比如删除根节点的时候，不会像 B 树那样会发生复杂的树的变化；
- B+ 树叶子节点之间用链表连接了起来，有利于范围查询，而 B 树要实现范围查询，因此只能通过树的遍历来完成范围查询，这会涉及多个节点的磁盘 I/O 操作，范围查询效率不如 B+ 树。

为什么 MysSQL 不用跳表?

- B+树的高度在3层时存储的数据可能已达千万级别，但对于跳表而言同样去维护千万的数据量那么所造成的跳表层数过高而导致的磁盘io次数增多，也就是使用B+树在存储同样的数据下磁盘io次数更少。

联合索引的实现原理

- 使用多个字段值作为key值
- 需要满足最左匹配，因为除了最左边的属性，剩下的属性都是全局无序，局部有序的，利用索引的前提时索引里的key时有序的

索引失效有哪些情况

- 当我们使用左或者左右模糊匹配的时候，也就是 like %xx 或者 like %xx%这两种方式都会造成索引失效；
- 当我们在查询条件中对索引列使用函数，就会导致索引失效。
- 当我们在查询条件中对索引列进行表达式计算，也是无法走索引的。
- MySQL 在遇到字符串和数字比较的时候，会自动把字符串转为数字，然后再进行比较。如果字符串是索引列，而条件语句中的输入参数是数字的话，那么索引列会发生隐式类型转换，由于隐式类型转换是通过 CAST 函数实现的，等同于对索引列使用了函数，所以就会导致索引失效。
- 联合索引要能正确使用需要遵循最左匹配原则，也就是按照最左优先的方式进行索引的匹配，否则就会导致索引失效。
- 在 WHERE 子句中，如果在 OR 前的条件列是索引列，而在 OR 后的条件列不是索引列，那么索引会失效。

什么情况下会回表查询

- 从物理存储的角度来看，索引分为聚簇索引（主键索引）、二级索引（辅助索引）。
- 它们的主要区别如下：
  - 主键索引的 B+Tree 的叶子节点存放的是实际数据，所有完整的用户记录都存放在主键索引的 B+Tree 的叶子节点里；
  - 二级索引的 B+Tree 的叶子节点存放的是主键值，而不是实际数据。
- 所以，在查询时使用了二级索引，如果查询的数据能在二级索引里查询的到，那么就不需要回表，这个过程就是覆盖索引。
- 如果查询的数据不在二级索引里，就会先检索二级索引，找到对应的叶子节点，获取到主键值后，然后再检索主键索引，就能查询到数据了，这个过程就是回表。

什么是覆盖索引?

- 覆盖索引是指一个索引包含了查询所需的所有列，因此不需要访问表中的数据行就能完成查询。
- 换句话说，查询所需的所有数据都能从索引中直接获取，而不需要进行回表查询。覆盖索引能够显著提高查询性能，因为减少了访问数据页的次数，从而减少了I/O操作。

如果一个列即使单列索引，又是联合索引，单独查它的话先走哪个?

- mysql 优化器会分析每个索引的查询成本，然后选择成本最低的方案来执行 sql。
- 如果单列索引是 a，联合索引是（a，b），那么针对下面这个查询：

```
select a, b from table where a = ? and b =?
```

- 优化器会选择联合索引，因为查询成本更低，查询也不需要回表，直接索引覆盖了。

索引已经建好了，那我再插入一条数据，索引会有哪些变化?

- 插入新数据可能导致B+树结构的调整和索引信息的更新，以保持B+树的平衡性和正确性，这些变化通常由数据库系统自动处理，确保数据的一致性和索引的有效性。
- 如果插入的数据导致叶子节点已满，可能会触发叶子节点的分裂操作，以保持B+树的平衡性。

索引的缺点

- 需要占用物理空间，数量越大，占用空间越大；
- 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增大；
- 会降低表的增删改的效率，因为每次增删改索引，B+ 树为了维护索引有序性，都需要进行动态维护。

怎么决定建立哪些索引

什么时候适合使用索引

- 字段有唯一性限制的，比如商品编码；
- 经常用于 WHERE 查询条件的字段，这样能够提高整个表的查询速度，如果查询条件不是一个字段，可以建立联合索引。
- 经常用于 GROUP BY 和 ORDER BY 的字段，这样在查询的时候就不需要再去做一次排序了，因为我们都已经知道了建立索引之后在 B+Tree 中的记录都是排序好的。

什么时候不需要创建索引

- WHERE 条件，GROUP BY，ORDER BY 里用不到的字段，索引的价值是快速定位，如果起不到定位的字段通常是不需要创建索引的，因为索引是会占用物理空间的。
- 字段中存在大量重复数据，不需要创建索引，比如性别字段，只有男女，如果数据库表中，男女的记录分布均匀，那么无论搜索哪个值都可能得到一半的数据。在这些情况下，还不如不要索引，因为 MySQL 还有一个查询优化器，查询优化器发现某个值出现在表的数据行中的百分比很高的时候，它一般会忽略索引，进行全表扫描。
- 表数据太少的时候，不需要创建索引；
- 经常更新的字段不用创建索引

索引优化详细讲讲

- 前缀索引优化：使用前缀索引是为了减小索引字段大小，可以增加一个索引页中存储的索引值的数量，有效提高索引的查询速度。在一些大字符串的字段作为索引时，使用前缀索引可以帮助我们减小索引项的大小。
- 覆盖索引优化：覆盖索引是指 SQL 中 query 的所有字段，在索引 B+Tree 的叶子节点上都能找得到的那些索引，从二级索引中查询得到记录，而不需要通过聚簇索引查询获得，可以避免回表的操作。
- 主键索引最好是自增的：
    - 如果我们使用自增主键，那么每次插入的新数据就会按顺序添加到当前索引节点的位置，不需要移动已有的数据，当页面写满，就会自动开辟一个新页面。因为每次插入一条新记录，都是追加操作，不需要重新移动数据，因此这种插入数据的方法效率非常高。
    - 如果我们使用非自增主键，由于每次插入主键的索引值都是随机的，因此每次插入新的数据时，就可能会插入到现有数据页中间的某个位置，这将不得不移动其它数据来满足新数据的插入，甚至需要从一个页面复制数据到另外一个页面，我们通常将这种情况称为页分裂。页分裂还有可能会造成大量的内存碎片，导致索引结构不紧凑，从而影响查询效率。
- 防止索引失效：
    - 当我们使用左或者左右模糊匹配的时候，也就是 like %xx 或者 like %xx%这两种方式都会造成索引失效；
    - 当我们在查询条件中对索引列做了计算、函数、类型转换操作，这些情况下都会造成索引失效；
    - 联合索引要能正确使用需要遵循最左匹配原则，也就是按照最左优先的方式进行索引的匹配，否则就会导致索引失效。
    - 在 WHERE 子句中，如果在 OR 前的条件列是索引列，而在 OR 后的条件列不是索引列，那么索引会失效。

事务

事务的特性

原子性

- 一个事务中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节，而且事务在执行过程中发生错误，会被回滚到事务开始前的状态，就像这个事务从来没有执行过一样，就好比买一件商品，购买成功时，则给商家付了钱，商品到手；购买失败时，则商品在商家手中，消费者的钱也没花出去。
- 通过 Undo log 来保证的；每个事务的操作都会记录一份“撤销日志”（Undo Log），用于回滚。

一致性

- 事务执行前后，数据库必须保持一致的状态，符合所有的约束规则。

隔离性

- 数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致，因为多个事务同时使用相同的数据时，不会相互干扰，每个事务都有一个完整的数据空间，对其他并发事务是隔离的。也就是说，消费者购买商品这个事务，是不影响其他消费者购买的。
- 通过 MVCC（多版本并发控制） 或锁机制来保证的；

持久性

- 事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。
- 所有事务提交前，都会将修改记录写入 Redo Log（重做日志），即使系统崩溃也能恢复。

MySQL可能出现的并发问题

脏读

- 一个事务读取了另一个事务尚未提交的数据。
- 事务A将余额从100修改为200，但还未提交，事务B读到了200，事务A回滚到100。事务B读到的是脏数据，因为它读取了一个最终被撤销的值

幻读

- 一个事务在读取某个范围的数据时，另一个事务插入了新数据，导致前后读取结果不一致。
- 事务A：查询所有价格大于100的商品，发现有5条。事务B：插入了一条价格为150的新商品，并提交。事务A再次查询，发现有6条记录。事务A“看到了幻影”，因为它在执行过程中，数据集合发生了变化。

不可重复读

- 一个事务在读取同一条记录时，前后读到的值不一致。
- 事务A：读取订单状态，发现是“未支付”。事务B：将该订单状态改为“已支付”，并提交。事务A再次读取该订单，发现状态变成了“已支付”。事务A在执行过程中，数据发生了变化，导致前后不一致。

MySQL如何解决并发问题的

- 锁机制：Mysql提供了多种锁机制来保证数据的一致性，包括行级锁、表级锁、页级锁等。通过锁机制，可以在读写操作时对数据进行加锁，确保同时只有一个操作能够访问或修改数据。
- 事务隔离级别：Mysql提供了多种事务隔离级别，包括读未提交、读已提交、可重复读和串行化。通过设置合适的事务隔离级别，可以在多个事务并发执行时，控制事务之间的隔离程度，以避免数据不一致的问题。
- MVCC（多版本并发控制）：Mysql使用MVCC来管理并发访问，它通过在数据库中保存不同版本的数据来实现不同事务之间的隔离。在读取数据时，Mysql会根据事务的隔离级别来选择合适的数据版本，从而保证数据的一致性。

事务的隔离级别有哪些

- 读未提交（read uncommitted），指一个事务还没提交时，它做的变更就能被其他事务看到；可能发生脏读、不可重复读和幻读现象；
- 读提交（read committed），指一个事务提交之后，它做的变更才能被其他事务看到；可能发生不可重复读和幻读现象，但是不可能发生脏读现象；
- 可重复读（repeatable read），指一个事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，MySQL InnoDB 引擎的默认隔离级别；可能发生幻读现象，但是不可能脏读和不可重复读现象；

- 串行化（serializable）；会对记录加上读写锁，在多个事务对这条记录进行读写操作时，如果发生了读写冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行；脏读、不可重复读和幻读现象都不可能会发生。

四种隔离级别如何实现的

- 对于「读未提交」隔离级别的事务来说，因为可以读到未提交事务修改的数据，所以直接读取最新的数据就好了；
- 对于「串行化」隔离级别的事务来说，通过加读写锁的方式来避免并行访问；
- 对于「读提交」和「可重复读」隔离级别的事务来说，它们是通过 Read View来实现的，它们的区别在于创建 Read View 的时机不同，「读提交」隔离级别是在「每个语句执行前」都会重新生成一个 Read View，而「可重复读」隔离级别是「启动事务时」生成一个 Read View，然后整个事务期间都在用这个 Read View。

可重复读下的幻读问题

- 我做了个实验去测试可重复读会不会出现幻读：
  - 事务A开启事务，搜索班级id为1的学生，此时为两条记录；
  - 事务B开启事务，插入一条数据进入表中，并提交；
  - 事务A再次搜索班级id为1的学生，此时为两条记录；
  - 事务A将所有名字改为“修改名字”；
  - 事务A再次搜索班级id为1的学生，此时为三条记录；
- 原因：事务A进行更新时，使用的是当前读，当前读不再用事务开始时的快照，而是直接去读最新数据，并对符合条件的行加锁，此时会把这条新插入的记录也读出来一起更新
- 🧩 关键原理
  - 事务的快照没变
    - 在 Repeatable Read 下，事务第一次做快照读时建立的一致性视图（snapshot）不会因为 UPDATE 而重建。
    - 这个视图仍然屏蔽掉事务开始后，其他事务提交的新行。
  - 本事务的修改优先可见
    - MVCC 有个规则：同一行如果存在本事务写过的版本，那即使这个版本在最初快照里不可见，也必须返回给本事务（Read-Your-Writes）。
    - 所以当你 UPDATE 时，假设它命中了原来快照中看不到的新行（比如被别的事务插入的小刚），此刻它被你生成了一个“由自己写的版本”。
  - 普通 SELECT 也会“看到”它
    - 虽然 SELECT 本身还是走快照读的逻辑，但会先检查“这行有没有我自己写的版本”。
    - 于是这条原本不在快照中的记录被“补进了”结果集，看起来就像快照更新了，实际上是可见性规则在起作用。

Mysql 设置了可重读隔离级后，怎么保证不发生幻读?

- 尽量在开启事务之后，马上执行 select ... for update 这类锁定读的语句，因为它会对记录加 next-key lock，从而避免其他事务插入一条新记录，就避免了幻读的问题。

串行化隔离级别是通过什么实现的?

- 是通过行级锁来实现的，序列化隔离级别下，普通的 select 查询是会对记录加 S 型的 next-key 锁，其他事务就没没办法对这些已经加锁的记录进行增删改操作了，从而避免了脏读、不可重复读和幻读现象。

MVCC实现原理

MVCC定义

- 是 InnoDB 在不加读锁的情况下，实现读写互不阻塞的关键机制。核心思路是：每行数据维护多个历史版本，读操作基于一致性快照获取可见版本，从而提高并发性能。

核心原理

- 隐藏列：针对每一行数据的聚簇索引，每行有 DB_TRX_ID（最后修改事务ID）和 DB_ROLL_PTR（指向 Undo 日志的上一个版本）。
- 版本链：旧数据不会直接覆盖，而是写入 Undo 日志，并通过 DB_ROLL_PTR 串成链。
- Read View：快照读时生成，有四个字段：
  - creator_trx_id：创建该Read View的事务ID
  - m_ids：创建时，当前数据库中活跃但未提交的事务ID列表
  - up_limit_id（min_trx_id）：创建时，当前数据库中活跃但未提交的最小事务ID
  - low_limit_id（max_trx_id）：创建时，当前数据库应该给下一个事务的ID值
- 可见性判断：根据事务ID和活跃事务列表决定读哪个版本。
  - 版本事务 ID < up_limit_id → 该事务早已提交，可见。
  - 版本事务 ID ≥ low_limit_id → 该事务在视图创建后才产生，不可见。
  - up_limit_id ≤ 版本事务 ID < low_limit_id
    - 如果 DB_TRX_ID 不在 m_ids（说明视图生成时它已经提交） → 可见。
    - 如果在 m_ids（说明当时事务还活跃） → 不可见。
  - 如果 DB_TRX_ID == creator_trx_id → 即使未提交，也对本事务可见（读你所写）。
  - “小于最小 ID 看得见，大于等于最大 ID 看不见；夹在中间要查表，表里有就看不见；自己永远能看到自己。”
- Purge 清理：无事务再引用的历史版本会被后台线程物理删除，释放空间。

一条update是不是原子性的？为什么?

- 是原子性，主要通过锁+undolog 日志保证原子性的
- 执行 update 的时候，会加行级别锁，保证了一个事务更新一条记录的时候，不会被其他事务干扰。
- 事务执行过程中，会生成 undolog，如果事务执行失败，就可以通过 undolog 日志进行回滚。

滥用事务，或者一个事务里有特别多sql的弊端

- 锁时间过长，导致并发性能下降
  - 事务未提交前，持有的锁会一直占着资源，其他会话只能干等
  - 高并发场景下，会放大阻塞甚至形成死锁
- 资源消耗大
  - 数据库需要保存事务的中间状态（Undo/Redo 日志、版本信息）

    - SQL 多、事务长，内存和磁盘压力都会上去
- 回滚成本高
    - 出错时需要撤销整个事务
    - SQL 数量多时，回滚耗时长，甚至可能产生锁等待的连锁反应
- 不利于扩展与分布式场景
    - 长事务会拖慢分布式事务协调器，增加跨节点一致性开销

锁

讲一下mysql里有哪些锁?

- 全局锁：通过flush tables with read lock 语句会将整个数据库就处于只读状态了，这时其他线程执行以下操作，增删改或者表结构修改都会阻塞。全局锁主要应用于做全库逻辑备份，这样在备份数据库期间，不会因为数据或表结构的更新，而出现备份文件的数据与预期的不一样。
- 表级锁：MySQL 里面表级别的锁有这几种：
    - 表锁：通过lock tables 语句可以对表加表锁，表锁除了会限制别的线程的读写外，也会限制本线程接下来的读写操作。
    - 元数据锁：当我们对数据库表进行操作时，会自动给这个表加上 MDL，对一张表进行 CRUD 操作时，加的是 MDL 读锁；对一张表做结构变更操作的时候，加的是 MDL 写锁；MDL 是为了保证当用户对表执行 CRUD 操作时，防止其他线程对这个表结构做了变更。
    - 意向锁：当执行插入、更新、删除操作，需要先对表加上「意向独占锁」，然后对该记录加独占锁。意向锁的目的是为了快速判断表里是否有记录被加锁。
- 行级锁：InnoDB 引擎是支持行级锁的，而 MyISAM 引擎并不支持行级锁。
    - 记录锁，锁住的是一条记录。而且记录锁是有 S 锁和 X 锁之分的，满足读写互斥，写写互斥
    - 间隙锁，只存在于可重复读隔离级别，目的是为了解决可重复读隔离级别下幻读的现象。
    - Next-Key Lock 称为临键锁，是 Record Lock + Gap Lock 的组合，锁定一个范围，并且锁定记录本身。

数据库的表锁和行锁有什么作用?

表锁

- 整体控制：表锁可以用来控制整个表的并发访问，当一个事务获取了表锁时，其他事务无法对该表进行任何读写操作，从而确保数据的完整性和一致性。
- 适用于大批量操作：表锁适合于需要大批量操作表中数据的场景，例如表的重建、大量数据的加载等。

行锁

- 细粒度控制：行锁可以精确控制对表中某行数据的访问，使得其他事务可以同时访问表中的其他行数据，在并发量大的系统中能够提高并发性能。
- 适用于频繁单行操作：行锁适合于需要频繁对表中单独行进行操作的场景，例如订单系统中的订单修改、删除等操作

MySQL两个线程的update语句同时处理一条数据，会不会有阻塞?

- 会，因为行级锁，陷入阻塞

两条update语句处理一张表的不同的主键范围的记录，一个<10，一个>15，会不会遇到阻塞？底层是为什么的?

- 不会，因为锁的范围不同，一个是小于10加锁，另一个是大于15加锁

如果2个范围不是主键或索引？还会阻塞吗?

- 如果2个范围查询的字段不是索引的话，那就代表 update 没有用到索引，这时候触发了全表扫描，全部索引都会加行级锁，这时候第二条 update 执行的时候，就会阻塞了。
- 因为如果 update 没有用到索引，在扫描过程中会对索引加锁，所以全表扫描的场景下，所有记录都会被加锁。

日志

日志文件是分成了哪几种?

- redo log 重做日志，是 Innodb 存储引擎层生成的日志，实现了事务中的持久性，主要用于掉电等故障恢复；
- undo log 回滚日志，是 Innodb 存储引擎层生成的日志，实现了事务中的原子性，主要用于事务回滚和 MVCC。
- bin log 二进制日志，是 Server 层生成的日志，主要用于数据备份和主从复制；
- relay log 中继日志，用于主从复制场景下，slave通过io线程拷贝master的bin log后本地生成的日志
- 慢查询日志，用于记录执行时间过长的sql，需要设置阈值后手动开启

讲一下binlog

- MySQL 在完成一条更新操作后，Server 层还会生成一条 binlog，等之后事务提交的时候，会将该事物执行过程中产生的所有 binlog 统一写 入 binlog 文件，binlog 是 MySQL 的 Server 层实现的日志，所有存储引擎都可以使用。
- binlog 是追加写，写满一个文件，就创建一个新的文件继续写，不会覆盖以前的日志，保存的是全量的日志，用于备份恢复、主从复制；
- binlog 文件是记录了所有数据库表结构变更和表数据修改的日志，不会记录查询类的操作，比如 SELECT 和 SHOW 操作。
- binlog 有 3 种格式类型，分别是 STATEMENT（默认格式）、ROW、 MIXED，区别如下：
- STATEMENT：每一条修改数据的 SQL 都会被记录到 binlog 中（相当于记录了逻辑操作，所以针对这种格式，binlog 可以称为逻辑日志），主从复制中 slave 端再根据 SQL 语句重现。但 STATEMENT 有动态函数的问题，比如你用了 uuid 或者 now 这些函数，你在主库上执行的结果并不是你在从库执行的结果，这种随时在变的函数会导致复制的数据不一致；
- ROW：记录行数据最终被修改成什么样了（这种格式的日志，就不能称为逻辑日志了），不会出现 STATEMENT 下动态函数的问题。但 ROW 的缺点是每行数据的变化结果都会被记录，比如执行批量 update 语句，更新多少行数据就会产生多少条记录，使 binlog 文件过大，而在 STATEMENT 格式下只会记录一个 update 语句而已；
- MIXED：包含了 STATEMENT 和 ROW 模式，它会根据不同的情况自动使用 ROW 模式和 STATEMENT 模式；

UndoLog日志的作用是什么?

- undo log 是一种用于撤销回退的日志，它保证了事务的 ACID 特性中的原子性（Atomicity）。
- 在事务没提交之前，MySQL 会先记录更新前的数据到 undo log 日志文件里面，当事务回滚时，可以利用 undo log 来进行回滚。
- 每当 InnoDB 引擎对一条记录进行操作（修改、删除、新增）时，要把回滚时需要的信息都记录到 undo log 里，比如：

- 在插入一条记录时，要把这条记录的主键值记下来，这样之后回滚时只需要把这个主键值对应的记录删掉
- 在删除一条记录时，要把这条记录中的内容都记下来，这样之后回滚时再把由这些内容组成的记录插入到表中
- 在更新一条记录时，要把被更新的列的旧值记下来，这样之后回滚时再把这些列更新为旧值

有了undolog为啥还需要redolog呢?

- 在数据库中，Undo Log 用于回滚未完成事务，保证原子性；Redo Log 用于在系统崩溃后重做已提交事务，保证持久性。两者相辅相成，缺一不可。
- Undo Log记录修改前的旧值，Redo Log记录修改后的新值。

redo log怎么保证持久性的?

WAL 原则（Write-Ahead Logging）

- 先写日志，再写数据文件
- 当事务提交时，必须先把 Redo Log 刷入磁盘（fsync 确保真正落盘），确认写成功后才算事务提交完成。
- 这样即使数据页还停留在缓冲池里没落盘，宕机也可依靠日志恢复。

顺序写 + 刷盘确认

- Redo Log 是顺序追加写（append-only），比随机写快得多，也更容易落盘成功。
- 顺序写降低了磁盘寻址开销，减少了因延迟导致的风险。

组提交（Group Commit）

- 多个事务的 Redo Log 可一次性刷盘，减少磁盘 I/O 次数，但提交确认前必须确保日志写入可靠。
- 提升性能的同时保持持久性。

崩溃恢复流程

- 数据库启动时会扫描 Redo Log 中的 已提交事务记录，将对应的修改“重做”到数据文件中。
- 因为提交前就已保证日志落盘，所以能准确恢复到崩溃前的最后提交状态。

binlog 两阶段提交过程

- 当客户端执行 commit 语句或者在自动提交的情况下，MySQL 内部开启一个 XA 事务，分两阶段来完成 XA 事务的提交
- 事务的提交过程有两个阶段，就是将 redo log 的写入拆成了两个步骤：prepare 和 commit，中间再穿插写入 binlog：
  - prepare 阶段：将 XID（内部 XA 事务的 ID） 写入到 redo log，同时将 redo log 对应的事务状态设置为 prepare，然后将 redo log 持久化到磁盘（innodb_flush_log_at_trx_commit = 1 的作用）；
  - commit 阶段：把 XID 写入到 binlog，然后将 binlog 持久化到磁盘（sync_binlog = 1 的作用），接着调用引擎的提交事务接口，将 redo log 状态设置为 commit，此时该状态并不需要持久化到磁盘，只需要 write 到文件系统的 page cache 中就够了，因为只要 binlog 写磁盘成功，就算 redo log 的状态还是 prepare 也没有关系，一样会被认为事务已经执行成功；
- 两阶段提交是以 binlog 写成功为事务提交成功的标识

为什么要写RedoLog，而不是直接写到B+树里面?

- 因为 redolog 写入磁盘是顺序写，而 b+树里数据页写入磁盘是随机写，顺序写的性能会比随机写好，这样可以提升事务提交的效率。
- 最重要的是redolog具备故障恢复的能力，Redo Log 记录的是物理级别的修改，包括页的修改，如插入、更新、删除操作在磁盘上的物理位置和修改内容。例如，当执行一个更新操作时，Redo Log 会记录修改的数据页的地址和更新后的数据，而不是 SQL 语句本身。
- 在数据页实际更新之前，先将修改操作写入 Redo Log。当数据库重启时，会进行恢复操作。首先，根据 Redo Log 检查哪些事务已经提交但数据页尚未完全写入磁盘。然后，使用 Redo Log 中的记录对这些事务进行重做（Redo）操作，将未完成的数据页修改完成，确保事务的修改生效。

Redis数据结构

Redis有哪些数据结构

类型	简介	特性	场景
String(字符串)	二进制安全	可以包含任何数据,比如jpg图片或者序列化的对象,一个键最大能存储512M	---
Hash(字典)	键值对集合,即编程语言中的Map类型	适合存储对象,并且可以像数据库中update一个属性一样只修改某一项属性值(Memcached中需要取出整个字符串反序列化成对象修改完再序列化存回去)	存储、读取、修改用户属性
List(列表)	链表(双向链表)	增删快,提供了操作某一段元素的API	1,最新消息排行等功能(比如朋友圈的时间线) 2,消息队列
Set(集合)	哈希表实现,元素不重复	1、添加、删除,查找的复杂度都是O(1) 2、为集合提供了求交集、并集、差集等操作	1、共同好友 2、利用唯一性,统计访问网站的所有独立ip 3、好友推荐时,根据tag求交集,大于某个阈值就可以推荐
Sorted Set(有序集合)	将Set中的元素增加一个权重参数score,元素按score有序排列	数据插入集合时,已经进行天然排序	1、排行榜 2、带权重的消息队列

Zset 底层是怎么实现的?

- Zset 类型的底层数据结构是由压缩列表或跳表实现的：
  - 如果有序集合的元素个数小于 128 个，并且每个元素的值小于 64 字节时，Redis 会使用压缩列表作为 Zset 类型的底层数据结构；
  - 如果有序集合的元素不满足上面的条件，Redis 会使用跳表作为 Zset 类型的底层数据结构；
- 在 Redis 7.0 中，压缩列表数据结构已经废弃了，交由 listpack 数据结构来实现了。

跳表是怎么设置层高的

- 常见实现（如 Redis 的 zset）是：插入新节点时，从第 1 层开始，丢一枚“硬币”，正面就晋升一层，直到反面为止，或者达到最大层数上限。

Redis为什么使用跳表而不是用B+树

内存友好

- B+ 树节点一般需要连续的内存块来存储 key 数组和指针，内存分配与管理更复杂，在频繁插入/删除时可能产生碎片。
- Redis 是单线程事件循环模型，越简单的算法越能减少代码复杂度和 CPU 消耗。

范围查询天然高效

- 跳表的“横向+纵向”链表结构，可以非常高效地从某个分数（score）开始顺序遍历范围内的元素。
- 在 B+ 树中做范围查询虽然也高效，但涉及节点间的磁盘/页式结构优势在 Redis 的内存场景里不明显。

性能稳定

- 跳表查找、插入、删除的平均时间复杂度都是 O(log n)，而且性能波动小。
- B+ 树在节点分裂或合并时有额外的复杂维护成本。

实现简单

- 跳表 基于链表+多层索引结构，用纯指针连接，不需要像 B+ 树那样维护复杂的节点分裂、合并逻辑。

Redis压缩列表怎么实现的

- 是 Redis 为了节约内存而开发的，它是由连续内存块组成的顺序型数据结构，有点类似于数组。

结构布局

字段	大小	作用
`zlbytes`	4 字节	记录整个 ziplist 占用的字节数，方便内存重新分配
`zltail`	4 字节	记录最后一个节点相对于列表起始位置的偏移量，快速找到尾节点
`zllen`	2 字节	节点数量（< 65535），超过这个值就需要遍历统计
entry[]	不定	存储每个数据节点
`zlend`	1 字节	特殊标志 `0xFF`，表示列表结束

Entry结点格式

- prevlen
  - 记录前一个 entry 的长度（1 字节或 5 字节）
  - 用来支持从尾向前遍历
- encoding
  - 记录当前节点数据类型和长度（可以表示字符串或整数）
  - 如果是小整数，会直接存在 encoding 里而不是单独分配空间

- data
    - 真正存储的内容（字符串字节数组或整数）

压缩列表的性质

- 当我们往压缩列表中插入数据时，压缩列表就会根据数据类型是字符串还是整数，以及数据的大小，会使用不同空间大小的 prevlen 和 encoding 这两个元素里保存的信息，这种根据数据大小和类型进行不同的空间大小分配的设计思想，正是 Redis 为了节省内存而采用的。
- 压缩列表的缺点是会发生连锁更新的问题，因此连锁更新一旦发生，就会导致压缩列表占用的内存空间要多次重新分配，这就会直接影响到压缩列表的访问性能。
- 所以说，虽然压缩列表紧凑型的内存布局能节省内存开销，但是如果保存的元素数量增加了，或是元素变大了，会导致内存重新分配，最糟糕的是会有「连锁更新」的问题。
- 因此，压缩列表只会用于保存的节点数量不多的场景，只要节点数量足够小，即使发生连锁更新，也是能接受的。

介绍一下 Redis 中的 listpack

quicklist

- 使用一个双向链表，链表的结点是压缩列表
- quicklist 虽然通过控制 quicklistNode 结构里的压缩列表的大小或者元素个数，来减少连锁更新带来的性能影响，但是并没有完全解决连锁更新的问题。
- 因为 quicklistNode 还是用了压缩列表来保存元素，压缩列表连锁更新的问题，来源于它的结构设计，所以要想彻底解决这个问题，需要设计一个新的数据结构。

listpack

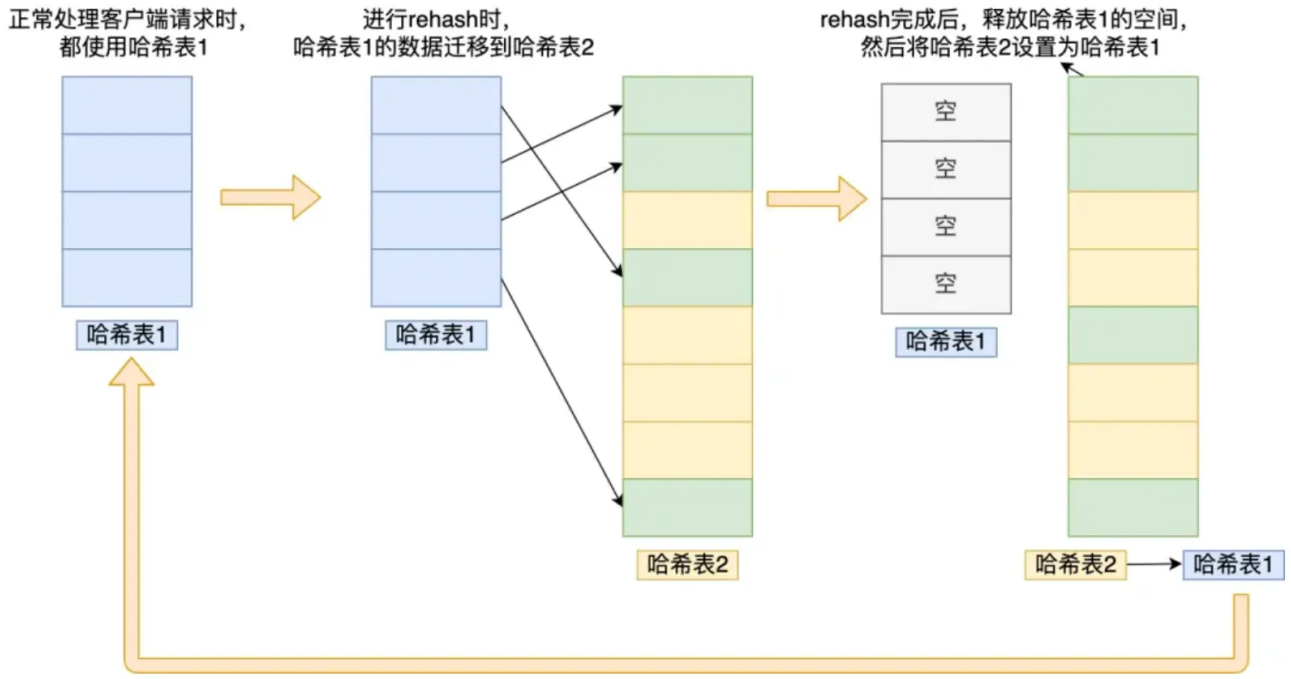
- 结构布局

字段	大小	作用
`total_bytes`	4 字节	整个 listpack 占用的总字节数
`num_elements`	2 字节	元素数量
`entry[]`	不定	依次存放每个数据节点
`end`	1 字节	固定值 `0xFF`，标识结尾

- 节点格式
    - encoding：存储数据类型及长度信息（变长编码，节省空间）
    - content：实际数据，可以是字符串或整数
    - backlen：记录整个 entry 占用的字节数，用于反向遍历

哈希表的扩容

过程



- 问题：如果「哈希表 1 」的数据量非常大，那么在迁移至「哈希表 2 」的时候，因为会涉及大量的数据拷贝，此时可能会对 Redis 造成阻塞，无法服务其他请求。

渐进式 rehash

- 给「哈希表 2」 分配空间；
- 在 rehash 进行期间，每次哈希表元素进行新增、删除、查找或者更新操作时，Redis 除了会执行对应的操作之外，还会顺序将「哈希表 1 」中索引位置上的所有 key-value 迁移到「哈希表 2」 上；新增只插入新表；查询时先查原表再查新表
- 随着处理客户端发起的哈希表操作请求数量越多，最终在某个时间点会把「哈希表 1 」的所有 key-value 迁移到「哈希表 2」，从而完成 rehash 操作。

String是用什么存储的？为什么不用 c 语言中的字符串

- Redis 的 String 并不是直接用 C 语言里原生的 char* 字符串来存储的，而是用 SDS（Simple Dynamic String，简单动态字符串） 这个自定义结构来实现的。 这是 Redis 的一个关键设计之一，目的是让字符串操作更高效、更安全、功能更丰富。

SDS的存储结构

字段	类型	含义
len	int / uint	记录当前已使用的字节数（不含末尾的 `\0`）
alloc	int / uint	已分配的缓冲区大小（不含末尾的 `\0`）
flags	unsigned char	记录 SDS 类型及编码等信息
buf[]	char 数组	实际存储数据的缓冲区，末尾会保留一个 `\0` 以便与 C 字符串兼容

不用 C 原生字符串

- C字符串获取长度为O(1)
- 支持二进制安全（Binary-safe）
  - C 字符串不能存 \0（会被当作结束符）
  - SDS 用 len 记录长度，可以存任意字节（包括 \0）
- 不会发生缓冲区溢出
  - C 语言的字符串标准库提供的字符串操作函数，大多数（比如 strcat 追加字符串函数）都是不安全的，因为这些函数把缓冲区大小是否满足操作需求的工作交由开发者来保证，程序内部并不会判断缓冲区大小是否足够用，当发生了缓冲区溢出就有可能造成程序异常结束。
  - 所以，Redis 的 SDS 结构里引入了 alloc 和 len 成员变量，这样 SDS API 通过 alloc - len 计算，可以算出剩余可用的空间大小，这样在对字符串做修改操作的时候，就可以由程序内部判断缓冲区大小是否足够用。
  - 而且，当判断出缓冲区大小不够用时，Redis 会自动将扩大 SDS 的空间大小，以满足修改所需的大小。

Reids线程模型

Redis使用单线程为什么快?

- Redis 的大部分操作都在内存中完成，并且采用了高效的数据结构，因此 Redis 瓶颈可能是机器的内存或者网络带宽，而并非 CPU，既然 CPU 不是瓶颈，那么自然就采用单线程的解决方案了；
- Redis 采用单线程模型可以避免了多线程之间的竞争，省去了多线程切换带来的时间和性能上的开销，而且也不会导致死锁问题。
- Redis 采用了 I/O 多路复用机制处理大量的客户端 Socket 请求，IO 多路复用机制是指一个线程处理多个 IO 流，就是我们经常听到的 select/epoll 机制。简单来说，在 Redis 只运行单线程的情况下，该机制允许内核中，同时存在多个监听 Socket 和已连接 Socket。内核会一直监听这些 Socket 上的连接请求或数据请求。一旦有请求到达，就会交给 Redis 线程处理，这就实现了一个 Redis 线程处理多个 IO 流的效果。

Redis哪些地方使用了多线程?

单线程

- Redis 单线程指的是「接收客户端请求->解析请求 ->进行数据读写等操作->发送数据给客户端」这个过程是由一个线程（主线程）来完成的，这也是我们常说 Redis 是单线程的原因。

后台线程

- Redis 在 2.6 版本，会启动 2 个后台线程，分别处理关闭文件、AOF 刷盘这两个任务；
- Redis 在 4.0 版本之后，新增了一个新的后台线程，用来异步释放 Redis 内存，也就是 lazyfree 线程。例如执行 unlink key / flushdb async / flushall async 等命令，会把这些删除操作交给后台线程来执行，好处是不会导致 Redis 主线程卡顿。因此，当我们要删除一个**大key**的时候，不要使用 del 命令删除，因为 del 是在主线程处理的，这样会导致 Redis 主线程卡顿，因此我们应该使用 unlink 命令来异步删除大key。
- 之所以单独创线程，是因为耗时，放在主线程会导致阻塞

网络请求

- 在 Redis 6.0 版本之后，也采用了多个 I/O 线程来处理网络请求，这是因为随着网络硬件的性能提升，Redis 的性能瓶颈有时会出现在网络 I/O 的处理上。
- 默认情况下，只针对响应数据使用多线程，要想开启多线程处理客户端请求，则需要修改配置

Redis会额外创建6个线程

- Redis-server： Redis的主线程，主要负责执行命令；
- bio_close_file、bio_aof_fsync、bio_lazy_free：三个后台线程，分别异步处理关闭文件任务、AOF刷盘任务、释放内存任务；
- io_thd_1、io_thd_2、io_thd_3：三个 I/O 线程，io-threads 默认是 4 ，所以会启动 3（4-1）个 I/O 多线程，用来分担 Redis 网络 I/O 的压力。

Redis怎么实现的io多路复用

- 因为 Redis 是跑在「单线程」中的，所有的操作都是按照顺序线性执行的，但是由于读写操作等待用户输入 或输出都是阻塞的，所以 I/O 操作在一般情况下往往不能直接返回，这会导致某一文件的 I/O 阻塞导，致整个进程无法对其它客户提供服务。而 I/O 多路复用就是为了解决这个问题而出现的。为了让单线程(进程)的服务端应用同时处理多个客户端的事件，Redis 采用了 IO 多路复用机制。
- 这里"多路"指的是多个网络连接客户端，"复用"指的是复用同一个线程(单进程)。I/O 多路复用其实是使用一个线程来检查多个 Socket 的就绪状态，在单个线程中通过记录跟踪每一个 socket（I/O流）的状态来管理处理多个 I/O 流。

Redis事务

如何实现Redis原子性

- redis 执行一条命令的时候是具备原子性的，因为 redis 执行命令的时候是单线程来处理的，不存在多线程安全的问题。
- 如果要保证 2 条命令的原子性的话，可以考虑用 lua 脚本，将多个操作写到一个 Lua 脚本中，Redis 会把整个 Lua 脚本作为一个整体执行，在执行的过程中不会被其他命令打断，从而保证了 Lua 脚本中操作的原子性。

除了lua有没有什么也能保证redis的原子性?

- redis 事务也可以保证多个操作的原子性。
- 如果 redis 事务正常执行，没有发生任何错误，那么使用 MULTI 和 EXEC 配合使用，就可以保证多个操作都完成。
- 但是，如果事务执行发生错误了，就没办法保证原子性了。比如说 2 个操作，第一个操作执行成果了，但是第二个操作执行的时候，命令出错了，那事务并不会回滚，因为Redis 中并没有提供回滚机制。

Redis日志

Redis有哪2种持久化方式？分别的优缺点是什么?

AOF 日志

- Redis 在执行完一条写操作命令后，就会把该命令以追加的方式写入到一个文件里，然后 Redis 重启时，会读取该文件记录的命令，然后逐一执行命令的方式来进行数据恢复。
- Redis写回硬盘策略
    - Always，这个单词的意思是「总是」，所以它的意思是每次写操作命令执行完后，同步将 AOF 日志数据写回硬盘；
    - Everysec，这个单词的意思是「每秒」，所以它的意思是每次写操作命令执行完后，先将命令写入到 AOF 文件的内核缓冲区，然后每隔一秒将缓冲区里的内容写回到硬盘；
    - No，意味着不由 Redis 控制写回硬盘的时机，转交给操作系统控制写回的时机，也就是每次写操作命令执行完后，先将命令写入到 AOF 文件的内核缓冲区，再由操作系统决定何时将缓冲区内容写回硬盘。

- 优点：首先，AOF提供了更好的数据安全性，因为它默认每接收到一个写命令就会追加到文件末尾。即使Redis服务器宕机，也只会丢失最后一次写入前的数据。其次，AOF支持多种同步策略（如everysec、always等），可以根据需要调整数据安全性和性能之间的平衡。同时，AOF文件在Redis启动时可以通过重写机制优化，减少文件体积，加快恢复速度。并且，即使文件发生损坏，AOF还提供了redis-check-aof工具来修复损坏的文件。
- 缺点：因为记录了每一个写操作，所以AOF文件通常比RDB文件更大，消耗更多的磁盘空间。并且，频繁的磁盘IO操作（尤其是同步策略设置为always时）可能会对Redis的写入性能造成一定影响。而且，当问个文件体积过大时，AOF会进行重写操作，AOF如果没有开启AOF重写或者重写频率较低，恢复过程可能较慢，因为它需要重放所有的操作命令。

RDB 快照

- 因为 AOF 日志记录的是操作命令，不是实际的数据，所以用 AOF 方法做故障恢复时，需要全量把日志都执行一遍，一旦 AOF 日志非常多，势必会造成 Redis 的恢复操作缓慢。为了解决这个问题，Redis 增加了 RDB 快照。
- Redis 提供了两个命令来生成 RDB 文件，分别是 save 和 bgsave，他们的区别就在于是否在「主线程」里执行：
  - 执行了 save 命令，就会在主线程生成 RDB 文件，由于和执行操作命令在同一个线程，所以如果写入 RDB 文件的时间太长，会阻塞主线程；
  - 执行了 bgsave 命令，会创建一个子进程来生成 RDB 文件，这样可以避免主线程的阻塞；
- 优点：RDB通过快照的形式保存某一时刻的数据状态，文件体积小，备份和恢复的速度非常快。并且，RDB是在主线程之外通过fork子进程来进行的，不会阻塞服务器处理命令请求，对Redis服务的性能影响较小。最后，由于是定期快照，RDB文件通常比AOF文件小得多。
- 缺点：RDB方式在两次快照之间，如果Redis服务器发生故障，这段时间的数据将会丢失。并且，如果在RDB创建快照到恢复期间有写操作，恢复后的数据可能与故障前的数据不完全一致

Redis缓存淘汰和过期删除

过期删除策略和内存淘汰策略有什么区别?

- 内存淘汰策略是在内存满了的时候，redis 会触发内存淘汰策略，来淘汰一些不必要的内存资源，以腾出空间，来保存新的内容
- 过期键删除策略是将已过期的键值对进行删除，Redis 采用的删除策略是惰性删除+定期删除。

介绍一下Redis 内存淘汰策略

- 在32位系统中，Redis最大内存默认是3GB，因为32位系统最多只有4GB内存

策略	描述	适用场景
noeviction	不淘汰，写入报错	适合只读缓存
volatile-lru	在设置了过期时间的键中，按 LRU 淘汰	节约有限空间，保留常用键
allkeys-lru	所有键都参与 LRU 淘汰	通用场景
volatile-lfu	在有过期时间的键中，按访问频率最少淘汰	频率优先
allkeys-lfu	所有键按 LFU 淘汰	高频数据优先保留
volatile-ttl	按过期时间最短（TTL 最小）淘汰	有明确过期时间的临时数据
volatile-random	在有过期时间的键中随机淘汰	较少用

策略	描述	适用场景
allkeys-random	所有键中随机淘汰	特殊场景测试

介绍一下Redis过期删除策略

惰性删除

- Redis 的惰性删除策略由 db.c 文件中的 expireIfNeeded 函数实现，代码如下：

```
int expireIfNeeded(redisDb *db, robj *key) {
    // 判断 key 是否过期
    if (!keyIsExpired(db,key)) return 0;
    ....
    /* 删除过期键 */
    ....
    // 如果 server.lazyfree_lazy_expire 为 1 表示异步删除，反之同步删除；
    return server.lazyfree_lazy_expire ? dbAsyncDelete(db,key) :
                                         dbSyncDelete(db,key);
}
```

- Redis 在访问或者修改 key 之前，都会调用 expireIfNeeded 函数对其进行检查，检查 key 是否过期：
  - 如果过期，则删除该 key，至于选择异步删除，还是选择同步删除，根据 lazyfree_lazy_expire 参数配置决定（Redis 4.0版本开始提供参数），然后返回 null 客户端；
  - 如果没有过期，不做任何处理，然后返回正常的键值对给客户端；

定期删除

- Redis 的定期删除是每隔一段时间「随机」从数据库中取出一定数量的 key 进行检查，并删除其中的过期key。
  - 默认每10秒检查一次，随机抽取一定数量的key进行过期检查
  - 随机抽查的数量默认是20
- 过程：
  - 从过期字典中随机抽取 20 个 key；
  - 检查这 20 个 key 是否过期，并删除已过期的 key；
  - 如果本轮检查的已过期 key 的数量，超过 5 个（20/4），也就是「已过期 key 的数量」占比「随机抽取 key 的数量」大于 25%，则继续重复步骤 1；如果已过期的 key 比例小于 25%，则停止继续删除过期 key，然后等待下一轮再检查。可以看到，定期删除是一个循环的流程。那 Redis 为了保证定期删除不会出现循环过度，导致线程卡死现象，为此增加了定期删除循环流程的时间上限，默认不会超过 25ms。

集群

Redis主从同步中的增量和完全同步怎么实现?

完全同步

- 完全同步发生在以下几种情况：
  - 初次同步：当一个从服务器（slave）首次连接到主服务器（master）时，会进行一次完全同步。
  - 从服务器数据丢失：如果从服务器数据由于某种原因（如断电）丢失，它会请求进行完全同步。
  - 主服务器数据发生变化：如果从服务器长时间未与主服务器同步，导致数据差异太大，也可能触发完全同步。

- 主从服务器间的第一次同步的过程可分为三个阶段：
  - 第一阶段是建立链接、协商同步；
  - 第二阶段是主服务器同步数据给从服务器；
  - 第三阶段是主服务器发送新写操作命令给从服务器。
- 过程：
  - 从服务器发送SYNC命令：从服务器向主服务器发送SYNC命令，请求开始同步。
  - 主服务器生成RDB快照：接收到SYNC命令后，主服务器会保存当前数据集的状态到一个临时文件，这个过程称为RDB（Redis Database）快照。
  - 传输RDB文件：主服务器将生成的RDB文件发送给从服务器。
  - 从服务器接收并应用RDB文件：从服务器接收RDB文件后，会清空当前的数据集，并载入RDB文件中的数据。
  - 主服务器记录写命令：在RDB文件生成和传输期间，主服务器会记录所有接收到的写命令到replication backlog buffer。
  - 传输写命令：一旦RDB文件传输完成，主服务器会将replication backlog buffer中的命令发送给从服务器，从服务器会执行这些命令，以保证数据的一致性。

增量同步

- 量同步允许从服务器从断点处继续同步，而不是每次都进行完全同步。它基于PSYNC命令，使用了运行ID（run ID）和复制偏移量（offset）的概念。
- 过程：
  - 从节点请求恢复
    - 断线重连时，从节点向主节点发送 PSYNC {runid} {offset}
    - runid：主节点的运行 ID（上次全量同步时获得）
    - offset：从节点当前同步到的字节偏移量
  - 主节点检查条件
    - 比对 runid 是否一致（同一个主节点实例）
    - 检查该 offset 是否仍在 复制积压缓冲区
    - 复制积压缓冲区是一个固定大小的环形缓冲区，保存主节点最近一段时间的写命令
  - 满足条件 → 部分重同步（Partial Resync）
    - 主节点将从 offset 之后的命令推送给从节点
    - 从节点应用这些命令，状态追平主节点
    - 同步完成后继续正常的命令传播阶段
  - 不满足条件 → 全量同步（Full Resync）
    - 如果 runid 不一致，或 offset 之前的数据已被环形缓冲区覆盖，就要重新执行 RDB 全量同步

redis主从和集群可以保证数据一致性吗？

- redis 主从和集群在CAP理论都属于AP模型，即在面临网络分区时选择保证可用性和分区容忍性，而牺牲了强一致性。这意味着在网络分区的情况下，Redis主从复制和集群可以继续提供服务并保持可用，但可能会出现部分节点之间的数据不一致。

哨兵机制原理是什么?

- Redis 在 2.8 版本以后提供的哨兵（Sentinel）机制，它的作用是实现主从节点故障转移。它会监测主节点是否存活，如果发现主节点挂了，它就会选举一个从节点切换为主节点，并且把新主节点的相关信息通知给从节点和客户端。
- 哨兵其实是一个运行在特殊模式下的 Redis 进程，所以它也是一个节点。从“哨兵”这个名字也可以看得出来，它相当于是“观察者节点”，观察的对象是主从节点。
- 哨兵节点主要负责三件事情：监控、选主、通知。

哨兵机制的选主节点的算法介绍一下

- 当redis集群的主节点故障时，Sentinel集群将从剩余的从节点中选举一个新的主节点，有以下步骤：

故障节点主观下线

- Sentinel集群的每一个Sentinel节点会定时对redis集群的所有节点发心跳包检测节点是否正常。如果一个节点在down-after-milliseconds时间内没有回复Sentinel节点的心跳包，则该redis节点被该Sentinel节点主观下线。

故障节点客观下线

- 当节点被一个Sentinel节点记为主观下线时，并不意味着该节点肯定故障了，还需要Sentinel集群的其他Sentinel节点共同判断为主观下线才行。
- 该Sentinel节点会询问其他Sentinel节点，如果Sentinel集群中超过quorum数量的Sentinel节点认为该redis节点主观下线，则该redis客观下线。
- quorum在设置主节点时设置
- 如果客观下线的redis节点是从节点或者是Sentinel节点，则操作到此为止，没有后续的操作了；如果客观下线的redis节点为主节点，则开始故障转移，从从节点中选举一个节点升级为主节点。

Sentinel集群选举Leader

- 每一个Sentinel节点都可以成为Leader，当一个Sentinel节点确认redis集群的主节点主观下线后，会请求其他Sentinel节点要求将自己选举为Leader。被请求的Sentinel节点如果没有同意过其他Sentinel节点的选举请求，则同意该请求(选举票数+1)，否则不同意。
- 如果一个Sentinel节点获得的选举票数达到Leader最低票数(quorum和Sentinel节点数/2+1的最大值)，则该Sentinel节点选举为Leader；否则重新进行选举。

Sentinel Leader决定新主节点

- 过滤故障的节点
- 选择优先级slave-priority最大的从节点作为主节点，如不存在则继续
- 选择复制偏移量（数据写入量的字节，记录写了多少数据。主服务器会把偏移量同步给从服务器，当主从的偏移量一致，则数据是完全同步）最大的从节点作为主节点，如不存在则继续
- 选择runid（redis每次启动的时候生成随机的runid作为redis的标识）最小的从节点作为主节点

Redis集群的模式了解吗 优缺点了解吗

Redis集群的模式

- 当 Redis 缓存数据量大到一台服务器无法缓存时，就需要使用 Redis 切片集群（Redis Cluster ）方案，它将数据分布在不同的服务器上，以此来降低系统对单主节点的依赖，从而提高 Redis 服务的读写性能。
- Redis Cluster 方案采用哈希槽（Hash Slot），来处理数据和节点之间的映射关系。在 Redis Cluster 方案中，一个切片集群共有 16384 个哈希槽，这些哈希槽类似于数据分区，每个键值对都会根据它的 key，被映射到一个哈希槽中，具体执行过程分为两大步：
  - 根据键值对的 key，按照 CRC16 算法计算一个 16 bit 的值。
  - 再用 16bit 值对 16384 取模，得到 0~16383 范围内的模数，每个模数代表一个相应编号的哈希槽。
- 接下来的问题就是，这些哈希槽怎么被映射到具体的 Redis 节点上的呢？有两种方案：
  - 平均分配： 在使用 cluster create 命令创建 Redis 集群时，Redis 会自动把所有哈希槽平均分布到集群节点上。比如集群中有 9 个节点，则每个节点上槽的个数为 16384/9 个。
  - 手动分配： 可以使用 cluster meet 命令手动建立节点间的连接，组成集群，再使用 cluster addslots 命令，指定每个节点上的哈希槽个数。

优点

- 高可用性：Redis集群最主要的优点是提供了高可用性，节点之间采用主从复制机制，可以保证数据的持久性和容错能力，哪怕其中一个节点挂掉，整个集群还可以继续工作。
- 高性能：Redis集群采用分片技术，将数据分散到多个节点，从而提高读写性能。当业务访问量大到单机Redis无法满足时，可以通过添加节点来增加集群的吞吐量。
- 扩展性好：Redis集群的扩展性非常好，可以根据实际需求动态增加或减少节点，从而实现可扩展性。集群模式中的某些节点还可以作为代理节点，自动转发请求，增加数据模式的灵活度和可定制性。

缺点

- **部署和维护较复杂：**Redis集群的部署和维护需要考虑到分片规则、节点的布置、主从配置以及故障处理等多个方面，需要较强的技术支持，增加了节点异常处理的复杂性和成本。
- **集群同步问题：**当某些节点失败或者网络出故障，集群中数据同步的问题也会出现。数据同步的复杂度和工作量随着节点的增加而增加，同步时间也较长，导致一定的读写延迟。
- **数据分片限制：**Redis集群的数据分片也限制了一些功能的实现，如在一个key上修改多次，可能会因为该key所在的节点位置变化而失败。此外，由于将数据分散存储到各个节点，某些操作不能跨节点实现，不同节点之间的一些操作需要额外注意。

为什么redis比mysql要快?

- 内存存储：Redis 是基于内存存储的 NoSQL 数据库，而 MySQL 是基于磁盘存储的关系型数据库。由于内存存储速度快，Redis 能够更快地读取和写入数据，而无需像 MySQL 那样频繁进行磁盘 I/O 操作。
- 简单数据结构：Redis 是基于键值对存储数据的，支持简单的数据结构（字符串、哈希、列表、集合、有序集合）。相比之下，MySQL 需要定义表结构、索引等复杂的关系型数据结构，因此在某些场景下 Redis 的数据操作更为简单高效，比如 Redis 用哈希表查询， 只需要O1 时间复杂度，而MySQL引擎的底层实现是B+Tree，时间复杂度是O(logn)
- 线程模型：Redis 采用单线程模型可以避免了多线程之间的竞争，省去了多线程切换带来的时间和性能上的开销，而且也不会导致死锁问题。

本地缓存VS.Redis

对比项	本地缓存（In-Memory Cache）	Redis缓存（分布式缓存）
存储位置	应用进程所在服务器的内存	独立的Redis服务（内存型数据库），可在本机或远程
访问速度	**极快**（内存直接读写，纳秒/微秒级）	很快，但需网络请求（微秒到毫秒级）
部署与维护	简单，无需额外服务	需要安装和维护Redis服务
数据共享	**无法跨进程/跨服务器共享**	天生支持分布式，多实例间共享数据
容量限制	受单机内存限制	可扩展，通过分片、集群方式扩容
一致性	多节点时数据一致性差（每台机器缓存不一样）	数据集中管理，一致性更好
常见场景	高频访问的小量数据、本机计算结果缓存	跨服务共享数据、大规模缓存、消息队列、分布式锁等

Redis除了缓存，还有哪些应用?

Redis实现消息队列

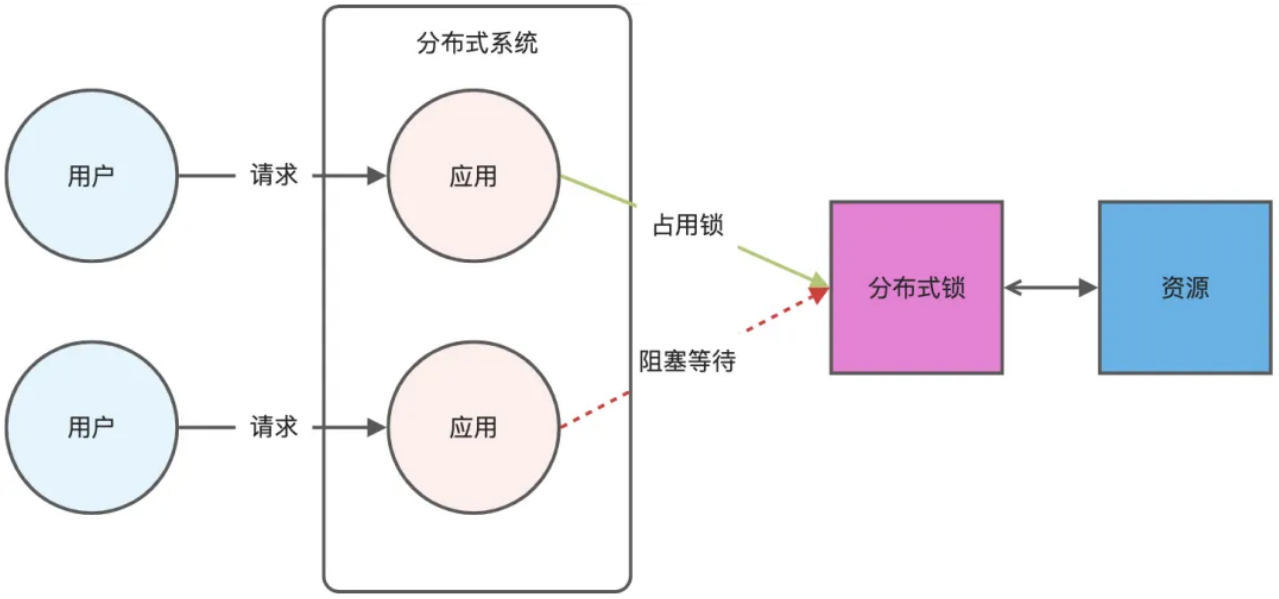
- **使用Pub/Sub模式：**Redis的Pub/Sub是一种基于发布/订阅的消息模式，任何客户端都可以订阅一个或多个频道，发布者可以向特定频道发送消息，所有订阅该频道的客户端都会收到此消息。该方式实现起来比较简单，发布者和订阅者完全解耦，支持模式匹配订阅。但是这种方式不支持消息持久化，消息发布后若无订阅者在线则会被丢弃；不保证消息的顺序和可靠性传输。
- 使用List结构：使用List的方式通常是使用LPUSH命令将消息推入一个列表，消费者使用BLPOP或BRPOP阻塞地从列表中取出消息（先进先出FIFO）。这种方式可以实现简单的任务队列。这种方式可以结合Redis的过期时间特性实现消息的TTL；通过Redis事务可以保证操作的原子性。但是需要客户端自己实现消息确认、重试等机制，相比专门的消息队列系统功能较弱。

Redis实现分布式锁

- set nx方式：Redis提供了几种方式来实现分布式锁，最常用的是基于SET命令的争抢锁机制。客户端可以使用SET resource_name lock_value NX PX milliseconds命令设置锁，其中NX表示只有当键不存在时才设置，PX指定锁的有效时间（毫秒）。如果设置成功，则认为客户端获得锁。客户端完成操作后，解锁的还需要先判断锁是不是自己，再进行删除，这里涉及到 2 个操作，为了保证这两个操作的原子性，可以用 lua 脚本来实现。
- **RedLock算法：**为了提高分布式锁的可靠性，Redis作者Antirez提出了RedLock算法，它基于多个独立的Redis实例来实现一个更安全的分布式锁。它的基本原理是客户端尝试在多数（大于半数）Redis实例上同时加锁，只有当在大多数实例上加锁成功时才认为获取锁成功。锁的超时时间应该远小于单个实例的超时时间，以避免死锁。该方式可以通过跨多个节点减少单点故障的影响，提高了锁的可用性和安全性。

Redis分布式锁的实现原理？什么场景下用到分布式锁？

- 分布式锁是用于分布式环境下并发控制的一种机制，用于控制某个资源在同一时刻只能被一个应用所使用。如下图所示：



加锁

- Redis 本身可以被多个客户端共享访问，正好就是一个共享存储系统，可以用来保存分布式锁，而且 Redis 的读写性能高，可以应对高并发的锁操作场景。Redis 的 SET 命令有个 NX 参数可以实现「key不存在才插入」，所以可以用它来实现分布式锁：
  - 如果 key 不存在，则显示插入成功，可以用来表示加锁成功；
  - 如果 key 存在，则会显示插入失败，可以用来表示加锁失败。
  - 基于 Redis 节点实现分布式锁时，对于加锁操作，我们需要满足三个条件。
- 加锁包括了读取锁变量、检查锁变量值和设置锁变量值三个操作，但需要以原子操作的方式完成，所以，我们使用 SET 命令带上 NX 选项来实现加锁；
- 锁变量需要设置过期时间，以免客户端拿到锁后发生异常，导致锁一直无法释放，所以，我们在 SET 命令执行时加上 EX/PX 选项，设置其过期时间；
- 锁变量的值需要能区分来自不同客户端的加锁操作，以免在释放锁时，出现误释放操作，所以，我们使用 SET 命令设置锁变量值时，每个客户端设置的值是一个唯一值，用于标识客户端；
- 满足这三个条件的分布式命令如下：

```
SET lock_key unique_value NX PX 10000
```

- lock_key 就是 key 键；
- unique_value 是客户端生成的唯一的标识，区分来自不同客户端的锁操作；
- NX 代表只在 lock_key 不存在时，才对 lock_key 进行设置操作；
- PX 10000 表示设置 lock_key 的过期时间为 10s，这是为了避免客户端发生异常而无法释放锁。

解锁

- 解锁的过程就是将 lock_key 键删除（del lock_key），但不能乱删，要保证执行操作的客户端就是加锁的客户端。所以，解锁的时候，我们要先判断锁的 unique_value 是否为加锁客户端，是的话，才将 lock_key 键删除。
- 可以看到，解锁是有两个操作，这时就需要 Lua 脚本来保证解锁的原子性，因为 Redis 在执行 Lua 脚本时，可以以原子性的方式执行，保证了锁释放操作的原子性。

```
// 释放锁时，先比较 unique_value 是否相等，避免锁的误释放
if redis.call("get",KEYS[1]) == ARGV[1] then
  return redis.call("del",KEYS[1])
else
  return 0
end
```

- 这样一来，就通过使用 SET 命令和 Lua 脚本在 Redis 单节点上完成了分布式锁的加锁和解锁。

大Key问题的缺点?

- 内存占用过高。大Key占用过多的内存空间，可能导致可用内存不足，从而触发内存淘汰策略。在极端情况下，可能导致内存耗尽，Redis实例崩溃，影响系统的稳定性。
- 性能下降。大Key会占用大量内存空间，导致内存碎片增加，进而影响Redis的性能。对于大Key的操作，如读取、写入、删除等，都会消耗更多的CPU时间和内存资源，进一步降低系统性能。
- 阻塞其他操作。某些对大Key的操作可能会导致Redis实例阻塞。例如，使用DEL命令删除一个大Key时，可能会导致Redis实例在一段时间内无法响应其他客户端请求，从而影响系统的响应时间和吞吐量。
- 网络拥塞。每次获取大key产生的网络流量较大，可能造成机器或局域网的带宽被打满，同时波及其他服务。例如：一个大key占用空间是1MB，每秒访问1000次，就有1000MB的流量。
- 主从同步延迟。当Redis实例配置了主从同步时，大Key可能导致主从同步延迟。由于大Key占用较多内存，同步过程中需要传输大量数据，这会导致主从之间的网络传输延迟增加，进而影响数据一致性。
- 数据倾斜。在Redis集群模式中，某个数据分片的内存使用率远超其他数据分片，无法使数据分片的内存资源达到均衡。另外也可能造成Redis内存达到maxmemory参数定义的上限导致重要的key被逐出，甚至引发内存溢出。

如何解决大Key问题

对大Key进行拆分存储

- 例如将user:123:profile拆分成`user:123:basic`、`user:123:contact`等

分页存储

- 例如order:list 存了用户的所有订单，几百万条，一旦 LRANGE 就阻塞很久。
- 拆分成：
  - order:list:1（第 1 页，存最近 100 条）
  - order:list:2（第 2 页）...

懒加载

- 例如article:content:999 里存了一篇几十 KB ~ MB 的文章全文，每次访问都直接从 Redis 取全量，浪费流量和 CPU。
- 初次访问时，只缓存常用的前几段文字到 Redis，其余段落不立即缓存
- 用户滚动到需要的段落时，再去数据库加载，并单独缓存到 article:content:999:part:{n}
- 下次再访问该段落时，直接命中缓存

什么是热Key

- 在 Redis 里，热 Key（Hot Key） 指的是某个 Key 被频繁访问，访问量远超其他 Key。热 Key 常见于爆款商品详情、库存、计数器、排行榜、用户会话与配置开关。
- 带来的影响：
  - CPU/网络压力集中：单个 Key 高并发访问，占用大量 CPU 和网络带宽
  - 缓存雪崩风险：热 Key 过期瞬间，海量请求直击数据库
  - 延迟波动：其他正常请求受到影响，整体响应变慢
  - 集群节点失衡：在分布式 Redis 中，热 Key 所在节点压力爆表

解决热Key问题

在Redis集群架构中对热Key进行复制

- 假设 item_stock:1001 是热门商品库存 Key，可拆分成：

```
item_stock:1001:0
item_stock:1001:1
...
```

- 下单时随机访问一个分片 Key，并在应用层汇总库存。

使用本地缓存

- 先在本地（应用服务器内存，例如 Guava Cache、Caffeine）缓存数据，减少 Redis 请求。
- 在分布式情况下需要保证一致性问题，多个服务实例需要保证看到同一个热key，京东开源的hotkey框架采用worker进行分布式计算统一识别，统一下发

使用读写分离架构

- **如果热Key的产生来自于读请求，您可以将实例改造成读写分离架构来降低每个数据分片的读请求压力，甚至可以不断地增加从节点。**但是读写分离架构在增加业务代码复杂度的同时，也会增加Redis集群架构复杂度。不仅要为多个从节点提供转发层（如Proxy，LVS等）来实现负载均衡，还要考虑从节点数量显著增加后带来故障率增加的问题。Redis集群架构变更会为监控、运维、故障处理带来了更大的挑战。

如何保证 redis 和 mysql 数据缓存一致性问题?

不一致场景

- 先修改数据库，在删除缓存，更新数据库成功，但删除缓存失败
- 先删除缓存，再更新数据库，如果删除缓存成功，未来得及更新数据库，此时用户再来一次请求，此时缓存会从数据库中取出旧值

双写事务

- 利用数据库事务和Redis事务放在同一个事务进行处理
- 优点是简单，缺点是无法保证完全一致性，适合小型系统简单读写，不适合高并发

延迟双删

- 数据库完成更新时，马上删除Redis缓存，延迟一段时间，再删一次
- 优点实现简单，缺点是延迟时间无法控制，可能会发生覆盖的风险

订阅更新机制

- 使用消息队列，数据库更新后向消息队列发送消息，Redis订阅这个消息队列更新缓存
- 优点是同步效果非常好，缺点是复杂性增加，需要维护消息队列

读写分离

- 把写操作全部操作数据库，再通过一个异步线程，更新Redis一致性
- 优点是缓存压力降低很多，适合大流量系统，缺点是项目开发复杂，要做好异步逻辑

Cannal订阅Binlog同步Redis

- 修改数据库时，数据库操作会写入bin log，使用Canal中间件订阅binlog变化，监听到变化后将更新的数据同步给redis

缓存雪崩、缓存击穿、缓存穿透

缓存雪崩

- 当大量缓存数据在同一时间过期（失效）或者 Redis 故障宕机时，如果此时有大量的用户请求，都无法在 Redis 中处理，于是全部请求都直接访问数据库
- 解决方案：
    - 避免给大量数据设置同一过期时间，使用随机数
    - 互斥锁：当业务线程在处理用户请求时，如果发现访问的数据不在 Redis 里，就加个互斥锁，保证同一时间内只有一个请求来构建缓存。未获取锁的请求则等待锁释放或返回默认值或者空值
    - 后台更新缓存：业务线程不再负责更新缓存，缓存也不设置有效期，而是让缓存“永久有效”，并将更新缓存的工作交由后台线程定时更新。

缓存击穿

- 如果缓存中的某个热点数据过期了，此时大量的请求访问了该热点数据，就无法从缓存中读取，直接访问数据库，数据库很容易就被高并发的请求冲垮
- 解决方案：
    - 互斥锁方案，保证同一时间只有一个业务线程更新缓存，未能获取互斥锁的请求，要么等待锁释放后重新读取缓存，要么就返回空值或者默认值。
    - 不给热点数据设置过期时间，由后台异步更新缓存，或者在热点数据准备要过期前，提前通知后台线程更新缓存以及重新设置过期时间；

缓存穿透

- 当用户访问的数据，既不在缓存中，也不在数据库中，导致请求在访问缓存时，发现缓存缺失，再去访问数据库时，发现数据库中也没有要访问的数据，没办法构建缓存数据，来服务后续的请求。那么当有大量这样的请求到来时，数据库的压力骤增，这就是缓存穿透的问题。
- 解决方案：

- 非法请求的限制：当有大量恶意请求访问不存在的数据的时候，也会发生缓存穿透，因此在 API 入口处我们要判断求请求参数是否合理，请求参数是否含有非法值、请求字段是否存在，如果判断出是恶意请求就直接返回错误，避免进一步访问缓存和数据库。
- 缓存空值或者默认值：当我们线上业务发现缓存穿透的现象时，可以针对查询的数据，在缓存中设置一个空值或者默认值，这样后续请求就可以从缓存中读取到空值或者默认值，返回给应用，而不会继续查询数据库。
- 布隆过滤器：我们可以在写入数据库数据时，使用布隆过滤器做个标记，然后在用户请求到来时，业务线程确认缓存失效后，可以通过查询布隆过滤器快速判断数据是否存在，如果不存在，就不用通过查询数据库来判断数据是否存在。即使发生了缓存穿透，大量请求只会查询 Redis 和布隆过滤器，而不会查询数据库，保证了数据库能正常运行，Redis 自身也是支持布隆过滤器的。

布隆过滤器原理介绍一下

- 布隆过滤器由「初始值都为 0 的位图数组」和「 N 个哈希函数」两部分组成。当我们在写入数据库数据时，在布隆过滤器里做个标记，这样下次查询数据是否在数据库时，只需要查询布隆过滤器，如果查询到数据没有被标记，说明不在数据库中。
- 布隆过滤器会通过 3 个操作完成标记：
  - 第一步，使用 N 个哈希函数分别对数据做哈希计算，得到 N 个哈希值；
  - 第二步，将第一步得到的 N 个哈希值对位图数组的长度取模，得到每个哈希值在位图数组的对应位置。
  - 第三步，将每个哈希值在位图数组的对应位置的值设置为 1；
- 举个例子，假设有一个位图数组长度为 8，哈希函数 3 个的布隆过滤器。
- 在数据库写入数据 x 后，把数据 x 标记在布隆过滤器时，数据 x 会被 3 个哈希函数分别计算出 3 个哈希值，然后在对这 3 个哈希值对 8 取模，假设取模的结果为 1、4、6，然后把位图数组的第 1、4、6 位置的值设置为 1。当应用要查询数据 x 是否数据库时，通过布隆过滤器只要查到位图数组的第 1、4、6 位置的值是否全为 1，只要有一个为 0，就认为数据 x 不在数据库中。
- 布隆过滤器由于是基于哈希函数实现查找的，高效查找的同时存在哈希冲突的可能性，比如数据 x 和数据 y 可能都落在第 1、4、6 位置，而事实上，可能数据库中并不存在数据 y，存在误判的情况。
- 所以，查询布隆过滤器说数据存在，并不一定证明数据库中存在这个数据，但是查询到数据不存在，数据库中一定就不存在这个数据。

如何设计秒杀场景处理高并发以及超卖现象？

使用数据库悲观锁

```
BEGIN;
SELECT * FROM goods WHERE goods_id = ? FOR UPDATE;  -- 阶段1：锁行
UPDATE goods SET stock = stock - 1
  WHERE goods_id = ? AND stock > 0;                 -- 阶段2：更新
COMMIT;                                             -- 释放锁
```

- 先对商品行加锁，再进行库存更新
- 悲观锁方案在高并发下会串行化对同一行的操作，安全但吞吐量会急剧下降。

使用数据库乐观锁

```
UPDATE goods
SET stock = stock - 1
WHERE goods_id = ? AND stock > 0;
```

- 这样让数据库自己做并发控制，受影响行数=0时再提示库存不足，可大幅减少阻塞。

使用分布式锁

- 同一个锁key，同一时间只能有一个客户端拿到锁，其他客户端会陷入无限的等待来尝试获取那个锁，只有获取到锁的客户端才能执行下面的业务逻辑。
- 这种方案的缺点是同一个商品在多用户同时下单的情况下，会基于分布式锁串行化处理，导致没法同时处理同一个商品的大量下单的请求。

利用分段式锁+分段缓存

- 把数据分成很多个段，每个段是一个单独的锁，所以多个线程过来并发修改数据的时候，可以并发的修改不同段的数据
- 假设场景：假如你现在商品有100个库存，在redis存放5个库存key，形如:

```
key1=goods-01,value=20;
key2=goods-02,value=20;
key3=goods-03，value=20
```

- 用户下单时对用户id进行%5计算，看落在哪个redis的key上，就去取哪个，这样每次就能够处理5个进程请求；或者使用轮询的方式
- 这种方案可以解决同一个商品在多用户同时下单的情况，但有个坑需要解决：当某段锁的库存不足，一定要实现自动释放锁然后换下一个分段库存再次尝试加锁处理，此种方案复杂比较高。

利用redis的incr、decr的原子性 + 异步队列

- 在系统初始化时，将商品的库存数量加载到redis缓存中
- 接收到秒杀请求时，在redis中使用lua脚本进行预减库存（利用redis decr的原子性），当redis中的库存不足时，直接返回秒杀失败，否则继续进行第3步；
- 将请求放入异步队列中，返回正在排队中；
- 服务端异步队列将请求出队（哪些请求可以出队，可以根据业务来判定，比如：判断对应用户是否已经秒杀过对应商品，防止重复秒杀），出队成功的请求可以生成秒杀订单，减少数据库库存（在扣减库存的sql如下，返回秒杀订单详情）

```
update goods set stock = stock - 1 where goods_id = ? and stock >0
```

- 用户在客户端申请秒杀请求后，进行轮询，查看是否秒杀成功，秒杀成功则进入秒杀订单详情，否则秒杀失败
- 缺点：由于是通过异步队列写入数据库中，可能存在数据不一致，其次引用多个组件复杂度比较高