# MySQL

## 索引

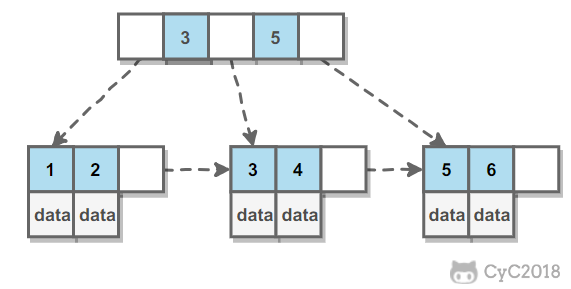
### B+Tree原理

#### 数据结构

B Tree指的是Balance Tree，也就是平衡树。平衡树是一颗查找树，并且所有的叶子节点都位于同一层。

B+Tree是基于B Tree和叶子节点顺序访问指针进行实现，它具有B Tree的平衡性，并且通过顺序访问指针来提高区间查询的性能。

在B+Tree中，一个点中的Key从左到右非递减排列，如果某个指针的左右相邻key分别是和，且不为null，则该指针指向节点的所有key大于等于且小于等于。



#### 操作

进行查找操作时，首先在根节点进行二分查找，找到一个key所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出key所对应的data。

插入删除操作会破坏平衡树的平衡性，因此在插入删除操作之后，需要对树进行一个分裂、合并、旋转等操作来维护平衡性。

#### 与红黑树的比较

红黑树等平衡树也可以用来实现索引，但是文件系统及数据库系统普遍采用B+ Tree作为索引结构，主要有以下两个原因：

* 更少的查找次数：平衡树查找操作的时间复杂度和树高h相关，，其中d为每个节点的出度。红黑树的出度为2，而B+ Tree的出度一般都非常大，所以红黑树的树高h很明显比B+ Tree大非常多，查找的次数也就更多。
* 利用磁盘预读的特性：为了减少磁盘I/O操作，磁盘往往不是严格按需读取，而是每次都会预读。预读过程中，磁盘进行顺序读取，顺序读取不需要进行磁盘寻道，并且只需要很短的磁盘旋转时间，速度会非常快。操作系统一般将内存和磁盘分割成固定大小的块，每一块成为一页，内存与磁盘以页为单位交换数据。数据库系统将索引的一个节点的大小设置为页的大小，使得一次I/O就能完全载入一个节点。并且可以利用预读特性，相邻的节点也能够被预先载入。

### MySQL索引

索引是在存储引擎层实现的，而不是在服务器层实现的，所以不同存储引擎具有不同的索引类型和实现。

#### B+Tree索引

B+Tree是大多数MySQL存储引擎的默认索引类型。

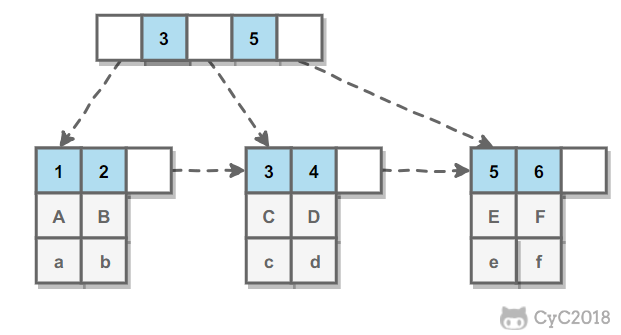
因为不再需要进行全表扫描，只需要对树进行搜索即可，所以查找速度快很多。

因为B+ Tree的有序性，所以除了用于查找，还可以用于排序和分组。

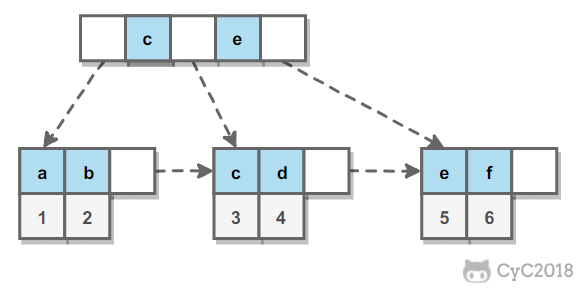
可以指定多个列作为索引列，多个索引列共同组成键。

适用于全键值、键值范围和键前缀查找，其中键前缀查找只适用于最左前缀查找。如果不是按照索引列的顺序进行查找，则无法使用索引。

InnoDB 的 B+Tree 索引分为主索引和辅助索引。主索引的叶子节点 data 域记录着完整的数据记录，这种索引方式被称为聚簇索引。因为无法把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引。



辅助索引的叶子节点的 data 域记录着主键的值，因此在使用辅助索引进行查找时，需要先查找到主键值，然后再到主索引中进行查找。



#### 哈希索引

哈希索引能够以O(1)时间进行查找，但是失去了有序性：

* 无法用于排序与分组
* 只支持精确查找，无法用于部分查找和范围查找

InnoDB 存储引擎有一个特殊的功能叫“自适应哈希索引”，当某个索引值被使用的非常频繁时，会在 B+Tree 索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，比如快速的哈希查找。

#### 全文索引

MyISAM 存储引擎支持全文索引，用于查找文本中的关键词，而不是直接比较是否相等。

查找条件使用 MATCH AGAINST，而不是普通的 WHERE。

全文索引使用倒排索引实现，它记录着关键词到其所在文档的映射。

InnoDB 存储引擎在 MySQL 5.6.4 版本中也开始支持全文索引。

#### 空间数据索引

MyISAM存储引擎支持空间数据索引（R-Tree），可以用于地理数据存储，空间数据索引会从所有维度来索引数据，可以有效地使用任意维度来进行组合查询。

必须使用GIS相关的函数来维护数据。

### 索引优化

#### 独立的列

在进行查询时，索引列不能是表达式的一部分，也不能是函数的参数，否则无法使用索引。

例如下面的查询不能使用actor\_id列的索引：

|  |
| --- |
| SELECT actor\_id FROM sakila.actor WHERE actor\_id + 1 = 5; |

#### 多列索引

在需要使用多个列作为条件进行查询时，使用多列索引比使用多个单列索引性能更好。例如下面语句中，最好把actor\_id和film\_id设置为多列索引。

|  |
| --- |
| SELECT film\_id, actor\_ id FROM sakila.film\_actor  WHERE actor\_id = 1 AND film\_id = 1; |

#### 索引列的顺序

让选择性最强的索引列放在前面。

索引的选择性是指：不重复的索引值和记录总数的比值。最大值为1，此时每个记录都有唯一的索引与其对应。选择性越高，每个记录的区分度越高，查询效率也越高。

例如下面显示的结果中customer\_id的选择性比staff\_id更高，因此最好把customer\_id列放在多列索引的前面。

|  |
| --- |
| SELECT COUNT(DISTINCT staff\_id)/COUNT(\*) AS staff\_id\_selectivity,  COUNT(DISTINCT customer\_id)/COUNT(\*) AS customer\_id\_selectivity,  COUNT(\*)  FROM payment;  结果：  staff\_id\_selectivity: 0.0001  customer\_id\_selectivity: 0.0373  COUNT(\*): 16049 |

#### 前缀索引

对于BLOB、TEXT和VARCHAR类型的列，必须使用前缀索引，只索引开始的部分字符。

前缀长度的选取需要根据索引选择性来确定。

#### 覆盖索引

索引包含所有需要查询的字段的值：

具有以下优点：

* 索引通常远远小于数据行的大小，只读取索引能大大减少数据访问量。
* 一些存储引擎（例如MyISAM）在内存中只缓存索引，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用（通常比较费时）。
* 对于InnoDB引擎，若辅助索引能够覆盖查询，则无需访问主索引。

### 索引的优点

* 大大减少了服务器需要扫描的数据行数；
* 帮助服务器避免进行排序和分组，以及避免创建临时表（B+Tree索引是有序的，可以用于ORDER BY和GROUP BY操作。临时表主要是在排序和分组过程中创建，不需要排序和分组，也就不需要创建临时表）。
* 将随机I/O变为顺序I/O（B+Tree索引是有序的，会将相邻的数据都存储在一起）。

### 索引的使用条件

* 对于非常小的表，大部分情况下简单的全表扫描比建立索引更高效。
* 对于中到大型的表，索引就非常有效。
* 但是对于特大型的表，建立和维护索引的代价将会随之增长。这种情况下，需要用到一种技术可以直接区分出需要查询的一组数据，而不是一条记录一条记录地匹配，例如可以使用分区技术。

## 查找性能优化

### 使用Explain进行分析

Explain用来分析SELECT查询语句，开发人员可以通过分析Explain结果来优化查询语句。

比较重要的字段有：

* select\_type：查询类型，有简单查询、联合查询、子查询等；
* key：使用的索引；
* rows：扫描的行数。

### 优化数据访问

#### 减少请求的数据量

* 只返回必要的列：最好不要使用SELECT \* 语句。
* 只返回必要的行：使用LIMIT语句来限制返回的数据。
* 缓存重复查询的数据：使用缓存可以避免在数据库中进行查询，特别在要查询的数据经常被重复查询时，缓存带来的查询性能提升将会是非常明显的。

#### 减少服务器端扫面的行数

最有效的方式是使用索引来覆盖查询。

### 重构查询方式

#### 切分大查询

一个大查询如果一次性执行的话，可能一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。

|  |
| --- |
| DELETE FROM messages WHERE create < DATE\_SUB(NOW(), INTERVAL 3 MONTH); |

|  |
| --- |
| rows\_affected = 0  do {  rows\_affected = do\_query(  "DELETE FROM messages WHERE create < DATE\_SUB(NOW(), INTERVAL 3 MONTH) LIMIT 10000")  } while rows\_affected > 0 |

#### 分解大连接查询

将一个大连接查询分解成对每一个表进行一次单表查询，然后在应用程序中进行关联，这样做的好处有：

* 让缓存更高效。对于连接查询，如果其中一个表发生变化，那么整个查询缓存就无法使用。儿分解后的多个查询，即时其中一个表发生变化，对其它表的查询缓存依然可以使用；
* 分解成多个单表查询，这些单表查询的缓存结果更可能被其它查询使用到，从而减少冗余记录的查询。
* 减少锁竞争；
* 在应用层进行连接，可以更容易对数据库进行拆分，从而更容易做到高性能和可伸缩。
* 查询本身效率也可能会有所提升。例如下面的例子中，使用IN()代替连接查询，可以让MySQL按照ID顺序进行查询，这可能比随机的连接要更高效。

|  |
| --- |
| SELECT \* FROM tab  JOIN tag\_post ON tag\_post.tag\_id=tag.id  JOIN post ON tag\_post.post\_id=post.id  WHERE tag.tag='mysql'; |

|  |
| --- |
| SELECT \* FROM tag WHERE tag='mysql';  SELECT \* FROM tag\_post WHERE tag\_id=1234;  SELECT \* FROM post WHERE post.id IN (123,456,567,9098,8904); |

## 存储引擎

### InnoDB

InnoDB是MySQL默认的事务型存储引擎，只有在需要它不支持的特性时，才考虑使用其它存储引擎。

实现了四个标准的隔离级别，默认级别是可重复读（REPEATABLE READ）。在可重复读隔离级别下，通过多版本并发控制（MVCC）+间隙锁（Next-Key Locking）防止幻影读。

主索引是聚簇索引，在索引中保存了数据，从而避免直接读取磁盘，因此对查询性能有很大的提升。

内部做了很多优化，包括从磁盘读取数据时采用的可预测性读、能够加快读操作并且自动创建的自适应哈希索引、能够加速插入操作的插入缓冲区等。

支持真正的在线热备份。其他存储引擎不支持在线热备份，要获取一致性视图需要停止对所有表的写入，而在读写混合场景中，停止写入可能也意味着停止读取。

### MyISAM

设计简单，数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，则依然可以使用它。

提供了大量的特性，包括压缩表、空间数据索引等。

不支持事务。

不支持行级锁，只能对整张表加锁，读取时会对需要读到的所有表加共享锁，写入时则对表加排它锁。但在表有读取操作的同时，也可以往表中插入新的记录，这被称为并发插入（CONCURRENT INSERT）。

可以手工或者自动执行检查和修复操作，但是和事务恢复以及崩溃恢复不同，可能导致一些数据丢失，而且修复操作是非常慢的。

如果指定了DELAY\_KEY\_WRITE选项，在每次修改执行完成时，不会立即将修改的索引数据写入磁盘，而是会写到内存中的键缓冲区，只有在清理键缓冲区或者关闭表的时候才会将对应的索引块写入磁盘。这种方式可以极大的提升写入性能，但是在数据库或者主机崩溃时会造成索引损坏，需要执行修复操作。

### 比较

* 事务：InnoDB是事务型的，可以使用Commit和Rollback语句。
* 并发：MyISAM只支持表级锁，而InnoDB还支持行级锁。
* 外键：InnoDB支持外键。
* 备份：InnoDB支持在线热备份。
* 崩溃恢复：MyISAM崩溃后发生损坏的概率比InnoDB高很多，而且恢复的速度也更慢。
* 其它特性：MyISAM支持压缩表和空间索引。

## 数据类型

### 整型

TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT分别使用8，16，24，32，64位存储空间，一般情况下越小的列越好。

INT（11）中的数字只是规定了交互工具显示字符的个数，对于存储和计算来说是没有意义的。

### 浮点数

FLOAT和DOUBLE为浮点类型，DECIMAL为高精度小数类型。CPU原生支持浮点运算，但是不支持DECIMAL类型的计算，因此DECIMAL的计算比浮点类型需要更高的代价。

FLOAT、DOUBLE和DECIMAL都可以指定列宽，例如DECIMAL（18, 9）表示总共18位，取9位存储小数部分，剩下9位存储整数部分。

### 字符串

主要有CHAR和VARCHAR两种类型：一种是定长的，一种是变长的。

VARCHAR这种变长类型能够节省空间，因为只需要存储必要的内容。但在执行UPDATE时可能会使行变得比原来长，当超出一个页所能容纳的大小时，就要执行额外的操作。MyISAM会将行拆成不同的片段存储，而InnoDB则需要分裂页来使行放进页内。

在进行存储和检索时，会保留VARCHAR末尾的空格，而会删除CHAR末尾的空格。

### 时间和日期

MySQL提供了两种相似的日期时间类型：DATETIME和TIMESTAMP。

#### DATETIME

能够保存从1000年到9999年的日期和时间，精度为秒，使用8字节的存储空间。

它与时区无关。

默认情况下，MySQL以一种可排序的、无歧义的格式显示DATETIME值，例如：“2008-01-16 22:37:08”，这是ANSI标准定义的日期和时间表示方法。

#### TIMESTAMP

和UNIX时间戳相同，保存从1970年1月1日午夜（格林威治时间）以来的秒数，使用4字节，只能表示从1970年到2038年。

它和时区有关，也就是说一个时间戳在不同的时区所代表的具体时间是不同的。

MySQL提供了FORM\_UNIXTIME()函数把UNIX时间戳转换为日期，并且提供了UNIX\_TIMESTAMP()函数把日期转换为UNIX时间戳。

默认情况下，如果插入时没有指定TIMESTAMP列的值，会将这个值设置为当前时间。

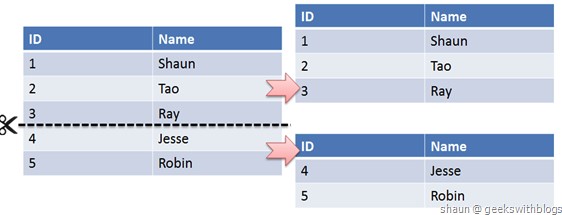
应该尽量使用TIMESTAMP，因为它比DATETIME空间效率更高。

## 切分

### 水平切分

水平切分又称为Sharing，它是将同一个表中的记录拆分到多个结构相同的表中。

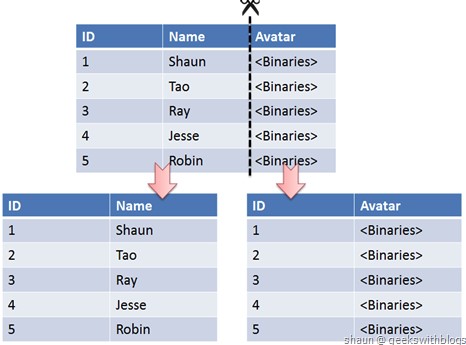
当一个表的数据不断增多时，Sharing是必然的选择，它可以将数据分布到集群的不同节点上，从而缓存单个数据库的压力。



### 垂直切分

垂直切分是将一张表按列切分成多个表，通常是按照列的关系密集程度进行切分，也可以利用垂直切分将经常被使用的列和不经常被使用的列切分到不同的表中。

在数据库的层面使用垂直切分将按照数据库中表的密集程度部署到不同的库中，例如将原来的电商数据库垂直切分成商品数据库、用户数据库等。



### Sharing策略

* 哈希取模：hash(key) % N;
* 范围：可以是ID范围也可以是时间范围；
* 映射表：使用单独的一个数据库来存储映射关系。

### Sharing存在的问题

#### 事务问题

使用分布式事务来解决，如：XA接口。

#### 连接

可以将原来的连接分解成多个单表查询，然后在用户程序中进行连接。

#### ID唯一性

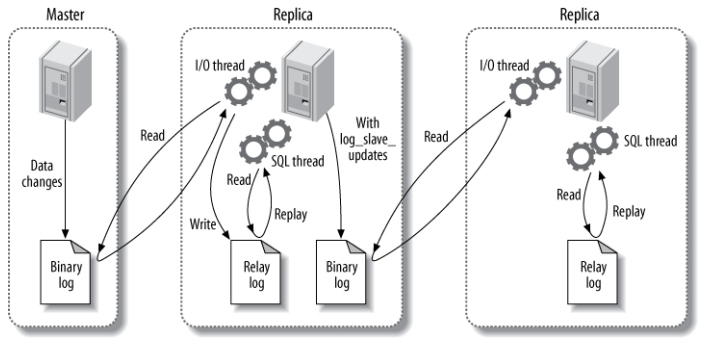
* 使用全局唯一ID（GUID）；
* 为每个分片指定一个ID范围；
* 分布式ID生成器（如Twitter的Snowflake算法）

## 复制

### 主从复制

主要涉及三个线程：binlog线程、I/O线程和SQL线程。

* binlog线程：负责将主服务器上的数据更改写入二进制日志（Binary Log）中。
* I/O线程：负责从主服务器上读取二进制日志，并写入从服务器的中继日志（Relay Log）。
* SQL线程：负责读取中继日志，解析出主服务器已经执行的数据更改并在从服务器中重放（Replay）。



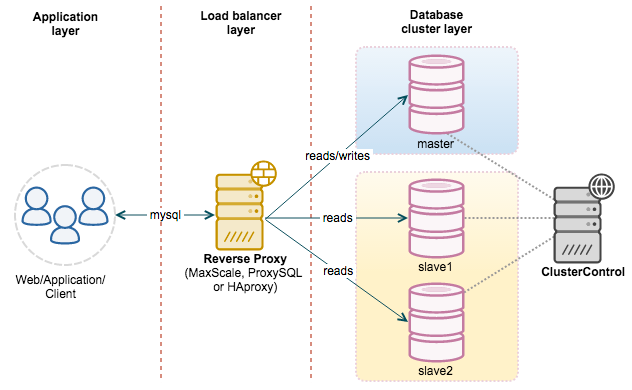
### 读写分离

主服务器处理写操作以及实时性要求比较高的读操作，而从服务器处理读操作。

读写分离能提高性能的原因在于：

* 主从服务器负责各自的读和写，极大程度缓解了锁的争用；
* 从服务器可以使用MyISAM，提升查询性能以及节约系统开销；
* 增加冗余，提高可用性。

读写分离常用代理方式来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。

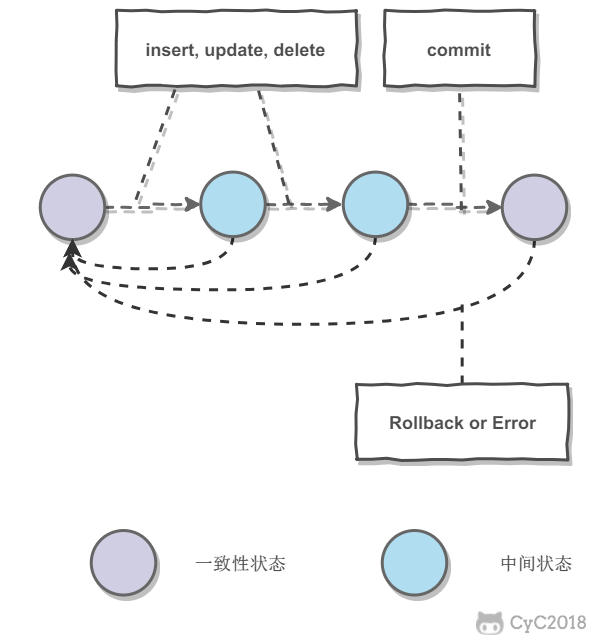


# 数据库系统原理

## 事务

### 概念

事务指的是满足ACID特性的一组操作，可以通过Commit提交一个事务，也可以使用Rollback进行回滚。



### ACID

#### 原子性（Atomicity）

事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚。

回滚可以用回滚日志来实现，回滚日志记录着事务所执行的修改操作，在回滚时反向执行这些操作即可。

#### 一致性（Consistency）

数据库在事务执行前后都保持一致性状态。在一致性状态下，所有事务对一个数据的读取结果都是相同的。

#### 隔离性（Isolation）

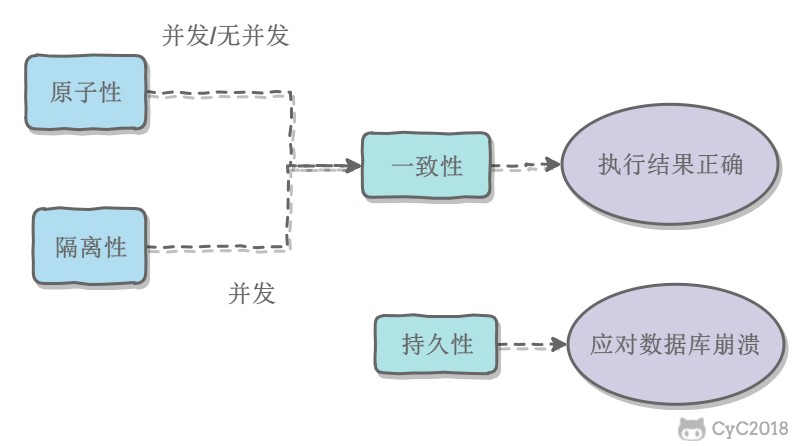
一个事务所做的修改在最终提交以前，对其它事务是不可见的。

#### 持久性（Durability）

一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失。使用重做日志来保证持久性。

事务的ACID特性概念简单，但不是很好理解，主要是因为这几个特性不是一种平级关系：

* 只有满足一致性，事务的执行结果才是正确的；
* 在无并发的情况下，事务串行执行，隔离性一定能够满足。此时，只要能满足原子性，就一定能够满足一致性。
* 在并发的情况下，多个事务并行执行，事务不仅要满足原子性，还要满足隔离性，才能满足一致性。
* 事务满足持久化是为了能应对数据库崩溃的情况。



### AUTOCOMMIT

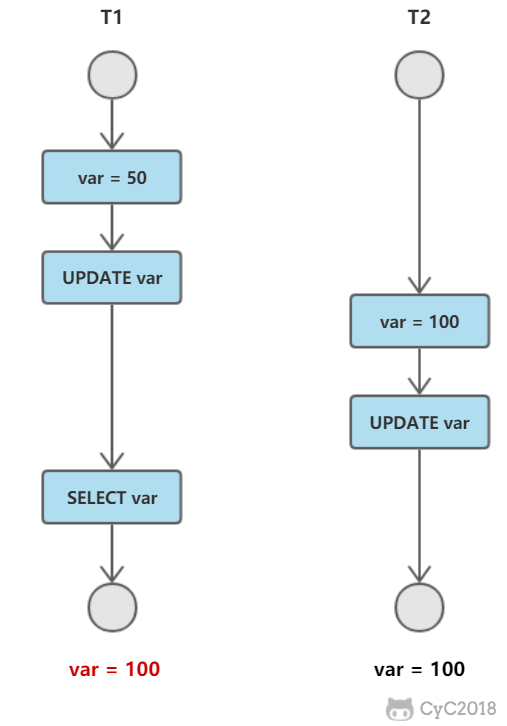
MySQL默认采用自动提交模式。也就是说，如果不显示使用START TRANSACTION语句来开始一个事务，那么每个查询都会被当做一个事务自动提交。

## 并发一致性问题

在并发环境下，事务的隔离性很难保证，因此会出现很多并发一致性问题。

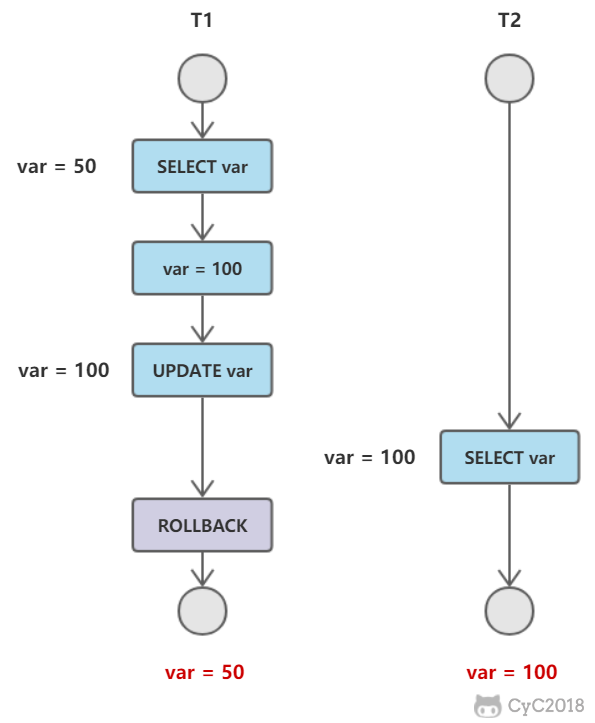
### 丢失修改

T1和T2两个事务都对一个数据进行修改，T1先修改，T2随后修改，T2的修改覆盖了T1的修改。



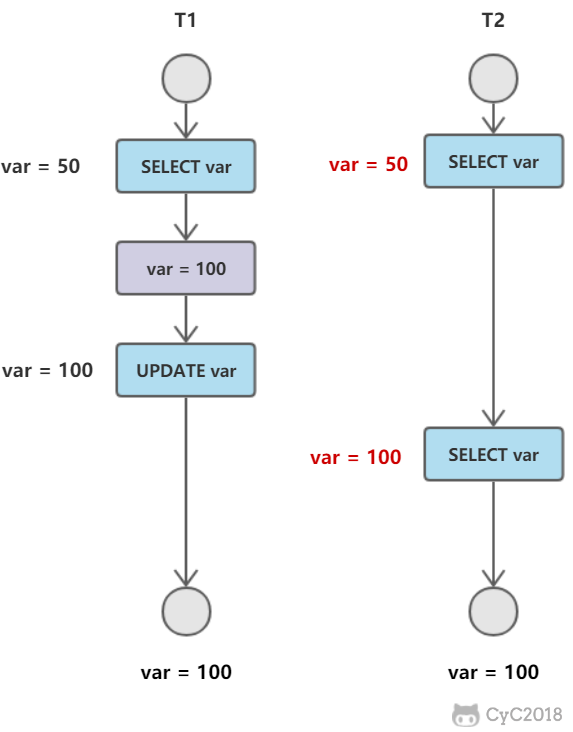
### 读脏数据

T1修改一个数据，T2随后读取这个数据，如果T1撤销了这次修改，那么T2读取的数据就是脏数据。



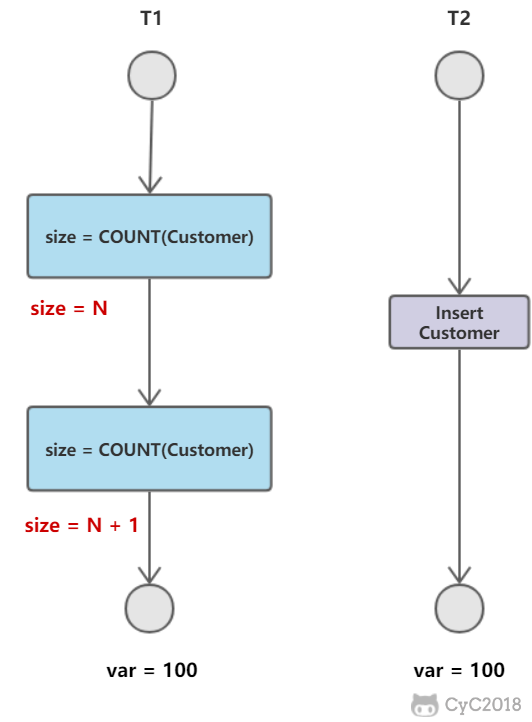
### 不可重复读

T2读取一个数据，T1对该数据做了修改。如果T2再次读取这个数据，此时读取的结果和第一次读取的结果不同。



### 幻影读

T1读取某个范围的数据，T2在这个范围内插入新的数据，T1再次读取这个范围的数据，此时读取的结果和第一次读取的结果不同。



产生并发不一致性问题主要原因是破坏了事务的隔离性，解决方法是通过并发控制来保证隔离性。并发控制可以通过封锁来实现，但是封锁操作需要用户自己控制，相当复杂。数据库管理系统提供了事务的隔离级别，让用户以一种更轻松的方式处理并发一致性问题。

## 封锁

### 封锁粒度

MySQL中提供了两种封锁粒度：行级锁和表级锁。

应该尽量只锁定需要修改的那部分数据，而不是所有的资源。锁定的数据量越少，发生锁争用的可能就越小，系统的并发程度就越高。

但是加锁需要消耗资源，锁的各种操作（包括获取锁、释放锁以及检查锁状态）都会增加系统开销。因此封锁粒度越小，系统开销越大。

在选择封锁粒度时，需要在锁开销和并发程度之间做一个权衡。

### 封锁类型

#### 读写锁

* 排它锁（Exclusive），简写为X锁，又称写锁；
* 共享锁（Shared），简写为S锁，又称读锁。

有以下两个规定：

* 一个事务对数据对象A加了X锁，就可以对A进行读取和更新。加锁期间其他事务不能对A加任何锁。
* 一个事务对数据对象A加了S锁，可以对A进行读取操作，但是不能进行更新操作。加锁期间其他事务能对A加S锁，但是不能加X锁。

锁的兼容关系：

|  |  |  |
| --- | --- | --- |
|  | X锁 | S锁 |
| X锁 | × | × |
| S锁 | × | √ |

#### 意向锁

使用意向锁（Intention Locks）可以更容易地支持多粒度封锁。

在存在行级锁和表级锁的情况下，事务T想要对表A加X锁，就需要先检测是否有其他事务对表A或者表A中的任意一行加了锁，那么就需要对表A的每一行都检测一次，这是非常耗时的。

意向锁在原来的X/S锁之上引入了IX/IS，IX/IS都是表锁，用来表示一个事务想要在表中的某个数据行上加X锁或S锁。有以下两个规定：

* 一个事务在获取某个数据行对象的S锁之前，必须先获得表的IS锁或者更强的锁。
* 一个事务在获取某个数据行对象的X锁之前，必须先获得表的IX锁。

各种锁的兼容关系如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | X锁 | IX锁 | S锁 | IS锁 |
| X锁 | × | × | × | × |
| IX锁 | × | √ | × | √ |
| S锁 | × | × | √ | √ |
| IS锁 | × | √ | √ | √ |

解释如下：

* 任意的IS/IX锁之间都是兼容的，因为他们只是表示想要对表加锁，而不是真正加锁。
* S锁只与S锁和IS锁兼容，也就是说事务T想要对数据行加S锁，其它事务可以已经获得对表或者表中的行的S锁。

### 封锁协议

#### 三级封锁协议

* 一级封锁协议

事务T要修改数据A时必须加X锁，直到T结束才释放锁。

可以解决丢失修改问题，因为不能同时有两个事务对同一个数据进行修改，那么事务的修改就不会被覆盖。

|  |  |
| --- | --- |
| T1 | T2 |
| Lock-X(A) |  |
| Read A=20 |  |
|  | Lock-X(A) |
|  | Wait |
| Write A=19 |  |
| Commit |  |
| Unlock-X(A) |  |
|  | Obtain |
|  | Read A=19 |
|  | Write A=21 |
|  | Commit |
|  | Unlock-X(A) |

* 二级封锁协议

在一级的基础上，要求读数据A时必须加S锁，读取完马上释放S锁。

可以解决读脏数据问题，因为如果一个事务在对数据A进行修改，根据1级封锁协议，会加X锁，那么就不能再加S锁了，也就是不会读入数据。

|  |  |
| --- | --- |
| T1 | T2 |
| Lock-X(A) |  |
| Read A=20 |  |
| Write A=19 |  |
|  | Lock-S(A) |
|  | Wait |
| Rollback |  |
| A=20 |  |
| Unlock-X(A) |  |
|  | Obtain |
|  | Read A=20 |
|  | Unlock-S(A) |
|  | Commit |

* 三级封锁协议

在二级的基础上，要求读取数据A时必须加S锁，直到事务结束了才能释放S锁。

可以解决不可重复读的问题，因为读A时，其它事务不能对A加X锁，从而避免了在读的期间数据发生改变。

|  |  |
| --- | --- |
| T1 | T2 |
| Lock-S(A) |  |
| Read A=20 |  |
|  | Lock-X(A) |
|  | Wait |
| Read A=20 |  |
| Commit |  |
| Unlock-S(A) |  |
|  | Obtain |
|  | Read A=20 |
|  | Write A=19 |
|  | Commit |
|  | Unlock-X(A) |

#### 两段锁协议

加锁和解锁分为两个阶段进行。

可串行化调度是指，通过并发控制，使得并发执行的事务结果与某个串行执行的事务结果相同。

事务遵循两段锁协议是保证可串行化调度的充分条件。例如以下操作满足两段锁协议，它是可串行化调度。

|  |
| --- |
| lock-x(A)...lock-s(B)...lock-s(C)...unlock(A)...unlock(C)...unlock(B) |

但不是必要条件，例如如下操作不满足两段锁协议，但是它还是可串行化调度。

|  |
| --- |
| lock-x(A)...unlock(A)...lock-s(B)...unlock(B)...lock-s(C)...unlock(C) |

### MySQL隐式与显示锁定

MySQL的InnoDB存储引擎采用两段锁协议，会根据隔离级别在需要的时候自动加锁，并且所有的锁都是在同一时刻被释放，这被称为隐式锁定。

InnoDB也可以使用特定的语句进行显示锁定：

|  |
| --- |
| SELECT ... LOCK In SHARE MODE;  SELECT ... FOR UPDATE; |

## 隔离级别

### 未提交读（READ UNCOMMITTED）

事务中的修改，即使没有提交，对其它事务也是可见的。

### 提交读（READ COMMITTED）

一个事务只能读取已经提交的事务所做的修改。换句话说，一个事务所做的修改在提交之前对其它事务是不可见的。

### 可重复读（REPEATABLE READ）

保证在同一个事务中多次读取同样数据的结果是一样的。

### 可串行化（SERIALIZABLE）

强制事务串行执行。

需要加锁实现，而其它隔离级别通常不需要。

|  |  |  |  |
| --- | --- | --- | --- |
| 隔离级别 | 脏读 | 不可重复读 | 幻影读 |
| 未提交读 | √ | √ | √ |
| 提交读 | × | √ | √ |
| 可重复读 | × | × | √ |
| 可串行化 | × | × | × |

## 多版本并发控制

多版本并发控制（Multi-Version Concurrency Control, MVCC）是MySQL的InnoDB存储引擎实现隔离级别的一种具体方式。用于实现提交读和可重复读这两种隔离级别。而未提交读隔离级别总是读取最新的数据行，无需使用MVCC。可串行化隔离级别需要对所有读取的行都加锁，单纯使用MVCC无法实现。

### 版本号

* 系统版本号：是一个递增的数字，每开始一个新的事务，系统版本号就会自动递增。
* 事务版本号：事务开始时的系统版本号。

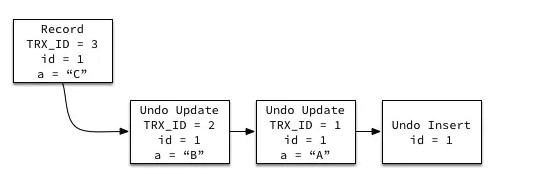
### 隐藏的列

MVCC在每行记录后面都保存着两个隐藏的列，用来存储两个版本号：

* 创建版本号：指示创建一个数据行的快照时的系统版本号；
* 删除版本号：如果该快照的删除版本号大于当前事务版本号表示该快照有效，否则表示该快照已经被删除了。

### Undo日志

MVCC使用到的快照存储在Undo日志中，该日志通过回滚指针把一个数据行（Record）的所有快照连接起来。



### 实现过程

以下实现过程针对可重复读隔离级别。

当开始一个事务时，该事务的版本号肯定大于当前所有数据行快照的创建版本号，理解这一点很关键。数据行快照的创建版本号是创建数据行快照时的系统版本号，系统版本号随着创建事务而递增，因此新创建一个事务时，这个事务的系统版本号比之前的系统版本号都大，也就是比所有数据行快照的创建版本号都大。

#### SELECT

多个事务必须读取到同一个数据行的快照，并且这个快照是距离现在最近的一个有效快照。但是也有例外，如果有一个事务正在修改该数据行，那么它可以读取事务本身所做的修改，而不用和其它事务的读取结果一致。

把没有对一个数据行做修改的事务称为T，T所要读取的数据行快照的创建版本号必须小于T的版本号，因为如果大于或者等于T的版本号，那么表示该数据行快照是其它事务的最新修改，因此不能去读取它。除此之外，T所要读取的数据行快照的删除版本号必须大于T的版本号，因为如果小于等于T的版本号，那么表示该数据行快照是已经被删除的，不应该去读取它。

#### INSERT

将当前系统版本号作为数据行快照的创建版本号。

#### DELETE

将当前系统版本号作为数据行快照的删除版本号。

#### UPDATE

将当前系统版本号作为更新前的数据行快照的删除版本号，并将当前系统版本号作为更新后的数据行快照的创建版本号。可以理解为先执行DELETE后执行INSERT。

### 快照读与当前读

#### 快照读

使用MVCC读取的是快照中的数据，这样可以减少加锁所带来的开销。

|  |
| --- |
| select \* from table ...; |

#### 当前读

读取的是最新的数据，需要加锁。以下第一个语句需要加S锁，其他都需要加X锁

|  |
| --- |
| select \* from table where ? lock in share mode;  select \* from table where ? for update;  insert;  update;  delete; |

## Next-Key Locks

Next-Key Locks是MySQL的InnoDB存储引擎的一种锁实现。

MVCC不能解决幻影读的问题，Next-Key Locks就是为了解决这个问题而存在。在可重复读（REPEATABLE READ）隔离级别下，使用MVCC+Next-Key Locks可以解决幻读问题。

### Record Locks

锁定一个记录上的索引，而不是记录本身。

如果表没有设置索引，InnoDB会自动在主键上穿件隐藏的聚簇索引，因此Record Locks依然可以使用。

### Gap Locks

锁定索引之间的间隙，但是不包含索引本身。例如当一个事务执行以下语句，其它事务就不能在tc中插入15。

|  |
| --- |
| SELECT c FROM t WHERE c BETWEEN 10 and 20 FOR UPDATE; |

### Next-Key Locks

它是Record Locks和Gap Locks的结合，不仅锁定一个记录上的索引，也锁定索引之间的间隙。例如一个索引包含一下的值：10，11，13，and 20，那么就需要锁定以下区间：

|  |
| --- |
| (-∞, 10]  (10, 11]  (11, 13]  (13, 20]  (20, +∞) |

## 关系数据库设计理论

### 函数依赖

记 A->B 表示 A 函数决定 B，也可以说 B 函数依赖于 A。

如果 {A1，A2，... ，An} 是关系的一个或多个属性的集合，该集合函数决定了关系的其它所有属性并且是最小的，那么该集合就称为键码。

对于 A->B，如果能找到 A 的真子集 A'，使得 A'-> B，那么 A->B 就是部分函数依赖，否则就是完全函数依赖。

对于 A->B，B->C，则 A->C 是一个传递函数依赖。

### 异常

以下的学生课程关系的函数依赖为{Sno, Cname} -> {Sname, Sdept, Mname, Grade}，键码为{Sno, Cname}。也就是说，确定学生和课程之后，就能确定其它信息。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Sno | Sname | Sdept | Mname | Cname | Grade |
| 1 | 学生-1 | 学院-1 | 院长-1 | 课程-1 | 90 |
| 2 | 学生-2 | 学院-2 | 院长-2 | 课程-2 | 80 |
| 3 | 学生-2 | 学院-2 | 院长-2 | 课程-1 | 100 |
| 4 | 学生-3 | 学院-2 | 院长-2 | 课程-2 | 95 |

不符合范式的关系，会产生很多异常，主要有以下四种异常：

* 冗余数据：例如“学生-2”出现了两次；
* 修改异常：修改了一个记录中的信息，但是另一个记录中相同的信息却没有被修改；
* 删除异常：删除一个信息，那么也会丢失其它信息。例如删除了“课程-1”需要删除第一行和第三行，那么“学生-1”的信息就会丢失。
* 插入异常：例如想要插入一个学生的信息，如果这个学生还没选课，那么就无法插入。

### 范式

范式理论是为了解决以上提到的四种异常。

高级别范式依赖于低级别的范式，1NF是最低级别的范式。

#### 第一范式（1NF）

属性不可分。

#### 第二范式（2NF）

每个非主属性完全函数依赖于键码。

可以通过分解来满足。

**分解前**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Sno | Sname | Sdept | Mname | Cname | Grade |
| 1 | 学生-1 | 学院-1 | 院长-1 | 课程-1 | 90 |
| 2 | 学生-2 | 学院-2 | 院长-2 | 课程-2 | 80 |
| 3 | 学生-2 | 学院-2 | 院长-2 | 课程-1 | 100 |
| 4 | 学生-3 | 学院-2 | 院长-2 | 课程-2 | 95 |

以上学生课程关系中，{Sno, Cname}为键码，有如下函数依赖：

* Sno -> Sname, Sdept
* Sdept -> Mname
* Sno, Cname -> Grade

Grade完全函数依赖于键码，他没有任何冗余数据，每个学生的每门课都有特定的成绩。

Sname, Sdept和Mname都部分依赖于键码，当一个学生选修了多门课时，这些数据就会出现多次，造成大量冗余数据。

**分解后**

关系-1

|  |  |  |  |
| --- | --- | --- | --- |
| Sno | Sname | Sdept | Mname |
| 1 | 学生-1 | 学院-1 | 院长-1 |
| 2 | 学生-2 | 学院-2 | 院长-2 |
| 3 | 学生-3 | 学院-2 | 院长-2 |

有以下函数依赖：

* Sno -> Sname, Sdept
* Sdept -> Mname

关系-2

|  |  |  |
| --- | --- | --- |
| Sno | Cname | Grade |
| 1 | 课程-1 | 90 |
| 2 | 课程-2 | 80 |
| 2 | 课程-1 | 100 |
| 3 | 课程-2 | 95 |

有以下函数依赖：

* Sno, Cname -> Grade

#### 第三范式（3NF）

非主属性不传递函数依赖于键码。

上面的 关系-1 中存在以下传递函数依赖：

* Sno -> Sdept -> Mname

可以进行以下分解：

关系-11

|  |  |  |
| --- | --- | --- |
| Sno | Sname | Sdept |
| 1 | 学生-1 | 学院-1 |
| 2 | 学生-2 | 学院-2 |
| 3 | 学生-3 | 学院-2 |

关系-12

|  |  |
| --- | --- |
| Sdept | Mname |
| 学院-1 | 院长-1 |
| 学院-2 | 院长-2 |

## ER图

Entity-Relationship，有三个组成部分：实体、属性、联系。

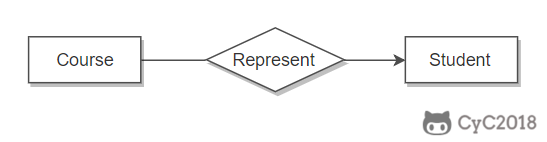
用来进行关系型数据库系统的概念设计。

### 实体的三种联系

包含一对一，一对多，多对多三种。

* 如果A到B是一对多关系，那么画个带箭头的线段指向B；
* 如果是一对一，画两个带箭头的线段；
* 如果是多对多，画两个不带箭头的线段。

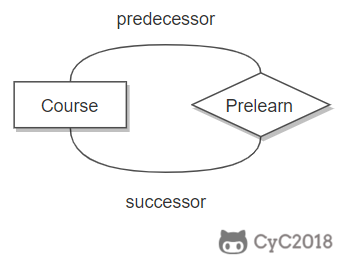
下图的Course和Student是一对多的关系。



### 表示出现多次的关系

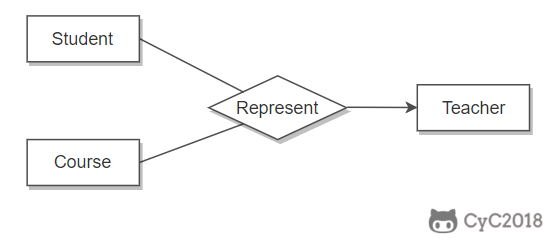
一个实体在联系出现几次，就要用几条线连接。

下图表示一个课程的先修关系，先修关系出现两个Course实体，第一个是先修课程，后一个是后修课程，因此需要用两条线来表示这种关系。



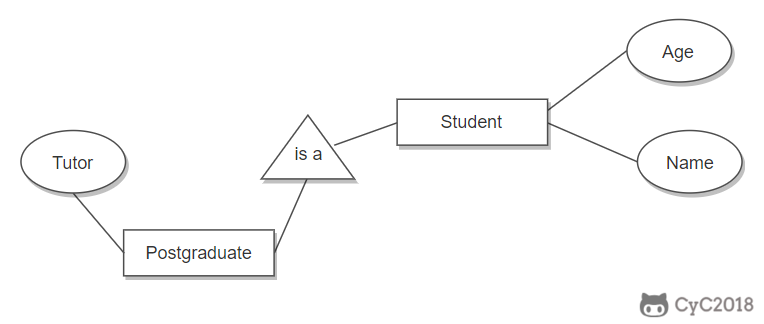
### 联系的多向性

虽然老师可以开设多门课，并且可以教授多名学生，但是对于特定的学生和课程，只有一个老师教授，这就构成了一个三元关系。



### 表示子类

用一个三角形和两条线来连接类和子类，与子类有关的属性和联系都连到子类上，而与父类和子类都有关的连到父类上。



# Redis

## 概述

Redis是速度非常快的非关系型（NoSQL）内存键值数据库，可以存储键和五种不同类型的值之间的映射。

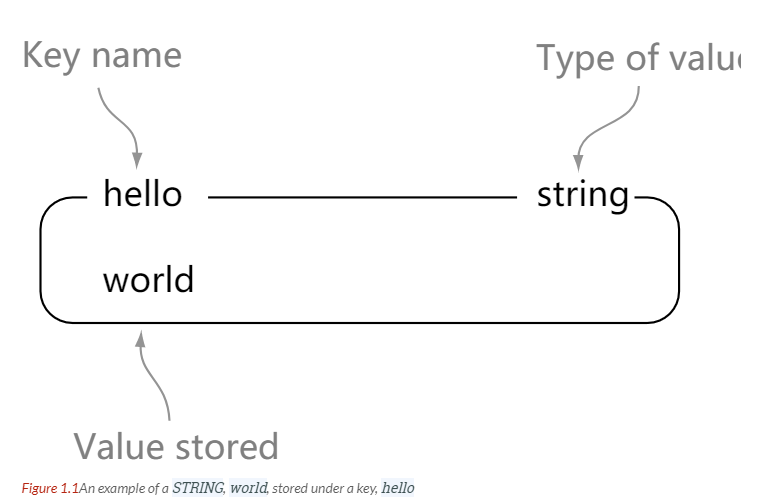
键的类型只能为字符串，值支持五种数据类型：字符串、列表、集合、散列表、有序集合。

Redis支持很多特性，例如将内存中的数据持久化到硬盘中，使用复制来扩展读性能，使用分片来扩展写性能。

## 数据类型

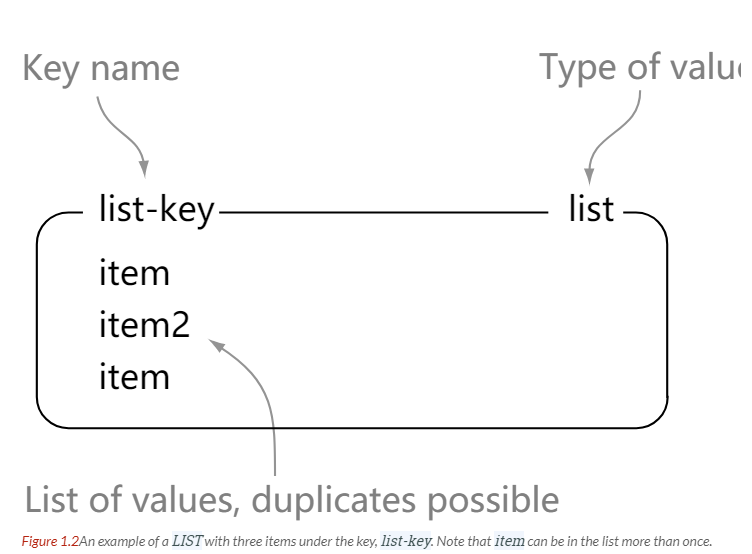
|  |  |  |
| --- | --- | --- |
| 数据类型 | 可以存储的值 | 操作 |
| STRING | 字符串、整数或者浮点数 | 对整个字符串或者字符串的其中一部分执行操作对整数和浮点数执行自增或者自减操作 |
| LIST | 列表 | 从两端压入或者弹出元素对单个或者多个元素进行修剪，只保留一个范围内的元素 |
| SET | 无序集合 | 添加、获取、移除单个元素检查一个元素是否存在于集合中计算交集、并集、差集从集合里面随机获取元素 |
| HASH | 包含键值对的无序散列表 | 添加、获取、移除单个键值对获取所有键值对检查某个键是否存在 |
| ZSET | 有序集合 | 添加、获取、删除元素根据分值范围或者成员来获取元素计算一个键的排名 |

### STRING



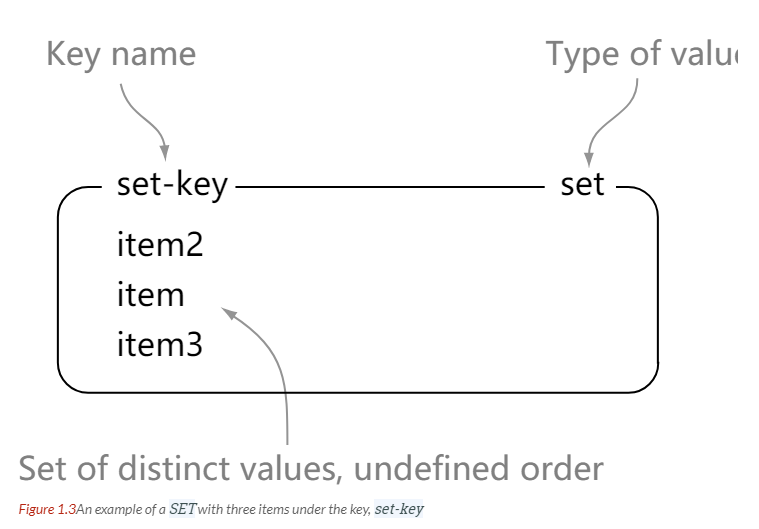
|  |
| --- |
| > set hello world  OK  > get hello  "world"  > del hello  (integer) 1  > get hello  (nil) |

### LIST



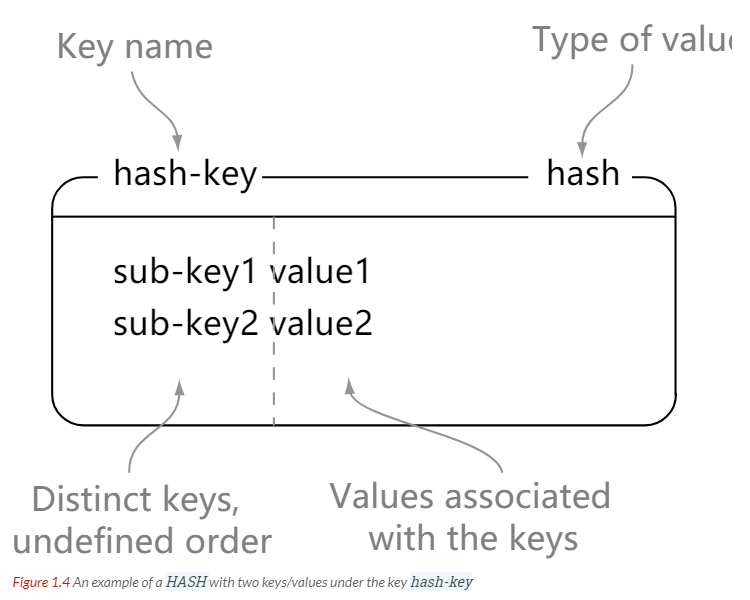
|  |
| --- |
| > rpush list-key item  (integer) 1  > rpush list-key item2  (integer) 2  > rpush list-key item  (integer) 3  > lrange list-key 0 -1  1) "item"  2) "item2"  3) "item"  > lindex list-key 1  "item2"  > lpop list-key  "item"  > lrange list-key 0 -1  1) "item2"  2) "item" |

### SET



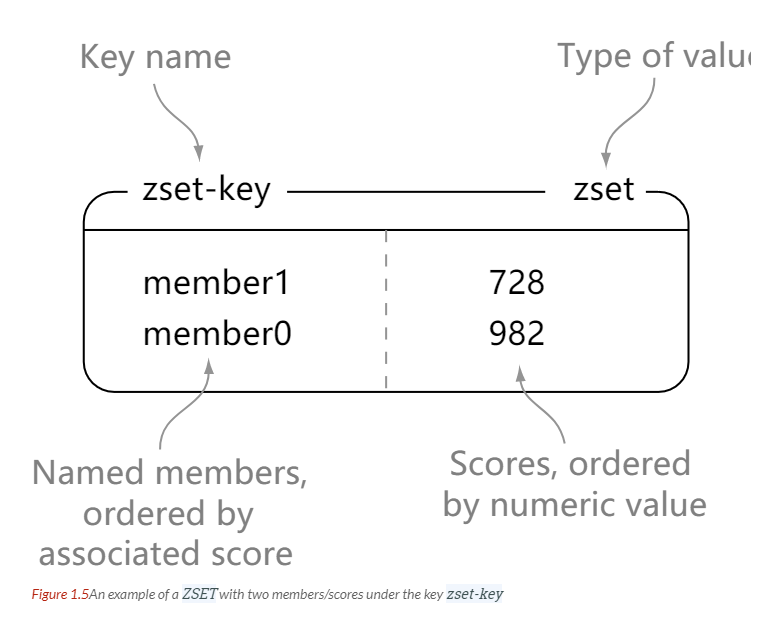
|  |
| --- |
| > sadd set-key item  (integer) 1  > sadd set-key item2  (integer) 1  > sadd set-key item3  (integer) 1  > sadd set-key item  (integer) 0  > smembers set-key  1) "item"  2) "item2"  3) "item3"  > sismember set-key item4  (integer) 0  > sismember set-key item  (integer) 1  > srem set-key item2  (integer) 1  > srem set-key item2  (integer) 0  > smembers set-key  1) "item"  2) "item3" |

### HASH



|  |
| --- |
| > hset hash-key sub-key1 value1  (integer) 1  > hset hash-key sub-key2 value2  (integer) 1  > hset hash-key sub-key1 value1  (integer) 0  > hgetall hash-key  1) "sub-key1"  2) "value1"  3) "sub-key2"  4) "value2"  > hdel hash-key sub-key2  (integer) 1  > hdel hash-key sub-key2  (integer) 0  > hget hash-key sub-key1  "value1"  > hgetall hash-key  1) "sub-key1"  2) "value1" |

### ZSET



|  |
| --- |
| > zadd zset-key 728 member1  (integer) 1  > zadd zset-key 982 member0  (integer) 1  > zadd zset-key 982 member0  (integer) 0  > zrange zset-key 0 -1 withscores  1) "member1"  2) "728"  3) "member0"  4) "982"  > zrangebyscore zset-key 0 800 withscores  1) "member1"  2) "728"  > zrem zset-key member1  (integer) 1  > zrem zset-key member1  (integer) 0  > zrange zset-key 0 -1 withscores  1) "member0"  2) "982" |

## 数据结构

### 字典

dictht是一个散列表结构，使用拉链法解决哈希冲突。

|  |
| --- |
| /\* This is our hash table structure. Every dictionary has two of this as we  \* implement incremental rehashing, for the old to the new table. \*/  typedef struct dictht {  dictEntry \*\*table;  unsigned long size;  unsigned long sizemask;  unsigned long used;  } dictht; |

|  |
| --- |
| typedef struct dictEntry {  void \*key;  union {  void \*val;  uint64\_t u64;  int64\_t s64;  double d;  } v;  struct dictEntry \*next;  } dictEntry; |

Redis的字典dict中包含两个哈希表dictht，这是为了方便进行rehash操作。在扩容时，将其中一个dictht上的键值对rehash到另一个dictht上面，完成之后释放空间并交换两个dictht的角色。

|  |
| --- |
| typedef struct dict {  dictType \*type;  void \*privdata;  dictht ht[2];  long rehashidx; /\* rehashing not in progress if rehashidx == -1 \*/  unsigned long iterators; /\* number of iterators currently running \*/  } dict; |

Rehash操作不是一次性完成，而是采用渐进方式，这是为了避免一次性执行过多的rehash操作给服务器带来过大的负担。

渐进式rehash通过记录dict的rehashidx完成，它从0开始，然后没执行一次rehash都会递增。例如在一次rehash中，要把dict[0]rehash到dict[1]，这一次会把dict[0]上table[rehashidx]的键值对rehash到dict[1]上，dict[0]的table[rehashidx]指向null，并令rehashidx++。

在rehash期间，每次对字典执行添加、删除、查找或者更新操作时，都会执行一次渐进式rehash。

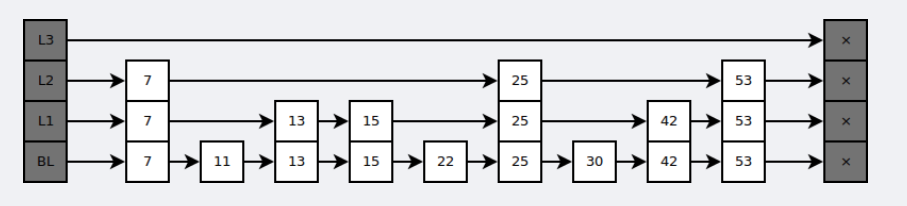
采用渐进式rehash会导致字典中的数据分散在两个dictht上，因此对字典的查找操作也需要到对应的dictht去执行。

|  |
| --- |
| /\* Performs N steps of incremental rehashing. Returns 1 if there are still  \* keys to move from the old to the new hash table, otherwise 0 is returned.  \*  \* Note that a rehashing step consists in moving a bucket (that may have more  \* than one key as we use chaining) from the old to the new hash table, however  \* since part of the hash table may be composed of empty spaces, it is not  \* guaranteed that this function will rehash even a single bucket, since it  \* will visit at max N\*10 empty buckets in total, otherwise the amount of  \* work it does would be unbound and the function may block for a long time. \*/  int dictRehash(dict \*d, int n) {  int empty\_visits = n \* 10; /\* Max number of empty buckets to visit. \*/  if (!dictIsRehashing(d)) return 0;  while (n-- && d->ht[0].used != 0) {  dictEntry \*de, \*nextde;  /\* Note that rehashidx can't overflow as we are sure there are more  \* elements because ht[0].used != 0 \*/  assert(d->ht[0].size > (unsigned long) d->rehashidx);  while (d->ht[0].table[d->rehashidx] == NULL) {  d->rehashidx++;  if (--empty\_visits == 0) return 1;  }  de = d->ht[0].table[d->rehashidx];  /\* Move all the keys in this bucket from the old to the new hash HT \*/  while (de) {  uint64\_t h;  nextde = de->next;  /\* Get the index in the new hash table \*/  h = dictHashKey(d, de->key) & d->ht[1].sizemask;  de->next = d->ht[1].table[h];  d->ht[1].table[h] = de;  d->ht[0].used--;  d->ht[1].used++;  de = nextde;  }  d->ht[0].table[d->rehashidx] = NULL;  d->rehashidx++;  }  /\* Check if we already rehashed the whole table... \*/  if (d->ht[0].used == 0) {  zfree(d->ht[0].table);  d->ht[0] = d->ht[1];  \_dictReset(&d->ht[1]);  d->rehashidx = -1;  return 0;  }  /\* More to rehash... \*/  return 1;  } |

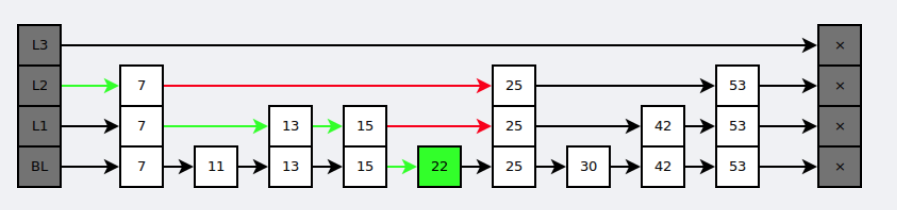
### 跳跃表

是有序集合的低层实现之一。

跳跃表是基于多指针有序链表实现的，可以看成多个有序链表。



在查找时，从上层指针开始查找，找到对应的区间之后再到下一层去找。下图演示了查找22 的过程。



与红黑树等平衡树相比，跳跃表具有以下优点：

* 插入速度非常快速，因为不需要进行旋转等操作来维护平衡性；
* 更容易实现；
* 支持无锁操作。

## 使用场景

### 计数器

可以对String进行自增自减运算，从而实现计数器功能。

Redis这种内存型数据库的读写性能非常高，很适合存储频繁读写的计数量。

### 缓存

将热点数据存放到内存中，设置内存的最大使用量以及淘汰策略来保证缓存的命中率。

### 查找表

例如DNS记录就很适合使用Redis进行存储

查找表和缓存类似，也是利用了Redis快速的查找特性。但是查找表的内容不能失效，而缓存的内容可以失效，因为缓存不作为可靠的数据来源。

### 消息队列

List是一个双向链表，可以通过lpush和rpop写入和读取消息

不过最好使用Kafka、RabbitMQ等消息中间件。

### 会话缓存

可以使用Redis来统一存储多台应用服务器的会话信息。

当应用服务器不在存储用户的会话信息，也就不再具有状态，一个用户可以请求任意一个应用服务器，从而更容易实现高可用性以及可伸缩性。

### 分布式锁实现

在分布式场景下，无法使用单机环境下的锁来对多个节点上的进程进行同步。

可以使用Redis自带的SETNX命令实现分布式锁，除此之外，还可以使用官方提供的RedLock分布式锁实现。

### 其它

Set可以实现交集，并集等操作，从而实现共同好友等功能。

ZSet可以实现有序性操作，从而实现排行榜等功能。

## Redis与Memcached

两者都是非关系型内存键值数据库，主要有以下不同：

### 数据类型

Memcached仅支持字符串类型，而Redis支持五种不同的数据类型，可以更灵活地解决问题。

### 数据持久化

Redis支持两种持久化策略：RDB快照和AOF日志，而Memcached不支持持久化。

### 分布式

Memcached不支持分布式，只能通过在客户端使用一致性哈希来实现分布式存储，这种方式在存储和查询时都需要先在客户端计算一次数据所在的节点。

Redis Cluster实现了分布式的支持。

### 内存管理机制

* 在Redis中，并不是所有数据都一直存储在内存中，可以将一些很久没用的value交换到磁盘，而Memcached的数据会一直在内存中。
* Memcached将内存分割成特定长度的块来存储数据，以完全解决内存碎片的问题，但是这种方式会使得内存的利用率不高，例如块的大小为128 bytes，只存储100 bytes的数据，那么剩下的28 bytes就浪费了。

## 键的过期时间

Redis可以为每个键设置过期时间，当键过期时，会自动删除该键。

对于散列表这种容器，只能为整个键设置过期时间（整个散列表），而不能为键里面的单个元素设置过期时间。

## 数据淘汰策略

可以设置内存最大使用量，当内存使用量超出时，会施行数据淘汰策略。

Redis具体有6种淘汰策略：

|  |  |
| --- | --- |
| 策略 | 描述 |
| Volatile-lru | 从已设置过期时间的数据集中挑选最近最少使用的数据淘汰 |
| Volatile-ttl | 从已设置过期时间的数据集中挑选将要过期的数据淘汰 |
| Volatile-random | 从已设置过期时间的数据集中任意选择数据淘汰 |
| Allkeys-lru | 从所有数据集中挑选最近最少使用的数据淘汰 |
| Allkeys-random | 从所有数据集中任意选择数据进行淘汰 |
| Noeviction | 禁止驱逐数据 |

作为内存数据库，出于对性能和内存消耗的考虑，Redis的淘汰算法实际实现上并非对所有key，而是抽样一小部分并且从中选出被淘汰的key。

使用Redis缓存数据时，为了提高缓存的命中率，需要保证缓存数据都是热点数据。可以将内存最大使用量设置为热点数据占用的内存量，然后启用allkeys-lru淘汰策略，将最近最少使用的数据淘汰。

Redis4.0引入了volatile-lfu和allkeys-lfu淘汰策略，LFU策略通过统计访问频率，将访问频率最少的键值对淘汰。

## 持久化

Redis是内存型数据库，为了保证数据在断电后不会丢失，需要将内存中的数据持久化到硬盘上。

### RDB持久化

将某个时间点的所有数据都存放到硬盘上。

可以将快照复制到其它服务器从而创建具有相同数据的服务器副本。

如果系统发生故障，将会丢失最后一次创建快照之后的数据。

如果数据量很大，保存快照的时间会很长。

### AOF持久化

将写命令添加到AOF文件（Append Only File）的末尾。

使用AOF持久化需要设置同步选项，从而确保写命令同步到磁盘文件上的时机。这是因为对文件进行写入并不会马上将内容同步到磁盘上，而是先存储到缓冲区，然后有操作系统决定什么时候同步到磁盘。有以下同步选项：

|  |  |
| --- | --- |
| 选项 | 同步频率 |
| Always | 每个写命令都同步 |
| Everysec | 每秒同步一次 |
| No | 让操作系统来决定何时同步 |

* Always选项会严重降低服务器的性能；
* Everysec选项比较合适，可以保证系统崩溃时只会丢失一秒左右的数据，并且Redis每秒执行一次同步对服务器性能几乎没有任何影响；
* No选项并不能给服务器性能带来多大的提升，而且也会增加系统崩溃时数据丢失的数量。

随着服务器写请求的增多，AOF文件会越来越大。Redis提供了一种将AOF重写的特性，能够去除AOF文件中的冗余写命令。

## 事务

一个事务包含了多个命令，服务器在执行事务期间，不会改去执行其它客户端的命令请求。

事务中的多个命令被一次性发送给服务器，而不是一条一条发送，这种方式被称为流水线，它可以减少客户端与服务器之间的网络通信次数从而提升性能。

Redis最简单的事务实现方式是使用MULTI和EXEC命令将事务操作包围起来。

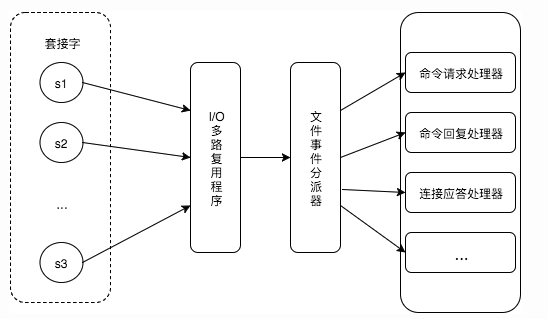
## 事件

Redis服务器是一个事件驱动程序。

### 文件事件

服务器通过套接字与客户端或者其它服务器进行通信，文件事件就是对套接字操作的抽象。

Redis基于Reactor模式开发了自己的网络事件处理器，使用I/O多路复用程序来同时监听多个套接字，并将到达的事件传送给文件事件分派器，分派器会根据套接字产生的事件类型调用相应的事件处理器。



### 时间事件