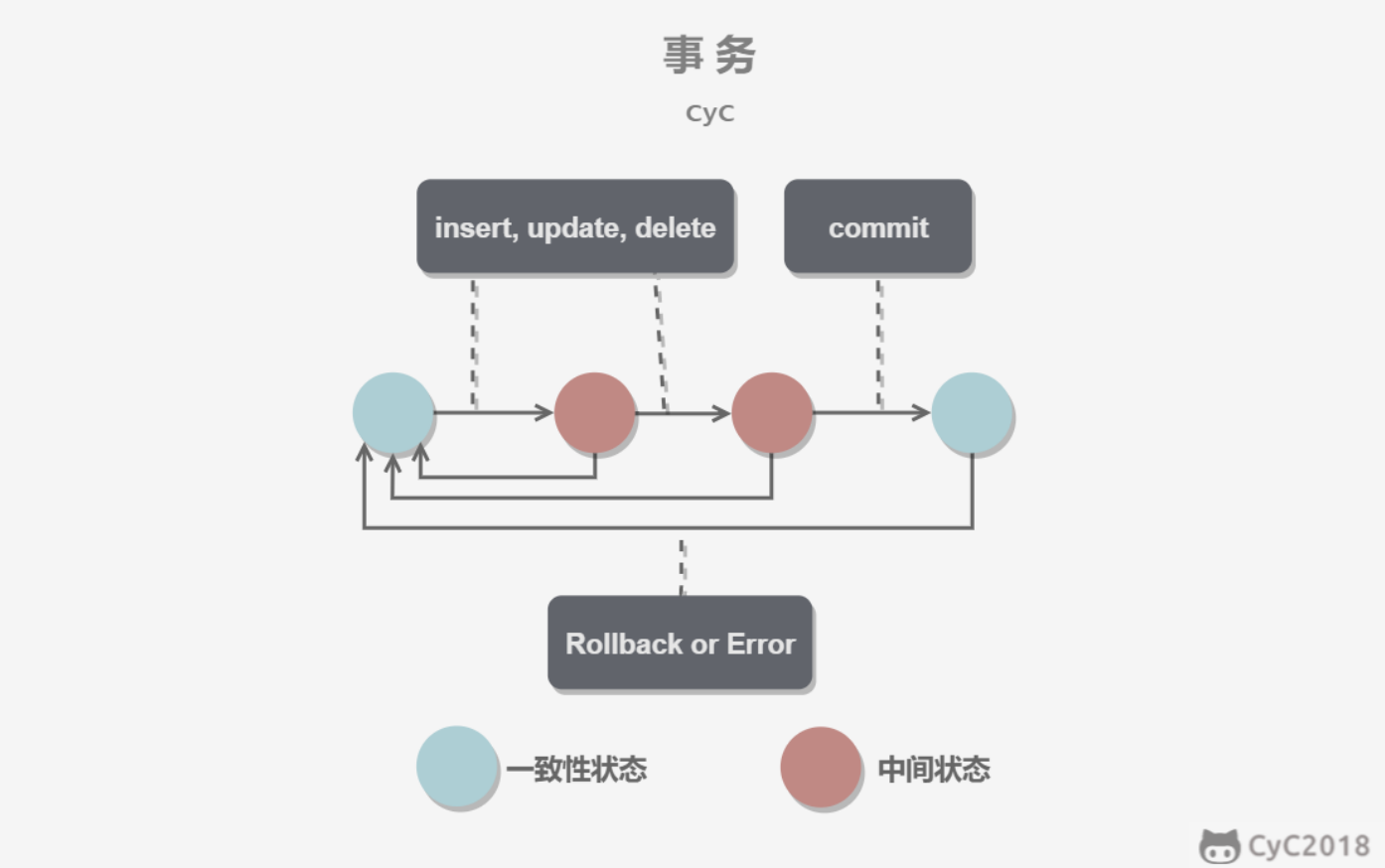
# 数据库

# 数据库系统原理

## 1.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

**1. 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（Undo Log）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修改操作即可。

**2. 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，所有事务对同一个数据的读取结果都是相同的。

**3. 隔离性（Isolation）**

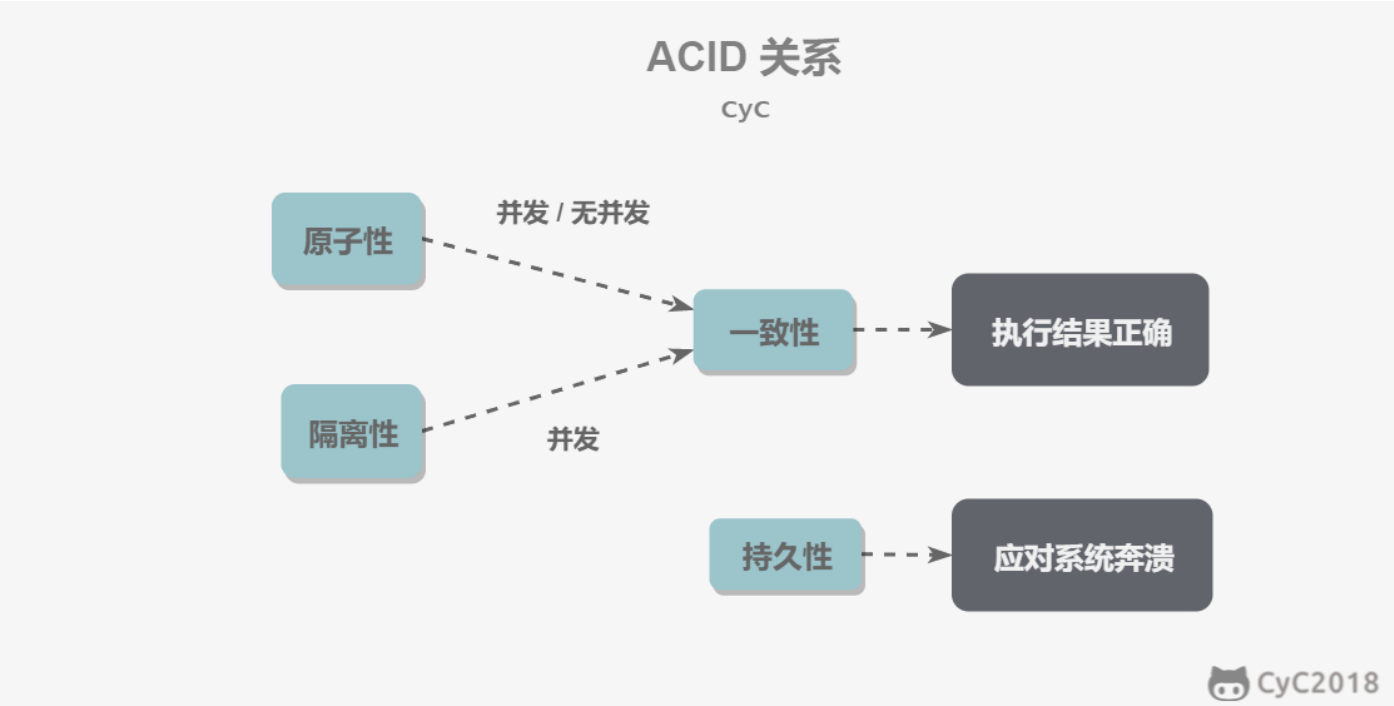
一个事务所做的修改在最终提交以前，对其它事务是不可见的。

**4. 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。系统发生奔溃可以用重做日志（Redo Log）进行恢复，从而实现持久性。与回滚日志记录数据的逻辑修改不同，重做日志记录的是数据页的物理修改。

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，还需要满足隔离性，才能满足一致性。
- 事务满足持久化是为了能应对系统崩溃的情况。




**AUTOCOMMIT**

MySQL 默认采用自动提交模式。也就是说，如果不显式使用 START TRANSACTION 语句来开始一个事务，那么每个查询操作都会被当做一个事务并自动提交。

## 2.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

丢失修改指一个事务的更新操作被另外一个事务的更新操作替换。

一般在现实生活中常会遇到。例如：T1 和 T2 两个事务都对一个数据进行修改，T1 先修改并提交生效，T2 随后修改，T2 的修改覆盖了 T1 的修改。

### 读脏数据

读脏数据指在不同的事务下，当前事务可以读到另外事务未提交的数据。

例如：T1 修改一个数据但未提交，T2 随后读取这个数据。如果 T1 撤销了这次修改，那么 T2 读取的数据是脏数据。

### 不可重复读

不可重复读指在一个事务内多次读取同一数据集合，在这一事务还未结束前，另一事务也访问了该同一数据集合并做了修改，由于第二个事务的修改，第一次事务的两次读取的数据可能不一致。

例如：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

幻读本质上也属于不可重复读的情况，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，此时读取的结果和和第一次读取的结果不同。

**产生并发不一致性问题的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。**

## 3.封锁

### 封锁粒度

MySQL 中提供了两种封锁粒度：行级锁以及表级锁。
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，系统开销就越大。
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

- 互斥锁（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。加锁期间其它事务不能对 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。加锁期间其它事务能对 A 加 S 锁，但是不能加 X 锁。

锁的兼容关系如下：

**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它事务对表 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：

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

通过引入意向锁，事务 T 想要对表 A 加 X 锁，只需要先检测是否有其它事务对表 A 加了 X/IX/S/IS 锁，如果加了就表示有其它事务正在使用这个表或者表中某一行的锁，因此事务 T 加 X 锁失败。

各种锁的兼容关系如下：


解释如下：

- 任意 IS/IX 锁之间都是兼容的，因为它们只表示想要对表加锁，而不是真正加锁；
- 这里兼容关系针对的是表级锁，而表级的 IX 锁和行级的 X 锁兼容，两个事务可以对两个数据行加 X 锁。（事务 T1 想要对数据行 R1 加 X 锁，事务 T2 想要对同一个表的数据行 R2 加 X 锁，两个事务都需要对该表加 IX 锁，但是 IX 锁是兼容的，并且 IX 锁与行级的 X 锁也是兼容的，因此两个事务都能加锁成功，对同一个表中的两个数据行做修改。）

## 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

### 二级封锁协议

在一级的基础上，要求读取数据 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。

### 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。

### 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。串行执行的事务互不干扰，不会出现并发一致性问题。

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，例如以下操作不满足两段锁协议，但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;
```

## 4.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一数据的结果是一样的。

### 可串行化（SERIALIZABLE）

强制事务串行执行，这样多个事务互不干扰，不会出现并发一致性问题。

该隔离级别需要加锁实现，因为要使用加锁机制保证同一时间只有一个事务执行，也就是保证事务串行执行。

## 5.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，要求很低，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。

### 基本思想

在封锁一节中提到，加锁能解决多个事务同时执行时出现的并发一致性问题。在实际场景中读操作往往多于写操作，因此又引入了读写锁来避免不必要的加锁操作，例如读和读没有互斥关系。读写锁中读和写操作仍然是互斥的，而 MVCC 利用了多版本的思想，写操作更新最新的版本快照，而读操作去读旧版本快照，没有互斥关系，这一点和 CopyOnWrite 类似。

在 MVCC 中事务的修改操作（DELETE、INSERT、UPDATE）会为数据行新增一个版本快照。

脏读和不可重复读最根本的原因是事务读取到其它事务未提交的修改。在事务进行读取操作时，为了解决脏读和不可重复读问题，MVCC 规定只能读取已经提交的快照。当然一个事务可以读取自身未提交的快照，这不算是脏读。

### 版本号

- 系统版本号 SYS_ID：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 TRX_ID ：事务开始时的系统版本号。

### Undo 日志

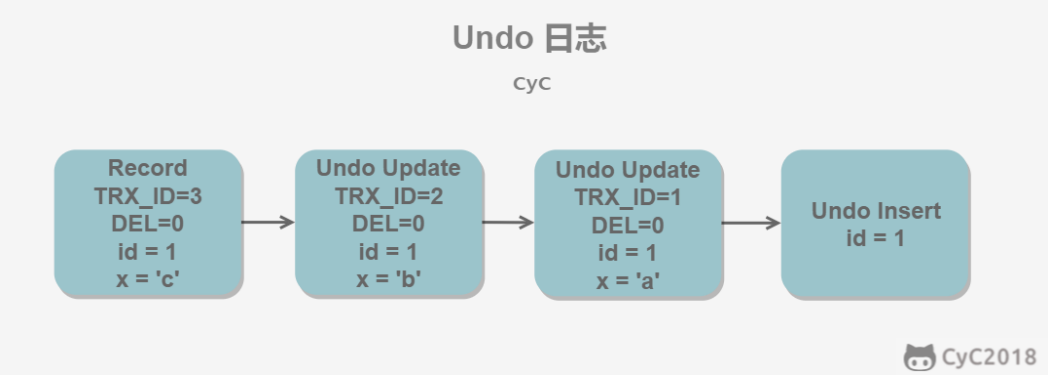
MVCC 的多版本指的是多个版本的快照，快照存储在 Undo 日志中，该日志通过回滚指针 ROLL_PTR 把一个数据行的所有快照连接起来。

例如在 MySQL 创建一个表 t，包含主键 id 和一个字段 x。我们先插入一个数据行，然后对该数据行执行两次更新操作。

```
INSERT INTO t(id, x) VALUES(1, "a");
UPDATE t SET x="b" WHERE id=1;
UPDATE t SET x="c" WHERE id=1;
```

因为没有使用 START TRANSACTION 将上面的操作当成一个事务来执行，根据 MySQL 的 AUTOCOMMIT 机制，每个操作都会被当成一个事务来执行，所以上面的操作总共涉及到三个事务。快照中除了记录事务版本号 TRX_ID 和操作之外，还记录了一个 bit 的 DEL 字段，用于标记是否被删除。

INSERT、UPDATE、DELETE 操作会创建一个日志，并将事务版本号 TRX_ID 写入。DELETE 可以看成是一个特殊的 UPDATE，还会额外将 DEL 字段设置为 1。




**ReadView**

MVCC 维护了一个 ReadView 结构，主要包含了当前系统未提交的事务列表 TRX_IDs {TRX_ID_1, TRX_ID_2, ...}，还有该列表的最小值 TRX_ID_MIN 和 TRX_ID_MAX。

在进行 SELECT 操作时，根据数据行快照的 TRX_ID 与 TRX_ID_MIN 和 TRX_ID_MAX 之间的关系，从而判断数据行快照是否可以使用：

- TRX_ID < TRX_ID_MIN，表示该数据行快照时在当前所有未提交事务之前进行更改的，因此可以使用。
- TRX_ID > TRX_ID_MAX，表示该数据行快照是在事务启动之后被更改的，因此不可使用。
- TRX_ID_MIN <= TRX_ID <= TRX_ID_MAX，需要根据隔离级别再进行判断：
    - 提交读：如果 TRX_ID 在 TRX_IDs 列表中，表示该数据行快照对应的事务还未提交，则该快照不可使用。否则表示已经提交，可以使用。
    - 可重复读：都不可以使用。因为如果可以使用的话，那么其它事务也可以读到这个数据行快照并进行修改，那么当前事务再去读这个数据行得到的值就会发生改变，也就是出现了不可重复读问题。

在数据行快照不可使用的情况下，需要沿着 Undo Log 的回滚指针 ROLL_PTR 找到下一个快照，再进行上面的判断。

**快照读与当前读**

**•快照读**

MVCC 的 SELECT 操作是快照中的数据，不需要进行加锁操作。

```
SELECT * FROM table ...;
```

**•当前读**

MVCC 其它会对数据库进行修改的操作（INSERT、UPDATE、DELETE）需要进行加锁操作，从而读取最新的数据。可以看到 MVCC 并不是完全不用加锁，而只是避免了 SELECT 的加锁操作。

```
INSERT;
UPDATE;
DELETE;
```

在进行 SELECT 操作时，可以强制指定进行加锁操作。以下第一个语句需要加 S 锁，第二个需要加 X 锁。

```
SELECT * FROM table WHERE ? lock in share mode;
SELECT * FROM table WHERE ? for update;
```

## 6.Next-Key Locks

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

MVCC 不能解决幻影读问题，Next-Key Locks 就是为了解决这个问题而存在的。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 会自动在主键上创建隐藏的聚簇索引，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 t.c 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。它锁定一个前开后闭区间，例如一个索引包含以下值：10, 11, 13, and 20，那么就需要锁定以下区间：

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

## 7.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 |
| 2 | 学生-2 | 学院-2 | 院长-2 | 课程-1 | 100 |
| 3 | 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。

2. 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 |
| 2 | 学生-2 | 学院-2 | 院长-2 | 课程-1 | 100 |

3　　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，它没有任何冗余数据，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

### 3. 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 |

## 8.ER 图

Entity-Relationship，有三个组成部分：实体、属性、联系。

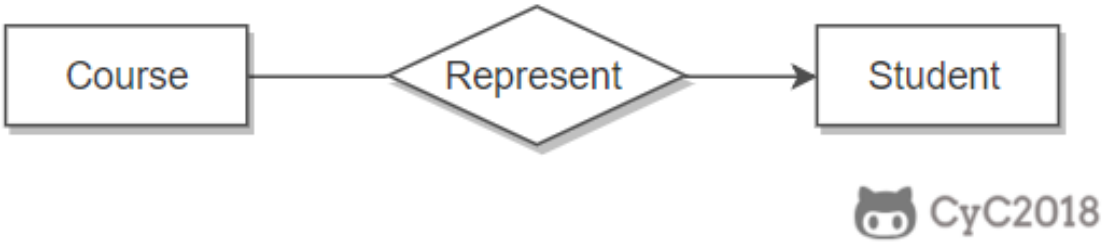
用来进行关系型数据库系统的概念设计。

### 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。

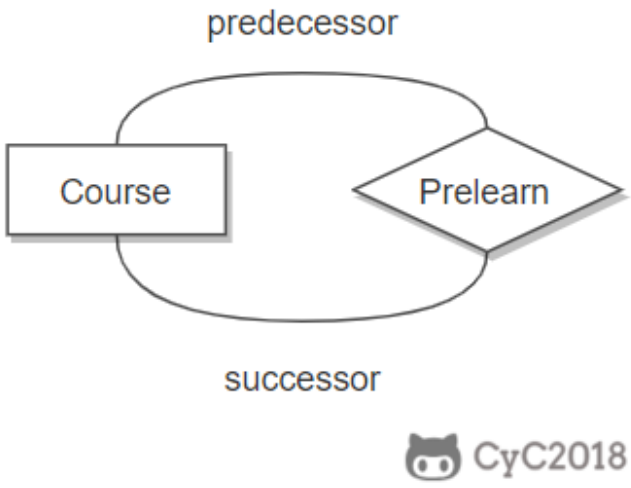
下图的 Course 和 Student 是一对多的关系。




### 表示出现多次的关系

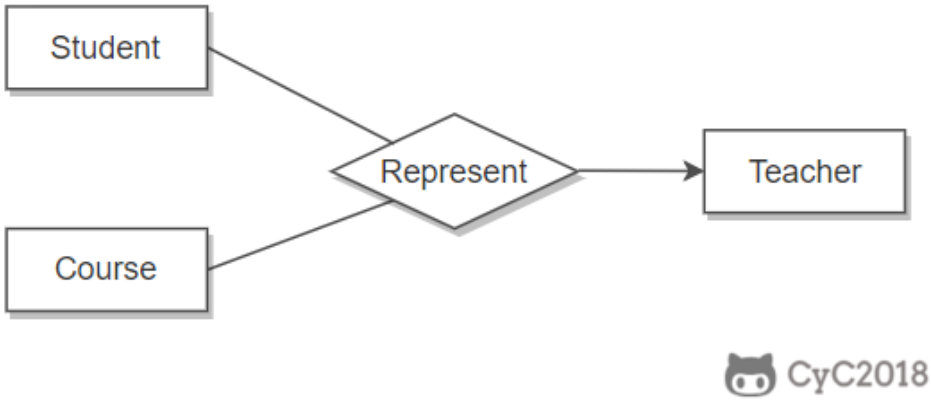
一个实体在联系出现几次，就要用几条线连接。

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。



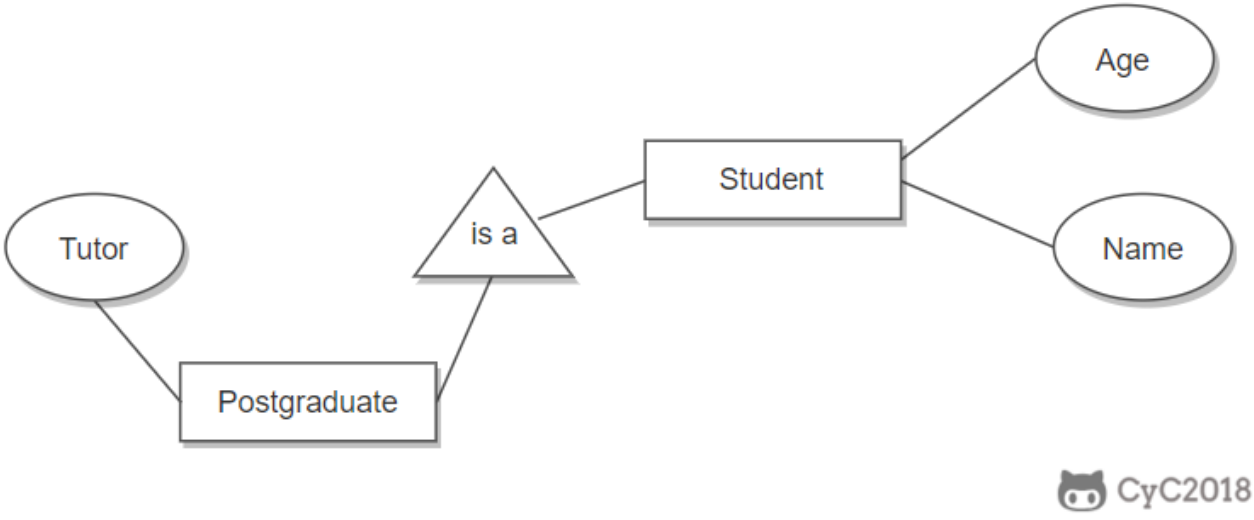
### 联系的多向性

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联系。




### 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。



# MYSQL

## 1.索引

### B+树原理

### 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，并且所有叶子节点位于同一层。

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，一个节点中的 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$key_i$和$key_{i+1}$，且不为 null，则该指针指向节点的所有 key 大于等于$key_i$且小于等于$key_{i+1}$。

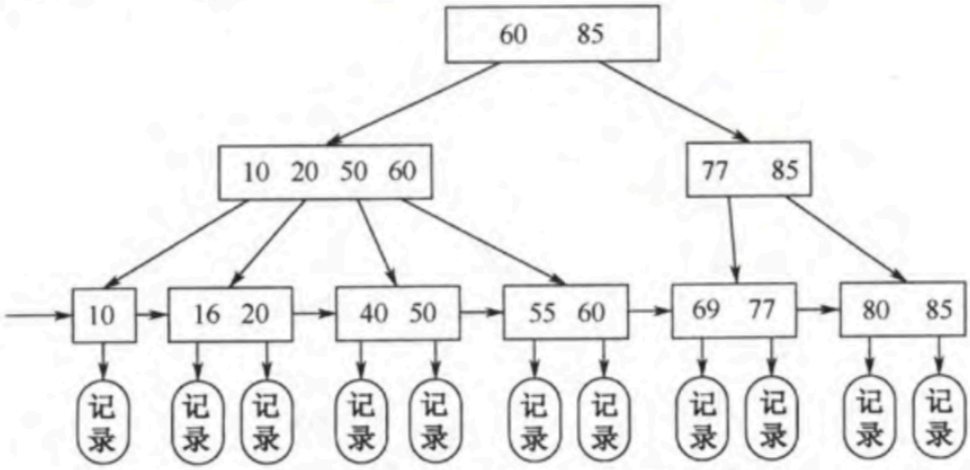


图 6-9 B+树结构示意图

### 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，因此在进行插入删除操作之后，需要对树进行分裂、合并、旋转等操作来维护平衡性。

**B+树与红黑树的比较**

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，这是因为使用 B+ 树访问磁盘数据有更高的性能。

**(1)B+树有更低的树高**

平衡树的树高 $O(h)=O(\log_d N)$，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。

**(2)磁盘访问原理**

操作系统一般将内存和磁盘分割成固定大小的块，每一块称为一页，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。

如果数据不在同一个磁盘块上，那么通常需要移动制动手臂进行寻道，而制动手臂因为其物理结构导致了移动效率低下，从而增加磁盘数据读取时间。B+ 树相对于红黑树有更低的树高，进行寻道的次数与树高成正比，在同一个磁盘块上进行访问只需要很短的磁盘旋转时间，所以 B+ 树更适合磁盘数据的读取。

**(3)磁盘预读特性**

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。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。

**1.B+树索引**

是大多数 MySQL 存储引擎的默认索引类型。

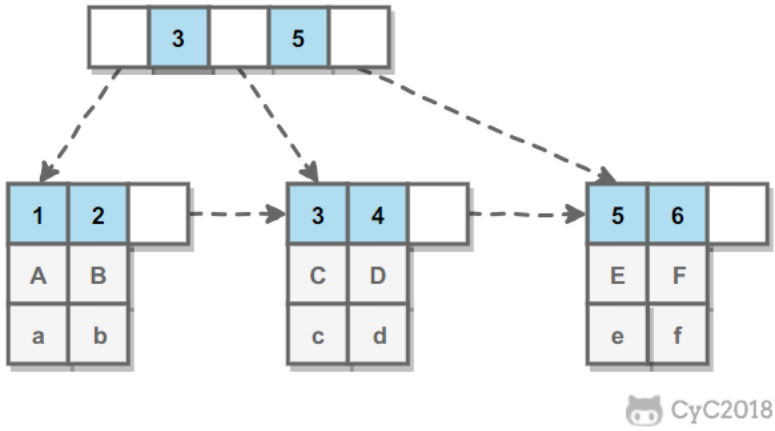
因为不再需要进行全表扫描，只需要对树进行搜索即可，所以查找速度快很多。

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。

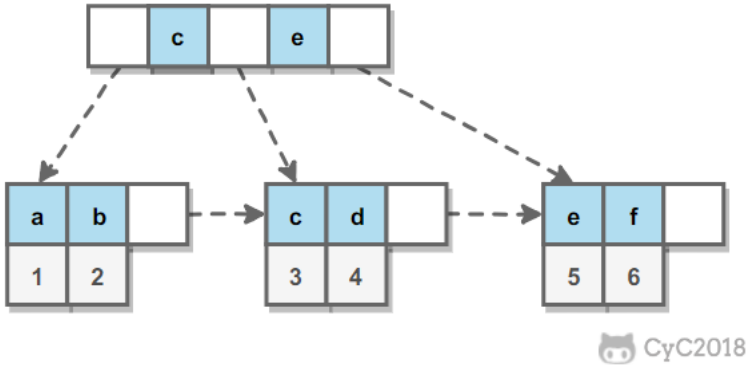
可以指定多个列作为索引列，多个索引列共同组成键。

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




### 2.哈希索引

哈希索引能以 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。

### 3.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。

### 4.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

### 1.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;
```

### 2.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。例如下面的语句中，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;
```

### 3.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;
   staff_id_selectivity: 0.0001
customer_id_selectivity: 0.0373
              COUNT(*): 16049
```

### 4.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。

### 5.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。

## 2.查询性能优化

### 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

#### 1.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。

#### 2.减少服务器端扫描的行数

最有效的方式是使用索引来覆盖查询。

### 重构查询方式

#### 1.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
```

### 2.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。而分解后的多个查询，即使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 tag
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);
```

## 3.存储引擎

### 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）+ 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。其它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。

### InnoDB 与 MyISAM 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 支持压缩表和空间数据索引。

## 4.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。但是在执行 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。

#### 1. 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。

#### 2. 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 个字节，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 FROM_UNIXTIME() 函数把 UNIX 时间戳转换为日期，并提供了 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。

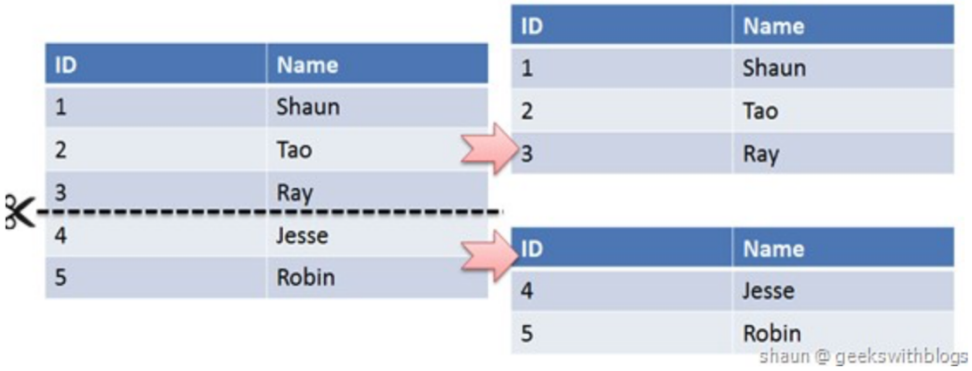
应该尽量使用 TIMESTAMP，因为它比 DATETIME 空间效率更高。

## 5.切分

### 水平切分

水平切分又称为 Sharding，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，Sharding 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。

### Sharding 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。

### Sharding 存在的问题

- 事务问题：使用分布式事务来解决，比如 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

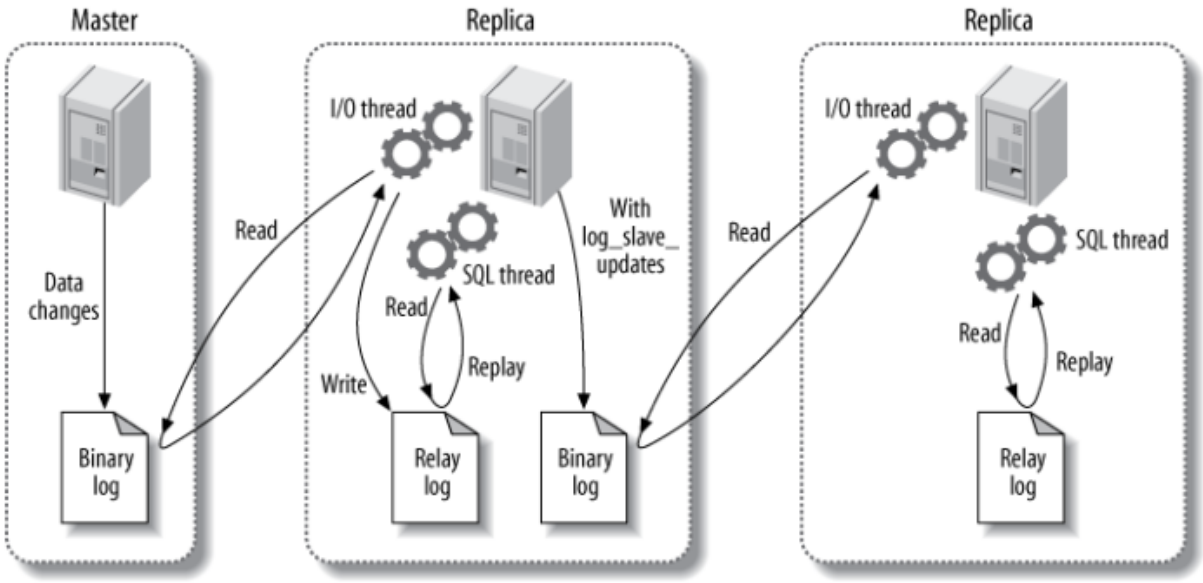
  分布式 ID 生成器 (如 Twitter 的 Snowflake 算法)

## 6.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。

## 7.MYSQL 常见面试题

### 什么是 MySQL?

MySQL 是一种关系型数据库，在 Java 企业级开发中非常常用，因为 MySQL 是开源免费的，并且方便扩 展。阿里巴巴数据库系统也大量用到了 MySQL，因此它的稳定性是有保障的。MySQL 是开放源代码的， 因此任何人都可以在 GPL(General Public License) 的许可下下载并根据个性化的需要对其进行修改。MySQL 的默认端口号是 **3306**。

### MyISAM 和 InnoDB 区别?

MyISAM 是 MySQL 的默认数据库引擎(5.5 版之前)。虽然性能极佳，而且提供了大量的特性，包括全文索 引、压缩、空间函数等，但 MyISAM 不支持事务和行级锁，而且最大的缺陷就是崩溃后无法安全恢复。不 过，5.5 版本之后，MySQL 引入了 InnoDB(事务性数据库引擎)，MySQL 5.5 版本后默认的存储引擎为 InnoDB。

大多数时候我们使用的都是 InnoDB 存储引擎，但是在某些情况下使用 MyISAM 也是合适的比如读密集 的情况下。(如果你不介意 MyISAM 崩溃恢复问题的话)。

**两者的对比:**

•是否支持行级锁 : MyISAM 只有表级锁(table-level locking)，而 InnoDB 支持行级锁(row- level locking)和表级锁,默认为行级锁。

•是否支持事务和崩溃后的安全恢复: MyISAM 强调的是性能，每次查询具有原子性,其执行速度比 InnoDB 类型更快，但是不提供事务支持。但是 InnoDB 提供事务支持事务，外部键等高级数据库功能。 具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities) 的事务安全(transaction-safe (ACID compliant))型表。

•是否支持外键: MyISAM 不支持，而 InnoDB 支持。

•是否支持 MVCC :仅 InnoDB 支持。应对高并发事务, MVCC 比单纯的加锁更高效;MVCC 只在 READ COMMITTED 和 REPEATABLE READ 两个隔离级别下工作;MVCC 可以使用 乐观 (optimistic)锁和 悲观(pessimistic)锁来实现;各数据库中 MVCC 实现并不统一。推荐阅读: MySQL-InnoDB-MVCC 多版本并发控制 https://segmentfault.com/a/11900000126505966

《MySQL 高性能》上面有一句话这样写到:

不要轻易相信“MyISAM 比 InnoDB 快”之类的经验之谈，这个结论往往不是绝对的。在很多我们已知场 景中，InnoDB 的速度都可以让 MyISAM 望尘莫及，尤其是用到了聚簇索引，或者需要访问的数据都可 以放入内存的应用。

一般情况下我们选择 InnoDB 都是没有问题的，但是某些情况下你并不在乎可扩展能力和并发能力，也 不需要事务支持，也不在乎崩溃后的安全恢复问题的话，选择 MyISAM 也是一个不错的选择。但是一般情 况下，我们都是需要考虑到这些问题的。

### 字符集及校对规则

字符集指的是一种从二进制编码到某类字符符号的映射。校对规则则是指某种字符集下的排序规则。

MySQL 中每一种字符集都会对应一系列的校对规则。

MySQL 采用的是类似继承的方式指定字符集的默认值，每个数据库以及每张数据表都有自己的默认值， 他们逐层继承。比如:某个库中所有表的默认字符集将是该数据库所指定的字符集(这些表在没有指定 字符集的情况下，才会采用默认字符集) PS:整理自《Java 工程师修炼之道》

详细内容可以参考:
MySQL 字符集及校对规则的理解 https://www.cnblogs.com/geaozhang/p/6724393.html

## 索引

MySQL 索引使用的数据结构主要有 **BTree 索引** 和 **哈希索引** 。对于哈希索引来说，底层的数据结构就是 哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快;其余大部分 场景，建议选择 BTree 索引。

MySQL 的 BTree 索引使用的是 B 树中的 B+Tree，但对于主要的两种存储引擎的实现方式是不同的。

**MyISAM:** B+Tree 叶节点的 data 域存放的是数据记录的地址。在索引检索的时候，首先按照 B+Tree
搜索算法搜索索引，如果指定的 Key 存在，则取出其 data 域的值，然后以 data 域的值为地址 读取相应的数据记录。这被称为“非聚簇索引”。

**InnoDB:** 其数据文件本身就是索引文件。相比 MyISAM，索引文件和数据文件是分离的，其表数据 文件本身就是按 B+Tree 组织的一个索引结构，树的叶节点 data 域保存了完整的数据记录。这个索 引的 key 是数据表的主键，因此 InnoDB 表数据文件本身就是主索引。这被称为“聚簇索引(或聚集 索引)”。而其余的索引都作为辅助索引，辅助索引的 data 域存储相应记录主键的值而不是地 址，这也是和 MyISAM 不同的地方。**在根据主索引搜索时，直接找到 key 所在的节点即可取出数据;在根据辅助索引查找时，则需要先取出主键的值，再走一遍主索引。 因此，在设计表的时候，不建议使用过长的字段作为主键，也不建议使用非单调的字段作为主键，这样会造成主索引频繁分裂。**

## 查询缓存的使用

执行查询语句的时候，会先查询缓存。不过，MySQL 8.0 版本后移除，因为这个功能不太实用
my.cnf 加入以下配置，重启 MySQL 开启查询缓存

```
query_cache_type=1
query_cache_size=600000
```

MySQL 执行以下命令也可以开启查询缓存

```
set global    query_cache_type=1;
set global    query_cache_size=600000;
```

如上，**开启查询缓存后在同样的查询条件以及数据情况下，会直接在缓存中返回结果**。这里的查询条件包括查询本身、当前要查询的数据库、客户端协议版本号等一些可能影响结果的信息。因此任何两个查询在任何字符上的不同都会导致缓存不命中。此外，如果查询中包含任何用户自定义函数、存储函数、 用户变量、临时表、MySQL 库中的系统表，其查询结果也不会被缓存。
缓存建立之后，MySQL 的查询缓存系统会跟踪查询中涉及的每张表，如果这些表(数据或结构)发生变化，那么和这张表相关的所有缓存数据都将失效。

**缓存虽然能够提升数据库的查询性能，但是缓存同时也带来了额外的开销，每次查询后都要做一次缓存 操作，失效后还要销毁。** 因此，开启缓存查询要谨慎，尤其对于写密集的应用来说更是如此。如果开 启，要注意合理控制缓存空间大小，一般来说其大小设置为几十 MB 比较合适。此外，**还可以通过 sql_cache 和 sql_no_cache 来控制某个查询语句是否需要缓存:**

## 什么是事务?

**事务是逻辑上的一组操作，要么都执行，要么都不执行。**

事务最经典也经常被拿出来说例子就是转账了。假如小明要给小红转账 1000 元，这个转账会涉及到两个关键操作就是:将小明的余额减少 1000 元，将小红的余额增加 1000 元。万一在这两个操作之间突然出现 错误比如银行系统崩溃，导致小明余额减少而小红的余额没有增加，这样就不对了。事务就是保证这两 个关键操作要么都成功，要么都要失败。

**事务的四大特性?**

**•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，多个事务对同一个数据读取的结果是 相同的;
**•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。

**并发事务带来哪些问题?**

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务(多个用户对同一数据进行操作)。并发虽然是必须的，但可能会导致以下的问题。

**•脏读(Dirty read):** 当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交 到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没 有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是 不正确的。

**•丢失修改(Lost to modify):** 指在一个事务读取一个数据时，另外一个事务也访问了该数据， 那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修 改结果就被丢失，因此称为丢失修改。 例如:事务 1 读取某表中的数据 A=20，事务 2 也读取 A=20，事务 1 修改 A=A-1，事务 2 也修改 A=A-1，最终结果 A=19，事务 1 的修改被丢失。

**•不可重复读(Unrepeatableread):** 指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修 改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不 一样的情况，因此称为不可重复读。

**•幻读(Phantom read):** 幻读与不可重复读类似。它发生在一个事务(T1)读取了几行数据，接 着另一个并发事务(T2)插入了一些数据时。在随后的查询中，第一个事务(T1)就会发现多了 一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

**不可重复读和幻读区别:** 不可重复读的重点是修改比如多次读取一条记录发现其中某些列的值被修改，幻读的重点在于新增或者删除比如多次读取一条记录发现记录增多或减少了。

**事务隔离级别有哪些?MySQL 的默认隔离级别是?**

SQL 标准定义了四个隔离级别:

**•READ-UNCOMMITTED(读取未提交):** 最低的隔离级别，允许读取尚未提交的数据变更，可能会导 致脏读、幻读或不可重复读。

**•READ-COMMITTED(读取已提交):** 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读 或不可重复读仍有可能发生。

•**REPEATABLE-READ(可重复读):** 对同一字段的多次读取结果都是一致的，除非数据是被本身事务 自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

•**SERIALIZABLE(可串行化):** 最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及 幻读。

MySQL InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ(可重读)。我们可以通过 SELECT @@tx_isolation;命令来查看

这里需要注意的是:与 SQL 标准不同的地方在于 InnoDB 存储引擎在 REPEATABLE-READ(可重读)事务隔离级别下使用的是 Next-Key Lock 锁算法，因此可以避免幻读的产生，这与其他数据库系统(如 SQL Server) 是不同的。所以说 InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ(可重 读) 已经可以完全保证事务的隔离性要求，即达到了 SQL 标准的 SERIALIZABLE(可串行化) 隔离级 别。因为隔离级别越低，事务请求的锁越少，所以大部分数据库系统的隔离级别都是 READ- COMMITTED(读取提交内容) ，但是你要知道的是 InnoDB 存储引擎默认使用 REPEAaTABLE-READ(可重 读) 并不会有任何性能损失。

InnoDB 存储引擎在 **分布式事务** 的情况下一般会用到 **SERIALIZABLE(可串行化) 隔离级别**。

## 锁机制与 InnoDB 锁算法

MyISAM 和 InnoDB 存储引擎使用的锁: MyISAM 采用表级锁(table-level locking)。 InnoDB 支持行级锁(row-level locking)和表级锁,默认为行级锁

**表级锁和行级锁对比:**

•**表级锁:** MySQL 中锁定 粒度最大 的一种锁，对当前操作的整张表加锁，实现简单，资源消耗也 比⍰少，加锁快，不会出现死锁。其锁定粒度最大，触发锁冲突的概率最高，并发度最低，MyISAM 和 InnoDB 引擎都支持表级锁。

•**行级锁:** MySQL 中锁定 粒度最小 的一种锁，只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，并发度高，但加锁的开销也最大，加锁慢，会出现死锁。详细内容可以参考: MySQL 锁机制简单了解一下:

https://blog.csdn.net/qq_34337272/article/details/80611486

InnoDB 存储引擎的锁的算法有三种:

•**Record lock:**单个行记录上的锁

•**Gap lock:**间隙锁，锁定一个范围，不包括记录本身

•**Next-key lock:record+gap** 锁定一个范围，包含记录本身

相关知识点:

•innodb 对于行的查询使用 next-key lock

•Next-locking keying 为了解决 Phantom Problem 幻读问题

•当查询的索引含有唯一属性时，将 next-key lock 降级为 record key

•Gap 锁设计的目的是为了阻止多个事务将记录插入到同一范围内，而这会导致幻读问题的产生

•有两种方式显式关闭 gap 锁:(除了外键约束和唯一性检查外，其余情况仅使用 record lock)

A.将事务隔离级别设置为 RC；

B.将参数 innodb_locks_unsafe_for_binlog 设置为 1；

## 大表优化

当 MySQL 单表记录数过大时，数据库的 CRUD 性能会明显下降，一些常见的优化措施如下:

**(1)限定数据的范围** ： 务必禁止不带任何限制数据范围条件的查询语句。比如:我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内；

**(2)读/写分离：**经典的数据库拆分方案，主库负责写，从库负责读;

**(3)垂直分区： 根据数据库里面数据表的相关性进行拆分。** 例如，用户表中既有用户的登录信息又有用户的基本信 息，可以将用户表拆分成两个单独的表，甚至放到单独的库做分库。

**简单来说垂直拆分是指数据表列的拆分，把一张列比较多的表拆分为多张表。** 如下图所示，这样来说 大家应该就更容易理解了。

**垂直拆分的优点:** 可以使得列数据变小，在查询时减少读取的 Block 数，减少 I/O 次数。此外，垂直分区可以简化表的结构，易于维护。

**垂直拆分的缺点:** 主键会出现冗余，需要管理冗余列，并会引起 Join 操作，可以通过在应用层进行 Join 来解决。此外，垂直分区会让事务变得更加复杂;

**(4)水平分区：保持数据表结构不变，通过某种策略存储数据分片。这样每一片数据分散到不同的表或者库中，达到了 分布式的目的。 水平拆分可以支撑非常大的数据量。**

水平拆分是指数据表行的拆分，表的行数超过 200 万行时，就会变慢，这时可以把一张的表的数据拆成 多张表来存放。举个例子:我们可以将用户信息表拆分成多个用户信息表，这样就可以避免单一表数据 量过大对性能造成影响。

水平拆分可以支持非常大的数据量。需要注意的一点是:分表仅仅是解决了单一表数据过大的问题，但由于表的数据还是在同一台机器上，其实对于提升 MySQL 并发能力没有什么意义，所 **水平拆分最好分库**。

水平拆分能够**支持非常大的数据量存储，应用端改造也少**，但**分片事务难以解决** ，跨节点 Join 性能 □差，逻辑复杂。《Java 工程师修炼之道》的作者推荐 **尽量不要对数据进行分片，因为拆分会带来逻 辑、部署、运维的各种复杂度** ，一般的数据表在优化得当的情况下支撑千万以下的数据量是没有太大问题的。如果实在要分片，尽量选择客户端分片架构，这样可以减少一次和中间件的网络 I/O。

**下面补充一下数据库分片的两种常见方案:**

**客户端代理: 分片逻辑在应用端，封装在 jar 包中，通过修改或者封装 JDBC 层来实现。** 当当网的 **Sharding-JDBC** 、阿里的 TDDL 是两种比□常用的实现。

**中间件代理: 在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中。** 我们现在谈的 **Mycat** 、360 的 Atlas、网易的 DDB 等等都是这种架构的实现。

详细内容可以参考: MySQL 大表优化方案: https://segmentfault.com/a/1190000006158186

**解释一下什么是池化设计思想。什么是数据库连接池?为什么需要数据库连接池?**

池化设计应该不是一个新名词。我们常见的如 java 线程池、jdbc 连接池、redis 连接池等就是这类设计 的代表实现。这种设计会初始预设资源，解决的问题就是抵消每次获取资源的消耗，如创建线程的开 销，获取远程连接的开销等。就好比你去食堂打饭，打饭的大妈会先把饭盛好几份放那里，你来了就直 接拿着饭盒加菜即可，不用再临时又盛饭又打菜，效率就高了。除了初始化资源，池化设计还包括如下 这些特征:池子的初始值、池子的活跃值、池子的最大值等，这些特征可以直接映射到 java 线程池和数 据库连接池的成员属性中。这篇文章对池化设计思想介绍的还不错，直接复制过来，避免重复造轮子 了。

数据库连接本质就是一个 socket 的连接。数据库服务端还要维护一些缓存和用户权限信息之类的 所 以占用了一些内存。我们可以把数据库连接池是看做是维护的数据库连接的缓存，以便将来需要对数据 库的请求时可以重用这些连接。为每个用户打开和维护数据库连接，尤其是对动态数据库驱动的网站应 用程序的请求，既昂贵又浪费资源。**在连接池中，创建连接后，将其放置在池**

**中，并再次使用它，因此 不必建立新的连接。如果使用了所有连接，则会建立一个新连接并将其添加到池中。** 连接池还减少了 用户必须等待建立与数据库的连接的时间。

**分库分表之后,id 主键如何处理?**

因为要是分成多个表之后，每个表都是从 1 开始累加，这样是不对的，我们需要一个全局唯一的 id 来支持。

生成全局 id 有下面这几种方式:

**UUID**:不适合作为主键，因为太长了，并且无序不可读，查询效率低。比□适合用于生成唯一的名字的标示比如文件的名字。

**数据库自增 id**：两台数据库分别设置不同步长，生成不重复 ID 的策略来实现高可用。这种方式 生成的 id 有序，但是需要独立部署数据库实例，成本高，还会有性能瓶颈。

**利用 redis 生成 id：**性能比□好，灵活方便，不依赖于数据库。但是，引入了新的组件造成 系统更加复杂，可用性降低，编码更加复杂，增加了系统成本。

**Twitter 的 snowflake 算法** :Github 地址:https://github.com/twitter-archive/snowflake。

**美团的 Leaf 分布式 ID 生成系统** :Leaf 是美团开源的分布式 ID 生成器，能保证全局唯一性、趋势 递增、单调递增、信息安全，里面也提到了几种分布式方案的对比，但也需要依赖关系数据库、Zookeeper 等中间件。感觉还不错。美团技术团队的一篇文章:https://tech.meituan.com/2017/04/21/mt-leaf.html 。

.....

**SQL 语句在 MySQL 中如何执行的**

一条 SQL 语句在 MySQL 中如何执行的

https://mp.weixin.qq.com/s?__biz=Mzg2OTA0Njk0OA==&mid=2247485097&idx=1&sn=84c89da477b1338bdf3e9fcd65514ac1&chksm=cea24962f9d5c074d8d3ff1ab04ee8f0d6486e3d015cfd783503685986485c11738ccb542ba7&token=79317275&lang=zh_CN%23rd

**MYSQL 高性能优化规范建议**

MySQL 高性能优化规范建议

https://mp.weixin.qq.com/s?__biz=Mzg2OTA0Njk0OA==&mid=2247485117&idx=1&sn=92361755b7c3de488b415ec4c5f46d73&chksm=cea24976f9d5c060babe50c3747616cce63df5d50947903a262704988143c2eeb4069ae45420&token=79317275&lang=zh_CN%23rd

**一条 SQL 语句执行得很慢的原因有哪些?**

腾讯面试:一条 SQL 语句执行得很慢的原因有哪些?---不看后悔系列

https://mp.weixin.qq.com/s?__biz=Mzg2OTA0Njk0OA==&mid=2247485185&idx=1&sn=66ef08b4ab6af5757792223a83fc0d45&chksm=cea248caf9d5c1dc72ec8a281ec16aa3ec3e8066dbb252e27362438a26c33fbe842b0e0adf47&token=79317275&lang=zh_CN%23rd

## 后端程序员必备:书写高质量 SQL 的 30 条建议

后端程序员必备:书写高质量 SQL 的 30 条建议

https://mp.weixin.qq.com/s?__biz=Mzg2OTA0Njk0OA==&mid=2247486461&idx=1&sn=60a22279196d084cc398936fe3b37772&chksm=cea24436f9d5cd20a4fa0e907590f3e700d7378b3f608d7b33bb52cfb96f503b7ccb65a1deed&token=1987003517&lang=zh_CN%23rd

# Redis

## 1.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。

## 2.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<br>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<br>对单个或者多个元素进行修剪，<br>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<br>检查一个元素是否存在于集合中<br>计算交集、并集、差集<br>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<br>获取所有键值对<br>检查某个键是否存在 |
| ZSET | 有序集合 | 添加、获取、删除元素<br>根据分值范围或者成员来获取元素<br>计算一个键的排名 |

## 3.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。

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。

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，然后每执行一次 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++。

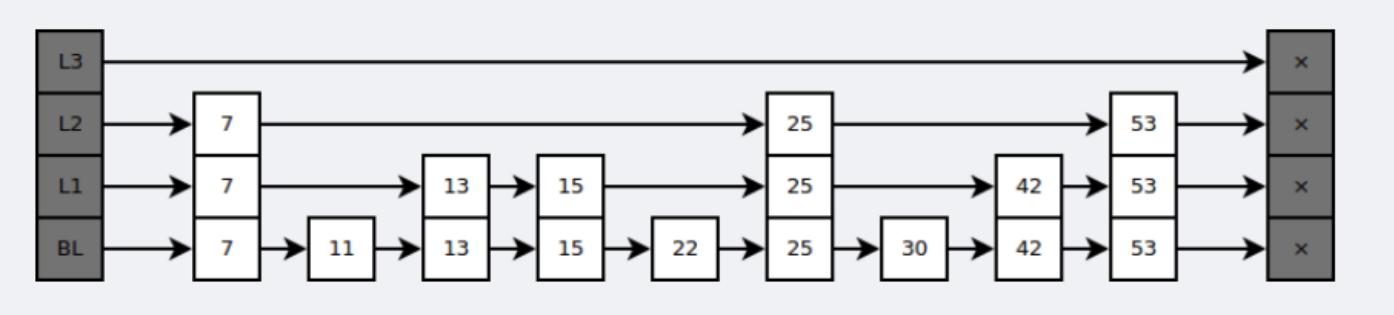
在 rehash 期间，每次对字典执行添加、删除、查找或者更新操作时，都会执行一次渐进式 rehash。

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。

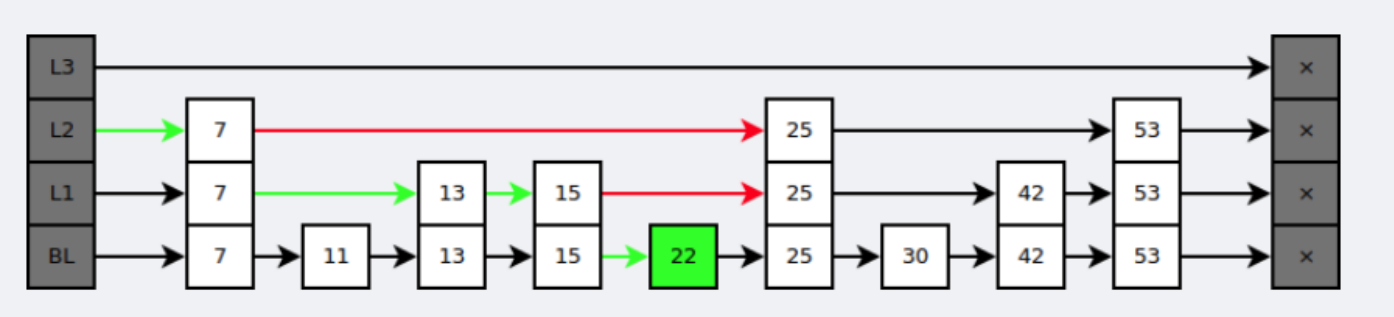
### 跳跃表

是有序集合的底层实现之一。

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查找。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；

## 4.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。

当应用服务器不再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。

## 5.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 的数据则会一直在内存中。
- 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 就浪费掉了。

## 6.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。

## 7.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针对所有 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。

Redis 4.0 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。

## 8.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，然后由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 选项会严重减低服务器的性能；
- 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。

## 9.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。

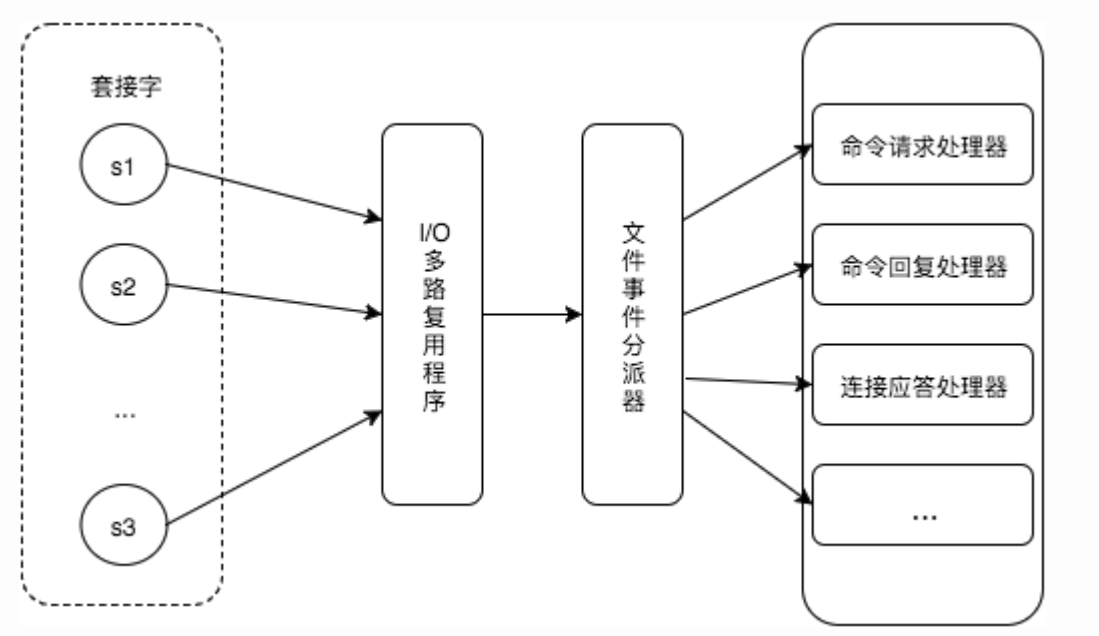
## 10. 事件

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

服务器有一些操作需要在给定的时间点执行，时间事件是对这类定时操作的抽象。

时间事件又分为：

- 定时事件：是让一段程序在指定的时间之内执行一次；
- 周期性事件：是让一段程序每隔指定时间就执行一次。

Redis 将所有时间事件都放在一个无序链表中，通过遍历整个链表查找出已到达的时间事件，并调用相应的事件处理器。

### 事件的调度与执行

服务器需要不断监听文件事件的套接字才能得到待处理的文件事件，但是不能一直监听，否则时间事件无法在规定的时间内执行，因此监听时间应该根据距离现在最近的时间事件来决定。

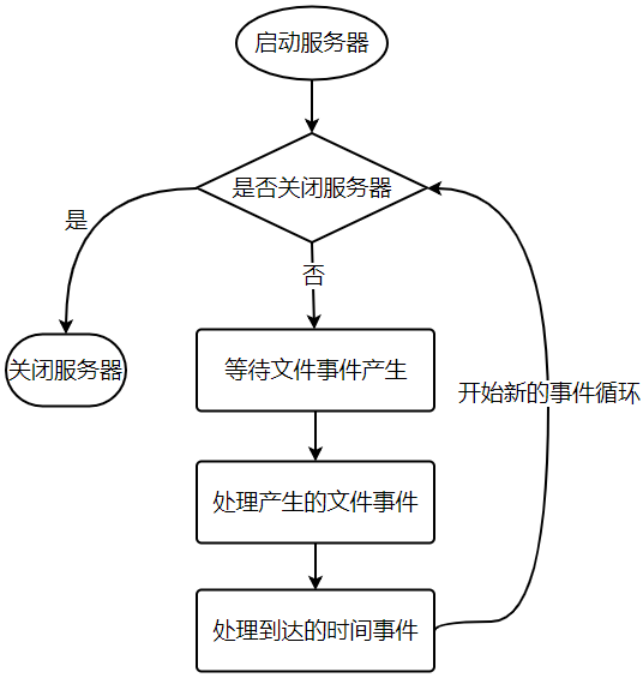
事件调度与执行由 aeProcessEvents 函数负责，伪代码如下：

```
def aeProcessEvents():
    # 获取到达时间离当前时间最接近的时间事件
    time_event = aeSearchNearestTimer()
    # 计算最接近的时间事件距离到达还有多少毫秒
    remaind_ms = time_event.when - unix_ts_now()
    # 如果事件已到达，那么 remaind_ms 的值可能为负数，将它设为 0
    if remaind_ms < 0:
        remaind_ms = 0
    # 根据 remaind_ms 的值，创建 timeval
    timeval = create_timeval_with_ms(remaind_ms)
    # 阻塞并等待文件事件产生，最大阻塞时间由传入的 timeval 决定
    aeApiPoll(timeval)
    # 处理所有已产生的文件事件
    procesFileEvents()
    # 处理所有已到达的时间事件
    processTimeEvents()
```

将 aeProcessEvents 函数置于一个循环里面，加上初始化和清理函数，就构成了 Redis 服务器的主函数，伪代码如下：

```
def main():
    # 初始化服务器
    init_server()
    # 一直处理事件，直到服务器关闭为止
    while server_is_not_shutdown():
        aeProcessEvents()
    # 服务器关闭，执行清理操作
    clean_server()
```

从事件处理的角度来看，服务器运行流程如下：




## 11.复制

通过使用 slaveof host port 命令来让一个服务器成为另一个服务器的从服务器。
一个从服务器只能有一个主服务器，并且不支持主主复制。

### 连接过程

(1)主服务器创建快照文件，发送给从服务器，并在发送期间使用缓冲区记录执行的写命令。快照文件发送完毕之后，开始向从服务器发送存储在缓冲区中的写命令；
(2)从服务器丢弃所有旧数据，载入主服务器发来的快照文件，之后从服务器开始接受主服务器发来的写命令；
(3)主服务器每执行一次写命令，就向从服务器发送相同的写命令。

### 主从链

随着负载不断上升，主服务器可能无法很快地更新所有从服务器，或者重新连接和重新同步从服务器将导致系统超载。为了解决这个问题，可以创建一个中间层来分担主服务器的复制工作。中间层的服务器是最上层服务器的从服务器，又是最下层服务器的主服务器。

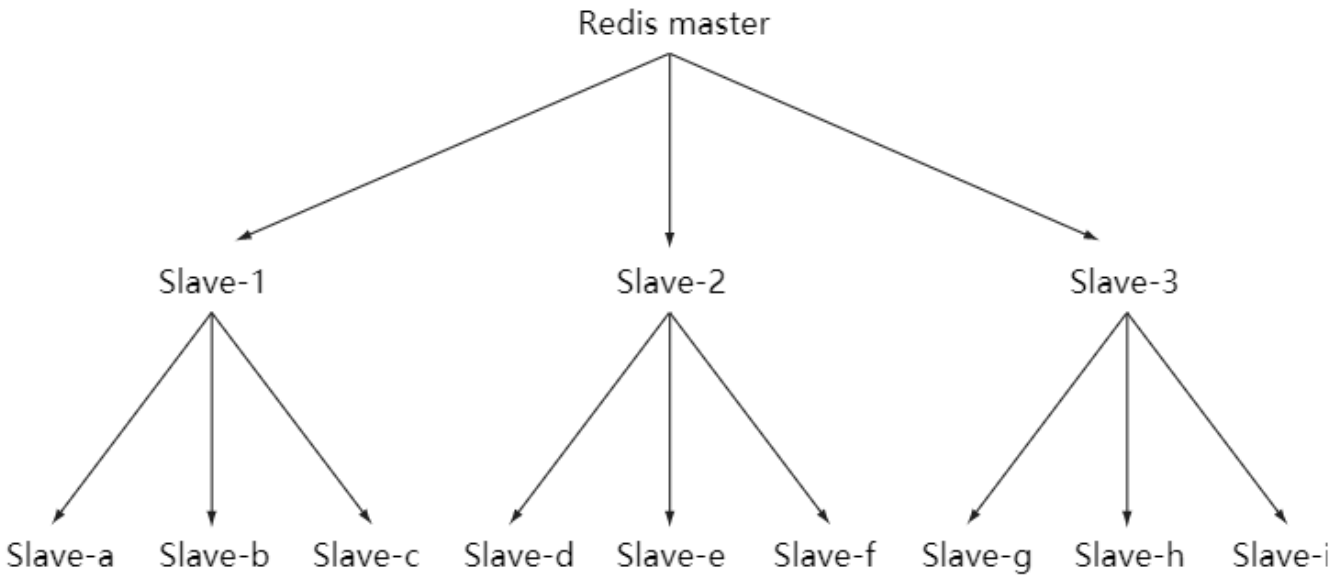



*Figure 4.1 An example Redis master/slave replica tree with nine lowest-level slaves and three intermediate replication helper servers*

## 12.Sentinel

Sentinel（哨兵）可以监听集群中的服务器，并在主服务器进入下线状态时，自动从从服务器中选举出新的主服务器。

## 13.分片

分片是将数据划分为多个部分的方法，可以将数据存储到多台机器里面，这种方法在解决某些问题时可以获得线性级别的性能提升。

假设有 4 个 Redis 实例 R0，R1，R2，R3，还有很多表示用户的键 user:1，user:2，... ，有不同的方式来选择一个指定的键存储在哪个实例中。

- 最简单的方式是范围分片，例如用户 id 从 0~1000 的存储到实例 R0 中，用户 id 从 1001~2000 的存储到实例 R1 中，等等。但是这样需要维护一张映射范围表，维护操作代价很高。
- 还有一种方式是哈希分片，使用 CRC32 哈希函数将键转换为一个数字，再对实例数量求模就能知道应该存储的实例。

根据执行分片的位置，可以分为三种分片方式：

- 客户端分片：客户端使用一致性哈希等算法决定键应当分布到哪个节点。

- 代理分片：将客户端请求发送到代理上，由代理转发请求到正确的节点上。
- 服务器分片：Redis Cluster。

## 14.一个简单的论坛系统分析

该论坛系统功能如下：

- 可以发布文章；
- 可以对文章进行点赞；
- 在首页可以按文章的发布时间或者文章的点赞数进行排序显示。

### 文章信息

文章包括标题、作者、赞数等信息，在关系型数据库中很容易构建一张表来存储这些信息，在 Redis 中可以使用 HASH 来存储每种信息以及其对应的值的映射。

Redis 没有关系型数据库中的表这一概念来将同种类型的数据存放在一起，而是使用命名空间的方式来实现这一功能。键名的前面部分存储命名空间，后面部分的内容存储 ID，通常使用 : 来进行分隔。例如下面的 HASH 的键名为 article:92617，其中 article 为命名空间，ID 为 92617。

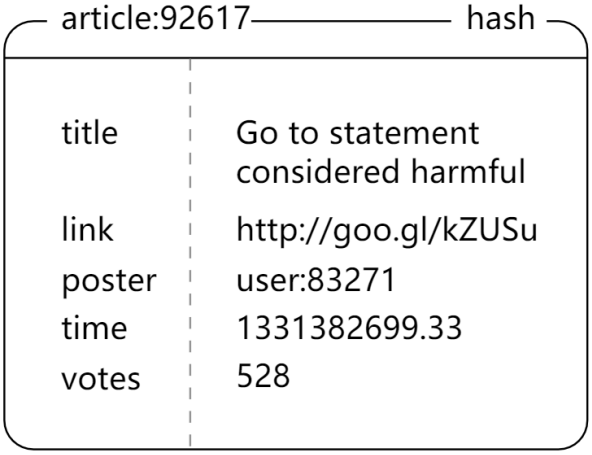



Figure 1.8An example article stored as a HASH for our article voting system

### 点赞功能

当有用户为一篇文章点赞时，除了要对该文章的 votes 字段进行加 1 操作，还必须记录该用户已经对该文章进行了点赞，防止用户点赞次数超过 1。可以建立文章的已投票用户集合来进行记录。

为了节约内存，规定一篇文章发布满一周之后，就不能再对它进行投票，而文章的已投票集合也会被删除，可以为文章的已投票集合设置一个一周的过期时间就能实现这个规定。

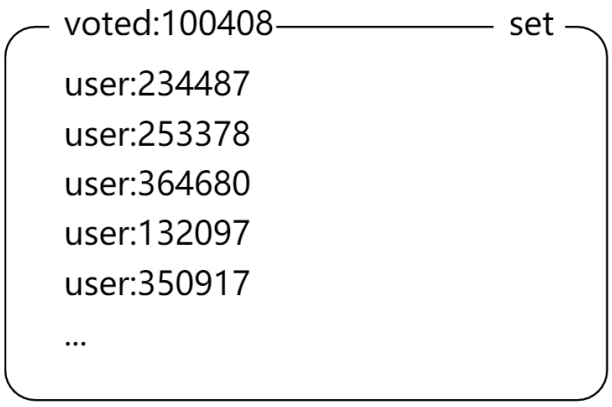



Figure 1.10 Some users who have voted for article 100408

**对文章进行排序**

为了按发布时间和点赞数进行排序，可以建立一个文章发布时间的有序集合和一个文章点赞数的有序集合。（下图中的 score 就是这里所说的点赞数；下面所示的有序集合分值并不直接是时间和点赞数，而是根据时间和点赞数间接计算出来的）

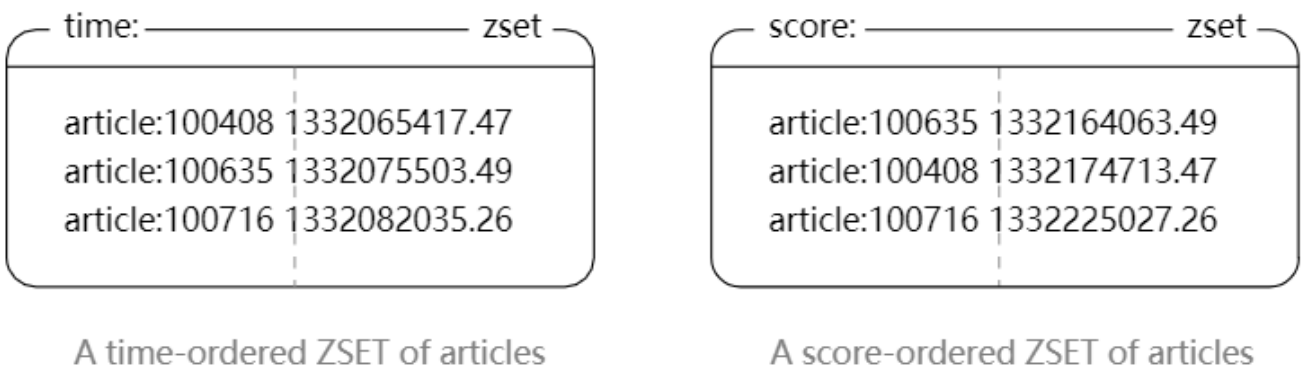



Figure 1.9 Two sorted sets representing time-ordered and score-ordered article indexes

# Redis 常见面试题

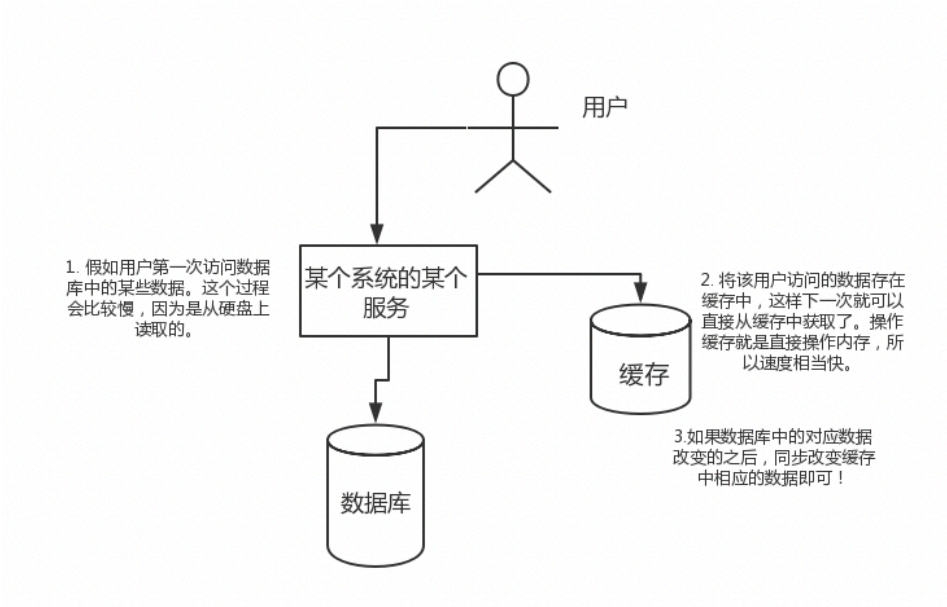
## Redis 简介

简单来说 redis 就是一个数据库，不过与传统数据库不同的是 redis 的数据是存在内存中的，所以读写速度非常快，因此 redis 被广泛应用于缓存方向。另外，redis 也经常用来做分布式锁。redis 提供了多种数据类型来支持不同的业务场景。除此之外，redis 支持事务 、持久化、LUA 脚本、LRU 驱动 事件、多种集群方案。

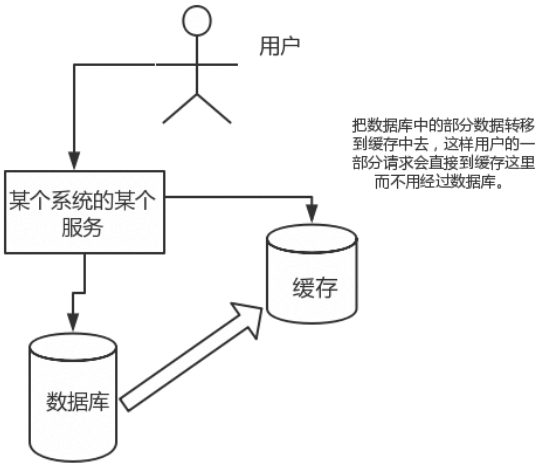
## 为什么要用 redis/为什么要用缓存？

主要从“高性能”和“高并发”这两点来看待这个问题。

**高性能:** 假如用户第一次访问数据库中的某些数据。这个过程会比慢，因为是从硬盘上读取的。将该用户访问的数据存在缓存中，这样下一次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，所以速度相当快。如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据即可!




**高并发:** 直接操作缓存能够承受的请求是远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。




## 为什么要用 redis 而不用 map/guava 做缓存?

缓存分为本地缓存和分布式缓存。以 Java 为例，使用自带的 map 或者 guava 实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况下，每个实例都 需要各自保存一份缓存，缓存不具有一致性。

使用 redis 或 memcached 之类的称为分布式缓存，在多实例的情况下，各实例共用一份缓存数据，缓存具有一致性。缺点是需要保持 redis 或 memcached 服务的高可用，整个程序架构上□为复杂。

## Redis 的线程模型

redis 内部使用文件事件处理器 file event handler，这个文件事件处理器是单线程的，所以 redis 才叫做单线程的模型。它采用 IO 多路复用机制同时监听多个 socket，根据 socket 上的事件来选择对应的事件处理器进行处理。

文件事件处理器的结构包含 4 个部分:

- 多个 socket;
- IO 多路复用程序;
- 文件事件分派器;
- 事件处理器(连接应答处理器、命令请求处理器、命令回复处理器) ;

多个 socket 可能会并发产生不同的操作，每个操作对应不同的文件事件，但是 IO 多路复用程序会监 听多个 socket，会将 socket 产生的事件放入队列中排队，事件分派器每次从队列中取出一个事件， 把该事件交给对应的事件处理器进行处理。

## redis 和 memcached 的区别

对于 redis 和 memcached 我总结了下面四点。现在公司一般都是用 redis 来实现缓存，而且 redis 自身也越来越强大了!

1. **redis 支持更丰富的数据类型(支持更复杂的应用场景)**:Redis 不仅仅支持简单的 k/v 类型的数据，同时还提供 list，set，zset，hash 等数据结构的存储。memcache 支持简单的数据类型，String。

2. **Redis 支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使 用,而 Memecache 把数据全部存在内存之中。**

3. **集群模式**:memcached 没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据;但 是 redis 目前是原生支持 cluster 模式的.

4. **Memcached 是多线程，非阻塞 IO 复用的网络模型;Redis 使用单线程的多路 IO 复用模型。**

| 对比参数 | Redis | Memcached |
| --- | --- | --- |
| 类型 | 1、支持内存<br>2、非关系型数据库 | 1、支持内存<br>2、key-value键值对形式<br>3、缓存系统 |
| 数据存储类型 | 1、String<br>2、List<br>3、Set<br>4、Hash<br>5、Sort Set 【俗称ZSet】 | 1、文本型<br>2、二进制类型【新版增加】 |
| 查询【操作】类型 | 1、批量操作<br>2、事务支持【虽然是假的事务】<br>3、每个类型不同的CRUD | 1、CRUD<br>2、少量的其他命令 |
| 附加功能 | 1、发布/订阅模式<br>2、主从分区<br>3、序列化支持<br>4、脚本支持【Lua脚本】 | 1、多线程服务支持 |
| 网络IO模型 | 1、单进程模式 | 2、多线程、非阻塞IO模式 |
| 事件库 | 自封装简易事件库AeEvent | 贵族血统的LibEvent事件库 |
| 持久化支持 | 1、RDB<br>2、AOF | 不支持 |

**redis 常见数据结构以及使用场景分析**

**String 常用命令: set,get,decr,incr,mget 等。**

String 数据结构是简单的 key-value 类型，value 其实不仅可以是 String，也可以是数字。

常规 key- value 缓存应用; 常规计数:微博数，粉丝数等。

**Hash 常用命令: hget,hset,hgetall 等。**

hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象，后续操作的时 候，你可以直接仅仅修改这个对象中的某个字段的值。 比如我们可以 hash 数据结构来存储用户信息，商品信息等等。比如下面我就用 hash 类型存放了我本人的一些信息:

```
key=JavaUser293847
value={
  “id”: 1,
  “name”: “SnailClimb”,
  “age”: 22,
  “location”: “Wuhan, Hubei”
}
```

**List 常用命令: lpush,rpush,lpop,rpop,lrange 等**

list 就是链表，Redis list 的应用场景非常多，也是 Redis 最重要的数据结构之一，比如微博的关注 列表，粉丝列表，消息列表等功能都可以用 Redis 的 list 结构来实现。

Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外 的内存开销。 另外可以通过 lrange 命令，就是从某个元素开始读取多少个元素，可以基于 list 实现分页查询，这 个很棒的一个功能，基于 redis 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西 (一页一页的往下走)，性能高。

**Set 常用命令: sadd,spop,smembers,sunion 等**

set 对外提供的功能与 list 类似是一个列表的功能，特殊之处在于 set 是可以自动排重的。

当你需要存储一个列表数据，又不希望出现重复数据时，set 是一个很好的选择，并且 set 提供了判断某 个成员是否在一个 set 集合内的重要接口，这个也是 list 所不能提供的。可以基于 set 轻易实现交集、 并集、差集的操作。

比如:在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。 Redis 可以非常方便的实现如共同关注、共同粉丝、共同喜好等功能。这个过程也就是求交集的过程， 具体命令如下:

`sinterstore key1 key2 key3` 将交集存在 key1 内

**Sorted Set 常用命令: zadd,zrange,zrem,zcard 等**

和 set 相比，sorted set 增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列。

举例: 在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息(可以理 解为按消息维度的消息排行榜)等信息，适合使用 Redis 中的 Sorted Set 结构进行存储。

**redis 设置过期时间**

Redis 中有个设置时间过期的功能，即对存储在 redis 数据库中的值可以设置一个过期时间。作为一个 缓存数据库，这是非常实用的。如我们一般项目中的 token 或者一些登录信息，尤其是短信验证码都 是有时间限制的，按照传统的数据库处理方式，一般都是自己判断过期，这样无疑会严重影响项目性 能。

我们 set key 的时候，都可以给一个 expire time，就是过期时间，通过过期时间我们可以指定这个 key 可以存活的时间。

如果假设你设置了一批 key 只能存活 1 个小时，那么接下来 1 小时后，redis 是怎么对这批 key 进行删除 的?

**定期删除+惰性删除。** 通过名字大概就能猜出这两个删除方式的意思了。

**定期删除**:redis 默认是每隔 100ms 就**随机抽取**一些设置了过期时间的 key，检查其是否过期， 如果过期就删除。注意这里是随机抽取的。为什么要随机呢?你想一想假如 redis 存了几十万 个 key ，每隔 100ms 就遍历所有的设置过期时间的 key 的话，就会给 CPU 带来很大的负载! **惰性删除** :定期删除可能会导致很多过期 key 到了时间并没有被删除掉。所以就有了惰性删 除。假如你的过期 key，靠定期删除没有被删除掉，还停留在内存里，除非你的系统去查一下那 个 key，才会被 redis 给删除掉。这就是所谓的惰性删除，也是够懒的哈!

但是仅仅通过设置过期时间还是有问题的。我们想一下:如果定期删除漏掉了很多过期 key，然后你也 没及时去查，也就没走惰性删除，此时会怎么样?如果大量过期 key 堆积在内存里，导致 redis 内存块耗 尽了。怎么解决这个问题呢? **redis 内存淘汰机制。**

**redis 内存淘汰机制(MySQL 里有 2000w 数据，Redis 中只存 20w 的数据，如何 保证 Redis 中的数据都是热点数据?)**

**redis 提供 6 种数据淘汰策略:**

1. **volatile-lru**:从已设置过期时间的数据集(server.db[i].expires)中挑选最近最少使用的数 据淘汰
2. **volatile-ttl**:从已设置过期时间的数据集(server.db[i].expires)中挑选将要过期的数据淘 汰
3. **volatile-random**:从已设置过期时间的数据集(server.db[i].expires)中任意选择数据淘汰
4. **allkeys-lru**:当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的 key(这个是最常用的)
5. **allkeys-random**:从数据集(server.db[i].dict)中任意选择数据淘汰
6. **no-eviction**:禁止驱逐数据，也就是说当内存不足以容纳新写入数据时，新写入操作会报错。这个应该没人使用吧!

4.0 版本后增加以下两种:

7. **volatile-lfu**:从已设置过期时间的数据集(server.db[i].expires)中挑选最不经常使用的数据 淘汰
8. **allkeys-lfu**:当内存不足以容纳新写入数据时，在键空间中，移除最不经常使用的 key

**redis 持久化机制(怎么保证 redis 挂掉之后再重启数据可以进行恢复)**

很多时候我们需要持久化数据也就是将内存中的数据写入到硬盘里面，大部分原因是为了之后重用数据 (比如重启机器、机器故障之后恢复数据)，或者是为了防止系统故障而将数据备份到一个远程位置。

Redis 不同于 Memcached 的很重一点就是，Redis 支持持久化，而且支持两种不同的持久化操作。**Redis 的 一种持久化方式叫快照(snapshotting，RDB)，另一种方式是只追加文件(append-only file,AOF)**。这两种方法各有千秋，下面我会详细这两种持久化方法是什么，怎么用，如何选择适合自 己的持久化方法。

**快照(snapshotting)持久化(RDB)**

Redis 可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。Redis 创建快照之后，可 以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本(Redis 主从结 构，主要用来提高 Redis 性能)，还可以将快照留在原地以便重启服务器的时候使用。快照持久化是 Redis 默认采用的持久化方式，在 redis.conf 配置文件中默认有此下配置:

```
save 900 1 #在 900 秒(15 分钟)之后，如果至少有 1 个 key 发生变化， Redis 就会自动触发 BGSAVE 命令创建快照。
save 300 10 #在 300 秒(5 分钟)之后，如果至少有 10 个 key 发生变化， Redis 就会自动触发 BGSAVE 命令创建快照。
save 60 10000 #在 60 秒(1 分钟)之后，如果至少有 10000 个 key 发生变化， Redis 就会自动触发 BGSAVE 命令创建快照。
```

**AOF(append-only file)持久化**

与快照持久化相比，AOF 持久化 的实时性更好，因此已成为主流的持久化方案。默认情况下 Redis 没有开启 AOF(append only file)方式的持久化，可以通过 appendonly 参数开启:

```
appendonly yes
```

开启 AOF 持久化后每执行一条会更改 Redis 中的数据的命令，Redis 就会将该命令写入硬盘中的 AOF 文件。 AOF 文件的保存位置和 RDB 文件的位置相同，都是通过 dir 参数设置的，默认的文件名是 appendonly.aof。

在 Redis 的配置文件中存在三种不同的 AOF 持久化方式，它们分别是:

```
appendfsync always #每次有数据修改发生时都会写入 A0F 文件,这样会严重降 低 Redis 的速度
appendfsync everysec #每秒钟同步一次，显示地将多个写命令同步到硬盘
appendfsync no #让操作系统决定何时进行同步
```

为了兼顾数据和写入性能，用户可以考虑 appendfsync everysec 选项 ，让 Redis 每秒同步一次 AOF 文 件，Redis 性能几乎没受到任何影响。而且这样即使出现系统崩溃，用户最多只会丢失一秒之内产生的 数据。当硬盘忙于执行写入操作的时候，Redis 还会优雅的放慢自己的速度以便适应硬盘的最大写入速 度。

**Redis 4.0 对于持久化机制的优化**

Redis 4.0 开始支持 RDB 和 AOF 的混合持久化(默认关闭，可以通过配置项 aof-use-rdb-preamble 开启)。 如果把混合持久化打开，AOF 重写的时候就直接把 RDB 的内容写到 AOF 文件开头。这样做的好处是可 以结合 RDB 和 AOF 的优点, 快速加载同时避免丢失过多的数据。当然缺点也是有的， AOF 里面的 RDB 部分是压缩格式不再是 AOF 格式，可读性□差。

**补充内容:AOF 重写** AOF 重写可以产生一个新的 AOF 文件，这个新的 AOF 文件和原有的 AOF 文件所保存的数据库状态一样，但体 积更小。

AOF 重写是一个有歧义的名字，该功能是通过读取数据库中的键值对来实现的，程序无须对现有 AOF 文件 进行任何读入、分析或者写入操作。

在执行 BGREWRITEAOF 命令时，Redis 服务器会维护一个 AOF 重写缓冲区，该缓冲区会在子进程创建 新 AOF 文件期间，记录服务器执行的所有写命令。当子进程完成创建新 AOF 文件的工作之后，服务器会将 重写缓冲区中的所有内容追加到新 AOF 文件的末尾，使得新旧两个 AOF 文件所保存的数据库状态一致。最 后，服务器用新的 AOF 文件替换旧的 AOF 文件，以此来完成 AOF 文件重写操作

## Redis 事务

Redis 通过 MULTI、EXEC、WATCH 等命令来实现事务(transaction)功能。事务提供了一种将多个命令 请求打包，然后一次性、按顺序地执行多个命令的机制，并且在事务执行期间，服务器不会中断事务而 改去执行其他客户端的命令请求，它会将事务中的所有命令都执行完毕，然后才去处理其他客户端的命 令请求。

在传统的关系式数据库中，常常用 ACID 性质来检验事务功能的可靠性和安全性。在 Redis 中，事务 总是具有原子性(Atomicity)、一致性(Consistency)和隔离性(Isolation)，并且当 Redis 运行在某种特定的持久化模式下时，事务也具有持久性(Durability)。

补充内容:

1. redis 同一个事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚。

## 缓存雪崩和缓存穿透问题解决方案

### 什么是缓存雪崩?

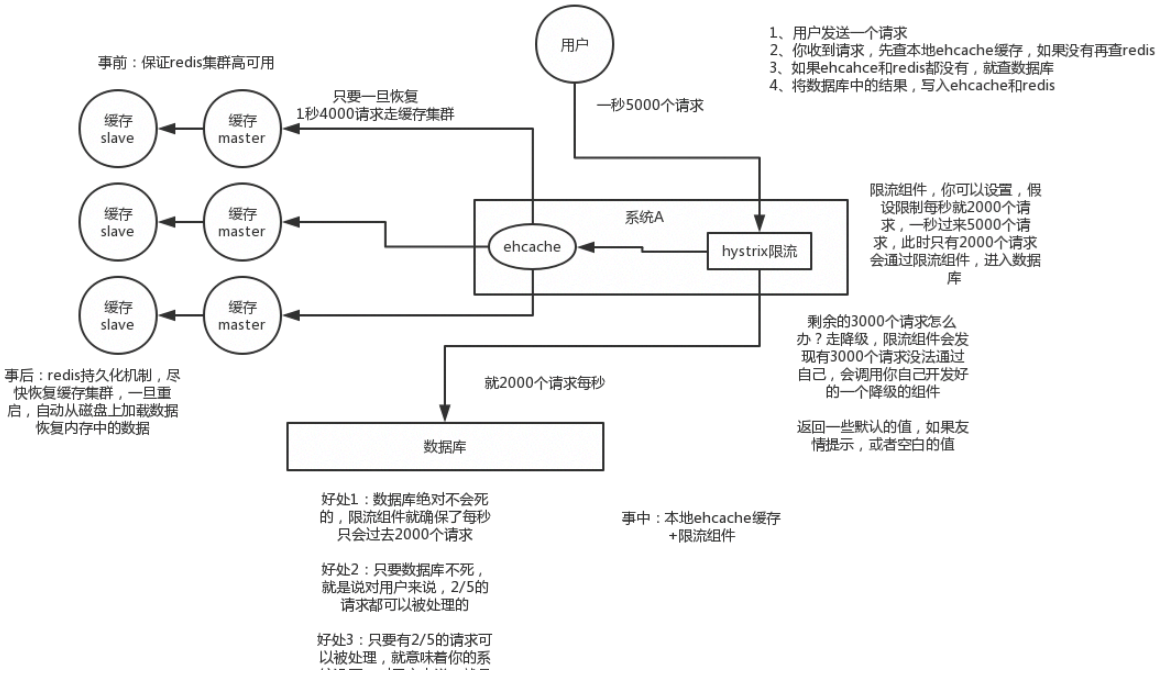
简介:缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

### 有哪些解决办法?

事前:尽量保证整个 redis 集群的高可用性，发现机器宕机尽快补上。选择合适的内存淘汰策略。

事中:本地 ehcache 缓存 + hystrix 限流&降级，避免 MySQL 崩掉

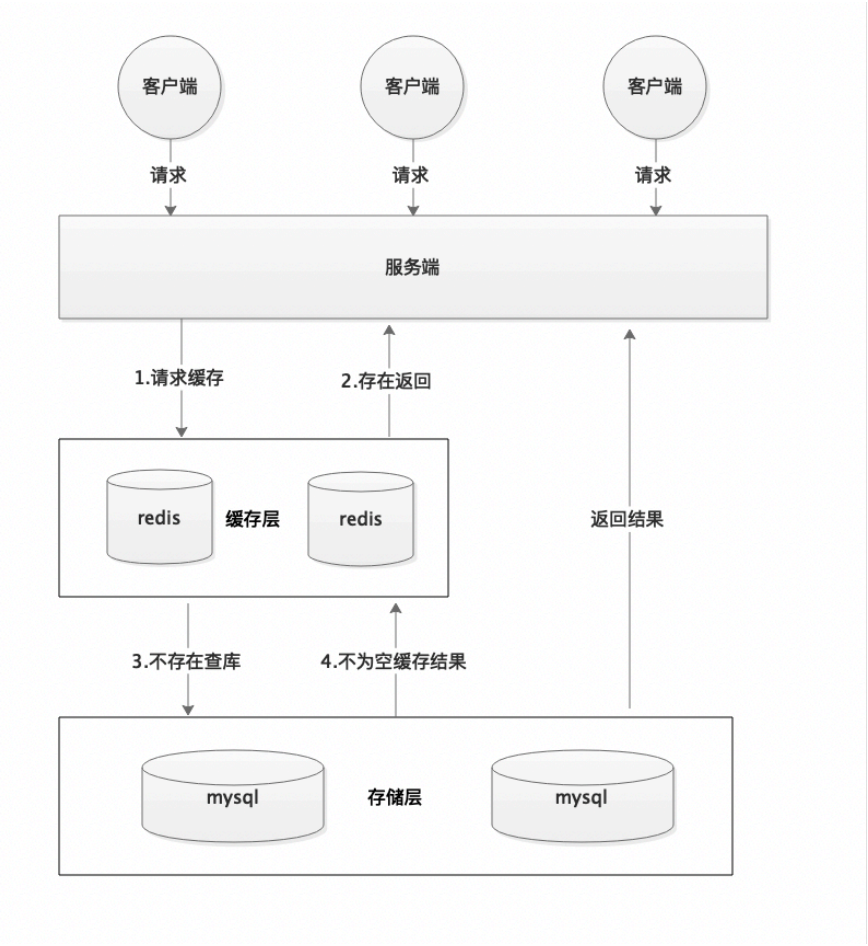
事后:利用 redis 持久化机制保存的数据尽快恢复缓存



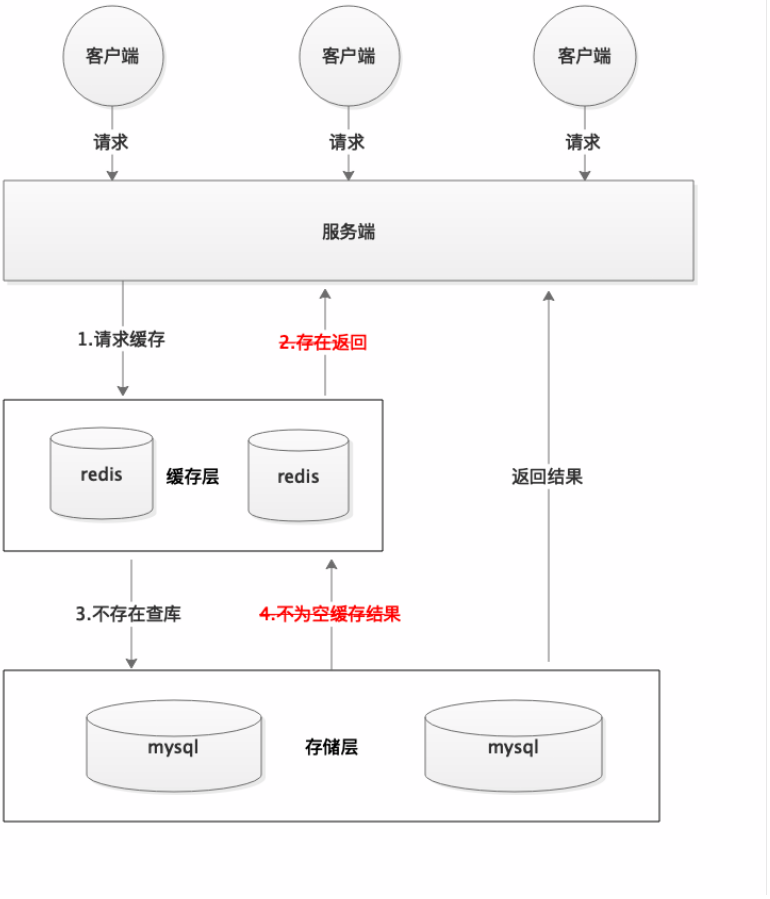

**什么是缓存穿透?**

缓存穿透说简单点就是大量请求的 key 根本不存在于缓存中，导致请求直接到了数据库上，根本没有经过缓存这一层。举个例子:某个黑客故意制造我们缓存中不存在的 key 发起大量请求，导致大量请 求落到数据库。下面用图片展示一下(这两张图片不是我画的，为了省事直接在网上找的，这里说明一 下):

**正常缓存处理流程:**


客户端
客户端
客户端
请求
请求
请求
服务端
1.请求缓存
2.存在返回
redis
缓存层
redis
返回结果
3.不存在查库
4.不为空缓存结果
mysql
存储层
mysql

缓存穿透情况处理流程：


客户端
客户端
客户端
请求
请求
请求
服务端
1.请求缓存
2.存在返回
redis
缓存层
redis
返回结果
3.不存在查库
4.不为空缓存结果
mysql
存储层
mysql

一般 MySQL 默认的最大连接数在 150 左右，这个可以通过 show variables like '%max_connections%'; 命令来查看。最大连接数一个还只是一个指标，cpu，内存，磁盘，网络等无力条件都是其运行指标，这些指标都会限制其并发能力!所以，一般 3000 个并发请求就能打死大部 分数据库了。

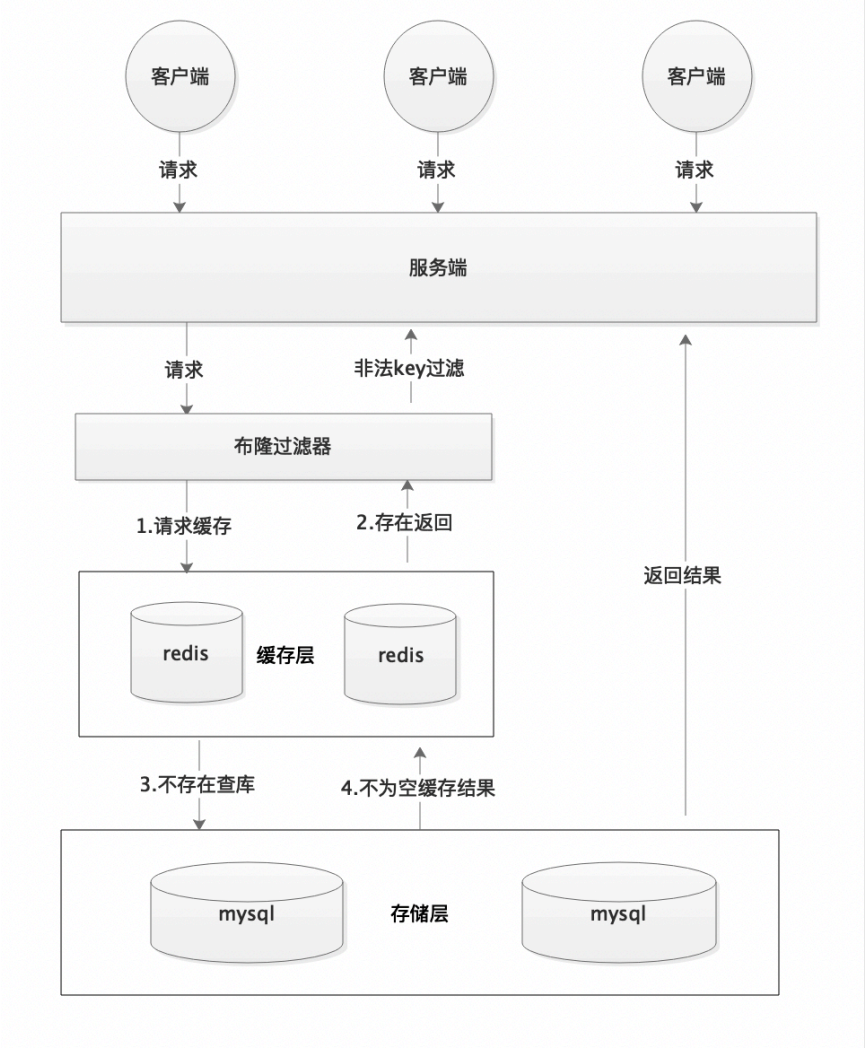
**有哪些解决办法?**

最基本的就是首先做好参数校验，一些不合法的参数请求直接抛出异常信息返回给客户端。比如查询的 数据库 id 不能小于 0、传入的邮箱格式不对的时候直接返回错误消息给客户端等等。

**(1)缓存无效 key**：如果缓存和数据库都查不到某个 key 的数据就写一个到 redis 中去并设置过期时 间，具体命令如下:SET key value EX 10086。这种方式可以解决请求的 key 变化不频繁的情况，如果黑客恶意攻击，每次构建不同的请求 key，会导致 redis 中缓存大量无效的 key 。很明显， 这种方案并不能从根本上解决此问题。如果非要用这种方式来解决穿透问题的话，尽量将无效的 key 的过期时间设置短一点比如 1 分钟。

另外，这里多说一嘴，一般情况下我们是这样设计 key 的: 表名:列名:主键名:主键值 。 如果用 Java 代码展示的话，差不多是下面这样的:

**(2)布隆过滤器:**布隆过滤器是一个非常神奇的数据结构，通过它我们可以非常方便地判断一个给定数 据是否存在与海量数据中。我们需要的就是判断 key 是否合法，有没有感觉布隆过滤器就是我们想要 找的那个“人”。具体是这样做的:把所有可能存在的请求的值都存放在布隆过滤器中，当用户请求过 来，我会先判断用户发来的请求的值是否存在于布隆过滤器中。不存在的话，直接返回请求参数错误信 息给客户端，存在的话才会走下面的流程。总结一下就是下面这张图



更多关于布隆过滤器的内容可以看这篇原创:《不了解布隆过滤器?一文给你整的明明白白!》
https://github.com/Snailclimb/JavaGuide/blob/master/docs/dataStructures-algorithms/data-structure/bloom-filter.md

## 如何解决 Redis 的并发竞争 Key 问题

所谓 Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，但是最后执行的顺序 和我们期望的顺序不同，这样也就导致了结果的不同!

推荐一种方案:分布式锁(zookeeper 和 redis 都可以实现分布式锁)。(如果不存在 Redis 的并发竞争 Key 问题，不要使用分布式锁，这样会影响性能)

基于 zookeeper 临时有序节点可以实现的分布式锁。大致思想为:每个客户端对某个方法加锁时，在 zookeeper 上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的 方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个瞬时节点删除即 可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。完成业务流程后，删除对应的 子节点释放锁。

在实践中，当然是从以可靠性为主。所以首推 Zookeeper。

## 如何保证缓存与数据库双写时的数据一致性?

一般情况下我们都是这样使用缓存的:先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。这种方式很明显会存在缓存和数据库的数据不一致的情况。

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题，那么你如何解决一致性问题?

一般来说，就是如果你的系统不是严格要求缓存+数据库必须一致性的话，缓存可以稍微的跟数据库偶 尔有不一致的情况，最好不要做这个方案，读请求和写请求串行化，串到一个内存队列里去，这样就可 以保证一定不会出现不一致的情况.

串行化之后，就会导致系统的吞吐量会大幅度的降低，用比正常情况下多几倍的机器去支撑线上的一个请求。