数据库和操作系统，这两部分内容我还是要看的。

~~加快效率，在放假前看完~~，然后假期要看计算机网络，放假前要把简历投递出去

# [MySQL中的事务](https://www.cnblogs.com/Soy-technology/p/11064382.html)

# 1. ****事务****

## 1.1. ****为什么需要事务****

现在的很多软件都是多用户，多程序，多线程的，对同一个表可能同时有很多人在用，为保持数据的一致性，所以提出了事务的概念。

A 给B 要划钱，A 的账户-1000元， B 的账户就要+1000元，这两个update 语句必须作为一个整体来执行，不然A 扣钱了，B 没有加钱这种情况很难处理。

数据库存储引擎：是数据库底层软件组织，数据库管理系统（DBMS）使用数据引擎进行创建、查询、更新和删除数据。不同的存储引擎提供不同的存储机制、索引技巧、锁定水平等功能，使用不同的存储引擎，还可以获得特定的功能。现在许多不同的数据库管理系统都支持多种不同的数据引擎。MySQL的核心就是插件式存储引擎。（默认InnoDB存储引擎，还有MyISAM存储引擎，MyISAM基于ISAM存储引擎，并对其进行扩展。它是在Web、数据仓储和其他应用环境下最常使用的存储引擎之一。MyISAM拥有较高的插入、查询速度，但不支持事务，不支持外键。MEMORY存储引擎将表中的数据存储到内存中，为查询和引用其他表数据提供快速访问。）

## 1.2. ****什么存储引擎支持事务****

1.查看数据库下面是否支持事务（InnoDB支持）？

  show engines;

2.查看mysql当前默认的存储引擎？

  show variables like '%storage\_engine%';

3.查看某张表的存储引擎？

  show create table 表名 ;

4.对于表的存储结构的修改？

建立InnoDB 表：Create table .... type=InnoDB； Alter table table\_name type=InnoDB;

**事务四大特性**

* 原子性：不可分割的操作单元，事务中所有操作，要么全部成功；要么撤回到执行事务之前的状态
* 一致性：如果在执行事务之前数据库是一致的，那么在执行事务之后数据库也还是一致的；
* 隔离性：事务操作之间彼此独立和透明互不影响。事务独立运行。这通常使用锁来实现。一个事务处理后的结果，影响了其他事务，那么其他事务会撤回。事务的100%隔离，需要牺牲速度。
* 持久性：事务一旦提交，其结果就是永久的。即便发生系统故障，也能恢复

**1.3.1. 原子性（atomicity）**

一个事务必须被视为一个不可分割的最小单元，整个事务中的所有操作要么全部提交成功，要么全部失败，对于一个事务来说，不可能只执行其中的一部分操作

老婆大人给Deer发生活费

1.老婆大人工资卡扣除500元

2.Deer工资卡增加500

整个事务要么全部成功，要么全部失败

1.3.2. **一致性（consistency）**

一致性是指事务将数据库从一种一致性转换到另外一种一致性状态，在事务开始之前和事务结束之后数据库中数据的完整性没有被破坏

老婆大人给Deer发生活费

1.老婆大人工资卡扣除500元

2.Deer工资卡增加500

2.Deer工资卡增加1000

扣除的钱（-500） 与增加的钱（500） 相加应该为0

1.3.3. **持久性（durability）**

一旦事务提交，则其所做的修改就会永久保存到数据库中。此时即使系统崩溃，已经提交的修改数据也不会丢失

并不是数据库的角度完全能解决

1.3.4. **隔离性（isolation）**

一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

（对数据库的并行执行，应该像串行执行一样）

**MySQL的事务隔离级别**

未提交读(Read Uncommitted)：允许脏读，其他事务只要修改了数据，即使未提交，本事务也能看到修改后的数据值。也就是可能读取到其他会话中未提交事务修改的数据

提交读(Read Committed)：只能读取到已经提交的数据。Oracle等多数数据库默认都是该级别 (不重复读)。

可重复读(Repeated Read)：可重复读。无论其他事务是否修改并提交了数据，在这个事务中看到的数据值始终不受其他事务影响。

串行读(Serializable)：完全串行化的读，每次读都需要获得表级共享锁，读写相互都会阻塞

**MySQL数据库(InnoDB引擎)默认使用可重复读（ Repeatable read)**

show variables like '%tx\_isolation%';

**1.3.4.1. 事务并发问题**

l 脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据

l 不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。

l 幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

***1.3.4.2. 未提交读（READ UNCOMMITED）脏读***

*show variables like '%tx\_isolation%';*

*set SESSION TRANSACTION ISOLATION LEVEL read UNCOMMITTED;*

*一个session中*

*start TRANSACTION*

*update account set balance = balance -50 where id = 1*

*另外一个session中查询*

*select \* from account*

*回到第一个session中 回滚事务*

*ROLLBACK*

*在第二个session中*

*select \* from account*

*在另外一个session中读取到了为提交的数据，这部分的数据为脏数据*

***1.3.4.3. 已提交读 （READ COMMITED）不可重复读***

*show variables like '%tx\_isolation%';*

*set SESSION TRANSACTION ISOLATION LEVEL read committed;*

*一个session中*

*start TRANSACTION*

*update account set balance = balance -50 where id = 1*

*另外一个session中查询 (数据并没改变)*

*select \* from account*

*回到第一个session中 回滚事务*

*commit*

*在第二个session中*

*select \* from account (数据已经改变)*

***1.3.4.4. 可重复读（REPEATABLE READ）***

*show variables like '%tx\_isolation%';*

*set SESSION TRANSACTION ISOLATION LEVEL repeatable read;*

*一个session中*

*start TRANSACTION*

*update account set balance = balance -50 where id = 1*

*另外一个session中查询 (数据并没改变)*

*select \* from account*

*回到第一个session中 回滚事务*

*commit*

*在第二个session中*

*select \* from account (数据并未改变)*

#### *1.3.4.5.******可串行化（SERIALIZABLE）*****

*show variables like '%tx\_isolation%';*

*set SESSION TRANSACTION ISOLATION LEVEL serializable;*

*1.开启一个事务*

*begin*

*select \* from account  发现3条记录*

*2.开启另外一个事务*

*begin*

*select \* from account  发现3条记录 也是3条记录*

*insert into account VALUES(4,'deer',500)  发现根本就不让插入*

*3.回到第一个事务 commit*

*在mysql中，可重复读已经解决了幻读问题，借助的就是间隙锁*

#### *1.1.1.1.******间隙锁（gap锁）*****

***实验1：（mysql默认可重复读，所以也就是会默认使用间隙锁）***

*select @@tx\_isolation;*

*create table t\_lock\_1 (a int primary key);*

*insert into t\_lock\_1 values(10),(11),(13),(20),(40);*

*begin*

*select \* from t\_lock\_1 where a <= 13 for update;*

*在另外一个会话中*

*insert into t\_lock\_1 values(21) 成功*

*insert into t\_lock\_1 values(19) 阻塞*

*在rr隔离级别中者个会扫描到当前值（13）的下一个值（20）,并把这些数据全部加锁*

***实验：2***

*create table t\_lock\_2 (a int primary key,b int, key (b));*

*insert into t\_lock\_2 values(1,1),(3,1),(5,3),(8,6),(10,8);*

*会话1*

*BEGIN*

*select \* from t\_lock\_2 where b=3 for update;*

*1 3 5 8 10*

*1 1 3 6 8*

*会话2*

*select \* from t\_lock\_2 where a = 5 lock in share mode; -- 不可执行，因为a=5上有一把记录锁*

*insert into t\_lock\_2 values(4, 2); -- 不可以执行，因为b=2在(1, 3]内*

*insert into t\_lock\_2 values(6, 5); -- 不可以执行，因为b=5在(3, 6)内*

*insert into t\_lock\_2 values(2, 0); -- 可以执行，(2, 0)均不在锁住的范围内*

*insert into t\_lock\_2 values(6, 7); -- 可以执行，(6, 7)均不在锁住的范围内*

*insert into t\_lock\_2 values(9, 6); -- 可以执行*

*insert into t\_lock\_2 values(7, 6); -- 不可以执行*

*二级索引锁住的范围是 (1, 3],（3， 6）*

*主键索引只锁住了a=5的这条记录 [5]*

## *1.1.******事务语法*****

### *1.1.1.******开启事务*****

*1、begin*

*2、START TRANSACTION（推荐）*

*3、begin work*

### *1.1.2.******事务回滚*****

*rollback*

### *1.1.3.******事务提交*****

*commit*

### *1.1.4.******还原点*****

*savepoint*

*show variables like '%autocommit%';  自动提交事务是开启的*

*set autocommit=0;    //关闭事务自动提交*

*insert into testdemo values(5,5,5);*

*savepoint s1;*

*insert into testdemo values(6,6,6);*

*savepoint s2;*

*insert into testdemo values(7,7,7);*

*savepoint s3;*

*select \* from testdemo*

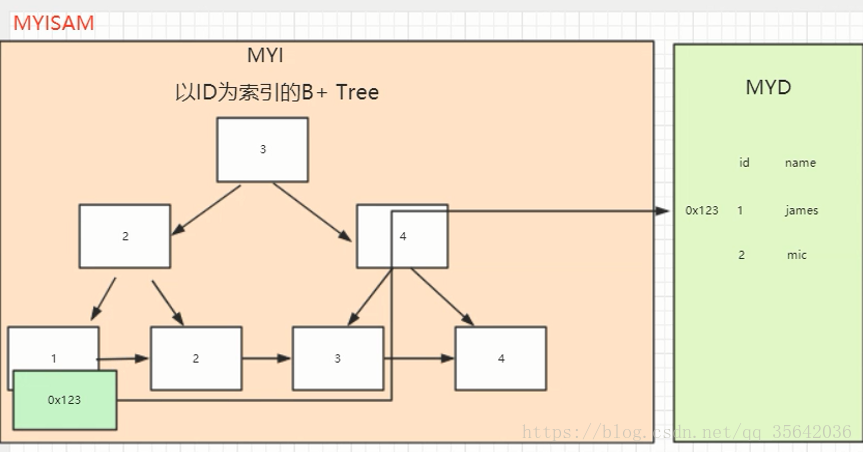
*rollback to savepoint s2*

*rollback*

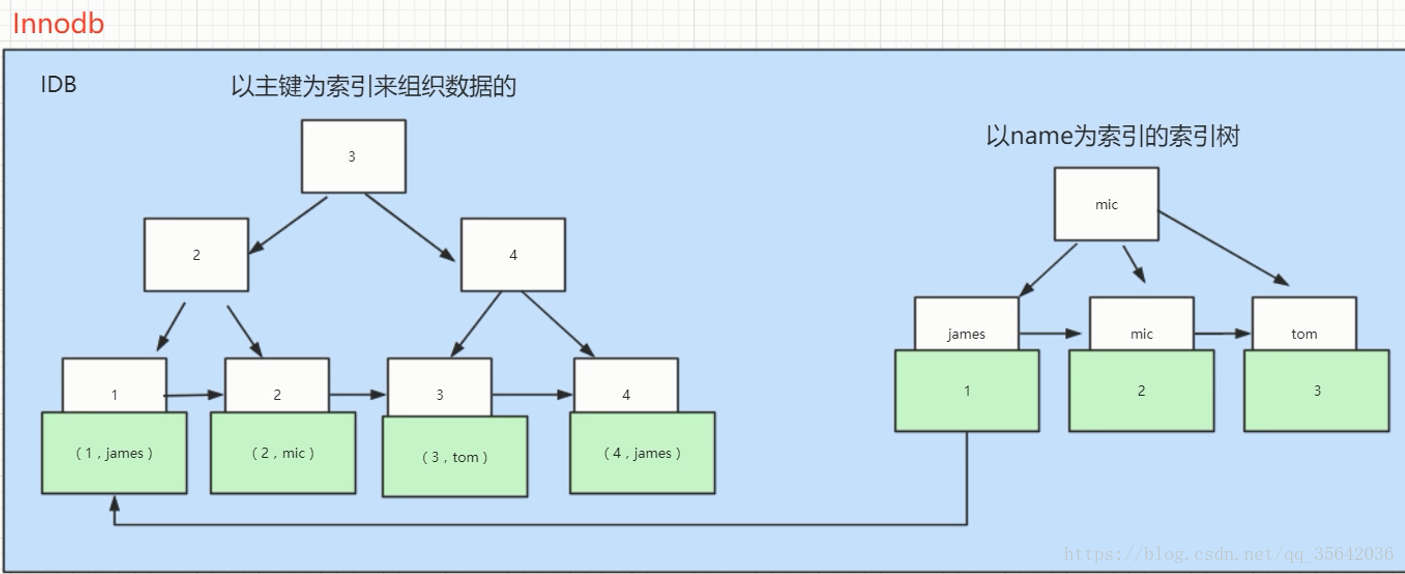
*commit过后rollback就没有意义了*

## 索引

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用 B\_TREE。B\_TREE 索引加速了数据访问，因为存储引擎不会再去扫描整张表得到需要的数据；相反，它从根节点开始，根节点保存了子节点的指针，存储引擎会根据指针快速寻找数据。  
  
**MyISAM引擎使用B+Tree作为索引结构，**叶节点的data域存放的是数据记录的地址，即：MyISAM索引文件和数据文件是分离的，MyISAM的索引文件仅仅保存数据记录的地址。MyISAM中索引检索的算法为首先按照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，读取相应数据记录。MyISAM的索引方式也叫做“非聚集”的。



**InnoDB引擎也使用B+Tree作为索引结构**，但是InnoDB的数据文件本身就是索引文件，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(索引是主键时data域才是完整数据记录)，因此InnoDB表数据文件本身就是主索引。这种索引叫做“聚焦索引”。InnoDB的辅助索引的data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域（当索引的key是非主键是索引文件中data存储的是主键而不是完整数据记录）。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。InnoDB的索引实现后，不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。在Innodb中也不建议使用非单调的字段作为主键，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，建议使用自增字段作为主键。



**MySQL数据库的四类索引:**

　　index  ----  普通索引,数据可以重复，没有任何限制。  
　　unique   ---- 唯一索引,要求索引列的值必须唯一，但允许有空值；如果是组合索引，那么列值的组合必须唯一。

　　primary key ---- 主键索引,是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值，一般是在创建表的同时创建主键索引。

**组合索引 ----**在多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。

　　fulltext ---- 全文索引,是对于大表的文本域：char，varchar，text列才能创建全文索引，主要用于查找文本中的关键字，并不是直接与索引中的值进行比较。fulltext更像是一个搜索引擎，配合match against操作使用，而不是一般的where语句加like。

　　注:全文索引目前只有MyISAM存储引擎支持全文索引，InnoDB引擎5.6以下版本还不支持全文索引

　　所有存储引擎对每个表至少支持16个索引，总索引长度至少为256字节，索引有两种存储类型，包括B型树索引和哈希索引。

　　索引可以提高查询的速度，但是创建和维护索引需要耗费时间，同时也会影响插入的速度，如果需要插入大量的数据时，最好是先删除索引，插入数据后再建立索引。

关于MySQL索引的好处，如果正确合理设计并且使用索引的MySQL是一辆兰博基尼的话，那么没有设计和使用索引的MySQL就是一个人力三轮车。对于没有索引的表，单表查询可能几十万数据就是瓶颈，而通常大型网站单日就可能会产生几十万甚至几百万的数据，没有索引查询会变的非常缓慢。还是以WordPress来说，其多个数据表都会对经常被查询的字段添加索引，比如wp\_comments表中针对5个字段设计了BTREE索引。

有了索引查找的效率是O（logn），没有索引查找的效率是O（n）

一、简单的对比测试

以我去年测试的数据作为一个简单示例，20多条数据源随机生成200万条数据，平均每条数据源都重复大概10万次，表结构比较简单，仅包含一个自增ID，一个char类型，一个text类型和一个int类型，单表2G大小，使用MyIASM引擎。开始测试未添加任何索引。

mysql> SELECT id,FROM\_UNIXTIME(time) FROM article WHERE a.title='测试标题'

查询需要的时间非常恐怖的，如果加上联合查询和其他一些约束条件，数据库会疯狂的消耗内存，并且会影响前端程序的执行。这时给title字段添加一个BTREE索引：

mysql> ALTER TABLE article ADD INDEX index\_article\_title ON title(200);

给title字段加上索引后，查询速度是立竿见影。

创建索引的sql语句

直接创建索引

CREATE INDEX index\_name ON table(column(length))

修改表结构的方式添加索引

ALTER TABLE table\_name ADD INDEX index\_name ON (column(length))

创建表的时候同时创建索引

CREATE TABLE `table` (

`id` int(11) NOT NULL AUTO\_INCREMENT ,

`title` char(255) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL ,

`content` text CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL ,

`time` int(10) NULL DEFAULT NULL ,

PRIMARY KEY (`id`),

INDEX index\_name (title(length))

)

删除索引

DROP INDEX index\_name ON table

2、唯一索引  
与普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须是唯一的，创建方法和普通索引类似。

创建唯一索引

CREATE UNIQUE INDEX index\_name ON table(column(length))

修改表结构

ALTER TABLE table\_name ADD UNIQUE INDEX index\_name ON (column(length))

创建表时同时创建索引

CREATE TABLE `table` (

`id` int(11) NOT NULL AUTO\_INCREMENT ,

`title` char(255) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL ,

`content` text CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL ,

`time` int(10) NULL DEFAULT NULL ,

PRIMARY KEY (`id`),

UNIQUE indexName (title(length))

);

3、组合索引  
平时用的SQL查询  
语句一般都有比较多的限制条件，所以为了进一步榨取MySQL的效率，就要考虑建立组合索引。例如上表中针对title和time建立一个组合索引：

ALTER TABLE article ADD INDEX index\_titme\_time (title(50),time(10))

建立这样的组合索引，其实是相当于分别建立了下面两组组合索引：

–title,time

–title

为什么没有time这样的组合索引呢？这是因为**MySQL组合索引**“**最左前缀**”的结果。  
**简单的理解就是只从最左面的开始组合。并不是只要包含这两列的查询都会用到该组合索引**，如下面的几个SQL所示：

使用到上面的索引

SELECT \* FROM article WHREE title='测试' AND time=1234567890;

SELECT \* FROM article WHREE utitle='测试';

未使用到上面的索引

SELECT \* FROM article WHREE time=1234567890;

四、索引的优化

上面说了使用索引的好处，但过多的使用索引将会造成滥用。因此索引也会有它的缺点：虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在一个大表上创建了多种组合索引，索引文件的会膨胀很快。索引只是提高效率的一个因素，如果你的MySQL有大数据量的表，就需要花时间研究建立最优秀的索引，或优化查询语句。下面是一些总结以及收藏的MySQL索引的注意事项和优化方法。

**1. 何时使用聚集索引或非聚集索引？**



事实上，我们可以通过前面聚集索引和非聚集索引的定义的例子来理解上表。如：返回某范围内的数据一项。比如您的某个表有一个时间列，恰好您把聚合索引建立在了该列，这时您查询2004年1月1日至2004年10月1日之间的全部数据时，这个速度就将是很快的，因为您的这本字典正文是按日期进行排序的，聚类索引只需要找到要检索的所有数据中的开头和结尾数据即可；而不像非聚集索引，必须先查到目录中查到每一项数据对应的页码，然后再根据页码查到具体内容。

2. 索引不会包含有NULL值的列

只要列中包含有NULL值都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为NULL。

3. 使用短索引

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个CHAR(255)的列，如果在前10个或20个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

4. 索引列排序

MySQL查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引。

5. like语句操作

一般情况下不鼓励使用like操作，如果非使用不可，如何使用也是一个问题。like “%aaa%” 不会使用索引而like “aaa%”可以使用索引。

6. 不要在列上进行运算

例如：select \* from users where YEAR(adddate)<2007，将在每个行上进行运算，这将导致索引失效而进行全表扫描，因此我们可以改成：select \* from users where adddate<’2007-01-01′。关于这一点可以围观：一个单引号引发的MYSQL性能损失。

最后总结一下，MySQL只对以下操作符才使用索引：<,<=,=,>,>=,between,in,以及某些时候的like(不以通配符%或\_开头的情形)。

而理论上每张表里面最多可创建16个索引，不过除非是数据量真的很多，否则过多的使用索引会引起反作用，索引虽好用，可不要太迷恋噢。

### ****索引生效条件****

假设index（a,b,c）

1. 最左前缀匹配：模糊查询时，使用%匹配时：’a%‘会使用索引，’%a‘不会使用索引
2. 条件中有or，索引不会生效
3. a and c，a生效，c不生效
4. b and c，都不生效
5. a and b > 5 and c,a和b生效，c不生效。

## sql语句分类：

1. DDL：数据定义语言（create drop）
2. DML：数据操作语句（insert update delete）
3. DQL：数据查询语句（select ）
4. DCL：数据控制语句，进行授权和权限回收（grant revoke）
5. TPL：数据事务语句（commit collback savapoint）

## 数据库三范式：

* 第一范式：1NF是对属性的原子性约束，要求字段具有原子性，不可再分解；(只要是关系型数据库都满足1NF)
* 第二范式：2NF是在满足第一范式的前提下，非主键字段不能出现部分依赖主键；解决：消除复合主键就可避免出现部分依赖，可增加单列关键字。
* 第三范式：3NF是在满足第二范式的前提下，非主键字段不能出现传递依赖，比如某个字段a依赖于主键，而一些字段依赖字段a，这就是传递依赖。解决：将一个实体信息的数据放在一个表内实现。

## 存储引擎 MyISAM和InnoDB区别：

1）InnoDB支持事务，MyISAM不支持。**对于InnoDB每一条SQL语言都默认封装成事务，自动提交，这样会影响速度，所以最好把多条SQL语言放在begin和commit之间，组成一个事务；**

2）MyISAM适合查询以及插入为主的应用，InnoDB适合频繁修改以及涉及到安全性较高的应用。

3）InnoDB支持外键，MyISAM不支持。**对一个包含外键的InnoDB表转为MYISAM会失败；**

4）从MySQL5.5.5以后，InnoDB是默认引擎。

5）MyISAM支持全文类型索引，而InnoDB不支持全文索引。**在涉及全文索引领域的查询效率上MyISAM速度更快高；PS：5.7以后的InnoDB支持全文索引了**

6）InnoDB中不保存表的总行数，select count(\*) from table时，InnoDB需要扫描整个表计算有多少行，但MyISAM只需简单读出保存好的总行数即可。注：当count(\*)语句包含where条件时MyISAM也需扫描整个表。

那么为什么InnoDB没有了这个变量呢？

因为InnoDB的事务特性，在同一时刻表中的行数对于不同的事务而言是不一样的，因此count统计会计算对于当前事务而言可以统计到的行数，而不是将总行数储存起来方便快速查询。InnoDB会尝试遍历一个尽可能小的索引除非优化器提示使用别的索引。如果二级索引不存在，InnoDB还会尝试去遍历其他聚簇索引。

如果索引并没有完全处于InnoDB维护的缓冲区（Buffer Pool）中，count操作会比较费时。可以建立一个记录总行数的表并让你的程序在INSERT/DELETE时更新对应的数据。和上面提到的问题一样，如果此时存在多个事务的话这种方案也不太好用。如果得到大致的行数值已经足够满足需求可以尝试SHOW TABLE STATUS

7）对于自增长的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中可以和其他字段一起建立联合索引。

8）清空整个表时，InnoDB是一行一行的删除，效率非常慢。MyISAM则会重建表。MyisAM使用delete语句删除后并不会立刻清理磁盘空间，需要定时清理，命令：OPTIMIZE table dept;

9）InnoDB支持行锁（某些情况下还是锁整表，如 update table set a=1 where user like ‘%lee%’）

10）Myisam创建表生成三个文件：.frm 数据表结构 、 .myd 数据文件 、 .myi 索引文件，Innodb只生成一个 .frm文件，数据存放在ibdata1.log

11）**InnoDB表必须有主键（用户没有指定的话会自己找或生产一个主键），而Myisam可以没有**

现在一般都选用InnoDB，主要是MyISAM的全表锁，读写串行问题，并发效率锁表，效率低，MyISAM对于读写密集型应用一般是不会去选用的。

应用场景：

* MyISAM不支持事务处理等高级功能，但它提供高速存储和检索，以及全文搜索能力（有一个全文索引）。如果应用中需要执行大量的SELECT查询，那么MyISAM是更好的选择。
* InnoDB用于需要事务处理的应用程序，包括ACID事务支持。如果应用中需要执行大量的INSERT或UPDATE操作，则应该使用InnoDB，这样可以提高多用户并发操作的性能。

如何选择：

1. 是否要支持事务，如果要请选择innodb，如果不需要可以考虑MyISAM；

2. 如果表中绝大多数都只是读查询，可以考虑MyISAM，如果既有读也有写，请使用InnoDB。

3. 系统奔溃后，MyISAM恢复起来更困难，能否接受；

4. MySQL5.5版本开始Innodb已经成为Mysql的默认引擎(之前是MyISAM)，说明其优势是有目共睹的，如果你不知道用什么，那就用InnoDB，至少不会差。

InnoDB为什么推荐使用自增ID作为主键？

答：自增ID可以保证每次插入时B+索引是从右边扩展的，可以避免B+树和频繁合并和分裂（对比使用UUID）。如果使用字符串主键和随机主键，会使得数据随机插入，效率比较差。

innodb引擎的4大特性

插入缓冲（insert buffer),二次写(double write),自适应哈希索引(ahi),预读(read ahead)

## CHAR和VARCHAR的区别：

* CHAR和VARCHAR类型在存储和检索方面有所不同
* CHAR列长度固定为创建表时声明的长度，长度值范围是1到255
* 当CHAR值被存储时，它们被用空格填充到特定长度，检索CHAR值时需删除尾随空格。

## Mysql中有哪几种锁？

* MyISAM支持表锁，InnoDB支持表锁和行锁，默认为行锁
* 表级锁：开销小，加锁快，不会出现死锁。锁定粒度大，发生锁冲突的概率最高，并发量最低
* 行级锁：开销大，加锁慢，会出现死锁。锁力度小，发生锁冲突的概率小，并发度最高

## 存储过程

我们常用的操作数据库语言SQL语句在执行的时候需要要先编译，然后执行，而存储过程（Stored Procedure）是一组为了完成特定功能的SQL语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给定参数（如果该存储过程带有参数）来调用执行它。

一个存储过程是一个可编程的函数，它在数据库中创建并保存。它可以有SQL语句和一些特殊的控制结构组成。当希望在不同的应用程序或平台上执行相同的函数，或者封装特定功能时，存储过程是非常有用的。数据库中的存储过程可以看做是对编程中面向对象方法的模拟。它允许控制数据的访问方式。

**优点：**

(1).存储过程增强了SQL语言的功能和灵活性。存储过程可以用流控制语句编写，有很强的灵活性，可以完成复杂的判断和较复杂的运算。

(2).存储过程允许标准组件是编程。存储过程被创建后，可以在程序中被多次调用，而不必重新编写该存储过程的SQL语句。而且数据库专业人员可以随时对存储过程进行修改，对应用程序源代码毫无影响。

(3).存储过程能实现较快的执行速度。如果某一操作包含大量的Transaction-SQL代码或分别被多次执行，那么存储过程要比批处理的执行速度快很多。因为存储过程是预编译的。在首次运行一个存储过程时查询，优化器对其进行分析优化，并且给出最终被存储在系统表中的执行计划。而批处理的Transaction-SQL语句在每次运行时都要进行编译和优化，速度相对要慢一些。

(4).存储过程能过减少网络流量。针对同一个数据库对象的操作（如查询、修改），如果这一操作所涉及的Transaction-SQL语句被组织程存储过程，那么当在客户计算机上调用该存储过程时，网络中传送的只是该调用语句，从而大大增加了网络流量并降低了网络负载。

(5).存储过程可被作为一种安全机制来充分利用。系统管理员通过执行某一存储过程的权限进行限制，能够实现对相应的数据的访问权限的限制，避免了非授权用户对数据的访问，保证了数据的安全。

## delete、drop、truncate区别

* truncate 和 delete只删除数据，不删除表结构 ,drop删除表结构，并且释放所占的空间。
* **删除数据的速度，**drop> truncate > delete
* delete属于DML语言，需要事务管理，commit之后才能生效。**DELETE**语句执行删除的过程是每次从表中删除一行，并且同时**将该行的删除操作作为事务记录在日志中保存以便进行进行回滚操作**。drop和truncate属于DDL语言，操作立刻生效，不可回滚。TRUNCATE TABLE 则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器。执行速度快。
* 表和索引所占空间。当表被TRUNCATE 后，这个表和索引所占用的空间会恢复到初始大小，DELETE操作不会减少表或索引所占用的空间。 drop语句将表所占用的空间全释放掉。
* **使用场合：**
  + 当你不再需要该表时， 用 drop;
  + 当你仍要保留该表，但要删除所有记录时， 用 truncate;
  + 当你要删除部分记录时（always with a where clause), 用 delete.
* **注意：** 对于**有主外键关系的表**，不能使用truncate而应该**使用不带where子句的delete语句**，由于truncate不记录在日志中，不能够激活触发器
* TRUNCATE 只能对TABLE； DELETE可以是table和view

# [数据库--视图的基本概念以及作用](https://www.cnblogs.com/yadongliang/p/8981801.html)

**转自:**[**数据库--视图的基本概念以及作用**](https://blog.csdn.net/buhuikanjian/article/details/53105416)

视图（子查询）：是从一个或多个表导出的虚拟的表，其内容由查询定义。具有普通表的结构，但是不实现数据存储。  
对视图的修改：单表视图一般用于查询和修改，会改变基本表的数据，  
多表视图一般用于查询，不会改变基本表的数据。

--创建视图--

create or replace view v\_student as select \* from student;

--从视图中检索数据--

select \* from v\_student;

--删除视图--

drop view v\_student;

①简化了操作，把经常使用的数据定义为视图。

我们在使用查询时，在很多时候我们要使用聚合函数，同时还要 显示其它字段的信息，可能还会需要关联到其它表，这时写的语句可能会很长，如果这个动作频繁发生的话，我们可以创建视图，这以后，我们只需要select \* from view就可以啦，这样很方便。

②安全性，用户只能查询和修改能看到的数据。

因为视图是虚拟的，物理上是不存在的，只是存储了数据的集合，我们可以将基表中重要的字段信息，可以不通过视图给用户，视图是动态的数据的集合，数据是随着基表的更新而更新。同时，用户对视图不可以随意的更改和删除，可以保证数据的安全性。

③逻辑上的独立性，屏蔽了真实表的结构带来的影响。

视图可以使应用程序和数据库表在一定程度上独立。如果没有视图，应用一定是建立在表上的。有了视图之后，程序可以建立在视图之上，从而程序与数据库表被视图分割开来。

缺点:

①性能差

数据库必须把视图查询转化成对基本表的查询，如果这个视图是由一个复杂的多表查询所定义，那么，即使是视图的一个简单查询，数据库也要把它变成一个复杂的结合体，需要花费一定的时间。

②修改限制

当用户试图修改视图的某些信息时，数据库必须把它转化为对基本表的某些信息的修改，对于简单的视图来说，这是很方便的，但是，对于比较复杂的试图，可能是不可修改的。

在定义数据库对象时，不能不加选择地来定义视图，应该权衡视图的优点和缺点，合理地定义视图。

视图（子查询）：是从一个或多个表导出的虚拟的表，其内容由查询定义。

具有普通表的结构，但是不实现数据存储。

对视图的修改：单表视图一般用于查询和修改，会改变基本表的数据，多表视图一般用于查询，不会改变基本表的数据。

作用：

①简化了操作，把经常使用的数据定义为视图。

②安全性，用户只能查询和修改能看到的数据。

③逻辑上的独立性，屏蔽了真实表的结构带来的影响。

Mysql存储引擎具备的四大特性

innodb引擎的4大特性

插入缓冲（insert buffer),二次写(double write),自适应哈希索引(ahi),预读(read ahead)

**插入缓冲**

插入缓冲是InnoDB存储引擎关键特性中最令人激动的。不过，这个名字可能会让人认为插入缓冲是缓冲池中的一个部分。其实不然，InnoDB缓冲池中有Insert Buffer信息固然不错，但是Insert Buffer和数据页一样，也是物理页的一个组成部分。

主键是行唯一的标识符，在应用程序中行记录的插入顺序是按照主键递增的顺序进行插入的。因此，插入聚集索引一般是顺序的，不需要磁盘的随机读取。

比如说我们按下列SQL定义的表：create table t（id int auto\_increment,name varchar(30),primary key(id));

id列是自增长的，这意味着当执行插入操作时，id列会自动增长，页中的行记录按id执行顺序存放。一般情况下，不需要随机读取另一页执行记录的存放。因此，在这样的情况下，插入操作一般很快就能完成。但是，不可能每张表上只有一个聚集索引，在更多的情况下，一张表上有多个非聚集的辅助索引（secondary index）。比如，我们还需要按照name这个字段进行查找，并且name这个字段不是唯一的。

表是按如下的SQL语句定义的：create table t (id int auto\_increment,name varchar(30)，primary key(id),key(name));

这样的情况下产生了一个非聚集的并且不是唯一的索引。在进行插入操作时，数据页的存放还是按主键id的执行顺序存放，但是对于非聚集索引，叶子节点的插入不再是顺序的了。这时就需要离散地访问非聚集索引页，插入性能在这里变低了。然而这并不是这个name字段上索引的错误，因为B+树的特性决定了非聚集索引插入的离散性。

InnoDB存储引擎开创性地设计了插入缓冲，对于非聚集索引的插入或更新操作，不是每一次直接插入索引页中，而是先判断插入的非聚集索引页是否在缓冲池中。如果在，则直接插入；如果不在，则先放入一个插入缓冲区中，好似欺骗数据库这个非聚集的索引已经插到叶子节点了，然后再以一定的频率执行插入缓冲和非聚集索引页子节点的合并操作，这时通常能将多个插入合并到一个操作中（因为在一个索引页中），这就大大提高了对非聚集索引执行插入和修改操作的性能。

**插入缓冲的使用需要满足以下两个条件：**

**1.索引是辅助索引。**

**2.索引不是唯一的。**

当满足以上两个条件时，InnoDB存储引擎会使用插入缓冲，这样就能提高性能了。不过考虑一种情况，应用程序执行大量的插入和更新操作，这些操作都涉及了不唯一的非聚集索引，如果在这个过程中数据库发生了宕机，这时候会有大量的插入缓冲并没有合并到实际的非聚集索引中。如果是这样，恢复可能需要很长的时间，极端情况下甚至需要几个小时来执行合并恢复操作。

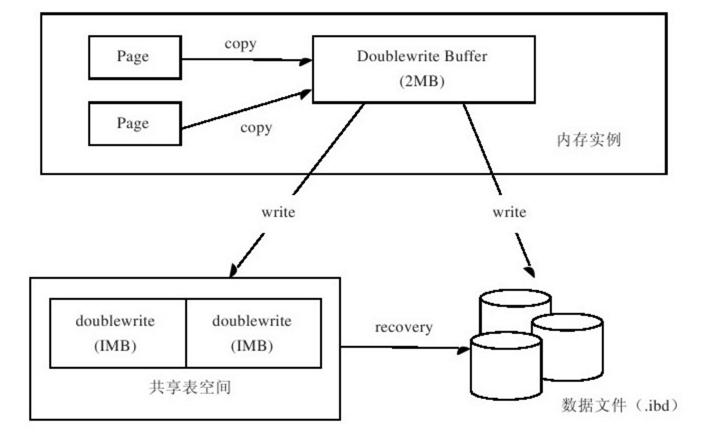
辅助索引不能是唯一的，因为在把它插入到插入缓冲时，我们并不去查找索引页的情况。如果去查找肯定又会出现离散读的情况，插入缓冲就失去了意义。

**两次写**

如果说插入缓冲带给InnoDB存储引擎的是性能，那么两次写带给InnoDB存储引擎的是数据的可靠性。当数据库宕机时，可能发生数据库正在写一个页面，而这个页只写了一部分（比如16K的页，只写前4K的页）的情况，我们称之为部分写失效（partial page write）。在InnoDB存储引擎未使用double write技术前，曾出现过因为部分写失效而导致数据丢失的情况。

有人也许会想，如果发生写失效，可以通过重做日志进行恢复。这是一个办法。但是必须清楚的是，重做日志中记录的是对页的物理操作，如偏移量800，写'aaaa'记录。如果这个页本身已经损坏，再对其进行重做是没有意义的。这就是说，在应用（apply）重做日志前，我们需要一个页的副本，当写入失效发生时，先通过页的副本来还原该页，再进行重做，这就是doublewrite。

InnoDB存储引擎doublewrite的体系架构如图所示



doublewrite由两部分组成：一部分是内存中的doublewrite buffer，大小为2MB；另一部分是物理磁盘上共享表空间中连续的128个页，即两个区（extent），大小同样为2MB(页的副本)。当缓冲池的脏页刷新时，并不直接写磁盘，而是会通过memcpy函数将脏页先拷贝到内存中的doublewrite buffer，之后通过doublewrite buffer再分两次，每次写入1MB到共享表空间的物理磁盘上，然后马上调用fsync函数，同步磁盘，避免缓冲写带来的问题。在这个过程中，因为doublewrite页是连续的，因此这个过程是顺序写的，开销并不是很大。在完成doublewrite页的写入后，再将doublewrite buffer中的页写入各个表空间文件（图中的数据文件）中，此时的写入则是离散的。

参数skip\_innodb\_doublewrite可以禁止使用两次写功能，这时可能会发生前面提及的写失效问题。不过，如果你有多台从服务器（slave server），需要提供较快的性能（如slave上做的是RAID0），也许启用这个参数是一个办法。不过，在需要提供数据高可靠性的主服务器（master server）上，任何时候我们都应确保开启两次写功能。

注意：有些文件系统本身就提供了部分写失效的防范机制，如ZFS文件系统。在这种情况下，我们就不要启用doublewrite了。

**自适应哈希索引**

哈希（hash）是一种非常快的查找方法，一般情况下查找的时间复杂度为O(1)。常用于连接（join）操作，如SQL Server和Oracle中的哈希连接（hash join）。但是SQL Server和Oracle等常见的数据库并不支持哈希索引（hash index）。MySQL的Heap存储引擎默认的索引类型为哈希，而InnoDB存储引擎提出了另一种实现方法，自适应哈希索引（adaptive hash index）。

InnoDB存储引擎会监控对表上索引的查找，如果观察到建立哈希索引可以带来速度的提升，则建立哈希索引，所以称之为自适应（adaptive）的。自适应哈希索引通过缓冲池的B+树构造而来，因此建立的速度很快。而且不需要将整个表都建哈希索引，InnoDB存储引擎会自动根据访问的频率和模式来为某些页建立哈希索引。

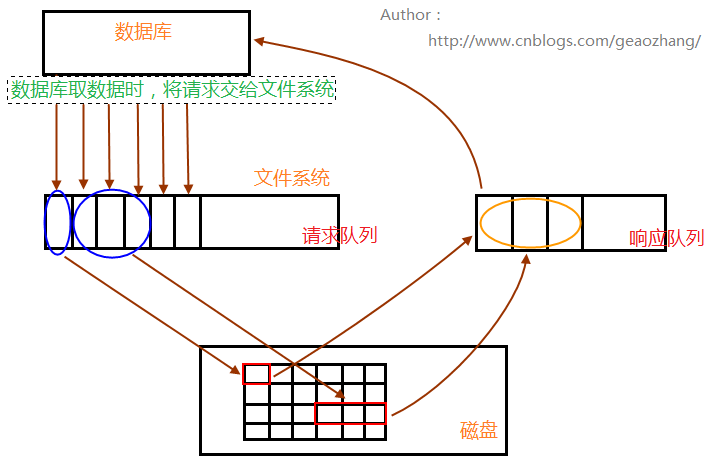
根据InnoDB的官方文档显示，启用自适应哈希索引后，读取和写入速度可以提高2倍；对于辅助索引的连接操作，性能可以提高5倍。自适应哈希索引是非常好的优化模式，其设计思想是数据库自优化（self-tuning），即无需DBA对数据库进行调整。

现在可以看到自适应哈希索引的使用信息了，包括自适应哈希索引的大小、使用情况、每秒使用自适应哈希索引搜索的情况。值得注意的是，哈希索引只能用来搜索等值的查询，如select \* from table where index\_col='xxx'，而对于其他查找类型，如范围查找，是不能使用的。因此，这里出现了non-hash searches/s的情况。用hash searches:non-hash searches命令可以大概了解使用哈希索引后的效率。

由于自适应哈希索引是由InnoDB存储引擎控制的，所以这里的信息只供我们参考。不过我们可以通过参数innodb\_adaptive\_hash\_index来禁用或启动此特性，默认为开启。

预读

InnoDB在I/O的优化上有个比较重要的特性为预读，预读请求是一个i/o请求，它会异步地在缓冲池中预先回迁多个页面，预计很快就会需要这些页面，这些请求在一个范围内引入所有页面。InnoDB以64个page为一个extent，那么InnoDB的预读是以page为单位还是以extent？



数据库请求数据的时候，会将读请求交给文件系统，放入请求队列中；相关进程从请求队列中将读请求取出，根据需求到相关数据区(内存、磁盘)读取数据；取出的数据，放入响应队列中，最后数据库就会从响应队列中将数据取走，完成一次数据读操作过程。

　　接着进程继续处理请求队列，(如果数据库是全表扫描的话，数据读请求将会占满请求队列)，判断后面几个数据读请求的数据是否相邻，再根据自身系统IO带宽处理量，进行预读，进行读请求的合并处理，一次性读取多块数据放入响应队列中，再被数据库取走。(如此，一次物理读操作，实现多页数据读取，rrqm>0（# iostat -x），假设是4个读请求合并，则rrqm参数显示的就是4)

1、线性预读（linear read-ahead）

线性预读方式有一个很重要的变量控制是否将下一个extent预读到buffer pool中，通过使用配置参数innodb\_read\_ahead\_threshold，控制触发innodb执行预读操作的时间。

如果一个extent中的被顺序读取的page超过或者等于该参数变量时，Innodb将会异步的将下一个extent读取到buffer pool中，innodb\_read\_ahead\_threshold可以设置为0-64的任何值(因为一个extent中也就只有64页)，默认值为56，值越高，访问模式检查越严格。

mysql> show variables like 'innodb\_read\_ahead\_threshold';

+--------------------------------------+-------+

| Variable\_name | Value |

+--------------------------------------+-------+

| innodb\_read\_ahead\_threshold | 56 |

+--------------------------------------+-------+

例如，如果将值设置为48，则InnoDB只有在顺序访问当前extent中的48个pages时才触发线性预读请求，将下一个extent读到内存中。如果值为8，InnoDB触发异步预读，即使程序段中只有8页被顺序访问。

　　可以在MySQL配置文件中设置此参数的值，或者使用SET GLOBAL需要该SUPER权限的命令动态更改该参数。

　　在没有该变量之前，当访问到extent的最后一个page的时候，innodb会决定是否将下一个extent放入到buffer pool中。

2、随机预读（randomread-ahead）

　　随机预读方式则是表示当同一个extent中的一些page在buffer pool中发现时，Innodb会将该extent中的剩余page一并读到buffer pool中。

mysql> show variables like 'innodb\_random\_read\_ahead';

+----------------------------------+-------+

| Variable\_name | Value |

+-----------------------------------+-------+

| innodb\_random\_read\_ahead | OFF |

+------------------------------------+-------+

由于随机预读方式给innodb code带来了一些不必要的复杂性，同时在性能也存在不稳定性，在5.5中已经将这种预读方式废弃，默认是OFF。若要启用此功能，即将配置变量设置innodb\_random\_read\_ahead为ON。

**三、监控Innodb的预读**

1、可以通过show engine innodb status\G显示统计信息

mysql> show engine innodb status\G

----------------------

BUFFER POOL AND MEMORY

----------------------

……

Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s

……

　　1、Pages read ahead：表示每秒读入的pages；

　　2、evicted without access：表示每秒读出的pages；

　　3、一般随机预读都是关闭的，也就是0。

2、通过两个状态值，评估预读算法的有效性

mysql> show global status like '%read\_ahead%';

+----------------------------------------------------+-------+

| Variable\_name | Value |

+----------------------------------------------------+-------+

| Innodb\_buffer\_pool\_read\_ahead\_rnd | 0 |

| Innodb\_buffer\_pool\_read\_ahead | 2303 |

| Innodb\_buffer\_pool\_read\_ahead\_evicted | 0 |

+-------------------------------------------------- --+-------+

3 rows in set (0.01 sec)

1、Innodb\_buffer\_pool\_read\_ahead：通过预读(后台线程)读入innodb buffer pool中数据页数

　　2、Innodb\_buffer\_pool\_read\_ahead\_evicted：通过预读来的数据页没有被查询访问就被清理的pages，无效预读页数

# [Redis常见面试题](https://www.cnblogs.com/jasontec/p/9699242.html)

介绍：Redis 是一个开源的使用 ANSI C 语言编写、遵守 BSD 协议、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，并提供多种语言的 API的非关系型数据库。

传统数据库遵循 ACID 规则。而 Nosql（Not Only SQL 的缩写，是对不同于传统的关系型数据库的数据库管理系统的统称） 一般为分布式而分布式一般遵循 CAP 定理。

**Redis支持的数据类型？**

String字符串：

格式: set key value

string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。

string类型是Redis最基本的数据类型，一个键最大能存储512MB。

Hash（哈希）

格式: hmset name  key1 value1 key2 value2

Redis hash 是一个键值(key=>value)对集合。

Redis 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。

List（列表）

Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）

格式: lpush  name  value

在 key 对应 list 的头部添加字符串元素

格式: rpush  name  value

在 key 对应 list 的尾部添加字符串元素

格式: lrem name  index

key 对应 list 中删除 count 个和 value 相同的元素

格式: llen name

返回 key 对应 list 的长度

Set（集合）

格式: sadd  name  value

Redis的Set是string类型的无序集合。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

zset(sorted set：有序集合)

格式: zadd  name score value

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

zset的成员是唯一的,但分数(score)却可以重复。

**什么是Redis持久化？Redis有哪几种持久化方式？优缺点是什么？**

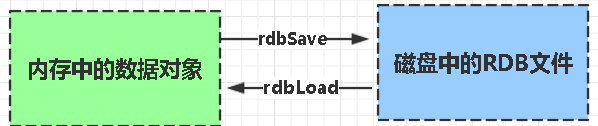
持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

Redis 提供了两种持久化方式:RDB（默认） 和AOF

**RDB：**

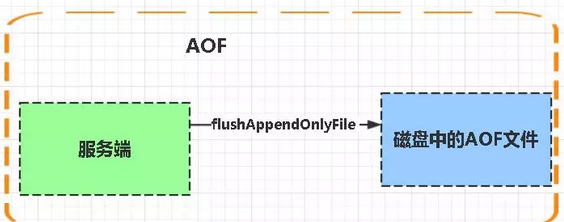
rdb是Redis DataBase缩写

功能核心函数rdbSave(生成RDB文件)和rdbLoad（从文件加载内存）两个函数



**AOF:**

Aof是Append-only file缩写



每当执行服务器(定时)任务或者函数时flushAppendOnlyFile 函数都会被调用， 这个函数执行以下两个工作

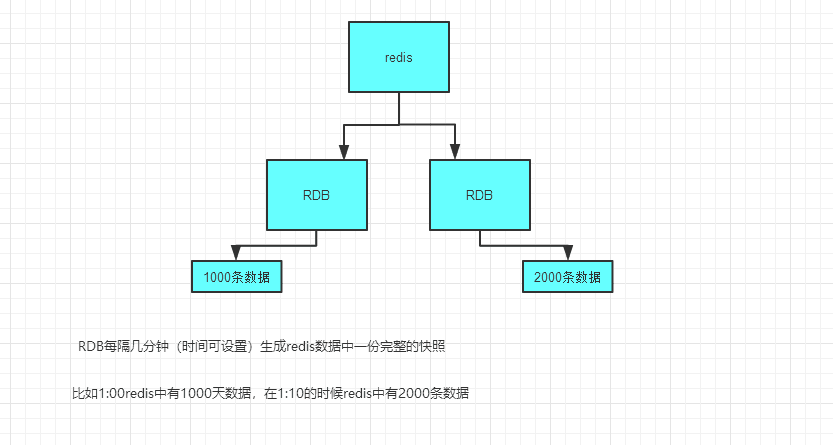
aof写入保存：

WRITE：根据条件，将 aof\_buf 中的缓存写入到 AOF 文件

SAVE：根据条件，调用 fsync 或 fdatasync 函数，将 AOF 文件保存到磁盘中。

首先要先说下redis持久化的意义：  
redis持久化的意义主要在于故障恢复，比如你部署一个redis，作为缓存有可能里边有一些比较重要的数据，如果没有持久化的时候，redis遇到灾难性故障的时候就会丢失所有的数据。  
多以持久化是必不可少的。

1.RDB和AOF两种持久化机制的介绍  
RDB持久化机制对redis中的数据执行周期性的持久化。  
AOF持久化机制对每条写入命令作为日志，以append-only模式写入一个日志文件中，在redis重启的时候，可以通过AOF写入的指令来重新构建整个数据集。  
通过RDB和AOF都可以将redis内存中的数据持久化到硬盘上，然后可以将数据备份到云服务器上。  
如果redis挂了可以从云服务器上的备份文件copy到指定位置然后重启redis，redis就会自动持久化文件中的数据，去恢复内存中的数据。  
如果同时使用RDB和AOF两种持久化机制，那么redis重启的时候，会使用AOF来构建数据，因为AOF的数据更加完整。

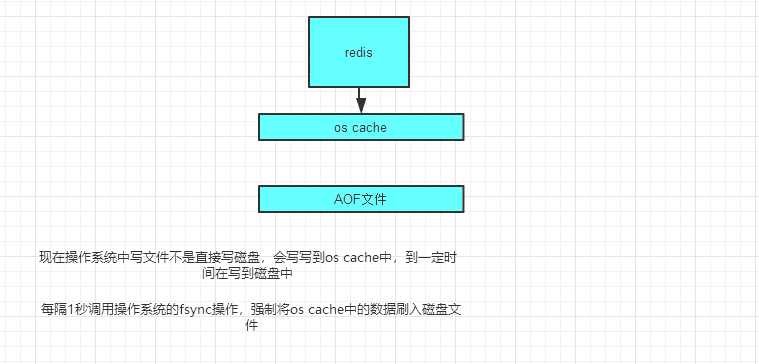


当满足条件时,redis需要执行RDB的时候服务器会执行以下操作：  
1.redis调用系统的fork()函数创建一个子进程  
2.子进程将数据集写入一个临时的RDB文件  
3.当子进程完成对临时的RDB文件的写入时，redis用新的RDB文件来替换原来旧的RDB文件，并将旧的RDB文件删除

redis在进行快照的过程中不会对RDB文件进行修改，只有快照结束后才会将旧快照替换成新快照，也就是说任何时候RDB都是完整的

RDB优点：  
（1）RDB会生成多个数据文件，每个数据文件都代表了某一个时刻中redis的数据，这种多个数据文件的方式，非常适合做冷备。  
（2）RDB对redis对外提供读写服务的时候，影像非常小，因为redis 主进程只需要fork一个子进程出来，让子进程对磁盘io来进行rdb持久化  
(3).RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

RDB缺点  
（1）如果redis要故障时要尽可能少的丢失数据，RDB没有AOF好，例如1:00进行的快照，在1:10又要进行快照的时候宕机了，这个时候就会丢失10分钟的数据。  
（2）RDB每次fork出子进程来执行RDB快照生成文件时，如果文件特别大，可能会导致客户端提供服务暂停数毫秒或者几秒



redis中的数据是有一定限量的，不可能说redis中的数据无限增长，进而导致AOF文件无限增长。  
内存大小是一定的，等到了一定大小redis 会采用淘汰策略lru,自动将内存中的数据清除掉  
AOF是存放每条写命令的，所以会不断的增大，当大到一定程度时，AOF会做rewrite操作,rewrite操作就是基于当时redis的数据重新构造一个小的AOF文件，然后将大的AOF文件删除。

AOF的优点：  
（1）AOF可以更好的保护数据不丢失，一般AOF会以每隔1秒，通过后台的一个线程去执行一次fsync操作，如果redis进程挂掉，最多丢失1秒的数据。  
（2）AOF以appen-only的模式写入，所以没有任何磁盘寻址的开销，写入性能非常高。  
（3）AOF日志文件的命令通过非常可读的方式进行记录，这个非常适合做灾难性的误删除紧急恢复，如果某人不小心用flushall命令清空了所有数据，只要这个时候还没有执行rewrite，那么就可以将日志文件中的flushall删除，进行恢复。  
AOF的缺点  
（1）对于同一份文件AOF文件比RDB数据快照要大。  
（2）AOF开启后支持写的QPS会比RDB支持的写的QPS低，因为AOF一般会配置成每秒fsync操作，每秒的fsync操作还是很高的  
（3）数据恢复比较慢，不适合做冷备。

RDB和AOF到底如何选择  
（1）不要仅仅使用RDB这样会丢失很多数据。  
（2）也不要仅仅使用AOF，因为这一会有两个问题，第一通过AOF做冷备没有RDB做冷备恢复的速度快；第二RDB每次简单粗暴生成数据快照，更加健壮。  
（3）综合AOF和RDB两种持久化方式，用AOF来保证数据不丢失，作为恢复数据的第一选择；用RDB来做不同程度的冷备，在AOF文件都丢失或损坏不可用的时候，可以使用RDB进行快速的数据恢复。

**存储结构:**

  内容是redis通讯协议(RESP)格式的命令文本存储。

**比较**：

1、aof文件比rdb更新频率高，优先使用aof还原数据。

2、aof比rdb更安全也更大

3、rdb性能比aof好

4、如果两个都配了优先加载AOF

**刚刚上面你有提到redis通讯协议(RESP )，能解释下什么是RESP？有什么特点？（可以看到很多面试其实都是连环炮，面试官其实在等着你回答到这个点，如果你答上了对你的评价就又加了一分）**

RESP 是redis客户端和服务端之间使用的一种通讯协议；

RESP 的特点：实现简单、快速解析、可读性好

For Simple Strings the first byte of the reply is "+" 回复

For Errors the first byte of the reply is "-" 错误

For Integers the first byte of the reply is ":" 整数

For Bulk Strings the first byte of the reply is "$" 字符串

For Arrays the first byte of the reply is "\*" 数组

# [Redis协议规范(RESP)](https://www.cnblogs.com/tommy-huang/p/6051577.html)

Redis 即 REmote Dictionary Server (远程字典服务)；

而Redis的协议规范是 Redis Serialization Protocol (Redis序列化协议)

该协议是用于与Redis服务器通信的，用的较多的是Redis-cli通过pipe与Redis服务器联系；

    协议如下：

          客户端以规定格式的形式发送命令给服务器；

          服务器在执行最后一条命令后，返回结果。

## 客户端发送命令的格式(类型)：5种类型

    间隔符号，在Linux下是\r\n，在Windows下是\n

### 1. 简单字符串 Simple Strings, 以 "+"加号 开头

      格式：+ 字符串 \r\n

               字符串不能包含 CR或者 LF(不允许换行)

      eg: "+OK\r\n"

      注意：为了发送二进制安全的字符串，一般推荐使用后面的 Bulk Strings类型

### 2. 错误 Errors, 以"-"减号 开头

　　格式：- 错误前缀 错误信息 \r\n

               错误信息不能包含 CR或者 LF(不允许换行)，Errors与Simple Strings很相似，不同的是Erros会被当作异常来看待

      eg: "-Error unknow command 'foobar'\r\n"

### 3. 整数型 Integer， 以 ":" 冒号开头

　　格式：: 数字 \r\n

      eg: ":1000\r\n"

### 4. 大字符串类型 Bulk Strings, 以 "$"美元符号开头，长度限制512M

　　格式：$ 字符串的长度 \r\n 字符串 \r\n

               字符串不能包含 CR或者 LF(不允许换行);

      eg: "$**6**\r\n**foobar**\r\n"    其中字符串为 foobar，而6就是foobar的字符长度

            "$0\r\n\r\n"       空字符串

            "$-1\r\n"           null

### 5. 数组类型 Arrays，以 "\*"星号开头

　　格式：\* 数组元素个数 \r\n 其他所有类型 (结尾不需要\r\n)

             注意：只有元素个数后面的\r\n是属于该数组的，结尾的\r\n一般是元素的

      eg: "\*0\r\n"      空数组

           "\*2\r\n$2\r\nfoo\r\n$3\r\nbar\r\n"      数组包含2个元素，分别是字符串foo和bar

　　　　"\*3\r\n:1\r\n:2\r\n:3\r\n"       数组包含3个整数：1、2、3

           "\*5\r\n:1\r\n:2\r\n:3\r\n:4\r\n$6\r\nfoobar\r\n"  包含混合类型的数组

           "\*-1\r\n"         Null数组

           "\*2\r\n\*3\r\n:1\r\n:2\r\n:3\r\n\*2\r\n+Foo\r\n-Bar\r\n"   数组嵌套，外层数组包含2个数组，整理后如下：

                 "\*2\r\n

　　　　　　\*3\r\n:1\r\n:2\r\n:3\r\n

　　　　　　\*2\r\n+Foo\r\n-Bar\r\n"

**Redis 有哪些架构模式？讲讲各自的特点**

各种web开发业务中最为常用的**key-value数据库**了

应用：

　　在业务中用其存储用户登陆态（Session存储），加速一些热数据的查询（相比较mysql而言，速度有数量级的提升），做简单的消息队列（LPUSH和BRPOP）、订阅发布（PUB/SUB）系统等等。

规模比较大的互联网公司，一般都会有专门的团队，将Redis存储以**基础服务**的形式提供给各个业务调用。

不过任何一个基础服务的提供方，都会被调用方问起的一个问题是：**你的服务是否具有高可用性？**最好不要因为你的服务经常出问题，导致我这边的业务跟着遭殃。

首先我们要定义一下对于Redis服务来说怎样才算是高可用，即在各种出现异常的情况下，依然可以正常提供服务。或者宽松一些，出现异常的情况下，只经过很短暂的时间即可恢复正常服务。

所谓异常，应该至少包含了以下几种可能性：

【异常1】某个节点服务器的某个进程突然down掉（例如某开发手残，把一台服务器的redis-server进程kill了）

【异常2】某台节点服务器down掉，相当于这个节点上所有进程都停了（例如某运维手残，把一个服务器的电源拔了；例如一些老旧机器出现硬件故障）

【异常3】任意两个节点服务器之间的通信中断了（例如某临时工手残，把用于两个机房通信的光缆挖断了）

其实以上任意一种异常都是小概率事件，而做到高可用性的基本指导思想就是：多个小概率事件同时发生的概率可以忽略不计。只要我们设计的系统可以容忍短时间内的单点故障，即可实现高可用性。

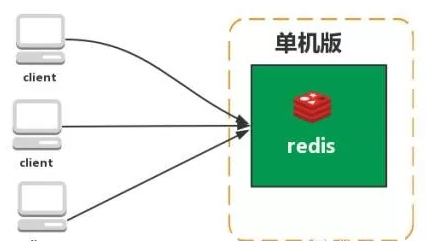
对于搭建**高可用Redis服务**，网上已有了很多方案，例如Keepalived，Codis，Twemproxy，Redis Sentinel。

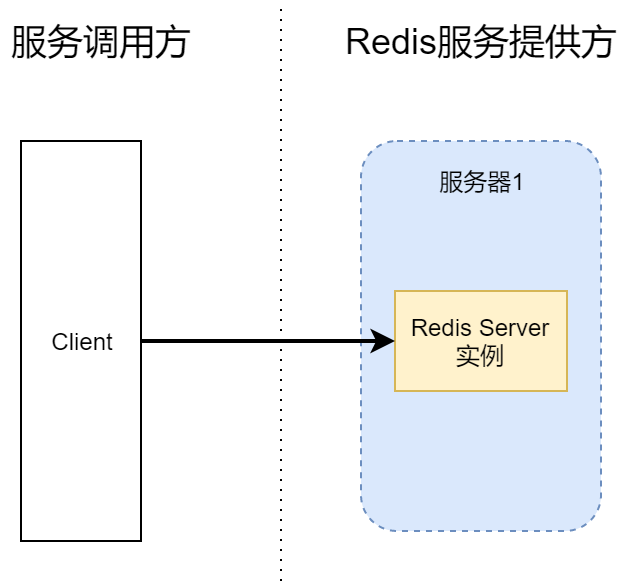
其中Codis和Twemproxy主要是用于**大规模的Redis集群**中，也是在Redis官方发布Redis Sentinel之前twitter和豌豆荚提供的开源解决方案。

我的业务中数据量并不大，所以搞集群服务反而是浪费机器了。最终在Keepalived和Redis Sentinel之间做了个选择，选择了官方的解决方案Redis Sentinel。

Redis Sentinel可以理解为一个**监控Redis Server服务是否正常**的进程，并且一旦检测到不正常，可以自动地将备份（slave）Redis Server启用，使得外部用户对Redis服务内部出现的异常无感知。我们按照由简至繁的步骤，搭建一个**最小型的高可用的Redis服务**。

**单机版**





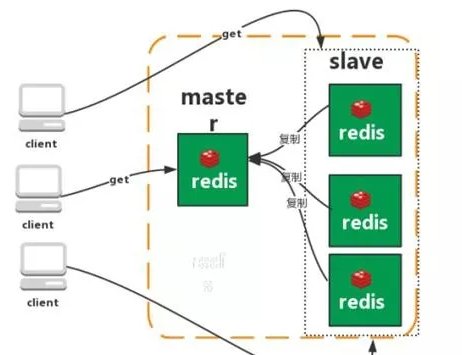
问题：

1、内存容量有限 2、处理能力有限 3、无法高可用。

这种搭配仅适合个人学习娱乐，毕竟这种配置总会有**单点故障**的问题无法解决

如果没有配置**Redis数据持久化**的话，Redis内部已经存储的数据也会丢失。

**主从复制**



Redis 的复制（replication）功能允许用户根据一个 Redis 服务器来创建任意多个该服务器的复制品，其中被复制的服务器为主服务器（master），而通过复制创建出来的服务器复制品则为从服务器（slave）。 只要主从服务器之间的网络连接正常，主从服务器两者会具有相同的数据，主服务器就会一直将发生在自己身上的数据更新同步 给从服务器，从而一直保证主从服务器的数据相同。

特点：

1、master/slave 角色

2、master/slave 数据相同

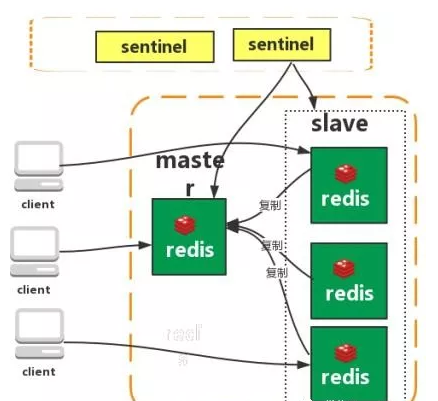
3、降低 master 读压力在转交从库

问题：

无法保证高可用

没有解决 master 写的压力

**哨兵**



Redis sentinel 是一个分布式系统中监控 redis 主从服务器，并在主服务器下线时自动进行故障转移。其中三个特性：

监控（Monitoring）：    Sentinel  会不断地检查你的主服务器和从服务器是否运作正常。

提醒（Notification）： 当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。

自动故障迁移（Automatic failover）： 当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作。

特点：

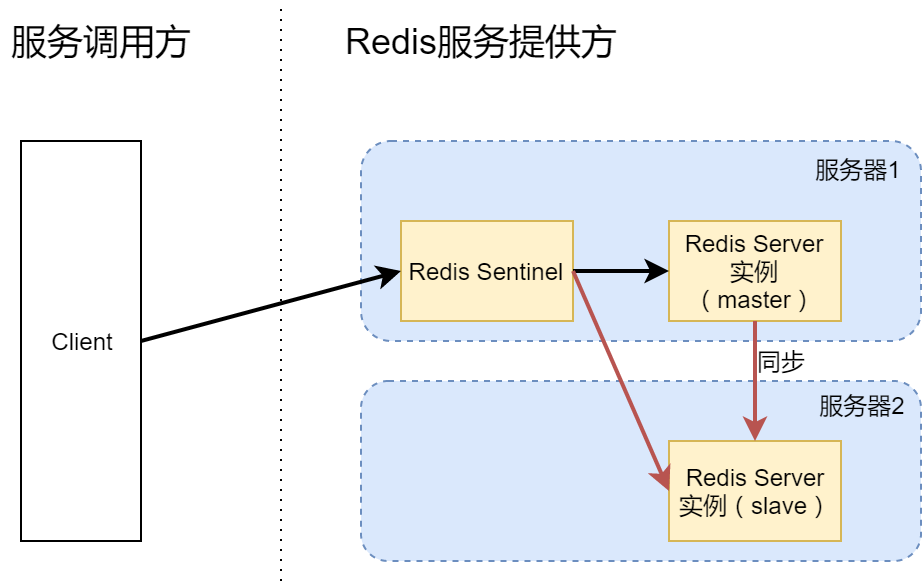
1、保证高可用

2、监控各个节点

3、自动故障迁移

缺点：主从模式，切换需要时间丢数据

没有解决 master 写的压力



解决单点故障问题，我们必须增加一个备份服务，

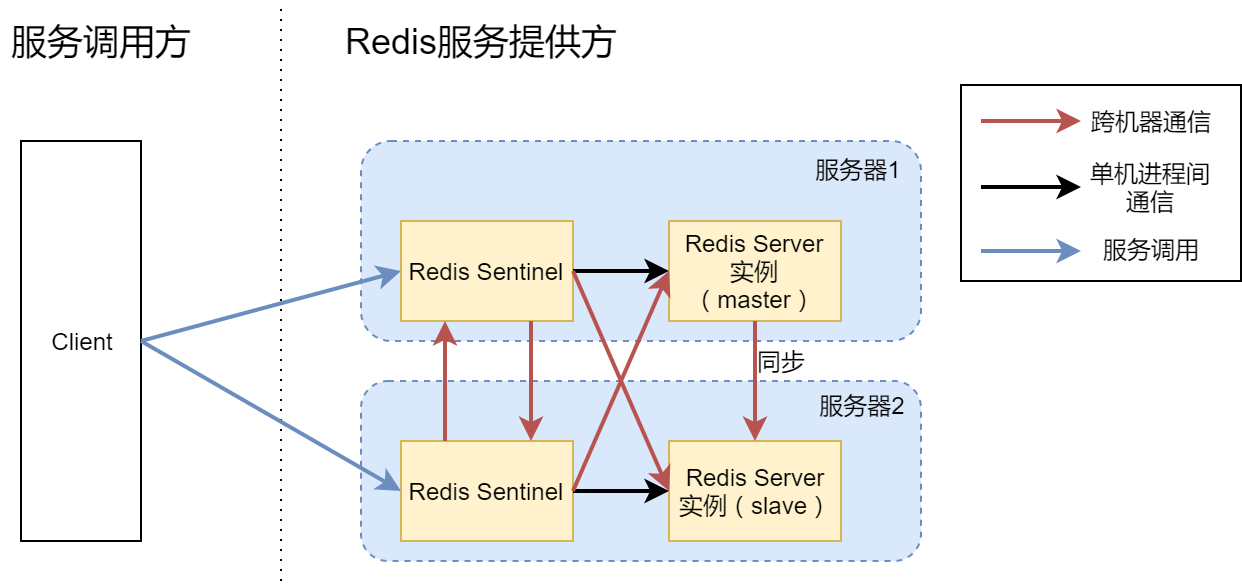
即在两台服务器上分别各启动一个Redis Server进程，一般情况下由master提供服务，slave只负责同步和备份。与此同时，在额外启动一个Sentinel进程，监控两个Redis Server实例的可用性，以便在master挂掉的时候，及时把slave提升到master的角色继续提供服务，这样就实现了Redis Server的高可用。

这基于一个高可用服务设计的依据，即单点故障本身就是个小概率事件，而多个单点同时故障（即master和slave同时挂掉），可以认为是（基本）不可能发生的事件。

对于Redis服务的调用方来说，现在要连接的是Redis Sentinel服务，而不是Redis Server了。常见的调用过程是，client先连接Redis Sentinel并询问目前Redis Server中哪个服务是master，哪些是slave，然后再去连接相应的Redis Server进行操作。当然目前的第三方库一般都已经实现了这一调用过程，不再需要我们手动去实现（例如Nodejs的ioredis，PHP的predis，Golang的go-redis/redis，JAVA的jedis等）。

然而，我们实现了Redis Server服务的主从切换之后，又引入了一个新的问题，即**Redis Sentinel本身也是个单点服务**，一旦Sentinel进程挂了，那么客户端就没办法链接Sentinel了。所以说，方案2的配置并无法实现高可用性。

方案3：主从同步Redis Server，双实例Sentinel



解决Redis Sentinel的单点故障

把Redis Sentinel进程也额外启动一份，两个Sentinel进程同时为客户端提供服务发现的功能。对于客户端来说，它可以连接任何一个Redis Sentinel服务，来获取当前Redis Server实例的基本信息。

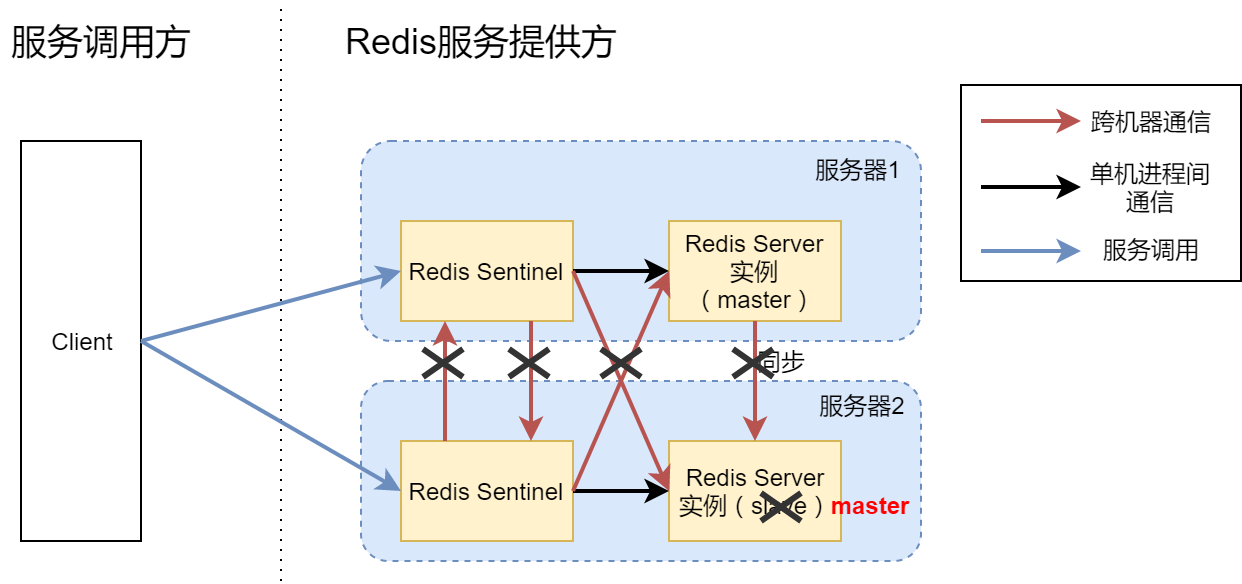
通常情况下，我们会在Client端配置多个Redis Sentinel的链接地址，Client一旦发现某个地址连接不上，会去试图连接其他的Sentinel实例，这当然也不需要我们手动实现，各个开发语言中比较热门的redis连接库都帮我们实现了这个功能。

我们预期是：即使其中一个Redis Sentinel挂掉了，还有另外一个Sentinel可以提供服务。

然而，愿景是美好的，现实却是很残酷的。如此架构下，**依然无法实现Redis服务的高可用**。

方案3示意图中，红线部分是两台服务器之间的通信，而我们所设想的异常场景（【异常2】）是，某台服务器整体down机，不妨假设服务器1停机，此时，只剩下服务器2上面的Redis Sentinel和slave Redis Server进程。这时，Sentinel其实是不会将仅剩的slave切换成master继续服务的，也就导致Redis服务不可用，因为Redis的设定是**只有当超过50%**的Sentinel进程**可以连通并投票选取新的master**时，才会真正发生**主从切换**。本例中两个Sentinel只有一个可以连通，等于50%并不在可以主从切换的场景中。

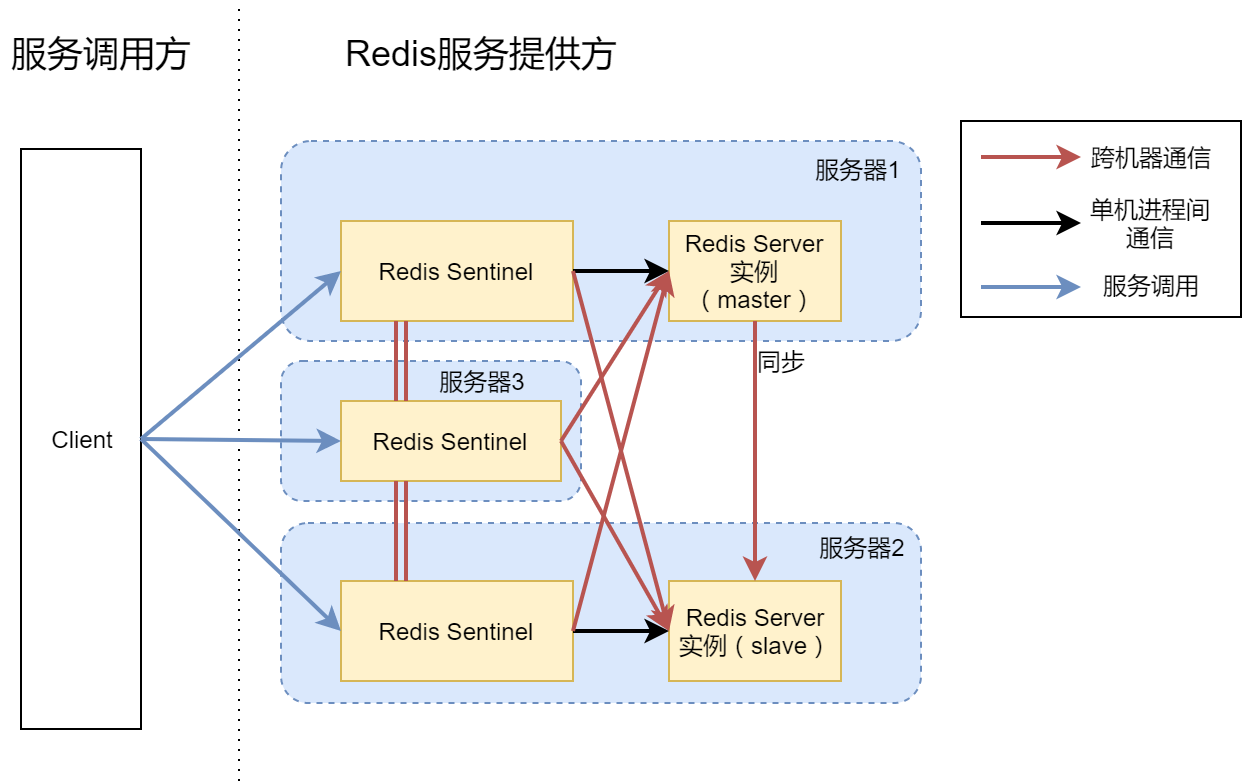
你可能会问，为什么Redis要有这个50%的设定？假设我们允许小于等于50%的Sentinel连通的场景下也可以进行主从切换。试想一下【异常3】，即**服务器1和服务器2之间的网络中断，但是服务器本身是可以运行的**。如下图所示：



实际上对于服务器2来说，服务器1直接down掉和服务器1网络连不通是一样的效果，反正都是突然就无法进行任何通信了。假设网络中断时我们允许服务器2的Sentinel把slave切换为master，结果就是你现在拥有了两个可以对外提供服务的Redis Server。**Client做任何的增删改操作，有可能落在服务器1的Redis上，也有可能落在服务器2的Redis上**（取决于Client到底连通的是哪个Sentinel），造成数据混乱。

即使后面服务器1和服务器2之间的网络又恢复了，那我们也无法把数据统一了（两份不一样的数据，到底该信任谁呢？），数据一致性完全被破坏。

方案4：主从同步Redis Server，三实例Sentinel



最终搭建的架构：

引入了服务器3，并且在3上面又搭建起一个Redis Sentinel进程，现在由三个Sentinel进程来管理两个Redis Server实例。这种场景下，不管是单一进程故障、还是单个机器故障、还是某两个机器网络通信故障，都可以继续对外提供Redis服务。

实际上，如果你的机器比较空闲，当然也可以把服务器3上面也开启一个Redis Server，形成1 master + 2 slave的架构，每个数据都有两个备份，可用性会提升一些。当然也并不是slave越多越好，毕竟主从同步也是需要时间成本的。

在方案4中，一旦服务器1和其他服务器的通信完全中断，那么服务器2和3会将slave切换为master。对于客户端来说，在这么一瞬间会有2个master提供服务，并且一旦网络恢复了，那么所有在**中断期间落在服务器1上的新数据都会丢失**。

如果想要部分解决这个问题，可以**配置Redis Server进程，让其在检测到自己网络有问题的时候，立即停止服务**，避免在网络故障期间还有新数据进来（可以参考Redis的min-slaves-to-write和min-slaves-max-lag这两个配置项）。

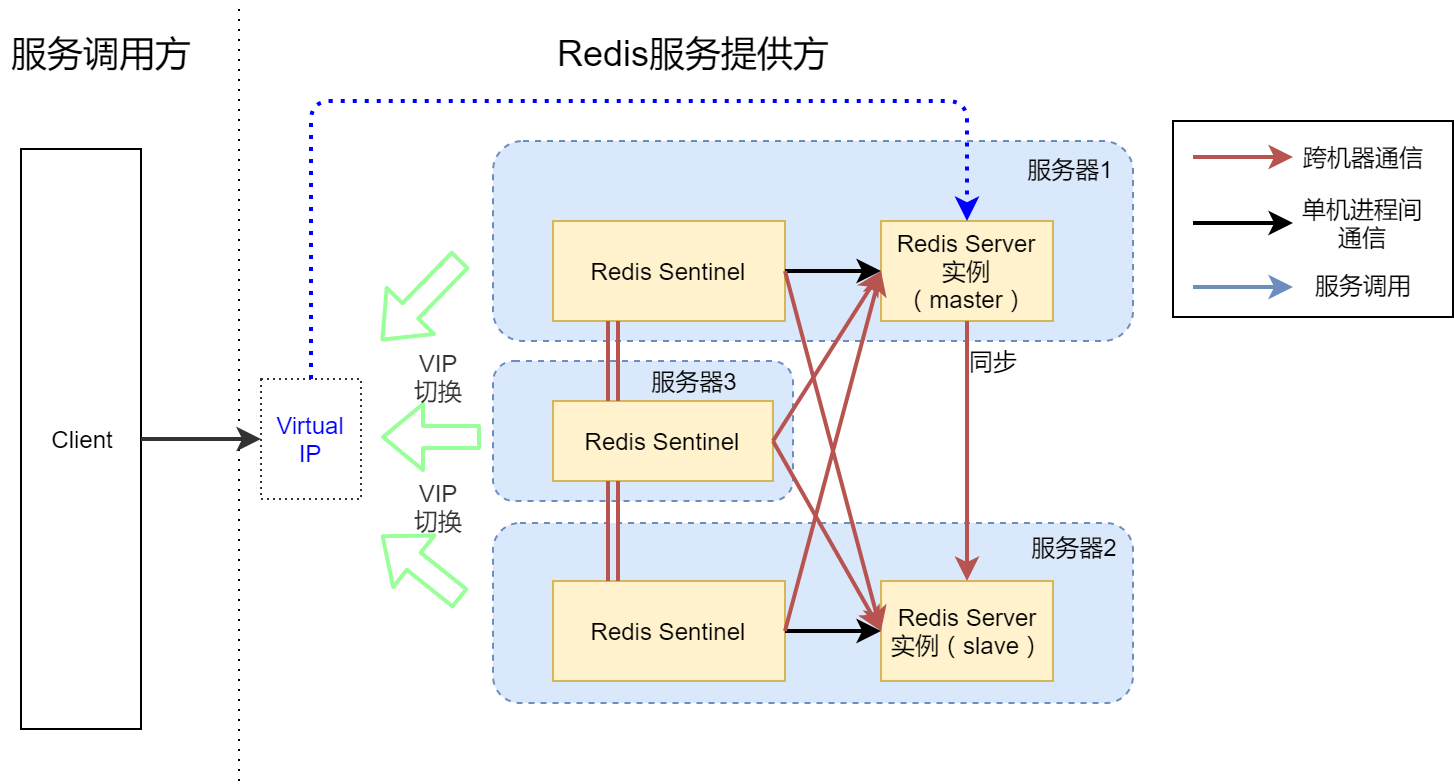
至此，我们就用3台机器搭建了一个高可用的Redis服务。其实网上还有更加节省机器的办法，就是把一个Sentinel进程放在Client机器上，而不是服务提供方的机器上。

只不过在公司里面，一般服务的提供方和调用方并不来自同一个团队。两个团队共同操作同一个机器，很容易因为沟通问题导致一些误操作，所以出于这种人为因素的考虑，我们还是采用了方案4的架构。

并且由于服务器3上面只跑了一个Sentinel进程，对服务器资源消耗并不多，还可以用服务器3来跑一些其他的服务。

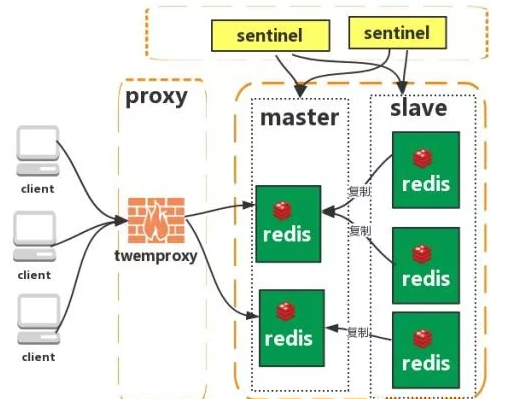
易用性：像使用单机版Redis一样使用Redis Sentinel

作为服务的提供方，我们总是会讲到用户体验问题。在上述方案当中始终有一个让Client端用的不是那么舒服的地方。对于单机版Redis，Client端直接连接Redis Server，我们只需要给一个ip和port，Client就可以使用我们的服务了。而改造成Sentinel模式之后，Client不得不采用一些支持Sentinel模式的外部依赖包，并且还要修改自己的Redis连接配置，这对于“矫情”的用户来讲显然是不能接收的。有没有办法还是像在使用单机版的Redis那样，**只给Client一个固定的ip和port就可以提供服务呢？**



答案当然是肯定的。这可能就要**引入虚拟IP（Virtual IP，VIP）**，如上图所示。我们可以把**虚拟IP指向Redis Server master所在的服务器，在发生Redis主从切换的时候，会触发一个回调脚本，回调脚本中将VIP切换至slave所在的服务器**。这样对于Client端来说，他仿佛在使用的依然是一个单机版的高可用Redis服务。

**集群（proxy 型）：**



Twemproxy 是一个 Twitter 开源的一个 redis 和 memcache 快速/轻量级代理服务器； Twemproxy 是一个快速的单线程代理程序，支持 Memcached ASCII 协议和 redis 协议。

特点：1、多种 hash 算法：MD5、CRC16、CRC32、CRC32a、hsieh、murmur、Jenkins

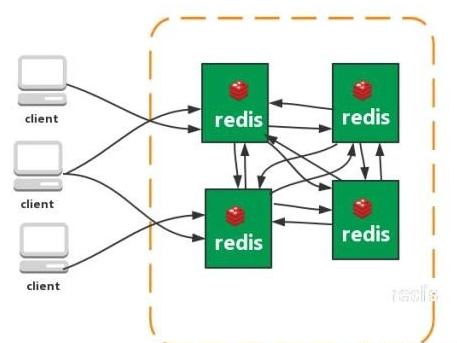
2、支持失败节点自动删除

3、后端 Sharding 分片逻辑对业务透明，业务方的读写方式和操作单个 Redis 一致

缺点：增加了新的 proxy，需要维护其高可用。

failover 逻辑需要自己实现，其本身不能支持故障的自动转移可扩展性差，进行扩缩容都需要手动干预

**集群（直连型）：**



从redis 3.0之后版本支持redis-cluster集群，Redis-Cluster采用无中心结构，每个节点保存数据和整个集群状态,每个节点都和其他所有节点连接。

特点：

1、无中心架构（不存在哪个节点影响性能瓶颈），少了 proxy 层。

2、数据按照 slot 存储分布在多个节点，节点间数据共享，可动态调整数据分布。

3、可扩展性，可线性扩展到 1000 个节点，节点可动态添加或删除。

4、高可用性，部分节点不可用时，集群仍可用。通过增加 Slave 做备份数据副本

5、实现故障自动 failover，节点之间通过 gossip 协议交换状态信息，用投票机制完成 Slave到 Master 的角色提升。

缺点：

1、资源隔离性较差，容易出现相互影响的情况。

2、数据通过异步复制,不保证数据的强一致性

**什么是一致性哈希算法？什么是哈希槽？**

这两个问题篇幅过长 网上找了两个解锁的不错的文章