MySQL

什么是数据库？

首先数据库不仅仅是一堆数据的集合，实际上比这个复杂的多。

有两个重要组成部分：数据库和实例。

1. 数据库：物理操作文件系统或者其他文件形式的集合
2. 实例：后台进程和共享内存区组成的运行态

在 MySQL 中，实例和数据库往往都是一一对应的，而我们也无法直接操作数据库，而是要通过数据库实例来操作数据库文件，可以理解为数据库实例是数据库为上层提供的一个专门用于操作的接口。在 Unix 上，启动一个 MySQL 实例往往会产生两个进程，`mysqld` 就是真正的数据库服务守护进程，而 `mysqld_safe` 是一个用于检查和设置 `mysqld` 启动的控制程序，它负责监控 MySQL 进程的执行，当 `mysqld` 发生错误时，`mysqld_safe` 会对其状态进行检查并在合适的条件下重启。

什么是SQL？什么是MySQL？

sql是一种结构化查询语言，用于在数据库中存储，查询和删除数据用的。

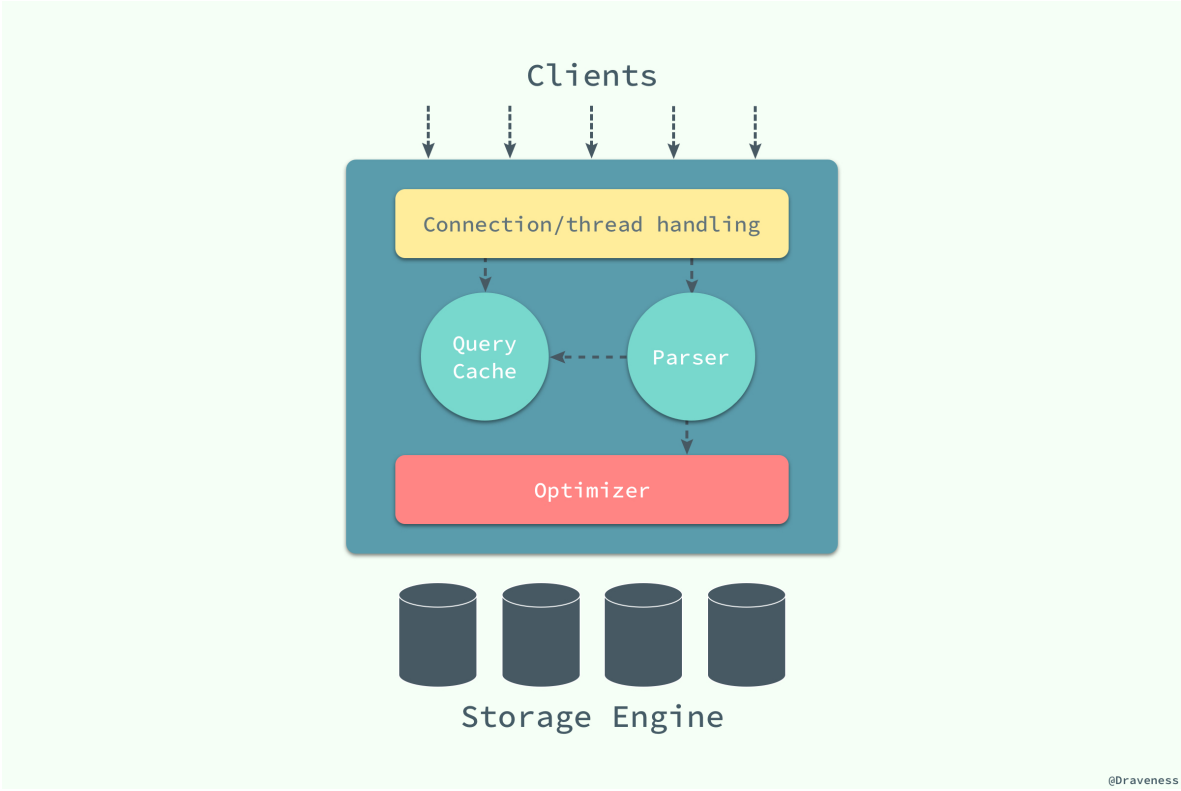
mysql是一个数据库管理系统，开源免费。

参考链接

MySQL都有哪些存储引擎？说一说InnoDB

功 能	MYISAM	Memory	InnoDB	Archive
存储限制	256TB	RAM	64TB	None
支持事物	No	No	Yes	No
支持全文索引	Yes	No	No	No
支持数索引	Yes	Yes	Yes	No
支持哈希索引	No	Yes	No	No
支持数据缓存	No	N/A	Yes	No
支持外键	No	No	Yes	No

MySQL存储架构



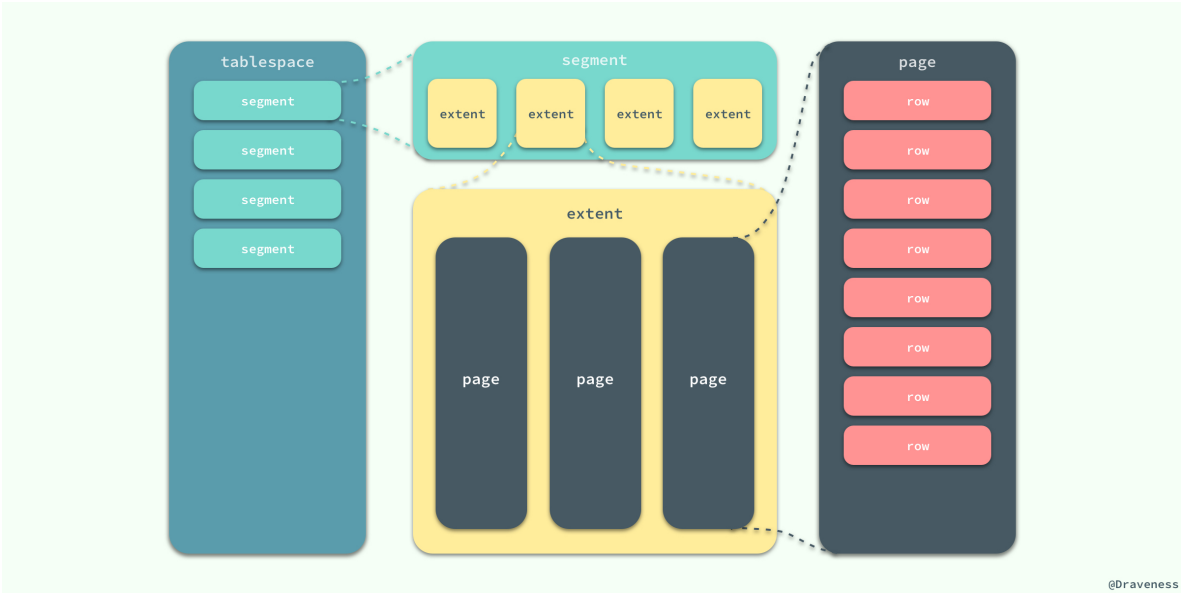
第一层用于连接、线程处理的部分并不是 MySQL 『发明』的，很多服务都有类似的组成部分；

第二层中包含了大多数 MySQL 的核心服务，包括了对 SQL 的解析、分析、优化和缓存等功能，存储过程、触发器和视图都是在这里实现的；

第三层就是 MySQL 中真正负责数据的存储和提取的存储引擎，例如：InnoDB、MyISAM 等，文中对存储引擎的介绍都是对 InnoDB 实现的分析。

innoDB引擎存储结构

在 InnoDB 存储引擎中，所有的数据都被**逻辑地**存放在表空间中，表空间（tablespace）是存储引擎中最高的存储逻辑单位，在表空间的下面又包括段（segment）、区（extent）、页（page）：



同一个数据库实例的所有表空间都有相同的页大小；默认情况下，表空间中的页大小都为 16KB，当然也可以通过改变 `innodb_page_size` 选项对默认大小进行修改，需要注意的是不同的页大小最终也会导致区大小的不同：

从图中可以看出，在 InnoDB 存储引擎中，一个区的大小最小为 1MB，页的数量最少为 64 个。

Relation Between Page Size & Extent Size

Page Size	Page Count	Extent Size
4KB	256	1MB
8KB	128	1MB
16KB	64	1MB
32KB	64	2MB
64KB	64	4MB

如何存储？

MySQL 使用 InnoDB 存储表时，会将**表的定义**和**数据，索引**等信息分开存储，其中前者存储在 `.frm` 文件中，后者存储在 `.ibd` 文件中，这一节就会对这两种不同的文件分别进行介绍。

.frm 文件

无论在 MySQL 中选择了哪个存储引擎，所有的 MySQL 表都会在硬盘上创建一个 `.frm` 文件用来描述表的格式或者说定义；`.frm` 文件的格式在不同的平台上都是相同的。

.ibd 文件

储了当前表的数据和相关的索引数据。

如何存储?

与现有的大多数存储引擎一样，InnoDB 使用页作为磁盘管理的最小单位；数据在 InnoDB 存储引擎中都是按行存储的，每个 16KB 大小的页中可以存放 2-200 行的记录。

数据库的三大范式？

一范式就是属性不可分割。第一范式是关系型数据库的基本要求，表示每一列都是不可分割的基本数据项。比如数据库中的“地址”属性，如果要经常访问地址中的"所在城市"，就要把“地址”属性分割成“省份”、“城市”、“街道”等基本项

二范式就是要有主键,其他字段都依赖于主键。就是说一张表的每一列都和主键相关，而不能只与主键的某一部分相关，不能把多种数据保存到同一张表中。

三范式就是要消除传递依赖,消除冗余,就是各种信息只在一个地方存储,不出现在多张表中（很多时候会牺牲第三范式）。比如下图这样：

第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。比如在设计一个订单数据表的时候，可以将客户编号作为一个外键和订单表建立相应的关系。而不可以在订单表中添加关于客户其它信息（比如姓名、所属公司等）的字段。

什么是存储过程？有哪些优缺点？

定义： 就是数据库 SQL 语言层面的代码封装与重用。存储过程是数据库系统中为了完成特定功能的SQL语句集合，经编译后保存在数据库中。普通的SQL语句我们都是保存到其他地方，每次需要编译后

订单信息表

订单编号	订单项目	负责人	业务员	订单数量	客户编号
001	挖掘机	刘明	李东明	1台	1
002	冲击钻	李刚	霍新峰	8个	2
003	铲车	郭新一	艾美丽	2辆	1

客户信息表

客户编号	客户名称	所属公司	联系方式
1	李聪	五一建设	13253661015
2	刘新明	个体经营	13285746958

才能执行，效率比较的低。而有了存储过程后，第一次编译后再次调用不需要再次编译，用户通过存储过程的名字来调用。

- 优点
    1. 效率高。编译一次后，就会存到数据库，每次调用时都直接执行。而普通的sql语句我们要保存到其他地方（记事本 ），都要先分析编译才会执行。
    2. 维护方便。当发生改动时候，修改之前的存储过程比较容易
    3. 复用性高。存储过程往往针对特定功能编写的，因此可以重复调用
    4. 安全性高。使用的时候有身份限制，只能特定用户使用
    5. 降低网络流量。存储过程编译好会放在数据库，我们在远程调用时，不会传输大量的字符串类型的sql语句。
- 缺点
    1. 不同厂商数据库系统之间不兼容。

mysql索引是什么？有哪几种类型？优缺点？

索引的定义

MySQL官方对索引的定义为：索引（Index）是帮助MySQL高效获取数据的数据结构。是加快检索表中数据的方法。对于一张表来说，如果不加索引的话就要从表的第一行开始查找，如果一个表有百万行的话效率会非常低。如果有了索引，利用数据结构就可以快速查找。

优缺点

- 索引的优点
    1. 加快数据的检索速度
- 索引的缺点
    1. 创建和维护高效的索引表比较麻烦
    2. 占用物理空间

索引的类型

- 索引的类型
  - 字段类型分类：
    1. 普通索引。没有任何约束，**允许空值和重复值，纯粹为了提高查询效率而存在。**
    2. 唯一索引。在普通索引上加上数据不允许重复，允许为null
    3. 主键索引。在唯一索引上加上不允许为null，一个表只能有一个主键
    4. 全文索引。没见过。
  - 按数据结构分类可分为：
    1. **B+tree索引**
    2. **Hash索引**
    3. **Full-text索引**
  - 其他
    1. 聚簇索引
    2. 非聚簇索引

算法原理

数据库事务

> 事务是并发控制的基本单位。事务他是一个操作序列，这些操作要么都执行，要么都不执行，是一个不可分割的单位。最简单的例子就是银行转账了，从一个账户汇钱到另一个账户，两个操作要么都执行要么都不执行。

- 数据库中的事务有以下四个特征：
  1. 原子性。事务中的操作被看成一个逻辑单元，这个逻辑单元的操作要么全做，要么全部做。
  2. 一致性。当对数据进行更改后，如果回滚会回到最初状态。
  3. 隔离性。允许多个用户对同一个数据进行并发访问同时不破坏数据的完整性和正确性。同时并行事务的修改必须和其他并行事务独立。
  4. 持久性。事务结束后，结果必须能持久保存。
- 事务的语句
  1. 开始事务。`BEGIN TRANSACTION`
  2. 提交事务。`COMMIT TRANSACTION`
  3. 回滚事务。`ROLLBACK TRANSACTION`

数据库锁机制

并发控制机制

并发控制的任务就是保证多个事务存取数据统一数据时候不破坏事务的隔离性和统一性。乐观锁和悲观锁其实都是并发控制的机制，同时它们在原理上就有着本质的差别：

- 悲观锁

  定义：悲观锁顾名思义，认为当前操作的数据会被外界其他事务所修改，因此在整个数据处理过程中，将数据锁定，屏蔽一切可能违反事务性质的操作。因此每次获取数据的时候都会进行加锁操作，防止外界修改。由于该数据加锁，因此对改数据进行读写操作的其他进程会进入等待状态。**悲观锁的实现需要数据库的锁机制来完成**，只有数据库系统的锁机制才能保证访问的排他性。

评价：效率上，加锁会让数据库产生额外的开销，同时还有增加死锁的机会。另外，如果是只读型事务的话，加锁是没必要的，因此频繁写入的业务可能需要。而且一旦某数据被加锁了，其它数据必须等待才行。

- 乐观锁

  定义：乐观锁假设认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做。

  实现方式：**使用版本号实现乐观锁，版本号的实现方式有两种，一个是数据版本机制，一个是时间戳机制。具体如下。**

  1. 为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。
  2. 时间戳机制，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp）, 和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。

乐观锁和悲观锁在本质上并不是同一种东西，一个是一种思想，另一个是一种真正的锁，但是它们都是一种并发控制机制。

乐观锁不会存在死锁的问题，但是由于更新后验证，所以当**冲突频率**和**重试成本**较高时更推荐使用悲观锁，而需要非常高的**响应速度**并且**并发量**非常大的时候使用乐观锁就能较好的解决问题，在这时使用悲观锁就可能出现严重的性能问题；在选择并发控制机制时，需要综合考虑上面的四个方面（冲突频率、重试成本、响应速度和并发量）进行选择。

锁的种类

对数据的操作其实只有两种，也就是读和写，而数据库在实现锁时，也会对这两种操作使用不同的锁；InnoDB 实现了标准的行级锁，也就是共享锁（Shared Lock）和互斥锁（Exclusive Lock）

- **共享锁（读锁）**：允许事务对一条行数据进行读取；
- **互斥锁（写锁）**：允许事务对一条行数据进行删除或更新；

而它们的名字也暗示着各自的另外一个特性，共享锁之间是兼容的，而互斥锁与其他任意锁都不兼容：稍微对它们的使用进行思考就能想明白它们为什么要这么设计，因为共享锁代表了读操作、互斥锁代表了写操作，所以我们可以在数据库中**并行读**，但是只能**串行写**，只有这样才能保证不会发生线程竞争，实现线程安全。

锁的粒度

无论是共享锁还是互斥锁其实都只是对某一个数据行进行加锁

InnoDB 支持多种粒度的锁，也就是**行锁**和**表锁**；为了支持多粒度锁定，InnoDB 存储引擎引入了意向锁（Intention Lock），意向锁就是一种表级锁。

- 行锁

  行级锁是Mysql中锁定粒度最细的一种锁，表示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，提高并发度。其加锁粒度最小，但加锁的开销也最大，还会出现死锁。行级锁分为共享锁和排他锁。

- 表锁

  级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分MySQL引擎支持。表级锁分为共享锁和排他锁。开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，并发度最低。

- 意向锁

  为了支持多粒度锁定，InnoDB 存储引擎引入了意向锁（Intention Lock）

  1. **意向共享锁**：事务想要在获得表中某些记录的共享锁，需要在表上先加意向共享锁；
  2. **意向互斥锁**：事务想要在获得表中某些记录的互斥锁，需要在表上先加意向互斥锁；

  意向锁其实不会阻塞全表扫描之外的任何请求，它们的主要目的是为了表示**是否有人请求锁定表中的某一行数据**。

  > 有的人可能会对意向锁的目的并不是完全的理解，我们在这里可以举一个例子：如果没有意向锁，当已经有人使用行锁对表中的某一行进行修改时，如果另外一个请求要对全表进行修改，那么就需要对所有的行是否被锁定进行扫描，在这种情况下，效率是非常低的；不过，在引入意向锁之后，当有人使用行锁对表中的某一行进行修改之前，会先为表添加意向互斥锁（IX），再为行记录添加互斥锁（X），在这时如果有人尝试对全表进行修改就不需要判断表中的每一行数据是否被加锁了，只需要通过等待意向互斥锁被释放就可以了。

锁的算法

介绍三种锁的算法：Record Lock、Gap Lock 和 Next-Key Lock。

- Record Lock记录锁

  通过索引建立的 B+ 树找到行记录并添加锁。但是如果InnoDB 不知道待修改的记录具体存放的位置，也无法对将要修改哪条记录提前做出判断就会锁定整个表。

- Gap Lock间隙锁

  记录锁是在存储引擎中最为常见的锁，除了记录锁之外，InnoDB 中还存在间隙锁（Gap Lock），间隙锁是对索引记录中的一段连续区域的锁；

  当使用类似 `SELECT * FROM users WHERE id BETWEEN 10 AND 20 FOR UPDATE;` 的 SQL 语句时，就会阻止其他事务向表中插入 `id = 15` 的记录，因为整个范围都被间隙锁锁定了。

- Next-Key Lock

  Gap Lock+Record Lock，锁定一个范围，并且锁定记录本身

  当我们更新一条记录，比如 `SELECT * FROM users WHERE age = 30 FOR UPDATE;`，InnoDB 不仅会在范围 `(21, 30]` 上加 Next-Key 锁，还会在这条记录后面的范围 `(30, 40]` 加间隙锁，所以插入 `(21, 40]` 范围内的记录都会被锁定。

死锁的产生

既然 InnoDB 中实现的锁是悲观的，那么不同事务之间就可能会互相等待对方释放锁造成死锁，最终导致事务发生错误；想要在 MySQL 中制造死锁的问题其实非常容易：两个会话都持有一个锁，并且尝试获取对方的锁时就会发生死锁，不过 MySQL 也能在发生死锁时及时发现问题，并保证其中的一个事务能够正常工作，这对我们来说也是一个好消息。

drop、truncate和delete的区别

- delete删除的过程是每次从表中删除一行，并且将该操作记录到日志中，可以回滚。
- truncate指一次性从表中删除所有的数据，不保存在日志中因此是不可恢复的。
- drop将表占用的空间删除掉

SQL的组成主要有四部分

- 数据定义。

  DDL(Data Definition Language）数据库定义语言。

  用于定义数据库的三级结构，包括外模式、概念模式、内模式及其相互之间的映像，定义数据的完整性、安全控制等约束。

  DDL不需要commit。

  `CREATE`
  `ALTER`
  `DROP`
  `TRUNCATE`
  `COMMENT`
  `RENAME`

- 数据操纵。

  DML（Data Manipulation Language）数据操纵语言

  用于让用户或程序员使用，实现对数据库中数据的操作。

  需要commit.。

  `SELECT`
  `INSERT`
  `UPDATE`
  `DELETE`
  `MERGE`
  `CALL`
  `EXPLAIN PLAN`
  `LOCK TABLE`

- 数据控制

  DCL（Data Control Language）数据库控制语言

  TCL（Transaction Control Language）事务控制语言

  GRANT 授权
  REVOKE 取消授权

  SAVEPOINT 设置保存点
  ROLLBACK 回滚
  SET TRANSACTION

- 嵌入式中的SQL

什么是视图？视图的使用场景有哪些？

定义：视图是一种虚拟的表，其内容由查询语句定义。同真实的表一样，视图包含一系列带有名称的列和行数据。可以对视图进行增，改，查，操作，视图通常是有一个表或者多个表的行或列的子集。视图本身并不包含任何数据，不在数据库中以存储的数据值集形式存在，它只包含映射到基表的一个查询语句，当基表数据发生变化，视图数据也随之变化。一种抽象的概念如下：

为什么用视图？关系型数据库中的数据是由一张一张的二维关系表所组成，简单的单表查询只需要遍历一个表，而复杂的多表查询需要将多个表连接起来进行查询任务。对于复杂的查询事件，每次查询都需要编写MySQL代码效率低下。为了解决这个问题，数据库提供了视图（view）功能。

查询的数据来源于不同的表，而查询者希望以统一的方式查询，这样也可以建立一个视图，把多个表查询结果联合起来，查询者只需要直接从视图中获取数据，不必考虑数据来源于不同表所带来的差异

常用场景：视图适合于多表连接浏览时使用。不适合增、删、改。

MYSQL索引和算法原理

MySQL索引背后的数据结构及算法原理

MySQL数据库支持多种索引类型，如BTree索引，哈希索引，全文索引等等。主要说一说Btree索引

对于索引的定义：数据库查询是数据库的最主要功能之一。我们都希望查询数据的速度能尽可能的快，因此数据库系统的设计者会从查询算法的角度进行优化。最基本的查询算法当然是顺序查找（linear search），这种复杂度为O(n)的算法在数据量很大时显然是糟糕的，好在计算机科学的发展提供了很多更优秀的查找算法，例如二分查找（binary search）、二叉树查找（binary tree search）等。如果稍微分析一下会发现，每种查找算法都只能应用于特定的数据结构之上，例如二分查找要求被检索数据有序，而二叉树查找只能应用于二叉查找树上，但是数据本身的组织结构不可能完全满足各种数据结构（例如，理论上不可能同时将两列都按顺序进行组织），所以，在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

目前大部分数据库系统及文件系统都采用B-Tree或其变种B+Tree作为索引结构

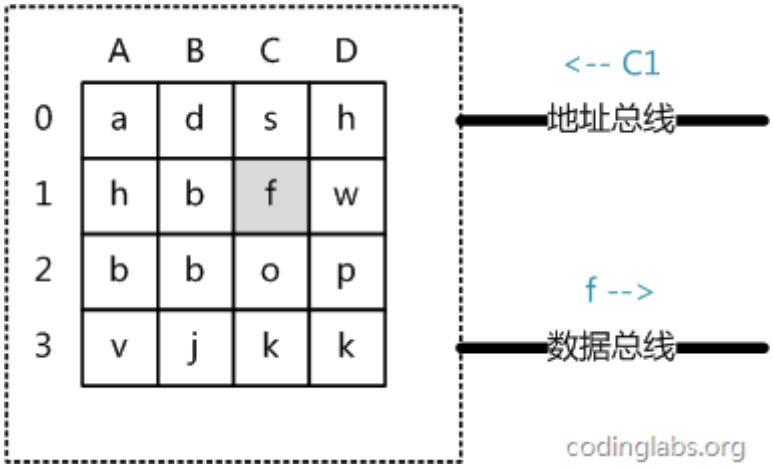
Btree索引

关于BTree和B+Tree的介绍再数据结构篇中，可以先去看一下，弄清楚B树和B+树的区别。

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储在磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级（内存是纳秒，磁盘是毫秒），所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。下面先介绍内存和磁盘存取原理，然后再结合这些原理分析B-/+Tree作为索引的效率。

主存存取

目前计算机使用的主存基本都是随机读写存储器（RAM），现代RAM的结构和存取原理比较复杂，所以这里就抽象出一个十分简单的存取模型来说明RAM的工作原理。



从抽象角度看，主存是一系列的存储单元组成的矩阵，每个存储单元存储固定大小的数据。每个存储单元有唯一的地址，现代主存的编址规则比较复杂，这里将其简化成一个二维地址：通过一个行地址和一个列地址可以唯一定位到一个存储单元。上

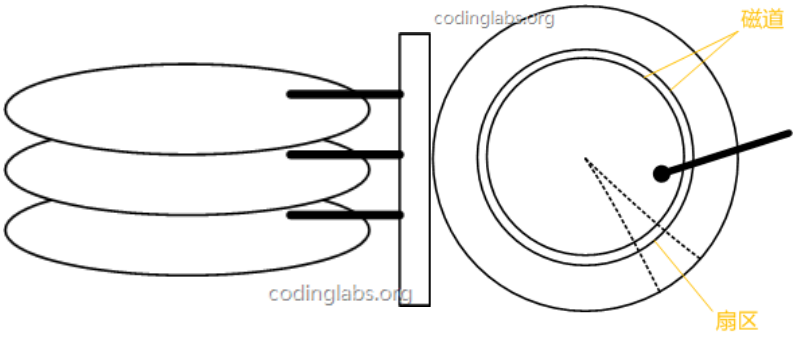
主存的存取过程如下：

1. 当系统需要读取主存时，则将地址信号放到地址总线上传给主存，主存读到地址信号后，解析信号并定位到指定存储单元，然后将此存储单元数据放到数据总线上，供其它部件读取。
2. 写主存的过程类似，系统将要写入单元地址和数据分别放在地址总线和数据总线上，主存读取两个总线的内容，做相应的写操作。

所以可以得出结论：这里可以看出，主存存取的时间仅与存取次数呈线性关系，因为不存在机械操作，两次存取的数据的“距离”不会对时间有任何影响，例如，先取A0再取A1和先取A0再取D3的时间消耗是一样的。

磁盘存取原理

索引一般以文件形式存储在磁盘上，索引检索需要磁盘I/O操作。与主存不同，磁盘I/O存在机械运动耗费，因此磁盘I/O的时间消耗是巨大的。



一个个磁盘，磁盘由磁道和扇区组成

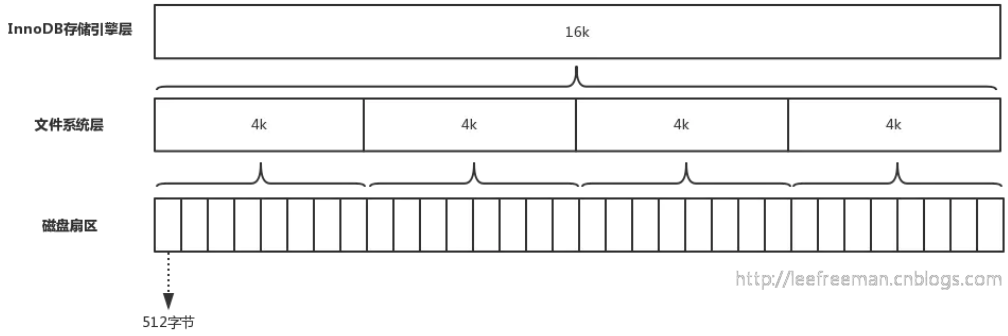
当需要从磁盘读取数据时，系统会将数据逻辑地址传给磁盘，磁盘的控制电路按照寻址逻辑将逻辑地址翻译成物理地址，即确定要读的数据在哪个磁道，哪个扇区。为了读取这个扇区的数据，需要将磁头放到这个扇区上方，为了实现这一点，磁头需要移动对准相应磁道，这个过程叫做寻道，所耗费时间叫做寻道时间，然后磁盘旋转将目标扇区旋转到磁头下，这个过程耗费的时间叫做旋转时间。所以机械时间加上存储介质的特性，磁盘IO肯定很慢。

解决办法：根据计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。**预读的长度一般为页（page）的整倍数**。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

性能分析

> 数据存储最小单元
>
> 在计算机中磁盘存储数据最小单元是扇区，一个扇区的大小是512字节
>
> 虚拟内存中小单元是页，一个页的大小是4k
>
> InnoDB存储引擎也有自己的最小储存单元——页（Page），一个页的大小是默认16K。



- 首先说Btree

  假设Btree一次检索要访问n个节点，数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

  B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为$O(h)=O(log_dN)$。一般实际应用中，阶d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。综上所述，用B-Tree作为索引结构效率是非常高的。

- 再说B+tree

  为什么B+tree比Btree更适合索引呢？因为d越大索引的性能越好，而阶的上限取决于节点内key和data的大小。因为一页大小有限，又存数据又存索引，导致阶会变小。所以B+数就是想方设法将数据去掉，使得节点里面全是索引(key)就行，因此d越大，$O(log_dN)$就会越小。

- 为什么红黑树不行？

  红黑树本质上是一种自平衡的二叉查找树，数据量大的话树的高度太高了。同时由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

- 一个B+索引树可以存多少行数据？

  答：首先看是几层树。对于 B+树而言，树的高度一般不超过 4 层。

  对于 MySQL 的 InnoDB 存储引擎而言，一个结点默认的存储空间为 16Kb。MySQL 的 InnoDB 存储引擎的索引一般用 bigint 存储，占用 8 个 byte，一个索引又会关联一个指向孩子结点的指针，这个指针占用 6 个 byte，也就是说结点中的一个关键字大概要用 14 byte 的空间，而一个结点的默认大小为 16kb ，那么一个结点可以存储关键的个数最多为$16kb/14byte = 1170$，即一个节点可以存储1170个指针，所以阶m=1170。

  一行数据是大小是1k，一个页的大小是16k，因此一页可以放16条数据。一个指针指向一个存放记录的页，一个页可以存放16条数据。这样我们根据高度就可以大致算出一颗B+树能存放多少数据了。**B+树索引本身并不能直接找到具体的一条记录，只能知道该记录在哪个页上，数据库会把页载入到内存，再通过二分查找定位到具体的记录。**

  所以一颗高度为2的B+树可以存放的数据是：`1170*16=18720`条数据。一颗高度为3的B+树可以存放的数据是：`1170*1170*16=21902400`条记录（两千万条）

  理论上就是这样，在InnoDB存储引擎中，B+树的高度一般为2-4层，就可以满足千万级数据的存储。查找数据的时候，一次页的查找代表一次IO，那我们通过主键索引查询的时候，其实最多只需要2-4次IO就可以了。

哈希索引

哈希索引就是采用一定的哈希算法，把键值换算成新的哈希值，检索时不需要类似B+树那样从根节点到叶子节点逐级查找，只需一次哈希算法即可立刻定位到相应的位置，速度非常快。

哈希缺点：

1. Hash 索引仅仅能满足等值查询，不能使用范围查询。
2. Hash 索引无法被用来避免数据的排序操作，由于 Hash 索引中存放的是经过 Hash 计算之后的 Hash 值，而且Hash值的大小关系并不一定和 Hash 运算前的键值完全一样，所以数据库无法利用索引的数据来避免任何排序运算；
3. 当碰撞太高的话，性能不一定很好，比如拉链法，后面跟了一长串。

聚簇索引和非聚簇索引

通俗解释：

聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据

非聚簇索引：将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，myisam通过key_buffer把索引先缓存到内存中，当需要访问数据时（通过索引访问数据），在内存中直接搜索索引，然后通过索引找到磁盘相应数据，这也就是为什么索引不在key buffer命中时，速度慢的原因

聚簇索引：聚簇索引就是按照每张表的主键构造一颗B+树，同时叶子节点中存放的就是整张表的行记录数据，也将聚集索引的叶子节点称为数据页。这个特性决定了索引组织表中数据也是索引的一部分，每张表只能拥有一个聚簇索引。

优点：

1. 数据访问更快，因为聚簇索引将索引和数据保存在同一个B+树中，因此从聚簇索引中获取数据比非聚簇索引更快
2. 聚簇索引对于主键的排序查找和范围查找速度非常快

缺点：

1. 更新主键的代价很高，维护索引很昂贵，因为将会导致被更新的行移动，导致数据被分到不同的页上。因此，对于InnoDB表，我们一般定义主键为不可更新。

使用聚簇索引的场景：

1. 适合用在排序的场合
2. 取出一定范围数据的时候

非聚簇索引：辅助索引叶子节点存储的不再是行的物理位置，而是主键值。通过辅助索引首先找到的是主键值，再通过主键值找到数据行的数据页，再通过数据页中的Page Directory找到数据行。

> 辅助索引使用主键作为"指针"而不是使用地址值作为指针的好处是，减少了当出现行移动或者数据页分裂时辅助索引的维护工作，使用主键值当作指针会让辅助索引占用更多的空间，换来的好处是InnoDB在移动行时无须更新辅助索引中的这个"指针"。也就是说行的位置（实现中通过16K的Page来定位）会随着数据库里数据的修改而发生变化（前面的B+树节点分裂以及Page的分裂），使用聚簇索引就可以保证不管这个主键B+树的节点如何变化，辅助索引树都不受影响。

二级索引需要两次索引查找，而不是一次才能取到数据，因为存储引擎第一次需要通过二级索引找到索引的叶子节点，从而找到数据的主键，然后在聚簇索引中用主键再次查找索引，再找到数据

mysql隔离级别

参考链接

本文所说的 MySQL 事务都是指在 InnoDB 引擎下

数据库事务指的是一组数据操作，事务内的操作要么就是全部成功，要么就是全部失败，什么都不做，其实不是没做，是可能做了一部分但是只要有一步失败，就要回滚所有操作，有点一不做二不休的意思。

事务具有原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）四个特性，简称 ACID，缺一不可。今天要说的就是**隔离性**。

- 概念说明，先搞清都是什么意思
    - 脏读：指的是读到了其他事务未提交的数据，未提交意味着这些数据可能会回滚，也就是可能最终不会存到数据库中，也就是不存在的数据。
    - 可重复读：事务A在读到一条数据之后，此时事务B对该数据进行了修改并提交，那么事务A再读该数据，读到的还是原来的内容。
    - 不可重复读：对比可重复读，不可重复读指的是在同一事务内，不同的时刻读到的同一批数据可能是不一样的，可能会受到其他事务的影响。
    - 幻读：针对数据插入操作来说的。假设事务A对某些行的内容作了更改，但是还未提交，此时事务B插入了与事务A更改前的记录相同的记录行，并且在事务A提交之前先提交了，而这时，在事务A中查询，会发现好像刚刚的更改对于某些数据未起作用，但其实是事务B刚插入进来的，让用户感觉很魔幻，感觉出现了幻觉
- 事务的隔离级别

> MySQL的事务隔离级别一共有四个，分别是读未提交、读已提交、可重复读以及可串行化。MySQL的隔离级别的作用就是让事务之间互相隔离，互不影响，这样可以保证事务的一致性。在Oracle，SqlServer中都是选择读已提交(Read Commited)作为默认的隔离级别，为什么Mysql不选择读已提交(Read Commited)作为默认隔离级别，而选择可重复读(Repeatable Read)作为默认的隔离级别
>
> 隔离级别比较：可串行化>可重复读>读已提交>读未提交
>
> 隔离级别对性能的影响比较：可串行化>可重复读>读已提交>读未提交
>
> 由此看出，隔离级别越高，所需要消耗的MySQL性能越大（如事务并发严重性），为了平衡二者，一般建议设置的隔离级别为可重复读，MySQL默认的隔离级别也是可重复读。

隔离级别	脏读	不可重复读	幻读
读未提交	可能	可能	可能
读提交	不可能	可能	可能
可重复读	不可能	不可能	可能
串行化	不可能	不可能	不可能

- 读未提交

  读未提交，其实就是可以读到其他事务未提交的数据，但没有办法保证你读到的数据最终一定是提交后的数据，如果中间发生回滚，那就会出现脏数据问题，读未提交没办法解决脏数据问题。更别提可重复读和幻读了，想都不要想。

  例子：启动两个事务，分别为事务A和事务B，在事务A中使用 update 语句，修改 age 的值为10，初始是1 ，在执行完 update 语句之后，在事务B中查询 user 表，会看到 age 的值已经是 10 了，这时候事务A还没有提交，而此时事务B有可能拿着已经修改过的 age=10 去进行其他操作了。在事务B进行操作的过程中，很有可能事务A由于某些原因，进行了事务回滚操作，那其实事务B得到的就是脏数据了，拿着脏数据去进行其他的计算，那结果肯定也是有问题的。

- 读提交

  读提交就是一个事务只能读到其他事务已经提交过的数据，也就是其他事务调用 commit 命令之后的数据。

  读提交事务隔离级别是大多数流行数据库的默认事务隔离界别，比如 Oracle

  例子：同样开启事务A和事务B两个事务，在事务A中使用 update 语句将 id=1 的记录行 age 字段改为 10。此时，在事务B中使用 select 语句进行查询，我们发现在事务A提交之前，事务B中查询到的记录 age 一直是1，直到事务A提交，此时在事务B中 select 查询，发现 age 的值已经是 10 了。这就出现了一个问题，在同一事务中(本例中的事务B)，事务的不同时刻同样的查询条件，查询出来的记录内容是不一样的，事务A的提交影响了事务B的查询结果，这就是不可重复读，也就是读提交隔离级别。

- 可重复读

  上面说不可重复读是指同一事物不同时刻读到的数据值可能不一致。而可重复读是指，事务不会读到其他事务对已有数据的修改，即使其他事务已提交。也就是说，事务开始时读到的已有数据是什么，在事务提交前的任意时刻，这些数据的值都是一样的。但是，对于其他事务新插入的数据是可以读到的，这也就引发了幻读问题。

  例子：事务A开始后，执行 update 操作，将 age = 1 的记录的 name 改为“风筝2号”；事务B开始后，在事务执行完 update 后，执行 insert 操作，插入记录 age =1， name = 古时的风筝，这和事务A修改的那条记录值相同，然后提交。事务B提交后，事务A中执行 select，查询 age=1 的数据，这时，会发现多了一行，并且发现还有一条 name = 古时的风筝，age = 1 的记录，这其实就是事务B刚刚插入的，这就是幻读。

- 串行化

  串行化是4种事务隔离级别中隔离效果最好的，解决了脏读、可重复读、幻读的问题，但是效果最差，它将事务的执行变为顺序执行，与其他三个隔离级别相比，它就相当于单线程，后一个事务的执行必须等待前一个事务结束。

- 应用场景

  项目中是不用读未提交(Read UnCommitted)和串行化(Serializable)两个隔离级别，原因有二：

1. 采用读未提交(Read UnCommitted),一个事务读到另一个事务未提交读数据，这个不用多说吧，从逻辑上都说不过去！
2. 采用串行化(Serializable)，每个次读操作都会加锁，快照读失效，一般是使用mysql自带分布式事务功能时才使用该隔离级别！(笔者从未用过mysql自带的这个功能，因为这是XA事务，是强一致性事务，性能不佳！互联网的分布式方案，多采用最终一致性的事务解决方案！)

所以我们只用考虑read committed或者read repeatable。一般互联网项目都用读已提交这个。

参考

MysqI里面为什么用B+树？

上面有写

一条MySQL语句执行过程

首先了解一下mysql的架构

首先大方向上要分为两层，server层和存储引擎层。

- server层

  Server 层包括连接器、查询缓存、分析器、优化器、执行器等，涵盖 MySQL 的大多数核心服务功能，以及所有的内置函数（如日期、时间、数学和加密函数等），所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图等。

- 存储引擎层

  存储引擎层负责数据的存储和提取。其架构模式是插件式的，支持 InnoDB、MyISAM、Memory 等多个存储引擎，现在最常用的存储引擎是 InnoDB，它从 MySQL 5.5.5 版本开始成为了默认存储引擎。

下面来说一下过程：

①第一步连接器

连接到mysql服务器会首先碰到连接器，连接器负责跟客户端建立连接、获取权限、维持和管理连接。完成经典的 TCP 握手后，连接器就要开始认证你的身份，这个时候用的就是你输入的用户名和密码。

②查询缓存

MySQL 拿到一个查询请求后，会先到查询缓存看看，之前是不是执行过这条语句。之前执行过的语句及其结果可能会以 key-value 对的形式，被直接缓存在内存中。key 是查询的语句，value 是查询的结果。如果你的查询能够直接在这个缓存中找到 key，那么这个 value 就会被直接返回给客户端。

如果语句不在查询缓存中，就会继续后面的执行阶段。执行完成后，执行结果会被存入查询缓存中。

查询缓存也有不好的地方。查询缓存的失效非常频繁，只要有对一个表的更新，这个表上所有的查询缓存都会被清空。因此很可能你费劲地把结果存起来，还没使用呢，就被一个更新全清空了。对于更新压力大的数据库来说，查询缓存的命中率会非常低。除非你的业务就是有一张静态表，很长时间才会更新一次。比如，一个系统配置表，那这张表上的查询才适合使用查询缓存。

③分析器

分析器先会做“词法分析”。你输入的是由多个字符串和空格组成的一条 SQL 语句，MySQL 需要识别出里面的字符串分别是什么，代表什么。做完了这些识别以后，就要做“语法分析”。根据词法分析的结果，语法分析器会根据语法规则，判断你输入的这个 SQL 语句是否满足 MySQL 语法规则，只有遵循它的规则，才能获取到在它规则管理内的数据

④优化器

优化器是在表里面有多个索引的时候，决定使用哪个索引；或者在一个语句有多表关联（join）的时候，决定各个表的连接顺序。但是执行的效率会有不同，而优化器的作用就是决定选择使用哪一个方案。

⑤执行器

开始执行语句。开始执行的时候，要先判断一下你对这个表 有没有执行对应操作的权限，如果没有，就会返回没有权限的错误；如果有权限，就打开表继续执行。打开表的时候，执行器就会根据表的引擎定义，去使用这个引擎提供的接口。

如果表没有索引，会从第一行一行一行地读取，根据where后面的条件是否满足，如果有索引则会根据索引的规则去寻找，然后执行生成结果集。

分库分表有哪些方案？有什么区别？

mysql调优

简单的优化方式

MySQL 分析表

分析表用于分析和存储表的关键字分布，分析的结果可以使得系统得到准确的统计信息，使得 SQL 生成正确的执行计划。如果用于感觉实际执行计划与预期不符，可以执行分析表来解决问题，分析表语法如下：

```
analyze table cxuan005;
```

分析结果涉及到的字段属性如下

Table：表示表的名称；

Op：表示执行的操作，analyze 表示进行分析操作，check 表示进行检查查找，optimize 表示进行优化操作；

Msg_type：表示信息类型，其显示的值通常是状态、警告、错误和信息这四者之一；

Msg_text：显示信息。

对表的定期分析可以改善性能，应该成为日常工作的一部分。因为通过更新表的索引信息对表进行分析，可改善数据库性能。

MySQL 检查表

数据库经常可能遇到错误，比如数据写入磁盘时发生错误，或是索引没有同步更新，或是数据库未关闭 MySQL 就停止了。遇到这些情况，数据就可能发生错误： **Incorrect key file for table: ' '. Try to repair it**. 此时，我们可以使用 Check Table 语句来检查表及其对应的索引。

```
check table cxuan005;
```

检查表的主要目的就是检查一个或者多个表是否有错误。Check Table 对 MyISAM 和 InnoDB 表有作用。Check Table 也可以检查视图的错误。

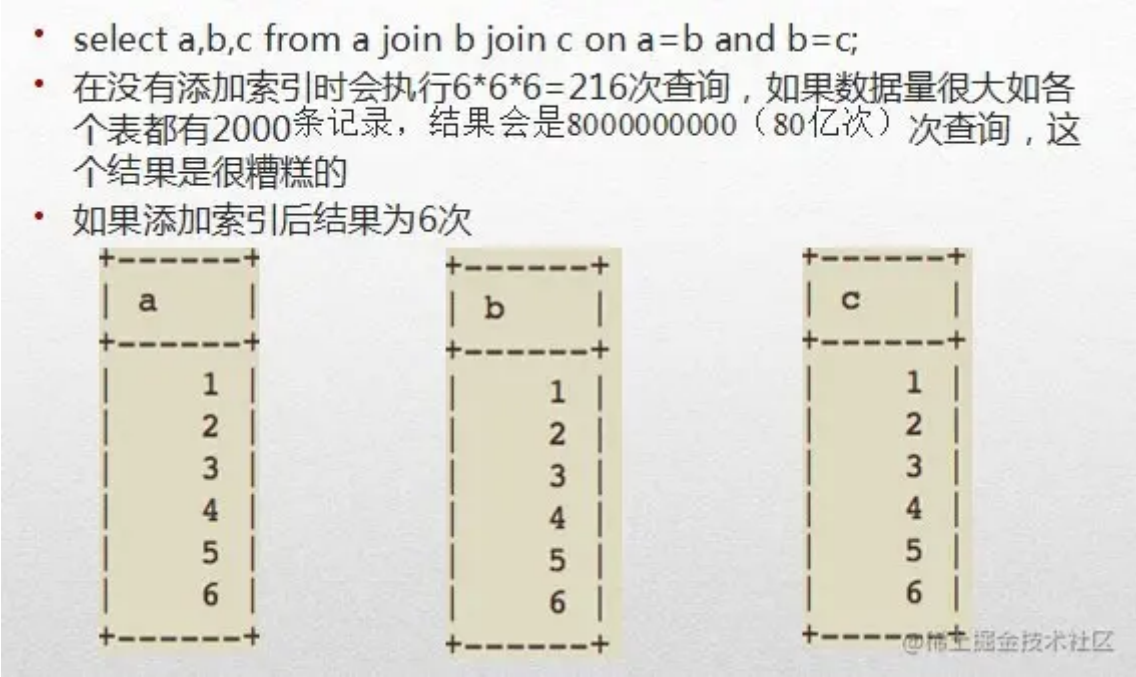
MySQL 优化表

MySQL 优化表适用于删除了大量的表数据，或者对包含 VARCHAR、BLOB 或则 TEXT 命令进行大量修改的情况。MySQL 优化表可以将大量的空间碎片进行合并，消除由于删除或者更新造成的空间浪费情况。它的命令如下

```
optimize table cxuan005;
```

查询时的优化

小表驱动大表



避免全表扫描

mysql在使用不等于(!=或者<>)的时候无法使用导致全表扫描。在查询的时候，如果对索引使用不等于的操作将会导致索引失效，进行全表扫描

避免mysql放弃索引查询

如果mysql估计使用全表扫描要比使用索引快，则不使用索引。（最典型的场景就是数据量少的时候）

使用覆盖索引，少使用select*

需要用到什么数据就查询什么数据，这样可以减少网络的传输和mysql的全表扫描。

尽量使用覆盖索引，比如索引为name，age，address的组合索引，那么尽量覆盖这三个字段之中的值，mysql将会直接在索引上取值（using index），并且返回值不包含不是索引的字段。

MySQL 对于千万级的大表要怎么优化？

https://www.zhihu.com/question/19719997

Redis

关系型数据库和非关系型数据库

关系型数据库

含义：采用关系模型来组织数据的数据库。简单说关系模型就是二维表格模型，而关系型数据库就是由二维表及其之间的联系组成的数据结构

优点：

1. 容易理解：二维表结构是非常贴近逻辑世界的一个概念，关系模型相对网状、层次等其他模型来说更容易理解
2. 使用方便：通用的SQL语言使得操作关系型数据库非常方便
3. 易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率

缺点：

1. 高并发读写需求。 网站的用户并发性非常高，往往达到每秒上万次读写请求，对于传统关系型数据库来说，硬盘I/O是一个很大的瓶颈
2. 海量数据的高效率读写。 网站每天产生的数据量是巨大的，对于关系型数据库来说，在一张包含海量数据的表中查询和修改，效率是非常低的
3. 高扩展性和可用性。在基于web的结构当中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增的时候，数据库却没有办法像web server和app server那样简单的通过添加更多的硬件和服务节点来扩展性能和负载能力。对于很多需要提供24小时不间断服务的网站来说，对数据库系统进行升级和扩展是非常痛苦的事情，往往需要停机维护和数据迁移。

一些不需要关系型数据库的情况：

1. 关系型数据库在对事物一致性的维护中有很大的开销，而现在很多web2.0系统对事物的读写一致性都不高
2. 对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比如发一条消息之后，过几秒乃至十几秒之后才看到这条动态是完全可以接受的
3. 任何大数据量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的复杂SQL报表查询，特别是SNS类型的网站（**SNS，专指社交网络服务，包括了社交软件和社交网站。** ）

非关系型数据库

含义：NoSQL一词首先是Carlo Strozzi在1998年提出来的，指的是他开发的一个没有SQL功能，轻量级的，开源的关系型数据库。但是NoSQL的发展慢慢偏离了初衷，我们要的不是“no sql”，而是“no relational(not noly)”，也就是我们现在常说的非关系型数据库了。

优点：

1. 格式灵活：存储数据的格式可以是key,value形式、文档形式、图片形式等等，文档形式、图片形式等等，使用灵活，应用场景广泛，而关系型数据库则只支持基础类型。
2. 速度快：nosql可以使用硬盘或者随机存储器作为载体，而关系型数据库只能使用硬盘；
3. 高扩展性。
4. 成本低：nosql数据库部署简单，基本都是开源软件。

缺点：

1. 不提供sql支持，学习和使用成本较高；
2. 无事务处理；
3. 数据结构相对复杂，复杂查询方面稍欠。

总结

关系型数据库的最大特点就是事务的一致性：传统的关系型数据库读写操作都是事务的，具有ACID的特点，这个特性使得关系型数据库可以用于几乎所有对一致性有要求的系统中，如典型的银行系统。

但是，在网页应用中，尤其是SNS应用中，一致性却不是显得那么重要，用户A看到的内容和用户B看到同一用户C内容更新不一致是可以容忍的，或者说，两个人看到同一好友的数据更新的时间差那么几秒是可以容忍的，因此，关系型数据库的最大特点在这里已经无用武之地，起码不是那么重要了。

相反地，关系型数据库为了维护一致性所付出的巨大代价就是其读写性能比较差，而像微博、facebook这类SNS的应用，对并发读写能力要求极高，关系型数据库已经无法应付因此，必须用新的一种数据结构存储来代替关系数据库。

关系数据库的另一个特点就是其具有固定的表结构，因此，其扩展性极差，而在SNS中，系统的升级，功能的增加，往往意味着数据结构巨大变动，这一点关系型数据库也难以应付，需要新的结构化数据存储。

于是，非关系型数据库应运而生，由于不可能用一种数据结构化存储应付所有的新的需求，因此，非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。

必须强调的是，数据的持久存储，尤其是海量数据的持久存储，还是需要一种关系数据库这员老将。

什么时候用redis什么时候用mysql？

Redis和MySQL不是相互替代的关系，而是相辅相成的，越来越多的项目组已经采用了redis+MySQL的架构来开发平台工具。

首先mysql是持久化数据库，是关系型数据库，是直接保存在硬盘上的。redis是非关系型数据库，是内存运行的数据存储获取工具。

但是数据量多少并不是redis和mysql选择的标准，因为都可以集群括展。

他们的使用场景是不同的：

1. 关系型数据库最重要的有两个点，第一是持久化存储的功能，即数据都存储在硬盘中。第二就是关系型数据库可以提供复杂的查询和统计功能。
2. 关系型数据库偏向于快速存取数据，用于实时响应要求高的场景，响应时间在毫秒级，通常作为热点数据的缓存使用。

数据多而且调用频繁的话，用mysql存储的话数据库连接被一直占用，其它的数据请求就进来了，导致连接超时，数据量大的话，数据库直接死机了。只能重启才能解决问题。这个时候如果把数据请求量大的数据放在redis中的话就可以分担一下mysql的压力，从而提高系统的性能，解决请求并发问题。

Redis持久化

RDB模式和AOF模式

在默认情况下，Redis将数据库快照保存在名为dump.rdb的二进制文件中

两个模式的选择：

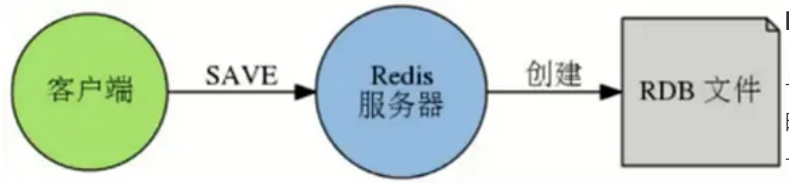
1. **如果主要充当缓存功能,或者可以承受数分钟数据的丢失, 通常生产环境一般只需启用RDB可,此也是默认值**
2. **如果数据需要持久保存,一点不能丢失,可以选择同时开启RDB和AOF,一般不建议只开启AOF**

RDB模式

具体原理有两种SAVE， BGSAVE

SAVE

SAVE是阻塞服务，在创建新文件dump.rdb替代旧文件时候无法响应客户端请求，生产环境中很少这样，一般都是停机维护时候才考虑



BGSAVE

与之对应的，BGSAVE就是非阻塞的。当创建RDB文件时候，会fork一个子进程来做这件事，同时父进程会正常接收处理来自客户端的请求。子进程执行RDB操作，处理完后会向父进程发送一个信号，通知父进程处理完毕，父进程用新的dump.rdb文件替代旧文件。可以说BGSAVE是一个异步命令。fork是指redis通过创建子进程来进行RDB操作，cow指的是**copy on write**，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分离开来。

RDB优点：

1. RDB保存了某个时间点的数据，可以保留多个备份。当出现问题的时候方便恢复到不同时间节点(多版本恢复)，同事文件格式支持不少第三方工具分析
2. RDB可以最大化Redis性能，父进程在保存RDB文件时候，唯一要做的就是fork一个子进程，接着子进程会接替保存工作，父进程无需执行任何磁盘io操作
3. RDB在大量数据的时候，恢复比AOF快
4. 文件单一紧凑，方便网络传输，适合灾难恢复

RDB缺点：

1. RDB不能实时保存数据，即这次保存数据和上次保存数据之间这段时间如果有新数据，可能会丢失这部分新数据。虽然Redis允许设置不同的保存点来控制保存RDB文件的频率，但是由于数据集的属性，这不是一个轻松地操作，因此会丢失好几分钟内的数据
2. 当数据量非常大的时候，从父进程fork子进程来保存RDB文件时候需要一点时间。当数据集很庞大的时候，fork会非常耗时，造成服务器在一定时间内停止处理客户端，会有毫秒或秒级响应。

AOF模式

AOF即Append Only File，需要手动开启，采用追加的方式保存，默认文件是appendonly.aof，记录所有写的命令。

AOF 方式不能保证绝对不丢失数据，目前常见的操作系统中，执行系统调用 write 函数，将一些内容写入到某个文件里面时，为了提高效率，系统通常不会直接将内容写入硬盘里 面，而是先将内容放入一个内存缓冲区（buffer）里面，等到缓冲区被填满，或者用户执行 fsync 调用和 fdatasync 调用时才将储存在缓冲区里的内容真正的写入到硬盘里，未写入磁盘之前，数据可能会丢失 。过程：

1. 命令追加：写到aof_buf中；
2. 写入文件：执行write操作；
3. 同步文件：同步到磁盘中。

优点：

1. 数据安全性相对较高，根据所使用的fsync策略(fsync是同步内存中redis所有已经修改的文件到存储设备)，默认是appendfsync everysec，即每秒执行一次 fsync,在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据(fsync会在后台线程执行，所以主线程可以继续努力地处理命令请求)
2. 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中不需要seek, 即使出现宕机现象，也不会破坏日志文件中已经存在的内容。然而如果本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，可以通过 redis-check-aof 工具来解决数据一致性的问题
3. Redis可以在 AOF文件体积变得过大时，自动地在后台对AOF进行重写,重写后的新AOF文件包含了恢复当前数据集所需的最小命令集合。整个重写操作是绝对安全的，因为Redis在创建新 AOF文件的过程中，append模式不断的将修改数据追加到现有的 AOF文件里面，即使重写过程中发停机，现有的 AOF文件也不会丢失。而一旦新AOF文件创建完毕，Redis就会从旧AOF文件切换到新AOF文件，并开始对新AOF文件进行追加操作。
4. AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，也可以通过该文件完成数据的重建AOF文件有序地保存了对数据库执行的所有写入操作，这些写入操作以Redis协议的格式保存，因此 AOF文件的内容非常容易被人读懂，对文件进行分析(parse)也很轻松。导出（export)AOF文件也非常简单:
5. 举个例子，如果你不小心执行了FLUSHALL.命令，但只要AOF文件未被重写，那么只要停止服务器，移除 AOF文件末尾的FLUSHAL命令，并重启Redis ,就可以将数据集恢复到FLUSHALL执行之前的状态。

缺点：

1. 即使有些操作是重复的也会全部记录，AOF 的文件大小要大于 RDB 格式的文件
2. AOF 在恢复大数据集时的速度比 RDB 的恢复速度要慢
3. 根据fsync策略不同,AOF速度可能会慢于RDB
4. bug 出现的可能性更多

Redis的数据结构讲一讲 + 使用场景

详细参考链接

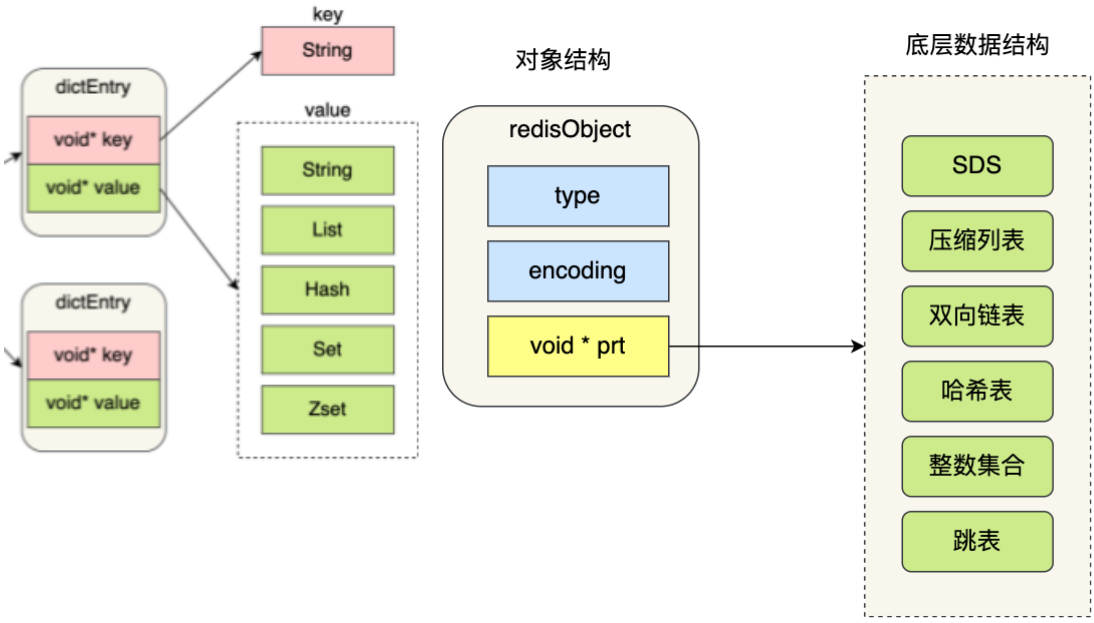
五种基本的数据类型：**String、Hash、List、Set、SortedSet**

更高级的有：**HyperLogLog、Geo、BloomFilter**

键值对数据库是怎么实现的？

> Redis 的键值对中的 key 就是字符串对象，而 **value 可以是字符串对象，也可以是集合数据类型的对象**，比如 List 对象、Hash 对象、Set 对象和 Zset 对象。

Redis 是使用了一个「哈希表」保存所有键值对，哈希表的最大好处就是让我们可以用 O(1) 的时间复杂度来快速查找到键值对。哈希表其实就是一个数组，数组中的元素叫做哈希桶。哈希桶存放的是指向键值对数据的指针,这样通过指针就能找到键值对数据，然后因为键值对的值可以保存字符串对象和集合数据类型的对象，所以键值对的数据结构中并不是直接保存值本身，而是保存了 void * key 和 void * value 指针，分别指向了实际的键对象和值对象，这样一来，即使值是集合数据，也可以通过 void * value 指针找到。如图：



特别说明：void * key 和 void * value 指针指向的是 **Redis 对象**，Redis 中的每个对象都由 redisObject 结构表示

string

String 是 Redis 最简单最常用的数据结构

Redis中的字符串，不是 C 语言中的字符串（即以空字符'\0'结尾的字符数组），是自己构建了一种名为**简单动态字符串（simple dynamic string,SDS）**的抽象类型，并将 SDS 作为 Redis的默认字符串表示。其数据结构如下所示：

```
/* Note: sdshdr5 is never used, we just access the flags byte directly.
 * However is here to document the layout of type 5 SDS strings. */
struct __attribute__ ((__packed__)) sdshdr5 {
    unsigned char flags; /* 3 lsb of type, and 5 msb of string length */
    char buf[];
};
struct __attribute__ ((__packed__)) sdshdr8 {
    uint8_t len; /* used */
    uint8_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};
struct __attribute__ ((__packed__)) sdshdr16 {
    uint16_t len; /* used */
    uint16_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};
struct __attribute__ ((__packed__)) sdshdr32 {
    uint32_t len; /* used */
    uint32_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};
struct __attribute__ ((__packed__)) sdshdr64 {
    uint64_t len; /* used */
    uint64_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};
```

上图中，uint8_t 表示8位无符号整数

为什么使用SDS?

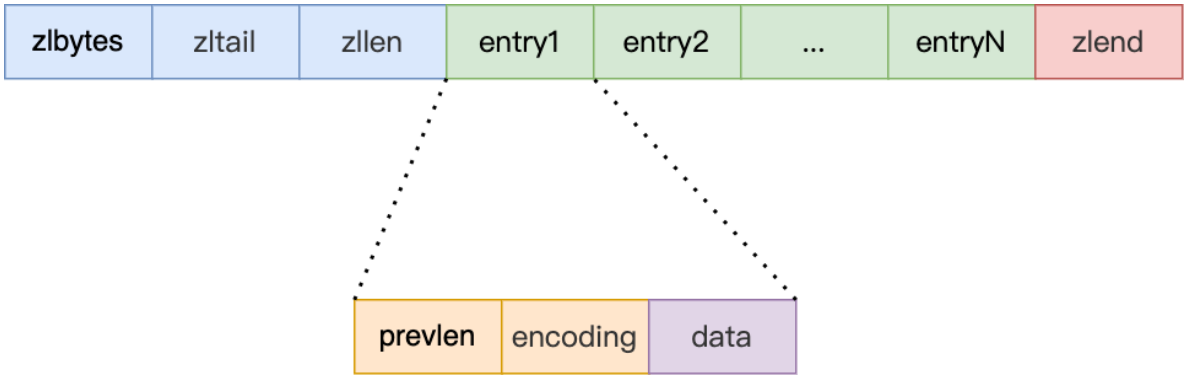
1. 由于 len 属性的存在，我们获取 SDS 字符串的长度只需要读取 len 属性，时间复杂度为 O(1)。而对于 C 语言，获取字符串的长度通常是经过遍历计数来实现的，时间复杂度为 O(n)。

2. C 语言中使用 `strcat` 函数来进行两个字符串的拼接，一旦没有分配足够长度的内存空间，就会造成缓冲区溢出。而对于 SDS 数据类型，在进行字符修改的时候，会首先根据记录的 len 属性检查内存空间是否满足需求，如果不满足，会进行相应的空间扩展，然后在进行修改操作，所以不会出现缓冲区溢出。
3. C语言由于不记录字符串的长度，所以如果要修改字符串，必须要重新分配内存（先释放再申请），因为如果没有重新分配，字符串长度增大时会造成内存缓冲区溢出，字符串长度减小时会造成内存泄露。而对于SDS，由于`len`属性和`alloc`属性的存在，对于修改字符串SDS实现了**空间预分配**和**惰性空间释放**两种策略
4. 因为C字符串以空字符作为字符串结束的标识，而对于一些二进制文件（如图片等），内容可能包括空字符串，因此C字符串无法正确存取；而所有 SDS 的API 都是以处理二进制的方式来处理 `buf` 里面的元素，并且 SDS 不是以空字符串来判断是否结束，而是以 len 属性表示的长度来判断字符串是否结束。

ZipList（压缩列表）

压缩列表的最大特点，就是它被设计成一种内存紧凑型的数据结构，占用一块连续的内存空间，不仅可以利用 CPU 缓存，而且会针对不同长度的数据，进行相应编码，这种方法可以有效地节省内存开销。

压缩列表的构成如下：



1. zlbytes，记录整个压缩列表占用对内存字节数；
2. zltail，记录压缩列表「尾部」节点距离起始地址由多少字节，也就是列表尾的偏移量；
3. zllen，记录压缩列表包含的节点数量；
4. zlend，标记压缩列表的结束点，固定值 0xFF（十进制255）。
5. prevlen，记录了「前一个节点」的长度；
6. encoding，记录了当前节点实际数据的类型以及长度；
7. data，记录了当前节点的实际数据；

当往压缩列表中插入数据时，压缩列表就会根据数据是字符串还是整数，以及数据的大小，会使用不同空间大小的 prevlen 和 encoding 这两个元素里保存的信息，这种根据数据大小和类型进行不同的空间大小分配的设计思想，正是 Redis 为了节省内存而采用的。

缺点：空间扩展操作也就是重新分配内存，因此连锁更新一旦发生，就会导致压缩列表占用的内存空间要多次重新分配，这就会直接影响到压缩列表的访问性能。所以说，虽然压缩列表紧凑型的内存布局能节省内存开销，但是如果保存的元素数量增加了，或是元素变大了，会导致内存重新分配，最糟糕的是会有「连锁更新」的问题。

hash表

Redis 散列可以存储多个键值对之间的映射。和字符串一样，散列存储的值既可以是字符串又可以是数值，并且用户同样可以对散列存储的数字值执行自增或自减操作。

整数集合

整数集合是 Set 对象的底层实现之一。当一个 Set 对象只包含整数值元素，并且元素数量不时，就会使用整数集这个数据结构作为底层实现。整数集合本质上是一块连续内存空间，它的结构定义如下：

```
typedef struct intset {
    //编码方式
    uint32_t encoding;
    //集合包含的元素数量
    uint32_t length;
    //保存元素的数组
    int8_t contents[];
} intset;
```

保存元素的容器是一个 contents 数组，虽然 contents 被声明为 int8_t 类型的数组，但是实际上 contents 数组并不保存任何 int8_t 类型的元素，contents 数组的真正类型取决于 intset 结构体里的 encoding 属性的值。

整数集合的升级操作

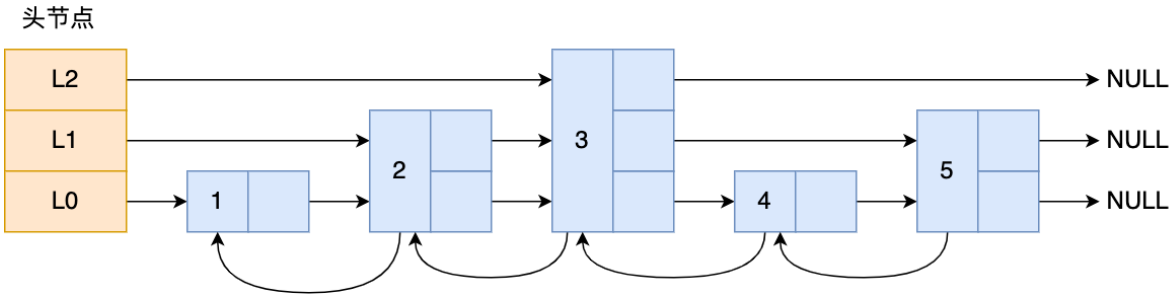
整数集合会有一个升级规则，就是当我们将一个新元素加入到整数集合里面，如果新元素的类型（int32_t）比整数集合现有所有元素的类型（int16_t）都要长时，整数集合需要先进行升级，也就是按新元素的类型（int32_t）扩展 contents 数组的空间大小，然后才能将新元素加入到整数集合里，当然升级的过程中，也要维持整数集合的有序性。

整数集合升级的过程不会重新分配一个新类型的数组，而是在原本的数组上扩展空间，然后在将每个元素按间隔类型大小分割。整数集合升级的好处是**节省内存资源**。

跳表

Redis 只有在 Zset 对象的底层实现用到了跳表，跳表的优势是能支持平均 O(logN) 复杂度的节点查找。Zset 对象是唯一一个同时使用了两个数据结构来实现的 Redis 对象，这两个数据结构一个是跳表，一个是哈希表。这样的好处是既能进行高效的范围查询，也能进行高效单点查询。

链表在查找元素的时候，因为需要逐一查找，所以查询效率非常低，时间复杂度是O(N)，于是就出现了跳表。**跳表是在链表基础上改进过来的，实现了一种「多层」的有序链表**，这样的好处是能快读定位数据。如图：



如果我们要在链表中查找节点 4 这个元素，只能从头开始遍历链表，需要查找 4 次，而使用了跳表后，只需要查找 2 次就能定位到节点 4，因为可以在头节点直接从 L2 层级跳到节点 3，然后再往前遍历找到节点 4。

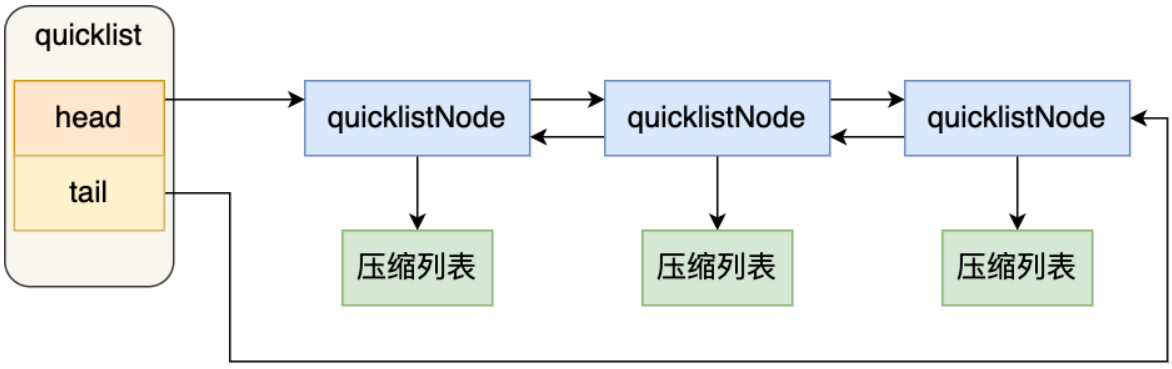
quicklist（快表）

在 Redis 3.0 之前，List 对象的底层数据结构是双向链表或者压缩列表。然后在 Redis 3.2 的时候，List 对象的底层改由 quicklist 数据结构实现。

其实 quicklist 就是「双向链表 + 压缩列表」组合，因为一个 quicklist 就是一个链表，而链表中的每个元素又是一个压缩列表。

在前面讲压缩列表的时候，我也提到了压缩列表的不足，虽然压缩列表是通过紧凑型的内存布局节省了内存开销，但是因为它的结构设计，如果保存的元素数量增加，或者元素变大了，压缩列表会有「连锁更新」的风险，一旦发生，会造成性能下降。

quicklist 解决办法，**通过控制每个链表节点中的压缩列表的大小或者元素个数，来规避连锁更新的问题。因为压缩列表元素越少或越小，连锁更新带来的影响就越小，从而提供了更好的访问性能。**



使用场景

Redis在互联网公司一般有以下应用:

- String：缓存、限流、计数器、分布式锁、分布式Session
- Hash：存储用户信息、用户主页访问量、组合查询
- List：微博关注人时间轴列表、简单队列
- Set：赞、踩、标签、好友关系
- Zset：排行榜

避免缓存穿透的利器之BloomFilter

本质就是用单向散列函数把数据映射到二进制向量中

布隆过滤器优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。即假阳性，就是说如果每一位为0表示一定没有，为1表示可能会出现没有的情况。

Bloom Filter跟单哈希函数Bit-Map不同之处在于：Bloom Filter使用了k个哈希函数，每个字符串跟k个bit对应。从而降低了冲突的概率。

因为布隆过滤器可以明确知道某个查询数据库不存在，所以可以过滤掉无效的查询到数据库，减少数据库的压力。

如果有大量的key需要设置同一时间过期，一般需要注意什么？

如果大量的key过期时间设置的过于集中，到过期的那个时间点，**Redis**可能会出现短暂的卡顿现象。严重的话会出现缓存雪崩，我们一般需要在时间上加一个随机值，使得过期时间分散一些。

缓存穿透，缓存击穿，缓存血崩

参考

- **缓存穿透**

定义：缓存穿透是指缓存和数据库都没有的数据，被大量请求，比如订单号不可能为`-1`，但是用户请求了大量订单号为`-1`的数据，由于数据不存在，缓存就也不会存在该数据，所有的请求都会直接穿透到数据库。
如果被恶意用户利用，疯狂请求不存在的数据，就会导致数据库压力过大，甚至垮掉。

解决：

- **缓存击穿**

  定义：缓存击穿是指数据库原本有得数据，但是缓存中没有，一般是缓存突然失效了，这时候如果有大量用户请求该数据，缓存没有则会去数据库请求，会引发数据库压力增大，可能会瞬间打垮。

  解决：

- **缓存血崩**

  定义：缓存雪崩是指缓存中有大量的数据，在同一个时间点，或者较短的时间段内，全部过期了，这个时候请求过来，缓存没有数据，都会请求数据库，则数据库的压力就会突增，扛不住就会宕机。

  解决：

redis高并发和快的原因

1. redis是基于内存的，内存的读写速度非常快；没有磁盘IO的开销
2. redis是单线程的，省去了很多上下文切换线程的时间；
3. redis使用多路复用技术，可以处理并发的连接。非阻塞IO 内部实现采用epoll，采用了epoll+自己实现的简单的事件框架。

Redis的单线程

Redis由很多个模块组成，如网络请求模块、索引模块、存储模块、高可用集群支撑模块、数据操作模块等。

很多人说Redis是单线程的，就认为Redis中所有模块的操作都是单线程的，其实这是不对的。我们所说的Redis单线程，指的是"其网络IO和键值对读写是由一个线程完成的"，也就是说，**Redis中只有网络请求模块和数据操作模块是单线程的。而其他的如持久化存储模块、集群支撑模块等是多线程的。**

原因：

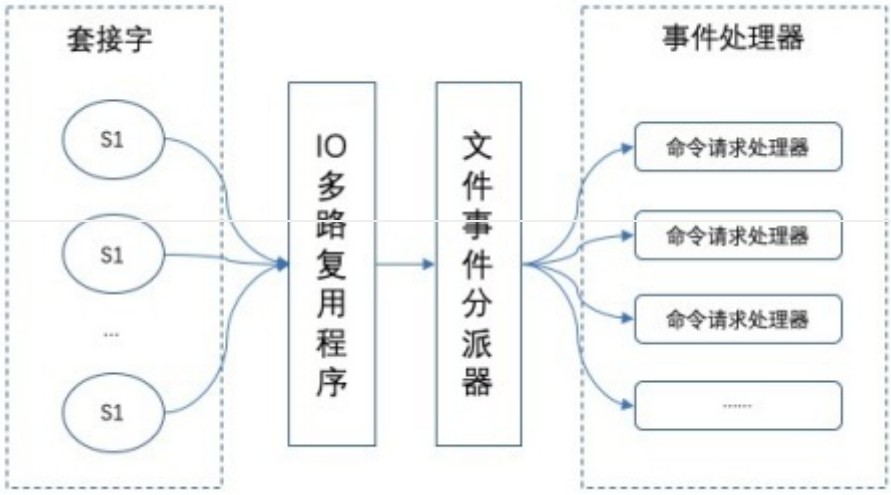
1. 锁带来的性能消耗。多线程可能会产生竞态条件，如果要对数据进行细粒度操作需要加锁，会加大开销增大延时。
2. CPU上下文切换带来的性能消耗。在多核CPU架构下，Redis如果在不同的核上运行，就需要频繁地进行上下文切换，这个过程会增加Redis的执行时间，客户端也会观察到较高的尾延迟了
3. redis是IO密集型程序，对于CPU是利用率没那么高，CPU并不是性能瓶颈
4. 在单线程中使用多路复用 I/O技术也能提升Redis的I/O利用率

总结：上面的原因说的也是多线程实现redis的劣势。我们可以从整体来看，一个计算机程序在执行的过程中，主要需要进行两种操作分别是读写操作和计算操作。其中读写操作主要是涉及到的就是I/O操作，其中包括网络I/O和磁盘I/O，计算操作主要涉及到CPU。**而多线程的目的，就是通过并发的方式来提升I/O的利用率和CPU的利用率。**那么，Redis需不需要通过多线程的方式来提升提升I/O的利用率和CPU的利用率呢？首先redis数据的存取对CPU的要求很小，所以说CPU不是redis性能的瓶颈。那么再看IO，提高IO效率多线程是一种方案，但不是唯一的一种，还有IO多路复用这个技术。所以在redis采用的是IO复用的技术来提高IO并发。

Redis的IO多路复用

redis是非阻塞IO+IO多路复用的技术来实现的

Linux的IO多路复用机制是指一个线程处理多个IO流，也就是select/epoll机制。

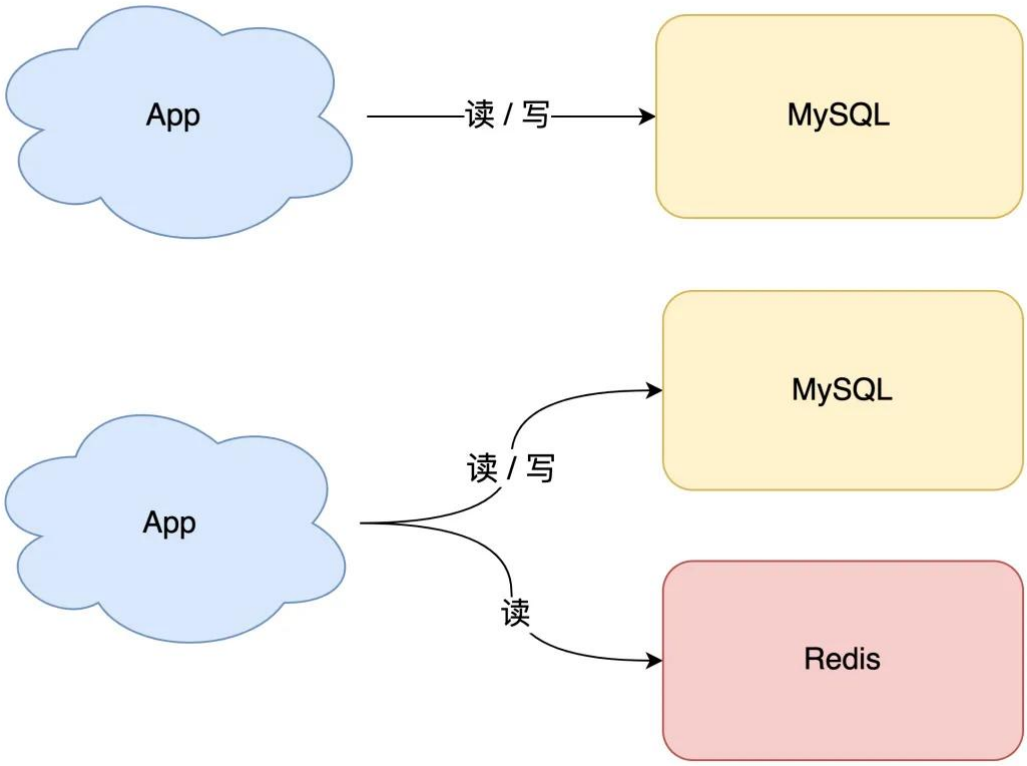


Redis怎么统计在线用户

参考链接

方案	特点
有序集合	能够同时储存在线用户的名单以及用户的上线时间，能够执行非常多的聚合计算操作，但是耗费的内存也非常多。
集合	能够储存在线用户的名单，也能够执行聚合计算，消耗的内存比有序集合少，但是跟有序集合一样，这个方案消耗的内存也会随着用户数量的增多而增多。
HyperLogLog	无论需要统计的用户有多少，只需要耗费 12 KB 内存，但由于概率算法的特性，只能给出在线人数的估算值，并且也无法获取准确的在线用户名单。
位图	在尽可能节约内存的情况下，记录在线用户的名单，并且能够对这些名单执行聚合操作。

redis，讲讲缓存一致性问题



果你的业务处于起步阶段，流量非常小，那无论是读请求还是写请求，直接操作数据库即可。但随着业务量的增长，你的项目请求量越来越大，这时如果每次都从数据库中读数据，那肯定会有性能问题。这个阶段通常的做法是，引入「缓存」来提高读性能，架构模型如上图。当下优秀的缓存中间件，当属 Redis 莫属，它不仅性能非常高，还提供了很多友好的数据类型，可以很好地满足我们的业务需求。但引入缓存之后，你就会面临一个问题：之前数据只存在数据库中，现在要放到缓存中读取，具体要怎么存呢?

最简单的方案：

1. 数据库的数据，全量刷入缓存(不设置失效时间)
2. 写请求只更新数据库，不更新缓存
3. 启动一个定时任务，定时把数据库的数据，更新到缓存中

缺点：

1. 缓存利用率低：不经常访问的数据，还一直留在缓存中
2. 数据不一致：因为是「定时」刷新缓存，缓存和数据库存在不一致(取决于定时任务的执行频率)

所以，这种方案一般更适合业务「体量小」，且对数据一致性要求不高的业务场景。

那么现在就有两个问题，缓存利用率和一致性问题

缓存利用率

想要缓存利用率「最大化」，只需要缓存中只保留最近访问的「热数据，可以这样做：

1. 写请求依旧只写数据库
2. 读请求先读缓存，如果缓存不存在，则从数据库读取，并重建缓存
3. 同时，写入缓存中的数据，都设置失效时间

这样一来，缓存中不经常访问的数据，随着时间的推移，都会逐渐「过期」淘汰掉，最终缓存中保留的，都是经常被访问的「热数据」，缓存利用率得以最大化。

一致性问题

大部分观点认为，做缓存不应该是去更新缓存，而是应该删除缓存，然后由下个请求去去缓存，发现不存在后再读取数据库，写入缓存。原因有如下两个：

1. 线程安全问题。有请求A和请求B进行更新操作，假如有以下情况：（1）线程A更新了数据库（2）线程B更新了数据库（3）线程B更新了缓存（4）线程A更新了缓存，于是这就出现请求A更新缓存应该比请求B更新缓存早才对，但是因为网络等原因，B却比A更早更新了缓存。这就导致了脏数据
2. 业务场景角度。如果你是一个写数据库场景比较多，而读数据场景比较少的业务需求，采用更新操作而不是删除，就会导致数据压根还没读到，缓存就被频繁的更新，浪费性能。其次，如果你写入数据库的值，并不是直接写入缓存的，而是要经过一系列复杂的计算再写入缓存。那么，每次写入数据库后，都再次计算写入缓存的值，无疑是浪费性能的。显然，删除缓存更为适合。

当数据发生更新时，我们不仅要操作数据库，还要一并操作缓存。具体操作就是，修改一条数据时，不仅要更新数据库，也要连带缓存一起更新。但数据库和缓存都更新，又存在先后问题，那对应的方案就有 2 个：

- 先更新缓存，后更新数据库

  如果缓存更新成功了，但数据库更新失败，那么此时缓存中是最新值，但数据库中是「旧值」。虽然此时读请求可以命中缓存，拿到正确的值，但是，一旦缓存「失效」，就会从数据库中读取到「旧值」，重建缓存也是这个旧值。这时用户会发现自己之前修改的数据又「变回去」了，对业务造成影响。

- 先更新数据库，后更新缓存

  如果数据库更新成功了，但缓存更新失败，那么此时数据库中是最新值，缓存中是「旧值」。之后的读请求读到的都是旧数据，只有当缓存「失效」后，才能从数据库中得到正确的值。这时用户会发现，自己刚刚修改了数据，但却看不到变更，一段时间过后，数据才变更过来，对业务也会有影响。

上述这两个方案都不行，以下给出解决方案：

- 如果先更新缓存，后更新数据库的话，使用延时双删策略

  延时双删的方案的思路是，为了避免更新数据库的时候，其他线程从缓存中读取不到数据，就在更新完数据库之后，再sleep一段时间，然后再次删除缓存。sleep的时间要对业务读写缓存的时间做出评估，sleep时间大于读写缓存的时间即可。

- 如果先更新数据库，后更新缓存的话，设置缓存过期时间，消息队列

  设置过期时间：每次放入缓存的时候，设置一个过期时间，比如5分钟，以后的操作只修改数据库，不操作缓存，等待缓存超时后从数据库重新读取。如果对于一致性要求不是很高的情况，可以采用这种方案。

  消息队列：先更新数据库，成功后往消息队列发消息，消费到消息后再删除缓存，借助消息队列的重试机制来实现，达到最终一致性的效果。

基于以上原因，redis官方选择了更简单、更快的方法，不支持错误回滚。这样的话，如果在我们的业务场景中需要保证原子性，那么就要求了开发者通过其他手段保证命令全部执行成功或失败，例如在执行命令前进行参数类型的校验，或在事务执行出现错误时及时做事务补偿。

Redis的事务满足原子性吗？

redis中的事务是不满足原子性的，在运行错误的情况下，并没有提供类似数据库中的回滚功能。那么为什么redis不支持回滚呢，官方文档给出了说明，大意如下：

1. redis命令失败只会发生在语法错误或数据类型错误的情况，这一结果都是由编程过程中的错误导致，这种情况应该在开发环境中检测出来，而不是生产环境
2. 不使用回滚，能使redis内部设计更简单，速度更快
3. 回滚不能避免编程逻辑中的错误，如果想要将一个键的值增加2却只增加了1，这种情况即使提供回滚也无法提供帮助

redis有那些命令是原子指令

为何Redis使用跳表而非红黑树实现SortedSet？

参考链接

首先要知道红黑树和跳表的插入删除，删除，查找时间复杂度是一样的。

redis作者说了三个原因：

1. 范围查找。跳表在区间查询的时候效率是高于红黑树的，跳表进行查找O(logn)的时间复杂度定位到区间的起点，然后在原始链表往后遍历就可以了 ，其他插入和单个条件查询，更新两者的复杂度都是相同的O(logn)。在平衡树上，我们找到指定范围的小值之后，还需要以中序遍历的顺序继续寻找其它不超过大值的节点。如果不对平衡树进行一定的改造，这里的中序遍历并不容易实现。
2. 易于实现
3. 从内存占用上来说，skiplist比平衡树更灵活一些。一般来说，平衡树每个节点包含2个指针（分别指向左右子树），而skiplist每个节点包含的指针数目平均为1/(1-p)，具体取决于参数p的大小。如果像Redis里的实现一样，取p=1/4，那么平均每个节点包含1.33个指针，比平衡树更有优势。

redis如何实现消息队列

消息队列是指利用高效可靠的消息传递机制进行与平台无关的数据交流，并基于数据通信来进行分布式系统的集成。

回顾下我们使用的消息队列有如下特点：

1. 三个角色：生产者、消费者、消息处理中心
2. 异步处理模式：生产者将消息发送到一条虚拟的通道(消息队列)上，而无须等待响应。消费者则订阅或是监听该通道，取出消息。两者互不干扰，甚至都不需要同时在线，也就是我们说的松耦合
3. 可靠性：消息要可以保证不丢失、不重复消费、有时可能还需要顺序性的保证

大key如何处理