MySQL的索引

原创 小孩子 我们都是小青蛙 2018-04-08

大多数人都在痴迷于XXX技术怎么用，你会XXX技术么？我们貌似被这些技术蒙蔽了双眼，成为了一个别人开发的软件的使用者，而忘记了这背后其实都是010101在作怪。为什么我们现在用的大多数所谓的XXX技术都是歪果仁弄出来的呢？很大一部分原因是因为我们太多的程序猿知其然不知其所以然，而国内的恶心技术书籍又让这些程序猿在学习这些原理的时候举步维艰，这包括把书后边介绍的概念在前边就使用到了，一个概念还没解释清楚就急于用这个概念解释别的概念，原理描述不清晰，好多时候一句话草草了事，在短时间内出现大量概念，目的就是让你懵逼，这样才能展示自己的水平。在这个互联网公司天天强调用户体验的年代，却偏偏没有人去关心程序猿的用户体验，这是何等荒唐。

本文是一份没有经过小白测试的草稿，各位在读文章的时候发现任何原理性错误或者阅读体验不畅的问题都可以私聊我，之后会升级体验以让更多的人收益。如果你觉得文章内容不错，有助你升职加薪，请关注转发，我现在最大的苦恼就是倾尽心血写的文章没多少人能看到，谢谢各位。下边是几条阅读建议：

1. 最好使用电脑观看。
2. 如果你非要使用手机观看，那请把字体调整到最小，这样观看效果会好一些。
3. 碎片化阅读并不会得到真正的知识提升，要想有提升还得找张书桌认认真真看一会书，或者我们公众号的文章😊😊。
4. 如果觉得不错，各位帮着转发转发，如果觉得有问题或者写的哪不清晰，务必私聊我～
5. 这篇文章是建立在之前的两篇文章的基础上的，如果你没读过，最好先读一下，要不然本篇文章里边出现的一些概念你可能理解不了，前两篇文章的链接链接是：InnoDB记录存储结构，InnoDB数据页结构。

上集回顾

上集我们详细唠叨了 `InnoDB` 数据页的7个组成部分，知道了各个数据页可以组成一个 双向链表 ，而每个数据页中的记录又可以组成一个 单向链表 ，每个数据页都会为存储在它里边儿的记录生成一个 页目录 ，在通过主键查找某条记录的时候可以在 页目录 中使用二分法快速定位到对应的槽，然后再遍历该槽对应分组中的记录即可快速找到指定的记录。页和记录的关系的示意图如下：

没有索引的查找

本集的主题是 索引 ，在正式介绍 索引 之前，我们需要了解一下没有索引的时候是怎么查找记录的。为了方便大家理解，我们下边先只唠叨搜索条件为对某个列精确匹配的情况，所谓精确匹配，就是搜索条件中用等于 = 连接起的表达式，比如这样：

```
SELECT [列名列表] FROM 表名 WHERE 列名 = xxx;
```

在一个页中的查找

假设目前表中的记录比较少，所有的记录都可以被存放到一个页中，在查找记录的时候可以根据搜索条件的不同分为两种情况：

- 以主键为搜索条件

  这个查找过程我们已经很熟悉了，可以在 页目录 中使用二分法快速定位到对应的槽，然后再遍历该槽对应分组中的记录即可快速找到指定的记录。

- 以其他列作为搜索条件

  对非主键列的查找的过程可就不这么幸运了，因为在数据页中并没有对非主键列建立所谓的 页目录 ，所以我们无法通过二分法快速定位相应的 槽 。这种情况下只能从 最小记录 开始依次遍历单链表中的每条记录，然后对比每条记录是不是符合搜索条件。很显然，这种查找的效率是非常低的。

在很多页中查找

大部分情况下我们表中存放的记录都是非常多的，需要好多的数据页来存储这些记录。在很多页中查找记录的话可以分为两个步骤：

1. 定位到记录所在的页。
2. 从所在的页内中查找相应的记录。

不论是根据主键列或者其他列的值进行查找，由于我们并不能快速的定位到记录所在的页，所以只能从第一个页沿着双向链表一直往下找，在每一个页中根据我们上边已经唠叨过的查找方式去查找指定的记录。因为要遍历所有的数据页，所以这种方式显然是超级耗时的，如果一个表有一亿条记录，使用这种方式去查找记录那要等到猴年马月才能等到查找结果。

索引

为了故事的顺利发展，我们先建一个表：

```
mysql> CREATE TABLE index_demo(
    ->     c1 INT,
    ->     c2 INT,
    ->     c3 CHAR(1),
    ->     PRIMARY KEY(c1)
    -> ) ROW_FORMAT = Compact;
Query OK, 0 rows affected (0.03 sec)
mysql>
```

这个新建的 `index_demo` 表中有2个 `INT` 类型的列，1个 `CHAR(1)` 类型的列，而且我们规定了 `c1` 列为主键，这个表使用 `Compact` 行格式来实际存储记录的。为了我们理解上的方便，我们简化了一下 `index_demo` 表的行格式示意图：

我们只在示意图里展示记录的这几个部分：

- `record_type`：记录头信息的一项属性，表示记录的类型，`0` 表示普通记录、`2` 表示最小记录、`3` 表示最大记录、`1` 我们还没用过，等会再说～
- `next_type`：记录头信息的一项属性，表示下一条地址的偏移量，为了方便大家理解，我们都会用箭头来表明下一条记录是谁。
- 各个列的值：就是各个数据列的值，其中我们用橘黄色的格子代表 `c1` 列，深蓝色的格子代表 `c2` 列，红色格子代表 `c3` 列。
- 其他信息：除了上述3种信息以外的所有信息，包括其他隐藏列的值以及记录的额外信息。

为了节省篇幅，我们之后的示意图中会把记录的 其他信息 这个部分省略掉，因为它占地方并且不会有什么观赏效果。另外，为了方便理解，我们觉得把记录竖着放看起来感觉更好，所以将记录格式示意图的 其他信息 去掉并把它竖起来的效果就是这样：

把一些记录放到页里边的示意图就是：

一个简单的索引方案

回到正题，我们为什么要遍历所有的数据页呢？因为各个页中的记录并没有规律，我们并不知道我们的搜索条件匹配哪些页中的记录，所以 ***不得不*** 依次遍历。所以如果我们想快速的定位到需要查找的记录在哪些数据页中该咋办？就像为数据页中的记录建立一个目录一样，我们也可以为所有的数据页建立一个目录呀，建这个目录必须完成下边这些事儿：

- 下一个数据页的主键值必须大于上一个页中的主键值。

  其实这句话的完整表述是这样的：下一个数据页中用户记录的主键值必须大于上一个页中用户记录的主键值。为了故事的顺利发展，我们这里需要做一个假设：假设我们的每个数据页最多能存放3条记录（实际上一个数据页非常大，可以存放下好多记录）。有了这个假设之后我们向 `index_demo` 表插入3条记录：

```
mysql> INSERT INTO index_demo VALUES(1, 4, 'u'), (3, 9, 'd'), (5, 3, 'y');
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql>
```

  那么这些记录已经按照主键值的大小串联成一个单向链表了，如图所示：

从途中可以看出来， `index_demo` 表中的3条记录都被插入到了编号为 `10` 的数据页中了。此时我们再来插入一条记录：

```
mysql> INSERT INTO index_demo VALUES(4, 4, 'a');
Query OK, 1 row affected (0.00 sec)

mysql>
```

因为 页10 最多只能放3条记录，所以我们不得不再分配一个新页：

咦？怎么分配的页号是 28 呀，不应该是 11 么？需要注意的一点是，新分配的数据页编号可能并不是连续的，也就是说我们使用的这些页在存储空间里可能并不挨着。它们只是通过维护着上一个页和下一个页的编号而建立了链表关系。另外， 页10 中用户记录最大的主键值是 5 ，而 页28 中有一条记录的主键值是 4 ，因为 5 > 4 ，所以这就不符合下一个数据页的主键值必须大于上一个页中的主键值的要求，所以在插入主键值为 4 的记录的时候需要伴随着一次记录移动，也就是把主键值为 5 的记录移动到 页28 中，然后再把主键值为 4 的记录插入到 页10 中，这个过程的示意图如下：

这个过程表明了在对页中的记录进行增删改操作的过程中，我们必须通过一些诸如记录移动的操作来始终保证这个状态一直成立：下一个数据页的主键值必须大于上一个页中的主键值。

- 给所有的页建立一个目录项。

  由于数据页的编号可能并不是连续的，所以在向 index_demo 表中插入许多条记录后，可能是这样的效果：

因为这些 16KB 的页在物理存储上并不挨着，所以如果想从这么多页中根据主键值快速定位某些记录所在的页，我们需要给它们做个目录，每个页对应一个目录项，每个目录项包括下边两个部分：

- 页的用户记录中最小的主键值，我们用 key 来表示。
- 页号，我们用 page_no 表示。

所以我们为上边几个页做好的目录就像这样子：

以 页28 为例，它对应 目录项2 ，这个目录项中包含着该页的页号 28 以及该页中用户记录的最小主键值 5 。我们只需要把几个目录项在物理存储器上连续存储，比如把他们放到一个数组里，就可以实现根据主键值快速查找某条记录的功能了。比方说我们想找主键值为 20 的记录，具体查找过程分两步：

1. 先从目录项中根据二分法快速确定出主键值为 20 的记录在 目录项3 中（因为 12 < 20 < 209 ），它对应的页是 页9 。

2. 再根据前边说的在页中查找记录的方式去 页9 中定位具体的记录。

至此，针对数据页做的简易目录就搞定了。不过忘了说了，这个 目录 有一个别名，称为 索引 。

InnoDB中的索引方案

上边之所以称为一个简易的索引方案，是因为我们假设所有目录项都可以在物理存储器上连续存储，但是这样做有几个问题：

- InnoDB 是使用页来作为管理存储空间的基本单位，也就是最多能保证 16KB 的连续存储空间，而随着表中记录数量的增多，需要非常大的连续的存储空间才能把所有的目录项都放下，这对记录数量非常多的表是不现实的。

- 我们时常会对记录进行增删，假设我们把 页28 中的记录都删除了， 页28 也就没有存在的必要了，那意味着 目录项2 也就没有存在的必要了，这就需要把 目录项2 后的目录项都向前移动一下，这种牵一发而动全身的设计不是什么好主意～

所以，设计 InnoDB 的大叔们需要一种可以灵活管理所有 目录项 的方式。他们灵光乍现，忽然发现这些 目录项 其实长得跟我们的用户记录差不多，只不过 目录项 中的两个列是 主键 和 页号 而已，所以他们复用了之前存储用户记录的数据页来存储目录项，为了和用户记录做一下区分，我们把这些用来表示目录项的记录称为 目录项记录 。那 InnoDB 怎么区分一条记录是普通的用户记录还是 目录项记录 呢？别忘了记录头信息里的 record_type 属性，它的各个取值代表的意思如下：

- 0 ：普通的用户记录
- 1 ：目录项记录
- 2 ：最小记录
- 3 ：最大记录

哈哈，原来这个值为 1 的 record_type 是这个意思呀，我们把前边使用到的目录项放到数据页中的样子就是这样：

从图中可以看出来，我们新分配了一个编号为 30 的页来专门存储 目录项记录 。这里再次强调一遍 目录项记录 和普通的 用户记录 的不同点：

- 目录项记录 的 record_type 值是1，而普通用户记录的 record_type 值是0。
- 目录项记录 只有主键值和页的编号两个列，而普通的用户记录的列是用户自己定义的，可能包含很多列，另外还有 InnoDB 自己添加的隐藏列。
- 还记得我们之前在唠叨记录头信息的时候说过一个叫 min_rec_mask 的属性么，只有在存储 目录项记录 的页中的主键值最小的 目录项记录 的 min_rec_mask 值为 1 ，其他别的记录的 min_rec_mask 值都是 0 。

除了上述几点外，这两者就没啥差别了，它们用的是一样的数据页（页面类型都是 0x45BF ，这个属性在 Page Header 中，忘了的话可以翻到前边的文章看），页的组成结构也是一样一样的（就是我们前边介绍过的7个部分），都会为主键值生成 Page Directory （页目录），以加快在页内的查询速度。所以现在根据某个主键值去查找记录的步骤可以大致拆分成下边两步，以查找主键为 20 的记录为例（因为都是从一个页中通过主键查某条记录，所以都可以使用 Page Directory 通过二分法而实现快速查找）：

1. 先到存储 目录项记录 的页中通过二分法快速定位到对应目录项，因为 12 < 20 < 209 ，所以定位到对应的记录所在的页就是 页9 .

2. 从 页9 中根据二分法快速定位到主键值为 20 的用户记录。

虽然说 目录项记录 中只存储主键值和对应的页号，比用户记录需要的存储空间小多了，但是不论怎么说一个页只有 16KB 大小，能存放的 目录项记录 也是有限的，那如果表中的数据太多，以至于一个数据页不足以存放所有的 目录项记录 ，该咋办呢?

当然是再多整一个存储 目录项记录 的页喽～ 为了大家更好的理解如何新分配一个 目录项记录 页的过程，我们假设一个存储 目录项记录 的页最多只能存放4条 目录项记录 （请注意是假设哦，真实情况下可以存放好多条的），所以如果此时我们再向上图中插入一条主键值为 320 的用户记录的话，那就需要一个分配一个新的存储 目录项记录 的页喽：

从图中可以看出，我们插入了一条主键值为 320 的用户记录之后新生成了2个数据页，以查找主键值为 20 的记录为例：

- 为存储该用户记录而新生成了 页31 。
- 因为原先存储 目录项记录 的 页30 的容量已满（我们前边假设只能存储4条 目录项记录 ），所以不得不新生成了一个 页32 来存放 页31 对应的目录项。

因为存储 目录项记录 的页不止一个，所以如果我们想根据主键值查找一条用户记录大致需要3个步骤：

1. 确定 目录项记录 页

   我们现在的存储 目录项记录 的页有两个，即 页30 和 页32 ， 又因为 页30 表示的目录项的主键值的范围是 [1, 320)， 页32 表示的目录项的主键值不小于 320 ，所以主键值为 20 的记录对应的目录项记录在 页30 中。

2. 通过 目录项记录 页确定用户记录真实所在的页。

   在一个存储 目录项记录 中定位一条目录项记录的方式说过了，不赘述了～

3. 在真实存储用户记录的页中定位到具体的记录。

> 在一个存储用户记录的页中定位一条真实的用户记录的方式已经说过200遍了，你再不会我就，我就，我就求你到上一篇唠叨数据页结构的文章中多看几遍，求你了～

那么问题来了，在这个查询步骤的第1步中我们需要定位存储 目录项记录 的页，但是这些页在存储空间中也可能不挨着，如果我们表中的数据非常多则会产生很多存储 目录项记录 的页，那我们怎么根据主键值快速定位一个存储 目录项记录 的页呢？其实也简单，为这些存储 目录项记录 的页再生成一个更高级的目录，就像是一个多级目录一样，大目录里嵌套小目录，小目录里才是实际的数据，所以现在各个页的示意图就是这样子：

如图，我们生成了一个存储更高级目录项的 页33 ，这个页中的两条记录分别代表 页30 和 页32 ，如果用户记录的主键值在 [1, 320) 之间，则到 页30 中查找更详细的 目录项记录 ，如果主键值不小于 320 的话，就到 页32 中查找更详细的 目录项记录 。不过这张图好漂亮喔，随着表中记录的增加，这个目录的层级会继续增加，如果简化一下，那么我们可以用下边这个图来描述它：

这玩意儿像不像一个倒过来的 树 呀！其实这是一种组织数据的形式，或者说是一种数据结构，它的名称是 B+ 树。

小贴士：
为啥叫`B+`呢，`B`树是个啥？喔对不起，这不是我们讨论的范围，你可以去找一本数据结构或算法的书来看。什么？数据结构的书看不懂？等我～

因为我们把数据页都存放到 B+ 树这个数据结构中了，所以我们也把我们的数据页称为 节点 。从图中可以看出来，我们的实际用户记录其实都存放在B+树的最底层的节点上，这些节点也被称为 叶子节点 或 叶节点 ，其余的节点都是用来存放 目录项 的，这些节点统统被称为 内节点 或者说 非叶节点 。其中最上边的那个节点也称为 根节点 。

从图中可以看出来，一个 B+ 树的节点其实可以分成好多层，设计 InnoDB 的大叔们为了讨论方便，规定最下边的那层，也就是存放我们用户记录的那层为第 0 层，之后依次往上加。上边我们做了一个非常极端的假设，存放用户记录的页最多存放3条记录，存放目录项记录的页最多存放4条记录，其实真实环境中一个页存放的记录数量是非常大的，假设，假设，假设所有的数据页，包括存储真实用户记录和目录项记录的页，都可以存放 1000 条记录，那么：

- 如果 B+ 树只有1层，也就是只有1个用于存放用户记录的节点，最多能存放 1000 条记录。
- 如果 B+ 树有2层，最多能存放 1000×1000=1000000 条记录。
- 如果 B+ 树有3层，最多能存放 1000×1000×1000=1000000000 条记录。

- 如果 B+ 树有4层，最多能存放 1000×1000×1000×1000=1000000000000 条记录。哇咔咔～这么多的记录！！！

你的表里能存放 1000000000000 条记录么？所以一般情况下，我们用到的 B+ 树都不会超过4层，那我们通过主键去查找某条记录最多只需要做4个页面内的查找，又因为在每个页面内有所谓的 Page Directory （页目录），所以在页面内也可以通过二分法实现快速定位记录，这不是很diao么，哈哈！

聚簇索引

我们上边介绍的 B+ 树本身就是一个目录，或者说本身就是一个索引。它有两个特点：

1. 使用记录主键值的大小进行记录和页的排序，这包括三个方面的含义：

    - 页内的记录是按照主键的大小顺序排成一个单向链表。
    - 各个存放用户记录的页也是根据页中记录的主键大小顺序排成一个双向链表。
    - 各个存放目录项的页也是根据页中记录的主键大小顺序排成一个双向链表。

2. B+ 树的叶子节点存储的是完整的用户记录。

    所谓完整的用户记录，就是指这个记录中存储了所有列的值。

我们把具有这两种特性的 B+ 树称为 聚簇索引 ，所有完整的用户记录都存放在这个 聚簇索引 的叶子节点处。这种 聚簇索引 并不需要我们在 MySQL 语句中显式的去创建， InnoDB 存储引擎会自动的为我们创建聚簇索引。另外有趣的一点是，在 InnoDB 存储引擎中， 聚簇索引 就是数据的存储方式（所有的用户记录都存储在了 叶子节点 ），也就是所谓的索引即数据。

二级索引

大家有木有发现，上边介绍的 聚簇索引 只能在搜索条件是主键值时才能发挥作用，因为 B+ 树中的数据都是按照主键进行排序的。那如果我们想以别的列作为搜索条件该咋办呢？难道只能从头到尾沿着链表依次遍历记录么？

不，我们可以多建几棵 B+ 树，不同的 B+ 树中的数据采用不同的排序规则。比方说我们用 c2 列的大小作为数据页、页中记录的排序规则，再建一棵 B+ 树，效果如下图所示：

这个 B+ 树与上边介绍的聚簇索引有几处不同：

- 使用记录 c2 列的大小进行记录和页的排序，这包括三个方面的含义：
  - 页内的记录是按照 c2 列的大小顺序排成一个单向链表。
  - 各个存放用户记录的页也是根据页中记录的 c2 列大小顺序排成一个双向链表。
  - 各个存放目录项的页也是根据页中记录的 c2 列大小顺序排成一个双向链表。
- B+ 树的叶子节点存储的并不是完整的用户记录，而只是 c2列+主键 这两个列的值。
- 目录项记录中不再是 主键+页号 的搭配，而变成了 c2列+页号 的搭配。

所以如果我们现在想通过 c2 列的值查找某些记录的话就可以使用我们刚刚建好的这个 B+ 树了，以查找 c2 列的值为 4 的记录为例，查找过程如下：

1. 确定 目录项记录 页

   根据 根页面 ，也就是 页44 ，可以快速定位到 目录项记录 所在的页为 页42 （因为 2 < 4 < 9 ）。

2. 通过 目录项记录 页确定用户记录真实所在的页。

在 页42 中可以快速定位到实际存储用户记录的页，但是由于 c2 列并没有唯一性约束，所以 c2 列值为 4 的记录可能分布在多个数据页中，又因为 2 < 4 ≤ 4 ，所以确定实际存储用户记录的页在 页34 和 页35 中。

3. 在真实存储用户记录的页中定位到具体的记录。

   到 页34 和 页35 中定位到具体的记录。

4. 但是这个 B+ 树的叶子节点中的记录只存储了 c2 和 c1 （也就是 主键 ）两个列，所以我们必须再根据主键值去聚簇索引中再查找一遍完整的用户记录。

各位各位，看到了么？我们根据这个以 c2 列大小排序的 B+ 树只能确定我们要查找记录的主键值，所以如果我们想根据 c2 列的值查找到完整的用户记录的话，仍然需要到 聚簇索引 中再查一遍，这个过程也被称为 回表 。也就是根据 c2 列的值查询一条完整的用户记录需要使用到 2 棵 B+ 树！！！

为什么我们还需要一次 回表 操作呢？直接把完整的用户记录放到 叶子节点 不就好了么？你说的对，如果把完整的用户记录放到 叶子节点 是可以不用 回表 ，但是太占地方了呀～相当于每建立一棵 B+ 树都需要把所有的用户记录再都拷贝一遍，这就有点太浪费存储空间了。因为这种按照 非主键列 建立的 B+ 树需要一次 回表 操作才可以定位到完整的用户记录，所以这种 B+ 树也被称为 二级索引 （英文名 secondary index ），或者 辅助索引 。由于我们使用的是 c2 列的大小作为 B+ 树的排序规则，所以我们也称这个 B+ 树为为c2列建立的索引。

联合索引

我们也可以同时以多个列的大小作为排序规则，也就是同时为多个列建立索引，比方说我们想让 B+ 树按照 c2 和 c3 列的大小进行排序，这个包含两层：

- 先把各个记录和页按照 c2 列进行排序。
- 在记录的 c2 列相同的情况下，采用 c3 列进行排序

为 c2 和 c3 列建立的索引的示意图如下：

如图所示，我们需要注意一下几点：

- 每条 目录项记录 都由 c2 、 c3 、 页号 这三个部分组成，各条记录先按照 c2 列的值进行排序，如果记录的 c2 列相同，则按照 c3 列的值进行排序。
- B+ 树叶子节点处的用户记录由 c2 、 c3 和主键 c1 列组成。

千万要注意一点，以c2和c3列的大小为排序规则建立的 B+ 树称为 联合索引 ，它的意思与分别为c2和c3列建立索引的表述是不同的，不同点如下：

- 建立 联合索引 只会建立如上图一样的1棵 B+ 树。
- 为c2和c3列建立索引会分别以 c2 和 c3 列的大小为排序规则建立2棵 B+ 树。

MyISAM中的索引方案

至此，我们介绍的都是 InnoDB 存储引擎中的索引方案，为了内容的完整性，以及各位可能在面试的时候遇到这类的问题，我们有必要再简单介绍一下 MyISAM 存储引擎中的索引方案。我们知道 InnoDB 中索引即数据，也就是聚簇索引的那棵 B+ 树的叶子节点中已经把所有完整的用户记录都包含了，而 MyISAM 的索引方案虽然也使用 B+ 树，但是却将索引和数据分开存储：

- 将表中的记录按照插入时间顺序的存储在一块存储空间上，我们可以通过行号而快速访问到一条记录（因为 index_demo 表的记录是定长的，所以可以使用行号来进行快

速访问，对于变长的记录MyISAM有不同的处理方案，我们这里就不介绍了），如图所示：

由于在插入数据的时候并没有刻意按照主键大小排序，所以我们并不能在这些数据上使用二分法进行查找。

- MyISAM 会单独为表的主键创建一个 B+ 树索引，只不过在 B+ 树的叶子节点中存储的不是完整的用户记录，而是 主键值 + 行号 的组合。也就是先通过索引找到对应的行号，再通过行号去找对应的记录！

  这一点和 InnoDB 是完全不相同的，在 InnoDB 存储引擎中，我们只需要根据主键值对 聚簇索引 进行一次查找能找到对应的记录，而在 MyISAM 中却需要进行一次 回表 操作，意味着 MyISAM 中建立的索引全部都是 二级索引 ！

- 如果有需要的话，我们也可以对其它的列分别建立索引或者建立联合索引，原理和 InnoDB 中的索引是一样的，只不过在叶子节点处存储的是 相应的列 + 行号 而

已。这些索引也全部都是 二级索引 。

MySQL中创建和删除索引的语句

光顾着唠叨索引的原理了，那我们如何使用 MySQL 语句去建立这种索引呢？ InnoDB 和 MyISAM 会自动为主键或者声明为 UNIQUE 的列去自动建立 B+ 树索引，但是如果我们想为其他的列建立索引就需要我们显式的去指明。为啥不自动为每个列都建立个索引呢？别忘了，每建立一个索引都会建立一棵 B+ 树，每插入一条记录都要维护各个记录、数据页的排序关系，这是很费性能和存储空间的。

我们可以在创建表的时候指定需要建立索引的单个列或者建立联合索引的多个列：

```
CREATE TALBE 表名 (
    各种列的信息 ··· ,
    [KEY|INDEX] 索引名 (需要被索引的单个列或多个列)
)
```

其中的 KEY 和 INDEX 是同义词，任意选用一个就可以。我们也可以在修改表结构的时候添加索引：

```
ALTER TABLE 表名 ADD [INDEX|KEY] 索引名 (需要被索引的单个列或多个列);
```

也可以在修改表结构的时候删除索引：

```
ALTER TABLE 表名 DROP [INDEX|KEY] 索引名;
```

比方说我们想在创建 index_demo 表的时候就为 c2 和 c3 列添加一个 联合索引 ，可以这么写建表语句：

```
CREATE TABLE index_demo(
    c1 INT,
    c2 INT,
    c3 CHAR(1),
    PRIMARY KEY(c1),
    INDEX idx_c2_c3 (c2, c3)
);
```

在这个建表语句中我们创建的索引名是 idx_c2_c3 ，这个名称可以随便起，不过我们还是建议以 idx_ 为前缀，后边跟着需要建立索引的列名，多个列名之间用下划线 _ 分隔开。

如果我们想删除这个索引，可以这么写：

```
ALTER TABLE index_demo DROP INDEX idx_c2_c3;
```

总结

1. 对于 InnoDB 存储引擎来说，在单个页中查找某条记录分为两种情况：

   - 以主键为搜索条件，可以使用 Page Directory 通过二分法快速定位相应的用户记录。
   - 以其他列为搜索条件，需要按照记录组成的单链表依次遍历各条记录。

2. 没有索引的情况下，不论是以主键还是其他列作为搜索条件，只能沿着页的双链表从左到右依次遍历各个页。

3. InnoDB 存储引擎的索引是一棵 B+ 树，完整的用户记录都存储在 B+ 树第 0 层的叶子节点，其他层次的节点都属于 内节点 ， 内节点 里存储的是 目录项记录 。 InnoDB 的索引分为两大种：

   - 聚簇索引

     以主键值的大小为页和记录的排序规则，在叶子节点处存储的记录包含了表中所有的列。

   - 二级索引

     以自定义的列的大小为页和记录的排序规则，在叶子节点处存储的记录内容是 列 + 主键 。

4. MyISAM 存储引擎的数据和索引分开存储，这种存储引擎的索引全部都是 二级索引 ，在叶子节点处存储的是 列 + 页号 。