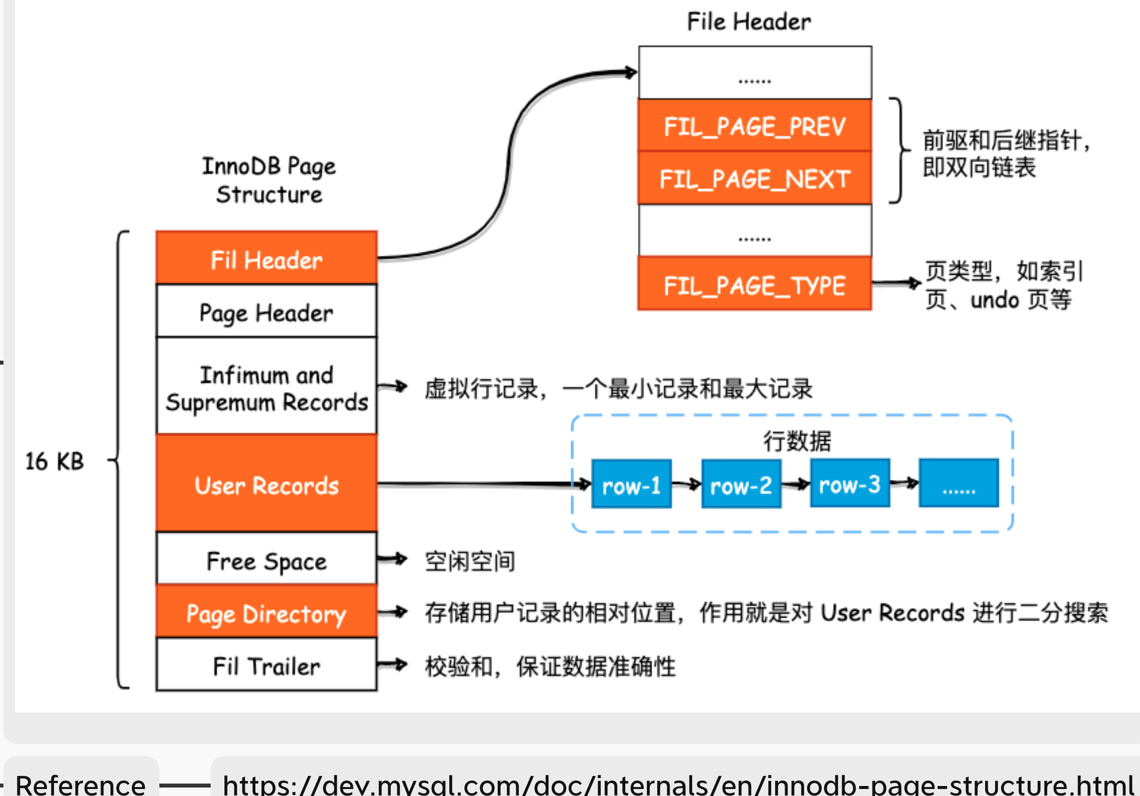
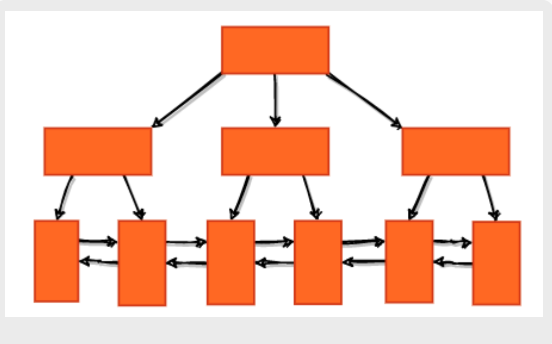
InnoDB Page

总览

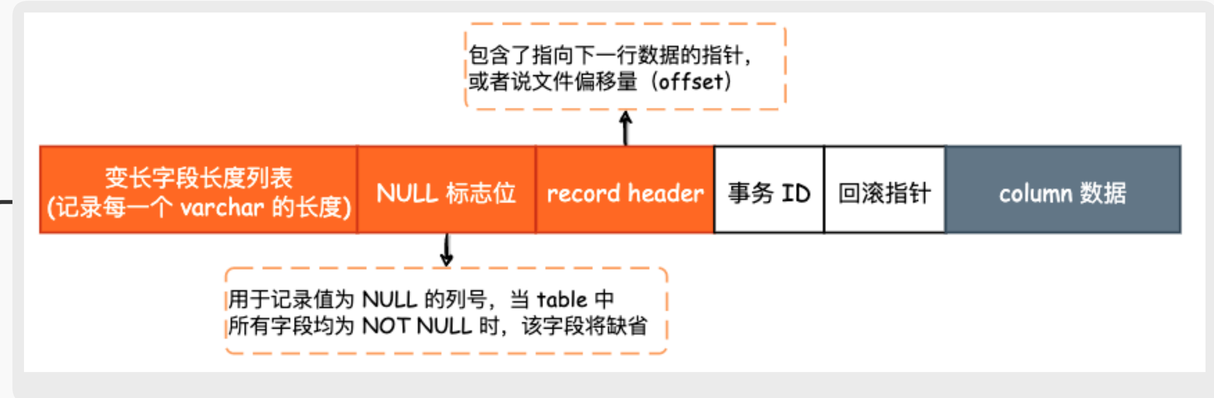


Reference — https://dev.mysql.com/doc/internals/en/innodb-page-structure.html

Fil Header

- 即 File Header，是对 InnoDB Page 的一个总体描述，包含了前驱指针、后继指针以及当前 Page 的类型等等
- **从 Fil Header 中内容就可以看出，InnoDB 聚簇索引的叶子节点使用双向链表链接的**
  - 
  - 也就是说，聚簇索引应该长这样儿。以后谁再跟你说 InnoDB 聚簇索引的叶子节点用单链表连接，就一搬砖儿呼死他
- 那么问题来了，为什么要用双向链表？单链表省空间不香吗？
  - 首先，Page 大部分的时间是待在硬盘里面儿的，只有在需要的时候才会被载入内存，并且，也只有少部分的 Page 会被载入。因此，使用单链表也不会节省多少内存，而至于硬盘空间，就更没有必要了
  - 其次，使用双向链表的一个重要原因就是使范围查询更快。例如 select * from Buz where id < 100 这条范围查询语句，当我们找到 id = 100 的叶子节点之后，通过前驱指针一直往前即可找到所有需要的数据，而无需再回到根节点进行查询

User Records

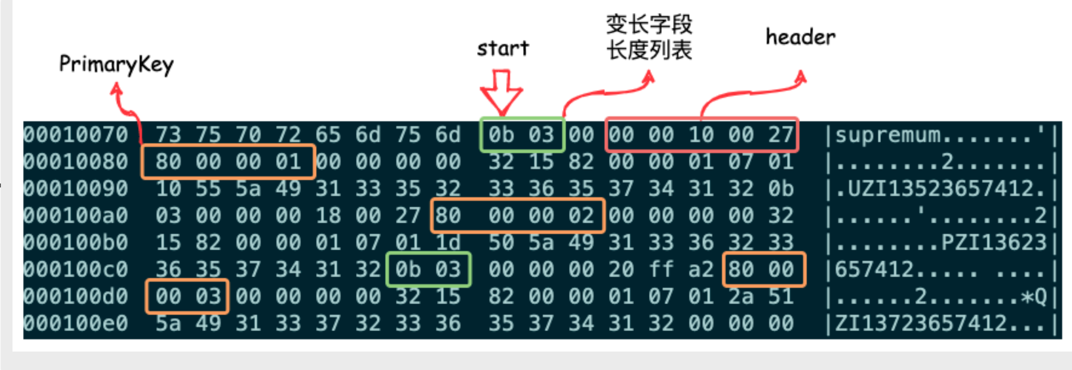
- InnoDB 规定，User Records 中至少要包含两条记录。如果允许只包含一条记录的话，那么 B+Tree 完全有可能变成链表
- 行记录格式
  - 对于不同版本的 InnoDB 引擎而言，行记录格式是不一样的
  - 不过在 5.7 版本之后，我们查看某一个 table 的 Row_format 的话，将会返回 Dynamic。简单的来说，Row_format 就是 InnoDB 如何确定其物理存储方式，也就是一行记录在硬盘和内存中的具体格式
- 

  - Compact 行记录格式，Dynamic 行格式和该格式类似
- 实验
  - 数据准备

```
CREATE SCHEMA `format` DEFAULT CHARACTER SET utf8mb4 ;

CREATE TABLE `format`.`user` (
  `id` INT NOT NULL,
  `name` VARCHAR(16),
  `mobile` VARCHAR(11),
  PRIMARY KEY (`id`));

INSERT INTO `format`.`user` (`id`, `name`, `mobile`) VALUES ('1', 'UZI', '13523657412');
INSERT INTO `format`.`user` (`id`, `name`, `mobile`) VALUES ('2', 'PZI', '13623657412');
INSERT INTO `format`.`user` (`id`, `name`, `mobile`) VALUES ('3', 'QZI', '13723657412');
```

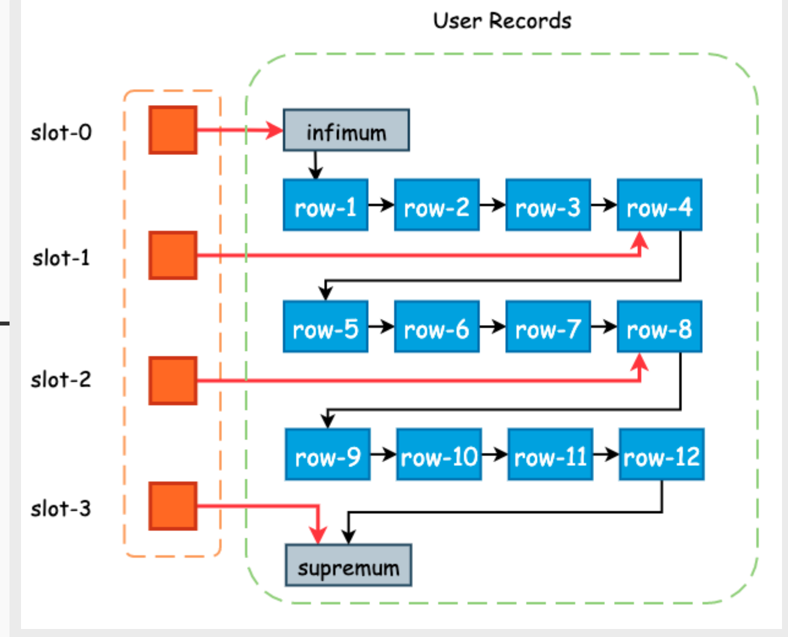
  - 查看 .ibd 文件
    - Linux — hexdump -C /var/lib/mysql/format/user.ibd
    - MacOS — hexdump -C /usr/local/var/mysql/format/user.ibd
    - 其实就是文件路径不一样
  - 

    - 数据从 10078 开始，我们从这里开始分析
  - 第一行数据

```
0b 03                     /* 变长字段长度列表, 逆序，第一列数据的长度为 3（"UZI"），第二列 mobile 的长度为 11 */
00                        /* NULL 标志位，第一行数据没有 NULL，所以为 0 */
00 00 10 00 27            /* header，0027 其实就是偏移量， 10078 + 0027 = 1009f，第二行数据就是从这里开始的*/
80 00 00 01               /* 主键 ID，实际值其实是 1 */
00 00 00 00 32 15         /* Transaction ID, 即事务 ID, 默认为 6 字节 */
82 00 00 01 07 01 10      /* 回滚指针, 默认为7字节 */
55 5a 49                  /* name 字段数据，为 "UZI" */
31 33 35 32 33 36 35 37 34 31 32   /* 长度为 11，也就是我们的 mobile 字段 */
```

  - 第二行数据
    - 具体的内容就不分析了，跳过第一行数据最后一个数据 "32"，下一个数据的起始位置为 1009f，正是我们上面通过偏移量相加得来的地址。也就是说，User Records 的确是通过单向链表连接的
- **因此，我们通过分析 .ibd 文件，可以确切的证明 User Records 中的用户数据是以单向链表排列的**

Page Directory

- 官方文档并没有对 Page Directory 做太多的说明，只是说明了这一块区域中包含了多个 "slot"，并且每个 "slot" 的大小是固定的，因此可以通过二分搜索的方式在 User Records 中找到具体的某一行数据
- 我们仍然可以通过分析 .ibd 文件来看 Page Directory 里面到底有什么，不过这次我们就不能再通过直接解读 .ibd 十六进制文件的方式来完成了，得借助 py_innodb_page_info 这个工具来分析
  - 看了一眼，还是用 python2 写的，并且对 mysql 8.0 的支持有问题，但是 5.7 测试是能够通过的
- 

  - 大概就是长这个样子，对 slot 进行二分搜索。初始化时 left = 0, right = 3，然后看 mid slot 所指向的记录和查找记录主键之间的大小关系
  - 因此，通过 Page Directory 所执行的二分搜索只能找到 record 在哪一个分组，然后在分组中顺着链表查找，所以严格上来讲算是"模糊的二分搜索"

记录检索

- 有了这些基础之后，我们就可以大致地给出 InnoDB 检索数据的方式了
- **① 首先从 B+Tree 的根节点开始，逐层检索，每一层的检索都需要一次逻辑 I/O，然后找到对应的叶子节点，也就是对应的数据页，将其载入内存**
- **② 根据 Page Directory 中的 slot 进行粗略的二分搜索，找到数据所在的记录分组，然后通过链表遍历的方式找到具体的那一条记录**