# 行格式

## 概述

## 使用

我们可以在创建或修改表的语句中指定行格式：

CREATE TABLE 表名 (列的信息) ROW\_FORMAT=行格式名称

ALTER TABLE 表名 ROW\_FORMAT=行格式名称

## 分类

一行记录可以以不同的格式存在InnoDB中，行格式分别是Compact、Redundant、Dynamic、Compressed行格式。

### COMPACT

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 变长字段长度列表 | NULL标志位 | 记录头信息 | 列1数据 | 列2数据 | … |

#### 记录的额外信息

这部分信息时服务器为了描述这条记录而不得不额外添加的一些信息，这些额外信息分为3类：

变长字段长度列表

NULL值列表

记录头信息

##### 变长字段长度列表

MySQL支持一些变长的数据类型，比如VARCHAR(M)、VARBINARY(M)、TEXT类型、BLOB类型，这些数据类型修饰列称为变长字段，变长字段中存储多少字节的数据不是固定的，所以我们在存储真实数据的时候需要顺便把这些数据占用的字节数也存起来。在Compact行格式中，把所有变长字段的真实数据占用的字节长度都存放在记录的开头部位，从而形成一个变长字段长度列表。

注：CHAR是一种固定长度的类型，VARCHAR是一种可变长度的类型。VARCHAR(M)，M代表最大能存储多少个字符（MySQL5.0.3以前是字节，以后就是字符）。

##### NULL值列表

Compact行格式会把可以为NULL的列统一管理起来，存一个标记位在NULL值列表中，如果表中没有允许存储NULL的列，则NULL值列表页不存在了。

二进制位的值为1时，代表该列的值为NULL。

二进制位的值为0时，代表该列的值不为NULL。

##### 记录头信息

除了变长字段长度列表、NULL值列表外，还有一个用于描述记录的记录头信息。它是由固定的5个字节组成，5个字节也就是40个二进制位，不同的位代表不同的意思，如图：

|  |  |  |
| --- | --- | --- |
| 名称 | 大小(单位bit) | 描述 |
| 预留位1 | 1 | 未使用 |
| 预留位2 | 1 | 未使用 |
| delete mask | 1 | 标记该记录是否被删除 |
| min\_rec\_mask | 1 | B+树的每层非叶子节点中的最小记录都会添加该标记 |
| n\_owned | 4 | 表示当前记录拥有的记录数 |
| heap\_no | 13 | 表示当前记录在记录堆的位置信息 |
| record\_type | 3 | 表示当前记录的类型：  0表示普通记录，  1表示B+树非叶子节点记录，  2表示最小记录，  3表示最大记录 |
| next\_record | 16 | 表示下一条记录的相对位置 |

#### 记录的真实数据

##### 隐藏列

记录的真实数据除了我们自己定义的列的数据以外，这会有三个隐藏列：

|  |  |  |  |
| --- | --- | --- | --- |
| 列名 | 是否需要 | 占用空间 | 描述 |
| row\_id | 否 | 6字节 | 行ID，唯一标识一条记录 |
| transaction\_id | 是 | 6字节 | 事务ID |
| roll\_pointer | 是 | 7字节 | 回滚指针 |

实际上这几列的真正名称其实是DB\_ROW\_ID、DB\_TRX\_ID、DB\_ROLL\_PTR。一个表没有手动定义主键，则会选取一个Unique键作为主键，如果连Unique键都没有的话，则会为表默认添加一个名为row\_id的隐藏列作为主键。所以row\_id是在没有自定义主键以及Unique键的情况下才会存在的。

##### 行溢出数据

VARCHAR(M)类型的列最多可以占用65535个字节，其中的M代表该类型最多存储的字符数量，如果我们使用ASCII字符集的话，一个字符就代表一个字节。

在CREATE TABLE的时候如创建的字段长度指定65535则提示Row size too large，是因为前面的额外信息占据3字节，如果创建65533则成功。

一个页大小一般是16KB即16384字节，而一个VARCHAR(M)类型的列最多可以存储65533个字节，这样就可能出现一个页存放不下一条记录。即这一行65533字节会占用多个页，这就是行溢出。

在跨多页的查询时，有两种方法：

1. 第一个页存储部分数据+下一页的地址，这样就可以实现多页的读取（其实就是B树底层实现）；
2. 第一页存储地址，快速定位到真正的数据地址，最后存储数据（B+树的底层实现）。

在Compact和Redundant行格式中，对于占用存储空间非常大的列，在记录的真实数据处只会存储该列的一部分数据，把剩下的数据分散存储在几个其他的页中，然后记录的真实数据处用20个字节存储指向这些页的地址（当然这20个字节中还包括这些分散在其他页面中的数据占用的字节数），从而可以找到剩余数据所在的页。

### Redundant

### Dynamic

Dynamic和Compressed行格式类似于Compact行格式，只不过在处理行溢出时有点分歧，它们不会在记录的真实数据处存储一部分数据，而是把所有的数据都存储到其他页面中（非聚集），只在记录的真实数据处存储其他页面的地址。另外，Compressed行格式会采用压缩算法对页面进行压缩。

### Compressed

# 数据页

## 概述

要了解页，需要了解局部性原理：从磁盘读取数据的时候，不是按照你需要的实际大小取数据，而是按照页为单位取数据（避免频繁的磁盘访问）。

页是InnoDB管理存储空间的基本单位，一个页的大小默认是16KB（操作系统默认是4KB，不要混淆）：

SHOW GLOBAL STATUS like ‘Innodb\_page\_size’;

## 结构



|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 中文名 | 占用空间 | 描述 |
| File Header | 文件头部 | 38字节 | 页的通用信息 |
| Page Header | 页面头部 | 56字节 | 数据页专有的信息 |
| Infimum+Supermum | 最小和最大记录 | 26字节 | 两个虚拟的行记录 |
| User Records | 用户记录 | 不确定 | 实际存储的行记录内容 |
| Free Space | 空闲空间 | 不确定 | 页中尚未使用的空间 |
| Page Directory | 页面目录 | 不确定 | 页中某些记录相对位置 |
| File Trailer | 文件尾部 | 8字节 | 校验页是否完整 |

### File Header

### Page Header

### Infimum+Supremum Records

### User Records

### Free Space

### Page Directory

### File Trailer

## 数据插入/查找



基本过程：

1. 首先第一条数据插入到用户数据区域；
2. 第二条数据插入前，首先进行排序（按照主键），然后确定插入位置（不一定第二条数据就在第一条数据后面）；
3. 以此类推；

注：这样的插入操作并不高，因为需要额外的排序，这样做的目的是提高查询的速度。

1. 查询的时候，先把数据所在的那一页数据全部从磁盘读取到内存，然后从记录中依次查找（但是极端情况下可能会遍历所有行数据的链表）；
2. 此时需要页目录进行快速定位（目录会用专门的算法对用户行数据进行分组），目录项中存储的是该组中主键（或索引）的最小值；
3. 这样查找数据的时候，先根据值的大小从目录项查找，然后具体定位到组，进而查找用户行数据；如果在对应组中没有找到，则不需要继续查找；
4. 如果数据非常大，则需要多个页：



1. 如果存储数据非常大，则页也非常多，页的链表很长，极端情况下数据查找可能会遍历整个页的链表，此时需要一个管理页的结构（每个结构存储所管理的页的最小值）：



上面叫目录页，主要存储的是目录的信息，下面的数据页，主要存储的是具体的数据。

数据页的next指针用于大于>操作，prev指针用于小于<操作（这与真正的B+树有所不同，严格的B+树只有next指针）。

# 索引