二、InnoDB记录存储结构

InnoDB记录存储结构

标签： MySQL 是怎样运行的

准备工作

到现在为止， `MySQL` 对于我们来说还是一个黑盒，我们只负责使用客户端发送请求并等待服务器返回结果，表中的数据到底存到了哪里？以什么格式存放的？ `MySQL` 是以什么方式来访问的这些数据？这些问题我们统统不知道，对于未知领域的探索向来就是社会主义核心价值观中的一部分，作为新一代社会主义接班人，不把它们搞懂怎么支援祖国建设呢？

我们前边唠叨请求处理过程的时候提到过， `MySQL` 服务器上负责对表中数据的读取和写入工作的部分是 `存储引擎` ，而服务器又支持不同类型的存储引擎，比如 `InnoDB` 、 `MyISAM` 、 `Memory` 啥的，不同的存储引擎一般是由不同的人为实现不同的特性而开发的，真实数据在不同存储引擎中存放的格式一般是不同的，甚至有的存储引擎比如 `Memory` 都不用磁盘来存储数据，也就是说关闭服务器后表中的数据就消失了。由于 `InnoDB` 是 `MySQL` 默认的存储引擎，也是我们最常用到的存储引擎，我们也没有那么多时间去把各个存储引擎的内部实现都看一遍，所以本集要唠叨的是使用 `InnoDB` 作为存储引擎的数据存储结构，了解了一个存储引擎的数据存储结构之后，其他的存储引擎都是依葫芦画瓢，等我们用到了再说哈～

InnoDB页简介

`InnoDB` 是一个将表中的数据存储到磁盘上的存储引擎，所以即使关机后重启我们的数据还是存在的。而真正处理数据的过程是发生在内存中的，所以需要把磁盘中的数据加载到内存中，如果是处理写入或修改请求的话，还需要把内存中的内容刷新到磁盘上。而我们知道读写磁盘的速度非常慢，和内存读写差了几个数量级，所以当我们想从表中获取某些记录时， `InnoDB` 存储引擎需要一条一条的把记录从磁盘上读出来么？不，那样会慢死， `InnoDB` 采取的方式是：将数据划分为若干个页，以页作为磁盘和内存之间交互的基本单位，InnoDB中页的大小一般为 *16* KB。也就是在一般情况下，一次最少从磁盘中读取16KB的内容到内存中，一次最少把内存中的16KB内容刷新到磁盘中。

InnoDB行格式

我们平时是以记录为单位来向表中插入数据的，这些记录在磁盘上的存放方式也被称为 `行格式` 或者 `记录格式` 。设计 `InnoDB` 存储引擎的大叔们到现在为止设计了4种不同类型的 `行格式` ，分别是 `Compact` 、 `Redundant` 、 `Dynamic` 和 `Compressed` 行格式，随着时间的推移，他们可能会设计出更多的行格式，但是不管怎么变，在原理上大体都是相同的。

指定行格式的语法

我们可以在创建或修改表的语句中指定 `行格式` ：

Plain Text 复制代码

```
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
```

比如我们在 `xiaohaizi` 数据库里创建一个演示用的表 `record_format_demo` ，可以这样指定它的 `行格式` ：

Plain Text 复制代码

```
mysql> USE xiaohaizi;
Database changed

mysql> CREATE TABLE record_format_demo (
    ->     c1 VARCHAR(10),
    ->     c2 VARCHAR(10) NOT NULL,
    ->     c3 CHAR(10),
    ->     c4 VARCHAR(10)
    -> ) CHARSET=ascii ROW_FORMAT=COMPACT;
Query OK, 0 rows affected (0.03 sec)
```

可以看到我们刚刚创建的这个表的 `行格式` 就是 `Compact` ，另外，我们还显式指定了这个表的字符集为 `ascii` ，因为 `ascii` 字符集只包括空格、标点符号、数字、大小写字母和一些不可见字符，所以我们的汉字是不能存到这个表里的。我们现在向这个表中插入两条记录：

```
mysql> INSERT INTO record_format_demo(c1, c2, c3, c4) VALUES('aaaa', 'bbb', 'cc', 'd'), ('eeee', 'fff', NULL, NULL);
Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

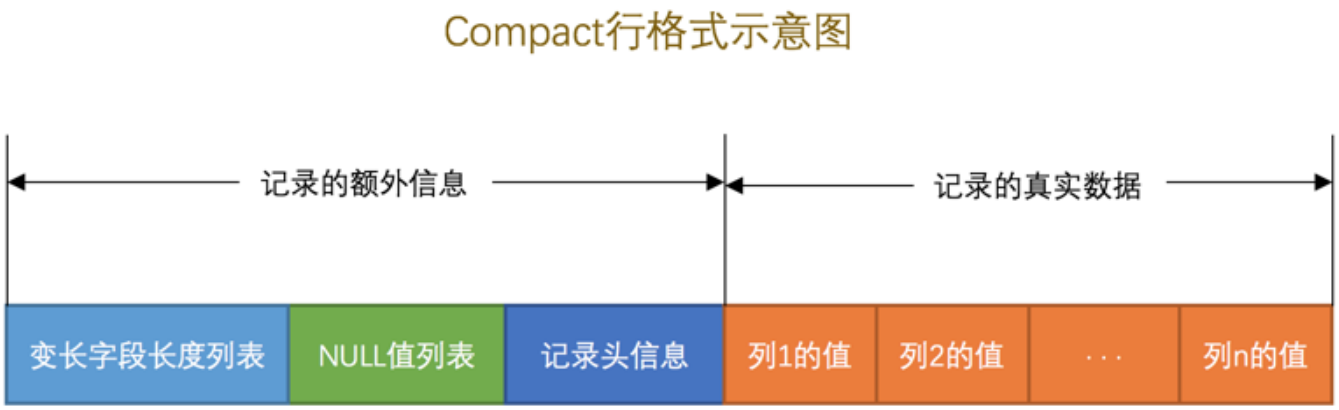
现在表中的记录就是这个样子的：

```
mysql> SELECT * FROM record_format_demo;
+------+-----+------+------+
| c1   | c2  | c3   | c4   |
+------+-----+------+------+
| aaaa | bbb | cc   | d    |
| eeee | fff | NULL | NULL |
+------+-----+------+------+
2 rows in set (0.00 sec)

mysql>
```

演示表的内容也填充好了，现在我们就来看看各个行格式下的存储方式到底有啥不同吧～

COMPACT行格式

废话不多说，直接看图：



大家从图中可以看出来，一条完整的记录其实可以被分为 `记录的额外信息` 和 `记录的真实数据` 两大部分，下边我们详细看一下这两部分的组成。

记录的额外信息

这部分信息是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，分别是 `变长字段长度列表` 、 `NULL值列表` 和 `记录头信息` ，我们分别看一下。

变长字段长度列表

我们知道 `MySQL` 支持一些变长的数据类型，比如 `VARCHAR(M)` 、 `VARBINARY(M)` 、各种 `TEXT` 类型，各种 `BLOB` 类型，我们也可以把拥有这些数据类型的列称为 `变长字段` ，变长字段中存储多少字节的数据是不固定的，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，这样才不至于把 `MySQL` 服务器搞懵，所以这些变长字段占用的存储空间分为两部分：

1. **真正的数据内容**
2. **占用的字节数**

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，各变长字段数据占用的字节数按照列的顺序逆序存放，我们再次强调一遍，是逆序存放！

我们拿 `record_format_demo` 表中的第一条记录来举个例子。因为 `record_format_demo` 表的 `c1` 、 `c2` 、 `c4` 列都是 `VARCHAR(10)` 类型的，也就是变长的数据类型，所以这三个列的值的长度都需要保存在记录开头处，因为 `record_format_demo` 表中的各个列都使用的是 `ascii` 字符集，所以每个字符只需要1个字节来进行编码，来看一下第一条记录各变长字段内容的长度：

列名	存储内容	内容长度（十进制表示）	内容长度（十六进制表示）
`c1`	`'aaaa'`	`4`	`0x04`
`c2`	`'bbb'`	`3`	`0x03`
`c4`	`'d'`	`1`	`0x01`

又因为这些长度值需要按照列的逆序存放，所以最后 `变长字段长度列表` 的字节串用十六进制表示的效果就是（各个字节之间实际上没有空格，用空格隔开只是方便理解）：

```
Plain Text | 复制代码
1   01 03 04
```

把这个字节串组成的 `变长字段长度列表` 填入上边的示意图中的效果就是：

由于第一行记录中 `c1` 、 `c2` 、 `c4` 列中的字符串都比较短，也就是说内容占用的字节数比较小，用1个字节就可以表示，但是如果变长列的内容占用的字节数比较多，可能就需要用2个字节来表示。具体用1个还是2个字节来表示真实数据占用的字节数， `InnoDB` 有它的一套规则，我们首先声明一下 `W` 、 `M` 和 `L` 的意思：

1. 假设某个字符集中表示一个字符最多需要使用的字节数为 `W` ，也就是使用 `SHOW CHARSET` 语句的结果中的 `Maxlen` 列，比方说 `utf8` 字符集中的 `W` 就是 `3` ， `gbk` 字符集中的 `W` 就是 `2` ， `ascii` 字符集中的 `W` 就是 `1` 。
2. 对于变长类型 `VARCHAR(M)` 来说，这种类型表示能存储最多 `M` 个字符（注意是字符不是字节），所以这个类型能表示的字符串最多占用的字节数就是 `M×W` 。
3. 假设它实际存储的字符串占用的字节数是 `L` 。

所以确定使用1个字节还是2个字节表示真正字符串占用的字节数的规则就是这样：

- 如果 `M×W <= 255` ，那么使用1个字节来表示真正字符串占用的字节数。

```
Plain Text | 复制代码
1   也就是说InnoDB在读记录的变长字段长度列表时先查看表结构，如果某个变长字段允许存储的最大字节数不大于255时，可以认为只使用1个字节来表示真正字符串占用的字节数。
```

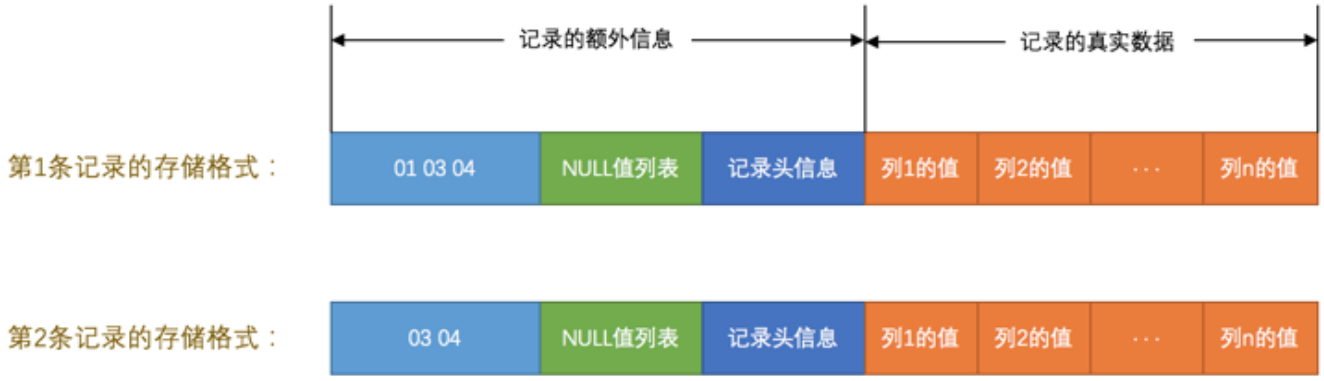
- 如果 `M×W > 255` ，则分为两种情况：

- 如果 `L <= 127` ，则用1个字节来表示真正字符串占用的字节数。
- 如果 `L > 127` ，则用2个字节来表示真正字符串占用的字节数。

```
InnoDB在读记录的变长字段长度列表时先查看表结构，如果某个变长字段允许存储的最大字节数大于255时，该怎么区分它正在读的某个字节是一个单独的字段长度还是半个字段长度呢？设计InnoDB的大叔使用该字节的第一个二进制位作为标志位：如果该字节的第一个位为0，那该字节就是一个单独的字段长度（使用一个字节表示不大于127的二进制的第一个位都为0），如果该字节的第一个位为1，那该字节就是半个字段长度。

对于一些占用字节数非常多的字段，比方说某个字段长度大于了16KB，那么如果该记录在单个页面中无法存储时，InnoDB会把一部分数据存放到所谓的溢出页中（我们后边会唠叨），在变长字段长度列表处只存储留在本页面中的长度，所以使用两个字节也可以存放下来。
```

总结一下就是说：如果该可变字段允许存储的最大字节数（ `M×W` ）超过255字节并且真实存储的字节数（ `L` ）超过127字节，则使用2个字节，否则使用1个字节。

另外需要注意的一点是，变长字段长度列表中只存储值为 ***非NULL*** 的列内容占用的长度，值为 ***NULL*** 的列的长度是不储存的 。也就是说对于第二条记录来说，因为 `c4` 列的值为 `NULL` ，所以第二条记录的 `变长字段长度列表` 只需要存储 `c1` 和 `c2` 列的长度即可。其中 `c1` 列存储的值为 `'eeee'` ，占用的字节数为 `4` ， `c2` 列存储的值为 `'fff'` ，占用的字节数为 `3` ，所以 `变长字段长度列表` 需2个字节。填充完 `变长字段长度列表` 的两条记录的对比图如下：



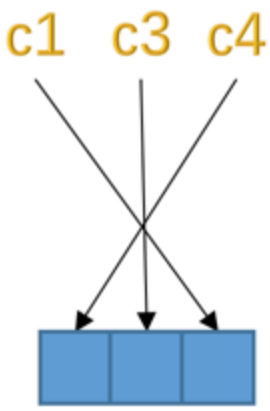

```
小贴士：

并不是所有记录都有这个 变长字段长度列表 部分，比方说表中所有的列都不是变长的数据类型的话，这一部分就不需要有。
```

NULL值列表

我们知道表中的某些列可能存储 `NULL` 值，如果把这些 `NULL` 值都放到 `记录的真实数据` 中存储会很占地方，所以 `Compact` 行格式把这些值为 `NULL` 的列统一管理起来，存储到 `NULL` 值列表中，它的处理过程是这样的：

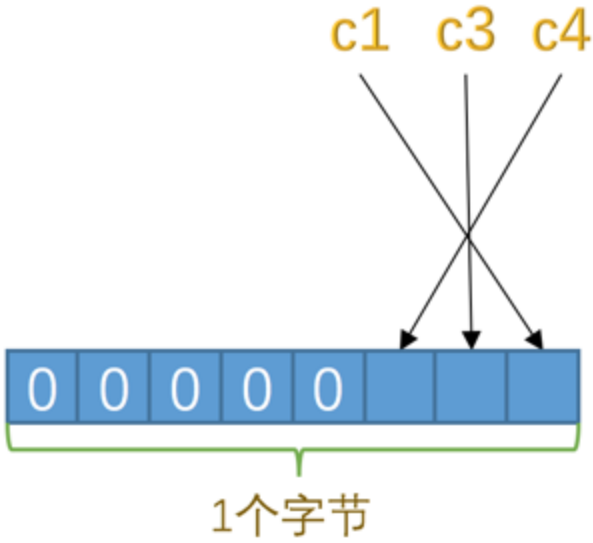
1. 首先统计表中允许存储 `NULL` 的列有哪些。
   我们前边说过，主键列、被 `NOT NULL` 修饰的列都是不可以存储 `NULL` 值的，所以在统计的时候不会把这些列算进去。比方说表 `record_format_demo` 的3个列 `c1` 、 `c3` 、 `c4` 都是允许存储 `NULL` 值的，而 `c2` 列是被 `NOT NULL` 修饰，不允许存储 `NULL` 值。
2. 如果表中没有允许存储 ***NULL*** 的列，则 *NULL值列表* 也不存在了，否则将每个允许存储 `NULL` 的列对应一个二进制位，二进制位按照列的顺序逆序排列，二进制位表示的意义如下：
   - 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` 。

因为表 `record_format_demo` 有3个值允许为 `NULL` 的列，所以这3个列和二进制位的对应关系就是这样：



再一次强调，二进制位按照列的顺序逆序排列，所以第一个列 `c1` 和最后一个二进制位对应。

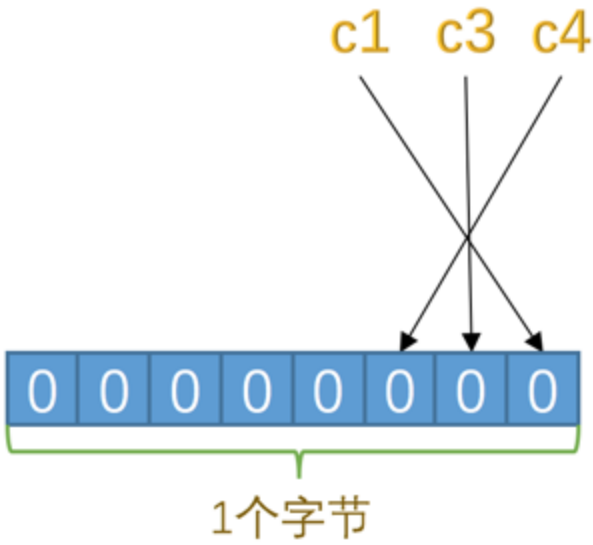
3. `MySQL` 规定 `NULL值列表` 必须用整数个字节的位表示，如果使用的二进制位个数不是整数个字节，则在字节的高位补 `0` 。
   表 `record_format_demo` 只有3个值允许为 `NULL` 的列，对应3个二进制位，不足一个字节，所以在字节的高位补 `0` ，效果就是这样：



以此类推，如果一个表中有9个允许为 `NULL` ，那这个记录的 `NULL` 值列表部分就需要2个字节来表示了。

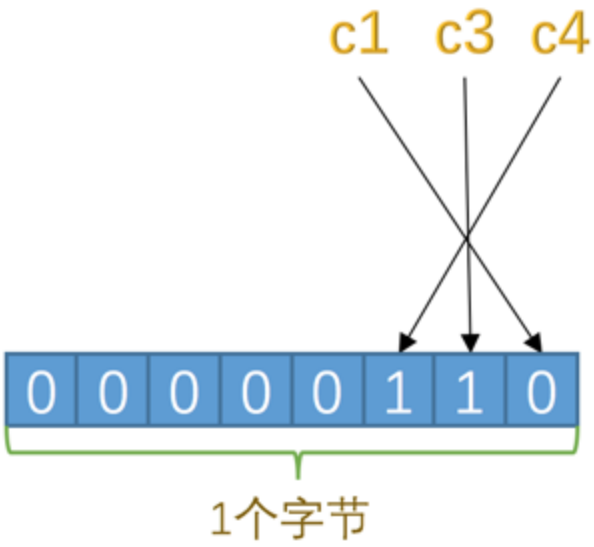
知道了规则之后，我们再返回头看表 `record_format_demo` 中的两条记录中的 `NULL值列表` 应该怎么储存。因为只有 `c1` 、 `c3` 、 `c4` 这3个列允许存储 `NULL` 值，所以所有记录的 `NULL值列表` 只需要一个字节。

- 对于第一条记录来说， `c1` 、 `c3` 、 `c4` 这3个列的值都不为 `NULL` ，所以它们对应的二进制位都是 `0` ，画个图就是这样：



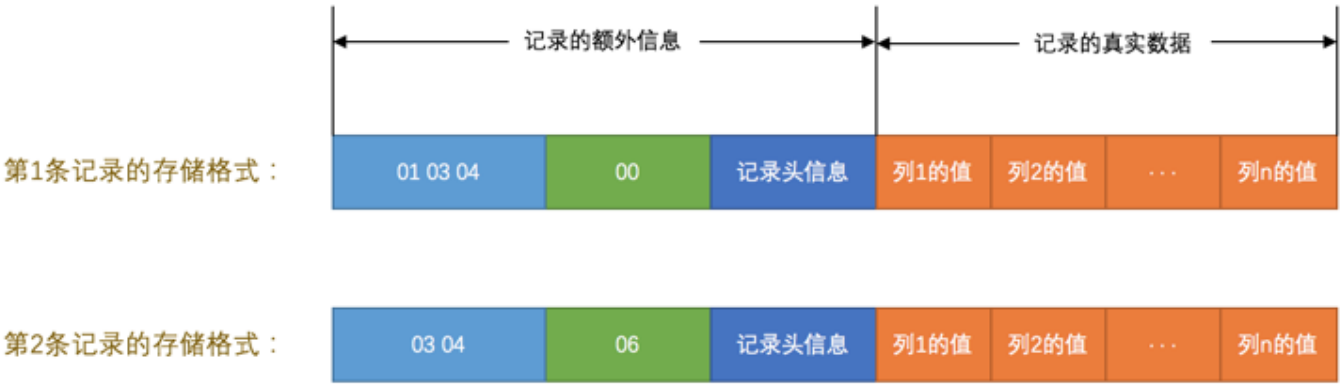
所以第一条记录的 `NULL值列表` 用十六进制表示就是： `0x00` 。

- 对于第二条记录来说， `c1` 、 `c3` 、 `c4` 这3个列中 `c3` 和 `c4` 的值都为 `NULL` ，所以这3个列对应的二进制位的情况就是：



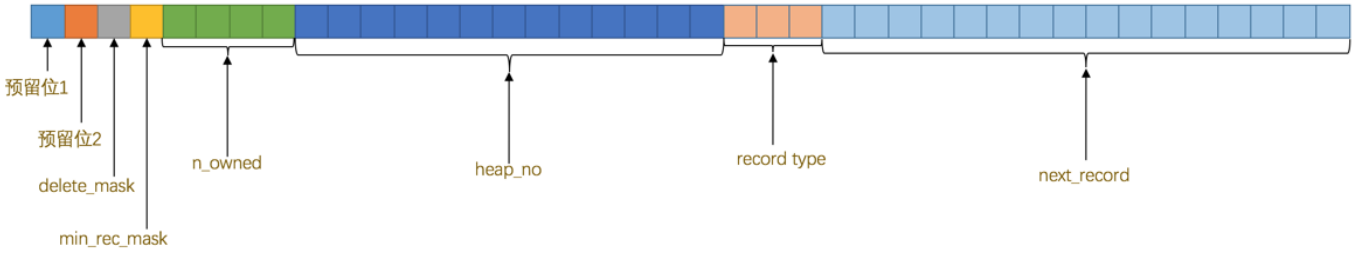
所以第二条记录的 `NULL值列表` 用十六进制表示就是： `0x06` 。

所以这两条记录在填充了 `NULL值列表` 后的示意图就是这样：



记录头信息

除了 `变长字段长度列表` 、 `NULL值列表` 之外，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：

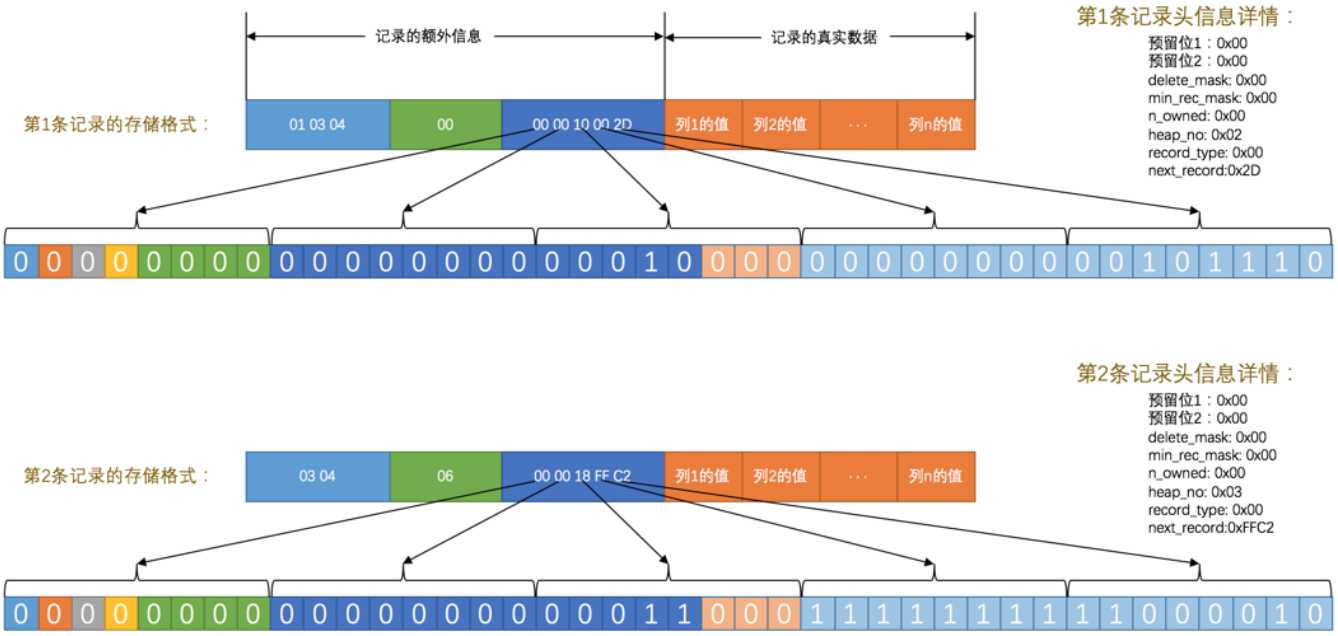


这些二进制位代表的详细信息如下表：

名称	大小（单位：bit）	描述
预留位1	1	没有使用
预留位2	1	没有使用
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

大家不要被这么多的属性和陌生的概念给吓着，我这里只是为了内容的完整性把这些位代表的意思都写了出来，现在没必要把它们的意思都记住，记住也没啥用，现在只需要看一遍混个脸熟，等之后用到这些属性的时候我们再回过头来看。

因为我们并不清楚这些属性详细的用法，所以这里就不分析各个属性值是怎么产生的了，之后我们遇到会详细看的。所以我们现在直接看一下 `record_format_demo` 中的两条记录的 `头信息` 分别是什么：




```
小贴士：再一次强调，大家如果看不懂记录头信息里各个位代表的概念千万别纠结，我们后边会说的～
```

记录的真实数据

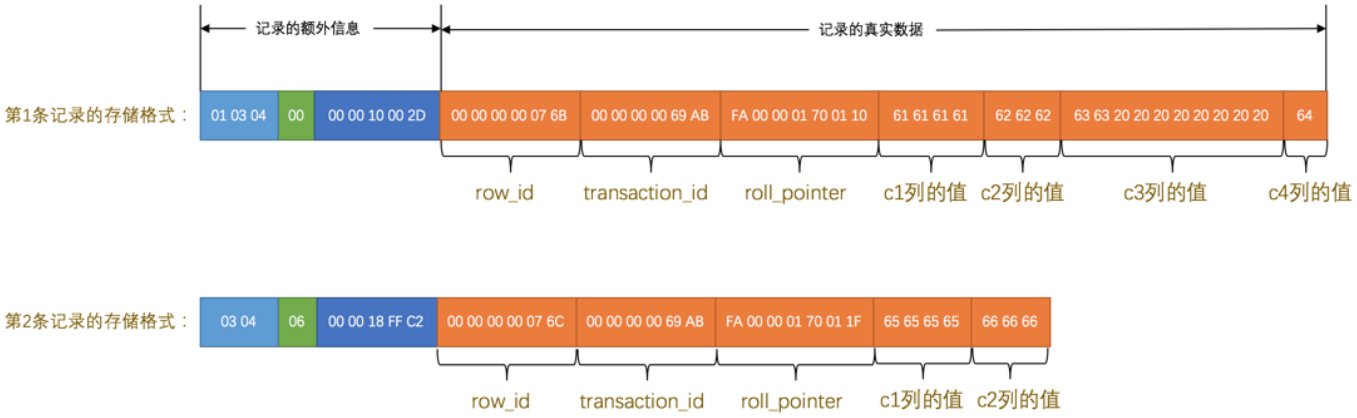
对于 `record_format_demo` 表来说， `记录的真实数据` 除了 `c1` 、 `c2` 、 `c3` 、 `c4` 这几个我们自己定义的列的数据以外， `MySQL` 会为每个记录默认的添加一些列（也称为 `隐藏列` ），具体的列如下：

列名	是否必须	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

```
1  小贴士：实际上这几个列的真正名称其实是：DB_ROW_ID、DB_TRX_ID、DB_ROLL_PTR，我们为了美观才写成了row_id、transaction_id和roll_pointer。
```

这里需要提一下 `InnoDB` 表对主键的生成策略：优先使用用户自定义主键作为主键，如果用户没有定义主键，则选取一个 `Unique` 键作为主键，如果表中连 `Unique` 键都没有定义的话，则 `InnoDB` 会为表默认添加一个名为 `row_id` 的隐藏列作为主键。所以我们从上表中可以看出：InnoDB存储引擎会为每条记录都添加 *transaction_id* 和 *roll_pointer* 这两个列，但是 *row_id* 是可选的（在没有自定义主键以及Unique键的情况下才会添加该列）。这些隐藏列的值不用我们操心， `InnoDB` 存储引擎会自己帮我们生成的。

因为表 `record_format_demo` 并没有定义主键，所以 `MySQL` 服务器会为每条记录增加上述的3个列。现在看一下加上 `记录的真实数据` 的两个记录长什么样吧：

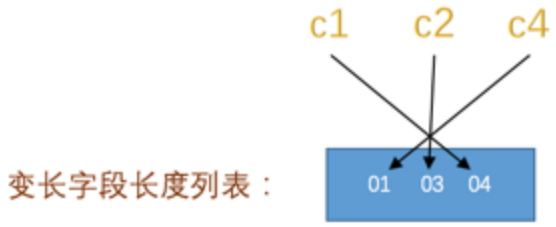


看这个图的时候我们需要注意几点：

1. 表 `record_format_demo` 使用的是 `ascii` 字符集，所以 `0x61616161` 就表示字符串 `'aaaa'` ， `0x626262` 就表示字符串 `'bbb'` ，以此类推。
2. 注意第1条记录中 `c3` 列的值，它是 `CHAR(10)` 类型的，它实际存储的字符串是： `'cc'` ，而 `ascii` 字符集中的字节表示是 `'0x6363'` ，虽然表示这个字符串只占用了2个字节，但整个 `c3` 列仍然占用了10个字节的空间，除真实数据以外的8个字节的统统都用空格字符填充，空格字符在 `ascii` 字符集的表示就是 `0x20` 。
3. 注意第2条记录中 `c3` 和 `c4` 列的值都为 `NULL` ，它们被存储在了前边的 `NULL值列表` 处，在记录的真实数据处就不再冗余存储，从而节省存储空间。

CHAR(M)列的存储格式

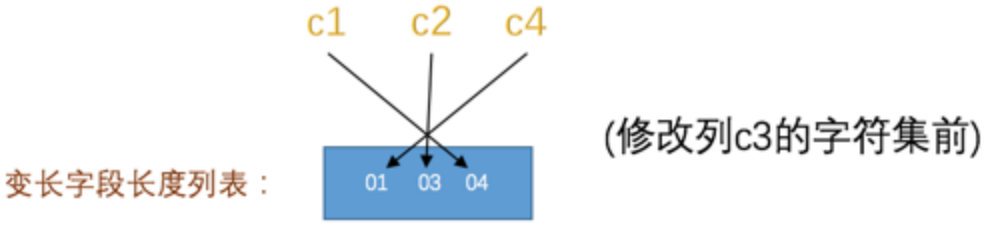
`record_format_demo` 表的 `c1` 、 `c2` 、 `c4` 列的类型是 `VARCHAR(10)` ，而 `c3` 列的类型是 `CHAR(10)` ，我们说在 `Compact` 行格式下只会把变长类型的列的长度逆序存到 `变长字段长度列表` 中，就像这样：

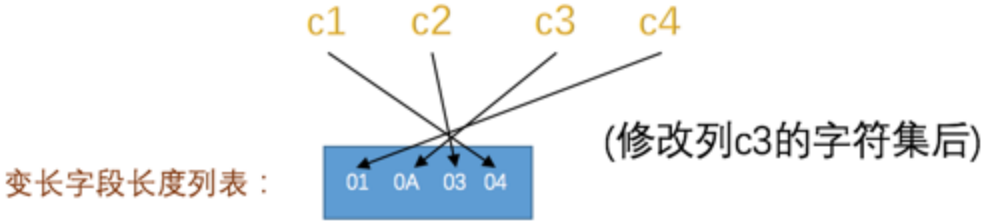


但是这只是因为我们的 `record_format_demo` 表采用的是 `ascii` 字符集，这个字符集是一个定长字符集，也就是说表示一个字符采用固定的一个字节，如果采用变长的字符集（也就是表示一个字符需要的字节数不确定，比如 `gbk` 表示一个字符要1~2个字节、`utf8`表示一个字符要1~3个字节等）的话， `c3` 列的长度也会被存储到 `变长字段长度列表` 中，比如我们修改一下 `record_format_demo` 表的字符集：

```
mysql> ALTER TABLE record_format_demo MODIFY COLUMN c3 CHAR(10) CHARACTER SET utf8;Query OK, 2 rows affected (0.02 sec)Records: 2  Duplicates: 0  Warnings: 0
```

修改该列字符集后记录的 `变长字段长度列表` 也发生了变化，如图：





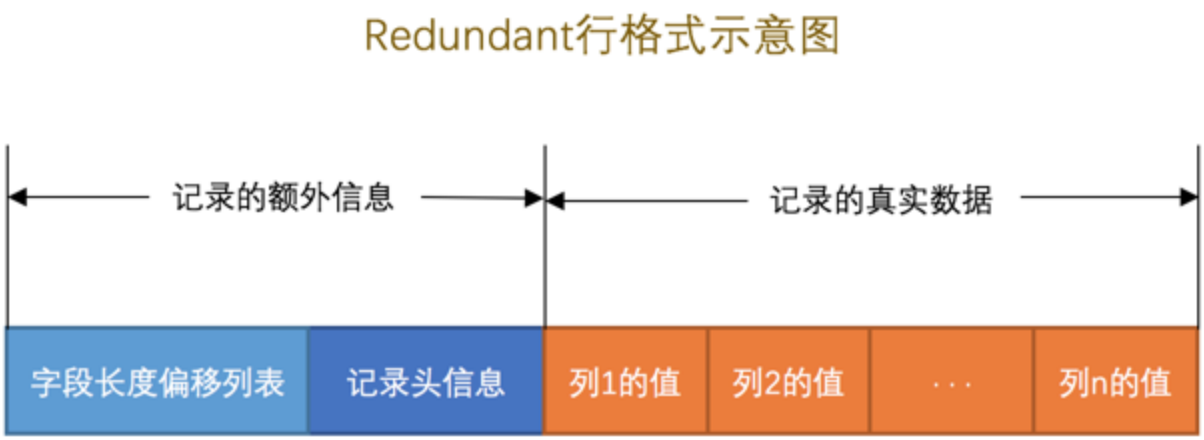
这就意味着：对于 *CHAR(M)* 类型的列来说，当列采用的是定长字符集时，该列占用的字节数不会被加到变长字段长度列表，而如果采用变长字符集时，该列占用的字节数也会被加到变长字段长度列表。

另外有一点还需要注意，变长字符集的 `CHAR(M)` 类型的列要求至少占用 `M` 个字节，而 `VARCHAR(M)` 却没有这个要求。比方说对于使用 `utf8` 字符集的 `CHAR(10)` 的列来说，该列存储的数据字节长度的范围是10～30个字节。即使我们向该列中存储一个空字符串也会占用 `10` 个字节，这是怕将来更新该列的值的字节长度大于原有值的字节长度而小于10个字节时，可以在该记录处直接更新，而不是在存储空间中重新分配一个新的记录空间，导致原有的记录空间称为所谓的碎片。（这里你感受到设计 `Compact` 行格式的大叔既想节省存储空间，又不想更新 `CHAR(M)` 类型的列产生碎片时的纠结心情了吧。）

Redundant行格式

其实知道了 `Compact` 行格式之后，其他的行格式就是依葫芦画瓢了。我们现在要介绍的 `Redundant` 行格式是 `MySQL5.0` 之前用的一种行格式，也就是说它已经非常老了，但是本着知识完整性的角度还是要提一下，大家乐呵乐呵的看就好。

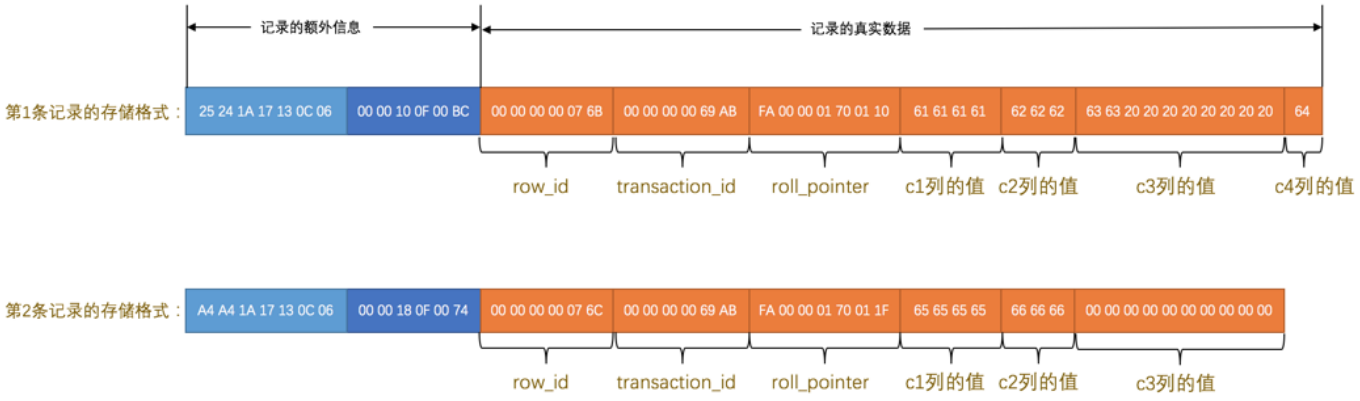
画个图展示一下 `Redundant` 行格式的全貌：



现在我们把表 `record_format_demo` 的行格式修改为 `Redundant` ：

```
mysql> ALTER TABLE record_format_demo ROW_FORMAT=Redundant;Query OK, 0 rows affected (0.05 sec)Records: 0  Duplicates: 0  Warnings: 0
```

为了方便大家理解和节省篇幅，我们直接把表 `record_format_demo` 在 `Redundant` 行格式下的两条记录的真实存储数据提供出来，之后我们着重分析两种行格式的不同即可。



下边我们从各个方面看一下 `Redundant` 行格式有什么不同的地方：

- 字段长度偏移列表
  注意 `Compact` 行格式的开头是 `变长字段长度列表` ，而 `Redundant` 行格式的开头是 `字段长度`

偏移列表，与 变长字段长度列表 有两处不同：

- 没有了变长两个字，意味着 Redundant 行格式会把该条记录中所有列（包括 隐藏列 ）的长度信息都按照逆序存储到 字段长度偏移列表 。
- 多了个偏移两个字，这意味着计算列值长度的方式不像 Compact 行格式那么直观，它是采用两个相邻数值的差值来计算各个列值的长度。
  比如第一条记录的 字段长度偏移列表 就是：
  因为它是逆序排放的，所以按照列的顺序排列就是：
  按照两个相邻数值的差值来计算各个列值的长度的意思就是：

```
Plain Text | 复制代码
25 24 1A 17 13 0C 06
```

```
Plain Text | 复制代码
06 0C 13 17 1A 24 25
```

```
Plain Text | 复制代码
第一列(`row_id`)的长度就是 0x06个字节，也就是6个字节。第二列(`transaction_id`)的长度就是 (0x0C - 0x06)个字节，也就是6个字节。第三列(`roll_pointer`)的长度就是 (0x13 - 0x0C)个字节，也就是7个字节。第四列(`c1`)的长度就是 (0x17 - 0x13)个字节，也就是4个字节。第五列(`c2`)的长度就是 (0x1A - 0x17)个字节，也就是3个字节。第六列(`c3`)的长度就是 (0x24 - 0x1A)个字节，也就是10个字节。第七列(`c4`)的长度就是 (0x25 - 0x24)个字节，也就是1个字节。
```

- 记录头信息
  Redundant 行格式的记录头信息占用 6 字节， 48 个二进制位，这些二进制位代表的意思如下：
  第一条记录中的头信息是：
  根据这六个字节可以计算出各个属性的值，如下：
  与 Compact 行格式的记录头信息对比来看，有两处不同：

名称	大小（单位：bit）	描述
预留位1	1	没有使用
预留位2	1	没有使用
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	表示当前记录在页面堆的位置信息
n_field	10	表示记录中列的数量
1byte_offs_flag	1	标记字段长度偏移列表中的偏移量是使用1字节还是2字节表示的
next_record	16	表示下一条记录的相对位置

```
00 00 10 0F 00 BC
```

```
预留位1：0x00预留位2：0x00delete_mask: 0x00min_rec_mask: 0x00n_owned: 0x00heap_no: 0x02n_field: 0x071byte_offs_flag: 0x01next_record:0xBC
```

- `Redundant` 行格式多了 `n_field` 和 `1byte_offs_flag` 这两个属性。
- `Redundant` 行格式没有 `record_type` 这个属性。

- `Redundant` 行格式中 `NULL` 值的处理

  因为 `Redundant` 行格式并没有 `NULL值列表` ，所以需要别的方式来存储字段的 `NULL` 值，具体策略如下：

  - 如果该存储 `NULL` 值的字段是变长数据类型的，则在字段长度偏移列表中记录即可，并不占用记录的真实数据部分。

    比如 `record_format_demo` 表的 `c4` 列是 `VARCHAR(10)` 类型的，而第二条记录的 `c4` 列存储的是 `NULL` 值，我们回过头看一下第二条记录的 `字段长度偏移列表` 如下：

    按照列的顺序排放就是：

    可以看到第二条记录的 `c4` 列的偏移长度和 `c3` 列的相同都是 `A4` ，意味着 `c4` 列的长度为0，也就意味着存储的是 `NULL` 值。

```
A4 A4 1A 17 13 0C 06
```

```
06 0C 13 17 1A A4 A4
```

  - 如果该存储 `NULL` 值的字段是 `CHAR(M)` 数据类型的，则将占用记录的真实数据部分，并把

该字段对应的数据使用 `0x00` 字节填充。
如图第二条记录的 `c3` 列的值是 `NULL` ，而 `c3` 列的类型是 `CHAR(10)` ，占用记录的真实数据部分10字节，所以我们看到在 `Redundant` 行格式中使用 `0x00000000000000000000` 来表示 `NULL` 值。

除了以上的几点之外， `Redundant` 行格式和 `Compact` 行格式还是大致相同的。

CHAR(M)列的存储格式

我们知道 `Compact` 行格式在 `CHAR(M)` 类型的列中存储数据的时候还挺麻烦，分变长字符集和定长字符集的情况，而在 `Redundant` 行格式中十分干脆，不管该列使用的字符集是啥，只要是使用 `CHAR(M)` 类型，占用的真实数据空间就是该字符集表示一个字符最多需要的字节数和 `M` 的乘积。比方说使用 `utf8` 字符集的 `CHAM(10)` 类型的列占用的真实数据空间始终为 `30` 个字节，使用 `gbk` 字符集的 `CHAM(10)` 类型的列占用的真实数据空间始终为 `20` 个字节。由此可以看出来，使用 `Redundant` 行格式的 `CHAR(M)` 类型的列是不会产生碎片的。

行溢出数据

VARCHAR(M)最多能存储的数据

我们知道对于 `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，我们看看 `VARCHAR(65535)` 是否可用：

Plain Text | 复制代码

```
mysql> CREATE TABLE varchar_size_demo(    ->     c VARCHAR(65535)    -> ) CHARSET=ascii ROW_FORMAT=Compact;ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not counting BLOBs, is 65535. This includes storage overhead, check the manual. You have to change some columns to TEXT or BLOBsmysql>
```

从报错信息里可以看出， `MySQL` 对一条记录占用的最大存储空间是有限制的，除了 `BLOB` 或者 `TEXT` 类型的列之外，其他所有的列（不包括隐藏列和记录头信息）占用的字节长度加起来不能超过 `65535` 个字节。所以 `MySQL` 服务器建议我们把存储类型改为 `TEXT` 或者 `BLOB` 的类型。这个 `65535` 个

字节除了列本身的数据之外，还包括一些其他的数据（`storage overhead`），比如说我们为了存储一个`VARCHAR(M)`类型的列，其实需要占用3部分存储空间：

- 真实数据
- 真实数据占用字节的长度
- `NULL`值标识，如果该列有`NOT NULL`属性则可以没有这部分存储空间

如果该`VARCHAR`类型的列没有`NOT NULL`属性，那最多只能存储`65532`个字节的数据，因为真实数据的长度可能占用2个字节，`NULL`值标识需要占用1个字节：

Plain Text 复制代码

```
1  mysql> CREATE TABLE varchar_size_demo(    ->      c VARCHAR(65532)    -> )
   CHARSET=ascii ROW_FORMAT=Compact;Query OK, 0 rows affected (0.02 sec)
```

如果`VARCHAR`类型的列有`NOT NULL`属性，那最多只能存储`65533`个字节的数据，因为真实数据的长度可能占用2个字节，不需要`NULL`值标识：

Plain Text 复制代码

```
1  mysql> DROP TABLE varchar_size_demo;Query OK, 0 rows affected (0.01 sec)mys
   ql> CREATE TABLE varchar_size_demo(    ->      c VARCHAR(65533) NOT NULL
    -> ) CHARSET=ascii ROW_FORMAT=Compact;Query OK, 0 rows affected (0.02 sec)
```

如果`VARCHAR(M)`类型的列使用的不是`ascii`字符集，那会怎么样呢？来看一下：

Plain Text 复制代码

```
1  mysql> DROP TABLE varchar_size_demo;Query OK, 0 rows affected (0.00 sec)mys
   ql> CREATE TABLE varchar_size_demo(    ->       c VARCHAR(65532)    -> ) CH
   ARSET=gbk ROW_FORMAT=Compact;ERROR 1074 (42000): Column length too big for
   column 'c' (max = 32767); use BLOB or TEXT insteadmysql> CREATE TABLE varch
   ar_size_demo(    ->       c VARCHAR(65532)    -> ) CHARSET=utf8 ROW_FORMAT=
   Compact;ERROR 1074 (42000): Column length too big for column 'c' (max = 218
   45); use BLOB or TEXT instead
```

从执行结果中可以看出，如果`VARCHAR(M)`类型的列使用的不是`ascii`字符集，那`M`的最大取值取决于该字符集表示一个字符最多需要的字节数。在列的值允许为`NULL`的情况下，`gbk`字符集表示

一个字符最多需要 `2` 个字符，那在该字符集下， `M` 的最大取值就是 `32766` （也就是：65532/2），也就是说最多能存储 `32766` 个字符； `utf8` 字符集表示一个字符最多需要 `3` 个字符，那在该字符集下， `M` 的最大取值就是 `21844` ，就是说最多能存储 `21844` （也就是：65532/3）个字符。

```
小贴士：上述所言在列的值允许为NULL的情况下，gbk字符集下M的最大取值就是32766，utf8字符集下M的最大取值就是21844，这都是在表中只有一个字段的情况下说的，一定要记住一个行中的所有列（不包括隐藏列和记录头信息）占用的字节长度加起来不能超过65535个字节！
```

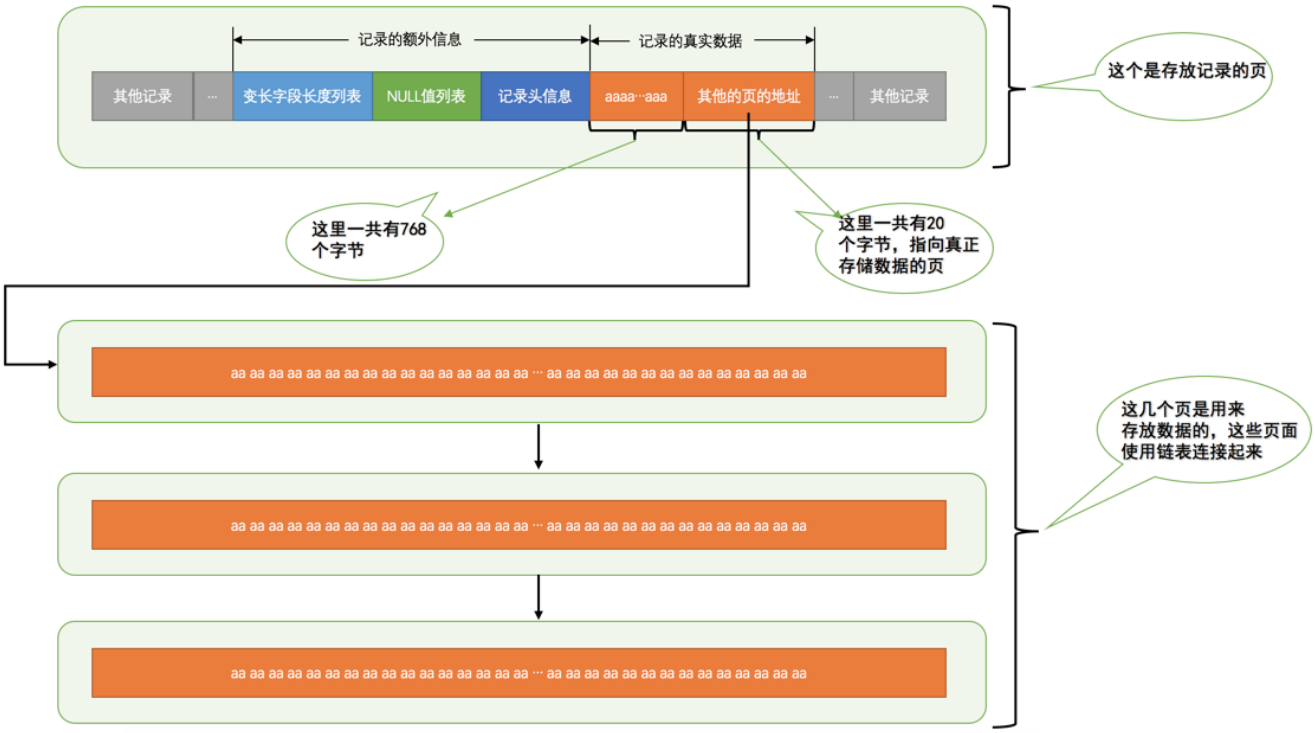
记录中的数据太多产生的溢出

我们以 `ascii` 字符集下的 `varchar_size_demo` 表为例，插入一条记录：

```
mysql> CREATE TABLE varchar_size_demo(    ->       c VARCHAR(65532)    -> ) CHARSET=ascii ROW_FORMAT=Compact;Query OK, 0 rows affected (0.01 sec)mysql> INSERT INTO varchar_size_demo(c) VALUES(REPEAT('a', 65532));Query OK, 1 row affected (0.00 sec)
```

其中的 `REPEAT('a', 65532)` 是一个函数调用，它表示生成一个把字符 `'a'` 重复 `65532` 次的字符串。前边说过， `MySQL` 中磁盘和内存交互的基本单位是 `页` ，也就是说 `MySQL` 是以 `页` 为基本单位来管理存储空间的，我们的记录都会被分配到某个 `页` 中存储。而一个页的大小一般是 `16KB` ，也就是 `16384` 字节，而一个 `VARCHAR(M)` 类型的列就最多可以存储 `65532` 个字节，这样就可能造成一个页存放不了一条记录的尴尬情况。

在 `Compact` 和 `Reduntant` 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，把剩余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的占用的字节数），从而可以找到剩余数据所在的页，如图所示：



从图中可以看出来，对于 `Compact` 和 `Reduntant` 行格式来说，如果某一列中的数据非常多的话，在本记录的真实数据处只会存储该列的前 `768` 个字节的数据和一个指向其他页的地址，然后把剩下的数据存放到其他页中，这个过程也叫做 `行溢出` ，存储超出 `768` 字节的那些页面也被称为 `溢出页` 。画一个简图就是这样：

最后需要注意的是，不只是 *VARCHAR(M)* 类型的列，其他的 *TEXT*、*BLOB* 类型的列在存储数据非常多的时候也会发生 `行溢出` 。

行溢出的临界点

那发生 `行溢出` 的临界点是什么呢？也就是说在列存储多少字节的数据时就会发生 `行溢出` ？

MySQL 中规定一个页中至少存放两行记录，至于为什么这么规定我们之后再说，现在看一下这个规定造成的影响。以上边的 varchar_size_demo 表为例，它只有一个列 c ，我们往这个表中插入两条记录，每条记录最少插入多少字节的数据才会 行溢出 的现象呢？这得分析一下页中的空间都是如何利用的。

- 每个页除了存放我们的记录以外，也需要存储一些额外的信息，乱七八糟的额外信息加起来需要 136 个字节的空间（现在只要知道这个数字就好了），其他的空间都可以被用来存储记录。
- 每个记录需要的额外信息是 27 字节。
  这27个字节包括下边这些部分：
  - 2个字节用于存储真实数据的长度
  - 1个字节用于存储列是否是NULL值
  - 5个字节大小的头信息
  - 6个字节的 row_id 列
  - 6个字节的 transaction_id 列
  - 7个字节的 roll_pointer 列

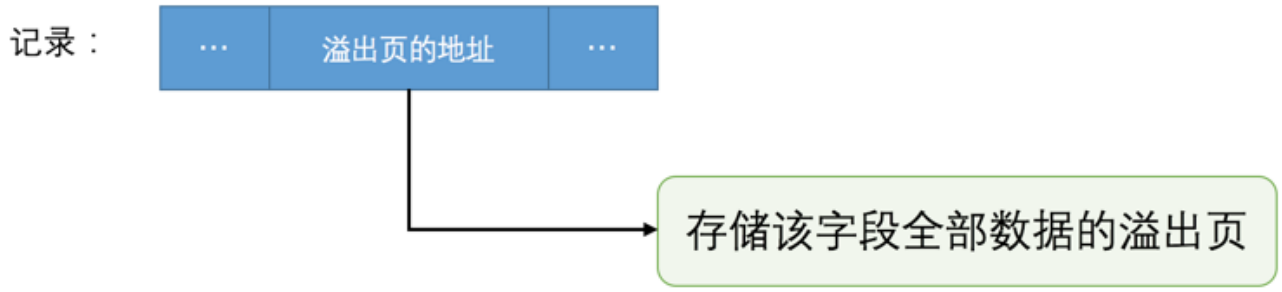
假设一个列中存储的数据字节数为n，那么发生 行溢出 现象时需要满足这个式子：

```
Plain Text | 复制代码
1  136 + 2×(27 + n) > 16384
```

求解这个式子得出的解是： n > 8098 。也就是说如果一个列中存储的数据不大于 8098 个字节，那就不会发生 行溢出 ，否则就会发生 行溢出 。不过这个 8098 个字节的结论只是针对只有一个列的 varchar_size_demo 表来说的，如果表中有多个列，那上边的式子和结论都需要改一改了，所以重点就是：你不用关注这个临界点是什么，只要知道如果我们想一个行中存储了很大的数据时，可能发生 行溢出 的现象。

Dynamic和Compressed行格式

下边要介绍另外两个行格式， Dynamic 和 Compressed 行格式，我现在使用的 MySQL 版本是 5.7 ，它的默认行格式就是 Dynamic ，这俩行格式和 Compact 行格式挺像，只不过在处理 行溢出 数据时有点儿分歧，它们不会在记录的真实数据处存储字段真实数据的前 768 个字节，而是把所有的字节都存储到其他页面中，只在记录的真实数据处存储其他页面的地址，就像这样：



`Compressed` 行格式和 `Dynamic` 不同的一点是， `Compressed` 行格式会采用压缩算法对页面进行压缩，以节省空间。

CHAR(M)中的M值过大的情况

`CHAR(M)` 类型的列可以存储的最大字节长度等于该列使用的字符集表示一个字符需要的最大字节数和 `M` 的乘积。如果某个列使用的是 `CHAR(M)` 类型，并且它可以存储的最大字节长度超过 `768` 字节，那么不论我们使用的是上述4种的哪种行格式， `InnoDB` 都会把该列当成变长字段看待。比方说采用 `utf8mb4` 的 `CHAR(255)` 类型的列将会被当作变长字段看待，因为 `4×255 > 768` 。

总结

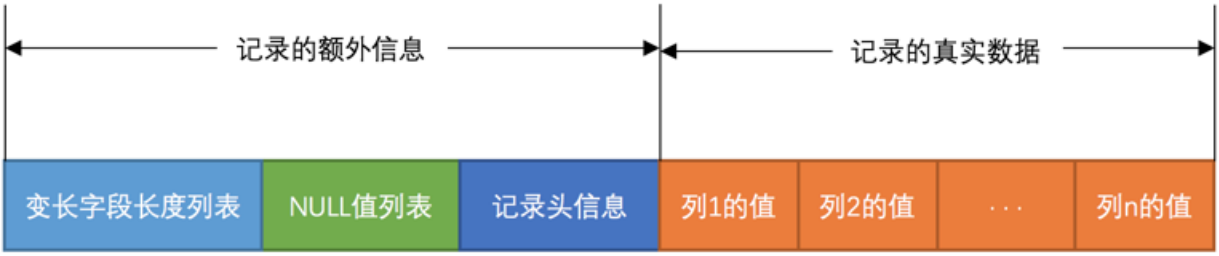
1. 页是 `MySQL` 中磁盘和内存交互的基本单位，也是 `MySQL` 是管理存储空间的基本单位。
2. 指定和修改行格式的语法如下：

```
Plain Text | 复制代码
1  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
```

3. `InnoDB` 目前定义了4中行格式
   - COMPACT行格式

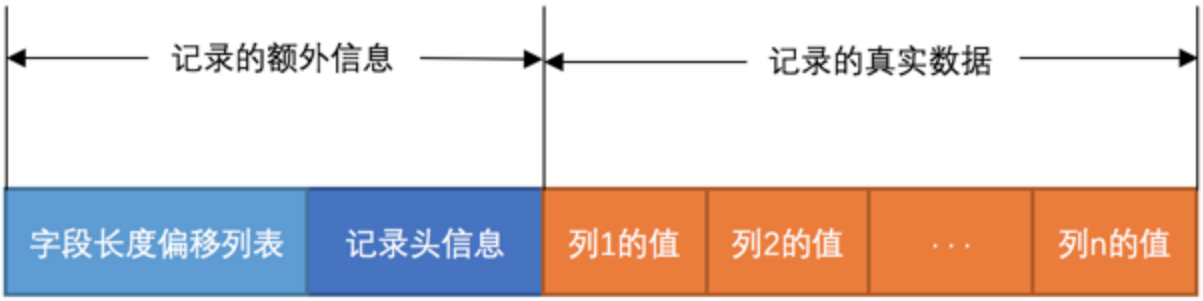
     具体组成如图：

Compact行格式示意图



- Redundant行格式
  具体组成如图：

Redundant行格式示意图



- Dynamic和Compressed行格式
  这两种行格式类似于`COMPACT行格式`，只不过在处理行溢出数据时有点儿分歧，它们不会在记录的真实数据处存储字符串的前768个字节，而是把所有的字节都存储到其他页面中，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。

- **一个页一般是`16KB`，当记录中的数据太多，当前页放不下的时候，会把多余的数据存储到其他页中，这种现象称为`行溢出`。**