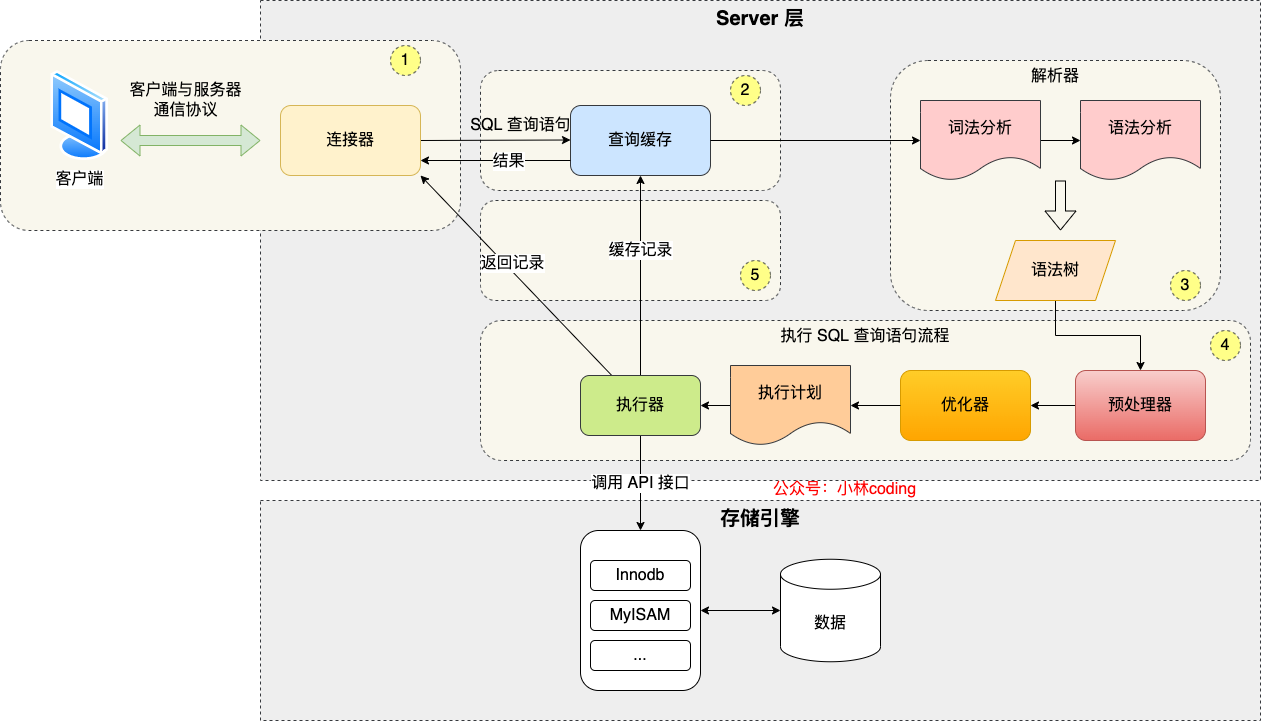
**执行一条 select 语句，期间发生了什么？**

## MySQL 执行流程是怎样的？

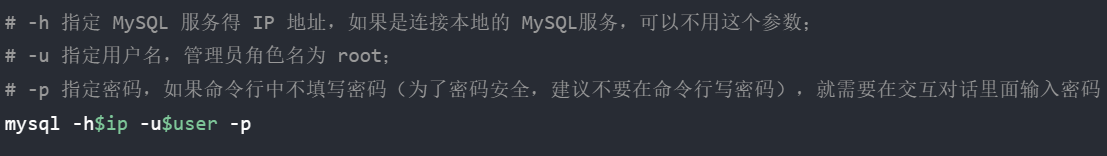


MySQL 的架构共分为两层：**Server 层和存储引擎层**

* **Server 层负责建立连接、分析和执行 SQL**。MySQL 大多数的核心功能模块都在这实现，主要包括连接器，查询缓存、解析器、预处理器、优化器、执行器等。另外，所有的内置函数（如日期、时间、数学和加密函数等）和所有跨存储引擎的功能（如存储过程、触发器、视图等。）都在 Server 层实现。
* **存储引擎层负责数据的存储和提取**。支持 InnoDB、MyISAM、Memory 等多个存储引擎，不同的存储引擎共用一个 Server 层。现在最常用的存储引擎是 InnoDB，从 MySQL 5.5 版本开始， InnoDB 成为了 MySQL 的默认存储引擎。我们常说的索引数据结构，就是由存储引擎层实现的，不同的存储引擎支持的索引类型也不相同，比如 InnoDB 支持索引类型是 B+树 ，且是默认使用，也就是说在数据表中创建的主键索引和二级索引默认使用的是 B+ 树索引。

## 第一步：连接器

Linux连接MySQL服务：



如果用户密码都没有问题，连接器就会获取该用户的权限，然后保存起来，后续该用户在此连接里的任何操作，都会基于连接开始时读到的权限进行权限逻辑的判断。

所以，如果一个用户已经建立了连接，**即使管理员中途修改了该用户的权限，也不会影响已经存在连接的权限**。修改完成后，只有再新建的连接才会使用新的权限设置。

MySQL 是基于 **TCP 协议**进行传输的

MySQL 的连接也跟 HTTP 一样，有短连接和长连接的概念



## 第二步：查询缓存

如果 SQL 是查询语句（select 语句），MySQL 就会先去**查询缓存**（ Query Cache ）里查找缓存数据，看看之前有没有执行过这一条命令，这个查询缓存是以 key-value 形式保存在内存中的，key 为 SQL 查询语句，value 为 SQL 语句查询的结果。如果查询的语句命中查询缓存，那么就会直接返回 value 给客户端。

缺点：

**对于更新比较频繁的表，查询缓存的命中率很低**的，因为只要一个表有更新操作，那么这个表的查询缓存就会被清空。**如果刚缓存了一个查询结果很大的数据，还没被使用的时候，刚好这个表有更新操作，查询缓冲就被清空了，相当于缓存了个寂寞。**所以，MySQL 8.0 版本直接将查询缓存删掉了，也就是说 MySQL 8.0 开始，执行一条 SQL 查询语句，不会再走到查询缓存这个阶段了。

**这里说的查询缓存是 server 层的，也就是 MySQL 8.0 版本移除的是 server 层的查询缓存，并不是 Innodb 存储引擎中的 buffer pool。**

## 第三步：解析 SQL

正式执行 SQL 查询语句之前， MySQL 会先对 SQL 语句做解析，这个工作交由「**解析器**」来完成

### 解析器

① **词法分析**：MySQL 根据输入的字符串识别出关键字

② **语法分析**：根据词法分析的结果，语法解析器会根据语法规则，判断输入的 SQL 语句是否满足 MySQL 语法，如果没问题就会构建出 SQL 语法树，方便后面模块获取 SQL 类型、表名、字段名、 where 条件等等。

## 第四步：执行 SQL

每条SELECT 查询语句流程主要可以分为下面这三个阶段：

① prepare 阶段，也就是**预处理**阶段；

② optimize 阶段，也就是**优化**阶段；

③ execute 阶段，也就是**执行**阶段；

### 预处理器

* 检查 SQL 查询语句中的**表或者字段是否存在**；
* 将 select \* 中的 **\* 符号，扩展为表上的所有列**；

### 优化器

**优化器主要负责将 SQL 查询语句的执行方案确定下来**，比如在表里面有多个索引的时候，优化器会基于查询成本的考虑，来决定选择使用哪个索引。

### 执行器

执行器和存储引擎交互，交互是**以记录为单位**的。

#### 主键索引查询



这条查询语句的查询条件用到了主键索引，而且是等值查询，同时主键 id 是唯一，不会有 id 相同的记录，所以优化器决定选用**访问类型为 const** 进行查询

#### 全表扫描



这条查询语句的查询条件没有用到索引，所以优化器决定选用**访问类型为 ALL** 进行查询，也就是全表扫描的方式查询

#### 索引下推

索引下推能够**减少二级索引**在查询时的**回表操作**，提高查询的效率，因为它将 Server 层部分负责的事情，交给存储引擎层去处理了。



① 不使用索引下推（MySQL 5.6 之前的版本）：

* Server 层首先调用存储引擎的接口定位到满足查询条件的**第一条**二级索引记录，也就是定位到 age > 20 的第一条记录；
* 存储引擎根据二级索引的 B+ 树快速定位到这条记录后，获取主键值，然后**进行回表操作**，将完整的记录返回给 Server 层；
* Server 层在判断该记录的 reward 是否等于 100000，如果成立则将其发送给客户端；否则跳过该记录；
* 接着，继续向存储引擎索要下一条记录，存储引擎在二级索引定位到记录后，获取主键值，然后回表操作，将完整的记录返回给 Server 层；
* 如此往复，直到存储引擎把表中的所有记录读完。

每查询到一条二级索引记录，都要进行回表操作

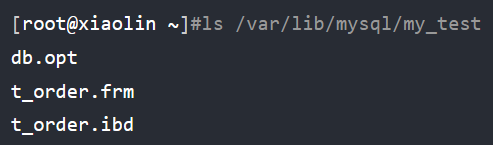
② 使用索引下推（MySQL 5.6 推出的查询优化策略）

* Server 层首先调用存储引擎的接口定位到满足查询条件的第一条二级索引记录，也就是定位到 age > 20 的第一条记录；
* **存储引擎定位到二级索引后，先不执行回表操作，而是先判断一下该索引中包含的列（reward列）的条件（reward 是否等于 100000）是否成立。如果条件不成立，则直接跳过该二级索引。如果成立，则执行回表操作，**将完成记录返回给 Server 层。
* Server 层再判断其他的查询条件（本次查询没有其他条件）是否成立，如果成立则将其发送给客户端；否则跳过该记录，然后向存储引擎索要下一条记录。
* 如此往复，直到存储引擎把表中的所有记录读完。

# MySQL 一行记录是怎么存储的？

## MySQL 的数据存放在哪个文件？

以 **InnoDB 存储引擎**为例。Linux中每创建一个 database（数据库） 都会在 /var/lib/mysql/ 目录里面创建一个以 database 为名的目录。



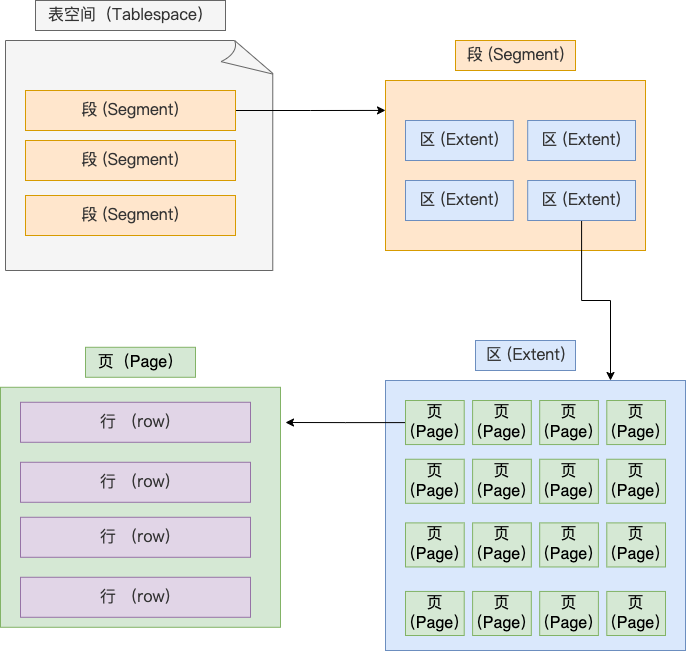
* db.opt，用来存储当前数据库的**默认字符集**和**字符校验规则**。
* t\_order.frm ，t\_order 的**表结构**会保存在这个文件。**在 MySQL 中每建立一张表都会生成一个.frm 文件，该文件是用来保存每个表的元数据信息的**，主要包含表结构定义。
* t\_order.ibd，t\_order 的**表数据**会保存在这个文件。表数据既可以存在共享表空间文件（文件名：ibdata1）里，也可以存放在独占表空间文件（文件名：表名字.ibd）。这个行为是由参数 innodb\_file\_per\_table 控制的，若设置了参数 innodb\_file\_per\_table 为 1，则会将存储的数据、索引等信息单独存储在一个独占表空间，**从 MySQL 5.6.6 版本开始**，它的默认值就是 1 了，因此从这个版本之后， **MySQL 中每一张表的数据都存放在一个独立的 .ibd 文件**。

「 表名字.ibd 」文件也称为**独占表空间文件**。

### 表空间文件的结构是怎么样的？

**表空间由段（segment）、区（extent）、页（page）、行（row）组成**

InnoDB存储引擎的逻辑存储结构大致如下图：



#### 1、行（row）

数据库表中的记录都是按行进行存放的，每行记录根据不同的行格式，有不同的存储结构。

#### 2、页（page）

InnoDB 的数据是**按「页」为单位来读写**的，也就是说，当需要读一条记录的时候，并不是将这个行记录从磁盘读出来，而是以页为单位，将其整体读入内存。**默认每个页的大小为 16KB。**

**页是 InnoDB 存储引擎磁盘管理的最小单元**

页的类型有很多，常见的有**数据页**、**undo 日志页**、**溢出页**等等。

#### 3、区（extent）

InnoDB 存储引擎是用 B+ 树来组织数据的。B+ 树中每一层都是通过双向链表连接起来的，如果是以页为单位来分配存储空间，那么链表中相邻的两个页之间的物理位置并不是连续的，可能离得非常远，那么磁盘查询时就会有大量的随机I/O，**随机 I/O 是非常慢的**。

解决这个问题也很简单，就是**让链表中相邻的页的物理位置也相邻**，这样就可以使用顺序 I/O 了，那么在范围查询（扫描叶子节点）的时候性能就会很高。

**在表中数据量大的时候，为某个索引分配空间的时候就不再按照页为单位分配了，而是按照区（extent）为单位分配。每个区的大小为 1MB，对于 16KB 的页来说，连续的 64 个页会被划为一个区，这样就使得链表中相邻的页的物理位置也相邻，就能使用顺序 I/O 了**。

#### 4、段（segment）

段一般分为数据段、索引段和回滚段等：

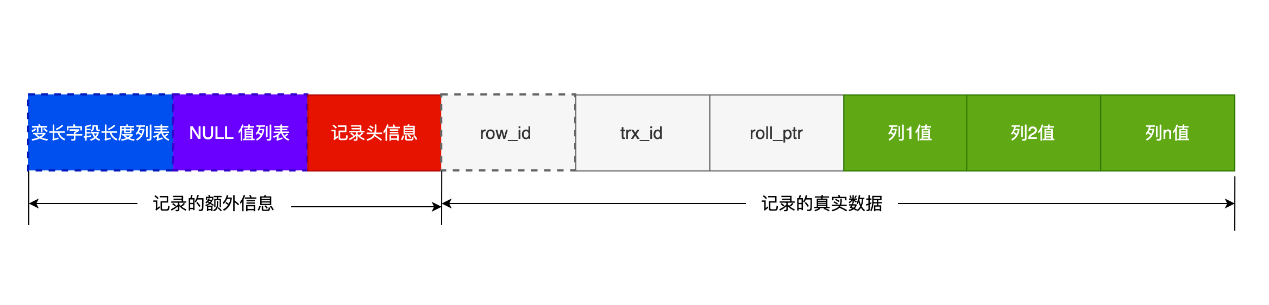
* **索引段**：存放 B + 树的非叶子节点的区的集合；
* **数据段**：存放 B + 树的叶子节点的区的集合；
* **回滚段**：存放的是回滚数据的区的集合， MVCC 利用了回滚段实现了多版本查询数据。

## InnoDB 行格式有哪些？

行格式（row\_format），就是**一条记录的存储结构**。InnoDB 提供了 4 种行格式，分别是 Redundant、Compact、Dynamic和 Compressed 行格式。

* Redundant 是MySQL 5.0 版本之前用的行格式，现在基本没人用了。Redundant **不是一种紧凑的行格式**。
* MySQL 5.0 之后引入了 Compact 行记录存储方式，Compact **是一种紧凑的行格式**，设计的初衷就是**为了让一个数据页中可以存放更多的行记录**，从 MySQL 5.1 版本之后，行格式默认设置成 Compact。
* Dynamic 和 Compressed 两个都是**紧凑的行格式**，它们的行格式都和 Compact 差不多，因为都是基于 Compact 改进一点东西。从 MySQL5.7 版本之后，默认使用 Dynamic 行格式。

## COMPACT 行格式长什么样？

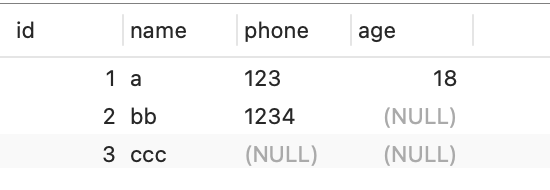


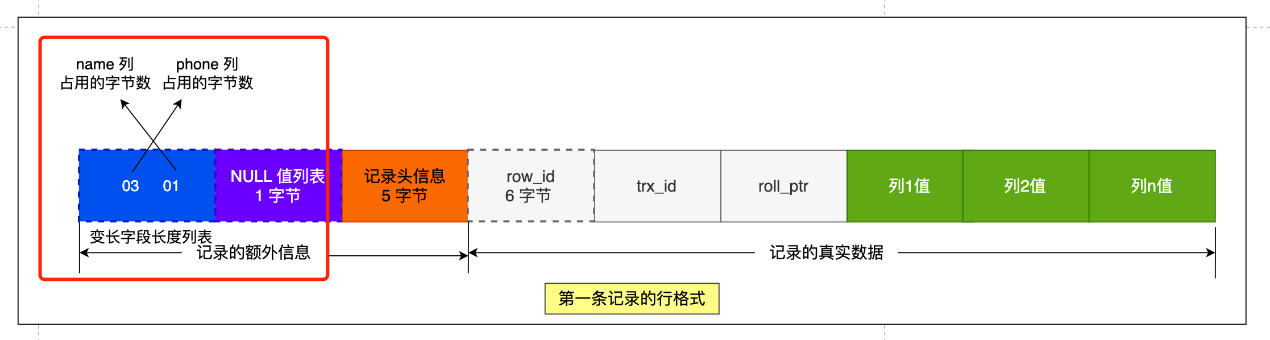
一条完整的记录分为「**记录的额外信息**」和「**记录的真实数据**」两个部分。

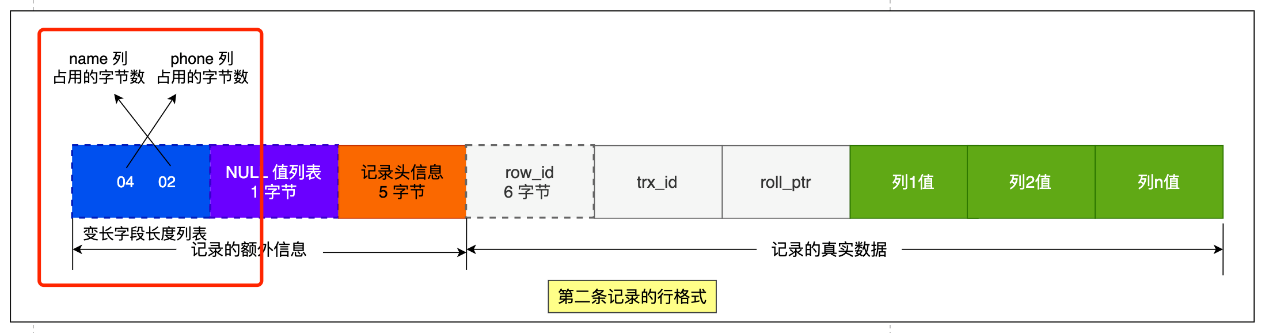
### 记录的额外信息

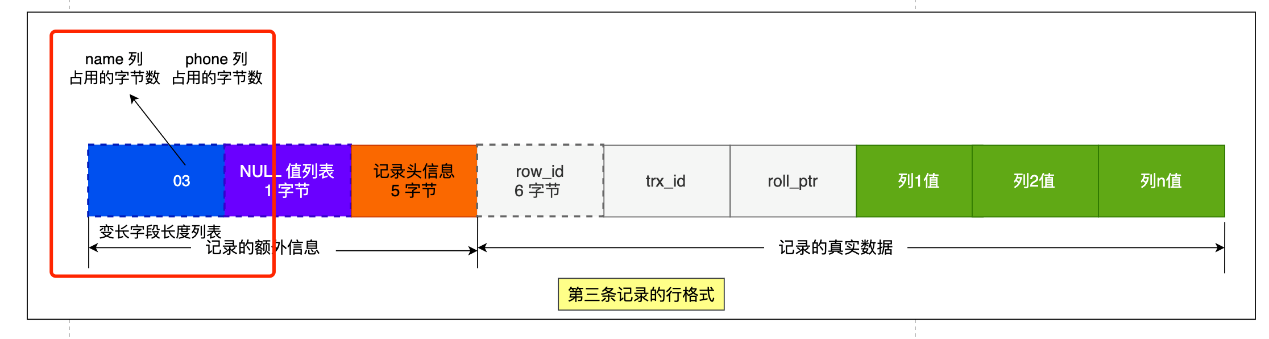
#### 1. 变长字段长度列表

在存储数据的时候，也要**把数据占用的大小存起来**（VARCHAR、TEXT、VARBINARY、BLOB等），存到「变长字段长度列表」里面，读取数据的时候才能根据这个「变长字段长度列表」去读取对应长度的数据。变长字段的真实数据占用的字节数会按照列的顺序**逆序存放**









**第三条记录**中 phone 列的值是 NULL，**NULL 是不会存放在行格式中记录的真实数据部分里的**，所以「变长字段长度列表」里不需要保存值为 NULL 的变长字段的长度。

为什么「变长字段长度列表」的信息要按照**逆序存放**？

**这个设计是有想法的，主要是因为「记录头信息」中指向下一个记录的指针，指向的是下一条记录的「记录头信息」和「真实数据」之间的位置，这样的好处是向左读就是记录头信息，向右读就是真实数据，比较方便。mysql的行记录解析并不是从可变长度列表开始解析的，而是从主键开始，同时往前解析头信息，null值列表，可变长度列表，同时往后解析真实数据**。

「变长字段长度列表」中的信息之所以要逆序存放，是因为这样可以**使得位置靠前的记录的真实数据和数据对应的字段长度信息可以同时在一个 CPU Cache Line 中，这样就可以提高 CPU Cache 的命中率**。 ？

每个数据库表的行格式都有「变长字段字节数列表」吗？

「变长字段长度列表」**只出现在数据表有变长字段的时候**。

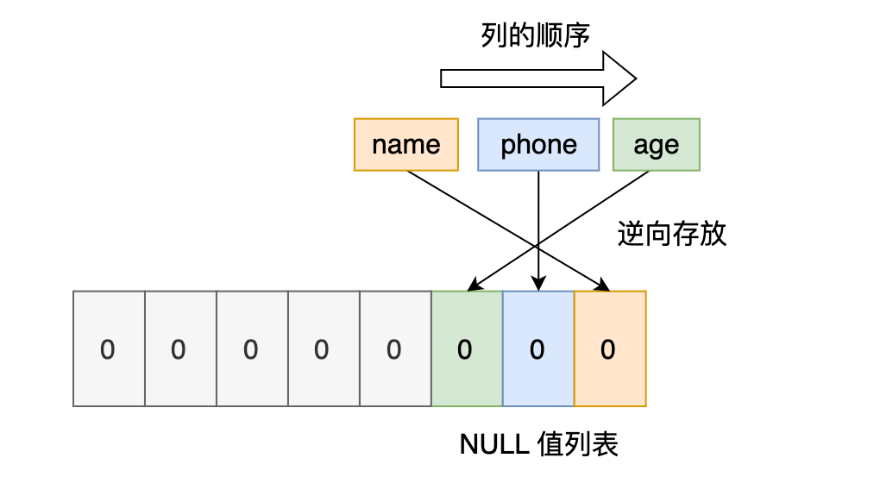
#### 2. NULL 值列表

如果存在允许 NULL 值的列，则每个列对应一个二进制位（bit），二进制位按照列的顺序**逆序排列**。

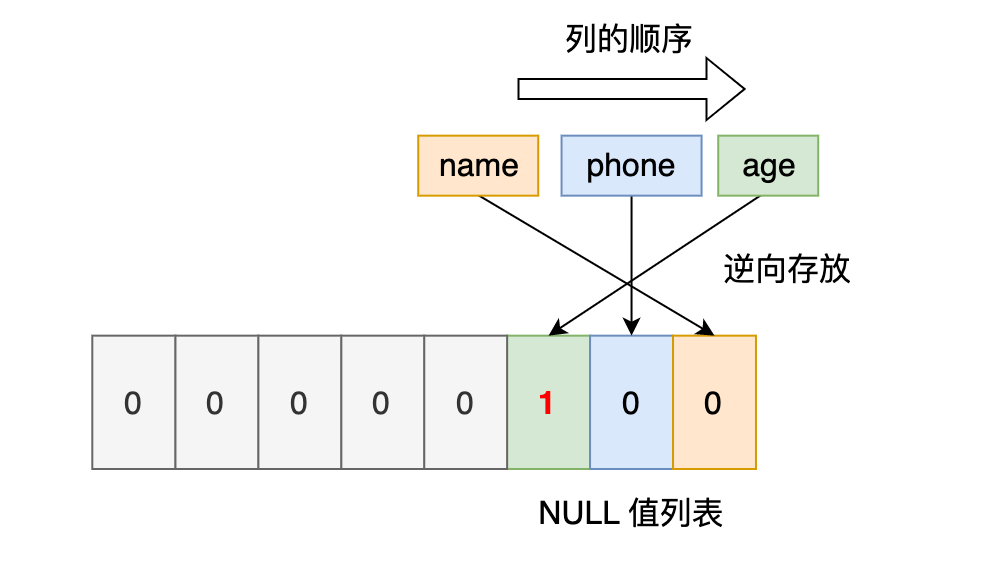
* 二进制位的值为1时，代表该列的值为NULL。
* 二进制位的值为0时，代表该列的值不为NULL。

InnoDB 是用整数字节的二进制位来表示 NULL 值列表的，不足 8 位要在高位补 0

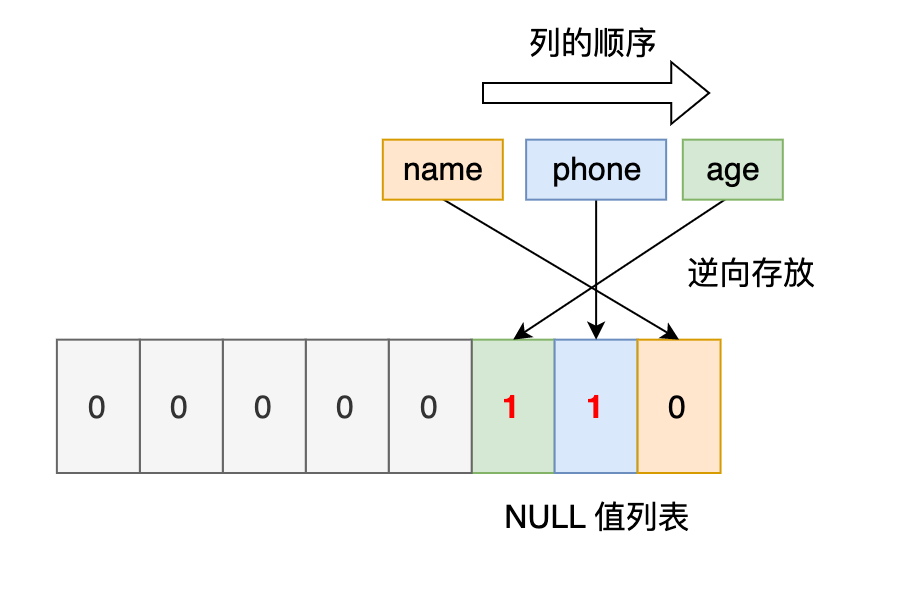
第一条记录：



第二条记录：



第三条记录：





每个数据库表的行格式都有「NULL 值列表」吗？

NULL 值列表也不是必须的。**当数据表的字段都定义成 NOT NULL 的时候，这时候表里的行格式就不会有 NULL 值列表了**。所以在设计数据库表的时候，**通常都是建议将字段设置为 NOT NULL，这样可以至少节省 1 字节的空间**

「NULL 值列表」是固定 1 字节空间吗？如果这样的话，一条记录有 9 个字段值都是 NULL，这时候怎么表示？

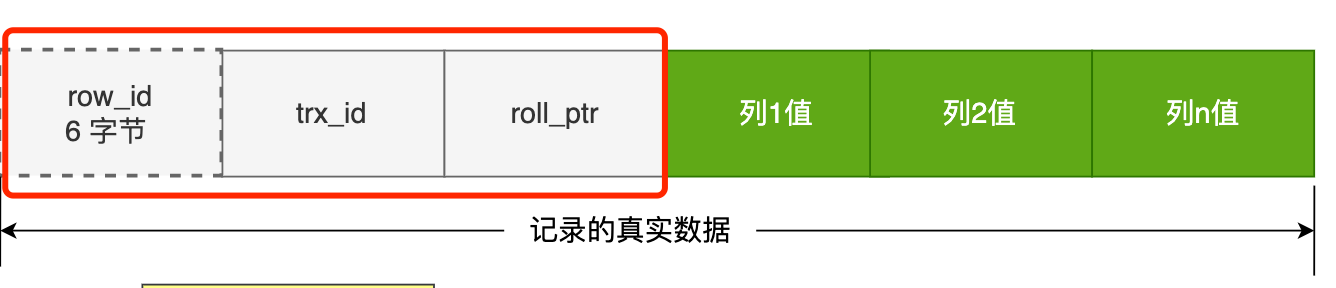
当一条记录有 9 个字段值都是 NULL，那么就会创建 2 字节空间的「NULL 值列表」，以此类推。

#### 3. 记录头信息

其中比较重要的有：

* delete\_mask ：**标识此条数据是否被删除**。从这里可以知道，**我们执行 detele 删除记录的时候，并不会真正的删除记录，只是将这个记录的 delete\_mask 标记为 1。**
* next\_record：**下一条记录的位置**。从这里可以知道，记录与记录之间是通过**链表**组织的。
* record\_type：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

### 记录的真实数据



* row\_id

如果建表的时候**指定了主键或者唯一约束列，那么就没有 row\_id 隐藏字段了**。如果**既没有指定主键，又没有唯一约束，那么 InnoDB 就会为记录添加 row\_id 隐藏字段**。row\_id不是必需的，占用 6 个字节。

* trx\_id

事务id，表示这个数据是**由哪个事务生成的**。 trx\_id是必需的，占用 6 个字节。

* roll\_pointer

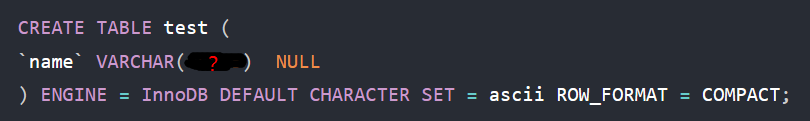
这条**记录上一个版本的指针**。roll\_pointer 是必需的，占用 7 个字节。

trx\_id 和 roll\_pointer与**MVCC机制**有关

## varchar(n) 中 n 最大取值为多少？

**MySQL 规定除了 TEXT、BLOBs 这种大对象类型之外，其他所有的列（不包括隐藏列和记录头信息，包括变长字段长度列表和null值列表）占用的字节长度加起来不能超过 65535 个字节**。

**varchar(n) 字段类型的 n 代表的是最多存储的字符数量，并不是字节大小**



**以允许null（存在null值列表）的情况为例：**

### 单字段的情况

每个变长字段的「变长字段长度」需要的字节数：

* 如果变长字段允许存储的最大字节数**小于等于 255 字节**，就会用 **1 字节**表示「变长字段长度」；
* 如果变长字段允许存储的最大字节数**大于 255 字节**，就会用 **2 字节**表示「变长字段长度」；

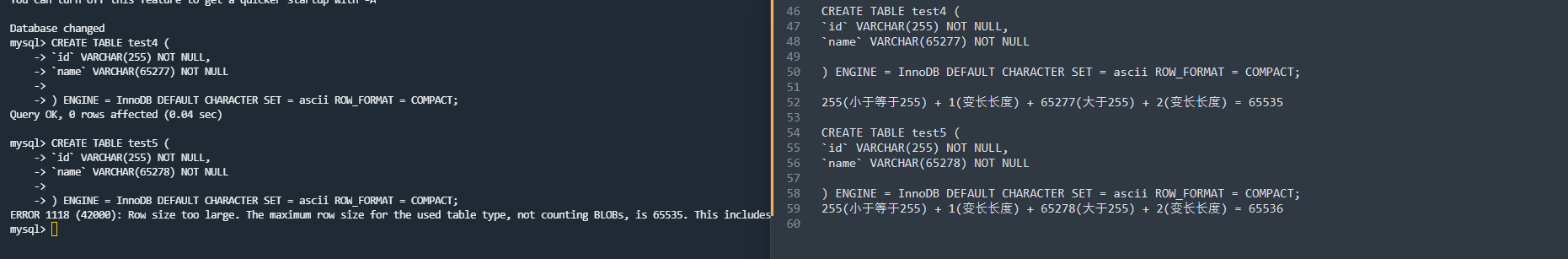
**结论：**

**① 在数据库表只有一个 varchar(n) 字段且字符集是 ascii 的情况下，varchar(n) 中 n 最大值 = 65535 - 2 - 1 = 65532**。

**② 在 UTF-8 字符集下，一个字符最多需要三个字节，varchar(n) 的 n 最大取值就是 65532/3 = 21844**

### 多字段的情况

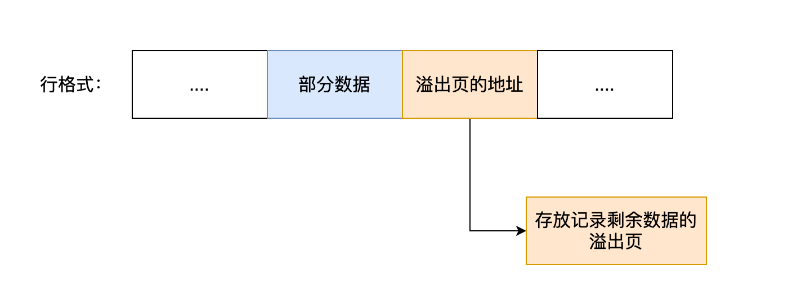
**保证所有字段的长度 + 变长字段字节数列表所占用的字节数 + NULL值列表所占用的字节数 <= 65535**



## 行溢出后，MySQL 是怎么处理的？

MySQL 中磁盘和内存交互的基本单位是页，一个页的大小一般是 16KB，也就是 16384字节，而一个 varchar(n) 类型的列最多可以存储 65532字节，一些大对象如 TEXT、BLOB 可能存储更多的数据，这时一个页可能就存不了一条记录。这个时候就会**发生行溢出。**一般情况下，InnoDB 的数据都是存放在 「数据页」中。但是当发生行溢出时，溢出的数据会存放到「**溢出页**」中。

**Compact格式**发生行溢出时，在记录的真实数据处只会**保存该列的一部分数据**，而**把剩余的数据放在「溢出页」中**，然后真实数据处用 20 字节存储指向溢出页的地址，从而可以找到剩余数据所在的页。



**Compressed** 和 **Dynamic 格式**和 Compact 非常类似，主要的区别在于处理行溢出数据时有些区别。这两种格式采用**完全的行溢出方式**，**记录的真实数据处不会存储该列的一部分数据，只存储 20 个字节的指针来指向溢出页**。而实际的数据都存储在溢出页中

