InnoDB数据页结构

原创 小孩子 我们都是小青蛙 2018-04-01

各位，看完顺便给关注了呗，要不然写文章连个读者都没有，你知道这啥感觉不～ 如果觉得写的不错帮到你了请转发，感激不尽。

本文可能是你读过的所有关于InnoDB页结构的全面描述中最简单的一个，请注意我写了个`可能`，如果你发现还有比本篇文章更简单的文章或书籍，千万千万要私聊我，我一定会让文章变得更简单的。虽说文章不难懂，但是还是要认真学习的，**碎片化阅读绝对是不可能从小白到大神的**，想实现薪资double一定要静下心来慢慢读，发现写的比较绕或者解释不清楚或者原理有错误的地方请留言，大家共同进步～

上集回顾

上集我们唠叨了 InnoDB 存储引擎中的各种行格式，并且完整的梳理了一遍一条普通的记录被存储到底层的存储器上的过程。 InnoDB 中目前支持 COMPACT 、 Redundant 、 Dynamic 和 Compressed 四种行格式，各种行格式都是由额外信息和实际数据组成的，这些额外信息很重要，存储引擎依靠着它们来访问具体的数据。

上集中还简单提了一下 页 的概念，它是 MySQL 管理存储空间的基本单位，一个页的大小一般是 16KB ，并且我们知道了记录其实是被存放在 页 中的，如果记录占用的空间太大还可能造成 行溢出 现象，这会导致一条记录被分散存储在多个页中。本集中将详细看一下InnoDB存储引擎中 页 的结构。

数据页结构的快速浏览

页 的本质就是一块 16KB 大小的存储空间， InnoDB 为了不同的目的而把 页 分为不同的类型，其中用于存放记录的页也称为 数据页 ，我们先看看这个用于存放记录的页长什么样。数据页代表的这块 16KB 大小的存储空间可以被划分为多个部分，不同部分有不同的功能，各个部分如图所示：

从图中可以看出，一个 InnoDB 数据页的存储空间被划分成了 7 个部分，每个部分又可以被划分为若干小部分。下边我们用表格的方式来大致描述一下这7个部分（快速的瞅一眼就行了，后边会详细唠叨的）：

名称	中文名	占用空间大小	简单描述
File Header	文件头	38 字节	一些描述页的信息
Page Header	页头	56 字节	页的状态信息
Infimum + Supremum	最小记录和最大记录	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

名称	中文名	占用空间大小	简单描述
Free Space	空闲空间	不确定	页中尚未使用的空间
Page Directory	页目录	不确定	页中的记录相对位置
File Trailer	文件结尾	8 字节	校验页是否完整

小贴士：

我们接下来并不打算按照页中各个部分的出现顺序来依次介绍它们，因为各个部分中会出现很多大家目前不理解

记录在页中的存储

在页的7个组成部分中，我们自己存储的记录会按照我们指定的 行格式 存储到 User Records 部分。但是在一开始生成页的时候，其实并没有 User Records 这个部分，每当我们插入一条记录，都会从 Free Space 部分，也就是尚未使用的存储空间中申请一个记录大小的空间划分到 User Records 部分，当 Free Space 部分的空间全部被 User Records 部分替代掉之后，也就意味着这个页使用完了，如果还有新的记录插入的话，就需要去申请新的页了，这个过程的图示如下：

为了更好的管理在 User Records 中的这些记录， MySQL 可费了一番力气呢，在哪费力气了呢？不就是把记录按照指定的行格式一条一条摆在 User Records 部分么？其实这话还得从记录行格式的 记录头信息 中说起。

记录头信息的秘密

为了故事的顺利发展，我们先创建一个表：

```
mysql> CREATE TABLE page_demo(
    ->     c1 INT,
    ->     c2 INT,
    ->     c3 VARCHAR(10000),
    ->     PRIMARY KEY (c1)
    -> ) CHARSET=ascii ROW_FORMAT=Compact;
Query OK, 0 rows affected (0.03 sec)
mysql>
```

这个新创建的 `page_demo` 表有3个列，其中 `c1` 和 `c2` 列是用来存储整数的， `c3` 列是用来存储字符串的。需要注意的是，我们把 ***c1*** 列指定为主键，所以在具体的行格式中MySQL就没必要为我们去创建那个所谓的 ***row_id*** 隐藏列了。而且我们为这个表指定了 `ascii` 字符集以及 `Compact` 的行格式。所以这个表中记录的行格式示意图就是这样的：

从图中可以看到，我们特意把 记录头信息 的5个字节的数据给标出来了，说明它很重要，我们先把这些 记录头信息 中各个属性的大体意思浏览一下（不用仔细看，我们下边会详细说明的）：

名称	大小（单位：bit）	描述

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	标记该记录是否为B+树的非叶子节点中的最小记录
n_owned	4	表示当前槽管理的记录数
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

由于我们现在主要在唠叨 记录头信息 的作用，所以为了大家理解上的方便，我们只在 page_demo 表的行格式演示图中画出有关的头信息属性以及 c1 、 c2 、 c3 列的信息（其他信息没画不代表它们不存在啊，只是为了理解上的方便在图中省略了～），简化后的行格式示意图就是这样：

下边我们试着向 page_demo 表中插入几条记录：

```
mysql> INSERT INTO page_demo VALUES(1, 100, 'aaaa'), (2, 200, 'bbbb'), (3, 300, '
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql>
```

看看这些记录在 页 的 User Records 部分中是怎么表示的，为了方便大家分析，我把记录中头信息和实际的列数据都用十进制表示出来了（其实是一堆二进制位），所以这些记录的示意图就是：

看这个图的时候需要注意一下，各条记录在 User Records 中存储的时候并没有空隙，这里只是为了大家观看方便才把每条记录单独画在一行中。我们对照着这个图来看看记录头信息中

的各个属性是啥意思：

`delete_mask`

这个属性标记着当前记录是否被删除，占用1个二进制位，值为 `0` 的时候代表记录并没有被删除，为 `1` 的时候代表记录被删除掉了。

what？被删除的记录还在 页 中么？是的，你看到的并不是它实际上的样子，你以为它删除了，可它还在真实的磁盘上[摊手]（忽然想起冠希～），只是打了个标记表示这条记录删除了而已～

> 小贴士：
> 这些被删除的记录之所以不立即从磁盘上移除，是因为移除它们之后把其他的记录在磁盘上重新排列需要性能
> 所以只是打个删除标记而已，而且这部分存储空间之后还可以重用，也就是说之后如果有新记录插入到表中的
> 可能把这些被删除的记录占用的存储空间覆盖掉。
>
> 如果你想彻底的从磁盘上移除这些被删除的记录，可以使用这个语句：
>
> `optimize table 表名;`
>
> 执行这个命令后服务器会重新规划表中记录的存储方式，把被标记为删除的记录从磁盘上移除。

- `min_rec_mask`

  这个属性标记该记录是否为 `B+` 树的非叶子节点中的最小记录，什么是个 `B+` 树？什么是个非叶子节点？好吧，等会再聊这个问题。反正我们自己插入的四条记录的 `min_rec_mask` 值都是 `0` ，意味着它们都不是 `B+` 树的非叶子节点中的最小记录。

- `n_owned`

  这个暂时保密，稍后它是主角～

- `heap_no`

  这个属性表示当前记录在本 页 中的位置，从图中可以看出来，我们插入的4条记录在本 页 中的位置分别是： `2` 、 `3` 、 `4` 、 `5` 。是不是少了点啥？是的，怎么不见 `heap_no` 值为 `0` 和 `1` 的记录呢？

  这其实是设计 `InnoDB` 的大叔们玩的一个小把戏，他们自动给每个页里边儿加了两个记录，由于这两个记录并不是我们自己插入的，所以有时候也称为 伪记录 或者 虚

拟记录 。这两个伪记录一个代表 最小记录 ，一个代表 最大记录 ，wait~，记录可以比大小么?

是的，记录也可以比大小，对于一条完整的记录来说，比较记录的大小就是比较 主键 的大小。比方说我们插入的4行记录的主键值分别是： 1 、 2 、 3 、 4 ，这也就意味着这4条记录的大小依次递增。

小贴士:
请注意我强调了对于`一条完整的记录`来说，比的是主键的大小。后边我们还会介绍只存储一条记录的部

但是不管我们向 页 中插入了多少自己的记录，设计 InnoDB 的大叔们都规定他们定义的两条伪记录分别为最小记录与最大记录。这两条记录的构造十分简单，都是由5字节大小的 记录头信息 和8字节大小的一个固定的部分组成的，如图所示

由于这两条记录不是我们自己定义的记录，所以它们并不存放在 页 的 User Records 部分，他们被单独放在一个称为 Infimum + Supremum 的部分，如图所示:

从图中我们可以看出来，最小记录和最大记录的 `heap_no` 值分别是 `0` 和 `1` ，也就是说它们的位置最靠前。

- `record_type`

  这个属性表示当前记录的类型，一共有4种类型的记录， `0` 表示普通记录， `1` 表示B+树非叶节点记录， `2` 表示最小记录， `3` 表示最大记录。从图中我们也可以看出来，我们自己插入的记录就是普通记录，它们的 `record_type` 值都是 `0` ，而最小记录和最大记录的 `record_type` 值分别为 `2` 和 `3` 。

  至于 `record_type` 为 `1` 的情况，我们之后在说索引的时候会重点强调的。

- `next_record`

  这玩意儿非常重要，它表示从当前记录的真实数据到下一条记录的真实数据的地址偏移量。比方说第一条记录的 `next_record` 值为 `36` ，意味着从第一条记录的真实数据的地址处向后找 `36` 个字节便是下一条记录的真实数据。如果你熟悉数据结构的话，就立即明白了，这其实是个 链表 ，可以通过一条记录找到它的下一条记录。但是需要注意注意再注意的一点是， 下一条记录 指得并不是按照我们插入顺序的下一条记录，而是按照主键值由小到大的顺序的下一条记录。而且规定 ***最小记录*** 的下一条记录就本页中主键值最小的记录，而本页中主键值最大的记录的下一条记录就是 ***最大记录*** ，为了更形象的表示一下这个 `next_record` 起到的作用，我们用箭头来替代一下 `next_record` 中的地址偏移量：

从图中可以看出来，我们的记录按照从小到大的顺序形成了一个单链表。 最大记录 的 next_record 的值为 0 ， 这也就是说最大记录是没有 下一条记录 了， 它是这个单链表中的最后一个节点。如果从中删除掉一条记录，这个链表也是会跟着变化的，比如我们把第2条记录删掉：

```
mysql> DELETE FROM page_demo WHERE c1 = 2;
Query OK, 1 row affected (0.02 sec)

mysql>
```

删掉第2条记录后的示意图就是：

从图中可以看出来，删除第2条记录前后主要发生了这些变化：

所以，不论我们怎么对页中的记录做增删改操作，InnoDB始终会维护一条记录的单链表，链表中的各个节点是按照主键值由小到大的顺序连接起来的。

- 第2条记录并没有从存储空间中移除，而是把该条记录的 `delete_mask` 值设置为 `1` 。
- 第2条记录的 `next_record` 值变为了0，意味着该记录没有下一条记录了。
- 第1条记录的 `next_record` 指向了第3条记录。
- 还有一点你可能忽略了，就是 `最大记录` 的 `n_owned` 值从 `5` 变成了 `4` ，关于这一点的变化我们稍后会详细说明的。

再来看一个有意思的事情，因为主键值为 `2` 的记录被我们删掉了，但是存储空间却没有回收，如果我们再次把这条记录插入到表中，会发生什么事呢?

```
mysql> INSERT INTO page_demo VALUES(2, 200, 'bbbb');
Query OK, 1 row affected (0.00 sec)

mysql>
```

我们看一下记录的存储情况：

从图中可以看到， `InnoDB` 并没有因为新记录的插入而为它申请新的存储空间，而是直接复用了原来被删除记录的存储空间。

页目录

上边我们了解了记录在页中按照主键值由小到大顺序串联成一个单链表，那如果我们想根据主键值查找页中的某条记录该咋办呢？比如说这样的查询语句：

```
SELECT * FROM page_demo WHERE c1 = 3;
```

最笨的办法：从最小记录开始，沿着链表一直往后找，总有一天会找到（或者找不到[摊手]），在找的时候还能投机取巧，因为链表中各个记录的值是按照从小到大顺序排列的，所以当链表的某个节点代表的记录的主键值大于你想要查找的主键值时，你就可以停止查找了，因为该节点后边的节点的主键值依次递增。

这个方法在页中存储的记录数量比较少的情况用起来也没啥问题，比方说现在我们的表里只有 4 条自己插入的记录，所以最多找 4 次就可以把所有记录都遍历一遍，但是如果一个页中存储了非常多的记录，这么查找对性能来说还是有损耗的，所以我们说这是一个 笨 办法。但是设计 InnoDB 的大叔们是什么人，他们能用这么笨的办法么，当然是要设计一种更6的查找方式喽，他们从书的目录中找到了灵感。

我们平常想从一本书中查找某个内容的时候，一般会先看目录，找到需要查找的内容对应的书的页码，然后到对应的页码查看内容。设计 InnoDB 的大叔们为我们的记录也制作了一个类似的目录，他们的制作过程是这样的：

1. 将所有正常的记录（包括最大和最小记录，不包括标记为已删除的记录）划分为几个组。
2. 每个组的最后一条记录的头信息中的 n_owned 属性表示该组内共有几条记录。
3. 将每个组的最后一条记录的地址偏移量按顺序存储起来，每个地址偏移量也被称为一个 槽 （英文名： Slot ）。这些地址偏移量都会被存储到靠近 页 的尾部的地方，页中存储地址偏移量的部分也被称为 Page Directory （可以看前边数据页的组成示意图）。

比方说现在的 page_demo 表中正常的记录共有6条， InnoDB 会把它们分成两组，第一组中只有一个最小记录，第二组中是剩余的5条记录，看下边的示意图：

从这个图中我们需要注意这么几点：

- 现在 `Page Directory` 部分中有两个槽，也就意味着我们的记录被分成了两个组， `槽0` 中的值是 `112` ，代表最大记录的地址偏移量； `槽1` 中的值是 `99` ，代表最小记录的地址偏移量。

- 注意最小和最大记录的头信息中的 `n_owned` 属性

    - 最小记录的 `n_owned` 值为 `1` ，这就代表着以最小记录结尾的这个分组中只有 `1` 条记录，也就是最小记录本身。
    - 最大记录的 `n_owned` 值为 `5` ，这就代表着以最大记录结尾的这个分组中只有 `5` 条记录，包括最大记录本身还有我们自己插入的 `4` 条记录。

`99` 和 `112` 这样的地址偏移量很不直观，我们用箭头指向的方式替代数字，这样更易于我们理解，所以修改后的示意图就是这样：

哎呀，咋看上去怪怪的，这么乱的图对于我这个强迫症真是不能忍，那我们就暂时不管各条记录在存储设备上的排列方式了，单纯从逻辑上看一下这些记录和页目录的关系：

这样看就顺眼多了嘛！为什么最小记录的 `n_owned` 值为1，而最大记录的 `n_owned` 值为 `5` 呢，这里头有什么猫腻么？

是的，设计 `InnoDB` 的大叔们对每个分组中的记录条数是有规定的，对于最小记录所在的分组只能有 ***1*** 条记录，最大记录所在的分组拥有的记录条数只能在 ***1~8*** 条之间，剩下的分组中记录

的条数范围只能在是 ***4~8*** 条之间。所以分组是按照下边的步骤进行的：

- 初始情况下一个数据页里只有最小记录和最大记录两条记录，它们分属于两个分组。
- 之后每插入一跳记录都把这条记录放到最大记录所在的组，直到最大记录所在组中的记录数等于8个。
- 在最大记录所在组中的记录数等于8个的时候再插入一条记录时，将最大记录所在组平均分裂成2个组，然后最大记录所在的组就剩下4条记录了，然后就可以把即将插入的那条记录放到该组中了。

由于现在 `page_demo` 表中的记录太少，无法演示添加了 页目录 之后加快查找速度的过程，所以再往 `page_demo` 表中添加一些记录：

```
mysql> INSERT INTO page_demo VALUES(5, 500, 'eeee'), (6, 600, 'ffff'), (7, 700, '
Query OK, 12 rows affected (0.00 sec)
Records: 12  Duplicates: 0  Warnings: 0

mysql>
```

哈，我们一口气又往表中添加了12条记录，现在就一共有16条正常的记录了（包括最小和最大记录），这些记录被分成了5个组，如图所示：

因为把16条记录的全部信息都画在一张图里太占地方，让人眼花缭乱的，所以只保留了头信息中的 `n_owned` 和 `next_record` 属性，也省略了各个记录之间的箭头，我没画不等于没有

啊！现在看怎么从这个 页目录 中查找记录。因为各个槽代表的记录的主键值都是从小到大排序的，所以我们可以使用所谓的 二分法 来进行快速查找。4个槽的编号分别是： 0 、 1 、 2 、 3 、 4 ，所以初始情况下最低的槽就是 low=0 ，最高的槽就是 high=4 。比方说我们想找主键值为 5 的记录，过程是这样的：

1. 计算中间槽的位置： (0+4)/2=2 ，所以查看 槽2 对应记录的主键值为 8 ，又因为 8 > 5 ，所以设置 high=2 ， low 保持不变。

2. 重新计算中间槽的位置： (0+2)/2=1 ，所以查看 槽1 对应的主键值为 4 。所以设置 low=1 ， high 保持不变。

3. 因为 high - low 的值为1，所以确定主键值为 5 的记录在槽1和槽2之间，接下来就是遍历链表的查找了。

所以在一个数据页中查找指定主键值的记录的过程分为两步：

1. 通过二分法确定该记录所在的槽。

2. 通过记录的 next_record 属性组成的链表遍历查找该槽中的各个记录。

小贴士：

如果你不知道二分法是个什么东西，找个基础算法书看看吧。什么？算法书写的看不懂？等我～

Page Header

设计 InnoDB 的大叔们为了能得到一个数据页中存储的记录的状态信息，比如本页中已经存储了多少条记录，第一条记录的地址是什么，页目录中存储了多少个槽等等，特意在页中定义了一个叫 Page Header 的部分，它是 页 结构的第二部分，这个部分占用固定的 56 个字节，专门存储各种状态信息，具体各个字节都是干嘛的看下表：

名称	占用空间大小	描述
PAGE_N_DIR_SLOTS	2 字节	在页目录中的槽数量

名称	占用空间大小	描述
PAGE_HEAP_TOP	2 字节	第一个记录的地址
PAGE_N_HEAP	2 字节	本页中的记录的数量（包括最小和最大记录以及标记为删除的记录）
PAGE_FREE	2 字节	指向可重用空间的地址（就是标记为删除的记录地址）
PAGE_GARBAGE	2 字节	已删除的字节数，行记录结构中 delete_flag 为1的记录大小总数
PAGE_LAST_INSERT	2 字节	最后插入记录的位置
PAGE_DIRECTION	2 字节	最后插入的方向
PAGE_N_DIRECTION	2 字节	一个方向连续插入的记录数量
PAGE_N_RECS	2 字节	该页中记录的数量（不包括最小和最大记录以及被标记为删除的记录）
PAGE_MAX_TRX_ID	8 字节	修改当前页的最大事务ID，该值仅在二级索引中定义
PAGE_LEVEL	2 字节	当前页在索引树中的位置，高度
PAGE_INDEX_ID	8 字节	索引ID，表示当前页属于哪个索引
PAGE_BTR	10 字节	非叶节点所在段的segment header，仅在B+树的Root页定义
PAGE_LEVEL	10 字节	B+树所在段的segment header，仅在B+树的Root页定义

如果大家认真看过前边的文章，那么大致能看明白这里头前边一半左右的状态信息的意思，剩下的状态信息看不明白不要着急，饭要一口一口吃，东西要一点一点学。在这里我们想强调以下 PAGE_DIRECTION 和 PAGE_N_DIRECTION 的意思。

- PAGE_DIRECTION

  假如新插入的一条记录的主键值比上一条记录的主键值比上一条记录大，我们说这条记录的插入方向是右边，反之则是左边。用来表示最后一条记录插入方向的状态就是 PAGE_DIRECTION 。

- PAGE_N_DIRECTION

  假设连续几次插入新记录的方向都是一致的， InnoDB 会把沿着同一个方向插入记录的条数记下来，这个条数就用 PAGE_N_DIRECTION 这个状态表示。当然，如果最后一条记录的插入方向改变了的话，这个状态的值会被清零重新统计。

如果大家是初次学习 Page Header 结构的话，可能从 PAGE_N_DIRECTION 这个状态信息之后的状态信息的释义会不太清楚，这些状态信息我们在后边唠叨索引的时候会继续说的，稍安勿躁。

File Header

如果说 Page Header 描述的是 页 内的各种状态信息，比方说页里头有多少个记录了呀，有多少个槽了呀，那么 File Header 描述的就是 页 外的各种状态信息，比方说这个页的编号是多少，它的上一个页、下一个页是谁啦吧啦吧啦～ 这两者的关系还真有点武警和解放军的关系。 File Header 是 InnoDB 页的第一部分，这个部分占用固定的 38 个字节，下边我们看看这个部分的各个字节都是代表啥意思吧：

名称	占用空间大小	描述
FIL_PAGE_SPACE_OR_CHKSUM	4 字节	页的校验和（checksum值）
FIL_PAGE_OFFSET	4 字节	页号

名称	占用空间大小	描述
FIL_PAGE_PREV	4 字节	上一个页的页号
FIL_PAGE_NEXT	4 字节	下一个页的页号
FIL_PAGE_LSN	8 字节	最后被修改的日志序列位置（英文名是：Log Sequence Number）
FIL_PAGE_TYPE	2 字节	该页的类型
FIL_PAGE_FILE_FLUSH_LSN	8 字节	仅在系统表空间的一个页中定义，代表文件至少被更新到了该LSN值，独立表空间中都是0
FIL_PAGE_ARCH_LOG_NO_OR_SPACE_ID	4 字节	页属于哪个表空间

对照着这个表格，我们看几个目前比较重要的部分：

- FIL_PAGE_SPACE_OR_CHKSUM

  这个代表当前页面的校验和（checksum）。啥是个校验和？就是对于一个很长很长的字节串来说，我们会通过某种算法来计算一个比较短的值来代表这个很长的字节串，这个比较短的值就称为 校验和 。这样在比较两个很长的字节串之前先比较这两个长字节串的校验和，如果校验和都不一样两个长字节串肯定是不同的，所以省去了直接比较两个比较长的字节串的时间损耗。

- FIL_PAGE_OFFSET

  每一个 页 都有一个单独的页号，就跟你的身份证号码一样， InnoDB 通过页号来可以唯一定位一个 页 。

- FIL_PAGE_TYPE

这个代表当前 页 的类型，我们前边说过， InnoDB 为了不同的目的而把页分为不同的类型，本集中介绍的其实都是存储记录的 数据页 ，其实还有很多别的类型的页，具体如下表：

名称	十六进制	描述
FIL_PAGE_ALLOCATED	0x0000	最新分配，还没使用
FIL_PAGE_UNDO_LOG	0x0002	Undo Log页
FIL_PAGE_INODE	0x0003	段信息的节点
FIL_PAGE_IBUUF_FRE_LIST	0x0004	Insert Buffer空闲列表
FIL_PAGE_IBUF_BITMAP	0x0005	Insert Buffer位图
FIL_PAGE_TYPE_SYS	0x0006	系统页
FIL_PAGE_TYPE_TRX_SYS	0x0007	事务系统数据
FIL_PAGE_TYPE_FSP_HDR	0x0008	File Space Header
FIL_PAGE_TYPE_XDES	0x0009	扩展描述页
FIL_PAGE_TYPE_BLOB	0x000A	BLOB页
FIL_PAGE_INDEX	0x45BF	B+树的子节点

我们存放记录的数据页的类型其实是 FIL_PAGE_INDEX ，也就是所谓的 B+树叶子节点 ，后边介绍索引的时候详细再说啥是个 B+树叶子节点 ～

- FIL_PAGE_PREV 和 FIL_PAGE_NEXT

一张表中可以有成千上万条记录，一个页只有 16KB ，所以可能需要好多页来存放数据， FIL_PAGE_PREV 和 FIL_PAGE_NEXT 就分别代表本页的上一个和下一个页的页号。需要注意的是，并不是所有类型的页都有上一个和下一个页的属性，不过我们本集中唠叨的 数据页 是有这两个属性的，所以所有的数据页其实是一个双链表，就像这样：

关于 File Header 的其他属性我们暂时用不到，等用到的时候再提哈～

File Trailer

我们知道 InnoDB 存储引擎会把数据存储到磁盘上，但是磁盘速度太慢，需要以 页 为单位把数据加载到内存中处理，如果该页中的数据在内存中被修改了，那么在修改后的某个时间需要把数据同步到磁盘中。但是在同步了一半的时候中断电了咋办，这不是莫名尴尬么？为了检测一个页是否完整（也就是在同步的时候有没有发生只同步一半的尴尬情况），设计 InnoDB 的大叔们在每个页的尾部都加了一个 File Trailer 部分，这个部分由 8 个字节组成，可以分成2个小部分：

- 前四个字节代表页的校验和

  这个部分是和 File Header 中的校验和相对应的。每当一个页面在内存中修改了，在同步之前就要把它的校验和算出来，因为 File Header 在页面的前边，所以校验和会被首先同步到磁盘，当完全写完时，校验和也会被写到页的尾部，如果完全同步成功，则页的首部和尾部的校验和应该是一致的，反之意味着同步中间出了错。

- 后四个字节代表日志序列位置（LSN）

  这个部分也是为了校验页的完整性的，只不过我们目前还没说 LSN 是个什么意思，所以大家可以先不用管这个属性。

总结

1. InnoDB为了不同的目的而设计了不同类型的页，用于存放我么记录的页也叫做 数据页 。

2. 一个数据页可以被分为7个部分，分别是

   - `File Header` ，表示文件头，占固定的38字节。
   - `Page Header` ，表示页里的一些状态信息，占固定的56个字节。
   - `Infimum + Supremum` ，两个虚拟的伪记录，分别表示页中的最小和最大记录，占固定的 `26` 个字节。
   - `User Records` ： 真实存储我们插入的记录的部分，大小不固定。
   - `Free Space` ： 页中尚未使用的部分，大小不确定。
   - `Page Directory` ： 页中的记录相对位置，也就是各个槽在页面中的地址偏移量，大小不固定，插入的记录越多，这个部分占用的空间越多。
   - `File Trailer` ： 用于检验页是否完整的部分，占用固定的8个字节。

3. 每个记录的头信息中都有一个 `next_record` 属性，从而使页中的所有记录串联成一个 单链表 。

4. `InnoDB` 会为把页中的记录划分为若干个组，每个组的最后一个记录的地址偏移量作为一个 槽 ，存放在 `Page Directory` 中，所以在一个页中根据主键查找记录是非常快的，分为两步：

   - 通过二分法确定该记录所在的槽。
   - 通过记录的next_record属性组成的链表遍历查找该槽中的各个记录。

5. 每个数据页的 `File Header` 部分都有上一个和下一个页的编号，所以所有的数据页会组成一个 双链表 。

6. 为保证从内存中同步到磁盘的页的完整性，在页的首部和尾部都会存储页中数据的校验和和 `LSN` 值，如果首部和尾部的校验和和 `LSN` 值校验不成功的话，就说明同步过程出现了问题。

文章已于2018-04-03修改