# InnoDB 的表空间

标签： MySQL 是怎样运行的

---

通过前边儿的内容大家知道，表空间是一个抽象的概念，对于系统表空间来说，对应着文件系统中一个或多个实际文件；对于每个独立表空间来说，对应着文件系统中一个名为表名.ibd的实际文件。大家可以把表空间想象成被切分为许许多多个页的池子，当我们想为某个表插入一条记录的时候，就从池子中捞出一个对应的页来把数据写进去。本章内容会深入到表空间的各个细节中，带领大家在InnoDB存储结构的池子中畅游。由于本章中将会涉及比较多的概念，虽然这些概念都不难，但是却相互依赖，所以奉劝大家在看的时候：

- 不要跳着看！
- 不要跳着看！
- 不要跳着看！

# 回忆一些旧知识

## 页面类型

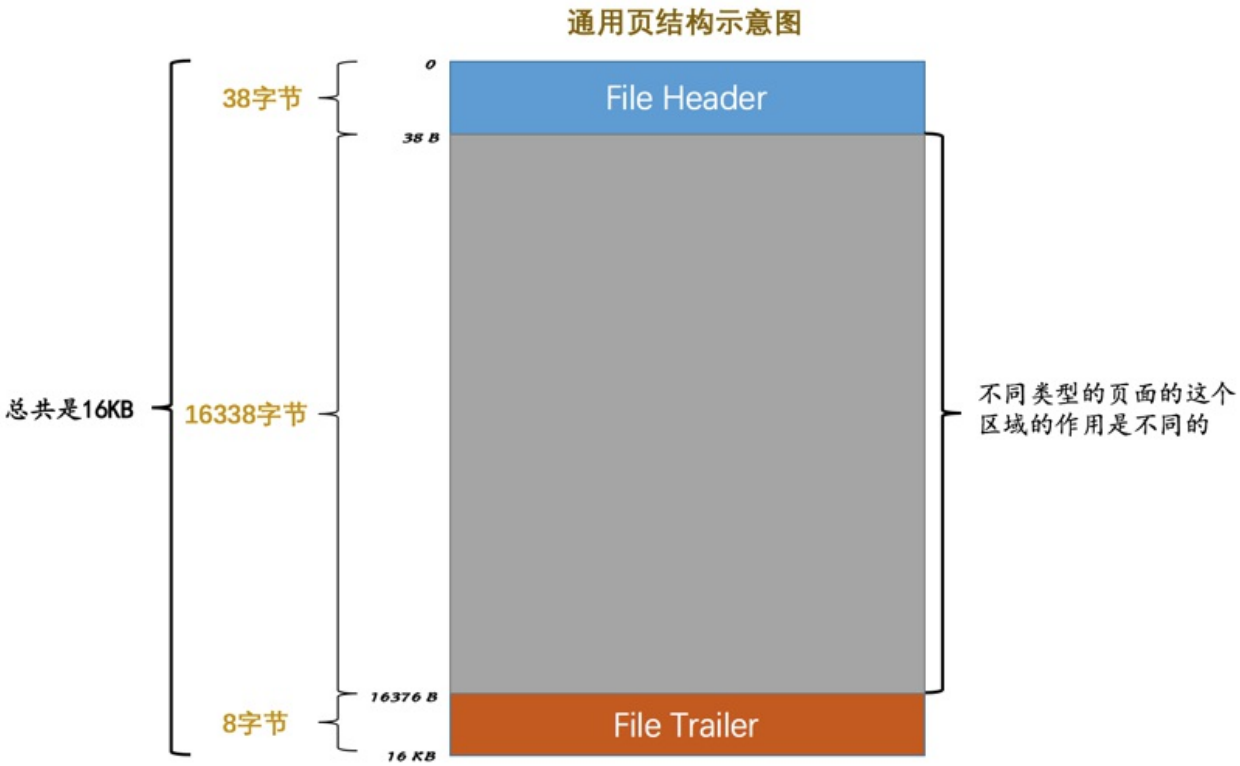
再一次强调，InnoDB 是以页为单位管理存储空间的，我们的聚簇索引（也就是完整的表数据）和其他的二级索引都是以B+树的形式保存到表空间的，而B+树的节点就是数据页。我们前边说过，这个数据页的类型名其实是：FIL_PAGE_INDEX，除了这种存放索引数据的页面类型之外，InnoDB 也为了不同的目的设计了若干种不同类型的页面，为了唤醒大家的记忆，我们再一次把各种常用的页面类型提出来：

| 类型名称 | 十六进制 | 描述 |
| --- | --- | --- |
| FIL_PAGE_TYPE_ALLOCATED | 0x0000 | 最新分配，还没使用 |
| FIL_PAGE_UNDO_LOG | 0x0002 | Undo日志页 |
| FIL_PAGE_INODE | 0x0003 | 段信息节点 |
| FIL_PAGE_IBUF_FREE_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 | 表空间头部信息 |
| FIL_PAGE_TYPE_XDES | 0x0009 | 扩展描述页 |
| FIL_PAGE_TYPE_BLOB | 0x000A | BLOB页 |
| FIL_PAGE_INDEX | 0x45BF | 索引页，也就是我们所说的数据页 |

因为页面类型前边都有个FIL_PAGE或者FIL_PAGE_TYPE的前缀，为简便起见我们后边唠叨页面类型的时候就把这些前缀省略掉了，比方说FIL_PAGE_TYPE_ALLOCATED类型称为ALLOCATED类型，FIL_PAGE_INDEX类型称为INDEX类型。

## 页面通用部分

我们前边说过数据页，也就是INDEX类型的页由7个部分组成，其中的两个部分是所有类型的页面都通用的。当然我不能寄希望于你把我说的话都记住，所以在这里重新强调一遍，任何类型的页面都有下边这种通用的结构：




从上图中可以看出，任何类型的页都会包含这两个部分：

- File Header：记录页面的一些通用信息
- File Trailer：校验页是否完整，保证从内存到磁盘刷新时内容的一致性。

对于File Trailer我们不再做过多强调，全部忘记了的话可以到将数据页的那一章回顾一下。我们这里再强调一遍File Header的各个组成部分：

| 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 |
| 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 | 页面被最后修改时对应的日志序列位置（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，代表文件至少被刷新到了对应的LSN值 |
| FIL_PAGE_ARCH_LOG_NO_OR_SPACE_ID | 4字节 | 页属于哪个表空间 |

现在除了名称里边儿带有LSN的两个字段大家可能看不懂以外，其他的字段肯定都是倍儿熟了，不过我们仍要强调这么几点：

- 表空间中的每一个页都对应着一个页号，也就是FIL_PAGE_OFFSET，这个页号由4个字节组成，也就是32个比特位，所以一个表空间最多可以拥有$2^{32}$个页，如果按照页的默认大小16KB来算，一个表空间最多支持64TB的数据。表空间的第一个页的页号为0，之后的页号分别是1，2，3...依此类推

- 某些类型的页可以组成链表，链表中的页可以不按照物理顺序存储，而是根据FIL_PAGE_PREV和FIL_PAGE_NEXT来存储上一个页和下一个页的页号。需要注意的是，这两个字段主要是为了INDEX类型的页，也就是我们之前一直说的数据页建立B+树后，为每层节点建立双向链表用的，一般类型的页是不使用这两个字段的。

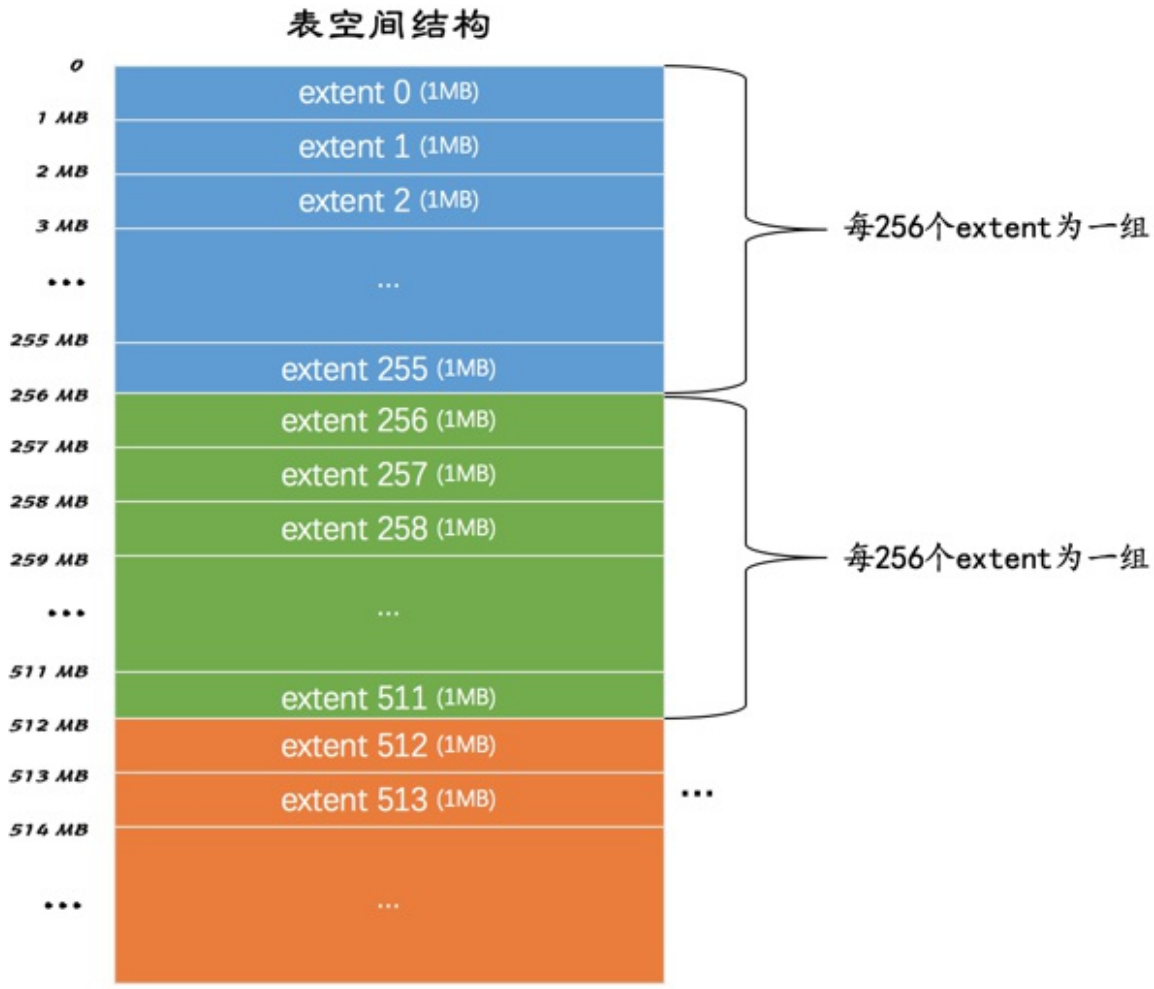
- 每个页的类型由FIL_PAGE_TYPE表示，比如像数据页的该字段的值就是0x45BF，我们后边会介绍各种不同类型的页，不同类型的页在该字段上的值是不同的。

# 独立表空间结构

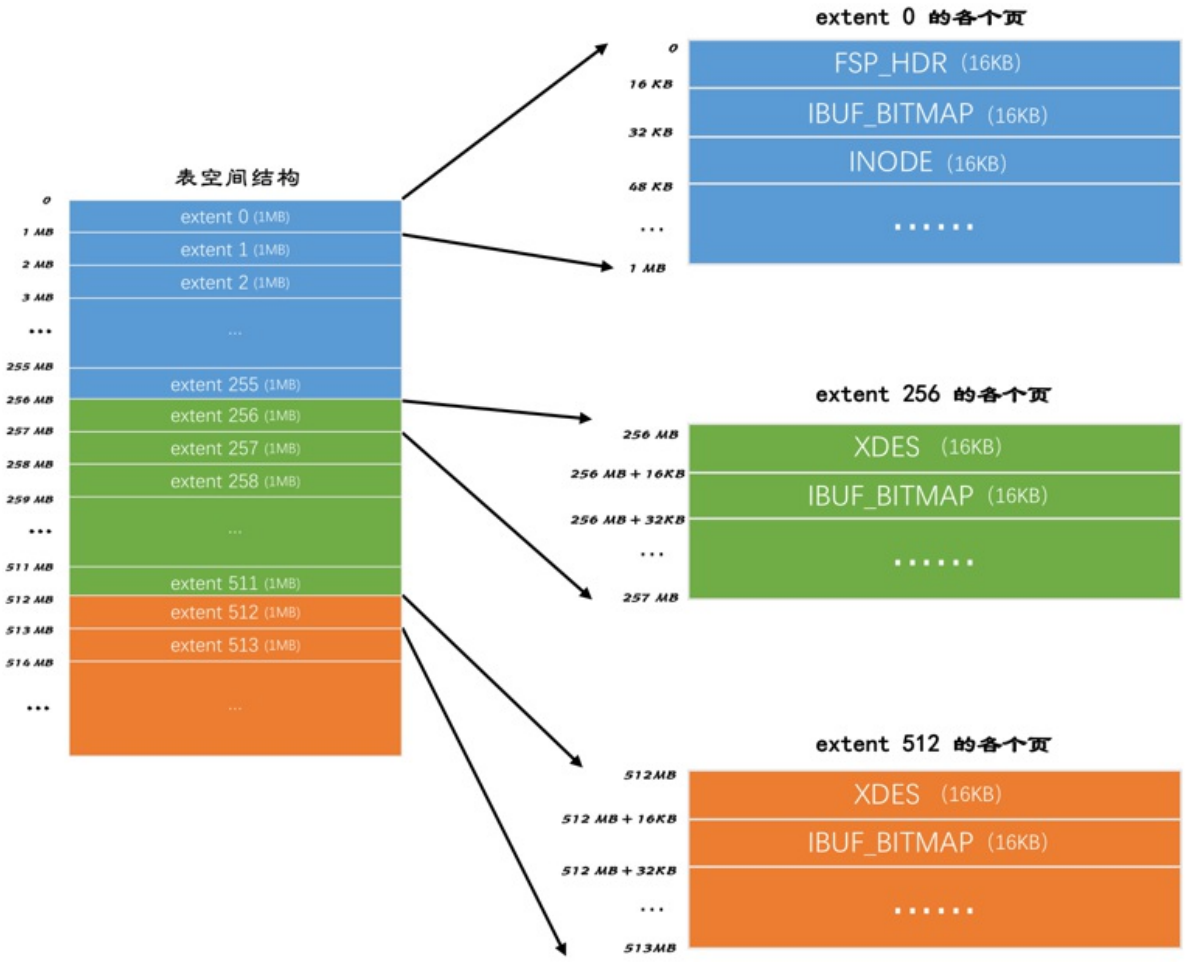
我们知道InnoDB支持许多种类型的表空间，本章重点关注独立表空间和系统表空间的结构。它们的结构比较相似，但是由于系统表空间中额外包含了一些关于整个系统的信息，所以我们先挑简单一点的独立表空间来唠叨，稍后再说系统表空间的结构。

## 区（extent）的概念

表空间中的页实在是太多了，为了更好的管理这些页面，设计InnoDB的大叔们提出了区（英文名：extent）的概念。对于16KB的页来说，连续的64个页就是一个区，也就是说一个区默认占用1MB空间大小。不论是系统表空间还是独立表空间，都可以看成是由若干个区组成的，每256个区被划分成一组。画个图表示就是这样：




其中extent 0 ~ extent 255这256个区算是第一个组，extent 256 ~ extent 511这256个区算是第二个组，extent 512 ~ extent 767这256个区算是第三个组（上图中并未画全第三个组全部的区，请自行脑补），依此类推可以划分更多的组。这些组的头几个页面的类型都是类似的，就像这样：




从上图中我们能得到如下信息：

- 第一个组最开始的由3个页面的类型是固定的，也就是说extent 0这个区最开始的3个页面的类型是固定的，分别是：
    - FSP_HDR类型：这个类型的页面是用来登记整个表空间的一些整体属性以及本组所有的区，也就是extent 0 ~ extent 255这256个区的属性，稍后详细唠叨。需要注意的一点是，整个表空间只有一个FSP_HDR类型的页面。
    - IBUF_BITMAP类型：这个类型的页面是存储本组所有的区的所有页面关于INSERT BUFFER的信息。当然，你现在不用知道啥是个INSERT BUFFER，后边会详细说到你吐。

    - INODE类型：这个类型的页面存储了许多称为INODE的数据结构，还是那句话，现在你不需要知道啥是个INODE，后边儿会说到你吐。

- 其余各组最开始的2个页面的类型是固定的，也就是说extent 256、extent 512这些区最开始的2个页面的类型是固定的，分别是：

    - XDES类型：全称是extent descriptor，用来登记本组256个区的属性，也就是说对于在extent 256区中的该类型页面存储的就是extent 256 ~ extent 511这些区的属性，对于在extent 512区中的该类型页面存储的就是extent 512 ~ extent 767这些区的属性。上边介绍的FSP_HDR类型的页面其实和XDES类型的页面的作用类似，只不过FSP_HDR类型的页面还会额外存储一些表空间的属性。

    - IBUF_BITMAP类型：上边介绍过了。

好了，宏观的结构介绍完了，里边儿的名词大家也不用记清楚，只要大致记得：表空间被划分为许多连续的区，每个区默认由64个页组成，每256个区划分为一组，每个组的最开始的几个页面类型是固定的就好了。

## 段（segment）的概念

为啥好端端的提出一个区（extent）的概念呢？我们以前分析问题的套路都是这样的：表中的记录存储到页里边儿，然后页作为节点组成B+树，这个B+树就是索引，然后吧啦吧啦一堆聚簇索引和二级索引的区别。这套路也没啥不妥的呀～

是的，如果我们表中数据量很少的话，比如说你的表中只有几十条、几百条数据的话，的确用不到区的概念，因为简单的几个页就能把对应的数据存储起来，但是你架不住表里的记录越来越多呀。

？？啥？？表里的记录多了又怎样？B+树的每一层中的页都会形成一个双向链表呀，File Header中的FIL_PAGE_PREV和FIL_PAGE_NEXT字段不就是为了形成双向链表设置的么？

是的是的，您说的都对，从理论上说，不引入区的概念只使用页的概念对存储引擎的运行并没啥影响，但是我们来考虑一下下边这个场景：

- 我们每向表中插入一条记录，本质上就是向该表的聚簇索引以及所有二级索引代表的B+树的节点中插入数据。而B+树的每一层中的页都会形成一个双向链表，如果是以页为单位来分配存储空间的话，双向链表相邻的两个页之间的物理位置可能离得非常远。我们介绍B+树索引的适用场景的时候特别提到范围查询只需要定位到最左边的记录和最右边的记录，然后沿着双向链表一直扫描就可以了，而如果链表中相邻的两个页物理位置离得非常远，就是所谓的随机I/O。再一次强调，磁盘的速度和内存的速度差了好几个数量级，随机I/O是非常慢的，所以我们应该尽量让链表中相邻的页的物理位置也相邻，这样进行范围查询的时候才可以使用所谓的顺序I/O。

所以，所以，所以才引入了区（extent）的概念，一个区就是在物理位置上连续的64个页。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，而是按照区为单位分配，甚至在表中的数据十分非常特别多的时候，可以一次性分配多个连续的区。虽然可能造成一点点空间的浪费（数据不足填充满整个区），但是从性能角度看，可以消除很多的随机I/O，功大于过嘛！

事情到这里就结束了么？太天真了，我们提到的范围查询，其实是对B+树叶子节点中的记录进行顺序扫描，而如果不区分叶子节点和非叶子节点，统统把节点代表的页面放到申请到的区中的话，进行范围扫描的效果就大打折扣了。所以设计InnoDB的大叔们对B+树的叶子节点和非叶子节点进行了区别对待，也就是说叶子节点有自己独有的区，非叶子节点也有自己独有的区。存放叶子节点的区的集合就算是

一个段（segment），存放非叶子节点的区的集合也算是一个段。也就是说一个索引会生成2个段，一个叶子节点段，一个非叶子节点段。

默认情况下一个使用InnoDB存储引擎的表只有一个聚簇索引，一个索引会生成2个段，而段是以区为单位申请存储空间的，一个区默认占用1M存储空间，所以默认情况下一个只存了几条记录的小表也需要2M的存储空间么？以后每次添加一个索引都要多申请2M的存储空间么？这对于存储记录比较少的表简直是天大的浪费。设计InnoDB的大叔们都挺节俭的，当然也考虑到了这种情况。这个问题的症结在于到现在为止我们介绍的区都是非常纯粹的，也就是一个区被整个分配给某一个段，或者说区中的所有页面都是为了存储同一个段的数据而存在的，即使段的数据填不满区中所有的页面，那余下的页面也不能挪作他用。现在为了考虑以完整的区为单位分配给某个段对于数据量较小的表太浪费存储空间的这种情况，设计InnoDB的大叔们提出了一个碎片（fragment）区的概念，也就是在一个碎片区中，并不是所有的页都是为了存储同一个段的数据而存在的，而是碎片区中的页可以用于不同的目的，比如有些页用于段A，有些页用于段B，有些页甚至哪个段都不属于。碎片区直属于表空间，并不属于任何一个段。所以此后为某个段分配存储空间的策略是这样的：

- 在刚开始向表中插入数据的时候，段是从某个碎片区以单个页面为单位来分配存储空间的。
- 当某个段已经占用了32个碎片区页面之后，就会以完整的区为单位来分配存储空间。

所以现在段不能仅定义为是某些区的集合，更精确的应该是某些零散的页面以及一些完整的区的集合。除了索引的叶子节点段和非叶子节点段之外，InnoDB中还有为存储一些特殊的数据而定义的段，比如回滚段，当然我们现在并不关心别的类型的段，现在只需要知道段是一些零散的页面以及一些完整的区的集合就好了。

## 区的分类

通过上边一通唠叨，大家知道了表空间的是由若干个区组成的，这些区大体上可以分为4种类型：

- 空闲的区：现在还没有用到这个区中的任何页面。
- 有剩余空间的碎片区：表示碎片区中还有可用的页面。
- 没有剩余空间的碎片区：表示碎片区中的所有页面都被使用，没有空闲页面。
- 附属于某个段的区。每一个索引都可以分为叶子节点段和非叶子节点段，除此之外 InnoDB 还会另外定义一些特殊作用的段，在这些段中的数据量很大时将使用区来作为基本的分配单位。

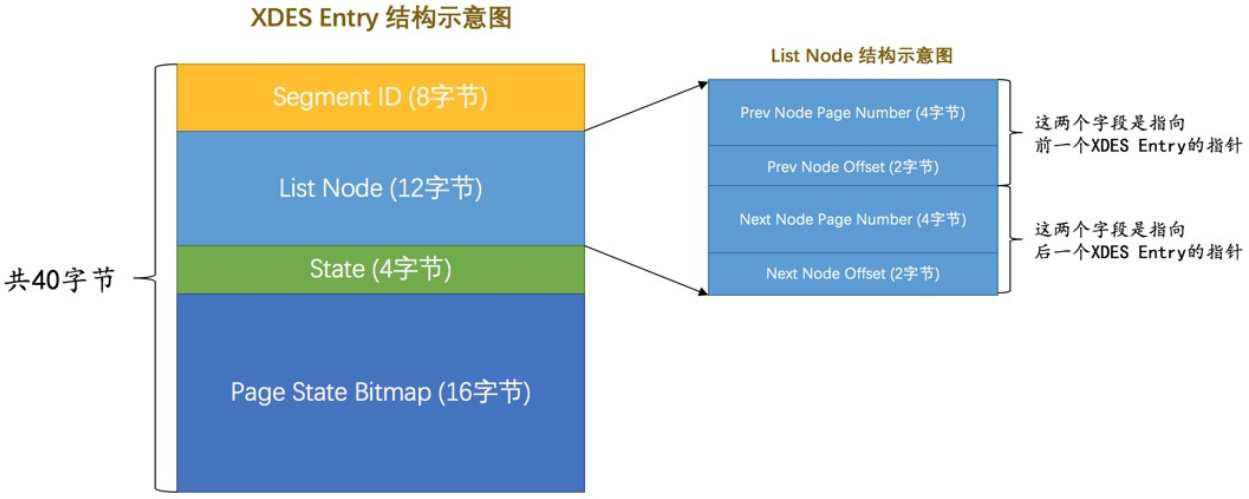
这4种类型的区也可以被称为区的4种状态（State），设计InnoDB的大叔们为这4种状态的区定义了特定的名词儿：

| 状态名 | 含义 |
| --- | --- |
| FREE | 空闲的区 |
| FREE_FRAG | 有剩余空间的碎片区 |
| FULL_FRAG | 没有剩余空间的碎片区 |
| FSEG | 附属于某个段的区 |

需要再次强调一遍的是，处于FREE、FREE_FRAG以及FULL_FRAG这三种状态的区都是独立的，算是直属于表空间；而处于FSEG状态的区是附属于某个段的。

> 小贴士：
>
> 如果把表空间比作是一个集团军，段就相当于师，区就相当于团。一般的团都是隶属于某个师的，就像是处于`FSEG`的区全都隶属于某个段，而处于`FREE`、`FREE_FRAG`以及`FULL_FRAG`这三种状态的区却直接隶属于表空间，就像独立团直接听命于军部一样。

为了方便管理这些区，设计InnoDB的大叔设计了一个称为XDES Entry的结构（全称就是Extent Descriptor Entry），每一个区都对应着一个XDES Entry结构，这个结构记录了对应的区的一些属性。我们先看图来对这个结构有个大致的了解：



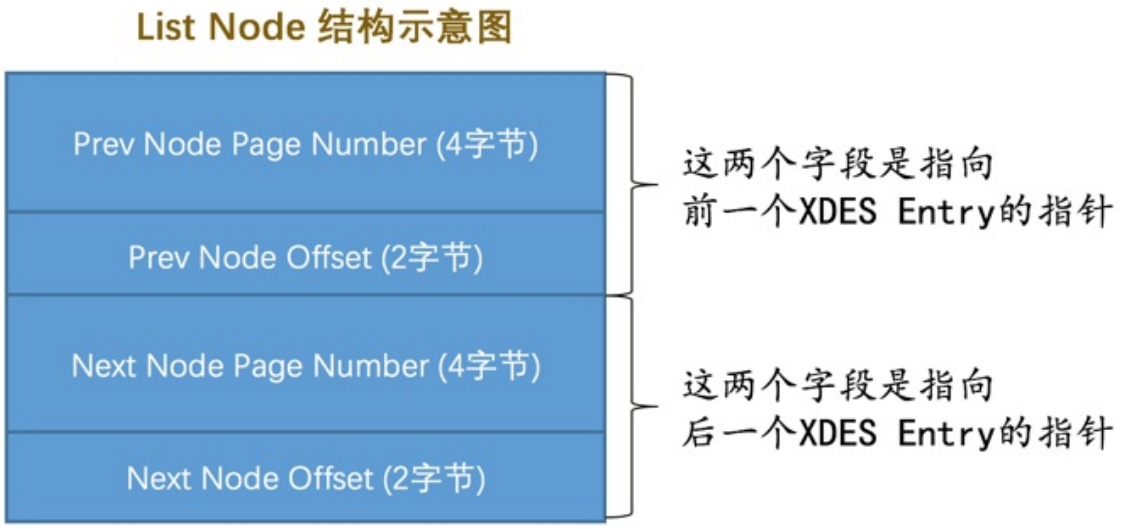

从图中我们可以看出，XDES Entry是一个40个字节的结构，大致分为4个部分，各个部分的释义如下：

- Segment ID（8字节）

  每一个段都有一个唯一的编号，用ID表示，此处的Segment ID字段表示就是该区所在的段。当然前提是该区已经被分配给某个段了，不然的话该字段的值没啥意义。

- List Node（12字节）

  这个部分可以将若干个XDES Entry结构串联成一个链表，大家看一下这个List Node的结构：




如果我们想定位表空间内的某一个位置的话，只需指定页号以及该位置在指定页号中的页内偏移量即可。所以：

- Pre Node Page Number和Pre Node Offset的组合就是指向前一个XDES Entry的指针
- Next Node Page Number和Next Node Offset的组合就是指向后一个XDES Entry的指针。

把一些XDES Entry结构连成一个链表有啥用？稍安勿躁，我们稍后唠叨XDES Entry结构组成的链表问题。

- State（4字节）

  这个字段表明区的状态。可选的值就是我们前边说过的那4个，分别是：FREE、FREE_FRAG、FULL_FRAG和FSEG。具体释义就不多唠叨了，前边说的够仔细了。

- Page State Bitmap（16字节）

  这个部分共占用16个字节，也就是128个比特位。我们说一个区默认有64个页，这128个比特位被划分为64个部分，每个部分2个比特位，对应区中的一个页。比如Page State Bitmap部分的第1和第2个比特位对应着区中的第1个页面，第

3和第4个比特位对应着区中的第2个页面，依此类推，Page State Bitmap部分的第127和128个比特位对应着区中的第64个页面。这两个比特位的第一个位表示对应的页是否是空闲的，第二个比特位还没有用。

### XDES Entry链表

到现在为止，我们已经提出了五花八门的概念，什么区、段、碎片区、附属于段的区、XDES Entry结构吧啦吧啦的概念，走远了千万别忘了自己为什么出发，我们把事情搞这么麻烦的初心仅仅是想提高向表插入数据的效率又不至于数据量少的表浪费空间。现在我们知道向表中插入数据本质上就是向表中各个索引的叶子节点段、非叶子节点段插入数据，也知道了不同的区有不同的状态，再回到最初的起点，捋一捋向某个段中插入数据的过程：

- 当段中数据较少的时候，首先会查看表空间中是否有状态为FREE_FRAG的区，也就是找还有空闲空间的碎片区，如果找到了，那么从该区中取一些零碎的页把数据插进去；否则到表空间下申请一个状态为FREE的区，也就是空闲的区，把该区的状态变为FREE_FRAG，然后从该新申请的区中取一些零碎的页把数据插进去。之后不同的段使用零碎页的时候都会从该区中取，直到该区中没有空闲空间，然后该区的状态就变成了FULL_FRAG。

  现在的问题是你怎么知道表空间里的哪些区是FREE的，哪些区的状态是FREE_FRAG的，哪些区是FULL_FRAG的？要知道表空间的大小是可以不断增大的，当增长到GB级别的时候，区的数量也就上千了，我们总不能每次都遍历这些区对应的XDES Entry结构吧？这时候就是XDES Entry中的List Node部分发挥奇效的时候了，我们可以通过List Node中的指针，做这么三件事：

    - 把状态为FREE的区对应的XDES Entry结构通过List Node来连接成一个链表，这个链表我们就称之为FREE链表。

    - 把状态为FREE_FRAG的区对应的XDES Entry结构通过List Node来连接成一个链表，这个链表我们就称之为FREE_FRAG链表。

    - 把状态为FULL_FRAG的区对应的XDES Entry结构通过List Node来连接成一个链表，这个链表我们就称之为FULL_FRAG链表。

  这样每当我们想找一个FREE_FRAG状态的区时，就直接把FREE_FRAG链表的头节点拿出来，从这个节点中取一些零碎的页来插入数据，当这个节点对应的区用完时，就修改一下这个节点的State字段的值，然后从FREE_FRAG链表中移到FULL_FRAG链表中。同理，如果FREE_FRAG链表中一个节点都没有，那么就直接从FREE链表中取一个节点移动到FREE_FRAG链表的状态，并修改该节点的STATE字段值为FREE_FRAG，然后从这个节点对应的区中获取零碎的页就好了。

- 当段中数据已经占满了32个零散的页后，就直接申请完整的区来插入数据了。

  还是那个问题，我们怎么知道哪些区属于哪个段的呢？再遍历各个XDES Entry结构？遍历是不可能遍历的，这辈子都不可能遍历的，有链表还遍历个毛线啊。所以我们把状态为FSEG的区对应的XDES Entry结构都加入到一个链表喽？傻呀，不同的段哪能共用一个区呢？你想把表a的聚簇索引的叶子节点段和表b的聚簇索引的叶子节点段都存储到一个区中么？显然我们想要每个段都有它独立的链表，所以可以根据段号（也就是Segment ID）来建立链表，有多少个段就建多少个链表？好像也有点问题，因为一个段中可以有好多个区，有的区是完

全空闲的，有的区还有一些页面可以用，有的区已经没有空闲页面可以用了，所以我们有必要继续细分，设计InnoDB的大叔们为每个段中的区对应的XDES Entry结构建立了三个链表：

- FREE链表：同一个段中，所有页面都是空闲的区对应的XDES Entry结构会被加入到这个链表。注意和直属于表空间的FREE链表区别开了，此处的FREE链表是附属于某个段的。
- NOT_FULL链表：同一个段中，仍有空闲空间的区对应的XDES Entry结构会被加入到这个链表。
- FULL链表：同一个段中，已经没有空闲空间的区对应的XDES Entry结构会被加入到这个链表。

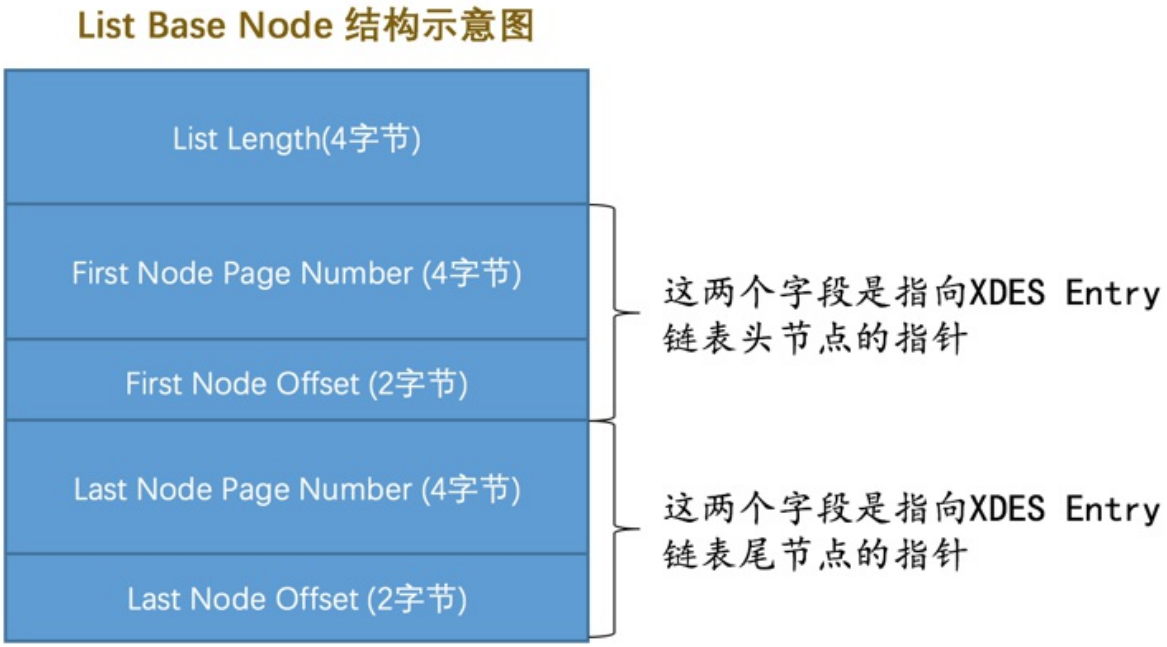
再次强调一遍，每一个索引都对应两个段，每个段都会维护上述的3个链表，比如下边这个表：

```
CREATE TABLE t (
    c1 INT NOT NULL AUTO_INCREMENT,
    c2 VARCHAR(100),
    c3 VARCHAR(100),
    PRIMARY KEY (c1),
    KEY idx_c2 (c2)
)ENGINE=InnoDB;
```

这个表t共有两个索引，一个聚簇索引，一个二级索引idx_c2，所以这个表共有4个段，每个段都会维护上述3个链表，所以这个表共需要维护12个链表。所以段在数据量比较大时插入数据的话，会先获取NOT_FULL链表的头节点，直接把数据插入这个头节点对应的区中即可，如果该区的空间已经被用完，就把该节点移到FULL链表中。

**链表基节点**

上边光是介绍了一堆链表，可我们怎么找到这些链表呢，或者说怎么找到某个链表的头节点或者尾节点在表空间中的位置呢？设计InnoDB的大叔当然考虑了这个问题，他们设计了一个叫List Base Node的结构，翻译成中文就是链表的基节点。这个结构中包含了链表的头节点和尾节点的指针以及这个链表中包含了多少节点的信息，我们画图看一下这个结构的示意图：




我们上边介绍的每个链表都对应这么一个List Base Node结构，其中：

- List Length表明该链表一共有多少节点，
- First Node Page Number和First Node Offset表明该链表的头节点在表空间中的位置。
- Last Node Page Number和Last Node Offset表明该链表的尾节点在表空间中的位置。

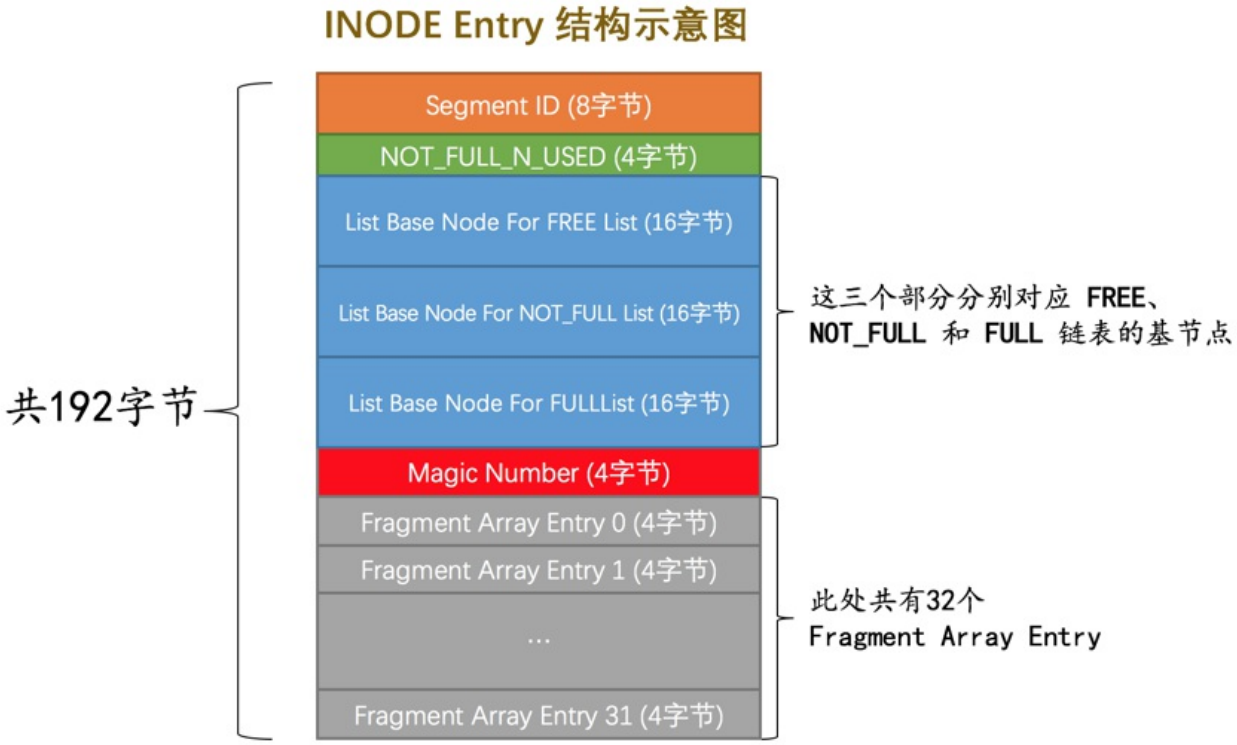
一般我们把某个链表对应的List Base Node结构放置在表空间中固定的位置，这样想找定位某个链表就变得so easy啦。

**链表小结**

综上所述，表空间是由若干个区组成的，每个区都对应一个XDES Entry的结构，直属于表空间的区对应的XDES Entry结构可以分成FREE、FREE_FRAG和FULL_FRAG这3个链表；每个段可以附属若干个区，每个段中的区对应的XDES Entry结构可以分成FREE、NOT_FULL和FULL这3个链表。每个链表都对应一个List Base Node的结构，这个结构里记录了链表的头、尾节点的位置以及该链表中包含的节点数。正是因为这些链表的存在，管理这些区才变成了一件so easy的事情。

## 段的结构

我们前边说过，段其实不对应表空间中某一个连续的物理区域，而是一个逻辑上的概念，由若干个零散的页面以及一些完整的区组成。像每个区都有对应的XDES Entry来记录这个区中的属性一样，设计InnoDB的大叔为每个段都定义了一个INODE Entry结构来记录一下段中的属性。大家看一下示意图：




它的各个部分释义如下：

- Segment ID

  就是指这个INODE Entry结构对应的段的编号（ID）。

- NOT_FULL_N_USED

  这个字段指的是在NOT_FULL链表各XDES Entry节点对应的区已经使用了多少页面。一个区中有64个页面，如果不标记已经使用了多少页面的话，每次向段中插入数据的时候都要从第一个页面进行遍历寻找空闲页面，有了这个字段之后就可以快速定位空闲页面。

- 3个List Base Node

  分别为段的FREE链表、NOT_FULL链表、FULL链表定义了List Base Node，这样我们想查找某个段的某个链表的头节点和尾节点的时候，就可以直接到这个部分找到对应链表的List Base Node。so easy!

- Magic Number：

  这个值是用来标记这个INODE Entry是否已经被初始化了（初始化的意思就是把各个字段的值都填进去了）。如果这个数字是值的97937874，表明该INODE Entry已经初始化，否则没有被初始化。（不用纠结这个值有啥特殊含义，人家规定的）。

- Fragment Array Entry

  我们前边强调过无数次段是一些零散页面和一些完整的区的集合，每个Fragment Array Entry结构都对应着一个零散的页面，这个结构一共4个字节，表示一个零散页面的页号。

结合着这个INODE Entry结构，大家可能对段是一些零散页面和一些完整的区的集合的理解再次深刻一些。

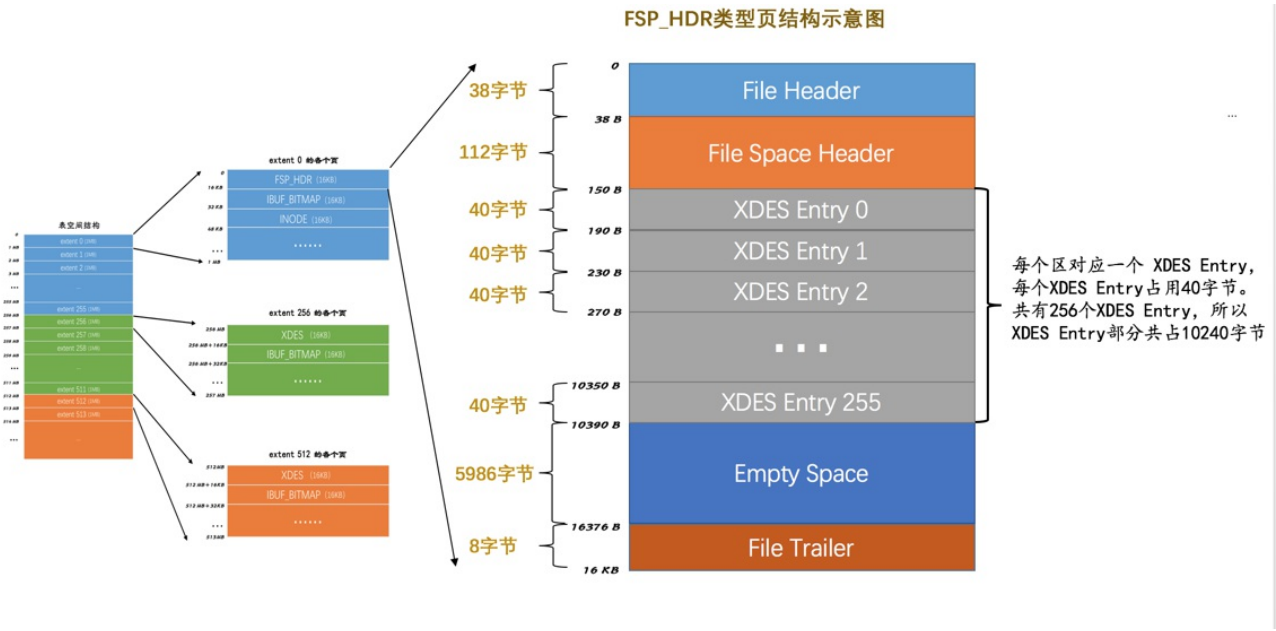
## 各类型页面详细情况

到现在为止我们已经大概清楚了表空间、段、区、XDES Entry、INODE Entry、各种以XDES Enty为节点的链表的基本概念了，可是总有一种飞在天上不踏实的感觉，每个区对应的XDES Entry结构到底存储在表空间的什么地方？直属于表空间的FREE、FREE_FRAG、FULL_FRAG链表的基节点到底存储在表空间的什么地方？每个段对应的INODE Entry结构到底存在表空间的什么地方？我们前边介绍了每256个连续的区算是一个组，想解决刚才提出来的这些个疑问还得从每个组开头的一些类型相同的页面说起，接下来我们一个页面一个页面的分析，真相马上就要浮出水面了。

### FSP_HDR类型

首先看第一个组的第一个页面，当然也是表空间的第一个页面，页号为0。这个页面的类型是FSP_HDR，它存储了表空间的一些整体属性以及第一个组内256个区的对应的XDES Entry结构，直接看这个类

型的页面的示意图：



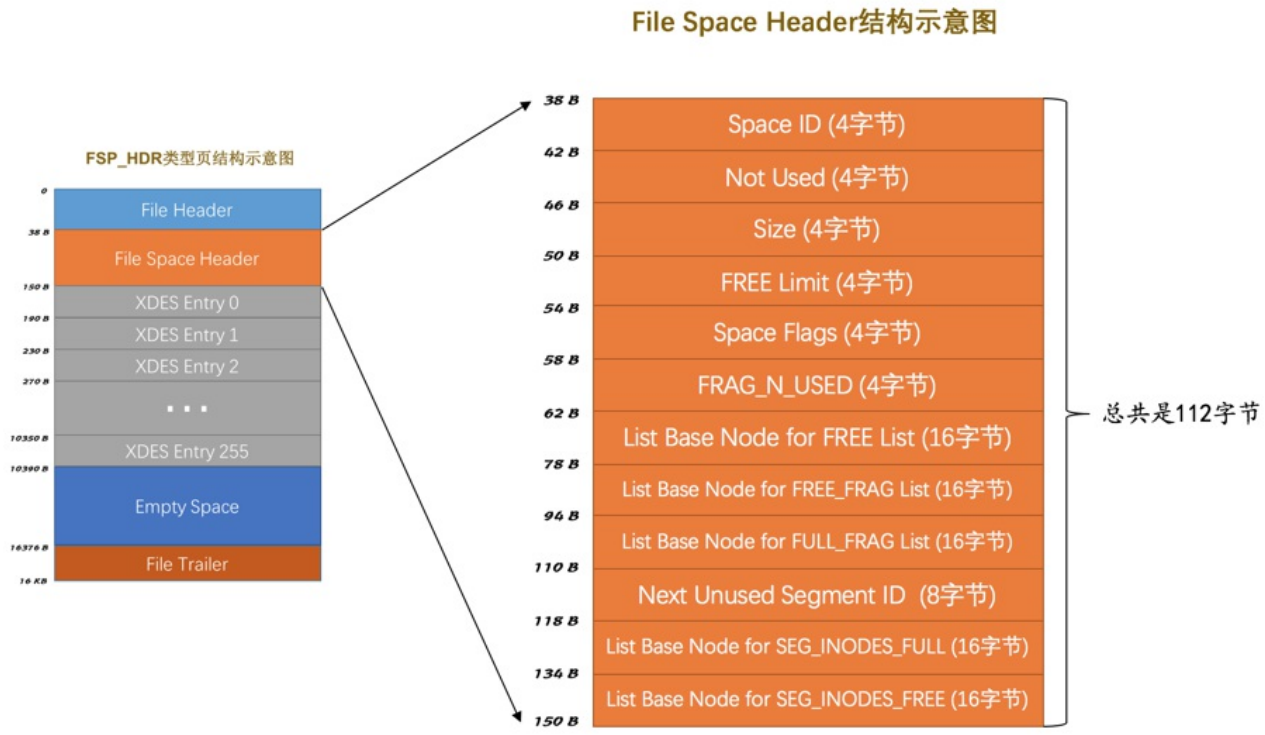

从图中可以看出，一个完整的FSP_HDR类型的页面大致由5个部分组成，各个部分的具体释义如下表：

| 名称 | 中文名 | 占用空间大小 | 简单描述 |
| --- | --- | --- | --- |
| File Header | 文件头部 | 38字节 | 页的一些通用信息 |
| File Space Header | 表空间头部 | 112字节 | 表空间的一些整体属性信息 |
| XDES Entry | 区描述信息 | 10240字节 | 存储本组256个区对应的属性信息 |
| Empty Space | 尚未使用空间 | 5986字节 | 用于页结构的填充，没啥实际意义 |
| File Trailer | 文件尾部 | 8字节 | 校验页是否完整 |

File Header和File Trailer就不再强调了，另外的几个部分中，Empty Space是尚未使用的空间，我们不用管它，重点来看看File Space Header和XDES Entry这两个部分。

**File Space Header部分**

从名字就可以看出来，这个部分是用来存储表空间的一些整体属性的，废话少说，看图：




哇唔，字段有点儿多哦，不急一个一个慢慢看。下面是各个属性的简单描述：

| 名称 | 占用空间大小 | 描述 |
| --- | --- | --- |
| Space ID | 4字节 | 表空间的ID |
| Not Used | 4字节 | 这4个字节未被使用，可以忽略 |
| Size | 4字节 | 当前表空间占有的页面数 |
| FREE Limit | 4字节 | 尚未被初始化的最小页号，大于或等于这个页号的区对应的XDES Entry结构都没有被加入FREE链表 |
| | 4字 | 表空间的一些占用存储空间比较小的属 |

| Space Flags | 节 | 性 |
| --- | --- | --- |
| FRAG_N_USED | 4字节 | FREE_FRAG链表中已使用的页面数量 |
| List Base Node for FREE List | 16字节 | FREE链表的基节点 |
| List Base Node for FREE_FRAG List | 16字节 | FREE_FREG链表的基节点 |
| List Base Node for FULL_FRAG List | 16字节 | FULL_FREG链表的基节点 |
| Next Unused Segment ID | 8字节 | 当前表空间中下一个未使用的 Segment ID |
| List Base Node for SEG_INODES_FULL List | 16字节 | SEG_INODES_FULL链表的基节点 |
| List Base Node for SEG_INODES_FREE List | 16字节 | SEG_INODES_FREE链表的基节点 |

这里头的Space ID、Not Used、Size这三个字段大家肯定一看就懂，其他的字段我们再详细瞅瞅，为了大家的阅读体验，我就不严格按照实际的字段顺序来解释各个字段了哈。

- List Base Node for FREE List、List Base Node for FREE_FRAG List、List Base Node for FULL_FRAG List。

这三个大家看着太亲切了，分别是直属于表空间的FREE链表的基节点、FREE_FRAG链表的基节点、FULL_FRAG链表的基节点，这三个链表的基节点在表空间的位置是固定的，就是在表空间的第一个页面（也就是FSP_HDR类型的页面）的File Space Header部分。所以之后定位这几个链表就so easy啦。

- FRAG_N_USED

  这个字段表明在FREE_FRAG链表中已经使用的页面数量，方便之后在链表中查找空闲的页面。

- FREE Limit

  我们知道表空间都对应着具体的磁盘文件，一开始我们创建表空间的时候对应的磁盘文件中都没有数据，所以我们需要对表空间完成一个初始化操作，包括为表空间中的区建立XDES Entry结构，为各个段建立INODE Entry结构，建立各种链表吧啦吧啦的各种操作。我们可以一开始就为表空间申请一个特别大的空间，但是实际上有绝大部分的区是空闲的，我们可以选择把所有的这些空闲区对应的XDES Entry结构加入FREE链表，也可以选择只把一部分的空闲区加入FREE链表，等啥时候空闲链表中的XDES Entry结构对应的区不够使了，再把之前没有加入FREE链表的空闲区对应的XDES Entry结构加入FREE链表，中心思想就是啥时候用到啥时候初始化，设计InnoDB的大叔采用的就是后者，他们为表空间定义了FREE Limit这个字段，在该字段表示的页号之前的区都被初始化了，之后的区尚未被初始化。

- Next Unused Segment ID

  表中每个索引都对应2个段，每个段都有一个唯一的ID，那当我们为某个表新创建一个索引的时候，就意味着要创建两个新的段。那怎么为这个新创建的段找一个唯一的ID呢？去遍历现

在表空间中所有的段么？我们说过，遍历是不可能遍历的，这辈子都不可能遍历，所以设计InnoDB的大叔们提出了这个名叫Next Unused Segment ID的字段，该字段表明当前表空间中最大的段ID的下一个ID，这样在创建新段的时候赋予新段一个唯一的ID值就so easy啦，直接使用这个字段的值就好了。

- Space Flags

表空间对于一些布尔类型的属性，或者只需要寥寥几个比特位搞定的属性都放在了这个Space Flags中存储，虽然它只有4个字节，32个比特位大小，却存储了好多表空间的属性，详细情况如下表：

| 标志名称 | 占用的空间（单位：bit） | 描述 |
| --- | --- | --- |
| POST_ANTELOPE | 1 | 表示文件格式是否大于ANTELOPE |
| ZIP_SSIZE | 4 | 表示压缩页面的大小 |
| ATOMIC_BLOBS | 1 | 表示是否自动把值非常长的字段放到BLOB页里 |
| PAGE_SSIZE | 4 | 页面大小 |
| DATA_DIR | 1 | 表示表空间是否是从默认的数据目录中获取的 |
| SHARED | 1 | 是否为共享表空间 |
| TEMPORARY | 1 | 是否为临时表空间 |
| ENCRYPTION | 1 | 表空间是否加密 |
| UNUSED | 18 | 没有使用到的比特位 |

> 小贴士：
>
> 不同MySQL版本里 SPACE_FLAGS 代表的属性可能有些差异，我们这里列举的是5.7.21版本的。不过大家现在不必深究它们的意思，因为我们一旦把这些概念展开，就需要非常大的篇幅，主要怕大家受不了。我们还是先挑重要的看，把主要的表空间结构了解完，这些 SPACE_FLAGS 里的属性的细节就暂时不深究了。

- List Base Node for SEG_INODES_FULL List和List Base Node for SEG_INODES_FREE List

  每个段对应的INODE Entry结构会集中存放到一个类型位INODE的页中，如果表空间中的段特别多，则会有多个INODE Entry结构，可能一个页放不下，这些INODE类型的页会组成两种列表：

  - SEG_INODES_FULL链表，该链表中的INODE类型的页面都已经被INODE Entry结构填充满了，没空闲空间存放额外的INODE Entry了。
  - SEG_INODES_FULL链表，该链表中的INODE类型的页面都已经仍有空闲空间来存放INODE Entry结构。

  由于我们现在还没有详细唠叨INODE类型页，所以等会说过INODE类型的页之后再回过头来看着两个链表。

#### XDES Entry部分

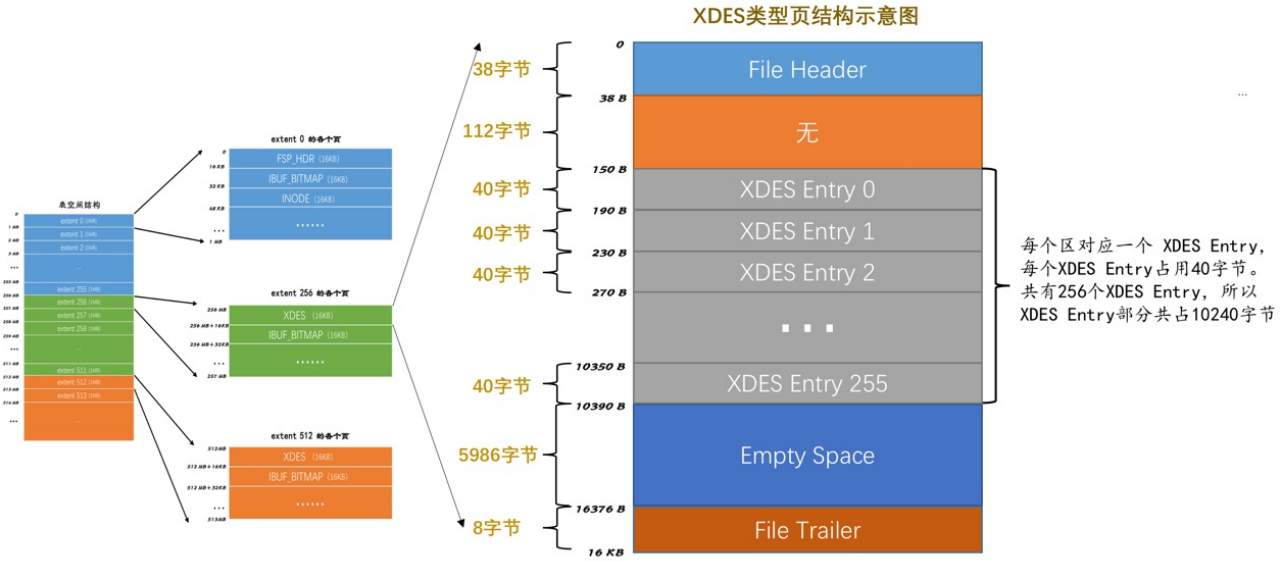
紧接着File Space Header部分的就是XDES Entry部分了，我们嘴上唠叨过无数次，却从没见过真身的XDES Entry就是在表空间的第一个页面中保存的。我们知道一个XDES Entry结构的大小是40字节，但是一个页面的大小有限，只能存放有限个XDES Entry结构，所以我们才把256个区划分成一组，在每组的第一个页面中存

放256个XDES Entry结构。大家回看那个FSP_HDR类型页面的示意图，XDES Entry 0就对应着extent 0，XDES Entry 1就对应着extent 1...依此类推，XDES Entry255就对应着extent 255。

因为每个区对应的XDES Entry结构的地址是固定的，所以我们访问这些结构就so easy啦，至于该结构的详细使用情况我们已经唠叨的够明白了，在这就不赘述了。

### XDES类型

我们说过，每一个XDES Entry结构对应表空间的一个区，虽然一个XDES Entry结构只占用40字节，但你抵不住表空间的区的数量也多啊。在区的数量非常多时，一个单独的页可能就不够存放足够多的XDES Entry结构，所以我们把表空间的区分为了若干个组，每组开头的一个页面记录着本组内所有的区对应的XDES Entry结构。由于第一个组的第一个页面有些特殊，因为它也是整个表空间的第一个页面，所以除了记录本组中的所有区对应的XDES Entry结构以外，还记录着表空间的一些整体属性，这个页面的类型就是我们刚刚说完的FSP_HDR类型，整个表空间里只有一个这个类型的页面。除去第一个分组以外，之后的每个分组的第一个页面只需要记录本组内所有的区对应的XDES Entry结构即可，不需要再记录表空间的属性了，为了和FSP_HDR类型做区别，我们把之后每个分组的第一个页面的类型定义为XDES，它的结构和FSP_HDR类型是非常相似的：



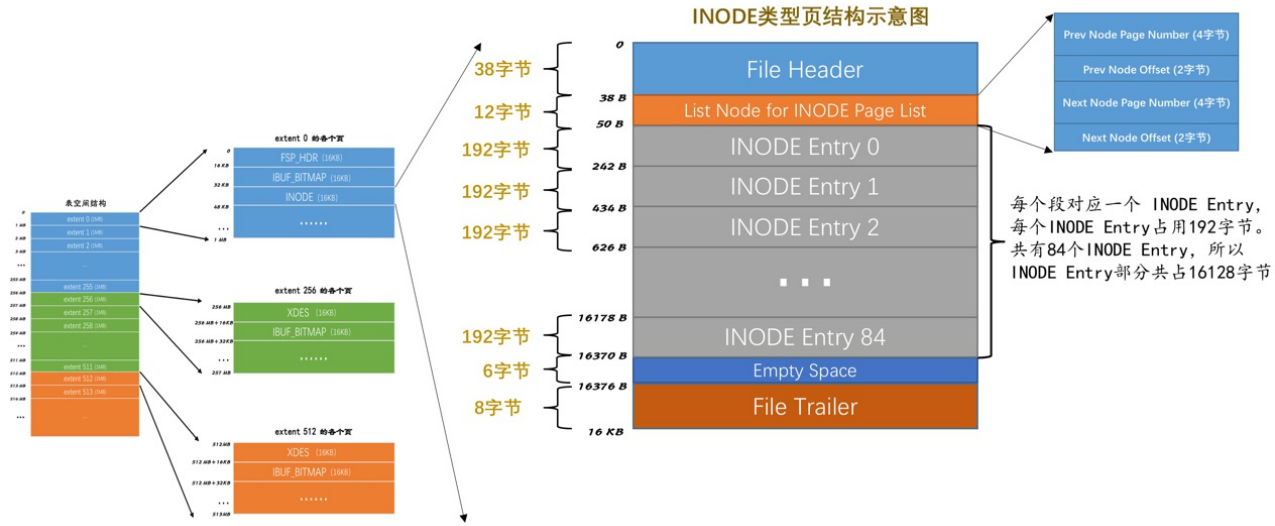

与FSP_HDR类型的页面对比，除了少了File Space Header部分之外，也就是除了少了记录表空间整体属性的部分之外，其余的部分是一样一样的。由于我们上边唠叨的已经够仔细了，对于XDES类型的页面也就不重复唠叨了哈。

### IBUF_BITMAP类型

对比前边介绍表空间的图，每个分组的第二个页面的类型都是IBUF_BITMAP，这种类型的页里边记录了一些有关Change Buffer的东东，由于这个Change Buffer里又包含了贼多的概念，考虑到大家在一章中接受这么多新概念有点呼吸不适，怕大家心脏病犯了所以就把Change Buffer的相关知识放到后边的章节中，大家稍安勿躁哈。

### INODE类型

再次对比前边介绍表空间的图，第一个分组的第三个页面的类型是INODE。我们前边说过设计InnoDB的大叔为每个索引定义了两个段，而且为某些特殊功能定义了些特殊的段。为了方便管理，他们又为每个段设计了一个INODE Entry结构，这个结构中记录了关于这个段的相关属性。而我们这会儿要介绍的这个INODE类型的页就是为了存储INODE Entry结构而存在的。好了，废话少说，直接看图：




从图中可以看出，一个INODE类型的页面是由这几部分构成的：

| 名称 | 中文名 | 占用空间大小 | 简单描述 |
| --- | --- | --- | --- |
| File Header | 文件头部 | 38字节 | 页的一些通用信息 |
| List Node for INODE Page List | 通用链表节点 | 112字节 | 存储上一个INODE页面和下一个INODE页面的指针 |
| INODE Entry | 段描述信息 | 10240字节 | |
| Empty Space | 尚未使用空间 | 6字节 | 用于页结构的填充，没啥实际意义 |
| File Trailer | 文件尾部 | 8字节 | 校验页是否完整 |

除了File Header、Empty Space、File Trailer这几个老朋友外，我们重点关注List Node for INODE Page List和INODE Entry这两个部分。

首先看INODE Entry部分，我们前边已经详细介绍过这个结构的组成了，主要包括对应的段内零散页面的地址以及附属于该段的FREE、NOT_FULL和FULL链表的基节点。每个INODE Entry结构

占用192字节，一个页面里可以存储84个这样的结构。

重点看一下List Node for INODE Page List这个玩意儿，因为一个表空间中可能存在超过84个段，所以可能一个INODE类型的页面不足以存储所有的段对应的INODE Entry结构，所以就需要额外的INODE类型的页面来存储这些结构。还是为了方便管理这些INODE类型的页面，设计InnoDB的大叔们将这些INODE类型的页面串联成两个不同的链表：

- SEG_INODES_FULL链表：该链表中的INODE类型的页面中已经没有空闲空间来存储额外的INODE Entry结构了。
- SEG_INODES_FREE链表：该链表中的INODE类型的页面中还有空闲空间来存储额外的INODE Entry结构了。

想必大家已经认出这两个链表了，我们前边提到过这两个链表的基节点就存储在File Space Header里边，也就是说这两个链表的基节点的位置是固定的，所以我们可以很轻松的访问到这两个链表。以后每当我们新创建一个段（创建索引时就会创建段）时，都会创建一个INODE Entry结构与之对应，存储INODE Entry的大致过程就是这样的：

- 先看看SEG_INODES_FREE链表是否为空，如果不为空，直接从该链表中获取一个节点，也就相当于获取到一个仍有空闲空间的INODE类型的页面，然后把该INODE Entry结构防到该页面中。当该页面中无剩余空间时，就把该页放到SEG_INODES_FULL链表中。
- 如果SEG_INODES_FREE链表为空，则需要从表空间的FREE_FRAG链表中申请一个页面，修改该页面的类型为INODE，把该页面放到SEG_INODES_FREE链表中，与此同时把该INODE Entry结构放入该页面。

## Segment Header 结构的运用

我们知道一个索引会产生两个段，分别是叶子节点段和非叶子节点段，而每个段都会对应一个INODE Entry结构，那我们怎么知道某个段对应哪个INODE Entry结构呢？所以得找个地方记下来这个对应关系。希望你还记得我们在唠叨数据页，也就是INDEX类型的页时有一个Page Header部分，当然我不能指望你记住，所以把Page Header部分再抄一遍给你看：

**Page Header部分**（为突出重点，省略了好多属性）

| 名称 | 占用空间大小 | 描述 |
| --- | --- | --- |
| ... | ... | ... |
| PAGE_BTR_SEG_LEAF | 10字节 | B+树叶子段的头部信息，仅在B+树的根页定义 |
| PAGE_BTR_SEG_TOP | 10字节 | B+树非叶子段的头部信息，仅在B+树的根页定义 |

其中的PAGE_BTR_SEG_LEAF和PAGE_BTR_SEG_TOP都占用10个字节，它们其实对应一个叫Segment Header的结构，该结构图示如下：

各个部分的具体释义如下：

| 名称 | 占用字节数 | 描述 |
| --- | --- | --- |
| Space ID of the INODE Entry | 4 | INODE Entry结构所在的表空间ID |
| Page Number of the INODE Entry | 4 | INODE Entry结构所在的页面页号 |
| Byte Offset of the INODE Ent | 2 | INODE Entry结构在该页面中的偏移量 |

这样子就很清晰了，PAGE_BTR_SEG_LEAF记录着叶子节点段对应的INODE Entry结构的地址是哪个表空间的哪个页面的哪个偏移量，PAGE_BTR_SEG_TOP记录着非叶子节点段对应的INODE Entry结构的地址是哪个表空间的哪个页面的哪个偏移量。这样子索引和其对应的段的关系就建立起来了。不过需要注意的一点是，因为一个索引只对应两个段，所以只需要在索引的根页面中记录这两个结构即可。

### 真实表空间对应的文件大小

等会儿等会儿，上边的这些概念已经压的快喘不过气了。不过独立表空间有那么大么？我到数据目录里看了，一个新建的表对应的.ibd文件只占用了96K，才6个页面大小，上边的内容该不是扯犊子吧？

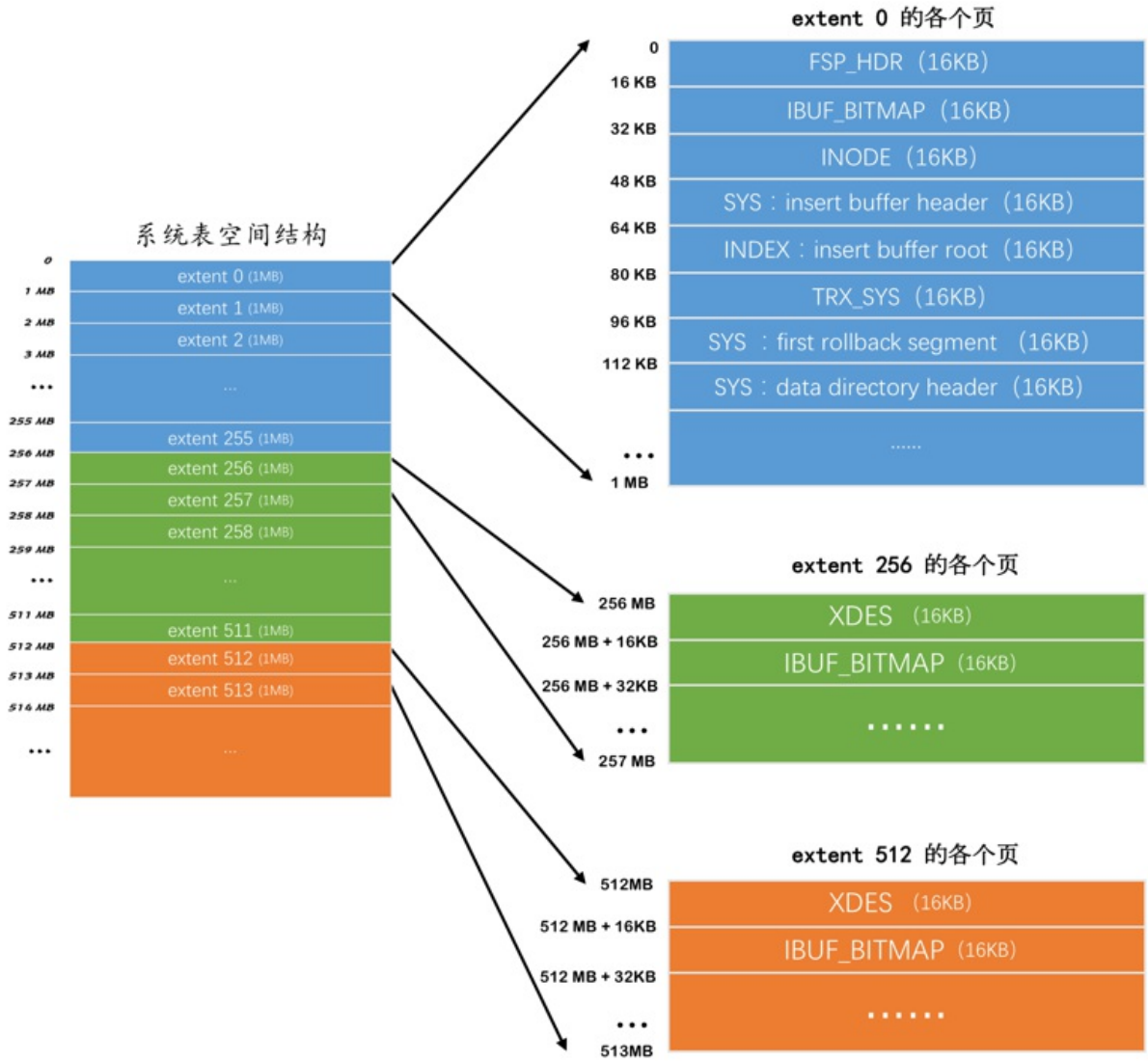
哈，一开始表空间占用的空间自然是很小，因为表里边都没有数据嘛！不过别忘了这些.ibd文件是自扩展的，随着表中数据的增多，表空间对应的文件也逐渐增大。

## 系统表空间

了解完了独立表空间的基本结构，系统表空间的结构也就好理解多了，系统表空间的结构和独立表空间基本类似，只不过由于整个MySQL进程只有一个系统表空间，在系统表空间中会额外记录一些有关整个系统信息的页面，所以会比独立表空间多出一些记录这些信息的页面。因为这个系统表空间最牛逼，相当于是表空间之首，所以它的表空间 ID（Space ID）是0。

## 系统表空间的整体结构

系统表空间与独立表空间的一个非常明显的不同之处就是在表空间开头有许多记录整个系统属性的页面，如图：



可以看到，系统表空间和独立表空间的前三个页面（页号分别为0、1、2，类型分别是FSP_HDR、IBUF_BITMAP、INODE）的类型是一致的，只是页号为3～7的页面是系统表空间特有的，我们来看一下这些多出来的页面都是干啥使的：

| 页号 | 页面类型 | 英文描述 | 描述 |
| --- | --- | --- | --- |
| 3 | SYS | Insert Buffer Header | 存储Insert Buffer的头部信息 |
| 4 | INDEX | Insert Buffer Root | 存储Insert Buffer的根页面 |
| 5 | TRX_SYS | Transction System Header | 事务系统的相关信息 |
| 6 | SYS | First Rollback Segment | 第一个回滚段的页面 |
| 7 | SYS | Data Dictionary Header | 数据字典头部信息 |

除了这几个记录系统属性的页面之外，系统表空间的extent 1和extent 2这两个区，也就是页号从64~127这128个页面被称为Doublewrite buffer，也就是双写缓冲区。不过上述的大部分知识都涉及到了事务和多版本控制的问题，这些问题我们会放在后边的章节集中唠叨，现在讲述太影响用户体验，所以现在我们只唠叨一下有关InnoDB数据字典的知识，其余的概念在后边再看。

**InnoDB数据字典**

我们平时使用INSERT语句向表中插入的那些记录称之为用户数据，MySQL只是作为一个软件来为我们来保管这些数据，提供方便的增删改查接口而已。但是每当我们向一个表中插入一条记录的时候，MySQL先要校验一下插入语句对应的表存不存在，插入的列和表中的列是否符合，如果语法没有问题的话，还需要知道该表的聚簇索引和所有二级索引对应的根页面是哪个表空间的哪个页面，然后把记录插入对应索引的B+树中。所以说，MySQL除了保存着我们插入的用户数据之外，还需要保存许多额外的信息，比方说：

- 某个表属于哪个表空间，表里边有多少列
- 表对应的每一个列的类型是什么
- 该表有多少索引，每个索引对应哪几个字段，该索引对应的根页面在哪个表空间的哪个页面
- 该表有哪些外键，外键对应哪个表的哪些列
- 某个表空间对应文件系统上文件路径是什么
- balabala ... 还有好多，不一一列举了

上述这些数据并不是我们使用INSERT语句插入的用户数据，实际上是为了更好的管理我们这些用户数据而不得已引入的一些额外数据，这些数据也称为元数据。InnoDB存储引擎特意定义了一些列的内部系统表（internal system table）来记录这些这些元数据：

| 表名 | 描述 |
| --- | --- |
| SYS_TABLES | 整个InnoDB存储引擎中所有的表的信息 |
| SYS_COLUMNS | 整个InnoDB存储引擎中所有的列的信息 |
| SYS_INDEXES | 整个InnoDB存储引擎中所有的索引的信息 |
| SYS_FIELDS | 整个InnoDB存储引擎中所有的索引对应的列的信息 |
| SYS_FOREIGN | 整个InnoDB存储引擎中所有的外键的信息 |
| SYS_FOREIGN_COLS | 整个InnoDB存储引擎中所有的外键对应列的信息 |
| SYS_TABLESPACES | 整个InnoDB存储引擎中所有的表空间信息 |
| SYS_DATAFILES | 整个InnoDB存储引擎中所有的表空间对应文件系统的文件路径信息 |
| SYS_VIRTUAL | 整个InnoDB存储引擎中所有的虚拟生成列的信息 |

这些系统表也被称为数据字典，它们都是以B+树的形式保存在系统表空间的某些页面中，其中SYS_TABLES、SYS_COLUMNS、SYS_INDEXES、SYS_FIELDS这四个表尤其重要，称之为基本系统表（basic system tables），我们先看看这4个表的结构：

#### SYS_TABLES表

**SYS_TABLES表的列**

| 列名 | 描述 |
| --- | --- |
| NAME | 表的名称 |
| ID | InnoDB存储引擎中每个表都有一个唯一的ID |
| N_COLS | 该表拥有列的个数 |
| TYPE | 表的类型，记录了一些文件格式、行格式、压缩等信息 |
| MIX_ID | 已过时，忽略 |
| MIX_LEN | 表的一些额外的属性 |
| CLUSTER_ID | 未使用，忽略 |
| SPACE | 该表所属表空间的ID |

这个SYS_TABLES表有两个索引：

- 以NAME列为主键的聚簇索引
- 以ID列建立的二级索引

#### SYS_COLUMNS表

**SYS_COLUMNS表的列**

| 列名 | 描述 |
| --- | --- |
| TABLE_ID | 该列所属表对应的ID |

| | |
|---|---|
| POS | 该列在表中是第几列 |
| NAME | 该列的名称 |
| MTYPE | main data type，主数据类型，就是那堆INT、CHAR、VARCHAR、FLOAT、DOUBLE之类的东东 |
| PRTYPE | precise type，精确数据类型，就是修饰主数据类型的那堆东东，比如是否允许NULL值，是否允许负数啥的 |
| LEN | 该列最多占用存储空间的字节数 |
| PREC | 该列的精度，不过这列貌似都没有使用，默认值都是0 |

这个SYS_COLUMNS表只有一个聚集索引：

- 以(TABLE_ID, POS)列为主键的聚簇索引

#### SYS_INDEXES表

**SYS_INDEXES表的列**

| 列名 | 描述 |
|---|---|
| TABLE_ID | 该索引所属表对应的ID |
| ID | InnoDB存储引擎中每个索引都有一个唯一的ID |
| NAME | 该索引的名称 |
| N_FIELDS | 该索引包含列的个数 |
| TYPE | 该索引的类型，比如聚簇索引、唯一索引、更改缓冲区的索引、全文索引、普通的二级索引等等各种类型 |
| SPACE | 该列最多占用存储空间的字节数 |
| PAGE_NO | 该列的精度，不过这列貌似都没有使用，默认值都是0 |
| MERGE_THRESHOLD | 如果页面中的记录被删除到某个比例，就把该页面和相邻页面合并，这个值就是这个比例 |

这个SYS_INEXES表只有一个聚集索引：

- 以(TABLE_ID, ID)列为主键的聚簇索引

#### SYS_FIELDS表

**SYS_FIELDS表的列**

| 列名 | 描述 |
| --- | --- |
| INDEX_ID | 该索引列所属的索引的ID |
| POS | 该索引列在某个索引中是第几列 |
| COL_NAME | 该索引列的名称 |

这个SYS_INEXES表只有一个聚集索引:

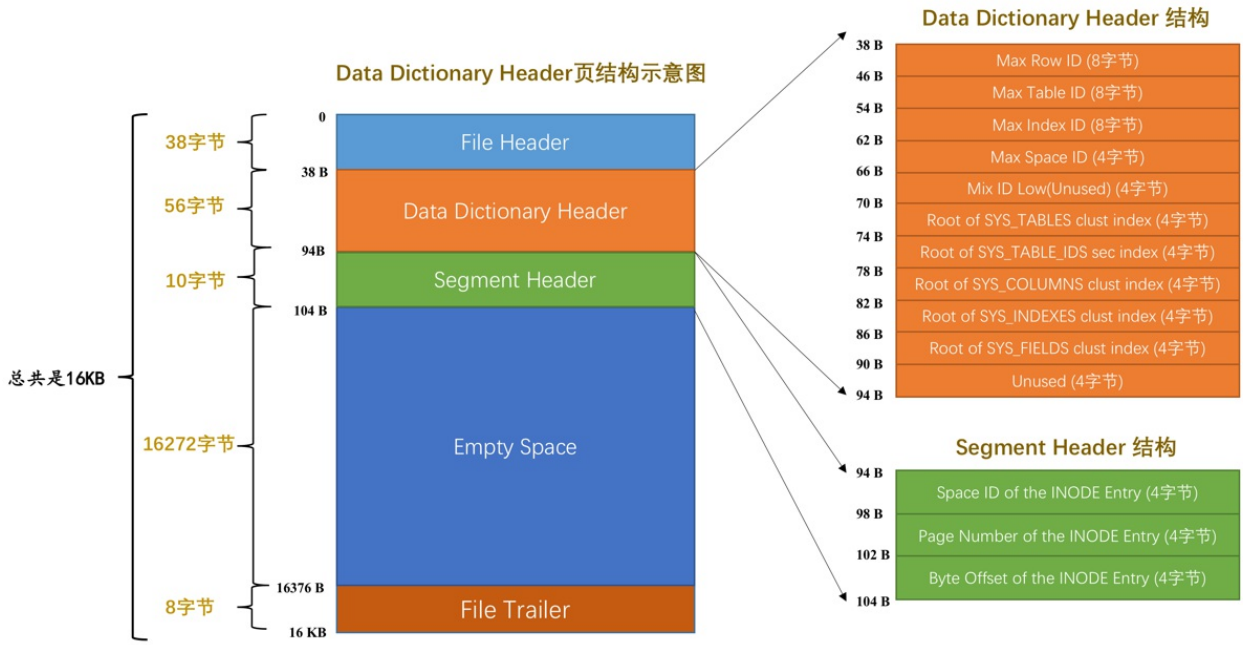
- 以(INDEX_ID, POS)列为主键的聚簇索引

#### Data Dictionary Header页面

只要有了上述4个基本系统表，也就意味着可以获取其他系统表以及用户定义的表的所有元数据。比方说我们想看看SYS_TABLESPACES这个系统表里存储了哪些表空间以及表空间对应的属性，那就可以:

- 到SYS_TABLES表中根据表名定位到具体的记录，就可以获取到SYS_TABLESPACES表的TABLE_ID
- 使用这个TABLE_ID到SYS_COLUMNS表中就可以获取到属于该表的所有列的信息。
- 使用这个TABLE_ID还可以到SYS_INDEXES表中获取所有的索引的信息，索引的信息中包括对应的INDEX_ID，还记录着该索引对应的B+数根页面是哪个表空间的哪个页面。
- 使用INDEX_ID就可以到SYS_FIELDS表中获取所有索引列的信息。

也就是说这4个表是表中之表，那这4个表的元数据去哪里获取呢？没法搞了，只能把这4个表的元数据，就是它们有哪些列、哪些索引等信息硬编码到代码中，然后设计InnoDB的大叔又拿出一个固定的页面来记录这4个表的聚簇索引和二级索引对应的B+树位置，这个页面就是页号为7的页面，类型为SYS，记录了Data Dictionary Header，也就是数据字典的头部信息。除了这4个表的5个索引的根页面信息外，这个页号为7的页面还记录了整个InnoDB存储引擎的一些全局属性，说话太啰嗦，直接看这个页面的示意图：




可以看到这个页面由下边几个部分组成：

| 名称 | 中文名 | 占用空间大小 | 简单描述 |
| --- | --- | --- | --- |
| File Header | 文件头部 | 38字节 | 页的一些通用信息 |
| Data Dictionary Header | 数据字典头部信息 | 56字节 | 记录一些基本系统表的根页面位置以及InnoDB存储引擎的一些全局信息 |
| Segment Header | 段头部信息 | 10字节 | 记录本页面所在段对应的INODE Entry位置信息 |
| Empty Space | 尚未使用 | 16272字 | 用于页结构的填充，没啥实际意 |

| | 空间 | 节 | 义 |
| --- | --- | --- | --- |
| File Trailer | 文件尾部 | 8字节 | 校验页是否完整 |

可以看到这个页面里竟然有Segment Header部分，意味着设计InnoDB的大叔把这些有关数据字典的信息当成一个段来分配存储空间，我们就姑且称之为数据字典段吧。由于目前我们需要记录的数据字典信息非常少（可以看到Data Dictionary Header部分仅占用了56字节），所以该段只有一个碎片页，也就是页号为7的这个页。

接下来我们需要细细唠叨一下Data Dictionary Header部分的各个字段：

- Max Row ID：我们说过如果我们不显式的为表定义主键，而且表中也没有UNIQUE索引，那么InnoDB存储引擎会默认为我们生成一个名为row_id的列作为主键。因为它是主键，所以每条记录的row_id列的值不能重复。原则上只要一个表中的row_id列不重复就可以了，也就是说表a和表b拥有一样的row_id列也没啥关系，不过设计InnoDB的大叔只提供了这个Max Row ID字段，不论哪个拥有row_id列的表插入一条记录时，该记录的row_id列的值就是Max Row ID对应的值，然后再把Max Row ID对应的值加1，也就是说这个Max Row ID是全局共享的。

- Max Table ID：InnoDB存储引擎中的所有的表都对应一个唯一的ID，每次新建一个表时，就会把本字段的值作为该表的ID，然后自增本字段的值。

- Max Index ID：InnoDB存储引擎中的所有的索引都对应一个唯一的ID，每次新建一个索引时，就会把本字段的值作为该索引的ID，然后自增本字段的值。

- Max Space ID：InnoDB存储引擎中的所有的表空间都对应一个唯一的ID，每次新建一个表空间时，就会把本字段的值作为该表空间的ID，然后自增本字段的值。
- Mix ID Low(Unused)：这个字段没啥用，跳过。
- Root of SYS_TABLES clust index：本字段代表SYS_TABLES表聚簇索引的根页面的页号。
- Root of SYS_TABLE_IDS sec index：本字段代表SYS_TABLES表为ID列建立的二级索引的根页面的页号。
- Root of SYS_COLUMNS clust index：本字段代表SYS_COLUMNS表聚簇索引的根页面的页号。
- Root of SYS_INDEXES clust： index本字段代表SYS_INDEXES表聚簇索引的根页面的页号。
- Root of SYS_FIELDS clust index：本字段代表SYS_FIELDS表聚簇索引的根页面的页号。
- Unused：这4个字节没用，跳过。

以上就是页号为7的页面的全部内容，初次看可能会懵逼（因为有点儿绕），大家多瞅几次。

#### information_schema系统数据库

需要注意一点的是，用户是不能直接访问InnoDB的这些内部系统表的，除非你直接去解析系统表空间对应文件系统上的文件。不过设计InnoDB的大叔考虑到查看这些表的内容可能有助于大家分析问题，所以在系统数据库information_schema中提供了一些以innodb_sys开头的表：

```
mysql> USE information_schema;
Database changed

mysql> SHOW TABLES LIKE 'innodb_sys%';
+-----------------------------------------+
| Tables_in_information_schema (innodb_sys%) |
+-----------------------------------------+
| INNODB_SYS_DATAFILES                    |
| INNODB_SYS_VIRTUAL                      |
| INNODB_SYS_INDEXES                      |
| INNODB_SYS_TABLES                       |
| INNODB_SYS_FIELDS                       |
| INNODB_SYS_TABLESPACES                  |
| INNODB_SYS_FOREIGN_COLS                 |
| INNODB_SYS_COLUMNS                      |
| INNODB_SYS_FOREIGN                      |
| INNODB_SYS_TABLESTATS                   |
+-----------------------------------------+
10 rows in set (0.00 sec)
```

在information_schema数据库中的这些以INNODB_SYS开头的表并不是真正的内部系统表（内部系统表就是我们上边唠叨的以SYS开头的那些表），而是在存储引擎启动时读取这些以SYS开头的系统表，然后填充到这些以INNODB_SYS开头的表中。以INNODB_SYS开头的表和以SYS开头的表中的字段并不完全一样，但供大家参考已经足矣。这些表太多了，我就不唠叨了，大家自个儿动手试着查一查这些表中的数据吧哈～