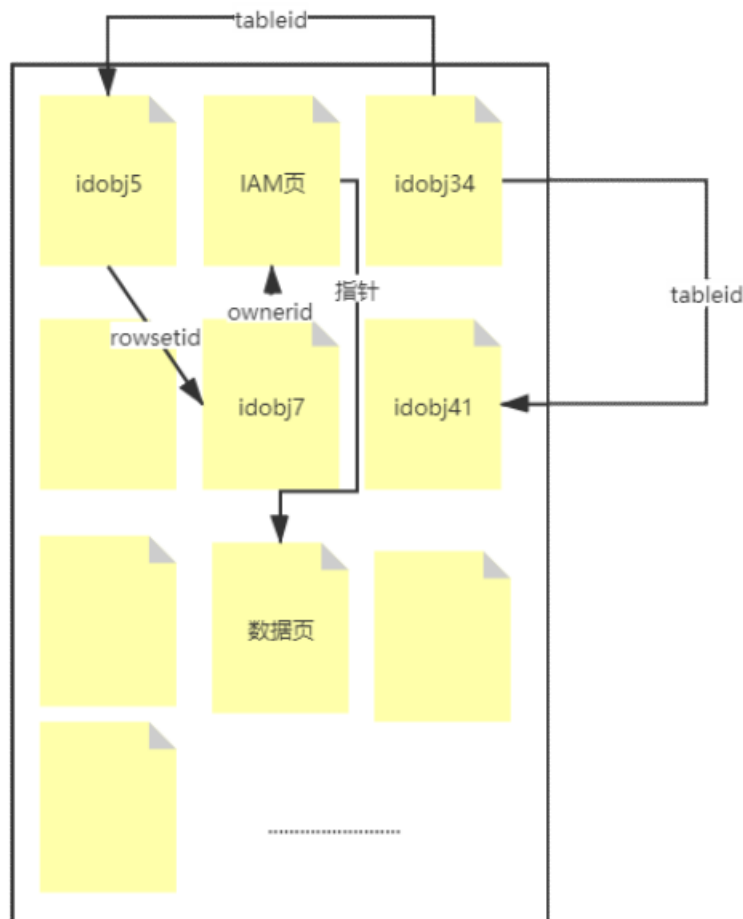


MDF 文件组织结构

Sqlserver 的数据文件主要有两种，一种是 mdf 文件，为主要数据文件，一种是 ldf，是日志文件，存放的是操作日志。在有些数据库，还能见到一种后缀为 ndf 的文件，是次要数据文件，其只会在选择拆分数据库后出现，作用和 mdf 文件相似。

MDF 文件和 NDF 文件主要是以页作为储存格式，即若干个字节组成一个数据页，若干数据页组成一个文件。数据页又由三个部分组成，分别是页头、数据区和数据插槽区。

每个数据页由 8192 个字节组成，其中页头占 96 个字节，数据区和数据插槽区不固定。数据插槽区从每个数据页的末尾开始，是一个一维数组，每个数组占两个字节，表明的是每条数据的第一个字节在页面中的偏移量。随着数据的增多，数据插槽区会向上增长，而数据区会向下增长，直到页面用尽。MDF 文件结构及其内部页面组织方式如下图所示。

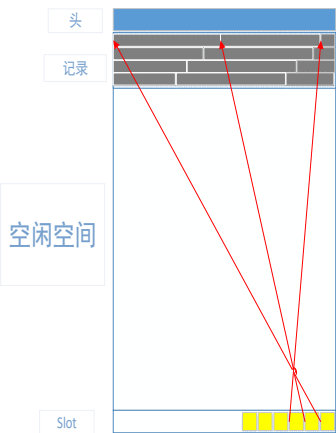


页面数据结构

页面结构

MDF 文件单个页面主要被分为四个部分，其页面结构如图

Slot 代表每条记录的开始偏移位置，是一个两个字节的偏移组，从每页末尾开始倒序向前增长，其顺序和 sqlserver 数据的逻辑顺序相同，当其中某条记录被修改时，sqlserver 并不会直接去修改物理文件中的该数据，而是在数据页中将修改后的数据作为一条新数据插入记录末尾，并将原指针指向修改后的数据开头。而删除也是同理，当删除某条数据时，并不会直接删除数据，而是将该记录对应 slot 置空。因此，slot 所代表的数据逻辑结构和记录的物理结构并不一致。并且这一特性为直接通过 mdf 文件恢复被删除数据提供了可能性。



页头结构

页面的前 96 个字节固定为页面头结构，页面头的结构如下图

第一行 (16字节)	页面头信息（一般恒定为1）	页面类型信息	TypeFlagbit (一般没用)	层级 (指页面在B树的层级)
	FlagBit (储存着描绘页面的一些信息)		IndexId	
	上一页ID（页码部分）			
	上一页ID（文件序号部分）		记录中的固定段长度	
第二行 (16字节)	下一页ID（页码部分）			
	下一页ID（文件序号部分）		插槽数（本页中有多少条记录）	
	idObj（用于识别表的id）			
	空闲字节数		已用字节数	
第三行 (16字节)	本页ID（页码部分）			
	本页页ID（文件序号部分）		ReservedCnt（释放页面空间辅助字节）	
	日志记录号（1）			
	日志记录号（2）			
第四行 (16字节)	日志记录号（3）		Reservedcnt最后添加的字段数量	
	操作ReservedCnt的内部事务ID（2）			
	操作ReservedCnt的内部事务ID（1）		页面内幽灵记录的数量	
	页面校验和			

在恢复数据过程中，需要重点关注的字段是页面类型、idObj、插槽数、固定段长度、本页

ID 和下一页 ID，其中本页 ID 被分为页码和文件号两部分，文件号为恢复多文件数据库提供了依据。其中页面我们需要关注的是 type 位 1、2、13，1 代表其是个数据页，2 代表其是索引页（仅做了解），13 是数据库的信息页，记录了版本等消息，一般在 MDF 文件第九页。本页 ID 和下一页 ID 的结构相同，都是由一个四字节的小端页面号和一个两字节的小端文件号组成，而固定长度则表明的是该条数据里所有固定长度段加起来的长度。

记录数据结构

记录数据结构的通用格式

Sqlserver 的 MDF 文件中，无论是表名、schema、还是数据，都共用一套数据格式，其数据结构如下表

Offset	长度(byte)	描述
0	1	记录的状态位
1	1	未使用
2	2	指向此条记录共有几列的字段的偏移
4	Pminlen-4	固定长度的数据
Pminlen	2	数据总共有几列
Pminlen+2	1+x/8	零位图
Pminlen+3+x/8	2	可变长度数据的列数
Pminlen+3+x/8+2	2	第一列可变长度的结束偏移
.....
Pminlen+3+x/8+2*y	2	最后一列可变长度的结束偏移
Pminlen+4+x/8+2*y	可变长度	可变长列的数据

表 1

状态位记录的是此条记录的一些状态信息，我们需要关注的是该字节的第一位，第五位和第六位，第一位代表的是版本信息，而第五位代表了此条记录是否有零位图，第六位则表示了记录中是否有可变长度列，此外，X 代表此条数据总共有多少列，Y 代表的的数据中总共有多少变长列。

零位图

零位图（nullbit map）是 sqlserver 中一直解析数据时的优化方案，其长度随数据列数变化而变化，其中每一位对应一数据中的一列，当零位图的某一位位 1 时，则代表对应位的数据值位 NULL，这对于解析变长列十分有效，如果没有零位图,将无法判断数据类型位 varchar 的列是一个长度为 0 的空字符串还是为 Null 值。

表名结构分析

数据库中的表名结构储存在页头的 objectid 为 34、页面 type 为 1 且 slot 数不为 0 的数据页面中，其具体的数据格式符合通用数据结构，下图是一张 16 进制下表名记录的截图

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	30	00	30	00	41	CF	AA	19	01	00	00	00	00	00	00	0E
10	00	55	20	00	00	00	00	01	04	00	00	00	75	2D	FD	00
20	F6	AB	00	00	75	2D	FD	00	F6	AB	00	00	00	00	00	00
30	0C	00	00	00	01	00	48	00	68	00	75	00	67	00	65	00
40	72	00	6F	00	77	00	73	00								

结合表一分析图一内容数据，需要重点关注的是划线字段，其 2 到 3 个字节值为“0x0030”，表明记录此条记录行数的数据储存在偏移为 0x0030 的位置上，4 到 7 个字节表明这个此表名对应的表 id 为“0x19AACF41”，通过这个 id，可以确定数据库表字段的记录。55 20 这两个字节表明的是表的类型，55 是 ASCII 码中的 U，表明这是一个用户表。而“0x000C”表明的是这条记录总共有 12 列数据，而 51 到 52 个字节“0x0001”表明此数据有一个变长列，由于此条记录是表名的记录，所以很容易得出，变长列对应的是表名记录，其结束的位置的偏移为“0x0048”，剩下的则是表名，将表名部分用 GBK 或 ASCII 编码可得出表名为：“hugerows”。值得注意的是在 sqlserver2012 及以后，表名数据新增了一个 status2 字段，其为四个字节的 int 类型，所以固定长度数据要比 2008 及以前长 4 个字节

表字段结构分析

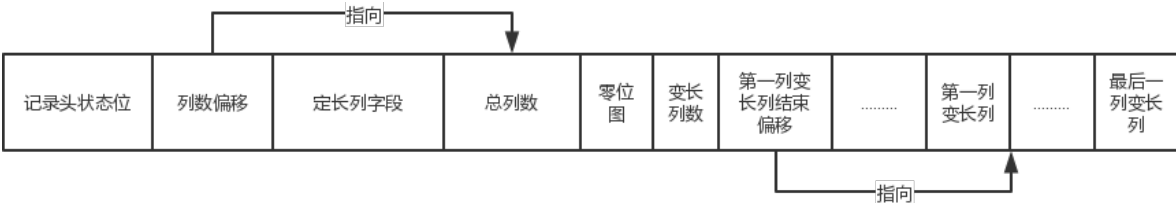
数据库的表字段结构储存在页头的 objid 为 41，type 为 1 的页面中

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	3c	00	2d	00	2d	57	01	6e	00	00	04	00	00	00	af	af
10	00	00	00	0c	00	00	00	24	d0	00	00	02	00	00	00	0c
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	0C	00	00	00	01	00	41	00	67	00	65	00	72	00	6f	00
40	77	00	73	00												

图为一个普通的 shcema 结构，第一条红线是对应表 ID，第一个蓝线部分为数据类型，图中数据类为 char，每个代码对应的数据类型请查看附录，第二个划线部分为长度，也就是说，图中数据在 sqlserver 中为 char（10），第三个数据为列名结束长度的偏移，最后红色划线部分为列名，其在储存格式中是两个字节代表一个字符，图中所示为 monkey

表记录结构分析

记录的结构字段符合上诉通用结构分析，记录的结构图可如下所示



解析时对定长列通过 `schema` 来判断其长度，对于变长列则通过结束偏移来判断其长度具体每个 `schema` 详细解析已在文章后面列出。

寻找指定数据页面

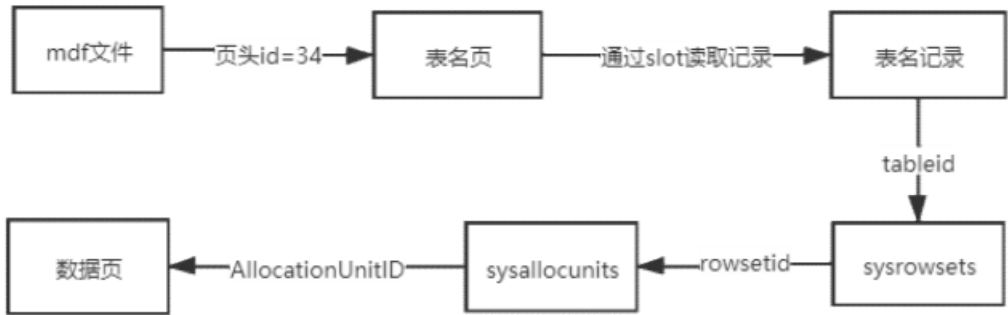
遍历法

正如在前面提到的，每一张表都有一个唯一的表 `id` 号，通过遍历所有 `objectID` 为 34 的页面，并用页面类型做筛选，我们可以筛选出所有的用户表，然后通过用户表的 `id`，可以在所有 `objectid` 为 41 的表里筛选出特定表对应的字段项。

但是，在搜索表对应的记录页面时候，事情并没有这么简单，在 `sqlserver2000` 及以前，页面页头的 `objectid` 和表 `id` 是一一对应，可以很轻松的通过 `objectid` 定位出数据页面的位置，而 `sqlserver2005` 及以后的版本改变了这一对应关系，表 `id` 和页头的 `objectid` 不再是简单的对应的关系。

想要找到特定表 `ID` 对应的数据页，首先需要解析 `sys.sysallocunits` 表，`sys.sysallocunits` 为系统基表，在通常状态下，必须进入 `DAC` 模式，才能查看该表内容，`sys.sysallocunits` 表的 `objectid` 在 `sqlserver` 当中固定为 7，其字段如表所示。其中需要重点关注多的列是 `audid` 列和 `ownerid` 列，但是为了确定我们要查询的目标表和 `sysallocunits` 每条数据的对应关系，还需要去检索 `object` 为 5 的 `sysrowsets` 表页面,其 `schema` 可以通过在在数据库中输入 `sp_help 'sys.sysrowsets'` 语句查询，具体可见附录。在 `sysrowsetsb` 表中需要被关注的是 `rowsetid`、`idmajor` 和 `idminor`，具体查数据页面的流程如下图。

Column_name	Type	Length
audid	bigint	8
type	tinyint	1
ownerid	bigint	8
status	int	4
fgid	smallint	2
pgfirst	binary	6
Pgroot	binary	6
pgfirstiam	binary	6
pcused	bigint	8
pcdata	bigint	8
pcreserved	bigint	8



图

如上图所示，在查询出了 `AllocationUnitID` 后，可以通过换算公式得到 `objectid` 和 `Indexid`，通过这两个值并结合 `type=1` 可以确定出对应表数据页，换算公式如下

$$\text{AllocUnitId} = (\text{objectid} \ll 48) | (\text{indexid} \ll 16)$$

通过这种方式遍历整个 `MDF` 文件，即可获取到所有的记录页面，这种方法优点在于当

数据库某些页面出现损坏时（如 IAM 页），也能恢复大部分数据。缺点也十分明显，当 MDF 文件较大时，整个过程性能会很低。

IAM 页结构

基于上述的解析过程，寻找数据页的过程可以进一步的简化，在正常情况下，数据库中的每个表都至少存在一个对应的 IAM 页面，IAM 页面中包含着该表的数据页面的索引。IAM 页的页头 type 为 10，sqlserver 数据储存被分为了两个区，一个是混合区，混合区中的数据页可能来自于多个不同的表。另一个则是统一区，统一区以八个页面为一个分区，当一个分区被分配给某个表后，就只能储存该表数据页面。

IAM 页也被分为了混合区和统一区，其中混合区有 8 个槽位，每个槽位对应一个页面，在 sqlserver2016 之前，页面储存会先储存在混合区，当 8 个槽位全部占满后，才会开始分配统一区。下图是 hugerow 表 IAM 页面统一区的物理结构

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	38	1f	00	00	00	00	00	00	00	20	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00

前两个字节为状态字节，之后的字节则代表的是统一区分配情况为 0 则是该区没有分配给目标表，图中统一区第 8 个字节的第 6 位为 1，则可以得知统一区第 62 个区被分配给了 hugerow 页面，由于一个区由 8 个连续页面组成，所以可以计算得出从 488 页开始的连续 8 个页面，都属于表 hugerows，一个 IAM 页面的混合区有 7988 个字节，也就是 63904 个区，511232 个页面，当页面大于这个值时，就会出现在第二个 IAM 页面上，一个 IAM 页面管理着大约 4G 的数据。

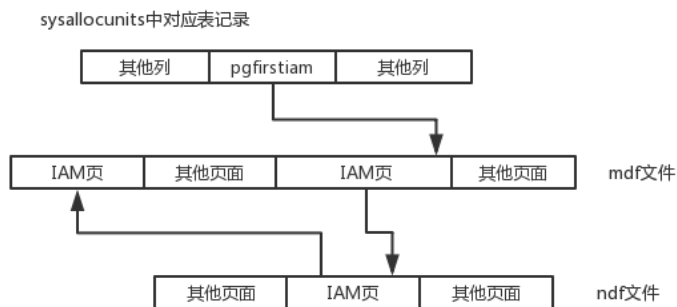
而混合区的物理结构如下

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	00	00	5e	00	00	00	00	00	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	<u>00</u>	<u>00</u>	<u>00</u>	<u>00</u>	<u>01</u>	<u>00</u>	<u>af</u>	<u>00</u>
30	00	00	01	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

其中第一个划线部分表示此 IAM 页面所对应的页面区域的开始部分，可以看出页面 ID 为 0，文件号为 1，所以这个 IAM 页对应从第 0 页开始的 511232 个页面,第二个划线部分为混合区的第一个槽位，其文件号为 1，页面 ID 为 175（小端模式），表明表被分配进了 175 开始的连续 8 个页面组成的统一区中。

IAM 索引法

通过 IAM 页找出数据页是一种比遍历法要高效的查询方式，其前置步骤和遍历法相似，区别是当找到 sysallocunits 页面中对应的条目时，所需要的不是 AllocationUnitID，而是 pgfirstiam 列，这一列指明的是对应表格的第一个 IAM 页的页码 ID。



图为在一个执行了拆分的数据库中。查找 IAM 页的流程图，如图所示，IAM 页面在数据文件中是链式组织的，当找到第一个 IAM 页面后，便可以通过页头中下一页 id(nextpageid) 找到下一个 IAM 页面，从而找到所有的 IAM 页。在找到所有的 IAM 页后，相应就能找到所有的数据页面。IAM 页索引查找的方式因为不用遍历整个文件，所以在数据文件较大的情况下，速度优势明显，且文件越大，优势越大。而缺点则是对 MDF 文件的完整性有着较高的要求，如果 IAM 链中的某一个页面损坏，则之后的数据会全部无法解析到。

特殊数据

Sqlserver 中存在几种特殊的数据格式，他们大多属于变长数据类型，但是储存方式又和普通的变长数据不同，接下来将详细介绍行溢出数据储存格式和 LOB 数据储存格式

行溢出数据

行溢出数据是 sqlserver2005 开始拥有的新特性，其只针对 varchar、nvarchar、varbinary 生效，sqlserver 中一个页面大小为 8192 字节，而实际能储存数据的只有 8060 个字节，当一条记录存在上述的类型其中之一，且数据大于了 8060 字节时，就会发生行溢出，上述字段的记录将不会储存在本页面中，而会单独储存在一个页面中，原来数据的位置将会变成一个 24 字节数据，其中包含了指向存放行溢出数据页面的指针。

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	02	00	00	00	01	00	00	00	29	00	00	00	34	08	00	00
10	cc	08	00	00	01	00	00	00								

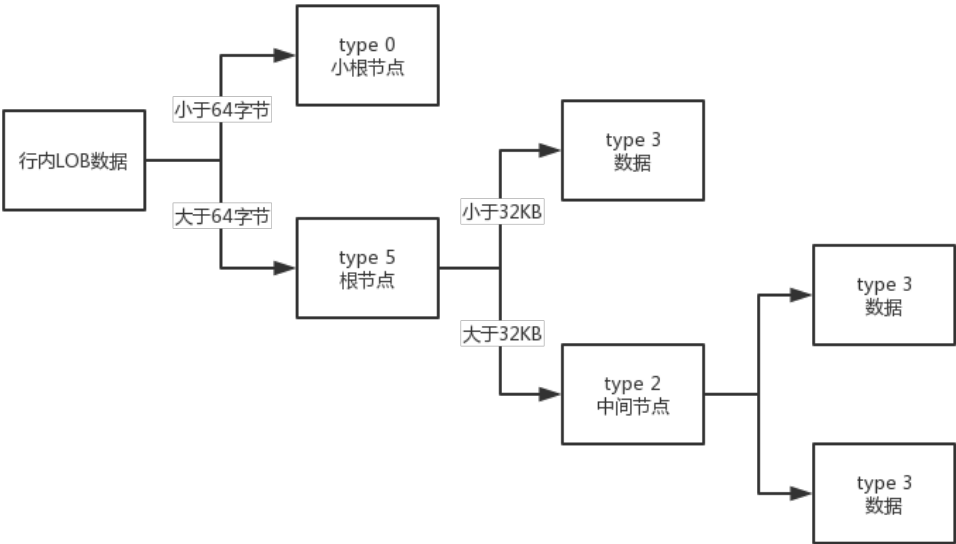
图为一个发生了行溢出的 varchar 列的数据指针，其第一个字节表代表的是特殊字段类型，2 代表的是这是一个行溢出数据类型，1 则代表的是 LOB 数据类型，而第 16 个字节开始的 8 个字节，则是行出数据页面的一个指针，前四个字节是页面，之后两个字节是文件号，最后两个字节是槽位，图中数据表明 varchar 数据溢出到了第一个文件的 2252 页，其在页面中的 slot 位置是 0，也就是该页面的第一个 slot 指向的是 varchar 的实际数据。行溢出数据所在的页面不一定只有行溢出数据。当我们解析数据的时候，要判断一个数据有没有发生行溢出，需要查看其长度偏移的第一个字节的 primary（第七位）位是否为 1，如果为 1，则表

明其是一个行溢出列。

LOB 数据

LOB 全称为 Large Objects，在 sqlserver 中，text、ntext、image、xml 为原生 LOB 数据，不管数据有多长，其都会将数据储存在专有的 LOB 数据页面，而 nvarchar、varchar 和 varbinary 当其长度设置为 max 的时候，也会由行溢出数据转变为 LOB 数据，LOB 数据在原数据位置会变成 16 个字节指针，其格式如表所示。LOB 数据是一颗 B 树结构，当 LOB 数据总数小于 32KB 的时候，行内数据将会指向一个文本根节点，而根节点最大有 5 个字节节点指向数据块，当 LOB 数据大于 32KB 时数据块和根节点之间会增加中继节点，而当数据小于 64 字节时，会直接储存在根节点中。具体的解析流程如图所示。

Offset	Bytes	Description
0	4	时间戳
4	4	未使用
8	4	页码 ID
12	2	文件号
14	2	插槽号



不同类型的节点会有不同类型的 type 值，节点记录在 MDF 文件页中的物理结构如下图所示

LOB记录头	节点头信息	节点信息
--------	-------	------

LOB 记录头中包含记录长度、状态位、时间戳、以及所记录的节点类型，不同类型的代号不同，type 5 表示记录为一个 B 树的根节点，根节点最多有 5 个子节点，type 0 表示记录为一个根节点，根节点中并没有储存子节点的指针，而是直接储存了 LOB 数据，只在数据小于 64 字节时候启用。Type 2 类型则是 B 树的中间节点，Type 3 类型则是储存数据的类型，具体见文章末尾附录

通过上述结构描述，可以准确无误的解析原生 LOB 数据，但是对于行溢出数据转变成的 LOB 数据，在 sqlserver 官方文档中描述中虽写和原生 LOB 一致，实际上储存结构有很大不同，当数据较小时，这些行溢出转变成的 LOB 数据依旧会将数据储存在行内，如同普通行溢出数据，当数据大过一定阈值，将会转变为 type 为 4 的行内根节点，行内根节点结构

十分简单，我们只需要知道从记录开始的第 14 个字节是第一个子节点指针、之后是第二个第三个，如 68 1f 00 00 c9 01 00 00 01 00 00 00 是其中一个节点，头四个字节是 LOB 数据分片后的结束偏移，之后四个字节是页码 ID，之后两个字节是文件 ID，再之后是插槽号。而除了根节点不一样，其他行溢出数据转变成的 LOB 数据和原生 LOB 数据基本一致。

行压缩数据

行压缩是 sqlserver2008 以后压缩数据的一种方式在进行行压缩后，会对指定表中每一行的数据进行压缩，数据格式会发生很大变化，且不同的数据受行压缩的影响不尽相同，行压缩后的记录格式如图。

其中标题区域是一个字节，和状态为相似，其每位含义如下：

位 0：表示是否是行压缩列（CD 记录），如果为 1 则表示进行了行压缩。

位 1：表示该行包含的版本控制信息。

位 2~4：当作一个三位的值，表示行中储存的是那种信息，具体如下：

000——主记录

001——备份空记录

010——转发记录

011——备份数据记录

100——已转发记录

101——备份已转发记录

110——索引记录

111——备份索引记录

位 5：表示该行包含一个长数据区域

剩下几位未使用

而列说明符被分为两个部分，第一部分为总列数（在书上说是短数据列数，经测试，书上是错的），当小于 127 列时，其为一个字段，当大于 127 列时，其位 7 会被置 1，此时用两个字节表示长度，总共能表示 32767 列，之后是列长度，列长度用四位表示，也就是一个字节可以表示 2 个列的长度,含义如下

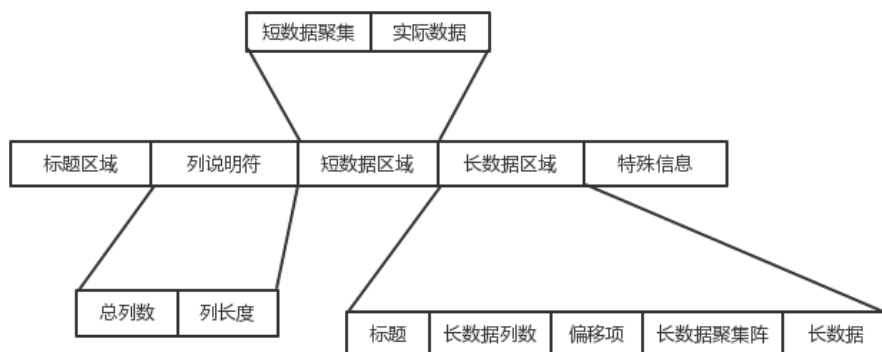
- 0 (0x0) 表示对应列为空
- 1 (0x1) 表示 0 个字节短值
- 2 (0x2) 表示 1 个字节的短值

.....

- 10 (0xa) 表示对应的是一个长数据值，并且不占用短数据区域的任何空间
- 11 (0xb) 用于值为 1 的 bit 类型列，不占用短 9 数据区域的空间
- 12 (0xc) 表示相应列是一个 1 字节的符号（和页压缩有关）

如一个列的长度区域是 a5 32,则表示其第一列长度为 4，第二列是个长数据列，第三位长度为 1，第四列长度为 2 。

短数据聚集列是一个快速索引短数据的字段，每个字节对应 30 列的偏移，当总列数小于 30 时，其为 0



长数据格式类似，首先标题字节只有两位被使用，位 0 表示长数据是否有两个字节的偏移，此恒为 1，位 1 表明长数据总是否有复杂列，接下来是偏移项，偏移项功能和普通数据中的变长列结束偏移相似，偏移项的开头第一个字节恒位 0，接下来每两个字节表示一个长数据的偏移，偏移项采用了小端记录，当其 **primary** 位为 1 时，说明其是一个复杂列（也就是 LOB 和行溢出列之类的）。具体每个数据在数据中如何压缩的，请参照源码，此处不在赘述。

主键问题

几乎所有的数据表都会设置一个主键，主键的存在会影响数据表物理储存顺序，没有设置主键的数据行会按数据行的逻辑顺序来储存数据，设置主键后，数据储存的物理顺序和逻辑顺序将不会一一对应，当第四列被设为主键时，其物理顺序将会被提到第一列，此时解析表数据就会十分困难，我们需要进行一个联表解析，首先解析 **objectid** 为 5 的表，获取对应表的 **rowsetid**，再通过 **rowsetid** 查询 **object** 为 3 的 **sysrscols** 表，**sysrscols** 表的字段结构请看表附录，其中 **rscolid** 是字段的逻辑顺序，**hbcolid** 是字段的物理顺序，由此可以判断出顺序，而 **ordkey** 为 1 时表明此字段为主键。

各数据类型结构分析

Int

Int 数据类型结构很简单，是四个字节的小端排序整数，如 09 00 00 00 则表明是 9。在启用行压缩后，实际占几个字节就用几个字节且首个字节的 **primary** 位会变为 1

BigInt

BigInt 类似于 int，但是其是 8 个字节的小端排序的整数。
启动行压缩后同 int

TinyInt

Tinyint 是一个字节的整数。
无法再压缩

Smallint

Smallint 是两个字节的整数。
同 int

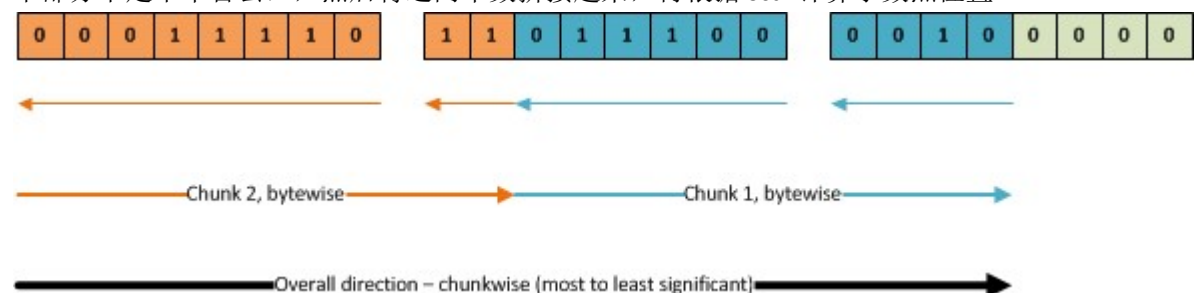
Float

float 是用 IEEE 754 标准储存的浮点数，对应的是 java 里的 double。
省略了尾数中带 0 的数值

Decimal

其对应的是 java 中的 BigDecimal，是一个 8 字节整数，并且需要根据 scale 来设定其小数点位置

启动行压缩后其储存格式比较复杂，其实际占用字节不确定，每十个字节组成一个数（剩下部分不足十个舍去），然后将这两个数拼接起来，再根据 scal 计算小数点位置



Date

date 是日期数据中比较好解析的一种，其是一个三个字节的固定长度类型，储存默认值 1900-1-1 日起到现在经过的天数，唯一的问题是.NET 没有三个字节整形的标准数据类

型，所以想要获取正确的天数，需要做一些位移，唯一公式如下

$$\text{day} = (\text{byte}[2] \ll 16) + (\text{byte}[1] \ll 8) + \text{byte}[0]$$

当启用行压缩后，其格式无任何改变

Datetime

`datetime` 不仅储存日期，也储存时间，所以 `datetime` 所占长度为 8 个字节，第一部分是时间部分，第二部分是日期部分，日期部分和 `date` 类型类似，不过这次其是用四个字节储存，所以不用位移处理，时间部分则比较特殊，其储存的是自午夜以来的 `tick` 数，一个 `tick` 为 `1/300s`，为了表示刻度值，我们可以先定义一个 `10d/3d` 的常数，然后通过如下式子计算：

小时：`X/300/60/60`

分钟：`X/300/60%60`

秒：`X/300%60`

毫秒：`x%300*10d/3d`

当启动行压缩后使用两个 4 字节整数表示整数数据。此整数值表示自 1900 年 1 月 1 日以来经过的天数。前 2 个字节最高可以表示 2079 年。在 2079 年之前，此类压缩始终可以节省 2 个字节。每个整数值表示 3.33 毫秒。压缩在前五分钟只占用前 2 个字节，在 4PM 之后将使用第四个字节的。因此，压缩在 4PM 之后只能节省 1 个字节。当 `datetime` 像任何其他整数一样进行压缩时，压缩可在日期方面节省 2 个字节。具体算法请参考代码

smallDateTime

`Smalldatetime` 只用四个字节储存，前两个字节是午夜以来的分钟数，后两个字节是 1900-1-1 以来的天数

小时：`X/60`

分钟：`X%60`

使用两个 2 字节整数表示整数数据。日期占用 2 个字节。它是自 1901 年 1 月 1 日以来经过的天数。从 1902 年起便需要 2 个字节。因此，自该时间之后的日期不会节省任何空间。时间是自午夜以来经过的分钟数。超过 4AM 后，时间值就开始使用第二个字节。如果 `smalldatetime` 只用于表示日期（常见情况），则时间为 0.0。通过为行压缩以最高有效字节格式存储时间，压缩操作可节省 2 个字节。但实际情况和微软文档所说不同，具体可以参考代码，

CHAR、varchar、Nchar、Nvarchar、text、Ntext

这四种类型储存相似唯一的区别是 `Nchar`、`Nvarchar`、`Ntext` 是用两个字节储存一个字符，`char` 等是一个字节储存一个字符

启动行压缩后就自动去掉所有 00 字节

Money 和 SmallMoney

Money 和 SmallMoney 一个占 8 字节，一个占 4 字节，都是小端排序，解析出来的值再除以 10000 得到的小数就是实际数值。

启动行压缩后其数据压缩类似 int，实际用到多少个字节就占多少字节，且首个字节的 primary 位会变为 1

Bit

虽然叫 BIT，但是其储存大小是一个字节，其字节的第 0 位代表值，为 1 的时候是 true，为 0 的时候是 false。

行压缩无效

Binary 和 VarBinary

直接顺序读取数据用 16 进制显示就是其实际值

启动行压缩后会删除尾随的 0

Uniqueidentifier

和 binary 类似，不过注意读取出来的数据需要手动加上短横线。

启动行压缩后无效

校验和

Sqlsever 自带有校验和机制以验证每一个页面完整形，其算法大致如下
读 8KB 进 BUF

将 BUF 头部 CHECKSUM 的 4 字节值清 0

uint32 checksum = 0 //初始 checksum
for i in range(0,15):

 //每扇区的初始 checksum

 overall = 0;

 for ii in range(0,127):

 //对当前扇区的每个 4 字节做累加异或

 overall = overall ^ BUF[i][ii];

 //对每扇区的 checksum 进行移位，方法为向左移位 15-i 位，

 //左边移出的 15-i 位补到最低位。

```
checksum = checksum ^ rol(overall, 15- i);  
return checksum; //Gets checksum  
其算法具体实现参照源码，此处不再赘述
```


如何获取 sqlserverMDF 文件位置

Sqlserver 的 MDF 文件储存位置在系统的注册表中有明确的记录,其记录在 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Microsoft SQL Server\实例名\MSSQLServer 中的 DefaultData 项里

 DefaultData REG_SZ C:\Program Files\Microsoft SQL Server\MSSQL14.MSSQLSERVER\MSSQL\DATA

如果该项为空，说明 data 文件处于默认路径，可以查看

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MicrosoftSQLServer\MSSQL14.MSSQLSERVER\Setup 中的 SQLDataRoot 来确定 data 文件位置

 SQLDataRoot REG_SZ C:\Program Files\Microsoft SQL Server\MSSQL14.MSSQLSERVER\MSSQL

利用 libguestfs 工具，我们可以轻松的获取 windows 系统注册表中的键值对，并且提取出其中的目标文件。

源码解释

本程序为 MDF 解析程序的一个原型程序，由原生 java 编写，大概 4K 行，现对程序一些关键部分做出讲解。

工具类

Checksum

Checksum 类的作用是对页面进行校验，以检查页面是否出现了错误，它会将一个页面分为 16 份，每份 128 段，每段四个字节，对其进行一个校验和的计算，以此判断页面是否受损，当返回值为 true 时页面完整，为 false 时，页面出错。

CorruptUtils

这是一个为测试编写的工具类，其主要作用是以 512 个字节为单位，随机将页面部分置零，模拟数据损坏，用于测试软件恢复性能

RecordCuter

这是一个切割记录类，本程序的解析思路是首先将记录按行进行切割，再进行解析，此类根据页面末尾的偏移指针定位每条记录的开头，对记录进行切割，并返回一个 `byte` 数组的 `list` 列表。

DeletedRecordCuter

这是一个强制切割数据的类，主要作用是用来恢复还没有被覆盖的已删除数据，本删除类不根据页面尾部的指针，而是直接从第 97 个字节开始遍历页面进行解析，如果页面受损，会有出错机率

HexUtil

此类是一个专门用来解析 16 进制和 2 进制的工具类，里面提供了多种解析 16 进制字节的方法。

LobRecordParser

此类专门用于解析 LOB 数据，当数据中出现 LOB 数据时，就会调用此类，此类会自动去遍历 LOB 数据的 B 树，并拼接好数据返回。此类的主方法传入参数为 LOB 数据所在页面以和 slot 的值

OverflowRecordParser

此类用于解析行溢出数据，当数据中拥有行溢出数据时，会自动调用此类，其中只有一个方法，参数是行溢出数据所在指针以及 slot

MainParserIndex

此类是解析数据的主要类，其主方法需要传入的参数为想要解析的数据的表 ID，然后开始在关键系统表的数据中进行查询匹配，在系统表中查询出数据后，通过 IAM 页对数据页面进行定位，再调用 `RecordCuter` 对数据进行剪切，然后将剪切后的数据传入行数据解析类，

解析出数据后返回。本方法不适合当数据库受损时采用，如果一个 IAM 页受损，之后 IAM 页的数据都将无法获取

MainParserForce

主数据强制解析，本类采用强制遍历整个文件的方法来遍历数据页，适合当文件出现损坏时的场景，此方法解析数据较为缓慢

PageCuter

页面切割类，其作用时当程序按字节将文件读入内存时，调用此类，将文件按 8192 个字节进行切割，返回是一个 byte 类型的二维数组。

RawColumnParser

此类用于解析数据，主方法有两个参数，一个是数据页面的 list 集合，一个是储存 schema 的 list, 此类通过这两个相结合，先将数页面数据进行切割，然后再通过 schema 解析切割之后的数据。并返回一个以 MAP 储存数据的 list 集合。

PageUtils

这是一个页面工具类，其主方法传入的参数为文件路径，传入后将读取文件。因为系统页都是建立了索引的，可以找出其中一页后就可以根据页头中的上一页和下一页，迅速找到链上所有页面

DOMAIN 类

Ischema

Ischema 为一个接口类，接口中共有 9 中方法，具体实现方法根据数据类型不同而不同，具体如下：

名称	输入	输出	描述
getValue	切割好的数据，开始结束偏移	对应数据类型解析结果	用于解析对应数据类型数据。
Name	无	String 类型列名	返回对应列名字
getLength	无	Int 类型长度	返回对应列长度
fixd	无	返回 int 值	表示其是否是定长列
isLOB	无	Bool 值	表示其是否是 LOB 数据
getRowCompressValue	切割好的数据 byte[], 开始偏移 staroffset, 长度 length, 复杂列 bool 值	对应数据类型解析结果	根据其是否是个复杂列来解析行压缩数据
getSqlSchema	无	返回 string 类型的 sql 的语句一部分	返回一个常见该列的 sql 语句，用于最后 sql 拼接
getOverflowValue	同 getValue	对应数据类型解析结果	用于解析发生了行溢出的数据

在个别数据类中还有一个 parserChangeLob 方法，主要用于解析转变为 LOB 数据的行溢出列。

PageHeader

这是页头内，其构造方法要求传入一个页面的 byte 数组，之后会自动解析前 96 字节内容，生成相应的一个页头类。

各数据类型对应 id

system_type_id	name
34	image
35	text
36	uniqueidentifi
40	date
41	time
42	datetime2
43	datetimeoffset
48	tinyint
52	smallint
56	int
58	smalldatetime
59	real
60	money
61	datetime
62	float
98	sql_variant
99	ntext
104	bit
106	decimal
108	numeric
122	smallmoney
127	bigint
165	varbinary
167	varchar
173	binary
175	char
189	timestamp
231	nvarchar
231	sysname
239	nchar
240	hierarchyid
240	geometry
240	geography
241	xml

Object 3 sys.syrscols 表的 schema

	Column_name	Type	Computed	Length	Prec	Scale	Nullable	TrimTrailingBlanks	FixedLenNullInSource	Collation
1	rsid	bigint	no	8	19	0	no	(n/a)	(n/a)	NULL
2	rscolid	int	no	4	10	0	no	(n/a)	(n/a)	NULL
3	hbcolid	int	no	4	10	0	no	(n/a)	(n/a)	NULL
4	rcmodified	bigint	no	8	19	0	no	(n/a)	(n/a)	NULL
5	ti	int	no	4	10	0	no	(n/a)	(n/a)	NULL
6	cid	int	no	4	10	0	no	(n/a)	(n/a)	NULL
7	ordkey	smallint	no	2	5	0	no	(n/a)	(n/a)	NULL
8	maxinrowlen	smallint	no	2	5	0	no	(n/a)	(n/a)	NULL
9	status	int	no	4	10	0	no	(n/a)	(n/a)	NULL
10	offset	int	no	4	10	0	no	(n/a)	(n/a)	NULL
11	nullbit	int	no	4	10	0	no	(n/a)	(n/a)	NULL
12	bitpos	smallint	no	2	5	0	no	(n/a)	(n/a)	NULL
13	colguid	varbi...	no	16			yes	no	yes	NULL

Object 5 sys.sysrowsets 表的 shcema

	Column_name	Type	Computed	Length	Prec	Scale	Nullable	TrimTrailingBlanks	FixedLenNullInSource	Collation
1	rowsetid	bigint	no	8	19	0	no	(n/a)	(n/a)	NULL
2	ownertype	tinyint	no	1	3	0	no	(n/a)	(n/a)	NULL
3	idmajor	int	no	4	10	0	no	(n/a)	(n/a)	NULL
4	idminor	int	no	4	10	0	no	(n/a)	(n/a)	NULL
5	numpart	int	no	4	10	0	no	(n/a)	(n/a)	NULL
6	status	int	no	4	10	0	no	(n/a)	(n/a)	NULL
7	fgidfs	smallint	no	2	5	0	no	(n/a)	(n/a)	NULL
8	rcrows	bigint	no	8	19	0	no	(n/a)	(n/a)	NULL
9	cmplevel	tinyint	no	1	3	0	no	(n/a)	(n/a)	NULL
10	fillfact	tinyint	no	1	3	0	no	(n/a)	(n/a)	NULL
11	maxnullbit	smallint	no	2	5	0	no	(n/a)	(n/a)	NULL
12	maxleaf	int	no	4	10	0	no	(n/a)	(n/a)	NULL
13	maxint	smallint	no	2	5	0	no	(n/a)	(n/a)	NULL
14	minleaf	smallint	no	2	5	0	no	(n/a)	(n/a)	NULL
15	minint	smallint	no	2	5	0	no	(n/a)	(n/a)	NULL
16	rsguid	varbinary	no	16			yes	no	yes	NULL
17	lockres	varbinary	no	8			yes	no	yes	NULL
18	scope_id	int	no	4	10	0	yes	(n/a)	(n/a)	NULL

Type 5 节点信息		
Offset	Bytes	Description
0	4	长度（offset）
2	2	页码 ID
4	2	文件号
6	4	插槽号

LOB 记录头		
Offset	Bytes	Description
0	1	状态位
1	1	未使用
2	2	记录长度
4	8	时间戳
12	2	记录类型

Type 5 头信息		
Offset	Bytes	Description
0	2	最大子节点数
2	2	现有子节点数
4	2	层级
6	4	未使用

Type 2 节点信息		
Offset	Bytes	Description
0	4	长度（offset）
4	4	未使用
8	4	页码 ID
12	2	文件号
14	2	插槽号

Type 2 头信息		
Offset	Bytes	Description
0	2	长度（offset）
2	2	未使用
4	2	页码 ID

Type 3 信息		
Offset	Bytes	Description
0	14	Type3 头
14	记录长-14	数据