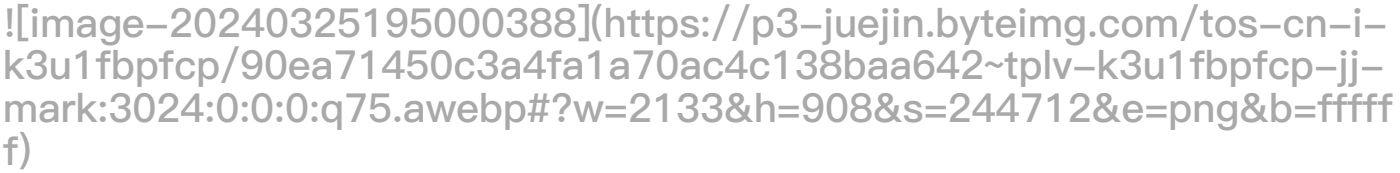
Please visit website: http://cxyroad.com

MySQL索引18连问，谁能顶住

前言

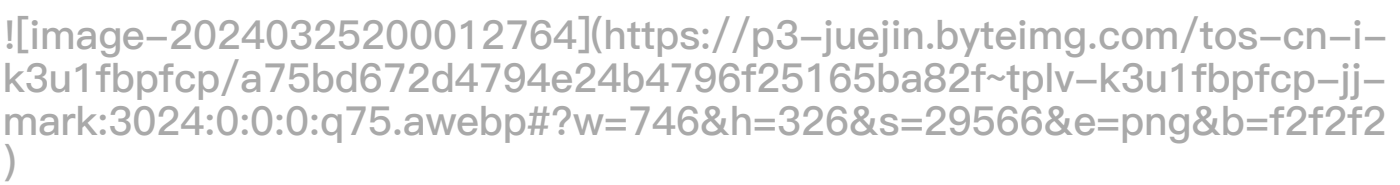
过完这个节，就要进入金银季，准备了 18 道 MySQL 索引题，一定用得上。



* 作者：
* 感谢每一个支持： [github](http://cxyroad.com/ ”https://github.com/Rodert/JavaPub”)

---

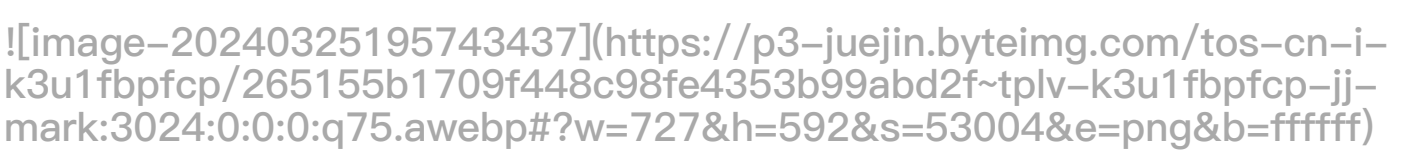
1. 索引是什么



* 索引是一种数据结构，用来帮助提升查询和检索数据速度。可以理解为一本书的目录，帮助定位数据位置。
* 索引是一个文件，它要占用物理空间。

---

2. MySQL索引有哪些类型



数据结构维度

* B+tree 索引： B+树是最常用的索引类型，所有数据都会存储在叶子节点上，时间复杂度是 `O(logn)` ，擅长**范围查询**。
* Hash 索引： 哈希索引就是采用哈希算法，将键值换算成新的哈希值，映射到对应槽位，然后存储到哈希表中，**擅长做对等比较（=，in）**。
* Full-text 索引： 全文索引是一种建立倒排索引，实现信息检索。在 MySQL 不同版本中支持程度不同。
* `R-Tree` 索引： 属于地理空间数据类型查询，通常使用较少。

物理存储维度

簇 `cù`

* 聚簇索引： `InnoDB 引擎` 要求必须有聚簇索引，也就是在主键字段建立聚簇索引。
* 非聚簇索引： 非聚簇索引就是以非主键创建的索引，在叶子节点存储的是表主键和索引列。 `InnoDB 引擎`

逻辑维度

* 主键索引： 主键索引是一种特殊的唯一索引，不允许值重复或者值为空。
* 普通索引： 普通索引是 MySQL 中最基本的索引类型，允许在定义索引的列中插入重复值和空值。
* 联合索引： 联合索引指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用联合索引时遵循最左前缀集合。
* 唯一索引： 唯一索引列的值必须唯一，允许有空值。
* 空间索引： 空间索引是一种针对空间数据类型（如点、线、多边形等）建立的特殊索引，用于加速地理空间数据的查询和检索操作。

---

3. 主键索引和唯一索引有什么区别

* 数量限制： 唯一索引有多个，但是主键索引一张表只能有一个。
* 本质区别： 被唯一索引约束的健可以为空，主键索引不可以。
* 外键引用： 主键可以被其他表作为外键，从而建立表之间的关系。而唯一索引则不能被其他表用作外键。

---

4. 什么是聚簇索引和非聚簇索引？它们在InnoDB存储引擎中是如何工作的？

聚簇索引是将表的数据按照索引顺序存储在磁盘上，聚簇索引的叶子节点直接存储了实际的数据行，而不是指向数据的指针。所以在查询的时候减少了磁盘的随机读取，无需进行多次磁盘I/O效率很高。

非聚簇索引是一种基于指针的索引，有时也叫它二级索引。非聚簇索引不直接存储实际的数据，seelec 语句在执行查询时，会先根据二级索引定位到数据所在的磁盘位置，然后再进行一次磁盘I/O操作，读取实际的数据行。

---

5. 复合索引和单列索引有何区别?

* 顾名思义，单列索引就是在一个列上创建的索引，复合索引就是多个列上创建的索引。
* 当只涉及到一个字段查询，单列是非常快速的。当涉及到多个字段查询，WHERE 子句引用了符合索引的所有列或者前导列时，查询速度会非常快。
* 在复合索引中，列的顺序非常重要。MySQL会按照索引中列的顺序从左到右进行匹配。例如，对于复合索引(a, b, c)，它可以支持a、a,b和a,b,c三种组合的查询，但不支持b,c进行查询。因此，在创建复合索引时，应把最常被访问和选择性较高的列放在前面。

当然具体如何选择需要看查询需求、数据分布和性能要求。如果你有开发需要欢迎在 JavaPub 下留言讨论。

---

6. Hash 索引和 B+ 树索引区别是什么？如何选择?

哈希索引：

* **工作原理**：通过哈希算法将被索引的列的值存储到一个固定长度的桶（Bucket）。使得在查询特定值的时候非常高效，因为可以直接计算出存储位置，快速定位到数据。
* **查询效率**：在等值查询下，哈希查询效率极高，可以在常数时间复杂度内定位到目标数据。但是范围查询和排序操作时，哈希索引的效率较低，因为哈希算法会导致数据随机分布，无法保持原有的顺序。
* **磁盘存储**：hash 索引的存储是随机的，可能导致磁盘的随机访问，从而降低磁盘的利用效率和查询效率。
* **插入和删除操作**：Hash 索引在插入和删除操作方面相对简单，只需要通过哈希函数确定存储位置即可。

[B+树白话详解_下载](http://cxyroad.com/)

B+树索引

* **工作原理**：B+树索引使用平衡树，将索引健的值按照顺序保存在树节点中，根据键值的大小关系，并通过节点之间的指针进行查找，快速定位存储了数据的叶子节点。
* **查询效率**：B+树擅长范围查询和排序操作，因为他是按照顺序存储数据，可以高效的支持范围查询和排序操作。
* **磁盘存储**：B+树索引的节点是有序存储的，有利于磁盘的顺序访问，从而减少磁盘的IO次数，提高查询效率。
* **插入和删除操作**：B+树在索引删除和插入操作时，需要维护树的平衡，可能进行节点的拆分和合并，相对哈希索引来说操作更复杂。

所以在选择上：

1. **查询维度**：如果查询主要是等值查询，且对性能要求较高，Hash 索引可能是一个好的选择。然而，如果查询涉及到范围查询、排序操作或模糊查询，B+ 树索引则更为合适。
2. **数据维度**：如果索引列具有大量重复值，Hash索引的效率可能会下降，因为哈希碰撞会导致性能下降。在这种情况下，B+ 树索引可能更为稳定。
3. **磁盘存储和I/O维度**：由于 Hash 索引可能导致磁盘的随机访问，如果磁盘 IO 是性能瓶颈，那么 B+ 树索引可能更适合，因为它更有利于磁盘的顺序访问。

从这三个维度可以很好的应用在你的开发工作中，如果是小数据量的 web 网站查询、直接用 B+ 树就可以了。对于数据量的大小评估，后面单开一篇讲解。

---

7. 索引是否越多越好？为什么？

不是。索引是建立在原数据上的数据结构，所以不论在查询还是更新维护、一定会带来开销。

比如一本书有 100 页，我构建了 50 页的目录，你觉查询起来还会方便吗？

* 数据量小的表不需要建立索引，建立索引反而会增加额外开销。
* 数据变更后索引也需要更新，更多的索引意味着更多的维护成本。
* 索引是放在磁盘的，更能的索引也意味着更多的存储空间。
* 数据重复且分布平均的字短没必要建立索引（比如：性别）

`索引并非银弹，正确使用才能发挥奇效。`

---

8. 索引什么时候会失效？

慢 SQL 是数据库使用中最长遇见的问题，当遇到慢 SQL 时，首先我们就要去看是不是索引失效。一般会有以下几种常见的情况：

1. **Where 条件中包含 OR**： 当查询条件中包含 OR，即使其中某些条件带有索引，也会全表扫描。下例中 username 没有索引，就算 id 走了索引也需要全表扫描，所以引擎大概率不会走索引。

失效索引： id 有索引， username 没有索引。

```
explain select * from t_user where id = 2 or username = 'javapub';
```

2. **多列索引没有最左匹配**： 对于复合索引，如果查询条件没有从索的第一部分匹配，则不会使用索引。也就是我们在使用联合索引时，要正确使用最左匹配。

例如，如果你有一个(id, name)的多列索引，但查询条件只使用了name，那么索引不会被使用。

3. **LIKE 查询以%开头**： 当使用LIKE操作符进行模糊查询，并且模式以%开头时，索引将不会生效。这是因为以%开头的模式匹配意味着匹配的字符串可以在任何位置，这使得索引无法有效定位数据。
4. **索引列参与计算**： 当我们在查询条件中对索引列进行表达式计算，也是无法走索引的。比如：

```
select * from t_user where id > age;
```

5. **类型不匹配导致隐式转换**： 当表里存的是 varchar 类型的字段时，用 int 类型去查询，导致全表扫描。如下例子中：

```
explain select * from t_user where id_no = 1002;
```

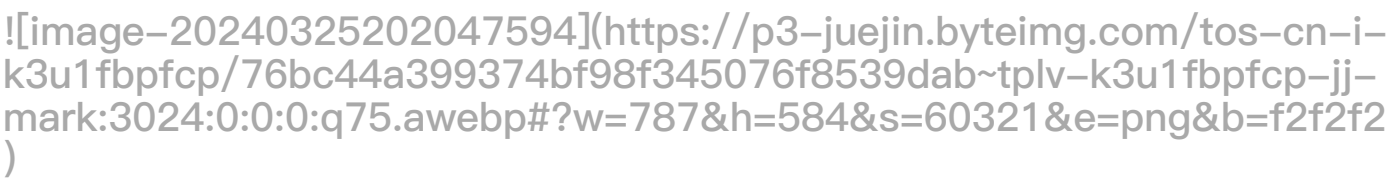
表里的 id_no 是 varchar 类型。

出了这几种情况还有一些导致索引失效。 例如：

* **全表扫描效率更优**：在某些情况下，MySQL 优 化器可能认为全表扫描比使用索引更快。
* **数据分布不均**：如果索引列的数据分布非常不均匀，MySQL 可能不会选择使用索引。
* **索引列包含 NULL 值**：如果索引列包含 NULL 值，MySQL可 能不会使用索引，因为 NULL 值的比较有特殊性。因为NULL值无法与其他值进行比较或匹配，所以无法使用索引。

---

9. 哪些情况下适合建立索引?



1. **高频查询列**： 对于经常出现在查询条件中的列，建立索引可以加快查询速度。例如，经常根据username或email字段查询的用户表。
2. **作为连接键的列**： 在执行表连接操作时，用于连接的列（通常在ON子句中指定）应该建立索引，以加快连接操作的速度。
3. **具有唯一性约束的列**： 对于需要保证唯一性的列，如主键或具有唯一约束的列，建立索引是必要的，因为索引可以帮助快速检查重复的数据。
4. **排序和分组操作的列**： 在ORDER BY、GROUP BY或DISTINCT操作中使用的列，通过建立索引可以加快排序和分组的处理速度。
5. **具有高选择性的列**： 选择性是指不同值的数量与总行数的比率。具有高选择性的列（即列中的值分布广泛）适合建立索引，因为这样的索引可以更有效地缩小搜索范围。
6. **多列查询的前导列**： 如果你经常执行涉及多个列的查询，可以在这些列上建立组合索引，其中最常用作查询条件的列应该放在索引的最前面。
7. **数据量大的表**： 对于数据量较大的表，合理地建立索引可以大幅提高查询效率。但是，对于数据量小的表，由于数据量本身就少，索引可能不会带来太大的性能提升，反而可能增加插入、更新和删除操作的开销。

在考虑建立索引时，也需要考虑以下因素：

* 更新频率：频繁更新的列可能不适合建立索引，因为每次更新都可能导致索引的重新构建，增加开销。
* 索引的维护成本：索引不仅占用存储空间，还会增加数据插入、删除和更新操作的维护成本。
* 查询类型：需要分析查询类型，确保索引能够被有效利用。例如，对于只读或几乎只读的表，建立索引可能没有太大必要。

---

10. 为什么要用 B+ 树，而不用二叉树?

* **查询性能稳定**： B+树通过多层索引结构，使得查询性能更加稳定。在最坏的情况下，B+树的查询时间复杂度仍然是对数级别（O(log n)），而二叉树在最坏情况下（退化成链表）的时间复杂度为线性（O(n)）。这意味着即使数据分布极不均匀，B+树也能保持较高的查询效率。
* **空间局部性**： B+树的叶子节点包含了所有数据记录，并且通过指针相互连接，形成了一个有序链表。这种结构使得范围查询和顺序访问更加高效，因为相邻的数据在物理存储上也是相邻的。而二叉树不具备这种空间局部性，数据的物理存储位置可能分散。
* **磁盘I/O优化**： 数据库操作经常涉及磁盘I/O，B+树的设计更适合减少磁盘访问次数。由于B+树的非叶子节点不存储实际数据，可以使得每个节点包含更多的键值，从而降低树的高度。这样，在一次磁盘I/O操作中可以读取更多的索引信息，减少了I/O次数。
* **高效的范围查询和排序**： B+树的有序链表结构使得它在执行范围查询和排序操作时非常高效。而二叉树需要进行中序遍历才能得到有序的结果，效率较低。
* **节点分裂和合并的开销**： 在二叉树中，插入和删除操作可能导致频繁的节点分裂和合并，增加了操作的复杂性。B+树通过减少节点分裂和合并的次数，降低了维护开销。
* **非叶子节点的简洁性**： B+树的非叶子节点仅用于索引，不存储实际数据，这样可以使得每个节点包含更多的键值对，进一步降低树的高度。
* **更新操作的效率**： 由于B+树的高度通常较低，更新操作（插入、删除）时需要遍历的节点数量较少，从而提高了更新操作的效率。

总的来说，B+树在数据库索引中提供了更稳定的查询性能、优化的磁盘I/O操作、高效的范围查询和排序，以及较低的维护成本。

---

11. 什么是回表？如何减少回表?

回表定义： MySQL回表查询是指在使用索引进行查询时，MySQL数据库引擎在通过索引定位到数据行后，发现需要访问表中的其他列数据，而不是直接通过索引就能获取到所需的数据。这种情况下，MySQL需要再次访问表中的数据行，这个过程就称为回表查询（Referring to the table）。

* **覆盖索引**： 覆盖索引是指一个查询可以完全通过索引来得到结果，而不需要访问数据表的行。如果查询只需要索引中包含的字段，那么就无需回表。设计良好的覆盖索引可以显著减少回表操作。
* **避免SELECT ***： 在编写查询时，尽量指定需要的列，而不是使用SELECT *来选择所有列。这样可以减少不必要的数据访问，从而减少回表。
* **索引包含所需列**： 确保查询中涉及的列都被包含在索引中。如果索引包含了所有需要的列，那么查询可能不需要回表。
* **使用复合索引**： 如果查询经常根据多个列进行过滤，可以考虑创建一个包含这些列的复合索引。这样可以在一个索引中完成查询，减少回表。
* **优化查询逻辑**： 分析查询逻辑，尽量减少不必要的回表操作。例如，如果查询中的某些条件不太可能同时满足，可以考虑将它们分开处理，或者使用临时表来存储中间结果。
* **使用物化视图或汇总表**： 对于频繁执行的复杂查询，可以考虑使用物化视图或汇总表来存储查询结果。这样，当需要这些数据时，可以直接从物化视图或汇总表中获取，而无需进行回表操作。

当然，不是所有情况都不允许回表，有时候，适当的回表是必要的，因为索引的设计需要平衡查询性能和存储空间的利用。

---

12. 能否解释什么是位图索引，以及它在MySQL中的使用场景?

位图索引是一种将数据列的所有可能值映射到二进制位上的索引。每个位表示某个值是否存在于该列中，从而帮助我们快速定位符合某个条件的行。与其他类型的索引相比，位图索引通常在低基数列（即列中有限的不同值）上表现更好。

> 可以参考 bitmap 数据结构来理解

例子：

在该示例中，我们为 age 和 country 列分别创建了位图索引。由于使用了位图索引，查询性能将大大提高。

```
CREATE TABLE users (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    age INT,
    country VARCHAR(50)
);

CREATE BITMAP INDEX idx_age ON users(age);
CREATE BITMAP INDEX idx_country ON users(country);

SELECT * FROM users WHERE age = 20 AND country = 'China';
```

---

13. 如何查看MySQL表中已有的索引?

两种方式：

* 使用 SHOW INDEX，也是最常用的。

```
SHOW INDEX FROM your_table_name;
```

* 查询 information_schema 数据库，information_schema 是 MySQL 中包含元数据的特殊数据库。我可以查询其中的 TABLES 和 STATISTICS 表来

获取索引信息。

```
SELECT
    TABLE_SCHEMA,
    TABLE_NAME,
    NON_UNIQUE,
    INDEX_NAME,
    INDEX_TYPE,
    INDEX_COMMENT,
    SEQ_IN_INDEX,
    COLUMN_NAME,
    CARDINALITY,
    SUB_PART,
    PACKED,
    NULLABLE,
    INDEX_DIR,
    INDEX_DISC
FROM
    information_schema.STATISTICS
WHERE
    TABLE_SCHEMA = 'your_database_name' AND
    TABLE_NAME = 'your_table_name';
```

---

14. 如何在MySQL中创建全文索引，并说明全文索引的使用场景?

正例:

```
CREATE TABLE articles (
    id INT NOT NULL AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    content TEXT NOT NULL,
    PRIMARY KEY (id),
    FULLTEXT INDEX (title, content) -- 创建联合全文索引
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

```

当已经建好表结构，使用 ALTER TABLE 创建：

```
ALTER TABLE articles
ADD FULLTEXT INDEX ft_index (title, content);
```

全文索引一般用于内容管理平台（CMS），问答社区等检索场景，然而，全文索引也有一些限制，比如它只能用于MyISAM或InnoDB存储引擎（在MySQL 5.6及以上版本中），并且全文索引的列不能是NULL值。

实际应用中其实很少会使用到，现在多数使用 ElasticSearch 来搭建全文搜索引擎。

---

### 15. 当表中的数据量非常大时，如何有效地维护和管理索引，以确保查询性能?

索引主要是为了优化查询性能而设计的。如果一个字段的查询频率远低于更新频率，那么为该字段创建索引可能不会带来预期的性能提升，反而可能因为维护索引而降低整体性能。

1. **性能开销**： 索引的维护需要额外的计算和存储资源。当对一个字段进行大量的更新操作时，数据库系统不仅需要更新数据本身，还需要更新所有相关的索引。这会导致性能开销增加，尤其是在高并发的写操作环境中。
2. **存储空间**： 索引本身占用存储空间。对于经常更新的字段，如果创建了索引，那么每次数据更新都可能导致索引的页面分裂，进而需要更多的存储空间来维护索引结构。
3. **索引失效**： 频繁的更新操作可能导致索引的页变得碎片化，从而降低索引的效率。索引页的碎片化意味着索引中的数据不再按照顺序存储，这会增加数据库在执行查询操作时的磁盘I/O次数，因为数据库可能需要读取多个不连续的页面来满足查询条件。
4. **更新锁竞争**： 在高并发的更新操作中，索引可能会成为锁竞争的瓶颈。当多个事务尝试更新同一索引页时，可能会发生锁等待，这会降低并发性能。
```

---

16. 假设你有一个包含大量数据的表，并且经常需要根据某个字段进行排序。你应如何优化这个字段的索引以提高排序操作的性能？

当你尝试为一个已经存在大量数据的表添加索引时，可能会遇到什么问题？如何解决这些问题？

首先：

如果是亿级大表，在建表时就要添加必要的索引，否则存入过多数据可能会出现加不成功的现象。

垂直拆分

按照业务维度拆分。

水平拆分

按照不同的行进行分片，分散到不同的物理表中。

创建索引

分区

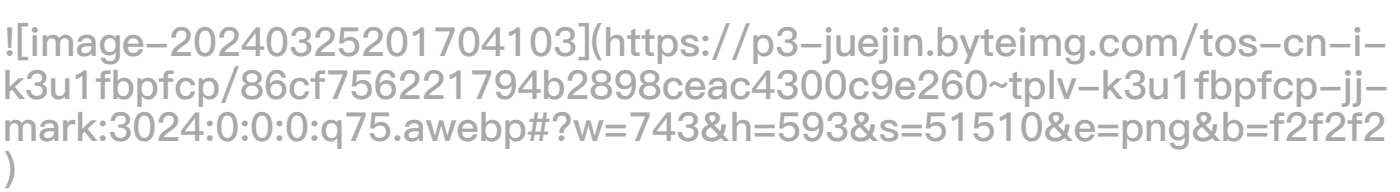
根据实际情况进行数据分区，但是要注意分区后可能影响写入性能。

优化查询语句

分布式数据库

---

17. 如何优化索引



当你遇到查询性能问题时，如何分析和优化索引的使用？开放性问题。

1. **评估索引的必要性**，不是所有字段都要走索引。
2. **选择正确的索引类型**，例如，B-tree索引适合范围查询和排序操作，Hash索引适合等值查询，Bitmap索引适合低基数（不同值的数量较少）的列。
3. **优化索引的列顺序**：在创建多列索引时，考虑列的访问模式和查询类型。通常，将最常用作查询条件的列放在索引的前面，因为数据库可以更有效地使用这些列来过滤数据。
4. **使用覆盖索引**：如果查询只访问索引中包含的列，使用覆盖索引可以避免访问数据行本身，从而提高查询性能。
5. **分析数据分布**：对于列的值分布进行分析，**避免在高度重复的列上创建索引**，因为这样的索引可能不会带来显著的性能提升。
6. **避免过度索引**：过多的索引会增加数据库的维护成本，尤其是在数据插入、更新和删除时。确保每个索引都有其明确的用途，并定期审查和清理不再需要的索引。

---

18. 请谈谈你对 MySQL 索引碎片化的理解，并说明如何检测和修复索引碎片化。

如何检测索引碎片化？两个方法

* **使用SHOW TABLE STATUS命令**： 通过执行 `SHOW TABLE STATUS LIKE 'table_name';` 可以获取表的状态信息，其中包括 Data_free 字段，它

表示表中未使用的空间百分比。如果这个值相对较高，可能表明表存在碎片化问题。
* **使用 INFORMATION_SCHEMA.TABLES 表**： 查询 `INFORMATION_SCHEMA.TABLES` 可以获取表的碎片化信息。例如：

```
SELECT table_name, table_schema, Data_free / Data_length * 100 AS碎片化百分比
FROM information_schema.TABLES
WHERE table_schema = 'your_database_name' AND Data_free > 0;
```

**如何修复索引碎片化？ **

* **优化表的存储引擎**：

对于 MyISAM 存储引擎，可以使用 `OPTIMIZE TABLE` 命令来重新组织表的数据，减少碎片化。对于 InnoDB 存储引擎，这个命令也会尝试优化表，但效果可能不如 MyISAM 明显。

```
OPTIMIZE TABLE table_name;
```

* **重建索引**：

对于 InnoDB 存储引擎，可以通过 `ALTER TABLE` 命令来重建表的索引，这通常比 OPTIMIZE TABLE 更有效。

```
ALTER TABLE table_name ENGINE=InnoDB;
```

* **定期维护**：

定期执行 `OPTIMIZE TABLE` 或 `ALTER TABLE` 命令可以帮助维持索引的健康状况，减少碎片化。

需要注意的是，优化表的操作可能会消耗大量的系统资源，并且可能需要较长的时间来完成，特别是对于大型表。因此，在执行这些操作之前，最好在测试环境中进行评估，并在业务低峰时段进行。此外，确保在执行优化操作之前备份数据，以防万一出现问题。

原文链接: https://juejin.cn/post/7350141012310327359