

MySQL进阶-索引与SQL优化

讲师 伊川

- 1.索引的概述与分类
- 2.索引原理-索引与B+Tree
- 3.慢查询与SQL优化

讲师 伊川

1.索引的概述与分类

什么是索引?

什么是索引?

索引类似图书的目录索引，可以提高数据检索的效率，降低数据库的IO成本。

MySQL在300万条记录左右性能开始逐渐下降，虽然官方文档说500~800w记录，

什么是索引？

索引类似图书的目录索引，可以提高数据检索的效率，降低数据库的IO成本。

MySQL官方对索引的定义为：

索引(Index)是帮助MySQL高效获取数据的数据结构。

我们可以简单理解为：快速查找排好序的一种数据结构。

所以在回答这类问题时，可以从两个地方说

1.索引类型图书的目录，可以提高数据检索的效率

2.索引其实就是一种排好序的数据结构。

关于数据结构的内容我们在下节课中将为大家详细讲解

索引的分类

索引的分类

- 主键索引
- 唯一索引
- 普通索引
- 全文索引
- 组合索引

索引的分类

主键索引

即主索引，根据主键 建立索引，**不允许重复，不允许空值；**

索引的分类

主键索引

即主索引，根据主键 建立索引，**不允许重复，不允许空值**；

```
-- 创建表时，直接创建主键索引
CREATE TABLE `users` (
  `id` int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

-- 修改时添加主键和自增
alter table users modify uid int primary key AUTO_INCREMENT;

-- 删除主键索引 注意需要先取消 自增,再删除主键
-- 先取消自增,修改字段
alter table users modify uid int;
-- 删除主键
alter table users drop primary key;
```

如果表中没有定义主键，InnoDB 会选择一个唯一的非空索引代替。

如果没有这样的索引，InnoDB 会隐式定义一个主键来作为聚簇索引

索引的分类

唯一索引

用来建立索引的列的值必须是**唯一的，允许空值**

索引的分类

唯一索引

用来建立索引的列的值必须是**唯一的**，允许空值

```
-- 创建表时，直接创建唯一索引
CREATE TABLE `users` (
  `name` VARCHAR(10) NOT NULL,
  UNIQUE KEY `name` (`name`),
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4

-- 添加唯一索引 UNIQUE 当前列要求唯一,但允许为空
alter table users add unique u_name(name);

-- 删除唯一索引 根据当前索引名去进行删除
alter table users drop index u_name;
```

索引的分类


普通索引

用表中的普通列构建的索引，没有任何限制

索引的分类

普通索引

用表中的普通列构建的索引，没有任何限制



```
-- 创建表时，直接创建普通索引
CREATE TABLE `users` (
  `email` varchar(10) NOT NULL,
  KEY `index_email` (`email`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4

-- 添加索引
alter table users add index in_name(email);

-- 删除索引
drop index in_name on users;
```

索引的分类


全文索引

用大文本对象的列构建的索引

索引的分类

全文索引

用大文本对象的列构建的索引




```
ALTER TABLE 'table_name' ADD FULLTEXT INDEX ft_index('col');
```

索引的分类

全文索引

用大文本对象的列构建的索引



```
ALTER TABLE 'table_name' ADD FULLTEXT INDEX ft_index('col');
```

5.6之前的版本中，全文索引只能用于MyISAM存储引擎

5.6 及以后的版本，MyISAM 和 InnoDB 均支持全文索引

在之前的MySQL中，全文索引只对英文有用，目前对中文还不支持

MySQL8的版本中支持了对中文分词的全文索引

为什么对中文不支持呢？因为中文和英文有本质上的区别。英文是单词语句组成，而且都有空格，好分辨，中文是有字组成的句，而且有不同的意思，因此不一样。当然目前中文分词技术已经非常成熟了，因此的mysql8种支持了中文的全文索引。

索引的分类


组合索引

用多个列组合构建的索引，这多个列中的值不允许有空值

索引的分类

组合索引

用多个列组合构建的索引，这多个列中的值不允许有空值



```
-- 添加索引
alter table users add index in_x(email,phone,uname);
-- 删除索引
alter table users drop index in_x;
```

索引的分类

组合索引的“最左”原则

索引的分类

组合索引的“最左”原则

```
-- 添加索引  
alter table users add index in_x(email,phone,uname);
```

email	phone	uname	✓
email	phone	uname	✓
email	phone	uname	✓
email	phone	uname	✗
email	phone	uname	✗
email	phone	uname	✗

索引概述与分类

总结

- 索引就像是一本书的目录是为了提高数据的检索速度
- 在MySQL中有不同的索引类型，要求和效率也各不一样

2.索引原理-索引与B+Tree

讲师 伊川

2.索引原理-索引与B+Tree

哈希索引

B+TREE索引

只有memory（内存）存储引擎支持哈希索引，哈希索引用索引列的值计算该值的hashCode，然后在hashCode相应的位置存储该值所在行数据的物理位置，因为使用散列算法，因此访问速度非常快，但是一个值只能对应一个hashCode，而且是散列的分布方式，因此哈希索引不支持范围查找和排序的功能。正常情况下，如果不指定索引的类型，那么一般是指B+Tree索引（或者B+Tree索引）。存储引擎以不同的方式使用B+Tree索引。性能也各有不同，但是InnoDB按照原数据格式进行存储。

2.索引原理-索引与B+Tree

那什么是 B+TREE？

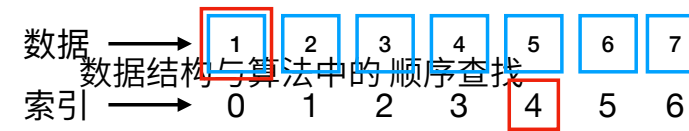
这？说来话长了。。。

不过呢，其实B+TREE 就是 B+树，而B+树呢就是通过B树演变过来的，所以如果想知道什么是B+树，那么就要先了解一下b树，要了解b树呢就得知道二叉树，这些都是数据结构的内容。

那数据结构又是啥呢？

数据结构是计算机存储、组织数据的方式。数据结构是指相互之间存在一种或多种特定关系的数据元素的集合。通常情况下，精心选择的数据结构可以带来更高的运行或者存储效率。数据结构往往同高效的检索算法和索引技术有关。

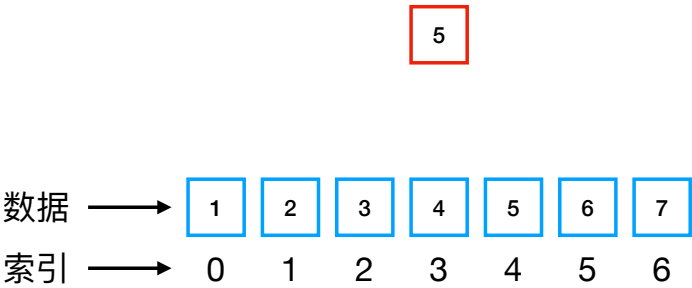
2.索引原理--了解数据结构



顺序查找：就是从第一个元素开始，按索引顺序遍历待查找序列，直到找出给定目标或者查找失败

缺点：效率低 -- 需要遍历整个待查序列

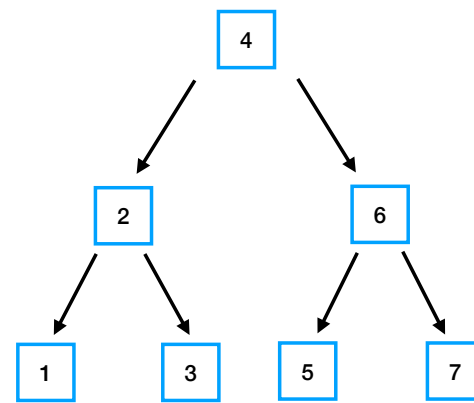
索引的实现原理-了解数据结构



二分法查找：也称为折半法，是一种在有序数组中查找特定元素的搜索算法。
如何实现二分查找呢？

索引的实现原理-了解数据结构

二叉树



索引的实现原理-了解数据结构

二叉树

根节点

4

子节点

2

6

叶子节点

1

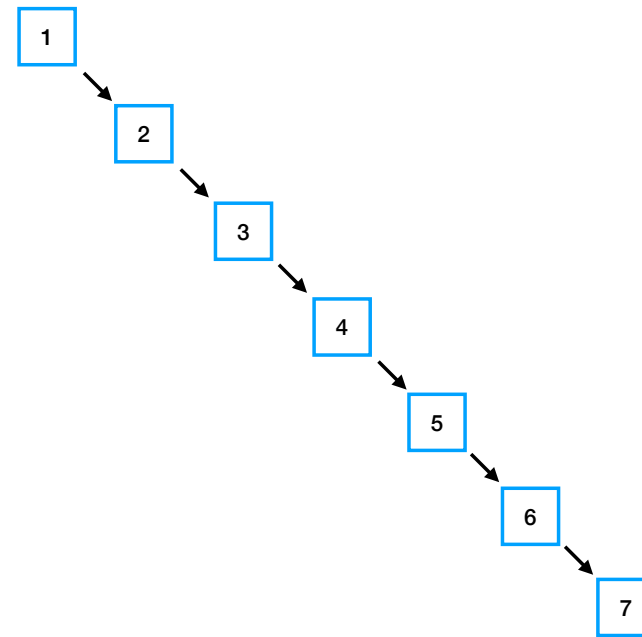
3

5

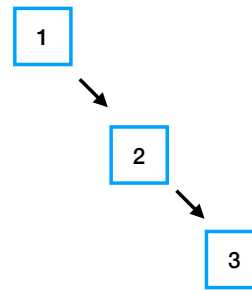
7

二叉树也存在缺点，就是当数据是顺序插入时就会改变树的形态（在非完全二叉树的时候）

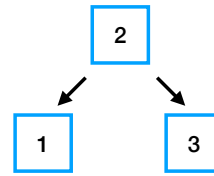
索引的实现原理-了解数据结构



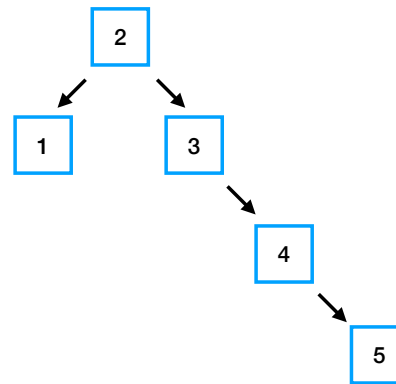
索引的实现原理-了解数据结构



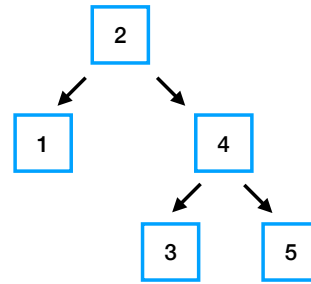
索引的实现原理-了解数据结构



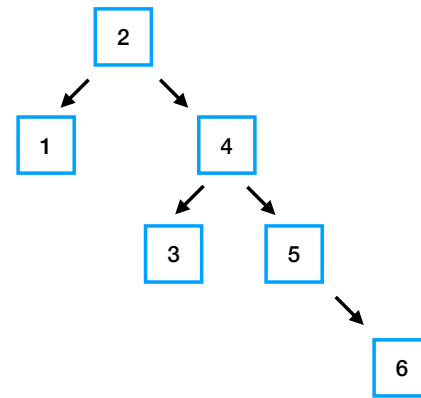
索引的实现原理-了解数据结构



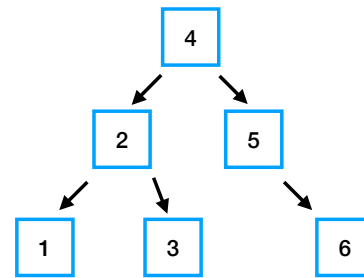
索引的实现原理-了解数据结构



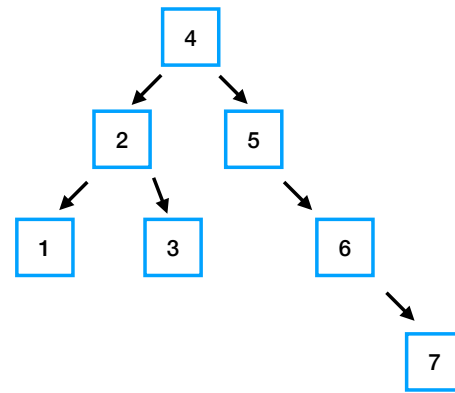
索引的实现原理-了解数据结构



索引的实现原理-了解数据结构

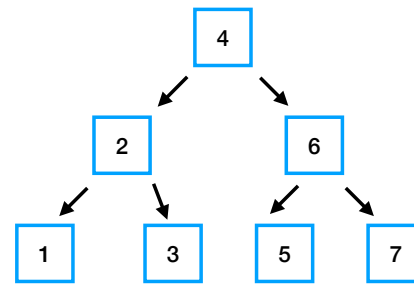


索引的实现原理-了解数据结构



索引的实现原理-了解数据结构

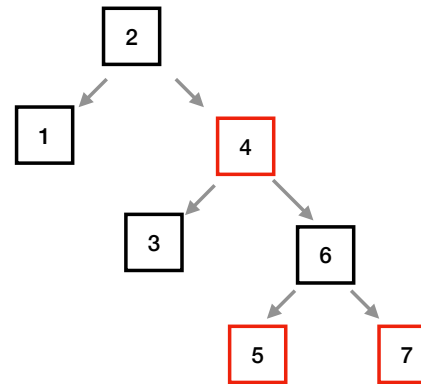
平衡二叉树



平衡二叉树经过条件的控制，再通过旋转的方式，完成树的平衡，不过，旋转次数过多。

索引的实现原理-了解数据结构

红黑树



在平衡二叉树稳定性的基础上，再优化一下，减少旋转次数，保证树的平衡性。

树的查找性能取决于树的高度，让树尽可能平衡，就是为了降低树的高度。

索引的实现原理-了解数据结构

当数据存在内存中，**红黑树**效率非常高，但是文件系统和数据库都是存在硬盘上的，如果数据量大的话，不一定能一次性加载到内存。

所以一棵树都无法一次性加载进内存，又如何谈查找。

因此就出现了专为磁盘等存储设备而设计的一种**平衡多路查找树**，也就是**B树**

与红黑树相比,在相同的的节点的情况下,一颗B/B+树的高度远远小于红黑树的高度

索引的实现原理-了解数据结构

B树即平衡查找树，一般理解为平衡多路查找树，也称为B-树、B_树。

B树是一种自平衡树状数据结构，一般较多用在存储系统上，比如数据库或文件系统。

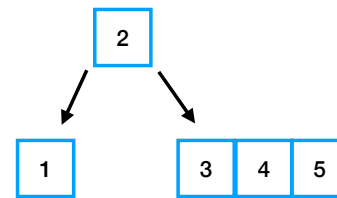
索引的实现原理-了解数据结构

一个3阶的B树

1	2	3
---	---	---

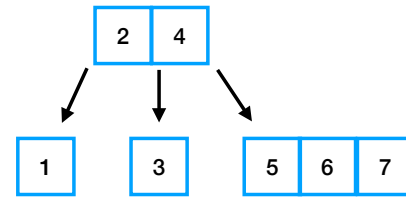
索引的实现原理-了解数据结构

一个3阶的B树



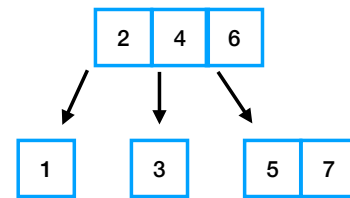
索引的实现原理-了解数据结构

一个3阶的B树



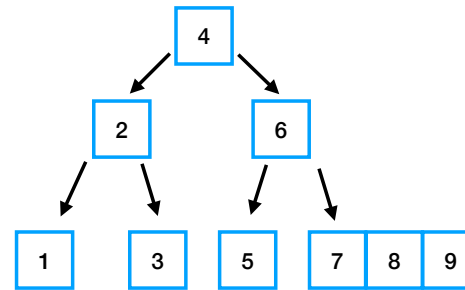
索引的实现原理-了解数据结构

一个3阶的B树



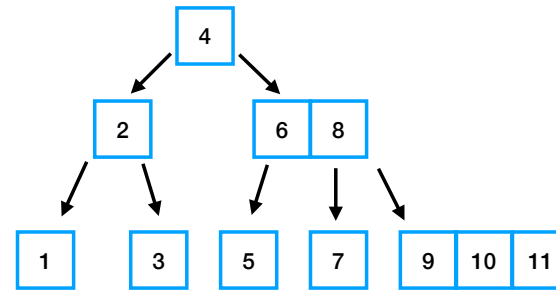
索引的实现原理-了解数据结构

一个3阶的B树



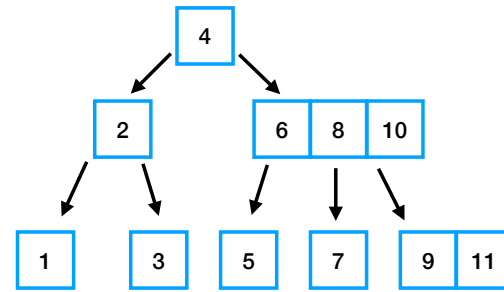
索引的实现原理-了解数据结构

一个3阶的B树



索引的实现原理-了解数据结构

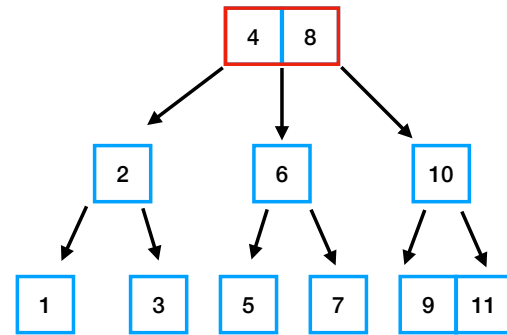
一个3阶的B树



索引的实现原理-了解数据结构

一个3阶的B树

假如在这棵树中查找数字7

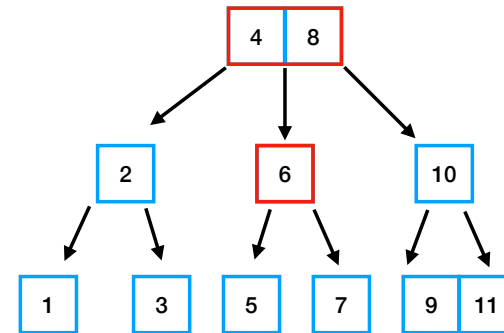


索引的实现原理-了解数据结构

一个3阶的B树

假如在这棵树中查找数字7 需要3步

假如在这棵树中查找数字10



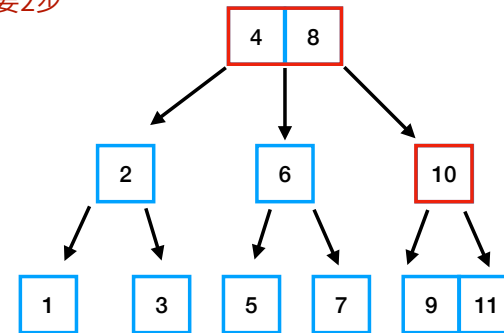
索引的实现原理-了解数据结构

一个3阶的B树

假如在这棵树中查找数字7 需要3步

假如在这棵树中查找数字10 需要2步

假如在需要在树中查找1-5



索引的实现原理-了解数据结构

一个3阶的B树

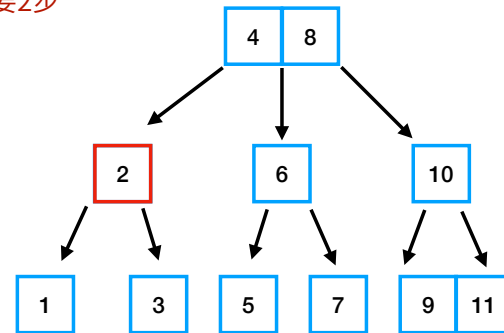
假如在这棵树中查找数字7 需要3步

假如在这棵树中查找数字10 需要2步

假如在需要在树中查找1-5

数字： 1 2 3 4 5

步数： 3 4 5 7 9



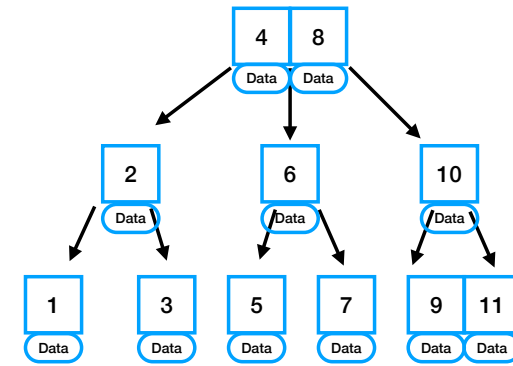
通过上面的三个案例就发现了几个问题

- 1.查找7和10 所用步数不一样，就意味着时间不同，效率就不均衡，有的可能快，有的可能慢
- 2.在进行范围查找时，需要反复的返回上一节点，在进入下一节点，这种情况其实是树的遍历，叫做中序遍历，消耗了时间
- 3.还有最重要的一点，在B树中，由于每一个节点就是一行数据，那么就是一次IO读区的节点更少。

索引的实现原理 - 了解数据结构

一个3阶的B树

1. 查找效率不均衡
2. 范围查找需要中序遍历
3. 每一个叶子节点上都带有数据

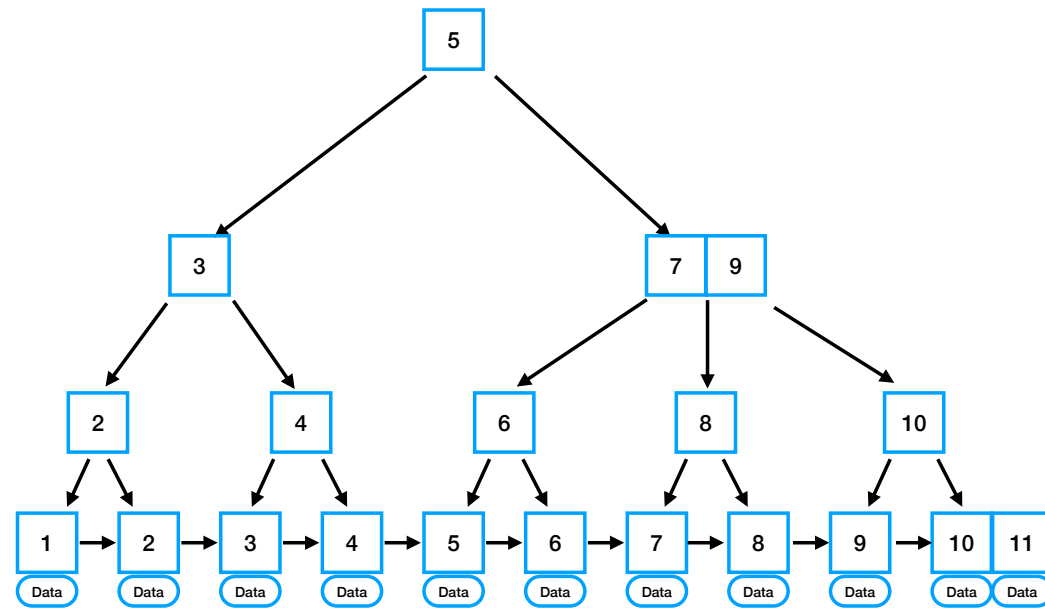


在计算机中,所有与空间相关的东西都是按照块(block)进行存取和操作的

每次读取都意味着一次I/O, 假设计算机中每个块的大小为4K, 行的大小为1k, 索引的大小为0.06K

如果需要寻址遍历的次数多, 就意味着更多的IO

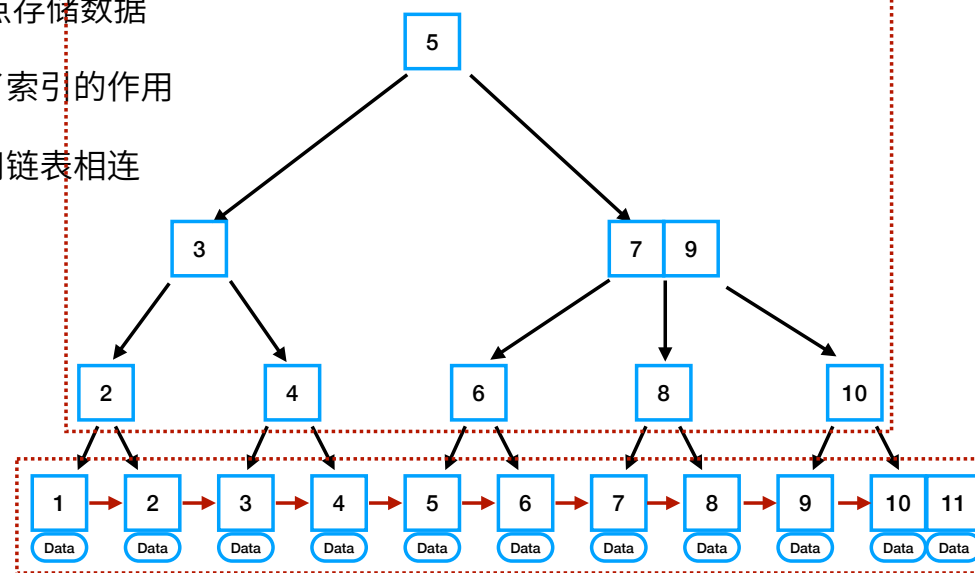
索引的实现原理-B+树



B+树是在B树的基础上作出的演变，那么大家可以看一下，B+树和B树的区别

索引的实现原理-B+树

- 1.B+树只有叶子节点存储数据
- 2.非叶子结点起到了索引的作用
- 3.所有叶子结点使用链表相连



索引的实现原理-B+树

1.磁盘读写代价更低

B树的数据和索引都在同一个节点上，那么每个块中包含的索引是少量的，如果想要取出比较深层的数据，意味着要读取更多的块，才能得到想要的索引和数据，那么就增加了IO次数

而B+树中每个块能存储的索引是B树的很多倍，那么获取比较深层的数据，也只需要读取少量的块就可以，那么就减少了磁盘的IO次数

2.随机IO的次数更少 B+树的优势是什么？

随机I/O是指读写操作时间连续，但访问地址不连续,时长约为 10ms。

顺序I/O是指读取和写入操作基于逻辑块逐个连续访问来自相邻地址的数据,时长约为 0.1ms

在相同情况下,B树要进行更多的随机IO,而B+树需要更多的顺序IO,因此B+树,效率也更快

3.查询速度更稳定

由于B+Tree非叶子节点不存储数据（data），因此所有的数据都要查询至叶子节点，而叶子节点的高度都是相同的，因此所有数据的查询速度都是一样的。

索引的实现原理-B+树

总结

在数据库中，索引是提高数据的检索速度的，而索引是基于B+Tree的数据结构实现的。

而使用B+Tree的好处是：

- 1.降低了磁盘读写代价
- 2.顺序I/O提高效率
- 3.查询速度更稳定

索引的实现原理-B+树

聚簇索引和非聚簇索引

索引又分为聚簇索引和非聚簇索引两种。

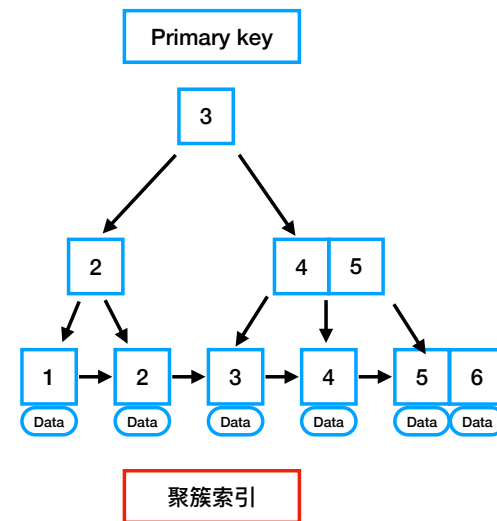
在索引的分类中，我们可以按照索引的键是否为主键来分为“主索引”和“辅助索引”

使用主键键值建立的索引称为“主索引”，其它的称为“辅助索引”。

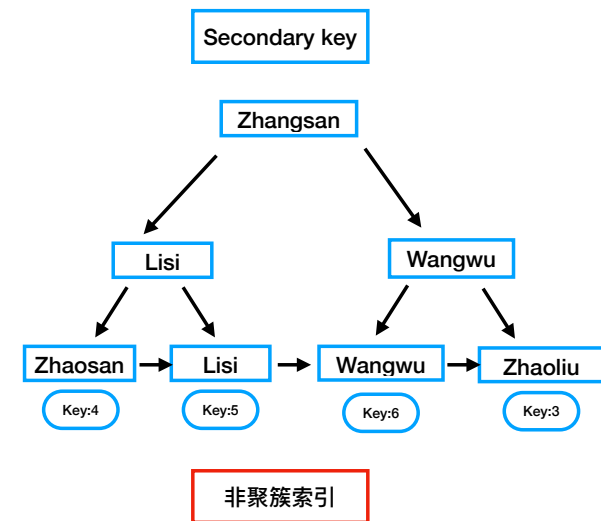
因此主索引只能有一个，辅助索引可以有很多个。

聚簇索引和非聚簇索引

InnoDB



索引即数据，数据即索引



找到索引仅仅是找到的当前索引值和key
如果需要索引外的内容，则需要回表

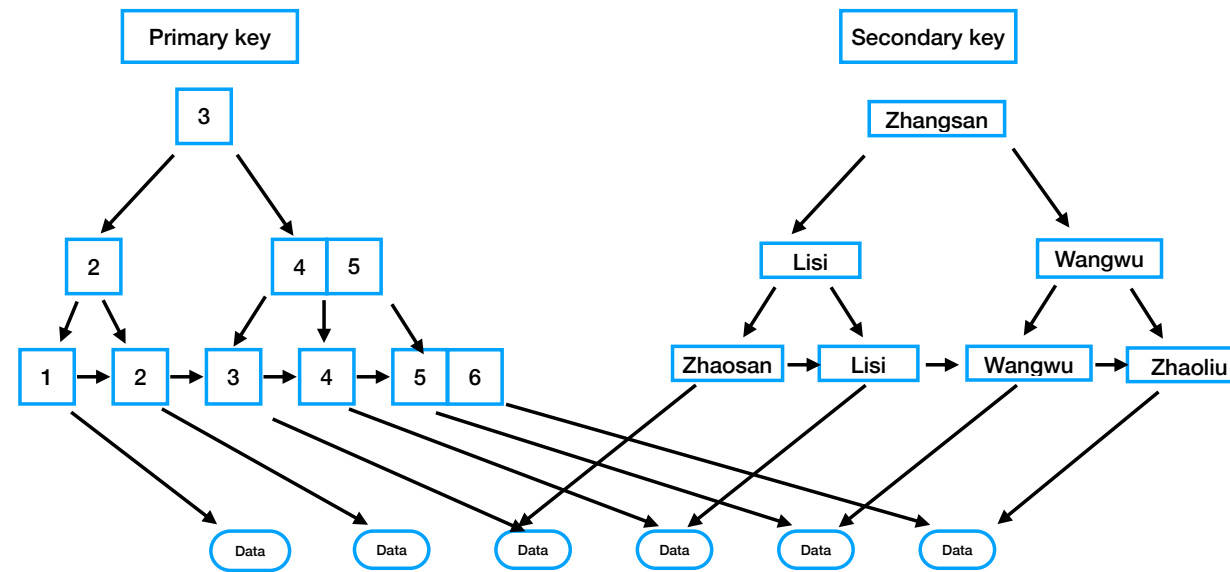
聚簇索引和非聚簇索引

以上关于索引原理和聚簇与非聚簇索引都是以InnoDB表引擎为基础

至此，我们介绍的都是InnoDB存储引擎中的索引方案，为了内容的完整性，以及各位可能在面试的时候遇到这类的问题，我们有必要再简单介绍一下MyISAM存储引擎中的索引方案。我们知道InnoDB中索引即数据，也就是聚簇索引的那棵B+树的叶子节点中已经把所有完整的用户记录都包含了，而MyISAM的索引方案虽然也使用树形结构，但是却将索引和数据分开存储：

聚簇索引和非聚簇索引

MyISAM



我们知道InnoDB中索引即数据，也就是聚簇索引的那棵B+树的叶子节点中已经把所有完整的数据都包含了，而MyISAM的索引方案虽然也使用树形结构，但是却将索引和数据分开存储：也就是把索引信息单独存到一个文件中，这个文件称为索引文件

MyISAM会单独为表的主键创建一个索引，只不过在索引的叶子节点中存储的不是完整的数据记录，而是主键值 + 行号的组合。也就是先通过索引找到对应的行号，再通过行号去找对应的记录！其它非主键索引也是一样的，这种情况我们称为‘回行’。所以在MyISAM中所有的索引都是非聚簇索引，也叫二级索引

聚簇索引和非聚簇索引

MyISAM和InnoDB的区别

- 数据存储方式：
 - InnoDB由两种文件组成，表结构，数据和索引
 - MyISAM由三种文件组成，表结构、数据、索引
- 索引的方式：
 - 索引的底层都是基于B+Tree的数据结构建立
 - InnoDB中主键索引为聚簇索引，辅助索引是非聚簇索引
 - MyISAM中数据和索引存在不同的文件中，因此都是非聚簇索引
- 事务的支持：
 - InnoDB支持事务
 - MyISAM不支持事务

索引的实现原理

总结

- 数据库的索引就是为了提高数据检索速度
- 而数据库的索引就是基于B+Tree的数据结构实现的
- 在InnoDB中主键是聚簇索引而辅助索引是非聚簇索引
- 在MyISAM中主键索引和辅助索引都是非聚簇索引

3.慢查询与SQL优化

讲师 伊川

在理解了MySQL中的索引类型及了解了索引的原理之后，我们就要知道索引是为了提高检索性能。那么如何更好的合理使用索引，并且对一些执行较慢的sql进行优化也是我们必须掌握的一种技能，因此本节课，我们结合索引的类型和原理，来一起看一下关于慢查询与SQL优化

3.慢查询与SQL优化

MySQL的慢查询，全名是慢查询日志，
是MySQL提供的一种日志记录，用来记录在MySQL中响应时间超过阈值的语句。
默认情况下，MySQL数据库并不启动慢查询日志，需要手动来设置这个参数。
如果不是调优需要的话，一般不建议启动该参数，开启慢查询日志会或多或少带来一定的性能影响。

慢查询是什么？

慢查询日志可用于查找需要很长时间才能执行的查询，因此是优化的候选者。

3.慢查询与SQL优化

```
-- // 查看“慢查询”的配置信息
mysql> show variables like '%slow%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| log_slow_admin_statements | OFF |
| log_slow_slave_statements | OFF |
| slow_launch_time | 2 |
| slow_query_log | OFF |
| slow_query_log_file | /usr/local/var/mysql/chuangede-MacBook-Pro-slow.log |
+-----+-----+

-- // 查看“慢查询”的时间定义
mysql> show variables like 'long_query_time';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| long_query_time | 10.000000 |
+-----+-----+

-- //设置“慢查询”的时间定义
set long_query_time=2;
-- //开启慢日志
set global slow_query_log='ON';
```

3.慢查询与SQL优化

```
-- 利用存储过程，插入一千万条数据
mysql> call p1();
Query OK, 0 rows affected (20 min 53.16 sec)
-- 查看数据
select count(id) from users;
+-----+
| count(id) |          创建表，插入测试数据
+-----+
| 10000000 |
+-----+
1 row in set (2.18 sec)
-- 修改部分数据后进行查询
select * from users where name = 'zhangsan';
+-----+-----+-----+-----+-----+-----+
| id      | name    | email                | phone      | age | sex |
+-----+-----+-----+-----+-----+-----+
| 1000001 | zhangsan | zhangsan@qq.com      | 13701383017 | 25  | 女  |
+-----+-----+-----+-----+-----+-----+
1 row in set (3.73 sec)
```

3.慢查询与SQL优化



```
# Time: 2020-06-15T08:54:40.150701Z
# User@Host: root[root] @ localhost: [未指定]
# Query_time: 3.711172  Lock_time: 0.000267 Rows_sent: 1  Rows_examined: 10000000
SET timestamp=1592211280;
select * from users where name = 'zhangsan';
```

查看慢查询日志

3.慢查询与SQL优化

一条查询语句在经过MySQL查询优化器的各种基于成本和规则的优化会后生成一个所谓的执行计划

这个执行计划展示了接下来具体执行查询的方式，比如多表连接的顺序是什么，对于每个表采用什么访问方法来具体执行查询等等。

MySQL为我们提供了EXPLAIN语句来帮助我们查看某个语句的具体执行计划。

3.慢查询与SQL优化

```
mysql> select * from users where name = 'zhangsan';
+-----+-----+-----+-----+-----+-----+
| id      | name    | email          | phone    | age  | sex  |
+-----+-----+-----+-----+-----+-----+
| 1000001 | zhangsan | zhangsan@qq.com | 13701383017 | 25  | 女   |
+-----+-----+-----+-----+-----+-----+
1 row in set (3.72 sec)

mysql> explain select * from users where name = 'zhangsan'\G;
***** 1. row *****
      id: 1
    select_type: SIMPLE
        table: users
    partitions: NULL
         type: ALL      访问方法
possible_keys: NULL    可能用到的索引
          key: NULL     实际上使用的索引
        key_len: NULL
          ref: NULL
         rows: 9750125  预计需要扫描的行数
    filtered: 10.00
      Extra: Using where
```

id 在一个大的查询语句中每个SELECT关键字都对应一个唯一的id

select_type SELECT关键字对应的那个查询的类型

table 表名

partitions 匹配的分区信息

type 针对单表的访问方法

possible_keys 可能用到的索引

key 实际上使用的索引

key_len 实际使用到的索引长度

ref 当使用索引列等值查询时，与索引列进行等值匹配的对象信息

rows 预估的需要读取的记录条数

filtered 某个表经过搜索条件过滤后剩余记录条数的百分比

Extra 一些额外的信息

3.慢查询与SQL优化

```
mysql> select * from users where id = 1000001;
+-----+-----+-----+-----+-----+-----+
| id    | name  | email      | phone   | age  | sex  |
+-----+-----+-----+-----+-----+-----+
| 1000001 | zhangsan | zhangsan@qq.com | 13701383017 | 25 | 女 |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> explain select * from users where id = 1000001\G;
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: users
   partitions: NULL
         type: const
possible_keys: PRIMARY
          key: PRIMARY
        key_len: 4
         ref: const
         rows: 1
   filtered: 100.00
    Extra: NULL
```

3.慢查询与SQL优化

```
-- 添加普通索引
mysql> alter table users add index index_name(name);
Query OK, 0 rows affected (26.09 sec)
-- 执行sql
mysql> select * from users where name = 'zhangsan';
```

id	name	email	phone	age	sex
1000001	zhangsan	zhangsan@qq.com	13701383017	25	女

1 row in set (0.01 sec)

3.慢查询与SQL优化

给name字段添加索引

```
-- 添加索引后, 执行Explain
mysql> explain select * from users where name = 'zhangsan'\G;
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: users
    partitions: NULL
           type: ref
possible_keys: index_name
           key: index_name
        key_len: 122
           ref: const
          rows: 1
   filtered: 100.00
      Extra: NULL
```


3.慢查询与SQL优化

大家看到，索引能给数据检索提高效率非常明显

那么是否意味着我们只要尽可能多的去建立索引就可以了呢？

每建立一个索引都会建立一棵B+树，并且需要维护，这是很费性能和存储空间的。

3.慢查询与SQL优化

适当建立索引
合理使用索引

3.慢查询与SQL优化

适当建立索引

- 1.创建并使用自增数字来建立主键索引
- 2.经常作为where条件的字段建立索引
- 3.添加索引的字段尽可能的保持唯一性
- 4.可考虑使用联合索引并进行索引覆盖

联合索引的索引覆盖 (多个字段组合成了一个联合索引,在查询时,所要的字段和查询条件中的索引是一致)

注意索引绝不是加的越多越好(1.索引会占空间. 2.索引会影响写入性能)

3.慢查询与SQL优化

合理使用索引

MySQL 索引通常是被用于提高 WHERE 条件的数据行匹配时的搜索速度，

在索引的使用过程中，存在一些使用细节和注意事项。

因为不合理的使用可能会导致建立了索引之后,不一定就使用上了索引

3.慢查询与SQL优化 合理使用索引



```
-- 不要在列上使用函数，这将导致索引失效而进行全表扫描。  
select * from news where year(publish_time) = 2017  
  
-- 改造为了使用索引，防止执行全表扫描，可以进行改造。  
select * from news where publish_time = '2017-01-01'  
1.不要在列上使用函数和进行运算  
  
-- 不要在列上进行运算，这也将导致索引失效而进行全表扫描。  
select * from news where id / 100 = 1  
  
-- 改造 为了使用索引，防止执行全表扫描，可以进行改造。  
select * from news where id = 1 * 100
```

3.慢查询与SQL优化 合理使用索引

```
mysql> select * from users where id / 10000 = 1;
+-----+-----+-----+-----+-----+-----+
| id    | name    | email          | phone    | age  | sex  |
+-----+-----+-----+-----+-----+-----+
| 10000 | user:10000 | user:10000@qq.com | 13701624809 | 71  | 男   |
+-----+-----+-----+-----+-----+-----+
1 row in set (5.20 sec)
```

```
mysql> explain select * from users where id / 10000 = 1 \G;
***** 1. row *****
           id: 1
    select_type: SIMPLE
           table: users
        partitions: NULL
                type: ALL
possible_keys: NULL
            key: NULL
         key_len: NULL
            ref: NULL
           rows: 9750125
    filtered: 100.00
      Extra: Using where
```

3.慢查询与SQL优化 合理使用索引

```
-- 修改表中的数据
mysql> update users set name = '123456' where id = 10086;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

-- 使用正常查询
mysql> select * from users where name = '123456';
+-----+-----+-----+-----+-----+-----+
| id | name | email | phone | age | sex |
+-----+-----+-----+-----+-----+-----+
| 10086 | 123456 | user:10086@qq.com | 13701584330 | 79 | 女 |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

-- 出现了隐式转换
mysql> select * from users where name = 123456;
+-----+-----+-----+-----+-----+-----+
| id | name | email | phone | age | sex |
+-----+-----+-----+-----+-----+-----+
| 10086 | 123456 | user:10086@qq.com | 13701584330 | 79 | 女 |
+-----+-----+-----+-----+-----+-----+
1 row in set, 65535 warnings (5.65 sec)
```

当查询条件左右两侧类型不匹配的时候会发生隐式转换，隐式转换带来的影响就是可能导致索引失效而进行全表扫描。

3.慢查询与SQL优化 合理使用索引

```
-- 当在尾部使用通配符时可以使用索引
mysql> select * from users where name like 'zhang%';
+-----+-----+-----+-----+-----+-----+
| id    | name   | email          | phone    | age  | sex  |
+-----+-----+-----+-----+-----+-----+
| 1000001 | zhangsan | zhangsan@qq.com | 13701383017 | 25 | 女 |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

-- 当在头部(左侧)使用通配符时, 则导致索引失效
mysql> select * from users where name like '%zhang%';
+-----+-----+-----+-----+-----+-----+
| id    | name   | email          | phone    | age  | sex  |
+-----+-----+-----+-----+-----+-----+
| 1000001 | zhangsan | zhangsan@qq.com | 13701383017 | 25 | 女 |
+-----+-----+-----+-----+-----+-----+
1 row in set (4.61 sec)
```


3.慢查询与SQL优化 合理使用索引

1.多个单列索引并不是最佳选择

MySQL 只能使用一个索引，会从多个索引中选择一个限制最为严格的索引，因此，为多个列创建单列索引，并不能提高 MySQL 的查询性能。

4.复合索引的使用(联合、组合)

- 3.慢查询与SQL优化 合理使用索引
- 4.复合索引的使用(联合、组合)
- 1.多个单列索引并不是最佳选择

```
mysql> desc users;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
name	varchar(30)	NO	MUL	NULL	
email	varchar(30)	YES	MUL	NULL	
phone	char(11)	YES	MUL	NULL	
age	int(11)	YES		NULL	
sex	char(1)	YES		NULL	

- 3.慢查询与SQL优化 合理使用索引
- 4.复合索引的使用(联合、组合)
- 1.多个单列索引并不是最佳选择

```
mysql> desc select * from users
where name = 'zhangsan' and email = 'zhangsan@qq.com' and phone = '13701383017'\G;
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: users
partitions: NULL
      type: ref
possible_keys: index_name,index_email,index_phone
              key: index_name
      key_len: 122
        ref: const
        rows: 1
   filtered: 5.00
      Extra: Using where
```

users.frm	9 KB	文稿
users.ibd	1.72 GB	文稿

事实上，MySQL 只能使用一个单列索引。这样既浪费了空间，又没有提高性能（因为需要回行）
为了提高性能，可以使用复合索引保证列都被索引覆盖。

3.慢查询与SQL优化 合理使用索引

4.复合索引的使用(联合、组合)

1.多个单列索引并不是最佳选择

MySQL 在进行查询分析时，会从多个索引中选择一个限制最为严格的索引，因此，为多个列创建单列索引，并不能提高MySQL的查询性能。

```
-- 先删除三个列的索引
mysql> alter table users drop index index_name;
mysql> alter table users drop index index_email;
mysql> alter table users drop index index_phone;
mysql> alter table users add index index_name_email_phone(name,email,phone);
mysql> desc select * from users
where email = 'zhangsan@cs.com' and name = 'zhangsan' and phone = '13701383017'\G;
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: users
  partitions: NULL
        type: ref
possible_keys: index_name_email_phone
           key: index_name_email_phone
        key_len: 290
           ref: const,const,const
          rows: 1
     filtered: 100.00
        Extra: NULL
```

3.慢查询与SQL优化

合理使用索引

4.复合索引的使用(联合、组合)

2.复合索引的最左前缀原则

查询条件中使用了复合索引的第一个字段，索引才会被使用。因此，在复合索引中索引列的顺序至关重要。如果不是按照索引的最左列开始查找，则无法使用索引。

3.慢查询与SQL优化

合理使用索引

4.复合索引的使用(联合、组合)

2.复合索引的

查询条件中使用
关重要。如果不

```
mysql> select * from users where email = 'zhangsan@qq.com';
+-----+-----+-----+-----+-----+-----+
| id    | name  | email          | phone   | age | sex |
+-----+-----+-----+-----+-----+-----+
| 1000001 | zhangsan | zhangsan@qq.com | 13701383017 | 25 | 女 |
+-----+-----+-----+-----+-----+-----+
1 row in set (3.82 sec)

mysql> desc select * from users where email = 'zhangsan@qq.com'\G;
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: users
  partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
        key_len: NULL
         ref: NULL
        rows: 9750125
   filtered: 10.00
    Extra: Using where
```

引列的顺序至

3.慢查询与SQL优化

合理使用索引

4.复合索引的使用(联合、组合)

3.尽可能达成索引覆盖

如果一个索引包含所有需要的查询的字段值，直接根据索引的查询结果返回数据，而无需读表，能够极大的提高性能。因此，可以定义一个让索引包含的额外的列，即使这个列对于索引而言是无用的。

3.慢查询与SQL优化

1. SQL语句的优化

1. 避免嵌套语句(子查询)
2. 避免多表查询(复杂查询简单化)

2. 索引优化

1. 适当建立索引
2. 合理使用索引

MySQL进阶-索引与SQL优化

- 索引的概述与分类
- 索引原理-索引与B+Tree
- 慢查询与SQL优化

讲师：伊川