1. MySQL架构与历史

1.1. MySQL逻辑架构

1.1.1. 连接管理与安全性

1.1.2. 优化与执行

1.2. 并发控制

1.2.1. 读写锁

1.2.2. 锁粒度

1.3. 事务

1.3.1. 隔离级别

1.3.2. 死锁

3.6. 总结

4. Schema与数据类型优化

根据系统将要执行的查询语句来设计Schema。

4.1. 选择优化的数据类型

4.1.0.1. 更小的通常更好

应尽量使用可以正确存储数据的最小数据类型。

确保没有低估需要存储的值得范围。（在Schema中的多个地方增加数据类型的范围是一个非常耗时和痛苦的工作）

4.1.0.2. 简单就好

简单数据类型的操作通常需要更少的CPU周期。

用整型存储IP地址。

4.1.0.3. 尽量避免NULL

可为NULL的列使得索引、索引统计和值比较都会变得复杂。

InnoDB使用单独的位（bit）存储NULL值，所以对稀疏数据有很好的空间效率。但是不适用于MyISAM。

TIMESTAMP只使用DATETIME一半的存储空间，并且会根据时区变化。但是允许的时间范围小。

4.1.1. 整数类型（TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT）

整数计算一般使用64位的BIGINT整数，即使在32位环境也是如此。 MySQL可以为整数指定宽度，例如INT(11)（不会限制值的合法范围，只是规定了交互工具用来显示字符的个数）。

4.1.2. 实数类型

定义：实数是带有小数部分的数字。但是不只是为了存储小数，也可以使用DECIMAL存储比BIGINT还大的整数。

1. MySQL既支持精确类型，也支持不精确类型。
2. CPU支持原生浮点类型，不支持DECIMAL类型，所以原生浮点类型运算更快。
3. DECIMAL可以指定小数点前后允许的最大位数，例如DECIMAL(18,9)。（每4个字节存9个数字，小数点本身占一个字节）。
4. MYSQL5.0后DECIMAL允许最多65个数字。
5. 在计算时DECIMAL会转换为DOUBLE类型。
6. 应尽量只在对小数进行精确计算时才使用DECIMAL。

4.1.3. 字符串类型

4.1.3.1. VARCHAR和CHAR类型

4.1.3.1.1. VARCHAR

1. 如果MySQL表使用ROW_FORMAT=FIXED创建的话，每一行都会使用定长存储。
2. VARCHAR需要使用1或2个额外字节记录字符串的长度。（根据列的长度大于或小于255个字节区分）
3. InnDB可以把过长的VARCHAR存储为BLOB。

4.1.3.1.2. CHAR（空格进行填充）

1. 当存储CHAR值时，MySQL会删除所有的末尾空格。
2. 优点：定长的CHAR类型不容易产生碎片。
3. 字符串长度定义不是字节数，是字符数，多字节字符集会需要更多的空间存储单个字符。

4. 填充和截取空格的行为在不同的存储引擎都是一样的，因为这是在MySQL服务器层处理的。
5. BINARY和VARBINARY存储二进制字符串，二进制字符串存储的是字节码而不是字符。（\0进行填充，检索时不会去掉）

4.1.3.1.3. BLOB和TEXT类型

1. 为存储很大的数据而设计的字符串数据类型，分别采用二进制和字符方式存储。
2. 只对每个列最前max_sort_length字节而不是整个字符串做排序。

如果EXPLAIN执行计划的Extra列包含“Using temporary”，则说明这个查询使用了隐式临时表。

4.1.3.1.4. 使用枚举(ENUM)代替字符串类型。

1. MySQL会在内部将每个值在列表中的位置保存为整数，并且在表的.frm文件中保存“数字-字符串”映射关系的“查找表”。
2. 枚举字段是按照内部存储的整数而不是定义的字符串进行排序的。（按照需要的方式定义枚举列）
3. 对于一系列未来会改变的字符串，使用枚举不是一个好主意。

4.1.4. 日期和时间类型

除了特殊行为之外，通常应该尽量使用TIMESTAMP,因为它比DATATIME空间效率更高。

4.1.5. 位数据类型

4.1.5.1. BIT

数字上下文场景异常，谨慎使用BIT类型。

4.1.5.2. SET

SET数据类型在MySQL内部是以一系列打包的位的集合来表示的。 一个替代SET的方式是使用一个整数包装一系列的位。

4.1.6. 选择标识符（identifier）

在比较操作室隐式类型转换可能导致很难发现的错误。

随机值导致缓存对所有类型的查询语句效果都很差。

对象关系映射（orm）系统是一种常见的性能噩梦。

4.1.7. 特殊类型数据

ipv4：MySQL提供INET_ATON()和INET_NTOA()函数在点分法字符串和无符号整数之间转换。

4.2. MySQl schema设计中的陷阱

4.2.0.1. 太多的列

MySQL存储引擎API工作时需要在服务器层和存储引擎层之间通过行缓冲格式拷贝数据，然后在服务器层将缓冲内容解码成各个列。从行缓冲中将编码过的列转换成行数据结构的操作代价是非常高的。转换的代价依赖于列的数量。

4.2.0.2. 太多的关联

4.2.0.3. 全能的枚举

应该用整数作为外键关联到字典表或者查找表来查找具体值。（防止表结构的修改）

4.2.0.4. 变相的枚举

如果每次只会出现一个值（Y或N），应使用枚举代替SET。

4.2.0.5. 非此发明（Not Invent Here）的NULL

1. 处理NULL确实不容易，但有时会比他的替代方案更好。
2. MySQL会在索引中存储NULL值，Oracle不会。

4.3. 范式和反范式

4.3.1. 范式的优点和缺点

优点：范式的表通常更小，可以更好地方在内存里，所以执行操作会更快。

缺点：通常需要关联。

4.3.2. 反范式的优点和缺点

单独的表能使用更高效的索引策略。

4.3.3. 混用范式化和反范式化

在不同的表中存储相同的特定列。

排序的需要。

4.4. 缓存表和汇总表

4.4.1. 物化视图

4.4.2. 计数器表

4.5. 加快ALTER TABLE操作的速度

1. 大部分ALTER TABLE操作将导致MySQL服务中断。
2. ALTER TABLE允许使用ALTER COLUMN、MODIFY COLUMN和CHANGE COLUMN语句修改列。这三种操作都是不一样的。
3. 影子拷贝。（通过重命名和删表交换两张表）

4.5.1. 只修改.frm文件

执行FLUSH TABLES READ LOCK会关闭所有正在使用的表，并且禁止任何表被打开。

执行UNLOCK TABLE释放读锁。

4.5.2. 快速创建MyISAM索引

为了高效地载入数据到MyISAM表中，先禁用索引、载入数据，然后重新启用索引。

```
mysql> ALTER TABLE test.load_data DISABLE KEYS;
--load the data
mysql> ALTER TABLE test.load_data ENABLE KEYS;
```

DISABLE KEYS只对非唯一索引有效。

InnoDB版本：先删除所有的非唯一索引，然后增加新的列，最后重新创建删除掉的索引。

4.6. 总结

避免使用MySQL已经遗弃的特性，例如指定浮点数的精度，或者整数的显示宽度。

5. 创建高性能索引

背景：不恰当使用索引，当数据量逐渐增大时，性能会急剧下降。

5.1. 索引基础

如果索引包含多个列，那么列的顺序也十分重要，因为MySql只能高效地使用索引的最左前缀列。 **Tip：关于ORM框架：除非只是生成非常基本的查询（例如仅是根据主键查询），否则很难生成适合索引的查询。**

5.1.1. 索引的类型

在MySQL中，索引是在引擎层而不是在服务器层实现的。（并不是所有的引擎都支持索引）

5.1.1.1. B-Tree索引

Tip：很多存储引擎都是用的B+Tree，即每一个叶子节点都包含指向下一个叶子节点的指针。 MyISAM索引通过数据的物理位置引用被索引的行；InnoDB则根据主键引用被索引的行（所以二级索引需要两次索引查找）。

5.1.1.1.1. 可以使用B-Tree索引的查询类型

1. 全值匹配
2. 匹配最左前缀
3. 匹配列前缀
4. 匹配范围值
5. 精确匹配某一列并范围匹配另外一列
6. 只访问索引的查询

Tip：如果不是按照索引的最左列开始查找，则无法使用索引。 不能跳过索引中的列。 如果查询中有某个列的范围查询，则其右边所有列都无法使用索引优化查找。

5.1.1.2. 哈希索引

1. 哈希索引只包含哈希值和行指针，而不存储字段值。
2. 哈希索引数据并不是按照索引值顺序存储的，所以也就无法用于排序
3. 哈希索引也不支持部分索引列匹配查找。
4. 哈希索引只支持等值比较查询。
5. 哈希冲突，遍历链表。

InnoDB存储引擎有一个特殊的功能：自适应哈希索引(adaptive hash index),当某些索引值用的非常频繁时，在内存中基于B-Tree索引之上再创建一个Hash索引。

不要使用SHA1()和MD5()作为哈希函数（太长，浪费空间），这两个是强加密函数，设计目标是最大限度消除冲突。

使用MD5()返回值的一部分来作为自定义哈希函数。

5.1.1.3. 空间数据索引

5.1.1.4. 全文索引

全文索引适用于MATCH AGAINST操作，而不是普通的WHERE条件操作。

5.2. 索引的优点

因为索引中存储了实际的列值，所以某些操作只使用索引就能完成全部的查询。

1. 索引大大减少了服务器需要扫描的数据量。
2. 索引可以帮助服务器避免排序和临时表。
3. 索引可以将随机I/O变为顺序I/O。

TB级别用块级别元数据急速代替索引。

5.3. 高性能的索引策略

5.3.1. 独立的列

如果查询中的列不是独立的，那么MySQL就不会使用索引。

5.3.2. 前缀索引和索引选择性

通常可以索引开始的部分字符，这样可以大大节约索引空间，从而提高索引效率。

对于BLOB、TEXT或者很长的VARCHAR类型的列，必须使用前缀索引，因为MySQL不准索引这些列的完整长度。

决定前缀的合适长度

1. 找到常见的值的列表，然后和最常见的前缀列表比较。
2. 计算完整列的选择性，并使前缀列的选择性接近完整列的选择性。

5.3.3. 多列索引

5.3.4. 选择合适的索引列排序

5.3.5. 聚簇索引

聚簇索引并不是一种单独的索引类型，而是一种存储方式。 因为是存储引擎负责索引，所以不是所有的存储引擎都是聚簇索引。 InnoDB只聚集在同一个页面中的记录，所以包含相邻键值的页面可能会相距甚远。 基于聚簇索引的表在插入新行，或者在主键被更新需要移动行的时候，可能面临页分裂（'page split'）的问题，页分裂会导致表占用更多的磁盘空间。

5.3.5.1. ）InnoDB和MyISAM的数据分布对比

5.3.5.1.1. MyISAM的数据分布

按照数据插入顺序存储在磁盘上。

5.3.5.1.2. InnoDB的数据分布

聚簇索引就是表。 InnoDB二级索引存储主键值的好处？减少出现行移动或者数据页分列时二级索引的维护工作。 B-Tree的非叶子节点包含了索引列和指向下一级节点的指针。

5.3.5.2.) 在InnoDB中按主键顺序插入行

如果正在使用InnoDB表并且没有数据需要聚集，可以定义一个代理键（surrogate key）作为主键。 example：AUTO_INCREMENT自增列

WARN:必须保证能够顺序写入。（不要使用UUID，会导致页分裂和碎片）

页的最大填充因子（InnoDB默认为15/16，留出部分空间用于以后的修改）

问题：OPTIMIZE TABLE的作用？ 问题：顺序的主键什么时候会造成更坏的效果？

5.3.6. 覆盖索引

原理：MySQL可以使用索引直接获取列的数据，不需要再读取数据行。如果索引的叶子节点已经包含要查询的数据，那么就是覆盖索引。

InnoDB的二级索引在叶子节点中保存了行的主键值，所以如果二级节点能够覆盖查询，则可以避免对主键索引的二次查询。

覆盖索引必须要存储索引列的值，MySQL只能使用B-Tree索引做覆盖索引。（哈希索引、空间索引和全文索引都不支持存储索引列的值）。 不同存储引擎实现覆盖索引的方式不同，而且不是所有引擎搜支持覆盖索引。

通过EXPLAIN测试，其中Extra表明使用了覆盖索引。

5.3.6.0.1. type列和Extra列的区别：type列表示查询访问数据的方式（join type），Extra列表示是否使用覆盖索引。

```
mysql> EXPLAIN SELECT store_id, film_id FROM sakila.inventory\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: inventory
   partitions: NULL
         type: index
possible_keys: NULL
          key: idx_store_id_film_id
      key_len: 3
          ref: NULL
         rows: 4581
     filtered: 100.00
        Extra: Using index
1 row in set, 1 warning (0.00 sec)

ERROR:
No query specified
```

MySQL只能在索引中执行最左前缀匹配的LIKE比较查询，不能执行通配符开头的LIKE查询。

5.3.6.0.2. 延迟关联（deferred join）

原理：在查询的第一阶段（子查询）覆盖索引。 索引：products(artist,title,prod_id)

```
SELECT *
FROM products
  JOIN (
    SELECT prod_id
    FROM products
    WHERE actor='SEAN CARREY' AND title LIKE '%APOLLO%'
  ) AS t1 ON (t1.prod_id=products.prod_id) ;
```

Tip：外查询可以利用其它索引。（已知prod_id）

问题： 索引条件推送（index condition pushdown）（可以非显式使用延迟关联）

5.3.7. 使用索引扫描来做排序

背景：MySQL有两种方式可以生成有序的结果：通过排序操作；按索引顺序扫描。

1、如果索引不能覆盖所查找的全部的列，那么每扫描到一条索引记录都会回表查询一次对应的行（基本上是随机I/O）。 2、只有当索引的列顺序和ORDER BY子句的顺序完全一致，并且所有列的排序方向（倒序或正序）都一样时，MySQL才能够使用索引来对结果做排序。 3、如果查询关联多张表，那么只有当ORDER BY子句引用的字段全部为第一张表时，才能用索引做排序。 4、索引中的列要满足最左前缀要求。（例外：前导列为常量）

5.3.8. 压缩（前缀压缩）索引（MyISAM）

5.3.8.1. 背景

MyISAM使用前缀压缩来减少索引的大小，从而让更多的索引放入内存中。某些情况下可以极大地提高性能。默认只压缩字符串，但是通过参数设置也可以对整数做压缩。

5.3.8.2. 方法

保存索引块中的第一个值，然后将其他值和第一个值进行比较得到相同前缀的字节数和剩余的不同后缀部分，保存这些数据。 example： perform（perform） performance（7,ance） MyISAM对行指针也采用类似的前缀压缩方式。

5.3.8.3. 缺点

无法进行二分查找，只能从头开始扫描。 倒序扫描速度慢，在块中查找某一行平均要扫描半个块。

5.3.8.4. 结论

适用于I/O密集型应用，不适用于CPU密集型应用。

5.3.9. 冗余和重复索引

背景：MySQl允许在相同列上创建多个索引。

重复索引是在相同列上按照相同的顺序创建的相同类型的索引。（发现后应立即移除）

5.3.10. 未使用的索引

1. 永远不用的索引考虑删除。
2. 在Percona Server或者MariaDB中打开userstates服务器变量，再通过查询INFORMATION_SCHEMA.INDEX_STATISTICS查询每个索引的使用频率。

5.3.11. 索引和锁

5.4. 索引案例学习

5.4.1. 支持多种过滤条件

5.4.2. 避免多个范围条件

5.4.3. 优化排序

5.5. 维护索引和表

5.5.1. 找到并修复损坏的表

5.5.2. 更新索引统计信息

5.5.3. 减少索引和数据的碎片

5.6. 总结

6. 查询性能优化

优化的方面：查询优化、索引优化、库表结构优化

6.1. 为什么查询速度会慢

快速的查询真正重要的是**响应时间**。

查询的生命周期

1. 客户端
2. 服务器
3. 在服务器上解析
4. 生成执行计划
5. 执行（最重要的阶段）
6. 返回结果给客户端

6.2. 慢查询基础：优化数据访问

查询性能低下最基本的原因是访问的数据太多 优化方式：减少访问的数据量

分析步骤：

1. 确认应用是否在检索大量超过需要的数据。
2. 确认MySQl服务器层是否在分析大量超过需要的数据行。

6.2.1. 是否向数据库请求了不需要的列

有些查询会请求超过实际需要的数据，然后多余的数据会被应用程序丢弃。

1. 查询不需要的记录。（在这样的查询后面加上LIMIT）
2. 多表关联时返回全部列。
3. 总是取出全部列。
4. 重复查询相同的数据。（应使用缓存技术）

6.2.2. MySQL是否在扫描额外的记录

衡量开销的指标

- 响应时间
- 扫描的行数
- 返回的行数

这三个指标全部记录在慢查询日志中。 检查慢查询日志是找出扫描行数过多的查询的好办法。

6.2.2.1. 响应时间

响应时间分为：服务时间和排队时间 影响响应时间的因素：存储引擎的锁（表锁、行锁）、高并发资源竞争、硬件响应等。

判断响应时间是否合理？ 了解这个查询需要哪些索引以及执行计划，计算大概需要多少个顺序和随机I/O，乘以在具体硬件下I/O的消耗时间，最后加起来，得到参考值。

6.2.2.2. 扫描的行数和返回的行数

扫描的行数

1. 分析查询时，查询扫描的行数可以说明该查询找到需要的数据的效率高不高。
2. 但是不是所有的行访问的代价都是相同的。

返回的行数

1. 扫描的行数对返回的行数的比例一般在1:1和10:1之间。（MySQL不会显示生成结果实际上需要扫描多少行数据，只会显示生成结果时一共扫描了多少行数据）

6.2.2.3. 扫描的行数和访问类型

1. 在评估查询开销时，需要考虑从表中找到某一行的成本。
2. EXPLAIN语句中的type列反映了访问类型？ 全表扫描、索引扫描、范围扫描、唯一索引查询、常数引用等（速度从快到慢）。
3. 如果查询不能找到合适的访问类型，通常应增加一个合适的索引。

6.2.2.4. 应用where条件的方式

1. 在索引中使用WHERE条件来过滤不匹配的记录。（在存储引擎层完成）
2. 使用索引覆盖扫描（在EXTRA列中出现了Using index）来返回记录，直接从索引中过滤不需要的记录并返回命中的结果。（这是在MySQl服务器中完成的，但无须再回表查询）
3. 从数据表中返回数据，然后过滤不满足的条件（在Extra列中出现Using where）。（在MySQl服务器中完成，需要先从数据表中读出记录然后过滤）

并不是说增加索引就能让扫描的行数等于返回的行数

6.2.2.5. 扫描大量数据只返回少数的行的解决方法

1. 使用覆盖索引扫描，把所有需要的列都放到索引中，这样存储引擎无须回表获取对应行就可以返回结果。
2. 改变库表结构。（例如使用单独的汇总表）
3. 重写这个复杂的查询。

6.3. 重构查询的方式

6.3.1. 一个复杂的查询还是多个简单的查询

是否需要将一个复杂的查询分成多个简单的查询。 原因：MySQl从设计上让链接和断开都很轻量级，在返回一个小的查询结果方面很高效。

如果一个查询能够胜任时还写成多个独立查询是不明智的。

6.3.2. 切分查询

1. 原理：“分而治之”，将大查询切分成小查询。
2. 例子：定期清除大量数据时，如果用一个大的语句一次性完成，则可能需要一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但是重要的查询。
3. 如果是事务型引擎，很多时候小事务能够更高效。

6.3.3. 分解关联查询

很多高性能的应用都会对关联查询进行分解，可以对每一个表进行一次单表查询，然后将结果在应用程序中进行关联。

分解关联查询的优势？

1. 让缓存的效率更高。
2. 将查询分解后，执行单个查询可以减少锁的竞争。
3. 在应用层做关联，可以更容易对数据库进行拆分，更容易做到高性能和可扩展。
4. 查询效率本身也会提升。（取出数据后再使用IN()操作比关联查询中的随机查询效率更高）
5. 减少冗余记录的查询。
6. 相当于在应用中实现了哈希关联，而不是用MySQL的嵌套循环关联。

关联查询的使用场景？

1. 当应用能够方便地缓存单个查询的结果的时候。
2. 当可以将数据拆分到不同的MySQL服务器上的时候。
3. 当能够使用IN()的方式代替关联查询的时候。
4. 当查询中使用同一个数据表的时候。

6.4. 执行查询的基础

6.4.1. MySQl客户端/服务器通信协议

MySQL客户端和服务器之间的通信协议是“半双工”的。（不能进行流量控制）

接受全部结果并缓存通常可以减少服务器压力，让查询能够早点结束、早点释放相应的资源。

6.4.1.1. 查询状态

```
SHOW FULL PROCESSLIST ;
```

1. Sleep
2. Query
3. Locked
4. Analyzing and statistics
5. Copy to tmp table [on disk]
6. Sorting result
7. Sending data

6.4.2. 查询缓存

在解析查询语句之前，如果查询缓存是打开的，优先查找缓存（通过大小写敏感的哈希查找实现），如果命中，检查用户权限（通过查询缓存中当前查询需要访问的表信息查看）

6.4.3. 查询优化处理

将SQL转换成一个执行计划。 过程：解析SQL、预处理、优化SQL执行计划。 能够优化的类型？

1. 重新定义关联表顺序。
2. 将外连接转化为内连接。
3. 使用等价变化规则。
4. 优化COUNT()、MIN()、MAX()
5. 预估并转化为常数表达式。
6. 覆盖索引扫描。
7. 子查询优化。
8. 提前终止查询。
9. 等值传播。
10. 列表IN()的比较。

6.4.4. 查询执行引擎

6.4.5. 返回结果给客户端

6.5. MySQL查询优化器的局限性

6.5.1. 关联子查询

6.5.2. UNION的限制

6.5.3. 索引合并优化

6.5.4. 等值传递

1. 分区表底层由多个物理子表组成。
2. 实现分区的代码是对一组底层表的句柄对象(Handler Object)的封装。
3. 索引是按照分区的字表定义的，**没有全局索引**。
4. 目的：将数据按照一个较粗的粒度分在不同的表中。

1. 试图本身是一个虚拟表，不存放任何数据。
2. 不能对试图创建触发器。
3. 实现方法：合并算法和临时表算法。

7.2.1. 可更新视图

7.2.2. 视图对性能的影响

7.2.3. 视图的限制

1. MySQL不支持物化视图，也不支持在视图中创建索引。
2. MySQL不会保存视图定义的原始语句。

7.3. 外键约束

1. InnoDB是MySQL中唯一支持外键的内置存储引擎。
2. MySQL强制外键使用索引。
3. 修改数据要进行额外的约束性检查。
4. 导致额外的锁等待。

7.4. 在MySQL内部存储代码

存储过程和存储函数统称为“存储程序”。

7.4.1. 存储过程和函数

7.4.2. 触发器

1. 使用触发器实现强制限制。
2. 对于每个表的每一个事件，最多定义一个触发器。
3. 只支持基于行的触发。

7.4.3. 事件

指定MySQL某个时候执行一段SQL代码，或者每隔一个时间间隔执行一段SQL代码。

7.4.4. 在存储过程中保留注释

使用版本相关的注释（只有版本号大于某个值才执行的代码）

7.5. 游标

1. MySQL在服务器端只提供只读的、单向的游标，而且只在存储过程或者更底层的客户端API中使用。
2. 需要使用limit来限制返回集。

7.6. 绑定变量

```
INSERT INTO tb1(col1, col2, col3) VALUES (?, ?, ?);
```

7.6.1. 绑定变量的优化

7.6.2. SQL接口的绑定变量

7.6.3. 绑定变量的限制

7.7. 用户自定义函数

可以使用支持C语言调用约定的任何编程语言来实现。

7.8. 插件

1. 存储过程插件
2. 后台查件
3. INFORMATION_SCHEMA插件
4. 全文解析插件
5. 审计插件
6. 认证插件

7.9. 字符集和校对

7.9.1. MySQL如何使用字符集

7.9.2. 选择字符集和校对规则

7.9.3. 字符集和校对规则如何影响查询

7.10. 全文索引

1. 目的：基于相似度的查询。
2. 没有索引也可以工作，有索引效率会更高。
3.

7.11. 分布式（XA）事务

7.12. 查询缓存

7.13. 总结

8. 优化服务器设置

8.1. MySQL配置的工作原理

1. MySQL从命令行参数和配置文件获得配置信息。
2. 在类UNIX系统中，配置文件的位置一般在/etc/my.cnf或者/etc/mysql/my.cnf。

8.2. 什么不该做

1. 没有一个适合所有场景的“最佳配置文件”。

8.3. 创建MySQL配置文件

8.3.1. 检查MySQL服务器状态变量

1. 使用 `mysqladmin extended-status -ri60` 每隔60s查看状态变量的增量变化。

8.4. 配置内存使用

1. 确认可以使用的内存上限。
2. 确认每个连接MySQL需要使用多少内存，例如排序缓冲和临时表。
3. 确认操作系统需要多少内存才够用。包括同一台机器上其他程序使用的内存，如定时任务。
4. 把省下的内存全部给MySQL的缓存，例如InnoDB的缓冲池，这样做很有意义。

8.4.1. MySQL可以使用多少内存