# 架构

## SQLiScanner：又一款基于SQLMAP和Charles的被动SQL 注入漏洞扫描工具 - 推酷

http://www.tuicool.com/articles/MZvA3m3

时间 2016-09-18 18:21:25  [FreeBuf](http://www.tuicool.com/sites/MNnABb" \t "_blank)

原文  [http://www.freebuf.com/sectool/114635.html](http://www.freebuf.com/sectool/114635.html?utm_source=tuicool&utm_medium=referral)

主题 [sqlmap](http://www.tuicool.com/topics/11250076)[SQL](http://www.tuicool.com/topics/11030067)

**项目地址：**[***SQLiScanner***](https://github.com/0xbug/SQLiScanner)

简介

叕一款基于SQLMAP和Charles的被动SQL 注入漏洞扫描工具

从内部安全平台 分离出来的一个模块, 支持 **Har** 文件的扫描(搭配 Charles 使用: Tools=>Auto Save)

特性

* 邮箱通知
* 任务统计
* sqlmap 复现命令生成

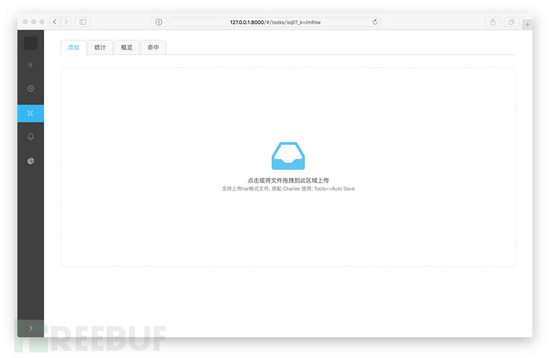
依赖

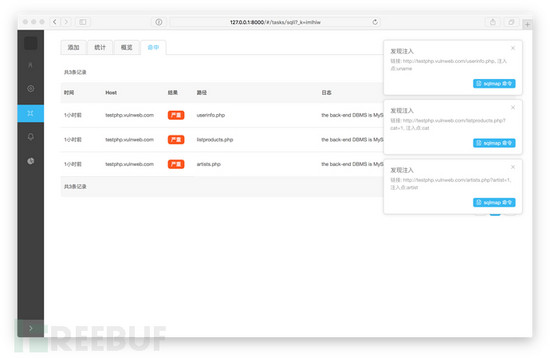
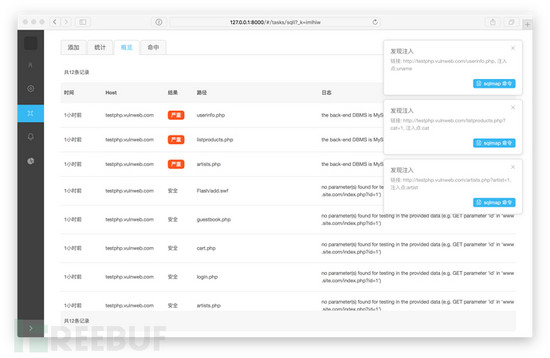
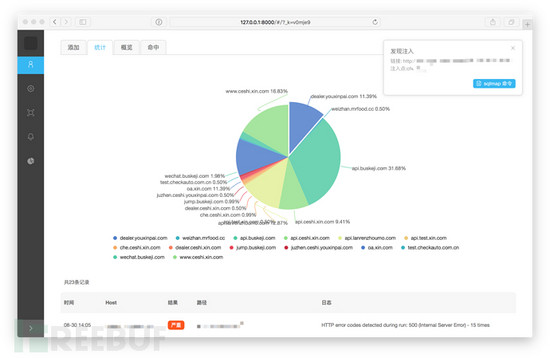
* Python 3.x
* Django 1.9
* PostgreSQL
* Celery
* sqlmap
* redis

支持平台

* Linux
* osx

截图





安装

克隆项目到本地

git **clone** https:*//github.com/0xbug/SQLiScanner.git --depth 1*

配置 sqlmap:

git clone https://github.com/sqlmapproject/sqlmap.git --depth 1

SQLiScanner 支持 Python version 3.x on Linux and osx.

安装依赖

cd SQLiScanner/

virtualenv --python=/usr/local/bin/python3.5 venv

source venv/bin/activate

pip install -r requirements.txt

创建数据库（需要配置数据库）

python manage.py makemigrations scanner

python manage.py migrate

创建 superuser

python manage.py createsuperuser

设置

数据库设置

SQLiScanner/settings.py:85

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': '',

'USER': '',

'PASSWORD': '',

'HOST': '127.0.0.1',

'PORT': '5432',

}

}

邮件通知配置

SQLiScanner/settings.py:152

*# Email*

EMAIL\_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'

EMAIL\_USE\_TLS = **False**

EMAIL\_HOST = ''

EMAIL\_PORT = 25

EMAIL\_HOST\_USER = ''

EMAIL\_HOST\_PASSWORD = ''

DEFAULT\_FROM\_EMAIL = ''

scanner/tasks.py:13

**class SqlScanTask(object):**

**def** **\_\_init\_\_**(self, sqli\_obj):

self.api\_url = "http://127.0.0.1:8775"

self.mail\_from = ""

self.mail\_to = [""]

运行

redis-server

python sqlmapapi.py -s -p 8775

python manage.py celery worker *--loglevel=info*

python manage.py runserver

**\*本文投稿作者：Blur，转载须注明来自FreeBuf.COM**

## 分库分表的几种常见形式以及可能遇到的难题 - 推酷

http://www.tuicool.com/articles/maAZNjj

时间 2016-09-19 09:22:37  [代码说](http://www.tuicool.com/sites/fE3iQv" \t "_blank)

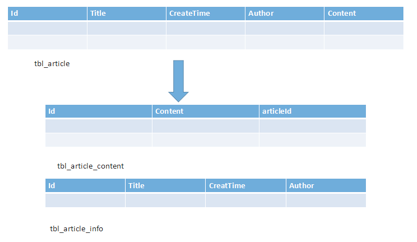
原文  [http://h2ex.com/1539](http://h2ex.com/1539?utm_source=tuicool&utm_medium=referral)

主题 [数据库](http://www.tuicool.com/topics/11000064)

在谈论数据库架构和数据库优化的时候，我们经常会听到“分库分表”、“分片”、“Sharding”…这样的关键词。让人感到高兴的是，这些朋友所服务的公司业务量正在（或者即将面临）高速增长，技术方面也面临着一些挑战。让人感到担忧的是，他们系统真的就需要“分库分表”了吗？“分库分表”有那么容易实践吗？为此，笔者整理了分库分表中可能遇到的一些问题，并结合以往经验介绍了对应的解决思路和建议。

垂直分表

垂直分表在日常开发和设计中比较常见，通俗的说法叫做“大表拆小表”，拆分是基于关系型数据库中的“列”（字段）进行的。通常情况，某个表中的字段比较多，可以新建立一张“扩展表”，将不经常使用或者长度较大的字段拆分出去放到“扩展表”中，如下图所示：



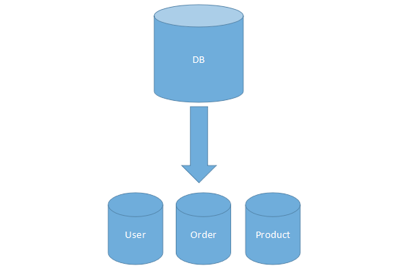
小结

在字段很多的情况下，拆分开确实更便于开发和维护（笔者曾见过某个遗留系统中，一个大表中包含100多列的）。某种意义上也能避免“跨页”的问题（MySQL、MSSQL底层都是通过“数据页”来存储的，“跨页”问题可能会造成额外的性能开销，这里不展开，感兴趣的朋友可以自行查阅相关资料进行研究）。

拆分字段的操作建议在数据库设计阶段就做好。如果是在发展过程中拆分，则需要改写以前的查询语句，会额外带来一定的成本和风险，建议谨慎。

垂直分库

垂直分库在“微服务”盛行的今天已经非常普及了。基本的思路就是按照业务模块来划分出不同的数据库，而不是像早期一样将所有的数据表都放到同一个数据库中。如下图：



小结

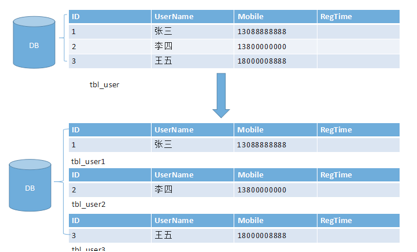
系统层面的“服务化”拆分操作，能够解决业务系统层面的耦合和性能瓶颈，有利于系统的扩展维护。而数据库层面的拆分，道理也是相通的。与服务的“治理”和“降级”机制类似，我们也能对不同业务类型的数据进行“分级”管理、维护、监控、扩展等。

众所周知，数据库往往最容易成为应用系统的瓶颈，而数据库本身属于“有状态”的，相对于Web和应用服务器来讲，是比较难实现“横向扩展”的。数据库的连接资源比较宝贵且单机处理能力也有限，在高并发场景下，垂直分库一定程度上能够突破IO、连接数及单机硬件资源的瓶颈，是大型分布式系统中优化数据库架构的重要手段。

然后，很多人并没有从根本上搞清楚为什么要拆分，也没有掌握拆分的原则和技巧，只是一味的模仿大厂的做法。导致拆分后遇到很多问题（例如：跨库join，分布式事务等）。

水平分表

水平分表也称为横向分表，比较容易理解，就是将表中不同的数据行按照一定规律分布到不同的数据库表中（这些表保存在同一个数据库中），这样来降低单表数据量，优化查询性能。最常见的方式就是通过主键或者时间等字段进行Hash和取模后拆分。如下图所示：

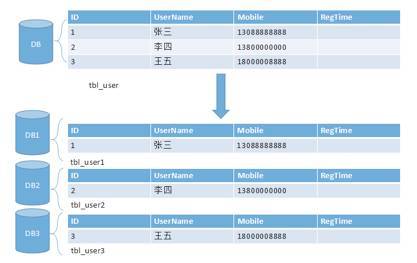


小结

水平分表，能够降低单表的数据量，一定程度上可以缓解查询性能瓶颈。但本质上这些表还保存在同一个库中，所以库级别还是会有IO瓶颈。所以，一般不建议采用这种做法。

水平分库分表

水平分库分表与上面讲到的水平分表的思想相同，唯一不同的就是将这些拆分出来的表保存在不同的数据中。这也是很多大型互联网公司所选择的做法。如下图：



某种意义上来讲，有些系统中使用的“冷热数据分离”（将一些使用较少的历史数据迁移到其他的数据库中。而在业务功能上，通常默认只提供热点数据的查询），也是类似的实践。在高并发和海量数据的场景下，分库分表能够有效缓解单机和单库的性能瓶颈和压力，突破IO、连接数、硬件资源的瓶颈。当然，投入的硬件成本也会更高。同时，这也会带来一些复杂的技术问题和挑战（例如：跨分片的复杂查询，跨分片事务等）

分库分表的难点

垂直分库带来的问题和解决思路：

跨库join的问题

在拆分之前，系统中很多列表和详情页所需的数据是可以通过sql join来完成的。而拆分后，数据库可能是分布式在不同实例和不同的主机上，join将变得非常麻烦。而且基于架构规范，性能，安全性等方面考虑，一般是禁止跨库join的。那该怎么办呢？首先要考虑下垂直分库的设计问题，如果可以调整，那就优先调整。如果无法调整的情况，下面笔者将结合以往的实际经验，总结几种常见的解决思路，并分析其适用场景。

跨库Join的几种解决思路

全局表

所谓全局表，就是有可能系统中所有模块都可能会依赖到的一些表。比较类似我们理解的“数据字典”。为了避免跨库join查询，我们可以将这类表在其他每个数据库中均保存一份。同时，这类数据通常也很少发生修改（甚至几乎不会），所以也不用太担心“一致性”问题。

字段冗余

这是一种典型的反范式设计，在互联网行业中比较常见，通常是为了性能来避免join查询。

举个电商业务中很简单的场景：

“订单表”中保存“卖家Id”的同时，将卖家的“Name”字段也冗余，这样查询订单详情的时候就不需要再去查询“卖家用户表”。

字段冗余能带来便利，是一种“空间换时间”的体现。但其适用场景也比较有限，比较适合依赖字段较少的情况。最复杂的还是数据一致性问题，这点很难保证，可以借助数据库中的触发器或者在业务代码层面去保证。当然，也需要结合实际业务场景来看一致性的要求。就像上面例子，如果卖家修改了Name之后，是否需要在订单信息中同步更新呢？

数据同步

定时A库中的tab\_a表和B库中tbl\_b有关联，可以定时将指定的表做同步。当然，同步本来会对数据库带来一定的影响，需要性能影响和数据时效性中取得一个平衡。这样来避免复杂的跨库查询。笔者曾经在项目中是通过ETL工具来实施的。

系统层组装

在系统层面，通过调用不同模块的组件或者服务，获取到数据并进行字段拼装。说起来很容易，但实践起来可真没有这么简单，尤其是数据库设计上存在问题但又无法轻易调整的时候。

具体情况通常会比较复杂。下面笔者结合以往实际经验，并通过伪代码方式来描述。

简单的列表查询的情况



伪代码很容易理解，先获取“我的提问列表”数据，然后再根据列表中的UserId去循环调用依赖的用户服务获取到用户的RealName，拼装结果并返回。

有经验的读者一眼就能看出上诉伪代码存在效率问题。循环调用服务，可能会有循环RPC，循环查询数据库…不推荐使用。再看看改进后的：



这种实现方式，看起来要优雅一点，其实就是把循环调用改成一次调用。当然，用户服务的数据库查询中很可能是In查询，效率方面比上一种方式更高。（坊间流传In查询会全表扫描，存在性能问题，传闻不可全信。其实查询优化器都是基本成本估算的，经过测试，在In语句中条件字段有索引的时候，条件较少的情况是会走索引的。这里不细展开说明，感兴趣的朋友请自行测试）。

小结

简单字段组装的情况下，我们只需要先获取“主表”数据，然后再根据关联关系，调用其他模块的组件或服务来获取依赖的其他字段（如例中依赖的用户信息），最后将数据进行组装。

通常，我们都会通过缓存来避免频繁RPC通信和数据库查询的开销。

列表查询带条件过滤的情况

在上述例子中，都是简单的字段组装，而不存在条件过滤。看拆分前的SQL：



这种连接查询并且还带条件过滤的情况，想在代码层面组装数据其实是非常复杂的（尤其是左表和右表都带条件过滤的情况会更复杂），不能像之前例子中那样简单的进行组装了。试想一下，如果像上面那样简单的进行组装，造成的结果就是返回的数据不完整，不准确。

有如下几种解决思路：

1. 查出所有的问答数据，然后调用用户服务进行拼装数据，再根据过滤字段state字段进行过滤，最后进行排序和分页并返回。

这种方式能够保证数据的准确性和完整性，但是性能影响非常大，不建议使用。

1. 查询出state字段符合/不符合的UserId，在查询问答数据的时候使用in/not in进行过滤，排序，分页等。过滤出有效的问答数据后，再调用用户服务获取数据进行组装。

这种方式明显更优雅点。笔者之前在某个项目的特殊场景中就是采用过这种方式实现。

跨库事务（分布式事务）的问题

按业务拆分数据库之后，不可避免的就是“分布式事务”的问题。以往在代码中通过spring注解简单配置就能实现事务的，现在则需要花很大的成本去保证一致性。这里不展开介绍，

感兴趣的读者可以自行参考《分布式事务一致性解决方案》，链接地址：

http://www.infoq.com/cn/articles/solution-of-distributed-system-transaction-consistency

垂直分库总结和实践建议

本篇中主要描述了几种常见的拆分方式，并着重介绍了垂直分库带来的一些问题和解决思路。读者朋友可能还有些问题和疑惑。

1. 我们目前的数据库是否需要进行垂直分库？

根据系统架构和公司实际情况来，如果你们的系统还是个简单的单体应用，并且没有什么访问量和数据量，那就别着急折腾“垂直分库”了，否则没有任何收益，也很难有好结果。

切记，“过度设计”和“过早优化”是很多架构师和技术人员常犯的毛病。

2. 垂直拆分有没有原则或者技巧？

没有什么黄金法则和标准答案。一般是参考系统的业务模块拆分来进行数据库的拆分。比如“用户服务”，对应的可能就是“用户数据库”。但是也不一定严格一一对应。有些情况下，数据库拆分的粒度可能会比系统拆分的粒度更粗。笔者也确实见过有些系统中的某些表原本应该放A库中的，却放在了B库中。有些库和表原本是可以合并的，却单独保存着。还有些表，看起来放在A库中也OK，放在B库中也合理。

如何设计和权衡，这个就看实际情况和架构师/开发人员的水平了。

3. 上面举例的都太简单了，我们的后台报表系统中join的表都有n个了，

**分库后该怎么查？**

有很多朋友跟我提过类似的问题。其实互联网的业务系统中，本来就应该尽量避免join的，如果有多个join的，要么是设计不合理，要么是技术选型有误。请自行科普下OLAP和OLTP，报表类的系统在传统BI时代都是通过OLAP数据仓库去实现的（现在则更多是借助离线分析、流式计算等手段实现），而不该向上面描述的那样直接在业务库中执行大量join和统计。

由于篇幅关系，下篇中我们再继续细聊“水平分库分表”相关的话题。

作者介绍

丁浪，技术架构师。关注高并发、高可用的架构设计，对系统服务化、分库分表、性能调优等方面有深入研究和丰富实践经验。热衷于技术研究和分享。

## 深入理解Raft及在NewSQL中的使用 - 推酷

http://www.tuicool.com/articles/IVfmauU

时间 2016-09-19 10:47:51  [听云博客](http://www.tuicool.com/sites/NZJNJvv" \t "_blank)

原文  [https://blog.tingyun.com/web/article/detail/1127](https://blog.tingyun.com/web/article/detail/1127?utm_source=tuicool&utm_medium=referral)

主题 [Raft](http://www.tuicool.com/topics/11090197)[NewSQL](http://www.tuicool.com/topics/11030112)

如需转载请联系听云College团队成员小尹 邮箱：yinhy#tingyun.com

本文整理自APMCon 2016中国应用性能管理大会 数据库性能优化专场 京东资深架构师 张成远 题为《深入理解Raft及在NewSQL中的使用》的演讲，现场解读Raft核心内容，剖析etcd Raft库的设计实现，针对此类复杂分布式系统算法的测试方法，并介绍业界主流NewSQL的详细设计，Raft在其中发挥的重要作用，以及如何使用等。

张成远：我是京东资深架构师张成远，目前主要是做分布式数据库相关的一些架构研发等工作。今天主要讲一下Raft在NewSQL的使用，因为内容涉及到Raft和NewSQL，我会先讲一下什么是NewSQL，什么是事务以及一致性等，还有NewSQL大概是怎么做的，然后再讲一下Raft是什么以及Raft是怎么样工作的。

一、What is NewSQL



从SQL到NoSQL再到NewSQL，一路走来大概经历过这样一个发展过程，一开始最早的时候是SQL类产品，后来慢慢有了NoSQL的东西，最近这一两年开始有NewSQL这个概念。SQL理论，也就是关系型数据库理论大概是上世纪70年代左右就是已经提出来了，这个理论下产生了很多的产品，像MySQL、Oracle、PostgreSQL、SQL Server等很多很多。

后来随着分布式系统的发展衍生了NoSQL这个概念，出现了很多的分布式存储系统像Redis、mongoDB、cassandra等，这些NoSQL类的产品特点是不支持SQL，主要以KV形式存储，在扩容方面非常容易，可以较为轻松的进行一些集群的伸缩等操作。NoSQL出来以后一直喊着要把SQL颠覆，后来在实际生产环境中我们发现这两者是一种长期并存的趋势，每类产品都有一定的适用场景。

等到后来，大家可能希望出现一种东西可以结合SQL跟NoSQL的优点，既支持SQL支持事务又可以比较容易的进行集群的伸缩扩容等操作，让海量数据方面的处理可以更加的容易，所以就出现了NewSQL的概念。NewSQL这个概念之下，国内外都有一些优秀的开源的项目在做像TiDB、CockroachDB等，但目前开源领域可能还没有在生产环境当中真正大规模使用的。如果要说真正落地使用的话可能当属google的F1/spanner了，但他们家没有开源只有论文流出。

二、What is transaction



然后，接下来讲一下NewSQL最大的特点，就是支持SQL同时支持事务，支持SQL比较好理解，相当于可以用SQL的方式来增删改查这个系统，至于支持事务那我们就要回到非常老的一个问题，到底什么是事务？支持事务，那事务到底是什么定义？事务有ACID属性，也就是 **原子性** 、 **一致性**、 **隔离性**还有 **持久性**。

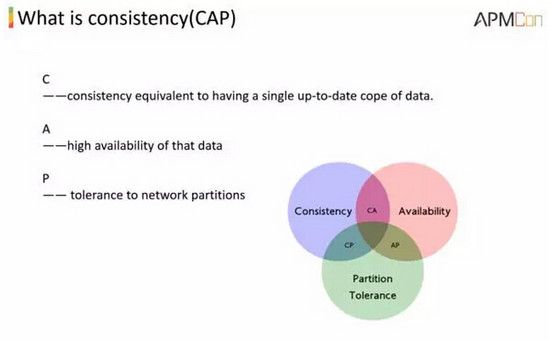
**原子性** 是指整个事务要么成功，要么失败(跟没有发生过一样)，不要有任何的中间状态。

**一致性** 其实在一定程度上来说主要是数据库层面的完整性，打个比方A的帐户里面100块钱，B的帐户里面50块钱，然后，A给B 25块钱，首先是从A上减去25块钱，然后将这25块钱加到B上，最终都是75块钱。不能出现A减25块钱然后B的25块钱没有加上去，这样全局数据库完整性就不一致了，这个属性需要应用程序自己做更多的考虑，另外数据库层面保证了原子性、隔离性以及持久性的时候，外加应用程序自身的逻辑是正常的，那这个一致性就是可以保证的。

什么是 **隔离性** 呢，一般是指并发事务操作彼此之间不可见，简单的说就是A做操作的时候不要让B事务看到。

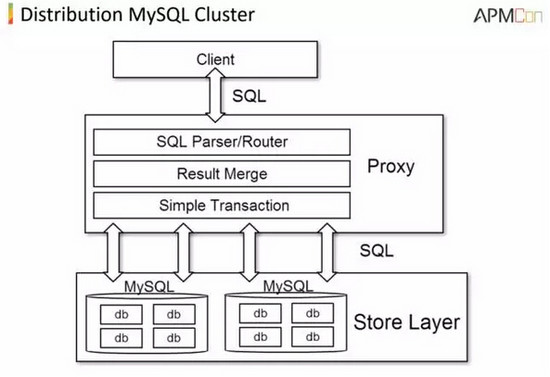
**持久性** ，简单的理解就是事务commit了以后要保证这个事务是一直不会丢失的。比如你把A减了100块钱或者A加了100块钱，这个事情如果告诉用户已经成功了就是成功了，不能出现因为机器挂掉之类导致这种修改丢失。

还有一个概念也要提一下，这个也是一个比较老的概念，就是CAP。



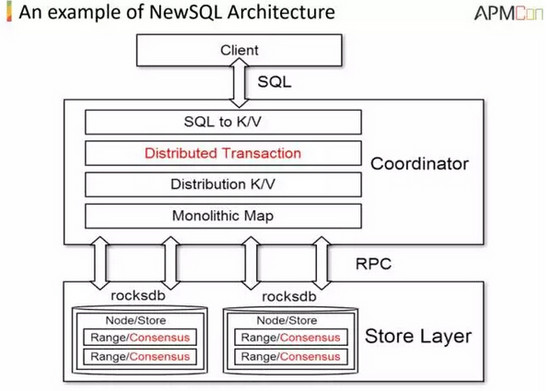
很多时候会把CAP一致性和ACID的一致性混在一起，CAP一致性跟ACID的一致性其实是两回事情。CAP的一致性是针对很多个节点，要求各个节点之间的数据是完全一致的，也就是要求实际存储的数据是完全一致的一致性，也就是纯粹的多副本方式。

CAP里的A其实就是高可用，简单的理解就是随时去访问都可以访问到，CAP里的P是说在发生一些网络异常状况的时候是否可以容忍。在绝大多数的系统当中，理论上来说CAP是不能同时满足的，但在实际生产环境中往往选择弱化的P，也就是当发生网络异常时直接将一部分异常节点舍弃保证剩余节点的服务都是正常的从而保证了整体的服务，所以从这个角度上来说CAP是可以全部满足的，或者说满足生产需求已经足够。



上图是我们在做分布式数据库的时候一般的一些做法，很多的时候直接基于一些现有的SQL类产品，比如MySQL，外加一层数据库中间件的方式比如引入Proxy的方式来解决这个分布式数据库问题，SQL发过来以后由Proxy接收然后解析拆解，再把相应的SQL发到相应的SQL产品上，支持简单的事务。为什么叫做简单的事务呢，因为如果只涉及到一个分片是可以的，但是多个就是支持不了的，确切的说是因为不能保证严格的分布式事务的语义，后面会详细讲一下。

三、NewSQL



那么NewSQL是怎么样的？NewSQL有很多种可能的实现方式，这里列举NewSQL其中的一种架构模式，如上图，上层兼容SQL协议的一个协调者，然后通过RPC之类的通信方式和底层存储通信，存储节点可能会选择Rocksdb之类的K/V存储系统。为什么说上图这个是一个协调者，但看到的效果看起来也像一个代理像一个Proxy？他的区别在于可以做很多的事情，可以对这个事务进行一些记录，记录到日志里面，还可以提供全局的事务ID之类，保证同一个大事务里的各个节点上的子事务ID可以和全局事务ID是一样，后面会详细讲一下。

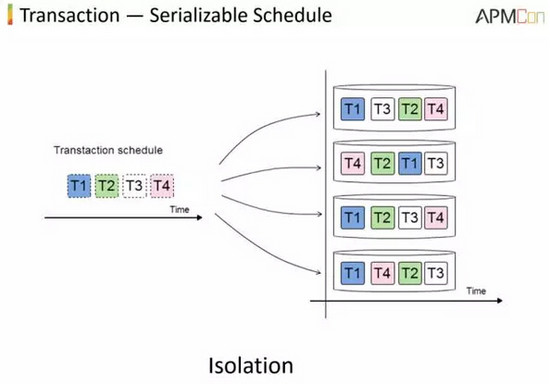
一般来说，NewSQL要支持SQL同时有NoSQL的优点，这个从实现上面来说给用户的感觉就是SQL产品，SQL过来以后协调者会把它拆成各种各样的KV操作，丢到各个节点上去，相当于丢到多个分片上，但同一个分片上有很多个节点，这些节点之间要保证数据是一致的，这样可以做到这个分片上万一有节点挂了，还有另外的节点可以对外提供服务。

1、隔离性

那怎么保证同一个分片内多个节点之间的数据是一致的呢？Raft就是用来保证各个节点是一致的。为什么上图Rocksdb这层要引入Range的概念呢？因为作为一个节点可能数据量会非常大，后期要支持扩容怎么扩？如果这个节点不拆分是没有办法扩的，所以要引入Range这个逻辑概念。需要扩容的时候就可以把Range进行拆分成多个，每一个Rocksdb上面放很多的Range，然后把这些Range迁移到其他的Rocksdb上面去，相当于通过拆分加迁移方式同时达到扩容整个集群容量的目的，每一个Range相当于一个分片里的节点，多个Range通过Raft构成一个分片，同一个分片里的Range的数据是一样的。

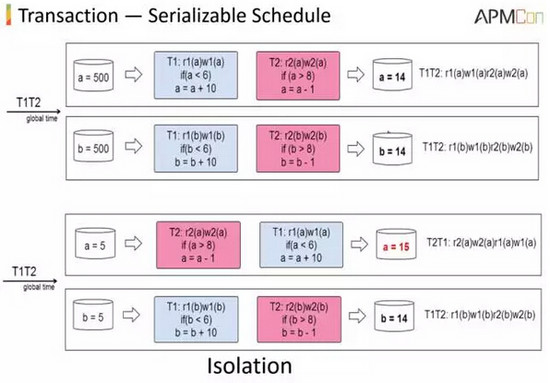
现在我们再回过头去讲一下，为什么基于现有的SQL类产品开发一个Proxy的方式其实是没有办法保证严格的分布式事务的，有一种场景很容易理解：假设以MySQL为例，如果一个事务里有多条SQL，这些SQL如果涉及到不同的MySQL实例，可能会出现一个分片的MySQL上提交成功了，另一个分片的MySQL挂了，那就会出现事务只提交一半的情况，这种情况就保证不了原子性。

如果有一个分片的MySQL挂了，事务的原子性无法保证这种场景是比较好理解的，但是其实就算所有的分片上的MySQL都是正常的也是无法严格保证事务的，为什么呢？因为前面讲过ACID里有一个隔离性属性，并发事务之间要求是不可见的，而基于MySQL中间件的方式是保证不了隔离性的。



如上图，假设有4个事务是同时发生的，对于理解就是并发过去的到Proxy节点的时候实际上肯定是有一个顺序的，这个顺序我们假设是T1、T2、T3、T4个顺序，如果4个事务涉及到4个分片，4个事务同时落到四个片的MySQL上但实际上在每个MySQL上这四个事务的执行顺序可能是不一样的，每个MySQL上会有自己的一个执行序列，也就是这些事务在每个MySQL上会组成一个可串行化调度的序列，所谓可串行化调度就是并发的事务操作会按照一定的顺序执行，执行以后的效果等价于这些事务按照某个顺序串行执行的效果，那么这个调度序列就是可串行化调度的，而这些等价的串行顺序可能是一样的也可能是不一样的。比如这四个事务T1、T2、T3、T4在分片一上的执行顺序可能是T1、T3、T2、T4，分片二上的执行顺序却是T4、T2、T1、T3，分片三的执行顺序是T1、T2、T3、T4，而分片四上的执行顺序是T1、T4、T2、T3。

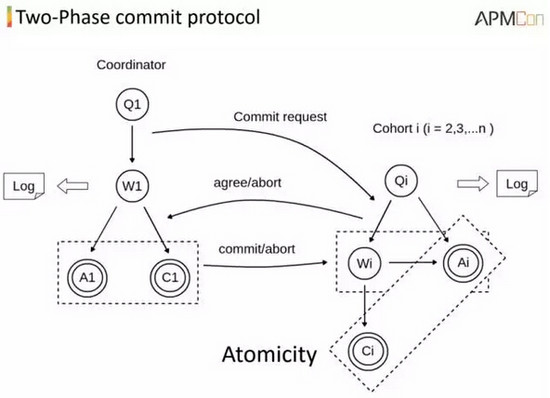
在这种情况下，分片2上就会出现事务T1会看到T4和T2的修改，而在分片1、3和4上，事务T1事务是先执行的，所以并不会看到T4和T2的修改，这就相当于在全局范围内，事务T4和T2的部分修改被T1读到了，而另一部分却没有被T1读取到，出现了T1和T2及T4之间的隔离是有问题的，所以在全局范围内是无法保证严格事务之间的隔离性的。



再举个例子如图所示，假设变量a和b都是5，此时有事务T1和T2且全局执行顺序是T1和T2，假设每个分片上都是按照T1和T2的方式执行，结果最终结果a和b都是14，此时是符合事务的可串行化调度的。

但是可能出现一种情况，在分片1上的执行顺序是T2、T1，分片2上的执行顺序是T1、T2，此时分片1上a就变成了15，而分片2上就变成了14， 这就不符合可串行化的条件了，因为从全局来说T1和T2是混在一起的，即不等价于T1T2的执行顺序也不等价于T2T1的执行顺序。怎么解决这个问题呢，这时一般会采用引入一个全局的事务ID，在实际执行的时候可能会采用时间戳的方式，具体细节这块今天先不展开讨论了。

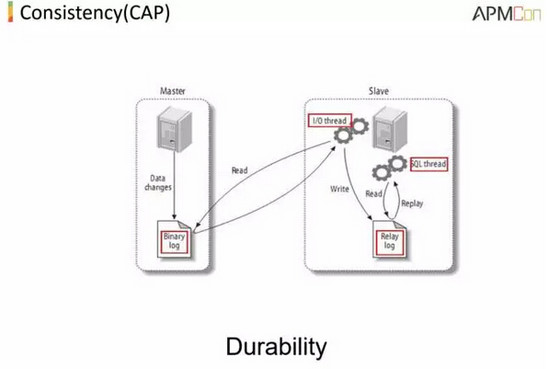
2、原子性



前面讲了隔离性问题，现在我们再讲一下原子性，在保证分布式事务的时候要支持分布式事务语义，那ACID 4个属性都要满足，前面介绍了隔离性，那原子性这个怎么解决呢？在实际实现的时候往往采用类似两阶段提交的方式，我这里画的是两阶段提交的理论上的定义，全局会有一个协调者，事务要提交的时候协调者会发送请求到各个参与节点，参与节点确认可以提交了就会回复给协调者表示通过，如果参与节点表示这个事务因为一些原因不能提交就会给协调者反馈说这个事务需要回滚，那么协调者就会根据这些反馈信息来综合决定事务是否需要提交。只有所有的参与节点都表示可以提交，协调者才会在自己先写日志提交这个事务，然后发送commit命令到所有的参与节点，如果出现任何一个参与节点表示需要回滚，那么协调者就会在先写日志表示abort这个事务，然后把abort信息发送给每个节点。图上的每一个圈都是一个系统的状态，是一个典型的状态机，其中两层圆圈表示的终止状态，整个状态机最终会到达终止状态上，状态机的东西在后面我也会再提一下。

3、持久性

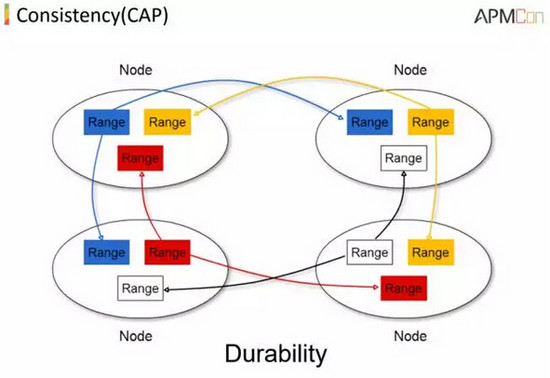
讲完NewSQL里怎么支持原子性和隔离性，我们再讲一下持久性。同时也要提一下CAP里的一致性，刚才已经说了CAP的一致性和ACID里的一致性不是一个概念，CAP里的一致性是解决多个节点副本之间数据一致性的问题，好比MySQL的主从数据完全一样，那就符合CAP里的一致性，从这个层面上来说CAP里的一致性其实对应事务里的持久性。



为什么这么说呢，大家想一个例子，还是以MySQL的主从为例来说明。主从主要可以解决高可用的问题也可以解决数据备份的问题，如果主从之间假设没有数据的复制，那么在主挂了或者出异常以后，从接替主开始对外服务相当于保证了CAP里的A，但是之前所有已经提交的事务都丢失了，因为主从之间没有进行数据复制，相当于丢失了ACID里的持久性。

假设主从之间有数据复制但是有延后，那么当发生主从切换的时候，依然会存在事务丢失的情况，因为可能会出现有几个事务在主上已经提交了，但是还没有复制到从上，此时主挂了从开始担任主对外提供服务依然保证了CAP里的A，但是依然会存在已经提交的事务丢失的情况。所以从这个角度来说CAP里的多个节点之间的一致性本质上是对应ACID属性里的持久性，如果不追求事务的持久性，那也就不用关心各个节点之间的数据是否一致了。

既然已经明确了CAP里的一致性本质上对应了持久性这个属性，那接下来的问题就是这个一致性也就是ACID里的持久性怎么做呢，对MySQL来说可能有半同步的复制方式，但是半同步在实际使用当中，如果真的出现一些问题比如网络问题或者机器性能问题等最终可能会变成异步复制，并不能保证严格的一致性。

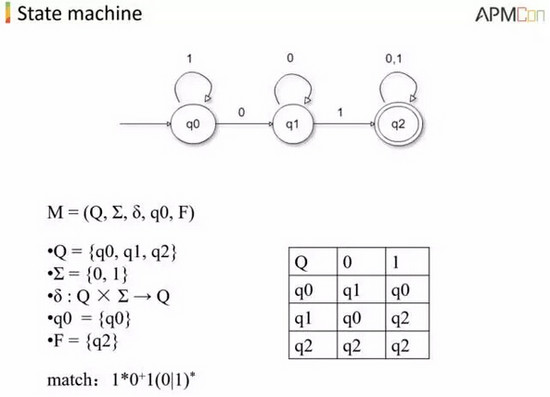


所以在NewSQL中一般会考虑采用Raft来保证一致性。如上所示各个Node里有很多的Range，各个Range之间通过Raft来保证彼此之间的一致性，多个Range组成的Raft组合相当于是一个分片，Raft保证了这些Range之间的一致性从而保证了事务在同一个节点的多副本上的持久性。

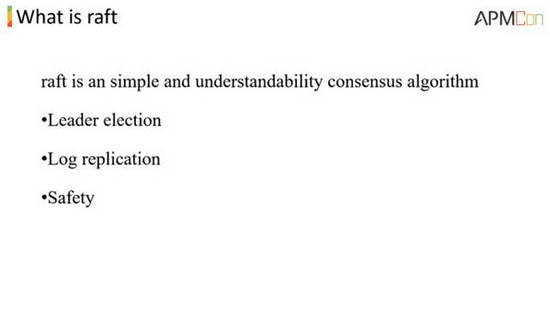
四、What is raft

在真正开始介绍Raft之前，我想先引入一个状态机的介绍，因为前面的部分以及后面的部分都会涉及到状态机，那什么是状态机呢，需要简单介绍一下。

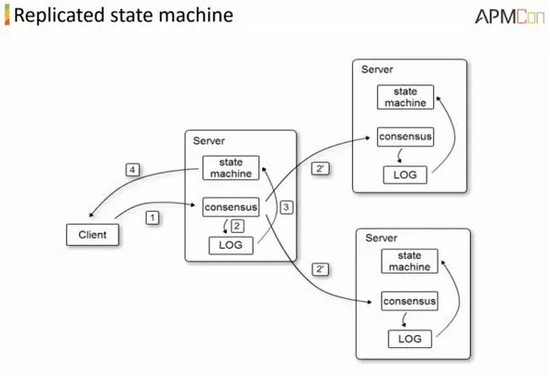
1、State machine



其实状态机可以简单的理解就是一个五元组，状态机可以记录为M={Q, Ʃ,δ, q0, F}，其中Q是状态集合；Ʃ是输入符号，在实际工程中也可以理解为具体的事件；δ是转移函数，表示在某个状态下接收了某个符号会进入到什么状态；q0是整个状态机的可能的起始状态；F是整个状态机的终止状态，在实际表示的时候一般会用上层的圆圈表示，终止状态的理解更恰当的说是指相对稳定的状态，所以并不是说到了终止状态以后整个状态机就不工作了，比如图上这个状态机的效果就是接收了1\*0+(0|1)\*的字符串。之所以在这里提一下状态机到底是一个什么东西是因为前面的两阶段提交就是一个状态机，后面的Raft也是到处都是状态机。

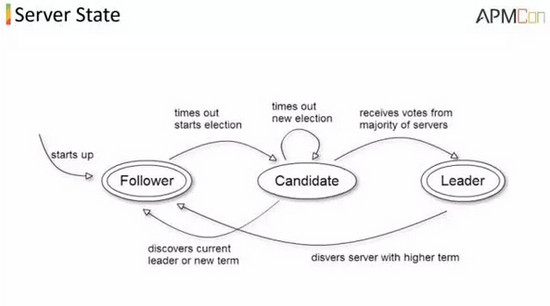


现在接下去讲一下什么是Raft，Raft简单的说就是相比传统的Paxos算法来说是一个更简单且比较容易理解，在工程上也比较容易实现的一种一致性算法，通过选举leader以及日志复制的方式完成工作，具体的实现会有一些措施来确保整个算法是可以安全工作的，比如如何保证操作复制到多数节点才生效，再比如出现异常时少数节点上有的日志在大多数节点上可能是没有的这种情况，如何通过算法将这些日志清除掉。



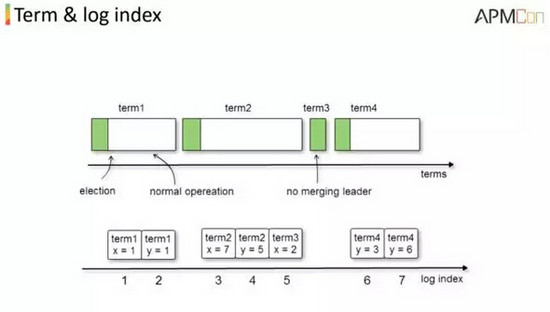
上图是Raft工作的示意图，Client发送请求到Leader上，consensus部分可以理解为Raft的主要工作部分，请求先发送到consensus模块，consensus部分会把操作先记录到本地日志中，然后同时将这个操作复制到大多数其他节点上去，等确认大多数节点已经接收到了这个操作，然后会把这个操作apply到上层的状态机，在实际实现的时候可以理解为把这个操作真正的应用到K/V存储上去，比如Rocksdb。

2、Server State



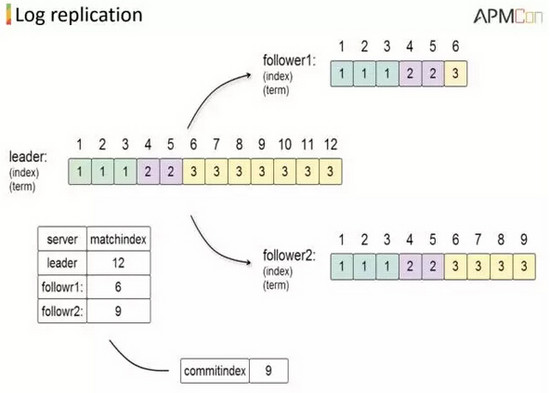
一个Raft组合相当于有很多节点，里面除了leader的角色还会有其他的角色，上图是一个角色转换图，一个新加入的节点首先是一个follower的角色，如果follower加入的时候发现没有leader就会变成Candidate的角色，发起选举准备竞选leader，如果这个candidate节点收到了大多数节点的投票，那么它就可以顺利的成为leader开始对外提供服务。但是如果此时没有选举成功且整组里还是没有leader，那么这个candidate会继续进行选举。如果这个candidate发起选举的时候发现已经有其他的节点成为了leader，那么它就会回到follower角色，这是一个典型的一个状态机，其中follower是起始状态，而follower和leader可以理解为终止状态，candidate是普通中间状态，也就是每一个节点的稳定状态最终一定会在停留在这两个状态中的一个，终止状态不代表一旦进入就要永远停留在这个状态，确切的说应该理解为一种相对稳定的状态。

3、Term & log index



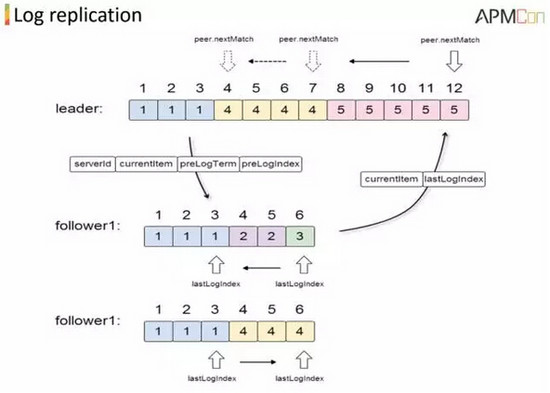
在介绍日志复制之前，需要介绍两个概念，Term和Log index，Term相当于一个抽象的逻辑时间单位，每个Term里只有一个leader，如果在这个Term里没有发生网络抖动或者leader出异常的情况也就是没有发生新的选举的情况，那么这个Term一直保持不变，leader也一直保持不变。Log index是指每条操作命令在记录到日志的时候所在日志的位置，可以将log理解为一个很大的数组，log index就像是这个大数组的下标，往Log里每添加一条操作，log index就加1。在添加操作的时候要同时记录该操作对应的index以及Term，所以会出现不同操作命令对应的Term是一样的情况，比如图上x=1这个操作Term是1，log index是1，y=1这个操作Term是2，log index是2，假如在此时发生了一次选举，后面的操作的记录情况可能出现如图上所示的情况，x=7这个操作Term=2，log index是3，y=5这个操作Term=2, log index是4，x=2这个操作Term=2，log index是5，假设此时又发生了选举，每次选举的时候Term都会往上增加，此时Term变成了3但是因为一些原因这个Term内选举没有成功，当这次选举失败以后发现还没有leader就继续将Term增大到4继续进行选举，所以后面的y=3的操作记录的时候对应的Term就是4，log index=6，紧接着之后y=6的操作Term=4，log index=7，后续其他操作的记录方式跟这个类似。

4、Log replication



下面我们再以日志复制的具体例子来分析一下日志是怎么在Raft的各个节点之间进行复制的。假设有3个节点，leader上有这些日志，从图上可以看到已经记录了有12条操作，方框里的数字是Term，从图上可以看到这个系统已经发生过几次选举，但是这个leader后来重新选举的时候又继续成为了leader，方框上面的数字代表的是log index，可以看到log index是依次递增的。这些所有的操作会被复制到所有的follower上，在leader这端相当于会维护一个信息记录每个follower都复制到哪里了，比如这个图上所示follower1已经复制到log index为6的位置了，而follow2已经复制到log index为9的位置了，因为leader自己已经到了log index为12的位置，那么在全局来说log index为9以及9之前的操作都已经复制到了大多数节点上，所以9这个位置及之前的操作都可以安全的apply到上层的状态机，在实际实现的时候可以理解为可以把这个操作安全的应用到Rocksdb上，而9这个位置也就是commit index, commit index代表的是这个log index所在位置及以前的操作在全局范围内已经复制到了大多数节点上是不会丢失的，本质是满足持久化条件的，所以可以安全的将这些操作真正的生效。

日志复制里还有两条非常关键的特性，在同一个Raft组里不同节点日志上的操作如果log index和Term是一样的，那么对应的操作一定是一样的；另一个是如果不同节点日志上的某个位置拥有相同的log index和Term，那么表示在这个位置之前的所有的操作都是一样的。



具体讲一下这个是怎么做到的，这里简单举个例子，上图示意的是一个leader和其中一个follower的情况，假设这个follower是follower1，从图上可以看出来Term为1的时候leader是leader，Term为2和3的时候，follower1成为了leader，不过从Term=4之后原来的leader又重新成为了leader，并且后来可能因为网络抖动等原因又发生了一次选举，leader节点依然被选为了leader，所以会看到Term=5的时候leader这个角色保持不变。

此时出现了一种什么情况呢，在follower1上我们可以看到Term=2和Term=3的时候对应的log index依次为4、5、6，而这三条操作在leader的日志里没有。这是一种什么样的情况呢，有可能的一种情况是在Term=2和Term3的时候，follower1成为了leader，把这三个操作记录到了自己的日志中，但是还没来得及复制到其他节点上结果因为网络抖动或者自身异常等一些原因，原来的leader重新发起选举又成为了leader。这个时候原来leader对外正常提供了服务，且日志里记录了Term=4及以后的操作，那这个follower1在接收leader日志的时候有一部分日志和leader的日志就是不一样的，这部分日志我们之前也已经提过需要剔除掉的。

那怎么剔除掉这部分日志呢？在这个follower重新连上leader的时候，leader会把当前自身的<serverid, currentITerm, preLogTerm, preLogIndex> 信息传给follower1,  preLogTerm和preLogIndex分别是leader当前日志里的最后一条日志的信息，在这个图中假设暂时没有新的发生选举，那么此时leader传给follower1的currentITerm、preLogTerm、preLogIndex分别是5,5,12。Follower1收到消息以后会检查自己日志的信息，一看自己当前的preLogTerm和preLogIndex分别是3和6，明显是不对应的，所以follower会告知leader自己当前的lastLogIndex，也就是自己当前日志的index位置，图中的情况follower1告诉leader自己的last log index是7，此时leader会直接到自己log index为7的位置准备把信息发给follower1，同时会把7位置之前的6位置的Term和log index信息发送给follower1，在leader上6位置的Term=4，而在follower1上6位置的Term=3，说明此时还是没有匹配上，于是leader会继续把该复制给follower1的日志的起始位置继续往日志起始的方向挪动，一直到log index=3的时候Term=3刚好对应上，那说明此时leader和follower1在log index=3之前的位置记录都是一样的，可以从log index=4的位置开始将操作记录一条一条的复制给follower1，从而做到即使因为一些极端的情况某些日志不一致了也可以将没有被复制到的大多数节点上的日志是安全清除。

最后小结一下，今天主要是讲了一下NewSQL怎么支持分布式事务，根据事务ACID的属性分析了一下，原子性主要是采用两阶段提交的方式来保证，一致性的话其实和传统的数据库的一致性一样，主要是数据库整体的完整性，隔离性的话可以考虑采用全局时间戳这类的方式实现，持久性的话采用的Raft的方式也就是对应CAP里的一致性。最后还讲了Raft师怎么工作的，最主要的就是两部分，一个是leader选举，一个是日志复制。

关于APMCon：

2016中国应用性能管理大会（简称APMCon 2016）于8月18日至19日在北京新云南皇冠假日酒店隆重召开。APMCon由听云、极客邦和InfoQ联合主办的作为国内APM领域最具影响力的技术大会，首次举办的APMCon以“驱动应用架构优化与创新”为主题，聚焦当前最为热门的移动端、Web端和Server端的性能监控和管理技术，整个会议设置包含了：性能可视化、服务端监控实践、运维自动化、数据库性能优化、APM云服务架构和HTML5调优最佳实践等话题，致力于推动APM在国内的成长与发展。

想阅读更多技术文章，请访问听云技术博客，访问听云官方网站感受更多应用性能优化魔力。

## MySQL Sharding详解\_小文正\_新浪博客

http://blog.sina.com.cn/s/blog\_821512b50101hyc1.html

一 背景

我们知道，当数据库中的数据量越来越大时，不论是读还是写，压力都会变得越来越大。采用MySQL Replication多master多slave方案，在上层做负载均衡，虽然能够一定程度上缓解压力。但是当一张表中的数据变得非常庞大时，压力还是 非常大的。试想，如果一张表中的数据量达到了千万甚至上亿级别的时候，不管是建索引，优化缓存等，都会面临巨大的性能压力。

二 定义

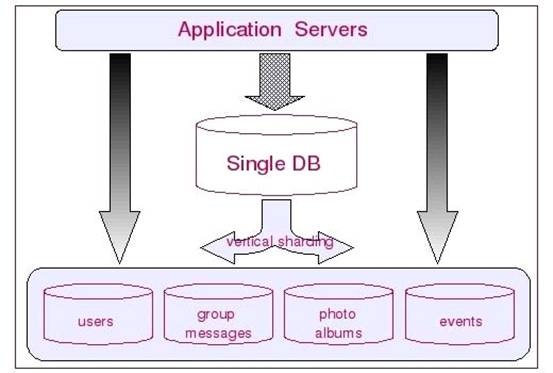
数据sharding，也称作数据切分，或分区。是指通过某种条件，把同一个数据库中的数据分散到多个数据库或多台机器上，以减小单台机器压力。

三 分类

数据分区根据切分规则，可以分为两类：

1、垂直切分

数据的垂直切分，也可以称之为纵向切分。将数据库想象成为由很多个一大块一大块的“数据块”（表）组成，我们垂直的将这些“数据块”切开，然后将他们分散到多台数据库主机上面。这样的切分方法就是一个垂直（纵向）的数据切分。以表为单位，把不同的表分散到不同的数据库或主机上。规则简单，实施方便，适合业务之间耦合度低的系统。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=821512b50101hyc1&url=http://album.sina.com.cn/pic/002nHcUtgy6GGtHi1Pd6b)

垂直切分的优点

（1）数据库的拆分简单明了，拆分规则明确；

（2）应用程序模块清晰明确，整合容易；

（3）数据维护方便易行，容易定位；

垂直切分的缺点

（1）部分表关联无法在数据库级别完成，需要在程序中完成；

（2）对于访问极其频繁且数据量超大的表仍然存在性能平静，不一定能满足要求；

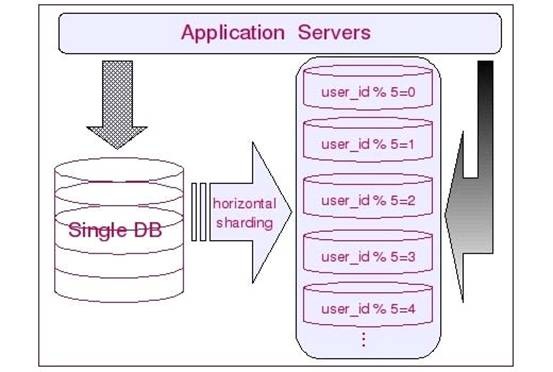
（3）事务处理相对更为复杂；

（4） 切分达到一定程度之后，扩展性会遇到限制；

（5）过读切分可能会带来系统过渡复杂而难以维护。

2、水平切分

一般来说，简单的水平切分主要是将某个访问极其平凡的表再按照某个字段的某种规则来分散到多个表之中，每个表中包含一部分数据。以行为单位，将同一个表中的数据按照某种条件拆分到不同的数据库或主机上。相对复杂，适合单表巨大的系统。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=821512b50101hyc1&url=http://album.sina.com.cn/pic/002nHcUtgy6GGtIT4gJ05)

水平切分的优点

（1）表关联基本能够在数据库端全部完成；

（2）不会存在某些超大型数据量和高负载的表遇到瓶颈的问题；

（3）应用程序端整体架构改动相对较少；

（4）事务处理相对简单；

（5）只要切分规则能够定义好，基本上较难遇到扩展性限制；

水平切分的缺点

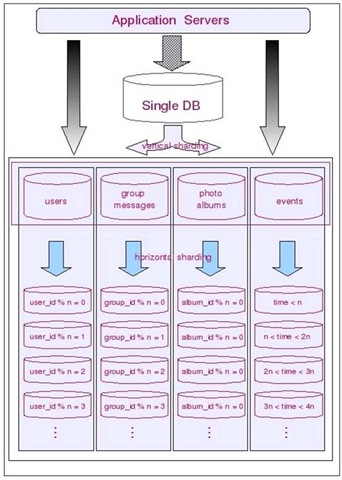
（1）切分规则相对更为复杂，很难抽象出一个能够满足整个数据库的切分规则；

（2）后期数据的维护难度有所增加，人为手工定位数据更困难；

（3）应用系统各模块耦合度较高，可能会对后面数据的迁移拆分造成一定的困难。

3、联合切分

实际的应用场景中，除了那些负载并不是太大，业务逻辑也相对较简单的系统可以通过上面两种切分方法之一来解决扩展性问题之外，恐怕其他大部分业务逻辑稍微复杂一点，系统负载大一些的系统，都无法通过上面任何一种数据的切分方法来实现较好的扩展性，而需要将上述两种切分方法结合使用，不同的场景使用不同的切分方法。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=821512b50101hyc1&url=http://album.sina.com.cn/pic/002nHcUtgy6GGtMiSDsb6)

联合切分的优点

（1）可以充分利用垂直切分和水平切分各自的优势而避免各自的缺陷；

（2）让系统扩展性得到最大化提升；

联合切分的缺点

（1）数据库系统架构比较复杂，维护难度更大；

（2）应用程序架构也相对更复杂；

四 实现方案

现在 Sharding 相关的软件实现其实不少，基于数据库层、DAO 层、不同语言下也都不乏案例。限于篇幅，此处只作一下简要的介绍。

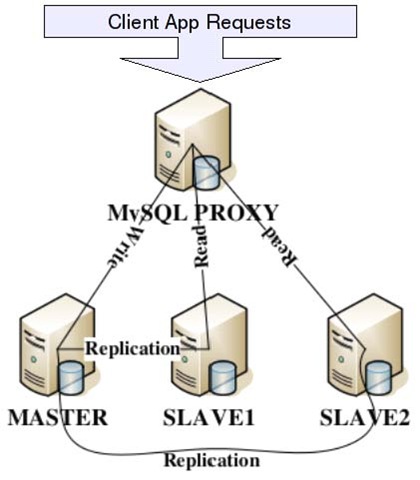
1、 Mysql Proxy + HASCALE

一套比较有潜力的方案。其中MySQL Proxy  是用 Lua 脚本实现的，介于客户端与服务器端之间，扮演Proxy 的角色，提供查询分析、失败接管、查询过滤、调整等功能。目前的 0.6 版本还做不到读、写分离。HSCALE 则是针对 MySQL Proxy 插件，也是用 Lua 实现的，对 Sharding 过程简化了许多。需要指出的是，MySQL Proxy 与 HSCALE 各自会带来一定的开销，但这个开销与集中式数据处理方式单条查询的开销还是要小的。

MySQLProxy是MySQL官方提供的一个数据库代理层产品，和MySQLServer一样，同样是一个基于GPL开源协议的开源产品。可用来监视、分析或者传输他们之间的通讯信息。他的灵活性允许你最大限度的使用它，目前具备的功能主要有连接路由，Query分析，Query过滤和修改，负载均衡，以及基本的HA机制等。

实际上，MySQLProxy本身并不具有上述所有的这些功能，而是提供了实现上述功能的基础。要实现这些功能，还需要通过我们自行编写LUA脚本来实现。

MySQLProxy实际上是在客户端请求与MySQLServer之间建立了一个连接池。所有客户端请求都是发向MySQLProxy，然后经由MySQLProxy进行相应的分析，判断出是读操作还是写操作，分发至对应的MySQLServer上。对于多节点Slave集群，也可以起做到负载均衡的效果。以下是MySQLProxy的基本架构图：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=821512b50101hyc1&url=http://album.sina.com.cn/pic/002nHcUtgy6GGtNvnnif8)

通过上面的架构简图，我们可以很清晰的看出MySQLProxy在实际应用中所处的位置，以及能做的基本事情。关于MySQLProxy更为详细的实施细则在MySQL官方文档中有非常详细的介绍和示例，感兴趣的读者朋友可以直接从MySQL官方网站免费下载或者在线阅读，我这里就不累述浪费纸张了。

http://forge.mysql.com/wiki/MySQL\_Proxy

2、 Hibernate Shards

这是 Google 技术团队贡献的项目（http://www.hibernate.org/414.html），该项目是在对Google 财务系统数据 Sharding 过程中诞生的。因为是在框架层实现的，所以有其独特的特性：标准的 Hibernate 编程模型，会用 Hibernate 就能搞定，技术成本较低；相对弹性的 Sharding 策略以及支持虚拟 Shard 等。

3、 Spock Proxy

这也是在实际需求中产生的一个开源项目，基于Mysql Proxy扩展。Spock（http://www.spock.com/）是一个人员查找的 Web 2.0 网站。通过对自己的单一 DB 进行有效 Sharding化 而产生了Spock Proxy(http://spockproxy.sourceforge.net/ ) 项目，Spock Proxy 算得上 MySQL Proxy 的一个分支，提供基于范围的 Sharding 机制。Spock 是基于 Rails 的，所以Spock Proxy 也是基于 Rails 构建，关注 ROR 的朋友不应错过这个项目。

http://spockproxy.sourceforge.net/

4、 Amoeba for MySQL

Amoeba是一个基于Java开发的，专注于解决分布式数据库数据源整合Proxy程序的开源框架，基于GPL3开源协议。目前，Amoeba已经具有Query路由，Query过滤，读写分离，负载均衡以及HA机制等相关内容。

Amoeba 主要解决的以下几个问题：

（1）数据切分后复杂数据源整合；

（2）提供数据切分规则并降低数据切分规则给数据库带来的影响；

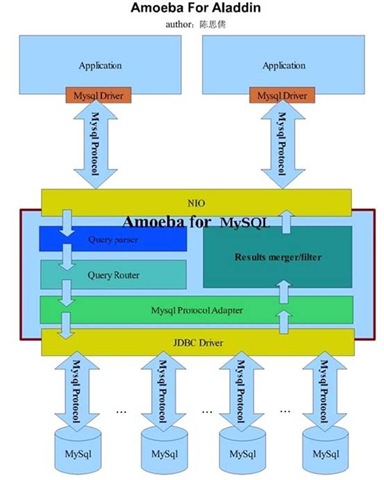
（3）降低数据库与客户端的连接数；

（4）读写分离路由。

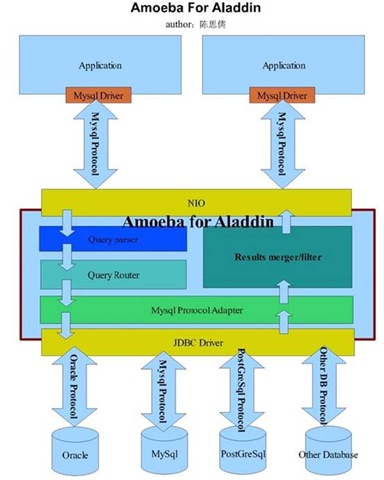
我们可以看出，Amoeba所做的事情，正好就是我们通过数据切分来提升数据库的扩展性所需要的。

Amoeba并不是一个代理层的Proxy程序，而是一个开发数据库代理层Proxy程序的开发框架，目前基于Amoeba所开发的Proxy程序有AmoebaForMySQL和AmoebaForAladin两个。

AmoebaForMySQL主要是专门针对MySQL数据库的解决方案，前端应用程序请求的协议以及后端连接的数据源数据库都必须是MySQL。对于客户端的任何应用程序来说，AmoebaForMySQL和一个MySQL数据库没有什么区别，任何使用MySQL协议的客户端请求，都可以被AmoebaForMySQL解析并进行相应的处理。下如可以告诉我们AmoebaForMySQL的架构信息（出自Amoeba开发者博客）：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=821512b50101hyc1&url=http://album.sina.com.cn/pic/002nHcUtgy6GGtPdXpxc3)

AmoebaForAladin则是一个适用更为广泛，功能更为强大的Proxy程序。他可以同时连接不同数据库的数据源为前端应用程序提供服务，但是仅仅接受符合MySQL协议的客户端应用程序请求。也就是说，只要前端应用程序通过MySQL协议连接上来之后，AmoebaForAladin会自动分析Query语句，根据Query语句中所请求的数据来自动识别出该所Query的数据源是在什么类型数据库的哪一个物理主机上面。下图展示了AmoebaForAladin的架构细节（出自Amoeba开发者博客）：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=821512b50101hyc1&url=http://album.sina.com.cn/pic/002nHcUtgy6GGtQKOlwae)

咋一看，两者好像完全一样嘛。细看之后，才会发现两者主要的区别仅在于通过MySQLProtocalAdapter处理之后，根据分析结果判断出数据源数据库，然后选择特定的JDBC驱动和相应协议连接后端数据库。

其实通过上面两个架构图大家可能也已经发现了Amoeba的特点了，他仅仅只是一个开发框架，我们除了选择他已经提供的ForMySQL和ForAladin这两款产品之外，还可以基于自身的需求进行相应的二次开发，得到更适应我们自己应用特点的Proxy程序。

当对于使用MySQL数据库来说，不论是AmoebaForMySQL还是AmoebaForAladin都可以很好的使用。当然，考虑到任何一个系统越是复杂，其性能肯定就会有一定的损失，维护成本自然也会相对更高一些。所以，对于仅仅需要使用MySQL数据库的时候，我还是建议使用AmoebaForMySQL。

AmoebaForMySQL的使用非常简单，所有的配置文件都是标准的XML文件，总共有四个配置文件。分别为：

（1）amoeba.xml：主配置文件，配置所有数据源以及Amoeba自身的参数设置；

（2）rule.xml：配置所有Query路由规则的信息；

（3）functionMap.xml：配置用于解析Query中的函数所对应的Java实现类；

（4）rullFunctionMap.xml：配置路由规则中需要使用到的特定函数的实现类；

如果您的规则不是太复杂，基本上仅需要使用到上面四个配置文件中的前面两个就可完成所有工作。Proxy程序常用的功能如读写分离，负载均衡等配置都在amoeba.xml中进行。此外，Amoeba已经支持了实现数据的垂直切分和水平切分的自动路由，路由规则可以在rule.xml进行设置。

目前Amoeba少有欠缺的主要就是其在线管理功能以及对事务的支持了，曾经在与相关开发者的沟通过程中提出过相关的建议，希望能够提供一个可以进行在线维护管理的命令行管理工具，方便在线维护使用，得到的反馈是管理专门的管理模块已经纳入开发日程了。另外在事务支持方面暂时还是Amoeba无法做到的，即使客户端应用在提交给Amoeba的请求是包含事务信息的，Amoeba也会忽略事务相关信息。当然，在经过不断完善之后，我相信事务支持肯定是Amoeba重点考虑增加的feature。

关于Amoeba更为详细的使用方法读者朋友可以通过Amoeba开发者博客（http://amoeba.sf.net）上面提供的使用手册获取，这里就不再细述了。

案例（http://pengranxiang.iteye.com/blog/1145342）

操作文档（http://docs.hexnova.com/amoeba/chap-getting-started.html）

5、 HiveDB

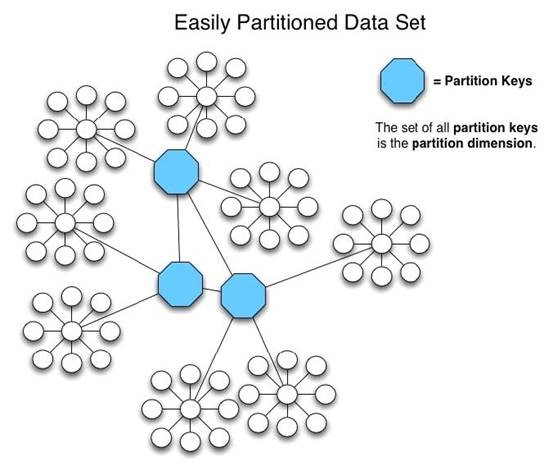
和前面的MySQLProxy以及Amoeba一样，HiveDB同样是一个基于Java针对MySQL数据库的提供数据切分及整合的开源框架，只是目前的HiveDB仅仅支持数据的水平切分。主要解决大数据量下数据库的扩展性及数据的高性能访问问题，同时支持数据的冗余及基本的HA机制。

HiveDB的实现机制与MySQLProxy和Amoeba有一定的差异，他并不是借助MySQL的Replication功能来实现数据的冗余，而是自行实现了数据冗余机制，而其底层主要是基于HibernateShards来实现的数据切分工作。

在HiveDB中，通过用户自定义的各种Partitionkeys（其实就是制定数据切分规则），将数据分散到多个MySQLServer中。在访问的时候，在运行Query请求的时候，会自动分析过滤条件，并行从多个MySQLServer中读取数据，并合并结果集返回给客户端应用程序。

单纯从功能方面来讲，HiveDB可能并不如MySQLProxy和Amoeba那样强大，但是其数据切分的思路与前面二者并无本质差异。此外，HiveDB并不仅仅只是一个开源爱好者所共享的内容，而是存在商业公司支持的开源项目。

下面是HiveDB官方网站上面一章图片，描述了HiveDB如何来组织数据的基本信息，虽然不能详细的表现出太多架构方面的信息，但是也基本可以展示出其在数据切分方面独特的一面了。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=821512b50101hyc1&url=http://album.sina.com.cn/pic/002nHcUtgy6GGtRWYB147)

http://www.hivedb.org/

6、 DataFabric

application-level sharding

master/slave replication

https://github.com/bpot/data\_fabric

7、PL/Proxy

前面几个都是针对MySQL 的 Sharding 方案，PL/Proxy 则是针对 PostgreSQL 的，设计思想类似 Teradata的 Hash 机制，数据存储对客户端是透明的，客户请求发送到 PL/Proxy 后，由这里分布式存储过程调用，统一分发。 PL/Proxy 的设计初衷就是在这一层充当”数据总线”的职责，所以，当数据吞吐量支撑不住的时候，只需要增加更多的 PL/Proxy 服务器即可。大名鼎鼎的 Skype 用的就是 PL/Proxy 的解决方案。

8、Pyshards

这是个基于Python的解决方案。该工具的设计目标还有个 Re-balancing 在里面，这倒是个比较激进的想法。目前只支持 MySQL 数据库。

http://code.google.com/p/pyshards/wiki/Pyshards

9、其他实现数据切分及整合的解决方案

除了上面介绍的几个数据切分及整合的整体解决方案之外，还存在很多其他同样提供了数据切分与整合的解决方案。如基于MySQLProxy的基础上做了进一步扩展的HSCALE，通过Rails构建的SpockProxy，以及基于Pathon的Pyshards等等。

不管大家选择使用哪一种解决方案，总体设计思路基本上都不应该会有任何变化，那就是通过数据的垂直和水平切分，增强数据库的整体服务能力，让应用系统的整体扩展能力尽可能的提升，扩展方式尽可能的便捷。

只要我们通过中间层Proxy应用程序较好的解决了数据切分和数据源整合问题，那么数据库的线性扩展能力将很容易做到像我们的应用程序一样方便，只需要通过添加廉价的PCServer服务器，即可线性增加数据库集群的整体服务能力，让数据库不再轻易成为应用系统的性能瓶颈。

五 注意事项

下面我们所说的分区，主要是指水平分区。

1、在实施分区前，我们可以查看所安装版本的mysql是否支持分区：

mysql> show variables like "%partition%";

如果支持则会显示：

        +-------------------+-------+

        | Variable\_name     | Value |

        +-------------------+-------+

        | have\_partitioning | YES   |

        +-------------------+-------+

2、分区适用于一个表的所有数据和索引，不能只对数据分区而不对索引分区，反之亦然，同时也不能只对表的一部分进行分区。

3、分区类型

（1）RANGE 分区：基于属于一个给定连续区间的列值，把多行分配给分区。

（2）LIST 分区：类似于按RANGE分区，区别在于LIST分区是基于列值匹配一个离散值集合中的某个值来进行选择。

（3）HASH分区：基于用户定义的表达式的返回值来进行选择的分区，该表达式使用将要插入到表中的这些行的列值进行计算（新浪微博采用的方案）。

（4）KEY 分区：类似于按HASH分区，区别在于KEY分区只支持计算一列或多列，且MySQL 服务器提供其自身的哈希函数。必须有一列或多列包含整数值。

无论使用何种类型的分区，分区总是在创建时就自动的顺序编号，且从0开始记录。当有一新行插入到一个分区表中时，就是使用这些分区编号来识别正确的分区。

4、MySQL提供了许多修改分区表的方式。添加、删除、重新定义、合并或拆分已经存在的分区是可能的。所有这些操作都可以通过使用ALTER TABLE 命令的分区扩展来实现。

5、可以对已经存在的表进行分区，直接使用alter table命令即可。

六 数据切分与整合可能存在的问题

这里，大家应该对数据切分与整合的实施有了一定的认识了，或许很多读者朋友都已经根据各种解决方案各自特性的优劣基本选定了适合于自己应用场景的方案，后面的工作主要就是实施准备了。

在实施数据切分方案之前，有些可能存在的问题我们还是需要做一些分析的。一般来说，我们可能遇到的问题主要会有以下几点：

1、引入分布式事务的问题

一旦数据进行切分被分别存放在多个MySQLServer中之后，不管我们的切分规则设计的多么的完美（实际上并不存在完美的切分规则），都可能造成之前的某些事务所涉及到的数据已经不在同一个MySQLServer中了。

在这样的场景下，如果我们的应用程序仍然按照老的解决方案，那么势必需要引入分布式事务来解决。而在MySQL各个版本中，只有从MySQL5.0开始以后的各个版本才开始对分布式事务提供支持，而且目前仅有Innodb提供分布式事务支持。不仅如此，即使我们刚好使用了支持分布式事务的MySQL版本，同时也是使用的Innodb存储引擎，分布式事务本身对于系统资源的消耗就是很大的，性能本身也并不是太高。而且引入分布式事务本身在异常处理方面就会带来较多比较难控制的因素。

怎么办？其实我们可以可以通过一个变通的方法来解决这种问题，首先需要考虑的一件事情就是：是否数据库是唯一一个能够解决事务的地方呢？其实并不是这样的，我们完全可以结合数据库以及应用程序两者来共同解决。各个数据库解决自己身上的事务，然后通过应用程序来控制多个数据库上面的事务。

也就是说，只要我们愿意，完全可以将一个跨多个数据库的分布式事务分拆成多个仅处于单个数据库上面的小事务，并通过应用程序来总控各个小事务。当然，这样作的要求就是我们的俄应用程序必须要有足够的健壮性，当然也会给应用程序带来一些技术难度。

2、跨节点Join的问题

上面介绍了可能引入分布式事务的问题，现在我们再看看需要跨节点Join的问题。数据切分之后，可能会造成有些老的Join语句无法继续使用，因为Join使用的数据源可能被切分到多个MySQLServer中了。

怎么办？这个问题从MySQL数据库角度来看，如果非得在数据库端来直接解决的话，恐怕只能通过MySQL一种特殊的存储引擎Federated来解决了。Federated存储引擎是MySQL解决类似于Oracle的DBLink之类问题的解决方案。和OracleDBLink的主要区别在于Federated会保存一份远端表结构的定义信息在本地。咋一看，Federated确实是解决跨节点Join非常好的解决方案。但是我们还应该清楚一点，那就似乎如果远端的表结构发生了变更，本地的表定义信息是不会跟着发生相应变化的。如果在更新远端表结构的时候并没有更新本地的Federated表定义信息，就很可能造成Query运行出错，无法得到正确的结果。

对待这类问题，我还是推荐通过应用程序来进行处理，先在驱动表所在的MySQLServer中取出相应的驱动结果集，然后根据驱动结果集再到被驱动表所在的MySQLServer中取出相应的数据。可能很多读者朋友会认为这样做对性能会产生一定的影响，是的，确实是会对性能有一定的负面影响，但是除了此法，基本上没有太多其他更好的解决办法了。而且，由于数据库通过较好的扩展之后，每台MySQLServer的负载就可以得到较好的控制，单纯针对单条Query来说，其响应时间可能比不切分之前要提高一些，所以性能方面所带来的负面影响也并不是太大。更何况，类似于这种需要跨节点Join的需求也并不是太多，相对于总体性能而言，可能也只是很小一部分而已。所以为了整体性能的考虑，偶尔牺牲那么一点点，其实是值得的，毕竟系统优化本身就是存在很多取舍和平衡的过程。

3、跨节点合并排序分页问题

一旦进行了数据的水平切分之后，可能就并不仅仅只有跨节点Join无法正常运行，有些排序分页的Query语句的数据源可能也会被切分到多个节点，这样造成的直接后果就是这些排序分页Query无法继续正常运行。其实这和跨节点Join是一个道理，数据源存在于多个节点上，要通过一个Query来解决，就和跨节点Join是一样的操作。同样Federated也可以部分解决，当然存在的风险也一样。

还是同样的问题，怎么办？我同样仍然继续建议通过应用程序来解决。

如何解决？解决的思路大体上和跨节点Join的解决类似，但是有一点和跨节点Join不太一样，Join很多时候都有一个驱动与被驱动的关系，所以Join本身涉及到的多个表之间的数据读取一般都会存在一个顺序关系。但是排序分页就不太一样了，排序分页的数据源基本上可以说是一个表（或者一个结果集），本身并不存在一个顺序关系，所以在从多个数据源取数据的过程是完全可以并行的。这样，排序分页数据的取数效率我们可以做的比跨库Join更高，所以带来的性能损失相对的要更小，在有些情况下可能比在原来未进行数据切分的数据库中效率更高了。当然，不论是跨节点Join还是跨节点排序分页，都会使我们的应用服务器消耗更多的资源，尤其是内存资源，因为我们在读取访问以及合并结果集的这个过程需要比原来处理更多的数据。

分析到这里，可能很多读者朋友会发现，上面所有的这些问题，我给出的建议基本上都是通过应用程序来解决。大家可能心里开始犯嘀咕了，是不是因为我是DBA，所以就很多事情都扔给应用架构师和开发人员了？

其实完全不是这样，首先应用程序由于其特殊性，可以非常容易做到很好的扩展性，但是数据库就不一样，必须借助很多其他的方式才能做到扩展，而且在这个扩展过程中，很难避免带来有些原来在集中式数据库中可以解决但被切分开成一个数据库集群之后就成为一个难题的情况。要想让系统整体得到最大限度的扩展，我们只能让应用程序做更多的事情，来解决数据库集群无法较好解决的问题。

七 MySQL基于Amoeba的水平和垂直分片示例

 详细的配置描述请参考：http://pengranxiang.iteye.com/blog/1145342

## MySQL高效处理大量并发的数据库连接\_hlidc\_新浪博客

http://blog.sina.com.cn/s/blog\_794a8d2e0100tc8t.html

单机单MySQL服务器处理数据库连接请求利用多线程机制，但正如任何处理方式一样都会有它的功能瓶颈。单机单MySQL服务器如何提高处理瞬时成千上万的数据库连接请求呢?

　　首先，我们从MySQL服务器的工作原理出发来思考问题。MySQL服务器利用多线程机制来充分发挥对多用户访问的支持，首先它分析客户发送过来的连接字符串，在数据库连接的线程缓存中寻找是否有这样的线程可以支持处理这一连接访问请求，如果存在就激活它;否则就新建一个线程来处理客户的访问连接请求。当多个数据库连接请求访问的是同一个数据库对象或者是同一部分数据，就会触发同步机制使得一些请求处于等待状态。

　　这就引出两个问题，首先是MySQL服务器是否可以虚拟为多个服务器来提供对数据库连接请求的处理?再就是如何对客户请求的资源进行快速响应，减少等待的时间。

　　第一个问题，通过开放多个端口来实现。同一服务上开启多个MySQL端口有两种方式，一种是在编译的时候指定特定端口，第二是修改配置文件来达到{mod}多个端口的目的。通过对MySQL\_multi.conf修改可以实现第二种方式。MySQLd\_multi可以管理多个帧听不同Unix套接字文件和TCP/IP端口的连接的MySQLd进程。它可以启动或停止服务器，或报告它们的当前状态。程序寻找my.cnf中的[MySQLdN]组(或--config-file选项指定的文件)。N 可以为任何正整数。

　　同时，在my.cnf配置文件中对以下各参数进行适当的设置。首先是与网络连接的性能配置项。

　　Max\_connections：整个MySQL允许的最大连接数。

　　如果客户的连接请求数超过max\_connections，就会产生大量连接请求的等待，所以应该在主机性能允许的情况下，将该参数设置的尽可能大一些。

　　Max\_user\_connections：每个用户允许的最大连接数。

　　Net\_buffer\_length：网络包传输中，传输消息之前的net buffer初始化大小。

　　Max\_allowed\_packet：在网络传输中，一次消息传输量的最大值

　　Back\_log：在MySQL的连接请求等待队列中允许存放的最大连接请求数。

　　再次是与连接线程相关的系统参数及状态变量

　　Thread\_cache\_size：Thread Cache池中应该存放的连接线程数。

　　Thread\_stack：每个连接线程被创建时，MySQL给它分配的内存大小。

　　第二个问题，主要通过以下几个方面来处理

　　第一， 适当分库。根据业务划分多个数据库，即database。通过SQL命令 showdatabases可以显示所有的数据库。这样可以将数据资源的访问进行有效地均衡，使得在各个业务数据的访问之间减少干扰，这样通过访问字符串进行数据库连接的时候也可以更为有效的利用MySQL的多线程机制。而且，通过这样的处理，也有效减少了对同一部分数据的访问。

　　第二，对MySQL进行Query Cache优化。

# mysql

## 打开MySQL数据库远程访问的权限 - whh - 博客园

http://www.cnblogs.com/ycsfwhh/archive/2012/08/07/2626597.html

GRANT ALL PRIVILEGES ON \*.\* TO 'admin'@'%'IDENTIFIED BY '123456' WITH GRANT OPTION;

FLUSH PRIVILEGES;

在我们使用mysql数据库时，有时我们的程序与数据库不在同一机器上，这时我们需要远程访问数据库。缺省状态下，mysql的用户没有远程访问的权限。下面介绍两种方法，解决这一问题。

1、改表法

可能是你的帐号不允许从远程登陆，只能在localhost。这个时候只要在localhost的那台电脑，登入mysql后，更改 "mysql" 数据库里的 "user" 表里的 "host" 项，从"localhost"改称"%"

mysql -u root -p

mysql>use mysql;

mysql>update user set host = '%' where user = 'root';

mysql>select host, user from user;

2、授权法

在安装mysql的机器上运行：

1、d:\mysql\bin\>mysql -h localhost -u root

//这样应该可以进入MySQL服务器

2、mysql>GRANT ALL PRIVILEGES ON \*.\* TO 'root'@'%'WITH GRANT OPTION

//赋予任何主机访问数据的权限

例如，你想myuser使用mypassword从任何主机连接到mysql服务器的话。

GRANT ALL PRIVILEGES ON \*.\* TO 'myuser'@'%'IDENTIFIED BY 'mypassword' WI

TH GRANT OPTION;

如果你想允许用户myuser从ip为192.168.1.6的主机连接到mysql服务器，并使用mypassword作为密码

GRANT ALL PRIVILEGES ON \*.\* TO 'myuser'@'192.168.1.3'IDENTIFIED BY

'mypassword' WITH GRANT OPTION;

3、mysql>FLUSH PRIVILEGES

//修改生效

4、mysql>EXIT

退出MySQL服务器，这样就可以在其它任何的主机上以root身份登录

分类: 软件开发

## Linux下修改Mysql的用户(root)的密码 - !!!!!!!!!!! - 博客园

http://www.cnblogs.com/daizhuacai/archive/2013/01/17/2865138.html

mysqladmin -u root password "123456"

修改的用户都以root为列。

一、拥有原来的myql的root的密码；

方法一：

在mysql系统外，使用mysqladmin

# mysqladmin -u root -p password "test123"

Enter password: 【输入原来的密码】

方法二：

通过登录mysql系统，

# mysql -uroot -p

Enter password: 【输入原来的密码】

mysql>use mysql;

mysql> update user set password=passworD("test") where user='root';

mysql> flush privileges;

mysql> exit;

二、忘记原来的myql的root的密码；

首先，你必须要有操作系统的root权限了。要是连系统的root权限都没有的话，先考虑root系统再走下面的步骤。

类似于安全模式登录系统，有人建议说是pkill mysql，但是我不建议哈。因为当你执行了这个命令后，会导致这样的状况：

/etc/init.d/mysqld status

mysqld dead but subsys locked

这样即使你是在安全模式下启动mysql都未必会有用的，所以一般是这样/etc/init.d/mysqld stop，如果你不幸先用了pkill，那么就start一下再stop咯。

# mysqld\_safe --skip-grant-tables &

&，表示在后台运行，不再后台运行的话，就再打开一个终端咯。

# mysql

mysql> use mysql;

mysql> UPDATE user SET password=password("test123") WHERE user='root';

mysql> flush privileges;

mysql> exit;

##本来mysql是不分大小写的，但是这个是修改的mysql中的mysql数据库的具体的值，要注意到。

## MySQL修改root密码的多种方法\_百度经验

http://jingyan.baidu.com/article/0320e2c198ad5f1b87507bc8.html

方法1： 用SET PASSWORD命令

　　mysql -u root

　　mysql> SET PASSWORD FOR 'root'@'localhost' = PASSWORD('newpass');

方法2：用mysqladmin

　　mysqladmin -u root password "newpass"

　　如果root已经设置过密码，采用如下方法

　　mysqladmin -u root password oldpass "newpass"

方法3： 用UPDATE直接编辑user表

　　mysql -u root

　　mysql> use mysql;

　　mysql> UPDATE user SET Password = PASSWORD('newpass') WHERE user = 'root';

　　mysql> FLUSH PRIVILEGES;

在丢失root密码的时候，可以这样

　　mysqld\_safe --skip-grant-tables&

　　mysql -u root mysql

　　mysql> UPDATE user SET password=PASSWORD("new password") WHERE user='root';

　　mysql> FLUSH PRIVILEGES;

## mysql-python安装时EnvironmentError: mysql\_config not found - 涛光 - 博客园

http://www.cnblogs.com/xiazh/archive/2012/12/12/2814289.html

在安装 mysql-python时，会出现：

复制代码

sh: mysql\_config: not found

Traceback (most recent call last):

File "setup.py", line 15, in <module>

metadata, options = get\_config()

File "/home/zhxia/apps/source/MySQL-python-1.2.3/setup\_posix.py", line 43, in get\_config

libs = mysql\_config("libs\_r")

File "/home/zhxia/apps/source/MySQL-python-1.2.3/setup\_posix.py", line 24, in mysql\_config

raise EnvironmentError("%s not found" % (mysql\_config.path,))

EnvironmentError: mysql\_config not found

复制代码

只要原因是没有安装:libmysqlclient-dev

sudo apt-get install libmysqlclient-dev

找到mysql\_config文件的路径

sudo updatedb

locate mysql\_config

mysql\_config的位置为：/usr/bin/mysql\_config

在mysql-python源码包下找到：setup\_posix.py 文件，然后找到文件中的 mysql\_config.path 将其值改为：/usr/bin/mysql\_config,然后 sudo python setup.py install ,就ok了

## Pinterest 数据切分：解惑快速扩展 MySQL 的数据量 - 推酷

http://www.tuicool.com/articles/U3qqyum

时间 2016-08-31 11:40:47  [开源中国翻译文章](http://www.tuicool.com/sites/QBvMFj" \t "_blank)

原文  [http://www.oschina.net/translate/sharding-pinterest-how-we-scaled-our-mysql-fleet](http://www.oschina.net/translate/sharding-pinterest-how-we-scaled-our-mysql-fleet?utm_source=tuicool&utm_medium=referral)

主题 [MySQL](http://www.tuicool.com/topics/11030000)

这是一个探讨怎样在多个MySQL服务器之间切分数据的技术问题。早在2012年我们就完成了这样分片方案，我们现在仍在使用这套方案存储核心数据。

在讨论怎样切分数据之前，我们不妨先与数据亲密接触。机舱里的灯光，附在草莓上的巧克力，星际迷航开始..

Pinterest是一个发现引擎， 寻找所有你所感兴趣的 。从数据的角度来看，Pinterest是世界上一大型的人类构建的的兴趣图。有超过500亿的Pin被Pin友保存 到10亿的板块上，用户再次Pin，喜欢其它人的Pin（简单的浅拷贝），关注其它Pin友、 板块 和兴趣，然后查看主页上所订阅Pin友的所有资讯、画板和他们关注的兴趣。太棒了，现在来拓展吧。

成长的痛

2011年我们公司获得大家的认同。 [***在***](http://www.zdnet.com/article/sap-netbase-pinterest-is-fastest-growing-social-site-ever/) [***一些***](http://techcrunch.com/2011/11/26/pinterest-viral/) [***评估***](http://www.businessinsider.com/holy-smoke-pinterest-is-the-fastest-growing-site-ever-2012-2) 里，我们的发展比先前任何初创公司都要快。大约在2011年9月份，我们的基础设施都超过负载。一些NoSQL技术导致灾难性的后果，同时大量用于读的MySQL从属服务器，产生了大量令人恼火的Bug，特别是缓存。我们重新架构了整个存储模型。令人欣喜的是，新架构还是比较有效果的，基本满足了我们的要求。

业务需求

* 系统总体要非常稳定，便于操作，便于拓展。我们想让数据库能从开始小存储量，能随着业务发展而拓展
* Pin友生成的内容必须能永久访问
* (支持）请求的N个Pin在板块中以确定的顺序(像按照创建时间倒序，或是按照用户特定的排序)（显示）。对于喜欢的Pin友，发Pin的Pin友列表等，也必须以特定的顺序.
* 为了简单，更新一般而言要保证最好的性能，为了获取最终一致性，需要额外的东西，如 [***分布式事物日志***](https://en.wikipedia.org/wiki/Transaction_log) 。这是一个有趣（但不太容易）的事情!

设计原理与笔记

由于我们想要的这些数据是横跨多个数据库的，我们不能使用数据库的join，外键或者收集所有数据的索引，不过他们可以被用于不能横跨数据库的子查询。

我们也需要支持负载均衡我们的数据。我们厌恶来回移动数据，尤其是逐项移动，因为容易出错，也容易让系统变得不必要的复杂。如果我们必须移动数据，最好移动一整个虚拟节点到物理节点。

为了实现快速成型，我们需要一个简单可用的解决方案，并且在我们的分布式数据平台上，节点要非常稳定。

所有的数据都需要被备份到从节点进行高可用，并为 MapReduce 转存到 S3。 在生产中，我们只与主节点交互。 **在生产中，你不能在从节点上** **读/写。** 从节点是滞后的，它会引发奇怪的bug。如果你共享数据，一般来说在生产中与从节点交互是没有优势的。

最终，我们需要一个好的方式来生成一个统一且唯一的ID（UUID）分配给所有我们的对象。

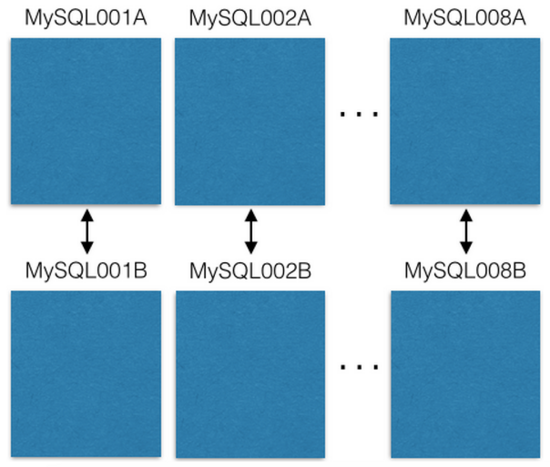
我们如何切分数据

无论我们如何构建我们的系统，都需要满足我们的业务要求并保证系统稳定，具有高性能，易于维护。换言之，我们需要我们的系统不糟糕 ，因此我们选择成熟的技术，MySQL作为我们构建系统的基础。我们有意避免使用具有自动扩展功能的新技术，例如 MongoDB, Cassandra 和 Membase等，因为他们还不够成熟（并且他们会以无法预知的方式崩溃）。

悄悄话： 我依旧建议初创公司避免使用花哨的新事物——尝试使用完全能够正常运行的MySQL.相信我,我有很多错误的实践（创伤）来证明这一点。

MySQL是成熟的，稳定的并且能够正常运行的。不仅我们使用MySQL，还有大量公司在广泛的使用.MySQL支持我们需要的：数据排序、范围查询功能并且具备行级事务。它还有很多的特性，但是我们不需要或使用它们。尽管MySQL非常适合我们，但是MySQL是单一解决方案，因此需要我们对数据进行分片。下面是我们的解决方案：

最开始时我们有8个 EC2 服务,每个服务对应一个MySQL实例：



为了应对MySQL服务崩溃，每一个MySQL服务都是应用主复制模式构建，即每一个MySQL服务都将数据备份到一个备份主机上。我们的应用只读写主服务.我建议你也这样做。它简化了一切操作并且避免了复制延迟问题。

每个MySQL数据库可以有多个库:



但每个库的命名唯一，如db00000， db00001， ...，dbNNNNN。 每个库都是我们数据的一个分片。我们的决计决策是一块数据只在一个分片中，而不会在其它片中。然而你可以通过增加容量，以便将片移到其它机器上（后续将讨论这个）。

我们维护了一个配置表，以记录这些数据分片在什么机器上：

**[{“range”: (0,511), “master”: “MySQL001A”, “slave”: “MySQL001B”},**

**{“range”: (512, 1023), “master”: “MySQL002A”, “slave”: “MySQL002B”},**

**...**

**{“range”: (3584, 4095), “master”: “MySQL008A”, “slave”: “MySQL008B”}]**

这个版本仅在我们需要移动分片或是替换某台机器时修改。如果主机挂掉了，我们可以将某台从属服务器升级为主机，然后再加台新的从属服务器。其配置保存在 [***ZooKeeper***](https://engineering.pinterest.com/blog/serving-configuration-data-scale-high-availability) ，更新时，配置发送到其它维护分片的MySQL服务器上。

每个分片包含相同的表：pins, boards, users\_has\_pins, users\_likes\_pins, pin\_liked\_by\_user等等。我后面什么时候会详细讲解。

那怎样将数据部署到这些分片上呢？

我们创建了一个64位的ID，包含了分片ID，包含数据的类型，及数据在那台服务器上（本地ID)。分片ID是10位，本地ID是36位。精明的专家对此提示只能加到62位。以前编译器和芯片设计的经验已经教育过我，预留位的长度是很重要的，所以我们有两个（设置为0）。

ID = (shard ID << 46) | (type ID << 36) | (local ID<<0)

给定这个Pin: [***https://www.pinterest.com/pin/241294492511762325/***](https://www.pinterest.com/pin/241294492511762325/) ，让我们开发剖析这个Pin ID 241294492511762325:

Shard ID = (241294492511762325 >> 46) & 0xFFFF = 3429

Type ID = (241294492511762325 >> 36) & 0x3FF = 1

Local ID = (241294492511762325 >> 0) & 0xFFFFFFFFF = 7075733

那这个Pin对象保存在3429分片上，它的类型是1(如‘Pin’)，及它在pin表的所在行是7075733。 举个例子，让我们假设这个分片是在MySQL012A.上，我们可能通过下面来访问到这个对象:

conn = MySQLdb.connect(host=”MySQL012A”)

conn.execute(“**SELECT** data **FROM** db03429.pins **where** local\_id=7075733”)

这些数据 有两个类型：objects（对象）和mappings（映射）。objects包含了具体的数据，例如Pin的数据。

对象表！

对象表存储了如Pin数据、用户数据、看板（boards）数据以及评论数据。每个对象都包含一个ID列（本地的ID，自增主键），和一个blob字段包含整个对象的JSON数据。

**CREATE** **TABLE** pins (

local\_id **INT** **PRIMARY** **KEY** AUTO\_INCREMENT, data TEXT,

ts **TIMESTAMP** **DEFAULT** **CURRENT\_TIMESTAMP**) ENGINE=InnoDB;

例如一个Pin数据:

{“details”: “**New** Star Wars character”, “link”: “http:*//webpage.com/asdf”, “user\_id”: 241294629943640797, “board\_id”: 241294561224164665, …}*

在创建新Pin的时候，需要将所有的对象数据收集起来并生成一个JSON blob。然后，需要确定一个分片（shard）ID（我们偏向于选择与它关联（inserted）的看板的分片ID相同的ID，但是没有必要）。Pin的类型码为1。连上数据库并在Pin对象表中插入JSON数据。MySQL会返回一个本地自增ID。一旦生成了分片ID、类型码和本地ID，就可以生成完整的64位ID！

编辑Pin的时候可以在一个 [***MySQL事务***](http://dev.mysql.com/doc/refman/5.7/en/innodb-locking-reads.html) 中以读-修改-写的方式修改JSON数据：

> **BEGIN**

> **SELECT** blob **FROM** db03429.pins **WHERE** local\_id=7075733 **FOR** **UPDATE**

[Modify the json blob]

> **UPDATE** db03429.pins **SET** blob=’<modified blob>’ **WHERE** local\_id=7075733

> **COMMIT**

删除Pin的时候可以删除MySQL中的一行。但是更好的方式是在JSON数据中添加一个‘active’字段，设置值为false，并且在客户端上过滤到这些信息。

映射表

映射表用于建立对象之间的连接，如看板（board）和Pin之间的连接。MySQL中建立的映射表包含3列，一个64位的来源对象ID‘from’、一个目的对象ID ‘to’ 以及一个序列ID ‘sequence’ 。在(from, to, sequence)上建立联合索引，并且每条记录以 ‘from’ 字段的分片ID分片。

**CREATE** **TABLE** board\_has\_pins (

board\_id **INT**,

pin\_id **INT**, sequence **INT**, INDEX(board\_id, pin\_id, sequence)

) ENGINE=InnoDB;

映射表中保存单向映射关系，如看板（board）到Pin的映射表‘board\_has\_pins’。如果需要反向的映射关系，需要建立一个独立的反向映射表‘pin\_owned\_by\_board’。‘sequence’ 主要用于排序（分片之间是不可以比较的，因为本地ID可能相互是冲突的）。当插入一个新的Pin到看板的时候，通常可以用unix本地的时间戳作为序列号（sequence ID = unix timestamp）。序列号可以是任意的数字，但是unix时间戳是一种使新增的列处于更大的数值的便捷方式，因为时间是单调递增的。查询映射表的方式如下：

**SELECT** pin\_id **FROM** board\_has\_pins

**WHERE** board\_id=241294561224164665 **ORDER** **BY** sequence LIMIT 50 OFFSET 150

这样可以返回至多50个pin\_id，再通过它们获取Pin对象。

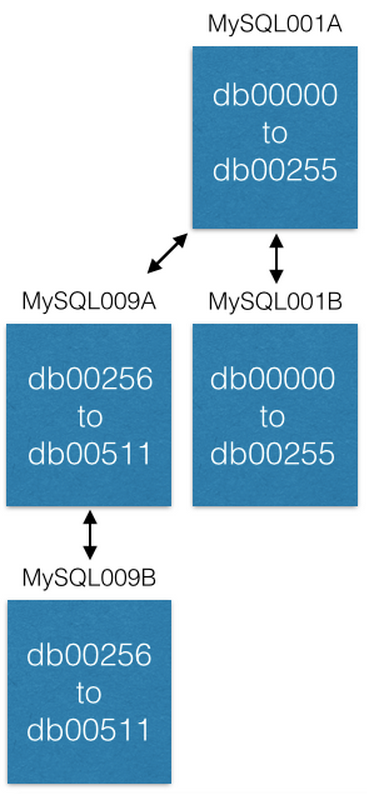
以上的方法便是一次应用层的连接(board\_id -> pin\_ids -> pin objects)。应用层连接的一个杰出的特性是可以在独立于对象数据的情况下缓存映射数据。在memcache集群中缓存pin\_id到pin对象的映射数据的同时还在redis集群中缓存board\_id到pin\_ids的映射数据。这样就可以实现与缓存数据更匹配的缓存技术。

扩容

我们系统有三种主要的扩容方式。最简单的就是升级机器（容量更大、速度更快的硬盘，更多内存，任何成为瓶颈的部分）。

第二种方式是开启新的分片范围。虽然我们的分片ID是16位的（最多64k个分片），但是一开始我们只创建了4096个分片。新建的对象只能存储在开头的4k个分片。某天我们决定添加新的MySQL服务器用来存储4096到8191的分片，并且开始填充数据。

最后一种方式是将一些分片移动到新的机器上。例如需要对MySQL001A（存储着0到511分片）扩容是，先新建一对主从节点（译者注：此处原文为“a new master-master pair”，可能为笔误），命名为最大的编号（比如是MySQL009A和MySQL009B），然后开始从MySQL001A复制数据。



当复制完成时，改变配置，使MySQL001A只存储0到255分片，MySQL009A只存储256到511分片。现在每个服务器只处理原先一般的数据量。



一些优点

如果你曾构建过生成 [***UUIDs***](https://en.wikipedia.org/wiki/Universally_unique_identifier) 的系统，你会发现这个系统可以轻松的实现这个需求！当你创建一个新的对象并插入到数据库的时候，会得到一个本地ID。这个本地ID和分片ID、类型ID组合便可得到一个UUID。

如果你曾用ALTER命令为MySQL数据表添加过字段，你肯定知道这种操作十分缓慢而且十分痛苦。本文的方法不需要任何级别的MYSQL ALTER操作。在Pinterest的过去三年中，我们可能完成了一次ALTER操作。给对象添加新的字段只需要让那些服务知道JSON模式中添加了几个新的字段。当反序列化一个JSON对象的时候如果没有某些字段，此时便可以返回预设的默认值。如果需要新的映射表，只需要创建一个，然后在需要的时候填充数据就可以。完成这些事情就可以发布新产品了。

取模分片（Mod Shard）

取模分片和 [***Mod Squad（译者注：一部电影）***](http://www.imdb.com/title/tt0062589/) 是一样的，唯一的区别是它们完全不同。

有一些对象并不是通过ID查询的。比如一个Pin友用Facebook账号登录的时候，我们需要将Facebook的ID映射到Pinterest的ID。Facebook的ID在我们看来就是比特位，所以我们将他们存储到一个独立的分片系统中，并命名为取模分片。包括其他诸如IP地址、用户名、邮箱地址等。

取模分片正如和上文中的分片系统类似，除了它们可以以任何数据查询。对查询输入取hash然后以系统中总分片数取模。结果就是数据将被存储或已经存储的位置，如下：

shard = md5(“1.2.3.4") % 4096

此例中 shard 值会被赋为1524。类似ID分片，需要维护一个配置文件，如下：

**[{“range”: (0, 511), “master”: “msdb001a”, “slave”: “msdb001b”},**

**{“range”: (512, 1023), “master”: “msdb002a”, “slave”: “msdb002b”},**

**{“range”: (1024, 1535), “master”: “msdb003a”, “slave”: “msdb003b”},**

**…]**

则，查询一个IP地址1.2.3.4的方式，如下：

conn = MySQLdb.connect(host=”msdb003a”)

conn.execute(“**SELECT** data **FROM** msdb001a.ip\_data **WHERE** ip='1.2.3.4'”)

# 备份

## Mysql 自动备份与恢复 - keenweiwei的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/keenweiwei/article/details/5903142

**自动备份MySql 5.0有三个方案：**

**备份方案一： 通过 mysqldump命令,直接生成一个完整的 .sql 文件**

Step 1: 创建一个批处理

(说明：root 是mysql默认用户名, aaaaaa 是mysql密码, bugtracker 是数据库名)

------------mySql\_backup.bat--------------------------------------------------------------------------------------

d:  
cd D:/AppServ/MySQL/bin  
mysqldump -uroot -paaaaaa bugtracker > E:/DataBase/Mysql\_bugtracker\_backup/bugtracker\_back.sql  
exit

------------------------------------------------------------------------------------------------------------------------

Step 2: 创建一个计划任务

"Start" -- > "Control Panel" --> "Administrative Tools" -- > "Task Scheduler"

"Create A Basic Task" --> --- > "Daily" (注意设置后面的 时 分 秒) --> ---- "Start a Program" --> "Browser" (定位选择到你刚才创建的批处理mySql\_backup.bat) --> --- > "Finish"

**对应的还原方法**

创建一个处理：

----------mySql\_restore.bat-----------------------------------------------------------------------------------------

d:  
cd D:/AppServ/MySQL/bin  
mysql -uroot -paaaaaa bugtracker < E:/DataBase/Mysql\_bugtracker\_backup/bugtracker\_back.sql  
exit

-------------------------------------------------------------------------------------------------------------------------

双击它即可自动执行还原

**备份方案二： 通过 xcopy 命令,直接把Mysql 的 data 目录下的对应的数据库全部的文件全部 copy 出来**

创建批处理：

------------------------------------------------------------------------------------------------------------------------------------------

xcopy D:/AppServ/MySQL/data/bugtracker E:/DataBase/Mysql\_bugtracker\_backup/bugtracker/ /e /h /d /y /r /v /f /k   
exit

-------------------------------------------------------------------------------------------------------------------------------------------

**对应的还原方法**

把 E:/DataBase/Mysql\_bugtracker\_backup 目录下的 bugtracker文件夹  直接 copy 到  D:/AppServ/MySQL/data 目录下,把这个目录的bugtracker文件夹 全部履盖掉

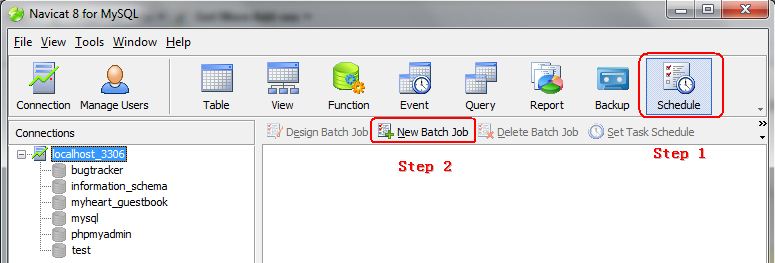
**备份方案三：安装Navicat For MySql 工具，通过它的 Schedule 功能设置自动备份**

Schedule -- > New Batch Job

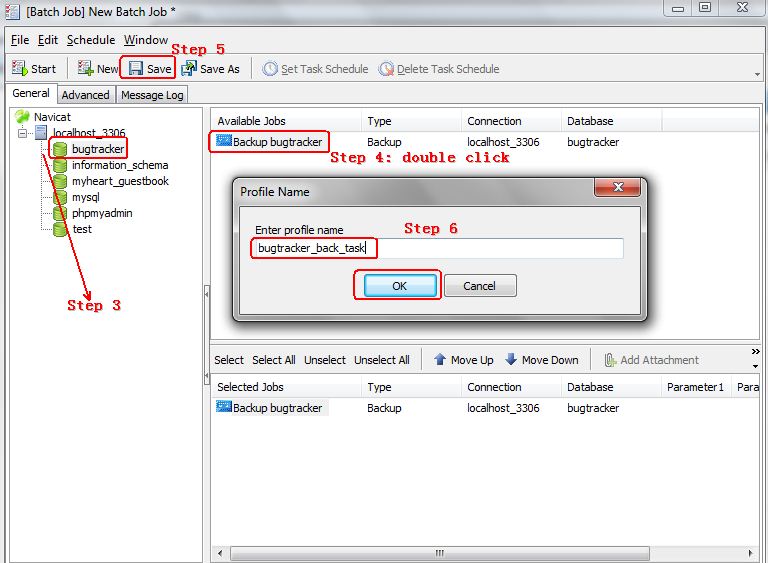
在新弹出的页面中 选择你的 DB （eg: bugtracker ），然后在右边面板(Available Jobs)中你会看到“Backup bugtracker”,双击它，它会出现在下边面板---

附上几张图片（全部的操作步骤）

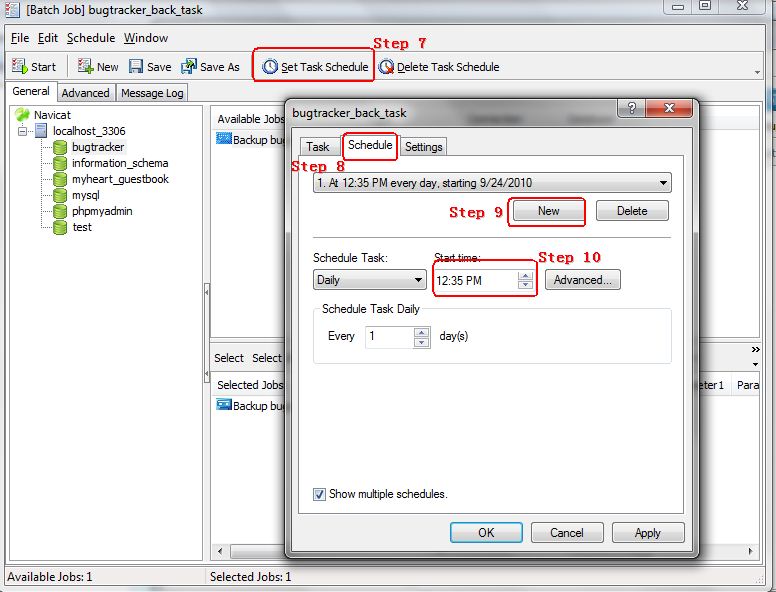
1.



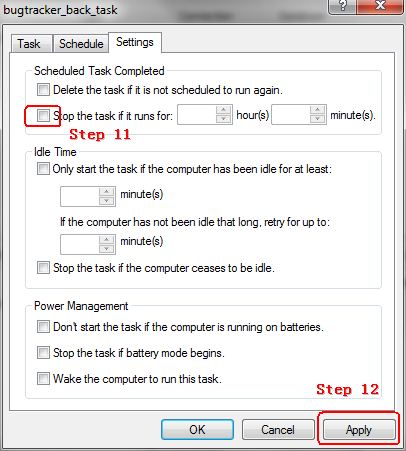
2.



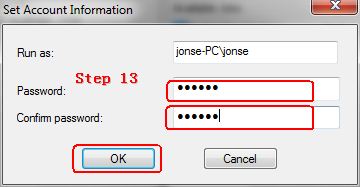
3.



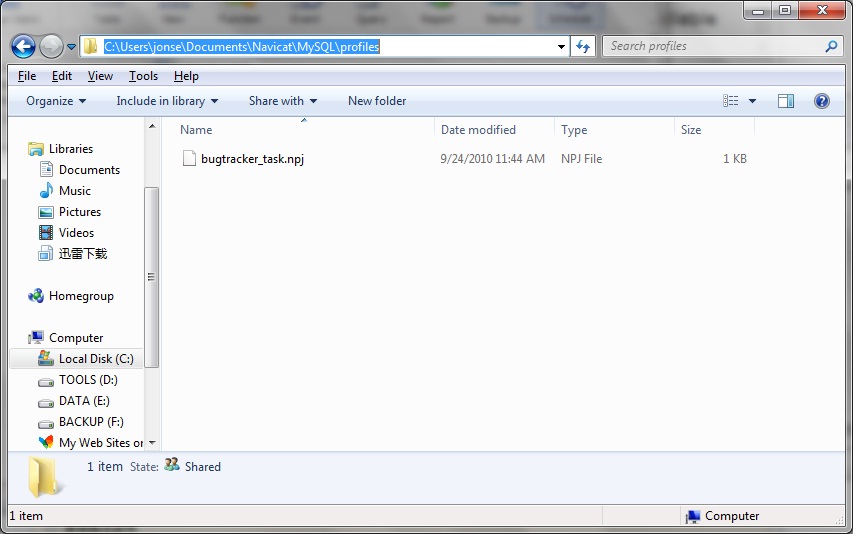
4.



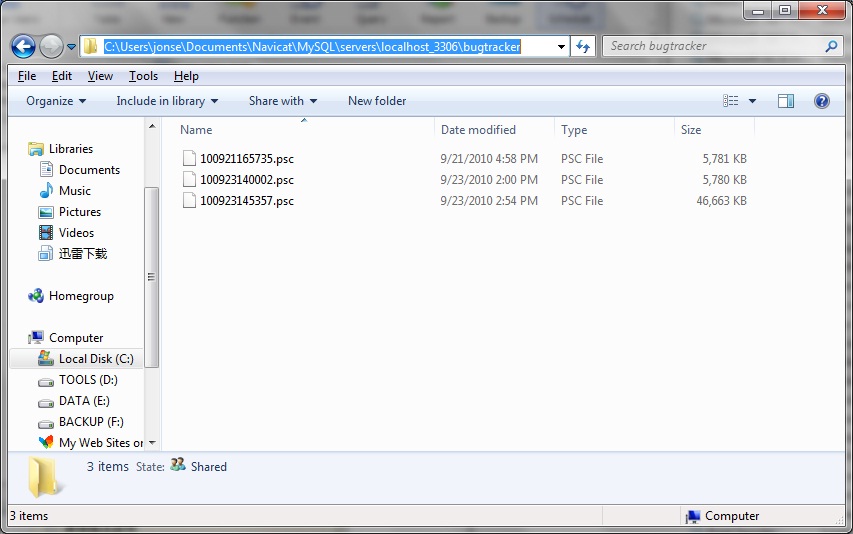
5.



6.



7.

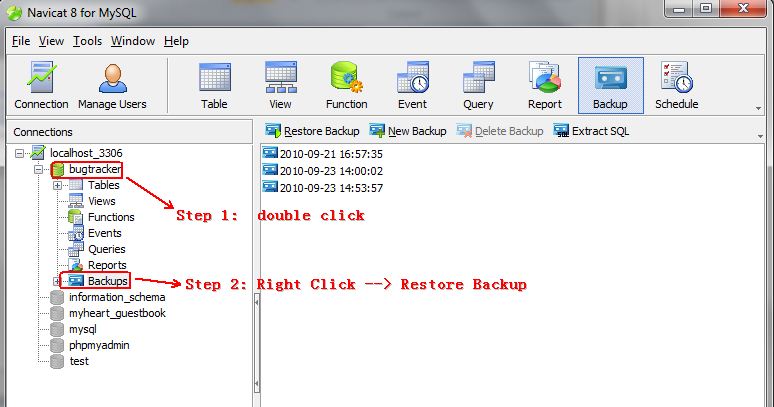


**对应的还原方法：**

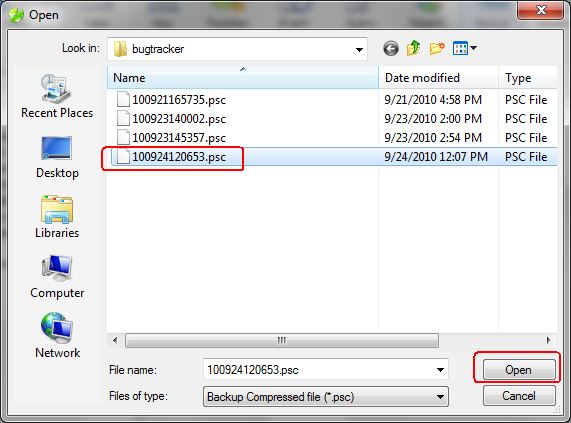
找到上述目录下对应的最新的 .psc 文件,然后通过Navicat For Mysql 工具还原

附上一些操作步骤的图片

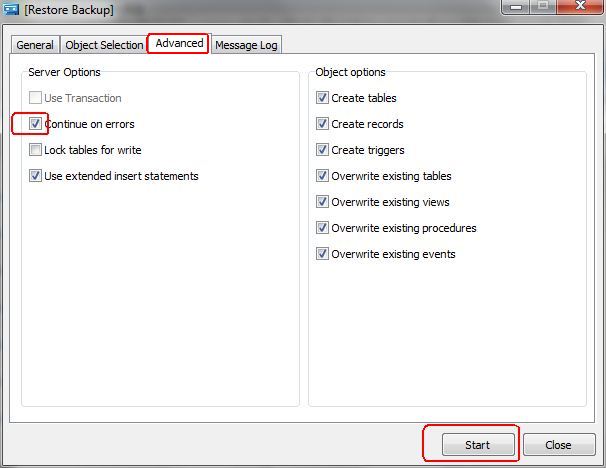
1.



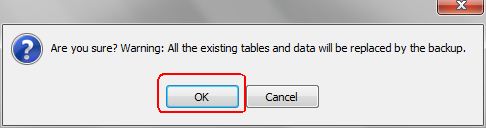
2.



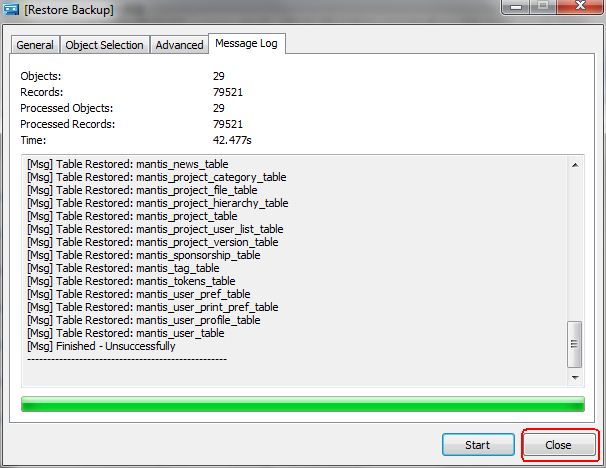
3.



4.



5.



希望这篇文间能对有需要的人有帮助！

## linux下的MySQL自动备份工具-第二版 - MySQL,数据库 - Tech - ITeye论坛

http://www.iteye.com/topic/1117136

第二版在第一版的基础上程序做了完善。主要包括自定义数据库主机和备份路径，支持备份多个数据库，无论多少个数据库，可以一次完成备份。可以控制是否开启过期备份删除，可以自定义过期时间等。并且增加了更加友好的提示。

备份工具用法：

1.将mysql.backup(或将代码保存为mysql.backup)上传至Linux服务器/usr/local/sbin目录下并赋予可执行权限。

2.用vi打开脚本配置数据库信息，主要配置以下信息即可

mysql\_user="root" #MySQL备份用户

mysql\_password="root" #MySQL备份用户的密码

mysql\_host="localhost" #要备份的主机，一般为localhost

mysql\_port="3306" #MySQL端口号，一般为3306

backup\_db\_arr=("dbachina" "99yuyu" "cowaw") #要备份的数据库名称，多个用空格分开隔开 如("db1" "db2" "db3")

backup\_location=/backup #备份数据存放位置，末尾请不要带"/",此项可以保持默认，程序会自动创建文件夹

expire\_backup\_delete="ON" #是否开启过期备份删除 ON为开启 OFF为关闭

expire\_days=3 #过期时间天数 默认为三天，此项只有在expire\_backup\_delete开启时有效

配置完成保存退出。

3.随时运行mysql.backup即可进行备份操作。

4.自动备份

利用Linux的crontab定时功能可以完成自动备份

在Linux下键入crontab -e

然后输入 30 02 \* \* \* /usr/local/sbin/mysql.backup 保存退出。

即可实现每天凌晨2点30分自动备份所有数据库。

下载地址：http://bbs.dba-china.com/thread-217-1-1.html

# 日志

#通过docker 映射到/data/mysql/data目录下

log\_error       =  /var/lib/mysql/error.log  
slow\_query\_log\_file = /var/lib/mysql/slow.log  
long\_query\_time = 2  
log\_bin         = /var/lib/mysql/mysql-bin

## Mysql 日志分析工具介绍 - 汪翰翔的Blog - 博客频道 - CSDN.NET

http://blog.csdn.net/babydavic/article/details/40152329

**1. 工具简介**

pt-query-digest是用于分析mysql慢查询的一个工具，它可以分析binlog、General log、slowlog，也可以通过SHOWPROCESSLIST或者通过tcpdump抓取的MySQL协议数据来进行分析。可以把分析结果输出到文件中，分析过程是先对查询语句的条件进行参数化，然后对参数化以后的查询进行分组统计，统计出各查询的执行时间、次数、占比等，可以借助分析结果找出问题进行优化。

pt-query-digest是一个perl脚本，只需下载并赋权即可执行。

[root@test1 ]# wget percona.com/get/pt-query-digest

[root@test1 ]# chmod u+x pt-query-digest

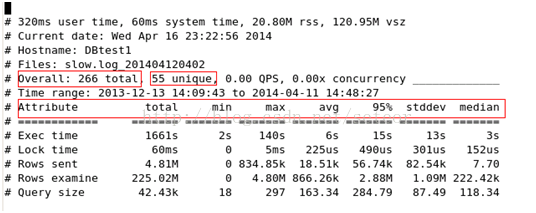
**2.语法及重要选项**

pt-query-digest [OPTIONS] [FILES] [DSN]

**--create-review-table**当使用--review参数把分析结果输出到表中时，如果没有表就自动创建。  
**--create-history-table**当使用--history参数把分析结果输出到表中时，如果没有表就自动创建。 **--filter**对输入的慢查询按指定的字符串进行匹配过滤后再进行分析  
**--limit**限制输出结果百分比或数量，默认值是20,即将最慢的20条语句输出，如果是50%则按总响应时间占比从大到小排序，输出到总和达到50%位置截止。  
**--host  [MySQL](http://lib.csdn.net/base/mysql" \t "_blank" \o "MySQL知识库)服务器地址  
--user**mysql用户名 **--password**mysql用户密码  
**--history** 将分析结果保存到表中，分析结果比较详细，下次再使用--history时，如果存在相同的语句，且查询所在的时间区间和历史表中的不同，则会记录到数据表中，可以通过查询同一CHECKSUM来比较某类型查询的历史变化。  
**--review** 将分析结果保存到表中，这个分析只是对查询条件进行参数化，一个类型的查询一条记录，比较简单。当下次使用--review时，如果存在相同的语句分析，就不会记录到数据表中。  
**--output** 分析结果输出类型，值可以是report(标准分析报告)、slowlog(Mysql slow log)、json、json-anon，一般使用report，以便于阅读。  
**--since** 从什么时间开始分析，值为字符串，可以是指定的某个”yyyy-mm-dd [hh:mm:ss]”格式的时间点，也可以是简单的一个时间值：s(秒)、h(小时)、m(分钟)、d(天)，如12h就表示从12小时前开始统计。  
**--until** 截止时间，配合—since可以分析一段时间内的慢查询。

**3.      标准分析报告解释**

**第一部分：总体统计结果,如下图**

****

Overall: 总共有多少条查询，上例为总共266个查询。

Time range: 查询执行的时间范围。

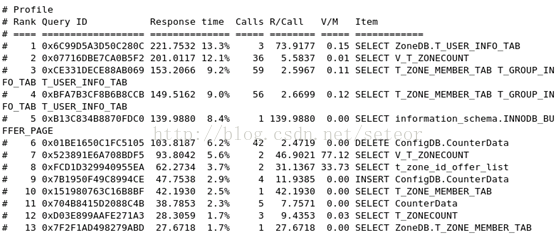
unique: 唯一查询数量，即对查询条件进行参数化以后，总共有多少个不同的查询，该例为55。

total: 总计   min:最小   max: 最大  avg:平均

95%: 把所有值从小到大排列，位置位于95%的那个数，这个数一般最具有参考价值。

median: 中位数，把所有值从小到大排列，位置位于中间那个数。

**第二部分：查询分组统计结果,如下图**



由上图可见，这部分对查询进行参数化并分组，然后对各类查询的执行情况进行分析，结果按总执行时长，从大到小排序。

Response: 总的响应时间。

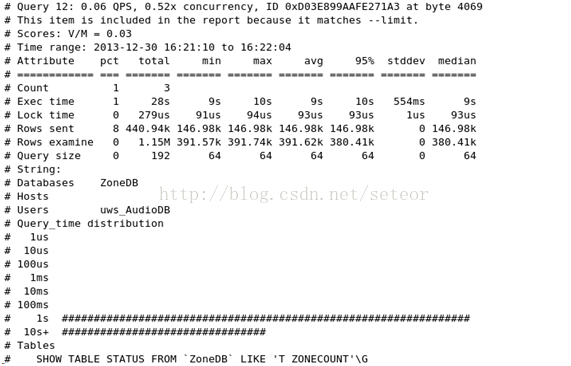
time: 该查询在本次分析中总的时间占比。

calls: 执行次数，即本次分析总共有多少条这种类型的查询语句。

R/Call: 平均每次执行的响应时间。

Item : 查询对象

**第三部分：每一种查询的详细统计结果，如下图：**



由上图可见，12号查询的详细统计结果，最上面的表格列出了执行次数、最大、最小、平均、95%等各项目的统计。

Databases: 库名

Users: 各个用户执行的次数（占比）

Query\_time distribution : 查询时间分布, 长短体现区间占比，本例中1s-10s之间查询数量是10s以上的两倍。

Tables: 查询中涉及到的表

Explain: 示例

**4.用法示例**

**(1)直接分析慢查询文件:**

pt-query-digest  slow.log > slow\_report.log

**(2)分析最近12小时内的查询：**

pt-query-digest  --since=12h  slow.log > slow\_report2.log

**(3)分析指定时间范围内的查询：**

pt-query-digest slow.log --since '2014-04-17 09:30:00' --until '2014-04-17 10:00:00'> > slow\_report3.log

**(4)分析指含有select语句的慢查询**  
pt-query-digest--filter '$event->{fingerprint} =~ m/^select/i' slow.log> slow\_report4.log  
  
**(5) 针对某个用户的慢查询**  
pt-query-digest--filter '($event->{user} || "") =~ m/^root/i' slow.log> slow\_report5.log  
  
**(6) 查询所有所有的全表扫描或full join的慢查询**  
pt-query-digest--filter '(($event->{Full\_scan} || "") eq "yes") ||(($event->{Full\_join} || "") eq "yes")' slow.log> slow\_report6.log  
 **(7)把查询保存到query\_review表**pt-query-digest  --user=root –password=abc123 --review  h=localhost,D=test,t=query\_review--create-review-table  slow.log

**(8)把查询保存到query\_history表**pt-query-digest  --user=root –password=abc123 --review  h=localhost,D=test,t=query\_ history--create-review-table  slow.log\_20140401pt-query-digest  --user=root –password=abc123--review  h=localhost,D=test,t=query\_history--create-review-table  slow.log\_20140402  
  
**(9)通过tcpdump抓取mysql的tcp协议数据，然后再分析**tcpdump -s 65535 -x -nn -q -tttt -i any -c 1000 port 3306 > mysql.tcp.txt  
pt-query-digest --type tcpdump mysql.tcp.txt> slow\_report9.log  
  
**(10)分析binlog**mysqlbinlog mysql-bin.000093 > mysql-bin000093.sql  
pt-query-digest  --type=binlog  mysql-bin000093.sql > slow\_report10.log  
  
**(11)分析general log**pt-query-digest  --type=genlog  localhost.log > slow\_report11.log

官方文档：http://www.percona.com/doc/percona-toolkit/2.2/pt-query-digest.html

## MySQL 大量日志的清理方法 - russle的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/russle/article/details/6928688

MySQL server运行有1年多时间，最近无法正常运行，启动也失败，后来检查了一下日志，发现没有空间写日志，只好找清理日志的方法，本来想直接删除那些日志，可是仔细一想，这些日志都是严格编号的，万一删除一个，导致mysql无法运行岂不误事。在网上search一下发现很多人都有同样的问题，现在把这些解决方便抄录过了，供大家参考。

我最后执行如下命令，就是只保留最近3天的日志，

mysql> PURGE MASTER LOGS BEFORE DATE\_SUB( NOW( ), INTERVAL 3 DAY);

还有个问题就是我吧innodb这个engine添加到我的mysql中了（

14.7. 插入存储引擎

能够使用存储引擎之前，必须使用INSTALL PLUGIN语句将存储引擎plugin（插件）装载到mysql。例如，要想加载example引擎，首先应加载ha\_example.so模块：

INSTALL PLUGIN ha\_example SONAME 'ha\_example.so';

文件.so必须位于MySQL服务器库目录下（典型情况下是installdir/lib）。

）

其他blog的解决办法

binlog就有40多G，原来根源出在这里，查看了一下my.cnf，看到binlog的size是1G就做分割，但没有看到删除的配置，在mysql里show了一下variables，

mysql> show variables like '%log%';

查到了

| expire\_logs\_days | 0 |

这个默认是0，也就是logs不过期，这个是一个global的参数，所以需要执行

set global expire\_logs\_days=8;

这样8天前的log就会被删除了，如果有回复的需要，请做好备份工作，但这样设置还不行，下次重启mysql了，配置又恢复默认了，所以需在my.cnf中设置

expire\_logs\_days = 8

这样重启也不怕了，

另外还有一种手动的删除binlog的方法，转自http://www.ourmysql.com/archives/293

想要恢愎数据库以前的资料,执行:show binlog events;

由于数据量很多，查看起来很麻烦，光打开个文件就要闪半天，所以应该适当删除部分可不用的日志。

并且如果使用的时间足够长的话，会把我的硬盘空间都给吃掉

1.登录系统，/usr/bin/mysql

使用mysql查看日志

mysql> show binary logs;

+—————-+———–+

| Log\_name | File\_size |

+—————-+———–+

| ablelee.000001 | 150462942 |

| ablelee.000002 | 120332942 |

| ablelee.000003 | 141462942 |

+—————-+———–+

2.删除bin-log(删除ablelee.000003之前的而没有包含ablelee.000003)

mysql> purge binary logs to ‘ablelee.000003′;

Query OK, 0 rows affected (0.16 sec)

3. 查询结果(现在只有一条记录了.)

mysql> show binlog events\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Log\_name: ablelee.000003

Pos: 4

Event\_type: Format\_desc

Server\_id: 1

End\_log\_pos: 106

Info: Server ver: 5.1.26-rc-log, Binlog ver: 4

1 row in set (0.01 sec)

(ablelee.000001和ablelee.000002已被删除)

mysql> show binary logs;

+—————-+———–+

| Log\_name | File\_size |

+—————-+———–+

| ablelee.000003 | 106 |

+—————-+———–+

1 row in set (0.00 sec)

(删除的其它格式运用!)

　 PURGE {MASTER | BINARY} LOGS TO ‘log\_name’

　　PURGE {MASTER | BINARY} LOGS BEFORE ‘date’

　　用于删除列于在指定的日志或日期之前的日志索引中的所有二进制日志。这些日志也会从记录在日志索引文件

中的清单中被删除，这样被给定的日志成为第一个。

　　例如：

　　PURGE MASTER LOGS TO ‘mysql-bin.010′;

　　PURGE MASTER LOGS BEFORE ‘2008-06-22 13:00:00′;

清除3天前的 binlog

PURGE MASTER LOGS BEFORE DATE\_SUB( NOW( ), INTERVAL 3 DAY);

　　BEFORE变量的date自变量可以为’YYYY-MM-DD hh:mm:ss’格式。MASTER和BINARY是同义词。

　　如果您有一个活性的从属服务器，该服务器当前正在读取您正在试图删除的日志之一，则本语句不会起作用，

而是会失败，并伴随一个错误。不过，如果从属服务器是休止的，并且您碰巧清理了其想要读取的日志之一，则从

属服务器启动后不能复制。当从属服务器正在复制时，本语句可以安全运行。您不需要停止它们。

　　要清理日志，需按照以下步骤：

　　1. 在每个从属服务器上，使用SHOW SLAVE STATUS来检查它正在读取哪个日志。

　　2. 使用SHOW MASTER LOGS获得主服务器上的一系列日志。

　　3. 在所有的从属服务器中判定最早的日志。这个是目标日志。如果所有的从属服务器是更新的，这是清单上的

最后一个日志。

　　4. 制作您将要删除的所有日志的备份。（这个步骤是自选的，但是建议采用。）

　　5. 清理所有的日志，但是不包括目标日志。

下面讲一下怎么从二进制文件恢复数据, 假如不小心执行了drop table xxx\_db, 假如你保留了完整的二进制日志的话, 先不要冒汗, 这是可以恢复的.

先看看日志

>mysqlbinlog /diskb/bin-logs/xxx\_db-bin.000001

找到执行create table xxx\_db之后和drop table xxx\_db之前的position, 假如是20, 1000.

>mysqlbinlog --start-position="4" --stop-position="1000" /diskb/bin-logs/xxx\_db-bin.000001 | mysql -u root

伴随着一大堆的ERROR 1062 (23000) at line 12355: Duplicate entry '139' for key 1, 数据库就这样恢复了, 不过--start-position="20"是不行的, 必须从--start-position="4"开始, 为什么要强制从4开始, 这个问题我也暂时没有搞清楚.

还有一种办法是根据日期来恢复

>mysqlbinlog --start-datetime="2009-09-14 0:20:00" --stop-datetim="2009-09-15 01:25:00" /diskb/bin-logs/xxx\_db-bin.000001 | mysql -u root

如果create table xxx\_db和drop table xxx\_db之间的时间相距是一年, 或者在不同的二进制日志中, 且位置相距好远, 就等着失眠吧! 做好备份, 小心操作才是正路啊...

如果MySQL服务器上有多个要执行的二进制日志，安全的方法是在一个连接中处理它们。下面是一个说明什么是不安全的例子：

shell> mysqlbinlog hostname-bin.000001 | mysql # DANGER!!shell> mysqlbinlog hostname-bin.000002 | mysql # DANGER!!使用与服务器的不同连接来处理二进制日志时，如果第1个日志文件包含一个CREATE TEMPORARY TABLE语句，第2个日志包含一个使用该临时表的语句，则会造成问题。当第1个mysql进程结束时，服务器撤销临时表。当第2个mysql进程想使用该表时，服务器报告 “不知道该表”。

要想避免此类问题，使用一个连接来执行想要处理的所有二进制日志中的内容。下面提供了一种方法：

shell> mysqlbinlog hostname-bin.000001 hostname-bin.000002 | mysql另一个方法是：

shell> mysqlbinlog hostname-bin.000001 > /tmp/statements.sqlshell> mysqlbinlog hostname-bin.000002 >> /tmp/statements.sqlshell> mysql -e "source /tmp/statements.sql"

## MySQL数据库恢复(使用mysqlbinlog命令)\_Mysql\_脚本之家

http://www.jb51.net/article/27919.htm

mysqlbinlog

1：开启binlog日志记录   
修改mysql配置文件mysql.ini，在[mysqld]节点下添加

复制代码代码如下:

# log-bin   
log-bin = E:/log/logbin.log

路径中不要包含中文和空格。重启mysql服务。通过命令行停止和启动mysql服务

复制代码代码如下:

c:\>net stop mysql;   
c:\>net start mysql;

进入命令行进入mysql并查看二进制日志是否已经启动   
Sql代码

复制代码代码如下:

mysql>show variables like 'log\_%';

日志成功开启后，会在E:/log/目录下创建logbin.index和logbin.000001两个文件。logbin.000001就是数据库的备份文件，以后就可以通过此文件对数据库进行恢复操作。   
  
2：查看备份的二进制文件   
Sql代码

复制代码代码如下:

c:\mysql\bin\>mysqlbinlog e:/log/logbin.000001

日后记录的操作多了，命令行方式基本就用不上了。可以使用将日志导出文件的方式来查看日志内容   
2.1 导出   
Xml代码

复制代码代码如下:

c:\mysql\bin\>mysqlbinlog e:/log/logbin.000001 > e:/log/log.txt

">": 导入到文件中; ">>": 追加到文件中   
如果有多个日志文件   
Sql代码

复制代码代码如下:

c:\mysql\bin\> mysqlbinlog e:/log/logbin.000001 > e:/log/log.sql   
c:\mysql\bin\> mysqlbinlog e:/log/logbin.000002 >> e:/log/log.sq

2.2 按指定位置导出：   
Sql代码

复制代码代码如下:

c:\mysql\bin\>mysqlbinlog --start-position=185 --stop-position=338 e:/log/logbin.000001 > e:/log/log3.txt

2.3 按指定时间导出：   
Xml代码

复制代码代码如下:

c:\mysql\bin\>mysqlbinlog --start-datetime="2010-01-07 11:25:56" --stop-datetime="2010-01-07 13:23:50" e:/log/logbin.000001 > e:/log/log\_by\_date22.txt

3：从备份恢复数据库   
做了一次更新操作，之后日志的内容如下：   
Sql代码

复制代码代码如下:

/\*!40019 SET @@session.max\_insert\_delayed\_threads=0\*/;   
/\*!50003 SET @OLD\_COMPLETION\_TYPE=@@COMPLETION\_TYPE,COMPLETION\_TYPE=0\*/;   
DELIMITER /\*!\*/;   
# at 4   
#110107 13:23:50 server id 1 end\_log\_pos 106 Start: binlog v 4, server v 5.1.53-community-log created 110107 13:23:50 at startup   
# Warning: this binlog is either in use or was not closed properly.   
ROLLBACK/\*!\*/;   
BINLOG '   
ZqMmTQ8BAAAAZgAAAGoAAAABAAQANS4xLjUzLWNvbW11bml0eS1sb2cAAAAAAAAAAAAAAAAAAAAA   
AAAAAAAAAAAAAAAAAABmoyZNEzgNAAgAEgAEBAQEEgAAUwAEGggAAAAICAgC   
'/\*!\*/;   
# at 106   
#110107 13:26:58 server id 1 end\_log\_pos 185 Query thread\_id=44 exec\_time=1 error\_code=0   
SET TIMESTAMP=1294378018/\*!\*/;   
SET @@session.pseudo\_thread\_id=44/\*!\*/;   
SET @@session.foreign\_key\_checks=1, @@session.sql\_auto\_is\_null=1, @@session.unique\_checks=1, @@session.autocommit=1/\*!\*/;   
SET @@session.sql\_mode=1344274432/\*!\*/;   
SET @@session.auto\_increment\_increment=1, @@session.auto\_increment\_offset=1/\*!\*/;   
/\*!\C utf8 \*//\*!\*/;   
SET @@session.character\_set\_client=33,@@session.collation\_connection=33,@@session.collation\_server=33/\*!\*/;   
SET @@session.lc\_time\_names=0/\*!\*/;   
SET @@session.collation\_database=DEFAULT/\*!\*/;   
BEGIN   
/\*!\*/;   
# at 185   
#110107 13:26:58 server id 1 end\_log\_pos 338 Query thread\_id=44 exec\_time=1 error\_code=0   
use ncl-interactive/\*!\*/;   
SET TIMESTAMP=1294378018/\*!\*/;   
UPDATE `t\_system\_id` SET `id\_value`='3000' WHERE (`table\_name`='t\_working\_day')   
/\*!\*/;   
# at 338   
#110107 13:26:58 server id 1 end\_log\_pos 365 Xid = 8016   
COMMIT/\*!\*/;   
DELIMITER ;   
DELIMITER /\*!\*/;   
DELIMITER ;   
# End of log file   
ROLLBACK /\* added by mysqlbinlog \*/;   
/\*!50003 SET COMPLETION\_TYPE=@OLD\_COMPLETION\_TYPE\*/;   
/\*!40019 SET @@session.max\_insert\_delayed\_threads=0\*/;   
/\*!50003 SET @OLD\_COMPLETION\_TYPE=@@COMPLETION\_TYPE,COMPLETION\_TYPE=0\*/;   
DELIMITER /\*!\*/;   
# at 4   
#110107 13:23:50 server id 1 end\_log\_pos 106 Start: binlog v 4, server v 5.1.53-community-log created 110107 13:23:50 at startup   
# Warning: this binlog is either in use or was not closed properly.   
ROLLBACK/\*!\*/;   
BINLOG '   
ZqMmTQ8BAAAAZgAAAGoAAAABAAQANS4xLjUzLWNvbW11bml0eS1sb2cAAAAAAAAAAAAAAAAAAAAA   
AAAAAAAAAAAAAAAAAABmoyZNEzgNAAgAEgAEBAQEEgAAUwAEGggAAAAICAgC   
'/\*!\*/;   
# at 106   
#110107 13:26:58 server id 1 end\_log\_pos 185 Query thread\_id=44 exec\_time=1 error\_code=0   
SET TIMESTAMP=1294378018/\*!\*/;   
SET @@session.pseudo\_thread\_id=44/\*!\*/;   
SET @@session.foreign\_key\_checks=1, @@session.sql\_auto\_is\_null=1, @@session.unique\_checks=1, @@session.autocommit=1/\*!\*/;   
SET @@session.sql\_mode=1344274432/\*!\*/;   
SET @@session.auto\_increment\_increment=1, @@session.auto\_increment\_offset=1/\*!\*/;   
/\*!\C utf8 \*//\*!\*/;   
SET @@session.character\_set\_client=33,@@session.collation\_connection=33,@@session.collation\_server=33/\*!\*/;   
SET @@session.lc\_time\_names=0/\*!\*/;   
SET @@session.collation\_database=DEFAULT/\*!\*/;   
BEGIN   
/\*!\*/;   
# at 185   
#110107 13:26:58 server id 1 end\_log\_pos 338 Query thread\_id=44 exec\_time=1 error\_code=0   
use ncl-interactive/\*!\*/;   
SET TIMESTAMP=1294378018/\*!\*/;   
UPDATE `t\_system\_id` SET `id\_value`='3000' WHERE (`table\_name`='t\_working\_day')   
/\*!\*/;   
# at 338   
#110107 13:26:58 server id 1 end\_log\_pos 365 Xid = 8016   
COMMIT/\*!\*/;   
DELIMITER ;   
DELIMITER /\*!\*/;   
DELIMITER ;   
# End of log file   
ROLLBACK /\* added by mysqlbinlog \*/;   
/\*!50003 SET COMPLETION\_TYPE=@OLD\_COMPLETION\_TYPE\*/;

3.1 恢复：   
Sql代码

复制代码代码如下:

c:\mysql\bin\>mysqlbinlog e:/log/logbin.000001 | mysql -u root -p

3.2 按指定位置恢复：   
Sql代码

复制代码代码如下:

c:\mysql\bin\>mysqlbinlog --start-position=185 --stop-position=338 e:/log/logbin.000001 | mysql -u root -p

3.3 按指定时间恢复：   
Xml代码

复制代码代码如下:

c:\mysql\bin\>mysqlbinlog --start-datetime="2010-01-07 11:25:56" --stop-datetime="2010-01-07 13:23:50" e:/log/logbin.000001 | mysql -u root -p

3.4 通过导出的脚本文件恢复   
Sql代码

复制代码代码如下:

c:\mysql\bin\>mysql -e "source e:/log/log.sql"

4.其他常用操作   
4.1 查看所有日志文件   
  
Sql代码

复制代码代码如下:

mysql>show master logs;

4.2 当前使用的binlog文件   
Sql代码

复制代码代码如下:

mysql>show binlog events \g;

4.3 产生一个新的binlog日志文件   
Sql代码

复制代码代码如下:

mysql>flush logs;

4.4 删除所有二进制日志，并从新开始记录（注意：reset master命令会删除所有的二进制日志）   
Sql代码

复制代码代码如下:

mysql > flush logs;   
mysql > reset master;

4.5 快速备份数据到sql文件   
  
Sql代码

复制代码代码如下:

c:\mysql\bin>mysqldump -u root -p --opt --quick interactive > e:/log/mysqldump.sql

为了方便查看，把从脚本恢复的命令在写一次   
Sql代码

复制代码代码如下:

c:\mysql\bin\>mysql -e "source e:/log/mysqldump.sql"

## MYSQL启用日志，和查看日志 - 明明 - 博客频道 - CSDN.NET

http://blog.csdn.net/a351945755/article/details/28239819

mysql有以下几种日志：    
   错误日志：     -log-err    
   查询日志：     -log    
   慢查询日志:   -log-slow-queries    
   更新日志:     -log-update    
   二进制日志： -log-bin    
  
  
是否启用了日志   
mysql>show variables like 'log\_%';   
  
怎样知道当前的日志   
mysql> show master status;   
  
顯示二進制日志數目   
mysql> show master logs;   
  
看二进制日志文件用mysqlbinlog   
shell>mysqlbinlog mail-bin.000001   
或者shell>mysqlbinlog mail-bin.000001 | tail   
  
在配置文件中指定log的輸出位置.   
Windows：Windows 的配置文件为 my.ini，一般在 MySQL 的安装目录下或者 c:\Windows 下。   
Linux：Linux 的配置文件为 my.cnf ，一般在 /etc 下。   
  
在linux下：

**Sql代码  收藏代码**

1. # 在[mysqld] 中輸入
2. #log
3. log-error=/usr/**local**/mysql/log/error.log
4. log=/usr/**local**/mysql/log/mysql.log
5. long\_query\_time=2
6. log-slow-queries= /usr/**local**/mysql/log/slowquery.log

windows下:

**Sql代码  收藏代码**

1. # 在[mysqld] 中輸入
2. #log
3. log-error="E:/PROGRA~1/EASYPH~1.0B1/mysql/logs/error.log"
4. log="E:/PROGRA~1/EASYPH~1.0B1/mysql/logs/mysql.log"
5. long\_query\_time=2
6. log-slow-queries= "E:/PROGRA~1/EASYPH~1.0B1/mysql/logs/slowquery.log"

开启慢查询   
long\_query\_time =2  --是指执行超过多久的sql会被log下来，这里是2秒   
log-slow-queries= /usr/local/mysql/log/slowquery.log  --将查询返回较慢的语句进行记录   
  
log-queries-not-using-indexes = nouseindex.log  --就是字面意思，log下来没有使用索引的query   
  
log=mylog.log  --对所有执行语句进行记录

## mysql日志详细解析 - KoMiles - 博客园

http://www.cnblogs.com/wangkongming/p/3684950.html

转自：http://pangge.blog.51cto.com/6013757/1319304

MySQL日志：

主要包含：错误日志、查询日志、慢查询日志、事务日志、二进制日志；

日志是mysql数据库的重要组成部分。日志文件中记录着mysql数据库运行期间发生的变化；也就是说用来记录mysql数据库的客户端连接状况、SQL语句的执行情况和错误信息等。当数据库遭到意外的损坏时，可以通过日志查看文件出错的原因，并且可以通过日志文件进行数据恢复。

错误日志

在mysql数据库中，错误日志功能是默认开启的。并且，错误日志无法被禁止。默认情况下，错误日志存储在mysql数据库的数据文件中。错误日志文件通常的名称为hostname.err。其中，hostname表示服务器主机名。

错误日志信息可以自己进行配置的，错误日志所记录的信息是可以通过log-error和log-warnings来定义的，其中log-err是定义是否启用错误日志的功能和错误日志的存储位置，log-warnings是定义是否将警告信息也定义至错误日志中。默认情况下错误日志大概记录以下几个方面的信息：服务器启动和关闭过程中的信息（未必是错误信息，如mysql如何启动InnoDB的表空间文件的、如何初始化自己的存储引擎的等等）、服务器运行过程中的错误信息、事件调度器运行一个事件时产生的信息、在从服务器上启动服务器进程时产生的信息。

下面我们来定义mysql错误日志的功能：

一般而言，日志级别的定义没有回话变量都只是在全局级别下进行定义。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | mysql> SHOW  GLOBAL VARIABLES LIKE '%log%';  +-----------------------------------------+-----------------------------------+  |  Variable\_name                            | Value                              |  +-----------------------------------------+-----------------------------------+  | back\_log                                | 50                                |  |  binlog\_cache\_size                        | 32768                              |  |  binlog\_direct\_non\_transactional\_updates | OFF                               |  |  binlog\_format                           | MIXED                             |  |  binlog\_stmt\_cache\_size                   | 32768                              |  |  expire\_logs\_days                         | 0                                  |  | general\_log                             | OFF                               |  |  general\_log\_file                         | /mydata/data/stu18.log             |  |  innodb\_flush\_log\_at\_trx\_commit           | 1                                  |  |  innodb\_locks\_unsafe\_for\_binlog           | OFF                               |  |  innodb\_log\_buffer\_size                   | 8388608                            |  |  innodb\_log\_file\_size                     | 5242880                            |  |  innodb\_log\_files\_in\_group                | 2                                  |  |  innodb\_log\_group\_home\_dir                | ./                                 |  |  innodb\_mirrored\_log\_groups               | 1                                  |  | log                                     |  OFF                               |  | log\_bin                                 | ON                                |  |  log\_bin\_trust\_function\_creators          | OFF                                |  | log\_error                               |  /mydata/data/stu18.magedu.com.err |定义错误日志  | log\_output                              | FILE                              |  |  log\_queries\_not\_using\_indexes            | OFF                                |  |  log\_slave\_updates                        | OFF                                |  |  log\_slow\_queries                        | OFF                               |  |  log\_warnings                             | 1               是否把警告信息写入错误日志中  |  max\_binlog\_cache\_size                    | 18446744073709547520               |  |  max\_binlog\_size                          | 1073741824                         |  |  max\_binlog\_stmt\_cache\_size               | 18446744073709547520               |  |  max\_relay\_log\_size                       | 0                                  |  | relay\_log                               |                                   |  |  relay\_log\_index                          |                                    |  |  relay\_log\_info\_file                      | relay-log.info                     |  |  relay\_log\_purge                          | ON                                 |  |  relay\_log\_recovery                       | OFF                                |  |  relay\_log\_space\_limit                    | 0                                  |  |  slow\_query\_log                           | OFF                                |  | slow\_query\_log\_file                     |  /mydata/data/stu18-slow.log       |  | sql\_log\_bin                             | ON                                |  | sql\_log\_off                             | OFF                               |  | sync\_binlog                             | 0                                 |  |  sync\_relay\_log                           | 0                                  |  |  sync\_relay\_log\_info                      | 0                                  |  +-----------------------------------------+-----------------------------------+ |

其中，log\_error可以直接定义为文件路径，也可以为ON|OFF；log\_warings只能使用1|0来定义开关启动。

更改错误日志位置可以使用log\_error来设置形式如下：

|  |  |
| --- | --- |
| 1  2  3 | [root@stu18 data]#  vim /etc/my.cnf  [mysqld]  Log\_error=DIR/[filename] |

解析：其中，DIR参数指定错误日志的路径filename参数是错误日志的名称，没有指定该参数时默认为主机名。重启mysql服务器即可生效。

查看mysql错误日志：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | [root@stu18 data]#  tail -20 stu18.magedu.com.err  130813  15:30:50  InnoDB: Starting shutdown...  130813  15:30:51  InnoDB: Shutdown completed;  log sequence number 1630920  130813 15:30:51  [Note] /usr/local/mysql/bin/mysqld: Shutdown complete  130813 15:30:52  mysqld\_safe mysqld from pid file /mydata/data/stu18.magedu.com.pid ended  130813 15:30:53  mysqld\_safe Starting mysqld daemon with databases from /mydata/data  130813 15:30:54  InnoDB: The InnoDB memory heap is disabled     #禁用了InnoDB memory的堆功能。  130813 15:30:54  InnoDB: Mutexes and rw\_locks use GCC atomic builtins #Mutexes（互斥量）和rw\_locks（行级锁）是GCC编译的是InnoDB内置的。  130813 15:30:54  InnoDB: Compressed tables use zlib 1.2.3     #默认压缩工具是zlib  130813 15:30:55  InnoDB: Initializing buffer pool, size = 128.0M    #InnoDB引擎的缓冲池（buffer pool）的值大小  130813 15:30:55  InnoDB: Completed initialization of buffer pool  130813 15:30:55  InnoDB: highest supported file format is Barracuda.  130813  15:30:57  InnoDB: Waiting for the  background threads to start  130813 15:30:58  InnoDB: 5.5.33 started; log sequence number 1630920  130813 15:30:58  [Note] Server hostname (bind-address): '0.0.0.0'; port: 3306  130813 15:30:58  [Note]   - '0.0.0.0' resolves to  '0.0.0.0';  #0.0.0.0会反解主机名，这里反解失败  130813 15:30:58  [Note] Server socket created on IP: '0.0.0.0'.  130813 15:30:58  [Note] Event Scheduler: Loaded 0 events    #事件调度器没有任何事件，因为没有装载。  130813 15:30:58  [Note] /usr/local/mysql/bin/mysqld: ready for connections. #mysql启动完成等待客户端的请求。  Version:  '5.5.33-log'  socket:  '/tmp/mysql.sock'  port: 3306  Source distribution  #创建一个本地sock用于本地连接。 |

删除错误日志：

在mysql5.5.7之前：数据库管理员可以删除很长时间之前的错误日志，以保证mysql服务器上的硬盘空间。mysql数据库中，可以使用mysqladmin命令开启新的错误日志。mysqladmin命令的语法如下：mysqladmin –u root –pflush-logs也可以使用登录mysql数据库中使用FLUSHLOGS语句来开启新的错误日志。

在mysql5.5.7之后：服务器将关闭此项功能。只能使用重命名原来的错误日志文件，手动冲洗日志创建一个新的：方式如下：

|  |  |
| --- | --- |
| 1  2  3  4 | [root@stu18 data]# mv stu18.magedu.com.err  stu18.magedu.com.err.old   [root@stu18 data]#  mysqladmin flush-logs  [root@stu18 data]# ls  hellodb  myclass  mysql-bin.000003  mysql-bin.index           stu18.magedu.com.pid     ibdata1      mysql     mysql-bin.000004  performance\_schema    ib\_logfile0  mysql-bin.000001  stu18.magedu.com.err           test   ib\_logfile1  mysql-bin.000002  stu18.magedu.com.err.old |

更多信息请查阅官方文档：<http://dev.mysql.com/doc/refman/5.5/en/error-log.html>

查询日志：

默认情况下查询日志是关闭的。由于查询日志会记录用户的所有操作，其中还包含增删查改等信息，在并发操作大的环境下会产生大量的信息从而导致不必要的磁盘IO，会影响mysql的性能的。如若不是为了调试数据库的目的建议不要开启查询日志。

查看查询日志是否开启：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | mysql> SHOW  GLOBAL VARIABLES LIKE '%log%';  +-----------------------------------------+-----------------------------------+  |  Variable\_name                            | Value                              |  +-----------------------------------------+-----------------------------------+  | back\_log                                | 50                                |  |  binlog\_cache\_size                        | 32768                              |  |  binlog\_direct\_non\_transactional\_updates | OFF                               |  |  binlog\_format                           | MIXED                             |  |  binlog\_stmt\_cache\_size                   | 32768                              |  |  expire\_logs\_days                         | 0                                  |  | general\_log                             | OFF        #定义查询日志是否开启  |  |  general\_log\_file                         | /mydata/data/stu18.log   #定义查询日志的文件地址名称                          |  |  innodb\_flush\_log\_at\_trx\_commit           | 1                                  |  |  innodb\_locks\_unsafe\_for\_binlog           | OFF                                |  |  innodb\_log\_buffer\_size                   | 8388608                            |  |  innodb\_log\_file\_size                     | 5242880                            |  |  innodb\_log\_files\_in\_group               | 2                                 |  |  innodb\_log\_group\_home\_dir                | ./                                 |  |  innodb\_mirrored\_log\_groups               | 1                                  |  | log                                     |  OFF          #是否开启日志  （若开启则表示开启所有的日志功能）                 |  | log\_bin                                 | ON                                |  |  log\_bin\_trust\_function\_creators          | OFF                                |  | log\_error                               | /mydata/data/stu18.magedu.com.err  |  | log\_output                              | FILE   #日志的输出的位置          |  |  log\_queries\_not\_using\_indexes            | OFF                                |  |  log\_slave\_updates                        | OFF                               |  |  log\_slow\_queries                         | OFF                                |  |  log\_warnings                             | 1                                  |  |  max\_binlog\_cache\_size                    | 18446744073709547520               |  |  max\_binlog\_size                          | 1073741824                         |  |  max\_binlog\_stmt\_cache\_size               | 18446744073709547520               |  |  max\_relay\_log\_size                       | 0                                  |  | relay\_log                               |                                   |  |  relay\_log\_index                          |                                    |  |  relay\_log\_info\_file                      | relay-log.info                     |  |  relay\_log\_purge                         | ON                                |  |  relay\_log\_recovery                       | OFF                                |  |  relay\_log\_space\_limit                    | 0                                  |  |  slow\_query\_log                           | OFF                                |  |  slow\_query\_log\_file                      | /mydata/data/stu18-slow.log        |  | sql\_log\_bin                             | ON                                |  | sql\_log\_off                             | OFF                               |  | sync\_binlog                             | 0                                 |  |  sync\_relay\_log                           | 0                                  |  |  sync\_relay\_log\_info                      | 0                                  |  +-----------------------------------------+-----------------------------------+  41 rows in set  (0.00 sec) |

拓展解析：日志的输出位置一般有三种方式：file(文件)，table(表)，none(不保存)；其中前两个输出位置可以同时定义，none表示是开启日志功能但是记录日志信息。file就是通过general\_log\_file |/mydata/data/stu18.log 等方式定义的，而输出位置定义为表时查看日志的内容方式为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | mysql> use  mysql;                    #在此数据库中  Database changed  mysql> show  tables;  +---------------------------+  |  Tables\_in\_mysql           |  +---------------------------+  |  columns\_priv              |  | db                        |  | event                     |  | func                      |  | general\_log               |      #这个就是查询日志的表输出位置  |  help\_category             |  |  help\_keyword              |  |  help\_relation             |  | help\_topic  ………………                |  +---------------------------+ |

慢查询日志：

慢查询日志是用来记录执行时间超过指定时间的查询语句。通过慢查询日志，可以查找出哪些查询语句的执行效率很低，以便进行优化。一般建议开启，它对服务器性能的影响微乎其微，但是可以记录mysql服务器上执行了很长时间的查询语句。可以帮助我们定位性能问题的。

查看慢查询日志的定义：

|  |  |
| --- | --- |
| 1  2  3 | mysql> SHOW  GLOBAL VARIABLES LIKE '%log%';  |  slow\_query\_log         | OFF    #定义慢查询日志的  |  slow\_query\_log\_file    |/mydata/data/stu18-slow.log    #输出方式为file（文件）时定义慢查询日志的位置 |

启动和设置慢查询日志：

1、通过配置文件my.cnf中的log-slow-queries选项可以开启慢查询日志；形式如下：

|  |  |
| --- | --- |
| 1  2  3  4 | [root@stu18 data]#  vim /etc/my.cnf  [mysqld]  slow\_query\_log=1  log-slow-queries [=  DIR/[filename] ] |

其中，DIR参数指定慢查询日志的存储路径；filename参数指定日志的文件名，生成日志文件的完成名称为filename-slow.log。如果不指定存储路径，慢查询日志默认存储到mysql数据库的数据文件下，如果不指定文件名，默认文件名为hostname-slow.log。

2、通过登录mysql服务器直接定义，方式如下：

首先要有全局权限；然后执行mysql>set global slow\_query\_log=1;

时间默认超过多少的称为慢查询日志？

一般都是通过long\_query\_time选项来设置这个时间值，时间以秒为单位，可以精确到微秒。如果查询时间超过了这个时间值（默认为10秒），这个查询语句将被记录到慢查询日志中。查看服务器默认时间值方式如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | mysql> SHOW  GLOBAL VARIABLES LIKE 'long%';  +-----------------+-----------+  |  Variable\_name   | Value     |  +-----------------+-----------+  | long\_query\_time  | 10.000000 |  +-----------------+-----------+  1 row in set (0.04  sec) |

注释：其中这个慢查询时间并不是只表示语句自身执行超过10秒还包含由于其他资源被征用造成阻塞的查询执行时间或其他原因等都被记录到慢查询中。所以这个慢查的时长表示从查询开始到查询结束中间包含可能的任何原因所经历的所有时间。

测试是否可以记录日志：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | mysql> set globalslow\_query\_log=1;       #开启慢查询日志  Query OK, 0 rowsaffected (0.35 sec)  mysql> setsession long\_query\_time=0.001;     #更改时间（当前session中，退出则重置）  Query OK, 0 rowsaffected (0.00 sec)  mysql> set globallong\_query\_time=0.001;      #更改时间（全局中，重启服务则重置）  mysql> SHOWVARIABLES LIKE 'long%';          #查询定义时间  +-----------------+----------+  | Variable\_name   | Value   |  +-----------------+----------+  | long\_query\_time |0.001000 |  +-----------------+----------+  1 row in set (0.00sec)  mysql> showglobal variables like "%slow%";  #查看慢查询日志开启状态  +---------------------+-----------------------------+  | Variable\_name       | Value                       |  +---------------------+-----------------------------+  |log\_slow\_queries    | ON                          |  |slow\_launch\_time    | 2                           |  |slow\_query\_log      | ON                          |  |slow\_query\_log\_file | /mydata/data/stu18-slow.log |  +---------------------+-----------------------------+  4 rows in set (0.03sec) |

查看慢查询日志：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | mysql> use mysql  mysql> selectuser,host,password from user where user="root";  +------+------------------+----------+  | user | host             | password |  +------+------------------+----------+  | root |localhost        |          |  | root |stu18.magedu.com |          |  | root |127.0.0.1        |          |  | root | ::1              |          |  +------+------------------+----------+  4 rows in set (0.08sec)       #查询时间为0.08  mysql> systemmore /mydata/data/stu18\_slow.log          #查询慢查询日志记录信息  /usr/local/mysql/bin/mysqld,Version: 5.5.33-log (Source distribution). started  with:  Tcp port: 3306  Unix socket: /tmp/mysql.sock  Time                 Id Command    Argument  >>>>>>>>>>>>>>>>部分已省略>>>>>>>>>>>>>>  # Time: 13100723:46:33  # User@Host:root[root] @ localhost []  # Query\_time:0.108459  Lock\_time: 0.000216 Rows\_sent:4  Rows\_examined: 6  SETtimestamp=1381160793;  selectuser,host,password from user where user="root"; |

事务日志：

事务日志（InnoDB特有的日志）可以帮助提高事务的效率。使用事务日志，存储引擎在修改表的数据时只需要修改其内存拷贝，再把改修改行为记录到持久在硬盘上的事务日志中，而不用每次都将修改的数据本身持久到磁盘。事务日志采用追加的方式，因此写日志的操作是磁盘上一小块区域内的顺序I/O，而不像随机I/O需要在磁盘的多个地方移动磁头，所以采用事务日志的方式相对来说要快得多。事务日志持久以后，内存中被修改的数据在后台可以慢慢的刷回到磁盘。目前大多数的存储引擎都是这样实现的，我们通常称之为预写式日志，修改数据需要写两次磁盘。

如果数据的修改已经记录到事务日志并持久化，但数据本身还没有写回磁盘，此时系统崩溃，存储引擎在重启时能够自动恢复这部分修改的数据。具有的恢复方式则视存储引擎而定。

查看事务日志的定义：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | mysql> SHOWGLOBAL VARIABLES LIKE '%log%';  +-----------------------------------------+-----------------------------------+  | Variable\_name                           | Value                             |  +-----------------------------------------+-----------------------------------+  | innodb\_flush\_log\_at\_trx\_commit          | 1        #在事务提交时innodb是否同步日志从缓冲到文件中1表示事务以提交就同步不提交每隔一秒同步一次，性能会很差造成大量的磁盘I/O；定义为2表示只有在事务提交时才会同步但是可能会丢失整个事务      |  |innodb\_locks\_unsafe\_for\_binlog          |OFF                               |  | innodb\_log\_buffer\_size                  | 8388608                           |  |innodb\_log\_file\_size                    |5242880                           |  | innodb\_log\_files\_in\_group               | 2      #至少有两个                |  |innodb\_log\_group\_home\_dir               |./      #定义innodb事务日志组的位置       |  |innodb\_mirrored\_log\_groups              |1    #表示对日志组做镜像          |  每个事务日志都是大小为5兆的文件：  [root@stu18 data]#ls -lh  -rw-rw---- 1 mysqlmysql 5.0M Oct  7 23:36 ib\_logfile0  -rw-rw---- 1 mysqlmysql 5.0M Aug 12 01:06 ib\_logfile1 |

二进制日志：

二进制日志也叫作变更日志，主要用于记录修改数据或有可能引起数据改变的mysql语句，并且记录了语句发生时间、执行时长、操作的数据等等。所以说通过二进制日志可以查询mysql数据库中进行了哪些变化。一般大小体积上限为1G。

二进制开启状态：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | mysql> showglobal variables like "%log\_bin%";  +---------------------------------+-------+  | Variable\_name                   | Value |  +---------------------------------+-------+  | log\_bin                         | ON    |  #已开启  |log\_bin\_trust\_function\_creators | OFF   |  | sql\_log\_bin                     | ON    |  +---------------------------------+-------+ |

二进制日志相关的参数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | mysql> showglobal variables like "%log%";  sql\_log\_bin ={ON|OFF}     #用于控制会话级别二进制日志功能的开启或关闭。默认为ON，表示启用记录功能。用户可以在会话级别修改此变量的值，但其必须具有SUPER权限。  binlog\_cache\_size =32768   #默认值32768 Binlog Cache用于在打开了二进制日志（binlog）记录功能的环境，是MySQL 用来提高binlog的记录效率而设计的一个用于短时间内临时缓存binlog数据的内存区域。一般来说，如果我们的数据库中没有什么大事务，写入也不是特别频繁，2MB～4MB是一个合适的选择。但是如果我们的数据库大事务较多，写入量比较大，可与适当调高binlog\_cache\_size。同时，我们可以通过binlog\_cache\_use 以及 binlog\_cache\_disk\_use来分析设置的binlog\_cache\_size是否足够，是否有大量的binlog\_cache由于内存大小不够而使用临时文件（binlog\_cache\_disk\_use）来缓存了。  binlog\_stmt\_cache\_size= 32768       #当非事务语句使用二进制日志缓存，但是超出binlog\_stmt\_cache\_size时，使用一个临时文件来存放这些语句。  log\_bin = mysql-bin#指定binlog的位置，默认在数据目录下。  binlog-format= {ROW|STATEMENT|MIXED}     #指定二进制日志的类型，默认为MIXED。如果设定了二进制日志的格式，却没有启用二进制日志，则MySQL启动时会产生警告日志信息并记录于错误日志中。  sync\_binlog = 10#设定多久同步一次二进制日志至磁盘文件中，0表示不同步，任何正数值都表示对二进制每多少次写操作之后同步一次。当autocommit的值为1时，每条语句的执行都会引起二进制日志同步，否则，每个事务的提交会引起二进制日志同步  max\_binlog\_cache\_size= {4096 .. 18446744073709547520}      #二进定日志缓存空间大小，5.5.9及以后的版本仅应用于事务缓存，其上限由max\_binlog\_stmt\_cache\_size决定。  max\_binlog\_stmt\_cache\_size= {4096 .. 18446744073709547520}    #二进定日志缓存空间大小，5.5.9及以后的版本仅应用于事务缓存  expire\_log\_days ={0..99}    #设定二进制日志的过期天数，超出此天数的二进制日志文件将被自动删除。默认为0，表示不启用过期自动删除功能。如果启用此功能，自动删除工作通常发生在MySQL启动时或FLUSH日志时。 |

二进制日志定义方式：

其一、log\_bin可以直接定义为文件路径，也可以为ON|OFF。

其二、通过编辑my.cnf中的log-bin选项可以开启二进制日志；形式如下：

|  |  |
| --- | --- |
| 1  2  3 | [root@stu18 ~]#my.cnf  [mysqld]  log-bin [=DIR \ [filename]] |

其中，DIR参数指定二进制文件的存储路径；filename参数指定二级制文件的文件名，其形式为filename.number，number的形式为000001、000002等。每次重启mysql服务或运行mysql> flush logs;都会生成一个新的二进制日志文件，这些日志文件的number会不断地递增。除了生成上述的文件外还会生成一个名为filename.index的文件。这个文件中存储所有二进制日志文件的清单又称为二进制文件的索引。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | [root@stu18 ~]# cd /mydata/data/  [root@stu18 data]#ls -lh  -rw-rw---- 1 mysqlmysql  14K Aug 13 15:30 mysql-bin.000001  -rw-rw---- 1 mysqlmysql  150 Aug 13 17:05 mysql-bin.000002  -rw-rw---- 1 mysqlmysql  150 Aug 13 17:06 mysql-bin.000003  -rw-rw---- 1 mysqlmysql  150 Aug 13 17:07 mysql-bin.000004  -rw-rw---- 1 mysqlmysql  150 Aug 13 17:39 mysql-bin.000005  -rw-rw---- 1 mysqlmysql  126 Aug 13 19:03 mysql-bin.000006  -rw-rw---- 1 mysqlmysql  126 Aug 13 19:03 mysql-bin.000007  -rw-rw---- 1 mysqlmysql  126 Aug 13 19:05 mysql-bin.000008  -rw-rw---- 1 mysqlmysql  107 Aug 13 19:05 mysql-bin.000009  -rw-rw---- 1 mysqlmysql  353 Oct  7 23:40 mysql-bin.000010  -rw-rw---- 1 mysqlmysql  190 Oct  7 20:43 mysql-bin.index  [root@stu18 data]#cat mysql-bin.index  ./mysql-bin.000001  ./mysql-bin.000002  ./mysql-bin.000003  ./mysql-bin.000004  ./mysql-bin.000005  ./mysql-bin.000006  ./mysql-bin.000007  ./mysql-bin.000008  ./mysql-bin.000009  ./mysql-bin.000010 |

如果说我们向某个表的某个字段插入一个数据而这个数据为当前时间(日期时间型)；过段时间将此二进制文件应用到另一台服务器上数据就会变动从而导致数据的不一致性所以说对于这种非确定性的数据使用默认的语句定义并不是可靠的；

二进制日志中常用的定义格式：

1、语句(statement)：默认的记录格式；

2、行(row)：定义的并非数据本身而是这一行的数据是什么；

3、混合模式(mixed)：交替使用行和语句、由mysql服务器自行判断。

其中基于行的定义格式数据量会大一些但是可以保证数据的精确性。

查看二进制日志：

二进制日志的定义方式为二进制格式；使用此格式可以存储更多的信息，并且可以使写入二进制日志的效率更高。但是不能直接使用查看命令打开并查看二进制日志。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | mysql> showbinary logs;     #显示当前服务器使用的二进制文件及大小  +------------------+-----------+  | Log\_name         | File\_size |  +------------------+-----------+  | mysql-bin.000001|     13814 |  | mysql-bin.000002|       150 |  | mysql-bin.000003|       150 |  | mysql-bin.000004|       150 |  | mysql-bin.000005|       150 |  | mysql-bin.000006|       126 |  | mysql-bin.000007|       126 |  | mysql-bin.000008|       126 |  | mysql-bin.000009|       107 |  | mysql-bin.000010|       353 |  +------------------+-----------+  10 rows in set (0.07sec)  mysql> showmaster logs;      #显示主服务器使用的二进制文件及大小  +------------------+-----------+  | Log\_name         | File\_size |  +------------------+-----------+  | mysql-bin.000001|     13814 |  | mysql-bin.000002|       150 |  | mysql-bin.000003|       150 |  | mysql-bin.000004|       150 |  | mysql-bin.000005|       150 |  | mysql-bin.000006|       126 |  | mysql-bin.000007|       126 |  | mysql-bin.000008|       126 |  | mysql-bin.000009|       107 |  | mysql-bin.000010|       353 |  +------------------+-----------+  10 rows in set (0.02sec)  mysql> showmaster status;   #当前使用的二进制文件及所处位置  +------------------+----------+--------------+------------------+  | File             | Position | Binlog\_Do\_DB |Binlog\_Ignore\_DB |  +------------------+----------+--------------+------------------+  | mysql-bin.000010|      353 |              |                  |  +------------------+----------+--------------+------------------+  1 row in set (0.00sec) |

小扩展：二进制日志的记录位置：通常为上一个事件执行结束时间的位置，每一个日志文件本身也有自己的元数据所以说对于当前版本的mysql来说二进制的开始位置通常为107；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | mysql> flushlogs;  Query OK, 0 rowsaffected (0.23 sec)  注意：flush logs一般只会滚动中继日志和二进制日志。  mysql> showmaster status;  +------------------+----------+--------------+------------------+  | File             | Position | Binlog\_Do\_DB |Binlog\_Ignore\_DB |  +------------------+----------+--------------+------------------+  | mysql-bin.000011|      107 |              |                  |  +------------------+----------+--------------+------------------+  1 row in set (0.00sec) |

查看当前二进制文件的信息：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | mysql> createdatabase yong;  Query OK, 1 rowaffected (0.12 sec)  mysql> createtable yong.tb1 (id int,name char(20));  Query OK, 0 rowsaffected (0.44 sec)  mysql> insertinto yong.tb1 values(1,'tom');  Query OK, 1 rowaffected (0.14 sec)  mysql> showmaster status;  +------------------+----------+--------------+------------------+  | File             | Position | Binlog\_Do\_DB |Binlog\_Ignore\_DB |  +------------------+----------+--------------+------------------+  | mysql-bin.000011|      479 |              |                  |  +------------------+----------+--------------+------------------+  1 row in set (0.00sec) |

查看二进制日志信息的命令：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | SHOW BINLOG EVENTS[IN 'log\_name'] [FROM pos] [LIMIT [offset,] row\_count]  mysql> showbinlog events\G         #查看所有的二进制信息  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*87. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*     Log\_name: mysql-bin.000001          Pos: 13580   Event\_type: Query    Server\_id: 1  End\_log\_pos: 13688         Info: use `hellodb`; /\*!40000 ALTERTABLE `toc` DISABLE KEYS \*/  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*88. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*     Log\_name: mysql-bin.000001          Pos: 13688   Event\_type: Query    Server\_id: 1  End\_log\_pos: 13795         Info: use `hellodb`; /\*!40000 ALTERTABLE `toc` ENABLE KEYS \*/  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*89. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*     Log\_name: mysql-bin.000001          Pos: 13795   Event\_type: Stop    Server\_id: 1  End\_log\_pos: 13814         Info:  89 rows in set (0.00sec)  mysql> showbinlog events in 'mysql-bin.000011';    #查看指定日志的二进制信息  +------------------+-----+-------------+-----------+-------------+----------------------------------------------+  | Log\_name         | Pos | Event\_type  | Server\_id | End\_log\_pos | Info                                         |  +------------------+-----+-------------+-----------+-------------+----------------------------------------------+  | mysql-bin.000011|   4 | Format\_desc |         1 |         107 | Server ver: 5.5.33-log, Binlogver: 4        |  | mysql-bin.000011 |107 | Query       |         1 |         190 | create database yong                         |  | mysql-bin.000011 |190 | Query       |         1 |         293 | create table yong.tb1 (idint,name char(20)) |  | mysql-bin.000011 |293 | Query       |         1 |         357 | BEGIN                                        |  | mysql-bin.000011 |357 | Query       |         1 |         452 | insert into yong.tb1values(1,'tom')         |  | mysql-bin.000011 |452 | Xid         |         1 |         479 | COMMIT /\* xid=103 \*/                         |  +------------------+-----+-------------+-----------+-------------+----------------------------------------------+  6 rows in set (0.00sec)  mysql> showbinlog events in 'mysql-bin.000011' from 190; #从指定的事件位置开始  +------------------+-----+------------+-----------+-------------+----------------------------------------------+  | Log\_name         | Pos | Event\_type | Server\_id |End\_log\_pos | Info                                         |  +------------------+-----+------------+-----------+-------------+----------------------------------------------+  | mysql-bin.000011 |190 | Query      |         1 |         293 | create table yong.tb1 (idint,name char(20)) |  | mysql-bin.000011 |293 | Query      |         1 |         357 | BEGIN                                        |  | mysql-bin.000011 |357 | Query      |         1 |         452 | insert into yong.tb1values(1,'tom')         |  | mysql-bin.000011 |452 | Xid        |         1 |         479 | COMMIT /\* xid=103 \*/                         |  +------------------+-----+------------+-----------+-------------+----------------------------------------------+  4 rows in set (0.00sec)  mysql> showbinlog events in 'mysql-bin.000011' from 190 limit 3;  #指定偏移量(不是语句，是事件)  +------------------+-----+------------+-----------+-------------+----------------------------------------------+  | Log\_name         | Pos | Event\_type | Server\_id |End\_log\_pos | Info                                         |  +------------------+-----+------------+-----------+-------------+----------------------------------------------+  | mysql-bin.000011 |190 | Query      |         1 |         293 | create table yong.tb1 (idint,name char(20)) |  | mysql-bin.000011 |293 | Query      |         1 |         357 | BEGIN                                        |  | mysql-bin.000011 |357 | Query      |         1 |         452 | insert into yong.tb1values(1,'tom')         |  +------------------+-----+------------+-----------+-------------+----------------------------------------------+  3 rows in set (0.00sec) |

命令行下查看二进制日志：

由于无法使用cat等方式直接打开并查看二进制日志；所以必须使用mysqlbinlog命令。但是当正在执行mysql读写操作时建议不要使用此打开正在使用的二进制日志文件；若非要打开可flush logs。mysqlbinlog命令的使用方式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | [root@stu18 data]#mysqlbinlog mysql-bin.000017        #必须在数据目录下  /\*!50530 SET @@SESSION.PSEUDO\_SLAVE\_MODE=1\*/;  /\*!40019 SET@@session.max\_insert\_delayed\_threads=0\*/;  /\*!50003 SET@OLD\_COMPLETION\_TYPE=@@COMPLETION\_TYPE,COMPLETION\_TYPE=0\*/;  DELIMITER /\*!\*/;  # at 4       #事件开始处  #131009  0:25:59 server id 1  end\_log\_pos 107   Start: binlog v 4, server v 5.5.33-log created 131009  0:25:59  # Warning: thisbinlog is either in use or was not closed properly.  BINLOG '  FzJUUg8BAAAAZwAAAGsAAAABAAQANS41LjMzLWxvZwAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAEzgNAAgAEgAEBAQEEgAAVAAEGggAAAAICAgCAA==  '/\*!\*/;  # at 107  #131009  0:26:36 server id 1  end\_log\_pos 192   Query   thread\_id=12    exec\_time=0 error\_code=0     #131009 0:26:36年月日的简写方式；end\_log\_pos事件结束处；thread\_id=12 哪个会话线程创建的此语句；exec\_time=0 执行时长单位为秒；error\_code=0 错误代码0表示没有  SET TIMESTAMP=1381249596/\*!\*/;      #预设信息(环境设定) |

导出此数据库的信息：

|  |  |
| --- | --- |
| 1 | [root@stu18 data]#mysqlbinlog mysql-bin.000017 > /tmp/a.sql |

导入此数据库的信息：

|  |  |
| --- | --- |
| 1 | [root@stu18 data]#mysql < a.sql |

删除二进制日志信息：

二进制日志会记录大量的信息（其中包含一些无用的信息）。如果很长时间不清理二进制日志，将会浪费很多的磁盘空间。但是，删除之后可能导致数据库崩溃时无法进行恢复，所以若要删除二进制日志首先将其和数据库备份一份，其中也只能删除备份前的二进制日志，新产生的日志信息不可删(可以做即时点还原)。也不可在关闭mysql服务器之后直接删除因为这样可能会给数据库带来错误的。若非要删除二进制日志需要做如下操作：导出备份数据库和二进制日志文件进行压缩归档存储。删除二进制文件的方法如下：

1、删除所有的二进制日志（不可效仿）：

使用RESET MASTER语句可以删除所有的二进制日志。该语句的形式如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | mysql> resetmaster;  Query OK, 0 rowsaffected (0.17 sec)  mysql> showbinary logs;  +------------------+-----------+  | Log\_name         | File\_size |  +------------------+-----------+  | mysql-bin.000001|       107 |  +------------------+-----------+  1 row in set (0.04sec) |

解析：首先不建议在生产环境下使用此操作；删除所有的二进制日志后，Mysql将会重新创建新的二进制日志。新二进制日志的编号从000001开始。

2、根据文件或时间点来删除二进制日志：

语法形式：

|  |  |
| --- | --- |
| 1 | mysql> PURGE { BINARY | MASTER } LOGS {TO 'log\_name' | BEFORE datetime\_expr } |

其中TO'log\_name'表示把这个文件之前的其他文件都删除掉，也可使用BEFORE datetime\_expr指定把哪个时间之前的二进制文件删除了。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | mysql> PURGEBINARY LOGS TO 'mysql-bin.000007';  Query OK, 0 rowsaffected (0.11 sec)  mysql> showbinary logs;  +------------------+-----------+  | Log\_name         | File\_size |  +------------------+-----------+  | mysql-bin.000007|       150 |  | mysql-bin.000008|       150 |  | mysql-bin.000009|       150 |  | mysql-bin.000010|       150 |  | mysql-bin.000011|       150 |  | mysql-bin.000012|       150 |  | mysql-bin.000013|       150 |  | mysql-bin.000014|       150 |  | mysql-bin.000015|       150 |  | mysql-bin.000016|       150 |  | mysql-bin.000017|       483 |  +------------------+-----------+  11 rows in set (0.04sec)  [root@stu18 data]#cat mysql-bin.index  ./mysql-bin.000007  ./mysql-bin.000008  ./mysql-bin.000009  ./mysql-bin.000010  ./mysql-bin.000011  ./mysql-bin.000012  ./mysql-bin.000013  ./mysql-bin.000014  ./mysql-bin.000015  ./mysql-bin.000016  ./mysql-bin.000017 |

由此可以看出这种清理二进制日志文件的方式是非常合理的，不会导致数据库的错误发生。

|  |  |
| --- | --- |
| 1  2 | mysql> PURGEBINARY LOGS BEFORE '13-10-19 10:26:36'; #使用时间来删除二进制日志  Query OK, 0 rowsaffected (0.05 sec) |

到此关于二进制的知识就解析完了，其中若有错误不足之处请指出！谢谢！注意二进制日志和错误日志很重要重点学之！！

千里之行，始于足下。改变现在，就是改变未来。改变未来，从现在开始。 个人网站：[http://www.wangkongming.cn](http://www.wangkongming.cn/)  
企业网站：<http://www.hlbaozhuangji.cn/>

# 管理

## 远程访问测试数据库

mysqldump -h120.24.229.176 -uroot -p -P3307 sagittarius > bak-uat-20160120.sql

mysql -h 120.24.229.176 -u root -p --port=3307

55BSgc{aq

## ★MySQL查询-分组取组中某字段最大（小）值所有记录 - 九九艳阳天 - 博客园

http://www.cnblogs.com/chenxl/p/4370278.html

select id,player\_id,efficiency,record\_time from (select \* from player\_game\_log order by record\_time desc) as pgl\_temp group by pgl\_temp.player\_id

最近做东西的时候，用到一个数据库的查询。将记录按某个字段分组，取每个分组中某个字段的最大值的所有记录。举栗子来说。

已知分数表“score”，包含字段“id", "name", "course", "score"。包含记录如下图所示。

　　s

　　现在，要求查询每个人分数最高的课程的那一条记录。即获得如下结果：

　　大概思路是，先通过order by将记录按score字段排序，创建临时表。然后按name字段分组查询临时表，即可获得所需查询结果。完整的sql语句如下：

SELECT ns.id, ns.name, ns.course, ns.score FROM ( SELECT id, name, course, score FROM score ORDER BY score DESC) AS ns GROUP BY ns.name ORDER BY ns.id;

　　当然要获取分组的最小值，只需在创建临时表ns时，按升序排列 (ORDER BY score ASC) 即可。

　　 以上。

## ★mysql查询昨天 一周前 一月前 一年前的数据\_Mysql\_脚本之家

http://www.jb51.net/article/50509.htm

mysql 昨天 一周前 一月前 一年前的数据 这里主要用到了DATE\_SUB，

参考如下

复制代码 代码如下:

SELECT \* FROM yh\_content

where inputtime>DATE\_SUB(CURDATE(), INTERVAL 1 DAY)

where inputtime>DATE\_SUB(CURDATE(), INTERVAL 1 WEEK)

where inputtime>DATE\_SUB(CURDATE(), INTERVAL 1 MONTH)

where inputtime>DATE\_SUB(CURDATE(), INTERVAL 1 YEAR)

注意：如果数据库中时间以UNIX时间戳的形式存放的话，在时间对比上需要更改为统一格式：

DATE\_SUB()返回的是格式化后的时间：2014-05-17

需要用UNIX\_TIMESTAMP()转化为UNIX时间戳形式对比：

复制代码 代码如下:

where inputtime>UNIX\_TIMESTAMP(DATE\_SUB(CURDATE(), INTERVAL 1 DAY))

where inputtime>UNIX\_TIMESTAMP(DATE\_SUB(CURDATE(), INTERVAL 1 WEEK))

where inputtime>UNIX\_TIMESTAMP(DATE\_SUB(CURDATE(), INTERVAL 1 MONTH))

where inputtime>UNIX\_TIMESTAMP(DATE\_SUB(CURDATE(), INTERVAL 1 YEAR))

详细请查看MySql时间函数：DATE\_SUB、DATE\_ADD、UNIX\_TIMESTAMP等函数的用法

mysql中获取一天、一周、一月时间数据的各种sql语句写法

今天抽时间整理了一篇mysql中与天、周、月有关的时间数据的sql语句的各种写法，部分是收集资料，全部手工整理，自己学习的同时，分享给大家，并首先默认创建一个表、插入2条数据，便于部分数据的测试，其中部分名词或函数进行了解释说明。直入主题！

创建表：

复制代码 代码如下:

create table if not exists t

(

id int,

addTime datetime default '0000-00-00 00:00:00′

)

添加两条初始数据：

insert t values(1, '2012-07-12 21:00:00′);

insert t values(2, '2012-07-22 21:00:00′);

一、当天或当日插入的数据：

1、传统对比判断：SELECT \* FROM `t` WHERE DATE\_FORMAT(addTime,'%Y-%m-%d') = date\_format(now(),'%Y-%m-%d')");

2、第一点的简写：SELECT \* FROM `t` WHERE addTime >= date\_format(NOW(),'%Y-%m-%d');

3、利用函数判断：SELECT \* FROM `t` WHERE DATEDIFF(addTime,NOW()) =0;//推荐此方法

4、利用时间戳判断：SELECT \* FROM `t` WHERE addTime BETWEEN (UNIX\_TIMESTAMP(now()-86440)) AND now();

注：返回当天时间的mysql函数有CURDATE()、CURRENT\_DATE()、CURRENT\_DATE、NOW()几种；其中NOW()获取的日期格式为0000-00-00 00:00:00的时间；CURDATE()、CURRENT\_DATE()、CURRENT\_DATE是获取日期格式为0000-00-00的时间，所以返回的时间无时分秒；

二、当月的相关数据写法：

1、今天是当月的第几天：SELECT DAYOFMONTH( NOW( ) );

2、获取当月的第一天时间：SELECT DATA\_SUB( NOW(), INTERVAL DAYOFMONTH(NOW()) – 1 DAY);

日期运算函数，句型：date\_add(date,INTERVAL expr type)和date\_sub(date,INTERVAL expr type)

date为一个datetime或date值，可当作起始时间，expr为一个表达式，用来指定增加或减少的时间间隔数目，可为一个字符串–若为负值，则在表达式前加个"-"符号。type为关键词，它表示了表达式被解释的方式，通常用年(year)、月(month)、日(day)、周(week)等。

INTERVAL的用户说明：

1、当函数使用时，即INTERVAL(),为比较函数，如：INTERVAL(10,1,3,5,7); 结果为4；

原理：10为被比较数，后面1,3,5,7为比较数，将后面四个依次与10比较，看后面数字组有多少个少于10，则返回其个数。前提是后面数字组为从小到大排列，否则返回结果0。

2、当关键词使用时，表示为设置时间间隔，常用在date\_add()与date\_sub()函数里，如：INTERVAL 1 DAY ，解释为将时间间隔设置为1天。

弄清楚了上面几个重要的日期运算函数，下面再来一个混合的日期运算。

3、上个月的第一天时间：SELECT DATE\_SUB( DATE\_SUB( NOW( ) , INTERVAL DAYOFMONTH( NOW( ) ) -1 DAY ) , INTERVAL 1 MONTH ); 是不是一目了然了！

三、当周的相关数据写法：

1、获取今天是一周第几天或星期几：SELECT WEEKDAY(now());返回的是数字：0为周一，6为周日

2、获取本周的第一天日期：SELECT DATE\_SUB(now(),INTERVAL WEEKDAY(now()) day);或SELECT DATE\_ADD(now(),INTERVAL -WEEKDAY(now()) day);或 SELECT CURDATE( ) – WEEKDAY( CURDATE( ) );

3、再写一个上周的第一天至现在的数据：（以表t为数据表）

SELECT \* FROM `t` WHERE addTime >= date\_format(date\_sub(date\_sub(now(), INTERVAL WEEKDAY(NOW()) DAY), INTERVAL 1 WEEK), ‘%Y-%m-%d');是不是有些感觉了！

注：若你是以时间戳保存的时间字段，那么请用from\_unixtime()函数转换为时间格式，如：from\_unixtime($timestamp)

四、Mysql中将日期转为时间戳

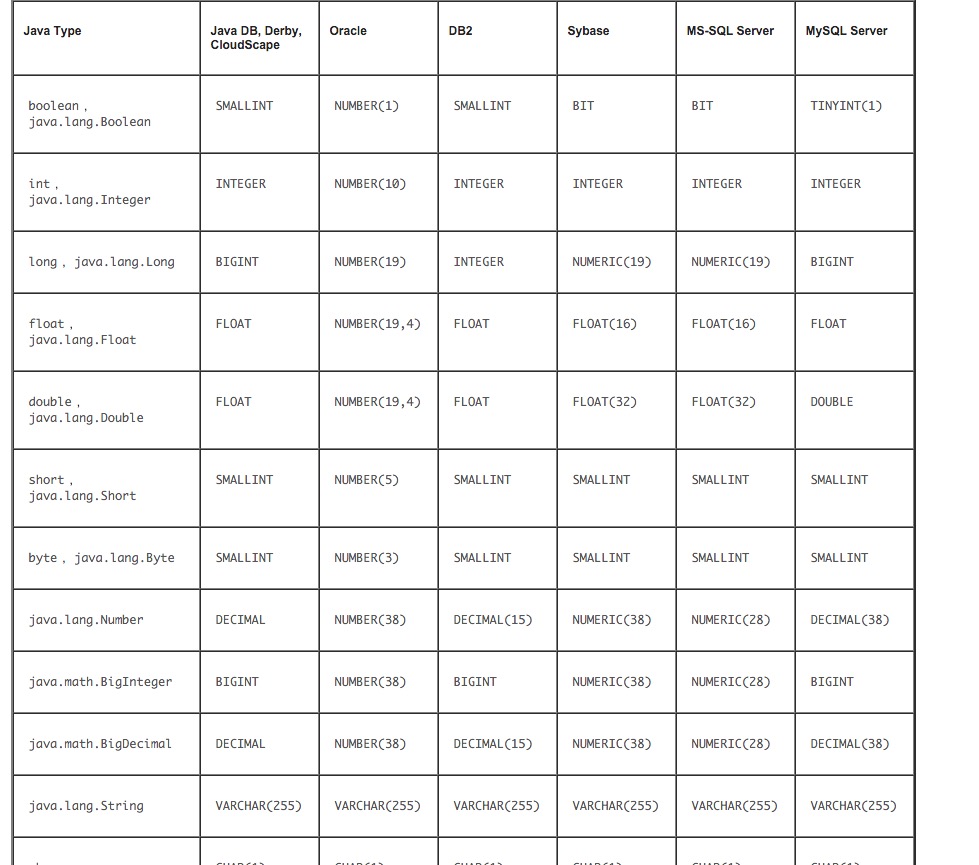
前三点的方法，讲的都是将日期转为相应时间常量，并没有提到时间戳与时间的转换，这里把它写出来，

1、时间戳转日期，方法是select from\_unixtime(1336542121);

2、日期转时间戳，方法是：select unix\_timestamp('2013-04-08′);

结合前面3点，对于时间戳就更能灵活运用了！最后更新于：2013.04.08

## 数据库映射



## MySql中varchar(10)和varchar(100)的区别==>>以及char的利弊 - imzoer的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/imzoer/article/details/8435540

一般初学会认为，二者占用的空间是一样的。比如说我存储5个char，二者都是实际占用了5个char了【勘误：varchar在实际存储的时候会多一个byte用来存放长度】。

但是深入一下，设计数据库的时候，二者一样吗？

答案是否定的【至少varchar类型需要在数据之前利用一个或者两个字节来存储数据的长度】【二者在内存中的操作方式也是不同的，下面的例子中有体现】。看下面的例子。

如现在用户需要存储一个地址信息。根据评估，只要使用100个字符就可以了。但是有些数据库管理员会认为，反正Varchar数据类型是根据实际的需要来分配长度的。还不如给其大一点的呢。为此他们可能会为这个字段一次性分配200个字符的存储空间。这VARCHAR(100)与VARCHAR(200)真的相同吗?结果是否定的。虽然他们用来存储90个字符的数据，其存储空间相同。但是对于内存的消耗是不同的。对于VARCHAR数据类型来说，硬盘上的存储空间虽然都是根据实际字符长度来分配存储空间的，但是对于内存来说，则不是。其时使用固定大小的内存块来保存值。简单的说，就是使用字符类型中定义的长度，即200个字符空间。显然，这对于排序或者临时表(这些内容都需要通过内存来实现)作业会产生比较大的不利影响。解释可以参见这里。所以如果某些字段会涉及到文件排序或者基于磁盘的临时表时，分配VARCHAR数据类型时仍然不能够太过于慷慨。还是要评估实际需要的长度，然后选择一个最长的字段来设置字符长度。如果为了考虑冗余，可以留10%左右的字符长度。千万不能认为其为根据实际长度来分配存储空间，而随意的分配长度，或者说干脆使用最大的字符长度。

----------------------------------char------------------------------------------

1、从碎片角度进行考虑，使用CHAR字符型时，由于存储空间都是一次性分配的。为此某个字段的内容，其都是存储在一起的。单从这个角度来讲，其不存在碎片的困扰。而可变长度的字符数据类型，其存储的长度是可变的。当其更改前后数据长度不一致时，就不可避免的会出现碎片的问题。故使用可变长度的字符型数据时，数据库管理员要时不时的对碎片进行整理。如执行数据库导出导入作业，来消除碎片。

2、考虑其长度的是否相近，如果某个字段其长度虽然比较长，但是其长度总是近似的，如一般在90个到100个字符之间，甚至是相同的长度。此时比较适合采用CHAR字符类型。比较典型的应用就是MD5哈希值。当利用MD5哈希值来存储用户密码时，就非常使用采用CHAR字符类型。因为其长度是相同的。另外，像用来存储用户的身份证号码等等，一般也建议使用CHAR类型的数据。

另外请大家考虑一个问题，CHAR(1)与VARCHAR(1)两这个定义，会有什么区别呢?虽然这两个都只能够用来保存单个的字符，但是VARCHAR要比CHAR多占用一个存储位置。这主要是因为使用VARCHAR数据类型时，会多用1个字节用来存储长度信息。这个管理上的开销char字符类型是没有的。

---------------------------------总结---------------------------------------------

二者在磁盘上存储占的空间是一样的。区别有二。第一、一个变长一个固定长度。第二、在内存中的操作方式，varchar也是按照最长的方式在内存中进行操作的。比如说要进行排序的时候，varcahr(100)是按照100这个长度来进行的。

-----------------------------------------------------------------------------------

varchar的最大长度是多少呢？

参见这里

mysql的vachar字段的类型虽然最大长度是65535，但是并不是能存这么多数据，最大可以到65533（不允许非空字段的时候），当允许非空字段的时候只能到65532【在允许空的时候，varchar(65532) will be 2 bytes (length) + up to 65532 chars (latin1) + 1 null byte】【还不清楚这一个null byte的作用是什么。后续了解】。

-------------------------------------------------------------------------------------

请注意所有MySQL校对规则属于PADSPACE类。这说明在MySQL中的所有CHAR和VARCHAR值比较时不需要考虑任何尾部空格。请注意所有MySQL版本均如此，并且它不受SQL服务器模式的影响。【参见这里的note部分上面几行文字】

下图为证：

根据上面地址提供的mysql手册，如果在一个char或者varchar列上建立唯一索引之后，那么'a'和'a '，会引起duplicate-key error。

## MySQL查询条件中字符串包含空格的问题 - delphiwcdj的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/delphiwcdj/article/details/16983993

问题

最近在联调某个业务时发现使用的签名总是验证不过，在MySQL中查询了该业务的私钥配置和业务侧的配置是一样的，问题就出在SQL查询这里，最后将配置导出到本地发现私钥后面多了一个空格，将空格删除然后签名计算OK。问题是：为什么在DB查询条件中的字符串没有包含空格也可以查到实际包含空格的这条记录呢？

原因

如果字段是char或varchar类型，那么在字符串比较的时候MySQL使用PADSPACE校对规则，会忽略字段末尾的空格字符，若想做到精确匹配可以使用下面几种方法：

方法1：使用like语句；

方法2：使用binary类型，例如，select binary 'a' = 'a   '

方法3：再添加一个length()条件，例如，select col from table where col = 'a   ' and LENGTH(col) = LENGTH('a   ')

官方手册说明（5.0版本）：

<http://dev.mysql.com/doc/refman/5.0/en/char.html>

11.1.6.1. The CHAR and VARCHAR Types

 All MySQL collations are of type PADSPACE. This means that all CHAR, VARCHAR, and TEXT values in MySQL are compared without regard to any trailing spaces. “Comparison” in this context does not include the [LIKE](http://dev.mysql.com/doc/refman/5.0/en/string-comparison-functions.html#operator_like) pattern-matching operator, for which trailing spaces are significant.

在stackoverfolw上也可以找到类似的问题：

MySQL comparison operator, spaces

<http://stackoverflow.com/questions/10495692/mysql-comparison-operator-spaces>

## MySQL数据类型：UNSIGNED注意事项 - blankqdb - 博客园

http://www.cnblogs.com/blankqdb/archive/2012/11/03/blank\_qdb.html

1. UNSIGNED

UNSIGNED属性就是将数字类型无符号化，与C、C++这些程序语言中的unsigned含义相同。例如，INT的类型范围是-2 147 483 648 ～ 2 147 483 647， INT UNSIGNED的范围类型就是0 ～ 4 294 967 295。

在MYSQL中整型范围：

类型                 大小            范围（有符号）                              范围（无符号） 用途

TINYINT           1 字节    (-128，127)                                    (0，255) 小整数值

SMALLINT        2 字节    (-32 768，32 767)                           (0，65 535) 大整数值

MEDIUMINT     3 字节    (-8 388 608，8 388 607)                  (0，16 777 215) 大整数值

INT或INTEGER  4 字节    (-2 147 483 648，2 147 483 647)     (0，4 294 967 295) 大整数值

源文档 <<http://www.cnblogs.com/bukudekong/archive/2011/06/27/2091590.html>>

看起来这是一个不错的属性选项，特别是对于主键是自增长的类型，因为一般来说，用户都希望主键是非负数。然而在实际使用中，UNSIGNED可能会带来一些负面的影响，示例如下：

mysql> CREATE TABLE t ( a INT UNSIGNED, b INT UNSIGNED )

ENGINE=INNODB;

Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO t SELECT 1,2;

Query OK, 1 row affected (0.00 sec)

Records: 1 Duplicates: 0 Warnings: 0

mysql> SELECT \* FROM t\G;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

a: 1

b: 2

1 row in set (0.00 sec)

我们创建了一个表t，存储引擎为InnoDB。表t上有两个UNSIGNED的INT类型。输入(1，2)这一行数据，目前看来都没有问题，接着运行如下语句：

SELECT a - b FROM t

这时结果会是什么呢?会是-1吗?答案是不确定的，可以是-1，也可以是一个很大的正值，还可能会报错。在Mac[操作系统](http://product.it168.com/list/b/0501_1.shtml)中(windows中也会)，MySQL数据库提示如下错误：

mysql> SELECT a-b FROM t;

ERROR 1690 (22003): BIGINT UNSIGNED value is out of range in '(`test`.`t`.`a` - `test`.`t`.`b`)'

这个错误乍看起来非常奇怪，提示BIGINT UNSIGNED超出了范围，但是我们采用的类型都是INT UNSIGNED啊!而在另一台Linux[操作系统](http://product.pcpop.com/Operating_System/10734_1.html)中，运行的结果却是：

mysql> SELECT a -b FROM t\G;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

a - b: 4294967295

1 row in set (0.00 sec)

在发生上述这个问题的时候，有开发人员跑来和笔者说，他发现了一个MySQL的Bug，MySQL怎么会这么“傻”呢?在听完他的叙述之后，我写了如下的代码并告诉他，这不是MySQL的Bug，C语言同样也会这么“傻”。

#include

int main(){

unsigned int a;

unsigned int b;

a = 1;

b = 2;

printf(a - b: %d\n,a-b);

printf(a - b: %u\n,a-b);

return 1;

}

上述代码的运行结果是：

a - b: -1

a - b: 4294967295

可以看到，在C语言中a-b也可以返回一个非常巨大的整型数，这个值是INT UNSIGNED的最大值。难道C语言也发生了Bug?这怎么可能呢?

在实际的使用过程中，MySQL给开发人员的印象就是存在很多Bug，只要结果出乎预料或者有开发人员不能理解的情况发生时，他们往往把这归咎于MySQL的 Bug。和其他数据库一样，MySQL的确存在一些Bug，其实并不是MySQL数据库的Bug比较多，去看一下Oracle RAC的Bug，那可能就更多了，它可是Oracle的一款旗舰产品。因此，不能简单地认为这个问题是MySQL的Bug。

对于上述这个问题，正如上述所分析的，如果理解整型数在数据库中的表示方法，那么这些就非常好理解了，这也是为什么之前强调需要看一些计算机组成原理方面相关书籍的原因。将上面的C程序做一些修改：

#include

int main(){

unsigned int a;

unsigned int b;

a = 1;

b = 2;

printf(a - b: %d,%x\n,a-b,a-b);

printf(a - b: %u,%x\n,a-b,a-b);

return 1;

}

这次不仅打印出a-b的结果，也打印出a-b的十六进制结果，运行程序后的结果如下所示：

a - b: -1,ffffffff

a - b: 4294967295,ffffffff

可以看到结果都是0xFFFFFFFF，只是0xFFFFFFFF可以代表两种值：对于无符号的整型值，其是整型数的最大值，即4 294 967 295;对于有符号的整型数来说，第一位代表符号位，如果是1，表示是负数，这时应该是取反加1得到负数值，即-1。

这个问题的核心是，在MySQL数据库中，对于UNSIGNED数的操作，其返回值都是UNSIGNED的。而正负数这个问题在《MySQL技术内幕：InnoDB存储引擎》中有更深入的分析，有兴趣的可以进一步研究。

那么，怎么获得-1这个值呢?这并不是一件难事，只要对SQL\_MODE这个参数进行设置即可，例如：

mysql>SET sql\_mode='NO\_UNSIGNED\_SUBTRACTION';

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT a-b FROM t\G;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

a-b: -1

1 row in set (0.00 sec)

后面会对SQL\_MODE进一步讨论，这里不进行深入的讨论。笔者个人的看法是尽量不要使用UNSIGNED，因为可能会带来一些意想不到的效果。另外，对于INT类型可能存放不了的数据，INT UNSIGNED同样可能存放不了，与其如此，还不如在数据库设计阶段将INT类型提升为BIGINT类型。

以上文字摘自<<http://tech.it168.com/a2012/0808/1382/000001382732.shtml>>

本人遇到的类似问题：(linux上)

当(a-b)在where子句后时也会出现相同的情况

以下是php使用Mysql查询的结果(每组的第一行是第二行[1]-[2]的结果)

86374

                       a                                       b

Array (  [1] => 1351843032  [2] => 1351756658  )

-2567

Array ( [1] => 1351843032  [2] => 1351845599  )

86374

Array ([1] => 1351843032  [2] => 1351756658 )

86374

Array (  [1] => 1351843032  [2] => 1351756658  )

-105849

Array (  [1] => 1351650809  [2] => 1351756658 )

86374

Array (  [1] => 1351843032 [2] => 1351756658  )

86374

Array ( [1] => 1351843032  [2] => 1351756658  )

下面在mysql语句中查询select \* from table where (a-b)>86374;

结果(按正常思路来讲，结果应该为空，但在Linux是却现出以下结果 )：

Array ( [1] => 1351843032  [2] => 1351845599  )

Array ( [1] => 1351650809  [2] => 1351756658  )

而这个结果恰是[1]-[2]为负数的那两行。

结论：如果使用unsigne并且在where子句后出现两列相减值小于0((a-b)<0)，在查询时，linux上的Mysql会将负数转换成unsigned后再进行查询( (-2576+4294967295+1)>86374,  (-105849+4294967295+1)>86374 )。

## ★查询到所有表字段的信息

select table\_schema,table\_name,column\_name   
from information\_schema.columns   
order by table\_schema,table\_name

select \*

from information\_schema.columns

where TABLE\_SCHEMA = 'sagittarius' and COLUMN\_TYPE like '%unsigned%'

order by table\_schema,table\_name;

select TABLE\_NAME,column\_name,column\_type

from information\_schema.columns

where TABLE\_SCHEMA = 'sagittarius' and COLUMN\_TYPE like '%unsigned%'

order by table\_schema,table\_name;

TABLE\_NAME column\_name column\_type

account source int(11) unsigned

contest entry\_fee int(11) unsigned

contest prize int(11) unsigned

player\_game\_stats\_item score double unsigned

player\_stats\_item score double unsigned

roster\_item salary int(11) unsigned

signing\_player points double unsigned

signing\_player salary int(11) unsigned

standings prize double unsigned

## ★EXPLAIN sql优化方法（3）DERIVED\_jordan18\_新浪博客

http://blog.sina.com.cn/s/blog\_3f2a82610100tqfh.html

派生表和视图的性能

从MySQL 4.1开始，它已经支持派生表、联机视图或者基本的FROM从句的子查询。

这些特性之间彼此相关，但是它们之间的性能比较如何呢？

MySQL 5.0 中的派生表似乎和视图实现的方式不同，尽管我从合并的代码基数来看觉得在查询优化上应该是一样的。

派生表仍然以临时表的方式显式地处理，而且还是没有索引的临时表（因此最好不要像在例子中那样连接2个派生表）

需要考虑的另一方面是，派生表需要被显式处理，尽管只是执行 EXPLAIN 语句。因此如果在 FROM 字句中的 SELELCT 操作上犯了错误，例如忘记了写上连接的条件，那么 EXPLAIN 可能会一直在运行。

视图则不同，它无需被显式处理，只是把查询简单地重写了一下。只有在无法合并查询或者试图创建者请求时才需要被显式处理。

这意味着它们在性能上的差别如下：

在基本的表 上执行有索引 的查询，这非常快

mysql> SELECT \* FROM test WHERE i=5 ;

+---+----------------------------------+

| i | j |

+---+----------------------------------+

| 5 | 0c88dedb358cd96c9069b73a57682a45 |

+---+----------------------------------+

1 row IN SET ( 0 .03 sec)

在派生表 上做同样 的查询，则如老牛拉破车

mysql> SELECT \* FROM ( SELECT \* FROM test) t WHERE i=5 ;

+---+----------------------------------+

| i | j |

+---+----------------------------------+

| 5 | 0c88dedb358cd96c9069b73a57682a45 |

+---+----------------------------------+

1 row IN SET ( 1 min 40 .86 sec)

在视图 上查询，又快起来了

mysql> CREATE VIEW v AS SELECT \* FROM test;

Query OK, 0 rows affected ( 0 .08 sec)

mysql> SELECT \* FROM v WHERE i=5 ;

+---+----------------------------------+

| i | j |

+---+----------------------------------+

| 5 | 0c88dedb358cd96c9069b73a57682a45 |

+---+----------------------------------+

1 row IN SET ( 0 .10 sec)

下面的2条EXPLAIN结果也许会让你很惊讶

mysql> EXPLAIN SELECT \* FROM v WHERE i=5 ;

+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+

| id | select\_type | TABLE | type | possible\_keys | KEY | key\_len | ref | rows | Extra |

+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+

| 1 | PRIMARY | test | const | PRIMARY | PRIMARY | 4 | const | 1 | |

+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+

1 row IN SET ( 0 .02 sec)

mysql> EXPLAIN SELECT \* FROM ( SELECT \* FROM test) t WHERE i=5 ;

+----+-------------+------------+------+---------------+------+---------+------+---------+-------------+

| id | select\_type | TABLE | type | possible\_keys | KEY | key\_len | ref | rows | Extra |

+----+-------------+------------+------+---------------+------+---------+------+---------+-------------+

| 1 | PRIMARY | <derived2> | ALL | NULL | NULL | NULL | NULL | 1638400 | USING WHERE |

| 2 | DERIVED | test | ALL | NULL | NULL | NULL | NULL | 1638400 | |

+----+-------------+------------+------+---------------+------+---------+------+---------+-------------+

2 rows IN SET ( 54 .90 sec)

避免使用派生表 -- 如果可能，最好采用其他方式来编写查询语句，大部分情况都比派生表来的快。很多情况下，甚至连独立的临时表都来的快，因为可以适当增加索引。

可以考虑使用临时试图来取代派生表 如果确实需要在 FROM 子句中使用到子查询，可以考虑在查询时创建试图，当查询完之后删除试图。

不适合多表视图，多表时用派生表 取代 视图

explain select sum(pdm.qty) pre\_total,pd.pre\_doc\_id from prepare\_doc pd

left join pre\_doc\_item pdm on pd.pre\_doc\_id=pdm.pre\_doc\_id group by pd.pre\_doc\_id

EXPLAIN <wbr>sql优化方法（3）DERIVED

## mysql格式化小数保留小数点后两位(小数点格式化)\_Mysql\_脚本之家

http://www.jb51.net/article/44378.htm

代码如下:

SELECT FORMAT(12562.6655,2);

结果：12,562.67

查看文档：Formats the number X to a format like '#,###,###.##', rounded to D decimal places, and returns the result as a string. If D is 0, the result has no decimal point or fractional part.整数部分超过三位的时候以逗号分割，并且返回的结果是string类型的。

复制代码 代码如下:

mysql> SELECT FORMAT(12332.123456, 4);

-> '12,332.1235'

mysql> SELECT FORMAT(12332.1,4);

-> '12,332.1000'

mysql> SELECT FORMAT(12332.2,0);

-> '12,332'

没有达到预期结果，想要的结果不要以逗号分隔，

复制代码 代码如下:

select truncate(4545.1366,2);

结果：4545.13，直接截取不四舍五入，还是有问题。

复制代码 代码如下:

select convert(4545.1366,decimal);

## 找到mysql配置文件位置

find . -name 'my.cnf'

## 特殊字符

**ALTER TABLE** contest MODIFY `name` **varchar**(50) **CHARACTER SET** utf8mb4 **COLLATE** utf8mb4\_unicode\_ci;  
**ALTER TABLE** contest CHARSET=utf8mb4;  
  
**ALTER TABLE user** MODIFY `nickname` **varchar**(50) **CHARACTER SET** utf8mb4 **COLLATE** utf8mb4\_unicode\_ci;  
**ALTER TABLE user** CHARSET=utf8mb4;

## Emoji表情符号录入MySQL数据库报错的解决方案 -mchdba-ITPUB博客

http://blog.itpub.net/26230597/viewspace-1243233/

find . -name 'my.cnf'

将容器中的mysql配置文件拷贝出来

docker cp mysql:/etc/mysql/my.cnf /home/mysql/conf

/etc/mysql/my.cnf

vim my.cnf

init-connect='SET NAMES utf8mb4'

character-set-server=utf8mb4

**前言：手机app应用评论的时候，恢复表情符号，提示失败。**  
  
**1，查看tomcat后台日志，核心报错信息如下：**  
 Caused by: java.sql.SQLException: Incorrect string value: '\xF0\x9F\x98\x97\xF0\x9F...' for column 'CONTENT' at row 1  
at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:1074)  
at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:4096)  
at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:4028)  
at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:2490)  
at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:2651)  
at com.mysql.jdbc.ConnectionImpl.execSQL(ConnectionImpl.java:2734)  
at com.mysql.jdbc.PreparedStatement.executeInternal(PreparedStatement.java:2155)  
at com.mysql.jdbc.PreparedStatement.execute(PreparedStatement.java:1379)  
at org.apache.commons.dbcp.DelegatingPreparedStatement.execute(DelegatingPreparedStatement.java:172)  
at org.apache.commons.dbcp.DelegatingPreparedStatement.execute(DelegatingPreparedStatement.java:172)  
at com.ibatis.sqlmap.engine.execution.SqlExecutor.executeUpdate(SqlExecutor.java:80)  
at com.ibatis.sqlmap.engine.mapping.statement.MappedStatement.sqlExecuteUpdate(MappedStatement.java:216)  
at com.ibatis.sqlmap.engine.mapping.statement.MappedStatement.executeUpdate(MappedStatement.java:94)  
... 46 more   
   
[root@mysqlvm2 ~]# mysql -root -p  
mysql> use test;  
Database changed  
mysql> show tables;  
Ignoring query to other database  
mysql> Ctrl-C -- exit!  
Aborted  
忘记输入-u参数了，冲洗你输入，OK，如下所示：  
[root@mysqlvm2 ~]# mysql -uroot -p  
Enter password:   
Welcome to the MySQL monitor.  Commands end with ; or \g.  
Your MySQL connection id is 4  
Server version: 5.6.12-log Source distribution  
Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.  
Oracle is a registered trademark of Oracle Corporation and/or its  
affiliates. Other names may be trademarks of their respective  
owners.  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
mysql>    
mysql> use test;  
Database changed  
mysql> show tables;  
+------------------------+  
| Tables\_in\_test         |  
+------------------------+  
| c                      |   
| lubin\_test             |   
| test                   |   
| tt                     |   
+------------------------+  
10 rows in set (0.00 sec)  
这种输入Incorrect string value: '\xF0\x9F\x98\x97\xF0\x9F...'的问题，多数都是字符集的，以前从latain改成gbk，从gbk改成utf8，而我的CONTENT字段已经utf8了，那么比utf8更多的就只有utf8mb4了，所以去修改表字段的字符集吧。  
  
mysql>  
  
  
**2，先去修改表字段字符集为utf8mb4：**  
ALTER TABLE UGC\_REVIEW\_CONTENT MODIFY `CONTENT` TEXT CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci COMMENT '评论内容';  
执行完后，再在手机端的app测试，依然报同样的错误。  
  
  
**3，再去修改表字符集utf8mb4：**  
ALTER TABLE UGC\_REVIEW\_CONTENT  CHARSET=utf8mb4 COMMENT='晒单/推荐的评论内容';  
执行完后，再在手机端的app测试，依然报同样的错误。  
  
  
**4，再去修改数据库的字符集utf8mb4:**  
vim my.cnf  
init-connect='SET NAMES utf8mb4'  
character-set-server=utf8mb4  
  
重启mysql数据库  
[root@mysqlvm4 ~]# service mysql restart  
Shutting down MySQL....                                    [确定]  
Starting MySQL.............................................[确定]..  
[root@mysqlvm4 ~]#   
  
查看db的字符集   
mysql> show variables like '%char%';  
+--------------------------+----------------------------------+  
| Variable\_name            | Value                            |  
+--------------------------+----------------------------------+  
| character\_set\_client     | utf8mb4                          |  
| character\_set\_connection | utf8mb4                          |  
| character\_set\_database   | utf8mb4                          |  
| character\_set\_filesystem | binary                           |  
| character\_set\_results    | utf8mb4                          |  
| character\_set\_server     | utf8mb4                          |  
| character\_set\_system     | utf8                             |  
| character\_sets\_dir       | /usr/local/mysql/share/charsets/ |  
+--------------------------+----------------------------------+  
8 rows in set (0.00 sec)  
mysql>   
  
**在手机app端重新评论，输入表情符号，点击提交提示评论成功。**  
  
**5，问题分析总结：**  
    ㈠ 原因  
        普通的字符串或者表情都是占位3个字节，所以utf8足够用了，但是移动端的表情符号占位是4个字节，普通的utf8就不够用了，为了应对无线互联网的机遇和挑战、避免 emoji 表情符号带来的问题、涉及无线相关的 MySQL 数据库建议都提前采用 utf8mb4 字符集，这必须要作为移动互联网行业的一个技术选型的要点  
    ㈡ 限制  
        需要 >= MySQL 5.5.3版本、从库也必须是5.5的了、低版本不支持这个字符集、复制报错。  
  
参考文章地址：http://bbs.csdn.net/topics/390055415

## mysql常用函数

SELECT UNIX\_TIMESTAMP('2015-09-19 19:00:00'), unix\_timestamp('2015-09-18 19:00:00');

select from\_unixtime(1443115345);

select from\_unixtime(unix\_timestamp())

前一小时的时间

date\_sub(NOW(),interval 60 minute);

一、MySQL中两个DateTime字段相减

得到两个日期字段之间的秒数

selec t (UNIX\_TIMESTAMP(endDateTime) - UNIX\_TIMESTAMP(beginDateTime)) dif\_second from tblName

得到两个日期字段之间的分数

selec t (UNIX\_TIMESTAMP(endDateTime) - UNIX\_TIMESTAMP(beginDateTime))/60 dif\_minute from tblName

得到两个日期字段之间的天数

selec t (UNIX\_TIMESTAMP(endDateTime) - UNIX\_TIMESTAMP(beginDateTime))/(60\*60\*24) dif\_minute from tblName

二、MySQL中两个Time字段相减

如果两个字段都为Time类型，如果两个时间都在同一天，相减可以得到相差的秒数，但如果跨天，月，年都有问题。

selec t (TIME\_TO\_SEC(endDateTime) - TIME\_TO\_SEC(beginDateTime)) dif\_second from tblName

## MYSQL在众多表中查询表名和字段名

SELECT \* from information\_schema.columns WHERE column\_name LIKE '%stats%';

## docker mysqldump

#远程dump

mysqldump -h 120.24.229.176 -uroot -p --port=3307 sagittarius > back-20151009.sql

#备份

mysqldump -uroot -p sagittarius > back-20150923.sql

#在本地执行，从容器拷贝到本地

docker cp /root/back-20150923.sql acceptance\_mysql\_1:/

#将数据库脚本导入到数据库

use sagittarius

source back-20150923.sql

source back-20151009.sql

## 产生序号

select user.nickname, @rownum:=@rownum+1 AS rownum,from\_unixtime(user\_login\_log.login\_time)

from user\_login\_log,user,(SELECT @rownum:=0) as r where user\_login\_log.user\_id=user.id order by user\_login\_log.login\_time;

## mysql随机字符串和数值的获取\_章郎虫\_新浪博客

http://blog.sina.com.cn/s/blog\_4adf6c7b01016a3x.html

今天博主在坛子里头看到一位朋友的提问，要求是插入随机字符串。具体描述如下：

**在MYSQL中，列出前6位固定为123456的16位随机字符串。随机字符串包括字母a-z、A-Z、0-9。**

方法很多，博主在这里和大家分享下：

一：select concat('123456',left(replace(uuid(),'-',''),10)) as rand\_str;”

二：SELECT CONCAT('123456',  
CASE ROUND((RAND())\*2) WHEN 0 THEN ROUND((RAND())\*9)  WHEN 1 THEN CHAR(ROUND((RAND())\*25)+97) ELSE CHAR(ROUND((RAND())\*25)+65) END,  
CASE ROUND((RAND())\*2) WHEN 0 THEN ROUND((RAND())\*9)  WHEN 1 THEN CHAR(ROUND((RAND())\*25)+97) ELSE CHAR(ROUND((RAND())\*25)+65) END,  
CASE ROUND((RAND())\*2) WHEN 0 THEN ROUND((RAND())\*9)  WHEN 1 THEN CHAR(ROUND((RAND())\*25)+97) ELSE CHAR(ROUND((RAND())\*25)+65) END,  
CASE ROUND((RAND())\*2) WHEN 0 THEN ROUND((RAND())\*9)  WHEN 1 THEN CHAR(ROUND((RAND())\*25)+97) ELSE CHAR(ROUND((RAND())\*25)+65) END,  
CASE ROUND((RAND())\*2) WHEN 0 THEN ROUND((RAND())\*9)  WHEN 1 THEN CHAR(ROUND((RAND())\*25)+97) ELSE CHAR(ROUND((RAND())\*25)+65) END,  
CASE ROUND((RAND())\*2) WHEN 0 THEN ROUND((RAND())\*9)  WHEN 1 THEN CHAR(ROUND((RAND())\*25)+97) ELSE CHAR(ROUND((RAND())\*25)+65) END,  
CASE ROUND((RAND())\*2) WHEN 0 THEN ROUND((RAND())\*9)  WHEN 1 THEN CHAR(ROUND((RAND())\*25)+97) ELSE CHAR(ROUND((RAND())\*25)+65) END,  
CASE ROUND((RAND())\*2) WHEN 0 THEN ROUND((RAND())\*9)  WHEN 1 THEN CHAR(ROUND((RAND())\*25)+97) ELSE CHAR(ROUND((RAND())\*25)+65) END,  
CASE ROUND((RAND())\*2) WHEN 0 THEN ROUND((RAND())\*9)  WHEN 1 THEN CHAR(ROUND((RAND())\*25)+97) ELSE CHAR(ROUND((RAND())\*25)+65) END,  
CASE ROUND((RAND())\*2) WHEN 0 THEN ROUND((RAND())\*9)  WHEN 1 THEN CHAR(ROUND((RAND())\*25)+97) ELSE CHAR(ROUND((RAND())\*25)+65) END  
) AS rand\_str;

其中+97就是产生小写字母a-z，+65就是产生大写字母A-Z。

第一个博主暂时看不懂，介绍下第二个吧。mysql中有一个随机函数rand()，会生成大于等于0，小于1(0到1)之间的小数。0到1的随机数乘以一个我们要随机范围，然后取整就可以，mysql中有round和floor函数都可以把小数变为整数。

FLOOR(X)  
返回不大于X的最大整数值。

ROUND(X,D)  
返回参数X的四舍五入的有D为小数的一个数字。如果D为0，结果将没有小数点或小数部分。

例如我们要0-99的随机数,只要下面的sql语句就可以了。

select floor(rand()\*100);

1到99的随机数就是

select floor(1+rand()\*99);

## MySQL select into 用法\_小鱼头\_新浪博客

http://blog.sina.com.cn/s/blog\_5623b16b0100hlgc.html

问题：要复制一张表中的部分字段到一张新表中去，并在新表中新增一个字段id，自动增加。  
  
1.首先想到的是用select into，但是MYSQL不支持这一语法  
MYSQL不支持:  
        Select \* Into new\_table\_name from old\_table\_name;  
替代方法:  
        Create table new\_table\_name (Select \* from old\_table\_name);  
  
Create table `sina20090712` (SELECT `title` , `time` , `original` , `content` , `author`  
FROM `sinapl\_article`  
ORDER BY `sinapl\_article`.`time` ASC);  
  
2.然后将这一表备份为sql文件，然后修改这一sql文件头部建立表结构的语句，增加一句  
  
DROP TABLE IF EXISTS `sina20090712`;  
CREATE TABLE `sina20090712` (  
  `id` int(11) NOT NULL AUTO\_INCREMENT,  
  `title` varchar(100) NOT NULL,  
  `time` varchar(20) DEFAULT NULL,  
  `original` varchar(20) DEFAULT NULL,  
  `content` text,  
  `author` varchar(100) DEFAULT NULL,  
    PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
注：红色部分为新增加的

## Mysql 多表Left join 查询\_crazydiy\_新浪博客

http://blog.sina.com.cn/s/blog\_7d2224110100uaxg.html

两个表这间的查询

SELECT c.serial,c.otherserial,c.os\_license\_id as date,u.name,u.firstname,u.realname from glpi\_computers as c left join glpi\_users as u on c.FK\_users=u.id

三个表之间的查询

SELECT c.serial,c.otherserial,c.os\_license\_id as date,u.name,u.firstname,u.realname,s.name from (glpi\_computers as c left join glpi\_users as u on c.FK\_users=u.id) left join glpi\_dropdown\_state as s on c.state=s.id

多个表之间的查询

SELECT c.serial as Serial\_Number,c.otherserial as FA\_Number,c.os\_license\_id as Buy\_Date, l.completename as Location,d.name as Section,m.name as Model,t.name as PC\_Type,u.name as GID,u.firstname as FirstName,u.realname as LastName,s.name as Status

from (((((glpi\_computers as c left join glpi\_dropdown\_domain as d on c.domain=d.id) left join glpi\_dropdown\_model as m on c.model=m.id) left join glpi\_dropdown\_state as s on c.state=s.id) left join glpi\_type\_computers as t on c.type=t.id) left join glpi\_users as u on c.FK\_users=u.id) left join glpi\_dropdown\_locations as l on c.location=l.id

SELECT c.serial as Serial\_Number,c.otherserial as FA\_Number,c.os\_license\_id as Buy\_Date,l.completename as Location,d.name as Section,m.name as Model,t.name as PC\_Type,u.name as GID,u.firstname as FirstName,u.realname as LastName,s.name as Status

from (((((glpi\_computers as c left join glpi\_dropdown\_domain as d on c.domain=d.id) left join glpi\_dropdown\_model as m on c.model=m.id) left join glpi\_dropdown\_state as s on c.state=s.id) left join

glpi\_type\_computers as t on c.type=t.id) left join glpi\_users as u on c.FK\_users=u.id) left join glpi\_dropdown\_locations as l on c.location=l.id

SELECT t.name as issue,c.completename as category,t.date as opendate,t.closedate,t.status FROM glpi\_tracking t, glpi\_dropdown\_tracking\_category c where t.category=c.id and c.completename like 'IT%' and t.date >'2011-07-31' and t.closedate <'2011-09-01'

## MySQL四舍五入函数ROUND(x)、ROUND(x,y)和TRUNCATE(x,y)\_百科369

http://www.baike369.com/content/?id=5591

MySQL四舍五入函数ROUND(x)

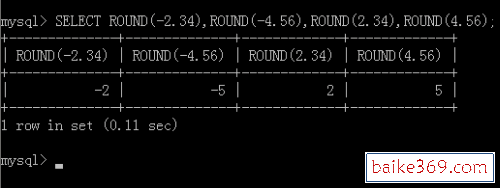
ROUND(x)函数返回最接近于参数x的整数，对x值进行四舍五入。

实例：

使用ROUND(x)函数对操作数进行四舍五入操作。SQL语句如下：

mysql>SELECT ROUND(-2.34),ROUND(-4.56),ROUND(2.34),ROUND(4.56);

ROUND(x)函数的执行结果如下图所示：



上图中代码执行的结果显示，进行四舍五入处理以后，只保留了各个值的整数部分。

MySQL四舍五入函数ROUND(x,y)

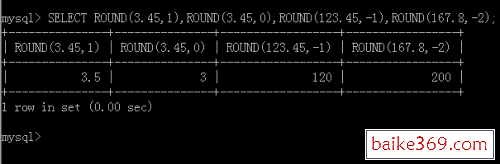
ROUND(x,y)函数返回最接近于参数x的数，其值保留到小数点后面y位，若y为负值，则将保留x值到小数点左边y位。

实例：

使用ROUND(x,y)函数对操作数进行四舍五入操作，结果保留小数点后面指定y位。SQL语句如下：

mysql>SELECT ROUND(3.45,1),ROUND(3.45,0),ROUND(123.45,-1),ROUND(167.8,-2);

ROUND(x,y)函数的执行结果如下图所示：



ROUND(3.45,1)保留小数点后面1位，四舍五入的结果为3.5；ROUND(3.45,0)保留小数点后面0位，即返回四舍五入后的整数值；ROUND(123.45,-1)保留小数点左边1位，即从小数点向左查1位，这1位用0代替，返回值为120；ROUND(167.8,-2)保留小数点左边2位，即从小数点向左查2位，这两位用0代替，返回四舍五入后的值200。

MySQL四舍五入函数TRUNCATE(x,y)

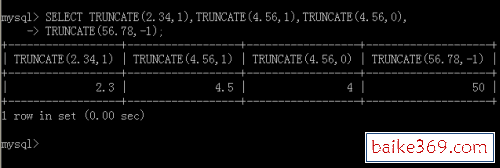
TRUNCATE(x,y)函数返回被舍去至小数点后y位的数字x。若y的值为0，则结果不带有小数点或不带有小数部分。若y设为负数，则截去（归零）x小数点左起第y位开始后面所有低位的值。

实例：

使用TRUNCATE(x,y)函数对操作数进行四舍五入操作，结果保留小数点后面指定y位。SQL语句如下：

mysql>SELECT TRUNCATE(2.34,1),TRUNCATE(4.56,1),TRUNCATE(4.56,0),TRUNCATE(56.78,-1);

TRUNCATE(x,y)函数的执行结果如下图所示：



TRUNCATE(2.34,1)和TRUNCATE(4.56,1)都保留小数点后1位数字，返回值分别为2.3和4.5；TRUNCATE(4.56,0)返回整数部分4；TRUNCATE（56.78,-1)截去小数点左边第1位后面的值，并将整数部分的1位数字置0，结果为50。

## MySQL-5.6.13免安装版配置方法 - Duke‘s IT Life - 博客频道 - CSDN.NET

http://blog.csdn.net/q98842674/article/details/12094777

. 下载MySQL Community Server 5.6.13

2. 解压MySQL压缩包  
    将以下载的MySQL压缩包解压到自定义目录下,我的解压目录是:  
    "D:\Program Files\MySQL\mysql-5.6.13-win32"  
    将解压目录下默认文件 my-default.ini 拷贝一份，改名 my.ini  
    复制下面的配置信息到 my.ini 保存  
    #如果没有my-default.ini,可自己新建my.ini或者从其他地方中获取  
#########################################################  
    [client]  
    port=3306  
    default-character-set=utf8  
    [mysqld]  
    port=3306  
    character\_set\_server=utf8  
    basedir=D:\Program Files\MySQL\mysql-5.6.13-win32  
    #解压目录  
    datadir=D:\Program Files\MySQL\mysql-5.6.13-win32\data  
    #解压目录下data目录  
    sql\_mode=NO\_ENGINE\_SUBSTITUTION,STRICT\_TRANS\_TABLES  
    [WinMySQLAdmin]  
    D:\Program Files\MySQL\mysql-5.6.13-win32\bin\mysqld.exe  
#########################################################

3. 添加环境变量

    操作如下：  
    1）右键单击我的电脑->属性->高级系统设置(高级)->环境变量  
      点击系统变量下的新建按钮  
      输入变量名：MYSQL\_HOME  
      输入变量值：D:\Program Files\mysql-5.6.11-winx64  
      #即为mysql的自定义解压目录。  
    2）选择系统变量中的Path  
      点击编辑按钮  
      在变量值中添加变量值：%MYSQL\_HOME%\bin  
      注意是在原有变量值后面加上这个变量，用;隔开，不能删除原来的变量值，

4. 将mysql注册为windows系统服务

    1）从控制台进入到MySQL解压目录下的 bin 目录下：  
    2）输入服务安装命令：  
    mysqld install MySQL --defaults-file="D:\Program Files\MySQL\mysql-5.6.13-win32\my.ini"  
    #解压目录下修改的my.ini文件  
    安装成功后会提示服务安装成功。  
    #注：my.ini文件放在MySQL解压后的根目录下  
    #移除服务命令为：mysqld remove

5. 启动MySQL服务

    方法一：  
        启动服务命令为：net start mysql  
    方法二：  
        打开管理工具 服务，找到MySQL服务。  
        通过右键选择启动或者直接点击左边的启动来启动服务。

6. 修改 root 账号的密码

    刚安装完成时root账号默认密码为空，此时可以将密码修改为指定的密码。如：123456  
    c:>mysql –uroot  
    mysql>show databases;  
    mysql>use mysql;  
    mysql>UPDATE user SET password=PASSWORD("123456") WHERE user='root';  
    mysql>FLUSH PRIVILEGES;  
    mysql>QUIT  
   
7. MySQL控制台快捷方式建立:  
    1）桌面右键->新建->快捷方式->对象位置输入：C:\Windows\System32\cmd.exe  
        快捷方式名称自己定义，确定，快捷方式建立成功  
    2）右键单击刚才建立的快捷方式->属性->把目标一栏修改成MySQL启动参数：  
        C:\Windows\System32\cmd.exe "D:\Program Files\MySQL\mysql-5.6.13-win32\bin" /k mysql -uroot -p inventory  
        解释:CMD路径 "MySQL路径bin目录" /k mysql -u用户名 -p密码 数据库名  
    3）修改完成后点击确定保存，直接双击快捷方式即可连接到MySQL数据库

## mysql join操作 - ggjucheng - 博客园

http://www.cnblogs.com/ggjucheng/archive/2012/11/06/2757972.html

**join的类型**

1．  内联结：将两个表中存在联结关系的字段符合联结关系的那些记录形成记录集的联结。

2．  外联结：分为外左联结和外右联结。

**案例背景**

create table java (name varchar(**255**));

insert into java values ('java1'),('java2'),('blue');

create table mysql (name varchar(**255**));

insert into mysql values ('mysql1'),('mysql2'),('blue');

**内联结**

select \* from java,mysql where java.name=mysql.name;

SELECT \* FROM java JOIN mysql ON java.name=mysql.name;

SELECT \* FROM java INNER JOIN mysql ON java.name=mysql.name;

SELECT \* FROM java CROSS JOIN mysql ON java.name=mysql.name;

SELECT \* FROM java STRAIGHT\_JOIN mysql ON java.name=mysql.name;

这四个语句都是内联结，返回结果都是

+------+------+

| name | name |

+------+------+

| blue | blue |

+------+------+

* table\_reference条目中的每个逗号都看作相当于一个内部联合
* 默认的JOIN都是INNER JOIN
* CROSS JOIN从语法上说与INNER JOIN等同
* STRAIGHT\_JOIN与JOIN相同。除了有一点不一样，左表会在右表之前被读取。STRAIGH\_JOIN可以被用于这样的情况，即联合优化符以错误的顺序排列表。

内联结的语法如下:

join\_table:

table\_reference [INNER | CROSS] JOIN table\_factor [join\_condition]

| table\_reference STRAIGHT\_JOIN table\_factor

| table\_reference STRAIGHT\_JOIN table\_factor ON condition

**外联结**

左联结

SELECT \* FROM java LEFT JOIN mysql ON java.name=mysql.name;

结果是

[复制代码](javascript:void(0);)

+-------+------+

| name | name |

+-------+------+

| java1 | NULL |

| java2 | NULL |

| blue | blue |

+-------+------+

[复制代码](javascript:void(0);)

所以从上面结果看出，因为java表中的java1和java2记录的name没有在mysql表中有对应name，因此为空，但java 所有的列栏仍有java1和java2记录，mysql表所有列栏为NULL。而剩下的blue的那条记录就是java表和mysql表内连接的结果。

如果对于在LEFT JOIN中的ON或USING部分中的右表没有匹配的记录，则所有列被设置为NULL的一个行被用于右表。如果一个表在其它表中没有对应部分，您可以使用这种方法在这种表中查找记录：

SELECT \* FROM java LEFT JOIN mysql ON java.name=mysql.name WHERE mysql.name IS NULL;

该sql找出在java的人，但是不在mysql里的人，这里明显是人员'java1'和'java2'符合要求。

右联结

SELECT \* FROM java RIGHT JOIN mysql ON java.name=mysql.name;

返回结果是

[复制代码](javascript:void(0);)

+------+--------+

| name | name |

+------+--------+

| NULL | mysql1 |

| NULL | mysql2 |

| blue | blue |

+------+--------+

[复制代码](javascript:void(0);)

右联结和左联接结果类似，只是这次是mysql表保存所有的结果集。

外联结的语法

join\_table:| table\_reference LEFT [OUTER] JOIN table\_reference join\_condition

| table\_reference NATURAL [LEFT [OUTER]] JOIN table\_factor

| table\_reference RIGHT [OUTER] JOIN table\_reference join\_condition

| table\_reference NATURAL [RIGHT [OUTER]] JOIN table\_factor

**USING(**column\_list**)子句**

用于为一系列的列进行命名，这些列必须同时在两个表中存在

SELECT java.\*,mysql.\* FROM java LEFT JOIN mysql USING (name);

结果返回

[复制代码](javascript:void(0);)

+-------+------+

| name | name |

+-------+------+

| java1 | NULL |

| java2 | NULL |

| blue | blue |

+-------+------+

[复制代码](javascript:void(0);)

**联结的运算顺序**

SELECT \* FROM t1 LEFT JOIN (t2, t3, t4) ON (t2.a=t1.a AND t3.b=t1.b AND t4.c=t1.c);

--相当于

SELECT \* FROM t1 LEFT JOIN (t2 CROSS JOIN t3 CROSS JOIN t4) ON (t2.a=t1.a AND t3.b=t1.b AND t4.c=t1.c)

括号对于join的顺序的影响

SELECT t1.id,t2.id,t3.id FROM t1,t2 LEFT JOIN t3 ON (t3.id=t1.id) WHERE t1.id=t2.id;

--实际上这么执行

SELECT t1.id,t2.id,t3.id FROM t1,( t2 LEFT JOIN t3 ON (t3.id=t1.id) ) WHERE t1.id=t2.id;

--应该这么写

SELECT t1.id,t2.id,t3.id FROM (t1,t2) LEFT JOIN t3 ON (t3.id=t1.id) WHERE t1.id=t2.id;

在这里括号是相当重要的，因此以后在写这样的查询的时候我们不要忘记了多写几个括号，至少这样能避免很多错误

## Windows 的 MySQL 5.6 的配置文件 my.ini 在哪里？ - 技术文章 - 爱看

http://2goo.info/article/detail/939

Windows 的 MySQL 5.6 的配置文件 my.ini 在哪里？找了很久还是没有找到，后来看了下windows 的启动项，发现mysql 启动的指令是：

"C:\Program Files\MySQL\MySQL Server 5.6\bin\mysqld.exe" --defaults-file="C:\ProgramData\MySQL\MySQL Server 5.6\my.ini" MySQL

仔细看了下，原来在 C:\ProgramData\MySQL\MySQL Server 5.6\my.ini 下，所以修改这个配置文件 才会有效果，如果你复制 my-default.ini 成 my.ini文件，增加一些 设置，重启后，应该不会生效。

C:\ProgramData\MySQL\MySQL Server 5.6\my.ini

## 裸设备与文件系统IO性能比较 - yuanboitliuyuan的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/yuanboitliuyuan/article/details/6887317

裸设备，也叫裸分区（原始分区），是一种没有经过格式化，不被Unix通过文件系统来读取的特殊字符设备。本文收集裸设备和Oracle问答20例。  
　　1．什么叫做裸设备？  
  
       裸设备，也叫裸分区（原始分区），是一种没有经过格式化，不被Unix通过文件系统来读取的特殊字符设备。它由应用程序负责对它进行读写操作。不经过文件系统的缓冲。  
  
　　2．如何辨别裸设备？  
  
　　在Unix的/dev 目录下，有许多文件，其中有两个大类：字符设备文件和块设备文件。  
  
　　字符设备特殊文件进行I/O操作不经过操作系统的缓冲区，而块设备特殊文件用来同外设进行定长的包传输。字符特殊文件与外设进行I/o操作时每次只传输一个字符。而对于块设备特殊文件来说，它用了cache机制，在外设和内存之间一次可以传送一整块数据。裸设备使用字符特殊文件。在/dev 目录下，你可以看到许多这样的文件。  
  
　　3．使用裸设备的好处  
  
　　因为使用裸设备避免了再经过Unix操作系统这一层，数据直接从Disk到Oracle进行传输，所以使用裸设备对于读写频繁的数据库应用来说，可以极大地提高数据库系统的性能。当然，这是以磁盘的 I/O 非常大，磁盘I/O已经称为系统瓶颈的情况下才成立。如果磁盘读写确实非常频繁，以至于磁盘读写成为系统瓶颈的情况成立，那么采用裸设备确实可以大大提高性能，最大甚至可以提高至40％，非常明显。  
  
　　而且，由于使用的是原始分区，没有采用文件系统的管理方式，对于Unix维护文件系统的开销也都没有了，比如不用再维护I-node，空闲块等，这也能够导致性能的提高。  
  
　　4．如何决定是否应该使用裸设备？  
  
　　判断是否使用裸设备要从以下方面进行考虑：首先，数据库系统本身需要已经被比较好的经过了优化。优化是一门很有些技术的话题，很难简单地讲述。其次，使用Unix命令来辨别是否存在磁盘读写瓶颈。比如Unix的vmstat, sar 等命令都可以较好的进行鉴别。如果决定采用裸设备，需要磁盘上还有空闲的分区。否则，就要新添磁盘，或者对原有系统重新规划。  
  
　　5．什么系统必须使用裸设备？  
  
　　如果使用了Oracle 并行服务器选项，则必须采用裸设备来存放所有的数据文件，控制文件，重做日志文件。只有把这些文件放到裸设备上，才能保证所有Oracle 实例都可以读取这个数据库的文件。这是由Unix操作系统的特性决定的。  
  
　　还有一种情况是，如果你想使用异步I/O，那么在有些Unix上也必须采用裸设备。这个需要参考具体Unix的相关文档。  
  
6．能够使用一个磁盘的第一个分区作为裸设备吗？  
  
　　可以，但是不推荐。在Unix的比较旧的版本是银行，磁盘的第一个分区常常包含这个磁盘的一些信息，以及逻辑卷的一些控制信息。若这些部分被裸设备覆盖的话，磁盘就会变得不可识别，导致系统崩溃。  
  
　　较新的Unix版本不会发生这样的情况，因为它们采用了更复杂的技术来管理磁盘，逻辑卷的一些信息。  
  
　　但是，除非很确信不要使用磁盘的第一个分区来作为裸设备。  
  
　　7．我可以把整个裸设备都作为Oracle的数据文件吗？  
  
　　不行。必须让数据文件的大小稍微小于该裸设备的实际大小。至少要空出两个oracle块的大小来。  
  
　　8．裸设备应该属于那个用户？  
  
　　应该由root来创建裸设备，然后再分配给Oracle用户以供使用。同时还要把它归入Oracle用户所在的那个组里边（通常都是DBA）。  
  
　　9．在创建数据文件时如何指定裸设备？  
  
　　和普通文件没有什么太大的区别，一样都是在单引号里边写上裸设备的详细路径就可以了。举一个例子：要在创建一个表空间，使用两个裸设备，每个分别为30M的大小，Oracle块的大小为4K，可以用下面的命令：  
　　CREATE TABLESPACE RAW\_TS  
　　DATAFILE '/dev/raw1' size 30712k  
　　DATAFILE '/dev/raw2' size 30712k;  
  
　　10.Oracle块的大小和裸设备有什么关系吗？  
  
　　Oracle会必须是裸设备上物理块大小的倍数。  
  
　　11．如何在裸设备上进行备份？  
  
　　在裸设备上，不能使用Unix实用程序来进行备份，唯一的办法是使用最基本的Unix命令：DD来进行备份。比如：dd if=/dev/raw1 of=/dev/rmt0 bs=16k。dd的具体语法可以参考unix手册，或者联机帮助。你也可以先用dd把裸设备上的数据文件备份到磁盘上，然后再利用Unix实用程序进一步处理。  
  
　　12．如果我没有使用Oracle并行服务器选项，我可以在数据库上让一部分数据文件使用文件系统，另一部分使用裸设备吗？  
  
　　可以。但是这样的话，会使备份过程更加复杂。  
  
13．我应该把联机重做日志文件放到裸设备上吗？  
  
　　这是一个极好的选择。联机重做日志文件是写操作非常频繁的文件，放到裸设备上非常合适。如果你使用了并行服务器选项，那么联机重做日志文件必须放到裸设备上面。  
  
　　14．可以把归档日志文件放到裸设备上吗？  
  
　　不行。归档日志文件必须放到常规的Unix文件系统上面，或者直接放到磁带上面去。  
  
　　15．我可以在裸设备上边放置多个数据文件吗？  
  
　　不行。所以你必须在设置裸设备时非常小心。太小的话，会导致空间很快用完，太大的话，空间就白白浪费了。  
  
　　16．因应该把几个裸设备放到同一个物理磁盘上吗？  
  
　　这样做不好。因为使用裸设备就是为了提高磁盘读写速度。而把多个裸设备放到同一个物理磁盘上会导致读写竞争，这样对于提高I/O速度是不利的。应该尽量分散裸设备到不同的物理磁盘上，最好是分散到不同的磁盘控制器上。这是最佳选择。  
  
　　17．需要把所有裸设备都定义成同样的大小吗？  
  
　　这不是必须得，但是划分成同样的大小对于管理数据库比较有利。  
  
　　18．为了在Unix上使用裸设备，我需要改变Unix核心参数吗？  
  
　　不需要。但可以选择减小缓冲区的大小，如果没有别的应用也在同一台Unix机器上运行。因为运用了裸设备以后，不再使用Unix的系统缓冲区。  
  
　　19．为了提高读写速度，在操作系统级别上，还有什么办法可以采取吗？  
  
　　使用RAID（廉价冗余磁盘阵列）也是非常有效的办法，尤其实那种读写非常频繁的系统。  
  
　　20．在考虑了以上所有方面后，还能有什么办法可以提高性能的吗？  
  
　　这就需要对Oracle 进行优化，并且购买更多的磁盘和磁盘控制器，来分散I/O到不同的磁盘上。

## 文件系统与裸设备性能比较(2) - yuanboitliuyuan的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/yuanboitliuyuan/article/details/6889853

现在有一种经典的说法是oracle安装的祼设备上面比安装在文件系统上面要快。

是不是任何情况下都是这样呢？

以下是我的一点个人分析，如有不对之处请指出。

目前一般的文件系统都是有日志的。为了尽量减少文件系统的不一致性，缩短操作系统的启动时间，文件系统需追踪引起系统改变的记录，这些记录存放在与文件系统相分离的地方，通常我们叫“日志”。一旦这些日志记录被安全地写入，日志文件系统就可以应用它们清除引起系统改变的记录。这个日志的作用就像oracle中redo的作用一样。

数据库建在文件系统上面主要有以下几点需要考虑：

1.日志维护的开销。这点要根据文件系统具体的日志模式，像ext3就有三种日志模式，为了安全起见，这个日志维护的开销将不可忽视。

2.OS维护和管理文件系统的开销。在数据库的情况下，用户一般都是在现存文件的中间写入数据，甚至覆盖现存文件。这种操作一般是先截断该文件，然后再写入数据。每次都会涉及到文件的改变，因此这部分的管理开销也将不小。

3.采用文件系统，就会涉及到文件系统的缓存(pagecache)。有这个文件系统的缓存，就会导致写操作比写祼设备的速度要慢。因为当发生一个IO，将oracle buffer中的块写入硬盘时，一般都是先将这个块写到pagecache中，然后再从pagecache flush到硬盘上。而oracle为了保证确保不会有数据丢失必须等这个块从pagecache flush到硬盘的操作完成后才会收到一个写入完成的消息。所以说采用文件系统，写操作比直接写到祼设备的操作要慢。

4.采用文件系统，因为有文件系统缓存，读操作一般会比采用祼设备要快。这点不仔细描述了。我想大多数人都能接受这一点。

根据上面4点，我觉的，如果一个系统在系统资源(包括CPU,内存，IO)不存为瓶颈时，同时读操作又多于写操作时，一般情况下采用文件系统会比采用祼设备要快。

如果能够预估到系统资源可能会成为瓶颈，或者写操作又较多时，应该考虑采用祼设备。

## 理解磁盘条带化\_polygun2000\_新浪博客

http://blog.sina.com.cn/s/blog\_704836f40100sbsl.html

**什么是条带化(striping)**

    当多个进程同时访问一个磁盘时，可能会出现磁盘冲突。大多数磁盘系统都对访问次数（每秒的 I/O 操作，IOPS）和数据传输率（每秒传输的数据量，TPS）有限制。当达到这些限制时，后面需要访问磁盘的进程就需要等待，这时就是所谓的磁盘冲突。

    避免磁盘冲突是优化 I/O 性能的一个重要目标，而 I/O 性能的优化与其他资源（如CPU和内存）的优化有着很大的区别 ,I/O 优化最有效的手段是将 I/O 最大限度的进行平衡。

    条带化技术就是一种自动的将 I/O 的负载均衡到多个物理磁盘上的技术，条带化技术就是将一块连续的数据分成很多小部分并把他们分别[存储](http://www.itvue.com/Article/NETWORKSUPPORT/Backup/Index.html)到不同磁盘上去。这就能使多个进程同时访问数据的多个不同部分而不会造成磁盘冲突，而且在需要对这种数据进行顺序访问的时候可以获得最大程度上的 I/O 并行能力，从而获得非常好的性能。很多操作系统、磁盘设备供应商、各种第三方软件都能做到条带化。

    图 1 描述的是一个未经条带化处理的连续数据的分布，图 2 描述的是一个已经被条带化处理的连续数据的分布，从中比较，我们可以发现图 2 中对连续数据的读写都有最大的并发能力。

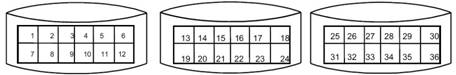
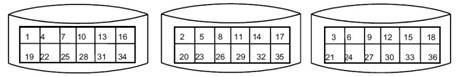
图 1. 未经条带化处理的连续数据  
[](http://photo.blog.sina.com.cn/showpic.html#blogid=704836f40100sbsl&url=http://s7.sinaimg.cn/orignal/704836f4ga6b2238658b6)

图 2. 已经被条带化处理的连续数据

[](http://photo.blog.sina.com.cn/showpic.html#blogid=704836f40100sbsl&url=http://s8.sinaimg.cn/orignal/704836f4ga6b223865797)

    由于条带化在 I/O 性能问题上的优越表现，以致于在应用系统所在的计算环境中的多个层次或平台都涉及到了条带化的技术，如操作系统和[存储](http://www.itvue.com/Article/NETWORKSUPPORT/Backup/Index.html)系统这两个层次中都可能使用条带化技术。

**影响条带化效果的两个因素**

    当对数据做条带化时，数据被切成一块块的小数据块，各小数据块分布存储在不同的硬盘上。从这个描述中我们可以看出，影响条带化效果的因素有两个，一是条带大小（stripe size），即数据被切成的小数据块的大小，另一个条带宽度（stripe width），即数据被存储到多少块硬盘上。

    条带宽度（stripe width）是指同时可以并发读或写的条带数量。这个数量等于RAID中的物理硬盘数量。例如一个经过条带化的，具有4块物理硬盘的阵列的条带宽度就是4。增加条带宽度，可以增加阵列的读写性能。道理很明显，增加更多的硬盘，也就增加了可以同时并发读或写的条带数量。在其他条件一样的前提下，一个由8块18G硬盘组成的阵列相比一个由4块36G硬盘组成的阵列具有更高的传输性能。

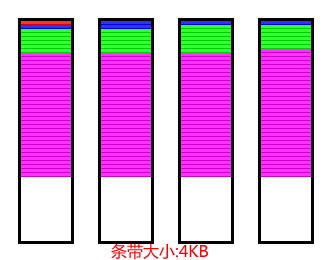
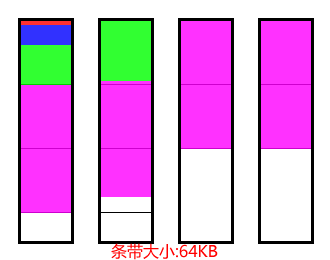
    条带大小（stripe size）,有时也被叫做*block size*, *chunk size*, *stripe length* 或者 *granularity*。这个参数指的是写在每块磁盘上的条带数据块的大小。RAID的数据块大小一般在2KB到512KB之间(或者更大)，其数值是2的次方，即2KB,4KB,8KB,16KB这样。

    条带大小对性能的影响比条带宽度难以量化的多。

    ·减小条带大小: 由于条带大小减小了，则文件被分成了更多个，更小的数据块。这些数据块会被分散到更多的硬盘上存储，因此提高了传输的性能，但是由于要多次寻找不同的数据块，磁盘定位的性能就下降了。

    ·增加条带大小: 与减小条带大小相反，会降低传输性能，提高定位性能。

    根据上边的论述，我们会发现根据不同的应用类型，不同的性能需求，不同驱动器的不同特点(如SSD硬盘)，不存在一个普遍适用的"最佳条带大小"。所以这也是存储厂家，文件系统编写者允许我们自己定义条带大小的原因。不同条带大小，对于文件如何存储有很大的影响，请看下边这两幅图的对比:

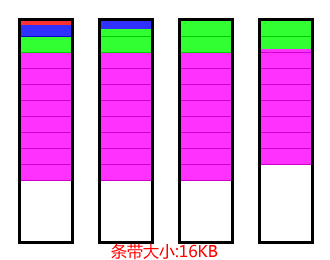
[](http://photo.blog.sina.com.cn/showpic.html#blogid=704836f40100sbsl&url=http://s10.sinaimg.cn/orignal/704836f4ga6b240a508c9)[](http://photo.blog.sina.com.cn/showpic.html#blogid=704836f40100sbsl&url=http://s2.sinaimg.cn/orignal/704836f4ga6b240c385e1)  
    这是一个由4块硬盘组成的RAID0阵列，左边的条带大小为4KB，右边的条带大小为64KB。

左边的图中的每一条细格表示4KB大小。

    图中红色文件大小是4KB，蓝色文件大小20KB，绿色文件大小为100KB，紫色文件大小为500KB。

    从图中我们可以看到，不同条带大小对"中型大小"文件的影响是很大的。对于红色的4KB文件来说，不论条带是4KB还64KB，它都分布在一块硬盘的一个数据块上。而对于紫色的500KB文件来说，无论条带是4KB还是64KB，它都会被分布在四块硬盘上。

    但是对于蓝色20KB的文件来说，如果采用64KB的条带大小，则它就会被分布在一块硬盘上，而不是像4KB条带时那样分布在四块硬盘上。同样绿色的100KB文件在64KB条带时，会被分布到2块硬盘，而4KB条带时则分布到4块硬盘上。可以看到，增加条带大小可以明显地增加定位性能。在上边的例子中，条带宽度理所当然是4。

    下图是使用16KB条带时的文件分布图，可以对应参考理解一下。  
[](http://photo.blog.sina.com.cn/showpic.html#blogid=704836f40100sbsl&url=http://s6.sinaimg.cn/orignal/704836f4ga6b240b44285)  
     那么我们应该选择多大的条带大小呢？最好的办法是尝试不同组合，根据应用的不同得到自己的经验规律。另外，不要过高估计不同条带大小间的性能差异。它有可能会差的很大，尤其是设置成4KB和256KB这样两个相对极端数值的时候，但对于相差不大的数值，它们的性能差异可能就不明显。对于大多数应用来说，可以参考这样的经验法则:大量的小文件读写时，采用较大的条带大小;少量的大文件的快速访问，采用比较小的条带；如果要平衡这两者，那么采用中间值。

参考资料:

<http://www.pcguide.com/ref/hdd/perf/raid/concepts/perfStripe-c.html>

<http://www.itvue.com/Article/DB/DB2/200904/11214.html>

# 行列转换

## mysql数据库中怎样实现多行合并为一行？-CSDN论坛-CSDN.NET-中国最大的IT技术社区

http://bbs.csdn.net/topics/110005890

D RE\_ID  
1 2001  
2 2001  
2 2003  
1 2002  
3 2003   
2 2002  
3 2004  
......  
我希望的到的数据是:  
ID RE\_ID  
1 2001,2002  
2 2001,2002,2003  
3 2003,2004  
原表并为对ID进行排序，数据库规模比较大(百万级记录)。  
求mysql多行合并为一行的语句，效率优先。

select id,group\_concat(re\_id order by re\_id separator ",") as re\_id  
from tablename  
group by id

## mysql 语句case when - john2000 - 博客园

http://www.cnblogs.com/john2000/archive/2010/09/21/1832729.html

表的创建

CREATE TABLE `lee` (  
`id` int(10) NOT NULL AUTO\_INCREMENT,    
`name` char(20) DEFAULT NULL,    
`birthday` datetime DEFAULT NULL,    
PRIMARY KEY (`id`)) ENGINE=InnoDB DEFAULT CHARSET=utf8

数据插入：

insert into lee(name,birthday) values ('sam','1990-01-01');

insert into lee(name,birthday) values ('lee','1980-01-01');

insert into lee(name,birthday) values ('john','1985-01-01');

使用case when语句

1。

select name,  
 case   
        when birthday<'1981' then 'old'  
        when birthday>'1988' then 'yong'  
        else 'ok' END YORN  
from lee;

2。

select NAME,  
 case name  
     when 'sam' then 'yong'  
        when 'lee' then 'handsome'  
        else 'good' end  
from lee;

当然了case when语句还可以复合

3。

select name,birthday,  
 case   
     when birthday>'1983' then 'yong'  
        when name='lee' then 'handsome'  
        else 'just so so ' end  
from lee;

在这里用sql语句进行日期比较的话，需要对年加引号。要不然可能结果可能和预期的结果会不同。我的mysql版本5.1

当然也可以用year函数来实现，以第一个sql为例

select NAME,  
 CASE  
     when year(birthday)>1988 then 'yong'  
        when year(birthday)<1980 then 'old'  
        else 'ok' END  
from lee;

﻿

create table penalties  
(  
 paymentno INTEGER not NULL,  
    payment\_date DATE not null,  
    amount DECIMAL(7,2) not null,  
    primary key(paymentno)  
)

insert into penalties values(1,'2008-01-01',3.45);  
insert into penalties values(2,'2009-01-01',50.45);  
insert into penalties values(3,'2008-07-01',80.45);

1.#对罚款登记分为三类，第一类low，包括大于0小于等于40的罚款，第二类moderate大于40  
#到80之间的罚款，第三类high包含所有大于80的罚款。

2.#统计出属于low的罚款编号。

第一道题的解法与上面的相同  
select paymentno,amount,  
 case   
     when amount>0 and amount<=40 then 'low'  
        when amount>40 and amount<=80 then 'moderate'  
        when amount>80 then 'high'  
        else 'incorrect' end lvl  
from `penalties`

2.#统计出属于low的罚款编号。重点看这里的解决方法  
方法1.  
select paymentno,amount  
from `penalties`  
where case   
 when amount>0 and  amount<=40 then 'low'  
    when amount>40 and amount<=80 then 'moderate'  
    when amount>80 then 'high'  
    else 'incorrect' end ='low';

方法2  
select \*   
from (select paymentno,amount,  
 case   
     when amount>0 and amount<=40 then 'low'  
        when amount>40 and amount<=80 then 'moderate'  
        when amount>80 then 'high'  
        else 'incorrect' end lvl  
from `penalties`) as p  
where p.lvl='low';

## mysql行列转换 - - 博客频道 - CSDN.NET

http://blog.csdn.net/u012501459/article/details/12945241

首先介绍与行列转换相关的函数和一些基本知识：

* IF()函数

语法：IF(expr1,expr2,expr3)

如果 expr1 是TRUE (expr1 <> 0 and expr1 <> NULL,<>表示不等于)，则 IF()的返回值为expr2; 否则返回值则为 expr3。IF() 的返回值为数字值或字符串值，具体情况视其所在语境而定。

mysql> SELECT IF(1>2,2,3);

-> 3

mysql> SELECT IF(1<2,'yes ','no');

-> 'yes'

mysql> SELECT IF(STRCMP('test','test1'),'no','yes');

-> 'no'

如果expr2 或expr3中只有一个明确是 NULL，则IF() 函数的结果类型为非NULL表达式的结果类型。

expr1 作为一个整数值进行计算，就是说，假如你正在验证浮点值或字符串值，那么应该使用比较运算进行检验。

mysql> SELECT IF(0.1,1,0);

-> 0

mysql> SELECT IF(0.1<>0,1,0);

-> 1

在所示的第一个例子中，IF(0.1)的返回值为0，原因是 0.1 被转化为整数值，从而引起一个对 IF(0)的检验。这或许不是你想要的情况。在第二个例子中，比较检验了原始浮点值，目的是为了了解是否其为非零值。比较结果使用整数。

IF() (这一点在其被储存到临时表时很重要 ) 的默认返回值类型按照以下方式计算：

|  |  |
| --- | --- |
| 表达式 | 返回值 |
| expr2 或expr3 返回值为一个字符串。 | 字符串 |
| expr2 或expr3 返回值为一个浮点值。 | 浮点 |
| expr2 或 expr3 返回值为一个整数。 | 整数 |

假如expr2 和expr3 都是字符串，且其中任何一个字符串区分大小写，则返回结果是区分大小写。

* IFNULL函数

语法：IFNULL(expr1,expr2)

假如expr1 不为 NULL，则 IFNULL() 的返回值为 expr1; 否则其返回值为 expr2。IFNULL()的返回值是数字或是字符串，具体情况取决于其所使用的语境。

mysql> SELECT IFNULL(1,0);

-> 1

mysql> SELECT IFNULL(NULL,10);

-> 10

mysql> SELECT IFNULL(1/0,10);

-> 10

mysql> SELECT IFNULL(1/0,'yes');

-> 'yes'

* CONCAT函数

语法：CONCAT(str1,str2,...)

返回结果为连接参数产生的字符串。如有任何一个参数为NULL ，则返回值为 NULL。或许有一个或多个参数。 如果所有参数均为非二进制字符串，则结果为非二进制字符串。 如果自变量中含有任一二进制字符串，则结果为一个二进制字符串。一个数字参数被转化为与之相等的二进制字符串格式；若要避免这种情况，可使用显式类型 cast, 例如： SELECT CONCAT(CAST(int\_col AS CHAR), char\_col)

mysql> SELECT CONCAT('My', 'S', 'QL');

-> 'MySQL'

mysql> SELECT CONCAT('My', NULL, 'QL');

-> NULL

mysql> SELECT CONCAT(14.3);

-> '14.3'

* with rollup

GROUP BY子句允许一个将额外行添加到简略输出端 WITH ROLLUP 修饰符。这些行代表高层(或高聚集)简略操作。ROLLUP 因而允许你在多层分析的角度回答有关问询的问题。例如，它可以用来向OLAP (联机分析处理) 操作提供支持。

设想一个名为sales 的表具有年份、国家、产品及记录销售利润的利润列：

CREATE TABLE sales

(

year INT NOT NULL,

country VARCHAR(20) NOT NULL,

product VARCHAR(32) NOT NULL,

profit INT

);

可以使用这样的简单GROUP BY，每年对表的内容做一次总结：

mysql> SELECT year, SUM(profit) FROM sales GROUP BY year;

+------+-------------+

| year | SUM(profit) |

+------+-------------+

| 2000 | 4525 |

| 2001 | 3010 |

+------+-------------+

这个输出结果显示了每年的总利润， 但如果你也想确定所有年份的总利润，你必须自己累加每年的单个值或运行一个加法询问。

或者你可以使用 ROLLUP, 它能用一个问询提供双层分析。将一个 WITH ROLLUP修饰符添加到GROUP BY 语句，使询问产生另一行结果，该行显示了所有年份的总价值：

mysql> SELECT year, SUM(profit) FROM sales GROUP BY year WITH ROLLUP;

+------+-------------+

| year | SUM(profit) |

+------+-------------+

| 2000 | 4525 |

| 2001 | 3010 |

| NULL | 7535 |

+------+-------------+

总计高聚集行被年份列中的NULL值标出。

当有多重 GROUP BY 列时，ROLLUP产生的效果更加复杂。这时，每次在除了最后一个分类列之外的任何列出现一个 “break” (值的改变) ，则问讯会产生一个高聚集累计行。

例如，在没有 ROLLUP的情况下，一个以年、国家和产品为基础的关于 sales 表的一览表可能如下所示：

mysql> SELECT year, country, product, SUM(profit)

-> FROM sales

-> GROUP BY year, country, product;

+------+---------+------------+-------------+

| year | country | product | SUM(profit) |

+------+---------+------------+-------------+

| 2000 | Finland | Computer | 1500 |

| 2000 | Finland | Phone | 100 |

| 2000 | India | Calculator | 150 |

| 2000 | India | Computer | 1200 |

| 2000 | USA | Calculator | 75 |

| 2000 | USA | Computer | 1500 |

| 2001 | Finland | Phone | 10 |

| 2001 | USA | Calculator | 50 |

| 2001 | USA | Computer | 2700 |

| 2001 | USA | TV | 250 |

+------+---------+------------+-------------+

表示总值的输出结果仅位于年/国家/产品的分析级别。当添加了 ROLLUP后， 问询会产生一些额外的行：

mysql> SELECT year, country, product, SUM(profit)

-> FROM sales

-> GROUP BY year, country, product WITH ROLLUP;

+------+---------+------------+-------------+

| year | country | product | SUM(profit) |

+------+---------+------------+-------------+

| 2000 | Finland | Computer | 1500 |

| 2000 | Finland | Phone | 100 |

| 2000 | Finland | NULL | 1600 |

| 2000 | India | Calculator | 150 |

| 2000 | India | Computer | 1200 |

| 2000 | India | NULL | 1350 |

| 2000 | USA | Calculator | 75 |

| 2000 | USA | Computer | 1500 |

| 2000 | USA | NULL | 1575 |

| 2000 | NULL | NULL | 4525 |

| 2001 | Finland | Phone | 10 |

| 2001 | Finland | NULL | 10 |

| 2001 | USA | Calculator | 50 |

| 2001 | USA | Computer | 2700 |

| 2001 | USA | TV | 250 |

| 2001 | USA | NULL | 3000 |

| 2001 | NULL | NULL | 3010 |

| NULL | NULL | NULL | 7535 |

+------+---------+------------+-------------+

对于这个问询， 添加ROLLUP 子句使村输出结果包含了四层分析的简略信息，而不只是一个下面是怎样解释ROLLUP输出：

* 一组给定的年份和国家的每组产品行后面, 会产生一个额外的总计行，显示所有产品的总值。这些行将产品列设置为 NULL。
* 一组给定年份的行后面，会产生一个额外的总计行，显示所有国家和产品的总值。这些行将国家和产品列设置为 NULL。
* 最后, 在所有其它行后面，会产生一个额外的总计列，显示所有年份、国家及产品的总值。这一行将年份、国家和产品列设置为 NULL。

* union

SELECT ...

UNION [ALL | DISTINCT]

SELECT ...

[UNION [ALL | DISTINCT]

SELECT ...]

UNION用于把来自许多SELECT语句的结果组合到一个结果集合中。

列于每个SELECT语句的对应位置的被选择的列应具有相同的类型。（例如，被第一个语句选择的第一列应和被其它语句选择的第一列具有相同的类型。）在第一个SELECT语句中被使用的列名称也被用于结果的列名称。

SELECT语句为常规的选择语句，但是受到如下的限定：

·只有最后一个SELECT语句可以使用INTO OUTFILE。

· HIGH\_PRIORITY不能与作为UNION一部分的SELECT语句同时使用。如果您对第一个SELECT指定了HIGH\_PRIORITY，则不会起作用。如果您对其它后续的SELECT语句指定了HIGH\_PRIORITY，则会产生语法错误。

如果您对UNION不使用关键词ALL，则所有返回的行都是唯一的，如同您已经对整个结果集合使用了DISTINCT。如果您指定了ALL，您会从所有用过的SELECT语句中得到所有匹配的行。

DISTINCT关键词是一个自选词，不起任何作用，但是根据SQL标准的要求，在语法中允许采用。（在MySQL中，DISTINCT代表一个共用体的默认工作性质。）

您可以在同一查询中混合UNION ALL和UNION DISTINCT。被混合的UNION类型按照这样的方式对待，即DISTICT共用体覆盖位于其左边的所有ALL共用体。DISTINCT共用体可以使用UNION DISTINCT明确地生成，或使用UNION（后面不加DISTINCT或ALL关键词）隐含地生成。

如果您想使用ORDER BY或LIMIT子句来对全部UNION结果进行分类或限制，则应对单个地SELECT语句加圆括号，并把ORDER BY或LIMIT放到最后一个的后面。以下例子同时使用了这两个子句：

(SELECT a FROM tbl\_name WHERE a=10 AND B=1)

UNION

(SELECT a FROM tbl\_name WHERE a=11 AND B=2)

ORDER BY a LIMIT 10;

这种ORDER BY不能使用包括表名称（也就是，采用tbl\_name.col\_name格式的名称）列引用。可以在第一个SELECT语句中提供一个列别名，并在ORDER BY中参阅别名，或使用列位置在ORDER BY中参阅列。（首选采用别名，因为不建议使用列位置。）

另外，如果带分类的一列有别名，则ORDER BY子句必须引用别名，而不能引用列名称。以下语句中的第一个语句必须运行，但是第二个会运行失败，出现在'order clause'中有未知列'a'的错误：

(SELECT a AS b FROM t) UNION (SELECT ...) ORDER BY b;

(SELECT a AS b FROM t) UNION (SELECT ...) ORDER BY a;

To apply ORDER BY or LIMIT to an individual SELECT, place the clause inside the parentheses that enclose the SELECT: 为了对单个SELECT使用ORDER BY或LIMIT，应把子句放入圆括号中。圆括号包含了SELECT：

(SELECT a FROM tbl\_name WHERE a=10 AND B=1 ORDER BY a LIMIT 10)

UNION

(SELECT a FROM tbl\_name WHERE a=11 AND B=2 ORDER BY a LIMIT 10);

圆括号中用于单个SELECT语句的ORDER BY只有当与LIMIT结合后，才起作用。否则，ORDER BY被优化去除。

* 用于预处理的sql语句

预制语句的SQL语法基于三个SQL语句：

PREPARE stmt\_name FROM preparable\_stmt;

EXECUTE stmt\_name [USING @var\_name [, @var\_name] ...];

{DEALLOCATE | DROP} PREPARE stmt\_name;

PREPARE语句用于预备一个语句，并赋予它名称stmt\_name，借此在以后引用该语句。语句名称对案例不敏感。preparable\_stmt可以是一个文字字符串，也可以是一个包含了语句文本的用户变量。该文本必须展现一个单一的SQL语句，而不是多个语句。使用本语句，‘?’字符可以被用于制作参数，以指示当您执行查询时，数据值在哪里与查询结合在一起。‘?’字符不应加引号，即使您想要把它们与字符串值结合在一起，也不要加引号。参数制作符只能被用于数据值应该出现的地方，不用于SQL关键词和标识符等。

如果带有此名称的预制语句已经存在，则在新的语言被预备以前，它会被隐含地解除分配。这意味着，如果新语句包含一个错误并且不能被预备，则会返回一个错误，并且不存在带有给定名称语句。

预制语句的范围是客户端会话。在此会话内，语句被创建。其它客户端看不到它。

在预备了一个语句后，您可使用一个EXECUTE语句（该语句引用了预制语句名称）来执行它。如果预制语句包含任何参数制造符，则您必须提供一个列举了用户变量（其中包含要与参数结合的值）的USING子句。参数值只能有用户变量提供，USING子句必须准确地指明用户变量。用户变量的数目与语句中的参数制造符的数量一样多。

您可以多次执行一个给定的预制语句，在每次执行前，把不同的变量传递给它，或把变量设置为不同的值。

要对一个预制语句解除分配，需使用DEALLOCATE PREPARE语句。尝试在解除分配后执行一个预制语句会导致错误。

如果您终止了一个客户端会话，同时没有对以前已预制的语句解除分配，则服务器会自动解除分配。

以下SQL语句可以被用在预制语句中：CREATE TABLE, DELETE, DO, INSERT, REPLACE, SELECT, SET, UPDATE和多数的SHOW语句。目前不支持其它语句。

以下例子显示了预备一个语句的两种方法。该语句用于在给定了两个边的长度时，计算三角形的斜边。

第一个例子显示如何通过使用文字字符串来创建一个预制语句，以提供语句的文本：

mysql> PREPARE stmt1 FROM 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';

mysql> SET @a = 3;

mysql> SET @b = 4;

mysql> EXECUTE stmt1 USING @a, @b;

+------------+

| hypotenuse |

+------------+

| 5 |

+------------+

mysql> DEALLOCATE PREPARE stmt1;

第二个例子是相似的，不同的是提供了语句的文本，作为一个用户变量：

mysql> SET @s = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';

mysql> PREPARE stmt2 FROM @s;

mysql> SET @a = 6;

mysql> SET @b = 8;

mysql> EXECUTE stmt2 USING @a, @b;

+------------+

| hypotenuse |

+------------+

| 10 |

+------------+

mysql> DEALLOCATE PREPARE stmt2;

预制语句的SQL语法不能被用于带嵌套的风格中。也就是说，被传递给PREPARE的语句本身不能是一个PREPARE, EXECUTE或DEALLOCATE PREPARE语句。

预制语句的SQL语法与使用预制语句API调用不同。例如，您不能使用mysql\_stmt\_prepare() C API函数来预备一个PREPARE, EXECUTE或DEALLOCATE PREPARE语句。

预制语句的SQL语法可以在已存储的过程中使用，但是不能在已存储的函数或触发程序中使用。

* 准备知识足够了，下面介绍如何进行行列转换

创建数据库（这个数据库是网上的）

create table tx(  
id int primary key,  
c1 char(2),  
c2 char(2),  
c3 int  
);  
insert into tx values  
(1 ,'A1','B1',9),  
(2 ,'A2','B1',7),  
(3 ,'A3','B1',4),  
(4 ,'A4','B1',2),  
(5 ,'A1','B2',2),  
(6 ,'A2','B2',9),  
(7 ,'A3','B2',8),  
(8 ,'A4','B2',5),  
(9 ,'A1','B3',1),  
(10 ,'A2','B3',8),  
(11 ,'A3','B3',8),  
(12 ,'A4','B3',6),  
(13 ,'A1','B4',8),  
(14 ,'A2','B4',2),  
(15 ,'A3','B4',6),  
(16 ,'A4','B4',9),  
(17 ,'A1','B4',3),  
(18 ,'A2','B4',5),  
(19 ,'A3','B4',2),  
(20 ,'A4','B4',5);

mysql> select \* from tx;  
+----+------+------+------+  
| id | c1   | c2   | c3   |  
+----+------+------+------+  
|  1 | A1   | B1   |    9 |  
|  2 | A2   | B1   |    7 |  
|  3 | A3   | B1   |    4 |  
|  4 | A4   | B1   |    2 |  
|  5 | A1   | B2   |    2 |  
|  6 | A2   | B2   |    9 |  
|  7 | A3   | B2   |    8 |  
|  8 | A4   | B2   |    5 |  
|  9 | A1   | B3   |    1 |  
| 10 | A2   | B3   |    8 |  
| 11 | A3   | B3   |    8 |  
| 12 | A4   | B3   |    6 |  
| 13 | A1   | B4   |    8 |  
| 14 | A2   | B4   |    2 |  
| 15 | A3   | B4   |    6 |  
| 16 | A4   | B4   |    9 |  
| 17 | A1   | B4   |    3 |  
| 18 | A2   | B4   |    5 |  
| 19 | A3   | B4   |    2 |  
| 20 | A4   | B4   |    5 |  
+----+------+------+------+  
20 rows in set (0.00 sec)  
mysql>  
期望结果  
+------+-----+-----+-----+-----+------+  
|C1    |B1   |B2   |B3   |B4   |Total |  
+------+-----+-----+-----+-----+------+  
|A1    |9    |2    |1    |11   |23    |  
|A2    |7    |9    |8    |7    |31    |  
|A3    |4    |8    |8    |8    |28    |  
|A4    |2    |5    |6    |14   |27    |  
|Total |22   |24   |23   |40   |109   |  
+------+-----+-----+-----+-----+------+  
1. 利用SUM(IF()) 生成列 + WITH ROLLUP 生成汇总行,并利用 IFNULL将汇总行标题显示为 Total  
mysql>  
SELECT  
IFNULL(c1,'total') AS total,  
SUM(IF(c2='B1',c3,0)) AS B1,  
SUM(IF(c2='B2',c3,0)) AS B2,  
SUM(IF(c2='B3',c3,0)) AS B3,  
SUM(IF(c2='B4',c3,0)) AS B4,  
SUM(IF(c2='total',c3,0)) AS total  
FROM (  
SELECT c1,IFNULL(c2,'total') AS c2,SUM(c3) AS c3  
FROM tx   GROUP BY c1,c2  
WITH ROLLUP   
HAVING c1 IS NOT NULL  
) AS A  
GROUP BY c1  
WITH ROLLUP;  
  
  
  
+-------+------+------+------+------+-------+  
| total | B1   | B2   | B3   | B4   | total |  
+-------+------+------+------+------+-------+  
| A1    |    9 |    2 |    1 |   11 |    23 |  
| A2    |    7 |    9 |    8 |    7 |    31 |  
| A3    |    4 |    8 |    8 |    8 |    28 |  
| A4    |    2 |    5 |    6 |   14 |    27 |  
| total |   22 |   24 |   23 |   40 |   109 |  
+-------+------+------+------+------+-------+  
5 rows in set, 1 warning (0.00 sec)  
2. 利用SUM(IF()) 生成列 + UNION 生成汇总行,并利用 IFNULL将汇总行标题显示为 Total  
mysql>  
select c1,  
sum(if(c2='B1',C3,0)) AS B1,  
sum(if(c2='B2',C3,0)) AS B2,  
sum(if(c2='B3',C3,0)) AS B3,  
sum(if(c2='B4',C3,0)) AS B4,SUM(C3) AS TOTAL  
from tx  
group by C1  
UNION  
SELECT 'TOTAL',sum(if(c2='B1',C3,0)) AS B1,  
sum(if(c2='B2',C3,0)) AS B2,  
sum(if(c2='B3',C3,0)) AS B3,  
sum(if(c2='B4',C3,0)) AS B4,SUM(C3) FROM TX;  
  
+-------+------+------+------+------+-------+  
| c1    | B1   | B2   | B3   | B4   | TOTAL |  
+-------+------+------+------+------+-------+  
| A1    |    9 |    2 |    1 |   11 |    23 |  
| A2    |    7 |    9 |    8 |    7 |    31 |  
| A3    |    4 |    8 |    8 |    8 |    28 |  
| A4    |    2 |    5 |    6 |   14 |    27 |  
| TOTAL |   22 |   24 |   23 |   40 |   109 |  
+-------+------+------+------+------+-------+  
5 rows in set (0.00 sec)  
mysql>  
  
3.  利用SUM(IF()) 生成列,直接生成结果不再利用子查询  
mysql>  
select  
ifnull(c1,'total'),  
sum(if(c2='B1',C3,0)) AS B1,  
sum(if(c2='B2',C3,0)) AS B2,  
sum(if(c2='B3',C3,0)) AS B3,  
sum(if(c2='B4',C3,0)) AS B4,SUM(C3) AS TOTAL  
from tx  
group by C1 with rollup ;  
  
+--------------------+------+------+------+------+-------+  
| ifnull(c1,'total') | B1   | B2   | B3   | B4   | TOTAL |  
+--------------------+------+------+------+------+-------+  
| A1                 |    9 |    2 |    1 |   11 |    23 |  
| A2                 |    7 |    9 |    8 |    7 |    31 |  
| A3                 |    4 |    8 |    8 |    8 |    28 |  
| A4                 |    2 |    5 |    6 |   14 |    27 |  
| total              |   22 |   24 |   23 |   40 |   109 |  
+--------------------+------+------+------+------+-------+  
5 rows in set (0.00 sec)  
mysql>  
  
4. 动态，适用于列不确定情况，  
mysql> SET @EE='';  
mysql> SELECT @EE:=CONCAT(@EE,'SUM(IF(C2=\'',C2,'\'',',C3,0)) AS ',C2,',') FROM (SELECT DISTINCT C2 FROM TX) A;  
  
mysql> SET @QQ=CONCAT('SELECT ifnull(c1,\'total\'),',LEFT(@EE,LENGTH(@EE)-1),' ,SUM(C3) AS TOTAL FROM TX GROUP BY C1 WITH ROLLUP');  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> PREPARE stmt2 FROM @QQ;  
Query OK, 0 rows affected (0.00 sec)  
Statement prepared  
mysql> EXECUTE stmt2;  
+--------------------+------+------+------+------+-------+  
| ifnull(c1,'total') | B1   | B2   | B3   | B4   | TOTAL |  
+--------------------+------+------+------+------+-------+  
| A1                 |    9 |    2 |    1 |   11 |    23 |  
| A2                 |    7 |    9 |    8 |    7 |    31 |  
| A3                 |    4 |    8 |    8 |    8 |    28 |  
| A4                 |    2 |    5 |    6 |   14 |    27 |  
| total              |   22 |   24 |   23 |   40 |   109 |  
+--------------------+------+------+------+------+-------+  
5 rows in set (0.00 sec)

# 日期

# MaxScale

MariaDB

## MaxScale：实现MySQL读写分离与负载均衡的中间件利器 - 推酷

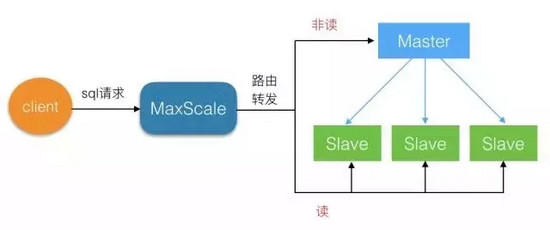
http://www.tuicool.com/articles/fyiuqaY

1

**MaxScale 是干什么的？**

配置好了MySQL的主从复制结构后，我们希望实现读写分离，把读操作分散到从服务器中，并且对多个从服务器能实现负载均衡。

**读写分离和负载均衡**是 MySQL 集群的基础需求，MaxScale 就可以帮着我们方便的实现这些功能。



2

MaxScale 的基础构成

MaxScale  是 MySQL 的兄弟公司 MariaDB 开发的，现在已经发展得非常成熟。MaxScale 是插件式结构，允许用户开发适合自己的插件。

MaxScale  目前提供的插件功能分为5类：

* 认证插件

提供了登录认证功能，MaxScale 会读取并缓存数据库中 user 表中的信息，当有连接进来时，先从缓存信息中进行验证，如果没有此用户，会从后端数据库中更新信息，再次进行验证

* 协议插件

包括客户端连接协议，和连接数据库的协议

* 路由插件

决定如何把客户端的请求转发给后端数据库服务器，读写分离和负载均衡的功能就是由这个模块实现的

* 监控插件

对各个数据库服务器进行监控，例如发现某个数据库服务器响应很慢，那么就不向其转发请求了

* 日志和过滤插件

提供简单的数据库防火墙功能，可以对SQL进行过滤和容错

3

MaxScale 的安装使用

例如有 3 台数据库服务器，是一主二从的结构。

过程概述

（1）配置好集群环境

（2）下载安装 MaxScale

（3）配置 MaxScale，添加各数据库信息

（4）启动 MaxScale，查看是否正确连接数据库

（5）客户端连接 MaxScale，进行测试

详细过程

（1）配置一主二从的集群环境

准备3台服务器，安装 MySQL ，配置一主二从的复制结构。

（2）安装 MaxScale

最好在另一台服务器上安装，如果资源不足，可以和某个 MySQL 放在一起。

MaxScale  的下载地址：

https://downloads.mariadb.com/files/MaxScale

根据自己的服务器选择合适的安装包。

以 centos 7 为例 安装步骤如下：

yum install libaio.x86\_64 libaio-devel.x86\_64 novacom-server.x86\_64 libedit -y

rpm -ivh maxscale-1.4.3-1.centos.7.x86\_64.rpm

（3）配置 MaxScale

在开始配置之前，需要在 master 中为 MaxScale 创建两个用户，用于监控模块和路由模块。

创建监控用户

mysql> create user scalemon@'%' identified by "111111";

mysql> grant replication slave, replication client on \*.\* to scalemon@'%';

创建路由用户

mysql> create user maxscale@'%' identified by "111111";

mysql> grant select on mysql.\* to maxscale@'%';

用户创建完成后，开始配置

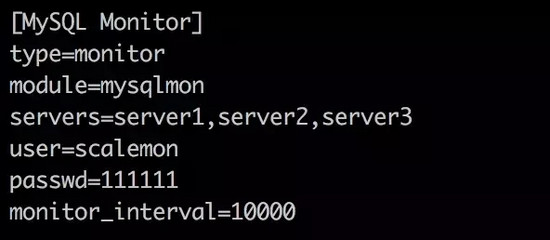
vi /etc/maxscale.cnf

找到 **[server1]**部分，修改其中的 address 和 port，指向 master 的 IP 和端口。

复制2次 [server1] 的整块儿内容，改为 [server2] 与 [server3]，同样修改其中的 address 和 port，分别指向 slave1 和 slave2：



找到  **[MySQL Monitor]**  部分，修改 servers 为 server1,server2,server3，修改 user 和 passwd 为之前创建的监控用户的信息（scalemon,111111）。



找到 **[Read-Write Service]** 部分，修改 servers 为 server1,server2,server3，修改 user 和 passwd 为之前创建的路由用户的信息（maxscale,111111）。



由于我们使用了 [Read-Write Service]，需要删除另一个服务 [Read-Only Service]，删除其整块儿内容即可。

配置完成，保存并退出编辑器。

（4）启动 MaxScale

执行启动命令

maxscale --config=/etc/maxscale.cnf

查看 MaxScale 的响应端口是否已经就绪

netstat -ntelp

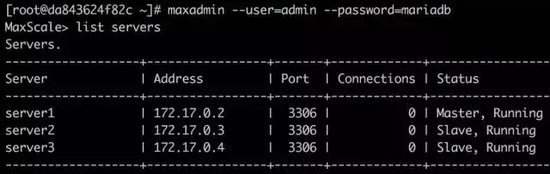
http://img0.tuicool.com/QVFZnyV.jpg!web

* 4006  是连接 MaxScale 时使用的端口
* 6603  是 MaxScale 管理器的端口

登录 MaxScale 管理器，查看一下数据库连接状态，默认的用户名和密码是 admin/mariadb。

maxadmin --user=admin --password=mariadb

MaxScale> list servers



可以看到，MaxScale 已经连接到了 master 和 slave。

（5）测试

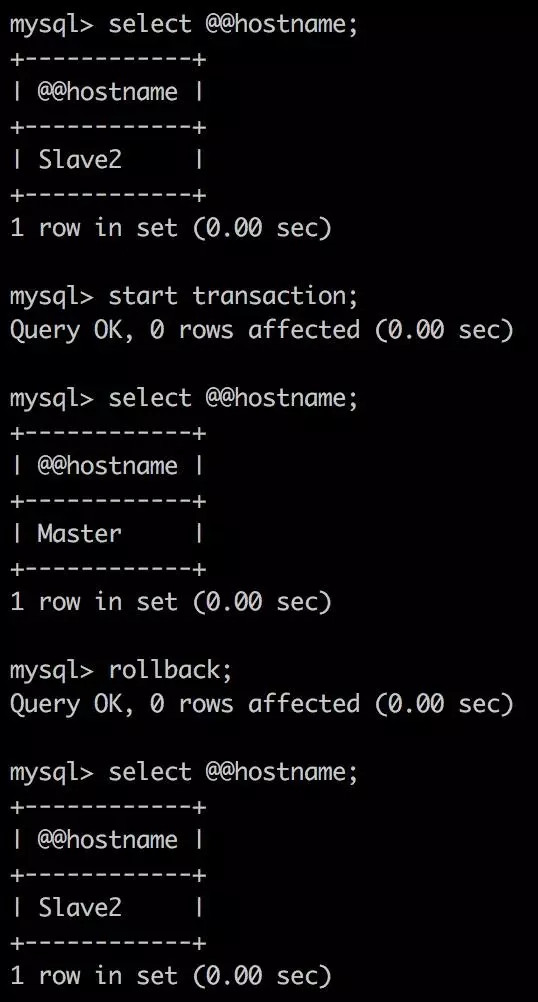
先在 master 上创建一个测试用户

mysql> grant ALL PRIVILEGES on \*.\* to rtest@"%" Identified by "111111";

使用 Mysql 客户端到连接 MaxScale

mysql -h MaxScale所在的IP -P 4006 -u rtest -p111111

执行查看数据库服务器名的操作来知道当前实际所在的数据库：



开启事务后，就自动路由到了 master，普通的查询操作，是在 slave上

MaxScale  的配置完成了。

4

**MaxScale 在 slave 有故障后的处理**

前面已经介绍了 MaxScale可以实现 MySQL 的读写分离和读负载均衡，那么当 slave 出现故障后，MaxScale 会如何处理呢？

例如有 3 台数据库服务器，一主二从的结构，数据库名称分别为 master, slave1, slave2。

现在我们实验以下两种情况：

（1）当一台从服务器（ slave1 或者 slave2 ）出现故障后，查看 MaxScale 如何应对，及故障服务器重新上线后的情况

（2）当两台从服务器（ slave1 和 slave2 ）都出现故障后，查看 MaxScale 如何应对，及故障服务器重新上线后的情况

准备

为了更深入的查看 MaxScale 的状态，需要把 MaxScale 的日志打开：

修改配置文件

vi /etc/maxscale.cnf

找到 [maxscale] 部分，这里用来进行全局设置，在其中添加日志。

配置

log\_info=1

logdir=/tmp/

通过开启 log\_info 级别，可以看到 MaxScale 的路由日志。

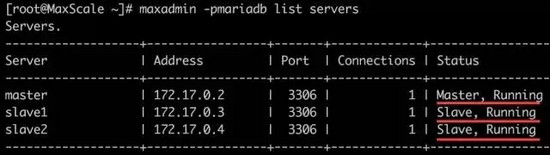
修改配置后，重启 MaxScale 。

实验过程

1

单个 slave 故障的情况

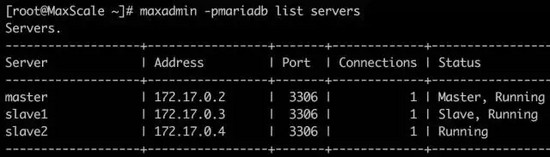
初始状态是一切正常。



停掉 slave2 的复制，登录 slave2 的 mysql 执行。

mysql> stop slave;

查看 MaxScale 服务器状态



slave2  已经失效了。

查看日志信息

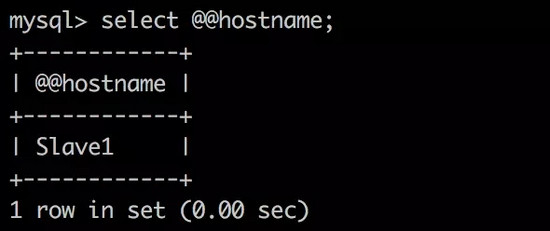
cat /tmp/maxscale1.log

尾部显示：

2016-08-15 12:26:02   notice : Server changed state: slave2[172.17.0.4:3306]: lost\_slave

提示 slave2 已经丢失。

查看客户端查询结果：



查询操作全都转到了 slave1。

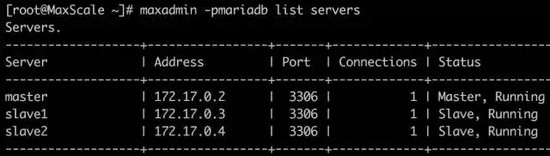
可以看到， 在有 slave 故障后，MaxScale 会自动进行排除，不再向其转发请求。

下面看下 slave2 再次 **上线后的情况。**

登录 slave2 的  MySQL 执行

mysql> start slave;

查看 MaxScale 服务器状态

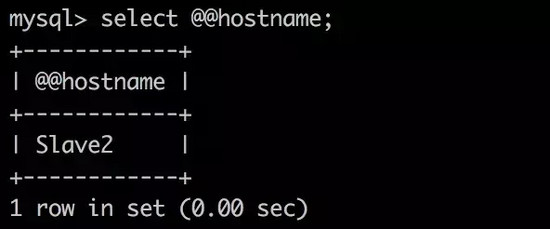


恢复了正常状态，重新识别到了 slave2。

查看日志信息，显示：

2016-08-15 12:32:36   notice : Server changed state: slave2[172.17.0.4:3306]: new\_slave

查看客户端查询结果：



slave2  又可以正常接受查询请求。

通过实验可以看到，在部分 slave 发生故障时，MaxScale 可以自动识别出来，并移除路由列表，当故障恢复重新上线后，MaxScale 也能自动将其加入路由，过程透明。

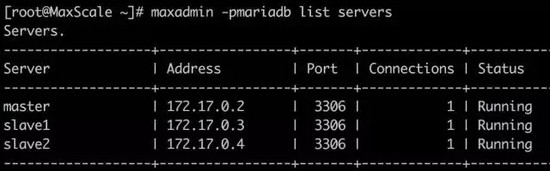
2

全部 slave 故障的情况

分别登陆  **slave1** 和 **slave2**的 MySQL ，执行停止复制的命令

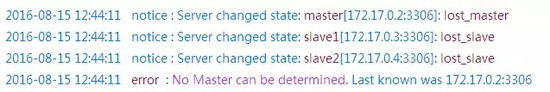
mysql> stop slave;

查看 MaxScale 服务器状态



发现各个服务器的角色都识别不出来了。

查看日志：



从日志中看到，MaxScale 发现2个slave 和 master 都丢了，然后报错：没有 master 了。

客户端连接 MaxScale 时也失败了。

http://img2.tuicool.com/u2Er2uV.jpg!web

说明从服务器全部失效后，会导致 master 也无法识别，使整个数据库服务都失效了。

对于 slave 全部失效的情况，能否让 master 还可用？这样至少可以正常提供数据库服务。

这需要修改 MaxScale 的配置，告诉 MaxScale 我们需要一个稳定的 master。

处理过程

先恢复两个 slave，让集群回到正常状态，登陆两个 slave 的MySQL。

mysql> start slave;

修改 MaxScale 配置文件，添加新的配置。

vi /etc/maxscale.cnf

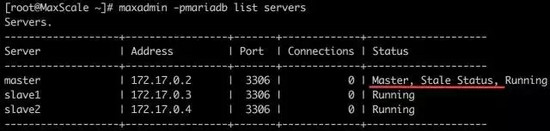
找到 [MySQL Monitor] 部分，添加：

detect\_stale\_master=true

保存退出，然后重启 MaxScale。

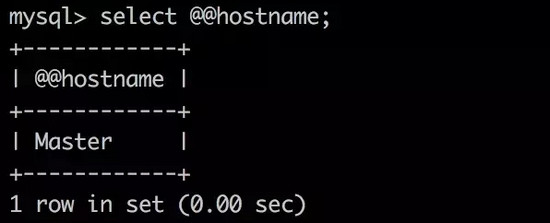
验证

停掉两台 slave ，查看 MaxScale 服务器状态。



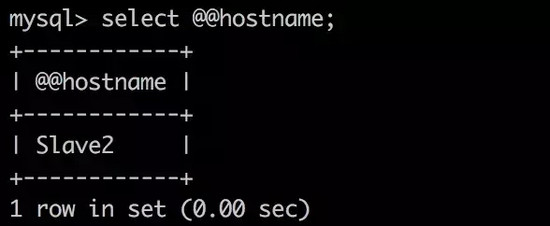
可以看到，虽然 slave 都无法识别了，但 master 还在，并提示处于稳定状态

客户端执行请求：



客户端可以连接 MaxScale，而且请求都转到了 master 上，说明 slave 全部失效时，由 master 支撑了全部请求。

当恢复两个 slave 后，整体状态自动恢复正常，从客户端执行请求时，又可以转到 slave 上。



小结

通过测试发现，在部分 slave 故障情况下，对于客户端是完全透明的，当全部 slave 故障时，经过简单的配置，MaxScale 也可以很好地处理。

经平台及作者同意授权转载

来源：性能与架构 订阅号（ID：yogoup）   
作者：杜亦舒

# innodb log

## 如何添加和调整mysql innodb log文件 - 运维生存时间

http://www.ttlsa.com/mysql/how-to-add-and-adjust-the-mysql-innodb-log-file/

数据在不断变化，数据库性能随着数据量的变大而变低，在对数据库进行性能分析时，为保障数据库处于高性能状态下，有时改变InnoDB日志文件大小或数量，这个过程需要停止mysql服务，否则会造成数据丢失或mysql服务启动失败。在说明如何添加和调整innodb log文件之前，先来看看与日志有关的参数。  
  
**innodb\_log\_group\_home\_dir**  
在事务被提交并写入到表空间磁盘文件上之前，事务数据存储在InnoDB的redo日志文件里。这些日志位于innodb\_log\_group\_home\_dir变量定义的目录中，通常我们把这个目录设置与innodb\_data\_home\_dir变量相同。为了获得最佳性能，建议分离innodb\_data\_home\_dir和innodb\_log\_group\_home\_dir到单独的物理存储阵列上，这样可以保证IO资源不起冲突，利于服务器处理大量高并发连接。

**innodb\_log\_file\_size**  
这个选项决定着性能，要慎重设置。默认设置为5M，难以满足生产环境下的需求。日志文件在mysql实例第一次启动时初始化，该文件是旋转的，因此可以根据文件修改时间来判断日志文件的旋转频率，旋转频率太频繁，说明日志文件太小了，要扩大。  
innodb\_log\_file\_size设置大小通常视innodb\_buffer\_pool\_size而定。影响日志文件性能的变量是innodb\_log\_buffer\_size，确保有足够大的日志缓冲区来保存脏数据在被写入到日志文件之前。  
对于比较小的innodb\_buffer\_pool\_size，建议是设置一样大。 但是，对于比较大的innodb\_buffer\_pool\_size，不建议这么设置，这会存在一个潜在的问题，那就是当mysql挂掉时，恢复数据需要很久，造成大量的停机时间。官方文档的建议设置是innodb\_buffer\_pool\_size/innodb\_log\_files\_in\_group。  
[warning]innodb\_log\_file\_size是静态的变量，需要以“干净”的方式更改并重新启动，否则mysql启动不起来。[/warning]

**innodb\_log\_buffer\_size**  
该变量将数据存导入到内存中，可以减少大量的IO资源消耗。当事务提交时，保存脏数据，后续在刷新到磁盘。当我们调整innodb\_buffer\_pool\_size大小时，innodb\_log\_buffer\_size和innodb\_log\_file\_size也应该做出相应的调整。

**innodb\_log\_files\_in\_group**  
该变量控制日志文件数。默认值为2。日志是以顺序的方式写入。  
[warning]innodb\_log\_files\_in\_group是静态的变量，需要以“干净”的方式更改并重新启动，否则mysql启动不起来。[/warning]

**添加和调整innodb log文件步骤：**  
1. 停止mysql服务



|  |  |
| --- | --- |
| 1 | # /etc/init.d/mysqld stop |

[warning]一定要正常的关闭。[/warning]2. 根据innodb\_log\_group\_home\_dir变量进入到日志目录下



|  |  |
| --- | --- |
| 1 | # cd /data/mysql\_data |

3. 备份旧的日志文件，以防不测便于回退



|  |  |
| --- | --- |
| 1 | # mv ib\_logfile\* /backup/ |

4. 按照要求调整日志文件大小或数量并写入到my.cnf文件中  
innodb\_log\_file\_size调整日志文件大小。  
innodb\_log\_files\_in\_group调整日志文件数量。  
innodb\_log\_group\_home\_dir调整日志文件位于目录。  
5. 重新启动mysql服务  
该过程将会按照新的日志配置来创建日志文件。同时，注意查看mysql错误日志，来监控是否出错。

就这么简单，下节来说说如果[调整表空间](http://www.ttlsa.com/html/2961.html)。

如需转载请注明出处： http://www.ttlsa.com/html/2937.html

# mysql 记录集中记录序号,MySQL中实现rownum功能类似的语句 - 老鸟的大心脏! - ITeye技术网站

http://qd5.iteye.com/blog/1160370

[MySQL中实现rownum功能类似的语句](http://gong-10140.iteye.com/blog/745685)

MySQL 如何实现 Oracle 的 ROWNUM2010/01/28 23:56mysql> SELECT \* FROM frutas;   
+-----------+----------+   
| nombre    | color    |   
+-----------+----------+   
| fresa     | rojo     |   
| platano   | amarillo |   
| manzana   | verde    |   
| uva       | verde    |   
| pera      | verde    |   
| mandarina | naranja  |   
| melocoton | marron   |   
| limon     | amarillo |   
+-----------+----------+   
8 rows in set (0,00 sec)   
  
【一条sql语句完成，不set变量】  
SELECT @rownum:=@rownum+1 AS rownum, frutas.\*   
FROM (SELECT @rownum:=0) r, frutas;   
  
mysql> SELECT @rownum:=@rownum+1 AS rownum, frutas.\*   
-> FROM (SELECT @rownum:=0) r, frutas;   
+--------+-----------+----------+   
| rownum | nombre    | color    |   
+--------+-----------+----------+   
|      1 | fresa     | rojo     |   
|      2 | platano   | amarillo |   
|      3 | manzana   | verde    |   
|      4 | uva       | verde    |   
|      5 | pera      | verde    |   
|      6 | mandarina | naranja  |   
|      7 | melocoton | marron   |   
|      8 | limon     | amarillo |   
+--------+-----------+----------+   
8 rows in set (0,00 sec)   
  
  
但是如何把这个查询放在 create view 里面   
mysql> CREATE VIEW vw\_frutas AS SELECT @rownum:=@rownum+1 AS rownum, frutas.\*   
->  FROM (SELECT @rownum:=0) r, frutas;   
ERROR 1351 (HY000): View's SELECT contains a variable or parameter

## 查看mysql版本的四种方法 -- 芽雨快跑

http://www.yayu.org/look.php?id=113

1：在终端下：mysql -V。

以下是代码片段：

[shengting@login ~]$ mysql -V

mysql Ver 14.7 Distrib 4.1.10a, for redhat-linux-gnu (i686)

2：在mysql中：mysql> status;

以下是代码片段：

mysql> status;

--------------

mysql Ver 14.7 Distrib 4.1.10a, for redhat-linux-gnu (i686)

Connection id: 416

SSL: Not in use

Current pager: stdout

Using outfile: ''

Using delimiter: ;

Server version: 3.23.56-log

Protocol version: 10

Connection: Localhost via UNIX socket

Client characterset: latin1

Server characterset: latin1

UNIX socket: /tmp/mysql\_3311.sock

Uptime: 62 days 21 hours 21 min 57 sec

Threads: 1 Questions: 584402560 Slow queries: 424 Opens: 59664208 Flush tables: 1 Open tables: 64 Queries per second avg: 107.551

3：在help里面查找

以下是代码片段：

[shengting@login ~]$ mysql --help | grep Distrib

mysql Ver 14.7 Distrib 4.1.10a, for redhat-linux-gnu (i686)

4：使用mysql的函数

以下是代码片段：

mysql> select version();

+-------------+

| version() |

+-------------+

| 3.23.56-log |

+-------------+

1 row in set (0.00 sec)

# 索引

## MySQL索引背后的数据结构及算法原理

http://www.uml.org.cn/sjjm/201107145.asp

**写在前面的话**

在编程领域有一句人尽皆知的法则“程序 = 数据结构 + 算法”，我个人是不太赞同这句话（因为我觉得程序不仅仅是数据结构加算法），但是在日常的学习和工作中我确认深深感受到数据结构和算法的重要性，很多东西，如果你愿意稍稍往深处挖一点，那么扑面而来的一定是各种数据结构和算法知识。例如几乎每个程序员都要打交道的数据库，如果仅仅是用来存个数据、建建表、建建索引、做做增删改查，那么也许觉得数据结构和这东西没什么关系。不过要是哪天心血来潮，想知道的多一点，想研究一下如何优化数据库，那么一定避免不了研究索引的原理，如果想要真正明白索引是怎么工作的，如何合理的使用索引以优化数据库，那么就免不了纠结于一堆数据结构与算法之间了。所以，如果说“程序的核心基础 = 数据结构 + 算法”我是十分赞同的，而一个想成为高手的程序员，一定会去学习程序的核心基础。

好吧，说了这么多，其实我的意思是如果想把数据库索引学个明明白白，就必须将数据结构和算法作为切入点去学习，遗憾的是我目前还没有在网上找到从原理层面去介绍数据库索引的资料（这里仅指在通俗资料领域没找到，不包括学术论文），倒不是说没有高水平的程序员，就只在我们公司范围内能把这一点讲透彻讲明白的数据库大牛也海了去了，只是由于工作的忙碌和个人兴趣原因，这些大牛们没有时间或没有兴趣去写这方面的文章。由于工作的需要，我这个半桶水的程序员这段时间也草草研究一些关于MySQL数据库索引的东西，虽然对这方面的理解相比那些大牛差的太远了，不过这里我还是将这些浅薄的知识总结成文吧。

[摘要](http://www.uml.org.cn/sjjm/201107145.asp#nav-1)

[数据结构及算法基础](http://www.uml.org.cn/sjjm/201107145.asp#nav-2)

[索引的本质](http://www.uml.org.cn/sjjm/201107145.asp#nav-2-1)

[B-Tree和B+Tree](http://www.uml.org.cn/sjjm/201107145.asp#nav-2-2)

[为什么实用B-Tree（B+Tree）](http://www.uml.org.cn/sjjm/201107145.asp#nav-2-3)

[MySQL索引实现](http://www.uml.org.cn/sjjm/201107145.asp#nav-3)

[MyISAM索引实现](http://www.uml.org.cn/sjjm/201107145.asp#nav-3-1)

[InnoDB索引实现](http://www.uml.org.cn/sjjm/201107145.asp#nav-3-2)

[索引用策略及优化](http://www.uml.org.cn/sjjm/201107145.asp#nav-4)

[示例数据库](http://www.uml.org.cn/sjjm/201107145.asp#nav-4-1)

[最左前缀原理与相关优化](http://www.uml.org.cn/sjjm/201107145.asp#nav-4-2)

[索引选择性与前缀索引](http://www.uml.org.cn/sjjm/201107145.asp#nav-4-3)

[InnoDB的主键选择与插入优化](http://www.uml.org.cn/sjjm/201107145.asp#nav-4-4)

[后记](http://www.uml.org.cn/sjjm/201107145.asp#nav-5)

[参考文献](http://www.uml.org.cn/sjjm/201107145.asp#nav-6)

摘要

本文以MySQL数据库为研究对象，讨论与数据库索引相关的一些话题。特别需要说明的是，MySQL支持诸多存储引擎，而各种存储引擎对索引的支持也各不相同，因此MySQL数据库支持多种索引类型，如BTree索引，哈希索引，全文索引等等。为了避免混乱，本文将只关注于BTree索引，因为这是平常使用MySQL时主要打交道的索引，至于哈希索引和全文索引本文暂不讨论。

文章主要内容分为四个部分。

第一部分主要从数据结构及算法理论层面讨论MySQL数据库索引的数理基础。

第二部分结合MySQL数据库中MyISAM和InnoDB数据存储引擎中索引的架构实现讨论聚集索引、非聚集索引及覆盖索引等话题。

第三部分根据上面的理论基础，讨论MySQL中高性能使用索引的策略。

数据结构及算法基础

索引的本质

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。**提取句子主干，就可以得到索引的本质：索引是数据结构。

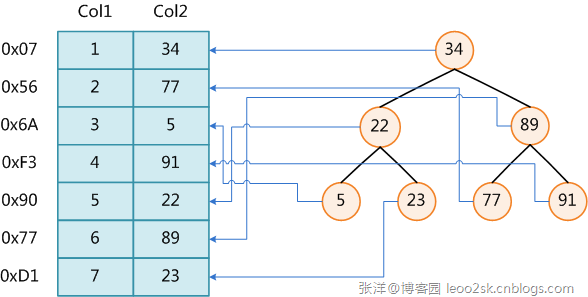
我们知道，数据库查询是数据库的最主要功能之一，例如下面的SQL语句：

|  |
| --- |
| SELECT \* FROM my\_table WHERE col2 = '77' |

可以从表“my\_table”中获得“col2”为“77”的数据记录。

我们都希望查询数据的速度能尽可能的快，因此数据库系统的设计者会从查询算法的角度进行优化。最基本的查询算法当然是[顺序查找](http://en.wikipedia.org/wiki/Linear_search)（linear search），遍历“my\_table”然后逐行匹配“col2”的值是否是“77”，这种复杂度为O(n)的算法在数据量很大时显然是糟糕的，好在计算机科学的发展提供了很多更优秀的查找算法，例如[二分查找](http://en.wikipedia.org/wiki/Binary_search_algorithm)（binary search）、[二叉树查找](http://en.wikipedia.org/wiki/Binary_search_tree)（binary tree search）等。如果稍微分析一下会发现，每种查找算法都只能应用于特定的数据结构之上，例如二分查找要求被检索数据有序，而二叉树查找只能应用于[二叉查找树](http://en.wikipedia.org/wiki/Binary_search_tree)上，但是数据本身的组织结构不可能完全满足各种数据结构（例如，理论上不可能同时将两列都按顺序进行组织），所以，**在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引**。

看一个例子：



**图1**

图1展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在O(log2n)的复杂度内获取到相应数据。

虽然这是一个货真价实的索引，但是实际的数据库系统几乎没有使用二叉查找树或其进化品种红黑树（red-black tree）实现的，原因会在下文介绍。

B-Tree和B+Tree

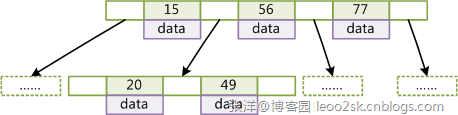
目前大部分数据库系统及文件系统都采用B-Tree或其变种B+Tree作为索引结构，在本文的下一节会结合存储器原理及计算机存取原理讨论为什么B-Tree和B+Tree在被如此广泛用于索引，这一节先单纯从数据结构角度描述它们。

B-Tree

为了描述B-Tree，首先定义一条数据记录为一个二元组[key, data]，key为记录的键值，对于不同数据记录，key是互不相同的；data为数据记录除key外的数据。那么B-Tree是满足下列条件的数据结构：

1. d为大于1的一个正整数，称为B-Tree的度。
2. h为一个正整数，称为B-Tree的高度。
3. 每个非叶子节点由n-1个key和n个指针组成，其中d<=n<=2d。
4. 每个叶子节点最少包含一个key和两个指针，最多包含2d-1个key和2d个指针，叶节点的指针均为null 。
5. 所有叶节点具有相同的深度，等于树高h。
6. key和指针互相间隔，节点两端是指针。
7. 一个节点中的key从左到右非递减排列。
8. 所有节点组成树结构。
9. 每个指针要么为null，要么指向另外一个节点。
10. 如果某个指针在节点node最左边且不为null，则其指向节点的所有key小于v(key1)，其中v(key1)为node的第一个key的值。
11. 如果某个指针在节点node最右边且不为null，则其指向节点的所有key大于v(keym)，其中v(keym)为node的最后一个key的值。
12. 如果某个指针在节点node的左右相邻key分别是keyi和keyi+1且不为null，则其指向节点的所有key小于v(keyi+1)且大于v(keyi)。

图2是一个d=2的B-Tree示意图。



**图2**

由于B-Tree的特性，在B-Tree中按key检索数据的算法非常直观：首先从根节点进行二分查找，如果找到则返回对应节点的data，否则对相应区间的指针指向的节点递归进行查找，直到找到节点或找到null指针，前者查找成功，后者查找失败。B-Tree上查找算法的伪代码如下：

BTree\_Search(node, key)

{

    if(node == null) return null;

    foreach(node.key)

    {

        if(node.key[i] == key) return node.data[i];

        if(node.key[i] > key) return BTree\_Search(point[i]->node);

    }

    return BTree\_Search(point[i+1]->node);

}

data = BTree\_Search(root, my\_key);

关于B-Tree有一系列有趣的性质，例如**一个度为d的B-Tree，设其索引N个key，则其树高h的上限为logd((N+1)/2)，检索一个key，其查找节点个数的渐进复杂度为O(logdN)。**从这点可以看出，B-Tree是一个非常有效率的索引数据结构。

另外，由于插入删除新的数据记录会破坏B-Tree的性质，因此在插入删除时，需要对树进行一个分裂、合并、转移等操作以保持B-Tree性质，本文不打算完整讨论B-Tree这些内容，因为已经有许多资料详细说明了B-Tree的数学性质及插入删除算法，有兴趣的朋友可以在本文末的参考文献一栏找到相应的资料进行阅读。

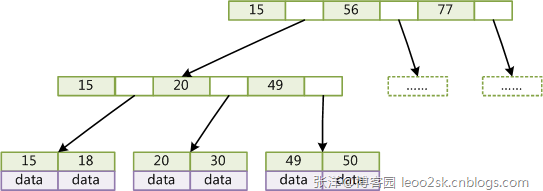
B+Tree

B-Tree有许多变种，其中最常见的是B+Tree，例如MySQL就普遍使用B+Tree实现其索引结构。

与B-Tree相比，B+Tree有以下不同点：

* 1. 每个节点的指针上限为2d而不是2d+1。
  2. 内节点不存储data，只存储key；叶子节点不存储指针。

图3是一个简单的B+Tree示意。



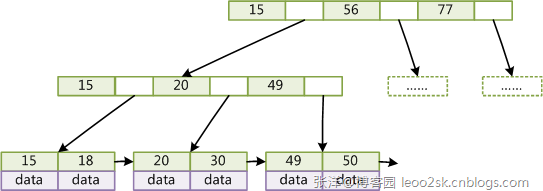
**图3**

由于并不是所有节点都具有相同的域，因此B+Tree中叶节点和内节点一般大小不同。这点与B-Tree不同，虽然B-Tree中不同节点存放的key和指针可能数量不一致，但是每个节点的域和上限是一致的，所以在实现中B-Tree往往对每个节点申请同等大小的空间。

一般来说，B+Tree比B-Tree更适合实现外存储索引结构，具体原因与外存储器原理及计算机存取原理有关，将在下面讨论。

带有顺序访问指针的B+Tree

一般在数据库系统或文件系统中使用的B+Tree结构都在经典B+Tree的基础上进行了优化，增加了顺序访问指针。



**图4**

如图4所示，在B+Tree的每个叶子节点增加一个指向相邻叶子节点的指针，就形成了带有顺序访问指针的B+Tree。做这个优化的目的是为了提高区间访问的性能，例如图4中如果要查询key为从18到49的所有数据记录，当找到18后，只需顺着节点和指针顺序遍历就可以一次性访问到所有数据节点，极大提到了区间查询效率。

这一节对B-Tree和B+Tree进行了一个简单的介绍，下一节结合存储器存取原理介绍为什么目前B+Tree是数据库系统实现索引的首选数据结构。

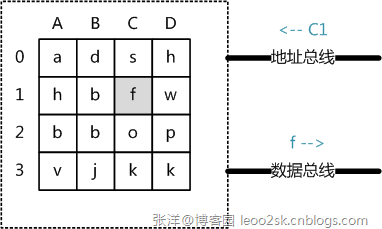
为什么使用B-Tree（B+Tree）

上文说过，红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用B-/+Tree作为索引结构，这一节将结合计算机组成原理相关知识讨论B-/+Tree作为索引的理论基础。

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。下面先介绍内存和磁盘存取原理，然后再结合这些原理分析B-/+Tree作为索引的效率。

主存存取原理

目前计算机使用的主存基本都是随机读写存储器（RAM），现代RAM的结构和存取原理比较复杂，这里本文抛却具体差别，抽象出一个十分简单的存取模型来说明RAM的工作原理。

[](http://www.uml.org.cn/sjjm/images/201107111112524101.png)

**图5**

从抽象角度看，主存是一系列的存储单元组成的矩阵，每个存储单元存储固定大小的数据。每个存储单元有唯一的地址，现代主存的编址规则比较复杂，这里将其简化成一个二维地址：通过一个行地址和一个列地址可以唯一定位到一个存储单元。图5展示了一个4 x 4的主存模型。

主存的存取过程如下：

当系统需要读取主存时，则将地址信号放到地址总线上传给主存，主存读到地址信号后，解析信号并定位到指定存储单元，然后将此存储单元数据放到数据总线上，供其它部件读取。

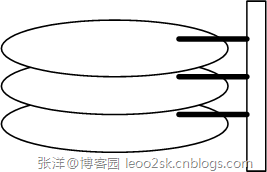
写主存的过程类似，系统将要写入单元地址和数据分别放在地址总线和数据总线上，主存读取两个总线的内容，做相应的写操作。

这里可以看出，主存存取的时间仅与存取次数呈线性关系，因为不存在机械操作，两次存取的数据的“距离”不会对时间有任何影响，例如，先取A0再取A1和先取A0再取D3的时间消耗是一样的。

磁盘存取原理

上文说过，索引一般以文件形式存储在磁盘上，索引检索需要磁盘I/O操作。与主存不同，磁盘I/O存在机械运动耗费，因此磁盘I/O的时间消耗是巨大的。

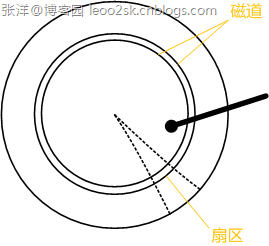
图6是磁盘的整体结构示意图。



**图6**

一个磁盘由大小相同且同轴的圆形盘片组成，磁盘可以转动（各个磁盘必须同步转动）。在磁盘的一侧有磁头支架，磁头支架固定了一组磁头，每个磁头负责存取一个磁盘的内容。磁头不能转动，但是可以沿磁盘半径方向运动（实际是斜切向运动），每个磁头同一时刻也必须是同轴的，即从正上方向下看，所有磁头任何时候都是重叠的（不过目前已经有多磁头独立技术，可不受此限制）。

图7是磁盘结构的示意图。



**图7**

盘片被划分成一系列同心环，圆心是盘片中心，每个同心环叫做一个磁道，所有半径相同的磁道组成一个柱面。磁道被沿半径线划分成一个个小的段，每个段叫做一个扇区，每个扇区是磁盘的最小存储单元。为了简单起见，我们下面假设磁盘只有一个盘片和一个磁头。

当需要从磁盘读取数据时，系统会将数据逻辑地址传给磁盘，磁盘的控制电路按照寻址逻辑将逻辑地址翻译成物理地址，即确定要读的数据在哪个磁道，哪个扇区。为了读取这个扇区的数据，需要将磁头放到这个扇区上方，为了实现这一点，磁头需要移动对准相应磁道，这个过程叫做寻道，所耗费时间叫做寻道时间，然后磁盘旋转将目标扇区旋转到磁头下，这个过程耗费的时间叫做旋转时间。

局部性原理与磁盘预读

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：

当一个数据被用到时，其附近的数据也通常会马上被使用。

程序运行期间所需要的数据通常比较集中。

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。

预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

B-/+Tree索引的性能分析

到这里终于可以分析B-/+Tree索引的性能了。

上文说过一般使用磁盘I/O次数评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

**B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)。**一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

综上所述，用B-Tree作为索引结构效率是非常高的。

而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

上文还说过，B+Tree更适合外存索引，原因和内节点出度d有关。从上面分析可以看到，d越大索引的性能越好，而出度的上限取决于节点内key和data的大小：

dmax = ***floor***(pagesize / (keysize + datasize + pointsize)) (pagesize – dmax >= pointsize)

或

dmax = ***floor***(pagesize / (keysize + datasize + pointsize)) - 1 (pagesize – dmax < pointsize)

floor表示向下取整。由于B+Tree内节点去掉了data域，因此可以拥有更大的出度，拥有更好的性能。

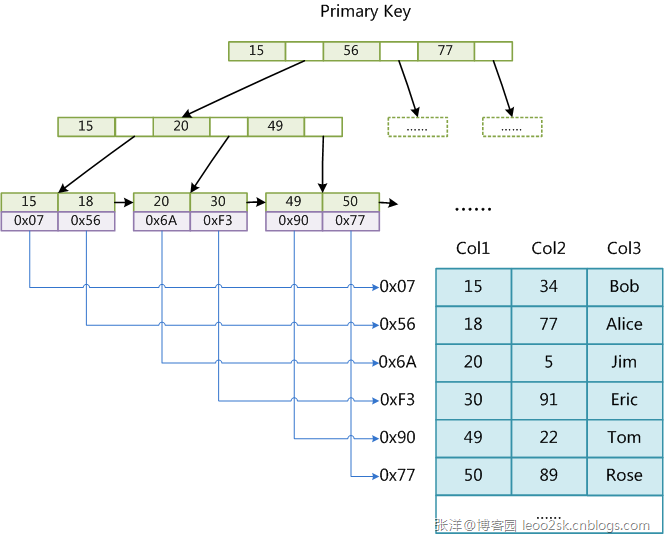
这一章从理论角度讨论了与索引相关的数据结构与算法问题，下一章将讨论B+Tree是如何具体实现为MySQL中索引，同时将结合MyISAM和InnDB存储引擎介绍非聚集索引和聚集索引两种不同的索引实现形式。

MySQL索引实现

在MySQL中，索引属于存储引擎级别的概念，不同存储引擎对索引的实现方式是不同的，本文主要讨论MyISAM和InnoDB两个存储引擎的索引实现方式。

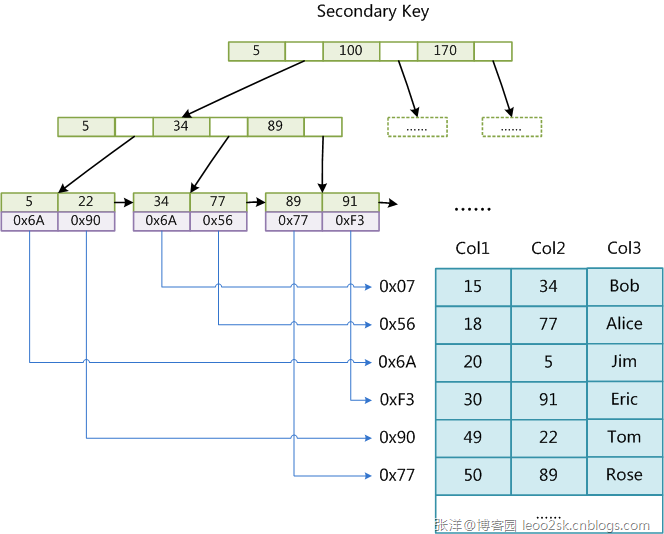
MyISAM索引实现

MyISAM引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址。下图是MyISAM索引的原理图：



**图8**

这里设表一共有三列，假设我们以Col1为主键，则图8是一个MyISAM表的主索引（Primary key）示意。可以看出MyISAM的索引文件仅仅保存数据记录的地址。**在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。**如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：



**图9**

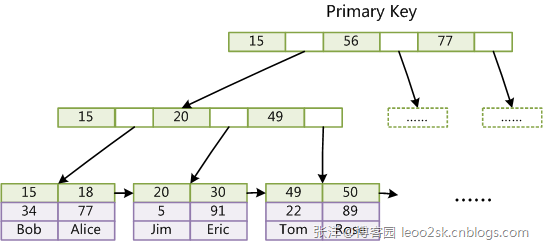
同样也是一颗B+Tree，data域保存数据记录的地址。因此，**MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取相应数据记录。**

MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

InnoDB索引实现

虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

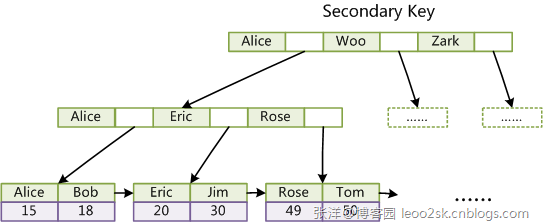
**第一个重大区别是InnoDB的数据文件本身就是索引文件。**从上文知道，MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。



**图10**

图10是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

**第二个与MyISAM索引的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地址。**换句话说，InnoDB的所有辅助索引都引用主键作为data域。例如，图11为定义在Col3上的一个辅助索引：



**图11**

这里以英文字符的ASCII码作为比较准则。**聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。**

了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，用非单调的字段作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

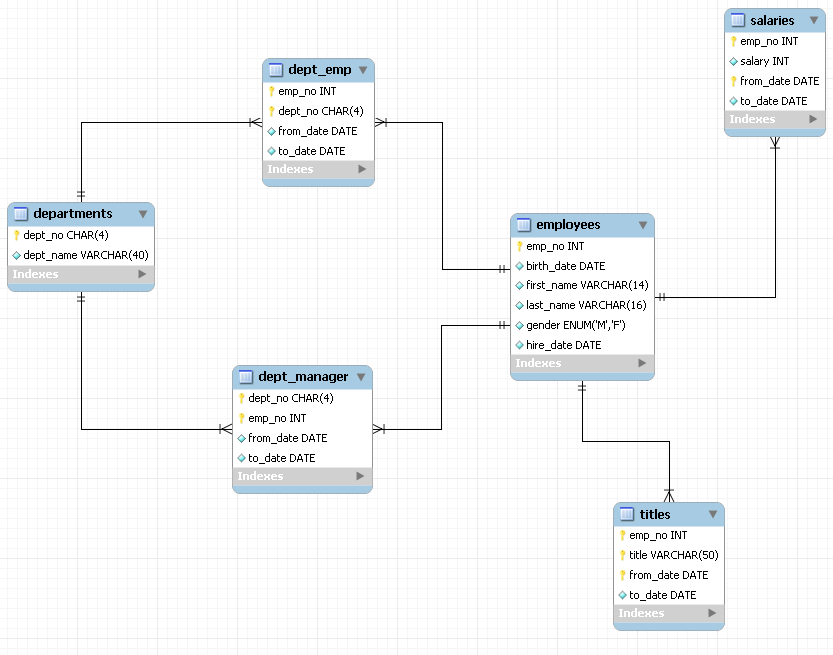
下一章将具体讨论这些与索引有关的优化策略。

索引使用策略及优化

MySQL的优化主要分为结构优化（Scheme optimization）和查询优化（Query optimization）。本章讨论的高性能索引策略主要属于结构优化范畴。本章的内容完全基于上文的理论基础，实际上一旦理解了索引背后的机制，那么选择高性能的策略就变成了纯粹的推理，并且可以理解这些策略背后的逻辑。

示例数据库

为了讨论索引策略，需要一个数据量不算小的数据库作为示例。本文选用MySQL官方文档中提供的示例数据库之一：employees。这个数据库关系复杂度适中，且数据量较大。下图是这个数据库的E-R关系图（引用自MySQL官方手册）：



**图12**

MySQL官方文档中关于此数据库的页面为<http://dev.mysql.com/doc/employee/en/employee.html>。里面详细介绍了此数据库，并提供了下载地址和导入方法，如果有兴趣导入此数据库到自己的MySQL可以参考文中内容。

最左前缀原理与相关优化

高效使用索引的首要条件是知道什么样的查询会使用到索引，这个问题和B+Tree中的“最左前缀原理”有关，下面通过例子说明最左前缀原理。

这里先说一下联合索引的概念。在上文中，我们都是假设索引只引用了单个的列，实际上，MySQL中的索引可以以一定顺序引用多个列，这种索引叫做联合索引，一般的，一个联合索引是一个有序元组<a1, a2, …, an>，其中各个元素均为数据表的一列，实际上要严格定义索引需要用到关系代数，但是这里我不想讨论太多关系代数的话题，因为那样会显得很枯燥，所以这里就不再做严格定义。另外，单列索引可以看成联合索引元素数为1的特例。

以employees.titles表为例，下面先查看其上都有哪些索引：

|  |
| --- |
| SHOW INDEX FROM employees.titles; |

|  |
| --- |
| +--------+------------+----------+--------------+-------------+-----------+-------------+------+------------+ |

|  |
| --- |
| | Table | Non\_unique | Key\_name | Seq\_in\_index | Column\_name | Collation | Cardinality | Null | Index\_type | |

|  |
| --- |
| +--------+------------+----------+--------------+-------------+-----------+-------------+------+------------+ |

|  |
| --- |
| | titles | 0 | PRIMARY | 1 | emp\_no | A | NULL | | BTREE | |

|  |
| --- |
| | titles | 0 | PRIMARY | 2 | title | A | NULL | | BTREE | |

|  |
| --- |
| | titles | 0 | PRIMARY | 3 | from\_date | A | 443308 | | BTREE | |

|  |
| --- |
| | titles | 1 | emp\_no | 1 | emp\_no | A | 443308 | | BTREE | |

|  |
| --- |
| +--------+------------+----------+--------------+-------------+-----------+-------------+------+------------+ |

从结果中可以到titles表的主索引为<emp\_no, title, from\_date>，还有一个辅助索引<emp\_no>。为了避免多个索引使事情变复杂（MySQL的SQL优化器在多索引时行为比较复杂），这里我们将辅助索引drop掉：

|  |
| --- |
| ALTER TABLE employees.titles DROP INDEX emp\_no; |

这样就可以专心分析索引PRIMARY的行为了。

情况一：全列匹配。

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no='10001' AND title='Senior Engineer' AND from\_date='1986-06-26'; |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+-------------------+------+-------+ |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+-------------------+------+-------+ |

|  |
| --- |
| | 1 | SIMPLE | titles | const | PRIMARY | PRIMARY | 59 | const,const,const | 1 | | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+-------------------+------+-------+ |

很明显，当按照索引中所有列进行精确匹配（这里精确匹配指“=”或“IN”匹配）时，索引可以被用到。这里有一点需要注意，理论上索引对顺序是敏感的，但是由于MySQL的查询优化器会自动调整where子句的条件顺序以使用适合的索引，例如我们将where中的条件顺序颠倒：

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.titles WHERE from\_date='1986-06-26' AND emp\_no='10001' AND title='Senior Engineer'; |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+-------------------+------+-------+ |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+-------------------+------+-------+ |

|  |
| --- |
| | 1 | SIMPLE | titles | const | PRIMARY | PRIMARY | 59 | const,const,const | 1 | | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+-------------------+------+-------+ |

效果是一样的。

情况二：最左前缀匹配。

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no='10001'; |

|  |
| --- |
| +----+-------------+--------+------+---------------+---------+---------+-------+------+-------+ |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+--------+------+---------------+---------+---------+-------+------+-------+ |

|  |
| --- |
| | 1 | SIMPLE | titles | ref | PRIMARY | PRIMARY | 4 | const | 1 | | |

|  |
| --- |
| +----+-------------+--------+------+---------------+---------+---------+-------+------+-------+ |

当查询条件精确匹配索引的左边连续一个或几个列时，如<emp\_no>或<emp\_no, title>，所以可以被用到，但是只能用到一部分，即条件所组成的最左前缀。上面的查询从分析结果看用到了PRIMARY索引，但是key\_len为4，说明只用到了索引的第一列前缀。

情况三：查询条件用到了索引中列的精确匹配，但是中间某个条件未提供。

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no='10001' AND from\_date='1986-06-26'; |

|  |
| --- |
| +----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+ |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+ |

|  |
| --- |
| | 1 | SIMPLE | titles | ref | PRIMARY | PRIMARY | 4 | const | 1 | Using where | |

|  |
| --- |
| +----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+ |

此时索引使用情况和情况二相同，因为title未提供，所以查询只用到了索引的第一列，而后面的from\_date虽然也在索引中，但是由于title不存在而无法和左前缀连接，因此需要对结果进行扫描过滤from\_date（这里由于emp\_no唯一，所以不存在扫描）。如果想让from\_date也使用索引而不是where过滤，可以增加一个辅助索引<emp\_no, from\_date>，此时上面的查询会使用这个索引。除此之外，还可以使用一种称之为“隔离列”的优化方法，将emp\_no与from\_date之间的“坑”填上。

首先我们看下title一共有几种不同的值：

|  |
| --- |
| SELECT DISTINCT(title) FROM employees.titles; |

|  |
| --- |
| +--------------------+ |

|  |
| --- |
| | title | |

|  |
| --- |
| +--------------------+ |

|  |
| --- |
| | Senior Engineer | |

|  |
| --- |
| | Staff | |

|  |
| --- |
| | Engineer | |

|  |
| --- |
| | Senior Staff | |

|  |
| --- |
| | Assistant Engineer | |

|  |
| --- |
| | Technique Leader | |

|  |
| --- |
| | Manager | |

|  |
| --- |
| +--------------------+ |

只有7种。在这种成为“坑”的列值比较少的情况下，可以考虑用“IN”来填补这个“坑”从而形成最左前缀：

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.titles |

|  |
| --- |
| WHERE emp\_no='10001' |

|  |
| --- |
| AND title IN ('Senior Engineer', 'Staff', 'Engineer', 'Senior Staff', 'Assistant Engineer', 'Technique Leader', 'Manager') |

|  |
| --- |
| AND from\_date='1986-06-26'; |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

|  |
| --- |
| | 1 | SIMPLE | titles | range | PRIMARY | PRIMARY | 59 | NULL | 7 | Using where | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

这次key\_len为59，说明索引被用全了，但是从type和rows看出IN实际上执行了一个range查询，这里检查了7个key。看下两种查询的性能比较：

|  |
| --- |
| SHOW PROFILES; |

|  |
| --- |
| +----------+------------+-------------------------------------------------------------------------------+ |

|  |
| --- |
| | Query\_ID | Duration | Query | |

|  |
| --- |
| +----------+------------+-------------------------------------------------------------------------------+ |

|  |
| --- |
| | 10 | 0.00058000 | SELECT \* FROM employees.titles WHERE emp\_no='10001' AND from\_date='1986-06-26'| |

|  |
| --- |
| | 11 | 0.00052500 | SELECT \* FROM employees.titles WHERE emp\_no='10001' AND title IN ... | |

|  |
| --- |
| +----------+------------+-------------------------------------------------------------------------------+ |

“填坑”后性能提升了一点。如果经过emp\_no筛选后余下很多数据，则后者性能优势会更加明显。当然，如果title的值很多，用填坑就不合适了，必须建立辅助索引。

情况四：查询条件没有指定索引第一列。

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.titles WHERE from\_date='1986-06-26'; |

|  |
| --- |
| +----+-------------+--------+------+---------------+------+---------+------+--------+-------------+ |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+--------+------+---------------+------+---------+------+--------+-------------+ |

|  |
| --- |
| | 1 | SIMPLE | titles | ALL | NULL | NULL | NULL | NULL | 443308 | Using where | |

|  |
| --- |
| +----+-------------+--------+------+---------------+------+---------+------+--------+-------------+ |

由于不是最左前缀，索引这样的查询显然用不到索引。

情况五：匹配某列的前缀字符串。

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no='10001' AND title LIKE 'Senior%'; |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ | |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

|  |
| --- |
| | 1 | SIMPLE | titles | range | PRIMARY | PRIMARY | 56 | NULL | 1 | Using where | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

此时可以用到索引，但是如果通配符不是只出现在末尾，则无法使用索引。

情况六：范围查询。

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no<'10010' and title='Senior Engineer'; |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

|  |
| --- |
| | 1 | SIMPLE | titles | range | PRIMARY | PRIMARY | 4 | NULL | 16 | Using where | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

范围列可以用到索引（必须是最左前缀），但是范围列后面的列无法用到索引。同时，索引最多用于一个范围列，因此如果查询条件中有两个范围列则无法全用到索引。

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.titles |

|  |
| --- |
| WHERE emp\_no<'10010' |

|  |
| --- |
| AND title='Senior Engineer' |

|  |
| --- |
| AND from\_date BETWEEN '1986-01-01' AND '1986-12-31'; |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

|  |
| --- |
| | 1 | SIMPLE | titles | range | PRIMARY | PRIMARY | 4 | NULL | 16 | Using where | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

可以看到索引对第二个范围索引无能为力。这里特别要说明MySQL一个有意思的地方，那就是仅用explain可能无法区分范围索引和多值匹配，因为在type中这两者都显示为range。同时，用了“between”并不意味着就是范围查询，例如下面的查询：

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.titles |

|  |
| --- |
| WHERE emp\_no BETWEEN '10001' AND '10010' |

|  |
| --- |
| AND title='Senior Engineer' |

|  |
| --- |
| AND from\_date BETWEEN '1986-01-01' AND '1986-12-31'; |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

|  |
| --- |
| | 1 | SIMPLE | titles | range | PRIMARY | PRIMARY | 59 | NULL | 16 | Using where | |

|  |
| --- |
| +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+ |

看起来是用了两个范围查询，但作用于emp\_no上的“BETWEEN”实际上相当于“IN”，也就是说emp\_no实际是多值精确匹配。可以看到这个查询用到了索引全部三个列。因此在MySQL中要谨慎地区分多值匹配和范围匹配，否则会对MySQL的行为产生困惑。

情况七：查询条件中含有函数或表达式。

很不幸，如果查询条件中含有函数或表达式，则MySQL不会为这列使用索引（虽然某些在数学意义上可以使用）。例如：

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no='10001' AND left(title, 6)='Senior'; |

|  |
| --- |
| +----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+ |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+ |

|  |
| --- |
| | 1 | SIMPLE | titles | ref | PRIMARY | PRIMARY | 4 | const | 1 | Using where | |

|  |
| --- |
| +----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+ |

虽然这个查询和情况五中功能相同，但是由于使用了函数left，则无法为title列应用索引，而情况五中用LIKE则可以。再如：

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no - 1='10000'; |

|  |
| --- |
| +----+-------------+--------+------+---------------+------+---------+------+--------+-------------+ |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+--------+------+---------------+------+---------+------+--------+-------------+ |

|  |
| --- |
| | 1 | SIMPLE | titles | ALL | NULL | NULL | NULL | NULL | 443308 | Using where | |

|  |
| --- |
| +----+-------------+--------+------+---------------+------+---------+------+--------+-------------+ |

显然这个查询等价于查询emp\_no为10001的函数，但是由于查询条件是一个表达式，MySQL无法为其使用索引。看来MySQL还没有智能到自动优化常量表达式的程度，因此在写查询语句时尽量避免表达式出现在查询中，而是先手工私下代数运算，转换为无表达式的查询语句。

索引选择性与前缀索引

既然索引可以加快查询速度，那么是不是只要是查询语句需要，就建上索引？答案是否定的。因为索引虽然加快了查询速度，但索引也是有代价的：索引文件本身要消耗存储空间，同时索引会加重插入、删除和修改记录时的负担，另外，MySQL在运行时也要消耗资源维护索引，因此索引并不是越多越好。一般两种情况下不建议建索引。

第一种情况是表记录比较少，例如一两千条甚至只有几百条记录的表，没必要建索引，让查询做全表扫描就好了。至于多少条记录才算多，这个个人有个人的看法，我个人的经验是以2000作为分界线，记录数不超过 2000可以考虑不建索引，超过2000条可以酌情考虑索引。

另一种不建议建索引的情况是索引的选择性较低。所谓索引的选择性（Selectivity），是指不重复的索引值（也叫基数，Cardinality）与表记录数（#T）的比值：

Index Selectivity = Cardinality / #T

显然选择性的取值范围为(0, 1]，选择性越高的索引价值越大，这是由B+Tree的性质决定的。例如，上文用到的employees.titles表，如果title字段经常被单独查询，是否需要建索引，我们看一下它的选择性：

|  |
| --- |
| SELECT count(DISTINCT(title))/count(\*) AS Selectivity FROM employees.titles; |

|  |
| --- |
| +-------------+ |

|  |
| --- |
| | Selectivity | |

|  |
| --- |
| +-------------+ |

|  |
| --- |
| | 0.0000 | |

|  |
| --- |
| +-------------+ |

title的选择性不足0.0001（精确值为0.00001579），所以实在没有什么必要为其单独建索引。

有一种与索引选择性有关的索引优化策略叫做前缀索引，就是用列的前缀代替整个列作为索引key，当前缀长度合适时，可以做到既使得前缀索引的选择性接近全列索引，同时因为索引key变短而减少了索引文件的大小和维护开销。下面以employees.employees表为例介绍前缀索引的选择和使用。

从图12可以看到employees表只有一个索引<emp\_no>，那么如果我们想按名字搜索一个人，就只能全表扫描了：

|  |
| --- |
| EXPLAIN SELECT \* FROM employees.employees WHERE first\_name='Eric' AND last\_name='Anido'; |

|  |
| --- |
| +----+-------------+-----------+------+---------------+------+---------+------+--------+-------------+ |

|  |
| --- |
| | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra | |

|  |
| --- |
| +----+-------------+-----------+------+---------------+------+---------+------+--------+-------------+ |

|  |
| --- |
| | 1 | SIMPLE | employees | ALL | NULL | NULL | NULL | NULL | 300024 | Using where | |

|  |
| --- |
| +----+-------------+-----------+------+---------------+------+---------+------+--------+-------------+ |

如果频繁按名字搜索员工，这样显然效率很低，因此我们可以考虑建索引。有两种选择，建<first\_name>或<first\_name, last\_name>，看下两个索引的选择性：

|  |
| --- |
| SELECT count(DISTINCT(first\_name))/count(\*) AS Selectivity FROM employees.employees; |

|  |
| --- |
| +-------------+ |

|  |
| --- |
| | Selectivity | |

|  |
| --- |
| +-------------+ |

|  |
| --- |
| | 0.0042 | |

|  |
| --- |
| +-------------+ |

|  |
| --- |
|  |

|  |
| --- |
| SELECT count(DISTINCT(concat(first\_name, last\_name)))/count(\*) AS Selectivity FROM employees.employees; |

|  |
| --- |
| +-------------+ |

|  |
| --- |
| | Selectivity | |

|  |
| --- |
| +-------------+ |

|  |
| --- |
| | 0.9313 | |

|  |
| --- |
| +-------------+ |

<first\_name>显然选择性太低，<first\_name, last\_name>选择性很好，但是first\_name和last\_name加起来长度为30，有没有兼顾长度和选择性的办法？可以考虑用first\_name和last\_name的前几个字符建立索引，例如<first\_name, left(last\_name, 3)>，看看其选择性：

|  |
| --- |
| SELECT count(DISTINCT(concat(first\_name, left(last\_name, 3))))/count(\*) AS Selectivity FROM employees.employees; |

|  |
| --- |
| +-------------+ |

|  |
| --- |
| | Selectivity | |

|  |
| --- |
| +-------------+ |

|  |
| --- |
| | 0.7879 | |

|  |
| --- |
| +-------------+ |

选择性还不错，但离0.9313还是有点距离，那么把last\_name前缀加到4：

|  |
| --- |
| SELECT count(DISTINCT(concat(first\_name, left(last\_name, 4))))/count(\*) AS Selectivity FROM employees.employees; |

|  |
| --- |
| +-------------+ |

|  |
| --- |
| | Selectivity | |

|  |
| --- |
| +-------------+ |

|  |
| --- |
| | 0.9007 | |

|  |
| --- |
| +-------------+ |

这时选择性已经很理想了，而这个索引的长度只有18，比<first\_name, last\_name>短了接近一半，我们把这个前缀索引 建上：

|  |
| --- |
| ALTER TABLE employees.employees |

|  |
| --- |
| ADD INDEX `first\_name\_last\_name4` (first\_name, last\_name(4)); |

此时再执行一遍按名字查询，比较分析一下与建索引前的结果：

|  |
| --- |
| SHOW PROFILES; |

|  |
| --- |
| +----------+------------+---------------------------------------------------------------------------------+ |

|  |
| --- |
| | Query\_ID | Duration | Query | |

|  |
| --- |
| +----------+------------+---------------------------------------------------------------------------------+ |

|  |
| --- |
| | 87 | 0.11941700 | SELECT \* FROM employees.employees WHERE first\_name='Eric' AND last\_name='Anido' | |

|  |
| --- |
| | 90 | 0.00092400 | SELECT \* FROM employees.employees WHERE first\_name='Eric' AND last\_name='Anido' | |

|  |
| --- |
| +----------+------------+---------------------------------------------------------------------------------+ |

性能的提升是显著的，查询速度提高了120多倍。

前缀索引兼顾索引大小和查询速度，但是其缺点是不能用于ORDER BY和GROUP BY操作，也不能用于Covering index（即当索引本身包含查询所需全部数据时，不再访问数据文件本身）。

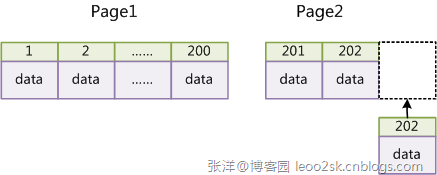
InnoDB的主键选择与插入优化

在使用InnoDB存储引擎时，如果没有特别的需要，请永远使用一个与业务无关的自增字段作为主键。

经常看到有帖子或博客讨论主键选择问题，有人建议使用业务无关的自增主键，有人觉得没有必要，完全可以使用如学号或身份证号这种唯一字段作为主键。不论支持哪种论点，大多数论据都是业务层面的。如果从数据库索引优化角度看，使用InnoDB引擎而不使用自增主键绝对是一个糟糕的主意。

上文讨论过InnoDB的索引实现，InnoDB使用聚集索引，数据记录本身被存于主索引（一颗B+Tree）的叶子节点上。这就要求同一个叶子节点内（大小为一个内存页或磁盘页）的各条数据记录按主键顺序存放，因此每当有一条新的记录插入时，MySQL会根据其主键将其插入适当的节点和位置，如果页面达到装载因子（InnoDB默认为15/16），则开辟一个新的页（节点）。

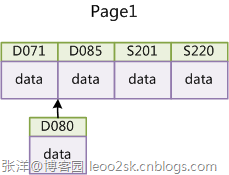
如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页。如下图所示：



**图13**

这样就会形成一个紧凑的索引结构，近似顺序填满。由于每次插入时也不需要移动已有数据，因此效率很高，也不会增加很多开销在维护索引上。

如果使用非自增主键（如果身份证号或学号等），由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置：



**图14**

此时MySQL不得不为了将新记录插到合适位置而移动数据，甚至目标页面可能已经被回写到磁盘上而从缓存中清掉，此时又要从磁盘上读回来，这增加了很多开销，同时频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构，后续不得不通过OPTIMIZE TABLE来重建表并优化填充页面。

因此，只要可以，请尽量在InnoDB上采用自增字段做主键。

后记

这篇文章断断续续写了半个月，主要内容就是上面这些了。不可否认，这篇文章在一定程度上有纸上谈兵之嫌，因为我本人对MySQL的使用属于菜鸟级别，更没有太多数据库调优的经验，在这里大谈数据库索引调优有点大言不惭。就当是我个人的一篇学习笔记了。

其实数据库索引调优是一项技术活，不能仅仅靠理论，因为实际情况千变万化，而且MySQL本身存在很复杂的机制，如查询优化策略和各种引擎的实现差异等都会使情况变得更加复杂。但同时这些理论是索引调优的基础，只有在明白理论的基础上，才能对调优策略进行合理推断并了解其背后的机制，然后结合实践中不断的实验和摸索，从而真正达到高效使用MySQL索引的目的。

另外，MySQL索引及其优化涵盖范围非常广，本文只是涉及到其中一部分。如与排序（ORDER BY）相关的索引优化及覆盖索引（Covering index）的话题本文并未涉及，同时除B-Tree索引外MySQL还根据不同引擎支持的哈希索引、全文索引等等本文也并未涉及。如果有机会，希望再对本文未涉及的部分进行补充吧。

参考文献

[1] Baron Scbwartz等 著，王小东等 译；高性能MySQL（High Performance MySQL）；电子工业出版社，2010

[2] Michael Kofler 著，杨晓云等 译；MySQL5权威指南（The Definitive Guide to MySQL5）；人民邮电出版社，2006

[3] 姜承尧 著；MySQL技术内幕-InnoDB存储引擎；机械工业出版社，2011

[4] D Comer, Ubiquitous B-tree; ACM Computing Surveys (CSUR), 1979

[5] Codd, E. F. (1970). "A relational model of data for large shared data banks". Communications of the ACM, , Vol. 13, No. 6, pp. 377-387

[6] MySQL5.1参考手册 - <http://dev.mysql.com/doc/refman/5.1/zh/index.html>

本文基于[署名-非商业性使用 3.0](http://creativecommons.org/licenses/by-nc/3.0/deed.zh)许可协议发布，欢迎转载，演绎，但是必须保留本文的署名[张洋](http://leoo2sk.cnblogs.com/)（包含链接），且不得用于商业目的。如您有任何疑问或者授权方面的协商，请[与我联系](mailto:ericzhang.buaa@gmail.com)。

# INSERT DELAYED

## MySQL :: MySQL 5.6 Reference Manual :: 13.2.5.2 INSERT DELAYED Syntax

http://dev.mysql.com/doc/refman/5.6/en/insert-delayed.html

## MYSQL INSERT DELAYED - magic.xiao - 博客园

http://www.cnblogs.com/magic-xiao/p/4150791.html

INSERT DELAYED 语法

DELAYED选项是标准的MYSQL扩展。

如果你不需要等待INSERT语句的完成，就立刻返回，那么 INSERT DELAYED非常适合。

通常情况下：使用MYSQL记录日志，但是在这个过程中又有SELECT或者UPDATE语句花费非常长的时间。

当使用INSERT DELAYED的时候，客户端会立刻收到ok的反馈，被插入的行会进行排队，然后的等待table没有别的进程使用的时候，再将行写入table中。

另外一个使用INSERT DELAYED的好处是，从不同客户端的延迟插入的行，会绑定在一起，作为一个block，写入表中。这比起分开的客户端别分插入table要快的多。

注意：当table没有别的进程在使用的时候，INSERT DELAYED要比普通的 INSERT慢得多，原因是：服务器需要一些额外的负载来处理被延迟写入的行。举个例子：服务器需要一个独立的线程来维护每个表。

队列中的行数据是放在内存中的，也就意味着，如果是强制关闭进程（KILL -9），那么队列中的数据就会丢失。

使用INSERT DELAYED有以下几个约束：

INSERT DELAYED只适用于MYISAM，MEMORY表，ARCHIVE表，BLACKHOLE表。

如果INSERT DELAYED的表被LOCK TABLE那么，会发生错误。

# SQL

## @深入mysql "ON DUPLICATE KEY UPDATE" 语法的分析\_Mysql\_脚本之家

http://www.jb51.net/article/39255.htm

mysql "ON DUPLICATE KEY UPDATE" 语法  
如果在INSERT语句末尾指定了ON DUPLICATE KEY UPDATE，并且插入行后会导致在一个UNIQUE索引或PRIMARY KEY中出现重复值，则在出现重复值的行执行UPDATE；如果不会导致唯一值列重复的问题，则插入新行。   
例如，如果列 a 为 主键 或 拥有UNIQUE索引，并且包含值1，则以下两个语句具有相同的效果：

复制代码代码如下:

INSERT INTO TABLE (a,c) VALUES (1,3) ON DUPLICATE KEY UPDATE c=c+1;  
UPDATE TABLE SET c=c+1 WHERE a=1;

如果行作为新记录被插入，则受影响行的值显示1；如果原有的记录被更新，则受影响行的值显示2。   
这个语法还可以这样用:   
如果INSERT多行记录(假设 a 为主键或 a 是一个 UNIQUE索引列):

复制代码代码如下:

INSERT INTO TABLE (a,c) VALUES (1,3),(1,7) ON DUPLICATE KEY UPDATE c=c+1;

执行后, c 的值会变为 4 (第二条与第一条重复, c 在原值上+1).

复制代码代码如下:

INSERT INTO TABLE (a,c) VALUES (1,3),(1,7) ON DUPLICATE KEY UPDATE c=VALUES(c);

执行后, c 的值会变为 7 (第二条与第一条重复, c 在直接取重复的值7).   
注意：ON DUPLICATE KEY UPDATE只是MySQL的特有语法，并不是SQL标准语法！   
这个语法和适合用在需要 判断记录是否存在,不存在则插入存在则更新的场景.  
  
**INSERT INTO .. ON DUPLICATE KEY更新多行记录**如果在INSERT语句末尾指定了ON DUPLICATE KEY UPDATE，并且插入行后会导致在一个UNIQUE索引或PRIMARY KEY中出现重复值，则执行旧行UPDATE；如果不会导致唯一值列重复的问题，则插入新行。例如，如果列a被定义为UNIQUE，并且包含值1，则以下两个语句具有相同的效果：

复制代码代码如下:

INSERT INTO TABLE (a,b,c)   
VALUES (1,2,3) ON DUPLICATE KEY UPDATE c=c+1;  
UPDATE TABLE SET c=c+1 WHERE a=1;

如果行作为新记录被插入，则受影响行的值为1；如果原有的记录被更新，则受影响行的值为2。  
如果你想了解更多关于INSERT INTO .. ON DUPLICATE KEY的功能说明，详见MySQL参考文档：13.2.4. INSERT语法  
  
现在问题来了，如果INSERT多行记录， ON DUPLICATE KEY UPDATE后面字段的值怎么指定？要知道一条INSERT语句中只能有一个ON DUPLICATE KEY UPDATE，到底他会更新一行记录，还是更新所有需要更新的行。这个问题困扰了我很久了，其实使用VALUES()函数一切问题都解决了。  
  
举个例子，字段a被定义为UNIQUE，并且原数据库表table中已存在记录(2,2,9)和(3,2,1)，如果插入记录的a值与原有记录重复，则更新原有记录，否则插入新行：

复制代码代码如下:

INSERT INTO TABLE (a,b,c) VALUES   
(1,2,3),  
(2,5,7),  
(3,3,6),  
(4,8,2)  
ON DUPLICATE KEY UPDATE b=VALUES(b);

以上SQL语句的执行，发现(2,5,7)中的a与原有记录(2,2,9)发生唯一值冲突，则执行ON DUPLICATE KEY UPDATE，将原有记录(2,2,9)更新成(2,5,9)，将(3,2,1)更新成(3,3,1)，插入新记录(1,2,3)和(4,8,2)  
注意：ON DUPLICATE KEY UPDATE只是MySQL的特有语法，并不是SQL标准语法！

您可能感兴趣的文章:

* [MYSQL 批量替换之replace语法的使用详解](http://www.jb51.net/article/39382.htm)
* [SQL SERVER中各类触发器的完整语法及参数说明](http://www.jb51.net/article/40290.htm)
* [sqlServer 数据库语法大全](http://www.jb51.net/article/40292.htm)
* [JDBC SQL语法](http://www.jb51.net/article/48143.htm)
* [sqlserver存储过程语法详解](http://www.jb51.net/article/49274.htm)
* [浅析Mysql Join语法以及性能优化](http://www.jb51.net/article/50427.htm)
* [Oracle实现分页查询的SQL语法汇总](http://www.jb51.net/article/53608.htm)
* [详解Sql基础语法](http://www.jb51.net/article/77340.htm)

# Show variables

show variables like 'max\_connect\_errors';

vi /etc/my.cnf

systemctl restart mariadb

show variables like 'max\_connect\_errors';

## Tools, tips and links on optimizing mySQL | Drupal.org

https://www.drupal.org/node/85768

Here are some basic, but high impact ways to optimize MySQL for Drupal (there are much more sophisticated and expensive ways to speed up your database of course):

Note that if you are on a shared hosting plan then only your host will be able to tune MySQL since you won't have access to the my.cnf file. Also, I can only confirm these setting for MySQL 4.0.2 thru the latest 4.0.x version, but I think it would work for 5.x (maybe someone can confirm this and leave a comment...). Actually, it will work for below 4.0.2 I think as long as you add set-variable = before each line (see this page for more on [set-variable](http://dev.mysql.com/doc/refman/4.1/en/program-variables.html))

1. You can analyze your MySQL performance with the [MySQL Performance Tuning Primer Script](http://day32.com/MySQL/).

# cd /usr/local/src/

# wget http://day32.com/MySQL/tuning-primer.sh

# chmod u+x tuning-primer.sh

# ./tuning-primer.sh

It will ask for your database root user name and password and print out a list of recommendations. Was shocked to learned that on my VPS the cache was not even enabled - very helpful to know (Note: [Read up on the query cache](http://www.tocker.ca/2013/09/27/how-do-you-use-the-query-cache.html) before changing any values there)!

Another MySQL tuning script using perl:

<https://github.com/major/MySQLTuner-perl>

# git clone https://github.com/major/MySQLTuner-perl.git

# cd MySQLTuner-perl

# chmod u+x mysqltuner.pl

# perl mysqltuner.pl

Example of Recommendations (on dedicated server with 2GB of RAM)

General recommendations:

Run OPTIMIZE TABLE to defragment tables for better performance

MySQL started within last 24 hours - recommendations may be inaccurate

Enable the slow query log to troubleshoot bad queries

Set thread\_cache\_size to 4 as a starting value

Increase table\_cache gradually to avoid file descriptor limits

Read this before increasing table\_cache over 64: http://bit.ly/1mi7c4C

Variables to adjust:

query\_cache\_size (>= 8M)

thread\_cache\_size (start at 4)

table\_cache (> 400)

innodb\_buffer\_pool\_size (>= 365M)

2. Next open your my.cnf file in pico or some kind of proper code/text editor:

Depending on the memory resources you have available you'll want to paste in something like these examples (adjust up or down depending on how your system differs, of course):

For a setup with 500mb of RAM your my.cnf file may look like this:

[mysqld]

max\_connections = 150

max\_user\_connections = 150

key\_buffer = 36M

myisam\_sort\_buffer\_size = 64M

join\_buffer\_size = 2M

read\_buffer\_size = 2M

sort\_buffer\_size = 3M

table\_cache = 1024

thread\_cache\_size = 286

interactive\_timeout = 25

wait\_timeout = 1800

connect\_timeout = 10

max\_allowed\_packet = 16M

max\_connect\_errors = 1000

query\_cache\_limit = 1M

query\_cache\_size = 16M

query\_cache\_type = 1

tmp\_table\_size = 16M

innodb-flush-log-at-trx-commit=2

For a system with 256mb of ram it may look like this:

[mysqld]

max\_connections = 75

max\_user\_connections = 75

key\_buffer = 16M

myisam\_sort\_buffer\_size = 32M

join\_buffer\_size = 1M

read\_buffer\_size = 1M

sort\_buffer\_size = 2M

table\_cache = 1024

thread\_cache\_size = 286

interactive\_timeout = 25

wait\_timeout = 1000

connect\_timeout = 10

max\_allowed\_packet = 1M

max\_connect\_errors = 1000

query\_cache\_limit = 1M

query\_cache\_size = 16M

query\_cache\_type = 1

tmp\_table\_size = 16M

innodb-flush-log-at-trx-commit=2

Please note that every server configuration is going to differ and simply pasting these in may cause unexpected results.

3. Save your my.cnf file and restart mySQL. This can be done via your control panel or the command line (on some unixes: service mysqld restart otherwise /etc/rc.d/init.d/mysqld restart or /etc/init.d/mysqld restart)

Your new settings are now active and you can run the script from above again and see the difference in your results. After some experimenting I've found that it is useful to look at the script results right after making a change just to see if your modifications were recognized by the system and get the early returns from whether things were improved or not -- but, to get a truly accurate reading from the script you should check back in 24-48 hours after rebooting mysql (this is actually noted at the top of the script itself, but it doesn't really explain why) depending on your site traffic. Also, I've found that the way I've got Drupal set up it is particularly demanding in the tmp\_table\_size and table\_cache areas (e.g., you may want to bump up the number for both of these areas in the settings above)

If you'd like to read up on more about mySQL tuning I suggest taking a look at these resources:

[Tuning MySQL for Drupal](http://drupal.org/node/51263) (Drupal 7)

## MySQL :: MySQL 5.7 Reference Manual :: B.5.2.6 Host 'host\_name' is blocked

http://dev.mysql.com/doc/refman/5.7/en/blocked-host.html

# show processlist

## mysql show processlist命令 详解 - 毒菇小妖的窝 - 51Testing软件测试网 51Testing软件测试网-中国软件测试人的精神家园

http://www.51testing.com/html/96/110296-69546.html

processlist命令的输出结果显示了有哪些线程在运行，可以帮助识别出有问题的查询语句，两种方式使用这个命令。

1.      进入mysql/bin目录下输入mysqladmin processlist;

2.      启动[**mysql**](javascript:;)，输入show processlist;

如果有SUPER权限，则可以看到全部的线程，否则，只能看到自己发起的线程（这是指，当前对应的MySQL帐户运行的线程）。

得到数据形式如下（只截取了三条）：

mysql> show processlist;

+-----+-------------+--------------------+-------+---------+-------+----------------------------------+----------

| Id | User | Host           | db  | Command | Time| State    | Info

+-----+-------------+--------------------+-------+---------+-------+----------------------------------+----------

|207|root |192.168.0.20:51718 |mytest | Sleep   | 5   |        | NULL

|208|root |192.168.0.20:51719 |mytest | Sleep   | 5   |        | NULL

|220|root |192.168.0.20:51731 |mytest |Query   | 84  | Locked |

select bookname,culture,value,type from book where id=001

先简单说一下各列的含义和用途，第一列，id，不用说了吧，一个标识，你要kill一个语句的时候很有用。user列，显示单前用户，如果不是root，这个命令就只显示你权限范围内的sql语句。host列，显示这个语句是从哪个ip的哪个端口上发出的。呵呵，可以用来追踪出问题语句的用户。db列，显示这个进程目前连接的是哪个**[数据库](javascript:;" \t "_self)**。command列，显示当前连接的执行的命令，一般就是休眠（sleep），查询（query），连接（connect）。time列，此这个状态持续的时间，单位是秒。state列，显示使用当前连接的sql语句的状态，很重要的列，后续会有所有的状态的描述，请注意，state只是语句执行中的某一个状态，一个sql语句，已查询为例，可能需要经过copying to tmp table，Sorting result，Sending data等状态才可以完成，info列，显示这个sql语句，因为长度有限，所以长的sql语句就显示不全，但是一个判断问题语句的重要依据。

这个命令中最关键的就是state列，mysql列出的状态主要有以下几种：

　　Checking table  
　正在检查数据表（这是自动的）。  
　Closing tables  
　正在将表中修改的数据刷新到磁盘中，同时正在关闭已经用完的表。这是一个很快的操作，如果不是这样的话，就应该确认磁盘空间是否已经满了或者磁盘是否正处于重负中。  
　Connect Out  
　复制从服务器正在连接主服务器。  
　Copying to tmp table on disk  
　由于临时结果集大于tmp\_table\_size，正在将临时表从内存存储转为磁盘存储以此节省内存。  
　Creating tmp table  
　正在创建临时表以存放部分查询结果。  
　deleting from main table  
　服务器正在执行多表删除中的第一部分，刚删除第一个表。  
　deleting from reference tables  
　服务器正在执行多表删除中的第二部分，正在删除[**其他**](javascript:;)表的记录。  
　Flushing tables  
　正在执行FLUSH TABLES，等待其他线程关闭数据表。  
　Killed  
　发送了一个kill请求给某线程，那么这个线程将会检查kill标志位，同时会放弃下一个kill请求。MySQL会在每次的主循环中检查kill标志位，不过有些情况下该线程可能会过一小段才能死掉。如果该线程程被其他线程锁住了，那么kill请求会在锁释放时马上生效。  
　Locked  
　被其他查询锁住了。  
　Sending data  
　正在处理SELECT查询的记录，同时正在把结果发送给客户端。  
　Sorting for group  
　正在为GROUP BY做排序。  
　Sorting for order  
　正在为ORDER BY做排序。  
　Opening tables  
　这个过程应该会很快，除非受到其他因素的干扰。例如，在执ALTER TABLE或LOCK TABLE语句行完以前，数据表无法被其他线程打开。正尝试打开一个表。  
　Removing duplicates  
　正在执行一个SELECT DISTINCT方式的查询，但是MySQL无法在前一个阶段优化掉那些重复的记录。因此，MySQL需要再次去掉重复的记录，然后再把结果发送给客户端。  
　Reopen table  
　获得了对一个表的锁，但是必须在表结构修改之后才能获得这个锁。已经释放锁，关闭数据表，正尝试重新打开数据表。  
　Repair by sorting  
　修复指令正在排序以创建索引。  
　Repair with keycache  
　修复指令正在利用索引缓存一个一个地创建新索引。它会比Repair by sorting慢些。  
　Searching rows for update  
　正在讲符合条件的记录找出来以备更新。它必须在UPDATE要修改相关的记录之前就完成了。  
　Sleeping  
　正在等待客户端发送新请求.  
　System lock  
　正在等待取得一个外部的系统锁。如果当前没有运行多个mysqld服务器同时请求同一个表，那么可以通过增加--skip-external-locking参数来禁止外部系统锁。  
　Upgrading lock  
　INSERT DELAYED正在尝试取得一个锁表以插入新记录。  
　Updating  
　正在搜索匹配的记录，并且修改它们。  
　User Lock  
　正在等待GET\_LOCK()。  
　Waiting for tables  
　该线程得到通知，数据表结构已经被修改了，需要重新打开数据表以取得新的结构。然后，为了能的重新打开数据表，必须等到所有其他线程关闭这个表。以下几种情况下会产生这个通知：FLUSH TABLES tbl\_name, ALTER TABLE, RENAME TABLE, REPAIR TABLE, ANALYZE TABLE,或OPTIMIZE TABLE。  
　waiting for handler insert  
　INSERT DELAYED已经处理完了所有待处理的插入操作，正在等待新的请求。  
　大部分状态对应很快的操作，只要有一个线程保持同一个状态好几秒钟，那么可能是有问题发生了，需要检查一下。  
　还有其他的状态没在上面中列出来，不过它们大部分只是在查看服务器是否有存在错误是才用得着。

mysql手册里有所有状态的说明，链接如下：<http://dev.mysql.com/doc/refman/5.0/en/general-thread-states.html>

中文说明取自<http://www.linuxpk.com/5747.html>

## SHOW PROCESSLIST - MariaDB Knowledge Base

https://mariadb.com/kb/en/mariadb/show-processlist/

## MySQL :: MySQL 5.1 Reference Manual :: 13.7.5.31 SHOW PROCESSLIST Syntax

https://dev.mysql.com/doc/refman/5.1/en/show-processlist.html