**1. 缓存和页面静态化**

**1.1 缓存**

数据量大这个问题最直接的解决方案就是使用缓存，缓存就是将数据库中获取的结果暂时保存起来在，在下次使用的时候无需从数据库中提取，这样可以大大降低数据库压力。常用的缓存框架有Ehcache、Memcache 和 Redis等。

不过缓存也不是什么情况都适用，它主要用于数据变化不是很频繁的情况。而且如果是定期失效（数据修改时不失效）的失效机制，实时性要求也不能太高，因为这样缓存中的数据和真实数据可能会不一致。

**2.1 页面静态化**

页面静态化是将程序最后生成的页面保存起来，以后就不用每次都调用重新生成页面了。页面静态化同时对数据量大和并发量高两大问题都有好处。

页面静态化可以在程序中使用模版技术生成，如常用的Freemarker 和 Velocity 都可以根据模版生成静态页面。另外也可以使用缓存服务器在应用服务器的上一层生成页面，如可以使用Squid，另外Nginx 也提供了响应的功能。

**2. 数据库优化**

要解决数据量大的问题，是避不开数据库优化的。数据库优化是不增加硬件的情况下提高处理效率，是一种用技术换金钱的方式。以下是常用的数据库优化方法。

**2.1 表结构优化**

表结构优化是数据库中最基础也是最重要的，如果表结构优化得不合理，就可能导致严重的性能问题，具体怎么设计更合理也没有固定不变的准则，需要根据实际情况具体处理。

**2.2 SQL 语句优化**

sql语句优化也是非常重要的，基础的sql 优化是语法层面的优化，不过更重要的是处理逻辑的优化，这也需要根据实际情况具体处理，而且要和索引缓存等配合食用。sql 优化有一个通用的做法就是，首先要将涉及大数据业务的sql 语句，通过仔细分析日志信息，和不同条件的执行时间，找出需要优化的语句和其中的问题，然后再有的放矢地优化。而不是不分重点的对每条语句都花同样的事件和精力优化。

**2.3 分区**

当一张表中的数据量变多的时候操作速度就变慢了，容易想到的解决方法就是分表，但是分表操作起来就比较麻烦。其实在常用数据库中可以不分表而达到跟分表类似的效果，那就是分区。

分区就是将一张表中的数据按照一定的规则分到不同的区进行保存，在查询数据时如果范围在同一个区内哪么可以只对一个区的数据进行操作。这样操作的数据量更少，速度更快。

**2.4 分表**

如果一张表中的数据可以分为几种固定不变的类型，而且如果同时对多种类型共同操作的情况不多，哪么都可以通过分表来处理。

分表方法分为两种，横切 和纵切。

**横切：**

假设数据库表中的数据有三个状态，处理、解决、关闭。由于数据量非常的大所以就可以将数据分别存在三个表中。第一个表中保存处理状态的数据，第二个表保存 解决状态的数据，第三个表保存关闭状态的数据，并且对每个表进行分区。由于报表一般都是按月份、季度、半年、和年来做的，所以分区也按月份，每一个月做一个分区。这样就可以大大的提高处理和统计速度哦

**竖切：**

如果一个表的操作频率很高们在增删改其中一部分字段数据的同时另一部分字段也可能被操作，而且（主要指查询）用不到被增删改的字段，哪么就可以吧不同类型的地段分别保存到不同的表中，这样可以减少操作时锁定数据的范围。不过这样分表后需要查询完整数据就得多表操作了。

**2.5 索引优化**

索引的大致原理是在数据发生变化（增删改）的时候就预先指定字段排序后保存到一个类似表的结构中，这样在查询索引字段为条件的记录时就可以很快的从索引中找到对应记录的指针并从表中获取到记录，这样速度就快多了。不过索引也是一把双刃剑，它在提高查询速度的同时也降低了增删改查的速度，因为每次数据的表话都需要更新相应的索引。

**2.6 使用存储过程代替操作**

在操作过程复杂而且调用频率高的业务中，可以通过使用存储过程代替直接操作来提高效率，因为存储过程主需要编译一次，而且可以在一个存储过程里面做一些复杂的操作。

**3. 分离活跃数据**

有些数据总数据量非常大，但是活跃数据并不多，这种情况就可以将活跃数据单独保存起来从而提高处理效率。

比如说网站的用户，将经常登录的一部分活跃用户信息存在一张表，将不活跃的用户存在另外一张表。（可以跑一个定时任务将不经常登录的用户转移到不活跃用户表）当用户登录时先从活跃用户的表中查询。查询不到后再去不活跃用户的表中查询。这样就可以提高查询的效率。

**4. 批量读取和延迟修改**

批量读取和延迟修改的原理是通过减少操作的次数来提高效率，如果使用的恰当，效率将呈数量级提升。

批量读取是将多次的查询合并到一次中进行。   
延迟修改主要针对高并发而且频繁修改（包括新增）的数据，如一些统计数据。可以先将需要修改的数据暂时保存到缓存中，然后定时将缓存中的数据保存到数据库中，程序读取数据可以同时读取数据库中的数据和缓存的数据。（这种方式下如果保存缓存的机器出现了故障可能会丢失数据。）所以如果是重要数据就需要做一些特殊处理。

**5. 读写分离**

读写分离的本质是对数据库进行集群，这样就可以在高并发的情况下将数据库的操作分配到多个数据库服务器去处理从而降低了单台服务器的压力。不过由于数据库的特殊性－－每台服务器所保存的数据都需要一致，所以数据同步就成了数据库集群中最核心的问题了。

一般的解决方案是：将写操作交给专门的一台服务器，这台专门负责写操作的服务器叫做主服务器。当主服务器写入（增删改）数据后从底层同步到别的服务器（从服务器），读取数据的时候到从服务器读取，从服务器可以有多台，这样就可以实现读写分离了。并且将请求分配到多个服务器处理。如果从服务器太多，主服务器同步数据时，先同步到一部分从服务器，同步过数据的从服务器再将数据同步到另外一部分没有同步数据的从服务器。

简单的数据同步方式可以采用数据库的热备份功能，不过读取到的数据可能会存在一定的滞后性。既然是集群就会涉及到负载均衡问题，负载均衡和读写分离操作一般采用专门的程序处理，而且对应用系统来说是透明的。

**6. 分布式数据库**

分布式数据库是将不同的表放到不同的数据库中然后放到不同的服务器。这样需要调用多个表，则可以让多台服务器同时处理，从而提高处理速度。

分布式数据库是解决单个请求本身就非常复杂的问题，它可以将单个请求分配到多个服务器，使用分布式后的每个节点还可以同时使用读写分离，从来组成多个节群。

使用分布式数据库有很多的问题需要解决，如分布式事务处理、多表查询等。

分布式的另外一种使用思路是将不同的业务数据表保存到不同的节点，让不同的业务调用不同的数据库，这样的做法其实就是和集群一样起分流作用，不过这种做法就不用同步数据了。

**7. Nosql 和 hadoop**

Nosql 是近年来发展非常迅速的一项技术，他的核心是非结构化。Nosql 通过多个块存储数据，操作大数据的速度非常快。

**高并发的解决方案**

除了数据量大，另外一个常见的问题就是并发量高，很多架构就是针对这个问题设计出来的。

**1.应用和静态资源分离**

刚开始的时候应用和静态资源是保存在一起的，当并发量达到一定程度的时候就需要将静态资源保存到专门的服务器中，静态资源主要包括图片、视频、js、css和一些资源文件等，这些文件因为没有状态所以分离比较简单，直接存放到响应的服务器就可以了，一般会使用专门的域名去访问。   
通过不同的域名可以让浏览器直接访问资源服务器而不需要再访问应用服务器了。架构图如下：

**2.页面缓存**

页面缓存是将应用生成的页面缓存起来，这样就不需要每次都生成页面了，从而可以节省大量的CPU资源，如果将缓存的页面放到内存中速度就更快了。如果使用Nginx服务器就可以使用它自带的缓存功能，当然也可以使用专门的Squid 服务器。页面缓存的默认失效机制一班都是按缓存时间处理的，当然也可以在修改数据之后手动让相应的缓存失效。   
页面缓存主要是使用在数据很少发生变化的页面，但是很多页面是大部分数据都很少发生变化，而其中很少一部分数据变化频率却非常高，比如说一个显示文章的页面，正常来说完全可以静态化，但是如果文章后面有“顶”和“踩”的功能而且显示的有响应的数量，这个数据的变化频率就比较高了，这就会影响静态化。这个问题可以用先生成静态页面然后使用Ajax来读取并修改响应的数据，这样就可以一举两得来，既可以使用页面缓存也可以实时显示一些变化频率高的数据来。

**3.集群与分布式**

集群是每台服务器都具有相同的功能，处理请求时调用那台服务器都可以，主要起分流作用。

分布式是将不同的业务放到不同的服务器中，处理一个请求可能需要用到多台服务器，这样就可以提高一个请求的处理速度，而且集群和分布式也可以同时使用。

集群有两个方式：一种是在静态资源集群。另一种是应用程序集群。静态资源集群比较简单。应用程序集群在处理过程中最核心的问题就是Session 同步问题。

Session 同步有两种处理方式：一种是在Session 发生变化后自动同步到其他服务器，另一种就是用个程序统一管理Session。所有集群的服务器都使用同一个Session，Tomcat 默认使用就是第一种方式，通过简单的配置就可以实现，第二种方式可以使用专门的服务器安装Mencached等高效的缓存程序统一来管理session，然后再应用程序中通过重写Request并覆盖getSession 方法来获取制定服务器中的Session。

对于集群来说还有一个核心的问题就是负载均衡，也就是接收到一个请求后具体分配到那个服务器去处理的问题，这个问题可以通过软件处理也可以使用专门的硬件（如：F5）解决。

**4. 反向代理**

反向代理指的是客户端直接访问的服务器并不真正提供服务，它从别的服务器获取资源然后将结果返回给用户。

**4.1 反向代理服务器和代理服务器的区别**

代理服务器的作用是代我门获取想要的资源然后将结果返回给我们，所要获取的资源是我门主动告诉代理服务器的，比如，我门想访问Facebook，但是直接访问不了，这时就可以让代理服务器访问，然后将结果返回给我们。

反向代理服务器是我门正常访问一台服务器的时候，服务器自己去调用了别的服务器资源并将结果返回给我们，我门自己并不知道。

代理服务器是我们主动使用的，是为我们服务的，他不需要有自己的域名；反向代理服务器是服务器自己试用的，我门并不知道，它有自己的域名，我门访问它和访问正常的网址没有任何区别。

反向代理服务器主要有三个作用：   
1. 可以作为前端服务器跟实际处理请求的服务器集成；   
2. 可以做负载均衡   
3. 转发请求，比如说可以将不同类型的资源请求转发到不同的服务器去处理。

**5. CDN**

cdn其实是一种特殊的集群页面缓存服务器，他和普通集群的多台页面缓存服务器相比，主要是它存放的位置和分配请求的方式有点特殊。CDN 服务器是分布在全国各地的，当接收到用户请求后会将请求分配到最合适的CDN服务器节点获取数据。比如联通的用户分配到联通的节点，上海的用户分配到上海的节点。

CDN的每个节点其实就是一个页面缓存服务器，如果没有请求资源的缓存就会从主服务器获取，否则直接返回缓存的页面。

CDN分配请求（负载均衡）的方式是用专门的CDN域名解析服务器在解析域名的时候就分配好的。一般的做法是在ISP哪里试用CNAME将域名解析到一个特定的域名，然后再将解析到的那个域名用专门的CDN服务器解析道相应的CDN节点。如图。

第二步访问CDN的DNS服务器是应为CNAME记录的目标域名使用NS记录指向了CDN的DNS服务器。CDN的每个节点可能也是集群了多台服务器。

**6. 底层的优化**

前面说的所有都是架构都是建立在最前面介绍的基础结构之上的。很多地方都需要通过网络传输数据，如果可以加快网络传输的速度，那将会让整个系统得到改善。

**7. 小结**

网站架构的整个演变过程主要是围绕大数据和高并发这两个问题展开的，解决方案主要分为使用缓存和多资源两种类型。多资源主要指多存储（包括多内存）、多CPU和多网络，对于多资源来说又可以分为单个资源处理一个完整的请求和多个资源合作处理一个请求两种类型，如多存储和多CPU中的集群和分布式，多网络中的CDN和静态资源分离。理解了整个思路之后就抓住了架构演变的本质，而且自己可能还可以设计出更好的架构。

**1，MySQL数据库引擎**

**引擎：存储、处理和保护数据的方式。**

在缺省情况下，MYSQL支持三个引擎：ISAM、MYISAM和HEAP。另外两种类型INNODB和BERKLEY（BDB）。

**ISAM**：ISAM是一个定义明确且历经时间考验的数据表格管理方法，它在设计之时就考虑到 数据库被查询的次数要远大于更新的次数。因此，ISAM执行读取操作的速度很快，而且不占用大量的内存和存储资源。ISAM的两个主要不足之处在于，它不 支持事务处理，也不能够容错。

**MyISAM**：MyISAM是MySQL的ISAM扩展格式和缺省的数据库引擎。除了提供ISAM里所没有的索引和字段管理的大量功能，MyISAM还使用一种表格锁定的机制，来优化多个并发的读写操作，其代价是你需要经常运行OPTIMIZE TABLE命令，来恢复被更新机制所浪费的空间。

**HEAP**：HEAP允许只驻留在内存里的临时表格。驻留在内存里让HEAP要比ISAM和MYISAM都快，但是它所管理的数据是不稳定的，而且如果在关机之前没有进行保存，那么所有的数据都会丢失。在数据行被删除的时候，HEAP也不会浪费大量的空间。HEAP表格在你需要使用SELECT表达式来选择和操控数据的时候非常有用。要记住，在用完表格之后就删除表格。

**InnoDB**：尽管要比ISAM和 MyISAM引擎慢很多，但是InnoDB包括了对事务处理和外来键的支持，这两点都是前两个引擎所没有的。

**2，Mysql的索引类型**

目前主要有以下几种索引类型：FULLTEXT，HASH，BTREE，RTREE。

**FULLTEXT**：即为全文索引，目前只有MyISAM引擎支持。全文索引并不是和MyISAM一起诞生的，它的出现是为了解决WHERE name LIKE “%word%"这类针对文本的模糊查询效率较低的问题。

**hash索引**可以一次定位，不需要像树形索引那样逐层查找,因此具有极高的效率。

（1）Hash 索引仅仅能满足"=","IN"和"<=>"查询，不能使用范围查询。   
（2）Hash 索引无法被用来避免数据的排序操作。   
（3）Hash 索引不能利用部分索引键查询。   
（4）Hash 索引在任何时候都不能避免表扫描。

（5）Hash 索引遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高。

**BTREE索引：**BTREE在MyISAM里的形式和Innodb稍有不同

在 Innodb里，有两种形态：一是primary key形态，其leaf node里存放的是数据，而且不仅存放了索引键的数据，还存放了其他字段的数据。二是secondary index，其leaf node和普通的BTREE差不多，只是还存放了指向主键的信息.

而在MyISAM里，主键和其他的并没有太大区别。不过和Innodb不太一样的地方是在MyISAM里，leaf node里存放的不是主键的信息，而是指向数据文件里的对应数据行的信息.

**RTREE**：在mysql很少使用，仅支持geometry数据类型，支持该类型的存储引擎只有MyISAM、BDb、InnoDb、NDb、Archive几种。空间数据库包含了很多 tuple ，这些 tuple 代表了空间数据。每一个 tuple 有一个唯一的标识（ ID ），这样可以访问到某个 tuple 。 RTree 中叶节点包含的索引记录的形式是 (I , tuple-identifier), 这里的 id 指向数据库中的一个 tuple 。 I 表示一个 n 维的矩形，这个矩形是一个 bounding box ，用来涵盖被索引的空间数据对象。

**3，一个数据库 连表查询 排序 截取**

SELECT \* FROM Customers c JOIN Products p WHERE c.id = p.id ORDER BY c.id DESC LIMIT 3,2; //4-5行

**4，索引失效**

1. 在Where子句中使用函数。
2. 在Where子句中使用IS NULL或者IS NOT NULL。
3. 用or分割开的条件，如果or前的条件中的列有索引，但是or后面的列中没有索引，那么涉及的索引都不会被用到。
4. 复合索引下，如果查询条件不包含最左边的部分，即不满足最左原则，是不会利用到复合索引的。

**5,mysql添加索引**

1. ALTER TABLE `table\_name` ADD PRIMARY KEY ( `column` )
2. ALTER TABLE `table\_name` ADD UNIQUE ( `column` )
3. ALTER TABLE `table\_name` ADD INDEX index\_name ( `column` )
4. ALTER TABLE `table\_name` ADD FULLTEXT ( `column`)
5. ALTER TABLE `table\_name` ADD INDEX index\_name ( `column1`, `column2`, `column3` )

**6,事务**

事务隔离级别

隔离级别是指若干个并发的事务之间的隔离程度。TransactionDefinition 接口中定义了五个表示隔离级别的常量：

TransactionDefinition.ISOLATION\_DEFAULT：这是默认值，表示使用底层数据库的默认隔离级别。对大部分数据库而言，通常这值就是TransactionDefinition.ISOLATION\_READ\_COMMITTED。

TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED：该隔离级别表示一个事务可以读取另一个事务修改但还没有提交的数据。该级别不能防止脏读和不可重复读，因此很少使用该隔离级别。

TransactionDefinition.ISOLATION\_READ\_COMMITTED：该隔离级别表示一个事务只能读取另一个事务已经提交的数据。该级别可以防止脏读，这也是大多数情况下的推荐值。

TransactionDefinition.ISOLATION\_REPEATABLE\_READ：该隔离级别表示一个事务在整个过程中可以多次重复执行某个查询，并且每次返回的记录都相同。即使在多次查询之间有新增的数据满足该查询，这些新增的记录也会被忽略。该级别可以防止脏读和不可重复读。

TransactionDefinition.ISOLATION\_SERIALIZABLE：所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

事务传播行为

所谓事务的传播行为是指，如果在开始当前事务之前，一个事务上下文已经存在，此时有若干选项可以指定一个事务性方法的执行行为。在TransactionDefinition定义中包括了如下几个表示传播行为的常量：

TransactionDefinition.PROPAGATION\_REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。

TransactionDefinition.PROPAGATION\_REQUIRES\_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起。

TransactionDefinition.PROPAGATION\_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。

TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。

TransactionDefinition.PROPAGATION\_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。

TransactionDefinition.PROPAGATION\_MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。

TransactionDefinition.PROPAGATION\_NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION\_REQUIRED。

**7，数据库锁**

**共享锁**(S)表示对数据进行读操作。因此多个事务可以同时为一个对象加共享锁。（如果试衣间的门还没被锁上，顾客都能够同时进去参观）

产生共享锁的sql：select \* from ad\_plan lock in share mode;

**排他锁**表示对数据进行写操作。如果一个事务对对象加了排他锁，其他事务就不能再给它加任何锁了。(某个顾客把试衣间从里面反锁了，其他顾客想要使用这个试衣间，就只有等待锁从里面给打开了)

产生排他锁的sql： select \* from ad\_plan for update;

根据粒度，MySQL有三种锁的级别：页级、表级、行级。

8，Mysql查看正在执行的sql语句

my.cnf:

general\_log = 1

general\_log\_file = /var/log/mysql/general\_sql.log

1. Mysql底层的数据结构：

B+树更适合范围查询，两个指针一个指针指向根节点，另一个指针指向最左的孩子节点

范围查询时，B树找到一个节点要进行中序遍历，而B+树直接遍历链表即可

聚集索引与非聚集索引：

**聚集索引**

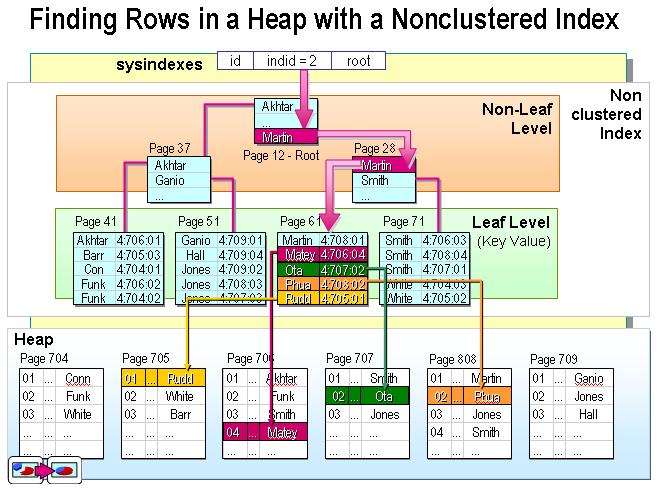
　　一种索引，该索引中键值的逻辑顺序决定了表中相应行的物理顺序。   
　　聚集索引确定表中数据的物理顺序。聚集索引类似于电话簿，后者按姓氏排列数据。由于聚集索引规定数据在表中的物理存储顺序，因此一个表只能包含一个聚集索引。但该索引可以包含多个列（组合索引），就像电话簿按姓氏和名字进行组织一样。   
　　　　   
　　　　 聚集索引对于那些经常要搜索范围值的列特别有效。使用聚集索引找到包含第一个值的行后，便可以确保包含后续索引值的行在物理相邻。例如，如果应用程序执行 的一个查询经常检索某一日期范围内的记录，则使用聚集索引可以迅速找到包含开始日期的行，然后检索表中所有相邻的行，直到到达结束日期。这样有助于提高此 类查询的性能。同样，如果对从表中检索的数据进行排序时经常要用到某一列，则可以将该表在该列上聚集（物理排序），避免每次查询该列时都进行排序，从而节 省成本。 

　　　　 当索引值唯一时，使用聚集索引查找特定的行也很有效率。例如，使用唯一雇员 ID 列 emp\_id 查找特定雇员的最快速的方法，是在 emp\_id 列上创建聚集索引或 PRIMARY KEY 约束。

**非聚集索引**

　　一种索引，该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同。

索引是通过二叉树的数据结构来描述的，我们可以这么理解聚簇索引：索引的叶节点就是数据节点。而非聚簇索引的叶节点仍然是索引节点，只不过有一个指针指向对应的数据块。如下图：



举例子理解：

其实，我们的汉语字典的正文本身就是一个聚集索引。比如，我们要查“安”字，就会很自然地翻开字典的前几页，因为“安”的拼音是“an”，而按照拼音排序汉字的字典是以英文字母“a”开头并以“z”结尾的，那么“安”字就自然地排在字典的前部。如果您翻完了所有以“a”开头的部分仍然找不到这个字，那么就说明您的字典中没有这个字；同样的，如果查“张”字，那您也会将您的字典翻到最后部分，因为“张”的拼音是“zhang”。也就是说，字典的正文部分本身就是一个目录，您不需要再去查其他目录来找到您需要找的内容。我们把这种正文内容本身就是一种按照一定规则排列的目录称为“聚集索引”。  
    　　如果您认识某个字，您可以快速地从自动中查到这个字。但您也可能会遇到您不认识的字，不知道它的发音，这时候，您就不能按照刚才的方法找到您要查的字，而需要去根据“偏旁部首”查到您要找的字，然后根据这个字后的页码直接翻到某页来找到您要找的字。但您结合“部首目录”和“检字表”而查到的字的排序并不是真正的正文的排序方法，比如您查“张”字，我们可以看到在查部首之后的检字表中“张”的页码是672页，检字表中“张”的上面是“驰”字，但页码却是63页，“张”的下面是“弩”字，页面是390页。很显然，这些字并不是真正的分别位于“张”字的上下方，现在您看到的连续的“驰、张、弩”三字实际上就是他们在非聚集索引中的排序，是字典正文中的字在非聚集索引中的映射。我们可以通过这种方式来找到您所需要的字，但它需要两个过程，先找到目录中的结果，然后再翻到您所需要的页码。我们把这种目录纯粹是目录，正文纯粹是正文的排序方式称为“非聚集索引”。

每个表只能有一个聚集索引，因为目录只能按照一种方法进行排序。

|  |  |  |
| --- | --- | --- |
| 动作描述 | 使用聚集索引 | 使用非聚集索引 |
| 列经常被分组排序 | 应 | 应 |
| 返回某范围内的数据 | 应 | 不应 |
| 一个或极少不同值 | 不应 | 不应 |
| 小数目的不同值 | 应 | 不应 |
| 大数目的不同值 | 不应 | 应 |
| 频繁更新的列 | 不应 | 应 |
| 外键列 | 应 | 应 |
| 主键列 | 应 | 应 |
| 频繁修改索引列 | 不应 | 应 |

事实上，我们可以通过前面聚集索引和非聚集索引的定义的例子来理解上表。如：返回某范围内的数据一项。比如您的某个表有一个时间列，恰好您把聚合索引建立在了该列，这时您查询2004年1月1日至2004年10月1日之间的全部数据时，这个速度就将是很快的，因为您的这本字典正文是按日期进行排序的，聚类索引只需要找到要检索的所有数据中的开头和结尾数据即可；而不像非聚集索引，必须先查到目录中查到每一项数据对应的页码，然后再根据页码查到具体内容。

一些误区：

1、主键就是聚集索引  
    　　这种想法笔者认为是极端错误的，是对聚集索引的一种浪费。虽然SQL SERVER默认是在主键上建立聚集索引的。  
    　　通常，我们会在每个表中都建立一个ID列，以区分每条数据，并且这个ID列是自动增大的，步长一般为1。我们的这个办公自动化的实例中的列Gid就是如此。此时，如果我们将这个列设为主键，SQL SERVER会将此列默认为聚集索引。这样做有好处，就是可以让您的数据在数据库中按照ID进行物理排序，但笔者认为这样做意义不大。  
    　　显而易见，聚集索引的优势是很明显的，而每个表中只能有一个聚集索引的规则，这使得聚集索引变得更加珍贵。  
    　　从我们前面谈到的聚集索引的定义我们可以看出，使用聚集索引的最大好处就是能够根据查询要求，迅速缩小查询范围，避免全表扫描。在实际应用中，因为 ID号是自动生成的，我们并不知道每条记录的ID号，所以我们很难在实践中用ID号来进行查询。这就使让ID号这个主键作为聚集索引成为一种资源浪费。其次，让每个ID号都不同的字段作为聚集索引也不符合“大数目的不同值情况下不应建立聚合索引”规则；当然，这种情况只是针对用户经常修改记录内容，特别是索引项的时候会负作用，但对于查询速度并没有影响。

2、只要建立索引就能显著提高查询速度

3、把所有需要提高查询速度的字段都加进聚集索引，以提高查询速度

索引降低了插入、删除、修改等维护任务的速度(虽然索引可以提高查询速度，但是它们也会导致数据库系统更新数据的性能下降，**因为大部分数据更新需要同时更新索引**)

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。

局部性原理与磁盘预读是聚集索引快的原因

B+tree中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)。**

数据库基础：

1 视图是一种基于数据表的一种虚表；视图建立在已有表的基础上, 视图赖以建立的这些表称为基表；向视图提供数据内容的语句为 SELECT 语句,可以将视图理解为存储起来的 SELECT 语句；视图没有存储真正的数据，真正的数据还是存储在基表中；程序员虽然操作的是视图，但最终视图还会转成操作基表

有的时候，我们可能只关系一张数据表中的某些字段，而另外的一些人只关系同一张数据表的某些字段...那么把全部的字段都都显示给他们看，这是不合理的。我们应该做到：他们想看到什么样的数据，我们就给他们什么样的数据...一方面就能够让他们只关注自己的数据，另一方面，我们也保证数据表一些保密的数据不会泄露出来...

我们在查询数据的时候，常常需要编写非常长的SQL语句，几乎每次都要写很长很长....上面已经说了，视图就是基于查询的一种虚表，也就是说，视图可以将查询出来的数据进行封装。。。那么我们在使用的时候就会变得非常方便...

视图主要用于简化检索，保护数据，并不用于更新，而且大部分视图都不可以更新

作用：

1. 简化了操作，把经常使用的数据定义为视图。通常需要写一大长串SQL语句时，不方便
2. 安全性，用户只能查询和修改能看到的数据。我们可以将基表中重要的字段信息，可以不通过视图给用户，视图是动态的数据的集合，数据是随着基表的更新而更新。同时，用户对视图不可以随意的更改和删除，可以保证数据的安全性。
3. 相当于把程序和真正的表分割开来

缺点就是性能差：转换需要一定的时间

2 存储过程：

如果你接触过其他的编程语言，那么就好理解了，存储过程就像是方法一样。

竟然他是方法那么他就有类似的方法名，方法要传递的变量和返回结果，所以存储过程有存储过程名有存储过程参数也有返回值。

在运行存储过程前，数据库已对其进行了语法和句法分析，并给出了优化执行方案。这种已经编译好的过程可极大地改善SQL语句的性能。

存储过程可以分为系统存储过程、扩展存储过程和用户自定义的存储过程

**系统存储过程：**

我们先来看一下系统存储过程，系统存储过程由系统定义，主要存放在MASTER数据库中，名称以"SP"开头或以"XP"开头。尽管这些系统存储过程在MASTER数据库中，

但我们在其他数据库还是可以调用系统存储过程。有一些系统存储过程会在创建新的数据库的时候被自动创建在当前数据库中。

语法：

CREATE PROC[EDURE] 存储过程名

@参数1 [数据类型]=[默认值] [OUTPUT]

@参数2 [数据类型]=[默认值] [OUTPUT]

AS

SQL语句

EXEC 过程名[参数]

create procedure procedure\_proc\_GetoffinfoById --(存储过程名)

@Id int--(参数名 参数类型)

as select Name from dbo.Office\_Info where Id=@Id--(sql语句)

exec procedure\_proc\_GetoffinfoById 2--(存储过程名称之后,空格加上参数,多个参数中间以逗号分隔)  
注:参数赋值是,第一个参数可以不写参数名称,后面传入参数,需要明确传入的是哪个参数名称

* **不再需要一张表的时候，用drop**
* **想删除部分数据行时候，用delete，并且带上where子句**
* **保留表而删除所有数据的时候用truncate**

事务主要是用来解决并发问题的，是并发和锁的结合体。如A用户转账给B用户，事务就是对整个过程加锁，只能有一个线程参与整个过程，而别的线程只能看到转账前后的状态，也就是保证了一致性。

**事务**：事务简单来说：**一个Session中所进行所有的操作，要么同时成功，要么同时失败**

**ACID — 数据库事务正确执行的四个基本要素**

假设一个情景，Bob有100元，Smith有0元，Bob转账100给Smith，

原子性:要么Bob有100块，Smith有0块，要么Bob有0块，Smith有100块。原子性的语意只保证记了一个回滚段，这个回滚段能回滚到之前的版本

一致性：在转账的过程中，另一个事务对Smith账户进行了修改，当前事务回滚时会造成数据不一致的情况，此时需要加锁，保证一致性。

隔离性：以性能为理由，对一致性的破坏

持久性：事务完成以后，该事务对数据库所做的更改便持久的保存在数据库之中

* 包含：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。

举个例子:**A向B转账，转账这个流程中如果出现问题，事务可以让数据恢复成原来一样【A账户的钱没变，B账户的钱也没变】。**

### 事务隔离级别

数据库定义了4个隔离级别：

1. Serializable【可避免脏读，不可重复读，虚读】可串行化，使用排它锁，保证事务的强一致性，但是性能很低，因为事务都串行了
2. Repeatable read【可避免脏读，不可重复读】读写锁，分为两种，根据当对一个事务单元加了一个读锁的时候，如果有新的写进来，这个读要不要放开，让写进去。当读锁不能被写锁升级时，只能做到读读可并行，即可重读级别。读锁可以被写锁升级，如当2个读针对一个事务单元加了一个读锁的时候，一个新的写来了，允许写请求将读锁升级为写请求，这样可以做到读读并行，读写并行（写读还不能哦）
3. Read committed【可避免脏读】读写并行，这样会出现第二次读的数据和第一次不一样
4. Read uncommitted【级别最低，什么都避免不了】未提交读这个隔离级别就是只加写锁，不加读锁，这样可以做到读读并行，读写并行，写读并行

脏读：**一个事务读取到另外一个事务未提交的数据**

例子：A向B转账，**A执行了转账语句，但A还没有提交事务，B读取数据，发现自己账户钱变多了**！B跟A说，我已经收到钱了。A回滚事务【rollback】，等B再查看账户的钱时，发现钱并没有多。

不可重复读：**一个事务读取到另外一个事务已经提交的数据，也就是说一个事务可以看到其他事务所做的修改**

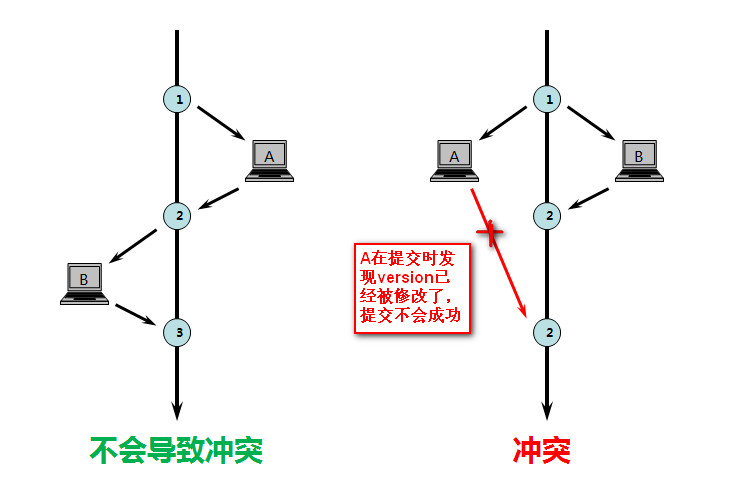
注：**A查询数据库得到数据，B去修改数据库的数据，导致A多次查询数据库的结果都不一样【危害：A每次查询的结果都是受B的影响的，那么A查询出来的信息就没有意思了】**

虚读(幻读)：**是指在一个事务内读取到了别的事务插入的数据，导致前后读取不一致。**

注：**和不可重复读类似，但虚读(幻读)会读到其他事务的插入的数据，导致前后读取不一致**

* 悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作
  + **在查询完数据的时候就把事务锁起来，直到提交事务**
  + 实现方式：使用数据库中的锁机制
* 乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。乐观锁不能解决脏读的问题
  + **在修改数据的时候把事务锁起来，通过version的方式来进行锁定**
  + 实现方式：使用version版本或者时间戳

1.使用**数据版本**（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。用下面的一张图来说明：



1. 乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型**使用时间戳（timestamp）**, 和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。

　在实际生产环境里边,如果并发量不大且不允许脏读，可以使用悲观锁解决并发问题；但如果系统的并发非常大的话,悲观锁定会带来非常大的性能问题,所以我们就要选择乐观锁定的方法.

如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 **"ABA"问题。**

* 超键：**在关系中能唯一标识元组的属性集称为关系模式的超键**。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。**超键包含候选键和主键**。
* **候选键(候选码)：是最小超键，即没有冗余元素的超键**。
* **主键(主码)：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合**。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。
* **外键：在一个表中存在的另一个表的主键称此表的外键**。

## SQL 约束有哪几种？

* NOT NULL: 用于控制字段的内容一定不能为空（NULL）。
* UNIQUE: 控件字段内容不能重复，一个表允许有多个 Unique 约束。
* PRIMARY KEY: 也是用于控件字段内容不能重复，但它在一个表只允许出现一个。
* FOREIGN KEY: 用于预防破坏表之间连接的动作，也能防止非法数据插入外键列，因为它必须是它指向的那个表中的值之一。
* CHECK: 用于控制字段的值范围。

MySQL常用的存储引擎有以下：

* **Innodb引擎**，Innodb引擎提供了对数据库ACID事务的支持。并且还提供了行级锁和外键的约束。它的设计的目标就是处理大数据容量的数据库系统。该引擎是不支持全文搜索的。同时，启动也比较的慢，它是不会保存表的行数的。当进行Select count(\*) from table指令的时候，需要进行扫描全表
* **MyIASM引擎**(原本Mysql的默认引擎),不提供事务的支持，也不支持行级锁和外键。。因此当执行Insert插入和Update更新语句时，即执行写操作的时候需要锁定这个表。所以会导致效率会降低。不过和Innodb不同的是，MyIASM引擎是保存了表的行数，于是当进行Select count(\*) from table语句时，可以直接的读取已经保存的值而不需要进行扫描全表。所以，如果表的读操作远远多于写操作时，并且不需要事务的支持的。可以将MyIASM作为数据库引擎的首先
* **MEMORY引擎**：所有的数据都在内存中，数据的处理速度快，但是安全性不高。

MyIASM引擎，B+树的数据结构中存储的内容实际上是实际数据的地址值。也就是说它的索引和实际数据是分开的，只不过使用索引指向了实际数据。这种索引的模式被称为非聚集索引。

Innodb引擎的索引的数据结构也是B+树，只不过数据结构中存储的都是实际的数据，这种索引有被称为聚集索引。

**Char是一种固定长度的类型，varchar是一种可变长度的类型**

# 数据库左连接、右连接、内连接、全连接

1.INNER JOIN （内连接）   
内连接是一种一一映射关系，就是两张表都有的才能显示出来

2.LEFT JOIN （左连接）

左连接是左边表的所有数据都有显示出来，右边的表数据只显示共同有的那部分，没有对应的部分只能补空显示，所谓的左边表其实就是指放在left join的左边的表

3.RIGHT JOIN（右连接）

右连接正好是和左连接相反的，这里的右边也是相对right join来说的，在这个右边的表就是右表

4.OUTER JOIN（外连接、全连接）

查询出左表和右表所有数据，但是去除两表的重复数据

5.LEFT JOIN EXCLUDING INNER JOIN（左连接不包含内连接）

这个查询是只查询左边表有的数据，共同有的也不查出来

7.OUTER JOIN EXCLUDING INNER JOIN（外连接不包含内连接）

意思就是查询左右表各自拥有的那部分数据

索引的一个主要目的就是加快检索表中数据的方法，亦即能协助信息搜索者尽快的找到符合限制条件的记录ID的辅助数据结构。

 事务处理可以确保除非事务性单元内的所有操作都成功完成，否则不会永久更新面向数据的资源。通过将一组相关操作组合为一个要么全部成功要么全部失败的单元，可以简化错误恢复并使应用程序更加可靠。一个逻辑工作单元要成为事务，必须满足所谓的ACID（原子性、一致性、隔离性和持久性）属性

同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。

1）原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，[删删删]因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

2）一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

3）隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

多个事务并发访问时，事务之间是隔离的，一个事务不应该影响其它事务运行效果。这指的是在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。

4）持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

添加索引原则

在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

定义为text、image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

## 请你说一说数据库的三大范式

### 参考回答：

第一范式：当关系模式R的所有属性都不能再分解为更基本的数据单位时，称R是满足第一范式，即属性不可分

第二范式：如果关系模式R满足第一范式，并且R得所有非主属性都完全依赖于R的每一个候选关键属性，称R满足第二范式

第三范式：设R是一个满足第一范式条件的关系模式，X是R的任意属性集，如果X非传递依赖于R的任意一个候选关键字，称R满足第三范式，即非主属性不传递依赖于键码

## 请你说一说mysql的四种隔离状态

### 参考回答：

Mysql主要包含四种隔离状态：

|  |  |  |  |
| --- | --- | --- | --- |
| 事务隔离级别 | 脏读 | 不可重复读 | 幻读 |
| 读未提交（read-uncommitted） | 是 | 是 | 是 |
| 不可重复读（read-committed） | 否 | 是 | 是 |
| 可重复读（repeatable-read） | 否 | 否 | 是 |
| 串行化（serializable） | 否 | 否 | 否 |

## 请你介绍一下mysql的MVCC机制

### 参考回答：

MVCC是一种多版本并发控制机制，是MySQL的InnoDB存储引擎实现隔离级别的一种具体方式，用于实现提交读和可重复读这两种隔离级别。MVCC是通过保存数据在某个时间点的快照来实现该机制，其在每行记录后面保存两个隐藏的列，分别保存这个行的创建版本号和删除版本号，然后Innodb的MVCC使用到的快照存储在Undo日志中，该日志通过回滚指针把一个数据行所有快照连接起来。

## 请你说一下MySQL引擎和区别

### 参考回答：

1、MySQL引擎

MySQL中的数据用各种不同的技术存储在文件（或者内存）中。这些技术中的每一种技术都使用不同的存储机制、索引技巧、锁定水平并且最终提供广泛的不同的功能和能力。通过选择不同的技术，你能够获得额外的速度或者功能，从而改善你的应用的整体功能。

数据库引擎是用于存储、处理和保护数据的核心服务。利用数据库引擎可控制访问权限并快速处理事务，从而满足企业内大多数需要处理大量数据的应用程序的要求。使用数据库引擎创建用于联机事务处理或联机分析处理数据的关系数据库。这包括创建用于存储数据的表和用于查看、管理和保护数据安全的数据库对象（如索引、视图和存储过程）。

MySQL存储引擎主要有： MyIsam、InnoDB、Memory、Blackhole、CSV、Performance\_Schema、Archive、Federated、Mrg\_Myisam。

但是最常用的是InnoDB和Mylsam。

2、InnoDB

InnoDB是一个事务型的存储引擎，有行级锁定和外键约束。

Innodb引擎提供了对数据库ACID事务的支持，并且实现了SQL标准的四种隔离级别，关于数据库事务与其隔离级别的内容请见数据库事务与其隔离级别这类型的文章。该引擎还提供了行级锁和外键约束，它的设计目标是处理大容量数据库系统，它本身其实就是基于MySQL后台的完整数据库系统，MySQL运行时Innodb会在内存中建立缓冲池，用于缓冲数据和索引。但是该引擎不支持FULLTEXT类型的索引，而且它没有保存表的行数，当SELECT COUNT(\*) FROM TABLE时需要扫描全表。当需要使用数据库事务时，该引擎当然是首选。由于锁的粒度更小，写操作不会锁定全表，所以在并发较高时，使用Innodb引擎会提升效率。但是使用行级锁也不是绝对的，如果在执行一个SQL语句时MySQL不能确定要扫描的范围，InnoDB表同样会锁全表。

适用场景：

经常更新的表，适合处理多重并发的更新请求。

支持事务。

可以从灾难中恢复（通过bin-log日志等）。

外键约束。只有他支持外键。

支持自动增加列属性auto\_increment。

索引结构：

InnoDB也是B+Treee索引结构。Innodb的索引文件本身就是数据文件，即B+Tree的数据域存储的就是实际的数据，这种索引就是聚集索引。这个索引的key就是数据表的主键，因此InnoDB表数据文件本身就是主索引。

InnoDB的辅助索引数据域存储的也是相应记录主键的值而不是地址，所以当以辅助索引查找时，会先根据辅助索引找到主键，再根据主键索引找到实际的数据。所以Innodb不建议使用过长的主键，否则会使辅助索引变得过大。建议使用自增的字段作为主键，这样B+Tree的每一个结点都会被顺序的填满，而不会频繁的分裂调整，会有效的提升插入数据的效率。

3、Mylsam

MyIASM是MySQL默认的引擎，但是它没有提供对数据库事务的支持，也不支持行级锁和外键，因此当INSERT或UPDATE数据时即写操作需要锁定整个表，效率便会低一些。MyIsam 存储引擎独立于操作系统，也就是可以在windows上使用，也可以比较简单的将数据转移到linux操作系统上去。

适用场景：

不支持事务的设计，但是并不代表着有事务操作的项目不能用MyIsam存储引擎，可以在service层进行根据自己的业务需求进行相应的控制。

不支持外键的表设计。

查询速度很快，如果数据库insert和update的操作比较多的话比较适用。

整天对表进行加锁的场景。

MyISAM极度强调快速读取操作。

MyIASM中存储了表的行数，于是SELECT COUNT(\*) FROM TABLE时只需要直接读取已经保存好的值而不需要进行全表扫描。如果表的读操作远远多于写操作且不需要数据库事务的支持，那么MyIASM也是很好的选择。

缺点：就是不能在表损坏后主动恢复数据。

索引结构：

MyISAM索引结构：MyISAM索引用的B+ tree来储存数据，MyISAM索引的指针指向的是键值的地址，地址存储的是数据。B+Tree的数据域存储的内容为实际数据的地址，也就是说它的索引和实际的数据是分开的，只不过是用索引指向了实际的数据，这种索引就是所谓的非聚集索引。

3、InnoDB和Mylsam的区别：

1）事务：MyISAM类型不支持事务处理等高级处理，而InnoDB类型支持，提供事务支持已经外部键等高级数据库功能。

2）性能：MyISAM类型的表强调的是性能，其执行数度比InnoDB类型更快。

3）行数保存：InnoDB 中不保存表的具体行数，也就是说，执行select count() fromtable时，InnoDB要扫描一遍整个表来计算有多少行，但是MyISAM只要简单的读出保存好的行数即可。注意的是，当count()语句包含where条件时，两种表的操作是一样的。

4）索引存储：对于AUTO\_INCREMENT类型的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中，可以和其他字段一起建立联合索引。MyISAM支持全文索引（FULLTEXT）、压缩索引，InnoDB不支持。

MyISAM的索引和数据是分开的，并且索引是有压缩的，内存使用率就对应提高了不少。能加载更多索引，而Innodb是索引和数据是紧密捆绑的，没有使用压缩从而会造成Innodb比MyISAM体积庞大不小。

InnoDB存储引擎被完全与MySQL服务器整合，InnoDB存储引擎为在主内存中缓存数据和索引而维持它自己的缓冲池。InnoDB存储它的表＆索引在一个表空间中，表空间可以包含数个文件（或原始磁盘分区）。这与MyISAM表不同，比如在MyISAM表中每个表被存在分离的文件中。InnoDB 表可以是任何尺寸，即使在文件尺寸被限制为2GB的操作系统上。

5）服务器数据备份：InnoDB必须导出SQL来备份，LOAD TABLE FROM MASTER操作对InnoDB是不起作用的，解决方法是首先把InnoDB表改成MyISAM表，导入数据后再改成InnoDB表，但是对于使用的额外的InnoDB特性(例如外键)的表不适用。

MyISAM应对错误编码导致的数据恢复速度快。MyISAM的数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。

InnoDB是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十G的时候就相对痛苦了。

6）锁的支持：MyISAM只支持表锁。InnoDB支持表锁、行锁 行锁大幅度提高了多用户并发操作的新能。但是InnoDB的行锁，只是在WHERE的主键是有效的，非主键的WHERE都会锁全表的。

## ● 请你说一下mysql引擎以及其区别

### 参考回答：

在Mysql数据库中，常用的引擎为Innodb和MyIASM,其中Innodb是一个事务型的存储引擎，有行级锁定和外键约束，提供了对数据库ACID事物的支持，实现了SQL标准的四种隔离级别，即读未提交，不可重复读，可重复读以及串行,其涉及目标就是处理大数据容量的数据库系统。而MyIASM引擎是Mysql默认的引擎，不提供数据库事务的支持，也不支持行级锁和外键，因此当写操作时需要锁定整个表，效率较低。不过其保存了表的行数，当金星select count(\*)form table时，可直接读取已经保存的值，不需要进行全表扫描。因此当表的读操作远多于写操作，并且不需要事务支持时，可以优先选择MyIASM

Redis相关

## 请你回答一下mongodb和redis的区别

### 参考回答：

内存管理机制上：Redis 数据全部存在内存，定期写入磁盘，当内存不够时，可以选择指定的 LRU 算法删除数据。MongoDB 数据存在内存，由 linux系统 mmap 实现，当内存不够时，只将热点数据放入内存，其他数据存在磁盘。

支持的数据结构上：Redis 支持的数据结构丰富，包括hash、set、list等。

MongoDB 数据结构比较单一，但是支持丰富的数据表达，索引，最类似关系型数据库，支持的查询语言非常丰富

相关面试题

<https://www.jianshu.com/p/36a646cef11a>

<https://www.nowcoder.com/tutorial/93/5268fb536d204ef3ad917d9d9ed92448>

Redis 与其他 key - value 缓存产品有以下三个特点：

* Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
* Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。
* Redis支持数据的备份，即master-slave模式的数据备份。

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

Redis hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。

每个 hash 可以存储 232 -1 键值对（40多亿）。

redis 127.0.0.1:6379> HMSET myhash field1 "Hello" field2 "World"

"OK"

redis 127.0.0.1:6379> HGET myhash field1

"Hello"

redis 127.0.0.1:6379> HGET myhash field2

"World"

List:

redis 127.0.0.1:6379> DEL runoob

redis 127.0.0.1:6379> lpush runoob redis

(integer) 1

redis 127.0.0.1:6379> lpush runoob mongodb

(integer) 2

redis 127.0.0.1:6379> lpush runoob rabitmq

(integer) 3

redis 127.0.0.1:6379> lrange runoob 0 10

1) "rabitmq"

2) "mongodb"

3) "redis"

Set:

Redis的Set是string类型的无序集合。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

redis 127.0.0.1:6379> DEL runoob

redis 127.0.0.1:6379> sadd runoob redis

(integer) 1

redis 127.0.0.1:6379> sadd runoob mongodb

(integer) 1

redis 127.0.0.1:6379> sadd runoob rabitmq

(integer) 1

redis 127.0.0.1:6379> sadd runoob rabitmq

(integer) 0

redis 127.0.0.1:6379> smembers runoob

1) "redis"

2) "rabitmq"

3) "mongodb"

Zset ;

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

zset的成员是唯一的,但分数(score)却可以重复。

redis 127.0.0.1:6379> DEL runoob

redis 127.0.0.1:6379> zadd runoob 0 redis

(integer) 1

redis 127.0.0.1:6379> zadd runoob 0 mongodb

(integer) 1

redis 127.0.0.1:6379> zadd runoob 0 rabitmq

(integer) 1

redis 127.0.0.1:6379> zadd runoob 0 rabitmq

(integer) 0

redis 127.0.0.1:6379> > ZRANGEBYSCORE runoob 0 1000

1) "mongodb"

2) "rabitmq"

3) "redis"

启动 redis 客户端，打开终端并输入命令 **redis-cli**。该命令会连接本地的 redis 服务。

$redis-cli

redis 127.0.0.1:6379>

redis 127.0.0.1:6379> PING

PONG

redis 127.0.0.1:6379> SET runoobkey redis

OK

redis 127.0.0.1:6379> DEL runoobkey

(integer) 1

在以上实例中 **DEL** 是一个命令， **runoobkey** 是一个键。 如果键被删除成功，命令执行后输出 **(integer) 1**，否则将输出 **(integer) 0**

redis 127.0.0.1:6379> SET runoobkey redis

OK

redis 127.0.0.1:6379> GET runoobkey

"redis"

Redis HyperLogLog 是用来做基数统计的算法，HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定 的、并且是很小的。

Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

Redis 客户端可以订阅任意数量的频道。

redis 127.0.0.1:6379> SUBSCRIBE redisChat

Reading messages... (press Ctrl-C to quit)

1) "subscribe"

2) "redisChat"

3) (integer) 1

现在，我们先重新开启个 redis 客户端，然后在同一个频道 redisChat 发布两次消息，订阅者就能接收到消息。

redis 127.0.0.1:6379> PUBLISH redisChat "Redis is a great caching technique"

(integer) 1

redis 127.0.0.1:6379> PUBLISH redisChat "Learn redis by runoob.com"

(integer) 1

# 订阅者的客户端会显示如下消息

1) "message"

2) "redisChat"

3) "Redis is a great caching technique"

1) "message"

2) "redisChat"

3) "Learn redis by runoob.com"

**Redis 事务**

Redis 事务可以一次执行多个命令， 并且带有以下三个重要的保证：

* 批量操作在发送 EXEC 命令前被放入队列缓存。
* 收到 EXEC 命令后进入事务执行，事务中任意命令执行失败，其余的命令依然被执行。
* 在事务执行过程，其他客户端提交的命令请求不会插入到事务执行命令序列中。

一个事务从开始到执行会经历以下三个阶段：

* 开始事务。
* 命令入队。
* 执行事务。
* 单个 Redis 命令的执行是原子性的，但 Redis 没有在事务上增加任何维持原子性的机制，所以 Redis 事务的执行并不是原子性的。
* 事务可以理解为一个打包的批量执行脚本，但批量指令并非原子化的操作，中间某条指令的失败不会导致前面已做指令的回滚，也不会造成后续的指令不做。

EXEC命令触发事务

Redis 连接命令主要是用于连接 redis 服务。

### 实例

以下实例演示了客户端如何通过密码验证连接到 redis 服务，并检测服务是否在运行：

redis 127.0.0.1:6379> AUTH "password"

OK

redis 127.0.0.1:6379> PING

PONG

Redis **SAVE** 命令用于创建当前数据库的备份。该命令将在 redis 安装目录中创建dump.rdb文件。

我们可以通过以下命令查看是否设置了密码验证：

127.0.0.1:6379> CONFIG get requirepass

1) "requirepass"

2) ""

默认情况下 requirepass 参数是空的，这就意味着你无需通过密码验证就可以连接到 redis 服务。

你可以通过以下命令来修改该参数：

127.0.0.1:6379> CONFIG set requirepass "runoob"

OK

127.0.0.1:6379> CONFIG get requirepass

1) "requirepass"

2) "runoob"

设置密码后，客户端连接 redis 服务就需要密码验证，否则无法执行命令

# Redis 客户端连接

Redis 通过监听一个 TCP 端口或者 Unix socket 的方式来接收来自客户端的连接，当一个连接建立后，Redis 内部会进行以下一些操作：

* 首先，客户端 socket 会被设置为非阻塞模式，因为 Redis 在网络事件处理上采用的是非阻塞多路复用模型。
* 然后为这个 socket 设置 TCP\_NODELAY 属性，禁用 Nagle 算法
* 然后创建一个可读的文件事件用于监听这个客户端 socket 的数据发送

#### 3.为什么redis是单线程的都那么快？

1.数据存于内存

2.用了多路复用I/O

3.单线程

redis能否将数据持久化，如何实现？

能，将内存中的数据异步写入硬盘中，两种方式：RDB（默认）和AOF

RDB持久化原理：通过bgsave命令触发，然后父进程执行fork操作创建子进程，子进程创建RDB文件，根据父进程内存生成临时快照文件，完成后对原有文件进行原子替换（定时一次性将所有数据进行快照生成一份副本存储在硬盘中）

优点：是一个紧凑压缩的二进制文件，Redis加载RDB恢复数据远远快于AOF的方式。

缺点：由于每次生成RDB开销较大，非实时持久化，

AOF持久化原理：开启后，Redis每执行一个修改数据的命令，都会把这个命令添加到AOF文件中。

优点：实时持久化。

缺点：所以AOF文件体积逐渐变大，需要定期执行重写操作来降低文件体积，加载慢

主从复制： master-slave

mysql主从复制用途

* 实时灾备，用于故障切换
* 读写分离，提供查询服务
* 备份，避免影响业务

主从复制原理:

从库生成两个线程，一个I/O线程，一个SQL线程；

i/o线程去请求主库 的binlog，并将得到的binlog日志写到relay log（中继日志） 文件中；

主库会生成一个 log dump 线程，用来给从库 i/o线程传binlog；

SQL 线程，会读取relay log文件中的日志，并解析成具体操作，来实现主从的操作一致，而最终数据一致；

mysql主从复制存在的问题：

* 主库宕机后，数据可能丢失
* 从库只有一个sql Thread，主库写压力大，复制很可能延时

解决方法：

* 半同步复制---解决数据丢失的问题
* 并行复制----解决从库复制延迟的问题

**半同步复制**

mysql semi-sync（半同步复制）

半同步复制：

* 5.5集成到mysql，以插件的形式存在，需要单独安装
* 确保事务提交后binlog至少传输到一个从库
* 不保证从库应用完这个事务的binlog
* 性能有一定的降低，响应时间会更长
* 网络异常或从库宕机，**卡主主库，直到超时或从库恢复**
* 半同步复制：
  + 原理:
    - 事务在主库写完binlog后需要从库返回一个已接受，才放回给客户端；

#### Redis集群的主从复制模型是怎样的？

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型,每个节点都会有N-1个复制品.

#### 怎么测试Redis的连通性？

ping

#### Redis中的管道有什么用？

一次请求/响应服务器能实现处理新的请求即使旧的请求还未被响应。这样就可以将多个命令发送到服务器，而不用等待回复，最后在一个步骤中读取该答复。

#### Redis key的过期时间和永久有效分别怎么设置？

EXPIRE和PERSIST命令。

#### Redis如何做内存优化？

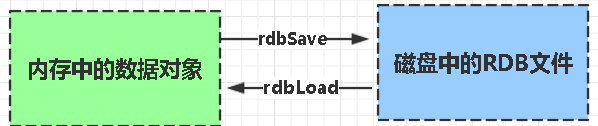
尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。

比如你的web系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的key,而是应该把这个用户的所有信息存储到一张散列表里面。

**RDB：**

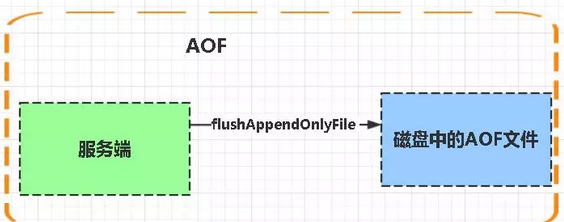
rdb是Redis DataBase缩写

功能核心函数rdbSave(生成RDB文件)和rdbLoad（从文件加载内存）两个函数



**AOF:**

Aof是Append-only file缩写



aof写入保存：

WRITE：根据条件，将 aof\_buf 中的缓存写入到 AOF 文件

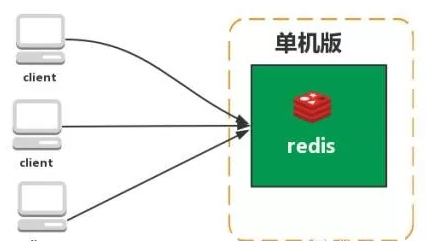
SAVE：根据条件，调用 fsync 或 fdatasync 函数，将 AOF 文件保存到磁盘中。

RESP 是redis客户端和服务端之前使用的一种通讯协议；

RESP 的特点：实现简单、快速解析、可读性好

**Redis 有哪些架构模式？讲讲各自的特点**

**单机版**

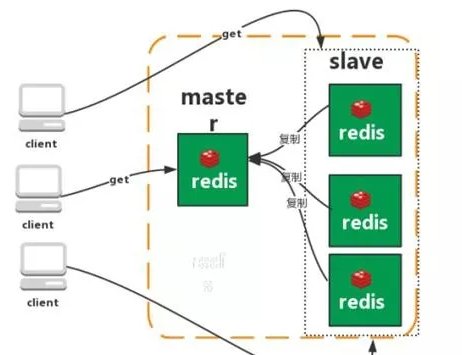


特点：简单

问题：

1、内存容量有限 2、处理能力有限 3、无法高可用。

**主从复制**

****

Redis 的复制（replication）功能允许用户根据一个 Redis 服务器来创建任意多个该服务器的复制品，其中被复制的服务器为主服务器（master），而通过复制创建出来的服务器复制品则为从服务器（slave）。 只要主从服务器之间的网络连接正常，主从服务器两者会具有相同的数据，主服务器就会一直将发生在自己身上的数据更新同步 给从服务器，从而一直保证主从服务器的数据相同。

特点：

1、master/slave 角色

2、master/slave 数据相同

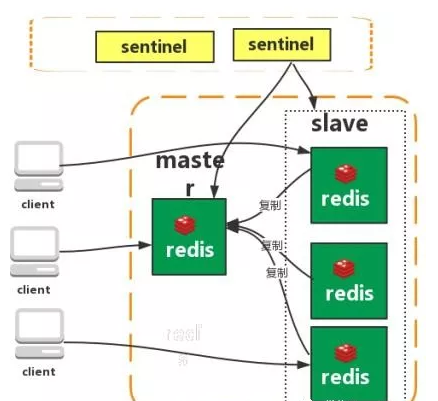
3、降低 master 读压力在转交从库

问题：

无法保证高可用

没有解决 master 写的压力

**哨兵**

****

Redis sentinel 是一个分布式系统中监控 redis 主从服务器，并在主服务器下线时自动进行故障转移。其中三个特性：

监控（Monitoring）：    Sentinel  会不断地检查你的主服务器和从服务器是否运作正常。

提醒（Notification）： 当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。

自动故障迁移（Automatic failover）： 当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作。

特点：

1、保证高可用

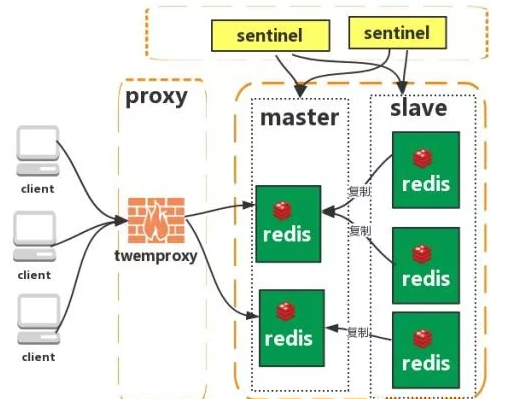
2、监控各个节点

3、自动故障迁移

缺点：主从模式，切换需要时间丢数据

没有解决 master 写的压力

**集群（proxy 型）：**

****

Twemproxy 是一个 Twitter 开源的一个 redis 和 memcache 快速/轻量级代理服务器； Twemproxy 是一个快速的单线程代理程序，支持 Memcached ASCII 协议和 redis 协议。

特点：1、多种 hash 算法：MD5、CRC16、CRC32、CRC32a、hsieh、murmur、Jenkins

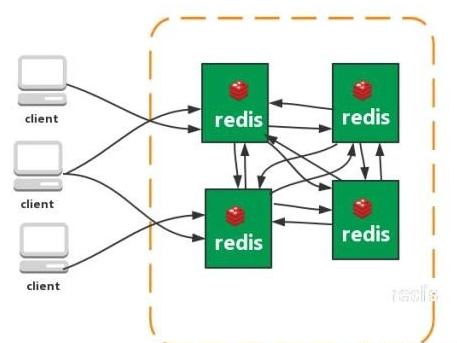
2、支持失败节点自动删除

3、后端 Sharding 分片逻辑对业务透明，业务方的读写方式和操作单个 Redis 一致

缺点：增加了新的 proxy，需要维护其高可用。

failover 逻辑需要自己实现，其本身不能支持故障的自动转移可扩展性差，进行扩缩容都需要手动干预

**集群（直连型）：**

****

从redis 3.0之后版本支持redis-cluster集群，Redis-Cluster采用无中心结构，每个节点保存数据和整个集群状态,每个节点都和其他所有节点连接。

特点：

1、无中心架构（不存在哪个节点影响性能瓶颈），少了 proxy 层。

2、数据按照 slot 存储分布在多个节点，节点间数据共享，可动态调整数据分布。

3、可扩展性，可线性扩展到 1000 个节点，节点可动态添加或删除。

4、高可用性，部分节点不可用时，集群仍可用。通过增加 Slave 做备份数据副本

5、实现故障自动 failover，节点之间通过 gossip 协议交换状态信息，用投票机制完成 Slave到 Master 的角色提升。

缺点：

1、资源隔离性较差，容易出现相互影响的情况。

2、数据通过异步复制,不保证数据的强一致性

缓存穿透

一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。一些恶意的请求会故意查询不存在的key,请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

如何避免？

1：对查询结果为空的情况也进行缓存，缓存时间设置短一点，或者该key对应的数据insert了之后清理缓存。

2：对一定不存在的key进行过滤。可以把所有的可能存在的key放到一个大的Bitmap中，查询时通过该bitmap过滤。

缓存雪崩

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，会给后端系统带来很大压力。导致系统崩溃。

如何避免？

1：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

2：做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期

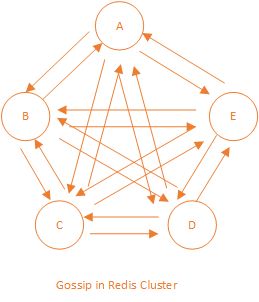
3：不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

一致性哈希算法：

<https://www.cnblogs.com/lpfuture/p/5796398.html>

哈希槽：

从redis 3.0之后版本支持redis-cluster集群，Redis-Cluster采用无中心结构，每个节点保存数据和整个集群状态,每个节点都和其他所有节点连接。



其结构特点：

     1、所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽。

     2、节点的fail是通过集群中超过半数的节点检测失效时才生效。

     3、客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可。

     4、redis-cluster把所有的物理节点映射到[0-16383]slot上（不一定是平均分配）,cluster 负责维护node<->slot<->value。

     5、Redis集群预分好16384个桶，当需要在 Redis 集群中放置一个 key-value 时，根据 CRC16(key) mod 16384的值，决定将一个key放到哪个桶中。