数据库的那些事

数据库是一个什么东西？数据库其实就是一个拿来持久化的一个东西。我们知道，计算机里的存储设备有很多，比如说像缓存，内存，HDD（机械硬盘），SSD（固态硬盘），这些东西都算作是储存设备。他们分为两大类，一种叫做易失性储存，另一种叫做非易失性储存。缓存和内存两者都属于易失性储存，而硬盘就是非易失性储存。其实这也很容易理解，我们从来没说过把文件保存到内存里吧？都是说保存在“盘”上嘛，大概说的就是这个意思了。对于服务器这种访问量大，而且必须要保存用户数据的地方，就必须要使用非易失性储存来进行存储，不然，一断电，全都木有了。

那么说到这里，易失性储存和非易失性储存还有什么区别呢？（除了丢不丢失之外）当然是有的，普遍来讲（这话不能说绝了，现在已经食大便了（时代变了））易失性储存比非易失性储存快得多，举个栗子来说，双通道的DDR4内存条，读写速度轻轻松松达到40GB/s，但是目前最最最牛逼的硬盘，读写速度只不过5-6GB/s，主流的机械硬盘还停留在100MB/s的速度呢。另外一个显著特点就是，内存要比硬盘贵多了，现在内存的价格大概要30-40块1GB，而顶级的固态硬盘，也只不过3块钱1GB而已，机械硬盘更是低到只要3毛钱1GB。因此，内存其实是非常珍贵的资源，我们根本不可能把这么大的数据塞进去内存里，这会造成极大的浪费。

有人肯定要问，我直接写入文件不好吗？再退一步讲我用Excel不好吗？干什么花这么大精神搞数据库？不妨让大家回想一下当年学C艹的时候文件流有多复杂吧，比如说对于字符串到底要用什么编码（锟斤拷，烫烫烫？），然后又要搞什么buffer reader，字节数组等等东西。说句难听点的，我写个程序还要处理这么多屁烂事情，不是很浪费精力吗？然后我再说，请你把文件中第1057292行中的数据修改为……咳咳咳，后果会怎么样我就不多说了。再回到Excel的问题，微软官方给出的说明中明确告诉你Excel最多只能放255列，104万行数据，生产环境中怎么可能只有这么点数据？

说了这么多……才终于引出数据库出场了。至于数据库有啥历史，大家还是去其它地方查吧……我尽量多讲点干货。数据库主要有两大阵营，一种就是传统的关系型数据库，另一种是现在非常流行的非关系型数据库。传统的数据库就相当多了，比如说MySQL，SQLite，SQL Server，DB2，Oracle等等。典型的非关系型数据库有mongoDB，Redis，Memcached（严格意义上说这个是缓存，还不能叫数据库），Tair等等。其实可以直说，非关系型数据库要比关系型数据库强大太多了，简直强到令人发指。但是，有些时候，传统的关系型数据库又会更胜一筹。

当然，为了应付考试，我准备从关系型数据库讲起，而且我不准备一上来就丢一堆概念给你们，不然你们全都点左上角了。我准备用我做的一个项目的实例来带你们看看数据库是怎么样运行的。

这个项目是用于统计高中宿舍各项扣分的一个实例，我们首先需要建立一张表来存储实时生成的扣分记录。为了保存这些内容，需要一些column来保存一些必要的特征信息，有点像Excel表格的一列，这个称之为“字段”。比如说在这张扣分记录表中，我们肯定要记录：谁，什么时候，干了什么事情，扣多少分，一共四个字段。这些字段有着不同的数据类型，比如说整型，浮点，字符串，时间戳等等。其实，数据类型只要归结于3种类型就好，因为它们在底层都是使用着同样的数据类型。（不是所有的数据库都有这些类型的，叫法也不一定相同）

第一种，numeric：

名称	说明
INT	也就是常说的int32
TINYINT	小整数。在有些数据库中，会把这个东西当成布尔值来用
BIGINT	大整数，也就是我们说的高精度整数
DECIMAL	高精度小数，一般用于精算和银行结算中
FLOAT	浮点数
BOOLEAN/BIT	布尔
TIMESTAMP	时间戳，本质就是一个长整数，表示从1970年1月1日的格林尼治时间至今的毫秒数
DATETIME	就是一个时间戳
DATE	其实就是那一天开始时刻（0时0分0秒0毫秒）的时间戳
COMPLEX	复数，比较少见，在一些科学计算中会有这个

第二种，字符串：

名称	说明
VARCHAR	定长字符串，比如说限定这个字段只能存255个字符或之类的，超出的部分就木有了
LONGTEXT	不限长字符串，就是为了解决VARCHAR的长度问题的，但是这个字段会导致性能不佳
JSON/BSON	JSON是一种形式，可以用来表示对象，通用泛型等等，B的意思就是Binary
UUID	这个多扯两句，这个是一种专门标识设备通用识别码，一般不会重复，以前经常被拿来做主键

第三种，二进制类型：

名称	说明
BLOB	以前，人们还喜欢把文件存进去数据库中，这个就是对应的类型，但是现在这种方式基本上已经被弃用了

回来正题，我刚刚说四个字段就可以记录我所有的信息了，但是，这真的够了吗？假设，我有一个人同时犯下了两项违纪，那我要怎么进行区分呢？而且，日后我要进行搜索的时候，我如何精准地找到我要的那一条记录呢？这个时候就需要给每一张表派发一个唯一的识别字段，叫做主键。有了主键，我就可以准确地定位到每一条数据，而且不会重复。但是，那什么东西来做主键比较好呢？既要不能重复，又不能太大，否则无端端占用我很多空间。早期人们用到的一个东西叫做UUID，它的本质就是一个定长字符串，而且由于它基本不会重复，所以它几乎在一段时间内成为了万金油，时至今日，像金蝶之类的办公软件数据库，依然还在使用UUID。但是后来，人们想到了另一个办法，就是用一个递增的序列作为主键。虽然这个主键毫无含义，但是却是最简单的一种实现方式。

用一个递增序列做主键没有意义啊，干嘛要拿这个这个东西做主键？我们不妨想象一下，数据库在进行查找的时候，需要进行大量的数据比对，你认为是比较一个字符串更快呢，还是一个整型更快呢？我在项目中填充了一百多万个数据做测试，当使用字符串做主键时，查询一次耗时大概0.6秒，但是一换到整型，查询一次只要0.3秒而已。那既然这样子做性能有很大的提升，那为啥早期的程序还要用UUID做主键？这个就涉及到一个东西叫做事务与锁，我在后面会继续讲为啥。

那我们现在已经获得了一张含有五个字段的扣分记录表（主键，谁，什么时候，干了什么，扣多少分），那我们要怎么用数据库呢来进行增删改查呢？我会分成两个部分来讲，一个是传统的SQL脚本，另一个是面向对象的数据库操作。

使用SQL语句进行增删改查并不困难，记就是了，比如说：

```
select `time` from `table` where name = "czf"
```

（其中time那里的不是单引号，是键盘左上角数字1隔壁的那个符号。）

同样的，要进行增加的时候，要用到insert语句，修改时使用update语句，删除时使用delete语句。这些基操我就不赘述了，网上一查一大把。但是SQL语句最难的地方，在于复杂查询和分页查询。例如，我想要“查询所有的同学的扣分值的总和并且帮我从大到小进行排序”。再这样的情况下，SQL语句就会变得极其复杂而且很容易由于拼写错误导致不必要的bug。

同时，SQL语句也引出了很多严重的安全问题。比如说我在进行SQL语句的拼接，比如说我拼接这样一个字符串：

```
"select" + userInput + "where" + variable + "=" + value;
```

那么，如果用户输入时给你恶意注入了一些代码例如union关键字，information_schema之类的东西，数据库可能会错误地执行这些代码而不小心将一些不该查的东西查出来了，比如说用户密码。这样子的网络攻击叫做SQL注入，这样子的注入攻击发生在很多老的系统里面，甚至华工的某些老系统也不例外。

除此之外，SQL语言还有很多不统一的问题。像Oracle数据库的SQL语句就和MySQL有些不同，尤其是主键自增长的问题，两者的运行机制完全不同。既然不统一，那肯定会给编程带来麻烦，引起一些不必要的错误。我印象中有一次，就是因为一颗语法糖的问题，卡了我很久没想明白bug在哪里。

那么我们有没有更加先进的工具来和数据库进行交互？那当然是有的。我们不妨这样想一想，如果我有一个类，它有私有成员：id，名字，时间，扣分理由，分值，然后我们把对应的数据类型准备好。像这样：

```
public class Record
{
    private int id;
    private String name;
    private Timestamp time;
    private String reason;
    private BigDecimal value;
}
```

仔细一想，那我这张表里的每一行数据，不就是这样的类的一个对象吗？这还没完，我们如果再把这样的类稍微封装一下，添加上findBy()，save()，delete()方法，那岂不是可以用调用这些方法而不需要去拼写又长又臭的SQL语句了？我用表格稍微解释一下这些方法是什么意思

方法	说明
findById(int id)	这个就是使用id（主键）进行查询的意思，如果有查询结果，就把Record对象返回，否则为null
save(record)	保存record对象的意思，这个save往往会自动判断到底是insert还是update操作，非常简便
delete(record)	顾名思义，就是把这一行数据删除的意思
CountByName(String name)	统计某个值出现了多少次，一般来讲在操作查询之前要先做一次这个，否则很容易引起空指针错误
findByName(String name)	不同于findById(int id)，这一次返回的记录可能有多条，所以，会自动帮你转换成List然后供你使用

有这些东西之后就相当方便了吧？Java里面就给出了这样的工具叫做Jpa。我前面说的那个Record类，叫做实体类(Entity)，那个对增删改查方法的封装，叫做DAO(data access object)。有了这样的工具，那就真的是美滋滋了，和数据库打交道再也不是一个麻烦的问题了，而且查询的结果还来到了对象的这个层次，这将会极大地方便我们面向对象编程呀！（强烈建议有兴趣的小伙伴试一下，就会知道这个东西有多香了）。不过，虽然这个很香，但是还不能解决我刚刚所说的复杂查询的问题。而且，如果查询出来的结果集太大，也会给内存造成很大的压力（毕竟这回操作的全都是对象啊）。

所以，拼写SQL有时候还确实是一个逃不掉的问题，但是，我上面所说的SQL语句不统一的问题，要怎么解决呢？这回还是Java，它给出了一个新的出路叫做JPQL(Java Persistence Query Language)这样的语言统一了我们在编写SQL语句时的规范，使得我们不再需要对不同的数据库区别对待。JPQL毕竟师出Java同门，所以它带有很明显的面向对象的色彩，我们又继续用Record类来举例子。那么我在查询某个人的扣分记录的时候，仅仅只需要写成这样：

```
select * from Record record where record.name = "czf"
```

诶它居然还弄了一个Record类的对象record，是不是具有很浓厚的面向对象色彩？用这个对象的一些数据成员来作为字段名，这样就一定程度上防范了拼写错误的问题了。（你要是select还能写错我也救不了你了）

那，对于SQL注入，JPQL有没有应对方法呢？当然是有的，请看操作：

```
@Query(value = "select * from Record record where record.name = :userInput")
public List<Record> findByName(@Param("userInput")String input);
```

以冒号为标志，JPQL会认为后面的这个东西是一个参数名，然后你只需要用@Param注解标识出来这个参数的值是多少即可。Java底层帮你做了防SQL注入的优化，只要我们写JPQL语句的时候遵循这个规矩，就可以轻松防住SQL注入。有了Jpa和JPQL的帮助，我们连接数据库和操作数据库变得不再困难，这也极大地降低了我们编程的难度。

讲到这肯定又有同学有疑问，都有Jpa工具了，为啥还要用JPQL构建复杂查询啊？（“查询所有的同学的扣分值的总和并且帮我从大到小进行排序”）我把所有扣分记录查出来，然后for循环加一加不也可以得到结果吗？嗯，我觉得挺有道理的，这就是为啥一个跑1.6秒，一个跑224秒。（咱都是学过数据结构的人了……我就提示一下，HashMap）

随着业务量越来越大，学校的扣分记录已经多达百万条了（学生：？？？）此时，数据库的日志告诉我们经常发生“慢查询”，甚至有一次，为了找一个数据，数据库还不惜一切代价动用了全表扫描呢！结果有一天，学校老师找到我说，小陈呀，你这个系统不行呀，咋这么卡呢？（以上故事纯属虚构，我早就在测试里解决了这样的问题）这个时候，MySQL给出了一本秘籍，上面写着：欲练神功必先自宫（划掉）索引！索引！此时，我把学生名字这个字段做了一个索引之后，使用速度暴涨十倍。

究竟索引是何方神圣？

我们在学习数据结构的时候，接触过索引这个概念，其实可以这样理解，就是让数据库快速查找的一个目录而已。索引优化得好，性能瞬间强十几倍毫不夸张。常见的用来索引的结构有树状索引，哈希表等等。那么，数据库用的索引是什么数据结构呢？我给出几个可能的答案大家先猜猜：BST，红黑树，哈希表，B/B+树。到底是哪个（些）呢？

BST肯定不是，老师说过，当数据单边增长的时候，BST会退化成一个链表，导致性能低下。

红黑树确实是目前的最快的索引树，打OA的同学肯定深知这玩意是啥（Map的底层实现就是这玩意）。但是，为啥数据库的索引并不是用这种结构呢？因为红黑树每个节点只有一个元素，如果数据量一大，那这棵树的高度就会很高，导致比较的次数过多。

第三个。没错，哈希表真的是可以拿来做索引的，但是，哈希表用的场合并不会太多，因为哈希表有一个天生的缺陷，范围查询。因为哈希函数出来的结果相比于输入的结果，就不一定还是一个连续的范围了，所以……出现范围查询时，哈希表就出问题了。

说到这大家肯定已经知道我想说啥了，MySQL底层的索引真的还就是用B+树做的。因为B+树一个节点可以存放多个元素，所以使得树的高度大大减小，查询自然就快起来了。

你说啥？不记得数据结构了？去这个网站看看先：

```
https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
```

诶等等，你弄B树不是还得一个节点内横向比较，而且不是说了红黑树是最快的嘛，你咋又说B+树效率高了？你tm是不是又在忽悠我？

这就要牵扯到一个叫做磁盘IO的东西了。上面我提到过，磁盘的速度是远远比不上内存的速度的，所以CPU在工作时，其实有很大一部分时间都浪费在了等待数据加载的过程中。然而对于红黑树，每走到一个节点，就需要读取一次磁盘，然后把读出来的值和查询的值比对，再走向下一个节点。然而在B树中，每走到一个节点，就把这个节点内所有的数据一次性加载到内存里，然后进行横向比较时，就是在内存中调取数据来进行了，速度自然就快起来了。综上所述，红黑树的速度最快这个并没有错，但是前提条件是，大家都在内存里进行，一旦涉及到多次的磁盘IO，红黑树的效率就被拉低了。

诶嘿，既然在内存里快的话，那我不如把一堆的数据都加载到内存里，这样我运行就快啦。是的，你说的很对，等你全部加载到内存的时候，我早都查完了……不要忽略磁盘读取文件的时间啊，读一个文件和三四个文件耗时差不多，但是，量变引起质变的啊。

其实这里还有一个概念叫做聚簇索引和非聚簇索引。不过……貌似我们不需要知道得太详细，我贴一个网址，大家看就好了：

```
https://www.jianshu.com/p/fa8192853184
```

说到这里，关于关系型数据库的基本操作和概念就已经差不多说完了，下面，就将开始关系型数据库的进阶篇了。

首先先来补一个前面的坑，我说了MySQL的主键自增长那里是有点点古怪的，还有就是那个为啥早期程序用UUID的问题。这个要涉及到一个数据库引擎的问题了，也就是说，数据库是如何管理大量的读写的呢？首先我们需要让大家明白一个概念叫做原子操作。因为数据库很可能不是一个线程在进行访问的，比如说我的那个服务，最大的并发量就高达200个。假设不同线程不小心修改了同一行的数据，那就发生撞车了，对吧？所以，数据库为了让读写有序进行而不会发生这些错误，就建立起了锁与事务的概念。所谓的事务，就是一块代码块（里面有若干个SQL语句），在执行这个代码块的中间，不可以有其它的操作打断或者插队。那么这个也意味着，这些代码，要么全部执行，要么全部不执行。

我下面讲一些非常深奥的概念哈，不想看的同学请忽略这一部分：

事务的特点：ACID（原子性，统一性，隔离性，持久性）

原子性：刚刚已经提过了，就是说这一串代码不可以拆分，要么全部执行，要么全部不执行，要么全部失败。

统一性：描述现实世界的一个真实情况，例如银行系统中，转出账户少了多少钱，转入账户就要多多少钱。

隔离性：假设好几个老师同时给一个同学扣分，那么无论他们执行的顺序如何，最后这个同学被扣的总分是一样的。

持久性：一旦提交成功，就会持久保存下来

原子性的实现依靠着一个叫做Undo log的东西。它相当于一个数据备份，当事务开始的时候，数据库会把原来的数据备份到Undo log当中，当事务执行到中间发生意外的时候，数据库会执行回滚操作，而回滚所需要的数据，就来自于Undo log。通过这样的办法，它可以保证事务失败后，数据库恢复原样。

持久性的实现依靠着一个叫做Redo log，和Undo log相反，Redo log记录了新生成的数据，也就是拿来进行前滚的一个操作。在事务提交之前，数据库会把Redo log先存储下来（持久化），而数据还不一定已经持久化。如果此时宕机，因为Redo log已经持久化了，所以我们可以通过Redo log把还没持久化的数据恢复出来。等等？怎么数据还没持久化呢？因为每一条记录都非常非常小，如果频繁地写入小文件，磁盘根本就顶不住。你们可以试一下复制2G的电影和2G的1KB小文件到底谁快。所以数据库都会想办法堆够了一些数据之后再进行顺序写入，一般而言，这个buffer size是64KB，不过这个可能根据操作系统有所不同。

当然，上面说的两种Log恢复机制并不是万能的，万一Log都扑街了，整个数据库当然扑街了。

隔离性是拿来保证操作的执行的统一性的，比如说扣分的问题，我们必须保证并行事务执行的结果不会和串行执行的结果不同。有一些情况下，为了防止并发导致数据错误，我们需要人为加上一些隔离。没错，隔离和并发就是冲突的，隔离等级太高，每次只能干一件事，浪费了系统资源，必然会导致性能下降。所以这也就是为什么我们需要在合适的范围之内，对隔离要求有所放宽。以MySQL为例，隔离等级从低到高是这样排列的：READ UNCOMMITED（就是不限制，容易出事）；READ COMMITED（读已提交）；REPETABLE READ（可重复读）；SERIALIZABLE（串行）。这几种隔离等级导致的后果如下表所示：

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITED	GG	GG	GG
READ COMMITED		翻车	翻车
REPEATABLE READ			还是会扑街
SERIALIZABLE			

我解释一下所有的概念：

READ UNCOMMITED：这个事务还没提交时，修改的东西就可以被访问到

READ COMMITED：只有提交之后，才可以被读取到

REPEATABLE READ：保证这个事务自始自终接触到的数据是一样的，不管其它的数据怎么变，这个事务中的不会变

SERIALIZABLE：会请求读锁和写锁，根据是否阻塞来进行操作

脏读：我在线程1中开启一个事务，它需要执行：小明扣1分；哎呀扣错了冤枉小明了，消分。然后我在线程2中开启一个事务：查询小明的扣分，扣分大于等于1分就要批评他。结果，线程1刚执行完小明扣1分时，线程2执行了，结果发现小明扣分大于等于1分了，于是小明就GG了，其实他没有扣分……。

不可重复读：在线程1中开启一个事务，它需要执行：小明扣1分；哎呀扣错了冤枉小明了，消分。然后在线程2中开启一个事务：查询小明的扣分总和，然后，分别找出来所有记录，呈现出来给老师看。结果，线程1刚执行完小明扣1分时，线程2执行了查询小明的扣分总和，得到了1分，然后线程1此时把小明的扣分取消了，然后，线程2开始找小明所有记录，发现是没有的。所以就得到了一个奇怪的结果，扣1分，但是又没有扣分记录……

幻读：在线程1中开启一个事务，它需要执行：在小明的扣分后面（假设刚刚小明已经扣分了）再加一条扣分，小华扣1分。线程2也执行一个：在小明的扣分后面（假设刚刚小明已经扣分了）再加一条扣分，小李扣1分。然后让线程1先跑，线程2后跑。结果线程2报错了：主键重复了。诶我明明刚刚都没有查询到小明后面有扣分啊，怎么就扣不下去了啊？？这个就是我在编写我的后台的时候遇到的最严重的问题，我当时用2000个线程轰炸它的时候，就出现了这样的错误。因为MySQL默认的隔离级别只到repeatable read，所以就翻车了。当然，我们并不能盲目调成SERIALIZABLE来解决问题，像这样的情况下，由于发生了行锁（把某一行锁住了），进而导致了死锁，所以线程2一直被阻塞就动不了了。（具体如何优化后面慢慢聊）

好了，回来回来。其实，统一性可以说就是事务的终极追求，我们做各种各样的优化，为的就是统一性。但是，往往在事务的执行期间，容易出现一些东西破坏了事务的一致性。比如说：并发执行，不小心修改了同一行的数据，哦吼。再比如，电脑突然宕机了，事务恰好执行了一半。为了解决这些问题，人们提出了一些对策：并发控制，日志恢复。日志我后面慢慢说，我们先来扯一扯并发控制中的——锁。

我们先来分成两种大的类别，一种叫做表锁，一种叫做行锁（其实还有库锁但是这里不谈了）

表锁里又分为读锁和写锁（也叫共享锁和独占锁），当读锁启用时，其它的用户依然可以进行读取操作，但是不可以进行写入操作；当写锁启用时，其他用户不允许写入，也不允许读取。那么这里可以看出来，一旦涉及到读写，整张表都会被锁起来，所以，性能被严重限制，这肯定不是一个好的办法。

行锁其实也有读锁和写锁（也叫共享锁和排他锁），其实含义和表锁差不多但是行锁的实现方式就和表有所区别。以MySQL为例，行锁会锁住索引字段（可能是主键，或者是其它的东西）当我们通过索引进行增删改查的时候，数据库才会使用行锁，否则还是会使用表锁。

有锁的存在，保证我们不会有机会同时修改同一个数据，同时，行锁的出现，使得在一张表上同时增删改查出现可能，大大提高了数据库的效率。好的，那我们回来说说，为啥以前人们经常用UUID做主键，而现在就不再这么干了呢？因为数据库引擎的改进，加入了事务这个概念，才使得主键自增长这件事情得以实施。如果说没有事务这个功能的话，要实现自增长主键，就会消耗极大的性能（因为要加上表锁之后，我才可以查出来当前的id值，然后再加一，才能插入一条数据），如果编程时不注意加锁，万一两个线程同时修改，那就会引起冲突。因为UUID不会重复，就不用担心冲突问题，所以在一些老系统中，为了减少并发时出现的问题，就索性拿UUID做主键了。

关于关系型数据库，还有一个知识点就是数据存储引擎，我们常说的有InnoDB和MYISAM，不过呢……目前而言，InnoDB已经是大势所趋了，想了解的同学上网查查就好。

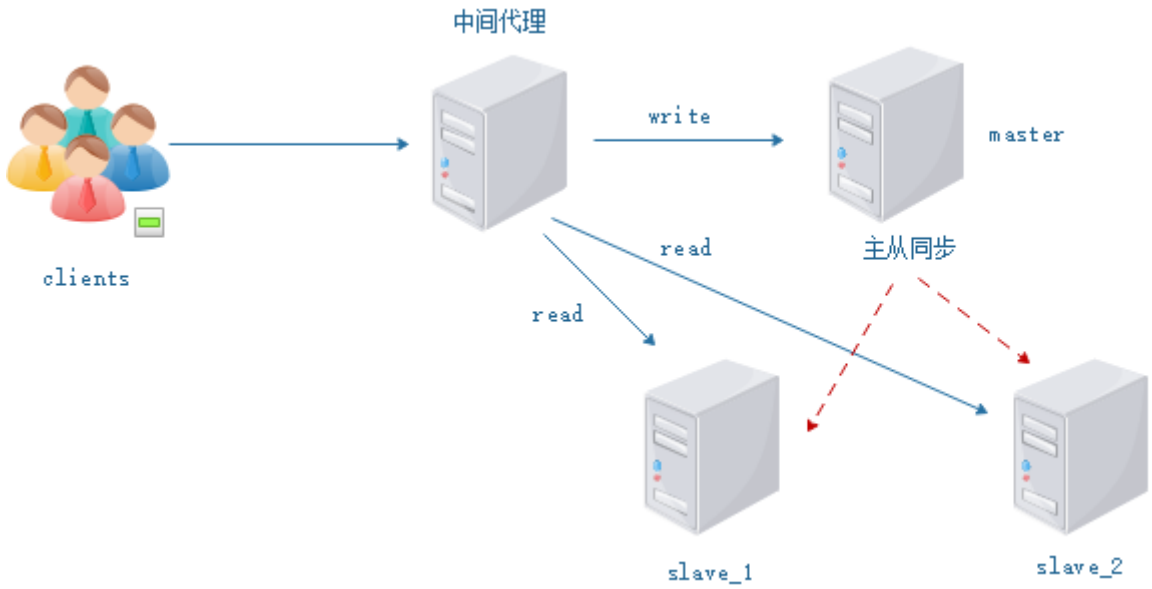
说完了关系型数据库，我们来谈谈一个后起之秀——非关系型数据库。其实我个人感觉这个翻译不是很准确哈，它的英文名称时Not Only SQL，准确来说叫做不完全是SQL，当然，它也可以当成关系型数据库来用，但是，非关系型数据库有着非常非常大的优势，可以说，基本上秒杀了很多传统的SQL数据库。但是，客观来说，技术没有优劣，关系型数据库和非关系型数据库各有各的优势，只是应用场景不同而已。

一讲到非关系型数据库……突然间有点心血来潮，因为……感觉好像自己正在经历一场巨大的变革。我们先从发展历程讲起，说一说到底数据库是如何进化的。我们还是用MySQL来举例子吧，毕竟这个确实是免费数据库中强者之一了。

第一个时期：早期的网站和服务器访问量和数据量并不是很大，一台服务器配上一个MySQL数据库足以应付这样的访问量，习惯上我们称这段时间为单机MySQL的黄金时代。毕竟，顶破天也就是几十人的访问量，区区几万几十万条数据而已，MySQL应付毫无压力。

第二个时期：这个时候网站的访问量大起来了，会出现某个时候的一个高峰，高峰的出现很可能会整瘫整台服务器，于是乎人们想到了一个办法，在数据库前面挡一个缓存，不要让那么大的查询量直接冲进去数据库。于是此时出现一个东西叫做Memcached，它挡在数据库的前面，可以起到缓冲的作用，也可以帮忙进行一些查询，比如说前面查过这个数据，在缓存里有，那么此时Memcatched就会直接返回查询结果而不需要真正地进去数据库进行查询。

第三个时期：很快缓存也架不住了，越来越大的数据量始终考验着数据库的吞吐量。因此人们没有办法了，只能将读取与写入进行拆分。由于真正多而且耗时的其实是查询，写入往往相对来说是更少也更快的。所以出现了一种结构叫做“主从复制，读写分离”，大概长这样：



主服务器负责写，然后进行复制，给两个从服务器，然后查询就通过两台从服务器进行。这样确实一定程度上解决了问题，而且也让数据有了多份备份，也就是所谓的容灾性会更好。只要我们将代理服务器做好，性能就可以得到很好的利用。比如说毛子发明的Nginx，就是一个很好的负载均衡工具。这样结构的出现，确实让传统的数据库又得到了新的生机。

第四个时期：由于数据还在继续膨胀，原来的单个实例中已经存储了百万甚至千万级别的数据，即便算法再优秀，一样还是很慢。怎么办呢？这个时候就要将MySQL的实例进行拓展，形成所谓的MySQL集群。假设我有9000万条数据，前3000万条去第一个库，3000-6000万条去第二个库，6000万之后的去第三个库。这样拆分之后，速度又很快了。

第五个时期：似乎我们不会遇到瓶颈了吧？如果服务器不够用了，再加一个实例不就好了嘛？并不。随着现在数据越来越多样化，很多时候，限定好字段的数据库就会给我们带来很多的麻烦。就举淘宝为例，有的店铺广告费不够，那么它只可以展示五张图片，然后有的店铺给的钱多，他就可以展示视频，音频甚至AR等等。面对多变而且不统一的字段，现在再来维护一个表就很困难了，不然一天到晚就要加字段，而且这些字段还不是每个人都会带有，所以后台还得做各种各样的分类操作。这样子就非常麻烦了。可以这么说，并不是性能卡住了传统数据库的脖子，而是它的拓展性带来了问题。而且，传统的数据库我们建表，加了字段之后，我们还要去构建相关的映射，相关的实体类，相关的DAL之类的所有东西，这会让程序员非常害怕，尤其是在维护一个巨型工程的时候。我曾经就极其不愿意在差不多完工的时候去加一张表和一个同学吵了起来……可见这个拓展有多麻烦吧。

那我们所说的非关系型数据库是怎么运行的呢？它的所有数据就是一个KV键值对，K就是Key，一般来讲就是hashkey，V就是Value，对应的是一个JSON（有的数据库是BSON）。那么这个数据库也就是变成了一个巨大的哈希表。由于JSON的特性，它可以非常轻松地扩展，几乎不需要任何成本就可以给这个数据库加入或者删除一个字段。虽然以前我很看不起这样的储存方式，但是直到有一天我遇到了一个项目，它彻底改变了我的看法。

那是一个游戏，游戏里有非常非常多的装备，然后我需要把用户的金币数量，等级，经验……所有东西都存下来，结果我发现……那张表大到根本没办法维护，上百个字段，头都弄大了。但是，换成是非关系型数据库，管你那么多，塞进来就完事了。不仅如此，这个数据库的性能还极其可怕，体量就没比SQLite大多少，但是却可以提供每秒多达几万次的超高查询次数，性能轻松秒杀传统的数据库。而且当数据量到达上亿条时，它依然可以保持超高的性能（传统的数据库谁敢搞几亿条数据进去？）

下次，咱们来聊聊非关系型数据库中的一个典型——Redis