目录

核心篇

数据存储

1、MySQL 索引使用的注意事项

EXPLAIN 可以帮助开发人员分析 SQL 问题,explain 显示了 mysql 如何使用索引来处理 select 语句以及连接表,可以帮助选择更好的索引和写出更优化的查询语句。

使用方法,在 select 语句前加上 Explain 就可以了：

索引虽然好处很多，但过多的使用索引可能带来相反的问题，索引也是有缺点的：

虽然索引大大提高了查询速度,同时却会降低更新表的速度,如对表进行 INSERT,UPDATE 和 DELETE。因为更新表时，mysql 不仅要保存数据，还要保存一下索引文件

建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在要给大表上建了多种组合索引，索引文件会膨胀很宽

索引只是提高效率的一个方式，如果 mysql 有大数据量的表，就要花时间研究建立最优的索引，或优化查询语句。

使用索引时，有一些技巧：

1.索引不会包含有 NULL 的列

只要列中包含有 NULL 值，都将不会被包含在索引中，复合索引中只要有一列含有 NULL 值，那么这一列对于此符合索引就是无效的。

2.使用短索引

对串列进行索引，如果可以就应该指定一个前缀长度。例如，如果有一个 char（255）的列，如果在前 10 个或 20 个字符内，多数值是唯一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和 I/O 操作。

3.索引列排序

mysql 查询只使用一个索引，因此如果 where 子句中已经使用了索引的话，那么 order by 中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作，尽量不要包含多个列的排序，如果需要最好给这些列建复合索引。

4.like 语句操作

一般情况下不鼓励使用 like 操作，如果非使用不可，注意正确的使用方式。like ‘%aaa%’不会使用索引，而 like ‘aaa%’可以使用索引。

5.不要在列上进行运算

6.不使用 NOT IN 、<>、！=操作，但<,<=，=，>,>=,BETWEEN,IN 是可以用到索引的

7.索引要建立在经常进行 select 操作的字段上。

这是因为，如果这些列很少用到，那么有无索引并不能明显改变查询速度。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。

8.索引要建立在值比较唯一的字段上。

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

10.在 where 和 join 中出现的列需要建立索引。

11.where 的查询条件里有不等号(where column != …),mysql 将无法使用索引。

12.如果 where 字句的查询条件里使用了函数(如：where DAY(column)=…),mysql 将无法使用索引。

13.在 join 操作中(需要从多个数据表提取数据时)，mysql 只有在主键和外键的数据类型相同时才能使用索引，否则及时建立了索引也不会使用。

2、说说反模式设计

简单的来说，反模式是指在对经常面对的问题经常使用的低效，不良，或者有待优化的设计模式/方法。甚至，反模式也可以是一种错误的开发思想/理念。在这里我举一个最简单的例子：在面向对象设计/编程中，有一条很重要的原则， 单一责任原则(Single responsibility principle)。其中心思想就是对于一个模块，或者一个类来说，这个模块或者这个类应该只对系统/软件的一个功能负责，而且该责任应该被该类完全封装起来。当开发人员需要修改系统的某个功能，这个模块/类是最主要的修改地方。相对应的一个反模式就是上帝类(God Class)，通常来说，这个类里面控制了很多其他的类，同时也依赖其他很多类。整个类不光负责自己的主要单一功能，而且还负责了其他很多功能，包括一些辅助功能。很多维护老程序的开发人员们可能都遇过这种类，一个类里有几千行的代码，有很多功能，但是责任不明确单一。单元测试程序也变复杂无比。维护/修改这个类的时间要远远超出其他类的时间。很多时候，形成这种情况并不是开发人员故意的。很多情况下主要是由于随着系统的年限，需求的变化，项目的资源压力，项目组人员流动，系统结构的变化而导致某些原先小型的，符合单一原则类慢慢的变的臃肿起来。最后当这个类变成了维护的噩梦(特别是原先熟悉的开发人员离职后)，重构该类就变成了一个不容易的工程。

3、说说分库与分表设计

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。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，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。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。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。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）

4、分库与分表带来的分布式困境与应对之策

数据迁移与扩容问题

前面介绍到水平分表策略归纳总结为随机分表和连续分表两种情况。连续分表有可能存在数据热点的问题，有些表可能会被频繁地查询从而造成较大压力，热数据的表就成为了整个库的瓶颈，而有些表可能存的是历史数据，很少需要被查询到。连续分表的另外一个好处在于比较容易，不需要考虑迁移旧的数据，只需要添加分表就可以自动扩容。随机分表的数据相对比较均匀，不容易出现热点和并发访问的瓶颈。但是，分表扩展需要迁移旧的数据。

针对于水平分表的设计至关重要，需要评估中短期内业务的增长速度，对当前的数据量进行容量规划，综合成本因素，推算出大概需要多少分片。对于数据迁移的问题，一般做法是通过程序先读出数据，然后按照指定的分表策略再将数据写入到各个分表中。

表关联问题

在单库单表的情况下，联合查询是非常容易的。但是，随着分库与分表的演变，联合查询就遇到跨库关联和跨表关系问题。在设计之初就应该尽量避免联合查询，可以通过程序中进行拼装，或者通过反范式化设计进行规避。

分页与排序问题

一般情况下，列表分页时需要按照指定字段进行排序。在单库单表的情况下，分页和排序也是非常容易的。但是，随着分库与分表的演变，也会遇到跨库排序和跨表排序问题。为了最终结果的准确性，需要在不同的分表中将数据进行排序并返回，并将不同分表返回的结果集进行汇总和再次排序，最后再返回给用户。

分布式事务问题

随着分库与分表的演变，一定会遇到分布式事务问题，那么如何保证数据的一致性就成为一个必须面对的问题。目前，分布式事务并没有很好的解决方案，难以满足数据强一致性，一般情况下，使存储数据尽可能达到用户一致，保证系统经过一段较短的时间的自我恢复和修正，数据最终达到一致。

分布式全局唯一 ID

在单库单表的情况下，直接使用数据库自增特性来生成主键 ID，这样确实比较简单。在分库分表的环境中，数据分布在不同的分表上，不能再借助数据库自增长特性。需要使用全局唯一 ID，例如 UUID、GUID 等。关于如何选择合适的全局唯一 ID，我会在后面的章节中进行介绍。

5、说说 SQL 优化之道

一、一些常见的 SQL 实践

（1）负向条件查询不能使用索引

```
select from order where status!=0 and stauts!=1
```

not in/not exists 都不是好习惯

可以优化为 in 查询：

select from order where status in(2,3)

（2）前导模糊查询不能使用索引

select from order where desc like '%XX'

而非前导模糊查询则可以：

select from order where desc like 'XX%'

（3）数据区分度不大的字段不宜使用索引

select from user where sex=1

原因：性别只有男，女，每次过滤掉的数据很少，不宜使用索引。

经验上，能过滤 80%数据时就可以使用索引。对于订单状态，如果状态值很少，不宜使用索引，如果状态值很多，能够过滤大量数据，则应该建立索引。

（4）在属性上进行计算不能命中索引

select from order where YEAR(date) < = '2017'

即使 date 上建立了索引，也会全表扫描，可优化为值计算：

select from order where date < = CURDATE()

或者：

select from order where date < = '2017-01-01'

二、并非周知的 SQL 实践

（5）如果业务大部分是单条查询，使用 Hash 索引性能更好，例如用户中心

select from user where uid=?

select from user where login_name=?

原因：B-Tree 索引的时间复杂度是 O(log(n))；Hash 索引的时间复杂度是 O(1)

（6）允许为 null 的列，查询有潜在大坑

单列索引不存 null 值，复合索引不存全为 null 的值，如果列允许为 null，可能会得到“不符合预期”的结果集

select from user where name != 'shenjian'

如果 name 允许为 null，索引不存储 null 值，结果集中不会包含这些记录。

所以，请使用 not null 约束以及默认值。

（7）复合索引最左前缀，并不是值 SQL 语句的 where 顺序要和复合索引一致

用户中心建立了(login_name, passwd)的复合索引

select from user where login_name=? and passwd=?

select from user where passwd=? and login_name=?

都能够命中索引

select from user where login_name=?

也能命中索引，满足复合索引最左前缀

select from user where passwd=?

不能命中索引，不满足复合索引最左前缀

（8）使用 ENUM 而不是字符串

ENUM 保存的是 TINYINT，别在枚举中搞一些“中国”“北京”“技术部”这样的字符串，字符串空间又大，效率又低。

三、小众但有用的 SQL 实践

（9）如果明确知道只有一条结果返回，limit 1 能够提高效率

select from user where login_name=?

可以优化为：

select from user where login_name=? limit 1

原因：你知道只有一条结果，但数据库并不知道，明确告诉它，让它主动停止游标移动

（10）把计算放到业务层而不是数据库层，除了节省数据的 CPU，还有意想不到的查询缓存优化效果

select from order where date < = CURDATE()

这不是一个好的 SQL 实践，应该优化为：

$curDate = date('Y-m-d');

$res = mysqlquery(

'select from order where date < = $curDate');

原因：

释放了数据库的 CPU

多次调用，传入的 SQL 相同，才可以利用查询缓存

（11）强制类型转换会全表扫描

select from user where phone=13800001234

你以为会命中 phone 索引么？大错特错了，这个语句究竟要怎么改？

末了，再加一条，不要使用 select *（潜台词，文章的 SQL 都不合格 ==），只返回需要的列，能够大大的节省数据传输量，与数据库的内存使用量哟。

6、MySQL 遇到的死锁问题

产生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

下列方法有助于最大限度地降低死锁：

（1）按同一顺序访问对象。

（2）避免事务中的用户交互。

（3）保持事务简短并在一个批处理中。

（4）使用低隔离级别。

（5）使用绑定连接。

7、存储引擎的 InnoDB 与 MyISAM

◆.InnoDB 不支持 FULLTEXT 类型的索引。

◆2.InnoDB 中不保存表的具体行数，也就是说，执行 select count() from table 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。

◆3.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。

◆4.DELETE FROM table 时，InnoDB 不会重新建立表，而是一行一行的删除。

◆5.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。

另外，InnoDB 表的行锁也不是绝对的，假如在执行一个 SQL 语句时 MySQL 不能确定要扫描的范围，InnoDB 表同样会锁全表，例如 update table set num=1 where name like “%aaa%”

8、数据库索引的原理

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用 B 树及其变种 B+树。

9、为什么要用 B-tree

B-tree：B 树属于多叉树又名平衡多路查找树。

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。而 B-/+/*Tree，经过改进可以有效的利用系统对磁盘的块读取特性，在读取相同磁盘块的同时，尽可能多的加载索引数据，来提高索引命中效率，从而达到减少磁盘 IO 的读取次数。

（1）B-tree 利用了磁盘块的特性进行构建的树。每个磁盘块一个节点，每个节点包含了很关键字。把树的节点关键字增多后树的层级比原来的二叉树少了，减少数据查找的次数和复杂度。

B-tree 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（每页为 4K），这样每个节点只需要一次 I/O 就可以完全载入。

B-tree 的数据可以存在任何节点中。

（2）B+tree 是 B-tree 的变种，数据只能存储在叶子节点。

B+tree 是 B-tree 的变种，B+tree 数据只存储在叶子节点中。这样在 B 树的基础上每个节点存储的关键字数更多，树的层级更少所以查询数据更快，所有指关键字指针都存在叶子节点，所以每次查找的次数都相同所以查询速度更稳定。

（3）B*tree 每个磁盘块中又添加了对下一个磁盘块的引用。这样可以在当前磁盘块满时，不用扩容直接存储到下一个临近磁盘块中。当两个邻近的磁盘块都满时，这两个磁盘块各分出 1/3 的数据重新分配一个磁盘块，这样这三个磁盘块的数据都为 2/3。

在 B+树的基础上因其初始化的容量变大，使得节点空间使用率更高，而又存有兄弟节点的指针，可以向兄弟节点转移关键字的特性使得 B*树额分解次数变得更少。

10、聚集索引与非聚集索引的区别

聚集索引，就是主键索引。

除了聚集索引以外的索引都是非聚集索引，只是人们想细分一下非聚集索引，分成普通索引，唯一索引，全文索引。

1).聚集索引一个表只能有一个，而非聚集索引一个表可以存在多个

2).聚集索引存储记录是物理上连续存在，而非聚集索引是逻辑上的连续，物理存储并不连续

3).聚集索引:物理存储按照索引排序；聚集索引是一种索引组织形式，索引的键值逻辑顺序决定了表数据行的物理存储顺序

非聚集索引:物理存储不按照索引排序；非聚集索引则就是普通索引了，仅仅只是对数据列创建相应的索引，不影响整个表的物理存储顺序.

4) .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。

11、limit 20000 加载很慢怎么解决

limit10000,20 的意思扫描满足条件的 10020 行，扔掉前面的 10000 行，返回最后的 20 行，问题就在这里。

mysql 的性能低是因为数据库要去扫描 N+M 条记录，然后又要放弃之前 N 条记录，开销很大。

解决思略：

1、前端加缓存，或者其他方式，减少落到库的查询操作，例如某些系统中数据在搜索引擎中有备份的，可以用 es 等进行搜索

2、使用延迟关联，即先通用 limit 得到需要数据的索引字段，然后再通过原表和索引字段关联获得需要数据。

select a.* from a,(select id from table_1 where is_deleted='N' limit 100000,20) b where a.id = b.id

3、从业务上实现，不分页如此多，例如只能分页前 100 页，后面的不允许再查了

4、不使用 limit N,M,而是使用 limit N，即将 offset 转化为 where 条件。

12、选择合适的分布式主键方案

1.数据库自增长序列或字段：

2.UUID：

对于 InnoDB 这种聚集主键类型的引擎来说，数据会按照主键进行物理排序，这对 auto_increment int 是个好消息，因为后一次插入的主键位置总是在最后。但是对 uuid 来说，这却是个坏消息，因为 uuid 是杂乱无章的，每次插入的主键位置是不确定的，可能在开头，也可能在中间，在进行主键物理排序的时候，势必会造成大量的 IO 操作影响效率，因此不适合使用 UUID 做物理主键。比较适合的做法是把 uuid 作为逻辑主键，物理主键依然使用 自增 ID

3.使用 UUID to Int64 的方法：

4.Redis 生成 ID：

假如一个集群中有 5 台 Redis，可以初始化每台 Redis 的值分别是 1,2,3,4,5，然后步长都是 5。各个 Redis 生成的 ID 为 ：

A : 1,6,11,16,21...

B : 2,7,12,17,22...

C : 3,8,13,18,23...

D : 4,9,14,19,24...

E : 5,10,15,20,25...

优点 : 1> 不依赖于数据库，灵活方便，且性能优于数据库

2> 数字 ID 天然排序，对分页或者需要排序的结果很有帮助

缺点 ：1> 如果系统中没有 Redis，还需要引入新的组件，增加系统复杂度

2> 需要编码和配置的工作量比较大

5. Twitter 的 snowflake 算法：

snowflake 算法 的原理就是用 64 位整数来表示主键。

1 bit 符号位 ： 表示正负，方便使用负数标识不正确的 ID

41 bit 毫秒时间 ：2^41 / (365 * 24 * 3600 * 1000) ≈ 69 年

10 bit 机房 ID + 机器 ID： 最大值为 1023

12 bit 递增序列 ： 最大值为 4095

6. 利用 zookeeper 生成唯一 ID：

zookeeper 主要通过其 znode 数据版本来生成序列号，可以生成 32 位和 64 位的数据版本号，客户端可以使用这个版本号来作为唯一的序列号。

很少会使用 zookeeper 来生成唯一 ID。主要是由于需要依赖 zookeeper，并且是多步调用 API，如果在竞争较大的情况下，需要考虑使用分布式锁。因此，性能在高并发的分布式环境下，也不甚理想

7. MongoDB 的 ObjectId:

MongoDB 的 ObjectId 和 snowflake 算法类似，它设计成轻量型的，不同的机器都能用全局唯一的同种方法方便地生成它。MongoDB 从一开始就设计用来作为分布式数据库，处理多个节点是一个核心要求，使其在分片环境中要容易生成得多。

13、选择合适的数据存储方案

1）关系型数据库 MySQL

MySQL 是一个最流行的关系型数据库，在互联网产品中应用比较广泛。一般情况下，MySQL 数据库是选择的第一方案，基本上有 80%～90% 的场景都是基于 MySQL 数据库的。因为，需要关系型数据库进行管理，此外，业务存在许多事务性的操作，需要保证事务的强一致性。同时，可能还存在一些复杂的 SQL 的查询。值得注意的是，前期尽量减少表的联合查询，便于后期数据量增大的情况下，做数据库的分库分表。

2）内存数据库 Redis

随着数据量的增长，MySQL 已经满足不了大型互联网类应用的需求。因此，Redis 基于内存存储数据，可以极大的提高查询性能，对产品在架构上很好的补充。例如，为了提高服务端接口的访问速度，尽可能将读频率高的热点数据存放在 Redis 中。这个是非常典型的以空间换时间的策略，使用更多的内存换取 CPU 资源，通过增加系统的内存消耗，来加快程序的运行速度。

在某些场景下，可以充分的利用 Redis 的特性，大大提高效率。这些场景包括缓存，会话缓存，时效性，访问频率，计数器，社交列表，记录用户判定信息，交集、并集和差集，热门列表与排行榜，最新动态等。

使用 Redis 做缓存的时候，需要考虑数据不一致与脏读、缓存更新机制、缓存可用性、缓存服务降级、缓存穿透、缓存预热等缓存使用问题。

3）文档数据库 MongoDB

MongoDB 是对传统关系型数据库的补充，它非常适合高伸缩性的场景，它是可扩展性的表结构。基于这点，可以将预期范围内，表结构可能会不断扩展的 MySQL 表结构，通过 MongoDB 来存储，这就可以保证表结构的扩展性。

此外，日志系统数据量特别大，如果用 MongoDB 数据库存储这些数据，利用分片集群支持海量数据，同时使用聚集分析和 MapReduce 的能力，是个很好的选择。

MongoDB 还适合存储大尺寸的数据，GridFS 存储方案就是基于 MongoDB 的分布式文

件存储系统。

4）列族数据库 HBase

HBase 适合海量数据的存储与高性能实时查询，它是运行于 HDFS 文件系统之上，并且作为 MapReduce 分布式处理的目标数据库，以支撑离线分析型应用。在数据仓库、数据集市、商业智能等领域发挥了越来越多的作用，在数以千计的企业中支撑着大量的大数据分析场景的应用。

5）全文搜索引擎 ElasticSearch

在一般情况下，关系型数据库的模糊查询，都是通过 like 的方式进行查询。其中，like “value%” 可以使用索引，但是对于 like “%value%” 这样的方式，执行全表查询，这在数据量小的表，不存在性能问题，但是对于海量数据，全表扫描是非常可怕的事情。ElasticSearch 作为一个建立在全文搜索引擎 Apache Lucene 基础上的实时的分布式搜索和分析引擎，适用于处理实时搜索应用场景。此外，使用 ElasticSearch 全文搜索引擎，还可以支持多词条查询、匹配度与权重、自动联想、拼写纠错等高级功能。因此，可以使用 ElasticSearch 作为关系型数据库全文搜索的功能补充，将要进行全文搜索的数据缓存一份到 ElasticSearch 上，达到处理复杂的业务与提高查询速度的目的。

ElasticSearch 不仅仅适用于搜索场景，还非常适合日志处理与分析的场景。著名的 ELK 日志处理方案，由 ElasticSearch、Logstash 和 Kibana 三个组件组成，包括了日志收集、聚合、多维度查询、可视化显示等。

14、ObjectId 规则

[0,1,2,3] [4,5,6] [7,8] [9,10,11]

时间戳 |机器码 |PID |计数器

前四位是时间戳，可以提供秒级别的唯一性。

接下来三位是所在主机的唯一标识符，通常是机器主机名的散列值。

接下来两位是产生 ObjectId 的 PID，确保同一台机器上并发产生的 ObjectId 是唯一的。

前九位保证了同一秒钟不同机器的不同进程产生的 ObjectId 时唯一的。

最后三位是自增计数器，确保相同进程同一秒钟产生的 ObjectId 是唯一的。

15、聊聊 MongoDB 使用场景

高伸缩性的场景

MongoDB 非常适合高伸缩性的场景，它是可扩展性的表结构。基于这点，可以将预期范围内，表结构可能会不断扩展的 MySQL 表结构，通过 MongoDB 来存储，这就可以保证表结构的扩展性。

日志系统的场景

日志系统数据量特别大，如果用 MongoDB 数据库存储这些数据，利用分片集群支持海量数据，同时使用聚集分析和 MapReduce 的能力，是个很好的选择。

分布式文件存储

MongoDB 还适合存储大尺寸的数据，之前介绍的 GridFS 存储方案，就是基于 MongoDB 的分布式文件存储系统。

16、倒排索引

倒排索引（英语：Inverted index），也常被称为反向索引、置入档案或反向档案，是一

种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。它是文档检索系统中最常用的数据结构。

有两种不同的反向索引形式：

一条记录的水平反向索引（或者反向档案索引）包含每个引用单词的文档的列表。

一个单词的水平反向索引（或者完全反向索引）又包含每个单词在一个文档中的位置。

17、聊聊 ElasticSearch 使用场景

ElasticSearch 的功能：

分布式的搜索引擎和数据分析引擎

搜索：网站的站内搜索，IT 系统的检索

数据分析：电商网站，统计销售排名前 10 的商家

全文检索，结构化检索，数据分析

全文检索：我想搜索商品名称包含某个关键字的商品

结构化检索：我想搜索商品分类为日化用品的商品都有哪些

数据分析：我们分析每一个商品分类下有多少个商品

对海量数据进行近实时的处理

分布式：ES 自动可以将海量数据分散到多台服务器上去存储和检索

海联数据的处理：分布式以后，就可以采用大量的服务器去存储和检索数据，自然而然就可以实现海量数据的处理了

近实时：检索数据要花费 1 小时（这就不要近实时，离线批处理，batch-processing）；在秒级别对数据进行搜索和分析

ElasticSearch 的应用场景：

维基百科

The Guardian（国外新闻网站）

Stack Overflow（国外的程序异常讨论论坛）

GitHub（开源代码管理）

电商网站

日志数据分析

商品价格监控网站

BI 系统

站内搜索

ElasticSearch 的特点：

可以作为一个大型分布式集群（数百台服务器）技术，处理 PB 级数据，服务大公司；也可以运行在单机上，服务小公司

Elasticsearch 不是什么新技术，主要是将全文检索、数据分析以及分布式技术，合并在了一起

对用户而言，是开箱即用的，非常简单，作为中小型的应用，直接 3 分钟部署一下 ES

Elasticsearch 作为传统数据库的一个补充,比如全文检索，同义词处理，相关度排名，复杂数据分析，海量数据的近实时处理。

缓存使用

18、Redis 有哪些类型

Redis 目前支持 5 种数据类型，分别是：
String（字符串）
List（列表）
Hash（字典）
Set（集合）
Sorted Set（有序集合）

1. String（字符串）

String 是简单的 key-value 键值对，value 不仅可以是 String，也可以是数字。String 在 redis 内部存储默认就是一个字符串，被 redisObject 所引用，当遇到 incr,decr 等操作时会转成数值型进行计算，此时 redisObject 的 encoding 字段为 int。

String 在 redis 内部存储默认就是一个字符串，被 redisObject 所引用，当遇到 incr,decr 等操作时会转成数值型进行计算，此时 redisObject 的 encoding 字段为 int。

应用场景

String 是最常用的一种数据类型，普通的 key/value 存储都可以归为此类，这里就不所做解释了。

2. List（列表）

Redis 列表是简单的字符串列表，可以类比到 C++中的 std::list，简单的说就是一个链表或者说是一个队列。可以从头部或尾部向 Redis 列表添加元素。列表的最大长度为 2^32 - 1，也即每个列表支持超过 40 亿个元素。

Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销，Redis 内部的很多实现，包括发送缓冲队列等也都是用的这个数据结构。

应用场景

Redis list 的应用场景非常多，也是 Redis 最重要的数据结构之一，比如 twitter 的关注列表、粉丝列表等都可以用 Redis 的 list 结构来实现，再比如有的应用使用 Redis 的 list 类型实现一个简单的轻量级消息队列，生产者 push，消费者 pop/bpop。

3. Hash（字典，哈希表）

类似 C#中的 dict 类型或者 C++中的 hash_map 类型。

Redis Hash 对应 Value 内部实际就是一个 HashMap，实际这里会有 2 种不同实现，这个 Hash 的成员比较少时 Redis 为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正的 HashMap 结构，对应的 value redisObject 的 encoding 为 zipmap,当成员数量增大时会自动转成真正的 HashMap,此时 encoding 为 ht。

应用场景

假设有多个用户及对应的用户信息，可以用来存储以用户 ID 为 key，将用户信息序列化为比如 json 格式做为 value 进行保存。

4. Set（集合）

可以理解为一堆值不重复的列表，类似数学领域中的集合概念，且 Redis 也提供了针对集合的求交集、并集、差集等操作。

set 的内部实现是一个 value 永远为 null 的 HashMap，实际就是通过计算 hash 的方式来快速排重的，这也是 set 能提供判断一个成员是否在集合内的原因。

应用场景

Redis set 对外提供的功能与 list 类似是一个列表的功能，特殊之处在于 set 是可以自动排重的，当你需要存储一个列表数据，又不希望出现重复数据时，set 是一个很好的选择，并且 set 提供了判断某个成员是否在一个 set 集合内的重要接口，这个也是 list 所不能提供

的。

又或者在微博应用中，每个用户关注的人存在一个集合中，就很容易实现求两个人的共同好友功能。

5. Sorted Set（有序集合）

Redis 有序集合类似 Redis 集合，不同的是增加了一个功能，即集合是有序的。一个有序集合的每个成员带有分数，用于进行排序。

Redis 有序集合添加、删除和测试的时间复杂度均为 O(1)(固定时间，无论里面包含的元素集合的数量)。列表的最大长度为 2^32- 1 元素(4294967295，超过 40 亿每个元素的集合)。

Redis sorted set 的内部使用 HashMap 和跳跃表(SkipList)来保证数据的存储和有序，HashMap 里放的是成员到 score 的映射，而跳跃表里存放的是所有的成员，排序依据是 HashMap 里存的 score,使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

使用场景：

Redis sorted set 的使用场景与 set 类似，区别是 set 不是自动有序的，而 sorted set 可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。当你需要一个有序的并且不重复的集合列表，那么可以选择 sorted set 数据结构，比如 twitter 的 public timeline 可以以发表时间作为 score 来存储，这样获取时就是自动按时间排好序的。

又比如用户的积分排行榜需求就可以通过有序集合实现。还有上面介绍的使用 List 实现轻量级的消息队列，其实也可以通过 Sorted Set 实现有优先级或按权重的队列。

19、Redis 内部结构

Redis 内部使用一个 redisObject 对象来表示所有的 key 和 value。type ：代表一个 value 对象具体是何种数据类型。

encoding ：是不同数据类型在 redis 内部的存储方式，比如：type=string 代表 value 存储的是一个普通字符串，那么对应的 encoding 可以是 raw 或者是 int，如果是 int 则代表实际 redis 内部是按数值型类存储和表示这个字符串的，当然前提是这个字符串本身可以用数值表示，比如："123" "456"这样的字符串。

vm 字段：只有打开了 Redis 的虚拟内存功能，此字段才会真正的分配内存，该功能默认是关闭状态的。 Redis 使用 redisObject 来表示所有的 key/value 数据是比较浪费内存的，当然这些内存管理成本的付出主要也是为了给 Redis 不同数据类型提供一个统一的管理接口，实际作者也提供了多种方法帮助我们尽量节省内存使用。

20、聊聊 Redis 使用场景

使用场景说明：

1 计数器

数据统计的需求非常普遍，通过原子递增保持计数。例如，点赞数、收藏数、分享数等。

2 排行榜

排行榜按照得分进行排序，例如，展示最近、最热、点击率最高、活跃度最高等等条件的 top list。

3 用于存储时间戳

类似排行榜，使用 redis 的 zset 用于存储时间戳，时间会不断变化。例如，按照用户关注用户的最新动态列表。

4 记录用户判定信息

记录用户判定信息的需求也非常普遍，可以知道一个用户是否进行了某个操作。例如，用户是否点赞、用户是否收藏、用户是否分享等。

5 社交列表

社交属性相关的列表信息，例如，用户点赞列表、用户收藏列表、用户关注列表等。

6 缓存

缓存一些热点数据，例如，PC 版本文件更新内容、资讯标签和分类信息、生日祝福寿星列表。

7 队列

Redis 能作为一个很好的消息队列来使用，通过 list 的 lpop 及 lpush 接口进行队列的写入和消费，本身性能较好能解决大部分问题。但是，不提倡使用，更加建议使用 rabbitmq 等服务，作为消息中间件。

8 会话缓存

使用 Redis 进行会话缓存。例如，将 web session 存放在 Redis 中。

9 业务使用方式

String(字符串): 应用数, 资讯数等, (避免了 select count(*) from ...)

Hash（哈希表）: 用户粉丝列表, 用户点赞列表, 用户收藏列表, 用户关注列表等。

List（列表）：消息队列, push/sub 提醒。

SortedSet（有序集合）：热门列表, 最新动态列表, TopN, 自动排序。

21、redis 持久化机制与实现

众所周知，redis 是内存数据库，它把数据存储在内存中，这样在加快读取速度的同时也对数据安全性产生了新的问题，即当 redis 所在服务器发生宕机后，redis 数据库里的所有数据将会全部丢失。

为了解决这个问题，redis 提供了持久化功能——RDB 和 AOF。通俗的讲就是将内存中的数据写入硬盘中。

一、持久化之全量写入：RDB（快照）

```
[redis@6381]$ more /usr/local/redis/conf/redis.conf
save 900 1
save 300 10
save 60 10000
dbfilename "dump.rdb"                #持久化文件名称
dir "/data/dbs/redis/6381"       #持久化数据文件存放的路径
```

上面是 redis 配置文件里默认的 RDB 持久化设置，前三行都是对触发 RDB 的一个条件，例如第一行的意思是每 900 秒钟里 redis 数据库有一条数据被修改则触发 RDB，依次类推；只要有一条满足就会调用 BGSAVE 进行 RDB 持久化。第四行 dbfilename 指定了把内存里的数据库写入本地文件的名称，该文件是进行压缩后的二进制文件，通过该文件可以把数据库还原到生成该文件时数据库的状态。第五行 dir 指定了 RDB 文件存放的目录。

配置文件修改需要重启 redis 服务，我们还可以在命令行里进行配置，即时生效，服务器重启后需重新配置。

而 RDB 持久化也分两种：SAVE 和 BGSAVE

SAVE 是阻塞式的 RDB 持久化，当执行这个命令时 redis 的主进程把内存里的数据库状态写入到 RDB 文件（即上面的 dump.rdb）中，直到该文件创建完毕的这段时间内 redis 将不能处理任何命令请求。

BGSAVE 属于非阻塞式的持久化，它会创建一个子进程专门去把内存中的数据库状态写入 RDB 文件里，同时主进程还可以处理来自客户端的命令请求。但子进程基本是复制的父进程，这等于两个相同大小的 redis 进程在系统上运行，会造成内存使用率的大幅增加。

而 RDB 持久化也分两种：SAVE 和 BGSAVE

SAVE 是阻塞式的 RDB 持久化，当执行这个命令时 redis 的主进程把内存里的数据库状态写入到 RDB 文件（即上面的 dump.rdb）中，直到该文件创建完毕的这段时间内 redis 将不能处理任何命令请求。

BGSAVE 属于非阻塞式的持久化，它会创建一个子进程专门去把内存中的数据库状态写入 RDB 文件里，同时主进程还可以处理来自客户端的命令请求。但子进程基本是复制的父进程，这等于两个相同大小的 redis 进程在系统上运行，会造成内存使用率的大幅增加。

二、持久化之增量写入：AOF

与 RDB 的保存整个 redis 数据库状态不同，AOF 是通过保存对 redis 服务端的写命令（如 set、sadd、rpush）来记录数据库状态的，即保存你对 redis 数据库的写操作，以下就是 AOF 文件的内容。

```
[redis@iZ]$ more ~/redis/conf/redis.conf
dir "/data/dbs/redis/6381"              #AOF 文件存放目录
appendonly yes                          #开启 AOF 持久化，默认关闭
appendfilename "appendonly.aof"         #AOF 文件名称（默认）
appendfsync no                          #AOF 持久化策略
auto-aof-rewrite-percentage 100         #触发 AOF 文件重写的条件（默认）
auto-aof-rewrite-min-size 64mb          #触发 AOF 文件重写的条件（默认）
```

要弄明白上面几个配置就得从 AOF 的实现去理解，AOF 的持久化是通过命令追加、文件写入和文件同步三个步骤实现的。当 reids 开启 AOF 后，服务端每执行一次写操作（如 set、sadd、rpush）就会把该条命令追加到一个单独的 AOF 缓冲区的末尾，这就是命令追加；然后把 AOF 缓冲区的内容写入 AOF 文件里。看上去第二步就已经完成 AOF 持久化了那第三步是干什么的呢？这就需要从系统的文件写入机制说起：一般我们现在所使用的操作系统，为了提高文件的写入效率，都会有一个写入策略，即当你往硬盘写入数据时，操作系统不是实时的将数据写入硬盘，而是先把数据暂时的保存在一个内存缓冲区里，等到这个内存缓冲区的空间被填满或者是超过了设定的时限后才会真正的把缓冲区内的数据写入硬盘中。也就是说当 redis 进行到第二步文件写入的时候，从用户的角度看是已经把 AOF 缓冲区里的数据写入到 AOF 文件了，但对系统而言只不过是把 AOF 缓冲区的内容放到了另一个内存缓冲区里而已，之后 redis 还需要进行文件同步把该内存缓冲区里的数据真正写入硬盘上才算是完成了一次持久化。而何时进行文件同步则是根据配置的 appendfsync 来进行：

appendfsync 有三个选项：always、everysec 和 no：

1、选择 always 的时候服务器会在每执行一个事件就把 AOF 缓冲区的内容强制性的写入硬盘上的 AOF 文件里，可以看成你每执行一个 redis 写入命令就往 AOF 文件里记录这条命令，这保证了数据持久化的完整性，但效率是最慢的，却也是最安全的；

2、配置成 everysec 的话服务端每执行一次写操作（如 set、sadd、rpush）也会把该条命令追加到一个单独的 AOF 缓冲区的末尾，并将 AOF 缓冲区写入 AOF 文件，然后每隔一秒才会进行一次文件同步把内存缓冲区里的 AOF 缓存数据真正写入 AOF 文件里，这个模式兼顾了效率的同时也保证了数据的完整性，即使在服务器宕机也只会丢失一秒内对 redis 数据库做的修改；

3、将 appendfsync 配置成 no 则意味 redis 数据库里的数据就算丢失你也可以接受，它也会把每条写命令追加到 AOF 缓冲区的末尾，然后写入文件，但什么时候进行文件同步真正把数据写入 AOF 文件里则由系统自身决定，即当内存缓冲区的空间被填满或者是超过了设定的时限后系统自动同步。这种模式下效率是最快的，但对数据来说也是最不安全的，如果 redis 里的数据都是从后台数据库如 mysql 中取出来的，属于随时可以找回或者不重要的数据，那么可以考虑设置成这种模式。

相比 RDB 每次持久化都会内存翻倍，AOF 持久化除了在第一次启用时会新开一个子进程创建 AOF 文件会大幅度消耗内存外，之后的每次持久化对内存使用都很小。但 AOF 也有一个不可忽视的问题：AOF 文件过大。你对 redis 数据库的每一次写操作都会让 AOF 文件里增加一条数据，久而久之这个文件会形成一个庞然大物。还好的是 redis 提出了 AOF 重写的机制，即我们上面配置的 auto-aof-rewrite-percentage 和 auto-aof-rewrite-min-size。AOF 重写机制这里暂不细述，之后本人会另开博文对此解释，有兴趣的同学可以看看。我们只要知道 AOF 重写既是重新创建一个精简化的 AOF 文件，里面去掉了多余的冗余命令，并对原 AOF 文件进行覆盖。这保证了 AOF 文件大小处于让人可以接受的地步。而上面的 auto-aof-rewrite-percentage 和 auto-aof-rewrite-min-size 配置触发 AOF 重写的条件。

Redis 会记录上次重写后 AOF 文件的文件大小，而当前 AOF 文件大小跟上次重写后 AOF 文件大小的百分比超过 auto-aof-rewrite-percentage 设置的值，同时当前 AOF 文件大小也超过 auto-aof-rewrite-min-size 设置的最小值，则会触发 AOF 文件重写。以上面的配置为例，当现在的 AOF 文件大于 64mb 同时也大于上次重写 AOF 后的文件大小，则该文件就会被 AOF 重写。

最后需要注意的是，如果 redis 开启了 AOF 持久化功能，那么当 redis 服务重启时会优先使用 AOF 文件来还原数据库。

22、Redis 集群方案与实现

实现基础——分区:

分区是分割数据到多个 Redis 实例的处理过程，因此每个实例只保存 key 的一个子集。

通过利用多台计算机内存的和值，允许我们构造更大的数据库。

通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。

集群的几种实现方式:

1 客户端分片

2 基于代理的分片

3 路由查询

客户端分片:

由客户端决定 key 写入或者读取的节点。

包括 jedis 在内的一些客户端，实现了客户端分片机制。

特性:

优点:

简单，性能高。

缺点:

业务逻辑与数据存储逻辑耦合

可运维性差

多业务各自使用 redis，集群资源难以管理

不支持动态增删节点

基于代理的分片:

客户端发送请求到一个代理，代理解析客户端的数据，将请求转发至正确的节点，然后将结果回复给客户端。

开源方案

Twemproxy

Codis

特性:

透明接入 Proxy 的逻辑和存储的逻辑是隔离的。

业务程序不用关心后端 Redis 实例，切换成本低。

代理层多了一次转发，性能有所损耗

路由查询

将请求发送到任意节点，接收到请求的节点会将查询请求发送到正确的节点上执行。

开源方案:Redis-cluster

集群的挑战

涉及多个 key 的操作通常是不被支持的。涉及多个 key 的 redis 事务不能使用。

举例来说，当两个 set 映射到不同的 redis 实例上时，你就不能对这两个 set 执行交集操作。

不能保证集群内的数据均衡。

分区的粒度是 key，如果某个 key 的值是巨大的 set、list，无法进行拆分。

增加或删除容量也比较复杂。

redis 集群需要支持在运行时增加、删除节点的透明数据平衡的能力。

Redis 集群各种方案原理

1)Twemproxy

Proxy-based

twtter 开源，C 语言编写，单线程。

支持 Redis 或 Memcached 作为后端存储。

Twemproxy 高可用部署架构

Twemproxy 特性:

支持失败节点自动删除

与 redis 的长连接，连接复用，连接数可配置

自动分片到后端多个 redis 实例上支持 redis pipelining 操作

多种 hash 算法：能够使用不同的分片策略和散列函数

支持一致性 hash，但是使用 DHT 之后，从集群中摘除节点时，不会进行 rehash 操作

可以设置后端实例的权重

支持状态监控

支持 select 切换数据库

Twemproxy 不足:

性能低：代理层损耗 && 本身效率低下

Redis 功能支持不完善

不支持针对多个值的操作，比如取 sets 的子交并补等（MGET 和 DEL 除外）

不支持 Redis 的事务操作

出错提示还不够完善

集群功能不够完善

仅作为代理层使用

本身不提供动态扩容，透明数据迁移等功能

失去维护

最近一次提交在一年之前。Twitter 内部已经不再使用。

2)Redis Cluster:

Redis 官网推出，可线性扩展到 1000 个节点

无中心架构

一致性哈希思想

客户端直连 redis 服务，免去了 proxy 代理的损耗

23、redis 为什么是单线程的

1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是 O(1)；

2、数据结构简单，对数据操作也简单，Redis 中的数据结构是专门进行设计的；

3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

4、使用多路 I/O 复用模型，非阻塞 IO；

5、使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis 直接自己构建了 VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。

官方 FAQ 表示，因为 Redis 是基于内存的操作，CPU 不是 Redis 的瓶颈，Redis 的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且 CPU 不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）。

24、缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级

一、缓存雪崩

缓存雪崩我们可以简单的理解为：由于原有缓存失效，新缓存未到期间(例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库 CPU 和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

碰到这种情况，一般并发量不是特别多的时候，使用最多的解决方案是加锁排队。加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。假设在高并发下，缓存重建期间 key 是锁着的，这是过来 1000 个请求 999 个都在阻塞的。同样会导致用户等待超时，这是个治标不治本的方法！

给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存。

1、缓存标记：记录缓存数据是否过期，如果过期会触发通知另外的线程在后台去更新实际 key 的缓存；

2、缓存数据：它的过期时间比缓存标记的时间延长 1 倍，例：标记缓存时间 30 分钟，数据缓存设置为 60 分钟。 这样，当缓存标记 key 过期后，实际缓存还能把旧数据返回给调用端，直到另外的线程在后台更新完成后，才会返回新缓存。

关于缓存崩溃的解决方法，这里提出了三种方案：使用锁或队列、设置过期标志更新缓存、为 key 设置不同的缓存失效时间，还有一各被称为“二级缓存”的解决方法，有兴趣的读者可以自行研究。

二、缓存穿透

缓存穿透是指用户查询数据，在数据库没有，自然在缓存中也不会有。这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库再查询一遍，然后返回空（相当于进行了两次无用的查询）。这样请求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。

缓存穿透解决方案：

（1）采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。

（2）如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。通过这个直接设置的默认值存放到缓存，这样第二次到缓存中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴！

三、缓存预热

缓存预热就是系统上线后，提前将相关的缓存数据直接加载到缓存系统。避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

缓存预热解决方案：

（1）直接写个缓存刷新页面，上线时手工操作下；

（2）数据量不大，可以在项目启动的时候自动进行加载；

（3）定时刷新缓存。

四、缓存更新

除了缓存服务器自带的缓存失效策略之外（Redis 默认的有 6 中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

（1）定时去清理过期的缓存；

（2）当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的 key 是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

五、缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

（1）一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

（2）警告：有些服务在一段时间内成功率有波动（如在 95~100%之间），可以自动降级或人工降级，并发送告警；

（3）错误：比如可用率低于 90%，或者数据库连接池被打爆了，或者访问量突然猛增

到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

（4）严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

25、使用缓存的合理性问题

1 热点数据

对于冷数据而言，读取频率低，大部分数据可能还没有再次访问到就已经被挤出内存，不仅占用内存，而且价值不大。

对于热点数据，读取频率高。如果不做缓存，给数据库造成很大的压力，可能被击穿。

2 修改频率

数据更新前至少读取两次，缓存才有意义。这个是最基本的策略，如果缓存还没有起作用就失效了，那就没有太大价值了。（读取频率>修改频率）

如果这个读取接口对数据库的压力很大，但是又是热点数据，这个时候就需要考虑通过缓存手段，减少数据库的压力，比如我们的某助手产品的，点赞数，收藏数，分享数等是非常典型的热点数据，但是又不断变化，此时就需要将数据同步保存到 Redis 缓存，减少数据库压力

3 缓存更新机制

一般情况下，我们采取缓存双淘汰机制，在更新数据库的时候淘汰缓存。此外，设定超时时间，例如 30 分钟。极限场景下，即使有脏数据入 cache，这个脏数据也最多存在三十分钟。

在高并发的情况下，设计上最好避免查询 Mysql，所以在更新数据库的时候更新缓存。

4 缓存可用性

缓存是提高数据读取性能的，缓存数据丢失和缓存不可用不会影响应用程序的处理。因此，一般的操作手段是，如果 Redis 出现异常，我们手动捕获这个异常，记录日志，并且去数据库查询数据返回给用户。

5 服务降级

服务降级的目的，是为了防止 Redis 服务故障，导致数据库跟着一起发生雪崩问题。因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis 出现问题，不去数据库查询，而是直接返回默认值给用户。

6 对于可用性、服务降级实际情况

在大公司，redis 都是 codis 集群，一般整个 codis 是不会挂掉的。所以在程序代码上没去实现可用性、服务降级。（不知我说的对不对，大家参考就好）

7 缓存预热

在新启动的缓存系统中，如果没有任何数据，在重建缓存数据过程中，系统的性能和数据库复制都不太好，那么最好的缓存系统启动时就把热点数据加载好，例如对于缓存信息，在启动缓存加载数据库中全部数据进行预热。一般情况下，我们会开通一个同步数据的接口，进行缓存预热。

26、redis 缓存过期机智

mySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据

相关知识：redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略（回收策略）。redis 提供 6 种数据淘汰策略：

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的

数据淘汰

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

no-enviction（驱逐）：禁止驱逐数据。

27、Spring 中 RedisTemplate 操作

1. RedisTemplate 储存 set

```
/**
     * redis 储存 set
     */
    @RequestMapping("/set")
    public void redisSet(){
        Set<String> set=new HashSet<String>();
        set.add("111");
        set.add("222");
        set.add("333");
        redisTemplate.opsForSet().add("set",set);
        Set<String> resultSet =redisTemplate.opsForSet().members("set");
        System.out.println("resultSet:"+resultSet);
    }
```

2. RedisTemplated 储存 map

```
/**
     * redis 储存 map
     */
    @RequestMapping("/map")
    public void redisMap(){
        Map<String,String> map=new HashMap<String,String>();
        map.put("111","111");
        map.put("222","222");
        map.put("333","333");
        map.put("444","444");
        map.put("555","555");
        redisTemplate.opsForHash().putAll("map",map);
        Map<String,String> resultMap= redisTemplate.opsForHash().entries("map");
        List<String> reslutMapList=redisTemplate.opsForHash().values("map");
        Set<String>resultMapSet=redisTemplate.opsForHash().keys("map");
        String value=(String)redisTemplate.opsForHash().get("map","111");
        System.out.println("value:"+value);
        System.out.println("resultMapSet:"+resultMapSet);
        System.out.println("resultMap:"+resultMap);
```

```
        System.out.println("resulreslutMapListtMap:"+reslutMapList);
    }
```

3. RedisTemplate 储存 list

```
/**
     * redis 储存 list
     */
    @RequestMapping("/list")
    public void redisList(){
        List<String> list1=new ArrayList<String>();
        list1.add("111");
        list1.add("222");
        list1.add("333");
        List<String> list2=new ArrayList<String>();
        list2.add("444");
        list2.add("555");
        list2.add("666");
        redisTemplate.opsForList().leftPush("list1",list1);
        redisTemplate.opsForList().rightPush("list2",list2);
        List<String> resultList1=(List<String>)redisTemplate.opsForList().leftPop("list1");
        List<String> resultList2=(List<String>)redisTemplate.opsForList().rightPop("list2");
        System.out.println("resultList1:"+resultList1);
        System.out.println("resultList2:"+resultList2);
    }
```

4. RedisTemplate 储存 key-value

```
/**
     * redis 储存 key-value
     */
    @RequestMapping("/key/value")
    public void redisKeyValue(){
        System.out.println("缓存正在设置。。。。。。。。。");
        redisTemplate.opsForValue().set("111","111");
        redisTemplate.opsForValue().set("222","222");
        redisTemplate.opsForValue().set("333","333");
        redisTemplate.opsForValue().set("444","444");
        System.out.println("缓存已经设置完毕。。。。。。。");
        String result1=redisTemplate.opsForValue().get("111").toString();
        String result2=redisTemplate.opsForValue().get("222").toString();
        String result3=redisTemplate.opsForValue().get("333").toString();
        System.out.println("缓存结果为：result："+result1+"   "+result2+"    "+result3);
    }
```

5. RedisTemplate 储存对象

```
/**
     * redis 储存对象
     */
```

```
@RequestMapping("/Object")
public void redisObject(){
    User user = new User();
    user.setId(11);
    user.setName("yi");
    user.setPhone("123456");
    user.setAge(22);
    redisTemplate.opsForValue().set("user", user);
    User users = (User) redisTemplate.opsForValue().get("user");
    System.out.println(users);
}
```

6. StringRedisTemplate 储存对象

```
/**
 * 存储 json 字符串对象和取出
 */
@RequestMapping("/template")
public void StringRedisTemplate(){
    User user = new User();
    user.setId(11);
    user.setName("yi");
    user.setPhone("123456");
    user.setAge(22);
    template.opsForValue().set("user", JSON.toJSONString(user));
    String str = template.opsForValue().get("user");
    User userName = JSON.parseObject(str, User.class);
    System.out.println(userName);
}
```

7. redis 之 pipeline

```
public String tsetRedis(){
    Long time = System.currentTimeMillis();
    for (int i = 0; i < 10000; i++) {
        stringRedisTemplate.opsForValue().set("yi" + i, "wo" + i);
    }
    Long time1 = System.currentTimeMillis();
    System.out.println("耗时： " + (time1 - time));
    long time4 = System.currentTimeMillis();
    stringRedisTemplate.executePipelined(new SessionCallback<Object>() {
        @Override
        public <K, V> Object execute(RedisOperations<K, V> redisOperations) throws
DataAccessException {
            for (int i = 0; i < 10000; i++) {
                stringRedisTemplate.opsForValue().set("qiang" + i, "wo" + i);
            }
```

```
                return null;    //RedisTemplate 执行 executePipelined 方法是有返回值的
            }
        });
        Long time2 = System.currentTimeMillis();
        System.out.println("耗时： " + (time2 - time4));
        return "redis 正常耗时： " + (time1 - time) + "<br/>" + "redis 管道耗时： " + (time2 - time4);
    }
```

消息队列

28、消息队列的使用场景

校验用户名等信息，如果没问题会在数据库中添加一个用户记录
如果是用邮箱注册会给你发送一封注册成功的邮件，手机注册则会发送一条短信
分析用户的个人信息，以便将来向他推荐一些志同道合的人，或向那些人推荐他
发送给用户一个包含操作指南的系统通知

29、消息的重发补偿解决思路

可靠消息服务定时查询状态为已发送并超时的消息
可靠消息将消息重新投递到 MQ 组件中
下游应用监听消息，在满足幂等性的条件下，重新执行业务。
下游应用通知可靠消息服务该消息已经成功消费。
通过消息状态确认和消息重发两个功能，可以确保上游应用、可靠消息服务和下游应用数据的最终一致性。

30、消息的幂等性解决思路

1 查询操作

查询一次和查询多次，在数据不变的情况下，查询结果是一样的。select 是天然的幂等操作

2 删除操作

删除操作也是幂等的，删除一次和多次删除都是把数据删除。(注意可能返回结果不一样，删除的数据不存在，返回 0，删除的数据多条，返回结果多个)

3 唯一索引，防止新增脏数据

比如：支付宝的资金账户，支付宝也有用户账户，每个用户只能有一个资金账户，怎么防止给用户创建资金账户多个，那么给资金账户表中的用户 ID 加唯一索引，所以一个用户新增成功一个资金账户记录

4token 机制，防止页面重复提交

5 悲观锁

获取数据的时候加锁获取

```
select * from table_xxx where id='xxx' for update;
```

注意：id 字段一定是主键或者唯一索引，不然是锁表，会死人的

悲观锁使用时一般伴随事务一起使用，数据锁定时间可能会很长，根据实际情况选用

乐观锁

乐观锁只是在更新数据那一刻锁表，其他时间不锁表，所以相对于悲观锁，效率更高。

6 分布式锁

还是拿插入数据的例子，如果是分布是系统，构建全局唯一索引比较困难，例如唯一性的字段没法确定，这时候可以引入分布式锁，通过第三方的系统(redis 或 zookeeper)，在业务系统插入数据或者更新数据，获取分布式锁，然后做操作，之后释放锁，这样其实是把多线程并发的锁的思路，引入多多个系统，也就是分布式系统中得解决思路。

7select + insert

并发不高的后台系统，或者一些任务 JOB，为了支持幂等，支持重复执行，简单的处理方法是，先查询下一些关键数据，判断是否已经执行过，在进行业务处理，就可以了

注意：核心高并发流程不要用这种方法

8 状态机幂等

在设计单据相关的业务，或者是任务相关的业务，肯定会涉及到状态机(状态变更图)，就是业务单据上面有个状态，状态在不同的情况下会发生变更，一般情况下存在有限状态机，这时候，如果状态机已经处于下一个状态，这时候来了一个上一个状态的变更，理论上是不能够变更的，这样的话，保证了有限状态机的幂等。

9 对外提供接口的 api 如何保证幂等

如银联提供的付款接口：需要接入商户提交付款请求时附带：source 来源，seq 序列号

source+seq 在数据库里面做唯一索引，防止多次付款，(并发时，只能处理一个请求)

31、消息的堆积解决思路

如果还没开始投入使用 kafka，那应该在设计分区数的时候，尽量设置的多点（当然也不要太大，太大影响延迟，具体可以参考我前面提到的文章），从而提升生产和消费的并行度，避免消费太慢导致消费堆积。

增大批次

瓶颈在消费吞吐量的时候，增加批次也可以改善性能

增加线程数

如果一些消费者组中的消费者线程还是有 1 个消费者线程消费多个分区的情况，建议增加消费者线程。尽量 1 个消费者线程对应 1 个分区，从而发挥现有分区数下的最大并行度。

32、自己如何实现消息队列

大体上的设计是由一条线程 1 执行从等待列表中获取任务插入任务队列再由线程池中的线程从任务队列中取出任务去执行。

添加一条线程 1 主要是防止在执行耗时的任务时阻塞主线程.当执行耗时任务时,添加的任务的操作快于取出任务的操作。

当任务队列长度达到最大值时,线程 1 将被阻塞,等待线程 2,3...从任务队列取出任务执行。

33、如何保证消息的有序性

通过轮询所有队列的方式来确定消息被发送到哪一个队列（负载均衡策略）。订单号相同的消息会被先后发送到同一个队列中，在获取到路由信息以后，会根据算法来选择一个队列，同一个 OrderId 获取到的肯定是同一个队列。