https://my.oschina.net/xiaomingnevermind/blog/1788099

核心篇

数据存储

* MySQL 索引使用的注意事项

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。

* 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。

* 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）。  
以上摘抄自： <http://www.infoq.com/cn/articles/key-steps-and-likely-problems-of-split-table>

* 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。  
摘抄自：<http://blog.csdn.net/jiangpingjiangping/article/details/78069480>

* 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 = mysql*query(  
'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。  
整理自：<https://cloud.tencent.com/developer/article/1054203>

* 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。  
整理自： <http://onwise.xyz/2017/04/20/mysql-%E6%AD%BB%E9%94%81%E9%97%AE%E9%A2%98%E5%8F%8A%E8%A7%A3%E5%86%B3/>

* 存储引擎的 InnoDB 与 MyISAM

　 ◆1.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%”

* 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。

* 为什么要用 B-tree

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。

* 聚集索引与非聚集索引的区别

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。

* limit 20000 加载很慢怎么解决

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。

* 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
  3. 使用UUID to Int64的方法
  4. Redis生成ID
  5. Twitter的snowflake算法
  6. 利用zookeeper生成唯一ID
  7. MongoDB的ObjectId
* 选择合适的数据存储方案

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。  
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。  
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。  
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。  
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。  
摘抄自：<http://blog.720ui.com/2017/db_better_db_use/>

* 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。  
<https://github.com/qianjiahao/MongoDB/wiki/MongoDB%E4%B9%8B_id%E7%94%9F%E6%88%90%E8%A7%84%E5%88%99>

* 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。  
摘抄自： <http://blog.720ui.com/2017/mongodb_core_use/>

* 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。

* 聊聊 ElasticSearch 使用场景

全文搜索，这个是用的最多的。加上分词插件、拼音插件什么的可以做成强大的全文搜索引擎。  
数据库，挺奇葩的用法，因为ES存数相同数据，更费空间，不过确实不错，因为他的强大统计分析汇总能力，再加上分布式P2P扩展能力，现在硬件又那么便宜，所以就有人拿来当数据库了。  
在线统计分析引擎，日志系统。logstash，不用解释了吧。可以实时动态分析数据，很是爽。

缓存使用

* Redis 有哪些类型

在Redis中有五种数据类型  
String----------字符串  
Hash------------字典  
List-------------列表  
Set--------------集合  
Sorted Set------有序集合

* 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。  
作者：zhanglbjames  
链接：<https://www.jianshu.com/p/f09480c05e42>  
來源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

* 聊聊 Redis 使用场景
  1. 缓存
  2. 会话缓存
  3. 时效性
  4. 访问频率
  5. 计数器
  6. 社交列表
  7. 记录用户判定信息
  8. 交集、并集和差集
  9. 热门列表与排行榜
  10. 最新动态
  11. 消息队列  
      摘抄自：<http://blog.720ui.com/2017/redis_core_use/>
* Redis 持久化机制

redis有两种持久化机制RDB与AOF。  
摘抄自： <http://shanks.leanote.com/post/Untitled-55ca439338f41148cd000759-22>

* Redis 如何实现持久化

RDB持久化方式会在一个特定的间隔保存那个时间点的一个数据快照。  
AOF持久化方式则会记录每一个服务器收到的写操作。在服务启动时，这些记录的操作会逐条执行从而重建出原来的数据。写操作命令记录的格式跟Redis协议一致，以追加的方式进行保存。  
Redis的持久化是可以禁用的，就是说你可以让数据的生命周期只存在于服务器的运行时间里。  
两种方式的持久化是可以同时存在的，但是当Redis重启时，AOF文件会被优先用于重建数据。

* Redis 集群方案与实现

客户端分片  
基于代理的分片  
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。

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 为什么是单线程的

因为CPU不是Redis的瓶颈。Redis的瓶颈最有可能是机器内存或者网络带宽。（以上主要来自官方FAQ）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。

* 缓存奔溃
  1. 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。
  2. 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。
* 缓存降级

页面降级：在大促或者某些特殊情况下，某些页面占用了一些稀缺服务资源，在紧急情况下可以对其整个降级，以达到丢卒保帅；  
页面片段降级：比如商品详情页中的商家部分因为数据错误了，此时需要对其进行降级；  
页面异步请求降级：比如商品详情页上有推荐信息/配送至等异步加载的请求，如果这些信息响应慢或者后端服务有问题，可以进行降级；  
服务功能降级：比如渲染商品详情页时需要调用一些不太重要的服务：相关分类、热销榜等，而这些服务在异常情况下直接不获取，即降级即可；  
读降级：比如多级缓存模式，如果后端服务有问题，可以降级为只读缓存，这种方式适用于对读一致性要求不高的场景；  
写降级：比如秒杀抢购，我们可以只进行Cache的更新，然后异步同步扣减库存到DB，保证最终一致性即可，此时可以将DB降级为Cache。  
爬虫降级：在大促活动时，可以将爬虫流量导向静态页或者返回空数据，从而保护后端稀缺资源。  
自动开关降级  
自动降级是根据系统负载、资源使用情况、SLA等指标进行降级。  
超时降级  
当访问的数据库/http服务/远程调用响应慢或者长时间响应慢，且该服务不是核心服务的话可以在超时后自动降级；比如商品详情页上有推荐内容/评价，但是推荐内容/评价暂时不展示对用户购物流程不会产生很大的影响；对于这种服务是可以超时降级的。如果是调用别人的远程服务，和对方定义一个服务响应最大时间，如果超时了则自动降级。  
摘抄自： <http://jinnianshilongnian.iteye.com/blog/2306477>

* 使用缓存的合理性问题
  1. 热点数据，缓存才有价值
  2. 频繁修改的数据，看情况考虑使用缓存
  3. 数据不一致性
  4. 缓存更新机制
  5. 缓存可用性
  6. 缓存服务降级
  7. 缓存预热
  8. 缓存穿透  
     摘抄自： <http://blog.720ui.com/2016/redis_action_01_use_core/>

消息队列

* 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

* 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。

* 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
  3. token机制，防止页面重复提交
  4. 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
  5. 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。
  7. select + 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)  
     摘抄自： <http://825635381.iteye.com/blog/2276077>
* 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。  
摘抄自： <https://kaimingwan.com/post/framworks/kafka/kafkaxiao-xi-dui-ji-chu-li>

* 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。  
作者：DrJasonZhang  
链接：<https://www.jianshu.com/p/2d2271ecc64d>  
來源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

* 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。