1.MySQL

1-1：为什么要使用数据库

因为文件保存数据存在几点缺陷比如 文件的安全性问题。 文件不利于查询和对数据的管理。 文件不利于存放海量数据 文件在程序中控制不方便 为了解决上述问题，专家们设计出更加利于管理数据的东西，数据库

1-2: 什么是SQL?

结构化查询语言 其实就是定义了操作所有关系型数据库的规则

1-3：什么是MySQL?

Mysql是一种关系型数据库管理系统，

2. 关系型数据库与非关系型数据库

2-1：非关系型数据库和关系型数据库定义

关系型数据库： 指采用了关系模型来组织数据的数据库。 非关系型数据库：指非关系型的，分布式的，且一般不保证遵循ACID原则的数据存储系统。

2-2：关系型数据库优缺点

优点 1.容易理解：二维表结构是非常贴近逻辑世界的一个概念， 关系模型相对网状、层次等其他模型来说更容易理解 2.使用方便：通用的SQL语言使得操作关系型数据库非常方便 3.易于维护：丰富的完整性 实体完整性、参照完整性和用户定义的完整性 大大减低了数据冗余和数据不一致的概率 存在的问题 1.网站的用户并发性非常高， 往往达到每秒上万次读写请求， 对于传统关系型数据库来说，硬盘I/O是一个很大的瓶颈 2.网站每天产生的数据量是巨大的， 对于关系型数据库来说， 在一张包含海量数据的表中查询，效率是非常低的 3.在基于web的结构当中， 数据库是最难进行横向扩展的， 当一个应用系统的用户量和访问量与日俱增的时候， 数据库却没有办法像web server和app server 那样简单的通过添加更多的硬件和服务节点来扩展性能和负载能力。 当需要对数据库系统进行升级和扩展时，往往需要停机维护和数据迁移。 4.性能欠佳：在关系型数据库中， 导致性能欠佳的最主要原因是多表的关联查询， 以及复杂的数据分析类型的复杂SQL报表查询。 为了保证数据库的ACID特性， 必须尽量按照其要求的范式进行设计， 关系型数据库中的表都是存储一个格式化的数据结构。

2-3：非关系型数据库优缺点

优点

1. 用户可以根据需要去添加自己需要的字段， 为了获取用户的不同信息，不像关系型数据库中， 要对多表进行关联查询。 仅需要根据id取出相应的value就可以完成查询。
2. 适用于SNS(Social Networking Services)中， 例如facebook，微博。系统的升级，功能的增加， 往往意味着数据结构巨大变动，这一点关系型数据库难以应付， 需要新的结构化数据存储。 由于不可能用一种数据结构化存储应付所有的新的需求， 因此，非关系型数据库严格上不是一种数据库， 应该是一种数据结构化存储方法的集合。 不足： 只适合存储一些较为简单的数据， 对于需要进行较复杂查询的数据，关系型数据库显的更为合适。不适合持久存储海量数据

2-4：非关系型数据库和关系型数据库区别

1.成本：Nosql数据库简单易部署， 基本都是开源软件，不需要像使用Oracle那样花费大量成本购买使用， 相比关系型数据库价格便宜。 2.查询速度：Nosql数据库将数据存储于缓存之中， 而且不需要经过SQL层的解析， 关系型数据库将数据存储在硬盘中， 自然查询速度远不及Nosql数据库。 3.存储数据的格式：Nosql的存储格式是key,value形式、文档形式、图片形式等等， 所以可以存储基础类型以及对象或者是集合等各种格式，而数据库则只支持基础类型。 4.扩展性：关系型数据库有类似join这样的多表查询机制的限制导致扩展很艰难。 Nosql基于键值对，数据之间没有耦合性，所以非常容易水平扩展。 5.持久存储：Nosql不使用于持久存储，海量数据的持久存储，还是需要关系型数据库 6.数据一致性：非关系型数据库一般强调的是数据最终一致性， 不像关系型数据库一样强调数据的强一致性， 从非关系型数据库中读到的有可能还是处于一个中间态的数据， Nosql不提供对事务的处理。

3.三大范式

3-1：数据库三大范式是什么

第一范式: 每个列都不可以再拆分. 第二范式: 非主键列完全依赖于主键,而不能是依赖于主键的一部分. 第三范式: 非主键列只依赖于主键,不依赖于其他非主键.

3-2：三大范式举例

4.数据库的数据类型

4-1：mysql的数据类型

整数类型：BIT、BOOL、TINY INT、 SMALL INT、MEDIUM INT、 INT、 BIG INT 浮点数类型：FLOAT、DOUBLE、DECIMAL 字符串类型：CHAR、VARCHAR、TINY TEXT、 TEXT、MEDIUM TEXT、LONGTEXT、 TINY BLOB、BLOB、MEDIUM BLOB、LONG BLOB 日期类型：Date、DateTime、TimeStamp、Time、Year

4-2：varchar与char的区别

char长度固定，varchar长度可以变化

比如说字符串abc char（10）就是占用10个字节， varchar（10）只占用3个字节，10只是最大值

4-2-1：varchar(50)中50的涵义

varchar(50)中50的涵义最多存放50个字符

4-3：int(20)中20的涵义

int(M)只是用来显示数据的宽度， 比如说int（20），mysql会自动补0

4-4：FLOAT和DOUBLE的区别是什么?

1. 在内存中占有的字节数不同　　单精度浮点数在机内存占4个字节　　双精度浮点数在机内存占8个字节
2. 有效数字位数不同　　单精度浮点数有效数字8位　　双精度浮点数有效数字16位
3. 数值取值范围

4. 在程序中处理速度不同　　一般来说，CPU处理单精度浮点数的速度比处理双精度浮点数快

5.SQL生命周期

1. 建立服务器与数据库连接
2. 数据库拿到SQL
3. 解析执行
4. 读取数据到内存，进行业务逻辑处理
5. 发给客户端
6. 关闭连接，释放资源

6.MySQL预编译

指的是数据库驱动在发送 sql 语句和参数 给 DBMS 之前对 sql 语句进行编译， 这样 DBMS 执行 sql 时，就不需要重新编译。

6-1：预编译出现的原因

1、很多情况下，一条SQL语句可能会反复执行，或者每次执行的时候只有个别的值不同 2、比如query的where条件的值不同， update的set的值不同,insert的values值不同，都会造成SQL语句的不同。 3、每次因为这些值的不同就进行词法语义解析、优化、制定执行计划，就会很影响效率。

6-2：预编译的好处

1、预编译之后的 SQL 多数情况下可以直接执行，DBMS 不需要再次编译。 2、越复杂的SQL，编译的复杂度将越大，预编译阶段可以合并多次操作为一个操作。 3、相同的预编译 SQL 可以重复利用。 把一个 SQL 预编译后产生的 PreparedStatement 对象缓存下来，　　下次对于同一个 SQL， 可以直接使用这个缓存的 PreparedState 对象。 4、可以将这类SQL语句中的值用占位符替代，不需要每次编译，可以直接执行，　　只需执行的时候，直接将每次请求的不同的值设置到占位符的位置。 5、预编译可以视为将sql语句模板化或者说参数化。

7.SQL注入

7-1：SQL注入简介

通过把SQL命令插入到Web表单提交或输入域名或页面请求的查询字符串， 最终达到欺骗服务器执行恶意的SQL命令。 它不是利用操作系统的BUG来实现攻击， 而是针对程序员编程时的疏忽， 通过SQL语句，实现无帐号登录，甚至篡改数据库。

7-2：SQL注入攻击实例

比如在一个登录界面，要求输入用户名和密码： 可以这样输入实现免帐号登录： 用户名： ‘or 1 = 1 – 密 码：空白 点登陆,如若没有做特殊处理,那么这个非法用户就很得意的登陆进去了. 从理论上说，后台认证程序中的SQL语句还是很正常的

```
String sql = "select * from user_table where username=
' "+userName+" ' and password=' "+password+" '";
```

但是当输入了上面的用户名和密码，上面的SQL语句变化了：

```
SELECT * FROM user_table WHERE username=
''or 1 = 1 -- and password=''
```

条件后面username=”or 1=1 用户名等于 ” 或1=1 那么这个条件一定会成功； 然后后面加两个-，这意味着注释， 它将后面的语句注释， 让他们不起作用， 这样语句永远都能正确执行， 用户轻易骗过系统，获取合法身份。

7-3：SQL注入解决方案

1. PreparedStatement最直接方法

采用预编译语句集，它内置了处理SQL注入的能力， 只要使用它的setXXX方法传值即可。 使用好处： (1).代码的可读性和可维护性. (2).PreparedStatement尽最大可能提高性能. (3).最重要的一点是极大地提高了安全性. 原理： sql注入只对sql语句的准备(编译)过程有破坏作用 而PreparedStatement已经准备好了,执行阶段只是把输入串作为数据处理, 而不再对sql语句进行解析,准备,因此也就避免了sql注入问题.

2. 使用正则表达式过滤传入的参数
3. 字符串过滤 比较通用的一个方法： (||之间的参数可以根据自己程序的需要添加)
4. jsp中调用该函数检查是否包函非法字符
5. JSP页面判断代码： 使用JavaScript在客户端进行不安全字符屏蔽 检查是否含有“’”,”\”,”/”

8.sql编写原则

8-0：SQL的分类

DDL-------数据定义语言 DQL-------数据查询语言 DML-------数据操纵语言 DCL-------数据控制语言

8-1：查询

主要的查询分为了基础查询、条件查询、 排序查询、子查询、 分页查询、多表查询

8-1-1：基础查询

8-1-4：子查询

子查询比如说在select嵌套select

8-1-4-1：子查询的三种情况

1. 子查询结果只要是单行单列， 肯定在 WHERE 后面作为条件
2. 子查询结果是多行单列，结果类似于一个数组，父查询使用 in 运算符

3. 子查询结果只要是多列，肯定在 From 后面作为表

8-1-6：多表查询

先对第一个和第二个表按照两表连接做查询，然后用查询结果和第三个表做连接查询，以此类推

8-1-6-1：笛卡尔积问题

就是两张表的记录 进行一个相乘的操作查询出来的结果就是笛卡尔积, 如果左表有n条记录,右表有m条记录, 笛卡尔积查询出有n*m条记录, 其中往往包含了很多错误的数据,所以这种查询方式并不常用。

8-1-6-2：笛卡尔积的解决方案

添加上连接条件

8-1-6-2-1：五种关联（连接）查询

1. 交叉连接(CROSS JOIN)
2. 内连接(INNER JOIN)
3. 外连接(LEFT JOIN/RIGHT JOIN)
4. 联合查询(UNION 与 UNION ALL)
5. 全连接(FULL JOIN)

8-1-6-2-1-1：各种连接定义

1. 内连接: 只连接匹配的行
2. 左外连接: 包含左边表的全部行，以及右边表中全部匹配的行
3. 右外连接: 包含右边表的全部行，以及左边表中全部匹配的行
4. 全外连接: 包含左、右两个表的全部行。
5. 交叉连接: 生成笛卡尔积 - 它不使用任何匹配或者选取条件， 而是直接将一个数据源中的每个行与另一个数据源 的每个行都一一匹配

8-1-6-2-1-2：UNION与UNION ALL的区别?

union：对两个结果集进行并集操作，不包括重复行，同时进行默认规则的排序； union All：对两个结果集进行并集操作，包括重复行，不进行排序；

8-2：常见函数

分为了单行函数与分组函数

8-2-1：单行函数

8-2-3：聚合函数

AVG() 返回某列的平均值 COUNT() 返回某列的行数 MAX() 返回某列的最大值 MIN() 返回某列的最小值 SUM() 返回某列值之和

8-2-3-1：count(*)、count(1)、count(column)的区别

count()*对行的数目进行计算, 包含NULL，count(1) 这个用法和count()*的结果是一样的 count(column)对特定的列的值具有的行数进行计算, 不包含NULL值。

8-3：sql执行顺序

1. from:需要从哪个数据表检索数据
2. join：联合多表查询返回记录时，并生成一张临时表
3. on：在生成临时表时使用的条件
4. where:过滤表中数据的条件
5. group by:如何将上面过滤出的数据分组
6. having:对上面已经分组的数据进行过滤的条件
7. select:查看结果集中的哪个列，或列的计算结果
8. order by :按照什么样的顺序来查看返回的数据
9. limit：限制查询结果返回的数量

8-3-1：一条sql执行过程

1. 连接Mysql 客户端连接MySql服务时是半双工通信， 客户端和服务端交互发送数据时必须一次发送完毕， 在传输特别大的数据包时系统性能开销非常大， 所以当客户端使用insert等语句发送大量的数据包时服务端就会拒绝连接， 原因是服务端默认限制客户端发送的数据包大小不能超过4MB， 这点可以通过修改MySql参数来更改大小限制， 或者将需要发送的数据包在客户端进行分批发送处理。 同理，在服务端返回给客户端数据时也要避免大量的数据包传输， 所以要避免使用不带Limit的查询语句进行批量查询， 或者可以先使用count做数据量预估，根据数据量进行分批查询。
2. 缓存与解析器 当缓存是打开的情况下，MySql服务端拿到Sql语句后， 首先会到缓存判断是否有完全一致的Sql语句查询记录（判断时Sql语句连空格都不能有误差）， 有就将相应的结果集返回给客户端， 当缓存中不存在时，会将Sql语句交给解析器来处理， 解析器通过关键字将SQL语句进行解析，并生成一棵对应的"解析树"， 接着会验证Sql语句是否有词法、语法等错误， 例如，它将验证是否使用错误的关键字，或者使用关键字的顺序是否正确等， 再或者它还会验证引号是否能前后正确匹配。
3. 预处理器（Preprocessor） 预处理器会对解析树进行进一步检查解析树是否合法， 例如，这里将检查数据表和数据列是否存在， 还会解析名字和别名， 看看它们是否有歧义。 接着预处理器会验证权限。 这通常很快，除非服务器上有非常多的权限配置。
4. 优化器（查询优化器 Query Optimizer） 一条Sql语句并不是只有一种执行路径。 当优化器拿到预处理器发来的解析树后， 会根据解析树生成不同的执行路径， 这些执行路径就是常说的执行计划（Execution Plan）， 优化器会对这些执行计划计算对应的开销（cost）， 当得到不同的执行计划与相应的开销后， 优化器将它认为最佳的执行计划去交给下一个部件去执行
5. 查询执行引擎 当优化器将执行计划交到查询执行引擎手里时， 剩下的任务就简单多了， 查询执行引擎调用相应的API接口来操作存储引擎， 将存储引擎返回的查询结果返回给客户端， 如果缓存开启的情况也会在缓存中进行缓存。
6. 存储引擎 存储引擎就有很多了，像Mysql5.5版本前默认使用的MyIsan存储引擎， 5.5版本后默认使用的InnoDB存储引擎，

8-3-2：truncate、 delete区别

1、TRUNCATE在各种表上无论是大的还是小的都非常快。 如果有ROLLBACK命令DELETE将被撤销，而TRUNCATE则不会被撤销。 2、truncate不能进行回滚操作。 3、truncate不触发任何delete触发器。 4、当表被truncate后，这个表和索引所占用的空间会恢复到初始大小， 而delete操作不会减少表或索引所占用的空间。 5、不能truncate一个带有外键的表，如果要删除首先要取消外键，然后再删除。 DELETE语句执行删除的过程是每次从表中删除一行， 并且同时将该行的的删除操作作为事务记录 在日志中保存以便进行进行回滚操作。

8-3-3：mysql的having用法

having字句可以让我们筛选成组后的各种数据， where字句在聚合前先筛选记录，也就是说作用在group by和having字句前。 而 having子句在聚合后对组记录进行筛选 我的理解就是真实表中没有此数据，这些数据是通过一些函数生存。

8-3-4：mysql中 in 和 exists 区别

1. exists()适合B表比A表数据大的情况
2. 当A表数据与B表数据一样大时,in与exists效率差不多,可任选一个使用

8-4：增删改查库表

8-4-1：一条sql更新/删除/增加语句时怎么执行的

1.执行器先取到ID=2这行。因为ID是主键，引擎直接用树搜索找到这一行。 如果ID=2这一行所在的数据页本来就在内存中， 就直接返回给执行器； 否则需要先从磁盘读入内存，然后再返回。 2.执行器拿到引擎给的行数据，把这个值加1， 比如原来N，现在是N+1，得到新的一行数据， 再调用引擎接口写入这行数据。 3.引擎将这行新数据更新到内存中， 同时将这个更新/删除/增加操作记录在redo log里面， 此时redo log处于prepare状态。 然后告知执行器执行完成了，随时可以提交事务。 4.执行器生成这个操作的binlog，并把binlog写入磁盘中。 5.执行器调用引擎的提交事务接口， 引擎把刚刚写入的redo log改成提交（commit）状态， 更新完成。

8-4-1-1：日志由来

因为MySQL整体来看，其实就有两块： 一块是Server层，它主要做的是MySQL功能层面的事情； 还有一块是引擎层，负责存储相关的具体事宜。 而redo log是InnoDB引擎特有的日志， 而Server层也有自己的日志，称为binlog（归档日志）。

8-4-1-1-1：什么是binlog

binlog其实就是记录了数据库表结构和表数据变更 我们的数据是保存在数据库里边的嘛， 假设我们对某个商品的某个字段的内容改了（数据库变更）， 而用户检索的出来数据是走搜索引擎的。 为了让用户能搜到最新的数据，我们需要把引擎的数据也改掉。 数据库的变更，搜索引擎的数据也需要变更。

binlog：存储着每条变更的SQL语句

8-4-1-1-1-1：binlog一般用来做什么

主要有两个作用：复制和恢复数据 MySQL一般都是一主多从结构的， 从服务器需要与主服务器的数据保持一致，这就是通过binlog来实现的 数据库的数据消失了，我们可以通过binlog来对数据进行恢复。 因为binlog他记录了数据库的变更，所以用binlog进行赋值和恢复数据

8-4-1-1-1-2：MySQL的binlog有有几种录入格式？分别有什么区别?

有三种格式,statement,row和mixed. statement模式下,记录单元为语句. 即每一个sql造成的影响会记录. 由于sql的执行是有上下文的, 因此在保存的时候需要保存相关的信息, 同时还有一些使用了函数之类的语句无法被记录复制.

row级别下,记录单元为每一行的改动, 基本是可以全部记下来但是由于很多操作, 会导致大量行的改动(比如alter table), 因此这种模式的文件保存的信息太多, 日志量太大.

mixed. 一种折中的方案, 普通操作使用statement记录, 当无法使用statement的时候使用row.

8-4-1-1-2：redo log

Mysql的基本存储结构是页(记录都存在页里边)， 所以MySQL是先把这条记录所在的页找到， 然后把该页加载到内存中，将对应记录进行修改。 当我们修改的时候，写完内存了，但数据还没真正写到磁盘的时候。 此时我们的数据库挂了，我们可以根据redo log来对数据进行恢复。 因为redo log是顺序IO，所以写入的速度很快，并且redo log记载的是物理变化（xxxx页做了xxx修改）， 文件的体积很小，恢复速度很快。

8-4-1-1-3：bin log和redo log比较

1. 存储内容来说 binlog记载的是update/delete/insert这样的SQL语句， 而redo log记载的是物理修改的内容（xxxx页修改了xxx）。
2. 功能 redo log的作用是为持久化而生的。写完内存， 如果数据库挂了， 那我们可以通过redo log来恢复内存还没来得及刷到磁盘的数据， 将redo log加载到内存里边， 那内存就能恢复到挂掉之前的数据了。 binlog的作用是复制和恢复而生的。 主从服务器需要保持数据的一致性， 通过binlog来同步数据。 如果整个数据库的数据都被删除了， binlog存储着所有的数据变更情况， 那么可以通过binlog来对数据进行恢复。
3. 第三方面来说，写入内容来说 redo log事务开始的时候， 就开始记录每次的变更信息， 而binlog是在事务提交的时候才记录。

8-4-1-1-3-1：我写其中的某一个log，失败了，那会怎么办?

如果说是先写redo log，再写binlog， 如果写redo log失败了， 那我们就认为这次事务有问题，回滚，不再写binlog。 如果写redo log成功了， 写binlog，写binlog写一半了，但失败了 我们还是会对这次的事务回滚，将无效的binlog给删除 （因为binlog会影响从库的数据，所以需要做删除操作） 如果写redo log和binlog都成功了，那这次算是事务才会真正成功。

8-4-1-1-4：两阶段提交意义

“两阶段提交”是为了让两份日志之间的逻辑一致。 由于redo log和binlog是两个独立的逻辑，如果不用两阶段提交 会出现错误 比如说 先写redo log后写binlog。 假设在redo log写完，binlog还没有写完的时候，MySQL进程异常重启。redo log写完之后， 系统即使崩溃，仍然能够把数据恢复回来， 所以恢复后这一行c的值是1。 但是由于binlog没写完就crash了， 这时候binlog里面就没有记录这个语句。 因此，之后备份日志的时候， 存起来的binlog里面就没有这条语句。 然后发现，如果需要用这个binlog来恢复临时库的话， 由于这个语句的binlog丢失， 这个临时库就会少了这一次更新， 恢复出来的这一行c的值就是0， 与原库的值不同。 如果先写binlog后写redo log。 如果在binlog写完之后crash， 由于redo log还没写， 崩溃恢复以后这个事务无效， 所以这一行c的值是0。 但是binlog里面已经记录了“把c从0改成1”这个日志。 所以，在之后用binlog来恢复的时候就多了一个事务出来， 恢复出来的这一行c的值就是1， 与原库的值不同。

8-4-1-1-4-1：MySQL如何保证redo log和binlog的数据是一致的

MySQL通过两阶段提交来保证redo log和binlog的数据是一致的。 阶段1：InnoDBredo log 写盘，InnoDB 事务进入 prepare 状态 阶段2：binlog 写盘，InooDB 事务进入 commit 状态 每个事务binlog的末尾，会

记录一个 XID event， 标志着事务是否提交成功， 也就是说，恢复过程中， binlog 最后一个 XID event 之后的内容都应该被 purge。

8-4-1-1-4-2：如果整个数据库的数据都被删除了，那我可以用redo log的记录来恢复吗?

不能 因为功能的不同，redo log 存储的是物理数据的变更， 如果我们内存的数据已经刷到了磁盘了，那redo log的数据就无效了。 所以redo log不会存储着历史所有数据的变更，文件的内容会被覆盖的。

8-4-2：一条sql查询语句时怎么执行的

1. 先连接到这个数据库上，这时候就是连接器。 连接器负责跟客户端建立连接、获取权限、维持和管理连接。
2. 连接建立完成后，就可以执行select语句了。进行查询缓存。 MySQL拿到一个查询请求后，会先到查询缓存看看，之前是不是执行过这条语句。 之前执行过的语句及其结果可能会以key-value对的形式， 被直接缓存在内存中。key是查询的语句，value是查询的结果。 如果你的查询能够直接在这个缓存中找到key，那么这个value就会被直接返回给客户端。 如果语句不在查询缓存中，就会继续后面的执行阶段。执行完成后，执行结果会被存入查询缓存中。 可以看到，如果查询命中缓存，MySQL不需要执行后面的复杂操作， 就可以直接返回结果，这个效率会很高。
3. 如果没有命中查询缓存，就要开始真正执行语句了。 MySQL需要知道你要做什么，因此需要对SQL语句做解析。
4. 经过了分析器，MySQL就知道你要做什么了。在开始执行之前， 还要先经过优化器的处理。优化器是在表里面有多个索引的时候， 决定使用哪个索引； 或者在一个语句有多表关联（join）的时候，决定各个表的连接顺序。
5. 于是就进入了执行器阶段，开始执行语句。 开始执行的时候，要先判断一下你对这个表user有没有执行查询的权限， 如果没有，就会返回没有权限的错误， 如果有权限，就打开表继续执行。 打开表的时候，执行器就会根据表的引擎定义， 去使用这个引擎提供的接口

8-4-2-1：mysql有关权限的表都有哪几个

1. user权限表：记录允许连接到服务器的用户帐号信息，里面的权限是全局级的。
2. db权限表：记录各个帐号在各个数据库上的操作权限。
3. table_priv权限表：记录数据表级的操作权限。
4. columns_priv权限表：记录数据列级的操作权限。
5. host权限表：配合db权限表对给定主机上数据库级操作权限作更细致的控制。 这个权限表不受GRANT和REVOKE语句的影响。

8-5：常见约束

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。

8-5-1：字段为什么要求定义为not null?

null值会占用更多的字节,且会在程序中造成很多与预期不符的情况

8-6：键

8-6-1：超键、候选键、主键、外键分别是什么?

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：是最小超键，即没有冗余元素的超键。 主键：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。

8-6-2：为什么用自增列作为主键

如果表使用自增主键，那么每次插入新的记录， 记录就会顺序添加到当前索引节点的后续位置， 当一页写满，就会自动开辟一个新的页 如果使用非自增主键，由于每次插入主键的值近似于随机， 因此每次新纪录都要被插到现有索引页得中间某个位置， 此时MySQL不得不为了将新记录插到合适位置而移动数据， 甚至目标页面可能已经被回写到磁盘上而从缓存中清掉， 此时又要从磁盘上读回来，这增加了很多开销， 同时频繁的移动、分页操作造成了大量的碎片， 得到了不够紧凑的索引结构， 后续不得不通过OPTIMIZE TABLE来重建表并优化填充页面。

8-6-2-1：主键使用自增ID还是UUID?

推荐使用自增ID,不要使用UUID.

因为在InnoDB存储引擎中,主键索引是作为聚簇索引存在的, 也就是说,主键索引的B+树叶子节点上存储了 主键索引以及全部的数据(按照顺序), 如果主键索引是自增ID, 那么只需要不断向后排列即可, 如果是UUID,由于到来的ID与原来的大小不确定, 会造成非常多的数据插入,数据移动, 然后导致产生很多的内存碎片, 进而造成插入性能的下降.

8-6-2-1：为什么MySQL不推荐使用uuid或者雪花id作为主键?

自增的主键的值是顺序的, 所以Innodb把每一条记录都存储在一条记录的后面。 当达到页面的最大填充因子时候:

①下一条记录就会写入新的页中， 一旦数据按照这种顺序的方式加载， 主键页就会近乎于顺序的记录填满，提升了页面的最大填充率，不会有页的浪费

②新插入的行一定会在原有的最大数据行下一行, mysql定位和寻址很快， 不会为计算新行的位置而做出额外的消耗

③减少了页分裂和碎片的产生

因为uuid相对顺序的自增id来说是毫无规律可言的, 新行的值不一定要比之前的主键的值要大, 所以innodb无法做到总是把新行插入到索引的最后, 而是需要为新行寻找新的合适的位置从而来分配新的空间。 这个过程需要做很多额外的操作， 数据的毫无顺序会导致数据分布散乱， 将会导致一些问题： 比如说 ①写入的目标页很可能已经刷新到磁盘上并且从缓存上移除， 或者还没有被加载到缓存中， innodb在插入之前不得不先找到 并从磁盘读取目标页到内存中， 这将导致大量的随机IO ②因为写入是乱序的,innodb不得不频繁的做页分裂操作, 以便为新的行分配空间,页分裂导致移动大量的数据， 一次插入最少需要修改三个页以上 ③由于频繁的页分裂，页会变得稀疏并被不规则的填充， 最终会导致数据会有碎片 在把随机值（uuid和雪花id） 载入到聚簇索引(innodb默认的索引类型)以后, 有时候会需要做一次OPTIMEIZE TABLE来重建表并优化页的填充， 这将又需要一定的时间消耗。

8-6-2-2：使用自增id的缺点

①别人一旦爬取你的数据库, 就可以根据数据库的自增id获取到你的业务增长信息， 很容易分析出你的经营情况 ②对于高并发的负载， innodb在按主键进行插入的时候会造成明显的锁争用， 主键的上界会成为争抢的热点， 因为所有的插入都发生在这里， 并发插入会导致间隙锁竞争 ③Auto_Increment锁机制 会造成自增锁的抢夺,有一定的性能损失 、

8-6-2-3：数据库主键自增怎么获取主键值

使用函数 LAST_INSERT_ID() 比如说如查看最新一次自增得到的id： select LAST_INSERT_ID(); 或者在mybatis中设置userGeneratedKeys属性值为true:使用自动增长的主键。 使用keyProperty设置把主键值设置给哪一个属性

```
<insert id="addEmp"
        parameterType="com.neuedu.mybatis.bean.Employee"
        useGeneratedKeys="true" keyProperty="id" databaseId="mysql">
     insert into tbl_employee(last_name,email,gender)
     values(#{lastName},#{gender},#{email})
</insert>
```

8-6-3：为什么要尽量设定一个主键?

主键是数据库确保数据行在整张表唯一性的保障, 即使业务上本张表没有主键, 也建议添加一个自增长的ID列作为主键. 设定了主键之后, 在后续的删改查的时候可能 更加快速以及确保操作数据范围安全.

8-7：视图

视图是虚拟的表

1. 重用SQL语句；
2. 简化复杂的SQL操作（可以方便的重用它而不必知道它的基本查询细节）；
3. 使用表的组成部分而不是整个表；
4. 保护数据（可以给用户授予表的部分访问权限而不是整个表的访问权限）；
5. 更改数据格式和表示（视图可返回与底层表的表示和格式不同的数据）。

8-8：存储过程

预先用SQL语句写好并用一个指定的名称存储起来，只需调用execute,即可自动完成命令。

8-8-1：存储过程有哪些优缺点?

1. 一般SQL语句每执行一次就编译一次,使用存储过程创建进行编译，可提高数据库执行速度。
2. 当对数据库进行复杂操作时 (如对多个表进行Update,Insert,Query,Delete时）， 可将此复杂操作用存储过程封装起来。
3. 存储过程可以重复使用,可减少数据库开发人员的工作量
4. 安全性高,可设定只有某此用户才具有对指定存储过程的使用权

8-9：触发器

数据库触发器是在数据库中发生特定操作时运行的特殊存储过程

8-9-1：触发器的使用场景有哪些?

1. 复杂的审计 可以使用触发器来跟踪对表所做的更改。 比如说认为这是涉及敏感操作的信息，进行了更改。
2. 执行业务规则 每次添加或修改客户记录时检查客户状态。

8-10：窗口函数

8-10-1：什么是窗口函数

具体定义记不太清楚，因为 经常会遇到需要在每组内排名，比如说 排名问题：每个部门按业绩来排名 或者找出每个部门排名前N的员工进行奖励

8-10-2：窗口函数的定义

```
<窗口函数> over (partition by <用于分组的列名>
                 order by <用于排序的列名>)
```

8-11：sql实战

8-11-1：去重重复数据

9.事务

9-1：什么是事务

事务是逻辑上的一组操作，要么都执行，要么都不执行。

9-2：数据库事务特性

1. 原子性（Atomicity）： 事务是最小的执行单位， 不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
2. 一致性（Consistency）： 执行事务前后，数据保持一致， 多个事务对同一个数据读取的结果是相同的；
3. 隔离性（Isolation）： 并发访问数据库时， 一个用户的事务不被其他事务所干扰， 各并发事务之间数据库是独立的；
4. 持久性（Durability）： 一个事务被提交之后。 它对数据库中数据的改变是持久的， 即使数据库发生故障也不应该对其有任何影响。

9-2-1：ACID靠什么保证的

A原子性由undo log日志保证， 它记录了需要回滚的日志信息， 事务回滚时撤销已经执行成功的sql C一致性一般由代码层面来保证 I隔离性由MVCC来保证 D持久性由内存+redo log来保证， mysql修改数据同时在内存和redo log记录这次操作， 事务提交的时候通过redo log刷盘， 宕机的时候可以从redo log恢复

9-2-1-1：什么是undo log

undo log主要有两个作用：回滚和多版本控制(MVCC) 在数据修改的时候，不仅记录了redo log，还记录undo log， 如果因为某些原因导致事务失败或回滚了，可以用undo log进行回滚 undo log主要存储的也是逻辑日志，比如我们要insert一条数据了， 那undo log会记录的一条对应的delete日志。我们要update一条记录时，它会记录一条对应相反的update记录。 因为回滚，跟需要修改的操作相反就好，这样就能达到回滚的目的。因为支持回滚操作， 所以我们就能保证：“一个事务包含多个操作，这些操作要么全部执行，要么全都不执行”。【原子性】 因为undo log存储着修改之前的数据，相当于一个前版本，MVCC实现的是读写不阻塞， 读的时候只要返回前一个版本的数据就行了。

9-2-1-1-1：Undo Log缺陷如何解决?

每个事务提交前将数据和Undo Log写入磁盘， 这样会导致大量的磁盘IO，因此性能很低。 因此引入了另外一种机制来实现持久化， 即Redo Log。 Redo Log记录的是新数据的备份。 在事务提交前，只要将Redo Log持久化即可， 不需要将数据持久化。当系统崩溃时， 虽然数据没有持久化，但是Redo Log已经 持久化。系统可以根据Redo Log的内容， 将所有数据恢复到最新的状态。

9-2-1-2：Java如何保证原子性

1. 第一个方法是循环CAS 只能保证一个共享变量的原子操作。 当对一个共享变量执行操作时， 我们可以使用循环CAS的方式来保证原子操作， 但是对多个共享变量操作时， 循环CAS就无法保证操作的原子性，这个时候就可以用锁 或者有一个取巧的办法， 就是把多个共享变量合并成一个共享变量来操作。 比如有两个共享变量i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij
2. 锁 锁机制保证了只有获得锁的线程能够操作锁定的内存区域。

9-2-1-3：数据库崩溃时事务的恢复机制

为了满足事务的原子性，在操作任何数据之前， 首先将数据备份到一个地方，这个存储数据备份的地方称为UndoLog。 然后进行数据的修改。 如果出现了错误或者用户执行了回滚， 系统可以利用Undo Log中的备份将数据恢复到事务开始之前的状态。

9-3：在并发环境下,事务会发生哪些问题?

1. 脏读 A事务读取B事务尚未提交的更改数据，并在这个数据的基础上进行操作， 这时候如果事务B回滚，那么A事务读到的数据是不被承认的。
2. 不可重复读 不可重复读是指A事务读取了B事务已经提交的更改数据。假 如A在取款事务的过程中，B往该账户转账100，A两次读取的余额发生不一致。
3. 幻读 A事务读取B事务提交的新增数据,会引发幻读问题。幻读一般发生在计算统计数据的事务中， 例如银行系统在同一个事务中两次统计存款账户的总金额， 在两次统计中，刚好新增了一个存款账户，存入了100，这时候两次统计的总金额不一致。
4. 第一类丢失更新 A事务撤销时，把已经提交的B事务的更新数据覆盖了。
5. 第二类丢失更新 A事务覆盖B事务已经提交的数据，造成B事务所做的操作丢失 。

9-3-1：不可重复读和幻读的区别

不可重复读是指读到了已经提交的事务的更改数据（修改或删除）， 幻读是指读到了其他已经提交事务的新增数据。 对于这两种问题解决采用不同的办法， 防止读到更改数据，只需对操作的数据添加行级锁， 防止操作中的数据发生变化； 二防止读到新增数据，往往需要添加表级锁， 将整张表锁定，防止新增数据（oracle采用多版本数据的方式实现）。

9-4：如何解决事务并发问题

采用隔离级别

9-4-1：四大隔离级别

1. 读取未提交： 最低的隔离级别，允许读取尚未提交的数据变更， 可能会导致脏读、幻读或不可重复读
2. 读取已提交： 允许读取并发事务已经提交的数据， 可以阻止脏读，但是幻读或不可重复读仍有可能发生。
3. 可重复读： 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改， 可以阻止脏读和不可重复读，但幻读仍有可能发生。
4. 可串行化： 最高的隔离级别， 该级别可以防止脏读、不可重复读以及幻读。

9-4-1-1：隔离级别的原理

1. READ_UNCOMMITED（读取未提交） 的原理
   - *事务对当前被读取的数据不加锁;*
   - 事务在更新某数据的瞬间（就是发生更新的瞬间）， 必须先对其加行级共享锁，直到事务结束才释放。
2. READ_COMMITED(读取已提交)的原理
   - *事务对当前被读取的数据加 行级共享锁（当读到时才加锁），一旦读完该行，立即释放该行级共享锁;*
   - 事务在更新某数据的瞬间（就是发生更新的瞬间），必须先对其加 行级排他锁，直到事务结束才释放。
3. REPEATABLE READ 的原理:
   - *事务在读取某数据的瞬间（就是开始读取的瞬间），必须先对其加 行级共享锁，直到事务结束才释放;*
   - 事务在更新某数据的瞬间（就是发生更新的瞬间），必须先对其加 行级排他锁，直到事务结束才释放。
4. SERIALIZABLE 的原理:
   - *事务在读取数据时，必须先对其加 表级共享锁 ，直到事务结束才释放;*
   - 事务在更新数据时，必须先对其加 表级排他锁 ，直到事务结束才释放。

9-4-1-1-1：为什么要有事物隔离级别

因为事物隔离级别越高，在并发下会产生的问题就越少， 但同时付出的性能消耗也将越大， 因此很多时候必须在并发性和性能之间做一个权衡。 所以设立了几种事物隔离级别， 以便让不同的项目可以根据自己项目的 并发情况选择合适的事物隔离级别， 对于在事物隔离级别之外会产生的并发问题，在代码中做补偿。

9-4-1-2：MySQL 中RC（读已提交）和RR（可重复读）隔离级别的区别

1. RR支持gap lock，而RC则没有gap lock（间隙锁）。 因为MySQL的RR需要gap lock来解决幻读问题。 而RC隔离级别则是允许存在不可重复读和幻读的。所以RC的并发一般要好于RR；
2. RC 隔离级别，通过 where 条件过滤之后，不符合条件的记录上的行锁， 会释放掉；但是RR隔离级别，即使不符合where条件的记录， 也不会是否行锁和gap lock；所以从锁方面来看，RC的并发应该要好于RR；
3. RC 隔离级别不支持 statement 格式的bin log， 因为该格式的复制，会导致主从数据的不一致； 只能使用 mixed 或者 row 格式的bin log

4. RC隔离级别时，事务中的每一条select语句会读取到他自己执行时已经提交了的记录， 而RR隔离级别时，事务中的一致性读的是以第一条select语句的运行时， 作为本事务的一致性读的建立时间点的。只能读取该时间点之前已经提交的数据
5. RC隔离级别下的update语句， 使用的是半一致性读(semi consistent)； 而RR隔离级别的update语句使用的是当前读；当前读会发生锁的阻塞

9-4-1-3：隔离级别用来做什么

隔离级别用于决定如何控制并发用户读写数据的操作

9-5：如何手动处理事务

回滚

10.引擎

10-1：MySQL存储引擎MyISAM与InnoDB区别

1、MyISAM是非事务安全的，而InnoDB是事务安全的 2、MyISAM锁的粒度是表级的，而 InnoDB支持行级锁 3、MyISAM支持全文类型索引，而InnoDB不支持全文索引 4、MyISAM相对简单，效率上要优于InnoDB，小型应用可以考虑使用MyISAM 5、MyISAM表保存成文件形式，跨平台使用更加方便 6、MyISAM管理非事务表，提供高速存储和检索以及全文搜索能力，如果在应用中执行大 量select 操作可选择 7、InnoDB用于事务处理，具有ACID事务支持等特性，如果在应用中执行大量 insert 和update操作，可选择。 8、InnoDB支持外键（从A表一个列（外键）去检索B表的主键） 9、MyISAM一般是非聚集索引，InnoDB是聚集索引

10-2：InnoDB引擎的4大特性

1. 插入缓冲（insert buffer)
2. 二次写(double write)
3. 自适应哈希索引(ahi)
4. 预读(read ahead) InnoDB使用两种预读算法来提高I/O性能：线性预读（linear read-ahead）和随机预读（randomread-ahead） 为了区分这两种预读的方式，我们可以把线性预读放到以extent为单位， 而随机预读放到以extent中的page为单位。线性预读着眼于将下一个extent提前读取到buffer pool中， 而随机预读着眼于将当前extent中的剩余的page提前读取到buffer pool中。

10-3：MyISAM和InnoDB存储引擎使用的锁

MyISAM采用表级锁(table-level locking)。 InnoDB支持行级锁(row-level locking)和表级锁,默认为行级锁

10-4：InnoDB存储引擎的锁的算法

Record lock：单个行记录上的锁 Gap lock：间隙锁，锁定一个范围，不包括记录本身 Next-key lock：record+gap 锁定一个范围，包含记录本身

11.锁

11-1：隔离级别与锁的关系

SQL使用锁来实现事务的隔离。 事务获取锁这种控制资源， 用于保护数据资源， 防止其他事务对数据进行冲突的或不兼容的访问。 比如说 读未提交，可以通过写操作加“持续-X锁”实现。 读已提交，可以通过写操作加“持续-X”锁，读操作加“临时-S锁”实现。 可重复读，可以通过写操作加“持续-X”锁，读操作加“持续-S锁”实现。

11-2：mysql锁的种类

1. 共享/排它锁(Shared and Exclusive Locks)
2. 意向锁(Intention Locks)
3. 记录锁(Record Locks)
4. 间隙锁(Gap Locks)
5. 临键锁(Next-key Locks)
6. 插入意向锁(Insert Intention Locks)
7. 自增锁(Auto-inc Locks)

11-2-1：共享/排它锁(Shared and Exclusive Locks)

共享锁（Share Locks，记为S锁），读取数据时加S锁 比如说执行查询某个表的所有信息时，首先锁定第一页，读取之后，释放对第一页的锁定， 然后锁定第二页。这样，就允许在读操作过程中，修改未被锁定的第一页。

排他锁（eXclusive Locks，记为X锁），修改数据时加X锁 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。

11-2-2：意向锁(Intention Locks)

未来的某个时刻，事务可能要加共享/排它锁了，先提前声明一个意向。

11-2-2-1：意向锁分类

意向共享锁(IS)，事务有意向对表中的某些行加共享S锁； 意向排它锁(IX)，事务有意向对表中的某些行加排它X锁；

```
select ... lock in share mode;    要设置IS锁;
select ... for update;            要设置IX锁;
```

11-2-3：记录锁(Record Locks)

记录锁，它封锁索引记录

11-2-4：间隙锁(Gap Locks)

它封锁索引记录中的间隔， 或者第一条索引记录之前的范围，又或者最后一条索引记录之后的范围。

比如说，某一个sql语句会封锁区间(8,15)，以阻止其他事务插入id位于该区间的记录。

间隙锁的主要目的，就是为了防止其他事务在间隔中插入数据， 以导致“不可重复读”。 如果把事务的隔离级别降级为读提交(Read Committed, RC)，间隙锁则会自动失效。

11-2-5：临键锁(Next-key Locks)

临键锁，是记录锁与间隙锁的组合，它的封锁范围，既包含索引记录，又包含索引区间。

比如说某个表中id降级为普通索引(key)， 也就是说即使这里声明了要加锁(for update)， 而且命中的是索引，但是因为索引在这里没有UK约束， 所以innodb会使用临键锁， 临键锁的主要目的， 也是为了避免幻读(Phantom Read)。 如果把事务的隔离级别降级为读已提交，临键锁则也会失效。

11-2-6：插入意向锁(Insert Intention Locks)

插入意向锁，是间隙锁(Gap Locks)的一种， 它是专门针对insert操作的。 多个事务，在同一个索引， 同一个范围区间插入记录时， 如果插入的位置不冲突，不会阻塞彼此。 比如说事务A先执行，在10与20两条记录中插入了一行，还未提交： 事务B后执行，也在10与20两条记录中插入了一行： 因为是插入操作，虽然是插入同一个区间， 但是插入的记录并不冲突， 所以使用的是插入意向锁，此处A事务并不会阻塞B事务。

11-2-7：自增锁(Auto-inc Locks)

专门针对事务插入AUTO_INCREMENT（自增列）类型的列。 比如说如果一个事务正在往表中插入记录， 所有其他事务的插入必须等待， 以便第一个事务插入的行，是连续的主键值。

11-3：行锁和表锁

11-3-1:表级锁

表级锁一次会将整个表锁定，所可以很好的避免死锁问题 锁定粒度大，锁冲突概率高、并发度低；

11-3-2：行级锁

锁定对象的颗粒度很小，发生锁冲突的概率低、并发度高； 缺点是开销大、加锁慢，行级锁容易发生死锁；

11-4：悲观锁与乐观锁

11-4-1：悲观锁

是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理） 修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。 读取数据时给加锁，其它事务无法修改这些数据。修改删除数据时也要加锁，其它事务无法读取这些数据。

11-4-2：乐观锁

乐观锁，大多是基于数据版本 （ Version ）记录机制实现。

11-4-2-1：数据版本

为数据增加一个版本标识，在基于数据库表的版本解决方案中， 一般是通过为数据库表增加一个 “version” 字段来实现。 读取出数据时，将此版本号一同读出，之后更新时， 对此版本号加一。

11-4-2-2：MVCC

代表多版本并发控制，读不加锁，读写不冲突

MVCC是通过在每行记录后面保存两个隐藏的列来实现的。 这两个列，一个保存了行的创建时间， 一个保存行的过期时间（或删除时间）。 当然存储的并不是实际的时间值， 而是系统版本号（system version number)。

每开始一个新的事务，系统版本号都会自动递增。 事务开始时刻的系统版本号会作为事务的版本号， 用来和查询到的每行记录的版本号进行比较。

11-5：数据库死锁的预防与解除

1. 应尽可能缩短事务。在同一DB中并发执行多个需要长时间运行的事务时， 发生死锁的概率较大。事务运行时间越长，其持有排它锁（exclusive锁）或更 新锁（update锁）的时间便越长， 从而堵塞了其它活动并可能导致死锁。 保持事务在一个批处理中， 可以最小化事务的网络通信往返量， 减少完成事务可能的延迟并释放锁。 同时，涉及多个表的查询更新操作， 若比较耗时，尽量不要放在一个事务内处理，能分割便分割。若不能分割，便尽可能使之在业务量较小的时间 (例如子夜或者午餐时间)执行。
2. 应按同一顺序访问数据对象。 如果所有并发事务按同一顺序访问对象， 则发生死锁的可能性会降低。例如，如果两个并发事务获得Supplier 表上的锁， 然后获得Part表上的锁， 则在其中一个事务完成之前， 另一个事务被阻塞在Supplier表上。 第一个事务提交或回滚后，第二个事务继续进行。 不发生死锁。 将存储过程用于所有的数据修改可以标准化访问对象的顺序。
3. 必须避免编写包含用户交互的事务。 因为运行没有用户交互的批处理的速度要 远远快于用户手动响应查询的速度， 若用户不能及时反馈，则此事务将挂起。 因而将严重降低系统的吞吐量， 因为事务持有的任何锁只有在事务提交或回滚时才会释放。 即使不出现死锁的情况， 访问同一资源的其它事务也会被阻塞，等待该事务完成。
4. 可使用低隔离级别。 确定事务是否能在更低的隔离级别上运行。 执行提交读允许事务读取另一个事务已读取（未修改）的数据， 而不必等待第一个事务完成。 使用较低的隔离级别（例如提交读） 而不使用较高的隔离级别（例如可串行读） 可以缩短持有共享锁的时间， 从而降低了锁定争夺。
5. 可考虑体系结构的优化与代码重构， 提高系统整体的运行效率。 例如尽可能不采用效率低下的计算模型， 复杂的业务应采用异步任务调度处理。
6. 可通过程序控制事务提交的时机。 如果一次检索出了10万条记录但只更改了其中的100条， 就可以通过代码来执行100个update。或是用分段提交， 即所有的修改使用多个事务进行提交， 但这样会使事务不完整，应酌情使用。
7. 宜将经常更新的数据库和查询数据库分开。 定期将不改变的数据导入查询数据库中， 这样查询和更新就可以分开进行，而降低死锁机率。
8. 在进行数据库模式设计时， 应注意外键引用的完整性，并对外键加索引。 如果更新了父表的主键，由于外键上没有索引， 所以子表会被锁定；如果删除了父表中的一行，整个子表也会被锁定。

12.索引

12-1：主键与索引的区别

主键是一种约束，唯一索引是一种索引，两者在本质上是不同的。

1、主键创建后一定包含一个唯一性索引，唯一性索引并不一定就是主键。 2、唯一性索引列允许空值，而主键列不允许为空值。 3、主键列在创建时，已经默认为空值 + 唯一索引了。 4、主键可以被其他表引用为外键，而唯一索引不能。 5、一个表最多只能创建一个主键，但可以创建多个唯一索引。 6、主键更适合那些不容易更改的唯一标识，如自动递增列、身份证号等。 7、在 RBO 模式下，主键的执行计划优先级要高于唯一索引。两者可以提高查询的速度。

12-2：索引的分类

1. 主键索引
2. 单值索引

3. 唯一索引
4. 联合（复合）索引

12-2-1：各种索引场景

12-1-2：联合索引

12-1-2-2：联合索引失效的条件

创建复合索引时，应该仔细考虑列的顺序。 对索引中的所有列执行搜索或仅对前几列执行搜索时， 复合索引非常有用； 仅对后面的任意列执行搜索时， 复合索引则没有用处。

12-1-2：单列索引和联合索引区别

1. 利用索引中的附加列，可以缩小搜索的范围， 创建复合索引时，应该仔细考虑列的顺序。 对索引中的所有列执行搜索或仅对前几列执行搜索时， 复合索引非常有用；仅对后面的任意列执行搜索时，复合索引则没有用处。 嗯，比如说一个例子吧， 人名由姓和名构成，假设我要去电话簿查找，首先按姓氏对进行排序， 然后按名字对有相同姓氏的人进行排序。如果知道姓，这样查找的就快迅速了； 但如果您只知道名不姓，电话簿将没有用处。
2. 多个单列索引在多条件查询时只会生效第一个索引 所以多条件联合查询时最好建联合索引

12-2：mysql索引的结构

1. B树索引与B+树索引
2. 聚簇索引与非聚簇索引
3. Hash索引
4. 全文索引
5. 空间索引

12-2-1：B+树与其他

12-2-1-1：B+树比B树的优势

1. B+树空间利用率更高，可减少I/O次数 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1O消耗。 而因为B+树的内部节点只是作为索引使用， 而不像B-树那样每个节点都需要存储硬盘指针。 也就是说: B+树中每个非叶子节点 没有指向某个关健字具体信息的指针， 所以好个节点可以存放更多的关键字数量， 减少了I/0操作。
2. 增删文件(节点)时，效率更高 因为B+树的叶子节点包含所有关键字， 并以有序的链表结构存储， 这样可很好提高增删效率， 基于范围查询更好。
3. B+树的查询效率更加稳定 因为B+树的每次查询过程中， 都需要遍历从根节点到叶子节点的某条路径。所有关键字的查询路径长度相同， 导致每一次查询的效率相当。

12-2-1-2：B+树与红黑树比较

1. 更少的查找次数 复杂度和树高h相关，红黑树的树高h很明显比B+Tee大非常多， 查找的次数也就更多。
2. 利用磁盘预读特性 为了减少磁盘IO操作，磁盘往往不是严格按需读取， 而是每次都会预读。预读过程中，磁盘进行顺序读取， 顺序读取不需要进行磁盘寻道， 并且只需要很短的旋转时间，速度会非常快。

12-2-1-3：B+树与hash索引比较

1. 如果是等值查询，那么哈希索引明显有绝对优势， 因为只需要经过一次算法即可找到相应的键值。 这个前提是，键值都是唯一的。 如果键值不是唯的，就需要先找到该键所在位置， 然后再根据链表往后扫描，直到找到相应的数据:
2. 如果是范围查询检索，原先是有序的键值， 经过哈希算法后，有可能变成不连续的了， 就没办法再利用索引完成范围查询检索:

12-2-2：聚簇索引与非聚簇索引概念

聚集索引即索引结构和数据一起存放的索引。主键索引属于聚集索引。 非聚集索引即索引结构和数据分开存放的索引。

12-2-2-1：聚簇索引的优缺点

一、优点 由于数据都是紧密相连，数据库不用从多个数据块中提取数据， 所以节省了大量的io操作。 二、缺点

1. 对于mysql数据库目前只有innodb数据引擎支持聚簇索引， 而Myisam并不支持聚簇索引。
2. 由于数据物理存储排序方式只能有一种， 所以每个Mysql的表只能有一个聚簇索引。 一般情况下就是该表的主键。
3. 为了充分利用聚簇索引的聚簇的特性， 所以innodb表的主键列尽量选用有序的顺序id， 而不建议用无序的id，比如uuid这种。

12-2-2-2：非聚簇索引的优缺点

一、优点 更新代价比聚集索引要小，非聚集索引的叶子节点是不存放数据的 二、缺点 非聚集索引也依赖于有序的数据 可能会二次查询(回表) 当查到索引对应的指针或主键后， 可能还需要根据指针或主键再到数据文件或表中查询。

12-3：索引

索引（Index）是帮助 MySQL 高效获取数据的数据结构，是一种排好序的数据结构

12-3-1：为什么要用索引（优点）

1. 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
2. 可以大大加快 数据的检索速度（大大减少的检索的数据量），这也是创建索引的最主要的原因。
3. 帮助服务器避免排序和临时表。
4. 将随机IO变为顺序IO
5. 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

12-3-2：索引这么多优点，为什么不对表中的每一个列创建一个索引呢？（缺点）

1. 当对表中的数据进行增加、删除和修改的时候， 索引也要动态的维护，这样就降低了数据的维护速度。
2. 索引需要占物理空间，除了数据表占数据空间之外， 每一个索引还要占一定的物理空间， 如果要建立聚簇索引，那么需要的空间就会更大。
3. 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

12-3-3：创建索引原则（使用场景）：

1. 选择唯一性索引 唯一性索引的值是唯一的， 可以更快速的通过该索引来确定某条记录。 例如，学生表中学号是具有唯一性的字段。 为该字段建立唯一性索引可以很快的确定某个学生的信息。 如果使用姓名的话，可能存在同名现象，从而降低查询速度。
2. 为经常需要排序、分组和联合操作的字段建立索引 经常需要ORDER BY、GROUP BY、DISTINCT和UNION等操作的字段， 排序操作会浪费很多时间。如果为其建立索引，可以有效地避免排序操作。
3. 为常作为查询条件的字段建立索引 如果某个字段经常用来做查询条件， 那么该字段的查询速度会影响整个表的查询速度。 因此，为这样的字段建立索引，可以提高整个表的查询速度。
4. 限制索引的数目 索引的数目不是越多越好。 每个索引都需要占用磁盘空间，索引越多， 需要的磁盘空间就越大。 修改表时，对索引的重构和更新很麻烦。 越多的索引，会使更新表变得很浪费时间。
5. 尽量使用数据量少的索引 如果索引的值很长，那么查询的速度会受到影响。 例如，对一个CHAR(100)类型的字段 进行全文检索需要的时间肯定要 比对CHAR(10)类型的字段需要的时间要多。
6. 尽量使用前缀来索引 如果索引字段的值很长，最好使用值的前缀来索引。 例如，TEXT和BLOG类型的字段， 进行全文检索会很浪费时间。 如果只检索字段的前面的若干个字符，这样可以提高检索速度。
7. 删除不再使用或者很少使用的索引 表中的数据被大量更新，或者数据的使用方式被改变后， 原有的一些索引可能不再需要。 数据库管理员应当定期找出这些索引， 将它们删除，从而减少索引对更新操作的影响。
8. 最左前缀匹配原则。 mysql会一直向右匹配 直到遇到范围查询(>、<、between、like)就停止匹配， 比如a 1=”” and=”” b=”2” c=””> 3 and d = 4 如果建立(a,b,c,d)顺序的索引， d是用不到索引的， 如果建立(a,b,d,c)的索引则都可以用到， a,b,d的顺序可以任意调整。
9. =和in可以乱序。 比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序， mysql的查询优化器会帮你优化成索引可以识别的形式
10. 尽量选择区分度高的列作为索引。 区分度的公式是count(distinct col)/count(*)， 表示字段不重复的比例， 比例越大我们扫描的记录数越少，
11. 索引列不能参与计算，保持列“干净”。 b+树中存的都是数据表中的字段值， 但进行检索时，需要把所有元素都应用函数才能比较， 显然成本太大。
12. 尽量的扩展索引，不要新建索引。 比如表中已经有a的索引， 现在要加(a,b)的索引，那么只需要修改原来的索引即可 因为选择索引的最终目的是为了使查询的速度变快。

12-3-3：最左前缀原则内部原理

修改

12-3-4：创建索引的注意事项

1. 限制表上的索引数目。
2. 避免在取值朝一个方向增长的字段（例如：日期类型的字段）上， 建立索引；对复合索引，避免将这种类型的字段放置在最前面。 由于字段的取值总是朝一个方向增长， 新记录总是存放在索引的最后一个叶页中， 从而不断地引起该叶页的访问竞争、新叶页的分配、中间分支页的拆分。 此外，如果所建索引是聚集索引，表中数据按照索引的排列顺序存放， 所有的插入操作都集中在最后一个数据页上进行，从而引起插入“热点”。
3. 对复合索引，按照字段在查询条件中出现的频度建立索引。 在复合索引中，记录首先按照第一个字段排序。 对于在第一个字段上取值相同的记录， 系统再按照第二个字段的取值排序，以此类推。 因此只有复合索引的第一个字段出现在查询条件中， 该索引才可能被使用。因此将应用频度高的字段， 放置在复合索引的前面， 会使系统最大可能地使用此索引，发挥索引的作用。
4. 删除不再使用，或者很少被使用的索引。表中的数据被大量更新， 或者数据的使用方式被改变后，原有的一些索引可能不再被需要。 数据库管理员应当定期找出这些索引，将它们删除， 从而减少索引对更新操作的影响。

12-3-5：索引失效

1. like 以%开头，索引无效；当like前缀没有%，后缀有%时，索引有效。 主要是因为MYSQL索引失效会变成全表扫描的操作
2. or语句前后没有同时使用索引。 当or左右查询字段只有一个是索引， 该索引失效，只有当or左右查询字段均为索引时，才会生效
3. 组合索引，不是使用第一列索引，索引失效。
4. 数据类型出现隐式转化。 如varchar不加单引号的话可能会自动转换为int型， 使索引无效，产生全表扫描。
5. 在索引列上使用IS NULL 或IS NOT NULL操作。 索引是不索引空值的，所以这样的操作不能使用索引，可以用其他的办法处理， 例如：数字类型，判断大于0，字符串类型设置一个默认值， 判断是否等于默认值即可。
6. 在索引字段上使用not，<>，!=。 不等于操作符是永远不会用到索引的， 因此对它的处理只会产生全表扫描。 优化方法： key<>0 改为 key>0 or key<0。
7. 对索引字段进行计算操作、字段上使用函数。 （索引为 emp(ename,empno,sal)）
8. 当全表扫描速度比索引速度快时，mysql会使用全表扫描，此时索引失效。

12-3-7：为什么索引能够提高查询速度

修改

12-3-8：创建索引的三种方式

```
#方法一：创建表时
        CREATE TABLE 表名 (
                字段名1  数据类型 [完整性约束条件…],
                字段名2  数据类型 [完整性约束条件…],
                [UNIQUE | FULLTEXT | SPATIAL ]   INDEX | KEY
                [索引名]  (字段名[(长度)]  [ASC |DESC])
                );

       CREATE TABLE t(
                        c1 INT PRIMARY KEY,
                        c2 INT NOT NULL,
                        c3 INT NOT NULL,
                        c4 VARCHAR(10),
                        INDEX (c2,c3)
                      );

#方法二：CREATE在已存在的表上创建索引
        CREATE  [UNIQUE | FULLTEXT | SPATIAL ]  INDEX  索引名
                     ON 表名 (字段名[(长度)]  [ASC |DESC]) ;

        CREATE INDEX index_name ON table_name (column_list)

#方法三：ALTER TABLE在已存在的表上创建索引
        ALTER TABLE 表名 ADD  [UNIQUE | FULLTEXT | SPATIAL ] INDEX
                             索引名 (字段名[(长度)]  [ASC |DESC]) ;

         alter table table_name add index index_name (column_list) ;
```

```
#删除索引：DROP INDEX 索引名 ON 表名字;
```

12-3-9：最左前缀原则

12-3-10：最左匹配原则

13.优化方案

13-1：explain

通过expalin就能知道MySQL是如何处理你的SQL语句的。 分析你的查询语句或是表结构的性能瓶颈

13-1-1：explain应用场景

1. 表的读取顺序
2. 哪些索引可以使用
3. 数据读取操作的操作类型
4. 哪些索引被实际使用
5. 表之间的引用
6. 每张表有多少行被优化器查询

13-1-2：如何使用explain

explain关键字+sql语句

13-1-3：explain主要包含的信息

13-1-3-0：主要包含信息

id select_type table partitions type possible_keys key key_len ref rows filtered extra

21-4-1：id

表示查询中执行 select子句 或 操作表 的顺序 id越大，优先级越高 一般的话，就得看sql的执行顺序

1. 情形一

```
```sql
EXPLAIN
SELECT *
FROM `departments`,`employees`
WHERE `departments`.`department_id`=`employees`.`department_id`
AND `departments`.`manager_id`=`employees`.`manager_id`;
```
最后的结果两个表是一起执行
```sql
id   select_type      table
```
```

```
1       SIMPLE      departments
1       SIMPLE      employees
```

2. 情形二

```sql
EXPLAIN
SELECT *
FROM `departments`
WHERE `location_id`<(
         SELECT `location_id`
         FROM `employees`
         WHERE `department_id`=90

);
```

由于存在了子查询，那么必须查到`employees`才可以

```sql
id      select_type         table
1          PRIMARY          departments
2          SUBQUERY         employees
```

21-4-2：select_type

13-2：为什么要优化

1. 避免网站页面出现访问错误

   由于慢查询造成页面无法加载 由于阻塞造成数据无法提交 增加数据库的稳定性

2. 很多数据库问题都是由于低效的查询引起的

13-3：数据库的优化（如果mysql数据过多，如何进行处理）

1. 限定数据的范围 务必禁止不带任何限制数据范围条件的查询语句。 比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内；
2. 读/写分离 经典的数据库拆分方案，主库负责写，从库负责读；
3. 垂直分区 根据数据库里面数据表的相关性进行拆分。
4. 水平分区 保持数据表结构不变，通过某种策略存储数据分片。 这样每一片数据分散到不同的表或者库中， 达到了分布式的目的。 水平拆分可以支撑非常大的数据量。

注意：垂直分区与水平分区*去看分库分表*

13-4：索引优化

1、建立聚集索引 首先聚合索引是提升查询速度的最有效的手段。 基于聚合索引的性质可以知道 数据库的物理存储顺序是按照聚合索引顺序排列的， 而通过聚合索引的B+树，我们可以迅速 的查找到任何一行的全部信息。 2、常查询数据建立索引或者组合索引 3、最左前缀原则 建立组合索引优化查询语句时，一定要考虑到最左前缀原则，否则你的索引建立的可以说毫无意义 4、较长的数据列建立前缀索引; 5、不要建立无意义的索引 对于查询次数很少的语句中的字段的索引、备注描述和大字段的索引等

13-5：查询优化

1、使用 Explain进行分析 Explain用来分析SELECT 查询语句，开发人员可以通过分析Explain结果者傥化参驾调句。 比较重要的字段有: select_type:查询类型，有简单查询、联合查询、子查询等 key:使用的索引 rows:扫描的行数 2、优化数据访问 （1）减少请求的数据量 只返回必要的列:最好不要使用SELECT*语句。 只返回必要的行:使用 LIMIT 语句来限制返回的数据。 缓存重复查询的数据:使用缓存可以避免在数据库中进行查询，特别在要查询的数据经常被 重复查询时，缓存带来的查询性能提升将会是非常明显的。 （2）减少服务器端扫描的行数 最有效的方式是使用索引来覆盖查询。 3、重构查询方式 （1）切分大查询 一个大查询如果一次性执行的话，可能一次锁住很多数据、耗尽系统资源、阻塞很多小的但重要的查询。 DELETE FROM messages WHERE create <DATE sLIB(NOWoINTERVAL 3 MONTH); rows_affected= O do { rows_affected= do_query("DELETEFROM messages WHEREcreate<DATE SUB(NOWO.INTERVAL 3 MONTH) LIMIT10000") }while rows_affected>0 (2）分解大连接查询 将一个大连接查询分解成对每一个表进行一次单表查询，然后在应用程序中进行关联，这样 做的好处有: 让缓存更高效。对于连接查询，如果其中一个表发生变化，那么整个查询缓存就无法使用。 而分解后的多个查询，即使其中一个表发生变化，对其它表的查询缓存依然可以使用。 分解成多个单表查询，这些单表查询的缓存结果更可能被其它查询使用到，从而减少冗余记 录的查询。

13-6：慢查询优化

1. 索引没起作用的情况 1. 使用LIKE关键字的查询语句 在使用LIKE关键字进行查询的查询语句中， 如果匹配字符串的第一个字符为“%”， 索引不会起作用。只有“%”不在第一个位置索引才会起作用。 2. 使用多列索引的查询语句 MySQL可以为多个字段创建索引。 一个索引最多可以包括16个字段。 对于多列索引，只有查询条件使用了 这些字段中的第一个字段时 索引才会被使用。
2. 优化数据库结构 合理的数据库结构不仅可以 使数据库占用更小的磁盘空间， 而且能够使查询速度更快。 数据库结构的设计， 需要考虑数据冗余、 查询和更新的速度、 字段的数据类型是否合理等多方面的内容。
    1. 将字段很多的表分解成多个表 对于字段比较多的表，如果有些字段的使用频率很低， 可以将这些字段分离出来形成新表。 因为当一个表的数据量很大时， 会由于使用频率低的字段的存在而变慢。
    2. 增加中间表 对于需要经常联合查询的表， 可以建立中间表以提高查询效率。 通过建立中间表，把需要经常联合 查询的数据插入到中间表中， 然后将原来的联合查询 改为对中间表的查询， 以此来提高查询效率。
3. 分解关联查询 将一个大的查询分解为多个小查询是很有必要的。 很多高性能的应用都会对关联查询进行分解， 就是可以对每一个表进行一次单表查询， 然后将查询结果在应用程序中进行关联， 很多场景下这样会更高效，
4. 优化LIMIT分页

13-7：一条SQL语句执行得很慢的原因有哪些

要分两种情形：

1. 大多数情况是正常的，只是偶尔会出现很慢的情况。

- 数据库在刷新脏页，例如 redo log 写满了需要同步到磁盘。

  当我们要往数据库插入一条数据、或者要更新一条数据的时候， 我们知道数据库会在内存中把对应字段的数据更新了，但是更新之后， 这些更新的字段并不会马上同步持久化到磁盘中去， 而是把这些更新的记录写入到 redo log 日记中去，等到空闲的时候， 在通过 redo log 里的日记把最新的数据同步到磁盘中去。

  当内存数据页跟磁盘数据页内容不一致的时候，我们称这个内存页为“脏页”。 比如说：

  1. redolog写满了： redo log 里的容量是有限的， 如果数据库一直很忙，更新又很频繁，这个时候 redo log 很快就会被写满了， 这个时候就没办法等到空闲的时候再把数据同步到磁盘的，只能暂停其他操作， 全身心来把数据同步到磁盘中去的，而这个时候， 就会导致我们平时正常的SQL语句突然执行的很慢， 所以说，数据库在在同步数据到磁盘的时候，就有可能导致我们的SQL语句执行的很慢了。

  2. 内存不够用了： 如果一次查询较多的数据，恰好碰到所查数据页不在内存中时， 需要申请内存，而此时恰好内存不足的时候就需要淘汰一部分内存数据页， 如果是干净页，就直接释放，如果恰好是脏页就需要刷脏页。

  3. MySQL 认为系统“空闲”的时候：这时系统没什么压力。

  4. MySQL 正常关闭的时候：这时候，MySQL 会把内存的脏页都 flush 到磁盘上， 这样下次 MySQL 启动的时候，就可以直接从磁盘上读数据，启动速度会很快。

- 执行的时候，遇到锁，如表锁、行锁。

2. 在数据量不变的情况下，这条SQL语句一直以来都执行的很慢。

   - 没有用上索引
   - 数据库选错了索引

13-7-1：为什么数据库会选错了索引

系统在执行的时候，会进行预测， 是走 c 索引扫描的行数少， 还是直接扫描全表扫描的行数少 扫描全表的话，那么扫描的次数就是这个表的总行数了， 假设为 n；而如果走索引c的话， 我们通过索引c找到主键之后，还得再通过主键索引来找我们整行的数据， 需要走两次索引， 而且，我们也不知道符合 这个条件的数据有多少行， 万一真的是n条， 那就惨了， 所以系统是有可能走全表扫描而不走索引的 系统如何进行预判 主要依赖于索引的区分度来判断的， 一个索引上不同的值越多， 意味着出现相同数值的索引越少， 意味着索引的区分度越高。 这个区分度也叫做基数， 系统当然是不会遍历全部来 获得一个索引的基数的， 代价太大了， 索引系统是通过遍历部分数据， 也就是通过采样的方式， 来预测索引的基数的 那么出现失误的地方就是采样， 比如采样的那一部分数据刚好基数很小， 然后就误以为索引的基数很小。 然后，系统就不走索引了， 直接走全部扫描了。 主要是由于统计的失误， 导致系统没有走索引， 而是走了全表扫描。

14.慢查询

它用来记录在MySQL中响应时间超过阀值的语句日志记录

15.主从复制

15-1：什么是主从复制

主从复制，是用来建立一个和主数据库完全一样的数据库环境， 称为从数据库；主数据库一般是准实时的业务数据库。

15-2：主从复制的作用（好处，或者说为什么要做主从）：

1、做数据的热备，作为后备数据库，主数据库服务器故障后， 可切换到从数据库继续工作，避免数据丢失。 2、架构的扩展。业务量越来越大，I/O访问频率过高， 单机无法满足，此时做多库的存储，降低磁盘I/O访问的频率， 提高单个机器的I/O性能。 3、读写分离，使数据库能支撑更大的并发。 在报表中尤其重要。由于部分报表sql语句非常的慢， 导致锁表，影响前台服务。如果前台使用master， 报表使用slave， 那么报表sql将不会造成前台锁，保证了前台速度。

15-3：主从复制的原理（重中之重，面试必问）

步骤一：主库db的更新事件(update、insert、delete)被写到binlog 步骤二：从库发起连接，连接到主库 步骤三：此时主库创建一个binlog dump thread，把binlog的内容发送到从库 步骤四：从库启动之后，创建一个I/O线程， 读取主库传过来的binlog内容并写入到relay log 具体需要三个线程来操作： binlog输出线程。每当有从库连接到主库的时候， 主库都会创建一个线程然后发送binlog内容到从库。 在从库里，当复制开始的时候，从库就会创建两个线程进行处理： 从库I/O线程。当START SLAVE语句在从库开始执行之后， 从库创建一个I/O线程， 该线程连接到主库并请求主库 发送binlog里面的更新记录到从库上。 从库I/O线程读取主库的binlog输出线程 发送的更新并拷贝这些更新到本地文件， 其中包括relay log文件。 从库的SQL线程。 从库创建一个SQL线程， 这个线程读取从库I/O线程 写到relay log的更新事件并执行。

15-4：主从复制的几种方式

1. 同步复制，意思是master的变化， 必须等待slave-1,slave-2,...,slave-n完成后才能返回。 不会使用，比如，在WEB前端页面上，用户增加了条记录， 需要等待很长时间。
2. 异步复制:如同AJAX请求一样。 master只需要完成自己的数据库操作即可。 至于slaves是否收到二进制日志， 是否完成操作，不用关心,MySQL的默认设置。
3. 半同步复制:master只保证slaves中的一个操作成功， 就返回，其他slave不管。 这个功能，是由google为MySQL引入的。

16.阻塞

当多个事务都需要对某一资源进行锁定时， 默认情况下会发生阻塞。 被阻塞的请求会一直等待， 直到原来的事务释放相关的锁。 锁定超时期限可以限制，这样就可以限制被 阻塞的请求在超时之前要等待的时间。 比如说 有两个事务 事务A请求资源S1， 事务不对资源S1进行操作 事务A用锁A锁定资源S1， 事务B请求对资源S1进行不兼容的锁定（锁B），锁B的请求被阻塞，事务B将进入等待状态 事务A正在释放锁A，事务B等待锁A释放，事务A的锁A已释放，事务B用锁B锁定资源S1

17.数据库池与JDBC

17-1：什么是数据库连接池?

因为吧，创建数据库连接是一个很耗时的操作， 也容易对数据库造成安全隐患。 所以，在程序初始化的时候，集中创建多个数据库连接，并把他们集中管理， 供程序使用， 可以保证较快的数据库读写速度，还更加安全可

靠。

17-1-1：数据库连接池种类

1, C3P0 C3P0是一个开放源代码的JDBC连接池， 它在lib目录中与Hibernate[1]一起发布, 包括了实现jdbc3和jdbc2扩展规范 说明的Connection 和Statement 池的DataSources 对象。 2,Druid，但它不仅仅是一个数据库连接池， 它还包含一个ProxyDriver， 一系列内置的JDBC组件库，一个SQL Parser。

17-1-2：传统的连接机制与数据库连接池的运行机制区别

传统统链接: 一般来说，Java应用程序访问数据库的过程是： ①装载数据库驱动程序； ②通过JDBC建立数据库连接； ③访问数据库，执行SQL语句； ④断开数据库连接。 使用了数据库连接池的机制：（1） 程序初始化时创建连接池 （2） 使用时向连接池申请可用连接 （3） 使用完毕，将连接返还给连接池（4） 程序退出时，断开所有连接，并释放资源

17-1-3：说说数据库连接池工作原理

1. 连接池的建立。 一般在系统初始化时，连接池会根据系统配置建立， 并在池中建立几个连接对象，以便使用时能从连接池中获取。 java 中提供了很多容器类，可以方便的构建连接池， 例如 Vector（线程安全类），linkedlist 等。
2. 连接池的管理。 连接池管理策略是连接池机制的核心， 连接池内连接的分配和释放对系统的性能有很大的影响。 主要就是当客户请求数据库连接时，首先查看连接池中是否有空闲连接， 如果存在空闲连接，则将连接分配给客户使用并作相应处理 也就是标记该连接为正在使用，引用计数加 1； 如果没有空闲连接，则查看当前所开的连接数是否已经达到最大连接数， 如果没有达到最大连接数，就重新创建一个连接给请求的客户； 如果达到，就按设定的最大等待时间进行等待， 如果超出最大等待时间，则抛出异常给客户。 当客户释放数据库连接时，先判断该连接的引用次数是否超过了规定值， 如果超过了就从连接池中删除该连接， 并判断当前连接池内总的连接数是否小于最小连接数， 若小于就将连接池充满；如果没超过就将该连接标记为开放状态， 可供再次复用。
3. 连接池的关闭。 当应用程序退出时，关闭连接池中所有的链接， 释放连接池相关资源，该过程正好与创建相反。

17-1-4：实现方案?

连接池使用集合来进行装载， 返回的Connection是原始Connection的代理， 代理Connection的close方法， 当调用close方法时，不是真正关连接，而是把它代 理的Connection对象放回到连接池中， 等待下一次重复利用。

17-1-5：数据库连接池最小连接数和最大连接

连接池最小连接数 最小连接数是连接池一直保持的数据连接。 如果应用程序对数据库连接的使用量不大，将会有大量的数据库连接资源被浪费掉。

连接池最大连接数 最大连接数是连接池能申请的最大连接数。如果数据连接请求超过此数，后面的数据连接请求将被加入到等待队列中，这会影响之后的数据库操作。

17-1-5-1：最小连接数和最大连接区别

17-1-5-2：最小连接数和最大连接数的设置要考虑到以下几个因素

1. 最小连接数:是连接池一直保持的数据库连接, 所以如果应用程序对数据库连接的使用量不大, 将会有大量的数据库连接资源被浪费.
2. 最大连接数:是连接池能申请的最大连接数, 如果数据库连接请求超过次数, 后面的数据库连接请求将被加入到等待队列中, 这会影响以后的数据库操作

如果最小连接数与最大连接数相差太大， 那么，最先的连接请求将会获利， 之后超过最小连接数量的连接请求等价于建立一个新的数据库连接。 不过，这些大于最小连接数的数据库连接 在使用完不会马上被释放， 它将被放到连接池中等待重复使用 或是空闲超时后被释放。

17-2：JDBC

17-2-1：JDBC数据库连接步骤

1. 注册数据库驱动
2. 建立数据库连接
3. 创建一个Statement
4. 执行SQL语句
5. 处理结果集
6. 关闭数据库连接

17-2-2：JDBC原理

JDBC（Java DataBase Connectivity）就是Java数据库连接，就是用Java语言来操作数据库 由SUN提供一套访问数据库的规范（就是一组接口）， 并提供连接数据库的协议标准， 然后各个数据库厂商会遵循SUN的规范 提供一套访问自己公司的数据库服务器的API出现。 SUN提供的规范命名为JDBC， 而各个厂商提供的，遵循了JDBC规范的， 可以访问自己数据库的API被称之为驱动!

17-2-3：JDBC编程原理

```
public class TestDB {
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;
        // MySQL的JDBC连接语句
        // URL编写格式：jdbc:mysql://主机名称：连接端口/数据库的名称?参数=值
        String url = "jdbc:mysql://localhost:3306/student?
user=root&password=123456";
        // 数据库执行的语句
        String sql = "insert into stuinfo
values('201307020010','zhangsan',21);";//插入一条记录
        //String sql = "create table stuinfo(id char(12),name char(20),age
int);";//创建一个表
        // 查询语句
        String cmd = "select * from stuinfo;";
        try {
            Class.forName("com.mysql.jdbc.Driver"); // 加载驱动
            conn = DriverManager.getConnection(url); // 获取数据库连接
            stmt = conn.createStatement(); // 创建执行环境
            stmt.execute(sql); // 执行SQL语句
```

```
            // 读取数据
            rs = stmt.executeQuery(cmd); // 执行查询语句，返回结果数据集
            rs.last(); // 将光标移到结果数据集的最后一行，用来下面查询共有多少行记录
            System.out.println("共有" + rs.getRow() + "行记录：");
            rs.beforeFirst(); // 将光标移到结果数据集的开头
            while (rs.next()) { // 循环读取结果数据集中的所有记录
                System.out.println(rs.getRow() + "、 学号:" + rs.getString("id")
                        + "\t姓名:" + rs.getString("name") + "\t年龄:"
                        + rs.getInt("age"));
            }
        } catch (ClassNotFoundException e) {
            System.out.println("加载驱动异常");
            e.printStackTrace();
        } catch (SQLException e) {
            System.out.println("数据库异常");
            e.printStackTrace();
        } finally {
            try {
                if (rs != null)
                    rs.close(); // 关闭结果数据集
                if (stmt != null)
                    stmt.close(); // 关闭执行环境
                if (conn != null)
                    conn.close(); // 关闭数据库连接
            } catch (SQLException e) {
                e.printStackTrace();}}}}
```

17-2-3-1：使用JDBC需要用到哪些类

DirverManager类：是JDBC的管理层， 作用bai于用户和驱动之间。 该类负责注du册和加载JDBC驱动。 Connection接口：代表与数据库dao的链接， 并拥有创建SQL语句的方法， 以完成基本的SQL操作， 同时为数据库事务提供提交和回滚方法。 Statement接口：用于执行不带参数的简单SQL语句。 创建Statement实例对象后可以 调用JDBC提供的3种执行SQL语句的方法：

17-2-3-1：JDBC的DriverManager是用来做什么的?

JDBC的DriverManager是一个工厂类，我们通过它来创建数据库连接。 当JDBC的Driver类被加载进来时，它会自己注册到DriverManager类里面 然后我们会把数据库配置信息传成DriverManager.getConnection()方法，DriverManager会使用注册到它里面的驱动来获取数据库连接，并返回给调用的程序。

17-2-3-2：statement的实现原理

Statement是用来向数据库发送要执行的SQL语句的

17-2-3-2-1：JDBC中的Statement 和PreparedStatement的区别?

PreparedStatement是预编译的SQL语句，效率高于Statement。 PreparedStatement支持?操作符，相对于Statement更加灵活。 PreparedStatement可以防止SQL注入，安全性高于Statement。 CallableStatement适用于执行存储过程。

17-2-3-2-2：PreparedStatement的缺点是什么，怎么解决这个问题?

PreparedStatement的一个缺点是，我们不能直接用它来执行in条件语句； 需要执行IN条件语句的话，下面有一些解决方案： 分别进行单条查询——这样做性能很差，不推荐。 使用存储过程——这取决于数据库的实现，不是所有数据库都支持。 动态生成PreparedStatement——这是个好办法， 但是不能享受PreparedStatement的缓存带来的好处了。 在PreparedStatement查询中使用NULL值—— 如果你知道输入变量的最大个数的话，这是个不错的办法， 扩展一下还可以支持无限参数。

17-2-3-3：execute，executeQuery，executeUpdate的区别是什么?

`Statement的execute`(String query)方法用来执行任意的SQL查询， 如果查询的结果是一个ResultSet，这个方法就返回true。 如果结果不是ResultSet，比如insert或者 update查询，它就会返回false。 我们可以通过它的getResultSet方法来获取ResultSet， 或者通过getUpdateCount()方法来获取更新的记录条数。 `Statement的executeQuery`(String query)接口用来执行select查询， 并且返回ResultSet。即使查询不到记录返回的ResultSet也不会为null。 我们通常使用executeQuery来执行查询语句， 这样的话如果传进来的是insert或者update语句的话， 它会抛出错误 `Statement的executeUpdate`(String query)方法用来执行 insert或者update/delete（DML）语句， 或者什么也不返回DDL语句。返回值是int类型， 如果是DML语句的话，它就是更新的条数， 如果是DDL的话，就返回0。 只有当你不确定是什么语句的时候才应该使用execute()方法， 否则应该使用executeQuery或者executeUpdate方法。

17-2-3-4：JDBC的ResultSet是什么?

在查询数据库后会返回一个ResultSet， 它就像是查询结果集的一张数据表。 ResultSet对象维护了一个游标，指向当前的数据行。 开始的时候这个游标指向的是第一行。 如果调用了ResultSet的next()方法游标会下移一行， 如果没有更多的数据了，next()方法会返回false。 可以在for循环中用它来遍历数据集。 默认的ResultSet是不能更新的， 游标也只能往下移。 也就是说你只能从第一行到最后一行遍历一遍。 不过也可以创建可以回滚或者可更新的ResultSet 当生成ResultSet的Statement对象 要关闭或者重新执行或是获取下一个ResultSet的时候， ResultSet对象也会自动关闭。 可以通过ResultSet的getter方法， 传入列名或者从1开始的序号来获取列数据。

17-2-3-4-1：有哪些不同的ResultSet?

一共有三种ResultSet对象。

1. ResultSet.TYPE_FORWARD_ONLY： 这是默认的类型，它的游标只能往下移。
2. ResultSet.TYPE_SCROLL_INSENSITIVE： 游标可以上下移动，一旦它创建后， 数据库里的数据再发生修改，对它来说是透明的。
3. ResultSet.TYPE_SCROLL_SENSITIVE： 游标可以上下移动， 如果生成后数据库还发生了修改操作， 它是能够感知到的。

17-2-3-4-2：JDBC的RowSet是什么，

RowSet用于存储查询的数据结果，和ResultSet相比，它更具灵活性。 RowSet继承自ResultSet，因此ResultSet能干的，它们也能， 而ResultSet做不到的，它们还是可以。RowSet接口定义在javax.sql包里。

17-2-3-4-3：有哪些不同的RowSet?

A. 连接型RowSet——这类对象与数据库进行连接， 和ResultSet很类似。 JDBC接口只提供了一种连接型RowSet， B. 离线型RowSet——这类对象不需要和数据库进行连接， 因此它们更轻量级，更容易序列化。 它们适用于在网络间传递数据。 有四种不同的离线型RowSet的实现。 CachedRowSet——可以通过他们获取连接，执行查询并读取ResultSet的数据到RowSet里。 我们可以在离线时对数据进行维护和更新， 然后重新连接到数据库里，并回写改动的数据。 WebRowSet继承自CachedRowSet——他可以读写XML文档。 JoinRowSet继承自WebRowSet——它不用连接数据库就可以执行SQL的join操作。 FilteredRowSet继承自WebRowSet——我们可以用它来设置过滤规则，这样只有选中的数据才可见。

17-2-3-4-2：ResultSet有两种并发类型。

1. ResultSet.CONCUR_READ_ONLY: ResultSet是只读的，这是默认类型。
2. ResultSet.CONCUR_UPDATABLE: 我们可以使用ResultSet的更新方法来更新里面的数据。

17-2-3-5：JDBC的DataSource是什么，有什么好处

跟DriverManager相比，它的功能要更强大。 我们可以用它来创建数据库连接， 当然驱动的实现类会实际去完成这个工作。 除了能创建连接外，还能够提供

1. 缓存PreparedStatement以便更快的执行
2. 可以设置连接超时时间
3. 提供日志记录的功能
4. ResultSet大小的最大阈值设置
5. 通过JNDI的支持，可以为servlet容器提供连接池的功能

17-2-3-5-1：如何通过JDBC的DataSource和Apache Tomcat的JNDI来创建连接池?

在META-INF目录下配置context.xml文件 导入Mysql或oracle开发包到tomcat的lib目录下 初始化JNDI->获取JNDI容器->检索以XXX为名字在JNDI容器存放的连接池

17-2-3-6：java.util.Date和java.sql.Date有什么区别?

java.util.Date包含日期和时间， java.sql.Date只包含日期信息，而没有具体的时间信息。 如果你想把时间信息存储在数据库里，可以考虑使用Timestamp或者DateTime字段

17-2-3-7：SQLWarning是什么，在程序中如何获取SQLWarning?

SQLWarning是SQLException的子类， 通过Connection, Statement, Result的getWarnings 方法都可以获取到它。 SQLWarning不会中断查询语句的执行， 只是用来提示用户存在相关的警告信息。

17-2-3-8：如果java.sql.SQLException: No suitable driver found该怎么办?

如果你的SQL URL串格式不正确的话，就会抛出这样的异常。 不管是使用DriverManager还是JNDI数据源来创建连接都有可能抛出这种异常。

17-2-4：JDBC事务

17-2-4-1：Java中如何进行事务的处理?

Connection类中提供了4个事务处理方法:

setAutoCommit(Boolean autoCommit):设置是否自动提交事务,默认为自动提交,即为true, 通过设置false禁止自动提交事务; commit():提交事务; rollback():回滚事务. savepoint:保存点，savepoint不会结束当前事务， 普通提交和回滚都会结束当前事务的

17-2-5：JDBC中大数据量的分页解决方法?

最好的办法是利用sql语句进行分页， 这样每次查询出的结果集中就只包含某页的数据内容。

17-2-6：常见的JDBC异常有哪些

`java.sql.SQLException` 这是JDBC异常的基类。 `java.sql.BatchUpdateException` 当批处理操作执行失败的时候可能会抛出这个异常。 这取决于具体的JDBC驱动的实现， 它也可能直接抛出基类异常java.sql.SQLException。 `java.sql.SQLWarning` SQL操作出现的警告信息。 `java.sql.DataTruncation` 字段值由于某些非正常原因被截断了 （不是因为超过对应字段类型的长度限制）。

17-2-7：JDBC中存在哪些不同类型的锁?

乐观锁——只有当更新数据的时候才会锁定记录。 悲观锁——从查询到更新和提交整个过程都会对数据记录进行加锁。

18.分库分表

18-1：为什么要分库分表？（看法）

数据库数据会随着业务的发展而不断增多， 因此数据操作，如增删改查的开销也会越来越大。 再加上物理服务器的资源有限（CPU、磁盘、内存、IO 等）。 最终数据库所能承载的数据量、数据处理能力都将遭遇瓶颈。 也就是说需要合理的数据库架构来存放不断增长的数据， 这个就是分库分表的设计初衷。 目的就是为了缓解数据库的压力，最大限度提高数据操作的效率。

18-2：数据分表

如果单表的数据量过大， 例如千万级甚至更多，那么在操作表的时候就会加大系统的开销。 每次查询会消耗数据库大量资源， 如果需要多表的联合查询，这种劣势就更加明显了。 比如说MySQL来说，在插入数据的时候，会对表进行加锁，分为表锁定和行锁定。 无论是哪种锁定方式，都意味着前面一条数据在操作表或者行的时候， 后面的请求都在排队，当访问量增加的时候，都会影响数据库的效率。

18-2-1：垂直分表

就是说，根据业务把一个表中的字段（Field）分到不同的表中。 这些被分出去的数据通常根据业务需要， 例如分出去一些不是经常使用的字段，一些长度较长的字段。 一般被拆分的表的字段数比较多。 主要是避免查询的时候出现因为数据量大而造成的“跨页”问题。 一般这种拆分在数据库设计之初就会考虑， 尽量在系统上线之前考虑调整。已经上线的项目，做这种操作是要慎重考虑的。

18-2-2：水平分表

将一个表中的数据，按照关键字（例如：ID）（或取 Hash 之后） 对一个具体的数字取模，得到的余数就是需要存放到的新表的位置。 用这种方式存放数据以后，在访问具体数据的时候需要 通过一个 Mapping Table 获取对应要响应的数据来自哪个数据表

18-3：数据分库

因为每个物理数据库支持数据都是有限的， 每一次的数据库请求都会产生一次数据库链接， 当一个库无法支持更多访问的时候， 我们会把原来的单个数据库分成多个，帮助分担压力。 比如说： 根据业务不同分库，这种情况都会把主营业务和其他功能分开。 例如可以分为订单数据库，核算数据库，评论数据库。 根据冷热数据进行分库，用数据访问频率来划分， 例如：近一个月的交易数据属于高频数据， 2-6 个月的交易数据属于中频数据， 大于 6 个月的数据属于低频数据。 根据访问数据的地域/时间范围进行分库。 通常数据分库之后，每一个数据库包含多个数据表， 多个数据库会组成一个 Cluster/Group， 提高了数据库的可用性，并且可以把读写做分离。 Master 库主要负责写操作，Slave 库主要负责读操作。 在应用访问数据库的时候会通过一个负载均衡代理， 通过判断读写操作把请求路由到对应的数据库。 如果是读操作，也会根据数据库设置的权重或者平均分配请求。 另外，还有数据库健康监控机制，定时发送心跳检测数据库的健康状况。 如果 Slave 出现问题，会启动熔断机制停止对其的访问； 如果 Master 出现问题，通过选举机制选择新的 Master 代替。

18-4：数据库的扩容

18-4-1：为什么要数据库扩容

分库之后的数据库会遇到数据扩容或者数据迁移的情况。

18-4-2：主从数据库扩容

假设有两个数据库集群，每个集群分别有 M1 S1 和 M2 S2 互为主备。 由于 M1 和 S1 互为主备所以数据是一样的， M2 和 S2 同样。把原有的 ID %2 模式切换成 ID %4 模式， 也就是把两个数据集群扩充到 4 个数据库集群。 负载均衡器直接把数据路由到原来两个 S1 和 S2 上面， 同时 S1 和 S2 会停止与 M1 和 M2 的数据同步，单独作为主库（写操作）存在。 这些修改不需要重启数据库服务， 只需要修改代理配置就可以完成。 由于 M1 M2 S1 S2 中会存在一些冗余的数据， 可以后台起服务将这些冗余数据删除，不会影响数据使用。 此时，再考虑数据库可用性，将扩展后的 4 个主库进行主备操作， 针对每个主库都建立对应的从库， 前者负责写操作，后者负责读操作。 下次如果需要扩容也可以按照类似的操作进行。

18-4-3：双写数据库扩容

在没有数据库主从配置的情况下的扩容，假设有数据库M1 M2 需要对目前的两个数据库做扩容， 扩容之后是4个库 新增的库是 M3，M4 路由的方式分别是 ID%2=0 和 ID%2=1。 这个时候新的数据会同时进入 M1 M2 M3 M4 四个库中， 而老数据的使用依旧从 M1 M2 中获取。 与此同时，后台服务对 M1 M3，M2 M4 做数据同步， 可以先做全量同步再做数据校验。 当完成数据同步之后，四个库的数据保持一致了， 修改负载均衡代理的配置为 ID%4 的模式。 此时扩容就完成了，从原来的 2 个数据库扩展成 4 个数据库。 当然会存在部分的数据冗余， 需要像上面一个方案一样通过后台服务删除这些冗余数据， 删除的过程不会影响业务。

18-5：分库分表，id如何处理

方式1——UUID：不适合作为主键，因为太长了， 并且无序不可读，查询效率低。 比较适合用于生成唯一的名字的 标示比如文件的名字。 方式2——数据库自增id : 两台数据库分别设置不同步长， 生成不重复ID的策略来实现高可用。 这种方式生成的 id 有序，但是需要独 立部署数据库实例，成本高， 还会有性能瓶颈。 方式3——利用 redis 生成 id : 性能比较好，灵活方便， 不依赖于数据库。但是， 引入了新的组件造成系统更加复杂可用性降低，编码更加复杂， 增加了系统成本。

1.缓存

1-1：缓存思想

我们为了避免用户在请求数据的时候获取速度过于缓慢， 所以我们在数据库之上增加了缓存这一层来弥补。

1-2：使用缓存为系统带来了什么问题

1. 系统复杂性增加：引入缓存之后， 你要维护缓存和数据库的数据一致性、维护热点缓存等等。
2. 系统开发成本增加：引入缓存意味着系统需要一个单独的缓存服务， 这是需要花费相应的成本的，并且这个成本还是很贵的， 毕竟耗费的是宝贵的内存。 如果你只是简单的使用一下本地缓存 存储一下简单的数据， 并且数据量不大的话， 那么就不需要单独去弄一个缓存服务。

1-3：本地缓存解决方案

一：JDK 自带的 HashMap 和 ConcurrentHashMap 了。

ConcurrentHashMap 可以看作是线程安全版本的 HashMap ， 两者都是存放 key/value 形式的键值对。 但是，大部分场景来说不会使用这两者当做缓存， 因为只提供了缓存的功能，并没有提供其他诸如过期时间之类的功能。

二：Spring Cache

使用 Spring Cache 的注解实现缓存的话， 代码会看着很干净和优雅， 但是很容易出现问题比如缓存穿透、内存溢出。

1-4：为什么要有分布式缓存?/为什么不直接用本地缓存?

其实分布式缓存类似于一种内存数据库的服务， 它的最终作用就是提供缓存数据的服务。 本地的缓存的优势是低依赖， 比较轻量并且通常相比于使用分布式缓存要更加简单。 本地缓存对分布式架构支持不友好， 比如同一个相同的服务部署在多台机器上的时候， 各个服务之间的缓存是无法共享的， 因为本地缓存只在当前机器上有。 本地缓存容量受服务部署所在的机器限制明显。 如果当前系统服务所耗费的内存多， 那么本地缓存可用的容量就很少。 使用分布式缓存之后， 缓存部署在一台单独的服务器上， 即使同一个相同的服务部署在再多机器上， 也是使用的同一份缓存。 并且，单独的分布式缓存服务的性能、 容量和提供的功能都要更加强大。 使用分布式缓存的缺点呢， 也很显而易见， 那就是你需要为分布式缓存引入额外的服务 比如Redis或Memcached， 你需要单独保证 Redis 或 Memcached 服务的高可用。

1-5：缓存读写模式/更新策略

1. Cache Aside Pattern（旁路缓存模式） 写：更新 DB，然后直接删除 cache 。 读：从 cache 中读取数据，读取到就直接返回， 读取不到的话，就从 DB 中取数据返回， 然后再把数据放到 cache 中。 Cache Aside Pattern 中服务端需要同时维系 DB 和 cache， 并且是以 DB 的结果为准。 另外，Cache Aside Pattern 有首次请求数据一定不在 cache 的问题， 对于热点数据可以提前放入缓存中。 比较适合读请求比较多的场景。

2. Read/Write Through Pattern（读写穿透） 写：先查 cache，cache 中不存在， 直接更新 DB。 cache 中存在， 则先更新 cache， 然后 cache 服务自己更新 DB（同步更新 cache 和 DB）。 读： 从 cache 中读取数据，读取到就直接返回 。 读取不到的话，先从 DB 加载， 写入到 cache 后返回响应。

Read-Through Pattern 实际只是在 Cache-Aside Pattern 之上进行了封装。 在 Cache-Aside Pattern 下，发生读请求的时候， 如果 cache 中不存在对应的数据， 是由客户端自己负责把数据写入 cache， 而 Read Through

Pattern 则 是 cache 服务自己来写入缓存的， 这对客户端是透明的。 服务端把 cache 视为主要数据存储， 从中读取数据并将数据写入其中。 cache 服务负责将此数据读取和写入 DB， 从而减轻了应用程序的职责。

3. Write Behind Pattern（异步缓存写入） 由 cache 服务来负责 cache 和 DB 的读写。 Write Behind Caching 则是只更新缓存， 不直接更新 DB，而是改为异步批量的方式来更新 DB。 Write Behind Pattern 下 DB 的写性能非常高， 尤其适合一些数据经常变化的业务场景 比如说一篇文章的点赞数量、阅读数量。 往常一篇文章被点赞 500 次的话， 需要重复修改 500 次 DB， 但是在 Write Behind Pattern 下可能只需要修改一次 DB 就可以了。 但是，这种模式同样也给 DB 和 Cache 一致性带来了新的考验， 很多时候如果数据还没异步更新到 DB 的话，Cache 服务宕机了。

1-6：缓存数据的处理流程是怎样的?

1. 如果用户请求的数据在缓存中就直接返回。
2. 缓存中不存在的话就看数据库中是否存在。
3. 数据库中存在的话就更新缓存中的数据。
4. 数据库中不存在的话就返回空数据。

2.Redis

2-1：为什么要用 redis/为什么要用缓存

1. 高性能方面，假如用户第一次访问数据库中的某些数据。 这个过程会比较慢，因为是从硬盘上读取的。 将该用户访问的数据存在缓存中， 这样下一次再访问这些数据的时候 就可以直接从缓存中获取了。 操作缓存就是直接操作内存， 所以速度相当快。 如果数据库中的对应数据改变的之后， 同步改变缓存中相应的数据即可!
2. 高并发：直接操作缓存能够承受的请求 是远远大于直接访问数据库的， 所以我们可以考虑把数据库中的部分数据转移到缓存中去， 这样用户的一部分请求会 直接到缓存这里而不用经过数据库。

2-2：为什么要用 redis 而不用 map/guava 做缓存?

缓存分为本地缓存和分布式缓存。 以 Java 为例，使用自带的 map 或者 guava 实现的是本地缓存， 最主要的特点是轻量以及快速， 生命周期随着 jvm 的销毁而结束， 并且在多实例的情况下， 每个实例都需要各自保存一份缓存， 缓存不具有一致性。

使用 redis 或 memcached 之类的称为分布式缓存， 在多实例的情况下，各实例共用一份缓存数据， 缓存具有一致性。 缺点是需要保持 redis 或 memcached服务的高可用， 整个程序架构上较为复杂。

2-3：说一下 Redis 和 Memcached 的区别和共同点

共同点：

都是基于内存的数据库，一般都用来当做缓存使用。 都有过期策略。 两者的性能都非常高。

区别：

1. Redis 支持更丰富的数据类型（支持更复杂的应用场景）。 Redis 不仅仅支持简单的 k/v 类型的数据， 同时还提供 list，set，zset，hash 等数据结构的存储。 Memcached 只支持最简单的 k/v 数据类型。
2. Redis 支持数据的持久化， 可以将内存中的数据保持在磁盘中， 重启的时候可以再次加载进行使用, 而 Memecache 把数据全部存在内存之中。

3. Redis 有灾难恢复机制。 因为可以把缓存中的数据持久化到磁盘上。
4. Redis 在服务器内存使用完之后， 可以将不用的数据放到磁盘上。 但是，Memcached 在服务器内存使用完之后， 就会直接报异常。
5. Memcached 没有原生的集群模式， 需要依靠客户端来实现往集群中分片写入数据； 但是 Redis 目前是原生支持 cluster 模式的.
6. Memcached 是多线程， 非阻塞 IO 复用的网络模型； Redis 使用单线程的多路 IO 复用模型。
7. Redis 支持发布订阅模型、Lua 脚本、事务等功能， 而 Memcached 不支持。 并且，Redis 支持更多的编程语言。
8. Memcached过期数据的删除策略 只用了惰性删除， 而 Redis 同时使用了惰性删除与定期删除。

2-4：为什么说Redis快

1. 完全基于内存，数据存在内存中， 类似于hashmap， hashmap的优势就是查找和操作的 时间复杂度是o（1）
2. 数据结构进行了特别的设计，
3. 比如说SDS结构中字符串长度len，压缩链表
4. 采用单线程，避免了不必要的上下文切换和竞争条件， 也不存在多线程或者多线程导致切换而消耗CPU， 不用去考虑各种所得问题，不存在枷锁释放 操作，没有因为可能出现死锁二导致的性能消耗
5. 使用多路I/O复用模型，非阻塞IO， 多路IO复用模型利用select、poll、epoll可以 同时监察多个流的IO事件时， 就从阻塞态中唤醒，也是程序就会轮询一遍所有的流， 并且只依次顺序的处理就绪流， 这样就可以避免了大量的无用操作
6. RESP协议也就是Redis的序列化协议， 文本协议，解析迅速
7. 持久化采用子线程进行磁盘操作

2-5：Redis应用场景

1. 热点数据的缓存 由于redis访问速度块、支持的数据类型比较丰富， 所以redis很适合用来存储热点数据， 另外结合expire，我们可以设置过期时间然后再进行缓存更新操作
2. 限时业务的运用 redis中可以使用expire命令设置一个键的生存时间， 到时间后redis会删除它。利用这一特性可以运用 在限时的优惠活动信息、手机验证码等业务场景。
3. 计数器相关问题 redis由于incrby命令可以实现原子性的递增， 所以可以运用于高并发的秒杀活动、分布式序列号的生成、 比如限制一个手机号发多少条短信、一个接口一分钟限 制多少请求、一个接口一天限制调用多少次等等。
4. 排行榜相关问题 关系型数据库在排行榜方面查询速度普遍偏慢， 所以可以借助redis的SortedSet进行热点数据的排序。
5. 分布式锁
6. 延时操作 没有做过具体操作， 比如在订单生产后我们占用了库存， 10分钟后去检验用户是够真正购买， 如果没有购买将该单据设置无效，同时还原库存。
7. 分页、模糊搜索 可以利用zrangebylex方法可以进行模糊查询功能， 这个也是目前我在redis中发现的唯一一个支持对存储内容进行模糊查询的特性。 对公司进行项目的数据进行了模拟测试， 公司存储数据6000万左右，响应时间在700ms左右， 比mysql的like查询稍微快一点， 但是由于它可以避免大量的数据io操作， 所以总体还是比直接mysql查询更利于系统的性能保障。
8. 点赞、好友等相互关系的存储 Redis set可以实现set是可以自动排重的， 比如说在微博应用中， 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。
9. 队列 由于redis有list push和list pop这样的命令， 所以能够很方便的执行队列操作。

3.Redis五种数据类型与编码方式

3-1：五种数据类型

1. string，常用命令set,get,strlen,exists,dect,incr,setex

常用

```
127.0.0.1:6379> set key value #设置 key-value 类型的值
OK
127.0.0.1:6379> get key # 根据 key 获得对应的 value
"value"
127.0.0.1:6379> exists key  # 判断某个 key 是否存在
(integer) 1
127.0.0.1:6379> strlen key # 返回 key 所储存的字符串值的长度。
(integer) 5
127.0.0.1:6379> del key # 删除某个 key 对应的值
(integer) 1
127.0.0.1:6379> get key
(nil)
```

批量设置

```
127.0.0.1:6379> mset key1 value1 key2 value2 # 批量设置 key-value 类型的值
OK
127.0.0.1:6379> mget key1 key2 # 批量获取多个 key 对应的 value
```

计数器（字符串的内容为整数的时候可以使用）：

```
127.0.0.1:6379> set number 1
OK
127.0.0.1:6379> incr number # 将 key 中储存的数字值增一
(integer) 2
127.0.0.1:6379> get number
"2"
127.0.0.1:6379> decr number # 将 key 中储存的数字值减一
(integer) 1
127.0.0.1:6379> get number
"1"
```

过期：

```
127.0.0.1:6379> expire key  60 # 数据在 60s 后过期
(integer) 1
127.0.0.1:6379> setex key 60 value # 数据在 60s 后过期 (setex:[set] + [ex]pire)
OK
127.0.0.1:6379> ttl key # 查看数据还有多久过期
(integer) 56
```

2. list，常用命令rpush,lpop,lpush,rpop,lrange、llen

通过 rpush/lpop 实现队列：

```
127.0.0.1:6379> rpush myList value1 # 向 list 的头部（右边）添加元素
(integer) 1
127.0.0.1:6379> rpush myList value2 value3 # 向list的头部（最右边）添加多个元素
(integer) 3
127.0.0.1:6379> lpop myList # 将 list的尾部(最左边)元素取出
"value1"
127.0.0.1:6379> lrange myList 0 1 # 查看对应下标的list列表， 0 为 start,1为 end
1) "value2"
2) "value3"
127.0.0.1:6379> lrange myList 0 -1 # 查看列表中的所有元素，-1表示倒数第一
1) "value2"
2) "value3"
```

通过 rpush/rpop 实现栈：

```
127.0.0.1:6379> rpush myList2 value1 value2 value3
(integer) 3
127.0.0.1:6379> rpop myList2 # 将 list的头部(最右边)元素取出
"value3"
```

通过 lrange 查看对应下标范围的列表元素：

```
127.0.0.1:6379> rpush myList value1 value2 value3
(integer) 3
127.0.0.1:6379> lrange myList 0 1 # 查看对应下标的list列表， 0 为 start,1为 end
1) "value1"
2) "value2"
127.0.0.1:6379> lrange myList 0 -1 # 查看列表中的所有元素，-1表示倒数第一
1) "value1"
2) "value2"
3) "value3"
```

通过 llen 查看链表长度：

```
127.0.0.1:6379> llen myList
(integer) 3
```

3. hash,常见命令hset,hmset,hexists,hget,hgetall,hkeys,hvals

```
127.0.0.1:6379> hset userInfoKey name "guide" description "dev" age "24"
OK
127.0.0.1:6379> hexists userInfoKey name # 查看 key 对应的 value中指定的字段是否存在。
(integer) 1
127.0.0.1:6379> hget userInfoKey name # 获取存储在哈希表中指定字段的值。
"guide"
127.0.0.1:6379> hget userInfoKey age
"24"
127.0.0.1:6379> hgetall userInfoKey # 获取在哈希表中指定 key 的所有字段和值
1) "name"
2) "guide"
3) "description"
```

```
4) "dev"
5) "age"
6) "24"
127.0.0.1:6379> hkeys userInfoKey # 获取 key 列表
1) "name"
2) "description"
3) "age"
127.0.0.1:6379> hvals userInfoKey # 获取 value 列表
1) "guide"
2) "dev"
3) "24"
127.0.0.1:6379> hset userInfoKey name "GuideGeGe" # 修改某个字段对应的值
127.0.0.1:6379> hget userInfoKey name
"GuideGeGe"
```

4. set

```
127.0.0.1:6379> sadd mySet value1 value2 # 添加元素进去
(integer) 2
127.0.0.1:6379> sadd mySet value1 # 不允许有重复元素
(integer) 0
127.0.0.1:6379> smembers mySet # 查看 set 中所有的元素
1) "value1"
2) "value2"
127.0.0.1:6379> scard mySet # 查看 set 的长度
(integer) 2
127.0.0.1:6379> sismember mySet value1 # 检查某个元素是否存在set 中，只能接收单个元素
(integer) 1
127.0.0.1:6379> sadd mySet2 value2 value3
(integer) 2
127.0.0.1:6379> sinterstore mySet3 mySet mySet2 # 获取 mySet 和 mySet2 的交集并存放在 mySet3 中
(integer) 1
127.0.0.1:6379> smembers mySet3
1) "value2"
```

5. zset,常用命令zadd,zcard,zscore,zrange,zrevrange,zrem

```
127.0.0.1:6379> zadd myZset 3.0 value1 # 添加元素到 sorted set 中 3.0 为权重
(integer) 1
127.0.0.1:6379> zadd myZset 2.0 value2 1.0 value3 # 一次添加多个元素
(integer) 2
127.0.0.1:6379> zcard myZset # 查看 sorted set 中的元素数量
(integer) 3
127.0.0.1:6379> zscore myZset value1 # 查看某个 value 的权重
"3"
127.0.0.1:6379> zrange  myZset 0 -1 # 顺序输出某个范围区间的元素，0 -1 表示输出所
```

```
有元素
   1) "value3"
   2) "value2"
   3) "value1"
   127.0.0.1:6379> zrange  myZset 0 1 # 顺序输出某个范围区间的元素，0 为 start  1 为
stop
   1) "value3"
   2) "value2"
   127.0.0.1:6379> zrevrange  myZset 0 1 # 逆序输出某个范围区间的元素，0 为 start  1
为 stop
   1) "value1"
   2) "value2"
```

3-2：String类型

3-2-1：String-内部结构

redis在内部存储string都是主要还是以sds的数据结构实现的， 但是，在整个redis的数据存储过程中为了提高性能，内部做了很多优化。整体选择顺序应该是：

1. 只对长度小于或等于 21 字节，并且可以被解释为整数的字符串进行编码，使用整数存储
2. 尝试将 RAW 编码的字符串编码为 EMBSTR 编码，使用EMBSTR 编码
3. 这个对象没办法进行编码，尝试从 SDS 中移除所有空余空间，使用SDS编码

3-2-1-1：sds简介

在C语言中，字符串可以用'\0'结尾的char数组标示。 这种简单的字符串表示，在大多数情况下都能满足要求，但是不能高效的计算length和append数据。 所以Redis自己实现了SDS（简单动态字符串）的抽象类型。 SDS的数据结构，len表示sdshdr中数据的长度， free表示sdshdr中剩余的空间，buf表示实际存储数据的空间。

3-2-1-2：embstr和sds的区别（编码方式）

主要就是在于内存的申请和回收

1. embstr的创建只需分配一次内存， 而raw为两次（一次为sds分配对象， 另一次为redisObject分配对象，embstr省去了第一次）。 相对地，释放内存的次数也由两次变为一次。
2. embstr的redisObject和sds放在一起，更好地利用缓存带来的优势
3. redis并未提供任何修改embstr的方式， 即embstr是只读的形式。 对embstr的修改实际上是先转换为raw再进行修改。

3-2-1-3：sds对象创建

sds对象创建sdsnewlen分配了一次内存。 robj对象的创建又分配了一次内存。 整个sds对象的创建其实就是 分配内存并初始化len和free字段。

3-2-1-4：sds内存扩容

当字符串长度小于SDS_MAX_PREALLOC (1024*1024)， 那么就以2倍的速度扩容， 当字符串长度大于SDS_MAX_PREALLOC， 那么就以+SDS_MAX_PREALLOC的速度扩容。

3-2-1-5：String-sds缩容

释放内存的过程中修改len和free字段， 并不释放实际占用内存。

3-2-2：动态字符串与C语言自带字符串的区别

1. 常数复杂度获取字符串长度 由于len属性的存在， 获取SDS字符串的长度只需要读取len属性，时间复杂度为O(1)。 而对于C语言， 获取字符串的长度通常是经过遍历计数来实现的，时间复杂度为O(n)
2. 避免缓冲区溢出 对于SDS数据类型，在进行字符修改的时候， 会首先根据记录的len属性检查内存空间是否满足需求， 如果不满足，会进行相应的空间扩展， 然后在进行修改操作，所以不会出现缓冲区溢出。
3. 减少修改字符串的内存重新分配次数 C语言由于不记录字符串的长度， 所以如果要修改字符串， 必须要重新分配内存（先释放再申请）， 因为如果没有重新分配， 字符串长度增大时会造成内存缓冲区溢出，字符串长度减小时会造成内存泄露。 而对于SDS，由于len属性和free属性的存在， 对于修改字符串SDS实现了空间预分配和惰性空间释放两种策略:
    - 1、空间预分配:对字符串进行空间扩展的时候， 扩展的内存比实际需要的多， 这样可以减少连续执行字符串增长 操作所需的内存重分配次数。
    - 2、惰性空间释放:对字符串进行缩短操作时， 程序不立即使用内存重新分配 来回收缩短后多余的字节， 而是使用 free属性将这些字节的数量记录下来， 等待后续使用。
4. 二进制安全 因为C语言的字符串以空字符作为字符串结束的标识， 而对于一些二进制文件（如图片等），内容可能包括空字符串， 因此C字符串无法正确存取; 而所有SDS的 API 都是 以处理二进制的方式来处理buf里面的元素， 并且SDS不是以空字符串来判断是否结束， 而是以len属性表示的长度来判断字符串是否结束。
5. 兼容部分C字符串函数 虽然 SDS 是二进制安全的， 但是一样遵从每个字符串都是以空字符串结尾的惯例， 这样可以重用C语言库<string.h>中的一部分函数。

3-3：list类型

3-3-1：list-内部结构

redis list数据结构底层 采用压缩列表ziplist或双向链表linkedlist两种数据结构进行存储， 首先以ziplist进行存储，在不满足ziplist的存储要求后转换为linkedlist列表。也就是说 列表对象保存的所有字符串元素的长度小于64字节，用ziplist。 列表对象保存的元素数量小于512个。

3-3-1-1：ziplist

ziplist的数据结构主要包括两层，ziplist和zipEntry。 ziplist包括zip header、zip entry、zip end三个模块。zipentry由prevlen、encoding&length、value三部分组成。 prevlen主要是指前面zipEntry的长度，coding&length是指编码字段长度和实际 存储value的长度，value是指真正的内容。 每个key/value存储结果中key 用一个zipEntry存储，value用一个zipEntry存储。

3-3-2：list元素添加过程

1. 创建list对象并添加到db的数据结构当中
2. 针对每个待插入的元素添加到list当中， list的每个元素的插入过程中，我们会对是否需要进行转码作两个判断:
    - 对每个插入元素的长度进行判断是否进行ziplist->linkedlist的转码。
    - 对list总长度是否超过ziplist最大长度的判断。

3-4：hash类型

3-4-1：hash底层存储结构

redis的哈希对象的底层存储可以 使用ziplist（压缩列表）和hashtable。 当hash对象可以同时满足 哈希对象保存的所有键值对的键和值的字符串长度都小于64字节 哈希对象保存的键值对数量小于512个 哈希对象就可以使用ziplist编码。

3-4-2：redis hash存储过程源码分析

我只是大体看过hset命令，这个过程应该是：

1. 首先查看hset中key对应的value是否存在
2. 判断key和value的长度确定是否需要从zipList到hashtab转换，
3. 对key/value进行string层面的编码，解决内存效率问题。
4. 更新hash节点中key/value问题。

3-4-3：Redis字典底层如何解决冲突

拉链法等方法

3-4-4：hash如何扩容

正常情况下，当hash表中元素的个数等于第一维数组的长度时， 就会开始扩容，扩容的新数组是原数组大小的2倍。 不过如果Redis正在做bgsave(持久化命令)， 为了减少内存也得过多分离， Redis 尽量不去扩容，但是如果hash表非常满了， 达到了第一维数组长度的 5 倍了， 这个时候就会强制扩容。 当hash表因为元素 逐渐被删除变得越来越稀疏时， Redis会对hash表进行缩容 来减少hash表的第一维数组空间占用。 所用的条件是元素个数 低于数组长度的10%， 缩容不会考虑Redis是否在做bgsave。

3-4-5：什么是渐进式--rehash

扩容和收缩操作不是一次性、集中式完成的， 而是分多次、渐进式完成的。 如果保存在Redis中的键值对只有几个几十个， 那么rehash操作可以瞬间完成， 但是如果键值对有几百万，几千万甚至几亿， 那么要一次性的进行 rehash， 势必会造成Redis一段时间内不能进行别的操作。 所以 Redis采用渐进式rehash 这样在进行渐进式rehash期间， 字典的删除查找更新等操作 可能会在两个哈希表上进行， 第一个哈希表没有找到， 就会去第二个哈希表上进行查找。 但是进行增加操作， 一定是在新的哈希表上进行的。

3-5：set类型

3-5-1：set底层存储

redis的集合对象set的底层存储结构底层 使用了intset和hashtable两种数据结构存储的， intset我认为应该是一种数组类型的， hashtable就是普通的哈希表（key为set的值，value为null）。 set的底层存储intset和hashtable是存在编码转换的， 使用intset存储必须满足 集合对象保存的所有元素都是整数值 集合对象保存的元素数量不超过512个 否则使用hashtable，

3-5-1-1：intset的数据结构

intset内部其实是一个数组（int8_t coentents[]数组）， 而且存储数据的时候是有序的，因为在查找数据的时候是通过二分查找来实现的。

3-5-2：set存储过程

set的sadd命令为例子，整个添加过程如下：

1. 检查set是否存在不存在则创建一个set结合。
2. 根据传入的set集合一个个进行添加， 添加的时候需要进行内存压缩。
3. setTypeAdd执行set添加过程中 会判断是否进行编码转换。

3-6：zset类型

3-6-1：zset底层存储结构

zset底层的存储结构 包括ziplist（压缩列表）或skiplist（跳表）， 在同时满足 有序集合保存的元素数量小于128个 有序集合保存的所有元素的长度小于64字节 的时候，可以使用ziplist，其他时候使用skiplist， 当ziplist作为zset的底层存储结构时候， 每个集合元素使用两个紧挨 在一起的压缩列表节点来保存， 第一个节点保存元素的成员， 第二个元素保存元素的分值。 当skiplist作为zset的底层存储结构的时候， 使用skiplist按序保存元素及分值， 使用dict来保存元素和分值的映射关系。

3-6-2：skiplist数据结构

skiplist作为zset的存储结构， 主要是包括一个dict对象和一个skiplist对象。 dict保存key/value，key为元素，value为分值； skiplist保存的有序的元素列表， 每个元素包括元素和分值。 两种数据结构下的元素指向相同的位置。

3-6-3：zset存储过程

以zadd的操作作为例子进行分析，

1. 解析参数得到每个元素及其对应的分值
2. 查找key对应的zset是否存在不存在则创建
3. 如果存储格式是ziplist， 那么在执行添加的过程中 我们需要区分元素存在和不存在两种情况， 存在情况下先删除后添加； 不存在情况下则添加并且需要考 虑元素的长度是否超出限制或 实际已有的元素个数是否超过 最大限制进而决定是否转为skiplist对象。
4. 如果存储格式是skiplist， 那么在执行添加的过程中 我们需要区分元素存在和不存在两种情况， 存在的情况下先删除后添加， 不存在情况下那么就直接添加， 在skiplist当中添加完以后 我们同时需要更新dict的对象。

3-6-4：skiplist与平衡树、哈希表的比较

1. skiplist和各种平衡树（如AVL、红黑树等）的元素是有序排列， 而哈希表不是有序的的。因此，在哈希表上只能做单个key的查找， 不适合做范围查找。 所谓范围查找，指的是查找那些大小在指定的两个值之间的所有节点。
2. 在做范围查找的时候，平衡树比skiplist操作要复杂。 在平衡树上，我们找到指定范围的小值之后， 还需要以中序遍历的顺序继续寻找其它不超过大值的节点。 而在skiplist上进行范围查找就非常简单， 只需要在找到小值之后， 对第1层链表进行若干步的遍历就可以实现。
3. 平衡树的插入和删除操作可能引发子树的调整， 逻辑复杂， 而skiplist的插入和删除 只需要修改相邻节点的指针， 操作简单又快速。
4. 从内存占用上来说，skiplist比平衡树更灵活一些。 一般来说，平衡树每个节点包含2个指针(分别指向左右子树)， 而skiplist每个节点包含的指针数目平均为1/(1-p)， 具体取决于参数p的大小。 如果像Redis里的实现一样，取p=1/4， 那么平均每个节点包含1.33个指针， 比平衡树更有优势。

5. 查找单个key，skiplist和平衡树的时间复杂度都为O(logn)， 大体相当; 而哈希表在保持较低的哈希值冲突概率的前提下， 查找时间复杂度接近O(1)， 性能更高一些。
6. 从算法实现难度上来比较，skiplist比平衡树要简单得多

4.Redis的单线程

4-1：为什么Redis是单线程

1. 可维护性对于一个项目来说非常重要， 如果代码难以调试和测试，问题也经常难以复现， 这对于任何一个项目来说都会严重地影响项目的可维护性。 多线程模型虽然在某些方面表现优异， 但是它却引入了程序执行顺序的不确定性， 代码的执行过程不再是串行的， 多个线程同时访问的变量如果没有谨慎处理就会带来诡异的问题。 引入了多线程，我们就必须要同时引入并发控制 来保证在多个线程同时访问数据时程序行为的正确性， 这就需要工程师额外维护并发控制的相关代码， 例如，我们会需要在可能被并发读写的变量上增加互斥锁： 在访问这些变量或者内存之前也需要先对获取互斥锁， 一旦忘记获取锁或者忘记释放锁就可能会导致各种诡异的问题， 管理相关的并发控制机制也需要付出额外的研发成本和负担。
2. 使用单线程模型也并不意味着程序不能并发的处理任务， Redis 虽然使用单线程模型处理用户的请求，但是它却使用I/O多路复用机制并发处理 来自客户端的多个连接，同时等待多个连接发送的请求。 在 I/O 多路复用模型中，最重要的函数调用就是select以及类似函数， 该方法的能够同时监控多个文件描述符的可读可写情况， 当其中的某些文件描述符可读或者可写时， select 方法就会返回可读以及可写的文件描述符个数。 使用 I/O 多路复用技术能够极大地减少系统的开销， 系统不再需要额外创建和维护进程和线程来监听来自客户端的大量连接， 减少了服务器的开发成本和维护成本。
3. Redis 选择单线程模型的决定性原因， Redis 并不是 CPU 密集型的服务， 如果不开启AOF备份，所有Redis的操作都会在内存中 完成不会涉及任何的 I/O 操作， 这些数据的读写由于只发生在内存中， 所以处理速度是非常快的； 整个服务的瓶颈在于网络传输带来的 延迟和等待客户端的数据传输， 也就是网络 I/O， 所以使用多线程模型处理全部的外部请求可能不是一个好的方案。 比如说多线程中 保存线程 1 的执行上下文； 加载线程 2 的执行上下文； 频繁的对线程的上下文进行切换 可能还会导致性能地急剧下降， 这可能会导致我们不仅没有提升请求处理的平均速度， 反而进行了负优化

4-2：既然是单线程，那怎么监听大量的客户端连接呢?

Redis 通过IO 多路复用程序来监听 来自客户端的大量连接（或者说是监听多个 socket）， 它会将感兴趣的事件及类型(读、写） 注册到内核中并监听每个事件是否发生。 这样的好处非常明显： I/O 多路复用技术的使用让 Redis 不需要额外创建多余的线程 来监听客户端的大量连接， 降低了资源的消耗（和 NIO 中的 Selector 组件很像）。

4-3：Redis为什么又采用了多线程

适用于单个Redis服务器的命令不适用于数据分区； 数据分区无法解决热点读/写问题； 数据偏斜，重新分配和放大/缩小变得更加复杂 所以就需要提高网络 IO 读写性能

5.过期删除策略

5-1：Redis 给缓存数据设置过期时间有啥用?

因为内存是有限的， 如果缓存中的所有数据都是一直保存的话， 分分钟直接OOM的。 很多时候，我们的业务场景就是 需要某个数据只在某一时间段内存在， 比如我们的短信验证码可能只在1分钟内有效， 用户登录的

token 可能只在 1 天内有效。 如果使用传统的数据库来处理的话， 一般都是自己判断过期， 这样更麻烦并且性能要差很多。

5-2：Redis是如何判断数据是否过期的呢?

Redis 通过一个叫做过期字典（可以看作是hash表） 来保存数据过期的时间。 过期字典的键指向Redis数据库中的某个key(键)， 过期字典的值是一个long long类型的整数， 这个整数保存了key所指向的数据库键的过期时间。

5-3：过期策略分类

1. 惰性删除： 只会在取出key的时候才对数据进行过期检查。 这样对CPU最友好， 但是可能会造成太多过期 key 没有被删除。
2. 定期删除： 每隔一段时间抽取一批key执行删除过期key操作。 并且，Redis 底层会并通过限制删除 操作执行的时长和频率来 减少删除操作对CPU时间的影响。
3. 立即删除。在设置键的过期时间时， 创建一个回调事件，当过期时间达到时， 由时间处理器自动执行键的删除操作。

5-4：缓存淘汰机制

volatile-lru（least frequently used）：从已设置过期时间的数据集 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 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 中任意选择数据淘汰 allkeys-lru（least recently used）：当内存不足以容纳新写入数据时， 在键空间中，移除最近最少使用的key allkeys-random：从数据集中任意选择数据淘汰 no-eviction：禁止驱逐数据，也就是说当内存不足以容纳新写入数据时， 新写入操作会报错。这个应该没人使用吧！ volatile-lfu（least frequently used）：从已设置过期时间的数据集 中挑选最不经常使用的数据淘汰 allkeys-lfu（least frequently used）：当内存不足以容纳新写入数据时， 在键空间中，移除最不经常使用的 key

5-5：LRU算法原理

如果一个数据在最近一段时间没有被访问到， 那么在将来它被访问的可能性也很小。 也就是说，当限定的空间已存满数据时， 应当把最久没有被访问到的数据淘汰。

5-5-1：如何实现LRU

方法三种

1. 用一个数组来存储数据，给每一个数据项标记一个访问时间戳， 每次插入新数据项的时候， 先把数组中存在的数据项的时间戳自增， 并将新数据项的时间戳置为0并插入到数组中。 每次访问数组中的数据项的时候， 将被访问的数据项的时间戳置为0。 当数组空间已满时， 将时间戳最大的数据项淘汰。
2. 利用一个链表来实现， 每次新插入数据的时候将新数据插到链表的头部； 每次缓存命中（即数据被访问）， 则将数据移到链表头部； 那么当链表满的时候， 就将链表尾部的数据丢弃。
3. 利用链表和hashmap。 当需要插入新的数据项的时候， 如果新数据项在链表中存在（一般称为命中），则把该节点移到链表头部， 如果不存在，则新建一个节点， 放到链表头部，若缓存满了， 则把链表最后一个节点删除即可。 在访问数据的时候， 如果数据项在链表中存在， 则把该节点移到链表头部， 否则返回-1。 这样一来在链表尾部的节点就是 最近最久未访问的数据项。

5-5-2：LRU手写

```
public class LRULinkedHashMap<K, V> extends LinkedHashMap<K, V> {
    private final int maxCapacity;
    private static final float DEFAULT_LOAD_FACTOR = 0.75f;
    private final Lock lock = new ReentrantLock();
    public LRULinkedHashMap(int maxCapacity) {
        super(maxCapacity, DEFAULT_LOAD_FACTOR, true);
        this.maxCapacity = maxCapacity;  }
    @Override
    protected boolean removeEldestEntry(java.util.Map.Entry<K, V> eldest) {
        return size() > maxCapacity;
    }
    @Override
    public boolean containsKey(Object key) {
        try {
            lock.lock();
            return super.containsKey(key);
        } finally {
            lock.unlock();  }  }
    @Override
    public V get(Object key) {
        try {
            lock.lock();
            return super.get(key);
        } finally {
            lock.unlock();  }  }
    @Override
    public V put(K key, V value) {
        try {
            lock.lock();
            return super.put(key, value);
        } finally {
            lock.unlock();  }  }
    public int size() {
        try {
            lock.lock();
            return super.size();
        } finally {
            lock.unlock();  }  }
    public void clear() {
        try {
            lock.lock();
            super.clear();
        } finally {
            lock.unlock();  }  }
    public Collection<Map.Entry<K, V>> getAll() {
        try {
            lock.lock();
            return new ArrayList<Map.Entry<K, V>>(super.entrySet());
        } finally {
            lock.unlock();  }  }  }
```

6.持久化机制

6-1：什么是Redis持久化?

将数据(如内存中的对象)保存到可永久保存的存储设备中。

6-2：Redis 为什么要持久化?

Redis 中的数据类型都支持Push/Pop、Add/Remove 及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。 在此基础上，Redis 支持各种不同方式的排序。 为了保证效率，数据都是缓存在内存中。 因为数据都是缓存在内存中的， 当你重启系统或者关闭系统后， 缓存在内存中的数据都会消失殆尽，再也找不回来了。所以，为了让数据能够长期保存， 就要将 Redis 放在缓存中的数据做持久化存储。

6-3：Redis 怎么实现持久化?

Redis有两种持久化的方式：快照（RDB文件）和追加式文件（AOF文件）

RDB 持久化：该机制可以在指定的时间间隔内生成数据集的时间点快照。 AOF 持久化：记录服务器执行的所有写操作命令， 并在服务器启动时，通过重新执行这些命令来还原数据集。 AOF文件中的命令全部以 Redis 协议的格式来保存， 新命令会被追加到文件的末尾。 Redis 还可以在后台对 AOF 文件进行重写（rewrite）， 使得 AOF 文件的体积不会超出保存数据集状态所需的实际大小 无持久化：让数据只在服务器运行时存在。 同时应用 AOF 和 RDB：当 Redis 重启时， 它会优先使用 AOF 文件来还原数据集， 因为 AOF 文件保存的数据集通常比 RDB 文件所保存的数据集更完整

6-3-1：RDB（快照）优缺点

优点： RDB 是一个非常紧凑（compact）的文件， 它保存了 Redis 在某个时间点上的数据集。 这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内， 每小时备份一次 RDB 文件， 并且在每个月的每一天， 也备份一个 RDB 文件。 这样的话，即使遇上问题， 也可以随时将数据集还原到不同的版本。 RDB 非常适用于灾难恢复：它只有一个文件， 并且内容都非常紧凑，可以在加密后将它传送到别的数据中心。 RDB 可以最大 Redis的性能：父进程在保存RDB文件时唯一要做的就是 fork出一个子进程，然后这个子进程就会处理接下来的所有保存工作， 父进程无须执行任何磁盘 I/O 操作。RDB 在恢复大数据集时的 速度比 AOF 的恢复速度要快。 缺点： 如果你需要尽量避免在服务器故障时丢失数据， RDB不太合适。 虽然 Redis 允许你设置不同的保存点来控制保存RDB文件的频率， 但是，因为RDB文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少5分钟才保存一次RDB文件。 在这种情况下，一旦发生故障停机， 你就可能会丢失好几分钟的数据。 每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程， 并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork() 可能会非常耗时， 造成服务器在某某毫秒内停止处理客户端； 如果数据集非常巨大， 并且 CPU 时间非常紧张的话， 那么这种停止时间甚至可能会长达整整一秒。

6-3-2：AOF 的优缺点。

优点： 1、使用 AOF 持久化会让 Redis 变得非常耐久： 你可以设置不同的 fsync 策略， 比如无 fsync ，每秒钟一次 fsync ， 或者每次执行写入命令时 fsync 。 AOF 的默认策略为每秒钟 fsync 一次， 在这种配置下，Redis 仍然可以保持良好的性能， 并且就算发生故障停机， 也最多只会丢失一秒钟的数据 fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求。 AOF 文件是一个只进行追加操作的日志文件， 因此对 AOF 文件的写入不需要进行 seek 即使日志因为某些原因而包含了未写入完整的命令 比如写入时磁盘已满，写入中途停机，等等， redis-check-aof 工具也可以轻易地修复这种问题。 2、Redis 可以在 AOF 文件体积变得过大时， 自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操

作是绝对安 全的，因为 Redis 在创建新 AOF 文件的过程中， 会继续将命令追加到现有的AOF 文件里面， 即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF文件创建完毕， Redis 就会从旧 AOF 文件切换到新 AOF 文件， 并开始对新 AOF 文件进行追加操作。 缺点： 对于相同的数据集来说， AOF 文件的体积通常要大于 RDB 文件的体积。 根据所使用的fsync策略， AOF 的速度可能会慢于RDB。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时， RDB 可以提供更有保证的最大延迟时间

6-4：Redis持久化数据和缓存怎么做扩容?

如果Redis被当做缓存使用，使用一致性哈希实现动态扩容缩容。 如果Redis被当做一个持久化存储使用， 必须使用固定的keys-to-nodes映射关系， 节点的数量一旦确定不能变化。 否则的话,在Redis节点需要动态变化的情况， 必须使用可以在运行时进行数据再平衡的一套系统

6-5：持久化期间工作流程

客户端向数据库发送写命令 (数据在客户端的内存中) 数据库接收到客户端的写请求 (数据在服务器的内存中) 数据库调用系统API将数据写入磁盘 (数据在内核缓冲区中) 操作系统将写缓冲区传输到磁盘控控制器 (数据在磁盘缓存中) 操作系统的磁盘控制器将数据写入实际的物理媒介 中 (数据在磁盘中)

6-6：持久化机制

修改

==========

==========

==========

==

7.集群主从复制

7-1：Redis的结构

Redis的主从结构可以采用一主多从或者级联结构， Redis主从复制可以根据是否是全量分为全量同步和增量同步

7-2：Redis主从复制的特点

1. redis采用异步方式复制数据到slave节点， 从redis2.8开始，slave节点会周期性地 确认自己每次复制的数据量；
2. 一个master节点可以配置多个slave节点；
3. slave节点可以连接其他的slave节点；
4. slave节点做复制的时候，不会阻塞master节点的正常工作；
5. slave节点做复制的时候，也不会阻塞对自己的查询操作， 它会用旧数据集来提供服务， 但在复制完成时，需要删除旧数据集， 加载新数据集，这时会暂停对外服务；
6. slave节点主要用来横向扩容，做读写分离， 扩容的slave节点可以提高读的吞吐量；

7. 如果采用主从架构，必须开启master节点的持久化， 不建议用slave节点作master节点的数据热备， 因为如果一旦关掉master的持久化， 可能在master宕机重启时数据是空的， 然后一经复制，slave节点也会随之丢失。

、 、
、 、
、 、 、 、 、 、 、 、

8.缓存穿透、雪崩、击穿

8-1：缓存穿透

8-1-1：什么是缓存穿透

解释 1：缓存查询一个没有的 key，同时数据库也没有， 如果黑客大量的使用这种方式，那么就会导致 DB 宕机。

解决方案：我们可以使用一个默认值来防止， 例如，当访问一个不存在的 key，然后再去访问数据库， 还是没有，那么就在缓存里放一个占位符，下次来的时候， 检查这个占位符，如果发生时占位符，就不去数据库查询了，防止 DB 宕机。

解释 2：大量请求查询一个刚刚失效的 key， 导致 DB 压力倍增，可能导致宕机， 但实际上，查询的都是相同的数据。

解决方案：可以在这些请求代码加上双重检查锁。 但是那个阶段的请求会变慢。不过总比 DB 宕机好。

8-1-2：缓存穿透的解决方案

1. 加入布隆过滤器 通过它我们可以非常方便地 判断一个给定数据是否存在于海量数据中。 我们需要的就是判断 key 是否合法， 有没有感觉布隆过滤器就是我们想要找的那个数据。 首先把所有可能存在的请求的值都存放在布隆过滤器中， 当用户请求过来，先判断用户发来的请求的值是否存在于布隆过滤器中。不存在的话，直接返回请求参数错误信息给客户端， 存在的话才会走下面的流程。
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

8-2：缓存雪崩

8-2-1：什么是缓存雪崩

缓存在同一时间大面积的失效， 后面的请求都直接落到了数据库上， 造成数据库短时间内承受大量请求。 比如说系统的缓存模块出了问题比如宕机导致不可用。 造成系统的所有访问，都要走数据库。 或者说秒杀活动，开始 12 个小时之前， 我们统一存放了一批商品到 Redis 中， 设置的缓存过期时间也是 12 个小时， 那么秒杀开始的时候， 这些秒杀的商品的访问直接就失效了。 导致的情况就是，相应的请求直接就落到了数据库上，就像雪崩一样可怕。

8-2-2：有哪些解决办法?

针对 Redis 服务不可用的情况： 采用 Redis 集群，避免单机出现问题整个缓存服务都没办法使用。 限流，避免同时处理大量的请求。 针对热点缓存失效的情况： 设置不同的失效时间比如随机设置缓存的失效时间。 缓存永不失效。

8-3：缓存击穿

8-3-1：什么是缓存击穿

对于一些设置了过期时间的key， 如果这些key可能会在某些时间点被超高并发地访问， 是一种非常“热点”的数据。 缓存在某个时间点过期的时候， 恰好在这个时间点对这个Key有大量的并发请求过来， 这些请求发现缓存过期一般都会从后端DB加载数据并回射到缓存， 这个时候大并发的请求可能会瞬间把后端DB压垮。

8-3-2：缓存击穿的解决方案

1. 使用互斥锁(mutex key) 是使用mutex。就是在缓存失效的时候（判断拿出来的值为空）， 不是立即去load db， 而是先使用缓存工具的某些带成功操作返回值的操作 （比如Redis的SETNX或者Memcache的ADD） 去set一个mutex key，当操作返回成功时， 再进行load db的操作并回设缓存； 否则，就重试整个get缓存的方法。

```
public String get(key) {
      String value = redis.get(key);
      if (value == null) { //代表缓存值过期
          //设置3min的超时，防止del操作失败的时候，下次缓存过期一直不能load db
          if (redis.setnx(key_mutex, 1, 3 * 60) == 1) {  //代表设置成功
               value = db.get(key);
                      redis.set(key, value, expire_secs);
                      redis.del(key_mutex);
              } else {  //这个时候代表同时候的其他线程已经load db并回设到缓存了，这时候重试获取缓存值即可
                      sleep(50);
                      get(key);  //重试
              }
          } else {
              return value;}}
```

2. "提前"使用互斥锁(mutex key) 在value内部设置1个超时值(timeout1), timeout1比实际的memcache timeout(timeout2)小。 当从cache读取到timeout1发现它已经过期时候， 马上延长timeout1并重新设置到cache。 然后再从数据库加载数据并设置到cache中。

```
v = memcache.get(key);
if (v == null) {
    if (memcache.add(key_mutex, 3 * 60 * 1000) == true) {
        value = db.get(key);
        memcache.set(key, value);
        memcache.delete(key_mutex);
    } else {
        sleep(50);
        retry();
    }
```

```
} else {
    if (v.timeout <= now()) {
        if (memcache.add(key_mutex, 3 * 60 * 1000) == true) {
            // extend the timeout for other threads
            v.timeout += 3 * 60 * 1000;
            memcache.set(key, v, KEY_TIMEOUT * 2);

            // load the latest value from db
            v = db.get(key);
            v.timeout = KEY_TIMEOUT;
            memcache.set(key, value, KEY_TIMEOUT * 2);
            memcache.delete(key_mutex);
        } else {
            sleep(50);
            retry();
        }
    }
}
```

3. 采用永远不过期的策略

(1) 从redis上看，确实没有设置过期时间， 这就保证了，不会出现热点key过期问题，也就是“物理”不过期。

(2) 从功能上看，如果不过期，可能就是成为静态的了 所以把过期时间存在key对应的value里， 如果发现要过期了， 通过一个后台的异步线程进行缓存的构建，也就是“逻辑”过期

```
String get(final String key) {
        V v = redis.get(key);
        String value = v.getValue();
        long timeout = v.getTimeout();
        if (v.timeout <= System.currentTimeMillis()) {
            // 异步更新后台异常执行
            threadPool.execute(new Runnable() {
                public void run() {
                    String keyMutex = "mutex:" + key;
                    if (redis.setnx(keyMutex, "1")) {
                        // 3 min timeout to avoid mutex holder crash
                        redis.expire(keyMutex, 3 * 60);
                        String dbValue = db.get(key);
                        redis.set(key, dbValue);
                        redis.delete(keyMutex);
                    }
                }
            });
        }
        return value;
}
```

8-3-3：几种方案优缺点

1. 方案一： 优势

    1. 思路简单
    2. 保证一致性 缺点
    3. 代码复杂度增大2. 存在死锁的风险3. 存在线程池阻塞的风险

2. 方案二 优点 加另外一个过期时间 保证一致性 缺点同上

3. 方案三： 优点

    1. 不过期(本文)
    2. 异步构建缓存，不会阻塞线程池 缺点
    3. 不保证一致性。
    4. 代码复杂度增大(每个value都要维护一个timekey)。
    5. 占用一定的内存空间(每个value都要维护一个timekey)。

8-4：什么是缓存并发竞争？怎么解决？

解释：多个客户端写一个 key，如果顺序错了， 数据就不对了。但是顺序我们无法控制。

解决方案：使用分布式锁，例如 zk， 同时加入数据的时间戳。同一时刻， 只有抢到锁的客户端才能写入，同时，写入时， 比较当前数据的时间戳和缓存中数据的时间戳。

9.缓存和数据库数据的一致性

9-1：什么是缓存和数据库双写不一致

连续写数据库和缓存，但是操作期间，出现并发了，数据不一致了。

9-2：解决方案

通常，更新缓存和数据库有几种顺序：

1. 先更新数据库，再更新缓存。
2. 先删缓存，再更新数据库。
3. 先更新数据库，再删除缓存。 那么 先更新数据库，再更新缓存。 这么做的问题是：当有 2 个请求同时更新数据， 那么如果不使用分布式锁， 将无法控制最后缓存的值到底是多少。 也就是并发写的时候有问题。 先删缓存，再更新数据库。 这么做的问题：如果在删除缓存后， 有客户端读数据，将可能读到旧数据， 并有可能设置到缓存中，导致缓存中的数据一直是老数据。 \有 2 种解决方案：\
    1. 使用“双删”，即删更删，最后一步的删除作为异步操作， 就是防止有客户端读取的时候设置了旧值。
    2. 使用队列，当这个 key 不存在时， 将其放入队列，串行执行， 必须等到更新数据库完毕才能读取数据。 先更新数据库，再删除缓存 如果先更新数据库，再删除缓存， 那么就会出现更新数据库之前有瞬间数据不是很及时。 同时，如果在更新之前，缓存刚好失效了， 读客户端有可能读到旧值， 然后在写客户端删除结束后再次设置了旧值，非常巧合的情况。 有 2 个前提条件：缓存在写

之前的时候失效， 同时，在写客户度删除操作结束后， 放置旧数据 —— 也就是读比写慢。设置有的写操作还会锁表。 如果出现了就可以使用双删 记录更新期间有没有客户端读数据库， 如果有，在更新完数据库之后，执行延迟删除。 还有一种可能，如果执行更新数据库，准备执行删除缓存时， 服务挂了，执行删除失败 可以通过订阅数据库的 binlog 来删除。

==========

==========

==========

10.事务

10-1：Redis事务的概念

Redis 事务的本质是一组命令的集合。 事务支持一次执行多个命令，一个事务中所有命令都会被序列化。 在事务执行过程，会按照顺序串行化执行队列中的命令， 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。

10-2：Redis事务的三个阶段

开启：以MULTI开始一个事务 入队：将多个命令入队到事务中， 接到这些命令并不会立即执行， 而是放到等待执行的事务队列里面 执行：由EXEC命令触发事务

10-3：Redis事务相关命令

multi，标记一个事务块的开始，返回 ok exec，执行所有事务块内，事务块内所有命令执行的先后顺序的返回值， 操作被，返回空值 nil discard，取消事务，放弃执行事务块内的所有命令，返回 ok watch，监视 key 在事务执行之前是否被其他指令改动， 若已修改则事务内的指令取消执行，返回 ok unwatch，取消 watch 命令对 key 的监视，返回 ok

10-4：Redis的ACID

Redis只一致性和隔离性两个特性，其他特性是不支持的。

10-4-1：原子性

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 如果一个事务队列中的所有命令都被成功地执行， 那么称这个事务执行成功 另一方面，如果Redis服务器进程在执行事务的过程中被停止 比如接到 KILL 信号、宿主机器停机，等等，那么事务执行失败事务失败时，Redis 也不会进行任何的重试或者回滚动作， 不满足要么全部全部执行，要么都不执行的条件

10-4-2：一致性

首先，如果一个事务的指令全部被执行， 那么数据库的状态是满足数据库完整性约束的 其次，如果一个事务中有的指令有错误， 那么数据库的状态是满足数据完整性约束的 最后，如果事务运行到某条指令时， 进程被kill掉了， 如果当前redis采用的是内存模式， 那么重启之后redis数据库是空的， 那么满足一致性条件 如果当前采用RDB模式存储的， 在执行事务时，Redis不会中断事务去执行保存RDB的工作， 只有在事务执行之后， 保存

RDB 的工作才有可能开始。 所以当 RDB 模式下的 Redis 服务器进程在事务中途被杀死时， 事务内执行的命令，不管成功了多少， 都不会被保存到 RDB 文件里。 恢复数据库需要使用现有的 RDB 文件， 而这个 RDB 文件的数据保存的是最近一次的数据库快照， 所以它的数据可能不是最新的， 但只要 RDB 文件本身没有因为 其他问题而出错， 那么还原后的数据库就是一致的 如果当前采用的是AOF存储的， 那么可能事务的内容还未写入到AOF文件， 那么此时肯定是满足一致性的， 如果事务的内容有部分写入到AOF文件中， 那么需要用工具把AOF中事务执行部分成功的指令移除， 这时，移除之后的AOF文件也是满足一致性的 所以，redis事务满足一致性约束

10-4-3：一致性

Redis 是单进程程序，并且它保证在执行事务时， 不会对事务进行中断， 事务可以运行直到执行完所有事务队列中的命令为止。 因此，Redis 的事务是总是带有隔离性的。

10-4-4：持久性

因为事务不过是用队列包裹起了一组 Redis 命令， 并没有提供任何额外的持久性功能， 所以事务的持久性由 Redis 所使用的持久化模式决定 在单纯的内存模式下，事务肯定是不持久的 在 RDB 模式下， 服务器可能在事务执行之后、 RDB 文件更新之前的这段时间失败， 所以 RDB 模式下的 Redis 事务也是不持久的 在 AOF 的"总是 SYNC "模式下， 事务的每条命令在执行成功之后， 都会立即调用 fsync 或 fdatasync 将事务数据写入到 AOF 文件。 但是，这种保存是由后台线程进行的， 主线程不会阻塞直到保存成功， 所以从命令执行成功到数据保存到硬盘之间， 还是有一段非常小的间隔， 所以这种模式下的事务也是不持久的。 其他 AOF 模式也和"总是 SYNC "模式类似， 所以它们都是不持久的。

～～～

11.Redis应用

11-1：为什么Redis 变慢了

1. 使用复杂度高的命令 通过查看慢日志记录， 我们就可以知道在什么时间执行哪些命令比较耗时， 如果你的业务经常使用O(n)以上复杂度的命令， 例如sort、sunion、zunionstore， 或者在执行O(n)命令时操作的数据量比较大， 这些情况下Redis处理数据时就会很耗时。 最好不使用这些复杂度较高的命令， 并且一次不要获取太多的数据， 每次尽量操作少量的数据， 让Redis可以及时处理返回。
2. 存储大key 针对大key的问题， Redis官方在4.0版本推出了lazy-free的机制， 用于异步释放大key的内存， 降低对Redis性能的影响。 即使这样，我们也不建议使用大key， 大key在集群的迁移过程中，也会影响到迁移的性能，
3. 集中过期 平时在使用Redis时没有延时比较大的情况， 但在某个时间点突然出现一波延时， 而且报慢的时间点很有规律， 例如某个整点，或者间隔多久就会发生一次。 如果出现这种情况， 就需要考虑是否存在大量key集中过期的情况。 在集中过期时增加一个随机时间， 把这些需要过期的key的时间打散即可。
4. 实例内存达到上限 有时我们把Redis当做纯缓存使用， 就会给实例设置一个内存上限maxmemory， 然后开启LRU淘汰策略。 当实例的内存达到了maxmemory后， 你会发现之后的每次写入新的数据，有可能变慢了。 导致变慢的原因是， 当Redis内存达到maxmemory后， 每次写入新的数据之前， 必须先踢出

一部分数据， 让内存维持在maxmemory之下。 这个踢出旧数据的逻辑也是需要消耗时间的， 而具体耗时的长短，要取决于配置的淘汰策略

5. fork耗时严重 如果你的Redis开启了自动生成RDB和AOF重写功能， 那么有可能在后台生成RDB和AOF重写时导致Redis的访问延迟增大， 而等这些任务执行完毕后，延迟情况消失。 生成RDB和AOF都需要父进程fork出一个子进程进行数据的持久化， 在fork执行过程中，父进程需要拷贝内存页表给子进程， 如果整个实例内存占用很大， 那么需要拷贝的内存页表会比较耗时， 此过程会消耗大量的CPU资源，在完成fork之前， 整个实例会被阻塞住，无法处理任何请求， 如果此时CPU资源紧张，那么fork的时间会更长， 甚至达到秒级。这会严重影响Redis的性能。 所以使用Redis时建议部署在物理机上，降低fork的影响。
6. 绑定CPU 绑定CPU的Redis，在进行数据持久化时， fork出的子进程，子进程会继承父进程的CPU使用偏好， 而此时子进程会消耗大量的CPU资源进行数据持久化， 子进程会与主进程发生CPU争抢， 这也会导致主进程的CPU资源不足访问延迟增大。 所以在部署Redis进程时， 如果需要开启RDB和AOF重写机制，一定不能进行CPU绑定操作!
7. 开启AOF 当执行AOF文件重写时会因为fork执行耗时 导致Redis延迟增大，除了这个之外， 如果开启AOF机制，设置的策略不合理， 也会导致性能问题。
8. 使用Swap 如果你发现Redis突然变得非常慢， 每次访问的耗时都达到了几百毫秒甚至秒级， 那此时就检查Redis是否使用到了Swap， 这种情况下Redis基本上已经无法提供高性能的服务。 我们知道，操作系统提供了Swap机制， 目的是为了当内存不足时， 可以把一部分内存中的数据换到磁盘上， 以达到对内存使用的缓冲。 但当内存中的数据被换到磁盘上后， 访问这些数据就需要从磁盘中读取， 这个速度要比内存慢太多!
9. 网卡负载过高 Redis也稳定运行了很长时间， 但在某个时间点之后开始， 访问Redis开始变慢了，而且一直持续到现在， 检查一下机器的网卡流量， 是否存在网卡流量被跑满的情况。 网卡负载过高，在网络层和TCP层就会出现数据发送延迟、 数据丢包等情况。Redis的高性能除了内存之外， 就在于网络IO，请求量突增会导致网卡负载变高。

12.锁

12-1：redis加锁的几种实现

redis能用的的加锁命令分表是INCR、SETNX、SET

1. 第一种锁命令INCR 这种加锁的思路是， key 不存在，那么 key 的值会先被初始化为0 ，然后再执行INCR操作进行加一。 然后其它用户在执行 INCR 操作进行加一时，如果返回的数大于 1 ，说明这个锁正在被使用当中。
2. 第二种锁SETNX 这种加锁的思路是，如果 key 不存在，将 key 设置为 value 如果 key 已存在，则 SETNX 不做任何动作
3. 第三种锁SET 之前方法都需要设置 key 过期。如果请求执行因为某些原因意外退出了， 导致创建了锁但是没有删除锁，那么这个锁将一直存在，以至于以后缓存再也得不到更新。 于是乎我们需要给锁加一个过期时间以防不测。 但是借助 Expire 来设置就不是原子性操作了。 所以还可以通过事务来确保原子性， 使用 SET 命令本身已经从版本 2.6.12 开始包含了设置过期时间的功能。

12-2：分布式锁

12-2-1：什么是分布式锁?

当多个进程不在同一个系统中，用分布式锁控制多个进程对资源的访问。

12-2-1-1：分布式锁使用场景

比如说线程A和线程B都共享某个变量X。 如果是单机情况下（单JVM），线程之间共享内存， 只要使用线程锁就可以解决并发问题。 如果是分布式情况下（多JVM）， 线程A和线程B很可能不是在同一JVM中， 这样线程锁就无法起到作用了， 这时候就要用到分布式锁来解决。

12-2-2：各种分布式锁

12-2-2-1：第一个版本

```
tryLock(){
    SETNX Key 1
    EXPIRE Key Seconds
}
release(){
  DELETE Key
}
```

首先给锁加一个过期时间操作是 为了避免应用在服务重启或者异常导致锁无法释放后， 不会出现锁一直无法被释放的情况。 这个方案的一个问题在于每次提交一个Redis请求， 如果执行完第一条命令后应用异常或者重启， 锁将无法过期，一种改善方案就是使用Lua脚本（包含SETNX和EXPIRE两条命令）， 但是如果Redis仅执行了一条命令后crash或者发生主从切换， 依然会出现锁没有过期时间，最终导致无法释放。 针对锁无法释放问题的一个解决方案基于GETSET命令来实现

12-2-2-2：基于GETSET

```
tryLock(){
    NewExpireTime=CurrentTimestamp+ExpireSeconds
    if(SETNX Key NewExpireTime Seconds){
         oldExpireTime = GET(Key)
          if( oldExpireTime < CurrentTimestamp){
              NewExpireTime=CurrentTimestamp+ExpireSeconds
              CurrentExpireTime=GETSET(Key,NewExpireTime)
              if(CurrentExpireTime == oldExpireTime){
                return 1;
              }else{
                return 0;
              }
          }
    }
}
release(){
        DELETE key
    }
```

思路：

1. SETNX(Key,ExpireTime)获取锁
2. 如果获取锁失败，通过GET(Key)返回的时间戳检查锁是否已经过期
3. GETSET(Key,ExpireTime)修改Value为NewExpireTime
4. 检查GETSET返回的旧值，如果等于GET返回的值，则认为获取锁成功 这个版本去掉了EXPIRE命令，改为通过Value时间戳值来判断过期 问题也随之而来， 在锁竞争较高的情况下， 会出现Value不断被覆盖，但是没有一个Client获取到锁 在获取锁的过程中不断的修改原有锁的数据， 设想一种场景C1，C2竞争锁，C1获取到了锁， C2锁执行了GETSET操作修改了C1锁的过期时间， 如果C1没有正确释放锁，锁的过期时间被延长， 其它Client需要等待更久的时间

12-2-2-3：基于SETNX

```
tryLock(){
    SETNX Key 1 Seconds
}
release(){
  DELETE Key
}
```

SETNX增加过期时间参数，这样就解决了两条命令无法保证原子性的问题。 假设某个场景 C1成功获取到了锁， 之后C1因为GC进入等待或者未知原因导致任务执行过长， 最后在锁失效前C1没有主动释放锁 C2在C1的锁超时后获取到锁， 并且开始执行， 这个时候C1和C2都同时在执行， 会因重复执行造成数据不一致等未知情况 C1如果先执行完毕，则会释放C2的锁， 此时可能导致另外一个C3进程获取到了锁 存在问题： 由于C1的停顿导致C1和C2同都获得了锁并且同时在执行， 在业务实现间接要求必须保证幂等性 C1释放了不属于C1的锁

12-2-2-4：第四个版本

```
tryLock(){
    SETNX Key UnixTimestamp Seconds
}
release(){
    EVAL(
      //LuaScript
      if redis.call("get",KEYS[1]) == ARGV[1] then
          return redis.call("del",KEYS[1])
      else
          return 0
      end
    )
}
```

这个方案通过指定Value为时间戳， 并在释放锁的时候检查锁的Value是否为获取锁的Value， 避免了基于GETSET版本中提到的C1释放了C2持有的锁的问题； 另外在释放锁的时候因为涉及到多个Redis操作， 并且考虑到Check And Set 模型的并发问题，所以使用Lua脚本来避免并发问题。 存在问题： 如果在并发极高的场景下， 比如抢红包场景，可能存在UnixTimestamp重复问题， 另外由于不能保证分布式环境下的物理时钟一致性， 也可能存在UnixTimestamp重复问题，只不过极少情况下会遇到。

12-2-2-5：第五个版本

```
tryLock(){
    SET Key UniqId Seconds
}
release(){
    EVAL(
      //LuaScript
      if redis.call("get",KEYS[1]) == ARGV[1] then
          return redis.call("del",KEYS[1])
      else
          return 0
      end
    )
}
```

Redis 2.6.12后SET同样提供了一个NX参数， 等同于SETNX命令，官方文档上提醒后面的版本 有可能去掉SETNX, SETEX, PSETEX,并用SET命令代替， 另外一个优化是使用一个自增的唯一UniqId代替时间戳来 规避之前提到的时钟问题。 这个方案是目前最优的分布式锁方案， 但是如果在Redis集群环境下依然存在问题： 由于Redis集群数据同步为异步， 假设在Master节点获取到锁后未完成数据 同步情况下Master节点crash， 此时在新的Master节点依然可以获取锁， 所以多个Client同时获取到了锁

12-2-2-6：Redlock

由于第五个版本的锁仅在单实例的场景下是安全的， 针对如何实现分布式Redis的锁 目前提出了分布式锁算法Redlock