## 1、数据结构与算法

1）剑指offer

a. 牛客网专题 &&对应扩展题目

b. 对应书籍

2）数据结构与算法书

a. 本科教材

b. 笔记总结

## 2、MySQL

1） 必知必会： 实操

2）《MysQL技术内幕》姜呈尧 看原理，看底层，看优化，做笔记

3）《高性能MySQL》1，2，4， 5，6，7

4） 牛客网专题 + 技术面试题

## 3、浦发银行资料搜索

1） 使用到 Math.power（）时候，考虑用位运算 << 和 >>，更加快速

String.replace()函数

```
Integer.sum(num1, num2);
```

# 一、MySQL必知必会

如果所查询的表 或 where条件筛选后得到的结果集为空，那么聚合函数sum() 或 avg()的返回值为NULL；count()函数的返回值为0

结论： 翻译 复制 搜索

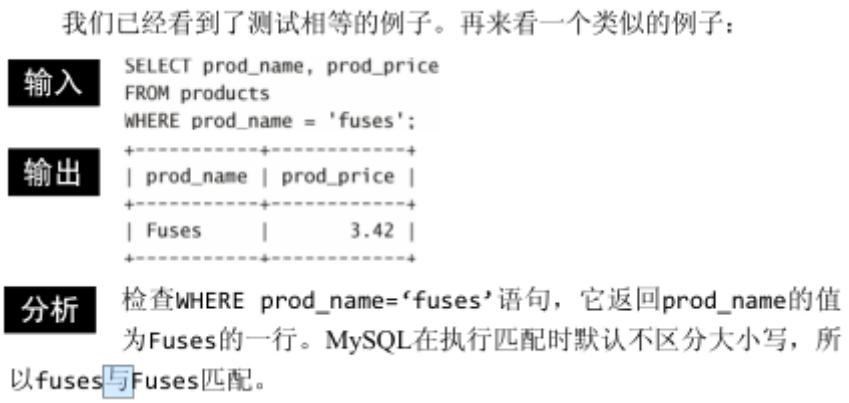
1.in或or在字段有添加索引的情况下，查询很快，两者查询速度没有什么区别；

2.in或or在字段没有添加索引的情况下,所连接的字段越多(1or2or3or4or.....)，or比in的查询效率低很多

MySQL已经成为世界上最受欢迎的数据库管理系统之一。无论是用在小型开发项目上，还是用来构建那些声名显赫的网站，MySQL都证明了自己是个稳定、可靠、快速、可信的系统，足以胜任任何数据存储业务的需要。

优点： 开源免费， 可修改，性能可靠快速， 学习简单。

在字典（dictionary）排序顺序中，A被视为与a相同，这是MySQL（和大多数数据库管理系统）的默认行为。但是，许多数据库管理员能够在需要时改变这种行为（如果你的数据库包含大量外语字符，可能必须这样做）。



我们已经看到了测试相等的例子。再来看一个类似的例子：

输入

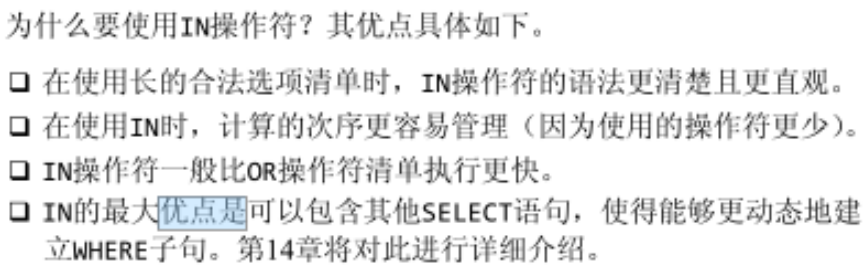
```
SELECT prod_name, prod_price
FROM products
WHERE prod_name = 'fuses';
```

输出

```
+-----------+------------+
| prod_name | prod_price |
+-----------+------------+
| Fuses     |       3.42 |
+-----------+------------+
```

分析 检查WHERE prod_name='fuses'语句，它返回prod_name的值为Fuses的一行。MySQL在执行匹配时默认不区分大小写，所以fuses与Fuses匹配。

SQL（像多数语言一样）在处理 OR 操作符前，优先处理 AND 操作符。



为什么要使用IN操作符？其优点具体如下。

- 在使用长的合法选项清单时，IN操作符的语法更清楚且更直观。
- 在使用IN时，计算的次序更容易管理（因为使用的操作符更少）。
- IN操作符一般比OR操作符清单执行更快。
- IN的最大优点是可以包含其他SELECT语句，使得能够更动态地建立WHERE子句。第14章将对此进行详细介绍。

## 通配符

like % 任意字符 _ 单个字符，不多也不少 。 注意 %是不能匹配 null 的

MySQL的通配符很有用。但这种功能是有代价的：通配符搜索的处理一般要比前面讨论的其他搜索所花时间更长。这里给出一些使用通配符要记住的技巧。

在确实需要使用通配符时，除非绝对有必要，否则不要把它们用在搜索模式的开始处。
把通配符置于搜索模式的开始处，搜索起来是最慢的。
仔细注意通配符的位置。如果放错地方，可能不会返回想要的数据。

## 正则表达式

where 列属性 regexp  '100'

like执行全等值的匹配，LIKE 匹配整个列 。
而regexp执行部分匹配。而 REGEXP是模糊匹配， 在列值内进行匹配，如果被匹配的文本在列值中出现， REGEXP 将会找到它，相应的行将被返回。
但是regexp使用定位符^ $即可进行全等匹配

正则匹配不区分大小写，可使用binary区分  where prod_name regexp  binary 'jetPack'
. 代表任意字符
[1-9] 其中1个　　[^1-9] 否定形式
\\ + 特殊字符  表示转义　. | ，\，[]，()，　\\n 换行　\\t 水平制表　\\v　\\r 回车

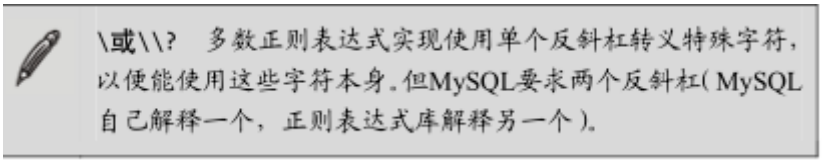



表9-2　字符类

| 类 | 说　明 |
|---|---|
| [:alnum:] | 任意字母和数字（同[a-zA-Z0-9]） |
| [:alpha:] | 任意字符（同[a-zA-Z]） |
| [:blank:] | 空格和制表（同[\\t]） |
| [:cntrl:] | ASCII控制字符（ASCII 0到31和127） |
| [:digit:] | 任意数字（同[0-9]） |
| [:graph:] | 与[:print:]相同，但不包括空格 |
| [:lower:] | 任意小写字母（同[a-z]） |
| [:print:] | 任意可打印字符 |
| [:punct:] | 既不在[:alnum:]又不在[:cntrl:]中的任意字符 |
| [:space:] | 包括空格在内的任意空白字符（同[\\f\\n\\r\\t\\v]） |
| [:upper:] | 任意大写字母（同[A-Z]） |
| [:xdigit:] | 任意十六进制数字（同[a-fA-F0-9]） |

表9-3　重复元字符

| 元　字　符 | 说　　明 |
| --- | --- |
| * | 0个或多个匹配 |
| + | 1个或多个匹配（等于{1,}） |
| ? | 0个或1个匹配（等于{0,1}） |
| {n} | 指定数目的匹配 |
| {n,} | 不少于指定数目的匹配 |
| {n,m} | 匹配数目的范围（m不超过255） |

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '[[:digit:]]{4}'
ORDER BY prod_name;
+---------------+
| prod_name     |
+---------------+
| JetPack 1000 |
| JetPack 2000 |
+---------------+
```

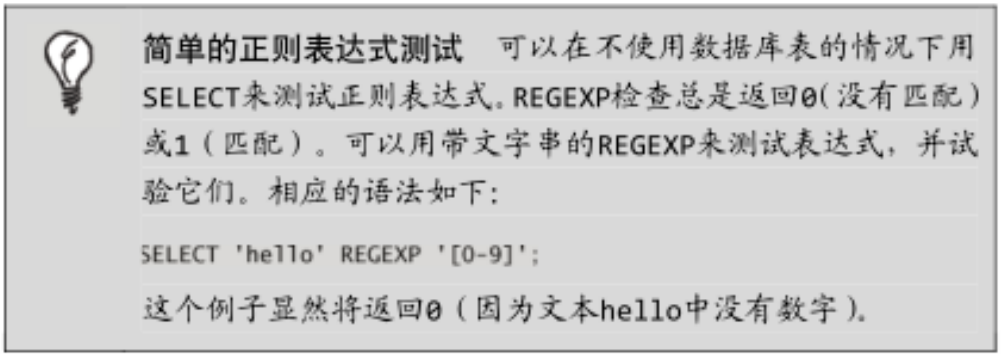
表9-4　定位元字符

| 元　字　符 | 说　　明 |
| --- | --- |
| ^ | 文本的开始 |
| $ | 文本的结尾 |
| [[:<:]] | 词的开始 |
| [[:>:]] | 词的结尾 |

例如，如果你想找出以一个数（包括以小数点开始的数）开始的所

**^的双重用途**　^有两种用法。在集合中（用[和]定义），用它来否定该集合，否则，用来指串的开始处。

**使REGEXP起类似LIKE的作用**　本章前面说过，LIKE和REGEXP的不同在于，LIKE匹配整个串而REGEXP匹配子串。利用定位符，通过用^开始每个表达式，用$结束每个表达式，可以使REGEXP的作用与LIKE一样。




在order by时候， null是最小的

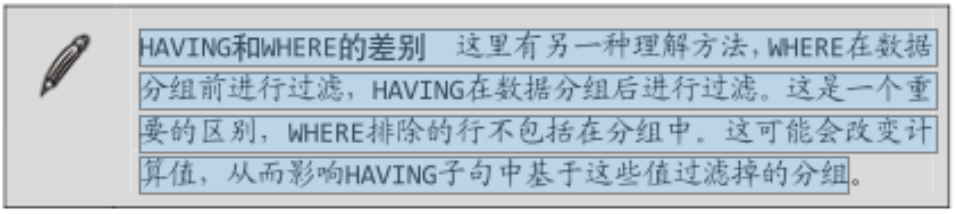
连接字段 select concat (属性1， ‘(’ ， 属性2， ‘)’ )
Concat() 拼接串，即把多个串连接起来形成一个较长的串。
Concat() 需要一个或多个指定的串，各个串之间用逗号分隔。

Rtrim() Ltrim() trim() now() 2019-07-31 09:22:54 Upper() Soundex() Abs() Rand() Sqrt() AVG() MAX() SUM()
NULL 值 SUM() 函数忽略列值为 NULL 的行。

表11-2 常用日期和时间处理函数

| 函 数 | 说 明 |
| --- | --- |
| AddDate() | 增加一个日期（天、周等） |
| AddTime() | 增加一个时间（时、分等） |
| CurDate() | 返回当前日期 |
| CurTime() | 返回当前时间 |
| Date() | 返回日期时间的日期部分 |
| DateDiff() | 计算两个日期之差 |
| Date_Add() | 高度灵活的日期运算函数 |
| Date_Format() | 返回一个格式化的日期或时间串 |
| Day() | 返回一个日期的天数部分 |
| DayOfWeek() | 对于一个日期，返回对应的星期几 |
| Hour() | 返回一个时间的小时部分 |
| Minute() | 返回一个时间的分钟部分 |
| Month() | 返回一个日期的月份部分 |
| Now() | 返回当前日期和时间 |
| Second() | 返回一个时间的秒部分 |
| Time() | 返回一个日期时间的时间部分 |
| Year() | 返回一个日期的年份部分 |

如果分组列中具有 NULL 值，则 NULL 将作为一个分组返回。如果列中有多行 NULL 值，它们将分为一组。




那么，有没有在一条语句中同时使用WHERE和HAVING子句的需要呢？

UNION 指示MySQL执行两条 SELECT 语句，并把输出组
合成单个查询结果集。
列数据类型必须兼容：类型不必完全相同，但必须是DBMS可以
隐含地转换的类型（例如，不同的数值类型或不同的日期类型）。
不适用 union all的话 ， 会自动排重 ， 效率会影响

为了进行全文本搜索，必须索引被搜索的列，而且要随着数据的改变不断地重新索引。
在对表列进行适当设计后，MySQL会自动进行所有的索引和重新索引。
fulltext()

在索引之后，使用两个函数Match()和Against()执行全文本搜索，其中Match()指定被搜索的列，Against()指定要使用的搜索表达式。

下面举一个例子：

输入

```
SELECT note_text
FROM productnotes
WHERE Match(note_text) Against('rabbit');
```



as rank

TRUNCATE TABLE
delete from table ； update XX set 一定要有where不然会容易修改所有行

alter table add/drop column XX. 一般先备份一份，再修改表

检验包含所需数据的新表；
- 重命名旧表（如果确定，可以删除它）；
- 用旧表原来的名字重命名新表； RENAME TABLE to
- 根据需要，重新创建触发器、存储过程、索引和外键。

## 视图

视图是虚拟的表。与包含数据的表不一样，视图只包含使用时动态检索数据的查询。视图本身不包含数据，因此它们返回的数据是从其他表中检索出来的。在添加或更改这些表中的数据时，视图将返回改变过的数据。可以隐藏复杂的SQL语句 ，更可以仅展示部分的表数据

我们已经看到了视图应用的一个例子。下面是视图的一些常见应用。

- 重用SQL语句。
- 简化复杂的SQL操作。在编写查询后，可以方便地重用它而不必知道它的基本查询细节。
- 使用表的组成部分而不是整个表。
- 保护数据。可以给用户授予表的特定部分的访问权限而不是整个表的访问权限。
- 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。

视图不能索引，也不能有关联的触发器或默认值。将视图用于检索 一般，应该将视图用于检索（SELECT 语句）而不用于更新（INSERT 、 UPDATE 和 DELETE ）。

**性能问题** 因为视图不包含数据，所以每次使用视图时，都必须处理查询执行时所需的任一个检索。如果你用多个联结和过滤创建了复杂的视图或者嵌套了视图，可能会发现性能下降得很厉害。因此，在部署使用了大量视图的应用前，应该进行测试。

- 视图用CREATE VIEW语句来创建。
- 使用SHOW CREATE VIEW viewname；来查看创建视图的语句。
- 用DROP删除视图，其语法为DROP VIEW viewname;。
- 更新视图时，可以先用DROP再用CREATE，也可以直接用CREATE OR REPLACE VIEW。如果要更新的视图不存在，则第2条更新语句会创建一个视图；如果要更新的视图存在，则第2条更新语句会替换原有视图。

通常，视图是可更新的（即，可以对它们使用INSERT、UPDATE和DELETE）。更新一个视图将更新其基表（可以回忆一下，视图本身没有数据）。如果你对视图增加或删除行，实际上是对其基表增加或删除行。



但是，并非所有视图都是可更新的。基本上可以说，如果MySQL不能正确地确定被更新的基数据，则不允许更新（包括插入和删除）。这实际上意味着，如果视图定义中有以下操作，则不能进行视图的更新：

- 分组（使用GROUP BY和HAVING）；
- 联结；
- 子查询；
- 并；
- 聚集函数（Min()、Count()、Sum()等）；

- DISTINCT；
- 导出（计算）列。

换句话说，本章许多例子中的视图都是不可更新的。这听上去好像是一个严重的限制，但实际上不是，因为视图主要用于数据检索。

KILL命令终结某个特定的进程（使用这个命令需要作为管理员登录）。

- 总是有不止一种方法编写同一条SELECT语句。应该试验联结、并、子查询等，找出最佳的方法。
- 使用EXPLAIN语句让MySQL解释它将如何执行一条SELECT语句。
- 一般来说，存储过程执行得比一条一条地执行其中的各条MySQL语句快。
- 应该总是使用正确的数据类型。
- 决不要检索比需求还要多的数据。换言之，不要用SELECT *（除非你真正需要每个列）。
- 有的操作（包括INSERT）支持一个可选的DELAYED关键字，如果使用它，将把控制立即返回给调用程序，并且一旦有可能就实际执行该操作。
- 在导入数据时，应该关闭自动提交。你可能还想删除索引（包括FULLTEXT索引），然后在导入完成后再重建它们。
- 必须索引数据库表以改善数据检索的性能。确定索引什么不是一件微不足道的任务，需要分析使用的SELECT语句以找出重复的WHERE和ORDER BY子句。如果一个简单的WHERE子句返回结果所花的时间太长，则可以断定其中使用的列（或几个列）就是需要索引的对象。
- 你的SELECT语句中有一系列复杂的OR条件吗？通过使用多条SELECT语句和连接它们的UNION语句，你能看到极大的性能改进。
- 索引改善数据检索的性能，但损害数据插入、删除和更新的性能。如果你有一些表，它们收集数据且不经常被搜索，则在有必要之前不要索引它们。（索引可根据需要添加和删除。）
- LIKE很慢。一般来说，最好是使用FULLTEXT而不是LIKE。
- 数据库是不断变化的实体。一组优化良好的表一会儿后可能就面目全非了。由于表的使用和内容的更改，理想的优化和配置也会改变。
- 最重要的规则就是，每条规则在某些条件下都会被打破。

## 储存过程：只需编译一次

什么是存储过程？为什么要用？如何使用？注意事项

what ？ 若干条语句的集合 ， 可以类比成编程中的一个函数，可以有参数可以无参数，仅一次编译

why ？ 经常一个完整的操作，需要多条语句才能执行，这就可以封装成一个存储过程，对外开放，内部细节屏蔽，简化复杂的操作流程，方便使用（需要执行的步骤越多，出错的可能性就越大。防止错误保证了数据的一致性。）

how ？ call调用 ， 注意分隔符

```
create procedure  xx()
    begin
..... ;#目标过程
```

end

drop proceduce XX名
DROP PROCEDURE IF EXISTS。

notice ? 编写比较复杂，需要更高的经验及技术水平，出错了不好调试，最好设置一定的权限才能编写储存过程，一般开发更多的是使用，而编写则需要特别小心，储存过程中任意一条语句的表格发生改变则可能使得整个储存过程不可使用，或者出现意想不到的结果。

## 游标

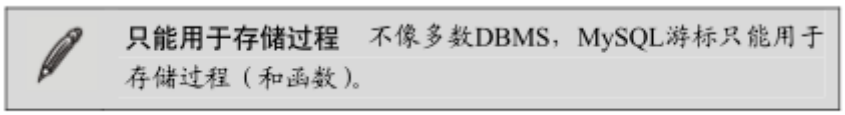
MySQL5之后添加的

what ?

游标（cursor）是一个存储在MySQL服务器上的数据库查询，它不是一条 SELECT 语句，而是被该语句检索出来的结果集。在存储了游标之后，应用程序可以根据需要滚动或浏览其中的数据。游标主要用于交互式应用，其中用户需要滚动屏幕上的数据，并对数据进行浏览或做出更改。

why ?

需要在检索出来的行中前进或后退一行或多行




how ? 使用之前定义， open， close , DECLARE

notice ? 只能用于存储过程

CLOSE 释放游标使用的所有内部内存和资源，因此在每个游标不再需要时都应该关闭。

## 触发器

what ? 某条语句在某个时间发生之后，自动执行，这就是触发器

why? 自动触发。 比如每订购商品一次，存库表中剩余量都自动减一；每当增加一个顾客到某个数据库表时，都检查其电话号码格式是否正确，州的缩写是否为大写； 应该用触发器来保证数据的一致性（大小写、格式等）

how? create triggger , drop ,

notice ?在delete ,insert , update之前之后

## 事务

what? 一个原子操作集合

why?事务处理（transaction processing）可以用来维护数据库的完整性，它保证成批的MySQL操作要么完全执行，要么完全不执行。

how? start transaction , rollback , commit , savepoint ,set autocommit = 0;

notice? 只有InnoDB支持

## 二、Redis

1)《Redis实战》 + 《Redis设计与实现》 + 《Redis开发与运维》 + 菜鸟教程

2）面试题目， 笔试题目， 注意多机版的Redis的高级特性

A man of hte people Redis之父： Salvatore Sanfilippo ,意大利程序员

### 1. 简介

远程字典服务器， 开源免费， 使用C语言编写的， 基于内存亦可持久化的非关系key-value数据库 。其主要value可以有String， hash， list , set ,sorted set五大部件， 也支持数据备份。默认6379号端口

特点：

1）速度快。 基于内存， 读写速度可达10W/s .

2) 丰富的数据结构。 String ,list， hash ,set， zset， 发布订阅， key生命周期

3) 原子特性 。单进程，所有操作都是原子类型的， 也支持多个操作的事务 multi exec

4) 支持持久化到磁盘的操作

config get * ; // 获取配置的命令

### 2. 数据类型： 数据结构与对象

1） String 类型 。 set del

Redis的基本类型， 是二进制安全的，可以存任何数据， 比如图片，序列化对象。 < 512M

自行构造了简单动态字符串SDS，内含长度， 使用空间， 未使用空间长度， 方便内存的可弹性变化

优点： 二进制安全， 存任何<=512M数据； 内存空间弹性变化,减少覆盖和重新分配内存的次数； 获取长度时间为O(1)

Redis 作为数据库， 经常被用于速度要求严苛、数据被频繁修改的场合， 如果每次修改字符串的长度都需要执行一次内存重分配的话， 那么光是执行内存重分配的时间就会占去修改字符串所用时间的一大部分， 如果这种修改频繁地发生的话， 可能还会对性能造成影响。

- 如果对 SDS 进行修改之后， SDS 的长度（也即是 `len` 属性的值）将小于 `1 MB` ， 那么程序分配和 `len` 属性同样大小的未使用空间， 这时 SDS `len` 属性的值将和 `free` 属性的值相同。 举个例子， 如果进行修改之后， SDS 的 `len` 将变成 `13` 字节， 那么程序也会分配 `13` 字节的未使用空间， SDS 的 `buf` 数组的实际长度将变成 `13 + 13 + 1 = 27` 字节（额外的一字节用于保存空字符）。
- 如果对 SDS 进行修改之后， SDS 的长度将大于等于 `1 MB` ， 那么程序会分配 `1 MB` 的未使用空间。 举个例子， 如果进行修改之后， SDS 的 `len` 将变成 `30 MB` ， 那么程序会分配 `1 MB` 的未使用空间， SDS 的 `buf` 数组的实际长度将为 `30 MB + 1 MB + 1 byte` 。

SDS 将连续增长 `N` 次字符串所需的内存重分配次数从必定 `N` 次降低为最多 `N` 次。

惰性空间释放用于优化 SDS 的字符串缩短操作： 当 SDS 的 API 需要缩短 SDS 保存的字符串时， 程序并不立即使用内存重分配来回收缩短后多出来的字节， 而是使用 `free` 属性将这些字节的数量记录起来， 并等待将来使用。

与此同时， SDS 也提供了相应的 API ， 让我们可以在有需要时， 真正地释放 SDS 里面的未使用空间， 所以不用担心惰性空间释放策略会造成内存浪费。sdsclear

可以使用部分String.h中的函数方法

除了用来保存数据库中的字符串值之外， SDS 还被用作缓冲区（buffer）： AOF 模块中的 AOF 缓冲区， 以及客户端状态中的输入缓冲区， 都是由 SDS 实现的

2) hash类型 hmset hget

适合于存储对象 ,可储存2^32-1个键值对， 约为40亿个

当一个哈希键包含的键值对比较多， 又或者键值对中的元素都是比较长的字符串时， Redis 就会使用字典作为哈希键的底层实现

先计算出key的hash值，然后 &（n-1）获得索引，放入索引位置，有冲突则拉链法（头插法）

MurmurHash 算法最初由 Austin Appleby 于 2008 年发明， 这种算法的优点在于， 即使输入的键是有规律的， 算法仍能给出一个很好的随机分布性， 并且算法的计算速度也非常快。

当以下条件中的任意一个被满足时， 程序会自动开始对哈希表执行扩展操作：

1. 服务器目前没有在执行 BGSAVE 命令或者 BGREWRITEAOF 命令， 并且哈希表的负载因子大于等于 `1` ；
2. 服务器目前正在执行 BGSAVE 命令或者 BGREWRITEAOF 命令， 并且哈希表的负载因子大于等于 `5` ；

另一方面， 当哈希表的负载因子小于 `0.1` 时， 程序自动开始对哈希表执行收缩操作。

rehash： 扩大/缩小为最接近的2^n的大小，有两个hash表ht[0] ,ht[1],做rehash时候用，有点类似垃圾回收算法的中的from 和to区间之间的的copy算法。rehash 动作并不是一次性、集中式地完成的， 而是分多次、渐进式地完成的。

渐进式 rehash 执行期间的哈希表操作

因为在进行渐进式 rehash 的过程中， 字典会同时使用 `ht[0]` 和 `ht[1]` 两个哈希表， 所以在渐进式 rehash 进行期间， 字典的删除（delete）、查找（find）、更新（update）等操作会在两个哈希表上进行： 比如说， 要在字典里面查找一个键的话， 程序会先在 `ht[0]` 里面进行查找， 如果没找到的话， 就会继续到 `ht[1]` 里面进行查找， 诸如此类。

另外， 在渐进式 rehash 执行期间， 新添加到字典的键值对一律会被保存到 `ht[1]` 里面， 而 `ht[0]` 则不再进行任何添加操作： 这一措施保证了 `ht[0]` 包含的键值对数量会只减不增， 并随着 rehash 操作的执行而最终变成空表。

3）列表

lpush　lrange　可储存2^32-1个

当一个列表键包含了数量比较多的元素， 又或者列表中包含的元素都是比较长的字符串时， Redis 就会使用链表作为列表键的底层实现。 双向的，无环，带有首尾指针，有记录个数的计数器

4) 集合　sadd smembers

是String的无序集合，通过hash实现的，所以增删查时间都是O(1)，可储存2^32-1个

5) zset　　zadd　zrange

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

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。zset的成员是唯一的,但分数(score)却可以重复。

跳跃表：

在大部分情况下， 跳跃表的效率可以和平衡树相媲美， 并且因为跳跃表的实现比平衡树要来得更为简单， 所以有不少程序都使用跳跃表来代替平衡树。

Redis 使用跳跃表作为有序集合键的底层实现之一： 如果一个有序集合包含的元素数量比较多， 又或者有序集合中元素的成员（member）是比较长的字符串时， Redis 就会使用跳跃表.Redis 只在两个地方用到了跳跃表， 一个是实现有序集合键， 另一个是在集群节点中用作内部数据结构， 除此之外， 跳跃表在 Redis 里面没有其他用途。来作为有序集合键的底层实现。

时间最好O(lg n),最坏O(n)， 类似二分查找过程，跳跃高度在1~32之间

- 跳跃表是有序集合的底层实现之一， 除此之外它在 Redis 中没有其他应用。
- Redis 的跳跃表实现由 `zskiplist` 和 `zskiplistNode` 两个结构组成， 其中 `zskiplist` 用于保存跳跃表信息（比如表头节点、表尾节点、长度）， 而 `zskiplistNode` 则用于表示跳跃表节点。
- 每个跳跃表节点的层高都是 `1` 至 `32` 之间的随机数。
- 在同一个跳跃表中， 多个节点可以包含相同的分值， 但每个节点的成员对象必须是唯一的。
- 跳跃表中的节点按照分值大小进行排序， 当分值相同时， 节点按照成员对象的大小进行排序。

整数集合：

当一个集合只包含整数值元素， 并且这个集合的元素数量不多时， Redis 就会使用整数集合作为集合键的底层实现。 只能升级不能降级 ， 提高灵活性，尽量节约内存

`int16_t` 、 `int32_t` 、 `int64_t`

压缩列表：

连锁更新问题。

- 压缩列表是一种为节约内存而开发的顺序型数据结构。
- 压缩列表被用作列表键和哈希键的底层实现之一。
- 压缩列表可以包含多个节点，每个节点可以保存一个字节数组或者整数值。
- 添加新节点到压缩列表， 或者从压缩列表中删除节点， 可能会引发连锁更新操作， 但这种操作出现的几率并不高。

## 3.命令集合

Redis 会共享值为 `0` 到 `9999` 的字符串对象

key ==> exists key , del , expire , persist ,ttl ,type, randomkey

String ==> set , get, getrange , getset , strlen , incr key num, decr , append key

hash ==> hset hgetall , hdel, hexists ,hget, hincr ,hdecr, hkeys ,hlen key, hvals

List ==> blpop , 阻塞弹出 , lindex, llen , lpop , lrem .

set ==> sadd , scard ,sdif（差集） , sinter(交集) , sunion , sdidstore , spop ,srem

zset ==> zadd , zcard , zincrby , zinterstore ,zinter,zunion ,zrank , zreverank,zrem

Hyperloglog ==> pfadd pfcount pfmerge(多个hyperloglog合并为一个) 0.8%误差

有一定误差的 。类似布隆过滤器

每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基 数

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

type , Object encoding

HyperLogLog 这个数据结构不是免费的，不是说使用这个数据结构要花钱，它需要占据一定 12k 的存储空间，所以它不适合统计单个用户相关的数据。如果你的用户上亿，可以算算，这个空间成本是非常惊人的。但是相比 set 存储方案，HyperLogLog 所使用的空间那真是可以使用千斤对比四两来形容了。

不过你也不必过于担心，因为 Redis 对 HyperLogLog 的存储进行了优化，在计数比较小时，它的存储空间采用稀疏矩阵存储，空间占用很小，仅仅在计数慢慢变大，稀疏矩阵占用空间渐渐超过了阈值时才会一次性转变成稠密矩阵，才会占用 12k 的空间。

### pf 的内存占用为什么是 12k?

我们在上面的算法中使用了 1024 个桶进行独立计数，不过在 Redis 的 HyperLogLog 实现中用到的是 16384 个桶，也就是 2^14，每个桶的 maxbits 需要 6 个 bits 来存储，最大可以表示 maxbits=63，于是总共占用内存就是2^14 * 6 / 8 = 12k字节。

发布订阅 ==》subscirbe XX频道 unsubcribe , publish

事务　==》 discard（取消事务，放弃执行所有的命令）multi ....exec ， watch（监控若干个key）

中间某条指令的失败不会导致前面已做指令的回滚，也不会造成后续的指令不做。

## Redis 事务

Redis 事务可以一次执行多个命令， 并且带有以下三个重要的保证：

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

一个事务从开始到执行会经历以下三个阶段：

- 开始事务。
- 命令入队。
- 执行事务。

## 4、脚本及其他连接操作

auth password . 验证密码是否正确
echo "打印字符串"
ping　==》 pong 是否能连接服务器
quit == > 关闭连接
select index ==> 选择数据库

info　查看统计信息： CPU信息， 客户端连接数， 进程id ,服务器状态

client kill XX 关闭某个客户端的连接
bgsave . 后台异步持久化
time
dbsize　数据库的key的数目
flushall　删除数据库所有的key
flushdb 删除当前数据库的所有key

执行lua脚本  eval

## 5、其他语言使用Redis脚本

引入一个外部jar包即可　　jedis.jar

数据备份与恢复：一般是在dump.rdb文件中　save　==》　bgsave

redis安全：客户端连接到 redis 服务就需要密码验证　config set requirepass "" ； auth password

性能测试：`redis-benchmark -n 10000  -q`

Redis客户端连接：

### Redis 客户端连接

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

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

最大客户端连接数可以自行修改，maxclients参数，在启动server时候指定 `redis-server --maxclients 100000`

Redis管道技术：

### Redis 管道技术

Redis是一种基于客户端-服务端模型以及请求/响应协议的TCP服务。这意味着通常情况下一个请求会遵循以下步骤:

- 客户端向服务端发送一个查询请求，并监听Socket返回，通常是以阻塞模式，等待服务端响应。
- 服务端处理命令，并将结果返回给客户端。

### Redis 管道技术

Redis 管道技术可以在服务端未响应时，客户端可以继续向服务端发送请求，并最终一次性读取所有服务端的响应。

管道技术最显著的优势是提高了 redis 服务的性能。

Redis分区：

# Redis 分区

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

## 分区的优势

- 通过利用多台计算机内存的和值，允许我们构造更大的数据库。
- 通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。

## 分区的不足

redis的一些特性在分区方面表现的不是很好：

- 涉及多个key的操作通常是不被支持的。举例来说，当两个set映射到不同的redis实例上时，你就不能对这两个set执行交集操作。
- 涉及多个key的redis事务不能使用。
- 当使用分区时，数据处理较为复杂，比如你需要处理多个rdb/aof文件，并且从多个实例和主机备份持久化文件。
- 增加或删除容量也比较复杂。redis集群大多数支持在运行时增加、删除节点的透明数据平衡的能力，但是类似于客户端分区、代理等其他系统则不支持这项特性。然而，一种叫做presharding的技术对此是有帮助的。

## 哈希分区

另外一种分区方法是hash分区。这对任何key都适用，也无需是object_name:这种形式，像下面描述的一样简单：

- 用一个hash函数将key转换为一个数字，比如使用crc32 hash函数。对key foobar执行crc32(foobar)会输出类似93024922的整数。
- 对这个整数取模，将其转化为0-3之间的数字，就可以将这个整数映射到4个Redis实例中的一个了。93024922 % 4 = 2，就是说key foobar应该被存到R2实例中。注意：取模操作是取除的余数，通常在多种编程语言中用%操作符实现。

使用Lua编写redis脚本。

好处：

在redis的官网上洋洋洒洒的大概提供了200多个命令，貌似看起来很多，但是这些都是别人预先给你定义好的，但你却不能按照自己的意图进行定制，

所以是不是感觉自己还是有一种被束缚的感觉，有这个感觉就对了。。。

说来也巧，redis的大老板给了你解决这种问题的方法，那就是Lua脚本，而且redis的最新版本也支持Lua Script debug，这应该也是未来Redis的一

- **Lua脚本：**Lua是一个高效的轻量级脚本语言，用标准C语言编写并以源代码形式开放， 其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。
- **使用脚本的好处：**

1. 减少网络开销，在Lua脚本中可以把多个命令放在同一个脚本中运行。
2. 原子操作，redis会将整个脚本作为一个整体执行，中间不会被其他命令插入。换句话说，编写脚本的过程中无需担心会出现竞态条件。
3. 复用性，客户端发送的脚本会永远存储在redis中，这意味着其他客户端可以复用这一脚本来完成同样的逻辑 。

原子操作也可以在服务器上用锁操作 ， 但是更为方便的是在redis端操作，lua脚本能够保证redis执行的原子性（当然如果lua脚本报错的话，无法回滚掉已执行的部分代码的）

1）直接使用 eval命令

`[EVAL] [脚本内容] [key参数的数量] [key …] [arg …]`

可以通过key和arg这两个参数向脚本中传递数据，他们的值可以在脚本中分别使用KEYS和ARGV 这两个类型的全局变量访问。 注意lua的下标是从1开始的

```
redis.call('get/set');  //直接使用redis命令
```

2）在命令行中使用

```
redis-cli  --eval lua_file key1 key2 , arg1 arg2 arg3
```

```
127.0.0.1:6379> eval "redis.call('SET', KEYS[1], ARGV[1]);redis.call('EXPIRE', KEYS[1], ARGV[2]); return 1;" 1 test 10 60
```

## Redis设计与实现（第2版） ： 多机操作装逼指南

- 字符串（string）、散列（hash）、列表（list）、集合（set）和有序集合（sorted set）这五种类型的键的底层实现数据结构。
- Redis 的对象处理机制以及数据库的实现原理。
- 事务实现原理。
- 订阅与发布实现原理。
- Lua 脚本功能的实现原理。
- `SORT` 命令的实现原理。
- `BITOP` 、 `BITCOUNT` 等二进制位处理命令的实现原理。
- 慢查询日志的实现原理。
- RDB 持久化和 AOF 持久化的实现原理。
- Redis 事件处理器的实现原理。
- Redis 服务器和客户端的实现原理。
- 复制（replication）、Sentinel 和集群（cluster）这三个多机功能的实现原理。

1、 redis的集合操作方便得多， 简单， 也常用于做缓存

# Redis单机操作

键空间： flushdb randomkey dbsize , exist , rename, keys ,expire

`redisDb` 结构的`dict` 字典保存了数据库中的所有键值对， 我们将这个字典称为键空间（key space）

当使用 Redis 命令对数据库进行读写时， 服务器不仅会对键空间执行指定的读写操作， 还会执行一些额外的维护操作:

1) 在读取一个键之后， 服务器会更新键的 LRU （最后一次使用）时间， 这个值可以用于计算键的闲置时间， 使用命令 OBJECT idletime 命令可以查看键 `key` 的闲置时间。

2) 如果服务器在读取一个键时， 发现该键已经过期， 那么服务器会先删除这个过期键， 然后才执行余下的其他操作，

3) 如果有客户端使用 WATCH 命令监视了某个键， 那么服务器在对被监视的键进行修改之后， 会将这个键标记为脏（dirty）， 从而让事务程序注意到这个键已经被修改过，

## 文件事件:

- 文件事件处理器使用 I/O 多路复用（multiplexing）程序来同时监听多个套接字， 并根据套接字目前执行的任务来为套接字关联不同的事件处理器。
- 当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时， 与操作相对应的文件事件就会产生， 这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。

虽然文件事件处理器以单线程方式运行， 但通过使用 I/O 多路复用程序来监听多个套接字， 文件事件处理器既实现了高性能的网络通信模型， 又可以很好地与 Redis 服务器中其他同样以单线程方式运行的模块进行对接， 这保持了 Redis 内部单线程设计的简单性。

## 持久化： RDB AOF

### 1）RDB：

- RDB 文件用于保存和还原 Redis 服务器所有数据库中的所有键值对数据。
- SAVE 命令由服务器进程直接执行保存操作，所以该命令会阻塞服务器。
- BGSAVE 命令由子进程执行保存操作，所以该命令不会阻塞服务器。
- 服务器状态中会保存所有用 `save` 选项设置的保存条件，当任意一个保存条件被满足时，服务器会自动执行 BGSAVE 命令。
- RDB 文件是一个经过压缩的二进制文件，由多个部分组成。
- 对于不同类型的键值对， RDB 文件会使用不同的方式来保存它们。

### 2）AOF：

- AOF 文件通过保存所有修改数据库的写命令请求来记录服务器的数据库状态。
- AOF 文件中的所有命令都以 Redis 命令请求协议的格式保存。
- 命令请求会先保存到 AOF 缓冲区里面， 之后再定期写入并同步到 AOF 文件。
- `appendfsync` 选项的不同值对 AOF 持久化功能的安全性、以及 Redis 服务器的性能有很大的影响。
- 服务器只要载入并重新执行保存在 AOF 文件中的命令， 就可以还原数据库本来的状态。
- AOF 重写可以产生一个新的 AOF 文件， 这个新的 AOF 文件和原有的 AOF 文件所保存的数据库状态一样， 但体积更小。
- AOF 重写是一个有歧义的名字， 该功能是通过读取数据库中的键值对来实现的， 程序无须对现有 AOF 文件进行任何读入、分析或者写入操作。
- 在执行 BGREWRITEAOF 命令时， Redis 服务器会维护一个 AOF 重写缓冲区， 该缓冲区会在子进程创建新 AOF 文件的期间， 记录服务器执行的所有写命令。 当子进程完成创建新 AOF 文件的工作之后， 服务器会将重写缓冲区中的所有内容追加到新 AOF 文件的末尾， 使得新旧两个 AOF 文件所保存的数据库状态一致。 最后， 服务器用新的 AOF 文件替换旧的 AOF 文件， 以此来完成 AOF 文件重写操作。

## Redis多机操作：

主从：读写分离，备份
哨兵：监控，自动转移，选主
集群：数据 hash 分片，同时包含主从及哨兵特性

### 1. 复制功能

Redis 的复制功能分为同步（sync）和命令传播（command propagate）两个操作：

- 其中， 同步操作用于将从服务器的数据库状态更新至主服务器当前所处的数据库状态。
- 而命令传播操作则用于在主服务器的数据库状态被修改， 导致主从服务器的数据库状态出现不一致时， 让主从服务器的数据库重新回到一致状态。

同步

当客户端向从服务器发送 SLAVEOF 命令， 要求从服务器复制主服务器时， 从服务器首先需要执行同步操作， 也即是，将从服务器的数据库状态更新至主服务器当前所处的数据库状态。

从服务器对主服务器的同步操作需要通过向主服务器发送 SYNC 命令来完成， 以下是 SYNC 命令的执行步骤：

1. 从服务器向主服务器发送 SYNC 命令。
2. 收到 SYNC 命令的主服务器执行 BGSAVE 命令， 在后台生成一个 RDB 文件， 并使用一个缓冲区记录从现在开始执行的所有写命令。
3. 当主服务器的 BGSAVE 命令执行完毕时， 主服务器会将 BGSAVE 命令生成的 RDB 文件发送给从服务器， 从服务器接收并载入这个 RDB 文件， 将自己的数据库状态更新至主服务器执行 BGSAVE 命令时的数据库状态。
4. 主服务器将记录在缓冲区里面的所有写命令发送给从服务器， 从服务器执行这些写命令， 将自己的数据库状态更新至主服务器数据库当前所处的状态。

## 2. 哨兵Sentinel

## 3. 集群

CLUSTER NODES

CLUSTER MEET

- 节点通过握手来将其他节点添加到自己所处的集群当中。
- 集群中的 `16384` 个槽可以分别指派给集群中的各个节点， 每个节点都会记录哪些槽指派给了自己， 而哪些槽又被指派给了其他节点。
- 节点在接到一个命令请求时， 会先检查这个命令请求要处理的键所在的槽是否由自己负责， 如果不是的话， 节点将向客户端返回一个 `MOVED` 错误， `MOVED` 错误携带的信息可以指引客户端转向至正在负责相关槽的节点。
- 对 Redis 集群的重新分片工作是由客户端执行的， 重新分片的关键是将属于某个槽的所有键值对从一个节点转移至另一个节点。
- 如果节点 A 正在迁移槽 `i` 至节点 B ， 那么当节点 A 没能在自己的数据库中找到命令指定的数据库键时， 节点 A 会向客户端返回一个 `ASK` 错误， 指引客户端到节点 B 继续查找指定的数据库键。
- `MOVED` 错误表示槽的负责权已经从一个节点转移到了另一个节点， 而 `ASK` 错误只是两个节点在迁移槽的过程中使用的一种临时措施。
- 集群里的从节点用于复制主节点， 并在主节点下线时， 代替主节点继续处理命令请求。
- 集群中的节点通过发送和接收消息来进行通讯， 常见的消息包括 `MEET` 、 `PING` 、 `PONG` 、 `PUBLISH` 、 `FAIL` 五种。

# Redis特色功能：

## 1、发布和订阅功能：

Redis 将所有频道的订阅关系都保存在服务器状态的 `pubsub_channels` 字典里面，这个字典的键是某个被订阅的频道， 而键的值则是一个链表， 链表里面记录了所有订阅这个频道的客户端，尾插法插入订阅者

SUBSCRIBE 命令　　UNSUBSCRIBE 命令

- 服务器状态在 `pubsub_channels` 字典保存了所有频道的订阅关系： SUBSCRIBE 命令负责将客户端和被订阅的频道关联到这个字典里面， 而 UNSUBSCRIBE 命令则负责解除客户端和被退订频道之间的关联。
- 服务器状态在 `pubsub_patterns` 链表保存了所有模式的订阅关系： PSUBSCRIBE 命令负责将客户端和被订阅的模式记录到这个链表中， 而UNSUBSCRIBE 命令则负责移除客户端和被退订模式在链表中的记录。
- PUBLISH 命令通过访问 `pubsub_channels` 字典来向频道的所有订阅者发送消息， 通过访问 `pubsub_patterns` 链表来向所有匹配频道的模式的订阅者发送消息。
- PUBSUB 命令的三个子命令都是通过读取 `pubsub_channels` 字典和 `pubsub_patterns` 链表中的信息来实现的。

2、事务

事务开始： multi

==》　命令入队，先进的先执行

==》　事务执行 exec（当一个处于事务状态的客户端向服务器发送 EXEC 命令时， 这个 EXEC 命令将立即被服务器执行： 服务器会遍历这个客户端的事务队列，执行队列中保存的所有命令， 最后将执行命令所得的结果全部返回给客户端。）

discard 取消事务

watch key 监视key ,如果再事务执行之前key发生改变， 事务将被打断

notice: 中间出现错误不会回滚， 只能保重ACID中ACI。

3、Lua脚本

Redis 在服务器内嵌了一个 Lua 环境（environment）， 并对这个 Lua 环境进行了一系列修改， 从而确保这个 Lua 环境可以满足 Redis 服务器的需要。

1. 创建一个基础的 Lua 环境， 之后的所有修改都是针对这个环境进行的。
2. 载入多个函数库到 Lua 环境里面， 让 Lua 脚本可以使用这些函数库来进行数据操作。
3. 创建全局表格 `redis` ， 这个表格包含了对 Redis 进行操作的函数， 比如用于在 Lua 脚本中执行 Redis 命令的 `redis.call` 函数。
4. 使用 Redis 自制的随机函数来替换 Lua 原有的带有副作用的随机函数， 从而避免在脚本中引入副作用。
5. 创建排序辅助函数， Lua 环境使用这个辅佐函数来对一部分 Redis 命令的结果进行排序， 从而消除这些命令的不确定性。
6. 创建 `redis.pcall` 函数的错误报告辅助函数， 这个函数可以提供更详细的出错信息。
7. 对 Lua 环境里面的全局环境进行保护， 防止用户在执行 Lua 脚本的过程中， 将额外的全局变量添加到了 Lua 环境里面。
8. 将完成修改的 Lua 环境保存到服务器状态的 `lua` 属性里面， 等待执行服务器传来的 Lua 脚本。

eval

evalsha : 内容长的脚本　　script load

在Lua脚本中也可以使用 redis.pcall 来调用Redis的命令，两个命令的区别在于如果 redis.call 命令执行失败，那么Lua脚本结束并且返回错误；如果 redis.pcall 命令执行失败，会忽略错误继续执行脚本。

4、排序

lpush lt 1 3 5 2;　　sort　lt　排序必须是double

5、getbit命令

在计算机里所有的数据都是以二进制的形式存储的，每一个非中文字符占一个字节（Byte），中文字符占两个字节，而一个字节又是占8bit。

01100010　01100001　01110010

setbit

6、慢日志查询

Redis 的慢查询日志功能用于记录执行时间超过给定时长的命令请求， 用户可以通过这个功能产生的日志来监视和优化查询速度。

CONFIG SET slowlog-log-slower-than 0　：超过阈值则记录

CONFIG SET slowlog-max-len 5 ： 超过最大条数，则删除最早进来的那条

SLOWLOG GET ：

```
# 日志的唯一标识符（uid）
# 命令执行时的 UNIX 时间戳
# 命令执行的时长，以微秒计算
# 命令以及命令参数
```

- Redis 的慢查询日志功能用于记录执行时间超过指定时长的命令。
- Redis 服务器将所有的慢查询日志保存在服务器状态的 `slowlog` 链表中，每个链表节点都包含一个 `slowlogEntry` 结构，每个 `slowlogEntry` 结构代表一条慢查询日志。
- 打印和删除慢查询日志可以通过遍历 `slowlog` 链表来完成。
- `slowlog` 链表的长度就是服务器所保存慢查询日志的数量。
- 新的慢查询日志会被添加到 `slowlog` 链表的表头，如果日志的数量超过 `slowlog-max-len` 选项的值，那么多出来的日志会被删除。

## 7、监视器

- 客户端可以通过执行 MONITOR 命令，将客户端转换成监视器，接收并打印服务器处理的每个命令请求的相关信息。
- 当一个客户端从普通客户端变为监视器时，该客户端的 `REDIS_MONITOR` 标识会被打开。
- 服务器将所有监视器都记录在 `monitors` 链表中。
- 每次处理命令请求时，服务器都会遍历 `monitors` 链表，将相关信息发送给监视器。

## Redis实现访问频率的几种方式：

key = media_id + game_id + plat_id　　1000条 / 秒

频次限制器模式是一种特殊的计数器，它常被用来限制某个操作可以被执行的频次。这个模式的实质其实是限制对一个公共API执行访问请求的次数限制

使用redis进行限流，其很好地解决了分布式环境下多实例所导致的并发问题。因为使用redis设置的计时器和计数器均是全局唯一的，不管多少个节点，它们使用的都是同样的计时器和计数器，因此可以做到非常精准的流控。同时，这种方案编码并不复杂，可能需要的代码不超过10行。

三、基于Redis功能的实现

简陋的设计思路：假设一个用户（用IP判断）每分钟访问某一个服务接口的次数不能超过10次，那么我们可以在Redis中创建一个键，并此时我们就设置键的过期时间为60秒，每一个用户对此服务接口的访问就把键值加1，在60秒内当键值增加到10的时候，就禁止访问服务接口。在某种场景中添加访问时间间隔还是很有必要的。

1）使用Redis的incr命令，将计数器作为Lua脚本

```
local current
current = redis.call("incr",KEYS[1])
if tonumber(current) == 1 then
    redis.call("expire",KEYS[1],1)
end
```

Lua脚本在Redis中运行，保证了incr和expire两个操作的原子性。

2）使用Reids的列表结构代替incr命令

```
FUNCTION LIMIT_API_CALL(ip)
current = LLEN(ip)
IF current > 10 THEN
    ERROR "too many requests per second"
ELSE
    IF EXISTS(ip) == FALSE
        MULTI
            RPUSH(ip,ip)
            EXPIRE(ip,1)
        EXEC
    ELSE
        RPUSHX(ip,ip)
    END
    PERFORM_API_CALL()
END
```

Rate Limit使用Redis的列表作为容器，LLEN用于对访问次数的检查，一个事物中包含了RPUSH和EXPIRE两个命令，用于在第一次执行计数是创建列表并设置过期时间，

RPUSHX在后续的计数操作中进行增加操作。

打包到一个脚本，避免竞争状态, 但是影响并发量。

其他限流算法： 漏桶和令牌环

# 深入理解Java虚拟机

为什么学习JVM？

在很多情况下，提升硬件无法等比例提高程序性能， 甚至对其没有改善作用 ， 为此就需要从软件层次上对程序优化 ， 这部分涉及到JVM的很多知识 。

==》 Java高级开发人员， 架构师，系统调优师

## 1、走进Java

一次编写，四处运行；

相对安全的内存管理机制和访问机制，减少内存泄漏和指针越界；

热点代码检测和编译优化，性能提升；

600W开发人员，无数的开源组件。第三方库。

Java = 语言 + JVM + Java API

发展趋势： 模块化（解决小功能而重新安装部署整个系统的问题）， 混合语言（JRuby, Groovy），多核并行，进一步丰富语法（Java 8 新特性），64位虚拟机（性能不高，15%差距，内存开销大，JavaEE应用需要超过4G的内存）

Java8新特性：

2014年oracle正式发布, 加入特性

1） Lambda表达式 。

允许把一个函数作为方法的参数 。 “ -> ” 简化和紧凑代码 ， 还可以不需要声明参数类型（有歧义最好声明） ，主要是用来定义内部执行的方法接口，比如重写sort函数。 但是不好维护，平时少写，提出这个功能，个人觉得不用写匿名函数，简化代码是一方面，更多的是借鉴其他比如Lua语言的特性 ， 引入新的功能吧

2） 接口

接口可以自己实现方法，使用default关键字申明，被实现类继承或者重写 ， 还可以有静态方法

*之前的接口是个双刃剑，好处是面向抽象而不是面向具体编程，缺陷是，当需要修改接口时候，需要修改全部实现该接口的类*

3）时间日期类API

提供时区本地化和国际化 。 通过制定的API处理日期问题，当前时区， 当前时区的时间日期

## 2、自动内存管理机制

1）内存区域划分：

程序计数器： 每个t私有的，指示字节码的行号 ， 如果正执行native方法 ， 则设置为undefined

栈： 分成JVM栈和本地方法栈 ， 以栈帧的方式存储局部变量表，操作数 ， 返回地址,每次方法的调用与返回就代表着栈帧的入栈和出栈。如果超过指定长度则StackOverflow ； 如果申请不到足够内存则OOM

堆： 放对象的

方法区： t共享区域 ， 存储被加载的类信息 ， 常量 ， 静态变量，JIT编译的代码 。 重要的是运 行时常量池

## 3、虚拟机执行子系统

贝壳　Java研发　　北京

工作职责：

1、 负责搜索引擎、分布式计算、大数据基础架构、CRM、工作流等的设计研发。

任职资格：

1、 了解java，了解常用框架，如spring、ibatis、struts等；

2、 优秀的数据库设计优化能力，至少精通一种数据库应用；

3、 熟悉多线程及并发技术，熟悉socket网络编程；

4、 熟悉大数据处理和高并发性能解决方案；

5、 熟悉数据安全解决方案；

6、 扎实的计算机基础，熟悉常用数据结构和算法；

7、 熟悉linux，能熟练应用shell/python等脚本语言；

8、 学习能力强，有较好的沟通能力，能迅速融入团队；

9、 较强的逻辑思维能力，具有较强产品意识者优先。