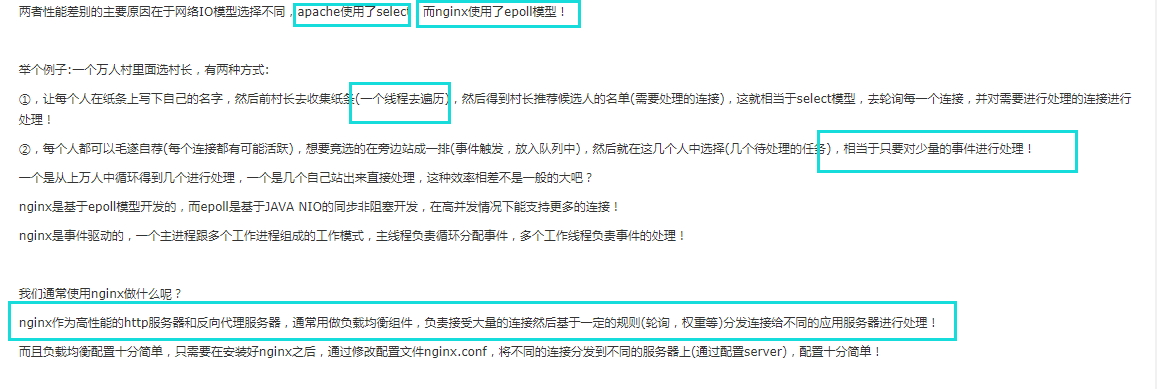
# <https://www.nowcoder.com/discuss/208511?type=post&order=time&pos=&page=1&subType=5>

# Nginx源码

## nginx为什么这么快



## Nginx基础架构

### Nginx的架构设计

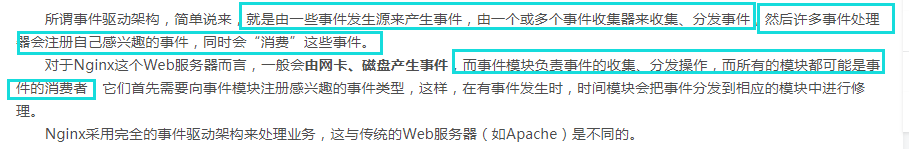
Nginx是一个功能堪比Apache的Web服务器。

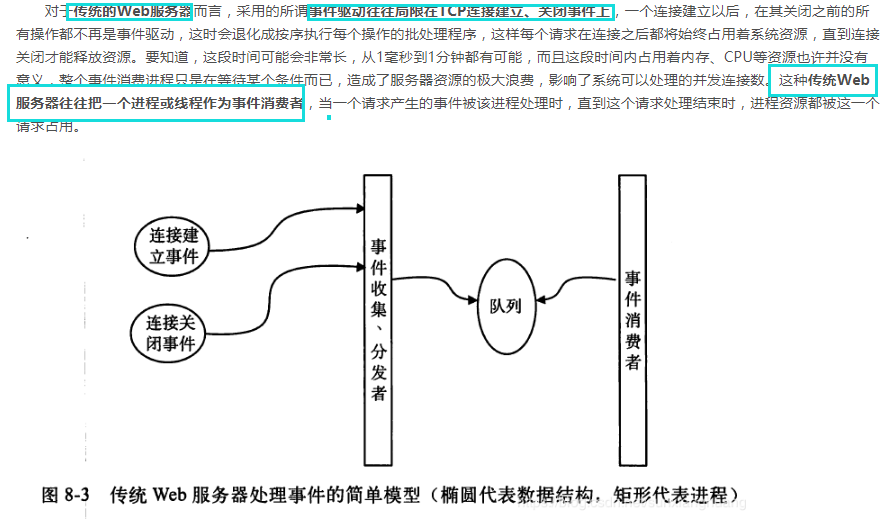
模块化，事件驱动，异步，单线程，非阻塞架构成为nginx代码的基础。Nginx大量使用多路复用和事件通知，并且给不同的进程分配不同的任务。数量有限的工作进程（Worker）使用高效的单线程循环处理连接。每个worker进程每秒可以处理数千个并发连接、请求。

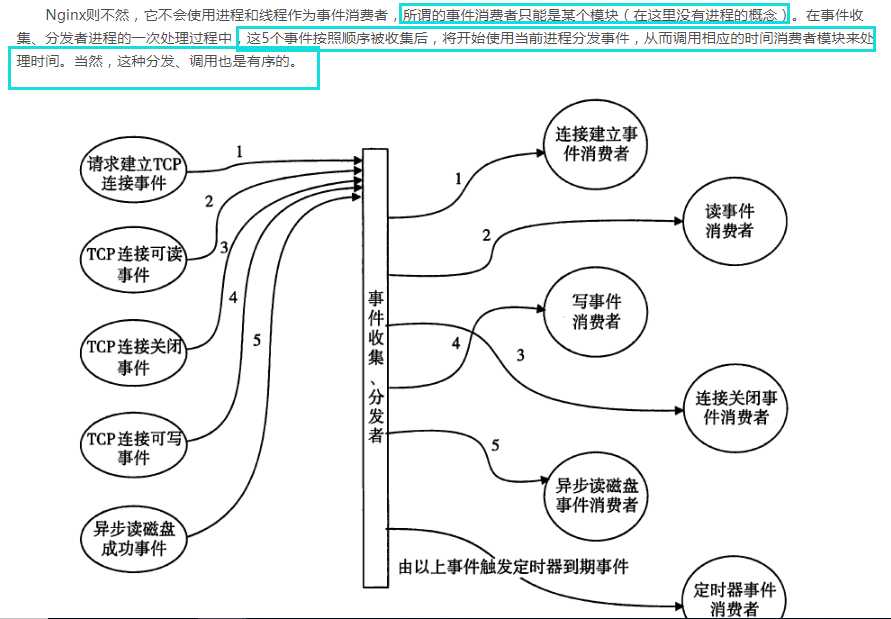
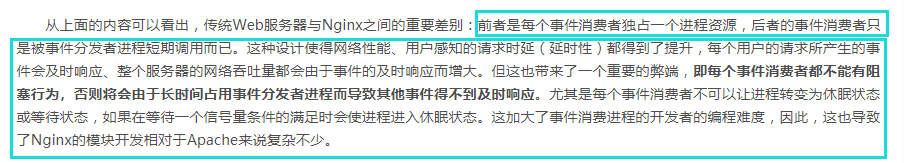
#### 优秀的模块化设计



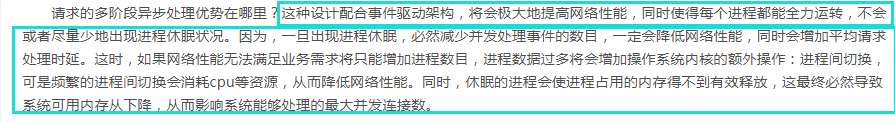
#### 事件驱动架构



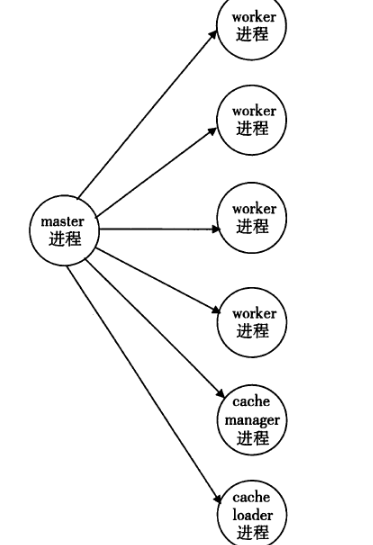


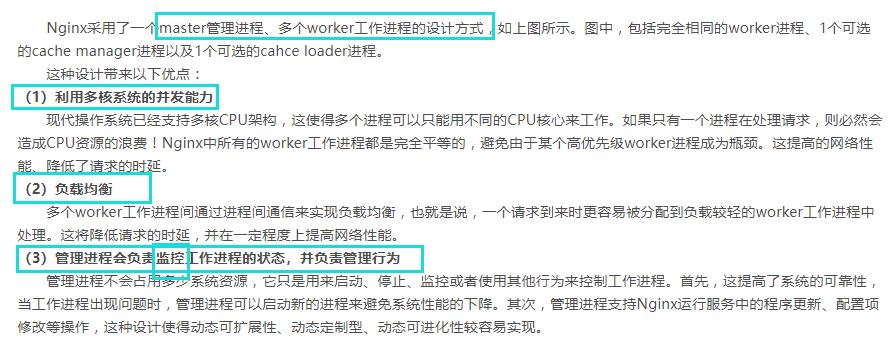
 

#### 请求多阶段异步处理

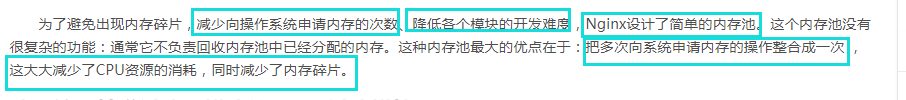


#### 管理进程，多工作进程设计

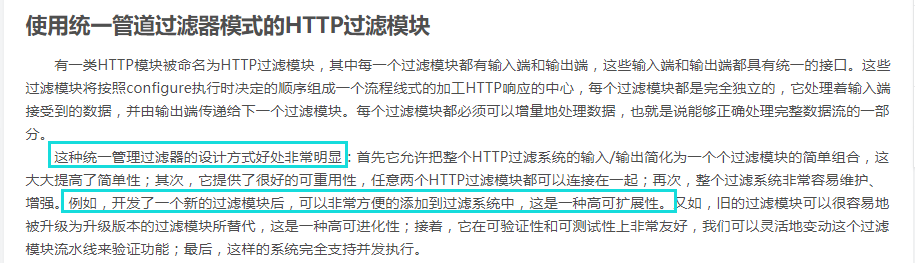




#### 内存池的设计



http过滤模块



### Nginx的性能（性能超越apache）

网络性能：

在大多数场景下，随着服务器上并发连接数的增加，网络性能都会有所下降。目前，我们在谈网络性能时，更多的是对应于高并发场景。例如，在几万或者几十万并发连接下，要求我们的服务器仍然可以保持较高的网络吞吐量，而不是当并发连接数达到一定数量时，服务器的CPU等资源大都浪费在进程间切换、休眠、等待等其他活动上，导致吞吐量大幅下降。

单次请求的延迟性

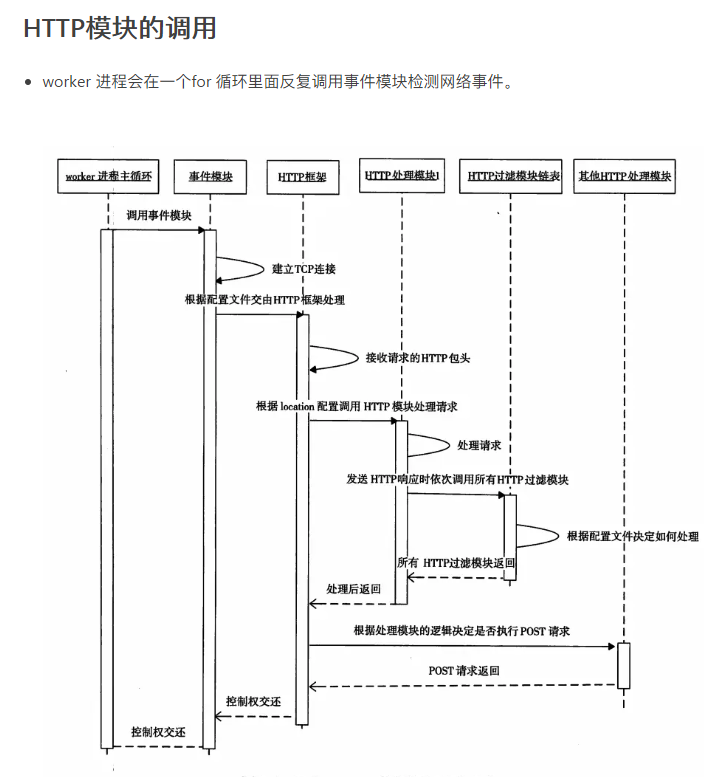
服务器在低并发和高并发连接数量下，单个请求的平均延迟时间肯定是不同的。Nginx在设计时，更应该考虑的是再高并发下如何保持平均实验性，使其不要上升得太快。

网络效率

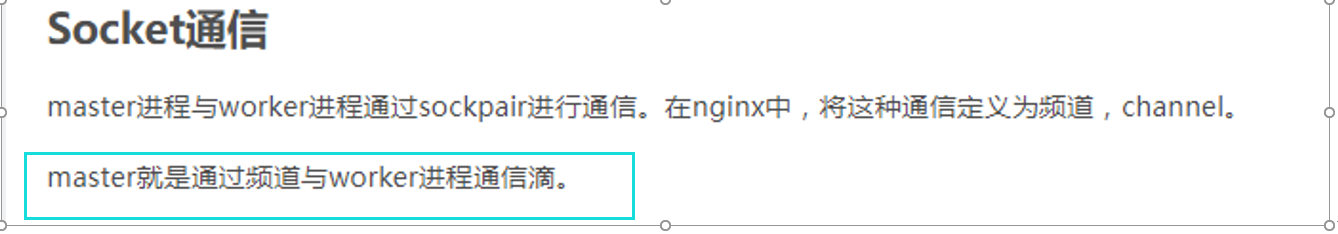
网络效率很好理解，就是使用网络的效率。例如，使用长连接（keepalive）代替短连接以减少建立、关闭连接带来的网络交互，使用压缩算法来增加相同吞吐量下信息携带量，使用缓存来减少网络交互次数等，它们都可以提高网络效率。

Nginx的

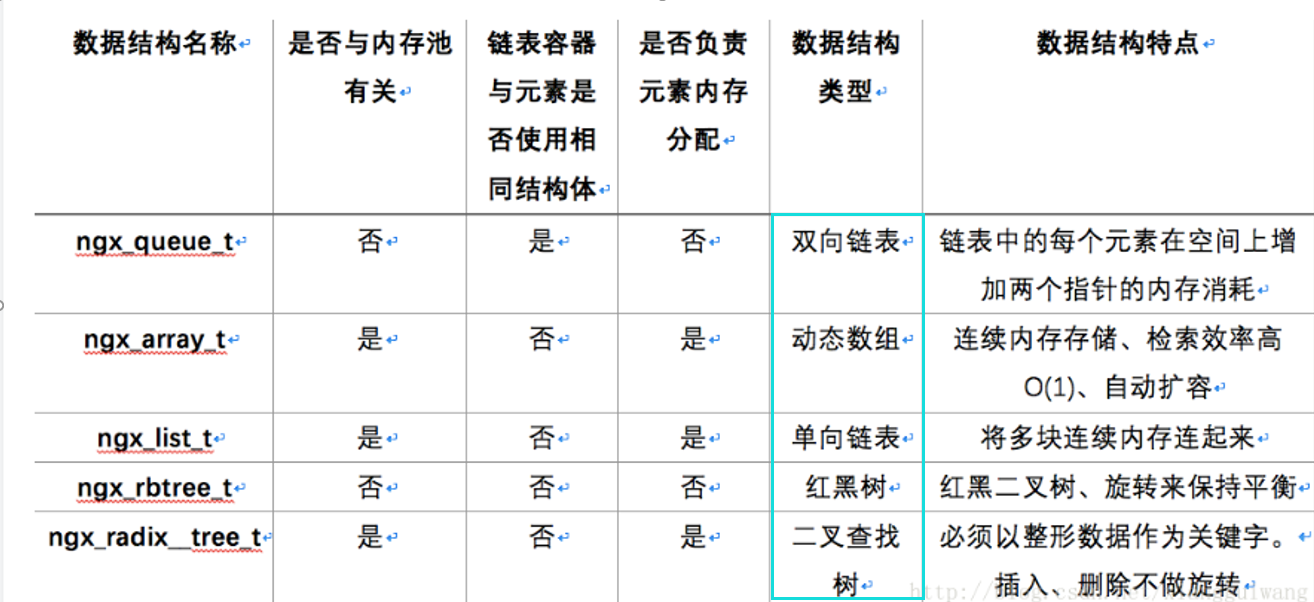
## HTTP框架



## 进程间的通信机制



## Nginx高级数据结构



## slab共享内存

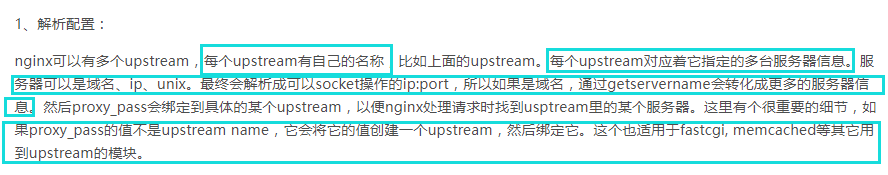
<https://www.cnblogs.com/jimodetiantang/p/9193858.html>

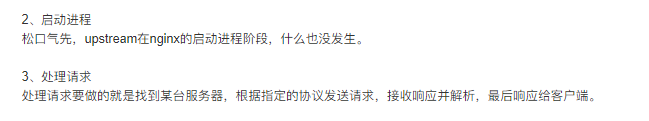
Nginx 的 slab 机制主要是和共享内存一起使用，Nginx 在解析完配置文件，把即将使用的共享内存全部以 list 链表的形式组织在全局变量 cf->cycle->shared\_memory 下之后，就会统一进行实际的内存分配，而 Nginx 的 slab 机制要做的就是对这些共享内存进行进一步的内部划分与管理。

## upstream机制设计

upstream可以连接很多服务器（upstream让nginx成为代理服务器，可以连接fastcgi协议的php，可以连接memcached、redis、mongodb、mysql handlersocket，uwsgi协议的服务器，甚至一切服务器）

nginx的生命周期有3个重要过程：解析配置、启动进程、处理请求。我们将围绕这3个阶段分析upstream如何被实现。





## apache和nginx的区别

### Nginx

1. 轻量级，采用 C 进行编写，同样的 web 服务，会占用更少的内存及资源
2. 抗并发，nginx 以 epoll and kqueue 作为开发模型，处理请求是异步非阻塞的，负载能力比 apache 高很多，而 apache 则是阻塞型的。在高并发下 nginx 能保持低资源低消耗高性能 ，而 apache 在 PHP 处理慢或者前端压力很大的情况下，很容易出现进程数飙升，从而拒绝服务的现象。
3. nginx 处理静态文件好，静态处理性能比 apache 高三倍以上
4. nginx 的设计高度模块化，编写模块相对简单
5. nginx 配置简洁，正则配置让很多事情变得简单，而且改完配置能使用 -t 测试配置有没有问题，apache 配置复杂 ，重启的时候发现配置出错了，会很崩溃
6. nginx 作为负载均衡服务器，支持 7 层负载均衡
7. nginx 本身就是一个反向代理服务器，而且可以作为非常优秀的邮件代理服务器
8. 启动特别容易, 并且几乎可以做到 7\*24 不间断运行，即使运行数个月也不需要重新启动，还能够不间断服务的情况下进行软件版本的升级
9. 社区活跃，各种高性能模块出品迅速

### Apache

1. apache 的 rewrite 比 nginx 强大，在 rewrite 频繁的情况下，用 apache
2. apache 发展到现在，模块超多，基本想到的都可以找到
3. apache 更为成熟，少 bug ，nginx 的 bug 相对较多
4. apache 超稳定
5. apache 对 PHP 支持比较简单，nginx 需要配合其他后端用
6. apache 在处理动态请求有优势，nginx 在这方面是鸡肋，一般动态请求要 apache 去做，nginx 适合静态和反向。
7. apache 仍然是目前的主流，拥有丰富的特性，成熟的技术和开发社区

### 总结

两者最核心的区别在于 apache 是同步多进程模型，一个连接对应一个进程，而 nginx 是异步的，多个连接（万级别）可以对应一个进程

一般来说，需要性能的 web 服务，用 nginx 。如果不需要性能只求稳定，更考虑 apache ，后者的各种功能模块实现得比前者，例如 ssl 的模块就比前者好，可配置项多。epoll(freebsd 上是 kqueue ) 网络 IO 模型是 nginx 处理性能高的根本理由，但并不是所有的情况下都是 epoll 大获全胜的，如果本身提供静态服务的就只有寥寥几个文件，apache 的 select 模型或许比 epoll 更高性能。当然，这只是根据网络 IO 模型的原理作的一个假设，真正的应用还是需要实测了再说的。

更为通用的方案是，前端 nginx 抗并发，后端 apache 集群，配合起来会更好。

## tomcat与nginx与apache的区别

题主说的Apache，指的应该是Apache软件基金会下的一个项目——Apache HTTP Server Project；Nginx同样也是一款开源的HTTP服务器软件（当然它也可以作为邮件代理服务器、通用的TCP代理服务器）

HTTP服务器本质上也是一种应用程序——它通常运行在服务器之上，绑定服务器的IP地址并监听某一个tcp端口来接收并处理HTTP请求，这样客户端（一般来说是IE, Firefox，Chrome这样的浏览器）就能够通过HTTP协议来获取服务器上的网页（HTML格式）、文档（PDF格式）、音频（MP4格式）、视频（MOV格式）等等资源

Apache Tomcat则是Apache基金会下的另外一个项目，与Apache HTTP Server相比，Tomcat能够动态的生成资源并返回到客户端。Apache HTTP Server和Nginx都能够将某一个文本文件的内容通过HTTP协议返回到客户端，但是这个文本文件的内容是固定的——也就是说无论何时、任何人访问它得到的内容都是完全相同的，这样的资源我们称之为静态资源。动态资源则与之相反，在不同的时间、不同的客户端访问得到的内容是不同的，例如：

包含显示当前时间的页面

显示当前IP地址的页面

Apache HTTP Server和Nginx本身不支持生成动态页面，但它们可以通过其他模块来支持（例如通过Shell、PHP、Python脚本程序来动态生成内容）。

如果想要使用Java程序来动态生成资源内容，使用这一类HTTP服务器很难做到。Java Servlet技术以及衍生的Java Server Pages技术可以让Java程序也具有处理HTTP请求并且返回内容（由程序动态控制）的能力，Tomcat正是支持运行Servlet/JSP应用程序的容器（Container）:

Tomcat运行在JVM之上，它和HTTP服务器一样，绑定IP地址并监听TCP端口，同时还包含以下指责：

* 管理Servlet程序的生命周期
* 将URL映射到指定的Servlet进行处理
* 与Servlet程序合作处理HTTP请求——根据HTTP请求生成HttpServletResponse对象并传递给Servlet进行处理，将Servlet中的HttpServletResponse对象生成的内容返回给浏览器

### 各用其长

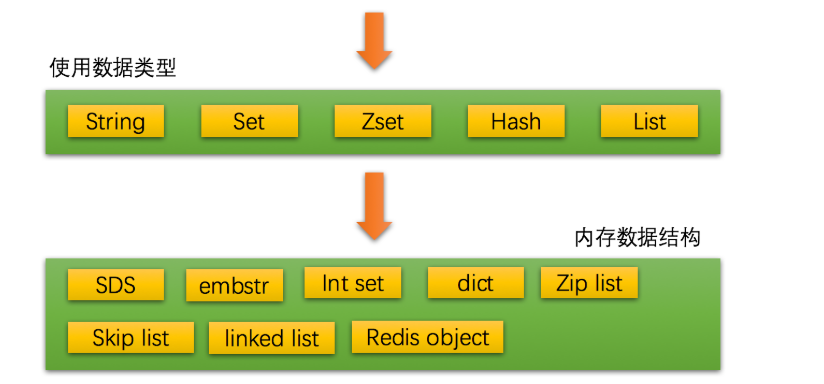
* 动静态资源分离——运用Nginx的反向代理功能分发请求：所有动态资源的请求交给Tomcat，而静态资源的请求（例如图片、视频、CSS、JavaScript文件等）则直接由Nginx返回到浏览器，这样能大大减轻Tomcat的压力。
* 负载均衡，当业务压力增大时，可能一个Tomcat的实例不足以处理，那么这时可以启动多个Tomcat实例进行水平扩展，而Nginx的负载均衡功能可以把请求通过算法分发到各个不同的实例进行处理

# Redis源码

## redis存储系统原理

### redis底层数据结构





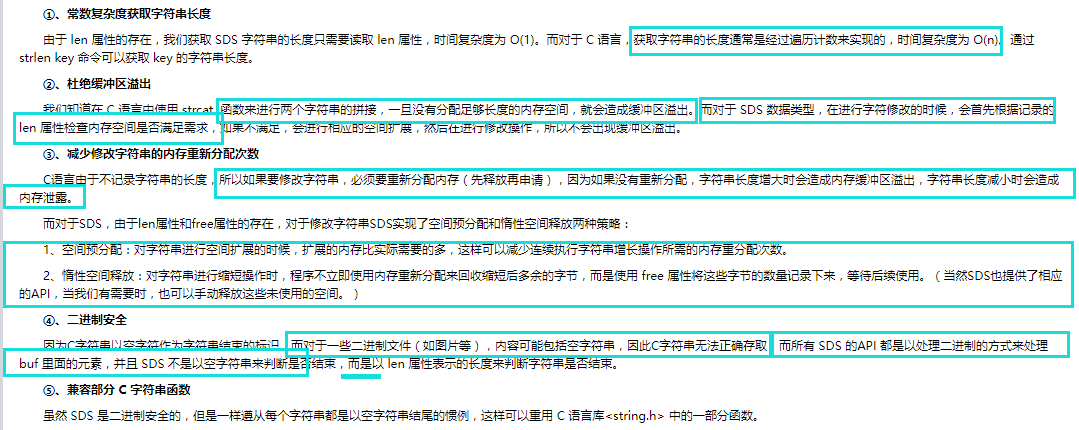
redisobject是一个redis对象，embstr是一个编码方式，string的两种编码方式（row和embstr）embstr编码是专门用于保存短字符串的一种优化编码方式，

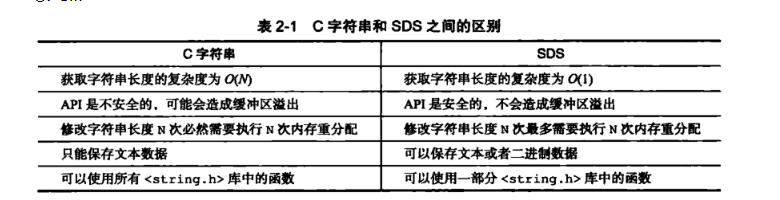
#### 简单动态字符串

Redis 是用 C 语言写的，但是对于Redis的字符串，却不是 C 语言中的字符串（即以空字符’\0’结尾的字符数组），它是自己构建了一种名为 简单动态字符串（simple dynamic string,SDS）的抽象类型，并将 SDS 作为 Redis的默认字符串表示。



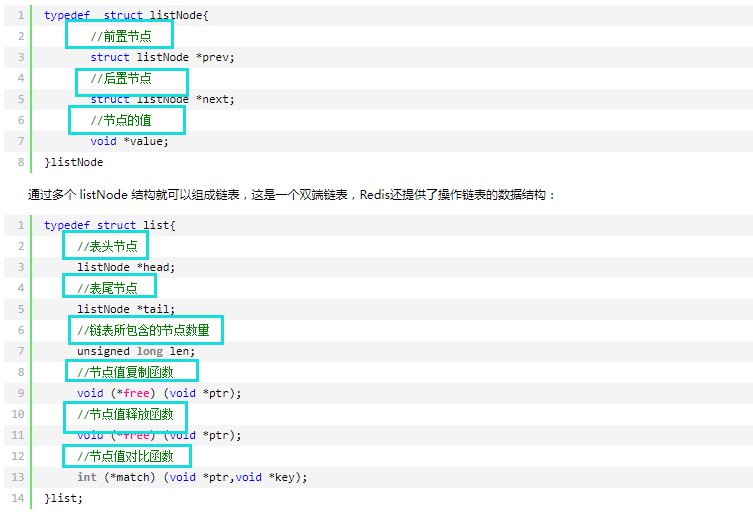
##### 为什么要使用SDS，使用SDS的好处

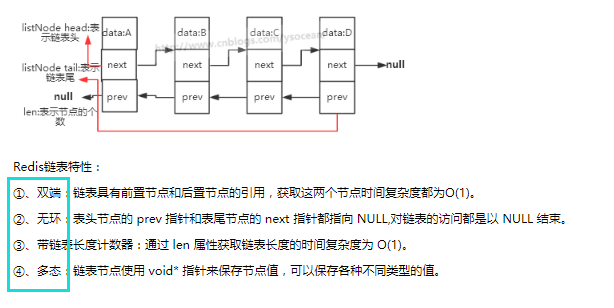




#### 链表

Redis自己构建了链表的实现

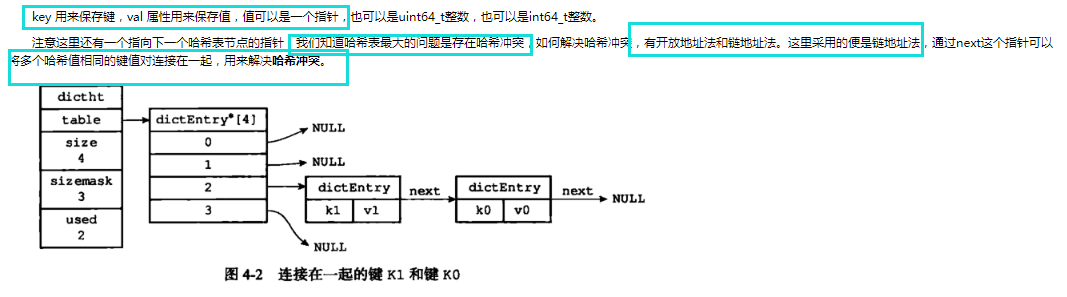


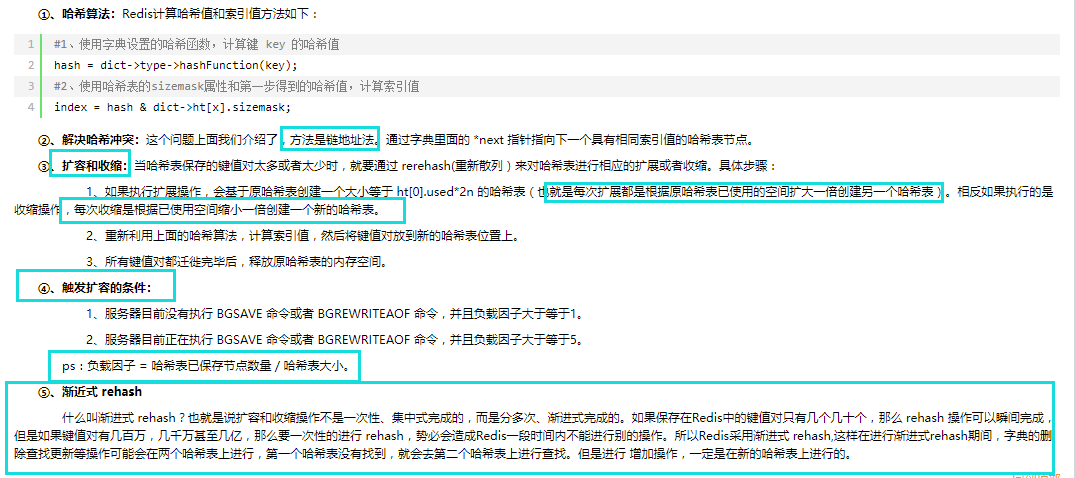


#### 字典

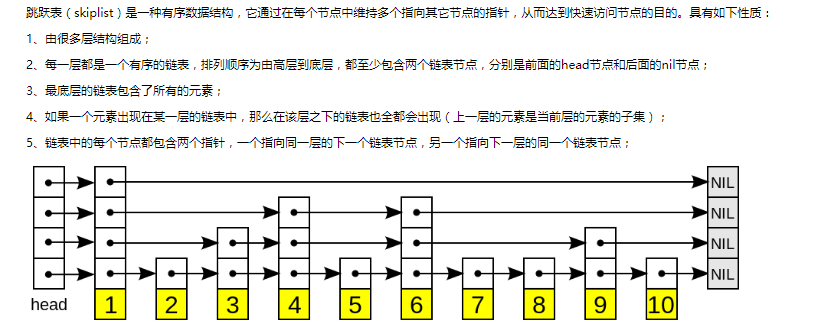
Redis 的字典使用哈希表作为底层实现



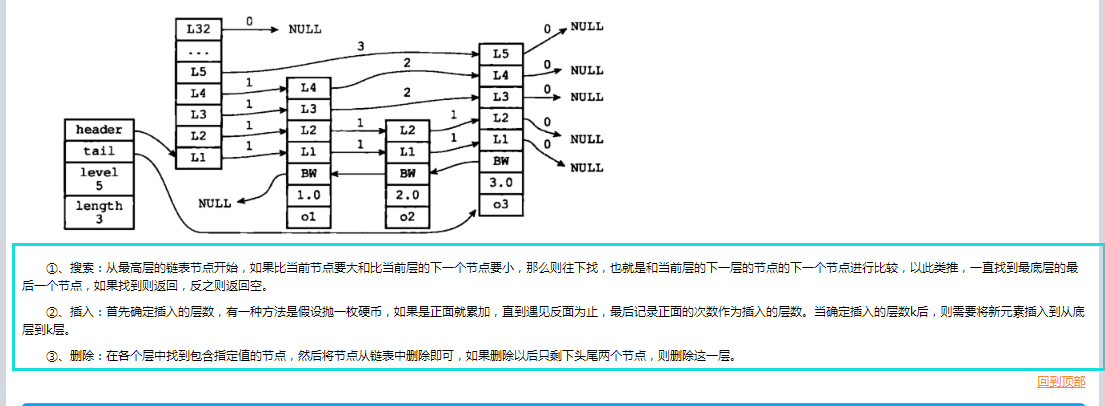




#### 跳跃表



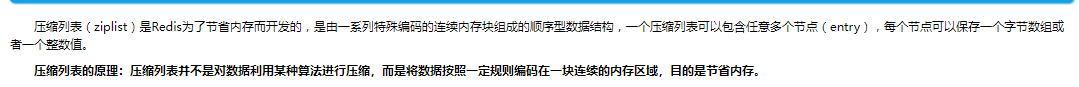




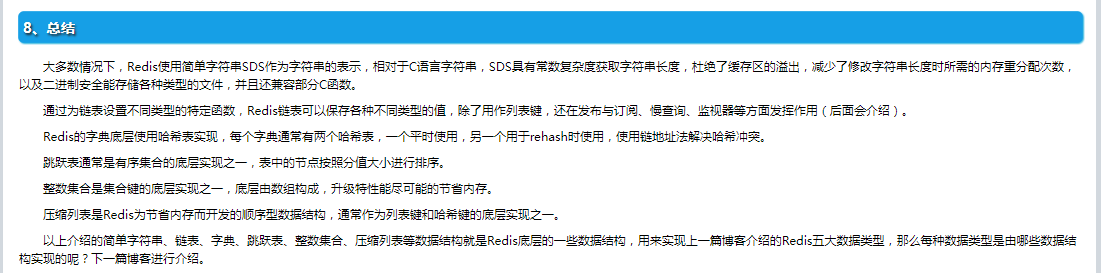
#### 整数集合



#### 压缩列表

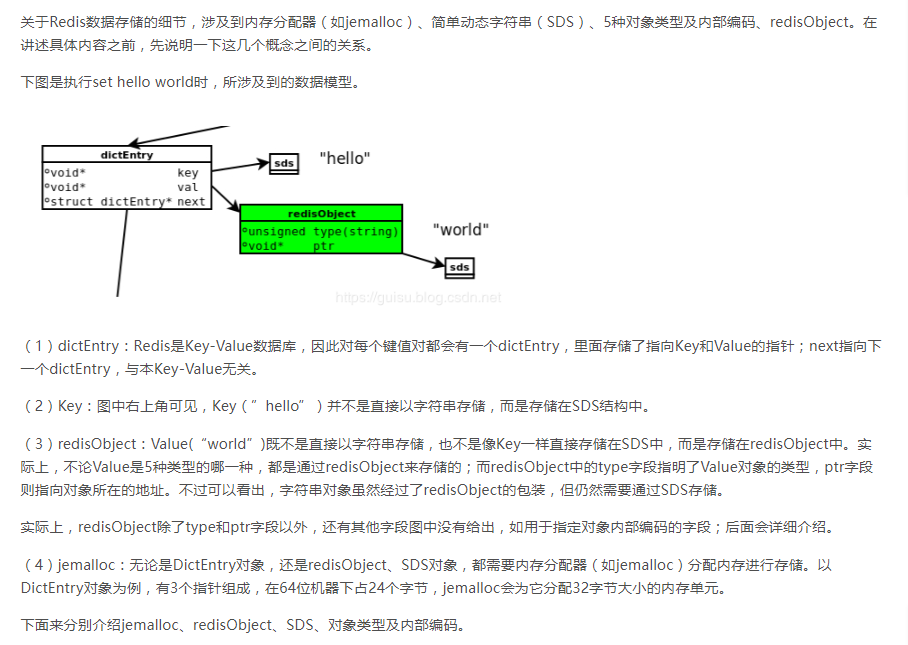


#### 总结



### redis底层是用什么结构来存储数据的呢？

redis底层是使用C语言来编写的，我们可以看到它的数据结构声明。一个 dict 有两个dictht，一个dictht有一个dictEntry数组，每个dictEntry有next指针，redisObject是真正存储redis各种类型的结构。因此是一个链表结构。从上面的分析可以看出Redis用拉链法解决冲突的哈希表结构。（这好像说的事dict）



### redis避免hash碰撞

“链地址法”的问题在于当碰撞剧烈时，性能退化严重，例如：当有n个数据，m个槽位，如果m=1，则整个Hash表退化为链表，查询复杂度O(n)

为了避免Hash碰撞，Redis的方案是“双dictht”，正常流程使用一个dictht，当发现碰撞剧烈（判断依据为当前槽位数和Key数的对比），分配一个更大的dictht，然后逐步将数据从老的dictht迁移到新的dictht上去。这就需要进行rehash。

### rehash方法的源码

rehash 操作不是一次性完成，而是采用渐进方式，这是为了避免一次性执行过多的 rehash 操作给服务器带来过大的负担。

渐进式 rehash 通过记录 dict 的 rehashidx 完成，它从0开始然后每执行一次rehash都会递增。例如在一次 rehash 中，要把 dict[0] rehash到dict[1]，这一次会把 dict[0] 上 table[rehashidx] 的键值对 rehash 到 dict[1] 上，dict[0] 的 table[rehashidx] 指向 null，并令 rehashidx++。

在 rehash 期间，每次对字典执行添加、删除、查找或者更新操作时，都会执行一次渐进式 rehash。

采用渐进式 rehash 会导致字典中的数据分散在两个 dictht 上，因此对字典的操作也需要到对应的 dictht 去执行。

### 底层数据的有序性是如何实现的

跳跃表是有序集合的底层实现之一。

跳跃表是基于多指针有序链表实现的，可以看成多个有序链表。

跳跃表是一种随机化数据结构，查找、添加、删除操作都可以在对数期望时间下完成。

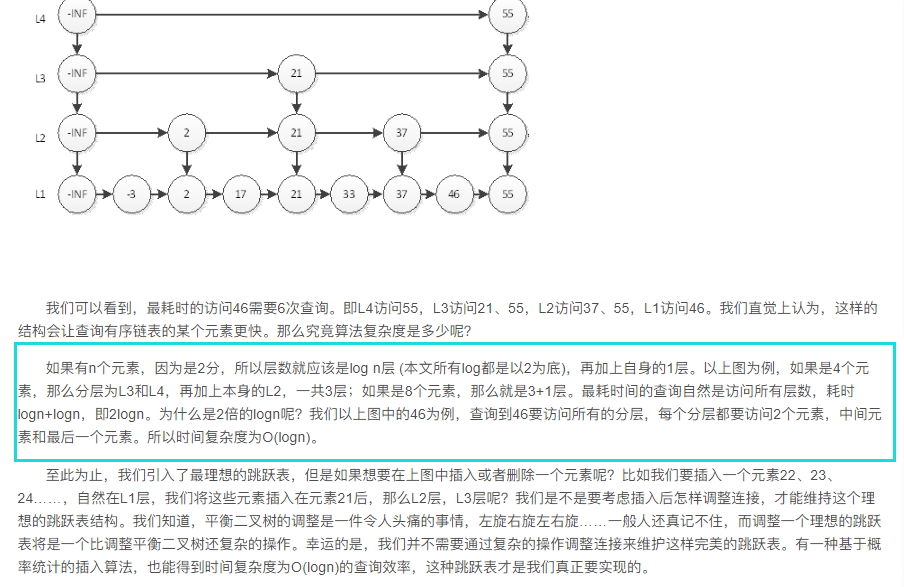
跳跃表目前在 Redis 的唯一作用，就是作为有序集类型的底层数据结构（之一，另一个构成有序集的结构是字典）。

与红黑树等平衡树相比，跳跃表具有以下优点：

插入速度非常快速，因为不需要平衡树的旋转操作；

更容易实现；

支持无锁操作。



## redis数据结构及使用场景

string:字符串和数字的直接存取，分布式的session系统（token,session）

hash：Redis 数据类型特点与使用场景

list:关注列表，最新消息排行，消息队列

set：去重，将用户所有关注的存在一个集合中

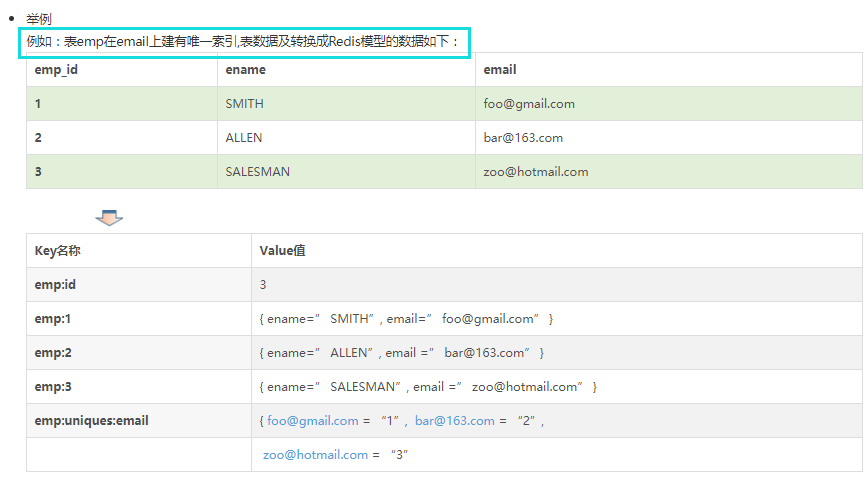
sorted set:排序（内部使用hashmap和跳跃表来实现有序）

## 数据模型与键值映射（库表与redis模型间的映射方式）

### 案例







## 内存操作与磁盘同步

### redis内存操作

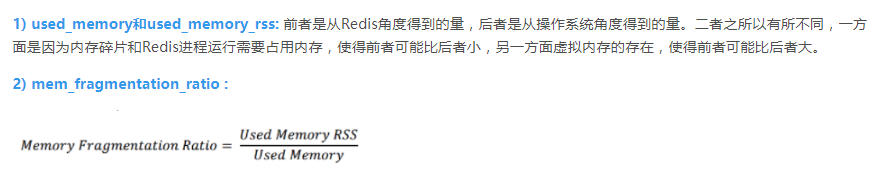
#### 1.内存使用情况

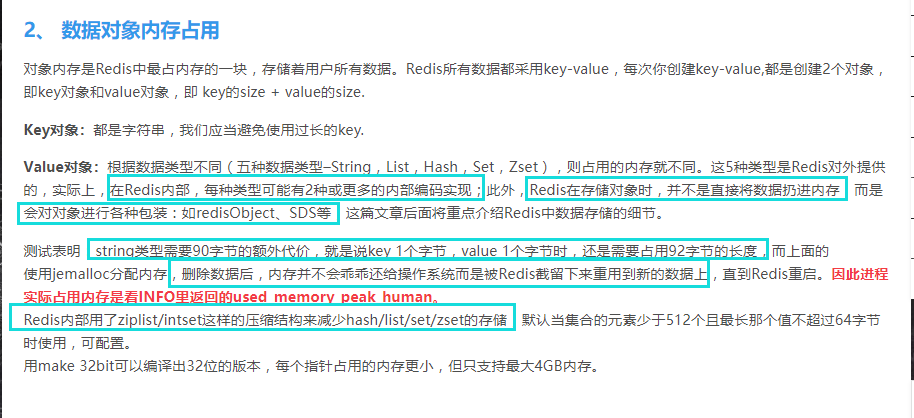
可以通过info memory命令查看内存使用情况

重点关注used\_memory以及used\_memory\_rss和碎片率mem\_fragmentation\_ratio。

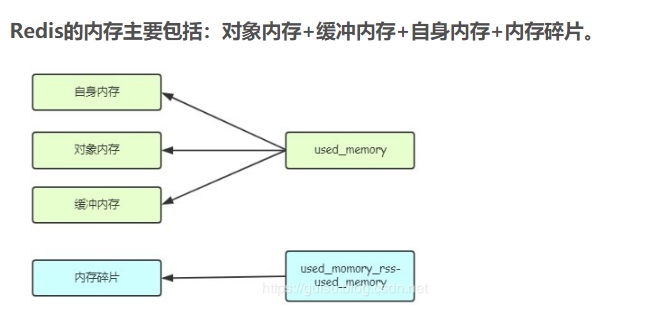
used\_memory: Redis分配的内存总量，即存储的所有数据占用的内存。包括redis进程内部开销和使用的虚拟内存（即swap)，单位byte。

used\_memory\_rss:从系统角度,显示Redis进程占用的物理内存总量，与top及ps命令看到的值是一致的；除了分配器分配的内存之外，used\_memory\_rss还包括进程运行本身需要的内存、内存碎片等，但是不包括虚拟内存。





#### 2.内存占用分析



#### 3内存管理

Redis通过控制内存上限和回收策略实现内存管理，使用maxmemory参数限制最大可用内存，限制内存的主要目的是：用于缓存场景，当超出内存上限时使用LRU等删除策略释放空间；防止所用内存超过服务器的物理内存。



### redis的磁盘同步

#### RDB

快照是默认的持久化方式。这种方式是就是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。可以通过配置设置自动做快照持久化的方式。我们可以配置redis在n秒内如果超过m个key被修改就自动做快照，下面是默认的快照保存配置：

快照保存流程

1. redis调用fork,现在有了子进程和父进程。

2. 父进程继续处理client请求，子进程负责将内存内容写入到临时文件。由于os的写时复制机制（copy on write)父子进程会共享相同的物理页面

，当父进程处理写请求时os会为父进程要修改的页面创建副本，而不是写共享的页面。所以子进程的地址空间内的数据是fork时刻整个数据库的一个快照。

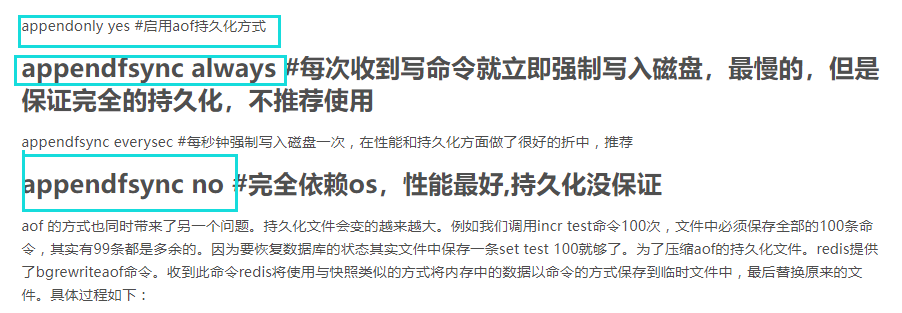
3. 当子进程将快照写入临时文件完毕后，用临时文件替换原来的快照文件，然后子进程退出（fork一个进程入内在也被复制了，即内存会是原来的两倍）。

client 也可以使用save或者bgsave命令通知redis做一次快照持久化。save操作是在主线程中保存快照的，由于redis是用一个主线程来处理所有 client的请求，这种方式会阻塞所有client请求。所以不推荐使用。

#### AOF

aof 比快照方式有更好的持久化性，是由于在使用aof持久化方式时，redis会将每一个收到的写命令都通过write函数追加到文件中(默认是appendonly.aof)。

不过我们可以通过配置文件告诉redis我们想要通过fsync函数强制os写入到磁盘的时机。



aof流程

1. redis调用fork ，现在有父子两个进程

2. 子进程根据内存中的数据库快照，往临时文件中写入重建数据库状态的命令

3. 父进程继续处理client请求，除了把写命令写入到原来的aof文件中。同时把收到的写命令缓存起来。这样就能保证如果子进程重写失败的话并不会出问题。

4. 当子进程把快照内容写入已命令方式写到临时文件中后，子进程发信号通知父进程。然后父进程把缓存的写命令也写入到临时文件。

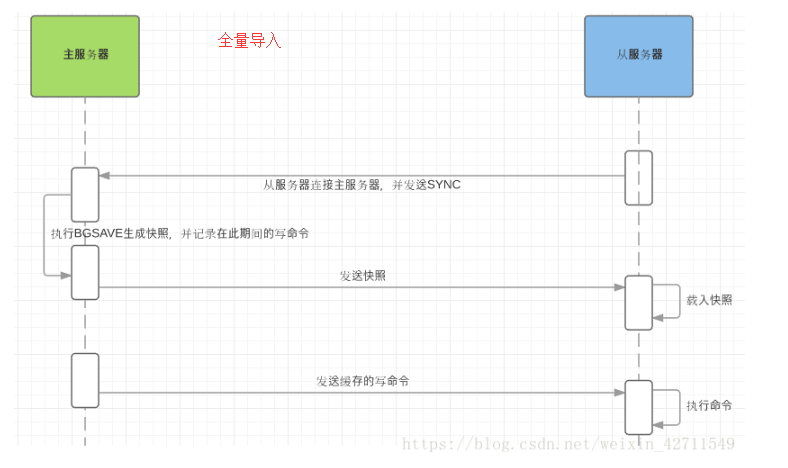
5. 现在父进程可以使用临时文件替换老的aof文件，并重命名，后面收到的写命令也开始往新的aof文件中追加。

需要注意到是重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件，这点和快照有点类似。

## 主从同步，原子操作

### 主从同步

Redis主从同步。数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。这使得Redis可执行单层树复制。存盘可以有意无意的对数据进行写操作。由于完全实现了发布/订阅机制，使得从数据库在任何地方同步树时，可订阅一个频道并接收主服务器完整的消息发布 记录。同步对读取操作的可扩展性和数据冗余很有帮助。



　Redis全量复制一般发生在Slave初始化阶段，这时Slave需要将Master上的所有数据都复制一份。具体步骤如下：

　　1）从服务器连接主服务器，发送SYNC命令；

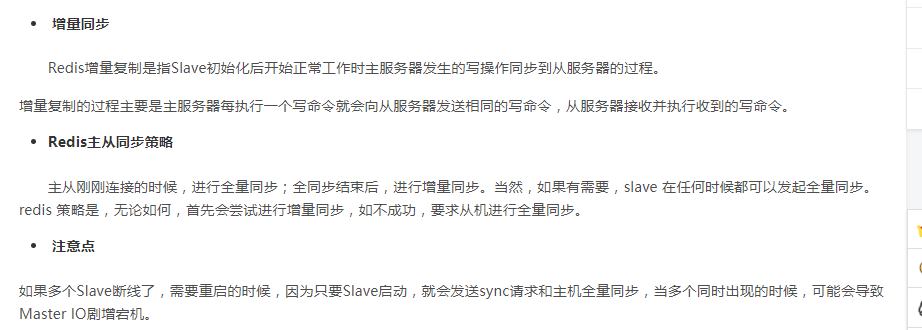
　　2）主服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；

　　3）主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；

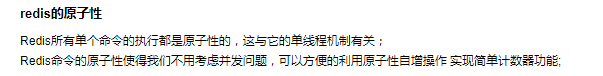
　　4）从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；

　　5）主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；

6）从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令；



### 原子操作



## 底层IO实现解析

首先，Redis 是跑在单线程中的，所有的操作都是按照顺序线性执行的，但是由于读写操作等待用户输入或输出都是阻塞的，所以 I/O 操作在一般情况下往往不能直接返回，这会导致某一文件的 I/O 阻塞导致整个进程无法对其它客户提供服务，而 I/O 多路复用就是为了解决这个问题而出现的。

Blocking I/O: 传统的阻塞 I/O 模型到底是如何工作的：当使用 read 或者 write 对某一个文件描述符（File Descriptor 以下简称 FD)进行读写时，如果当前 FD 不可读或不可写，整个 Redis 服务就不会对其它的操作作出响应，导致整个服务不可用。

I/O多路复用：

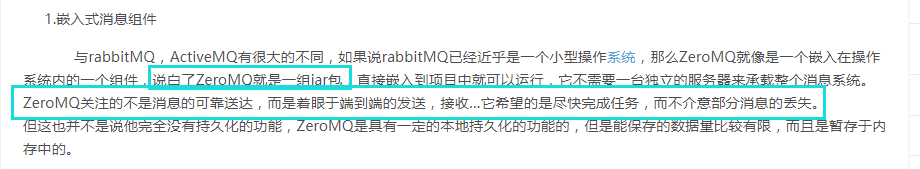
## redis为什么这么快

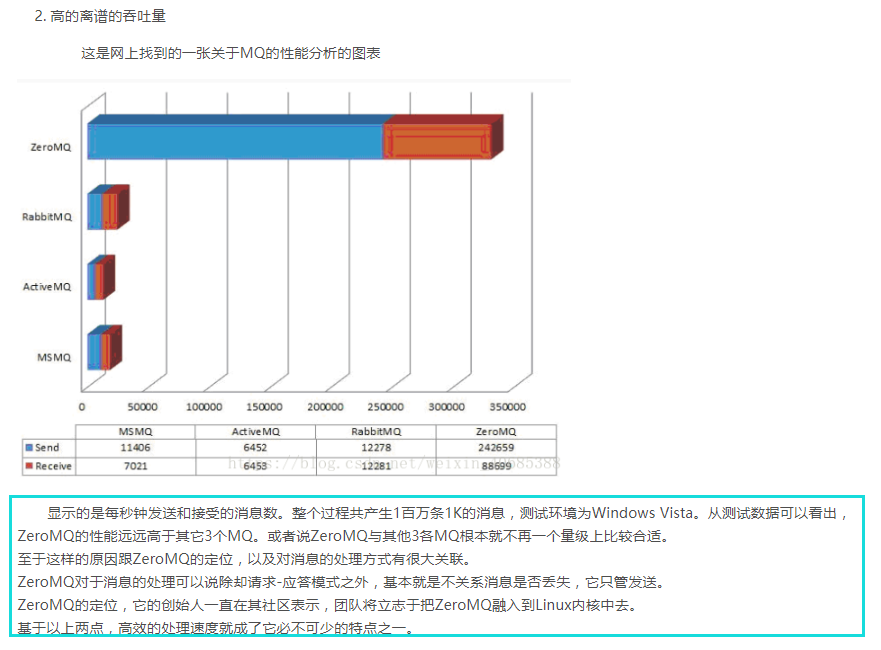


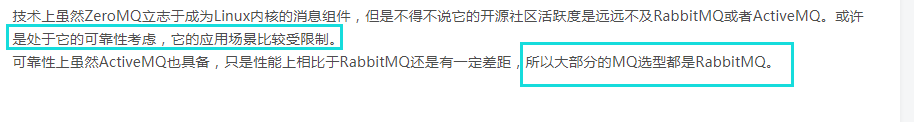
# ZeroMQ源码(稍微了解了一下)

阅读了 ZMQ 的 Guide 文档后，我的理解是，这是个类似于 Socket 的一系列接口，他跟 Socket 的区别是：普通的 socket 是端到端的（1:1的关系），而 ZMQ 却是可以N：M 的关系，人们对 BSD 套接字的了解较多的是点对点的连接，点对点连接需要显式地建立连接、销毁连接、选择协议（TCP/UDP）和处理错误等，而 ZMQ 屏蔽了这些细节，让你的网络编程更为简单。ZMQ 用于 node 与 node 间的通信，node 可以是主机或者是进程。

## ZeroMq特点







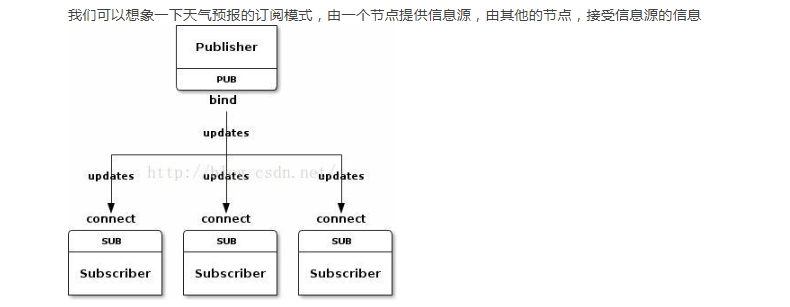
## 消息模型 发布订阅/推拉模型和其他两种

分别是“Request-Reply “，”Publisher-Subscriber“，”Parallel Pipeline**”**

### Request-Reply

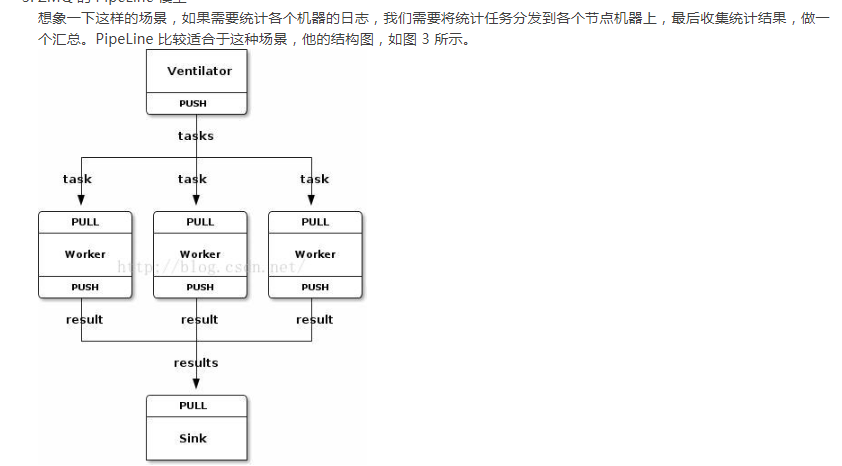


### Publicsher-Subscriber





### parallel pipeline



## 通信协议 inproc/ipc/tcp/pgm

## 性能分析与经典MQ对比

## 底层网络通信实现机制

## zmq系统架构与实现原理

# 多路复用机制

(1)select==>时间复杂度O(n)

它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以select具有O(n)的无差别轮询复杂度，同时处理的流越多，无差别轮询时间就越长。

(2)poll==>时间复杂度O(n)

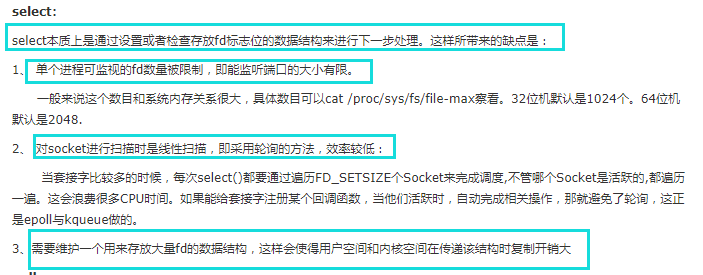
poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， 但是它没有最大连接数的限制，原因是它是基于链表来存储的.

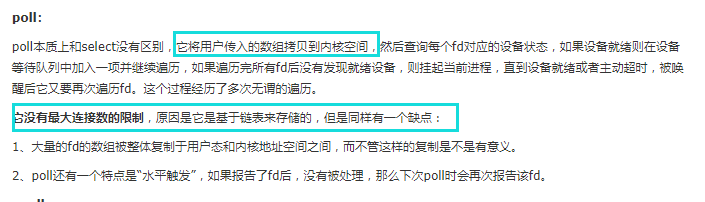
(3)epoll==>时间复杂度O(1)

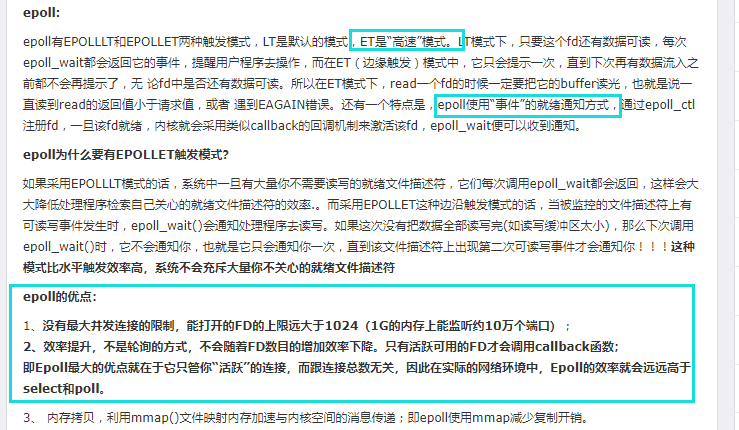
epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是事件驱动（每个事件关联上fd）的，此时我们对这些流的操作都是有意义的。（复杂度降低到了O(1)）

select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

epoll跟select都能提供多路I/O复用的解决方案。在现在的Linux内核里有都能够支持，其中epoll是Linux所特有，而select则应该是POSIX所规定，一般操作系统均有实现









# BIO和NIO

