1. 简单动态字符串
2. 概述

Redis没有直接使用C语言传统的字符串表示（以空字符结尾的字符数组），而是自己构建了一种名为简单动态字符串（SDS）的抽象类型，作为默认字符串表示。在Redis中，C字符串数组只会作为字符串字面量用在一些无需对字符串值进行修改的地方，如打印日志中。对于可修改的字符串值，都会使用SDS来表示字符串值，如包含字符串值的键值对。

1. SDS的定义

SDS的定义如下所示：

struct sdshdr{

int len;

int free;

char buf[];

};

len属性表示SDS所保存字符串的长度，即buf数组已使用字节的数量；

free属性表示buf数组中未使用字节的数量；

buf属性表示保存字符串的字节数组；

SDS中依然保持了以空字符结尾的管理，因为可以直接重用一部分C字符串函数库里的函数。

1. SDS与C字符串数组的区别
2. 获取字符串长度

C字符串数组并不记录自身的长度信息，因此为了获取长度需要遍历整个字符串，直到遇到空字符为止，复杂度为O(N)；

SDS中保存了len属性，可以直接获取SDS的长度信息，复杂度为O(1)

1. 杜绝了缓冲区的溢出

C字符串数组不记录长度容易造成缓冲区溢出，有些字符串函数在向字符串数组中添加字符时已经默认分配了足够多的内存，一旦假设不成立，就会产生缓冲区溢出；

SDS则不会有这种情况，当SDS的API需要对SDS进行修改时，首先检查SDS的空间是否满足修改所需的要求，如果不满足，API会自动将SDS的空间扩展至执行修改所需的大小

1. 减少修改字符串的内存重分配次数

对于C字符串数组来说，包含了N个字符的数组底层实现总是一个N+1个字符长的数组，因此每次增长或者缩短一个C字符串，都要对这个数组进行内存重分配操作；

对于SDS，通过未使用空间实现空间预分配和惰性空间释放两种策略。对于字符串增长操作，会为SDS分配额外的未使用空间，如果SDS的长度小于1MB，则分配和len属性同样大小的free空间，否则分配1MB的free空间，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。对于字符串缩短操作，不会立即回收缩短后多出来的字节，而是使用free空间将这些字节的数量记录起来，等待将来使用，减少了内存重分配次数。

1. 二进制安全

C字符串数组中的字符必须符合某种编码，并且除了字符串的末尾之外，字符串里面不能包含空字符，因为C字符串数组只能保存文本数据，不能保存图片，音频等二进制数据；

为了使Redis可以适用于各种场景，SDS的API都是二进制安全的，都会以处理二进制的方式来处理SDS存放在buf数组中的数据，buf数组用来保存二进制数据。

1. 链表
2. 概述

链表提供了高效的节点重排能力，以及顺序性的节点访问方式，并且可以通过增删节点来灵活地调整链表的长度。Redis使用C语言编写，C语言中没有链表实现，因此Redis构建了自己的链表实现。链表再Redis中的应用很广泛，如列表键的底层实现之一就是链表，当一个列表键中包含了较多的元素，或是列表中包含的元素都是比较长的字符串是，Redis就会使用链表作为列表键的底层实现。

1. 链表和链表节点的实现

链表由链表节点连接而成，每个链表节点使用listNode结构表示：

typedef struct listNode{

struct listNode \*prev;

struct listNode \*next;

void \*value;

}listNode;

prev为前置节点的指针，next为后置节点的指针，value为节点的值，多个listNode可以通过前后指针来组成双端链表。

链表的结构由list表示：

typedef struct list{

listNode \*head;

listNode \*tail;

unsigned long len;

void \*(\*dup)(void \*ptr);

void (\*free)(void \*ptr);

int (\*match)(void \*ptr, void \*key);

}list;

表头指针head，表尾指针tail，链表长度计数器len，以及后面的节点值复制函数dup，节点值释放函数free，节点值对比函数match，注意这几个函数都是使用void\*指针来保存节点值，可保存各种不同类型的值。