Redis没有直接使用C字符串（以’\0’结尾的字符数组），而是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用作Redis的默认字符串表示。在Redis里面，C字符串只会作常量值，比如打印日志：

redisLog**(**REDIS\_WARNING**,**"Fatal: Can't initialize Background Jobs."**);**

当Redis需要一个可被修改的字符串时，就会使用SDS来表示字符串值，在Redis中，包含字符串值的键值对在底层都是由SDS实现的。比如执行下面的命令：

redis**>** rpush fruits "apple" "banana" "cherry"

**(**integer**)** 3

Redis将在数据库中创建一个新的key-value对，其中：

key是一个字符串对象，对象的底层实现是一个保存了字符串”fruits”的SDS；

value是一个列表对象，列表对象包含了三个字符串对象，这三个字符串对象分别由三个SDS实现：第一个SDS保存着字符申”apple”，第二个SD5保存着字符串”banana”，第三个SDS保存着字符串”cherry”。

除了用来保存字符串之外，SDS还被用作缓冲区：AOF模块中的AOF缓冲区，以及客户端状态中的输人缓冲区，都是由SD5实现的。

1：SDS的定义

在sds.h中，定义了结构体sdshdr表示SDS，其定义如下：

struct sdshdr **{**

unsigned int len**;**

unsigned int free**;**

char buf**[];**

**};**

len记录buf中已使用的字节数量，也就是SDS保存的字符串的长度（不包括末尾的’\0’）；free记录buf中未使用的字节数量（也不包括’\0’）；buf是字节数组，用于保存字符串。比如下面的例子：



len为5，表示字符串长度；free为5，表示buf中尚有5字节的空间，buf存储了字符串”Redis”。SDS的API： sdslen和sdsavail用于根据SDS获得其len和free属性，函数实现如下：

static inline size\_t sdslen**(**const sds s**)**

**{**

struct sdshdr **\***sh **=** **(**void**\*)(**s**-(sizeof(**struct sdshdr**)));**

**return** sh**->**len**;**

**}**

static inline size\_t sdsavail**(**const sds s**)**

**{**

struct sdshdr **\***sh **=** **(**void**\*)(**s**-(sizeof(**struct sdshdr**)));**

**return** sh**->**free**;**

**}**

SDS遵循C字符串以’\0’结尾的惯例，注意该字节不计算在SDS的len属性里面。遵循空字符结尾这一惯例的好处是，SDS可以直接重用一部分C字符串函数库里面的函数。

2：SDS与C字符串的区别

C字符串不记录自身的长度信息，获取一个C字符串的长度的时间复杂度为O(N)。SDS在len属性中直接记录了字符串的长度，所以，获取一个SDS字符串长度的事件复杂度是O(1)。这确保获取字符串长度的工作不会成为Redis的性能瓶颈。即使对一个非常长的字符串键反复执行”strlen”命令，也不会对系统性能造成任何影响，因为”strlen”命令的复杂度仅为O(1)。

C字符串不记录长度带来的另一个问题是容易造成缓冲区滋出。比如strcat函数将src字符串中的内容拼接到dest字符串的末尾:

char **\***strcat**(**char **\***dst**,** const char **\***src**);**

如果dst的长度不足以容纳src，就会产生缓冲区滥出。

与C字符串不同，SDS的空间分配策略完全杜绝了发生缓冲区溢出的可能性：当SDS的API需要对SDS进行修改时，API会先检查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，然后才执行实际的修改操作。比如下面的函数实现：

sds sdscatlen**(**sds s**,** const void **\***t**,** size\_t len**)**

**{**

struct sdshdr **\***sh**;**

size\_t curlen **=** sdslen**(**s**);**

s **=** sdsMakeRoomFor**(**s**,**len**);**

**if** **(**s **==** **NULL)** **return** **NULL;**

sh **=** **(**void**\*)** **(**s**-(sizeof(**struct sdshdr**)));**

memcpy**(**s**+**curlen**,** t**,** len**);**

sh**->**len **=** curlen**+**len**;**

sh**->**free **=** sh**->**free**-**len**;**

s**[**curlen**+**len**]** **=** '\0'**;**

**return** s**;**

**}**

sds sdscat**(**sds s**,** const char **\***t**)**

**{**

**return** sdscatlen**(**s**,** t**,** strlen**(**t**));**

**}**

sdscat是通过sdscatlen实现的，在sdscatlen中，首先用sdsMakeRoomFor保证SDS具有足够的空间（sdsMakeRoomFor的函数实现见下面），然后才是将字符串t追加到s中。其他所有修改SDS的API都会通过sdsMakeRoomFor保证缓冲区不会溢出。

对于C字符串来说，一个包含N个字符的C字符串，它的底层实现总是一个长度为N+1的字节数组。因为C字符串的长度和底层字节数组的长度之间存这种关联性，所以每次增长或者缩短一个C字符串，程序总要对保存这个C字符串的数组进行一次内存重分配操作。

内存重分配涉及复杂的算法，且可能需要执行系统调用，所以它通常是一个比较耗时的操作。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，若每次修改字符串的长度都需要执行一次内存重分配，且这种修改频繁地发生的话，可能会对Redis的性能造成影响。

为了避免C字符串的这种缺陷，SDS通过未使用空间解除了字符申长度和底层数组长度之间的关联：在SDS中，buf数组的长度不一定就是字符数量加一，数组里面可以包含未使用的字节，而这些字节的数量就由SDS的free属性记录。SDS实现了空间预分配和惰性空间释放两种优化策略。

空间预分配用于优化SDS的字符串增长操作：当SDS的API需要对SDS进行空间扩展的时候，程序不仅会为SDS分配修改所必须要的空间，还会为SDS分配额外的未使用空间。 其中，额外分配的未使用空间数量的算法如下：

a：如果对SDS进行修改之后，SDS的长度（也即是len的值）将小于1MB，那么程序僵分配和len属性同样大小的未使用空间，这时SDS的 len属性的值将和free属性的值相同。比如，如果修改后，SDS的len将变成13字节，则程序也会分配13字节的未使用空间，因此SDS的buf数组的实际长度将变成13+13+1个字节。

b：如果对SDS进行修改之后，SDS的长度将大于等于1MB，那么程序会分配1MB的未 使用空间。比如，如果进行修改之后，SDS的len将变成30MB，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3 0 MB+1MB+1byte。

在扩展SDS空间之前，SDS的API会先检查未使用空间是否足够，如果足够的话，API就会直接使用未使用空间，而无须执行内存重分配。这就是sdsMakeRoomFor函数的作用，它的实现如下：

sds sdsMakeRoomFor**(**sds s**,** size\_t addlen**)**

**{**

struct sdshdr **\***sh**,** **\***newsh**;**

size\_t free **=** sdsavail**(**s**);**

size\_t len**,** newlen**;**

**if** **(**free **>=** addlen**)** **return** s**;**

len **=** sdslen**(**s**);**

sh **=** **(**void**\*)** **(**s**-(sizeof(**struct sdshdr**)));**

newlen **=** **(**len**+**addlen**);**

**if** **(**newlen **<** SDS\_MAX\_PREALLOC**)**

newlen **\*=** 2**;**

**else**

newlen **+=** SDS\_MAX\_PREALLOC**;**

newsh **=** zrealloc**(**sh**,** **sizeof(**struct sdshdr**)+**newlen**+**1**);**

**if** **(**newsh **==** **NULL)** **return** **NULL;**

newsh**->**free **=** newlen **-** len**;**

**return** newsh**->**buf**;**

**}**

其中，SDS\_MAX\_PREALLOC的值就是1024\*1024，也就是1M。参数addlen表示需要扩容的长度，该函数用于向SDS添加新内容时，保证SDS能容纳新内容。该函数不改变SDS的len属性，只改变free属性。扩容规则跟上面的描述是一致的。

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。

通过惰性空间释放策略，SDS避免了缩短字符串时所需的内存重分配操作，并为将来可能有的增长操作提供了优化。比如下面的sdsclear函数：

void sdsclear**(**sds s**)**

**{**

struct sdshdr **\***sh **=** **(**void**\*)** **(**s**-(sizeof(**struct sdshdr**)));**

sh**->**free **+=** sh**->**len**;**

sh**->**len **=** 0**;**

sh**->**buf**[**0**]** **=** '\0'**;**

**}**

C字符串中，除了字符串的末尾之外，字符串中不能包含空字符，这些限制使得C字符申只能保存文本数据，而不能保存像图片、音频这样的二进制数据。

虽然数据库一般用于保存文本数据，但使用数据库来保存二进制数据的场景也不少见，因此，为确保Redis可以适用于各种不同的使用场景，SDS的API都是二进制安全的(binary-safe)，所有SDS的API都会以处理二进制的方式来处理SDS存放在buf数组里的数据，程序不会对其中的数据做任何限制、过滤、或者假设，数据在写人时是什么样的，它被读取时就是什么样。

SDS使用len属性的值，而不是空字符来判断字符串是否结束，它可以保存类似于下面的字符串：



通过使用二进制安全的SDS，而不是C字符串，使得Redis不仅可以保存文本数据，还可以保存任意格式的二进制数据。

虽然SDS的API都是二进制安全的，但它们一样遵循C字符串以空字符结尾的惯例：这些API总会将SDS保存的数据的末尾设置为’\0’，并且总会在为buf数组分配空间时多分配一个字节来容纳这个空字符。保持这样的惯例，是为了让那些保存纯文本数据的SDS可以重用一部分<string.h>中的函数。比如像下面这种用法：

strcasecmp**(**sds**->**buf**,** "hello world"**);**

strcat**(**c\_string**,** sds**->**buf**);**

SDS的函数实现都在src/sds.c中，代码都比较简单，没有涉及复杂的算法，带有注释的sds.c，可以参考：

https://github.com/gqtc/redis-3.0.5/blob/master/redis-3.0.5/src/sds.c