# 简单动态字符串

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，以下简称C字符串），而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 用作Redis 的默认字符串表示。

在Redis 里面， C 字符串只会作为字符串字面量（ s回ng literal ）用在一些无须对字符串值进行修改的地方，比如打印日志。

redisLog(REDIS\_WARNING,”Redis is now ready to exit, bye bye ...”) ;

当Redis 需要的不仅仅是一个字符串字面量，而是一个可以被修改的字符串值时， Redis就会使用SDS 来表示字符串值，比如在Redis 的数据库里面，包含字符串值的键值对在底层都是由SDS 实现的。

如果客户端执行命令：redis> set msg “hello world”

那么Redis 将在数据库中创建一个新的键值对，其中：

* 键值对的键是一个字符串对象，对象的底层实现是一个保存着字符串”msg”的SDS。
* 键值对的值也是一个字符串对象，对象的底层实现是一个保存着字符串”hello world ”的SDS 。

如果客户端执行命令：redis> RPUSH fruits "apple" "banana" "cherry”

那么Redis 将在数据库中创建一个新的键值对，其中：

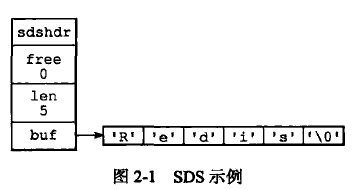
* 键值对的键是一个字符串对象，对象的底层实现是－斗保存了字符串”fruits ”的SDS。
* 键值对的值是一个列表对象，列表对象包含了三个字符串对象，这三个字符串对象分别由三个SDS 实现：第一个SDS 保存着字符串” apple ”，第二个SDS 保存着字符串＂ banana ”，第三个SDS 保存着字符串＂ cherry ”。

除了用来保存数据库中的字符串值之外， sos 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人缓冲区，都是由SDS 实现的。

## SDS 的定义

每个sds.h/sdshdr 结构表示一个SDS 值：

|  |
| --- |
| struct sdshdr {  // 记录buf 数组中已使用字节的数量  // 等于SDS 所保存字符串的长度  int len;  // 记录buf 数组中未使用字节的数量  int free;  // 字节数组，用于保存字符串  char buf []:  } |



SDS 遵循C 字符串以空字符结尾的惯例，保存空字符的1 字节空间不计算在SDS 的len 属性里面，并且为空字符分配额外的1 字节空间，以及添加空字符到字符串末尾等操作，都是由SDS 画数自动完成的，所以这个空字符对于SDS 的使用者来说是完全透明的。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画数。举个例子，如果我们有一个指向图2-1 所示SDS 的指针s ，那么我们可以直接使用<stdio.h>/printf 函数，通过执行以下语句来打印出SDS 保存的字符串值”Redis ”。

printf(“%d”, s->buf)

## SDS 与C 字符串的区到

根据传统， C 语言使用长度为N+l 的字符数组来表示长度为N 的字符串，并且字符数组的最后一个元素总是空字符’\0’。 C 语言使用的这种简单的字符串表示方式，并不能满足Redis 对字符串在安全性、效率以及功能方面的要求。

### 常数复杂度获取字符串长度

因为C 字符串并不记录自身的长度信息，所以为了获取一个C 字符串的长度，程序必须遍历整个字符串，对遇到的每个字符进俯＋数，直到遇到代表字符串结尾的空字符为止，这个操作的复杂度为O(N)的。和C 字符串不同，因为SDS 在len 属性中记录了SDS 本身的长度，所以获取一个SDS 长度的复杂度仅为O(1)。

通过使用SDS 而不是C 字符串， Redis 将获取字符串长度所需的复杂度从O(N) 降低到了O(1)，这确保了获取字符串长度的工作不会成为Redis 的性能瓶颈。例如，因为字符串键在底层使用SDS 来实现，所以即使我们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ST.缸，EN命令的复杂度仅为O(1)。

### 杜绝缓冲区溢出

C 字符串不记录自身长度带来的另一个问题是容易造成缓冲区溢出（ buffer overflow ）。因为C 字符串不记录自身的长度，所以C语言API strcat 方法假定用户在执行这个函数时，已经为dest 分配了足够多的内存，可以容纳src 字符串中的所有内容，而一旦这个假定不成立时，就会产生缓冲区溢出。

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 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，如果不满足的话， 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，所以使用SDS 既不需要手动修改SDS 的空间大小，也不会出现前面所说的缓冲区溢出问题。

### 减少修改字符串时带来的内存重分配次数

因为C 字符串并不记录自身的长度，所以对于一个包含了N 个字符的C 字符串来说，这个C 字符串的底层实现总是一个N+l 个字符长的数组（额外的一个字符空间用于保存空字符）。因为C 字符串的长度和底层数组的长度之间存在着这种关联性，所以每次增长或者缩短一个C 字符串，程序都总要对保存这个C 字符串的数组进行一次内存重分配操作：

* 如果程序执行的是增长字符串的操作，比如拼接操作（ append ），那么在执行这个操作之前，程序需要先通过内存重分配来扩展底层数组的空间大小一一如果忘了这一步就会产生缓冲区溢出。
* 如果程序执行的是缩短字符串的操作，比如截断操作（ trim ），那么在执行这个操作之后，程序需要通过内存重分配来释放字符串不再使用的那部分空间一一如果忘了这一步就会产生内存泄漏。

因为内存重分配涉及复杂的算法，并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：

* 在一般程序中，如果修改字符串长度的情况不太常出现，那么每次修改都执行一次内存重分配是可以接受的。
* 但是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，如果每次修改宇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。

为了避免C 字符串的这种缺陆， SDS通过未使用空间解除了字符串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。通过未使用空间， SDS 实现了空间预分配和惰性空间释放两种优化策略。

通过未使用空间， 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：

空间预分配用于优化sos 的字符串增长操作：当sos 的API 对一个SDS 进行修改，并且需要对sos 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由以下公式决定：

* 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1MB，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1MB，那么程序会分配lMB 的未使用空间。举个例子，如果进行修改之后， SDS 的len 将变成30MB，那么程序会分配1MB的未使用空间， SDS 的buf 数组的实际长度将为30 MB＋ 1MB＋ lbyte 。

在扩展SDS 空间之前， SDSAPI 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。通过这种预分配策略， 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。

#### 惰性空间释披

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A凹，让我们可以在有需要时，真正地释放SDS 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。

### 二进制安全

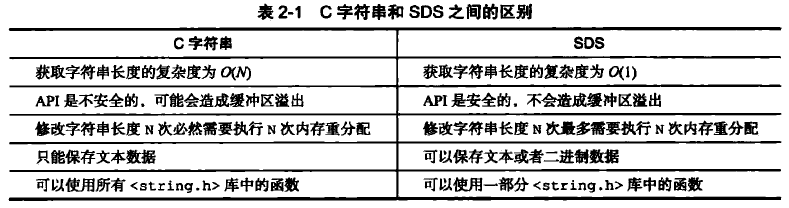
C 字符串中的字符必须符合某种编码（比如ASCII），并且除了字符串的末尾之外，字符串里面不能包含空字符，否则最先被程序读人的空字符将被误认为是字符串结尾，这些限制使得C 字符串只能保存文本数据，而不能保存像图片、音频、视频、压缩文件这样的二进制数据。

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，为了确保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 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，数据在写入时是什么样的，它被读取时就是什么样。这也是我们将SDS 的buf 属性称为字节数组的原因——Redis 不是用这个数组来保存字符，而是用它来保存一系列二进制数据。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。

### 兼容部分C 字符串函数

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空字符，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，这是为了让那些保存文本数据的SDS 可以重用一部分<string.h ＞库定义的函数，从而避免了不必要的代码重复。

### 总结



## SDS API

