【Redis5.X源码分析】系列之字符串

原创 王江华 华仔聊技术 2021-04-26 13:57

收录于话题
#面试 14 #redis 2

引入简单动态字符串

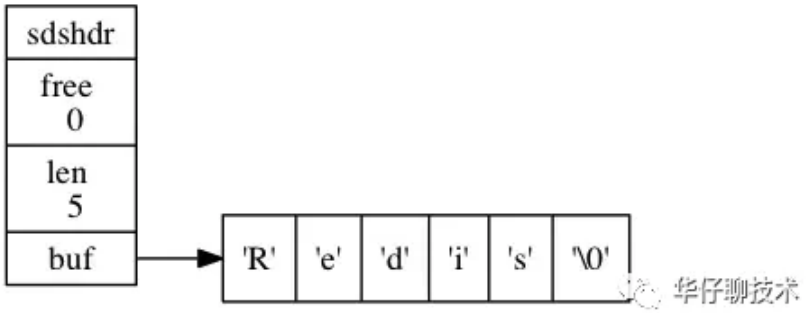
简单动态字符串（Simple Dynamic String 简称SDS）是Redis为了高效安全的存储而设计的一个数据结构，在兼容C语言标准的字符串处理函数基础上保证了二进制的安全。下面为大家一一揭晓SDS的神秘面纱

SDS新老数据结构的对比

在redis3.2.x之前，SDS的存储结构如下：

```
struct sdshdr {
    int len; //buf中已占用字节数 存字符串长度
    int free; //buf中剩余可用字节数 存字符串内容的柔性数组的剩余空间
    char buf[]; //柔性数组，真正存放字符串值
};
```

- 以“Redis”字符串为例，它在旧版SDS结构中是这样存储的：



> - 其中，free字段为0，代表buf字段没有剩余存储空间
> - len字段为5，代表字符串长度为5
> - buf字段存储真正的字符串内容“Redis”
> - 存储字符串内容的柔性数组占用内存大小为6字节，其余字段所占用8个字节（4+4+6 = 14字节）

在最新版本redis5.x中，为了进一步减少字符串存储过程中的内存占用，划分了5种适应不同字符串长度专用的存储结构：

```
在redis5源码中，对类型的宏定义如下(sds.h 76~80):
#define SDS_TYPE_5  0
#define SDS_TYPE_8  1
#define SDS_TYPE_16 2
#define SDS_TYPE_32 3
#define SDS_TYPE_64 4
#define SDS_TYPE_MASK 7  前3位 最大二进制111,十进制7
#define SDS_TYPE_BITS 3  sdshdr5的计算位的长度, 前3位是类型

对数据结构的定义如下（sds.h  45~74）
//存储长度为小于32的字符串
struct __attribute__ ((__packed__)) sdshdr5 {
    unsigned char flags; //低三位存储类型，高5位存储字符串长度，主要存储长度小于32的短字符串
    char buf[]; //存储字符串内容的柔性数组
};
//存储长度为小于256的字符串
struct __attribute__ ((__packed__)) sdshdr8 {
    uint8_t len; //已使用字符串长度，用1字节存储 （不包含C中字符串的结束符'\0'的长度1）
    uint8_t alloc; //已分配的总长度， 用1字节存储 减去len就是未使用的空间，初始时和len相等
    unsigned char flags; //标识是哪种存储类型 低3位存储类型，高5位预留
    char buf[]; //存储字符串内容的柔性数组
};
//存储长度为小于2^16的字符串
struct __attribute__ ((__packed__)) sdshdr16 {
    uint16_t len; //已使用字符串长度，用2字节存储 （不包含C中字符串的结束符'\0'的长度1）
    uint16_t alloc;  //已分配的总长度， 用2字节存储 减去len就是未使用的空间，初始时和len相等
    unsigned char flags; //标识是哪种存储类型 低3位存储类型，高5位预留
    char buf[]; //存储字符串内容的柔性数组
};
//存储长度为小于2^32的字符串这里有一点注意，若是机器的LONG_MAX不等于LLONG_MAX，则返回sdshdr64类型
struct __attribute__ ((__packed__)) sdshdr32 {
    uint32_t len; //已使用字符串长度，用4字节存储 （不包含C中字符串的结束符'\0'的长度1）
    uint32_t alloc;  //已分配的总长度， 用4字节存储 减去len就是未使用的空间，初始时和len相等
    unsigned char flags; //标识是哪种存储类型 低3位存储类型，高5位预留
    char buf[]; //存储字符串内容的柔性数组
};
//其他的都用此类存储
struct __attribute__ ((__packed__)) sdshdr64 {
    uint64_t len; //已使用字符串长度，用8字节存储 （不包含C中字符串的结束符'\0'的长度1）
    uint64_t alloc;  //已分配的总长度， 用8字节存储 减去len就是未使用的空间，初始时和len相等
    unsigned char flags; //标识是哪种存储类型 低3位存储类型，高5位预留
    char buf[]; //存储字符串内容的柔性数组
};
```

一个SDS字符串的完整结构，由在内存地址上前后相邻的两部分组成：

- 一个header。通常包含字符串的长度(len)、最大容量(alloc)和flags。sdshdr5有所不同。header信息中的定义这么多字段，其中一个很重要的作用就是实现对字符串的灵活操作并且尽量减少内存重新分配和回收操作。
- 一个字符数组。这个字符数组的长度等于最大容量+1。真正有效的字符串数据，其长度通常小于最大容量。在真正的字符串数据之后，是空余未用的字节（一般以字节0填充），允许在不重新分配内存的前提下让字符串数据向后做有限的扩展。在真正的字符串数据之后，还有一个NULL结束

符，即ASCII码为0的'\0'字符。这是为了和传统C字符串兼容。之所以字符数组的长度比最大容量多1个字节，就是为了在字符串长度达到最大容量时仍然有1个字节存放NULL结束符。

> - 从上面的定义我们可以看到，SDS结构从一种变成了五种，定义这5种不同的类型是为了尽量减少sdshdr占用的空间，这决定了这种结构能够最大存储多长的字符串（2^8 2^16 2^32 2^64）不考虑sdshdr5。
> - 另外聪明的你们可能会注意到这些结构体都带有attribute ((packed))关键字，它是告诉编译器不进行结构体的内存对齐。而在C/C++中，对一个结构体创建时，都会进行字节对齐操作，使得结构体的大小比其变量占用的字节要多一些（那样的话，就不能保证header和sds的数据部分紧紧前后相邻，也不能按照固定向低地址方向偏移1个字节的方式来获取flags字段了）。而Redis为了能直接从buf直接找到到flags，定义时在结构体声明中加上attribute((packed))， 强制不要按字节对齐(表示取消字节对齐，按照紧凑排列的方式)，这样不管是哪种类型的hdr，都可以用buf[-1]找到对应的flags。
> - 在各个header的定义中最后有一个char buf[]。我们注意到这是一个没有指明长度的字符数组，这是C语言中定义字符数组的一种特殊写法，称为柔性数组（柔性数组），只能定义在一个结构体的最后一个字段上。它在这里只是起到一个标记的作用，表示在flags字段后面就是一个字符数组，或者说，它指明了紧跟在flags字段后面的这个字符数组在结构体中的偏移位置。而程序在为header分配的内存的时候，它并不占用内存空间。如果计算sizeof(struct sdshdr16)的值，那么结果是5个字节，其中没有buf字段。
> - sdshdr5与其它几个header结构不同，它不包含alloc字段，而长度使用flags的高5位来存储。因此，它不能为字符串分配空余空间。如果字符串需要动态增长，那么它就必然要重新分配内存才行。所以说，这种类型的sds字符串更适合存储静态的短字符串（长度小于32）。

至此，我们非常清楚地看到了：sds字符串的header，其实隐藏在真正的字符串数据的前面（低地址方向）。这样的一个定义，有如下几个好处：

- header和数据相邻，而不用分成两块内存空间来单独分配。这有利于减少内存碎片，提高存储效率（memory efficiency）。
- 虽然header有多个类型，但sds可以用统一的char *来表达。且它与传统的C语言字符串保持类型兼容。如果一个sds里面存储的是可打印字符串，那么我们可以直接把它传给C函数，比如使用strcmp比较字符串大小，或者使用printf进行打印。

Redis设计SDS的原因以及与C的对比

在C语言中都是使用长度为N+1的字符串数组来表示长度为N的字符串；字符串数组的最后一个元素一定是'\0'。然而这种方式并不能满足Redis对字符串在安全性、功能性以及效率性的要求。

获取字符串长度时间复杂度o(1)

结构体中保存len属性，因此获取SDS字符串的长度只需要读取len属性，时间复杂度为O(1)。而对于 C 语言，获取字符串的长度通常是经过遍历计数来实现的，时间复杂度为O(n)。

二进制安全

C语言中，用'\0'表示字符串的结束，如果字符串中本身就有'\0'字符，那么字符串就会被截断，即非二进制安全，而SDS不是以空字符来判断结束，而是以len长度来判断结束，因此SDS是二进制安全的。因此buf[]不是用于保存字符，而是保存一系列二进制数据。

杜绝缓冲区溢出，边界检查 根据len长度检测是否结束

减少修改字符串长度时所需的内存重分配次数 后面会重点讲解

兼容部分C语言函数

可以使用部分C语言库中的函数，如<string.h>/strcasecmp等

SDS常用API代码分析

源码详细参考Redis的sds.h源码和sds.h的实现sds.c源码

- 接下来开始介绍sds相关的api函数，第一批是声明、定义在sds.h文件内的静态函数，这些函数都是针对动态字符串头部的属性的获取与修改，简单易懂

```
//通过结构体类型与字符串开始字节，获取到动态字符串头部的开始位置，并赋值给sh指针
#define SDS_HDR_VAR(T,s) struct sdshdr##T *sh = (void*)((s)-(sizeof(struct sdshdr##T)));
//通过类型与字符串开始字节，返回动态字符串头部的指针
#define SDS_HDR(T,s) ((struct sdshdr##T *)((s)-(sizeof(struct sdshdr##T))))
//获取SDS_TYPE_5类型的长度
#define SDS_TYPE_5_LEN(f) ((f)>>SDS_TYPE_BITS)
//获取动态字符串长度
static inline size_t sdslen(const sds s) {
    //获取头部信息中的flags属性，因内存紧密相连，可以直接通过这种方式获取
    unsigned char flags = s[-1];
    //获取类型，SDS_TASK_MASK为7，所以flags&SDS_TASK_MASK等于flags
    switch(flags&SDS_TYPE_MASK) {
        case SDS_TYPE_5:
            //SDS_TYPE_5类型的长度获取稍微不同，它的长度被定义在flags的高5位当中，具体可查看之后的sdsnewlen函
            return SDS_TYPE_5_LEN(flags);
        case SDS_TYPE_8:
            //SDS_HDR函数通过类型与字符串开始字节获取头部，以此获取字符串的长度
            return SDS_HDR(8,s)->len;
        case SDS_TYPE_16:
            return SDS_HDR(16,s)->len;
        case SDS_TYPE_32:
            return SDS_HDR(32,s)->len;
        case SDS_TYPE_64:
            return SDS_HDR(64,s)->len;
    }
    return 0;
}

//获取动态字符串的剩余内存
static inline size_t sdsavail(const sds s) {
```

```
    //获取flags
    unsigned char flags = s[-1];
    switch(flags&SDS_TYPE_MASK) {
        case SDS_TYPE_5: {
            //SDS_TYPE_5直接返回0,  不可扩展 字符串为0直接转成SDS_TYPE_8
            return 0;
        }
        case SDS_TYPE_8: {
            //通过SDS_HDR_VAR函数，将头部指针放置在sh变量
            SDS_HDR_VAR(8,s);
            //总内存大小 - 字符串长度，获取可用内存大小
            return sh->alloc - sh->len;
        }
        case SDS_TYPE_16: {
            SDS_HDR_VAR(16,s);
            return sh->alloc - sh->len;
        }
        case SDS_TYPE_32: {
            SDS_HDR_VAR(32,s);
            return sh->alloc - sh->len;
        }
        case SDS_TYPE_64: {
            SDS_HDR_VAR(64,s);
            return sh->alloc - sh->len;
        }
    }
    return 0;
}

//重置字符串长度
static inline void sdssetlen(sds s, size_t newlen) {
    //获取flags
    unsigned char flags = s[-1];
    switch(flags&SDS_TYPE_MASK) {
        case SDS_TYPE_5:
            {
                //SDS_TYPE_5的长度设置较为特殊，长度信息写在flags的高5位
                unsigned char *fp = ((unsigned char*)s)-1;
                *fp = SDS_TYPE_5 | (newlen << SDS_TYPE_BITS);
            }
            break;
        //其他类型则是统一通过newlen来修改len属性的值
        case SDS_TYPE_8:
            SDS_HDR(8,s)->len = newlen;
            break;
        case SDS_TYPE_16:
            SDS_HDR(16,s)->len = newlen;
            break;
        case SDS_TYPE_32:
            SDS_HDR(32,s)->len = newlen;
            break;
        case SDS_TYPE_64:
            SDS_HDR(64,s)->len = newlen;
            break;
    }
}

//按照指定数值，增加字符串长度
static inline void sdsinclen(sds s, size_t inc) {
    //获取flags
    unsigned char flags = s[-1];
    switch(flags&SDS_TYPE_MASK) {
        case SDS_TYPE_5:
            {
```

```
                //SDS_TYPE_5类型使用SDS_TYPE_5_LEN函数，获取长度、更新、设置
                unsigned char *fp = ((unsigned char*)s)-1;
                unsigned char newlen = SDS_TYPE_5_LEN(flags)+inc;
                *fp = SDS_TYPE_5 | (newlen << SDS_TYPE_BITS);
            }
            break;
        //其他类型则直接通过SDS_HDR函数，更新len值
        case SDS_TYPE_8:
            SDS_HDR(8,s)->len += inc;
            break;
        case SDS_TYPE_16:
            SDS_HDR(16,s)->len += inc;
            break;
        case SDS_TYPE_32:
            SDS_HDR(32,s)->len += inc;
            break;
        case SDS_TYPE_64:
            SDS_HDR(64,s)->len += inc;
            break;
    }
}

/* sdsalloc() = sdsavail() + sdslen() 获取动态字符串的总内存*/
static inline size_t sdsalloc(const sds s) {
    //获取flags
    unsigned char flags = s[-1];
    switch(flags&SDS_TYPE_MASK) {
        //SDS_TYPE_5直接通过SDS_TYPE_5_LEN函数返回
        case SDS_TYPE_5:
            return SDS_TYPE_5_LEN(flags);
        //其他类型则通过SDS_HDR返回头部信息中的alloc属性
        case SDS_TYPE_8:
            return SDS_HDR(8,s)->alloc;
        case SDS_TYPE_16:
            return SDS_HDR(16,s)->alloc;
        case SDS_TYPE_32:
            return SDS_HDR(32,s)->alloc;
        case SDS_TYPE_64:
            return SDS_HDR(64,s)->alloc;
    }
    return 0;
}

//重置字符串内存大小
static inline void sdssetalloc(sds s, size_t newlen) {
    //获取flags
    unsigned char flags = s[-1];
    switch(flags&SDS_TYPE_MASK) {
        case SDS_TYPE_5:
            //官方注释，SDS_TYPE_5不做任何操作
            /* Nothing to do, this type has no total allocation info. */
            break;
        //其他类型通过SDS_HDR直接修改头部信息中的alloc属性
        case SDS_TYPE_8:
            SDS_HDR(8,s)->alloc = newlen;
            break;
        case SDS_TYPE_16:
            SDS_HDR(16,s)->alloc = newlen;
            break;
        case SDS_TYPE_32:
            SDS_HDR(32,s)->alloc = newlen;
            break;
        case SDS_TYPE_64:
            SDS_HDR(64,s)->alloc = newlen;
```

```
            break;
        }
    }

    sds的一些基础函数
    sdslen(const sds s): 获取sds字符串长度。
    sdssetlen(sds s, size_t newlen): 设置sds字符串长度。
    sdsinclen(sds s, size_t inc): 增加sds字符串长度。
    sdsalloc(const sds s): 获取sds字符串容量。
    sdssetalloc(sds s, size_t newlen): 设置sds字符串容量。
    sdsavail(const sds s): 获取sds字符串空余空间（即alloc - len）。
    sdsHdrSize(char type): 根据header类型得到header大小。
    sdsReqType(size_t string_size): 根据字符串数据长度计算所需要的header类型。
```

- 第二批是声明,定义在sds.c中的静态方法

```
    const char *SDS_NOINIT = "SDS_NOINIT";
    //获取结构体头部大小
    static inline int sdsHdrSize(char type) {
        switch(type&SDS_TYPE_MASK) {
            case SDS_TYPE_5:
                return sizeof(struct sdshdr5);
            case SDS_TYPE_8:
                return sizeof(struct sdshdr8);
            case SDS_TYPE_16:
                return sizeof(struct sdshdr16);
            case SDS_TYPE_32:
                return sizeof(struct sdshdr32);
            case SDS_TYPE_64:
                return sizeof(struct sdshdr64);
        }
        return 0;
    }
    //根据长度确定类型
    static inline char sdsReqType(size_t string_size) {
        if (string_size < 1<<5)
            return SDS_TYPE_5;
        if (string_size < 1<<8)
            return SDS_TYPE_8;
        if (string_size < 1<<16)
            return SDS_TYPE_16;
    #if (LONG_MAX == LLONG_MAX)
        if (string_size < 1ll<<32)
            return SDS_TYPE_32;
        return SDS_TYPE_64;
    #else
        return SDS_TYPE_32;
    #endif
    }
```

创建字符串

- Redis通过sdsnewlen函数来创建SDS，其中会根据字符串长度来选择合适的类型：

```c
/* 创建新字符串方法，传入目标长度，初始化方法 */
sds sdsnewlen(const void *init, size_t initlen) {
    void *sh;
    sds s;
    //根据字符串长度来选择不同的类型
    char type = sdsReqType(initlen);
    /* 对于空字符串来说 SDS_TYPE_5  强制转换成 SDS_TYPE_8 如果要创建一个长度
     * 为0的空字符串，那么不使用SDS_TYPE_5类型的header，而是转而使用
     * SDS_TYPE_8类型的header。这是因为创建的空字符串一般接下来的操作很可能是
     * 追加数据，但SDS_TYPE_5类型的sds字符串不适合追加数据（会引发内存重新分配）*/
    if (type == SDS_TYPE_5 && initlen == 0) type = SDS_TYPE_8;
    //计算不同类型头部需要的长度 1，2，4，8
    int hdrlen = sdsHdrSize(type);
    unsigned char *fp;  //指向flags的指针

    sh = s_malloc(hdrlen+initlen+1);  //申请足够的内存，头部大小+初始化大小+结尾符‘\0’
    if (init==SDS_NOINIT)
        init = NULL;
    else if (!init)
        memset(sh, 0, hdrlen+initlen+1);  //重新清空内存，目的就是void *sh; 是未知的,所以需要清空，不然会有
    if (sh == NULL) return NULL;
    s = (char*)sh+hdrlen;  //获取字符串起始字节 s是指向buf的指针
    fp = ((unsigned char*)s)-1; //s是柔性数组buf的指针，-1即指向flags
    switch(type) {
       //SDS_TYPE_5 有单独的初始化方案
        case SDS_TYPE_5: {
            *fp = type | (initlen << SDS_TYPE_BITS);
            break;
        }
       //其他类型通过SDS_HDR_VAR函数获取头部信息，并逐步初始化
        case SDS_TYPE_8: {
            SDS_HDR_VAR(8,s);
            sh->len = initlen;
            sh->alloc = initlen;
            *fp = type;
            break;
        }
        case SDS_TYPE_16: {
            SDS_HDR_VAR(16,s);
            sh->len = initlen;
            sh->alloc = initlen;
            *fp = type;
            break;
        }
        case SDS_TYPE_32: {
            SDS_HDR_VAR(32,s);
            sh->len = initlen;
            sh->alloc = initlen;
            *fp = type;
            break;
        }
        case SDS_TYPE_64: {
            SDS_HDR_VAR(64,s);
            sh->len = initlen;
            sh->alloc = initlen;
            *fp = type;
            break;
        }
    }
    //根据init与initlen，将内容复制给字符串
    if (initlen && init)
        memcpy(s, init, initlen);
    s[initlen] = '\0';  //添加末尾的结束符
```

```
    return s;
}
```

释放字符串

```
/* Free an sds string. No operation is performed if 's' is NULL. */
void sdsfree(sds s) {
    if (s == NULL) return;
    //通过对s的偏移,定位到SDS结构体的首部,并通过s_free直接释放
    s_free((char*)s-sdsHdrSize(s[-1]));
}
```

惰性空间释放

- 惰性空间释放主要是用于优化 SDS 的字符串截取或缩短操作。当 SDS 的 API 需要缩短 SDS 保存的字符串时，程序并不立即回收缩短后多出来的字节。如果将来要对 SDS 进行增长操作的话，这些未使用空间就可能会派上用场。比如我们将“hello world”缩短为“hello”，然后又改成“heoll world”，这样，如果我们立刻回收缩短后多出来的字节，然后再重新分配内存空间，是非常浪费时间的。如果等待一段时间之后再回收，可以很好地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扩展空间

```
/* Modify an sds string in-place to make it empty (zero length).
 * However all the existing buffer is not discarded but set as free space
 * so that next append operations will not require allocations up to the
 * number of bytes previously available. */
//为了优化性能减少申请开销,提供了不直接释放内存,而是通过重置统计值达到清空的方法
void sdsclear(sds s) {
    sdssetlen(s, 0);  //SDS统计len值归0, 已存在的buf没有真正被清除,新数据可以覆盖而不需要重新申请内存
    s[0] = '\0'; 清空buf
}
```

空间预分配

- SDS结构体中有个alloc,它是目前给存储字符串的柔性数组总共分配了多少字节的空间。那么记录这个字段的作用何在呢？那就是空间预分配和惰性空间释放的设计思想了

```
/*
 描述：对字符串扩展空间
 参数：
    IN：
        s：需要扩展的字符串
        addlen：扩展长度
 返回值：
    成功：返回扩展后的sds
    失败：NULL
```

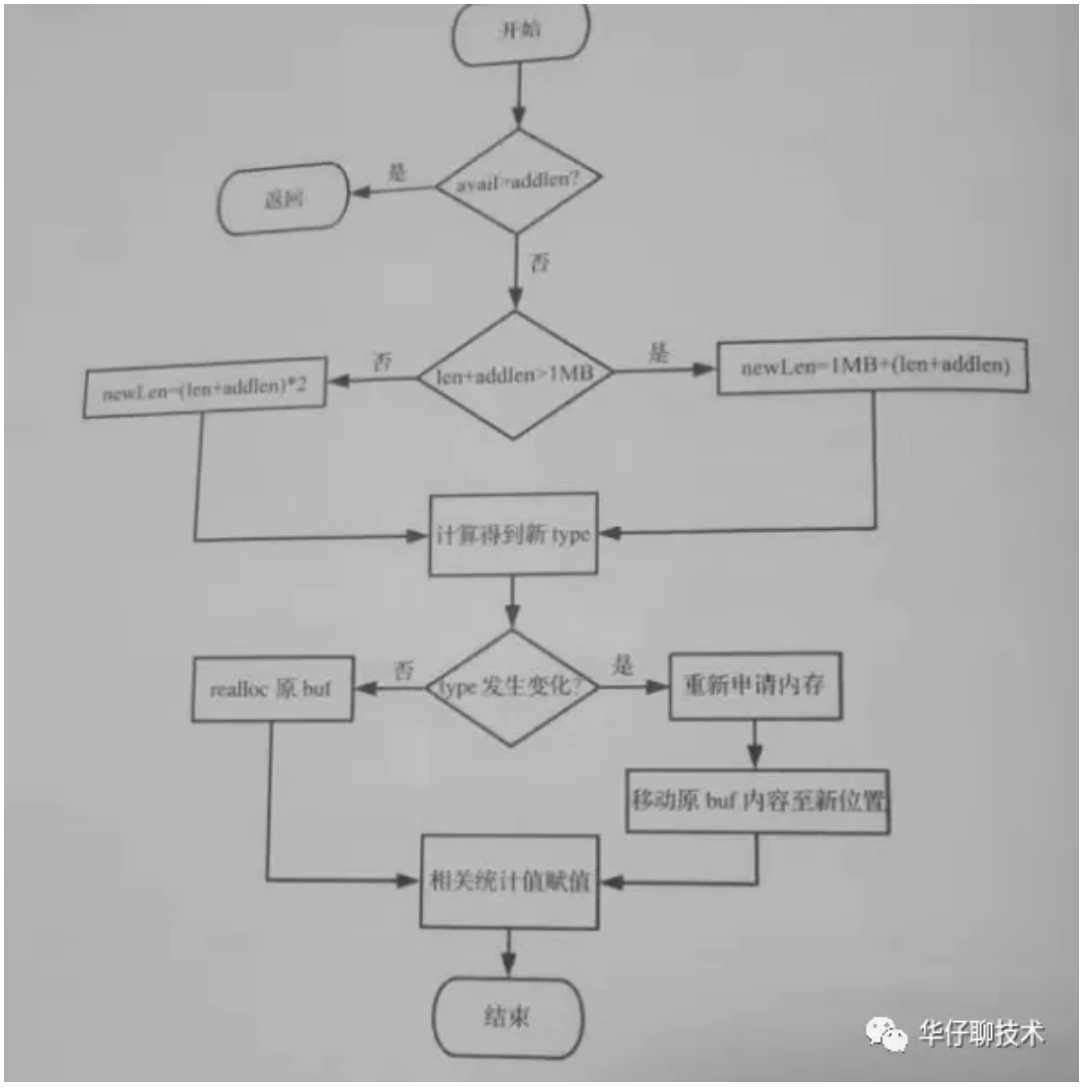
```
 */
sds sdsMakeRoomFor(sds s, size_t addlen) {
    void *sh, *newsh;
    size_t avail = sdsavail(s);  //获取当前字符串可用剩余空间
    size_t len, newlen;
    char type, oldtype = s[-1] & SDS_TYPE_MASK;
    int hdrlen;

    /* Return ASAP if there is enough space left. */
    //如果可用空间大于追加部分的长度，说明当前字符串还有额外的空间，足够容纳扩容
    //后的字符串，不用分配额外空间，直接返回
    if (avail >= addlen) return s;

    len = sdslen(s);
    sh = (char*)s-sdsHdrSize(oldtype);
    newlen = (len+addlen);
    /*
      SDS_MAX_PREALLOC = 2014*1024 = 1M
      如果扩容后的<1M,那么新的字符串为扩容后的2倍
      如果>=1M,那么新的字符串加上1M
    */
    if (newlen < SDS_MAX_PREALLOC)
        newlen *= 2; //SDS_MAX_PREALLOC = 1MB，如果扩容后的长度小于1MB，直接额外分配扩容后字符串长度*2的空间
    else
        newlen += SDS_MAX_PREALLOC; //扩容后长度大于等于1MB，额外分配扩容后字符串+1MB的空间
    //长度变了，需要重新获取新字符串的type
    type = sdsReqType(newlen);

    /* Don't use type 5: the user is appending to the string and type 5 is
     * not able to remember empty space, so sdsMakeRoomFor() must be called
     * at every appending operation. */
    if (type == SDS_TYPE_5) type = SDS_TYPE_8;

    hdrlen = sdsHdrSize(type);
    //和oldtype比较，然后根据情况分配空间
    if (oldtype==type) {
        newsh = s_realloc(sh, hdrlen+newlen+1);
        if (newsh == NULL) return NULL;
        s = (char*)newsh+hdrlen;
    } else {
        /* Since the header size changes, need to move the string forward,
         * and can't use realloc */
        newsh = s_malloc(hdrlen+newlen+1);
        if (newsh == NULL) return NULL;
        memcpy((char*)newsh+hdrlen, s, len+1);
        //释放sh指针，其实是释放s
        s_free(sh);
        //sh = (char*)s-sdsHdrSize(oldtype); 为s重新指向新分配的newsh
        s = (char*)newsh+hdrlen;  //重新给s赋值
        s[-1] = type;
        sdssetlen(s, len);
    }
    sdssetalloc(s, newlen);
    return s;
}
```



字符串扩展空间分析 摘自陈雷老师的Redis5设计与源码分析

其他API

```
/*
 描述：把长度为len的字符串t连接到s，类似于strcat
 参数：
    IN：
        s：目标字符串
        t：源字符串
        len：t的长度
返回值：
    成功：新的sds
    失败：NULL
 */
sds sdscatlen(sds s, const void *t, size_t len) {
    size_t curlen = sdslen(s);

    s = sdsMakeRoomFor(s,len);  //扩展连接字符串的长度
    if (s == NULL) return NULL;
    memcpy(s+curlen, t, len);
    sdssetlen(s, curlen+len);
    s[curlen+len] = '\0';
    return s;
}

//描述：格式化字符串
sds sdscatvprintf(sds s, const char *fmt, va_list ap) {
```

```
    va_list cpy;
    char staticbuf[1024], *buf = staticbuf, *t;
    size_t buflen = strlen(fmt)*2;

    /* We try to start using a static buffer for speed.
     * If not possible we revert to heap allocation. */
   /*
    * 这里先用栈区的staticbuf，如果fmt的2倍长度超过这个staticbuf在从堆去分配
    * 这样做如果是小于1024的，直接用栈区的内存，非常快
    * 这是预分配冗余空间的惯用手段，减小对内存的频繁分配
    */
    if (buflen > sizeof(staticbuf)) {
        buf = s_malloc(buflen);
        if (buf == NULL) return NULL;
    } else {
        buflen = sizeof(staticbuf);
    }

    /* Try with buffers two times bigger every time we fail to
     * fit the string in the current buffer size. */
    while(1) {
        buf[buflen-2] = '\0';//设置倒数第二个字符为结束符，
        //方便后面判断是否超过了最终长度
        va_copy(cpy,ap);
        vsnprintf(buf, buflen, fmt, cpy); //调用是的vsprintf家族函数
        va_end(cpy);
        if (buf[buflen-2] != '\0') {  //说明已经写满了，需要重新分配2倍大小，继续写
            if (buf != staticbuf) s_free(buf);
            buflen *= 2;
            buf = s_malloc(buflen);
            if (buf == NULL) return NULL;
            continue;
        }
        break;
    }

    /* Finally concat the obtained string to the SDS string and return it. */
    t = sdscat(s, buf);  //这里底层调用的sdscatlen，是安全的
    if (buf != staticbuf) s_free(buf);
    return t;
}

/*
 描述：更高效的格式化字符串，没有调用vsprintf家族函数
 */
sds sdscatfmt(sds s, char const *fmt, ...) {
    size_t initlen = sdslen(s);
    const char *f = fmt;
    long i;
    va_list ap;

    va_start(ap,fmt);
    f = fmt;    /* Next format specifier byte to process. */
    i = initlen; /* Position of the next byte to write to dest str. */
    while(*f) {
        char next, *str;
        size_t l;
        long long num;
        unsigned long long unum;

        /* Make sure there is always space for at least 1 char. */
        //判断是否有 可用空间,没有的话扩展
        if (sdsavail(s)==0) {
            s = sdsMakeRoomFor(s,1);
```

```
}

switch(*f) {
case '%':
    next = *(f+1);
    f++;
    switch(next) {
    case 's':
    case 'S':
        str = va_arg(ap,char*);
        //计算长度
        l = (next == 's') ? strlen(str) : sdslen(str);
        //如果可用空间不够，扩展
        if (sdsavail(s) < l) {
            s = sdsMakeRoomFor(s,l);
        }
        memcpy(s+i,str,l);
        sdsinclen(s,l);  //增加len的长度l
        i += l;
        break;
    case 'i':
    case 'I':
        if (next == 'i')
            num = va_arg(ap,int);
        else
            num = va_arg(ap,long long);
        {
            char buf[SDS_LLSTR_SIZE];
            //逻辑处理同上，把long long 转为str
            l = sdsll2str(buf,num);
            if (sdsavail(s) < l) {
                s = sdsMakeRoomFor(s,l);
            }
            memcpy(s+i,buf,l);
            sdsinclen(s,l);
            i += l;
        }
        break;
    case 'u':
    case 'U':
        if (next == 'u')
            unum = va_arg(ap,unsigned int);
        else
            unum = va_arg(ap,unsigned long long);
        {
            char buf[SDS_LLSTR_SIZE];
             //逻辑处理同上，把unsignt long long 转为str
            l = sdsull2str(buf,unum);
            if (sdsavail(s) < l) {
                s = sdsMakeRoomFor(s,l);
            }
            memcpy(s+i,buf,l);
            sdsinclen(s,l);
            i += l;
        }
        break;
    default: /* Handle %% and generally %<unknown>. */
        //除了上面的字符，其他的默认支持，即使是%
        s[i++] = next;
        sdsinclen(s,1);
        break;
    }
    break;
default:
```

```
            s[i++] = *f;
            sdsinclen(s,1);
            break;
        }
        f++;
    }
    va_end(ap);

    /* Add null-term */
    s[i] = '\0';
    return s;
}

/*
 * 除去字符串s中在字符串cset中出现的所有字符
 */
sds sdstrim(sds s, const char *cset) {
    char *start, *end, *sp, *ep;
    size_t len;

    sp = start = s;
    ep = end = s+sdslen(s)-1;
    while(sp <= end && strchr(cset, *sp)) sp++;
    while(ep > sp && strchr(cset, *ep)) ep--;
    len = (sp > ep) ? 0 : ((ep-sp)+1);
    if (s != sp) memmove(s, sp, len);
    s[len] = '\0';
    sdssetlen(s,len);
    return s;
}

/*
 *  取区间[start,end]的字符串，下标从0开始，-1表示最后一个
 */
void sdsrange(sds s, ssize_t start, ssize_t end) {
    size_t newlen, len = sdslen(s);

    if (len == 0) return;
    if (start < 0) {
        start = len+start;
        if (start < 0) start = 0;
    }
    if (end < 0) {
        end = len+end;
        if (end < 0) end = 0;
    }
    newlen = (start > end) ? 0 : (end-start)+1;
    if (newlen != 0) {
        if (start >= (ssize_t)len) {
            newlen = 0;
        } else if (end >= (ssize_t)len) {
            end = len-1;
            newlen = (start > end) ? 0 : (end-start)+1;
        }
    } else {
        start = 0;
    }
    if (start && newlen) memmove(s, s+start, newlen);
    s[newlen] = 0;
    sdssetlen(s,newlen);
}
```