第4章-武技-常见轮子前仆

本章是关于系统中常见出现的轮子的简单介绍. 构建框架中最基础最简单的组件. 保障咱们'战斗'过程中的生命线. 定位是练气期的武技, 融合了那些在妖魔大战中无数前辈们的英魄构建的套路. 武技的宗旨就是让你成为战场上能苟住能偷袭的小强 ┗| ` O'|┛ . 嗷 ~

4.1 那些年写过的日志库

用过很多日志库轮子, 也写过不少. 见过漫天飞花, 也遇到过一个个地狱火撕裂天空, 最后展示核心代码不足 20 行的简单的日志库, 来追求最简单的美好. 越简单越优美越让人懂的代码总会出彩, 不是吗? 一个高性能的日志库突破点无外乎

- 1' 缓存
- 2' 无锁
- 3' 定位

那开始对这个日志武技轮子, 深入剖析.

4.1.1 小小日志库

先看接口 **log.h** 设计部分, 感受几个宏横扫幺蛾子的玄幻.

```c
#pragma once

#include <errno.h>
#include <stdlib.h>
#include <string.h>

#include "times.h"

extern FILE * log_instance;

//
// LOG_PRINT - 构建拼接输出的格式串
// V          : 标识 日志前缀串必须 "" 包裹
// X          : fmt 自己要打印的串, 必须 "" 包裹
// return     : void
//
#define LOG_PRINT(V, X, ...)                                                \
fprintf(log_instance, "[%s]"V"[%s:%s:%d][%d:%s]"X"\n", times(),             \
    __FILE__, __func__, __LINE__, errno, strerror(errno), ##__VA_ARGS__)

//
// log 有些朴实, 迅速 ~
//
#define LOG_ERROR(fmt, ...) LOG_PRINT("[ERROR]", fmt, ##__VA_ARGS__)
#define LOG_INFOS(fmt, ...) LOG_PRINT("[INFOS]", fmt, ##__VA_ARGS__)
#ifndef NDEBUG
#define LOG_DEBUG(fmt, ...) LOG_PRINT("[DEBUG]", fmt, ##__VA_ARGS__)
#else
#define LOG_DEBUG(fmt, ...) /*  (^_-)☆ */
#endif
```

log.h 继承自 **times.h**, 唯一依赖的是其中 **times_fmt** 接口. 协助得到特定时间格式串, 填充到日志的头部.

```
// TIMES_FMT_STR - "{年}.{月}.{日}.{时}.{分}.{秒}.{纳秒} {时区}"
#define TIMES_FMT_STR "%04d-%02d-%02d %02d:%02d:%02d.%09ld %s"

// times TLS time str 版本
extern const char * times(void) {
    static _Thread_local times_t out;

    struct tm m;
    struct timespec s;

    timespec_get(&s, TIME_UTC);
    localtime_get(&m, s.tv_sec);

    sprintf(out, TIMES_FMT_STR,
            m.tm_year + 1900, m.tm_mon + 1, m.tm_mday,
            m.tm_hour, m.tm_min, m.tm_sec,
            s.tv_nsec, tzname[0]);

    return out;
}
```

日志库小小核心构造源码 **log.c** 展开:

```
#include "log.h"

FILE * log_instance;

// log_init 日志库初始化
void log_init(const char * path) {
    if ((log_instance = fopen(path, "ab")) == NULL) {
        // log 初始化失败, 程序默认启动失败.
        fprintf(stderr, "fopen ab error %"PRId64", %s\n", time(NULL), path);
        exit(EXIT_FAILURE);
    }
}
```

其中 log_init 可以通过 EXTERN_RUN 在 main 中初始化注册.

```
//
// EXTERN_RUN - 函数包装宏，声明并立即使用
// frun     : 需要执行的函数名称
// ...      : 可变参数，保留
//
#define EXTERN_RUN(frun, ...)                           \
do {                                                    \
    extern void frun();                                 \
    frun (__VA_ARGS__);                                 \
} while(0)

EXTERN_RUN(log_init, LOG_PATH_STR);
```

是不是很恐怖, 一个日志库这就完了. fprintf 是系统库输出函数, 默认自带缓冲机制. 缓冲说白了就是批量处理, 存在非及时性. vsnprintf 属于 printf 函数簇, 自带文件锁. 有兴趣的可以详细研究 **printf**, C 入门最早用的函数, 也是最复杂的函数之一. 那目前就差生成业务了! 也就是第三点定位, 这也是小小日志库的另一个高明之处, 借天罚来隔绝妖魔鬼怪.

4.1.2 小小 VT 二连

先在 Linux 平台构建一下测试环境. 模拟一个妖魔大战的场景 ~ 嗖 ~ 依次看下去

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define PATH_STR   "simplec.log"

//
// logrotate hello world
//
int main(int argc, char * argv[]) {
    FILE * log = fopen(PATH_STR, "ab");
    if (NULL == log) {
        fputs("fopen ab err =%d, path = " PATH_STR "!\n", errno, stderr);
        exit(EXIT_FAILURE);
    }

    // Ctrl + C 中断结束
    for (int id = 0; ; ++id) {
        printf(PATH_STR" id = %d\n", id);
        fprintf(log, PATH_STR" id = %d\n", id);
        fflush(log);
        sleep(1);
    }

    return fclose(log);
}
```

顺带给个编译文件 Makefile

```
.PHONY : all clean

all : simplec.exe

clean :
        -rm -rf *~
        -rm -rf simplec.exe
        -rm -rf simplec.log simplec.log-*

simplec.exe : simplec.c
        gcc -O2 -g -Wall -Wextra -Werror -o $@ $^
```

通过 make 得到 simplec.exe 运行起来, 就开始持续在日志文件中输出. 有关试炼场的环境已经搭建完成. 那么是时候主角 **logrotate** 出场了. 很久前在 centos 测试构建过看图:

```
[ceshi@test_HuNanMJ logrote]$ make
gcc -g -Wall -O2 -o simplec.exe simplec.c
[ceshi@test_HuNanMJ logrote]$ ls
Makefile  simplec.c  simplec.exe
[ceshi@test_HuNanMJ logrote]$ su root
Password:
[root@test_HuNanMJ logrote]# yum install logrotate crontabs
Loaded plugins: fastestmirror
Setting up Install Process
```

安装好 logrotate 和 crontabs 工具, 那么日志轮询器就能够开始使用了. 推荐自己查相关手册, 我这里只带大家弄个简单 Demo. Ok 开始搞起来, 看下面所做的 shell 批处理:

```
su root

cd /etc/logrotate.d
vi simplec
i

#
# 添加 logrotate 规则
# daily : 日志文件将按天轮循
# rotate 7 : 存档7次，时间最久的删除，LRU
# dateext : 日志添加日期后缀名
# copytruncate : 复制截断，(懒得写 SIGHUP 信号处理解决方案)
# create 644 root root : 创建的日志文件权限
#
# size = 100 : 测试用的，超过 100B 就生成日志备份文件，单位 K，M 默认 B
#
/home/ceshi/wangzhi/logrote/simplec.log {
        daily
        rotate 7
        dateext
        copytruncate
        create 644 root root
        size = 100
}

Esc
:wq!

logrotate -vf /etc/logrotate.d/simplec
```

copytruncate 复制截断存在一个隐患是 logrotate 在 copy 后 truncate 时候会丢失那一瞬间新加的日志. 如果不想日志发生丢失, 可以自行加重实现, 最终取舍在于你对于业务的认识和取舍. 此刻所搭建的环境:

```
[root@test_HuNanMJ logrote]# ll
total 28
-rw-rw-r-- 1 ceshi ceshi   181 Aug 28 16:55 Makefile
-rw-rw-r-- 1 ceshi ceshi   515 Aug 28 16:16 simplec.c
-rwxrwxr-x 1 ceshi ceshi 10582 Aug 28 16:56 simplec.exe
-rw-rw-r-- 1 ceshi ceshi   100 Aug 28 16:56 simplec.log
-rw-rw-r-- 1 ceshi ceshi   370 Aug 28 16:56 simplec.log-20170828
[root@test_HuNanMJ logrote]#
```

如果你有幸遇到贵人, 也只会给你一条路, 随后就是自己双手双脚的主场. 如果没有那么是时候 -> 冲冲冲, 四驱兄弟在心中 ~

以往小小 VT 二连之后, 可以再 A 一下. 那就利用自带的定时器了, 例如 crontabs 等等, 这些事情那就留给专业工具做吧. 以上是精简的日志库简单架构. 对于普通选手可能难以吹 NB(说服别人), 因而这里会再来分析一波所见过日志库的套路, 知彼知己选择才会更贴合. 日志库大体实现还存在一种套路, 开个线程跑日志消息队列. 这类日志库在游戏服务器中极其常见, 例如端游中大量日志打印, 运维备份的时候, 同步日志会将业务机卡死(日志无法写入, 玩家业务挂起). 所以构造出消息队列来缓存日志. 此类日志库可以秀一下代码功底, 毕竟线程轮询, 消息队列, 资源竞争, 对象池, 日志构建这些都需要有. 个人看法他很重. 难有摘叶伤人来的迅捷呀. 其缓冲层消息队列, 还不一定比不进行 fflush 的系统层面输出接口来的快捷. 而且启动一个单独线程处理日志, 那么就一定重度依赖对象池. 一环套一环, 收益普通 ~ 业务设计的时候能不用线程就别用. 因为线程脾气可大了, 还容易琢磨不透. 到这也扯的差不多了, 如果以后和人交流的时候, 被问到这个日志库为什么高效. 记住

- 1' 无锁编程, 利用 fprintf IO 锁
- 2' fprintf 最大限度利用系统 IO 缓冲层, 没必要 fflush, 从消息队列角度分析
- 3' 各司其职, 小小日志库只负责写, 其他交给系统层面最合适的工具搞. 定位单一

随着日志业务和日志库接触多了, 我们决策时候, 首先基于日志业务**定位**, 承载和联动的**功能**. 选择或者构建出当下很合适, 满足核心需求, 就很不错, 各有各的好 ~

4.2 开胃点心, 高效随机数库

为什么来个随机数库呢? 因为不同平台的随机数实现不一样, 有些期望结果可用性差异很大. 顺便嫌弃系统 rand 函数不够安全并且低效. 随机函数算法诞生对于计算机行业的发展真不得了, 奠定了人类模拟未知的一种可能. 随机和期望非常有意思, 在概率分析学上一种神奇的常识是: "概率为 0 的事情, 也可能发生 "! 有点呵呵参照无穷小, 非标准分析中可能有答案. 数学的诞生与推动不仅仅是为了解决具体遇到问题, 多数是人内部思维的升华 -> 自己爽就好了. 就如同这个时代最强数学家俄罗斯[格里戈里·佩雷尔曼]渡劫真君(注: 渡劫 > 化神), 嗨了一发就影响了整个人类思维的跳跃. 我们的随机函数算法是从 redis 源码上拔下来的, redis 是从 pysam 源码上拔下来. 可以算是薪火相传, 生生不息, 哭 ~ 首先看 **rand.h** 接口设计.

```c
#pragma once

#include <stdint.h>
#include <assert.h>

//
// 传承(抄袭)不灭(创新) rand 库
// 大体思路
//  { r | r[n+1] = (a*r[n] + c) mod m
//      , n >= 0
//      , m = 0xFFFFFFFFFFFF = 2 ^ 48
//      , a = 0x0005DEECE66D = 0x5DEECE66D,
//      , c = 0x000B         = 0xB
//  }
//
struct rand {
    uint32_t x0, x1, x2;
    uint32_t a0, a1, a2;
    uint32_t c;
};

typedef struct rand rand_t[1];

//
// rand_init - 随机函数对象初始化种子
// r         : 随机函数对象
// seed      : 种子数
// return    : void
//
extern void rand_init(rand_t r, int64_t seed);

//
// rand_get  - 获取一个随机值
// r         : 随机函数对象
// return    : 返回 [0, INT32_MAX] 随机数
//
extern int32_t rand_get(rand_t r);

//
// r_rand - 得到 [0, INT32_MAX] 随机数
// r_ranb - 得到 [0, INT64_MAX] (int64 = big int32) 随机数
// r_rang - 得到 range [min, max] 随机数
//
extern int32_t r_rand(void);

extern int64_t r_ranb(void);

inline int32_t r_rang(int32_t min, int32_t max) {
    assert(max >= min);
    return r_rand() % (max - min + 1) + min;
```

```
}
```

其中工程实现用到离散数学原理, 感兴趣同学可以查阅源码, 网上搜相关原理解释文章, 我们不做过多介入, 也介入不好. 我们宗旨是重工程部分, 培养开发工程师工程能力. 也没法介入研发工程师培养. 我们的小册子为什么成篇的刷代码? 主要是让你一个个对着敲到你的本地, 培养手感, 当然如果找出作者错误 ~ 非常的感谢 ❤

其实我们写的也是个伪随机数, 他算法核心依赖前期输入种子. **seed** 种子确定了, 算法输出值就确定了, 反过来有特别多算法输出值也能够模拟出种子 **seed**, 这就是伪随机由来.

```
//
// rand_init - 随机函数对象初始化种子
// r         : 随机函数对象
// seed      : 种子数
// return    : void
//
inline void rand_init(rand_t r, int64_t seed) {
    r->x0 = X0; r->x1 = LOW(seed); r->x2 = HIGH(seed);
    r->a0 = A0; r->a1 = A1; r->a2 = A2;
    r->c = C;
}
```

代码整体解决随机数库的平台无关性 ~ 目前多平台测试良好.

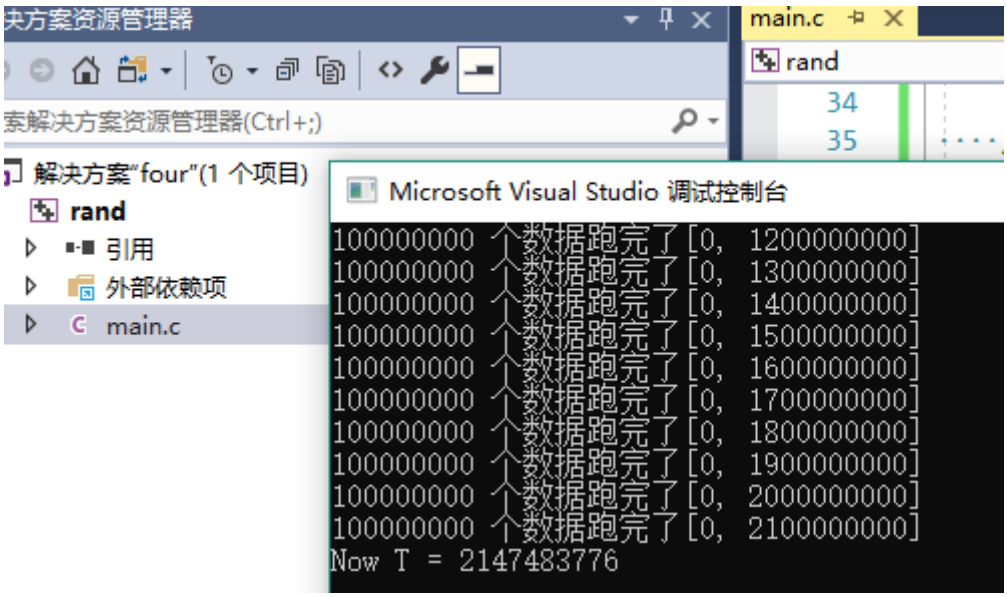
```
/*
 describe:
        1 亿的数据量，测试随机生成函数
        front system rand, back r_rand rand

 test code

 // 1 亿的数据测试
 #define TEST_INT       (100000000)

 static int test_rand(int (* trand)(void)) {
         int rd = 0;
         for (int i = 0; i < TEST_INT; ++i)
                rd = trand();
         return rd;
 }

 winds test :
        cl version 14 Visual Studio 2015 旗舰版(Window 10 专业版)

        Debug
        The current code block running time:1.743000 seconds
        The current code block running time:4.408000 seconds

        Release
        The current code block running time:1.649000 seconds
        The current code block running time:0.753000 seconds

 linux test :
        gcc version 6.3.0 20170406 (Ubuntu 6.3.0-12ubuntu2)
        -g -O2
        The current code block running time:0.775054 seconds
        The current code block running time:0.671887 seconds
 */
```

有了这些信息, 前戏做的够足了, 这里不妨带大家去武当山抓个宝宝.

```c
#include <stdio.h>
#include <stdlib.h>

#define R_INT         (128)
#define F_INT         (100000000)

// getr - 得到 rand() 返回值，并写入到文件中
static int getr(long long * pnt) {
    static int cnt;

    int r = rand();
    long long t = *pnt + 1;

    // 每次到万再提醒一下
    if(t % F_INT == 0)
        fprintf(stdout, "%d 个数据跑完了[%d, %lld]\n", F_INT, cnt, t);

    if(t < 0) { // 数据超标了
        ++cnt;

        fprintf(stderr, "Now %d T > %lld\n", cnt, t - 1);
        *pnt = 0; // 重新开始一轮
    }
    *pnt = t;
    return r;
}

// main - 验证 rand 函数的周期
int main(int argc, char* argv[]) {
    int base[R_INT];
    int r, i = -1;
    long long cnt = 0;


    // 先产生随机函数
    while(++i < R_INT)
        base[i] = getr(&cnt);

    // 这里开始随机了
    for(;;) {
        r = getr(&cnt);
        if (r != base[0])
            continue;

        // 继续匹配查找
        for(i = 1; i < R_INT; ++i) {
            r = getr(&cnt);
            if(r != base[i])
                break;
        }
```

```
            // 找见了数据
            if(i == R_INT) {
                printf("Now T = %lld\n", cnt);
                break;
            }
        }

        return EXIT_SUCCESS;
    }
```

可以将 R_INT 修改为 (1024) 最终得到结果也是一样. 因为抓到了 window 平台上面 rand() 伪随机函数的周期 G 点. 希望大家玩的开心.



那如何产生**真随机数呢**? 相关研究很多, 核心原理是**升维**, 借助对我们不可控系统产生不可控值. 例如噪音, 天气, 量子不可预测等等, 当然实际算法远比说的复杂, 抛开神明, 对于普通人而言确实足够随机了.

4.3 文件操作

文件相关操作包括删除创建获取文件属性等. 更加具体点的需求有, 想获取程序的运行目录, 需要多级删除目录, 需要多级创建目录... 这里先解决以上提出需求. 先展示部分设计, 再逐个击破.

4.3.1 文件操作辅助库 stdext

stdext.h

```c
#pragma once

#include <fcntl.h>
#include <stdint.h>
#include <sys/stat.h>
#include <sys/types.h>

#include "struct.h"

// 此库对于目录相关操作，并没有较好的屏蔽平台相关差异性. 依赖使用者求同存异.
// 例如 怎么看待目录: logs/heoos/gghh\\gggs/g/
// window 文件分隔符为 \ , 并且也兼容 /. 所以他看见的是 logs heoos gghh gggs g
// linux 文件分隔符为 /, 所以他看见的目录是 logs heoos gghh\gggs g
// 这些差别会影响 remove 和 mkdir 行为, 依赖使用者去怎么用对
//

#if defined(__linux__) && defined(__GNUC__)

#include <unistd.h>
#include <termios.h>

//
// mkdir - 单层目录创建函数宏, 类比 mkdir path
// path     : 目录路径
// return   : 0 表示成功, -1 表示失败, errno 存原因
//
#undef  mkdir
#define mkdir(path)                                     \
mkdir(path, S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH)

// getch - 立即得到用户输入的一个字符
inline int getch(void) {
    struct termios now, old;
    // 得到当前终端标准输入的设置
    if (tcgetattr(STDIN_FILENO, &old))
        return EOF;
    now = old;

    // 设置终端为 Raw 原始模式, 让输入数据全以字节单位被处理
    cfmakeraw(&now);
    // 设置上更改之后的设置
    if (tcsetattr(STDIN_FILENO, TCSANOW, &now))
        return EOF;

    int c = getchar();

    // 设置还原成老的模式
    if (tcsetattr(STDIN_FILENO, TCSANOW, &old))
        return EOF;
    return c;
```

```
}

// cls - 屏幕清除，依赖系统脚本
inline void clrscr(void) { printf("\ec"); }

#endif

//
// fmtime - 得到文件最后修改时间
// path     : 文件路径
// return   : 返回时间戳，-1 表示失败
//
inline time_t fmtime(const char * path) {
    struct stat st;
    // 数据最后的修改时间
    return stat(path, &st) ? -1 : st.st_mtime;
}

//
// fsize - 得到文件内容内存大小
// path     : 文件路径
// return   : 返回文件内存
//
inline int64_t fsize(const char * path) {
    struct stat st;
    // 数据最后的修改时间
    return stat(path, &st) ? -1 : st.st_size;
}

//
// removes - 删除非空目录 or 文件
// path     : 文件路径
// return   : not 0 is error, equal 0 is success
//
extern int removes(const char * path);

//
// mkdirs - 创建多级目录
// path     : 目录路径
// return   : < 0 is error, 0 is success
//
extern int mkdirs(const char * path);

//
// fmkdir - 通过文件路径创建目录
// path     : 文件路径
// return   : < 0 is error, 0 is success
//
extern int fmkdir(const char * path);

//
```

```
// getawd - 得到程序运行目录, \\ or / 结尾
// buf      : 存储地址
// size     : 存储大小
// return   : 返回长度, -1 or >= size is unusual
//
extern int getawd(char * buf, size_t size);
```

removes, mkdirs, mkfdir, getawd 有了这些接口, 以后写操作目录代码方便了很多. 其中 **removes** 借力的通过系统 shell 的能力来实现的.

```
#include "stdext.h"

//
// removes - 删除非空目录 or 文件
// path     : 文件路径
// return   : not 0 is error, equal 0 is success
//
inline int removes(const char * path) {
    // On success (all requested permissions granted, or mode is F_OK
    // and the file exists), zero is returned.  On  error  (at  least
    // one bit in mode asked for a permission that is denied, or mode
    // is F_OK and the file does  not  exist,  or  some  other  error
    // occurred), -1 is returned, and errno is set appropriately.
    if (access(path, F_OK)) {
        return 0;
    }

    char s[BUFSIZ];

# ifndef RMRF_STR
#   if defined(__linux__) && defined(__GNUC__)
#     define RMRF_STR    "rm -rf \"%s\""
#   endif
# endif

    // 发生异常 或者 path 超过缓冲区长度, 返回异常
    int sz = snprintf(s, sizeof s, RMRF_STR, path);
    if (sz < 0 || sz == sizeof s)
        return -1;

    return system(s);
}
```

access 用于检查 path 是否存在, 存在返回 0. 不存在返回 -1, 并且执行 system RMRF_STR 相关操作. 而 mkdirs 和 fmkdir 核心同样 access 和 mkdir 来回瞎搞.

```
//
// mkdirs - 创建多级目录
// path     : 目录路径
// return   : < 0 is error, 0 is success
//
int
mkdirs(const char * path) {
    char c, * p, * s;

    // 参数错误直接返回
    if (!path || !*path) return -1;

    // 文件存在 or 文件一次创建成功 直接返回
    if (!access(path, F_OK) || !mkdir(path))
        return 0;

    // 跳过第一个 ['/' | '\\'] 检查是否是多级目录
    p = (char *)path;
    while ((c = *++p) != '\0')
        if (c == '/' || c == '\\')
            break;
    if (c == '\0') return -1;

    // 开始循环构建多级目录
    s = p = strdup(path);
    if (p == NULL) {
        RETURN(-1, "strdup path = %p panic", path);
    }

    while ((c = *++p) != '\0') {
        if (c == '/' || c == '\\') {
            *p = '\0';

            if (access(s, F_OK)) {
                // 文件不存在，开始创建，创建失败直接返回错误
                if (mkdir(s)) {
                    free(s);
                    return -1;
                }
            }

            *p = c;
        }
    }

    // 最后善尾
    c = p[-1]; free(s);
    if (c == '/' || c == '\\')
        return 0;
    // 剩下最后文件路径，开始构建
```

```
    return mkdir(path);
}

//
// fmkdir - 通过文件路径创建目录
// path     : 文件路径
// return   : < 0 is error, 0 is success
//
int
fmkdir(const char * path) {
    const char * r;
    char c, * p, * s;
    if (!path) return -1;

    for (r = path + strlen(path); r >= path; --r)
        if ((c = *r) == '/' || c == '\\')
            break;
    if (r < path) return -1;

    // 复制地址地址并构建
    s = p = strdup(path);
    if (p == NULL) {
        RETURN(-1, "strdup path = %p panic", path);
    }

    p[r - path] = '\0';

    while ((c = *++p) != '\0') {
        if (c == '/' || c == '\\') {
            *p = '\0';

            if (access(s, F_OK)) {
                // 文件不存在，开始创建，创建失败直接返回错误
                if (mkdir(s)) {
                    free(s);
                    return -1;
                }
            }

            *p = c;
        }
    }

    // 一定不是 / or \\ 结尾直接，构建返回
    if (access(s, F_OK)) {
        if (mkdir(s)) {
            free(s);
            return -1;
        }
    }
    free(s);
```

```
    return 0;
}
```

最后 getawd 获取程序运行目录

```
//
// getawd - 得到程序运行目录, \\ or / 结尾
// buf      : 存储地址
// size     : 存储大小
// return   : 返回长度, -1 or >= size is unusual
//
int
getawd(char * buf, size_t size) {
    char * tail;

# ifndef getawe
#   if defined(__linux__) && defined(__GNUC__)
#     define getawe(b, s)    (int)readlink("/proc/self/exe", b, s);
#   endif
# endif

    int r = getawe(buf, size);
    if (r <= 0)    return -1;
    if ((size_t)r >= size) return  r;

    for (tail = buf + r - 1; tail > buf; --tail)
        if ((r = *tail) == '/' || r == '\\')
            break;
    // believe getawe return
    *++tail = '\0';
    return (int)(tail - buf);
}
```

主要使用场景, 通过 getawd 得到程序运行目录, 随后拼接出各种文件的绝对路径. 再去嗨.

```
#define LOG_PATH_STR        "logs/structc.log"

char path[BUFSIZ];
// 一切皆有可能 🙂
size_t n = getawd(path, LEN(path));
assert(0 < n && n < sizeof r);

// 日志模块初始化
memcpy(path + n, LOGS_PATH_STR, LEN(LOGS_PATH_STR));
fmkdir(path);
EXTERN_RUN(log_init, path);
```

stdext 拓展库主要围绕文件, 创建和删除还有文件属性等. 这些功能用系统本地 api 也许更好, 我们这里不少是借助 **system shell** 能力, 也是一种能用思路欢迎借鉴, 多看注释.

4.3.2 配置文件刷新小练习

很多时候有这样一个需求, 某些配置需要支持可刷新. 完成这个功能方式大致有两种, **1' 主动监控 2' 系统推送**. 这类配置文件动态刷新刷新在业务场景也非常常见. 存在两个主要使用场景, 客户端和服务器. 客户端需求很直白, 我本地配置变更, 程序能及时和非及时的重刷到系统中. 服务器相比客户端做法要多些环节, 服务器本地会有一份配置兜底, 配置中心中配置发生改变会推送给触发给服务器触发内部更新操作. 我们这里主要聊场景偏向于客户端, 本地配置发生改变, 我们如何来更新内存中配置?

我们这类简单点采用 **1' 主动监控** 附加是基于 **stdext.h 中的 mtime** 文件最后一次修改时间来处理这个需求. 对于 **2' 系统推送** 不同平台 api 不一样, 有兴趣可以多查查资料, 例如 **man inotify**. 好的我们先大致设计接口 **timer.h**

```
#pragma once

#include "struct.h"
#include "strext.h"
#include "spinlock.h"

//
// file_f - 文件更新行为
//
typedef void (* file_f)(FILE * c, void * arg);

//
// file_set - 文件注册更新行为
// path     : 文件路径
// func     : NULL 标识清除, 正常 update -> func(path -> FILE, arg)
// arg      : func 额外参数
// return   : void
//
extern void file_set(const char * path, file_f func, void * arg);

//
// file_update - 配置文件刷新操作
// return   : void
//
extern void file_update(void);
```

file_set 注册需要监控的文件, file_f 是监控到变化后触发的行为. file_update 是全局的更新行为, 用于监控是否有文件发生了变化. 他的本质是依赖 mtime 获取最后一次文件变化的时间. 用于确定此文件当前是否发生了变化. 有了这些我们开始三种思路实现.

1. 普通正常版本

```
#include "file.h"

struct file {
    time_t last;            // 文件最后修改时间点
    char * path;            // 文件全路径
    unsigned hash;          // 文件路径 hash 值

    file_f func;            // 执行行为
    void * arg;             // 行为参数

    struct file * next;     // 文件下一个结点
};

static struct files {
    struct file * list;     // 当前文件对象集
} f_s;
```

我们通过上面数据结构定义, 很清晰知道 **hash 和 path** 查找映射关系, **struct file * next;** 是个链表为 **file_update 循环遍历服务**. 因为上面是无锁的, 所以需要业务使用上避免线程并发问题, 需要程序启动一开始注册好所以需要主动监控的文件.

2. 多线程 list 加锁粗暴版本

```
#include "spinlock.h"

static struct files {
    atomic_flag lock;
    struct file * list;
} f_s;
```

我们希望引入 **atomic_flag lock;** 来处理 **struct file * list;** 并发的 add 和 remove 还有 get 问题. lock 确实这个问题, 但同样引入另外一个问题. 因为 lock 为了 file_set 和 file_update 服务, **file_update 操作颗粒时间一般会较长, 会阻塞 file_set 操作**. 这种思路不可能出现在实战中.

3. 多线程 乒乓 dict 复杂版本

```
#include "dict.h"
#include "strext.h"
#include "spinlock.h"

struct file {
    time_t last;            // 文件最后修改时间点
    file_f func;            // 执行行为
    void * arg;             // 行为参数
};

struct files {
    atomic_flag data_lock;
    // const char * path key -> value struct file
    // 用于 update 数据
    volatile dict_t data;

    atomic_flag backup_lock;
    // const char * path key -> value struct file
    // 在 update 兜底备份数据
    volatile dict_t backup;
};

static struct files F = {
    .data_lock = ATOMIC_FLAG_INIT,
    .backup_lock = ATOMIC_FLAG_INIT,
};

extern void file_init() {
    F.data = dict_create(file_delete);
    F.backup = dict_create(file_delete);
}
```

其中向 data 中添加数据时候, step 1 : 尝试竞争 data lock, step 2 : data lock 没有竞争到, 直接竞争 backup lock.

```
//
// file_set - 文件注册更新行为
// path     : 文件路径
// func     : NULL 标识清除，正常 update -> func(path -> FILE, arg)
// arg      : func 额外参数
// return   : void
//
void
file_set(const char * path, file_f func, void * arg) {
    struct file * fu = NULL;
    assert(path && *path);

    // step 1 : 尝试竞争 data lock
    if (atomic_flag_trylock(&F.data_lock)) {
        if (NULL != func) {
            fu = file_create(path, func, arg);
        }
        dict_set(F.data, path, fu);
        return atomic_flag_unlock(&F.data_lock);
    }

    // step 2 : data lock 没有竞争到，直接竞争 backup lock
    atomic_flag_lock(&F.backup_lock);
    fu = file_create(path, func, arg);
    dict_set(F.backup, path, fu);
    atomic_flag_unlock(&F.backup_lock);
}
```

3. 多线程 乒乓 list 返璞归真版本

```
struct file {
    file_f func;        // 执行行为, NULL 标识删除
    void * arg;         // 行为参数
    char * path;        // 文件路径
    unsigned hash;      // path hash
    time_t lasttime;    // 文件最后修改时间点

    struct file * next;
};

struct files {
    atomic_flag data_lock;
    // 用于 update 数据 empty head
    struct file data;

    atomic_flag backup_lock;
    // 在 update 兜底备份数据 empty head
    struct file backup;
};

static struct files F = {
    .data_lock = ATOMIC_FLAG_INIT,
    .backup_lock = ATOMIC_FLAG_INIT,
};
```

这种配置文件操作, 核心在于更新, 而不是频繁 update, add, delete 等. 所以 list 够用了. 当然如果写算法题, 那参照 LRU 套路 list + map.

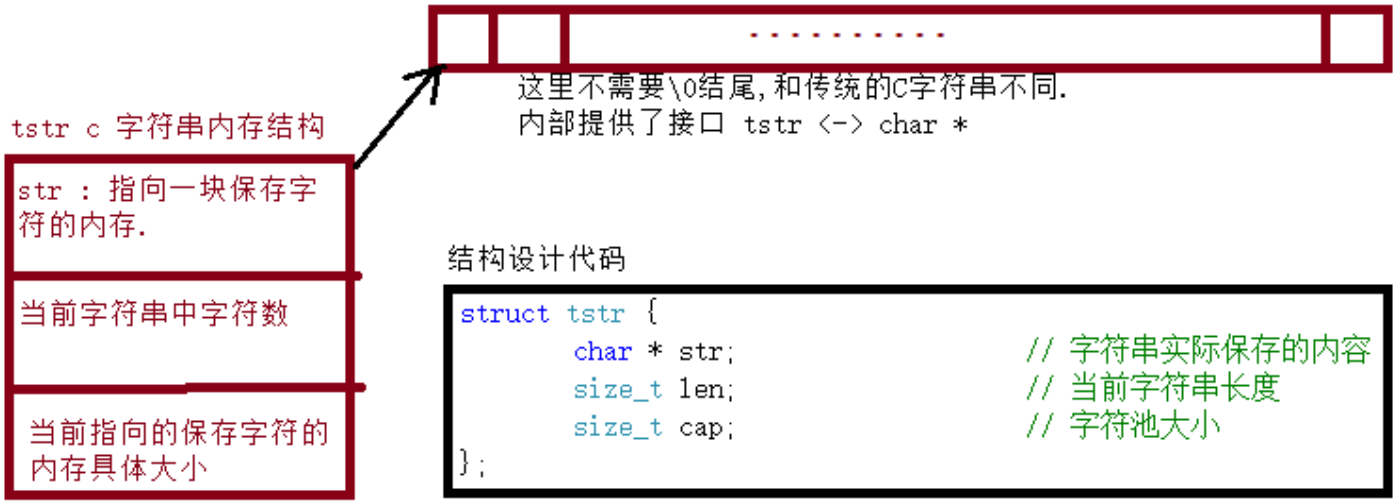
去感受其中数据结构设计的思路. 很多时候数据结构设计敲定了, 整体代码设计也就确定了. 其中 **#include "dict.h"** kv 库我们下一章节简单讲解. 本人用 C 写代码很顺手, 但有时候觉得 C 在现在阶段, 不是专业吃这个饭的, 可以尝试用其他更加高级语言来轻松快捷表达自己的想法和完成工程落地. 对于开发生涯作者花了很多年找到自己定位, 我的底层核心是一名软件开发工程师. 然后语言和技术以及商业工程问题陆续通顺起来了. 希望对你们有帮助, 思维的构建在使用好工具的下一个阶段将会很舒服.

4.4 C 造 json 轮子

在我刚做开发的时候, 那时候维护的系统, 所有配置走的是 xml 和 csv. 刚好 json 在国内刚兴起. 就一时兴起为其写了个解释器. 过了 1 年接触到 cJSON 库, 直接把自己当初写的那个删了. 用起了 cJSON, 后面觉得 cJSON 真的丑的不行不行的, 就琢磨写了个简单的 C json. 这小节, 就带大家写写这个 C json 的解析引擎, 清洁高效小. 能够保证的就是和 cJSON 对比学习更佳.

4.4.1 C json 设计布局

首先分析 C json 的实现部分. 最关心的是 C json 的内存布局, 实现层面引入了之前封装 str 库内存布局 (tstr 已经 301 cstr). 设计结构图如下 :



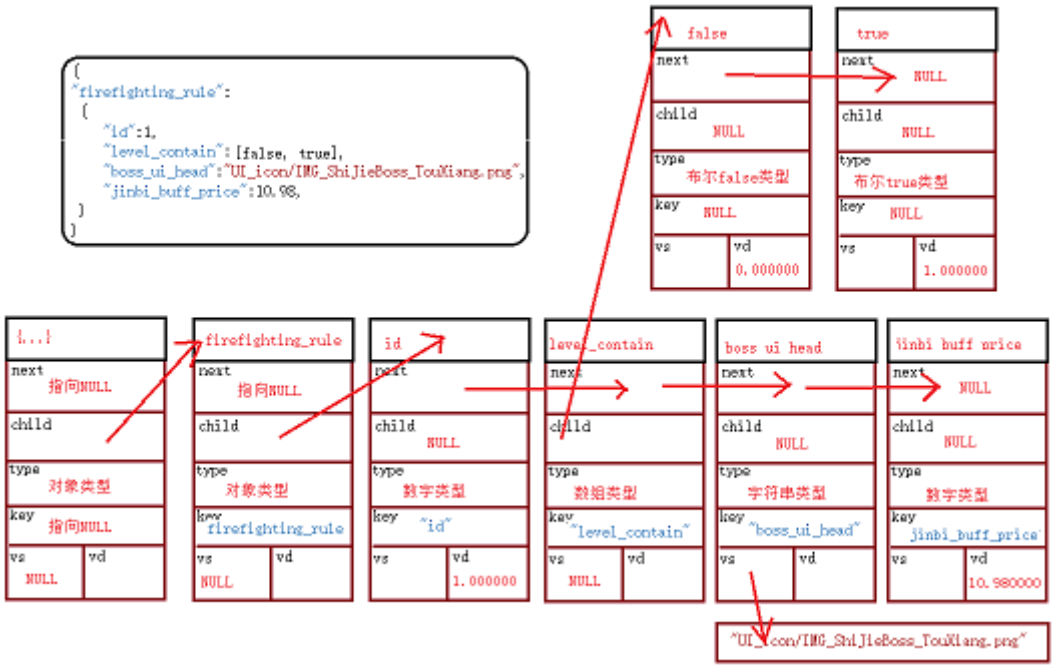
str 指向内存常量, cstr 指向内存不怎么变, 所以采用两块内存保存. tstr 存在目的是个中转站. 因为读取文件内容, 中间 json 内容清洗, 例如注释, 去空白, 压缩需要一块内存. 这就是引入目的. 再看看 C json 结构代码设计:

```c
#pragma once

#include <math.h>
#include <float.h>
#include <limits.h>
#include <stdbool.h>

#include "chars.h"
#include "strext.h"

//
// cj json fast parse, type is all design
// https://www.json.org/json-zh.html
//
#ifndef JSON_NULL

#define JSON_NULL           (0u << 0)
#define JSON_TRUE           (1u << 1)
#define JSON_FALSE          (1u << 2)
#define JSON_NUMBER         (1u << 3)
#define JSON_STRING         (1u << 4)
#define JSON_OBJECT         (1u << 5)
#define JSON_ARRAY          (1u << 6)
#define JSON_CONST          (1u << 7)

// JSON_OBJECT or JSON_ARRAY data struct
// |
// child
//     -> next -> ... -> next
//                         |
//                         child
//                             -> next -> ..
struct json {
    unsigned type;          // C JSON_NULL - JSON_ARRAY and JSON_CONST
    struct json * next;     // type & JSON_ARRAY | JSON_OBJECT -> 同级下个结点
    struct json * child;    // type & JSON_ARRAY | JSON_OBJECT -> 子结点

    char * key;             // json 结点的 key
    union {
        int len;            // type & JSON_ARRAY | JSON_OBJECT is json child len
        char * str;         // type & JSON_STRING is 字符串
        double num;         // type & JSON_NUMBER is number
    };
};

// 定义 json 对象类型
//
typedef struct json * json_t;

// json_int - 得到结点的 int 值.
```

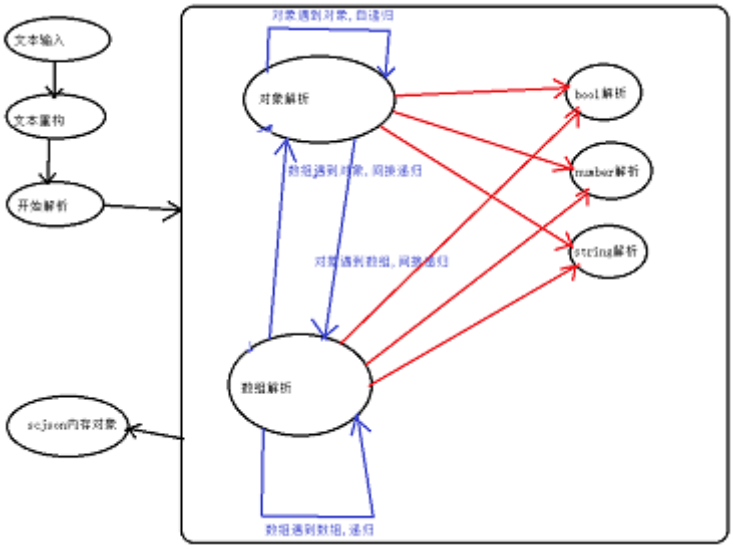
```
// double 10 他可以表示十进制的15或16位有效数字.
// int 范围 [-2^63, 2^63-1] 即 -2,147,483,648 到 2,147,483,647 约 9 到 10 位
#define json_int(item) ((int)(item)->num)

#endif//JSON_NULL

//
// json_delete - json 对象销毁
// cj        : json 对象
// return    : void
//
extern void json_delete(json_t cj);

//
// json_len - 获取 json 对象长度
// cj        : json 对象
// return    : 返回 json 对象长度
//
extern int json_len(json_t cj);

//
// json_array - 通过索引获取 json 数组中子结点
// aj        : json 数组
// i         : [0, json_len()) 索引
// return    : 返回对应的数组结点
//
extern json_t json_array(json_t aj, int i);

//
// json_object - 获取 json 对象中子对象
// obj       : json 对象
// k         : key
// return    : 返回对应的对象结点
//
extern json_t json_object(json_t obj, const char * k);

// json_mini - json 清洗函数 low level api
size_t json_mini(char * str);
// json_parse - json 解析函数 low level api
json_t json_parse(const char * str);

// json_create  解析字符串构造 json 对象
extern json_t json_create(const char * str);
// json_file 通过文件 path 解析文件内容构造 json 对象
extern json_t json_file(const char * path);

// json_detach_str - json 字符串分离, 需要自行 free
inline char * json_detach_str(json_t item) {
    item->type &= JSON_CONST;
    return item->str;
}
```

```
// json_detach_array - i ∈ [0, len()) 索引, 分离出 json 子对象
extern json_t json_detach_array(json_t aj, int i);
extern json_t json_detach_object(json_t obj, const char * k);

//
// json_string - 生成 json 对象 char * 字符串
// cj         : json_t 对象
// return     : 返回生成的 json 字符串, 需要自行 free
//
extern char * json_string(json_t cj);
```

使用 C99 的匿名结构体挺爽的, 整个 struct json 内存详细布局如下:



C json 中处理的类型类型无外乎 JSON_NULL, JSON_BOOL, JSON_NUMBER, JSON_STRING, JSON_OBJECT, JSON_ARRAY. 其中 JSON_CONST 是用于实现修饰用的. JSON_NUMBER 本质是 double, 通过 json_int 包装得到 int 值. 基于以上具体结构类型, 下面简单分析一下文本解析规则. 思路是递归下降分析. 到这里基本关于 C json 详细设计图介绍完毕了. 后面会看见这只麻雀代码极少 ヽ(✿゜▽゜)ノ



4.4.2 C json 详细设计

当初写这类东西, 就是对着协议文档开撸 ~ 这类代码是协议文档和作者思路的杂糅体, 推荐最好对着 json 官方协议加代码手敲一遍, 自行加注释, 琢磨后吸收. 来看看 C json 的删除函数

```c
#include "json.h"
#include "q.h"

void json_delete_recursion(json_t cj) {
    while (cj) {
        json_t next = cj->next;
        unsigned type = cj->type;

        free(cj->key);
        if ((type & JSON_STRING) && !(type & JSON_CONST))
            free(cj->str);

        // 子结点继续走深度递归删除
        if (cj->child)
            json_delete(cj->child);

        free(cj);
        cj = next;
    }
}

void json_delete(json_t cj) {
    if (cj == NULL) {
        return;
    }

    struct q q;
    if (q_init(&q) == false) {
        // 把崩溃从堆推向了栈
        return json_delete_recursion(cj);
    }

    do {
        json_t next = cj->next;
        unsigned type = cj->type;

        free(cj->key);
        if ((type & JSON_STRING) && !(type & JSON_CONST))
            free(cj->str);

        if (cj->child) {
            if (q_push(&q, cj->child) == false) {
                // 堆上以及没有内存，这时候尝试崩溃，当然这里纯为了学习，而无中生有
                EXIT("q_push panic memory");
            }
        }

        free(cj);
        cj = next ? next : q_pop(&q);
    } while (cj);
```

```
    q_release(&q);
}
```

上面操作无外乎就是递归找到最下面的儿子结点, 期间删除自己挂载的结点. 然后依次按照 next 链表顺序循环执行. 随后通过代码逐个分析思维过程, 例如我们得到一个 json 串, 这个串中可能存在多余的空格, 多余的注释等. 就需要做洗词的操作, 只留下最有用的 json 字符串.

```c
// json_mini - 清洗 str 中冗余的串并返回最终串的长度. 纪念 mini 比男的还平 :)
// EF BB BF     = UTF-8                (可选标记, 因为 Unicode 标准未有建议)
// FE FF        = UTF-16, big-endian   (大尾字节序标记)
// FF FE        = UTF-16, little-endian (小尾字节序标记, windows Unicode 编码默认标记)
// 00 00 FE FF  = UTF-32, big-endian   (大尾字节序标记)
// FF FE 00 00  = UTF-32, little-endian (小尾字节序标记)
//
size_t json_mini(char * str) {
    char c, * in = str;
    unsigned char * to = (unsigned char *)str;

    // 跳过 UTF-8 With BOM 前三个字节
    if (to[0] == 0xEF && to[1] == 0xBB && to[2] == 0xBF)
        to += 3;

    while ((c = *to)) {
        // step 0 : 处理字面串
        if (c == '`') {
            *in++ = c;
            while ((c = *++to) && c != '`')
                *in++ = c;
            if (c) {
                *in++ = c;
                ++to;
            }
            continue;
        }

        // step 1 : 处理字符串
        if (c == '"') {
            *in++ = c;
            while ((c = *++to) && (c != '"' || to[-1] == '\\'))
                *in++ = c;
            if (c) {
                *in++ = c;
                ++to;
            }
            continue;
        }
        // step 2 : 处理不可见特殊字符
        if (c < '!') {
            ++to;
            continue;
        }
        if (c == '/') {
            // step 3 : 处理 // 解析到行末尾
            if (to[1] == '/') {
                while ((c = *++to) && c != '\n')
                    ;
                continue;
```

```
            }
            // step 4 : 处理 /*
            if (to[1] == '*') {
                while ((c = *++to) && (c != '*' || to[1] != '/'))
                    ;
                if (c)
                    to += 2;
                continue;
            }
        }
        // step 5 : 合法数据直接保存
        *in++ = *to++;
    }

    *in = '\0';
    return in - str;
}
```

以上操作主要目的是让解析器能够处理 json 串中 // 和 /**/, 并删除些不可见字符. 开始上真正的解析器入口函数:

```
//
// json_parse - json 解析函数
// str      : json 字符串串
// return   : json 对象, NULL 表示解析失败
//
json_t json_parse(const char * str) {
    json_t cj = json_new();
    if (parse_value(cj, str) == NULL) {
        json_delete(cj);
        return NULL;
    }
    return cj;
}

json_t
json_create(const char * str) {
    if (str == NULL || *str == 0)
        return NULL;

    char * ss = strdup(str);
    // 清洗 + 解析
    json_mini(ss);
    json_t c = json_parse(ss);
    free(ss);
    return c;
}

json_t
json_file(const char * path) {
    char * str = str_freads(path);
    // 读取文件中内容, 并事先检查参数
    if (str == NULL)
        return NULL;

    // 尝试解析结果
    json_t c = json_create(str);
    free(str);
    return c;
}
```

真正的解从 json_create 看起, 声明了栈上字符串 cs 填充 str, 随后进行 json_mini 洗词, 然后通过 json_parse 解析出最终结果并返回. 随后可以看哈 json_parse 实现非常好理解, 核心调用的是 parse_value. 而 parse_value 就是我们的重头戏, 本质就是走分支. 不同分支走不同的解析操作.

```
//
// parse_value - 递归下降解析
// item     : json 结点
// str      : 语句源串
// return   : 解析后剩下的串
//
static const char * parse_value(json_t item, const char * str);

static const char * parse_value(json_t item, const char * str) {
    if (!str) return NULL;
    switch (*str) {
    // n or N = null, f or F = false, t or T = true
    case 'n': case 'N':
        if (strncasecmp(str + 1, "ull", sizeof "ull" - 1)) return NULL;
        item->type = JSON_NULL;
        return str + sizeof "ull"; // exists invalid is you!
    case 't': case 'T':
        if (strncasecmp(str + 1, "rue", sizeof "rue" - 1)) return NULL;
        item->type = JSON_TRUE; item->num = true;
        return str + sizeof "rue";
    case 'f': case 'F':
        if (strncasecmp(str + 1, "alse", sizeof "alse"-1)) return NULL;
        item->type = JSON_FALSE;
        return str + sizeof "alse";
    case '+': case '-': case '.':
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        return parse_number(item, str);
    case '`': return parse_literal(item, str + 1);
    case '"': return parse_string (item, str + 1);
    case '{': return parse_object (item, str + 1);
    case '[': return parse_array  (item, str + 1);
    }
    return NULL;
}
```

由 parse_value 引出了 parse_number, parse_literal, parse_string, parse_object, parse_array. 是不是后面五个 parse 写好了 parse_value 就写好了. 那随后开始逐个击破, parse_number 走起.

```c
// parse_number - number 解析
static const char * parse_number(json_t item, const char * str) {
    char c;
    double n = 0;
    int e, eign, sign = 1;

    // 正负号处理判断
    if ((c = *str) == '-' || c == '+') {
        sign = c == '-' ? -1 : 1;
        c = *++str;
    }

    // 整数处理部分
    while (c >= '0' && c <= '9') {
        n = n * 10 + c - '0';
        c = *++str;
    }
    // 处理小数部分
    if (c == '.') {
        int d = 0;
        double s = 1;
        while ((c = *++str) && c >= '0' && c <= '9') {
            d = d * 10 + c - '0';
            s *= 0.1;
        }
        // 得到整数和小数部分
        n += s * d;
    }

    // 添加正负号
    n *= sign;

    // 不是科学计数内容直接返回
    item->type = JSON_NUMBER;
    if (c != 'e' && c != 'E') {
        item->num = n;
        return str;
    }

    // 处理科学计数法
    if ((c = *++str) == '-' || c == '+')
        ++str;
    eign = c == '-' ? -1 : 1;

    e = 0;
    while ((c = *str) >= '0' && c <= '9') {
        e = e * 10 + c - '0';
        ++str;
    }
```

```
    // number = +/- number.fraction * 10^+/- exponent
    item->num = n * pow(10, eign * e);
    return str;
}
```

parse_number 特别像下面两兄弟. 大体功能相似, 用于将字符串解析成浮点数.

```
extern double __cdecl strtod(char const * _String, char ** _EndPtr);

inline double __cdecl atof(char const * _String) {
    return strtod(_String, NULL);
}
```

parse_literal 用于解析 `` 包裹的字符常量. 输入额外添加的私货.

```
// parse_literal - 字面串解析
static const char * parse_literal(json_t item, const char * str) {
    char c;
    size_t size;
    const char * etr = '\n' == *str ? ++str : str;

    // 获取到 '``' 字符结尾处
    while ((c = *etr) != '``' && c)
        ++etr;
    if ('``' != c) return NULL;

    // 尝试吃掉 `` 开头第一个和结尾最后一个 \n, 方便整齐划一
    size = '\n' == etr[-1] ? etr - str - 1 : etr - str;

    // 开始构造和填充 json string 结点
    item->type = JSON_STRING;
    item->str = malloc(size + 1);
    memcpy(item->str, str, size);
    item->str[size] = '\0';

    return etr + 1;
}
```

是不是也很骨骼精奇. 快要进入小高潮了 parse_string 解析难点在于 UTF-8 \uxxxx 字符的处理. 我们 copy 原先 cJSON 的代码. 作为程序员, 有些地方还是得低头 ~

```
// parse_hex4 - parse 4 digit hexadecimal number
static unsigned parse_hex4(const char str[]) {
    unsigned h = 0;
    for (unsigned i = 0; ; ++str) {
        unsigned char c = *str;
        if (c >= '0' && c <= '9')
            h += c - '0';
        else if (c >= 'a' && c <= 'f')
            h += c - 'a' + 10;
        else if (c >= 'A' && c <= 'F')
            h += c - 'A' + 10;
        else return 0; // invalid

        // shift left to make place for the next nibble
        if (4 == ++i) break;
        h <<= 4;
    }

    return h;
}

// parse_string - string 解析
static const char * parse_string(json_t item, const char * str) {
    unsigned len = 1;
    char c, * cursor, * out;
    const char * ptr, * etr = str;

    while ((c = *etr) != '"' && c) {
        ++etr;
        // 转义字符特殊处理
        if (c == '\\') {
            if (*etr == '\0')
                return NULL;
            ++etr;
        }
        ++len;
    }
    if (c != '"') return NULL;

    // 开始复制拷贝内容
    cursor = out = malloc(len);
    assert(out != NULL);
    for (ptr = str; ptr < etr; ++ptr) {
        // 普通字符直接添加处理
        if ((c = *ptr) != '\\') {
            *cursor++ = c;
            continue;
        }
        // 转义字符处理
        switch ((c = *++ptr)) {
```

```
case 'b': *cursor++ = '\b'; break;
case 'f': *cursor++ = '\f'; break;
case 'n': *cursor++ = '\n'; break;
case 'r': *cursor++ = '\r'; break;
case 't': *cursor++ = '\t'; break;
// transcode UTF16 to UTF8. See RFC2781 and RFC3629
case 'u': {
    // first bytes of UTF8 encoding for a given length in bytes
    static const unsigned char marks[] = {
        0x00, 0x00, 0xC0, 0xE0, 0xF0, 0xF8, 0xFC
    };
    unsigned oc, uc = parse_hex4(ptr + 1);
    // check for invalid
    if ((ptr += 4) >= etr) goto faild_free;
    if ((uc >= 0xDC00 && uc <= 0xDFFF) || uc == 0)
        goto faild_free;

    // UTF16 surrogate pairs
    if (uc >= 0xD800 && uc <= 0xDBFF) {
        if ((ptr + 6) >= etr) goto faild_free;
        // missing second-half of surrogate
        if ((ptr[1] != '\\') || (ptr[2] != 'u' && ptr[2] != 'U'))
            goto faild_free;

        oc = parse_hex4(ptr + 3);
        ptr += 6; // parse \uXXXX
        // invalid second-half of surrogate
        if (oc < 0xDC00 || oc > 0xDFFF) goto faild_free;
        // calculate unicode codepoint from the surrogate pair
        uc = 0x10000 + (((uc & 0x3FF) << 10) | (oc & 0x3FF));
    }

    // encode as UTF8
    // takes at maximum 4 bytes to encode:

    // normal ascii, encoding 0xxxxxxx
    if (uc < 0x80) len = 1;
    // two bytes, encoding 110xxxxx 10xxxxxx
    else if (uc < 0x800) len = 2;
    // three bytes, encoding 1110xxxx 10xxxxxx 10xxxxxx
    else if (uc < 0x10000) len = 3;
    // 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
    else len = 4;
    cursor += len;

    switch (len) {
    // 10xxxxxx
    case 4: *--cursor = ((uc | 0x80) & 0xBF); uc >>= 6; __attribute__((fallthrough));
    // 10xxxxxx
    case 3: *--cursor = ((uc | 0x80) & 0xBF); uc >>= 6; __attribute__((fallthrough));
    // 10xxxxxx
```

```
            case 2: *--cursor = ((uc | 0x80) & 0xBF); uc >>= 6; __attribute__((fallthrough));
            // depending on the length in bytes this determines the
            // encoding ofthe first UTF8 byte
            case 1: *--cursor = ((uc | marks[len]));
            }
            cursor += len;
            break;
        }
        default : *cursor++ = c;
        }
    }
    *cursor = '\0';
    item->str = out;
    item->type = JSON_STRING;
    return ptr + 1;

faild_free:
    free(out);
    return NULL;
}
```

编码转换有兴趣需要详细看 Unicode 字符集中 UTF-8, UTF-16, UTF-32 编码关系. 扯一点, 很久以前对于编码解决方案. 采用的是 libiconv 方案, 将其移植到 window 上. 后面学到一招, 因为国内开发最多的需求就是 gbk 和 utf-8 国际标准的来回切. 那就直接把这个编码转换的算法拔下来, 岂不最好 ~ 所以后面抄录了一份 **utf8.h**. 有兴趣同学可以去作者主页找下来看看, 这里只带大家看看接口设计.

```
#pragma once

#include "struct.h"

//
// utf8 和 gbk 基础能力处理库
//
// g = gbk 是 ascii 扩展码, u8 = utf8
// 2 * LEN(g) >= LEN(u8) >= LEN(g)
//
// 编码相关小知识科普: https://madmalls.com/blog/post/unicode-and-utf8/

//
// utf82gbk - utf8 to gbk save d mem
// gbk2utf8 - gbk to utf8 save d mem by size n
// d        : mem
// n        : size
// return   : void
//
extern void utf82gbk(char d[]);
extern void gbk2utf8(char d[], size_t n);

//
// isutf8s - 判断字符串是否是utf8编码
// s        : 输入的串
// return   : true 表示 utf8 编码
//
extern bool isutf8s(const char * s);

//
// isutf8 - check is utf8
// d        : mem
// n        : size
// return   : true 表示 utf8 编码
//
extern bool isutf8(const char d[], size_t n);
```

> 引述一丁点维基百科上 UTF-8 编码字节含义:
>
> 对于 UTF-8 编码中的任意字节 B, 如果 B 的第一位为 0，则 B 独立的表示一个字符(是 ASCII 码);
> 如果 B 的第一位为 1, 第二位为 0, 则 B 为一个多字节字符中的一个字节(非 ASCII 字符);
> 如果 B 的前两位为 1, 第三位为 0, 则 B 为两个字节表示的字符中的第一个字节;
> 如果 B 的前三位为 1, 第四位为 0, 则 B 为三个字节表示的字符中的第一个字节;
> 如果 B 的前四位为 1, 第五位为 0, 则 B 为四个字节表示的字符中的第一个字节;

> 因此, 对 UTF-8 编码中的任意字节, 根据第一位, 可判断是否为 ASCII 字符; 根据前二位,
> 可判断该字节是否为一个字符编码的第一个字节; 根据前四位(如果前两位均为 1),
> 可确定该字节为字符编码的第一个字节, 并且可判断对应的字符由几个字节表示;
> 根据前五位(如果前四位为 1), 可判断编码是否有错误或数据传输过程中是否有错误.

有了插播的内容, 写个判断是否是 utf-8 编码还是容易的. 希望对你理解 parse_string 有所帮助.

```
// isutf8_local - 判断是否是 utf8 串的临时状态
static bool isutf8_local(unsigned char c, unsigned char * bytes, bool * ascii) {
    // ascii 码最高位为 0, 0xxx xxxx
    if ((c & 0x80)) *ascii = false;

    // 计算字节数
    if (0 == *bytes) {
        if (c >= 0x80) {
            if (c >= 0xFC && c <= 0xFD) *bytes = 6;
            else if (c >= 0xF8) *bytes = 5;
            else if (c >= 0xF0) *bytes = 4;
            else if (c >= 0xE0) *bytes = 3;
            else if (c >= 0xC0) *bytes = 2;
            else return false; // 异常编码直接返回
            --*bytes;
        }
    } else {
        // 多字节的非首位字节, 应为 10xx xxxx
        if ((c & 0xC0) != 0x80) return false;
        // bytes 来回变化, 最终必须为 0
        --*bytes;
    }
    return true;
}

//
// isutf8s - 判断字符串是否是utf8编码
// s        : 输入的串
// return   : true 表示 utf8 编码
//
bool
isutf8s(const char * s) {
    bool ascii = true;
    // bytes 表示编码字节数, utf8 [1, 6] 字节编码
    unsigned char bytes = 0;

    for (unsigned char c; (c = *s); ++s)
        if (!isutf8_local(c, &bytes, &ascii))
            return false;

    return !ascii && bytes == 0;
}

//
// isutf8 - check is utf8
// d        : mem
// n        : size
// return   : true 表示 utf8 编码
//
bool
```

```
isutf8(const char d[], size_t n) {
    bool ascii = true;
    // bytes 表示编码字节数, utf8 [1, 6] 字节编码
    unsigned char bytes = 0;

    for (size_t i = 0; i < n; ++i)
        if (!isutf8_local(d[i], &bytes, &ascii))
            return false;

    return !ascii && bytes == 0;
}
```

我们写代码最好也要逐步同**国际接轨**, **编码能用 UTF-8 就用 UTF-8!**

4.4.3 parse array value

到结尾戏了. 递归下降分析的两位主角 parse_array 和 parse_object. 希望带给你不一样的体验.

```
// parse_array - array 解析
static const char * parse_array(json_t item, const char * str) {
    json_t child;
    item->type = JSON_ARRAY;
    // 空数组直接解析完毕退出
    if (']' == *str) return str + 1;

    // 开始解析数组中数据
    item->child = child = json_new();
    str = parse_value(child, str);
    if (!str) return NULL;
    item->len++;

    // array ',' cut
    while (',' == *str) {
        // 支持行尾多一个 ','
        if (']' == *++str)
            return str + 1;

        child->next = json_new();
        child = child->next;
        // 继续间接递归处理值
        str = parse_value(child, str);
        if (!str) return NULL;
            item->len++;
    }

    return ']' == *str ? str + 1 : NULL;
}
```

parse_array 处理的格式 '[... , ... , ...]' 串. 同样 parse_object 处理的格式如下 '{ "key":..., "key":..., ... }'

```
// parse_object - object 解析
static const char * parse_object(json_t item, const char * str) {
    json_t child;
    item->type = JSON_OBJECT;
    if ('}' == *str) return str + 1;
    // "key" check invalid
    if ('"' != *str && *str != '`') return NULL;

    // {"key":value,...} 先处理 key
    item->child = child = json_new();
    if ('"' == *str)
        str = parse_string (child, str + 1);
    else
        str = parse_literal(child, str + 1);

    if (!str || *str != ':') return NULL;
    child->key = child->str;
    child->str = NULL;

    // 再处理 value
    str = parse_value(child, str + 1);
    if (!str) return NULL;
    item->len++;

    // 开始间接递归解析
    while (*str == ',') {
        // 多行解析直接返回结果
        if ('}' == *++str) return str + 1;
        if ('"' != *str && *str != '`') return NULL;

        child->next = json_new();
        child = child->next;
        if ('"' == *str)
            str = parse_string (child, str + 1);
        else
            str = parse_literal(child, str + 1);

        if (!str || *str != ':') return NULL;
        child->key = child->str;
        child->str = NULL;

        str = parse_value(child, str + 1);
        if (!str) return NULL;
        item->len++;
    }

    return '}' == *str ? str + 1 : NULL;
}
```

关于 json 串的解析部分就完工了. 核心是学习递归下降分析的套路, 间接递归. 通过上面演示的思路, 花些心思也可以构建出 json 对象转 json 字符串的套路. 麻烦点有 JSON_STRING 转换, 我们简单提提, 有心人可以作为拓展修炼. 有了 json 的处理库, 有没有感觉基础的业务配置就很轻松了.

```
// json_string_string - string 编码
static char * json_string_string(char * str, struct chars * p) {
    unsigned char c;
    const char * ptr;
    char * cursor, * out;
    // 什么都没有 返回 "" empty string
    if (NULL == str || *str == 0) {
        out = chars_expand(p, 3);
        out[0] = out[1] = '"'; out[2] = '\0';
        p->len += 2;
        return out;
    }

    // 获取最终字符输出长度
    size_t len = 0;
    for (ptr = str; (c = *ptr); ++ptr) {
        ++len;
        switch (c) {
        case '\b': case '\t': case '\n':
        case '\f': case '\r':
        case '\\': case '"': ++len; break;
        default:
            if (c < 32) {
                // UTF-16 escape sequence uXXXX
                len += 5;
            }
        }
    }

    // 开始分配内存
    cursor = out = chars_expand(p, len+3);
    out[len+2] = 0;
    *cursor++ = '"';

    // 没有特殊字符直接返回
    if (len == (size_t)(ptr - str)) {
        memcpy(cursor, str, len);
        goto ret_out;
    }

    // 存在特殊字符挨个处理
    for (ptr = str; (c = *ptr); ++ptr) {
        if (c >= 32 && c != '"' && c != '\\') {
            *cursor++ = c;
            continue;
        }
        *cursor++ = '\\';
        switch(c) {
        case '\b': *cursor++ = 'b'; break;
        case '\t': *cursor++ = 't'; break;
```

```
        case '\n': *cursor++ = 'n'; break;
        case '\f': *cursor++ = 'f'; break;
        case '\r': *cursor++ = 'r'; break;
        case '"': case '\\': *cursor++ = c; break;
        // escape and print as unicode codepoint
        default: sprintf(cursor, "u%04x", c); cursor += 5;
        }
    }

ret_out:
    out[len+1] = '"';
    p->len += len+2;
    return out;
}
```

4.5 C json 小练习 config 配置库

有了上面 json 解析库, 我们不妨运用 C json 解析能力, 构建配置解析库. 这年头配置解析库有不少, 例如 ini, csv, xml, json, yaml, toml, 自定义 ... 比较推荐是 json 和 toml. json 推荐原因在于至今通用性最好, 配置, 协议传输, javascript 可直接使用等等优势. 我们先看待解析的 **配置文件 conf/conf.conf**.

```
/*
 * struct c config
 *
 * 框架支持的 json 配置
 *
 */

{
  `description` : "我不知道风是在哪一个方向吹。",

  "image"       : `
*---------------------------------** struct c **---------------------------------*
| Welcome to the struct c v2.0.0                                                  |
| More detailed information need to http://www.cnblogs.com/life2refuel/           |
| Thank you so much by wangzhi @ https://github.com/wangzhione/structc           |
|                                                                                 |
|      _________                                                                  |
|     / ======= \                                                                |
|    / __________\                                                               |
|   | ___________ |                                                              |
|   | | -       | |                                                              |
|   | |  Hello  | |                                                              |
|   | |_________| |________________                                              |
|   \=____________/                )                                             |
|   / ........... \              /   >   ->   go -->   go   ->>   go   ~>--     |
|  / ::::::::::::: \          =D-'                                                |
| (___________________)                                                           |
|                                                                                 |
| ---- __@                 __@          __@    -----  __@                 __@ |
| --- _\\<,_              _\\<,_       _\\<,_  ----  _\\<,_              _\\<,|
| -- (*)/ (*)            (*)/ (*)    (*)/ (*)  ---  (*)/ (*)            (*)/ (|
|                                                                                 |
*---------------------------------** struct c **---------------------------------*
`,

  //
  // add you config, 再在 :: conf.h / conf.c 中添加映射关系
  //

}
```

** config 配置库定义**

conf.h 接口构思是配置文件同运行程序中内存一一映射, 一条配置程序就有一个字段和其对应. 可以从 struct conf 中字段看出来.

```c
#pragma once

#include "utf8.h"
#include "json.h"

//
// config 映射配置
//
struct conf {
    char * description;
    char * image;
};

//
// conf_instance - 获取配置
// return   : 返回详细配置内容
//
extern struct conf * conf_instance(void);

//
// conf_init - 初始化读取配置内容
// path     : 配置初始化路径
// return   : true 表示解析成功
//
bool conf_init(const char * path);
```

实现层面考虑了文件格式可能是 gdk 和 utf8 两种情况. 具体见 locals 实现代码.

```c
#include "conf.h"

//
// conf_instance - 获取配置
// return   : 返回详细配置内容
//
inline struct conf * conf_instance(void) {
    //
    // 配置存储信息
    //
    static struct conf conf;

    return &conf;
}

// CONFIG_PARSE_JSON_STR - json field -> conf field
#define CONFIG_PARSE_JSON_STR(json, conf, field)            \
json_t $##field = json_object(json, #field);                \
if (!$##field || $##field->type != JSON_STRING) {           \
    RETURN(false, "json_object err "#field" %p", $##field); \
}                                                           \
free(conf->field);                                          \
conf->field = json_detach_str($##field);

// conf_parse - 解析内容，并返回解析结果
static bool conf_parse(json_t json, struct conf * conf) {
    CONFIG_PARSE_JSON_STR(json, conf, description);
    CONFIG_PARSE_JSON_STR(json, conf, image);

    // ... .. .

    return true;
}

//
// conf_init - 初始化读取配置内容
// path     : 配置初始化路径
// return   : true 表示解析成功
//
bool conf_init(const char * path) {
    json_t json = json_file(path);
    if (!json) {
        RETURN(false, "json_file err path is %s", path);
    }

    // 解析 json 内容，并返回详细配置内容
    bool ret = conf_parse(json, conf_instance());
    json_delete(json);
    return ret;
```

```
}
```

使用的时候先要在业务使用之前注册 **conf_init**, 随后就可以通过 **conf_instance()** 来获取配置中内容. 经过这些是不是觉得, 到练气也不过如此. 气随心动.

4.6 奥特曼, 通用头文件

在实战项目中, 都会有个出现频率特别高的一个头文件, 项目中基本每个业务头文件都继承自他. 同样此刻要出现的就是筑基期至强奥义, 一切从头开始 **base.h**.

```
#pragma once

#include "log.h"
#include "rand.h"
#include "alloc.h"
#include "thread.h"
#include "strext.h"
#include "sundries.h"
```

base.h 相关内容比较很简单, 就是汇总常用头文件. **思想就是让业务使用者不再如数家珍去记忆常用头文件**. 其中 check.h 可以放入一些参数校验的函数. 可以随着自身对业务修炼的理解, 自主添加. 目前这里只是加了个 email 校验操作.

```c
#include "sundries.h"

//
// is_email - 判断是否是邮箱
// mail     : email 串
// return   : true is email
#define EMAIL_INT (255)
bool
is_email(const char * mail) {
    //
    // A@B.xx size <= 255
    // [0-9][a-z][A-Z] .-_ 字符构成
    // @ . 分割处首位和末位只能是 [0-9][a-z][A-Z]
    // 不能出现 .. 连续
    //
    int c, i, b, d;
    if (!mail) return false;
    // check A
    c = *mail;
    if (!((c >= '0' && c <= '9')
       || (c >= 'A' && c <= 'Z')
       || (c >= 'a' && c <= 'z'))) return false;
    for (i = 1; (c = *++mail) && c != '@' && i < EMAIL_INT; ++i) {
        // 非法字符直接返回
        if (!((c >= '0' && c <= '9')
        || (c >= 'A' && c <= 'Z')
        || (c >= 'a' && c <= 'z'))) {
            if (c == '-' || c == '_')
                continue;
            return false;
        }
    }

    // check A end
    if (c != '@' || i >= EMAIL_INT
                 || mail[-1] == '-' || mail[-1] == '_')
        return false;

    // check b start
    for (b = d = false; (c = *++mail) && i < EMAIL_INT; ++i) {
        // 非法字符直接返回
        if (!((c >= '0' && c <= '9')
        || (c >= 'A' && c <= 'Z')
        || (c >= 'a' && c <= 'z'))) {
            // 首字符不能是 非数字和字母
            if (b) {
                if (c == '-' || c == '_')
                    continue;
                else if (c == '.') {
                    b = false;
```

```
                    d = true;
                    continue;
                }
            }
            return false;
        }
        b = true;
    }
    // 必须存在 ., 最后 '\0' 结尾, 255 以内
    return b && d && !c && i < EMAIL_INT
             && (mail[-1] < '0' || mail[-1] > '9');
}

//
// url_encode - url 编码, 需要自己 free
// s        : url串
// len      : url串长度
// nen      : 返回编码后串长度
// return   : 返回编码后串的首地址
//
char *
url_encode(const char * s, int len, int * nen) {
    if (s == NULL || *s == '\0' || len <= 0) {
        if (nen) *nen = 0;
        return NULL;
    }

    const unsigned char * from = (unsigned char *)s;
    const unsigned char * end = from + len;
    unsigned char * to = calloc(3 * len + 1, 1);
    unsigned char * start = to;

    while (from < end) {
        register unsigned char c = *from++;
        if (c == ' ') {
            *to++ = '+';
            continue;
        }

        // [a-z] [A-Z] [0-9] [&-./:=?_] 以外字符采用二进制替代
        if ((c < '0' && c != '&' && c != '-' && c != '.' && c != '/') ||
            (c < 'A' && c >  '9' && c != ':' && c != '=' && c != '?') ||
            (c > 'Z' && c  < 'a' && c != '_') ||
            (c > 'z')) {
            to[0] = '%';
            to[1] = "0123456789ABCDEF"[c >> 4];
            to[2] = "0123456789ABCDEF"[c & 15];
            to += 3;
            continue;
        }
```

```
        *to++ = c;
    }
    *to = '\0';

    // 返回结果
    if (nen) *nen = (int)(to - start);
    return (char *)start;
}

// htoc - 2 字节变成 16 进制数表示
inline char htoc(char * s) {
    int v, c= s[0];
    // 小写变大写是兼容性写法
    if (islower(c)) c = toupper(c);
    v =  (c >= '0' && c <= '9' ? c - '0' : c - 'A' + 10) * 16;

    c = s[1];
    if (islower(c)) c = toupper(c);
    v += (c >= '0' && c <= '9' ? c - '0' : c - 'A' + 10);

    return (char)v;
}

//
// url_decode - url 解码，解码后也是放在 s[] 中
// s        : 待解码的串
// len      : 解码串长度
// return   : 返回解码串的长度，< 0 表示失败
//
int
url_decode(char s[], int len) {
    if (s == NULL || *s == '\0' || len <= 0)
        return -1;

    char * dest = s,  * data = s;
    while (len--) {
        char c = *data++;
        // 反向解码
        if (c == '+')
            *dest = ' ';
        else if (c == '%' && len >= 2
                          && isxdigit(data[0])
                          && isxdigit(data[1])) {
            *dest = htoc(data);
            data += 2;
            len -= 2;
        }
        else {
            *dest = c;
        }
        ++dest;
```

```
    }
    *dest = '\0';

    return (int)(dest - s);
}
```

如果有问题可以在修真岁月中道友间互相探讨补充. **stdext.h** 中 **getch** 函数可以重点关注下. 很久以前一位化神期巨擘说过: 由于 linux 对于 getch 支持不友好, 导致了 linux 错失了很多游戏开发人员. 我是挺喜欢 getch 的, 让立即交互变得轻松. 所以就顺手补上了. 继承 base.h 会让业务轻装上阵. 美好从此刻开始 ~

新的风暴已经出现, 怎么能够停滞不前. 穿越时空竭尽全力, 我会来到你身边 ~

4.7 阅读理解 csv 解析

很久以前桌面项目配置文件多数采用 csv 文件配置. 采用 ',' 分隔. 同 excel 表格形式. 维护人员通过 notepad++ or excel 编辑操作. 我们直接读取开撸, 展示个自己写的解决方案, 灰常节约内存. 首先展示 **csv.h** interface.

```c
#pragma once

//
// csv readonly parse
// 1. 相邻字段必须被单个逗号分隔开，CRLF 换行
// 2. 每个被嵌入的双引号字符必须被表示为两个双引号字符
// 3. 字段都可以被双引号包裹，有回车换行符双引号或者逗号，那必须包裹
//

#include "struct.h"
#include "strext.h"

typedef struct {   // struct in heap malloc
    int    rlen;   // 数据行数，索引 [0, rlen)
    int    clen;   // 数据列数，索引 [0, clen)
    char * data[]; // 保存数据，rlen * clen 二维数组
} * csv_t;

//
// csv_get - 获取 csv[r][c] 位置的字符串
// csv     : csv_t 对象
// r       : 行索引 [0, csv->rlen)
// c       : 列索引 [0, csv->clen)
// return  : 返回 csv[r][c]，后续可以 atoi, atof, strdup ...
//
inline const char * csv_get(csv_t csv, int r, int c) {
    DCODE({
        if (!csv || r < 0 || r >= csv->rlen || c < 0 || c >= csv->clen) {
            RETNUL("params is error csv:%p, r:%d, c:%d.", csv, r, c);
        }
    });

    // 返回 csv[r][c] 索引位置字符串
    return csv->data[r * csv->clen + c];
}

//
// csv_delete - 释放 csv_t 对象
// csv     : csv_t 对象
// return  : void
//
inline void csv_delete(csv_t csv) {
    free(csv);
}

//
// csv_create - 文件中构建 csv_t 对象
// path    : 文件路径
// return  : 返回 csv_t 对象，NULL is error
//
```

```
extern csv_t csv_create(const char * path);
```

这里**只提供了读**接口, 比较有特色的思路是 csv_t 采用一整块内存构建. 非常干净.

```c
#include "csv.h"

// csv_parse_partial - 解析和检查 csv 文件内容，返回构造的合法串长度
static int csv_parse_partial(char * str, int * pr, int * pc) {
    int c;
    int rnt = 0, cnt = 0;
    char * tar = str, * s = str;
    while ((c = *tar++) != '\0') {
        // csv 内容解析，状态机切换
        switch (c) {
        case '"' : // 双引号包裹的特殊字符处理
            while ((c = *tar++) != '\0') {
                if ('"' == c) {
                    // 有效字符再次压入栈，顺带去掉多余 " 字符
                    if (*tar != '"')
                        break;
                    ++tar;
                }
                // 添加得到的字符
                *s++ = c;
            }
            // 继续判断,只有是 c == '"' 才会继续，否则都是异常
            if (c != '"')
                goto ret_faild;
            break;
        case ',' : *s++ = '\0'; ++cnt; break;
        case '\r': break;
        case '\n': *s++ = '\0'; ++cnt; ++rnt; break;
        default  : *s++ = c; // 其它所有情况只添加数据就可以了
        }
    }
    // CRLF 处理
    if (str != s && (c = tar[-2]) && c != '\n') {
        *s++ = '\0'; ++cnt; ++rnt;
    }

    // 检查，行列个数是否正常
    if (rnt == 0 || cnt % rnt) {
ret_faild:
        RETURN(-1, "csv parse error %d, %d, %d.", c, rnt, cnt);
    }

    // 返回最终内容
    *pr = rnt; *pc = cnt;
    return (int)(s - str);
}

// csv_parse - 解析字节流返回 csv 对象
csv_t csv_parse(char * str) {
    int n, rnt, cnt;
```

```
    if ((n = csv_parse_partial(str, &rnt, &cnt)) < 0)
        return NULL;

    // 分配最终内存
    csv_t csv = malloc(n + sizeof *csv + sizeof(char *) * cnt);
    if (csv == NULL) {
        RETNUL("malloc panic return null, n = %d, cnt = %d", n, cnt);
    }

    char * s = (char *)csv + sizeof *csv + sizeof(char *) * cnt;
    memcpy(s, str, n);

    // 开始内存整理, csv 字段填充
    n = 0;
    csv->rlen = rnt;
    csv->clen = cnt / rnt;
    do {
        csv->data[n] = s;
        while (*s++)
            ;
    } while (++n < cnt);

    return csv;
}

//
// csv_create - 文件中构建 csv_t 对象
// path    : 文件路径
// return  : 返回 csv_t 对象, NULL is error
//
csv_t
csv_create(const char * path) {
    char * str = str_freads(path);
    if (str) {
        // 开始解析 csv 文件内容, 并返回最终结果
        csv_t csv = csv_parse(str);
        free(str);
        return csv;
    }
    // 意外返回 NULL
    RETNUL("str_freads path = %s is error!", path);
}
```

核心重点 **csv_parse** 负责内存布局和协议解析. 代码很短, 但却很有效不是吗? 希望上面的阅读理解你能喜欢 ~

4.8 展望

妖魔战场逐渐急促起来, 练气顶级功法也就介绍到此. 数据结构和算法可能要勤学苦练, 而这些轮子多数只需 3 遍后, 战无不利, 终身会用. 本章多数在抠细节, 协助熟悉常用基础轮子开发套路. 从 log -> rand -> json -> utf8 -> conf -> base -> csv 遇到的妖魔鬼怪也不过如此. 真实开发中这类基础库, 要么是行业前辈遗留下来的馈赠. 要么就是远古大能的传世组件. 但总的而言, 如果你想把前辈英魂用的更自然, 显然你得懂行(自己会写). 不要担心瞎搞, 有足够喜欢最终都是殊途同归 <-- 😲

西门吹雪忽然道: "你学剑?"

叶孤城道: "我就是剑."

西门吹雪道: "你知不知道剑的精义何在?"

叶孤城道: "你说."

西门吹雪道: "在于诚."

叶孤城道: "诚?"

西门吹雪道: "唯有诚心正义, 才能到达剑术的颠峰, 不诚的人, 根本不足论剑."

叶孤城的瞳孔突又收缩.

西门吹雪盯着他, 道: "你不诚."

叶孤城沉默了很久, 忽然也问道: "你学剑?"

西门吹雪道: "学无止境, 剑更无止境."

叶孤城道: "你既学剑, 就该知道学剑的人只在诚于剑, 并不必诚于人."

思绪有些乱, 梦幻间想起 ~ 我们仨 ~ 是他们这些飞升地仙 ~, 撑起种族底蕴与苦难 ~