一. C 特点

C 語言不只是快，還具有指標，容易與組合語言連結，具有巨集、條件式編譯、inline 函數、結構化、可以使用記憶體映射輸出入。

二. 编译链接

2.1 编译

通过 CFLAGS 参数制定-g –Wall –I$(Path)/include 来制定头文件即可。

2.2 静态链接

2.2.1 说明

如果采用静态链接，即目标库文件为.a 文件，将所有的代码直接编译到可执行文件中，之后无需再指定动态库(so)文件的地址。

2.2.2 操作

链接时，通过 LDFLAGS 参数制定库文件路径，$(path)/xx.a，之后链接即可。

注意：**静态库直接指定.a 文件，不用-L 以及-lxx 哦**。

2.3 动态链接

2.3.1 说明

代码执行时采取动态加载代码，减少了第三方依赖，组件化，便捷化。

2.3.2 操作

2.3.2.1 LDFLAGS

操作原理和静态加载一样，但是路径的设置为-L$(path) -lxx。

2.3.2.2 LD_LIBRARY_PATH

修改环境变量 LD_LIBRARY_PATH 的值，从而保证程序执行时可以加载代码。

建议使用 ldconfig 命令来添加库搜索路径，关于 ldconfig 命令，请见笔记。

2.4 configure 配置

可以在 configure 命令行中配置 CFLAGS 和 LDFLAGS，但是别忘了-I 和-L 选项

2.4.1 例子

```
CPPFLAGS=-I/home/bamboo/grocery-shop/language/gcc/libcurl/libs/include \
LDFLAGS=-L/home/bamboo/grocery-shop/language/gcc/libcurl/libs/lib/ \
./configure --prefix=/home/bamboo/grocery-shop/language/gcc/libcurl/libs
make
make install
```

三. 编程规范（一致性）

3.1 文件命名

头文件——.h

程序文件——.c

文件名——全部小写，使用'_'分割

3.2 头文件

3.2.1 define 保护

3.2.1.1 命名风格

#define <PROJECT>_<PATH>_<FILE>_H_

3.2.1.2 例子

```
#ifndef _SRC_CURL_JSON_H_
#define _SRC_CURL_JSON_H_
```

3.2.2 前置声明

前置声明——在.c 文件的开头声明各种结构体、函数名等，用以替代#include 包含，尽可能的避免“包含依赖”错误。

3.2.2.1 错误

注意：Forward Declaration 是一个不完整的类型，只能使用引用或者指针，如果直接实例化则会碰到如下问题。

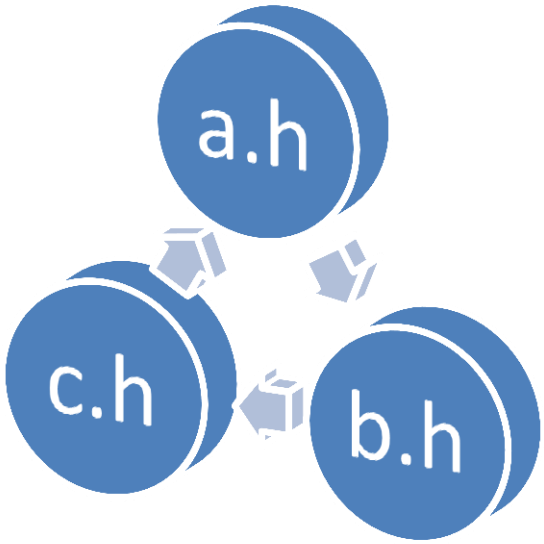
```
agg_msg_t                    head;
```

```
:85: 错误: 字段‘head’的类型不完全
```

3.2.3 依赖

#include 包含的很大一个弊端就是依赖文件代码的重新编译，各种重叠的文件包含是代码编译消耗的最大一部分原因，**尽量少的在.h 文件中包含其他.h 文件**。

3.2.3.1 问题-循环依赖



此类结构虽然在加上“#define”保护后，避免了编译错误，但是编译任意一个.c 文件中，任意一个.h 文件的修改都会涉及大量的代码重编译，从而降低了编译时间。

3.2.3.2 问题-定义变量

在.h 文件中定义变量，会导致某一个全局变量在编译阶段进行多次定义，降低了编译效率。

解决方式：

- 头文件声明全局变量
- 代码文件定义全局变量

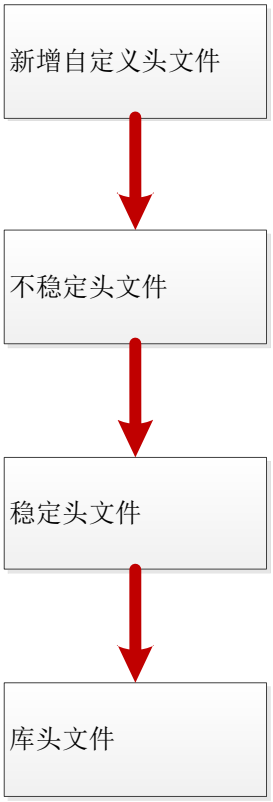
3.2.3.3 解决-单一功能头文件

在编写.h 文件，没必要一定要将所有的通用信息都放在某一个.h 文件中，尽量使每一个头文件的功能是单一的、最小化的，即使如此可能会增加.h 文件的数量，但是该方法能够尽量的减少依赖关系（看情况而定）。

3.2.3.4 解决-自包含

自包含——每一个.h 文件都是可以单独编译的，其不依赖任何.h 文件，独立的完成编译功能，该项限制非常像“单一功能”的描述。

3.2.4 包含顺序



此类设计原则的最主要目的：在编译时尽量早的显示出编译错误（新增代码、不稳定代码），而不用浪费时间去编译“稳定头文件”以及“库文件”。

3.2.5 接口化

类似面向对象接口的思想，每一个模块都对外提供一个唯一的接口“.h”文件，方便外部使用者、方便模块内部的代码修改不会影响到使用者（这里首先必须保证模块的功能尽可能的单一）。

3.2.5.1 功能单一模块

一个唯一的.h 文件。

3.2.5.2 多个子模块

如果一个模块包含多个子模块，那么就必须保证每一个子模块都提供一个唯一的接口”.h”文件。

3.3 命名规则

3.3.1 常用缩写

	缩写	全称
a		
	addr	address
	admin/adm	administrator
	app	application
	arg	argument
	asm	assemble(聚集、汇编)
	asyn	asynchronization(异步)
	avg	average
b		
	bg	background
	bk	back
	bmp	bitmap(位图)
	brk	break
	btn	button
	buf	buffer
c		
	calc	calculate(计算)
	char	character
	chg	change
	clk	click(点击)
	clr	color
	cmd	command
	cmp	compare
	col	column
	coord	coordinate
	cpy	copy
	ctrl/ctl	control

	cur	current
	cyl	cylinder(柱面)
d		
	dat	data(.dat 为数据文件)
	db	database
	dbg	debug
	dbl	double
	dec	decrease(降低)
	def	default
	del	delete
	dest/dst	destination
	dev	device
	dict	dictionary
	diff	different
	dir	directory
	disp	display
	div	divide(分割、除、划分)
	dlg	dialog
	doc	document
	drv	driver
	dyna	dynamic
e		
	edt	edit
	env	environment
	err	error
	esc	escape(逃跑，逃脱，泄露)
	ex/ext	extend
	exec	execute
f		
	flg	flag
	frm	frame
	func/fn	function
g		

grd	grid(了解、控制，紧握)
grp	group
h	
horz	horizontal(水平的，横线)
i	
idx/indx	index
img	image
impl	implement
inc	increase(增加)
info	information
init	initial/initialize/initialization
ins	insert
inst	instance
intr	interrupt
l	
len	length
lib	library
lnk	link
lst	list
m	
max	maximum
mem	memory
mgr	manager
mid	middle
min	minimum
msg	message
mul	multiply(乘、多倍的、增多)
n	
num	number
o	
obj	object

ofs	offset
org	origin/original
p	
param	parameter
pic	picture
pkg	package
pos	position
prev/pre	previous(先前的，前一个)
prg	program
prn	print
proc	process
prop	properties(特性)
pnt/pt	point
ptr	pointer
pub	public
pwd/psw	password
prev	previous
r	
rc	rect(矩形)
ref	reference
reg	register
req	request
res	resource(资源，办法)
ret	return
rgn	region
rst	result
s	
scr	screen
sec	second
seg	segment(段、节)
sel	select
sem	semaphore(信号量)
src	source

	srv	server
	stat	statistic
	std	standard
	stg	storage(存储)
	stm	stream
	str	string
	sub	subtract(减去，扣除)
	sum	summation(总结，求和)
	sync	synchronization(同步）--asyn
	sys	system
t		
	tbl	table
	temp	temperature(温度，气温)
	tmp	temporary(临时性，暂时的)
	trans	translate(翻译、转化)
	tst	test
	txt	text
u		
	unk	unknown
	upd	update
	upg	upgrade
	usr	user
	util	utiltity(效用，有用，功用)
v		
	var	variable
	ver	version
	vert	vertical
	vir	virus(病毒)
w		
	win/wnd	windows

3.3.2 常用反义词组

说明	变量名
增加、删除	add/remove
开头、结尾	begin/end
创建、销毁	create/destroy
插入、剪切	insert/delete
首部、尾部	first/last
分配、释放	get/release
递增、递减	increment/decrement
推入、弹出	put/get
增加、删除	add/delete
锁	lock/unlock
打开、关闭	open/close
最大、最小	max/min
旧、新	old/new
启动、关闭	start/stop
前续、后继	previous/next
源、目标	source/target
展示、隐藏	show/hide
发送、接收	send/receive
源、目的	source/destination
拷贝、黏贴	copy/paste
上、下	up/down

3.3.3 变量命名

结构体定义时可能采用“匈牙利”法（加前缀），但是变量不建议使用

3.3.3.1 全局变量

格式：g_变量名（因为非常丑陋）

3.3.3.2 静态变量

格式：s_变量名

普通变量

3.3.3.3 变量

格式：驼峰法，首字母小写，这是为了特意和函数、类型（结构体）的命名区分开来

词语组成：形容词+名词（**跟函数不同，并非一个动作，而是一个形容对象**）

例如：

sendPackage——发送的报文

servAddr——服务器的地址

sockLen——socket 的长度

注意：**最好的方式就是使用单节字符串**

3.3.4 函数命名

格式：动词+ '_' + 名词

3.3.5 宏命名

格式：大写+ '_' + 大写

3.4 注释

3.4.1 位置

文件头：版权、文件功能、版本信息

类定义：类功能、注意事项

成员变量定义：属性的功能描述

成员函数定义：函数的简要定义

函数实现：详细注释

代码实现：某些重要代码

3.4.2 常用指令

符号	说明
@file	文件名
@brief	简短说明（brief summary）
@param	参数名
@return	返回值的具体情况
@retval	返回值类型
@note	注解
@attention	注意
@warning	警告信息
@class	引入某个类
@exception	可能产生的异常
@todo	对将来要做的事情进行注释（程序员阅读）
@pre	前提条件
@post	代码执行之后的使用条件

3.5 文件后缀名

后缀格式	说明
.c	c 语言源文件
.a	静态库文件
.so	动态库文件
.h	头文件
.C/.cc/.cxx	c++源文件
.m	object-c 源文件
.i	预处理过的 c 源文件
.ii	预处理过的 C++源文件
.o	编译后的目标文件
.s	汇编语言源文件
.S	预编译后的汇编语言文件
.inc	c++中存放数据并包含的文件

四. 宏语法

4.1 typeof

4.1.1 语法

typeof 表达式|类型 变量

4.1.2 宏中用法

```
#define SWAP(a,b) {\
    typeof(a) _t=a;\
     a=b;\
    b=_t;}
```

从而实现任意类型的传递。

4.1.3 例子

typeof(int *) a, b 等价于 int * a , b;

4.2 __attribute__

4.2.1 功能和语法格式

功能：设置函数属性、变量属性、类型属性

格式：__attribute__((attribute-list))

位置约束：放在声明符号”;”之前

4.2.2 函数属性

函数属性可以帮助开发者把一些特性添加到函数声明中，从而可以使编译器在错误检查方面的功能更强大。

4.2.2.1 unused

如果某些函数，临时不可用，可以设置该函数属性，保证编译时不会出现告警信息，要知道告警信息是非常重要的哦，尤其是你还是一个菜鸟的时候

```
__attribute__((unused))
static void hexdump(FILE *f, const char *title,
        const unsigned char *s, int l)
```

4.2.3 变量属性

4.2.4 类型属性

aligned, packed, transparent_union, unused, deprecated 和 may_alias

五. 系统信息

5.1 主机名

5.1.1 函数

```
#include <unistd.h>
int gethostname(char *name, size_t len);
```

其中 name 不能为空内存，len 表示可以 name 最大的接收长度

5.1.2 例子

```
// hostname
rst = gethostname(kv->hostname, COMMON_NAME_SIZE);
if (rst < 0) {
    zlog_warn(kv->error, "Get hostname failed.");
    return false;
}
```

5.2 errno 操作

5.2.1 系统

- linux 环境：errno
- WIN32 环境：GetLastError

5.2.2 设计初衷

errno 的设计更多的是系统编程中的一种折衷之举。

- 在返回整数值的函数调用中（例如 NGTOS 中的各项返回值），可以通过不同的返回值，详细的表现出具体的错误信息；
- 但是在返回指针的函数调用中，调用者无法详细的了解具体的错误原因，于是 errno 便产生
- errno 用于存放错误原因，从而获取具体的错误描述

5.2.3 线程安全

一般而言，编译器会自动保证 errno 的使用是线程安全的：

```
#  if !defined _LIBC || defined _LIBC_REENTRANT
/* When using threads, errno is a per-thread value.  */
#   define errno (*__errno_location ())
#  endif
# endif /* !__ASSEMBLER__ */
#endif /* _ERRNO_H */
```

未定义_LIBC 或者定义_LIBC_REENTRANT 时是线程安全的。

5.2.4 正常使用

5.2.4.1 strerror

头文件	<string.h>以及<errno.h>
函数定义	char *strerror(int errnum);
返回值	返回 errnum 对应的字符串描述信息
参数说明	errnum，对应的 errno 编号，可以用 extern int errno 获取全局信息。

类比函数	int strerror_r(int errnum, char *buf, size_t buflen);
	或者
	char *strerror_r(int errnum, char *buf, size_t buflen);

5.2.4.2 perror

头文件	**<stdio.h>**
函数定义	void perror(const char *)
返回值	将 S 以及错误信息输出到 stderr 中
参数说明	S—用户自定义的信息说明

5.2.5 文件流使用

5.2.5.1 clearerr

头文件	**<stdio.h>**
函数定义	void clearerr(FILE * stream);
返回值	清除（复位）文件流的错误标志位，并使文件结束标志位为 0，无返回值
参数说明	某一个指定的文件流，和 ferror 以及进行判断验证

5.2.5.2 ferror

头文件	**<stdio.h>**
函数定义	int ferror(FILE *stream);
返回值	检测文件流是否发生了错误，错误返回非 0 值
参数说明	某一个指定的文件流

5.2.5.3 例子

```
void main( void )
{
   int c;
   // 向stdin写入数据时会产生错误
   putc( 'c', stdin );
   if( ferror( stdin ) )
   {
      perror( "Write error" );
      clearerr( stdin );
   }

   // 检测读操作是否有错
   printf( "Will input cause an error? " );
   c = getc( stdin );
   if( ferror( stdin ) )
   {
      perror( "Read error" );
      clearerr( stdin );
   }
}
```

六. 时间操作

6.1 时间戳

6.1.1 微妙（系统调用）

6.1.1.1 获取

函数声明：int gettimeofday(struct timeval*tv,struct timezone *tz)

其中 tv 中存放当前的秒级、微秒级时间戳、tz 中存放时区信息

结构体：

```
struct timeval{
    long    tv_sec;
    long    tv_usec;
};
struct timezone{
    int     tz_minuteswest;
    int     tz_dsttime;
};
```

注意：这是一个系统调用，不适用与频繁使用的场景中。

6.1.1.2 设置

七. 套接字编程

7.1 What is socket?

Well, they're this: a way to speak to other programs using standard Unix file descriptors.

"Jeez, *everything* in Unix is a file!"他们可以是网络连接、FIFO、pipe、terminal、real on-the-disk file、或者其他任何。

7.1.1 socket 为何不能像 file descriptors 使用？

答案：为何不可以，read 和 write，不过 send 和 recv 做了更好的封装

7.1.2 如何获取一个 socket fd？

答案：socket()，返回一个 socket fd。

7.1.3 参考

http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html

7.1.4 Layered Network Model

七层：

- Application
- Presentation
- Session
- Transport
- Network
- Data Link
- Physical

五层

- Application Layer (*telnet, ftp, etc.*)
- Host-to-Host Transport Layer (*TCP, UDP*)
- Internet Layer (*IP and routing*)
- Network Access Layer (*Ethernet, wi-fi, or whatever*)

7.2 socket 类型

7.2.1 Stream Socket

TCP（The Transmission Control Protocol）-SOCK_STREAM 连接

7.2.2 Datagram Socket

UDP（User Datagram Protocol）-SOCK_DGRAM，connectionless sockets 连接

7.3 协议族、地址结构

每一个协议族都定义了自己的“套接字地址结构”，这些结构体的命名格式：

Sockaddr_协议名称

7.3.1 网际套接字地址结构

7.3.1.1 地址族

AF_INET

7.3.1.2 数据结构

```
/* Structure describing an Internet socket address.  */
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;			/* Port number.  */
    struct in_addr sin_addr;		/* Internet address.  */

    /* Pad to size of `struct sockaddr'.  */
    unsigned char sin_zero[sizeof (struct sockaddr) -
                           __SOCKADDR_COMMON_SIZE -
                           sizeof (in_port_t) -
                           sizeof (struct in_addr)];
  };
```

```
#define __SOCKADDR_COMMON(sa_prefix) \
  sa_family_t sa_prefix##family
```

```
// (IPv4 only--see struct in6_addr for IPv6)

// Internet address (a structure for historical reasons)
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};
```

7.3.2 IPV6 套接字地址结构

7.3.2.1 地址族

AF_INET6

7.3.2.2 数据结构

```
#ifndef __USE_KERNEL_IPV6_DEFS
/* Ditto, for IPv6.  */
struct sockaddr_in6
  {
    __SOCKADDR_COMMON (sin6_);
    in_port_t sin6_port;    /* Transport layer port # */
    uint32_t sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr;  /* IPv6 address */
    uint32_t sin6_scope_id; /* IPv6 scope-id */
  };
#endif /* !__USE_KERNEL_IPV6_DEFS */
```

```
struct in6_addr {
    unsigned char   s6_addr[16];   // IPv6 address
};
```

7.3.3 通用套接字地址结构

由于该结构在 ANSI C 之前定义，目前在 IPV4 的很多套接字函数中仍然大量存在，适用于 ipv4 和 ipv6 应用场景，故编程时存在大量的“指针强制转换”。

```
/* Structure describing a generic socket address.  */
struct sockaddr
  {
    __SOCKADDR_COMMON (sa_);    /* Common data: address family and length.  */
    char sa_data[14];       /* Address data.  */
  };
```

强制装换：

sockaddr_in -> sockaddr

7.3.4 addrinfo 地址结构

一般和 getaddrinfo 函数联用

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname

    struct addrinfo *ai_next;      // linked list, next node
};
```

7.4 Socket 辅助函数

7.4.1 地址转换

7.4.1.1 函数说明

头文件	#include <arpa/inet.h>
函数定义	int bind(int sockfd, struct sockaddr * my_addr, int addrlen);
返回值	成功返回 0，失败返回-1，存在 errno 中
参数说明	sockfd：由 socket 函数获取的 sock 句柄； my_addr：初始化的地址服务器地址结构体 addrlen：my_addr 的大小

7.4.1.2 示例

```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6

inet_pton(AF_INET, "10.12.110.57", &(sa.sin_addr)); // IPv4
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```

```
// IPv4:

char ip4[INET_ADDRSTRLEN];  // space to hold the IPv4 string
struct sockaddr_in sa;      // pretend this is loaded with something

inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);

printf("The IPv4 address is: %s\n", ip4);


// IPv6:

char ip6[INET6_ADDRSTRLEN]; // space to hold the IPv6 string
struct sockaddr_in6 sa6;    // pretend this is loaded with something

inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);

printf("The address is: %s\n", ip6);
```

7.4.2 getpeername(Who are you?)

7.4.3 gethostname(Who am I?)

7.5 TCP 核心通信编程

7.5.1 bind

7.5.1.1 函数说明

头文件	<sys/types.h>，<sys/socket.h>
函数定义	int bind(int sockfd, struct sockaddr * my_addr, int addrlen);
返回值	成功返回 0，失败返回-1，存在 errno 中
参数说明	sockfd：由 socket 函数获取的 sock 句柄； my_addr：初始化的地址服务器地址结构体 addrlen：my_addr 的大小

7.5.1.2 示例

新的方式：

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

旧有方式：

```
// !!! THIS IS THE OLD WAY !!!

int sockfd;
struct sockaddr_in my_addr;

sockfd = socket(PF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT);     // short, network byte order
my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr);
```

7.5.2 listen(Will somebody please call me?)

7.5.2.1 函数说明

头文件	<sys/socket.h>
函数定义	int listen(int sockfd, int backlog);
返回值	成功返回 0，失败返回-1，存在 errno 中
参数说明	sockfd：由 socket 函数获取的 sock 句柄； backlog：同时能够处理的最大连接请求，一般设置为 20 以下

7.5.2.2 示例

7.5.3 accept(Thank you for calling port 3977)

7.5.3.1 函数说明

头文件	<sys/types.h>，<sys/socket.h>
函数定义	int accept(int csockfd, struct sockaddr * addr, int * addrlen);
返回值	成功返回新的 socket 句柄，用于此时连接，失败返回-1，存在 errno 中
参数说明	csockfd：由 socket 函数获取的 sock 句柄； addr：远程客户端地址信息 addrlen：addr 的大小

7.5.3.2 示例

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPORT "3490"  // the port users will be connecting to
#define BACKLOG 10     // how many pending connections queue will hold

int main(void)
{
    struct sockaddr_storage their_addr;
    socklen_t addr_size;
    struct addrinfo hints, *res;
    int sockfd, new_fd;

    // !! don't forget your error checking for these calls !!

    // first, load up address structs with getaddrinfo():

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

    getaddrinfo(NULL, MYPORT, &hints, &res);

    // make a socket, bind it, and listen on it:

    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    bind(sockfd, res->ai_addr, res->ai_addrlen);
    listen(sockfd, BACKLOG);

    // now accept an incoming connection:

    addr_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);
```

7.5.4 connect(Hey,you!)

7.5.4.1 函数说明

头文件	<sys/types.h>，<sys/socket.h>
函数定义	int connect(int sockfd, struct sockaddr * serv_addr, int addrlen);
返回值	成功返回 0，失败返回-1，存在 errno 中
参数说明	sockfd：由 socket 函数获取的 sock 句柄； addr：远程 server 地址信息 addrlen：addr 的大小

7.5.4.2 示例

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("www.example.com", "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// connect!

connect(sockfd, res->ai_addr, res->ai_addrlen);
```

7.5.5 send(Talk to me,baby!)

7.5.5.1 函数说明

头文件	<sys/types.h>，<sys/socket.h>
函数定义	int send(int csockfd, const void * msg, int len, unsigned int flags);
返回值	成功返回实际发送的字符数，失败返回-1，存在 errno 中
注意	send 和 recv 本身对网络字节顺序进行转换

参数说明	csockfd：由 socket 函数获取的 sock 句柄； msg：数据 len：msg 的字符长度，不包括'\0' flags：一般为 0

7.5.5.2 示例

```
char *msg = "Beej was here!";
int len, bytes_sent;
.
.
.
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
```

7.5.6 recv(Talk to me)

头文件	**<sys/types.h>，<sys/socket.h>**
函数定义	int recv(int csockfd, void *buf, int len, unsigned int flags);
返回值	成功返回实际接收的字符数，失败返回-1，存在 errno 中
参数说明	csockfd：由 socket 函数获取的 sock 句柄； buf：接收缓冲区 len：buf 接收缓冲区最大长度，一般为 sizeof(buf)-1 flags：一般为 0

7.5.7 示例

7.5.7.1 初始化

```
listenfd = socket(AF_INET, SOCK_STREAM, 0);
if(listenfd == -1)
{
    zlog_warn(kv->error, "Tcp main server creating socket failed.");
    kv->quit = true;
    return false;
}

// reuse addr
opt = SO_REUSEADDR;
setsockopt(listenfd,SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

// init
bzero(servaddr, sizeof(struct sockaddr_in));
servaddr->sin_family      = AF_INET;
servaddr->sin_addr.s_addr = htonl(INADDR_ANY);
servaddr->sin_port        = htons(td->port);

// bind
rst = bind(listenfd, (struct sockaddr*)servaddr,
                sizeof(struct sockaddr_in));
if (rst < 0) {
    zlog_info(kv->error, "Bind socket(port:%d, addrss:any) failed.\n",
            td->port);
    kv->quit = true;
    return false;
}

// listen
rst = listen(listenfd, TCP_LISTEN);
if (rst < 0) {
    zlog_info(kv->error, "Listen tcp port failed.\n");
    kv->quit = true;
    return false;
}
```

7.5.7.2 IO 操作

```
// handle
FD_ZERO(&allset);
FD_SET(listenfd, &allset);
while (!kv->quit) {
    rset = allset;
    timeOut.tv_sec = 2;
    timeOut.tv_usec = 0;
    select(listenfd+1, &rset, NULL, NULL, &timeOut);
    if (FD_ISSET(listenfd, &rset)) {
        addrlen = sizeof(cliaddr);
        connectfd = accept(listenfd, (struct sockaddr*)&cliaddr,
                           &addrlen);
        if (connectfd < 0) {
            zlog_info(kv->error, "Tcp server accept connect failed.\n");
            continue;
        }
        /*DEBUG_MSG("Tcp server receive a new connect.\n");*/
        thandle = (kv_tcp_thread_t*)malloc(sizeof(kv_tcp_thread_t));
        thandle->kv = kv;
        thandle->fd = connectfd;
        thpool_add_work(kv->thpool, (void*)thd_tcp_handle_server,
                        (void*)thandle);
    }
    FD_SET(listenfd, &allset);
}
```

```
pLen = recv(fd, iNode.msg, INPUT_MSG_LENGTH-1, 0);
if (pLen < 0) {
    zlog_info(kv->error, "Tcp server receive package failed.\n");
    break;
} else if (pLen == 0) {
    break;
} else {
    iNode.msg[pLen] = '\0';

    if (strncasecmp(iNode.msg, TCP_QUIT, quitLen) == 0) {
        zlog_debug(kv->debug, "Tcp handle server ready to quit.\n");
        break;
    } else {
        add_msg_to_iqueue(kv, &iNode);
    }
}
```

7.6 UDP 编程

7.6.1 bind

见 TCP 中的说明

7.6.2 recvfrom

头文件	<sys/types.h>，<sys/socket.h>
函数定义	int recvfrom(int csockfd, void *buf, int len, unsigned int flags, struct sockaddr *from,int *fromlen);
返回值	成功返回实际接收的字符数，失败返回-1，存在 errno 中
参数说明	csockfd：由 socket 函数获取的 sock 句柄； buf：接收缓冲区 len：buf 接收缓冲区最大长度，一般为 sizeof(buf)-1 from：远程地址信息 fromlen：from 的长度信息

7.6.3 sendto

头文件	<sys/types.h>，<sys/socket.h>
函数定义	int sendto(int cscokfd, const void * msg, int len, unsigned int flags, const struct sockaddr * to, int tolen);
返回值	成功返回实际发送的字符数，失败返回-1，存在 errno 中
参数说明	csockfd：由 socket 函数获取的 sock 句柄； msg：接收缓冲区 len：msg 最大长度，strlen(msg) flags：一般设置为 0 to：远程地址信息 tolen：to 的长度信息

7.6.4 示例

7.6.4.1 初始化

```
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if (sockfd < 0) {
    zlog_info(kv->error, "Socket error.\n");
    kv->quit = true;
    return false;
}

bzero(servaddr, sizeof(struct sockaddr_in));
servaddr->sin_family      = AF_INET;
servaddr->sin_addr.s_addr = htonl(INADDR_ANY);
servaddr->sin_port        = htons(td->port);

rst = bind(sockfd, (struct sockaddr*)servaddr,
                   sizeof(struct sockaddr_in));
if (rst < 0) {
    zlog_info(kv->error, "Bind socket(port:%d, addrss:any) failed.\n",
            td->port);
    kv->quit = true;
    return false;
}
```

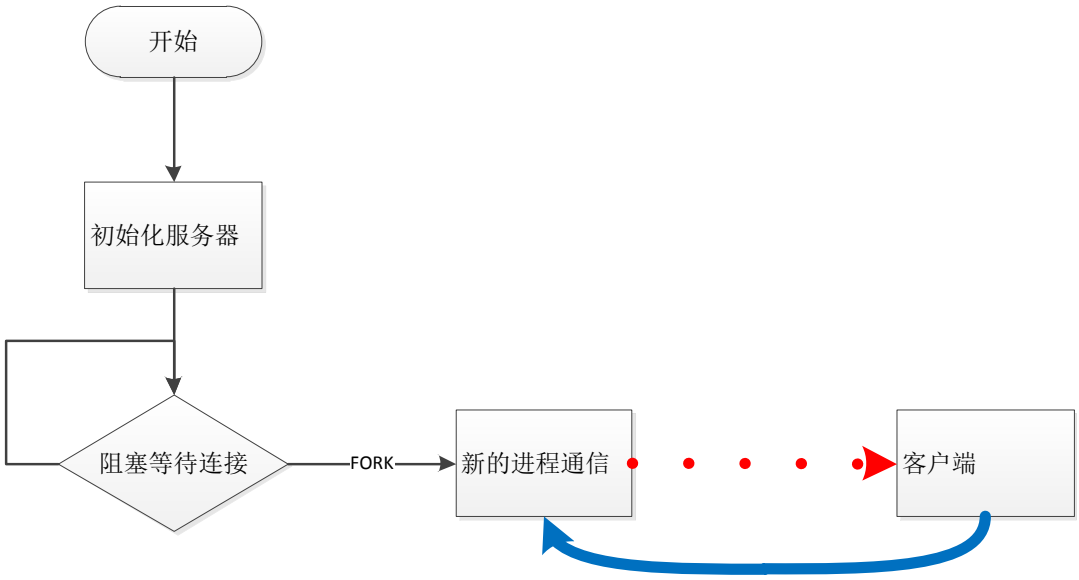
7.6.4.2 IO 操作

```
// handle
FD_ZERO(&allset);
while (!kv->quit) {
    FD_SET(sockfd, &allset);
    rset = allset;
    timeOut.tv_sec = 2;
    timeOut.tv_usec = 0;
    rst = select(sockfd+1, &rset, NULL, NULL, &timeOut);
    if (!rst) {
        continue;
    }
    // get
    cliLen = 0;
    pLen = recvfrom(sockfd, iNode.msg, INPUT_MSG_LENGTH-1, 0,
                    (struct sockaddr*)&cliaddr, &cliLen);
    if (pLen <= 0) {
        zlog_info(kv->error, "Udp server receive package failed.\n");
        continue;
    }
    iNode.msg[pLen] = '\0';
    zlog_debug(kv->debug, "Receive a package(Length=%ld)"
                ", msg:%s.", pLen, iNode.msg);
    add_msg_to_iqueue(kv, &iNode);
}
```

7.7 非阻塞 socket

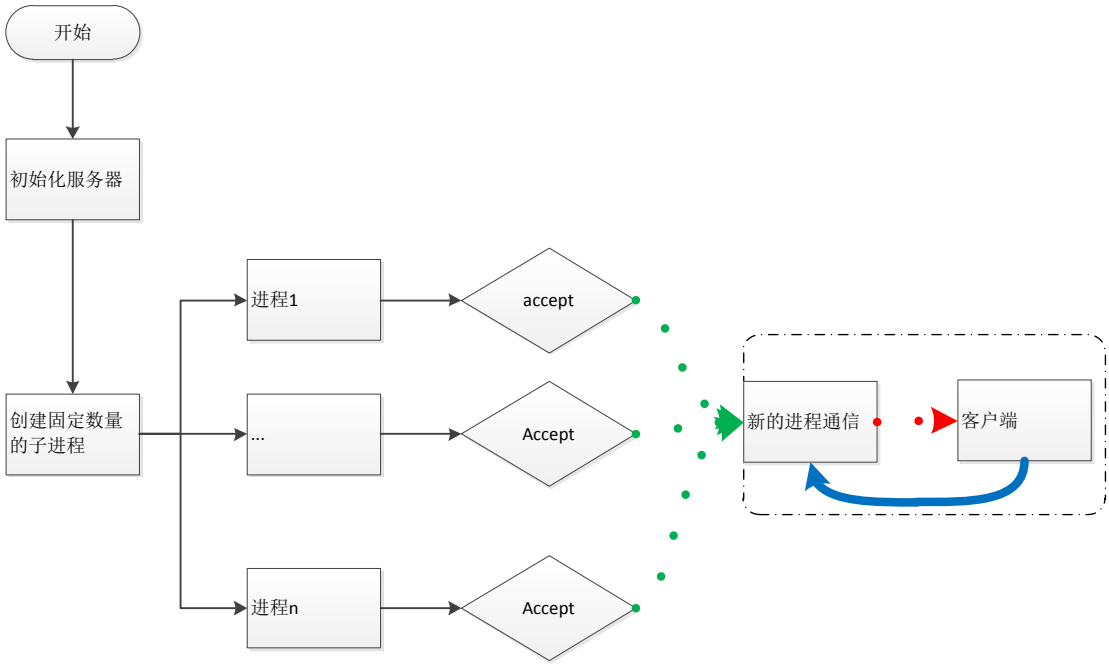
7.8 TCP 并发服务器模式

7.8.1 单连接单进程



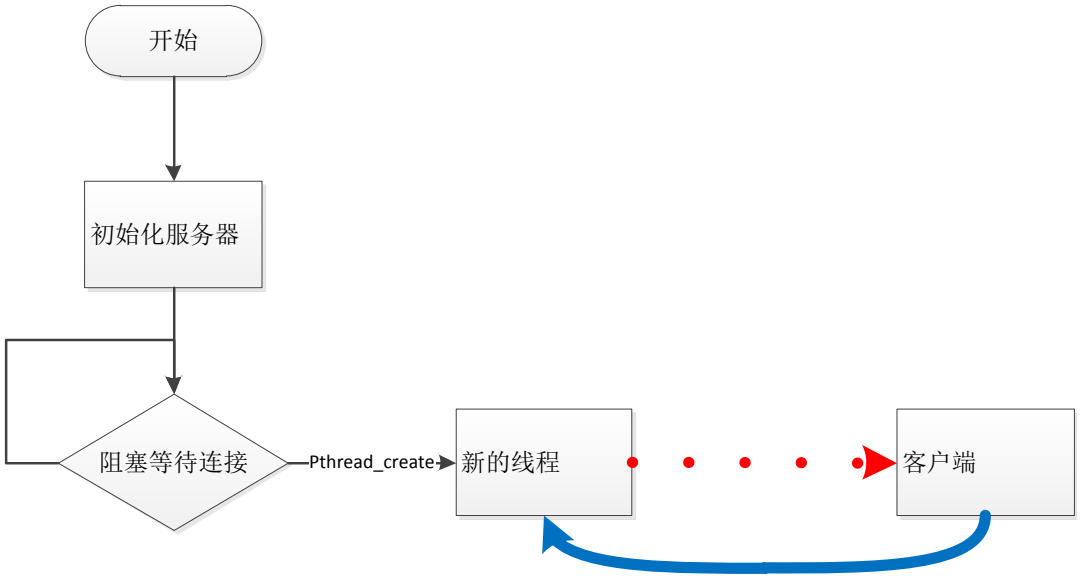
7.8.2 预创建子进程数目

动态的预创建子进程数目，尽量避免“惊群”现象的发生，其中 accept 可以进行上锁避免“惊群”的发生，具体有互斥锁、文件锁等



7.8.3 单连接单线程

其中主线程中的 accept 阻塞等待所有的 connect 连接。

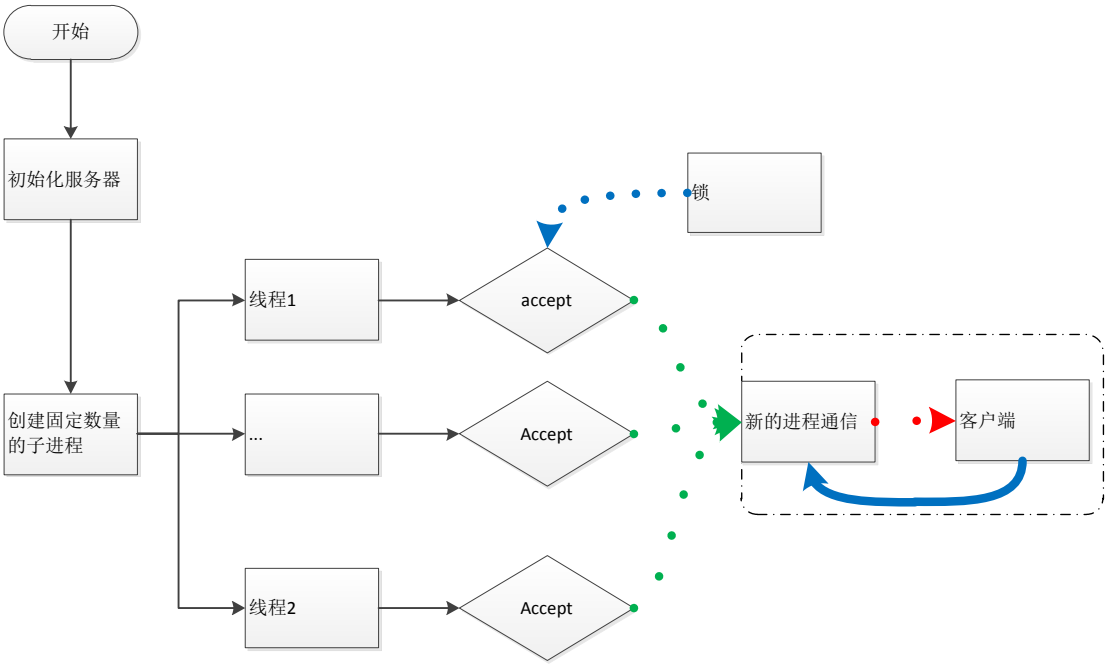


7.8.4 预创建线程数

模式：Leader/Follower 模式

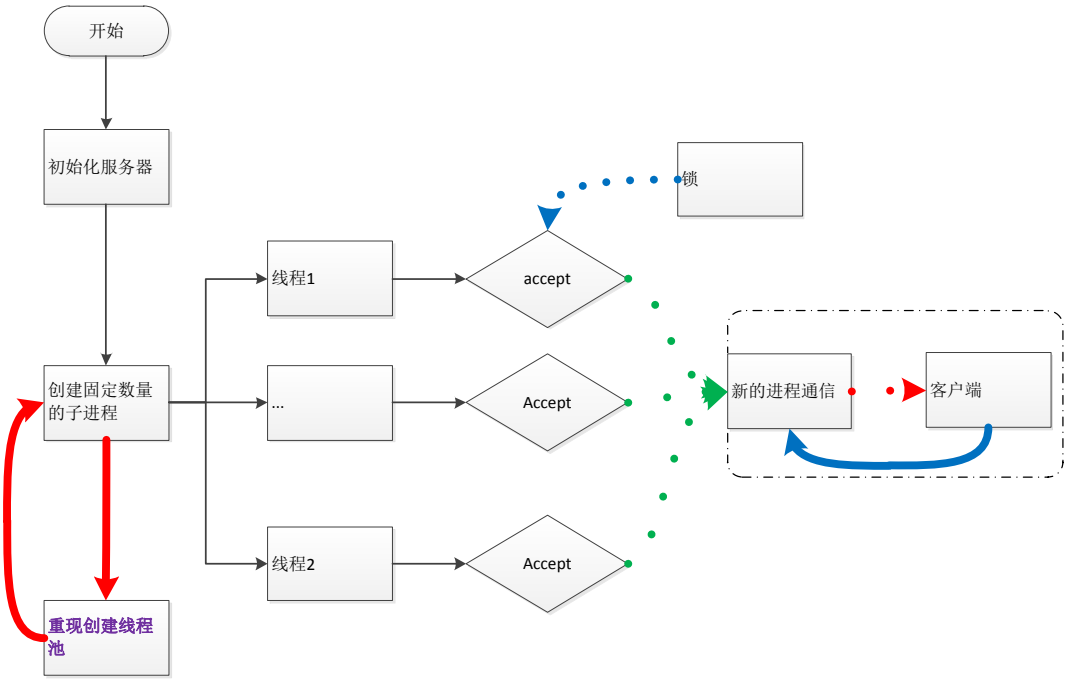
accept:互斥锁，否则会发生“惊群”现象

connect：各个子线程单独进行 accept 接收连接



7.8.5 动态的保持线程池

动态的增加或者减少线程池中线程数量，合理的利用资源，在非稳定的运行环境中，该模式相比上一种较好。



八. 数据类型

8.1 固定位数整数

8.1.1 类型

头文件	数据类型	说明
sys/types.h	int8_t	有符号 8 位整数
	uint8_t	无符号 8 位整数
	int16_t	有符号 16 位整数
	uint16_t	无符号 16 位整数
	int32_t	有符号 32 位整数
	uint32_t	无符号 32 位整数

8.2 套接字

8.2.1 类型

头文件	数据类型	说明
sys/socket.h	sa_family_t	套接字地址结构的地址族
	socket_len_t	套接字地址结构的长度
	scokaddr	通用套接字地址结构
	sockaddr_storage	新的通用套接字地址结构

8.3 IP 地址

8.3.1 类型

头文件	数据类型	说明
netinet/in.h	in_addr_t	IPV4 地址，一般为 uint32_t
	in_addr	IPV4 地址的结构

	in_port_t	TCP 或者 UDP 端口，uint16_t
	in6_addr	IPV6 地址的结构
	sockaddr_in	IPV4 套接字地址结构
	sockaddr_in6	IPV6 套接字地址结构

8.3.2 例子

```
/* Internet address.  */
typedef uint32_t in_addr_t;
struct in_addr
  {
    in_addr_t s_addr;
  };
```

九. 格式化

9.1 语法

```
格式段落的語法: %[flags][width][.precision][length]specifier
  % 代表變數開始
  [flag]
    -:靠左
    +:輸出正負號
    (space):當不輸出正負號時，就輸出空白，首先必須设置了width
        在 8 或 16 進位 (o,x,X) 時，強制輸出 0x 作為開頭，
        在浮點數 (e, E, f) 時，強制輸出小數點，
        在浮點數 (g, G) 時，強制輸出小數點，但尾端的 0 會被去掉。
    0:在開頭處(左側) 補 0，而非補空白。
  [width]
    最小輸出寬度 (或 *)
  [.precision]
    精確度，小數點之後的輸出位數
  [length]
    長度符號 h, I, L
  [specifier]
    型態描述元，可以是 c, d, e, E, f, g, G, o, s, u, x, X 等基本型態
```

其中，**如果 width 为“*”，表必须输入两个参数，首参数表示域宽度**。

https://zh.wikipedia.org/wiki/%E6%A0%BC%E5%BC%8F%E5%8C%96%E5%AD%97%E7%AC%A6%E4%B8%B2（维基百科）

9.2 格式化函数

9.2.1 总览

分类	函数	说明
开启和关闭		
	fopen()	打开一个文件并返回文件指针
	fclose()	关闭文件流
	freopen()	文件流重定向，流替换
	fread()	从文件流中读取数据
	fwrite()	向文件流中写入数据
缓冲区		
	setbuf()	把缓冲区与流相关联
	setvbuf()	设置文件流的缓冲区
	fflush()	清空缓冲区
退回		
	ungetc()	把字符退回到输入流
	ungetch()	把一个字符退回到键盘缓冲区
文件指针位置		
	feof()	检查流上文件的结束标识
	fgetpos()	获得当前文件的读写指针
	fsetpos()	设置当前文件的读写指针
	fseek()	移动文件的读写指针到指定的位置
	ftell()	获取文件读写指针的当前位置
	rewind()	将文件指针重新指向文件开头
错误处理		
	clearerr()	清除（复位）文件流的错误标识
	ferror()	检测文件流是否出错
	perror()	打印最近一次系统错误信息
字符操作		
	getc()	从流中读取字符
	getchar()	从控制台读取字符并立即回显
	fgetc()	从文件流中读取一个字符
	fgetchar()	从文件流中读取一个字符
	getche()	从控制台读取字符并立即回显
	putc()	写文件函数(将一指定字符写入文件中)

	putchar()	向控制台输出一个字符
	fputchar()	将一个字符输出到标准输出流中
字符串操作		
	gets()	从流中读取字符串
	fgets()	从文件流中读取一行或指定个数的字符
	puts()	将一个字符串放入标准输出流(stdout)中
	fputs()	将指定的字符串写入到文件流
格式化操作		
	printf()	格式化输出函数
	sprintf()	将格式化的数据写入字符串
	snprintf()	将格式化的数据写入字符串（长度）
	fprintf()	将格式化数据输出到文件流
	scanf()	格式化输入函数
	sscanf()	从字符串中读取指定格式的数据
	fscanf()	将文件流中的数据格式化输入
文件属性操作		
	remove()	删除文件或目录
	rename()	重命名文件或目录
整数操作		
	getw()	以二进制形式从文件流中读取整数
	putw()	以二进制形式向文件中写入整数
临时文件		
	temfile()	以二进制形式创建一个临时文件并打开
	tmpnam()	产生一个唯一的包含路径的文件名

链接地址：http://c.biancheng.net/cpp/u/stdio_h/

9.2.2 字符串

9.2.2.1 snprintf

9.2.2.1.1 原型

snprintf()函数用于将格式化的数据写入字符串，其原型为：

```
int snprintf(char *str, int n, char * format [, argument, ...]);
```

【参数】str为要写入的字符串；n为要写入的字符的最大数目，超过n会被截断；format为格式化字符串，与printf()函数相同；argument为变量。

即数组 star 长度为 n+1，则传入的参数为 n

9.2.2.1.2 例子

```c
#include <stdio.h>
int main()
{
    char str[5];
    int ret = snprintf(str, 3, "%s", "abcdefg");
    printf("%d\n",ret);
    printf("%s",str);
    return 0;
}
```

9.3 字符串和基本数据类型

9.3.1 字符串转整型

9.3.1.1 总览（stdlib.h）

函数	说明
atof	字符串转 double 类型，其中 **a 代表 ascii**
atoi	字符串转 int 类型
atol	字符串转 long 类型
strtod	字符串转为 double 类型，操作不同于上面
strtol	字符串转为 long 类型
strtoul	字符串转为 unsigned long 类型

9.3.1.2 说明

http://c.biancheng.net/cpp/html/129.html

9.3.2 整型转字符串

使用 snprintf 格式化字符串来进行转化，函数说明请见前面介绍，例子：

```
snprintf(timeStr, TIME_LENGTH-1, "%012ld", time);
snprintf(rowkey, ROWKEY_LENGTH-1, "%d:%s", kv, timeStr);
```

9.4 字符串切割

函数	说明
strtok	分割字符串，会改变原始字串，一般和 strdup 或者 strdupa 函数配合（单线程）； **首次调用，就会替换字符串中所有的字符，之后逐个取出**；
strsep	同 strtok，速度快于前者，替换 strtok 函数

9.5 字符串验证

9.5.1 字符验证

函数	说明
isspace	是否为空白字符
ispunct	是否为特殊符号（自定义）
isdigit	是否为阿拉伯数字
isxdigit	是否为 16 进制数

9.6 字符串查找

具体函数定义说明，请到 api.chm 文档或者

http://wiki.jikexueyuan.com/project/c/c-standard-library-string-h.html 查看。

9.6.1 字符查找

函数	说明
rindex	获取字符最后一次出现的指针

index	获取字符第一次出现的指针
strchr	同 index，只不过不比较'\0'字符，注意
strrchr	同 rindex，无 rindex(s1, '\0')的用法
memchr	在某一个内存范围内查找某一个字符
strpbrk	检索第一个字符，匹配目标为字符组成的字符串
strspn	返回字符串中连续包含指定字符串内容的字符数，返回的是匹配的字符数目
strcspn	返回字符串中连续不包含指定字符串内容的字符数，返回的是匹配的字符数目

9.6.2 字符串查找

函数	说明
strstr	查找首个字串并返回相应指针
lsearch	线性查找，其他和 bsearch 类同
bsearch	二分查找，必须保证数组是有序的，一般和 qsort 函数联用； **必须提供比较的回调函数**；

9.7 字符串动态拷贝

函数	说明
strdup	分配内存并拷贝
strdupa	allocate 分配并拷贝

单线程安全。

9.8 字符串比较

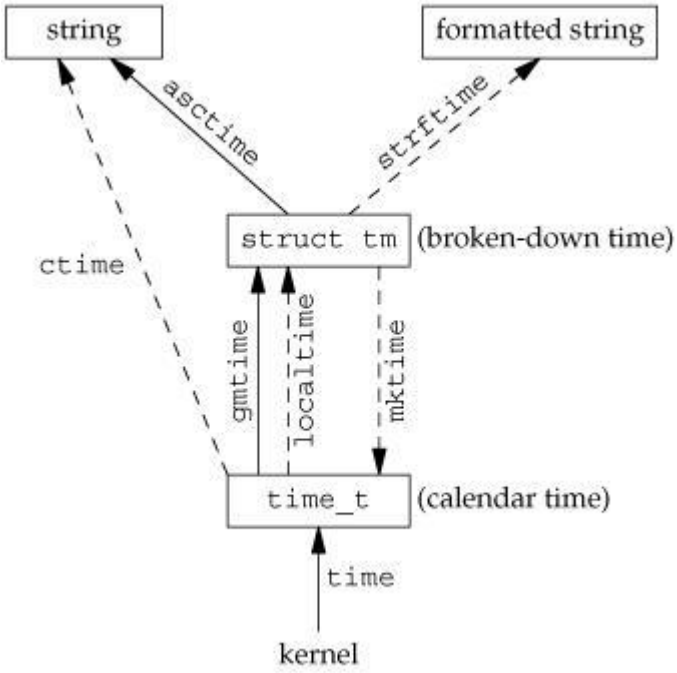
函数	说明
strnicmp	不区分大小写的比较固定长度的串
strncmpi	同上

strcasecmp	不区分大小写比较
strcmpi	同上
strcmp	比较字符串
strncmp	比较固定长度的字符串

十. 时间操作

10.1 转换图

该转换图也同样适用于 python 中大部分的时间操作。



10.2 日期格式化符号

%y 两位数的年份表示（00-99）
%Y 四位数的年份表示（000-9999）
%m 月份（01-12）
%d 月内中的一天（0-31）
%H 24 小时制小时数（0-23）
%I 12 小时制小时数（01-12）

%M 分钟数（00=59）
%S 秒（00-59）

%a 本地简化星期名称
%A 本地完整星期名称
%b 本地简化的月份名称
%B 本地完整的月份名称
%c 本地相应的日期表示和时间表示
%j 年内的一天（001-366）
%p 本地 A.M.或 P.M.的等价符
%U 一年中的星期数（00-53）星期天为星期的开始
%w 星期（0-6），星期天为星期的开始
%W 一年中的星期数（00-53）星期一为星期的开始
%x 本地相应的日期表示
%X 本地相应的时间表示
%Z 当前时区的名称
%% %号本身

例如：

"***%a,%d-%b-%Y %T GMT***"的输出格式为：'***Mon,19-Oct-2015 10:07:43 GMT***'

"***%Y%m%d%H%M%S***"的输出格式为：'***20151019100743***'

10.3 mktime

头文件	<time.h>
函数定义	time_t mktime(struct tm *)
返回值	将 tm 结构转化为 timestamp（秒），返回从 1970 年到现在的秒数
参数说明	tm 结构体

10.4 localtime

10.4.1 单线程

头文件	<time.h>
函数定义	struct tm *localtime(const time_t * timep);
返回值	将 time_t 数值转为 tm 结构，一般必须和 time 联用，返回系统分配的 tm
参数说明	time()生成的秒数

10.4.2 多线程

头文件	<time.h>
函数定义	struct tm *localtime_r(const time_t *timep, struct tm *result);
返回值	返回 NULL 或者 tm 结构
参数说明	tm 结构体对象以及 time 生成的秒数

10.5 线程安全

10.5.1 说明

mktime、localtime_r 在实现上都考虑了时区的切换，而时区的计算都会涉及全局变量 tzname/timezone/daylight 的使用，所以本质上都不是线程安全的。

10.5.2 解决

锁机制

十一. 内存操作

11.1 拷贝

11.1.1 memcpy

原型：**void *memcpy(void*dest, const void *src, size_t n);**

头文件：<sring.h>

功能：

由 src 指向地址为起始地址的连续 n 个字节的数据复制到以 destin 指向地址为起始地址的空间内

注意：

memcpy 一定会拷贝 n 个字节的数据到 dest 中，其不去关注 src 的长度，此种方法可能会造成内存越界拷贝。

漏洞：

如果【dest, dest+n】内存区域、【src, src+n】存在重叠，见下面的说明

11.1.2 memccpy

原型：void *memccpy(void *dest, const void *src, int c, size_t n);

头文件：#include <string.h>

功能：

主要功能同 memcpy 相同，**但是一旦出现字符 c，马上返回**，这个非常有用，可以将该函数看成 strcpy 和 memcpy 的结合体。

11.1.3 memmove

原型：void *memmove(void *dest, const void *src, size_t n);

头文件：#include <string.h>

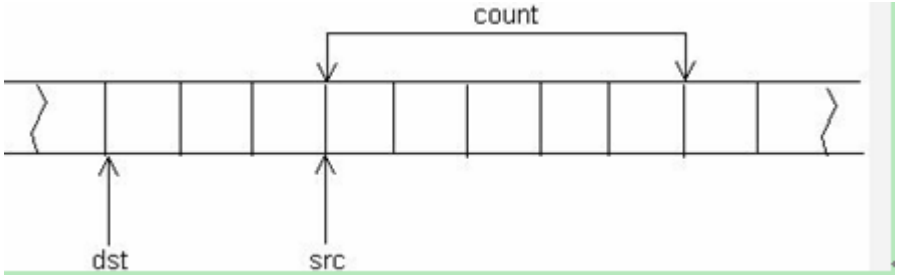
功能：

同 memcpy

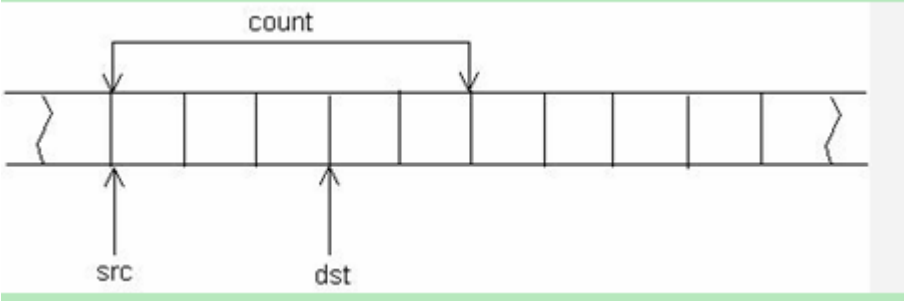
11.1.4 memmove、memcpy 区别

11.1.4.1 情况 1

（结果相同，都会改变 src 中的值）：



11.1.4.2 情况 2



- memcpy 拷贝的结果有误
- 一个从 0 开始拷贝，一个从结尾开始往下拷贝，结果出现两种现象
- 两者都可能改变 src 中的值

11.1.5 strcpy

原型：char *strcpy(char *dest,const char *src);

头文件：#include<string.h>

功能：

将 src 字符串拷贝到 dest 中，注意和 strncpy 的区别

11.1.6 strncpy

原型：char * strncpy(char *dest,const char *src,size_t n);

头文件：#include<string.h>

功能：

将 src 的前 n 个字符拷贝到 dest 中。

注意：

如果 n >=sizeof(src)，则无需做任何操作；

如果 n<sizeof(src)，则需要注意，**不会再 dest 的末尾添加'\0'**

11.1.7 其他

bcopy/bzeo 等

十二. 数据库

12.1 mysql

12.1.1 编译链接

12.1.1.1 库文件

shell 输出：

```
[bamboo@devops-bifeng gcc]$ mysql_config  --libs
-rdynamic -L/usr/lib64/mysql -lmysqlclient -lz -lcrypt -lnsl -lm -lssl -lcrypto
```

编译时：

```
MYSQL_LIB=`mysql_config --libs`
MYSQL_CFLAGS=`mysql_config --cflags`
```

12.1.1.2 头文件

shell 输出：

```
[bamboo@devops-bifeng gcc]$ mysql_config  --cflags
-I/usr/include/mysql  -g -pipe -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -f
 -D_GNU_SOURCE -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE -fno-strict-
```

编译时：

```
MYSQL_LIB=`mysql_config --libs`
MYSQL_CFLAGS=`mysql_config --cflags`
```

十三. 排序算法

13.1 快速排序

13.1.1 qsort 函数

原型：

void qsort(void *base, size_t n, size_t size, int (*compare)(const void *, const void *));

头文件：<stdlib.h>

功能：

qsort 只对数组进行排序，第一个参数 base 是数组的首地址，第二个参数 n 是待排序元素个数，第三个参数 size 是每个元素的大小(以字节为单位)，第四个参数 compare 是元素的比较函数，也就是判断标准

例子（整数数组）：

```
int cmp(const void *pa, const void *pb)
{
    return *(int*)pa - *(int*)pb;
}
```

```
int data[] = {10, 8, 20, 7, 4, 3, 100, 80};
int i;

for(i=0; i<sizeof(data)/sizeof(data[0]); i++)
    printf("%d ", data[i]);

qsort(data, sizeof(data)/sizeof(data[0]), sizeof(data[0]), cmp);    // asc
```