**openwrt常用库用法**

1. **libubox**

**1.1. 概述**

libubox是openwrt的一个基础库，openwrt下大部分应用都是基于它开发的（ubus、uhttpd、uci等）。

libubox主要提供了两类功能：

[1]. 一套完整的基于事件驱动的机制

[2]. 多个常用的功能模块(链表、avl树、json、消息传输单元、md5)

**1.2. 基于事件驱动机制**

基于事件驱动机制是libubox的核心部分，这套机制主要实现了一套通用的非阻塞I/O多路复用（epoll）平台。

以下是这部分的常用API整理

|  |  |
| --- | --- |
| API | 用法 |
| int uloop\_init(void) | 创建epoll句柄 |
| void uloop\_done(void) | 注销epoll模块 |
| int uloop\_fd\_add(struct uloop\_fd \*sock,unsigned int flags) | 将需要监听的fd以及要监听的事件注册到epoll（默认拥有水平触发和非阻塞的特性）  @sock - 需要被epoll监听的fd管理块，主要记录了自己创建的fd、事件处理回调  @flags - 监听的事件类型，必须至少包含ULOOP\_READ或ULOOP\_WRITE或ULOOP\_PRI（这个是本人补丁追加的） |
| int uloop\_fd\_delete(struct uloop\_fd \*sock) | 从epoll监听池中删除指定的fd  @sock - 要从监听池中删除的fd管理块 |
| void uloop\_run(void) | 事件驱动模型的主循环，在这里面进行[ 等待事件/超时->处理事件/超时->等待事件]的循环，通过信号退出 |
| int uloop\_timeout\_set(struct uloop\_timeout \*timeout,int msecs) | 给定时器设置一个超时值，并激活  @timeout - 定时器模块，主要记录了超时值、超时处理函数  @msecs - 超时值，单位ms |
| int uloop\_timeout\_cancel(struct uloop\_timeout \*timeout) | 关掉定时器  @timeout - 要关掉的定时器模块 |

代码示范(未处理返回值)：

void recv\_handler(struct uloop\_fd \*h,uint32\_t events)

{

/\* do your recv handler\*/

}

void timeout\_handler(struct uloop\_timeout \*timer)

{

/\* do your timeout handler\*/

}

int main(int argc,char \*\*argv)

{

struct uloop\_fd myfd;

struct uloop\_timeout mytimer;

uloop\_init();

myfd.fd = socket(PF\_PACKET,SOCK\_RAW,htons(ETH\_P\_PAE));

myfd.cb = recv\_handler;

uloop\_fd\_add(&myfd);

mytimer.cb = timeout\_handler;

uloop\_timeout\_set(&mytimer,100);

uloop\_run();

uloop\_fd\_delete(&myfd);

close(myfd.fd);

uloop\_timeout\_cancel(&mytimer);

uloop\_done();

return 0;

}

**1.3. 链表**

libubox的链表是完全参照内核链表的设计理念，跟普通链表的区别在于，它不是将数据结构塞入链表，而是将链表节点塞入数据结构。

以下是链表的数据结构：

struct list\_head {

struct list\_head \*next;

struct list\_head \*prev;

};

struct list\_head就是抽象出来的链表模块，使用方法就是将其插入到自定义的结构中，范例如下：

struct fox {

int len;

int weight;

struct list\_head list;

};

上述结构中，fox中的list.next指向下一个元素，list.prev指向前一个元素。

通常，对链表的管理还需要一个标准的索引指针指向整个链表，即链表的头指针，这个特殊的头指针事实上也就是一个常规的list\_head.

以下是链表操作API：

|  |  |
| --- | --- |
| LIST\_HEAD(name) | 创建并初始化链表头节点（前驱和后继都指向自己） |
| static inline void INIT\_LIST\_HEAD(struct list\_head \*list) | 初始化链表头节点 |
| static inline void list\_empty(const struct list\_head \*head) | 判断链表是否为空，空返回1，非空返回0 |
| static inline void list\_del(struct list\_head \*entry) | 删除指定节点 |
| static inline void list\_add(struct list\_head \*new,struct list\_head \*head) | 节点从头部插入链表 |
| static inline void list\_add\_tail(struct list\_head \*new,struct list\_head \*head) | 节点从尾部插入链表 |
| list\_entry(ptr,type,field) | 根据链表模块地址反推得到父结构地址  @ptr - 链表模块地址（比如上面的 “&fox.list”）  @type - 父结构类型（比如上面的”struct fox”）  @field - 父结构中链表模块的名字（比如上面的”list”） |
| list\_for\_each\_entry(p,h,field) | 遍历链表的父结构（不支持遍历中删除操作）  @p - 遍历出来的父结构指针  @h - 链表头节点指针  @field - 父结构中链表模块的名字 |
| list\_for\_each\_entry\_safe(p,n,h,field) | 遍历链表的父结构（可以支持遍历中删除操作）  @n - 备份指针 |

备注：操作链表(添加、删除、移动、合并)的时间复杂度为O(1)，意味着无论操作的链表大小以及参数如何，他们都是在恒定时间内完成的；

遍历链表的时间复杂度为O(n)，n是链表包含的节点数目，意味着遍历时间跟节点数量呈线性关系

**1.4. 数据传输单元**

libubox提供了一套通用的数据传输机制，数据类型可以是

enum blobmsg\_type {

BLOBMSG\_TYPE\_UNSPEC,

BLOBMSG\_TYPE\_ARRAY,

BLOBMSG\_TYPE\_TABLE,

BLOBMSG\_TYPE\_STRING,

BLOBMSG\_TYPE\_INT64,

BLOBMSG\_TYPE\_INT32,

BLOBMSG\_TYPE\_INT16,

BLOBMSG\_TYPE\_INT8,

};

其中array和table两种数据类型还支持嵌套使用。

struct blob\_buf表示一个完整的数据传输单元，这个数据结构是这套数据传输机制的核心，需要注意的是，blob\_buf使用完后一定要调用blob\_buf\_free销毁，特别是当定义成局部变量时!

以下是这套数据传输机制的常用API整理：

|  |  |
| --- | --- |
| int blob\_buf\_init(struct blob\_buf \*buf,int id) | blob\_buf初始化，也可以理解为复位  @buf - 数据传输单元  @id - 数据类型 |
| void blob\_buf\_free(struct blob\_buf \*buf) | 注销blob\_buf，实际就是释放blob\_buf使用过程中申请的空间 |
| static inline const char \*blobmsg\_name(const struct blob\_attr \*attr) | 获取消息的名字 |
| static inline int blobmsg\_type(const struct blob\_attr \*attr) | 获取消息的类型 |
| static inline int blobmsg\_len(const struct blob\_attr \*attr) | 获取消息值长度 |
| static inline uint8\_t blobmsg\_get\_u8(struct blob\_attr \*attr) | 获取uint8\_t类型的消息值 |
| static inline uint16\_t blobmsg\_get\_u16(struct blob\_attr \*attr) | 获取uint16\_t类型的消息值 |
| static inline uint32\_t blobmsg\_get\_u32(struct blob\_attr \*attr) | 获取uint32\_t类型的消息值 |
| static inline uint64\_t blobmsg\_get\_u64(struct blob\_attr \*attr) | 获取uint64\_t类型的消息值 |
| static inline char \*blobmsg\_get\_string(struct blob\_attr \*attr) | 获取字符串类型的消息值 |
| static inline int blobmsg\_add\_u8(struct blob\_buf \*buf,const char \*name,uint8\_t val) | 添加一条uint8\_t类型的消息  @buf - 数据传输单元  @name - 消息名  @val - uint8\_t类型的消息值 |
| static inline int blobmsg\_add\_u16(struct blob\_buf \*buf,const char \*name,uint16\_t val) | 添加一条uint16\_t类型的消息  @buf - 数据传输单元  @name - 消息名  @val - uint16\_t类型的消息值 |
| static inline int blobmsg\_add\_u32(struct blob\_buf \*buf,const char \*name,uint32\_t val) | 添加一条uint32\_t类型的消息  @buf - 数据传输单元  @name - 消息名  @val - uint32\_t类型的消息值 |
| static inline int blobmsg\_add\_u64(struct blob\_buf \*buf,const char \*name,uint64\_t val) | 添加一条uint64\_t类型的消息  @buf - 数据传输单元  @name - 消息名  @val - uint64\_t类型的消息值 |
| static inline int blobmsg\_add\_string(struct blob\_buf \*buf,const char \*name,const char \*string) | 添加一条字符串类型的消息  @buf - 数据传输单元  @name - 消息名  @val - 字符串类型的消息值 |
| static inline void \*blobmsg\_open\_array(struct blob\_buf \*buf,const char \*name) | 开启一个array类型的消息  @buf - 数据传输单元  @name - 消息名  返回值 - 指向这个开启的array，也就是下面用到的”cookie” |
| static inline void blobmsg\_close\_array(struct blob\_buf \*buf,void \*cookie) | 关闭一个array类型的消息  @buf - 数据传输单元  @cookie - 指向这个开启的array |
| static inline void \*blobmsg\_open\_table(struct blob\_buf \*buf,const char \*name) | 开启一个table类型的消息  @buf - 数据传输单元  @name - 消息名  返回值 - 指向这个开启的array，也就是下面用到的”cookie” |
| static inline void blobmsg\_close\_table(struct blob\_buf \*buf,void \*cookie) | 关闭一个table类型的消息  @buf - 数据传输单元  @cookie - 指向这个开启的table |

备注：完成一条array或table消息的创建需要成对调用开启、关闭两个函数。

**1.5. json**

暂略

**1.6. avl树**

暂略

**1.7. md5**

暂略

**1.8 socket**

目前只封装了unix-sock 和inet-sock，个人觉得比较鸡肋，暂不分析。

1. **libubus**

ubus是openwrt引入的一个消息总线，类似于桌面linux系统中的dbus，其设计理念也基本一致，就是提供系统级的IPC和RPC。

**2.1 基本原理**

整套ubus基于libubox库实现，通信的基础是unix-sock，ubus提供了一个后台服务器ubusd进行所有ubus消息的中转处理。

所以，对于开发者来说，关注点就全放在客户端。ubus将客户端分为2种角色：

服务提供者

服务消费者

备注：当然某个客户端既可以是一些服务的提供者，同时又可以是另一些服务的消费者

ubus对其上面承载的消息格式进行了定义：采用json消息格式。

ubus将服务抽象成为“对象”和“方法”，一个对象可以包含多个方法。“对象”必须先注册到ubus后台服务器，才能被消费者调用。

ubus支持以 “阅订-通知”的方式进行进程通信，即进程A提供阅订服务，其他进程可以选择阅订或退订该服务，进程A就可以向所有阅订者广播推送通知。

**2.1. C接口使用方法**

libubus就是其C接口库，以下是常用API整理：

|  |  |
| --- | --- |
| struct ubus\_context \*ubus\_connect(const char \*path) | 创建ubus客户端并发起连接  @path - ubus后台服务器socket地址，默认就是/var/run/ubus.sock  返回值 - 成功则返回一个ubus客户端控制块 |
| void ubus\_free(struct ubus\_context \*ctx) | 注销ubus客户端 |
| static inline void ubus\_add\_uloop(struct ubus\_context \*ctx) | ubus客户端注册到libubox库epoll监听池中 |
| int ubus\_add\_object(struct ubus\_context \*ctx,struct ubus\_object \*obj) | ubus服务提供者调用，用来注册一个“对象”  @ctx - ubus客户端控制块  @obj - 要注册的对象控制块 |
| int ubus\_lookup\_id(struct ubus\_context \*ctx,const char \*path,uint32\_t \*id) | ubus服务消费者调用，根据“对象”名查找对应的id号  @ctx - ubus客户端控制块  @path - 对象名  @id - 记录查到的对象id |
| int ubus\_register\_subscriber(struct ubus\_context \*ctx,struct ubus\_subscriber \*obj) | ubus客户端注册一个阅订模块，后续阅订服务用  @ctx - ubus客户端控制块  @obj - 要注册的阅订控制块 |
| int ubus\_invoke(struct ubus\_context \*ctx,uint32\_t obj const char \*method,struct blob\_attr \*msg,ubus\_data\_handler\_t cb,void \*priv,int timeout) | ubus服务消费者调用，用于调用一个指定服务（同步调用）  @ctx - ubus客户端控制块  @obj - 对象id  @method - 指定“对象”包含的某个“方法”名  @msg - 具体的调用消息  @cb - 收到返回消息的处理函数  @priv - 用户自定义项，cb函数中可以使用  @timeout - 本次调用的超时时间 |
| int ubus\_invoke\_async(struct ubus\_context \*ctx,uint32\_t obj const char \*method,struct blob\_attr \*msg,struct ubus\_request \*req) | ubus服务消费者调用，用于调用一个指定服务（异步调用）  @ctx - ubus客户端控制块  @obj - 对象id  @method - 指定“对象”包含的某个“方法”名  @msg - 具体的调用消息  @req - 用来记录本次异步调用的相关信息 |
| int ubus\_send\_reply(struct ubus\_context \*ctx,struct ubus\_request\_data \*req,struct blob\_attr \*msg) | ubus服务提供者调用，用于回复服务消费者的调用请求  @ctx - ubus客户端控制块  @req - 用来记录本次调用的相关信息  @msg - 具体的回复消息 |
| int ubus\_notify(struct ubus\_context \*ctx,struct ubus\_object \*obj,const char \*type,struct blob\_attr \*msg,int timeout) | ubus客户端向所有阅订者广播推送通知  @ctx - ubus客户端控制块  @obj - 自身对象控制块  @type - 通知类型名  @msg - 具体的通知消息  @timeout - 本次通知的超时值 |

**2.2 使用范例(不考虑异常情况)**

**2.2.1 作为服务提供者**

enum {

TEST\_HELLO,

\_MAX\_TEST,

};

static struct ubus\_context \*ctx;

static const struct blobmsg\_policy policy[] = {

[TEST\_HELLO]{.name = “test\_hello”, .type = BLOBMSG\_TYPE\_STRING},

};

static int test\_handler(struct ubus\_context \*ctx,struct ubus\_object \*obj,struct ubus\_request\_data \*req,const char \*method,struct blob\_attr \*msg)

{

/\* do your handler\*/

}

static struct ubus\_method methods[] = {

UBUS\_METHOD(“test\_hello”,test\_handler,policy),

};

static struct ubus\_object\_type object\_type = UBUS\_OBJECT\_TYPE(“test”,methods);

static struct ubus\_object object = {

.name = “test”,

.type = &object\_type,

.methods = methods,

.n\_methodds = ARRAY\_SIZE(methods),

};

int main(int argc,char \*\*argv)

{

uloop\_init();

ctx = ubus\_connect(NULL);

uloop\_add\_uloop(ctx);

ubus\_add\_object(ctx,&object);

uloop\_run();

ubus\_free(ctx);

uloop\_done();

return 0;

}

**2.2.2 作为服务消费者**

static struct blob\_buf b;

static void reply\_handler(struct ubus\_request \*req,int type,struct blob\_attr \*msg)

{

/\* do your handler\*/

}

static int call\_server(struct ubus\_context \*ctx)

{

int id;

ubus\_lookup\_id(ctx,”test”,&id);

blob\_buf\_init(&b,0);

blobmsg\_add\_string(&b,”name”,”value”);

int ret = -1;

return ubus\_invoke(ctx,id,”test\_hello”,b.head,reply\_handler,&ret,1000);

}

int main(int argc,char \*\*argv)

{

uloop\_init();

ctx = ubus\_connect(NULL);

uloop\_add\_uloop(ctx);

call\_server();

uloop\_run();

ubus\_free(ctx);

uloop\_done();

return 0;

}

**2.2.3 作为阅订者**

static struct ubus\_subscriber event;

static void remove\_handler(struct ubus\_context \*ctx,struct ubus\_subscriber \*s,uint32\_t id)

{

/\*do your handler\*/

}

static int event\_handler(struct ubus\_context \*ctx,struct ubus\_object \*obj,struct ubus\_request\_data \*req,const char \*method,struct blob\_attr \*msg)

{

/\*do your handler\*/

}

int main(int argc,char \*\*argv)

{

uloop\_init();

ctx = ubus\_connect(NULL);

uloop\_add\_uloop(ctx);

ubus\_register\_subscriber(ctx,&event);

event.remove\_cb = remove\_handler;

event.cb = event\_handler;

ubus\_lookup\_id(ctx,”test”,id);

ubus\_subscribe(ctx,&event,id);

ubus\_run();

ubus\_free(ctx);

uloop\_done();

return 0;

}

**2.2.4 作为通知方**

int main(int argc,char \*\*argv)

{

uloop\_init();

ctx = ubus\_connect(NULL);

uloop\_add\_uloop(ctx);

ubus\_add\_object(ctx,&object);

ubus\_notify(ctx,&object,”notify”,NULL,-1);

ubus\_run();

ubus\_free(ctx);

uloop\_done();

return 0;

}

**3.libuci**

暂略