tips for stream using

compared with socket programming

socket与stream：

使用stream与socket非常相似，stream对socket作一层封装，在内部调用socket提供的接口，对外提供新的api，下面从socket基本的客户端服务器通信流程引入stream的使用方法。

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

客户端要与服务器通信时，先需要建立socket ＝ 客户端和服务器建立通信双方同样会建立一个

连接，通信双方会各自生成一个socket数据结 ＝ 各自的stream结构体，此外服务器还会唯一

构。 ＝ 构造一个pstream。

| socket结构体 | stream结构体 |
| --- | --- |
| socket数据结构：（单指面向连接的tcp应用）  1.通信协议  2.本地地址  3.本地端口  4.远程地址  5.远程端口  每个socket结构体对象使用int类型的socket描述符来标识。 | 客户端：stream  服务器：pstream stream  两端的stream与每条通信链接关联，所以与socket一样包含其中的1、2、3、4、5。  stream中不再使用描述符，而是直接使用stream结构体来作为api调用的参数，其中api的实现主要是调用stream结构体中的stream\_class（这是一个结构体指针）来实现的，在创建的时候使用一个name字符，例：pstream－ptcp:1234  stream－tcp: 127.0.0.1:1234 |

stream结构体代码： pstream结构体代码：

struct stream {

const struct stream\_class \*class; //包含大量函数指针，作为对外的api的内部实现。

int state; //连接状态

int error;

ovs\_be32 remote\_ip; //远程地址

ovs\_be16 remote\_port; //远程端口

ovs\_be32 local\_ip; //本地地址

ovs\_be16 local\_port; //本地端口

char \*name; //连接类型

};

struct pstream {

const struct pstream\_class \*class;

//包含一些函数指针

char \*name; //连接类型

ovs\_be16 bound\_port; //本服务器绑定的端口号，用于服务器监听连接请求

};

以基本的socket通信和stream通信类比两者的api功能

服务器端：

| （1）socket通信流程－开启服务 | （1）stream通信流程－开启服务 |
| --- | --- |
| 1. **创建**socket   int socket( int domain,  int type,  int protocol);  返回值：socket描述符。  int domain: 使用的协议族(tcp/ip为PF\_INE)  int type: socket类型；流式／数据报式。  int protocol:协议，一般取0。  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝   1. **初始化**本机地址信息   ：协议族 *sockaddr\_in.sin\_family＝*  ：端口号 *sockaddr\_in.sin\_port＝*  ：ip地址 *sockaddr\_in.sin\_addr.s\_addr＝*  使用*sockaddr\_in接收以上三项信息，并作为下一步绑定的输入参数。*  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝   1. **绑定**上本机地址与端口号   int bind( int sockfd,  struct sockaddr \*my\_addr,  int addrlen);  使用第一步得到的socket描述符和第二步得到的sockaddr\_in作为参数实现绑定。  int addrlen：常设为sizeof(struct sockaddr)  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝   1. **监听**对应的socket   int listen( int sockfd，  int backlog);  返回值：出错返回－1，否则监听成功  int sockfd：第一步得到的socket描述符。  int backlog：监听队列允许的最大长度。 | 1.**打开**stream（**与**socket通信的1、2、3、4功能上相对应）  int pstream\_open(  const char \*name,  struct pstream \*\*pstreamp,  uint8\_t dscp)  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝  返回值：执行成功为0，否则执行失败。  char ＊name： 连接名，以TYPE:ARGS的格式，pstream对应的ARGS字段只有一个端口号（stream对应端口号和ip号两个）。  pstream \*\*pstreamp：传入的pstream结构体，会在函数体中被初始化。pstream结构体作用相当于socket中用于监听说建立的socket结构体，在socket中监听和连接的socket不作区分，在stream中使用pstream监听接受连接，使用stream关联每条连接。  uint8\_t dscp：dscp号。  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝  函数实现机制   * 初始化pstream结构体，调用*pstream\_lookup\_class()*通过传入的name为pstream中的pstream\_class指针赋值（通过name字段的前半部分）。相当于socket中的1和2。 * 调用pstream中的pstream\_class中的*pstream\_class->listen()*函数将pstream与对应端口（name字段的后半部分）绑定。相当于socket中的3与4。 |

pstream\_open函数

pstream\_lookup\_class()函数代码：

int

pstream\_open(const char \*name, struct pstream \*\*pstreamp, uint8\_t dscp)

{

const struct pstream\_class \*class;

struct pstream \*pstream;

char \*suffix\_copy;

int error;

/\*COVERAGE\_INC(pstream\_open);\*/

/\* Look up the class. \*/

error = pstream\_lookup\_class(name, &class); //传入name，在函数内部实现对class

//的初始化，函数内容见之后的代码

if (!class) {

goto error;

}

/\* Call class's "open" function. \*/

suffix\_copy = xstrdup(strchr(name, ':') + 1); //获取name字段的后半段，内容为端口号

error = class->listen(name, suffix\_copy, &pstream, dscp);

//listen函数实现将pstream与对应的端口

等信息绑定，以供监听客户端连接。

free(suffix\_copy);

if (error) {

goto error;

}

/\* Success. \*/

\*pstreamp = pstream; //实现为pstream初始化的功能。

return 0;

error:

\*pstreamp = NULL;

return error;

}

static int

pstream\_lookup\_class(const char \*name, const struct pstream\_class \*\*classp)

{ size\_t prefix\_len;

size\_t i;

\*classp = NULL;

prefix\_len = strcspn(name, ":"); //获取name字段中连接名字段的长度

if (name[prefix\_len] == '\0') { //该字段可取：tcp；ssl；unix

return EAFNOSUPPORT;}

for (i = 0; i < ARRAY\_SIZE(pstream\_classes); i++) {

const struct pstream\_class \*class = pstream\_classes[i];

//用pstream\_class数组初始化 pstream\_class指针

if (strlen(class->name) == prefix\_len

&& !memcmp(class->name, name, prefix\_len)) { //赋值给class的name参数

\*classp = class;

return 0; }}

return EAFNOSUPPORT;

}

| （2）socket通信流程－接受连接 | （2）stream通信流程－接受连接 |
| --- | --- |
| 1. **接受**连接请求   int accept( int sockfd,  void \*addr,  int \*addrlen);  返回值：不成功为－1，成功返回新建的socket描述符。  int sockfd：之前通信开启中建立的用于监听的socket对应的描述符。  int \*addrlen：常指向sizeof(struct sockaddr\_in)。  原理：  当accept函数监视的 socket收到连接请求时，socket执行体将建立一个新的socket，执行体将这个新socket和请求连接进程的地址联系起来，收到服务请求的 初始socket仍可以继续在以前的 socket上监听，同时可以在新的socket描述符上进行数据传输操作。 | 1. **接受**连接请求   int pstream\_accept  ( struct pstream \*pstream,  struct stream \*\*new\_stream)  返回值：成功为0，否则失败  pstream \*pstream：之前通信开启中已经开始监听的pstream结构体指针。  new\_stream：新建立的连接将关联到这个传入的结构体中，类型是stream结构体对象。  原理：  这是一个非阻塞函数，同socket几乎一样，监听到连接后用对应的数据结构保持连接，以供之后数据传输使用。具体实现调用拉pstream->class->accept()函数实现连接接受，和pstream->class-connect()函数建立连接。  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝  ps：stream提供了一个同功能的阻塞函数  int pstream\_accept\_block  ( struct pstream \*pstream,  struct stream \*\*new\_stream)  其中调用了pstream\_accept()，在该函数运行成功（即接受建立拉一条连接）或者出错之前一直阻塞。 |

pstream\_accpet函数代码：

int

pstream\_accept(struct pstream \*pstream, struct stream \*\*new\_stream)

{

int retval = (pstream->class->accept)(pstream, new\_stream);//调用accept函数指针实现

具体的连接接受，并保存

if (retval) { 在new\_stream中。

\*new\_stream = NULL; //连接不成功处理

} else { //连接成功处理

ovs\_assert((\*new\_stream)->state != SCS\_CONNECTING

|| (\*new\_stream)->class->connect); //调用connect函数指针建立连接

} //执行之后state会变为SCS\_CONNECTING

return retval;

}

stream\_connect函数

stream中将连接分为三个状态（stream结构体中的state字段）

：无连接 － SCS\_DISCONNECTED

：正在连接 － SCS\_CONNECTING

：已经连接成功 － SCS\_CONNECTED

stream\_connect函数只处理正在连接状态的stream，以完成连接。

代码：

int

stream\_connect(struct stream \*stream)

{

enum stream\_state last\_state;

do {

last\_state = stream->state;

switch (stream->state) {

case SCS\_CONNECTING: //正在连接

scs\_connecting(stream); //调用scs\_connecting()完成连接

break;

case SCS\_CONNECTED: //如果连接已经完成

return 0; //不作处理，返回成功

case SCS\_DISCONNECTED: //连接不存在

return stream->error; //返回出错

default:

NOT\_REACHED();

}

} while (stream->state != last\_state);

return EAGAIN;

}

作用：由accept函数将连接建立并置状态为 SCS\_CONNECTING

之后进行数据传输的之前先调用stream\_connect完成连接，以供数据传输。

| （3）socket通信流程－传输数据 | （3）stream通信流程－传输数据 |
| --- | --- |
| 1. 发送数据   int send( int sockfd,  const void \*msg,  int len,  int flags);  返回值：实际发送的字节数  int sockfd：用于数据传输的socket描述符，从上一步接受连接中得到。  void \*msg：指向要发送数据的指针  len：希望发送数据的长度  flags：一般为0  ps：一般需要将send函数的返回值（实际发送出去的数据字节数）与参数len作个比较，以判断是否出现了发送问题。  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝   1. 接收数据   int recv( int sockfd,  void \*buf,  int len,  unsigned int flags);  返回值：实际接受到的数据，错误返回－1  int sockfd：与数据传输连接关联的socket描述符。  buf：存放接受到的数据的缓冲区。  len：缓冲的长度 。  flags：一般为0。 | 1. 发送数据   int stream\_send  ( struct stream \*stream,  const void \*buffer,  size\_t n)  返回值：实际发送出去的字节数  stream \*stream：上一步接受建立连接中关联的stream结构体。  buffer：发送缓冲区  n：希望发送数据的长度  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝   1. 接受数据   int stream\_recv  ( struct stream \*stream,  void \*buffer,  size\_t n)  返回值：接收到的数据字节数  stream \*stream：关联连接的stream结构体指针  buffer：用于存放接受数据的缓冲区  n：缓冲长度  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝  原理：   * 调用stream\_connect()完成连接的建立 * 调用stream->class->recv/send()函数完成数据的传输。 |

stream\_send()函数代码：

int

stream\_send(struct stream \*stream, const void \*buffer, size\_t n)

{

int retval = stream\_connect(stream); //若连接未完成，先完成整个连接

return (retval ? -retval

: n == 0 ? 0

: (stream->class->send)(stream, buffer, n)); //调用stream->class->send执行发送操作

}

stream\_recv()函数代码：

int

stream\_recv(struct stream \*stream, void \*buffer, size\_t n)

{

int retval = stream\_connect(stream); //若连接未建立，先完成整个连接

printf("%d\n",retval);

return (retval ? -retval

: n == 0 ? 0

: (stream->class->recv)(stream, buffer, n)); //调用sream->class->recv执行接受操作

}

| （4）socket通信流程－关闭 | （4）stream通信流程－关闭 |
| --- | --- |
| 1. 关闭连接 2. 关闭服务器   close(sockfd);  sockfd：关闭连接传入与连接关联的socket描述符，关闭服务器传入服务器用于监听的socket描述符。 | 1. 关闭连接   void stream\_close  ( struct pstream \*pstream )   1. 关闭服务器   void pstream\_close  ( struct stream \*stream)  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝  原理：   * 释放分配的空间。 * 调用stream/pstream->class->close()完成关闭。 |

stream\_close()代码：

void

stream\_close(struct stream \*stream)

{

if (stream != NULL) {

char \*name = stream->name;

(stream->class->close)(stream); //调用stream->class->close完成关闭连接

free(name); //释放分配出的空间

}

}

pstream\_close()代码：

void

pstream\_close(struct pstream \*pstream)

{

if (pstream != NULL) {

char \*name = pstream->name;

(pstream->class->close)(pstream); //调用stream->class->close完成关闭连接

free(name); //释放分配出的空间

}

}

| socket通信流程－连接服务器 | stream通信流程－连接服务器 |
| --- | --- |
| 1. **创建**socket   int socket( int domain,  int type,  int protocol);  返回值：socket描述符。  int domain: 使用的协议族(tcp/ip为PF\_INE)  int type: socket类型；流式／数据报式。  int protocol:协议，一般取0。  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝   1. **初始化**本机地址信息   ：协议族 *sockaddr\_in.sin\_family＝*  ：端口号 *sockaddr\_in.sin\_port＝*  ：ip地址 *sockaddr\_in.sin\_addr.s\_addr＝*  使用*sockaddr\_in接收以上三项信息，并作为下一步建立连接的输入参数。*  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝   1. 建立连接   int connect( int sockfd,  struct sockaddr \*serv\_addr,  int addrlen);  返回值：出错为－1否则成功。  sockfd：第一步创建的socket对应的描述符。  serv\_addr：第二步中的结构体sockaddr\_in作为传入对象。  addrlen：常设为sizeof(struct sockaddr)  Connect函数启动和远端主机的直接连接。只需知道目的服务器的IP地址和端口，而客户通过哪个端口与服务器建立连接并不需要关心，socket执行体为你的程序自动选择一个未被占用的端口，并通知你的程序数据什么时候到达端口。  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝  客户端的第一步中没用绑定端口的操作。  服务器端的第一步没用建立连接的过程，服务器只是被动的接受连接。 | 1. **打开**stream（与socket通信中的1、2、3、4相对应）   int stream\_open( const char \*name,  struct stream \*\*streamp,  uint8\_t dscp)  返回值：执行成功为0，否则执行失败。  char ＊name 连接名，以TYPE:ARGS的格式，stream对应的ARGS字段有一个端口号和一个ip号。  （name一般为－tcp：ip地址：端口号）  stream \*streamp：用该指针指向新生成的连接。  dscp：dscp号。  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝  函数实现机制：   * 初始化stream结构体，调用*stream\_lookup\_class()*通过传入的name为pstream中的pstream\_class指针赋值（通过name字段的前半部分）。－－相当于socket中的1和2中初始化的第一个协议族。 * 调用pstream中的pstream\_class中的*stream\_class->open()*函数，将stream指向，由对应端口和ip地址（name字段的后半部分）建立起来的连接。－－相当于socket中的2中初始化端口和地址和第3步建立连接（并不没有完成连接）。   ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝  stream提供一个同功能的阻塞函数  int stream\_open\_block(int error, struct stream \*\*streamp)  使用方法为：  stream\_open\_block(  stream\_open("tcp:1.2.3.4:5", &stream), &stream);  即调用stream\_open函数开始不停尝试开启，若不成功则一直阻塞，直到连接开启成功或是出错。 |

**客户端：**

stream\_open()函数代码：

int

stream\_open(const char \*name, struct stream \*\*streamp, uint8\_t dscp)

{

const struct stream\_class \*class;

struct stream \*stream;

char \*suffix\_copy;

int error;

/\*COVERAGE\_INC(stream\_open);\*/

/\* Look up the class. \*/

error = stream\_lookup\_class(name, &class); //传入name 和streamp的class指针，通过name

if (!class) { //给class赋值，具体实现见下面函数体。

goto error;

}

/\* Call class's "open" function. \*/

suffix\_copy = xstrdup(strchr(name, ':') + 1); //获取name字段的后半段，

//包括所连接的IP号和端口号

error = class->open(name, suffix\_copy, &stream, dscp); //调用stream->class->open()

free(suffix\_copy); //使用由上面获取到的ip和端口，传入stream引用来接受建立的连接。

if (error) {

goto error;

}

/\* Success. \*/

\*streamp = stream;

return 0;

error:

\*streamp = NULL;

return error;

}

stream\_lookup\_class()函数代码：

static int

stream\_lookup\_class(const char \*name, const struct stream\_class \*\*classp)

{ size\_t prefix\_len;

size\_t i;

\*classp = NULL;

prefix\_len = strcspn(name, ":"); //获取name字段前部分的长度

if (name[prefix\_len] == '\0') {

return EAFNOSUPPORT;}

for (i = 0; i < ARRAY\_SIZE(stream\_classes); i++) { //使用stream\_classes[]给class赋值

const struct stream\_class \*class = stream\_classes[i];

if (strlen(class->name) == prefix\_len //给class->name字段赋值

&& !memcmp(class->name, name, prefix\_len)) {

\*classp = class;

return 0;}}

return EAFNOSUPPORT;}

| socket通信流程－完成连接 | stream通信流程－完成连接 |
| --- | --- |
| 在上一步中使用connect()函数，若执行成功已经完成了连接的建立。 | 1. *完成连接*   *实际上在上一步open中已经进行部分连接操作，一般需要调用stream\_connect()函数完成整个连接的建立。*  *stream\_connect()代码在p5*  *stream\_connect()中主要通过scs\_connecting()完成连接的建立。*  *static void scs\_connecting*  *( struct stream \*stream)*  *stream \*stream：用于接受建立连接的stream结构体指针。*  *＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝*  *实现机制：*   * *调用stream->class->connect()建立连接* * *判断上一步是否成功，并修改stream中的state（状态）字段。* |

scs\_connecting()代码

static void

scs\_connecting(struct stream \*stream)

{

int retval = (stream->class->connect)(stream); //调用class中的connect函数指针

ovs\_assert(retval != EINPROGRESS);

if (!retval) {

stream->state = SCS\_CONNECTED; //如果上一步的函数执行成功，改状态为已连接

} else if (retval != EAGAIN) { //若retval为EAGAIN，表示正在尝试连接。

stream->state = SCS\_DISCONNECTED; //连接未建立，改状态为未连接

stream->error = retval;

}

}

这部分服务器端和客户端没用使用上的区别，可以无差别的传输或是接受数据，两者都用相同结构的stream结构体对象来管理对应的连接，可以看到使用的函数名都是以stream开头（非pstream）。

| （3）socket通信流程－传输数据 | （3）stream通信流程－传输数据 |
| --- | --- |
| 1. 发送数据   int send( int sockfd,  const void \*msg,  int len,  int flags);  返回值：实际发送的字节数  int sockfd：用于数据传输的socket描述符，从上一步接受连接中得到。  void \*msg：指向要发送数据的指针  len：希望发送数据的长度  flags：一般为0  ps：一般需要将send函数的返回值（实际发送出去的数据字节数）与参数len作个比较，以判断是否出现了发送问题。  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝   1. 接收数据   int recv( int sockfd,  void \*buf,  int len,  unsigned int flags);  返回值：实际接受到的数据，错误返回－1  int sockfd：与数据传输连接关联的socket描述符。  buf：存放接受到的数据的缓冲区。  len：缓冲的长度 。  flags：一般为0。 | 1. 发送数据   int stream\_send  ( struct stream \*stream,  const void \*buffer,  size\_t n)  返回值：实际发送出去的字节数  stream \*stream：上一步接受建立连接中关联的stream结构体。  buffer：发送缓冲区  n：希望发送数据的长度  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝   1. 接受数据   int stream\_recv  ( struct stream \*stream,  void \*buffer,  size\_t n)  返回值：接收到的数据字节数  stream \*stream：关联连接的stream结构体指针  buffer：用于存放接受数据的缓冲区  n：缓冲长度  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝  原理：   * 调用stream\_connect()完成连接的建立 * 调用stream->class->recv/send()函数完成数据的传输。 |

stream\_send()与stream\_recv()函数的代码见p6~p7 。

| （4）socket通信流程－关闭 | （4）stream通信流程－关闭 |
| --- | --- |
| 1. 关闭连接 2. 关闭服务器   close(sockfd);  sockfd：关闭对象为－与连接关联的socket描述符。 | 1. 关闭连接   void stream\_close  ( struct pstream \*pstream )  客户端只有关闭连接这一项  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝  原理：   * 释放分配的空间。 * 调用stream/pstream->class->close()完成关闭。 |

stream\_close()代码见p7 。