

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连接，通信双方会各自生成一个socket数据结构。

= 客户端和服务端建立通信双方同样会建立一个
= 各自的stream结构体，此外服务器还会唯一
= 构造一个pstream。

socket结构体	stream结构体
<p>socket数据结构：（单指面向连接的tcp应用）</p> <ol style="list-style-type: none">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 <p>每个socket结构体对象使用int类型的socket描述符来标识。</p>	<p>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。</p> <p>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</p>

stream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
};
```

pstream结构体代码：

```
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
<p>1. 创建socket</p> <pre>int socket(int domain, int type, int protocol);</pre> <p>返回值: <i>socket</i>描述符。 <i>int domain</i>: 使用的协议族(<i>tcp/ip</i>为<i>PF_INET</i>) <i>int type</i>: <i>socket</i>类型; 流式/数据报式。 <i>int protocol</i>: 协议, 一般取0。</p> <p>=====</p> <p>2. 初始化本机地址信息</p> <pre> : 协议族 sockaddr_in.sin_family = : 端口号 sockaddr_in.sin_port = : ip地址 sockaddr_in.sin_addr.s_addr =</pre> <p>使用<i>sockaddr_in</i>接收以上三项信息, 并作为下一步绑定的输入参数。</p> <p>=====</p> <p>3. 绑定上本机地址与端口号</p> <pre>int bind(int sockfd, struct sockaddr *my_addr, int addrlen);</pre> <p>使用第一步得到的<i>socket</i>描述符和第二步得到的<i>sockaddr_in</i>作为参数实现绑定。 <i>int addrlen</i>: 常设为<i>sizeof(struct sockaddr)</i></p> <p>=====</p> <p>4. 监听对应的socket</p> <pre>int listen(int sockfd, int backlog);</pre> <p>返回值: 出错返回 - 1, 否则监听成功 <i>int sockfd</i>: 第一步得到的<i>socket</i>描述符。 <i>int backlog</i>: 监听队列允许的最大长度。</p>	<p>1. 打开stream (与socket通信的1、2、3、4功能上相对应)</p> <pre>int pstream_open(const char *name, struct pstream **pstream, uint8_t dscp)</pre> <p>=====</p> <p>返回值: 执行成功为0, 否则执行失败。 <i>char * name</i>: 连接名, 以<i>TYPE:ARGS</i>的格式, <i>pstream</i>对应的<i>ARGS</i>字段只有一个端口号 (<i>stream</i>对应端口号和<i>ip</i>号两个)。</p> <p><i>pstream **pstream</i>: 传入的<i>pstream</i>结构体, 会在函数体中被初始化。<i>pstream</i>结构体作用相当于<i>socket</i>中用于监听说建立的<i>socket</i>结构体, 在<i>socket</i>中监听和连接的<i>socket</i>不作区分, 在<i>stream</i>中使用<i>pstream</i>监听接受连接, 使用<i>stream</i>关联每条连接。</p> <p><i>uint8_t dscp</i>: <i>dscp</i>号。</p> <p>=====</p> <p>函数实现机制</p> <ul style="list-style-type: none"> 初始化<i>pstream</i>结构体, 调用 <i>pstream_lookup_class()</i>通过传入的<i>name</i>为<i>pstream</i>中的<i>pstream_class</i>指针赋值 (通过<i>name</i>字段的前半部分)。相当于<i>socket</i>中的1和2。 调用<i>pstream</i>中的<i>pstream_class</i>中的 <i>pstream_class->listen()</i>函数将<i>pstream</i>与对应端口 (<i>name</i>字段的后半部分) 绑定。相当于<i>socket</i>中的3与4。

pstream_open函数

```
int
pstream_open(const char *name, struct pstream **pstream, 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. */
    *pstream = pstream; //实现为pstream初始化的功能。
    return 0;

error:
    *pstream = NULL;
    return error;
}
```

pstream_lookup_class()函数代码：

```
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
<p>1. 接受连接请求</p> <pre>int accept(int sockfd, void *addr, int *addrlen);</pre> <p>返回值：不成功为 - 1，成功返回新建的 <i>socket</i> 描述符。 <i>int sockfd</i>: 之前通信开启中建立的用于监听的 <i>socket</i> 对应的描述符。 <i>int *addrlen</i>: 常指向 <i>sizeof(struct sockaddr_in)</i>。</p> <p>原理： 当 <i>accept</i> 函数监视的 <i>socket</i> 收到连接请求时，<i>socket</i> 执行体将建立一个新的 <i>socket</i>，执行体将这个新 <i>socket</i> 和请求连接进程的地址联系起来，收到服务请求的 初始 <i>socket</i> 仍可以继续以前的 <i>socket</i> 上监听，同时可以在新的 <i>socket</i> 描述符上进行数据传输操作。</p>	<p>1. 接受连接请求</p> <pre>int pstream_accept (struct pstream *pstream, struct stream **new_stream)</pre> <p>返回值：成功为 0，否则失败 <i>pstream *pstream</i>: 之前通信开启中已经开始监听的 <i>pstream</i> 结构体指针。 <i>new_stream</i>: 新建的连接将关联到这个传入的结构体中，类型是 <i>stream</i> 结构体对象。</p> <p>原理： 这是一个非阻塞函数，同 <i>socket</i> 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 <i>pstream->class->accept()</i> 函数实现连接接受，和 <i>pstream->class->connect()</i> 函数建立连接。</p> <p>=====</p> <p>ps: stream 提供了一个同功能的阻塞函数</p> <pre>int pstream_accept_block (struct pstream *pstream, struct stream **new_stream)</pre> <p>其中调用了 <i>pstream_accept()</i>，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。</p>

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
                                                                在new_stream中。
    if (retval) {
        *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
    }
}
```

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
<p>1. 发送数据</p> <pre>int send(int sockfd, const void *msg, int len, int flags);</pre> <p>返回值：实际发送的字节数 <i>int sockfd</i>: 用于数据传输的<i>socket</i>描述符，从上一步接受连接中得到。 <i>void *msg</i>: 指向要发送数据的指针 <i>len</i>: 希望发送数据的长度 <i>flags</i>: 一般为0</p> <p>ps: 一般需要将<i>send</i>函数的返回值（实际发送出去的数据字节数）与参数<i>len</i>作个比较，以判断是否出现了发送问题。</p> <p>=====</p> <p>2. 接收数据</p> <pre>int recv(int sockfd, void *buf, int len, unsigned int flags);</pre> <p>返回值：实际接受到的数据，错误返回 - 1 <i>int sockfd</i>: 与数据传输连接关联的<i>socket</i>描述符。 <i>buf</i>: 存放接受到的数据的缓冲区。 <i>len</i>: 缓冲的长度。 <i>flags</i>: 一般为0。</p>	<p>1. 发送数据</p> <pre>int stream_send (struct stream *stream, const void *buffer, size_t n)</pre> <p>返回值：实际发送出去的字节数 <i>stream *stream</i>: 上一步接受建立连接中关联的<i>stream</i>结构体。 <i>buffer</i>: 发送缓冲区 <i>n</i>: 希望发送数据的长度</p> <p>=====</p> <p>2. 接受数据</p> <pre>int stream_recv (struct stream *stream, void *buffer, size_t n)</pre> <p>返回值：接收到的数据字节数 <i>stream *stream</i>: 关联连接的<i>stream</i>结构体指针 <i>buffer</i>: 用于存放接受数据的缓冲区 <i>n</i>: 缓冲长度</p> <p>=====</p> <p>原理：</p> <ul style="list-style-type: none"> 调用<i>stream_connect()</i>完成连接的建立 调用<i>stream->class->recv/send()</i>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stream->class->recv执行接受操作
}
```

(4) socket通信流程 – 关闭	(4) stream通信流程 – 关闭
<ol style="list-style-type: none">1. 关闭连接2. 关闭服务器 <p><i>close(sockfd);</i></p> <p><i>sockfd</i>: 关闭连接传入与连接关联的<i>socket</i>描述符，关闭服务器传入服务器用于监听的<i>socket</i>描述符。</p>	<ol style="list-style-type: none">1. 关闭连接 <i>void stream_close</i> (<i> struct pstream *pstream</i>)2. 关闭服务器 <i>void pstream_close</i> (<i> struct stream *stream</i>) <p>=====</p> <p>原理：</p> <ul style="list-style-type: none">• 释放分配的空间。• 调用<i>stream/pstream->class->close()</i>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
    }
}
```


客户端：

socket通信流程 – 连接服务器	stream通信流程 – 连接服务器
<p>1. 创建socket</p> <pre>int socket(int domain, int type, int protocol);</pre> <p>返回值：socket描述符。 <i>int domain</i>: 使用的协议族(<i>tcp/ip</i>为<i>PF_INET</i>) <i>int type</i>: socket类型；流式／数据报式。 <i>int protocol</i>: 协议，一般取0。</p> <p>=====</p> <p>2. 初始化本机地址信息</p> <pre>: 协议族 sockaddr_in.sin_family = : 端口号 sockaddr_in.sin_port = : ip地址 sockaddr_in.sin_addr.s_addr =</pre> <p>使用<i>sockaddr_in</i>接收以上三项信息，并作为下一步建立连接的输入参数。</p> <p>=====</p> <p>3. 建立连接</p> <pre>int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);</pre> <p>返回值：出错为-1否则成功。 <i>sockfd</i>: 第一步创建的socket对应的描述符。 <i>serv_addr</i>: 第二步中的结构体<i>sockaddr_in</i>作为传入对象。 <i>addrlen</i>: 常设为sizeof(struct sockaddr)</p> <p>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。</p> <p>=====</p> <p>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。</p>	<p>1. 打开stream（与socket通信中的1、2、3、4相对应）</p> <pre>int stream_open(const char *name, struct stream **streamp, uint8_t dscp)</pre> <p>返回值：执行成功为0，否则执行失败。 <i>char * name</i> 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。 （<i>name</i>一般为-tcp: ip地址: 端口号） <i>stream *streamp</i>: 用该指针指向新生成的连接。 <i>dscp</i>: dscp号。</p> <p>=====</p> <p>函数实现机制：</p> <ul style="list-style-type: none"> 初始化stream结构体，调用<i>stream_lookup_class()</i>通过传入的<i>name</i>为<i>pstream</i>中的<i>pstream_class</i>指针赋值（通过<i>name</i>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<i>pstream</i>中的<i>pstream_class</i>中的<i>stream_class->open()</i>函数，将stream指向，由对应端口和ip地址（<i>name</i>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）。 <p>=====</p> <p>stream提供一个同功能的阻塞函数</p> <pre>int stream_open_block(int error, struct stream **streamp)</pre> <p>使用方法为：</p> <pre>stream_open_block(stream_open("tcp:1.2.3.4:5", &stream), &stream);</pre> <p>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。</p>

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
<p>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。</p>	<p>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</p> <p>stream_connect()中主要通过scs_connecting()完成连接的建立。</p> <p>static void scs_connecting (struct stream *stream) stream *stream: 用于接受建立连接的stream结构体指针。</p> <p>=====</p> <p>实现机制：</p> <ul style="list-style-type: none"> • 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
<p>1. 发送数据</p> <pre>int send(int sockfd, const void *msg, int len, int flags);</pre> <p>返回值：实际发送的字节数 <i>int sockfd</i>: 用于数据传输的socket描述符，从上一步接受连接中得到。 <i>void *msg</i>: 指向要发送数据的指针 <i>len</i>: 希望发送数据的长度 <i>flags</i>: 一般为0</p> <p>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。</p> <p>=====</p> <p>2. 接收数据</p> <pre>int recv(int sockfd, void *buf, int len, unsigned int flags);</pre> <p>返回值：实际接受到的数据，错误返回 - 1 <i>int sockfd</i>: 与数据传输连接关联的socket描述符。 <i>buf</i>: 存放接受到的数据的缓冲区。 <i>len</i>: 缓冲的长度。 <i>flags</i>: 一般为0。</p>	<p>1. 发送数据</p> <pre>int stream_send (struct stream *stream, const void *buffer, size_t n)</pre> <p>返回值：实际发送出去的字节数 <i>stream *stream</i>: 上一步接受建立连接中关联的stream结构体。 <i>buffer</i>: 发送缓冲区 <i>n</i>: 希望发送数据的长度</p> <p>=====</p> <p>2. 接受数据</p> <pre>int stream_recv (struct stream *stream, void *buffer, size_t n)</pre> <p>返回值：接收到的数据字节数 <i>stream *stream</i>: 关联连接的stream结构体指针 <i>buffer</i>: 用于存放接受数据的缓冲区 <i>n</i>: 缓冲长度</p> <p>=====</p> <p>原理：</p> <ul style="list-style-type: none"> • 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
<div>1. 关闭连接</div> <div>2. 关闭服务器</div> <div><code>close(sockfd);</code></div> <div><code>sockfd</code>: 关闭对象为 – 与连接关联的<code>socket</code>描述符。</div>	<div>1. 关闭连接</div> <div><code>void stream_close</code> <code>(struct pstream *pstream)</code></div> <div>客户端只有关闭连接这一项</div> <div>=====</div> <div>原理:</div> <div><ul style="list-style-type: none">• 释放分配的空间。• 调用<code>stream/pstream->class->close()</code>完成关闭。</div>

`stream_close()`代码见p7。