Erlang Socket

a．最核心的概念 － socket 控制进程

b．基本的 C/S 结构的例子（服务器只能处理一个客户端连接）

c．顺序型服务器的例子（服务器顺序的处理客户端的请求，每次只能处理一个，处理完一个处理下一个）

d．并发型服务器的例子（服务器并发的处理多个客户端的请求）

e．控制逻辑 － 主动型消息接收（非阻塞）

f．控制逻辑 － 被动型消息接收（阻塞）

g．控制逻辑 － 混合型消息接收（半阻塞）

1．最基本的 Erlang C/S 结构的例子:

<1> 创建一个 socket 的进程（调用 gen_tcp:accept 或 gen_tcp:connect）也就是 socket 的控制进程，这个 socket 接收到的所有数据都转发给控制进程，如果控制进程消亡，socket 也会自行关闭，可以调用 gen_tcp:controlling_process(Socket, NewPid)来把一个 socket 的控制进程改为新的进程。

<2> 服务器端和客户端使用的{packet, N}的参数必须一致

<3> 在接收到一个连接的时候，显示的设置 socket 的属性，是一个好的策略

```
{ok, Socket} = gen_tcp:accept(Listen),
inet:setopts(Socket, [{packet, 4}, {active, true}, {nodelay, true}])
```

```
-module(server).
-export([start/0]).

start() ->
  {ok, Listen} = gen_tcp:listen(2345, [binary, {packet, 4},
                                       {reuseaddr, true},
                                       {active, true}]),
  {ok, Socket} = gen_tcp:accept(Listen),
  gen_tcp:close(Listen),
  loop(Socket).

loop(Socket) ->
  receive
    {tcp, Socket, Bin} ->
        io:format("received: ~p~n", [Bin]),
        gen_tcp:send(Socket, iolist_to_binary(["server#",Bin])),
        loop(Socket);
    {tcp_closed, Socket} ->
```

```
        io:format("[~p] tcp_closed~n", [Socket]);
    {tcp_error, Socket, Reason} ->
        io:format("[~p] tcp_error: ~p~n", [Socket, Reason])
  end.


-module(client).
-export([echo/1]).

echo(Data) ->
  {ok, Socket} = gen_tcp:connect("localhost", 2345, [binary, {packet, 4}]),
  ok = gen_tcp:send(Socket, Data),
  receive
    {tcp, Socket, Bin} ->
        io:format("~p~n", [Bin]),
        gen_tcp:close(Socket)
  end.
```

2．顺序型服务器的例子

```
start() ->
  {ok, Listen} = gen_tcp:listen(2345, [binary, {packet, 4},
                                       {reuseaddr, true},
                                       {active, true}]),
  seq_accept(Listen).

seq_accept(Listen) ->
  {ok, Socket} = gen_tcp:accept(Listen),
  loop(Socket),
  seq_accept(Listen).

loop(Socket) … 不变
```

3．并发型服务器的例子

```
start() ->
  {ok, Listen} = gen_tcp:listen(2345, [binary, {packet, 4},
                    {reuseaddr, true},
                    {active, true}]),
  spawn(fun() -> accept(Listen) end).

accept(Listen) ->
  {ok, Socket} = gen_tcp:accept(Listen),
  spawn(fun() -> accept(Listen) end),
  loop(Socket).

loop(Socket) … 不变
```

4．控制逻辑

<1> {active, true} — 主动 socket — 非阻塞模式

当数据到达系统之后，会向控制进程发送{tcp, Socket, Data}的消息，而控制进程无法控制这些消息，一个独立的客户端可能向系统发送上万条消息，这些消息都会发送到控制进程，控制进程无法控制挺掉这些消息。

<2> {active, false} — 被动 socket — 阻塞模式

如果是被动 socket，则 socket 必须调用 gen_tcp:recv(Socket, N)来接收数据，它尝试接收 N 字节的数据，如果 N 为 0，那么所有可用的字节都会返回。

默认是: gen_tcp:(Socket, N, infinity)，也即使无限等待直到有数据可以接收，所以是阻塞的模式。

<3> {active, once} — 半主动 socket

会创建一个主动 socket,可以接收一条消息，但这个 socket 所接收一条消息以后，如果让它接收下一条消息，必须激活它。

inet:setopts(Socket, [{active, once}]) — 激活 Socket 的方式

5．主动型接收 — 非阻塞模式— 异步服务器

```
{ok, Listen} = gen_tcp:listen( … {active, true}  … ),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
  receive
    {tcp, Socket, Data} ->
      ...;
    {tcp_closed, Socket} ->
       ...
  end.
```

6．被动型接收 — 阻塞模式 - 同步服务器

```
{ok, Listen} = gen_tcp:listen( … {active, false}  … ),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
  case gen_tcp:recv(Socket, 0) of
    {ok, Data} ->
       …
       loop(Socket);
    {error, closed} ->
       ...
  end.
```

7．混合型模式 — 半同步服务器

```
{ok, Listen} = gen_tcp:listen( … {active, once}  … ),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
  receive
    {tcp, Socket, Data} ->
      …
      inet:setopts(Socket, [{active, once}]),
      loop(Socket);
    {tcp_closed, Socket} ->
      ...
  end.
```

8．socket 的出错处理：

<1> 每个 socket 都对应一个控制进程，如果控制进程消亡，则 socket 也会自动关闭

<2> 如果服务器端应为逻辑崩溃，那么服务器端的 socket 会自动关闭，同时客户端也会收到 {tcp_closed, Socket}的消息