



心跳包机制设计详解

 范蠡 发表于 高性能服务器开发

168

分享

TCP keepalive 选项

应用层的心跳包机制设计

带业务数据的心跳包

心跳包与流量

心跳包与调试

心跳包与日志

存在下面两种情形：

情形一：一个客户端连接服务器以后，如果长期没有和服务器有数据来往，可能会被防火墙程序关闭连接，有时候我们并不想要被关闭连接。例如，对于一个即时通讯软件，如果服务器没有消息时，我们确实不会和服务器有任何数据交换，但是如果连接被关闭了，有新消息来时，我们再也无法收到了，这就违背了“即时通讯”的设计要求。

情形二：通常情况下，服务器与某个客户端一般不是位于同一个网络，其之间可能经过数个路由器和交换机，如果其中某个必经路由器或者交换器出现了故障，并且一段时间内没有恢复，导致这之间的链路不再畅通，而此时服务器与客户端之间也没有数据进行交换，由于 TCP 连接是状态机，对于这种情况，无论是客户端或者服务器都无法感知与对方的连接是否正常，这类连接我们一般称之为“死链”。

情形一中的应用场景要求必须保持客户端与服务器之间的连接正常，就是我们通常所说的“保活”。如上文所述，当服务器与客户端一定时间内没有有效业务数据来往时，我们只需要给对端发送心跳包即可实现保活。

情形二中的死链，只要我们此时任意一端给对端发送一个数据包即可检测链路是否正常，这类数据包我们也称之为“心跳包”，这种操作我们称之为“心跳检测”。顾名思义，如果一个人没有心跳了，可能已经死亡了；一个连接长时间没有正常数据来往，也没有心跳包来往，就可以认为这个连接已经不存在，为了节约服务器连接资源，我们可以通过关闭 socket，回收连接资源。

根据上面的分析，让我再强调一下，心跳检测一般有两个作用：

- 保活
- 检测死链

TCP keepalive 选项

操作系统的 TCP/IP 协议栈其实提供了这个的功能，即 keepalive 选项。在 Linux 操作系统中，我们可以通过代码启用一个 socket 的心跳检测（即每隔一定时间间隔发送一个心跳检测包给对端），代码如下：

```
//on 是 1 表示打开 keepalive 选项，为 0 表示关闭，0 是默认值
int on = 1;
setsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, &on, sizeof(on));
```

但是，即使开启了这个选项，这个选项默认发送心跳检测数据包的时间间隔是 7200 秒（2 小时），这时间间隔实在是太长了，一定也不使用。

我们可以通过继续设置 `keepalive` 相关的三个选项来改变这个时间间隔，它们分别是 `TCP_KEEPIDLE`、`TCP_KEEPINTVL` 和 `TCP_KEEPCNT`，示例代码如下：

```
//发送 keepalive 报文的时间间隔
int val = 7200;
setsockopt(fd, IPPROTO_TCP, TCP_KEEPIDLE, &val, sizeof(val));

//两次重试报文的时间间隔
int interval = 75;
setsockopt(fd, IPPROTO_TCP, TCP_KEEPINTVL, &interval, sizeof(interval));

int cnt = 9;
setsockopt(fd, IPPROTO_TCP, TCP_KEEPCNT, &cnt, sizeof(cnt));
```

`TCP_KEEPIDLE` 选项设置了发送 `keepalive` 报文的时间间隔，发送时如果对端回复 `ACK`。则本端 `TCP` 协议栈认为该连接依然存活，继续等 7200 秒后再发送 `keepalive` 报文；如果对端回复 `RESET`，说明对端进程已经重启，本端的应用程序应该关闭该连接。

如果对端没有任何回复，则本端做重试，如果重试 9 次（`TCP_KEEPCNT` 值）（前后重试间隔为 75 秒（`TCP_KEEPINTVL` 值））仍然不可达，则向应用程序返回 `ETIMEOUT`（无任何应答）或 `EHOST` 错误信息。

我们可以使用如下命令查看 `Linux` 系统上的上述三个值的设置情况：

```
[root@iZ238vnojlyZ ~]# sysctl -a | grep keepalive
net.ipv4.tcp_keepalive_intvl = 75
net.ipv4.tcp_keepalive_probes = 9
net.ipv4.tcp_keepalive_time = 7200
```

在 `Windows` 系统设置 `keepalive` 及对应选项的代码略有不同：

```
//开启 keepalive 选项
const char on = 1;
setsockopt(socket, SOL_SOCKET, SO_KEEPALIVE, (char *)&on, sizeof(on));

// 设置超时详细信息
DWORD cbBytesReturned;
tcp_keepalive klive;
// 启用保活
klive.onoff = 1;
klive.keepalivetime = 7200;
// 重试间隔为10秒
klive.keepaliveinterval = 1000 * 10;
WSAIoctl(socket, SIO_KEEPALIVE_VALS, &klive, sizeof(tcp_keepalive), NULL, 0,
&cbBytesReturned, NULL, NULL);
```

应用层的心跳包机制设计

由于 `keepalive` 选项需要为每个连接中的 `socket` 开启，这不一定是必须的，可能会产生大量无意义的带宽浪费，且 `keepalive` 选项不能与应用层很好地交互，因此一般实际的服务开发中，还是建议读者在应用层设计自己的心跳包机制。那么如何设计呢？

从技术来讲，心跳包其实就是一个预先规定好格式的数据包，在程序中启动一个定时器，定时发送即可，这是最简单的实现思路。但是，如果通信的两端有频繁的数据来往，此时到了下一个发心跳包的时间点了，此时发送一个心跳包。这其实是一个流量的浪费，既然通信双方不断有正常的业务数据包来往，这些数据包本身就可以起到保活作用，为什么还要浪费流量去发送这些心跳包呢？所以，对于用于保活的心跳包，我们最佳做法是，设置一个上次包时间，每次收数据和发数据时，都更新一下这个包时间，而心跳检测计时器每次检测时，将这个包时间与当前系统时间做一个对比，如果时间间隔大于允许的最大时间间隔（实际开发中根据需求设置成 15 ~ 45 秒不等），则发送一次心跳包。总而言之，就是在与对端之间，没有数据来往达到一定时间间隔时才发送一次心跳包。

发心跳包的伪码：

```
bool CIUSocket::Send()
{
    int nSentBytes = 0;
    int nRet = 0;
    while (true)
    {
        nRet = ::send(m_hSocket, m_strSendBuf.c_str(), m_strSendBuf.length(), 0);
        if (nRet == SOCKET_ERROR)
        {
            if (::WSAGetLastError() == WSAEWOULDBLOCK)
                break;
            else
            {
                LOG_ERROR("Send data error, disconnect server:%s, port:%d.",
                    m_strServer.c_str(), m_nPort);
                Close();
                return false;
            }
        }
        else if (nRet < 1)
        {
            //一旦出现错误就立刻关闭Socket
            LOG_ERROR("Send data error, disconnect server:%s, port:%d.",
                m_strServer.c_str(), m_nPort);
            Close();
            return false;
        }

        m_strSendBuf.erase(0, nRet);
        if (m_strSendBuf.empty())
            break;

        ::Sleep(1);
    }

    {
        //记录一下最近一次发包时间
        std::lock_guard<std::mutex> guard(m_mutexLastDataTime);
        m_nLastDataTime = (long)time(NULL);
    }

    return true;
}

bool CIUSocket::Recv()
{
    int nRet = 0;
    char buff[10 * 1024];
    while (true)
    {
```

```

nRet = ::recv(m_hSocket, buff, 10 * 1024, 0);
if (nRet == SOCKET_ERROR) //一旦出现错误就立刻关闭Socket
{
    if (::WSAGetLastError() == WSAEWOULDBLOCK)
        break;
    else
    {
        LOG_ERROR("Recv data error, errorNO=%d.", ::WSAGetLastError());
        //Close();
        return false;
    }
}
else if (nRet < 1)
{
    LOG_ERROR("Recv data error, errorNO=%d.", ::WSAGetLastError());
    //Close();
    return false;
}

m_strRecvBuf.append(buff, nRet);

::Sleep(1);
}

{
    std::lock_guard<std::mutex> guard(m_mutexLastDataTime);
    //记录一下最近一次收包时间
    m_nLastDataTime = (long)time(NULL);
}

return true;
}

void CIUSocket::RecvThreadProc()
{
    LOG_INFO("Recv data thread start...");

    int nRet;
    //上网方式
    DWORD dwFlags;
    BOOL bAlive;
    while (!m_bStop)
    {
        //检测到数据则收数据
        nRet = CheckReceivedData();
        //出错
        if (nRet == -1)
        {
            m_pRecvMsgThread->NotifyNetError();
        }
        //无数据
        else if (nRet == 0)
        {
            long nLastDataTime = 0;
            {
                std::lock_guard<std::mutex> guard(m_mutexLastDataTime);
                nLastDataTime = m_nLastDataTime;
            }

            if (m_nHeartbeatInterval > 0)
            {
                //当前系统时间与上一次收发数据包的时间间隔超过了m_nHeartbeatInterval
                //则发一次心跳包
                if ((time(NULL) - nLastDataTime) >= m_nHeartbeatInterval)
                    SendHeartbeatPackage();
            }
        }
        //有数据
    }
}

```

```

        else if (nRet == 1)
        {
            if (!Recv())
            {
                m_pRecvMsgThread->NotifyNetError();
                continue;
            }

            DecodePackages();
        } // end if
    } // end while-loop

    LOG_INFO("Recv data thread finish...");
}

```

同理，检测心跳包的一端，应该是在与对端没有数据来往达到一定时间间隔时才做一次心跳检测。

心跳检测一端的伪码如下：

```

void BusinessSession::send(const char* pData, int dataLength)
{
    bool sent = TcpSession::send(pData, dataLength);

    //发送完数据更新下发包时间
    updateHeartbeatTime();
}

void BusinessSession::handlePackge(char* pMsg, int msgLength, bool& closeSession,
std::vector<std::string>& vectorResponse)
{
    //对数据合法性进行校验
    if (pMsg == NULL || pMsg[0] == 0 || msgLength <= 0 || msgLength > MAX_DATA_LENGTH)
    {
        //非法探测请求，不做任何应答，直接关闭连接
        closeSession = true;
        return;
    }

    //更新下收包时间
    updateHeartbeatTime();

    //省略包处理代码...
}

void BusinessSession::updateHeartbeatTime()
{
    std::lock_guard<std::mutex> scoped_guard(m_mutexForlastPackageTime);
    m_lastPackageTime = (int64_t)time(nullptr);
}

bool BusinessSession::doHeartbeatCheck()
{
    const Config& cfg = Singleton<Config>::Instance();
    int64_t now = (int64_t)time(nullptr);

    std::lock_guard<std::mutex> lock_guard(m_mutexForlastPackageTime);
    if (now - m_lastPackageTime >= cfg.m_MaxClientDataInterval)
    {
        //心跳包检测，超时，关闭连接
        LOGE("heartbeat expired, close session");
        shutdown();
        return true;
    }

    return false;
}

```

```

void TcpServer::checkSessionHeartbeat()
{
    int64_t now = (int64_t)time(nullptr);
    if (now - m_nLastCheckHeartbeatTime >= m_nHeartbeatCheckInterval)
    {
        m_spSessionManager->checkSessionHeartbeat();
        m_nLastCheckHeartbeatTime = (int64_t)time(nullptr);
    }
}

void SessionManager::checkSessionHeartbeat()
{
    std::lock_guard<std::mutex> scoped_lock(m_mutexForSession);
    for (const auto& iter : m_mapSessions)
    {
        //这里调用 BusinessSession::doHeartbeatCheck()
        iter.second->doHeartbeatCheck();
    }
}

```

需要注意的是：一般是客户端主动给服务器端发送心跳包，服务器端做心跳检测决定是否断开连接。而不是反过来，从客户端的角度来说，客户端为了让自己得到服务器端的正常服务有必要主动和服务端保持连接状态正常，而服务器端不会局限于某个特定的客户端，如果客户端不能主动和其保持连接，那么就会主动回收与该客户端的连接。当然，服务器端在收到客户端的心跳包时应该给客户端一个心跳应答。

带业务数据的心跳包

上面介绍的心跳包是从纯技术的角度来说的，在实际应用中，有时候我们需要定时或者不定时从服务器端更新一些数据，我们可以把这类数据放在心跳包中，定时或者不定时更新。

这类带业务数据的心跳包，就不再是纯粹技术上的作用了（这里说的技术的作用指的上文中介绍的心跳包起保活和检测死链作用）。

这类心跳包实现也很容易，即在心跳包数据结构里面加上需要的业务字段信息，然后在定时器中定时发送，客户端发给服务器，服务器在应答心跳包中填上约定的业务数据信息即可。

心跳包与流量

通常情况下，多数应用场景下，与服务器端保持连接的多个客户端中，同一时间段活跃用户（这里指的是与服务器有频繁数据来往的客户端）一般不会太多。当连接数较多时，进出服务器程序的数据包通常都是心跳包（为了保活）。所以为了减轻网络代码压力，节省流量，尤其是针对一些 3/4 G 手机应用，我们在设计心跳包数据格式时应该尽量减小心跳包的数据大小。

心跳包与调试

如前文所述，对于心跳包，服务器端的逻辑一般是在一定时间间隔内没有收到客户端心跳包时会主动断开连接。在我们开发调试程序过程中，我们可能需要将程序通过断点中断下来，这个过程可能是几秒到几十秒不等。等程序恢复执行时，连接可能因为心跳检测逻辑已经被断开。

调试过程中，我们更多的关注的是业务数据处理的逻辑是否正确，不想被一堆无意义的心跳包数据干扰实践。

鉴于以上两点原因，我们一般在调试模式下关闭或者禁用心跳包检测机制。代码大致如下：

```

ChatSession::ChatSession(const std::shared_ptr<TcpConnection>& conn, int sessionid) :
    TcpSession(conn),
    m_id(sessionid),

```

```
m_seq(0),
m_isLogin(false)
{
    m_userinfo.userid = 0;
    m_lastPackageTime = time(NULL);

    //这里设置了非调试模式下才开启心跳包检测功能
#ifdef _DEBUG
    EnableHearbeatCheck();
#endif
}
```

当然，你也可以将开启心跳检测的开关做成配置信息放入程序配置文件中。

心跳包与日志

实际生产环境，我们一般会将程序收到的和发出去的数据包写入日志中，但是无业务信息的心跳包信息是个例外，一般会刻意不写入日志，这是因为心跳包数据一般比较多，如果写入日志会导致日志文件变得很大，且充斥大量无意义的心跳包日志，所以一般在写日志时会屏蔽心跳包信息写入。

我这里的建议是，可以将心跳包信息是否写入日志做成一个配置开关，一般处于关闭状态，有需要时再开启。例如，对于一个 WebSocket 服务，ping 和 pong 是心跳包数据，下面示例代码按需输出心跳日志信息：

```
void BusinessSession::send(std::string_view strResponse)
{
    bool success = WebSocketSession::send(strResponse);

    if (success)
    {
        bool enablePingPongLog = Singleton<Config>::Instance().m_bPingPongLogEnabled;

        //其他消息正常打印，心跳消息按需打印
        if (strResponse != "pong" || enablePingPongLog)
        {
            LOGI("msg sent to client [%s], sessionId: %s, session: 0x%0x, clientId: %s, accountId: %s, frontId: %s, msg: %s",
                getClientInfo(), m_strSessionId.c_str(), (int64_t)this,
                m_strClientId.c_str(), m_strAccountID.c_str(), BusinessSession::m_strFrontId.c_str(),
                strResponse.data());
        }
    }
}
```

需要说明的是，以上示例代码使用 C/C++ 语言编写，但是本节介绍的心跳包机制设计思路和注意事项是普适性原理，同样适用于其他编程语言。

欢迎关注公众号『easyserverdev』，同时，您也可以加入我的 QQ 群578019391。

原文发布于微信公众号 - 高性能服务器开发（easyserverdev）
原文发表时间：2019-08-07
本文参与[腾讯云自媒体分享计划](#)，欢迎正在阅读的你也加入，一起分享。
发表于 2019-08-09

举报



194 篇文章 • 121 人订阅

订阅专栏

- Linux内核中双向链表的经典实现
- C++17，使用 `string_view` 来避免复制
- 高并发下漏洞桶限流设计方案 - Redis
- 如何设计断线自动重连机制
- TCP 协议如何解决粘包、半包问题

我来说两句

0 条评论

登录后参与评论

上一篇：[JS基础测试: 下列使用中正确的是？](#)

下一篇：[通过phpmyadmin GETshell](#)

社区

专栏文章

互动问答

技术沙龙

技术快讯

团队主页

开发者手册

智能钛AI

活动

原创分享计划

自媒体分享计划

资源

在线学习中心

技术周刊

社区标签


开发者实验室

关于

社区规范

免责声明

联系我们



扫码关注云+社区

领取腾讯云代金券

Copyright © 2013-2019

Tencent Cloud. All Rights Reserved.

腾讯云 版权所有

京ICP备11018762号

京公网安备 11010802020287