服务器技术文档

创建时间：2016-09-07

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **版本** | **修改时间** | **修改人** | **审批人** | **备注** |
| **V0.0.1** | 2016-9-7 | 张亚磊 |  | 创建文档 |

目录

[I. 服务器结构 3](#_Toc310409005)

[1. 多进程架构图 3](#_Toc310409006)

[2. 线程模型 4](#_Toc310409007)

[II. 服务器环境部署 8](#_Toc310409008)

[III. 服务器功能 13](#_Toc310409013)

[1. 消息包结构 13](#_Toc310409014)

[2. 消息包流动过程 14](#_Toc310409015)

[3. 进程功能介绍 1](#_Toc310409016)7

[4. JS脚本模块 18](#_Toc310409020)

[5. 配置管理 20](#_Toc310409020)

[6. 代码风格 20](#_Toc310409020)

# 服务器结构

服务器是异步非阻塞的多进程多线程架构，运行于CentOS7操作系统。服务器底层使用C++编写，高效稳定，使用TCP协议进行进程间以及与客户端的通信。同时也支持UDP、Websocket、Http协议。网络通信协议使用自定义格式，具备加密功能，安全高效。游戏数据库支持MongoDB和Mysql，可以选择自己想要的数据库。逻辑层使用简单高效的Javascript进行开发，集成Google V8引擎解析脚本，保证脚本运行效率，有js经验的开发人员可以快速上手。

服务器分为LoginServer，GateServer，GameServer，MasterServer，DBServer，LogServer。

LoginServer：客户端发送账号密码到LoginServer，LoginServer去AccountDb验证，验证失败断开连接，验证成功，则生成session， 根据玩家账号hash和gate在线人数选择GateIp,GatePort，发送给客户端。

GateServer：接受客户端发过来的连接，向LoginServer进行session验证，验证成功玩家可以正常登录，然后就开始接收客户端消息，转发到Game，Master，类似网关中转器。

GameServer：1、创建角色，角色登录，从DbServer加载数据返回给client，存储玩家数据 2、场景管理模块 3、AOI模块 4、定时器管理模块 5、怪物AI模块 6、Player管理模块 7、C++消息解析包装成js对象

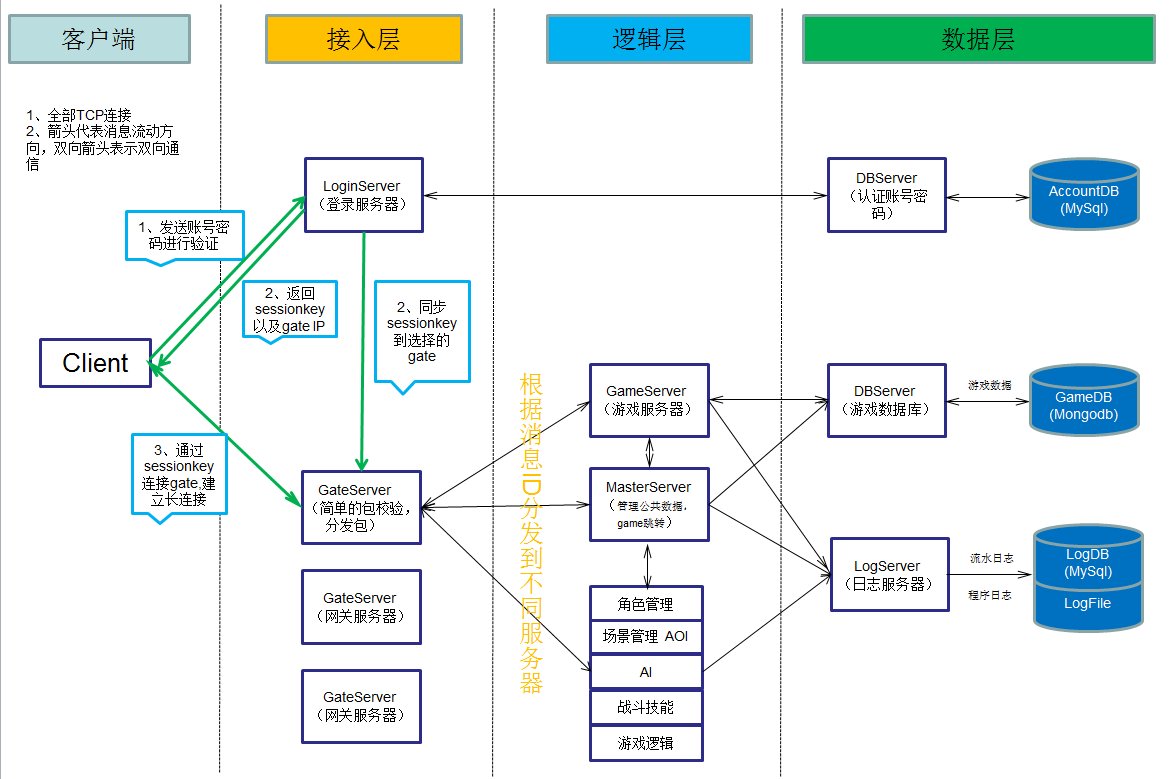
MasterServer： 1、管理GameServer，玩家在GameServer进程间跳转 2、存储在线玩家数据，进行相关校验验证，存储每个玩家对应的gameserver 3、存储公共数据，例如帮派，好友，邮件，聊天，组队

V8：V8引擎管理模块，js代码解析，执行js代码，C++函数导出

DbServer： 1、建立mongodb引擎和存取数据接口 2、建立Mysql引擎和存取数据接口

LogServer: 1、实现写服务器文件日志功能 2、实现写玩家操作日志到Mysql Log数据库

1、**多进程架构图**



2、**线程模型**

**Server线程：**

此线程包括3个网络线程，是逻辑上的接受TCP连接的服务端。A\_B\_Server即为在A进程运行的供B进程连接的服务器模块。连接建立后A与B的数据交换都由该线程模块完成。具体如下图

A\_B\_Server

Server\_Send

Server\_Receive

Server\_Accept

其中server\_accept线程负责监听端口，并接受客户端的连接封装成svc端点；Server\_receive线程负责从socket接受缓存中获取tcp包并按照规则解包，将解出后得到的完整逻辑包丢到服务器逻辑层以待逻辑层处理；Server\_send线程用于将队列中待发送的数据包丢进系统socket发送缓存。

**Connector线程：**

此线程包括3个网络线程，是逻辑上的发起TCP连接的客户端。A\_Connector即为在本地进程向A进程发起连接的子线程组。连接建立通过该组子线程完成与服务端A之间的通信。具体如下图

A\_Connector

Connector\_Connect

Connector\_Receive

Connector\_Send

其中connector\_connector线程负责向目标端口发起连接请求。并将对端封装成svc节点；connector\_receive线程负责从socket接收缓存中获取由目标端口发来的tcp包，将其解包后进行逻辑处理；connector\_send线程将数据包丢进发送缓存准备发往目的端口；

**Manager线程：**

Manager类线程是逻辑主线程，循环从队列中取出解出来的逻辑包，进行处理或者转投到脚本层。是维持整个进程运转的心脏。

**Login\_Server：**

Login\_Manager

Log\_Connector

Login\_Client\_Server

Timer

Login\_Gate\_Server

**Gate\_Server：**

Log\_Connector

Gate\_Manager

Gate\_Client\_Server

Gate\_Login\_Connector

Timer

Gate\_Game\_Connector

Gate\_Game\_Connector

Gate\_Game\_Connector

Gate\_Master\_Connector

**Game\_Server：**

Game\_Gate\_Server

Log\_Connector

Game\_Manager

Timer

Game\_DB\_Connector

Game\_Master\_Connector

Game\_V8\_Manager

**Master\_Server：**

Master\_Gate\_Server

Log\_Connector

Master\_Manager

Timer

Master\_V8\_Manager

Master\_DB\_Connector

Master\_Game\_Server

Master\_Http\_Server

**Log\_Server：**

Log\_Server

Log\_Connector

Log\_Manager

Timer

**DB\_Server：**

DB\_Manager

DB\_Server

Log\_Connector

DB\_Worker

DB\_Worker

DB\_Worker

1. **服务器环境部署**

环境：安装Vm 10.0以上版本，Centos 7.0以上版本

使用root账号进入系统，执行以下步骤

一、安装eclipse

http://www.eclipse.rg/downloads/download.php?file=/technology/epp/downloads/release/mars/1/eclipse-cpp-mars-1-linux-gtk-x86\_64.tar.gz

从这里下载64位eclipse for c++，放到/root 目录下面，执行下面命令

tar -zxvf eclipse-cpp-mars-1-linux-gtk-x86\_64.tar.gz

二、安装boost

yum install boost

yum install boost-devel

三、安装openssl

1、安装zlib

yum install zlib-devel

2、源码安装openssl

rm /usr/bin/pod2man

cd /root

tar -zxvf openssl-1.0.1g.tar.gz

cd openssl-1.0.1g

./config shared zlib-dynamic

make && make install

重命名原来的openssl

mv /usr/bin/openssl /usr/bin/openssl.old

将安装好的openssl 的openssl命令软连到/usr/bin/openssl

ln -s /usr/local/ssl/bin/openssl /usr/bin/openssl

将安装好的openssl 的openssl目录软连到/usr/include/openssl

ln -s /usr/local/ssl/include/openssl /usr/include/openssl

修改系统自带的openssl库文件，如/usr/local/lib64/libssl.so(根据机器环境而定) 软链到升级后的libssl.so

ln -s /usr/local/ssl/lib/libssl.so /usr/local/lib64/libssl.so

ln -s /usr/local/ssl/lib/libcrypto.so /usr/local/lib64/libcrypto.so

在/etc/ld.so.conf文件中写入openssl库文件的搜索路径

echo "/usr/local/ssl/lib" >> /etc/ld.so.conf

echo "/usr/local/lib64" >> /etc/ld.so.conf

使修改后的/etc/ld.so.conf生效

ldconfig

执行命令查看openssl依赖库版本是否为1.0.1g：

openssl version

四、安装curl

1、tar -zxf curl-7.50.1.tar.gz

2、cd curl-7.50.1

3、./configure --prefix=/usr/local/curl

4、make && make install

5、cp -rf /usr/local/curl/include /usr/local/include

6、cp -rf /usr/local/curl/lib/libcurl.so.4.4.0 /usr/local/curl/lib/libcurl.so.4

/usr/local/curl/lib/libcurl.so /usr/local/lib64

ldconfig

五、安装jsoncpp

1、tar -jxvf jsoncpp-bin-0.5.0-release.tar.bz2

2、cd jsoncpp-bin-0.5.0-release

3、cp -rf include/json /usr/local/include

4、cd libs/linux-gcc-4.4.7 cp libjson\_linux-gcc-4.4.7\_libmt.so /usr/local/lib64

5、建立软连接

cd /usr/local/lib64

ln -s libjson\_linux-gcc-4.4.7\_libmt.so libjsoncpp.so

ldconfig

如果想挑战下，可以尝试源码编译安装jsoncpp

tar -zxvf scons-2.1.0.tar.gz

export MYSCONS=/root/scons-2.1.0

export SCONS\_LIB\_DIR=$MYSCONS/engine

tar -zxvf jsoncpp-src-0.5.0.tar.gz

cd jsoncpp-src-0.5.0/

python $MYSCONS/script/scons platform=linux-gcc 编译生成相应的库

cp libs/linux-gcc-4.8.3/libjson\_linux-gcc-4.8.3\_libmt.a /usr/local/lib 将json库拷贝到系统库文件

cp libs/linux-gcc-4.8.3/libjson\_linux-gcc-4.8.3\_libmt.so /usr/local/lib

六、安装mysql

1. 下载mysql的repo源

wget http://repo.mysql.com/mysql-community-release-el7-5.noarch.rpm

2. 安装mysql-community-release-el7-5.noarch.rpm包

rpm -ivh mysql-community-release-el7-5.noarch.rpm

安装这个包后，会获得两个mysql的yum repo源：/etc/yum.repos.d/mysql-community.repo，/etc/yum.repos.d/mysql-community-source.repo。

3. 安装mysql

sudo yum install mysql-server

根据步骤安装就可以了，不过安装完成后，没有密码，需要重置密码。

4. 重置密码

重置密码前，首先要登录

mysql -u root

登录时有可能报这样的错：ERROR 2002 (HY000): Can‘t connect to local MySQL server through socket ‘/var/lib/mysql/mysql.sock‘ (2)，原因是/var/lib/mysql的访问权限问题。下面的命令把/var/lib/mysql的拥有者改为当前用户：

chown -R root:root /var/lib/mysql

然后，重启服务：

service mysqld restart

接下来登录重置密码：

mysql -u root

use mysql;

update user set password=password('123456') where user='root';

flush privileges;

5. 开放3306端口

vim /etc/sysconfig/iptables

添加以下内容：

-A INPUT -p tcp -m state --state NEW -m tcp --dport 3306 -j ACCEPT

保存后重启防火墙：

systemctl restart firewalld.service

这样从其它客户机也可以连接上mysql服务了。

七、安装mongodb（安装到/root/mongodb）

1、安装mongodb服务器

tar -zxvf mongodb-bin-linux-x86\_64-3.0.8.tar.gz

mv mongodb-linux-x86\_64-3.0.8 mongodb

mkdir /data

cd data

mkdir db

mkdir log

开启mongodb服务器

/root/mongodb/bin/mongod -dbpath=/data/db --fork --port 27017 --logpath=/data/log/work.log --logappend

将该命令添加到/etc/rc.local,保存，即可实现开机自动启动

启动mongodb客户端 /root/mongdb/bin/mongo

2、搭建mongodb编译环境

tar -jxvf mongodb-lib-3.0.8.tar.gz

cd mongodb-lib-3.0.8/

cp -rf include/mongo /usr/local/include

cp lib/libmongoclient.so /usr/local/lib64

cp lib/libmongoclient.a /usr/local/lib

ldconfig

3、如果有兴趣，可以进行源码编译安装mongodb

1、从yum安装boost, pcre, pcre-devel等依赖库。

执行命令 yum install boost

yum install pcre

yum install pcre-devel

2、安装scons

解压scons tar -zxvf scons-2.4.0.tar.gz cd scons-2.4.0

安装scons python setup.py install

3、编译mongodb-client

解压mongodb tar -zxvf mongodb-src-r2.0.4.tar.gz cd mongodb-src-r2.0.4

编译mongodb scons --prefix=/usr/mongo --sharedclient install

将在usr/mongo下生成bin, include, lib目录，lib下包括动态、静态mongoclient库。

八、拷贝netlib库 mysql-connector库 v8库

进入server/doc目录下面

1、拷贝netlib

tar -zxvf libnetlib.tar.gz

cd netlib/

cp libnetlib.so /usr/local/lib64

cp ./\* /usr/local/include/netlib

2、拷贝mysql-connector

tar -zxvf mysql-connector-c++-1.1.1-linux-el6-x86-64bit.tar.gz

cd mysql-connector-c++-1.1.1-linux-el6-x86-64bit/

cp -rf include/\* /usr/local/include

cp lib/libmysqlcppconn.so lib/libmysqlcppconn.so.6 lib/libmysqlcppconn.so.6.1.1.1 /usr/local/lib64

cp lib/libmysqlcppconn-static.a /usr/local/lib

3、拷贝v8运行环境

tar -zxvf v8.tar.gz

将所有so文件拷贝到/usr/local/lib64

将所有a文件拷贝到/usr/local/lib

将include文件夹拷贝到/usr/local/include

将natives\_blob.bin snapshot\_blob.bin拷贝到server同目录

执行命令ldconfig

1. **服务器功能**
2. **消息包结构**

+-----------------+----------------------+----------------------+

| head\_bytes| readable bytes | writeable bytes |

| | (CONTENT) | |

+-----------------+----------------------+----------------------+

0 read\_index(init\_offset) write\_index vector::size()

client message head:

int32(cid);

int16(len);

int32(serial\_cipher);

int32(msg\_time\_cipher);

int32(msg\_id);

int32(status);

gate to game,master,login,chat message head:

int32(cid);

int16(len);

int32(msg\_id);

int32(status);

int32(player\_cid);

inner message head;

int32(cid);

int16(len);

int32(msg\_id);

int32(status);

进行开发时消息包如何定义和获取参见《III-5配置管理》一节。

1. **消息包流动过程**

Server类表示一个服务器线程，包括完整的消息流动过程。

该类包括三个线程：

accept\_接受连接线程，

receive\_接收数据线程，线程内开启epoll\_watch，监听EPOLLIN事件

send\_发送数据线程，线程内开启epoll\_watch，监听EPOLLOUT事件

包含两个数据结构，用来处理svc连接

Object\_Pool<Server\_Svc, Spin\_Lock> svc\_static\_list\_ 存放svc的列表，所有client到server的连接都保存在这里

Svc\_Static\_List<Server\_Svc \*, Spin\_Lock> svc\_pool\_ svc的内存管理类

Socket流程简介：

客户端：

通过socket函数创建socketfd，connect服务器ip，port，连接成功后，可通过socketfd进行read/write与服务器进行通信。

服务端：

通过socket函数创建listenfd，bind相应的端口/ip，然后listen，等待有客户端连接。

当有客户端连接上来时候，通过accept接受连接，返回个connfd，然后可以通过connfd进行read/write，与客户端进行通信。

消息流动过程：

1、接受连接

Accept线程启动后，会在run\_handler函数里面调用server\_listen进行监听连接，有客户端连接到达时候，调用server\_accept接受该连接，然后调用accept\_svc处理连接后的svc，将svc存到svc\_static\_list\_里面，然后调用register\_recv\_handler将该svc注册到receiver线程的epoll里面，调用register\_send\_handler将该svc注册到sender线程的epoll里面，等待接受和发送数据。

流程如下：

Accept::thread\_create->run\_handler->server\_listen->server\_accept->

Server\_Accept::accept\_svc->

Svc\_Static\_List::record\_svc->

Server\_Svc::register\_recv\_handler->register\_send\_handler

2、接收数据

Receive线程里面的epoll\_watch在监听到EPOLLIN事件后，会触发Event\_Hander的handle\_input事件，因为Svc继承Event\_Handler类，receive的epoll监听了该svc，会调用Svc的handle\_input函数，从socket缓冲区读取到的数据Block\_Buffer通过push\_recv\_block函数放到Svc类的recv\_block\_list\_里面，然后调用Svc\_Handler的handler\_pack函数进行解包，最后将buffer放到server的block\_list\_

流程如下：

Server\_Svc::register\_recv\_handler->

Receive::register\_svc->(reactor\_->add(svc, Epoll\_Watcher::EVENT\_INPUT))

Epoll\_Watcher::watcher\_loop->Svc::handle\_input->

push\_recv\_block->recv\_block\_list\_(该结构存储接收到的数据BlockBuffer)

Svc\_Hander->handle\_pack->Server::block\_list\_

3、数据处理

以Game\_Gate\_Server为例，在启动时候，会开启一个循环，循环从Server的block\_list\_中取数据，通过GAME\_MANAGER->push\_game\_gate\_data(buf)放到Game\_Manager的game\_gate\_data\_list\_中。然后js脚本线程启动后，会不断的从game\_gate\_data\_list\_里面pop消息进行逻辑处理

流程如下：

Game\_Gate\_Server:thread\_create->run\_handler->process\_list->

Game\_Manager::push\_game\_gate\_data

game\_server.js::pop\_game\_gate\_msg\_object->process\_game\_gate\_msg

4、发送数据

当有数据需要发送给客户端时候，调用Gate\_Manger里面的send\_to\_client，然后将数据发送给send\_，存到append\_list\_里面，当send\_线程里面epoll监听的定时器时间到期时候，会调用send的hanlder\_timeout函数，该函数先调用append\_send\_block将append\_list\_里面的数据传送到svc的send\_block\_list\_里面，然后通过svc的handle\_send将数据发送到socket缓冲区，然后通过网络发送给客户端。

流程如下：

Game\_Manager::send\_to\_client->

Send::push\_data\_block\_with\_len->

Svc::append\_list\_(该结构存放待发送的Block\_Buffer)

Send:thread\_create->run\_handler->register\_self\_timer->

Epoll\_Watcher::watcher\_loop->

Send::handle\_timeout->append\_send\_block->

Svc::push\_send\_block->send\_block\_list\_(循环从list中取出数据通过socket发送)

Svc::handle\_send

5、连接关闭流程

Receive里面注册了个io心跳超时事件，当epoll监听到该事件时候，说明已经无法连接到客户端，服务器会主动关闭该连接，首先调用epoll\_watch里面的handle\_timeout函数，然后调用到svc类里面的handle\_close函数，然后调用Server\_Svc里面的close\_handler，该函数会通知Receive掉线的cid，放到drop\_list\_里面，在Receive的epoll处理drop\_list\_时候，又会通过Server\_Receive的drop\_handler函数通知Send处理掉线，放到Send的drop\_list\_里面，Send超时函数处理时候，在Send的drop\_handler里面，会回收svc，关闭fd，通知Game\_Manger，通过Game\_Manger的process\_drop\_cid让Game\_Player掉线。

流程如下：

Receive::init->Epoll\_Watcher::WITH\_IO\_HEARTBEAT

Epoll\_Watcher::watch\_loop->handle\_timeout->

svc:handle\_close->

Server\_Svc::close\_handler->

Receive::push\_drop->drop\_list->

Server\_Receive::drop\_handler->

Send::push\_drop\_->

Server\_Send:drop\_handler->

Game\_Manager::push\_drop\_cid->drop\_cid\_list\_->

Server::svc\_static\_list\_-> erase\_svc

关于cid

cid是服务器生成的一个id,每个客户端连接到服务器后，都会生成一个唯一的cid，用来查找客户端的连接Svc,cid保存在svc类中，在Svc::handle\_input时候，将cid写入到BlockBuffer的头部，然后消息组包，丢到逻辑层处理。当服务器需要向客户端发送消息时候，在Svc::handle\_send中，会将cid去掉，将包的内容发给客户端，所以在客户端和服务器的通信消息中，是不带cid的

1. **进程功能介绍**

Login\_Server

Login进程是客户端向服务器发起连接需要接触的第一个进程。它的主要作用是接收客户端的登录请求，并在账号数据库中查询角色信息。如果查找失败，则会通知客户端创建角色。查找成功则生成sessionkey并同步至负载较低的gate服务器，并返回该key和gate服务器的地址与端口给客户端，同时断开客户端连接。

Gate\_Server

Gate进程为网关进程。负责服务器和客户端之间消息的转发，隔离开逻辑服务器与客户端，对客户端发来的包进行校验，保证安全性。Gate进程在一组服务器环境中有多个副本，用于负载均衡。每个gate进程均需要连接所有的game逻辑进程和master进程。

客户端从login服务器获得sessionkey和地址端口等信息后连接gate服务器。并发送sessionkey进行验证。Gate服务器校验失败的话断开与客户端的连接。校验成功后gate会向特定的game服务器请求登陆，登陆成功后将角色信息返回客户端，并开始进行客户端消息和逻辑进程消息的中转。

Game\_Server

Game进程是处理游戏逻辑的主要进程。Game进程包含一个V8线程，拥有一个V8虚拟机实例，并读取脚本目录下的game\_server.js来执行逻辑处理。Game进程启动的时候根据配置文件读取相关场景和地图，并向master服务器报告注册自己的状态。Gate进程发来登陆请求后便开始登陆流程，通过DB进程获取玩家数据库中的信息返回给客户端，并在内存中生成玩家对应的实体对象。接着开始执行脚本中的大循环，处理具体逻辑。

Master\_Server

Master进程整个服务器组中只有一个，Gate进程和Game进程都要连接它，用来保存全局性质的数据，如公会，排行榜等，并对各个game的情况做统计，比如某个game上开辟了哪些场景等等。涉及到位于不同game进程的玩家交互的操作也需要通过Master来完成，比如聊天等等。Master也含有一个V8线程，拥有自己的V8虚拟机实例，可以通过脚本编写Master的具体逻辑。

Log\_Server

保存日志信息的进程，接收服务器组中各个组件进程的日志消息，插入mysql的日志数据库中。

DB\_Server

操作数据库的进程。目前已经支持了Mysql和Mongodb两种数据库。玩家登陆时拉取玩家信息，并每隔一段时间接收game或master传来的保存命令保存数据。

1. **JS脚本模块**

Javascript是新服务器所使用的脚本语言。开发的过程中，逻辑开发程序员只需要修改js脚本就能完成服务器功能的更新，不用重新编译二进制框架代码，很方便，也是目前主流框架普遍采用的做法。

新服务器使用google的v8引擎来解析脚本，在需要编写逻辑的Game和Master进程中都分别运行了V8虚拟机。Game进程读取运行脚本目录下game\_server.js脚本，Master进程则读取运行master\_server.js脚本。这两个xxx\_server脚本是v8线程运行的主循环脚本，脚本开头将其他脚本加载进内存，并初始化一些全局变量，然后就进入逻辑处理大循环。

主要脚本文件包括：game\_server.js；master\_server.js；game\_player.js；master\_player.js；error.js；message.js；struct.js；config.js；util.js；timer.js。

**game\_server.js、master\_server.js：**

game\_server.js和master\_server.js脚本分别是game进程和master进程的主循环脚本，也是被v8虚拟机加载执行的第一个脚本。通过require（）函数导入其他脚本后，便能在整个v8虚拟机环境中使用被导入的脚本定义的对象和方法。

脚本中执行的操作开始于main（）函数。在main（）函数中维持了一个大循环，每次循环会依次从队列中获取来自gate，db，master或game进程的数据包，并进行相应处理。同时，主循环还会检查是否有超时的定时器，如果存在超时定时器则执行完所有的超时操作后才会继续进行大循环逻辑。关于定时器部分参看timer.js的解析。

**game\_player.js、master\_player.js：**

Game\_server.js和master\_server.js定义了game玩家实体和master玩家实体。当玩家登录时，为玩家提供服务的game和master都会创建对应的玩家实体，用来进行逻辑操作和同步。Js中定义对象方法的时候通常使用原型定义的方法，比如



这样做的好处是可以节省内存，每个对象实例的方法都会通过原型指向唯一的一个函数索引。

Game\_player.js中定义了名为cplayer和centity的属性。这两者是用来操作c++层的对象的。因为js不是面向对象的语言，无法直接继承c++所实现的类型，因此采用把c++类型导出为脚本层对象属性的方式，让开发人员可以通过该属性访问c++层对应的类对象。

Player脚本里面已经定义好了玩家上线下线所必须的操作和回调，开发时有扩展需求只需添加原型函数即可。

**erro.js**：

Error.js脚本定义了所有需要使用的错误号

**message.js；struct.js：**

message.js中定义了解析和发送消息时能使用的各种消息号，struct.js中则定义了包括消息体在内的所有js层需要使用的类型。

这两个文件都是使用工具自动生成的，不要手动更改其内容，需要更改的话要改相应的配置文件，并使用序列化工具重新生成。详细过程参见《配置管理》一节。

**config.js**

Config.js是v8虚拟机能引用的配置文件。脚本初始化时便会根据config.js定义的全局变量和路径去读取供脚本层使用的json配置文件，在开发过程中使用config来访问各种配置参数比较方便。

**util.js**

Util.js脚本定义了各种公共操作，比如取时间啊，判断类型啊等等。

**timer.js**

Timer.js是操作定时器的文件。因为定时功能需要比较细致的时间区分度，因此在c++层来实现定时器，并抛出给js层使用。Js注册定时器的时候只需要提供一个定时器编号，并绑定回调函数，相关定时器超时时c++层会抛出超时定时器的编号到js层，js脚本获取该编号，执行相应的回调函数。

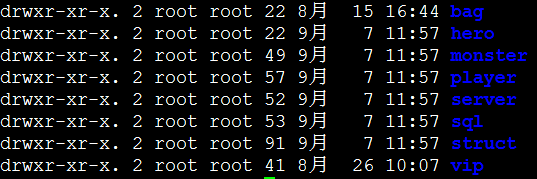
**客户端登录流程**

客户端发送120001到game\_c++,game\_c++将消息转到game\_js，game\_js发送150001到db加载数据，如果数据存在，db将数据发生给game\_c++,转到game\_js,game\_js解包数据，存储在game\_js内存中，同时将gate\_cid, player\_cid, role\_id设置到game\_c++,发送消息160000同步数据到master\_c++,master\_c++将数据转给master\_js,master\_js解包数据，存储在master\_js内存中，登录完成

如果数据不存在，game\_js返回520001给客户端，客户端发送120002创建角色，创建完成从数据库加载数据，然后流程同上

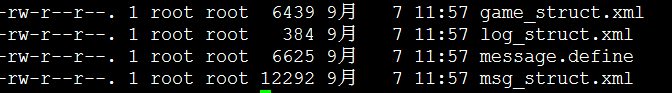
1. **配置管理**

新服务器使用的数据结构，消息结构，服务器启动方式，AI等都是通过配置表配的。配置文件存放于根目录的config文件夹下。目前包含以下配置：



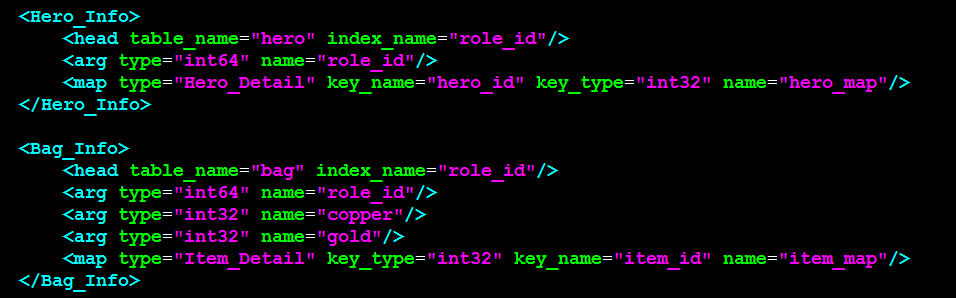
其中开发时需要经常改动的是struct文件夹下的配置文件和monster文件夹下的配置文件。

Struct目录下有以下配置表：



服务器xml解析功能用的是开源工具tinyxml，很小巧又功能齐全的一个xml解析工具。配表的时候，整张表需要新建一个根节点，名称为“root”，在根节点内新增的子节点就是服务器能读取并使用的具体节点。

其中game\_struct.xml为配置服务器基本数据结构的表。在定义DB进程所需的保存数据或是消息结构的时候，这里能定义一些复杂类型在msg\_struct.xml表中直接使用。以下图为例：



标签名即是该数据结构能被其他xml所引用的类型名。每个结构中都有子标签。

head子标签是该结构的头信息，包含head子标签的结构是将来要存入数据库的数据类型，其中table\_name属性是保存入数据库时的表名，index\_name属性是保存的时候用来作为索引的变量名。

Arg标签表示该结构含有一个基础数据类型成员，type属性为具体基础类型，name属性为该成员在脚本中可被引用的字段名。

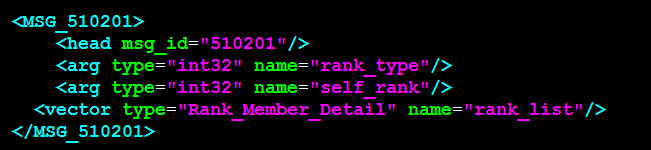
Vector标签表示该结构有一个数组成员，type属性是数组成员的类型，name是可被引用的变量名。

Map标签表示该结构有一个map对象，type属性为map里存放的类型，key\_name属性是该map的索引字段名，key\_type属性是该索引字段类型，name是此map本身可被引用的字段名。

Struct标签表示该结构含有一个复合结构字段，type是类型名，可以使任何在game\_struct中定义的其他类型。Name为可被引用的字段名。

Log\_struct.xml标签含义同game\_struct.xml。它定义的是存到数据库的日志的各字段信息。

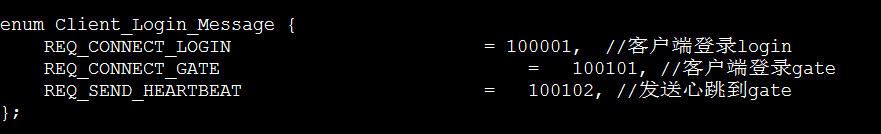
msg\_struct.xml中定义的是各种消息包的结构信息。如下所示：



标签名为脚本中可以被new出来的对象类型名。命名规则一般为MSG\_XXXXX，XXXXX为该消息的消息ID号。Head标签的属性msg\_id为该消息的消息id号，在message.Define文件中定义。其他标签属性和game\_struct.xml一样。

msg\_struct.xml定义的消息结构需要转化为js可用的脚本struct.js才能在v8虚拟机中正常使用。转化工具在server/doc下，载相对路径不变的前提下，直接执行server/doc/gen\_msg.sh脚本，即可自行生成struct.js文件。如果目录有变动，则需手动更改gen\_msg.sh配置正确路径。

Message.define文件为消息号定义文件，内容如下：



采用枚举的形式定义，枚举名称即是js脚本中可用的消息号别名， 枚举值是消息号本身。该文件更改后和msg\_struct.xml一样，需要运行gen\_msg.sh脚本才能正确生成v8虚拟机可运行的message.js文件。

1. **代码风格**

a、所有类名,结构体名首字母大写，如果有多个单词，单词中间用下划线分开

例如：Game\_Player

b、所有函数名字小写，如果有多个单词，单词中间用下划线分开

例如：int respond\_success\_result(int msg\_id, Block\_Buffer \*buf = 0);

c、所有变量名字小写，如果有多个单词，单词中间用下划线分开

例如：int gate\_cid\_;

d、所有宏定义，枚举类型大写，如果有多个单词，单词中间用下划线分开

例如：REQ\_FETCH\_ROLE\_INFO = 120001, //获取角色信息

e、消息号,消息结构体定义必须成对出现

struct MSG\_120001 {

};

struct MSG\_520001 {

};