Libsimulator 设计文档

**Revision record 修订记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **描述** | **作者** |
| 2014-06-30 | 1.0.0 | 创建文档 | owent |

# 1. 概述

Libsimulator 是由北极光工作室多款手游和页游自动化测试工具中提取出的一个快速搭建服务器自测工具和自动化测试系统的工具框架。

Libsimulator的设计目标是提供一个易于拓展、易于集成、操作便捷的测试框架。

# 2. 框架结构

## 2.1 系统要求

系统tLinux: 1.0 及以上

编译器: GCC 4.4 及以上，开启c++0x、c++11或c++1y标准

Readline库: 默认带tLinux+GCC4.4.5编译好的readline库，如果在其他平台或者需要更新readline库请手动更换lib目录下的两个.a文件和替换readline目录

## 2.2 整体结构

Libsimulator整体上是多线程结构。线程管理分为前台线程和后台线程。前台线程只有一个，用于控制命令行输入。另外还有一个后台线程池，用于处理一个或多个登入玩家的收发包和协议处理。整体结构如下所示：

Libsimulator整体结构图

Libsimulator的组成包含多个小工具，大致的结构如下：

* CmdShell、readline：用于管理终端输入
* ShellFont：管理终端颜色
* CmdOption：提供命令行分发和参数绑定支持
* ReqMgr、RspMgr、PlayerPool、DispatcherMgr：提供协议逻辑处理支持和线程管理
* Proto：【用户自定义部分】具体的协议处理逻辑
* 其他组件：【用户自定义部分】如脚本系统等，可以增加一些额外功能

终端输入时，输入的命令会由前台进程转入到对应玩家的命令列表里，然后等待后台进程触发。流程如下：



Libsimulator命令管理

# 业务接入

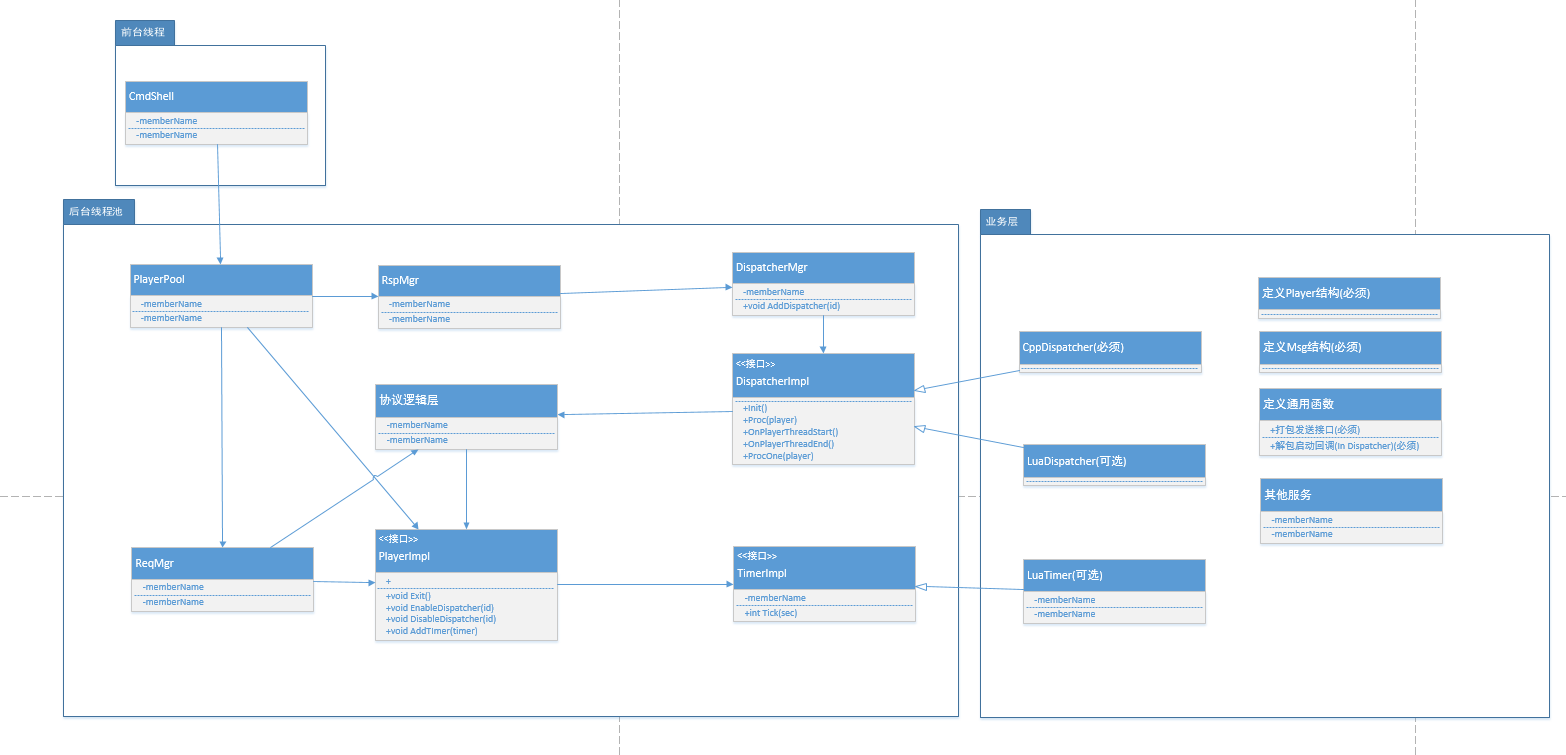
## 3.1 业务接入点

由于需要支持多中接入方式和协议。所以libsimulator功能拆分的地方比较多。主要有：

* 玩家
  + 定义客户端的玩家结构类型
  + 维护连接信息和玩家资源管理
* 协议
  + 定义协议结构体类型
  + 定义协议打包方式
  + 定义ToString方法【可选】
  + 定义初始化方法【可选】
  + 定义命令关联的业务协议和回包的分发方式
  + 定义命令行自动补全支持
* 调度器
  + 至少定义C++协议调度器类型
  + 可选定义其他调度器，比如接入lua脚本或者javascript脚本调度器
* 定时器
  + 可以定义定时器来自动处理一些功能

## 3.2 业务接入

业务接入后整体结构如图所示:



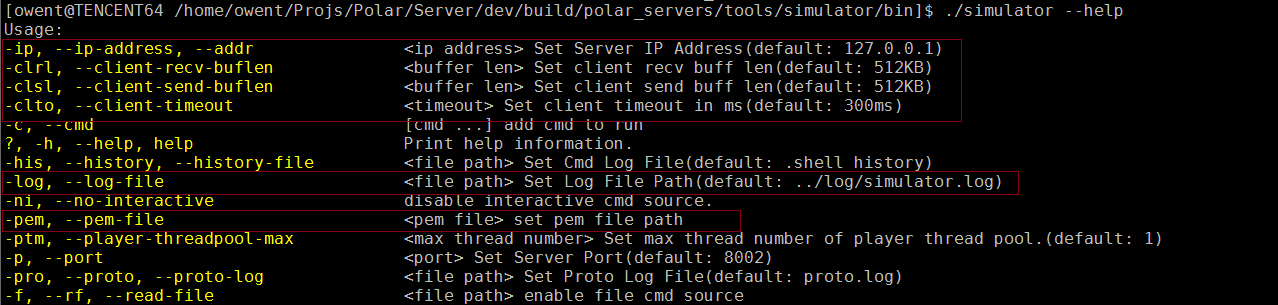
Libsimulator整体接入结构

简单的示例见【源码目录】/test/**simple\_test.cpp**

# 接入示例

## 4.1 自定义命令行参数

首先是提供友好并可定制的CLI工具:



自定义命令行参数

以上附加命令行参数的实现代码如下

// 追加命令绑定

BindingProgramParams(app\_simu\_t::*GetArgsMgr*());

copt::CmdOptionCI::ptr\_type **BindingProgramParams**(copt::CmdOptionCI::ptr\_type handler)

{

**using** **namespace** simulator::phoenix;

**using** simulator::detail::GenCmdHelpMsg;

// 绑定设置IP

handler->BindCmd("-ip, --ip-address, --addr", AssignAction<std::string>(AppConfigEx::*m\_strIP*))

->SetHelpMsg(GenCmdHelpMsg("-ip, --ip-address, --addr", "<ip address> Set Server IP Address(default: 127.0.0.1)").c\_str());

// 绑定设置端口

handler->BindCmd("-p, --port", AssignAction<**int**>(AppConfigEx::*m\_iPort*))

->SetHelpMsg(GenCmdHelpMsg("-p, --port", "<port> Set Server Port(default: 8002)").c\_str());

// 绑定设置日志文件路径

handler->BindCmd("-log, --log-file", AssignAction<std::string>(AppConfigEx::*m\_strLogFile*))

->SetHelpMsg(GenCmdHelpMsg("-log, --log-file", "<file path> Set Log File Path(default: ../log/simulator.log)").c\_str());

// 绑定设置每个角色的客户端接收缓冲区长度

handler->BindCmd("-clrl, --client-recv-buflen", AssignAction<size\_t>(AppConfigEx::*m\_iClientRecvBuffLen*))

->SetHelpMsg(GenCmdHelpMsg("-clrl, --client-recv-buflen", "<buffer len> Set client recv buff len(default: 512KB)").c\_str());

// 绑定设置每个角色的客户端发送缓冲区长度

handler->BindCmd("-clsl, --client-send-buflen", AssignAction<size\_t>(AppConfigEx::*m\_iClientSendBuffLen*))

->SetHelpMsg(GenCmdHelpMsg("-clsl, --client-send-buflen", "<buffer len> Set client send buff len(default: 512KB)").c\_str());

// 绑定设置每个角色的客户端超时时间

handler->BindCmd("-clto, --client-timeout", AssignAction<**int**>(AppConfigEx::*m\_iClientOprTimeout*))

->SetHelpMsg(GenCmdHelpMsg("-clto, --client-timeout", "<timeout> Set client timeout in ms(default: 300ms)").c\_str());

// 绑定密钥文件路径

handler->BindCmd("-pem, --pem-file", AssignAction<std::string>(AppConfigEx::*m\_strPemFile*))

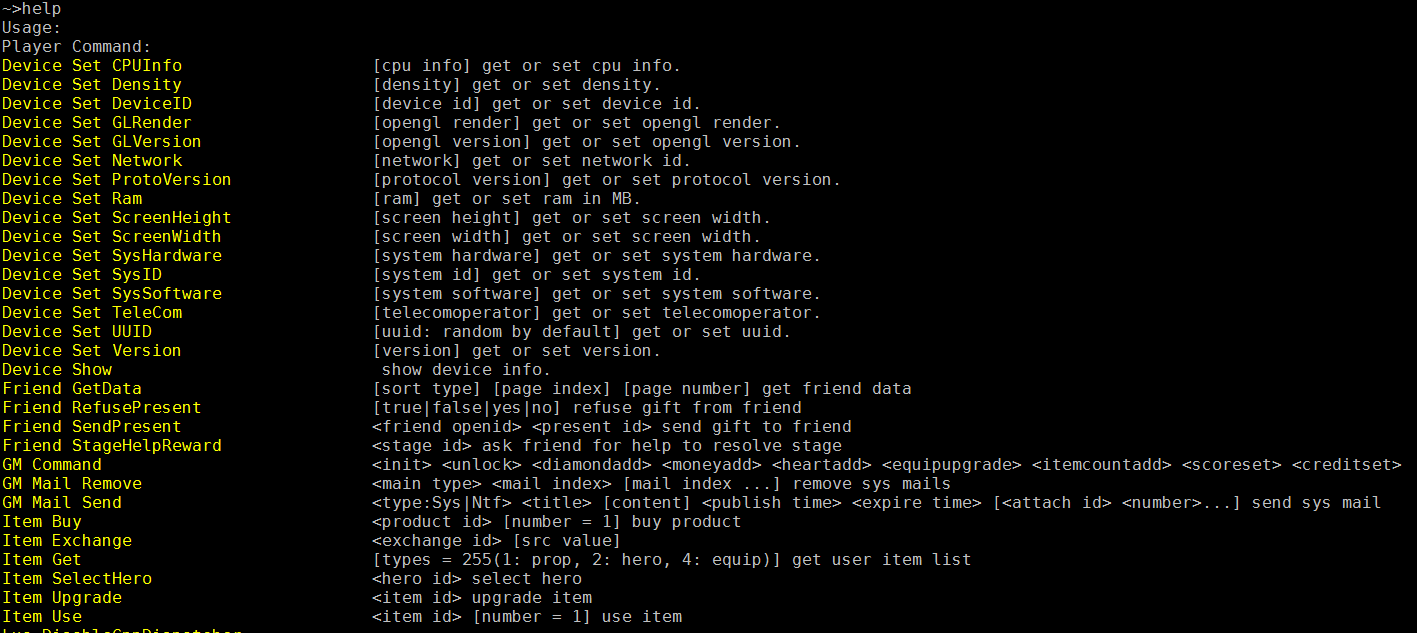
->SetHelpMsg(GenCmdHelpMsg("-pem, --pem-file", "<pem file> set pem file path").c\_str());

**return** handler;

}

## 4.2 自定义命令和响应回包

所有协议都一个通过一个命令行发送，输入help可以查看所有命令：



自定义命令

拿上图中的Friend GetData命令举例，命令协议申明如下：

**class** FriendDataProto: **public** simulator::ReqRspBase<FriendDataProto, app\_simu\_t::msg\_t, app\_simu\_t::player\_t>

{

**private**:

/\*\*

\* 初始化自动完成

\*/

**void** **\_init\_auto\_complete**();

/\*\*

\* 初始化请求命令

\*/

**void** **\_init\_req\_command**();

/\*\*

\* 初始化回包

\*/

**void** **\_init\_rsp\_callback**();

**public**:

**FriendDataProto**();

**~FriendDataProto**();

**public**:

// ============== 请求命令函数 =============

**void** **OnCmdGetData**(copt::callback\_param stParam);

// ============== 回包响应函数 =============

**void** **OnRspGetData**(std::shared\_ptr<CPlayer>, **const** Polar::CSPbMsg& stPkg);

};

CPP文件中，首先要用**ACTIVE\_PROTO**注册这个类

// 激活协议分发器

ACTIVE\_PROTO(FriendDataProto)

然后分别注册自动完成、命令回调函数和回包回调函数

**void** **FriendDataProto::\_init\_auto\_complete**()

{

RegisterAutoComplete()["Friend"]["GetData"];

}

**void** **FriendDataProto::\_init\_req\_command**()

{

std::shared\_ptr<copt::CmdOptionCI> pBase = GetCmdManager("Friend");

pBase->BindCmd("GetData", &FriendDataProto::OnCmdGetData, **this**)

->SetHelpMsg(simulator::ReqMgrBase::*GenCmdHelpMsg*("GetData", "[sort type] [page index] [page number] get friend data", 30).c\_str());

}

**void** **FriendDataProto::\_init\_rsp\_callback**()

{

SetRspProcessFunc(**static\_cast**<uint32\_t>(Polar::*SCMSG\_FRIEND\_GET\_DATA*),

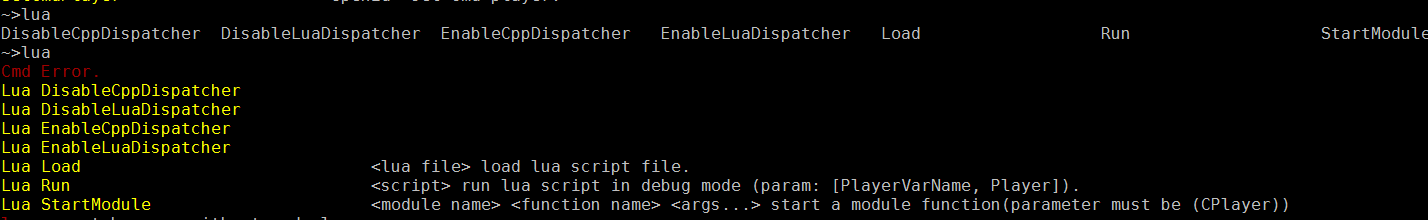
std::bind(&FriendDataProto::OnRspGetData, **this**, std::placeholders::\_1, std::placeholders::\_2));

}

然后在对应的OnCmdGetData函数和OnRspGetData函数里填充请求和处理回包即可。

## 4.3 接入其他组件

其他组建需要手动接入，这里我们有一个接入了Lua的示例截图：



接入Lua

允许系统启动lua状态机并使用lua来处理逻辑。如果需要使用Lua定时器，可以在玩家构造函数中添加脚本定时器。

**int** **CPlayer::OnCreated**()

{

// 添加Lua计时器

pLuaTimer = AddTimer<LuaTimer>();

// 初始化连接

**return** Connect(AppConfigEx::*m\_strIP*, AppConfigEx::*m\_iPort*);

}

Lua定时器可以如下所示：

**LuaTimer::LuaTimer**()

{

m\_tTimeNow = **time**(NULL);

}

**LuaTimer::~LuaTimer**()

{

}

**int** **LuaTimer::Init**()

{

**return** 0;

}

**int** **LuaTimer::Tick**(time\_t sec)

{

m\_tTimeNow = sec;

// 处理 lua timeout 回调

std::multimap<time\_t, LuaTimeoutHanlder>::iterator iter = m\_stLuaTimeoutHanlder.begin();

**for** (; iter != m\_stLuaTimeoutHanlder.end() && iter->first < m\_tTimeNow; )

{

std::string strResult;

SimulatorLuaScript::*Instance*()->DoCallFunction(iter->second.strFnName, &strResult, GetOwner<player\_t>()->shared\_from\_this(), iter->second.strParam);

m\_stLuaTimeoutHanlder.erase(iter ++);

}

**return** 0;

}

**int** **LuaTimer::SetTimeout**(int32\_t time\_sec, **const** std::string& strFuncName)

{

time\_t tTarget = **static\_cast**<time\_t>(time\_sec) + m\_tTimeNow;

m\_stLuaTimeoutHanlder.insert(std::make\_pair(tTarget, LuaTimeoutHanlder(strFuncName)));

**return** 0;

}

**int** **LuaTimer::SetTimeoutEx**(int32\_t time\_sec, **const** std::string& strFuncName, **const** std::string& strParam)

{

time\_t tTarget = **static\_cast**<time\_t>(time\_sec) + m\_tTimeNow;

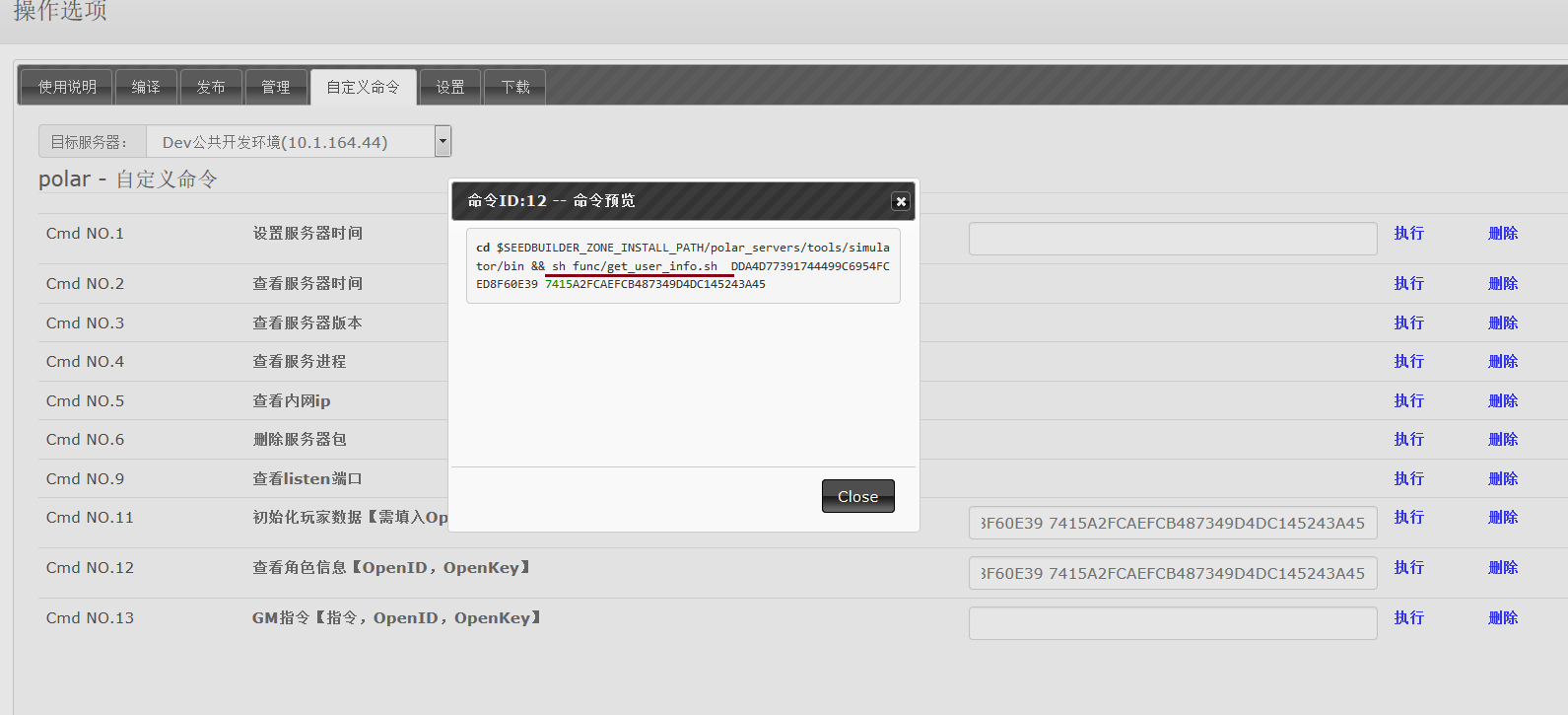
m\_stLuaTimeoutHanlder.insert(std::make\_pair(tTarget, LuaTimeoutHanlder(strFuncName, strParam)));

**return** 0;

}

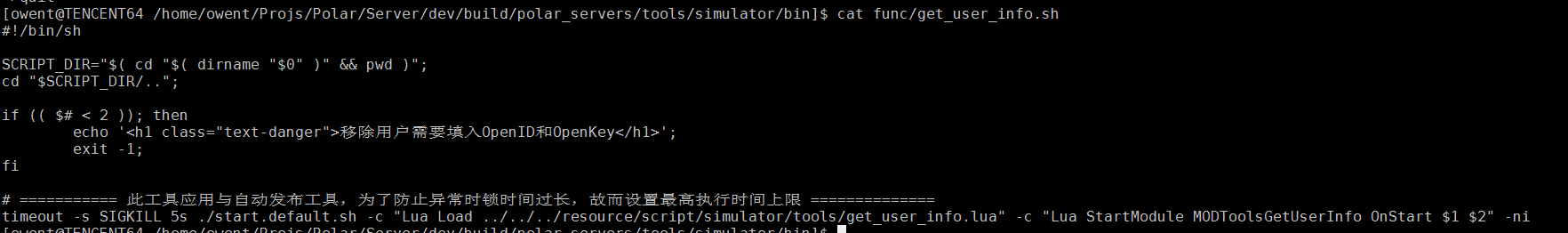
## 4.4 接入外部系统

同时simulator的-ni选项可以支持让simulator支持无人值守的功能。以此可以接入一些外部系统。比如可以接入Web工具集。



外部工具

其中红线部分命令如下：



外部工具支持脚本

可以看到，这里面用到了 -ni 来关闭终端输入，然后可以通过-c命令来插入用户命令。这里的用户命令是启动Lua脚本，所以最终逻辑将会转向Lua处理。

# 更新历史

2014/06/30 - 1.0.0

分离simulatir库，创建文档。