操作系统：CentOS 7.2 64位

服务器：Apache/2.4.6

后台语言：PHP 5.4.16

数据库：MongoDB 2.6.12

PHP扩展工具：Swoole

远程登录工具：Xshell 5

文件传输ftp工具：Xftp 5

php编辑器：notepad++ v6.9.2

小程序前端编辑器：

权限：root

（注：Apache支持php但不支持java，tomcat支持java但不支持php。本文操作中同时安装了tomcat 7，但未使用。）

所有执行的操作：

1. 服务器环境配置
2. 安装Apache、PHP包：

yum -y install httpd php

1. 安装Apache常用拓展包：

yum -y install httpd-manual mod\_ssl mod\_perl

1. 配置Apache开机启动：

chkconfig httpd on

1. 启动apache：

service httpd start 或 systemctl restart httpd（重启）

1. 查看apache版本：

httpd -v

1. 查看默认tomcat版本：

yum list available tomcat\*

1. 安装该版本tomcat：

yum install tomcat tomcat-webapps tomcat-admin-webapps

1. 开机启动tomcat：

chkconfig tomcat on

1. 启动tomcat

service tomcat start 或 systemctl restart tomcat（重启）

1. 安装w3m：

yum install w3m

1. 访问localhost网页：

w3m <http://localhost:8080/>

出现提示说明tomcat安装成功，tomcat的安装路径为/etc/tomcat，默认网站解析目录为/var/lib/tomcat/webapps/ROOT

1. 腾讯云服务器设置安全组

打开腾讯云服务器管理页面<https://console.qcloud.com/cvm/index>，点击左侧“安全组”，在规则列表中，点击所使用规则右侧“编辑规则”，在“入站规则”和“出站规则”中加入以下端口：80（apache），8080（tomcat），443（https）

点击左侧“云主机”，在主机列表中，点击所使用的主机右侧“更多”->“配置安全组”，使用刚才所配置的安全组规则。

（可以使用以下网址检查端口是否开启：<http://tool.chinaz.com/port/>，如果未开放：http://xingerheyaolong.iteye.com/blog/2115181）

1. 在浏览器中访问：

在浏览器中访问：http://公网IP:8080/，能够出现tomcat的欢迎页就代表成功。（注意此处必须是IP不可是域名）

在浏览器中访问：http://公网IP/，能够出现“Testing 123..”页面就代表apache安装成功。（注意此处必须是IP不可是域名）,apache的安装路径为/etc/httpd，默认网站解析目录为/var/www/html，编写php文件放于此目录下即可在浏览器中显示。

// 示例

编写文件test.php：

<?php

echo "Hello php";

?>

将该文件放在/var/www/html路径下，在浏览器中访问<http://115.159.49.38/test.php>，将输出Hello php。

1. 修改apache默认网站解析目录：

1） 创建目录：

mkdir /data/release/xjtudance

2） 修改apache配置文件：

（1）打开文件/etc/httpd/conf/httpd.conf，修改DocumentRoot "/var/www/html"为DocumentRoot "/data/release/xjtudance"

（2）注释以下段落：

<Directory "/var/www">

AllowOverride None

# Allow open access:

Require all granted

</Directory>

（3）修改<Directory "/var/www/html">为<Directory "/data/release/xjtudance">

（4）将上面编写的文件test.php放在目录/data/release/xjtudance下，即可在浏览器端显示。

其中/data/release/xjtudance为服务器端根目录。

1. Apache https证书配置：

1）申请证书（略）

2）安装openssl：

yum install mod\_ssl openssl

3）在apache根目录/etc/httpd/下新建文件夹certs，将证书文件（.crt,.crt,.key，共三个）放到该文件夹下

4）修改配置文件/etc/httpd/conf.d/ssl.conf：

修改SSLCertificateFile /etc/pki/tls/certs/localhost.crt为

SSLCertificateFile /etc/httpd/certs/2\_57247578.qcloud.la.crt,

修改SSLCertificateKeyFile /etc/pki/tls/private/localhost.key为

SSLCertificateKeyFile /etc/httpd/certs/3\_57247578.qcloud.la.key,

5）systemctl restart httpd

1. Apache服务器绑定域名：

（以下步骤整了半天不知道为什么有问题，最后又不知道为什么好了，仅将所有操作记录如下，完成后可使用https+域名访问，http无法使用域名访问，只能通过ip地址访问，有可能是因为我在腾讯云申请的证书只适用于https。）

1）由于之前我安装了nginx服务器，调试过程中经常受到影响，一气之下直接删除：

yum remove nginx

2）使用netstat –nap|grep 80、netstat –nap|grep 443确认占用端口的是httpd

3）修改apache配置文件/etc/httpd/conf/httpd.conf，在末尾添加：

<VirtualHost \*:80>

ServerAdmin root@localhost

DocumentRoot /data/release/xjtudance

ServerName 57247578.qcloud.la

</VirtualHost>

其中ServerAdmin抄的原文件中的配置（原文件中搜索ServerAdmin），DocumentRoot为解析根目录，ServerName为默认域名。

4）service httpd restart

1. 安装并配置MongoDB：

1）安装 MongoDB：

使用 Yum 在机器上安装 MongoDB 及其客户端命令行工具：

yum install mongodb-server mongodb –y

安装结束后，可以使用下面的命令查看安装的版本：

mongod --version

mongo –version

2）启动 MongoDB

创建目录，用于 MongoDB 数据和日志存储：

mkdir -p /data/release/xjtudance/data/mongodb

mkdir -p /data/release/xjtudance/data/logs/mongodb

创建后，使用下面的命令来启动 MongoDB（MongoDB 首次启动可能会花费大概 1min 时间，请耐心等待）：

mongod --fork --dbpath /data/release/xjtudance/data/mongodb --logpath /data/release/xjtudance/data/logs/mongodb/weapp.log

可以使用下面的命令来检查是否启动成功，MongoDB 默认监听27017端口等待连接，下面的命令查看当前27017端口被哪个进程占用，如果是MongoDB的进程，则表示启动成功。

netstat -ltp | grep 27017

（以后可使用service mongod start启动服务。）

开机启动mongodb：

chkconfig mongod on

3）安装php-mongo扩展：

yum install php-pecl-mongo mongodb-devel

重启apache服务器：systemctl restart tomcat

编写test.php文件：

<?php

echo "Hello php";

phpinfo();

?>

放于服务器网站解析根目录/data/release/xjtudance/下，在浏览器中输入<https://57247578.qcloud.la/test.php>，会输出php配置信息，如有mongo项，则说明php-mongo扩展安装成功。

（安装扩展可能遇到的问题见本文档“遇到的问题”章节。）

4）通过PHP代码测试mongodb，看是否能够正常运行：

<?php

$mongo = new Mongo();

$alldb = $mongo->listDBs();//获取所有数据库名称

echo $alldb;

如果正常返回则说明php操作mongodb成功。

3）添加 MongoDB 用户

登录本地 MongoDB 服务：

mongo

登录后，创建一个用户 weapp（创建的用户和密码将用于下一步中连接数据库时使用，如果使用不同的用户或密码，注意要保存好）：

use weapp;

db.createUser({ user: 'weapp', pwd: 'weapp-dev', roles: ['dbAdmin', 'readWrite']});

创建完成后，使用 exit 退出命令行工具。

1. 安装Swoole

Swoole是一个PHP插件，提供WebSocket等功能，网址为<http://www.swoole.com/>。

0）预先安装：

yum install gcc

yum install make

yum install autoconf

yum install pcre-devel

1）pecl install swoole

如出现错误Can't find PHP headers in /usr/include/php...，顺序执行以下操作：

yum install php-pear

yum install gcc gcc-c++ autoconf automake

pecl install swoole

注意自己的php版本。

2）配置php.ini：

用php -i |grep php.ini来定位到php.ini的绝对路径，修改php.ini加入extension = swoole.so。通过php -m或phpinfo()来查看是否成功加载了swoole。

（参见https://wiki.swoole.com/wiki/page/6.html）

遇到的问题：

1. 安装php-mongo扩展时报错：所需php版本不一致：
2. 在PHP中向MongoDB插入数据时不支持中文字符：

参见腾讯云开发者实验室：使用CentOS搭建微信小程序服务

**一、准备域名和证书**

1. 域名注册

2. 域名解析

3. 申请SSL证书

**二、搭建小程序开发环境**

（使用微信web开发者工具）

1. 注册开发者账号

2. 配置小程序服务器信息

登录微信公众平台后，依次进入 设置 - 开发设置 - 服务器域名 - 修改。

扫码完成身份校验后，request 合法域名和 socket 合法域名均填写在上一步准备好的域名地址。

配置完成后，点击 保存并提交。

3. 运行配套小程序代码

4. 设置实验域名

在开发工具的 编辑 面板中，选中 app.js 进行编辑，需要修改小程序通信域名

，请参考下面的配置：

App({

config: {

host: '' // 这个地方填写你的域名（dance为57247578.qcloud.la）

},

onLaunch () {

console.log('App.onLaunch()');

}

});

**三、搭建HTTP服务**

0. 在控制台设置主机root密码，获取云主机公网IP，输入Xshell地址栏，打开主机，用户名为root，密码为设置的密码。

1. 安装 NodeJS 和 NPM

使用下面的命令安装 NodeJS 和 NPM

yum install nodejs npm –y

安装完成后，使用下面的命令测试安装结果

node –v

2. 编写 HTTP Server 源码

2.1 创建工作目录

使用下面的命令在服务器创建一个工作目录：

mkdir -p /data/release/weapp （app目录名）

进入此工作目录

cd /data/release/weapp

2.2 创建 package.json

在刚才创建的工作目录创建 package.json，添加我们服务器包的名称和版本号，可参考下面的示例。

{

"name": "weapp",

"version": "1.0.0"

}

2.3 添加 Server 源码

在工作目录创建 app.js，使用 Express.js 来监听 8765 端口

，可参考下面的示例代码。

// 引用 express 来支持 HTTP Server 的实现

const express = require('express');

// 创建一个 express 实例

const app = express();

// 实现唯一的一个中间件，对于所有请求，都输出 "Response from express"

app.use((request, response, next) => {

response.write('Response from express');

response.end();

});

// 监听端口，等待连接

const port = 8765;

app.listen(port);

// 输出服务器启动日志

console.log(`Server listening at [http://127.0.0.1:${port}`](http://127.0.0.1:$%7bport%7d%60));

3. 运行 HTTP 服务

3.1 安装 PM2

在开始之前，我们先来安装 PM2

npm install pm2 –global

PM2 安装时间可能稍长，请耐心等候。（NPM 仓库在国内访问速度可能不太理想，如果实在太慢可以尝试使用 CNPM 的 Registry 进行安装：npm install pm2 -g --registry=https://r.cnpmjs.org/）

3.2 安装 Express

我们的服务器源码里使用到了 Express 模块，下面的命令使用 NPM 来安装 Express

cd /data/release/weapp

npm install express -save

3.3 启动服务

安装完成后，使用 PM2 来启动 HTTP 服务

cd /data/release/weapp

pm2 start app.js

现在，您的 HTTP 服务已经在 http://115.159.49.38 运行

要查看服务输出的日志，可以使用下面的命令：

pm2 logs

如果要重启服务，可以使用下面的命令：

pm2 restart app

**四、搭建 HTTPS 服务**

1. 安装 Nginx

在 CentOS 上，可直接使用 yum 来安装 Nginx

yum install nginx -y

安装完成后，使用 nginx 命令启动 Nginx：

nginx

此时，访问 http://115.159.49.38 可以看到 Nginx 的测试页面

如果无法访问，请重试用 nginx -s reload 命令重启 Nginx

2. 配置 HTTPS 反向代理

外网用户访问服务器的 Web 服务由 Nginx 提供，Nginx 需要配置反向代理才能使得 Web 服务转发到本地的 Node 服务。

先将之前下载的 SSL 证书(解压后 Nginx 目录分别以 crt 和 key 作为后缀的文件)通过拖动到左侧文件浏览器/etc/nginx目录的方式来上传文件到服务器上。

Nginx 配置目录在 /etc/nginx/conf.d，我们在该目录创建 ssl.conf：

server {

listen 443;

server\_name www.example.com; # 改为绑定证书的域名

# ssl 配置

ssl on;

ssl\_certificate 1\_www.example.com\_bundle.crt; # 改为自己申请得到的 crt 文件的名称

ssl\_certificate\_key 2\_www.example.com.key; # 改为自己申请得到的 key 文件的名称

ssl\_session\_timeout 5m;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

ssl\_ciphers ECDHE-RSA-AES128-GCM-SHA256:HIGH:!aNULL:!MD5:!RC4:!DHE;

ssl\_prefer\_server\_ciphers on;

location / {

proxy\_pass http://127.0.0.1:8765;

}

}

按 Ctrl + S 保存配置文件，让 Nginx 重新加载配置使其生效：

nginx -s reload

3. 在小程序中测试 HTTPS 访问

打开配套的小程序，点击 实验一：HTTPS，点击 发送请求 来测试访问结果。

如果服务器响应成功，请点击下一步。

**五、小程序会话**

小程序不支持 Cookie 存储和跟踪，服务器需要自行实现会话层

1. 安装 MongoDB

使用 Yum 在机器上安装 MongoDB 及其客户端命令行工具：

yum install mongodb-server mongodb –y

安装结束后，可以使用下面的命令查看安装的版本：

mongod --version

mongo –version

2. 启动 MongoDB

创建目录，用于 MongoDB 数据和日志存储：

mkdir -p /data/mongodb

mkdir -p /data/logs/mongodb

创建后，使用下面的命令来启动 MongoDB（MongoDB 首次启动可能会花费大概 1min 时间，请耐心等待）：

mongod --fork --dbpath /data/mongodb --logpath /data/logs/mongodb/weapp.log

可以使用下面的命令来检查是否启动成功，MongoDB 默认监听 27017 端口等待连接，下面的命令查看当前 27017 端口被哪个进程占用，如果是 MongoDB 的进程，则表示启动成功。

netstat -ltp | grep 27017

3. 添加 MongoDB 用户

登录本地 MongoDB 服务：

mongo

登录后，创建一个用户 weapp（创建的用户和密码将用于下一步中连接数据库时使用，如果使用不同的用户或密码，注意要保存好）：

use weapp;

db.createUser({ user: 'weapp', pwd: 'weapp-dev', roles: ['dbAdmin', 'readWrite']});

创建完成后，使用 exit 退出命令行工具。

4. 安装 Node 模块

实现小程序的会话功能，我们需要安装 connect-mongo 和 wafer-node-session

cd /data/release/weapp

npm install connect-mongo wafer-node-session –save

5. 实现小程序会话

在工作目录（/data/release/weapp/）创建配置文件 config.js，用于保存我们服务所用的配置（随着服务变得复杂，我们可以把配置集中起来方便管理，比如目前我们需要保存：服务器运行端口、小程序配置、MongoDB 连接配置），可参考下面的实现(注：请将参考配置文件中的 YORU\_APP\_ID 和 YOUR\_APP\_SECRET 替换为你申请的小程序对应的 AppID 和 AppSecret)：

module.exports = {

serverPort: '8765',

// 小程序 appId 和 appSecret

// 请到 https://mp.weixin.qq.com 获取 AppID 和 AppSecret

appId: 'YORU\_APP\_ID',

appSecret: 'YOUR\_APP\_SECRET',

// mongodb 连接配置，生产环境请使用更复杂的用户名密码

mongoHost: '127.0.0.1',

mongoPort: '27017',

mongoUser: 'weapp',

mongoPass: 'weapp-dev',

mongoDb: 'weapp'

};

编辑 app.js，添加会话实现逻辑，可参考下面的代码：

// 引用 express 来支持 HTTP Server 的实现

const express = require('express');

// 引用 wafer-session 支持小程序会话

const waferSession = require('wafer-node-session');

// 使用 MongoDB 作为会话的存储

const MongoStore = require('connect-mongo')(waferSession);

// 引入配置文件

const config = require('./config');

// 创建一个 express 实例

const app = express();

// 添加会话中间件，登录地址是 /login

app.use(waferSession({

appId: config.appId,

appSecret: config.appSecret,

loginPath: '/login',

store: new MongoStore({

url: `mongodb://${config.mongoUser}:${config.mongoPass}@${config.mongoHost}:${config.mongoPort}/${config.mongoDb}`

})

}));

// 在路由 /me 下，输出会话里包含的用户信息

app.use('/me', (request, response, next) => {

response.json(request.session ? request.session.userInfo : { noBody: true });

if (request.session) {

console.log(`Wafer session success with openId=${request.session.userInfo.openId}`);

}

});

// 实现一个中间件，对于未处理的请求，都输出 "Response from express"

app.use((request, response, next) => {

response.write('Response from express');

response.end();

});

// 监听端口，等待连接

app.listen(config.serverPort);

// 输出服务器启动日志

console.log(`Server listening at [http://127.0.0.1:${config.serverPort}`](http://127.0.0.1:$%7bconfig.serverPort%7d%60));

源码编写完成后，重启服务：

pm2 restart app

重启后，使用配套的小程序完成会话测试：打开配套小程序 - 点击 实验二：会话 - 获取会话，如果您能看到您的微信头像，那就表示会话已经成功获取了。

**六、WebSocket 服务**

1. 安装 Node 模块

本实验使用 ws 模块来在服务器上支持 WebSocket 协议，下面使用 NPM 来安装：

cd /data/release/weapp

npm install ws –save

2. 实现 WebSocket 服务器

创建 websocket.js，实现 WebSocket 服务，可参考下面的代码：

示例代码：/data/release/weapp/websocket.js

/\*\*

\* 在 HTTP Server 上处理 WebSocket 请求

\* @param {http.Server} server

\* @param {wafer.SessionMiddleware} sessionMiddleware

\*/

function listen(server, sessionMiddleware) {

// 使用 HTTP Server 创建 WebSocket 服务，使用 path 参数指定需要升级为 WebSocket 的路径

const wss = new ws.Server({ server, path: '/ws' });

// 监听 WebSocket 连接建立

wss.on('connection', (ws,request) => {// 要升级到 WebSocket 协议的 HTTP 连接

// 被升级到 WebSocket 的请求不会被 express 处理，

// 需要使用会话中间节获取会话

sessionMiddleware(request, null, () => {

const session = request.session;

if (!session) {

// 没有获取到会话，强制断开 WebSocket 连接

ws.send(JSON.stringify(request.sessionError) || "No session avaliable");

ws.close();

return;

}

// 保留这个日志的输出可让实验室能检查到当前步骤是否完成

console.log(`WebSocket client connected with openId=${session.userInfo.openId}`);

serveMessage(ws, session.userInfo);

});

});

// 监听 WebSocket 服务的错误

wss.on('error', (err) => {

console.log(err);

});

}

/\*\*

\* 进行简单的 WebSocket 服务，对于客户端发来的所有消息都回复回去

\*/

function serveMessage(ws, userInfo) {

// 监听客户端发来的消息

ws.on('message', (message) => {

console.log(`WebSocket received: ${message}`);

ws.send(`Server: Received(${message})`);

});

// 监听关闭事件

ws.on('close', (code, message) => {

console.log(`WebSocket client closed (code: ${code}, message: ${message || 'none'})`);

});

// 连接后马上发送 hello 消息给会话对应的用户

ws.send(`Server: 恭喜，${userInfo.nickName}`);

}

编辑 app.js，调用 WebSocket 服务，可参考下面代码：

示例代码：/data/release/weapp/app.js

// HTTP 模块同时支持 Express 和 WebSocket

const http = require('http');

// 引用 express 来支持 HTTP Server 的实现

const express = require('express');

// 引用 wafer-session 支持小程序会话

const waferSession = require('wafer-node-session');

// 使用 MongoDB 作为会话的存储

const MongoStore = require('connect-mongo')(waferSession);

// 引入配置文件

const config = require('./config');

// 引入 WebSocket 服务实现

const websocket = require('./websocket');

// 创建一个 express 实例

const app = express();

// 独立出会话中间件给 express 和 ws 使用

const sessionMiddleware = waferSession({

appId: config.appId,

appSecret: config.appSecret,

loginPath: '/login',

store: new MongoStore({

url: `mongodb://${config.mongoUser}:${config.mongoPass}@${config.mongoHost}:${config.mongoPort}/${config.mongoDb}`

})

});

app.use(sessionMiddleware);

// 在路由 /me 下，输出会话里包含的用户信息

app.use('/me', (request, response, next) => {

response.json(request.session ? request.session.userInfo : { noBody: true });

if (request.session) {

console.log(`Wafer session success with openId=${request.session.userInfo.openId}`);

}

});

// 实现一个中间件，对于未处理的请求，都输出 "Response from express"

app.use((request, response, next) => {

response.write('Response from express');

response.end();

});

// 创建 HTTP Server 而不是直接使用 express 监听

const server = http.createServer(app);

// 让 WebSocket 服务在创建的 HTTP 服务器上监听

websocket.listen(server, sessionMiddleware);

// 启动 HTTP 服务

server.listen(config.serverPort);

// 输出服务器启动日志

console.log(`Server listening at [http://127.0.0.1:${config.serverPort}`](http://127.0.0.1:$%7bconfig.serverPort%7d%60));

修改完成后，按 Ctrl + S 保存文件，并重启服务：

pm2 restart app

2. 更新 Nginx 代理

编辑 Nginx 配置 ssl.conf，添加 WebSocket 支持，可参考下面的配置(注：请将参考配置文件中的 www.example.com 替换为前面步骤申请的域名，将 1\_www.example.com.crt 和 2\_www.example.com.key 替换为前面步骤申请并上传的 SSL 证书的名称)：

示例代码：/etc/nginx/conf.d/ssl.conf

# WebSocket 配置

map $http\_upgrade $connection\_upgrade {

default upgrade;

'' close;

}

server {

listen 443;

server\_name www.example.com; # 改为绑定证书的域名

# ssl 配置

ssl on;

ssl\_certificate 1\_www.example.com.crt; # 改为自己申请得到的 crt 文件的名称

ssl\_certificate\_key 2\_www.example.com.key; # 改为自己申请得到的 key 文件的名称

ssl\_session\_timeout 5m;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

ssl\_ciphers ECDHE-RSA-AES128-GCM-SHA256:HIGH:!aNULL:!MD5:!RC4:!DHE;

ssl\_prefer\_server\_ciphers on;

# WebSocket 配置

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection $connection\_upgrade;

location / {

proxy\_pass http://127.0.0.1:8765;

}

}

配置完成后，按 Ctrl + S 保存，并且通知 Nginx 进程重新加载配置：

nginx -s reload

3. 测试 WebSocket

打开配套的小程序，点击 实验三：WebSocket。进入测试页面后，点击 连接 按钮，如果出现连接成功的提示，表示 WebSocket 服务已经正常运行，可以收发消息。

剪刀石头布小游戏

任务时间：45min ~ 90min

实现游戏房间逻辑

创建 /data/release/weapp/game 目录用于存放剪刀石头布小游戏的代码

mkdir -p /data/release/weapp/game

添加 game/Room.js 实现游戏房间逻辑[?]，可参考下面的代码：

示例代码：/data/release/weapp/game/Room.js

/\*\*

enum GameChoice {

// 剪刀

Scissors = 1,

// 石头

Rock = 2,

// 布

Paper = 3

}

\*/

function judge(choice1, choice2) {

// 和局

if (choice1 == choice2) return 0;

// Player 1 没出，Player 2 胜出

if (!choice1) return 1;

// Player 2 没出，Player 1 胜出

if (!choice2) return -1;

// 都出了就这么算

return (choice1 - choice2 + 3) % 3 == 1 ? -1 : 1;

}

/\*\* @type {Room[]} \*/

const globalRoomList = [];

// 每个房间最多两人

const MAX\_ROOT\_MEMBER = 2;

// 游戏时间，单位秒

const GAME\_TIME = 3;

let nextRoomId = 0;

/\*\* 表示一个房间 \*/

module.exports = class Room {

/\*\* 获取所有房间 \*/

static all() {

return globalRoomList.slice();

}

/\*\* 获取有座位的房间 \*/

static findRoomWithSeat() {

return globalRoomList.find(x => !x.isFull());

}

/\*\* 创建新房间 \*/

static create() {

const room = new Room();

globalRoomList.unshift(room);

return room;

}

constructor() {

this.id = `room${nextRoomId++}`;

this.players = [];

}

/\*\* 添加玩家 \*/

addPlayer(player) {

const { uid, uname } = player.user;

console.log(`Player ${uid}(${uname}) enter ${this.id}`);

this.players.push(player);

if (this.isFull()) {

this.startGame();

}

}

/\*\* 删除玩家 \*/

removePlayer(player) {

const { uid, uname } = player.user;

console.log(`Player ${uid}(${uname}) leave ${this.id}`);

const playerIndex = this.players.indexOf(player);

if (playerIndex != -1) {

this.players.splice(playerIndex, 1);

}

if (this.players.length === 0) {

console.log(`Room ${this.id} is empty now`);

const roomIndex = globalRoomList.indexOf(this);

if (roomIndex > -1) {

globalRoomList.splice(roomIndex, 1);

}

}

}

/\*\* 玩家已满 \*/

isFull() {

return this.players.length == MAX\_ROOT\_MEMBER;

}

/\*\* 开始游戏 \*/

startGame() {

// 保留这行日志输出可以让实验室检查到实验的完成情况

console.log('game started!');

// 当局积分清零

this.players.forEach(player => player.gameData.roundScore = 0);

// 集合玩家用户和游戏数据

const players = this.players.map(player => Object.assign({}, player.user, player.gameData));

// 通知所有玩家开始

for (let player of this.players) {

player.send('start', {

gameTime: GAME\_TIME,

players

});

}

// 计时结束

setTimeout(() => this.finishGame(), GAME\_TIME \* 1000);

}

/\*\* 结束游戏 \*/

finishGame() {

const players = this.players;

// 两两对比算分

for (let i = 0; i < MAX\_ROOT\_MEMBER; i++) {

let p1 = players[i];

if (!p1) break;

for (let j = i + 1; j < MAX\_ROOT\_MEMBER; j++) {

let p2 = players[j];

const result = judge(p1.gameData.choice, p2.gameData.choice);

p1.gameData.roundScore -= result;

p2.gameData.roundScore += result;

}

}

// 计算连胜奖励

for (let player of players) {

const gameData = player.gameData;

// 胜局积分

if (gameData.roundScore > 0) {

gameData.winStreak++;

gameData.roundScore \*= gameData.winStreak;

}

// 败局清零

else if (gameData.roundScore < 0) {

gameData.roundScore = 0;

gameData.winStreak = 0;

}

// 累积总分

gameData.totalScore += gameData.roundScore;

}

// 计算结果

const result = players.map(player => {

const { uid } = player.user;

const { roundScore, totalScore, winStreak, choice } = player.gameData;

return { uid, roundScore, totalScore, winStreak, choice };

});

// 通知所有玩家游戏结果

for (let player of players) {

player.send('result', { result });

}

}

}

处理游戏开始、计算结果、积分等逻辑

实现玩家逻辑

添加 game/Player.js 实现玩家逻辑[?]，可参考下面的代码：

示例代码：/data/release/weapp/game/Player.js

const Room = require("./Room");

/\*\*

\* 表示一个玩家，处理玩家的公共游戏逻辑，消息处理部分需要具体的玩家实现（请参考 ComputerPlayer 和 HumanPlayer）

\*/

module.exports = class Player {

constructor(user) {

this.id = user.uid;

this.user = user;

this.room = null;

this.gameData = {

// 当前的选择（剪刀/石头/布）

choice: null,

// 局积分

roundScore: 0,

// 总积分

totalScore: 0,

// 连胜次数

winStreak: 0

};

}

/\*\*

\* 上线当前玩家，并且异步返回给玩家分配的房间

\*/

online(room) {

// 处理玩家 'join' 消息

// 为玩家寻找一个可用的房间，并且异步返回

this.receive('join', () => {

if (this.room) {

this.room.removePlayer(this);

}

room = this.room = room || Room.findRoomWithSeat() || Room.create();

room.addPlayer(this);

});

// 处理玩家 'choise' 消息

// 需要记录玩家当前的选择，并且通知到房间里的其它玩家

this.receive('choice', ({ choice }) => {

this.gameData.choice = choice;

this.broadcast('movement', {

uid: this.user.uid,

movement: "choice"

});

});

// 处理玩家 'leave' 消息

// 让玩家下线

this.receive('leave', () => this.offline);

}

/\*\*

\* 下线当前玩家，从房间离开

\*/

offline() {

if (this.room) {

this.room.removePlayer(this);

this.room = null;

}

this.user = null;

this.gameData = null;

}

/\*\*

\* 发送指定消息给当前玩家，需要具体子类实现

\* @abstract

\* @param {string} message 消息类型

\* @param {\*} data 消息数据

\*/

send(message, data) {

throw new Error('Not implement: AbstractPlayer.send()');

}

/\*\*

\* 处理玩家发送的消息，需要具体子类实现

\* @abstract

\* @param {string} message 消息类型

\* @param {Function} handler

\*/

receive(message, handler) {

throw new Error('Not implement: AbstractPlayer.receive()');

}

/\*\*

\* 给玩家所在房间里的其它玩家发送消息

\* @param {string} message 消息类型

\* @param {any} data 消息数据

\*/

broadcast(message, data) {

if (!this.room) return;

this.others().forEach(neighbor => neighbor.send(message, data));

}

/\*\*

\* 获得玩家所在房间里的其他玩家

\*/

others() {

return this.room.players.filter(player => player != this);

}

}

处理玩家加入游戏、选择出拳、通知其他玩家等逻辑

实现电脑玩家

在实现人类玩家之前，我们先来创建 ComputerPlayer.js 来实现电脑玩家[?]

示例代码：/data/release/weapp/game/ComputerPlayer.js

const EventEmitter = require('events');

const Player = require('./Player');

let nextComputerId = 0;

/\*\*

\* 机器人玩家实现，使用 EventEmitter 接收和发送消息

\*/

module.exports = class ComputerPlayer extends Player {

constructor() {

const computerId = `robot-${++nextComputerId}`;

super({

uid: computerId,

uname: computerId,

uavatar: 'http://www.scoutiegirl.com/wp-content/uploads/2015/06/Blue-Robot.png'

});

this.emitter = new EventEmitter();

}

/\*\*

\* 模拟玩家行为

\*/

simulate() {

this.receive('start', () => this.play());

this.receive('result', () => this.stop());

this.send('join');

}

/\*\*

\* 游戏开始后，随机时间后随机选择

\*/

play() {

this.playing = true;

const randomTime = () => Math.floor(100 + Math.random() \* 2000);

const randomChoice = () => {

if (!this.playing) return;

this.send("choice", {

choice: Math.floor(Math.random() \* 10000) % 3 + 1

});

setTimeout(randomChoice, randomTime());

}

setTimeout(randomChoice, 10);

}

/\*\*

\* 游戏结束后，标记起来，阻止继续随机选择

\*/

stop() {

this.playing = false;

}

/\*\*

\* 发送消息给当前玩家，直接转发到 emitter

\*/

send(message, data) {

this.emitter.emit(message, data);

}

/\*\*

\* 从当前的 emitter 处理消息

\*/

receive(message, handle) {

this.emitter.on(message, handle);

}

}

测试游戏逻辑的时候，可能没有其它人可以一起参与，实现一个电脑玩家是不错的选择

实现人类玩家

人类玩家通过 WebSocket 信道来实现玩家的输入输出[?]，我们需要添加 game/Tunnel.js 和 game/HumanPlayer.js 来实现人类玩家逻辑，可参考下面的代码：

示例代码：/data/release/weapp/game/Tunnel.js

const EventEmitter = require('events');

/\*\*

\* 封装 WebSocket 信道

\*/

module.exports = class Tunnel {

constructor(ws) {

this.emitter = new EventEmitter();

this.ws = ws;

ws.on('message', packet => {

try {

// 约定每个数据包格式：{ message: 'type', data: any }

const { message, data } = JSON.parse(packet);

this.emitter.emit(message, data);

} catch (err) {

console.log('unknown packet: ' + packet);

}

});

}

on(message, handle) {

this.emitter.on(message, handle);

}

emit(message, data) {

this.ws.send(JSON.stringify({ message, data }));

}

}

示例代码：/data/release/weapp/game/HumanPlayer.js

const co = require('co');

const Player = require('./Player');

const ComputerPlayer = require('./ComputerPlayer');

const Tunnel = require('./Tunnel');

/\*\*

\* 人类玩家实现，通过 WebSocket 信道接收和发送消息

\*/

module.exports = class HumanPlayer extends Player {

constructor(user, ws) {

super(user);

this.ws = ws;

this.tunnel = new Tunnel(ws);

this.send('id', user);

}

/\*\*

\* 人类玩家上线后，还需要监听信道关闭，让玩家下线

\*/

online(room) {

super.online(room);

this.ws.on('close', () => this.offline());

// 人类玩家请求电脑玩家

this.receive('requestComputer', () => {

const room = this.room;

while(room && !room.isFull()) {

const computer = new ComputerPlayer();

computer.online(room);

computer.simulate();

}

});

}

/\*\*

\* 下线后关闭信道

\*/

offline() {

super.offline();

if (this.ws && this.ws.readyState == this.ws.OPEN) {

this.ws.close();

}

this.ws = null;

this.tunnel = null;

if (this.room) {

// 清理房间里面的电脑玩家

for (let player of this.room.players) {

if (player instanceof ComputerPlayer) {

this.room.removePlayer(player);

}

}

this.room = null;

}

}

/\*\*

\* 通过 WebSocket 信道发送消息给玩家

\*/

send(message, data) {

this.tunnel.emit(message, data);

}

/\*\*

\* 从 WebSocket 信道接收玩家的消息

\*/

receive(message, callback) {

this.tunnel.on(message, callback);

}

}

人类玩家和电脑玩家的逻辑是一致的，但是 IO 不同，人类玩家使用之前实现的 WebSocket 服务进行输入输出，而电脑玩家直接使用 EventEmiter 处理

添加游戏服务入口

游戏的实现已经完成了，接下来，编辑 websocket.js 添加服务入口，可参考下面的代码：

示例代码：/data/release/weapp/websocket.js

// 引入 url 模块用于解析 URL

const url = require('url');

// 引入 ws 支持 WebSocket 的实现

const ws = require('ws');

// 引入人类玩家

const HumanPlayer = require('./game/HumanPlayer');

// 导出处理方法

exports.listen = listen;

/\*\*

\* 在 HTTP Server 上处理 WebSocket 请求

\* @param {http.Server} server

\* @param {wafer.SessionMiddleware} sessionMiddleware

\*/

function listen(server, sessionMiddleware) {

// 使用 HTTP Server 创建 WebSocket 服务，使用 path 参数指定需要升级为 WebSocket 的路径

const wss = new ws.Server({ server });

// 同时支持 /ws 和 /game 的 WebSocket 连接请求

wss.shouldHandle = (request) => {

const path = url.parse(request.url).pathname;

request.path = path;

return ['/ws', '/game'].indexOf(path) > -1;

};

// 监听 WebSocket 连接建立

wss.on('connection', (ws, request) => {

// request: 要升级到 WebSocket 协议的 HTTP 连接

// 被升级到 WebSocket 的请求不会被 express 处理，

// 需要使用会话中间节获取会话

sessionMiddleware(request, null, () => {

const session = request.session;

if (!session) {

// 没有获取到会话，强制断开 WebSocket 连接

ws.send(JSON.stringify(request.sessionError) || "No session avaliable");

ws.close();

return;

}

console.log(`WebSocket client connected with openId=${session.userInfo.openId}`);

// 根据请求的地址进行不同处理

switch (request.path) {

case '/ws': return serveMessage(ws, session.userInfo);

case '/game': return serveGame(ws, session.userInfo);

default: return ws.close();

}

});

});

// 监听 WebSocket 服务的错误

wss.on('error', (err) => {

console.log(err);

});

}

/\*\*

\* 进行简单的 WebSocket 服务，对于客户端发来的所有消息都回复回去

\*/

function serveMessage(ws, userInfo) {

// 监听客户端发来的消息

ws.on('message', (message) => {

console.log(`WebSocket received: ${message}`);

ws.send(`Server: Received(${message})`);

});

// 监听关闭事件

ws.on('close', (code, message) => {

console.log(`WebSocket client closed (code: ${code}, message: ${message || 'none'})`);

});

// 连接后马上发送 hello 消息给会话对应的用户

ws.send(`Server: 恭喜，${userInfo.nickName}`);

}

/\*\*

\* 使用 WebSocket 进行游戏服务

\*/

function serveGame(ws, userInfo) {

const user = {

uid: userInfo.openId,

uname: userInfo.nickName,

uavatar: userInfo.avatarUrl

};

// 创建玩家

const player = new HumanPlayer(user, ws);

// 玩家上线

player.online();

}

安装 co 模块

我们的源码中使用到了 co 进行协程管理，启动游戏服务前，需要先安装：

cd /data/release/weapp

npm install co --save

测试游戏服务

重启 Node 服务：

pm2 restart app

打开配套的小程序，点击 实验四 - 剪刀石头布小游戏，点击 开始 按钮进行游戏。

完成实验

恭喜！您已经完成了小程序服务的全部实验内容！你可以选择保留已经运行的服务，继续进行小程序的学习研究，建议留用机器。