技术分享-游戏框架

1. 前置
    a. 游戏与其他软件的相同与不同
        i. 需求的来源
            1. 游戏----制作过程中全造策划拍脑袋，上线后会结合一定的用户的反馈
            2. 一般APP----与用户沟通等途径进行需求分析
        ii. 用户间交互的频繁度
            1. 游戏----多
            2. 一般APP----少
        iii. 用户的协议通讯的频繁度
            1. 游戏----场景类巨多，射击类游戏平均1秒4个消息，休闲类平均1秒1个消息左右
            2. 一般APP----相对少
2. 分享四套框架----顺序为我接触到的时间顺序，并无优劣之分
    a. 魔兽争霸
        i. 服务器事件类型
            1. 事件触发
            2. 时间触发
        ii. 暴雪自研脚本语言----jass语言
            1. 国内爱好者把功能翻译成了lua https://github.com/actboy168/jass2lua
            2. 一些资源
                a. 之前sourceforge的文档和一些工具： https://jass.sourceforge.net/doc/
                b. beginner tutorial：https://www.hiveworkshop.com/threads/beginning-jass-tutorial-series.30765/
                c. complete guild：https://www.thehelper.net/threads/the-complete-guide-to-jass-vjass-and-cjass.122588/
                d. jasscraft：
                    i. 源码（pascal语言，工作中没用过，但是还是比较好理解的）：https://github.com/Zoxc/JassCraft
                    ii. 下载：https://gamebanana.com/tools/2372
        iii. trigger/event/condition/action体系

1. 参考：https://blog.csdn.net/u013412391/article/details/109039410
2. trigger:整个一个包含event/condition/action的基于触发的一个逻辑闭环，称为一个触发器
3. event:事件，例如:角色升级、角色进入某区域、定时器时间到
4. condition:条件，例如：玩家等级是否达到5级及以上、玩家的hp是否>=50%
5. action:希望去做的事情，例如将角色升一级、将角色传送到指定坐标附近、扣除角色一定血量
6. 实际场景1：当英雄单位进入某区域时，如果英雄等级>=5级，则传送英雄至指定位置
    a. event:任意单位进入某区域
    b. condition:
        i. 单位为英雄
        ii. 单位等级>=5级
    c. action:
        i. 传送该单位至指定位置(达到该区域禁止高于5级的英雄单位进入的效果)
    d. 以下是AI帮写的参考代码，仅供参考
3. common.j | blizzard.j | test1.j

```
// 假设已经定义了区域ID和目标位置的坐标
const int REGION_ID = 1; // 区域ID，请根据实际情况替换
const location TARGET_POSITION = Location(0,0,0); // 目标位置坐标，请根据实际情况替换

// 自定义函数来检查单位是否为英雄且等级是否足够
function CheckHeroLevelAndTeleport takes nothing returns nothing
    local unit EnteringUnit;
    local integer HeroLevel;

    // 获取进入区域的单位
    EnteringUnit = GetTriggerUnit();

    // 检查单位是否为英雄
    if (IsUnitType(EnteringUnit, UNIT_TYPE_HERO)) then
        // 获取英雄等级
        HeroLevel = GetUnitLevel(EnteringUnit);

        // 判断英雄等级是否大于等于5
        if (HeroLevel >= 5) then
            // 传送英雄到目标位置
            IssueOrderById(EnteringUnit, ORDER_TELEPORT_TO_LOCATION, TARGET_POSITION.x, TARGET_POSITION.y, TARGET_POSITION.z);
        endif;
    endif;
endfunction;

// 在触发器的动作部分调用上述函数
RegisterTriggerEvent(GetTriggeringTrigger(), EVENT_UNIT_ENTERED_REGION, REGION_ID);
SetTriggerAction(GetTriggeringTrigger(), CheckHeroLevelAndTeleport);
```

```C++
// 假设已经定义了区域ID和目标位置的坐标
const int REGION_ID = 1; // 区域ID，请根据实际情况替换
const location TARGET_POSITION = Location(0,0,0); // 目标位置坐标，请根据实际情况替换

// 自定义函数来检查单位是否为英雄且等级是否足够
function CheckHeroLevelAndTeleport takes nothing returns nothing
    local unit EnteringUnit;
    local integer HeroLevel;

    // 获取进入区域的单位
    EnteringUnit = GetTriggerUnit();

    // 检查单位是否为英雄
    if (IsUnitType(EnteringUnit, UNIT_TYPE_HERO)) then
        // 获取英雄等级
        HeroLevel = GetUnitLevel(EnteringUnit);

        // 判断英雄等级是否大于等于5
        if (HeroLevel >= 5) then
            // 传送英雄到目标位置
            IssueOrderById(EnteringUnit, ORDER_TELEPORT_TO_LOCATION, TARGET_POSITION.x, TARGET_POSITION.y, TARGET_POSITION.z);
        endif;
    endif;
endfunction;

// 在触发器的动作部分调用上述函数
RegisterTriggerEvent(GetTriggeringTrigger(), EVENT_UNIT_ENTERED_REGION, REGION_ID);
SetTriggerAction(GetTriggeringTrigger(), CheckHeroLevelAndTeleport);
```

1. 实际场景2：做一个火焰之地(一定区域)，所有进入非机械单位每秒受到100点火焰(魔法)伤害
    a. event:加个定时器，每1秒触发一次
    b. condition:
        i. 在火焰之地区域内
        ii. 非机械单位
    c. action:
        i. 造成100点魔法伤害(需要算魔抗所以最终扣除hp需根据单位属性计算)
    d. 以下为AI的参考实现（这里使用了不止一个触发器，注释较多不赘述）：

4.

common.j | blizzard.j | test2.j

```
// 假设你已经定义了一个区域（Region）名为"FlameArea"
region FlameArea = CreateRegion(...); // 使用实际坐标和尺寸初始化区域

// 全局变量用于存储受伤害的单位
set g_HeroesInFlameArea = CreateGroup();


// 触发器1：单位进入火焰区域
TriggerEnterFlameArea = CreateTrigger();
TriggerAddCondition(TriggerEnterFlameArea, UnitEntersRegion(FlameArea));
TriggerAddAction(TriggerEnterFlameArea, function AddUnitToDamageList takes nothing returns nothing {
    local unit EnteringUnit = GetTriggerUnit();
    if (not IsUnitType(EnteringUnit, UNIT_TYPE_MECHANICAL)) then
        GroupAddUnit(g_HeroesInFlameArea, EnteringUnit, false);
        // 可以在这里开始对单位应用伤害
    }
});

// 触发器2：单位离开火焰区域
TriggerLeaveFlameArea = CreateTrigger();
TriggerAddCondition(TriggerLeaveFlameArea, UnitLeavesRegion(FlameArea));
TriggerAddAction(TriggerLeaveFlameArea, function RemoveUnitFromDamageList takes nothing returns nothing {
    local unit LeavingUnit = GetTriggerUnit();
    GroupRemoveUnit(g_HeroesInFlameArea, LeavingUnit);
});


// 创建一个定时器每秒检查并施加伤害
TimerFlameDamage = CreateTimer();
TimerStart(TimerFlameDamage, 1.00, false, function DealFlameDamage takes nothing returns nothing {
    foreach g_HeroesInFlameArea as unit Target do
        if (IsUnitAlive(Target)) then
            // 施加100点魔法伤害
            DamageTarget(Target, NULL, 100, true, WAR魔法师, DAMAGE_TYPE_MAGICAL);
        endif;
    endloop;
});
```

```C++
// 假设你已经定义了一个区域（Region）名为"FlameArea"
region FlameArea = CreateRegion(...); // 使用实际坐标和尺寸初始化区域

// 全局变量用于存储受伤害的单位
set g_HeroesInFlameArea = CreateGroup();


// 触发器1：单位进入火焰区域
TriggerEnterFlameArea = CreateTrigger();
TriggerAddCondition(TriggerEnterFlameArea, UnitEntersRegion(FlameArea));
TriggerAddAction(TriggerEnterFlameArea, function AddUnitToDamageList takes nothing returns nothing {
    local unit EnteringUnit = GetTriggerUnit();
    if (not IsUnitType(EnteringUnit, UNIT_TYPE_MECHANICAL)) then
        GroupAddUnit(g_HeroesInFlameArea, EnteringUnit, false);
        // 可以在这里开始对单位应用伤害
    }
});

// 触发器2：单位离开火焰区域
TriggerLeaveFlameArea = CreateTrigger();
TriggerAddCondition(TriggerLeaveFlameArea, UnitLeavesRegion(FlameArea));
TriggerAddAction(TriggerLeaveFlameArea, function RemoveUnitFromDamageList takes nothing returns nothing {
    local unit LeavingUnit = GetTriggerUnit();
    GroupRemoveUnit(g_HeroesInFlameArea, LeavingUnit);
});


// 创建一个定时器每秒检查并施加伤害
TimerFlameDamage = CreateTimer();
TimerStart(TimerFlameDamage, 1.00, false, function DealFlameDamage takes nothing returns nothing {
    foreach g_HeroesInFlameArea as unit Target do
        if (IsUnitAlive(Target)) then
            // 施加100点魔法伤害
            DamageTarget(Target, NULL, 100, true, WAR魔法师, DAMAGE_TYPE_MAGICAL);
        endif;
    endloop;
});
```

i. 网络游戏同步机制

1. 状态同步----服务器计算，同步计算结果，作弊空间相对较小

2. 帧同步----电影级效果，服务器只同步玩家的操作，各客户端基于游戏开始时下发的随机种子，相同的计算顺序，相同的伪随机算法，分别计算，网络带宽占用少，maphacker类的技术层面一定可作弊（数据都在内存里，变成了道高一尺魔高一丈的猫鼠游戏）

ii. 极致的优化

1. 技能朝向：地图编辑器中，将冲击波技能魔耗改为1或者0，CD改为0，牛头去升几级加个冲击波，360度旋转放冲击波，就能发现，在一定角度，会跳到下一角度，并不是360度，底层是用了一个byte甚至更低bit位来表示技能的朝向，才会有这样的效果
2. 阴影懒更新：如果某玩家一直没察看某一块地图的信息，即使友方有视野，也不会更新本地的显示性内存信息，如果拖过去，会发现那些树一瞬间消失了。或者技能移除的树，一瞬间恢复了。
3. 限帧策略：随便什么RPG地图，如果暴一地兵，会卡起来，当单位数量足够多的时候，可以看到，兵是一批一批动的。根据效果反推----实现层面是把所有参战单位放在一个列表中，保存一个游标记录上次更新到的单位下标，并限制每帧能计算的事件的单位数量

iii. 地图编辑器占掉研发的整体时间上的绝大部分，很多图都是高玩、爱好者做出来的

iv. 技术体系----核心c/c++自研脚本语言jass（游戏逻辑主要都在jass中，文件行数经常过万）

a. BigWorld引擎

i. 比传统游戏进程多了一层抽象维度的entity体系

1. base----与坐标不相关的----像玩家的背包中的东西、等级名称等信息，基本在base
2. cell----与坐标相关的----实时坐标、所在地图、技能召唤物等
3. client----客户端

ii. 游戏地图与进程的关系

1. 多个进程，每个进程一张地图
2. 每个进程一张地图可配置多条分线
3. 每个进程可配置一定地图一定分线
4. 每个进程组一个世界所有地图的一个分线
5. 无缝地图----抽象维度比之前的都至少高出一个维度

iii. 极为完善的文档体系和后端理念

1. https://github.com/v2v3v4/BigWorld-Engine-14.4.1
2. https://github.com/v2v3v4/BigWorld-Engine-2.0.1

3. 理念：

> **Client/server bandwidth is valuable**
>
> The most important resource is the bandwidth between the client and server. After this, it is probably CPU, and then intra-server bandwidth.

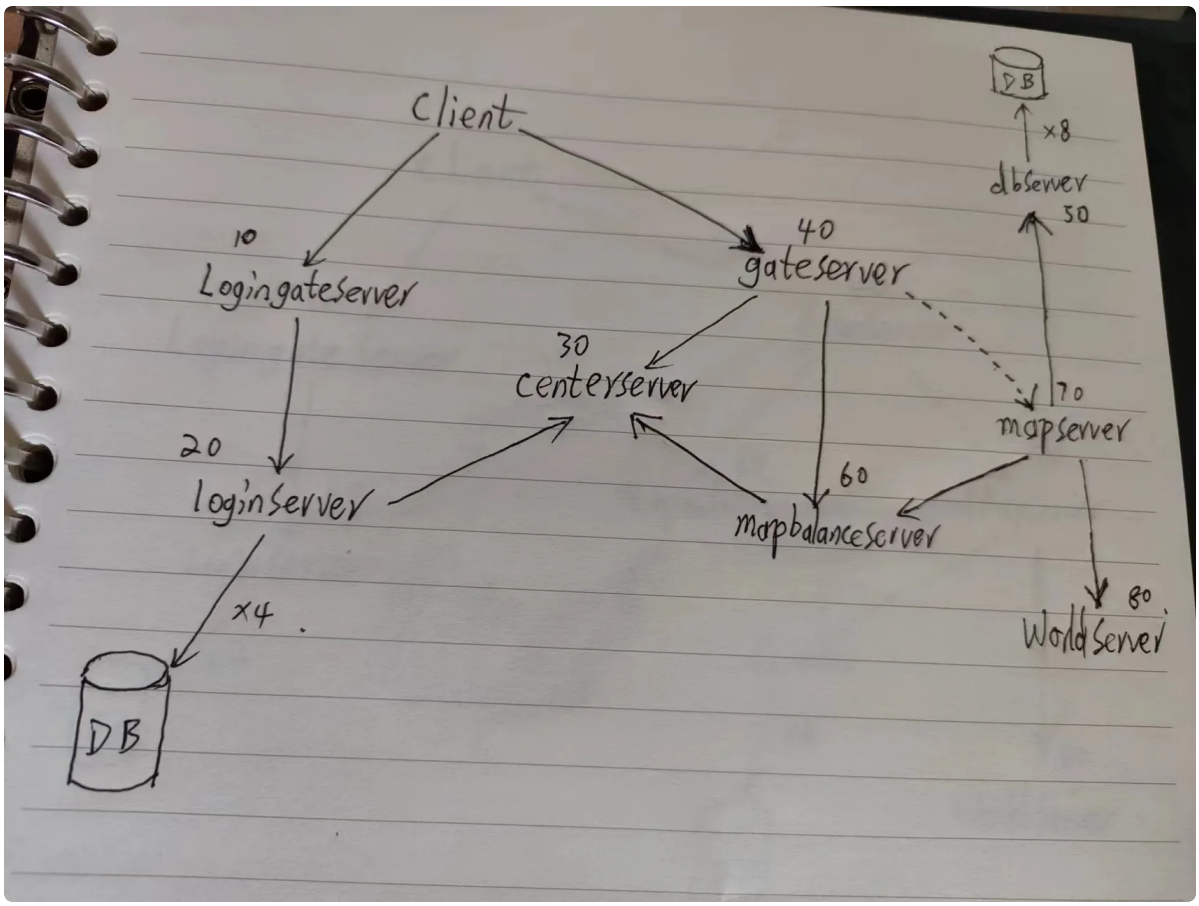
iv. 游戏产品：

1. 坦克世界
2. 网易天下3

v. 技术体系----核心框架c++，业务逻辑python(80%~90%)

b. 盛大的组件式多进程框架

i. 进程级网络拓朴图

ii. 

iii. 一堆纯虚的接口----定义接口，基于接口编程

1. 好处：
   a. 底层可以做到极致的性能优化、完善、压测，不影响上层业务逻辑
   b. 底层出现新技术时，更新底层技术体系，不会影响上层的已有业务逻辑，一行都不需要动
   c. 随着时间的推移，项目的迭代，底层会越来越趋向于稳定
2. 坏处：

a. 技术中心的技术容易与项目脱钩，项目中绝大多数技术实力没有到专精的水平的会对技术中心的技术有抵触情绪
b. 新人如果不花时间精力去研究这套底层框架，技术上没有突破
c. 底层出现大的根本性突破这种业务模型的新技术时，框架的更新会变的很被动，工作量大，全员学习成本高----所有使用原框架的必须学习新框架的思路

iv. 技术体系----最初核心框架c/c++，业务逻辑c++(90%)/lua(10%)。后来提升了lua的逻辑占比到50%~70%左右

盛大的纯虚接口 C++

```cpp
namespace SGDP{
    class ISDPacketParser;
    class ISDSession;
    class ISDSessionFactory;

    class ISDConnection {
    public:
    virtual ~ISDConnection() {}
    virtual bool SDAPI IsConnected(void) = 0;
    virtual void SDAPI Send(const char* pBuf,UINT32 dwLen) = 0;
    virtual void SDAPI SetOpt(UINT32 dwType, void* pOpt) = 0;
    virtual void SDAPI Disconnect(void) = 0;
    virtual const UINT32 SDAPI GetRemoteIP(void) = 0;
    virtual const char* SDAPI GetRemoteIPStr(void) = 0;
    virtual UINT16 SDAPI GetRemotePort(void) = 0;
    virtual const UINT32 SDAPI GetLocalIP(void) = 0;
    virtual const char* SDAPI GetLocalIPStr(void) = 0;
    virtual UINT16 SDAPI GetLocalPort(void) = 0;
    virtual UINT32 SDAPI GetSendBufFree(void) = 0;
};

    class ISDListener {
    public:
    virtual ~ISDListener() {}
    virtual void SDAPI SetPacketParser(ISDPacketParser* poPacketParser) =
0;
    virtual void SDAPI SetSessionFactory(ISDSessionFactory* poSessionFacto
ry) = 0;
    virtual void SDAPI SetBufferSize(UINT32 dwRecvBufSize, UINT32 dwSendBu
fSize) = 0;
    virtual void SDAPI SetOpt(UINT32 dwType, void* pOpt) = 0;
    virtual bool SDAPI Start(const char* pszIP, UINT16 wPort, bool bReUseA
ddr = true) = 0;
    virtual bool SDAPI Stop(void) = 0;
    virtual void SDAPI Release(void) = 0;
};

    class ISDConnector {
    public:
    virtual ~ISDConnector() {}
    virtual void SDAPI SetPacketParser(ISDPacketParser* poPakcetParser) =
0;
    virtual void SDAPI SetSession(ISDSession* poSession) = 0;
    virtual void SDAPI SetBufferSize(UINT32 dwRecvBufSize, UINT32 dwSendBu
fSize) = 0;
```

```
    virtual bool SDAPI Connect(const char* pszIP, UINT16 wPort) = 0;
    virtual bool SDAPI ReConnect(void) = 0;
    virtual void SDAPI Release(void) = 0;
    virtual void SDAPI SetOpt(UINT32 dwType, void* pOpt) = 0;
};

    class ISDNet : public ISDBase {
    public:
    virtual ~ISDNet() {};
    virtual ISDConnector* SDAPI CreateConnector(UINT32 dwNetIOType) = 0;
    virtual ISDListener* SDAPI CreateListener(UINT32 dwNetIOType)  = 0;
    virtual bool SDAPI Run(INT32 nCount = -1) = 0;
};

    class ISDPacketParser {
    public:
    virtual ~ISDPacketParser() {}
    virtual INT32 SDAPI ParsePacket(const char* pBuf, UINT32 dwLen) = 0;
};

    class ISDSession {
    public:
    virtual ~ISDSession() {}
    virtual void SDAPI SetConnection(ISDConnection* poConnection) = 0;
    virtual void SDAPI OnEstablish(void) = 0;
    virtual void SDAPI OnTerminate(void) = 0;
    virtual bool SDAPI OnError(INT32 nModuleErr, INT32 nSysErr) = 0;
    virtual void SDAPI OnRecv(const char* pBuf, UINT32 dwLen) = 0;
    virtual void SDAPI Release(void) = 0;
};

    class ISDSessionFactory {
    public:
    virtual ~ISDSessionFactory() {}
    virtual ISDSession* SDAPI CreateSession(ISDConnection* poConnection)
= 0;
};
}
```

c. 基于协程的无状态平行扩展框架

i. 上述组件式多进程纯异步回调式带来的问题与思考：

1. 问题：回调地狱，业务逻辑被强制分拆成多段。例如一个消息的执行流程需访问三次数据库，则需要写四个回调。一整段连续的逻辑被拆成了四段：

a. 第一个是收到消息的回调

b. 第二个是第一次数据库操作的回调

c. 第三个是第二次数据库操作的回调

d. 第四个是第三次数据库操作的回调

```C++
int Player::msgproc_SomeMsg(const SomeMsgReq* req) {
    // ...
    DBGetOrCheckSomeData1(req->SomeData(), OnFinishDBGetOrCheckSomeData1);
    return 0;
}

int Player::OnFinishDBGetOrCheckSomeData1(DBResult* res) {
    // ...
    DBGetOrCheckSomeData2(req->SomeData(), OnFinishDBGetOrCheckSomeData2);
    return 0;
}

int Player::OnFinishDBGetOrCheckSomeData2(DBResult* res) {
    // ...
    DBGetOrCheckSomeData3(req->SomeData(), OnFinishDBGetOrCheckSomeData3);
    return 0;
}

int Player::OnFinishDBGetOrCheckSomeData2(DBResult* res) {
    // ...
    Send2Client(res->SomeData());
    return 0;
}
```

2. 问题的本质：逻辑单线程，不能等异步的回调。

    a. 算笔账：如果有1000个玩家，每个玩家每秒产生一个消息，每个消息因访问内网redis，需要消耗1ms时间，那么如果单线程的逻辑1000个玩家同步等redis的操作返回，光这一步就已经耗光了全部时长，如果再来500个玩家，消息就会越堆越多。----线上的症状：CPU很低，内存使用量很低，操作响应极慢在8秒甚至10秒以上甚至无响应，消息队列如果写的比较好的超掉或者超量有错误日志可查到。其实就是单线程在等的时间加起来已经堆满了整个单线程的时间轴。

3. 两种突破：

    a. 有栈协程

        i. 最早apache的思路是每一个用户开一个线程，所以负载能力非常有限，几百个人就负载上限了。因为进程开的线程数量是非常有限的。有栈协程应运而

生。go就是使用的有栈协程。c++当时google的人提的想进标准的也是有栈协程，最终微软的人提的无栈协程胜出了，进了标准，但是写业务不适合，适合写底层框架。

ii. 全局/每个用户/每个逻辑组/每个消息一个协程，协程之间通讯走channel或走消息队列。

1. 全局：A是否能改B的数据---可以 A是否能改A的数据---可以
2. 每个用户：A是否能改B的数据---不可以 A是否能改A的数据---可以
3. 每个逻辑组：A是否能改B的数据---不一定，同逻辑组可改 A是否能改A的数据---可以
4. 每个消息一个协程：A是否能改B的数据---不可以 A是否能改A的数据---不可以

iii. 系统可以很轻松的开几十万协程

iv. 协程间数据需加锁或做一致性

v. 对从线程编程里成长起来的程序猿们来说，理解成本较低

b. 无栈协程

i. 本质上是把cpu当做了一个救火队员，把所有的顺序的业务逻辑拆分成了独立的任务任务列表让CPU去调度，跑到一个里面去执行一定的逻辑，然后如果发生异步，立刻跳走去其他地方执行。

ii. 系统可以很轻松开几十万协程

iii. 协程间数据无需加锁，数据大量使用sharedptr之类，nodejs体系也是无栈协程，自己保障了数据的安全性。

iv. 对从线程编程里成长起来的程序猿来说，理解成本相对较高

ii. 大量数据存在redis中，go为主，每个进程无状态，或半无状态，状态都在redis中（之前的思路在单节点中）。

iii. 进程分为两种

1. 彻底无状态，任何节点均可处理
2. 半无状态。每次需要被路由到同一个节点中，状态在redis中的半无状态。这样做的好处是进程路由到同一个节点，可以做内存缓存

iv. 技术体系----纯go